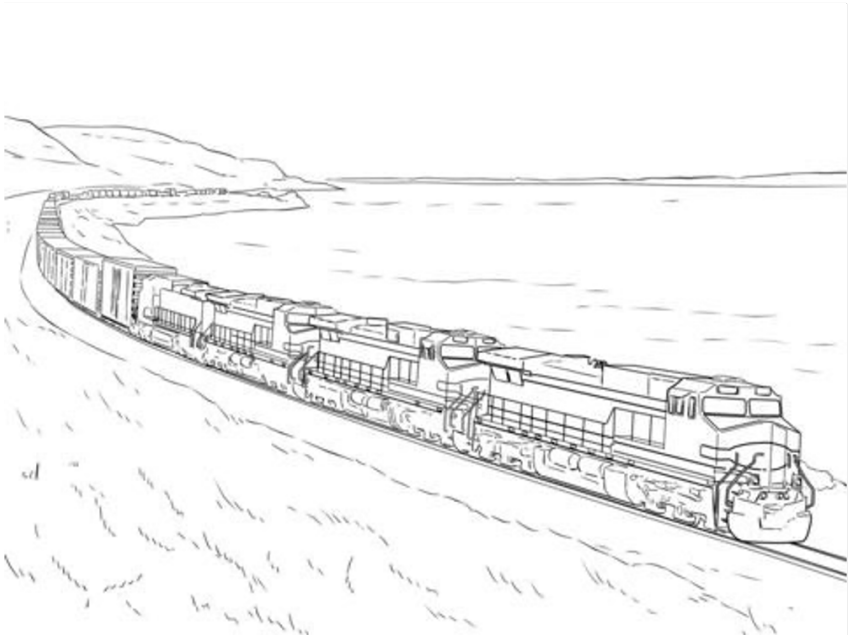


A Layout Control System for Model Railroads

Helmut Fieres December 2, 2024

Contents

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 controlling 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 an 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. Further-

more, 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. Am 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 understanding 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 lay out 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 The 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 message 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 break-out 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.

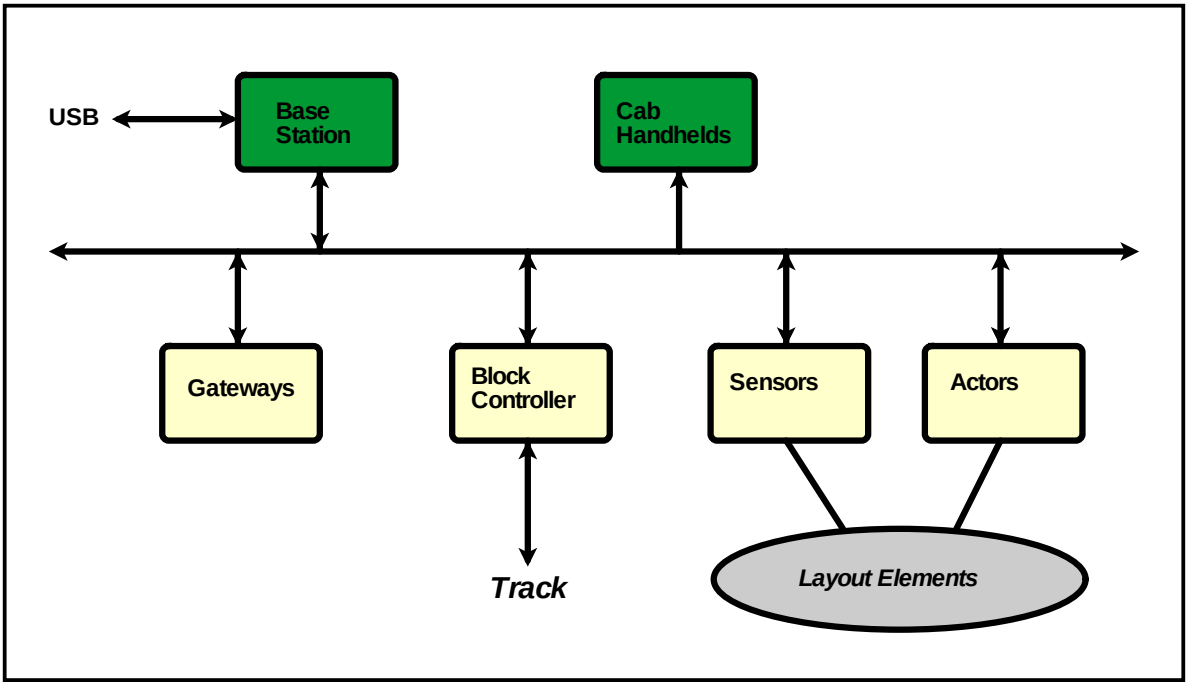
I have added most of the source files in the appendix for direct reference. They can also be found also on GitHub. (Note: still to do...) Every building block schematic shown was used and tested in one component or another. However, sometimes the book may not exactly match the material found on the web 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 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. The 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

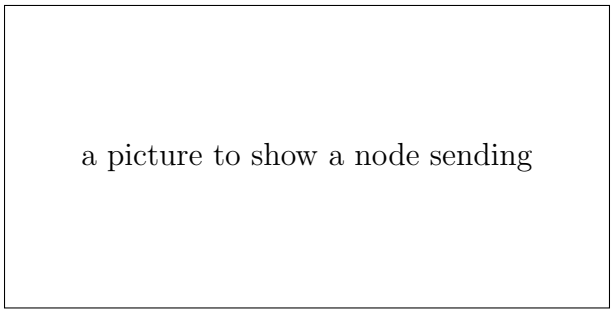
to node and ports. Node, port and attribute data are backed by non-volatile memory, so that a restart will use defined initial values. Nodes and their ports are also directly addressable, which is needed for configuration purposes and the directly addressable components model. Using the producer / consumer 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)` 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					
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.

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 3.3: 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

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 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.

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.

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.

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						

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			

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.

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 3.13: RRaw DCC Packet Management

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

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.

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.

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)

Table 4.7: Node and Port Access

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

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 4.8: 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 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.

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

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

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.

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

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.

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.

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

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.

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.

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

Table 4.14: Locomotive Session Management

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

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 trough 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.

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

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

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 relief 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.

picture

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 form 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 date 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.

picture: the high level structure of the node map...

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.

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.

Table 6.1: 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	191	Node or Port Attributes first copied from NVM to MEM and then returned
192	255	Node or Port Attributes first copied from NVM to MEM and then returned

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 though 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 programer 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)`.

```
// ... the busMgt msg handler routine
void busMgtMsgHandler( uint8_t *msgBuf ) {
        //... handle the cases of busMgt messages
}
...
// during module firmware initialization ...
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.

```
// ... the node init msg handler routine
void nodeInitHandler( uint16_t nodeId ) { ... }
...
```

```
// during module firmware initialization ...
lcsLib -> registerInitCallback( NIL_PORT_ID, nodeInitHandler )
```

Note that a portID or `NIL_PORT_ID` will refer to the node. Registering an initialization callback fro 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.

```
// ... the node query handler routine
void nodeReqHandler(    uint16_t nodeId,
                        uint8_t portId,
                        uint8_t item,
                        uint16_t val1,
                        uint16_t val2 ) { ... }
...
// during module firmware initialization ...
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.

```
uint8_t ( *infoHandler ) ( uint8_t portId,
                           uint8_t item,
                           uint16_t *arg1,
                           uint16_t *arg2 ) { ... }

uint8_t ( *ctrlHandler ) ( uint8_t portId,
                           uint8_t item,
                           uint16_t arg1,
                           uint16_t arg2 ) { ... }
...
// during module firmware initialization ...
lcsLib -> registerInfoCallback( portId, infoHandler );
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.

```
// ... the inbound event handler routine
void eventHandler ( uint16_t nodeId,
                    uint8_t portId,
                    uint8_t eAction,
                    uint16_t eId,
                    uint16_t eData ) { ... }
...
// during module firmware initialization ...
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.

```
// ... the command line handler routine
uint8_t commandLineHandler( char *line ) { ... }
...
// during module firmware initialization ...
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.

```
// ... a periodic task to be registered
void aTask( ) { ... }
...
// during module firmware initialization ...
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. Other than functions and callbacks, how can you access the library ? Wouldn't is 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.

```
<!?>
<#?>
```

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 "¡" and "¿" 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 ?

11 The DCC Subsystem

The LCS core 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 shows a simple packet.

picture

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 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.

picture

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 tarts 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 of 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 form different encoders are transmitted one at a time and there is enough tie for the addressed decoder to answer. Als, 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.

picture

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, the 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 tack 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 a 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 tack and also used for the programming track decoder acknowledge power consumption fluctuation. Exceeding the configured power consumption limits will result an 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 is 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.

picture

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 thongs 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. I 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 block**s 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. Definitively 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 more 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.

Continued on next page

Attributes	Notes
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.
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 as the "Euro PCB" format. The main controller board has on the left side the connectors for the LCS bus and the power input. On the left side, there are two connectors toward an extension board. The middle one is the extension connector described earlier, the lower right side is the power lines routed through from the power in connector.

Extension boards have three connectors pairs, in and out. The lower pair just routes power through to the next extension board. The middle connector pair will route a subset of the extension connector signals from the main controller. What exactly is routed is described in the extension board chapters. Finally, the extension boards have an optional third line, which is the track power connectors. This line is used by the base station and block controller boards. Again, all this will be explained in the later chapters. To ease the hardware schematic development and ensure that all boards fit together, the PCB boards along with their connectors are available as 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	
5	RSV	
6	VS	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. Furthermore, an extension board should have two connectors so we can for example add two or more extension to the main controller board. This concept is very similar to the the shield concept found in the Arduino or Raspberry PI universe, except that we do not stack boards, we place them next to each other. Not all IO lines of a controller are exported to the extension board. For example, the SPI interface, configuration switches and status LEDs are local to the main controller board. The I2C interface will be the main communication method between the boards. Nevertheless, a rather rich functionality set 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. Many pins of a controller chip double up in function. All of these special purpose pins can also be used just as plain digital input/output pins. The following table shows the extension connector pin assignments.

??? do the double duty of digital ins still apply for the pico ?

Table 13.3: Controller Attributes

Pin	Name	Purpose
1	**DCC-SIG1**	The DCC "+" signal as generated by the DCC Signal Generator. On the main controller board, the DCC signals are routed from the extension connector directly to the LCS Bus connector.
2	**DCC-SIG2**	The DCC "-" signal as generated by the DCC Signal Generator. On the main controller board, the DCC signals are routed from the extension connector directly to the LCS Bus connector.
3	GND	Common ground.
4	GND	Common ground.
5	ADC-0	Analog input pin. The input is not protected. The analog voltage range is 0 to VCC.
6	ADC-1	Analog input pin. The input is not protected. The analog voltage range is 0 to VCC.
7	-	reserved.
8	RST	Reset line available to the extension boards. A reset line is active low.
9	DIO-0 (RX-1)	Digital pin, UART RX capable. The pin is protected.
10	DIO-1 (TX-1)	Digital pin, UART TX capable. The pin is protected.
11	DIO-2	Plain digital Pin, input or output. The pin is protected. The pin DIO-2 and DIO3 are on the same controller IO port and can be set simultaneously.
12	DIO-3	Plain digital Pin, input or output. The pin is protected. The pin DIO-2 and DIO-3 are on the same controller IO port and can be set simultaneously.
13	DIO-4	Plain digital Pin, input or output. The pin is protected. The pin DIO-4 and DIO5 are on the same controller IO port and can be set simultaneously.
14	DIO-5	Plain digital Pin, input or output. The pin is protected. The pin DIO-4 and DIO-5 are on the same controller IO port and can be set simultaneously.
15	DIO-6 (PWM-1)	Plain digital Pin, PWM capable. The pin is protected. The pin DIO-6 and DIO-7 are on the same controller IO port and can be set simultaneously.

Continued on next page

Pin	Name	Purpose
16	DIO-7 (PWM-2)	Plain digital Pin, PWM capable. The pin is protected. The pin DIO-6 and DIO-7 are on the same controller IO port and can be set simultaneously.
17	VCC	VCC 5V supply to extension boards.
18	VCC	VCC 5V supply to extension boards.
19	SCL	I2C Interface SCL line. The line is protected with a serial resistor and there is a pull-up resistor to VCC.
20	SDA	I2C Interface SDA line. The line is protected with a serial resistor and there is a pull-up resistor to VCC.

Since the connector chosen is a 2x10 connector, the signal pin numbers shown above change their row numbering, depending whether it is output or input connector. When looking at the schematics and the connector layout on the PCB, the signals on the output connector have pin 1 to 10 on the leftmost row, and on the rightmost row on the input connector, such that pin 1 of the output connector connects to pin 1 of the input connector and so on. The same is true for pins 11 to 20.

There are EasyEDA symbols that offer the connector pins with you going through these details. The appendix contains EasyEDA symbols for the most common board dimensions with the connectors placed in the right location. A new projects can just start with this EasyEDA symbol. There is also the option of cascading extensions. An extension connector could therefore be on both ends of the board and pass the GND, VCC, Reset, DCC and I2C lines from board to board. For this type of board, EasyEDA symbols are also available.

A key question is how many controller pins are available to an extension board. Most of the extension boards would just need the I2C bus. However, if there is a rather complex extension board, such as a block controller shown in one of the next chapters, the IO pins needed from the controller board to the extension are many and quickly reach the limit of the extension connector. Why not place a connector with more pins on the boards ? First, a different controller may not have that many IO pins and there would be no easy mix and match between main and extension boards. Second, the majority of extension boards are rather encapsulated and most often just need the I2C bus to communicate. To find a middle ground, the 20-pin connector along with the pin capabilities outlined was chosen.

For more complex extension boards, it is perhaps the better idea to combine a main board with an extension board to one monolithic board and still keep the extension connector for other not so complex boards to attach. As a convention, 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 reset line, the I2C signal and the power lines.

13.7 Power Line Connectors

The **power line connector** forward the power line input of the main controller board to an extension board. This connector is primarily needed for power unit extension to power the H-Bridges on such a board. Extensions that do not require this power input forwarding just leave it out. The first pair of connector pins is always connected, all the others are optional. Boards with a high current consumption could pool more than one connector pin pair.

Table 13.4: Power Line Connectors

Pin	Name	Purpose
1	GND	Common ground.
2	GND	Common ground.
3	GND	Common ground.
4	GND	Common ground.
5	VS	Input voltage forward, always connected.
6	VS	Input voltage forward, always connected.
7	VS	Input voltage forward, always connected.
8	VS	Input voltage forward, always connected.

13.8 Track Power Connectors

In addition to the extension board and power line 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 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 13.5: Power Line Connectors

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

When using all four bridge signal outputs, each each output 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.

13.9 Summary

This chapter introduced the basic architecture of a hardware modules, it connectors and board layout. A key concept is the idea of a common component, the main controller, and extension 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. Throughout the chapter to come, you will see how easy boards can be combined using the three connectors lanes and standards behind them. Currently, the boards are designed in a layout, where they just connect next to each other. Conceptually, they could also be stacked. It is a matter of PCS layout design.

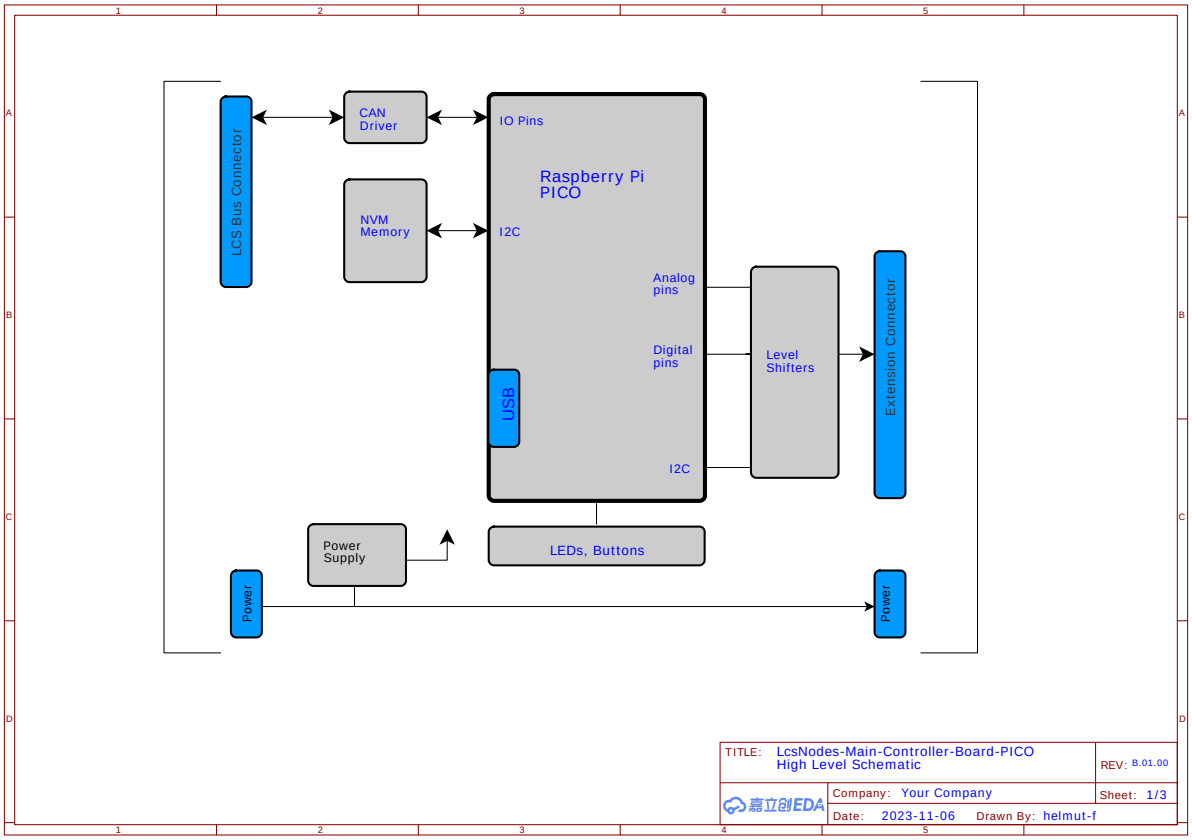
Ready for the first hardware work ? All aboard, the train leaves for the next chapter.

A Tests

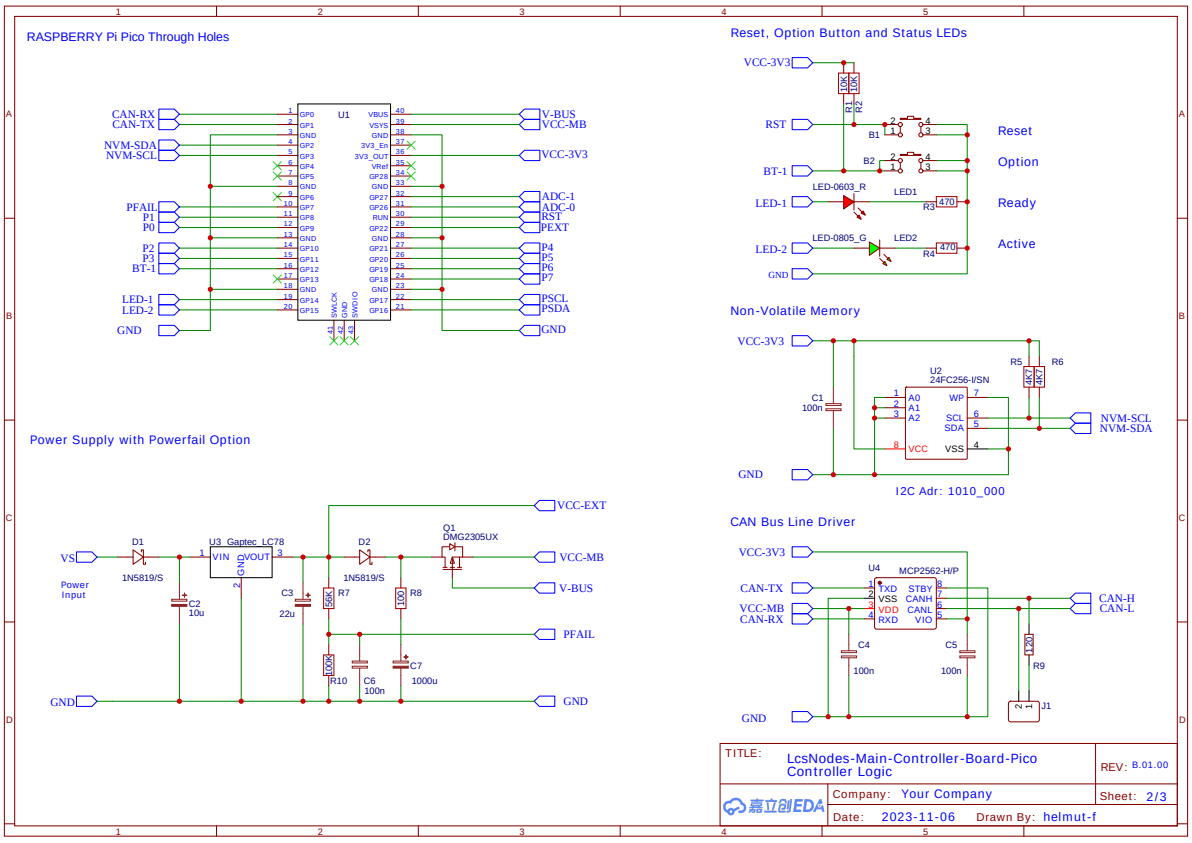
A.1 Schematics

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

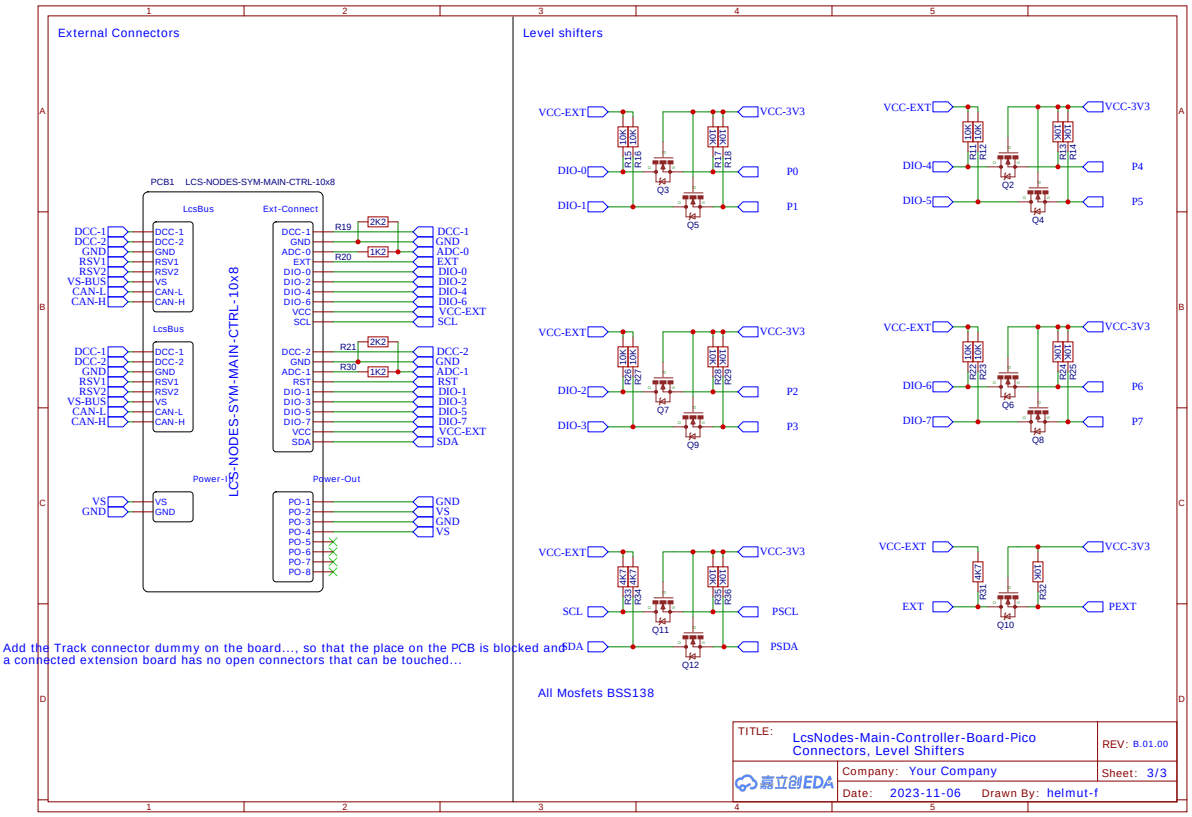
A.1.1 part 1



A.1.2 part 2


RASPBERRY Pi Pico Through Holes
CAN-RX
CAN-TX
NVM-SDA
NVM-SCL
PFAIL
P1
P0
P2
P3
BT-1
LED-1
LED-2
GND
U1
V-BUS
VCC-MB
VCC-3V3
ADC-1
ADC-0
RST
PEXT
P4
P5
P6
P7
PSCL
PSDA
Reset, Option Button and Status LEDs
VCC-3V3
RST
BT-1
LED-1
LED-2
LED-0603_R
LED1
LED-0805_G
LED2
Reset
Option
Ready
Active
Non-Volatile Memory
U2
24FC256-I/SN
C1
100n
I2C Adr: 1010_000
Power Supply with Powerfail Option
VS
Power Input
D1
1N5819/S
U3_Gaptec_LC78
VIN
VOUT
GND
C2
10u
C3
22u
D2
Q1
DMG2305UX
VCC-EXT
VCC-MB
V-BUS
PFAIL
R7
56K
R8
100
R10
100K
C6
100n
C7
1000u
CAN Bus Line Driver
U4
MCP2562-H/P
TXD
VSS
VDD
RXD
STBY
CANH
CANL
VIO
C4
C5
CAN-H
CAN-L
R9
120
J1
TITLE:
LcsNodes-Main-Controller-Board-Pico Controller Logic
REV: B.01.00
Company: Your Company
Sheet: 2/3
Date: 2023-11-06
Drawn By: helmut-f

A.1.3 part 3



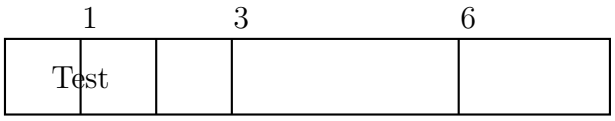
A.2 Lists

A.2.1 A simple list

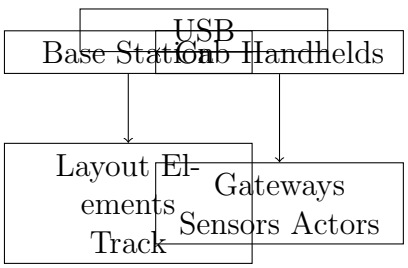
- First bullet point
- Second bullet point
- Third bullet point

A.2.2 An instruction word layout

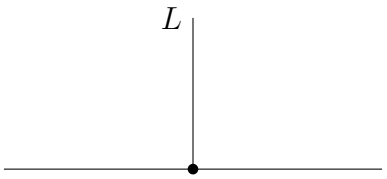
A little test for an instruction word layout ... will be a bit fiddling work ...



First attempt ...



Hugo
Berta
Carla



B Listings test

B.1 Base Station

```
//--------------------------------------------------------------------------------------------------------------
//
// LCS Base Station - Include file
//
//--------------------------------------------------------------------------------------------------------------
//
// LCS - Base Station
// Copyright (C) 2019 - 2024  Helmut Fieres
//
// This program is free software: you can redistribute it and/or modify it under the terms of the GNU General
// Public License as published by the Free Software Foundation, either version 3 of the License, or (at your
// option) any later version.
//
// This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
// implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
// for more details.
//
// You should have received a copy of the GNU General Public License along with this program. If not, see
// http://www.gnu.org/licenses
//
//  GNU General Public License:  http://opensource.org/licenses/GPL-3.0
//
//--------------------------------------------------------------------------------------------------------------
#ifndef LcsBaseStation_h
#define LcsBaseStation_h

#include "LcsCdcLib.h"
#include "LcsRuntimeLib.h"

//--------------------------------------------------------------------------------------------------------------
// The base station maintains a set of debug flags. The overall concept is very similar to the LCS runtime
// library debug mask. Then following debug flags are defined:
//
//      DBG_BS_CONFIG                   -   DEBUG base station enabled
//      DBG_BS_SESSION                  -   show the session management actions
//      DBG_BS_LCS_MSG_INTERFACE        -   show the incoming LCS messages
//      DBG_BS_TRACK_POWER_MGMT         -   show the track power measurement data
//      DBG_BS_DCC_ACK_DETECT           -   display decoder ACK power measurements
//      DBG_BS_CHECK_ALIVE_SESSIONS     -   displays that a session seems no longer be alive
//      DBG_BS_RAILCOM                  -   show the RailCom activity
//
// The way to use these flags is for example:
//
//      if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_SESSION ))
//
// ??? should have a command to set the debug mask on the fly...
//--------------------------------------------------------------------------------------------------------------
enum BaseStationDebugFlags : uint16_t {

    DBG_BS_CONFIG                   = 1 << 15,       // DEBUG base station enabled

    DBG_BS_SESSION                  = 1 << 0,        // show the session management actions
    DBG_BS_LCS_MSG_INTERFACE        = 1 << 1,        // show the incoming LCS messages
    DBG_BS_TRACK_POWER_MGMT         = 1 << 2,        // show the track power measurement data
    DBG_BS_DCC_ACK_DETECT           = 1 << 3,        // display decoder ACK power measurements
    DBG_BS_CHECK_ALIVE_SESSIONS     = 1 << 4,        // displays that a session seems no longer be alive
    DBG_BS_RAILCOM                  = 1 << 5         // show the RailCom activity

};

//--------------------------------------------------------------------------------------------------------------
// Base station errors. Note that they need to be in the assigned to the user number range of errors defined
// in the LCS runtime library.
//
//--------------------------------------------------------------------------------------------------------------
enum BaseStationErrors : uint8_t {

    BASE_STATION_ERR_BASE             = 128,

    ERR_NO_SVC_MODE                   = BASE_STATION_ERR_BASE + 1,
    ERR_CV_OP_FAILED                  = BASE_STATION_ERR_BASE + 2,

    ERR_LOCO_NOT_FOUND                = BASE_STATION_ERR_BASE + 4,
    ERR_SESSION_NOT_FOUND             = BASE_STATION_ERR_BASE + 5,
    ERR_LOCO_SESSION_ALLOCATE         = BASE_STATION_ERR_BASE + 6,
    ERR_LOCO_SESSION_CANCELLED        = BASE_STATION_ERR_BASE + 7,

    ERR_SESSION_SETUP                 = BASE_STATION_ERR_BASE + 9,
    ERR_MSG_INTERFACE_SETUP           = BASE_STATION_ERR_BASE + 10,
    ERR_DCC_TRACK_CONFIG              = BASE_STATION_ERR_BASE + 11,
    ERR_DCC_PIN_CONFIG                = BASE_STATION_ERR_BASE + 12,

```

```
    ERR_NVM_HW_SETUP                = BASE_STATION_ERR_BASE + 15,
    ERR_PIO_HW_SETUP                = BASE_STATION_ERR_BASE + 16
};

//-----------------------------------------------------------------------------------------------------------
// DCC packet definition. A DCC packet is the payload data without the checksum. Besides the length in bytes
// and the buffer, there is a repeat counter to specify how often this packet will be repeatedly transmitted
// after the first transmission. Currently, a DCC packet is at most 15 bytes long, excluding the checksum
// byte. This is true for XPOM and DCC-A support, otherwise it is historically a maximum of 6 bytes.
//
//-----------------------------------------------------------------------------------------------------------
const uint8_t DCC_PACKET_SIZE = 16;

struct DccPacket {

    uint8_t len;
    uint8_t repeat;
    uint8_t buf[ DCC_PACKET_SIZE ];
};

//-----------------------------------------------------------------------------------------------------------
// DCC packet payload data definitions we need often, so these constants come in handy.
//
//-----------------------------------------------------------------------------------------------------------
const uint8_t   idleDccPacketData[ ]    = { 0xFF, 0x00 };
const uint8_t   resetDccPacketData[ ]   = { 0x00, 0x00 };
const uint8_t   eStopDccPacketData[ ]   = { 0x00, 0x01 };

//-----------------------------------------------------------------------------------------------------------
// Setup options to set for the DCC track. They are set when the track object is created.
//
//  DT_OPT_SERVICE_MODE_TRACK  - The track is a PROG track.
//  DT_OPT_CUTOUT              - The track is configured to emit a cutout during the DCC packet preamble.
//  DT_OPT_RAILCOM             - The track support Railcom detection.
//
//-----------------------------------------------------------------------------------------------------------
enum DccTrackOptions : uint16_t {

    DT_OPT_DEFAULT_SETTING      = 0,
    DT_OPT_SERVICE_MODE_TRACK   = 1 << 0,
    DT_OPT_CUTOUT               = 1 << 1,
    DT_OPT_RAILCOM              = 1 << 2
};

//-----------------------------------------------------------------------------------------------------------
// The DCC track object has a set of flags to indicate its current status.
//
//  DT_F_POWER_ON             - The track is under power.
//  DT_F_POWER_OVERLOAD       - An overload situation was detected.
//  DT_F_MEASUREMENT_ON       - The power measurement is enabled.
//  DT_F_SERVICE_MODE_ON      - The track is currently in service mode, i.e. is a PROG track.
//  DT_F_CUTOUT_MODE_ON       - The track has the cutout generation enabled.
//  DT_F_RAILCOM_MODE_ON      - The track has the railcom detect enabled.
//  DT_F_RAILCOM_MSG_PENDING  - If railcom is enabled, a received datagram is indicated.
//  DT_F_CONFIG_ERROR         - The passed configuration descriptor has invalid options configured.
//
//-----------------------------------------------------------------------------------------------------------
enum DccTrackFlags : uint16_t {

    DT_F_DEFAULT_SETTING      = 0,
    DT_F_POWER_ON             = 1 << 0,
    DT_F_POWER_OVERLOAD       = 1 << 1,
    DT_F_MEASUREMENT_ON       = 1 << 2,
    DT_F_SERVICE_MODE_ON      = 1 << 3,
    DT_F_CUTOUT_MODE_ON       = 1 << 4,
    DT_F_RAILCOM_MODE_ON      = 1 << 5,
    DT_F_DCC_PACKET_PENDING   = 1 << 6,
    DT_F_RAILCOM_MSG_PENDING  = 1 << 7,
    DT_F_CONFIG_ERROR         = 1 << 15
};

//-----------------------------------------------------------------------------------------------------------
// The following constants are for the current consumption RMS measurement. The idea is to record the measured
// ADC values in a circular buffer, every time a certain amount of milliseconds has passed. This work is done
// by the DCC track state machine as part of the power on state.
//
//-----------------------------------------------------------------------------------------------------------
const uint8_t   PWR_SAMPLE_BUF_SIZE             = 64;
const uint32_t  PWR_SAMPLE_TIME_INTERVAL_MILLIS  = 16;

//-----------------------------------------------------------------------------------------------------------
// The RailCom buffer size. During the cutout period up to eight bytes of raw data are sent by the decoder if
// the Railcom option is enabled.
//
//-----------------------------------------------------------------------------------------------------------
const uint8_t   RAILCOM_BUF_SIZE = 8;

//-----------------------------------------------------------------------------------------------------------
// The session map options. These are options initially set when the base station starts. They are used to
// set the flags, which are then used for processing the the actual settings.
//
//  SM_KEEP_ALIVE_CHECKING  - enable keep alive checking. When enabled, the locomotive session need to receive
//                            a keep alive LCS message periodically.
//  SM_ENABLE_REFRESH       - refresh the session data. This will send the locomotive speed and direction as
//                            well as the function flags periodically in a round robin processing of the
//
//-----------------------------------------------------------------------------------------------------------
enum SessionMapOptions : uint16_t {

```

```
    SM_OPT_DEFAULT_SETTING        = 0,
    SM_OPT_KEEP_ALIVE_CHECKING    = 1 << 0,
    SM_OPT_ENABLE_REFRESH         = 1 << 1
};

//-----------------------------------------------------------------------------------------------------------
// The session map flags. The apply to all sessions in the session map. The initial values are copied from
// session option initial values.
//
//  SM_F_KEEP_ALIVE_CHECKING  - enable keep alive checking. When enabled, the locomotive session need to receive
//                              a keep alive LCS message periodically.
//  SM_F_ENABLE_REFRESH       - refresh the session data. This will send the locomotive speed and direction as
//                              well as the function flags periodically in a round robin processing of the
//
//-----------------------------------------------------------------------------------------------------------
enum SessionMapFlags : uint16_t {

    SM_F_DEFAULT_SETTING        = 0,
    SM_F_KEEP_ALIVE_CHECKING    = 1 << 0,
    SM_F_ENABLE_REFRESH         = 1 << 1
};

//-----------------------------------------------------------------------------------------------------------
// Each session map entry has a set of flags.
//
//  SME_ALLOCATED           - the session is allocated, the entry valid.
//  SME_COMBINED_REFRESH    - locomotive speed/dir and functions are refreshed using the combined DCC packet.
//  SME_SPDIR_REFRESH       - locomotive speed/dir are refreshed.
//  SME_FUNC_REFRESH        - locomotive functions are refreshed.
//  SME_DISPATCHED          -
//  SME_SHARED              -
//
//
// ??? when the base station has a config value of using the DCC spdir/func command, these flags need to be
// named slightly different. Should we still have the option to enable or disable it even though the base
// station can do it ? A decoder might not support this packet type...
//-----------------------------------------------------------------------------------------------------------
enum SessionMapEntryFlags : uint16_t {

    SME_DEFAULT_SETTING     = 0,
    SME_ALLOCATED           = 1 << 0,
    SME_COMBINED_REFRESH    = 1 << 1,
    SME_SPDIR_ONLY_REFRESH  = 1 << 2, // ??? phase out...
    SME_SPDIR_REFRESH       = 1 << 2,
    SME_FUNC_REFRESH        = 1 << 3,
    SME_DISPATCHED          = 1 << 4,
    SME_SHARED              = 1 << 5
};

//-----------------------------------------------------------------------------------------------------------
// The base station items for nodeInfo and nodeControl calls .... tbd
//
// ??? the are mapped in the MEM / NVM range as well as in the USER range.
// ??? how to do it consistently and understandably ?
//-----------------------------------------------------------------------------------------------------------
enum BaseStationItems : uint8_t {

    // or use GET in all constants

    BS_ITEM_SESSION_MAP_OPTIONS   = 128,
    BS_ITEM_SESSION_MAP_FLAGS     = 129,
    BS_ITEM_MAX_SESSIONS          = 130,
    BS_ITEM_ACTIVE_SESSIONS       = 131,

    BS_ITEM_INIT_CURRENT_VAL      = 140,
    BS_ITEM_LIMIT_CURRENT_VAL     = 140,
    BS_ITEM_MAX_CURRENT_VAL       = 140,
    BS_ITEM_ACTUAL_CURRENT_VAL     = 140,

    // thresholds

    // eventID to send for events ?

};

//-----------------------------------------------------------------------------------------------------------
//
//
//-----------------------------------------------------------------------------------------------------------
const uint32_t  MAIN_TRACK_STATE_TIME_INTERVAL  = 10;
const uint32_t  PROG_TRACK_STATE_TIME_INTERVAL  = 10;
const uint32_t  SESSION_REFRESH_TASK_INTERVAL   = 50;

const uint16_t  MAX_CAB_SESSIONS                = 64;

//-----------------------------------------------------------------------------------------------------------
// For creating the Loco Session object the session map object is described by the following descriptor.
//
//-----------------------------------------------------------------------------------------------------------
struct LcsBaseStationSessionMapDesc {

    uint16_t    options       = SM_OPT_DEFAULT_SETTING;
    uint16_t    maxSessions   = MAX_CAB_SESSIONS;
};

//-----------------------------------------------------------------------------------------------------------
// For creating the DCC track object, the track is described by the data structure below. In addition to the
// hardware pins enablePin, dcc1Pin1, dccPin2 and sensePin, there are the limits for current consumption
// values, all specified in milliAmps. The initial current sets the current consumption limit after the track
```

```
281 // is turned on. The limit current consumption specifies the actual configured value that is checked for a
282 // track current overload situation. The maximum current defines what current the power module should never
283 // exceed. For the measurements to work, the power module needs to deliver a voltage that corresponds to the
284 // current drawn on the track. The value is measured in milliVolt per Ampere drawn. Finally, there are
285 // threshold times for managing the track overload and restart capability.
286 //
287 //-----------------------------------------------------------------------------------------------------------
288 struct LcsBaseStationTrackDesc {
289
290     uint16_t  options                       = SM_OPT_DEFAULT_SETTING;
291
292     uint8_t   enablePin                     = CDC::UNDEFINED_PIN;
293     uint8_t   dccSigPin1                    = CDC::UNDEFINED_PIN;
294     uint8_t   dccSigPin2                    = CDC::UNDEFINED_PIN;
295     uint8_t   sensePin                      = CDC::UNDEFINED_PIN;
296     uint8_t   uartRxPin                     = CDC::UNDEFINED_PIN;
297
298     uint16_t  initCurrentMilliAmp           = 0;
299     uint16_t  limitCurrentMilliAmp          = 0;
300     uint16_t  maxCurrentMilliAmp            = 0;
301     uint16_t  milliVoltPerAmp               = 0;
302
303     uint16_t  startTimeThresholdMillis      = 0;
304     uint16_t  stopTimeThresholdMillis       = 0;
305     uint16_t  overloadTimeThresholdMillis   = 0;
306     uint16_t  overloadEventThreshold        = 0;
307     uint16_t  overloadRestartThreshold      = 0;
308 };
309
310 //-----------------------------------------------------------------------------------------------------------
311 // DCC track definition. The DCC track object is responsible for managing the track power as well as building
312 // and sending the DCC packet bit stream. A packet consists of the preamble bits, the postamble bit, the data
313 // bytes separated with a ZERO bit and a checksum byte. Creating the DCC bit stream is done with the signal
314 // generation routines. The signal state machine, running on a 29 microsecond tick, takes a DCC packet and
315 // gets it out to the track. The DCC signal state machine also invokes follow up actions that measure the
316 // actual power consumption, read in a railcom message and so on. There is also a DCC log facility which
317 // records internal events for testing and debugging.
318 //
319 // The other state machine will manage the actual track power. This machine is responsible for the periodic
320 // checking of power consumption and resulting power control. In contrast to the DCC signal state machine,
321 // this machine is not driven by a periodic interrupt but invoked periodically via the LCS runtime task
322 // manager.
323 //
324 // For a base station, there will be two track objects. One is the MAIN track and the other one is the PROG
325 // track. Each track has a DCC track object associated with it. In addition to the two track objects, there
326 // are class level static routines to manage the timer hardware functions, the analog signal read for current
327 // measurement and the serial IO for the optional RailCom message processing. The current version is AtMega
328 // specific.
329 //
330 //-----------------------------------------------------------------------------------------------------------
331 struct LcsBaseStationDccTrack {
332
333     public:
334
335     LcsBaseStationDccTrack( );
336
337     uint8_t                     setupDccTrack( LcsBaseStationTrackDesc* trackDesc );
338     void                        loadPacket( const uint8_t *packet, uint8_t len, uint8_t repeat = 0 );
339
340     uint16_t                    getFlags( );
341     uint16_t                    getOptions( );
342
343     bool                        isServiceModeOn( );
344     void                        serviceModeOn( );
345     void                        serviceModeOff( );
346
347     void                        runDccTrackStateMachine( );
348     void                        powerStart( );
349     void                        powerStop( );
350     bool                        isPowerOn( );
351     bool                        isPowerOverload( );
352
353     void                        cutoutOn( );
354     void                        cutoutOff( );
355     bool                        isCutoutOn( );
356
357     void                        railComOn( );
358     void                        railComOff( );
359     bool                        isRailComOn( );
360
361     void                        setLimitCurrent( uint16_t val );
362     uint16_t                    getLimitCurrent( );
363     uint16_t                    getActualCurrent( );
364     uint16_t                    getInitCurrent( );
365     uint16_t                    getMaxCurrent( );
366     uint16_t                    getRMSCurrent( );
367
368     uint16_t                    decoderAckBaseline( uint8_t resetPacketsToSend );
369     bool                        decoderAckDetect( uint16_t baseValue, uint8_t retries );
370     void                        checkOverload( );
371
372     void                        runDccSignalStateMachine( volatile uint8_t *timeToInterrupt, uint8_t *followUpAction );
373
374     void                        getNextBit( );
375     void                        getNextPacket( );
376     void                        powerMeasurement( );
377
378     void                        startRailComIO( );
379     void                        stopRailComIO( );
```

```
380     uint8_t                     handleRailComMsg( );
381     uint8_t                     getRailComMsg( uint8_t *buf, uint8_t bufLen );
382
383     uint32_t                    getDccPacketsSend( );
384     uint32_t                    getPwrSamplesTaken( );
385     uint16_t                    getPwrSamplesPerSec( );
386
387     void                        printDccTrackConfig( );
388     void                        printDccTrackStatus( );
389
390     void                        enableLog( bool arg );
391     void                        beginLog( );
392     void                        endLog( );
393     void                        printLog( );
394
395     void                        writeLogData( uint8_t id, uint8_t *buf, uint8_t len );
396     void                        writeLogId( uint8_t id );
397     void                        writeLogTs( );
398     void                        writeLogVal( uint8_t valId, uint16_t val );
399
400     private:
401
402     uint16_t                    options                     = DT_OPT_DEFAULT_SETTING;
403     volatile uint16_t           flags                       = DT_F_DEFAULT_SETTING;
404
405     volatile uint8_t            trackState                  = 0;
406     volatile uint8_t            signalState                 = 0;
407
408     volatile uint32_t           trackTimeStamp              = 0;
409     volatile uint8_t            overloadEventCount          = 0;
410     volatile uint8_t            overloadRestartCount        = 0;
411
412     uint8_t                     enablePin                   = CDC::UNDEFINED_PIN;
413     uint8_t                     dccSigPin1                  = CDC::UNDEFINED_PIN;
414     uint8_t                     dccSigPin2                  = CDC::UNDEFINED_PIN;
415     uint8_t                     sensePin                    = CDC::UNDEFINED_PIN;
416     uint8_t                     uartRxPin                   = CDC::UNDEFINED_PIN;
417
418     uint16_t                    initCurrentMilliAmp         = 0;
419     uint16_t                    limitCurrentMilliAmp        = 0;
420     uint16_t                    maxCurrentMilliAmp          = 0;
421
422     uint16_t                    startTimeThreshold          = 0;
423     uint16_t                    stopTimeThreshold           = 0;
424     uint16_t                    overloadTimeThreshold       = 0;
425     uint16_t                    overloadEventThreshold      = 0;
426     uint16_t                    overloadRestartThreshold    = 0;
427
428     uint16_t                    milliVoltPerAmp             = 0;
429     uint16_t                    digitsPerAmp                = 0;
430     volatile uint16_t           actualCurrentDigitValue     = 0;
431     volatile uint16_t           highWaterMarkDigitValue     = 0;
432     volatile uint16_t           limitCurrentDigitValue      = 0;
433     uint16_t                    ackThresholdDigitValue      = 0;
434
435     uint32_t                    totalPwrSamplesTaken        = 0;
436     uint32_t                    lastPwrSampleTimeStamp      = 0;
437
438     uint32_t                    lastPwrSamplePerSecTaken    = 0;
439     uint32_t                    lastPwrSamplePerSecTimeStamp= 0;
440     uint32_t                    pwrSamplesPerSec            = 0;
441
442     uint8_t                     preambleLen                 = 0;
443     uint8_t                     postambleLen                = 0;
444     volatile bool               currentBit                  = false;
445     volatile uint8_t            bytesSent                   = 0;
446     volatile uint8_t            bitsSent                    = 0;
447     volatile uint8_t            preambleSent                = 0;
448     volatile uint8_t            postambleSent               = 0;
449     uint32_t                    dccPacketsSend              = 0;
450
451     DccPacket                   dccBuf1;
452     DccPacket                   dccBuf2;
453     DccPacket                   *activeBufPtr   = nullptr;
454     DccPacket                   *pendingBufPtr  = nullptr;
455
456     // ??? to add....
457     // base station capabilities according to RCN200 - 4 16 bit words
458     // sample values per second for samples and dcc packets
459     // buffers for POM / XPOM data
460     // queue for POM / XPOM commands
461
462     uint8_t                     railComBufIndex                 = 0;
463     uint8_t                     railComMsgBuf[ RAILCOM_BUF_SIZE ] = { 0 };
464
465     uint8_t                     pwrSampleBufIndex                   = 0;
466     uint16_t                    pwrSampleBuf[ PWR_SAMPLE_BUF_SIZE ]     = { 0 };
467
468     public:
469
470     static void                 startDccProcessing( );
471
472 };
473
474 //-----------------------------------------------------------------------------------------------------
475 // Every allocated loco session is described by the sessionMap structure. There are the engine cab Id, speed,
476 // direction and function information. There is also a field that indicates when we received information for
477 // this session from a cab control handheld. The function flags are stored in an array, each byte representing
478 // a group. Most of the fields are actually used for a DCC type locomotive. When the locomotive is an analog
```

```
// engine, only a subset of the fields is actually used. Nevertheless, even for an analog engine we will
// have a session. The base station will however not generate packets for this engine.
//
//-----------------------------------------------------------------------------------------------------------
struct SessionMapEntry {

  uint16_t        flags               = SME_DEFAULT_SETTING;
  uint16_t        cabId               = LCS::NIL_CAB_ID;
  uint8_t         speed               = 0;
  uint8_t         speedSteps          = 128;
  uint8_t         direction           = 0;
  uint8_t         engineState         = 0;
  uint8_t         nextRefreshStep     = 0;
  unsigned long   lastKeepAliveTime   = 0;
  uint8_t         functions[ LCS::MAX_DCC_FUNC_GROUP_ID ] = { 0 };

};

//-----------------------------------------------------------------------------------------------------------
// The loco session object is the central data structure for the base station locomotive management. For a
// DCC type engine it manages the loco sessions and assembles the DCC packets and drives the DCC track objects
// to send out the relevant DCC packages. For an analog engine it will just manage the session entry and
// communicate via the LCS bus with the block controller that actually owns the engine at the moment.
//
//-----------------------------------------------------------------------------------------------------------
struct LcsBaseStationLocoSession {

  public:

    LcsBaseStationLocoSession( );

    uint8_t setupSessionMap(

        LcsBaseStationSessionMapDesc  *sessionMapDesc,
        LcsBaseStationDccTrack        *mainTrack,
        LcsBaseStationDccTrack        *progTrack
    );

    uint8_t                   requestSession( uint16_t cabId, uint8_t mode, uint8_t *sId );
    uint8_t                   releaseSession( uint8_t sId );
    uint8_t                   updateSession( uint8_t sId, uint8_t flags );

    uint8_t                   markSessionAlive( uint8_t sId );
    void                      refreshActiveSessions( );
    uint32_t                  getSessionKeepAliveInterval( );

    uint16_t                  getOptions( );
    uint16_t                  getFlags( );
    uint8_t                   getSessionMapHwm( );
    uint8_t                   getActiveSessions( );
    uint8_t                   getSessionIdByCabId( uint16_t cabId );
    void                      emergencyStopAll( );

    uint8_t                   setThrottle( uint8_t sId, uint8_t speed, uint8_t direction );
    uint8_t                   setDccFunctionBit( uint8_t sId, uint8_t funcNum, uint8_t val );
    uint8_t                   setDccFunctionGroup( uint8_t sId, uint8_t fGroup, uint8_t dccByte );

    uint8_t                   writeCVMain( uint8_t sId, uint16_t cvId, uint8_t mode, uint8_t val );
    uint8_t                   writeCVByteMain( uint8_t sId, uint16_t cvId, uint8_t val );
    uint8_t                   writeCVBitMain( uint8_t sId, uint16_t cvId, uint8_t bitPos, uint8_t val );

    uint8_t                   readCV( uint16_t cvId, uint8_t mode, uint8_t *val );
    uint8_t                   readCVByte( uint16_t cvId, uint8_t *val );
    uint8_t                   readCVBit( uint16_t cvId, uint8_t bitPos, uint8_t *val );

    uint8_t                   writeCV( uint16_t cvId, uint8_t mode, uint8_t val );
    uint8_t                   writeCVByte( uint16_t cvId, uint8_t val );
    uint8_t                   writeCVBit( uint16_t cvId, uint8_t bitPos, uint8_t val );

    uint8_t                   writeDccPacketMain( uint8_t *buf,  uint8_t len, uint8_t nRepeat );
    uint8_t                   writeDccPacketProg( uint8_t *buf,  uint8_t len, uint8_t nRepeat );

    void                      printSessionMapConfig( );
    void                      printSessionMapInfo( );

    SessionMapEntry           *lookupSessionEntry( uint16_t cabId );
    SessionMapEntry           *getSessionMapEntryPtr( uint8_t sId );

    private:

    uint8_t                   setThrottle( SessionMapEntry *csptr, uint8_t speed, uint8_t direction );
    uint8_t                   setDccFunctionGroup( SessionMapEntry *csPtr, uint8_t fGroup, uint8_t dccByte );

    SessionMapEntry           *allocateSessionEntry( uint16_t cabId );
    void                      deallocateSessionEntry( SessionMapEntry *csPtr );
    void                      refreshSessionEntry( SessionMapEntry *csPtr );
    void                      initSessionEntry( SessionMapEntry *csPtr );
    void                      printSessionEntry( SessionMapEntry *csPtr );

    private:

    LcsBaseStationDccTrack    *mainTrack              = nullptr;
    LcsBaseStationDccTrack    *progTrack              = nullptr;

    uint16_t                  options                 = DT_OPT_DEFAULT_SETTING;
    uint16_t                  flags                   = DT_F_DEFAULT_SETTING;
    uint32_t                  lastAliveCheckTime      = 0L;
    uint32_t                  refreshAliveTimeOutVal  = 2000L;  // ??? a constant name ...

```

```
578     SessionMapEntry           *sessionMap            = nullptr;
579     SessionMapEntry           *sessionMapNextRefresh  = nullptr;
580     SessionMapEntry           *sessionMapHwm          = nullptr;
581     SessionMapEntry           *sessionMapLimit        = nullptr;
582
583 };
584
585 //-----------------------------------------------------------------------------------------------------------------
586 // One of the key duties of the base station is to listen and react to DCC commands coming via the LCS bus.
587 // The interface works very closely with the session management and the two DCC track objects.
588 //
589 // ??? how about we make the handleLcsMsg handler a routine vs. an object ?
590 // ??? would make the any REQ/REP scheme easier ?
591 //-----------------------------------------------------------------------------------------------------------------
592 struct LcsBaseStationMsgInterface {
593
594     public:
595
596     LcsBaseStationMsgInterface( );
597
598     uint8_t setupLcsMsgInterface( LcsBaseStationLocoSession   *locoSessions,
599                                   LcsBaseStationDccTrack      *mainTrack,
600                                   LcsBaseStationDccTrack      *progTrack
601                                 );
602
603     void handleLcsMsg( uint8_t *msg );
604
605     private:
606
607     LcsBaseStationLocoSession   *locoSessions   = nullptr;
608     LcsBaseStationDccTrack      *mainTrack      = nullptr;
609     LcsBaseStationDccTrack      *progTrack      = nullptr;
610
611 };
612
613 //-----------------------------------------------------------------------------------------------------------------
614 // The base station implements a serial IO command interface. The command interface uses the DCC++ syntax of
615 // a command line and where it is a original DCC++ command it implements them in a compatible way. The idea
616 // is to one day connect to the programs of the JMRI world, which support the DCC++ style command interface.
617 //
618 //-----------------------------------------------------------------------------------------------------------------
619 struct LcsBaseStationCommand {
620
621     public:
622
623     LcsBaseStationCommand( );
624
625     uint8_t setupSerialCommand( LcsBaseStationLocoSession  *locoSessions,
626                                 LcsBaseStationDccTrack     *mainTrack,
627                                 LcsBaseStationDccTrack     *progTrack );
628
629     void handleSerialCommand( char *s );
630
631     private:
632
633     void openSessionCmd( char *s );
634     void closeSessionCmd( char *s );
635
636     void setThrottleCmd( char *s );
637     void setFunctionBitCmd( char *s );
638     void setFunctionGroupCmd( char *s );
639     void emergencyStopCmd( );
640
641     void readCVCmd( char *s );
642     void writeCVByteCmd( char *s );
643     void writeCVBitCmd( char *s );
644     void writeCVByteMainCmd( char *s );
645     void writeCVBitMainCmd( char *s );
646
647     void writeDccPacketMainCmd( char *s );
648     void writeDccPacketProgCmd( char *s );
649
650     void setTrackOptionCmd( char *s );
651     void turnPowerOnAllCmd( );
652     void turnPowerOnMainCmd( );
653     void turnPowerOnProgCmd( );
654     void turnPowerOffAllCmd( );
655
656     void printStatusCmd( char *s );
657     void printTrackCurrentCmd( char *s );
658     void printBaseStationConfigCmd( );
659     void printHelpCmd( );
660     void printVersionInfo( );
661     void printConfiguration( );
662     void printSessionMap( );
663     void printTrackStatusMain( );
664     void printTrackStatusProg( );
665
666     void printDccLogCommand( char *s );
667
668     private:
669
670     LcsBaseStationLocoSession *locoSessions = nullptr;
671     LcsBaseStationDccTrack    *mainTrack    = nullptr;
672     LcsBaseStationDccTrack    *progTrack    = nullptr;
673 };
674
675 #endif
```

```
//--------------------------------------------------------------------------------------------------------------
//
// LCS Base Station - Serial Command Interface - implementation file
//
//--------------------------------------------------------------------------------------------------------------
// The serial command interface is used to directly send commands to the session and DCC track objects. The
// command syntax is patterned after the DCC++ command syntax. Available commands that have a DCC++ counter
// part are implemented exactly after the DCC++ command specification. The main motivation is to use this
// interface for testing and debugging as well as third party tools that also implement the DCC++ command set
// to send commands to this base station as well when calling the serial IO interface. For the layout control
// system, the approach would rather be to send LCS messages for all tasks.
//
//--------------------------------------------------------------------------------------------------------------
//
// LCS - Base Station
// Copyright (C) 2019 - 2024  Helmut Fieres
//
// This program is free software: you can redistribute it and/or modify it under the terms of the GNU General
// Public License as published by the Free Software Foundation, either version 3 of the License, or (at your
// option) any later version.
//
// This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
// implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
// for more details.
//
// You should have received a copy of the GNU General Public License along with this program. If not, see
// http://www.gnu.org/licenses
//
//  GNU General Public License:  http://opensource.org/licenses/GPL-3.0
//
//--------------------------------------------------------------------------------------------------------------
#include "LcsBaseStation.h"

using namespace LCS;

//--------------------------------------------------------------------------------------------------------------
// External global variables.
//
//--------------------------------------------------------------------------------------------------------------
extern uint16_t debugMask;

//--------------------------------------------------------------------------------------------------------------
// The object constructor. Nothing to do here.
//
//--------------------------------------------------------------------------------------------------------------
LcsBaseStationCommand::LcsBaseStationCommand( ) { }

//--------------------------------------------------------------------------------------------------------------
// The object setup command. We need to remember the other objects we use in handling the commands. For the
// serial IO itself nothing to do, it was already done in the LCS runtime setup.
//
//--------------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationCommand::setupSerialCommand(

    LcsBaseStationLocoSession *locoSessions,
    LcsBaseStationDccTrack    *mainTrack,
    LcsBaseStationDccTrack    *progTrack ) {

    this -> locoSessions  = locoSessions;
    this -> mainTrack     = mainTrack;
    this -> progTrack     = progTrack;

    return ( ALL_OK );
}

//--------------------------------------------------------------------------------------------------------------
// "handleSerialCommand" analyzes the command line and invokes the respective command handler. The first
// character in a command is the command letter. The command is followed by the arguments. For compatibility
// with the DCC++ original command set, each command that is also a DCC++ command is implemented exactly as
// the original. This allows external tools, such as the JMRI Decoder Pro configuration tool to be used. The
// command handler supports command sequences "<" ... ">" in one line which are processed once the carriage
// return is hit.
//
//--------------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::handleSerialCommand( char *s ) {

    int     charIndex       = 0;
    char    cmdStr[ 256 ]   = { 0 };

    while ( s[ charIndex ] != '\0' ) {

        switch ( s[ charIndex ] ) {

            case '<': {

                cmdStr[ 0 ] = '\0';
                charIndex ++;

            } break;

            case '>': {

                switch ( cmdStr[ 0 ] ) {

                    case 'O': openSessionCmd( cmdStr + 1 ); break;
                    case 'K': closeSessionCmd( cmdStr + 1 ); break;

                    case 't': setThrottleCmd( cmdStr + 1 );  break;
```

```
                    case 'f': setFunctionGroupCmd( cmdStr + 1 ); break;
                    case 'v': setFunctionBitCmd( cmdStr + 1 ); break;

                    case 'R': readCVCmd( cmdStr + 1 ); break;
                    case 'W': writeCVByteCmd( cmdStr + 1 ); break;
                    case 'B': writeCVBitCmd( cmdStr + 1 ); break;
                    case 'w': writeCVByteMainCmd( cmdStr + 1 ); break;
                    case 'b': writeCVBitMainCmd( cmdStr + 1 ); break;

                    case 'M': writeDccPacketMainCmd( cmdStr + 1 ); break;
                    case 'P': writeDccPacketProgCmd( cmdStr + 1 ); break;

                    case 'C': setTrackOptionCmd( cmdStr + 1 ); break;
                    case 'Y': printDccLogCommand( cmdStr + 1 ); break;

                    case 'X': emergencyStopCmd( ); break;
                    case 'O': turnPowerOffAllCmd( ); break;
                    case '1': turnPowerOnAllCmd( ); break;
                    case '2': turnPowerOnMainCmd( ); break;
                    case '3': turnPowerOnProgCmd( ); break;

                    case 's': printStatusCmd( cmdStr + 1 ); break;
                    case 'S': printBaseStationConfigCmd( ); break;
                    case 'L': printSessionMap( ); break;

                    case 'a': printTrackCurrentCmd( cmdStr + 1 ); break;

                    case '?': printHelpCmd( ); break;

                    case ' ': printf( "\n" ); break;

                    case 'e':
                    case 'E':
                    case 'D':
                    case 'T':
                    case 'Z':
                    case 'Q':
                    case 'F': printf( "<Not implemented>\n" ); break;

                    default: printf( "<Unknown command, use '?' for help>\n" );
                }

                charIndex ++;

            } break;

            default: {

                if ( strlen( cmdStr ) < sizeof( cmdStr) ) strncat( cmdStr, &s[ charIndex ], 1 );
                charIndex ++;
            }
        }
    }
}

//--------------------------------------------------------------------------------------------------------------
// "openSessionCmd" handles the session creation command. This command is used to allocate a loco session.
// We are passed the cab ID and return a session Id.
//
//    <O cabId>
//
//    cabId      -  the requesting cab number, from 1 to MAX_CAB_ID.
//
//    returns: <O sId>
//
//--------------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::openSessionCmd( char *s ) {

    uint16_t  cabId = NIL_CAB_ID;
    uint8_t   sId   = 0;

    if ( sscanf( s, "%hu", &cabId ) != 1 ) return;

    int ret = locoSessions -> requestSession( cabId, LSM_NORMAL, &sId );

    printf( "<O %d>", (( ret == ALL_OK ) ? sId : -1 ));
}

//--------------------------------------------------------------------------------------------------------------
// "closeSessionCmd" handles the session release command. The return code is the CabSession error code. A zero
// indicates a successful execution.
//
//    <K sId>
//
//    sId      -  the session number.
//
//    returns: <K status>
//
//--------------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::closeSessionCmd( char *s ) {

    uint8_t sId = NIL_LOCO_SESSION_ID;

    if ( sscanf( s, "%hhu", &sId ) != 1 ) return;

    int ret = locoSessions -> releaseSession( sId );

    printf( "<K %d>", ret );
}
```

```
198
199 //--------------------------------------------------------------------------------------------------------------
200 // "setThrottleCmd" handles the throttle command. The original DCC++ interface uses both the register Id and
201 // the cabId. In the new version the sId is sufficient. But just to be compatible with the original
202 // DCC++ command, we also pass the cabId. It should be either zero or match the cabId in the allocated session.
203 //
204 //    <t sId cabId speed direction>
205 //
206 //    sId         -  the allocated session number.
207 //    cabId       -  the Cab Id. The number must match the can number in the session or be zero.
208 //    speed       -  throttle speed from 0-126, or -1 for emergency stop (resets SPEED to 0)
209 //    direction   -  the direction: 1=forward, 0=reverse. Setting direction when speed=0 only effects
210 //                   direction of cab lighting for a stopped train.
211 //
212 //    returns: <t sId speed direction >
213 //
214 //--------------------------------------------------------------------------------------------------------------
215 void LcsBaseStationCommand::setThrottleCmd( char *s ) {
216
217   uint8_t   sId         = NIL_LOCO_SESSION_ID;
218   uint16_t  cabId       = NIL_CAB_ID;
219   uint8_t   speed       = 0;
220   uint8_t   direction   = 0;
221
222   if ( sscanf( s, "%hhu %hu %hhu %hhu", &sId, &cabId, &speed, &direction ) != 4 ) return;
223   if (( cabId != NIL_CAB_ID ) && ( locoSessions -> getSessionIdByCabId( cabId ) != sId )) return;
224
225   locoSessions -> setThrottle( sId, speed, direction );
226
227   printf( "<t %d %d %d>", sId, speed, direction );
228 }
229
230 //--------------------------------------------------------------------------------------------------------------
231 // "setFunctionBitCmd" turns on and off the engine decoder functions F0-F68 (F0 is sometimes called FL). This
232 // new command directly transmits the function setting to the engine decoder. The command interface is
233 // handling one function number at a time. The base station will handle the DCC byte generation.
234 //
235 //    <v sId funcId val >
236 //
237 //    sId     -  the allocated session number, from 1 to MAX_MAIN_REGISTERS.
238 //    funcId  -  the function number, currently implemented for F0 - F68.
239 //    val     -  the value to set, 1 or 0.
240 //
241 //    returns: NONE.
242 //
243 //--------------------------------------------------------------------------------------------------------------
244 void LcsBaseStationCommand::setFunctionBitCmd( char *s ) {
245
246     uint8_t sId = NIL_LOCO_SESSION_ID;
247     uint8_t funcNum    = 0;
248     uint8_t val        = 0;
249
250     if ( sscanf( s, "%hhu %hhu %hhu", &sId, &funcNum, &val ) != 3 ) return;
251
252     locoSessions -> setDccFunctionBit( sId, funcNum, val );
253 }
254
255 //--------------------------------------------------------------------------------------------------------------
256 // "setFunctionGroupCmd" sets the engine decoder functions F0-F68 by group byte using the DCC byte instruction
257 // format. The user needs to do the calculation as shown in the list below. This command directly transmits
258 // the command to the engine decoder. This function requires some user math, and is only there for the DCC++
259 // command interface compatibility.
260 //
261 //    <f cabId byte1 [ byte2 ] >
262 //
263 //    cabId          -  the cab number
264 //    byte1          -  see below for encoding
265 //    byte2          -  see below for encoding
266 //
267 //    returns: NONE
268 //
269 //    The DCC packet data for setting function groups is defined as follows:
270 //
271 //      Group 1:  F0, F4, F3, F2, F1      DCC Command Format: 100DDDDD
272 //      Group 2:  F8, F7, F6, F5          DCC Command Format: 1011DDDD
273 //      Group 3:  F12, F11, F10, F9       DCC Command Format: 1010DDDD
274 //      Group 4:  F20 .. F13              DCC Command Format: 0xDE DDDDDDDD
275 //      Group 5:  F28 .. F21              DCC Command Format: 0xDF DDDDDDDD
276 //      Group 6:  F36 .. F29              DCC Command Format: 0xD8 DDDDDDDD
277 //      Group 7:  F44 .. F37              DCC Command Format: 0xD9 DDDDDDDD
278 //      Group 8:  F52 .. F45              DCC Command Format: 0xDA DDDDDDDD
279 //      Group 9:  F60 .. F53              DCC Command Format: 0xDB DDDDDDDD
280 //      Group 10: F68 .. F61              DCC Command Format: 0xDC DDDDDDDD
281 //
282 //    To set functions F0-F4 on (=1) or off (=0):
283 //
284 //      BYTE1:  128 + F1*1 + F2*2 + F3*4 + F4*8 + F0*16
285 //      BYTE2:  omitted
286 //
287 //    To set functions F5-F8 on (=1) or off (=0):
288 //
289 //      BYTE1:  176 + F5*1 + F6*2 + F7*4 + F8*8
290 //      BYTE2:  omitted
291 //
292 //    To set functions F9-F12 on (=1) or off (=0):
293 //
294 //      BYTE1:  160 + F9*1 +F10*2 + F11*4 + F12*8
295 //      BYTE2:  omitted
296 //
```

```
297 //    For the remaining groups, the two byte format is used. Byte one is:
298 //
299 //          0xde ( 222 ) -> F13-F20
300 //          0xdf ( 223 ) -> F21-F28
301 //          0xd8 ( 216 ) -> F29-F36
302 //          0xd9 ( 217 ) -> F37-F44
303 //          0xda ( 218 ) -> F45-F52
304 //          0xdb ( 219 ) -> F53-F60
305 //          0xdc ( 220 ) -> F61-F68
306 //
307 //    Byte two with N being the starting group index is always:
308 //
309 //      BYTE2: (FN)*1 + (FN+1)*2 + (FN+2)*4 + (FN+3)*8 + (FN+4)*16 + (FN+5)*32 + (FN+6)*64 + (FN+7)*128
310 //
311 //-------------------------------------------------------------------------------------------------------------
312 void LcsBaseStationCommand::setFunctionGroupCmd( char *s ) {
313
314   uint16_t  cabId    = NIL_CAB_ID;
315   uint8_t   byte1    = 0;
316   uint8_t   byte2    = 0;
317
318     if ( sscanf( s, "%hu %hhu %hhu", &cabId, &byte1, &byte2 ) < 2 ) return;
319
320     uint8_t sId = locoSessions -> getSessionIdByCabId( cabId );
321
322     if ( sId == NIL_LOCO_SESSION_ID ) return;
323
324     if (( byte2 == 0 ) && ( byte1 >= 128 ) && ( byte1 < 160 )) {
325
326         locoSessions -> setDccFunctionGroup( sId, 1, byte1 );
327     }
328     else if (( byte2 == 0 ) && ( byte1 >= 160 ) && ( byte1 < 176 )) {
329
330         locoSessions -> setDccFunctionGroup( sId, 3, byte1 );
331     }
332     else if (( byte2 == 0 ) && ( byte1 >= 176 ) && ( byte1 < 192 )) {
333
334         locoSessions -> setDccFunctionGroup( sId, 2, byte1 );
335     }
336     else if ( byte1 == 0xde ) locoSessions -> setDccFunctionGroup( sId, 4, byte2 );
337     else if ( byte1 == 0xdf ) locoSessions -> setDccFunctionGroup( sId, 5, byte2 );
338     else if ( byte1 == 0xd8 ) locoSessions -> setDccFunctionGroup( sId, 6, byte2 );
339     else if ( byte1 == 0xd9 ) locoSessions -> setDccFunctionGroup( sId, 7, byte2 );
340     else if ( byte1 == 0xda ) locoSessions -> setDccFunctionGroup( sId, 8, byte2 );
341     else if ( byte1 == 0xdb ) locoSessions -> setDccFunctionGroup( sId, 9, byte2 );
342     else if ( byte1 == 0xdc ) locoSessions -> setDccFunctionGroup( sId, 10, byte2 );
343 }
344
345 //-------------------------------------------------------------------------------------------------------------
346 // "readCVCmd" reads a configuration variable from the engine decoder on the programming track. The
347 // callbacknum and callbacksub parameter are ignored by the base station and just passed back to the caller
348 // for identification purposes.
349 //
350 //    <R cvId [ callbacknum callbacksub ]>
351 //
352 //    cvId          -  the configuration variable ID, 1 ... 1024.
353 //    callbacknum   -  a number echoed back, ignored by the base station
354 //    callbacksub   -  a number echoed back, ignored by the base station
355 //
356 //    returns: <R callbacknum|callbacksub|cvId value>
357 //
358 //    where value is 0 - 255 of the CV variable or -1 if the value could not be verified.
359 //
360 //-------------------------------------------------------------------------------------------------------------
361 void LcsBaseStationCommand::readCVCmd( char *s ) {
362
363     uint16_t  cvId          = NIL_DCC_CV_ID;
364     uint8_t   val           = 0;
365     int       callbacknum   = 0;
366     int       callbacksub   = 0;
367     int       ret           = 0;
368
369     if ( sscanf( s, "%hu %d %d", &cvId, &callbacknum, &callbacksub )  < 1 ) return;
370
371     ret = locoSessions -> readCV( cvId, 0, &val );
372
373     printf( "<R %d|%d|%d %d>", callbacknum, callbacksub, cvId, (( ret == ALL_OK ) ? val : -1 ));
374 }
375
376 //-------------------------------------------------------------------------------------------------------------
377 // "writeCVByteCmd" writes a data byte to the engine decoder on the programming track and then verifies it.
378 // The callbacknum and callbacksub parameter are ignored by the base station and just passed back to the
379 // caller for identification purposes.
380 //
381 //    <W cvId val [ callbacknum callbacksub ]>
382 //
383 //    cvId          -  the configuration variable ID, 1 ... 1024.
384 //    val           -  the data byte.
385 //    callbacknum   -  a number echoed back, ignored by the base station
386 //    callbacksub   -  a number echoed back, ignored by the base station
387 //
388 //    returns: <W callbacknum|callbacksub|cvId Value>
389 //
390 //    where Value is 0 - 255 of the CV variable or -1 if the verification failed.
391 //
392 //-------------------------------------------------------------------------------------------------------------
393 void LcsBaseStationCommand::writeCVByteCmd( char *s ) {
394
395     uint16_t  cvId          = NIL_DCC_CV_ID;
```

```
396     uint8_t   val           = 0;
397     int       callbacknum   = 0;
398     int       callbacksub   = 0;
399     int       ret           = 0;
400
401     if ( sscanf( s, "%hu %hhu %d %d", &cvId, &val, &callbacknum, &callbacksub ) < 2 ) return;
402
403     ret = locoSessions -> writeCVByte( cvId, val );
404
405     printf( "<W %d|%d|%d %d>", callbacknum, callbacksub, cvId, (( ret == ALL_OK ) ? val : -1 ));
406 }
407
408 //--------------------------------------------------------------------------------------------------------------
409 // "writeCVBitCmd" writes a bit to the engine decoder on the programming track and then verifies the
410 // operation. The callbacknum and callbacksub parameter are ignored by the base station and just passed back
411 // to the caller for identification purposes.
412 //
413 //    <B cvId bitPos bitVal callbacknum callbacksub>
414 //
415 //    cvId          -  the configuration variable ID, 1 ... 1024.
416 //    bitPos        -  the bit position of the bit, 0 .. 7.
417 //    bitVal        -  the data bit.
418 //    callbacknum   -  a number echoed back, ignored by the base station
419 //    callbacksub   -  a number echoed back, ignored by the base station
420 //
421 //    returns: <B callbacknum|callbacksub|cvId bitPos Value>
422 //
423 //    where Value is 0 or 1 of the bit or -1 if the verification failed.
424 //
425 //--------------------------------------------------------------------------------------------------------------
426 void LcsBaseStationCommand::writeCVBitCmd( char *s ) {
427
428     uint16_t  cvId          = NIL_DCC_CV_ID;
429     uint8_t   bitPos        = 0;
430     uint8_t   bitVal        = 0;
431     int       callbacknum   = 0;
432     int       callbacksub   = 0;
433     int       ret           = 0;
434
435     if ( sscanf( s, "%hu %hhu %hhu %d %d", &cvId, &bitPos, &bitVal, &callbacknum, &callbacksub ) != 5 ) return;
436
437     ret = locoSessions -> writeCVBit( cvId, bitPos, bitVal );
438
439     printf( "<B %d|%d|%d|%d %d>", callbacknum, callbacksub, cvId, bitPos, (( ret == ALL_OK ) ? bitVal : -1 ));
440 }
441
442 //--------------------------------------------------------------------------------------------------------------
443 // "writeCVByteMainCmd" writes a data byte to the engine decoder on the main track, without any verification.
444 // To be compatible with the DCC++ command set, the command is using the cabId to identify the loco we talk
445 // about.
446 //
447 //    <w cabId cvId val >
448 //
449 //    cabId       -  the cabId number.
450 //    cvId        -  the configuration variable ID, 1 ... 1024.
451 //    val         -  the data byte.
452 //
453 //    returns: NONE
454 //
455 //--------------------------------------------------------------------------------------------------------------
456 void LcsBaseStationCommand::writeCVByteMainCmd( char *s ) {
457
458     uint16_t  cabId = NIL_CAB_ID;
459     uint16_t  cvId  = NIL_DCC_CV_ID;
460     uint8_t   val   = 0;
461
462     if ( sscanf( s, "%hu %hu %hhu", &cabId, &cvId, &val ) != 3 ) return;
463
464     locoSessions -> writeCVByteMain( locoSessions -> getSessionIdByCabId( cabId ), cvId, val );
465 }
466
467 //--------------------------------------------------------------------------------------------------------------
468 // "writeCVBitMainCmd" writes a data byte to the engine decoder on the main track, without any verification.
469 // To be compatible with the DCC++ command set, the command is using the cabId to identify the loco we talk
470 // about.
471 //
472 //    <b cabId cvId bitPos bitVal >
473 //
474 //    cabId      -  the cabId number.
475 //    cvId       -  the configuration variable ID, 1 ... 1024.
476 //    bitPos     -  the bit position of the bit, 0 .. 7.
477 //    bitVal     -  the data bit.
478 //
479 //    returns: NONE
480 //
481 //--------------------------------------------------------------------------------------------------------------
482 void LcsBaseStationCommand::writeCVBitMainCmd( char *s ) {
483
484     uint16_t  cabId = NIL_CAB_ID;
485     uint16_t  cvId    = NIL_DCC_CV_ID;
486     uint8_t   bitPos = 0;
487     uint8_t   bitVal = 0;
488
489     if ( sscanf(s, "%hu %hu %hhu %hhu", &cabId, &cvId, &bitPos, &bitVal ) != 4 ) return;
490
491     locoSessions -> writeCVBitMain( locoSessions -> getSessionIdByCabId( cabId ), cvId, bitPos, bitVal );
492 }
493
494 //--------------------------------------------------------------------------------------------------------------
```

```
495 // "writeDccPacketMainCmd" writes a DCC packet to the main operations track. This is for testing and debugging
496 // and you better know the DCC packet standard by heart :-). The DCC standards define packets up to 15 data
497 // bytes payload.
498 //
499 //    <M byte1 byte2 [ byte3 ... byte10 ]>
500 //
501 //    byte1 .. byte10   - the packet data in hexadecimal
502 //
503 //    returns: NONE
504 //
505 //---------------------------------------------------------------------------------------------------------------
506 void LcsBaseStationCommand::writeDccPacketMainCmd( char *s ) {
507
508     uint8_t b[ 16 ] = { 0 };
509     uint8_t nBytes  = sscanf( s,
510                               "%hhu %hhu %hhu %hhu %hhu %hhu %hhu %hhu"
511                               "%hhu %hhu %hhu %hhu %hhu %hhu %hhu %hhu",
512                               b, b + 1, b + 2, b + 3, b + 4, b + 5, b + 6, b + 7,
513                               b + 8, b + 9, b + 10, b + 11, b + 12, b + 13, b + 14, b + 15 );
514
515     if ( nBytes >= 3 && nBytes <= 10 ) locoSessions -> writeDccPacketMain( b, nBytes, 0 );
516 }
517
518 //---------------------------------------------------------------------------------------------------------------
519 // "writeDccPacketProgCmd" writes a DCC packet to the programming track. This is for testing and debugging and
520 // you better know the DCC packet standard by heart :-). The DCC standards define packets up to 15 data
521 // bytes payload.
522 //
523 ///    <P byte1 byte2 [ byte3 ... byte10 ]>
524 //
525 //    byte1 .. byte10   - the packet data in hexadecimal
526 //
527 //    returns: NONE
528 //
529 //---------------------------------------------------------------------------------------------------------------
530 void LcsBaseStationCommand::writeDccPacketProgCmd( char *s ) {
531
532     uint8_t b[ 16 ] = { 0 };
533     uint8_t nBytes  = sscanf( s,
534                               "%hhu %hhu %hhu %hhu %hhu %hhu %hhu %hhu"
535                               "%hhu %hhu %hhu %hhu %hhu %hhu %hhu %hhu",
536                               b, b + 1, b + 2, b + 3, b + 4, b + 5, b + 6, b + 7,
537                               b + 8, b + 9, b + 10, b + 11, b + 12, b + 13, b + 14, b + 15 );
538
539     if ( nBytes >= 3 && nBytes <= 10 ) locoSessions -> writeDccPacketProg( b, nBytes, 0 );
540 }
541
542 //---------------------------------------------------------------------------------------------------------------
543 // "emergencyStopCmd" handles the emergencyStop command. This new command causes the base station to send out
544 // the emergency stop broadcast DCC command.
545 //
546 //    <X>
547 //
548 //    returns: <X>
549 //
550 //---------------------------------------------------------------------------------------------------------------
551 void LcsBaseStationCommand::emergencyStopCmd( ) {
552
553     locoSessions -> emergencyStopAll( );
554     printf( "<X>" );
555 }
556
557 //---------------------------------------------------------------------------------------------------------------
558 // "turnPowerOnXXX" and "turnPowerOff" enables/disables the main and/or the programming track.
559 //
560 //    <0> - turn operations and programming track power off
561 //    <1> - turn operations and programming track power on
562 //    <2> - turn operations track power on
563 //    <3> - turn programming track power on
564 //
565 //---------------------------------------------------------------------------------------------------------------
566 void LcsBaseStationCommand::turnPowerOnAllCmd( ) {
567
568     mainTrack -> powerStart( );
569     progTrack -> powerStart( );
570     printf( "<p1>" );
571 }
572
573 void LcsBaseStationCommand::turnPowerOffAllCmd( ) {
574
575     mainTrack -> powerStop( );
576     progTrack -> powerStop( );
577     printf( "<p0>" );
578 }
579
580 void LcsBaseStationCommand::turnPowerOnMainCmd( ) {
581
582     mainTrack -> powerStart( );
583     printf( "<p1 MAIN>" );
584 }
585
586 void LcsBaseStationCommand::turnPowerOnProgCmd( ) {
587
588     progTrack -> powerStart( );
589     printf( "<p1 PROG>" );
590 }
591
592 //---------------------------------------------------------------------------------------------------------------
593 // "setTrackOptionCmd" turns on and off capabilities of the operations or service track.
```

```
//
//    <C option>
//
//    option   - the option value.
//
//        1 -> set main track Cutout mode on.
//        2 -> set main track Cutout mode off.
//        3 -> set main track Railcom mode on.
//        4 -> set main track Railcom mode off.
//
//        10 -> set service track into operations mode.
//        11 -> set service track into service mode.
//
//    returns: NONE
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::setTrackOptionCmd( char *s ) {

    uint8_t option = 0;

    if ( sscanf( s, "%hhu", &option ) == 1 ) {

        switch ( option ) {

            case 1: mainTrack -> cutoutOn( );   break;
            case 2: mainTrack -> cutoutOff( );  break;
            case 3: mainTrack -> railComOn( );  break;
            case 4: mainTrack -> railComOff( ); break;

            case 10: progTrack -> serviceModeOff( );  break;
            case 11: progTrack -> serviceModeOn( ); break;
        }
    }
}

//----------------------------------------------------------------------------------------------------------
// "printStatusCmd" list information about the base station. Using just a "s" for a summary status is always
// a good idea to do this just as a first basic test if things are running at all. The level is a positive
// integer that specifies the information items to be listed.
//
//    <s [ opt ]> - the kind of status to display.
//
//    returns:  series of status information that can be read by an interface to determine status of the base
//              station and important settings
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::printStatusCmd( char *s ) {

    uint8_t opt = 0;

    if ( sscanf( s, "%hhu", &opt ) > 0 ) {

        switch ( opt ) {

            case 0: printVersionInfo( );      break;
            case 1: printConfiguration( );    break;
            case 2: printSessionMap( );       break;
            case 3: printTrackStatusMain( );  break;
            case 4: printTrackStatusProg( );  break;

            case 9: {

                printConfiguration( );
                printSessionMap( );
                printTrackStatusMain( );
                printTrackStatusProg( );

            } break;

            default: printVersionInfo( );
        }
    } else printVersionInfo( );
}

//----------------------------------------------------------------------------------------------------------
// "printBaseStationConfigCmd" list information about the base in a DCC++ compatible way.
//
//    <S> - the basestation configuration.
//
//    returns:  series of status information that can be read by an interface to determine status of the base
//              station and important settings
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::printBaseStationConfigCmd( ) {

    printConfiguration( );
}

//----------------------------------------------------------------------------------------------------------
// "printConfiguration" lists out the key hardware and software settings. Also very useful as the first
// trouble shooting task.
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::printConfiguration( ) {

    printVersionInfo( );
    locoSessions -> printSessionMapConfig( );
    mainTrack -> printDccTrackConfig( );
    progTrack -> printDccTrackConfig( );
```

```
693 }
694
695 //--------------------------------------------------------------------------------------------------------------
696 // "printVersionInfo" list out the Arduino type and software version of this program.
697 //
698 //--------------------------------------------------------------------------------------------------------------
699 void LcsBaseStationCommand::printVersionInfo( ) {
700
701     printf( "<\nLCS Base Station / Version: tbd / %s %s >\n", __DATE__, __TIME__  );
702 }
703
704 //--------------------------------------------------------------------------------------------------------------
705 // "printSessionMap" list out the active session table content.
706 //
707 //--------------------------------------------------------------------------------------------------------------
708 void LcsBaseStationCommand::printSessionMap( ) {
709
710     locoSessions -> printSessionMapInfo( );
711 }
712
713 //--------------------------------------------------------------------------------------------------------------
714 // "printTrackStatusMain" lists out the current MAIN track status
715 //
716 //--------------------------------------------------------------------------------------------------------------
717 void LcsBaseStationCommand::printTrackStatusMain( ) {
718
719     mainTrack -> printDccTrackStatus( );
720 }
721
722 //--------------------------------------------------------------------------------------------------------------
723 // "printTrackStatusProg" lists out the current PROG track status
724 //
725 //--------------------------------------------------------------------------------------------------------------
726 void LcsBaseStationCommand::printTrackStatusProg( ) {
727
728     progTrack -> printDccTrackStatus( );
729 }
730
731 //--------------------------------------------------------------------------------------------------------------
732 // "printTrackCurrentCmd" reads the actual current being drawn on the main operations track.
733 //
734 //    <a [ track ]>
735 //
736 // where "track" == 0 or omitted is the MAIN track, "track" == 1 is the PROG track.
737 //
738 //    returns: <a current>, where current is the actual power consumption in milliAmps.
739 //
740 //--------------------------------------------------------------------------------------------------------------
741 void LcsBaseStationCommand::printTrackCurrentCmd( char *s ) {
742
743     int opt = -1;
744
745     sscanf( s, "%d", &opt );
746
747     printf( "<a " );
748
749     switch ( opt ) {
750
751         case 0: printf( "%d", mainTrack -> getActualCurrent( )); break;
752         case 1: printf( "%d", progTrack -> getActualCurrent( )); break;
753         case 2: printf( "%d %d", mainTrack -> getActualCurrent( ), progTrack -> getActualCurrent( )); break;
754
755         case 10: printf( "%d", mainTrack -> getRMSCurrent( )); break;
756         case 11: printf( "%d", progTrack -> getRMSCurrent( )); break;
757         case 12: printf( "%d %d", mainTrack -> getRMSCurrent( ), progTrack -> getRMSCurrent( )); break;
758
759         default: printf( "%d", mainTrack -> getRMSCurrent( ));
760     }
761
762     printf( ">" );
763 }
764
765 //--------------------------------------------------------------------------------------------------------------
766 // "printDccLogCommandCommand" is the command to manage the DCC log for tracing and debugging purposes.
767 //
768 //    <Y [ opt ]> where "opt" is the command to execute from the DCC Log function.
769 //
770 //      Main track:
771 //
772 //      0 - disable DCC logging
773 //      1 - enable DCC logging
774 //      2 - start DCC logging
775 //      3 - stop DCC logging
776 //      4 - list log entries
777 //
778 //      Prog track:
779 //
780 //      10 - disable DCC logging
781 //      11 - enable DCC logging
782 //      12 - start DCC logging
783 //      13 - stop DCC logging
784 //      14 - list log entries
785 //
786 //      RailCom:
787 //
788 //      20 - show real time RailCom buffer, experimental
789 //
790 //--------------------------------------------------------------------------------------------------------------
791 void LcsBaseStationCommand::printDccLogCommand( char *s ) {
```

```
792
793     int opt = -1;
794
795     sscanf( s, "%d", &opt );
796
797     printf( "<Y %d ", opt );
798
799     switch ( opt ) {
800
801         case 0:     mainTrack -> enableLog( false );  break;
802         case 1:     mainTrack -> enableLog( true );   break;
803         case 2:     mainTrack -> beginLog( );         break;
804         case 3:     mainTrack -> endLog( );           break;
805         case 4:     mainTrack -> printLog( );         break;
806
807         case 10:    progTrack -> enableLog( false );  break;
808         case 11:    progTrack -> enableLog( true );   break;
809         case 12:    progTrack -> beginLog( );         break;
810         case 13:    progTrack -> endLog( );           break;
811         case 14:    progTrack -> printLog( );         break;
812
813         case 20: {
814
815             uint8_t buf[ 16 ];
816
817             mainTrack -> getRailComMsg( buf, sizeof( buf ));
818
819             printf( "RC: " );
820             for ( uint8_t i = 0; i < 8; i++ ) printf( "0x%x ", buf[ i ]);
821
822         } break;
823
824         default: ;
825     }
826
827     printf( ">" );
828 }
829
830 //--------------------------------------------------------------------------------------------------------------
831 // "printHelp" lists a short version of all the command.
832 //
833 //--------------------------------------------------------------------------------------------------------------
834 void LcsBaseStationCommand::printHelpCmd( ) {
835
836     printf( "\nCommands:\n" );
837
838     printf( "<O cabId>                              - allocate a session for the cab\n" );
839     printf( "<K sId>                                - release a session\n" );
840     printf( "<t sId cabId speed dir>                - set cab speed / direction\n" );
841     printf( "<f cabId funcId val >                  - set cab function value, group DCC format\n" );
842     printf( "<v sId funcId val >                    - set cab function value, individual\n" );
843     printf( "<R cvId callbacknum callbacksub >      - read CV byte\n" );
844     printf( "<W cvId val callbacknum callbacksub>   - write CV byte on programming track\n" );
845     printf( "<B cvId bitPos bitVal callbacknum callbacksub> - write CV bit on programming track\n" );
846     printf( "<w cabId cvId val > - write CV byte on operations track\n" );
847     printf( "<b cabId cvId bitPos bitVal > - write CV bit on operations track\n" );
848     printf( "<M sId byte1 byte2 [ byte3 ... byte10 ]> - send DCC packet on operations track to Reg n\n" );
849     printf( "<P sId byte1 byte2 [ byte3 ... byte10 ]> - send DCC packet on programming track to Reg n\n" );
850
851     printf( "<C track [option] - set track option, track = 0 -> MAIN, track = 1 -> PROG\n" );
852     printf( "             " " - 1  - set main track cutout on\n" );
853     printf( "             " " - 2  - set main track cutout off\n" );
854     printf( "             " " - 3  - set main track RailCom on\n" );
855     printf( "             " " - 4  - set main track RailCom off\n" );
856     printf( "             " " - 10 - set prog track in operations mode\n" );
857     printf( "             " " - 11 - set prog track in service mode\n" );
858
859     printf( "<X>  - emergency stop all\n" );
860
861     printf( "<0> - turn operations and programming track power off\n" );
862     printf( "<1> - turn operations and programming track power on\n" );
863     printf( "<2> - turn operations track power on\n" );
864     printf( "<3> - turn programming track power on\n" );
865
866     printf( "<a [ opt ]>  " " - list current consumption, default is RMS for MAIN\n" );
867     printf( "              " "   - opt 0  - actual - MAIN\n" );
868     printf( "              " "   - opt 1  - actual - PROG\n" );
869     printf( "              " "   - opt 2  - actual - both\n" );
870     printf( "              " "   - opt 10 - RMS - MAIN\n" );
871     printf( "              " "   - opt 11 - RMS - PROG\n" );
872     printf( "              " "   - opt 12 - RMS - both\n" );
873
874     printf( "<C <option>> - turn on/off the Railcom option on the main track( 0 - off, 1 - on)\n" );
875
876     printf( "<s [ level ]>" " - list status at detail level, default is summary\n" );
877     printf( "              " "   - level 0 - summary\n" );
878     printf( "              " "   - level 1 - configuration\n" );
879     printf( "              " "   - level 2 - session map\n" );
880     printf( "              " "   - level 3 - main track current\n" );
881     printf( "              " "   - level 4 - prog track current\n" );
882     printf( "              " "   - level 9 - all of the above\n" );
883
884     printf( "<S> - list base station configuration\n" );
885     printf( "<L> - list base station session table" );
886
887     printf( "<Y [ opt ]> - DCC log options ( used for debugging and tracing )\n" );
888     printf( "              " "   - 0  - disable main track logging\n" );
889     printf( "              " "   - 1  - enable main track logging\n" );
890     printf( "              " "   - 2  - begin main track logging\n" );
```

```
891     printf( "             " "   - 3  - end main track logging\n" );
892     printf( "             " "   - 4  - print main track logging data\n" );
893     printf( "             " "   - 10 - disable prog track logging\n" );
894     printf( "             " "   - 11 - enable prog track logging\n" );
895     printf( "             " "   - 12 - begin prog track logging\n" );
896     printf( "             " "   - 13 - end prog track logging\n" );
897     printf( "             " "   - 14 - print prog track logging data\n" );
898 
899     printf( "<?> - list this help\n" );
900 
901     printf( "\n" );
902 }
```

```
1  //--------------------------------------------------------------------------------------------------------------
2  //
3  // LCS Base Station - DCC Track - implementation file
4  //
5  //--------------------------------------------------------------------------------------------------------------
6  // The DCC track object is one of the the key objects for the DCC subsystem. It is responsible for the DCC
7  // track signal generation and the power management functions. There will be exactly two objects of this kind,
8  // one for the MAIN track and the other for the PROG track. The DCC track object has two major functional
9  // parts. The first is to transmit a DCC packet to the track. This is the most important task, as with no
10 // packets no power is on the tracks and the locomotive will not work. The second task is to continuously
11 // monitor the current consumption. Finally, for the RailCom option, the cutout generation and receiving
12 // of the RailCOm packets is handled.
13 //
14 //--------------------------------------------------------------------------------------------------------------
15 //
16 // LCS - Base Station DCC Track implementation file
17 // Copyright (C) 2019 - 2024  Helmut Fieres
18 //
19 // This program is free software: you can redistribute it and/or modify it under the terms of the GNU General
20 // Public License as published by the Free Software Foundation, either version 3 of the License, or (at your
21 // option) any later version.
22 //
23 // This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
24 // implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
25 // for more details.
26 //
27 // You should have received a copy of the GNU General Public License along with this program. If not, see
28 // http://www.gnu.org/licenses
29 //
30 //  GNU General Public License:  http://opensource.org/licenses/GPL-3.0
31 //
32 //--------------------------------------------------------------------------------------------------------------
33 #include "LcsBaseStation.h"
34 #include <math.h>
35
36 //--------------------------------------------------------------------------------------------------------------
37 // External global variables.
38 //
39 //--------------------------------------------------------------------------------------------------------------
40 extern uint16_t debugMask;
41
42 //--------------------------------------------------------------------------------------------------------------
43 // DCC Signal debugging. A tick is defined to last 29 microseconds. There is a debugging option to set the
44 // clock much slower so that the waveform can be seen.
45 //
46 // ??? take out, we are past that ..... since a long time. -> one last check than out ...
47 //--------------------------------------------------------------------------------------------------------------
48
49 #define DEBUG_WAVE_FORM 0
50
51 #if DEBUG_WAVE_FORM == 1
52 #define TICK_IN_MICROSECONDS  400000
53 #else
54 #define TICK_IN_MICROSECONDS  29
55 #endif
56
57 //--------------------------------------------------------------------------------------------------------------
58 // The DccTrack Object local definitions. The DCC track object is a bit special. There are exactly two object
59 // instances created, MAIN and PROG. Both however share the global mechanism for generating the DCC hardware
60 // signals. There are callback functions for the DCC timer and the serial I/O capability for the RailCom
61 // feature. The hardware lower layers can be found in controller dependent code (CDC) layer.
62 //
63 //--------------------------------------------------------------------------------------------------------------
64 namespace {
65
66 using namespace LCS;
67
68 //--------------------------------------------------------------------------------------------------------------
69 // The DCC Track will allocate two DCC Track Objects. For the interrupt system to work, references to the
70 // objects must be static variables. The initialization sequence outside of this class will allocate the two
71 // objects and we keep a copy of the respective DCC track object created right here.
72 //
73 // ??? when we use the global variables in the "main" file, can this go away ?
74  //-------------------------------------------------------------------------------------------------------------
75 LcsBaseStationDccTrack  *mainTrack  = nullptr;
76 LcsBaseStationDccTrack  *progTrack  = nullptr;
77
78 //--------------------------------------------------------------------------------------------------------------
79 // DCC packet definitions. A DCC packet payload is at most 15 bytes long, excluding the checksum byte. This
80 // is true for XPOM and DCC-A support, otherwise it is according to NMRA up to 6 bytes. The preamble is a
81 // series of "ONE" bits, which helps the decoders to sync to the bit stream. The standard specifies a
82 // minimum of 16 ONE bits for the MAIN track and 22 ONE bits for the PROG track. The postamble is exactly
83 // one "ONE" bit. If the cutout period option is enabled, the cutout overlays the first ONE bits the
84 // preamble.
85 //
86 //--------------------------------------------------------------------------------------------------------------
87 const uint8_t   MAIN_PACKET_PREAMBLE_LEN    = 17;
88 const uint8_t   MAIN_PACKET_POSTAMBLE_LEN   = 1;
89 const uint8_t   PROG_PACKET_PREAMBLE_LEN    = 22;
90 const uint8_t   PROG_PACKET_POSTAMBLE_LEN   = 1;
91 const uint8_t   DCC_PACKET_CUTOUT_LEN       = 4;
92 const uint8_t   MIN_DCC_PACKET_SIZE         = 2;
93 const uint8_t   MAX_DCC_PACKET_SIZE         = 16;
94 const uint8_t   MIN_DCC_PACKET_REPEATS      = 0;
95 const uint8_t   MAX_DCC_PACKET_REPEATS      = 8;
96 const uint8_t   RAILCOM_BUFFER_SIZE         = 8;
97
98 //--------------------------------------------------------------------------------------------------------------
```

```
99  // Constant values definition. We need the RESET and IDLE packet as well as a bit mask for a quick bit
100 // select in the data byte.
101 //
102 //-----------------------------------------------------------------------------------------------------------
103 DccPacket       idleDccPacket       = { 3, 0, { 0xFF, 0x00, 0xFF }};
104 DccPacket       resetDccPacket      = { 3, 0, { 0x00, 0x00, 0x00 }};
105 const uint8_t   bitMask9[ ]         = { 0x00, 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01 };
106
107 //-----------------------------------------------------------------------------------------------------------
108 // Programming decoders require to detect a short rise in power consumption. The value is at least 60mA,
109 // but decoders can raise anything from 100mA to 250mA. This is a bit touchy and the value set to 100mA
110 // was done after testing several decoders. Still, a bit flaky ...
111 //
112 //-----------------------------------------------------------------------------------------------------------
113 const uint8_t ACK_TRESHOLD_VAL      = 100;
114
115 //-----------------------------------------------------------------------------------------------------------
116 // The DCC signal generator thinks in ticks. With a DCC ONE based on 58 microseconds and a DCC ZERO based
117 // on 116 microseconds half period, we define a tick as a 29 microsecond interval. Although, ONE and ZERO
118 // bit signals could be implemented using a multiple of 58 microseconds, the cutout function requires a
119 // signal length of 29 microseconds at the beginning of the period, right after the packet end bit of the
120 // previous packet. Luckily 2 * 29 is 58, 2 * 58 is 116. Perfect for DCC packets.
121 //
122 // ??? think directly in microseconds ?
123 //-----------------------------------------------------------------------------------------------------------
124 const uint32_t TICKS_29_MICROS              =  1;
125 const uint32_t TICKS_58_MICROS              =  TICKS_29_MICROS * 2;
126 const uint32_t TICKS_116_MICROS             =  TICKS_29_MICROS * 4;
127 const uint32_t TICKS_CUTOUT_MICROS          =  TICKS_29_MICROS * 16;
128
129 //-----------------------------------------------------------------------------------------------------------
130 // Base Station global limits. Perhaps to move to a configurable place...
131 //
132 //-----------------------------------------------------------------------------------------------------------
133 const uint16_t MILLI_VOLT_PER_DIGIT         = 5;
134 const uint16_t MILLI_VOLT_PER_AMP           = 1500;
135
136 //-----------------------------------------------------------------------------------------------------------
137 // DCC track power management is also a a state machine managing the state of the power track. Maximum values
138 // for the DCC track power start and stop sequence as well as limits for power overload events are defined.
139 // We also define reasonable default values.
140 //
141 //-----------------------------------------------------------------------------------------------------------
142 const uint16_t MAX_START_TIME_THRESHOLD_MILLIS      = 2000;
143 const uint16_t MAX_STOP_TIME_THRESHOLD_MILLIS       = 1000;
144 const uint16_t MAX_OVERLOAD_TIME_THRESHOLD_MILLIS   = 500;
145 const uint16_t MAX_OVERLOAD_EVENT_COUNT             = 10;
146 const uint16_t MAX_OVERLOAD_RESTART_COUNT           = 10;
147
148 const uint16_t DEF_START_TIME_THRESHOLD_MILLIS      = 1000;
149 const uint16_t DEF_STOP_TIME_THRESHOLD_MILLIS       = 500;
150 const uint16_t DEF_OVERLOAD_TIME_THRESHOLD_MILLIS   = 300;
151 const uint16_t DEF_OVERLOAD_EVENT_COUNT             = 10;
152 const uint16_t DEF_OVERLOAD_RESTART_COUNT           = 10;
153
154 //-----------------------------------------------------------------------------------------------------------
155 // Track state machine state definitions. See the track state machine routine for an explanation of the
156 // individual states.
157 //
158 //-----------------------------------------------------------------------------------------------------------
159 enum DccTrackState : uint8_t {
160
161     DCC_TRACK_POWER_OFF       = 0,
162     DCC_TRACK_POWER_ON        = 1,
163     DCC_TRACK_POWER_OVERLOAD  = 2,
164     DCC_TRACK_POWER_START1    = 3,
165     DCC_TRACK_POWER_START2    = 4,
166     DCC_TRACK_POWER_STOP1     = 5,
167     DCC_TRACK_POWER_STOP2     = 6
168 };
169
170 //-----------------------------------------------------------------------------------------------------------
171 // DCC Track signal state machine states. See the DCC signal state machine routine for an explanation of
172 // the states.
173 //
174 //-----------------------------------------------------------------------------------------------------------
175 enum DccSignalState : uint8_t {
176
177     DCC_SIG_CUTOUT_START      = 0,
178     DCC_SIG_CUTOUT_1          = 1,
179     DCC_SIG_CUTOUT_2          = 2,
180     DCC_SIG_CUTOUT_3          = 3,
181     DCC_SIG_CUTOUT_END        = 4,
182     DCC_SIG_START_BIT         = 5,
183     DCC_SIG_TEST_BIT          = 6,
184     DCC_SIG_ZERO_SECOND_HALF  = 7
185 };
186
187 // ??? idea: each state has a number of ticks it will set. Have an array where to get this value and just
188 // set it from the table...
189 //
190 uint8_t ticksForState[ ] = {
191
192     TICKS_29_MICROS,        // DCC_SIG_CUTOUT_START
193     TICKS_CUTOUT_MICROS,    // DCC_SIG_CUTOUT_1
194     TICKS_29_MICROS,        // DCC_SIG_CUTOUT_2
195     TICKS_58_MICROS,        // DCC_SIG_CUTOUT_3
196     TICKS_58_MICROS,        // DCC_SIG_CUTOUT_END
197     TICKS_58_MICROS,        // DCC_SIG_START_BIT
```

```
198     TICKS_58_MICROS,        // DCC_TEST_BIT,
199     TICKS_116_MICROS        // DCC_SIG_ZERO_SECOND_HALF
200 };
201 
202 //-----------------------------------------------------------------------------------------------------------------
203 // DCC Track signal state machine follow up request items. The signal state machine first sets the hardware
204 // signal for both tracks and then determines whether a follow up action is required. See the track state
205 // machine routine for an explanation of the individual follow up actions.
206 //
207 //-----------------------------------------------------------------------------------------------------------------
208 enum DccSignalStateFollowup : uint8_t {
209 
210     DCC_SIG_FOLLOW_UP_NONE                = 0,
211     DCC_SIG_FOLLOW_UP_GET_BIT             = 1,
212     DCC_SIG_FOLLOW_UP_GET_PACKET          = 2,
213     DCC_SIG_FOLLOW_UP_MEASURE_CURRENT     = 3,
214     DCC_SIG_FOLLOW_UP_START_RAILCOM_IO    = 4,
215     DCC_SIG_FOLLOW_UP_STOP_RAILCOM_IO     = 5,
216     DCC_SIG_FOLLOW_UP_RAILCOM_MSG         = 6,
217 };
218 
219 //-----------------------------------------------------------------------------------------------------------------
220 // The hardware timer needs to be set to the ticks we want to pass before interrupting again. There are
221 // three things to remember between interrupts. First, the current time interval, which tells us how many
222 // ticks will have passed when the timer interrupts again. Next, for each DCC track signal state we need to
223 // remember how many ticks are left before the state machine needs to run again. Each time the timer will
224 // interrupt, the passed ticks are subtracted from the ticks left counters. When the counter becomes zero,
225 // the state machine for the track will run.
226 //
227 //-----------------------------------------------------------------------------------------------------------------
228 volatile uint8_t timeToInterrupt    = 0;
229 volatile uint8_t timeLeftMainTrack  = 0;
230 volatile uint8_t timeLeftProgTrack  = 0;
231 
232 //-----------------------------------------------------------------------------------------------------------------
233 // The DCC track object maintains an internal log facility for test and debugging purposes. During operation
234 // a set of log entries can be recorded to a log buffer. A log entry consist of the header byte, which
235 // contains in the first byte the 4-bit log id and the 4-bit length of the log data. A log entry can therefore
236 // record up to 16 bytes of payload.
237 //
238 //-----------------------------------------------------------------------------------------------------------------
239 enum LogId : uint8_t {
240 
241     LOG_NIL       = 0,
242     LOG_BEGIN     = 1,
243     LOG_END       = 2,
244     LOG_TSTAMP    = 3,
245     LOG_DCC_IDL   = 4,
246     LOG_DCC_RST   = 5,
247     LOG_DCC_PKT   = 6,
248     LOG_DCC_RCM   = 7,
249     LOG_VAL       = 8,
250     LOG_INV       = 15
251 };
252 
253 //-----------------------------------------------------------------------------------------------------------------
254 // The log buffer and the log index. When writing to the log buffer, the index will always point to the
255 // next available position. Once the buffer is full, no further data can be added.
256 //
257 //-----------------------------------------------------------------------------------------------------------------
258 const uint16_t  LOG_BUF_SIZE            = 4096;
259 
260 bool            logEnabled              = false;
261 bool            logActive               = false;
262 uint16_t        logBufIndex             = 0;
263 uint8_t         logBuf[ LOG_BUF_SIZE ]  = { 0 };
264 
265 //-----------------------------------------------------------------------------------------------------------------
266 // RailCom decoder table. The Railcom communication will send raw bytes where only four bits are "one" in
267 // a byte ( hamming weight 4 ). The first two bytes are labelled "channel1" and the remaining six bytes
268 // are labelled "channel2". The actual data is then encode using the table below. Each raw byte will be
269 // translated to a 6 bits of data for the datagram to assemble. In total there are therefore a maximum
270 // of 48bits that are transmitted in a railcom message.
271 //
272 //-----------------------------------------------------------------------------------------------------------------
273 enum RailComDataBytes : uint8_t {
274 
275     INV   = 0xff,
276     BUSY  = 0xfe,
277     ACK   = 0xfd,
278     NACK  = 0xfc,
279     RSV1  = 0xfa,
280     RSV2  = 0xf9,
281     RSV3  = 0xf8
282 };
283 
284 const uint8_t railComDecode[256] = {
285 
286     INV,    INV,    INV,    INV,    INV,    INV,    INV,    INV,    // 0
287     INV,    INV,    INV,    INV,    INV,    INV,    INV,    ACK,
288 
289     INV,    INV,    INV,    INV,    INV,    INV,    INV,    0x33,   // 1
290     INV,    INV,    INV,    0x34,   INV,    0x35,   0x36,   INV,
291 
292     INV,    INV,    INV,    INV,    INV,    INV,    INV,    0x3A,   // 2
293     INV,    INV,    INV,    0x3B,   INV,    0x3C,   0x37,   INV,
294 
295     INV,    INV,    INV,    0x3F,   INV,    0x3D,   0x38,   INV,    // 3
296     INV,    0x3E,   0x39,   INV,    NACK,   INV,    INV,    INV,
```

```
297
298     INV,    INV,    INV,    INV,    INV,    INV,    INV,    0x24,   // 4
299     INV,    INV,    INV,    0x23,   INV,    0x22,   0x21,   INV,
300
301     INV,    INV,    INV,    0x1F,   INV,    0x1E,   0x20,   INV,    // 5
302     INV,    0x1D,   0x1C,   INV,    0x1B,   INV,    INV,    INV,
303
304     INV,    INV,    INV,    0x19,   INV,    0x18,   0x1A,   INV,    // 6
305     INV,    0x17,   0x16,   INV,    0x15,   INV,    INV,    INV,
306
307     INV,    0x25,   0x14,   INV,    0x13,   INV,    INV,    INV,    // 7
308     0x32,   INV,    INV,    INV,    INV,    INV,    INV,    INV,
309
310     INV,    INV,    INV,    INV,    INV,    INV,    INV,    RSV2,   // 8
311     INV,    INV,    INV,    0x0E,   INV,    0x0D,   0x0C,   INV,
312
313     INV,    INV,    INV,    0x0A,   INV,    0x09,   0x0B,   INV,    // 9
314     INV,    0x08,   0x07,   INV,    0x06,   INV,    INV,    INV,
315
316     INV,    INV,    INV,    0x04,   INV,    0x03,   0x05,   INV,    // a
317     INV,    0x02,   0x01,   INV,    0x00,   INV,    INV,    INV,
318
319     INV,    0x0F,   0x10,   INV,    0x11,   INV,    INV,    INV,    // b
320     0x12,   INV,    INV,    INV,    INV,    INV,    INV,    INV,
321
322     INV,    INV,    INV,    RSV1,   INV,    0x2B,   0x30,   INV,    // c
323     INV,    0x2A,   0x2F,   INV,    0x31,   INV,    INV,    INV,
324
325     INV,    0x29,   0x2E,   INV,    0x2D,   INV,    INV,    INV,    // d
326     0x2C,   INV,    INV,    INV,    INV,    INV,    INV,    INV,
327
328     INV,    RSV3,   0x28,   INV,    0x27,   INV,    INV,    INV,    // e
329     0x26,   INV,    INV,    INV,    INV,    INV,    INV,    INV,
330
331     ACK,    INV,    INV,    INV,    INV,    INV,    INV,    INV,    // f
332     INV,    INV,    INV,    INV,    INV,    INV,    INV,    INV,
333 };
334
335 //------------------------------------------------------------------------------------------------------------------
336 // Railcom datagrams are sent from a mobile or a stationary decoder.
337 //
338 //------------------------------------------------------------------------------------------------------------------
339 enum railComDatagramType : uint8_t {
340
341     RX_DG_TYPE_UNDEFINED  = 0,
342     RC_DG_TYPE_MOB        = 1,
343     RC_DG_TYPE_STAT       = 2
344 };
345
346 //------------------------------------------------------------------------------------------------------------------
347 // Each mobile decoder railcom datagram will start with an ID field of four bits. Channel one will use only
348 // the ADR_HIG and ADR_LOW Ids. All IDs can be used for channel 2. Since decoders answer on channel one
349 // for each DCC packet they receive, here is a good chance that channel 1 will contains nonsense data. This
350 // is different for channel two, where only the addressed decoder explicitly answers. To decide whether
351 // a railcom message is valid, you should perhaps ignore channel 1 data and just check channel 2 for this
352 // purpose. A RC datagram starts with the 4-bit ID and an 8 to 32bit payload.
353 //
354 //      RC_DG_MOB_ID_POM       ( 0  )  - 12bit
355 //      RC_DG_MOB_ID_ADR_HIGH  ( 1  )  - 12bit
356 //      RC_DG_MOB_ID_ADR_LOW   ( 2  )  - 12bit
357 //      RC_DG_MOB_ID_APP_EXT   ( 3  )  - 18bit
358 //      RC_DG_MOB_ID_APP_DYN   ( 7  )  - 18bit
359 //      RC_DG_MOB_ID_XPOM_1    ( 8  )  - 36bit
360 //      RC_DG_MOB_ID_XPOM_2    ( 9  )  - 36bit
361 //      RC_DG_MOB_ID_XPOM_3    ( 10 )  - 36bit
362 //      RC_DG_MOB_ID_XPOM_4    ( 11 )  - 36bit
363 //      RC_DG_MOB_ID_TEST      ( 12 )  - ignore
364 //      RC_DG_MOB_ID_SEARCH    ( 14 )  - 48bit
365 //
366 // A datagram with the ID 14 is a DDC-A datagram and all 8 datagram bytes are combined to an 48bit datagram.
367 // A datagram packet can also contain more than one datagram. For example there could be two 18-bit length
368 // datagram in one packet or 3 12-bit packets and so on. Finally, unused bytes in channel two could contain
369 // an ACK to fill them up.
370 //
371 //------------------------------------------------------------------------------------------------------------------
372 enum railComDatagramMobId : uint8_t {
373
374     RC_DG_MOB_ID_POM        = 0,
375     RC_DG_MOB_ID_ADR_HIGH   = 1,
376     RC_DG_MOB_ID_ADR_LOW    = 2,
377     RC_DG_MOB_ID_APP_EXT    = 3,
378     RC_DG_MOB__IDAPP_DYN    = 7,
379     RC_DG_MOB_ID_XPOM_1     = 8,
380     RC_DG_MOB_ID_XPOM_2     = 9,
381     RC_DG_MOB_ID_XPOM_3     = 10,
382     RC_DG_MOB_ID_XPOM_4     = 11,
383     RC_DG_MOB_ID_TEST       = 12,
384     RC_DG_MOB_ID_SEARCH     = 14
385 };
386
387 //------------------------------------------------------------------------------------------------------------------
388 // Similar to the mobile decode, a stationary decoder datagram will start an ID field of four bits. Stationary
389 // decoders also define a datagram with "SRQ" and no ID field to request service from the base station.
390 //
391 // ??? to fill in ...
392 //
393 //      RC_DG_STAT_ID_SRQ       ( 0  )  - 12bit
394 //      RC_DG_STAT_ID_POM       ( 1  )  - 12bit
395 //      RC_DG_STAT_ID_STAT1     ( 4  )  - 12bit
```

```
396 //      RC_DG_STAT_ID_TIME      ( 5  )  - xxbit
397 //      RC_DG_STAT_ID_ERR       ( 6  )  - xxbit
398 //      RC_DG_STAT_ID_XPOM_1    ( 8  )  - 36bit
399 //      RC_DG_STAT_ID_XPOM_2    ( 9  )  - 36bit
400 //      RC_DG_STAT_ID_XPOM_3    ( 10 )  - 36bit
401 //      RC_DG_STAT_ID_XPOM_4    ( 11 )  - 36bit
402 //      RC_DG_STAT_ID_TEST      ( 12 )  - ignore
403 //
404 //-----------------------------------------------------------------------------------------------------------
405 enum railComDatagramStatId : uint8_t {
406
407     RC_DG_STAT_ID_SRQ       = 0,
408     RC_DG_STAT_ID_POM       = 1,
409     RC_DG_STAT_ID_STAT1     = 4,
410     RC_DG_STAT_ID_TIME      = 5,
411     RC_DG_STAT_ID_ERR       = 6,
412     RC_DG_STAT_ID_DYN       = 7,
413     RC_DG_STAT_ID_XPOM_1    = 8,
414     RC_DG_STAT_ID_XPOM_2    = 9,
415     RC_DG_STAT_ID_XPOM_3    = 10,
416     RC_DG_STAT_ID_XPOM_4    = 11,
417     RC_DG_STAT_ID_TEST      = 12
418 };
419
420 //-----------------------------------------------------------------------------------------------------------
421 // Utility routine for number range checks.
422 //
423 //-----------------------------------------------------------------------------------------------------------
424 bool isInRangeU( uint8_t val, uint8_t lower, uint8_t upper ) {
425
426     return (( val >= lower ) && ( val <= upper ));
427 }
428
429 //-----------------------------------------------------------------------------------------------------------
430 // Utility function to map a DCC address to a railcom decoder type.
431 //
432 //-----------------------------------------------------------------------------------------------------------
433 inline uint8_t mapDccAdrToRailComDatagramType( uint16_t adr ) {
434
435     if      (( adr >= 1 )   && ( adr <= 127 ))  return ( RC_DG_TYPE_MOB );
436     else if (( adr >= 128 ) && ( adr <= 191 ))  return ( RC_DG_TYPE_STAT );
437     else if (( adr >= 192 ) && ( adr <= 231 ))  return ( RC_DG_TYPE_MOB );
438     else                                        return ( RX_DG_TYPE_UNDEFINED );
439 }
440
441 //-----------------------------------------------------------------------------------------------------------
442 // Conversion functions between milliAmps and digit values as report4de by the analog to digital converter
443 // hardware. For a better precision, the formula uses 32 bit computation and stores the result back in a
444 // 16 bit quantity.
445 //
446 //-----------------------------------------------------------------------------------------------------------
447 uint16_t milliAmpToDigitValue( uint16_t milliAmp, uint16_t digitsPerAmp ) {
448
449     #if 0
450     uint32_t mA = milliAmp;
451     uint32_t dPA = digitsPerAmp;
452     return (( uint16_t ) ( mA * dPA / 1000 ));
453     #endif
454
455     return ((uint16_t) ((((uint32_t) milliAmp ) * ((uint32_t) digitsPerAmp )) / 1000 ));
456 }
457
458 uint16_t digitValueToMilliAmp( uint16_t digitValue, uint16_t digitsPerAmp ) {
459
460     #if 0
461     uint32_t dV = digitValue;
462     uint32_t dPA = digitsPerAmp;
463     return ((uint16_t)( dV * 1000 / dPA ));
464     #endif
465
466     return ((uint16_t) ((((uint32_t) digitValue ) * 1000 ) / ((uint32_t) digitsPerAmp )));
467 }
468
469 //-----------------------------------------------------------------------------------------------------------
470 // The DccTrack timer interrupt handler routine implements the heartbeat of the DCC system. The two DCC
471 // track signal generators state machines MAIN and PROG use the same timer interrupt handler. Upon the timer
472 // interrupt, we first will update the time left counters. If a counter falls to zero, the signal state
473 // machine for that track will run and set the DCC signal levels. The state machine returns the next time
474 // interval it expects to be called again and a possible follow up action code. After handling both state
475 // machines, the timer is set to the smaller new remaining minimum time interval of both state machines.
476 // This is the time when the next state machine in one of the  signal generators needs to run. It is
477 // important to always have the timer running, so we keep decrementing the ticks to interrupt values.
478 //
479 // If a state machine determined that it needs to do some more elaborate action, the interrupt handler runs
480 // part two of its work. This split allows to run the time sensitive signal level settings first and any
481 // actions, such as getting the next packet, after both signal generator signal settings have been processed.
482 // Follow up actions are getting the next bit value to transmit, the next packet to send, a power consumption
483 // measurement and Railcom message processing. As we do not have all time in the world, these follow up
484 // actions still should be brief. The state machine carefully selects the spot for requesting such follow up
485 // actions in the DCC bit stream.
486 //
487 // The timer interrupt routine and all it calls runs with interrupts disabled. As said, better be quick.
488 // Top priority is to fetch the next bit and the next packet. Next is the Railcom processing if enabled. If
489 // there are power consumption measurement follow up actions, they are run last. Since the ADC converter
490 // hardware serializes the analog measurements, we will only do one measurement and drop the other. MAIN
491 // always has the higher priority.
492 //
493 // For the MAIN track with cutout enabled, the entry and exit of that cutout is a 29us timer call. That is
494 // awfully short and no follow-up action is scheduled there. All other intervals are either 58us or 116us
```

```
// or even longer for the cutout itself and give us some more room.
//
// ??? we could use timerVal, but this is in microseconds, not ticks. Convert one day...
//-----------------------------------------------------------------------------------------------------
void timerCallback( uint32_t timerVal ) {

    uint8_t followUpMain = DCC_SIG_FOLLOW_UP_NONE;
    uint8_t followUpProg = DCC_SIG_FOLLOW_UP_NONE;

    timeLeftMainTrack -= timeToInterrupt;
    timeLeftProgTrack -= timeToInterrupt;

    if ( timeLeftMainTrack == 0 ) mainTrack -> runDccSignalStateMachine( &timeLeftMainTrack, &followUpMain );
    if ( timeLeftProgTrack == 0 ) progTrack -> runDccSignalStateMachine( &timeLeftProgTrack, &followUpProg );

    // take out after test ...
    // timeToInterrupt = min( timeLeftMainTrack, timeLeftProgTrack );

    timeToInterrupt = (( timeLeftMainTrack < timeLeftProgTrack ) ? timeLeftMainTrack : timeLeftProgTrack );

    CDC::setRepeatingTimerLimit( timeToInterrupt * TICK_IN_MICROSECONDS );

    if (( followUpMain != DCC_SIG_FOLLOW_UP_NONE ) && ( followUpMain != DCC_SIG_FOLLOW_UP_MEASURE_CURRENT )) {

        if      ( followUpMain == DCC_SIG_FOLLOW_UP_GET_BIT )           mainTrack -> getNextBit( );
        else if ( followUpMain == DCC_SIG_FOLLOW_UP_GET_PACKET )        mainTrack -> getNextPacket( );
        else if ( followUpMain == DCC_SIG_FOLLOW_UP_START_RAILCOM_IO )  mainTrack -> startRailComIO( );
        else if ( followUpMain == DCC_SIG_FOLLOW_UP_STOP_RAILCOM_IO )   mainTrack -> stopRailComIO( );
        else if ( followUpMain == DCC_SIG_FOLLOW_UP_RAILCOM_MSG )       mainTrack -> handleRailComMsg( );
    }

    if (( followUpProg != DCC_SIG_FOLLOW_UP_NONE ) && ( followUpProg != DCC_SIG_FOLLOW_UP_MEASURE_CURRENT )) {

        if      ( followUpProg == DCC_SIG_FOLLOW_UP_GET_BIT )     progTrack -> getNextBit( );
        else if ( followUpProg == DCC_SIG_FOLLOW_UP_GET_PACKET )  progTrack -> getNextPacket( );
    }

    if      ( followUpMain == DCC_SIG_FOLLOW_UP_MEASURE_CURRENT ) mainTrack -> powerMeasurement( );
    else if ( followUpProg == DCC_SIG_FOLLOW_UP_MEASURE_CURRENT ) progTrack -> powerMeasurement( );

} // timerCallback

//-----------------------------------------------------------------------------------------------------
// When all DCC track objects are initialized, the last thing to do before operation is to  start the timer
// heartbeat. We start b firing up the timer with a first short delay, so when it expires the timer routine
// will be called. The current time tick of zero and no ticks left, so the state machine for the signals
// will run.
//
//-----------------------------------------------------------------------------------------------------
void initDccTrackProcessing( ) {

    timeToInterrupt    = 0;
    timeLeftMainTrack  = 0;
    timeLeftProgTrack  = 0;

    CDC::startRepeatingTimer( TICK_IN_MICROSECONDS );
}

//-----------------------------------------------------------------------------------------------------
// DCC log functions for printing the DCC log buffer. The fist byte of each log entry has encoded the log
// entry type and the entry length. Depending on the log entry type, data is displayed as just the header,
// a numeric 16-bit value, a numeric 32-bit vale or as an array of data bytes. We return the length of the
// DCC log entry.
//
//-----------------------------------------------------------------------------------------------------
void printLogTimeStamp( uint16_t index ) {

    uint32_t ts = logBuf[ index ];
    ts = ( ts << 8 ) | logBuf[ index + 1 ];
    ts = ( ts << 8 ) | logBuf[ index + 2 ];
    ts = ( ts << 8 ) | logBuf[ index + 3 ];
    printf( "0x%x", ts );
}

void printLogVal( uint16_t index ) {

    uint16_t val = logBuf[ index ] << 8 | logBuf[ index + 1 ];
    printf( "0x%04x", val );
}

void printLogData( uint16_t index, uint8_t len ) {

    for ( int i = 0; i < len; i++ ) printf( "0x%02x ", logBuf[ index + i ] );
}

uint8_t printLogEntry( uint16_t index ) {

    if ( index < LOG_BUF_SIZE ) {

        uint8_t logEntryId  = logBuf[ index ] >> 4;
        uint8_t logEntryLen = logBuf[ index ] & 0x0F;

        switch ( logEntryId ) {

            case LOG_NIL:      printf( "NIL        " ); break;
            case LOG_BEGIN:    printf( "BEGIN      " ); break;
            case LOG_END:      printf( "END        " ); break;
            case LOG_TSTAMP:   printf( "TSTAMP     " ); break;
            case LOG_DCC_IDL:  printf( "DCC_IDLE   " ); break;
```

```
            case LOG_DCC_RST:  printf( "DCC_RESET  " ); break;
            case LOG_DCC_PKT:  printf( "DCC_PKT    " ); break;
            case LOG_DCC_RCM:  printf( "DCC_RCOM   " ); break;
            case LOG_VAL:      printf( "VAL        " ); break;
            default:           printf( "INVALID ( 0x%02 )", logBuf[ index ] >> 4 );
        }

        if      ( logEntryId == LOG_TSTAMP  )  printLogTimeStamp( index + 1 );
        else if ( logEntryId == LOG_VAL     )  printLogVal( index + 1 );
        else                                   printLogData( index + 1, logEntryLen );

        return ( logEntryLen + 1 );
    }
    else return ( 0 );
}

//-----------------------------------------------------------------------------------------------------------
// There are a couple of routines to write the log data. For convenience, some of the log entry types are
// available as a direct call. The order of data entry for numeric types is big endian, i.e. most significant
// byte first.
//
//-----------------------------------------------------------------------------------------------------------
void writeLogData( uint8_t id, uint8_t *buf, uint8_t len ) {

    if ( logActive ) {

        len = len % 16;
        if ( logBufIndex + len + 1 < LOG_BUF_SIZE ) {

            logBuf[ logBufIndex ++ ] = ( id << 4 ) | len;
            for ( uint8_t i = 0; i < len; i++ ) logBuf[ logBufIndex ++ ] = buf[ i ];
        }
    }
}

void writeLogId( uint8_t id ) {

    if ( logActive ) logBuf[ logBufIndex ++ ] = ( id << 4 ) | 1;
}

void writeLogTs( ) {

    if ( logActive ) {

        uint32_t ts = CDC::getMicros( );
        logBuf[ logBufIndex ++ ] = ( LOG_TSTAMP << 4 ) | 4;
        logBuf[ logBufIndex ++ ] = ( ts >> 24 ) & 0xFF;
        logBuf[ logBufIndex ++ ] = ( ts >> 16 ) & 0xFF;
        logBuf[ logBufIndex ++ ] = ( ts >> 8  ) & 0xFF;
        logBuf[ logBufIndex ++ ] = ( ts >> 0  ) & 0xFF;
    }
}

void writeLogVal( uint8_t valId, uint16_t val ) {

    if ( logActive ) {

        logBuf[ logBufIndex ++ ] = ( LOG_VAL << 4 ) | 3;
        logBuf[ logBufIndex ++ ] = valId;
        logBuf[ logBufIndex ++ ] = val >> 8;
        logBuf[ logBufIndex ++ ] = val & 0xFF;
    }
}

//-----------------------------------------------------------------------------------------------------------
// The log management routines. A typical transaction to log would start the logging process and then end
// it after the operation to analyze/debug. The "enableLog" call should be used to enable the logging
// process all together, the other calls will only do work when the log is enabled. With this call the
// recording process could be controlled from a command line setting or so.
//
//-----------------------------------------------------------------------------------------------------------
void enableLog( bool arg ) {

    logEnabled = arg;
    logActive  = false;
}

void beginLog( ) {

    if ( logEnabled ) {

        logActive   = true;
        logBufIndex = 0;
        writeLogId( LOG_BEGIN );
        writeLogTs( );
    }
}

void endLog( ) {

    if ( logActive ) {

        writeLogTs( );
        writeLogId( LOG_END );
        logActive = false;
    }
}

//-----------------------------------------------------------------------------------------------------------
```

```
693 // A simple routine to print out the log data, one entry on one line.
694 //
695 // ??? what is exactly the stop condition ? The END entry having a length of zero ?
696 //-------------------------------------------------------------------------------------------------------
697 void printLog( ) {
698
699     if ( logEnabled ) {
700
701         if ( ! logActive ) {
702
703             if ( logBufIndex > 0 ) {
704
705                 printf( "\n" );
706
707                 uint16_t entryIndex  = 0;
708                 uint8_t  entryLen    = 0;
709
710                 while ( entryIndex < logBufIndex ) {
711
712                     entryLen = printLogEntry( entryIndex );
713                     printf( "\n" );
714
715                     if ( entryLen > 0 ) entryIndex += entryLen;
716                     else                break;
717                 }
718             }
719             else printf( "DCC Log Buf: Nothing recorded\n" );
720         }
721         else printf( "DCC Log Active\n" );
722     }
723     else printf( "DCC Log disabled\n" );
724 }
725
726 }; // namespace
727
728
729 //=======================================================================================================
730 //=======================================================================================================
731 //
732 // Object part.
733 //
734 //=======================================================================================================
735 //=======================================================================================================
736
737
738 //-------------------------------------------------------------------------------------------------------
739 // "startDccProcessing" will kick off the DCC timer for the track signal processing. The idea is that the
740 // program first creates all the DCC track objects, does whatever else needs to be initialized and then starts
741 // the signal generation with this routine.
742 //
743 //-------------------------------------------------------------------------------------------------------
744 void LcsBaseStationDccTrack::startDccProcessing( ) {
745
746     initDccTrackProcessing( );
747 }
748
749 //-------------------------------------------------------------------------------------------------------
750 // Object instance section. The DccTrack constructor. Nothing to do so far.
751 //
752 //-------------------------------------------------------------------------------------------------------
753 LcsBaseStationDccTrack::LcsBaseStationDccTrack( ) { }
754
755 //-------------------------------------------------------------------------------------------------------
756 // "setupDccTrack" performs the setup tasks for the DCC track.  We will configure the hardware, the DCC
757 // packet options such as preamble and postamble length, the initial state machine state current consumption
758 // limit and load the initial packet into the active buffer. There is quite a list of parameters and options
759 // that can be set. This routine does the following checking:
760 //
761 //    - the pins used in the CDC layer must be a pair ( for atmega controllers ).
762 //    - the sensePin must be an analog input pin.
763 //    - if the track is a service track, cutout and RailCom are not supported.
764 //    - if RailCom is set, Cutout must be set too.
765 //    - the initial current limit consumption setting must be less than the current limit setting.
766 //    - the current limit setting must be less than the maximum current limit setting.
767 //
768 // Once the DCC track object is initialized, the last thing to do is to remember the object instance in the
769 // file static variables. This is necessary for the interrupt handlers to work. If any of the checks fails,
770 // the flag field will have the error bit set.
771 //
772 //-------------------------------------------------------------------------------------------------------
773 uint8_t LcsBaseStationDccTrack::setupDccTrack( LcsBaseStationTrackDesc* trackDesc ) {
774
775     if ((  trackDesc -> enablePin  == CDC::UNDEFINED_PIN ) ||
776         (  trackDesc -> dccSigPin1 == CDC::UNDEFINED_PIN ) ||
777         (  trackDesc -> dccSigPin2 == CDC::UNDEFINED_PIN ) ||
778         (  trackDesc -> sensePin   == CDC::UNDEFINED_PIN )) {
779
780         flags = DT_F_CONFIG_ERROR;
781         return ( ERR_DCC_PIN_CONFIG );
782     }
783
784     if ((( trackDesc -> options & DT_OPT_SERVICE_MODE_TRACK ) && ( trackDesc -> options & DT_OPT_CUTOUT ))    ||
785         (( trackDesc -> options & DT_OPT_SERVICE_MODE_TRACK ) && ( trackDesc -> options & DT_OPT_RAILCOM ))   ||
786         (( trackDesc -> options & DT_OPT_RAILCOM ) && ( ! ( trackDesc -> options & DT_OPT_CUTOUT )))          ||
787         ( trackDesc -> initCurrentMilliAmp  > trackDesc -> limitCurrentMilliAmp )                             ||
788         ( trackDesc -> limitCurrentMilliAmp > trackDesc -> maxCurrentMilliAmp )                               ||
789         ( trackDesc -> startTimeThresholdMillis > MAX_START_TIME_THRESHOLD_MILLIS )                           ||
790         ( trackDesc -> stopTimeThresholdMillis > MAX_STOP_TIME_THRESHOLD_MILLIS )                             ||
791         ( trackDesc -> overloadTimeThresholdMillis > MAX_OVERLOAD_TIME_THRESHOLD_MILLIS )                     ||
```

```
792         ( trackDesc -> overloadEventThreshold > MAX_OVERLOAD_EVENT_COUNT )                                  ||
793         ( trackDesc -> overloadRestartThreshold > MAX_OVERLOAD_RESTART_COUNT )
794         ) {
795
796         flags = DT_F_CONFIG_ERROR;
797         return ( ERR_DCC_TRACK_CONFIG );
798     }
799
800     signalState               = DCC_SIG_START_BIT;
801     trackState                = DCC_TRACK_POWER_OFF;
802     flags                     = DT_F_DEFAULT_SETTING;
803     options                   = trackDesc -> options;
804     enablePin                 = trackDesc -> enablePin;
805     dccSigPin1                = trackDesc -> dccSigPin1;
806     dccSigPin2                = trackDesc -> dccSigPin2;
807     sensePin                  = trackDesc -> sensePin;
808     uartRxPin                 = trackDesc -> uartRxPin;
809     initCurrentMilliAmp       = trackDesc -> initCurrentMilliAmp;
810     limitCurrentMilliAmp      = trackDesc -> limitCurrentMilliAmp;
811     maxCurrentMilliAmp        = trackDesc -> maxCurrentMilliAmp;
812     startTimeThreshold        = trackDesc -> startTimeThresholdMillis;
813     stopTimeThreshold         = trackDesc -> stopTimeThresholdMillis;
814     overloadTimeThreshold     = trackDesc -> overloadTimeThresholdMillis;
815     overloadEventThreshold    = trackDesc -> overloadEventThreshold;
816     overloadRestartThreshold  = trackDesc -> overloadRestartThreshold;
817
818     // ??? MILLI_VOLT_PER_DIGIT is actually 4,72V / 1024 = 4,6 mV. How to make this more precise ?
819
820     milliVoltPerAmp           = trackDesc -> milliVoltPerAmp;
821     digitsPerAmp              = milliVoltPerAmp / MILLI_VOLT_PER_DIGIT;
822
823     limitCurrentDigitValue    = milliAmpToDigitValue( initCurrentMilliAmp, digitsPerAmp );
824     ackThresholdDigitValue    = milliAmpToDigitValue( ACK_TRESHOLD_VAL, digitsPerAmp );
825     actualCurrentDigitValue   = 0;
826     dccPacketsSend            = 0;
827     totalPwrSamplesTaken      = 0;
828     lastPwrSamplePerSecTaken  = 0;
829     pwrSamplesPerSec          = 0;
830
831     CDC::configureDio( enablePin, CDC::OUT );
832     CDC::configureDio( dccSigPin1, CDC::OUT );
833     CDC::configureDio( dccSigPin2, CDC::OUT );
834     CDC::configureAdc( sensePin );
835
836     CDC::writeDio( enablePin, false );
837     CDC::writeDioPair( dccSigPin1, false, dccSigPin2, false );
838
839     CDC::onTimerEvent( timerCallback );
840
841     if ( options & DT_OPT_SERVICE_MODE_TRACK ) {
842
843         progTrack     =   this;
844         preambleLen   =   PROG_PACKET_PREAMBLE_LEN;
845         postambleLen  =   PROG_PACKET_POSTAMBLE_LEN;
846         flags         |=  DT_F_SERVICE_MODE_ON;
847         activeBufPtr  = &resetDccPacket;
848         pendingBufPtr = &dccBuf1;
849     }
850     else {
851
852         mainTrack     =   this;
853         preambleLen   =   MAIN_PACKET_PREAMBLE_LEN;
854         postambleLen  =   MAIN_PACKET_POSTAMBLE_LEN;
855         activeBufPtr  = &idleDccPacket;
856         pendingBufPtr = &dccBuf1;
857     }
858
859     if ( trackDesc -> options & DT_OPT_CUTOUT ) {
860
861         preambleLen =  MAIN_PACKET_PREAMBLE_LEN - DCC_PACKET_CUTOUT_LEN;
862         flags       |= DT_F_CUTOUT_MODE_ON;
863         signalState =  DCC_SIG_CUTOUT_START;
864     }
865
866     if ( trackDesc -> options & DT_OPT_RAILCOM ) {
867
868         flags |= DT_F_RAILCOM_MODE_ON;
869         if ( CDC::configureUart( uartRxPin, CDC::UNDEFINED_PIN, 250000, CDC::UART_MODE_8N1 ) != ALL_OK ) {
870
871             flags = DT_F_CONFIG_ERROR;
872             return ( ERR_DCC_TRACK_CONFIG );
873         }
874     }
875
876     return ( ALL_OK );
877 }
878
879 //-------------------------------------------------------------------------------------------------------
880 // DCC signal generation is done through a state machine that is invoked when the DCC timer interrupts. The
881 // interrupt timer thinks in multiples of 29us, which we will just call a "tick" in the description below. It
882 // runs as part of the timer interrupt handler, so we need to be short and quick. First, the HW signals are
883 // set. This keeps the track signals in their timing. Next, the new signal state, time to run again and any
884 // other follow up action of this invocation are set. The idea is to separate HW signal generation and follow
885 // up actions. The timer interrupt handler will first call both state machines, MAIN and PROG, and then work
886 // on the optional follow-up actions. The state machine has the following states:
887 //
888 // DCC_SIG_CUTOUT_START: if the cutout option is on, a new DCC packet starts with this signal state. The
889 // DCC signal goes HIGH for one tick and the signal state advances to signal state DCC_SIG_CUTOUT_1.
890 //
```

```
// DCC_SIG_CUTOUT_1: this stage sets the signal to CUTOUT for cutout period ticks. Also, if the RailCom
// is enabled, there is a follow up request to start the serial IO read function. The signal state advances
// to signal state DCC_SIG_CUTOUT_2.
//
// DCC_SIG_CUTOUT_2: this stage sets the signal to LOW for the cutout end tick. The signal state advances
// to signal state DCC_SIG_CUTOUT_3.
//
// DC_SIG_CUTOUT_3: the DC_SIG_CUTOUT_3 and DC_SIG_END_CUTOUT states represent the first DCC "One" after
// the cutout. The DCC signal is set to HIGH and the next period is two ticks. The follow-up request is to
// disable the UART receiver. The signal state advances to DC_SIG_CUTOUT_END.
//
// DC_SIG_CUTOUT_END: The DC_SIG_END_CUTOUT state is the second half of the DCC one. The signal is set
// to low and the next period to two ticks. If RailCom is enabled, this is the state where a follow up
// to handle the RailCom data takes place. The next state is then DCC_SIG_START_BIT to handle the next
// packet, starting with the preamble of DCC ones.
//
// DCC_SIG_START_BIT: this stage is the start of the DCC packet bits, which are preamble, the data bytes
// with separators and postamble. If the cutout option is off, this is also the start for the DCC packet.
// The signal is set HIGH, the tick count is two and we need a follow up to get the current bit, which
// determines the length of the signal for the bit we just started. The next stage is signal state
// DCC_SIG_TEST_BIT.
//
// DCC_SIG_TEST_BIT: coming from signal state DCC_SIG_START_BIT, we need to see if the current bit is a ONE
// or ZERO bit. If a ONE bit, the signal needs to become LOW, the next period is 2 ticks and the next state
// is signal state DCC_SIG_START_BIT. If it is the last ONE bit of the postamble, the next packet and
// signal state needs to be determined. For a CUTOUT enabled track this is state DCC_SIG_START_CUTOUT, else
// DCC_SIG_START_BIT. If a ZERO bit, the signal is kept HIGH for another two ticks and the state is
// DCC_SIG_ZERO_SECOND_HALF.
//
// The ZERO bit case is also a good place to do a current measurement. We are already two ticks into the
// signal polarity change and there should be no spike from the signal level transition. However, we do
// not want to measure all zero bits since this would mean several hundreds to few thousands per second.
// Each data byte starts with a DCC ZERO bit. We will just sample the current there and end up with a few
// hundred samples per second, which is less of a burden but still often enough for overload detection
// and so on.
//
// DCC_SIG_ZERO_SECOND_HALF: coming from signal state DCC_SIG_TEST_BIT, we need to transmit the second half
// of the ZERO bit. The signal is set to LOW for four ticks and set the next stage is signal state to
// DCC_SIG_START_BIT.
//
// Note: for a 16Mhz Atmega the implementation for the cutout support is a close call. If the timer value
// setting takes place after the internal timer counter HW has passed this value, you wrap around and the
// interrupt happens the next time the timer value matches, which is about 4 milliseconds later! If you see
// such a gap in the DCC signal, this is perhaps the issue. When using the railcom/cutout option it is
// recommended to set the processor frequency to 20Mhz, which you can do in your own design, but not on
// an Arduino board.
//
//-----------------------------------------------------------------------------------------------------------
void LcsBaseStationDccTrack::runDccSignalStateMachine(

    volatile uint8_t  *timeToInterrupt,
    uint8_t           *followUpAction

    ) {

    switch ( signalState ) {

        case DCC_SIG_CUTOUT_START: {

            CDC::writeDioPair( dccSigPin1, true, dccSigPin2, false );
            *timeToInterrupt  = TICKS_29_MICROS;
            *followUpAction   = DCC_SIG_FOLLOW_UP_NONE;
            signalState       = DCC_SIG_CUTOUT_1;

        } break;

        case DCC_SIG_CUTOUT_1: {

            CDC::writeDioPair( dccSigPin1, false, dccSigPin2, false );
            *timeToInterrupt  = TICKS_CUTOUT_MICROS;
            *followUpAction   = (( flags & DT_F_RAILCOM_MODE_ON ) ?
                                  DCC_SIG_FOLLOW_UP_START_RAILCOM_IO : DCC_SIG_FOLLOW_UP_NONE );
            signalState       = DCC_SIG_CUTOUT_2;

        } break;

        case DCC_SIG_CUTOUT_2: {

            CDC::writeDioPair( dccSigPin1, false, dccSigPin2, true );
            *timeToInterrupt  = TICKS_29_MICROS;
            *followUpAction   = DCC_SIG_FOLLOW_UP_NONE;
            signalState       = DCC_SIG_CUTOUT_3;

        } break;

        case DCC_SIG_CUTOUT_3: {

            CDC::writeDioPair( dccSigPin1, true, dccSigPin2, false );
            *timeToInterrupt  = TICKS_58_MICROS;
            signalState       = DCC_SIG_CUTOUT_END;

            if ( flags & DT_F_RAILCOM_MODE_ON ) {

                flags           |= DT_F_RAILCOM_MSG_PENDING;
                *followUpAction = DCC_SIG_FOLLOW_UP_STOP_RAILCOM_IO;
            }
            else *followUpAction = DCC_SIG_FOLLOW_UP_NONE;

        } break;
```

```
990
991         case DCC_SIG_CUTOUT_END: {
992
993             CDC::writeDioPair( dccSigPin1, false, dccSigPin2, true );
994             *timeToInterrupt  = TICKS_58_MICROS;
995             *followUpAction   = (( flags & DT_F_RAILCOM_MODE_ON ) ?
996                                   DCC_SIG_FOLLOW_UP_RAILCOM_MSG : DCC_SIG_FOLLOW_UP_NONE );
997             signalState       = DCC_SIG_START_BIT;
998
999         } break;
1000
1001        case DCC_SIG_START_BIT: {
1002
1003            CDC::writeDioPair( dccSigPin1, true, dccSigPin2, false );
1004            *timeToInterrupt  = TICKS_58_MICROS;
1005            *followUpAction   = DCC_SIG_FOLLOW_UP_GET_BIT;
1006            signalState       = DCC_SIG_TEST_BIT;
1007
1008        } break;
1009
1010        case DCC_SIG_TEST_BIT: {
1011
1012            if ( currentBit ) {
1013
1014                CDC::writeDioPair( dccSigPin1, false, dccSigPin2, true );
1015
1016                if ( postambleSent >= postambleLen ) {
1017
1018                    *followUpAction = DCC_SIG_FOLLOW_UP_GET_PACKET;
1019                    signalState     = (( flags & DT_F_CUTOUT_MODE_ON ) ? DCC_SIG_CUTOUT_START : DCC_SIG_START_BIT );
1020                }
1021                else {
1022
1023                    *followUpAction = DCC_SIG_FOLLOW_UP_NONE;
1024                    signalState     = DCC_SIG_START_BIT;
1025                }
1026            }
1027            else {
1028
1029                *followUpAction   = (( bitsSent == 0 ) ? DCC_SIG_FOLLOW_UP_MEASURE_CURRENT :  DCC_SIG_FOLLOW_UP_NONE );
1030                signalState       = DCC_SIG_ZERO_SECOND_HALF;
1031            }
1032
1033            *timeToInterrupt  = TICKS_58_MICROS;
1034
1035        } break;
1036
1037        case DCC_SIG_ZERO_SECOND_HALF: {
1038
1039            CDC::writeDioPair( dccSigPin1, false, dccSigPin2, true );
1040            *timeToInterrupt  = TICKS_116_MICROS;
1041            *followUpAction   = DCC_SIG_FOLLOW_UP_NONE;
1042            signalState       = DCC_SIG_START_BIT;
1043
1044        } break;
1045
1046        default: {
1047
1048            *followUpAction   = DCC_SIG_FOLLOW_UP_NONE;
1049            *timeToInterrupt  = TICKS_58_MICROS;
1050        }
1051    }
1052}
1053
1054//-----------------------------------------------------------------------------------------------------------
1055// The "getNextBit" routine works through the active packet buffer bit for bit. A packet consists of the
1056// optional cutout sequence, the preamble bits, the data bytes separated by a ZERO bit and the postamble bits.
1057// The cutout option, the preamble and postamble are configured at DCC track object init time. The preamble
1058// length is different for MAIN and PROG tracks with the the cutout period overlaid at the beginning of the
1059// preamble. The postamble is currently always just one HIGH bit, according to standard.
1060//
1061// The routine works first through the preamble bit count, then through the data byte bits, and finally
1062// through the postamble bits. The bits to select from the data byte is done with a 9-bit mask. Remember that
1063// the first bit to send is the data byte separator, which is always a zero. We run from 0 to 8 through the
1064// bit mask, the first bit being the ZERO bit.
1065//
1066//-----------------------------------------------------------------------------------------------------------
1067void LcsBaseStationDccTrack::getNextBit( ) {
1068
1069    if ( preambleSent < preambleLen ) {
1070
1071        currentBit = true;
1072        preambleSent ++;
1073    }
1074    else if ( bytesSent < activeBufPtr -> len ) {
1075
1076        currentBit = activeBufPtr -> buf[ bytesSent ] & bitMask9[ bitsSent ];
1077        bitsSent ++;
1078
1079        if ( bitsSent == 9 ) {
1080
1081            bytesSent ++;
1082            bitsSent = 0;
1083        }
1084    }
1085    else if ( postambleSent < postambleLen ) {
1086
1087        currentBit = true;
1088        postambleSent ++;
```

```
1089        }
1090    }
1091
1092    //-----------------------------------------------------------------------------------------------------
1093    // If all bits of a packet have been processed, the next packet will be determined during the last ONE bit
1094    // transmission of the postamble. If there is a non-zero repeat count on the current packet, the same packet
1095    // is sent again until the repeat count drops to zero. On a zero repeat count, we check if there is a pending
1096    // packet. If so, it is copied to the active buffer and the pending flag is reset. This signals anyone waiting,
1097    // that the next packet can be queued. If there is no pending packet, we still need to keep the track going and
1098    // will load an IDLE or RESET packet.
1099    //
1100    // For non-service mode packets, there is a requirement that a decoder should not be receive two consecutive
1101    // packets. The standards talks about 5 milliseconds between two packets to the same decoder. For now, we will
1102    // not do anything special. A decoder will most likely, if there is more than one decoder active, not be
1103    // addressed in two consecutive packets, simply because the session refresh mechanism will go round robin
1104    // through the session list. However, if there is only one decoder active, two packets will be sent in a
1105    // row, but the decoders are robust enough to ignore this fact. Better run more than one loco :-).
1106    //
1107    // This routine is the central place to submit a DCC packet to the track and therefore a good place to write
1108    // a DCC_LOG record. We distinguish between a RESET, an IDLE and a data packet. Note that these records will
1109    // only be written when DCC logging is enabled.
1110    //
1111    //-----------------------------------------------------------------------------------------------------------
1112    void LcsBaseStationDccTrack::getNextPacket( ) {
1113
1114        bytesSent     = 0;
1115        bitsSent      = 0;
1116        preambleSent  = 0;
1117        postambleSent = 0;
1118
1119        if ( activeBufPtr -> repeat > 0 ) {
1120
1121            activeBufPtr -> repeat --;
1122
1123            writeLogData( LOG_DCC_PKT, activeBufPtr -> buf, activeBufPtr -> len );
1124        }
1125        else if ( flags & DT_F_DCC_PACKET_PENDING ) {
1126
1127            activeBufPtr  = pendingBufPtr;
1128            pendingBufPtr = (( pendingBufPtr == &dccBuf1 ) ? &dccBuf2 : &dccBuf1 );
1129            flags &= ~ DT_F_DCC_PACKET_PENDING;
1130
1131            writeLogData( LOG_DCC_PKT, activeBufPtr -> buf, activeBufPtr -> len );
1132        }
1133        else {
1134
1135            if ( flags & DT_F_SERVICE_MODE_ON ) {
1136
1137                activeBufPtr = &resetDccPacket;
1138                writeLogId( LOG_DCC_RST );
1139            }
1140            else {
1141
1142                activeBufPtr = &idleDccPacket;
1143                writeLogId( LOG_DCC_IDL );
1144            }
1145        }
1146
1147        dccPacketsSend ++;
1148    }
1149
1150    //-----------------------------------------------------------------------------------------------------
1151    // Railcom. If the cutout period and the RailCom feature is enabled, the signal state machine will also start
1152    // and stop the UART reader for RailCom data. The final message is then to handle that message. In the cutout
1153    // period, a decoder sends 8 data bytes. They are divided into two channels, 2bytes and another 6 bytes. The
1154    // bytes themselves are encoded such that each byte has four bits set, i.e. a hamming weight of 4. The first
1155    // channel is used to just send the locomotive address when the decoder is addressed. The second channel is
1156    // used only when the decoder is explicitly addressed via a CV operation command to provide the answer to the
1157    // request.
1158    //
1159    // The received datagrams are also recorded in the DCC_LOG, if enabled.
1160    //
1161    // ??? under construction....
1162    // ??? we could store the last loco address in some global variable.
1163    // ??? we could store the channel 2 datagram in the corresponding session.
1164    // ??? still, both pieces of data needs to go somewhere before the next message is received...
1165    //-----------------------------------------------------------------------------------------------------
1166    void LcsBaseStationDccTrack::startRailComIO( ) {
1167
1168        CDC::startUartRead( uartRxPin );
1169    }
1170
1171    void LcsBaseStationDccTrack::stopRailComIO( ) {
1172
1173        CDC::stopUartRead( uartRxPin );
1174    }
1175
1176    uint8_t LcsBaseStationDccTrack::handleRailComMsg( ) {
1177
1178        railComBufIndex = CDC::getUartBuffer( uartRxPin, railComMsgBuf, sizeof( railComMsgBuf ));
1179
1180        writeLogData( LOG_DCC_RCM, railComMsgBuf, railComBufIndex );
1181
1182        for ( uint8_t i = 0; i < railComBufIndex; i++ ) {
1183
1184            uint8_t dataByte = railComDecode[ railComMsgBuf[ i ]];
1185
1186            if      ( dataByte == ACK ) ;
1187            else if ( dataByte == NACK ) ;
```

```
1188            else if ( dataByte == BUSY ) ;
1189            else if ( dataByte < 64 ) {
1190
1191                // ??? valid
1192                // ??? a railCom message can have multiple datagrams
1193                // we would need to handle each datagram, one at a time or fill them into a kind of structure
1194                // that has a slot for the up to maximum 4 datagrams per railCom cutout period.
1195            }
1196            else {
1197
1198                // ??? invalid packet ... if this is channel2, discard the entire message.
1199            }
1200
1201            railComMsgBuf[ i ] = dataByte;
1202        }
1203
1204        flags &= ~ DT_F_RAILCOM_MSG_PENDING;
1205        return ( ALL_OK );
1206    }
1207
1208    // ??? not very useful, but good for debugging and initial testing .... and it works like a champ :-)
1209
1210    uint8_t LcsBaseStationDccTrack::getRailComMsg( uint8_t *buf, uint8_t bufLen ) {
1211
1212        if (( railComBufIndex > 0 ) && ( bufLen > 0 )) {
1213
1214            uint8_t i = 0;
1215
1216            do {
1217
1218                buf[ i ] = railComMsgBuf[ i ];
1219                i++;
1220
1221            } while (( i < railComBufIndex ) && ( i < bufLen ));
1222
1223            return ( i );
1224
1225        } else return ( 0 );
1226    }
1227
1228    //-------------------------------------------------------------------------------------------------------
1229    // DCC track power is not just a matter of turning power on or off. To address all the requirements of the
1230    // standard, the track is managed by a state machine that implements the start and stop sequences. It is also
1231    // important that we do not really block the progress of the entire base station, so any timing calls are
1232    // handled by timestamp comparison in state machine WAIT states. The track state machine routine is expected
1233    // to be called very often.
1234    //
1235    //  DCC_TRACK_POWER_START1    - this is the first state of a start sequence. When the track should be powered
1236    //                              on, the first activity is to set the status flags and enable the power module.
1237    //                              We set the power module current consumption to the initial limit configured.
1238    //                              The next state is TRACK_POWER_START2.
1239    //
1240    //  DCC_TRACK_POWER_START2    - we stay in this state until the threshold time has passed. Once the threshold
1241    //                              is reached, the current consumption limit is set to the configured limit.
1242    //                              Then we move on to DCC_TRACK_POWER_ON.
1243    //
1244    //  DCC_TRACK_POWER_ON        - this is the state when power is on and things are running normal. An overload
1245    //                              situation is set by the current measurement routines through setting the
1246    //                              overload status flag. We make sure that we have seen a couple of overloads
1247    //                              in a row before taking action which is to turn power off and set the
1248    //                              DCC_TRACK_POWER_OVERLOAD state. Otherwise we stay in this state.
1249    //
1250    //  DCC_TRACK_POWER_OVERLOAD  - with power turned off, we stay in this state until the threshold time has
1251    //                              passed. If passed, the overload restart count is incremented and checked for
1252    //                              its threshold. If reached, we have tried to restart several times and failed.
1253    //                              The track state becomes DCC_TRACK_POWER_STOP1, something is wrong on the track.
1254    //                              If not, we move on to DCC_TRACK_POWER_START1.
1255    //
1256    //  DCC_TRACK_POWER_STOP1     - this state initiates a shutdown sequence. We disable the power module, set
1257    //                              status flags and advance to the DCC_TRACK_POWER_STOP2 state.
1258    //
1259    //  DCC_TRACK_POWER_STOP2     - we stay in this state until the configured threshold has passed. Then we move
1260    //                              on to DCC_TRACK_POWER_OFF. The key reason for this time delay is to implement
1261    //                              the requirement that track turned off and perhaps switched to another mode,
1262    //                              should be powerless for one second. Switch track modes becomes simply a matter
1263    //                              of stopping and then starting again.
1264    //
1265    //  DCC_TRACK_POWER_OFF       - the track is disabled. We just stay in this state until the state is set to
1266    //                              a different state from outside.
1267    //
1268    // During the power on state, we also append the actual current measurement value to a circular buffer when
1269    // the time interval for this kind of measurement has passed. The idea is to measure the samples at a more
1270    // or less constant interval rate and compute the power consumption RMS value from the data in the buffer
1271    // when requested. In the interest of minimizing the controller load, the calculation is done in digit values
1272    // the result is presented in then in milliAmps.
1273    //
1274    //-------------------------------------------------------------------------------------------------------
1275    void LcsBaseStationDccTrack::runDccTrackStateMachine( ) {
1276
1277        switch ( trackState ) {
1278
1279            case DCC_TRACK_POWER_START1: {
1280
1281                // ??? do we need a way to check for overload during this initial phase, just like we do when ON ?
1282
1283                trackTimeStamp          = CDC::getMillis( );
1284                flags                   |= DT_F_POWER_ON;
1285                flags                   &= ~DT_F_POWER_OVERLOAD;
1286                flags                   &= ~DT_F_MEASUREMENT_ON;
```

```
1287            limitCurrentDigitValue  = milliAmpToDigitValue( initCurrentMilliAmp, digitsPerAmp );
1288
1289            CDC::writeDio( enablePin, true );
1290            trackState = DCC_TRACK_POWER_START2;
1291
1292        }  break;
1293
1294        case DCC_TRACK_POWER_START2: {
1295
1296            if (( CDC::getMillis( ) - trackTimeStamp ) > startTimeThreshold ) {
1297
1298                highWaterMarkDigitValue = 0;
1299                actualCurrentDigitValue = 0;
1300                overloadRestartCount    = 0;
1301                overloadEventCount      = 0;
1302                flags                   |= DT_F_POWER_ON | DT_F_MEASUREMENT_ON;
1303                limitCurrentDigitValue  = milliAmpToDigitValue( limitCurrentMilliAmp, digitsPerAmp );
1304
1305                CDC::writeDio( enablePin, true );
1306                trackState = DCC_TRACK_POWER_ON;
1307            }
1308
1309        } break;
1310
1311        case DCC_TRACK_POWER_ON: {
1312
1313            if (( CDC::getMillis( ) - lastPwrSampleTimeStamp ) > PWR_SAMPLE_TIME_INTERVAL_MILLIS ) {
1314
1315                pwrSampleBuf[ pwrSampleBufIndex % DCC_TRACK_POWER_ON ] = actualCurrentDigitValue;
1316                pwrSampleBufIndex ++;
1317                lastPwrSampleTimeStamp = CDC::getMillis( );
1318            }
1319
1320            if (( CDC::getMillis( ) - lastPwrSamplePerSecTimeStamp ) > 1000 ) {
1321
1322                pwrSamplesPerSec          = totalPwrSamplesTaken - lastPwrSamplePerSecTaken;
1323                lastPwrSamplePerSecTaken  = totalPwrSamplesTaken;
1324                lastPwrSamplePerSecTimeStamp = CDC::getMillis( );
1325            }
1326
1327            if ( flags & DT_F_POWER_OVERLOAD ) {
1328
1329                overloadEventCount  ++;
1330
1331                if ( overloadEventCount > overloadEventThreshold ) {
1332
1333                    if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_TRACK_POWER_MGMT )) {
1334
1335                        printf( "Overload detected: " );
1336
1337                        if ( options &  DT_OPT_SERVICE_MODE_TRACK ) printf( "Prog Track: " );
1338                        else                                        printf( "Main Track: " );
1339
1340                        #if 0
1341                        printf( "(hwm(mA): %d : limit(mA): %d )\n",
1342                                digitValueToMilliAmp( highWaterMarkDigitValue, digitsPerAmp ),
1343                                digitValueToMilliAmp( limitCurrentDigitValue, digitsPerAmp ));
1344
1345                        #else
1346                        printf( "(hwm(dVal): %d  : limit(dVal): %d )\n", highWaterMarkDigitValue, limitCurrentDigitValue );
1347                        #endif
1348                    }
1349
1350                    trackTimeStamp  = CDC::getMillis( );
1351                    flags           |= DT_F_POWER_OVERLOAD;
1352                    flags           &= ~DT_F_POWER_ON;
1353                    flags           &= ~DT_F_MEASUREMENT_ON;
1354
1355                    CDC::writeDio( enablePin, false );
1356                    trackState = DCC_TRACK_POWER_OVERLOAD;
1357                }
1358            }
1359
1360        }  break;
1361
1362        case DCC_TRACK_POWER_OVERLOAD: {
1363
1364            if ( CDC::getMillis( ) - trackTimeStamp > overloadTimeThreshold ) {
1365
1366                overloadRestartCount ++;
1367
1368                if ( overloadRestartCount > overloadRestartThreshold ) {
1369
1370                    if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_TRACK_POWER_MGMT )) {
1371
1372                        printf( "Overload restart failed, Cnt:%d\n", overloadRestartCount );
1373                    }
1374
1375                    trackState = DCC_TRACK_POWER_STOP1;
1376                }
1377                else trackState = DCC_TRACK_POWER_START1;
1378            }
1379
1380        }  break;
1381
1382        case DCC_TRACK_POWER_STOP1: {
1383
1384            trackTimeStamp  = CDC::getMillis( );
1385            flags           &= ~DT_F_POWER_ON;
```

```
1386            flags           &= ~DT_F_POWER_OVERLOAD;
1387            flags           &= ~DT_F_MEASUREMENT_ON;
1388
1389            CDC::writeDio( enablePin, false );
1390            trackState = DCC_TRACK_POWER_STOP2;
1391
1392        }  break;
1393
1394        case DCC_TRACK_POWER_STOP2: {
1395
1396            if ( CDC::getMillis( ) - trackTimeStamp > stopTimeThreshold ) trackState = DCC_TRACK_POWER_OFF;
1397
1398        } break;
1399
1400        case DCC_TRACK_POWER_OFF: {
1401
1402        } break;
1403    }
1404 }
1405
1406 //--------------------------------------------------------------------------------------------------------------
1407 // Some getter functions. Straightforward.
1408 //
1409 //--------------------------------------------------------------------------------------------------------------
1410 uint16_t LcsBaseStationDccTrack::getFlags( ) {
1411
1412     return ( flags );
1413 }
1414
1415 uint16_t LcsBaseStationDccTrack::getOptions( ) {
1416
1417     return ( options );
1418 }
1419
1420 uint32_t LcsBaseStationDccTrack::getDccPacketsSend( ) {
1421
1422     return ( dccPacketsSend );
1423 }
1424
1425 uint32_t LcsBaseStationDccTrack::getPwrSamplesTaken( ) {
1426
1427     return ( totalPwrSamplesTaken );
1428 }
1429
1430 uint16_t LcsBaseStationDccTrack::getPwrSamplesPerSec( ) {
1431
1432     return ( pwrSamplesPerSec );
1433 }
1434
1435 bool LcsBaseStationDccTrack::isPowerOn( ) {
1436
1437     return ( flags & DT_F_POWER_ON );
1438 }
1439
1440 bool LcsBaseStationDccTrack::isPowerOverload( ) {
1441
1442     return ( flags & DT_F_POWER_OVERLOAD );
1443 }
1444
1445 bool LcsBaseStationDccTrack::isServiceModeOn( ) {
1446
1447     return ( flags & DT_F_SERVICE_MODE_ON );
1448 }
1449
1450 bool LcsBaseStationDccTrack::isCutoutOn( ) {
1451
1452     return ( flags & DT_F_CUTOUT_MODE_ON );
1453 }
1454
1455 bool LcsBaseStationDccTrack::isRailComOn( ) {
1456
1457     return ( flags & DT_F_RAILCOM_MODE_ON );
1458 }
1459
1460 //--------------------------------------------------------------------------------------------------------------
1461 // DCC track power management functions. The actual state of track power is kept in the track status field
1462 // and can be queried or set by setting the respective flag. Starting and stopping track power is done by
1463 // setting the respective START or STOP state.
1464 //
1465 //--------------------------------------------------------------------------------------------------------------
1466 void LcsBaseStationDccTrack::powerStart( ) {
1467
1468     trackState = DCC_TRACK_POWER_START1;
1469 }
1470
1471 void LcsBaseStationDccTrack::powerStop( ) {
1472
1473     trackState = DCC_TRACK_POWER_STOP1;
1474 }
1475
1476 void LcsBaseStationDccTrack::serviceModeOn( ) {
1477
1478     if ( options & DT_OPT_SERVICE_MODE_TRACK ) flags |= DT_F_SERVICE_MODE_ON;
1479 }
1480
1481 void LcsBaseStationDccTrack::serviceModeOff( ) {
1482
1483     if ( options & DT_OPT_SERVICE_MODE_TRACK ) flags &= ~DT_F_SERVICE_MODE_ON;
1484 }
```

```
1485
1486 void LcsBaseStationDccTrack::cutoutOn( ) {
1487
1488     if ( ! ( options & DT_OPT_SERVICE_MODE_TRACK )) {
1489
1490         preambleLen =  MAIN_PACKET_PREAMBLE_LEN - DCC_PACKET_CUTOUT_LEN;
1491         flags       |= DT_F_CUTOUT_MODE_ON;
1492     }
1493 }
1494
1495 void LcsBaseStationDccTrack::cutoutOff( ) {
1496
1497     if ( ! ( options & DT_OPT_SERVICE_MODE_TRACK )) {
1498
1499         preambleLen =  MAIN_PACKET_PREAMBLE_LEN;
1500         flags       &= ~DT_F_CUTOUT_MODE_ON;
1501         flags       &= ~DT_F_RAILCOM_MODE_ON;
1502     }
1503 }
1504
1505 void LcsBaseStationDccTrack::railComOn( ) {
1506
1507     if ( ! ( options & DT_OPT_SERVICE_MODE_TRACK )) {
1508
1509         flags |= DT_F_CUTOUT_MODE_ON | DT_F_RAILCOM_MODE_ON;
1510     }
1511 }
1512
1513 void LcsBaseStationDccTrack::railComOff( ) {
1514
1515     if ( ! ( options & DT_OPT_SERVICE_MODE_TRACK )) flags &= ~DT_F_RAILCOM_MODE_ON;
1516 }
1517
1518 //-------------------------------------------------------------------------------------------------------------
1519 // Power Consumption Management. There are two key values. The first is the actual current consumption as
1520 // measured by the ADC hardware on each ZERO DCC bit. This value is used to do the power overload checking.
1521 // The second value is the high water mark built from these measurements. This values is used for the DCC
1522 // decoder programming logic. The high water mark will be set to zero before collecting measurements. All
1523 // measurement values are actually ADC digit values for performance reason. Only on limit setting and external
1524 // data access are these values converted from and to milliAmps.
1525 //
1526 //-------------------------------------------------------------------------------------------------------------
1527 uint16_t LcsBaseStationDccTrack::getLimitCurrent( ) {
1528
1529     return ( limitCurrentMilliAmp );
1530 }
1531
1532 uint16_t LcsBaseStationDccTrack::getActualCurrent( ) {
1533
1534     return ( digitValueToMilliAmp( actualCurrentDigitValue, digitsPerAmp ));
1535 }
1536
1537 uint16_t LcsBaseStationDccTrack::getInitCurrent( ) {
1538
1539     return ( initCurrentMilliAmp );
1540 }
1541
1542 uint16_t LcsBaseStationDccTrack::getMaxCurrent( ) {
1543
1544     return ( maxCurrentMilliAmp );
1545 }
1546
1547 void LcsBaseStationDccTrack::setLimitCurrent( uint16_t val ) {
1548
1549     if      ( val < initCurrentMilliAmp )  val = initCurrentMilliAmp;
1550     else if ( val > maxCurrentMilliAmp  )  val = maxCurrentMilliAmp;
1551
1552     limitCurrentMilliAmp    = val;
1553     limitCurrentDigitValue  = milliAmpToDigitValue( val, digitsPerAmp );
1554 }
1555
1556 //-------------------------------------------------------------------------------------------------------------
1557 // The "getRMSCurrent" function returns the power consumption based on the samples taken and stored in the
1558 // sample buffer. The function computes the square root of the sum of the squares of the array elements. The
1559 // result is returned in milliAmps. Note that our measurement is based on unsigned 16-bit quantities that come
1560 // from the controller ADC converter. We compute the RMS based on 16-bit unsigned integers, which compared
1561 // to floating point computation is not really precise. However, for our purpose to just show a rough power
1562 // consumption, the error should be not a big issue. We will not use RMS values for power overload detection
1563 // or decoder ACK detection.
1564 //
1565 //-------------------------------------------------------------------------------------------------------------
1566 uint16_t LcsBaseStationDccTrack::getRMSCurrent( ) {
1567
1568     uint32_t res = 0;
1569
1570     for ( uint8_t i = 0; i < PWR_SAMPLE_BUF_SIZE; i++ ) res += pwrSampleBuf[ i ] * pwrSampleBuf[ i ];
1571
1572     return ( digitValueToMilliAmp( sqrt( res / PWR_SAMPLE_BUF_SIZE ), digitsPerAmp ));
1573 }
1574
1575 //-------------------------------------------------------------------------------------------------------------
1576 // This function is called whenever a power measurement operation completes from the analog conversion
1577 // interrupt handler. This typically takes place on the first half of the DCC "0" bit. If power measurement
1578 // is enabled, we increment the number of samples taken, check the measured value for an overload situation
1579 // and also set the high water mark accordingly. Since we are part of an interrupt handler, keep the amount
1580 // work really short.
1581 //
1582 //-------------------------------------------------------------------------------------------------------------
1583 void LcsBaseStationDccTrack::powerMeasurement( ) {
```

```
1584
1585        if ( flags & DT_F_MEASUREMENT_ON ) {
1586
1587            actualCurrentDigitValue = CDC::readAdc( sensePin );
1588
1589            totalPwrSamplesTaken ++;
1590
1591            if ( actualCurrentDigitValue > highWaterMarkDigitValue ) highWaterMarkDigitValue = actualCurrentDigitValue;
1592            if ( actualCurrentDigitValue > limitCurrentDigitValue ) flags |= DT_F_POWER_OVERLOAD;
1593        }
1594    }
1595
1596    //-------------------------------------------------------------------------------------------------------
1597    // The DCC decoder programming requires the detection of a current consumption change. This is the way a DCC
1598    // decoder signals an acknowledgement. To detect the consumption change we need first an idea what the actual
1599    // average current baseline consumption of the decoder is. This method will send the required DCC reset packets
1600    // according to the DCC standard and at the same time determine the current consumption as a baseline. We use
1601    // the high water mark for this purpose.
1602    //
1603    // ??? although the routines for decoder ACK detection work, they will produce quite a number of packets.
1604    // During this time, other LCS work is blocked. Perhaps we need a kind of state machine approach to cut the
1605    // long sequence in smaller chunks to allow other work in between.
1606    //-------------------------------------------------------------------------------------------------------
1607    uint16_t LcsBaseStationDccTrack::decoderAckBaseline( uint8_t resetPacketsToSend ) {
1608
1609        if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_DCC_ACK_DETECT )) {
1610
1611            printf( "\nDecoder Ack setup: ( " );
1612        }
1613
1614        uint16_t sum = 0;
1615
1616        for ( uint8_t i = 0; i < resetPacketsToSend; i++ ) {
1617
1618            highWaterMarkDigitValue = 0;
1619
1620            loadPacket( resetDccPacketData, 2, 0 );
1621
1622            if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_DCC_ACK_DETECT )) {
1623
1624                printf( "%d ", highWaterMarkDigitValue );
1625            }
1626
1627            sum += highWaterMarkDigitValue;
1628        }
1629
1630        if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_DCC_ACK_DETECT )) {
1631
1632            printf( ") -> %d\n", ( sum + resetPacketsToSend - 1 ) / resetPacketsToSend );
1633        }
1634
1635        return (( sum + resetPacketsToSend - 1 ) / resetPacketsToSend );
1636    }
1637
1638    //-------------------------------------------------------------------------------------------------------
1639    // "decoderAckDetect" is the counterpart to the decoder ack setup routine. The setup method established a base
1640    // line for the power consumption and put the decoder in CV programming mode by sending the RESET packets. The
1641    // decoder ACK detect routine now sends out resets packets to follow the programming packets required and
1642    // monitors the current consumption. We use the high water mark for this purpose. The DCC standard specifies
1643    // a time window in which the decoder should raise its power consumption level and signal an acknowledge this
1644    // way. We will send out a series of reset packets and monitor after each packet the consumption level. The
1645    // number of retries depends on whether it is a read ( 50ms window ) or a write ( 100ms window). If we detect
1646    // a raised value the decoder did signal a positive outcome. If not, we time out after the last reset packet.
1647    // The programming operation either failed or the decoder did on purpose not answer. We cannot tell.
1648    //
1649    // ??? although the routines for decoder ACK detection work, they will produce quite a number of packets.
1650    // During this time, other LCS work is blocked. Perhaps we need a kind of state machine approach to cut the
1651    // long sequence in smaller chunks to allow other work in between.
1652    //-------------------------------------------------------------------------------------------------------
1653    bool LcsBaseStationDccTrack::decoderAckDetect( uint16_t baseDigitValue, uint8_t retries ) {
1654
1655        if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_DCC_ACK_DETECT )) {
1656
1657            printf( "Decoder Ack detect: ( %d : %d : ( ", baseDigitValue, ackThresholdDigitValue );
1658        }
1659
1660        for ( uint8_t i = 0; i < retries; i++ ) {
1661
1662            highWaterMarkDigitValue = 0;
1663
1664            loadPacket( resetDccPacketData, 2, 0 );
1665
1666            if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_DCC_ACK_DETECT )) {
1667
1668                printf( "%d ", highWaterMarkDigitValue );
1669            }
1670
1671            if (( highWaterMarkDigitValue >= baseDigitValue ) &&
1672                ( highWaterMarkDigitValue - baseDigitValue >= ackThresholdDigitValue )) {
1673
1674                if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_DCC_ACK_DETECT )) {
1675
1676                    printf( "[ %d ] ) -> OK\n", abs( highWaterMarkDigitValue - baseDigitValue ));
1677                }
1678
1679            return ( true );
1680            }
1681        }
1682
```

```
1683        if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_DCC_ACK_DETECT )) {
1684
1685            printf( ") -> FAILED" );
1686        }
1687
1688        return ( false );
1689    }
1690
1691    //-----------------------------------------------------------------------------------------------------------
1692    // LoadPacket is the central entry point to submit a DCC packet. The incoming packet is the the data to be
1693    // sent without checksum, i.e. it is just the payload. The DCC track signal generator has two packet buffers.
1694    // The first buffer holds the packet currently being transmitted. The second is the pending buffer. If it is
1695    // used, we will simply busy wait for our turn to load the packet into the pending buffer. Upon completion of
1696    // sending the active packet, the interrupt handler copies the currently pending buffer to the active buffer
1697    // and then resets the pending flag. Either way, then it is our turn. We fill the pending buffer, compute the
1698    // checksum and set the pending flag.
1699    //
1700    // ??? For a high number of session we may want to think about a queuing approach. Right now, this routine
1701    // waits when there is a packet already queued, i.e. pending. This may cause issues in delaying other tasks
1702    // such as receiving a CAN bus message.
1703    //-----------------------------------------------------------------------------------------------------------
1704    void LcsBaseStationDccTrack::loadPacket( const uint8_t *packet, uint8_t len, uint8_t repeat ) {
1705
1706        if ( ! isInRangeU( len, MIN_DCC_PACKET_SIZE, MAX_DCC_PACKET_SIZE )) return;
1707        if ( ! isInRangeU( repeat, MIN_DCC_PACKET_REPEATS, MAX_DCC_PACKET_REPEATS )) return;
1708
1709        while ( flags & DT_F_DCC_PACKET_PENDING );
1710
1711        pendingBufPtr -> len    = len + 1;
1712        pendingBufPtr -> repeat = repeat;
1713
1714        uint8_t checkSum = 0;
1715        uint8_t *bufPtr  = pendingBufPtr -> buf;
1716
1717        for ( uint8_t i = 0; i < len; i++ ) {
1718
1719            bufPtr[ i ] =   packet[ i ];
1720            checkSum    ^=  bufPtr[ i ];
1721        }
1722
1723        bufPtr[ len ] = checkSum;
1724        flags         |= DT_F_DCC_PACKET_PENDING;
1725    }
1726
1727    //-----------------------------------------------------------------------------------------------------------
1728    // The log management routines. A typical transaction to log would start the logging process and then end
1729    // it after the operation to analyze/debug. The "enableLog" call should be used to enable the logging
1730    // process all together, the other calls will only do work when the log is enabled. With this call the
1731    // recording process could be controlled from a command line setting or so. "beginLog" and "endLog" start
1732    // and end a recording sequence.
1733    //
1734    //-----------------------------------------------------------------------------------------------------------
1735    void LcsBaseStationDccTrack::enableLog( bool arg ) {
1736
1737        logEnabled = arg;
1738        logActive  = false;
1739    }
1740
1741    void LcsBaseStationDccTrack::beginLog( ) {
1742
1743        if ( logEnabled ) {
1744
1745            logActive   = true;
1746            logBufIndex = 0;
1747            writeLogId( LOG_BEGIN );
1748            writeLogTs( );
1749        }
1750    }
1751
1752    void LcsBaseStationDccTrack::endLog( ) {
1753
1754        if ( logActive ) {
1755
1756            writeLogTs( );
1757            writeLogId( LOG_END );
1758            logActive = false;
1759        }
1760    }
1761
1762    //-----------------------------------------------------------------------------------------------------------
1763    // There are a couple of routines to write the log data when the logging is active. For convenience, some of
1764    // the log entry types are available as a direct call. The order of data entry for numeric types is big endian,
1765    // i.e. most significant byte first.
1766    //
1767    //-----------------------------------------------------------------------------------------------------------
1768    void LcsBaseStationDccTrack::writeLogData( uint8_t id, uint8_t *buf, uint8_t len ) {
1769
1770        if ( logActive ) {
1771
1772            len = len % 16;
1773            if ( logBufIndex + len + 1 < LOG_BUF_SIZE ) {
1774
1775                logBuf[ logBufIndex ++ ] = ( id << 4 ) | len;
1776                for ( uint8_t i = 0; i < len; i++ ) logBuf[ logBufIndex ++ ] = buf[ i ];
1777            }
1778        }
1779    }
1780
1781    void LcsBaseStationDccTrack::writeLogId( uint8_t id ) {
```

```
1782
1783        if ( logActive ) logBuf[ logBufIndex ++ ] = ( id << 4 );
1784    }
1785
1786    void LcsBaseStationDccTrack::writeLogTs( ) {
1787
1788        if ( logActive ) {
1789
1790            uint32_t ts = CDC::getMicros( );
1791            logBuf[ logBufIndex ++ ] = ( LOG_TSTAMP << 4 ) | 4;
1792            logBuf[ logBufIndex ++ ] = ( ts >> 24 ) & 0xFF;
1793            logBuf[ logBufIndex ++ ] = ( ts >> 16 ) & 0xFF;
1794            logBuf[ logBufIndex ++ ] = ( ts >> 8  ) & 0xFF;
1795            logBuf[ logBufIndex ++ ] = ( ts >> 0  ) & 0xFF;
1796        }
1797    }
1798
1799    void LcsBaseStationDccTrack::writeLogVal( uint8_t valId, uint16_t val ) {
1800
1801        if ( logActive ) {
1802
1803            logBuf[ logBufIndex ++ ] = ( LOG_VAL << 4 ) | 3;
1804            logBuf[ logBufIndex ++ ] = valId;
1805            logBuf[ logBufIndex ++ ] = val >> 8;
1806            logBuf[ logBufIndex ++ ] = val & 0xFF;
1807        }
1808    }
1809
1810    //-----------------------------------------------------------------------------------------------------------------
1811    // Print out the log data, one entry on one line. We only print the log buffer when there is no log sequence
1812    // active.
1813    //
1814    //-----------------------------------------------------------------------------------------------------------------
1815    void LcsBaseStationDccTrack::printLog( ) {
1816
1817        if ( logEnabled ) {
1818
1819            if ( ! logActive ) {
1820
1821                if ( logBufIndex > 0 ) {
1822
1823                    printf( "\n" );
1824
1825                    uint16_t entryIndex  = 0;
1826                    uint8_t  entryLen    = 0;
1827
1828                    while ( entryIndex < logBufIndex ) {
1829
1830                        entryLen = printLogEntry( entryIndex );
1831                        printf( "\n" );
1832
1833                        if ( entryLen > 0 ) entryIndex += entryLen;
1834                        else                break;
1835                    }
1836                }
1837                else printf( "DCC Log Buf: Nothing recorded\n" );
1838            }
1839            else printf( "DCC Log Active\n" );
1840        }
1841        else printf( "DCC Log disabled\n" );
1842    }
1843
1844    //-----------------------------------------------------------------------------------------------------------------
1845    // Print out the DCC Track configuration data. For debugging purposes.
1846    //
1847    //-----------------------------------------------------------------------------------------------------------------
1848    void LcsBaseStationDccTrack::printDccTrackConfig( ) {
1849
1850        printf( "DccTrack Config: " );
1851
1852        if ( options & DT_OPT_SERVICE_MODE_TRACK ) printf( "PROG \n" );
1853        else                                       printf( "MAIN \n" );
1854
1855        printf( " Config options: ( 0x%x ) -> ", flags );
1856
1857        if ( options &  DT_OPT_SERVICE_MODE_TRACK ) printf( "SvcMode Track " );
1858        if ( options & DT_OPT_CUTOUT ) printf( "Cutout " );
1859        if ( options & DT_OPT_RAILCOM ) printf( "Railcom " );
1860        printf( "\n" );
1861
1862        printf( " Current Initial(mA): %d Current Limit(mA): %d Current Max(mA): %d\n",
1863                getInitCurrent( ), getLimitCurrent( ), getMaxCurrent( ));
1864        printf( " milliVoltPerAmp: %d\n", milliVoltPerAmp );
1865        printf( " digitsPerAmp: %d\n", digitsPerAmp );
1866
1867        printf( " Limit Digit Value: %d\n", limitCurrentDigitValue );
1868        printf( " Ack Threshold Digit Value:%d\n", ackThresholdDigitValue );
1869
1870        printf( " CDC enable Pin: %d, DCC signal Pins: (%d:%d), Sensor Pin: %d, RailCom Pin: %d\n",
1871                enablePin, dccSigPin1, dccSigPin2, sensePin, uartRxPin );
1872
1873        printf( " PreambleLen: %d, PostambleLen: %d\n", preambleLen, postambleLen );
1874    }
1875
1876    //-----------------------------------------------------------------------------------------------------------------
1877    // Print out the DCC Track status.
1878    //
1879    //-----------------------------------------------------------------------------------------------------------------
1880    void LcsBaseStationDccTrack::printDccTrackStatus( ) {
```

```
1881
1882        printf( "DccTrack: " );
1883
1884        if ( options & DT_OPT_SERVICE_MODE_TRACK )  printf( "PROG" );
1885        else                                        printf( "MAIN" );
1886
1887        printf( ", Track Status: ( 0x%x ) -> ", flags );
1888
1889        if ( flags & DT_F_POWER_ON          ) printf( "PowerOn " );
1890        if ( flags & DT_F_POWER_OVERLOAD    ) printf( "PowerOverload " );
1891        if ( flags & DT_F_MEASUREMENT_ON    ) printf( "PowerMeasOn " );
1892        if ( flags & DT_F_SERVICE_MODE_ON   ) printf( "SvcModeOn " );
1893        if ( flags & DT_F_CUTOUT_MODE_ON    ) printf( "CutoutOn " );
1894        if ( flags & DT_F_RAILCOM_MODE_ON   ) printf( "RailcomOn " );
1895        if ( flags & DT_F_CONFIG_ERROR      ) printf( "ConfigError " );
1896        printf( "\n" );
1897
1898        printf( "Packets Send: %d\n", dccPacketsSend );
1899        printf( "Total Power Samples: %d\n", totalPwrSamplesTaken );
1900        printf( "Power Samples per Sec: %d\n", pwrSamplesPerSec );
1901        printf( "Power consumption (RMS): %d\n", getRMSCurrent( ));
1902        printf( "\n" );
1903    }
```

```
//--------------------------------------------------------------------------------------------------------------
//
// LCS Base Station - Loco Session Management - implementation file
//
//--------------------------------------------------------------------------------------------------------------
// The locomotive session object is the besides the two DCC tracks the other main component of a base station.
// Each engine to run needs a session on this session object. Typically, the handheld will "open" a session.
// The session identifier is then the handle to the locomotive.
//
//
//
//--------------------------------------------------------------------------------------------------------------
//
// LCS - Base Station
// Copyright (C) 2019 - 2024  Helmut Fieres
//
// This program is free software: you can redistribute it and/or modify it under the terms of the GNU General
// Public License as published by the Free Software Foundation, either version 3 of the License, or (at your
// option) any later version.
//
// This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
// implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
// for more details.
//
// You should have received a copy of the GNU General Public License along with this program. If not, see
// http://www.gnu.org/licenses
//
//  GNU General Public License:  http://opensource.org/licenses/GPL-3.0
//
//--------------------------------------------------------------------------------------------------------------
#include "LcsBaseStation.h"
#include <malloc.h>

using namespace LCS;

//--------------------------------------------------------------------------------------------------------------
// External global variables.
//
//--------------------------------------------------------------------------------------------------------------
extern uint16_t debugMask;

//--------------------------------------------------------------------------------------------------------------
// Loco Session implementation file - local declarations.
//
//--------------------------------------------------------------------------------------------------------------
namespace {

//--------------------------------------------------------------------------------------------------------------
// DCC packet definitions. A DCC packet payload is at most 10 bytes long, excluding the checksum byte. This
// is true for XPOM support, otherwise it is according to NMRA up to 6 bytes.
//
//--------------------------------------------------------------------------------------------------------------
const uint8_t   MIN_DCC_PACKET_SIZE         = 2;
const uint8_t   MAX_DCC_PACKET_SIZE         = 16;
const uint8_t   MIN_DCC_PACKET_REPEATS      = 0;
const uint8_t   MAX_DCC_PACKET_REPEATS      = 8;

//--------------------------------------------------------------------------------------------------------------
// Utility routines.
//
//--------------------------------------------------------------------------------------------------------------
bool isInRangeU( uint8_t val, uint8_t lower, uint8_t upper ) {

    return (( val >= lower ) && ( val <= upper ));
}

bool isInRangeU( uint16_t val, uint16_t lower, uint16_t upper ) {

    return (( val >= lower ) && ( val <= upper ));
}

bool isInRangeU( uint32_t val, uint32_t lower, uint32_t upper ) {

    return (( val >= lower ) && ( val <= upper ));
}

bool validCabId( uint16_t cabId ) {

    return ( isInRangeU( cabId, MIN_CAB_ID, MAX_CAB_ID ));
}

bool validCvId( uint16_t cvId ) {

    return ( isInRangeU( cvId, MIN_DCC_CV_ID, MAX_DCC_CV_ID ));
}

bool validFunctionId( uint8_t fId ) {

    return ( isInRangeU( fId, MIN_DCC_FUNC_ID, MAX_DCC_FUNC_ID ));
}

bool validFunctionGroupId( uint8_t fGroup ) {

    return ( isInRangeU( fGroup, MIN_DCC_FUNC_GROUP_ID , MAX_DCC_FUNC_GROUP_ID ));
}

bool validDccPacketlen( uint8_t len ) {

```

```
99      return ( isInRangeU( len, MIN_DCC_PACKET_SIZE, MAX_DCC_PACKET_SIZE ));
100 }
101
102 bool validDccPacketRepeatCnt( uint8_t nRepeat ) {
103
104     return ( isInRangeU( nRepeat, MIN_DCC_PACKET_REPEATS, MAX_DCC_PACKET_REPEATS ));
105 }
106
107 uint8_t lowByte( uint16_t arg ) {
108
109     return( arg & 0xFF );
110 }
111
112 uint8_t highByte( uint16_t arg ) {
113
114     return( arg >> 8 );
115 }
116
117 uint8_t bitRead( uint8_t arg, uint8_t pos ) {
118
119     return ( arg >> ( pos % 8 )) & 1;
120 }
121
122 void bitWrite( uint8_t *arg, uint8_t pos, bool val ) {
123
124     if ( val )  *arg |= ( 1 << pos );
125     else        *arg &= ~( 1 << pos );
126 }
127
128 //-----------------------------------------------------------------------------------------------------------
129 // DDC function flags. The DCC function flags F0 .. F68 are stored in ten groups. Group 0 contains F0 .. F4
130 // stored in DCC command byte format. Group 1 contains F5 .. F8, Group 2 contains F9 .. F12 in DCC command
131 // byte format. The remainder F13 .. F68 are stored in 8 bits groups also in DCC command byte format. The
132 // routines support the get/set of an individual bit as well as setting an entire function group. A DCC
133 // function group is labelled starting with index 1.
134 //
135 //-----------------------------------------------------------------------------------------------------------
136 bool getDccFuncBit( uint8_t *funcFlags, uint8_t fNum ) {
137
138     if      ( fNum == 0 )                 return ( bitRead( funcFlags[ 0 ], 4 ));
139     else if ( isInRangeU( fNum, 1, 4 ))   return ( bitRead( funcFlags[ 0 ], fNum - 1 ));
140     else if ( isInRangeU( fNum, 5, 8 ))   return ( bitRead( funcFlags[ 1 ], fNum - 5 ));
141     else if ( isInRangeU( fNum, 9, 12 ))  return ( bitRead( funcFlags[ 2 ], fNum - 9 ));
142     else if ( isInRangeU( fNum, 13, 68 )) {
143
144         return ( bitRead( funcFlags[ ( fNum - 13 ) / 8 + 3 ], ( fNum - 13 ) % 8 ));
145     }
146     else return false;
147 }
148
149 void setDccFuncBit( uint8_t *funcFlags, uint8_t fNum, bool val ) {
150
151     if      ( fNum == 0 )                 bitWrite( &funcFlags[ 0 ], 4, val );
152     else if ( isInRangeU( fNum, 1, 4 ))   bitWrite( &funcFlags[ 0 ], fNum - 1, val );
153     else if ( isInRangeU( fNum, 5, 8 ))   bitWrite( &funcFlags[ 1 ], fNum - 5, val );
154     else if ( isInRangeU( fNum, 9, 12 ))  bitWrite( &funcFlags[ 2 ], fNum - 9, val );
155     else if ( isInRangeU( fNum, 13, 68 )) {
156
157         bitWrite( &funcFlags[ ( fNum - 13 ) / 8 + 3 ], ( fNum - 13 ) % 8, val );
158     }
159 }
160
161 void setDccFuncGroupByte( uint8_t *funcFlags, uint8_t fGroup, uint8_t dccByte ) {
162
163     if       ( fGroup == 1 )                funcFlags[ 0 ] = dccByte & 0x1F;
164     else if  ( fGroup == 2 )                funcFlags[ 1 ] = dccByte & 0x0F;
165     else if  ( fGroup == 3 )                funcFlags[ 2 ] = dccByte & 0x0F;
166     else if  ( isInRangeU( fGroup, 4, 10 ))  funcFlags[ fGroup - 1 ] = dccByte;
167 }
168
169 uint8_t dccFunctionBitToGroup( uint8_t fNum ) {
170
171     if      ( isInRangeU( fNum, 0, 4 ))     return ( 1 );
172     else if ( isInRangeU( fNum, 5, 8 ))     return ( 2 );
173     else if ( isInRangeU( fNum, 9, 12 ))    return ( 3 );
174     else if ( isInRangeU( fNum, 13, 68 ))   return (( fNum - 13 ) / 8 + 4 );
175     else                                    return ( 0 );
176 }
177
178 }; // namespace
179
180 //===========================================================================================================
181 //===========================================================================================================
182 //
183 // Object part.
184 //
185 //===========================================================================================================
186 //===========================================================================================================
187
188 //-----------------------------------------------------------------------------------------------------------
189 // "LocoSession" constructor. Nothing to do here.
190 //
191 //-----------------------------------------------------------------------------------------------------------
192 LcsBaseStationLocoSession::LcsBaseStationLocoSession( ) { }
193
194 //-----------------------------------------------------------------------------------------------------------
195 // Loco Session Map configuration. The session map contains an array of loco sessions entries. We are passed
196 // the sessionMap descriptor and object handles to the core library and the two tracks. Loco sessions are
197 // numbered from 1 to MAX_SESSION_ID. During compilation there is a maximum number of sessions that the
```

```
198 // session map will support. This number cannot be changed other than recompile with a different setting.
199 //
200 //-------------------------------------------------------------------------------------------------------------
201 uint8_t LcsBaseStationLocoSession::setupSessionMap(
202
203     LcsBaseStationSessionMapDesc  *sessionMapDesc,
204     LcsBaseStationDccTrack        *mainTrack,
205     LcsBaseStationDccTrack        *progTrack
206
207     ) {
208
209     if (( mainTrack == nullptr                            ) ||
210         ( progTrack == nullptr                            ) ||
211         ( sessionMapDesc -> maxSessions > MAX_CAB_SESSIONS  )) return ( ERR_SESSION_SETUP );
212
213     this -> mainTrack     = mainTrack;
214     this -> progTrack     = progTrack;
215
216     options               = sessionMapDesc -> options;
217     flags                 = SM_F_DEFAULT_SETTING;
218     sessionMap            = (SessionMapEntry *) calloc( sessionMapDesc -> maxSessions, sizeof( SessionMapEntry ));
219     lastAliveCheckTime    = CDC::getMillis( );
220
221     sessionMapHwm         = sessionMap;
222     sessionMapLimit       = &sessionMap[ sessionMapDesc -> maxSessions ];
223     sessionMapNextRefresh = sessionMap;
224
225     if ( options & SM_OPT_ENABLE_REFRESH )      flags |= SM_F_ENABLE_REFRESH;
226     if ( options & SM_OPT_KEEP_ALIVE_CHECKING ) flags |= SM_F_KEEP_ALIVE_CHECKING;
227
228     for ( SessionMapEntry *smePtr = sessionMap; smePtr < sessionMapLimit; smePtr++ ) initSessionEntry( smePtr );
229
230     return ( ALL_OK );
231 }
232
233 //-------------------------------------------------------------------------------------------------------------
234 // "requestSession" is the entry point to establish a session. There are several modes. The NORMAL mode is
235 // to allocate a new session. There should be no session already existing for this cabId. The STEAL mode
236 // grabs an existing session from the current session holder. The use case is that a dispatched locomotive
237 // can be taken over by another handheld. The SHARED option allows several handheld controller to share the
238 // session entry and issue commands to the same locomotive. Right now, the STEAL and SHARED option are not
239 // implemented.
240 //
241 //-------------------------------------------------------------------------------------------------------------
242 uint8_t LcsBaseStationLocoSession::requestSession( uint16_t cabId, uint8_t mode, uint8_t *sId ) {
243
244     *sId = NIL_LOCO_SESSION_ID;
245     if ( ! validCabId( cabId )) return ( ERR_INVALID_CAB_ID );
246
247     switch ( mode ) {
248
249         case LSM_NORMAL: {
250
251             SessionMapEntry *smePtr = allocateSessionEntry( cabId );
252             if ( smePtr == nullptr ) return ( ERR_LOCO_SESSION_ALLOCATE );
253
254             smePtr -> flags |= SME_SPDIR_ONLY_REFRESH;
255
256             *sId = smePtr - sessionMap + 1;
257             return ( ALL_OK );
258         }
259
260         case LSM_STEAL: {
261
262             // ??? need to inform the current handheld and put the new handheld in its place.
263             return ( ERR_NOT_IMPLEMENTED );
264
265         } break;
266
267         case LSM_SHARED: {
268
269             // ??? essentially, add another handheld to the session. We perhaps need a counter on how many handhelds
270             // share the session ...
271             return ( ERR_NOT_IMPLEMENTED );
272
273         } break;
274
275         default: return ( ERR_NOT_IMPLEMENTED ); // ??? rather "invalid mode" ?
276     }
277 }
278
279 //-------------------------------------------------------------------------------------------------------------
280 // A cab session can be released, freeing up the slot in the cab session table.
281 //
282 // ??? for a shared session, what does this mean ?
283 //-------------------------------------------------------------------------------------------------------------
284 uint8_t LcsBaseStationLocoSession::releaseSession( uint8_t sId ) {
285
286     SessionMapEntry *smePtr = getSessionMapEntryPtr( sId );
287     if ( smePtr == nullptr ) return ( ERR_INVALID_SESSION_ID );
288
289     deallocateSessionEntry( smePtr );
290     return ( ALL_OK );
291 }
292
293 //-------------------------------------------------------------------------------------------------------------
294 // "updateSession" informs the base station about changes in the loco session setting. To be implemented once
295 // we know what the flags and the update concept should be ...
296 //
```

```
297 //--------------------------------------------------------------------------------------------------------------
298 uint8_t LcsBaseStationLocoSession::updateSession( uint8_t sId, uint8_t flags ) {
299 
300     SessionMapEntry *smePtr = getSessionMapEntryPtr( sId );
301     if ( smePtr == nullptr ) return ( ERR_INVALID_SESSION_ID );
302 
303     return ( ERR_NOT_IMPLEMENTED );
304 }
305 
306 //--------------------------------------------------------------------------------------------------------------
307 // "markSessionAlive" sets the keep alive time stamp on a loco session. This routine is typically called by
308 // the LCS message receiver to update the session last "alive" timestamp. The base station will periodically
309 // check this value to see if a session is still alive.
310 //
311 //--------------------------------------------------------------------------------------------------------------
312 uint8_t LcsBaseStationLocoSession::markSessionAlive( uint8_t sId ) {
313 
314     SessionMapEntry *smePtr = getSessionMapEntryPtr( sId );
315     if ( smePtr == nullptr ) return ( ERR_INVALID_SESSION_ID );
316 
317     smePtr -> lastKeepAliveTime = CDC::getMillis( );
318     return ( ALL_OK );
319 }
320 
321 //--------------------------------------------------------------------------------------------------------------
322 // "refreshActiveSessions" walks through the session map up to the high water mark and invokes the session
323 // refresh function for each used entry. As the refresh entry routine will show, we will do this refreshing
324 // in small pieces in order to stay responsive to external requests.
325 //
326 //
327 // ??? this may should perhaps all be reworked. There are many more duties to do periodically.
328 // ??? an active loco ( speed > 0 ) needs to be address at least every 2.5 seconds.
329 //
330 // ??? also a base station needs to broadcast its capabilities every
331 //
332 //--------------------------------------------------------------------------------------------------------------
333 void LcsBaseStationLocoSession::refreshActiveSessions( ) {
334 
335     if (( flags & SM_F_ENABLE_REFRESH ) && ( sessionMapHwm > sessionMap )) {
336 
337         refreshSessionEntry( sessionMapNextRefresh );
338 
339         sessionMapNextRefresh ++;
340         if ( sessionMapNextRefresh >= sessionMapHwm ) sessionMapNextRefresh = sessionMap;
341     }
342 }
343 
344 //--------------------------------------------------------------------------------------------------------------
345 // "refreshSessionEntry" checks first that the session is still alive and then issues the next DCC packet for
346 // refreshing the loco session. To avoid DCC bandwidth issues, a loco session refresh is done in several small
347 // steps. There is one state for speed and direction and steps to refresh the function groups 1 to 5. If the
348 // function refresh option is set, we use the DCC command that sets speed, direction and the function flags in
349 // one DCC command.
350 //
351 //    Step 0  -> refresh speed and direction ( if FUNC_REFRESH is set also functions F0 .. F28 )
352 //    Step 1  -> refresh function group 0 ( F0  .. F4  )
353 //    Step 2  -> refresh function group 1 ( F5  .. F8  )
354 //    Step 3  -> refresh function group 2 ( F9  .. F12 )
355 //    Step 4  -> refresh function group 3 ( F13 .. F20 )
356 //    Step 5  -> refresh function group 4 ( F21 .. F28 )
357 //
358 // ??? should we alternate when SPDIR and FUNC are sent separately ?
359 // ??? is it something like: SPDIR, FG1, SPDIR, FG2, ...
360 //
361 // ??? what to do for emergency stop, keep refreshing ? keep alive checking ?
362 // ??? how do we integrate the STEAL/SHARE/DISPATCHED concept ?
363 //
364 // ??? separate out the check alive functionality ? it is a separate task...
365 // ??? sessionMapNextAliveCheck var needed ...
366 //--------------------------------------------------------------------------------------------------------------
367 void LcsBaseStationLocoSession::refreshSessionEntry( SessionMapEntry *smePtr ) {
368 
369     // ??? introduce a return status ?
370 
371     if ( smePtr -> cabId != NIL_CAB_ID ) {
372 
373         if ( flags & SM_F_KEEP_ALIVE_CHECKING ) {
374 
375             if (( CDC::getMillis( ) - smePtr -> lastKeepAliveTime ) > refreshAliveTimeOutVal ) {
376 
377                 if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_CHECK_ALIVE_SESSIONS )) {
378 
379                     printf( "Session: %d expired\n", smePtr - sessionMap );
380                 }
381 
382                 deallocateSessionEntry( smePtr );
383             }
384         }
385 
386         // ??? separate keep alive checking and refresh options...
387 
388         else {
389 
390             // ??? if ( smePtr -> speed > 0 )  // only active locos are refreshed...
391 
392             if ( smePtr -> nextRefreshStep == 0 ) {
393 
394                 setThrottle( smePtr, smePtr -> speed, smePtr -> direction );
395 
```

```
396                 smePtr -> nextRefreshStep = ((( smePtr -> flags & SME_COMBINED_REFRESH ) ||
397                                              ( smePtr -> flags & SME_SPDIR_ONLY_REFRESH )) ? 0 : 1 );
398             }
399             else if ( smePtr -> nextRefreshStep <= 5 ) {
400 
401             uint8_t fGroup = smePtr -> nextRefreshStep;
402 
403             setDccFunctionGroup( smePtr, fGroup, smePtr -> functions[ fGroup - 1 ] );
404             smePtr -> nextRefreshStep = (( smePtr -> nextRefreshStep >= 5 ) ? 0 : smePtr -> nextRefreshStep + 1 );
405 
406             }
407         }
408     }
409 }
410 
411 //--------------------------------------------------------------------------------------------------------------
412 // "emergencyStopAll" is called when one of the clients issued an emergency stop all request. There is a DCC
413 // broadcast packet that causes all decoders to stop the locos. In addition, the base station is expected to
414 // discontinue sending non-zero speed packets until the situation is cleared. The standard does not really say
415 // what exactly to do. In our base station, we will first issue the ESTOP DCC broadcast packet and then set
416 // the speed value in each session to one, which is the value for emergency stop. All else is unchanged.
417 //
418 //--------------------------------------------------------------------------------------------------------------
419 void LcsBaseStationLocoSession::emergencyStopAll( ) {
420 
421     mainTrack -> loadPacket( eStopDccPacketData, 2, 4 );
422 
423     for ( SessionMapEntry *smePtr = sessionMap; smePtr < sessionMapHwm; smePtr++ ) {
424 
425         if ( smePtr -> cabId != NIL_CAB_ID ) smePtr -> speed = 1;
426     }
427 }
428 
429 //--------------------------------------------------------------------------------------------------------------
430 // Getter methods for session related info. Straightforward.
431 //
432 //--------------------------------------------------------------------------------------------------------------
433 uint8_t LcsBaseStationLocoSession::getSessionIdByCabId( uint16_t cabId ) {
434 
435     SessionMapEntry *smePtr = lookupSessionEntry( cabId );
436     return (( smePtr == nullptr ) ? NIL_LOCO_SESSION_ID : (( smePtr - sessionMap ) + 1 ));
437 }
438 
439 uint16_t LcsBaseStationLocoSession::getOptions( ) {
440 
441     return ( options );
442 }
443 
444 uint16_t LcsBaseStationLocoSession::getFlags( ) {
445 
446     return ( flags );
447 }
448 
449 uint8_t LcsBaseStationLocoSession::getSessionMapHwm( ) {
450 
451     return ( sessionMapHwm - sessionMap );
452 }
453 
454 uint32_t LcsBaseStationLocoSession::getSessionKeepAliveInterval( ) {
455 
456     return ( refreshAliveTimeOutVal );
457 }
458 
459 uint8_t LcsBaseStationLocoSession::getActiveSessions( ) {
460 
461     uint8_t sessionCnt = 0;
462 
463     for ( SessionMapEntry *smePtr = sessionMap; smePtr < sessionMapHwm; smePtr++ ) {
464 
465         if ( smePtr -> cabId != NIL_CAB_ID ) sessionCnt++;
466     }
467 
468     return ( sessionCnt );
469 }
470 
471 //--------------------------------------------------------------------------------------------------------------
472 // "setThrottle" is perhaps the most used function. After all, we want to run engines on the track. This
473 // signature will just locate the session map entry and then invoke the internal signature with accepts a
474 // pointer to the entry.
475 //
476 //--------------------------------------------------------------------------------------------------------------
477 uint8_t LcsBaseStationLocoSession::setThrottle( uint8_t sId, uint8_t speed, uint8_t direction ) {
478 
479     SessionMapEntry *smePtr = getSessionMapEntryPtr( sId );
480     if ( smePtr == nullptr ) return ( ERR_INVALID_SESSION_ID );
481 
482     return ( setThrottle( smePtr, speed, direction ));
483 }
484 
485 //--------------------------------------------------------------------------------------------------------------
486 // "setThrottle" will send a DCC packet with speed and direction for a loco. If the combined speed and
487 // function refresh option is enabled, the DCC command will specify speed, direction and functions to refresh
488 // in one packet.
489 //
490 //--------------------------------------------------------------------------------------------------------------
491 uint8_t LcsBaseStationLocoSession::setThrottle( SessionMapEntry *smePtr, uint8_t speed, uint8_t direction ) {
492 
493     uint8_t pBuf[ MAX_DCC_PACKET_SIZE ];
494     uint8_t pLen = 0;
```

```
495
496     smePtr -> speed      = speed & 0x7F;
497     smePtr -> direction  = direction % 2;
498
499     if ( smePtr -> cabId > 127 ) pBuf[pLen++] = highByte( smePtr -> cabId ) | 0xC0;
500     pBuf[pLen++] = lowByte( smePtr -> cabId );
501
502     pBuf[pLen++] = (( smePtr -> flags & SME_COMBINED_REFRESH ) ? 0x3c : 0x3F );
503     pBuf[pLen++] = (( smePtr -> speed & 0x7F ) | (( smePtr -> direction ) ? 0x80 : 0 ));
504
505     if ( smePtr -> flags & SME_COMBINED_REFRESH ) {
506
507         pBuf[pLen++]  = ((( smePtr -> functions[0] & 0x10 ) >> 4 ) |
508                          (( smePtr -> functions[0] & 0x0F ) << 1 ) |
509                          (( smePtr -> functions[1] & 0x07 ) << 5 ));
510
511         pBuf[pLen++]  = ((( smePtr -> functions[1] & 0x0F ) >> 3 ) |
512                          (( smePtr -> functions[2] & 0x0F ) << 1 ) |
513                          (( smePtr -> functions[3] & 0x07 ) << 5 ));
514
515         pBuf[pLen++]  = ((( smePtr -> functions[3] & 0xf80 ) >> 3 ) |
516                          (( smePtr -> functions[4] & 0x07 ) << 5 ));
517
518         pBuf[pLen++]  = (( smePtr -> functions[4] & 0xf80 ) >> 3 );
519     }
520
521     mainTrack -> loadPacket( pBuf, pLen );
522     return ( ALL_OK );
523 }
524
525 //-------------------------------------------------------------------------------------------------------
526 // "setDccFunctionBit" controls the functions in a decoder. The DCC function flags F0 .. F68 are stored in
527 // ten groups. The routines first updates the function bit in the loco session entry data structure, so we
528 // can keep track of the values. This is important as the DCC commands send out entire groups only. The
529 // actual work is then done by the "setDccFunctionGroup" method.
530 //
531 //-------------------------------------------------------------------------------------------------------
532 uint8_t LcsBaseStationLocoSession::setDccFunctionBit( uint8_t sId, uint8_t fNum, uint8_t val ) {
533
534     SessionMapEntry *smePtr = getSessionMapEntryPtr( sId );
535     if ( smePtr == nullptr ) return ( ERR_INVALID_SESSION_ID );
536
537     if ( ! validFunctionId( fNum )) return ( ERR_INVALID_FUNC_ID );
538     setDccFuncBit( smePtr -> functions, fNum, val );
539
540     uint8_t fGroup = dccFunctionBitToGroup( fNum );
541
542     return ( setDccFunctionGroup( smePtr, fGroup, smePtr -> functions[ fGroup - 1 ] ));
543 }
544
545 //-------------------------------------------------------------------------------------------------------
546 // "setDccFunctionGroup" sets an entire group of function flags. This signature will first find the session
547 // entry, do the argument checks and the invoke the internal signature.
548 //
549 //-------------------------------------------------------------------------------------------------------
550 uint8_t LcsBaseStationLocoSession::setDccFunctionGroup( uint8_t sId, uint8_t fGroup, uint8_t dccByte ) {
551
552     SessionMapEntry *smePtr = getSessionMapEntryPtr( sId );
553     if ( smePtr == nullptr ) return ( ERR_INVALID_SESSION_ID );
554
555     return ( setDccFunctionGroup( smePtr, fGroup, dccByte ));
556 }
557
558 //-------------------------------------------------------------------------------------------------------
559 // "setDccFunctionGroup" sets an entire group of function flags.The DCC function flags F0 .. F68 are stored
560 // in ten groups.
561 //
562 //      Group 1:  F0, F4, F3, F2, F1      DCC Command Format: 100DDDDD
563 //      Group 2:  F8, F7, F6, F5          DCC Command Format: 1011DDDD
564 //      Group 3:  F12, F11, F10, F9       DCC Command Format: 1010DDDD
565 //      Group 4:  F20 .. F13              DCC Command Format: 0xDE DDDDDDDD
566 //      Group 5:  F28 .. F21              DCC Command Format: 0xDF DDDDDDDD
567 //      Group 6:  F36 .. F29              DCC Command Format: 0xD8 DDDDDDDD
568 //      Group 7:  F44 .. F37              DCC Command Format: 0xD9 DDDDDDDD
569 //      Group 8:  F52 .. F45              DCC Command Format: 0xDA DDDDDDDD
570 //      Group 9:  F60 .. F53              DCC Command Format: 0xDB DDDDDDDD
571 //      Group 10: F68 .. F61              DCC Command Format: 0xDC DDDDDDDD
572 //
573 // The routines updates the entire function group byte in the loco session entry, so we can keep track of the
574 // values. The function command is repeated 4 times to the track.
575 //
576 //-------------------------------------------------------------------------------------------------------
577 uint8_t LcsBaseStationLocoSession::setDccFunctionGroup( SessionMapEntry *smePtr, uint8_t fGroup, uint8_t dccByte ) {
578
579     if ( ! validFunctionGroupId( fGroup )) return ( ERR_INVALID_FGROUP_ID );
580     setDccFuncGroupByte( smePtr -> functions, fGroup, dccByte );
581
582     uint8_t pBuf[ MAX_DCC_PACKET_SIZE];
583     uint8_t pLen = 0;
584
585     if ( smePtr -> cabId > 127 ) pBuf[pLen++] = highByte( smePtr -> cabId ) | 0xC0;
586     pBuf[pLen++] = lowByte( smePtr -> cabId );
587
588     switch ( fGroup - 1 ) {
589
590         case 0: pBuf[pLen++] = ( smePtr -> functions[ 0 ] & 0x1F ) | 0x80; break;
591         case 1: pBuf[pLen++] = ( smePtr -> functions[ 1 ] & 0x0F ) | 0xB0; break;
592         case 2: pBuf[pLen++] = ( smePtr -> functions[ 2 ] & 0x0F ) | 0xA0; break;
593
```

```
        case 3: pBuf[pLen++] = 0xDE; pBuf[pLen++] = smePtr -> functions[ 3 ]; break;
        case 4: pBuf[pLen++] = 0xDF; pBuf[pLen++] = smePtr -> functions[ 4 ]; break;
        case 5: pBuf[pLen++] = 0xD8; pBuf[pLen++] = smePtr -> functions[ 5 ]; break;
        case 6: pBuf[pLen++] = 0xD9; pBuf[pLen++] = smePtr -> functions[ 6 ]; break;
        case 7: pBuf[pLen++] = 0xDA; pBuf[pLen++] = smePtr -> functions[ 7 ]; break;
        case 8: pBuf[pLen++] = 0xDB; pBuf[pLen++] = smePtr -> functions[ 8 ]; break;
        case 9: pBuf[pLen++] = 0xDC; pBuf[pLen++] = smePtr -> functions[ 9 ]; break;
    }

    mainTrack -> loadPacket( pBuf, pLen, 4 );
    return ( ALL_OK );
}

//-------------------------------------------------------------------------------------------------------------
// "writeCVMain" writes a CV value to the decoder on the main track. CV numbers range from 1 to 1024, but are
// encoded from 0 to 1023. The DCC standard defines various modes for retrieving CV values. This function
// implements CV write mode mode 0 and 1, by calling the respective method. The other modes are not supported.
// For bit mode access, the bit position and bit value are encoded in the "val" parameter with bit 3 containing
// the data and bit 0 ..2 the bit offset.
//
//    0 Direct Byte
//    1 Direct Bit
//    2 Page Mode
//    3 Register Mode
//    4 Address Only Mode
//
//
// Note on the MAIN track, there is no way for the decoder to answer via a raise in power consumption. The
// command shown here is just sent. If however RailCom is available, the decoder can answer with the CV
// value in a following cutout. This is currently not implemented.
//-------------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::writeCVMain( uint8_t sId, uint16_t cvId, uint8_t mode, uint8_t val ) {

    if        ( mode == 0 )  return ( writeCVByteMain( sId, cvId, val ));
    else if   ( mode == 1 )  return ( writeCVBitMain( sId, cvId, ( val & 0x07 ), (( val & 0x08 ) >> 3 )));
    else                     return ( ERR_INVALID_CV_MODE );
}

//-------------------------------------------------------------------------------------------------------------
// "writeCVByteMain" writes a byte to the CV while the loco is on the main track. The CV numbers range from
// 1 to 1024, but are encoded from 0 to 1023. This function implements CV write mode mode 0, which is write
// a byte at a time. There is no way to validate our operation, only writes are possible. The packet is sent
// four times.
//
//-------------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::writeCVByteMain( uint8_t sId, uint16_t cvId, uint8_t val ) {

    uint8_t   pBuf[ MAX_DCC_PACKET_SIZE ];
    uint8_t   pLen = 0;

    SessionMapEntry *smePtr = getSessionMapEntryPtr( sId );
    if ( smePtr == nullptr ) return ( ERR_INVALID_SESSION_ID );

    if ( ! validCvId( cvId )) return ( ERR_INVALID_CV_ID );
    cvId--;

    if ( smePtr -> cabId > 127 ) pBuf[pLen++] = highByte( smePtr -> cabId ) | 0xC0;
    pBuf[pLen++] = lowByte( smePtr -> cabId );
    pBuf[pLen++] = 0xEC + ( highByte( cvId ) & 0x03 );
    pBuf[pLen++] = lowByte( cvId );
    pBuf[pLen++] = val;

    mainTrack -> loadPacket( pBuf, pLen, 4 );
    return ( ALL_OK );
}

//-------------------------------------------------------------------------------------------------------------
// "writeCVBitMain" writes a bit to the CV while the loco is on the main track. The CV numbers range from 1
// to 1024, but are encoded from 0 to 1023. his function implements CV write mode mode 1, which is write a
// bit at a time. On input the "val" parameter encodes the bit position in bits 0 - 2 and the bit value in
// bit 3.  There is no way to validate our operation, only CV writes are possible. The packet is sent four
// times.
//
//-------------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::writeCVBitMain( uint8_t sId, uint16_t cvId, uint8_t bitPos, uint8_t val ) {

    SessionMapEntry *smePtr = getSessionMapEntryPtr( sId );
    if ( smePtr == nullptr ) return ( ERR_INVALID_SESSION_ID );

    if ( ! validCvId( cvId )) return ( ERR_INVALID_CV_ID );
    cvId--;

    uint8_t pBuf[ MAX_DCC_PACKET_SIZE ];
    uint8_t pLen = 0;

    if ( smePtr -> cabId > 127 ) pBuf[pLen++] = highByte( smePtr -> cabId ) | 0xC0;
    pBuf[pLen++] = lowByte( smePtr -> cabId );
    pBuf[pLen++] = 0xE8 + (highByte( cvId ) & 0x03 );
    pBuf[pLen++] = lowByte( cvId );
    pBuf[pLen++] = 0xF0 + (( val % 2 ) << 3 ) + ( bitPos % 8 );

    mainTrack -> loadPacket( pBuf, pLen, 4 );
    return ( ALL_OK );
}

//-------------------------------------------------------------------------------------------------------------
// "readCV" retrieves a CV value from the decoder in service mode. CV numbers range from 1 to 1024, but are
// encoded from 0 to 1023. This command is only available in service mode, i.e. on a programming track. The
// DCC standard defines various modes for retrieving CV values. We only support mode 0 and 1. The other modes
```

```
693 // are not supported. For bit mode access, the bit position and bit value are encoded in the "val" parameter
694 // with bit 3 containing the data and bit 0 ..2 the bit offset.
695 //
696 //    0 - Direct Byte
697 //    1 - Direct Bit
698 //    2 - Page Mode
699 //    3 - Register Mode
700 //    4 - Address Only Mode
701 //
702 // This function implements the CV read mode 0 and 1, which is reading a byte or a bit at a time by calling
703 // the respective method.
704 //
705 //-----------------------------------------------------------------------------------------------------------
706 uint8_t LcsBaseStationLocoSession::readCV( uint16_t cvId, uint8_t mode, uint8_t *val ) {
707
708     if        ( mode == 0 )  return ( readCVByte( cvId, val ));
709     else if   ( mode == 1 )  return ( readCVBit( cvId, *val % 8, val ));
710     else                     return ( ERR_INVALID_CV_MODE );
711 }
712
713 //-----------------------------------------------------------------------------------------------------------
714 // "readCVByte" will retrieve a complete byte from the decoder. CV numbers range from 1 to 1024, but are
715 // encoded from 0 to 1023. This command is only available in service mode, i.e. on a programming track.
716 // Reading a CV value where the decoder can only respond with a "yes" or "no" is a tedious matter. We are
717 // actually reading the CV value bit by bit and then ask if the assembled byte read is the one just read. The
718 // general packet sequence is a according to DCC standard standard 3 or more RESET packets, 5 or more identical
719 // READ packets and then RESET packages until acknowledge or timeout. The RESET packet preamble and postamble
720 // series are sent during the decoder ack setup and detect call to the DCC track object. During the preamble
721 // we  figure out the base current consumption of the decoder, during the postamble packets we measure to get
722 // the decoder acknowledge, which is a short raise in power consumption to indicate an ACK.
723 //
724 //
725 // ??? This command may take a long time, a lot of packets are sent. While this not an issue with the signal
726 // generation, which is done via interrupt handlers, it may be an issue with any other work of the base
727 // station. This code needs to be redesigned to use a kind of state machine that sends a packet at a time
728 // so other work can interleave.
729 //-----------------------------------------------------------------------------------------------------------
730 uint8_t LcsBaseStationLocoSession::readCVByte( uint16_t cvId, uint8_t *val ) {
731
732     if ( ! ( progTrack -> isServiceModeOn( ))) return ( ERR_NO_SVC_MODE );
733     if ( ! validCvId( cvId )) return ( ERR_INVALID_CV_ID );
734     cvId--;
735
736     uint8_t   pBuf[ MAX_DCC_PACKET_SIZE ];
737     uint8_t   bValue  = 0;
738     uint16_t  base    = progTrack -> decoderAckBaseline( 5 );
739
740     pBuf[0]  = 0x78 + ( highByte( cvId ) & 0x03 );
741     pBuf[1]  = lowByte( cvId );
742
743     for ( int i = 0; i < 8; i++ ) {
744
745         pBuf[2] = 0xE8 + i;
746         progTrack -> loadPacket( pBuf, 3, 5 );
747         bitWrite( &bValue, i, progTrack -> decoderAckDetect( base, 9 ));
748     }
749
750     *val    = bValue;
751     pBuf[0] = 0x74 + ( highByte( cvId ) & 0x03 );
752     pBuf[1] = lowByte( cvId );
753     pBuf[2] = bValue;
754     progTrack -> loadPacket( pBuf, 3, 5 );
755
756     return (( progTrack -> decoderAckDetect( base, 9 )) ? ALL_OK : (LcsErrorCodes) ERR_CV_OP_FAILED );
757 }
758
759 //-----------------------------------------------------------------------------------------------------------
760 // "readCVBit" will retrieve one bit from a CV variable from the decoder. CV numbers range from 1 to 1024,
761 // but are encoded from 0 to 1023. This command is only available in service mode, i.e. on a programming
762 // track. The "val" parameter encodes the bit position in bits 0 - 2. We are reading the CV value bit and
763 // then ask if the bit read is the one just read. We first try to validate a zero bit. If that succeeds,
764 // fine. Otherwise we try to validate a one bit. If that succeeds, fine. Otherwise we have a CV read error.
765 // The general packet sequence is a according to DCC standard 3 or more RESET packets, 5 or more identical
766 // READ packets and then RESET packages until acknowledge or timeout. The RESET packet preamble and postamble
767 // are sent during the decoder ack setup and detect call to the DCC track object. During the preamble we
768 // figure out the base current consumption of the decoder, during the postamble we measure to get the decoder
769 // acknowledge, which is a short raise in power consumption to indicate an ACK.
770 //
771 // ??? This command may take a long time, a lot of packets are sent. While this not an issue with the signal
772 // generation, which is done via interrupt handlers, it may be an issue with any other work of the base
773 // station. This code needs to be redesigned to use a kind of state machine that sends a packet at a time
774 // so other work can interleave.
775 //-----------------------------------------------------------------------------------------------------------
776 uint8_t LcsBaseStationLocoSession::readCVBit( uint16_t cvId, uint8_t bitPos, uint8_t *val ) {
777
778     if ( ! ( progTrack -> isServiceModeOn( ))) return ( ERR_NO_SVC_MODE );
779
780     if ( ! validCvId( cvId )) return ( ERR_INVALID_CV_ID );
781     cvId--;
782
783     uint8_t  pBuf[ MAX_DCC_PACKET_SIZE ];
784     int   base = progTrack -> decoderAckBaseline( 5 );
785
786     pBuf[0]  = 0x78 + ( highByte( cvId ) & 0x03 );
787     pBuf[1]  = lowByte( cvId );
788     pBuf[2]  = 0xE8 + ( bitPos % 8 );
789
790     progTrack -> loadPacket( pBuf, 3, 5 );
791
```

```
792     if ( ! ( progTrack -> decoderAckDetect( base, 9 ))) {
793
794         pBuf[2] = 0xE8 + 8 + ( bitPos % 8 );
795         progTrack -> loadPacket( pBuf, 3, 5 );
796
797         if ( progTrack -> decoderAckDetect( base, 9 )) {
798
799             *val = 1;
800             return ( ALL_OK );
801         }
802         else return ( ERR_CV_OP_FAILED );
803     }
804     else return ( ALL_OK );
805 }
806
807 //-----------------------------------------------------------------------------------------------------------
808 // "writeCV" writes a CV value to the decoder. CV numbers range from 1 to 1024, but are encoded from 0 to
809 // 1023. This command is only available in service mode, i.e. on a programming track. The DCC standard defines
810 // various modes for accessing CV values. For bit mode access, the bit position and bit value are encoded in
811 // the "val" parameter with bit 3 containing the data and bit 0 .. 2 the bit offset.
812 //
813 //    0 Direct Byte
814 //    1 Direct Bit
815 //    2 Page Mode
816 //    3 Register Mode
817 //    4 Address Only Mode
818 //
819 // This function implements the CV write mode 0 and 1, which is writing a byte or a bit at a time by calling
820 // the respective method.
821 //
822 //-----------------------------------------------------------------------------------------------------------
823 uint8_t LcsBaseStationLocoSession::writeCV( uint16_t cvId, uint8_t mode, uint8_t val ) {
824
825     if        ( mode == 0 )  return ( writeCVByte( cvId, val ));
826     else if   ( mode == 1 )  return ( writeCVBit( cvId, ( val & 0x07 ), (( val & 0x08 ) >> 3 )));
827     else                     return ( ERR_INVALID_CV_MODE );
828 }
829
830 //-----------------------------------------------------------------------------------------------------------
831 // "writeCVByte" puts a data byte into the CV on the decoder. This function is only available in service mode.
832 // The CV numbers range from 1 to 1024, but are encoded from 0 to 1023. The data byte written will also be
833 // verified. The packet sequence follows the DCC standard. We will send the CV byte write packet four times,
834 // send out several RESET packets and the send the verify packets to get the acknowledge from the decoder that
835 // the operation was successful.
836 //
837 // ??? This command may take a long time, a lot of packets are sent. While this not an issue with the signal
838 // generation, which is done via interrupt handlers, it may be an issue with any other work of the base
839 // station. This code needs to be redesigned to use a kind of state machine that sends a packet at a time
840 // so other work can interleave.
841 //-----------------------------------------------------------------------------------------------------------
842 uint8_t LcsBaseStationLocoSession::writeCVByte( uint16_t cvId, uint8_t val ) {
843
844     if ( ! ( progTrack -> isServiceModeOn( ))) return ( ERR_NO_SVC_MODE );
845
846     if ( ! validCvId( cvId )) return ( ERR_INVALID_CV_ID );
847     cvId--;
848
849     uint8_t pBuf[ MAX_DCC_PACKET_SIZE ];
850     int     base = progTrack -> decoderAckBaseline( 5 );
851
852     pBuf[0] = 0x7C + ( highByte( cvId ) & 0x03 );
853     pBuf[1] = lowByte( cvId );
854     pBuf[2] = val;
855
856     progTrack -> loadPacket( pBuf, 3, 4 );
857     progTrack -> loadPacket( resetDccPacketData, 2, 11 );
858
859     pBuf[0] = 0x74 + ( highByte( cvId ) & 0x03 );
860     progTrack -> loadPacket( pBuf, 3, 5 );
861
862     return (( progTrack -> decoderAckDetect( base, 9 )) ? ALL_OK : (LcsErrorCodes) ERR_CV_OP_FAILED );
863 }
864
865 //-----------------------------------------------------------------------------------------------------------
866 // "writeCVBit" puts a data bit into the CV on the decoder. This function is only available in session mode.
867 // The CV numbers range from 1 to 1024, but are encoded from 0 to 1023. For the bit mode,  the "val" parameter
868 // encodes the bit position in bits 0 - 2 and the bit value in bit 3. The packet sequence follows the DCC
869 // standard, similar to the byte write operation.
870 //
871 // ??? This command may take a long time, a lot of packets are sent. While this not an issue with the signal
872 // generation, which is done via interrupt handlers, it may be an issue with any other work of the base
873 // station. This code needs to be redesigned to use a kind of state machine that sends a packet at a time
874 // so other work can interleave.
875 //-----------------------------------------------------------------------------------------------------------
876 uint8_t LcsBaseStationLocoSession::writeCVBit( uint16_t cvId, uint8_t bitPos, uint8_t val ) {
877
878     if ( ! ( progTrack -> isServiceModeOn( ))) return ( ERR_NO_SVC_MODE );
879     if ( ! validCvId( cvId )) return ( ERR_INVALID_CV_ID );
880     cvId--;
881
882     uint8_t pBuf[ MAX_DCC_PACKET_SIZE ];
883     int     base = progTrack -> decoderAckBaseline( 5 );
884
885     pBuf[0] = 0x78 + ( highByte( cvId ) & 0x03 );
886     pBuf[1] = lowByte( cvId );
887     pBuf[2] = 0xF0 + (( val % 2 ) * 8 ) + ( bitPos % 8 );
888
889     progTrack -> loadPacket( pBuf, 3, 4 );
890     progTrack -> loadPacket( resetDccPacketData, 2, 11 );
```

```
891
892     bitWrite( &pBuf[2], 4, false );
893     progTrack -> loadPacket( pBuf, 3, 5 );
894
895     return (( progTrack -> decoderAckDetect( base, 9 )) ? ALL_OK : (LcsErrorCodes) ERR_CV_OP_FAILED );
896 }
897
898 //--------------------------------------------------------------------------------------------------------------
899 // "writeDccPacketMain" just load the DCC packet into the buffer and out it goes to the main track without
900 // any further checks.
901 //
902 //--------------------------------------------------------------------------------------------------------------
903 uint8_t LcsBaseStationLocoSession::writeDccPacketMain( uint8_t *pBuf, uint8_t pLen, uint8_t nRepeat ) {
904
905     if ( ! validDccPacketlen( pLen )) return ( ERR_INVALID_PACKET_LEN );
906     if ( ! validDccPacketRepeatCnt( nRepeat )) return ( ERR_INVALID_REPEATS );
907
908     mainTrack -> loadPacket( pBuf, pLen, nRepeat );
909     return ( ALL_OK );
910 }
911
912 //--------------------------------------------------------------------------------------------------------------
913 // "writeDccPacketProg" just load the DCC packet into the buffer and out it goes to the programming track
914 // without any further checks.
915 //
916 //--------------------------------------------------------------------------------------------------------------
917 uint8_t LcsBaseStationLocoSession::writeDccPacketProg( uint8_t *pBuf, uint8_t pLen, uint8_t nRepeat ) {
918
919     if ( ! validDccPacketlen( pLen )) return ( ERR_INVALID_PACKET_LEN );
920     if ( ! validDccPacketRepeatCnt( nRepeat )) return ( ERR_INVALID_REPEATS );
921
922     progTrack -> loadPacket( pBuf, pLen, nRepeat );
923     return ( ALL_OK );
924 }
925
926 //--------------------------------------------------------------------------------------------------------------
927 // "allocateSessionEntry" allocates a new loco session entry and returns a pointer to the entry. We first
928 // check if there is already a session for the cabId and if so, we return a null pointer. If not, we try to
929 // find a free entry and if that fails try to raise the high water mark. If that fails, we are out of luck
930 // and return a null pointer.
931 //
932 //--------------------------------------------------------------------------------------------------------------
933 SessionMapEntry* LcsBaseStationLocoSession::allocateSessionEntry( uint16_t cabId ) {
934
935     if ( lookupSessionEntry( cabId ) != nullptr ) return ( nullptr );
936
937     SessionMapEntry *freePtr = lookupSessionEntry( NIL_CAB_ID );
938
939     if (( freePtr == nullptr ) && ( sessionMapHwm < sessionMapLimit )) freePtr = sessionMapHwm ++;
940
941     if ( freePtr != nullptr ) {
942
943         initSessionEntry( freePtr );
944         freePtr -> cabId  = cabId;
945         freePtr -> flags  |= SME_ALLOCATED;
946
947         if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_SESSION )) {
948
949             printf( "Allocate session entry: %d, HWM: %d\n",
950                 ( freePtr - sessionMap + 1 ), ( sessionMapHwm - sessionMap ));
951         }
952     }
953
954     return ( freePtr );
955 }
956
957 //--------------------------------------------------------------------------------------------------------------
958 // "deallocateSessionEntry" is the counterpart to the entry allocation. We just free up the entry. If the
959 // entry is at the high water mark, we try to free up all possibly free entries from the high water mark
960 // downward, decrementing the high water mark. This way the high water mark shrinks again and we do not need
961 // to work through unused entries in the middle.
962 //
963 //--------------------------------------------------------------------------------------------------------------
964 void LcsBaseStationLocoSession::deallocateSessionEntry( SessionMapEntry *smePtr ) {
965
966     if (( smePtr != nullptr ) && ( smePtr >= sessionMap ) && ( smePtr < sessionMapHwm )) {
967
968         if ( smePtr == ( sessionMapHwm - 1 )) {
969
970             do {
971
972                 initSessionEntry( smePtr );
973                 smePtr --;
974             }
975             while (( smePtr -> cabId == NIL_CAB_ID ) && ( smePtr >= sessionMap ));
976
977             sessionMapHwm = smePtr + 1;
978         }
979         else initSessionEntry( smePtr );
980
981        if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_SESSION )) {
982
983             printf( "Released Session, sId: %d, ,new  HWM: %d\n",
984                 ( smePtr - sessionMap + 1 ), ( sessionMapHwm - sessionMap ));
985        }
986     }
987 }
988
989 //--------------------------------------------------------------------------------------------------------------
```

```
990  // "lookupSessionEntry" scans the session map for a session entry for the cabId. If none is found, a nullptr
991  // is returned. Note that a NIL_CAB_ID as argument is also a valid input and will return the first free entry.
992  //
993  //----------------------------------------------------------------------------------------------------------------
994  SessionMapEntry *LcsBaseStationLocoSession::lookupSessionEntry( uint16_t cabId ) {
995
996      SessionMapEntry *smePtr = sessionMap;
997
998      while ( smePtr < sessionMapHwm ) {
999
1000         if ( smePtr -> cabId == cabId ) return ( smePtr );
1001         else smePtr ++;
1002     }
1003
1004     return ( nullptr );
1005 }
1006
1007 //----------------------------------------------------------------------------------------------------------------
1008 // "initSessionEntry" initializes a session map entry with default values.
1009 //
1010 //----------------------------------------------------------------------------------------------------------------
1011 void LcsBaseStationLocoSession::initSessionEntry( SessionMapEntry *smePtr ) {
1012
1013     smePtr -> flags              = SME_DEFAULT_SETTING;
1014     smePtr -> cabId              = NIL_CAB_ID;
1015     smePtr -> speedSteps         = DCC_SPEED_STEPS_128;
1016     smePtr -> speed              = 0;
1017     smePtr -> direction          = 0;
1018     smePtr -> engineState        = 0;
1019     smePtr -> lastKeepAliveTime  = 0;
1020     smePtr -> nextRefreshStep    = 0;
1021
1022     for ( int i = 0; i < MAX_DCC_FUNC_GROUP_ID; i++ ) smePtr -> functions[ i ] = 0;
1023 }
1024
1025 //----------------------------------------------------------------------------------------------------------------
1026 // "getSessionMapEntryPtr" returns a pointer to a valid and used sessionMap entry. The sessionId starts with
1027 // index 1.
1028 //
1029 //----------------------------------------------------------------------------------------------------------------
1030 SessionMapEntry *LcsBaseStationLocoSession::getSessionMapEntryPtr( uint8_t sId ) {
1031
1032     if ( ! isInRangeU( sId, MIN_LOCO_SESSION_ID, ( sessionMapHwm - sessionMap ))) return ( nullptr );
1033     return (( sessionMap[ sId - 1 ].cabId == NIL_CAB_ID ) ? nullptr : &sessionMap[ sId - 1 ] );
1034 }
1035
1036 //----------------------------------------------------------------------------------------------------------------
1037 // "printSessionMapConfig" lists cab session map configuration data.
1038 //
1039 //----------------------------------------------------------------------------------------------------------------
1040 void LcsBaseStationLocoSession::printSessionMapConfig( ) {
1041
1042     printf( "Session Map Config\n" );
1043     printf( " Options: 0x%x\n", options );
1044     printf( " Session Map Size: %d\n", ( sessionMapLimit - sessionMap ));
1045 }
1046
1047 //----------------------------------------------------------------------------------------------------------------
1048 // "printSessionMapInfo" lists the cab session map data.
1049 //
1050 //----------------------------------------------------------------------------------------------------------------
1051 void LcsBaseStationLocoSession::printSessionMapInfo( ) {
1052
1053     printf( "Session Map Info\n" );
1054
1055     printf( " Flags: 0x%x\n", flags );
1056
1057     // ??? decode the flags ?  e.g. "[ f f f f ]"
1058
1059     printf( " Session Map Hwm: %d\n", ( sessionMapHwm - sessionMap ));
1060
1061     for ( SessionMapEntry *smePtr = sessionMap; smePtr < sessionMapHwm; smePtr ++ ) {
1062
1063         if ( smePtr -> cabId != NIL_CAB_ID ) printSessionEntry( smePtr );
1064     }
1065
1066      printf( "\n" );
1067 }
1068
1069 //----------------------------------------------------------------------------------------------------------------
1070 // "printSessionEntry" lists a cab session.
1071 //
1072 //----------------------------------------------------------------------------------------------------------------
1073 void LcsBaseStationLocoSession::printSessionEntry( SessionMapEntry *smePtr ) {
1074
1075   if ( smePtr != nullptr ) {
1076
1077     printf( " sId: %d, cabId: %d, speed: %d ", ( smePtr - sessionMap + 1 ), smePtr -> cabId, smePtr -> speed );
1078
1079     printf( "%s", (( smePtr -> direction ) ? "Rev" : "Fwd" ));
1080     printf( ", functions: " );
1081
1082     for ( uint8_t i = 0; i < MAX_DCC_FUNC_GROUP_ID; i++ ) {
1083
1084       printf( " 0x%x ", smePtr -> functions[ i ] );
1085     }
1086
1087     printf( " Flags: 0x%x", ( smePtr -> flags ));
1088
```

```
1089      // ??? decode the flags ? e.g. "[ f f f f ]"
1090    }
1091
1092    printf( "\n" );
1093  }
```

```
1  //--------------------------------------------------------------------------------------------------------------
2  //
3  // LCS - Base Station
4  //
5  //--------------------------------------------------------------------------------------------------------------
6  // This is the main program for the LCS base station. Every layout would need at least a base station. Its
7  // primary task is to manage the DCC loco sessions, generate the DCC signals and manage the dual DCC track
8  // power outputs.
9  //
10 // Like all other LcsNodes, the base station will provide a rich set of variable that can be set and queried.
11 // In addition, the base features a command line extension which implements the DCC++ style commands and
12 // some more base station specific commands. The idea for the DCC++ command syntax and commands is that these
13 // command can also be submitted by a third party software ( e.g. JMRI ). An example would be the JMRI CV
14 // programming tool.
15 //
16 // ??? we need an idea of system time like DCC. To be broadcasted periodically.
17 // ??? we also need a broadcast of the layout system capabilities....
18 //
19 //--------------------------------------------------------------------------------------------------------------
20 //
21 // LCS - Controller Dependent Code - Raspberry PI Pico Implementation
22 // Copyright (C) 2022 - 2024 Helmut Fieres
23 //
24 // This program is free software: you can redistribute it and/or modify it under the terms of the GNU General
25 // Public License as published by the Free Software Foundation, either version 3 of the License, or (at your
26 // option) any later version.
27 //
28 // This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
29 // implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
30 // for more details.
31 //
32 // You should have received a copy of the GNU General Public License along with this program. If not, see
33 // http://www.gnu.org/licenses
34 //
35 //  GNU General Public License:  http://opensource.org/licenses/GPL-3.0
36 //
37 //--------------------------------------------------------------------------------------------------------------
38 #include "LcsCdcLib.h"
39 #include "LcsRuntimeLib.h"
40 #include "LcsBaseStation.h"
41
42 using namespace LCS;
43
44 //--------------------------------------------------------------------------------------------------------------
45 // Base station global data.
46 //
47 // ??? can the objects for track and session just use these variables instead of keeping them locally as a
48 // field ?
49 //--------------------------------------------------------------------------------------------------------------
50 uint16_t                        debugMask;
51 CDC::CdcConfigDesc              cdcConfig;
52 LCS::LcsConfigDesc              lcsConfig;
53 LcsBaseStationCommand           serialCmd;
54 LcsBaseStationDccTrack          mainTrack;
55 LcsBaseStationDccTrack          progTrack;
56 LcsBaseStationLocoSession       locoSessions;
57 LcsBaseStationMsgInterface      msgInterface;
58
59 //--------------------------------------------------------------------------------------------------------------
60 // Setup the configuration of the HW board. The CDC config contains the HW pin mapping. The dual bridge pins
61 // for enabling the bridge and controlling its direction. The pins are mapped to the CDC pin names DIO2 to
62 // DIO7 as show below. DIO-0 and DIO-1 are routed to the extension connector board.
63 //
64 //      cdcConfig.DIO_PIN_0     -> DIO-0
65 //      cdcConfig.DIO_PIN_1     -> DIO-1
66 //      cdcConfig.DIO_PIN_2     -> Main dcc1
67 //      cdcConfig.DIO_PIN_3     -> Main dcc2
68 //      cdcConfig.DIO_PIN_4     -> Prog ddc1
69 //      cdcConfig.DIO_PIN_5     -> Prog ddc2
70 //      cdcConfig.DIO_PIN_6     -> Main enable
71 //      cdcConfig.DIO_PIN_7     -> Prog enable
72 //
73 // Current mapping: Main Controller Board B.01.00 - PICO - newest version.
74 //
75 //      cdcConfig.DIO_PIN_0     = 8;
76 //      cdcConfig.DIO_PIN_1     = 12;
77 //      cdcConfig.DIO_PIN_2     = 21;
78 //      cdcConfig.DIO_PIN_3     = 20;
79 //      cdcConfig.DIO_PIN_4     = 19;
80 //      cdcConfig.DIO_PIN_5     = 18;
81 //      cdcConfig.DIO_PIN_6     = 6;
82 //      cdcConfig.DIO_PIN_7     = 7;
83 //
84 // In addition, the HW pins for I2C, analog inputs and so on are set. Check the schematic for the board
85 // to see all pin assign,ents.
86 //
87 // ??? one day we will have several base station versions. Although they will perhaps differ, their the CDC
88 // pin names used should not change. But we would need to come up with an idea which configuration to use
89 // when preparing an image for the base station board.
90 //--------------------------------------------------------------------------------------------------------------
91 void setupConfigInfo( ) {
92
93     cdcConfig = CDC::getConfigDefault( );
94     lcsConfig = LCS::getConfigDefault( );
95
96     cdcConfig.ADC_PIN_0             = 26;
97     cdcConfig.ADC_PIN_1             = 27;
98
```

```
99      cdcConfig.PFAIL_PIN             = 5;
100     cdcConfig.EXT_INT_PIN           = 22;
101     cdcConfig.READY_LED_PIN         = 14;
102     cdcConfig.ACTIVE_LED_PIN        = 15;
103
104     cdcConfig.DIO_PIN_0             = 8;
105     cdcConfig.DIO_PIN_1             = 12;
106     cdcConfig.DIO_PIN_2             = 21;
107     cdcConfig.DIO_PIN_3             = 20;
108     cdcConfig.DIO_PIN_4             = 19;
109     cdcConfig.DIO_PIN_5             = 18;
110     cdcConfig.DIO_PIN_6             = 6;
111     cdcConfig.DIO_PIN_7             = 7;
112
113     cdcConfig.UART_RX_PIN_1         = 13;
114     cdcConfig.UART_RX_PIN_2         = 9;
115
116     cdcConfig.NVM_I2C_SCL_PIN       = 3;
117     cdcConfig.NVM_I2C_SDA_PIN       = 2;
118     cdcConfig.NVM_I2C_ADR_ROOT      = 0x50;
119
120     cdcConfig.EXT_I2C_SCL_PIN       = 17;
121     cdcConfig.EXT_I2C_SDA_PIN       = 16;
122     cdcConfig.EXT_I2C_ADR_ROOT      = 0x50;
123
124     cdcConfig.CAN_BUS_RX_PIN        = 0;
125     cdcConfig.CAN_BUS_TX_PIN        = 1;
126     cdcConfig.CAN_BUS_CTRL_MODE     = CAN_BUS_LIB_PICO_PIO_125K_M_CORE;
127     cdcConfig.CAN_BUS_DEF_ID        = 100;
128
129     cdcConfig.NODE_NVM_SIZE         = 8192;
130     cdcConfig.EXT_NVM_SIZE          = 4096;
131
132     lcsConfig.options               |= NOPT_SKIP_NODE_ID_CONFIG;
133 }
134
135 //-----------------------------------------------------------------------------------------------------------
136 // Some little helper functions.
137 //
138 //-----------------------------------------------------------------------------------------------------------
139 void printLcsMsg( uint8_t *msg ) {
140
141   int msgLen = (( msg[0] >> 5 ) + 1 ) % 8;
142
143   for ( int i = 0; i < msgLen; i++ ) printf( "0x%x ", msg[i] );
144   printf( "\n" );
145 }
146
147 uint8_t printStatus (uint8_t status ) {
148
149   printf( "Status: " );
150   if ( status == LCS::ALL_OK ) printf( "OK\n" );
151   else printf ( "FAILED: %d\n", status );
152   return ( status );
153 }
154
155 //-----------------------------------------------------------------------------------------------------------
156 // The node and port initialization callback.
157 //
158 // ??? when we know what ports we actually need / use, disable the rest of the ports.
159 //-----------------------------------------------------------------------------------------------------------
160 uint8_t lcsInitCallback( uint16_t npId ) {
161
162     switch ( npId & 0xF ) {
163
164         case 0:     printf( "Node Init Callback: 0x%x\n", npId >> 4     ); break;
165         default:    printf( "Port Init Callback: 0x%x\n", npId &  0xF    );
166     }
167
168     return( ALL_OK );
169 }
170
171 //-----------------------------------------------------------------------------------------------------------
172 // The node or port reset callback.
173 //
174 //-----------------------------------------------------------------------------------------------------------
175 uint8_t lcsResetCallback( uint16_t npId ) {
176
177     switch ( npId & 0xF ) {
178
179         case 0:     printf( "Node Reset Callback: 0x%x\n", npId >> 4     ); break;
180         default:    printf( "Port Reset Callback: 0x%x\n", npId &  0xF    );
181     }
182
183     return( ALL_OK );
184 }
185
186 //-----------------------------------------------------------------------------------------------------------
187 // The node or port power fail callback.
188 //
189 //-----------------------------------------------------------------------------------------------------------
190 uint8_t lcsPfailCallback( uint16_t npId ) {
191
192     switch ( npId & 0xF ) {
193
194         case 0:     printf( "Node Power Fail Callback: 0x%x\n", npId >> 4     ); break;
195         default:    printf( "Port Power Fail Callback: 0x%x\n", npId &  0xF    );
196     }
197
```

```
198     return( ALL_OK );
199 }
200 
201 //-----------------------------------------------------------------------------------------------------------
202 // The base station has also a command line interpreter. The callback is invoked by the core library when
203 // there is a command that it does not handle.
204 //
205 //-----------------------------------------------------------------------------------------------------------
206 uint8_t lcsCmdCallback( char *cmdLine ) {
207 
208     serialCmd.handleSerialCommand( cmdLine );
209     return( ALL_OK );
210 }
211 
212 //-----------------------------------------------------------------------------------------------------------
213 // Other LCS message callbacks. All we do is to list their invocation. ( for now )
214 //
215 //-----------------------------------------------------------------------------------------------------------
216 uint8_t lcsMsgCallback( uint8_t *msg ) {
217 
218     printf( "MsgCallback: ", msg  );
219 
220     for ( int i = 0; i < 8; i++ ) printf( "0x%2x ");
221     printf( "\n" );
222     return( ALL_OK );
223 }
224 
225 //-----------------------------------------------------------------------------------------------------------
226 // The LCS core library ends in a loop that manages its internal workings, invoking the callbacks where
227 // needed. One set of callbacks are the periodic tasks. The base station needs to periodically run the DCC
228 // track state machine for power consumption measurement and so on.  Another periodic task is to refresh the
229 // active locomotive session entries.
230 //
231 //-----------------------------------------------------------------------------------------------------------
232 uint8_t bsMainTrackCallback( ) {
233 
234     mainTrack.runDccTrackStateMachine( );
235     return( ALL_OK );
236 }
237 
238 uint8_t bsProgTrackCallback( ) {
239 
240     progTrack.runDccTrackStateMachine( );
241     return( ALL_OK );
242 }
243 
244 uint8_t bsRefreshActiveSessionCallback( ) {
245 
246     locoSessions.refreshActiveSessions( );
247     return( ALL_OK );
248 }
249 
250 //-----------------------------------------------------------------------------------------------------------
251 // When the base station node receives a request with an item defined in the user item range or the base
252 // station itself issues such a request, the defined callback is invoked.
253 //
254 //-----------------------------------------------------------------------------------------------------------
255 uint8_t lcsReqCallback( uint8_t npId, uint8_t item, uint16_t *arg1, uint16_t *arg2 ) {
256 
257     printf( "REQ callback: npId: 0x%x, item: %d", npId, item );
258     if ( arg1 != nullptr ) printf( ", arg1: %d, ", *arg1 ); else printf( ", arg1: null" );
259     if ( arg2 != nullptr ) printf( ", arg2: %d, ", *arg2 ); else printf( ", arg2: null" );
260     return( ALL_OK );
261 }
262 
263 //-----------------------------------------------------------------------------------------------------------
264 // When the base station gets a reply message for a request previously sent, this callback is invoked.
265 //
266 //-----------------------------------------------------------------------------------------------------------
267 uint8_t lcsRepCallback( uint8_t npId, uint8_t item, uint16_t arg1, uint16_t arg2, uint8_t ret ) {
268 
269     printf( "REP callback: npId: 0x%x, item: %d, arg1: %d, arg2: %d, ret: %d ", npId, item , arg1, arg2, ret );
270     return( ALL_OK );
271 }
272 
273 //-----------------------------------------------------------------------------------------------------------
274 // For any event on the LCS system that the base station is interested in, this callback is invoked.
275 //
276 //-----------------------------------------------------------------------------------------------------------
277 uint8_t lcsEventCallback( uint16_t npId, uint16_t eId, uint8_t eAction, uint16_t eData ) {
278 
279     printf( "Event: npId: 0x%x, eId: %d, eAction: %d, eData: %d\n", npId, eId, eAction, eData );
280     return( ALL_OK );
281 }
282 
283 //-----------------------------------------------------------------------------------------------------------
284 // Init the Runtime.
285 //
286 //-----------------------------------------------------------------------------------------------------------
287 uint8_t initLcsRuntime( ) {
288 
289     setupConfigInfo( );
290 
291     uint8_t rStat = LCS::initRuntime( &lcsConfig, &cdcConfig );
292     printf( "LCS Base Station\n" );
293 
294     CDC::printConfigInfo( &cdcConfig );
295 
296     printStatus( rStat );
```

```
297     return( rStat );
298 }
299 
300 //--------------------------------------------------------------------------------------------------------
301 // This routine initializes the Loco Session Map Object.
302 //
303 //--------------------------------------------------------------------------------------------------------
304 uint8_t setupLocoSessions( ) {
305 
306   LcsBaseStationSessionMapDesc sessionDesc;
307 
308   sessionDesc.options     = SM_OPT_ENABLE_REFRESH;
309   sessionDesc.maxSessions = 16;
310 
311   printf( "Setup Session Map -> " );
312   return ( printStatus( locoSessions.setupSessionMap( &sessionDesc, &mainTrack, &progTrack )));
313 }
314 
315 //--------------------------------------------------------------------------------------------------------
316 // This routine initializes the MAIN track object.
317 //
318 // ??? define constants such as: SENSE_OR1_OPAMP_11 to set the milliVolts per Amp.
319 //--------------------------------------------------------------------------------------------------------
320 int setupDccTrackMain( ) {
321 
322   LcsBaseStationTrackDesc mainTrackDesc;
323 
324   mainTrackDesc.options                    = DT_OPT_RAILCOM | DT_OPT_CUTOUT;
325 
326   mainTrackDesc.enablePin                  = cdcConfig.DIO_PIN_6;
327   mainTrackDesc.dccSigPin1                 = cdcConfig.DIO_PIN_2;
328   mainTrackDesc.dccSigPin2                 = cdcConfig.DIO_PIN_3;
329   mainTrackDesc.sensePin                   = cdcConfig.ADC_PIN_0;
330   mainTrackDesc.uartRxPin                  = cdcConfig.UART_RX_PIN_1;
331 
332   mainTrackDesc.initCurrentMilliAmp        = 500;
333   mainTrackDesc.limitCurrentMilliAmp       = 1500;
334   mainTrackDesc.maxCurrentMilliAmp         = 2000;
335   mainTrackDesc.milliVoltPerAmp            = 100 * 11;  // ??? opAmp has Factor eleven ...
336   mainTrackDesc.startTimeThresholdMillis   = 1000;
337   mainTrackDesc.stopTimeThresholdMillis    = 500;
338   mainTrackDesc.overloadTimeThresholdMillis = 500;
339   mainTrackDesc.overloadEventThreshold     = 10;
340   mainTrackDesc.overloadRestartThreshold   = 5;
341 
342   printf( "Setup MAIN track -> " );
343   return ( printStatus( mainTrack.setupDccTrack( &mainTrackDesc )));
344 }
345 
346 //--------------------------------------------------------------------------------------------------------
347 // This routine initializes the PROG track object.
348 //
349 // ??? define constants such as: SENSE_OR1_OPAMP_11 to set the milliVolts per Amp.
350 //--------------------------------------------------------------------------------------------------------
351 uint8_t setupDccTrackProg( ) {
352 
353   LcsBaseStationTrackDesc progTrackDesc;
354 
355   progTrackDesc.options                    = DT_OPT_SERVICE_MODE_TRACK;
356 
357   progTrackDesc.enablePin                  = cdcConfig.DIO_PIN_7;
358   progTrackDesc.dccSigPin1                 = cdcConfig.DIO_PIN_4;
359   progTrackDesc.dccSigPin2                 = cdcConfig.DIO_PIN_5;
360   progTrackDesc.sensePin                   = cdcConfig.ADC_PIN_1;
361   progTrackDesc.uartRxPin                  = cdcConfig.UART_RX_PIN_2;
362 
363   progTrackDesc.initCurrentMilliAmp        = 500;
364   progTrackDesc.limitCurrentMilliAmp       = 500;
365   progTrackDesc.maxCurrentMilliAmp         = 1000;
366   progTrackDesc.milliVoltPerAmp            = 100 * 11;  // ??? opAmp has Factor eleven ...
367   progTrackDesc.startTimeThresholdMillis   = 1000;
368   progTrackDesc.stopTimeThresholdMillis    = 500;
369   progTrackDesc.overloadTimeThresholdMillis = 500;
370   progTrackDesc.overloadEventThreshold     = 10;
371   progTrackDesc.overloadRestartThreshold   = 5;
372 
373   printf( "Setup PROG track -> " );
374   return ( printStatus( progTrack.setupDccTrack( &progTrackDesc )));
375 }
376 
377 //--------------------------------------------------------------------------------------------------------
378 // The base station has also a command interpreter, primarily for the DCC++ commands.
379 //
380 //--------------------------------------------------------------------------------------------------------
381 uint8_t setupSerialCommand( ) {
382 
383   printf( "Setup Serial Command -> " );
384   return ( printStatus( serialCmd.setupSerialCommand( &locoSessions, &mainTrack, &progTrack )));
385 }
386 
387 //--------------------------------------------------------------------------------------------------------
388 // The LCS message interface is initialized in the LCS core library. This routine will set up the receiver
389 // handler for incoming LCS message that concern the base station.
390 //
391 //--------------------------------------------------------------------------------------------------------
392 uint8_t setupMsgInterface( ) {
393 
394   printf( "Setup LCS Msg Interface -> " );
395   return ( printStatus( msgInterface.setupLcsMsgInterface( &locoSessions, &mainTrack, &progTrack )));
```

```
396 }
397
398 //------------------------------------------------------------------------------------------------------------
399 // After the initial setup of the runtime library, the callback are registered.
400 //
401 //------------------------------------------------------------------------------------------------------------
402 uint8_t registerCallbacks( ) {
403
404     printf( "Registering Callbacks\n" );
405
406     registerLcsMsgCallback( lcsMsgCallback );
407     registerCmdCallback( lcsCmdCallback );
408     registerInitCallback( lcsInitCallback );
409     registerResetCallback( lcsResetCallback );
410     registerPfailCallback( lcsPfailCallback );
411     registerReqCallback( lcsReqCallback );
412     registerRepCallback( lcsRepCallback );
413     registerEventCallback( lcsEventCallback );
414     registerTaskCallback( bsMainTrackCallback, MAIN_TRACK_STATE_TIME_INTERVAL );
415     registerTaskCallback( bsProgTrackCallback, PROG_TRACK_STATE_TIME_INTERVAL );
416     registerTaskCallback( bsRefreshActiveSessionCallback, SESSION_REFRESH_TASK_INTERVAL );
417
418     return( ALL_OK );
419 }
420
421 //------------------------------------------------------------------------------------------------------------
422 // Fire up the base station. First all base station modules are initialized. If this is OK, the DCC tack
423 // signal generation is enabled, i.e. the interrupt driven DCC packet broadcasting starts. Finally, the
424 // track power is turned on and we give control to the LCS runtime for processing events and requests.
425 //
426 //------------------------------------------------------------------------------------------------------------
427 uint8_t startBaseStation( ) {
428
429     uint8_t rStat = ALL_OK;
430
431     if ( rStat == ALL_OK ) rStat = setupSerialCommand( );
432     if ( rStat == ALL_OK ) rStat = setupMsgInterface( );
433     if ( rStat == ALL_OK ) rStat = setupLocoSessions( );
434     if ( rStat == ALL_OK ) rStat = setupDccTrackMain( );
435     if ( rStat == ALL_OK ) rStat = setupDccTrackProg( );
436
437     if ( rStat == ALL_OK ) {
438
439         LcsBaseStationDccTrack::startDccProcessing( );
440
441         mainTrack.powerStart( );
442         progTrack.powerStart( );
443
444         // ??? bracket so that it is not printed when no console...
445         mainTrack.printDccTrackStatus( );
446         progTrack.printDccTrackStatus( );
447         printf( "Ready...\n" );
448
449         startRuntime( );
450   }
451
452   return( ALL_OK );
453 }
454
455 //------------------------------------------------------------------------------------------------------------
456 // The main program. Setup the runtime, register the callbacks, and get the show on the road.
457 //
458 //------------------------------------------------------------------------------------------------------------
459 int main( ) {
460
461     uint8_t rStat = ALL_OK;
462
463     if ( rStat == ALL_OK ) rStat = initLcsRuntime( );
464     if ( rStat == ALL_OK ) rStat = registerCallbacks( );
465     if ( rStat == ALL_OK ) return( startBaseStation( ));
466 }
```

B.2 CDC Lib

```
//-----------------------------------------------------------------------------------------------------------------
//
// LCS - Controller Dependent Code - Include file
//
//-----------------------------------------------------------------------------------------------------------------
// The controller dependent code layer concentrates all processor dependent code into one library. The idea
// is twofold. First, there needs to be a way to isolate the controller specific hardware from the LCS runtime
// Library as well as the extension module firmware. The Raspberry PI Pico offers a C++ SDK with a set of
// libraries to invoke the desired function rather than access to registers.The Pico also offers a great
// flexibility of pin assignment for the hardware IO functions. Second, within the hardware IO boundaries of
// the controller family the individual hardware pin assignment used may vary from board to board design.
// Nevertheless, the Extension Connector layout and basic functions available should be the same for all
// controllers used. For the upper software layers, the CDC library offers a structured way to describe
// the possible pins assignments.
//
// Note that this layer is not a generic HW abstraction. The layer is very specific to the LCS controller
// boards described in the book. Nevertheless, some pins can vary, depending on the board version. Currently,
// only the Raspberry PI Pico Board is supported.
//
//-----------------------------------------------------------------------------------------------------------------
//
// LCS - Controller Dependent Code - Include file
// Copyright (C) 2022 - 2024  Helmut Fieres
//
// This program is free software: you can redistribute it and/or modify it under the terms of the GNU General
// Public License as published by the Free Software Foundation, either version 3 of the License, or (at your
// option) any later version.
//
// This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
// implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
// for more details.
//
// You should have received a copy of the GNU General Public License along with this program. If not, see
// http://www.gnu.org/licenses
//
//  GNU General Public License:  http://opensource.org/licenses/GPL-3.0
//
//-----------------------------------------------------------------------------------------------------------------
#ifndef LcsCdcLib_h
#define LcsCdcLib_h

//-----------------------------------------------------------------------------------------------------------------
// Include files.
//
//-----------------------------------------------------------------------------------------------------------------
#include <stdio.h>
#include <stdint.h>
#include <cstring>

//-----------------------------------------------------------------------------------------------------------------
// All definitions and functions are in the CDC name space.
//
//-----------------------------------------------------------------------------------------------------------------
namespace CDC {

//-----------------------------------------------------------------------------------------------------------------
// Error status codes. The errors are used when setting up the Hal library. During operation, all routines
// validate the input for correctness. If they are not correct, the call is simply not performed and an
// error is returned.
//
// ??? clean up a little ... what is really needed ?
//-----------------------------------------------------------------------------------------------------------------
enum CdcStatus : uint8_t {

    NO_ERR              = 0,
    INIT_PENDING        = 1,
    NOT_SUPPORTED       = 2,
    NOT_IMPLEMENTED     = 3,

    MEM_SIZE_ERR        = 10,

    READY_LED_PIN_ERR   = 12,
    ACTIVE_LED_PIN_ERR  = 13,
    BUTTON_PIN_ERR      = 14,
    PFAIL_PIN_ERR       = 15,
    EXT_INT_PIN_ERR     = 16,
    DIO_PIN_ERR         = 17,
    ADC_PIN_ERR         = 18,
    PWM_PIN_ERR         = 19,

    UART_PORT_ERR       = 20,
    UART_CONFIG_ERR     = 21,
    UART_WRITE_ERR      = 22,
    UAT_READ_ERR        = 23,

    SPI_PORT_ERR        = 25,
    SPI_CONFIG_ERR      = 26,
    SPI_WRITE_ERR       = 27,
    SPI_READ_ERR        = 28,

    I2C_PORT_ERR        = 30,
    I2C_CONFIG_ERR      = 31,
    I2C_WRITE_ERR       = 32,
    I2C_READ_ERR        = 33
```

```
95
96  };
97
98  //-----------------------------------------------------------------------------------------------------------
99  // Controller pin related definitions. A pin can be valid, undefined or illegal. An undefined pin for a pin
100 // field in the configuration structure indicates that  the pin has not been used by the firmware
101 // implementation but is a pin that the particular controller would support. An illegal pin means that the
102 // pin is not offered by this controller and cannot be assigned at all.
103 //
104 //-----------------------------------------------------------------------------------------------------------
105 const uint8_t UNDEFINED_PIN    = 255;
106 const uint8_t ILLEGAL_PIN      = 254;
107
108 //-----------------------------------------------------------------------------------------------------------
109 // The controller families. Currently, there is only the Raspberry PI Pico models.
110 //
111 //-----------------------------------------------------------------------------------------------------------
112 enum ControllerFamily : uint8_t {
113
114     CF_UNDEFINED     = 0,
115     CF_RP_PICO       = 1
116 };
117
118 //-----------------------------------------------------------------------------------------------------------
119 // DIO pin related definitions. A digital pin can be an input pin, with or without pull-up, or an output
120 // pin. DIO pins can also be associated with an interrupt handler. The handler itself is mapped to an edge
121 // or level event.
122 //
123 //-----------------------------------------------------------------------------------------------------------
124 enum dioMode : uint8_t {
125
126     IN             = 0,
127     OUT            = 1,
128     IN_PULLUP      = 2
129 };
130
131 //-----------------------------------------------------------------------------------------------------------
132 // GPIO interrupts are detected as level change or edge changes.
133 //
134 //-----------------------------------------------------------------------------------------------------------
135 enum intEventTyp : uint8_t {
136
137     EVT_NONE     = 0,
138     EVT_LOW      = 1,
139     EVT_HIGH     = 2,
140     EVT_FALL     = 3,
141     EVT_RISE     = 4,
142     EVT_CHANGE   = 5
143 };
144
145 //-----------------------------------------------------------------------------------------------------------
146 // The UART modes. There are two implementations. The PICO offers two hardware UARTS. We use them with 8
147 // bits with a parity bit. The second type UART is a software implementation based on the PICO PIO blocks.
148 //
149 //-----------------------------------------------------------------------------------------------------------
150 enum UartMode : uint8_t {
151
152     UART_MODE_UNDEFINED = 0,
153     UART_MODE_8N1       = 1,
154     UART_MODE_8N1_PIO   = 2
155 };
156
157 //-----------------------------------------------------------------------------------------------------------
158 // Callback functions signatures.
159 //
160 //-----------------------------------------------------------------------------------------------------------
161 extern "C" {
162
163     typedef void ( *TimerCallback ) ( uint32_t timerVal );
164     typedef void ( *GpioCallback ) ( uint8_t pin, uint8_t event );
165 }
166
167 //-----------------------------------------------------------------------------------------------------------
168 // CDC features a data structure that records all HW specific pins and flags. The values are set by the
169 // initialization code in a project and are validated. All modules in a project will then just use the
170 // data structure fields using the data for calls to the Hal layer. For example, an application that
171 // uses DIO_PIN_0 and DIO_PIN_1 will set the HW pin numbers of the controller / board combination used
172 // in a config data structure "cfg". A call to write a value to the DIO pin, will then just use
173 // "cfg.DIO_PIN_1" as argument in the "writeDio" call. The "writeDio" call itself will not check the
174 // value of the configured DIO pin, all it will do is to ensure that it is not UNDEFINED. Note that the
175 // structure has more pins defined that a potential controller may have. If so, these fields are set to
176 // UNDEFINED. The structure is the superset of all possible HW items to configure.
177 //
178 // In a later runtime version, we may put this structure as constant data into the non-volatile chip on
179 // the board. It will then just be read from there.
180 //
181 //-----------------------------------------------------------------------------------------------------------
182 struct CdcConfigDesc {
183
184     uint8_t    CFG_STATUS;
185
186     uint8_t    PFAIL_PIN;
187     uint8_t    EXT_INT_PIN;
188     uint8_t    READY_LED_PIN;
189     uint8_t    ACTIVE_LED_PIN;
190
191     uint8_t    DIO_PIN_0;
192     uint8_t    DIO_PIN_1;
193     uint8_t    DIO_PIN_2;
```

```
194     uint8_t   DIO_PIN_3;
195     uint8_t   DIO_PIN_4;
196     uint8_t   DIO_PIN_5;
197     uint8_t   DIO_PIN_6;
198     uint8_t   DIO_PIN_7;
199     uint8_t   DIO_PIN_8;
200     uint8_t   DIO_PIN_9;
201     uint8_t   DIO_PIN_10;
202     uint8_t   DIO_PIN_11;
203     uint8_t   DIO_PIN_12;
204     uint8_t   DIO_PIN_13;
205     uint8_t   DIO_PIN_14;
206     uint8_t   DIO_PIN_15;
207
208     uint8_t   ADC_PIN_0;
209     uint8_t   ADC_PIN_1;
210     uint8_t   ADC_PIN_2;
211     uint8_t   ADC_PIN_3;
212
213     uint8_t   PWM_PIN_0;
214     uint8_t   PWM_PIN_1;
215     uint8_t   PWM_PIN_2;
216     uint8_t   PWM_PIN_3;
217
218     uint8_t   UART_RX_PIN_0;
219     uint8_t   UART_TX_PIN_0;
220
221     uint8_t   UART_RX_PIN_1;
222     uint8_t   UART_TX_PIN_1;
223
224     uint8_t   UART_RX_PIN_2;
225     uint8_t   UART_TX_PIN_2;
226
227     uint8_t   UART_RX_PIN_3;
228     uint8_t   UART_TX_PIN_3;
229
230     uint8_t   SPI_MOSI_PIN_0;
231     uint8_t   SPI_MISO_PIN_0;
232     uint8_t   SPI_SCLK_PIN_0;
233
234     uint8_t   SPI_MOSI_PIN_1;
235     uint8_t   SPI_MISO_PIN_1;
236     uint8_t   SPI_SCLK_PIN_1;
237
238     uint8_t   NVM_I2C_SCL_PIN;
239     uint8_t   NVM_I2C_SDA_PIN;
240     uint8_t   NVM_I2C_ADR_ROOT;
241
242     uint8_t   EXT_I2C_SCL_PIN;
243     uint8_t   EXT_I2C_SDA_PIN;
244     uint8_t   EXT_I2C_ADR_ROOT;
245
246     uint32_t  NODE_NVM_SIZE;
247     uint32_t  EXT_NVM_SIZE;
248
249     uint8_t   CAN_BUS_CTRL_MODE;
250     uint8_t   CAN_BUS_RX_PIN;
251     uint8_t   CAN_BUS_TX_PIN;
252     uint32_t  CAN_BUS_DEF_ID;
253 };
254
255 //-----------------------------------------------------------------------------------------------------------
256 // The routines that make up the hardware abstraction layer. The routines expect hardware pin numbers.
257 // To recap, the CDC layer offers a set of reserved resource names, such as "DIO_PIN_0", which describes
258 // the resource containing the hardware pin and some flags. The configuration routines in this layer will use
259 // these pins and other data stored to configure the hardware. Under the defined resource name name all
260 // upper layers refer to the hardware using the to the configured IO capabilities.
261 //
262 // Complex resources, such as the UART or SPI interface, have more than one HW pin they will use. In this
263 // case one of the HW pins, see the function documentation, will serve as the handle to the resource.
264 //
265 //-----------------------------------------------------------------------------------------------------------
266
267 //-----------------------------------------------------------------------------------------------------------
268 // The console IO functions. We will provide a serial IO via the USB connector of the PICO. The files
269 // need to be linked with the "tinyUSB" library and the cmake file needs to set the option. Then we can
270 // use scanf and printf and so on. In addition, we need  function  that just attempts to read a character
271 // and returns immediately when there is none.
272 //
273 //-----------------------------------------------------------------------------------------------------------
274 uint8_t         configureConsoleIO( );
275 bool            isConsoleConnected( );
276 char            getConsoleChar( uint32_t timeoutVal = 0 );
277
278 //-----------------------------------------------------------------------------------------------------------
279 // CDC setup and configuration routines. The idea is to help the library write with a default configuration
280 // structure. All pins HW that are fixed in their location will be set. A library programmer will just get
281 // that default structure and set the values necessary for the particular case.
282 //
283 //-----------------------------------------------------------------------------------------------------------
284 CdcConfigDesc   getConfigDefault( );
285 CdcConfigDesc   *getConfigActual( );
286 void            printConfigInfo( CdcConfigDesc *ci );
287 void            setDebugLevel( uint8_t level = 0 );
288
289 uint8_t         init( CdcConfigDesc *ci );
290 void            fatalError( uint8_t n );
291 void            fatalErrorMsg( char *str, uint8_t n, uint8_t rStat );
292
```

```
293 //-------------------------------------------------------------------------------------------------------------
294 // General controller routines.
295 //
296 //-------------------------------------------------------------------------------------------------------------
297 uint16_t      getFamily( );
298 uint32_t      getVersion( );
299 uint32_t      getChipMemSize( );
300 uint32_t      getChipNvmSize( );
301 uint32_t      getCpuFrequency( );
302 uint32_t      getMillis( );
303 uint32_t      getMicros( );
304 void          sleepMillis( uint32_t val );
305 void          sleepMicros( uint32_t val );
306
307 //-------------------------------------------------------------------------------------------------------------
308 // The LCS runtime needs to build a unique ID for the node.
309 //
310 //-------------------------------------------------------------------------------------------------------------
311 uint32_t      createUid( );
312
313 //-------------------------------------------------------------------------------------------------------------
314 // Timer management routines.
315 //
316 //-------------------------------------------------------------------------------------------------------------
317 void          onTimerEvent( TimerCallback functionId );
318 void          startRepeatingTimer( uint32_t val );
319 void          setRepeatingTimerLimit( uint32_t val );
320 uint32_t      getRepeatingTimerLimit( );
321 void          stopRepeatingTimer( );
322
323 //-------------------------------------------------------------------------------------------------------------
324 // Analog input routines.
325 //
326 //-------------------------------------------------------------------------------------------------------------
327 uint8_t       configureAdc( uint8_t adcPin );
328 uint16_t      getAdcRefVoltage( );
329 uint16_t      getAdcDigitRange( );
330 uint16_t      readAdc( uint8_t adcPin );
331
332 //-------------------------------------------------------------------------------------------------------------
333 // Digital Input/Output routines.
334 //
335 //-------------------------------------------------------------------------------------------------------------
336 uint8_t       configureDio( uint8_t dioPin, uint8_t Mode = IN );
337 void          registerDioCallback( uint8_t dioPin, uint8_t event, CDC::GpioCallback func );
338 void          unregisterDioCallback( uint8_t dioPin );
339 bool          readDio( uint8_t dioPin );
340 uint8_t       writeDio( uint8_t dioPin, bool val );
341 uint8_t       toggleDio( uint8_t dioPin );
342 uint32_t      readDioMask( uint32_t dioMask );
343 uint8_t       writeDioMask( uint32_t dioMask, uint32_t dioVal );
344 uint8_t       writeDioPair( uint8_t dioPin1, bool val1, uint8_t dioPin2, bool val2 );
345
346 //-------------------------------------------------------------------------------------------------------------
347 // PWM output routines.
348 //
349 //-------------------------------------------------------------------------------------------------------------
350 uint8_t       configurePwm( uint8_t   pwmPin,
351                             uint32_t  pwmFreqency,
352                             bool      phaseCorrect  = true,
353                             bool      inverted      = false
354                           );
355
356 uint8_t       writePwm( uint8_t pwmPin, uint8_t dutyCycle );
357
358 //-------------------------------------------------------------------------------------------------------------
359 // Serial IO routines.
360 //
361 //-------------------------------------------------------------------------------------------------------------
362 uint8_t       configureUart( uint8_t rxPin, uint8_t txPin, uint32_t baudRate, UartMode mode );
363 uint8_t       startUartRead( uint8_t rxPin );
364 uint8_t       stopUartRead( uint8_t rxPin );
365 uint8_t       getUartBuffer( uint8_t rxPin, uint8_t *buf, uint8_t bufLen );
366
367 //-------------------------------------------------------------------------------------------------------------
368 // I2C management routines.
369 //
370 //-------------------------------------------------------------------------------------------------------------
371 uint8_t       configureI2C( uint8_t sclPin, uint8_t sdaPin, uint32_t baudRate = 100 * 1000 );
372 uint8_t       i2cWrite( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit = false );
373 uint8_t       i2cRead( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit = false );
374
375 //-------------------------------------------------------------------------------------------------------------
376 // SPI management routines.
377 //
378 //-------------------------------------------------------------------------------------------------------------
379 uint8_t       configureSPI( uint8_t sclkPin, uint8_t mosiPin, uint8_t misoPin, uint32_t baudRate = 10 * 1000 * 1000 );
380 uint8_t       spiBeginTransaction( uint8_t sclkPin, uint8_t csPin );
381 uint8_t       spiEndTransaction( uint8_t sclkPin, uint8_t csPin );
382 uint8_t       spiRead( uint8_t sclkPin, uint8_t *buf, uint32_t len );
383 uint8_t       spiWrite( uint8_t sclkPin, uint8_t *buf, uint32_t len );
384
385 };
386
387 #endif
```

```
1  //-----------------------------------------------------------------------------------------------------------
2  //
3  // LCS - Controller dependent code Layer - Raspberry PI Pico Implementation
4  //
5  //-----------------------------------------------------------------------------------------------------------
6  // This source file contains the the RP2040 controller family hardware library code. The idea of this library
7  // is to shield the actual hardware of processor and board implementation from the upper layers but still keep
8  // the flexibility and performance of the underlying hardware. The library works with the concept of HW pins,
9  // which are identifiers for an HW entity. This is easy for a GPIO pin, where the mapping is directly one to
10 // one. For more complex HW entries such as the I2C or UART hardware, one pin is selected as the identifier to
11 // that entity. For each complex entity an instance variable is maintained where all the relevant data is kept.
12 //
13 // A historic note. The original LCS code was written for Atmega and Pico. With the complete shift to PICO,
14 // the CDC library just serves as a simple interface to the PICO functions. One day, we may see more different
15 // controllers and controller families. The idea is that the LCS runtime is shielded from them.
16 //
17 //-----------------------------------------------------------------------------------------------------------
18 //
19 // LCS - Controller Dependent Code - Raspberry PI Pico Implementation
20 // Copyright (C) 2022 - 2024 Helmut Fieres
21 //
22 // This program is free software: you can redistribute it and/or modify it under the terms of the GNU General
23 // Public License as published by the Free Software Foundation, either version 3 of the License, or (at your
24 // option) any later version.
25 //
26 // This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
27 // implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
28 // for more details.
29 //
30 // You should have received a copy of the GNU General Public License along with this program. If not, see
31 // http://www.gnu.org/licenses
32 //
33 //  GNU General Public License:  http://opensource.org/licenses/GPL-3.0
34 //
35 //-----------------------------------------------------------------------------------------------------------
36 #include <stdio.h>
37 #include <stdint.h>
38 #include <inttypes.h>
39
40 #include "pico/stdlib.h"
41 #include "pico/stdio.h"
42 #include "tusb_config.h"
43 #include "hardware/regs/usb.h"
44 #include "hardware/regs/rosc.h"
45 #include "hardware/regs/addressmap.h"
46 #include "hardware/clocks.h"
47 #include "hardware/gpio.h"
48 #include "hardware/adc.h"
49 #include "hardware/pwm.h"
50 #include "hardware/uart.h"
51 #include "hardware/i2c.h"
52 #include "hardware/spi.h"
53
54 #include "LcsCdcLib.h"
55
56 //-----------------------------------------------------------------------------------------------------------
57 // Local name space. This file has two sections. The first is this local name space with all internal
58 // variables and routines local to the file. The second part contains the exported routines to be called by
59 // the core library and the firmware designers that need access to the underlying HW portion managed by this
60 // lowest layer.
61 //
62 //-----------------------------------------------------------------------------------------------------------
63 namespace {
64
65 using namespace CDC;
66
67 //-----------------------------------------------------------------------------------------------------------
68 // "CDC_DEBUG" is the local define for printing debug information. In contrast to the rest of the debugging
69 // and tracing of LCS libraries and programs, this library will have to be recompiled to enable debugging.
70 //
71 //-----------------------------------------------------------------------------------------------------------
72 #define CDC_DEBUG 0
73
74 //-----------------------------------------------------------------------------------------------------------
75 // Debug and Trace support. Instead of conditional compilation, we will print debug messages based on the
76 // setting of the debug level.
77 //-----------------------------------------------------------------------------------------------------------
78 uint8_t debugLevel = 0;
79
80 //-----------------------------------------------------------------------------------------------------------
81 // The CDC Library version data.
82 //
83 //-----------------------------------------------------------------------------------------------------------
84 const uint8_t CDC_LIB_MAJOR_VERSION = 1;
85 const uint8_t CDC_LIB_MINOR_VERSION = 0;
86
87 //-----------------------------------------------------------------------------------------------------------
88 // Valid pin mapping for the Raspberry PI Pico board. We construct a set of bitmask for the pin numbers.
89 // Pin Numbers range from 0 to 28. The bitmasks specify wether a pin can be assigned to the hardware type
90 // purpose. During configuration of a CDC function, the pins are checked against these bitmasks. All pins
91 // can be used as GPIO pins or PWM pins. All other hardware functions are bound to dedicated pins. Note
92 // that we do not check for assigning a pin to several different hardware functions. All we check is that
93 // the pin can be used for the desired purpose. A check performed by the CDC library routines is simply
94 // done through:
95 //
96 //    if (( 1 <<  pin ) & VALID_xxx )
97 //
98 //-----------------------------------------------------------------------------------------------------------
```

```
99  const uint8_t  MAX_PIN_NUM          = 28;
100
101 const uint32_t VALID_GPIO_PINS      = 0x1FFFFFFF;
102 const uint32_t VALID_PWM_PINS       = 0x1FFFFFFF;
103 const uint32_t VALID_ADC_PINS       = ( 1 << 26 ) | ( 1 << 27 ) | ( 1 << 28 );
104
105 const uint32_t VALID_I2C_0_SDA_PINS = ( 1 << 0  ) | ( 1 << 4  ) | ( 1 << 8  ) |
106                                       ( 1 << 12 ) | ( 1 << 16 ) | ( 1 << 20 );
107 const uint32_t VALID_I2C_0_SCL_PINS = ( 1 << 1  ) | ( 1 << 5  ) | ( 1 << 9  ) |
108                                       ( 1 << 13 ) | ( 1 << 17 ) | ( 1 << 21 );
109
110 const uint32_t VALID_I2C_1_SDA_PINS = ( 1 << 2  ) | ( 1 << 6  ) | ( 1 << 10 ) |
111                                       ( 1 << 14 ) | ( 1 << 18 ) | ( 1 << 26 );
112 const uint32_t VALID_I2C_1_SCL_PINS = ( 1 << 3  ) | ( 1 << 7  ) | ( 1 << 11 ) |
113                                       ( 1 << 15 ) | ( 1 << 19 ) | ( 1 << 27 );
114
115 const uint32_t VALID_UART_0_TX_PINS = ( 1 << 0  ) | ( 1 << 12 ) | ( 1 << 16 );
116 const uint32_t VALID_UART_0_RX_PINS = ( 1 << 1  ) | ( 1 << 13 ) | ( 1 << 17 );
117
118 const uint32_t VALID_UART_1_TX_PINS = ( 1 << 4  ) | ( 1 << 8  );
119 const uint32_t VALID_UART_1_RX_PINS = ( 1 << 5  ) | ( 1 << 9  );
120
121 const uint32_t VALID_SPI_0_SCK_PINS = ( 1 << 2  ) | ( 1 << 6  ) | ( 1 << 18 );
122 const uint32_t VALID_SPI_0_TX_PINS  = ( 1 << 3  ) | ( 1 << 7  ) | ( 1 << 19 );
123 const uint32_t VALID_SPI_0_RX_PINS  = ( 1 << 0  ) | ( 1 << 4  ) | ( 1 << 16 );
124
125 const uint32_t VALID_SPI_1_SCK_PINS = ( 1 << 10 ) | ( 1 << 14 );
126 const uint32_t VALID_SPI_1_TX_PINS  = ( 1 << 11 ) | ( 1 << 15 );
127 const uint32_t VALID_SPI_1_RX_PINS  = ( 1 << 8  ) | ( 1 << 12 );
128
129 const uint32_t VALID_I2C_0_PINS     = VALID_I2C_0_SDA_PINS | VALID_I2C_0_SCL_PINS;
130 const uint32_t VALID_I2C_1_PINS     = VALID_I2C_1_SDA_PINS | VALID_I2C_1_SCL_PINS;
131
132 const uint32_t VALID_UART_0_PINS    = VALID_UART_0_TX_PINS | VALID_UART_0_RX_PINS;
133 const uint32_t VALID_UART_1_PINS    = VALID_UART_1_TX_PINS | VALID_UART_1_RX_PINS;
134
135 const uint32_t VALID_SPI_0_PINS     = VALID_SPI_0_SCK_PINS | VALID_SPI_0_TX_PINS | VALID_SPI_0_RX_PINS;
136 const uint32_t VALID_SPI_1_PINS     = VALID_SPI_1_SCK_PINS | VALID_SPI_1_TX_PINS | VALID_SPI_1_RX_PINS;
137
138 //-----------------------------------------------------------------------------------------------------
139 // Characteristics of the Raspberry Pi Pico and some key constants for the CDC library.
140 //
141 //-----------------------------------------------------------------------------------------------------
142 const uint16_t CONTROLLER_FAMILY          = CDC::CF_RP_PICO;
143 const uint32_t CHIP_MEM_SIZE              = 264 * 1024;
144 const uint32_t CHIP_NVM_SIZE              = 0;
145
146 const uint16_t ADC_DIGIT_RANGE            = 1024;
147 const uint16_t ADC_REF_VOLTAGE_MILLI_VOLT = 3300;
148
149 const uint8_t  MAX_UART_BUF_SIZE          = 8;
150
151 const uint32_t I2C_FREQUENCY              = 100 * 1000;
152 const uint32_t I2C_TIME_OUT_IN_MS         = 250;
153
154 const uint32_t SPI_FREQUENCY              = 10000000L;
155
156 const uint16_t  MAX_CPU_CORE              = 2;
157 const uint16_t  MAX_INT_PIN               = 24;
158
159 //-----------------------------------------------------------------------------------------------------
160 // A timer instance. We currently support inly one HW timer.
161 //
162 //-----------------------------------------------------------------------------------------------------
163 struct TimerInst {
164
165     bool                configured  = false;
166     repeating_timer_t   timerData;
167
168 };
169
170 //-----------------------------------------------------------------------------------------------------
171 // An ADC instance. The PICO supports up to three ADC inputs. When we use such an input, the corresponding
172 // instance data is kept in this structure. We also keep the PICO ADC number, so we can select the correct
173 // instance.
174 //
175 //-----------------------------------------------------------------------------------------------------
176 struct AdcInst {
177
178     bool      configured  = false;
179     uint8_t   adcPin      = CDC::UNDEFINED_PIN;
180     uint8_t   adcNum      = 0;
181 };
182
183 //-----------------------------------------------------------------------------------------------------
184 // A PWM output instance. GPIO pins can also be used as PWM output pins. The PWM output related data is
185 // kept in this instance.
186 //
187 //-----------------------------------------------------------------------------------------------------
188 struct PwmInst {
189
190     bool      configured  = false;
191     uint8_t   pwmPin      = CDC::UNDEFINED_PIN;
192     uint32_t  wrap        = 0;
193
194     // ??? what else to keep around ?
195 };
196
197 //-----------------------------------------------------------------------------------------------------
```

```
198 // A UART instance. UARTS are used to read in a serial stream from the RailCom detectors. There can be two
199 // hardware based UART instances, or up to four software defined instances. The instance also keeps a small
200 // buffer where the data is read into. We also keep the PICO UART HW instance used.
201 //
202 //--------------------------------------------------------------------------------------------------------------
203 struct UartInst {
204
205     bool               configured    = false;
206     uint8_t            rxPin         = CDC::UNDEFINED_PIN;
207     uint8_t            txPin         = CDC::UNDEFINED_PIN;
208     uint16_t           baudSetting   = 0;
209     uint8_t            dataBits      = 8;
210     uart_parity_t      parityMode    = UART_PARITY_NONE;
211     uint8_t            stopBits      = 1;
212     int                uartIrq       = 0;
213     uint8_t            uartMode      = 0;
214
215     volatile uint8_t   rxBufIndex    = 0;
216     volatile uint8_t   rxDataBuf[ MAX_UART_BUF_SIZE ] = { 0 };
217
218     uart_inst_t        *uartHw       = nullptr;
219 };
220
221 //--------------------------------------------------------------------------------------------------------------
222 // The I2C instance. The PICO features two HW instances of an I2C port. The instance data contains the
223 // assigned GPIO pins, the baud rate and a timeout. We also keep the I2C HW instance used.
224 //
225 //--------------------------------------------------------------------------------------------------------------
226 struct I2CInst {
227
228     bool         configured    = false;
229     uint8_t      sclPin        = CDC::UNDEFINED_PIN;
230     uint8_t      sdaPin        = CDC::UNDEFINED_PIN;
231     uint32_t     baudRate      = I2C_FREQUENCY;
232     uint32_t     timeoutValMs  = I2C_TIME_OUT_IN_MS;
233
234     i2c_inst_t   *i2cHw        = nullptr;
235 };
236
237 //--------------------------------------------------------------------------------------------------------------
238 // The SPI instance. The PICO features two SPI HW instances. We keep the assigned GPIO pins for the SPI
239 // interface as well as the PICO HW instance. Since the SPI protocol explicitly sets the selected HW select
240 // pin, we remember that we are in a transaction with perhaps more than one call to the SPI routines.
241 //
242 //--------------------------------------------------------------------------------------------------------------
243 struct SPIInst {
244
245     bool         configured  = false;
246     bool         active      = false;
247     uint8_t      selectPin   = CDC::UNDEFINED_PIN;
248     uint8_t      mosiPin     = CDC::UNDEFINED_PIN;
249     uint8_t      misoPin     = CDC::UNDEFINED_PIN;
250     uint8_t      sclkPin     = CDC::UNDEFINED_PIN;
251     uint32_t     frequency   = SPI_FREQUENCY;
252
253     spi_inst_t   *spiHw      = nullptr;
254 };
255
256 //--------------------------------------------------------------------------------------------------------------
257 // The interrupt table for the GPIO pin interrupts. The PICO can have only one interrupt handler. We will
258 // allocate a table where a handler can be set for each pin. When an interrupt comes in and there is a
259 // handler configured, it will be called.
260 //
261 //--------------------------------------------------------------------------------------------------------------
262 struct GpioIsrTable {
263
264     uint16_t                 numOfHandlers = 0;
265     CDC::GpioCallback        gpioIsrTable[ MAX_CPU_CORE ][ MAX_INT_PIN + 1 ];
266 };
267
268 //--------------------------------------------------------------------------------------------------------------
269 // Local variables. We maintain an instance variable for each of the possible HW entities, such as an I2C
270 // interface or a UART. Note that not all are used at the same time. The instance variables map from the
271 // simple pin numbers to the PICO structures and whatever else we need to remember for this entity.
272 //
273 //--------------------------------------------------------------------------------------------------------------
274 CDC::CdcConfigDesc          cfg;
275 CDC::TimerCallback          timerCallback = nullptr;
276 GpioIsrTable                cdcIntHandlers;
277 repeating_timer_t           timerData;
278 AdcInst                     CdcAdc0;
279 AdcInst                     CdcAdc1;
280 AdcInst                     CdcAdc2;
281 AdcInst                     CdcAdc3;
282 I2CInst                     CdcI2C0;
283 I2CInst                     CdcI2C1;
284 SPIInst                     CdcSPI0;
285 SPIInst                     CdcSPI1;
286 UartInst                    CdcUart0;
287 UartInst                    CdcUart1;
288 UartInst                    CdcUart2;
289 UartInst                    CdcUart3;
290 PwmInst                     CdcPwm0;
291 PwmInst                     CdcPwm1;
292 PwmInst                     CdcPwm2;
293 PwmInst                     CdcPwm3;
294
295 //--------------------------------------------------------------------------------------------------------------
296 // "validPin" is called to check that a pin is in the correct number range, defined and matches the bitmask
```

```
297 // for the desired purpose. For example, configuring an I2C port will check that the two GPIO pins are
298 // indeed routable to the I2C HW block in the PICO.
299 //
300 //-----------------------------------------------------------------------------------------------------------
301 bool validPin( uint8_t pin, uint32_t mask ) {
302
303     if ( pin == CDC::UNDEFINED_PIN )  return ( true );
304     if ( pin > MAX_PIN_NUM )          return ( false );
305     return (( 1 << pin ) & mask );
306 }
307
308 //-----------------------------------------------------------------------------------------------------------
309 // When no interrupt is configured for a GPIO pin, we set the table entry to a dummy handler. This way
310 // we do not have to check for a valid procedure label when we handle an interrupt.
311 //
312 //-----------------------------------------------------------------------------------------------------------
313 void dummyIsrHandler ( uint8_t pin, uint8_t event ) { }
314
315 //-----------------------------------------------------------------------------------------------------------
316 // Setup the ISR table. The PICO can have only one interrupt handler. When you want a handler per GPIO pin,
317 // the solution is to have a table when you keep the handler on a per pin base.
318 //
319 //-----------------------------------------------------------------------------------------------------------
320 void initIsrTable( ) {
321
322     for ( uint16_t i = 0; i < MAX_CPU_CORE; i++ ) {
323
324         for ( uint16_t j = 0; j < MAX_INT_PIN; j++ ) {
325
326             cdcIntHandlers.gpioIsrTable[ i ][ j ] = dummyIsrHandler;
327         }
328     }
329 }
330
331 //-----------------------------------------------------------------------------------------------------------
332 // The PICO uses a set of constants to describe the interrupt type. We map our interrupt types to the PICO
333 // GPIO_IRQ_xxx types.
334 //
335 //-----------------------------------------------------------------------------------------------------------
336 uint32_t mapGpioIntEvent( uint8_t event ) {
337
338     switch ( event ) {
339
340         case CDC::EVT_LOW:     return( GPIO_IRQ_LEVEL_LOW );
341         case CDC::EVT_HIGH:    return( GPIO_IRQ_LEVEL_HIGH );
342         case CDC::EVT_FALL:    return( GPIO_IRQ_EDGE_FALL );
343         case CDC::EVT_RISE:    return( GPIO_IRQ_EDGE_RISE );
344         case CDC::EVT_CHANGE:  return( GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL );
345         default:       return( 0 );
346     }
347 }
348
349 //-----------------------------------------------------------------------------------------------------------
350 // The PICO uses a set of constants to describe the interrupt type. We map them to our types.
351 //
352 //-----------------------------------------------------------------------------------------------------------
353 uint8_t mapPicoGpioEvent( uint32_t event ) {
354
355     switch ( event ) {
356
357         case GPIO_IRQ_LEVEL_LOW:  return( CDC::EVT_LOW );
358         case GPIO_IRQ_LEVEL_HIGH: return( CDC::EVT_HIGH );
359         case GPIO_IRQ_EDGE_FALL:  return( CDC::EVT_FALL );
360         case GPIO_IRQ_EDGE_RISE:  return( CDC::EVT_RISE );
361         default:                  return( 0 );
362     }
363 }
364
365 //-----------------------------------------------------------------------------------------------------------
366 // Global Interrupt handlers. The hardware and low level library will call these handlers, which in turn
367 // will invoke the respective callback function if configured. The GPIO interrupt handler manages the
368 // handler for all possible IO pins. The PICO can only have one interrupt routine, so we feature an array
369 // of handlers where a handler for a GPIO pin can be registered. If there is a handler set, we just invoke
370 // it. The other handlers are for the timer and the UART hardware.
371 //
372 //-----------------------------------------------------------------------------------------------------------
373 void gpioCallback( uint gpioPin, uint32_t event ) {
374
375     cdcIntHandlers.gpioIsrTable[ get_core_num( )][ gpioPin] ( gpioPin, mapPicoGpioEvent( event ));
376 }
377
378 bool repeatingTimerAlarm( repeating_timer_t *rt ) {
379
380     if ( timerCallback != nullptr ) timerCallback((uint32_t) ( - timerData.delay_us ));
381     return ( true );
382 }
383
384 void uartRxCallback0( ) {
385
386     while ( uart_is_readable( uart0 )) {
387
388         uint8_t ch = uart_getc( uart0 );
389         if ( CdcUart0.rxBufIndex < MAX_UART_BUF_SIZE ) CdcUart0.rxDataBuf[CdcUart0.rxBufIndex++ ] = ch;
390     }
391 }
392
393 void uartRxCallback1( ) {
394
395     while ( uart_is_readable( uart1 )) {
```

```
396
397         uint8_t ch = uart_getc( uart1 );
398         if ( CdcUart1.rxBufIndex < MAX_UART_BUF_SIZE ) CdcUart1.rxDataBuf[ CdcUart1.rxBufIndex++ ] = ch;
399     }
400 }
401
402 //-----------------------------------------------------------------------------------------------------------
403 // The default configuration descriptor. The Application program fills in such a structure, which can be
404 // seen as the HW pin assignments for the PICO controllers and the particular board on which the application
405 // will be deployed. The application will simply use the field names to address the particular PICO HW
406 // function. For example, a configuration has mapped DIO_PIN_5 to GPIO pin 12, because that is where the
407 // particular board has mapped DIO_PIN_5 to the hardware line. The application will just use the DIO_PIN_5
408 // field when talking to that GPIO pin. Whenever the board layout changes, there could be another PICO GPIO
409 // pin, but the name "DIO_PIN_5" for the application upper layers does not change.
410 //
411 // Note that there is a great flexibility what a PICO HW  pin can do and hence a lot of our fields are just
412 // "UNDEFINED" with no constraints. Nevertheless, there is a function which will do some plausibility checks
413 // for such a structure. Also, each configuration routine will do again a check that the GPIO pins used do
414 // indeed map to a PICO HW block for the desired purpose.
415 //
416 // The configuration structure does not replace the actual configuration calls to make to the CDC library.
417 // It is just a mapping of reserved names to actual GPIO pins.
418 //
419 //-----------------------------------------------------------------------------------------------------------
420 CDC::CdcConfigDesc getConfigDefaultRP2040( ) {
421
422     CDC::CdcConfigDesc tmp;
423
424     tmp.CFG_STATUS          = CDC::INIT_PENDING;
425
426     // ??? controller family ?
427     // ??? what other characteristics ? ( e.g. mem size ? )
428
429     tmp.READY_LED_PIN       = CDC::UNDEFINED_PIN;
430     tmp.ACTIVE_LED_PIN      = CDC::UNDEFINED_PIN;
431
432     tmp.EXT_INT_PIN         = CDC::UNDEFINED_PIN;
433     tmp.PFAIL_PIN           = CDC::UNDEFINED_PIN;
434
435     tmp.DIO_PIN_0           = CDC::UNDEFINED_PIN;
436     tmp.DIO_PIN_1           = CDC::UNDEFINED_PIN;
437     tmp.DIO_PIN_2           = CDC::UNDEFINED_PIN;
438     tmp.DIO_PIN_3           = CDC::UNDEFINED_PIN;
439     tmp.DIO_PIN_4           = CDC::UNDEFINED_PIN;
440     tmp.DIO_PIN_5           = CDC::UNDEFINED_PIN;
441     tmp.DIO_PIN_6           = CDC::UNDEFINED_PIN;
442     tmp.DIO_PIN_7           = CDC::UNDEFINED_PIN;
443     tmp.DIO_PIN_8           = CDC::UNDEFINED_PIN;
444     tmp.DIO_PIN_9           = CDC::UNDEFINED_PIN;
445     tmp.DIO_PIN_10          = CDC::UNDEFINED_PIN;
446     tmp.DIO_PIN_11          = CDC::UNDEFINED_PIN;
447     tmp.DIO_PIN_12          = CDC::UNDEFINED_PIN;
448     tmp.DIO_PIN_13          = CDC::UNDEFINED_PIN;
449     tmp.DIO_PIN_14          = CDC::UNDEFINED_PIN;
450     tmp.DIO_PIN_15          = CDC::UNDEFINED_PIN;
451
452     tmp.ADC_PIN_0           = CDC::UNDEFINED_PIN;
453     tmp.ADC_PIN_1           = CDC::UNDEFINED_PIN;
454     tmp.ADC_PIN_2           = CDC::UNDEFINED_PIN;
455     tmp.ADC_PIN_3           = CDC::ILLEGAL_PIN;
456
457     tmp.PWM_PIN_0           = CDC::UNDEFINED_PIN;
458     tmp.PWM_PIN_1           = CDC::UNDEFINED_PIN;
459     tmp.PWM_PIN_2           = CDC::UNDEFINED_PIN;
460     tmp.PWM_PIN_3           = CDC::UNDEFINED_PIN;
461
462     tmp.UART_RX_PIN_0       = CDC::UNDEFINED_PIN;
463     tmp.UART_TX_PIN_0       = CDC::UNDEFINED_PIN;
464
465     tmp.UART_RX_PIN_1       = CDC::UNDEFINED_PIN;
466     tmp.UART_TX_PIN_1       = CDC::UNDEFINED_PIN;
467
468     tmp.UART_RX_PIN_2       = CDC::UNDEFINED_PIN;
469     tmp.UART_TX_PIN_2       = CDC::UNDEFINED_PIN;
470
471     tmp.UART_RX_PIN_3       = CDC::UNDEFINED_PIN;
472     tmp.UART_TX_PIN_3       = CDC::UNDEFINED_PIN;
473
474     tmp.SPI_MOSI_PIN_0      = CDC::UNDEFINED_PIN;
475     tmp.SPI_MISO_PIN_0      = CDC::UNDEFINED_PIN;
476     tmp.SPI_SCLK_PIN_0      = CDC::UNDEFINED_PIN;
477
478     tmp.SPI_MOSI_PIN_1      = CDC::UNDEFINED_PIN;
479     tmp.SPI_MISO_PIN_1      = CDC::UNDEFINED_PIN;
480     tmp.SPI_SCLK_PIN_1      = CDC::UNDEFINED_PIN;
481
482     tmp.NVM_I2C_SCL_PIN     = CDC::UNDEFINED_PIN;
483     tmp.NVM_I2C_SDA_PIN     = CDC::UNDEFINED_PIN;
484
485     tmp.EXT_I2C_SCL_PIN     = 17;
486     tmp.EXT_I2C_SDA_PIN     = 16;
487
488     tmp.CAN_BUS_RX_PIN      = CDC::UNDEFINED_PIN;
489     tmp.CAN_BUS_TX_PIN      = CDC::UNDEFINED_PIN;
490
491     return ( tmp );
492 }
493
494 //-----------------------------------------------------------------------------------------------------------
```

```
495 // Validate a configuration structure. This routine will do basic checking of the pin configuration passed.
496 // The PICO is very flexible when it comes to what a pin can do. However, there are still some rules to
497 // follow. Also, we have dedicated settings for at least the I2C channels and the CAN bus IO pins.
498 //
499 //-------------------------------------------------------------------------------------------------------------
500 uint8_t validateConfigRP20040( CDC::CdcConfigDesc *ci ) {
501
502     // ??? a ton of "validXXX" ?
503
504     return ( NO_ERR ); // for now....
505 }
506
507 }; // namespace
508
509
510 //-------------------------------------------------------------------------------------------------------------
511 // Bane CDC. All routines and definitions exported are in this name space.
512 //
513 //-------------------------------------------------------------------------------------------------------------
514 namespace CDC {
515
516 //-------------------------------------------------------------------------------------------------------------
517 // For debugging purposes. Instead of conditional compilations, the debug level will enable the printing of
518 // debug and trace data.
519 //
520 //-------------------------------------------------------------------------------------------------------------
521 void setDebugLevel( uint8_t level ) {
522
523     debugLevel = level;
524 }
525
526 //-------------------------------------------------------------------------------------------------------------
527 // "getConfigDefault" initializes a configuration structure and sets the pre-assigned values. A typical
528 // sequence for an application start sequence would be to create an initial structure this way and then set
529 // the relevant pins and values according to the actual hardware configuration.
530 //
531 //-------------------------------------------------------------------------------------------------------------
532 CdcConfigDesc getConfigDefault( ) {
533
534    return ( getConfigDefaultRP2040( ));
535 }
536
537 //-------------------------------------------------------------------------------------------------------------
538 // "getConfigActual" will return a pointer to the copy we kept when calling the init routine with the config
539 // structure to use. There is no need for the upper layers to keep the structure used at initialization time.
540 //
541 //-------------------------------------------------------------------------------------------------------------
542 CdcConfigDesc *getConfigActual( ) {
543
544    return ( &cfg );
545 }
546
547 //-------------------------------------------------------------------------------------------------------------
548 // CDC library setup. The "init" routine will ready the CDC library. The main task is to validate the pins and
549 // values for the particular controller capabilities. The init routine can be called more than once without a
550 // problem.
551 //
552 //-------------------------------------------------------------------------------------------------------------
553 uint8_t init( CdcConfigDesc *ci ) {
554
555     cfg = *ci;
556
557     initIsrTable( );
558     configureConsoleIO( );
559
560     return ( validateConfigRP20040( ci ));
561 }
562
563 //-------------------------------------------------------------------------------------------------------------
564 // "fatalError" is the error communication method when we cannot get anything to work, except the onboard
565 // LED. The Raspberry Pi PICO has a small Led on the board. We will use this LED to "blink" an error code.
566 // There are up to eight codes. The sequence is as follows:
567 //
568 //    repeat forever:
569 //
570 //    - 1s ON, 0.5s OFF
571 //    - for ( int i = 0; i < n; i++ ) { 0.5s ON; 0.5s OFF; }
572 //
573 // The only way to get out of this loop is then to reset the board. Fatal errors are hopefully not many. One
574 // obvious one is when we cannot detect the NVM and thus know nothing about the board.
575 //
576 //-------------------------------------------------------------------------------------------------------------
577 void fatalError( uint8_t n ) {
578
579     const uint8_t   ledPin      = 25;
580     const uint32_t  longPulse   = 1000;
581     const uint32_t  shortPulse  = 250;
582
583     n = n % 8;
584
585     gpio_init( ledPin );
586     gpio_set_dir( ledPin, GPIO_OUT );
587
588     while ( true ) {
589
590         sleep_ms( longPulse );
591
592         for ( int i = 0; i < n; i++ ) {
593
```

```
            gpio_put( ledPin, true );
            sleep_ms( shortPulse );
            gpio_put( ledPin, false );
            sleep_ms( shortPulse );
        }
    }
}

//-------------------------------------------------------------------------------------------------------------
// "fatalErrorMsg" will result in a fatal error, but we attempt to first write an error message to the
// console.
//
//-------------------------------------------------------------------------------------------------------------
void fatalErrorMsg( char *str, uint8_t n, uint8_t rStat ) {

    if ( isConsoleConnected( )) printf( "Fatal Error: %d: %s, rStat: %d\n", n, str, rStat );
    fatalError( n );
}

//-------------------------------------------------------------------------------------------------------------
// Processor general values required by the low level LCS core library functions.
//
//-------------------------------------------------------------------------------------------------------------
uint16_t getFamily( ) {

    return ( CONTROLLER_FAMILY );
}

uint32_t getVersion( ) {

    return ( CDC_LIB_MAJOR_VERSION << 8 | CDC_LIB_MINOR_VERSION );
}

uint32_t getChipMemSize( ) {

    return ( CHIP_MEM_SIZE );
}

uint32_t getChipNvmSize( ) {

    return ( CHIP_NVM_SIZE   );
}

uint32_t getCpuFrequency( ) {

    return ( clock_get_hz( clk_sys ));
}

uint32_t getMillis( ) {

    return ( to_ms_since_boot( get_absolute_time( )));
}

uint32_t getMicros( ) {

    return ( to_us_since_boot( get_absolute_time( )));
}

void sleepMillis( uint32_t val ) {

    sleep_ms( val );
}

void sleepMicros( uint32_t val ) {

    sleep_us( val );
}

//-------------------------------------------------------------------------------------------------------------
// "createUid" is the routine that produces a unique ID for the node. The scheme is still based on a random
// number. This is the PICO version for creating a random number. Alternatively we could use the unique
// flash chip ID on the board. TBD ...
//
//-------------------------------------------------------------------------------------------------------------
uint32_t createUid( ) {

    uint32_t rVal = 0;

    volatile uint32_t *rnd_reg = (uint32_t *) ( ROSC_BASE + ROSC_RANDOMBIT_OFFSET );

    for ( int k = 0; k < 32; k++ ) {

        rVal = rVal << 1;
        rVal = rVal + ( 0x00000001 & ( *rnd_reg ));
    }

    return ( rVal );
}

//-------------------------------------------------------------------------------------------------------------
// Console IO section. We set up the stdio via the USB connector. As part of the CDC init call, the configure
// call should be done rather early, so that we can print out debug messages. In normal LCS node operation
// there is no USB connected. Detecting a connection helps to decide whether we can report an error or need
// to resort to a fatal error call at startup.
//
// There are two basic ways to detect an USB connection. The first is to simply check if there is power on
// the USB port. The PICO features an internal GPIO pin for this purpose. Using this method still does not
// mean that we have someone connected to the USB, but just that there is a cable with power. Well, good
// enough for us. The second method truly detects that there is a USB host connected. This check is provided
```

```
693 // via the PICO libraries which in turn use the tinyUSB library. However, there could be a timing problem
694 // where the USB stack is not ready and we conclude wrongly that there is no USB connection. For now, let's
695 // rather go with the risk that there is just power on the USB connector.
696 //
697 // Finally, there is a routine to get a character for the command interfaces. Since the function just reads
698 // in a character, optionally with a timeout how long to wait for any inout.
699 //
700 // PS: The USB check way would be "return( stdio_usb_connected( ));" instead of the GPIO check.
701 //-----------------------------------------------------------------------------------------------------------
702 uint8_t configureConsoleIO( ) {
703
704     stdio_init_all( );
705     return( NO_ERR );
706 }
707
708 bool isConsoleConnected( ) {
709
710     gpio_init( PICO_VBUS_PIN );
711     gpio_set_dir( PICO_VBUS_PIN, GPIO_IN );
712
713     return( gpio_get( PICO_VBUS_PIN ));
714 }
715
716 char getConsoleChar( uint32_t timeoutVal ) {
717
718     int ch = getchar_timeout_us( timeoutVal );
719     return(( ch == PICO_ERROR_TIMEOUT ) ? 0 : ch );
720 }
721
722 //-----------------------------------------------------------------------------------------------------------
723 // Timer section. The CDC library features one generic repeating timer with a microsecond resolution. The
724 // routines start and stop the timer and allow to set a new limit. The PICO offers a high level function that
725 // schedules a repeating timer with the property of measuring the interval also from the start of the
726 // callback invocation. This is exactly what we need to implement the tick interrupt for the DCC signal state
727 // machine. The "setRepeatingTimerLimit" function will adjust the timer limit counter while the timer already
728 // is counting toward a limit. Note that the timer option that already start the next round while the timer
729 // interrupt handler executes is specified by using negative limit values. The timer functionality also
730 // offers two timestamp routines to get the number of milliseconds and number of microseconds since system
731 // start.
732 //
733 // ??? would we one day need more than one timer instance ?
734 //-----------------------------------------------------------------------------------------------------------
735 void startRepeatingTimer( uint32_t val ) {
736
737     int64_t limit = val;
738     add_repeating_timer_us( - limit, repeatingTimerAlarm, nullptr, &timerData );
739 }
740
741 void stopRepeatingTimer( ) {
742
743     cancel_repeating_timer( &timerData );
744 }
745
746 uint32_t getRepeatingTimerLimit( ) {
747
748     return ((uint32_t) ( - timerData.delay_us ));
749 }
750
751 void setRepeatingTimerLimit( uint32_t val ) {
752
753     int64_t limit = val;
754     timerData.delay_us = ((int64_t) - limit );
755 }
756
757 void onTimerEvent( CDC::TimerCallback functionId ) {
758
759     timerCallback = functionId;
760 }
761
762 //-----------------------------------------------------------------------------------------------------------
763 // DIO section. A digital pin is the bread and butter hardware resource and can be an input or output pin. For
764 // inputs, an internal pull-up resistor can be set.There are a couple of interfaces. First the single pin
765 // read, write and toggle. Next are read and write mask routines which work on all IO pins at once. Note that
766 // no cross checking is done if the pins are used by other CDC functions. Finally there is a convenience
767 // routine which write a pair of data. This is typically used for the H-Bridge control pins, which are set at
768 // the same time.
769 //
770 // A GPIO pin can also have an attached interrupt handler. When we register a handler for a pin, there are
771 // two different PICO lib routines to use. When there is no handler registered so far, we register the
772 // common callback and store the particular GPIO handler in the handler table. Otherwise, we just store the
773 // handler and enable the GPIO pin for interrupts.
774 //
775 //-----------------------------------------------------------------------------------------------------------
776 uint8_t configureDio( uint8_t dioPin, uint8_t mode ) {
777
778     if ( ! validPin( dioPin, VALID_GPIO_PINS )) return ( DIO_PIN_ERR );
779
780     gpio_init( dioPin );
781
782     switch ( mode ) {
783
784         case IN:  gpio_set_dir( dioPin, false );  break;
785         case OUT: {
786
787             gpio_set_dir( dioPin, true );
788             gpio_set_drive_strength ( dioPin, GPIO_DRIVE_STRENGTH_12MA );
789
790         }  break;
791
```

```
792         case IN_PULLUP: {
793
794             gpio_set_dir( dioPin, false );
795             gpio_pull_up( dioPin );
796
797         } break;
798
799         default: gpio_set_dir( dioPin, false );
800     }
801
802   return ( NO_ERR );
803 }
804
805 void registerDioCallback( uint8_t dioPin, uint8_t event, CDC::GpioCallback func ) {
806
807     if ( dioPin <= MAX_INT_PIN ) {
808
809         if ( cdcIntHandlers.numOfHandlers == 0 )
810             gpio_set_irq_enabled_with_callback( dioPin, mapGpioIntEvent( event ), true, gpioCallback );
811         else
812             gpio_set_irq_enabled( dioPin, mapGpioIntEvent( event ), true);
813
814         cdcIntHandlers.gpioIsrTable[ get_core_num( ) ][ dioPin ] = func;
815         cdcIntHandlers.numOfHandlers ++;
816     }
817 }
818
819 void unregisterDioCallback( uint8_t dioPin ) {
820
821     if ( dioPin <= MAX_INT_PIN ) {
822
823         if ( cdcIntHandlers.gpioIsrTable[ get_core_num( ) ][ dioPin ] != nullptr ) {
824
825             gpio_set_irq_enabled( dioPin, 0, false );
826             cdcIntHandlers.gpioIsrTable[ get_core_num( ) ][ dioPin ] = dummyIsrHandler;
827             cdcIntHandlers.numOfHandlers --;
828         }
829     }
830 }
831
832 bool readDio( uint8_t dioPin ) {
833
834     return ( gpio_get( dioPin ));
835 }
836
837 uint8_t writeDio( uint8_t dioPin, bool val ) {
838
839     gpio_put( dioPin, val );
840     return ( NO_ERR );
841 }
842
843 uint8_t toggleDio( uint8_t dioPin ) {
844
845     writeDio( dioPin, ! readDio( dioPin ));
846     return ( NO_ERR );
847 }
848
849 uint8_t writeDioPair( uint8_t dioPin1, bool val1, uint8_t dioPin2, bool val2 ) {
850
851     uint32_t maskData = ( 1UL << dioPin1 ) | ( 1UL << dioPin2 );
852     uint32_t valData  = (( val1 ) ? ( 1 << dioPin1 ) : 0 ) | (( val2 ) ? ( 1 << dioPin2 ) : 0 );
853
854     gpio_put_masked( maskData, valData );
855     return ( NO_ERR );
856 }
857
858 uint32_t readDioMask( uint32_t dioMask ) {
859
860     return ( gpio_get_all( ) & dioMask );
861 }
862
863 uint8_t writeDioMask( uint32_t dioMask, uint32_t dioVal ) {
864
865     gpio_put_masked( dioMask, dioVal );
866     return ( NO_ERR );
867 }
868
869 //-----------------------------------------------------------------------------------------------------------
870 // ADC section. The analog input channel represented by the pin is configured. At initialization, the ADC pin
871 // number is validated and the ADC subsystem initialized. The PICO does an analog read in about 2us. This is
872 // so fast, it does for our purpose make not much sense to implement an asynchronous option. Furthermore, the
873 // ADC value scaled down to a 10-bit resolution.
874 //
875 //-----------------------------------------------------------------------------------------------------------
876 uint8_t configureAdc( uint8_t adcPin ) {
877
878     if ( ! validPin( adcPin, VALID_ADC_PINS ))  return ( ADC_PIN_ERR );
879
880     AdcInst *tmp = nullptr;
881
882     if ( adcPin == cfg.ADC_PIN_0 ) {
883
884         tmp = &CdcAdc0;
885         tmp -> adcPin = adcPin;
886         tmp -> adcNum = 0;
887
888     }
889      else if ( adcPin == cfg.ADC_PIN_1 ) {
890
```

```
        tmp = &CdcAdc1;
        tmp -> adcPin = adcPin;
        tmp -> adcNum = 1;
    }
    else if ( adcPin == cfg.ADC_PIN_2 ) {

        tmp = &CdcAdc2;
        tmp -> adcPin = adcPin;
        tmp -> adcNum = 2;
    }
    else return ( ADC_PIN_ERR );

    adc_init( );
    adc_gpio_init( tmp -> adcPin );
    tmp -> configured = true;

    return ( NO_ERR );
}

uint16_t getAdcRefVoltage( ) {

    return ( ADC_REF_VOLTAGE_MILLI_VOLT );
}

uint16_t getAdcDigitRange( ) {

    return ( ADC_DIGIT_RANGE );
}

uint16_t readAdc( uint8_t adcPin ) {

    AdcInst *tmp = nullptr;

    if      ( adcPin == CdcAdc0.adcPin ) tmp = &CdcAdc0;
    else if ( adcPin == CdcAdc1.adcPin ) tmp = &CdcAdc1;
    else if ( adcPin == CdcAdc2.adcPin ) tmp = &CdcAdc2;
    else return ( 0 );
    adc_select_input( tmp -> adcNum );
    return ( adc_read( ) >> 2 );
}

//-------------------------------------------------------------------------------------------------------------
// UART section. The UART interface is primarily used for the RailCom Detector that sends a serial signal.
// So far, only the receiver portion is implemented because that is all what is needed for RailCom messages.
// There are two general categories. The first uses the PICO built-in UART hardware blocks. The second
// implements a software UART based on the PICO PIO blocks.
//
// There are three routines. The "startUartRead" will enable the UART and start reading bytes into the local
// buffer. The "stopUartRead" will then finish the byte collection and disable the UART again. Finally, the
// "getUartBuffer" routine will return the bytes received. Again, note that this is not a generic UART read
// interface.
//
// The work on the PIO based UART version has not started yet ... it will be needed for the quad block
// controller. Looking forward to it ...:-)
//
//-------------------------------------------------------------------------------------------------------------
uint8_t configureUart( uint8_t rxPin, uint8_t txPin, uint32_t baudRate, UartMode mode ) {

    UartInst *uart = nullptr;

    if ( mode == UART_MODE_8N1 ) {

        if (( validPin( rxPin, VALID_UART_0_RX_PINS )) && ( validPin( txPin, VALID_UART_0_TX_PINS ))) {

            uart                = &CdcUart0;
            uart -> uartMode    = mode;
            uart -> rxPin       = rxPin;
            uart -> txPin       = txPin;
            uart -> dataBits    = 8;
            uart -> stopBits    = 1;
            uart -> parityMode  = UART_PARITY_NONE;
            uart -> uartHw      = uart0;
            uart -> uartIrq     = UART0_IRQ;
        }
        else if (( validPin( rxPin, VALID_UART_1_RX_PINS )) && ( validPin( txPin, VALID_UART_1_TX_PINS ))) {

            uart                = &CdcUart1;
            uart -> uartMode    = mode;
            uart -> rxPin       = rxPin;
            uart -> txPin       = txPin;
            uart -> dataBits    = 8;
            uart -> stopBits    = 1;
            uart -> parityMode  = UART_PARITY_NONE;
            uart -> uartHw      = uart1;
            uart -> uartIrq     = UART1_IRQ;
        }
        else return ( UART_PORT_ERR );

        uart_init( uart -> uartHw, baudRate );
        gpio_set_function( rxPin, GPIO_FUNC_UART );
        gpio_set_function( txPin, GPIO_FUNC_UART );
        uart_set_hw_flow( uart -> uartHw, false, false );
        uart_set_format( uart -> uartHw, uart -> dataBits, uart -> stopBits, uart -> parityMode );
        uart_set_fifo_enabled( uart -> uartHw, false );

        if      ( uart -> uartIrq == UART0_IRQ ) irq_set_exclusive_handler( uart -> uartIrq, uartRxCallback0 );
        else if ( uart -> uartIrq == UART1_IRQ ) irq_set_exclusive_handler( uart -> uartIrq, uartRxCallback1 );

        irq_set_enabled( uart -> uartIrq, true );
```

```
990
991         return ( NO_ERR );
992     }
993     else if ( mode == UART_MODE_8N1_PIO ) {
994
995         return ( NOT_SUPPORTED );
996     }
997     else return ( NOT_SUPPORTED );
998 }
999
1000 uint8_t startUartRead( uint8_t rxPin ) {
1001
1002     UartInst *uart = nullptr;
1003
1004     if      ( rxPin == CdcUart0.rxPin ) uart = &CdcUart0;
1005     else if ( rxPin == CdcUart1.rxPin ) uart = &CdcUart1;
1006     else if ( rxPin == CdcUart2.rxPin ) uart = &CdcUart2;
1007     else if ( rxPin == CdcUart3.rxPin ) uart = &CdcUart3;
1008     else                                return ( CDC::UART_PORT_ERR );
1009
1010     if (( uart != nullptr ) && ( uart -> uartMode == UART_MODE_8N1 )) {
1011
1012         uart_set_irq_enables( uart -> uartHw, true, false );
1013         uart -> rxBufIndex = 0;
1014         return ( NO_ERR );
1015     }
1016     else if (( uart != nullptr ) && ( uart -> uartMode == UART_MODE_8N1_PIO )) {
1017
1018         return ( NOT_SUPPORTED );
1019     }
1020     else return ( UART_PORT_ERR );
1021 }
1022
1023 uint8_t stopUartRead( uint8_t rxPin ) {
1024
1025     UartInst *uart = nullptr;
1026
1027     if      ( rxPin == CdcUart0.rxPin ) uart = &CdcUart0;
1028     else if ( rxPin == CdcUart1.rxPin ) uart = &CdcUart1;
1029     else if ( rxPin == CdcUart2.rxPin ) uart = &CdcUart2;
1030     else if ( rxPin == CdcUart3.rxPin ) uart = &CdcUart3;
1031
1032     if (( uart != nullptr ) && ( uart ->uartMode == UART_MODE_8N1 )) {
1033
1034         uart_set_irq_enables( uart -> uartHw, false, false );
1035         return ( NO_ERR );
1036     }
1037     else if (( uart != nullptr ) && ( uart -> uartMode == UART_MODE_8N1_PIO )) {
1038
1039         return ( NOT_SUPPORTED );
1040     }
1041     else return ( UART_PORT_ERR );
1042 }
1043
1044 uint8_t getUartBuffer( uint8_t rxPin, uint8_t *buf, uint8_t bufLen ) {
1045
1046     UartInst *uart = nullptr;
1047
1048     if      ( rxPin == CdcUart0.rxPin ) uart = &CdcUart0;
1049     else if ( rxPin == CdcUart1.rxPin ) uart = &CdcUart1;
1050     else if ( rxPin == CdcUart2.rxPin ) uart = &CdcUart2;
1051     else if ( rxPin == CdcUart3.rxPin ) uart = &CdcUart3;
1052     else                                return ( 0 );
1053
1054     if (( uart != nullptr ) && ( uart -> rxBufIndex > 0 ) && ( bufLen > 0 )) {
1055
1056         uint8_t i = 0;
1057         while (( i < uart -> rxBufIndex ) && ( i < bufLen )) {
1058
1059             buf[ i ] = uart -> rxDataBuf[ i ];
1060             i++;
1061         }
1062
1063         return ( i );
1064     }
1065     else return ( 0 );
1066 }
1067
1068 //-------------------------------------------------------------------------------------------------------------
1069 // PWM section. The PICO is quite flexible when it comes to PWM signals. We implement a simple PWM capability.
1070 // There is the frequency which set during configuration and there is the write operation which set the duty
1071 // cycle. The calculations are best described in the PICO C++ SDK. We do the setting of phase, wrap count,
1072 // etc. once when we configure the PWM channel. All the "writePwm" function then will do is to manipulate the
1073 // duty cycle. In other words, when we change the frequency we need to configure again.
1074 //
1075 // There is one small issue left. Channel come in pairs. For some reason there is no call to individually
1076 // set the "inverted" option on a channel. When we set the inverted option for a pin, we currently also set
1077 //  the inverted option for the other channel since we just don't know better. To be correct, all possible
1078 // PWM pins and their "inverted" option would need to be stored somewhere.
1079 //
1080 // To do .... ( there is a way via the pwm_Config CSR field... )
1081 //
1082 // ??? should we have also a kind of PWM pair ? Is that even possible ?
1083 //-------------------------------------------------------------------------------------------------------------
1084 uint8_t configurePwm( uint8_t pwmPin, uint32_t pwmFreqency, bool phaseCorrect, bool inverted ) {
1085
1086     PwmInst *pwm = nullptr;
1087
1088     if      ( pwmPin == cfg.PWM_PIN_0 ) pwm = &CdcPwm0;
```

```
1089        else if ( pwmPin == cfg.PWM_PIN_1 ) pwm = &CdcPwm1;
1090        else if ( pwmPin == cfg.PWM_PIN_2 ) pwm = &CdcPwm2;
1091        else if ( pwmPin == cfg.PWM_PIN_3 ) pwm = &CdcPwm3;
1092        else                                return ( PWM_PIN_ERR );
1093
1094        if ( phaseCorrect ) pwmFreqency = pwmFreqency * 2;
1095
1096        uint32_t sysClock = getCpuFrequency( );
1097        uint32_t clkDiv   = sysClock / pwmFreqency / 4096 + ( sysClock % ( pwmFreqency * 4096 ) != 0 );
1098
1099        if ( clkDiv / 16 == 0 ) clkDiv = 16;
1100
1101        pwm -> pwmPin  = pwmPin;
1102        pwm -> wrap    = sysClock * 16 / clkDiv / pwmFreqency - 1;
1103
1104        pwm_config pwmConfig = pwm_get_default_config( );
1105        gpio_set_function( pwm -> pwmPin, GPIO_FUNC_PWM );
1106        pwm_config_set_wrap( &pwmConfig, pwm -> wrap );
1107        pwm_config_set_phase_correct( &pwmConfig, phaseCorrect );
1108        pwm_config_set_output_polarity( &pwmConfig, inverted, inverted );
1109        pwm_init ( pwm_gpio_to_slice_num( pwm -> pwmPin ), &pwmConfig, false );
1110        pwm_set_clkdiv_int_frac( pwm_gpio_to_slice_num( pwm -> pwmPin ), clkDiv / 16, clkDiv & 0xF );
1111
1112        #if CDC_DEBUG == 1
1113
1114        printf( "PWM Pin: % d, fPwm: % d, phase: % d, inverted: % d, clkDiv: % d, wrap: % d \n",
1115                    pwm -> pwmPin, pwmFreqency,  phaseCorrect, inverted, clkDiv, pwm -> wrap );
1116        #endif
1117
1118        return ( NO_ERR );
1119    }
1120
1121    uint8_t writePwm( uint8_t pwmPin, uint8_t dutyCycle ) {
1122
1123        PwmInst *pwm = nullptr;
1124
1125        if      ( pwmPin == cfg.PWM_PIN_0 ) pwm = &CdcPwm0;
1126        else if ( pwmPin == cfg.PWM_PIN_1 ) pwm = &CdcPwm1;
1127        else if ( pwmPin == cfg.PWM_PIN_2 ) pwm = &CdcPwm2;
1128        else if ( pwmPin == cfg.PWM_PIN_3 ) pwm = &CdcPwm3;
1129        else                                return ( PWM_PIN_ERR );
1130
1131        uint sliceNum = pwm_gpio_to_slice_num( pwmPin );
1132        uint channel  = pwm_gpio_to_channel( pwmPin );
1133
1134        if ( dutyCycle == 0 ) {
1135
1136            pwm_set_enabled( sliceNum, false );
1137            writeDio( pwmPin, false );
1138        }
1139        else if ( dutyCycle == 255 ) {
1140
1141            pwm_set_enabled( sliceNum, false );
1142            writeDio( pwmPin, true );
1143        }
1144        else {
1145
1146            pwm_set_chan_level( sliceNum, channel, ( pwm -> wrap * dutyCycle / 256 ));
1147            pwm_set_enabled( sliceNum, true );
1148        }
1149
1150        return ( NO_ERR );
1151    }
1152
1153    //-------------------------------------------------------------------------------------------------------------
1154    // I2C Section. The PICO has two HW blocks for I2C interfaces. The interface implements a simple read and
1155    // write access to an I2C element. There is a timeout to avoid waiting forever on an operation.
1156    //
1157    //-------------------------------------------------------------------------------------------------------------
1158    uint8_t configureI2C( uint8_t sclPin, uint8_t sdaPin, uint32_t baudRate ) {
1159
1160        I2CInst *i2c = nullptr;
1161
1162        if ((( 1 << sclPin ) & VALID_I2C_0_SCL_PINS ) && (( 1 << sdaPin ) & VALID_I2C_0_SDA_PINS )) {
1163
1164            i2c = &CdcI2C0;
1165            i2c -> i2cHw = i2c0;
1166        }
1167        else if ((( 1 << sclPin ) & VALID_I2C_1_SCL_PINS ) && (( 1 << sdaPin ) & VALID_I2C_1_SDA_PINS )) {
1168
1169            i2c = &CdcI2C1;
1170            i2c -> i2cHw = i2c1;
1171        }
1172        else return ( CDC::I2C_PORT_ERR );
1173
1174        i2c -> sclPin       = sclPin;
1175        i2c -> sdaPin       = sdaPin;
1176        i2c -> baudRate     = baudRate;
1177        i2c -> timeoutValMs = I2C_TIME_OUT_IN_MS;
1178        i2c -> configured   = true;
1179
1180        i2c_init( i2c -> i2cHw, i2c -> baudRate );
1181        i2c_set_slave_mode( i2c -> i2cHw, false, 0 );
1182
1183        gpio_set_function( i2c -> sclPin, GPIO_FUNC_I2C );
1184        gpio_set_function( i2c -> sdaPin, GPIO_FUNC_I2C);
1185        gpio_pull_up( i2c -> sclPin );
1186        gpio_pull_up( i2c -> sdaPin );
1187
```

```
1188        return ( NO_ERR );
1189    }
1190
1191    uint8_t i2cRead( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit ) {
1192
1193        I2CInst *i2c = nullptr;
1194
1195        if      (( CdcI2C0.sclPin == sclPin ) && ( CdcI2C0.configured )) i2c = &CdcI2C0;
1196        else if (( CdcI2C1.sclPin == sclPin ) && ( CdcI2C1.configured )) i2c = &CdcI2C1;
1197        else return ( I2C_PORT_ERR );
1198
1199        auto ret = i2c_read_blocking_until( i2c -> i2cHw,
1200                                            i2cAdr,
1201                                            buf,
1202                                            len,
1203                                            stopBit,
1204                                            make_timeout_time_ms( i2c -> timeoutValMs ));
1205
1206        #if CDC_DEBUG == 1
1207        printf( "i2cRead: scl: %d, i2c: 0x%x, buf: %p, buf[0] %x, buf[1] %x, len: %d, stop: %d\n",
1208                  sclPin, i2cAdr, buf, buf[0], buf[1], len, stopBit );
1209        if ( ret == PICO_ERROR_GENERIC ) printf( "I2C read, PICO generic error\n" );
1210        if ( ret == PICO_ERROR_TIMEOUT ) printf( "I2C read, PICO timeout error\n" );
1211        #endif
1212
1213        if (( ret == PICO_ERROR_GENERIC ) || ( ret == PICO_ERROR_TIMEOUT )) return ( I2C_READ_ERR );
1214
1215        return ( NO_ERR );
1216    }
1217
1218    uint8_t i2cWrite( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit ) {
1219
1220        #if CDC_DEBUG == 1
1221        printf( "i2cWrite: scl: %d, i2c: 0x%x, buf: %p, buf[0] %x, buf[1] %x, len: %d, stop: %d\n",
1222                 sclPin, i2cAdr, buf, buf[0], buf[1], len, stopBit );
1223        #endif
1224
1225
1226        I2CInst *i2c = nullptr;
1227
1228        if      (( CdcI2C0.sclPin == sclPin ) && ( CdcI2C0.configured )) i2c = &CdcI2C0;
1229        else if (( CdcI2C1.sclPin == sclPin ) && ( CdcI2C1.configured )) i2c = &CdcI2C1;
1230        else return ( I2C_PORT_ERR );
1231
1232        auto ret = i2c_write_blocking_until( i2c -> i2cHw,
1233                                             i2cAdr,
1234                                             buf,
1235                                             len,
1236                                             stopBit,
1237                                             make_timeout_time_ms( i2c -> timeoutValMs ));
1238
1239        #if CDC_DEBUG == 1
1240        if ( ret == PICO_ERROR_GENERIC ) printf( "I2C write, PICO generic error\n" );
1241        if ( ret == PICO_ERROR_TIMEOUT ) printf( "I2C write, PICO timeout error\n" );
1242        #endif
1243
1244        if (( ret == PICO_ERROR_TIMEOUT) || ( ret == PICO_ERROR_GENERIC ) || ( ret != len )) return ( I2C_WRITE_ERR );
1245
1246        return ( NO_ERR );
1247    }
1248
1249    //-----------------------------------------------------------------------------------------------------------
1250    // SPI interface section. The PICO features two SPI HW blocks. We implement a simple SPI interface with a
1251    // a fixed set of SPI options for frequency, bit order and mode. One day this may change. We do not take
1252    // care of the chip select stuff and expect that the caller manages the select pin.
1253    //
1254    //-----------------------------------------------------------------------------------------------------------
1255    uint8_t configureSPI( uint8_t sclkPin, uint8_t mosiPin, uint8_t misoPin, uint32_t baudRate ) {
1256
1257        SPIInst *spi = nullptr;
1258
1259         if ((( 1 << sclkPin ) & VALID_SPI_0_SCK_PINS ) &&
1260             (( 1 << mosiPin ) & VALID_SPI_0_TX_PINS  ) &&
1261             (( 1 << misoPin ) & VALID_SPI_0_RX_PINS  )) {
1262
1263            spi = &CdcSPI0;
1264            spi -> spiHw = spi0;
1265        }
1266        else if ((( 1 << sclkPin ) & VALID_SPI_1_SCK_PINS ) &&
1267                 (( 1 << mosiPin ) & VALID_SPI_1_TX_PINS  ) &&
1268                 (( 1 << misoPin ) & VALID_SPI_1_RX_PINS  )) {
1269
1270            spi = &CdcSPI1;
1271            spi -> spiHw = spi1;
1272        }
1273        else return ( SPI_PORT_ERR );
1274
1275        spi -> mosiPin     = mosiPin;
1276        spi -> misoPin     = misoPin;
1277        spi -> sclkPin     = sclkPin;
1278        spi -> frequency   = SPI_FREQUENCY;
1279        spi -> configured  = true;
1280        spi -> active      = false;
1281
1282        spi_init( spi -> spiHw, SPI_FREQUENCY );
1283
1284        spi_set_format( spi -> spiHw,     // SPI instance
1285                        8,                // Number of bits per transfer
1286                        SPI_CPOL_1,       // Polarity (CPOL)
```

```
1287                    SPI_CPHA_1,       // Phase (CPHA)
1288                    SPI_MSB_FIRST );
1289
1290    gpio_set_function( sclkPin, GPIO_FUNC_SPI );
1291    gpio_set_function( mosiPin, GPIO_FUNC_SPI );
1292    gpio_set_function( misoPin, GPIO_FUNC_SPI );
1293
1294    return ( NO_ERR );
1295}
1296
1297uint8_t  spiBeginTransaction( uint8_t sclkPin, uint8_t csPin ) {
1298
1299    SPIInst *spi = nullptr;
1300
1301    if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
1302    else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
1303    else return ( SPI_PORT_ERR );
1304
1305    if ( spi -> active ) {
1306
1307        // ??? should we check who is active and just ignore when the same ? else "error " ?
1308
1309        return ( NO_ERR );
1310
1311    } else {
1312
1313        spi -> active     = true;
1314        spi -> selectPin  = csPin;
1315
1316        CDC::writeDio( csPin, false );
1317        return ( NO_ERR );
1318    }
1319}
1320
1321uint8_t spiEndTransaction( uint8_t sclkPin, uint8_t csPin ) {
1322
1323    SPIInst *spi = nullptr;
1324
1325    if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
1326    else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
1327    else return ( SPI_PORT_ERR );
1328
1329    if ( spi -> active ) {
1330
1331        // ??? check that this is the correct pin ?
1332
1333        CDC::writeDio( csPin, true );
1334
1335        spi -> active     = false;
1336        spi -> selectPin  = UNDEFINED_PIN;
1337
1338        return ( NO_ERR );
1339
1340    }
1341    else return ( NO_ERR ); // ???  "error "  not active...
1342}
1343
1344uint8_t spiRead( uint8_t sclkPin, uint8_t *buf, uint32_t len ) {
1345
1346    SPIInst *spi = nullptr;
1347
1348    if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
1349    else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
1350    else return ( SPI_PORT_ERR );
1351
1352    if ( spi -> active ) {
1353
1354        int bytesRead = spi_read_blocking( spi -> spiHw, 0, buf, len );
1355        return ( NO_ERR );
1356
1357    } else return ( NO_ERR ); // ??? fix : not active ...
1358}
1359
1360uint8_t spiWrite( uint8_t sclkPin, uint8_t *buf, uint32_t len ) {
1361
1362    SPIInst *spi = nullptr;
1363
1364    if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
1365    else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
1366    else return ( SPI_PORT_ERR );
1367
1368    if ( spi -> active ) {
1369
1370        spi_write_blocking( spi -> spiHw, buf, len );
1371        return ( NO_ERR );
1372
1373    } else return ( NO_ERR ); // ??? fix : not active ...
1374}
1375
1376//-----------------------------------------------------------------------------------------------------------
1377// Print out the Config Structure.
1378//
1379//-----------------------------------------------------------------------------------------------------------
1380void printConfigInfo( CdcConfigDesc *ci ) {
1381
1382    printf( "CDC Pin Configuration Info ( status %d ): \n", ci -> CFG_STATUS );
1383
1384    printf( "Pfail pin: %2d, ExtInt pin: %2d \n", ci -> PFAIL_PIN, ci -> EXT_INT_PIN );
1385
```

```
1386        printf( "ReadyLed pin: %2d, ActiveLed pin: %2d \n", ci -> READY_LED_PIN, ci -> ACTIVE_LED_PIN );
1387
1388        printf( "DIO pins ( 0 .. 7 ): %2d %2d %2d %2d %2d %2d %2d %2d\n",
1389                ci -> DIO_PIN_0, ci -> DIO_PIN_1, ci -> DIO_PIN_2, ci -> DIO_PIN_3,
1390                ci -> DIO_PIN_4, ci -> DIO_PIN_5, ci -> DIO_PIN_6, ci -> DIO_PIN_7 );
1391
1392        printf( "DIO pins ( 8 .. 15 ): %2d %2d %2d %2d %2d %2d %2d %2d\n",
1393                ci -> DIO_PIN_8, ci -> DIO_PIN_9, ci -> DIO_PIN_10, ci -> DIO_PIN_11,
1394                ci -> DIO_PIN_12, ci -> DIO_PIN_13, ci -> DIO_PIN_14, ci -> DIO_PIN_15 );
1395
1396        printf( "ADC pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
1397                ci -> ADC_PIN_0, ci -> ADC_PIN_1, ci -> ADC_PIN_2, ci -> ADC_PIN_3 );
1398
1399        printf( "PWM pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
1400                ci -> PWM_PIN_0, ci -> PWM_PIN_1, ci -> PWM_PIN_2, ci -> PWM_PIN_3 );
1401
1402        printf( "UART RX pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
1403                ci -> UART_RX_PIN_0, ci -> UART_RX_PIN_1, ci -> UART_RX_PIN_2, ci -> UART_RX_PIN_3 );
1404
1405        printf( "UART TX pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
1406                ci -> UART_TX_PIN_0, ci -> UART_TX_PIN_1, ci -> UART_TX_PIN_2, ci -> UART_TX_PIN_3 );
1407
1408        printf( "SPI0 Pins: MOSI: %2d, MISO: %2d, SCLK: %2d \n",
1409                ci -> SPI_MOSI_PIN_0, ci -> SPI_MISO_PIN_0, ci -> SPI_SCLK_PIN_0 );
1410
1411        printf( "SPI1 Pins: MOSI: %2d, MISO: %2d, SCLK: %2d \n",
1412                ci -> SPI_MOSI_PIN_1, ci -> SPI_MISO_PIN_1, ci -> SPI_SCLK_PIN_1 );
1413
1414        printf( "NVM I2C Pins: SCL: %2d, SDA: %2d, I2C Root: 0x%x \n",
1415                ci -> NVM_I2C_SCL_PIN, ci -> NVM_I2C_SDA_PIN, ci -> NVM_I2C_ADR_ROOT );
1416
1417        printf( "EXT I2C Pins: SCL: %2d, SDA: %2d, I2C Root: 0x%x \n",
1418                ci -> EXT_I2C_SCL_PIN, ci -> EXT_I2C_SDA_PIN, ci -> EXT_I2C_ADR_ROOT );
1419
1420        printf( "\n" );
1421
1422    }
1423
1424    }; // namespace CDC
```