A Layout Control System for Model Railroads

Helmut Fieres

November 27, 2024

Contents

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.
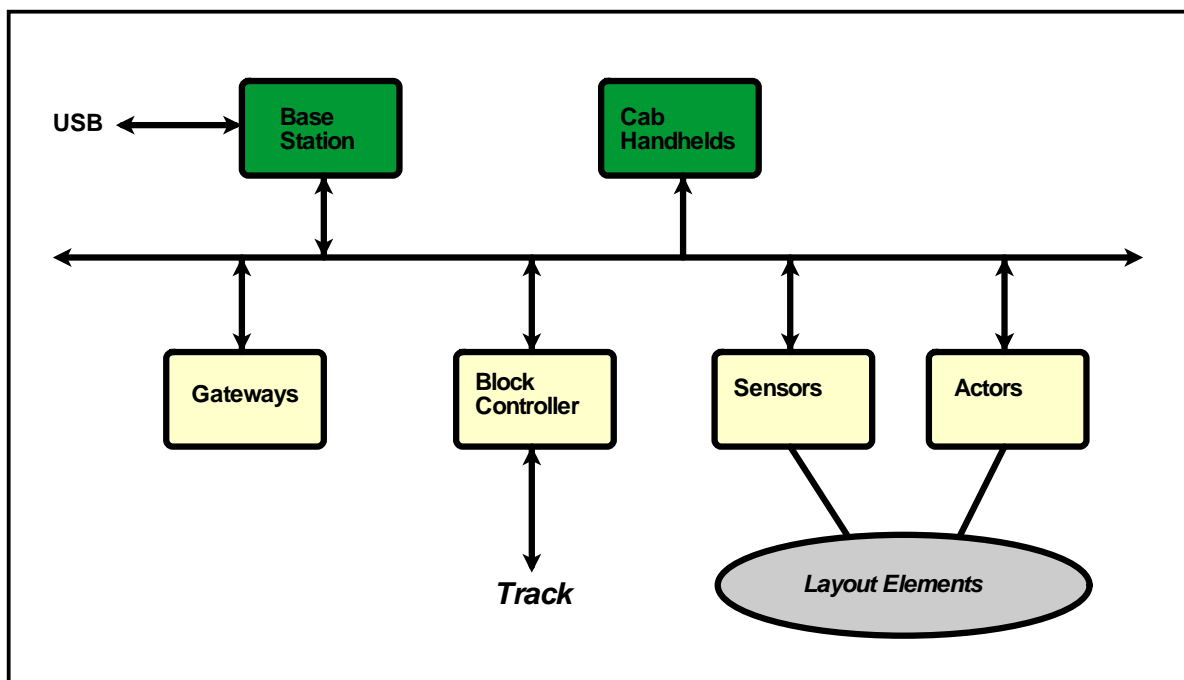
2.1 Layout Control Bus

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 outline the message data formats as they flow on the layout control bus. It is the foundation of the layout control system, so let's have a first brief look at all the messages defined. 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. The layout control system messages 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

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. 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.

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.

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.

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	

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

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

Table 3.3: Node and Port Management

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

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

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)

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					

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.

Table 3.8: DCC Locomotive Decoder Management

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

Locomotive decoders contain configuration variables too. They are called CV vari-
ables. 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 sup-
ports 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 3.9: 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.

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.

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

3.9   Raw DCC Packet Management

The base station allows to send raw DCC packets to the track. The (SEND-DCC3), (SEND-DCC4), (SEND-DCC5) and (SEND-DCC6) are the messages to send these packets. Any node can broadcast such a message, the base station is the target for these messages and will just send them without further checking. So you better put the DCC standard document under your pillow.

Table 3.12: RRaw DCC Packet Management

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

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

Table 3.13: RRaw DCC Packet Management

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

3.10   DCC errors and status

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

Table 3.14: RRaw DCC Packet Management

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

3.11   Analog Engines

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

3.12   Summary

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

4 Message Protocols

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

4.1 Node startup

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

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

Table 4.1: Node startup

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

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

The above nodeId setup scheme requires the presence of a central node, such a base station, to validate and assign node identifiers. In addition, the nodeId can also be

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

4.2 Switching between Modes

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

Table 4.2: Switching between Configuration and Operations mode

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

4.3 Setting a new Node Id

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

Table 4.3: Switching between Configuration and Operations mode

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

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

4.4   Node Ping

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

Table 4.4: Node ping

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

4.5   Node and Port Reset

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

Table 4.5: Node and Port Reset

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

4.6   Node and Port Access

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

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

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...

??? struct of nodeMap ?

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.

??? old map, replace

```
1   struct LcsPortMapEntry {
2
3       uint16_t              flags                     = 0;
4       uint8_t               portType                  = 0;
5
6       uint16_t              eventNodeId               = NIL_NODE_ID;
7       uint16_t              eventId                   = NIL_EVENT_ID;
8       uint8_t               eventAction               = PEA_EVENT_IDLE;
9       uint16_t              eventValue                = 0;
10      uint32_t              eventTimeStamp            = 0L;
11      uint8_t               eventDelayTime            = 0;
12
13      char                  portName[ MAX_NODE_NAME_SIZE ]       = { 0 };
14      uint16_t              portAttrMap[ MAX_PORT_ATTR_MAP_SIZE ]   = { 0 };
15  };
```

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

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

6.3   Node and Port Items

The term "item" came up numerous times by now. Nodes and ports features to access their attributes through an **item Id**. An item Id is just a number in the range from 1

to 255. Here is the definition from the library include file. The include file also contains the item numbers for the reserved node info and control items.

Table 6.1: 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

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

6.4 Event Map

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

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

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

Like all other maps, the event map is stored in two places. The non-volatile version of the eventMap is an array of event map entries. Whenever a new entry is added, a free entry is used to store this information. The memory version of the event map is a

sorted version of all used non-volatile entries. The entries are first sorted by event Id. For entries with the same event Id, the port Id is then sorted in ascending order.

In addition to the search function, event map entries can be added and deleted by specifying the eventId and portId. EventMap entries can also be accessed by their position in the event map. This is necessary to read out the event map for example 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

```
... code snippet here ...
```

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

```
... code snippet here ...
```

6.9 Driver map

for extension boards to be explained later...

```
... 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      #include "LcsRuntimeLib.h"
3
4      // code to initialize HW structures ( to be discussed later... )
5      CDC::CdcConfigInfo cfg;
6
7      ...
8
9      int rStat;
10
11     // ... create the LCS library object and startup
12     rStat = LCS::initRuntime( &cfg );
13
14     // ... register callbacks with lcsLib
15     // ... other node specific things to do
16     // ... invoke the run method. We never return from it ...
17     LcsRuntimeLib::run( );
```

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   int         rStat;
2   uint16_t    value;
3
4   rStat = LCS::nodeInfo( 0, NPI_PORT_MAP_ENTRIES, &value );
5   if ( rStat != ALL_OK ) { ... }
```

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   int         rStat = ALL_OK;
2   uint16_t    value = 1;
3
4   rStat = LCS::nodeControl(  0, NPC_SET_READY_LED, &value );
5   if ( rStat != ALL_OK ) { ... }
```

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   int      rStat      = ALL_OK;
2   uint16_t nodeId      = LCS::getNodeId( );
3   uint16_t eventId     = 200;
4
5   ...
6
7   rStat = sendEventOn( nodeId, eventId );
8   if ( rStat != ALL_OK ) { ... }
```

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

7.7 Summary

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

8 RtLib Callbacks

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

8.1 General Callbacks

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

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

8.2 Node and Port Initialization Callback

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

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

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

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

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

8.4   Node and Port Control and Info Callback

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

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

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

8.5   Inbound Event Callback

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

```
1  // ... the inbound event handler routine
2  void eventHandler ( uint16_t nodeId, uint8_t portId, uint8_t eAction, uint16_t eId, uint16_t eData ) { ... }
3  ...
4  // during module firmware initialization ...
5  lcsLib -> registerPortEventCallback( eventHandler )
```

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

8.6   Console Command Line Callback

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

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

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

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

8.7   DCC Message Callback

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

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

8.8   RailCom Message Callback

Railcom is a concept for the DCC decoders to communicate back. DCC is inherently a broadcast protocol just like a radio station. There was no way to communicate back. Railcom was design to allow for a decoder to send back data when the DCC channel is

told to "pause". The chapter on the DCC subsystem will explain DCC and RailCom in greater detail. The Railcom Message callback is the function callback that will be invoked when a RailCom Messages is received.

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

8.9   LCS Periodic Task Callback

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

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

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

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

8.10   Summary

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

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

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

""" ¡lcs-command-char [arguments] ¿

"""

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

9.1 Configuration Mode Commands

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

—Command — Arguments — Operation — —:—-—:——:—— —!c — — enter node configuration mode — —!o — — enter node operations 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 Tests

11.1 Schematics

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

11.1.1 part 1
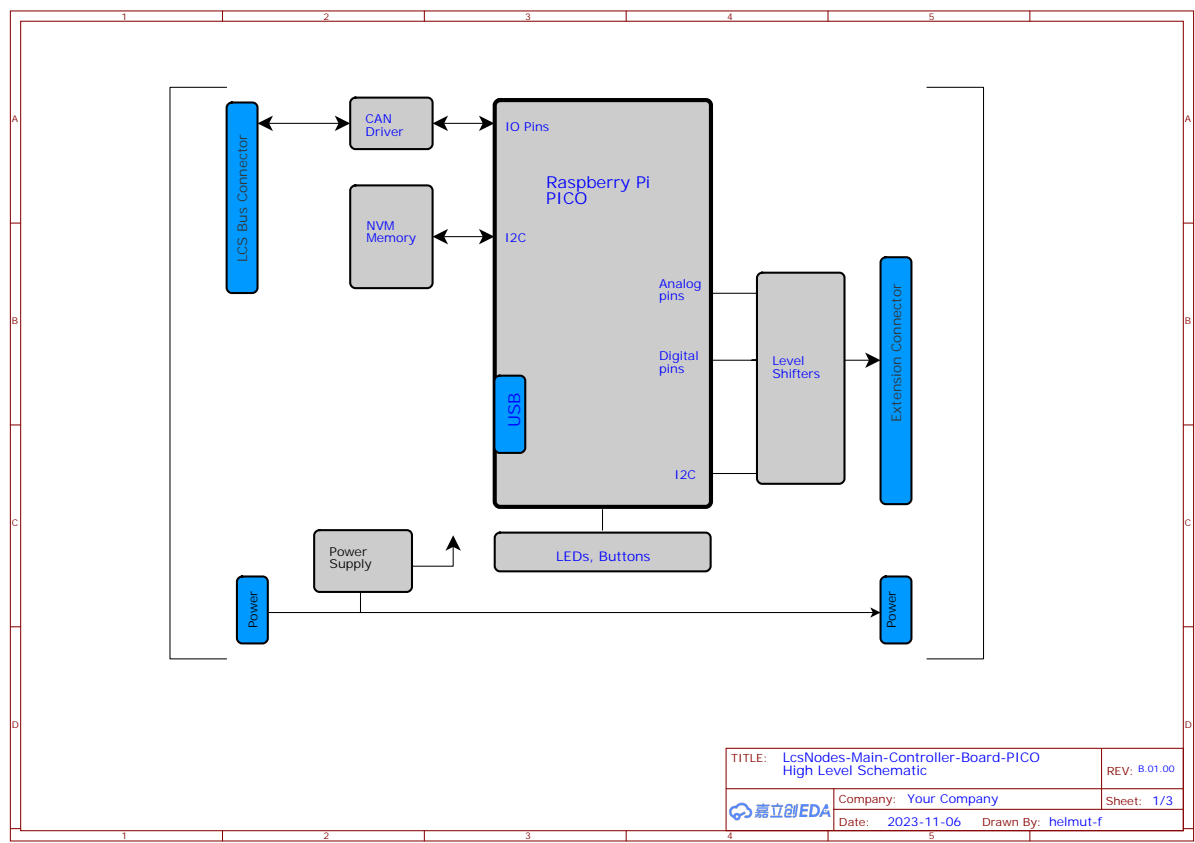
11.1.2 part 2
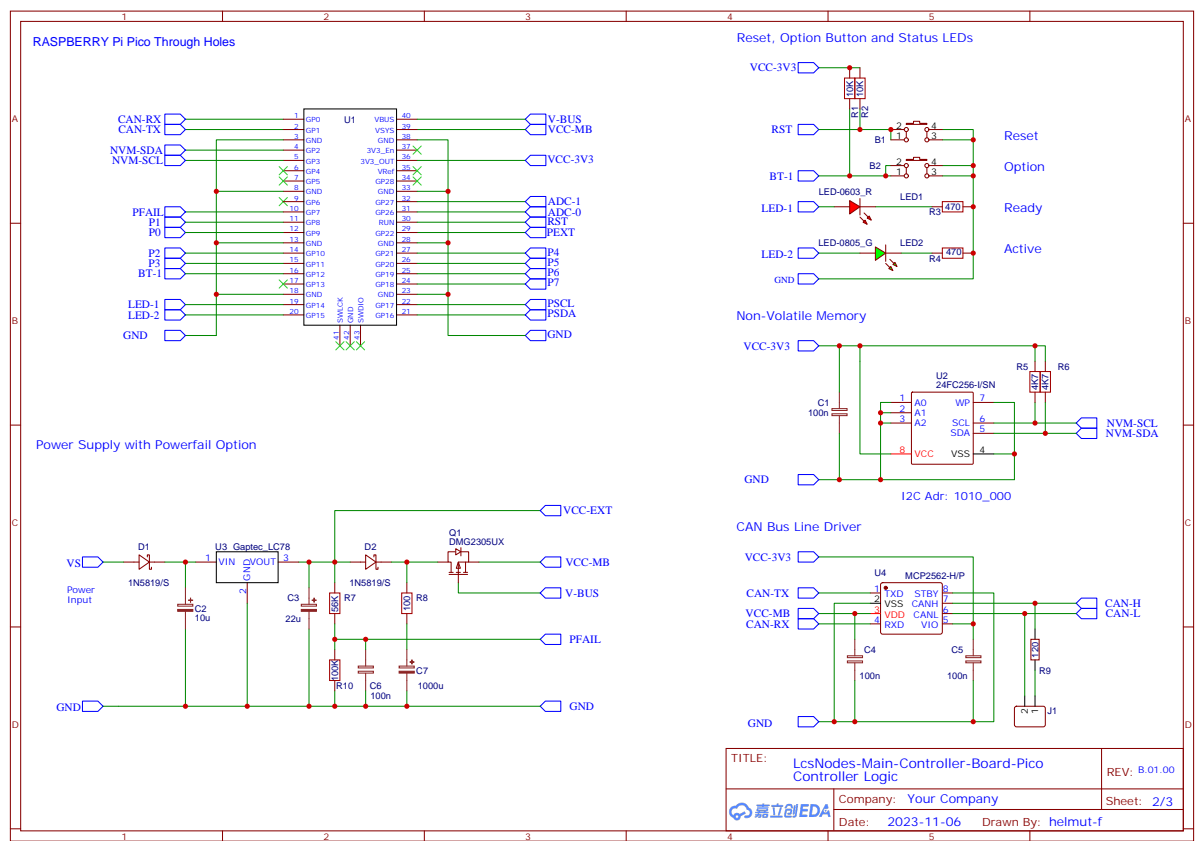
11.1.3 part 3
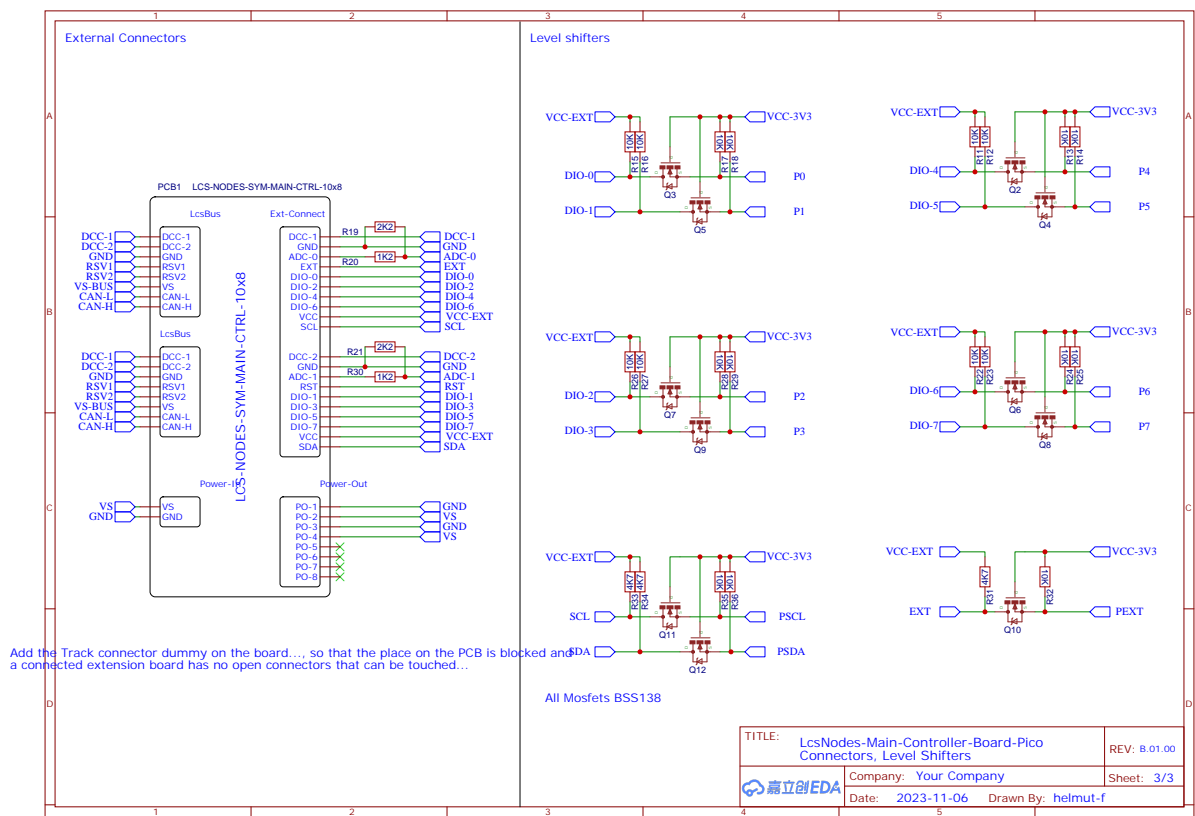
11.2 Code Snippets

```
int main ( int argc , char **argv ) {

    return ( 0 );
}
```

11.3 Lists

11.3.1 A simple list

- First bullet point

- Second bullet point

- Third bullet point

11.3.2 An instruction word layout

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

1		3	6
Test			

12   Listings test

Here is a little test how a listing part might be shown ...

12.1 Base Station

```
  1   //------------------------------------------------------------------------------------------------------------
  2   //
  3   // LCS Base Station - Include file
  4   //
  5   //------------------------------------------------------------------------------------------------------------
  6   //
  7   // LCS - Base Station
  8   // Copyright (C) 2019 - 2024  Helmut Fieres
  9   //
 10   // This program is free software: you can redistribute it and/or modify it under the terms of the GNU General
 11   // Public License as published by the Free Software Foundation, either version 3 of the License, or (at your
 12   // option) any later version.
 13   //
 14   // This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
 15   // implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
 16   // for more details.
 17   //
 18   // You should have received a copy of the GNU General Public License along with this program. If not, see
 19   // http://www.gnu.org/licenses
 20   //
 21   //  GNU General Public License:  http://opensource.org/licenses/GPL-3.0
 22   //
 23   //------------------------------------------------------------------------------------------------------------
 24   #ifndef LcsBaseStation_h
 25   #define LcsBaseStation_h
 26
 27   #include "LcsCdcLib.h"
 28   #include "LcsRuntimeLib.h"
 29
 30   //------------------------------------------------------------------------------------------------------------
 31   // The base station maintains a set of debug flags. The overall concept is very similar to the LCS runtime
 32   // library debug mask. Then following debug flags are defined:
 33   //
 34   //      DBG_BS_CONFIG                  -   DEBUG base station enabled
 35   //      DBG_BS_SESSION                 -   show the session management actions
 36   //      DBG_BS_LCS_MSG_INTERFACE       -   show the incoming LCS messages
 37   //      DBG_BS_TRACK_POWER_MGMT        -   show the track power measurement data
 38   //      DBG_BS_DCC_ACK_DETECT          -   display decoder ACK power measurements
 39   //      DBG_BS_CHECK_ALIVE_SESSIONS    -   displays that a session seems no longer be alive
 40   //      DBG_BS_RAILCOM                 -   show the RailCom activity
 41   //
 42   // The way to use these flags is for example:
 43   //
 44   //      if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_SESSION ))
 45   //
 46   // ??? should have a command to set the debug mask on the fly...
 47   //------------------------------------------------------------------------------------------------------------
 48   enum BaseStationDebugFlags : uint16_t {
 49
 50       DBG_BS_CONFIG                  = 1 << 15,      // DEBUG base station enabled
 51
 52       DBG_BS_SESSION                 = 1 << 0,       // show the session management actions
 53       DBG_BS_LCS_MSG_INTERFACE       = 1 << 1,       // show the incoming LCS messages
 54       DBG_BS_TRACK_POWER_MGMT        = 1 << 2,       // show the track power measurement data
 55       DBG_BS_DCC_ACK_DETECT          = 1 << 3,       // display decoder ACK power measurements
 56       DBG_BS_CHECK_ALIVE_SESSIONS    = 1 << 4,       // displays that a session seems no longer be alive
 57       DBG_BS_RAILCOM                 = 1 << 5        // show the RailCom activity
 58
 59   };
 60
 61   //------------------------------------------------------------------------------------------------------------
 62   // Base station errors. Note that they need to be in the assigned to the user number range of errors defined
 63   // in the LCS runtime library.
 64   //
 65   //------------------------------------------------------------------------------------------------------------
 66   enum BaseStationErrors : uint8_t {
 67
 68       BASE_STATION_ERR_BASE          = 128,
 69
 70       ERR_NO_SVC_MODE                = BASE_STATION_ERR_BASE + 1,
 71       ERR_CV_OP_FAILED               = BASE_STATION_ERR_BASE + 2,
 72
 73       ERR_LOCO_NOT_FOUND             = BASE_STATION_ERR_BASE + 4,
 74       ERR_SESSION_NOT_FOUND          = BASE_STATION_ERR_BASE + 5,
 75       ERR_LOCO_SESSION_ALLOCATE      = BASE_STATION_ERR_BASE + 6,
 76       ERR_LOCO_SESSION_CANCELLED     = BASE_STATION_ERR_BASE + 7,
 77
 78       ERR_SESSION_SETUP              = BASE_STATION_ERR_BASE + 9,
 79       ERR_MSG_INTERFACE_SETUP        = BASE_STATION_ERR_BASE + 10,
 80       ERR_DCC_TRACK_CONFIG           = BASE_STATION_ERR_BASE + 11,
 81       ERR_DCC_PIN_CONFIG             = BASE_STATION_ERR_BASE + 12,
 82
 83       ERR_NVM_HW_SETUP               = BASE_STATION_ERR_BASE + 15,
 84       ERR_PIO_HW_SETUP               = BASE_STATION_ERR_BASE + 16
 85   };
 86
 87   //------------------------------------------------------------------------------------------------------------
 88   // DCC packet definition. A DCC packet is the payload data without the checksum. Besides the length in bytes
 89   // and the buffer, there is a repeat counter to specify how often this packet will be repeatedly transmitted
 90   // after the first transmission. Currently, a DCC packet is at most 15 bytes long, excluding the checksum
 91   // byte. This is true for XPOM and DCC-A support, otherwise it is historically a maximum of 6 bytes.
 92   //
 93   //------------------------------------------------------------------------------------------------------------
 94   const uint8_t DCC_PACKET_SIZE = 16;
```

```
 95
 96   struct DccPacket {
 97
 98       uint8_t len;
 99       uint8_t repeat;
100       uint8_t buf[ DCC_PACKET_SIZE ];
101   };
102
103   //----------------------------------------------------------------------------------------------------
104   // DCC packet payload data definitions we need often, so these constants come in handy.
105   //
106   //----------------------------------------------------------------------------------------------------
107   const uint8_t   idleDccPacketData[ ]   = { 0xFF, 0x00 };
108   const uint8_t   resetDccPacketData[ ]  = { 0x00, 0x00 };
109   const uint8_t   eStopDccPacketData[ ]  = { 0x00, 0x01 };
110
111   //----------------------------------------------------------------------------------------------------
112   // Setup options to set for the DCC track. They are set when the track object is created.
113   //
114   //   DT_OPT_SERVICE_MODE_TRACK  - The track is a PROG track.
115   //   DT_OPT_CUTOUT              - The track is configured to emit a cutout during the DCC packet preamble.
116   //   DT_OPT_RAILCOM             - The track support Railcom detection.
117   //
118   //----------------------------------------------------------------------------------------------------
119   enum DccTrackOptions : uint16_t {
120
121       DT_OPT_DEFAULT_SETTING      = 0,
122       DT_OPT_SERVICE_MODE_TRACK   = 1 << 0,
123       DT_OPT_CUTOUT               = 1 << 1,
124       DT_OPT_RAILCOM              = 1 << 1
125   };
126
127   //----------------------------------------------------------------------------------------------------
128   // The DCC track object has a set of flags to indicate its current status.
129   //
130   //   DT_F_POWER_ON              - The track is under power.
131   //   DT_F_POWER_OVERLOAD        - An overload situation was detected.
132   //   DT_F_MEASUREMENT_ON        - The power measurement is enabled.
133   //   DT_F_SERVICE_MODE_ON       - The track is currently in service mode, i.e. is a PROG track.
134   //   DT_F_CUTOUT_MODE_ON        - The track has the cutout generation enabled.
135   //   DT_F_RAILCOM_MODE_ON       - The track has the railcom detect enabled.
136   //   DT_F_RAILCOM_MSG_PENDING   - If railcom is enabled, a received datagram is indicated.
137   //   DT_F_CONFIG_ERROR          - The passed configuration descriptor has invalid options configured.
138   //
139   //----------------------------------------------------------------------------------------------------
140   enum DccTrackFlags : uint16_t {
141
142       DT_F_DEFAULT_SETTING      = 0,
143       DT_F_POWER_ON             = 1 << 0,
144       DT_F_POWER_OVERLOAD       = 1 << 1,
145       DT_F_MEASUREMENT_ON       = 1 << 2,
146       DT_F_SERVICE_MODE_ON      = 1 << 3,
147       DT_F_CUTOUT_MODE_ON       = 1 << 4,
148       DT_F_RAILCOM_MODE_ON      = 1 << 5,
149       DT_F_DCC_PACKET_PENDING   = 1 << 6,
150       DT_F_RAILCOM_MSG_PENDING  = 1 << 7,
151       DT_F_CONFIG_ERROR         = 1 << 15
152   };
153
154   //----------------------------------------------------------------------------------------------------
155   // The following constants are for the current consumption RMS measurement. The idea is to record the measured
156   // ADC values in a circular buffer, every time a certain amount of milliseconds has passed. This work is done
157   // by the DCC track state machine as part of the power on state.
158   //
159   //----------------------------------------------------------------------------------------------------
160   const uint8_t   PWR_SAMPLE_BUF_SIZE             = 64;
161   const uint32_t  PWR_SAMPLE_TIME_INTERVAL_MILLIS   = 16;
162
163   //----------------------------------------------------------------------------------------------------
164   // The RailCom buffer size. During the cutout period up to eight bytes of raw data are sent by the decoder if
165   // the Railcom option is enabled.
166   //
167   //----------------------------------------------------------------------------------------------------
168   const uint8_t   RAILCOM_BUF_SIZE = 8;
169
170   //----------------------------------------------------------------------------------------------------
171   // The session map options. These options initially set when the base station starts. They are used to
172   // set the flags, which are then used for processing the the actual settings.
173   //
174   //   SM_KEEP_ALIVE_CHECKING  - enable keep alive checking. When enabled, the locomotive session need to receive
175   //                             a keep alive LCS message periodically.
176   //   SM_ENABLE_REFRESH       - refresh the session data. This will send the locomotive speed and direction as
177   //                             well as the function flags periodically in a round robin processing of the
178   //
179   //----------------------------------------------------------------------------------------------------
180   enum SessionMapOptions : uint16_t {
181
182       SM_OPT_DEFAULT_SETTING       = 0,
183       SM_OPT_KEEP_ALIVE_CHECKING   = 1 << 0,
184       SM_OPT_ENABLE_REFRESH        = 1 << 1
185   };
186
187   //----------------------------------------------------------------------------------------------------
188   // The session map flags. The apply to all sessions in the session map. The initial values are copied from
189   // session option initial values.
190   //
191   //   SM_F_KEEP_ALIVE_CHECKING  - enable keep alive checking. When enabled, the locomotive session need to receive
192   //                               a keep alive LCS message periodically.
193   //   SM_F_ENABLE_REFRESH       - refresh the session data. This will send the locomotive speed and direction as
```

```
194  //                                  well as the function flags periodically in a round robin processing of the
195  //
196  //-------------------------------------------------------------------------------------------------------------
197  enum SessionMapFlags : uint16_t {
198
199      SM_F_DEFAULT_SETTING        = 0,
200      SM_F_KEEP_ALIVE_CHECKING    = 1 << 0,
201      SM_F_ENABLE_REFRESH         = 1 << 1
202  };
203
204  //-------------------------------------------------------------------------------------------------------------
205  // Each session map entry has a set of flags.
206  //
207  //   SME_ALLOCATED           - the session is allocated, the entry valid.
208  //   SME_COMBINED_REFRESH    - locomotive speed/dir and functions are refreshed using the combined DCC packet.
209  //   SME_SPDIR_REFRESH       - locomotive speed/dir are refreshed.
210  //   SME_FUNC_REFRESH        - locomotive functions are refreshed.
211  //   SME_DISPATCHED          -
212  //   SME_SHARED              -
213  //
214  //
215  // ??? when the base station has a config value of using the DCC spdir/func command, these flags need to be
216  // named slightly different. Should we still have the option to enable or disable it even though the base
217  // station can do it ? A decoder might not support this packet type...
218  //-------------------------------------------------------------------------------------------------------------
219  enum SessionMapEntryFlags : uint16_t {
220
221      SME_DEFAULT_SETTING     = 0,
222      SME_ALLOCATED           = 1 << 0,
223      SME_COMBINED_REFRESH    = 1 << 1,
224      SME_SPDIR_ONLY_REFRESH  = 1 << 2, // ??? phase out...
225      SME_SPDIR_REFRESH       = 1 << 2,
226      SME_FUNC_REFRESH        = 1 << 3,
227      SME_DISPATCHED          = 1 << 4,
228      SME_SHARED              = 1 << 5
229  };
230
231  //-------------------------------------------------------------------------------------------------------------
232  // The base station items for nodeInfo and nodeControl calls .... tbd
233  //
234  // ??? the are mapped in the MEM / NVM range as well as in the USER range.
235  // ??? how to do it consistently and understandably ?
236  //-------------------------------------------------------------------------------------------------------------
237  enum BaseStationInfoItems : uint8_t {
238
239      // or use GET in all constants
240
241      BS_ITEM_SESSION_MAP_OPTIONS   = 128,
242      BS_ITEM_SESSION_MAP_FLAGS     = 129,
243      BS_ITEM_MAX_SESSIONS          = 130,
244      BS_ITEM_ACTIVE_SESSIONS       = 131,
245
246      BS_ITEM_INIT_CURRENT_VAL      = 140,
247      BS_ITEM_LIMIT_CURRENT_VAL     = 140,
248      BS_ITEM_MAX_CURRENT_VAL       = 140,
249      BS_ITEM_ACTUAL_CURRENT_VAL    = 140,
250
251      // thresholds
252
253      // eventID to send for events ?
254
255  };
256
257  //-------------------------------------------------------------------------------------------------------------
258  //
259  //
260  //-------------------------------------------------------------------------------------------------------------
261  const uint32_t  MAIN_TRACK_STATE_TIME_INTERVAL  = 10;
262  const uint32_t  PROG_TRACK_STATE_TIME_INTERVAL  = 10;
263  const uint32_t  SESSION_REFRESH_TASK_INTERVAL   = 50;
264
265  const uint16_t  MAX_CAB_SESSIONS                = 64;
266
267  //-------------------------------------------------------------------------------------------------------------
268  // For creating the Loco Session object the session map object is described by the following descriptor.
269  //
270  //-------------------------------------------------------------------------------------------------------------
271  struct LcsBaseStationSessionMapDesc {
272
273      uint16_t    options      = SM_OPT_DEFAULT_SETTING;
274      uint16_t    maxSessions  = MAX_CAB_SESSIONS;
275  };
276
277  //-------------------------------------------------------------------------------------------------------------
278  // For creating the DCC track object, the track is described by the data structure below. In addition to the
279  // hardware pins enablePin, dcc1Pin1, dccPin2 and sensePin, there are the limits for current consumption
280  // 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
479 // engine, only a subset of the fields is actually used. Nevertheless, even for an analog engine we will
480 // have a session. The base station will however not generate packets for this engine.
481 //
482 //-------------------------------------------------------------------------------------------------------
483 struct SessionMapEntry {
484
485   uint16_t        flags              = SME_DEFAULT_SETTING;
486   uint16_t        cabId              = LCS::NIL_CAB_ID;
487   uint8_t         speed              = 0;
488   uint8_t         speedSteps         = 128;
489   uint8_t         direction          = 0;
490   uint8_t         engineState        = 0;
```

```
491     uint8_t          nextRefreshStep      = 0;
492     unsigned long    lastKeepAliveTime    = 0;
493     uint8_t          functions[ LCS::MAX_DCC_FUNC_GROUP_ID ] = { 0 };
494
495   };
496
497   //-------------------------------------------------------------------------------------------------------
498   // The loco session object is the central data structure for the base station locomotive management. For a
499   // DCC type engine it manages the loco sessions and assembles the DCC packets and drives the DCC track objects
500   // to send out the relevant DCC packages. For an analog engine it will just manage the session entry and
501   // communicate via the LCS bus with the block controller that actually owns the engine at the moment.
502   //
503   //-------------------------------------------------------------------------------------------------------
504   struct LcsBaseStationLocoSession {
505
506     public:
507
508        LcsBaseStationLocoSession( );
509
510        uint8_t setupSessionMap(
511
512           LcsBaseStationSessionMapDesc  *sessionMapDesc,
513           LcsBaseStationDccTrack        *mainTrack,
514           LcsBaseStationDccTrack        *progTrack
515        );
516
517        uint8_t                 requestSession( uint16_t cabId, uint8_t mode, uint8_t *sId );
518        uint8_t                 releaseSession( uint8_t sId );
519        uint8_t                 updateSession( uint8_t sId, uint8_t flags );
520
521        uint8_t                 markSessionAlive( uint8_t sId );
522        void                    refreshActiveSessions( );
523        uint32_t                getSessionKeepAliveInterval( );
524
525        uint16_t                getOptions( );
526        uint16_t                getFlags( );
527        uint8_t                 getSessionMapHwm( );
528        uint8_t                 getActiveSessions( );
529        uint8_t                 getSessionIdByCabId( uint16_t cabId );
530        void                    emergencyStopAll( );
531
532        uint8_t                 setThrottle( uint8_t sId, uint8_t speed, uint8_t direction );
533        uint8_t                 setDccFunctionBit( uint8_t sId, uint8_t funcNum, uint8_t val );
534        uint8_t                 setDccFunctionGroup( uint8_t sId, uint8_t fGroup, uint8_t dccByte );
535
536        uint8_t                 writeCVMain( uint8_t sId, uint16_t cvId, uint8_t mode, uint8_t val );
537        uint8_t                 writeCVByteMain( uint8_t sId, uint16_t cvId, uint8_t val );
538        uint8_t                 writeCVBitMain( uint8_t sId, uint16_t cvId, uint8_t bitPos, uint8_t val );
539
540        uint8_t                 readCV( uint16_t cvId, uint8_t mode, uint8_t *val );
541        uint8_t                 readCVByte( uint16_t cvId, uint8_t *val );
542        uint8_t                 readCVBit( uint16_t cvId, uint8_t bitPos, uint8_t *val );
543
544        uint8_t                 writeCV( uint16_t cvId, uint8_t mode, uint8_t val );
545        uint8_t                 writeCVByte( uint16_t cvId, uint8_t val );
546        uint8_t                 writeCVBit( uint16_t cvId, uint8_t bitPos, uint8_t val );
547
548        uint8_t                 writeDccPacketMain( uint8_t *buf,  uint8_t len, uint8_t nRepeat );
549        uint8_t                 writeDccPacketProg( uint8_t *buf,  uint8_t len, uint8_t nRepeat );
550
551        void                    printSessionMapConfig( );
552        void                    printSessionMapInfo( );
553
554        SessionMapEntry         *lookupSessionEntry( uint16_t cabId );
555        SessionMapEntry         *getSessionMapEntryPtr( uint8_t sId );
556
557     private:
558
559        uint8_t                 setThrottle( SessionMapEntry *csptr, uint8_t speed, uint8_t direction );
560        uint8_t                 setDccFunctionGroup( SessionMapEntry *csPtr, uint8_t fGroup, uint8_t dccByte );
561
562        SessionMapEntry         *allocateSessionEntry( uint16_t cabId );
563        void                    deallocateSessionEntry( SessionMapEntry *csPtr );
564        void                    refreshSessionEntry( SessionMapEntry *csPtr );
565        void                    initSessionEntry( SessionMapEntry *csPtr );
566        void                    printSessionEntry( SessionMapEntry *csPtr );
567
568     private:
569
570        LcsBaseStationDccTrack    *mainTrack              = nullptr;
571        LcsBaseStationDccTrack    *progTrack              = nullptr;
572
573        uint16_t                  options                = DT_OPT_DEFAULT_SETTING;
574        uint16_t                  flags                  = DT_F_DEFAULT_SETTING;
575        uint32_t                  lastAliveCheckTime      = 0L;
576        uint32_t                  refreshAliveTimeOutVal  = 2000L;  // ??? a constant name ...
577
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 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
```

```
1   //------------------------------------------------------------------------------------------------------------
2   //
3   // LCS Base Station - Serial Command Interface - implementation file
4   //
5   //------------------------------------------------------------------------------------------------------------
6   // The serial command interface is used to directly send commands to the session and DCC track objects. The
7   // command syntax is patterned after the DCC++ command syntax. Available commands that have a DCC++ counter
8   // part are implemented exactly after the DCC++ command specification. The main motivation is to use this
9   // interface for testing and debugging as well as third party tools that also implement the DCC++ command set
10  // to send commands to this base station as well when calling the serial IO interface. For the layout control
11  // system, the approach would rather be to send LCS messages for all tasks.
12  //
13  //------------------------------------------------------------------------------------------------------------
14  //
15  // LCS - Base Station
16  // Copyright (C) 2019 - 2024  Helmut Fieres
17  //
18  // This program is free software: you can redistribute it and/or modify it under the terms of the GNU General
19  // Public License as published by the Free Software Foundation, either version 3 of the License, or (at your
20  // option) any later version.
21  //
22  // This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
23  // implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
24  // for more details.
25  //
26  // You should have received a copy of the GNU General Public License along with this program. If not, see
27  // http://www.gnu.org/licenses
28  //
29  //  GNU General Public License:  http://opensource.org/licenses/GPL-3.0
30  //
31  //------------------------------------------------------------------------------------------------------------
32  #include "LcsBaseStation.h"
33
34  using namespace LCS;
35
36  //------------------------------------------------------------------------------------------------------------
37  // External global variables.
38  //
39  //------------------------------------------------------------------------------------------------------------
40  extern uint16_t debugMask;
41
42  //------------------------------------------------------------------------------------------------------------
43  // The object constructor. Nothing to do here.
44  //
45  //------------------------------------------------------------------------------------------------------------
46  LcsBaseStationCommand::LcsBaseStationCommand( ) { }
47
48  //------------------------------------------------------------------------------------------------------------
49  // The object setup command. We need to remember the other objects we use in handling the commands. For the
50  // serial IO itself nothing to do, it was already done in the LCS runtime setup.
51  //
52  //------------------------------------------------------------------------------------------------------------
53  uint8_t LcsBaseStationCommand::setupSerialCommand(
54
55      LcsBaseStationLocoSession  *locoSessions,
56      LcsBaseStationDccTrack     *mainTrack,
57      LcsBaseStationDccTrack     *progTrack ) {
58
59      this -> locoSessions  = locoSessions;
60      this -> mainTrack     = mainTrack;
61      this -> progTrack     = progTrack;
62
63      return ( ALL_OK );
64  }
65
66  //------------------------------------------------------------------------------------------------------------
67  // "handleSerialCommand" analyzes the command line and invokes the respective command handler. The first
68  // character in a command is the command letter. The command is followed by the arguments. For compatibility
69  // with the DCC++ original command set, each command that is also a DCC++ command is implemented exactly as
70  // the original. This allows external tools, such as the JMRI Decoder Pro configuration tool to be used. The
71  // command handler supports command sequences "<" ... ">" in one line which are processed once the carriage
72  // return is hit.
73  //
74  //------------------------------------------------------------------------------------------------------------
75  void LcsBaseStationCommand::handleSerialCommand( char *s ) {
76
77      int     charIndex      = 0;
78      char    cmdStr[ 256 ]  = { 0 };
79
80      while ( s[ charIndex ] != '\0' ) {
81
82          switch ( s[ charIndex ] ) {
83
84              case '<': {
85
86                  cmdStr[ 0 ] = '\0';
87                  charIndex ++;
88
89              } break;
90
91              case '>': {
92
93                  switch ( cmdStr[ 0 ] ) {
94
95                      case 'O': openSessionCmd( cmdStr + 1 ); break;
96                      case 'K': closeSessionCmd( cmdStr + 1 ); break;
97
98                      case 't': setThrottleCmd( cmdStr + 1 );  break;
```

```
 99                        case 'f': setFunctionGroupCmd( cmdStr + 1 ); break;
100                        case 'v': setFunctionBitCmd( cmdStr + 1 ); break;
101
102                        case 'R': readCVCmd( cmdStr + 1 ); break;
103                        case 'W': writeCVByteCmd( cmdStr + 1 ); break;
104                        case 'B': writeCVBitCmd( cmdStr + 1 ); break;
105                        case 'w': writeCVByteMainCmd( cmdStr + 1 ); break;
106                        case 'b': writeCVBitMainCmd( cmdStr + 1 ); break;
107
108                        case 'M': writeDccPacketMainCmd( cmdStr + 1 ); break;
109                        case 'P': writeDccPacketProgCmd( cmdStr + 1 ); break;
110
111                        case 'C': setTrackOptionCmd( cmdStr + 1 ); break;
112                        case 'Y': printDccLogCommand( cmdStr + 1 ); break;
113
114                        case 'X': emergencyStopCmd( ); break;
115                        case '0': turnPowerOffAllCmd( ); break;
116                        case '1': turnPowerOnAllCmd( ); break;
117                        case '2': turnPowerOnMainCmd( ); break;
118                        case '3': turnPowerOnProgCmd( ); break;
119
120                        case 's': printStatusCmd( cmdStr + 1 ); break;
121                        case 'S': printBaseStationConfigCmd( ); break;
122                        case 'L': printSessionMap( ); break;
123
124                        case 'a': printTrackCurrentCmd( cmdStr + 1 ); break;
125
126                        case '?': printHelpCmd( ); break;
127
128                        case ' ': printf( "\n" ); break;
129
130                        case 'e':
131                        case 'E':
132                        case 'D':
133                        case 'T':
134                        case 'Z':
135                        case 'Q':
136                        case 'F': printf( "<Not implemented>\n" ); break;
137
138                        default: printf( "<Unknown command, use '?' for help>\n" );
139                    }
140
141                    charIndex ++;
142
143                } break;
144
145                default: {
146
147                    if ( strlen( cmdStr ) < sizeof( cmdStr ) ) strncat( cmdStr, &s[ charIndex ], 1 );
148                    charIndex ++;
149                }
150            }
151        }
152 }
153
154 //------------------------------------------------------------------------------------------------------------
155 // "openSessionCmd" handles the session creation command. This command is used to allocate a loco session.
156 // We are passed the cab ID and return a session Id.
157 //
158 //     <O cabId>
159 //
160 //     cabId      -  the requesting cab number, from 1 to MAX_CAB_ID.
161 //
162 //     returns: <O sId>
163 //
164 //------------------------------------------------------------------------------------------------------------
165 void LcsBaseStationCommand::openSessionCmd( char *s ) {
166
167     uint16_t  cabId = NIL_CAB_ID;
168     uint8_t   sId   = 0;
169
170     if ( sscanf( s, "%hu", &cabId ) != 1 ) return;
171
172     int ret = locoSessions -> requestSession( cabId, LSM_NORMAL, &sId );
173
174     printf( "<O %d>", (( ret == ALL_OK ) ? sId : -1 ));
175 }
176
177 //------------------------------------------------------------------------------------------------------------
178 // "closeSessionCmd" handles the session release command. The return code is the CabSession error code. A zero
179 // indicates a successful execution.
180 //
181 //     <K sId>
182 //
183 //     sId       -  the session number.
184 //
185 //     returns: <K status>
186 //
187 //------------------------------------------------------------------------------------------------------------
188 void LcsBaseStationCommand::closeSessionCmd( char *s ) {
189
190     uint8_t sId = NIL_LOCO_SESSION_ID;
191
192     if ( sscanf( s, "%hhu", &sId ) != 1 ) return;
193
194     int ret = locoSessions -> releaseSession( sId );
195
196     printf( "<K %d>", ret );
197 }
```

```
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.
```

```
594   //
595   //      <C option>
596   //
597   //      option   - the option value.
598   //
599   //          1 -> set main track Cutout mode on.
600   //          2 -> set main track Cutout mode off.
601   //          3 -> set main track Railcom mode on.
602   //          4 -> set main track Railcom mode off.
603   //
604   //          10 -> set service track into operations mode.
605   //          11 -> set service track into service mode.
606   //
607   //      returns: NONE
608   //
609   //-----------------------------------------------------------------------------------------------------------
610   void LcsBaseStationCommand::setTrackOptionCmd( char *s ) {
611
612       uint8_t option = 0;
613
614       if ( sscanf( s, "%hhu", &option ) == 1 ) {
615
616           switch ( option ) {
617
618               case 1: mainTrack -> cutoutOn( );   break;
619               case 2: mainTrack -> cutoutOff( );  break;
620               case 3: mainTrack -> railComOn( );  break;
621               case 4: mainTrack -> railComOff( ); break;
622
623               case 10: progTrack -> serviceModeOff( );  break;
624               case 11: progTrack -> serviceModeOn( ); break;
625           }
626       }
627   }
628
629   //-----------------------------------------------------------------------------------------------------------
630   // "printStatusCmd" list information about the base station. Using just a "s" for a summary status is always
631   // a good idea to do this just as a first basic test if things are running at all. The level is a positive
632   // integer that specifies the information items to be listed.
633   //
634   //      <s [ opt ]> - the kind of status to display.
635   //
636   //      returns:  series of status information that can be read by an interface to determine status of the base
637   //                station and important settings
638   //
639   //-----------------------------------------------------------------------------------------------------------
640   void LcsBaseStationCommand::printStatusCmd( char *s ) {
641
642       uint8_t opt = 0;
643
644       if ( sscanf( s, "%hhu", &opt ) > 0 ) {
645
646           switch ( opt ) {
647
648               case 0: printVersionInfo( );      break;
649               case 1: printConfiguration( );    break;
650               case 2: printSessionMap( );       break;
651               case 3: printTrackStatusMain( );  break;
652               case 4: printTrackStatusProg( );  break;
653
654               case 9: {
655
656                   printConfiguration( );
657                   printSessionMap( );
658                   printTrackStatusMain( );
659                   printTrackStatusProg( );
660
661               } break;
662
663               default: printVersionInfo( );
664           }
665       } else printVersionInfo( );
666   }
667
668   //-----------------------------------------------------------------------------------------------------------
669   // "printBaseStationConfigCmd" list information about the base in a DCC++ compatible way.
670   //
671   //      <S> - the basestation configuration.
672   //
673   //      returns:  series of status information that can be read by an interface to determine status of the base
674   //                station and important settings
675   //
676   //-----------------------------------------------------------------------------------------------------------
677   void LcsBaseStationCommand::printBaseStationConfigCmd( ) {
678
679       printConfiguration( );
680   }
681
682   //-----------------------------------------------------------------------------------------------------------
683   // "printConfiguration" lists out the key hardware and software settings. Also very useful as the first
684   // trouble shooting task.
685   //
686   //-----------------------------------------------------------------------------------------------------------
687   void LcsBaseStationCommand::printConfiguration( ) {
688
689       printVersionInfo( );
690       locoSessions -> printSessionMapConfig( );
691       mainTrack -> printDccTrackConfig( );
692       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
```

```
495  // or even longer for the cutout itself and give us some more room.
496  //
497  // ??? we could use timerVal, but this is in microseconds, not ticks. Convert one day...
498  //-------------------------------------------------------------------------------------------------------
499  void timerCallback( uint32_t timerVal ) {
500
501      uint8_t followUpMain = DCC_SIG_FOLLOW_UP_NONE;
502      uint8_t followUpProg = DCC_SIG_FOLLOW_UP_NONE;
503
504      timeLeftMainTrack -= timeToInterrupt;
505      timeLeftProgTrack -= timeToInterrupt;
506
507      if ( timeLeftMainTrack == 0 ) mainTrack -> runDccSignalStateMachine( &timeLeftMainTrack, &followUpMain );
508      if ( timeLeftProgTrack == 0 ) progTrack -> runDccSignalStateMachine( &timeLeftProgTrack, &followUpProg );
509
510      // take out after test ...
511      // timeToInterrupt = min( timeLeftMainTrack, timeLeftProgTrack );
512
513      timeToInterrupt = (( timeLeftMainTrack < timeLeftProgTrack ) ? timeLeftMainTrack : timeLeftProgTrack );
514
515      CDC::setRepeatingTimerLimit( timeToInterrupt * TICK_IN_MICROSECONDS );
516
517      if (( followUpMain != DCC_SIG_FOLLOW_UP_NONE ) && ( followUpMain != DCC_SIG_FOLLOW_UP_MEASURE_CURRENT )) {
518
519          if      ( followUpMain == DCC_SIG_FOLLOW_UP_GET_BIT )          mainTrack -> getNextBit( );
520          else if ( followUpMain == DCC_SIG_FOLLOW_UP_GET_PACKET )       mainTrack -> getNextPacket( );
521          else if ( followUpMain == DCC_SIG_FOLLOW_UP_START_RAILCOM_IO ) mainTrack -> startRailComIO( );
522          else if ( followUpMain == DCC_SIG_FOLLOW_UP_STOP_RAILCOM_IO )  mainTrack -> stopRailComIO( );
523          else if ( followUpMain == DCC_SIG_FOLLOW_UP_RAILCOM_MSG )      mainTrack -> handleRailComMsg( );
524      }
525
526      if (( followUpProg != DCC_SIG_FOLLOW_UP_NONE ) && ( followUpProg != DCC_SIG_FOLLOW_UP_MEASURE_CURRENT )) {
527
528          if      ( followUpProg == DCC_SIG_FOLLOW_UP_GET_BIT )    progTrack -> getNextBit( );
529          else if ( followUpProg == DCC_SIG_FOLLOW_UP_GET_PACKET ) progTrack -> getNextPacket( );
530      }
531
532      if      ( followUpMain == DCC_SIG_FOLLOW_UP_MEASURE_CURRENT ) mainTrack -> powerMeasurement( );
533      else if ( followUpProg == DCC_SIG_FOLLOW_UP_MEASURE_CURRENT ) progTrack -> powerMeasurement( );
534
535  } // timerCallback
536
537  //-------------------------------------------------------------------------------------------------------
538  // When all DCC track objects are initialized, the last thing to do before operation is to  start the timer
539  // heartbeat. We start b firing up the timer with a first short delay, so when it expires the timer routine
540  // will be called. The current time tick of zero and no ticks left, so the state machine for the signals
541  // will run.
542  //
543  //-------------------------------------------------------------------------------------------------------
544  void initDccTrackProcessing( ) {
545
546      timeToInterrupt   = 0;
547      timeLeftMainTrack = 0;
548      timeLeftProgTrack = 0;
549
550      CDC::startRepeatingTimer( TICK_IN_MICROSECONDS );
551  }
552
553  //-------------------------------------------------------------------------------------------------------
554  // DCC log functions for printing the DCC log buffer. The fist byte of each log entry has encoded the log
555  // entry type and the entry length. Depending on the log entry type, data is displayed as just the header,
556  // a numeric 16-bit value, a numeric 32-bit vale or as an array of data bytes. We return the length of the
557  // DCC log entry.
558  //
559  //-------------------------------------------------------------------------------------------------------
560  void printLogTimeStamp( uint16_t index ) {
561
562      uint32_t ts = logBuf[ index ];
563      ts = ( ts << 8 ) | logBuf[ index + 1 ];
564      ts = ( ts << 8 ) | logBuf[ index + 2 ];
565      ts = ( ts << 8 ) | logBuf[ index + 3 ];
566      printf( "0x%x", ts );
567  }
568
569  void printLogVal( uint16_t index ) {
570
571      uint16_t val = logBuf[ index ] << 8 | logBuf[ index + 1 ];
572      printf( "0x%04x", val );
573  }
574
575  void printLogData( uint16_t index, uint8_t len ) {
576
577      for ( int i = 0; i < len; i++ ) printf( "0x%02x ", logBuf[ index + i ] );
578  }
579
580  uint8_t printLogEntry( uint16_t index ) {
581
582      if ( index < LOG_BUF_SIZE ) {
583
584          uint8_t logEntryId  = logBuf[ index ] >> 4;
585          uint8_t logEntryLen = logBuf[ index ] & 0x0F;
586
587          switch ( logEntryId ) {
588
589              case LOG_NIL:     printf( "NIL       " ); break;
590              case LOG_BEGIN:   printf( "BEGIN     " ); break;
591              case LOG_END:     printf( "END       " ); break;
592              case LOG_TSTAMP:  printf( "TSTAMP    " ); break;
593              case LOG_DCC_IDL: printf( "DCC_IDLE  " ); break;
```

```
594              case LOG_DCC_RST:  printf( "DCC_RESET  " ); break;
595              case LOG_DCC_PKT:  printf( "DCC_PKT    " ); break;
596              case LOG_DCC_RCM:  printf( "DCC_RCOM   " ); break;
597              case LOG_VAL:      printf( "VAL        " ); break;
598              default:           printf( "INVALID ( 0x%02 )", logBuf[ index ] >> 4 );
599          }
600
601          if      ( logEntryId == LOG_TSTAMP  )  printLogTimeStamp( index + 1 );
602          else if ( logEntryId == LOG_VAL     )  printLogVal( index + 1 );
603          else                                   printLogData( index + 1, logEntryLen );
604
605          return ( logEntryLen + 1 );
606      }
607      else return ( 0 );
608  }
609
610  //----------------------------------------------------------------------------------------------------
611  // There are a couple of routines to write the log data. For convenience, some of the log entry types are
612  // available as a direct call. The order of data entry for numeric types is big endian, i.e. most significant
613  // byte first.
614  //
615  //----------------------------------------------------------------------------------------------------
616  void writeLogData( uint8_t id, uint8_t *buf, uint8_t len ) {
617
618      if ( logActive ) {
619
620          len = len % 16;
621          if ( logBufIndex + len + 1 < LOG_BUF_SIZE ) {
622
623              logBuf[ logBufIndex ++ ] = ( id << 4 ) | len;
624              for ( uint8_t i = 0; i < len; i++ ) logBuf[ logBufIndex ++ ] = buf[ i ];
625          }
626      }
627  }
628
629  void writeLogId( uint8_t id ) {
630
631      if ( logActive ) logBuf[ logBufIndex ++ ] = ( id << 4 ) | 1;
632  }
633
634  void writeLogTs( ) {
635
636      if ( logActive ) {
637
638          uint32_t ts = CDC::getMicros( );
639          logBuf[ logBufIndex ++ ] = ( LOG_TSTAMP << 4 ) | 4;
640          logBuf[ logBufIndex ++ ] = ( ts >> 24 ) & 0xFF;
641          logBuf[ logBufIndex ++ ] = ( ts >> 16 ) & 0xFF;
642          logBuf[ logBufIndex ++ ] = ( ts >> 8  ) & 0xFF;
643          logBuf[ logBufIndex ++ ] = ( ts >> 0  ) & 0xFF;
644      }
645  }
646
647  void writeLogVal( uint8_t valId, uint16_t val ) {
648
649      if ( logActive ) {
650
651          logBuf[ logBufIndex ++ ] = ( LOG_VAL << 4 ) | 3;
652          logBuf[ logBufIndex ++ ] = valId;
653          logBuf[ logBufIndex ++ ] = val >> 8;
654          logBuf[ logBufIndex ++ ] = val & 0xFF;
655      }
656  }
657
658  //----------------------------------------------------------------------------------------------------
659  // The log management routines. A typical transaction to log would start the logging process and then end
660  // it after the operation to analyze/debug. The "enableLog" call should be used to enable the logging
661  // process all together, the other calls will only work when the log is enabled. With this call the
662  // recording process could be controlled from a command line setting or so.
663  //
664  //----------------------------------------------------------------------------------------------------
665  void enableLog( bool arg ) {
666
667      logEnabled = arg;
668      logActive  = false;
669  }
670
671  void beginLog( ) {
672
673      if ( logEnabled ) {
674
675          logActive   = true;
676          logBufIndex = 0;
677          writeLogId( LOG_BEGIN );
678          writeLogTs( );
679      }
680  }
681
682  void endLog( ) {
683
684      if ( logActive ) {
685
686          writeLogTs( );
687          writeLogId( LOG_END );
688          logActive = false;
689      }
690  }
691
692  //----------------------------------------------------------------------------------------------------
```

```
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 listrupt 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 //
```

```
891   // DCC_SIG_CUTOUT_1: this stage sets the signal to CUTOUT for cutout period ticks. Also, if the RailCom
892   // is enabled, there is a follow up request to start the serial IO read function. The signal state advances
893   // to signal state DCC_SIG_CUTOUT_2.
894   //
895   // DCC_SIG_CUTOUT_2: this stage sets the signal to LOW for the cutout end tick. The signal state advances
896   // to signal state DCC_SIG_CUTOUT_3.
897   //
898   // DC_SIG_CUTOUT_3: the DC_SIG_CUTOUT_3 and DC_SIG_END_CUTOUT states represent the first DCC "One" after
899   // the cutout. The DCC signal is set to HIGH and the next period is two ticks. The follow-up request is to
900   // disable the UART receiver. The signal state advances to DC_SIG_CUTOUT_END.
901   //
902   // DC_SIG_CUTOUT_END: The DC_SIG_END_CUTOUT state is the second half of the DCC one. The signal is set
903   // to low and the next period to two ticks. If RailCom is enabled, this is the state where a follow up
904   // to handle the RailCom data takes place. The next state is then DCC_SIG_START_BIT to handle the next
905   // packet, starting with the preamble of DCC ones.
906   //
907   // DCC_SIG_START_BIT: this stage is the start of the DCC packet bits, which are preamble, the data bytes
908   // with separators and postamble. If the cutout option is off, this is also the start for the DCC packet.
909   // The signal is set HIGH, the tick count is two and we need a follow up to get the current bit, which
910   // determines the length of the signal for the bit we just started. The next stage is signal state
911   // DCC_SIG_TEST_BIT.
912   //
913   // DCC_SIG_TEST_BIT: coming from signal state DCC_SIG_START_BIT, we need to see if the current bit is a ONE
914   // or ZERO bit. If a ONE bit, the signal needs to become LOW, the next period is 2 ticks and the next state
915   // is signal state DCC_SIG_START_BIT. If it is the last ONE bit of the postamble, the next packet and
916   // signal state needs to be determined. For a CUTOUT enabled track this is state DCC_SIG_START_CUTOUT, else
917   // DCC_SIG_START_BIT. If a ZERO bit, the signal is kept HIGH for another two ticks and the state is
918   // DCC_SIG_ZERO_SECOND_HALF.
919   //
920   // The ZERO bit case is also a good place to do a current measurement. We are already two ticks into the
921   // signal polarity change and there should be no spike from the signal level transition. However, we do
922   // not want to measure all zero bits since this would mean several hundreds to few thousands per second.
923   // Each data byte starts with a DCC ZERO bit. We will just sample the current there and end up with a few
924   // hundred samples per second, which is less of a burden but still often enough for overload detection
925   // and so on.
926   //
927   // DCC_SIG_ZERO_SECOND_HALF: coming from signal state DCC_SIG_TEST_BIT, we need to transmit the second half
928   // of the ZERO bit. The signal is set to LOW for four ticks and set the next stage is signal state to
929   // DCC_SIG_START_BIT.
930   //
931   // Note: for a 16Mhz Atmega the implementation for the cutout support is a close call. If the timer value
932   // setting takes place after the internal timer counter HW has passed this value, you wrap around and the
933   // interrupt happens the next time the timer value matches, which is about 4 milliseconds later! If you see
934   // such a gap in the DCC signal, this is perhaps the issue. When using the railcom/cutout option it is
935   // recommended to set the processor frequency to 20Mhz, which you can do in your own design, but not on
936   // an Arduino board.
937   //
938   //----------------------------------------------------------------------------------------------------------
939   void LcsBaseStationDccTrack::runDccSignalStateMachine(
940
941       volatile uint8_t   *timeToInterrupt,
942       uint8_t            *followUpAction
943
944       ) {
945
946       switch ( signalState ) {
947
948           case DCC_SIG_CUTOUT_START: {
949
950               CDC::writeDioPair( dccSigPin1, true, dccSigPin2, false );
951               *timeToInterrupt  = TICKS_29_MICROS;
952               *followUpAction   = DCC_SIG_FOLLOW_UP_NONE;
953               signalState       = DCC_SIG_CUTOUT_1;
954
955           } break;
956
957           case DCC_SIG_CUTOUT_1: {
958
959               CDC::writeDioPair( dccSigPin1, false, dccSigPin2, false );
960               *timeToInterrupt  = TICKS_CUTOUT_MICROS;
961               *followUpAction   = (( flags & DT_F_RAILCOM_MODE_ON ) ?
962                                   DCC_SIG_FOLLOW_UP_START_RAILCOM_IO : DCC_SIG_FOLLOW_UP_NONE );
963               signalState       = DCC_SIG_CUTOUT_2;
964
965           } break;
966
967           case DCC_SIG_CUTOUT_2: {
968
969               CDC::writeDioPair( dccSigPin1, false, dccSigPin2, true );
970               *timeToInterrupt  = TICKS_29_MICROS;
971               *followUpAction   = DCC_SIG_FOLLOW_UP_NONE;
972               signalState       = DCC_SIG_CUTOUT_3;
973
974           } break;
975
976           case DCC_SIG_CUTOUT_3: {
977
978               CDC::writeDioPair( dccSigPin1, true, dccSigPin2, false );
979               *timeToInterrupt  = TICKS_58_MICROS;
980               signalState       = DCC_SIG_CUTOUT_END;
981
982               if ( flags & DT_F_RAILCOM_MODE_ON ) {
983
984                   flags            |= DT_F_RAILCOM_MSG_PENDING;
985                   *followUpAction = DCC_SIG_FOLLOW_UP_STOP_RAILCOM_IO;
986               }
987               else *followUpAction = DCC_SIG_FOLLOW_UP_NONE;
988
989           } 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 //-------------------------------------------------------------------------------------------------------------
1067 void 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 }
```

```cpp
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  }
```

```
1   //------------------------------------------------------------------------------------------------------------
2   //
3   // LCS Base Station - Loco Session Management - implementation file
4   //
5   //------------------------------------------------------------------------------------------------------------
6   // The locomotive session object is the besides the two DCC tracks the other main component of a base station.
7   // Each engine to run needs a session on this session object. Typically, the handheld will "open" a session.
8   // The session identifier is then the handle to the locomotive.
9   //
10  //
11  //
12  //------------------------------------------------------------------------------------------------------------
13  //
14  // LCS - Base Station
15  // Copyright (C) 2019 - 2024  Helmut Fieres
16  //
17  // This program is free software: you can redistribute it and/or modify it under the terms of the GNU General
18  // Public License as published by the Free Software Foundation, either version 3 of the License, or (at your
19  // option) any later version.
20  //
21  // This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
22  // implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
23  // for more details.
24  //
25  // You should have received a copy of the GNU General Public License along with this program. If not, see
26  // http://www.gnu.org/licenses
27  //
28  //  GNU General Public License:  http://opensource.org/licenses/GPL-3.0
29  //
30  //------------------------------------------------------------------------------------------------------------
31  #include "LcsBaseStation.h"
32  #include <malloc.h>
33
34  using namespace LCS;
35
36  //------------------------------------------------------------------------------------------------------------
37  // External global variables.
38  //
39  //------------------------------------------------------------------------------------------------------------
40  extern uint16_t debugMask;
41
42  //------------------------------------------------------------------------------------------------------------
43  // Loco Session implementation file - local declarations.
44  //
45  //------------------------------------------------------------------------------------------------------------
46  namespace {
47
48  //------------------------------------------------------------------------------------------------------------
49  // DCC packet definitions. A DCC packet payload is at most 10 bytes long, excluding the checksum byte. This
50  // is true for XPOM support, otherwise it is according to NMRA up to 6 bytes.
51  //
52  //------------------------------------------------------------------------------------------------------------
53  const uint8_t   MIN_DCC_PACKET_SIZE        = 2;
54  const uint8_t   MAX_DCC_PACKET_SIZE        = 16;
55  const uint8_t   MIN_DCC_PACKET_REPEATS     = 0;
56  const uint8_t   MAX_DCC_PACKET_REPEATS     = 8;
57
58  //------------------------------------------------------------------------------------------------------------
59  // Utility routines.
60  //
61  //------------------------------------------------------------------------------------------------------------
62  bool isInRangeU( uint8_t val, uint8_t lower, uint8_t upper ) {
63
64      return (( val >= lower ) && ( val <= upper ));
65  }
66
67  bool isInRangeU( uint16_t val, uint16_t lower, uint16_t upper ) {
68
69      return (( val >= lower ) && ( val <= upper ));
70  }
71
72  bool isInRangeU( uint32_t val, uint32_t lower, uint32_t upper ) {
73
74      return (( val >= lower ) && ( val <= upper ));
75  }
76
77  bool validCabId( uint16_t cabId ) {
78
79      return ( isInRangeU( cabId, MIN_CAB_ID, MAX_CAB_ID ));
80  }
81
82  bool validCvId( uint16_t cvId ) {
83
84      return ( isInRangeU( cvId, MIN_DCC_CV_ID, MAX_DCC_CV_ID ));
85  }
86
87  bool validFunctionId( uint8_t fId ) {
88
89      return ( isInRangeU( fId, MIN_DCC_FUNC_ID, MAX_DCC_FUNC_ID ));
90  }
91
92  bool validFunctionGroupId( uint8_t fGroup ) {
93
94      return ( isInRangeU( fGroup, MIN_DCC_FUNC_GROUP_ID , MAX_DCC_FUNC_GROUP_ID ));
95  }
96
97  bool validDccPacketlen( uint8_t len ) {
98
```

```
 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
```

```
594            case 3: pBuf[pLen++] = 0xDE; pBuf[pLen++] = smePtr -> functions[ 3 ]; break;
595            case 4: pBuf[pLen++] = 0xDF; pBuf[pLen++] = smePtr -> functions[ 4 ]; break;
596            case 5: pBuf[pLen++] = 0xD8; pBuf[pLen++] = smePtr -> functions[ 5 ]; break;
597            case 6: pBuf[pLen++] = 0xD9; pBuf[pLen++] = smePtr -> functions[ 6 ]; break;
598            case 7: pBuf[pLen++] = 0xDA; pBuf[pLen++] = smePtr -> functions[ 7 ]; break;
599            case 8: pBuf[pLen++] = 0xDB; pBuf[pLen++] = smePtr -> functions[ 8 ]; break;
600            case 9: pBuf[pLen++] = 0xDC; pBuf[pLen++] = smePtr -> functions[ 9 ]; break;
601        }
602
603        mainTrack -> loadPacket( pBuf, pLen, 4 );
604        return ( ALL_OK );
605 }
606
607 //-----------------------------------------------------------------------------------------------------------
608 // "writeCVMain" writes a CV value to the decoder on the main track. CV numbers range from 1 to 1024, but are
609 // encoded from 0 to 1023. The DCC standard defines various modes for retrieving CV values. This function
610 // implements CV write mode mode 0 and 1, by calling the respective method. The other modes are not supported.
611 // For bit mode access, the bit position and bit value are encoded in the "val" parameter with bit 3 containing
612 // the data and bit 0 ..2 the bit offset.
613 //
614 //     0 Direct Byte
615 //     1 Direct Bit
616 //     2 Page Mode
617 //     3 Register Mode
618 //     4 Address Only Mode
619 //
620 //
621 // Note on the MAIN track, there is no way for the decoder to answer via a raise in power consumption. The
622 // command shown here is just sent. If however RailCom is available, the decoder can answer with the CV
623 // value in a following cutout. This is currently not implemented.
624 //-----------------------------------------------------------------------------------------------------------
625 uint8_t LcsBaseStationLocoSession::writeCVMain( uint8_t sId, uint16_t cvId, uint8_t mode, uint8_t val ) {
626
627        if         ( mode == 0 )  return ( writeCVByteMain( sId, cvId, val ));
628        else if    ( mode == 1 )  return ( writeCVBitMain( sId, cvId, ( val & 0x07 ), (( val & 0x08 ) >> 3 )));
629        else                      return ( ERR_INVALID_CV_MODE );
630 }
631
632 //-----------------------------------------------------------------------------------------------------------
633 // "writeCVByteMain" writes a byte to the CV while the loco is on the main track. The CV numbers range from
634 // 1 to 1024, but are encoded from 0 to 1023. This function implements CV write mode mode 0, which is write
635 // a byte at a time. There is no way to validate our operation, only writes are possible. The packet is sent
636 // four times.
637 //
638 //-----------------------------------------------------------------------------------------------------------
639 uint8_t LcsBaseStationLocoSession::writeCVByteMain( uint8_t sId, uint16_t cvId, uint8_t val ) {
640
641        uint8_t    pBuf[ MAX_DCC_PACKET_SIZE ];
642        uint8_t    pLen = 0;
643
644        SessionMapEntry *smePtr = getSessionMapEntryPtr( sId );
645        if ( smePtr == nullptr ) return ( ERR_INVALID_SESSION_ID );
646
647        if ( ! validCvId( cvId )) return ( ERR_INVALID_CV_ID );
648        cvId--;
649
650        if ( smePtr -> cabId > 127 ) pBuf[pLen++] = highByte( smePtr -> cabId ) | 0xC0;
651        pBuf[pLen++] = lowByte( smePtr -> cabId );
652        pBuf[pLen++] = 0xEC + ( highByte( cvId ) & 0x03 );
653        pBuf[pLen++] = lowByte( cvId );
654        pBuf[pLen++] = val;
655
656        mainTrack -> loadPacket( pBuf, pLen, 4 );
657        return ( ALL_OK );
658 }
659
660 //-----------------------------------------------------------------------------------------------------------
661 // "writeCVBitMain" writes a bit to the CV while the loco is on the main track. The CV numbers range from 1
662 // to 1024, but are encoded from 0 to 1023. his function implements CV write mode mode 1, which is write a
663 // bit at a time. On input the "val" parameter encodes the bit position in bits 0 - 2 and the bit value in
664 // bit 3.  There is no way to validate our operation, only CV writes are possible. The packet is sent four
665 // times.
666 //
667 //-----------------------------------------------------------------------------------------------------------
668 uint8_t LcsBaseStationLocoSession::writeCVBitMain( uint8_t sId, uint16_t cvId, uint8_t bitPos, uint8_t val ) {
669
670        SessionMapEntry *smePtr = getSessionMapEntryPtr( sId );
671        if ( smePtr == nullptr ) return ( ERR_INVALID_SESSION_ID );
672
673        if ( ! validCvId( cvId )) return ( ERR_INVALID_CV_ID );
674        cvId--;
675
676        uint8_t pBuf[ MAX_DCC_PACKET_SIZE ];
677        uint8_t pLen = 0;
678
679        if ( smePtr -> cabId > 127 ) pBuf[pLen++] = highByte( smePtr -> cabId ) | 0xC0;
680        pBuf[pLen++] = lowByte( smePtr -> cabId );
681        pBuf[pLen++] = 0xE8 + ( highByte( cvId ) & 0x03 );
682        pBuf[pLen++] = lowByte( cvId );
683        pBuf[pLen++] = 0xF0 + (( val % 2 ) << 3 ) + ( bitPos % 8 );
684
685        mainTrack -> loadPacket( pBuf, pLen, 4 );
686        return ( ALL_OK );
687 }
688
689 //-----------------------------------------------------------------------------------------------------------
690 // "readCV" retrieves a CV value from the decoder in service mode. CV numbers range from 1 to 1024, but are
691 // encoded from 0 to 1023. This command is only available in service mode, i.e. on a programming track. The
692 // 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   }
```

12.2   CDC Lib

```
1   //------------------------------------------------------------------------------------------------------------
2   //
3   // LCS - Controller Dependent Code - Include file
4   //
5   //------------------------------------------------------------------------------------------------------------
6   // The controller dependent code layer concentrates all processor dependent code into one library. The idea
7   // is twofold. First, there needs to be a way to isolate the controller specific hardware from the LCS runtime
8   // Library as well as the extension module firmware. The Raspberry PI Pico offers a C++ SDK with a set of
9   // libraries to invoke the desired function rather than access to registers.The Pico also offers a great
10  // flexibility of pin assignment for the hardware IO functions. Second, within the hardware IO boundaries of
11  // the controller family the individual hardware pin assignment used may vary from board to board design.
12  // Nevertheless, the Extension Connector layout and basic functions available should be the same for all
13  // controllers used. For the upper software layers, the CDC library offers a structured way to describe
14  // the possible pins assignments.
15  //
16  // Note that this layer is not a generic HW abstraction. The layer is very specific to the LCS controller
17  // boards described in the book. Nevertheless, some pins can vary, depending on the board version. Currently,
18  // only the Raspberry PI Pico Board is supported.
19  //
20  //------------------------------------------------------------------------------------------------------------
21  //
22  // LCS - Controller Dependent Code - Include file
23  // Copyright (C) 2022 - 2024  Helmut Fieres
24  //
25  // This program is free software: you can redistribute it and/or modify it under the terms of the GNU General
26  // Public License as published by the Free Software Foundation, either version 3 of the License, or (at your
27  // option) any later version.
28  //
29  // This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
30  // implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
31  // for more details.
32  //
33  // You should have received a copy of the GNU General Public License along with this program. If not, see
34  // http://www.gnu.org/licenses
35  //
36  //  GNU General Public License:  http://opensource.org/licenses/GPL-3.0
37  //
38  //------------------------------------------------------------------------------------------------------------
39  #ifndef LcsCdcLib_h
40  #define LcsCdcLib_h
41
42  //------------------------------------------------------------------------------------------------------------
43  // Include files.
44  //
45  //------------------------------------------------------------------------------------------------------------
46  #include <stdio.h>
47  #include <stdint.h>
48  #include <cstring>
49
50  //------------------------------------------------------------------------------------------------------------
51  // All definitions and functions are in the CDC name space.
52  //
53  //------------------------------------------------------------------------------------------------------------
54  namespace CDC {
55
56  //------------------------------------------------------------------------------------------------------------
57  // Error status codes. The errors are used when setting up the Hal library. During operation, all routines
58  // validate the input for correctness. If they are not correct, the call is simply not performed and an
59  // error is returned.
60  //
61  // ??? clean up a little ... what is really needed ?
62  //------------------------------------------------------------------------------------------------------------
63  enum CdcStatus : uint8_t {
64
65      NO_ERR              = 0,
66      INIT_PENDING        = 1,
67      NOT_SUPPORTED       = 2,
68      NOT_IMPLEMENTED     = 3,
69
70      MEM_SIZE_ERR        = 10,
71
72      READY_LED_PIN_ERR   = 12,
73      ACTIVE_LED_PIN_ERR  = 13,
74      BUTTON_PIN_ERR      = 14,
75      PFAIL_PIN_ERR       = 15,
76      EXT_INT_PIN_ERR     = 16,
77      DIO_PIN_ERR         = 17,
78      ADC_PIN_ERR         = 18,
79      PWM_PIN_ERR         = 19,
80
81      UART_PORT_ERR       = 20,
82      UART_CONFIG_ERR     = 21,
83      UART_WRITE_ERR      = 22,
84      UAT_READ_ERR        = 23,
85
86      SPI_PORT_ERR        = 25,
87      SPI_CONFIG_ERR      = 26,
88      SPI_WRITE_ERR       = 27,
89      SPI_READ_ERR        = 28,
90
91      I2C_PORT_ERR        = 30,
92      I2C_CONFIG_ERR      = 31,
93      I2C_WRITE_ERR       = 32,
94      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   //-------------------------------------------------------------------------------------------------------------
133   // GPIO interrupts are detected as level change or edge changes.
134   //
135   //-------------------------------------------------------------------------------------------------------------
136   enum intEventTyp : uint8_t {
137
138       EVT_NONE    = 0,
139       EVT_LOW     = 1,
140       EVT_HIGH    = 2,
141       EVT_FALL    = 3,
142       EVT_RISE    = 4,
143       EVT_CHANGE  = 5
144   };
145
146   //-------------------------------------------------------------------------------------------------------------
147   // The UART modes. There are two implementations. The PICO offers two hardware UARTS. We use them with 8
148   // bits with a parity bit. The second type UART is a software implementation based on the PICO PIO blocks.
149   //
150   //-------------------------------------------------------------------------------------------------------------
151   enum UartMode : uint8_t {
152
153       UART_MODE_UNDEFINED = 0,
154       UART_MODE_8N1       = 1,
155       UART_MODE_8N1_PIO   = 2
156   };
157
158   //-------------------------------------------------------------------------------------------------------------
159   // Callback functions signatures.
160   //
161   //-------------------------------------------------------------------------------------------------------------
162   extern "C" {
163
164       typedef void ( *TimerCallback ) ( uint32_t timerVal );
165       typedef void ( *GpioCallback ) ( uint8_t pin, uint8_t event );
166   }
167
168   //-------------------------------------------------------------------------------------------------------------
169   // CDC features a data structure that records all HW specific pins and flags. The values are set by the
170   // initialization code in a project and are validated. All modules in a project will then just use the
171   // data structure fields using the data for calls to the Hal layer. For example, an application that
172   // uses DIO_PIN_0 and DIO_PIN_1 will set the HW pin numbers of the controller / board combination used
173   // in a config data structure "cfg". A call to write a value to the DIO pin, will then just use
174   // "cfg.DIO_PIN_1" as argument in the "writeDio" call. The "writeDio" call itself will not check the
175   // value of the configured DIO pin, all it will do is to ensure that it is not UNDEFINED. Note that the
176   // structure has more pins defined that a potential controller may have. If so, these fields are set to
177   // UNDEFINED. The structure is the superset of all possible HW items to configure.
178   //
179   // In a later runtime version, we may put this structure as constant data into the non-volatile chip on
180   // the board. It will then just be read from there.
181   //
182   //-------------------------------------------------------------------------------------------------------------
183   struct CdcConfigDesc {
184
185       uint8_t   CFG_STATUS;
186
187       uint8_t   PFAIL_PIN;
188       uint8_t   EXT_INT_PIN;
189       uint8_t   READY_LED_PIN;
190       uint8_t   ACTIVE_LED_PIN;
191
192       uint8_t   DIO_PIN_0;
193       uint8_t   DIO_PIN_1;
```

```
194        uint8_t    DIO_PIN_2;
195        uint8_t    DIO_PIN_3;
196        uint8_t    DIO_PIN_4;
197        uint8_t    DIO_PIN_5;
198        uint8_t    DIO_PIN_6;
199        uint8_t    DIO_PIN_7;
200        uint8_t    DIO_PIN_8;
201        uint8_t    DIO_PIN_9;
202        uint8_t    DIO_PIN_10;
203        uint8_t    DIO_PIN_11;
204        uint8_t    DIO_PIN_12;
205        uint8_t    DIO_PIN_13;
206        uint8_t    DIO_PIN_14;
207        uint8_t    DIO_PIN_15;
208
209        uint8_t    ADC_PIN_0;
210        uint8_t    ADC_PIN_1;
211        uint8_t    ADC_PIN_2;
212        uint8_t    ADC_PIN_3;
213
214        uint8_t    PWM_PIN_0;
215        uint8_t    PWM_PIN_1;
216        uint8_t    PWM_PIN_2;
217        uint8_t    PWM_PIN_3;
218
219        uint8_t    UART_RX_PIN_0;
220        uint8_t    UART_TX_PIN_0;
221
222        uint8_t    UART_RX_PIN_1;
223        uint8_t    UART_TX_PIN_1;
224
225        uint8_t    UART_RX_PIN_2;
226        uint8_t    UART_TX_PIN_2;
227
228        uint8_t    UART_RX_PIN_3;
229        uint8_t    UART_TX_PIN_3;
230
231        uint8_t    SPI_MOSI_PIN_0;
232        uint8_t    SPI_MISO_PIN_0;
233        uint8_t    SPI_SCLK_PIN_0;
234
235        uint8_t    SPI_MOSI_PIN_1;
236        uint8_t    SPI_MISO_PIN_1;
237        uint8_t    SPI_SCLK_PIN_1;
238
239        uint8_t    NVM_I2C_SCL_PIN;
240        uint8_t    NVM_I2C_SDA_PIN;
241        uint8_t    NVM_I2C_ADR_ROOT;
242
243        uint8_t    EXT_I2C_SCL_PIN;
244        uint8_t    EXT_I2C_SDA_PIN;
245        uint8_t    EXT_I2C_ADR_ROOT;
246
247        uint32_t   NODE_NVM_SIZE;
248        uint32_t   EXT_NVM_SIZE;
249
250        uint8_t    CAN_BUS_CTRL_MODE;
251        uint8_t    CAN_BUS_RX_PIN;
252        uint8_t    CAN_BUS_TX_PIN;
253        uint32_t   CAN_BUS_DEF_ID;
254  };
255
256  //------------------------------------------------------------------------------------------------------------
257  // The routines that make up the hardware abstraction layer. The routines expect hardware pin numbers.
258  // To recap, the CDC layer offers a set of reserved resource names, such as "DIO_PIN_0", which describes
259  // the resource containing the hardware pin and some flags. The configuration routines in this layer will use
260  // these pins and other data stored to configure the hardware. Under the defined resource name name all
261  // upper layers refer to the hardware using the to the configured IO capabilities.
262  //
263  // Complex resources, such as the UART or SPI interface, have more than one HW pin they will use. In this
264  // case one of the HW pins, see the function documentation, will serve as the handle to the resource.
265  //
266  //------------------------------------------------------------------------------------------------------------
267
268  //------------------------------------------------------------------------------------------------------------
269  // The console IO functions. We will provide a serial IO via the USB connector of the PICO. The files
270  // need to be linked with the "tinyUSB" library and the cmake file needs to set the option. Then we can
271  // use scanf and printf and so on. In addition, we need  function  that just attempts to read a character
272  // and returns immediately when there is none.
273  //
274  //------------------------------------------------------------------------------------------------------------
275  uint8_t          configureConsoleIO( );
276  bool             isConsoleConnected( );
277  char             getConsoleChar( uint32_t timeoutVal = 0 );
278
279  //------------------------------------------------------------------------------------------------------------
280  // CDC setup and configuration routines. The idea is to help the library write with a default configuration
281  // structure. All pins HW that are fixed in their location will be set. A library programmer will just get
282  // that default structure and set the values necessary for the particular case.
283  //
284  //------------------------------------------------------------------------------------------------------------
285  CdcConfigDesc   getConfigDefault( );
286  CdcConfigDesc   *getConfigActual( );
287  void            printConfigInfo( CdcConfigDesc *ci );
288  void            setDebugLevel( uint8_t level = 0 );
289
290  uint8_t         init( CdcConfigDesc *ci );
291  void            fatalError( uint8_t n );
292  void            fatalErrorMsg( char *str, uint8_t n, uint8_t rStat );
```

```
293
294    //-------------------------------------------------------------------------------------------------------------
295    // General controller routines.
296    //
297    //-------------------------------------------------------------------------------------------------------------
298    uint16_t        getFamily( );
299    uint32_t        getVersion( );
300    uint32_t        getChipMemSize( );
301    uint32_t        getChipNvmSize( );
302    uint32_t        getCpuFrequency( );
303    uint32_t        getMillis( );
304    uint32_t        getMicros( );
305    void            sleepMillis( uint32_t val );
306    void            sleepMicros( uint32_t val );
307
308    //-------------------------------------------------------------------------------------------------------------
309    // The LCS runtime needs to build a unique ID for the node.
310    //
311    //-------------------------------------------------------------------------------------------------------------
312    uint32_t        createUid( );
313
314    //-------------------------------------------------------------------------------------------------------------
315    // Timer management routines.
316    //
317    //-------------------------------------------------------------------------------------------------------------
318    void            onTimerEvent( TimerCallback functionId );
319    void            startRepeatingTimer( uint32_t val );
320    void            setRepeatingTimerLimit( uint32_t val );
321    uint32_t        getRepeatingTimerLimit( );
322    void            stopRepeatingTimer( );
323
324    //-------------------------------------------------------------------------------------------------------------
325    // Analog input routines.
326    //
327    //-------------------------------------------------------------------------------------------------------------
328    uint8_t         configureAdc( uint8_t adcPin );
329    uint16_t        getAdcRefVoltage( );
330    uint16_t        getAdcDigitRange( );
331    uint16_t        readAdc( uint8_t adcPin );
332
333    //-------------------------------------------------------------------------------------------------------------
334    // Digital Input/Output routines.
335    //
336    //-------------------------------------------------------------------------------------------------------------
337    uint8_t         configureDio( uint8_t dioPin, uint8_t Mode = IN );
338    void            registerDioCallback( uint8_t dioPin, uint8_t event, CDC::GpioCallback func );
339    void            unregisterDioCallback( uint8_t dioPin );
340    bool            readDio( uint8_t dioPin );
341    uint8_t         writeDio( uint8_t dioPin, bool val );
342    uint8_t         toggleDio( uint8_t dioPin );
343    uint32_t        readDioMask( uint32_t dioMask );
344    uint8_t         writeDioMask( uint32_t dioMask, uint32_t dioVal );
345    uint8_t         writeDioPair( uint8_t dioPin1, bool val1, uint8_t dioPin2, bool val2 );
346
347    //-------------------------------------------------------------------------------------------------------------
348    // PWM output routines.
349    //
350    //-------------------------------------------------------------------------------------------------------------
351    uint8_t         configurePwm( uint8_t   pwmPin,
352                                  uint32_t  pwmFreqency,
353                                  bool      phaseCorrect  = true,
354                                  bool      inverted      = false
355                                );
356
357    uint8_t         writePwm( uint8_t pwmPin, uint8_t dutyCycle );
358
359    //-------------------------------------------------------------------------------------------------------------
360    // Serial IO routines.
361    //
362    //-------------------------------------------------------------------------------------------------------------
363    uint8_t         configureUart( uint8_t rxPin, uint8_t txPin, uint32_t baudRate, UartMode mode );
364    uint8_t         startUartRead( uint8_t rxPin );
365    uint8_t         stopUartRead( uint8_t rxPin );
366    uint8_t         getUartBuffer( uint8_t rxPin, uint8_t *buf, uint8_t bufLen );
367
368    //-------------------------------------------------------------------------------------------------------------
369    // I2C management routines.
370    //
371    //-------------------------------------------------------------------------------------------------------------
372    uint8_t         configureI2C( uint8_t sclPin, uint8_t sdaPin, uint32_t baudRate = 100 * 1000 );
373    uint8_t         i2cWrite( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit = false );
374    uint8_t         i2cRead( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit = false );
375
376    //-------------------------------------------------------------------------------------------------------------
377    // SPI management routines.
378    //
379    //-------------------------------------------------------------------------------------------------------------
380    uint8_t         configureSPI( uint8_t sclkPin, uint8_t mosiPin, uint8_t misoPin, uint32_t baudRate = 10 * 1000 * 1000 );
381    uint8_t         spiBeginTransaction( uint8_t sclkPin, uint8_t csPin );
382    uint8_t         spiEndTransaction( uint8_t sclkPin, uint8_t csPin );
383    uint8_t         spiRead( uint8_t sclkPin, uint8_t *buf, uint32_t len );
384    uint8_t         spiWrite( uint8_t sclkPin, uint8_t *buf, uint32_t len );
385
386    };
387
388    #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   // An ADC instance. The PICO supports up to three ADC inputs. When we use such an input, the corresponding
161   // instance data is kept in this structure. We also keep the PICO ADC number, so we can select the correct
162   // instance.
163   //
164   //----------------------------------------------------------------------------------------------------
165   struct AdcInst {
166
167       bool       configured  = false;
168       uint8_t    adcPin      = CDC::UNDEFINED_PIN;
169       uint8_t    adcNum      = 0;
170   };
171
172   //----------------------------------------------------------------------------------------------------
173   // A PWM output instance. GPIO pins can also be used as PWM output pins. The PWM output related data is
174   // kept in this instance.
175   //
176   //----------------------------------------------------------------------------------------------------
177   struct PwmInst {
178
179       bool       configured  = false;
180       uint8_t    pwmPin      = CDC::UNDEFINED_PIN;
181       uint32_t   wrap        = 0;
182
183       // ??? what else to keep around ?
184   };
185
186   //----------------------------------------------------------------------------------------------------
187   // A UART instance. UARTS are used to read in a serial stream from the RailCom detectors. There can be two
188   // hardware based UART instances, or up to four software defined instances. The instance also keeps a small
189   // buffer where the data is read into. We also keep the PICO UART HW instance used.
190   //
191   //----------------------------------------------------------------------------------------------------
192   struct UartInst {
193
194       bool             configured  = false;
195       uint8_t          rxPin       = CDC::UNDEFINED_PIN;
196       uint8_t          txPin       = CDC::UNDEFINED_PIN;
197       uint16_t         baudSetting = 0;
```

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

```
297  //------------------------------------------------------------------------------------------------------
298  // When no interrupt is configured for a GPIO pin, we set the table entry to a dummy handler. This way
299  // we do not have to check for a valid procedure label when we handle an interrupt.
300  //
301  //------------------------------------------------------------------------------------------------------
302  void dummyIsrHandler ( uint8_t pin, uint8_t event ) { }
303
304  //------------------------------------------------------------------------------------------------------
305  // Setup the ISR table. The PICO can have only one interrupt handler. When you want a handler per GPIO pin,
306  // the solution is to have a table when you keep the handler on a per pin base.
307  //
308  //------------------------------------------------------------------------------------------------------
309  void initIsrTable( ) {
310
311      for ( uint16_t i = 0; i < MAX_CPU_CORE; i++ ) {
312
313          for ( uint16_t j = 0; j < MAX_INT_PIN; j++ ) {
314
315              cdcIntHandlers.gpioIsrTable[ i ][ j ] = dummyIsrHandler;
316          }
317      }
318  }
319
320  //------------------------------------------------------------------------------------------------------
321  // The PICO uses a set of constants to describe the interrupt type. We map our interrupt types to the PICO
322  // GPIO_IRQ_xxx types.
323  //
324  //------------------------------------------------------------------------------------------------------
325  uint32_t mapGpioIntEvent( uint8_t event ) {
326
327      switch ( event ) {
328
329          case CDC::EVT_LOW:     return( GPIO_IRQ_LEVEL_LOW );
330          case CDC::EVT_HIGH:    return( GPIO_IRQ_LEVEL_HIGH );
331          case CDC::EVT_FALL:    return( GPIO_IRQ_EDGE_FALL );
332          case CDC::EVT_RISE:    return( GPIO_IRQ_EDGE_RISE );
333          case CDC::EVT_CHANGE:  return( GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL );
334          default:      return( 0 );
335      }
336  }
337
338  //------------------------------------------------------------------------------------------------------
339  // The PICO uses a set of constants to describe the interrupt type. We map them to our types.
340  //
341  //------------------------------------------------------------------------------------------------------
342  uint8_t mapPicoGpioEvent( uint32_t event ) {
343
344      switch ( event ) {
345
346          case GPIO_IRQ_LEVEL_LOW:  return( CDC::EVT_LOW );
347          case GPIO_IRQ_LEVEL_HIGH: return( CDC::EVT_HIGH );
348          case GPIO_IRQ_EDGE_FALL:  return( CDC::EVT_FALL );
349          case GPIO_IRQ_EDGE_RISE:  return( CDC::EVT_RISE );
350          default:                  return( 0 );
351      }
352  }
353
354  //------------------------------------------------------------------------------------------------------
355  // Global Interrupt handlers. The hardware and low level library will call these handlers, which in turn
356  // will invoke the respective callback function if configured. The GPIO interrupt handler manages the
357  // handler for all possible IO pins. The PICO can only have one interrupt routine, so we feature an array
358  // of handlers where a handler for a GPIO pin can be registered. If there is a handler set, we just invoke
359  // it. The other handlers are for the timer and the UART hardware.
360  //
361  //------------------------------------------------------------------------------------------------------
362  void gpioCallback( uint gpioPin, uint32_t event ) {
363
364      cdcIntHandlers.gpioIsrTable[ get_core_num( )][ gpioPin] ( gpioPin, mapPicoGpioEvent( event ));
365  }
366
367  bool repeatingTimerAlarm( repeating_timer_t *rt ) {
368
369      if ( timerCallback != nullptr ) timerCallback((uint32_t) ( - timerData.delay_us ));
370      return ( true );
371  }
372
373  void uartRxCallback0( ) {
374
375      while ( uart_is_readable( uart0 )) {
376
377          uint8_t ch = uart_getc( uart0 );
378          if ( CdcUart0.rxBufIndex < MAX_UART_BUF_SIZE ) CdcUart0.rxDataBuf[CdcUart0.rxBufIndex++ ] = ch;
379      }
380  }
381
382  void uartRxCallback1( ) {
383
384      while ( uart_is_readable( uart1 )) {
385
386          uint8_t ch = uart_getc( uart1 );
387          if ( CdcUart1.rxBufIndex < MAX_UART_BUF_SIZE ) CdcUart1.rxDataBuf[ CdcUart1.rxBufIndex++ ] = ch;
388      }
389  }
390
391  //------------------------------------------------------------------------------------------------------
392  // The default configuration descriptor. The Application program fills in such a structure, which can be
393  // seen as the HW pin assignments for the PICO controllers and the particular board on which the application
394  // will be deployed. The application will simply use the field names to address the particular PICO HW
395  // function. For example, a configuration has mapped DIO_PIN_5 to GPIO pin 12, because that is where the
```

```
396   // particular board has mapped DIO_PIN_5 to the hardware line. The application will just use the DIO_PIN_5
397   // field when talking to that GPIO pin. Whenever the board layout changes, there could be another PICO GPIO
398   // pin, but the name "DIO_PIN_5" for the application upper layers does not change.
399   //
400   // Note that there is a great flexibility what a PICO HW  pin can do and hence a lot of our fields are just
401   // "UNDEFINED" with no constraints. Nevertheless, there is a function which will do some plausibility checks
402   // for such a structure. Also, each configuration routine will do again a check that the GPIO pins used do
403   // indeed map to a PICO HW block for the desired purpose.
404   //
405   // The configuration structure does not replace the actual configuration calls to make to the CDC library.
406   // It is just a mapping of reserved names to actual GPIO pins.
407   //
408   //-----------------------------------------------------------------------------------------------------------
409   CDC::CdcConfigDesc getConfigDefaultRP2040( ) {
410
411       CDC::CdcConfigDesc tmp;
412
413       tmp.CFG_STATUS          = CDC::INIT_PENDING;
414
415       // ??? controller family ?
416       // ??? what other characteristics ? ( e.g. mem size ? )
417
418       tmp.READY_LED_PIN       = CDC::UNDEFINED_PIN;
419       tmp.ACTIVE_LED_PIN      = CDC::UNDEFINED_PIN;
420
421       tmp.EXT_INT_PIN         = CDC::UNDEFINED_PIN;
422       tmp.PFAIL_PIN           = CDC::UNDEFINED_PIN;
423
424       tmp.DIO_PIN_0           = CDC::UNDEFINED_PIN;
425       tmp.DIO_PIN_1           = CDC::UNDEFINED_PIN;
426       tmp.DIO_PIN_2           = CDC::UNDEFINED_PIN;
427       tmp.DIO_PIN_3           = CDC::UNDEFINED_PIN;
428       tmp.DIO_PIN_4           = CDC::UNDEFINED_PIN;
429       tmp.DIO_PIN_5           = CDC::UNDEFINED_PIN;
430       tmp.DIO_PIN_6           = CDC::UNDEFINED_PIN;
431       tmp.DIO_PIN_7           = CDC::UNDEFINED_PIN;
432       tmp.DIO_PIN_8           = CDC::UNDEFINED_PIN;
433       tmp.DIO_PIN_9           = CDC::UNDEFINED_PIN;
434       tmp.DIO_PIN_10          = CDC::UNDEFINED_PIN;
435       tmp.DIO_PIN_11          = CDC::UNDEFINED_PIN;
436       tmp.DIO_PIN_12          = CDC::UNDEFINED_PIN;
437       tmp.DIO_PIN_13          = CDC::UNDEFINED_PIN;
438       tmp.DIO_PIN_14          = CDC::UNDEFINED_PIN;
439       tmp.DIO_PIN_15          = CDC::UNDEFINED_PIN;
440
441       tmp.ADC_PIN_0           = CDC::UNDEFINED_PIN;
442       tmp.ADC_PIN_1           = CDC::UNDEFINED_PIN;
443       tmp.ADC_PIN_2           = CDC::UNDEFINED_PIN;
444       tmp.ADC_PIN_3           = CDC::ILLEGAL_PIN;
445
446       tmp.PWM_PIN_0           = CDC::UNDEFINED_PIN;
447       tmp.PWM_PIN_1           = CDC::UNDEFINED_PIN;
448       tmp.PWM_PIN_2           = CDC::UNDEFINED_PIN;
449       tmp.PWM_PIN_3           = CDC::UNDEFINED_PIN;
450
451       tmp.UART_RX_PIN_0       = CDC::UNDEFINED_PIN;
452       tmp.UART_TX_PIN_0       = CDC::UNDEFINED_PIN;
453
454       tmp.UART_RX_PIN_1       = CDC::UNDEFINED_PIN;
455       tmp.UART_TX_PIN_1       = CDC::UNDEFINED_PIN;
456
457       tmp.UART_RX_PIN_2       = CDC::UNDEFINED_PIN;
458       tmp.UART_TX_PIN_2       = CDC::UNDEFINED_PIN;
459
460       tmp.UART_RX_PIN_3       = CDC::UNDEFINED_PIN;
461       tmp.UART_TX_PIN_3       = CDC::UNDEFINED_PIN;
462
463       tmp.SPI_MOSI_PIN_0      = CDC::UNDEFINED_PIN;
464       tmp.SPI_MISO_PIN_0      = CDC::UNDEFINED_PIN;
465       tmp.SPI_SCLK_PIN_0      = CDC::UNDEFINED_PIN;
466
467       tmp.SPI_MOSI_PIN_1      = CDC::UNDEFINED_PIN;
468       tmp.SPI_MISO_PIN_1      = CDC::UNDEFINED_PIN;
469       tmp.SPI_SCLK_PIN_1      = CDC::UNDEFINED_PIN;
470
471       tmp.NVM_I2C_SCL_PIN     = CDC::UNDEFINED_PIN;
472       tmp.NVM_I2C_SDA_PIN     = CDC::UNDEFINED_PIN;
473
474       tmp.EXT_I2C_SCL_PIN     = 17;
475       tmp.EXT_I2C_SDA_PIN     = 16;
476
477       tmp.CAN_BUS_RX_PIN      = CDC::UNDEFINED_PIN;
478       tmp.CAN_BUS_TX_PIN      = CDC::UNDEFINED_PIN;
479
480       return ( tmp );
481   }
482
483   //-----------------------------------------------------------------------------------------------------------
484   // Validate a configuration structure. This routine will do basic checking of the pin configuration passed.
485   // The PICO is very flexible when it comes to what a pin can do. However, there are still some rules to
486   // follow. Also, we have dedicated settings for at least the I2C channels and the CAN bus IO pins.
487   //
488   //-----------------------------------------------------------------------------------------------------------
489   uint8_t validateConfigRP20040( CDC::CdcConfigDesc *ci ) {
490
491       // ??? a ton of "validXXX" ?
492
493       return ( NO_ERR ); // for now....
494   }
```

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

```
594   //
595   //------------------------------------------------------------------------------------------------------------
596   void fatalErrorMsg( char *str, uint8_t n, uint8_t rStat ) {
597
598       if ( isConsoleConnected( )) printf( "Fatal Error: %d: %s, rStat: %d\n", n, str, rStat );
599       fatalError( n );
600   }
601
602   //------------------------------------------------------------------------------------------------------------
603   // Processor general values required by the low level LCS core library functions.
604   //
605   //------------------------------------------------------------------------------------------------------------
606   uint16_t getFamily( ) {
607
608       return ( CONTROLLER_FAMILY );
609   }
610
611   uint32_t getVersion( ) {
612
613       return ( CDC_LIB_MAJOR_VERSION << 8 | CDC_LIB_MINOR_VERSION );
614   }
615
616   uint32_t getChipMemSize( ) {
617
618       return ( CHIP_MEM_SIZE );
619   }
620
621   uint32_t getChipNvmSize( ) {
622
623       return ( CHIP_NVM_SIZE   );
624   }
625
626   uint32_t getCpuFrequency( ) {
627
628       return ( clock_get_hz( clk_sys ));
629   }
630
631   uint32_t getMillis( ) {
632
633       return ( to_ms_since_boot( get_absolute_time( )));
634   }
635
636   uint32_t getMicros( ) {
637
638       return ( to_us_since_boot( get_absolute_time( )));
639   }
640
641   void sleepMillis( uint32_t val ) {
642
643       sleep_ms( val );
644   }
645
646   void sleepMicros( uint32_t val ) {
647
648       sleep_us( val );
649   }
650
651   //------------------------------------------------------------------------------------------------------------
652   // "createUid" is the routine that produces a unique ID for the node. The scheme is still based on a random
653   // number. This is the PICO version for creating a random number. Alternatively we could use the unique
654   // flash chip ID on the board. TBD ...
655   //
656   //------------------------------------------------------------------------------------------------------------
657   uint32_t createUid( ) {
658
659       uint32_t rVal = 0;
660
661       volatile uint32_t *rnd_reg = (uint32_t *) ( ROSC_BASE + ROSC_RANDOMBIT_OFFSET );
662
663       for ( int k = 0; k < 32; k++ ) {
664
665           rVal = rVal << 1;
666           rVal = rVal + ( 0x00000001 & ( *rnd_reg ));
667       }
668
669       return ( rVal );
670   }
671
672   //------------------------------------------------------------------------------------------------------------
673   // Console IO section. We set up the stdio via the USB connector. As part of the CDC init call, the configure
674   // call should be done rather early, so that we can print out debug messages. In normal LCS node operation
675   // there is no USB connected. Detecting a connection helps to decide whether we can report an error or need
676   // to resort to a fatal error call at startup.
677   //
678   // There are two basic ways to detect an USB connection. The first is to simply check if there is power on
679   // the USB port. The PICO features an internal GPIO pin for this purpose. Using this method still does not
680   // mean that we have someone connected to the USB, but just that there is a cable with power. Well, good
681   // enough for us. The second method truly detects that there is a USB host connected. This check is provided
682   // via the PICO libraries which in turn use the tinyUSB library. However, there could be a timing problem
683   // where the USB stack is not ready and we conclude wrongly that there is no USB connection. For now, let's
684   // rather go with the risk that there is just power on the USB connector.
685   //
686   // Finally, there is a routine to get a character for the command interfaces. Since the function just reads
687   // in a character, optionally with a timeout how ling to wait for any inout.
688   //
689   // PS: The USB check way would be "return( stdio_usb_connected( ));" instead of the GPIO check.
690   //------------------------------------------------------------------------------------------------------------
691   uint8_t configureConsoleIO( ) {
692
```

```
693        stdio_init_all( );
694        return( NO_ERR );
695    }
696
697    bool isConsoleConnected( ) {
698
699        gpio_init( PICO_VBUS_PIN );
700        gpio_set_dir( PICO_VBUS_PIN, GPIO_IN );
701
702        return( gpio_get( PICO_VBUS_PIN ));
703    }
704
705    char getConsoleChar( uint32_t timeoutVal ) {
706
707        int ch = getchar_timeout_us( timeoutVal );
708        return(( ch == PICO_ERROR_TIMEOUT ) ? 0 : ch );
709    }
710
711    //-------------------------------------------------------------------------------------------------------
712    // Timer section. The CDC library features one generic repeating timer with a microsecond resolution. The
713    // routines start and stop the timer and allow to set a new limit. The PICO offers a high level function that
714    // schedules a repeating timer with the property of measuring the interval also from the start of the
715    // callback invocation. This is exactly what we need to implement the tick interrupt for the DCC signal state
716    // machine. The "setRepeatingTimerLimit" function will adjust the timer limit counter while the timer already
717    // is counting toward a limit. Note that the timer option that already start the next round while the timer
718    // interrupt handler executes is specified by using negative limit values. The timer functionality also
719    // offers two timestamp routines to get the number of milliseconds and number of microseconds since system
720    // start.
721    //
722    // ??? would we one day need more than one timer instance ?
723    //-------------------------------------------------------------------------------------------------------
724    void startRepeatingTimer( uint32_t val ) {
725
726        int64_t limit = val;
727        add_repeating_timer_us( - limit, repeatingTimerAlarm, nullptr, &timerData );
728    }
729
730    void stopRepeatingTimer( ) {
731
732        cancel_repeating_timer( &timerData );
733    }
734
735    uint32_t getRepeatingTimerLimit( ) {
736
737        return ((uint32_t) ( - timerData.delay_us ));
738    }
739
740    void setRepeatingTimerLimit( uint32_t val ) {
741
742        int64_t limit = val;
743        timerData.delay_us = ((int64_t) - limit );
744    }
745
746    void onTimerEvent( CDC::TimerCallback functionId ) {
747
748        timerCallback = functionId;
749    }
750
751    //-------------------------------------------------------------------------------------------------------
752    // DIO section. A digital pin is the bread and butter hardware resource and can be an input or output pin. For
753    // inputs, an internal pull-up resistor can be set.There are a couple of interfaces. First the single pin
754    // read, write and toggle. Next are read and write mask routines which work on all IO pins at once. Note that
755    // no cross checking is done if the pins are used by other CDC functions. Finally there is a convenience
756    // routine which write a pair of data. This is typically used for the H-Bridge control pins, which are set at
757    // the same time.
758    //
759    // A GPIO pin can also have an attached interrupt handler. When we register a handler for a pin, there are
760    // two different PICO lib routines to use. When there is no handler registered so far, we register the
761    // common callback and store the particular GPIO handler in the handler table. Otherwise, we just store the
762    // handler and enable the GPIO pin for interrupts.
763    //
764    //-------------------------------------------------------------------------------------------------------
765    uint8_t configureDio( uint8_t dioPin, uint8_t mode ) {
766
767        if ( ! validPin( dioPin, VALID_GPIO_PINS )) return ( DIO_PIN_ERR );
768
769        gpio_init( dioPin );
770
771        switch ( mode ) {
772
773            case IN:  gpio_set_dir( dioPin, false );  break;
774            case OUT: {
775
776                gpio_set_dir( dioPin, true );
777                gpio_set_drive_strength ( dioPin, GPIO_DRIVE_STRENGTH_12MA );
778
779            }  break;
780
781            case IN_PULLUP: {
782
783                gpio_set_dir( dioPin, false );
784                gpio_pull_up( dioPin );
785
786            } break;
787
788            default: gpio_set_dir( dioPin, false );
789        }
790
791        return ( NO_ERR );
```

```
792  }
793
794  void registerDioCallback( uint8_t dioPin, uint8_t event, CDC::GpioCallback func ) {
795
796      if ( dioPin <= MAX_INT_PIN ) {
797
798          if ( cdcIntHandlers.numOfHandlers == 0 )
799              gpio_set_irq_enabled_with_callback( dioPin, mapGpioIntEvent( event ), true, gpioCallback );
800          else
801              gpio_set_irq_enabled( dioPin, mapGpioIntEvent( event ), true);
802
803          cdcIntHandlers.gpioIsrTable[ get_core_num( ) ][ dioPin ] = func;
804          cdcIntHandlers.numOfHandlers ++;
805      }
806  }
807
808  void unregisterDioCallback( uint8_t dioPin ) {
809
810      if ( dioPin <= MAX_INT_PIN ) {
811
812          if ( cdcIntHandlers.gpioIsrTable[ get_core_num( ) ][ dioPin ] != nullptr ) {
813
814              gpio_set_irq_enabled( dioPin, 0, false );
815              cdcIntHandlers.gpioIsrTable[ get_core_num( ) ][ dioPin ] = dummyIsrHandler;
816              cdcIntHandlers.numOfHandlers --;
817          }
818      }
819  }
820
821  bool readDio( uint8_t dioPin ) {
822
823      return ( gpio_get( dioPin ));
824  }
825
826  uint8_t writeDio( uint8_t dioPin, bool val ) {
827
828      gpio_put( dioPin, val );
829      return ( NO_ERR );
830  }
831
832  uint8_t toggleDio( uint8_t dioPin ) {
833
834      writeDio( dioPin, ! readDio( dioPin ));
835      return ( NO_ERR );
836  }
837
838  uint8_t writeDioPair( uint8_t dioPin1, bool val1, uint8_t dioPin2, bool val2 ) {
839
840      uint32_t maskData = ( 1UL << dioPin1 ) | ( 1UL << dioPin2 );
841      uint32_t valData  = (( val1 ) ? ( 1 << dioPin1 ) : 0 ) | (( val2 ) ? ( 1 << dioPin2 ) : 0 );
842
843      gpio_put_masked( maskData, valData );
844      return ( NO_ERR );
845  }
846
847  uint32_t readDioMask( uint32_t dioMask ) {
848
849      return ( gpio_get_all( ) & dioMask );
850  }
851
852  uint8_t writeDioMask( uint32_t dioMask, uint32_t dioVal ) {
853
854      gpio_put_masked( dioMask, dioVal );
855      return ( NO_ERR );
856  }
857
858  //-------------------------------------------------------------------------------------------------------------------
859  // ADC section. The analog input channel represented by the pin is configured. At initialization, the ADC pin
860  // number is validated and the ADC subsystem initialized. The PICO does an analog read in about 2us. This is
861  // so fast, it does for our purpose make not much sense to implement an asynchronous option. Furthermore, the
862  // ADC value scaled down to a 10-bit resolution.
863  //
864  //-------------------------------------------------------------------------------------------------------------------
865  uint8_t configureAdc( uint8_t adcPin ) {
866
867      if ( ! validPin( adcPin, VALID_ADC_PINS ))  return ( ADC_PIN_ERR );
868
869      AdcInst *tmp = nullptr;
870
871      if ( adcPin == cfg.ADC_PIN_0 ) {
872
873          tmp = &CdcAdc0;
874          tmp -> adcPin = adcPin;
875          tmp -> adcNum = 0;
876
877      }
878       else if ( adcPin == cfg.ADC_PIN_1 ) {
879
880          tmp = &CdcAdc1;
881          tmp -> adcPin = adcPin;
882          tmp -> adcNum = 1;
883      }
884      else if ( adcPin == cfg.ADC_PIN_2 ) {
885
886          tmp = &CdcAdc2;
887          tmp -> adcPin = adcPin;
888          tmp -> adcNum = 2;
889      }
890      else return ( ADC_PIN_ERR );
```

```
891
892        adc_init( );
893        adc_gpio_init( tmp -> adcPin );
894        tmp -> configured = true;
895
896        return ( NO_ERR );
897    }
898
899    uint16_t getAdcRefVoltage( ) {
900
901        return ( ADC_REF_VOLTAGE_MILLI_VOLT );
902    }
903
904    uint16_t getAdcDigitRange( ) {
905
906        return ( ADC_DIGIT_RANGE );
907    }
908
909    uint16_t readAdc( uint8_t adcPin ) {
910
911        AdcInst *tmp = nullptr;
912
913        if      ( adcPin == CdcAdc0.adcPin ) tmp = &CdcAdc0;
914        else if ( adcPin == CdcAdc1.adcPin ) tmp = &CdcAdc1;
915        else if ( adcPin == CdcAdc2.adcPin ) tmp = &CdcAdc2;
916        else return ( 0 );
917        adc_select_input( tmp -> adcNum );
918        return ( adc_read( ) >> 2 );
919    }
920
921    //------------------------------------------------------------------------------------------------------------
922    // UART section. The UART interface is primarily used for the RailCom Detector that sends a serial signal.
923    // So far, only the receiver portion is implemented because that is all what is needed for RailCom messages.
924    // There are two general categories. The first uses the PICO built-in UART hardware blocks. The second
925    // implements a software UART based on the PICO PIO blocks.
926    //
927    // There are three routines. The "startUartRead" will enable the UART and start reading bytes into the local
928    // buffer. The "stopUartRead" will then finish the byte collection and disable the UART again. Finally, the
929    // "getUartBuffer" routine will return the bytes received. Again, note that this is not a generic UART read
930    // interface.
931    //
932    // The work on the PIO based UART version has not started yet ... it will be needed for the quad block
933    // controller. Looking forward to it ...:-)
934    //
935    //------------------------------------------------------------------------------------------------------------
936    uint8_t configureUart( uint8_t rxPin, uint8_t txPin, uint32_t baudRate, UartMode mode ) {
937
938        UartInst *uart = nullptr;
939
940        if ( mode == UART_MODE_8N1 ) {
941
942            if (( validPin( rxPin, VALID_UART_0_RX_PINS )) && ( validPin( txPin, VALID_UART_0_TX_PINS ))) {
943
944                uart                = &CdcUart0;
945                uart -> uartMode    = mode;
946                uart -> rxPin       = rxPin;
947                uart -> txPin       = txPin;
948                uart -> dataBits    = 8;
949                uart -> stopBits    = 1;
950                uart -> parityMode  = UART_PARITY_NONE;
951                uart -> uartHw      = uart0;
952                uart -> uartIrq     = UART0_IRQ;
953            }
954            else if (( validPin( rxPin, VALID_UART_1_RX_PINS )) && ( validPin( txPin, VALID_UART_1_TX_PINS ))) {
955
956                uart                = &CdcUart1;
957                uart -> uartMode    = mode;
958                uart -> rxPin       = rxPin;
959                uart -> txPin       = txPin;
960                uart -> dataBits    = 8;
961                uart -> stopBits    = 1;
962                uart -> parityMode  = UART_PARITY_NONE;
963                uart -> uartHw      = uart1;
964                uart -> uartIrq     = UART1_IRQ;
965            }
966            else return ( UART_PORT_ERR );
967
968            uart_init( uart -> uartHw, baudRate );
969            gpio_set_function( rxPin, GPIO_FUNC_UART );
970            gpio_set_function( txPin, GPIO_FUNC_UART );
971            uart_set_hw_flow( uart -> uartHw, false, false );
972            uart_set_format( uart -> uartHw, uart -> dataBits, uart -> stopBits, uart -> parityMode );
973            uart_set_fifo_enabled( uart -> uartHw, false );
974
975            if      ( uart -> uartIrq == UART0_IRQ ) irq_set_exclusive_handler( uart -> uartIrq, uartRxCallback0 );
976            else if ( uart -> uartIrq == UART1_IRQ ) irq_set_exclusive_handler( uart -> uartIrq, uartRxCallback1 );
977
978            irq_set_enabled( uart -> uartIrq, true );
979
980            return ( NO_ERR );
981        }
982        else if ( mode == UART_MODE_8N1_PIO ) {
983
984            return ( NOT_SUPPORTED );
985        }
986        else return ( NOT_SUPPORTED );
987    }
988
989    uint8_t startUartRead( uint8_t rxPin ) {
```

```
990
991        UartInst *uart = nullptr;
992
993        if       ( rxPin == CdcUart0.rxPin ) uart = &CdcUart0;
994        else if ( rxPin == CdcUart1.rxPin ) uart = &CdcUart1;
995        else if ( rxPin == CdcUart2.rxPin ) uart = &CdcUart2;
996        else if ( rxPin == CdcUart3.rxPin ) uart = &CdcUart3;
997        else                               return ( CDC::UART_PORT_ERR );
998
999        if (( uart != nullptr ) && ( uart -> uartMode == UART_MODE_8N1 )) {
1000
1001           uart_set_irq_enables( uart -> uartHw, true, false );
1002           uart -> rxBufIndex = 0;
1003           return ( NO_ERR );
1004        }
1005        else if (( uart != nullptr ) && ( uart -> uartMode == UART_MODE_8N1_PIO )) {
1006
1007           return ( NOT_SUPPORTED );
1008        }
1009        else return ( UART_PORT_ERR );
1010    }
1011
1012    uint8_t stopUartRead( uint8_t rxPin ) {
1013
1014        UartInst *uart = nullptr;
1015
1016        if       ( rxPin == CdcUart0.rxPin ) uart = &CdcUart0;
1017        else if ( rxPin == CdcUart1.rxPin ) uart = &CdcUart1;
1018        else if ( rxPin == CdcUart2.rxPin ) uart = &CdcUart2;
1019        else if ( rxPin == CdcUart3.rxPin ) uart = &CdcUart3;
1020
1021        if (( uart != nullptr ) && ( uart ->uartMode == UART_MODE_8N1 )) {
1022
1023           uart_set_irq_enables( uart -> uartHw, false, false );
1024           return ( NO_ERR );
1025        }
1026        else if (( uart != nullptr ) && ( uart -> uartMode == UART_MODE_8N1_PIO )) {
1027
1028           return ( NOT_SUPPORTED );
1029        }
1030        else return ( UART_PORT_ERR );
1031    }
1032
1033    uint8_t getUartBuffer( uint8_t rxPin, uint8_t *buf, uint8_t bufLen ) {
1034
1035        UartInst *uart = nullptr;
1036
1037        if       ( rxPin == CdcUart0.rxPin ) uart = &CdcUart0;
1038        else if ( rxPin == CdcUart1.rxPin ) uart = &CdcUart1;
1039        else if ( rxPin == CdcUart2.rxPin ) uart = &CdcUart2;
1040        else if ( rxPin == CdcUart3.rxPin ) uart = &CdcUart3;
1041        else                               return ( 0 );
1042
1043        if (( uart != nullptr ) && ( uart -> rxBufIndex > 0 ) && ( bufLen > 0 )) {
1044
1045           uint8_t i = 0;
1046           while (( i < uart -> rxBufIndex ) && ( i < bufLen )) {
1047
1048              buf[ i ] = uart -> rxDataBuf[ i ];
1049              i++;
1050           }
1051
1052           return ( i );
1053        }
1054        else return ( 0 );
1055    }
1056
1057    //-------------------------------------------------------------------------------------------------------
1058    // PWM section. The PICO is quite flexible when it comes to PWM signals. We implement a simple PWM capability.
1059    // There is the frequency which set during configuration and there is the write operation which set the duty
1060    // cycle. The calculations are best described in the PICO C++ SDK. We do the setting of phase, wrap count,
1061    // etc. once when we configure the PWM channel. All the "writePwm" function then will do is to manipulate the
1062    // duty cycle. In other words, when we change the frequency we need to configure again.
1063    //
1064    // There is one small issue left. Channel come in pairs. For some reason there is no call to individually
1065    // set the "inverted" option on a channel. When we set the inverted option for a pin, we currently also set
1066    //  the inverted option for the other channel since we just don't know better. To be correct, all possible
1067    // PWM pins and their "inverted" option would need to be stored somewhere.
1068    //
1069    // To do .... ( there is a way via the pwm_Config CSR field... )
1070    //
1071    //-------------------------------------------------------------------------------------------------------
1072    uint8_t configurePwm( uint8_t pwmPin, uint32_t pwmFreqency, bool phaseCorrect, bool inverted ) {
1073
1074        PwmInst *pwm = nullptr;
1075
1076        if       ( pwmPin == cfg.PWM_PIN_0 ) pwm = &CdcPwm0;
1077        else if ( pwmPin == cfg.PWM_PIN_1 ) pwm = &CdcPwm1;
1078        else if ( pwmPin == cfg.PWM_PIN_2 ) pwm = &CdcPwm2;
1079        else if ( pwmPin == cfg.PWM_PIN_3 ) pwm = &CdcPwm3;
1080        else                               return ( PWM_PIN_ERR );
1081
1082        if ( phaseCorrect ) pwmFreqency = pwmFreqency * 2;
1083
1084        uint32_t sysClock = getCpuFrequency( );
1085        uint32_t clkDiv  = sysClock / pwmFreqency / 4096 + ( sysClock % ( pwmFreqency * 4096 ) != 0 );
1086
1087        if ( clkDiv / 16 == 0 ) clkDiv = 16;
1088
```

```
1089        pwm -> pwmPin   = pwmPin;
1090        pwm -> wrap     = sysClock * 16 / clkDiv / pwmFreqency - 1;
1091
1092        pwm_config pwmConfig = pwm_get_default_config( );
1093        gpio_set_function( pwm -> pwmPin, GPIO_FUNC_PWM );
1094        pwm_config_set_wrap( &pwmConfig, pwm -> wrap );
1095        pwm_config_set_phase_correct( &pwmConfig, phaseCorrect );
1096        pwm_config_set_output_polarity( &pwmConfig, inverted, inverted );
1097        pwm_init ( pwm_gpio_to_slice_num( pwm -> pwmPin ), &pwmConfig, false );
1098        pwm_set_clkdiv_int_frac( pwm_gpio_to_slice_num( pwm -> pwmPin ), clkDiv / 16, clkDiv & 0xF );
1099
1100        #if CDC_DEBUG == 1
1101
1102        printf( "PWM Pin: % d, fPwm: % d, phase: % d, inverted: % d, clkDiv: % d, wrap: % d \n",
1103                   pwm -> pwmPin, pwmFreqency,  phaseCorrect, inverted, clkDiv, pwm -> wrap );
1104        #endif
1105
1106        return ( NO_ERR );
1107    }
1108
1109    uint8_t writePwm( uint8_t pwmPin, uint8_t dutyCycle ) {
1110
1111        PwmInst *pwm = nullptr;
1112
1113        if      ( pwmPin == cfg.PWM_PIN_0 ) pwm = &CdcPwm0;
1114        else if ( pwmPin == cfg.PWM_PIN_1 ) pwm = &CdcPwm1;
1115        else if ( pwmPin == cfg.PWM_PIN_2 ) pwm = &CdcPwm2;
1116        else if ( pwmPin == cfg.PWM_PIN_3 ) pwm = &CdcPwm3;
1117        else                                return ( PWM_PIN_ERR );
1118
1119        uint sliceNum = pwm_gpio_to_slice_num( pwmPin );
1120        uint channel  = pwm_gpio_to_channel( pwmPin );
1121
1122        if ( dutyCycle == 0 ) {
1123
1124            pwm_set_enabled( sliceNum, false );
1125            writeDio( pwmPin, false );
1126        }
1127        else if ( dutyCycle == 255 ) {
1128
1129            pwm_set_enabled( sliceNum, false );
1130            writeDio( pwmPin, true );
1131        }
1132        else {
1133
1134            pwm_set_chan_level( sliceNum, channel, ( pwm -> wrap * dutyCycle / 256 ));
1135            pwm_set_enabled( sliceNum, true );
1136        }
1137
1138        return ( NO_ERR );
1139    }
1140
1141    //-------------------------------------------------------------------------------------------------------
1142    // I2C Section. The PICO has two HW blocks for I2C interfaces. The interface implements a simple read and
1143    // write access to an I2C element. There is a timeout to avoid waiting forever on an operation.
1144    //
1145    //-------------------------------------------------------------------------------------------------------
1146    uint8_t configureI2C( uint8_t sclPin, uint8_t sdaPin, uint32_t baudRate ) {
1147
1148        I2CInst *i2c = nullptr;
1149
1150        if ((( 1 << sclPin ) & VALID_I2C_0_SCL_PINS ) && (( 1 << sdaPin ) & VALID_I2C_0_SDA_PINS )) {
1151
1152            i2c = &CdcI2C0;
1153            i2c -> i2cHw = i2c0;
1154        }
1155        else if ((( 1 << sclPin ) & VALID_I2C_1_SCL_PINS ) && (( 1 << sdaPin ) & VALID_I2C_1_SDA_PINS )) {
1156
1157            i2c = &CdcI2C1;
1158            i2c -> i2cHw = i2c1;
1159        }
1160        else return ( CDC::I2C_PORT_ERR );
1161
1162        i2c -> sclPin      = sclPin;
1163        i2c -> sdaPin      = sdaPin;
1164        i2c -> baudRate    = baudRate;
1165        i2c -> timeoutValMs = I2C_TIME_OUT_IN_MS;
1166        i2c -> configured  = true;
1167
1168        i2c_init( i2c -> i2cHw, i2c -> baudRate );
1169        i2c_set_slave_mode( i2c -> i2cHw, false, 0 );
1170
1171        gpio_set_function( i2c -> sclPin, GPIO_FUNC_I2C );
1172        gpio_set_function( i2c -> sdaPin, GPIO_FUNC_I2C);
1173        gpio_pull_up( i2c -> sclPin );
1174        gpio_pull_up( i2c -> sdaPin );
1175
1176        return ( NO_ERR );
1177    }
1178
1179    uint8_t i2cRead( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit ) {
1180
1181        I2CInst *i2c = nullptr;
1182
1183        if      (( CdcI2C0.sclPin == sclPin ) && ( CdcI2C0.configured )) i2c = &CdcI2C0;
1184        else if (( CdcI2C1.sclPin == sclPin ) && ( CdcI2C1.configured )) i2c = &CdcI2C1;
1185        else return ( I2C_PORT_ERR );
1186
1187        auto ret = i2c_read_blocking_until( i2c -> i2cHw,
```

129

```
1188                                                i2cAdr ,
1189                                                buf ,
1190                                                len ,
1191                                                stopBit ,
1192                                                make_timeout_time_ms ( i2c -> timeoutValMs ));
1193
1194        #if CDC_DEBUG == 1
1195        printf ( "i2cRead: scl: %d, i2c: 0x%x, buf: %p, buf[0] %x, buf[1] %x, len: %d, stop: %d\n",
1196                   sclPin , i2cAdr , buf , buf [0], buf [1], len , stopBit );
1197        if ( ret == PICO_ERROR_GENERIC ) printf ( "I2C read, PICO generic error\n" );
1198        if ( ret == PICO_ERROR_TIMEOUT ) printf ( "I2C read, PICO timeout error\n" );
1199        #endif
1200
1201        if (( ret == PICO_ERROR_GENERIC ) || ( ret == PICO_ERROR_TIMEOUT )) return ( I2C_READ_ERR );
1202
1203        return ( NO_ERR );
1204    }
1205
1206    uint8_t i2cWrite ( uint8_t sclPin , uint8_t i2cAdr , uint8_t *buf , uint16_t len , bool stopBit ) {
1207
1208        #if CDC_DEBUG == 1
1209        printf ( "i2cWrite: scl: %d, i2c: 0x%x, buf: %p, buf[0] %x, buf[1] %x, len: %d, stop: %d\n",
1210                   sclPin , i2cAdr , buf , buf [0], buf [1], len , stopBit );
1211        #endif
1212
1213
1214        I2CInst *i2c = nullptr;
1215
1216        if       (( CdcI2C0.sclPin == sclPin ) && ( CdcI2C0.configured )) i2c = &CdcI2C0;
1217        else if (( CdcI2C1.sclPin == sclPin ) && ( CdcI2C1.configured )) i2c = &CdcI2C1;
1218        else return ( I2C_PORT_ERR );
1219
1220        auto ret = i2c_write_blocking_until ( i2c -> i2cHw ,
1221                                              i2cAdr ,
1222                                              buf ,
1223                                              len ,
1224                                              stopBit ,
1225                                              make_timeout_time_ms ( i2c -> timeoutValMs ));
1226
1227        #if CDC_DEBUG == 1
1228        if ( ret == PICO_ERROR_GENERIC ) printf ( "I2C write, PICO generic error\n" );
1229        if ( ret == PICO_ERROR_TIMEOUT ) printf ( "I2C write, PICO timeout error\n" );
1230        #endif
1231
1232        if (( ret == PICO_ERROR_TIMEOUT) || ( ret == PICO_ERROR_GENERIC ) || ( ret != len )) return ( I2C_WRITE_ERR );
1233
1234        return ( NO_ERR );
1235    }
1236
1237    //-------------------------------------------------------------------------------------------------------
1238    // SPI interface section. The PICO features two SPI HW blocks. We implement a simple SPI interface with a
1239    // a fixed set of SPI options for frequency, bit order and mode. One day this may change.
1240    //
1241    // ??? we do not take care of the chip select stuff. Expect to put a chip select / deselect around the calls?
1242    //-------------------------------------------------------------------------------------------------------
1243    uint8_t configureSPI ( uint8_t sclkPin , uint8_t mosiPin , uint8_t misoPin , uint32_t baudRate ) {
1244
1245        SPIInst *spi = nullptr;
1246
1247        if (( CdcSPI0.sclkPin == sclkPin ) &&
1248            ( CdcSPI0.mosiPin == mosiPin ) &&
1249            ( CdcSPI0.misoPin == misoPin )) {
1250
1251            spi = &CdcSPI0;
1252            spi -> spiHw = spi0;
1253
1254        } else if (( CdcSPI1.sclkPin == sclkPin ) &&
1255                    ( CdcSPI1.mosiPin == mosiPin ) &&
1256                    ( CdcSPI1.misoPin == misoPin ) &&
1257                    ( CdcSPI1.configured )) {
1258
1259            spi = &CdcSPI1;
1260            spi -> spiHw = spi1;
1261        }
1262        else return ( SPI_PORT_ERR );
1263
1264        spi -> mosiPin     = mosiPin ;
1265        spi -> misoPin     = misoPin ;
1266        spi -> sclkPin     = sclkPin ;
1267        spi -> frequency   = SPI_FREQUENCY ;
1268        spi -> configured  = true ;
1269        spi -> active      = false ;
1270
1271        spi_init ( spi -> spiHw , SPI_FREQUENCY );
1272
1273        spi_set_format ( spi -> spiHw ,       // SPI instance
1274                         8,                   // Number of bits per transfer
1275                         SPI_CPOL_1 ,         // Polarity (CPOL)
1276                         SPI_CPHA_1 ,         // Phase (CPHA)
1277                         SPI_MSB_FIRST );
1278
1279        gpio_set_function ( sclkPin , GPIO_FUNC_SPI );
1280        gpio_set_function ( mosiPin , GPIO_FUNC_SPI );
1281        gpio_set_function ( misoPin , GPIO_FUNC_SPI );
1282
1283        return ( NO_ERR );
1284    }
1285
1286    uint8_t  spiBeginTransaction ( uint8_t sclkPin , uint8_t csPin ) {
```

```
1287
1288        SPIInst *spi = nullptr;
1289
1290        if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
1291        else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
1292        else return ( SPI_PORT_ERR );
1293
1294        if ( spi -> active ) {
1295
1296            // ??? should we check who is active and just ignore when the same ? else "error " ?
1297
1298            return ( NO_ERR );
1299
1300        } else {
1301
1302            spi -> active     = true;
1303            spi -> selectPin  = csPin;
1304
1305            CDC::writeDio( csPin, false );
1306            return ( NO_ERR );
1307        }
1308 }
1309
1310 uint8_t spiEndTransaction( uint8_t sclkPin, uint8_t csPin ) {
1311
1312        SPIInst *spi = nullptr;
1313
1314        if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
1315        else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
1316        else return ( SPI_PORT_ERR );
1317
1318        if ( spi -> active ) {
1319
1320            // ??? check that this is the correct pin ?
1321
1322            CDC::writeDio( csPin, true );
1323
1324            spi -> active     = false;
1325            spi -> selectPin  = UNDEFINED_PIN;
1326
1327            return ( NO_ERR );
1328
1329        }
1330        else return ( NO_ERR ); // ???  "error "  not active...
1331 }
1332
1333 uint8_t spiRead( uint8_t sclkPin, uint8_t *buf, uint32_t len ) {
1334
1335        SPIInst *spi = nullptr;
1336
1337        if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
1338        else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
1339        else return ( SPI_PORT_ERR );
1340
1341        if ( spi -> active ) {
1342
1343            int bytesRead = spi_read_blocking( spi -> spiHw, 0, buf, len );
1344            return ( NO_ERR );
1345
1346        } else return ( NO_ERR ); // ??? fix : not active ...
1347 }
1348
1349 uint8_t spiWrite( uint8_t sclkPin, uint8_t *buf, uint32_t len ) {
1350
1351        SPIInst *spi = nullptr;
1352
1353        if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
1354        else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
1355        else return ( SPI_PORT_ERR );
1356
1357        if ( spi -> active ) {
1358
1359            spi_write_blocking( spi -> spiHw, buf, len );
1360            return ( NO_ERR );
1361
1362        } else return ( NO_ERR ); // ??? fix : not active ...
1363 }
1364
1365 //----------------------------------------------------------------------------------------------------------
1366 // Print out the Config Structure.
1367 //
1368 //----------------------------------------------------------------------------------------------------------
1369 void printConfigInfo( CdcConfigDesc *ci ) {
1370
1371        printf( "CDC Pin Configuration Info ( status %d ): \n", ci -> CFG_STATUS );
1372
1373        printf( "Pfail pin: %2d, ExtInt pin: %2d \n", ci -> PFAIL_PIN, ci -> EXT_INT_PIN );
1374
1375        printf( "ReadyLed pin: %2d, ActiveLed pin: %2d \n", ci -> READY_LED_PIN, ci -> ACTIVE_LED_PIN );
1376
1377        printf( "DIO pins ( 0 .. 7 ): %2d %2d %2d %2d %2d %2d %2d %2d\n",
1378                ci -> DIO_PIN_0, ci -> DIO_PIN_1, ci -> DIO_PIN_2, ci -> DIO_PIN_3,
1379                ci -> DIO_PIN_4, ci -> DIO_PIN_5, ci -> DIO_PIN_6, ci -> DIO_PIN_7 );
1380
1381        printf( "DIO pins ( 8 .. 15 ): %2d %2d %2d %2d %2d %2d %2d %2d\n",
1382                ci -> DIO_PIN_8, ci -> DIO_PIN_9, ci -> DIO_PIN_10, ci -> DIO_PIN_11,
1383                ci -> DIO_PIN_12, ci -> DIO_PIN_13, ci -> DIO_PIN_14, ci -> DIO_PIN_15 );
1384
1385        printf( "ADC pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
```

```
1386                   ci -> ADC_PIN_0 , ci -> ADC_PIN_1 , ci -> ADC_PIN_2 , ci -> ADC_PIN_3 );
1387
1388       printf( "PWM pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
1389                   ci -> PWM_PIN_0 , ci -> PWM_PIN_1 , ci -> PWM_PIN_2 , ci -> PWM_PIN_3 );
1390
1391       printf( "UART RX pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
1392                   ci -> UART_RX_PIN_0 , ci -> UART_RX_PIN_1 , ci -> UART_RX_PIN_2 , ci -> UART_RX_PIN_3 );
1393
1394       printf( "UART TX pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
1395                   ci -> UART_TX_PIN_0 , ci -> UART_TX_PIN_1 , ci -> UART_TX_PIN_2 , ci -> UART_TX_PIN_3 );
1396
1397       printf( "SPI0 Pins: MOSI: %2d, MISO: %2d, SCLK: %2d \n",
1398                   ci -> SPI_MOSI_PIN_0 , ci -> SPI_MISO_PIN_0 , ci -> SPI_SCLK_PIN_0 );
1399
1400       printf( "SPI1 Pins: MOSI: %2d, MISO: %2d, SCLK: %2d \n",
1401                   ci -> SPI_MOSI_PIN_1 , ci -> SPI_MISO_PIN_1 , ci -> SPI_SCLK_PIN_1 );
1402
1403       printf( "NVM I2C Pins: SCL: %2d, SDA: %2d, I2C Root: 0x%x \n",
1404                   ci -> NVM_I2C_SCL_PIN , ci -> NVM_I2C_SDA_PIN , ci -> NVM_I2C_ADR_ROOT );
1405
1406       printf( "EXT I2C Pins: SCL: %2d, SDA: %2d, I2C Root: 0x%x \n",
1407                   ci -> EXT_I2C_SCL_PIN , ci -> EXT_I2C_SDA_PIN , ci -> EXT_I2C_ADR_ROOT );
1408
1409       printf( "\n" );
1410
1411   }
1412
1413   }; // namespace CDC
```

First attempt ...

Base Station USB Handhelds

Layout El-
ements Gateways
Track Sensors Actors