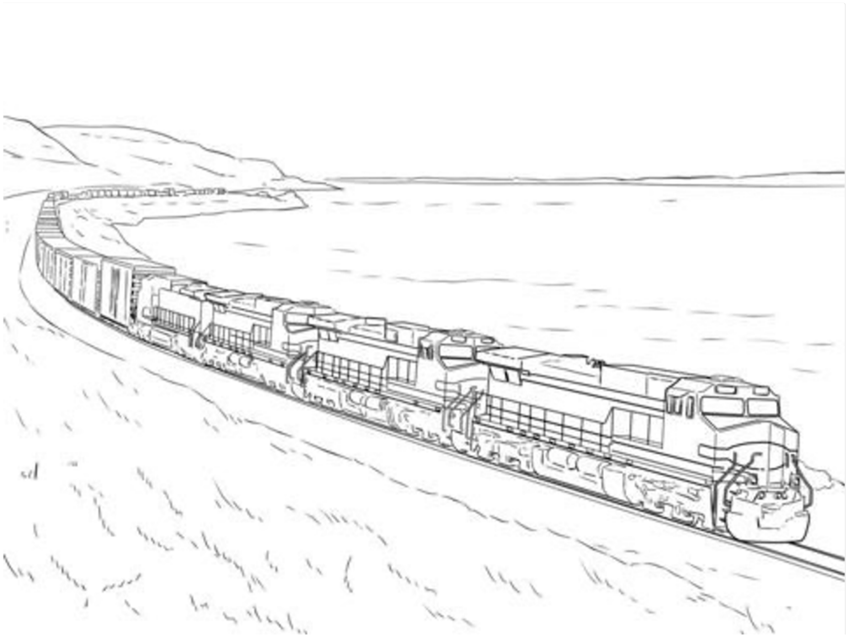


A Layout Control System for Model Railroads

Helmut Fieres

November 25, 2024

Contents

1 Introduction

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

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

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

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

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

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

1.1 Elements of a Layout Control System

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

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

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

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

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

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

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

Configuration as well as operations should be possible through sending the defined messages as well as a simple ASCII commands send to the base station which in turn generates the messages to broadcast via the common bus. A computer with a graphical UI would connect via the USB serial interface using the text commands.

1.2 Standards, Components and Compatibility

The DCC family of standards is the overall guiding standard. The layout system assumes the usage of DCC locomotive decoder equipped running gear and DCC stationary decoder accessories. Beyond this set of standards, it is not a requirement to be compatible with other model railroad electronic products and communication protocols. This does however not preclude gateways to interact in one form or another with such systems. 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 breakout boards. The book will however describe modules from the ground up and not use controller boards or shields. This way the principles are easier to see. The appendix section provides further information and links on how to build a system with some of the shelf parts instead of building it all yourself. With the concepts and software explained, it should not be a big issue to build your own mix of hardware and software.

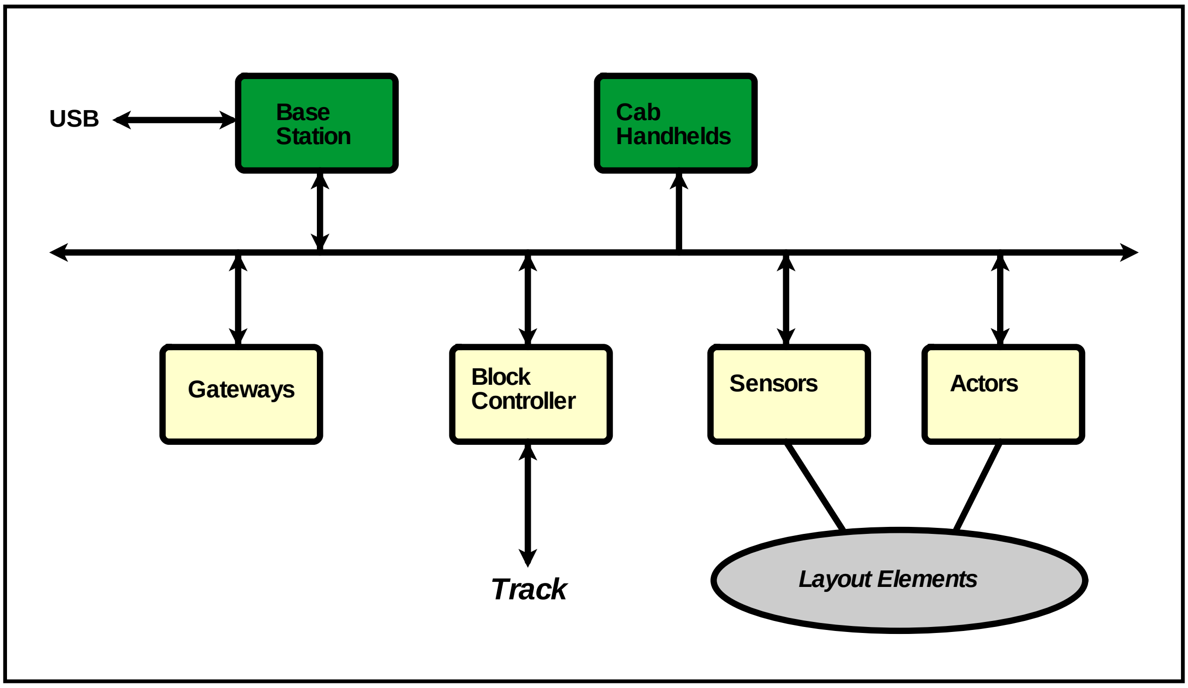
I have added most of the source files in the appendix for direct reference. They can also be found also on GitHub. (Note: still to do...) Every building block schematic shown was used and tested in one component or another. However, sometimes the book may not exactly match the material found on the web or be slightly different until the next revision is completed. Still, looking at portions of the source in the text explain quite well what it will do. As said, it is 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 variables. The base station node supports the decoder CV programming on a dedicated track with the `(REQ-CVS)`, `(REP-CVS)` and `(SET-CVS)` messages. The `(SET-CVM)` message supports setting a CV while the engine is on the main track. `(DCC-ERR)` is returned when an invalid operation is detected.

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

Table 3.9: 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

```
struct LcsPortMapEntry {

    uint16_t                  flags                        = 0;
    uint8_t                   portType                     = 0;

    uint16_t                  eventNodeId                  = NIL_NODE_ID;
    uint16_t                  eventId                      = NIL_EVENT_ID;
    uint8_t                   eventAction                  = PEA_EVENT_IDLE;
    uint16_t                  eventValue                   = 0;
    uint32_t                  eventTimeStamp               = 0L;
    uint8_t                   eventDelayTime               = 0;

    char                      portName[ MAX_NODE_NAME_SIZE ]          = { 0 };
    uint16_t                  portAttrMap[ MAX_PORT_ATTR_MAP_SIZE ]   = { 0 };
};
```

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.

```
struct LcsEventMapEntry {

    uint16_t    eventId;
    uint16_t    portId;
};
```

??? 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.

```

#include "LcsRuntimeLib.h"

// code to initialize HW structures ( to be discussed later... )
CDC::CdcConfigInfo cfg;

...

int rStat;

// ... create the LCS library object and startup
rStat = LCS::initRuntime( &cfg );

// ... register callbacks with lcsLib
// ... other node specific things to do
// ... invoke the run method. We never return from it ...
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.

```
int           rStat;
uint16_t      value;

rStat = LCS::nodeInfo( 0, NPI_PORT_MAP_ENTRIES, &value );
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.

```
int           rStat = ALL_OK;
uint16_t      value = 1;

rStat = LCS::nodeControl(  0, NPC_SET_READY_LED, &value );
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".

```
int       rStat       = ALL_OK;
uint16_t nodeId      = LCS::getNodeId( );
uint16_t eventId     = 200;

...

rStat = sendEventOn( nodeId, eventId );
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)`.

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

8.2 Node and Port Initialization Callback

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

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

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

8.3 Node and Port Request Reply Callback

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

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

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

8.4 Node and Port Control and Info Callback

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

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

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

8.5 Inbound Event Callback

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

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

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

8.6 Console Command Line Callback

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

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

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

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

8.7 DCC Message Callback

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

```
// ... the DCC message handler routine for DCC messages
void dccMsgHandler( uint8_t *msg ) { ... }
...
// during module firmware initialization ...
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.

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

8.9 LCS Periodic Task Callback

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

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

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

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

8.10 Summary

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

Wouldn't is be nice to have a simple interface to access the node data, set some options and simply test new hardware ? That is the subject of the next chapter.

9 RtLib Command Interface

??? explain the general concept ...

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

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

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

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

“‘ ¡lcs-command-char [arguments] ¿

“‘

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

9.1 Configuration Mode Commands

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

—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

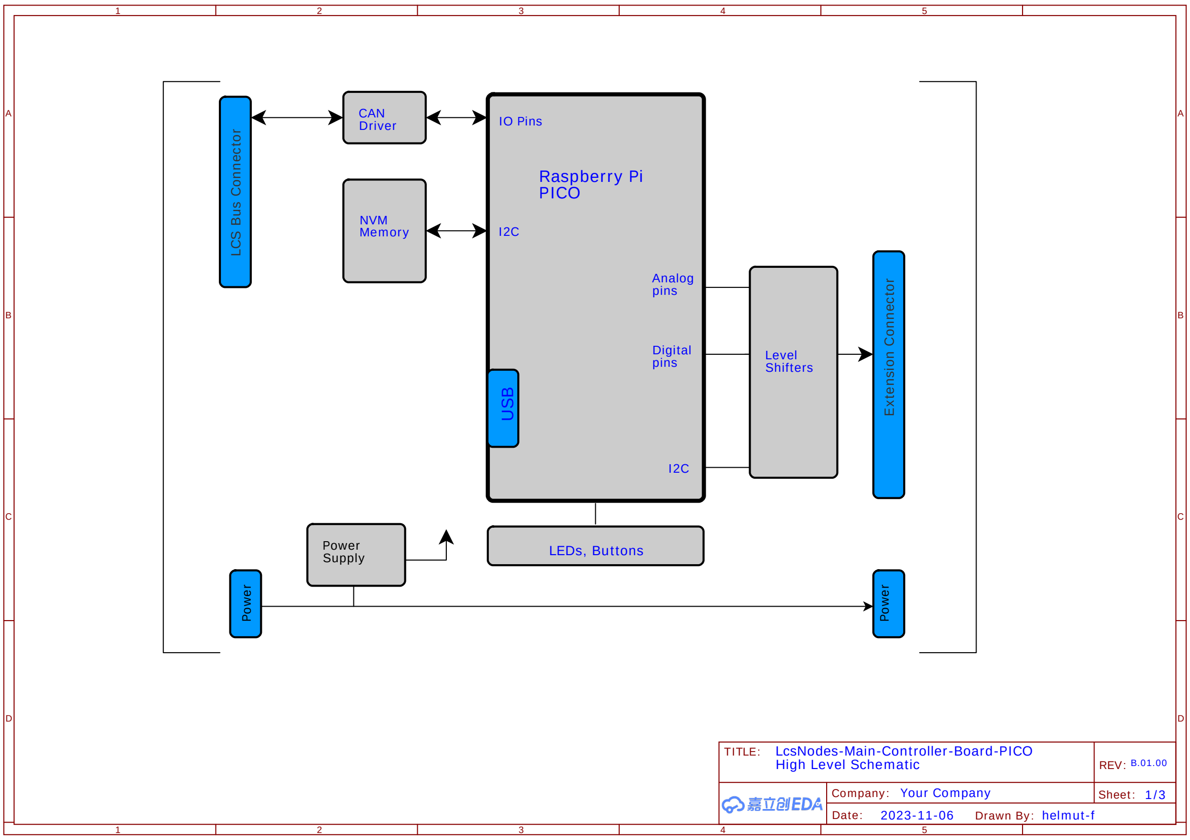
??? what is a good comprehensive example ?

11 Tests

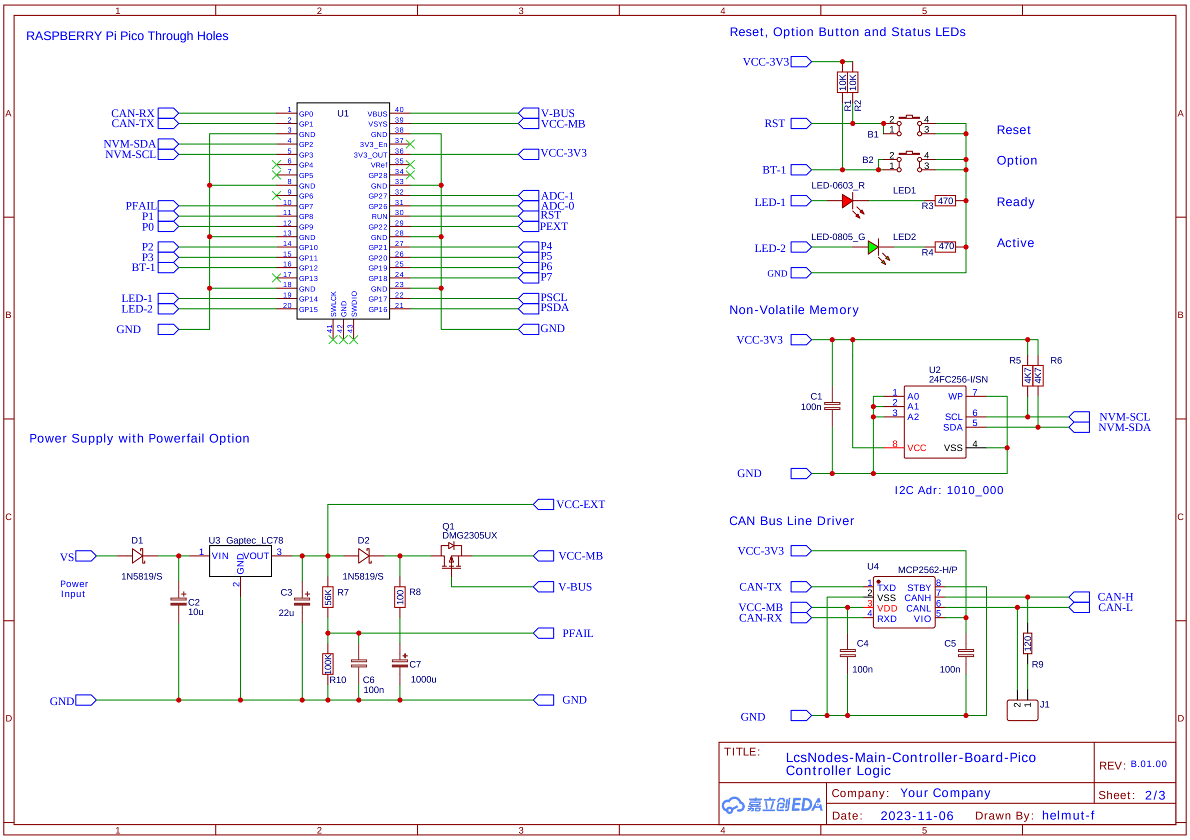
11.1 Schematics

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

11.1.1 part 1

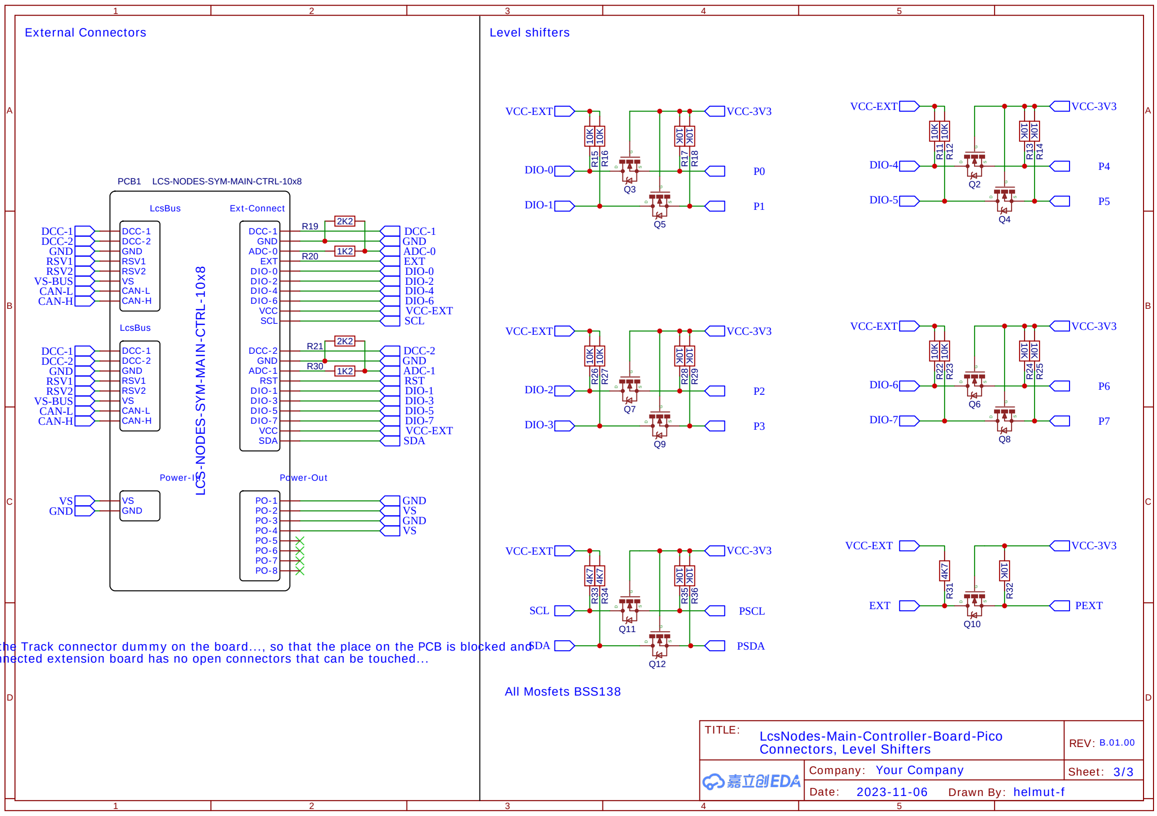


11.1.2 part 2



RASPBERRY Pi Pico Through Holes
Reset, Option Button and Status LEDs
Reset
Option
Ready
Active
Non-Volatile Memory
I2C Adr: 1010_000
Power Supply with Powerfail Option
CAN Bus Line Driver
TITLE: LcsNodes-Main-Controller-Board-Pico Controller Logic
REV: B.01.00
Company: Your Company
Sheet: 2/3
Date: 2023-11-06
Drawn By: helmut-f

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

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

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

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

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

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

};

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

    BASE_STATION_ERR_BASE              = 128,

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

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

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

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

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

```

struct DccPacket {

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

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

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

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

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

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

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

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

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

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

//------------------------------------------------------------------------------------------------------------
// The session map flags. The apply to all sessions in the session map. The initial values are copied from
// session option initial values.
//
//  SM_F_KEEP_ALIVE_CHECKING  - enable keep alive checking. When enabled, the locomotive session need to receive
//                              a keep alive LCS message periodically.
//  SM_F_ENABLE_REFRESH       - refresh the session data. This will send the locomotive speed and direction as
```

```
//                              well as the function flags periodically in a round robin processing of the
//
//------------------------------------------------------------------------------------------------------------
enum SessionMapFlags : uint16_t {

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

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

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

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

    // or use GET in all constants

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

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

    // thresholds

    // eventID to send for events ?

};

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

const uint16_t  MAX_CAB_SESSIONS                = 64;

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

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

//------------------------------------------------------------------------------------------------------------
// For creating the DCC track object, the track is described by the data structure below. In addition to the
// hardware pins enablePin, dcc1Pin1, dccPin2 and sensePin, there are the limits for current consumption
// values, all specified in milliAmps. The initial current sets the current consumption limit after the track
// is turned on. The limit current consumption specifies the actual configured value that is checked for a
// track current overload situation. The maximum current defines what current the power module should never
// exceed. For the measurements to work, the power module needs to deliver a voltage that corresponds to the
// current drawn on the track. The value is measured in milliVolt per Ampere drawn. Finally, there are
// threshold times for managing the track overload and restart capability.
//
//------------------------------------------------------------------------------------------------------------
struct LcsBaseStationTrackDesc {

    uint16_t   options                      = SM_OPT_DEFAULT_SETTING;

    uint8_t    enablePin                    = CDC::UNDEFINED_PIN;
```

```
    uint8_t   dccSigPin1                  = CDC::UNDEFINED_PIN;
    uint8_t   dccSigPin2                  = CDC::UNDEFINED_PIN;
    uint8_t   sensePin                    = CDC::UNDEFINED_PIN;
    uint8_t   uartRxPin                   = CDC::UNDEFINED_PIN;

    uint16_t  initCurrentMilliAmp         = 0;
    uint16_t  limitCurrentMilliAmp        = 0;
    uint16_t  maxCurrentMilliAmp          = 0;
    uint16_t  milliVoltPerAmp             = 0;

    uint16_t  startTimeThresholdMillis    = 0;
    uint16_t  stopTimeThresholdMillis     = 0;
    uint16_t  overloadTimeThresholdMillis = 0;
    uint16_t  overloadEventThreshold      = 0;
    uint16_t  overloadRestartThreshold    = 0;
};

//----------------------------------------------------------------------------------------------------------
// DCC track definition. The DCC track object is responsible for managing the track power as well as building
// and sending the DCC packet bit stream. A packet consists of the preamble bits, the postamble bit, the data
// bytes separated with a ZERO bit and a checksum byte. Creating the DCC bit stream is done with the signal
// generation routines. The signal state machine, running on a 29 microsecond tick, takes a DCC packet and
// gets it out to the track. The DCC signal state machine also invokes follow up actions that measure the
// actual power consumption, read in a railcom message and so on. There is also a DCC log facility which
// records internal events for testing and debugging.
//
// The other state machine will manage the actual track power. This machine is responsible for the periodic
// checking of power consumption and resulting power control. In contrast to the DCC signal state machine,
// this machine is not driven by a periodic interrupt but invoked periodically via the LCS runtime task
// manager.
//
// For a base station, there will be two track objects. One is the MAIN track and the other one is the PROG
// track. Each track has a DCC track object associated with it. In addition to the two track objects, there
// are class level static routines to manage the timer hardware functions, the analog signal read for current
// measurement and the serial IO for the optional RailCom message processing. The current version is AtMega
// specific.
//
//----------------------------------------------------------------------------------------------------------
struct LcsBaseStationDccTrack {

    public:

    LcsBaseStationDccTrack( );

    uint8_t                     setupDccTrack( LcsBaseStationTrackDesc* trackDesc );
    void                        loadPacket( const uint8_t *packet, uint8_t len, uint8_t repeat = 0 );

    uint16_t                    getFlags( );
    uint16_t                    getOptions( );

    bool                        isServiceModeOn( );
    void                        serviceModeOn( );
    void                        serviceModeOff( );

    void                        runDccTrackStateMachine( );
    void                        powerStart( );
    void                        powerStop( );
    bool                        isPowerOn( );
    bool                        isPowerOverload( );

    void                        cutoutOn( );
    void                        cutoutOff( );
    bool                        isCutoutOn( );

    void                        railComOn( );
    void                        railComOff( );
    bool                        isRailComOn( );

    void                        setLimitCurrent( uint16_t val );
    uint16_t                    getLimitCurrent( );
    uint16_t                    getActualCurrent( );
    uint16_t                    getInitCurrent( );
    uint16_t                    getMaxCurrent( );
    uint16_t                    getRMSCurrent( );

    uint16_t                    decoderAckBaseline( uint8_t resetPacketsToSend );
    bool                        decoderAckDetect( uint16_t baseValue, uint8_t retries );
    void                        checkOverload( );

    void                        runDccSignalStateMachine( volatile uint8_t *timeToInterrupt, uint8_t *followUpAction );

    void                        getNextBit( );
    void                        getNextPacket( );
    void                        powerMeasurement( );

    void                        startRailComIO( );
    void                        stopRailComIO( );
    uint8_t                     handleRailComMsg( );
    uint8_t                     getRailComMsg( uint8_t *buf, uint8_t bufLen );

    uint32_t                    getDccPacketsSend( );
    uint32_t                    getPwrSamplesTaken( );
    uint16_t                    getPwrSamplesPerSec( );

    void                        printDccTrackConfig( );
    void                        printDccTrackStatus( );

    void                        enableLog( bool arg );
    void                        beginLog( );
```

```
    void                        endLog( );
    void                        printLog( );

    void                        writeLogData( uint8_t id, uint8_t *buf, uint8_t len );
    void                        writeLogId( uint8_t id );
    void                        writeLogTs( );
    void                        writeLogVal( uint8_t valId, uint16_t val );

    private:

    uint16_t                    options                      = DT_OPT_DEFAULT_SETTING;
    volatile uint16_t           flags                        = DT_F_DEFAULT_SETTING;

    volatile uint8_t            trackState                   = 0;
    volatile uint8_t            signalState                  = 0;

    volatile uint32_t           trackTimeStamp               = 0;
    volatile uint8_t            overloadEventCount           = 0;
    volatile uint8_t            overloadRestartCount         = 0;

    uint8_t                     enablePin                    = CDC::UNDEFINED_PIN;
    uint8_t                     dccSigPin1                   = CDC::UNDEFINED_PIN;
    uint8_t                     dccSigPin2                   = CDC::UNDEFINED_PIN;
    uint8_t                     sensePin                     = CDC::UNDEFINED_PIN;
    uint8_t                     uartRxPin                    = CDC::UNDEFINED_PIN;

    uint16_t                    initCurrentMilliAmp          = 0;
    uint16_t                    limitCurrentMilliAmp         = 0;
    uint16_t                    maxCurrentMilliAmp           = 0;

    uint16_t                    startTimeThreshold           = 0;
    uint16_t                    stopTimeThreshold            = 0;
    uint16_t                    overloadTimeThreshold        = 0;
    uint16_t                    overloadEventThreshold       = 0;
    uint16_t                    overloadRestartThreshold     = 0;

    uint16_t                    milliVoltPerAmp              = 0;
    uint16_t                    digitsPerAmp                 = 0;
    volatile uint16_t           actualCurrentDigitValue      = 0;
    volatile uint16_t           highWaterMarkDigitValue      = 0;
    volatile uint16_t           limitCurrentDigitValue       = 0;
    uint16_t                    ackThresholdDigitValue       = 0;

    uint32_t                    totalPwrSamplesTaken         = 0;
    uint32_t                    lastPwrSampleTimeStamp       = 0;

    uint32_t                    lastPwrSamplePerSecTaken     = 0;
    uint32_t                    lastPwrSamplePerSecTimeStamp = 0;
    uint32_t                    pwrSamplesPerSec             = 0;

    uint8_t                     preambleLen                  = 0;
    uint8_t                     postambleLen                 = 0;
    volatile bool               currentBit                   = false;
    volatile uint8_t            bytesSent                    = 0;
    volatile uint8_t            bitsSent                     = 0;
    volatile uint8_t            preambleSent                 = 0;
    volatile uint8_t            postambleSent                = 0;
    uint32_t                    dccPacketsSend               = 0;

    DccPacket                   dccBuf1;
    DccPacket                   dccBuf2;
    DccPacket                   *activeBufPtr    = nullptr;
    DccPacket                   *pendingBufPtr   = nullptr;

    // ??? to add....
    // base station capabilities according to RCN200 - 4 16 bit words
    // sample values per second for samples and dcc packets
    // buffers for POM / XPOM data
    // queue for POM / XPOM commands

    uint8_t                     railComBufIndex                  = 0;
    uint8_t                     railComMsgBuf[ RAILCOM_BUF_SIZE ] = { 0 };

    uint8_t                     pwrSampleBufIndex                   = 0;
    uint16_t                    pwrSampleBuf[ PWR_SAMPLE_BUF_SIZE ]      = { 0 };

    public:

    static void                 startDccProcessing( );

};

//----------------------------------------------------------------------------------------------------------
// Every allocated loco session is described by the sessionMap structure. There are the engine cab Id, speed,
// direction and function information. There is also a field that indicates when we received information for
// this session from a cab control handheld. The function flags are stored in an array, each byte representing
// a group. Most of the fields are actually used for a DCC type locomotive. When the locomotive is an analog
// engine, only a subset of the fields is actually used. Nevertheless, even for an analog engine we will
// have a session. The base station will however not generate packets for this engine.
//
//----------------------------------------------------------------------------------------------------------
struct SessionMapEntry {

  uint16_t        flags               = SME_DEFAULT_SETTING;
  uint16_t        cabId               = LCS::NIL_CAB_ID;
  uint8_t         speed               = 0;
  uint8_t         speedSteps          = 128;
  uint8_t         direction           = 0;
  uint8_t         engineState         = 0;
```

```
  uint8_t         nextRefreshStep     = 0;
  unsigned long   lastKeepAliveTime   = 0;
  uint8_t         functions[ LCS::MAX_DCC_FUNC_GROUP_ID ] = { 0 };

};

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

  public:

    LcsBaseStationLocoSession( );

    uint8_t setupSessionMap(

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

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

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

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

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

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

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

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

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

    void                        printSessionMapConfig( );
    void                        printSessionMapInfo( );

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

    private:

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

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

    private:

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

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

    SessionMapEntry             *sessionMap             = nullptr;
    SessionMapEntry             *sessionMapNextRefresh  = nullptr;
    SessionMapEntry             *sessionMapHwm          = nullptr;
    SessionMapEntry             *sessionMapLimit        = nullptr;

};

//----------------------------------------------------------------------------------------------------------
// One of the key duties of the base station is to listen and react to DCC commands coming via the LCS bus.
// The interface works very closely with the session management and the two DCC track objects.
//
// ??? how about we make the handleLcsMsg handler a routine vs. an object ?
```

```
// ??? would make the any REQ/REP scheme easier ?
//-------------------------------------------------------------------------------------------------------
struct LcsBaseStationMsgInterface {

    public:

    LcsBaseStationMsgInterface( );

    uint8_t setupLcsMsgInterface( LcsBaseStationLocoSession   *locoSessions,
                                  LcsBaseStationDccTrack      *mainTrack,
                                  LcsBaseStationDccTrack      *progTrack
                                );

    void handleLcsMsg( uint8_t *msg );

    private:

    LcsBaseStationLocoSession   *locoSessions   = nullptr;
    LcsBaseStationDccTrack      *mainTrack      = nullptr;
    LcsBaseStationDccTrack      *progTrack      = nullptr;

};

//-------------------------------------------------------------------------------------------------------
// The base station implements a serial IO command interface. The command interface uses the DCC++ syntax of
// a command line and where it is a original DCC++ command it implements them in a compatible way. The idea
// is to one day connect to the programs of the JMRI world, which support the DCC++ style command interface.
//
//-------------------------------------------------------------------------------------------------------
struct LcsBaseStationCommand {

    public:

    LcsBaseStationCommand( );

    uint8_t setupSerialCommand( LcsBaseStationLocoSession  *locoSessions,
                                LcsBaseStationDccTrack     *mainTrack,
                                LcsBaseStationDccTrack     *progTrack );

    void handleSerialCommand( char *s );

    private:

    void openSessionCmd( char *s );
    void closeSessionCmd( char *s );

    void setThrottleCmd( char *s );
    void setFunctionBitCmd( char *s );
    void setFunctionGroupCmd( char *s );
    void emergencyStopCmd( );

    void readCVCmd( char *s );
    void writeCVByteCmd( char *s );
    void writeCVBitCmd( char *s );
    void writeCVByteMainCmd( char *s );
    void writeCVBitMainCmd( char *s );

    void writeDccPacketMainCmd( char *s );
    void writeDccPacketProgCmd( char *s );

    void setTrackOptionCmd( char *s );
    void turnPowerOnAllCmd( );
    void turnPowerOnMainCmd( );
    void turnPowerOnProgCmd( );
    void turnPowerOffAllCmd( );

    void printStatusCmd( char *s );
    void printTrackCurrentCmd( char *s );
    void printBaseStationConfigCmd( );
    void printHelpCmd( );
    void printVersionInfo( );
    void printConfiguration( );
    void printSessionMap( );
    void printTrackStatusMain( );
    void printTrackStatusProg( );

    void printDccLogCommand( char *s );

    private:

    LcsBaseStationLocoSession *locoSessions = nullptr;
    LcsBaseStationDccTrack    *mainTrack    = nullptr;
    LcsBaseStationDccTrack    *progTrack    = nullptr;
};

#endif
```

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

using namespace LCS;

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

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

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

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

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

    return ( ALL_OK );
}

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

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

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

        switch ( s[ charIndex ] ) {

            case '<': {

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

            } break;

            case '>': {

                switch ( cmdStr[ 0 ] ) {

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

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

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

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

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

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

                    case 'X': emergencyStopCmd( ); break;
                    case '0': turnPowerOffAllCmd( ); break;
                    case '1': turnPowerOnAllCmd( ); break;
                    case '2': turnPowerOnMainCmd( ); break;
                    case '3': turnPowerOnProgCmd( ); break;

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

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

                    case '?': printHelpCmd( ); break;

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

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

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

                charIndex ++;

            } break;

            default: {

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

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

    uint16_t  cabId = NIL_CAB_ID;
    uint8_t   sId   = 0;

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

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

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

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

    uint8_t sId = NIL_LOCO_SESSION_ID;

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

    int ret = locoSessions -> releaseSession( sId );

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

```

//----------------------------------------------------------------------------------------------------
// "setThrottleCmd" handles the throttle command. The original DCC++ interface uses both the register Id and
// the cabId. In the new version the sId is sufficient. But just to be compatible with the original
// DCC++ command, we also pass the cabId. It should be either zero or match the cabId in the allocated session.
//
//    <t sId cabId speed direction>
//
//    sId         -  the allocated session number.
//    cabId       -  the Cab Id. The number must match the can number in the session or be zero.
//    speed       -  throttle speed from 0-126, or -1 for emergency stop (resets SPEED to 0)
//    direction   -  the direction: 1=forward, 0=reverse. Setting direction when speed=0 only effects
//                   direction of cab lighting for a stopped train.
//
//    returns: <t sId speed direction >
//
//----------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::setThrottleCmd( char *s ) {

  uint8_t   sId         = NIL_LOCO_SESSION_ID;
  uint16_t  cabId       = NIL_CAB_ID;
  uint8_t   speed       = 0;
  uint8_t   direction   = 0;

  if ( sscanf( s, "%hhu %hu %hhu %hhu", &sId, &cabId, &speed, &direction ) != 4 ) return;
  if (( cabId != NIL_CAB_ID ) && ( locoSessions -> getSessionIdByCabId( cabId ) != sId )) return;

  locoSessions -> setThrottle( sId, speed, direction );

  printf( "<t %d %d %d>", sId, speed, direction );
}

//----------------------------------------------------------------------------------------------------
// "setFunctionBitCmd" turns on and off the engine decoder functions F0-F68 (F0 is sometimes called FL). This
// new command directly transmits the function setting to the engine decoder. The command interface is
// handling one function number at a time. The base station will handle the DCC byte generation.
//
//    <v sId funcId val >
//
//    sId     -  the allocated session number, from 1 to MAX_MAIN_REGISTERS.
//    funcId  -  the function number, currently implemented for F0 - F68.
//    val     -  the value to set, 1 or 0.
//
//    returns: NONE.
//
//----------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::setFunctionBitCmd( char *s ) {

    uint8_t sId = NIL_LOCO_SESSION_ID;
    uint8_t funcNum    = 0;
    uint8_t val        = 0;

    if ( sscanf( s, "%hhu %hhu %hhu", &sId, &funcNum, &val ) != 3 ) return;

    locoSessions -> setDccFunctionBit( sId, funcNum, val );
}

//----------------------------------------------------------------------------------------------------
// "setFunctionGroupCmd" sets the engine decoder functions F0-F68 by group byte using the DCC byte instruction
// format. The user needs to do the calculation as shown in the list below. This command directly transmits
// the command to the engine decoder. This function requires some user math, and is only there for the DCC++
// command interface compatibility.
//
//    <f cabId byte1 [ byte2 ] >
//
//    cabId         -  the cab number
//    byte1         -  see below for encoding
//    byte2         -  see below for encoding
//
//    returns: NONE
//
//    The DCC packet data for setting function groups is defined as follows:
//
//      Group 1:  F0, F4, F3, F2, F1       DCC Command Format: 100DDDDD
//      Group 2:  F8, F7, F6, F5           DCC Command Format: 1011DDDD
//      Group 3:  F12, F11, F10, F9        DCC Command Format: 1010DDDD
//      Group 4:  F20 .. F13               DCC Command Format: 0xDE DDDDDDDD
//      Group 5:  F28 .. F21               DCC Command Format: 0xDF DDDDDDDD
//      Group 6:  F36 .. F29               DCC Command Format: 0xD8 DDDDDDDD
//      Group 7:  F44 .. F37               DCC Command Format: 0xD9 DDDDDDDD
//      Group 8:  F52 .. F45               DCC Command Format: 0xDA DDDDDDDD
//      Group 9:  F60 .. F53               DCC Command Format: 0xDB DDDDDDDD
//      Group 10: F68 .. F61               DCC Command Format: 0xDC DDDDDDDD
//
//    To set functions F0-F4 on (=1) or off (=0):
//
//      BYTE1:  128 + F1*1 + F2*2 + F3*4 + F4*8 + F0*16
//      BYTE2:  omitted
//
//    To set functions F5-F8 on (=1) or off (=0):
//
//      BYTE1:  176 + F5*1 + F6*2 + F7*4 + F8*8
//      BYTE2:  omitted
//
//    To set functions F9-F12 on (=1) or off (=0):
//
//      BYTE1:  160 + F9*1 +F10*2 + F11*4 + F12*8
//      BYTE2:  omitted
//
```

```
//    For the remaining groups, the two byte format is used. Byte one is:
//
//          0xde ( 222 ) -> F13-F20
//          0xdf ( 223 ) -> F21-F28
//          0xd8 ( 216 ) -> F29-F36
//          0xd9 ( 217 ) -> F37-F44
//          0xda ( 218 ) -> F45-F52
//          0xdb ( 219 ) -> F53-F60
//          0xdc ( 220 ) -> F61-F68
//
//    Byte two with N being the starting group index is always:
//
//      BYTE2: (FN)*1 + (FN+1)*2 + (FN+2)*4 + (FN+3)*8 + (FN+4)*16 + (FN+5)*32 + (FN+6)*64 + (FN+7)*128
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::setFunctionGroupCmd( char *s ) {

  uint16_t  cabId   = NIL_CAB_ID;
  uint8_t   byte1   = 0;
  uint8_t   byte2   = 0;

    if ( sscanf( s, "%hu %hhu %hhu", &cabId, &byte1, &byte2 ) < 2 ) return;

    uint8_t sId = locoSessions -> getSessionIdByCabId( cabId );

    if ( sId == NIL_LOCO_SESSION_ID ) return;

    if (( byte2 == 0 ) && ( byte1 >= 128 ) && ( byte1 < 160 )) {

        locoSessions -> setDccFunctionGroup( sId, 1, byte1 );
    }
    else if (( byte2 == 0 ) && ( byte1 >= 160 ) && ( byte1 < 176 )) {

        locoSessions -> setDccFunctionGroup( sId, 3, byte1 );
    }
    else if (( byte2 == 0 ) && ( byte1 >= 176 ) && ( byte1 < 192 )) {

        locoSessions -> setDccFunctionGroup( sId, 2, byte1 );
    }
    else if ( byte1 == 0xde ) locoSessions -> setDccFunctionGroup( sId, 4, byte2 );
    else if ( byte1 == 0xdf ) locoSessions -> setDccFunctionGroup( sId, 5, byte2 );
    else if ( byte1 == 0xd8 ) locoSessions -> setDccFunctionGroup( sId, 6, byte2 );
    else if ( byte1 == 0xd9 ) locoSessions -> setDccFunctionGroup( sId, 7, byte2 );
    else if ( byte1 == 0xda ) locoSessions -> setDccFunctionGroup( sId, 8, byte2 );
    else if ( byte1 == 0xdb ) locoSessions -> setDccFunctionGroup( sId, 9, byte2 );
    else if ( byte1 == 0xdc ) locoSessions -> setDccFunctionGroup( sId, 10, byte2 );
}

//----------------------------------------------------------------------------------------------------------
// "readCVCmd" reads a configuration variable from the engine decoder on the programming track. The
// callbacknum and callbacksub parameter are ignored by the base station and just passed back to the caller
// for identification purposes.
//
//    <R cvId [ callbacknum callbacksub ]>
//
//    cvId          -  the configuration variable ID, 1 ... 1024.
//    callbacknum   -  a number echoed back, ignored by the base station
//    callbacksub   -  a number echoed back, ignored by the base station
//
//    returns: <R callbacknum|callbacksub|cvId value>
//
//    where value is 0 - 255 of the CV variable or -1 if the value could not be verified.
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::readCVCmd( char *s ) {

    uint16_t  cvId          = NIL_DCC_CV_ID;
    uint8_t   val           = 0;
    int       callbacknum   = 0;
    int       callbacksub   = 0;
    int       ret           = 0;

    if ( sscanf( s, "%hu %d %d", &cvId, &callbacknum, &callbacksub )  < 1 ) return;

    ret = locoSessions -> readCV( cvId, 0, &val );

    printf( "<R %d|%d|%d %d>", callbacknum, callbacksub, cvId, (( ret == ALL_OK ) ? val : -1 ));
}

//----------------------------------------------------------------------------------------------------------
// "writeCVByteCmd" writes a data byte to the engine decoder on the programming track and then verifies it.
// The callbacknum and callbacksub parameter are ignored by the base station and just passed back to the
// caller for identification purposes.
//
//    <W cvId val [ callbacknum callbacksub ]>
//
//    cvId          -  the configuration variable ID, 1 ... 1024.
//    val           -  the data byte.
//    callbacknum   -  a number echoed back, ignored by the base station
//    callbacksub   -  a number echoed back, ignored by the base station
//
//    returns: <W callbacknum|callbacksub|cvId Value>
//
//    where Value is 0 - 255 of the CV variable or -1 if the verification failed.
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::writeCVByteCmd( char *s ) {

    uint16_t  cvId          = NIL_DCC_CV_ID;
```

```
    uint8_t   val           = 0;
    int       callbacknum   = 0;
    int       callbacksub   = 0;
    int       ret           = 0;

    if ( sscanf( s, "%hu %hhu %d %d", &cvId, &val, &callbacknum, &callbacksub ) < 2 ) return;

    ret = locoSessions -> writeCVByte( cvId, val );

    printf( "<W %d|%d|%d %d>", callbacknum, callbacksub, cvId, (( ret == ALL_OK ) ? val : -1 ));
}

//------------------------------------------------------------------------------------------------------------
// "writeCVBitCmd" writes a bit to the engine decoder on the programming track and then verifies the
// operation. The callbacknum and callbacksub parameter are ignored by the base station and just passed back
// to the caller for identification purposes.
//
//    <B cvId bitPos bitVal callbacknum callbacksub>
//
//    cvId          -  the configuration variable ID, 1 ... 1024.
//    bitPos        -  the bit position of the bit, 0 .. 7.
//    bitVal        -  the data bit.
//    callbacknum   -  a number echoed back, ignored by the base station
//    callbacksub   -  a number echoed back, ignored by the base station
//
//    returns: <B callbacknum|callbacksub|cvId bitPos Value>
//
//    where Value is 0 or 1 of the bit or -1 if the verification failed.
//
//------------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::writeCVBitCmd( char *s ) {

    uint16_t  cvId          = NIL_DCC_CV_ID;
    uint8_t   bitPos        = 0;
    uint8_t   bitVal        = 0;
    int       callbacknum   = 0;
    int       callbacksub   = 0;
    int       ret           = 0;

    if ( sscanf( s, "%hu %hhu %hhu %d %d", &cvId, &bitPos, &bitVal, &callbacknum, &callbacksub ) != 5 ) return;

    ret = locoSessions -> writeCVBit( cvId, bitPos, bitVal );

    printf( "<B %d|%d|%d|%d %d>", callbacknum, callbacksub, cvId, bitPos, (( ret == ALL_OK ) ? bitVal : -1 ));
}

//------------------------------------------------------------------------------------------------------------
// "writeCVByteMainCmd" writes a data byte to the engine decoder on the main track, without any verification.
// To be compatible with the DCC++ command set, the command is using the cabId to identify the loco we talk
// about.
//
//    <w cabId cvId val >
//
//    cabId       -  the cabId number.
//    cvId        -  the configuration variable ID, 1 ... 1024.
//    val         -  the data byte.
//
//    returns: NONE
//
//------------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::writeCVByteMainCmd( char *s ) {

    uint16_t  cabId = NIL_CAB_ID;
    uint16_t  cvId  = NIL_DCC_CV_ID;
    uint8_t   val   = 0;

    if ( sscanf( s, "%hu %hu %hhu", &cabId, &cvId, &val ) != 3 ) return;

    locoSessions -> writeCVByteMain( locoSessions -> getSessionIdByCabId( cabId ), cvId, val );
}

//------------------------------------------------------------------------------------------------------------
// "writeCVBitMainCmd" writes a data byte to the engine decoder on the main track, without any verification.
// To be compatible with the DCC++ command set, the command is using the cabId to identify the loco we talk
// about.
//
//    <b cabId cvId bitPos bitVal >
//
//    cabId      -  the cabId number.
//    cvId       -  the configuration variable ID, 1 ... 1024.
//    bitPos     -  the bit position of the bit, 0 .. 7.
//    bitVal     -  the data bit.
//
//    returns: NONE
//
//------------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::writeCVBitMainCmd( char *s ) {

    uint16_t  cabId = NIL_CAB_ID;
    uint16_t  cvId    = NIL_DCC_CV_ID;
    uint8_t   bitPos = 0;
    uint8_t   bitVal = 0;

    if ( sscanf(s, "%hu %hu %hhu %hhu", &cabId, &cvId, &bitPos, &bitVal ) != 4 ) return;

    locoSessions -> writeCVBitMain( locoSessions -> getSessionIdByCabId( cabId ), cvId, bitPos, bitVal );
}

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

```
// "writeDccPacketMainCmd" writes a DCC packet to the main operations track. This is for testing and debugging
// and you better know the DCC packet standard by heart :-). The DCC standards define packets up to 15 data
// bytes payload.
//
//    <M byte1 byte2 [ byte3 ... byte10 ]>
//
//    byte1 .. byte10   - the packet data in hexadecimal
//
//    returns: NONE
//
//-----------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::writeDccPacketMainCmd( char *s ) {

    uint8_t b[ 16 ] = { 0 };
    uint8_t nBytes  = sscanf( s,
                                "%hhu %hhu %hhu %hhu %hhu %hhu %hhu %hhu"
                                "%hhu %hhu %hhu %hhu %hhu %hhu %hhu %hhu",
                                b, b + 1, b + 2, b + 3, b + 4, b + 5, b + 6, b + 7,
                                b + 8, b + 9, b + 10, b + 11, b + 12, b + 13, b + 14, b + 15 );

    if ( nBytes >= 3 && nBytes <= 10 ) locoSessions -> writeDccPacketMain( b, nBytes, 0 );
}

//-----------------------------------------------------------------------------------------------------------
// "writeDccPacketProgCmd" writes a DCC packet to the programming track. This is for testing and debugging and
// you better know the DCC packet standard by heart :-). The DCC standards define packets up to 15 data
// bytes payload.
//
///    <P byte1 byte2 [ byte3 ... byte10 ]>
//
//    byte1 .. byte10   - the packet data in hexadecimal
//
//    returns: NONE
//
//-----------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::writeDccPacketProgCmd( char *s ) {

    uint8_t b[ 16 ] = { 0 };
    uint8_t nBytes  = sscanf( s,
                                "%hhu %hhu %hhu %hhu %hhu %hhu %hhu %hhu"
                                "%hhu %hhu %hhu %hhu %hhu %hhu %hhu %hhu",
                                b, b + 1, b + 2, b + 3, b + 4, b + 5, b + 6, b + 7,
                                b + 8, b + 9, b + 10, b + 11, b + 12, b + 13, b + 14, b + 15 );

    if ( nBytes >= 3 && nBytes <= 10 ) locoSessions -> writeDccPacketProg( b, nBytes, 0 );
}

//-----------------------------------------------------------------------------------------------------------
// "emergencyStopCmd" handles the emergencyStop command. This new command causes the base station to send out
// the emergency stop broadcast DCC command.
//
//    <X>
//
//    returns: <X>
//
//-----------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::emergencyStopCmd( ) {

    locoSessions -> emergencyStopAll( );
    printf( "<X>" );
}

//-----------------------------------------------------------------------------------------------------------
// "turnPowerOnXXX" and "turnPowerOff" enables/disables the main and/or the programming track.
//
//    <0> - turn operations and programming track power off
//    <1> - turn operations and programming track power on
//    <2> - turn operations track power on
//    <3> - turn programming track power on
//
//-----------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::turnPowerOnAllCmd( ) {

    mainTrack -> powerStart( );
    progTrack -> powerStart( );
    printf( "<p1>" );
}

void LcsBaseStationCommand::turnPowerOffAllCmd( ) {

    mainTrack -> powerStop( );
    progTrack -> powerStop( );
    printf( "<p0>" );
}

void LcsBaseStationCommand::turnPowerOnMainCmd( ) {

    mainTrack -> powerStart( );
    printf( "<p1 MAIN>" );
}

void LcsBaseStationCommand::turnPowerOnProgCmd( ) {

    progTrack -> powerStart( );
    printf( "<p1 PROG>" );
}

//-----------------------------------------------------------------------------------------------------------
// "setTrackOptionCmd" turns on and off capabilities of the operations or service track.
```

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

    uint8_t option = 0;

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

        switch ( option ) {

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

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

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

    uint8_t opt = 0;

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

        switch ( opt ) {

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

            case 9: {

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

            } break;

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

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

    printConfiguration( );
}

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

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

```
}

//--------------------------------------------------------------------------------------------------------------
// "printVersionInfo" list out the Arduino type and software version of this program.
//
//--------------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::printVersionInfo( ) {

    printf( "<\nLCS Base Station / Version: tbd / %s %s >\n", __DATE__, __TIME__  );
}

//--------------------------------------------------------------------------------------------------------------
// "printSessionMap" list out the active session table content.
//
//--------------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::printSessionMap( ) {

    locoSessions -> printSessionMapInfo( );
}

//--------------------------------------------------------------------------------------------------------------
// "printTrackStatusMain" lists out the current MAIN track status
//
//--------------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::printTrackStatusMain( ) {

    mainTrack -> printDccTrackStatus( );
}

//--------------------------------------------------------------------------------------------------------------
// "printTrackStatusProg" lists out the current PROG track status
//
//--------------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::printTrackStatusProg( ) {

    progTrack -> printDccTrackStatus( );
}

//--------------------------------------------------------------------------------------------------------------
// "printTrackCurrentCmd" reads the actual current being drawn on the main operations track.
//
//    <a [ track ]>
//
// where "track" == 0 or omitted is the MAIN track, "track" == 1 is the PROG track.
//
//    returns: <a current>, where current is the actual power consumption in milliAmps.
//
//--------------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::printTrackCurrentCmd( char *s ) {

    int opt = -1;

    sscanf( s, "%d", &opt );

    printf( "<a " );

    switch ( opt ) {

        case 0: printf( "%d", mainTrack -> getActualCurrent( )); break;
        case 1: printf( "%d", progTrack -> getActualCurrent( )); break;
        case 2: printf( "%d %d", mainTrack -> getActualCurrent( ), progTrack -> getActualCurrent( )); break;

        case 10: printf( "%d", mainTrack -> getRMSCurrent( )); break;
        case 11: printf( "%d", progTrack -> getRMSCurrent( )); break;
        case 12: printf( "%d %d", mainTrack -> getRMSCurrent( ), progTrack -> getRMSCurrent( )); break;

        default: printf( "%d", mainTrack -> getRMSCurrent( ));
    }

    printf( ">" );
}

//--------------------------------------------------------------------------------------------------------------
// "printDccLogCommandCommand" is the command to manage the DCC log for tracing and debugging purposes.
//
//    <Y [ opt ]> where "opt" is the command to execute from the DCC Log function.
//
//      Main track:
//
//      0 - disable DCC logging
//      1 - enable DCC logging
//      2 - start DCC logging
//      3 - stop DCC logging
//      4 - list log entries
//
//      Prog track:
//
//      10 - disable DCC logging
//      11 - enable DCC logging
//      12 - start DCC logging
//      13 - stop DCC logging
//      14 - list log entries
//
//      RailCom:
//
//      20 - show real time RailCom buffer, experimental
//
//--------------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::printDccLogCommand( char *s ) {
```

```

    int opt = -1;

    sscanf( s, "%d", &opt );

    printf( "<Y %d ", opt );

    switch ( opt ) {

        case 0:     mainTrack -> enableLog( false );  break;
        case 1:     mainTrack -> enableLog( true );   break;
        case 2:     mainTrack -> beginLog( );         break;
        case 3:     mainTrack -> endLog( );           break;
        case 4:     mainTrack -> printLog( );         break;

        case 10:    progTrack -> enableLog( false );  break;
        case 11:    progTrack -> enableLog( true );   break;
        case 12:    progTrack -> beginLog( );         break;
        case 13:    progTrack -> endLog( );           break;
        case 14:    progTrack -> printLog( );         break;

        case 20: {

            uint8_t buf[ 16 ];

            mainTrack -> getRailComMsg( buf, sizeof( buf ));

            printf( "RC: " );
            for ( uint8_t i = 0; i < 8; i++ ) printf( "0x%x ", buf[ i ]);

        } break;

        default: ;
    }

    printf( ">" );
}

//----------------------------------------------------------------------------------------------------------
// "printHelp" lists a short version of all the command.
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationCommand::printHelpCmd( ) {

    printf( "\nCommands:\n" );

    printf( "<O cabId>                                 - allocate a session for the cab\n" );
    printf( "<K sId>                                   - release a session\n" );
    printf( "<t sId cabId speed dir>                   - set cab speed / direction\n" );
    printf( "<f cabId funcId val >                     - set cab function value, group DCC format\n" );
    printf( "<v sId funcId val >                       - set cab function value, individual\n" );
    printf( "<R cvId callbacknum callbacksub >         - read CV byte\n" );
    printf( "<W cvId val callbacknum callbacksub>      - write CV byte on programming track\n" );
    printf( "<B cvId bitPos bitVal callbacknum callbacksub> - write CV bit on programming track\n" );
    printf( "<w cabId cvId val > - write CV byte on operations track\n" );
    printf( "<b cabId cvId bitPos bitVal > - write CV bit on operations track\n" );
    printf( "<M sId byte1 byte2 [ byte3 ... byte10 ]> - send DCC packet on operations track to Reg n\n" );
    printf( "<P sId byte1 byte2 [ byte3 ... byte10 ]> - send DCC packet on programming track to Reg n\n" );

    printf( "<C track [option] - set track option, track = 0 -> MAIN, track = 1 -> PROG\n" );
    printf( "            " " - 1  - set main track cutout on\n" );
    printf( "            " " - 2  - set main track cutout off\n" );
    printf( "            " " - 3  - set main track RailCom on\n" );
    printf( "            " " - 4  - set main track RailCom off\n" );
    printf( "            " " - 10 - set prog track in operations mode\n" );
    printf( "            " " - 11 - set prog track in service mode\n" );

    printf( "<X>  - emergency stop all\n" );

    printf( "<0> - turn operations and programming track power off\n" );
    printf( "<1> - turn operations and programming track power on\n" );
    printf( "<2> - turn operations track power on\n" );
    printf( "<3> - turn programming track power on\n" );

    printf( "<a [ opt ]>  " " - list current consumption, default is RMS for MAIN\n" );
    printf( "             " "   - opt 0  - actual - MAIN\n" );
    printf( "             " "   - opt 1  - actual - PROG\n" );
    printf( "             " "   - opt 2  - actual - both\n" );
    printf( "             " "   - opt 10 - RMS - MAIN\n" );
    printf( "             " "   - opt 11 - RMS - PROG\n" );
    printf( "             " "   - opt 12 - RMS - both\n" );

    printf( "<C <option>> - turn on/off the Railcom option on the main track( 0 - off, 1 - on)\n" );

    printf( "<s [ level ]>" " - list status at detail level, default is summary\n" );
    printf( "             " "   - level 0 - summary\n" );
    printf( "             " "   - level 1 - configuration\n" );
    printf( "             " "   - level 2 - session map\n" );
    printf( "             " "   - level 3 - main track current\n" );
    printf( "             " "   - level 4 - prog track current\n" );
    printf( "             " "   - level 9 - all of the above\n" );

    printf( "<S> - list base station configuration\n" );
    printf( "<L> - list base station session table" );

    printf( "<Y [ opt ]> - DCC log options ( used for debugging and tracing )\n" );
    printf( "             " "   - 0  - disable main track logging\n" );
    printf( "             " "   - 1  - enable main track logging\n" );
    printf( "             " "   - 2  - begin main track logging\n" );
```

```
    printf( "             " "   - 3  - end main track logging\n" );
    printf( "             " "   - 4  - print main track logging data\n" );
    printf( "             " "   - 10 - disable prog track logging\n" );
    printf( "             " "   - 11 - enable prog track logging\n" );
    printf( "             " "   - 12 - begin prog track logging\n" );
    printf( "             " "   - 13 - end prog track logging\n" );
    printf( "             " "   - 14 - print prog track logging data\n" );

    printf( "<?> - list this help\n" );

    printf( "\n" );
}
```

```
//-------------------------------------------------------------------------------------------------------
//
// LCS Base Station - DCC Track - implementation file
//
//-------------------------------------------------------------------------------------------------------
// The DCC track object is one of the the key objects for the DCC subsystem. It is responsible for the DCC
// track signal generation and the power management functions. There will be exactly two objects of this kind,
// one for the MAIN track and the other for the PROG track. The DCC track object has two major functional
// parts. The first is to transmit a DCC packet to the track. This is the most important task, as with no
// packets no power is on the tracks and the locomotive will not work. The second task is to continuously
// monitor the current consumption. Finally, for the RailCom option, the cutout generation and receiving
// of the RailCOm packets is handled.
//
//-------------------------------------------------------------------------------------------------------
//
// LCS - Base Station DCC Track implementation file
// Copyright (C) 2019 - 2024  Helmut Fieres
//
// This program is free software: you can redistribute it and/or modify it under the terms of the GNU General
// Public License as published by the Free Software Foundation, either version 3 of the License, or (at your
// option) any later version.
//
// This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
// implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
// for more details.
//
// You should have received a copy of the GNU General Public License along with this program. If not, see
// http://www.gnu.org/licenses
//
//  GNU General Public License:  http://opensource.org/licenses/GPL-3.0
//
//-------------------------------------------------------------------------------------------------------
#include "LcsBaseStation.h"
#include <math.h>

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

//-------------------------------------------------------------------------------------------------------
// DCC Signal debugging. A tick is defined to last 29 microseconds. There is a debugging option to set the
// clock much slower so that the waveform can be seen.
//
// ??? take out, we are past that ...... since a long time. -> one last check than out ...
//-------------------------------------------------------------------------------------------------------

#define DEBUG_WAVE_FORM 0

#if DEBUG_WAVE_FORM == 1
#define TICK_IN_MICROSECONDS  400000
#else
#define TICK_IN_MICROSECONDS  29
#endif

//-------------------------------------------------------------------------------------------------------
// The DccTrack Object local definitions. The DCC track object is a bit special. There are exactly two object
// instances created, MAIN and PROG. Both however share the global mechanism for generating the DCC hardware
// signals. There are callback functions for the DCC timer and the serial I/O capability for the RailCom
// feature. The hardware lower layers can be found in controller dependent code (CDC) layer.
//
//-------------------------------------------------------------------------------------------------------
namespace {

using namespace LCS;

//-------------------------------------------------------------------------------------------------------
// The DCC Track will allocate two DCC Track Objects. For the interrupt system to work, references to the
// objects must be static variables. The initialization sequence outside of this class will allocate the two
// objects and we keep a copy of the respective DCC track object created right here.
//
// ??? when we use the global variables in the "main" file, can this go away ?
 //-------------------------------------------------------------------------------------------------------
LcsBaseStationDccTrack  *mainTrack  = nullptr;
LcsBaseStationDccTrack  *progTrack  = nullptr;

//-------------------------------------------------------------------------------------------------------
// DCC packet definitions. A DCC packet payload is at most 15 bytes long, excluding the checksum byte. This
// is true for XPOM and DCC-A support, otherwise it is according to NMRA up to 6 bytes. The preamble is a
// series of "ONE" bits, which helps the decoders to sync to the bit stream. The standard specifies a
// minimum of 16 ONE bits for the MAIN track and 22 ONE bits for the PROG track. The postamble is exactly
// one "ONE" bit. If the cutout period option is enabled, the cutout overlays the first ONE bits the
// preamble.
//
//-------------------------------------------------------------------------------------------------------
const uint8_t   MAIN_PACKET_PREAMBLE_LEN    = 17;
const uint8_t   MAIN_PACKET_POSTAMBLE_LEN   = 1;
const uint8_t   PROG_PACKET_PREAMBLE_LEN    = 22;
const uint8_t   PROG_PACKET_POSTAMBLE_LEN   = 1;
const uint8_t   DCC_PACKET_CUTOUT_LEN       = 4;
const uint8_t   MIN_DCC_PACKET_SIZE         = 2;
const uint8_t   MAX_DCC_PACKET_SIZE         = 16;
const uint8_t   MIN_DCC_PACKET_REPEATS      = 0;
const uint8_t   MAX_DCC_PACKET_REPEATS      = 8;
const uint8_t   RAILCOM_BUFFER_SIZE         = 8;

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

```
// Constant values definition. We need the RESET and IDLE packet as well as a bit mask for a quick bit
// select in the data byte.
//
//----------------------------------------------------------------------------------------------------
DccPacket       idleDccPacket       = { 3, 0, { 0xFF, 0x00, 0xFF }};
DccPacket       resetDccPacket      = { 3, 0, { 0x00, 0x00, 0x00 }};
const uint8_t   bitMask9[ ]         = { 0x00, 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01 };

//----------------------------------------------------------------------------------------------------
// Programming decoders require to detect a short rise in power consumption. The value is at least 60mA,
// but decoders can raise anything from 100mA to 250mA. This is a bit touchy and the value set to 100mA
// was done after testing several decoders. Still, a bit flaky ...
//
//----------------------------------------------------------------------------------------------------
const uint8_t ACK_TRESHOLD_VAL      = 100;

//----------------------------------------------------------------------------------------------------
// The DCC signal generator thinks in ticks. With a DCC ONE based on 58 microseconds and a DCC ZERO based
// on 116 microseconds half period, we define a tick as a 29 microsecond interval. Although, ONE and ZERO
// bit signals could be implemented using a multiple of 58 microseconds, the cutout function requires a
// signal length of 29 microseconds at the beginning of the period, right after the packet end bit of the
// previous packet. Luckily 2 * 29 is 58, 2 * 58 is 116. Perfect for DCC packets.
//
// ??? think directly in microseconds ?
//----------------------------------------------------------------------------------------------------
const uint32_t TICKS_29_MICROS              =  1;
const uint32_t TICKS_58_MICROS              =  TICKS_29_MICROS * 2;
const uint32_t TICKS_116_MICROS             =  TICKS_29_MICROS * 4;
const uint32_t TICKS_CUTOUT_MICROS          =  TICKS_29_MICROS * 16;

//------------------------------------------------------------------------------------------------------
// Base Station global limits. Perhaps to move to a configurable place...
//
//-----------------------------------------------------------------------------------------------------
const uint16_t MILLI_VOLT_PER_DIGIT         = 5;
const uint16_t MILLI_VOLT_PER_AMP           = 1500;

//----------------------------------------------------------------------------------------------------
// DCC track power management is also a a state machine managing the state of the power track. Maximum values
// for the DCC track power start and stop sequence as well as limits for power overload events are defined.
// We also define reasonable default values.
//
//----------------------------------------------------------------------------------------------------
const uint16_t MAX_START_TIME_THRESHOLD_MILLIS      = 2000;
const uint16_t MAX_STOP_TIME_THRESHOLD_MILLIS       = 1000;
const uint16_t MAX_OVERLOAD_TIME_THRESHOLD_MILLIS   = 500;
const uint16_t MAX_OVERLOAD_EVENT_COUNT             = 10;
const uint16_t MAX_OVERLOAD_RESTART_COUNT           = 10;

const uint16_t DEF_START_TIME_THRESHOLD_MILLIS      = 1000;
const uint16_t DEF_STOP_TIME_THRESHOLD_MILLIS       = 500;
const uint16_t DEF_OVERLOAD_TIME_THRESHOLD_MILLIS   = 300;
const uint16_t DEF_OVERLOAD_EVENT_COUNT             = 10;
const uint16_t DEF_OVERLOAD_RESTART_COUNT           = 10;

//----------------------------------------------------------------------------------------------------
// Track state machine state definitions. See the track state machine routine for an explanation of the
// individual states.
//
//----------------------------------------------------------------------------------------------------
enum DccTrackState : uint8_t {

    DCC_TRACK_POWER_OFF       = 0,
    DCC_TRACK_POWER_ON        = 1,
    DCC_TRACK_POWER_OVERLOAD  = 2,
    DCC_TRACK_POWER_START1    = 3,
    DCC_TRACK_POWER_START2    = 4,
    DCC_TRACK_POWER_STOP1     = 5,
    DCC_TRACK_POWER_STOP2     = 6
};

//----------------------------------------------------------------------------------------------------
// DCC Track signal state machine states. See the DCC signal state machine routine for an explanation of
// the states.
//
//----------------------------------------------------------------------------------------------------
enum DccSignalState : uint8_t {

    DCC_SIG_CUTOUT_START      = 0,
    DCC_SIG_CUTOUT_1          = 1,
    DCC_SIG_CUTOUT_2          = 2,
    DCC_SIG_CUTOUT_3          = 3,
    DCC_SIG_CUTOUT_END        = 4,
    DCC_SIG_START_BIT         = 5,
    DCC_SIG_TEST_BIT          = 6,
    DCC_SIG_ZERO_SECOND_HALF  = 7
};

// ??? idea: each state has a number of ticks it will set. Have an array where to get this value and just
// set it from the table...
//
uint8_t ticksForState[ ] = {

    TICKS_29_MICROS,        // DCC_SIG_CUTOUT_START
    TICKS_CUTOUT_MICROS,    // DCC_SIG_CUTOUT_1
    TICKS_29_MICROS,        // DCC_SIG_CUTOUT_2
    TICKS_58_MICROS,        // DCC_SIG_CUTOUT_3
    TICKS_58_MICROS,        // DCC_SIG_CUTOUT_END
    TICKS_58_MICROS,        // DCC_SIG_START_BIT
```

```
    TICKS_58_MICROS,        // DCC_TEST_BIT,
    TICKS_116_MICROS        // DCC_SIG_ZERO_SECOND_HALF
};

//-----------------------------------------------------------------------------------------------------
// DCC Track signal state machine follow up request items. The signal state machine first sets the hardware
// signal for both tracks and then determines whether a follow up action is required. See the track state
// machine routine for an explanation of the individual follow up actions.
//
//-----------------------------------------------------------------------------------------------------
enum DccSignalStateFollowup : uint8_t {

    DCC_SIG_FOLLOW_UP_NONE                 = 0,
    DCC_SIG_FOLLOW_UP_GET_BIT              = 1,
    DCC_SIG_FOLLOW_UP_GET_PACKET           = 2,
    DCC_SIG_FOLLOW_UP_MEASURE_CURRENT      = 3,
    DCC_SIG_FOLLOW_UP_START_RAILCOM_IO     = 4,
    DCC_SIG_FOLLOW_UP_STOP_RAILCOM_IO      = 5,
    DCC_SIG_FOLLOW_UP_RAILCOM_MSG          = 6,
};

//-----------------------------------------------------------------------------------------------------
// The hardware timer needs to be set to the ticks we want to pass before interrupting again. There are
// three things to remember between interrupts. First, the current time interval, which tells us how many
// ticks will have passed when the timer interrupts again. Next, for each DCC track signal state we need to
// remember how many ticks are left before the state machine needs to run again. Each time the timer will
// interrupt, the passed ticks are subtracted from the ticks left counters. When the counter becomes zero,
// the state machine for the track will run.
//
//-----------------------------------------------------------------------------------------------------
volatile uint8_t timeToInterrupt    = 0;
volatile uint8_t timeLeftMainTrack  = 0;
volatile uint8_t timeLeftProgTrack  = 0;

//-----------------------------------------------------------------------------------------------------
// The DCC track object maintains an internal log facility for test and debugging purposes. During operation
// a set of log entries can be recorded to a log buffer. A log entry consist of the header byte, which
// contains in the first byte the 4-bit log id and the 4-bit length of the log data. A log entry can therefore
// record up to 16 bytes of payload.
//
//-----------------------------------------------------------------------------------------------------
enum LogId : uint8_t {

    LOG_NIL        = 0,
    LOG_BEGIN      = 1,
    LOG_END        = 2,
    LOG_TSTAMP     = 3,
    LOG_DCC_IDL    = 4,
    LOG_DCC_RST    = 5,
    LOG_DCC_PKT    = 6,
    LOG_DCC_RCM    = 7,
    LOG_VAL        = 8,
    LOG_INV        = 15
};

//-----------------------------------------------------------------------------------------------------
// The log buffer and the log index. When writing to the log buffer, the index will always point to the
// next available position. Once the buffer is full, no further data can be added.
//
//-----------------------------------------------------------------------------------------------------
const uint16_t  LOG_BUF_SIZE            = 4096;

bool            logEnabled              = false;
bool            logActive               = false;
uint16_t        logBufIndex             = 0;
uint8_t         logBuf[ LOG_BUF_SIZE ]  = { 0 };

//-----------------------------------------------------------------------------------------------------
// RailCom decoder table. The Railcom communication will send raw bytes where only four bits are "one" in
// a byte ( hamming weight 4 ). The first two bytes are labelled "channel1" and the remaining six bytes
// are labelled "channel2". The actual data is then encode using the table below. Each raw byte will be
// translated to a 6 bits of data for the datagram to assemble. In total there are therefore a maximum
// of 48bits that are transmitted in a railcom message.
//
//-----------------------------------------------------------------------------------------------------
enum RailComDataBytes : uint8_t {

    INV   = 0xff,
    BUSY  = 0xfe,
    ACK   = 0xfd,
    NACK  = 0xfc,
    RSV1  = 0xfa,
    RSV2  = 0xf9,
    RSV3  = 0xf8
};

const uint8_t railComDecode[256] = {

    INV,    INV,    INV,    INV,    INV,    INV,    INV,    INV,    // 0
    INV,    INV,    INV,    INV,    INV,    INV,    INV,    ACK,

    INV,    INV,    INV,    INV,    INV,    INV,    INV,    0x33,   // 1
    INV,    INV,    INV,    0x34,   INV,    0x35,   0x36,   INV,

    INV,    INV,    INV,    INV,    INV,    INV,    INV,    0x3A,   // 2
    INV,    INV,    INV,    0x3B,   INV,    0x3C,   0x37,   INV,

    INV,    INV,    INV,    0x3F,   INV,    0x3D,   0x38,   INV,    // 3
    INV,    0x3E,   0x39,   INV,    NACK,   INV,    INV,    INV,
```

```
    INV,   INV,   INV,   INV,   INV,   INV,   INV,   0x24,  // 4
    INV,   INV,   INV,   0x23,  INV,   0x22,  0x21,  INV,

    INV,   INV,   INV,   0x1F,  INV,   0x1E,  0x20,  INV,   // 5
    INV,   0x1D,  0x1C,  INV,   0x1B,  INV,   INV,   INV,

    INV,   INV,   INV,   0x19,  INV,   0x18,  0x1A,  INV,   // 6
    INV,   0x17,  0x16,  INV,   0x15,  INV,   INV,   INV,

    INV,   0x25,  0x14,  INV,   0x13,  INV,   INV,   INV,   // 7
    0x32,  INV,   INV,   INV,   INV,   INV,   INV,   INV,

    INV,   INV,   INV,   INV,   INV,   INV,   INV,   RSV2,  // 8
    INV,   INV,   INV,   0x0E,  INV,   0x0D,  0x0C,  INV,

    INV,   INV,   INV,   0x0A,  INV,   0x09,  0x0B,  INV,   // 9
    INV,   0x08,  0x07,  INV,   0x06,  INV,   INV,   INV,

    INV,   INV,   INV,   0x04,  INV,   0x03,  0x05,  INV,   // a
    INV,   0x02,  0x01,  INV,   0x00,  INV,   INV,   INV,

    INV,   0x0F,  0x10,  INV,   0x11,  INV,   INV,   INV,   // b
    0x12,  INV,   INV,   INV,   INV,   INV,   INV,   INV,

    INV,   INV,   INV,   RSV1,  INV,   0x2B,  0x30,  INV,   // c
    INV,   0x2A,  0x2F,  INV,   0x31,  INV,   INV,   INV,

    INV,   0x29,  0x2E,  INV,   0x2D,  INV,   INV,   INV,   // d
    0x2C,  INV,   INV,   INV,   INV,   INV,   INV,   INV,

    INV,   RSV3,  0x28,  INV,   0x27,  INV,   INV,   INV,   // e
    0x26,  INV,   INV,   INV,   INV,   INV,   INV,   INV,

    ACK,   INV,   INV,   INV,   INV,   INV,   INV,   INV,   // f
    INV,   INV,   INV,   INV,   INV,   INV,   INV,   INV,
};

//----------------------------------------------------------------------------------------------------------
// Railcom datagrams are sent from a mobile or a stationary decoder.
//
//----------------------------------------------------------------------------------------------------------
enum railComDatagramType : uint8_t {

    RX_DG_TYPE_UNDEFINED  = 0,
    RC_DG_TYPE_MOB        = 1,
    RC_DG_TYPE_STAT       = 2
};

//----------------------------------------------------------------------------------------------------------
// Each mobile decoder railcom datagram will start with an ID field of four bits. Channel one will use only
// the ADR_HIG and ADR_LOW Ids. All IDs can be used for channel 2. Since decoders answer on channel one
// for each DCC packet they receive, here is a good chance that channel 1 will contains nonsense data. This
// is different for channel two, where only the addressed decoder explicitly answers. To decide whether
// a railcom message is valid, you should perhaps ignore channel 1 data and just check channel 2 for this
// purpose. A RC datagram starts with the 4-bit ID and an 8 to 32bit payload.
//
//      RC_DG_MOB_ID_POM       ( 0  )  - 12bit
//      RC_DG_MOB_ID_ADR_HIGH  ( 1  )  - 12bit
//      RC_DG_MOB_ID_ADR_LOW   ( 2  )  - 12bit
//      RC_DG_MOB_ID_APP_EXT   ( 3  )  - 18bit
//      RC_DG_MOB_ID_APP_DYN   ( 7  )  - 18bit
//      RC_DG_MOB_ID_XPOM_1    ( 8  )  - 36bit
//      RC_DG_MOB_ID_XPOM_2    ( 9  )  - 36bit
//      RC_DG_MOB_ID_XPOM_3    ( 10 )  - 36bit
//      RC_DG_MOB_ID_XPOM_4    ( 11 )  - 36bit
//      RC_DG_MOB_ID_TEST      ( 12 )  - ignore
//      RC_DG_MOB_ID_SEARCH    ( 14 )  - 48bit
//
// A datagram with the ID 14 is a DDC-A datagram and all 8 datagram bytes are combined to an 48bit datagram.
// A datagram packet can also contain more than one datagram. For example there could be two 18-bit length
// datagram in one packet or 3 12-bit packets and so on. Finally, unused bytes in channel two could contain
// an ACK to fill them up.
//
//----------------------------------------------------------------------------------------------------------
enum railComDatagramMobId : uint8_t {

    RC_DG_MOB_ID_POM        = 0,
    RC_DG_MOB_ID_ADR_HIGH   = 1,
    RC_DG_MOB_ID_ADR_LOW    = 2,
    RC_DG_MOB_ID_APP_EXT    = 3,
    RC_DG_MOB__IDAPP_DYN    = 7,
    RC_DG_MOB_ID_XPOM_1     = 8,
    RC_DG_MOB_ID_XPOM_2     = 9,
    RC_DG_MOB_ID_XPOM_3     = 10,
    RC_DG_MOB_ID_XPOM_4     = 11,
    RC_DG_MOB_ID_TEST       = 12,
    RC_DG_MOB_ID_SEARCH     = 14
};

//----------------------------------------------------------------------------------------------------------
// Similar to the mobile decode, a stationary decoder datagram will start an ID field of four bits. Stationary
// decoders also define a datagram with "SRQ" and no ID field to request service from the base station.
//
// ??? to fill in ...
//
//      RC_DG_STAT_ID_SRQ       ( 0  )  - 12bit
//      RC_DG_STAT_ID_POM       ( 1  )  - 12bit
//      RC_DG_STAT_ID_STAT1     ( 4  )  - 12bit
```

```
//      RC_DG_STAT_ID_TIME      ( 5  )  - xxbit
//      RC_DG_STAT_ID_ERR       ( 6  )  - xxbit
//      RC_DG_STAT_ID_XPOM_1    ( 8  )  - 36bit
//      RC_DG_STAT_ID_XPOM_2    ( 9  )  - 36bit
//      RC_DG_STAT_ID_XPOM_3    ( 10 )  - 36bit
//      RC_DG_STAT_ID_XPOM_4    ( 11 )  - 36bit
//      RC_DG_STAT_ID_TEST      ( 12 )  - ignore
//
//----------------------------------------------------------------------------------------------------------
enum railComDatagramStatId : uint8_t {

    RC_DG_STAT_ID_SRQ       = 0,
    RC_DG_STAT_ID_POM       = 1,
    RC_DG_STAT_ID_STAT1     = 4,
    RC_DG_STAT_ID_TIME      = 5,
    RC_DG_STAT_ID_ERR       = 6,
    RC_DG_STAT_ID_DYN       = 7,
    RC_DG_STAT_ID_XPOM_1    = 8,
    RC_DG_STAT_ID_XPOM_2    = 9,
    RC_DG_STAT_ID_XPOM_3    = 10,
    RC_DG_STAT_ID_XPOM_4    = 11,
    RC_DG_STAT_ID_TEST      = 12
};

//----------------------------------------------------------------------------------------------------------
// Utility routine for number range checks.
//
//----------------------------------------------------------------------------------------------------------
bool isInRangeU( uint8_t val, uint8_t lower, uint8_t upper ) {

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

//----------------------------------------------------------------------------------------------------------
// Utility function to map a DCC address to a railcom decoder type.
//
//----------------------------------------------------------------------------------------------------------
inline uint8_t mapDccAdrToRailComDatagramType( uint16_t adr ) {

    if      (( adr >= 1 )   && ( adr <= 127 ))  return ( RC_DG_TYPE_MOB );
    else if (( adr >= 128 ) && ( adr <= 191 ))  return ( RC_DG_TYPE_STAT );
    else if (( adr >= 192 ) && ( adr <= 231 ))  return ( RC_DG_TYPE_MOB );
    else                                        return ( RX_DG_TYPE_UNDEFINED );
}

//----------------------------------------------------------------------------------------------------------
// Conversion functions between milliAmps and digit values as report4de by the analog to digital converter
// hardware. For a better precision, the formula uses 32 bit computation and stores the result back in a
// 16 bit quantity.
//
//----------------------------------------------------------------------------------------------------------
uint16_t milliAmpToDigitValue( uint16_t milliAmp, uint16_t digitsPerAmp ) {

    #if 0
    uint32_t mA = milliAmp;
    uint32_t dPA = digitsPerAmp;
    return (( uint16_t ) ( mA * dPA / 1000 ));
    #endif

    return ((uint16_t) ((((uint32_t) milliAmp ) * ((uint32_t) digitsPerAmp )) / 1000 ));
}

uint16_t digitValueToMilliAmp( uint16_t digitValue, uint16_t digitsPerAmp ) {

    #if 0
    uint32_t dV = digitValue;
    uint32_t dPA = digitsPerAmp;
    return ((uint16_t)( dV * 1000 / dPA ));
    #endif

    return ((uint16_t) ((((uint32_t) digitValue ) * 1000 ) / ((uint32_t) digitsPerAmp )));
}

//----------------------------------------------------------------------------------------------------------
// The DccTrack timer interrupt handler routine implements the heartbeat of the DCC system. The two DCC
// track signal generators state machines MAIN and PROG use the same timer interrupt handler. Upon the timer
// interrupt, we first will update the time left counters. If a counter falls to zero, the signal state
// machine for that track will run and set the DCC signal levels. The state machine returns the next time
// interval it expects to be called again and a possible follow up action code. After handling both state
// machines, the timer is set to the smaller new remaining minimum time interval of both state machines.
// This is the time when the next state machine in one of the  signal generators needs to run. It is
// important to always have the timer running, so we keep decrementing the ticks to interrupt values.
//
// If a state machine determined that it needs to do some more elaborate action, the interrupt handler runs
// part two of its work. This split allows to run the time sensitive signal level settings first and any
// actions, such as getting the next packet, after both signal generator signal settings have been processed.
// Follow up actions are getting the next bit value to transmit, the next packet to send, a power consumption
// measurement and Railcom message processing. As we do not have all time in the world, these follow up
// actions still should be brief. The state machine carefully selects the spot for requesting such follow up
// actions in the DCC bit stream.
//
// The timer interrupt routine and all it calls runs with interrupts disabled. As said, better be quick.
// Top priority is to fetch the next bit and the next packet. Next is the Railcom processing if enabled. If
// there are power consumption measurement follow up actions, they are run last. Since the ADC converter
// hardware serializes the analog measurements, we will only do one measurement and drop the other. MAIN
// always has the higher priority.
//
// For the MAIN track with cutout enabled, the entry and exit of that cutout is a 29us timer call. That is
// awfully short and no follow-up action is scheduled there. All other intervals are either 58us or 116us
```

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

    uint8_t followUpMain = DCC_SIG_FOLLOW_UP_NONE;
    uint8_t followUpProg = DCC_SIG_FOLLOW_UP_NONE;

    timeLeftMainTrack -= timeToInterrupt;
    timeLeftProgTrack -= timeToInterrupt;

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

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

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

    CDC::setRepeatingTimerLimit( timeToInterrupt * TICK_IN_MICROSECONDS );

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

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

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

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

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

} // timerCallback

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

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

    CDC::startRepeatingTimer( TICK_IN_MICROSECONDS );
}

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

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

void printLogVal( uint16_t index ) {

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

void printLogData( uint16_t index, uint8_t len ) {

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

uint8_t printLogEntry( uint16_t index ) {

    if ( index < LOG_BUF_SIZE ) {

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

        switch ( logEntryId ) {

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

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

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

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

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

    if ( logActive ) {

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

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

void writeLogId( uint8_t id ) {

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

void writeLogTs( ) {

    if ( logActive ) {

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

void writeLogVal( uint8_t valId, uint16_t val ) {

    if ( logActive ) {

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

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

    logEnabled = arg;
    logActive  = false;
}

void beginLog( ) {

    if ( logEnabled ) {

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

void endLog( ) {

    if ( logActive ) {

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

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

```
// A simple routine to print out the log data, one entry on one line.
//
// ??? what is exactly the stop condition ? The END entry having a length of zero ?
//---------------------------------------------------------------------------------------------------------------
void printLog( ) {

    if ( logEnabled ) {

        if ( ! logActive ) {

            if ( logBufIndex > 0 ) {

                printf( "\n" );

                uint16_t entryIndex  = 0;
                uint8_t  entryLen    = 0;

                while ( entryIndex < logBufIndex ) {

                    entryLen = printLogEntry( entryIndex );
                    printf( "\n" );

                    if ( entryLen > 0 ) entryIndex += entryLen;
                    else                break;
                }
            }
            else printf( "DCC Log Buf: Nothing recorded\n" );
        }
        else printf( "DCC Log Active\n" );
    }
    else printf( "DCC Log disabled\n" );
}

}; // namespace


//===============================================================================================================
//===============================================================================================================
//
// Object part.
//
//===============================================================================================================
//===============================================================================================================


//---------------------------------------------------------------------------------------------------------------
// "startDccProcessing" will kick off the DCC timer for the track signal processing. The idea is that the
// program first creates all the DCC track objects, does whatever else needs to be initialized and then starts
// the signal generation with this routine.
//
//---------------------------------------------------------------------------------------------------------------
void LcsBaseStationDccTrack::startDccProcessing( ) {

    initDccTrackProcessing( );
}

//---------------------------------------------------------------------------------------------------------------
// Object instance section. The DccTrack constructor. Nothing to do so far.
//
//---------------------------------------------------------------------------------------------------------------
LcsBaseStationDccTrack::LcsBaseStationDccTrack( ) { }

//---------------------------------------------------------------------------------------------------------------
// "setupDccTrack" performs the setup tasks for the DCC track.  We will configure the hardware, the DCC
// packet options such as preamble and postamble length, the initial state machine state current consumption
// limit and load the initial packet into the active buffer. There is quite a list of parameters and options
// that can be set. This routine does the following checking:
//
//    - the pins used in the CDC layer must be a pair ( for atmega controllers ).
//    - the sensePin must be an analog input pin.
//    - if the track is a service track, cutout and RailCom are not supported.
//    - if RailCom is set, Cutout must be set too.
//    - the initial current limit consumption setting must be less than the current limit setting.
//    - the current limit setting must be less than the maximum current limit setting.
//
// Once the DCC track object is initialized, the last thing to do is to remember the object instance in the
// file static variables. This is necessary for the interrupt handlers to work. If any of the checks fails,
// the flag field will have the error bit set.
//
//---------------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationDccTrack::setupDccTrack( LcsBaseStationTrackDesc* trackDesc ) {

    if ((  trackDesc -> enablePin   == CDC::UNDEFINED_PIN ) ||
        (  trackDesc -> dccSigPin1  == CDC::UNDEFINED_PIN ) ||
        (  trackDesc -> dccSigPin2  == CDC::UNDEFINED_PIN ) ||
        (  trackDesc -> sensePin    == CDC::UNDEFINED_PIN )) {

        flags = DT_F_CONFIG_ERROR;
        return ( ERR_DCC_PIN_CONFIG );
    }

    if ((( trackDesc -> options & DT_OPT_SERVICE_MODE_TRACK ) && ( trackDesc -> options & DT_OPT_CUTOUT ))    ||
        (( trackDesc -> options & DT_OPT_SERVICE_MODE_TRACK ) && ( trackDesc -> options & DT_OPT_RAILCOM ))   ||
        (( trackDesc -> options & DT_OPT_RAILCOM ) && ( ! ( trackDesc -> options & DT_OPT_CUTOUT )))          ||
        ( trackDesc -> initCurrentMilliAmp  > trackDesc -> limitCurrentMilliAmp )                             ||
        ( trackDesc -> limitCurrentMilliAmp > trackDesc -> maxCurrentMilliAmp )                               ||
        ( trackDesc -> startTimeThresholdMillis > MAX_START_TIME_THRESHOLD_MILLIS )                           ||
        ( trackDesc -> stopTimeThresholdMillis > MAX_STOP_TIME_THRESHOLD_MILLIS )                             ||
        ( trackDesc -> overloadTimeThresholdMillis > MAX_OVERLOAD_TIME_THRESHOLD_MILLIS )                     ||
```

```
        ( trackDesc -> overloadEventThreshold > MAX_OVERLOAD_EVENT_COUNT )                              ||
        ( trackDesc -> overloadRestartThreshold > MAX_OVERLOAD_RESTART_COUNT )
        ) {

        flags = DT_F_CONFIG_ERROR;
        return ( ERR_DCC_TRACK_CONFIG );
    }

    signalState                = DCC_SIG_START_BIT;
    trackState                 = DCC_TRACK_POWER_OFF;
    flags                      = DT_F_DEFAULT_SETTING;
    options                    = trackDesc -> options;
    enablePin                  = trackDesc -> enablePin;
    dccSigPin1                 = trackDesc -> dccSigPin1;
    dccSigPin2                 = trackDesc -> dccSigPin2;
    sensePin                   = trackDesc -> sensePin;
    uartRxPin                  = trackDesc -> uartRxPin;
    initCurrentMilliAmp        = trackDesc -> initCurrentMilliAmp;
    limitCurrentMilliAmp       = trackDesc -> limitCurrentMilliAmp;
    maxCurrentMilliAmp         = trackDesc -> maxCurrentMilliAmp;
    startTimeThreshold         = trackDesc -> startTimeThresholdMillis;
    stopTimeThreshold          = trackDesc -> stopTimeThresholdMillis;
    overloadTimeThreshold      = trackDesc -> overloadTimeThresholdMillis;
    overloadEventThreshold     = trackDesc -> overloadEventThreshold;
    overloadRestartThreshold   = trackDesc -> overloadRestartThreshold;

    // ??? MILLI_VOLT_PER_DIGIT is actually 4,72V / 1024 = 4,6 mV. How to make this more precise ?

    milliVoltPerAmp            = trackDesc -> milliVoltPerAmp;
    digitsPerAmp               = milliVoltPerAmp / MILLI_VOLT_PER_DIGIT;

    limitCurrentDigitValue     = milliAmpToDigitValue( initCurrentMilliAmp, digitsPerAmp );
    ackThresholdDigitValue     = milliAmpToDigitValue( ACK_TRESHOLD_VAL, digitsPerAmp );
    actualCurrentDigitValue    = 0;
    dccPacketsSend             = 0;
    totalPwrSamplesTaken       = 0;
    lastPwrSamplePerSecTaken   = 0;
    pwrSamplesPerSec           = 0;

    CDC::configureDio( enablePin, CDC::OUT );
    CDC::configureDio( dccSigPin1, CDC::OUT );
    CDC::configureDio( dccSigPin2, CDC::OUT );
    CDC::configureAdc( sensePin );

    CDC::writeDio( enablePin, false );
    CDC::writeDioPair( dccSigPin1, false, dccSigPin2, false );

    CDC::onTimerEvent( timerCallback );

    if ( options & DT_OPT_SERVICE_MODE_TRACK ) {

        progTrack      =   this;
        preambleLen    =   PROG_PACKET_PREAMBLE_LEN;
        postambleLen   =   PROG_PACKET_POSTAMBLE_LEN;
        flags          |=  DT_F_SERVICE_MODE_ON;
        activeBufPtr   = &resetDccPacket;
        pendingBufPtr  = &dccBuf1;
    }
    else {

        mainTrack      =   this;
        preambleLen    =   MAIN_PACKET_PREAMBLE_LEN;
        postambleLen   =   MAIN_PACKET_POSTAMBLE_LEN;
        activeBufPtr   = &idleDccPacket;
        pendingBufPtr  = &dccBuf1;
    }

    if ( trackDesc -> options & DT_OPT_CUTOUT ) {

        preambleLen =  MAIN_PACKET_PREAMBLE_LEN - DCC_PACKET_CUTOUT_LEN;
        flags       |= DT_F_CUTOUT_MODE_ON;
        signalState =  DCC_SIG_CUTOUT_START;
    }

    if ( trackDesc -> options & DT_OPT_RAILCOM ) {

        flags |= DT_F_RAILCOM_MODE_ON;
        if ( CDC::configureUart( uartRxPin, CDC::UNDEFINED_PIN, 250000, CDC::UART_MODE_8N1 ) != ALL_OK ) {

            flags = DT_F_CONFIG_ERROR;
            return ( ERR_DCC_TRACK_CONFIG );
        }
    }

    return ( ALL_OK );
}

//----------------------------------------------------------------------------------------------------------
// DCC signal generation is done through a state machine that is invoked when the DCC timer interrupts. The
// interrupt timer thinks in multiples of 29us, which we will just call a "tick" in the description below. It
// runs as part of the timer interrupt handler, so we need to be short and quick. First, the HW signals are
// set. This keeps the track signals in their timing. Next, the new signal state, time to run again and any
// other follow up action of this invocation are set. The idea is to separate HW signal generation and follow
// up actions. The timer interrupt handler will first call both state machines, MAIN and PROG, and then work
// on the optional follow-up actions. The state machine has the following states:
//
// DCC_SIG_CUTOUT_START: if the cutout option is on, a new DCC packet starts with this signal state. The
// DCC signal goes HIGH for one tick and the signal state advances to signal state DCC_SIG_CUTOUT_1.
//
```

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

    volatile uint8_t  *timeToInterrupt,
    uint8_t           *followUpAction

    ) {

    switch ( signalState ) {

        case DCC_SIG_CUTOUT_START: {

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

        } break;

        case DCC_SIG_CUTOUT_1: {

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

        } break;

        case DCC_SIG_CUTOUT_2: {

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

        } break;

        case DCC_SIG_CUTOUT_3: {

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

            if ( flags & DT_F_RAILCOM_MODE_ON ) {

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

        } break;
```

```
        case DCC_SIG_CUTOUT_END: {

            CDC::writeDioPair( dccSigPin1, false, dccSigPin2, true );
            *timeToInterrupt  = TICKS_58_MICROS;
            *followUpAction   = (( flags & DT_F_RAILCOM_MODE_ON ) ?
                                DCC_SIG_FOLLOW_UP_RAILCOM_MSG : DCC_SIG_FOLLOW_UP_NONE );
            signalState       = DCC_SIG_START_BIT;

        } break;

        case DCC_SIG_START_BIT: {

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

        } break;

        case DCC_SIG_TEST_BIT: {

            if ( currentBit ) {

                CDC::writeDioPair( dccSigPin1, false, dccSigPin2, true );

                if ( postambleSent >= postambleLen ) {

                    *followUpAction = DCC_SIG_FOLLOW_UP_GET_PACKET;
                    signalState     = (( flags & DT_F_CUTOUT_MODE_ON ) ? DCC_SIG_CUTOUT_START : DCC_SIG_START_BIT );
                }
                else {

                    *followUpAction = DCC_SIG_FOLLOW_UP_NONE;
                    signalState     = DCC_SIG_START_BIT;
                }
            }
            else {

                *followUpAction   = (( bitsSent == 0 ) ? DCC_SIG_FOLLOW_UP_MEASURE_CURRENT :  DCC_SIG_FOLLOW_UP_NONE );
                signalState       = DCC_SIG_ZERO_SECOND_HALF;
            }

            *timeToInterrupt  = TICKS_58_MICROS;

        } break;

        case DCC_SIG_ZERO_SECOND_HALF: {

            CDC::writeDioPair( dccSigPin1, false, dccSigPin2, true );
            *timeToInterrupt  = TICKS_116_MICROS;
            *followUpAction   = DCC_SIG_FOLLOW_UP_NONE;
            signalState       = DCC_SIG_START_BIT;

        } break;

        default: {

            *followUpAction   = DCC_SIG_FOLLOW_UP_NONE;
            *timeToInterrupt  = TICKS_58_MICROS;
        }
    }
}

//------------------------------------------------------------------------------------------------------------
// The "getNextBit" routine works through the active packet buffer bit for bit. A packet consists of the
// optional cutout sequence, the preamble bits, the data bytes separated by a ZERO bit and the postamble bits.
// The cutout option, the preamble and postamble are configured at DCC track object init time. The preamble
// length is different for MAIN and PROG tracks with the the cutout period overlaid at the beginning of the
// preamble. The postamble is currently always just one HIGH bit, according to standard.
//
// The routine works first through the preamble bit count, then through the data byte bits, and finally
// through the postamble bits. The bits to select from the data byte is done with a 9-bit mask. Remember that
// the first bit to send is the data byte separator, which is always a zero. We run from 0 to 8 through the
// bit mask, the first bit being the ZERO bit.
//
//------------------------------------------------------------------------------------------------------------
void LcsBaseStationDccTrack::getNextBit( ) {

    if ( preambleSent < preambleLen ) {

        currentBit = true;
        preambleSent ++;
    }
    else if ( bytesSent < activeBufPtr -> len ) {

        currentBit = activeBufPtr -> buf[ bytesSent ] & bitMask9[ bitsSent ];
        bitsSent ++;

        if ( bitsSent == 9 ) {

            bytesSent ++;
            bitsSent = 0;
        }
    }
    else if ( postambleSent < postambleLen ) {

        currentBit = true;
        postambleSent ++;
```

```
    }
}

//-----------------------------------------------------------------------------------------------------------
// If all bits of a packet have been processed, the next packet will be determined during the last ONE bit
// transmission of the postamble. If there is a non-zero repeat count on the current packet, the same packet
// is sent again until the repeat count drops to zero. On a zero repeat count, we check if there is a pending
// packet. If so, it is copied to the active buffer and the pending flag is reset. This signals anyone waiting,
// that the next packet can be queued. If there is no pending packet, we still need to keep the track going and
// will load an IDLE or RESET packet.
//
// For non-service mode packets, there is a requirement that a decoder should not be receive two consecutive
// packets. The standards talks about 5 milliseconds between two packets to the same decoder. For now, we will
// not do anything special. A decoder will most likely, if there is more than one decoder active, not be
// addressed in two consecutive packets, simply because the session refresh mechanism will go round robin
// through the session list. However, if there is only one decoder active, two packets will be sent in a
// row, but the decoders are robust enough to ignore this fact. Better run more than one loco :-).
//
// This routine is the central place to submit a DCC packet to the track and therefore a good place to write
// a DCC_LOG record. We distinguish between a RESET, an IDLE and a data packet. Note that these records will
// only be written when DCC logging is enabled.
//
//-----------------------------------------------------------------------------------------------------------
void LcsBaseStationDccTrack::getNextPacket( ) {

    bytesSent     = 0;
    bitsSent      = 0;
    preambleSent  = 0;
    postambleSent = 0;

    if ( activeBufPtr -> repeat > 0 ) {

        activeBufPtr -> repeat --;

        writeLogData( LOG_DCC_PKT, activeBufPtr -> buf, activeBufPtr -> len );
    }
    else if ( flags & DT_F_DCC_PACKET_PENDING ) {

        activeBufPtr  = pendingBufPtr;
        pendingBufPtr = (( pendingBufPtr == &dccBuf1 ) ? &dccBuf2 : &dccBuf1 );
        flags &= ~ DT_F_DCC_PACKET_PENDING;

        writeLogData( LOG_DCC_PKT, activeBufPtr -> buf, activeBufPtr -> len );
    }
    else {

        if ( flags & DT_F_SERVICE_MODE_ON ) {

            activeBufPtr = &resetDccPacket;
            writeLogId( LOG_DCC_RST );
        }
        else {

            activeBufPtr = &idleDccPacket;
            writeLogId( LOG_DCC_IDL );
        }
    }

    dccPacketsSend ++;
}

//-----------------------------------------------------------------------------------------------------------
// Railcom. If the cutout period and the RailCom feature is enabled, the signal state machine will also start
// and stop the UART reader for RailCom data. The final message is then to handle that message. In the cutout
// period, a decoder sends 8 data bytes. They are divided into two channels, 2bytes and another 6 bytes. The
// bytes themselves are encoded such that each byte has four bits set, i.e. a hamming weight of 4. The first
// channel is used to just send the locomotive address when the decoder is addressed. The second channel is
// used only when the decoder is explicitly addressed via a CV operation command to provide the answer to the
// request.
//
// The received datagrams are also recorded in the DCC_LOG, if enabled.
//
// ??? under construction....
// ??? we could store the last loco address in some global variable.
// ??? we could store the channel 2 datagram in the corresponding session.
// ??? still, both pieces of data needs to go somewhere before the next message is received...
//-----------------------------------------------------------------------------------------------------------
void LcsBaseStationDccTrack::startRailComIO( ) {

    CDC::startUartRead( uartRxPin );
}

void LcsBaseStationDccTrack::stopRailComIO( ) {

    CDC::stopUartRead( uartRxPin );
}

uint8_t LcsBaseStationDccTrack::handleRailComMsg( ) {

    railComBufIndex = CDC::getUartBuffer( uartRxPin, railComMsgBuf, sizeof( railComMsgBuf ));

    writeLogData( LOG_DCC_RCM, railComMsgBuf, railComBufIndex );

    for ( uint8_t i = 0; i < railComBufIndex; i++ ) {

        uint8_t dataByte = railComDecode[ railComMsgBuf[ i ]];

        if      ( dataByte == ACK ) ;
        else if ( dataByte == NACK ) ;
```

```
        else if ( dataByte == BUSY ) ;
        else if ( dataByte < 64 ) {

            // ??? valid
            // ??? a railCom message can have multiple datagrams
            // we would need to handle each datagram, one at a time or fill them into a kind of structure
            // that has a slot for the up to maximum 4 datagrams per railCom cutout period.
        }
        else {

            // ??? invalid packet ... if this is channel2, discard the entire message.
        }

        railComMsgBuf[ i ] = dataByte;
    }

    flags &= ~ DT_F_RAILCOM_MSG_PENDING;
    return ( ALL_OK );
}

// ??? not very useful, but good for debugging and initial testing .... and it works like a champ :-)

uint8_t LcsBaseStationDccTrack::getRailComMsg( uint8_t *buf, uint8_t bufLen ) {

    if (( railComBufIndex > 0 ) && ( bufLen > 0 )) {

        uint8_t i = 0;

        do {

            buf[ i ] = railComMsgBuf[ i ];
            i++;

        } while (( i < railComBufIndex ) && ( i < bufLen ));

        return ( i );

    } else return ( 0 );
}

//---------------------------------------------------------------------------------------------------------
// DCC track power is not just a matter of turning power on or off. To address all the requirements of the
// standard, the track is managed by a state machine that implements the start and stop sequences. It is also
// important that we do not really block the progress of the entire base station, so any timing calls are
// handled by timestamp comparison in state machine WAIT states. The track state machine routine is expected
// to be called very often.
//
//  DCC_TRACK_POWER_START1    - this is the first state of a start sequence. When the track should be powered
//                              on, the first activity is to set the status flags and enable the power module.
//                              We set the power module current consumption to the initial limit configured.
//                              The next state is TRACK_POWER_START2.
//
//  DCC_TRACK_POWER_START2    - we stay in this state until the threshold time has passed. Once the threshold
//                              is reached, the current consumption limit is set to the configured limit.
//                              Then we move on to DCC_TRACK_POWER_ON.
//
//  DCC_TRACK_POWER_ON        - this is the state when power is on and things are running normal. An overload
//                              situation is set by the current measurement routines through setting the
//                              overload status flag. We make sure that we have seen a couple of overloads
//                              in a row before taking action which is to turn power off and set the
//                              DCC_TRACK_POWER_OVERLOAD state. Otherwise we stay in this state.
//
//  DCC_TRACK_POWER_OVERLOAD  - with power turned off, we stay in this state until the threshold time has
//                              passed. If passed, the overload restart count is incremented and checked for
//                              its threshold. If reached, we have tried to restart several times and failed.
//                              The track state becomes DCC_TRACK_POWER_STOP1, something is wrong on the track.
//                              If not, we move on to DCC_TRACK_POWER_START1.
//
//  DCC_TRACK_POWER_STOP1     - this state initiates a shutdown sequence. We disable the power module, set
//                              status flags and advance to the DCC_TRACK_POWER_STOP2 state.
//
//  DCC_TRACK_POWER_STOP2     - we stay in this state until the configured threshold has passed. Then we move
//                              on to DCC_TRACK_POWER_OFF. The key reason for this time delay is to implement
//                              the requirement that track turned off and perhaps switched to another mode,
//                              should be powerless for one second. Switch track modes becomes simply a matter
//                              of stopping and then starting again.
//
//  DCC_TRACK_POWER_OFF       - the track is disabled. We just stay in this state until the state is set to
//                              a different state from outside.
//
// During the power on state, we also append the actual current measurement value to a circular buffer when
// the time interval for this kind of measurement has passed. The idea is to measure the samples at a more
// or less constant interval rate and compute the power consumption RMS value from the data in the buffer
// when requested. In the interest of minimizing the controller load, the calculation is done in digit values
// the result is presented in then in milliAmps.
//
//---------------------------------------------------------------------------------------------------------
void LcsBaseStationDccTrack::runDccTrackStateMachine( ) {

    switch ( trackState ) {

        case DCC_TRACK_POWER_START1: {

            // ??? do we need a way to check for overload during this initial phase, just like we do when ON ?

            trackTimeStamp          = CDC::getMillis( );
            flags                   |= DT_F_POWER_ON;
            flags                   &= ~DT_F_POWER_OVERLOAD;
            flags                   &= ~DT_F_MEASUREMENT_ON;
```

```
            limitCurrentDigitValue  = milliAmpToDigitValue( initCurrentMilliAmp, digitsPerAmp );

            CDC::writeDio( enablePin, true );
            trackState = DCC_TRACK_POWER_START2;

        }  break;

        case DCC_TRACK_POWER_START2: {

            if (( CDC::getMillis( ) - trackTimeStamp ) > startTimeThreshold ) {

                highWaterMarkDigitValue = 0;
                actualCurrentDigitValue = 0;
                overloadRestartCount    = 0;
                overloadEventCount      = 0;
                flags                   |= DT_F_POWER_ON | DT_F_MEASUREMENT_ON;
                limitCurrentDigitValue  = milliAmpToDigitValue( limitCurrentMilliAmp, digitsPerAmp );

                CDC::writeDio( enablePin, true );
                trackState = DCC_TRACK_POWER_ON;
            }

        } break;

        case DCC_TRACK_POWER_ON: {

            if (( CDC::getMillis( ) - lastPwrSampleTimeStamp ) > PWR_SAMPLE_TIME_INTERVAL_MILLIS ) {

                pwrSampleBuf[ pwrSampleBufIndex % DCC_TRACK_POWER_ON ] = actualCurrentDigitValue;
                pwrSampleBufIndex ++;
                lastPwrSampleTimeStamp = CDC::getMillis( );
            }

            if (( CDC::getMillis( ) - lastPwrSamplePerSecTimeStamp ) > 1000 ) {

                pwrSamplesPerSec           = totalPwrSamplesTaken - lastPwrSamplePerSecTaken;
                lastPwrSamplePerSecTaken   = totalPwrSamplesTaken;
                lastPwrSamplePerSecTimeStamp = CDC::getMillis( );
            }

            if ( flags & DT_F_POWER_OVERLOAD ) {

                overloadEventCount  ++;

                if ( overloadEventCount > overloadEventThreshold ) {

                    if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_TRACK_POWER_MGMT )) {

                        printf( "Overload detected: " );

                        if ( options &  DT_OPT_SERVICE_MODE_TRACK ) printf( "Prog Track: " );
                        else                                        printf( "Main Track: " );

                        #if 0
                        printf( "(hwm(mA): %d : limit(mA): %d )\n",
                                digitValueToMilliAmp( highWaterMarkDigitValue, digitsPerAmp ),
                                digitValueToMilliAmp( limitCurrentDigitValue, digitsPerAmp ));

                        #else
                        printf( "(hwm(dVal): %d  : limit(dVal): %d )\n", highWaterMarkDigitValue, limitCurrentDigitValue );
                        #endif
                    }

                    trackTimeStamp  = CDC::getMillis( );
                    flags           |= DT_F_POWER_OVERLOAD;
                    flags           &= ~DT_F_POWER_ON;
                    flags           &= ~DT_F_MEASUREMENT_ON;

                    CDC::writeDio( enablePin, false );
                    trackState = DCC_TRACK_POWER_OVERLOAD;
                }
            }

        }  break;

        case DCC_TRACK_POWER_OVERLOAD: {

            if ( CDC::getMillis( ) - trackTimeStamp > overloadTimeThreshold ) {

                overloadRestartCount ++;

                if ( overloadRestartCount > overloadRestartThreshold ) {

                    if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_TRACK_POWER_MGMT )) {

                        printf( "Overload restart failed, Cnt:%d\n", overloadRestartCount );
                    }

                    trackState = DCC_TRACK_POWER_STOP1;
                }
                else trackState = DCC_TRACK_POWER_START1;
            }

        }  break;

        case DCC_TRACK_POWER_STOP1: {

            trackTimeStamp  = CDC::getMillis( );
            flags           &= ~DT_F_POWER_ON;
```

```
            flags           &= ~DT_F_POWER_OVERLOAD;
            flags           &= ~DT_F_MEASUREMENT_ON;

            CDC::writeDio( enablePin, false );
            trackState = DCC_TRACK_POWER_STOP2;

        }  break;

        case DCC_TRACK_POWER_STOP2: {

            if ( CDC::getMillis( ) - trackTimeStamp > stopTimeThreshold ) trackState = DCC_TRACK_POWER_OFF;

        } break;

        case DCC_TRACK_POWER_OFF: {

        } break;
    }
}

//----------------------------------------------------------------------------------------------------------
// Some getter functions. Straightforward.
//
//----------------------------------------------------------------------------------------------------------
uint16_t LcsBaseStationDccTrack::getFlags( ) {

    return ( flags );
}

uint16_t LcsBaseStationDccTrack::getOptions( ) {

    return ( options );
}

uint32_t LcsBaseStationDccTrack::getDccPacketsSend( ) {

    return ( dccPacketsSend );
}

uint32_t LcsBaseStationDccTrack::getPwrSamplesTaken( ) {

    return ( totalPwrSamplesTaken );
}

uint16_t LcsBaseStationDccTrack::getPwrSamplesPerSec( ) {

    return ( pwrSamplesPerSec );
}

bool LcsBaseStationDccTrack::isPowerOn( ) {

    return ( flags & DT_F_POWER_ON );
}

bool LcsBaseStationDccTrack::isPowerOverload( ) {

    return ( flags & DT_F_POWER_OVERLOAD );
}

bool LcsBaseStationDccTrack::isServiceModeOn( ) {

    return ( flags & DT_F_SERVICE_MODE_ON );
}

bool LcsBaseStationDccTrack::isCutoutOn( ) {

    return ( flags & DT_F_CUTOUT_MODE_ON );
}

bool LcsBaseStationDccTrack::isRailComOn( ) {

    return ( flags & DT_F_RAILCOM_MODE_ON );
}

//----------------------------------------------------------------------------------------------------------
// DCC track power management functions. The actual state of track power is kept in the track status field
// and can be queried or set by setting the respective flag. Starting and stopping track power is done by
// setting the respective START or STOP state.
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationDccTrack::powerStart( ) {

    trackState = DCC_TRACK_POWER_START1;
}

void LcsBaseStationDccTrack::powerStop( ) {

    trackState = DCC_TRACK_POWER_STOP1;
}

void LcsBaseStationDccTrack::serviceModeOn( ) {

    if ( options & DT_OPT_SERVICE_MODE_TRACK ) flags |= DT_F_SERVICE_MODE_ON;
}

void LcsBaseStationDccTrack::serviceModeOff( ) {

    if ( options & DT_OPT_SERVICE_MODE_TRACK ) flags &= ~DT_F_SERVICE_MODE_ON;
}
```

```

void LcsBaseStationDccTrack::cutoutOn( ) {

    if ( ! ( options & DT_OPT_SERVICE_MODE_TRACK )) {

        preambleLen =  MAIN_PACKET_PREAMBLE_LEN - DCC_PACKET_CUTOUT_LEN;
        flags       |= DT_F_CUTOUT_MODE_ON;
    }
}

void LcsBaseStationDccTrack::cutoutOff( ) {

    if ( ! ( options & DT_OPT_SERVICE_MODE_TRACK )) {

        preambleLen =  MAIN_PACKET_PREAMBLE_LEN;
        flags       &= ~DT_F_CUTOUT_MODE_ON;
        flags       &= ~DT_F_RAILCOM_MODE_ON;
    }
}

void LcsBaseStationDccTrack::railComOn( ) {

    if ( ! ( options & DT_OPT_SERVICE_MODE_TRACK )) {

        flags |= DT_F_CUTOUT_MODE_ON | DT_F_RAILCOM_MODE_ON;
    }
}

void LcsBaseStationDccTrack::railComOff( ) {

    if ( ! ( options & DT_OPT_SERVICE_MODE_TRACK )) flags &= ~DT_F_RAILCOM_MODE_ON;
}

//----------------------------------------------------------------------------------------------------------
// Power Consumption Management. There are two key values. The first is the actual current consumption as
// measured by the ADC hardware on each ZERO DCC bit. This value is used to do the power overload checking.
// The second value is the high water mark built from these measurements. This values is used for the DCC
// decoder programming logic. The high water mark will be set to zero before collecting measurements. All
// measurement values are actually ADC digit values for performance reason. Only on limit setting and external
// data access are these values converted from and to milliAmps.
//
//----------------------------------------------------------------------------------------------------------
uint16_t LcsBaseStationDccTrack::getLimitCurrent( ) {

    return ( limitCurrentMilliAmp );
}

uint16_t LcsBaseStationDccTrack::getActualCurrent( ) {

    return ( digitValueToMilliAmp( actualCurrentDigitValue, digitsPerAmp ));
}

uint16_t LcsBaseStationDccTrack::getInitCurrent( ) {

    return ( initCurrentMilliAmp );
}

uint16_t LcsBaseStationDccTrack::getMaxCurrent( ) {

    return ( maxCurrentMilliAmp );
}

void LcsBaseStationDccTrack::setLimitCurrent( uint16_t val ) {

    if      ( val < initCurrentMilliAmp )  val = initCurrentMilliAmp;
    else if ( val > maxCurrentMilliAmp  )  val = maxCurrentMilliAmp;

    limitCurrentMilliAmp    = val;
    limitCurrentDigitValue  = milliAmpToDigitValue( val, digitsPerAmp );
}

//----------------------------------------------------------------------------------------------------------
// The "getRMSCurrent" function returns the power consumption based on the samples taken and stored in the
// sample buffer. The function computes the square root of the sum of the squares of the array elements. The
// result is returned in milliAmps. Note that our measurement is based on unsigned 16-bit quantities that come
// from the controller ADC converter. We compute the RMS based on 16-bit unsigned integers, which compared
// to floating point computation is not really precise. However, for our purpose to just show a rough power
// consumption, the error should be not a big issue. We will not use RMS values for power overload detection
// or decoder ACK detection.
//
//----------------------------------------------------------------------------------------------------------
uint16_t LcsBaseStationDccTrack::getRMSCurrent( ) {

    uint32_t res = 0;

    for ( uint8_t i = 0; i < PWR_SAMPLE_BUF_SIZE; i++ ) res += pwrSampleBuf[ i ] * pwrSampleBuf[ i ];

    return ( digitValueToMilliAmp( sqrt( res / PWR_SAMPLE_BUF_SIZE ), digitsPerAmp ));
}

//----------------------------------------------------------------------------------------------------------
// This function is called whenever a power measurement operation completes from the analog conversion
// interrupt handler. This typically takes place on the first half of the DCC "0" bit. If power measurement
// is enabled, we increment the number of samples taken, check the measured value for an overload situation
// and also set the high water mark accordingly. Since we are part of an interrupt handler, keep the amount
// work really short.
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationDccTrack::powerMeasurement( ) {
```

```
    if ( flags & DT_F_MEASUREMENT_ON ) {

        actualCurrentDigitValue = CDC::readAdc( sensePin );

        totalPwrSamplesTaken ++;

        if ( actualCurrentDigitValue > highWaterMarkDigitValue ) highWaterMarkDigitValue = actualCurrentDigitValue;
        if ( actualCurrentDigitValue > limitCurrentDigitValue ) flags |= DT_F_POWER_OVERLOAD;
    }
}

//----------------------------------------------------------------------------------------------------------
// The DCC decoder programming requires the detection of a current consumption change. This is the way a DCC
// decoder signals an acknowledgement. To detect the consumption change we need first an idea what the actual
// average current baseline consumption of the decoder is. This method will send the required DCC reset packets
// according to the DCC standard and at the same time determine the current consumption as a baseline. We use
// the high water mark for this purpose.
//
// ??? although the routines for decoder ACK detection work, they will produce quite a number of packets.
// During this time, other LCS work is blocked. Perhaps we need a kind of state machine approach to cut the
// long sequence in smaller chunks to allow other work in between.
//----------------------------------------------------------------------------------------------------------
uint16_t LcsBaseStationDccTrack::decoderAckBaseline( uint8_t resetPacketsToSend ) {

    if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_DCC_ACK_DETECT )) {

        printf( "\nDecoder Ack setup: ( " );
    }

    uint16_t sum = 0;

    for ( uint8_t i = 0; i < resetPacketsToSend; i++ ) {

        highWaterMarkDigitValue = 0;

        loadPacket( resetDccPacketData, 2, 0 );

        if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_DCC_ACK_DETECT )) {

            printf( "%d ", highWaterMarkDigitValue );
        }

        sum += highWaterMarkDigitValue;
    }

    if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_DCC_ACK_DETECT )) {

        printf( ") -> %d\n", ( sum + resetPacketsToSend - 1 ) / resetPacketsToSend );
    }

    return (( sum + resetPacketsToSend - 1 ) / resetPacketsToSend );
}

//----------------------------------------------------------------------------------------------------------
// "decoderAckDetect" is the counterpart to the decoder ack setup routine. The setup method established a base
// line for the power consumption and put the decoder in CV programming mode by sending the RESET packets. The
// decoder ACK detect routine now sends out resets packets to follow the programming packets required and
// monitors the current consumption. We use the high water mark for this purpose. The DCC standard specifies
// a time window in which the decoder should raise its power consumption level and signal an acknowledge this
// way. We will send out a series of reset packets and monitor after each packet the consumption level. The
// number of retries depends on whether it is a read ( 50ms window ) or a write ( 100ms window). If we detect
// a raised value the decoder did signal a positive outcome. If not, we time out after the last reset packet.
// The programming operation either failed or the decoder did on purpose not answer. We cannot tell.
//
// ??? although the routines for decoder ACK detection work, they will produce quite a number of packets.
// During this time, other LCS work is blocked. Perhaps we need a kind of state machine approach to cut the
// long sequence in smaller chunks to allow other work in between.
//----------------------------------------------------------------------------------------------------------
bool LcsBaseStationDccTrack::decoderAckDetect( uint16_t baseDigitValue, uint8_t retries ) {

    if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_DCC_ACK_DETECT )) {

        printf( "Decoder Ack detect: ( %d : %d : ( ", baseDigitValue, ackThresholdDigitValue );
    }

    for ( uint8_t i = 0; i < retries; i++ ) {

        highWaterMarkDigitValue = 0;

        loadPacket( resetDccPacketData, 2, 0 );

        if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_DCC_ACK_DETECT )) {

            printf( "%d ", highWaterMarkDigitValue );
        }

        if (( highWaterMarkDigitValue >= baseDigitValue ) &&
            ( highWaterMarkDigitValue - baseDigitValue >= ackThresholdDigitValue )) {

            if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_DCC_ACK_DETECT )) {

                printf( "[ %d ] ) -> OK\n", abs( highWaterMarkDigitValue - baseDigitValue ));
            }

        return ( true );
        }
    }

```

```
    if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_DCC_ACK_DETECT )) {

        printf( ") -> FAILED" );
    }

    return ( false );
}

//----------------------------------------------------------------------------------------------------------
// LoadPacket is the central entry point to submit a DCC packet. The incoming packet is the the data to be
// sent without checksum, i.e. it is just the payload. The DCC track signal generator has two packet buffers.
// The first buffer holds the packet currently being transmitted. The second is the pending buffer. If it is
// used, we will simply busy wait for our turn to load the packet into the pending buffer. Upon completion of
// sending the active packet, the interrupt handler copies the currently pending buffer to the active buffer
// and then resets the pending flag. Either way, then it is our turn. We fill the pending buffer, compute the
// checksum and set the pending flag.
//
// ??? For a high number of session we may want to think about a queuing approach. Right now, this routine
// waits when there is a packet already queued, i.e. pending. This may cause issues in delaying other tasks
// such as receiving a CAN bus message.
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationDccTrack::loadPacket( const uint8_t *packet, uint8_t len, uint8_t repeat ) {

    if ( ! isInRangeU( len, MIN_DCC_PACKET_SIZE, MAX_DCC_PACKET_SIZE )) return;
    if ( ! isInRangeU( repeat, MIN_DCC_PACKET_REPEATS, MAX_DCC_PACKET_REPEATS )) return;

    while ( flags & DT_F_DCC_PACKET_PENDING );

    pendingBufPtr -> len    = len + 1;
    pendingBufPtr -> repeat = repeat;

    uint8_t checkSum = 0;
    uint8_t *bufPtr  = pendingBufPtr -> buf;

    for ( uint8_t i = 0; i < len; i++ ) {

        bufPtr[ i ] =   packet[ i ];
        checkSum    ^=  bufPtr[ i ];
    }

    bufPtr[ len ] = checkSum;
    flags         |= DT_F_DCC_PACKET_PENDING;
}

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

    logEnabled = arg;
    logActive  = false;
}

void LcsBaseStationDccTrack::beginLog( ) {

    if ( logEnabled ) {

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

void LcsBaseStationDccTrack::endLog( ) {

    if ( logActive ) {

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

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

    if ( logActive ) {

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

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

void LcsBaseStationDccTrack::writeLogId( uint8_t id ) {
```

```

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

void LcsBaseStationDccTrack::writeLogTs( ) {

    if ( logActive ) {

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

void LcsBaseStationDccTrack::writeLogVal( uint8_t valId, uint16_t val ) {

    if ( logActive ) {

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

//--------------------------------------------------------------------------------------------------------
// Print out the log data, one entry on one line. We only print the log buffer when there is no log sequence
// active.
//
//--------------------------------------------------------------------------------------------------------
void LcsBaseStationDccTrack::printLog( ) {

    if ( logEnabled ) {

        if ( ! logActive ) {

            if ( logBufIndex > 0 ) {

                printf( "\n" );

                uint16_t entryIndex  = 0;
                uint8_t  entryLen    = 0;

                while ( entryIndex < logBufIndex ) {

                    entryLen = printLogEntry( entryIndex );
                    printf( "\n" );

                    if ( entryLen > 0 ) entryIndex += entryLen;
                    else                break;
                }
            }
            else printf( "DCC Log Buf: Nothing recorded\n" );
        }
        else printf( "DCC Log Active\n" );
    }
    else printf( "DCC Log disabled\n" );
}

//--------------------------------------------------------------------------------------------------------
// Print out the DCC Track configuration data. For debugging purposes.
//
//--------------------------------------------------------------------------------------------------------
void LcsBaseStationDccTrack::printDccTrackConfig( ) {

    printf( "DccTrack Config: " );

    if ( options & DT_OPT_SERVICE_MODE_TRACK ) printf( "PROG \n" );
    else                                       printf( "MAIN \n" );

    printf( " Config options: ( 0x%x ) -> ", flags );

    if ( options &  DT_OPT_SERVICE_MODE_TRACK ) printf( "SvcMode Track " );
    if ( options & DT_OPT_CUTOUT ) printf( "Cutout " );
    if ( options & DT_OPT_RAILCOM ) printf( "Railcom " );
    printf( "\n" );

    printf( " Current Initial(mA): %d Current Limit(mA): %d Current Max(mA): %d\n",
            getInitCurrent( ), getLimitCurrent( ), getMaxCurrent( ));
    printf( " milliVoltPerAmp: %d\n", milliVoltPerAmp );
    printf( " digitsPerAmp: %d\n", digitsPerAmp );

    printf( " Limit Digit Value: %d\n", limitCurrentDigitValue );
    printf( " Ack Threshold Digit Value:%d\n", ackThresholdDigitValue );

    printf( " CDC enable Pin: %d, DCC signal Pins: (%d:%d), Sensor Pin: %d, RailCom Pin: %d\n",
            enablePin, dccSigPin1, dccSigPin2, sensePin, uartRxPin );

    printf( " PreambleLen: %d, PostambleLen: %d\n", preambleLen, postambleLen );
}

//--------------------------------------------------------------------------------------------------------
// Print out the DCC Track status.
//
//--------------------------------------------------------------------------------------------------------
void LcsBaseStationDccTrack::printDccTrackStatus( ) {
```

```
    printf( "DccTrack: " );

    if ( options & DT_OPT_SERVICE_MODE_TRACK )  printf( "PROG" );
    else                                        printf( "MAIN" );

    printf( ", Track Status: ( 0x%x ) -> ", flags );

    if ( flags & DT_F_POWER_ON          ) printf( "PowerOn " );
    if ( flags & DT_F_POWER_OVERLOAD    ) printf( "PowerOverload " );
    if ( flags & DT_F_MEASUREMENT_ON    ) printf( "PowerMeasOn " );
    if ( flags & DT_F_SERVICE_MODE_ON   ) printf( "SvcModeOn " );
    if ( flags & DT_F_CUTOUT_MODE_ON    ) printf( "CutoutOn " );
    if ( flags & DT_F_RAILCOM_MODE_ON   ) printf( "RailcomOn " );
    if ( flags & DT_F_CONFIG_ERROR      ) printf( "ConfigError " );
    printf( "\n" );

    printf( "Packets Send: %d\n", dccPacketsSend );
    printf( "Total Power Samples: %d\n", totalPwrSamplesTaken );
    printf( "Power Samples per Sec: %d\n", pwrSamplesPerSec );
    printf( "Power consumption (RMS): %d\n", getRMSCurrent( ));
    printf( "\n" );
}
```

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

using namespace LCS;

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

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

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

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

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

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

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

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

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

bool validCabId( uint16_t cabId ) {

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

bool validCvId( uint16_t cvId ) {

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

bool validFunctionId( uint8_t fId ) {

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

bool validFunctionGroupId( uint8_t fGroup ) {

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

bool validDccPacketlen( uint8_t len ) {

```

```
    return ( isInRangeU( len, MIN_DCC_PACKET_SIZE, MAX_DCC_PACKET_SIZE ));
}

bool validDccPacketRepeatCnt( uint8_t nRepeat ) {

    return ( isInRangeU( nRepeat, MIN_DCC_PACKET_REPEATS, MAX_DCC_PACKET_REPEATS ));
}

uint8_t lowByte( uint16_t arg ) {

    return( arg & 0xFF );
}

uint8_t highByte( uint16_t arg ) {

    return( arg >> 8 );
}

uint8_t bitRead( uint8_t arg, uint8_t pos ) {

    return ( arg >> ( pos % 8 )) & 1;
}

void bitWrite( uint8_t *arg, uint8_t pos, bool val ) {

    if ( val )  *arg |= ( 1 << pos );
    else        *arg &= ~( 1 << pos );
}

//--------------------------------------------------------------------------------------------------------------
// DDC function flags. The DCC function flags F0 .. F68 are stored in ten groups. Group 0 contains F0 .. F4
// stored in DCC command byte format. Group 1 contains F5 .. F8, Group 2 contains F9 .. F12 in DCC command
// byte format. The remainder F13 .. F68 are stored in 8 bits groups also in DCC command byte format. The
// routines support the get/set of an individual bit as well as setting an entire function group. A DCC
// function group is labelled starting with index 1.
//
//--------------------------------------------------------------------------------------------------------------
bool getDccFuncBit( uint8_t *funcFlags, uint8_t fNum ) {

    if      ( fNum == 0 )                  return ( bitRead( funcFlags[ 0 ], 4 ));
    else if ( isInRangeU( fNum, 1, 4 ))    return ( bitRead( funcFlags[ 0 ], fNum - 1 ));
    else if ( isInRangeU( fNum, 5, 8 ))    return ( bitRead( funcFlags[ 1 ], fNum - 5 ));
    else if ( isInRangeU( fNum, 9, 12 ))   return ( bitRead( funcFlags[ 2 ], fNum - 9 ));
    else if ( isInRangeU( fNum, 13, 68 )) {

        return ( bitRead( funcFlags[ ( fNum - 13 ) / 8 + 3 ], ( fNum - 13 ) % 8 ));
    }
    else return false;
}

void setDccFuncBit( uint8_t *funcFlags, uint8_t fNum, bool val ) {

    if      ( fNum == 0 )                  bitWrite( &funcFlags[ 0 ], 4, val );
    else if ( isInRangeU( fNum, 1, 4 ))    bitWrite( &funcFlags[ 0 ], fNum - 1, val );
    else if ( isInRangeU( fNum, 5, 8 ))    bitWrite( &funcFlags[ 1 ], fNum - 5, val );
    else if ( isInRangeU( fNum, 9, 12 ))   bitWrite( &funcFlags[ 2 ], fNum - 9, val );
    else if ( isInRangeU( fNum, 13, 68 )) {

        bitWrite( &funcFlags[ ( fNum - 13 ) / 8 + 3 ], ( fNum - 13 ) % 8, val );
    }
}

void setDccFuncGroupByte( uint8_t *funcFlags, uint8_t fGroup, uint8_t dccByte ) {

    if        ( fGroup == 1 )                    funcFlags[ 0 ] = dccByte & 0x1F;
    else if   ( fGroup == 2 )                    funcFlags[ 1 ] = dccByte & 0x0F;
    else if   ( fGroup == 3 )                    funcFlags[ 2 ] = dccByte & 0x0F;
    else if   ( isInRangeU( fGroup, 4, 10 ))     funcFlags[ fGroup - 1 ] = dccByte;
}

uint8_t dccFunctionBitToGroup( uint8_t fNum ) {

    if      ( isInRangeU( fNum, 0, 4 ))      return ( 1 );
    else if ( isInRangeU( fNum, 5, 8 ))      return ( 2 );
    else if ( isInRangeU( fNum, 9, 12 ))     return ( 3 );
    else if ( isInRangeU( fNum, 13, 68 ))    return (( fNum - 13 ) / 8 + 4 );
    else                                     return ( 0 );
}

}; // namespace

//==============================================================================================================
//==============================================================================================================
//
// Object part.
//
//==============================================================================================================
//==============================================================================================================

//--------------------------------------------------------------------------------------------------------------
// "LocoSession" constructor. Nothing to do here.
//
//--------------------------------------------------------------------------------------------------------------
LcsBaseStationLocoSession::LcsBaseStationLocoSession( ) { }

//--------------------------------------------------------------------------------------------------------------
// Loco Session Map configuration. The session map contains an array of loco sessions entries. We are passed
// the sessionMap descriptor and object handles to the core library and the two tracks. Loco sessions are
// numbered from 1 to MAX_SESSION_ID. During compilation there is a maximum number of sessions that the
```

```
// session map will support. This number cannot be changed other than recompile with a different setting.
//
//----------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::setupSessionMap(

    LcsBaseStationSessionMapDesc  *sessionMapDesc,
    LcsBaseStationDccTrack        *mainTrack,
    LcsBaseStationDccTrack        *progTrack

    ) {

    if (( mainTrack == nullptr                                 ) ||
        ( progTrack == nullptr                                 ) ||
        ( sessionMapDesc -> maxSessions > MAX_CAB_SESSIONS  )) return ( ERR_SESSION_SETUP );

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

    options                = sessionMapDesc -> options;
    flags                  = SM_F_DEFAULT_SETTING;
    sessionMap             = (SessionMapEntry *) calloc( sessionMapDesc -> maxSessions, sizeof( SessionMapEntry ));
    lastAliveCheckTime     = CDC::getMillis( );

    sessionMapHwm          = sessionMap;
    sessionMapLimit        = &sessionMap[ sessionMapDesc -> maxSessions ];
    sessionMapNextRefresh = sessionMap;

    if ( options & SM_OPT_ENABLE_REFRESH )      flags |= SM_F_ENABLE_REFRESH;
    if ( options & SM_OPT_KEEP_ALIVE_CHECKING ) flags |= SM_F_KEEP_ALIVE_CHECKING;

    for ( SessionMapEntry *smePtr = sessionMap; smePtr < sessionMapLimit; smePtr++ ) initSessionEntry( smePtr );

    return ( ALL_OK );
}

//----------------------------------------------------------------------------------------------------------
// "requestSession" is the entry point to establish a session. There are several modes. The NORMAL mode is
// to allocate a new session. There should be no session already existing for this cabId. The STEAL mode
// grabs an existing session from the current session holder. The use case is that a dispatched locomotive
// can be taken over by another handheld. The SHARED option allows several handheld controller to share the
// session entry and issue commands to the same locomotive. Right now, the STEAL and SHARED option are not
// implemented.
//
//----------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::requestSession( uint16_t cabId, uint8_t mode, uint8_t *sId ) {

    *sId = NIL_LOCO_SESSION_ID;
    if ( ! validCabId( cabId )) return ( ERR_INVALID_CAB_ID );

    switch ( mode ) {

        case LSM_NORMAL: {

            SessionMapEntry *smePtr = allocateSessionEntry( cabId );
            if ( smePtr == nullptr ) return ( ERR_LOCO_SESSION_ALLOCATE );

            smePtr -> flags |= SME_SPDIR_ONLY_REFRESH;

            *sId = smePtr - sessionMap + 1;
            return ( ALL_OK );
        }

        case LSM_STEAL: {

            // ??? need to inform the current handheld and put the new handheld in its place.
            return ( ERR_NOT_IMPLEMENTED );

        } break;

        case LSM_SHARED: {

            // ??? essentially, add another handheld to the session. We perhaps need a counter on how many handhelds
            // share the session ...
            return ( ERR_NOT_IMPLEMENTED );

        } break;

        default: return ( ERR_NOT_IMPLEMENTED ); // ??? rather "invalid mode" ?
    }
}

//----------------------------------------------------------------------------------------------------------
// A cab session can be released, freeing up the slot in the cab session table.
//
// ??? for a shared session, what does this mean ?
//----------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::releaseSession( uint8_t sId ) {

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

    deallocateSessionEntry( smePtr );
    return ( ALL_OK );
}

//----------------------------------------------------------------------------------------------------------
// "updateSession" informs the base station about changes in the loco session setting. To be implemented once
// we know what the flags and the update concept should be ...
//
```

```
//----------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::updateSession( uint8_t sId, uint8_t flags ) {

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

    return ( ERR_NOT_IMPLEMENTED );
}

//----------------------------------------------------------------------------------------------------------
// "markSessionAlive" sets the keep alive time stamp on a loco session. This routine is typically called by
// the LCS message receiver to update the session last "alive" timestamp. The base station will periodically
// check this value to see if a session is still alive.
//
//----------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::markSessionAlive( uint8_t sId ) {

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

    smePtr -> lastKeepAliveTime = CDC::getMillis( );
    return ( ALL_OK );
}

//----------------------------------------------------------------------------------------------------------
// "refreshActiveSessions" walks through the session map up to the high water mark and invokes the session
// refresh function for each used entry. As the refresh entry routine will show, we will do this refreshing
// in small pieces in order to stay responsive to external requests.
//
//
// ??? this may should perhaps all be reworked. There are many more duties to do periodically.
// ??? an active loco ( speed > 0 ) needs to be address at least every 2.5 seconds.
//
// ??? also a base station needs to broadcast its capabilities every
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationLocoSession::refreshActiveSessions( ) {

    if (( flags & SM_F_ENABLE_REFRESH ) && ( sessionMapHwm > sessionMap )) {

        refreshSessionEntry( sessionMapNextRefresh );

        sessionMapNextRefresh ++;
        if ( sessionMapNextRefresh >= sessionMapHwm ) sessionMapNextRefresh = sessionMap;
    }
}

//----------------------------------------------------------------------------------------------------------
// "refreshSessionEntry" checks first that the session is still alive and then issues the next DCC packet for
// refreshing the loco session. To avoid DCC bandwidth issues, a loco session refresh is done in several small
// steps. There is one state for speed and direction and steps to refresh the function groups 1 to 5. If the
// function refresh option is set, we use the DCC command that sets speed, direction and the function flags in
// one DCC command.
//
//    Step 0  -> refresh speed and direction ( if FUNC_REFRESH is set also functions F0 .. F28 )
//    Step 1  -> refresh function group 0 ( F0  .. F4  )
//    Step 2  -> refresh function group 1 ( F5  .. F8  )
//    Step 3  -> refresh function group 2 ( F9  .. F12 )
//    Step 4  -> refresh function group 3 ( F13 .. F20 )
//    Step 5  -> refresh function group 4 ( F21 .. F28 )
//
// ??? should we alternate when SPDIR and FUNC are sent separately ?
// ??? is it something like: SPDIR, FG1, SPDIR, FG2, ...
//
// ??? what to do for emergency stop, keep refreshing ? keep alive checking ?
// ??? how do we integrate the STEAL/SHARE/DISPATCHED concept ?
//
// ??? separate out the check alive functionality ? it is a separate task...
// ??? sessionMapNextAliveCheck var needed ...
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationLocoSession::refreshSessionEntry( SessionMapEntry *smePtr ) {

    // ??? introduce a return status ?

    if ( smePtr -> cabId != NIL_CAB_ID ) {

        if ( flags & SM_F_KEEP_ALIVE_CHECKING ) {

            if (( CDC::getMillis( ) - smePtr -> lastKeepAliveTime ) > refreshAliveTimeOutVal ) {

                if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_CHECK_ALIVE_SESSIONS )) {

                    printf( "Session: %d expired\n", smePtr - sessionMap );
                }

                deallocateSessionEntry( smePtr );
            }
        }

        // ??? separate keep alive checking and refresh options...

        else {

            // ??? if ( smePtr -> speed > 0 )  // only active locos are refreshed...

            if ( smePtr -> nextRefreshStep == 0 ) {

                setThrottle( smePtr, smePtr -> speed, smePtr -> direction );

```

```
                smePtr -> nextRefreshStep = ((( smePtr -> flags & SME_COMBINED_REFRESH ) ||
                                            ( smePtr -> flags & SME_SPDIR_ONLY_REFRESH )) ? 0 : 1 );
            }
            else if ( smePtr -> nextRefreshStep <= 5 ) {

            uint8_t fGroup = smePtr -> nextRefreshStep;

            setDccFunctionGroup( smePtr, fGroup, smePtr -> functions[ fGroup - 1 ] );
            smePtr -> nextRefreshStep = (( smePtr -> nextRefreshStep >= 5 ) ? 0 : smePtr -> nextRefreshStep + 1 );

            }
        }
    }
}

//----------------------------------------------------------------------------------------------------------
// "emergencyStopAll" is called when one of the clients issued an emergency stop all request. There is a DCC
// broadcast packet that causes all decoders to stop the locos. In addition, the base station is expected to
// discontinue sending non-zero speed packets until the situation is cleared. The standard does not really say
// what exactly to do. In our base station, we will first issue the ESTOP DCC broadcast packet and then set
// the speed value in each session to one, which is the value for emergency stop. All else is unchanged.
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationLocoSession::emergencyStopAll( ) {

    mainTrack -> loadPacket( eStopDccPacketData, 2, 4 );

    for ( SessionMapEntry *smePtr = sessionMap; smePtr < sessionMapHwm; smePtr++ ) {

        if ( smePtr -> cabId != NIL_CAB_ID ) smePtr -> speed = 1;
    }
}

//----------------------------------------------------------------------------------------------------------
// Getter methods for session related info. Straightforward.
//
//----------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::getSessionIdByCabId( uint16_t cabId ) {

    SessionMapEntry *smePtr = lookupSessionEntry( cabId );
    return (( smePtr == nullptr ) ? NIL_LOCO_SESSION_ID : (( smePtr - sessionMap ) + 1 ));
}

uint16_t LcsBaseStationLocoSession::getOptions( ) {

    return ( options );
}

uint16_t LcsBaseStationLocoSession::getFlags( ) {

    return ( flags );
}

uint8_t LcsBaseStationLocoSession::getSessionMapHwm( ) {

    return ( sessionMapHwm - sessionMap );
}

uint32_t LcsBaseStationLocoSession::getSessionKeepAliveInterval( ) {

    return ( refreshAliveTimeOutVal );
}

uint8_t LcsBaseStationLocoSession::getActiveSessions( ) {

    uint8_t sessionCnt = 0;

    for ( SessionMapEntry *smePtr = sessionMap; smePtr < sessionMapHwm; smePtr++ ) {

        if ( smePtr -> cabId != NIL_CAB_ID ) sessionCnt++;
    }

    return ( sessionCnt );
}

//----------------------------------------------------------------------------------------------------------
// "setThrottle" is perhaps the most used function. After all, we want to run engines on the track. This
// signature will just locate the session map entry and then invoke the internal signature with accepts a
// pointer to the entry.
//
//----------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::setThrottle( uint8_t sId, uint8_t speed, uint8_t direction ) {

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

    return ( setThrottle( smePtr, speed, direction ));
}

//----------------------------------------------------------------------------------------------------------
// "setThrottle" will send a DCC packet with speed and direction for a loco. If the combined speed and
// function refresh option is enabled, the DCC command will specify speed, direction and functions to refresh
// in one packet.
//
//----------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::setThrottle( SessionMapEntry *smePtr, uint8_t speed, uint8_t direction ) {

    uint8_t pBuf[ MAX_DCC_PACKET_SIZE ];
    uint8_t pLen = 0;
```

```

    smePtr -> speed      = speed & 0x7F;
    smePtr -> direction  = direction % 2;

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

    pBuf[pLen++] = (( smePtr -> flags & SME_COMBINED_REFRESH ) ? 0x3c : 0x3F );
    pBuf[pLen++] = (( smePtr -> speed & 0x7F ) | (( smePtr -> direction ) ? 0x80 : 0 ));

    if ( smePtr -> flags & SME_COMBINED_REFRESH ) {

        pBuf[pLen++]  = ((( smePtr -> functions[0] & 0x10 ) >> 4 ) |
                        (( smePtr -> functions[0] & 0x0F ) << 1 ) |
                        (( smePtr -> functions[1] & 0x07 ) << 5 ));

        pBuf[pLen++]  = ((( smePtr -> functions[1] & 0x0F ) >> 3 ) |
                        (( smePtr -> functions[2] & 0x0F ) << 1 ) |
                        (( smePtr -> functions[3] & 0x07 ) << 5 ));

        pBuf[pLen++]  = ((( smePtr -> functions[3] & 0xf80 ) >> 3 ) |
                        (( smePtr -> functions[4] & 0x07 ) << 5 ));

        pBuf[pLen++]  = (( smePtr -> functions[4] & 0xf80 ) >> 3 );
    }

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

//-----------------------------------------------------------------------------------------------------------
// "setDccFunctionBit" controls the functions in a decoder. The DCC function flags F0 .. F68 are stored in
// ten groups. The routines first updates the function bit in the loco session entry data structure, so we
// can keep track of the values. This is important as the DCC commands send out entire groups only. The
// actual work is then done by the "setDccFunctionGroup" method.
//
//-----------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::setDccFunctionBit( uint8_t sId, uint8_t fNum, uint8_t val ) {

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

    if ( ! validFunctionId( fNum )) return ( ERR_INVALID_FUNC_ID );
    setDccFuncBit( smePtr -> functions, fNum, val );

    uint8_t fGroup = dccFunctionBitToGroup( fNum );

    return ( setDccFunctionGroup( smePtr, fGroup, smePtr -> functions[ fGroup - 1 ] ));
}

//-----------------------------------------------------------------------------------------------------------
// "setDccFunctionGroup" sets an entire group of function flags. This signature will first find the session
// entry, do the argument checks and the invoke the internal signature.
//
//-----------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::setDccFunctionGroup( uint8_t sId, uint8_t fGroup, uint8_t dccByte ) {

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

    return ( setDccFunctionGroup( smePtr, fGroup, dccByte ));
}

//-----------------------------------------------------------------------------------------------------------
// "setDccFunctionGroup" sets an entire group of function flags.The DCC function flags F0 .. F68 are stored
// in ten groups.
//
//      Group 1:  F0, F4, F3, F2, F1        DCC Command Format: 100DDDDD
//      Group 2:  F8, F7, F6, F5            DCC Command Format: 1011DDDD
//      Group 3:  F12, F11, F10, F9         DCC Command Format: 1010DDDD
//      Group 4:  F20 .. F13                DCC Command Format: 0xDE DDDDDDDD
//      Group 5:  F28 .. F21                DCC Command Format: 0xDF DDDDDDDD
//      Group 6:  F36 .. F29                DCC Command Format: 0xD8 DDDDDDDD
//      Group 7:  F44 .. F37                DCC Command Format: 0xD9 DDDDDDDD
//      Group 8:  F52 .. F45                DCC Command Format: 0xDA DDDDDDDD
//      Group 9:  F60 .. F53                DCC Command Format: 0xDB DDDDDDDD
//      Group 10: F68 .. F61                DCC Command Format: 0xDC DDDDDDDD
//
// The routines updates the entire function group byte in the loco session entry, so we can keep track of the
// values. The function command is repeated 4 times to the track.
//
//-----------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::setDccFunctionGroup( SessionMapEntry *smePtr, uint8_t fGroup, uint8_t dccByte ) {

    if ( ! validFunctionGroupId( fGroup )) return ( ERR_INVALID_FGROUP_ID );
    setDccFuncGroupByte( smePtr -> functions, fGroup, dccByte );

    uint8_t pBuf[ MAX_DCC_PACKET_SIZE];
    uint8_t pLen = 0;

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

    switch ( fGroup - 1 ) {

        case 0: pBuf[pLen++] = ( smePtr -> functions[ 0 ] & 0x1F ) | 0x80; break;
        case 1: pBuf[pLen++] = ( smePtr -> functions[ 1 ] & 0x0F ) | 0xB0; break;
        case 2: pBuf[pLen++] = ( smePtr -> functions[ 2 ] & 0x0F ) | 0xA0; break;

```

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

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

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

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

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

    uint8_t   pBuf[ MAX_DCC_PACKET_SIZE ];
    uint8_t   pLen = 0;

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

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

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

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

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

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

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

    uint8_t pBuf[ MAX_DCC_PACKET_SIZE ];
    uint8_t pLen = 0;

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

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

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

```
// are not supported. For bit mode access, the bit position and bit value are encoded in the "val" parameter
// with bit 3 containing the data and bit 0 ..2 the bit offset.
//
//     0 - Direct Byte
//     1 - Direct Bit
//     2 - Page Mode
//     3 - Register Mode
//     4 - Address Only Mode
//
// This function implements the CV read mode 0 and 1, which is reading a byte or a bit at a time by calling
// the respective method.
//
//----------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::readCV( uint16_t cvId, uint8_t mode, uint8_t *val ) {

    if         ( mode == 0 )  return ( readCVByte( cvId, val ));
    else if    ( mode == 1 )  return ( readCVBit( cvId, *val % 8, val ));
    else                      return ( ERR_INVALID_CV_MODE );
}

//----------------------------------------------------------------------------------------------------------
// "readCVByte" will retrieve a complete byte from the decoder. CV numbers range from 1 to 1024, but are
// encoded from 0 to 1023. This command is only available in service mode, i.e. on a programming track.
// Reading a CV value where the decoder can only respond with a "yes" or "no" is a tedious matter. We are
// actually reading the CV value bit by bit and then ask if the assembled byte read is the one just read. The
// general packet sequence is a according to DCC standard standard 3 or more RESET packets, 5 or more identical
// READ packets and then RESET packages until acknowledge or timeout. The RESET packet preamble and postamble
// series are sent during the decoder ack setup and detect call to the DCC track object. During the preamble
// we  figure out the base current consumption of the decoder, during the postamble packets we measure to get
// the decoder acknowledge, which is a short raise in power consumption to indicate an ACK.
//
//
// ??? This command may take a long time, a lot of packets are sent. While this not an issue with the signal
// generation, which is done via interrupt handlers, it may be an issue with any other work of the base
// station. This code needs to be redesigned to use a kind of state machine that sends a packet at a time
// so other work can interleave.
//----------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::readCVByte( uint16_t cvId, uint8_t *val ) {

    if ( ! ( progTrack -> isServiceModeOn( ))) return ( ERR_NO_SVC_MODE );
    if ( ! validCvId( cvId )) return ( ERR_INVALID_CV_ID );
    cvId--;

    uint8_t    pBuf[ MAX_DCC_PACKET_SIZE ];
    uint8_t    bValue  = 0;
    uint16_t   base    = progTrack -> decoderAckBaseline( 5 );

    pBuf[0]   = 0x78 + ( highByte( cvId ) & 0x03 );
    pBuf[1]   = lowByte( cvId );

    for ( int i = 0; i < 8; i++ ) {

        pBuf[2] = 0xE8 + i;
        progTrack -> loadPacket( pBuf, 3, 5 );
        bitWrite( &bValue, i, progTrack -> decoderAckDetect( base, 9 ));
    }

    *val    = bValue;
    pBuf[0] = 0x74 + ( highByte( cvId ) & 0x03 );
    pBuf[1] = lowByte( cvId );
    pBuf[2] = bValue;
    progTrack -> loadPacket( pBuf, 3, 5 );

    return (( progTrack -> decoderAckDetect( base, 9 )) ? ALL_OK : (LcsErrorCodes) ERR_CV_OP_FAILED );
}

//----------------------------------------------------------------------------------------------------------
// "readCVBit" will retrieve one bit from a CV variable from the decoder. CV numbers range from 1 to 1024,
// but are encoded from 0 to 1023. This command is only available in service mode, i.e. on a programming
// track. The "val" parameter encodes the bit position in bits 0 - 2. We are reading the CV value bit and
// then ask if the bit read is the one just read. We first try to validate a zero bit. If that succeeds,
// fine. Otherwise we try to validate a one bit. If that succeeds, fine. Otherwise we have a CV read error.
// The general packet sequence is a according to DCC standard 3 or more RESET packets, 5 or more identical
// READ packets and then RESET packages until acknowledge or timeout. The RESET packet preamble and postamble
// are sent during the decoder ack setup and detect call to the DCC track object. During the preamble we
// figure out the base current consumption of the decoder, during the postamble we measure to get the decoder
// acknowledge, which is a short raise in power consumption to indicate an ACK.
//
// ??? This command may take a long time, a lot of packets are sent. While this not an issue with the signal
// generation, which is done via interrupt handlers, it may be an issue with any other work of the base
// station. This code needs to be redesigned to use a kind of state machine that sends a packet at a time
// so other work can interleave.
//----------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::readCVBit( uint16_t cvId, uint8_t bitPos, uint8_t *val ) {

    if ( ! ( progTrack -> isServiceModeOn( ))) return ( ERR_NO_SVC_MODE );

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

    uint8_t  pBuf[ MAX_DCC_PACKET_SIZE ];
    int   base = progTrack -> decoderAckBaseline( 5 );

    pBuf[0]   = 0x78 + ( highByte( cvId ) & 0x03 );
    pBuf[1]   = lowByte( cvId );
    pBuf[2]   = 0xE8 + ( bitPos % 8 );

    progTrack -> loadPacket( pBuf, 3, 5 );

```

```
    if ( ! ( progTrack -> decoderAckDetect( base, 9 ))) {

        pBuf[2] = 0xE8 + 8 + ( bitPos % 8 );
        progTrack -> loadPacket( pBuf, 3, 5 );

        if ( progTrack -> decoderAckDetect( base, 9 )) {

            *val = 1;
            return ( ALL_OK );
        }
        else return ( ERR_CV_OP_FAILED );
    }
    else return ( ALL_OK );
}

//----------------------------------------------------------------------------------------------------------------
// "writeCV" writes a CV value to the decoder. CV numbers range from 1 to 1024, but are encoded from 0 to
// 1023. This command is only available in service mode, i.e. on a programming track. The DCC standard defines
// various modes for accessing CV values. For bit mode access, the bit position and bit value are encoded in
// the "val" parameter with bit 3 containing the data and bit 0 .. 2 the bit offset.
//
//    0 Direct Byte
//    1 Direct Bit
//    2 Page Mode
//    3 Register Mode
//    4 Address Only Mode
//
// This function implements the CV write mode 0 and 1, which is writing a byte or a bit at a time by calling
// the respective method.
//
//----------------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::writeCV( uint16_t cvId, uint8_t mode, uint8_t val ) {

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

//----------------------------------------------------------------------------------------------------------------
// "writeCVByte" puts a data byte into the CV on the decoder. This function is only available in service mode.
// The CV numbers range from 1 to 1024, but are encoded from 0 to 1023. The data byte written will also be
// verified. The packet sequence follows the DCC standard. We will send the CV byte write packet four times,
// send out several RESET packets and the send the verify packets to get the acknowledge from the decoder that
// the operation was successful.
//
// ??? This command may take a long time, a lot of packets are sent. While this not an issue with the signal
// generation, which is done via interrupt handlers, it may be an issue with any other work of the base
// station. This code needs to be redesigned to use a kind of state machine that sends a packet at a time
// so other work can interleave.
//----------------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::writeCVByte( uint16_t cvId, uint8_t val ) {

    if ( ! ( progTrack -> isServiceModeOn( ))) return ( ERR_NO_SVC_MODE );

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

    uint8_t pBuf[ MAX_DCC_PACKET_SIZE ];
    int     base = progTrack -> decoderAckBaseline( 5 );

    pBuf[0] = 0x7C + ( highByte( cvId ) & 0x03 );
    pBuf[1] = lowByte( cvId );
    pBuf[2] = val;

    progTrack -> loadPacket( pBuf, 3, 4 );
    progTrack -> loadPacket( resetDccPacketData, 2, 11 );

    pBuf[0] = 0x74 + ( highByte( cvId ) & 0x03 );
    progTrack -> loadPacket( pBuf, 3, 5 );

    return (( progTrack -> decoderAckDetect( base, 9 )) ? ALL_OK : (LcsErrorCodes) ERR_CV_OP_FAILED );
}

//----------------------------------------------------------------------------------------------------------------
// "writeCVBit" puts a data bit into the CV on the decoder. This function is only available in session mode.
// The CV numbers range from 1 to 1024, but are encoded from 0 to 1023. For the bit mode,  the "val" parameter
// encodes the bit position in bits 0 - 2 and the bit value in bit 3. The packet sequence follows the DCC
// standard, similar to the byte write operation.
//
// ??? This command may take a long time, a lot of packets are sent. While this not an issue with the signal
// generation, which is done via interrupt handlers, it may be an issue with any other work of the base
// station. This code needs to be redesigned to use a kind of state machine that sends a packet at a time
// so other work can interleave.
//----------------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::writeCVBit( uint16_t cvId, uint8_t bitPos, uint8_t val ) {

    if ( ! ( progTrack -> isServiceModeOn( ))) return ( ERR_NO_SVC_MODE );
    if ( ! validCvId( cvId )) return ( ERR_INVALID_CV_ID );
    cvId--;

    uint8_t pBuf[ MAX_DCC_PACKET_SIZE ];
    int     base = progTrack -> decoderAckBaseline( 5 );

    pBuf[0] = 0x78 + ( highByte( cvId ) & 0x03 );
    pBuf[1] = lowByte( cvId );
    pBuf[2] = 0xF0 + (( val % 2 ) * 8 ) + ( bitPos % 8 );

    progTrack -> loadPacket( pBuf, 3, 4 );
    progTrack -> loadPacket( resetDccPacketData, 2, 11 );
```

```

    bitWrite( &pBuf[2], 4, false );
    progTrack -> loadPacket( pBuf, 3, 5 );

    return (( progTrack -> decoderAckDetect( base, 9 )) ? ALL_OK : (LcsErrorCodes) ERR_CV_OP_FAILED );
}

//----------------------------------------------------------------------------------------------------------
// "writeDccPacketMain" just load the DCC packet into the buffer and out it goes to the main track without
// any further checks.
//
//----------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::writeDccPacketMain( uint8_t *pBuf, uint8_t pLen, uint8_t nRepeat ) {

    if ( ! validDccPacketlen( pLen )) return ( ERR_INVALID_PACKET_LEN );
    if ( ! validDccPacketRepeatCnt( nRepeat )) return ( ERR_INVALID_REPEATS );

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

//----------------------------------------------------------------------------------------------------------
// "writeDccPacketProg" just load the DCC packet into the buffer and out it goes to the programming track
// without any further checks.
//
//----------------------------------------------------------------------------------------------------------
uint8_t LcsBaseStationLocoSession::writeDccPacketProg( uint8_t *pBuf, uint8_t pLen, uint8_t nRepeat ) {

    if ( ! validDccPacketlen( pLen )) return ( ERR_INVALID_PACKET_LEN );
    if ( ! validDccPacketRepeatCnt( nRepeat )) return ( ERR_INVALID_REPEATS );

    progTrack -> loadPacket( pBuf, pLen, nRepeat );
    return ( ALL_OK );
}

//----------------------------------------------------------------------------------------------------------
// "allocateSessionEntry" allocates a new loco session entry and returns a pointer to the entry. We first
// check if there is already a session for the cabId and if so, we return a null pointer. If not, we try to
// find a free entry and if that fails try to raise the high water mark. If that fails, we are out of luck
// and return a null pointer.
//
//----------------------------------------------------------------------------------------------------------
SessionMapEntry* LcsBaseStationLocoSession::allocateSessionEntry( uint16_t cabId ) {

    if ( lookupSessionEntry( cabId ) != nullptr ) return ( nullptr );

    SessionMapEntry *freePtr = lookupSessionEntry( NIL_CAB_ID );

    if (( freePtr == nullptr ) && ( sessionMapHwm < sessionMapLimit )) freePtr = sessionMapHwm ++;

    if ( freePtr != nullptr ) {

        initSessionEntry( freePtr );
        freePtr -> cabId  = cabId;
        freePtr -> flags  |= SME_ALLOCATED;

        if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_SESSION )) {

            printf( "Allocate session entry: %d, HWM: %d\n",
                ( freePtr - sessionMap + 1 ), ( sessionMapHwm - sessionMap ));
        }
    }

    return ( freePtr );
}

//----------------------------------------------------------------------------------------------------------
// "deallocateSessionEntry" is the counterpart to the entry allocation. We just free up the entry. If the
// entry is at the high water mark, we try to free up all possibly free entries from the high water mark
// downward, decrementing the high water mark. This way the high water mark shrinks again and we do not need
// to work through unused entries in the middle.
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationLocoSession::deallocateSessionEntry( SessionMapEntry *smePtr ) {

    if (( smePtr != nullptr ) && ( smePtr >= sessionMap ) && ( smePtr < sessionMapHwm )) {

        if ( smePtr == ( sessionMapHwm - 1 )) {

            do {

                initSessionEntry( smePtr );
                smePtr --;
            }
            while (( smePtr -> cabId == NIL_CAB_ID ) && ( smePtr >= sessionMap ));

            sessionMapHwm = smePtr + 1;
        }
        else initSessionEntry( smePtr );

       if (( debugMask & DBG_BS_CONFIG ) && ( debugMask & DBG_BS_SESSION )) {

            printf( "Released Session, sId: %d, ,new  HWM: %d\n",
                ( smePtr - sessionMap + 1 ), ( sessionMapHwm - sessionMap ));
       }
    }
}

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

```
// "lookupSessionEntry" scans the session map for a session entry for the cabId. If none is found, a nullptr
// is returned. Note that a NIL_CAB_ID as argument is also a valid input and will return the first free entry.
//
//----------------------------------------------------------------------------------------------------------
SessionMapEntry *LcsBaseStationLocoSession::lookupSessionEntry( uint16_t cabId ) {

    SessionMapEntry *smePtr = sessionMap;

    while ( smePtr < sessionMapHwm ) {

        if ( smePtr -> cabId == cabId ) return ( smePtr );
        else smePtr ++;
    }

    return ( nullptr );
}

//----------------------------------------------------------------------------------------------------------
// "initSessionEntry" initializes a session map entry with default values.
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationLocoSession::initSessionEntry( SessionMapEntry *smePtr ) {

    smePtr -> flags             = SME_DEFAULT_SETTING;
    smePtr -> cabId             = NIL_CAB_ID;
    smePtr -> speedSteps        = DCC_SPEED_STEPS_128;
    smePtr -> speed             = 0;
    smePtr -> direction         = 0;
    smePtr -> engineState       = 0;
    smePtr -> lastKeepAliveTime = 0;
    smePtr -> nextRefreshStep   = 0;

    for ( int i = 0; i < MAX_DCC_FUNC_GROUP_ID; i++ ) smePtr -> functions[ i ] = 0;
}

//----------------------------------------------------------------------------------------------------------
// "getSessionMapEntryPtr" returns a pointer to a valid and used sessionMap entry. The sessionId starts with
// index 1.
//
//----------------------------------------------------------------------------------------------------------
SessionMapEntry *LcsBaseStationLocoSession::getSessionMapEntryPtr( uint8_t sId ) {

    if ( ! isInRangeU( sId, MIN_LOCO_SESSION_ID, ( sessionMapHwm - sessionMap ))) return ( nullptr );
    return (( sessionMap[ sId - 1 ].cabId == NIL_CAB_ID ) ? nullptr : &sessionMap[ sId - 1 ] );
}

//----------------------------------------------------------------------------------------------------------
// "printSessionMapConfig" lists cab session map configuration data.
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationLocoSession::printSessionMapConfig( ) {

    printf( "Session Map Config\n" );
    printf( " Options: 0x%x\n", options );
    printf( " Session Map Size: %d\n", ( sessionMapLimit - sessionMap ));
}

//----------------------------------------------------------------------------------------------------------
// "printSessionMapInfo" lists the cab session map data.
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationLocoSession::printSessionMapInfo( ) {

    printf( "Session Map Info\n" );

    printf( " Flags: 0x%x\n", flags );

    // ??? decode the flags ?  e.g. "[ f f f f ]"

    printf( " Session Map Hwm: %d\n", ( sessionMapHwm - sessionMap ));

    for ( SessionMapEntry *smePtr = sessionMap; smePtr < sessionMapHwm; smePtr ++ ) {

        if ( smePtr -> cabId != NIL_CAB_ID ) printSessionEntry( smePtr );
    }

     printf( "\n" );
}

//----------------------------------------------------------------------------------------------------------
// "printSessionEntry" lists a cab session.
//
//----------------------------------------------------------------------------------------------------------
void LcsBaseStationLocoSession::printSessionEntry( SessionMapEntry *smePtr ) {

  if ( smePtr != nullptr ) {

    printf( " sId: %d, cabId: %d, speed: %d ", ( smePtr - sessionMap + 1 ), smePtr -> cabId, smePtr -> speed );

    printf( "%s", (( smePtr -> direction ) ? "Rev" : "Fwd" ));
    printf( ", functions: " );

    for ( uint8_t i = 0; i < MAX_DCC_FUNC_GROUP_ID; i++ ) {

      printf( " 0x%x ", smePtr -> functions[ i ] );
    }

    printf( " Flags: 0x%x", ( smePtr -> flags ));

```

```
    // ??? decode the flags ? e.g. "[ f f f f ]"
  }

  printf( "\n" );
}
```

```
//----------------------------------------------------------------------------------------------------------
//
// LCS - Base Station
//
//----------------------------------------------------------------------------------------------------------
// This is the main program for the LCS base station. Every layout would need at least a base station. Its
// primary task is to manage the DCC loco sessions, generate the DCC signals and manage the dual DCC track
// power outputs.
//
// Like all other LcsNodes, the base station will provide a rich set of variable that can be set and queried.
// In addition, the base features a command line extension which implements the DCC++ style commands and
// some more base station specific commands. The idea for the DCC++ command syntax and commands is that these
// command can also be submitted by a third party software ( e.g. JMRI ). An example would be the JMRI CV
// programming tool.
//
// ??? we need an idea of system time like DCC. To be broadcasted periodically.
// ??? we also need a broadcast of the layout system capabilities....
//
//----------------------------------------------------------------------------------------------------------
//
// LCS - Controller Dependent Code - Raspberry PI Pico Implementation
// Copyright (C) 2022 - 2024 Helmut Fieres
//
// This program is free software: you can redistribute it and/or modify it under the terms of the GNU General
// Public License as published by the Free Software Foundation, either version 3 of the License, or (at your
// option) any later version.
//
// This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
// implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
// for more details.
//
// You should have received a copy of the GNU General Public License along with this program. If not, see
// http://www.gnu.org/licenses
//
//  GNU General Public License:  http://opensource.org/licenses/GPL-3.0
//
//----------------------------------------------------------------------------------------------------------
#include "LcsCdcLib.h"
#include "LcsRuntimeLib.h"
#include "LcsBaseStation.h"

using namespace LCS;

//----------------------------------------------------------------------------------------------------------
// Base station global data.
//
// ??? can the objects for track and session just use these variables instead of keeping them locally as a
// field ?
//----------------------------------------------------------------------------------------------------------
uint16_t                        debugMask;
CDC::CdcConfigDesc              cdcConfig;
LCS::LcsConfigDesc              lcsConfig;
LcsBaseStationCommand           serialCmd;
LcsBaseStationDccTrack          mainTrack;
LcsBaseStationDccTrack          progTrack;
LcsBaseStationLocoSession       locoSessions;
LcsBaseStationMsgInterface      msgInterface;

//---------------------------------------------------------------------------------------------------------
// Setup the configuration of the HW board. The CDC config contains the HW pin mapping. The dual bridge pins
// for enabling the bridge and controlling its direction. The pins are mapped to the CDC pin names DIO2 to
// DIO7 as show below. DIO-0 and DIO-1 are routed to the extension connector board.
//
//       cdcConfig.DIO_PIN_0      -> DIO-0
//       cdcConfig.DIO_PIN_1      -> DIO-1
//       cdcConfig.DIO_PIN_2      -> Main dcc1
//       cdcConfig.DIO_PIN_3      -> Main dcc2
//       cdcConfig.DIO_PIN_4      -> Prog ddc1
//       cdcConfig.DIO_PIN_5      -> Prog ddc2
//       cdcConfig.DIO_PIN_6      -> Main enable
//       cdcConfig.DIO_PIN_7      -> Prog enable
//
// Current mapping: Main Controller Board B.01.00 - PICO - newest version.
//
//       cdcConfig.DIO_PIN_0      = 8;
//       cdcConfig.DIO_PIN_1      = 12;
//       cdcConfig.DIO_PIN_2      = 21;
//       cdcConfig.DIO_PIN_3      = 20;
//       cdcConfig.DIO_PIN_4      = 19;
//       cdcConfig.DIO_PIN_5      = 18;
//       cdcConfig.DIO_PIN_6      = 6;
//       cdcConfig.DIO_PIN_7      = 7;
//
// In addition, the HW pins for I2C, analog inputs and so on are set. Check the schematic for the board
// to see all pin assign,ents.
//
// ??? one day we will have several base station versions. Although they will perhaps differ, their the CDC
// pin names used should not change. But we would need to come up with an idea which configuration to use
// when preparing an image for the base station board.
//---------------------------------------------------------------------------------------------------------
void setupConfigInfo( ) {

    cdcConfig = CDC::getConfigDefault( );
    lcsConfig = LCS::getConfigDefault( );

    cdcConfig.ADC_PIN_0             = 26;
    cdcConfig.ADC_PIN_1             = 27;

```

```
    cdcConfig.PFAIL_PIN             = 5;
    cdcConfig.EXT_INT_PIN           = 22;
    cdcConfig.READY_LED_PIN         = 14;
    cdcConfig.ACTIVE_LED_PIN        = 15;

    cdcConfig.DIO_PIN_0             = 8;
    cdcConfig.DIO_PIN_1             = 12;
    cdcConfig.DIO_PIN_2             = 21;
    cdcConfig.DIO_PIN_3             = 20;
    cdcConfig.DIO_PIN_4             = 19;
    cdcConfig.DIO_PIN_5             = 18;
    cdcConfig.DIO_PIN_6             = 6;
    cdcConfig.DIO_PIN_7             = 7;

    cdcConfig.UART_RX_PIN_1         = 13;
    cdcConfig.UART_RX_PIN_2         = 9;

    cdcConfig.NVM_I2C_SCL_PIN       = 3;
    cdcConfig.NVM_I2C_SDA_PIN       = 2;
    cdcConfig.NVM_I2C_ADR_ROOT      = 0x50;

    cdcConfig.EXT_I2C_SCL_PIN       = 17;
    cdcConfig.EXT_I2C_SDA_PIN       = 16;
    cdcConfig.EXT_I2C_ADR_ROOT      = 0x50;

    cdcConfig.CAN_BUS_RX_PIN        = 0;
    cdcConfig.CAN_BUS_TX_PIN        = 1;
    cdcConfig.CAN_BUS_CTRL_MODE     = CAN_BUS_LIB_PICO_PIO_125K_M_CORE;
    cdcConfig.CAN_BUS_DEF_ID        = 100;

    cdcConfig.NODE_NVM_SIZE         = 8192;
    cdcConfig.EXT_NVM_SIZE          = 4096;

    lcsConfig.options               |= NOPT_SKIP_NODE_ID_CONFIG;
}

//----------------------------------------------------------------------------------------------------------
// Some little helper functions.
//
//----------------------------------------------------------------------------------------------------------
void printLcsMsg( uint8_t *msg ) {

  int msgLen = (( msg[0] >> 5 ) + 1 ) % 8;

  for ( int i = 0; i < msgLen; i++ ) printf( "0x%x ", msg[i] );
  printf( "\n" );
}

uint8_t printStatus (uint8_t status ) {

  printf( "Status: " );
  if ( status == LCS::ALL_OK ) printf( "OK\n" );
  else printf ( "FAILED: %d\n", status );
  return ( status );
}

//--------------------------------------------------------------------------------------------------------
// The node and port initialization callback.
//
// ??? when we know what ports we actually need / use, disable the rest of the ports.
//--------------------------------------------------------------------------------------------------------
uint8_t lcsInitCallback( uint16_t npId ) {

    switch ( npId & 0xF ) {

        case 0:     printf( "Node Init Callback: 0x%x\n", npId >> 4     ); break;
        default:    printf( "Port Init Callback: 0x%x\n", npId &  0xF   );
    }

    return( ALL_OK );
}

//--------------------------------------------------------------------------------------------------------
// The node or port reset callback.
//
//--------------------------------------------------------------------------------------------------------
uint8_t lcsResetCallback( uint16_t npId ) {

    switch ( npId & 0xF ) {

        case 0:     printf( "Node Reset Callback: 0x%x\n", npId >> 4     ); break;
        default:    printf( "Port Reset Callback: 0x%x\n", npId &  0xF   );
    }

    return( ALL_OK );
}

//--------------------------------------------------------------------------------------------------------
// The node or port power fail callback.
//
//--------------------------------------------------------------------------------------------------------
uint8_t lcsPfailCallback( uint16_t npId ) {

    switch ( npId & 0xF ) {

        case 0:     printf( "Node Power Fail Callback: 0x%x\n", npId >> 4     ); break;
        default:    printf( "Port Power Fail Callback: 0x%x\n", npId &  0xF   );
    }

```

```
    return( ALL_OK );
}

//------------------------------------------------------------------------------------------------------
// The base station has also a command line interpreter. The callback is invoked by the core library when
// there is a command that it does not handle.
//
//------------------------------------------------------------------------------------------------------
uint8_t lcsCmdCallback( char *cmdLine ) {

    serialCmd.handleSerialCommand( cmdLine );
    return( ALL_OK );
}

//------------------------------------------------------------------------------------------------------
// Other LCS message callbacks. All we do is to list their invocation. ( for now )
//
//------------------------------------------------------------------------------------------------------
uint8_t lcsMsgCallback( uint8_t *msg ) {

    printf( "MsgCallback: ", msg  );

    for ( int i = 0; i < 8; i++ ) printf( "0x%2x ");
    printf( "\n" );
    return( ALL_OK );
}

//--------------------------------------------------------------------------------------------------------
// The LCS core library ends in a loop that manages its internal workings, invoking the callbacks where
// needed. One set of callbacks are the periodic tasks. The base station needs to periodically run the DCC
// track state machine for power consumption measurement and so on.  Another periodic task is to refresh the
// active locomotive session entries.
//
//--------------------------------------------------------------------------------------------------------
uint8_t bsMainTrackCallback( ) {

    mainTrack.runDccTrackStateMachine( );
    return( ALL_OK );
}

uint8_t bsProgTrackCallback( ) {

    progTrack.runDccTrackStateMachine( );
    return( ALL_OK );
}

uint8_t bsRefreshActiveSessionCallback( ) {

    locoSessions.refreshActiveSessions( );
    return( ALL_OK );
}

//--------------------------------------------------------------------------------------------------------
// When the base station node receives a request with an item defined in the user item range or the base
// station itself issues such a request, the defined callback is invoked.
//
//--------------------------------------------------------------------------------------------------------
uint8_t lcsReqCallback( uint8_t npId, uint8_t item, uint16_t *arg1, uint16_t *arg2 ) {

    printf( "REQ callback: npId: 0x%x, item: %d", npId, item );
    if ( arg1 != nullptr ) printf( ", arg1: %d, ", *arg1 ); else printf( ", arg1: null" );
    if ( arg2 != nullptr ) printf( ", arg2: %d, ", *arg2 ); else printf( ", arg2: null" );
    return( ALL_OK );
}

//--------------------------------------------------------------------------------------------------------
// When the base station gets a reply message for a request previously sent, this callback is invoked.
//
//--------------------------------------------------------------------------------------------------------
uint8_t lcsRepCallback( uint8_t npId, uint8_t item, uint16_t arg1, uint16_t arg2, uint8_t ret ) {

    printf( "REP callback: npId: 0x%x, item: %d, arg1: %d, arg2: %d, ret: %d ", npId, item , arg1, arg2, ret );
    return( ALL_OK );
}

//--------------------------------------------------------------------------------------------------------
// For any event on the LCS system that the base station is interested in, this callback is invoked.
//
//--------------------------------------------------------------------------------------------------------
uint8_t lcsEventCallback( uint16_t npId, uint16_t eId, uint8_t eAction, uint16_t eData ) {

    printf( "Event: npId: 0x%x, eId: %d, eAction: %d, eData: %d\n", npId, eId, eAction, eData );
    return( ALL_OK );
}

//------------------------------------------------------------------------------------------------------
// Init the Runtime.
//
//------------------------------------------------------------------------------------------------------
uint8_t initLcsRuntime( ) {

    setupConfigInfo( );

    uint8_t rStat = LCS::initRuntime( &lcsConfig, &cdcConfig );
    printf( "LCS Base Station\n" );

    CDC::printConfigInfo( &cdcConfig );

    printStatus( rStat );
```

```
    return( rStat );
}

//----------------------------------------------------------------------------------------------------
// This routine initializes the Loco Session Map Object.
//
//----------------------------------------------------------------------------------------------------
uint8_t setupLocoSessions( ) {

  LcsBaseStationSessionMapDesc sessionDesc;

  sessionDesc.options     = SM_OPT_ENABLE_REFRESH;
  sessionDesc.maxSessions = 16;

  printf( "Setup Session Map -> " );
  return ( printStatus( locoSessions.setupSessionMap( &sessionDesc, &mainTrack, &progTrack )));
}

//----------------------------------------------------------------------------------------------------
// This routine initializes the MAIN track object.
//
// ??? define constants such as: SENSE_0R1_OPAMP_11 to set the milliVolts per Amp.
//----------------------------------------------------------------------------------------------------
int setupDccTrackMain( ) {

  LcsBaseStationTrackDesc mainTrackDesc;

  mainTrackDesc.options                    = DT_OPT_RAILCOM | DT_OPT_CUTOUT;

  mainTrackDesc.enablePin                  = cdcConfig.DIO_PIN_6;
  mainTrackDesc.dccSigPin1                 = cdcConfig.DIO_PIN_2;
  mainTrackDesc.dccSigPin2                 = cdcConfig.DIO_PIN_3;
  mainTrackDesc.sensePin                   = cdcConfig.ADC_PIN_0;
  mainTrackDesc.uartRxPin                  = cdcConfig.UART_RX_PIN_1;

  mainTrackDesc.initCurrentMilliAmp        = 500;
  mainTrackDesc.limitCurrentMilliAmp       = 1500;
  mainTrackDesc.maxCurrentMilliAmp         = 2000;
  mainTrackDesc.milliVoltPerAmp            = 100 * 11;  // ??? opAmp has Factor eleven ...
  mainTrackDesc.startTimeThresholdMillis   = 1000;
  mainTrackDesc.stopTimeThresholdMillis    = 500;
  mainTrackDesc.overloadTimeThresholdMillis = 500;
  mainTrackDesc.overloadEventThreshold     = 10;
  mainTrackDesc.overloadRestartThreshold   = 5;

  printf( "Setup MAIN track -> " );
  return ( printStatus( mainTrack.setupDccTrack( &mainTrackDesc )));
}

//----------------------------------------------------------------------------------------------------
// This routine initializes the PROG track object.
//
// ??? define constants such as: SENSE_0R1_OPAMP_11 to set the milliVolts per Amp.
//----------------------------------------------------------------------------------------------------
uint8_t setupDccTrackProg( ) {

  LcsBaseStationTrackDesc progTrackDesc;

  progTrackDesc.options                    = DT_OPT_SERVICE_MODE_TRACK;

  progTrackDesc.enablePin                  = cdcConfig.DIO_PIN_7;
  progTrackDesc.dccSigPin1                 = cdcConfig.DIO_PIN_4;
  progTrackDesc.dccSigPin2                 = cdcConfig.DIO_PIN_5;
  progTrackDesc.sensePin                   = cdcConfig.ADC_PIN_1;
  progTrackDesc.uartRxPin                  = cdcConfig.UART_RX_PIN_2;

  progTrackDesc.initCurrentMilliAmp        = 500;
  progTrackDesc.limitCurrentMilliAmp       = 500;
  progTrackDesc.maxCurrentMilliAmp         = 1000;
  progTrackDesc.milliVoltPerAmp            = 100 * 11;  // ??? opAmp has Factor eleven ...
  progTrackDesc.startTimeThresholdMillis   = 1000;
  progTrackDesc.stopTimeThresholdMillis    = 500;
  progTrackDesc.overloadTimeThresholdMillis = 500;
  progTrackDesc.overloadEventThreshold     = 10;
  progTrackDesc.overloadRestartThreshold   = 5;

  printf( "Setup PROG track -> " );
  return ( printStatus( progTrack.setupDccTrack( &progTrackDesc )));
}

//----------------------------------------------------------------------------------------------------
// The base station has also a command interpreter, primarily for the DCC++ commands.
//
//----------------------------------------------------------------------------------------------------
uint8_t setupSerialCommand( ) {

  printf( "Setup Serial Command -> " );
  return ( printStatus( serialCmd.setupSerialCommand( &locoSessions, &mainTrack, &progTrack )));
}

//----------------------------------------------------------------------------------------------------
// The LCS message interface is initialized in the LCS core library. This routine will set up the receiver
// handler for incoming LCS message that concern the base station.
//
//----------------------------------------------------------------------------------------------------
uint8_t setupMsgInterface( ) {

  printf( "Setup LCS Msg Interface -> " );
  return ( printStatus( msgInterface.setupLcsMsgInterface( &locoSessions, &mainTrack, &progTrack )));
```

```
}

//------------------------------------------------------------------------------------------------------------
// After the initial setup of the runtime library, the callback are registered.
//
//------------------------------------------------------------------------------------------------------------
uint8_t registerCallbacks( ) {

    printf( "Registering Callbacks\n" );

    registerLcsMsgCallback( lcsMsgCallback );
    registerCmdCallback( lcsCmdCallback );
    registerInitCallback( lcsInitCallback );
    registerResetCallback( lcsResetCallback );
    registerPfailCallback( lcsPfailCallback );
    registerReqCallback( lcsReqCallback );
    registerRepCallback( lcsRepCallback );
    registerEventCallback( lcsEventCallback );
    registerTaskCallback( bsMainTrackCallback, MAIN_TRACK_STATE_TIME_INTERVAL );
    registerTaskCallback( bsProgTrackCallback, PROG_TRACK_STATE_TIME_INTERVAL );
    registerTaskCallback( bsRefreshActiveSessionCallback, SESSION_REFRESH_TASK_INTERVAL );

    return( ALL_OK );
}

//------------------------------------------------------------------------------------------------------------
// Fire up the base station. First all base station modules are initialized. If this is OK, the DCC tack
// signal generation is enabled, i.e. the interrupt driven DCC packet broadcasting starts. Finally, the
// track power is turned on and we give control to the LCS runtime for processing events and requests.
//
//------------------------------------------------------------------------------------------------------------
uint8_t startBaseStation( ) {

    uint8_t rStat = ALL_OK;

    if ( rStat == ALL_OK ) rStat = setupSerialCommand( );
    if ( rStat == ALL_OK ) rStat = setupMsgInterface( );
    if ( rStat == ALL_OK ) rStat = setupLocoSessions( );
    if ( rStat == ALL_OK ) rStat = setupDccTrackMain( );
    if ( rStat == ALL_OK ) rStat = setupDccTrackProg( );

    if ( rStat == ALL_OK ) {

        LcsBaseStationDccTrack::startDccProcessing( );

        mainTrack.powerStart( );
        progTrack.powerStart( );

        // ??? bracket so that it is not printed when no console...
        mainTrack.printDccTrackStatus( );
        progTrack.printDccTrackStatus( );
        printf( "Ready...\n" );

        startRuntime( );
  }

  return( ALL_OK );
}

//------------------------------------------------------------------------------------------------------------
// The main program. Setup the runtime, register the callbacks, and get the show on the road.
//
//------------------------------------------------------------------------------------------------------------
int main( ) {

    uint8_t rStat = ALL_OK;

    if ( rStat == ALL_OK ) rStat = initLcsRuntime( );
    if ( rStat == ALL_OK ) rStat = registerCallbacks( );
    if ( rStat == ALL_OK ) return( startBaseStation( ));
}
```

12.2 CDC Lib

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

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

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

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

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

    MEM_SIZE_ERR        = 10,

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

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

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

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

```

};

//-----------------------------------------------------------------------------------------------------------
// Controller pin related definitions. A pin can be valid, undefined or illegal. An undefined pin for a pin
// field in the configuration structure indicates that  the pin has not been used by the firmware
// implementation but is a pin that the particular controller would support. An illegal pin means that the
// pin is not offered by this controller and cannot be assigned at all.
//
//-----------------------------------------------------------------------------------------------------------
const uint8_t UNDEFINED_PIN    = 255;
const uint8_t ILLEGAL_PIN      = 254;

//-----------------------------------------------------------------------------------------------------------
// The controller families. Currently, there is only the Raspberry PI Pico models.
//
//-----------------------------------------------------------------------------------------------------------
enum ControllerFamily : uint8_t {

    CF_UNDEFINED     = 0,
    CF_RP_PICO       = 1
};

//-----------------------------------------------------------------------------------------------------------
// DIO pin related definitions. A digital pin can be an input pin, with or without pull-up, or an output
// pin. DIO pins can also be associated with an interrupt handler. The handler itself is mapped to an edge
// or level event.
//
//-----------------------------------------------------------------------------------------------------------
enum dioMode : uint8_t {

    IN             = 0,
    OUT            = 1,
    IN_PULLUP      = 2
};


//-----------------------------------------------------------------------------------------------------------
// GPIO interrupts are detected as level change or edge changes.
//
//-----------------------------------------------------------------------------------------------------------
enum intEventTyp : uint8_t {

    EVT_NONE    = 0,
    EVT_LOW     = 1,
    EVT_HIGH    = 2,
    EVT_FALL    = 3,
    EVT_RISE    = 4,
    EVT_CHANGE  = 5
};

//-----------------------------------------------------------------------------------------------------------
// The UART modes. There are two implementations. The PICO offers two hardware UARTS. We use them with 8
// bits with a parity bit. The second type UART is a software implementation based on the PICO PIO blocks.
//
//-----------------------------------------------------------------------------------------------------------
enum UartMode : uint8_t {

    UART_MODE_UNDEFINED = 0,
    UART_MODE_8N1       = 1,
    UART_MODE_8N1_PIO   = 2
};

//-----------------------------------------------------------------------------------------------------------
// Callback functions signatures.
//
//-----------------------------------------------------------------------------------------------------------
extern "C" {

    typedef void ( *TimerCallback ) ( uint32_t timerVal );
    typedef void ( *GpioCallback ) ( uint8_t pin, uint8_t event );
}

//-----------------------------------------------------------------------------------------------------------
// CDC features a data structure that records all HW specific pins and flags. The values are set by the
// initialization code in a project and are validated. All modules in a project will then just use the
// data structure fields using the data for calls to the Hal layer. For example, an application that
// uses DIO_PIN_0 and DIO_PIN_1 will set the HW pin numbers of the controller / board combination used
// in a config data structure "cfg". A call to write a value to the DIO pin, will then just use
// "cfg.DIO_PIN_1" as argument in the "writeDio" call. The "writeDio" call itself will not check the
// value of the configured DIO pin, all it will do is to ensure that it is not UNDEFINED. Note that the
// structure has more pins defined that a potential controller may have. If so, these fields are set to
// UNDEFINED. The structure is the superset of all possible HW items to configure.
//
// In a later runtime version, we may put this structure as constant data into the non-volatile chip on
// the board. It will then just be read from there.
//
//-----------------------------------------------------------------------------------------------------------
struct CdcConfigDesc {

    uint8_t   CFG_STATUS;

    uint8_t   PFAIL_PIN;
    uint8_t   EXT_INT_PIN;
    uint8_t   READY_LED_PIN;
    uint8_t   ACTIVE_LED_PIN;

    uint8_t   DIO_PIN_0;
    uint8_t   DIO_PIN_1;
```

```
    uint8_t   DIO_PIN_2;
    uint8_t   DIO_PIN_3;
    uint8_t   DIO_PIN_4;
    uint8_t   DIO_PIN_5;
    uint8_t   DIO_PIN_6;
    uint8_t   DIO_PIN_7;
    uint8_t   DIO_PIN_8;
    uint8_t   DIO_PIN_9;
    uint8_t   DIO_PIN_10;
    uint8_t   DIO_PIN_11;
    uint8_t   DIO_PIN_12;
    uint8_t   DIO_PIN_13;
    uint8_t   DIO_PIN_14;
    uint8_t   DIO_PIN_15;

    uint8_t   ADC_PIN_0;
    uint8_t   ADC_PIN_1;
    uint8_t   ADC_PIN_2;
    uint8_t   ADC_PIN_3;

    uint8_t   PWM_PIN_0;
    uint8_t   PWM_PIN_1;
    uint8_t   PWM_PIN_2;
    uint8_t   PWM_PIN_3;

    uint8_t   UART_RX_PIN_0;
    uint8_t   UART_TX_PIN_0;

    uint8_t   UART_RX_PIN_1;
    uint8_t   UART_TX_PIN_1;

    uint8_t   UART_RX_PIN_2;
    uint8_t   UART_TX_PIN_2;

    uint8_t   UART_RX_PIN_3;
    uint8_t   UART_TX_PIN_3;

    uint8_t   SPI_MOSI_PIN_0;
    uint8_t   SPI_MISO_PIN_0;
    uint8_t   SPI_SCLK_PIN_0;

    uint8_t   SPI_MOSI_PIN_1;
    uint8_t   SPI_MISO_PIN_1;
    uint8_t   SPI_SCLK_PIN_1;

    uint8_t   NVM_I2C_SCL_PIN;
    uint8_t   NVM_I2C_SDA_PIN;
    uint8_t   NVM_I2C_ADR_ROOT;

    uint8_t   EXT_I2C_SCL_PIN;
    uint8_t   EXT_I2C_SDA_PIN;
    uint8_t   EXT_I2C_ADR_ROOT;

    uint32_t  NODE_NVM_SIZE;
    uint32_t  EXT_NVM_SIZE;

    uint8_t   CAN_BUS_CTRL_MODE;
    uint8_t   CAN_BUS_RX_PIN;
    uint8_t   CAN_BUS_TX_PIN;
    uint32_t  CAN_BUS_DEF_ID;
};

//----------------------------------------------------------------------------------------------------------
// The routines that make up the hardware abstraction layer. The routines expect hardware pin numbers.
// To recap, the CDC layer offers a set of reserved resource names, such as "DIO_PIN_0", which describes
// the resource containing the hardware pin and some flags. The configuration routines in this layer will use
// these pins and other data stored to configure the hardware. Under the defined resource name name all
// upper layers refer to the hardware using the to the configured IO capabilities.
//
// Complex resources, such as the UART or SPI interface, have more than one HW pin they will use. In this
// case one of the HW pins, see the function documentation, will serve as the handle to the resource.
//
//----------------------------------------------------------------------------------------------------------

//----------------------------------------------------------------------------------------------------------
// The console IO functions. We will provide a serial IO via the USB connector of the PICO. The files
// need to be linked with the "tinyUSB" library and the cmake file needs to set the option. Then we can
// use scanf and printf and so on. In addition, we need  function  that just attempts to read a character
// and returns immediately when there is none.
//
//----------------------------------------------------------------------------------------------------------
uint8_t         configureConsoleIO( );
bool            isConsoleConnected( );
char            getConsoleChar( uint32_t timeoutVal = 0 );

//----------------------------------------------------------------------------------------------------------
// CDC setup and configuration routines. The idea is to help the library write with a default configuration
// structure. All pins HW that are fixed in their location will be set. A library programmer will just get
// that default structure and set the values necessary for the particular case.
//
//----------------------------------------------------------------------------------------------------------
CdcConfigDesc   getConfigDefault( );
CdcConfigDesc   *getConfigActual( );
void            printConfigInfo( CdcConfigDesc *ci );
void            setDebugLevel( uint8_t level = 0 );

uint8_t         init( CdcConfigDesc *ci );
void            fatalError( uint8_t n );
void            fatalErrorMsg( char *str, uint8_t n, uint8_t rStat );
```

```

//----------------------------------------------------------------------------------------------------------
// General controller routines.
//
//----------------------------------------------------------------------------------------------------------
uint16_t      getFamily( );
uint32_t      getVersion( );
uint32_t      getChipMemSize( );
uint32_t      getChipNvmSize( );
uint32_t      getCpuFrequency( );
uint32_t      getMillis( );
uint32_t      getMicros( );
void          sleepMillis( uint32_t val );
void          sleepMicros( uint32_t val );

//----------------------------------------------------------------------------------------------------------
// The LCS runtime needs to build a unique ID for the node.
//
//----------------------------------------------------------------------------------------------------------
uint32_t      createUid( );

//----------------------------------------------------------------------------------------------------------
// Timer management routines.
//
//----------------------------------------------------------------------------------------------------------
void          onTimerEvent( TimerCallback functionId );
void          startRepeatingTimer( uint32_t val );
void          setRepeatingTimerLimit( uint32_t val );
uint32_t      getRepeatingTimerLimit( );
void          stopRepeatingTimer( );

//----------------------------------------------------------------------------------------------------------
// Analog input routines.
//
//----------------------------------------------------------------------------------------------------------
uint8_t       configureAdc( uint8_t adcPin );
uint16_t      getAdcRefVoltage( );
uint16_t      getAdcDigitRange( );
uint16_t      readAdc( uint8_t adcPin );

//----------------------------------------------------------------------------------------------------------
// Digital Input/Output routines.
//
//----------------------------------------------------------------------------------------------------------
uint8_t       configureDio( uint8_t dioPin, uint8_t Mode = IN );
void          registerDioCallback( uint8_t dioPin, uint8_t event, CDC::GpioCallback func );
void          unregisterDioCallback( uint8_t dioPin );
bool          readDio( uint8_t dioPin );
uint8_t       writeDio( uint8_t dioPin, bool val );
uint8_t       toggleDio( uint8_t dioPin );
uint32_t      readDioMask( uint32_t dioMask );
uint8_t       writeDioMask( uint32_t dioMask, uint32_t dioVal );
uint8_t       writeDioPair( uint8_t dioPin1, bool val1, uint8_t dioPin2, bool val2 );

//----------------------------------------------------------------------------------------------------------
// PWM output routines.
//
//----------------------------------------------------------------------------------------------------------
uint8_t       configurePwm( uint8_t   pwmPin,
                            uint32_t  pwmFreqency,
                            bool      phaseCorrect  = true,
                            bool      inverted      = false
                          );

uint8_t       writePwm( uint8_t pwmPin, uint8_t dutyCycle );

//----------------------------------------------------------------------------------------------------------
// Serial IO routines.
//
//----------------------------------------------------------------------------------------------------------
uint8_t       configureUart( uint8_t rxPin, uint8_t txPin, uint32_t baudRate, UartMode mode );
uint8_t       startUartRead( uint8_t rxPin );
uint8_t       stopUartRead( uint8_t rxPin );
uint8_t       getUartBuffer( uint8_t rxPin, uint8_t *buf, uint8_t bufLen );

//----------------------------------------------------------------------------------------------------------
// I2C management routines.
//
//----------------------------------------------------------------------------------------------------------
uint8_t       configureI2C( uint8_t sclPin, uint8_t sdaPin, uint32_t baudRate = 100 * 1000 );
uint8_t       i2cWrite( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit = false );
uint8_t       i2cRead( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit = false );

//----------------------------------------------------------------------------------------------------------
// SPI management routines.
//
//----------------------------------------------------------------------------------------------------------
uint8_t       configureSPI( uint8_t sclkPin, uint8_t mosiPin, uint8_t misoPin, uint32_t baudRate = 10 * 1000 * 1000 );
uint8_t       spiBeginTransaction( uint8_t sclkPin, uint8_t csPin );
uint8_t       spiEndTransaction( uint8_t sclkPin, uint8_t csPin );
uint8_t       spiRead( uint8_t sclkPin, uint8_t *buf, uint32_t len );
uint8_t       spiWrite( uint8_t sclkPin, uint8_t *buf, uint32_t len );

};

#endif
```

```
//------------------------------------------------------------------------------------------------------------
//
// LCS - Controller dependent code Layer - Raspberry PI Pico Implementation
//
//------------------------------------------------------------------------------------------------------------
// This source file contains the the RP2040 controller family hardware library code. The idea of this library
// is to shield the actual hardware of processor and board implementation from the upper layers but still keep
// the flexibility and performance of the underlying hardware. The library works with the concept of HW pins,
// which are identifiers for an HW entity. This is easy for a GPIO pin, where the mapping is directly one to
// one. For more complex HW entries such as the I2C or UART hardware, one pin is selected as the identifier to
// that entity. For each complex entity an instance variable is maintained where all the relevant data is kept.
//
// A historic note. The original LCS code was written for Atmega and Pico. With the complete shift to PICO,
// the CDC library just serves as a simple interface to the PICO functions. One day, we may see more different
// controllers and controller families. The idea is that the LCS runtime is shielded from them.
//
//------------------------------------------------------------------------------------------------------------
//
// LCS - Controller Dependent Code - Raspberry PI Pico Implementation
// Copyright (C) 2022 - 2024 Helmut Fieres
//
// This program is free software: you can redistribute it and/or modify it under the terms of the GNU General
// Public License as published by the Free Software Foundation, either version 3 of the License, or (at your
// option) any later version.
//
// This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
// implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
// for more details.
//
// You should have received a copy of the GNU General Public License along with this program. If not, see
// http://www.gnu.org/licenses
//
//  GNU General Public License:  http://opensource.org/licenses/GPL-3.0
//
//------------------------------------------------------------------------------------------------------------
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

#include "pico/stdlib.h"
#include "pico/stdio.h"
#include "tusb_config.h"
#include "hardware/regs/usb.h"
#include "hardware/regs/rosc.h"
#include "hardware/regs/addressmap.h"
#include "hardware/clocks.h"
#include "hardware/gpio.h"
#include "hardware/adc.h"
#include "hardware/pwm.h"
#include "hardware/uart.h"
#include "hardware/i2c.h"
#include "hardware/spi.h"

#include "LcsCdcLib.h"

//------------------------------------------------------------------------------------------------------------
// Local name space. This file has two sections. The first is this local name space with all internal
// variables and routines local to the file. The second part contains the exported routines to be called by
// the core library and the firmware designers that need access to the underlying HW portion managed by this
// lowest layer.
//
//------------------------------------------------------------------------------------------------------------
namespace {

using namespace CDC;

//------------------------------------------------------------------------------------------------------------
// "CDC_DEBUG" is the local define for printing debug information. In contrast to the rest of the debugging
// and tracing of LCS libraries and programs, this library will have to be recompiled to enable debugging.
//
//------------------------------------------------------------------------------------------------------------
#define CDC_DEBUG 0

//------------------------------------------------------------------------------------------------------------
// Debug and Trace support. Instead of conditional compilation, we will print debug messages based on the
// setting of the debug level.
//------------------------------------------------------------------------------------------------------------
uint8_t debugLevel = 0;

//------------------------------------------------------------------------------------------------------------
// The CDC Library version data.
//
//------------------------------------------------------------------------------------------------------------
const uint8_t CDC_LIB_MAJOR_VERSION = 1;
const uint8_t CDC_LIB_MINOR_VERSION = 0;

//------------------------------------------------------------------------------------------------------------
// Valid pin mapping for the Raspberry PI Pico board. We construct a set of bitmask for the pin numbers.
// Pin Numbers range from 0 to 28. The bitmasks specify wether a pin can be assigned to the hardware type
// purpose. During configuration of a CDC function, the pins are checked against these bitmasks. All pins
// can be used as GPIO pins or PWM pins. All other hardware functions are bound to dedicated pins. Note
// that we do not check for assigning a pin to several different hardware functions. All we check is that
// the pin can be used for the desired purpose. A check performed by the CDC library routines is simply
// done through:
//
//    if (( 1 <<  pin ) & VALID_xxx )
//
//------------------------------------------------------------------------------------------------------------
```

```
const uint8_t  MAX_PIN_NUM          = 28;

const uint32_t VALID_GPIO_PINS      = 0x1FFFFFFF;
const uint32_t VALID_PWM_PINS       = 0x1FFFFFFF;
const uint32_t VALID_ADC_PINS       = ( 1 << 26 ) | ( 1 << 27 ) | ( 1 << 28 );

const uint32_t VALID_I2C_0_SDA_PINS = ( 1 << 0  ) | ( 1 << 4  ) | ( 1 << 8  ) |
                                      ( 1 << 12 ) | ( 1 << 16 ) | ( 1 << 20 );
const uint32_t VALID_I2C_0_SCL_PINS = ( 1 << 1  ) | ( 1 << 5  ) | ( 1 << 9  ) |
                                      ( 1 << 13 ) | ( 1 << 17 ) | ( 1 << 21 );

const uint32_t VALID_I2C_1_SDA_PINS = ( 1 << 2  ) | ( 1 << 6  ) | ( 1 << 10 ) |
                                      ( 1 << 14 ) | ( 1 << 18 ) | ( 1 << 26 );
const uint32_t VALID_I2C_1_SCL_PINS = ( 1 << 3  ) | ( 1 << 7  ) | ( 1 << 11 ) |
                                      ( 1 << 15 ) | ( 1 << 19 ) | ( 1 << 27 );

const uint32_t VALID_UART_0_TX_PINS = ( 1 << 0  ) | ( 1 << 12 ) | ( 1 << 16 );
const uint32_t VALID_UART_0_RX_PINS = ( 1 << 1  ) | ( 1 << 13 ) | ( 1 << 17 );

const uint32_t VALID_UART_1_TX_PINS = ( 1 << 4  ) | ( 1 << 8  );
const uint32_t VALID_UART_1_RX_PINS = ( 1 << 5  ) | ( 1 << 9  );

const uint32_t VALID_SPI_0_SCK_PINS = ( 1 << 2  ) | ( 1 << 6  ) | ( 1 << 18 );
const uint32_t VALID_SPI_0_TX_PINS  = ( 1 << 3  ) | ( 1 << 7  ) | ( 1 << 19 );
const uint32_t VALID_SPI_0_RX_PINS  = ( 1 << 0  ) | ( 1 << 4  ) | ( 1 << 16 );

const uint32_t VALID_SPI_1_SCK_PINS = ( 1 << 10 ) | ( 1 << 14 );
const uint32_t VALID_SPI_1_TX_PINS  = ( 1 << 11 ) | ( 1 << 15 );
const uint32_t VALID_SPI_1_RX_PINS  = ( 1 << 8  ) | ( 1 << 12 );

const uint32_t VALID_I2C_0_PINS     = VALID_I2C_0_SDA_PINS | VALID_I2C_0_SCL_PINS;
const uint32_t VALID_I2C_1_PINS     = VALID_I2C_1_SDA_PINS | VALID_I2C_1_SCL_PINS;

const uint32_t VALID_UART_0_PINS    = VALID_UART_0_TX_PINS | VALID_UART_0_RX_PINS;
const uint32_t VALID_UART_1_PINS    = VALID_UART_1_TX_PINS | VALID_UART_1_RX_PINS;

const uint32_t VALID_SPI_0_PINS     = VALID_SPI_0_SCK_PINS | VALID_SPI_0_TX_PINS | VALID_SPI_0_RX_PINS;
const uint32_t VALID_SPI_1_PINS     = VALID_SPI_1_SCK_PINS | VALID_SPI_1_TX_PINS | VALID_SPI_1_RX_PINS;

//----------------------------------------------------------------------------------------------
// Characteristics of the Raspberry Pi Pico and some key constants for the CDC library.
//
//----------------------------------------------------------------------------------------------
const uint16_t CONTROLLER_FAMILY          = CDC::CF_RP_PICO;
const uint32_t CHIP_MEM_SIZE              = 264 * 1024;
const uint32_t CHIP_NVM_SIZE              = 0;

const uint16_t ADC_DIGIT_RANGE            = 1024;
const uint16_t ADC_REF_VOLTAGE_MILLI_VOLT = 3300;

const uint8_t  MAX_UART_BUF_SIZE          = 8;

const uint32_t I2C_FREQUENCY              = 100 * 1000;
const uint32_t I2C_TIME_OUT_IN_MS         = 250;

const uint32_t SPI_FREQUENCY              = 10000000L;

const uint16_t  MAX_CPU_CORE              = 2;
const uint16_t  MAX_INT_PIN               = 24;

//------------------------------------------------------------------------------------------------
// An ADC instance. The PICO supports up to three ADC inputs. When we use such an input, the corresponding
// instance data is kept in this structure. We also keep the PICO ADC number, so we can select the correct
// instance.
//
//------------------------------------------------------------------------------------------------
struct AdcInst {

    bool       configured   = false;
    uint8_t    adcPin       = CDC::UNDEFINED_PIN;
    uint8_t    adcNum       = 0;
};

//------------------------------------------------------------------------------------------------
// A PWM output instance. GPIO pins can also be used as PWM output pins. The PWM output related data is
// kept in this instance.
//
//------------------------------------------------------------------------------------------------
struct PwmInst {

    bool       configured   = false;
    uint8_t    pwmPin       = CDC::UNDEFINED_PIN;
    uint32_t   wrap         = 0;

    // ??? what else to keep around ?
};

//------------------------------------------------------------------------------------------------
// A UART instance. UARTS are used to read in a serial stream from the RailCom detectors. There can be two
// hardware based UART instances, or up to four software defined instances. The instance also keeps a small
// buffer where the data is read into. We also keep the PICO UART HW instance used.
//
//------------------------------------------------------------------------------------------------
struct UartInst {

    bool                configured    = false;
    uint8_t             rxPin         = CDC::UNDEFINED_PIN;
    uint8_t             txPin         = CDC::UNDEFINED_PIN;
    uint16_t            baudSetting   = 0;
```

```
    uint8_t           dataBits      = 8;
    uart_parity_t     parityMode    = UART_PARITY_NONE;
    uint8_t           stopBits      = 1;
    int               uartIrq       = 0;
    uint8_t           uartMode      = 0;

    volatile uint8_t  rxBufIndex    = 0;
    volatile uint8_t  rxDataBuf[ MAX_UART_BUF_SIZE ] = { 0 };

    uart_inst_t       *uartHw       = nullptr;
};

//----------------------------------------------------------------------------------------------------------
// The I2C instance. The PICO features two HW instances of an I2C port. The instance data contains the
// assigned GPIO pins, the baud rate and a timeout. We also keep the I2C HW instance used.
//
//----------------------------------------------------------------------------------------------------------
struct I2CInst {

    bool        configured    = false;
    uint8_t     sclPin        = CDC::UNDEFINED_PIN;
    uint8_t     sdaPin        = CDC::UNDEFINED_PIN;
    uint32_t    baudRate      = I2C_FREQUENCY;
    uint32_t    timeoutValMs  = I2C_TIME_OUT_IN_MS;

    i2c_inst_t  *i2cHw        = nullptr;
};

//----------------------------------------------------------------------------------------------------------
// The SPI instance. The PICO features two SPI HW instances. We keep the assigned GPIO pins for the SPI
// interface as well as the PICO HW instance. Since the SPI protocol explicitly sets the selected HW select
// pin, we remember that we are in a transaction with perhaps more than one call to the SPI routines.
//
//----------------------------------------------------------------------------------------------------------
struct SPIInst {

    bool        configured  = false;
    bool        active      = false;
    uint8_t     selectPin   = CDC::UNDEFINED_PIN;
    uint8_t     mosiPin     = CDC::UNDEFINED_PIN;
    uint8_t     misoPin     = CDC::UNDEFINED_PIN;
    uint8_t     sclkPin     = CDC::UNDEFINED_PIN;
    uint32_t    frequency   = SPI_FREQUENCY;

    spi_inst_t  *spiHw      = nullptr;
};

//----------------------------------------------------------------------------------------------------------
// The interrupt table for the GPIO pin interrupts. The PICO can have only one interrupt handler. We will
// allocate a table where a handler can be set for each pin. When an interrupt comes in and there is a
// handler configured, it will be called.
//
//----------------------------------------------------------------------------------------------------------
struct GpioIsrTable {

    uint16_t                    numOfHandlers = 0;
    CDC::GpioCallback           gpioIsrTable[ MAX_CPU_CORE ][ MAX_INT_PIN + 1 ];
};

//----------------------------------------------------------------------------------------------------------
// Local variables. We maintain an instance variable for each of the possible HW entities, such as an I2C
// interface or a UART. Note that not all are used at the same time. The instance variables map from the
// simple pin numbers to the PICO structures and whatever else we need to remember for this entity.
//
//----------------------------------------------------------------------------------------------------------
CDC::CdcConfigDesc           cfg;
CDC::TimerCallback           timerCallback = nullptr;
GpioIsrTable                 cdcIntHandlers;
repeating_timer_t            timerData;
AdcInst                      CdcAdc0;
AdcInst                      CdcAdc1;
AdcInst                      CdcAdc2;
AdcInst                      CdcAdc3;
I2CInst                      CdcI2C0;
I2CInst                      CdcI2C1;
SPIInst                      CdcSPI0;
SPIInst                      CdcSPI1;
UartInst                     CdcUart0;
UartInst                     CdcUart1;
UartInst                     CdcUart2;
UartInst                     CdcUart3;
PwmInst                      CdcPwm0;
PwmInst                      CdcPwm1;
PwmInst                      CdcPwm2;
PwmInst                      CdcPwm3;

//----------------------------------------------------------------------------------------------------------
// "validPin" is called to check that a pin is in the correct number range, defined and matches the bitmask
// for the desired purpose. For example, configuring an I2C port will check that the two GPIO pins are
// indeed routable to the I2C HW block in the PICO.
//
//----------------------------------------------------------------------------------------------------------
bool validPin( uint8_t pin, uint32_t mask ) {

    if ( pin == CDC::UNDEFINED_PIN )  return ( true );
    if ( pin > MAX_PIN_NUM )          return ( false );
    return (( 1 << pin ) & mask );
}

```

```
//----------------------------------------------------------------------------------------------------
// When no interrupt is configured for a GPIO pin, we set the table entry to a dummy handler. This way
// we do not have to check for a valid procedure label when we handle an interrupt.
//
//----------------------------------------------------------------------------------------------------
void dummyIsrHandler ( uint8_t pin, uint8_t event ) { }

//----------------------------------------------------------------------------------------------------
// Setup the ISR table. The PICO can have only one interrupt handler. When you want a handler per GPIO pin,
// the solution is to have a table when you keep the handler on a per pin base.
//
//----------------------------------------------------------------------------------------------------
void initIsrTable( ) {

    for ( uint16_t i = 0; i < MAX_CPU_CORE; i++ ) {

        for ( uint16_t j = 0; j < MAX_INT_PIN; j++ ) {

            cdcIntHandlers.gpioIsrTable[ i ][ j ] = dummyIsrHandler;
        }
    }
}

//----------------------------------------------------------------------------------------------------
// The PICO uses a set of constants to describe the interrupt type. We map our interrupt types to the PICO
// GPIO_IRQ_xxx types.
//
//----------------------------------------------------------------------------------------------------
uint32_t mapGpioIntEvent( uint8_t event ) {

    switch ( event ) {

        case CDC::EVT_LOW:     return( GPIO_IRQ_LEVEL_LOW );
        case CDC::EVT_HIGH:    return( GPIO_IRQ_LEVEL_HIGH );
        case CDC::EVT_FALL:    return( GPIO_IRQ_EDGE_FALL );
        case CDC::EVT_RISE:    return( GPIO_IRQ_EDGE_RISE );
        case CDC::EVT_CHANGE:  return( GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL );
        default:      return( 0 );
    }
}

//----------------------------------------------------------------------------------------------------
// The PICO uses a set of constants to describe the interrupt type. We map them to our types.
//
//----------------------------------------------------------------------------------------------------
uint8_t mapPicoGpioEvent( uint32_t event ) {

    switch ( event ) {

        case GPIO_IRQ_LEVEL_LOW:  return( CDC::EVT_LOW );
        case GPIO_IRQ_LEVEL_HIGH: return( CDC::EVT_HIGH );
        case GPIO_IRQ_EDGE_FALL:  return( CDC::EVT_FALL );
        case GPIO_IRQ_EDGE_RISE:  return( CDC::EVT_RISE );
        default:                  return( 0 );
    }
}

//----------------------------------------------------------------------------------------------------
// Global Interrupt handlers. The hardware and low level library will call these handlers, which in turn
// will invoke the respective callback function if configured. The GPIO interrupt handler manages the
// handler for all possible IO pins. The PICO can only have one interrupt routine, so we feature an array
// of handlers where a handler for a GPIO pin can be registered. If there is a handler set, we just invoke
// it. The other handlers are for the timer and the UART hardware.
//
//----------------------------------------------------------------------------------------------------
void gpioCallback( uint gpioPin, uint32_t event ) {

    cdcIntHandlers.gpioIsrTable[ get_core_num( )][ gpioPin] ( gpioPin, mapPicoGpioEvent( event ));
}

bool repeatingTimerAlarm( repeating_timer_t *rt ) {

    if ( timerCallback != nullptr ) timerCallback((uint32_t) ( - timerData.delay_us ));
    return ( true );
}

void uartRxCallback0( ) {

    while ( uart_is_readable( uart0 )) {

        uint8_t ch = uart_getc( uart0 );
        if ( CdcUart0.rxBufIndex < MAX_UART_BUF_SIZE ) CdcUart0.rxDataBuf[CdcUart0.rxBufIndex++ ] = ch;
    }
}

void uartRxCallback1( ) {

    while ( uart_is_readable( uart1 )) {

        uint8_t ch = uart_getc( uart1 );
        if ( CdcUart1.rxBufIndex < MAX_UART_BUF_SIZE ) CdcUart1.rxDataBuf[ CdcUart1.rxBufIndex++ ] = ch;
    }
}

//----------------------------------------------------------------------------------------------------
// The default configuration descriptor. The Application program fills in such a structure, which can be
// seen as the HW pin assignments for the PICO controllers and the particular board on which the application
// will be deployed. The application will simply use the field names to address the particular PICO HW
// function. For example, a configuration has mapped DIO_PIN_5 to GPIO pin 12, because that is where the
```

```
// particular board has mapped DIO_PIN_5 to the hardware line. The application will just use the DIO_PIN_5
// field when talking to that GPIO pin. Whenever the board layout changes, there could be another PICO GPIO
// pin, but the name "DIO_PIN_5" for the application upper layers does not change.
//
// Note that there is a great flexibility what a PICO HW  pin can do and hence a lot of our fields are just
// "UNDEFINED" with no constraints. Nevertheless, there is a function which will do some plausibility checks
// for such a structure. Also, each configuration routine will do again a check that the GPIO pins used do
// indeed map to a PICO HW block for the desired purpose.
//
// The configuration structure does not replace the actual configuration calls to make to the CDC library.
// It is just a mapping of reserved names to actual GPIO pins.
//
//-----------------------------------------------------------------------------------------------------------
CDC::CdcConfigDesc getConfigDefaultRP2040( ) {

    CDC::CdcConfigDesc tmp;

    tmp.CFG_STATUS          = CDC::INIT_PENDING;

    // ??? controller family ?
    // ??? what other characteristics ? ( e.g. mem size ? )

    tmp.READY_LED_PIN       = CDC::UNDEFINED_PIN;
    tmp.ACTIVE_LED_PIN      = CDC::UNDEFINED_PIN;

    tmp.EXT_INT_PIN         = CDC::UNDEFINED_PIN;
    tmp.PFAIL_PIN           = CDC::UNDEFINED_PIN;

    tmp.DIO_PIN_0           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_1           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_2           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_3           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_4           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_5           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_6           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_7           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_8           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_9           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_10          = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_11          = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_12          = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_13          = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_14          = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_15          = CDC::UNDEFINED_PIN;

    tmp.ADC_PIN_0           = CDC::UNDEFINED_PIN;
    tmp.ADC_PIN_1           = CDC::UNDEFINED_PIN;
    tmp.ADC_PIN_2           = CDC::UNDEFINED_PIN;
    tmp.ADC_PIN_3           = CDC::ILLEGAL_PIN;

    tmp.PWM_PIN_0           = CDC::UNDEFINED_PIN;
    tmp.PWM_PIN_1           = CDC::UNDEFINED_PIN;
    tmp.PWM_PIN_2           = CDC::UNDEFINED_PIN;
    tmp.PWM_PIN_3           = CDC::UNDEFINED_PIN;

    tmp.UART_RX_PIN_0       = CDC::UNDEFINED_PIN;
    tmp.UART_TX_PIN_0       = CDC::UNDEFINED_PIN;

    tmp.UART_RX_PIN_1       = CDC::UNDEFINED_PIN;
    tmp.UART_TX_PIN_1       = CDC::UNDEFINED_PIN;

    tmp.UART_RX_PIN_2       = CDC::UNDEFINED_PIN;
    tmp.UART_TX_PIN_2       = CDC::UNDEFINED_PIN;

    tmp.UART_RX_PIN_3       = CDC::UNDEFINED_PIN;
    tmp.UART_TX_PIN_3       = CDC::UNDEFINED_PIN;

    tmp.SPI_MOSI_PIN_0      = CDC::UNDEFINED_PIN;
    tmp.SPI_MISO_PIN_0      = CDC::UNDEFINED_PIN;
    tmp.SPI_SCLK_PIN_0      = CDC::UNDEFINED_PIN;

    tmp.SPI_MOSI_PIN_1      = CDC::UNDEFINED_PIN;
    tmp.SPI_MISO_PIN_1      = CDC::UNDEFINED_PIN;
    tmp.SPI_SCLK_PIN_1      = CDC::UNDEFINED_PIN;

    tmp.NVM_I2C_SCL_PIN     = CDC::UNDEFINED_PIN;
    tmp.NVM_I2C_SDA_PIN     = CDC::UNDEFINED_PIN;

    tmp.EXT_I2C_SCL_PIN     = 17;
    tmp.EXT_I2C_SDA_PIN     = 16;

    tmp.CAN_BUS_RX_PIN      = CDC::UNDEFINED_PIN;
    tmp.CAN_BUS_TX_PIN      = CDC::UNDEFINED_PIN;

    return ( tmp );
}

//-----------------------------------------------------------------------------------------------------------
// Validate a configuration structure. This routine will do basic checking of the pin configuration passed.
// The PICO is very flexible when it comes to what a pin can do. However, there are still some rules to
// follow. Also, we have dedicated settings for at least the I2C channels and the CAN bus IO pins.
//
//-----------------------------------------------------------------------------------------------------------
uint8_t validateConfigRP20040( CDC::CdcConfigDesc *ci ) {

    // ??? a ton of "validXXX" ?

    return ( NO_ERR ); // for now....
}
```

```
}; // namespace


//----------------------------------------------------------------------------------------------------------
// Bane CDC. All routines and definitions exported are in this name space.
//
//----------------------------------------------------------------------------------------------------------
namespace CDC {

//----------------------------------------------------------------------------------------------------------
// For debugging purposes. Instead of conditional compilations, the debug level will enable the printing of
// debug and trace data.
//
//----------------------------------------------------------------------------------------------------------
void setDebugLevel( uint8_t level ) {

    debugLevel = level;
}

//----------------------------------------------------------------------------------------------------------
// "getConfigDefault" initializes a configuration structure and sets the pre-assigned values. A typical
// sequence for an application start sequence would be to create an initial structure this way and then set
// the relevant pins and values according to the actual hardware configuration.
//
//----------------------------------------------------------------------------------------------------------
CdcConfigDesc getConfigDefault( ) {

   return ( getConfigDefaultRP2040( ));
}

//----------------------------------------------------------------------------------------------------------
// "getConfigActual" will return a pointer to the copy we kept when calling the init routine with the config
// structure to use. There is no need for the upper layers to keep the structure used at initialization time.
//
//----------------------------------------------------------------------------------------------------------
CdcConfigDesc *getConfigActual( ) {

   return ( &cfg );
}

//----------------------------------------------------------------------------------------------------------
// CDC library setup. The "init" routine will ready the CDC library. The main task is to validate the pins and
// values for the particular controller capabilities. The init routine can be called more than once without a
// problem.
//
//----------------------------------------------------------------------------------------------------------
uint8_t init( CdcConfigDesc *ci ) {

    cfg = *ci;

    initIsrTable( );
    configureConsoleIO( );

    return ( validateConfigRP20040( ci ));
}

//----------------------------------------------------------------------------------------------------------
// "fatalError" is the error communication method when we cannot get anything to work, except the onboard
// LED. The Raspberry Pi PICO has a small Led on the board. We will use this LED to "blink" an error code.
// There are up to eight codes. The sequence is as follows:
//
//     repeat forever:
//
//     - 1s ON, 0.5s OFF
//     - for ( int i = 0; i < n; i++ ) { 0.5s ON; 0.5s OFF; }
//
// The only way to get out of this loop is then to reset the board. Fatal errors are hopefully not many. One
// obvious one is when we cannot detect the NVM and thus know nothing about the board.
//
//----------------------------------------------------------------------------------------------------------
void fatalError( uint8_t n ) {

    const uint8_t   ledPin      = 25;
    const uint32_t  longPulse   = 1000;
    const uint32_t  shortPulse  = 250;

    n = n % 8;

    gpio_init( ledPin );
    gpio_set_dir( ledPin, GPIO_OUT );

    while ( true ) {

        sleep_ms( longPulse );

        for ( int i = 0; i < n; i++ ) {

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

//----------------------------------------------------------------------------------------------------------
// "fatalErrorMsg" will result in a fatal error, but we attempt to first write an error message to the
// console.
```

```
//
//----------------------------------------------------------------------------------------------------
void fatalErrorMsg( char *str, uint8_t n, uint8_t rStat ) {

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

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

    return ( CONTROLLER_FAMILY );
}

uint32_t getVersion( ) {

    return ( CDC_LIB_MAJOR_VERSION << 8 | CDC_LIB_MINOR_VERSION );
}

uint32_t getChipMemSize( ) {

    return ( CHIP_MEM_SIZE );
}

uint32_t getChipNvmSize( ) {

    return ( CHIP_NVM_SIZE   );
}

uint32_t getCpuFrequency( ) {

    return ( clock_get_hz( clk_sys ));
}

uint32_t getMillis( ) {

    return ( to_ms_since_boot( get_absolute_time( )));
}

uint32_t getMicros( ) {

    return ( to_us_since_boot( get_absolute_time( )));
}

void sleepMillis( uint32_t val ) {

    sleep_ms( val );
}

void sleepMicros( uint32_t val ) {

    sleep_us( val );
}

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

    uint32_t rVal = 0;

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

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

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

    return ( rVal );
}

//----------------------------------------------------------------------------------------------------
// Console IO section. We set up the stdio via the USB connector. As part of the CDC init call, the configure
// call should be done rather early, so that we can print out debug messages. In normal LCS node operation
// there is no USB connected. Detecting a connection helps to decide whether we can report an error or need
// to resort to a fatal error call at startup.
//
// There are two basic ways to detect an USB connection. The first is to simply check if there is power on
// the USB port. The PICO features an internal GPIO pin for this purpose. Using this method still does not
// mean that we have someone connected to the USB, but just that there is a cable with power. Well, good
// enough for us. The second method truly detects that there is a USB host connected. This check is provided
// via the PICO libraries which in turn use the tinyUSB library. However, there could be a timing problem
// where the USB stack is not ready and we conclude wrongly that there is no USB connection. For now, let's
// rather go with the risk that there is just power on the USB connector.
//
// Finally, there is a routine to get a character for the command interfaces. Since the function just reads
// in a character, optionally with a timeout how ling to wait for any inout.
//
// PS: The USB check way would be "return( stdio_usb_connected( ));" instead of the GPIO check.
//----------------------------------------------------------------------------------------------------
uint8_t configureConsoleIO( ) {

```

```
    stdio_init_all( );
    return( NO_ERR );
}

bool isConsoleConnected( ) {

    gpio_init( PICO_VBUS_PIN );
    gpio_set_dir( PICO_VBUS_PIN, GPIO_IN );

    return( gpio_get( PICO_VBUS_PIN ));
}

char getConsoleChar( uint32_t timeoutVal ) {

    int ch = getchar_timeout_us( timeoutVal );
    return(( ch == PICO_ERROR_TIMEOUT ) ? 0 : ch );
}

//----------------------------------------------------------------------------------------------------------
// Timer section. The CDC library features one generic repeating timer with a microsecond resolution. The
// routines start and stop the timer and allow to set a new limit. The PICO offers a high level function that
// schedules a repeating timer with the property of measuring the interval also from the start of the
// callback invocation. This is exactly what we need to implement the tick interrupt for the DCC signal state
// machine. The "setRepeatingTimerLimit" function will adjust the timer limit counter while the timer already
// is counting toward a limit. Note that the timer option that already start the next round while the timer
// interrupt handler executes is specified by using negative limit values. The timer functionality also
// offers two timestamp routines to get the number of milliseconds and number of microseconds since system
// start.
//
// ??? would we one day need more than one timer instance ?
//----------------------------------------------------------------------------------------------------------
void startRepeatingTimer( uint32_t val ) {

    int64_t limit = val;
    add_repeating_timer_us( - limit, repeatingTimerAlarm, nullptr, &timerData );
}

void stopRepeatingTimer( ) {

    cancel_repeating_timer( &timerData );
}

uint32_t getRepeatingTimerLimit( ) {

    return ((uint32_t) ( - timerData.delay_us ));
}

void setRepeatingTimerLimit( uint32_t val ) {

    int64_t limit = val;
    timerData.delay_us = ((int64_t) - limit );
}

void onTimerEvent( CDC::TimerCallback functionId ) {

    timerCallback = functionId;
}

//----------------------------------------------------------------------------------------------------------
// DIO section. A digital pin is the bread and butter hardware resource and can be an input or output pin. For
// inputs, an internal pull-up resistor can be set.There are a couple of interfaces. First the single pin
// read, write and toggle. Next are read and write mask routines which work on all IO pins at once. Note that
// no cross checking is done if the pins are used by other CDC functions. Finally there is a convenience
// routine which write a pair of data. This is typically used for the H-Bridge control pins, which are set at
// the same time.
//
// A GPIO pin can also have an attached interrupt handler. When we register a handler for a pin, there are
// two different PICO lib routines to use. When there is no handler registered so far, we register the
// common callback and store the particular GPIO handler in the handler table. Otherwise, we just store the
// handler and enable the GPIO pin for interrupts.
//
//----------------------------------------------------------------------------------------------------------
uint8_t configureDio( uint8_t dioPin, uint8_t mode ) {

    if ( ! validPin( dioPin, VALID_GPIO_PINS )) return ( DIO_PIN_ERR );

    gpio_init( dioPin );

    switch ( mode ) {

        case IN:  gpio_set_dir( dioPin, false );  break;
        case OUT: {

            gpio_set_dir( dioPin, true );
            gpio_set_drive_strength ( dioPin, GPIO_DRIVE_STRENGTH_12MA );

        }  break;

        case IN_PULLUP: {

            gpio_set_dir( dioPin, false );
            gpio_pull_up( dioPin );

        } break;

        default: gpio_set_dir( dioPin, false );
    }

  return ( NO_ERR );
```

```
}

void registerDioCallback( uint8_t dioPin, uint8_t event, CDC::GpioCallback func ) {

    if ( dioPin <= MAX_INT_PIN ) {

        if ( cdcIntHandlers.numOfHandlers == 0 )
            gpio_set_irq_enabled_with_callback( dioPin, mapGpioIntEvent( event ), true, gpioCallback );
        else
            gpio_set_irq_enabled( dioPin, mapGpioIntEvent( event ), true);

        cdcIntHandlers.gpioIsrTable[ get_core_num( ) ][ dioPin ] = func;
        cdcIntHandlers.numOfHandlers ++;
    }
}

void unregisterDioCallback( uint8_t dioPin ) {

    if ( dioPin <= MAX_INT_PIN ) {

        if ( cdcIntHandlers.gpioIsrTable[ get_core_num( ) ][ dioPin ] != nullptr ) {

            gpio_set_irq_enabled( dioPin, 0, false );
            cdcIntHandlers.gpioIsrTable[ get_core_num( ) ][ dioPin ] = dummyIsrHandler;
            cdcIntHandlers.numOfHandlers --;
        }
    }
}

bool readDio( uint8_t dioPin ) {

    return ( gpio_get( dioPin ));
}

uint8_t writeDio( uint8_t dioPin, bool val ) {

    gpio_put( dioPin, val );
    return ( NO_ERR );
}

uint8_t toggleDio( uint8_t dioPin ) {

    writeDio( dioPin, ! readDio( dioPin ));
    return ( NO_ERR );
}

uint8_t writeDioPair( uint8_t dioPin1, bool val1, uint8_t dioPin2, bool val2 ) {

    uint32_t maskData = ( 1UL << dioPin1 ) | ( 1UL << dioPin2 );
    uint32_t valData  = (( val1 ) ? ( 1 << dioPin1 ) : 0 ) | (( val2 ) ? ( 1 << dioPin2 ) : 0 );

    gpio_put_masked( maskData, valData );
    return ( NO_ERR );
}

uint32_t readDioMask( uint32_t dioMask ) {

    return ( gpio_get_all( ) & dioMask );
}

uint8_t writeDioMask( uint32_t dioMask, uint32_t dioVal ) {

    gpio_put_masked( dioMask, dioVal );
    return ( NO_ERR );
}

//----------------------------------------------------------------------------------------------------
// ADC section. The analog input channel represented by the pin is configured. At initialization, the ADC pin
// number is validated and the ADC subsystem initialized. The PICO does an analog read in about 2us. This is
// so fast, it does for our purpose make not much sense to implement an asynchronous option. Furthermore, the
// ADC value scaled down to a 10-bit resolution.
//
//----------------------------------------------------------------------------------------------------
uint8_t configureAdc( uint8_t adcPin ) {

    if ( ! validPin( adcPin, VALID_ADC_PINS ))  return ( ADC_PIN_ERR );

    AdcInst *tmp = nullptr;

    if ( adcPin == cfg.ADC_PIN_0 ) {

        tmp = &CdcAdc0;
        tmp -> adcPin = adcPin;
        tmp -> adcNum = 0;

    }
     else if ( adcPin == cfg.ADC_PIN_1 ) {

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

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

```

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

    return ( NO_ERR );
}

uint16_t getAdcRefVoltage( ) {

    return ( ADC_REF_VOLTAGE_MILLI_VOLT );
}

uint16_t getAdcDigitRange( ) {

    return ( ADC_DIGIT_RANGE );
}

uint16_t readAdc( uint8_t adcPin ) {

    AdcInst *tmp = nullptr;

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

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

    UartInst *uart = nullptr;

    if ( mode == UART_MODE_8N1 ) {

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

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

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

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

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

        irq_set_enabled( uart -> uartIrq, true );

        return ( NO_ERR );
    }
    else if ( mode == UART_MODE_8N1_PIO ) {

        return ( NOT_SUPPORTED );
    }
    else return ( NOT_SUPPORTED );
}

uint8_t startUartRead( uint8_t rxPin ) {
```

```

    UartInst *uart = nullptr;

    if      ( rxPin == CdcUart0.rxPin ) uart = &CdcUart0;
    else if ( rxPin == CdcUart1.rxPin ) uart = &CdcUart1;
    else if ( rxPin == CdcUart2.rxPin ) uart = &CdcUart2;
    else if ( rxPin == CdcUart3.rxPin ) uart = &CdcUart3;
    else                                return ( CDC::UART_PORT_ERR );

    if (( uart != nullptr ) && ( uart -> uartMode == UART_MODE_8N1 )) {

        uart_set_irq_enables( uart -> uartHw, true, false );
        uart -> rxBufIndex = 0;
        return ( NO_ERR );
    }
    else if (( uart != nullptr ) && ( uart -> uartMode == UART_MODE_8N1_PIO )) {

        return ( NOT_SUPPORTED );
    }
    else return ( UART_PORT_ERR );
}

uint8_t stopUartRead( uint8_t rxPin ) {

    UartInst *uart = nullptr;

    if      ( rxPin == CdcUart0.rxPin ) uart = &CdcUart0;
    else if ( rxPin == CdcUart1.rxPin ) uart = &CdcUart1;
    else if ( rxPin == CdcUart2.rxPin ) uart = &CdcUart2;
    else if ( rxPin == CdcUart3.rxPin ) uart = &CdcUart3;

    if (( uart != nullptr ) && ( uart ->uartMode == UART_MODE_8N1 )) {

        uart_set_irq_enables( uart -> uartHw, false, false );
        return ( NO_ERR );
    }
    else if (( uart != nullptr ) && ( uart -> uartMode == UART_MODE_8N1_PIO )) {

        return ( NOT_SUPPORTED );
    }
    else return ( UART_PORT_ERR );
}

uint8_t getUartBuffer( uint8_t rxPin, uint8_t *buf, uint8_t bufLen ) {

    UartInst *uart = nullptr;

    if      ( rxPin == CdcUart0.rxPin ) uart = &CdcUart0;
    else if ( rxPin == CdcUart1.rxPin ) uart = &CdcUart1;
    else if ( rxPin == CdcUart2.rxPin ) uart = &CdcUart2;
    else if ( rxPin == CdcUart3.rxPin ) uart = &CdcUart3;
    else                                return ( 0 );

    if (( uart != nullptr ) && ( uart -> rxBufIndex > 0 ) && ( bufLen > 0 )) {

        uint8_t i = 0;
        while (( i < uart -> rxBufIndex ) && ( i < bufLen )) {

            buf[ i ] = uart -> rxDataBuf[ i ];
            i++;
        }

        return ( i );
    }
    else return ( 0 );
}

//-------------------------------------------------------------------------------------------------------
// PWM section. The PICO is quite flexible when it comes to PWM signals. We implement a simple PWM capability.
// There is the frequency which set during configuration and there is the write operation which set the duty
// cycle. The calculations are best described in the PICO C++ SDK. We do the setting of phase, wrap count,
// etc. once when we configure the PWM channel. All the "writePwm" function then will do is to manipulate the
// duty cycle. In other words, when we change the frequency we need to configure again.
//
// There is one small issue left. Channel come in pairs. For some reason there is no call to individually
// set the "inverted" option on a channel. When we set the inverted option for a pin, we currently also set
//  the inverted option for the other channel since we just don't know better. To be correct, all possible
// PWM pins and their "inverted" option would need to be stored somewhere.
//
// To do .... ( there is a way via the pwm_Config CSR field... )
//
//-------------------------------------------------------------------------------------------------------
uint8_t configurePwm( uint8_t pwmPin, uint32_t pwmFreqency, bool phaseCorrect, bool inverted ) {

    PwmInst *pwm = nullptr;

    if      ( pwmPin == cfg.PWM_PIN_0 ) pwm = &CdcPwm0;
    else if ( pwmPin == cfg.PWM_PIN_1 ) pwm = &CdcPwm1;
    else if ( pwmPin == cfg.PWM_PIN_2 ) pwm = &CdcPwm2;
    else if ( pwmPin == cfg.PWM_PIN_3 ) pwm = &CdcPwm3;
    else                                return ( PWM_PIN_ERR );

    if ( phaseCorrect ) pwmFreqency = pwmFreqency * 2;

    uint32_t sysClock = getCpuFrequency( );
    uint32_t clkDiv   = sysClock / pwmFreqency / 4096 + ( sysClock % ( pwmFreqency * 4096 ) != 0 );

    if ( clkDiv / 16 == 0 ) clkDiv = 16;

```

```
    pwm -> pwmPin  = pwmPin;
    pwm -> wrap    = sysClock * 16 / clkDiv / pwmFreqency - 1;

    pwm_config pwmConfig = pwm_get_default_config( );
    gpio_set_function( pwm -> pwmPin, GPIO_FUNC_PWM );
    pwm_config_set_wrap( &pwmConfig, pwm -> wrap );
    pwm_config_set_phase_correct( &pwmConfig, phaseCorrect );
    pwm_config_set_output_polarity( &pwmConfig, inverted, inverted );
    pwm_init ( pwm_gpio_to_slice_num( pwm -> pwmPin ), &pwmConfig, false );
    pwm_set_clkdiv_int_frac( pwm_gpio_to_slice_num( pwm -> pwmPin ), clkDiv / 16, clkDiv & 0xF );

    #if CDC_DEBUG == 1

    printf( "PWM Pin: % d, fPwm: % d, phase: % d, inverted: % d, clkDiv: % d, wrap: % d \n",
            pwm -> pwmPin, pwmFreqency,  phaseCorrect, inverted, clkDiv, pwm -> wrap );
    #endif

    return ( NO_ERR );
}

uint8_t writePwm( uint8_t pwmPin, uint8_t dutyCycle ) {

    PwmInst *pwm = nullptr;

    if      ( pwmPin == cfg.PWM_PIN_0 ) pwm = &CdcPwm0;
    else if ( pwmPin == cfg.PWM_PIN_1 ) pwm = &CdcPwm1;
    else if ( pwmPin == cfg.PWM_PIN_2 ) pwm = &CdcPwm2;
    else if ( pwmPin == cfg.PWM_PIN_3 ) pwm = &CdcPwm3;
    else                                return ( PWM_PIN_ERR );

    uint sliceNum = pwm_gpio_to_slice_num( pwmPin );
    uint channel  = pwm_gpio_to_channel( pwmPin );

    if ( dutyCycle == 0 ) {

        pwm_set_enabled( sliceNum, false );
        writeDio( pwmPin, false );
    }
    else if ( dutyCycle == 255 ) {

        pwm_set_enabled( sliceNum, false );
        writeDio( pwmPin, true );
    }
    else {

        pwm_set_chan_level( sliceNum, channel, ( pwm -> wrap * dutyCycle / 256 ));
        pwm_set_enabled( sliceNum, true );
    }

    return ( NO_ERR );
}

//----------------------------------------------------------------------------------------------------------
// I2C Section. The PICO has two HW blocks for I2C interfaces. The interface implements a simple read and
// write access to an I2C element. There is a timeout to avoid waiting forever on an operation.
//
//----------------------------------------------------------------------------------------------------------
uint8_t configureI2C( uint8_t sclPin, uint8_t sdaPin, uint32_t baudRate ) {

    I2CInst *i2c = nullptr;

    if ((( 1 << sclPin ) & VALID_I2C_0_SCL_PINS ) && (( 1 << sdaPin ) & VALID_I2C_0_SDA_PINS )) {

        i2c = &CdcI2C0;
        i2c -> i2cHw = i2c0;
    }
    else if ((( 1 << sclPin ) & VALID_I2C_1_SCL_PINS ) && (( 1 << sdaPin ) & VALID_I2C_1_SDA_PINS )) {

        i2c = &CdcI2C1;
        i2c -> i2cHw = i2c1;
    }
    else return ( CDC::I2C_PORT_ERR );

    i2c -> sclPin       = sclPin;
    i2c -> sdaPin       = sdaPin;
    i2c -> baudRate     = baudRate;
    i2c -> timeoutValMs = I2C_TIME_OUT_IN_MS;
    i2c -> configured   = true;

    i2c_init( i2c -> i2cHw, i2c -> baudRate );
    i2c_set_slave_mode( i2c -> i2cHw, false, 0 );

    gpio_set_function( i2c -> sclPin, GPIO_FUNC_I2C );
    gpio_set_function( i2c -> sdaPin, GPIO_FUNC_I2C);
    gpio_pull_up( i2c -> sclPin );
    gpio_pull_up( i2c -> sdaPin );

    return ( NO_ERR );
}

uint8_t i2cRead( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit ) {

    I2CInst *i2c = nullptr;

    if      (( CdcI2C0.sclPin == sclPin ) && ( CdcI2C0.configured )) i2c = &CdcI2C0;
    else if (( CdcI2C1.sclPin == sclPin ) && ( CdcI2C1.configured )) i2c = &CdcI2C1;
    else return ( I2C_PORT_ERR );

    auto ret = i2c_read_blocking_until( i2c -> i2cHw,
```

```
                                          i2cAdr,
                                          buf,
                                          len,
                                          stopBit,
                                          make_timeout_time_ms( i2c -> timeoutValMs ));

    #if CDC_DEBUG == 1
    printf( "i2cRead: scl: %d, i2c: 0x%x, buf: %p, buf[0] %x, buf[1] %x, len: %d, stop: %d\n",
               sclPin, i2cAdr, buf, buf[0], buf[1], len, stopBit );
    if ( ret == PICO_ERROR_GENERIC ) printf( "I2C read, PICO generic error\n" );
    if ( ret == PICO_ERROR_TIMEOUT ) printf( "I2C read, PICO timeout error\n" );
    #endif

    if (( ret == PICO_ERROR_GENERIC ) || ( ret == PICO_ERROR_TIMEOUT )) return ( I2C_READ_ERR );

    return ( NO_ERR );
}

uint8_t i2cWrite( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit ) {

    #if CDC_DEBUG == 1
    printf( "i2cWrite: scl: %d, i2c: 0x%x, buf: %p, buf[0] %x, buf[1] %x, len: %d, stop: %d\n",
             sclPin, i2cAdr, buf, buf[0], buf[1], len, stopBit );
    #endif


    I2CInst *i2c = nullptr;

    if      (( CdcI2C0.sclPin == sclPin ) && ( CdcI2C0.configured )) i2c = &CdcI2C0;
    else if (( CdcI2C1.sclPin == sclPin ) && ( CdcI2C1.configured )) i2c = &CdcI2C1;
    else return ( I2C_PORT_ERR );

    auto ret = i2c_write_blocking_until( i2c -> i2cHw,
                                           i2cAdr,
                                           buf,
                                           len,
                                           stopBit,
                                           make_timeout_time_ms( i2c -> timeoutValMs ));

    #if CDC_DEBUG == 1
    if ( ret == PICO_ERROR_GENERIC ) printf( "I2C write, PICO generic error\n" );
    if ( ret == PICO_ERROR_TIMEOUT ) printf( "I2C write, PICO timeout error\n" );
    #endif

    if (( ret == PICO_ERROR_TIMEOUT) || ( ret == PICO_ERROR_GENERIC ) || ( ret != len )) return ( I2C_WRITE_ERR );

    return ( NO_ERR );
}

//-------------------------------------------------------------------------------------------------------------
// SPI interface section. The PICO features two SPI HW blocks. We implement a simple SPI interface with a
// a fixed set of SPI options for frequency, bit order and mode. One day this may change.
//
// ??? we do not take care of the chip select stuff. Expect to put a chip select / deselect around the calls?
//-------------------------------------------------------------------------------------------------------------
uint8_t configureSPI( uint8_t sclkPin, uint8_t mosiPin, uint8_t misoPin, uint32_t baudRate ) {

    SPIInst *spi = nullptr;

    if (( CdcSPI0.sclkPin == sclkPin ) &&
        ( CdcSPI0.mosiPin == mosiPin ) &&
        ( CdcSPI0.misoPin == misoPin )) {

        spi = &CdcSPI0;
        spi -> spiHw = spi0;

    } else if (( CdcSPI1.sclkPin == sclkPin ) &&
                ( CdcSPI1.mosiPin == mosiPin ) &&
                ( CdcSPI1.misoPin == misoPin ) &&
                ( CdcSPI1.configured )) {

        spi = &CdcSPI1;
        spi -> spiHw = spi1;
    }
    else return ( SPI_PORT_ERR );

    spi -> mosiPin     = mosiPin;
    spi -> misoPin     = misoPin;
    spi -> sclkPin     = sclkPin;
    spi -> frequency   = SPI_FREQUENCY;
    spi -> configured  = true;
    spi -> active      = false;

    spi_init( spi -> spiHw, SPI_FREQUENCY );

    spi_set_format( spi -> spiHw,     // SPI instance
                    8,                // Number of bits per transfer
                    SPI_CPOL_1,       // Polarity (CPOL)
                    SPI_CPHA_1,       // Phase (CPHA)
                    SPI_MSB_FIRST );

    gpio_set_function( sclkPin, GPIO_FUNC_SPI );
    gpio_set_function( mosiPin, GPIO_FUNC_SPI );
    gpio_set_function( misoPin, GPIO_FUNC_SPI );

    return ( NO_ERR );
}

uint8_t  spiBeginTransaction( uint8_t sclkPin, uint8_t csPin ) {
```

```

    SPIInst *spi = nullptr;

    if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
    else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
    else return ( SPI_PORT_ERR );

    if ( spi -> active ) {

        // ??? should we check who is active and just ignore when the same ? else "error " ?

        return ( NO_ERR );

    } else {

        spi -> active     = true;
        spi -> selectPin  = csPin;

        CDC::writeDio( csPin, false );
        return ( NO_ERR );
    }
}

uint8_t spiEndTransaction( uint8_t sclkPin, uint8_t csPin ) {

    SPIInst *spi = nullptr;

    if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
    else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
    else return ( SPI_PORT_ERR );

    if ( spi -> active ) {

        // ??? check that this is the correct pin ?

        CDC::writeDio( csPin, true );

        spi -> active     = false;
        spi -> selectPin  = UNDEFINED_PIN;

        return ( NO_ERR );

    }
    else return ( NO_ERR ); // ???  "error "  not active...
}

uint8_t spiRead( uint8_t sclkPin, uint8_t *buf, uint32_t len ) {

    SPIInst *spi = nullptr;

    if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
    else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
    else return ( SPI_PORT_ERR );

    if ( spi -> active ) {

        int bytesRead = spi_read_blocking( spi -> spiHw, 0, buf, len );
        return ( NO_ERR );

    } else return ( NO_ERR ); // ??? fix : not active ...
}

uint8_t spiWrite( uint8_t sclkPin, uint8_t *buf, uint32_t len ) {

    SPIInst *spi = nullptr;

    if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
    else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
    else return ( SPI_PORT_ERR );

    if ( spi -> active ) {

        spi_write_blocking( spi -> spiHw, buf, len );
        return ( NO_ERR );

    } else return ( NO_ERR ); // ??? fix : not active ...
}

//------------------------------------------------------------------------------------------------------------
// Print out the Config Structure.
//
//------------------------------------------------------------------------------------------------------------
void printConfigInfo( CdcConfigDesc *ci ) {

    printf( "CDC Pin Configuration Info ( status %d ): \n", ci -> CFG_STATUS );

    printf( "Pfail pin: %2d, ExtInt pin: %2d \n", ci -> PFAIL_PIN, ci -> EXT_INT_PIN );

    printf( "ReadyLed pin: %2d, ActiveLed pin: %2d \n", ci -> READY_LED_PIN, ci -> ACTIVE_LED_PIN );

    printf( "DIO pins ( 0 .. 7 ): %2d %2d %2d %2d %2d %2d %2d %2d\n",
            ci -> DIO_PIN_0, ci -> DIO_PIN_1, ci -> DIO_PIN_2, ci -> DIO_PIN_3,
            ci -> DIO_PIN_4, ci -> DIO_PIN_5, ci -> DIO_PIN_6, ci -> DIO_PIN_7 );

    printf( "DIO pins ( 8 .. 15 ): %2d %2d %2d %2d %2d %2d %2d %2d\n",
            ci -> DIO_PIN_8, ci -> DIO_PIN_9, ci -> DIO_PIN_10, ci -> DIO_PIN_11,
            ci -> DIO_PIN_12, ci -> DIO_PIN_13, ci -> DIO_PIN_14, ci -> DIO_PIN_15 );

    printf( "ADC pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
```

```
            ci -> ADC_PIN_0 , ci -> ADC_PIN_1 , ci -> ADC_PIN_2 , ci -> ADC_PIN_3 );

    printf( "PWM pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
            ci -> PWM_PIN_0 , ci -> PWM_PIN_1 , ci -> PWM_PIN_2 , ci -> PWM_PIN_3 );

    printf( "UART RX pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
            ci -> UART_RX_PIN_0 , ci -> UART_RX_PIN_1 , ci -> UART_RX_PIN_2 , ci -> UART_RX_PIN_3 );

    printf( "UART TX pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
            ci -> UART_TX_PIN_0 , ci -> UART_TX_PIN_1 , ci -> UART_TX_PIN_2 , ci -> UART_TX_PIN_3 );

    printf( "SPI0 Pins: MOSI: %2d, MISO: %2d, SCLK: %2d \n",
            ci -> SPI_MOSI_PIN_0 , ci -> SPI_MISO_PIN_0 , ci -> SPI_SCLK_PIN_0 );

    printf( "SPI1 Pins: MOSI: %2d, MISO: %2d, SCLK: %2d \n",
            ci -> SPI_MOSI_PIN_1 , ci -> SPI_MISO_PIN_1 , ci -> SPI_SCLK_PIN_1 );

    printf( "NVM I2C Pins: SCL: %2d, SDA: %2d, I2C Root: 0x%x \n",
            ci -> NVM_I2C_SCL_PIN , ci -> NVM_I2C_SDA_PIN , ci -> NVM_I2C_ADR_ROOT );

    printf( "EXT I2C Pins: SCL: %2d, SDA: %2d, I2C Root: 0x%x \n",
            ci -> EXT_I2C_SCL_PIN , ci -> EXT_I2C_SDA_PIN , ci -> EXT_I2C_ADR_ROOT );

    printf( "\n" );

}

}; // namespace CDC
```