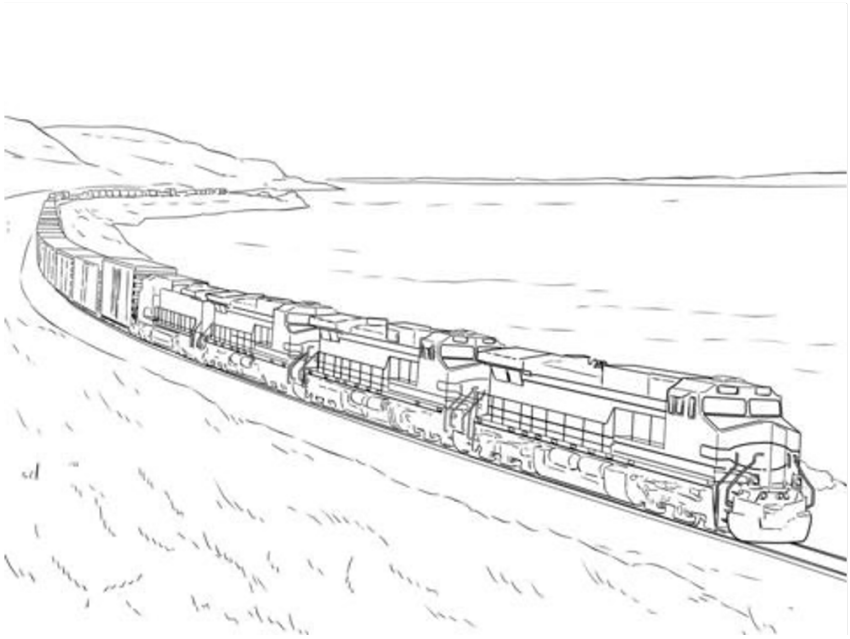


# A Layout Control System for Model Railroads

Helmut Fieres December 16, 2024

# Contents

# 1 LCS Hardware Module Design

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

- communication
- controller
- function block(s)

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

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

## 1.1 Selecting the controller

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

Meanwhile, the Raspberry PI Pico `(PICO)` controller joined the club. And it has a lot to offer. The PICO is a dual core controller running at up to 133 Mhz. It features a whopping 16Mbytes of flash and 264 Kbytes of main memory. There are plenty of

IO ports, and functional blocks for UARTS, SPI and I2C interfaces. What makes this controller especially interesting are the PIO state machines that allow for implementing your own I/O protocols. There is CAN bus software library built using these state machines. This way no extra CAN bus controller is needed. The PICO comes with its own software development kit and also an Arduino IDE integration is available.

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

## 1.2 The Controller Platform

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

Table 1.1: Controller Attributes

| **Attributes** | **Notes** |
|---|---|
| **Processor** | For a typical module, the PICO offers plenty in terms of CPU power. Since we use a software implementation for the CAN bus, running the software in one core and the CAN bus state machine in the other will well match what the PICO offers. |
| **Memory** | Memory depends on the size requirements of the node, port and event maps and the node-specific firmware data demands. A simple module would perhaps get by with 2Kb, a base station could easily use 32Kb or even more. |
| **Program Memory** | The LCS library already uses round about 64Kb of code storage. A simple module would get by with 32Kb, a base station could easily use 128Kb and more. |
| **External NVM** | Additional NVM storage is allocated in a separate EEPROM or FRAM. The capacity is highly dependent on the module use case. External NVM components typically also require the SPI or I2C interface. Most external EEPROM chips have write cycles of more than a million. At a minimum, a chip size of 32Kb is recommended. The PICO does not offer an internal EEPROM, so an external NVM is always required. |
| **Digital channels** | The bulk of control lines is digital and used heavily. For some hardware modules, a subset of the digital pins should also be PWM capable. |

*Continued on next page*

| Attributes | Notes |
|---|---|
| **Analog channels** | Analog input is typically used for the power module for analog voltage measurements. Otherwise, it is perhaps optional. The PICO allows for only three inputs. If more are desired, an external multiplexer needs to be implemented. |
| **I2C** | The I2C interface comes in very handy to connect a large variety of chips. Communication to the external NVM and also to chips that implement functions such as a servo controller will require this bus. |
| **Serial I/O** | The serial I/O is used in some hardware modules for implementation of RailCom detectors. The PICO features two hardware UARTS and the option to implement more in software using the PIO state machines. |
| **Console I/O** | Serial I/O is used for console I/O. Rather than using dedicated I/O pins and a UART block in the controller, the PICO serial I/O will be implemented via the USB connector. |
| **LEDs, Button and Dip Switches** | A hardware module could make use of LEDs to indicate readiness and activity, as well as a set of switches to configure a hardware option. Not really required but certainly useful. |
| **WLAN** | WLAN is optional. But there is a PICO version with WLAN capability integrated. |

## 1.3 Hardware Module Schematics

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

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

## 1.4 Controller and Extension Board

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

picture

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

## 1.5 LCS Bus connector

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

Table 1.2: Bus Connector Pins

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

## 1.6 LCSNodes Extension Board Connector

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

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

Table 1.3: Controller Attributes

| Pin | Name | Pin | Name | Purpose |
|---|---|---|---|---|
| 1 | **DCC-1** | 2 | **DCC-2** | The DCC "+" and "-" signal as generated by the DCC Signal Generator. These pins are typically driven by the base station generating the layout DCC signal. |
| 3 | **GND** | 4 | **GND** | Common ground pins. |
| 5 | **ADC-0** | 6 | **ADC-1** | Analog input pins. The input is not protected. The analog voltage range is 0 to VCC. |
| 7 | **GND** | 8 | **GND** | Common ground pins. |
| 9 | **DIO-0** | 10 | **DIO-1** | Plain digital Pins, input or output. The pins are protected. |
| 11 | **DIO-2** | 12 | **DIO-3** | Plain digital Pins, input or output. The pins are protected. |
| 13 | **DIO-4** | 14 | **DIO-5** | Plain digital Pins, input or output. The pins are protected. |
| 15 | **DIO-6** | 16 | **DIO-7** | Plain digital Pins, input or output. The pins are protected. |
| 17 | **DIO-8** | 18 | **DIO-9** | Plain digital Pins, input or output. The pins are protected. |
| 19 | **DIO-10** | 20 | **DIO-11** | Plain digital Pins, input or output. The pins are protected. |
| 21 | **GND** | 22 | **GND** | Common ground pins. |
| 23 | **BI-0** | 24 | **BI-1** | Bus Address input lines. Up to four extension boards can be connected, the BI pins are used to determine the I2C address on the I2C extension bus. |
| 25 | **BO-0** | 26 | **BO-1** | Bus Address output lines. The BO lines are computed form the BI lines. If for example BI is 1:0 the BO lines will become 1:1. The starting output pins values are 1:1. |
| 27 | **SCL** | 28 | **SDA** | I2C extension bus channel. The lines are protected with a serial resistor and there is a pull-up resistor to VCC. |
| 29 | **RST** | 30 | **EXT** | RST is reset line. Active Low. EXT is the external interrupt line which be raised from an extension board. Active low. |
| 31 | **VCC** | 32 | **VCC** | VCC 5V supply to extension boards. |

*Continued on next page*

| Pin | Name | Pin | Name | Purpose |
|---|---|---|---|---|
| 33 | **GND** | 34 | **GND** | Common ground pins. |
| 35 | **VS** | 36 | **VS** | Board Input voltage forward. These connector pins are primarily used by extension boards that need the high power input. Examples are H-Bridges on such a board or boards that have their power supply circuitry. |
| 37 | **VS** | 38 | **VS** | Board Input voltage forward. |
| 39 | **GND** | 40 | **GND** | Common ground pins. |

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

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

For even more complex extension boards, it is perhaps the better idea to combine a main board with an extension board capabilities to one monolithic board but still keep the extension connector for other not so complex extension boards to attach. As a guideline, only the first extension board will benefit from all signals coming from the main controller board. All follow on extension boards will only get the DCC signals, the interrupt and reset line, the I2C signal and the power lines.

## 1.7 Track Power Connectors

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

Table 1.4: Controller Attributes

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

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

## 1.8 Summary

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

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

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

# A Listings test

## A.1 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

};

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

```
99  // Controller pin related definitions. A pin can be valid, undefined or illegal. An undefined pin for a pin
100 // field in the configuration structure indicates that  the pin has not been used by the firmware
101 // implementation but is a pin that the particular controller would support. An illegal pin means that the
102 // pin is not offered by this controller and cannot be assigned at all.
103 //
104 //-----------------------------------------------------------------------------------------------------------
105 const uint8_t UNDEFINED_PIN    = 255;
106 const uint8_t ILLEGAL_PIN      = 254;
107
108 //-----------------------------------------------------------------------------------------------------------
109 // The controller families. Currently, there is only the Raspberry PI Pico models.
110 //
111 //-----------------------------------------------------------------------------------------------------------
112 enum ControllerFamily : uint8_t {
113
114     CF_UNDEFINED     = 0,
115     CF_RP_PICO       = 1
116 };
117
118 //-----------------------------------------------------------------------------------------------------------
119 // DIO pin related definitions. A digital pin can be an input pin, with or without pull-up, or an output
120 // pin. DIO pins can also be associated with an interrupt handler. The handler itself is mapped to an edge
121 // or level event.
122 //
123 //-----------------------------------------------------------------------------------------------------------
124 enum dioMode : uint8_t {
125
126     IN             = 0,
127     OUT            = 1,
128     IN_PULLUP      = 2
129 };
130
131 //-----------------------------------------------------------------------------------------------------------
132 // GPIO interrupts are detected as level change or edge changes.
133 //
134 //-----------------------------------------------------------------------------------------------------------
135 enum intEventTyp : uint8_t {
136
137     EVT_NONE     = 0,
138     EVT_LOW      = 1,
139     EVT_HIGH     = 2,
140     EVT_FALL     = 3,
141     EVT_RISE     = 4,
142     EVT_CHANGE   = 5
143 };
144
145 //-----------------------------------------------------------------------------------------------------------
146 // The UART modes. There are two implementations. The PICO offers two hardware UARTS. We use them with 8
147 // bits with a parity bit. The second type UART is a software implementation based on the PICO PIO blocks.
148 //
149 //-----------------------------------------------------------------------------------------------------------
150 enum UartMode : uint8_t {
151
152     UART_MODE_UNDEFINED = 0,
153     UART_MODE_8N1       = 1,
154     UART_MODE_8N1_PIO   = 2
155 };
156
157 //-----------------------------------------------------------------------------------------------------------
158 // Callback functions signatures.
159 //
160 //-----------------------------------------------------------------------------------------------------------
161 extern "C" {
162
163     typedef void ( *TimerCallback ) ( uint32_t timerVal );
164     typedef void ( *GpioCallback ) ( uint8_t pin, uint8_t event );
165 }
166
167 //-----------------------------------------------------------------------------------------------------------
168 // CDC features a data structure that records all HW specific pins and flags. The values are set by the
169 // initialization code in a project and are validated. All modules in a project will then just use the
170 // data structure fields using the data for calls to the Hal layer. For example, an application that
171 // uses DIO_PIN_0 and DIO_PIN_1 will set the HW pin numbers of the controller / board combination used
172 // in a config data structure "cfg". A call to write a value to the DIO pin, will then just use
173 // "cfg.DIO_PIN_1" as argument in the "writeDio" call. The "writeDio" call itself will not check the
174 // value of the configured DIO pin, all it will do is to ensure that it is not UNDEFINED. Note that the
175 // structure has more pins defined that a potential controller may have. If so, these fields are set to
176 // UNDEFINED. The structure is the superset of all possible HW items to configure.
177 //
178 // In a later runtime version, we may put this structure as constant data into the non-volatile chip on
179 // the board. It will then just be read from there.
180 //
181 //-----------------------------------------------------------------------------------------------------------
182 struct CdcConfigDesc {
183
184     uint8_t   CFG_STATUS;
185
186     uint8_t   PFAIL_PIN;
187     uint8_t   EXT_INT_PIN;
188     uint8_t   READY_LED_PIN;
189     uint8_t   ACTIVE_LED_PIN;
190
191     uint8_t   DIO_PIN_0;
192     uint8_t   DIO_PIN_1;
193     uint8_t   DIO_PIN_2;
194     uint8_t   DIO_PIN_3;
195     uint8_t   DIO_PIN_4;
196     uint8_t   DIO_PIN_5;
197     uint8_t   DIO_PIN_6;
```

```
198     uint8_t   DIO_PIN_7;
199     uint8_t   DIO_PIN_8;
200     uint8_t   DIO_PIN_9;
201     uint8_t   DIO_PIN_10;
202     uint8_t   DIO_PIN_11;
203     uint8_t   DIO_PIN_12;
204     uint8_t   DIO_PIN_13;
205     uint8_t   DIO_PIN_14;
206     uint8_t   DIO_PIN_15;
207
208     uint8_t   ADC_PIN_0;
209     uint8_t   ADC_PIN_1;
210     uint8_t   ADC_PIN_2;
211     uint8_t   ADC_PIN_3;
212
213     uint8_t   PWM_PIN_0;
214     uint8_t   PWM_PIN_1;
215     uint8_t   PWM_PIN_2;
216     uint8_t   PWM_PIN_3;
217
218     // ??? do we need more for quad controller ?
219
220     uint8_t   UART_RX_PIN_0;
221     uint8_t   UART_TX_PIN_0;
222
223     uint8_t   UART_RX_PIN_1;
224     uint8_t   UART_TX_PIN_1;
225
226     uint8_t   UART_RX_PIN_2;
227     uint8_t   UART_TX_PIN_2;
228
229     uint8_t   UART_RX_PIN_3;
230     uint8_t   UART_TX_PIN_3;
231
232     uint8_t   SPI_MOSI_PIN_0;
233     uint8_t   SPI_MISO_PIN_0;
234     uint8_t   SPI_SCLK_PIN_0;
235
236     uint8_t   SPI_MOSI_PIN_1;
237     uint8_t   SPI_MISO_PIN_1;
238     uint8_t   SPI_SCLK_PIN_1;
239
240     uint8_t   NVM_I2C_SCL_PIN;
241     uint8_t   NVM_I2C_SDA_PIN;
242     uint8_t   NVM_I2C_ADR_ROOT;
243
244     uint8_t   EXT_I2C_SCL_PIN;
245     uint8_t   EXT_I2C_SDA_PIN;
246     uint8_t   EXT_I2C_ADR_ROOT;
247
248     uint32_t  NODE_NVM_SIZE;
249     uint32_t  EXT_NVM_SIZE;
250
251     uint8_t   CAN_BUS_CTRL_MODE;
252     uint8_t   CAN_BUS_RX_PIN;
253     uint8_t   CAN_BUS_TX_PIN;
254     uint32_t  CAN_BUS_DEF_ID;
255 };
256
257 //-----------------------------------------------------------------------------------------------------------
258 // The routines that make up the hardware abstraction layer. The routines expect hardware pin numbers.
259 // To recap, the CDC layer offers a set of reserved resource names, such as "DIO_PIN_0", which describes
260 // the resource containing the hardware pin and some flags. The configuration routines in this layer will use
261 // these pins and other data stored to configure the hardware. Under the defined resource name name all
262 // upper layers refer to the hardware using the to the configured IO capabilities.
263 //
264 // Complex resources, such as the UART or SPI interface, have more than one HW pin they will use. In this
265 // case one of the HW pins, see the function documentation, will serve as the handle to the resource.
266 //
267 //-----------------------------------------------------------------------------------------------------------
268
269 //-----------------------------------------------------------------------------------------------------------
270 // The console IO functions. We will provide a serial IO via the USB connector of the PICO. The files
271 // need to be linked with the "tinyUSB" library and the cmake file needs to set the option. Then we can
272 // use scanf and printf and so on. In addition, we need  function  that just attempts to read a character
273 // and returns immediately when there is none.
274 //
275 //-----------------------------------------------------------------------------------------------------------
276 uint8_t         configureConsoleIO( );
277 bool            isConsoleConnected( );
278 char            getConsoleChar( uint32_t timeoutVal = 0 );
279
280 //-----------------------------------------------------------------------------------------------------------
281 // CDC setup and configuration routines. The idea is to help the library write with a default configuration
282 // structure. All pins HW that are fixed in their location will be set. A library programmer will just get
283 // that default structure and set the values necessary for the particular case.
284 //
285 //-----------------------------------------------------------------------------------------------------------
286 CdcConfigDesc   getConfigDefault( );
287 CdcConfigDesc   *getConfigActual( );
288 void            printConfigInfo( CdcConfigDesc *ci );
289 void            setDebugLevel( uint8_t level = 0 );
290
291 uint8_t         init( CdcConfigDesc *ci );
292 void            fatalError( uint8_t n );
293 void            fatalErrorMsg( char *str, uint8_t n, uint8_t rStat );
294
295 //-----------------------------------------------------------------------------------------------------------
296 // General controller routines.
```

```
297 //
298 //-------------------------------------------------------------------------------------------------------------
299 uint16_t      getFamily( );
300 uint32_t      getVersion( );
301 uint32_t      getChipMemSize( );
302 uint32_t      getChipNvmSize( );
303 uint32_t      getCpuFrequency( );
304 uint32_t      getMillis( );
305 uint32_t      getMicros( );
306 void          sleepMillis( uint32_t val );
307 void          sleepMicros( uint32_t val );
308
309 //-------------------------------------------------------------------------------------------------------------
310 // The LCS runtime needs to build a unique ID for the node.
311 //
312 //-------------------------------------------------------------------------------------------------------------
313 uint32_t      createUid( );
314
315 //-------------------------------------------------------------------------------------------------------------
316 // Timer management routines.
317 //
318 //-------------------------------------------------------------------------------------------------------------
319 void          onTimerEvent( TimerCallback functionId );
320 void          startRepeatingTimer( uint32_t val );
321 void          setRepeatingTimerLimit( uint32_t val );
322 uint32_t      getRepeatingTimerLimit( );
323 void          stopRepeatingTimer( );
324
325 //-------------------------------------------------------------------------------------------------------------
326 // Analog input routines.
327 //
328 //-------------------------------------------------------------------------------------------------------------
329 uint8_t       configureAdc( uint8_t adcPin );
330 uint16_t      getAdcRefVoltage( );
331 uint16_t      getAdcDigitRange( );
332 uint16_t      readAdc( uint8_t adcPin );
333
334 //-------------------------------------------------------------------------------------------------------------
335 // Digital Input/Output routines.
336 //
337 //-------------------------------------------------------------------------------------------------------------
338 uint8_t       configureDio( uint8_t dioPin, uint8_t Mode = IN );
339 void          registerDioCallback( uint8_t dioPin, uint8_t event, CDC::GpioCallback func );
340 void          unregisterDioCallback( uint8_t dioPin );
341 bool          readDio( uint8_t dioPin );
342 uint8_t       writeDio( uint8_t dioPin, bool val );
343 uint8_t       toggleDio( uint8_t dioPin );
344 uint32_t      readDioMask( uint32_t dioMask );
345 uint8_t       writeDioMask( uint32_t dioMask, uint32_t dioVal );
346 uint8_t       writeDioPair( uint8_t dioPin1, bool val1, uint8_t dioPin2, bool val2 );
347
348 //-------------------------------------------------------------------------------------------------------------
349 // PWM output routines.
350 //
351 //-------------------------------------------------------------------------------------------------------------
352 uint8_t       configurePwm( uint8_t   pwmPin,
353                             uint32_t  pwmFreqency,
354                             bool      phaseCorrect  = true,
355                             bool      inverted      = false
356                           );
357
358 uint8_t       writePwm( uint8_t pwmPin, uint8_t dutyCycle );
359
360 //-------------------------------------------------------------------------------------------------------------
361 // Serial IO routines.
362 //
363 //-------------------------------------------------------------------------------------------------------------
364 uint8_t       configureUart( uint8_t rxPin, uint8_t txPin, uint32_t baudRate, UartMode mode );
365 uint8_t       startUartRead( uint8_t rxPin );
366 uint8_t       stopUartRead( uint8_t rxPin );
367 uint8_t       getUartBuffer( uint8_t rxPin, uint8_t *buf, uint8_t bufLen );
368
369 //-------------------------------------------------------------------------------------------------------------
370 // I2C management routines.
371 //
372 //-------------------------------------------------------------------------------------------------------------
373 uint8_t       configureI2C( uint8_t sclPin, uint8_t sdaPin, uint32_t baudRate = 100 * 1000 );
374 uint8_t       i2cWrite( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit = false );
375 uint8_t       i2cRead( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit = false );
376
377 //-------------------------------------------------------------------------------------------------------------
378 // SPI management routines.
379 //
380 //-------------------------------------------------------------------------------------------------------------
381 uint8_t       configureSPI( uint8_t sclkPin, uint8_t mosiPin, uint8_t misoPin, uint32_t baudRate = 10 * 1000 * 1000 );
382 uint8_t       spiBeginTransaction( uint8_t sclkPin, uint8_t csPin );
383 uint8_t       spiEndTransaction( uint8_t sclkPin, uint8_t csPin );
384 uint8_t       spiRead( uint8_t sclkPin, uint8_t *buf, uint32_t len );
385 uint8_t       spiWrite( uint8_t sclkPin, uint8_t *buf, uint32_t len );
386
387 };
388
389 #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 )
//
//-----------------------------------------------------------------------------------------------------------
```

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

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

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

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

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

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

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

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

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

    return ( CONTROLLER_FAMILY );
}

uint32_t getVersion( ) {

    return ( CDC_LIB_MAJOR_VERSION << 8 | CDC_LIB_MINOR_VERSION );
}

uint32_t getChipMemSize( ) {

    return ( CHIP_MEM_SIZE );
}

uint32_t getChipNvmSize( ) {

    return ( CHIP_NVM_SIZE   );
}

uint32_t getCpuFrequency( ) {

    return ( clock_get_hz( clk_sys ));
}

uint32_t getMillis( ) {

    return ( to_ms_since_boot( get_absolute_time( )));
}

uint32_t getMicros( ) {

    return ( to_us_since_boot( get_absolute_time( )));
}

void sleepMillis( uint32_t val ) {

    sleep_ms( val );
}

void sleepMicros( uint32_t val ) {

    sleep_us( val );
}

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

    uint32_t rVal = 0;

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

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

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

    return ( rVal );
}

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

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

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

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

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

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

    return ( NO_ERR );
}

uint16_t getAdcRefVoltage( ) {

    return ( ADC_REF_VOLTAGE_MILLI_VOLT );
}

uint16_t getAdcDigitRange( ) {

    return ( ADC_DIGIT_RANGE );
}

uint16_t readAdc( uint8_t adcPin ) {

    AdcInst *tmp = nullptr;

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

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

    UartInst *uart = nullptr;

    if ( mode == UART_MODE_8N1 ) {

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

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

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

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

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

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

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

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

```
1188
1189        i2c_init( i2c -> i2cHw, i2c -> baudRate );
1190        i2c_set_slave_mode( i2c -> i2cHw, false, 0 );
1191
1192        gpio_set_function( i2c -> sclPin, GPIO_FUNC_I2C );
1193        gpio_set_function( i2c -> sdaPin, GPIO_FUNC_I2C);
1194        gpio_pull_up( i2c -> sclPin );
1195        gpio_pull_up( i2c -> sdaPin );
1196
1197        return ( NO_ERR );
1198    }
1199
1200    uint8_t i2cRead( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit ) {
1201
1202        I2CInst *i2c = nullptr;
1203
1204        if      (( CdcI2C0.sclPin == sclPin ) && ( CdcI2C0.configured )) i2c = &CdcI2C0;
1205        else if (( CdcI2C1.sclPin == sclPin ) && ( CdcI2C1.configured )) i2c = &CdcI2C1;
1206        else return ( I2C_PORT_ERR );
1207
1208        auto ret = i2c_read_blocking_until( i2c -> i2cHw,
1209                                            i2cAdr,
1210                                            buf,
1211                                            len,
1212                                            stopBit,
1213                                            make_timeout_time_ms( i2c -> timeoutValMs ));
1214
1215        #if CDC_DEBUG == 1
1216        printf( "i2cRead: scl: %d, i2c: 0x%x, buf: %p, buf[0] %x, buf[1] %x, len: %d, stop: %d\n",
1217                  sclPin, i2cAdr, buf, buf[0], buf[1], len, stopBit );
1218        if ( ret == PICO_ERROR_GENERIC ) printf( "I2C read, PICO generic error\n" );
1219        if ( ret == PICO_ERROR_TIMEOUT ) printf( "I2C read, PICO timeout error\n" );
1220        #endif
1221
1222        if (( ret == PICO_ERROR_GENERIC ) || ( ret == PICO_ERROR_TIMEOUT )) return ( I2C_READ_ERR );
1223
1224        return ( NO_ERR );
1225    }
1226
1227    uint8_t i2cWrite( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit ) {
1228
1229        #if CDC_DEBUG == 1
1230        printf( "i2cWrite: scl: %d, i2c: 0x%x, buf: %p, buf[0] %x, buf[1] %x, len: %d, stop: %d\n",
1231                 sclPin, i2cAdr, buf, buf[0], buf[1], len, stopBit );
1232        #endif
1233
1234
1235        I2CInst *i2c = nullptr;
1236
1237        if      (( CdcI2C0.sclPin == sclPin ) && ( CdcI2C0.configured )) i2c = &CdcI2C0;
1238        else if (( CdcI2C1.sclPin == sclPin ) && ( CdcI2C1.configured )) i2c = &CdcI2C1;
1239        else return ( I2C_PORT_ERR );
1240
1241        auto ret = i2c_write_blocking_until( i2c -> i2cHw,
1242                                             i2cAdr,
1243                                             buf,
1244                                             len,
1245                                             stopBit,
1246                                             make_timeout_time_ms( i2c -> timeoutValMs ));
1247
1248        #if CDC_DEBUG == 1
1249        if ( ret == PICO_ERROR_GENERIC ) printf( "I2C write, PICO generic error\n" );
1250        if ( ret == PICO_ERROR_TIMEOUT ) printf( "I2C write, PICO timeout error\n" );
1251        #endif
1252
1253        if (( ret == PICO_ERROR_TIMEOUT) || ( ret == PICO_ERROR_GENERIC ) || ( ret != len )) return ( I2C_WRITE_ERR );
1254
1255        return ( NO_ERR );
1256    }
1257
1258    //-------------------------------------------------------------------------------------------------------
1259    // SPI interface section. The PICO features two SPI HW blocks. We implement a simple SPI interface with a
1260    // a fixed set of SPI options for frequency, bit order and mode. One day this may change. We do not take
1261    // care of the chip select stuff and expect that the caller manages the select pin.
1262    //
1263    //-------------------------------------------------------------------------------------------------------
1264    uint8_t configureSPI( uint8_t sclkPin, uint8_t mosiPin, uint8_t misoPin, uint32_t baudRate ) {
1265
1266        SPIInst *spi = nullptr;
1267
1268         if ((( 1 << sclkPin ) & VALID_SPI_0_SCK_PINS ) &&
1269             (( 1 << mosiPin ) & VALID_SPI_0_TX_PINS  ) &&
1270             (( 1 << misoPin ) & VALID_SPI_0_RX_PINS  )) {
1271
1272            spi = &CdcSPI0;
1273            spi -> spiHw = spi0;
1274        }
1275        else if ((( 1 << sclkPin ) & VALID_SPI_1_SCK_PINS ) &&
1276                 (( 1 << mosiPin ) & VALID_SPI_1_TX_PINS  ) &&
1277                 (( 1 << misoPin ) & VALID_SPI_1_RX_PINS  )) {
1278
1279            spi = &CdcSPI1;
1280            spi -> spiHw = spi1;
1281        }
1282        else return ( SPI_PORT_ERR );
1283
1284        spi -> mosiPin     = mosiPin;
1285        spi -> misoPin     = misoPin;
1286        spi -> sclkPin     = sclkPin;
```

```
1287    spi -> frequency   = SPI_FREQUENCY;
1288    spi -> configured  = true;
1289    spi -> active      = false;
1290
1291    spi_init( spi -> spiHw, SPI_FREQUENCY );
1292
1293    spi_set_format( spi -> spiHw,     // SPI instance
1294                    8,                // Number of bits per transfer
1295                    SPI_CPOL_1,       // Polarity (CPOL)
1296                    SPI_CPHA_1,       // Phase (CPHA)
1297                    SPI_MSB_FIRST );
1298
1299    gpio_set_function( sclkPin, GPIO_FUNC_SPI );
1300    gpio_set_function( mosiPin, GPIO_FUNC_SPI );
1301    gpio_set_function( misoPin, GPIO_FUNC_SPI );
1302
1303    return ( NO_ERR );
1304}
1305
1306uint8_t  spiBeginTransaction( uint8_t sclkPin, uint8_t csPin ) {
1307
1308    SPIInst *spi = nullptr;
1309
1310    if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
1311    else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
1312    else return ( SPI_PORT_ERR );
1313
1314    if ( spi -> active ) {
1315
1316        // ??? should we check who is active and just ignore when the same ? else "error " ?
1317
1318        return ( NO_ERR );
1319
1320    } else {
1321
1322        spi -> active     = true;
1323        spi -> selectPin  = csPin;
1324
1325        CDC::writeDio( csPin, false );
1326        return ( NO_ERR );
1327    }
1328}
1329
1330uint8_t spiEndTransaction( uint8_t sclkPin, uint8_t csPin ) {
1331
1332    SPIInst *spi = nullptr;
1333
1334    if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
1335    else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
1336    else return ( SPI_PORT_ERR );
1337
1338    if ( spi -> active ) {
1339
1340        // ??? check that this is the correct pin ?
1341
1342        CDC::writeDio( csPin, true );
1343
1344        spi -> active     = false;
1345        spi -> selectPin  = UNDEFINED_PIN;
1346
1347        return ( NO_ERR );
1348
1349    }
1350    else return ( NO_ERR ); // ???  "error "  not active...
1351}
1352
1353uint8_t spiRead( uint8_t sclkPin, uint8_t *buf, uint32_t len ) {
1354
1355    SPIInst *spi = nullptr;
1356
1357    if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
1358    else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
1359    else return ( SPI_PORT_ERR );
1360
1361    if ( spi -> active ) {
1362
1363        int bytesRead = spi_read_blocking( spi -> spiHw, 0, buf, len );
1364        return ( NO_ERR );
1365
1366    } else return ( NO_ERR ); // ??? fix : not active ...
1367}
1368
1369uint8_t spiWrite( uint8_t sclkPin, uint8_t *buf, uint32_t len ) {
1370
1371    SPIInst *spi = nullptr;
1372
1373    if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
1374    else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
1375    else return ( SPI_PORT_ERR );
1376
1377    if ( spi -> active ) {
1378
1379        spi_write_blocking( spi -> spiHw, buf, len );
1380        return ( NO_ERR );
1381
1382    } else return ( NO_ERR ); // ??? fix : not active ...
1383}
1384
1385//-------------------------------------------------------------------------------------------------------
```

```
1386 // Print out the Config Structure.
1387 //
1388 //-------------------------------------------------------------------------------------------------------------
1389 void printConfigInfo( CdcConfigDesc *ci ) {
1390
1391     printf( "CDC Pin Configuration Info ( status %d ): \n", ci -> CFG_STATUS );
1392
1393     printf( "Pfail pin: %2d, ExtInt pin: %2d \n", ci -> PFAIL_PIN, ci -> EXT_INT_PIN );
1394
1395     printf( "ReadyLed pin: %2d, ActiveLed pin: %2d \n", ci -> READY_LED_PIN, ci -> ACTIVE_LED_PIN );
1396
1397     printf( "DIO pins ( 0 .. 7 ): %2d %2d %2d %2d %2d %2d %2d %2d\n",
1398             ci -> DIO_PIN_0, ci -> DIO_PIN_1, ci -> DIO_PIN_2, ci -> DIO_PIN_3,
1399             ci -> DIO_PIN_4, ci -> DIO_PIN_5, ci -> DIO_PIN_6, ci -> DIO_PIN_7 );
1400
1401     printf( "DIO pins ( 8 .. 15 ): %2d %2d %2d %2d %2d %2d %2d %2d\n",
1402             ci -> DIO_PIN_8, ci -> DIO_PIN_9, ci -> DIO_PIN_10, ci -> DIO_PIN_11,
1403             ci -> DIO_PIN_12, ci -> DIO_PIN_13, ci -> DIO_PIN_14, ci -> DIO_PIN_15 );
1404
1405     printf( "ADC pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
1406             ci -> ADC_PIN_0, ci -> ADC_PIN_1, ci -> ADC_PIN_2, ci -> ADC_PIN_3 );
1407
1408     printf( "PWM pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
1409             ci -> PWM_PIN_0, ci -> PWM_PIN_1, ci -> PWM_PIN_2, ci -> PWM_PIN_3 );
1410
1411     printf( "UART RX pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
1412             ci -> UART_RX_PIN_0, ci -> UART_RX_PIN_1, ci -> UART_RX_PIN_2, ci -> UART_RX_PIN_3 );
1413
1414     printf( "UART TX pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
1415             ci -> UART_TX_PIN_0, ci -> UART_TX_PIN_1, ci -> UART_TX_PIN_2, ci -> UART_TX_PIN_3 );
1416
1417     printf( "SPI0 Pins: MOSI: %2d, MISO: %2d, SCLK: %2d \n",
1418             ci -> SPI_MOSI_PIN_0, ci -> SPI_MISO_PIN_0, ci -> SPI_SCLK_PIN_0 );
1419
1420     printf( "SPI1 Pins: MOSI: %2d, MISO: %2d, SCLK: %2d \n",
1421             ci -> SPI_MOSI_PIN_1, ci -> SPI_MISO_PIN_1, ci -> SPI_SCLK_PIN_1 );
1422
1423     printf( "NVM I2C Pins: SCL: %2d, SDA: %2d, I2C Root: 0x%x \n",
1424             ci -> NVM_I2C_SCL_PIN, ci -> NVM_I2C_SDA_PIN, ci -> NVM_I2C_ADR_ROOT );
1425
1426     printf( "EXT I2C Pins: SCL: %2d, SDA: %2d, I2C Root: 0x%x \n",
1427             ci -> EXT_I2C_SCL_PIN, ci -> EXT_I2C_SDA_PIN, ci -> EXT_I2C_ADR_ROOT );
1428
1429     printf( "\n" );
1430
1431 }
1432
1433 }; // namespace CDC
```

## A.2 LCS Runtime Lib

```
//-----------------------------------------------------------------------------------------------------------
//
// Layout Control System - Runtime Library include file
//
//-----------------------------------------------------------------------------------------------------------
// At the heart of the layout control system, LCS, is the runtime library implementing the basic functions.
// Please refer to the document for information on concepts and implementation notes. This is the external
// include file for the firmware programmer. All external definitions of key constants and types are
// included here.
//
//-----------------------------------------------------------------------------------------------------------
//
// LCS - Runtime Library
// Copyright (C) 2021 - 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 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/>.
//
//-----------------------------------------------------------------------------------------------------------
#ifndef LCS_RT_LIB_h
#define LCS_RT_LIB_h

//-----------------------------------------------------------------------------------------------------------
// Include files.
//
//-----------------------------------------------------------------------------------------------------------
#include <stdint.h>
#include <inttypes.h>
#include "LcsCdcLib.h"

//-----------------------------------------------------------------------------------------------------------
// All LCS Library definitions are in a name space "LCS". You can prefix each constant, type or function
// with the "LCS::" prefix, or declare using the namespace in your code.
//
//-----------------------------------------------------------------------------------------------------------
namespace LCS {

//-----------------------------------------------------------------------------.-----------------------------
// A node is identified through the node number. Node numbers start with one. The nodeId of zero represents
// the NIL node Id. The node Id is a 12-bit number, so up to 4095 nodes can be addressed. The nodeId, a
// unique Id for the LCS nodes in a layout, is also used as the canId used for the CAN bus. Keep in mind
// that a CAN bus nevertheless can reasonably handle about 127 nodes at the same time.
//
//-----------------------------------------------------------------------------------------------------------
enum LcsNodeId : uint16_t {

    NIL_NODE_ID    = 0,
    MIN_NODE_ID    = 1,
    MAX_NODE_ID    = 4095
};

//-----------------------------------------------------------------------------------------------------------
// A node type can be assigned to a node. NodeId types start with one. The nodeType of zero represents the
// NIL node type. A node type is arbitrarily defined by the firmware programmer.
//
//-----------------------------------------------------------------------------------------------------------
enum LcsNodeTypeId : uint8_t {

    NIL_NODE_TYPE     = 0,
    MIN_NODE_TYPE_ID  = 1,
    MAX_NODE_TYPE_ID  = 255
};

//-----------------------------------------------------------------------------------------------------------
// The port Id identifies the port on a node. Port numbers start with one. The port number zero represents
// the NIL port number and usually refers to the node itself. A node can have up to 15 ports.
//
//-----------------------------------------------------------------------------------------------------------
enum LcsPortId : uint8_t {

    NIL_PORT_ID    = 0,
    MIN_PORT_ID    = 1,
    MAX_PORT_ID    = 15
};

//-----------------------------------------------------------------------------------------------------------
// A port type can be assigned to a port. Port types start with one. The portType of zero represents the
// NIL port type. A port type is arbitrarily defied by the firmware programmer.
//
//-----------------------------------------------------------------------------------------------------------
  enum LcsPortTypeId : uint8_t {

    NIL_PORT_TYPE     = 0,
    MIN_PORT_TYPE_ID  = 1,
    MAX_PORT_TYPE_ID  = 255
};

//-----------------------------------------------------------------------------------------------------------
// Events are just numbers assigned to an event by a configuration tool. Event id numbers start with one.
// The  event number zero represents the NIL event number. The maximum event id number is 65535.
//
//-----------------------------------------------------------------------------------------------------------
```

```
99  enum LcsEventId : uint16_t {
100
101     NIL_EVENT_ID  = 0,
102     MIN_EVENT_ID  = 1,
103     MAX_EVENT_ID  = 65535
104 };
105
106 //-----------------------------------------------------------------------------------------------------------
107 // Each locomotive has a type. There are STEAM, DIESEL and ELECTRIC engines so far.
108 //
109 // ??? additional types ?
110 //-----------------------------------------------------------------------------------------------------------
111 enum DccLocoType : uint8_t {
112
113     LOC_T_NIL       = 0,
114     LOC_T_STEAM     = 1,
115     LOC_T_DIESEL    = 2,
116     LOC_T_ELECTRIC  = 3
117 };
118
119 //-----------------------------------------------------------------------------------------------------------
120 // The base station maintains the locomotive sessions. A session is assigned by the base station and commands
121 // for the locomotive use this session number. Session Ids start with 1, up to 255 simultaneous sessions are
122 //  supported.
123 //
124 //-----------------------------------------------------------------------------------------------------------
125 enum DccSessionId : uint8_t {
126
127     NIL_LOCO_SESSION_ID  = 0,
128     MIN_LOCO_SESSION_ID  = 1,
129     MAX_LOCO_SESSION_ID  = 255
130 };
131
132 //-----------------------------------------------------------------------------------------------------------
133 // The cabId is the locomotive number or address. For DCC type locomotives, there is a short and long address
134 // for a decoder. The short address ranges from 1 .. 127, the long address from 1 .. to 10239. However, most
135 // base stations support just up to 9999 locomotives IDs and so do we. Analog engines do not really have a
136 // cabId. When an analog engines is introduced to the system by telling the base station about the engine and
137 // lo cation, the base station assign a cabId. Refer to the book for the details.
138 //
139 //-----------------------------------------------------------------------------------------------------------
140 enum DccCabId : uint16_t {
141
142     NIL_CAB_ID  = 0,
143     MIN_CAB_ID  = 1,
144     MAX_CAB_ID  = 9999
145 };
146
147 //-----------------------------------------------------------------------------------------------------------
148 // A DCC decoder features configuration variables, called CVs. CVs are numbered starting with 1, the maximum
149 // number is 1024.
150 //
151 //-----------------------------------------------------------------------------------------------------------
152 enum DccCvId : uint16_t {
153
154     NIL_DCC_CV_ID  = 0,
155     MIN_DCC_CV_ID  = 1,
156     MAX_DCC_CV_ID  = 1024
157 };
158
159 //-----------------------------------------------------------------------------------------------------------
160 // A locomotive decoder features up to 69 functions Ids. They are numbered from 0 to 68.
161 //
162 //-----------------------------------------------------------------------------------------------------------
163 enum DccFuncId : uint8_t {
164
165     NIL_DCC_FUNC_ID   = 255,
166     MIN_DCC_FUNC_ID   = 0,
167     MAX_DCC_FUNC_ID   = 68
168 };
169
170 //-----------------------------------------------------------------------------------------------------------
171 // According to the DCC standard, DCC decoder functions are grouped in ten groups, labelled from 1 to 10 .
172 //
173 //-----------------------------------------------------------------------------------------------------------
174 enum DccFuncGroupId : uint8_t {
175
176     MIN_DCC_FUNC_GROUP_ID   = 1,
177     MAX_DCC_FUNC_GROUP_ID   = 10
178 };
179
180 //-----------------------------------------------------------------------------------------------------------
181 // DCC decoder function mapping Ids. The LCS system defines a set of functions used by the handhelds such
182 // as horn, lights and so on. These identifiers are standardized for our handhelds and mapped to the DCC
183 // function.
184 //
185 //-----------------------------------------------------------------------------------------------------------
186 enum LcsDccFuncId : uint8_t {
187
188     NIL_LCS_DCC_FUNC_ID = 0,
189     MIN_LCS_DCC_FUNC_ID = 1,
190
191     // ??? function IDs go here...
192
193     MAX_LCS_DCC_FUNC_ID = 68
194 };
195
196 //-----------------------------------------------------------------------------------------------------------
197 // "CvModeOptions" is used by the DCC CV variables access routines to specify the access mode. Only options
```

```
198 // 0 and 1 are supported. The others are there for historic reasons, the functionality was found in older
199 // decoders and should not be supported anymore.
200 //
201 //------------------------------------------------------------------------------------------------------------
202 enum DccCvModeOptions : uint8_t {
203 
204     CVM_BYTE      = 0,
205     CVM_BIT       = 1,
206     CVM_PAGE      = 2,
207     CVM_REGISTER  = 3,
208     CVM_ADR_ONLY  = 4
209 };
210 
211 //------------------------------------------------------------------------------------------------------------
212 // The DCC standard defines several speed step modes. Today, the 28 speed step option is the one used in all
213 // new decoders. The other speed steps are mapped to the 128 value range.
214 //
215 //------------------------------------------------------------------------------------------------------------
216 enum DccSpeedSteps : uint8_t {
217 
218     DCC_SPEED_STEPS_14    = 1,
219     DCC_SPEED_STEPS_28    = 2,
220     DCC_SPEED_STEPS_128   = 3
221 };
222 
223 //------------------------------------------------------------------------------------------------------------
224 // The locomotive decoder speed. The range is defined for a DCC 128 speed step decoder, from 0 to 127. The
225 // speed of 1 represents the emergency speed stop. In normal operations, speed stops would thus go from 2
226 // to 0 and back. For analog engines, we keep this scheme and map it to the respective power levels.
227 //
228 //------------------------------------------------------------------------------------------------------------
229 enum LocSpeed : uint8_t {
230 
231     MIN_LOCO_SPEED      = 0,
232     ESTOP_LOCO_SPEED    = 1,
233     MAX_LOCO_SPEED      = 127
234 };
235 
236 //------------------------------------------------------------------------------------------------------------
237 // Locomotive direction.
238 //
239 //------------------------------------------------------------------------------------------------------------
240 enum LocoDirection : uint8_t {
241 
242     LOCO_DIR_LOCO_NEUTRAL  = 0,
243     LOCO_DIR_LOCO_FORWARD  = 1,
244     LOCO_DIR_LOCO_REVERSE  = 2
245 };
246 
247 //------------------------------------------------------------------------------------------------------------
248 // "LocSessionModes" specify the options when creating a session for the loco. Besides creating a normal
249 // session an existing session can be taken over or even shared among multiple handhelds.
250 //
251 //------------------------------------------------------------------------------------------------------------
252 enum LocoSessionModes : uint8_t {
253 
254     LSM_NORMAL  = 1,
255     LSM_STEAL   = 2,
256     LSM_SHARED  = 3
257 };
258 
259 //------------------------------------------------------------------------------------------------------------
260 // The defined board types. When the runtime is initialized, the firmware will pass the type to specify what
261 // board it expects. This value is compared to what is actually stored in the NVM of the main controller
262 // board. If they don't match, it is considered an error and the NVM needs to be configured to support the
263 // firmware.
264 //
265 //------------------------------------------------------------------------------------------------------------
266 enum LcsBoardType : uint16_t {
267 
268     BT_NIL                  = 0,
269     BT_MAIN_CONTROLLER      = 1,
270     BT_BASE_STATION         = 2,
271     BT_BLOCK_CONTROLLER     = 3,
272     BT_CAB_HANDHELD         = 4,
273 
274     BT_EXT_NIL              = 10,
275     BT_EXT_OCC_DETECT       = 11,
276     BT_EXT_SERVO            = 12,
277     BT_EXT_GPIO             = 13
278 };
279 
280 //------------------------------------------------------------------------------------------------------------
281 // The defined chip families. There are controller chip families such as the controller family RP2040, or
282 // chip families for the NVM chips used, and so on.
283 //
284 //------------------------------------------------------------------------------------------------------------
285 enum LcsControllerFamilyType : uint16_t {
286 
287     CF_FAM_NIL              = 0,
288     CF_FAM_RPICO            = 1,
289     CF_FAM_MICROCHIP        = 3,
290     CF_FAM_NXP              = 4
291 };
292 
293 //------------------------------------------------------------------------------------------------------------
294 // Extension board driver flags. The flag are set when a board is detected and also during board operation.
295 //
296 //------------------------------------------------------------------------------------------------------------
```

```
297 enum LcsBoardFlags : uint16_t {
298
299     BF_NIL                  = 0,
300     BF_EXT_BOARD_PRESENT    = ( 1U << 0 ),
301     BF_EXT_BOARD_VALID      = ( 1U << 1 ),
302     BF_EXT_BOARD_READY      = ( 1U << 2 )
303 };
304
305 //------------------------------------------------------------------------------------------------------------
306 // The configuration descriptor and node map have an option field. The following constants define the
307 // options that can be set.
308 //
309 //  NOPT_SKIP_NODE_ID_CONFIG - during startup, skip the nodeId configuration protocol.
310 //  NOPT_SKIP_NODE_INIT_STEP - during startup, skip the node initialization step.
311 //  NOPT_SKIP_PORT_INIT_STEP - during startup, skip the port initialization step.
312 //  NOPT_DEBUG_DURING_SETUP  - during startup print debug info until we use the mask of nodeMap
313 //
314 //------------------------------------------------------------------------------------------------------------
315 enum LcsNodeOptions : uint16_t {
316
317     NOPT_NIL                    = 0,
318     NOPT_SKIP_NODE_ID_CONFIG    = ( 1 << 0 ),
319     NOPT_SKIP_NODE_INIT_STEP    = ( 1 << 1 ),
320     NOPT_DEBUG_DURING_SETUP     = ( 1 << 2 ),
321     NOPT_FORMAT_RUNTIME         = ( 1 << 3 )
322 };
323
324 //------------------------------------------------------------------------------------------------------------
325 // Node Flags. Flags are initialized at library startup and represent library state information.
326 //
327 //  NFLAGS_EXT_PRESENT          - extension boards are present.
328 //  NFLAGS_NVM_WRITE_ENABLED    - write to the protected NVM areas is enabled.
329 //
330 //
331 //------------------------------------------------------------------------------------------------------------
332 enum LcsNodeFlags : uint16_t {
333
334     NFLAGS_NIL                  = 0,
335     NFLAGS_EXT_PRESENT          = ( 1 << 0 ),
336     NFLAGS_NVM_WRITE_ENABLED    = ( 1 << 1 )
337 };
338
339 //------------------------------------------------------------------------------------------------------------
340 // Nodes, ports and drivers are accessed with three main routines, GET, SET and REQ.
341 //
342 // GET - the get routine will use the item numbers to retrieve the data labelled by the item.
343 //
344 // SET - the set routine will use the item numbers to set the value. Note that not all items that can be
345 // read can also be written to. An attempt will result in an error return.
346 //
347 // REQ - the request call will transmit the request parameters to the node / port / driver where a registered
348 // callback or the driver entry point will be invoked. The result is returned via the parameters.
349 //
350 // One argument is the item. Items range from  0 ... 255 and are defined as follows:
351 //
352 //   0          -   NIL item, not used
353 //   1  ..  63  -   Node / port / driver reserved area items, global items for GET/SET/REQ requests.
354 //  64  .. 127  -   User defined items, specific meaning, accessed via the REQ routine.
355 // 128  .. 191  -   Node / port / driver data attributes returned from MEM for GET/SET.
356 // 192  .. 255  -   Node / port / driver data attributes copied from NVM to MEM for GET, copied from MEM to NVM
357 //                  for SET. The item range mirrors items 128 - 191. For example, 128 and 192 refer to the same
358 //                  attribute. Note that for a SET on a driver the HW needs to be enabled.
359 //
360 // The following declarations does just list the item numbers defined. The ranges are defined in the internal
361 // include file. The ranges as well as the reserved items defined here should not be tampered with.
362 //
363 // ??? to be sorted when more stable...
364 //------------------------------------------------------------------------------------------------------------
365 enum LcsItems : uint8_t {
366
367     ITEM_ID_OPTIONS                     = 1,
368     ITEM_ID_FLAGS                       = 2,
369     ITEM_ID_DEBUG_MASK                  = 3,
370     ITEM_ID_VERSION                     = 4,
371     ITEM_ID_TYPE                        = 5,
372
373     ITEM_ID_BOARD_VERSION               = 6,
374     ITEM_ID_CONTROLLER_FAMILY           = 7,
375     ITEM_ID_NVM_CHIP_FAMILY             = 8,
376
377     ITEM_ID_NODE_ID                     = 10,
378     ITEM_ID_NODE_UID                    = 11,
379     ITEM_ID_RESTART_COUNT               = 12,
380
381     ITEM_ID_PORT_MAP_ENTRIES            = 13,
382     ITEM_ID_EVENT_MAP_ENTRIES           = 14,
383     ITEM_ID_ATTR_MAP_ENTRIES            = 15,
384
385     ITEM_ID_NAME_1                      = 17,
386     ITEM_ID_NAME_2                      = 18,
387     ITEM_ID_NAME_3                      = 19,
388     ITEM_ID_NAME_4                      = 20,
389
390     ITEM_ID_EVENT_DELAY_TICKS           = 21,
391
392     ITEM_ID_RESET                       = 22,
393     ITEM_ID_SYNC                        = 23,
394     ITEM_ID_FORMAT                      = 24,
395
```

```
396     ITEM_ID_ADD_EVENT_MAP_ENTRY         = 25,
397     ITEM_ID_DEL_EVENT_MAP_ENTRY         = 26,
398     ITEM_ID_GET_EVENT_MAP_ENTRY         = 27,
399 
400     ITEM_ID_SET_READY_LED               = 30,
401     ITEM_ID_SET_ACTIVITY_LED            = 31,
402     ITEM_ID_TOGGLE_READY_LED            = 32,
403     ITEM_ID_TOGGLE_ACTIVITY_LED         = 33,
404 
405     ITEM_ID_NVM_PROTECTED_ACCESS        = 35,
406 
407     ITEM_ID_ENABLE_EVENT_PROCESSING     = 40,
408 
409     // ??? add stop and enable periodic processing ?
410 };
411 
412 //-----------------------------------------------------------------------------------------------------------
413 // The portMap entry has a flag field. The constants defined here indicate the bit positions and fields
414 // defined.
415 //
416 //  PF_PORT_ENABLED                 - the port is initialized and active
417 //  PF_PORT_EVENT_HANDLING_ENABLED  - the port has event handling enabled
418 //  PF_EVENT_PENDING                - an event has been received for this port and is pending.
419 //
420 //-----------------------------------------------------------------------------------------------------------
421 enum LcsPortFlags : uint16_t {
422 
423     PF_NIL                          = 0,
424     PF_PORT_ENABLED                 = ( 1 << 15 ),
425     PF_PORT_EVENT_HANDLING_ENABLED  = ( 1 << 14 ),
426     PF_EVENT_PENDING                = ( 1 << 13 )
427 };
428 
429 //-----------------------------------------------------------------------------------------------------------
430 // The port event action. When an event is received, it will be of the type shown below. There is an
431 // optional port specific time delay configured between the actual receipt of an event message and the
432 // invocation of the event callback.
433 //
434 //  PEA_EVENT_IDLE              - the port is idle.
435 //  PEA_EVENT_ON                - an "ON" event was received.
436 //  PEA_EVENT_OFF               - an "OFF" event was received.
437 //  PEA_EVENT_EVT               - an event with additional arguments was received.
438 //
439 //-----------------------------------------------------------------------------------------------------------
440 enum LcsPortEventAction : uint8_t {
441 
442     PEA_EVENT_IDLE    = 0,
443     PEA_EVENT_ON      = 1,
444     PEA_EVENT_OFF     = 2,
445     PEA_EVENT_EVT     = 3
446 };
447 
448 //-----------------------------------------------------------------------------------------------------------
449 // The debug mask. The library has a debug mask where each major part of the library has a flag. There could
450 // also be flags reserved for the firmware. There is an ITEM to read and set this mask. Wherever debugging is
451 // needed, the bit mask will be used to determine whether to print debugging data or not. From a performance
452 // perspective, the test will take just a few instructions. In other words we do not take out debugging code
453 // when going into production. Never liked this approach of conditional debug anyway.
454 //
455 // The usage of the debug mask is generally:
456 //
457 //      if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_xxx )) ....
458 //
459 // The DBG_CONFIG bit allows for the entire debugging messages to be enabled or disabled. This feature will
460 // also be used when we test whether we even have a console or not. If there is no console, all the prints
461 // will not be executed.
462 //
463 //-----------------------------------------------------------------------------------------------------------
464 enum DebugOtions : uint16_t {
465 
466     DBG_CONFIG          = ( 1U << 15 ),
467     DBG_SETUP           = ( 1U << 0 ),
468     DBG_NVM_ACCESS      = ( 1U << 1 ),
469     DBG_CAN_BUS         = ( 1U << 2 ),
470     DBG_MSG_BUS         = ( 1U << 3 ),
471     DBG_ATTRIBUTES      = ( 1U << 4 ),
472     DBG_EVENTS          = ( 1U << 5 )
473 };
474 
475 //-----------------------------------------------------------------------------------------------------------
476 // The message operation code identifies the LCS bus message. It is always the first data byte of the
477 // message. We encode the number of payload data bytes in the first three bits of the opCode. For each
478 // message length there is a maximum of 32 opCode possible. The OPC  macro helps to define the opcodes.
479 // The first argument is the length of the data bytes, the second the opcodeId within the group.
480 //
481 // ??? note: this list is work in progress, please us always the names rather than the numbers.
482 //-----------------------------------------------------------------------------------------------------------
483 #define OPC( len, id ) ((uint8_t) (( len << 5 ) + ( id & 0x1F )))
484 
485 enum LcsMsgOpCodes : uint8_t {
486 
487     LCS_OP_NO_MSG           = OPC( 0, 0 ),
488     LCS_OP_CFG              = OPC( 0, 1 ),
489     LCS_OP_OPS              = OPC( 0, 2 ),
490     LCS_OP_BON              = OPC( 0, 3 ),
491     LCS_OP_BOF              = OPC( 0, 4 ),
492 
493     LCS_OP_REL_LOC          = OPC( 1, 1 ),
494     LCS_OP_QRY_LOC          = OPC( 1, 2 ),
```

```
    LCS_OP_KEEP_LOC         = OPC( 1, 3 ),
    LCS_OP_ESTP             = OPC( 1, 4 ),

    LCS_OP_PING             = OPC( 2, 1 ),
    LCS_OP_ACK              = OPC( 2, 2 ),
    LCS_OP_DCC_ACK          = OPC( 2, 3 ),
    LCS_OP_SET_LSPD         = OPC( 2, 4 ),
    LCS_OP_SET_LMOD         = OPC( 2, 5 ),
    LCS_OP_LOC_FON          = OPC( 2, 6 ),
    LCS_OP_LOC_FOF          = OPC( 2, 7 ),
    LCS_OP_BACC             = OPC( 2, 8 ),
    LCS_OP_EACC             = OPC( 2, 9 ),
    LCS_OP_TON              = OPC( 2, 10 ),
    LCS_OP_TOF              = OPC( 2, 11 ),

    LCS_OP_RESET            = OPC( 3, 1 ),
    LCS_OP_REQ_LOC          = OPC( 3, 2 ),
    LCS_OP_SET_LCON         = OPC( 3, 3 ),
    LCS_OP_LOC_FGRP         = OPC( 3, 4 ),
    LCS_OP_SEND_DCC3        = OPC( 3, 5 ),
    LCS_OP_DCC_ERR          = OPC( 3, 6 ),
    LCS_OP_REQ_CVS          = OPC( 3, 7 ),

    LCS_OP_EVT_ON           = OPC( 4, 1 ),
    LCS_OP_EVT_OFF          = OPC( 4, 2 ),
    LCS_OP_SEND_DCC4        = OPC( 4, 3 ),
    LCS_OP_REP_CVS          = OPC( 4, 4 ),
    LCS_OP_SET_CVS          = OPC( 4, 5 ),
    LCS_SYS_TIME            = OPC( 4, 6 ),

    LCS_OP_ERR              = OPC( 5, 1 ),
    LCS_OP_SET_CVM          = OPC( 5, 2 ),
    LCS_OP_SEND_DCC5        = OPC( 5, 3 ),

    LCS_OP_EVT              = OPC( 6, 1 ),
    LCS_OP_SEND_DCC6        = OPC( 6, 2 ),
    LCS_OP_NCOL             = OPC( 6, 6 ),

    LCS_OP_REQ_NID          = OPC( 7, 1 ),
    LCS_OP_REP_NID          = OPC( 7, 2 ),
    LCS_OP_SET_NID          = OPC( 7, 3 ),
    LCS_OP_NODE_GET         = OPC( 7, 4 ),
    LCS_OP_NODE_PUT         = OPC( 7, 5 ),
    LCS_OP_NODE_REQ         = OPC( 7, 6 ),
    LCS_OP_NODE_REP         = OPC( 7, 7 ),
    LCS_OP_REP_LOC          = OPC( 7, 8 ),
    LCS_INFO                = OPC( 7, 9 )
};

//-----------------------------------------------------------------------------------------------------------
// LCS Core Library Error codes. The status code is used as a return value from most of the library methods.
// The numbers are grouped in a LCS library portion and a user firmware portion. The LCS library portion
// ranges from 1 to 127, the user portion from 128 to 255. The value of zero is generally an "OK".
//
// ??? add NVM errors, also CDC errors ?
//-----------------------------------------------------------------------------------------------------------
enum LcsErrorCodes : uint8_t {

    ALL_OK                              = 0,
    ERR_NOT_IMPLEMENTED                 = 1,
    ERR_NOT_SUPPORTED                   = 2,
    ERR_LIB_NOT_INITIALIZED             = 3,

    ERR_CDC_SETUP                       = 10,
    ERR_NVM_SETUP                       = 11,
    ERR_MEM_SETUP                       = 12,
    ERR_CAN_SETUP                       = 13,

    ERR_NVM_CHIP_SIZE_DETECT            = 14,
    ERR_NVM_NODE_MAP_CORRUPT            = 15,
    ERR_NVM_SIZE_EXCEEDED               = 16,
    ERR_MEM_SIZE_EXCEEDED               = 17,
    ERR_NVM_OP_FAILED                   = 18,

    ERR_NODE_NOT_OPS_STATE              = 20,
    ERR_NODE_NOT_CONFIG_STATE           = 21,
    ERR_NODE_OUTSTANDING_REQ_LIMIT      = 22,
    ERR_TASK_MAP_SIZE_EXCEEDED          = 23,

    ERR_INVALID_NODE_ID                 = 30,
    ERR_INVALID_PORT_ID                 = 31,
    ERR_INVALID_ITEM_ID                 = 32,
    ERR_INVALID_EVENT_ID                = 33,
    ERR_INVALID_BOARD_ID                = 34,
    ERR_INVALID_DRV_ITEM                = 35,
    ERR_INVALID_ATTR_ARG                = 36,

    ERR_INVALID_EVENT_MAP_INDEX         = 51,
    ERR_EVENT_MAP_FULL                  = 52,
    ERR_PENDING_REQ_MAP_FULL            = 53,
    ERR_REQ_TIMEOUT                     = 54,

    ERR_INVALID_SESSION_ID              = 60,
    ERR_INVALID_CAB_ID                  = 61,
    ERR_INVALID_LOCO_SPEED              = 62,
    ERR_INVALID_FGROUP_ID               = 63,
    ERR_INVALID_FUNC_ID                 = 64,
    ERR_INVALID_CV_ID                   = 65,
    ERR_INVALID_CV_MODE                 = 66,
```

```
    ERR_CV_OP_NO_ACK                    = 67,
    ERR_INVALID_BIT_POS                 = 68,
    ERR_INVALID_PACKET_LEN              = 69,
    ERR_INVALID_REPEATS                 = 70,

    ERR_DRV_FUNC_MAP_FULL               = 75,
    ERR_DRV_PUT_ERR                     = 76,
    ERR_DRV_GET_ERR                     = 77,

    ERR_CAN_BUS_INIT                    = 81,
    ERR_CAN_INVALID_MODE                = 82,
    ERR_CAN_BUS_MSG_SIZE                = 83,
    ERR_CAN_MSG_SEND                    = 84,
    ERR_CAN_MSG_RECV                    = 85,
    ERR_CAN_MSG_NO_MSG                  = 86,
    ERR_CAN_ID_COLLISION                = 87,
    ERR_CAN_ID_CHANGED                  = 88,

    ERR_EXT_BOARD_NOT_VALID             = 254,

    ERR_USER_SPECIFIC_BASE              = 128
};

//-----------------------------------------------------------------------------------------------------------
// The CAN bus mode. The PICO_PIO_xxx modes use the Raspberry Pi Pico "can2040" library, which is a software
// implementation of the CAN bus. The "can2040" library could run on the same or on the separate processor
// core. Technically, the PICO could also run the MCP2515 via the SPI interface, but so far we just use the
// software version and avoid the additional controller hardware.
//
//-----------------------------------------------------------------------------------------------------------
enum CanBusControllerMode : uint8_t {

    CAN_BUS_LIB_PICO_PIO_125K               = 1,
    CAN_BUS_LIB_PICO_PIO_250K               = 2,
    CAN_BUS_LIB_PICO_PIO_500K               = 3,
    CAN_BUS_LIB_PICO_PIO_1000K              = 4,

    CAN_BUS_LIB_PICO_PIO_125K_M_CORE        = 11,
    CAN_BUS_LIB_PICO_PIO_250K_M_CORE        = 12,
    CAN_BUS_LIB_PICO_PIO_500K_M_CORE        = 13,
    CAN_BUS_LIB_PICO_PIO_1000K_M_CORE       = 14,
};

//-----------------------------------------------------------------------------------------------------------
// "MsgPriority" defines the values for the message priority. It tracks the general definition found in the
// sendMsg routines of the LCS library. For the CAN bus, the priority is encoded in the CAN address field.
// A CAN Id consists of the CAN Id number and the priority. Messages start out with a hard coded priority and
// on message timeout are raised in their priority. This done transparently to the firmware programmer.
//
//-----------------------------------------------------------------------------------------------------------
enum MsgPriority : uint8_t {

    MSG_PRI_VERY_HIGH   = 0,
    MSG_PRI_HIGH        = 1,
    MSG_PRI_NORMAL      = 2,
    MSG_PRI_LOW         = 3
};

//-----------------------------------------------------------------------------------------------------------
// Core library callback function signatures. Callbacks are registered by the firmware at setup time and
// invoked as the communication back to the firmware layer.
//
//      LcsMsgCallback      -   is called with a LCS management message received.
//      LcsCmdCallback      -   when the command interpreter detects a non LCS command, the command line
//                              is passed on to the callback.
//      LcsTaskCallback     -   a callback for a previously registered task. The callback is invoked on
//                              the configured periodic basis.
//
//      LcsResetCallback    -   a callback invoked when a reset is performed. The npId is passed so that
//                              the callback can detect wether a port or node is the target.
//
//      LcsInitCallback     -   a callback called at initialization time, as part of "startRuntime". The
//                              npId is passed so that the callback can detect wether a port or node is
//                              the target.
//
//      LcsPfailCallback    -   a callback when a power fail situation is detected. The npId is passed
//                              so that the callback can detect wether a port or node is the target.
//
//      LcsReqCallback      -   a callback to invoke for a user request message. The callback is passed
//                              the item and a reference to the two input / output arguments.
//
//      LcsRepCallback      -   a callback to return the reply message for a previous LCS message sent. The
//                              reply can be a data reply, an ACK or NACK or a timeout error. The arguments
//                              are the item that was requested, the arguments and the return status of the
//                              operation.
//
//      LcsEventCallback    -   a callback for a received event. The arguments are the issuing npId, the
//                              event type and the optional arguments.
//
// All callback functions need to return a status, which is ALL_OK if the callback was successful.
//
//-----------------------------------------------------------------------------------------------------------
extern "C" {

    typedef uint8_t ( *LcsMsgCallback ) ( uint8_t *msg );
    typedef uint8_t ( *LcsCmdCallback ) ( char *cmdLine );
    typedef uint8_t ( *LcsTaskCallback ) ( void );

    typedef uint8_t ( *LcsResetCallback ) ( uint16_t npId );
```

```
693     typedef uint8_t ( *LcsInitCallback ) ( uint16_t npId );
694     typedef uint8_t ( *LcsPfailCallback ) ( uint16_t npId );
695
696     typedef uint8_t ( *LcsReqCallback ) ( uint8_t portId, uint8_t item, uint16_t *arg1, uint16_t *arg2 );
697     typedef uint8_t ( *LcsRepCallback ) ( uint8_t portId, uint8_t item, uint16_t arg1, uint16_t arg2, uint8_t ret );
698
699     typedef uint8_t ( *LcsEventCallback ) ( uint16_t npId, uint16_t eId, uint8_t eAction, uint16_t eData );
700 }
701
702 //-----------------------------------------------------------------------------------------------------------
703 // A  driver for an extension board is invoked through this routine signature. During setup, the correct
704 // driver label needs to be set in the driver map.
705 //
706 //-----------------------------------------------------------------------------------------------------------
707 extern "C" {
708
709     typedef uint8_t ( *LcsDrvReqFunc ) ( uint8_t boardId, uint8_t item, uint16_t *arg1, uint16_t *arg2 );
710 }
711
712 //-----------------------------------------------------------------------------------------------------------
713 // "LcsConfigDesc" is the data structure that contains initial data for setting up a node. There is the
714 // option field with bits.
715 //
716 // ??? also add the CAN bus mode ?
717 //-----------------------------------------------------------------------------------------------------------
718 struct LcsConfigDesc {
719
720     uint16_t options  = 0;
721
722 };
723
724 //-----------------------------------------------------------------------------------------------------------
725 // Library functions. The main function are the initialization and start of the LCS runtime. Between "init"
726 // and "start", the firmware should do its own setup and register the required callbacks. We will not return
727 // from the "start" routine. All call a plain C style library calls. Since we have only one instance of this
728 // library, there is not really a need to encapsulate the internal data and function pointers in a structure
729 // that is passed to each call. However, data structures are kept wherever possible local to the file and
730 // only structures that are references throughout the library file set are available externally. Perhaps
731 // one day, we encapsulate all data in a private structure for security reasons. To be determined.
732 //
733 //-----------------------------------------------------------------------------------------------------------
734 LcsConfigDesc       getConfigDefault( );
735 uint8_t             initRuntime( LcsConfigDesc *lcsConfig, CDC::CdcConfigDesc *cdcConfig );
736 void                startRuntime( );
737
738 //-----------------------------------------------------------------------------------------------------------
739 // Access the node local GET/SET/REQ items. Although we pass the node/port Id, only the port Id is actually
740 // used to identify whether we refer to a local port or the local node. Accessing a remote node / port is
741 // implemented with the LCS library message send calls.
742 //
743 //-----------------------------------------------------------------------------------------------------------
744 uint8_t             nodeGet( uint16_t npId, uint8_t item, uint16_t *arg1, uint16_t *arg2 = nullptr );
745 uint8_t             nodePut( uint16_t npId, uint8_t item, uint16_t arg1, uint16_t arg2 = 0 );
746 uint8_t             nodeReq( uint16_t npId, uint8_t item, uint16_t *arg1 = nullptr, uint16_t *arg2 = nullptr );
747
748 //-----------------------------------------------------------------------------------------------------------
749 // Function registration routines for callbacks, tasks, driver types, etc.
750 //
751 //-----------------------------------------------------------------------------------------------------------
752 void                registerLcsMsgCallback( LcsMsgCallback functionId );
753 void                registerDccMsgCallback( LcsMsgCallback functionId );
754 void                registerCmdCallback( LcsCmdCallback functionId );
755 void                registerInitCallback( LcsInitCallback handler );
756 void                registerResetCallback( LcsResetCallback handler );
757 void                registerPfailCallback( LcsPfailCallback handler );
758 void                registerEventCallback( LcsEventCallback functionId );
759 void                registerReqCallback( LcsReqCallback handler );
760 void                registerRepCallback( LcsRepCallback handler );
761 uint8_t             registerTaskCallback( LcsTaskCallback task, uint32_t interval = 0 );
762 uint8_t             registerDrvFunc( uint16_t drvType, LcsDrvReqFunc drvReqFunction );
763
764 //-----------------------------------------------------------------------------------------------------------
765 // A set of convenience functions to send an LCS message.
766 //
767 //-----------------------------------------------------------------------------------------------------------
768 uint8_t             sendCfg( uint16_t npId );
769 uint8_t             sendOps( uint16_t npId );
770 uint8_t             sendReset( uint16_t npId );
771 uint8_t             sendBusOn( );
772 uint8_t             sendBusOff( );
773 uint8_t             sendPing( uint16_t npId );
774 uint8_t             sendAck( uint16_t npId );
775 uint8_t             sendErr( uint16_t npId, uint8_t errCode, uint8_t arg1 = 0, uint8_t arg2 = 0 );
776
777 uint8_t             sendReqNodeId( uint16_t npId, uint32_t nodeUID, uint8_t flags );
778 uint8_t             sendRepNodeId( uint16_t npId, uint32_t nodeUID );
779 uint8_t             sendSetNodeId( uint16_t npId, uint32_t nodeUID );
780 uint8_t             sendNodeIdCollision( uint16_t npId, uint32_t nodeUID );
781
782 uint8_t             sendGetNode( uint16_t npId, uint8_t item, uint16_t arg1 = 0, uint16_t arg2 = 0 );
783 uint8_t             sendSetNode( uint16_t npId, uint8_t item, uint16_t arg1 = 0, uint16_t arg2 = 0 );
784 uint8_t             sendRepNode( uint16_t npId, uint8_t item, uint16_t val1, uint16_t val2  );
785 uint8_t             sendReqNode( uint16_t npId, uint8_t item, uint16_t val1, uint16_t val2  );
786
787 uint8_t             sendEventOn( uint16_t npId, uint16_t eventId );
788 uint8_t             sendEventOff( uint16_t npId, uint16_t eventId );
789 uint8_t             sendEvent( uint16_t npId, uint16_t eventId, uint16_t arg );
790
791 uint8_t             sendTrackOn( );
```

```
792 uint8_t             sendTrackOff( );
793 uint8_t             sendEstop( );
794
795 uint8_t             sendReqLoc( uint16_t locAdr, uint8_t flags  );
796 uint8_t             sendRelLoc( uint8_t sId  );
797 uint8_t             sendRepLoc( uint8_t sId, uint16_t locAdr, uint8_t spDir, uint8_t fn1 = 0, uint8_t fn2 = 0, uint8_t fn3 = 0 );
798 uint8_t             sendLocConsist( uint8_t sId, uint8_t consId, uint8_t flags );
799 uint8_t             sendQueryLoc( uint8_t sId  );
800 uint8_t             sendKeepLoc( uint8_t sId  );
801 uint8_t             sendSetLocSpDir( uint8_t sId, uint8_t spDir );
802 uint8_t             sendSetLocMode( uint8_t sId, uint8_t mode );
803 uint8_t             sendSetLocFuncOn( uint8_t sId, uint8_t fNum );
804 uint8_t             sendSetLocFuncOff( uint8_t sId, uint8_t fNum );
805 uint8_t             sendSetLocFgroup( uint8_t sId, uint8_t fGroup, uint8_t data );
806
807 uint8_t             sendSetLocCvMain( uint8_t sId, uint16_t cvId, uint8_t mode, uint8_t val );
808 uint8_t             sendSetLocCvProg( uint16_t cvId, uint8_t mode, uint8_t val );
809 uint8_t             sendReqLocCvProg( uint16_t cvId, uint8_t mode );
810 uint8_t             sendRepLocCvProg( uint16_t cvId, uint8_t val );
811
812 uint8_t             sendSetBacc( uint16_t accAdr, uint8_t flags  );
813 uint8_t             sendSetEacc( uint16_t accAdr, uint8_t val  );
814
815 uint8_t             sendDccPacket( uint8_t arg1, uint8_t arg2, uint8_t arg3 );
816 uint8_t             sendDccPacket( uint8_t arg1, uint8_t arg2, uint8_t arg3, uint8_t arg4 );
817 uint8_t             sendDccPacket( uint8_t arg1, uint8_t arg2, uint8_t arg3, uint8_t arg4, uint8_t arg5 );
818 uint8_t             sendDccPacket( uint8_t arg1, uint8_t arg2, uint8_t arg3, uint8_t arg4, uint8_t arg5, uint8_t arg6 );
819
820 uint8_t             sendDccAck( );
821 uint8_t             sendDccErr( uint8_t errCode, uint8_t arg1 = 0, uint8_t arg2 = 0 );
822
823 uint8_t             sendRawMsg( uint8_t *msgBuf );
824
825 void                printLcsMs( uint8_t *msgBuf );
826
827 //-------------------------------------------------------------------------------------------------------
828 // The driver interface. The firmware communicated with an extension board by making calls to the driver
829 // for the board. Just like for the node / port items, there are routines to GET/SET/REQ driver data and
830 // functions. The init function is only called by the library at setup time.
831 //
832 //-------------------------------------------------------------------------------------------------------
833 uint8_t             drvInit( uint8_t boardId );
834 uint8_t             drvGet( uint8_t boardId, uint8_t item, uint16_t *arg );
835 uint8_t             drvPut(uint8_t boardId, uint8_t item, uint16_t arg );
836 uint8_t             drvReq( uint8_t boardId, uint8_t item, uint16_t *arg1 = nullptr, uint16_t *arg2 = nullptr );
837
838 //-------------------------------------------------------------------------------------------------------
839 // The User Map interface. The LCS library offers a set of routines for the firmware to access the user
840 // NVM area. The size is dependent on what the actual chip on the board offers. The meaning of this data
841 // area is entirely firmware specific. Note that there are also routines for accessing the runtime data
842 // area as well as the individual extension board areas. They are declared in the internal include file.
843 //
844 //-------------------------------------------------------------------------------------------------------
845 uint8_t             usrNvmPutWord( uint32_t ofs, uint16_t word );
846 uint8_t             usrNvmGetWord( uint32_t ofs, uint16_t *word );
847 uint8_t             usrNvmPutBytes( uint32_t ofs, uint8_t *buf, uint32_t len );
848 uint8_t             usrNvmGetBytes( uint32_t ofs, uint8_t *buf, uint32_t len );
849 uint8_t             usrNvmInitArea( uint32_t ofs, uint32_t len, uint8_t val);
850 uint32_t            usrNvmGetSize( );
851
852 }; // LCS NameSpace
853
854 #endif
```

```
//-----------------------------------------------------------------------------------------------------------
//
// Layout Control System - Runtime Library internals include file
//
//-----------------------------------------------------------------------------------------------------------
// The LCS library internal definitions are all grouped in this include file. A firmware write needs to only
// include the external include file. There is nothing in here that is needed outside.
//
//-----------------------------------------------------------------------------------------------------------
//
// LCS - Core Library
// Copyright (C) 2021 - 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 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/>.
//
//-----------------------------------------------------------------------------------------------------------
#ifndef LCS_RT_LIB_INT_h
#define LCS_RT_LIB_INT_h

//-----------------------------------------------------------------------------------------------------------
// Include files. Besides the standard C libraries, there is the external LCS runtime include file, and the
// dependent code library include file.
//
//-----------------------------------------------------------------------------------------------------------
#include <stdint.h>
#include <ctype.h>
#include "LcsCdcLib.h"
#include "LcsRuntimeLib.h"

namespace LCS {

//-----------------------------------------------------------------------------------------------------------
// The LCS Runtime needs to maintain a couple of internal data structures. As a general concept, most of the
// data areas are stored in the NVM and shadowed by a memory copy. Upon reset or power up the memory areas
// are initialized from their NVM counter parts. Data that needs to be changed permanently is flushed from
// memory to NVM so that it is the initial value on the next restart. All data is stored in controller native
// endianness. Only the messages exchanged via the LcsMsgBus are transmitted in big endian order.
//
// The NVM layout is a fixed one. We have the nodeMap starting at offset zero, The CDC map starting at offset
// 0x200, the portMap starting at offset 0x400, the attributeMap starting at offset 0x800 and the eventMap
// at offset 0x1000. The system area is in total 8 Kbytes. The optional user map occupies all the remaining
// bytes in the NVM and starts at 0x2000. A firmware programmer can access the system as well as the user
// data areas. However, note that dangerous things can be done when modifying the system area directly.
//
//        0x0000    :-------------------------------------------:
//                  :                                           :
//                  :       Node Map                            :
//                  :                                           :
//        0x0200    :-------------------------------------------:
//                  :                                           :
//                  :       CDC Map                             :
//                  :                                           :
//        0x0400    :-------------------------------------------:
//                  :                                           :
//                  :       Port Map                            :
//                  :                                           :
//        0x0800    :-------------------------------------------:
//                  :                                           :
//                  :       Attribute Map                       :
//                  :                                           :
//        0x1000    :-------------------------------------------:
//                  :                                           :
//                  :       Event Map                           :
//                  :                                           :
//        0x2000    :-------------------------------------------:
//                  :                                           :
//                  :                                           :
//                  :                                           :
//                  :       Optional User Map                   :
//                  :                                           :
//                  :                                           :
//                  :                                           :
//        0xNNNN    :-------------------------------------------:
//
// The node map and port map do not fill the entire area allocated for them. Yet. For future developments,
// each area has some spare room. The attribute map contains the variables for the node and ports. Each
// entity has 64 attributes max, the attribute map is 1Kbyte in total. By putting all attributes in one
// area, access to an attribute value is easy to calculate and quick.
//
// The event map is an area with 4-byte entries. A node can keep track of up to 1024 event/port pairs.
// The event map is a sorted map, lookup is done via a binary search. Finally, the optional user map
// data area is just a set of bytes with a structure only know to the firmware designer.
//
// In general each of the runtime areas could have also been designed in a way that they are dynamically
// configurable in size. For example, a port map could be up to 15 ports but also less. The attributes of
// a node or port could be up to 64 attributes or less. Considering the size and price of NVM chips as well
// as the memory size of the supported controller platforms, the current implementation uses fixed sizes
// for each area, avoiding configuration complexity.
//
//-----------------------------------------------------------------------------------------------------------
const uint16_t  MAX_NODE_DATA_BLOCKS            = 16;
```

```
99  const uint16_t  MAX_ATTR_MAP_ENTRIES            = 64;
100 const uint16_t  MAX_PORT_MAP_ENTRIES            = 15;
101 const uint16_t  MAX_EVENT_MAP_ENTRIES           = 1024;
102 const uint16_t  MAX_TASK_MAP_ENTRIES            = 16;
103
104 const uint16_t  MAX_LCS_MSG_SIZE                = 8;
105 const uint16_t  MAX_NODE_NAME_SIZE              = 16;
106 const uint16_t  MAX_PORT_NAME_SIZE              = 16;
107 const uint16_t  MAX_BOARD_NAME_SIZE             = 16;
108 const uint16_t  MAX_COMMAND_LINE_SIZE           = 256;
109
110 const uint16_t  MAX_EXT_BOARD_MAP_ENTRIES       = 4;
111 const uint16_t  MAX_PENDING_REQ_MAP_ENTRIES     = 8;
112 const uint16_t  EVENT_DELAY_TICK_MILLIS         = 32;
113
114 const uint8_t   MAX_DRV_TYPES                   = 8;
115 const uint8_t   MAX_EXT_BOARDS                  = 4;
116 const uint8_t   MAX_DRV_DATA_SIZE               = 64;
117
118 const uint16_t  NVM_NODE_MAP_START              = 0;
119 const uint16_t  NVM_CDC_MAP_START               = 0x200;
120 const uint16_t  NVM_PORT_MAP_START              = 0x400;
121 const uint16_t  NVM_NODE_DATA_START             = 0x800;
122 const uint16_t  NVM_EVENT_MAP_START             = 0x1000;
123 const uint16_t  NVM_USER_MAP_START              = 0x2000;
124 const uint16_t  NVM_RUNTIME_AREA_SIZE           = 0x2000;
125
126 //-------------------------------------------------------------------------------------------------------
127 // The nodeMap on NVM has two locations with a "magic" word. We simply read in a nodeMap and check these
128 // two locations for the magic words. If found, the area was configured before. It would be quite unlikely
129 // that a random NVM content has these two words at the right spot. In a similar way, we have two magic
130 // words for the NVM in an extension board. Same idea, same logic. But even if the area was configured
131 // before, it does not automatically mean that all the data is correct. Further checking will be done
132 // during startup.
133 //
134 //-------------------------------------------------------------------------------------------------------
135 const uint16_t NVM_MWORD_1 = (uint16_t) ( 'L' << 8 ) + 'C';
136 const uint16_t NVM_MWORD_2 = (uint16_t) ( 'S' << 8 ) + '0';
137
138 //-------------------------------------------------------------------------------------------------------
139 // The node states. The node starts in the INIT state and once all is initialized and registered ends up in
140 // the OPS or CFG mode.
141 //
142 //  NS_NIL            -
143 //  NS_FAIL           -   The node startup failed.
144 //  NS_PFAIL          -   The node startup detected that we come up after a power fail.
145 //  NS_INIT           -   The node entered the startup state.
146 //  NS_REGISTER       -   The node entered the node register state, awaiting a nodeId.
147 //  NS_COLLISION      -   The node detected a nodeId collision on the LCS bus.
148 //  NS_HALTED         -   The node was halted.
149 //  NS_CONFIG         -   The node is in configuration mode.
150 //  NS_OPERATE        -   The node is on operations mode.
151 //
152 //-------------------------------------------------------------------------------------------------------
153 enum LcsNodeState : uint16_t {
154
155     NS_NIL              = 0,
156     NS_FAIL             = 1,
157     NS_PFAIL            = 2,
158     NS_INIT             = 3,
159     NS_REGISTER         = 4,
160     NS_COLLISION        = 5,
161     NS_HALTED           = 6,
162     NS_CONFIG           = 7,
163     NS_OPERATE          = 8
164 };
165
166 //-------------------------------------------------------------------------------------------------------
167 // Nodes, ports and drivers are accessed with three main routines, GET, SET and REQ.
168 //
169 // GET - the get routine will use the item numbers to retrieve the data labelled by the item.
170 //
171 // SET - the set routine will use the item numbers to set the value. Note that not all items that can be
172 // read can also be written to. An attempt will result in an error return.
173 //
174 // REQ - the request call will transmit the request parameters to the node / port / driver where a registered
175 // callback or the driver entry point will be invoked. The result is returned via the parameters.
176 //
177 // One argument is the item. Items range from  0 ... 255 and are defined as follows:
178 //
179 //   0          -   NIL item, not used
180 //   1  ..  63  -   Node / port / driver reserved area items, global items for GET/SET/REQ requests.
181 //  64  .. 127  -   User defined items, specific meaning, accessed via the REQ routine.
182 // 128  .. 191  -   Node / port / driver data attributes returned from MEM for GET/SET.
183 // 192  .. 255  -   Node / port / driver data attributes copied from NVM to MEM for GET, copied from MEM to NVM
184 //                  for SET. The item range mirrors items 128 - 191. For example, 128 and 192 refer to the same
185 //                  attribute. Note that for a SET on a driver the HW needs to be enabled.
186 //
187 // The items are defined in the external include file. This part here defined the boundaries for internal
188 // checking.
189 //
190 //-------------------------------------------------------------------------------------------------------
191 enum ItemRanges : uint8_t {
192
193     IR_NIL                      = 0,
194
195     IR_LIB_MAP_RANGE_START      = 1,
196     IR_LIB_MAP_RANGE_END        = 63,
197
```

```
    IR_USER_RANGE_START         = 64,
    IR_USER_RANGE_END           = 127,

    IR_ATTR_MEM_RANGE_START     = 128,
    IR_ATTR_MEM_RANGE_END       = 191,

    IR_ATTR_NVM_RANGE_START     = 192,
    IR_ATTR_NVM_RANGE_END       = 255,

    IR_MAX_ITEMS                = 255
};

//-----------------------------------------------------------------------------------------------------------
// "LcsMsgBusCAN" is the CAN bus interface. The two key routines are the send and receive routines. For
// debugging purposes a debug level can be set so that diagnostic messages are displayed to the console.
//
//-----------------------------------------------------------------------------------------------------------
struct LcsMsgBusCAN {

    public:

    uint8_t     init( uint16_t canId, uint8_t pinRx, uint8_t pinTx, uint8_t fMode = CAN_BUS_LIB_PICO_PIO_125K );

    uint8_t     sendLcsMsg ( uint8_t *msgBuf, uint8_t msgPri = MSG_PRI_NORMAL );
    uint8_t     receiveLcsMsg( uint8_t *msg );
    void        setDebugLevel( uint8_t level );

    private:

    uint16_t  canId = 0;
};


//-----------------------------------------------------------------------------------------------------------
// Each NVM memory, ie the NVM on the controller board or an extension board, starts with the header data
// structure. This structure contains information to detect that the NVM was formatted, as well as some
// hardware specific data to identify the board and relevant chips on it. The data in this header must be
// "programmed" during a board setup. This is easily accomplished trough console commands and needs of
// course only be done once per board. The data structure size is 32 bytes.
//
//-----------------------------------------------------------------------------------------------------------
struct LcsNvmHeader {

    uint16_t    magicWord1                      = NVM_MWORD_1;
    uint16_t    boardType                       = BT_NIL;
    uint16_t    boardVersion                    = 0;
    uint16_t    controllerFamily                = CF_FAM_RPICO;
    uint16_t    nvmChipFamily                   = CF_FAM_MICROCHIP;
    uint16_t    reservedArea[ 10 ]              = { 0 };
    uint16_t    magicWord2                      = NVM_MWORD_2;
};

//-----------------------------------------------------------------------------------------------------------
// Every LCS board uses the CDC layer to access the controller hardware. The CDC descriptor contains the
// pin configuration data. Currently, the CDC config data is set directly by the application. We copy this
// data to the "cfg" structure. One day, we may store this data in the descriptor. So far, this is more of
// a place holder.
//-----------------------------------------------------------------------------------------------------------
struct LcsCdcMap {

    CDC::CdcConfigDesc cfg;
};

//-----------------------------------------------------------------------------------------------------------
// An LCS node and the ports on the node each have an area of variables that are in memory as well as in
// the node NVM. Typical usage examples are configuration items such as a limit value. Upon power up or
// reset, the node data from the NVM area is copied to the MEM counterpart. Although the node and port
// attributes are logically part of the portMap and nodeMap, they are kept in this separate structure,
// which then is a nice 2 Kbyte block of 16 areas of 64 words each and thus are very easy to access.
//
//-----------------------------------------------------------------------------------------------------------
struct LcsNodeData {

    uint16_t map[ MAX_PORT_MAP_ENTRIES + 1 ][ MAX_ATTR_MAP_ENTRIES ] = { 0 };
};

//-----------------------------------------------------------------------------------------------------------
// The port map contains an array of ports, each described by a port map entry. The portMap entry contains
// the fields that deal with the actual event received. There are fields for the sending node, the event
// and its action. An event can also be invoked with a delay time. The are fifteen entries in the port map.
// The portMap starts fixed at NVM offset 0x1000. Each port also has an area of attributes, which are
// stored in the data block area.
//
//-----------------------------------------------------------------------------------------------------------
struct LcsPortMapEntry {

    uint16_t  options                       = 0;
    uint16_t  flags                         = 0;
    uint16_t  type                          = 0;

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

    char      name[ MAX_PORT_NAME_SIZE  ]   = { 0 };
};
```

```
297
298 struct LcsPortMap {
299
300     LcsPortMapEntry map[ MAX_PORT_MAP_ENTRIES ];
301 };
302
303 //------------------------------------------------------------------------------------------------------
304 // The event map entry contains the mapping from eventId to portId. Every port interested in a certain event
305 // will have an entry in the event map. It is a sorted table of event and port pairs. A port id of zero
306 // refers to all ports with the event Id. This table is searched for an incoming event to find the ports
307 // that are interested in the event.
308 //
309 //------------------------------------------------------------------------------------------------------
310 struct LcsEventMapEntry {
311
312     uint16_t eventId  = NIL_EVENT_ID;
313     uint16_t portId   = NIL_PORT_ID;
314 };
315
316 struct LcsEventMap {
317
318     LcsEventMapEntry    map[ MAX_EVENT_MAP_ENTRIES ];
319 };
320
321 //------------------------------------------------------------------------------------------------------
322 // The first locations of the NVM area on the controller board NVM chip represent the nodeMap. It is the
323 // heart of all data on the node. When bringing up a node, we read in the node map from the NVM. The first
324 // check is whether the nodeMap read is a valid nodeMap.
325 //
326 //------------------------------------------------------------------------------------------------------
327 struct LcsNodeMap {
328
329     //------------------------------------------------------------------------------------------------------
330     // NMV header. We read this in first an check for validity.
331     //------------------------------------------------------------------------------------------------------
332     LcsNvmHeader    head;
333
334     //------------------------------------------------------------------------------------------------------
335     // Node data.
336     //
337     //------------------------------------------------------------------------------------------------------
338     uint16_t        nodeState                       = NS_NIL;
339     uint16_t        nodeOptions                     = 0;
340     uint16_t        nodeFlags                       = 0;
341     uint16_t        nodeId                          = NIL_NODE_ID;
342     uint32_t        nodeUID                         = 0L;
343     uint16_t        nodeType                        = NIL_NODE_TYPE;
344     uint16_t        nodeSwVersion                   = 0;
345     uint16_t        nodeSwPatchLevel                = 0;
346     uint16_t        nodeRestartCnt                  = 0;
347     uint32_t        nodeSystemTime                  = 0;
348     uint16_t        nodeMapSize                     = sizeof( LcsNodeMap );
349     char            name[ MAX_NODE_NAME_SIZE ]      = { 0 };
350
351     //------------------------------------------------------------------------------------------------------
352     // Runtime area offsets in the NVM. We also keep track of the data structure sizes and check when we
353     // read in the maps that they match a give library version.
354     //
355     //------------------------------------------------------------------------------------------------------
356     uint16_t        nvmNodeMapOfs                   = NVM_NODE_MAP_START;
357     uint16_t        nvmNodeMapSize                  = sizeof( LcsNodeMap );
358
359     uint16_t        nvmCdcMapOfs                    = NVM_CDC_MAP_START;
360     uint16_t        nvmCdcMapSize                   = sizeof( LcsCdcMap );
361
362     uint16_t        nvmPortMapOfs                   = NVM_PORT_MAP_START;
363     int16_t         nvmPortMapSize                  = sizeof( LcsPortMap );
364
365     uint16_t        nvmNodeDataOfs                  = NVM_NODE_DATA_START;
366     uint16_t        nvmNodeDataSize                 = sizeof( LcsNodeData );
367
368     uint16_t        nvmEventMapOfs                  = NVM_EVENT_MAP_START;
369     uint16_t        nvmEventMapSize                 = sizeof( LcsEventMap );
370
371     uint16_t        nvmUserMapOfs                   = NVM_USER_MAP_START;
372     uint16_t        nvmUserMapSize                  = 0;
373
374     uint32_t        nvmMemSize                      = NVM_RUNTIME_AREA_SIZE;
375
376     //------------------------------------------------------------------------------------------------------
377     // The number of entries in the core areas and a high water mark.
378     //
379     //------------------------------------------------------------------------------------------------------
380     uint16_t        portMapEntries                  = MAX_PORT_MAP_ENTRIES;
381     uint16_t        portMapHwm                      = 0;
382
383     uint16_t        eventMapEntries                 = MAX_EVENT_MAP_ENTRIES;
384     uint16_t        eventMapHwm                     = 0;
385
386     uint16_t        taskMapEntries                  = MAX_TASK_MAP_ENTRIES;
387     uint16_t        taskMapHwm                      = 0;
388
389     uint16_t        pendingMapEntries               = MAX_PENDING_REQ_MAP_ENTRIES;
390     uint16_t        pendingMapHwm                   = 0;
391
392     uint16_t        drvFuncMapEntries               = MAX_DRV_TYPES;
393     uint16_t        drvFuncMapHwm                   = 0;
394
395     uint16_t        drvMapEntries                   = MAX_EXT_BOARD_MAP_ENTRIES;
```

```
396     uint16_t        drvMapHwm                       = 0;
397 };
398 
399 //-----------------------------------------------------------------------------------------------------
400 // The LCS runtime communicates back to the firmware via callbacks. There are global callbacks for message
401 // receipt and events as well as callbacks for the node and ports that can be registered.
402 //
403 //-----------------------------------------------------------------------------------------------------
404 struct LcsCallbackMap {
405 
406     LcsMsgCallback          lcsMsgCallback          = nullptr;
407     LcsMsgCallback          dccMsgCallback          = nullptr;
408     LcsCmdCallback          cmdLineCallback         = nullptr;
409     LcsEventCallback        eventCallback           = nullptr;
410 
411     LcsInitCallback         initCallback            = nullptr;
412     LcsResetCallback        resetCallback           = nullptr;
413     LcsPfailCallback        pfailCallback           = nullptr;
414 
415     LcsReqCallback          reqCallback             = nullptr;
416     LcsRepCallback          repCallback             = nullptr;
417 };
418 
419 //-----------------------------------------------------------------------------------------------------
420 // The core library maintains an array of periodic task items. To balance the needs of other core library
421 // internal periodic tasks, such as checking for incoming messages, the periodic tasks are run one at a
422 // time, round robin, with the other internal tasks interleaving. The structure maintains the task procedure
423 // label, the time it ran the last time, and the interval between invocations. Note that the timing is not
424 // very accurate, but it is guaranteed that a task will eventually run when the interval is reached.
425 //
426 //-----------------------------------------------------------------------------------------------------
427 struct LcsPTaskMapEntry {
428 
429     LcsTaskCallback     task          = nullptr;
430     uint32_t            timeStamp     = 0;
431     uint32_t            interval      = 0;
432 };
433 
434 struct LcsTaskMap {
435 
436     LcsPTaskMapEntry    map[ MAX_TASK_MAP_ENTRIES ];
437 };
438 
439 //-----------------------------------------------------------------------------------------------------
440 // The pending request map keeps track of outstanding requests to another node. We add an entry when our
441 // node sends a "REQ" type packet and clear the entry when the reply comes in. The idea is that we only
442 // invoke the callback when we expect a reply. Additionally, there is a timeout value, so that we can
443 // can invoke the reply callback with a timeout message if requested.
444 //
445 //-----------------------------------------------------------------------------------------------------
446 struct LcsPendingReqEntry {
447 
448     uint16_t  npId;
449     int32_t   reqTimeoutTs;
450 };
451 
452 struct LcsPendingReqMap {
453 
454     LcsPendingReqEntry map[ MAX_PENDING_REQ_MAP_ENTRIES ];
455 };
456 
457 //-----------------------------------------------------------------------------------------------------
458 // Each extension board will have a NVM to store the board configuration data. Similar to the node map of
459 // the controller board, this extension board will have a data structure that is read at initialization time.
460 // The structure of this data is rather simple. We have the common 8-word header which describes the board in
461 // general an area which contains driver relevant information. The board type will tell the setup routines
462 // what driver to load for the extension board. The driver data area is entirely driver specific and the
463 // meaning is only know to the driver software. At startup time, all we have to do then is locate the board
464 // type, load the  respective driver and let the driver code do whatever needs to be done according to the
465 // data area content.
466 //
467 // Note that the extension board NVM data is "read only". To write to it, a jumper is set on the board. The
468 // data area is configured and then the jumper should be removed. This does however not mean that that data
469 // once it is loaded during setup cannot be changed during operations. For example, the driver area is the
470 // "working area" for the driver to keep temporary values. At node restart, all data is set back to the NVM
471 //  data configured on the extension board chip.
472 //
473 //-----------------------------------------------------------------------------------------------------
474 struct LcsDrvBoardDesc {
475 
476     LcsNvmHeader    head;
477     uint16_t        driverData[ MAX_DRV_DATA_SIZE ]  = { 0 };
478 };
479 
480 //-----------------------------------------------------------------------------------------------------
481 // An extension board is accessed via a dedicated driver. The firmware is required to register the available
482 // drivers with the runtime. The type and function label are kept in the driver label map. This data is
483 // used when a board os detected to select the correct driver.
484 //
485 //-----------------------------------------------------------------------------------------------------
486 struct LcsDrvFuncEntry {
487 
488     uint16_t        drvType = BT_NIL;
489     LcsDrvReqFunc   drvFunc = nullptr;
490 };
491 
492 struct LcsDrvFuncMap {
493 
494     LcsDrvFuncEntry map[ MAX_DRV_TYPES ] = { 0 };
```

```
495 };
496
497 //-----------------------------------------------------------------------------------------------------------
498 // The runtime library maintains a driver table, which has for each of the extension boards an entry. The
499 // first board has an index of zero.  While the drivers are set regardless of the order of the extension
500 // boards, the boardId would change with the order of extension boards connected. A firmware either needs
501 // to insist on the correct order or map the extension boards regardless of order.
502 //
503 // The entry contains a set of flags about the driver, the procedure label for the driver code and the
504 // extension board descriptor, which is read in from the extension board NVM area. During startup all
505 // extension boards will be located, if there are any. For each board the correct driver procedure label
506 // will be stored in the driver map entry.
507 //
508 // If the extension board descriptor is invalid, the driver map entry is marked as failed. We can however
509 // still access the data area from configuration tools, when the jumper to enable writing to the board is
510 // set.
511 //
512 //-----------------------------------------------------------------------------------------------------------
513 struct LcsDrvEntry {
514
515     uint16_t            flags       = 0;
516     uint16_t            lastErr     = 0;
517     LcsDrvReqFunc       drvFunc     = nullptr;
518
519     LcsDrvBoardDesc     extBoard;
520 };
521
522 struct LcsDrvMap {
523
524     LcsDrvEntry     map[ MAX_EXT_BOARDS ];
525 };
526
527 //-----------------------------------------------------------------------------------------------------------
528 // The LCS runtime routine signatures of routines used across the different source files.
529 //
530 // ??? keep this list short... maybe keep local to each file....
531 //-----------------------------------------------------------------------------------------------------------
532 uint8_t     configNvm( CDC::CdcConfigDesc *ci );
533
534 uint8_t     rtNvmPutWord( uint32_t ofs, uint16_t word );
535 uint8_t     rtNvmGetWord( uint32_t ofs, uint16_t *word );
536 uint8_t     rtNvmPutBytes( uint32_t ofs, uint8_t *buf, uint32_t len );
537 uint8_t     rtNvmGetBytes( uint32_t ofs, uint8_t *buf, uint32_t len );
538 uint8_t     rtNvmClearArea( uint32_t ofs, uint32_t len, uint8_t val = 0 );
539 uint32_t    rtNvmGetSize( );
540
541 uint8_t     extNvmPutWord( uint8_t boardId, uint32_t ofs, uint16_t word );
542 uint8_t     extNvmGetWord( uint8_t boardId, uint32_t ofs, uint16_t *word );
543 uint8_t     extNvmPutBytes( uint8_t boardId, uint32_t ofs, uint8_t *buf, uint32_t len );
544 uint8_t     extNvmGetBytes( uint8_t boardId, uint32_t ofs, uint8_t *buf, uint32_t len );
545 uint8_t     extNvmClearArea( uint8_t boardId, uint32_t ofs, uint32_t len, uint8_t val = 0 );
546 uint32_t    extNvmGetSize( );
547
548 uint8_t     resetNode( uint16_t npId );
549
550 uint8_t     syncEventMap( );
551 uint8_t     addEvent( uint16_t eventId, uint16_t portId = NIL_PORT_ID );
552 uint8_t     removeEvent( uint16_t eventId, uint16_t portId = NIL_PORT_ID );
553 int         searchEvent( uint16_t eventId, uint16_t portId = NIL_PORT_ID );
554 uint8_t     getMemEmapEntry( uint16_t index, uint16_t *evId, uint16_t *pId );
555
556 void        handleMsgLcsMgt( uint8_t *msg );
557 void        handleMsgEvent( uint8_t *msg );
558
559 uint8_t     setupSerialCommand( );
560 uint8_t     handleSerialCommand( );
561
562 void        handleNodeState( );
563
564 } // namespace LCS
565
566 #endif
```

```
1  //-----------------------------------------------------------------------------------------------------------------
2  //
3  // "LcsMsgBusCAN" - CAN Bus Interface for Raspberry PI Pico
4  //
5  //-----------------------------------------------------------------------------------------------------------------
6  // The "LcsMsgBusCAN" object implements the LCS message bus as a CAN bus. The CAN bus is a widely established
7  // bus, which is quite robust. We use the standard CAN bus with a maximum CAN Id of 29 bits. In our case the
8  // 16 bit node / port ID along with a 2 bit priority field is used as the CAN address.
9  //
10 // On the PICO, there is a library, "can2040", available that implements the CAN bus protocol in software,
11 // using the PICO PIO state machines. This saves us an external controller. In addition, we allow for the
12 // option to run the CAN bus state machine on a separate core. This is highly recommend as the LCS Runtime
13 // has a lot of other things to do. Using a queue from the PICO C++ SDK, the core running the CAN state
14 // machine will just queue the received message to be picked up by the other core when ready.
15 //
16 //-----------------------------------------------------------------------------------------------------------------
17 //
18 // LCS - Can Bus Interface Library
19 // Copyright (C) 2022 - 2024,  Helmut Fieres
20 //
21 // This program is free software: you can redistribute it and/or modify it under the terms of the GNU
22 // General Public License as published by the Free Software Foundation, either version 3 of the License,
23 // or any later version.
24 //
25 // This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even
26 // the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public
27 // License for more details. You should have received a copy of the GNU General Public License along with
28 // this program.  If not, see <http://www.gnu.org/licenses/>.
29 //
30 //-----------------------------------------------------------------------------------------------------------------
31 #include <stdio.h>
32 #include "pico/stdlib.h"
33 #include "pico/stdio.h"
34 #include "pico/util/queue.h"
35 #include "pico/multicore.h"
36
37 #include "LcsRuntimeLib.h"
38 #include "LcsRtLibInt.h"
39
40 //-----------------------------------------------------------------------------------------------------------------
41 // The can2040 is a C library. Make it extern C, otherwise the linker gets confused.
42 //
43 //-----------------------------------------------------------------------------------------------------------------
44 extern "C" {
45
46     #include "./Can2040Lib/can2040.h"
47 }
48
49 //-----------------------------------------------------------------------------------------------------------------
50 // The debug mask. See the internal include file for details.
51 //
52 //-----------------------------------------------------------------------------------------------------------------
53 namespace LCS {
54
55     extern uint16_t debugMask;
56 };
57
58 //-----------------------------------------------------------------------------------------------------------------
59 // The name space for file local declarations.
60 //
61 //-----------------------------------------------------------------------------------------------------------------
62 namespace {
63
64 using namespace LCS;
65
66 //-----------------------------------------------------------------------------------------------------------------
67 // The maximum message length of a CAN bus ( and LCS ) message. The LCS library still uses the "classic"
68 // CAN bus message size. Finally, the CAN bus library for the RP2040 needs a static opaque structure. We also
69 // need a receiver queue for storing the received messages when they come in.
70 //
71 //-----------------------------------------------------------------------------------------------------------------
72 const uint8_t   MAX_CAN_MSG_SIZE  = 8;
73 const uint8_t   RX_QUEUE_SIZE     = 4;
74 const uint8_t   TX_RETRY_TIMEOUT  = 5;
75
76 //-----------------------------------------------------------------------------------------------------------------
77 // The setup and start of the CAN Bus can run on ether core 0 or core 1, depending whether a multi-core
78 // implementation is desired. The "Can2040ConfigDesc" structure holds all the necessary configuration data
79 // for the initialization routine to use.
80 //
81 //-----------------------------------------------------------------------------------------------------------------
82 struct Can2040ConfigDesc {
83
84     uint32_t        mcPioNum;
85     uint32_t        mcSysClock;
86     uint32_t        mcBitRate;
87     uint32_t        mcRxPin;
88     uint32_t        mcTxPin;
89     can2040_rx_cb   mcRxCallback;
90     uint32_t        mcRxQueueSize;
91     bool            mcRunOnCore1;
92     bool            mcSetupOK;
93 };
94
95 //-----------------------------------------------------------------------------------------------------------------
96 // File local variables and constants.
97 //
98 //-----------------------------------------------------------------------------------------------------------------
```

```
99  Can2040ConfigDesc       cfg;
100 struct can2040          cBus;
101 queue_t                 rxQueue;
102
103 //-----------------------------------------------------------------------------------------------------------
104 // The "buildCanBusMsgHeader" constructs the canId header for the message. It encodes the canId itself and
105 // flags such as EXT or RTR.
106 //
107 //-----------------------------------------------------------------------------------------------------------
108 inline uint32_t buildCanBusMsgHeader( uint16_t canId, uint8_t msgPri, bool RTR = false ) {
109
110     uint32_t header = canId | ((uint32_t)( msgPri & 0x3 ) << 16 ) | 0x80000000;
111
112     if ( RTR ) header |=  0x40000000;
113
114     return ( header );
115 }
116
117 //-----------------------------------------------------------------------------------------------------------
118 // The interrupt signature to register with the RP2040 for PIO interrupts. The interrupt handler itself is
119 // provided by the can2040 library.
120 //
121 //-----------------------------------------------------------------------------------------------------------
122 void CanBusPIOIrqHandler( ) {
123
124     can2040_pio_irq_handler( &cBus );
125 }
126
127 //-----------------------------------------------------------------------------------------------------------
128 // For each messages transmitted or received this callback is invoked from within the interrupt handler, so
129 // all we can do is a quick non-blocking action. The callback allows to react to a message sent, a message
130 // received and an internal buffer overflow error.
131 //
132 // The callback could be used to filter messages directly at this stage. Only messages that concern this
133 // node should be processed. Easy said, but perhaps no so easy to do. We can basically to filtering at the
134 // higher message bus level or at the lower layers. The benefit for doing it here is that when we run at the
135 // other core, the main core is relieved even further. To think about one day.
136 //
137 //-----------------------------------------------------------------------------------------------------------
138 void canBusEventCallback( struct can2040 *cd, uint32_t notify, struct can2040_msg *msg ) {
139
140     if ( notify == CAN2040_NOTIFY_RX ) {
141
142         if ( ! queue_try_add( &rxQueue, msg )) {
143
144             // ??? we could not add ... what to do ?
145         }
146     }
147     else if ( notify == CAN2040_NOTIFY_TX ) {
148
149         // ??? transmit completed successfully
150     }
151     else if ( notify == CAN2040_NOTIFY_ERROR ) {
152
153         // ??? internal buffer overflow ... what to do ?
154     }
155 }
156
157 //-----------------------------------------------------------------------------------------------------------
158 // "canBusCore" is the routine that encapsulates the can2040 setup and launch work. For the multi-core
159 // version it needs to be a routine that can be called from the current core or be launched on the other
160 // core. The routine communicates the successful setup with a boolean flag in the configuration descriptor.
161 // Note that the setup routine must be a void procedure with no parameters. This is expected by the launch
162 // routine in the PICO C++ SDK.
163 //
164 //-----------------------------------------------------------------------------------------------------------
165 void canBusCore( ) {
166
167     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_CAN_BUS )) {
168
169         printf( "canBusSetup -> pio: %d, clk: %d, bitRate: %d, rxPin: %d, txPin: %d, cb: %u, rxQS: %d, MC: %d\n",
170                 cfg.mcPioNum, cfg.mcSysClock, cfg.mcBitRate,
171                 cfg.mcRxPin, cfg.mcTxPin, cfg.mcRxCallback,
172                 cfg.mcRxQueueSize, cfg.mcRunOnCore1 );
173     }
174
175     queue_init( &rxQueue, sizeof( can2040_msg ), cfg.mcRxQueueSize );
176
177     can2040_setup( &cBus, cfg.mcPioNum );
178     can2040_callback_config( &cBus, cfg.mcRxCallback );
179
180     if ( cfg.mcPioNum == 0 ) {
181
182         irq_set_exclusive_handler( PIO0_IRQ_0, CanBusPIOIrqHandler );
183         irq_set_priority( PIO0_IRQ_0, 1 );
184         irq_set_enabled( PIO0_IRQ_0, true );
185     }
186     else if ( cfg.mcPioNum == 1 ) {
187
188         irq_set_exclusive_handler( PIO1_IRQ_0, CanBusPIOIrqHandler );
189         irq_set_priority( PIO1_IRQ_0, 1 );
190         irq_set_enabled( PIO1_IRQ_0, true );
191     }
192
193     can2040_start( &cBus, cfg.mcSysClock, cfg.mcBitRate, cfg.mcRxPin, cfg.mcTxPin );
194
195     cfg.mcSetupOK = true;
196
197    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_CAN_BUS )) {
```

```
198
199         printf( "CAN Bus Initialized, runs on Core: %d\n", get_core_num( ));
200     }
201
202     if ( cfg.mcRunOnCore1 ) {
203
204         while ( true ) tight_loop_contents( );
205     }
206 }
207
208 }; // namespace
209
210
211 //---------------------------------------------------------------------------------------------------------------
212 // The LCS name space CanBus Object methods declared in this file.
213 //
214 //---------------------------------------------------------------------------------------------------------------
215 namespace LCS {
216
217 //---------------------------------------------------------------------------------------------------------------
218 // "init" is called to setup the CAN bus interface. We will first check the parameters and setup the CAN bus.
219 // Next set up the interrupt handler and start the CAN bus processing. This is also the time to set the
220 // initial canId.
221 //
222 //---------------------------------------------------------------------------------------------------------------
223 uint8_t LcsMsgBusCAN::init( uint16_t canId, uint8_t rxPin, uint8_t txPin, uint8_t fMode ) {
224
225     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_CAN_BUS )) {
226
227         printf( "Init Can Bus -> Node: %d, Rx: %d, Tx: %d, Mode: %d\n", canId, rxPin, txPin, fMode );
228     }
229
230     if (( rxPin == CDC::UNDEFINED_PIN ) || ( txPin == CDC::UNDEFINED_PIN )) return ( ERR_CAN_BUS_INIT );
231
232     this -> canId = canId;
233
234     cfg.mcSetupOK       = true;
235     cfg.mcRxPin         = rxPin;
236     cfg.mcTxPin         = txPin;
237     cfg.mcRxCallback    = canBusEventCallback;
238     cfg.mcSysClock      = CDC::getCpuFrequency( );
239     cfg.mcPioNum        = 0;
240     cfg.mcRxQueueSize   = RX_QUEUE_SIZE;
241
242     cfg.mcRunOnCore1    =  (( fMode == CAN_BUS_LIB_PICO_PIO_125K_M_CORE ) ||
243                             ( fMode == CAN_BUS_LIB_PICO_PIO_250K_M_CORE ) ||
244                             ( fMode == CAN_BUS_LIB_PICO_PIO_500K_M_CORE ) ||
245                             ( fMode == CAN_BUS_LIB_PICO_PIO_1000K_M_CORE ));
246
247     switch ( fMode ) {
248
249         case CAN_BUS_LIB_PICO_PIO_125K:
250         case CAN_BUS_LIB_PICO_PIO_125K_M_CORE:   cfg.mcBitRate = 125000; break;
251
252         case CAN_BUS_LIB_PICO_PIO_250K:
253         case CAN_BUS_LIB_PICO_PIO_250K_M_CORE:   cfg.mcBitRate = 250000; break;
254
255         case CAN_BUS_LIB_PICO_PIO_500K:
256         case CAN_BUS_LIB_PICO_PIO_500K_M_CORE:   cfg.mcBitRate = 500000; break;
257
258         case CAN_BUS_LIB_PICO_PIO_1000K:
259         case CAN_BUS_LIB_PICO_PIO_1000K_M_CORE:  cfg.mcBitRate = 1000000; break;
260
261         default: cfg.mcSetupOK = false;
262     }
263
264     if ( cfg.mcSetupOK ) {
265
266         if ( cfg.mcRunOnCore1 ) multicore_launch_core1( canBusCore );
267         else canBusCore( );
268     }
269
270     return (( cfg.mcSetupOK ) ? ALL_OK : ERR_CAN_BUS_INIT );
271 }
272
273 //---------------------------------------------------------------------------------------------------------------
274 // "sendLcsMsg" will send a data packet. We are passed the message buffer and the message priority. The
275 // message length is encoded in the first message byte, which represents the LCS message opCode as well as
276 // the length of the message. The message has a certain initial priority. When we cannot send the message
277 // right away, the priority is raised. When we cannot send at the highest priority, the message send
278 // failed.
279 //
280 //---------------------------------------------------------------------------------------------------------------
281 uint8_t LcsMsgBusCAN::sendLcsMsg ( uint8_t *msgBuf, uint8_t msgPri ) {
282
283     can2040_msg msg;
284
285     msg.id  = buildCanBusMsgHeader( canId, msgPri );
286     msg.dlc = ( msgBuf[ 0 ] >> 5 ) + 1;
287
288     for ( uint32_t i = 0; i < msg.dlc; i++ ) msg.data[ i ] = msgBuf[ i ];
289
290     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_CAN_BUS )) {
291
292         printf( "CAN Send (TS: 0x%x)(Id: 0x%x, Pri: %d)(Data: ", CDC::getMillis( ), canId, msgPri );
293         for ( int i = 0; i < msg.dlc; i++ ) printf( " 0x%x", msgBuf[ i ] );
294         printf( ")\n" );
295     }
296
```

```
297     if ( can2040_transmit( &cBus, &msg ) != 0 ) {
298
299         CDC::sleepMillis( TX_RETRY_TIMEOUT );
300
301         if ( msgPri > MSG_PRI_VERY_HIGH ) return ( sendLcsMsg( msgBuf, msgPri - 1 ));
302         else                              return ( ERR_CAN_MSG_SEND );
303
304     } else return ( ALL_OK );
305 }
306
307 //-------------------------------------------------------------------------------------------------------------------
308 // "receiveLcsMsg" will check for a CAN Bus message and if one is available fill the passed message buffer.
309 // With the "can2040" library CAN bus messages are received via a callback function, which will store the
310 // each received message in the local receiver queue.
311 //
312 // Besides receiving a message, there is the handling of CAN Id collisions. When we detect a non-zero length
313 // message with a Can Id that is our own, we have a collision and report an error. This could happen for
314 // example when a node hardware is connected to another layout. Both nodes will then stop and wait for
315 // manual resolution.
316 //
317 // In addition to message processing, we also need to react to RTR messages. We answer such a request with
318 // sending a zero length message response. Replying to such a message from other nodes results in a status
319 // return of "ERR_CAN_MSG_NO_MSG" on this call as no LCS message was actually received. This is also the
320 // case when the message queue is empty.
321 //
322 //-------------------------------------------------------------------------------------------------------------------
323 uint8_t LcsMsgBusCAN::receiveLcsMsg( uint8_t *msgBuf ) {
324
325     can2040_msg msg;
326
327     if ( queue_try_remove( &rxQueue, &msg )) {
328
329         if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_CAN_BUS )) {
330
331             printf( "CAN Recv (TS: 0x%x)(Id: 0x%x, len: %d)(Data: ", CDC::getMillis( ), msg.id, msg.dlc );
332             for ( uint32_t i = 0; i < msg.dlc; i++ ) printf( " 0x%x", msg.data[ i ] );
333             printf( ")\n" );
334         }
335
336         bool      rtrFlag     = ( msg.id & 0x40000000 );
337         bool      extFlag     = ( msg.id & 0x80000000 );
338         uint16_t  remoteCanId = (( extFlag ) ? ( msg.id & 0x3FFF ) : ( msg.id & 0x7F ));
339
340         if (( remoteCanId == canId ) && ( msg.dlc > 0 )) {
341
342             return ( ERR_CAN_ID_COLLISION );
343         }
344         else if ( rtrFlag ) {
345
346             msg.id          = canId;
347             msg.dlc         = 0;
348             msg.data32[ 0 ] = 0;
349             msg.data32[ 1 ] = 0;
350
351             can2040_transmit( &cBus, &msg );
352             return ( ERR_CAN_MSG_NO_MSG );
353         }
354         else {
355
356             memcpy( msgBuf, msg.data, msg.dlc );
357             return ( ALL_OK );
358         }
359     }
360     else return ( ERR_CAN_MSG_NO_MSG );
361 }
362
363 }; // namespace LCS
```

```
1  //-----------------------------------------------------------------------------------------------------------
2  //
3  // LCS Runtime library - Non volatile storage based on the M24LCxxx chip family
4  //
5  //-----------------------------------------------------------------------------------------------------------
6  // This file implements the LCS runtime library non-volatile memory. The hardware is the AA24xxx chip family,
7  // which offers an I2C protocol based chip with various capacities. They all share the same pin layout and
8  // command structure.
9  //
10 // In addition we also support the M24C04 chip, which is used on the extension boards as a configuration
11 // storage. This chip will however be replaced by 24AA32, a 4K chip of the same chip family as the other
12 // chips on the controller board.
13 //
14 //-----------------------------------------------------------------------------------------------------------
15 //
16 // LCS Core library - Non volatile storage based on the M24LCxxx chip family
17 // Copyright (C) 2021 - 2024  Helmut Fieres
18 //
19 // This program is free software: you can redistribute it and/or modify it under the terms of the GNU General
20 // Public License as published by the Free Software Foundation, either version 3 of the License, or (at your
21 // option) any later version.
22 //
23 // This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
24 // implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
25 // for more details.
26 //
27 // You should have received a copy of the GNU General Public License along with this program. If not, see
28 // http://www.gnu.org/licenses
29 //
30 //  GNU General Public License:  http://opensource.org/licenses/GPL-3.0
31 //
32 //-----------------------------------------------------------------------------------------------------------
33
34 //-----------------------------------------------------------------------------------------------------------
35 // Include files.
36 //
37 //-----------------------------------------------------------------------------------------------------------
38 #include "LcsRuntimeLib.h"
39 #include "LcsRtLibInt.h"
40
41 //-----------------------------------------------------------------------------------------------------------
42 // Externals.
43 //
44 //-----------------------------------------------------------------------------------------------------------
45 namespace LCS {
46
47     extern uint16_t debugMask;
48 };
49
50 //-----------------------------------------------------------------------------------------------------------
51 // Local file declarations.
52 //
53 //-----------------------------------------------------------------------------------------------------------
54 namespace {
55
56 using namespace LCS;
57
58 //-----------------------------------------------------------------------------------------------------------
59 // Definitions for the M24LCxxx chips page size and total size. The chips have a pageSize which is the unit
60 // updated in case of a write. A write cannot across a page boundary and must be split into several writes
61 // if necessary. Reads do not have this problem. A1 chips have the same I2C address root which is "1010".
62 // There are three address lines A2, A1 and A0, which are used to select a chips. Up to eight chips can be
63 // addressed on a single I2C bus.
64 //
65 // The pageSizes on the chip are a multiple of 32bytes. For now, we use this size as the common denominator.
66 // Block handling and chipSize page handling are nicely taken care of this way. The downside is however that
67 // a write will update the chip page up to four times for a pageSize of 128. However, since the chips have
68 // more than a million write cycles and we rarely write large chunks of data, this will hopefully not be an
69 // issue in the near future.
70 //
71 // ??? the M24C04 is to be phased out ... we do not use that chip anymore...
72 //-----------------------------------------------------------------------------------------------------------
73 const uint16_t  BUFFER_BLOCK_SIZE           = 16; // ??? until we remove the M24C04 chip.
74
75 const uint16_t  M24LC32_PAGE_SIZE           = 32;
76 const uint32_t  M24LC32_MAX_SIZE            = 4096;
77
78 const uint16_t  M24LC64_PAGE_SIZE           = 32;
79 const uint32_t  M24LC64_MAX_SIZE            = 8192;
80
81 const uint16_t  M24LC128_PAGE_SIZE          = 64;
82 const uint32_t  M24LC128_MAX_SIZE           = 16384;
83
84 const uint16_t  M24LC256_PAGE_SIZE          = 64;
85 const uint32_t  M24LC256_MAX_SIZE           = 32768;
86
87 const uint16_t  M24LC512_PAGE_SIZE          = 128;
88 const uint32_t  M24LC512_MAX_SIZE           = 65536;
89
90 const uint16_t  M24C04_PAGE_SIZE            = 16;
91 const uint32_t  M24C04_MAX_SIZE             = 512;
92
93 const uint8_t   NVM_I2C_ADR_ROOT            = 0b1010000;
94 const uint8_t   EXT_I2C_ADR_ROOT            = 0b1010000;
95
96 const uint8_t   NVM_WRITE_DELAY             = 0x05;
97
98 //-----------------------------------------------------------------------------------------------------------
```

```
99  // Runtime NVM sizes. The runtime map has a maximum of 8Kb. The maximum size of a NVM chip is 64Kb.  The
100 // maximum size for an extension board NVM chip is 4Kb.
101 //
102 //--------------------------------------------------------------------------------------------------------
103 const uint32_t  NVM_RUNTIME_MAP_SIZE        = 0x2000;
104 const uint32_t  NVM_MAX_NVM_SIZE            = 0x10000;
105 const uint32_t  NVM_MAX_EXT_SIZE            = 0x1000;
106
107 //--------------------------------------------------------------------------------------------------------
108 // Module global data. A LCS node board has two NVM channels. The "NVM" channel refers to the NVM chip on
109 // main controller board. The "EXT" channel is the I2C bus that reaches out the the extension boards. On
110 // each extension board there is again a small NVM chip with configuration data. Besides the hardware pins
111 // there are the sizes of the chips.
112 //
113 // There is no easy way to determine the size of the actual chip. By convention, the extension board NVM
114 // chip has a fixed 4 Kbytes. The NVM chip on the main controller board is at least 8Kbytes, which is the
115 // architected runtime data structures size. The maximum size is 64 Kbytes. All the chips are from a hardware
116 // perspective identical. When we start a node, the nodeMap structure, i.e. the first few hundred bytes,
117 // contains a field that holds the actual size configured for the chip on the particular board. The difference
118 // between the runtime map size and the particular NVM chip maximum is considered "user NVM space" which the
119 // firmware can use as needed.
120 //
121 //--------------------------------------------------------------------------------------------------------
122 uint32_t    nodeNvmSize                     = 0;
123 uint32_t    extNvmSize                      = 0;
124
125 uint8_t     nvmSclPin                       = CDC::UNDEFINED_PIN;
126 uint8_t     nvmSdaPin                       = CDC::UNDEFINED_PIN;
127
128 uint8_t     extSclPin                       = CDC::UNDEFINED_PIN;
129 uint8_t     extSdaPin                       = CDC::UNDEFINED_PIN;
130
131 //--------------------------------------------------------------------------------------------------------
132 // "testNvmChipMemorySize" will check the NVM chip for its size. Since the chip itself has no way of telling
133 // its memory capacity, we need to go a rather cumbersome way. For each possible size, read the last byte,
134 // store a new value there, read it again. If the values match, it is a valid memory location. Don't forget
135 // to restore the previous value. If we are not successful, try the next smaller size. Currently, the LCS
136 // hardware uses The chip family M24LCxxx with sizes of 4, 8, 16, 32 and 64Kbytes.
137 //
138 // ??? not tested yet ...
139 //--------------------------------------------------------------------------------------------------------
140 uint32_t testNvmChipMemorySize( uint8_t sclPin, uint8_t i2cAdr ) {
141
142     uint32_t    nvmSize         = M24LC512_MAX_SIZE;
143     uint32_t    testAdr         = nvmSize - 1;
144     uint8_t     originalValue   = 0;
145     uint8_t     testValue       = 0xab;
146     uint8_t     tmpValue        = 0;
147     uint8_t     tmpBuf[ 3 ]     = { 0 };
148     uint8_t     rStat           = ALL_OK;
149
150     while ( nvmSize >= M24LC32_MAX_SIZE ) {
151
152         tmpBuf[ 0 ] = testAdr >> 8 & 0xFF;
153         tmpBuf[ 1 ] = testAdr &0xFF;
154
155         rStat = CDC::i2cWrite( sclPin, i2cAdr, tmpBuf, 2, true );
156         if ( rStat == ALL_OK ) rStat = CDC::i2cRead( sclPin, i2cAdr, &originalValue, 1 );
157         if ( rStat != ALL_OK ) return( ERR_NVM_CHIP_SIZE_DETECT );
158
159         tmpBuf[ 2 ] = testValue;
160
161         rStat = CDC::i2cWrite( sclPin, i2cAdr, tmpBuf, sizeof( tmpBuf ));
162         if ( rStat == ALL_OK ) {
163
164             CDC::sleepMillis( NVM_WRITE_DELAY );
165
166             rStat = CDC::i2cWrite( sclPin, i2cAdr, tmpBuf, 2, true );
167             if ( rStat == ALL_OK ) rStat = CDC::i2cRead( sclPin, i2cAdr, &tmpValue, 1 );
168             if ( rStat == ALL_OK ) {
169
170                 if ( tmpValue == testValue ) {
171
172                     rStat = CDC::i2cWrite( sclPin, i2cAdr, tmpBuf, sizeof( tmpBuf ));
173                     CDC::sleepMillis( NVM_WRITE_DELAY );
174                     return( nvmSize );
175                 }
176                 else {
177
178                     nvmSize = nvmSize / 2;
179                     testAdr = nvmSize - 1;
180                 }
181             }
182         }
183         else return( ERR_NVM_CHIP_SIZE_DETECT );
184     }
185
186     return( nvmSize );
187 }
188
189 //--------------------------------------------------------------------------------------------------------
190 // Each NVM chip has certain size. This function will round the size to the next lower chip memory size.
191 // We expect however that the programmer used the correct size, so this is done just in case. The lowest
192 // value is the 4Kb chip.
193 //
194 //--------------------------------------------------------------------------------------------------------
195 uint32_t roundNvmMaxSize( uint16_t chipSize ) {
196
197     if      ( chipSize <= M24C04_MAX_SIZE )   return ( M24C04_MAX_SIZE );
```

```
    else if ( chipSize <= M24LC32_MAX_SIZE )  return ( M24LC32_MAX_SIZE );
    else if ( chipSize <= M24LC64_MAX_SIZE )  return ( M24LC64_MAX_SIZE );
    else if ( chipSize <= M24LC128_MAX_SIZE ) return ( M24LC128_MAX_SIZE );
    else if ( chipSize <= M24LC256_MAX_SIZE ) return ( M24LC256_MAX_SIZE );
    else if ( chipSize <= M24LC512_MAX_SIZE ) return ( M24LC512_MAX_SIZE );
    else                                      return ( M24LC32_MAX_SIZE );
}

//-----------------------------------------------------------------------------------------------------------
// The nvmSize in buffer size blocks.
//
//-----------------------------------------------------------------------------------------------------------
uint16_t nvmSizeInBlocks( uint32_t nvmSize ) {

    return( nvmSize / BUFFER_BLOCK_SIZE );
}

//-----------------------------------------------------------------------------------------------------------
// "nvmGetBytesFromPage" transmits a set of data bytes only within the page boundary. Although a read can
// cross a page boundary, we follow the same principle as we do for writes when it comes to page boundaries.
// The read is send ing the address with retaining the bus. The PICO library will then use the restart
// condition. Just like we did in the write buffer counterpart, we need to send the address as one buffer.
//
//-----------------------------------------------------------------------------------------------------------
uint8_t nvmGetBytesFromPage( uint8_t sclPin, uint8_t i2cAdr, uint32_t ofs, uint8_t *buf, uint32_t len ) {

    uint8_t rStat = ALL_OK;

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_NVM_ACCESS )) {

        printf( "nvmGetBytesFromPage: sclPin: %d, i2cAdr: 0x%x, ofs: 0x%x, buf: %p, len: %d\n",
                sclPin, i2cAdr, ofs, buf, len );
    }

    uint32_t nvmSize = (( sclPin == nvmSclPin ) ? nodeNvmSize : extNvmSize );

    if ( nvmSize == M24C04_MAX_SIZE ) {

        uint8_t tmpAdr  = i2cAdr | (( ofs >> 8 ) & 0x01 );
        uint8_t tmpData = ofs & 0xFF;

        rStat = CDC::i2cWrite( sclPin, tmpAdr, &tmpData, sizeof( tmpData ), true );
        if ( rStat == ALL_OK ) rStat = CDC::i2cRead( sclPin, tmpAdr, buf, len );
    }
    else {

        uint8_t adr[ 2 ];

        adr[ 0 ] =  ( ofs  >> 8 ) & 0xFF;
        adr[ 1 ] =  ofs & 0xFF;

        rStat = CDC::i2cWrite( sclPin, i2cAdr, adr, 2, true );
        if ( rStat == ALL_OK ) rStat = CDC::i2cRead( sclPin, i2cAdr, buf, len );
    }

   if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_NVM_ACCESS )) {

        printf( "nvmGetBytesFromPage: %d\n", rStat );
    }

    return ( rStat );
}

//-----------------------------------------------------------------------------------------------------------
// "nvmPutBytesInPage" transmits a set of data bytes only within the page boundary. In general, a write
// cannot cross a chip internal page boundary. The Chip expects a write to be one sequence with the address
// bytes first followed by the data bytes with no stop or restart condition in between. This costed my
// quite some debugging to figure this out. We will have a local buffer where we combine the address and
// data and then send it.
//
//-----------------------------------------------------------------------------------------------------------
uint8_t nvmPutBytesInPage( uint8_t sclPin, uint8_t i2cAdr, uint32_t ofs, uint8_t *buf, uint32_t len ) {

    uint8_t rStat = ALL_OK;
    uint8_t dataBuf[ BUFFER_BLOCK_SIZE + 2 ];

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_NVM_ACCESS )) {

        printf( "nvmPutBytesInPage: sclPin: %d, i2cAdr: 0x%x, ofs: 0x%x, bufAdr: %p, len: %d\n",
        sclPin, i2cAdr, ofs, buf, len );
    }

    uint32_t nvmSize = (( sclPin == nvmSclPin ) ? nodeNvmSize : extNvmSize );

    if ( nvmSize == M24C04_MAX_SIZE ) {

        dataBuf[ 0 ] = ofs & 0xFF;
        for ( int i = 0; i < len; i++ ) dataBuf[ i + 1 ] = buf[ i ];

        uint8_t tmpAdr = i2cAdr | (( ofs >> 8 ) & 0x01 );
        rStat = CDC::i2cWrite( sclPin, tmpAdr, dataBuf, len + 1 );
    }
    else {

        dataBuf[ 0 ]    = ( ofs  >> 8 ) & 0xFF;
        dataBuf[ 1 ]    = ofs & 0xFF;

        for ( int i = 0; i < len; i++ ) dataBuf[ i + 2 ] = buf[ i ];

```

```
297         rStat = CDC::i2cWrite( sclPin, i2cAdr, dataBuf, len + 2 );
298     }
299
300     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_NVM_ACCESS )) {
301
302         printf( "nvmPutBytesInPage: ret: %d\n", rStat );
303     }
304
305     return ( rStat );
306 }
307
308 //-----------------------------------------------------------------------------------------------------------
309 // "nvmGetBytes" reads a set of data bytes from the memory. Although read operations do not have a page
310 // boundary issue, we stick to the concept to read within page boundaries as we may one day use more than
311 // chip to build NVMs and then we have no problems with crossing chip boundaries.
312 //
313 //-----------------------------------------------------------------------------------------------------------
314 uint8_t nvmGetBytes( uint8_t sclPin, uint8_t i2cAdr, uint32_t ofs, uint8_t *buf, uint32_t len ) {
315
316     uint8_t rStat = ALL_OK;
317
318     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_NVM_ACCESS )) {
319
320         printf( "nvmGetBytes: scl: %d, i2c: 0x%x, ofs: 0x%x, bufAdr: %p, len: %d\n",
321                 sclPin, i2cAdr, ofs, (uint32_t) buf, len );
322     }
323
324     uint32_t nvmSize = (( sclPin == nvmSclPin ) ? nodeNvmSize : extNvmSize );
325     if ( ofs + len > nvmSize ) return ( ERR_NVM_SIZE_EXCEEDED );
326
327     uint32_t  bytesLeft     = len;
328     uint32_t  pageBytesLeft = BUFFER_BLOCK_SIZE - ofs % BUFFER_BLOCK_SIZE;
329
330     while ( bytesLeft > pageBytesLeft ) {
331
332         rStat = nvmGetBytesFromPage( sclPin, i2cAdr, ofs + len - bytesLeft, buf + len - bytesLeft, pageBytesLeft );
333         if ( rStat != ALL_OK ) break;
334
335         bytesLeft       -= pageBytesLeft;
336         pageBytesLeft   = BUFFER_BLOCK_SIZE;
337     }
338
339     if (( rStat == ALL_OK ) && ( bytesLeft > 0 )) {
340
341         rStat = nvmGetBytesFromPage( sclPin, i2cAdr, ofs + len - bytesLeft, buf + len - bytesLeft, bytesLeft );
342     }
343
344     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_NVM_ACCESS )) {
345
346         printf( "nvmGetBytes: ret: %d\n", rStat );
347     }
348
349     return( rStat );
350 }
351
352 //-----------------------------------------------------------------------------------------------------------
353 // "nvmPutBytes" transmits a set of data bytes to the memory. We cannot write across the internal NVM page
354 // boundary and also across a chip boundary. This routine will split the data to write only within one page
355 // in a given write cycle.
356 //
357 // There is a quirk with figuring out that a chip is ready for the next write instruction. The data sheet
358 // suggest a writing of one byte to see of the chip acknowledges. If not it is still in a write operation.
359 // This approach does not seem to work with the PICO i2c libraries. So, we will go the "slow" way of giving
360 // the chip the time to complete the write cycle before issuing another one. Since we do not often write
361 // to the NVM, the slow mode is perhaps acceptable for now.
362 //
363 //-----------------------------------------------------------------------------------------------------------
364 uint8_t nvmPutBytes( uint8_t sclPin, uint8_t i2cAdr, uint32_t ofs, uint8_t *buf, uint32_t len ) {
365
366     uint8_t rStat = ALL_OK;
367
368     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_NVM_ACCESS )) {
369
370         printf( "nvmPutBytes: scl: %d, i2c: 0x%x, ofs: 0x%x, buf: %p, len: %d\n", sclPin, i2cAdr, ofs, buf, len );
371      }
372
373     uint32_t nvmSize = (( sclPin == nvmSclPin ) ? nodeNvmSize : extNvmSize );
374     if ( ofs + len > nvmSize ) return ( ERR_NVM_SIZE_EXCEEDED );
375
376     uint32_t  bytesLeft     = len;
377     uint32_t  pageBytesLeft = BUFFER_BLOCK_SIZE - ofs % BUFFER_BLOCK_SIZE;
378
379     while ( bytesLeft > pageBytesLeft ) {
380
381         rStat = nvmPutBytesInPage( sclPin, i2cAdr, ofs + len - bytesLeft, buf + len - bytesLeft, pageBytesLeft );
382         if ( rStat != ALL_OK ) break;
383
384         bytesLeft       -= pageBytesLeft;
385         pageBytesLeft   = BUFFER_BLOCK_SIZE;
386
387         CDC::sleepMillis( NVM_WRITE_DELAY );
388     }
389
390     if (( rStat == ALL_OK ) && ( bytesLeft > 0 )) {
391
392        rStat = nvmPutBytesInPage( sclPin, i2cAdr, ofs + len - bytesLeft, buf + len - bytesLeft, bytesLeft );
393        CDC::sleepMillis( NVM_WRITE_DELAY );
394     }
395
```

```
396     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_NVM_ACCESS )) {
397
398         printf( "nvmPutBytes: ret: %d\n", rStat );
399     }
400
401     return( rStat );
402 }
403
404 //-----------------------------------------------------------------------------------------------------------
405 // "nvmClearArea" wipes out an area of the NVM chip. To speed up the writing, we fill a local buffer with
406 // the value and then write blocks at a time.
407 //
408 //-----------------------------------------------------------------------------------------------------------
409 uint8_t nvmClearArea( uint8_t sclPin, uint8_t i2cAdr, uint32_t ofs, uint32_t len, uint8_t val ) {
410
411     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_NVM_ACCESS )) {
412
413         printf( "nvmClearArea: scl: %d, i2c: 0x%x, ofs: 0x%x, len: %d, val: %d\n", sclPin, i2cAdr, ofs, len, val );
414     }
415
416     uint8_t     tmpBuf[ BUFFER_BLOCK_SIZE ];
417     uint8_t     rStat   = ALL_OK;
418     uint32_t    nvmSize = (( sclPin == nvmSclPin ) ? nodeNvmSize : extNvmSize );
419     uint32_t    limit   = ofs + len;
420
421     if ( ofs + len > nvmSize ) return ( ERR_NVM_SIZE_EXCEEDED );
422
423     for ( int i = 0; i < BUFFER_BLOCK_SIZE; i ++ ) tmpBuf[ i ] = val;
424
425     while ( len > BUFFER_BLOCK_SIZE ) {
426
427         rStat = nvmPutBytes( sclPin, i2cAdr, ofs, tmpBuf, sizeof( tmpBuf ));
428         if ( rStat != ALL_OK ) break;
429
430         ofs += BUFFER_BLOCK_SIZE;
431         len -= BUFFER_BLOCK_SIZE;
432     }
433
434     if (( rStat == ALL_OK ) && ( len > 0 )) {
435
436         rStat = nvmPutBytes( sclPin, i2cAdr, ofs, tmpBuf, len );
437     }
438
439     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_NVM_ACCESS )) {
440
441         printf( "nvmClearArea: ret: %d\n", rStat );
442     }
443
444     return( rStat );
445 }
446
447 }; // namespace
448
449
450 //-----------------------------------------------------------------------------------------------------------
451 // The LCS name space routines declared in this file.
452 //
453 //-----------------------------------------------------------------------------------------------------------
454 namespace LCS {
455
456 //-----------------------------------------------------------------------------------------------------------
457 // "configNvm" will setup the module local variables. We copy the I2C hardware pins and the NVM related data
458 // from the CDC descriptors. The CDC descriptor also contains the configured sizes for the NVM chips.
459 //
460 //-----------------------------------------------------------------------------------------------------------
461 uint8_t configNvm( CDC::CdcConfigDesc *ci ) {
462
463     nvmSclPin     = ci -> NVM_I2C_SCL_PIN;
464     nvmSdaPin     = ci -> NVM_I2C_SDA_PIN;
465     extSclPin     = ci -> EXT_I2C_SCL_PIN;
466     extSdaPin     = ci -> EXT_I2C_SDA_PIN;
467     nodeNvmSize   = ci -> NODE_NVM_SIZE;
468     extNvmSize    = ci -> EXT_NVM_SIZE;
469
470     if ( nodeNvmSize > NVM_MAX_NVM_SIZE )   nodeNvmSize = NVM_MAX_NVM_SIZE;
471     if ( extNvmSize > NVM_MAX_EXT_SIZE )    extNvmSize  = NVM_MAX_EXT_SIZE;
472
473     return( ALL_OK );
474 }
475
476 //-----------------------------------------------------------------------------------------------------------
477 // Controller Board Runtime Map access routines. The runtime map occupies the first 8 Kbytes of the main
478 // controller NVM chip. There are routines for getting and setting a word as well as routines to read and
479 // write a buffer. All access routines are prefixed with "rt".
480 //
481 //-----------------------------------------------------------------------------------------------------------
482 uint8_t rtNvmPutWord( uint32_t ofs, uint16_t word ) {
483
484     return( nvmPutBytes( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, (uint8_t *) &word, sizeof( uint16_t )));
485 }
486
487 uint8_t rtNvmGetWord( uint32_t ofs, uint16_t *word ) {
488
489     return( nvmGetBytes( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, (uint8_t *) word, sizeof( uint16_t )));
490 }
491
492 uint8_t rtNvmPutBytes( uint32_t ofs, uint8_t *buf, uint32_t len ) {
493
494     return( nvmPutBytes( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, buf, len ));
```

```
495 }
496
497 uint8_t rtNvmGetBytes( uint32_t ofs, uint8_t *buf, uint32_t len ) {
498
499     return( nvmGetBytes( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, buf, len ));
500 }
501
502 uint8_t rtNvmClearArea( uint32_t ofs, uint32_t len, uint8_t val ) {
503
504     return( nvmClearArea( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, len, val ));
505 }
506
507 uint32_t rtNvmGetSize( ) {
508
509     return( NVM_RUNTIME_MAP_SIZE );
510 }
511
512 //-----------------------------------------------------------------------------------------------------------
513 // Extension Board Map access routines. These routines access the NVM on the extension board. The I2C address
514 // is formed by the chip common I2C address plus the address bits of the chip to select the chip on the
515 // particular extension board. Similar to the runtime NVM access routines, there are routines for getting
516 // and setting a word as well as routines to read and  write a buffer. All access routines are prefixed with
517 // "ext".
518 //
519 //-----------------------------------------------------------------------------------------------------------
520 uint8_t extNvmPutWord( uint8_t boardId, uint32_t ofs, uint16_t word ) {
521
522     uint8_t i2cAdr = EXT_I2C_ADR_ROOT + (( boardId % MAX_EXT_BOARD_MAP_ENTRIES ) << 1 );
523     return( nvmPutBytes( extSclPin, i2cAdr, ofs, (uint8_t *) &word, sizeof( uint16_t )));
524 }
525
526 uint8_t extNvmGetWord( uint8_t boardId, uint32_t ofs, uint16_t *word ) {
527
528     uint8_t i2cAdr = EXT_I2C_ADR_ROOT + (( boardId % MAX_EXT_BOARD_MAP_ENTRIES ) << 1 );
529     return( nvmGetBytes( extSclPin, i2cAdr, ofs, (uint8_t *) word, sizeof( uint16_t )));
530 }
531
532 uint8_t extNvmPutBytes( uint8_t boardId, uint32_t ofs, uint8_t *buf, uint32_t len ) {
533
534     uint8_t i2cAdr = EXT_I2C_ADR_ROOT + (( boardId % MAX_EXT_BOARD_MAP_ENTRIES ) << 1 );
535     return( nvmPutBytes( extSclPin, i2cAdr, ofs, buf, len ));
536 }
537
538 uint8_t extNvmGetBytes( uint8_t boardId, uint32_t ofs, uint8_t *buf, uint32_t len ) {
539
540     uint8_t i2cAdr = EXT_I2C_ADR_ROOT + (( boardId % MAX_EXT_BOARD_MAP_ENTRIES ) << 1 );
541     return( nvmGetBytes( extSclPin, i2cAdr, ofs, buf, len ));
542 }
543
544 uint8_t extNvmClearArea( uint8_t boardId, uint32_t ofs, uint32_t len, uint8_t val ) {
545
546     uint8_t i2cAdr = EXT_I2C_ADR_ROOT + (( boardId % MAX_EXT_BOARD_MAP_ENTRIES ) << 1 );
547     return( nvmClearArea( extSclPin, i2cAdr, ofs, len, val ));
548 }
549
550 uint32_t extNvmGetSize( ) {
551
552     return( extNvmSize );
553 }
554
555 //-----------------------------------------------------------------------------------------------------------
556 // Controller Board User Map access routines. The area between the main controller NVM chip runtime area and
557 // the chips hardware maximum size is the memory area available for the firmware programmer. Again, there
558 // are routines for getting and setting a word as well as routines to read and  write a buffer. All access
559 // routines are prefixed with "nvm".
560 //
561 //-----------------------------------------------------------------------------------------------------------
562 uint8_t usrNvmPutWord( uint32_t ofs, uint16_t word ) {
563
564     ofs = ofs + NVM_USER_MAP_START;
565     return( nvmPutBytes( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, (uint8_t *) &word, sizeof( uint16_t )));
566 }
567
568 uint8_t usrNvmGetWord( uint32_t ofs, uint16_t *word ) {
569
570     ofs = ofs + NVM_USER_MAP_START;
571     return( nvmGetBytes( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, (uint8_t *) word, sizeof( uint16_t )));
572 }
573
574 uint8_t usrNvmPutBytes( uint32_t ofs, uint8_t *buf, uint32_t len ) {
575
576     ofs = ofs + NVM_USER_MAP_START;
577     return( nvmPutBytes( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, buf, len ));
578 }
579
580 uint8_t usrNvmGetBytes( uint32_t ofs, uint8_t *buf, uint32_t len ) {
581
582     ofs = ofs + NVM_USER_MAP_START;
583     return( nvmGetBytes( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, buf, len ));
584 }
585
586 uint32_t usrNvmGetSize( ) {
587
588     return ( nodeNvmSize - NVM_USER_MAP_START );
589 }
590
591 }; // namespace LCS
```

```
//-----------------------------------------------------------------------------------------------------------
//
// Layout Control System - Runtime setup file.
//
//-----------------------------------------------------------------------------------------------------------
// The file implements a part of the LcsRuntimeLib that deals with the setup and start sequence of a node.
// There is a lot to do. First, we need to initialize the CDC layer, our lower layer foundation. Next the
// NVM of the nodeMap is located and checked for validity. The nodeMap contains all the information for
// setting up the entire node. If this steps fails, we either need to configure the nodeMap, or we have a
// data error and the node is not usable.
//
// With a correct node map in place, the memory structures for the node, the ports, events, callbacks and
// periodic tasks are created. The node is basically ready to do work. For a node that has no extension
// boards connected, we are done.
//
// Next is the extension board setup. We try to locate all connected extension boards and install the
// corresponding driver. A driver is just a procedure that knows how to talk to the particular extension
// board. A failure in this part of the sequence sequence does not necessarily mean that the node cannot be
// used.
//
// Assuming all went fine, the runtime library is ready to accept calls for registering callbacks and a few
// other library calls. Once all this work is done, the last call of the node firmware would be to start
// the runtime, which would as the very first thing invoke all registered initialization callbacks and the
// enter the processing loop. We will not return from that routine.
//
// An error in the setup sequence does not necessarily mean that the node is unusable. For example, when
// the nodeMap is not valid, the setup routine will report an error, but we can still call the runtime
// loop. The runtime loop will handle LCS messages and also provide the console IO, which in turn allows us
// manually correct the node data for a successful restart. In a similar way, extension board errors can be
// be addressed.
//
// This file contains the library global data declarations if the LCS runtime library. All other files will
// refer to them as "extern".
//
//-----------------------------------------------------------------------------------------------------------
//
// LCS - Core Library
// Copyright (C) 2021 - 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 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/>.
//
//-----------------------------------------------------------------------------------------------------------
#include "LcsRuntimeLib.h"
#include "LcsRtLibInt.h"
#include "LcsDrvOccDetectLib.h"
#include "LcsDrvServoLib.h"

//-----------------------------------------------------------------------------------------------------------
// Runtime globals. This file contains the global data structure declarations. They are declared in the LCS
// name space. All other files in the runtime library will declare them as "extern" if needed.

// There is also the debug mask. The idea is to have a debug mask where each major part of the library has a
// bit. There could also be bits reserved for the firmware. Then we have control items to set these bits.
// Wherever debugging or tracing is needed, the bit mask will be used to determine whether to print debugging
// data or not. From a performance perspective, the test will take just a couple of ARM instructions. In other
// words we do not take out debugging code when going into production. Never liked this approach of conditional
// debug code via "ifdefs".
//
//-----------------------------------------------------------------------------------------------------------
namespace LCS {

    uint16_t                debugMask    = 0;
    uint16_t                startOptions = 0;

    LcsCdcMap               cdcMap;
    LcsMsgBusCAN            *msgBus;
    LcsNodeData             nodeData;
    LcsNodeMap              nodeMap;
    LcsPortMap              portMap;
    LcsEventMap             eventMap;
    LcsCallbackMap          callbackMap;
    LcsPendingReqMap        pendingReqMap;
    LcsTaskMap              taskMap;
    LcsDrvFuncMap           drvFuncMap;
    LcsDrvMap               drvMap;
}

//-----------------------------------------------------------------------------------------------------------
// The LcsCoreLibConfig implementation file local declarations and routines.
//
//-----------------------------------------------------------------------------------------------------------
namespace {

using namespace LCS;

//-----------------------------------------------------------------------------------------------------------
// Utility routines and constants.
//
//-----------------------------------------------------------------------------------------------------------
const char  *nodeDefName  = "Node Name";

```

```
99  uint16_t roundup( uint16_t elements, uint16_t alignSize ) {
100
101     return ((( elements + alignSize - 1 ) / alignSize ) * alignSize );
102 }
103
104 bool isInRangeU( uint16_t val, uint16_t lower, uint16_t upper ) {
105
106     return (( val >= lower ) && ( val <= upper ));
107 }
108
109 uint16_t buildNpId( uint16_t nodeId, uint16_t portId ) {
110
111     return(( nodeId << 4 ) | ( portId & 0xF ));
112 }
113
114 uint16_t nodeId( uint16_t npId ) {
115
116     return( npId >> 4 );
117 }
118
119 uint16_t portId( uint16_t npId ) {
120
121     return( npId & 0xF );
122 }
123
124 //-------------------------------------------------------------------------------------------------------------
125 // "buildDefaultNodeMap" build a nodeMap with the default values from the declaration structure. The default
126 // nodeMap is used for initializing the memory runtime node map for formatting a new or corrupted runtime NVM.
127 //
128 //-------------------------------------------------------------------------------------------------------------
129 void buildDefaultNodeMap( LcsNodeMap *nMap ) {
130
131     LcsNodeMap tmp;
132
133     snprintf( tmp.name, MAX_NODE_NAME_SIZE, "Node" );
134     tmp.nodeUID = CDC::createUid( );
135
136     *nMap = tmp;
137 }
138
139 //-------------------------------------------------------------------------------------------------------------
140 // "buildDefaultPortMap" will initialize the portMap data structure on MEM or NVM, It is just an array of
141 // portMap entries. Each port will get a default name.
142 //
143 //-------------------------------------------------------------------------------------------------------------
144 void buildDefaultPortMap( LcsPortMap *pMap ) {
145
146     for ( uint16_t i = 0; i < MAX_PORT_MAP_ENTRIES; i++ ) {
147
148         LcsPortMapEntry pEntry;
149
150         snprintf( pEntry.name, MAX_PORT_NAME_SIZE, "Port-%d", i + 1 );
151         pMap -> map[ i ] = pEntry;
152     }
153 }
154
155 //-------------------------------------------------------------------------------------------------------------
156 // "buildDefaultEventMap" initializes the event map on MEM or NVM. It is just an array of eventMap entries.
157 //
158 //-------------------------------------------------------------------------------------------------------------
159 void buildDefaultEventMap( LcsEventMap *eMap ) {
160
161     LcsEventMapEntry e;
162     for ( uint16_t i = 0; i < MAX_EVENT_MAP_ENTRIES; i++ ) eMap -> map[ i ] = e;
163 }
164
165 //-------------------------------------------------------------------------------------------------------------
166 // "buildDefaultNodeData" initializes the node data map on MEM or NVM. It is just an array of variables. We
167 // clear them out.
168 //
169 //-------------------------------------------------------------------------------------------------------------
170 void buildDefaultNodeData( LcsNodeData *nData ) {
171
172     memset( nData -> map, 0, MAX_NODE_DATA_BLOCKS * MAX_ATTR_MAP_ENTRIES * sizeof( uint16_t));
173 }
174
175 //-------------------------------------------------------------------------------------------------------------
176 // "buildNvmRuntimeStructures" initializes a new or corrupt runtime NVM with default data. After successful
177 // completion, we will have a valid runtime map.
178 //
179 // ??? CDC map business ???
180 //-------------------------------------------------------------------------------------------------------------
181 uint8_t buildNvmRuntimeStructures( ) {
182
183     uint8_t rStat;
184
185     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "buildNvmRuntimeStructures\n" );
186
187     rtNvmClearArea( NVM_NODE_MAP_START, NVM_RUNTIME_AREA_SIZE );
188
189     // ??? build default CDC map ?
190
191     buildDefaultNodeMap( &nodeMap );
192     buildDefaultPortMap( &portMap );
193     buildDefaultEventMap( &eventMap );
194     buildDefaultNodeData( &nodeData );
195
196     rStat = rtNvmPutBytes( NVM_NODE_MAP_START, (uint8_t *) &nodeMap, sizeof( LcsNodeMap ));
197     if ( rStat == ALL_OK ) rtNvmPutBytes( NVM_PORT_MAP_START, (uint8_t *) &portMap, sizeof( LcsPortMap ));
```

```
198     if ( rStat == ALL_OK ) rtNvmPutBytes( NVM_EVENT_MAP_START, (uint8_t *) &eventMap, sizeof( LcsEventMap ));
199     if ( rStat == ALL_OK ) rtNvmPutBytes( NVM_NODE_DATA_START, (uint8_t *) &nodeData, sizeof( LcsNodeData ));
200
201     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
202         printf( "buildNvmRuntimeStructures, stat: %d\n", rStat  );
203
204     return( rStat );
205 }
206
207 //-----------------------------------------------------------------------------------------------------------
208 // "buildDefaultBoardDesc" initializes the extension board structure. An extension board has a header
209 // structure identical to the nodeMap structure. In addition, there is the driver data area.
210 //
211 //-----------------------------------------------------------------------------------------------------------
212 void buildDefaultBoardDesc( LcsDrvBoardDesc *bDesc ) {
213
214     LcsNvmHeader tmp;
215
216     bDesc -> head = tmp;
217     memset( bDesc -> driverData, 0, MAX_DRV_DATA_SIZE * sizeof( uint16_t ));
218 }
219
220 //-----------------------------------------------------------------------------------------------------------
221 // "buildDefaultDrvMap" initializes the driver map memory structure.
222 //
223 //-----------------------------------------------------------------------------------------------------------
224 void buildDefaultDrvMap( LcsDrvMap *drv ) {
225
226     LcsDrvBoardDesc initDesc;
227
228     for ( uint16_t i = 0; i < MAX_EXT_BOARD_MAP_ENTRIES; i ++ ) {
229
230         drvMap.map[ i ].flags       = 0;
231         drvMap.map[ i ].drvFunc     = nullptr;
232         drvMap.map[ i ].extBoard    = initDesc;
233     }
234 }
235
236 //-----------------------------------------------------------------------------------------------------------
237 // "buildNvmRuntimeStructures" initializes a new or corrupt runtime NVM with default data. However, the write
238 // will only succeed when we have the board write enabled. Otherwise an error is returned.
239 //
240 //-----------------------------------------------------------------------------------------------------------
241 uint8_t buildNvmExtBoardStructure( uint8_t boardId ) {
242
243     uint8_t     rStat;
244     LcsDrvEntry entry;
245
246     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) {
247
248         printf( "buildNvmExtBoardStructure for board: %d\n", boardId );
249     }
250
251     rStat = extNvmClearArea( boardId, 0, sizeof( LcsDrvEntry ));
252     if ( rStat == ALL_OK ) extNvmPutBytes( boardId, 0, (uint8_t *) &entry, sizeof( entry ));
253
254     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) {
255
256         printf( "buildNvmExtBoardStructure, stat: %d\n", rStat  );
257     }
258
259     return( rStat ); // ??? own error constant ?
260 }
261
262 }; // namespace
263
264
265 //-----------------------------------------------------------------------------------------------------------
266 // The LCS name space routines declared in this file.
267 //
268 //-----------------------------------------------------------------------------------------------------------
269 namespace LCS {
270
271 //-----------------------------------------------------------------------------------------------------------
272 // When the node is powered on, the very first thing to do is to setup the CDC library and setup the "active"
273 // and "ready" LED pins used by the board. The pins need to to be configured. We also make a call to to
274 // initialize the CDC. Note that this may have been done before, when for example the firmware programmer
275 // wants to use the HW before calling any library setup code. It is no problem to call the CDC init routine
276 // several times.
277 //
278 // There are two basic modes. The first is when we have a console connected. We will prompt and wait for a
279 // start command. There are several options for starting a node. The easiest is "R" which just starts the
280 // node. The "D" command will start with debugging enabled. We will set the setup debug flags to check any
281 // issues during the startup phase. Finally, there is there "F" command, which will format the NVM runtime
282 // area. However, all that is happening in this routine is to set these options to be executed at the right
283 // place in the setup sequence.
284 //
285 // The second mode is when there no console connected. In this case, Debug is disabled and we just setup
286 // the node. This mode should be the normal case for all the nodes in a layout.
287 //
288 // Perhaps one day, this routine could be enhanced to allow commands to pile up the start options followed
289 // by the final start command to get the show going. especially the debug mask would be a candidate.
290 //
291 //-----------------------------------------------------------------------------------------------------------
292 uint8_t initCdcLayer( CDC::CdcConfigDesc *ci ) {
293
294     const uint32_t CONSOLE_TIMEOUT = 1024 * 1024 * 4;
295
296     cdcMap.cfg = *ci;
```

```
297
298     CDC::init( ci );
299
300     if ( ci -> READY_LED_PIN != CDC::UNDEFINED_PIN ) CDC::configureDio( ci -> READY_LED_PIN, CDC::OUT );
301     if ( ci -> ACTIVE_LED_PIN != CDC::UNDEFINED_PIN ) CDC::configureDio( ci -> ACTIVE_LED_PIN, CDC::OUT );
302
303     if ( CDC::isConsoleConnected( )) {
304
305         CDC::writeDio( ci -> READY_LED_PIN, true );
306
307         while ( true ) {
308
309             printf( "=>" );
310
311             char ch = CDC::getConsoleChar( CONSOLE_TIMEOUT );
312
313             if (( ch == 'R' ) || ( ch == 'r' )) {
314
315                 printf( "Starting - normal mode\n" );
316
317                 debugMask       &= ~ DBG_CONFIG;
318                 startOptions    = NOPT_NIL;
319                 return( ALL_OK );
320             }
321             else if (( ch == 'D' ) || ( ch == 'd' )) {
322
323                  printf( "Starting - debug mode\n" );
324
325                 debugMask       = DBG_CONFIG | DBG_SETUP | DBG_EVENTS;
326                 startOptions    = NOPT_NIL;
327                 return( ALL_OK );
328             }
329             else if (( ch == 'F' ) || ( ch == 'f' )) {
330
331                 printf( "Starting - format mode\n" );
332
333                 debugMask       &= ~ DBG_CONFIG;
334
335                 debugMask       = DBG_CONFIG | DBG_SETUP | DBG_NVM_ACCESS;
336
337                 startOptions    = NOPT_FORMAT_RUNTIME;
338                 return( ALL_OK );
339             }
340             else if ( ch == '?' ) {
341
342                 printf( "Setup options:\n" );
343                 printf( "r, R -> start the node with debug initially disabled\n" );
344                 printf( "d, D -> start the node with \"setup\" debug options enabled\n" );
345                 printf( "f, F -> start the node with a newly formatted runtime map\n" );
346             }
347             else printf( "\n" );
348         }
349     }
350     else {
351
352         debugMask       &= ~ DBG_CONFIG;
353         startOptions    = NOPT_NIL;
354         return ( ALL_OK );
355     }
356 }
357
358 //-------------------------------------------------------------------------------------------------------------
359 // The NVM library functions will work after this routine. We first set up the I2C channels, which are the
360 // heart of any internal board communication. After the I2C channels are initialized, we will configure the
361 // NVM library. If all is OK, we can talk to all NVMs on the boards making up the node.
362 //
363 // ??? should we assume a "architectural" setting of the IC2 channels and not rely on CDC map ?
364 //-------------------------------------------------------------------------------------------------------------
365 uint8_t initNvmChannels( CDC::CdcConfigDesc *ci ) {
366
367    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) {
368
369         printf( "initNvmChannels: nvmSCL: %d, nvmSDA: %d, extSCL: %d, extSDA: %d\n",
370                 ci -> NVM_I2C_SCL_PIN, ci -> NVM_I2C_SDA_PIN, ci -> EXT_I2C_SCL_PIN, ci -> EXT_I2C_SDA_PIN );
371     }
372
373     uint8_t rStat;
374
375     if (( ci -> NVM_I2C_SCL_PIN != CDC::UNDEFINED_PIN ) && ( ci -> NVM_I2C_SDA_PIN != CDC::UNDEFINED_PIN )) {
376
377         rStat = CDC::configureI2C( ci -> NVM_I2C_SCL_PIN , ci -> NVM_I2C_SDA_PIN );
378         if ( rStat != ALL_OK ) return( rStat );
379     }
380
381     if (( ci -> EXT_I2C_SCL_PIN != CDC::UNDEFINED_PIN ) && ( ci -> EXT_I2C_SDA_PIN != CDC::UNDEFINED_PIN )) {
382
383         rStat = CDC::configureI2C( ci -> EXT_I2C_SCL_PIN , ci -> EXT_I2C_SDA_PIN );
384         if ( rStat != ALL_OK ) return( rStat );
385     }
386
387     rStat = configNvm( ci );
388
389     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
390         printf( "initNvmChannels, status: %d\n", rStat );
391
392     return ( rStat );
393 }
394
395 //-------------------------------------------------------------------------------------------------------------
```

```
396 // Next is CAN bus setup. The message bus is the central communication mechanism. If we can also get it up
397 // early we could use it not only for configurations and operations but perhaps for remote troubleshooting.
398 //
399 // ??? should we assume a "architectural" setting of the CAN channel and not rely on CDC map ?
400 //-----------------------------------------------------------------------------------------------------------
401 uint8_t initCanBus( CDC::CdcConfigDesc *ci ) {
402 
403     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "initCanBus\n" );
404 
405     msgBus = new LcsMsgBusCAN( );
406 
407     uint8_t rStat = msgBus -> init( 0, ci -> CAN_BUS_RX_PIN, ci -> CAN_BUS_TX_PIN, ci -> CAN_BUS_CTRL_MODE );
408     if ( rStat != ALL_OK ) {
409 
410         if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
411             printf( "initCanBus, CAN status: %d\n", rStat );
412 
413         rStat = ERR_CAN_SETUP;
414     }
415 
416     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
417         printf( "initCanBus, status: %d\n", rStat );
418 
419     return ( rStat );
420 }
421 
422 //-----------------------------------------------------------------------------------------------------------
423 // "setupNodeMap" sets up the nodeMap. It is the first routine after all the basic hardware settings is in
424 // place. Unless the start options tell us to just format a new runtime area, we read in the nodeMap from
425 // the node NVM. A quick check of the magic words and the the nodeMap size field will tell us whether this
426 // nodeMap was initialized before. If this is not the case, we must assume a corrupt nodeMap or a new board.
427 // The runtime area data structures are created with default values and written to the NVM.
428 //
429 // In any case, the follow-on setup routines can assume a valid data structure to work from and just read
430 // the NVM as normal. If this routine has an error it should be considered as a fatal error.
431 //
432 // ??? check the sizes of the maps, load CDC map if we only have a board type ?
433 //-----------------------------------------------------------------------------------------------------------
434 uint8_t setupNodeMap( LcsConfigDesc *cfg ) {
435 
436     uint8_t rStat = ALL_OK;
437 
438     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "setupNodeMap\n" );
439 
440     if ( startOptions & NOPT_FORMAT_RUNTIME ) {
441 
442         rStat = buildNvmRuntimeStructures( );
443     }
444 
445     rStat = rtNvmGetBytes( NVM_NODE_MAP_START, (uint8_t *) &nodeMap, sizeof( LcsNodeMap ));
446     if ( rStat != ALL_OK ) return( rStat );
447 
448      if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) {
449 
450         uint16_t *ptr = (uint16_t *) &nodeMap;
451 
452         printf( "NodeMap Head: " );
453         for ( int i = 0; i < 16; i++ ) printf( "0x%x ", ptr[ i ] );
454         printf( "\n" );
455      }
456 
457     if (( nodeMap.head.magicWord1 != NVM_MWORD_1 ) ||
458         ( nodeMap.head.magicWord2 != NVM_MWORD_2 ) ||
459         ( nodeMap.nodeMapSize != sizeof( LcsNodeMap ))) {
460 
461             // ??? check the sizes of the maps ???
462 
463         if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
464             printf( "setupNodeMap: invalid header, re-format\n" );
465 
466         rStat = buildNvmRuntimeStructures( );
467     }
468 
469     nodeMap.nodeOptions = cfg -> options;
470 
471     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
472         printf( "setupNodeMap, status: %d\n", rStat );
473 
474     return ( rStat );
475 }
476 
477 //-----------------------------------------------------------------------------------------------------------
478 // "setupPortMap" will read the port data the NVM port map data area into the memory counterpart.
479 //
480 //-----------------------------------------------------------------------------------------------------------
481 uint8_t setupPortMap( ) {
482 
483     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "setupPortMap\n" );
484 
485     uint8_t rStat = rtNvmGetBytes( NVM_PORT_MAP_START, (uint8_t *) &portMap, sizeof( LcsPortMap ));
486 
487     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
488         printf( "setupPortMap, status: %d\n", rStat );
489 
490     return ( rStat );
491 }
492 
493 //-----------------------------------------------------------------------------------------------------------
494 // "setupNodeDataMap" will read the node data blocks.
```

```
495 //
496 //-----------------------------------------------------------------------------------------------------------------
497 uint8_t setupNodeDataMap( ) {
498
499     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "setupNodeDataMap\n" );
500
501     uint8_t rStat = rtNvmGetBytes( NVM_NODE_DATA_START, (uint8_t *) &nodeData.map, sizeof( nodeData.map ));
502
503     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
504         printf( "setupNodeDataMap, status: %d\n", rStat );
505
506     return ( rStat );
507 }
508
509 //-----------------------------------------------------------------------------------------------------------------
510 // "setupCdcMap" will read the CDC descriptor from the NVM.
511 //
512 // ??? we should read it from the NVM, which implies that we need a way to handle vanilla boards...
513 // ??? we should also have a file with CDC descriptors for all boards we currently have...
514 // ??? if we detect a board with the vanilla board type, we can set the correct board type and the
515 // reboot...
516 //-----------------------------------------------------------------------------------------------------------------
517 uint8_t setupCdcMap( CDC::CdcConfigDesc *cdcConfig ) {
518
519     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "setupCdcMap\n" );
520
521     uint8_t rStat = ALL_OK;
522
523
524     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
525         printf( "setupCdcMap, status: %d\n", rStat );
526
527     return ( rStat );
528 }
529
530 //-----------------------------------------------------------------------------------------------------------------
531 // The event map stores all event/port pairs this node is interested to process. The map is a sorted map and
532 // there is a high water mark, so that we only read up to the last used entry in the map. Just like other
533 // data structures we could just read in all entries. However, this is a large map. It would be better to
534 // just read up to the HWM, if the HWM is valid. If this is not the case, we have to assume that there are
535 // bigger issues with the event map. In this case we will read the entire map entry by entry, add used
536 // entries, i.e. entries with a non-NIL event ID to the memory map. After reading all entries, the newly
537 // created event map is written back to the NVM place.
538 //
539 //-----------------------------------------------------------------------------------------------------------------
540 uint8_t setupEventMap( ) {
541
542     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) {
543
544         printf( "setupEventMap, entries: %d, HWM: %d\n", nodeMap.eventMapEntries, nodeMap.eventMapHwm );
545     }
546
547     uint8_t rStat = ALL_OK;
548
549     if ( nodeMap.eventMapHwm <= MAX_EVENT_MAP_ENTRIES ) {
550
551         for ( uint16_t i = 0; i < nodeMap.eventMapHwm; i++ ) {
552
553             rStat = rtNvmGetBytes(  NVM_EVENT_MAP_START + i * sizeof( LcsEventMapEntry),
554                                     (uint8_t *) &eventMap.map[ i ],
555                                     sizeof( LcsEventMapEntry ));
556         }
557     }
558     else {
559
560         LcsEventMapEntry e;
561         for ( uint16_t i = 0; i < nodeMap.eventMapEntries; i++ ) eventMap.map[ i ] = e;
562
563         nodeMap.eventMapHwm = 0;
564         for ( uint16_t i = 0; i < nodeMap.eventMapEntries; i++ ) {
565
566             LcsEventMapEntry eventEntry;
567
568             rStat = rtNvmGetBytes(  NVM_EVENT_MAP_START + i * sizeof(LcsEventMapEntry),
569                                     (uint8_t *) &eventEntry,
570                                     sizeof(LcsEventMapEntry));
571
572             if (( rStat == ALL_OK ) && ( eventEntry.eventId != NIL_EVENT_ID )) {
573
574                 addEvent( eventEntry.eventId, eventEntry.portId );
575             }
576         }
577
578         rStat = syncEventMap( );
579     }
580
581     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
582         printf( "setupEventMap, status: %d\n", rStat );
583
584     return ( rStat );
585 }
586
587 //-----------------------------------------------------------------------------------------------------------------
588 // The user map is the additional NVM storage that the chip set offers beyond the area allocated for the
589 // system. Since we have no idea what the user is doing, we do nothing for now. It is just a placeholder.
590 //
591 //-----------------------------------------------------------------------------------------------------------------
592 uint8_t setupUserMap( ) {
593
```

```
594     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "setupUserMap\n" );
595
596     uint8_t rStat = ALL_OK;
597
598     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
599         printf( "setupUserMap, status: %d\n", rStat );
600
601     return ( ALL_OK );
602 }
603
604 //-----------------------------------------------------------------------------------------------------------
605 // "setupCallbackMap" initializes the callback map. We expect the user to register their callbacks between
606 // the runtime init and runtime start routine.
607 //
608 //-----------------------------------------------------------------------------------------------------------
609 uint8_t setupCallbackMap( ) {
610
611     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "setupCallbackMap\n" );
612
613     uint8_t rStat = ALL_OK;
614
615     callbackMap.lcsMsgCallback      = nullptr;
616     callbackMap.dccMsgCallback      = nullptr;
617     callbackMap.cmdLineCallback     = nullptr;
618
619     callbackMap.initCallback        = nullptr;
620     callbackMap.resetCallback       = nullptr;
621     callbackMap.pfailCallback       = nullptr;
622
623     callbackMap.eventCallback       = nullptr;
624     callbackMap.reqCallback         = nullptr;
625     callbackMap.repCallback         = nullptr;
626
627     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
628         printf( "setupCallbackMap, status: %d\n", rStat );
629
630     return( rStat );
631 }
632
633 //-----------------------------------------------------------------------------------------------------------
634 // "setupTaskMap" initializes the task map. A user can register routines that are executed on a periodic
635 // basis.
636 //
637 //-----------------------------------------------------------------------------------------------------------
638 uint8_t setupTaskMap( ) {
639
640     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "setupTaskMap\n" );
641
642     uint8_t rStat = ALL_OK;
643
644     nodeMap.taskMapEntries  = MAX_TASK_MAP_ENTRIES;
645     nodeMap.taskMapHwm      = 0;
646
647     for ( int i = 0; i < MAX_TASK_MAP_ENTRIES; i++ ) {
648
649         taskMap.map[ i ].task       = nullptr;
650         taskMap.map[ i ].interval   = 0;
651         taskMap.map[ i ].timeStamp  = 0;
652     }
653
654     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
655         printf( "setupTaskMap, status: %d\n", rStat );
656
657     return( rStat );
658 }
659
660 //-----------------------------------------------------------------------------------------------------------
661 // "setupPendingReqMap" initializes the pending request map. Currently, we do not use a HWM approach, but
662 // just use all entries when searching the map.
663 //
664 //-----------------------------------------------------------------------------------------------------------
665 uint8_t setupPendingReqMap( ) {
666
667     uint8_t rStat = ALL_OK;
668
669     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "setupPendingReqMap\n" );
670
671     nodeMap.pendingMapEntries   = MAX_PENDING_REQ_MAP_ENTRIES;
672     nodeMap.pendingMapHwm       = MAX_PENDING_REQ_MAP_ENTRIES;
673
674     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
675         printf( "setupPendingReqMap, status: %d\n", rStat );
676
677     return( rStat );
678 }
679
680 //-----------------------------------------------------------------------------------------------------------
681 // "setupDrvFuncMap" initializes the driver function label map. This table is used when we need to find the
682 // driver for an extension board type.
683 //
684 // ??? we could pre register all known drivers... a user could still register a new one and also overwrite
685 // a pre-registered driver with a new func label.
686 //-----------------------------------------------------------------------------------------------------------
687 uint8_t setupDrvFuncMap( ) {
688
689     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "setupDrvLabelMap\n" );
690
691     uint8_t rStat = ALL_OK;
692
```

```
693     nodeMap.drvFuncMapEntries = MAX_DRV_TYPES;
694     nodeMap.drvFuncMapHwm     = MAX_DRV_TYPES;
695 
696     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
697         printf( "setupDrvLabelMap, status: %d\n", rStat );
698 
699     return( rStat );
700 }
701 
702 //-----------------------------------------------------------------------------------------------------------
703 // "setupDrvMap" initializes the driver map. For each possible extension board, up to four, there is an
704 // entry in this map.
705 //
706 //-----------------------------------------------------------------------------------------------------------
707 uint8_t setupDrvMap( ) {
708 
709     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "setupDrvMap\n" );
710 
711     uint8_t rStat = ALL_OK;
712 
713     nodeMap.pendingMapEntries   = MAX_PENDING_REQ_MAP_ENTRIES;
714     nodeMap.pendingMapHwm       = MAX_PENDING_REQ_MAP_ENTRIES;
715 
716     buildDefaultDrvMap( &drvMap );
717 
718     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
719         printf( "setupDrvMap, status: %d\n", rStat  );
720 
721     return( rStat );
722 }
723 
724 //-----------------------------------------------------------------------------------------------------------
725 // The runtime library will one day perhaps a set of internal functions to execute periodically. Right now,
726 // this routine will do nothing.
727 //
728 //-----------------------------------------------------------------------------------------------------------
729 uint8_t registerInternalTasks( ) {
730 
731     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "registerInternalTasks\n" );
732 
733     uint8_t rStat = ALL_OK;
734 
735 
736     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
737         printf( "registerInternalTasks, status: %d\n" );
738 
739     return( rStat );
740 }
741 
742 //-----------------------------------------------------------------------------------------------------------
743 // With the node properly initialized, it is time to see whether we have connected extension boards. An
744 // extension board has also a small NVM on the board that will tell us what the board type is and keep driver
745 // configuration data. There can be up to four boards, numbered from 0 to 3. The runtime has a driver map
746 // with four extension descriptor entries. After initializing the driver map, we attempt to read from each
747 // extension NVM on an extension board. If the read fails, there is no board at that location, which will
748 // mark in the flags field. The drvMap HWM will tell us how many board we actually found. Note that the
749 // boards are by hardware always connected in the order 0,1,2 and 3. If for example only two boards are
750 // connected, the HWM would be 2.
751 //
752 //-----------------------------------------------------------------------------------------------------------
753 uint8_t detectExtensionBoards( ) {
754 
755     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "detectExtensionBoards\n" );
756 
757     uint8_t rStat = ALL_OK;
758 
759     for ( int i = 0; i < MAX_EXT_BOARD_MAP_ENTRIES; i++ ) {
760 
761         if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
762             printf( "detectExtensionBoard, boardId: %d\n", i );
763 
764         LcsDrvEntry *drvEntry = &drvMap.map[ i ];
765 
766         rStat = extNvmGetBytes( i, 0, (uint8_t *) &drvEntry -> extBoard, sizeof( LcsDrvBoardDesc ));
767         if ( rStat == ALL_OK ) {
768 
769             uint16_t *ptr = (uint16_t *) &drvEntry -> extBoard;
770 
771             if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) {
772 
773                 printf( "Extension Board Desc Head: " );
774                 for ( int j = 0; j < 16; j++ ) printf( "0x%x ", ptr[ j ] );
775                 printf( "\n" );
776             }
777 
778             nodeMap.nodeFlags |= NFLAGS_EXT_PRESENT;
779             nodeMap.drvMapHwm ++;
780 
781             drvEntry -> flags |= BF_EXT_BOARD_PRESENT;
782 
783             if (( drvEntry -> extBoard.head.magicWord1 == NVM_MWORD_1 ) &&
784                 ( drvEntry -> extBoard.head.magicWord2 == NVM_MWORD_2 )) {
785 
786                 drvEntry -> flags |= BF_EXT_BOARD_VALID;
787 
788                 if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
789                     printf( "detectExtensionBoard, boardId: %d -> valid\n", i );
790             }
791             else  {
```

```
792
793                 if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
794                     printf( "detectExtensionBoard, boardId: %d -> inValid\n", i );
795             }
796         }
797         else {
798
799             if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
800                 printf( "detectExtensionBoard, boardId: %d, rStat: %d\n", i, rStat );
801         }
802     }
803
804     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
805         printf( "detectExtensionBoards, status: %d\n", ALL_OK );
806
807     return ( ALL_OK );
808 }
809
810 //-----------------------------------------------------------------------------------------------------------
811 // "lookupDrvFunc" searches the driver function label map. It is called when we setup the extension board.
812 // When the type is unknown, a nullptr is returned.
813 //
814 //-----------------------------------------------------------------------------------------------------------
815 LcsDrvReqFunc lookupDrvFunc( uint16_t drvType ) {
816
817     for ( int i = 0; i < MAX_DRV_TYPES; i++ ) {
818
819         if ( drvFuncMap.map[ i ].drvType == drvType ) return( drvFuncMap.map[ i ].drvFunc );
820     }
821
822     return( nullptr );
823 }
824
825 //-----------------------------------------------------------------------------------------------------------
826 // For all detected extension boards, we will first check that the board descriptor at slot "n" is there and
827 // that the board descriptor is reasonable. If so, the board type is used to load the respective driver.
828 // Note that during normal operations we cannot manipulate the NVM, as it is read protected. The jumper on
829 // the extension board needs to be removed for this. When removed, the extension board NVM can be written to
830 // with commands from the runtime.
831 //
832 // When the driver function is not pre registered, we do not have a function to set in the driver map.
833 // In this case the driver is not usable yet. The problem is that we can only make driver registration
834 // calls after LcsInitRuntime. But the all driver boards have been detected. There are two ways. For driver
835 // know already, we "pre-register" them. For any other driver, the driver function registration routine will
836 // check the driver table for the driver type and the patch the function label. All this should takes place
837 // before the final call to "startRuntime".
838 //
839 //-----------------------------------------------------------------------------------------------------------
840 uint8_t setupExtensionBoards( ) {
841
842     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "setupExtensionBoards\n" );
843
844     uint8_t rStat = ALL_OK;
845
846     for ( int i = 0; i < MAX_EXT_BOARD_MAP_ENTRIES; i++ ) {
847
848         LcsDrvEntry *drvEntry = &drvMap.map[ i ];
849
850         if (( drvEntry -> flags & BF_EXT_BOARD_PRESENT ) && ( drvEntry -> flags & BF_EXT_BOARD_VALID )) {
851
852             LcsDrvReqFunc drvFunc = lookupDrvFunc( drvEntry -> extBoard.head.boardType );
853
854             if ( drvFunc != nullptr ) {
855
856                 drvEntry -> drvFunc = drvFunc;
857                 drvEntry -> flags   |= BF_EXT_BOARD_READY;
858                 drvEntry -> lastErr = drvEntry -> drvFunc( i, ITEM_ID_RESET, nullptr, nullptr );
859
860                 if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) {
861
862                     printf( "Driver setup, type: %d, stat: %d\n",
863                         drvEntry -> extBoard.head.boardType, drvEntry -> lastErr );
864                 }
865             }
866             else {
867
868                 if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) {
869
870                     printf( "Driver setup, type not found: %d\n", drvEntry -> extBoard.head.boardType );
871                 }
872             }
873         }
874     }
875
876     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
877         printf( "setupExtensionBoards, status: %d\n", rStat );
878
879     return ( rStat );
880 }
881
882 //-----------------------------------------------------------------------------------------------------------
883 // Driver function registration. There is a simple table which maintains extension boards types and the
884 // driver function form them. If the type is already registered, we just overwrite the function signature.
885 // Otherwise we find a free entry and use it.
886 //
887 // The driver function registration can only be called after the initialization of the LCS runtime. By then
888 // the extension boards are however already detected, and if there are no drivers pre-registered no driver
889 // function was registered. The extension board is therefore not ready and was also not reseted. Consequently,
890 // the  registration call will check or detected valid extension boards and patch the driver function label
```

```
// and invoke the RESET request.
//
//-------------------------------------------------------------------------------------------------------
uint8_t registerDrvFunc(  uint16_t drvType, LcsDrvReqFunc drvReqFunction ) {

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
        printf( "registerDrvFunc, type: %d, func: %p\n", drvType, drvReqFunction );

    bool found = false;

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

        if ( drvFuncMap.map[ i ].drvType == drvType ) {

            if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
                printf( "registerDrvFunc, overwrite: %d\n", i );

            drvFuncMap.map[ i ].drvFunc = drvReqFunction;
            found = true;
            break;
        }
    }

    if ( ! found ) {

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

            if ( drvFuncMap.map[ i ].drvType == BT_NIL ) {

                if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
                    printf( "registerDrvFunc, allocate: %d\n", i );

                drvFuncMap.map[ i ].drvType = drvType;
                drvFuncMap.map[ i ].drvFunc = drvReqFunction;
                found = true;
                break;
            }
        }
    }

    if ( ! found ) {

        if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
                printf( "registerDrvFunc, table full\n" );

        return( ERR_DRV_FUNC_MAP_FULL );
    }

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

        LcsDrvEntry *drvEntry = &drvMap.map[ i ];

        if (( drvEntry -> flags & BF_EXT_BOARD_PRESENT ) && ( drvEntry -> flags & BF_EXT_BOARD_VALID )) {

            if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
                printf( "registerDrvFunc, set func for board: %d\n", i );

            if ( drvEntry -> extBoard.head.boardType == drvType ) {

                drvEntry -> drvFunc     = drvReqFunction;
                drvEntry -> flags       |= BF_EXT_BOARD_READY;
                drvEntry -> lastErr     = drvEntry -> drvFunc( i, ITEM_ID_RESET, nullptr, nullptr );
            }

        }
     }

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
                printf( "registerDrvFunc, ret: ALL_OK\n" );

    return( ALL_OK );
}

//-------------------------------------------------------------------------------------------------------
// "powerFailHandler" is the routine called when the hardware detects an imminent loss of power. We will save
// crucial data to NVM. Finally, the optionally registered firmware power fail callback is called. The node
// state becomes "PFAIL".
//
//-------------------------------------------------------------------------------------------------------
uint8_t powerFailHandler( ) {

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "powerFailHandler\n" );

    uint8_t rStat = ALL_OK;

    nodeMap.nodeState = NS_PFAIL;
    rStat = rtNvmPutWord( NVM_NODE_MAP_START + offsetof( LcsNodeMap, nodeState ), NS_PFAIL );

    if ( callbackMap.pfailCallback != nullptr ) callbackMap.pfailCallback( nodeMap.nodeId );

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
        printf( "powerFailHandler, status: %d\n", rStat );

    return ( rStat );
}

//-------------------------------------------------------------------------------------------------------
// "resetNode" restarts a node. We first rebuild the MEM areas from their NVM counterparts. Next, the optional
// reset call back is invoked. Finally, all ports are reseted as well.
```

```
990 //
991 // ??? read NVM to MEM.
992 // ??? would a node reset clear any outstanding requests ? or do we need to inform potential waiters ?
993 // ??? would it just drop all periodic tasks ? who registers them again ?
994 // ??? how do we make sure we only cover ports that are used ?
995 // ??? or would we need to be a bit more sensible what reset mean ?
996 //-------------------------------------------------------------------------------------------------------
997 uint8_t resetNode( uint16_t npId ) {
998
999     uint8_t rStat = ALL_OK;
1000
1001     if ( callbackMap.resetCallback != nullptr ) {
1002
1003          // ??? load MEM from NVM....
1004
1005         if ( callbackMap.resetCallback != nullptr ) {
1006
1007             rStat = callbackMap.resetCallback( buildNpId( nodeMap.nodeId, 0  ));
1008         }
1009
1010
1011     }
1012
1013     if ( rStat == ALL_OK ) {
1014
1015         for ( uint8_t i = 1; i <= MAX_PORT_MAP_ENTRIES; i++ ) {
1016
1017             // ??? load MEM from NVM....
1018
1019             if ( callbackMap.resetCallback != nullptr ) {
1020
1021                 rStat = callbackMap.resetCallback( buildNpId( nodeMap.nodeId, i ));
1022             }
1023         }
1024     }
1025
1026     if ( rStat == ALL_OK ) {
1027
1028         // ??? reset the drivers...
1029     }
1030
1031     return ( rStat );
1032 }
1033
1034 //-------------------------------------------------------------------------------------------------------
1035 // The LCS library has a set of configuration parameters. Currently this is only the "options" field. In the
1036 // future there may be more fields. This routines returns a config structure with reasonable defaults set.
1037 //
1038 //-------------------------------------------------------------------------------------------------------
1039 LcsConfigDesc getConfigDefault( ) {
1040
1041     LcsConfigDesc cfg;
1042
1043     cfg.options = 0;
1044
1045     return( cfg );
1046 }
1047
1048 //-------------------------------------------------------------------------------------------------------
1049 // "initRuntime" is the routine that takes a controller board and initializes the whole show. It is the very
1050 // first thing to call in a node firmware program. There is a lot to do. First, the CDC layer is initialized.
1051 // NVM and CanBus follow. An error in this stage will result in a fatal error, we are not able to set up a
1052 // valid runtime.
1053 //
1054 // If the HW setup worked, we are ready to read in the nodeMap. A nodeMap can be valid or not. It is defined
1055 // as a map with valid "magic" words and reasonable values for the other fields. In case of an invalid
1056 // nodeMap, a new default map is created and written back to the NVM. An invalid nodeMap could result from
1057 // erroneous writes to NVM locations or simply a brand new HW board. If all is OK, we have a valid basic
1058 // nodeMap that we can work from.
1059 //
1060 // The setup of the portMap follows. The flag field contains dynamic flags that are always reseted on node
1061 // start or reset. Other fields in a map are read in from the NVM first and set to a default state this way.
1062 //
1063 // The eventMap initialization is a bit special, in that it is a rather large map and potentially only a
1064 // portion is used. There is an eventMao high water mark field in the nodeMap that will tell how many entries
1065 // are actually used in the event map. Adding increases, deleting decreases the high water mark. Note that
1066 // the eventMap is a sorted map. Every time we insert or remove the eventMap is rebuilt. Instead of
1067 // immediately updating the NVM storage, a dedicated command will SYNC between the MEM and the NVM eventMap.
1068 //
1069 // Next, we will set up the pending request map, callback function and task map. They are just memory data
1070 // areas to be initialized. Up to here an error detected will result in a fatal error. If a console is
1071 // connected the error messages are listed for analysis.
1072 //
1073 // If all is OK so far, extension boards are located, and if there are any, their initialization follows.
1074 // First, we try to detect any. For all detected entries we validate the extension board NVM header and
1075 // set the driver for a valid header found. The driver data area is copied to its memory counter part.
1076 // All drivers are ready by then.
1077 //
1078 // The overall logic of the startup routine code below is that if there is a fault, the follow on steps are
1079 // simply  skipped and the node is put into the FAIL state. Note that we still are able to access the node
1080 // via the USB console and one day also via diagnostic LCS messages. The idea is to allow the correct
1081 // configuration  of the nodeMap, so that we can restart with a correct nodeMap.
1082 //
1083 // ??? how do we deal wit PFAIL restarts ?
1084 // ??? we could have also callbacks for the "restart" case ? or pass to init a flag...
1085 //-------------------------------------------------------------------------------------------------------
1086 uint8_t initRuntime( LcsConfigDesc *lcsConfig, CDC::CdcConfigDesc *cdcConfig ) {
1087
1088     uint8_t rStat = ALL_OK;
```

```
1089
1090        if ( rStat == ALL_OK )  rStat = initCdcLayer( cdcConfig );
1091        if ( rStat != ALL_OK )  CDC::fatalErrorMsg((char *) "Fatal: CDC Setup failed", 1, rStat );
1092
1093        CDC::writeDio( cdcConfig -> READY_LED_PIN, false );
1094        CDC::writeDio( cdcConfig -> ACTIVE_LED_PIN, true );
1095
1096        if ( rStat == ALL_OK )  rStat = initCanBus( cdcConfig );
1097        if ( rStat == ALL_OK )  rStat = initNvmChannels( cdcConfig );
1098        if ( rStat != ALL_OK )  CDC::fatalErrorMsg((char *) "Fatal: CAN bus or NVM Setup failed", 2, rStat );
1099
1100        if ( rStat == ALL_OK )  rStat = setupNodeMap( lcsConfig );
1101        if ( rStat == ALL_OK )  rStat = setupCdcMap( cdcConfig );
1102        if ( rStat == ALL_OK )  rStat = setupPortMap( );
1103        if ( rStat == ALL_OK )  rStat = setupNodeDataMap( );
1104        if ( rStat == ALL_OK )  rStat = setupEventMap( );
1105        if ( rStat == ALL_OK )  rStat = setupUserMap( );
1106        if ( rStat == ALL_OK )  rStat = setupCallbackMap( );
1107        if ( rStat == ALL_OK )  rStat = setupTaskMap( );
1108        if ( rStat == ALL_OK )  rStat = setupPendingReqMap( );
1109        if ( rStat == ALL_OK )  rStat = setupDrvFuncMap( );
1110        if ( rStat == ALL_OK )  rStat = setupDrvMap( );
1111        if ( rStat == ALL_OK )  rStat = registerInternalTasks( );
1112        if ( rStat != ALL_OK )  CDC::fatalErrorMsg((char *) "Node setup Setup failed", 3, rStat );
1113
1114        if ( rStat == ALL_OK )  rStat = detectExtensionBoards( );
1115        if ( rStat == ALL_OK )  rStat = setupExtensionBoards( );
1116        if ( rStat != ALL_OK )  printf( "Extension boards setup Setup failed, stat: %d\n", rStat );
1117
1118        if ( rStat == ALL_OK )  nodeMap.nodeState = NS_INIT;
1119        else                    nodeMap.nodeState = NS_FAIL;
1120
1121        if ( rStat == ALL_OK )  {
1122
1123            CDC::writeDio( cdcConfig -> READY_LED_PIN, true );
1124            CDC::writeDio( cdcConfig -> ACTIVE_LED_PIN, false );
1125        }
1126
1127        if ( debugMask & ( DBG_CONFIG && DBG_SETUP )) printf( "init LCS runtime, status: %d \n", rStat );
1128        return ( rStat );
1129    }
1130
1131    //-----------------------------------------------------------------------------------------------------
1132    // "startRuntime" is the main routine of the node activity processing. All it does is to call the node
1133    // state machine.
1134    //
1135    //-----------------------------------------------------------------------------------------------------
1136    void startRuntime( ) {
1137
1138         if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "Start LCS runtime\n");
1139
1140         handleNodeState( );
1141    }
1142
1143    }; // namespace LCS
```

```
//-----------------------------------------------------------------------------------------------------------------
//
// "LcsRtMsgBus" - implementation file.
//
//-----------------------------------------------------------------------------------------------------------------
// At the message level, the LCS runtime offers a message to which all nodes are connected. Currently, it is
// a CAN bus. Pretty straightforward and robust. This file contains the routines to set up the communication
// as well as a set of convenience functions for sending a LCS message taking care of filling the message
// frame. Some LCS message are of a "request/reply" nature. When a request is sent out a entry is made in
// the pending request map. Since the message layer sees all reply message, this pending map is used to
// filter for the request we are waiting for.
//
//-----------------------------------------------------------------------------------------------------------------
//
// LCS - Core Library
// Copyright (C) 2021 - 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 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/>.
//
//-----------------------------------------------------------------------------------------------------------------
#include "LcsRuntimeLib.h"
#include "LcsRtLibInt.h"

//-----------------------------------------------------------------------------------------------------------------
// External declaration to global structures defined in "LcsRtSetup".
//
//-----------------------------------------------------------------------------------------------------------------
namespace LCS {

    extern uint16_t                 debugMask;
    extern LCS::LcsNodeMap          nodeMap;
    extern LCS::LcsCallbackMap      callbackMap;
    extern LCS::LcsPendingReqMap    pendingReqMap;
    extern LCS::LcsTaskMap          taskMap;
    extern LCS::LcsMsgBusCAN        *msgBus;
};

//-----------------------------------------------------------------------------------------------------------------
// File local declarations.
//
//-----------------------------------------------------------------------------------------------------------------
namespace {

using namespace LCS;

//-----------------------------------------------------------------------------------------------------------------
// Little helper functions and constants.
//
//-----------------------------------------------------------------------------------------------------------------
const uint32_t DEF_REQ_TIMEOUT_VAL_MS = 50000;

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

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

uint16_t buildNpId( uint16_t nodeId, uint16_t portId ) {

    return(( nodeId << 4 ) | ( portId & 0xF ));
}

uint16_t nodeId( uint16_t npId ) {

    return( npId >> 4 );
}

uint16_t portId( uint16_t npId ) {

    return( npId & 0xF );
}

uint8_t lowByte( uint16_t arg ) {

    return( arg & 0xFF );
}

uint8_t highByte( uint16_t arg ) {

    return( arg >> 8 );
}

//-----------------------------------------------------------------------------------------------------------------
// There are some LCS messages that expect a reply message. The library maintains a small pending request
// buffer. When a request type message is sent we add the target node and a timer value to the buffer. Easy
// and simple. Note that there can be more than one entry for the same node / port combination in the buffer.
// If the buffer is full, an error is returned. We have too many outstanding requests then.
//
// A request can also be registered with a timeout value. When the timeout expires, the caller is informed
// that the request timed out. A timeout value of zero means that we wait indefinitely.
//
//-----------------------------------------------------------------------------------------------------------------
```

```
99  uint8_t addToPendingReqMap( uint16_t npId, uint32_t timeoutVal = 0 ) {
100
101     uint32_t ts = CDC::getMillis( );
102
103     for ( uint8_t i = 0; i < MAX_PENDING_REQ_MAP_ENTRIES; i++ ) {
104
105         if ( pendingReqMap.map[ i ].npId == 0 ) {
106
107             pendingReqMap.map[ i ].npId = npId;
108             pendingReqMap.map[ i ].reqTimeoutTs = (( timeoutVal != 0 ) ? ts + timeoutVal : timeoutVal );
109             return ( ALL_OK );
110         }
111     }
112
113     return ( ERR_PENDING_REQ_MAP_FULL );
114 }
115
116 //-----------------------------------------------------------------------------------------------------------
117 // "removeFromPendingReqMap" removes an entry from the pending reply buffer. If the entry is not found, we
118 // received a reply for a request that we do not know. Right now, we just ignore this error.
119 //
120 //-----------------------------------------------------------------------------------------------------------
121 uint8_t removeFromPendingReqMap( uint16_t npId ) {
122
123     for ( uint8_t i = 0; i < MAX_PENDING_REQ_MAP_ENTRIES; i++ ) {
124
125         if ( pendingReqMap.map[ i ].npId == npId ) pendingReqMap.map[ i ].npId = NIL_NODE_ID;
126     }
127
128     return ( ALL_OK );
129 }
130
131 //-----------------------------------------------------------------------------------------------------------
132 // "searchPendingReqMap" searches the pending request buffer for a matching node. We just return a boolean
133 // answer whether the entry is there or not.
134 //
135 //-----------------------------------------------------------------------------------------------------------
136 bool searchPendingReqMap( uint16_t npId ) {
137
138     for ( uint8_t i = 0; i < MAX_PENDING_REQ_MAP_ENTRIES; i++ ) {
139
140         if ( pendingReqMap.map[ i ].npId == npId ) return ( true );
141     }
142
143     return ( false );
144 }
145
146 }; // namespace
147
148
149 //-----------------------------------------------------------------------------------------------------------
150 // The LCS name space routines declared in this file.
151 //
152 //-----------------------------------------------------------------------------------------------------------
153 namespace LCS {
154
155 //-----------------------------------------------------------------------------------------------------------
156 // "setupMsgBus" is called during node initialization to setup the LCS message bus interface. Right now,
157 // only the CAN bus is supported. We first create the CAN Bus object and then call its initialization routine.
158 // The canId and nodeId are identical. We ensure through LCS configuration that the nodeIds are unique.
159 //
160 //
161 // ??? how do we configure the CAN control mode ?
162 //-----------------------------------------------------------------------------------------------------------
163 uint8_t setupMsgBus( ) {
164
165     uint8_t     rStat           = ALL_OK;
166     uint16_t    canBusCtrlMode  = CAN_BUS_LIB_PICO_PIO_125K_M_CORE;
167     uint8_t     canBusTxPin     = 0;
168     uint8_t     canBusRxPin     = 0;
169
170     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_MSG_BUS )) {
171
172         printf( "setupMsgBus -> %d:%d:%d%d\n", nodeMap.nodeId, canBusRxPin, canBusTxPin, canBusCtrlMode );
173     }
174
175     switch ( canBusCtrlMode ) {
176
177         case CAN_BUS_LIB_PICO_PIO_125K:
178         case CAN_BUS_LIB_PICO_PIO_250K:
179         case CAN_BUS_LIB_PICO_PIO_500K:
180         case CAN_BUS_LIB_PICO_PIO_1000K:
181         case CAN_BUS_LIB_PICO_PIO_125K_M_CORE:
182         case CAN_BUS_LIB_PICO_PIO_250K_M_CORE:
183         case CAN_BUS_LIB_PICO_PIO_500K_M_CORE:
184         case CAN_BUS_LIB_PICO_PIO_1000K_M_CORE: {
185
186             msgBus = new LcsMsgBusCAN( );
187             rStat = (( LcsMsgBusCAN *) msgBus ) -> init( nodeMap.nodeId, canBusRxPin, canBusTxPin, canBusCtrlMode );
188
189         } break;
190
191         default: rStat = ERR_CAN_SETUP;
192     }
193
194     if ( rStat != ALL_OK ) {
195
196          if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_MSG_BUS ))
197             printf( "setup CAN Bus failed: %d\n", rStat );
```

```
198
199         return ( ERR_CAN_SETUP );
200     }
201      else {
202
203          if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_MSG_BUS )) printf( " -> OK\n" );
204         return ( ALL_OK );
205     }
206 }
207
208 //-----------------------------------------------------------------------------------------------------------
209 // The primary task of the receive function is to receive an LCS messages and pass them to the respective
210 // handler method. In order to not always check whether a valid message was processed, this routine will
211 // always return a valid message opCode. The "LCS_NO_MSG" pseudo message is used to indicate that something
212 // else happened and no further message processing is required. We also maintain a request / reply map to
213 // keep track of outstanding requests transparently to the caller.
214 //
215 // ??? should we have a pre-filter for message ID and nodeId match ? this would be perhaps useful, when we
216 // run CAN bus on two cores on the pico. Then the checking would run on the interrupt handler processor.
217 // However, the pending request map is then accessed by two cores and we need to sync the access.
218 //-----------------------------------------------------------------------------------------------------------
219 uint8_t receiveLcsMsg( uint8_t *msg ) {
220
221     int rStat = msgBus -> receiveLcsMsg( msg );
222
223     if ( rStat == ALL_OK )  {
224
225          if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_MSG_BUS )) {
226
227              printf( "Can Msg Received (OpCode): 0x%x\n", msg[ 0 ] );
228         }
229
230         if (( msg[ 0 ] == LCS_OP_NODE_REP ) || ( msg[ 0 ] == LCS_OP_ACK ) || ( msg[ 0 ] == LCS_OP_ERR )) {
231
232              uint16_t nodeId = (( msg[1] << 8 ) + msg[2] ) >> 4;
233
234             if ( searchPendingReqMap( nodeId )) {
235
236                 removeFromPendingReqMap( nodeId );
237                 return ( msg[ 0 ] );
238
239             } else return ( LCS_OP_NO_MSG );
240
241         } else return ( msg[ 0 ] );
242
243     } else return ( LCS_OP_NO_MSG );
244 }
245
246 //-----------------------------------------------------------------------------------------------------------
247 // A simple helper to print an LCS message.
248 //
249 //-----------------------------------------------------------------------------------------------------------
250 void printLcsMsg( uint8_t *msg ) {
251
252     printf( "LCS MSG: op: %d, data: ", msg[ 0 ] & 0x1F );
253     for ( int i = 0; i < ( msg[ 0 ] >> 5 ) + 1; i ++ ) printf( "0x%x ", msg[ i ] );
254     printf( "\n" );
255 }
256
257 //-----------------------------------------------------------------------------------------------------------
258 // "processPendingReqMapTimeouts" is part of the periodic processing of the node. It will check wether any
259 // requests waiting for a reply have timed out. In this case, we should invoke the reply callback with an
260 // error code.
261 //
262 //-----------------------------------------------------------------------------------------------------------
263 void processPendingReqMapTimeouts( ) {
264
265     uint32_t ts = CDC::getMillis( );
266
267     for ( uint8_t i = 0; i < MAX_PENDING_REQ_MAP_ENTRIES; i++ ) {
268
269         if ( pendingReqMap.map[ i ].reqTimeoutTs != 0 ) {
270
271             if ( pendingReqMap.map[ i ].reqTimeoutTs < ts ) {
272
273                 if ( callbackMap.repCallback != nullptr ) {
274
275                     callbackMap.repCallback( pendingReqMap.map[ i ].npId, 0, 0, 0, ERR_REQ_TIMEOUT );
276                 }
277
278                 // ??? clear entry ?
279
280             }
281         }
282     }
283 }
284
285 //-----------------------------------------------------------------------------------------------------------
286 // Some messages are requests that expect a reply. We maintain a pending request map which keeps track of
287 // outstanding requests.
288 //
289 //-----------------------------------------------------------------------------------------------------------
290 uint8_t sendTimedReq( uint16_t npId, uint8_t *msg, uint8_t msgPri, uint32_t timeout ) {
291
292     if ( addToPendingReqMap( npId ) == ALL_OK ) {
293
294         return ( msgBus -> sendLcsMsg( msg, msgPri ));
295     }
296     else return ( ERR_NODE_OUTSTANDING_REQ_LIMIT );
```

```
297 }
298
299 //-----------------------------------------------------------------------------------------------------------
300 // LCB message send routines. They all follow the same pattern. There is a method for each message opcode,
301 // which maps the input parameters to the byte array and then send it. Straightforward. For messages that are
302 // a part of a request / reply pair, the requesting messages will also add the requesting nodeId to the
303 // pending request map. This way we know that there is an outstanding request. The receiving message handler
304 // will clear the entry upon the matching receipt.
305 //
306 //-----------------------------------------------------------------------------------------------------------
307 uint8_t sendCfg( uint16_t npId ) {
308
309     uint8_t msgBuf[ 8 ] = { LCS_OP_CFG };
310     msgBuf[ 1 ] = highByte( npId );
311     msgBuf[ 2 ] = lowByte( npId );
312
313     return( sendTimedReq( npId, msgBuf,  MSG_PRI_HIGH , 0 ));
314 }
315
316 uint8_t sendOps( uint16_t npId ) {
317
318     uint8_t msgBuf[ 8 ] = { LCS_OP_OPS };
319     msgBuf[ 1 ] = highByte( npId );
320     msgBuf[ 2 ] = lowByte( npId );
321
322     return( sendTimedReq( npId, msgBuf,  MSG_PRI_HIGH , 0 ));
323 }
324
325 uint8_t sendReset( uint16_t npId ) {
326
327     uint8_t msgBuf[ 8 ] = { LCS_OP_RESET };
328     msgBuf[ 1 ] = highByte( npId );
329     msgBuf[ 2 ] = lowByte( npId );
330
331     return ( sendTimedReq( npId, msgBuf, MSG_PRI_HIGH, 0 ));
332 }
333
334 uint8_t sendBusOn( ) {
335
336     uint8_t msgBuf[ 8 ] = { LCS_OP_BON };
337     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_VERY_HIGH ));
338 }
339
340 uint8_t sendBusOff( ) {
341
342     uint8_t msgBuf[ 8 ] = { LCS_OP_BOF };
343     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_VERY_HIGH ));
344 }
345
346 uint8_t sendErr( uint16_t npId, uint8_t errCode, uint8_t arg1, uint8_t arg2 ) {
347
348     uint8_t msgBuf[ 8 ] = { LCS_OP_ERR };
349     msgBuf[ 1 ] = highByte( npId );
350     msgBuf[ 2 ] = lowByte( npId );
351     msgBuf[ 3 ] = errCode;
352     msgBuf[ 4 ] = arg1;
353     msgBuf[ 5 ] = arg2;
354     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
355 }
356
357 uint8_t sendPing( uint16_t npId ) {
358
359     uint8_t msgBuf[ 8 ] = { LCS_OP_PING };
360     msgBuf[ 1 ] = highByte( npId );
361     msgBuf[ 2 ] = lowByte( npId );
362     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
363 }
364
365 uint8_t sendAck( uint16_t npId ) {
366
367     uint8_t msgBuf[ 8 ] = { LCS_OP_ACK };
368     msgBuf[ 1 ] = highByte( npId );
369     msgBuf[ 2 ] = lowByte( npId );
370     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
371 }
372
373 uint8_t sendReqNodeId( uint16_t npId, uint32_t nodeUID, uint8_t flags ) {
374
375     uint8_t msgBuf[ 8 ] = { LCS_OP_REQ_NID };
376     msgBuf[ 1 ] = highByte( npId );
377     msgBuf[ 2 ] = lowByte( npId );
378     msgBuf[ 3 ] = ( nodeUID & 0xFF000000 ) >> 24;
379     msgBuf[ 4 ] = ( nodeUID & 0x00FF0000 ) >> 16;
380     msgBuf[ 5 ] = ( nodeUID & 0x0000FF00 ) >> 8;
381     msgBuf[ 6 ] = ( nodeUID & 0x000000FF );
382     msgBuf[ 7 ] = flags;
383     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
384 }
385
386 uint8_t sendRepNodeId( uint16_t npId, uint32_t nodeUID ) {
387
388     uint8_t msgBuf[ 8 ] = { LCS_OP_REP_NID };
389     msgBuf[ 1 ] = highByte( npId );
390     msgBuf[ 2 ] = lowByte( npId );
391     msgBuf[ 3 ] = ( nodeUID & 0xFF000000 ) >> 24;
392     msgBuf[ 4 ] = ( nodeUID & 0x00FF0000 ) >> 16;
393     msgBuf[ 5 ] = ( nodeUID & 0x0000FF00 ) >> 8;
394     msgBuf[ 6 ] = ( nodeUID & 0x000000FF );
395     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
```

```
396 }
397
398 uint8_t sendSetNodeId( uint16_t npId, uint32_t nodeUID ) {
399
400     uint8_t msgBuf[ 8 ] = { LCS_OP_SET_NID };
401     msgBuf[ 1 ] = highByte( npId );
402     msgBuf[ 2 ] = lowByte( npId );
403     msgBuf[ 3 ] = ( nodeUID & 0xFF000000 ) >> 24;
404     msgBuf[ 4 ] = ( nodeUID & 0x00FF0000 ) >> 16;
405     msgBuf[ 5 ] = ( nodeUID & 0x0000FF00 ) >> 8;
406     msgBuf[ 6 ] = ( nodeUID & 0x000000FF );
407     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
408 }
409
410 uint8_t sendNodeIdCollision( uint16_t npId, uint32_t nodeUID ) {
411
412     uint8_t msgBuf[ 8 ] = { LCS_OP_NCOL };
413     msgBuf[ 1 ] = highByte( npId );
414     msgBuf[ 2 ] = lowByte( npId );
415     msgBuf[ 3 ] = ( nodeUID & 0xFF000000 ) >> 24;
416     msgBuf[ 4 ] = ( nodeUID & 0x00FF0000 ) >> 16;
417     msgBuf[ 5 ] = ( nodeUID & 0x0000FF00 ) >> 8;
418     msgBuf[ 6 ] = ( nodeUID & 0x000000FF );
419     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_HIGH ));
420 }
421
422 uint8_t sendGetNode( uint16_t npId, uint8_t item, uint16_t val1, uint16_t val2 ) {
423
424     uint8_t msgBuf[ 8 ] = { LCS_OP_NODE_GET };
425     msgBuf[ 1 ] = highByte( npId );
426     msgBuf[ 2 ] = lowByte( npId );
427     msgBuf[ 3 ] = item;
428     msgBuf[ 4 ] = highByte( val1 );
429     msgBuf[ 5 ] = lowByte( val1 );
430     msgBuf[ 6 ] = highByte( val2 );
431     msgBuf[ 7 ] = lowByte( val2 );
432     return( sendTimedReq( npId, msgBuf,  MSG_PRI_NORMAL, 0 ));
433 }
434
435 uint8_t sendSetNode( uint16_t npId, uint8_t item, uint16_t val1, uint16_t val2 ) {
436
437     uint8_t msgBuf[ 8 ] = { LCS_OP_NODE_PUT };
438     msgBuf[ 1 ] = highByte( npId );
439     msgBuf[ 2 ] = lowByte( npId );
440     msgBuf[ 3 ] = item;
441     msgBuf[ 4 ] = highByte( val1 );
442     msgBuf[ 5 ] = lowByte( val1 );
443     msgBuf[ 6 ] = highByte( val2 );
444     msgBuf[ 7 ] = lowByte( val2 );
445     return( sendTimedReq( npId, msgBuf,  MSG_PRI_NORMAL, 0 ));
446 }
447
448 uint8_t sendReqNode( uint16_t npId, uint8_t item, uint16_t val1, uint16_t val2 ) {
449
450     uint8_t msgBuf[ 8 ] = { LCS_OP_NODE_REQ };
451     msgBuf[ 1 ] = highByte( npId );
452     msgBuf[ 2 ] = lowByte( npId );
453     msgBuf[ 3 ] = item;
454     msgBuf[ 4 ] = highByte( val1 );
455     msgBuf[ 5 ] = lowByte( val1 );
456     msgBuf[ 6 ] = highByte( val2 );
457     msgBuf[ 7 ] = lowByte( val2 );
458     return( sendTimedReq( npId, msgBuf,  MSG_PRI_LOW, 0 ));
459 }
460
461 uint8_t sendRepNode( uint16_t npId, uint8_t item, uint16_t val1, uint16_t val2 ) {
462
463     uint8_t msgBuf[ 8 ] = { LCS_OP_NODE_REP };
464     msgBuf[ 1 ] = highByte( npId );
465     msgBuf[ 2 ] = lowByte( npId );
466     msgBuf[ 3 ] = item;
467     msgBuf[ 4 ] = highByte( val1 );
468     msgBuf[ 5 ] = lowByte( val1 );
469     msgBuf[ 6 ] = highByte( val2 );
470     msgBuf[ 7 ] = lowByte( val2 );
471     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
472 }
473
474 uint8_t sendEventOn( uint16_t npId, uint16_t eventId ) {
475
476     uint8_t msgBuf[ 8 ] = { LCS_OP_EVT_ON };
477     msgBuf[ 1 ] = highByte( npId );
478     msgBuf[ 2 ] = lowByte( npId );
479     msgBuf[ 3 ] = highByte( eventId );
480     msgBuf[ 4 ] = lowByte( eventId );
481
482      if ( nodeId( npId ) == nodeMap.nodeId ) {
483
484         handleMsgEvent( msgBuf );
485         return( ALL_OK );
486     }
487     else return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
488 }
489
490 uint8_t sendEventOff( uint16_t npId, uint16_t eventId ) {
491
492     uint8_t msgBuf[ 8 ] = { LCS_OP_EVT_OFF };
493     msgBuf[ 1 ] = highByte( npId );
494     msgBuf[ 2 ] = lowByte( npId );
```

```
495     msgBuf[ 3 ] = highByte( eventId );
496     msgBuf[ 4 ] = lowByte( eventId );
497 
498     if ( nodeId( npId ) == nodeMap.nodeId ) {
499 
500         handleMsgEvent( msgBuf );
501         return( ALL_OK );
502     }
503     else return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
504 }
505 
506 uint8_t sendEvent( uint16_t npId, uint16_t eventId, uint16_t arg ) {
507 
508     uint8_t msgBuf[ 8 ] = { LCS_OP_EVT };
509     msgBuf[ 1 ] = highByte( npId );
510     msgBuf[ 2 ] = lowByte( npId );
511     msgBuf[ 3 ] = highByte( eventId );
512     msgBuf[ 4 ] = lowByte( eventId );
513     msgBuf[ 5 ] = highByte( arg );
514     msgBuf[ 6 ] = lowByte( arg );
515 
516     if ( nodeId( npId ) == nodeMap.nodeId ) {
517 
518         handleMsgEvent( msgBuf );
519         return( ALL_OK );
520     }
521     else return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
522 }
523 
524 uint8_t sendTrackOn( ) {
525 
526     uint8_t msgBuf[ 8 ] = { LCS_OP_TON };
527     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_HIGH ));
528 }
529 
530 uint8_t sendTrackOff( ) {
531 
532     uint8_t msgBuf[ 8 ] = { LCS_OP_TOF };
533     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_HIGH ));
534 }
535 
536 uint8_t sendEstop( ) {
537 
538     uint8_t msgBuf[ 8 ] = { LCS_OP_ESTP };
539     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_VERY_HIGH ));
540 }
541 
542 uint8_t sendReqLoc( uint16_t locAdr, uint8_t flags ) {
543 
544     uint8_t msgBuf[ 8 ] = { LCS_OP_REQ_LOC };
545     msgBuf[ 1 ] = highByte( locAdr );
546     msgBuf[ 2 ] = lowByte( locAdr );
547     msgBuf[ 3 ] = flags;
548     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
549 }
550 
551 uint8_t sendRelLoc( uint8_t sId ) {
552 
553     uint8_t msgBuf[ 8 ] = { LCS_OP_REL_LOC };
554     msgBuf[ 1 ] = sId;
555     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
556 }
557 
558 uint8_t sendRepLoc( uint8_t sId, uint16_t locAdr, uint8_t spDir, uint8_t fn1, uint8_t fn2, uint8_t fn3  ) {
559 
560     uint8_t msgBuf[ 8 ] = { LCS_OP_REP_LOC };
561     msgBuf[ 1 ] = sId;
562     msgBuf[ 2 ] = highByte( locAdr );
563     msgBuf[ 3 ] = lowByte( locAdr );
564     msgBuf[ 4 ] = spDir;
565     msgBuf[ 5 ] = fn1;
566     msgBuf[ 6 ] = fn2;
567     msgBuf[ 7 ] = fn3;
568     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
569 }
570 
571 uint8_t sendLocConsist( uint8_t sId, uint8_t consId, uint8_t flags ) {
572 
573     uint8_t msgBuf[ 8 ] = { LCS_OP_SET_LCON };
574     msgBuf[ 1 ] = sId;
575     msgBuf[ 2 ] = consId;
576     msgBuf[ 3 ] = flags;
577     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
578 }
579 
580 uint8_t sendQueryLoc( uint8_t sId ) {
581 
582     uint8_t msgBuf[ 8 ] = { LCS_OP_QRY_LOC };
583     msgBuf[ 1 ] = sId;
584     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
585 }
586 
587 uint8_t sendKeepLoc( uint8_t sId ) {
588 
589     uint8_t msgBuf[ 8 ] = { LCS_OP_KEEP_LOC };
590     msgBuf[ 1 ] = sId;
591     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
592 }
593 
```

```
uint8_t sendSetLocSpDir( uint8_t sId, uint8_t spDir ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_SET_LSPD };
    msgBuf[ 1 ] = sId;
    msgBuf[ 2 ] = spDir;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendSetLocMode( uint8_t sId, uint8_t mode ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_SET_LMOD };
    msgBuf[ 1 ] = sId;
    msgBuf[ 2 ] = mode;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendSetLocFuncOn( uint8_t sId, uint8_t fNum ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_LOC_FON };
    msgBuf[ 1 ] = sId;
    msgBuf[ 2 ] = fNum;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendSetLocFuncOff( uint8_t sId, uint8_t fNum ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_LOC_FOF };
    msgBuf[ 1 ] = sId;
    msgBuf[ 2 ] = fNum;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendSetLocFgroup( uint8_t sId, uint8_t fGroup, uint8_t data ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_LOC_FGRP };
    msgBuf[ 1 ] = sId;
    msgBuf[ 2 ] = fGroup;
    msgBuf[ 3 ] = data;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendSetLocCvMain( uint8_t sId, uint16_t cvId, uint8_t mode, uint8_t val ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_SET_CVM };
    msgBuf[ 1 ] = sId;
    msgBuf[ 2 ] = highByte( cvId );
    msgBuf[ 3 ] = lowByte( cvId );
    msgBuf[ 4 ] = mode;
    msgBuf[ 5 ] = val;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendSetLocCvProg( uint16_t cvId, uint8_t mode, uint8_t val ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_SET_CVS };
    msgBuf[ 1 ] = highByte( cvId );
    msgBuf[ 2 ] = lowByte( cvId );
    msgBuf[ 3 ] = mode;
    msgBuf[ 4 ] = val;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendReqLocCvProg( uint16_t cvId, uint8_t mode ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_REQ_CVS };
    msgBuf[ 1 ] = highByte( cvId );
    msgBuf[ 2 ] = lowByte( cvId );
    msgBuf[ 3 ] = mode;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendRepLocCvProg( uint16_t cvId, uint8_t val ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_REP_CVS };
    msgBuf[ 1 ] = highByte( cvId );
    msgBuf[ 2 ] = lowByte( cvId );
    msgBuf[ 3 ] = val;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendSetBacc( uint16_t accAdr, uint8_t flags ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_BACC };
    msgBuf[ 1 ] = highByte( accAdr );
    msgBuf[ 2 ] = lowByte( accAdr );
    msgBuf[ 3 ] = flags;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendSetEacc( uint16_t accAdr, uint8_t val ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_EACC };
    msgBuf[ 1 ] = highByte( accAdr );
    msgBuf[ 2 ] = lowByte( accAdr );
    msgBuf[ 3 ] = val;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendDccPacket( uint8_t arg1, uint8_t arg2, uint8_t arg3 ) {
```

```
693
694     uint8_t msgBuf[ 8 ] = { LCS_OP_SEND_DCC3 };
695     msgBuf[ 1 ] = arg1;
696     msgBuf[ 2 ] = arg2;
697     msgBuf[ 3 ] = arg3;
698     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
699 }
700
701 uint8_t sendDccPacket( uint8_t arg1, uint8_t arg2, uint8_t arg3, uint8_t arg4 ) {
702
703     uint8_t msgBuf[ 8 ] = { LCS_OP_SEND_DCC4 };
704     msgBuf[ 1 ] = arg1;
705     msgBuf[ 2 ] = arg2;
706     msgBuf[ 3 ] = arg3;
707     msgBuf[ 4 ] = arg4;
708     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
709 }
710
711 uint8_t sendDccPacket( uint8_t arg1, uint8_t arg2, uint8_t arg3, uint8_t arg4, uint8_t arg5 ) {
712
713     uint8_t msgBuf[ 8 ] = { LCS_OP_SEND_DCC5 };
714     msgBuf[ 1 ] = arg1;
715     msgBuf[ 2 ] = arg2;
716     msgBuf[ 3 ] = arg3;
717     msgBuf[ 4 ] = arg4;
718     msgBuf[ 5 ] = arg5;
719     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
720 }
721
722 uint8_t sendDccPacket( uint8_t arg1, uint8_t arg2, uint8_t arg3, uint8_t arg4, uint8_t arg5, uint8_t arg6 ) {
723
724     uint8_t msgBuf[ 8 ] = { LCS_OP_SEND_DCC6 };
725     msgBuf[ 1 ] = arg1;
726     msgBuf[ 2 ] = arg2;
727     msgBuf[ 3 ] = arg3;
728     msgBuf[ 4 ] = arg4;
729     msgBuf[ 5 ] = arg5;
730     msgBuf[ 6 ] = arg6;
731     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
732 }
733
734 uint8_t sendDccAck( ) {
735
736     uint8_t msgBuf[ 8 ] = { LCS_OP_DCC_ACK };
737     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
738 }
739
740 uint8_t sendDccErr( uint8_t errCode, uint8_t arg1, uint8_t arg2 ) {
741
742     uint8_t msgBuf[ 8 ] = { LCS_OP_DCC_ERR };
743     msgBuf[ 1 ] = errCode;
744     msgBuf[ 2 ] = arg1;
745     msgBuf[ 3 ] = arg2;
746     return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
747 }
748
749 }; // namespace LCS
```

```
//--------------------------------------------------------------------------------------------------------------
//
// Layout Control System - node access routines.
//
//--------------------------------------------------------------------------------------------------------------
// The file contains the LCS runtime routines that implement node access. There are three routines that allow
// to manipulate node and port data as well as issue requests to a node or port. The key are the node/port ID
// and the item number. The "npId" will indicate which node and port the call refers to. The node portion is
// typically our own node Id, the port Id refers to a ports on the node, with a port Id of zero referring to
// the node itself. Any node can access another node. In this case request comes via a message and the
// message handler will call the local routines in this file.
//
//--------------------------------------------------------------------------------------------------------------
//
// LCS - Core Library
// Copyright (C) 2021 - 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 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/>.
//
//--------------------------------------------------------------------------------------------------------------
#include "LcsRuntimeLib.h"
#include "LcsRtLibInt.h"

//--------------------------------------------------------------------------------------------------------------
// External declaration to global structures defined in "LcsRtSetup".
//
//--------------------------------------------------------------------------------------------------------------
namespace LCS {

    extern uint16_t                debugMask;
    extern LcsCdcMap               cdcMap;
    extern LcsNodeMap              nodeMap;
    extern LcsNodeData             nodeData;
    extern LcsPortMap              portMap;
    extern LcsCallbackMap          callbackMap;
};

//--------------------------------------------------------------------------------------------------------------
// The LcsCoreLib implementation file local declarations and routines.
//
//--------------------------------------------------------------------------------------------------------------
namespace {

    using namespace LCS;

    //----------------------------------------------------------------------------------------------------------
    // The node or port name cannot be set with a single LCS message. We will store the parts in this
    // temporary buffer and set the name when all parts are received.
    //
    //----------------------------------------------------------------------------------------------------------
    char tempName[ MAX_NODE_NAME_SIZE + 1 ] = { 0 };

    //----------------------------------------------------------------------------------------------------------
    // Utility routines.
    //
    //----------------------------------------------------------------------------------------------------------
    bool isInRangeU( uint16_t val, uint16_t lower, uint16_t upper ) {

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

    uint8_t lowByte( uint16_t arg ) {

        return( arg & 0xFF );
    }

    uint8_t highByte( uint16_t arg ) {

        return( arg >> 8 );
    }

     uint16_t nodeId( uint16_t arg ) {

        return( arg >> 4 );
    }

    uint16_t portId( uint16_t arg ) {

        return( arg & 0xF);
    }

    //----------------------------------------------------------------------------------------------------------
    // "readAttrMem" gets a value from the node or port attribute map in MEM. As an internal function, we
    // expect a valid block and item argument. The "block" argument will refer to the node and port data
    // attributes. Block 0 is the node, all others the port.
    //
    //----------------------------------------------------------------------------------------------------------
    uint8_t readAttrMem( uint8_t block, uint8_t item, uint16_t *arg ) {

        *arg = nodeData.map[ block ][ item - IR_ATTR_MEM_RANGE_START ];
        return ( LCS::ALL_OK );
```

```
99      }
100
101     //-----------------------------------------------------------------------------------------------------------
102     // "writeAttrMem" stores a value to a node or port attribute map in MEM. As an internal function, we
103     // expect a valid block and item argument. The "block" argument will refer to the node and port data
104     // attributes. Block 0 is the node, all others the port.
105     //
106     //-----------------------------------------------------------------------------------------------------------
107     uint8_t writeAttrMem( uint8_t block, uint8_t item, uint16_t arg ) {
108
109         nodeData.map[ block ][ item - IR_ATTR_MEM_RANGE_START ] = arg;
110         return ( LCS::ALL_OK );
111     }
112
113     //-----------------------------------------------------------------------------------------------------------
114     // "readAttrNvm" gets an attribute from the NVM storage. We read the value from NVM, store it in the MEM
115     // counterpart and then return it. For the NVM access, the byte offset into the storage needs to be
116     // computed. As an internal function, we expect a valid block and item argument.
117     //
118     //-----------------------------------------------------------------------------------------------------------
119     uint8_t readAttrNvm( uint8_t block, uint8_t item, uint16_t *arg ) {
120
121         if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_ATTRIBUTES )) {
122
123             printf( "readAttrNvm: block: 0x%x, item: %d\n", block, item );
124         }
125
126         uint16_t index  = item - IR_ATTR_NVM_RANGE_START;
127         uint16_t ofs    = NVM_NODE_DATA_START + (( block * MAX_ATTR_MAP_ENTRIES ) + index  ) * sizeof( uint16_t );
128
129         printf( "Ofs: 0x%x\n", ofs );
130
131         uint8_t rStat = rtNvmGetWord( ofs, &nodeData.map[ block ][ index ] );
132
133         printf( "rStat: 0x%x\n", rStat );
134
135         *arg = (( rStat == ALL_OK ) ? nodeData.map[ block ][ index ] : 0 );
136         return ( rStat );
137     }
138
139     //-----------------------------------------------------------------------------------------------------------
140     // "writeAttrNvm" stores an attribute to the NVM storage. We first update the corresponding MEM attribute
141     // and then write the value to NVM storage. For the NVM access, the byte offset into the storage needs to
142     // be computed. As an internal function, we expect a valid block and item argument.
143     //
144     //-----------------------------------------------------------------------------------------------------------
145     uint8_t writeAttrNvm( uint8_t block, uint8_t item, uint16_t arg ) {
146
147         if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_ATTRIBUTES )) {
148
149             printf( "readAttrNvm: block: 0x%x, item: %d\n", block, item );
150         }
151
152         uint16_t index  = item - IR_ATTR_NVM_RANGE_START;
153         uint16_t ofs    = NVM_NODE_DATA_START + (( block * MAX_ATTR_MAP_ENTRIES ) + index  ) * sizeof( uint16_t );
154
155         printf( "Ofs: 0x%x\n", ofs );
156
157         nodeData.map[ block ][ index ] = arg;
158         return ( rtNvmPutWord( ofs, arg ));
159     }
160
161     //-----------------------------------------------------------------------------------------------------------
162     // User callback function invocation routine. Items 64 to 127 are user defined items. We will simply
163     // invoke a previously registered callback passing the arguments.
164     //
165     //-----------------------------------------------------------------------------------------------------------
166     uint8_t invokeUserItemCallback( uint8_t portId, uint8_t item, uint16_t *arg1, uint16_t *arg2 ) {
167
168         if ( callbackMap.reqCallback != nullptr ) {
169
170             return ( callbackMap.reqCallback( portId, item, arg1, arg2 ));
171         }
172         else return( ERR_INVALID_ITEM_ID );
173     }
174
175 } // namespace
176
177
178 //-----------------------------------------------------------------------------------------------------------
179 // The LCS name space routines declared in this file.
180 //
181 //-----------------------------------------------------------------------------------------------------------
182 namespace LCS {
183
184 //-----------------------------------------------------------------------------------------------------------
185 // "nodeGet" will lookup a value from the node, port or the attribute data map. The "npId" argument contains
186 // the node and port Id. However, we will only use the portId portion, which represents the block index.
187 //
188 //-----------------------------------------------------------------------------------------------------------
189 uint8_t nodeGet( uint16_t npId, uint8_t item, uint16_t *arg1, uint16_t *arg2 ) {
190
191     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_ATTRIBUTES )) {
192
193         printf( "nodeGet: 0x%x:%d", npId, item  );
194         if ( arg1 != nullptr ) printf( ":%d", *arg1 ); else printf( "null" );
195         if ( arg2 != nullptr ) printf( ":%d", *arg2 ); else printf( "null" );
196     }
197
```

```
198     if ( isInRangeU( item, IR_ATTR_MEM_RANGE_START, IR_ATTR_MEM_RANGE_END )) {
199
200         return ( readAttrMem( portId( npId ), item, arg1 ));
201     }
202     else if ( isInRangeU( item, IR_ATTR_NVM_RANGE_START, IR_ATTR_NVM_RANGE_END )) {
203
204         return ( readAttrNvm( portId( npId ), item, arg1 ));
205     }
206     else {
207
208         switch ( item ) {
209
210             case ITEM_ID_DEBUG_MASK: {
211
212                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
213                 *arg1 = debugMask;
214                 return( ALL_OK );
215             }
216
217             case ITEM_ID_OPTIONS: {
218
219                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
220
221                 *arg1 = nodeMap.nodeOptions;
222                 return ( ALL_OK );
223             }
224
225             case ITEM_ID_FLAGS: {
226
227                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
228
229                 if ( portId( npId ) == 0 )   *arg1 = nodeMap.nodeFlags;
230                 else                        *arg1 = portMap.map[ portId( npId ) - 1 ].flags;
231
232                 return ( ALL_OK );
233             }
234
235             case ITEM_ID_VERSION: {
236
237                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
238
239                 *arg1 = nodeMap.nodeSwVersion;
240                 return ( ALL_OK );
241             }
242
243             case ITEM_ID_TYPE: {
244
245                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
246
247                 if ( portId( npId ) == 0 )   *arg1 = nodeMap. nodeType;
248                 else                        *arg1 = portMap.map[ portId( npId ) - 1 ].type;
249
250                 return ( ALL_OK );
251             }
252
253             case ITEM_ID_CONTROLLER_FAMILY: {
254
255                  if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
256
257                 *arg1 = nodeMap.head.controllerFamily;
258                 return ( ALL_OK );
259             }
260
261             case ITEM_ID_NODE_ID: {
262
263                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
264
265                 *arg1 = nodeMap.nodeId;
266                 return ( ALL_OK );
267             }
268
269             case ITEM_ID_NODE_UID: {
270
271                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
272                 if ( arg2 == nullptr ) return( ERR_INVALID_ATTR_ARG );
273
274                 *arg1 = nodeMap.nodeUID >> 16;
275                 *arg2 = nodeMap.nodeUID & 0xFFFF;
276                 return ( ALL_OK );
277             }
278
279             case ITEM_ID_RESTART_COUNT: {
280
281                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
282
283                 *arg1 = nodeMap.nodeRestartCnt;
284                 return ( ALL_OK );
285             }
286
287             case ITEM_ID_PORT_MAP_ENTRIES: {
288
289                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
290
291                 *arg1 = nodeMap.portMapEntries;
292                 return ( ALL_OK );
293             }
294
295             case ITEM_ID_EVENT_MAP_ENTRIES: {
296
```

```
297                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
298
299                 *arg1 = nodeMap.eventMapEntries;
300                 return ( ALL_OK );
301             }
302
303             case ITEM_ID_ATTR_MAP_ENTRIES: {
304
305                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
306
307                 *arg1 = MAX_ATTR_MAP_ENTRIES;
308                 return ( ALL_OK );
309             }
310
311             case ITEM_ID_GET_EVENT_MAP_ENTRY: {
312
313                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
314                 if ( arg2 == nullptr ) return( ERR_INVALID_ATTR_ARG );
315
316                 return ( getMemEmapEntry( *arg1, arg1, arg2 ));
317             }
318
319             case ITEM_ID_NAME_1: {
320
321                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
322                 if ( arg2 == nullptr ) return( ERR_INVALID_ATTR_ARG );
323
324                 if ( portId( npId ) == 0 ) {
325
326                     *arg1 = ((uint16_t) ( nodeMap.name[ 0 ] << 8  ) | nodeMap.name[ 1 ] );
327                     *arg2 = ((uint16_t) ( nodeMap.name[ 2 ] << 8  ) | nodeMap.name[ 3 ] );
328                 }
329                 else {
330
331                     LcsPortMapEntry *pEntry = &portMap.map[ portId( npId ) -1 ];
332
333                     *arg1 = ((uint16_t) ( pEntry -> name[ 0 ] << 8  ) | pEntry -> name[ 1 ] );
334                     *arg2 = ((uint16_t) ( pEntry -> name[ 2 ] << 8  ) | pEntry -> name[ 3 ] );
335                 }
336
337                 return ( ALL_OK );
338             }
339
340             case ITEM_ID_NAME_2: {
341
342                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
343                 if ( arg2 == nullptr ) return( ERR_INVALID_ATTR_ARG );
344
345                 if ( portId( npId ) == 0 ) {
346
347                     *arg1 = ((uint16_t) ( nodeMap.name[ 4 ] << 8  ) | nodeMap.name[ 5 ] );
348                     *arg2 = ((uint16_t) ( nodeMap.name[ 6 ] << 8  ) | nodeMap.name[ 7 ] );
349                 }
350                 else {
351
352                     LcsPortMapEntry *pEntry = &portMap.map[ portId( npId ) - 1 ];
353
354                     *arg1 = ((uint16_t) ( pEntry -> name[ 4 ] << 8  ) | pEntry -> name[ 5 ] );
355                     *arg2 = ((uint16_t) ( pEntry -> name[ 6 ] << 8  ) | pEntry -> name[ 7 ] );
356                 }
357
358                 return ( ALL_OK );
359             }
360
361             case ITEM_ID_NAME_3: {
362
363                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
364                 if ( arg2 == nullptr ) return( ERR_INVALID_ATTR_ARG );
365
366                 if ( portId( npId ) == 0 ) {
367
368                     *arg1 = ((uint16_t) ( nodeMap.name[ 8 ] << 8  ) | nodeMap.name[ 9 ] );
369                     *arg2 = ((uint16_t) ( nodeMap.name[ 10 ] << 8  ) | nodeMap.name[ 11 ] );
370                 }
371                 else {
372
373                     LcsPortMapEntry *pEntry = &portMap.map[ portId( npId ) - 1 ];
374
375                     *arg1 = ((uint16_t) ( pEntry -> name[ 8 ] << 8  ) | pEntry -> name[ 9 ] );
376                     *arg2 = ((uint16_t) ( pEntry -> name[ 10 ] << 8  ) | pEntry -> name[ 11 ] );
377                 }
378
379                 return ( ALL_OK );
380             }
381
382             case ITEM_ID_NAME_4: {
383
384                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
385                 if ( arg2 == nullptr ) return( ERR_INVALID_ATTR_ARG );
386
387                 if ( portId( npId ) == 0 ) {
388
389                     *arg1 = ((uint16_t) ( nodeMap.name[ 12 ] << 8  ) | nodeMap.name[ 13 ] );
390                     *arg2 = ((uint16_t) ( nodeMap.name[ 14 ] << 8  ) | nodeMap.name[ 15 ] );
391                 }
392                 else {
393
394                     LcsPortMapEntry *pEntry = &portMap.map[ portId( npId ) - 1 ];
395
```

```
396                     *arg1 = ((uint16_t) ( pEntry -> name[ 12 ] << 8  ) | pEntry -> name[ 13 ] );
397                     *arg2 = ((uint16_t) ( pEntry -> name[ 14 ] << 8  ) | pEntry -> name[ 15 ] );
398                 }
399 
400                 return ( ALL_OK );
401             }
402 
403             case ITEM_ID_NVM_PROTECTED_ACCESS: {
404 
405                 // ??? access to protected NVM data areas.
406                 // ??? arg 1 -> offset
407                 // ??? arg 2 -> value
408 
409                 return ( ALL_OK );
410 
411             } break;
412 
413             case ITEM_ID_EVENT_DELAY_TICKS: {
414 
415                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
416 
417                 *arg1 = portMap.map[ portId( npId ) - 1 ].eventDelayTime;
418                 return ( ALL_OK );
419             }
420 
421             default: return ( ERR_INVALID_ITEM_ID );
422         }
423     }
424 }
425 
426 //-------------------------------------------------------------------------------------------------------------
427 // "nodePut" will write a value to the node, port or the attribute data map. The "npId" argument contains
428 // the node and port Id. However, we will only use the portId portion.
429 //
430 //-------------------------------------------------------------------------------------------------------------
431 uint8_t nodePut( uint16_t npId, uint8_t item, uint16_t val1, uint16_t val2 ) {
432 
433     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_ATTRIBUTES )) {
434 
435         printf( "nodePut: 0x%x:%d:%d:%d\n", npId, item, val1, val2  );
436     }
437 
438    if ( isInRangeU( item, IR_ATTR_MEM_RANGE_START, IR_ATTR_MEM_RANGE_END )) {
439 
440         return ( writeAttrMem( portId( npId ), item, val1 ));
441     }
442     else if ( isInRangeU( item, IR_ATTR_NVM_RANGE_START, IR_ATTR_NVM_RANGE_END )) {
443 
444         return ( writeAttrNvm( portId( npId ), item, val1 ));
445     }
446     else {
447 
448         switch ( item ) {
449 
450             case ITEM_ID_DEBUG_MASK: {
451 
452                 if ( CDC::isConsoleConnected( ))    debugMask = val1 | DBG_CONFIG;
453                 else                                debugMask = val1 & ~ DBG_CONFIG;
454 
455                 return( ALL_OK );
456             }
457 
458             case ITEM_ID_VERSION: {
459 
460                 nodeMap.nodeSwVersion = val1;
461                 return( rtNvmPutWord( NVM_NODE_MAP_START + offsetof( LcsNodeMap, nodeSwVersion ), val1 ));
462             }
463 
464             case ITEM_ID_OPTIONS: {
465 
466                 nodeMap.nodeOptions = val1;
467                 return( rtNvmPutWord( NVM_NODE_MAP_START + offsetof( LcsNodeMap, nodeOptions ), val1 ));
468             }
469 
470             case ITEM_ID_FLAGS: {
471 
472                 nodeMap.nodeFlags = val1;
473                 return( rtNvmPutWord( NVM_NODE_MAP_START + offsetof( LcsNodeMap, nodeFlags ), val1 ));
474             }
475 
476             case ITEM_ID_NODE_ID: {
477 
478                 nodeMap.nodeId = val1;
479                 return( rtNvmPutWord( NVM_NODE_MAP_START + offsetof( LcsNodeMap, nodeId ), nodeMap.nodeId ));
480             }
481 
482             case ITEM_ID_TYPE: {
483 
484                 if ( portId( npId ) == 0 ) {
485 
486                     nodeMap.nodeType = lowByte( val1 );
487                     return( rtNvmPutWord( NVM_NODE_MAP_START + offsetof( LcsNodeMap, nodeType ), val1 ));
488                 }
489                 else {
490 
491                     portMap.map[ portId( npId ) - 1 ].type = lowByte( val1 );
492 
493                     uint16_t ofs =  NVM_PORT_MAP_START +
494                                     offsetof( LcsPortMap, map ) +
```

```
                                    (( portId( npId ) - 1 ) * sizeof( LcsPortMapEntry )) +
                                    offsetof( LcsPortMapEntry, type );

                    return ( rtNvmPutWord( ofs, portMap.map[ portId( npId ) - 1 ].type ));
                }
            }

            case ITEM_ID_EVENT_DELAY_TICKS: {

                if ( isInRangeU ( portId( npId ) - 1, 0, MAX_PORT_MAP_ENTRIES )) {

                    portMap.map[ portId( npId ) - 1 ].eventDelayTime = val1;

                    uint16_t ofs =  NVM_PORT_MAP_START +
                                    offsetof( LcsPortMap, map ) +
                                    (( portId( npId ) - 1 ) * sizeof( LcsPortMapEntry )) +
                                    offsetof( LcsPortMapEntry, eventDelayTime );

                    return ( rtNvmPutWord( ofs, val1 ));
                }
                else return( ERR_INVALID_PORT_ID );
            }

            case ITEM_ID_NAME_1: {

                tempName[ 0 ] = highByte( val1 );
                tempName[ 1 ] = lowByte( val1 );
                tempName[ 2 ] = highByte( val2 );
                tempName[ 3 ] = lowByte( val2 );

                if ( portId( npId ) == 0 ) {

                    memcpy((uint8_t *) nodeMap.name, (uint8_t *)tempName, MAX_NODE_NAME_SIZE );
                    return( rtNvmPutBytes(  NVM_NODE_MAP_START + offsetof( LcsNodeMap, name ),
                                            (uint8_t *)tempName,
                                            MAX_NODE_NAME_SIZE ));
                }
                else {

                    memcpy((uint8_t *) portMap.map[ portId( npId ) ].name, (uint8_t *)tempName, MAX_PORT_NAME_SIZE );
                    uint16_t ofs =  NVM_PORT_MAP_START +
                                    offsetof( LcsPortMap, map ) +
                                    (( portId( npId ) - 1 ) * sizeof( LcsPortMapEntry )) +
                                    offsetof( LcsPortMapEntry, name );
                    return( rtNvmPutBytes( ofs, (uint8_t *)tempName, MAX_PORT_NAME_SIZE ));
                }
            }

            case ITEM_ID_NAME_2: {

                tempName[ 4 ]   = highByte( val1 );
                tempName[ 5 ]   = lowByte( val1 );
                tempName[ 6 ]   = highByte( val2 );
                tempName[ 7 ]   = lowByte( val2 );
                return ( ALL_OK );
            }

            case ITEM_ID_NAME_3: {

                tempName[ 8 ]   = highByte( val1 );
                tempName[ 9 ]   = lowByte( val1 );
                tempName[ 10 ]  = highByte( val2 );
                tempName[ 11 ]  = lowByte( val2 );
                return ( ALL_OK );
            }

            case ITEM_ID_NAME_4: {

                memset( tempName, 0, MAX_NODE_NAME_SIZE );
                tempName[ 12 ]  = highByte( val1 );
                tempName[ 13 ]  = lowByte( val1 );
                tempName[ 14 ]  = highByte( val2 );
                tempName[ 15 ]  = lowByte( val2 );
                return ( ALL_OK );
            }

            case ITEM_ID_NVM_PROTECTED_ACCESS: {

                // ??? access to protected NVM data areas.
                // ??? arg 1 -> offset
                // ??? arg 2 -> value

                return ( ALL_OK );

            } break;

            default: return ( ERR_INVALID_ITEM_ID );
        }
    }
}

//-------------------------------------------------------------------------------------------------------
// "nodeReq" will carry out a node or port function. A function, represented by an item, can be a node or port
// defined item, or a user defined item. For the latter we will invoke the user defined callback, if any.
//
// ??? have an option to set the debug level ?
//-------------------------------------------------------------------------------------------------------
uint8_t nodeReq( uint16_t npId, uint8_t item, uint16_t *arg1, uint16_t *arg2 ) {

```

```
    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_ATTRIBUTES )) {

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

     if ( isInRangeU( item, IR_USER_RANGE_START, IR_USER_RANGE_END )) {

        return( invokeUserItemCallback( npId, item, arg1, arg2 ));
    }
    else {

        switch ( item ) {

            case ITEM_ID_RESET: {

                debugMask = *arg1;
                return ( resetNode( npId ));
            }

            case ITEM_ID_ADD_EVENT_MAP_ENTRY: {

                return ( addEvent( *arg1, *arg2 ));
            }

            case ITEM_ID_DEL_EVENT_MAP_ENTRY: {

                return ( removeEvent( *arg1, *arg2 ));
            }

            case ITEM_ID_SYNC: {

                // ??? options what to sync ? For now it is only the event map...
                // ??? use arg 1 as an option number... ?
                return( syncEventMap( ));
            }

            case ITEM_ID_NODE_ID: {

                if ( isInRangeU( *arg1, MIN_NODE_ID, MAX_NODE_ID )) {

                    nodeMap.nodeId = nodeId( *arg1 );
                    return( rtNvmPutBytes(  NVM_NODE_MAP_START +
                                            offsetof( LcsNodeMap, nodeId ),
                                            (uint8_t *) &nodeMap.nodeId,
                                            sizeof( uint16_t )));
                }
                else return ( ERR_INVALID_NODE_ID );
            }

            case ITEM_ID_ENABLE_EVENT_PROCESSING: {

                if ( *arg1 )    portMap.map[ portId( npId ) - 1 ].flags |= PF_PORT_EVENT_HANDLING_ENABLED;
                else            portMap.map[ portId( npId ) - 1 ].flags &= ~ PF_PORT_EVENT_HANDLING_ENABLED;

                return ( ALL_OK );
            }

            case ITEM_ID_SET_READY_LED: {

                return ( CDC::writeDio( cdcMap.cfg.READY_LED_PIN, *arg1 ));
            }

            case ITEM_ID_SET_ACTIVITY_LED: {

                return ( CDC::writeDio( cdcMap.cfg.ACTIVE_LED_PIN, *arg1 ));
            }

            case ITEM_ID_TOGGLE_READY_LED:  {

                return ( CDC::toggleDio( cdcMap.cfg.READY_LED_PIN ));
            }

            case ITEM_ID_TOGGLE_ACTIVITY_LED: {

                return ( CDC::toggleDio( cdcMap.cfg.ACTIVE_LED_PIN ));
            }

            default: return ( ERR_INVALID_ITEM_ID );
        }
    }
}

} // namespace LCS
```

```
//-------------------------------------------------------------------------------------------------------------
//
// Layout Control System - implementation file.
//
//-------------------------------------------------------------------------------------------------------------
// The file contains the part of the LCS Runtime Library that implements the node event handling. At the
// heart of LCS is the concept of events. Events are broadcasted by a node and any other node that is
// interested in it registers a callback or this event. The runtime functions provide the management of the
// event map and the search routines.
//
// The event map can be found as a MEM and an NVM structure. During operations, the sorted MEM event map is
// the map to work with. Entries are sorted by eventId and as a secondary sort key the portId. New events
// can be added, old removed and the map can be searched. There is a SYNC function to write the contents
// of the MEM event map to the NV event map. The idea is that all changes are made to the MEM version and
// then written back in one swoop.
//
// On node start or reset, the NVM event map is read as part of the overall NVM read process. Since we only
// write a sorted version to the NVM event map, we can always assume a sorted NVM version, except when the
// eventMap high water mark is not valid. In this case we read entry by entry from the NVM and add it
// sorted to the MEM twin. The high water mark specifies the number of entires actually used.
//
//-------------------------------------------------------------------------------------------------------------
//
// LCS - Core Library
// Copyright (C) 2021 - 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 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/>.
//
//-------------------------------------------------------------------------------------------------------------
#include "LcsRuntimeLib.h"
#include "LcsRtLibInt.h"
#include <stdlib.h>

//-------------------------------------------------------------------------------------------------------------
// External declaration to global structures defined in "LcsRtSetup".
//
//-------------------------------------------------------------------------------------------------------------
namespace LCS {

    extern uint16_t             debugMask;
    extern LCS::LcsNodeMap      nodeMap;
    extern LCS::LcsEventMap     eventMap;
};

//-------------------------------------------------------------------------------------------------------------
// The LcsCoreLib implementation file local declarations and routines.
//
//-------------------------------------------------------------------------------------------------------------
namespace {

using namespace LCS;

//-------------------------------------------------------------------------------------------------------------
// Utility routines for number range check.
//
//-------------------------------------------------------------------------------------------------------------
bool isInRangeU( uint16_t val, uint16_t lower, uint16_t upper ) {

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

//-------------------------------------------------------------------------------------------------------------
// "compareEventEntry" is a little helper function to compare event and portId to the data in an eventMap
// entry.
//
//-------------------------------------------------------------------------------------------------------------
int compareEventEntry( LcsEventMapEntry *e1, uint16_t eventId2, uint16_t portId2 ) {

    if      ( e1 -> eventId < eventId2 )  return ( -1 );
    else if ( e1 -> eventId > eventId2 )  return ( 1 );
    else if ( e1 -> portId < portId2 )    return ( -1 );
    else if ( e1 -> portId > portId2 )    return ( 1 );
    else return ( 0 );
}

int compareEventEntry( const LcsEventMapEntry *arg1, const LcsEventMapEntry *arg2 ) {

    LcsEventMapEntry *e1 = (LcsEventMapEntry *)  arg1;
    LcsEventMapEntry *e2 = (LcsEventMapEntry *)  arg2;

    if      ( e1 -> eventId < e2 -> eventId )  return ( -1 );
    else if ( e1 -> eventId > e2 -> eventId )  return ( 1 );
    else if ( e1 -> portId  < e2 -> portId )   return ( -1 );
    else if ( e1 -> portId  > e2 -> portId )   return ( 1 );
    else return ( 0 );
}

//-------------------------------------------------------------------------------------------------------------
// "addToMemEventMap" adds an event / port combination to the MEM event map if not already there. Given
// there is still room in the table, the entry is added in sorted order.
//
```

```
99  //-------------------------------------------------------------------------------------------------------
100 uint8_t addToMemEventMap( uint16_t eventId, uint16_t portId ) {
101
102     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_EVENTS )) {
103
104         printf( "Add to MEM Event Map: %d : %d\n", eventId, portId );
105     }
106
107     if ( searchEvent( eventId, portId ) >= 0 )    return ( ALL_OK );
108     if ( nodeMap.eventMapHwm >= MAX_EVENT_MAP_ENTRIES )  return ( ERR_EVENT_MAP_FULL );
109
110     uint16_t index = nodeMap.eventMapHwm;
111
112     if ( nodeMap.eventMapHwm > 0 ) {
113
114         while (( index > 0 ) && ( compareEventEntry( &eventMap.map[ index - 1 ], eventId, portId ) > 0 )) {
115
116             eventMap.map[ index ] = eventMap.map[ index - 1 ];
117             index --;
118         }
119     }
120
121     eventMap.map[ index ].eventId = eventId;
122     eventMap.map[ index ].portId  = portId;
123     nodeMap.eventMapHwm++;
124
125     return ( ALL_OK );
126 }
127
128 //-------------------------------------------------------------------------------------------------------
129 // "removeFromMemEventMap" removes an entry from the memory event map. The sorted order is maintained.
130 //
131 //-------------------------------------------------------------------------------------------------------
132 uint8_t removeFromMemEventMap( uint16_t eventId, uint16_t portId ) {
133
134      if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_EVENTS )) {
135
136         printf( "Remove from MEM Event Map: %d : %d \n", eventId, portId );
137     }
138
139     int index = searchEvent( eventId, portId );
140
141     if ( index >= 0 ) {
142
143         nodeMap.eventMapHwm--;
144
145         for ( uint16_t i = index; i < nodeMap.eventMapHwm; i++ )
146             eventMap.map[ i ] = eventMap.map[ i + 1 ];
147     }
148
149     return ( ALL_OK );
150 }
151
152 } // namespace
153
154 //-------------------------------------------------------------------------------------------------------
155 // The LCS name space routines declared in this file.
156 //
157 //-------------------------------------------------------------------------------------------------------
158 namespace LCS {
159
160 //-------------------------------------------------------------------------------------------------------
161 // The "addEvent" routine will add an eventId/portId combination to the event map if not already there. If
162 // the portId parameter is a NIL_PORT_ID, the event is added to all portMap entries.
163 //
164 //-------------------------------------------------------------------------------------------------------
165 uint8_t addEvent( uint16_t eventId, uint16_t portId ) {
166
167     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_EVENTS )) {
168
169         printf( "Add Event: event: %d, port: %d\n", eventId, portId );
170     }
171
172     int rStat = ALL_OK;
173
174     if ( ! isInRangeU( eventId, MIN_EVENT_ID, MAX_EVENT_ID )) return ( ERR_INVALID_EVENT_ID );
175     if ( portId >  MAX_PORT_ID ) return ( ERR_INVALID_PORT_ID );
176
177     if ( portId == NIL_PORT_ID ) {
178
179         for ( uint8_t p = 1; p <= MAX_PORT_MAP_ENTRIES; p++ ) {
180
181             rStat = addToMemEventMap( eventId, p );
182             if ( rStat != ALL_OK ) break;
183         }
184     }
185     else {
186
187         rStat = addToMemEventMap( eventId, portId );
188     }
189
190     return ( rStat );
191 }
192
193 //-------------------------------------------------------------------------------------------------------
194 // The "removeEvent" routine will remove an event Id / port Id from the MEM event map. If the port ID is
195 // NIL_PORT_ID, all port map entries matching event Id are removed.
196 //
197 //-------------------------------------------------------------------------------------------------------
```

```
198 uint8_t removeEvent( uint16_t eventId, uint16_t portId ) {
199
200     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_EVENTS )) {
201
202         printf( "Remove Event: %d : %d\n", eventId, portId );
203     }
204
205     int rStat = ALL_OK;
206
207     if ( ! isInRangeU( eventId, MIN_EVENT_ID, MAX_EVENT_ID )) return ( ERR_INVALID_EVENT_ID );
208
209     if ( portId == NIL_PORT_ID ) {
210
211         for ( uint16_t p = 1; p <= MAX_PORT_MAP_ENTRIES; p++ ) {
212
213             rStat = removeFromMemEventMap( eventId, p );
214             if ( rStat != ALL_OK ) break;
215         }
216     }
217     else if ( isInRangeU( portId, MIN_PORT_ID, MAX_PORT_ID )) {
218
219         rStat = removeFromMemEventMap( eventId, portId );
220     }
221     else rStat = ERR_INVALID_PORT_ID;
222
223     return ( rStat );
224 }
225
226 //-------------------------------------------------------------------------------------------------------------
227 // The event search function performs a binary search of the event map using the event Id and the port Id.
228 // If the port Id is NIL, a matching entry with lowest portId is returned. All eventMap entries with the
229 // same eventId follow. If the entry cannot be found, a -1 is returned.
230 //
231 //-------------------------------------------------------------------------------------------------------------
232 int searchEvent( uint16_t eventId, uint16_t portId ) {
233
234     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_EVENTS )) {
235
236         printf( "Search Event: %d : %d", eventId, portId );
237     }
238
239     int   res   = -1;
240     int   low   = 0;
241     int   high  = nodeMap.eventMapHwm - 1;
242
243     if ( portId == NIL_PORT_ID ) {
244
245         while ( low <= high ) {
246
247             int mid = low + ( high - low + 1 ) / 2;
248
249             if      ( eventMap.map[ mid ].eventId < eventId ) low  = mid + 1;
250             else if ( eventMap.map[ mid ].eventId > eventId ) high = mid - 1;
251             else if ( eventMap.map[ mid ].eventId == eventId ) {
252
253                 res   = mid;
254                 high  = mid - 1;
255             }
256         }
257     }
258     else {
259
260         while ( low <= high ) {
261
262             int mid = low + ( high - low ) / 2;
263             int tst = compareEventEntry( &eventMap.map[ mid ], eventId, portId );
264
265             if      ( tst < 0 ) low   = mid + 1;
266             else if ( tst > 0 ) high  = mid - 1;
267             else {
268
269                 res = mid;
270                 break;
271             }
272         }
273     }
274
275     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_EVENTS )) printf( "-> %d\n", res );
276     return ( res );
277 }
278
279 //-------------------------------------------------------------------------------------------------------------
280 // "syncEventMap" will write back the sorted MEM event map. We only write up to the HWM mark, which points
281 // right after the last element in the sorted MEM event map. The idea is that all adds and removes are done
282 // on the MEM event map and a SYNC control call will flush the sorted MEM event map to NVM.
283 //
284 //-------------------------------------------------------------------------------------------------------------
285 uint8_t syncEventMap( ) {
286
287     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_EVENTS )) printf( "sync EventMap \n" );
288
289     uint8_t rStat =  rtNvmPutBytes( NVM_EVENT_MAP_START,
290                                     (uint8_t *) eventMap.map,
291                                     nodeMap.eventMapHwm * sizeof( LcsEventMapEntry ));
292
293     if ( rStat == ALL_OK ) {
294
295         uint32_t ofs = NVM_NODE_MAP_START + offsetof( LcsNodeMap, eventMapHwm );
296         rStat = rtNvmPutWord( ofs, nodeMap.eventMapHwm );
```

```
297     }
298
299     return ( rStat );
300 }
301
302 //-------------------------------------------------------------------------------------------------------
303 // "getMemEmapEntry" returns the eventId and portId pair from the MEM event map. It is used by the console
304 // command interface and also the LCS message handler to obtain that data. The index starts at 0.
305 //
306 //-------------------------------------------------------------------------------------------------------
307 uint8_t getMemEmapEntry( uint16_t index, uint16_t *evId, uint16_t *pId ) {
308
309     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_EVENTS )) {
310
311         printf( "Get Emap Entry: %d\n", index );
312     }
313
314     if ( index <  nodeMap.eventMapHwm ) {
315
316         *evId = eventMap.map[ index ].eventId;
317         *pId  = eventMap.map[ index ].portId;
318         return ( ALL_OK );
319     }
320     else return ( ERR_INVALID_EVENT_MAP_INDEX );
321 }
322
323 };
```

```
//-----------------------------------------------------------------------------------------------------------------
//
// LCS Runtime - command line interface.
//
//-----------------------------------------------------------------------------------------------------------------
// Based on the Raspberry Pi PICO controller USB interface, the LCS node has an option to accept commands and
// display data. This interface is used for manual node and extension board configuration as well as debug
// and troubleshooting. Most commands are sensitive to the node/port ID. If there is another node than our
// own node, specified with a zero node ID value, the commands is sent to the bus.
//
//-----------------------------------------------------------------------------------------------------------------
//
// LCS - Core Library
// Copyright (C) 2021 - 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 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/>.
//
//-----------------------------------------------------------------------------------------------------------------
#include "LcsRuntimeLib.h"
#include "LcsRtLibInt.h"

//-----------------------------------------------------------------------------------------------------------
// External declaration to global structures defined in "LcsRtSetup".
//
//-----------------------------------------------------------------------------------------------------------------
namespace LCS {

    extern uint16_t             debugMask;
    extern LcsCdcMap            cdcMap;
    extern LcsNodeMap           nodeMap;
    extern LcsNodeData          nodeData;
    extern LcsPortMap           portMap;
    extern LcsEventMap          eventMap;
    extern LcsCallbackMap       callbackMap;
    extern LcsTaskMap           taskMap;
    extern LcsPendingReqMap     pendingReqMap;
    extern LcsDrvFuncMap        drvFuncMap;
    extern LcsDrvMap            drvMap;
    extern LcsMsgBusCAN         *msgBus;
};

//-----------------------------------------------------------------------------------------------------------------
// Local declarations.
//
//-----------------------------------------------------------------------------------------------------------------
namespace {

using namespace LCS;

//-----------------------------------------------------------------------------------------------------------------
// The command line buffer.
//
//-----------------------------------------------------------------------------------------------------------------
char  commandBuf [ MAX_COMMAND_LINE_SIZE ];

//-----------------------------------------------------------------------------------------------------------------
// "dumpMemData" lists the memory data content of the storage area passed. The data is displayed in 16-bit
// quantities.  Because the PICO uses little-endian format, ASCII characters may appear reversed when
// interpreted directly.
//
//-----------------------------------------------------------------------------------------------------------------
void dumpMemData( uint16_t *area, uint16_t len, uint8_t itemsPerLine = 8, bool printAscii = false ) {

    uint16_t  index   = 0;
    uint16_t  limit   = ( len + 1 ) / 2;
    uint16_t  *ptr    = area;

    while ( index < limit ) {

        printf( "0x%08x: ", index * sizeof( uint16_t ));

        for ( uint16_t i = 0; i < itemsPerLine; i++ ) {

            if ( index + i < limit ) printf( "0x%04x ", ptr[ index + i ] );
        }

        if ( printAscii ) {

            if ( index + itemsPerLine >= limit ) {

                int tmp = index + itemsPerLine - limit;
                for ( int i = 0; i < tmp; i++ ) printf( "       " );
            };

            printf( "  " );

            for ( uint16_t i = 0; i < itemsPerLine; i++ ) {

                if ( index + i < limit ) {

                    if ( isprint( ptr[ index + i ] >> 8  )) printf( "%c", ptr[ index + i ] >> 8 );
```

```
                    else                                    printf( "." );

                    if ( isprint( ptr[ index + i ] & 0xff )) printf( "%c ", ptr[ index + i ] & 0xff );
                    else                                     printf( ". " );
                }
            }
        }


        index += itemsPerLine;
        printf( "\n" );
    }
}

//------------------------------------------------------------------------------------------------------------
// List the NVM storage data. The function receives the absolute byte offset within the NVM area and the
// length in bytes. The data is displayed in 16-bit quantities. Because the PICO uses little-endian format,
// ASCII characters may appear reversed when interpreted directly.
//
//------------------------------------------------------------------------------------------------------------
void dumpNvmData( uint32_t start, uint32_t len, uint32_t itemsPerLine = 8, bool printAscii = false ) {

    uint8_t     rStat = ALL_OK;
    uint32_t    limit = start + len;
    uint16_t    val   = 0;

    while ( start < limit ) {

        printf( "0x%08x: ", start );

        for ( uint16_t i = 0; i < itemsPerLine; i++ ) {

            uint32_t ofs = ( start + ( i * sizeof(uint16_t)));

            if ( ofs < limit ) {

                rStat = rtNvmGetWord( ofs, &val );
                if ( rStat == ALL_OK ) printf( "0x%04x ", val );
            }
        }

        if ( printAscii ) {

            if ( start + ( itemsPerLine * sizeof(uint16_t)) >= limit ) {

                int tmp = ( start + ( itemsPerLine * sizeof(uint16_t)) - limit ) / sizeof( uint16_t);
                for ( int i = 0; i < tmp; i++ ) printf( "       " );
            };

            printf( "  " );

            for ( uint16_t i = 0; i < itemsPerLine; i++ ) {

                uint32_t ofs = start + ( i * sizeof(uint16_t));

                if ( ofs < limit ) {

                    rStat = rtNvmGetWord( ofs, &val );
                    if ( rStat == ALL_OK ) {

                        if ( isprint( val >> 8  )) printf( "%c", val >> 8 );
                        else                       printf( "." );

                        if ( isprint( val & 0xff )) printf( "%c ", val & 0xFF );
                        else                        printf( ". " );
                    }
                }
            }
        }

        start = start + ( itemsPerLine * sizeof(uint16_t));
        printf( "\n" );
    }
}

//------------------------------------------------------------------------------------------------------------
// List extension board NVM storage data. We are passed the absolute offset into the NVM area and the
// length in bytes.
//
//------------------------------------------------------------------------------------------------------------
void dumpExtNvmData( uint8_t boardId, uint32_t start, uint32_t len, uint32_t itemsPerLine = 8 ) {

    uint8_t     rStat = ALL_OK;
    uint32_t    limit = start + len;
    uint16_t    val   = 0;

    while ( start < limit ) {

        printf( "0x%08x: ", start );

        for ( uint16_t i = 0; i < itemsPerLine; i++ ) {

            uint32_t ofs = ( start + ( i * sizeof(uint16_t)));

            if ( ofs < limit ) {

                rStat = extNvmGetWord( boardId, ofs, &val );
                if ( rStat == ALL_OK ) printf( "0x%04x ", val );
            }
```

```
198         }
199
200         for ( uint16_t i = 0; i < itemsPerLine; i++ ) {
201
202             uint32_t ofs = ( start + ( i * sizeof(uint16_t)));
203
204             if ( ofs < limit ) {
205
206                 rStat = extNvmGetWord( boardId, ofs, &val );
207                 if ( rStat == ALL_OK ) {
208
209                     if ( isprint( val >> 8  )) printf( "%c", val >> 8 );
210                     else                       printf( "." );
211
212                     if ( isprint( val & 0xff )) printf( "%c ", val & 0xFF );
213                     else                        printf( ". " );
214                 }
215             }
216         }
217
218         start = start + itemsPerLine * sizeof(uint16_t);
219         printf( "\n" );
220     }
221 }
222
223 //--------------------------------------------------------------------------------------------------------------
224 // Routines to list contents of the various memory areas. Right now, we just dump out hex data. It would be
225 // nice to show formatted data. Perhaps one day...
226 //
227 //--------------------------------------------------------------------------------------------------------------
228 void printSummary( ) {
229
230     printf( "LCS Node: \"" );
231     for ( uint8_t i = 0; i < MAX_NODE_NAME_SIZE; i++ ) {
232
233         if ( nodeMap.name[ i ] != 0 ) printf( "%c", nodeMap.name[ i ] );
234     }
235
236     printf( "\"\n" );
237     printf( "LCS Library Version: %d.%d\n", nodeMap.nodeSwVersion >> 8, nodeMap.nodeSwVersion & 0xFF );
238 }
239
240 void dumpMemNodeMap( ) {
241
242     printf( "MEM Node Map: \n\n" );
243     dumpMemData((uint16_t *) &nodeMap, sizeof( LcsNodeMap ), 8, true);
244     printf( "\n" );
245 }
246
247 void dumpMemCdcMap( ) {
248
249     printf( "MEM CDC Map: \n\n" );
250     // dumpMemData((uint16_t *) &nodeMap, sizeof( LcsNodeMap ));
251     printf( "\n" );
252 }
253
254 void dumpMemPortMap( ) {
255
256     printf( "MEM Port Map (Size: %d, Hwm: %d): \n\n", nodeMap.portMapEntries, nodeMap.portMapHwm );
257
258     for ( int i  = 0; i < MAX_PORT_MAP_ENTRIES; i++ ) {
259
260         printf( "Port %d:\n", i + 1 );
261         dumpMemData((uint16_t *) &portMap.map[ i ], sizeof( LcsPortMapEntry ), 8, true );
262         printf( "\n" );
263     }
264 }
265
266 void dumpMemNodeData( ) {
267
268     printf( "MEM Node Data: \n\n" );
269
270     for ( int i  = 0; i < MAX_NODE_DATA_BLOCKS; i++ ) {
271
272         printf( "Port %d:\n", i );
273         dumpMemData((uint16_t *) &nodeData.map[ i ], MAX_ATTR_MAP_ENTRIES * sizeof( uint16_t ));
274         printf( "\n" );
275     }
276 }
277
278 void dumpMemEventMap( ) {
279
280     printf( "MEM Event Map (Size: %d, Hwm: %d): \n\n", nodeMap.eventMapEntries, nodeMap.eventMapHwm );
281     dumpMemData((uint16_t *) &eventMap, sizeof( LcsEventMap ));
282     printf( "\n" );
283 }
284
285 void dumpMemPendingReqMap( ) {
286
287     printf( "MEM Pending Req Map: (Size: %d, Hwm: %d) \n\n", nodeMap.pendingMapEntries, nodeMap.pendingMapHwm );
288     dumpMemData((uint16_t *) &pendingReqMap, sizeof( LcsPendingReqMap ));
289     printf( "\n" );
290 }
291
292 void dumpMemCallbackMap( ) {
293
294     printf( "MEM Callback Map: \n\n" );
295     dumpMemData((uint16_t *) &callbackMap, sizeof( LcsCallbackMap ));
296     printf( "\n" );
```

```
297 }
298
299 void dumpMemTaskMap( ) {
300
301     printf( "MEM Task Map: (Size: %d, Hwm: %d) \n\n", nodeMap.taskMapEntries, nodeMap.taskMapHwm );
302     dumpMemData((uint16_t *) &taskMap, sizeof( LcsTaskMap ));
303     printf( "\n" );
304 }
305
306 void dumpMemDrvFuncMap( ) {
307
308     printf( "MEM Driver Function Map: (Size: %d) \n\n", nodeMap.drvFuncMapEntries );
309
310     for ( int i  = 0; i < MAX_DRV_TYPES; i++ ) {
311
312         LcsDrvFuncEntry *entry = &drvFuncMap.map[ i ];
313         printf( "%d: Type: %d, Func: %p\n", i, entry -> drvType, entry -> drvFunc );
314     }
315
316      printf( "\n" );
317 }
318
319 void dumpMemDrvMap( ) {
320
321     printf( "MEM Driver Map: (Size: %d) \n\n", nodeMap.drvMapEntries );
322
323     for ( int i  = 0; i < MAX_EXT_BOARDS; i++ ) {
324
325         LcsDrvEntry *entry = &drvMap.map[ i ];
326
327         printf( "Board %d: ( Flags: 0x%04x, LastErr: %d, Drv: %p\n",
328                 i, entry -> flags, entry -> lastErr, entry -> drvFunc );
329
330         dumpMemData(( uint16_t*) &drvMap.map[ i ].extBoard, sizeof( LcsDrvBoardDesc ), 8, true );
331         printf( "\n" );
332     }
333
334      printf( "\n" );
335 }
336
337 void dumpMemRuntimeArea( ) {
338
339     printf( "MEM Area Dump: \n\n" );
340     dumpMemNodeMap( );
341     dumpMemCdcMap( );
342     dumpMemPortMap( );
343     dumpMemEventMap( );
344     dumpMemPendingReqMap( );
345     dumpMemTaskMap( );
346     dumpMemCallbackMap( );
347     dumpMemDrvFuncMap( );
348     dumpMemDrvMap( );
349     printf( "\n" );
350 }
351
352 //-------------------------------------------------------------------------------------------------------------
353 // Routines to list contents of the various NVM areas. Right now, we just dump out hex data. It would be
354 // nice to show formatted data. Perhaps one day...
355 //
356 //-------------------------------------------------------------------------------------------------------------
357 void dumpNvmNodeMap( ) {
358
359     printf( "NVM Node Map Dump: \n\n" );
360     dumpNvmData( NVM_NODE_MAP_START, sizeof( LcsNodeMap ), 8, true );
361     printf( "\n" );
362 }
363
364 void dumpNvmCdcMap( ) {
365
366     printf( "MEM CDC Map Dump: \n\n" );
367     dumpNvmData( NVM_CDC_MAP_START, sizeof( CDC::CdcConfigDesc ));
368     printf( "\n" );
369 }
370
371 void dumpNvmPortMap( ) {
372
373     printf( "NVM Port Map Dump: \n\n" );
374
375     for ( int i  = 0; i < MAX_PORT_MAP_ENTRIES; i++ ) {
376
377         uint32_t ofs = NVM_PORT_MAP_START + ( i * sizeof( LcsPortMapEntry ));
378
379         printf( "Port %d, NVM ofs: 0x%04x \n", i + 1, ofs );
380         dumpNvmData( ofs, sizeof( LcsPortMapEntry ), 8, true );
381         printf( "\n" );
382     }
383
384     printf( "\n" );
385 }
386
387 void dumpNvmNodeData( ) {
388
389     printf( "NVM Port Map Dump: \n\n" );
390
391     for ( int i  = 0; i < MAX_NODE_DATA_BLOCKS; i++ ) {
392
393         uint32_t ofs = NVM_NODE_DATA_START + ( i * MAX_ATTR_MAP_ENTRIES  * sizeof( uint16_t ));
394
395         printf( "Node data block: %d, NVM ofs: 0x%04x \n", i, ofs );
```

```
396         dumpNvmData( ofs, MAX_ATTR_MAP_ENTRIES  * sizeof( uint16_t ));
397         printf( "\n" );
398     }
399
400     printf( "\n" );
401 }
402
403 void dumpNvmEventMap( ) {
404
405     printf( "NVM Node Event Dump: \n\n" );
406     dumpNvmData( NVM_EVENT_MAP_START, sizeof( LcsEventMap ));
407     printf( "\n" );
408 }
409
410 void dumpNvmRuntimeArea( ) {
411
412     printf( "NVM Runtime Area Dump: \n\n" );
413     dumpNvmData( 0, NVM_RUNTIME_AREA_SIZE , 8, true );
414     printf( "\n" );
415 }
416
417 void dumpNvmDrvData( uint16_t boardId ) {
418
419     if ( boardId >= MAX_EXT_BOARD_MAP_ENTRIES ) {
420
421         printf( "Invalid board ID\n" );
422         return;
423     }
424
425     if ( drvMap.map[ boardId ].flags & BF_EXT_BOARD_PRESENT ) {
426
427         printf( "NVM Driver Data( board: %d ): \n\n", boardId );
428         dumpExtNvmData( boardId, 0, sizeof( LcsDrvBoardDesc ));
429         printf( "\n" );
430     }
431     else printf( "No board found \n" );
432 }
433
434 void dumpNvmUserArea( ) {
435
436     printf( "NVM Area Dump: \n\n" );
437     dumpNvmData( NVM_USER_MAP_START, usrNvmGetSize( ), 8, true );
438     printf( "\n" );
439 }
440
441 //-----------------------------------------------------------------------------------------------------------
442 // Print memory structures in a formatted way. Note that not all memory structures are printed. Some of the
443 // maps contain dynamic data, which changed rapidly. There is no point in showing that kind of data.
444 //
445 //-----------------------------------------------------------------------------------------------------------
446 void printMemNodeMap( ) {
447
448     printf( "MEM Node Map: \n\n" );
449
450 /*
451     uint16_t        magicWord1                      = NVM_MWORD_1;
452     uint16_t        boardType                       = BT_NIL;
453     uint16_t        boardVersion                    = 0;
454     uint16_t        controllerFamily                = CF_FAM_RPICO;
455     uint16_t        nvmChipFamily                   = CF_FAM_MICROCHIP;
456     uint16_t        reservedArea[ 10 ]               = { 0 };
457     uint16_t        magicWord2                      = NVM_MWORD_2;
458
459     uint16_t        nodeState                       = NS_NIL;
460     uint16_t        nodeOptions                     = 0;
461     uint16_t        nodeFlags                       = 0;
462     uint16_t        nodeId                          = NIL_NODE_ID;
463     uint32_t        nodeUID                         = 0L;
464     uint16_t        nodeType                        = NIL_NODE_TYPE;
465     uint16_t        nodeSwVersion                   = 0;
466     uint16_t        nodeSwPatchLevel                = 0;
467     uint16_t        nodeRestartCnt                  = 0;
468     uint32_t        nodeSystemTime                  = 0;
469     uint16_t        nodeMapSize                     = sizeof( LcsNodeMap );
470     char            name[ MAX_NODE_NAME_SIZE ]      = { 0 };
471
472     uint16_t        nvmNodeMapOfs                   = NVM_NODE_MAP_START;
473     uint16_t        nvmCdcMapOfs                    = NVM_CDC_MAP_START;
474     uint16_t        nvmPortMapOfs                   = NVM_PORT_MAP_START;
475     uint16_t        nvmNodeDataOfs                  = NVM_NODE_DATA_START;
476     uint16_t        nvmEventMapOfs                  = NVM_EVENT_MAP_START;
477     uint16_t        nvmUserMapOfs                   = NVM_USER_MAP_START;
478     uint32_t        nvmMemSize                      = NVM_RUNTIME_AREA_SIZE;
479
480     uint16_t        portMapEntries                  = MAX_PORT_MAP_ENTRIES;
481     uint16_t        portMapHwm                      = 0;
482
483     uint16_t        eventMapEntries                 = MAX_EVENT_MAP_ENTRIES;
484     uint16_t        eventMapHwm                     = 0;
485
486     uint16_t        taskMapEntries                  = MAX_TASK_MAP_ENTRIES;
487     uint16_t        taskMapHwm                      = 0;
488
489     uint16_t        pendingMapEntries               = MAX_PENDING_REQ_MAP_ENTRIES;
490     uint16_t        pendingMapHwm                   = 0;
491
492     uint16_t        drvFuncMapEntries               = MAX_DRV_TYPES;
493     uint16_t        drvFuncMapHwm                   = 0;
494
```

```
    uint16_t        drvMapEntries                   = MAX_EXT_BOARD_MAP_ENTRIES;
    uint16_t        drvMapHwm                       = 0;

    // add the node data area ....
*/

    printf( "\n" );
}

void printMemCdcMap( ) {

    printf( "MEM CDC Map: \n\n" );

    // resort to CDC::printInfo command ?


    printf( "\n" );
}

void printMemPortMap( ) {

    printf( "MEM Port Map (Size: %d, Hwm: %d): \n\n", nodeMap.portMapEntries, nodeMap.portMapHwm );

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

        LcsPortMapEntry *ptr = &portMap.map[ i ];

        printf( "Port %02d: Type: %02d, Options: 0x%04x, Flags: 0x%04x\n",
                i + 1,
                ptr -> type,
                ptr -> options,
                ptr -> flags );

        /*
        uint16_t  eventNodeId                   = NIL_NODE_ID;
        uint16_t  eventId                       = NIL_EVENT_ID;
        uint16_t  eventValue                    = 0;
        uint16_t  eventAction                   = PEA_EVENT_IDLE;
        uint16_t  eventDelayTime                = 0;
        uint32_t  eventTimeStamp                = 0L;
        char      name[ MAX_PORT_NAME_SIZE  ]   = { 0 };

        // add the port data area ?

        */

        printf( "\n" );
    }
}

void printMemEventMap( ) {

    printf( "MEM Event Map (Size: %d, Hwm: %d): \n\n", nodeMap.eventMapEntries, nodeMap.eventMapHwm );

    // print my entries, hwm

    // print the entries up to the HWM, 4 in a row ?

    printf( "\n" );
}

void printMemDrvMap( ) {

    printf( "MEM Driver Map: (Size: %d) \n\n", nodeMap.drvMapEntries );

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

        LcsDrvEntry *entry = &drvMap.map[ i ];

        printf( "Board %d: ( Flags: 0x%04x, LastErr: %d, Drv: %p\n",
                i, entry -> flags, entry -> lastErr, entry -> drvFunc );

        // add to print the driver data area...

        printf( "\n" );
    }

     printf( "\n" );
}

//-------------------------------------------------------------------------------------------------------------
// "scanI2CBus" and "listDevicesI2C" are two routines that will list all chips found on the NVM and EXT bus.
//
//-------------------------------------------------------------------------------------------------------------
void scanI2CBus( uint8_t sclPin ) {

    uint8_t rStat     = 0;
    uint8_t i2cAdr    = 0;
    uint8_t nDevices  = 0;
    uint8_t buf       = 0;

    for ( i2cAdr = 1; i2cAdr < 127; i2cAdr++ ) {

        rStat = CDC::i2cRead( sclPin, i2cAdr, &buf, 1 );

        if ( rStat == 0 ) {

            printf( "I2C device found at i2cAdr 0x%x\n", i2cAdr );
            nDevices ++;
```

```
        }
    }

    if ( nDevices == 0 )  printf( "No I2C devices found\n" );
    else                  printf( "Scan done\n" );
}

void listDevicesI2C( ) {

    if ( cdcMap.cfg.NVM_I2C_SCL_PIN != CDC::UNDEFINED_PIN ) {

      printf( "Scanning NVM I2C Bus: scl:%d, sda: %d \n", cdcMap.cfg.NVM_I2C_SCL_PIN, cdcMap.cfg.NVM_I2C_SDA_PIN );
      scanI2CBus(  cdcMap.cfg.NVM_I2C_SCL_PIN );
      printf( "\n" );
    }

    if ( cdcMap.cfg.EXT_I2C_SCL_PIN != CDC::UNDEFINED_PIN ) {

      printf( "Scanning EXT I2C Bus: scl:%d, sda: %d \n", cdcMap.cfg.EXT_I2C_SCL_PIN, cdcMap.cfg.EXT_I2C_SDA_PIN );
      scanI2CBus(  cdcMap.cfg.EXT_I2C_SCL_PIN );
      printf( "\n" );
    }
}

//-------------------------------------------------------------------------------------------------------------
// Little helper functions.
//
//-------------------------------------------------------------------------------------------------------------
bool isInRangeU( uint16_t val, uint16_t lower, uint16_t upper ) {

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

uint16_t buildNpId( uint16_t nodeId, uint16_t portId ) {

    return(( nodeId << 4 ) | ( portId & 0xF ));
}

uint16_t nodeId( uint16_t npId ) {

    return( npId >> 4 );
}

uint16_t portId( uint16_t npId ) {

    return( npId & 0xF );
}

uint8_t lowByte( uint16_t arg ) {

    return( arg & 0xFF );
}

uint8_t highByte( uint16_t arg ) {

    return( arg >> 8 );
}

//-------------------------------------------------------------------------------------------------------------
// Helper routines for error status handling.
//
// ??? one day, combine all error strings in one routine and print them from there...
//-------------------------------------------------------------------------------------------------------------
void errorArgList( ) {

    printf( "Argument list error, use \"?\" )for help\n" );
}

void errorStatusMsg( char *msg, uint8_t ret ) {

    printf( "Error: %s ( %d )\n", msg, ret );
}

}; // namespace


//-------------------------------------------------------------------------------------------------------------
// Routines in LCS name space.
//
//-------------------------------------------------------------------------------------------------------------
namespace LCS {

//-------------------------------------------------------------------------------------------------------------
// "c" switches a node to CFG mode. For a local node command, we construct the LCS_OP_CFG message payload
// data and invoke the msg handler for switching the node mode. For any other node, we will just send a LCS
// message.
//
//    c [ npId ]
//
//    returns: none
//
//-------------------------------------------------------------------------------------------------------------
void switchToConfigCommand( char *s ) {

    int npId = NIL_NODE_ID;

    if ( sscanf( s, "%i", &npId ) < 1 ) return( errorArgList( ));

    uint16_t tmpNpId = (uint16_t) npId;
```

```
693
694     if (( npId == 0 ) || ( nodeId( tmpNpId ) == nodeMap.nodeId )) {
695
696         uint8_t msg[ 8 ] = { LCS_OP_CFG };
697         handleMsgLcsMgt( msg );
698     }
699     else {
700
701         uint8_t ret = sendCfg( tmpNpId );
702         if ( ret != ALL_OK ) errorStatusMsg((char *) "Remote Node send error", ret );
703     }
704 }
705
706 //-----------------------------------------------------------------------------------------------------------
707 // "o" switches the nodes to OPS mode. For a local node command, we construct the LCS_OP_OPS message payload
708 // data and invoke the msg handler for switching the node mode. For any other node, we will just send a LCS
709 // message.
710 //
711 //    o [ npId ]
712 //
713 //-----------------------------------------------------------------------------------------------------------
714 void switchToOperationsCommand( char *s ) {
715
716     int npId = NIL_NODE_ID;
717
718     if ( sscanf( s, "%i", &npId ) < 1 ) return( errorArgList( ));
719
720     uint16_t tmpNpId = (uint16_t) npId;
721
722     if (( npId == 0 ) || ( nodeId( tmpNpId ) == nodeMap.nodeId )) {
723
724         uint8_t msg[ 8 ] = { LCS_OP_OPS };
725         handleMsgLcsMgt( msg );
726     }
727     else {
728
729         uint8_t ret = sendOps( tmpNpId );
730         if ( ret != ALL_OK ) errorStatusMsg((char *) "Remote Node send error", ret );
731     }
732 }
733
734 //-----------------------------------------------------------------------------------------------------------
735 // "a" adds an eventId / portId to the event map. If the portId is omitted, every port of the node will be
736 // registered for the event. For a non-local npId we will send a message.
737 //
738 //      a npId eventId [ portId ]
739 //
740 //      npId      - the node and port Id for which the event is added.
741 //      eventId   - the eventId.
742 //      portId    - the port number.
743 //
744 //-----------------------------------------------------------------------------------------------------------
745 void enterEventCommand( char *s ) {
746
747     int  npId      = NIL_NODE_ID;
748     int  eventId   = NIL_EVENT_ID;
749     int  portId    = NIL_PORT_ID;
750
751     if ( sscanf( s, "%i %i %i ", &npId, &eventId, &portId ) < 2 ) return( errorArgList( ));
752
753     uint16_t tmpNpId    = (uint16_t) npId;
754     uint16_t tmpEvent   = (uint16_t) eventId;
755     uint16_t tmpPort    = (uint16_t) portId;
756
757    if (( tmpNpId == 0 ) || ( nodeId( tmpNpId ) == nodeMap.nodeId )) {
758
759         uint8_t ret = nodeReq((uint16_t) tmpNpId, ITEM_ID_ADD_EVENT_MAP_ENTRY, &tmpEvent, &tmpPort );
760        if ( ret != ALL_OK ) errorStatusMsg((char *) "Node enter event error", ret );
761     }
762     else {
763
764         uint8_t ret = sendReqNode( nodeId( tmpNpId ), ITEM_ID_ADD_EVENT_MAP_ENTRY, tmpEvent, tmpPort );
765         if ( ret != ALL_OK ) errorStatusMsg((char *) "Remote Node send error", ret );
766     }
767 }
768
769 //-----------------------------------------------------------------------------------------------------------
770 // "r"  removes a eventId / portId combination from the event map. If the portId is omitted, all eventMap
771 // entries with the eventId are removed.
772 //
773 //      r npId eventId [ portId ]
774 //
775 //      npId      - the node and port Id for which the event is added.
776 //      eventId   - the eventId.
777 //      portId    - the port number.
778 //
779 //-----------------------------------------------------------------------------------------------------------
780 void removeEventCommand( char *s ) {
781
782     int  npId      = NIL_NODE_ID;
783     int  eventId   = NIL_EVENT_ID;
784     int  portId    = NIL_PORT_ID;
785
786     if ( sscanf( s, "%i %i %i ", &npId, &eventId, &portId ) < 1 ) return( errorArgList( ));
787
788     uint16_t tmpNpId    = (uint16_t) npId;
789     uint16_t tmpEvent   = (uint16_t) eventId;
790     uint16_t tmpPort    = (uint16_t) portId;
791
```

```
792     if (( tmpNpId == 0 ) || ( nodeId( tmpNpId ) == nodeMap.nodeId )) {
793
794         int ret = nodeReq( tmpNpId, ITEM_ID_DEL_EVENT_MAP_ENTRY, &tmpEvent, &tmpPort );
795         if ( ret != ALL_OK ) errorStatusMsg((char *) "Node remove event error", ret );
796     }
797     else {
798
799         uint8_t ret = sendReqNode( tmpNpId, ITEM_ID_DEL_EVENT_MAP_ENTRY, tmpEvent, tmpPort );
800         if ( ret != ALL_OK ) errorStatusMsg((char *) "Remote Node send error", ret );
801     }
802 }
803
804 //-------------------------------------------------------------------------------------------------------------
805 // "f" searches the event map for the eventId / portId combination and returns the index if found. If the
806 // portId is omitted, the first event map entry with the matching eventId is returned. This is a local
807 // command and cannot be called from a remote node.
808 //
809 //      f eventId [ portId ]
810 //
811 //      eventId   - the eventId.
812 //      portId    - the port number.
813 //
814 //-------------------------------------------------------------------------------------------------------------
815 void findEventCommand( char *s ) {
816
817     int  eventId   = NIL_EVENT_ID;
818     int  portId    = NIL_PORT_ID;
819
820     if ( sscanf( s, "%i %i ", &eventId, &portId ) < 1 ) return( errorArgList( ));
821
822     uint16_t tmpEvent   = (uint16_t) eventId;
823     uint16_t tmpPort    = (uint16_t) portId;
824
825     int ret = searchEvent( tmpEvent, tmpPort );
826     printf( "Event map index: %d", ret );
827 }
828
829 //-------------------------------------------------------------------------------------------------------------
830 // "e" will send an event. We will broadcast a message and also simulates receiving an event on the local
831 // node. Sending to ourselves is also quite useful for debugging event callback handlers.
832 //
833 //    e mode npId eventId [ arg ]
834 //
835 //    mode      - 0 - ON, 1 - OFF, 2 - DATA
836 //    npId      - the sending node / port Id
837 //    eventId   - the event Id
838 //    arg       - optional data argument for the event.
839 //
840 //-------------------------------------------------------------------------------------------------------------
841 void sendEventCommand( char *s ) {
842
843     uint8_t msg[ 8 ]    = { };
844     int     npId        = NIL_NODE_ID;
845     int     eventId     = NIL_EVENT_ID;
846     int     mode        = 0;
847     int     arg         = 0;
848     int     len         = 0;
849     uint8_t ret         = ALL_OK;
850
851     len = sscanf( s, "%i %i %i %i", &mode, &npId, &eventId, &arg );
852
853     uint16_t tmpNpId    = (uint16_t) npId;
854     uint16_t tmpEvent   = (uint16_t) eventId;
855     uint16_t tmpArg     = (uint16_t) arg;
856
857     if ( len < 3 ) return( errorArgList( ));
858
859     msg[ 0 ] = 0;
860     msg[ 1 ] = highByte( tmpNpId );
861     msg[ 2 ] = lowByte( tmpNpId );
862     msg[ 3 ] = highByte( tmpEvent );
863     msg[ 4 ] = lowByte( tmpEvent );
864     msg[ 5 ] = highByte( tmpArg );
865     msg[ 6 ] = lowByte( tmpArg );
866     msg[ 7 ] = 0;
867
868     if ( mode == 0 ) {
869
870         msg[ 0 ] = LCS_OP_EVT_ON;
871         ret = sendEventOn( tmpNpId, tmpEvent );
872     }
873     else if ( mode == 0 ) {
874
875         msg[ 0 ] = LCS_OP_EVT_OFF;
876         ret = sendEventOn( tmpNpId, tmpEvent );
877     }
878     else if ( mode == 2 ) {
879
880         msg[ 0 ] = LCS_OP_EVT;
881         ret = sendEvent( tmpNpId, tmpEvent, tmpArg );
882     }
883
884     if ( ret != ALL_OK ) errorStatusMsg((char *) "Send event error", ret );
885 }
886
887 //-------------------------------------------------------------------------------------------------------------
888 // "g" handles the node/port attribute query command. If the node is our node, we call the local access
889 // routines. Otherwise we send a message.
890 //
```

```
//    <!g npId item [ val1 [ val2 ]]>
//
//    npId      - the node/port Id.
//    item      - the node item to query, the result items will be listed in HEX format.
//    val1      - the argument 1 on input.
//    val2      - the argument 2 on input.
//
//-------------------------------------------------------------------------------------------------------------
void getNodeCommand( char *s ) {

    int     npId    = 0;
    int     item    = 0;
    int     arg1    = 0;
    int     arg2    = 0;
    uint8_t ret     = ALL_OK;

    if ( sscanf(  s, "%i %i %i %i", &npId, &item, &arg1, &arg2 ) < 2 ) return( errorArgList( ));

    uint16_t tmpNpId    = (uint16_t) npId;
    uint8_t  tmpItem    = (uint8_t) item;
    uint16_t tmpArg1    = (uint16_t) arg1;
    uint16_t tmpArg2    = (uint16_t) arg2;

    if (( tmpNpId == 0 ) || ( nodeId( tmpNpId ) == nodeMap.nodeId )) {

        ret = nodeGet ( tmpNpId, tmpItem, &tmpArg1, &tmpArg2 );
        if ( ret != ALL_OK ) errorStatusMsg((char *) "Node GET error", ret );
        else printf( "Node: 0x%x, item: %d, arg1: 0x%x, arg2: 0x%x\n", tmpNpId, tmpItem, tmpArg1, tmpArg2 );
    }
    else {

        ret = sendGetNode( tmpNpId, tmpItem, tmpArg1, tmpArg2 );
         if ( ret != ALL_OK ) errorStatusMsg((char *) "Remote Node GET error", ret );
    }
}

//-------------------------------------------------------------------------------------------------------------
// "p" handles the node or port attribute value set command. If the node is out node, we call the local
// access routines. Otherwise we send a message.
//
//    <!p npId item [ val1 [ val2 ]]>
//
//    npId      - the node/port Id.
//    item      - the port item to control
//    val1      - the item value 1
//    val2      - the item value 2 ( optional )
//
//-------------------------------------------------------------------------------------------------------------
void putNodeCommand( char *s ) {

    int     npId    = 0;
    int     item    = 0;
    int     val1    = 0;
    int     val2    = 0;
    uint8_t ret     = ALL_OK;

    if ( sscanf(  s, "%i %i %i %i", &npId, &item, &val1, &val2 ) < 2 ) return( errorArgList( ));

    uint16_t tmpNpId    = (uint16_t) npId;
    uint8_t  tmpItem    = (uint8_t)  item;
    uint16_t tmpVal1    = (uint16_t) val1;
    uint16_t tmpVal2    = (uint16_t) val2;

    printf ( "val1: %d\n", val1 );

    if (( tmpNpId == 0 ) || ( nodeId( tmpNpId ) == nodeMap.nodeId )) {

        ret = nodePut( tmpNpId, tmpItem, tmpVal1, tmpVal2 );
        if ( ret != ALL_OK ) errorStatusMsg((char *) "Node PUT error", ret );
        else printf( "Node: 0x%x, item: %d, val1: 0x%x, val2: 0x%x\n", tmpNpId, tmpItem, tmpVal1, tmpVal2 );
    }
    else {

        ret = sendSetNode( tmpNpId, tmpItem, tmpVal1, tmpVal2 );
        if ( ret != ALL_OK ) errorStatusMsg((char *) "Remote Node PUT error", ret );
    }
}

//-------------------------------------------------------------------------------------------------------------
// "r" handles the node / port request command. If the node is out node, we call the local access routines.
// Otherwise we send a message.
//
//    r npId item [ val1 [ val2 ]]
//
//    npId      - the node/port Id.
//    item      - the port item to control
//    val1      - the item value 1
//    val2      - the item value 2 ( optional )
//
//-------------------------------------------------------------------------------------------------------------
void reqNodeCommand( char *s ) {

    int     npId    = 0;
    int     item    = 0;
    int     val1    = 0;
    int     val2    = 0;
    uint8_t ret     = ALL_OK;

    if ( sscanf(  s, "%i %i %i %i", &npId, &item, &val1, &val2 ) < 2 ) return( errorArgList( ));
```

```
990
991     uint16_t tmpNpId    = (uint16_t) npId;
992     uint8_t  tmpItem    = (uint8_t)  item;
993     uint16_t tmpVal1    = (uint16_t) val1;
994     uint16_t tmpVal2    = (uint16_t) val2;
995
996     if (( tmpNpId == 0 ) || ( nodeId( tmpNpId ) == nodeMap.nodeId )) {
997
998         ret = nodeReq( tmpNpId, tmpItem, &tmpVal1, &tmpVal2 );
999         if ( ret != ALL_OK ) errorStatusMsg((char *) "Node REQ error", ret );
1000        else printf( "Node: 0x%x, item: %d, val1: 0x%x, val2: 0x%x\n", tmpNpId, tmpItem, tmpVal1, tmpVal1 );
1001    }
1002    else {
1003
1004        ret = sendReqNode( tmpNpId, tmpItem, tmpVal1, tmpVal2 );
1005        if ( ret != ALL_OK ) errorStatusMsg((char *) "Remote Node REQ error", ret );
1006    }
1007}
1008
1009//-----------------------------------------------------------------------------------------------------------
1010// "B" broadcasts a LCS message. Mainly used for debugging purposes. Although most commands in the LCS
1011// console interface can also send messages to other nodes, not all messages are covered. This command sends
1012// any kind of message, even undefined ones.
1013//
1014//    B byte1 [ byte2 ... byte8 ]
1015//
1016//    byte1 .. byte8   - the packet data in hexadecimal
1017//
1018//-----------------------------------------------------------------------------------------------------------
1019void broadcastLcsMsgCommand( char *s ) {
1020
1021    int     inBuf[ 8 ]  = { 0 };
1022    uint8_t b[ 8 ]      = { 0 };
1023    uint8_t nBytes  = sscanf( s, "%i %i %i %i %i %i %i %i",
1024                        inBuf, inBuf + 1, inBuf + 2, inBuf + 3, inBuf + 4, inBuf + 5, inBuf + 6, inBuf + 7 );
1025
1026    if ( nBytes >= 1 && nBytes <= 8 ) {
1027
1028        for ( int i = 0; i < 8; i++ ) b[ i ] == (uint8_t) inBuf[ i ];
1029
1030        uint8_t ret = msgBus -> sendLcsMsg( b );
1031        if ( ret != ALL_OK ) errorStatusMsg((char *) "Can Bus send error", ret );
1032    }
1033    else errorArgList( );
1034}
1035
1036//-----------------------------------------------------------------------------------------------------------
1037// "G" sends a GET request to a driver. The commands will typically work on the MEM image of the driver data.
1038// We will use the same idea of item ranges for MEM and NVM, except that the NVM range will work only if the
1039// extension board is write-enabled.
1040//
1041//    G board item
1042//
1043//    board - the extension board the driver handles.
1044//    item  - the driver specific item which is the requested operation.
1045//
1046//-----------------------------------------------------------------------------------------------------------
1047void drvGetCommand( char *s ) {
1048
1049    int  boardId    = 0;
1050    int  item       = 0;
1051    int  arg        = 0;
1052    int  ret        = ALL_OK;
1053
1054    if ( sscanf( s, "%i %i", &boardId, &item ) < 2 ) return( errorArgList( ));
1055
1056    uint16_t tmpBoard   = (uint8_t) boardId;
1057    uint8_t  tmpItem    = (uint8_t)  item;
1058    uint16_t tmpArg     = (uint16_t) arg;
1059
1060    ret = drvGet( tmpBoard, tmpItem, &tmpArg );
1061
1062    if ( ret != ALL_OK ) errorStatusMsg((char *) "Driver GET error", ret );
1063    else printf( "Board: %d, item: %d, arg: 0x%x\n", tmpBoard, tmpItem, tmpArg );
1064}
1065
1066//-----------------------------------------------------------------------------------------------------------
1067// "P" sends a PUT request to a driver. The commands will typically work on the MEM image of the driver data.
1068// We will use the same idea of item ranges for MEM and NVM, except that the NVM range will work only if the
1069// extension board is write-enabled.
1070//
1071//    P board item arg
1072//
1073//    board - the extension board the driver handles.
1074//    item  - the driver specific item which is the requested operation.
1075//    arg   - the data argument to the driver.
1076//
1077//-----------------------------------------------------------------------------------------------------------
1078void drvPutCommand( char *s ) {
1079
1080    int  boardId    = 0;
1081    int  item       = 0;
1082    int  val        = 0;
1083    int  ret        = ALL_OK;
1084
1085    if ( sscanf( s, "%i %i %i", &boardId, &item, &val ) < 3 ) return( errorArgList( ));
1086
1087    uint16_t tmpBoard   = (uint8_t) boardId;
1088    uint8_t  tmpItem    = (uint8_t)  item;
```

```
1089    uint16_t tmpVal     = (uint16_t) val;
1090
1091    ret = drvPut( tmpBoard, tmpItem, tmpVal );
1092
1093    if ( ret != ALL_OK ) errorStatusMsg((char *) "Driver PUT error", ret );
1094    else printf( "Board: %d, item: %d, arg: 0x%x\n", tmpBoard, tmpItem, tmpVal );
1095}
1096
1097//---------------------------------------------------------------------------------------------------------------
1098// "R" sends a REQ request to a driver.
1099//
1100//    R board item arg1 [ arg 2 ]
1101//
1102//    board - the extension board the driver handles.
1103//    item  - the driver specific item which is the requested operation.
1104//    arg1  - the first argument to the driver.
1105//    arg2  - the optional second argument to the driver and also output from the driver.
1106//
1107//---------------------------------------------------------------------------------------------------------------
1108void drvReqCommand( char *s ) {
1109
1110    int     boardId  = 0;
1111    int     item     = 0;
1112    int     arg1     = 0;
1113    int     arg2     = 0;
1114    uint8_t ret      = ALL_OK;
1115
1116    if ( sscanf( s, "%i %i %i %i", &boardId, &item, &arg1, &arg2 ) < 2 ) return( errorArgList( ));
1117
1118    uint16_t tmpBoard   = (uint8_t) boardId;
1119    uint8_t  tmpItem    = (uint8_t)  item;
1120    uint16_t tmpArg1     = (uint16_t) arg1;
1121    uint16_t tmpArg2     = (uint16_t) arg2;
1122
1123    ret = drvReq( tmpBoard, tmpItem, &tmpArg1, &tmpArg2 );
1124
1125    if ( ret != ALL_OK ) errorStatusMsg((char *) "Driver REQ error", ret );
1126    else printf( "Board: %d, item: %d, arg1: 0x%x, arg2: 0x%x\n", tmpBoard, tmpItem, tmpArg1, tmpArg2 );
1127}
1128
1129//---------------------------------------------------------------------------------------------------------------
1130// "s" lists status information. The level argument specifies the what and the detail level.
1131//
1132//    s [ level ]
1133//
1134//    returns:  NONE.
1135//
1136//---------------------------------------------------------------------------------------------------------------
1137void listStatusCommand( char *s ) {
1138
1139    int level = 0;
1140
1141    if ( sscanf( s, " %i", &level ) > 0 ) {
1142
1143        switch ( level ) {
1144
1145            case 0:     printSummary( );            break;
1146
1147            case 1:     dumpMemNodeMap( );          break;
1148            case 2:     dumpNvmCdcMap( );           break;
1149            case 3:     dumpMemPortMap( );          break;
1150            case 4:     dumpMemNodeData( );         break;
1151            case 5:     dumpMemEventMap( );         break;
1152            case 6:     dumpMemPendingReqMap( );    break;
1153            case 7:     dumpMemTaskMap( );          break;
1154            case 8:     dumpMemCallbackMap( );      break;
1155            case 9:     dumpMemDrvFuncMap( );       break;
1156            case 10:    dumpMemDrvMap( );           break;
1157            case 11:    dumpMemRuntimeArea( );      break;
1158
1159            case 21:    dumpNvmNodeMap( );          break;
1160            case 22:    dumpNvmCdcMap( );           break;
1161            case 23:    dumpNvmPortMap( );          break;
1162            case 24:    dumpNvmNodeData( );         break;
1163            case 25:    dumpNvmEventMap( );         break;
1164            case 26:    dumpNvmRuntimeArea( );      break;
1165
1166            case 31:    printMemNodeMap( );         break;
1167            case 32:    printMemCdcMap( );          break;
1168            case 33:    printMemPortMap( );         break;
1169            case 35:    printMemEventMap( );        break;
1170            case 38:    printMemDrvMap( );          break;
1171
1172            case 40:    dumpNvmDrvData( 0 );        break;
1173            case 41:    dumpNvmDrvData( 1 );        break;
1174            case 42:    dumpNvmDrvData( 2 );        break;
1175            case 43:    dumpNvmDrvData( 3 );        break;
1176
1177            case 50:    listDevicesI2C( );          break;
1178
1179            default: printf( "Unknown help option, use '?' for help\n" );
1180        }
1181    }
1182    else printSummary( );
1183}
1184
1185//---------------------------------------------------------------------------------------------------------------
1186// "?" lists core library help information. We just list the available commands and a short description.
1187//
```

```
1188 //    ?
1189 //
1190 //-----------------------------------------------------------------------------------------------------------------
1191 void listCoreLibHelpCommand( ) {
1192
1193     printf( "\nCommands: \n" );
1194     printf( "c [ npId ] - enter config mode\n" );
1195     printf( "o [ npId ] - enter operations mode\n" );
1196
1197     printf( "a npId eventId [ npId ] - add an event to the event tab\n" );
1198     printf( "d npId eventId [ npId ] - remove an event from the event tab\n" );
1199     printf( "e npId eventId mode [ arg ] - simulate sending an event ( mode: 0 - ON, 1 - OFF, 2 - EVT\n" );
1200     printf( "f eventId [ npId ] - search an event on the event tab\n" );
1201
1202     printf( "g npId item - gets a node attribute\n" );
1203     printf( "p npId item val1 [ val2 ] - puts a node attribute\n" );
1204     printf( "r npId item val1 [ val2 ] - request a node function\n" );
1205
1206     printf( "B byte1 [ byte2 ... byte8 ] - broadcast a raw LCS message\n" );
1207
1208     printf( "G board item - send a GET request to an extension board n\n" );
1209     printf( "P board item arg - send a PUT request to an extension board n\n" );
1210     printf( "R board item [ arg1 [ arg2 ]] - send a REQ request to an extension board n\n" );
1211
1212     printf( "s [ level ] - list status, default is summary\n" );
1213     printf( "               " " -   0  - summary\n" );
1214     printf( "               " " -   1  - MEM Node Map\n" );
1215     printf( "               " " -   2  - MEM CDC Map\n" );
1216     printf( "               " " -   3  - MEM Port Map\n" );
1217     printf( "               " " -   4  - MEM Node Data\n" );
1218     printf( "               " " -   5  - MEM Event Map\n" );
1219     printf( "               " " -   6  - MEM Pending Request Map\n" );
1220     printf( "               " " -   7  - MEM Task Map\n" );
1221     printf( "               " " -   8  - MEM Callback Map\n" );
1222     printf( "               " " -   9  - MEM Driver Function Map\n" );
1223     printf( "               " " -  10  - MEM Driver Map\n" );
1224     printf( "               " " -  11  - MEM Runtime Area\n" );
1225
1226     printf( "               " " -  21  - NVM Node Map\n" );
1227     printf( "               " " -  22  - NVM CDC Map\n" );
1228     printf( "               " " -  23  - NVM Port Map\n" );
1229     printf( "               " " -  24  - NVM Node Data\n" );
1230     printf( "               " " -  25  - NVM Event Map\n" );
1231     printf( "               " " -  26  - NVM Runtime Area\n" );
1232
1233     printf( "               " " -  31  - MEM Node Map formatted\n" );
1234     printf( "               " " -  32  - MEM CDC Map formatted\n" );
1235     printf( "               " " -  33  - NVM Port Map formatted\n" );
1236     printf( "               " " -  35  - MEM Event Map formatted\n" );
1237     printf( "               " " -  38  - MEM Driver Map formatted\n" );
1238
1239     printf( "               " " -  40  - NVM Extension Board 0\n" );
1240     printf( "               " " -  41  - NVM Extension Board 1\n" );
1241     printf( "               " " -  42  - NVM Extension Board 2\n" );
1242     printf( "               " " -  43  - NVM Extension Board 3\n" );
1243
1244     printf( "               " " -  50  - I2C Devices\n" );
1245 }
1246
1247 //-----------------------------------------------------------------------------------------------------------------
1248 // "setupSerialCommand" initializes the serial interface. We use the PICO USB as console IO. The CDC lib
1249 // contains functions for reading and writing to the console.
1250 //
1251 //-----------------------------------------------------------------------------------------------------------------
1252 uint8_t setupSerialCommand( ) {
1253
1254     return ( CDC::configureConsoleIO( ));
1255 }
1256
1257 //-----------------------------------------------------------------------------------------------------------------
1258 // "handleSerialCommand" reads characters from the console. The command interpreter is a simple character
1259 // based input. Note that this routine is called as part of the runtime loop. Consequently, it cannot not
1260 // block for IO. The interface is designed in a way that it assembles the character input when there are
1261 // characters until a carriage return is received. The first character is the command. If it is not a
1262 // command we know and there is a callback, the callback gets his chance to handle the input string.
1263 // Since we are pretty basic on a character by character basis, we also add a bit of luxury and echo back
1264 // the  what was typed in as well as process the backspace character.
1265 //
1266 //-----------------------------------------------------------------------------------------------------------------
1267 uint8_t handleSerialCommand( ) {
1268
1269     char c;
1270
1271     while (( c = CDC::getConsoleChar( )) > 0 ) {
1272
1273         switch( c ) {
1274
1275             case '\r': {
1276
1277                 printf( "\n" );
1278
1279                 if ( strlen( commandBuf) > 0 ) {
1280
1281                     switch ( commandBuf[ 0 ] ) {
1282
1283                         case 'C': switchToConfigCommand( commandBuf + 1 );        break;
1284                         case 'O': switchToOperationsCommand( commandBuf + 1 );    break;
1285
1286                         case 'a': enterEventCommand( commandBuf + 1 );          break;
```

```
1287                        case 'd': removeEventCommand( commandBuf + 1 );         break;
1288                        case 'f': findEventCommand( commandBuf + 1 );           break;
1289                        case 'e': sendEventCommand( commandBuf + 1 );           break;
1290
1291                        case 'g': getNodeCommand( commandBuf + 1 );             break;
1292                        case 'p': putNodeCommand( commandBuf + 1 );             break;
1293                        case 'r': reqNodeCommand( commandBuf + 1 );             break;
1294
1295                        case 'B': broadcastLcsMsgCommand( commandBuf + 1 );     break;
1296
1297                        case 'G': drvGetCommand( commandBuf + 1 );              break;
1298                        case 'P': drvPutCommand( commandBuf + 1 );              break;
1299                        case 'R': drvReqCommand( commandBuf + 1 );              break;
1300
1301                        case 's': listStatusCommand( commandBuf + 1 );          break;
1302                        case '?': listCoreLibHelpCommand( );                    break;
1303
1304                        default: {
1305
1306                            if ( callbackMap.cmdLineCallback != nullptr ) {
1307
1308                                    callbackMap.cmdLineCallback( commandBuf );
1309                                }
1310                            else printf( "<Unknown command, use '?' for help>" );
1311                        }
1312                    }
1313                }
1314
1315                commandBuf[ 0 ] = '\0';
1316                printf( "->");
1317
1318            } break;
1319
1320            case '\b': {
1321
1322                printf( "\b \b" );
1323                if ( strlen( commandBuf ) > 0 ) commandBuf[ strlen( commandBuf ) - 1 ] = '\0';
1324
1325            } break;
1326
1327            default: {
1328
1329                printf( "%c", c );
1330                if ( strlen( commandBuf ) < MAX_COMMAND_LINE_SIZE ) strncat( commandBuf, &c, 1 );
1331            }
1332        }
1333    }
1334
1335    return( ALL_OK );
1336}
1337
1338}; // namespace LCS
```

```
//---------------------------------------------------------------------------------------------------------------
//
// Layout Control System - node access routines.
//
//---------------------------------------------------------------------------------------------------------------
// The file contains the part of the LCS Runtime that implements the GET, SET and REQ access for the driver
// that manages an extension board.
//
// ??? what else to explain ?
//---------------------------------------------------------------------------------------------------------------
//
// LCS - Core Library
// Copyright (C) 2021 - 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 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/>.
//
//---------------------------------------------------------------------------------------------------------------
#include "LcsRuntimeLib.h"
#include "LcsRtLibInt.h"

//---------------------------------------------------------------------------------------------------------------
// External declaration to global structures defined in "LcsRtSetup".
//
//---------------------------------------------------------------------------------------------------------------
namespace LCS {

    extern uint16_t         debugMask;
    extern LcsCdcMap        cdcMap;
    extern LcsNodeMap       nodeMap;
    extern LcsDrvMap        drvMap;
};

//---------------------------------------------------------------------------------------------------------------
// The LcsCoreLib implementation file local declarations and routines.
//
//---------------------------------------------------------------------------------------------------------------
namespace {

using namespace LCS;

//---------------------------------------------------------------------------------------------------------------
// Utility routines.
//
//---------------------------------------------------------------------------------------------------------------
bool isInRangeU( uint16_t val, uint16_t lower, uint16_t upper ) {

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

uint8_t lowByte( uint16_t arg ) {

    return( arg & 0xFF );
}

uint8_t highByte( uint16_t arg ) {

    return( arg >> 8 );
}

//---------------------------------------------------------------------------------------------------------------
// "buildDrvBoardDescArea" will create a default data area and initializes the extension board NVM with this
// data.
//
//---------------------------------------------------------------------------------------------------------------
uint8_t buildDrvBoardDescArea( uint8_t boardId ) {

    uint8_t         rStat;
    LcsDrvBoardDesc tmp;

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
        printf( "buildDrvBoardDescArea, boardId: %d\n", boardId );

    rStat = extNvmPutBytes( boardId, 0, (uint8_t *) &tmp, sizeof( tmp ));

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
        printf( "buildDrvBoardDescArea, stat: %d\n", rStat  );

    return( rStat );
}

} // namespace


//---------------------------------------------------------------------------------------------------------------
// The LCS name space routines declared in this file.
//
//---------------------------------------------------------------------------------------------------------------
namespace LCS {

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

```
99  //
100 //-------------------------------------------------------------------------------------------------------------
101 uint8_t drvInit( ) {
102
103     return ( ALL_OK );
104 }
105
106 //-------------------------------------------------------------------------------------------------------------
107 // "drvGet" returns a value from the driver data array.
108 //
109 //-------------------------------------------------------------------------------------------------------------
110 uint8_t drvGet( uint8_t boardId, uint8_t item, uint16_t *arg ) {
111
112     if ( boardId >= MAX_EXT_BOARD_MAP_ENTRIES )                 return ( ERR_INVALID_BOARD_ID );
113     if ( ! drvMap.map[ boardId ].flags & BF_EXT_BOARD_PRESENT ) return( ERR_INVALID_BOARD_ID );
114
115     if ( isInRangeU( item, IR_ATTR_MEM_RANGE_START, IR_ATTR_MEM_RANGE_END )) {
116
117         *arg = drvMap.map[ boardId ].extBoard.driverData[ item - IR_ATTR_MEM_RANGE_START ];
118         return( ALL_OK );
119     }
120     else if ( isInRangeU( item, IR_ATTR_NVM_RANGE_START, IR_ATTR_NVM_RANGE_END )) {
121
122         uint32_t ofs =  offsetof( LcsDrvBoardDesc, driverData ) +
123                         (( item - IR_ATTR_NVM_RANGE_START ) * sizeof( uint16_t ));
124
125        if ( extNvmGetWord( boardId, ofs, arg ) != ALL_OK ) return( ERR_DRV_GET_ERR );
126        return( ALL_OK );
127     }
128     else if ( item == ITEM_ID_BOARD_VERSION ) {
129
130         *arg = drvMap.map[ boardId ].extBoard.head.boardVersion;
131         return( ALL_OK );
132     }
133     else if ( item == ITEM_ID_TYPE ) {
134
135         *arg = drvMap.map[ boardId ].extBoard.head.boardType;
136         return( ALL_OK );
137     }
138     else return( ERR_INVALID_ITEM_ID );
139 }
140
141 //-------------------------------------------------------------------------------------------------------------
142 // "drvPut" sets a value in the driver data array. Note that this is during normal operations only the MEM
143 // portion. The NVM chip on the extension board is write disabled after initial configuration. When the
144 // jumper on the board is taken out, writing to the NVM is enabled. The PUT and GET routines can be called
145 // for a board that has no driver associated yet. This way, for example, the driver type and other initial
146 // data can be set.
147 //
148 //
149 // ??? should we restart the extension board after setting the driver type ?
150 //-------------------------------------------------------------------------------------------------------------
151 uint8_t drvPut(uint8_t boardId, uint8_t item, uint16_t arg ) {
152
153     if ( boardId >= MAX_EXT_BOARD_MAP_ENTRIES )                 return( ERR_INVALID_BOARD_ID );
154     if ( ! drvMap.map[ boardId ].flags & BF_EXT_BOARD_PRESENT ) return( ERR_INVALID_BOARD_ID );
155
156      if ( isInRangeU( item, IR_ATTR_MEM_RANGE_START, IR_ATTR_MEM_RANGE_END )) {
157
158         drvMap.map[ boardId ].extBoard.driverData[ item - IR_ATTR_MEM_RANGE_START ] = arg;
159         return( ALL_OK );
160     }
161     else if ( isInRangeU( item, IR_ATTR_NVM_RANGE_START, IR_ATTR_NVM_RANGE_END )) {
162
163         uint32_t ofs =  offsetof( LcsDrvBoardDesc, driverData ) +
164                         (( item - IR_ATTR_NVM_RANGE_START ) * sizeof( uint16_t ));
165
166        if ( extNvmPutWord( boardId, ofs, arg ) != ALL_OK ) return( ERR_DRV_PUT_ERR );
167        return( ALL_OK );
168     }
169     else if ( item == ITEM_ID_BOARD_VERSION ) {
170
171         uint8_t rStat = ALL_OK;
172
173         rStat = extNvmPutWord( boardId, offsetof( LcsDrvBoardDesc, head.boardVersion ), arg );
174         if ( rStat == ALL_OK ) drvMap.map[ boardId ].extBoard.head.boardVersion = arg;
175         return( rStat );
176     }
177     else if ( item == ITEM_ID_TYPE ) {
178
179         uint8_t rStat = ALL_OK;
180
181         rStat = extNvmPutWord( boardId, offsetof( LcsDrvBoardDesc, head.boardType ), arg );
182         if ( rStat == ALL_OK ) drvMap.map[ boardId ].extBoard.head.boardType = arg;
183         return( rStat );
184     }
185     else return( ERR_INVALID_ITEM_ID );
186 }
187
188 //-------------------------------------------------------------------------------------------------------------
189 // "drvReq" is the entry point to an extension board. For each extension board type there is driver function.
190 // This function is called when we access that extension board. Note that the REQ call will only work when
191 // there is a board with a driver associated. There is however the case that the header area is a new area
192 // or an invalid area. We have a board detected bit could not setup the driver for it. The "ITEM_ID_FORMAT"
193 // item is used to setup the extension board NVM. It works without checking for a valid driver.
194 //
195 // The PUT and GET routines can be called for a board that has no driver associated yet. This way, for
196 // example, the driver type and other initial data can be set.
197 //
```

```
198 //-----------------------------------------------------------------------------------------------------
199 uint8_t drvReq( uint8_t boardId, uint8_t item, uint16_t *arg1, uint16_t *arg2 ) {
200
201     if ( boardId >= MAX_EXT_BOARD_MAP_ENTRIES )                 return ( ERR_INVALID_BOARD_ID );
202     if ( ! drvMap.map[ boardId ].flags & BF_EXT_BOARD_PRESENT ) return( ERR_INVALID_BOARD_ID );
203
204     if ( item == ITEM_ID_FORMAT ) {
205
206             LcsDrvEntry entry;
207
208             entry.flags = BF_EXT_BOARD_PRESENT | BF_EXT_BOARD_VALID;
209
210             uint8_t rStat = buildDrvBoardDescArea( boardId );
211             if ( rStat == ALL_OK ) drvMap.map[ boardId ] = entry;
212
213             return( rStat );
214     }
215     else {
216
217         if ( drvMap.map[ boardId ].drvFunc != nullptr ) {
218
219             return( drvMap.map[ boardId ].drvFunc( boardId - 1, item, arg1, arg2 ));
220         }
221         else return( ERR_EXT_BOARD_NOT_VALID );
222     }
223 }
224
225 } // namespace LCS
```

```
//-------------------------------------------------------------------------------------------------------------
//
// Layout Control System - runtime core.
//
//-------------------------------------------------------------------------------------------------------------
// The file contains the runtime core routines. They implement the node state machine that reacts to messages
// and advances according to state. The routines are called from the runtime loop in the setup file.
//
//-------------------------------------------------------------------------------------------------------------
//
// LCS - Core Library
// Copyright (C) 2021 - 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 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/>.
//
//-------------------------------------------------------------------------------------------------------------
#include "LcsRuntimeLib.h"
#include "LcsRtLibInt.h"

//-------------------------------------------------------------------------------------------------------------
// External declaration to global structures defined in "LcsRtSetup".
//
//-------------------------------------------------------------------------------------------------------------
namespace LCS {

    extern uint16_t             debugMask;
    extern LcsCdcMap            cdcMap;
    extern LcsNodeMap           nodeMap;
    extern LcsPortMap           portMap;
    extern LcsEventMap          eventMap;
    extern LcsCallbackMap       callbackMap;
    extern LcsTaskMap           taskMap;
    extern LcsPendingReqMap     pendingReqMap;
    extern LcsDrvFuncMap        drvFuncMap;
    extern LcsDrvMap            drvMap;
    extern LcsMsgBusCAN         *msgBus;
};

//-------------------------------------------------------------------------------------------------------------
// The LcsCoreLib implementation file local declarations and routines.
//
//-------------------------------------------------------------------------------------------------------------
namespace {

using namespace LCS;

//-------------------------------------------------------------------------------------------------------------
// Local constants and helper functions.
//
//-------------------------------------------------------------------------------------------------------------
const uint32_t  NODE_SETUP_RETRY_TIMER_VAL_MS   = 1000L;
uint32_t        timerVal                        = 0L;

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

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

uint16_t buildNpId( uint16_t nodeId, uint16_t portId ) {

    return(( nodeId << 4 ) | ( portId & 0xF ));
}

uint16_t nodeId( uint16_t npId ) {

    return( npId >> 4 );
}

uint16_t portId( uint16_t npId ) {

    return( npId & 0xF );
}

} // namespace

//-------------------------------------------------------------------------------------------------------------
//  The LCS name space routines declared in this file.
//
//-------------------------------------------------------------------------------------------------------------
namespace LCS {

//-------------------------------------------------------------------------------------------------------------
// General callback registration functions. They just set the function Id field. Straightforward.
//
//-------------------------------------------------------------------------------------------------------------
void registerLcsMsgCallback( LcsMsgCallback functionId ) {

  callbackMap.lcsMsgCallback = functionId;
}

void registerDccMsgCallback( LcsMsgCallback functionId ) {
```

```
99
100   callbackMap.dccMsgCallback = functionId;
101 }
102
103 void registerCmdCallback( LcsCmdCallback functionId ) {
104
105   callbackMap.cmdLineCallback = functionId;
106 }
107
108 void registerEventCallback( LcsEventCallback functionId ) {
109
110   callbackMap.eventCallback = functionId;
111 }
112
113 void registerInitCallback( LcsInitCallback functionId ) {
114
115     callbackMap.initCallback = functionId;
116 }
117
118 void registerResetCallback( LcsInitCallback functionId ) {
119
120     callbackMap.resetCallback = functionId;
121 }
122
123 void registerPfailCallback( LcsInitCallback functionId ) {
124
125     callbackMap.pfailCallback = functionId;
126 }
127
128 void registerReqCallback( LcsReqCallback functionId ) {
129
130     callbackMap.reqCallback = functionId;
131 }
132
133 void registerRepCallback( LcsRepCallback functionId ) {
134
135     callbackMap.repCallback = functionId;
136 }
137
138 //-------------------------------------------------------------------------------------------------------
139 // The core library features a very simple periodic task system. This routine adds a task callback to the
140 // pTaskMap. We only add entries, never remove them. A high water mark is used to record the highest entry
141 // used, so that processing will not run through empty entries.
142 //
143 // ??? perhaps this needs to be reworked to use HW driven timers.
144 //-------------------------------------------------------------------------------------------------------
145 uint8_t registerTaskCallback( LcsTaskCallback task, uint32_t interval ) {
146
147     if ( nodeMap.taskMapHwm < MAX_TASK_MAP_ENTRIES ) {
148
149         taskMap.map[ nodeMap.taskMapHwm ].task       = task;
150         taskMap.map[ nodeMap.taskMapHwm ].interval   = interval;
151         taskMap.map[ nodeMap.taskMapHwm ].timeStamp  = CDC::getMillis( );
152         nodeMap.taskMapHwm ++;
153         return ( ALL_OK );
154
155     } else return ( ERR_TASK_MAP_SIZE_EXCEEDED );
156 }
157
158 //-------------------------------------------------------------------------------------------------------
159 // "handleNodePortEvents" will be called for processing inbound port events on each loop iteration. Note that
160 // it does not matter where the events came from, i.e. whether another node sends an event or the event was
161 // created by a firmware call on this node. The event callback can be delayed with a timer value.
162 //
163 //-------------------------------------------------------------------------------------------------------
164 void handleNodePortEvents( ) {
165
166     if ( callbackMap.eventCallback == nullptr ) return;
167
168     uint32_t ts = CDC::getMillis( );
169
170     for ( uint8_t i = 0; i < MAX_PORT_MAP_ENTRIES; i ++ ) {
171
172         LcsPortMapEntry *pPtr = & portMap.map[ i ];
173
174         if (( pPtr -> flags & PF_PORT_ENABLED                ) &&
175             ( pPtr -> flags & PF_PORT_EVENT_HANDLING_ENABLED ) &&
176             ( pPtr -> flags & PF_EVENT_PENDING               )) {
177
178             if ( ts > pPtr -> eventTimeStamp ) {
179
180                 callbackMap.eventCallback(  pPtr -> eventNodeId,
181                                             pPtr -> eventId,
182                                             pPtr -> eventAction,
183                                             pPtr -> eventValue );
184             }
185
186             pPtr -> flags &= ~ PF_EVENT_PENDING;
187         }
188     }
189 }
190
191 //-------------------------------------------------------------------------------------------------------
192 // "handlePeriodicTasks" is called from the core library main processing loop. The idea is that there is a
193 // lot of periodic processing that needs to be one by any firmware implementation. Instead of the firmware
194 // developer writing its own handler, there is a crude method that just samples the timestamps and interval
195 // and triggers the callback then the interval is reached. Note that this is not very accurate from a timing
196 // perspective but will do for simple periodic processing.
197 //
```

```
198 //-----------------------------------------------------------------------------------------------------------
199 void handlePeriodicTasks( ) {
200
201     uint32_t ts = CDC::getMillis( );
202
203     for ( int i = 0; i < nodeMap.taskMapHwm; i++ ) {
204
205         LcsPTaskMapEntry *thisEntry = &taskMap.map[ i ];
206
207         if ( ts > thisEntry -> timeStamp  ) {
208
209             if ( thisEntry -> task != nullptr ) thisEntry -> task( );
210             thisEntry -> timeStamp = ts + thisEntry -> interval;
211         }
212     }
213 }
214
215 //-----------------------------------------------------------------------------------------------------------
216 // "handleMsgRepNid" handles the message that the configuring node sends to our node in response to a nodeId
217 // setup request. If the UID matches, the message is for our node and we update our nodeId accordingly in
218 // MEM and NVM. The next node state is OPERATE.
219 //
220 //-----------------------------------------------------------------------------------------------------------
221 void handleMsgRepNid( uint8_t *msg ) {
222
223     uint16_t nodeId   = ( msg[1] << 8 ) + msg[2];
224     uint32_t nodeUID  = ((uint32_t) msg[3] << 24 ) +
225                         ((uint32_t) msg[4] << 16 ) +
226                         ((uint32_t) msg[5] << 8 ) +
227                         msg[6];
228
229     if ( nodeUID == nodeMap.nodeUID ) {
230
231         if ( nodeMap.nodeId != nodeId ) {
232
233             nodeMap.nodeId = nodeId;
234             uint8_t rStat = rtNvmPutWord( NVM_NODE_MAP_START + offsetof( LcsNodeMap, nodeId ), nodeId );
235         }
236
237         nodeMap.nodeState = NS_OPERATE;
238     }
239 }
240
241 //-----------------------------------------------------------------------------------------------------------
242 // LCS management deals with messages concerning the general LCS management. If there is a callback defined
243 // it will be invoked. Then the node state is changed accordingly. Most updates are just to the MEM nodeMap.
244 // In addition, the READY and ACTIVITY LEDs are set.
245 //
246 //-----------------------------------------------------------------------------------------------------------
247 void handleMsgLcsMgt( uint8_t *msg ) {
248
249     switch ( msg[ 0 ] ) {
250
251         case LCS_OP_OPS: {
252
253             nodeMap.nodeState = NS_OPERATE;
254             if ( callbackMap.lcsMsgCallback != nullptr ) callbackMap.lcsMsgCallback( msg );
255
256         } break;
257
258         case LCS_OP_CFG: {
259
260             nodeMap.nodeState = NS_CONFIG;
261             if ( callbackMap.lcsMsgCallback != nullptr ) callbackMap.lcsMsgCallback( msg );
262
263         } break;
264
265         case LCS_OP_BON: {
266
267             CDC::writeDio( cdcMap.cfg.READY_LED_PIN, true );
268             nodeMap.nodeState = NS_OPERATE;
269             if ( callbackMap.lcsMsgCallback != nullptr ) callbackMap.lcsMsgCallback( msg );
270
271         } break;
272
273         case LCS_OP_BOF: {
274
275             CDC::writeDio( cdcMap.cfg.READY_LED_PIN, false );
276             nodeMap.nodeState = NS_HALTED;
277             if ( callbackMap.lcsMsgCallback != nullptr ) callbackMap.lcsMsgCallback( msg );
278
279         } break;
280
281         case LCS_OP_NCOL: {
282
283             CDC::writeDio( cdcMap.cfg.READY_LED_PIN, false );
284             CDC::writeDio( cdcMap.cfg.ACTIVE_LED_PIN, true );
285             nodeMap.nodeState = NS_COLLISION;
286             if ( callbackMap.lcsMsgCallback != nullptr ) callbackMap.lcsMsgCallback( msg );
287
288         } break;
289
290         case LCS_OP_RESET: {
291
292             uint16_t npId = (( msg[1] << 8 ) + msg[2] );
293             uint8_t rStat = resetNode( npId );
294
295             if ( rStat == ALL_OK ) LCS::sendAck( npId );
296             else                   LCS::sendErr( npId, rStat, 0, 0 );
```

```
297
298         } break;
299
300         case LCS_OP_SET_NID: {
301
302             uint16_t nodeId   = ( msg[1] << 8 ) + msg[2];
303             uint32_t nodeUID  = ((uint32_t) msg[3] << 24 ) + ((uint32_t) msg[4] << 16 ) +
304                                 ((uint32_t) msg[5] << 8 ) + msg[6];
305
306             if ( nodeUID == nodeMap.nodeUID ) {
307
308                 if ( nodeMap.nodeState == NS_CONFIG ) {
309
310                     if ( nodeId != nodeMap.nodeId ) nodeMap.nodeId = nodeId;
311                     uint8_t rStat = rtNvmPutWord( NVM_NODE_MAP_START + offsetof( LcsNodeMap, nodeId ), nodeId );
312
313                     sendAck( nodeId );
314                 }
315                 else sendErr( nodeId, ERR_NODE_NOT_CONFIG_STATE, 0, 0 );
316             }
317
318         } break;
319     }
320 }
321
322 //--------------------------------------------------------------------------------------------------------------
323 // "handleMsgGetNode" processes an incoming GET message for a node or port attribute. We construct the
324 // reply message with the requested data.
325 //
326 //--------------------------------------------------------------------------------------------------------------
327 void handleMsgGetNode( uint8_t *msg ) {
328
329     uint16_t npId =  (( msg[1] << 8 ) + msg[2] );
330
331     if ( nodeId( npId ) == nodeMap.nodeId ) {
332
333         uint8_t   item  = msg[3];
334         uint16_t  arg1  = ( msg[4] << 8 ) + msg[5];
335         uint16_t  arg2  = ( msg[6] << 8 ) + msg[7];
336         uint8_t   ret   = nodeGet( npId, item, &arg1, &arg2 );
337
338         if ( ret == ALL_OK )  sendRepNode( npId, item, arg1, arg2 );
339         else                  sendErr( npId, ret, 0, 0 );
340     }
341 }
342
343 //--------------------------------------------------------------------------------------------------------------
344 // "handleMsgPutNode" processes an incoming PUT message for a node or port attribute. We update the data
345 // and send a confirmation.
346 //
347 //--------------------------------------------------------------------------------------------------------------
348 void handleMsgPutNode( uint8_t *msg ) {
349
350     uint16_t npId =  (( msg[1] << 8 ) + msg[2] );
351
352     if ( nodeId( npId ) == nodeMap.nodeId ) {
353
354         uint8_t   item  = msg[3];
355         uint16_t  arg1  = ( msg[4] << 8 ) + msg[5];
356         uint16_t  arg2  = ( msg[6] << 8 ) + msg[7];
357         uint8_t   ret   = nodePut( npId, item, arg1, arg2 );
358
359         if ( ret == ALL_OK )  sendAck( npId );
360         else                  sendErr( npId, ret, 0, 0 );
361     }
362 }
363
364 //--------------------------------------------------------------------------------------------------------------
365 // "handleMsgRepNode" processes the answer to a previously sent node query. The incoming message will only
366 // result in an action when we have an outstanding request for that node. That is, this handler will only
367 // be called when the we passed the outstanding reply map check done before. All reply messages are routed
368 // to this one callback. It is up to the firmware programmer to analyze for what and whom the reply really
369 // is.
370 //
371 //--------------------------------------------------------------------------------------------------------------
372 void handleMsgRepNode( uint8_t *msg ) {
373
374     uint16_t  npId    = (( msg[1] << 8 ) + msg[2] );
375     uint8_t   item    = msg[3];
376     uint16_t  arg1    = ( msg[4] << 8 ) + msg[5];
377     uint16_t  arg2    = ( msg[6] << 8 ) + msg[7];
378
379     if ( callbackMap.repCallback != nullptr ) callbackMap.repCallback( npId, item, arg1, arg2, ALL_OK );
380 }
381
382 //--------------------------------------------------------------------------------------------------------------
383 // "handleMsgReqNode" processes an incoming request for a node or port. The REQ message request will result
384 // in invoking the register firmware callback. We send a confirmation message.
385 //
386 //--------------------------------------------------------------------------------------------------------------
387 void handleMsgReqNode( uint8_t *msg ) {
388
389     uint16_t npId = (( msg[1] << 8 ) + msg[2] );
390
391     if ( nodeId( npId ) == nodeMap.nodeId ) {
392
393         uint8_t   item  = msg[3];
394         uint16_t  arg1  = ( msg[4] << 8 ) + msg[5];
395         uint16_t  arg2  = ( msg[6] << 8 ) + msg[7];
```

```
396         uint8_t   ret   = nodeReq( npId, item, &arg1, &arg2 );
397
398         if ( ret == ALL_OK )  sendAck( npId );
399         else                  sendErr( npId, ret, 0, 0 );
400     }
401 }
402
403 //-----------------------------------------------------------------------------------------------------------
404 // "handleMsgEvent" deals with the event messages for inbound ports. For all matching events in the eventMap,
405 // the respective port map entry fields are set. The searchEvent function will return us the first matching
406 // entry in the sorted event map, if any. From there, we just hop along the eventMap until the eventId does
407 // not match any longer. All we do in this routine is to record the event data and the optional future time
408 // stamp when the event should result in a callback. The actual event processing is done in the port event
409 // processing routine, which will manage the timely invocation of the event callbacks.
410 //
411 // Note that we also are called from the event sending routine because another port on our node could be
412 // interested in this event. It is up to the firmware programmer to ensure that a port does send itself an
413 // event and may trigger an infinite loop.
414 //
415 //-----------------------------------------------------------------------------------------------------------
416 void handleMsgEvent( uint8_t *msg ) {
417
418     uint16_t  eventId = ( msg[3] * 256 ) + msg[4];
419     int       index   = searchEvent( eventId );
420
421     if ( index >= 0 ) {
422
423         uint8_t   opCode            = msg[0];
424         uint16_t  nodeId            = ( msg[1] * 256 ) + msg[2];
425         uint16_t  eventData         = ( msg[5] * 256 ) + msg[6];
426         uint8_t   eventAction       = PEA_EVENT_IDLE;
427         uint32_t  ts                = CDC::getMillis( );
428
429         switch ( opCode ) {
430
431             case LCS_OP_EVT_ON:   eventAction = PEA_EVENT_ON;   break;
432             case LCS_OP_EVT_OFF:  eventAction = PEA_EVENT_OFF;  break;
433             case LCS_OP_EVT:      eventAction = PEA_EVENT_EVT;  break;
434         }
435
436         while (( index < nodeMap.eventMapHwm ) && ( eventMap.map[ index ].eventId == eventId )) {
437
438             LcsPortMapEntry *pPtr = &portMap.map[ index ];
439
440             if (( pPtr -> flags & PF_PORT_ENABLED                  ) &&
441                 ( pPtr -> flags & PF_PORT_EVENT_HANDLING_ENABLED   )) {
442
443                 pPtr -> eventNodeId     = nodeId;
444                 pPtr -> eventId         = eventId;
445                 pPtr -> eventAction     = eventAction;
446                 pPtr -> eventValue      = eventData;
447                 pPtr -> eventTimeStamp  = ts + ( pPtr -> eventDelayTime * EVENT_DELAY_TICK_MILLIS );
448                 pPtr -> flags           |= PF_EVENT_PENDING;
449             }
450
451             index++;
452         }
453     }
454 }
455
456 //-----------------------------------------------------------------------------------------------------------
457 // We received a DCC subsystem message. These messages are handler solely by firmware, which is typically
458 // the base station, a handheld, or a decoder alike device. All we do is to pass the message to the call
459 // back routine. One day, we could decode the message a bit more and invoke more specialized callback.
460 //
461 //-----------------------------------------------------------------------------------------------------------
462 void handleMsgDccMgt( uint8_t *msg ) {
463
464     if ( callbackMap.dccMsgCallback != NULL ) callbackMap.dccMsgCallback( msg );
465 }
466
467 //-----------------------------------------------------------------------------------------------------------
468 // Node state INIT. This is the first state after the initial library setup. The runtime init call created
469 // all memory areas and initialized the data structures. After a successful init call, the state is INIT and
470 // the firmware programmer can register the necessary callback functions and do other firmware specific work.
471 // Eventually, the runtime loop method is called. If the "init" option is set, the node init and port init
472 // callback routine will be invoked. If the nodeId validation option is set, the node will request a nodeId
473 // and enter the state SETUP. Otherwise the next state is OPERATE.
474 //
475 // ??? idea: a port init call cold return a status that says "this port is not used". This way we could
476 // set a better HWM.
477 //-----------------------------------------------------------------------------------------------------------
478 void handleNodeStateInit( ) {
479
480     if ( ! ( nodeMap.nodeOptions & NOPT_SKIP_NODE_INIT_STEP )) {
481
482         if ( callbackMap.initCallback != nullptr ) {
483
484             if ( callbackMap.initCallback ) callbackMap.initCallback( nodeMap.nodeId << 4 );
485         }
486
487         for ( uint8_t i = 0; i < MAX_PORT_MAP_ENTRIES; i++ ) {
488
489             if ( callbackMap.initCallback ) callbackMap.initCallback(( nodeMap.nodeId << 4 ) | i + 1 );
490
491             portMap.map[ i ].flags |= PF_PORT_ENABLED;
492             portMap.map[ i ].flags |= PF_PORT_EVENT_HANDLING_ENABLED;
493         }
494     }
```

```
495
496     if ( ! ( nodeMap.nodeOptions & NOPT_SKIP_NODE_ID_CONFIG )) {
497
498         sendReqNodeId( nodeMap.nodeId, nodeMap.nodeUID, 0 );
499         timerVal  = CDC::getMillis( );
500
501         nodeMap.nodeState = NS_REGISTER;
502
503     } else nodeMap.nodeState = NS_OPERATE;
504 }
505
506 //-----------------------------------------------------------------------------------------------------------
507 // Node State FAIL. This is the state after the node startup failed.
508 //
509 // ??? would we ever come here ? if the INIT failed, the firmware programmer should do what ?
510 //-----------------------------------------------------------------------------------------------------------
511 void handleNodeStateFail( ) {
512
513     handleSerialCommand( );
514
515 }
516
517 //-----------------------------------------------------------------------------------------------------------
518 // Node State Power FAIL. This is the state after when the node starts up after a power fail. We have this
519 // state so that the firmware programmer can take some recovery action before the power goes away. x
520 //
521 //-----------------------------------------------------------------------------------------------------------
522 void handleNodeStatePfail( ) {
523
524     if ( callbackMap.pfailCallback != nullptr ) {
525
526         callbackMap.pfailCallback( nodeMap.nodeId << 4 );
527     }
528
529     for ( uint8_t i = 0; i < MAX_PORT_MAP_ENTRIES; i++ ) {
530
531         callbackMap.pfailCallback(( nodeMap.nodeId << 4 ) | i + 1 );
532     }
533
534     handleSerialCommand( );
535 }
536
537 //-----------------------------------------------------------------------------------------------------------
538 // Node State REGISTER. This is the state after the INIT state when a nodeId setup was requested. We are
539 // waiting for a nodeId reply message. If there is a timely reply message, we will handle the message reply
540 // and the node state will advance. If there is no timely reply, we will resubmit the request.
541 //
542 //-----------------------------------------------------------------------------------------------------------
543 void handleNodeStateRegister( ) {
544
545     uint8_t msg[ MAX_LCS_MSG_SIZE ];
546
547     switch ( msgBus -> receiveLcsMsg( msg )) {
548
549         case LCS_OP_REP_NID: handleMsgRepNid( msg );  break;
550         case LCS_OP_RESET:   handleMsgLcsMgt( msg );  break;
551
552         default: {
553
554             if (( CDC::getMillis( ) - timerVal ) > NODE_SETUP_RETRY_TIMER_VAL_MS ) {
555
556                 sendReqNodeId( nodeMap.nodeId, nodeMap.nodeUID, 0 );
557                 timerVal = CDC::getMillis( );
558              }
559         }
560     }
561
562     handleSerialCommand( );
563 }
564
565 //-----------------------------------------------------------------------------------------------------------
566 // Node State COLLISION. This is the state after the node receiver routine detected a nodeId collision. We
567 // will stay in this state and only react to RESET and SET_NID messages.
568 //
569 //-----------------------------------------------------------------------------------------------------------
570 void handleNodeStateCollision( ) {
571
572     uint8_t msg[ MAX_LCS_MSG_SIZE ];
573
574     switch ( msgBus -> receiveLcsMsg( msg )) {
575
576         case LCS_OP_RESET:
577         case LCS_OP_SET_NID:  handleMsgLcsMgt( msg ); break;
578     }
579
580     handleSerialCommand( );
581 }
582
583 //-----------------------------------------------------------------------------------------------------------
584 // Node State HALTED. The LCS communication bus was halted for all nodes. Note that the bus is still there,
585 // just not active. We just listen to the BON or RESET message to get going again.
586 //
587 //-----------------------------------------------------------------------------------------------------------
588 void handleNodeStateHalted( ) {
589
590     uint8_t msg[ MAX_LCS_MSG_SIZE ];
591
592     switch ( msgBus -> receiveLcsMsg( msg )) {
593
```

```
594         case LCS_OP_BON:
595         case LCS_OP_RESET: handleMsgLcsMgt( msg ); break;
596     }
597 }
598 
599 //-------------------------------------------------------------------------------------------------------------
600 // Node State CONFIG. A node can be placed into configuration state. We process any LCS message, handle the
601 // periodic tasks registered and port events that may have been received. Note that we just listen to messages
602 // valid for that mode and invoke the respective handler. All other messages are ignored.
603 //
604 //-------------------------------------------------------------------------------------------------------------
605 void handleNodeStateConfig( ) {
606 
607     uint8_t msg[ MAX_LCS_MSG_SIZE ];
608 
609     switch ( msgBus -> receiveLcsMsg( msg )) {
610 
611         case LCS_OP_OPS:
612         case LCS_OP_RESET:
613         case LCS_OP_BON:
614         case LCS_OP_BOF:
615         case LCS_OP_ACK:
616         case LCS_OP_ERR:
617         case LCS_OP_SET_NID:
618         case LCS_OP_NCOL:           handleMsgLcsMgt( msg );              break;
619 
620         case LCS_OP_NODE_GET:       handleMsgGetNode( msg );             break;
621         case LCS_OP_NODE_REP:       handleMsgRepNode( msg );             break;
622         case LCS_OP_NODE_REQ:       handleMsgReqNode( msg );             break;
623     }
624 
625     handlePeriodicTasks( );
626     handleNodePortEvents( );
627     handleSerialCommand( );
628 }
629 
630 //-------------------------------------------------------------------------------------------------------------
631 // Node State OPERATIONS. Most of the time the node state is in operations mode. We process any LCS message,
632 // handle the periodic tasks registered and port events that may have been received. Note that we just listen
633 // to messages valid for that mode and invoke the respective handler. All other messages are ignored.
634 //
635 //-------------------------------------------------------------------------------------------------------------
636 void handleNodeStateOperations( ) {
637 
638     uint8_t msg [ MAX_LCS_MSG_SIZE ];
639 
640     switch ( msgBus -> receiveLcsMsg( msg )) {
641 
642         case LCS_OP_CFG:
643         case LCS_OP_RESET:
644         case LCS_OP_BON:
645         case LCS_OP_BOF:
646         case LCS_OP_ACK:
647         case LCS_OP_ERR:
648         case LCS_OP_REQ_NID:
649         case LCS_OP_NCOL:           handleMsgLcsMgt( msg );             break;
650 
651         case LCS_OP_NODE_GET:       handleMsgGetNode( msg );            break;
652         case LCS_OP_NODE_PUT:       handleMsgPutNode( msg );            break;
653         case LCS_OP_NODE_REQ:       handleMsgReqNode( msg );            break;
654         case LCS_OP_NODE_REP:       handleMsgRepNode( msg );            break;
655 
656         case LCS_OP_EVT_ON:
657         case LCS_OP_EVT_OFF:
658         case LCS_OP_EVT:            handleMsgEvent( msg );              break;
659 
660         case LCS_OP_REQ_LOC:
661         case LCS_OP_REL_LOC:
662         case LCS_OP_REP_LOC:
663         case LCS_OP_SET_LCON:
664         case LCS_OP_KEEP_LOC:
665         case LCS_OP_SET_LSPD:
666         case LCS_OP_SET_LMOD:
667         case LCS_OP_LOC_FON:
668         case LCS_OP_LOC_FOF:
669 
670         case LCS_OP_SET_CVM:
671         case LCS_OP_REQ_CVS:
672         case LCS_OP_REP_CVS:
673         case LCS_OP_SET_CVS:
674 
675         case LCS_OP_TON:
676         case LCS_OP_TOF:
677         case LCS_OP_ESTP:
678 
679         case LCS_OP_SEND_DCC3:
680         case LCS_OP_SEND_DCC4:
681         case LCS_OP_SEND_DCC5:
682         case LCS_OP_SEND_DCC6:
683 
684         case LCS_OP_DCC_ACK:
685         case LCS_OP_DCC_ERR:        handleMsgDccMgt( msg );             break;
686     }
687 
688     handlePeriodicTasks( );
689     handleNodePortEvents( );
690     handleSerialCommand( );
691 }
692 
```

```
693 //-------------------------------------------------------------------------------------------------------------
694 // "handleNodeState" is the main routine of the node activity processing. It is the method called after all
695 // setup is done. Running in a loop, the primary function is to handle the activities according to the node
696 // state. The run loop also processes the serial commands. Note that this function will not return.
697 //
698 //-------------------------------------------------------------------------------------------------------------
699 void handleNodeState( ) {
700
701     while ( true ) {
702
703         switch ( nodeMap.nodeState ) {
704
705             case NS_INIT:       handleNodeStateInit( );       break;
706             case NS_FAIL:       handleNodeStateFail( );       break;
707             case NS_REGISTER:   handleNodeStateRegister( );   break;
708             case NS_COLLISION:  handleNodeStateCollision( );  break;
709             case NS_HALTED:     handleNodeStateHalted( );     break;
710             case NS_CONFIG:     handleNodeStateConfig( );     break;
711             case NS_OPERATE:    handleNodeStateOperations( ); break;
712             default: ;
713         }
714     }
715 }
716
717 }; // namespace
```

## A.3 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;
```

```
99      uint8_t repeat;
100     uint8_t buf[ DCC_PACKET_SIZE ];
101 };
102 
103 //-----------------------------------------------------------------------------------------------------------
104 // DCC packet payload data definitions we need often, so these constants come in handy.
105 //
106 //-----------------------------------------------------------------------------------------------------------
107 const uint8_t   idleDccPacketData[ ]    = { 0xFF, 0x00 };
108 const uint8_t   resetDccPacketData[ ]   = { 0x00, 0x00 };
109 const uint8_t   eStopDccPacketData[ ]   = { 0x00, 0x01 };
110 
111 //-----------------------------------------------------------------------------------------------------------
112 // Setup options to set for the DCC track. They are set when the track object is created.
113 //
114 //  DT_OPT_SERVICE_MODE_TRACK  - The track is a PROG track.
115 //  DT_OPT_CUTOUT              - The track is configured to emit a cutout during the DCC packet preamble.
116 //  DT_OPT_RAILCOM             - The track support Railcom detection.
117 //
118 //-----------------------------------------------------------------------------------------------------------
119 enum DccTrackOptions : uint16_t {
120 
121     DT_OPT_DEFAULT_SETTING      = 0,
122     DT_OPT_SERVICE_MODE_TRACK   = 1 << 0,
123     DT_OPT_CUTOUT               = 1 << 1,
124     DT_OPT_RAILCOM              = 1 << 2
125 };
126 
127 //-----------------------------------------------------------------------------------------------------------
128 // The DCC track object has a set of flags to indicate its current status.
129 //
130 //  DT_F_POWER_ON             - The track is under power.
131 //  DT_F_POWER_OVERLOAD       - An overload situation was detected.
132 //  DT_F_MEASUREMENT_ON       - The power measurement is enabled.
133 //  DT_F_SERVICE_MODE_ON      - The track is currently in service mode, i.e. is a PROG track.
134 //  DT_F_CUTOUT_MODE_ON       - The track has the cutout generation enabled.
135 //  DT_F_RAILCOM_MODE_ON      - The track has the railcom detect enabled.
136 //  DT_F_RAILCOM_MSG_PENDING  - If railcom is enabled, a received datagram is indicated.
137 //  DT_F_CONFIG_ERROR         - The passed configuration descriptor has invalid options configured.
138 //
139 //-----------------------------------------------------------------------------------------------------------
140 enum DccTrackFlags : uint16_t {
141 
142     DT_F_DEFAULT_SETTING      = 0,
143     DT_F_POWER_ON             = 1 << 0,
144     DT_F_POWER_OVERLOAD       = 1 << 1,
145     DT_F_MEASUREMENT_ON       = 1 << 2,
146     DT_F_SERVICE_MODE_ON      = 1 << 3,
147     DT_F_CUTOUT_MODE_ON       = 1 << 4,
148     DT_F_RAILCOM_MODE_ON      = 1 << 5,
149     DT_F_DCC_PACKET_PENDING   = 1 << 6,
150     DT_F_RAILCOM_MSG_PENDING  = 1 << 7,
151     DT_F_CONFIG_ERROR         = 1 << 15
152 };
153 
154 //-----------------------------------------------------------------------------------------------------------
155 // The following constants are for the current consumption RMS measurement. The idea is to record the measured
156 // ADC values in a circular buffer, every time a certain amount of milliseconds has passed. This work is done
157 // by the DCC track state machine as part of the power on state.
158 //
159 //-----------------------------------------------------------------------------------------------------------
160 const uint8_t   PWR_SAMPLE_BUF_SIZE               = 64;
161 const uint32_t  PWR_SAMPLE_TIME_INTERVAL_MILLIS   = 16;
162 
163 //-----------------------------------------------------------------------------------------------------------
164 // The RailCom buffer size. During the cutout period up to eight bytes of raw data are sent by the decoder if
165 // the Railcom option is enabled.
166 //
167 //-----------------------------------------------------------------------------------------------------------
168 const uint8_t   RAILCOM_BUF_SIZE = 8;
169 
170 //-----------------------------------------------------------------------------------------------------------
171 // The session map options. These are options initially set when the base station starts. They are used to
172 // set the flags, which are then used for processing the the actual settings.
173 //
174 //  SM_KEEP_ALIVE_CHECKING  - enable keep alive checking. When enabled, the locomotive session need to receive
175 //                            a keep alive LCS message periodically.
176 //  SM_ENABLE_REFRESH       - refresh the session data. This will send the locomotive speed and direction as
177 //                            well as the function flags periodically in a round robin processing of the
178 //
179 //-----------------------------------------------------------------------------------------------------------
180 enum SessionMapOptions : uint16_t {
181 
182     SM_OPT_DEFAULT_SETTING        = 0,
183     SM_OPT_KEEP_ALIVE_CHECKING    = 1 << 0,
184     SM_OPT_ENABLE_REFRESH         = 1 << 1
185 };
186 
187 //-----------------------------------------------------------------------------------------------------------
188 // The session map flags. The apply to all sessions in the session map. The initial values are copied from
189 // session option initial values.
190 //
191 //  SM_F_KEEP_ALIVE_CHECKING  - enable keep alive checking. When enabled, the locomotive session need to receive
192 //                            a keep alive LCS message periodically.
193 //  SM_F_ENABLE_REFRESH       - refresh the session data. This will send the locomotive speed and direction as
194 //                            well as the function flags periodically in a round robin processing of the
195 //
196 //-----------------------------------------------------------------------------------------------------------
197 enum SessionMapFlags : uint16_t {
```

```
198
199     SM_F_DEFAULT_SETTING        = 0,
200     SM_F_KEEP_ALIVE_CHECKING    = 1 << 0,
201     SM_F_ENABLE_REFRESH         = 1 << 1
202 };
203
204 //-----------------------------------------------------------------------------------------------------------
205 // Each session map entry has a set of flags.
206 //
207 //  SME_ALLOCATED           - the session is allocated, the entry valid.
208 //  SME_COMBINED_REFRESH    - locomotive speed/dir and functions are refreshed using the combined DCC packet.
209 //  SME_SPDIR_REFRESH       - locomotive speed/dir are refreshed.
210 //  SME_FUNC_REFRESH        - locomotive functions are refreshed.
211 //  SME_DISPATCHED          -
212 //  SME_SHARED              -
213 //
214 //
215 // ??? when the base station has a config value of using the DCC spdir/func command, these flags need to be
216 // named slightly different. Should we still have the option to enable or disable it even though the base
217 // station can do it ? A decoder might not support this packet type...
218 //-----------------------------------------------------------------------------------------------------------
219 enum SessionMapEntryFlags : uint16_t {
220
221     SME_DEFAULT_SETTING     = 0,
222     SME_ALLOCATED           = 1 << 0,
223     SME_COMBINED_REFRESH    = 1 << 1,
224     SME_SPDIR_ONLY_REFRESH  = 1 << 2, // ??? phase out...
225     SME_SPDIR_REFRESH       = 1 << 2,
226     SME_FUNC_REFRESH        = 1 << 3,
227     SME_DISPATCHED          = 1 << 4,
228     SME_SHARED              = 1 << 5
229 };
230
231 //-----------------------------------------------------------------------------------------------------------
232 // The base station items for nodeInfo and nodeControl calls .... tbd
233 //
234 // ??? the are mapped in the MEM / NVM range as well as in the USER range.
235 // ??? how to do it consistently and understandably ?
236 //-----------------------------------------------------------------------------------------------------------
237 enum BaseStationItems : uint8_t {
238
239     // or use GET in all constants
240
241     BS_ITEM_SESSION_MAP_OPTIONS   = 128,
242     BS_ITEM_SESSION_MAP_FLAGS     = 129,
243     BS_ITEM_MAX_SESSIONS          = 130,
244     BS_ITEM_ACTIVE_SESSIONS       = 131,
245
246     BS_ITEM_INIT_CURRENT_VAL      = 140,
247     BS_ITEM_LIMIT_CURRENT_VAL     = 140,
248     BS_ITEM_MAX_CURRENT_VAL       = 140,
249     BS_ITEM_ACTUAL_CURRENT_VAL     = 140,
250
251     // thresholds
252
253     // eventID to send for events ?
254
255 };
256
257 //-----------------------------------------------------------------------------------------------------------
258 //
259 //
260 //-----------------------------------------------------------------------------------------------------------
261 const uint32_t  MAIN_TRACK_STATE_TIME_INTERVAL  = 10;
262 const uint32_t  PROG_TRACK_STATE_TIME_INTERVAL  = 10;
263 const uint32_t  SESSION_REFRESH_TASK_INTERVAL   = 50;
264
265 const uint16_t  MAX_CAB_SESSIONS                = 64;
266
267 //-----------------------------------------------------------------------------------------------------------
268 // For creating the Loco Session object the session map object is described by the following descriptor.
269 //
270 //-----------------------------------------------------------------------------------------------------------
271 struct LcsBaseStationSessionMapDesc {
272
273     uint16_t    options       = SM_OPT_DEFAULT_SETTING;
274     uint16_t    maxSessions   = MAX_CAB_SESSIONS;
275 };
276
277 //-----------------------------------------------------------------------------------------------------------
278 // For creating the DCC track object, the track is described by the data structure below. In addition to the
279 // hardware pins enablePin, dcc1Pin1, dccPin2 and sensePin, there are the limits for current consumption
280 // values, all specified in milliAmps. The initial current sets the current consumption limit after the track
281 // is turned on. The limit current consumption specifies the actual configured value that is checked for a
282 // track current overload situation. The maximum current defines what current the power module should never
283 // exceed. For the measurements to work, the power module needs to deliver a voltage that corresponds to the
284 // current drawn on the track. The value is measured in milliVolt per Ampere drawn. Finally, there are
285 // threshold times for managing the track overload and restart capability.
286 //
287 //-----------------------------------------------------------------------------------------------------------
288 struct LcsBaseStationTrackDesc {
289
290     uint16_t  options                       = SM_OPT_DEFAULT_SETTING;
291
292     uint8_t   enablePin                     = CDC::UNDEFINED_PIN;
293     uint8_t   dccSigPin1                    = CDC::UNDEFINED_PIN;
294     uint8_t   dccSigPin2                    = CDC::UNDEFINED_PIN;
295     uint8_t   sensePin                      = CDC::UNDEFINED_PIN;
296     uint8_t   uartRxPin                     = CDC::UNDEFINED_PIN;
```

```
297
298     uint16_t  initCurrentMilliAmp           = 0;
299     uint16_t  limitCurrentMilliAmp          = 0;
300     uint16_t  maxCurrentMilliAmp            = 0;
301     uint16_t  milliVoltPerAmp               = 0;
302
303     uint16_t  startTimeThresholdMillis      = 0;
304     uint16_t  stopTimeThresholdMillis       = 0;
305     uint16_t  overloadTimeThresholdMillis   = 0;
306     uint16_t  overloadEventThreshold        = 0;
307     uint16_t  overloadRestartThreshold      = 0;
308 };
309
310 //-----------------------------------------------------------------------------------------------------------
311 // DCC track definition. The DCC track object is responsible for managing the track power as well as building
312 // and sending the DCC packet bit stream. A packet consists of the preamble bits, the postamble bit, the data
313 // bytes separated with a ZERO bit and a checksum byte. Creating the DCC bit stream is done with the signal
314 // generation routines. The signal state machine, running on a 29 microsecond tick, takes a DCC packet and
315 // gets it out to the track. The DCC signal state machine also invokes follow up actions that measure the
316 // actual power consumption, read in a railcom message and so on. There is also a DCC log facility which
317 // records internal events for testing and debugging.
318 //
319 // The other state machine will manage the actual track power. This machine is responsible for the periodic
320 // checking of power consumption and resulting power control. In contrast to the DCC signal state machine,
321 // this machine is not driven by a periodic interrupt but invoked periodically via the LCS runtime task
322 // manager.
323 //
324 // For a base station, there will be two track objects. One is the MAIN track and the other one is the PROG
325 // track. Each track has a DCC track object associated with it. In addition to the two track objects, there
326 // are class level static routines to manage the timer hardware functions, the analog signal read for current
327 // measurement and the serial IO for the optional RailCom message processing. The current version is AtMega
328 // specific.
329 //
330 //-----------------------------------------------------------------------------------------------------------
331 struct LcsBaseStationDccTrack {
332
333     public:
334
335     LcsBaseStationDccTrack( );
336
337     uint8_t                     setupDccTrack( LcsBaseStationTrackDesc* trackDesc );
338     void                        loadPacket( const uint8_t *packet, uint8_t len, uint8_t repeat = 0 );
339
340     uint16_t                    getFlags( );
341     uint16_t                    getOptions( );
342
343     bool                        isServiceModeOn( );
344     void                        serviceModeOn( );
345     void                        serviceModeOff( );
346
347     void                        runDccTrackStateMachine( );
348     void                        powerStart( );
349     void                        powerStop( );
350     bool                        isPowerOn( );
351     bool                        isPowerOverload( );
352
353     void                        cutoutOn( );
354     void                        cutoutOff( );
355     bool                        isCutoutOn( );
356
357     void                        railComOn( );
358     void                        railComOff( );
359     bool                        isRailComOn( );
360
361     void                        setLimitCurrent( uint16_t val );
362     uint16_t                    getLimitCurrent( );
363     uint16_t                    getActualCurrent( );
364     uint16_t                    getInitCurrent( );
365     uint16_t                    getMaxCurrent( );
366     uint16_t                    getRMSCurrent( );
367
368     uint16_t                    decoderAckBaseline( uint8_t resetPacketsToSend );
369     bool                        decoderAckDetect( uint16_t baseValue, uint8_t retries );
370     void                        checkOverload( );
371
372     void                        runDccSignalStateMachine( volatile uint8_t *timeToInterrupt, uint8_t *followUpAction );
373
374     void                        getNextBit( );
375     void                        getNextPacket( );
376     void                        powerMeasurement( );
377
378     void                        startRailComIO( );
379     void                        stopRailComIO( );
380     uint8_t                     handleRailComMsg( );
381     uint8_t                     getRailComMsg( uint8_t *buf, uint8_t bufLen );
382
383     uint32_t                    getDccPacketsSend( );
384     uint32_t                    getPwrSamplesTaken( );
385     uint16_t                    getPwrSamplesPerSec( );
386
387     void                        printDccTrackConfig( );
388     void                        printDccTrackStatus( );
389
390     void                        enableLog( bool arg );
391     void                        beginLog( );
392     void                        endLog( );
393     void                        printLog( );
394
395     void                        writeLogData( uint8_t id, uint8_t *buf, uint8_t len );
```

```
396     void                        writeLogId( uint8_t id );
397     void                        writeLogTs( );
398     void                        writeLogVal( uint8_t valId, uint16_t val );
399
400     private:
401
402     uint16_t                    options                     = DT_OPT_DEFAULT_SETTING;
403     volatile uint16_t           flags                       = DT_F_DEFAULT_SETTING;
404
405     volatile uint8_t            trackState                  = 0;
406     volatile uint8_t            signalState                 = 0;
407
408     volatile uint32_t           trackTimeStamp              = 0;
409     volatile uint8_t            overloadEventCount          = 0;
410     volatile uint8_t            overloadRestartCount        = 0;
411
412     uint8_t                     enablePin                   = CDC::UNDEFINED_PIN;
413     uint8_t                     dccSigPin1                  = CDC::UNDEFINED_PIN;
414     uint8_t                     dccSigPin2                  = CDC::UNDEFINED_PIN;
415     uint8_t                     sensePin                    = CDC::UNDEFINED_PIN;
416     uint8_t                     uartRxPin                   = CDC::UNDEFINED_PIN;
417
418     uint16_t                    initCurrentMilliAmp         = 0;
419     uint16_t                    limitCurrentMilliAmp        = 0;
420     uint16_t                    maxCurrentMilliAmp          = 0;
421
422     uint16_t                    startTimeThreshold          = 0;
423     uint16_t                    stopTimeThreshold           = 0;
424     uint16_t                    overloadTimeThreshold       = 0;
425     uint16_t                    overloadEventThreshold      = 0;
426     uint16_t                    overloadRestartThreshold    = 0;
427
428     uint16_t                    milliVoltPerAmp             = 0;
429     uint16_t                    digitsPerAmp                = 0;
430     volatile uint16_t           actualCurrentDigitValue     = 0;
431     volatile uint16_t           highWaterMarkDigitValue     = 0;
432     volatile uint16_t           limitCurrentDigitValue      = 0;
433     uint16_t                    ackThresholdDigitValue      = 0;
434
435     uint32_t                    totalPwrSamplesTaken        = 0;
436     uint32_t                    lastPwrSampleTimeStamp      = 0;
437
438     uint32_t                    lastPwrSamplePerSecTaken    = 0;
439     uint32_t                    lastPwrSamplePerSecTimeStamp = 0;
440     uint32_t                    pwrSamplesPerSec            = 0;
441
442     uint8_t                     preambleLen                 = 0;
443     uint8_t                     postambleLen                = 0;
444     volatile bool               currentBit                  = false;
445     volatile uint8_t            bytesSent                   = 0;
446     volatile uint8_t            bitsSent                    = 0;
447     volatile uint8_t            preambleSent                = 0;
448     volatile uint8_t            postambleSent               = 0;
449     uint32_t                    dccPacketsSend              = 0;
450
451     DccPacket                   dccBuf1;
452     DccPacket                   dccBuf2;
453     DccPacket                   *activeBufPtr   = nullptr;
454     DccPacket                   *pendingBufPtr  = nullptr;
455
456     // ??? to add....
457     // base station capabilities according to RCN200 - 4 16 bit words
458     // sample values per second for samples and dcc packets
459     // buffers for POM / XPOM data
460     // queue for POM / XPOM commands
461
462     uint8_t                     railComBufIndex                 = 0;
463     uint8_t                     railComMsgBuf[ RAILCOM_BUF_SIZE ] = { 0 };
464
465     uint8_t                     pwrSampleBufIndex                   = 0;
466     uint16_t                    pwrSampleBuf[ PWR_SAMPLE_BUF_SIZE ]     = { 0 };
467
468     public:
469
470     static void                 startDccProcessing( );
471
472 };
473
474 //-----------------------------------------------------------------------------------------------------------
475 // Every allocated loco session is described by the sessionMap structure. There are the engine cab Id, speed,
476 // direction and function information. There is also a field that indicates when we received information for
477 // this session from a cab control handheld. The function flags are stored in an array, each byte representing
478 // a group. Most of the fields are actually used for a DCC type locomotive. When the locomotive is an analog
479 // engine, only a subset of the fields is actually used. Nevertheless, even for an analog engine we will
480 // have a session. The base station will however not generate packets for this engine.
481 //
482 //-----------------------------------------------------------------------------------------------------------
483 struct SessionMapEntry {
484
485   uint16_t        flags               = SME_DEFAULT_SETTING;
486   uint16_t        cabId               = LCS::NIL_CAB_ID;
487   uint8_t         speed               = 0;
488   uint8_t         speedSteps          = 128;
489   uint8_t         direction           = 0;
490   uint8_t         engineState         = 0;
491   uint8_t         nextRefreshStep     = 0;
492   unsigned long   lastKeepAliveTime   = 0;
493   uint8_t         functions[ LCS::MAX_DCC_FUNC_GROUP_ID ] = { 0 };
494
```

```
};

//-----------------------------------------------------------------------------------------------------------------
// 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 {

```

```
594     public:
595
596     LcsBaseStationMsgInterface( );
597
598     uint8_t setupLcsMsgInterface( LcsBaseStationLocoSession   *locoSessions,
599                                   LcsBaseStationDccTrack      *mainTrack,
600                                   LcsBaseStationDccTrack      *progTrack
601                                 );
602
603     void handleLcsMsg( uint8_t *msg );
604
605     private:
606
607     LcsBaseStationLocoSession   *locoSessions   = nullptr;
608     LcsBaseStationDccTrack      *mainTrack      = nullptr;
609     LcsBaseStationDccTrack      *progTrack      = nullptr;
610
611 };
612
613 //--------------------------------------------------------------------------------------------------------------
614 // The base station implements a serial IO command interface. The command interface uses the DCC++ syntax of
615 // a command line and where it is a original DCC++ command it implements them in a compatible way. The idea
616 // is to one day connect to the programs of the JMRI world, which support the DCC++ style command interface.
617 //
618 //--------------------------------------------------------------------------------------------------------------
619 struct LcsBaseStationCommand {
620
621     public:
622
623     LcsBaseStationCommand( );
624
625     uint8_t setupSerialCommand( LcsBaseStationLocoSession  *locoSessions,
626                                 LcsBaseStationDccTrack     *mainTrack,
627                                 LcsBaseStationDccTrack     *progTrack );
628
629     void handleSerialCommand( char *s );
630
631     private:
632
633     void openSessionCmd( char *s );
634     void closeSessionCmd( char *s );
635
636     void setThrottleCmd( char *s );
637     void setFunctionBitCmd( char *s );
638     void setFunctionGroupCmd( char *s );
639     void emergencyStopCmd( );
640
641     void readCVCmd( char *s );
642     void writeCVByteCmd( char *s );
643     void writeCVBitCmd( char *s );
644     void writeCVByteMainCmd( char *s );
645     void writeCVBitMainCmd( char *s );
646
647     void writeDccPacketMainCmd( char *s );
648     void writeDccPacketProgCmd( char *s );
649
650     void setTrackOptionCmd( char *s );
651     void turnPowerOnAllCmd( );
652     void turnPowerOnMainCmd( );
653     void turnPowerOnProgCmd( );
654     void turnPowerOffAllCmd( );
655
656     void printStatusCmd( char *s );
657     void printTrackCurrentCmd( char *s );
658     void printBaseStationConfigCmd( );
659     void printHelpCmd( );
660     void printVersionInfo( );
661     void printConfiguration( );
662     void printSessionMap( );
663     void printTrackStatusMain( );
664     void printTrackStatusProg( );
665
666     void printDccLogCommand( char *s );
667
668     private:
669
670     LcsBaseStationLocoSession *locoSessions = nullptr;
671     LcsBaseStationDccTrack    *mainTrack    = nullptr;
672     LcsBaseStationDccTrack    *progTrack    = nullptr;
673 };
674
675 #endif
```

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

using namespace LCS;

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

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

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

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

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

    return ( ALL_OK );
}

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

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

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

        switch ( s[ charIndex ] ) {

            case '<': {

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

            } break;

            case '>': {

                switch ( cmdStr[ 0 ] ) {

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

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

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

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

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

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

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

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

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

                    case '?': printHelpCmd( ); break;

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

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

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

                charIndex ++;

            } break;

            default: {

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

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

    uint16_t  cabId = NIL_CAB_ID;
    uint8_t   sId   = 0;

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

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

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

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

    uint8_t sId = NIL_LOCO_SESSION_ID;

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

    int ret = locoSessions -> releaseSession( sId );

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

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

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

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

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

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

    uint8_t option = 0;

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

        switch ( option ) {

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

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

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

    uint8_t opt = 0;

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

        switch ( opt ) {

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

            case 9: {

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

            } break;

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

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

    printConfiguration( );
}

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

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

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

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

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

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

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

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

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

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

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

    uint8_t followUpMain = DCC_SIG_FOLLOW_UP_NONE;
    uint8_t followUpProg = DCC_SIG_FOLLOW_UP_NONE;

    timeLeftMainTrack -= timeToInterrupt;
    timeLeftProgTrack -= timeToInterrupt;

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

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

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

    CDC::setRepeatingTimerLimit( timeToInterrupt * TICK_IN_MICROSECONDS );

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

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

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

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

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

} // timerCallback

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

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

    CDC::startRepeatingTimer( TICK_IN_MICROSECONDS );
}

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

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

void printLogVal( uint16_t index ) {

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

void printLogData( uint16_t index, uint8_t len ) {

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

uint8_t printLogEntry( uint16_t index ) {

    if ( index < LOG_BUF_SIZE ) {

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

        switch ( logEntryId ) {

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

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

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

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

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

    volatile uint8_t  *timeToInterrupt,
    uint8_t           *followUpAction

    ) {

    switch ( signalState ) {

        case DCC_SIG_CUTOUT_START: {

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

        } break;

        case DCC_SIG_CUTOUT_1: {

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

        } break;

        case DCC_SIG_CUTOUT_2: {

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

        } break;

        case DCC_SIG_CUTOUT_3: {

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

            if ( flags & DT_F_RAILCOM_MODE_ON ) {

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

        } break;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    uint8_t   pBuf[ MAX_DCC_PACKET_SIZE ];
    uint8_t   pLen = 0;

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

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

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

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

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

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

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

    uint8_t pBuf[ MAX_DCC_PACKET_SIZE ];
    uint8_t pLen = 0;

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

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

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

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

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

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

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

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

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

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

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

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

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

## A.4 Block Controller

```
//------------------------------------------------------------------------------------------------------------------
//
// LCS Block Controller - Include file
//
//------------------------------------------------------------------------------------------------------------------
//
// ??? this is a first cut at the block controller software. It remains to be seen what we should factor out
// and use across base station and block controller.
//
//
//
//------------------------------------------------------------------------------------------------------------------
//
// LCS Block Controller
// 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 LcsBlockController_h
#define LcsBlockController_h

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

//------------------------------------------------------------------------------------------------------------------
//
// Ideas how to use the node data:
//
// There is a static data portion, which describes the block. This is data is entered when the block is configured.
//
//  - block ID
//  - block length
//  - block name
//  - previous block(s)
//  - next block(s)
//
//  - number of sections
//  - section lengths
//  - speed level - slow, middle, high ...
//  - support DCC and analog flag
//  - max current limit
//  - periodic time to send data
//  - timeout values of all kinds ?
//
//
// There is a dynamic data portion, which contains the data about the block current state
//
//  - mode ( DCC or analog or off )
//  - actual current
//  - section occupancy
//  - section enter / leave timestamps
//  -
//
// ??? what is retrieved from the dynamic data on a restart ?
//
// The node attributes contains data about how many blocks this node contains ( nodeId + portId -> blockId )
//
// Most of the data is stored in attributes for the port.
//
//
// Finally, there are items that represent commands to the block.
//
//  - emergency stop
//  - switch to DCC or analog mode
//  - block on or off
//  - signals setting
//  - turnout setting
//  - ...

// There are predefined events that the controller node will send.
//
//  - block state change
//  - section occupied
//  - section entered
//  - section left
//  -
//
//
//------------------------------------------------------------------------------------------------------------------


//------------------------------------------------------------------------------------------------------------------
// The block controller 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_BC_CONFIG                   -   DEBUG base station enabled
```

```
99  //      DBG_BC_SETUP                    -   show the setup steps
100 //      DBG_BC_LCS_MSG_INTERFACE        -   show the incoming LCS messages
101 //      DBG_BC_TRACK_POWER_MGMT         -   show the track power measurement data
102 //      DBG_BC_RAILCOM                  -   show the RailCom activity
103 //
104 // The way to use these flags is for example:
105 //
106 //      if (( debugMask & DBG_BC_CONFIG ) && ( debugMask & DBG_BC_SESSION ))
107 //
108 //--------------------------------------------------------------------------------------------------------------
109 enum BlockControllerDebugFlags : uint16_t {
110
111     DBG_BC_CONFIG                   = 1 << 15,      // DEBUG enabled
112     DBG_BC_SETUP                    = 1 << 1,       // show the setup steps
113     DBG_BC_LCS_MSG_INTERFACE        = 1 << 2,       // show the incoming LCS messages
114     DBG_BC_TRACK_POWER_MGMT         = 1 << 3,       // show the track power measurement data
115     DBG_BC_RAILCOM                  = 1 << 4        // show the RailCom activity
116 };
117
118 //--------------------------------------------------------------------------------------------------------------
119 // The base station items for nodeInfo and nodeControl calls .... tbd
120 //
121 // ??? the are mapped in the MEM / NVM range as well as in the USER range.
122 // ??? how to do it consistently and understandably ?
123 //--------------------------------------------------------------------------------------------------------------
124 enum BlockControllerItems : uint8_t {
125
126     BC_ITEM_SET_TRACK_STATE         = 64,
127
128     BC_ITEM_INIT_CURRENT_VAL        = 140,
129     BC_ITEM_LIMIT_CURRENT_VAL       = 141,
130     BC_ITEM_MAX_CURRENT_VAL         = 142,
131     BC_ITEM_ACTUAL_CURRENT_VAL      = 143,
132
133     // thresholds
134     // eventID to send for events ?
135 };
136
137 //--------------------------------------------------------------------------------------------------------------
138 // Base station errors. Note that they need to be in the assigned to the user number range of errors defined
139 // in the LCS runtime library.
140 //
141 //--------------------------------------------------------------------------------------------------------------
142 enum BlockControllerErrors : uint8_t {
143
144     BLOCK_CONTROLLER_ERR_BASE       = 128,
145
146     ERR_MSG_INTERFACE_SETUP         = BLOCK_CONTROLLER_ERR_BASE + 10,
147     ERR_DCC_TRACK_CONFIG            = BLOCK_CONTROLLER_ERR_BASE + 11,
148     ERR_PIN_CONFIG                  = BLOCK_CONTROLLER_ERR_BASE + 12,
149     ERR_TRACK_CONFIG                = BLOCK_CONTROLLER_ERR_BASE + 13,
150
151     ERR_NVM_HW_SETUP                = BLOCK_CONTROLLER_ERR_BASE + 15,
152     ERR_PIO_HW_SETUP                = BLOCK_CONTROLLER_ERR_BASE + 16
153 };
154
155 //--------------------------------------------------------------------------------------------------------------
156 // Setup options to set for the DCC track. They are set when the track object is created.
157 //
158 //  DT_OPT_SERVICE_MODE_TRACK   - The track is a PROG track.
159 //  DT_OPT_RAILCOM              - The track support Railcom detection.
160 //
161 //--------------------------------------------------------------------------------------------------------------
162 enum BlockControllerTrackOptions : uint16_t {
163
164     BT_OPT_DEFAULT_SETTING      = 0,
165     BT_OPT_RAILCOM              = 1 << 1
166 };
167
168 //--------------------------------------------------------------------------------------------------------------
169 // The block track object has a set of flags to indicate its current status.
170 //
171 //  DT_F_POWER_ON               - The track is under power.
172 //  DT_F_POWER_OVERLOAD         - An overload situation was detected.
173 //  DT_F_MEASUREMENT_ON         - The power measurement is enabled.
174 //  DT_F_SERVICE_MODE_ON        - The track is currently in service mode, i.e. is a PROG track.
175 //  DT_F_CUTOUT_MODE_ON         - The track has the cutout generation enabled.
176 //  DT_F_RAILCOM_MODE_ON        - The track has the railcom detect enabled.
177 //  DT_F_RAILCOM_MSG_PENDING    - If railcom is enabled, a received datagram is indicated.
178 //  DT_F_CONFIG_ERROR           - The passed configuration descriptor has invalid options configured.
179 //
180 //--------------------------------------------------------------------------------------------------------------
181 enum TrackFlags : uint16_t {
182
183     BT_F_DEFAULT_SETTING        = 0,
184     BT_F_POWER_ON               = 1 << 0,
185     BT_F_POWER_OVERLOAD         = 1 << 1,
186     BT_F_MEASUREMENT_ON         = 1 << 2,
187     BT_F_CONFIG_ERROR           = 1 << 15
188 };
189
190 //--------------------------------------------------------------------------------------------------------------
191 // The following constants are for the current consumption RMS measurement. The idea is to record the measured
192 // ADC values in a circular buffer, every time a certain amount of milliseconds has passed. This work is done
193 // by the DCC track state machine as part of the power on state.
194 //
195 //--------------------------------------------------------------------------------------------------------------
196 const uint8_t   PWR_SAMPLE_BUF_SIZE             = 64;
197 const uint32_t  PWR_SAMPLE_TIME_INTERVAL_MILLIS  = 16;
```

```
198
199 //-------------------------------------------------------------------------------------------------------------
200 // The track state machine runs at a time interval.
201 //
202 //-------------------------------------------------------------------------------------------------------------
203 const uint32_t TRACK_STATE_TIME_INTERVAL  = 10;
204
205 //-------------------------------------------------------------------------------------------------------------
206 // A block track can be in four states.
207 //
208 //-------------------------------------------------------------------------------------------------------------
209 enum BlockTrackMode : uint16_t {
210
211     BT_MODE_OFF         = 0,
212     BT_MODE_PWM_FWD     = 1,
213     BT_MODE_PWM_REV     = 2,
214     BT_MODE_DCC         = 3
215 };
216
217 //-------------------------------------------------------------------------------------------------------------
218 // The block controller can contain up to four blocks. Each block track is described by the LcsBlockDesc
219 // descriptor. There are the hardware pins sel1Pin1, selPin2, sensePin and uartRxPin. In addition there are
220 // the limits for current consumption values, all specified in milliAmps. The initial current sets the current
221 // consumption limit after the track is turned on. The limit current consumption specifies the actual
222 // configured value that is checked for a track current overload situation. The maximum current defines what
223 // current the power module should never exceed. For the measurements to work, the power module needs to
224 // deliver a voltage that corresponds to the current drawn on the track. The value is measured in milliVolt
225 // per Ampere drawn. Finally, there are threshold times for managing the track overload and restart
226 // capability.
227 //
228 //-------------------------------------------------------------------------------------------------------------
229 struct LcsBlockTrackDesc {
230
231     uint16_t    options;
232     uint8_t     selPin1                         = CDC::UNDEFINED_PIN;
233     uint8_t     selPin2                         = CDC::UNDEFINED_PIN;
234     uint8_t     sensePin                        = CDC::UNDEFINED_PIN;
235
236     uint16_t    pwmFrequency                    = 70;
237     uint16_t    initialTrackMode                = BT_MODE_OFF;
238     uint16_t    initialTrackSpeed               = 0;
239
240     uint16_t    initCurrentMilliAmp             = 0;
241     uint16_t    limitCurrentMilliAmp            = 0;
242     uint16_t    maxCurrentMilliAmp              = 0;
243     uint16_t    milliVoltPerAmp                 = 0;
244
245     uint16_t    startTimeThresholdMillis        = 0;
246     uint16_t    stopTimeThresholdMillis         = 0;
247     uint16_t    overloadTimeThresholdMillis     = 0;
248     uint16_t    overloadEventThreshold          = 0;
249     uint16_t    overloadRestartThreshold        = 0;
250 };
251
252 //-------------------------------------------------------------------------------------------------------------
253 // The "LcsBlockTrack" manages the track of a block. This primarily the power management and control of the
254 // H-Bridge settings. There is one object per track block. At the heart of the object is a state machine that
255 // is executed very often for measuring the power consumption and overload detection logic. The tack can
256 // operate in digital or analog mode. In digital mode, the DCC signal from the LCS bus is routed though to
257 // the H-Bridge, in analog mode a PWM signal is used to set the H-Bridge emitting a PWM signal with a
258 // positive or negative voltage.
259 //
260 //-------------------------------------------------------------------------------------------------------------
261 struct LcsBlockTrack {
262
263     public:
264
265     LcsBlockTrack( );
266
267     uint8_t                     setupBlockTrack( LcsBlockTrackDesc* trackDesc );
268     uint8_t                     setTrackState( uint16_t state );
269     uint8_t                     setTrackMode( uint16_t mode, uint8_t speed = 0 );
270
271     uint16_t                    getFlags( );
272     uint16_t                    getOptions( );
273
274     void                        runTrackStateMachine( );
275
276     void                        powerStart( );
277     void                        powerStop( );
278     bool                        isPowerOn( );
279     bool                        isPowerOverload( );
280
281     void                        setLimitCurrent( uint16_t val );
282     uint16_t                    getLimitCurrent( );
283     uint16_t                    getActualCurrent( );
284     uint16_t                    getInitCurrent( );
285     uint16_t                    getMaxCurrent( );
286     uint16_t                    getRMSCurrent( );
287
288     void                        checkOverload( );
289     void                        powerMeasurement( );
290
291     uint32_t                    getPwrSamplesTaken( );
292     uint16_t                    getPwrSamplesPerSec( );
293
294     void                        printTrackConfig( );
295     void                        printTrackStatus( );
296
```

```
297     private:
298
299     uint16_t                    options                         = BT_OPT_DEFAULT_SETTING;
300     volatile uint16_t           flags                           = BT_F_DEFAULT_SETTING;
301
302     volatile uint16_t           trackState                      = 0;
303     volatile uint16_t           trackMode                       = 0;
304     volatile uint16_t           trackSpeed                      = 0;
305     volatile uint32_t           trackTimeStamp                  = 0;
306     volatile uint8_t            overloadEventCount              = 0;
307     volatile uint8_t            overloadRestartCount            = 0;
308
309     uint8_t                     selPin1                         = CDC::UNDEFINED_PIN;
310     uint8_t                     selPin2                         = CDC::UNDEFINED_PIN;
311     uint8_t                     sensePin                        = CDC::UNDEFINED_PIN;
312
313     uint16_t                    pwmFrequency                    = 0;
314     uint16_t                    initialTrackMode                = 0;
315     uint16_t                    initialTrackSpeed               = 0;
316     uint16_t                    initCurrentMilliAmp             = 0;
317     uint16_t                    limitCurrentMilliAmp            = 0;
318     uint16_t                    maxCurrentMilliAmp              = 0;
319
320     uint16_t                    startTimeThreshold              = 0;
321     uint16_t                    stopTimeThreshold               = 0;
322     uint16_t                    overloadTimeThreshold           = 0;
323     uint16_t                    overloadEventThreshold          = 0;
324     uint16_t                    overloadRestartThreshold        = 0;
325
326     uint16_t                    milliVoltPerAmp                 = 0;
327     uint16_t                    digitsPerAmp                    = 0;
328     volatile uint16_t           actualCurrentDigitValue         = 0;
329     volatile uint16_t           highWaterMarkDigitValue         = 0;
330     volatile uint16_t           limitCurrentDigitValue          = 0;
331
332     volatile uint32_t           totalPwrSamplesTaken            = 0;
333     uint32_t                    lastPwrSampleTimeStamp          = 0;
334
335     uint32_t                    lastPwrSamplePerSecTaken        = 0;
336     uint32_t                    lastPwrSamplePerSecTimeStamp    = 0;
337     uint32_t                    pwrSamplesPerSec                = 0;
338
339     uint8_t                     pwrSampleBufIndex                       = 0;
340     uint16_t                    pwrSampleBuf[ PWR_SAMPLE_BUF_SIZE ]     = { 0 };
341
342 };
343
344 //--------------------------------------------------------------------------------------------------------------
345 // "LcsOccDetect" manages an Occupancy detector extension board. The track power output of a block controller
346 // track is routed to an extension board which implements a set of current detectors. The extension board is
347 // access via the extension I2C bus.
348 //
349 //--------------------------------------------------------------------------------------------------------------
350 struct LcsOccDetect {
351
352     public:
353
354     LcsOccDetect( );
355
356     uint8_t getOccDetectMask( uint16_t *mask );
357
358     private:
359
360     // ??? need to remember the extension board ID.
361
362 };
363
364 //--------------------------------------------------------------------------------------------------------------
365 // "LcsSignal" manages a signal. A block has a signal for each direction to indicate the state of the next
366 // block in a route.
367 //
368 //--------------------------------------------------------------------------------------------------------------
369 struct LcsSignalControl {
370
371     public:
372
373     LcsSignalControl( );
374
375
376     private:
377
378     // ??? need to remember the extension board ID.
379
380 };
381
382 //--------------------------------------------------------------------------------------------------------------
383 // "LcsTurnout" manages the optional turnouts at the end of a block.
384 //
385 //
386 //--------------------------------------------------------------------------------------------------------------
387 struct LcsTurnoutControl {
388
389     public:
390
391     LcsTurnoutControl( );
392
393     private:
394
395     // ??? need to remember the extension board ID.
```

```
396 };
397
398 //-----------------------------------------------------------------------------------------------------------------
399 // "LcsRailComDetect" manages the optional RailCom interface for the block.
400 //
401 //-----------------------------------------------------------------------------------------------------------------
402 struct LcsRailComDetect {
403
404     public:
405
406     LcsRailComDetect( );
407
408     private:
409
410     // ??? need to remember the extension board ID.
411 };
412
413 //-----------------------------------------------------------------------------------------------------------------
414 // "LcsBlockControl" manages a block. A block consists mainly of the tack itself and the optional elements
415 // detectors, signal and turnouts. The block logic, i.e. what to do when the next block is occupied, is
416 // handled here.
417 //
418 //
419 // ??? runs the block logic
420 // ??? how to assign occ detect an signals to the block ?
421 // ??? should message handling be a separate part ?
422 // ??? can we build the control logic in such a way that it is configurable via ITEMs ?
423 //-----------------------------------------------------------------------------------------------------------------
424 struct LcsBlockControl {
425
426     LcsBlockControl(  );
427
428     uint8_t handleLcsRequest( uint8_t *msg );
429
430     private:
431
432     // ??? handles to detect, signal and turnout object.
433
434 };
435
436 //-----------------------------------------------------------------------------------------------------------------
437 // A LCS block controller node can host up to four blocks. This object is the main object that manages the
438 // blocks on the node.
439 //
440 // ??? the node descriptor is an array of block descriptors. They are kept in the NVM ?
441 // ??? manages the LCS messages and forwards them to the target block.
442 //-----------------------------------------------------------------------------------------------------------------
443 struct LcsBlockControllerNode {
444
445     public:
446
447     LcsBlockControllerNode( );
448
449     // ??? setup
450     // ??? targets for the LCS callbacks ?
451
452     private:
453
454     uint16_t    options     = 0;
455     uint16_t    flags       = 0;
456     uint16_t    hwm         = 0;
457
458     LcsBlockControl map[ 4 ];
459 };
460
461 #endif
```

```
//-----------------------------------------------------------------------------------------------------------------
//
// LCS - Block Controller
//
//-----------------------------------------------------------------------------------------------------------------
// This source file contains ...
//
//
//-----------------------------------------------------------------------------------------------------------------
//
// LCS - Block Controller - 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 "LcsBlockController.h"

using namespace LCS;

//-----------------------------------------------------------------------------------------------------------------
// Block Controller global data.
//
//-----------------------------------------------------------------------------------------------------------------
uint16_t                        debugMask;
CDC::CdcConfigDesc              cdcConfig;
LCS::LcsConfigDesc              lcsConfig;
LcsBlockTrackDesc               block1Desc;
LcsBlockTrackDesc               block2Desc;

LcsBlockTrack                   *block1 = nullptr;
LcsBlockTrack                   *block2 = nullptr;

// ??? other BC specific global data ...

//-----------------------------------------------------------------------------------------------------------------
// 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
// DIO5 as shown below.
//
//      cdcConfig.DIO_PIN_0     -> undefined
//      cdcConfig.DIO_PIN_1     -> undefined
//      cdcConfig.DIO_PIN_2     -> Select-0-1
//      cdcConfig.DIO_PIN_3     -> Select-0-2
//      cdcConfig.DIO_PIN_4     -> Select-1-1
//      cdcConfig.DIO_PIN_5     -> Select-1-2
//      cdcConfig.DIO_PIN_6     -> undefined
//      cdcConfig.DIO_PIN_7     -> Cut-Signal
//
// Current mapping: Dual Block Controller Board B.00.01 - PICO - newest version.
//
//      cdcConfig.DIO_PIN_2     = 21;
//      cdcConfig.DIO_PIN_3     = 20;
//      cdcConfig.DIO_PIN_4     = 19;
//      cdcConfig.DIO_PIN_5     = 18;
//      cdcConfig.DIO_PIN_7     = 4;
//
// 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.
//-----------------------------------------------------------------------------------------------------------------
uint8_t 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_2             = 21;
    cdcConfig.DIO_PIN_3             = 20;
    cdcConfig.DIO_PIN_4             = 19;
    cdcConfig.DIO_PIN_5             = 18;
    cdcConfig.DIO_PIN_7             = 4;

    cdcConfig.PWM_PIN_0             = 21;
    cdcConfig.PWM_PIN_1             = 20;
```

```
    cdcConfig.PWM_PIN_2             = 19;
    cdcConfig.PWM_PIN_3             = 18;

    // ??? more PWM channels ?

    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;

    return( ALL_OK );
}

uint8_t setupBlockDesc1( ) {

    block1Desc.options                          = 0;
    block1Desc.selPin1                          = cdcConfig.PWM_PIN_0;
    block1Desc.selPin2                          = cdcConfig.PWM_PIN_1;
    block1Desc.sensePin                         = cdcConfig.ADC_PIN_0;

    block1Desc.pwmFrequency                     = 70;

    block1Desc.initCurrentMilliAmp              = 500;
    block1Desc.limitCurrentMilliAmp             = 1500;
    block1Desc.maxCurrentMilliAmp               = 2000;
    block1Desc.milliVoltPerAmp                  = 100 * 11; // ??? opAmp has Factor eleven ...

    block1Desc.startTimeThresholdMillis         = 1000;
    block1Desc.stopTimeThresholdMillis          = 500;
    block1Desc.overloadTimeThresholdMillis      = 500;
    block1Desc.overloadEventThreshold           = 10;
    block1Desc.overloadRestartThreshold         = 5;

    return( ALL_OK );
}

uint8_t setupBlockDesc2( ) {

    block1Desc.options                          = 0;
    block1Desc.selPin1                          = cdcConfig.PWM_PIN_2;
    block1Desc.selPin2                          = cdcConfig.PWM_PIN_3;
    block1Desc.sensePin                         = cdcConfig.ADC_PIN_1;

    block1Desc.pwmFrequency                     = 70;

    block1Desc.initCurrentMilliAmp              = 500;
    block1Desc.limitCurrentMilliAmp             = 1500;
    block1Desc.maxCurrentMilliAmp               = 2000;
    block1Desc.milliVoltPerAmp                  = 100 * 11; // ??? opAmp has Factor eleven ...

    block1Desc.startTimeThresholdMillis         = 1000;
    block1Desc.stopTimeThresholdMillis          = 500;
    block1Desc.overloadTimeThresholdMillis      = 500;
    block1Desc.overloadEventThreshold           = 10;
    block1Desc.overloadRestartThreshold         = 5;

    return( ALL_OK );
}

//-----------------------------------------------------------------------------------------------------------
// 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 );
}

// ??? pass the callbacks to block controller logic ?
// ??? the object must implement these methods...

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

```
198 // The node and port initialization callback.
199 //
200 // ??? when we know what ports we actually need / use, disable the rest of the ports.
201 // ??? the number of ports / blocks should be note in the block descriptor.
202 // ??? invoke the configured block reset method in the block controller logic object...
203 //-------------------------------------------------------------------------------------------------------------
204 uint8_t lcsInitCallback( uint16_t npId ) {
205 
206     switch ( npId & 0xF ) {
207 
208         case 0:     printf( "Node Init Callback: 0x%x\n", npId >> 4     ); break;
209         default:    printf( "Port Init Callback: 0x%x\n", npId &  0xF    );
210     }
211 
212     return( ALL_OK );
213 }
214 
215 //-------------------------------------------------------------------------------------------------------------
216 // The node or port reset callback.
217 //
218 // ??? invoke the configured block reset method in the block controller logic object...
219 //-------------------------------------------------------------------------------------------------------------
220 uint8_t lcsResetCallback( uint16_t npId ) {
221 
222     switch ( npId & 0xF ) {
223 
224         case 0:     printf( "Node Reset Callback: 0x%x\n", npId >> 4     ); break;
225         default:    printf( "Port Reset Callback: 0x%x\n", npId &  0xF    );
226     }
227 
228     return( ALL_OK );
229 }
230 
231 //-------------------------------------------------------------------------------------------------------------
232 // The node or port power fail callback.
233 //
234 // ??? invoke the configured block pfail method in the block controller logic object...
235 //-------------------------------------------------------------------------------------------------------------
236 uint8_t lcsPfailCallback( uint16_t npId ) {
237 
238     switch ( npId & 0xF ) {
239 
240         case 0:     printf( "Node Power Fail Callback: 0x%x\n", npId >> 4     ); break;
241         default:    printf( "Port Power Fail Callback: 0x%x\n", npId &  0xF    );
242     }
243 
244     return( ALL_OK );
245 }
246 
247 //-------------------------------------------------------------------------------------------------------------
248 // Other LCS message callbacks. All we do is to list their invocation. ( for now )
249 //
250 // ??? this should go out ....
251 //-------------------------------------------------------------------------------------------------------------
252 uint8_t lcsMsgCallback( uint8_t *msg ) {
253 
254     printf( "MsgCallback: ", msg  );
255     printLcsMsg( msg );
256     return( ALL_OK );
257 }
258 
259 //-------------------------------------------------------------------------------------------------------------
260 // When the base station node receives a request with an item defined in the user item range or the base
261 // station itself issues such a request, the defined callback is invoked.
262 //
263 // ??? pass to the block controller logic...
264 //-------------------------------------------------------------------------------------------------------------
265 uint8_t lcsReqCallback( uint8_t npId, uint8_t item, uint16_t *arg1, uint16_t *arg2 ) {
266 
267     printf( "REQ callback: npId: 0x%x, item: %d", npId, item );
268     if ( arg1 != nullptr ) printf( ", arg1: %d, ", *arg1 ); else printf( ", arg1: null" );
269     if ( arg2 != nullptr ) printf( ", arg2: %d, ", *arg2 ); else printf( ", arg2: null" );
270 
271     switch( item ) {
272 
273         case 64: {
274 
275             uint16_t port = npId & 0xF;
276 
277             if      ( port == 1 ) block1 -> setTrackMode( *arg1 & 0xFF, *arg2 & 0xFF );
278             else if ( port == 2 ) block2 -> setTrackMode( *arg1 & 0xFF, *arg2 & 0xFF );
279 
280         } break;
281 
282         default: {
283 
284         }
285     }
286 
287     return( ALL_OK );
288 }
289 
290 //-------------------------------------------------------------------------------------------------------------
291 // When the base station gets a reply message for a request previously sent, this callback is invoked.
292 //
293 // ??? pass to the block that requested...
294 //-------------------------------------------------------------------------------------------------------------
295 uint8_t lcsRepCallback( uint8_t npId, uint8_t item, uint16_t arg1, uint16_t arg2, uint8_t ret ) {
296 
```

```
297     printf( "REP callback: npId: 0x%x, item: %d, arg1: %d, arg2: %d, ret: %d ", npId, item , arg1, arg2, ret );
298     return( ALL_OK );
299 }
300 
301 //------------------------------------------------------------------------------------------------------------------
302 // For any event on the LCS system that the block controller is interested in, this callback is invoked.
303 //
304 // ??? what events to listen to ? where are they configured/set ?
305 //------------------------------------------------------------------------------------------------------------------
306 uint8_t lcsEventCallback( uint16_t npId, uint16_t eId, uint8_t eAction, uint16_t eData ) {
307 
308     printf( "Event: npId: 0x%x, eId: %d, eAction: %d, eData: %d\n", npId, eId, eAction, eData );
309     return( ALL_OK );
310 }
311 
312 //------------------------------------------------------------------------------------------------------------------
313 // We need to run the track state machines on a periodic basis.
314 //
315 //
316 // ??? we actually need an array of track machines ?
317 // ??? or should we register each one individually ?
318 //------------------------------------------------------------------------------------------------------------------
319 uint8_t trackStateMachine( ) {
320 
321     if ( block1 != nullptr ) block1 -> runTrackStateMachine( );
322     if ( block2 != nullptr ) block2 -> runTrackStateMachine( );
323     return( ALL_OK );
324 }
325 
326 //------------------------------------------------------------------------------------------------------------------
327 // Init the Runtime.
328 //
329 //------------------------------------------------------------------------------------------------------------------
330 uint8_t initLcsRuntime( ) {
331 
332     printf( "LCS Block Controller\n" );
333     printf( "initLcsRuntime\n" );
334 
335     setupConfigInfo( );
336 
337     uint8_t rStat = initRuntime( &lcsConfig, &cdcConfig );
338 
339     CDC::printConfigInfo( &cdcConfig );
340     return( printStatus( rStat ));
341 }
342 
343 //------------------------------------------------------------------------------------------------------------------
344 // After the initial setup of the runtime library, the callback are registered.
345 //
346 //------------------------------------------------------------------------------------------------------------------
347 uint8_t registerCallbacks( ) {
348 
349     printf( "Registering Callbacks\n" );
350 
351     registerLcsMsgCallback( lcsMsgCallback );
352     registerInitCallback( lcsInitCallback );
353     registerResetCallback( lcsResetCallback );
354     registerPfailCallback( lcsPfailCallback );
355     registerReqCallback( lcsReqCallback );
356     registerRepCallback( lcsRepCallback );
357     registerEventCallback( lcsEventCallback );
358     registerTaskCallback( trackStateMachine, TRACK_STATE_TIME_INTERVAL );
359 
360     return( printStatus( ALL_OK ));
361 }
362 
363 //------------------------------------------------------------------------------------------------------------------
364 // Fire up the base station. First all base station modules are initialized. If this is OK, the DCC tack
365 // signal generation is enabled, i.e. the interrupt driven DCC packet broadcasting starts. Finally, the
366 // track power is turned on and we give control to the LCS runtime for processing events and requests.
367 //
368 //------------------------------------------------------------------------------------------------------------------
369 uint8_t startBlockController( ) {
370 
371     printf( "Start Block controller\n" );
372 
373     uint8_t rStat = ALL_OK;
374 
375     block1 = new LcsBlockTrack( );
376     block2 = new LcsBlockTrack( );
377 
378     printf( "Configure Block 1\n" );
379     rStat = block1 -> setupBlockTrack( &block1Desc );
380     if ( rStat != ALL_OK ) printStatus( rStat );
381 
382     printf( "Configure Block 2\n" );
383     rStat = block1 -> setupBlockTrack( &block2Desc );
384     if ( rStat != ALL_OK ) printStatus( rStat );
385 
386     #if 0
387     // ??? for the quick test ....
388     printf( "Configure PWM pins\n" );
389     rStat = CDC::configurePwm( cdcConfig.PWM_PIN_0, 70 );
390     if ( rStat != ALL_OK ) printStatus( rStat );
391 
392     rStat = CDC::configurePwm( cdcConfig.PWM_PIN_1, 70 );
393     if ( rStat != ALL_OK ) printStatus( rStat );
394 
395     rStat = CDC::configurePwm( cdcConfig.PWM_PIN_2, 70 );
```

```
396     if ( rStat != ALL_OK ) printStatus( rStat );
397
398     rStat = CDC::configurePwm( cdcConfig.PWM_PIN_3, 70 );
399     if ( rStat != ALL_OK ) printStatus( rStat );
400
401     printf( "Set output to pins\n" );
402     rStat = CDC::writePwm(cdcConfig.PWM_PIN_0, 255 );
403     if ( rStat != ALL_OK ) printStatus( rStat );
404     rStat = CDC::writePwm(cdcConfig.PWM_PIN_1, 255 );
405     if ( rStat != ALL_OK ) printStatus( rStat );
406     rStat = CDC::writePwm(cdcConfig.PWM_PIN_2, 0 );
407     if ( rStat != ALL_OK ) printStatus( rStat );
408     rStat = CDC::writePwm(cdcConfig.PWM_PIN_3, 120 );
409     if ( rStat != ALL_OK ) printStatus( rStat );
410     #endif
411
412     printf( "Block 1 Config:" );
413     if ( block1 != nullptr ) block1 -> printTrackConfig( );
414
415     printf( "Block 2 Config:" );
416     if ( block2 != nullptr ) block2 -> printTrackConfig( );
417
418     if ( rStat == ALL_OK ) {
419
420         printf( "Ready...\n" );
421         startRuntime( );
422     }
423
424     return( ALL_OK );
425 }
426
427 //-----------------------------------------------------------------------------------------------------------------
428 // The main program. Setup the runtime, register the callbacks, and get the show on the road.
429 //
430 //-----------------------------------------------------------------------------------------------------------------
431 int main( ) {
432
433     uint8_t rStat = ALL_OK;
434
435     if ( rStat == ALL_OK ) rStat = initLcsRuntime( );
436     if ( rStat == ALL_OK ) rStat = registerCallbacks( );
437     if ( rStat == ALL_OK ) return( startBlockController( ));
438 }
```

```
//-------------------------------------------------------------------------------------------------------------
//
// LCS Block Controller - Control Logic
//
//-------------------------------------------------------------------------------------------------------------
//
// LCS Block Controller
// 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 "LcsBlockController.h"

// ??? contains the main code, the setup, the message handler, etc.

using namespace LCS;

//-------------------------------------------------------------------------------------------------------------
//
//
//-------------------------------------------------------------------------------------------------------------
LcsBlockControl::LcsBlockControl(  ) {



}

//-------------------------------------------------------------------------------------------------------------
//
//
//-------------------------------------------------------------------------------------------------------------
 uint8_t LcsBlockControl::handleLcsRequest( uint8_t *msg ) {

    switch ( msg[ 0 ] ) {

        // ??? define a few request for testing ...

        default: ;
    }

    return( ALL_OK );
 }
```

```
//--------------------------------------------------------------------------------------------------------------
//
// LCS Block Controller - Block Track
//
//--------------------------------------------------------------------------------------------------------------
// The Block Controller track power module manages the track of the block. Each block is associated with a
// port on the node and the this object essentially controls the H-Bridge. At the heart is a state machine
// manages the power state. The power consumption is measured on a periodic base and an overload leads to
// switching the block track off.
//
// The block can operate in two basic modes. The first mode is the DCC mode. The control select pins are set
// to route the DCC signal from the LCS bus to the H-Bridge. The second mode is the analog mode. There are
// two sub modes, which are forward and reverse PWM setting.
//
//--------------------------------------------------------------------------------------------------------------
//
// LCS Block Controller - Block Track
// 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 "LcsBlockController.h"
#include <math.h>

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

//--------------------------------------------------------------------------------------------------------------
// The Block Track Object local definitions. The hardware lower layers can be found in controller dependent
// code (CDC) layer.
//
//--------------------------------------------------------------------------------------------------------------
namespace {

using namespace LCS;

//--------------------------------------------------------------------------------------------------------------
// Block controller 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;

//--------------------------------------------------------------------------------------------------------------
// Block track power management is a state machine managing the setting of the power track. Maximum values
// for the 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 {

    TRACK_POWER_OFF       = 0,
    TRACK_POWER_ON        = 1,
    TRACK_POWER_OVERLOAD  = 2,
    TRACK_POWER_START1    = 3,
    TRACK_POWER_START2    = 4,
    TRACK_POWER_STOP1     = 5,
    TRACK_POWER_STOP2     = 6
};

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

```

```
99      return (( val >= lower ) && ( val <= upper ));
100 }
101
102 //-----------------------------------------------------------------------------------------------------------
103 // Conversion functions between milliAmps and digit values as report4de by the analog to digital converter
104 // hardware. For a better precision, the formula uses 32 bit computation and stores the result back in a
105 // 16 bit quantity.
106 //
107 //-----------------------------------------------------------------------------------------------------------
108 uint16_t milliAmpToDigitValue( uint16_t milliAmp, uint16_t digitsPerAmp ) {
109
110     #if 0
111     uint32_t mA = milliAmp;
112     uint32_t dPA = digitsPerAmp;
113     return (( uint16_t ) ( mA * dPA / 1000 ));
114     #endif
115
116     return ((uint16_t) ((((uint32_t) milliAmp ) * ((uint32_t) digitsPerAmp )) / 1000 ));
117 }
118
119 uint16_t digitValueToMilliAmp( uint16_t digitValue, uint16_t digitsPerAmp ) {
120
121     #if 0
122     uint32_t dV = digitValue;
123     uint32_t dPA = digitsPerAmp;
124     return ((uint16_t)( dV * 1000 / dPA ));
125     #endif
126
127     return ((uint16_t) ((((uint32_t) digitValue ) * 1000 ) / ((uint32_t) digitsPerAmp )));
128 }
129
130 }; // namespace
131
132
133 //===========================================================================================================
134 //===========================================================================================================
135 //
136 // Object part.
137 //
138 //===========================================================================================================
139 //===========================================================================================================
140
141
142 //-----------------------------------------------------------------------------------------------------------
143 // Object instance section. The DccTrack constructor. Nothing to do so far.
144 //
145 //-----------------------------------------------------------------------------------------------------------
146 LcsBlockTrack::LcsBlockTrack( ) { }
147
148
149 //-----------------------------------------------------------------------------------------------------------
150 // "setupDccTrack" performs the setup tasks for the DCC track.  We will configure the hardware, the DCC
151 // packet options such as preamble and postamble length, the initial state machine state current consumption
152 // limit and load the initial packet into the active buffer. There is quite a list of parameters and options
153 // that can be set. This routine does the following checking:
154 //
155 //    - the pins used in the CDC layer must be a pair ( for atmega controllers ).
156 //    - the sensePin must be an analog input pin.
157 //    - if the track is a service track, cutout and RailCom are not supported.
158 //    - if RailCom is set, Cutout must be set too.
159 //    - the initial current limit consumption setting must be less than the current limit setting.
160 //    - the current limit setting must be less than the maximum current limit setting.
161 //
162 // Once the DCC track object is initialized, the last thing to do is to remember the object instance in the
163 // file static variables. This is necessary for the interrupt handlers to work. If any of the checks fails,
164 // the flag field will have the error bit set.
165 //
166 //-----------------------------------------------------------------------------------------------------------
167 uint8_t LcsBlockTrack::setupBlockTrack( LcsBlockTrackDesc* trackDesc ) {
168
169     if ((  trackDesc -> selPin1     == CDC::UNDEFINED_PIN ) ||
170         (  trackDesc -> selPin2     == CDC::UNDEFINED_PIN ) ||
171         (  trackDesc -> sensePin    == CDC::UNDEFINED_PIN )) {
172
173         flags = BT_F_CONFIG_ERROR;
174         return ( ERR_PIN_CONFIG );
175     }
176
177     if (( trackDesc -> initCurrentMilliAmp  > trackDesc -> limitCurrentMilliAmp )                  ||
178         ( trackDesc -> limitCurrentMilliAmp > trackDesc -> maxCurrentMilliAmp )                    ||
179         ( trackDesc -> startTimeThresholdMillis > MAX_START_TIME_THRESHOLD_MILLIS )                ||
180         ( trackDesc -> stopTimeThresholdMillis > MAX_STOP_TIME_THRESHOLD_MILLIS )                  ||
181         ( trackDesc -> overloadTimeThresholdMillis > MAX_OVERLOAD_TIME_THRESHOLD_MILLIS )          ||
182         ( trackDesc -> overloadEventThreshold > MAX_OVERLOAD_EVENT_COUNT )                         ||
183         ( trackDesc -> overloadRestartThreshold > MAX_OVERLOAD_RESTART_COUNT )
184         ) {
185
186         flags = BT_F_CONFIG_ERROR;
187         return ( ERR_TRACK_CONFIG );
188     }
189
190     trackState                  = TRACK_POWER_OFF;
191     flags                       = BT_F_DEFAULT_SETTING;
192     options                     = trackDesc -> options;
193     selPin1                     = trackDesc -> selPin1;
194     selPin2                     = trackDesc -> selPin2;
195     sensePin                    = trackDesc -> sensePin;
196     pwmFrequency                = trackDesc -> pwmFrequency;
197     initialTrackMode            = trackDesc -> initialTrackMode;
```

```
    initialTrackSpeed           = trackDesc -> initialTrackSpeed;
    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 );
    actualCurrentDigitValue     = 0;
    totalPwrSamplesTaken        = 0;
    lastPwrSamplePerSecTaken    = 0;
    pwrSamplesPerSec            = 0;

    uint8_t rStat = CDC::configurePwm( selPin1, pwmFrequency );
    if ( rStat == ALL_OK ) rStat = CDC::configurePwm( selPin2, pwmFrequency );
    if ( rStat == ALL_OK ) rStat = CDC::configureAdc( sensePin );
    if ( rStat == ALL_OK ) rStat = setTrackMode( initialTrackMode, initialTrackSpeed );

    if ( rStat != ALL_OK ) flags |= BT_F_CONFIG_ERROR;
    return ( rStat );
}

//-------------------------------------------------------------------------------------------------------
// "setBlockTrackState" sets the control output pins for the block controller H-Bridge. The H-Bridge has two
// half bridge control in puts and an enable input. The setting of these three inputs are encoded into a
// pair of select pins with the following settings:
//
//      BT_MODE_OFF         -   both select pins are set to zero. This leads putting the H-Bridge into a high
//                              impedance state.
//
//      BT_MODE_PWM_FWD     -   select pin 1 is set to the PWM signal, select pin 2 is set to zero. The
//                              speed parameter specifies the duty cycle on a range from 0 to 255.
//
//      BT_MODE_PWM_RE      -   select pin 1 is set to zero, select pin 2 is set to the PWM signal. The
//                              speed parameter specifies the duty cycle on a range from 0 to 255.
//
//      BT_MODE_DCC         -   both select pins are set to one.
//
//-------------------------------------------------------------------------------------------------------
uint8_t LcsBlockTrack::setTrackMode( uint16_t state, uint8_t speed ) {

    if (( debugMask & DBG_BC_CONFIG ) && ( debugMask & DBG_BC_TRACK_POWER_MGMT )) {

        printf( "setTrackMode: mode: %d, speed: %d\n", state, speed );
    }

    uint8_t rStat;

    switch( state ) {

        case BT_MODE_PWM_FWD: {

            rStat = CDC::writePwm( selPin1, speed );
            if ( rStat == ALL_OK ) rStat = CDC::writePwm( selPin2, 0 );
            return( rStat );

        } break;

        case BT_MODE_PWM_REV: {

            rStat = CDC::writePwm( selPin1, 0 );
            if ( rStat == ALL_OK ) rStat = CDC::writePwm( selPin2, speed );
            return( rStat );

        } break;

        case BT_MODE_DCC: {

            rStat = CDC::writePwm( selPin1, 255 );
            if ( rStat == ALL_OK ) rStat = CDC::writePwm( selPin2, 255 );
            return( rStat );

        } break;

        case BT_MODE_OFF:   default: {

            rStat = CDC::writePwm( selPin1, 0 );
            if ( rStat == ALL_OK ) rStat = CDC::writePwm( selPin2, 0 );
            return( rStat );

        } break;
    }
}

//-------------------------------------------------------------------------------------------------------
// Track power is not just a matter of turning power on or off. To address all the requirements of the DCC
// standard, the track is managed by a state machine that implements the start and stop sequences. They will
// also be executed in analog mode. It is important that we do not really block the progress of the entire
// block controller, 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.
//
```

```
297 //  TRACK_POWER_START1    - this is the first state of a start sequence. When the track should be powered
298 //                          on, the first activity is to set the status flags and enable the power module.
299 //                          We set the power module current consumption to the initial limit configured.
300 //                          The next state is TRACK_POWER_START2.
301 //
302 //  TRACK_POWER_START2    - we stay in this state until the threshold time has passed. Once the threshold
303 //                          is reached, the current consumption limit is set to the configured limit.
304 //                          Then we move on to TRACK_POWER_ON.
305 //
306 //  TRACK_POWER_ON        - this is the state when power is on and things are running normal. An overload
307 //                          situation is set by the current measurement routines through setting the
308 //                          overload status flag. We make sure that we have seen a couple of overloads
309 //                          in a row before taking action which is to turn power off and set the
310 //                          TRACK_POWER_OVERLOAD state. Otherwise we stay in this state.
311 //
312 //  TRACK_POWER_OVERLOAD  - with power turned off, we stay in this state until the threshold time has
313 //                          passed. If passed, the overload restart count is incremented and checked for
314 //                          its threshold. If reached, we have tried to restart several times and failed.
315 //                          The track state becomes TRACK_POWER_STOP1, something is wrong on the track.
316 //                          If not, we move on to TRACK_POWER_START1.
317 //
318 //  TRACK_POWER_STOP1     - this state initiates a shutdown sequence. We disable the power module, set
319 //                          status flags and advance to the TRACK_POWER_STOP2 state.
320 //
321 //  TRACK_POWER_STOP2     - we stay in this state until the configured threshold has passed. Then we move
322 //                          on to TRACK_POWER_OFF. The key reason for this time delay is to implement
323 //                          the requirement that track turned off and perhaps switched to another mode,
324 //                          should be powerless for one second. Switch track modes becomes simply a matter
325 //                          of stopping and then starting again.
326 //
327 //  TRACK_POWER_OFF       - the track is disabled. We just stay in this state until the state is set to
328 //                          a different state from outside.
329 //
330 // During the power on state, we append the actual current measurement value to a circular buffer when the
331 // time interval for this kind of measurement has passed. The idea is to measure the samples at a more or
332 // less constant interval rate and compute the power consumption RMS value from the data in the buffer when
333 // requested. In the interest of minimizing the controller load, the calculation is done in digit values
334 // the result is presented in then in milliAmps.
335 //
336 //-------------------------------------------------------------------------------------------------------
337 void LcsBlockTrack::runTrackStateMachine( ) {
338
339     switch ( trackState ) {
340
341         case TRACK_POWER_START1: {
342
343             // ??? do we need a way to check for overload during this initial phase, just like we do when ON ?
344
345             trackTimeStamp          = CDC::getMillis( );
346             flags                   |= BT_F_POWER_ON;
347             flags                   &= ~BT_F_POWER_OVERLOAD;
348             flags                   &= ~BT_F_MEASUREMENT_ON;
349             limitCurrentDigitValue  = milliAmpToDigitValue( initCurrentMilliAmp, digitsPerAmp );
350
351             setTrackMode( initialTrackMode, 0 );
352             trackState = TRACK_POWER_START2;
353
354         }  break;
355
356         case TRACK_POWER_START2: {
357
358             if (( CDC::getMillis( ) - trackTimeStamp ) > startTimeThreshold ) {
359
360                 highWaterMarkDigitValue = 0;
361                 actualCurrentDigitValue = 0;
362                 overloadRestartCount    = 0;
363                 overloadEventCount      = 0;
364                 flags                   |= BT_F_POWER_ON | BT_F_MEASUREMENT_ON;
365                 limitCurrentDigitValue  = milliAmpToDigitValue( limitCurrentMilliAmp, digitsPerAmp );
366
367                 trackState = TRACK_POWER_ON;
368             }
369
370         } break;
371
372         case TRACK_POWER_ON: {
373
374             if (( CDC::getMillis( ) - lastPwrSampleTimeStamp ) > PWR_SAMPLE_TIME_INTERVAL_MILLIS ) {
375
376                 pwrSampleBuf[ pwrSampleBufIndex % TRACK_POWER_ON ] = actualCurrentDigitValue;
377                 pwrSampleBufIndex ++;
378                 lastPwrSampleTimeStamp = CDC::getMillis( );
379             }
380
381             if (( CDC::getMillis( ) - lastPwrSamplePerSecTimeStamp ) > 1000 ) {
382
383                 pwrSamplesPerSec          = totalPwrSamplesTaken - lastPwrSamplePerSecTaken;
384                 lastPwrSamplePerSecTaken  = totalPwrSamplesTaken;
385                 lastPwrSamplePerSecTimeStamp = CDC::getMillis( );
386             }
387
388             if ( flags & BT_F_POWER_OVERLOAD ) {
389
390                 overloadEventCount  ++;
391
392                 if ( overloadEventCount > overloadEventThreshold ) {
393
394                     if (( debugMask & DBG_BC_CONFIG ) && ( debugMask & DBG_BC_TRACK_POWER_MGMT )) {
395
```

```
396                         printf( "Overload detected: " );
397
398                         #if 0
399                         printf( "(hwm(mA): %d : limit(mA): %d )\n",
400                                 digitValueToMilliAmp( highWaterMarkDigitValue, digitsPerAmp ),
401                                 digitValueToMilliAmp( limitCurrentDigitValue, digitsPerAmp ));
402                         #else
403                         printf( "(hwm(dVal): %d  : limit(dVal): %d )\n", highWaterMarkDigitValue, limitCurrentDigitValue );
404                         #endif
405                     }
406
407                     trackTimeStamp  = CDC::getMillis( );
408                     flags           |= BT_F_POWER_OVERLOAD;
409                     flags           &= ~BT_F_POWER_ON;
410                     flags           &= ~BT_F_MEASUREMENT_ON;
411
412                     setTrackMode( BT_MODE_OFF );
413                     trackState = TRACK_POWER_OVERLOAD;
414                 }
415             }
416
417         }  break;
418
419         case TRACK_POWER_OVERLOAD: {
420
421             if ( CDC::getMillis( ) - trackTimeStamp > overloadTimeThreshold ) {
422
423                 overloadRestartCount ++;
424
425                 if ( overloadRestartCount > overloadRestartThreshold ) {
426
427                     if (( debugMask & DBG_BC_CONFIG ) && ( debugMask & DBG_BC_TRACK_POWER_MGMT )) {
428
429                         printf( "Overload restart failed, Cnt:%d\n", overloadRestartCount );
430                     }
431
432                     trackState = TRACK_POWER_STOP1;
433                 }
434                 else trackState = TRACK_POWER_START1;
435             }
436
437         }  break;
438
439         case TRACK_POWER_STOP1: {
440
441             trackTimeStamp  = CDC::getMillis( );
442             flags           &= ~BT_F_POWER_ON;
443             flags           &= ~BT_F_POWER_OVERLOAD;
444             flags           &= ~BT_F_MEASUREMENT_ON;
445
446             setTrackMode( BT_MODE_OFF );
447             trackState = TRACK_POWER_STOP2;
448
449         }  break;
450
451         case TRACK_POWER_STOP2: {
452
453             if ( CDC::getMillis( ) - trackTimeStamp > stopTimeThreshold ) trackState = TRACK_POWER_OFF;
454
455         } break;
456
457         case TRACK_POWER_OFF: {
458
459         } break;
460     }
461 }
462
463 //-----------------------------------------------------------------------------------------------
464 // Some getter functions. Straightforward.
465 //
466 //-----------------------------------------------------------------------------------------------
467 uint16_t LcsBlockTrack::getFlags( ) {
468
469     return ( flags );
470 }
471
472 uint16_t LcsBlockTrack::getOptions( ) {
473
474     return ( options );
475 }
476
477 uint32_t LcsBlockTrack::getPwrSamplesTaken( ) {
478
479     return ( totalPwrSamplesTaken );
480 }
481
482 uint16_t LcsBlockTrack::getPwrSamplesPerSec( ) {
483
484     return ( pwrSamplesPerSec );
485 }
486
487 bool LcsBlockTrack::isPowerOn( ) {
488
489     return ( flags & BT_F_POWER_ON );
490 }
491
492 bool LcsBlockTrack::isPowerOverload( ) {
493
494     return ( flags & BT_F_POWER_OVERLOAD );
```

```
495 }
496
497 //----------------------------------------------------------------------------------------------------------------
498 // Track power management functions. The actual state of track power is kept in the track status field
499 // and can be queried or set by setting the respective flag. Starting and stopping track power is done by
500 // setting the respective START or STOP state.
501 //
502 //----------------------------------------------------------------------------------------------------------------
503 void LcsBlockTrack::powerStart( ) {
504
505     trackState = TRACK_POWER_START1;
506 }
507
508 void LcsBlockTrack::powerStop( ) {
509
510     trackState = TRACK_POWER_STOP1;
511 }
512
513 //----------------------------------------------------------------------------------------------------------------
514 // Power Consumption Management. There are two key values. The first is the actual current consumption as
515 // measured by the ADC hardware on each ZERO DCC bit. This value is used to do the power overload checking.
516 // The second value is the high water mark built from these measurements. This values is used for the DCC
517 // decoder programming logic. The high water mark will be set to zero before collecting measurements. All
518 // measurement values are actually ADC digit values for performance reason. Only on limit setting and external
519 // data access are these values converted from and to milliAmps.
520 //
521 //----------------------------------------------------------------------------------------------------------------
522 uint16_t LcsBlockTrack::getLimitCurrent( ) {
523
524     return ( limitCurrentMilliAmp );
525 }
526
527 uint16_t LcsBlockTrack::getActualCurrent( ) {
528
529     return ( digitValueToMilliAmp( actualCurrentDigitValue, digitsPerAmp ));
530 }
531
532 uint16_t LcsBlockTrack::getInitCurrent( ) {
533
534     return ( initCurrentMilliAmp );
535 }
536
537 uint16_t LcsBlockTrack::getMaxCurrent( ) {
538
539     return ( maxCurrentMilliAmp );
540 }
541
542 void LcsBlockTrack::setLimitCurrent( uint16_t val ) {
543
544     if      ( val < initCurrentMilliAmp )  val = initCurrentMilliAmp;
545     else if ( val > maxCurrentMilliAmp  )  val = maxCurrentMilliAmp;
546
547     limitCurrentMilliAmp    = val;
548     limitCurrentDigitValue  = milliAmpToDigitValue( val, digitsPerAmp );
549 }
550
551 //----------------------------------------------------------------------------------------------------------------
552 // The "getRMSCurrent" function returns the power consumption based on the samples taken and stored in the
553 // sample buffer. The function computes the square root of the sum of the squares of the array elements. The
554 // result is returned in milliAmps. Note that our measurement is based on unsigned 16-bit quantities that come
555 // from the controller ADC hardware. We compute the RMS based on 16-bit unsigned integers, which compared
556 // to floating point computation is not really precise. However, for our purpose to just show a rough power
557 // consumption, the error should be not a big issue. We will not use RMS values for power overload detection
558 // or decoder ACK detection.
559 //
560 //----------------------------------------------------------------------------------------------------------------
561 uint16_t LcsBlockTrack::getRMSCurrent( ) {
562
563     uint32_t res = 0;
564
565     for ( uint8_t i = 0; i < PWR_SAMPLE_BUF_SIZE; i++ ) res += pwrSampleBuf[ i ] * pwrSampleBuf[ i ];
566
567     return ( digitValueToMilliAmp( sqrt( res / PWR_SAMPLE_BUF_SIZE ), digitsPerAmp ));
568 }
569
570 //----------------------------------------------------------------------------------------------------------------
571 // This function is called whenever we want to measure the power consumption. Typically this routine will be
572 // called from an timer or interrupt handler.
573 //
574 //----------------------------------------------------------------------------------------------------------------
575 void LcsBlockTrack::powerMeasurement( ) {
576
577     if ( flags & BT_F_MEASUREMENT_ON ) {
578
579         actualCurrentDigitValue = CDC::readAdc( sensePin );
580
581         totalPwrSamplesTaken ++;
582
583         if ( actualCurrentDigitValue > highWaterMarkDigitValue ) highWaterMarkDigitValue = actualCurrentDigitValue;
584         if ( actualCurrentDigitValue > limitCurrentDigitValue ) flags |= BT_F_POWER_OVERLOAD;
585     }
586 }
587
588 //----------------------------------------------------------------------------------------------------------------
589 // Print out the DCC Track configuration data. For debugging purposes.
590 //
591 //----------------------------------------------------------------------------------------------------------------
592 void LcsBlockTrack::printTrackConfig( ) {
593
```

```
594     printf( "DccTrack Config: " );
595
596     printf( "Config options: ( 0x%x ) -> ", flags );
597     if ( options & BT_OPT_RAILCOM ) printf( "Railcom " );
598     printf( "\n" );
599
600     printf( "Sel1 Pin: %d, Sel2 Pin: %d, Sensor Pin: %d\n", selPin1, selPin2, sensePin );
601
602     printf( "Initial Block State: %d, speed: %d\n", initialTrackMode, initialTrackSpeed );
603
604     printf( "Current Initial(mA): %d Current Limit(mA): %d Current Max(mA): %d\n",
605             getInitCurrent( ), getLimitCurrent( ), getMaxCurrent( ));
606
607     printf( "milliVoltPerAmp: %d\n", milliVoltPerAmp );
608     printf( "digitsPerAmp: %d\n", digitsPerAmp );
609     printf( "Limit Digit Value: %d\n", limitCurrentDigitValue );
610 }
611
612 //-----------------------------------------------------------------------------------------------------------
613 // Print out the DCC Track status.
614 //
615 //-----------------------------------------------------------------------------------------------------------
616 void LcsBlockTrack::printTrackStatus( ) {
617
618     printf( ", Track Status: ( 0x%x ) -> ", flags );
619
620     if ( flags & BT_F_POWER_ON          ) printf( "PowerOn " );
621     if ( flags & BT_F_POWER_OVERLOAD    ) printf( "PowerOverload " );
622     if ( flags & BT_F_MEASUREMENT_ON    ) printf( "PowerMeasOn " );
623     if ( flags & BT_F_CONFIG_ERROR      ) printf( "ConfigError " );
624     printf( "\n" );
625
626     printf( "Total Power Samples: %d\n", totalPwrSamplesTaken );
627     printf( "Power Samples per Sec: %d\n", pwrSamplesPerSec );
628     printf( "Power consumption (RMS): %d\n", getRMSCurrent( ));
629     printf( "\n" );
630 }
```

```
//-------------------------------------------------------------------------------------------------------------
//
// LCS Block Controller - Occupancy Detect
//
//-------------------------------------------------------------------------------------------------------------
//
// LCS Block Controller
// 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
//
//-------------------------------------------------------------------------------------------------------------

// ??? contains the routines that manage the track section occupancy detection
```

```
//-------------------------------------------------------------------------------------------------------------
//
// LCS Block Controller - Signal Control
//
//-------------------------------------------------------------------------------------------------------------
//
// LCS Block Controller
// 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
//
//-------------------------------------------------------------------------------------------------------------


// ??? contains the routines that manage the signal settings
```

```
//--------------------------------------------------------------------------------------------------------------
//
// LCS Block Controller - Turnout Control
//
//--------------------------------------------------------------------------------------------------------------
//
// LCS Block Controller
// 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
//
//--------------------------------------------------------------------------------------------------------------


// ??? contains the routines that manage the signal settings
```

```
//--------------------------------------------------------------------------------------------------------------------
//
// LCS Block Controller - RailCom Support
//
//--------------------------------------------------------------------------------------------------------------------
//
// LCS Block Controller
// 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
//
//--------------------------------------------------------------------------------------------------------------------

// ??? to work on ...

#if 0


//--------------------------------------------------------------------------------------------------------------------
// 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 );
}


//--------------------------------------------------------------------------------------------------------------------
// 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,
```

```
99
100     INV,    INV,    INV,    RSV1,   INV,    0x2B,   0x30,   INV,    // c
101     INV,    0x2A,   0x2F,   INV,    0x31,   INV,    INV,    INV,
102
103     INV,    0x29,   0x2E,   INV,    0x2D,   INV,    INV,    INV,    // d
104     0x2C,   INV,    INV,    INV,    INV,    INV,    INV,    INV,
105
106     INV,    RSV3,   0x28,   INV,    0x27,   INV,    INV,    INV,    // e
107     0x26,   INV,    INV,    INV,    INV,    INV,    INV,    INV,
108
109     ACK,    INV,    INV,    INV,    INV,    INV,    INV,    INV,    // f
110     INV,    INV,    INV,    INV,    INV,    INV,    INV,    INV,
111 };
112
113 //-------------------------------------------------------------------------------------------------------------
114 // Railcom datagrams are sent from a mobile or a stationary decoder.
115 //
116 //-------------------------------------------------------------------------------------------------------------
117 enum railComDatagramType : uint8_t {
118
119     RX_DG_TYPE_UNDEFINED  = 0,
120     RC_DG_TYPE_MOB        = 1,
121     RC_DG_TYPE_STAT       = 2
122 };
123
124 //-------------------------------------------------------------------------------------------------------------
125 // Each mobile decoder railcom datagram will start with an ID field of four bits. Channel one will use only
126 // the ADR_HIG and ADR_LOW Ids. All IDs can be used for channel 2. Since decoders answer on channel one
127 // for each DCC packet they receive, here is a good chance that channel 1 will contains nonsense data. This
128 // is different for channel two, where only the addressed decoder explicitly answers. To decide whether
129 // a railcom message is valid, you should perhaps ignore channel 1 data and just check channel 2 for this
130 // purpose. A RC datagram starts with the 4-bit ID and an 8 to 32bit payload.
131 //
132 //      RC_DG_MOB_ID_POM       ( 0  )  - 12bit
133 //      RC_DG_MOB_ID_ADR_HIGH  ( 1  )  - 12bit
134 //      RC_DG_MOB_ID_ADR_LOW   ( 2  )  - 12bit
135 //      RC_DG_MOB_ID_APP_EXT   ( 3  )  - 18bit
136 //      RC_DG_MOB_ID_APP_DYN   ( 7  )  - 18bit
137 //      RC_DG_MOB_ID_XPOM_1    ( 8  )  - 36bit
138 //      RC_DG_MOB_ID_XPOM_2    ( 9  )  - 36bit
139 //      RC_DG_MOB_ID_XPOM_3    ( 10 )  - 36bit
140 //      RC_DG_MOB_ID_XPOM_4    ( 11 )  - 36bit
141 //      RC_DG_MOB_ID_TEST      ( 12 )  - ignore
142 //      RC_DG_MOB_ID_SEARCH    ( 14 )  - 48bit
143 //
144 // A datagram with the ID 14 is a DDC-A datagram and all 8 datagram bytes are combined to an 48bit datagram.
145 // A datagram packet can also contain more than one datagram. For example there could be two 18-bit length
146 // datagram in one packet or 3 12-bit packets and so on. Finally, unused bytes in channel two could contain
147 // an ACK to fill them up.
148 //
149 //-------------------------------------------------------------------------------------------------------------
150 enum railComDatagramMobId : uint8_t {
151
152     RC_DG_MOB_ID_POM        = 0,
153     RC_DG_MOB_ID_ADR_HIGH   = 1,
154     RC_DG_MOB_ID_ADR_LOW    = 2,
155     RC_DG_MOB_ID_APP_EXT    = 3,
156     RC_DG_MOB__IDAPP_DYN    = 7,
157     RC_DG_MOB_ID_XPOM_1     = 8,
158     RC_DG_MOB_ID_XPOM_2     = 9,
159     RC_DG_MOB_ID_XPOM_3     = 10,
160     RC_DG_MOB_ID_XPOM_4     = 11,
161     RC_DG_MOB_ID_TEST       = 12,
162     RC_DG_MOB_ID_SEARCH     = 14
163 };
164
165 //-------------------------------------------------------------------------------------------------------------
166 // Similar to the mobile decode, a stationary decoder datagram will start an ID field of four bits. Stationary
167 // decoders also define a datagram with "SRQ" and no ID field to request service from the base station.
168 //
169 // ??? to fill in ...
170 //
171 //      RC_DG_STAT_ID_SRQ       ( 0  )  - 12bit
172 //      RC_DG_STAT_ID_POM       ( 1  )  - 12bit
173 //      RC_DG_STAT_ID_STAT1     ( 4  )  - 12bit
174 //      RC_DG_STAT_ID_TIME      ( 5  )  - xxbit
175 //      RC_DG_STAT_ID_ERR       ( 6  )  - xxbit
176 //      RC_DG_STAT_ID_XPOM_1    ( 8  )  - 36bit
177 //      RC_DG_STAT_ID_XPOM_2    ( 9  )  - 36bit
178 //      RC_DG_STAT_ID_XPOM_3    ( 10 )  - 36bit
179 //      RC_DG_STAT_ID_XPOM_4    ( 11 )  - 36bit
180 //      RC_DG_STAT_ID_TEST      ( 12 )  - ignore
181 //
182 //-------------------------------------------------------------------------------------------------------------
183 enum railComDatagramStatId : uint8_t {
184
185     RC_DG_STAT_ID_SRQ       = 0,
186     RC_DG_STAT_ID_POM       = 1,
187     RC_DG_STAT_ID_STAT1     = 4,
188     RC_DG_STAT_ID_TIME      = 5,
189     RC_DG_STAT_ID_ERR       = 6,
190     RC_DG_STAT_ID_DYN       = 7,
191     RC_DG_STAT_ID_XPOM_1    = 8,
192     RC_DG_STAT_ID_XPOM_2    = 9,
193     RC_DG_STAT_ID_XPOM_3    = 10,
194     RC_DG_STAT_ID_XPOM_4    = 11,
195     RC_DG_STAT_ID_TEST      = 12
196 };
197
```

```
198 //-----------------------------------------------------------------------------------------------------------
199 // The RailCom buffer size. During the cutout period up to eight bytes of raw data are sent by the decoder if
200 // the Railcom option is enabled.
201 //
202 //-----------------------------------------------------------------------------------------------------------
203 const uint8_t   RAILCOM_BUF_SIZE = 8;
204
205
206
207 struct RailCom {
208
209
210
211     void                        startRailComIO( );
212     void                        stopRailComIO( );
213     uint8_t                     handleRailComMsg( );
214     uint8_t                     getRailComMsg( uint8_t *buf, uint8_t bufLen );
215
216 };
217
218
219
220
221 //-----------------------------------------------------------------------------------------------------------
222 // Railcom. If the cutout period and the RailCom feature is enabled, the signal state machine will also start
223 // and stop the UART reader for RailCom data. The final message is then to handle that message. In the cutout
224 // period, a decoder sends 8 data bytes. They are divided into two channels, 2bytes and another 6 bytes. The
225 // bytes themselves are encoded such that each byte has four bits set, i.e. a hamming weight of 4. The first
226 // channel is used to just send the locomotive address when the decoder is addressed. The second channel is
227 // used only when the decoder is explicitly addressed via a CV operation command to provide the answer to the
228 // request.
229 //
230 // The received datagrams are also recorded in the DCC_LOG, if enabled.
231 //
232 // ??? under construction....
233 // ??? we could store the last loco address in some global variable.
234 // ??? we could store the channel 2 datagram in the corresponding session.
235 // ??? still, both pieces of data needs to go somewhere before the next message is received...
236 //-----------------------------------------------------------------------------------------------------------
237 void LcsBaseStationDccTrack::startRailComIO( ) {
238
239     CDC::startUartRead( uartRxPin );
240 }
241
242 void LcsBaseStationDccTrack::stopRailComIO( ) {
243
244     CDC::stopUartRead( uartRxPin );
245 }
246
247 uint8_t LcsBaseStationDccTrack::handleRailComMsg( ) {
248
249     railComBufIndex = CDC::getUartBuffer( uartRxPin, railComMsgBuf, sizeof( railComMsgBuf ));
250
251     writeLogData( LOG_DCC_RCM, railComMsgBuf, railComBufIndex );
252
253     for ( uint8_t i = 0; i < railComBufIndex; i++ ) {
254
255         uint8_t dataByte = railComDecode[ railComMsgBuf[ i ]];
256
257         if      ( dataByte == ACK ) ;
258         else if ( dataByte == NACK ) ;
259         else if ( dataByte == BUSY ) ;
260         else if ( dataByte < 64 ) {
261
262             // ??? valid
263             // ??? a railCom message can have multiple datagrams
264             // we would need to handle each datagram, one at a time or fill them into a kind of structure
265             // that has a slot for the up to maximum 4 datagrams per railCom cutout period.
266         }
267         else {
268
269             // ??? invalid packet ... if this is channel2, discard the entire message.
270         }
271
272         railComMsgBuf[ i ] = dataByte;
273     }
274
275     flags &= ~ DT_F_RAILCOM_MSG_PENDING;
276     return ( ALL_OK );
277 }
278
279 // ??? not very useful, but good for debugging and initial testing .... and it works like a champ :-)
280
281 uint8_t LcsBaseStationDccTrack::getRailComMsg( uint8_t *buf, uint8_t bufLen ) {
282
283     if (( railComBufIndex > 0 ) && ( bufLen > 0 )) {
284
285         uint8_t i = 0;
286
287         do {
288
289             buf[ i ] = railComMsgBuf[ i ];
290             i++;
291
292         } while (( i < railComBufIndex ) && ( i < bufLen ));
293
294         return ( i );
295
296     } else return ( 0 );
```

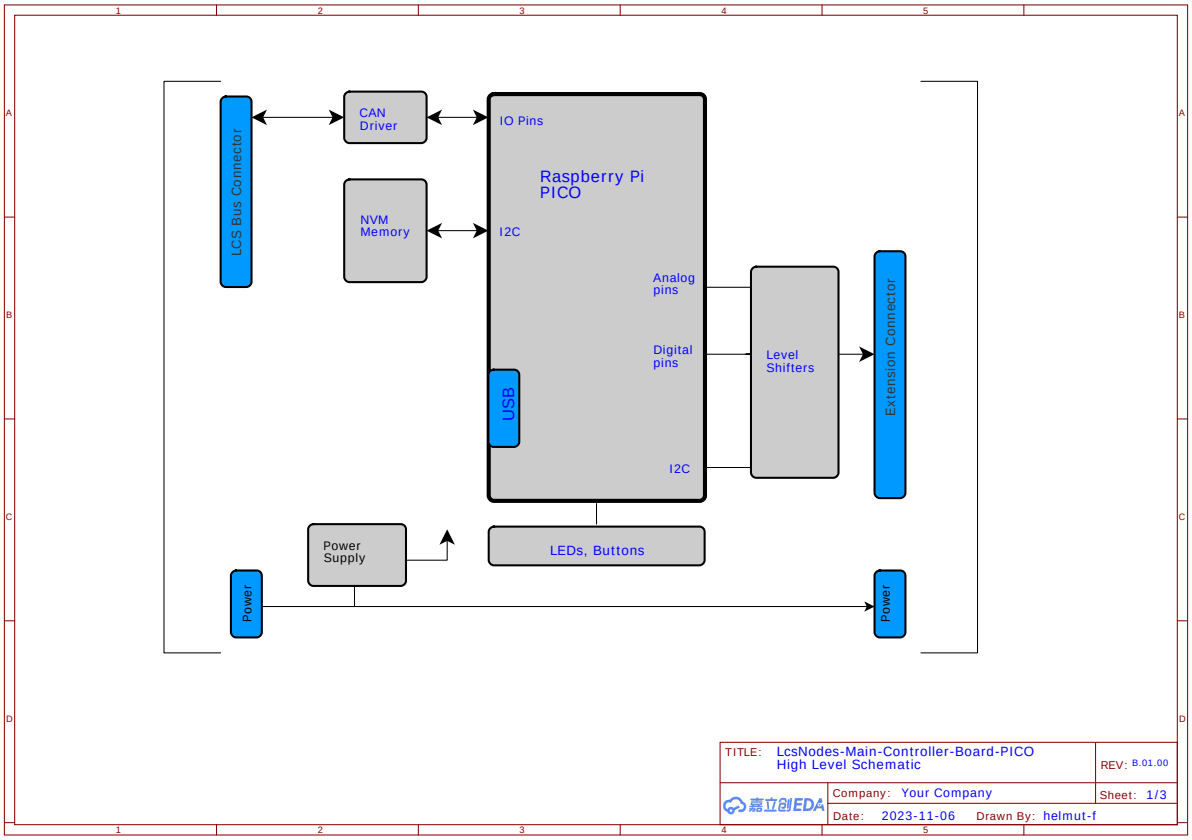
```
297 }
298
299 #endif
```

# B Tests

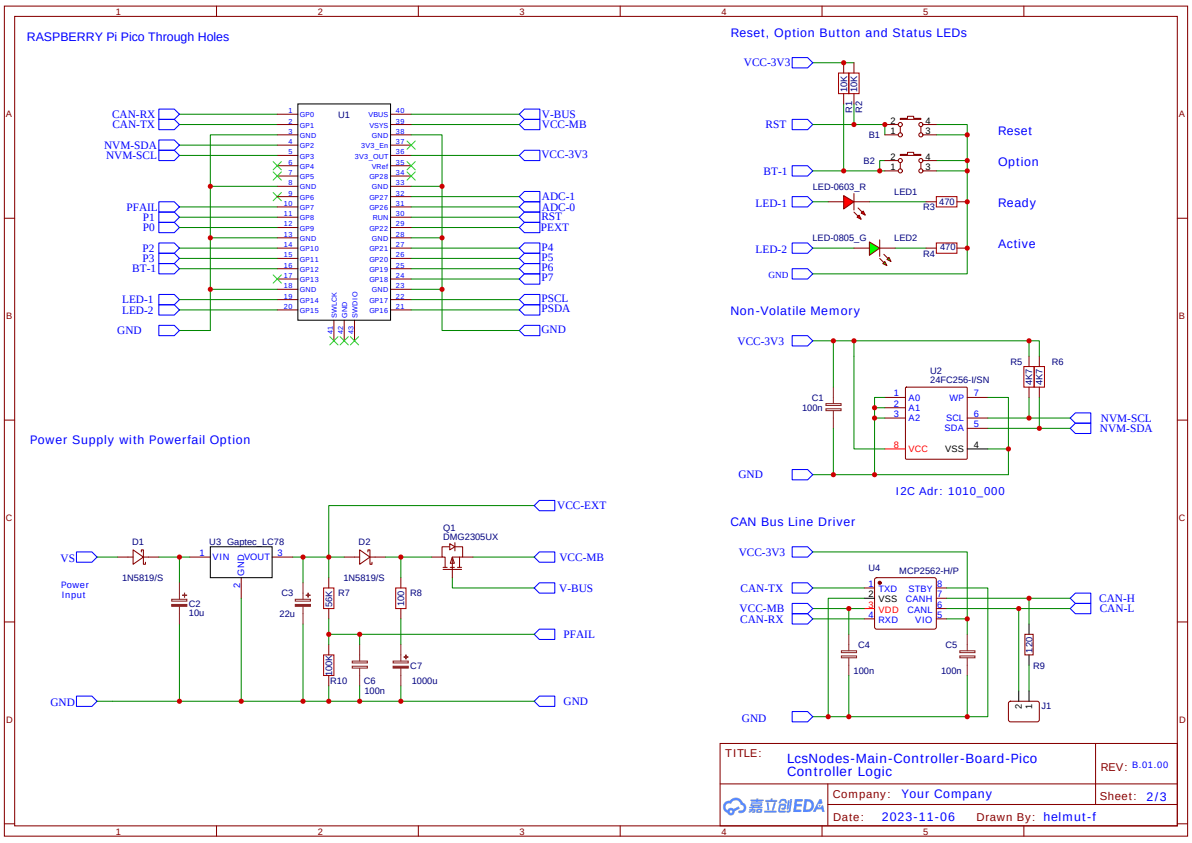
## B.1 Schematics

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

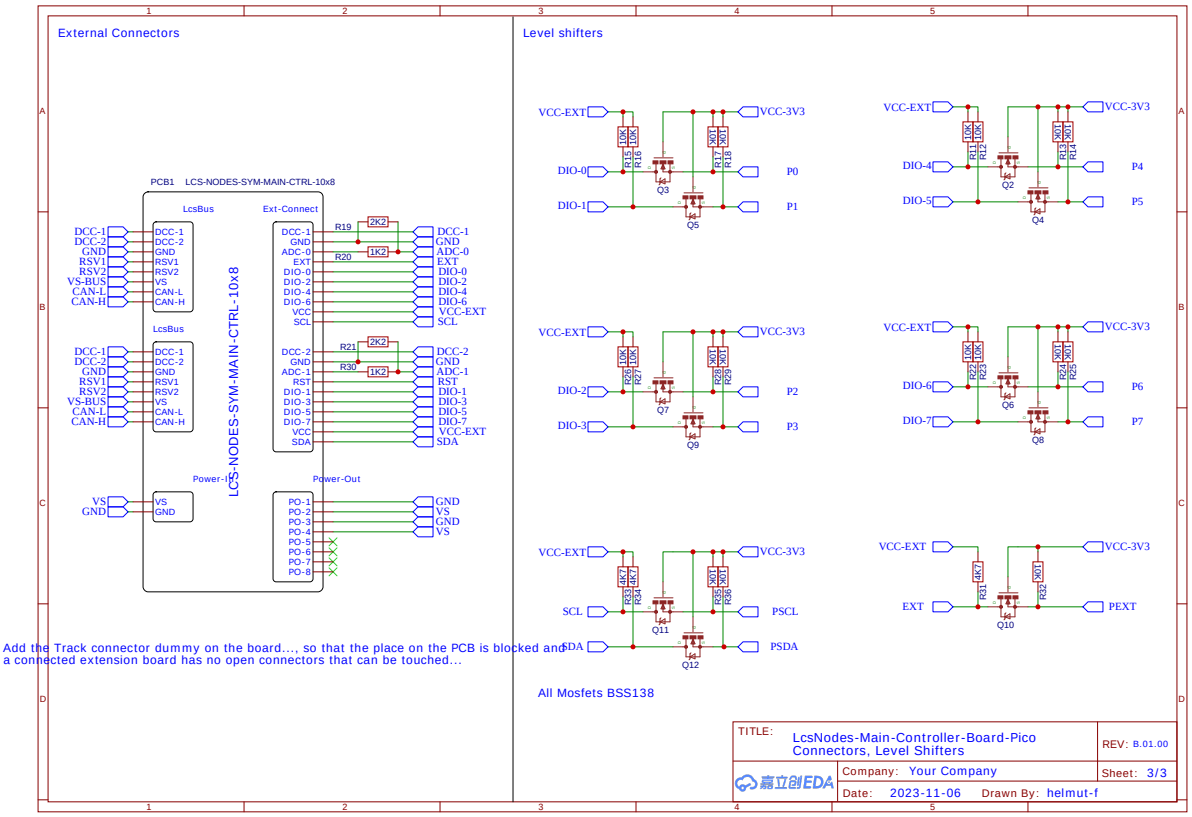
### B.1.1 part 1




### B.1.2 part 2


RASPBERRY Pi Pico Through Holes
CAN-RX
CAN-TX
NVM-SDA
NVM-SCL
PFAIL
P1
P0
P2
P3
BT-1
LED-1
LED-2
GND
U1
V-BUS
VCC-MB
VCC-3V3
ADC-1
ADC-0
RST
PEXT
P4
P5
P6
P7
PSCL
PSDA
GND
Reset, Option Button and Status LEDs
VCC-3V3
10K
RST
B1
Reset
BT-1
B2
Option
LED-0603_R
LED1
LED-1
470
Ready
LED-0805_G
LED2
LED-2
Active
Non-Volatile Memory
C1
100n
U2
24FC256-I/SN
R5
R6
4K7
NVM-SCL
NVM-SDA
I2C Adr: 1010_000
Power Supply with Powerfail Option
VS
Power Input
D1
1N5819/S
U3_Gaptec_LC78
VIN
VOUT
GND
C2
10u
C3
22u
D2
Q1
DMG2305UX
VCC-EXT
VCC-MB
V-BUS
PFAIL
56K
R7
100
R8
100K
R10
C6
100n
C7
1000u
CAN Bus Line Driver
U4
MCP2562-H/P
TXD
VSS
VDD
RXD
STBY
CANH
CANL
VIO
CAN-H
CAN-L
C4
100n
C5
100n
120
R9
J1
TITLE:
LcsNodes-Main-Controller-Board-Pico Controller Logic
REV: B.01.00
Company: Your Company
Sheet: 2/3
Date: 2023-11-06
Drawn By: helmut-f

## B.1.3 part 3




# B.2 Lists

## B.2.1 A simple list

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

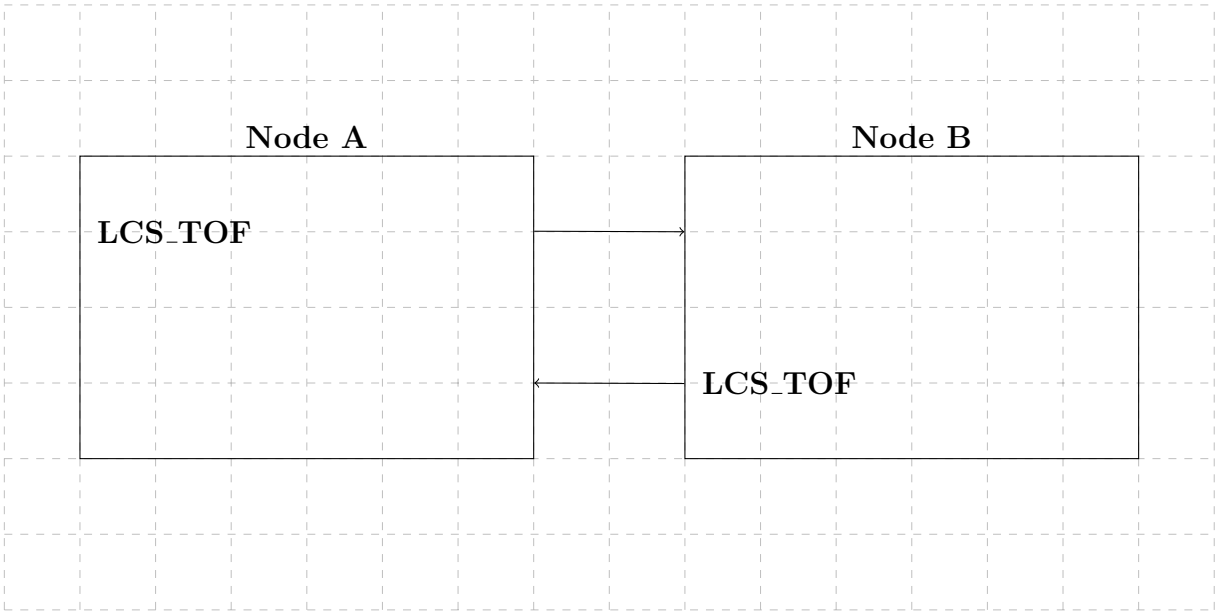
## B.2.2 An instruction word layout

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

| 1 | 3 | 6 |
|---|---|---|
| Test | | |

# B.3 Protocol boxes

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



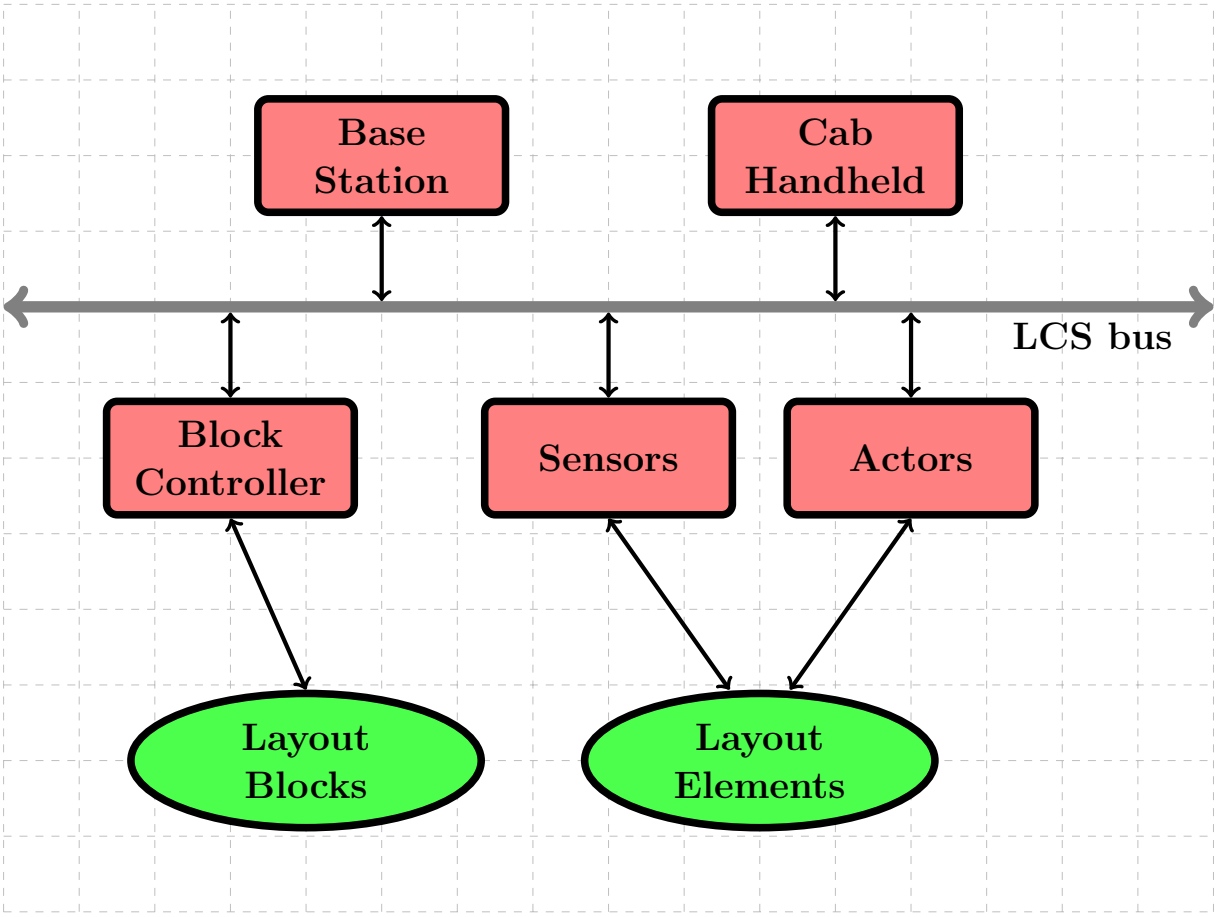

## B.4 Split rectangle

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

| Hugo |
| --- |
| Berta |
| Carla |

## B.5 Using tikzstyle

Another test with tikzstyle. It still is a lot of work to even make simple pictures look nice :-)


Base Station
Cab Handheld
LCS bus
Block Controller
Sensors
Actors
Layout Blocks
Layout Elements