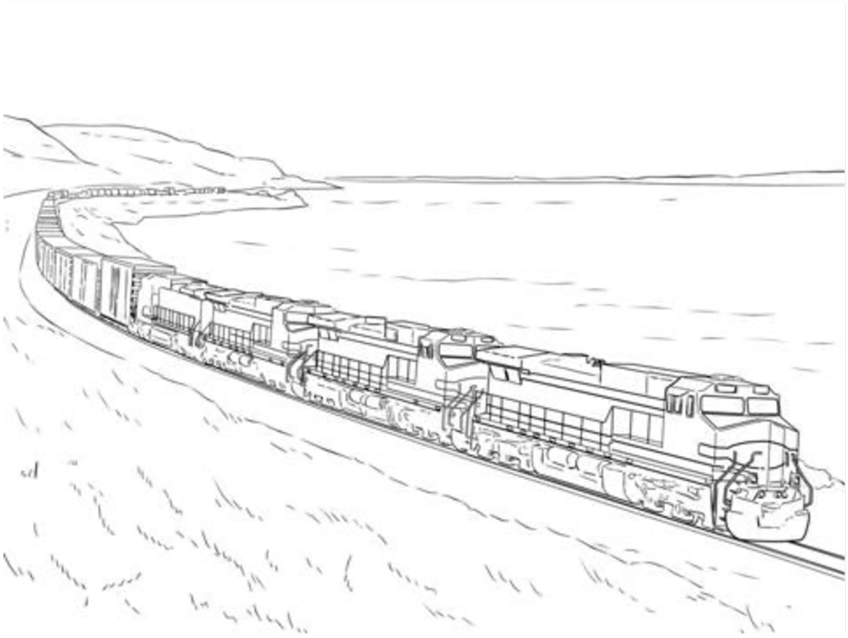


A Layout Control System for Model Railroads

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

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 as the "Euro PCB" format. The main controller board has on the left side the connectors for the LCS bus and the power input. On the 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. Nevertheless, a rather rich functionality set from the controller should be available to the extension board for flexibility. There should be ports for digital input and output, analog input, PWM outputs, serial outputs and so on. 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	**GND**	2	**GND**	Common ground pins.

Continued on next page

Pin	Name	Pin	Name	Purpose
3	**DCC-1**	4	**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.
5	**GND**	6	**GND**	Common ground pins.
7	**RST**	8	**EXT**	RST is reset line. Active Low. EXT is the external interrupt line which be raised from an extension board. Active low.
9	**ADC-0**	10	**ADC-1**	Analog input pins. The input is not protected. The analog voltage range is 0 to VCC.
11	**GND**	12	**GND**	Common ground pins.
13	**DIO-0**	14	**DIO-1**	Plain digital Pins, input or output. The pins are protected.
15	**DIO-2**	16	**DIO-3**	Plain digital Pins, input or output. The pins are protected.
17	**DIO-4**	18	**DIO-5**	Plain digital Pins, input or output. The pins are protected.
19	**DIO-6**	20	**DIO-7**	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	**GND**	30	**GND**	Common ground pins.
31	**VCC**	32	**VCC**	VCC 5V supply to extension boards.
33	**GND**	34	**GND**	Common ground pins.

Continued on next page

Pin	Name	Pin	Name	Purpose
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 connectors on the boards are female connectors.

Since the connector chosen is a 2x20 connector, the signal pin numbers shown above change their row numbering, depending whether it is output or input connector. When looking at the schematics and the connector layout on the PCB, the signals on the output connector have pin 1 to 10 on the leftmost row, and on the rightmost row on the input connector, such that pin 1 of the output connector connects to pin 1 of the input connector and so on. The same is true for pins 11 to 20.

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

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

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

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 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 architecture of a hardware modules, it connectors and board layout. A key concept is the idea of a common component, the main controller, and extension that can be connected. Nevertheless, there are good cases for combining a main controller and the extension hardware into one monolithic board. But in any case, the connectors and their purposes stay the same from board to board. While the main controller boards always have the LCS bus and power input on the left side, the extension connector and tack 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

};

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

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

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

    CF_UNDEFINED    = 0,
    CF_RP_PICO      = 1
};

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

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

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

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

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

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

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

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

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

    uint8_t   CFG_STATUS;

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

    uint8_t   DIO_PIN_0;
    uint8_t   DIO_PIN_1;
    uint8_t   DIO_PIN_2;
    uint8_t   DIO_PIN_3;
    uint8_t   DIO_PIN_4;
    uint8_t   DIO_PIN_5;
    uint8_t   DIO_PIN_6;
```

```
    uint8_t   DIO_PIN_7;
    uint8_t   DIO_PIN_8;
    uint8_t   DIO_PIN_9;
    uint8_t   DIO_PIN_10;
    uint8_t   DIO_PIN_11;
    uint8_t   DIO_PIN_12;
    uint8_t   DIO_PIN_13;
    uint8_t   DIO_PIN_14;
    uint8_t   DIO_PIN_15;

    uint8_t   ADC_PIN_0;
    uint8_t   ADC_PIN_1;
    uint8_t   ADC_PIN_2;
    uint8_t   ADC_PIN_3;

    uint8_t   PWM_PIN_0;
    uint8_t   PWM_PIN_1;
    uint8_t   PWM_PIN_2;
    uint8_t   PWM_PIN_3;

    // ??? do we need more for quad controller ?

    uint8_t   UART_RX_PIN_0;
    uint8_t   UART_TX_PIN_0;

    uint8_t   UART_RX_PIN_1;
    uint8_t   UART_TX_PIN_1;

    uint8_t   UART_RX_PIN_2;
    uint8_t   UART_TX_PIN_2;

    uint8_t   UART_RX_PIN_3;
    uint8_t   UART_TX_PIN_3;

    uint8_t   SPI_MOSI_PIN_0;
    uint8_t   SPI_MISO_PIN_0;
    uint8_t   SPI_SCLK_PIN_0;

    uint8_t   SPI_MOSI_PIN_1;
    uint8_t   SPI_MISO_PIN_1;
    uint8_t   SPI_SCLK_PIN_1;

    uint8_t   NVM_I2C_SCL_PIN;
    uint8_t   NVM_I2C_SDA_PIN;
    uint8_t   NVM_I2C_ADR_ROOT;

    uint8_t   EXT_I2C_SCL_PIN;
    uint8_t   EXT_I2C_SDA_PIN;
    uint8_t   EXT_I2C_ADR_ROOT;

    uint32_t  NODE_NVM_SIZE;
    uint32_t  EXT_NVM_SIZE;

    uint8_t   CAN_BUS_CTRL_MODE;
    uint8_t   CAN_BUS_RX_PIN;
    uint8_t   CAN_BUS_TX_PIN;
    uint32_t  CAN_BUS_DEF_ID;
};

//------------------------------------------------------------------------------------------------------------
// The routines that make up the hardware abstraction layer. The routines expect hardware pin numbers.
// To recap, the CDC layer offers a set of reserved resource names, such as "DIO_PIN_0", which describes
// the resource containing the hardware pin and some flags. The configuration routines in this layer will use
// these pins and other data stored to configure the hardware. Under the defined resource name name all
// upper layers refer to the hardware using the to the configured IO capabilities.
//
// Complex resources, such as the UART or SPI interface, have more than one HW pin they will use. In this
// case one of the HW pins, see the function documentation, will serve as the handle to the resource.
//
//------------------------------------------------------------------------------------------------------------

//------------------------------------------------------------------------------------------------------------
// The console IO functions. We will provide a serial IO via the USB connector of the PICO. The files
// need to be linked with the "tinyUSB" library and the cmake file needs to set the option. Then we can
// use scanf and printf and so on. In addition, we need  function  that just attempts to read a character
// and returns immediately when there is none.
//
//------------------------------------------------------------------------------------------------------------
uint8_t         configureConsoleIO( );
bool            isConsoleConnected( );
char            getConsoleChar( uint32_t timeoutVal = 0 );

//------------------------------------------------------------------------------------------------------------
// CDC setup and configuration routines. The idea is to help the library write with a default configuration
// structure. All pins HW that are fixed in their location will be set. A library programmer will just get
// that default structure and set the values necessary for the particular case.
//
//------------------------------------------------------------------------------------------------------------
CdcConfigDesc   getConfigDefault( );
CdcConfigDesc   *getConfigActual( );
void            printConfigInfo( CdcConfigDesc *ci );
void            setDebugLevel( uint8_t level = 0 );

uint8_t         init( CdcConfigDesc *ci );
void            fatalError( uint8_t n );
void            fatalErrorMsg( char *str, uint8_t n, uint8_t rStat );

//------------------------------------------------------------------------------------------------------------
// General controller routines.
```

```
//
//-------------------------------------------------------------------------------------------------------------
uint16_t      getFamily( );
uint32_t      getVersion( );
uint32_t      getChipMemSize( );
uint32_t      getChipNvmSize( );
uint32_t      getCpuFrequency( );
uint32_t      getMillis( );
uint32_t      getMicros( );
void          sleepMillis( uint32_t val );
void          sleepMicros( uint32_t val );

//-------------------------------------------------------------------------------------------------------------
// The LCS runtime needs to build a unique ID for the node.
//
//-------------------------------------------------------------------------------------------------------------
uint32_t      createUid( );

//-------------------------------------------------------------------------------------------------------------
// Timer management routines.
//
//-------------------------------------------------------------------------------------------------------------
void          onTimerEvent( TimerCallback functionId );
void          startRepeatingTimer( uint32_t val );
void          setRepeatingTimerLimit( uint32_t val );
uint32_t      getRepeatingTimerLimit( );
void          stopRepeatingTimer( );

//-------------------------------------------------------------------------------------------------------------
// Analog input routines.
//
//-------------------------------------------------------------------------------------------------------------
uint8_t       configureAdc( uint8_t adcPin );
uint16_t      getAdcRefVoltage( );
uint16_t      getAdcDigitRange( );
uint16_t      readAdc( uint8_t adcPin );

//-------------------------------------------------------------------------------------------------------------
// Digital Input/Output routines.
//
//-------------------------------------------------------------------------------------------------------------
uint8_t       configureDio( uint8_t dioPin, uint8_t Mode = IN );
void          registerDioCallback( uint8_t dioPin, uint8_t event, CDC::GpioCallback func );
void          unregisterDioCallback( uint8_t dioPin );
bool          readDio( uint8_t dioPin );
uint8_t       writeDio( uint8_t dioPin, bool val );
uint8_t       toggleDio( uint8_t dioPin );
uint32_t      readDioMask( uint32_t dioMask );
uint8_t       writeDioMask( uint32_t dioMask, uint32_t dioVal );
uint8_t       writeDioPair( uint8_t dioPin1, bool val1, uint8_t dioPin2, bool val2 );

//-------------------------------------------------------------------------------------------------------------
// PWM output routines.
//
//-------------------------------------------------------------------------------------------------------------
uint8_t       configurePwm( uint8_t   pwmPin,
                            uint32_t  pwmFreqency,
                            bool      phaseCorrect  = true,
                            bool      inverted      = false
                        );

uint8_t       writePwm( uint8_t pwmPin, uint8_t dutyCycle );

//-------------------------------------------------------------------------------------------------------------
// Serial IO routines.
//
//-------------------------------------------------------------------------------------------------------------
uint8_t       configureUart( uint8_t rxPin, uint8_t txPin, uint32_t baudRate, UartMode mode );
uint8_t       startUartRead( uint8_t rxPin );
uint8_t       stopUartRead( uint8_t rxPin );
uint8_t       getUartBuffer( uint8_t rxPin, uint8_t *buf, uint8_t bufLen );

//-------------------------------------------------------------------------------------------------------------
// I2C management routines.
//
//-------------------------------------------------------------------------------------------------------------
uint8_t       configureI2C( uint8_t sclPin, uint8_t sdaPin, uint32_t baudRate = 100 * 1000 );
uint8_t       i2cWrite( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit = false );
uint8_t       i2cRead( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit = false );

//-------------------------------------------------------------------------------------------------------------
// SPI management routines.
//
//-------------------------------------------------------------------------------------------------------------
uint8_t       configureSPI( uint8_t sclkPin, uint8_t mosiPin, uint8_t misoPin, uint32_t baudRate = 10 * 1000 * 1000 );
uint8_t       spiBeginTransaction( uint8_t sclkPin, uint8_t csPin );
uint8_t       spiEndTransaction( uint8_t sclkPin, uint8_t csPin );
uint8_t       spiRead( uint8_t sclkPin, uint8_t *buf, uint32_t len );
uint8_t       spiWrite( uint8_t sclkPin, uint8_t *buf, uint32_t len );

};

#endif
```

```
//----------------------------------------------------------------------------------------------------------
//
// LCS - Controller dependent code Layer - Raspberry PI Pico Implementation
//
//----------------------------------------------------------------------------------------------------------
// This source file contains the the RP2040 controller family hardware library code. The idea of this library
// is to shield the actual hardware of processor and board implementation from the upper layers but still keep
// the flexibility and performance of the underlying hardware. The library works with the concept of HW pins,
// which are identifiers for an HW entity. This is easy for a GPIO pin, where the mapping is directly one to
// one. For more complex HW entries such as the I2C or UART hardware, one pin is selected as the identifier to
// that entity. For each complex entity an instance variable is maintained where all the relevant data is kept.
//
// A historic note. The original LCS code was written for Atmega and Pico. With the complete shift to PICO,
// the CDC library just serves as a simple interface to the PICO functions. One day, we may see more different
// controllers and controller families. The idea is that the LCS runtime is shielded from them.
//
//----------------------------------------------------------------------------------------------------------
//
// LCS - Controller Dependent Code - Raspberry PI Pico Implementation
// Copyright (C) 2022 - 2024 Helmut Fieres
//
// This program is free software: you can redistribute it and/or modify it under the terms of the GNU General
// Public License as published by the Free Software Foundation, either version 3 of the License, or (at your
// option) any later version.
//
// This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the
// implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
// for more details.
//
// You should have received a copy of the GNU General Public License along with this program. If not, see
// http://www.gnu.org/licenses
//
//  GNU General Public License:  http://opensource.org/licenses/GPL-3.0
//
//----------------------------------------------------------------------------------------------------------
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

#include "pico/stdlib.h"
#include "pico/stdio.h"
#include "tusb_config.h"
#include "hardware/regs/usb.h"
#include "hardware/regs/rosc.h"
#include "hardware/regs/addressmap.h"
#include "hardware/clocks.h"
#include "hardware/gpio.h"
#include "hardware/adc.h"
#include "hardware/pwm.h"
#include "hardware/uart.h"
#include "hardware/i2c.h"
#include "hardware/spi.h"

#include "LcsCdcLib.h"

//----------------------------------------------------------------------------------------------------------
// Local name space. This file has two sections. The first is this local name space with all internal
// variables and routines local to the file. The second part contains the exported routines to be called by
// the core library and the firmware designers that need access to the underlying HW portion managed by this
// lowest layer.
//
//----------------------------------------------------------------------------------------------------------
namespace {

using namespace CDC;

//----------------------------------------------------------------------------------------------------------
// "CDC_DEBUG" is the local define for printing debug information. In contrast to the rest of the debugging
// and tracing of LCS libraries and programs, this library will have to be recompiled to enable debugging.
//
//----------------------------------------------------------------------------------------------------------
#define CDC_DEBUG 0

//----------------------------------------------------------------------------------------------------------
// Debug and Trace support. Instead of conditional compilation, we will print debug messages based on the
// setting of the debug level.
//----------------------------------------------------------------------------------------------------------
uint8_t debugLevel = 0;

//----------------------------------------------------------------------------------------------------------
// The CDC Library version data.
//
//----------------------------------------------------------------------------------------------------------
const uint8_t CDC_LIB_MAJOR_VERSION = 1;
const uint8_t CDC_LIB_MINOR_VERSION = 0;

//----------------------------------------------------------------------------------------------------------
// Valid pin mapping for the Raspberry PI Pico board. We construct a set of bitmask for the pin numbers.
// Pin Numbers range from 0 to 28. The bitmasks specify wether a pin can be assigned to the hardware type
// purpose. During configuration of a CDC function, the pins are checked against these bitmasks. All pins
// can be used as GPIO pins or PWM pins. All other hardware functions are bound to dedicated pins. Note
// that we do not check for assigning a pin to several different hardware functions. All we check is that
// the pin can be used for the desired purpose. A check performed by the CDC library routines is simply
// done through:
//
//    if (( 1 <<  pin ) & VALID_xxx )
//
//----------------------------------------------------------------------------------------------------------
```

```
const uint8_t  MAX_PIN_NUM           = 28;

const uint32_t VALID_GPIO_PINS       = 0x1FFFFFFF;
const uint32_t VALID_PWM_PINS        = 0x1FFFFFFF;
const uint32_t VALID_ADC_PINS        = ( 1 << 26 ) | ( 1 << 27 ) | ( 1 << 28 );

const uint32_t VALID_I2C_0_SDA_PINS = ( 1 << 0  ) | ( 1 << 4  ) | ( 1 << 8  ) |
                                      ( 1 << 12 ) | ( 1 << 16 ) | ( 1 << 20 );
const uint32_t VALID_I2C_0_SCL_PINS = ( 1 << 1  ) | ( 1 << 5  ) | ( 1 << 9  ) |
                                      ( 1 << 13 ) | ( 1 << 17 ) | ( 1 << 21 );

const uint32_t VALID_I2C_1_SDA_PINS = ( 1 << 2  ) | ( 1 << 6  ) | ( 1 << 10 ) |
                                      ( 1 << 14 ) | ( 1 << 18 ) | ( 1 << 26 );
const uint32_t VALID_I2C_1_SCL_PINS = ( 1 << 3  ) | ( 1 << 7  ) | ( 1 << 11 ) |
                                      ( 1 << 15 ) | ( 1 << 19 ) | ( 1 << 27 );

const uint32_t VALID_UART_0_TX_PINS = ( 1 << 0  ) | ( 1 << 12 ) | ( 1 << 16 );
const uint32_t VALID_UART_0_RX_PINS = ( 1 << 1  ) | ( 1 << 13 ) | ( 1 << 17 );

const uint32_t VALID_UART_1_TX_PINS = ( 1 << 4  ) | ( 1 << 8  );
const uint32_t VALID_UART_1_RX_PINS = ( 1 << 5  ) | ( 1 << 9  );

const uint32_t VALID_SPI_0_SCK_PINS = ( 1 << 2  ) | ( 1 << 6  ) | ( 1 << 18 );
const uint32_t VALID_SPI_0_TX_PINS  = ( 1 << 3  ) | ( 1 << 7  ) | ( 1 << 19 );
const uint32_t VALID_SPI_0_RX_PINS  = ( 1 << 0  ) | ( 1 << 4  ) | ( 1 << 16 );

const uint32_t VALID_SPI_1_SCK_PINS = ( 1 << 10 ) | ( 1 << 14 );
const uint32_t VALID_SPI_1_TX_PINS  = ( 1 << 11 ) | ( 1 << 15 );
const uint32_t VALID_SPI_1_RX_PINS  = ( 1 << 8  ) | ( 1 << 12 );

const uint32_t VALID_I2C_0_PINS     = VALID_I2C_0_SDA_PINS | VALID_I2C_0_SCL_PINS;
const uint32_t VALID_I2C_1_PINS     = VALID_I2C_1_SDA_PINS | VALID_I2C_1_SCL_PINS;

const uint32_t VALID_UART_0_PINS    = VALID_UART_0_TX_PINS | VALID_UART_0_RX_PINS;
const uint32_t VALID_UART_1_PINS    = VALID_UART_1_TX_PINS | VALID_UART_1_RX_PINS;

const uint32_t VALID_SPI_0_PINS     = VALID_SPI_0_SCK_PINS | VALID_SPI_0_TX_PINS | VALID_SPI_0_RX_PINS;
const uint32_t VALID_SPI_1_PINS     = VALID_SPI_1_SCK_PINS | VALID_SPI_1_TX_PINS | VALID_SPI_1_RX_PINS;

//------------------------------------------------------------------------------------------------
// Characteristics of the Raspberry Pi Pico and some key constants for the CDC library.
//
//------------------------------------------------------------------------------------------------
const uint16_t CONTROLLER_FAMILY           = CDC::CF_RP_PICO;
const uint32_t CHIP_MEM_SIZE               = 264 * 1024;
const uint32_t CHIP_NVM_SIZE               = 0;

const uint16_t ADC_DIGIT_RANGE             = 1024;
const uint16_t ADC_REF_VOLTAGE_MILLI_VOLT = 3300;

const uint8_t  MAX_UART_BUF_SIZE           = 8;

const uint32_t I2C_FREQUENCY               = 100 * 1000;
const uint32_t I2C_TIME_OUT_IN_MS          = 250;

const uint32_t SPI_FREQUENCY               = 10000000L;

const uint16_t  MAX_CPU_CORE               = 2;
const uint16_t  MAX_INT_PIN                = 24;

//--------------------------------------------------------------------------------------------------
// A timer instance. We currently support inly one HW timer.
//
//--------------------------------------------------------------------------------------------------
struct TimerInst {

    bool                configured  = false;
    repeating_timer_t   timerData;

};

//--------------------------------------------------------------------------------------------------
// An ADC instance. The PICO supports up to three ADC inputs. When we use such an input, the corresponding
// instance data is kept in this structure. We also keep the PICO ADC number, so we can select the correct
// instance.
//
//--------------------------------------------------------------------------------------------------
struct AdcInst {

    bool      configured   = false;
    uint8_t   adcPin       = CDC::UNDEFINED_PIN;
    uint8_t   adcNum       = 0;
};

//--------------------------------------------------------------------------------------------------
// A PWM output instance. GPIO pins can also be used as PWM output pins. The PWM output related data is
// kept in this instance.
//
//--------------------------------------------------------------------------------------------------
struct PwmInst {

    bool        configured   = false;
    uint8_t     pwmPin       = CDC::UNDEFINED_PIN;
    uint        wrap         = 0;
    uint        channel      = 0;
    uint        sliceNum     = 0;
};

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

```
// A UART instance. UARTS are used to read in a serial stream from the RailCom detectors. There can be two
// hardware based UART instances, or up to four software defined instances. The instance also keeps a small
// buffer where the data is read into. We also keep the PICO UART HW instance used.
//
//----------------------------------------------------------------------------------------------------------
struct UartInst {

    bool                configured    = false;
    uint8_t             rxPin         = CDC::UNDEFINED_PIN;
    uint8_t             txPin         = CDC::UNDEFINED_PIN;
    uint16_t            baudSetting   = 0;
    uint8_t             dataBits      = 8;
    uart_parity_t       parityMode    = UART_PARITY_NONE;
    uint8_t             stopBits      = 1;
    int                 uartIrq       = 0;
    uint8_t             uartMode      = 0;

    volatile uint8_t    rxBufIndex    = 0;
    volatile uint8_t    rxDataBuf[ MAX_UART_BUF_SIZE ] = { 0 };

    uart_inst_t         *uartHw       = nullptr;
};

//----------------------------------------------------------------------------------------------------------
// The I2C instance. The PICO features two HW instances of an I2C port. The instance data contains the
// assigned GPIO pins, the baud rate and a timeout. We also keep the I2C HW instance used.
//
//----------------------------------------------------------------------------------------------------------
struct I2CInst {

    bool        configured    = false;
    uint8_t     sclPin        = CDC::UNDEFINED_PIN;
    uint8_t     sdaPin        = CDC::UNDEFINED_PIN;
    uint32_t    baudRate      = I2C_FREQUENCY;
    uint32_t    timeoutValMs  = I2C_TIME_OUT_IN_MS;

    i2c_inst_t  *i2cHw        = nullptr;
};

//----------------------------------------------------------------------------------------------------------
// The SPI instance. The PICO features two SPI HW instances. We keep the assigned GPIO pins for the SPI
// interface as well as the PICO HW instance. Since the SPI protocol explicitly sets the selected HW select
// pin, we remember that we are in a transaction with perhaps more than one call to the SPI routines.
//
//----------------------------------------------------------------------------------------------------------
struct SPIInst {

    bool        configured  = false;
    bool        active      = false;
    uint8_t     selectPin   = CDC::UNDEFINED_PIN;
    uint8_t     mosiPin     = CDC::UNDEFINED_PIN;
    uint8_t     misoPin     = CDC::UNDEFINED_PIN;
    uint8_t     sclkPin     = CDC::UNDEFINED_PIN;
    uint32_t    frequency   = SPI_FREQUENCY;

    spi_inst_t  *spiHw      = nullptr;
};

//----------------------------------------------------------------------------------------------------------
// The interrupt table for the GPIO pin interrupts. The PICO can have only one interrupt handler. We will
// allocate a table where a handler can be set for each pin. When an interrupt comes in and there is a
// handler configured, it will be called.
//
//----------------------------------------------------------------------------------------------------------
struct GpioIsrTable {

    uint16_t                numOfHandlers = 0;
    CDC::GpioCallback       gpioIsrTable[ MAX_CPU_CORE ][ MAX_INT_PIN + 1 ];
};

//----------------------------------------------------------------------------------------------------------
// Local variables. We maintain an instance variable for each of the possible HW entities, such as an I2C
// interface or a UART. Note that not all are used at the same time. The instance variables map from the
// simple pin numbers to the PICO structures and whatever else we need to remember for this entity.
//
//----------------------------------------------------------------------------------------------------------
CDC::CdcConfigDesc          cfg;
CDC::TimerCallback          timerCallback = nullptr;
GpioIsrTable                cdcIntHandlers;
repeating_timer_t           timerData;
AdcInst                     CdcAdc0;
AdcInst                     CdcAdc1;
AdcInst                     CdcAdc2;
AdcInst                     CdcAdc3;
I2CInst                     CdcI2C0;
I2CInst                     CdcI2C1;
SPIInst                     CdcSPI0;
SPIInst                     CdcSPI1;
UartInst                    CdcUart0;
UartInst                    CdcUart1;
UartInst                    CdcUart2;
UartInst                    CdcUart3;
PwmInst                     CdcPwm0;
PwmInst                     CdcPwm1;
PwmInst                     CdcPwm2;
PwmInst                     CdcPwm3;

//----------------------------------------------------------------------------------------------------------
// "validPin" is called to check that a pin is in the correct number range, defined and matches the bitmask
```

```
// for the desired purpose. For example, configuring an I2C port will check that the two GPIO pins are
// indeed routable to the I2C HW block in the PICO.
//
//-------------------------------------------------------------------------------------------------------
bool validPin( uint8_t pin, uint32_t mask ) {

    if ( pin == CDC::UNDEFINED_PIN )  return ( true );
    if ( pin > MAX_PIN_NUM )          return ( false );
    return (( 1 << pin ) & mask );
}

//-------------------------------------------------------------------------------------------------------
// When no interrupt is configured for a GPIO pin, we set the table entry to a dummy handler. This way
// we do not have to check for a valid procedure label when we handle an interrupt.
//
//-------------------------------------------------------------------------------------------------------
void dummyIsrHandler ( uint8_t pin, uint8_t event ) { }

//-------------------------------------------------------------------------------------------------------
// Setup the ISR table. The PICO can have only one interrupt handler. When you want a handler per GPIO pin,
// the solution is to have a table when you keep the handler on a per pin base.
//
//-------------------------------------------------------------------------------------------------------
void initIsrTable( ) {

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

        for ( uint16_t j = 0; j < MAX_INT_PIN; j++ ) {

            cdcIntHandlers.gpioIsrTable[ i ][ j ] = dummyIsrHandler;
        }
    }
}

//-------------------------------------------------------------------------------------------------------
// The PICO uses a set of constants to describe the interrupt type. We map our interrupt types to the PICO
// GPIO_IRQ_xxx types.
//
//-------------------------------------------------------------------------------------------------------
uint32_t mapGpioIntEvent( uint8_t event ) {

    switch ( event ) {

        case CDC::EVT_LOW:     return( GPIO_IRQ_LEVEL_LOW );
        case CDC::EVT_HIGH:    return( GPIO_IRQ_LEVEL_HIGH );
        case CDC::EVT_FALL:    return( GPIO_IRQ_EDGE_FALL );
        case CDC::EVT_RISE:    return( GPIO_IRQ_EDGE_RISE );
        case CDC::EVT_CHANGE:  return( GPIO_IRQ_EDGE_RISE | GPIO_IRQ_EDGE_FALL );
        default:      return( 0 );
    }
}

//-------------------------------------------------------------------------------------------------------
// The PICO uses a set of constants to describe the interrupt type. We map them to our types.
//
//-------------------------------------------------------------------------------------------------------
uint8_t mapPicoGpioEvent( uint32_t event ) {

    switch ( event ) {

        case GPIO_IRQ_LEVEL_LOW:  return( CDC::EVT_LOW );
        case GPIO_IRQ_LEVEL_HIGH: return( CDC::EVT_HIGH );
        case GPIO_IRQ_EDGE_FALL:  return( CDC::EVT_FALL );
        case GPIO_IRQ_EDGE_RISE:  return( CDC::EVT_RISE );
        default:                  return( 0 );
    }
}

//-------------------------------------------------------------------------------------------------------
// Global Interrupt handlers. The hardware and low level library will call these handlers, which in turn
// will invoke the respective callback function if configured. The GPIO interrupt handler manages the
// handler for all possible IO pins. The PICO can only have one interrupt routine, so we feature an array
// of handlers where a handler for a GPIO pin can be registered. If there is a handler set, we just invoke
// it. The other handlers are for the timer and the UART hardware.
//
//-------------------------------------------------------------------------------------------------------
void gpioCallback( uint gpioPin, uint32_t event ) {

    cdcIntHandlers.gpioIsrTable[ get_core_num( )][ gpioPin] ( gpioPin, mapPicoGpioEvent( event ));
}

bool repeatingTimerAlarm( repeating_timer_t *rt ) {

    if ( timerCallback != nullptr ) timerCallback((uint32_t) ( - timerData.delay_us ));
    return ( true );
}

void uartRxCallback0( ) {

    while ( uart_is_readable( uart0 )) {

        uint8_t ch = uart_getc( uart0 );
        if ( CdcUart0.rxBufIndex < MAX_UART_BUF_SIZE ) CdcUart0.rxDataBuf[CdcUart0.rxBufIndex++ ] = ch;
    }
}

void uartRxCallback1( ) {

    while ( uart_is_readable( uart1 )) {
```

```

        uint8_t ch = uart_getc( uart1 );
        if ( CdcUart1.rxBufIndex < MAX_UART_BUF_SIZE ) CdcUart1.rxDataBuf[ CdcUart1.rxBufIndex++ ] = ch;
    }
}

//----------------------------------------------------------------------------------------------------------
// The default configuration descriptor. The Application program fills in such a structure, which can be
// seen as the HW pin assignments for the PICO controllers and the particular board on which the application
// will be deployed. The application will simply use the field names to address the particular PICO HW
// function. For example, a configuration has mapped DIO_PIN_5 to GPIO pin 12, because that is where the
// particular board has mapped DIO_PIN_5 to the hardware line. The application will just use the DIO_PIN_5
// field when talking to that GPIO pin. Whenever the board layout changes, there could be another PICO GPIO
// pin, but the name "DIO_PIN_5" for the application upper layers does not change.
//
// Note that there is a great flexibility what a PICO HW  pin can do and hence a lot of our fields are just
// "UNDEFINED" with no constraints. Nevertheless, there is a function which will do some plausibility checks
// for such a structure. Also, each configuration routine will do again a check that the GPIO pins used do
// indeed map to a PICO HW block for the desired purpose.
//
// The configuration structure does not replace the actual configuration calls to make to the CDC library.
// It is just a mapping of reserved names to actual GPIO pins.
//
//----------------------------------------------------------------------------------------------------------
CDC::CdcConfigDesc getConfigDefaultRP2040( ) {

    CDC::CdcConfigDesc tmp;

    tmp.CFG_STATUS          = CDC::INIT_PENDING;

    // ??? controller family ?
    // ??? what other characteristics ? ( e.g. mem size ? )

    tmp.READY_LED_PIN       = CDC::UNDEFINED_PIN;
    tmp.ACTIVE_LED_PIN      = CDC::UNDEFINED_PIN;

    tmp.EXT_INT_PIN         = CDC::UNDEFINED_PIN;
    tmp.PFAIL_PIN           = CDC::UNDEFINED_PIN;

    tmp.DIO_PIN_0           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_1           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_2           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_3           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_4           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_5           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_6           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_7           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_8           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_9           = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_10          = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_11          = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_12          = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_13          = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_14          = CDC::UNDEFINED_PIN;
    tmp.DIO_PIN_15          = CDC::UNDEFINED_PIN;

    tmp.ADC_PIN_0           = CDC::UNDEFINED_PIN;
    tmp.ADC_PIN_1           = CDC::UNDEFINED_PIN;
    tmp.ADC_PIN_2           = CDC::UNDEFINED_PIN;
    tmp.ADC_PIN_3           = CDC::ILLEGAL_PIN;

    tmp.PWM_PIN_0           = CDC::UNDEFINED_PIN;
    tmp.PWM_PIN_1           = CDC::UNDEFINED_PIN;
    tmp.PWM_PIN_2           = CDC::UNDEFINED_PIN;
    tmp.PWM_PIN_3           = CDC::UNDEFINED_PIN;

    tmp.UART_RX_PIN_0       = CDC::UNDEFINED_PIN;
    tmp.UART_TX_PIN_0       = CDC::UNDEFINED_PIN;

    tmp.UART_RX_PIN_1       = CDC::UNDEFINED_PIN;
    tmp.UART_TX_PIN_1       = CDC::UNDEFINED_PIN;

    tmp.UART_RX_PIN_2       = CDC::UNDEFINED_PIN;
    tmp.UART_TX_PIN_2       = CDC::UNDEFINED_PIN;

    tmp.UART_RX_PIN_3       = CDC::UNDEFINED_PIN;
    tmp.UART_TX_PIN_3       = CDC::UNDEFINED_PIN;

    tmp.SPI_MOSI_PIN_0      = CDC::UNDEFINED_PIN;
    tmp.SPI_MISO_PIN_0      = CDC::UNDEFINED_PIN;
    tmp.SPI_SCLK_PIN_0      = CDC::UNDEFINED_PIN;

    tmp.SPI_MOSI_PIN_1      = CDC::UNDEFINED_PIN;
    tmp.SPI_MISO_PIN_1      = CDC::UNDEFINED_PIN;
    tmp.SPI_SCLK_PIN_1      = CDC::UNDEFINED_PIN;

    tmp.NVM_I2C_SCL_PIN     = CDC::UNDEFINED_PIN;
    tmp.NVM_I2C_SDA_PIN     = CDC::UNDEFINED_PIN;

    tmp.EXT_I2C_SCL_PIN     = 17;
    tmp.EXT_I2C_SDA_PIN     = 16;

    tmp.CAN_BUS_RX_PIN      = CDC::UNDEFINED_PIN;
    tmp.CAN_BUS_TX_PIN      = CDC::UNDEFINED_PIN;

    return ( tmp );
}

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

```
// Validate a configuration structure. This routine will do basic checking of the pin configuration passed.
// The PICO is very flexible when it comes to what a pin can do. However, there are still some rules to
// follow. Also, we have dedicated settings for at least the I2C channels and the CAN bus IO pins.
//
//----------------------------------------------------------------------------------------------------------
uint8_t validateConfigRP20040( CDC::CdcConfigDesc *ci ) {

    // ??? a ton of "validXXX" ?

    return ( NO_ERR ); // for now....
}

}; // namespace


//----------------------------------------------------------------------------------------------------------
// Bane CDC. All routines and definitions exported are in this name space.
//
//----------------------------------------------------------------------------------------------------------
namespace CDC {

//----------------------------------------------------------------------------------------------------------
// For debugging purposes. Instead of conditional compilations, the debug level will enable the printing of
// debug and trace data.
//
//----------------------------------------------------------------------------------------------------------
void setDebugLevel( uint8_t level ) {

    debugLevel = level;
}

//----------------------------------------------------------------------------------------------------------
// "getConfigDefault" initializes a configuration structure and sets the pre-assigned values. A typical
// sequence for an application start sequence would be to create an initial structure this way and then set
// the relevant pins and values according to the actual hardware configuration.
//
//----------------------------------------------------------------------------------------------------------
CdcConfigDesc getConfigDefault( ) {

   return ( getConfigDefaultRP2040( ));
}

//----------------------------------------------------------------------------------------------------------
// "getConfigActual" will return a pointer to the copy we kept when calling the init routine with the config
// structure to use. There is no need for the upper layers to keep the structure used at initialization time.
//
//----------------------------------------------------------------------------------------------------------
CdcConfigDesc *getConfigActual( ) {

   return ( &cfg );
}

//----------------------------------------------------------------------------------------------------------
// CDC library setup. The "init" routine will ready the CDC library. The main task is to validate the pins and
// values for the particular controller capabilities. The init routine can be called more than once without a
// problem.
//
//----------------------------------------------------------------------------------------------------------
uint8_t init( CdcConfigDesc *ci ) {

    cfg = *ci;

    initIsrTable( );
    configureConsoleIO( );

    return ( validateConfigRP20040( ci ));
}

//----------------------------------------------------------------------------------------------------------
// "fatalError" is the error communication method when we cannot get anything to work, except the onboard
// LED. The Raspberry Pi PICO has a small Led on the board. We will use this LED to "blink" an error code.
// There are up to eight codes. The sequence is as follows:
//
//    repeat forever:
//
//    - 1s ON, 0.5s OFF
//    - for ( int i = 0; i < n; i++ ) { 0.5s ON; 0.5s OFF; }
//
// The only way to get out of this loop is then to reset the board. Fatal errors are hopefully not many. One
// obvious one is when we cannot detect the NVM and thus know nothing about the board.
//
//----------------------------------------------------------------------------------------------------------
void fatalError( uint8_t n ) {

    const uint8_t   ledPin      = 25;
    const uint32_t  longPulse   = 1000;
    const uint32_t  shortPulse  = 250;

    n = n % 8;

    gpio_init( ledPin );
    gpio_set_dir( ledPin, GPIO_OUT );

    while ( true ) {

        sleep_ms( longPulse );

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

```

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

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

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

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

    return ( CONTROLLER_FAMILY );
}

uint32_t getVersion( ) {

    return ( CDC_LIB_MAJOR_VERSION << 8 | CDC_LIB_MINOR_VERSION );
}

uint32_t getChipMemSize( ) {

    return ( CHIP_MEM_SIZE );
}

uint32_t getChipNvmSize( ) {

    return ( CHIP_NVM_SIZE   );
}

uint32_t getCpuFrequency( ) {

    return ( clock_get_hz( clk_sys ));
}

uint32_t getMillis( ) {

    return ( to_ms_since_boot( get_absolute_time( )));
}

uint32_t getMicros( ) {

    return ( to_us_since_boot( get_absolute_time( )));
}

void sleepMillis( uint32_t val ) {

    sleep_ms( val );
}

void sleepMicros( uint32_t val ) {

    sleep_us( val );
}

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

    uint32_t rVal = 0;

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

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

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

    return ( rVal );
}

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

```
// via the PICO libraries which in turn use the tinyUSB library. However, there could be a timing problem
// where the USB stack is not ready and we conclude wrongly that there is no USB connection. For now, let's
// rather go with the risk that there is just power on the USB connector.
//
// Finally, there is a routine to get a character for the command interfaces. Since the function just reads
// in a character, optionally with a timeout how long to wait for any inout.
//
// PS: The USB check way would be "return( stdio_usb_connected( ));" instead of the GPIO check.
//------------------------------------------------------------------------------------------------------
uint8_t configureConsoleIO( ) {

    stdio_init_all( );
    return( NO_ERR );
}

bool isConsoleConnected( ) {

    gpio_init( PICO_VBUS_PIN );
    gpio_set_dir( PICO_VBUS_PIN, GPIO_IN );

    return( gpio_get( PICO_VBUS_PIN ));
}

char getConsoleChar( uint32_t timeoutVal ) {

    int ch = getchar_timeout_us( timeoutVal );
    return(( ch == PICO_ERROR_TIMEOUT ) ? 0 : ch );
}

//------------------------------------------------------------------------------------------------------
// Timer section. The CDC library features one generic repeating timer with a microsecond resolution. The
// routines start and stop the timer and allow to set a new limit. The PICO offers a high level function that
// schedules a repeating timer with the property of measuring the interval also from the start of the
// callback invocation. This is exactly what we need to implement the tick interrupt for the DCC signal state
// machine. The "setRepeatingTimerLimit" function will adjust the timer limit counter while the timer already
// is counting toward a limit. Note that the timer option that already start the next round while the timer
// interrupt handler executes is specified by using negative limit values. The timer functionality also
// offers two timestamp routines to get the number of milliseconds and number of microseconds since system
// start.
//
// ??? would we one day need more than one timer instance ?
//------------------------------------------------------------------------------------------------------
void startRepeatingTimer( uint32_t val ) {

    int64_t limit = val;
    add_repeating_timer_us( - limit, repeatingTimerAlarm, nullptr, &timerData );
}

void stopRepeatingTimer( ) {

    cancel_repeating_timer( &timerData );
}

uint32_t getRepeatingTimerLimit( ) {

    return ((uint32_t) ( - timerData.delay_us ));
}

void setRepeatingTimerLimit( uint32_t val ) {

    int64_t limit = val;
    timerData.delay_us = ((int64_t) - limit );
}

void onTimerEvent( CDC::TimerCallback functionId ) {

    timerCallback = functionId;
}

//------------------------------------------------------------------------------------------------------
// DIO section. A digital pin is the bread and butter hardware resource and can be an input or output pin. For
// inputs, an internal pull-up resistor can be set.There are a couple of interfaces. First the single pin
// read, write and toggle. Next are read and write mask routines which work on all IO pins at once. Note that
// no cross checking is done if the pins are used by other CDC functions. Finally there is a convenience
// routine which write a pair of data. This is typically used for the H-Bridge control pins, which are set at
// the same time.
//
// A GPIO pin can also have an attached interrupt handler. When we register a handler for a pin, there are
// two different PICO lib routines to use. When there is no handler registered so far, we register the
// common callback and store the particular GPIO handler in the handler table. Otherwise, we just store the
// handler and enable the GPIO pin for interrupts.
//
//------------------------------------------------------------------------------------------------------
uint8_t configureDio( uint8_t dioPin, uint8_t mode ) {

    if ( ! validPin( dioPin, VALID_GPIO_PINS )) return ( DIO_PIN_ERR );

    gpio_init( dioPin );

    switch ( mode ) {

        case IN:  gpio_set_dir( dioPin, false );  break;
        case OUT: {

            gpio_set_dir( dioPin, true );
            gpio_set_drive_strength ( dioPin, GPIO_DRIVE_STRENGTH_12MA );

        }  break;

```

```
        case IN_PULLUP: {

            gpio_set_dir( dioPin, false );
            gpio_pull_up( dioPin );

        } break;

        default: gpio_set_dir( dioPin, false );
    }

  return ( NO_ERR );
}

void registerDioCallback( uint8_t dioPin, uint8_t event, CDC::GpioCallback func ) {

    if ( dioPin <= MAX_INT_PIN ) {

        if ( cdcIntHandlers.numOfHandlers == 0 )
            gpio_set_irq_enabled_with_callback( dioPin, mapGpioIntEvent( event ), true, gpioCallback );
        else
            gpio_set_irq_enabled( dioPin, mapGpioIntEvent( event ), true);

        cdcIntHandlers.gpioIsrTable[ get_core_num( ) ][ dioPin ] = func;
        cdcIntHandlers.numOfHandlers ++;
    }
}

void unregisterDioCallback( uint8_t dioPin ) {

    if ( dioPin <= MAX_INT_PIN ) {

        if ( cdcIntHandlers.gpioIsrTable[ get_core_num( ) ][ dioPin ] != nullptr ) {

            gpio_set_irq_enabled( dioPin, 0, false );
            cdcIntHandlers.gpioIsrTable[ get_core_num( ) ][ dioPin ] = dummyIsrHandler;
            cdcIntHandlers.numOfHandlers --;
        }
    }
}

bool readDio( uint8_t dioPin ) {

    return ( gpio_get( dioPin ));
}

uint8_t writeDio( uint8_t dioPin, bool val ) {

    gpio_put( dioPin, val );
    return ( NO_ERR );
}

uint8_t toggleDio( uint8_t dioPin ) {

    writeDio( dioPin, ! readDio( dioPin ));
    return ( NO_ERR );
}

uint8_t writeDioPair( uint8_t dioPin1, bool val1, uint8_t dioPin2, bool val2 ) {

    uint32_t maskData = ( 1UL << dioPin1 ) | ( 1UL << dioPin2 );
    uint32_t valData  = (( val1 ) ? ( 1 << dioPin1 ) : 0 ) | (( val2 ) ? ( 1 << dioPin2 ) : 0 );

    gpio_put_masked( maskData, valData );
    return ( NO_ERR );
}

uint32_t readDioMask( uint32_t dioMask ) {

    return ( gpio_get_all( ) & dioMask );
}

uint8_t writeDioMask( uint32_t dioMask, uint32_t dioVal ) {

    gpio_put_masked( dioMask, dioVal );
    return ( NO_ERR );
}

//----------------------------------------------------------------------------------------------------------
// ADC section. The analog input channel represented by the pin is configured. At initialization, the ADC pin
// number is validated and the ADC subsystem initialized. The PICO does an analog read in about 2us. This is
// so fast, it does for our purpose make not much sense to implement an asynchronous option. Furthermore, the
// ADC value scaled down to a 10-bit resolution.
//
//----------------------------------------------------------------------------------------------------------
uint8_t configureAdc( uint8_t adcPin ) {

    if ( ! validPin( adcPin, VALID_ADC_PINS ))  return ( ADC_PIN_ERR );

    AdcInst *tmp = nullptr;

    if ( adcPin == cfg.ADC_PIN_0 ) {

        tmp = &CdcAdc0;
        tmp -> adcPin = adcPin;
        tmp -> adcNum = 0;

    }
     else if ( adcPin == cfg.ADC_PIN_1 ) {

```

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

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

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

    return ( NO_ERR );
}

uint16_t getAdcRefVoltage( ) {

    return ( ADC_REF_VOLTAGE_MILLI_VOLT );
}

uint16_t getAdcDigitRange( ) {

    return ( ADC_DIGIT_RANGE );
}

uint16_t readAdc( uint8_t adcPin ) {

    AdcInst *tmp = nullptr;

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

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

    UartInst *uart = nullptr;

    if ( mode == UART_MODE_8N1 ) {

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

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

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

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

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

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

```
        return ( NO_ERR );
    }
    else if ( mode == UART_MODE_8N1_PIO ) {

        return ( NOT_SUPPORTED );
    }
    else return ( NOT_SUPPORTED );
}

uint8_t startUartRead( uint8_t rxPin ) {

    UartInst *uart = nullptr;

    if      ( rxPin == CdcUart0.rxPin ) uart = &CdcUart0;
    else if ( rxPin == CdcUart1.rxPin ) uart = &CdcUart1;
    else if ( rxPin == CdcUart2.rxPin ) uart = &CdcUart2;
    else if ( rxPin == CdcUart3.rxPin ) uart = &CdcUart3;
    else                                return ( CDC::UART_PORT_ERR );

    if (( uart != nullptr ) && ( uart -> uartMode == UART_MODE_8N1 )) {

        uart_set_irq_enables( uart -> uartHw, true, false );
        uart -> rxBufIndex = 0;
        return ( NO_ERR );
    }
    else if (( uart != nullptr ) && ( uart -> uartMode == UART_MODE_8N1_PIO )) {

        return ( NOT_SUPPORTED );
    }
    else return ( UART_PORT_ERR );
}

uint8_t stopUartRead( uint8_t rxPin ) {

    UartInst *uart = nullptr;

    if      ( rxPin == CdcUart0.rxPin ) uart = &CdcUart0;
    else if ( rxPin == CdcUart1.rxPin ) uart = &CdcUart1;
    else if ( rxPin == CdcUart2.rxPin ) uart = &CdcUart2;
    else if ( rxPin == CdcUart3.rxPin ) uart = &CdcUart3;

    if (( uart != nullptr ) && ( uart ->uartMode == UART_MODE_8N1 )) {

        uart_set_irq_enables( uart -> uartHw, false, false );
        return ( NO_ERR );
    }
    else if (( uart != nullptr ) && ( uart -> uartMode == UART_MODE_8N1_PIO )) {

        return ( NOT_SUPPORTED );
    }
    else return ( UART_PORT_ERR );
}

uint8_t getUartBuffer( uint8_t rxPin, uint8_t *buf, uint8_t bufLen ) {

    UartInst *uart = nullptr;

    if      ( rxPin == CdcUart0.rxPin ) uart = &CdcUart0;
    else if ( rxPin == CdcUart1.rxPin ) uart = &CdcUart1;
    else if ( rxPin == CdcUart2.rxPin ) uart = &CdcUart2;
    else if ( rxPin == CdcUart3.rxPin ) uart = &CdcUart3;
    else                                return ( 0 );

    if (( uart != nullptr ) && ( uart -> rxBufIndex > 0 ) && ( bufLen > 0 )) {

        uint8_t i = 0;
        while (( i < uart -> rxBufIndex ) && ( i < bufLen )) {

            buf[ i ] = uart -> rxDataBuf[ i ];
            i++;
        }

        return ( i );
    }
    else return ( 0 );
}

//------------------------------------------------------------------------------------------------------
// PWM section. The PICO is quite flexible when it comes to PWM signals. We implement a simple PWM capability.
// There is the frequency which set during configuration and there is the write operation which set the duty
// cycle. The calculations are best described in the PICO C++ SDK. We do the setting of phase, wrap count,
// etc. once when we configure the PWM channel. All the "writePwm" function then will do is to manipulate the
// duty cycle. In other words, when we change the frequency we need to configure again.
//
// There is one small issue left. Channel come in pairs. For some reason there is no call to individually
// set the "inverted" option on a channel. When we set the inverted option for a pin, we currently also set
//  the inverted option for the other channel since we just don't know better. To be correct, all possible
// PWM pins and their "inverted" option would need to be stored somewhere.
//
// To do .... ( there is a way via the pwm_Config CSR field... )
//
// ??? should we have also a kind of PWM pair ? Is that even possible ?
// ??? do we need more PWM pins ? The PICO is really flexible ?
// ??? combine DIO and PWM somehow ?
//------------------------------------------------------------------------------------------------------
uint8_t configurePwm( uint8_t pwmPin, uint32_t pwmFreqency, bool phaseCorrect, bool inverted ) {

    PwmInst *pwm = nullptr;
```

```

    if      ( pwmPin == cfg.PWM_PIN_0 ) pwm = &CdcPwm0;
    else if ( pwmPin == cfg.PWM_PIN_1 ) pwm = &CdcPwm1;
    else if ( pwmPin == cfg.PWM_PIN_2 ) pwm = &CdcPwm2;
    else if ( pwmPin == cfg.PWM_PIN_3 ) pwm = &CdcPwm3;
    else                                return ( PWM_PIN_ERR );

    if ( phaseCorrect ) pwmFreqency = pwmFreqency * 2;

    uint32_t sysClock = getCpuFrequency( );
    uint32_t clkDiv   = sysClock / pwmFreqency / 4096 + ( sysClock % ( pwmFreqency * 4096 ) != 0 );

    if ( clkDiv / 16 == 0 ) clkDiv = 16;

    pwm -> pwmPin   = pwmPin;
    pwm -> wrap     = sysClock * 16 / clkDiv / pwmFreqency - 1;
    pwm -> sliceNum = pwm_gpio_to_slice_num( pwmPin );
    pwm -> channel  = pwm_gpio_to_channel( pwmPin );

    pwm_config pwmConfig = pwm_get_default_config( );
    gpio_set_function( pwm -> pwmPin, GPIO_FUNC_PWM );
    pwm_config_set_wrap( &pwmConfig, pwm -> wrap );
    pwm_config_set_phase_correct( &pwmConfig, phaseCorrect );
    pwm_config_set_output_polarity( &pwmConfig, inverted, inverted );
    pwm_init ( pwm_gpio_to_slice_num( pwm -> pwmPin ), &pwmConfig, false );
    pwm_set_clkdiv_int_frac( pwm_gpio_to_slice_num( pwm -> pwmPin ), clkDiv / 16, clkDiv & 0xF );
    pwm_set_enabled( pwm_gpio_to_slice_num( pwmPin ), true );

    #if CDC_DEBUG == 0
    printf( "PWM Pin: % d, fPwm: % d, phase: % d, inverted: % d, "
            "clkDiv: % d, wrap: %d, sliceNum: %d, channel: %d\n",
            pwm -> pwmPin, pwmFreqency,  phaseCorrect, inverted,
            clkDiv, pwm -> wrap, pwm -> sliceNum, pwm -> channel );
    #endif

    return ( NO_ERR );
}

uint8_t writePwm( uint8_t pwmPin, uint8_t dutyCycle ) {

    #if CDC_DEBUG == 0
    printf( "Write PWM: Pin: %d, duty: %d\n", pwmPin, dutyCycle );
    #endif

    PwmInst *pwm = nullptr;

    if      ( pwmPin == cfg.PWM_PIN_0 ) pwm = &CdcPwm0;
    else if ( pwmPin == cfg.PWM_PIN_1 ) pwm = &CdcPwm1;
    else if ( pwmPin == cfg.PWM_PIN_2 ) pwm = &CdcPwm2;
    else if ( pwmPin == cfg.PWM_PIN_3 ) pwm = &CdcPwm3;
    else                                return ( PWM_PIN_ERR );

    if ( dutyCycle == 0 ) {

        gpio_set_function( pwm -> pwmPin, GPIO_FUNC_SIO );
        gpio_set_dir( pwm -> pwmPin, GPIO_OUT );
        gpio_put( pwm -> pwmPin, 0 );
    }
    else if ( dutyCycle == 255 ) {

        gpio_set_function( pwm -> pwmPin, GPIO_FUNC_SIO );
        gpio_set_dir( pwm -> pwmPin, GPIO_OUT);
        gpio_put( pwm -> pwmPin, 1 );
    }
    else {

        pwm_set_chan_level( pwm -> sliceNum, pwm -> channel, ( pwm -> wrap * dutyCycle / 256 ));
        pwm_set_enabled( pwm -> sliceNum, true );
    }

    return ( NO_ERR );
}

//-----------------------------------------------------------------------------------------------------
// I2C Section. The PICO has two HW blocks for I2C interfaces. The interface implements a simple read and
// write access to an I2C element. There is a timeout to avoid waiting forever on an operation.
//
//-----------------------------------------------------------------------------------------------------
uint8_t configureI2C( uint8_t sclPin, uint8_t sdaPin, uint32_t baudRate ) {

    I2CInst *i2c = nullptr;

    if ((( 1 << sclPin ) & VALID_I2C_0_SCL_PINS ) && (( 1 << sdaPin ) & VALID_I2C_0_SDA_PINS )) {

        i2c = &CdcI2C0;
        i2c -> i2cHw = i2c0;
    }
    else if ((( 1 << sclPin ) & VALID_I2C_1_SCL_PINS ) && (( 1 << sdaPin ) & VALID_I2C_1_SDA_PINS )) {

        i2c = &CdcI2C1;
        i2c -> i2cHw = i2c1;
    }
    else return ( CDC::I2C_PORT_ERR );

    i2c -> sclPin       = sclPin;
    i2c -> sdaPin       = sdaPin;
    i2c -> baudRate     = baudRate;
    i2c -> timeoutValMs = I2C_TIME_OUT_IN_MS;
    i2c -> configured   = true;
```

```

    i2c_init( i2c -> i2cHw, i2c -> baudRate );
    i2c_set_slave_mode( i2c -> i2cHw, false, 0 );

    gpio_set_function( i2c -> sclPin, GPIO_FUNC_I2C );
    gpio_set_function( i2c -> sdaPin, GPIO_FUNC_I2C);
    gpio_pull_up( i2c -> sclPin );
    gpio_pull_up( i2c -> sdaPin );

    return ( NO_ERR );
}

uint8_t i2cRead( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit ) {

    I2CInst *i2c = nullptr;

    if      (( CdcI2C0.sclPin == sclPin ) && ( CdcI2C0.configured )) i2c = &CdcI2C0;
    else if (( CdcI2C1.sclPin == sclPin ) && ( CdcI2C1.configured )) i2c = &CdcI2C1;
    else return ( I2C_PORT_ERR );

    auto ret = i2c_read_blocking_until( i2c -> i2cHw,
                                        i2cAdr,
                                        buf,
                                        len,
                                        stopBit,
                                        make_timeout_time_ms( i2c -> timeoutValMs ));

    #if CDC_DEBUG == 1
    printf( "i2cRead: scl: %d, i2c: 0x%x, buf: %p, buf[0] %x, buf[1] %x, len: %d, stop: %d\n",
               sclPin, i2cAdr, buf, buf[0], buf[1], len, stopBit );
    if ( ret == PICO_ERROR_GENERIC ) printf( "I2C read, PICO generic error\n" );
    if ( ret == PICO_ERROR_TIMEOUT ) printf( "I2C read, PICO timeout error\n" );
    #endif

    if (( ret == PICO_ERROR_GENERIC ) || ( ret == PICO_ERROR_TIMEOUT )) return ( I2C_READ_ERR );

    return ( NO_ERR );
}

uint8_t i2cWrite( uint8_t sclPin, uint8_t i2cAdr, uint8_t *buf, uint16_t len, bool stopBit ) {

    #if CDC_DEBUG == 1
    printf( "i2cWrite: scl: %d, i2c: 0x%x, buf: %p, buf[0] %x, buf[1] %x, len: %d, stop: %d\n",
             sclPin, i2cAdr, buf, buf[0], buf[1], len, stopBit );
    #endif


    I2CInst *i2c = nullptr;

    if      (( CdcI2C0.sclPin == sclPin ) && ( CdcI2C0.configured )) i2c = &CdcI2C0;
    else if (( CdcI2C1.sclPin == sclPin ) && ( CdcI2C1.configured )) i2c = &CdcI2C1;
    else return ( I2C_PORT_ERR );

    auto ret = i2c_write_blocking_until( i2c -> i2cHw,
                                         i2cAdr,
                                         buf,
                                         len,
                                         stopBit,
                                         make_timeout_time_ms( i2c -> timeoutValMs ));

    #if CDC_DEBUG == 1
    if ( ret == PICO_ERROR_GENERIC ) printf( "I2C write, PICO generic error\n" );
    if ( ret == PICO_ERROR_TIMEOUT ) printf( "I2C write, PICO timeout error\n" );
    #endif

    if (( ret == PICO_ERROR_TIMEOUT) || ( ret == PICO_ERROR_GENERIC ) || ( ret != len )) return ( I2C_WRITE_ERR );

    return ( NO_ERR );
}

//----------------------------------------------------------------------------------------------------------
// SPI interface section. The PICO features two SPI HW blocks. We implement a simple SPI interface with a
// a fixed set of SPI options for frequency, bit order and mode. One day this may change. We do not take
// care of the chip select stuff and expect that the caller manages the select pin.
//
//----------------------------------------------------------------------------------------------------------
uint8_t configureSPI( uint8_t sclkPin, uint8_t mosiPin, uint8_t misoPin, uint32_t baudRate ) {

    SPIInst *spi = nullptr;

     if ((( 1 << sclkPin ) & VALID_SPI_0_SCK_PINS ) &&
         (( 1 << mosiPin ) & VALID_SPI_0_TX_PINS  ) &&
         (( 1 << misoPin ) & VALID_SPI_0_RX_PINS  )) {

        spi = &CdcSPI0;
        spi -> spiHw = spi0;
    }
    else if ((( 1 << sclkPin ) & VALID_SPI_1_SCK_PINS ) &&
             (( 1 << mosiPin ) & VALID_SPI_1_TX_PINS  ) &&
             (( 1 << misoPin ) & VALID_SPI_1_RX_PINS  )) {

        spi = &CdcSPI1;
        spi -> spiHw = spi1;
    }
    else return ( SPI_PORT_ERR );

    spi -> mosiPin     = mosiPin;
    spi -> misoPin     = misoPin;
    spi -> sclkPin     = sclkPin;
```

```
    spi -> frequency   = SPI_FREQUENCY;
    spi -> configured  = true;
    spi -> active      = false;

    spi_init( spi -> spiHw, SPI_FREQUENCY );

    spi_set_format( spi -> spiHw,      // SPI instance
                    8,                 // Number of bits per transfer
                    SPI_CPOL_1,        // Polarity (CPOL)
                    SPI_CPHA_1,        // Phase (CPHA)
                    SPI_MSB_FIRST );

    gpio_set_function( sclkPin, GPIO_FUNC_SPI );
    gpio_set_function( mosiPin, GPIO_FUNC_SPI );
    gpio_set_function( misoPin, GPIO_FUNC_SPI );

    return ( NO_ERR );
}

uint8_t  spiBeginTransaction( uint8_t sclkPin, uint8_t csPin ) {

    SPIInst *spi = nullptr;

    if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
    else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
    else return ( SPI_PORT_ERR );

    if ( spi -> active ) {

        // ??? should we check who is active and just ignore when the same ? else "error " ?

        return ( NO_ERR );

    } else {

        spi -> active     = true;
        spi -> selectPin  = csPin;

        CDC::writeDio( csPin, false );
        return ( NO_ERR );
    }
}

uint8_t spiEndTransaction( uint8_t sclkPin, uint8_t csPin ) {

    SPIInst *spi = nullptr;

    if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
    else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
    else return ( SPI_PORT_ERR );

    if ( spi -> active ) {

        // ??? check that this is the correct pin ?

        CDC::writeDio( csPin, true );

        spi -> active     = false;
        spi -> selectPin  = UNDEFINED_PIN;

        return ( NO_ERR );

    }
    else return ( NO_ERR ); // ???  "error "  not active...
}

uint8_t spiRead( uint8_t sclkPin, uint8_t *buf, uint32_t len ) {

    SPIInst *spi = nullptr;

    if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
    else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
    else return ( SPI_PORT_ERR );

    if ( spi -> active ) {

        int bytesRead = spi_read_blocking( spi -> spiHw, 0, buf, len );
        return ( NO_ERR );

    } else return ( NO_ERR ); // ??? fix : not active ...
}

uint8_t spiWrite( uint8_t sclkPin, uint8_t *buf, uint32_t len ) {

    SPIInst *spi = nullptr;

    if      (( CdcSPI0.sclkPin == sclkPin ) && ( CdcSPI0.configured )) spi = &CdcSPI0;
    else if (( CdcSPI1.sclkPin == sclkPin ) && ( CdcSPI1.configured )) spi = &CdcSPI1;
    else return ( SPI_PORT_ERR );

    if ( spi -> active ) {

        spi_write_blocking( spi -> spiHw, buf, len );
        return ( NO_ERR );

    } else return ( NO_ERR ); // ??? fix : not active ...
}

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

```
// Print out the Config Structure.
//
//----------------------------------------------------------------------------------------------------------
void printConfigInfo( CdcConfigDesc *ci ) {

    printf( "CDC Pin Configuration Info ( status %d ): \n", ci -> CFG_STATUS );

    printf( "Pfail pin: %2d, ExtInt pin: %2d \n", ci -> PFAIL_PIN , ci -> EXT_INT_PIN );

    printf( "ReadyLed pin: %2d, ActiveLed pin: %2d \n", ci -> READY_LED_PIN , ci -> ACTIVE_LED_PIN );

    printf( "DIO pins ( 0 .. 7 ): %2d %2d %2d %2d %2d %2d %2d %2d\n",
            ci -> DIO_PIN_0 , ci -> DIO_PIN_1 , ci -> DIO_PIN_2 , ci -> DIO_PIN_3 ,
            ci -> DIO_PIN_4 , ci -> DIO_PIN_5 , ci -> DIO_PIN_6 , ci -> DIO_PIN_7 );

    printf( "DIO pins ( 8 .. 15 ): %2d %2d %2d %2d %2d %2d %2d %2d\n",
            ci -> DIO_PIN_8 , ci -> DIO_PIN_9 , ci -> DIO_PIN_10 , ci -> DIO_PIN_11 ,
            ci -> DIO_PIN_12 , ci -> DIO_PIN_13 , ci -> DIO_PIN_14 , ci -> DIO_PIN_15 );

    printf( "ADC pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
            ci -> ADC_PIN_0 , ci -> ADC_PIN_1 , ci -> ADC_PIN_2 , ci -> ADC_PIN_3 );

    printf( "PWM pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
            ci -> PWM_PIN_0 , ci -> PWM_PIN_1 , ci -> PWM_PIN_2 , ci -> PWM_PIN_3 );

    printf( "UART RX pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
            ci -> UART_RX_PIN_0 , ci -> UART_RX_PIN_1 , ci -> UART_RX_PIN_2 , ci -> UART_RX_PIN_3 );

    printf( "UART TX pins ( 0 .. 3 ): %2d %2d %2d %2d\n",
            ci -> UART_TX_PIN_0 , ci -> UART_TX_PIN_1 , ci -> UART_TX_PIN_2 , ci -> UART_TX_PIN_3 );

    printf( "SPI0 Pins: MOSI: %2d, MISO: %2d, SCLK: %2d \n",
            ci -> SPI_MOSI_PIN_0 , ci -> SPI_MISO_PIN_0 , ci -> SPI_SCLK_PIN_0 );

    printf( "SPI1 Pins: MOSI: %2d, MISO: %2d, SCLK: %2d \n",
            ci -> SPI_MOSI_PIN_1 , ci -> SPI_MISO_PIN_1 , ci -> SPI_SCLK_PIN_1 );

    printf( "NVM I2C Pins: SCL: %2d, SDA: %2d, I2C Root: 0x%x \n",
            ci -> NVM_I2C_SCL_PIN , ci -> NVM_I2C_SDA_PIN , ci -> NVM_I2C_ADR_ROOT );

    printf( "EXT I2C Pins: SCL: %2d, SDA: %2d, I2C Root: 0x%x \n",
            ci -> EXT_I2C_SCL_PIN , ci -> EXT_I2C_SDA_PIN , ci -> EXT_I2C_ADR_ROOT );

    printf( "\n" );

}

}; // namespace CDC
```

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.
//
//----------------------------------------------------------------------------------------------------
```

```
enum LcsEventId : uint16_t {

    NIL_EVENT_ID  = 0,
    MIN_EVENT_ID  = 1,
    MAX_EVENT_ID  = 65535
};

//--------------------------------------------------------------------------------------------------------------
// Each locomotive has a type. There are STEAM, DIESEL and ELECTRIC engines so far.
//
// ??? additional types ?
//--------------------------------------------------------------------------------------------------------------
enum DccLocoType : uint8_t {

    LOC_T_NIL       = 0,
    LOC_T_STEAM     = 1,
    LOC_T_DIESEL    = 2,
    LOC_T_ELECTRIC  = 3
};

//--------------------------------------------------------------------------------------------------------------
// The base station maintains the locomotive sessions. A session is assigned by the base station and commands
// for the locomotive use this session number. Session Ids start with 1, up to 255 simultaneous sessions are
//  supported.
//
//--------------------------------------------------------------------------------------------------------------
enum DccSessionId : uint8_t {

    NIL_LOCO_SESSION_ID  = 0,
    MIN_LOCO_SESSION_ID  = 1,
    MAX_LOCO_SESSION_ID  = 255
};

//--------------------------------------------------------------------------------------------------------------
// The cabId is the locomotive number or address. For DCC type locomotives, there is a short and long address
// for a decoder. The short address ranges from 1 .. 127, the long address from 1 .. to 10239. However, most
// base stations support just up to 9999 locomotives IDs and so do we. Analog engines do not really have a
// cabId. When an analog engines is introduced to the system by telling the base station about the engine and
// lo cation, the base station assign a cabId. Refer to the book for the details.
//
//--------------------------------------------------------------------------------------------------------------
enum DccCabId : uint16_t {

    NIL_CAB_ID  = 0,
    MIN_CAB_ID  = 1,
    MAX_CAB_ID  = 9999
};

//--------------------------------------------------------------------------------------------------------------
// A DCC decoder features configuration variables, called CVs. CVs are numbered starting with 1, the maximum
// number is 1024.
//
//--------------------------------------------------------------------------------------------------------------
enum DccCvId : uint16_t {

    NIL_DCC_CV_ID  = 0,
    MIN_DCC_CV_ID  = 1,
    MAX_DCC_CV_ID  = 1024
};

//--------------------------------------------------------------------------------------------------------------
// A locomotive decoder features up to 69 functions Ids. They are numbered from 0 to 68.
//
//--------------------------------------------------------------------------------------------------------------
enum DccFuncId : uint8_t {

    NIL_DCC_FUNC_ID   = 255,
    MIN_DCC_FUNC_ID   = 0,
    MAX_DCC_FUNC_ID   = 68
};

//--------------------------------------------------------------------------------------------------------------
// According to the DCC standard, DCC decoder functions are grouped in ten groups, labelled from 1 to 10 .
//
//--------------------------------------------------------------------------------------------------------------
enum DccFuncGroupId : uint8_t {

    MIN_DCC_FUNC_GROUP_ID   = 1,
    MAX_DCC_FUNC_GROUP_ID   = 10
};

//--------------------------------------------------------------------------------------------------------------
// DCC decoder function mapping Ids. The LCS system defines a set of functions used by the handhelds such
// as horn, lights and so on. These identifiers are standardized for our handhelds and mapped to the DCC
// function.
//
//--------------------------------------------------------------------------------------------------------------
enum LcsDccFuncId : uint8_t {

    NIL_LCS_DCC_FUNC_ID = 0,
    MIN_LCS_DCC_FUNC_ID = 1,

    // ??? function IDs go here...

    MAX_LCS_DCC_FUNC_ID = 68
};

//--------------------------------------------------------------------------------------------------------------
// "CvModeOptions" is used by the DCC CV variables access routines to specify the access mode. Only options
```

```
// 0 and 1 are supported. The others are there for historic reasons, the functionality was found in older
// decoders and should not be supported anymore.
//
//----------------------------------------------------------------------------------------------------------
enum DccCvModeOptions : uint8_t {

    CVM_BYTE      = 0,
    CVM_BIT       = 1,
    CVM_PAGE      = 2,
    CVM_REGISTER  = 3,
    CVM_ADR_ONLY  = 4
};

//----------------------------------------------------------------------------------------------------------
// The DCC standard defines several speed step modes. Today, the 28 speed step option is the one used in all
// new decoders. The other speed steps are mapped to the 128 value range.
//
//----------------------------------------------------------------------------------------------------------
enum DccSpeedSteps : uint8_t {

    DCC_SPEED_STEPS_14    = 1,
    DCC_SPEED_STEPS_28    = 2,
    DCC_SPEED_STEPS_128   = 3
};

//----------------------------------------------------------------------------------------------------------
// The locomotive decoder speed. The range is defined for a DCC 128 speed step decoder, from 0 to 127. The
// speed of 1 represents the emergency speed stop. In normal operations, speed stops would thus go from 2
// to 0 and back. For analog engines, we keep this scheme and map it to the respective power levels.
//
//----------------------------------------------------------------------------------------------------------
enum LocSpeed : uint8_t {

    MIN_LOCO_SPEED      = 0,
    ESTOP_LOCO_SPEED    = 1,
    MAX_LOCO_SPEED      = 127
};

//----------------------------------------------------------------------------------------------------------
// Locomotive direction.
//
//----------------------------------------------------------------------------------------------------------
enum LocoDirection : uint8_t {

    LOCO_DIR_LOCO_NEUTRAL  = 0,
    LOCO_DIR_LOCO_FORWARD  = 1,
    LOCO_DIR_LOCO_REVERSE  = 2
};

//----------------------------------------------------------------------------------------------------------
// "LocSessionModes" specify the options when creating a session for the loco. Besides creating a normal
// session an existing session can be taken over or even shared among multiple handhelds.
//
//----------------------------------------------------------------------------------------------------------
enum LocoSessionModes : uint8_t {

    LSM_NORMAL  = 1,
    LSM_STEAL   = 2,
    LSM_SHARED  = 3
};

//----------------------------------------------------------------------------------------------------------
// The defined board types. When the runtime is initialized, the firmware will pass the type to specify what
// board it expects. This value is compared to what is actually stored in the NVM of the main controller
// board. If they don't match, it is considered an error and the NVM needs to be configured to support the
// firmware.
//
//----------------------------------------------------------------------------------------------------------
enum LcsBoardType : uint16_t {

    BT_NIL                   = 0,
    BT_MAIN_CONTROLLER       = 1,
    BT_BASE_STATION          = 2,
    BT_BLOCK_CONTROLLER      = 3,
    BT_CAB_HANDHELD          = 4,

    BT_EXT_NIL               = 10,
    BT_EXT_OCC_DETECT        = 11,
    BT_EXT_SERVO             = 12,
    BT_EXT_GPIO              = 13
};

//----------------------------------------------------------------------------------------------------------
// The defined chip families. There are controller chip families such as the controller family RP2040, or
// chip families for the NVM chips used, and so on.
//
//----------------------------------------------------------------------------------------------------------
enum LcsControllerFamilyType : uint16_t {

    CF_FAM_NIL               = 0,
    CF_FAM_RPICO             = 1,
    CF_FAM_MICROCHIP         = 3,
    CF_FAM_NXP               = 4
};

//----------------------------------------------------------------------------------------------------------
// Extension board driver flags. The flag are set when a board is detected and also during board operation.
//
//----------------------------------------------------------------------------------------------------------
```

```
enum LcsBoardFlags : uint16_t {

    BF_NIL                      = 0,
    BF_EXT_BOARD_PRESENT        = ( 1U << 0 ),
    BF_EXT_BOARD_VALID          = ( 1U << 1 ),
    BF_EXT_BOARD_READY          = ( 1U << 2 )
};

//-----------------------------------------------------------------------------------------------------------
// The configuration descriptor and node map have an option field. The following constants define the
// options that can be set.
//
//   NOPT_SKIP_NODE_ID_CONFIG - during startup, skip the nodeId configuration protocol.
//   NOPT_SKIP_NODE_INIT_STEP - during startup, skip the node initialization step.
//   NOPT_SKIP_PORT_INIT_STEP - during startup, skip the port initialization step.
//   NOPT_DEBUG_DURING_SETUP  - during startup print debug info until we use the mask of nodeMap
//
//-----------------------------------------------------------------------------------------------------------
enum LcsNodeOptions : uint16_t {

    NOPT_NIL                    = 0,
    NOPT_SKIP_NODE_ID_CONFIG    = ( 1 << 0 ),
    NOPT_SKIP_NODE_INIT_STEP    = ( 1 << 1 ),
    NOPT_DEBUG_DURING_SETUP     = ( 1 << 2 ),
    NOPT_FORMAT_RUNTIME         = ( 1 << 3 )
};

//-----------------------------------------------------------------------------------------------------------
// Node Flags. Flags are initialized at library startup and represent library state information.
//
//   NFLAGS_EXT_PRESENT         - extension boards are present.
//   NFLAGS_NVM_WRITE_ENABLED   - write to the protected NVM areas is enabled.
//
//
//-----------------------------------------------------------------------------------------------------------
enum LcsNodeFlags : uint16_t {

    NFLAGS_NIL                  = 0,
    NFLAGS_EXT_PRESENT          = ( 1 << 0 ),
    NFLAGS_NVM_WRITE_ENABLED    = ( 1 << 1 )
};

//-----------------------------------------------------------------------------------------------------------
// Nodes, ports and drivers are accessed with three main routines, GET, SET and REQ.
//
// GET - the get routine will use the item numbers to retrieve the data labelled by the item.
//
// SET - the set routine will use the item numbers to set the value. Note that not all items that can be
// read can also be written to. An attempt will result in an error return.
//
// REQ - the request call will transmit the request parameters to the node / port / driver where a registered
// callback or the driver entry point will be invoked. The result is returned via the parameters.
//
// One argument is the item. Items range from  0 ... 255 and are defined as follows:
//
//    0          -   NIL item, not used
//    1  ..  63  -   Node / port / driver reserved area items, global items for GET/SET/REQ requests.
//   64  .. 127  -   User defined items, specific meaning, accessed via the REQ routine.
//  128  .. 191  -   Node / port / driver data attributes returned from MEM for GET/SET.
//  192  .. 255  -   Node / port / driver data attributes copied from NVM to MEM for GET, copied from MEM to NVM
//                   for SET. The item range mirrors items 128 - 191. For example, 128 and 192 refer to the same
//                   attribute. Note that for a SET on a driver the HW needs to be enabled.
//
// The following declarations does just list the item numbers defined. The ranges are defined in the internal
// include file. The ranges as well as the reserved items defined here should not be tampered with.
//
// ??? to be sorted when more stable...
//-----------------------------------------------------------------------------------------------------------
enum LcsItems : uint8_t {

    ITEM_ID_OPTIONS                     = 1,
    ITEM_ID_FLAGS                       = 2,
    ITEM_ID_DEBUG_MASK                  = 3,
    ITEM_ID_VERSION                     = 4,
    ITEM_ID_TYPE                        = 5,

    ITEM_ID_BOARD_VERSION               = 6,
    ITEM_ID_CONTROLLER_FAMILY           = 7,
    ITEM_ID_NVM_CHIP_FAMILY             = 8,

    ITEM_ID_NODE_ID                     = 10,
    ITEM_ID_NODE_UID                    = 11,
    ITEM_ID_RESTART_COUNT               = 12,

    ITEM_ID_PORT_MAP_ENTRIES            = 13,
    ITEM_ID_EVENT_MAP_ENTRIES           = 14,
    ITEM_ID_ATTR_MAP_ENTRIES            = 15,

    ITEM_ID_NAME_1                      = 17,
    ITEM_ID_NAME_2                      = 18,
    ITEM_ID_NAME_3                      = 19,
    ITEM_ID_NAME_4                      = 20,

    ITEM_ID_EVENT_DELAY_TICKS           = 21,

    ITEM_ID_RESET                       = 22,
    ITEM_ID_SYNC                        = 23,
    ITEM_ID_FORMAT                      = 24,

```

```
    ITEM_ID_ADD_EVENT_MAP_ENTRY         = 25,
    ITEM_ID_DEL_EVENT_MAP_ENTRY         = 26,
    ITEM_ID_GET_EVENT_MAP_ENTRY         = 27,

    ITEM_ID_SET_READY_LED               = 30,
    ITEM_ID_SET_ACTIVITY_LED            = 31,
    ITEM_ID_TOGGLE_READY_LED            = 32,
    ITEM_ID_TOGGLE_ACTIVITY_LED         = 33,

    ITEM_ID_NVM_PROTECTED_ACCESS        = 35,

    ITEM_ID_ENABLE_EVENT_PROCESSING     = 40,

    // ??? add stop and enable periodic processing ?
};

//----------------------------------------------------------------------------------------------------------
// The portMap entry has a flag field. The constants defined here indicate the bit positions and fields
// defined.
//
//  PF_PORT_ENABLED                 - the port is initialized and active
//  PF_PORT_EVENT_HANDLING_ENABLED  - the port has event handling enabled
//  PF_EVENT_PENDING                - an event has been received for this port and is pending.
//
//----------------------------------------------------------------------------------------------------------
enum LcsPortFlags : uint16_t {

    PF_NIL                          = 0,
    PF_PORT_ENABLED                 = ( 1 << 15 ),
    PF_PORT_EVENT_HANDLING_ENABLED  = ( 1 << 14 ),
    PF_EVENT_PENDING                = ( 1 << 13 )
};

//----------------------------------------------------------------------------------------------------------
// The port event action. When an event is received, it will be of the type shown below. There is an
// optional port specific time delay configured between the actual receipt of an event message and the
// invocation of the event callback.
//
//  PEA_EVENT_IDLE                - the port is idle.
//  PEA_EVENT_ON                  - an "ON" event was received.
//  PEA_EVENT_OFF                 - an "OFF" event was received.
//  PEA_EVENT_EVT                 - an event with additional arguments was received.
//
//----------------------------------------------------------------------------------------------------------
enum LcsPortEventAction : uint8_t {

    PEA_EVENT_IDLE    = 0,
    PEA_EVENT_ON      = 1,
    PEA_EVENT_OFF     = 2,
    PEA_EVENT_EVT     = 3
};

//------------------------------------------------------------------------------------------------------------
// The debug mask. The library has a debug mask where each major part of the library has a flag. There could
// also be flags reserved for the firmware. There is an ITEM to read and set this mask. Wherever debugging 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 few instructions. In other words we do not take out debugging code
// when going into production. Never liked this approach of conditional debug anyway.
//
// The usage of the debug mask is generally:
//
//      if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_xxx )) ....
//
// The DBG_CONFIG bit allows for the entire debugging messages to be enabled or disabled. This feature will
// also be used when we test whether we even have a console or not. If there is no console, all the prints
// will not be executed.
//
//------------------------------------------------------------------------------------------------------------
enum DebugOtions : uint16_t {

    DBG_CONFIG          = ( 1U << 15 ),
    DBG_SETUP           = ( 1U << 0 ),
    DBG_NVM_ACCESS      = ( 1U << 1 ),
    DBG_CAN_BUS         = ( 1U << 2 ),
    DBG_MSG_BUS         = ( 1U << 3 ),
    DBG_ATTRIBUTES      = ( 1U << 4 ),
    DBG_EVENTS          = ( 1U << 5 )
};

//-------------------------------------------------------------------------------------------------------
// The message operation code identifies the LCS bus message. It is always the first data byte of the
// message. We encode the number of payload data bytes in the first three bits of the opCode. For each
// message length there is a maximum of 32 opCode possible. The OPC  macro helps to define the opcodes.
// The first argument is the length of the data bytes, the second the opcodeId within the group.
//
// ??? note: this list is work in progress, please us always the names rather than the numbers.
//----------------------------------------------------------------------------------------------------------
#define OPC( len, id ) ((uint8_t) (( len << 5 ) + ( id & 0x1F )))

enum LcsMsgOpCodes : uint8_t {

    LCS_OP_NO_MSG           = OPC( 0, 0 ),
    LCS_OP_CFG              = OPC( 0, 1 ),
    LCS_OP_OPS              = OPC( 0, 2 ),
    LCS_OP_BON              = OPC( 0, 3 ),
    LCS_OP_BOF              = OPC( 0, 4 ),

    LCS_OP_REL_LOC          = OPC( 1, 1 ),
    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 );
```

```
    typedef uint8_t ( *LcsInitCallback ) ( uint16_t npId );
    typedef uint8_t ( *LcsPfailCallback ) ( uint16_t npId );

    typedef uint8_t ( *LcsReqCallback ) ( uint8_t portId, uint8_t item, uint16_t *arg1, uint16_t *arg2 );
    typedef uint8_t ( *LcsRepCallback ) ( uint8_t portId, uint8_t item, uint16_t arg1, uint16_t arg2, uint8_t ret );

    typedef uint8_t ( *LcsEventCallback ) ( uint16_t npId, uint16_t eId, uint8_t eAction, uint16_t eData );
}

//------------------------------------------------------------------------------------------------------
// A  driver for an extension board is invoked through this routine signature. During setup, the correct
// driver label needs to be set in the driver map.
//
//------------------------------------------------------------------------------------------------------
extern "C" {

    typedef uint8_t ( *LcsDrvReqFunc ) ( uint8_t boardId, uint8_t item, uint16_t *arg1, uint16_t *arg2 );
}

//----------------------------------------------------------------------------------------------------
// "LcsConfigDesc" is the data structure that contains initial data for setting up a node. There is the
// option field with bits.
//
// ??? also add the CAN bus mode ?
//----------------------------------------------------------------------------------------------------
struct LcsConfigDesc {

    uint16_t options  = 0;

};

//------------------------------------------------------------------------------------------------------
// Library functions. The main function are the initialization and start of the LCS runtime. Between "init"
// and "start", the firmware should do its own setup and register the required callbacks. We will not return
// from the "start" routine. All call a plain C style library calls. Since we have only one instance of this
// library, there is not really a need to encapsulate the internal data and function pointers in a structure
// that is passed to each call. However, data structures are kept wherever possible local to the file and
// only structures that are references throughout the library file set are available externally. Perhaps
// one day, we encapsulate all data in a private structure for security reasons. To be determined.
//
//------------------------------------------------------------------------------------------------------
LcsConfigDesc       getConfigDefault( );
uint8_t             initRuntime( LcsConfigDesc *lcsConfig, CDC::CdcConfigDesc *cdcConfig );
void                startRuntime( );

//----------------------------------------------------------------------------------------------------
// Access the node local GET/SET/REQ items. Although we pass the node/port Id, only the port Id is actually
// used to identify whether we refer to a local port or the local node. Accessing a remote node / port is
// implemented with the LCS library message send calls.
//
//----------------------------------------------------------------------------------------------------
uint8_t             nodeGet( uint16_t npId, uint8_t item, uint16_t *arg1, uint16_t *arg2 = nullptr );
uint8_t             nodePut( uint16_t npId, uint8_t item, uint16_t arg1, uint16_t arg2 = 0 );
uint8_t             nodeReq( uint16_t npId, uint8_t item, uint16_t *arg1 = nullptr, uint16_t *arg2 = nullptr );

//----------------------------------------------------------------------------------------------------
// Function registration routines for callbacks, tasks, driver types, etc.
//
//----------------------------------------------------------------------------------------------------
void                registerLcsMsgCallback( LcsMsgCallback functionId );
void                registerDccMsgCallback( LcsMsgCallback functionId );
void                registerCmdCallback( LcsCmdCallback functionId );
void                registerInitCallback( LcsInitCallback handler );
void                registerResetCallback( LcsResetCallback handler );
void                registerPfailCallback( LcsPfailCallback handler );
void                registerEventCallback( LcsEventCallback functionId );
void                registerReqCallback( LcsReqCallback handler );
void                registerRepCallback( LcsRepCallback handler );
uint8_t             registerTaskCallback( LcsTaskCallback task, uint32_t interval = 0 );
uint8_t             registerDrvFunc( uint16_t drvType, LcsDrvReqFunc drvReqFunction );

//----------------------------------------------------------------------------------------------------
// A set of convenience functions to send an LCS message.
//
//----------------------------------------------------------------------------------------------------
uint8_t             sendCfg( uint16_t npId );
uint8_t             sendOps( uint16_t npId );
uint8_t             sendReset( uint16_t npId );
uint8_t             sendBusOn( );
uint8_t             sendBusOff( );
uint8_t             sendPing( uint16_t npId );
uint8_t             sendAck( uint16_t npId );
uint8_t             sendErr( uint16_t npId, uint8_t errCode, uint8_t arg1 = 0, uint8_t arg2 = 0 );

uint8_t             sendReqNodeId( uint16_t npId, uint32_t nodeUID, uint8_t flags );
uint8_t             sendRepNodeId( uint16_t npId, uint32_t nodeUID );
uint8_t             sendSetNodeId( uint16_t npId, uint32_t nodeUID );
uint8_t             sendNodeIdCollision( uint16_t npId, uint32_t nodeUID );

uint8_t             sendGetNode( uint16_t npId, uint8_t item, uint16_t arg1 = 0, uint16_t arg2 = 0 );
uint8_t             sendSetNode( uint16_t npId, uint8_t item, uint16_t arg1 = 0, uint16_t arg2 = 0 );
uint8_t             sendRepNode( uint16_t npId, uint8_t item, uint16_t val1, uint16_t val2  );
uint8_t             sendReqNode( uint16_t npId, uint8_t item, uint16_t val1, uint16_t val2  );

uint8_t             sendEventOn( uint16_t npId, uint16_t eventId );
uint8_t             sendEventOff( uint16_t npId, uint16_t eventId );
uint8_t             sendEvent( uint16_t npId, uint16_t eventId, uint16_t arg );

uint8_t             sendTrackOn( );
```

```
uint8_t              sendTrackOff( );
uint8_t              sendEstop( );

uint8_t              sendReqLoc( uint16_t locAdr, uint8_t flags  );
uint8_t              sendRelLoc( uint8_t sId  );
uint8_t              sendRepLoc( uint8_t sId, uint16_t locAdr, uint8_t spDir, uint8_t fn1 = 0, uint8_t fn2 = 0, uint8_t fn3 = 0 );
uint8_t              sendLocConsist( uint8_t sId, uint8_t consId, uint8_t flags );
uint8_t              sendQueryLoc( uint8_t sId  );
uint8_t              sendKeepLoc( uint8_t sId  );
uint8_t              sendSetLocSpDir( uint8_t sId, uint8_t spDir );
uint8_t              sendSetLocMode( uint8_t sId, uint8_t mode );
uint8_t              sendSetLocFuncOn( uint8_t sId, uint8_t fNum );
uint8_t              sendSetLocFuncOff( uint8_t sId, uint8_t fNum );
uint8_t              sendSetLocFgroup( uint8_t sId, uint8_t fGroup, uint8_t data );

uint8_t              sendSetLocCvMain( uint8_t sId, uint16_t cvId, uint8_t mode, uint8_t val );
uint8_t              sendSetLocCvProg( uint16_t cvId, uint8_t mode, uint8_t val );
uint8_t              sendReqLocCvProg( uint16_t cvId, uint8_t mode );
uint8_t              sendRepLocCvProg( uint16_t cvId, uint8_t val );

uint8_t              sendSetBacc( uint16_t accAdr, uint8_t flags  );
uint8_t              sendSetEacc( uint16_t accAdr, uint8_t val  );

uint8_t              sendDccPacket( uint8_t arg1, uint8_t arg2, uint8_t arg3 );
uint8_t              sendDccPacket( uint8_t arg1, uint8_t arg2, uint8_t arg3, uint8_t arg4 );
uint8_t              sendDccPacket( uint8_t arg1, uint8_t arg2, uint8_t arg3, uint8_t arg4, uint8_t arg5 );
uint8_t              sendDccPacket( uint8_t arg1, uint8_t arg2, uint8_t arg3, uint8_t arg4, uint8_t arg5, uint8_t arg6 );

uint8_t              sendDccAck( );
uint8_t              sendDccErr( uint8_t errCode, uint8_t arg1 = 0, uint8_t arg2 = 0 );

uint8_t              sendRawMsg( uint8_t *msgBuf );

void                 printLcsMs( uint8_t *msgBuf );

//----------------------------------------------------------------------------------------------------
// The driver interface. The firmware communicated with an extension board by making calls to the driver
// for the board. Just like for the node / port items, there are routines to GET/SET/REQ driver data and
// functions. The init function is only called by the library at setup time.
//
//----------------------------------------------------------------------------------------------------
uint8_t              drvInit( uint8_t boardId );
uint8_t              drvGet( uint8_t boardId, uint8_t item, uint16_t *arg );
uint8_t              drvPut(uint8_t boardId, uint8_t item, uint16_t arg );
uint8_t              drvReq( uint8_t boardId, uint8_t item, uint16_t *arg1 = nullptr, uint16_t *arg2 = nullptr );

//----------------------------------------------------------------------------------------------------
// The User Map interface. The LCS library offers a set of routines for the firmware to access the user
// NVM area. The size is dependent on what the actual chip on the board offers. The meaning of this data
// area is entirely firmware specific. Note that there are also routines for accessing the runtime data
// area as well as the individual extension board areas. They are declared in the internal include file.
//
//----------------------------------------------------------------------------------------------------
uint8_t              usrNvmPutWord( uint32_t ofs, uint16_t word );
uint8_t              usrNvmGetWord( uint32_t ofs, uint16_t *word );
uint8_t              usrNvmPutBytes( uint32_t ofs, uint8_t *buf, uint32_t len );
uint8_t              usrNvmGetBytes( uint32_t ofs, uint8_t *buf, uint32_t len );
uint8_t              usrNvmInitArea( uint32_t ofs, uint32_t len, uint8_t val);
uint32_t             usrNvmGetSize( );

}; // LCS NameSpace

#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;
```

```
const uint16_t  MAX_ATTR_MAP_ENTRIES            = 64;
const uint16_t  MAX_PORT_MAP_ENTRIES            = 15;
const uint16_t  MAX_EVENT_MAP_ENTRIES           = 1024;
const uint16_t  MAX_TASK_MAP_ENTRIES            = 16;

const uint16_t  MAX_LCS_MSG_SIZE                = 8;
const uint16_t  MAX_NODE_NAME_SIZE              = 16;
const uint16_t  MAX_PORT_NAME_SIZE              = 16;
const uint16_t  MAX_BOARD_NAME_SIZE             = 16;
const uint16_t  MAX_COMMAND_LINE_SIZE           = 256;

const uint16_t  MAX_EXT_BOARD_MAP_ENTRIES       = 4;
const uint16_t  MAX_PENDING_REQ_MAP_ENTRIES     = 8;
const uint16_t  EVENT_DELAY_TICK_MILLIS         = 32;

const uint8_t   MAX_DRV_TYPES                   = 8;
const uint8_t   MAX_EXT_BOARDS                  = 4;
const uint8_t   MAX_DRV_DATA_SIZE               = 64;

const uint16_t  NVM_NODE_MAP_START              = 0;
const uint16_t  NVM_CDC_MAP_START               = 0x200;
const uint16_t  NVM_PORT_MAP_START              = 0x400;
const uint16_t  NVM_NODE_DATA_START             = 0x800;
const uint16_t  NVM_EVENT_MAP_START             = 0x1000;
const uint16_t  NVM_USER_MAP_START              = 0x2000;
const uint16_t  NVM_RUNTIME_AREA_SIZE           = 0x2000;

//----------------------------------------------------------------------------------------------------------
// The nodeMap on NVM has two locations with a "magic" word. We simply read in a nodeMap and check these
// two locations for the magic words. If found, the area was configured before. It would be quite unlikely
// that a random NVM content has these two words at the right spot. In a similar way, we have two magic
// words for the NVM in an extension board. Same idea, same logic. But even if the area was configured
// before, it does not automatically mean that all the data is correct. Further checking will be done
// during startup.
//
//----------------------------------------------------------------------------------------------------------
const uint16_t NVM_MWORD_1 = (uint16_t) ( 'L' << 8 ) + 'C';
const uint16_t NVM_MWORD_2 = (uint16_t) ( 'S' << 8 ) + '0';

//----------------------------------------------------------------------------------------------------------
// The node states. The node starts in the INIT state and once all is initialized and registered ends up in
// the OPS or CFG mode.
//
//   NS_NIL             -
//   NS_FAIL            -   The node startup failed.
//   NS_PFAIL           -   The node startup detected that we come up after a power fail.
//   NS_INIT            -   The node entered the startup state.
//   NS_REGISTER        -   The node entered the node register state, awaiting a nodeId.
//   NS_COLLISION       -   The node detected a nodeId collision on the LCS bus.
//   NS_HALTED          -   The node was halted.
//   NS_CONFIG          -   The node is in configuration mode.
//   NS_OPERATE         -   The node is on operations mode.
//
//----------------------------------------------------------------------------------------------------------
enum LcsNodeState : uint16_t {

    NS_NIL              = 0,
    NS_FAIL             = 1,
    NS_PFAIL            = 2,
    NS_INIT             = 3,
    NS_REGISTER         = 4,
    NS_COLLISION        = 5,
    NS_HALTED           = 6,
    NS_CONFIG           = 7,
    NS_OPERATE          = 8
};

//------------------------------------------------------------------------------------------------------------
// Nodes, ports and drivers are accessed with three main routines, GET, SET and REQ.
//
// GET - the get routine will use the item numbers to retrieve the data labelled by the item.
//
// SET - the set routine will use the item numbers to set the value. Note that not all items that can be
// read can also be written to. An attempt will result in an error return.
//
// REQ - the request call will transmit the request parameters to the node / port / driver where a registered
// callback or the driver entry point will be invoked. The result is returned via the parameters.
//
// One argument is the item. Items range from  0 ... 255 and are defined as follows:
//
//    0        -    NIL item, not used
//    1  ..  63  -  Node / port / driver reserved area items, global items for GET/SET/REQ requests.
//   64  .. 127  -  User defined items, specific meaning, accessed via the REQ routine.
//  128  .. 191  -  Node / port / driver data attributes returned from MEM for GET/SET.
//  192  .. 255  -  Node / port / driver data attributes copied from NVM to MEM for GET, copied from MEM to NVM
//                  for SET. The item range mirrors items 128 - 191. For example, 128 and 192 refer to the same
//                  attribute. Note that for a SET on a driver the HW needs to be enabled.
//
// The items are defined in the external include file. This part here defined the boundaries for internal
// checking.
//
//------------------------------------------------------------------------------------------------------------
enum ItemRanges : uint8_t {

    IR_NIL                      = 0,

    IR_LIB_MAP_RANGE_START      = 1,
    IR_LIB_MAP_RANGE_END        = 63,

```

```
    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 };
};
```

```
struct LcsPortMap {

    LcsPortMapEntry map[ MAX_PORT_MAP_ENTRIES ];
};

//----------------------------------------------------------------------------------------------
// The event map entry contains the mapping from eventId to portId. Every port interested in a certain event
// will have an entry in the event map. It is a sorted table of event and port pairs. A port id of zero
// refers to all ports with the event Id. This table is searched for an incoming event to find the ports
// that are interested in the event.
//
//----------------------------------------------------------------------------------------------
struct LcsEventMapEntry {

    uint16_t eventId  = NIL_EVENT_ID;
    uint16_t portId   = NIL_PORT_ID;
};

struct LcsEventMap {

    LcsEventMapEntry    map[ MAX_EVENT_MAP_ENTRIES ];
};

//----------------------------------------------------------------------------------------------
// The first locations of the NVM area on the controller board NVM chip represent the nodeMap. It is the
// heart of all data on the node. When bringing up a node, we read in the node map from the NVM. The first
// check is whether the nodeMap read is a valid nodeMap.
//
//----------------------------------------------------------------------------------------------
struct LcsNodeMap {

    //------------------------------------------------------------------------------------------
    // NMV header. We read this in first an check for validity.
    //------------------------------------------------------------------------------------------
    LcsNvmHeader    head;

    //------------------------------------------------------------------------------------------
    // Node data.
    //
    //------------------------------------------------------------------------------------------
    uint16_t        nodeState                       = NS_NIL;
    uint16_t        nodeOptions                     = 0;
    uint16_t        nodeFlags                       = 0;
    uint16_t        nodeId                          = NIL_NODE_ID;
    uint32_t        nodeUID                         = 0L;
    uint16_t        nodeType                        = NIL_NODE_TYPE;
    uint16_t        nodeSwVersion                   = 0;
    uint16_t        nodeSwPatchLevel                = 0;
    uint16_t        nodeRestartCnt                  = 0;
    uint32_t        nodeSystemTime                  = 0;
    uint16_t        nodeMapSize                     = sizeof( LcsNodeMap );
    char            name[ MAX_NODE_NAME_SIZE ]      = { 0 };

    //------------------------------------------------------------------------------------------
    // Runtime area offsets in the NVM. We also keep track of the data structure sizes and check when we
    // read in the maps that they match a give library version.
    //
    //------------------------------------------------------------------------------------------
    uint16_t        nvmNodeMapOfs                   = NVM_NODE_MAP_START;
    uint16_t        nvmNodeMapSize                  = sizeof( LcsNodeMap );

    uint16_t        nvmCdcMapOfs                    = NVM_CDC_MAP_START;
    uint16_t        nvmCdcMapSize                   = sizeof( LcsCdcMap );

    uint16_t        nvmPortMapOfs                   = NVM_PORT_MAP_START;
    int16_t         nvmPortMapSize                  = sizeof( LcsPortMap );

    uint16_t        nvmNodeDataOfs                  = NVM_NODE_DATA_START;
    uint16_t        nvmNodeDataSize                 = sizeof( LcsNodeData );

    uint16_t        nvmEventMapOfs                  = NVM_EVENT_MAP_START;
    uint16_t        nvmEventMapSize                 = sizeof( LcsEventMap );

    uint16_t        nvmUserMapOfs                   = NVM_USER_MAP_START;
    uint16_t        nvmUserMapSize                  = 0;

    uint32_t        nvmMemSize                      = NVM_RUNTIME_AREA_SIZE;

    //------------------------------------------------------------------------------------------
    // The number of entries in the core areas and a high water mark.
    //
    //------------------------------------------------------------------------------------------
    uint16_t        portMapEntries                  = MAX_PORT_MAP_ENTRIES;
    uint16_t        portMapHwm                      = 0;

    uint16_t        eventMapEntries                 = MAX_EVENT_MAP_ENTRIES;
    uint16_t        eventMapHwm                     = 0;

    uint16_t        taskMapEntries                  = MAX_TASK_MAP_ENTRIES;
    uint16_t        taskMapHwm                      = 0;

    uint16_t        pendingMapEntries               = MAX_PENDING_REQ_MAP_ENTRIES;
    uint16_t        pendingMapHwm                   = 0;

    uint16_t        drvFuncMapEntries               = MAX_DRV_TYPES;
    uint16_t        drvFuncMapHwm                   = 0;

    uint16_t        drvMapEntries                   = MAX_EXT_BOARD_MAP_ENTRIES;
```

```
    uint16_t        drvMapHwm                       = 0;
};

//---------------------------------------------------------------------------------------------------------
// The LCS runtime communicates back to the firmware via callbacks. There are global callbacks for message
// receipt and events as well as callbacks for the node and ports that can be registered.
//
//---------------------------------------------------------------------------------------------------------
struct LcsCallbackMap {

    LcsMsgCallback          lcsMsgCallback          = nullptr;
    LcsMsgCallback          dccMsgCallback          = nullptr;
    LcsCmdCallback          cmdLineCallback         = nullptr;
    LcsEventCallback        eventCallback           = nullptr;

    LcsInitCallback         initCallback            = nullptr;
    LcsResetCallback        resetCallback           = nullptr;
    LcsPfailCallback        pfailCallback           = nullptr;

    LcsReqCallback          reqCallback             = nullptr;
    LcsRepCallback          repCallback             = nullptr;
};

//---------------------------------------------------------------------------------------------------------
// The core library maintains an array of periodic task items. To balance the needs of other core library
// internal periodic tasks, such as checking for incoming messages, the periodic tasks are run one at a
// time, round robin, with the other internal tasks interleaving. The structure maintains the task procedure
// label, the time it ran the last time, and the interval between invocations. Note that the timing is not
// very accurate, but it is guaranteed that a task will eventually run when the interval is reached.
//
//---------------------------------------------------------------------------------------------------------
struct LcsPTaskMapEntry {

    LcsTaskCallback     task            = nullptr;
    uint32_t            timeStamp       = 0;
    uint32_t            interval        = 0;
};

struct LcsTaskMap {

    LcsPTaskMapEntry    map[ MAX_TASK_MAP_ENTRIES ];
};

//---------------------------------------------------------------------------------------------------------
// The pending request map keeps track of outstanding requests to another node. We add an entry when our
// node sends a "REQ" type packet and clear the entry when the reply comes in. The idea is that we only
// invoke the callback when we expect a reply. Additionally, there is a timeout value, so that we can
// can invoke the reply callback with a timeout message if requested.
//
//---------------------------------------------------------------------------------------------------------
struct LcsPendingReqEntry {

    uint16_t  npId;
    int32_t   reqTimeoutTs;
};

struct LcsPendingReqMap {

    LcsPendingReqEntry map[ MAX_PENDING_REQ_MAP_ENTRIES ];
};

//-----------------------------------------------------------------------------------------------------------
// Each extension board will have a NVM to store the board configuration data. Similar to the node map of
// the controller board, this extension board will have a data structure that is read at initialization time.
// The structure of this data is rather simple. We have the common 8-word header which describes the board in
// general an area which contains driver relevant information. The board type will tell the setup routines
// what driver to load for the extension board. The driver data area is entirely driver specific and the
// meaning is only know to the driver software. At startup time, all we have to do then is locate the board
// type, load the  respective driver and let the driver code do whatever needs to be done according to the
// data area content.
//
// Note that the extension board NVM data is "read only". To write to it, a jumper is set on the board. The
// data area is configured and then the jumper should be removed. This does however not mean that that data
// once it is loaded during setup cannot be changed during operations. For example, the driver area is the
// "working area" for the driver to keep temporary values. At node restart, all data is set back to the NVM
//  data configured on the extension board chip.
//
//-----------------------------------------------------------------------------------------------------------
struct LcsDrvBoardDesc {

    LcsNvmHeader    head;
    uint16_t        driverData[ MAX_DRV_DATA_SIZE ]  = { 0 };
};

//---------------------------------------------------------------------------------------------------------
// An extension board is accessed via a dedicated driver. The firmware is required to register the available
// drivers with the runtime. The type and function label are kept in the driver label map. This data is
// used when a board os detected to select the correct driver.
//
//---------------------------------------------------------------------------------------------------------
struct LcsDrvFuncEntry {

    uint16_t        drvType = BT_NIL;
    LcsDrvReqFunc   drvFunc = nullptr;
};

struct LcsDrvFuncMap {

    LcsDrvFuncEntry map[ MAX_DRV_TYPES ] = { 0 };
```

```
};

//-----------------------------------------------------------------------------------------------------------
// The runtime library maintains a driver table, which has for each of the extension boards an entry. The
// first board has an index of zero.  While the drivers are set regardless of the order of the extension
// boards, the boardId would change with the order of extension boards connected. A firmware either needs
// to insist on the correct order or map the extension boards regardless of order.
//
// The entry contains a set of flags about the driver, the procedure label for the driver code and the
// extension board descriptor, which is read in from the extension board NVM area. During startup all
// extension boards will be located, if there are any. For each board the correct driver procedure label
// will be stored in the driver map entry.
//
// If the extension board descriptor is invalid, the driver map entry is marked as failed. We can however
// still access the data area from configuration tools, when the jumper to enable writing to the board is
// set.
//
//-----------------------------------------------------------------------------------------------------------
struct LcsDrvEntry {

    uint16_t            flags       = 0;
    uint16_t            lastErr     = 0;
    LcsDrvReqFunc       drvFunc     = nullptr;

    LcsDrvBoardDesc     extBoard;
};

struct LcsDrvMap {

    LcsDrvEntry     map[ MAX_EXT_BOARDS ];
};

//-----------------------------------------------------------------------------------------------------------
// The LCS runtime routine signatures of routines used across the different source files.
//
// ??? keep this list short... maybe keep local to each file....
//-----------------------------------------------------------------------------------------------------------
uint8_t     configNvm( CDC::CdcConfigDesc *ci );

uint8_t     rtNvmPutWord( uint32_t ofs, uint16_t word );
uint8_t     rtNvmGetWord( uint32_t ofs, uint16_t *word );
uint8_t     rtNvmPutBytes( uint32_t ofs, uint8_t *buf, uint32_t len );
uint8_t     rtNvmGetBytes( uint32_t ofs, uint8_t *buf, uint32_t len );
uint8_t     rtNvmClearArea( uint32_t ofs, uint32_t len, uint8_t val = 0 );
uint32_t    rtNvmGetSize( );

uint8_t     extNvmPutWord( uint8_t boardId, uint32_t ofs, uint16_t word );
uint8_t     extNvmGetWord( uint8_t boardId, uint32_t ofs, uint16_t *word );
uint8_t     extNvmPutBytes( uint8_t boardId, uint32_t ofs, uint8_t *buf, uint32_t len );
uint8_t     extNvmGetBytes( uint8_t boardId, uint32_t ofs, uint8_t *buf, uint32_t len );
uint8_t     extNvmClearArea( uint8_t boardId, uint32_t ofs, uint32_t len, uint8_t val = 0 );
uint32_t    extNvmGetSize( );

uint8_t     resetNode( uint16_t npId );

uint8_t     syncEventMap( );
uint8_t     addEvent( uint16_t eventId, uint16_t portId = NIL_PORT_ID );
uint8_t     removeEvent( uint16_t eventId, uint16_t portId = NIL_PORT_ID );
int         searchEvent( uint16_t eventId, uint16_t portId = NIL_PORT_ID );
uint8_t     getMemEmapEntry( uint16_t index, uint16_t *evId, uint16_t *pId );

void        handleMsgLcsMgt( uint8_t *msg );
void        handleMsgEvent( uint8_t *msg );

uint8_t     setupSerialCommand( );
uint8_t     handleSerialCommand( );

void        handleNodeState( );

} // namespace LCS

#endif
```

```
//----------------------------------------------------------------------------------------------------------
//
// "LcsMsgBusCAN" - CAN Bus Interface for Raspberry PI Pico
//
//----------------------------------------------------------------------------------------------------------
// The "LcsMsgBusCAN" object implements the LCS message bus as a CAN bus. The CAN bus is a widely established
// bus, which is quite robust. We use the standard CAN bus with a maximum CAN Id of 29 bits. In our case the
// 16 bit node / port ID along with a 2 bit priority field is used as the CAN address.
//
// On the PICO, there is a library, "can2040", available that implements the CAN bus protocol in software,
// using the PICO PIO state machines. This saves us an external controller. In addition, we allow for the
// option to run the CAN bus state machine on a separate core. This is highly recommend as the LCS Runtime
// has a lot of other things to do. Using a queue from the PICO C++ SDK, the core running the CAN state
// machine will just queue the received message to be picked up by the other core when ready.
//
//----------------------------------------------------------------------------------------------------------
//
// LCS - Can Bus Interface Library
// 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 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 <stdio.h>
#include "pico/stdlib.h"
#include "pico/stdio.h"
#include "pico/util/queue.h"
#include "pico/multicore.h"

#include "LcsRuntimeLib.h"
#include "LcsRtLibInt.h"

//----------------------------------------------------------------------------------------------------------
// The can2040 is a C library. Make it extern C, otherwise the linker gets confused.
//
//----------------------------------------------------------------------------------------------------------
extern "C" {

    #include "./Can2040Lib/can2040.h"
}

//----------------------------------------------------------------------------------------------------------
// The debug mask. See the internal include file for details.
//
//----------------------------------------------------------------------------------------------------------
namespace LCS {

    extern uint16_t debugMask;
};

//----------------------------------------------------------------------------------------------------------
// The name space for file local declarations.
//
//----------------------------------------------------------------------------------------------------------
namespace {

using namespace LCS;

//----------------------------------------------------------------------------------------------------------
// The maximum message length of a CAN bus ( and LCS ) message. The LCS library still uses the "classic"
// CAN bus message size. Finally, the CAN bus library for the RP2040 needs a static opaque structure. We also
// need a receiver queue for storing the received messages when they come in.
//
//----------------------------------------------------------------------------------------------------------
const uint8_t   MAX_CAN_MSG_SIZE  = 8;
const uint8_t   RX_QUEUE_SIZE     = 4;
const uint8_t   TX_RETRY_TIMEOUT  = 5;

//----------------------------------------------------------------------------------------------------------
// The setup and start of the CAN Bus can run on ether core 0 or core 1, depending whether a multi-core
// implementation is desired. The "Can2040ConfigDesc" structure holds all the necessary configuration data
// for the initialization routine to use.
//
//----------------------------------------------------------------------------------------------------------
struct Can2040ConfigDesc {

    uint32_t        mcPioNum;
    uint32_t        mcSysClock;
    uint32_t        mcBitRate;
    uint32_t        mcRxPin;
    uint32_t        mcTxPin;
    can2040_rx_cb   mcRxCallback;
    uint32_t        mcRxQueueSize;
    bool            mcRunOnCore1;
    bool            mcSetupOK;
};

//----------------------------------------------------------------------------------------------------------
// File local variables and constants.
//
//----------------------------------------------------------------------------------------------------------
```

```
Can2040ConfigDesc        cfg;
struct can2040           cBus;
queue_t                  rxQueue;

//----------------------------------------------------------------------------------------------------------
// The "buildCanBusMsgHeader" constructs the canId header for the message. It encodes the canId itself and
// flags such as EXT or RTR.
//
//----------------------------------------------------------------------------------------------------------
inline uint32_t buildCanBusMsgHeader( uint16_t canId, uint8_t msgPri, bool RTR = false ) {

    uint32_t header = canId | ((uint32_t)( msgPri & 0x3 ) << 16 ) | 0x80000000;

    if ( RTR ) header |=  0x40000000;

    return ( header );
}

//----------------------------------------------------------------------------------------------------------
// The interrupt signature to register with the RP2040 for PIO interrupts. The interrupt handler itself is
// provided by the can2040 library.
//
//----------------------------------------------------------------------------------------------------------
void CanBusPIOIrqHandler( ) {

    can2040_pio_irq_handler( &cBus );
}

//----------------------------------------------------------------------------------------------------------
// For each messages transmitted or received this callback is invoked from within the interrupt handler, so
// all we can do is a quick non-blocking action. The callback allows to react to a message sent, a message
// received and an internal buffer overflow error.
//
// The callback could be used to filter messages directly at this stage. Only messages that concern this
// node should be processed. Easy said, but perhaps no so easy to do. We can basically to filtering at the
// higher message bus level or at the lower layers. The benefit for doing it here is that when we run at the
// other core, the main core is relieved even further. To think about one day.
//
//----------------------------------------------------------------------------------------------------------
void canBusEventCallback( struct can2040 *cd, uint32_t notify, struct can2040_msg *msg ) {

    if ( notify == CAN2040_NOTIFY_RX ) {

        if ( ! queue_try_add( &rxQueue, msg )) {

            // ??? we could not add ... what to do ?
        }
    }
    else if ( notify == CAN2040_NOTIFY_TX ) {

        // ??? transmit completed successfully
    }
    else if ( notify == CAN2040_NOTIFY_ERROR ) {

        // ??? internal buffer overflow ... what to do ?
    }
}

//----------------------------------------------------------------------------------------------------------
// "canBusCore" is the routine that encapsulates the can2040 setup and launch work. For the multi-core
// version it needs to be a routine that can be called from the current core or be launched on the other
// core. The routine communicates the successful setup with a boolean flag in the configuration descriptor.
// Note that the setup routine must be a void procedure with no parameters. This is expected by the launch
// routine in the PICO C++ SDK.
//
//----------------------------------------------------------------------------------------------------------
void canBusCore( ) {

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_CAN_BUS )) {

        printf( "canBusSetup -> pio: %d, clk: %d, bitRate: %d, rxPin: %d, txPin: %d, cb: %u, rxQS: %d, MC: %d\n",
                cfg.mcPioNum, cfg.mcSysClock, cfg.mcBitRate,
                cfg.mcRxPin, cfg.mcTxPin, cfg.mcRxCallback,
                cfg.mcRxQueueSize, cfg.mcRunOnCore1 );
    }

    queue_init( &rxQueue, sizeof( can2040_msg ), cfg.mcRxQueueSize );

    can2040_setup( &cBus, cfg.mcPioNum );
    can2040_callback_config( &cBus, cfg.mcRxCallback );

    if ( cfg.mcPioNum == 0 ) {

        irq_set_exclusive_handler( PIO0_IRQ_0, CanBusPIOIrqHandler );
        irq_set_priority( PIO0_IRQ_0, 1 );
        irq_set_enabled( PIO0_IRQ_0, true );
    }
    else if ( cfg.mcPioNum == 1 ) {

        irq_set_exclusive_handler( PIO1_IRQ_0, CanBusPIOIrqHandler );
        irq_set_priority( PIO1_IRQ_0, 1 );
        irq_set_enabled( PIO1_IRQ_0, true );
    }

    can2040_start( &cBus, cfg.mcSysClock, cfg.mcBitRate, cfg.mcRxPin, cfg.mcTxPin );

    cfg.mcSetupOK = true;

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

```

        printf( "CAN Bus Initialized, runs on Core: %d\n", get_core_num( ));
    }

    if ( cfg.mcRunOnCore1 ) {

        while ( true ) tight_loop_contents( );
    }
}

}; // namespace


//----------------------------------------------------------------------------------------------------------
// The LCS name space CanBus Object methods declared in this file.
//
//----------------------------------------------------------------------------------------------------------
namespace LCS {

//----------------------------------------------------------------------------------------------------------
// "init" is called to setup the CAN bus interface. We will first check the parameters and setup the CAN bus.
// Next set up the interrupt handler and start the CAN bus processing. This is also the time to set the
// initial canId.
//
//----------------------------------------------------------------------------------------------------------
uint8_t LcsMsgBusCAN::init( uint16_t canId, uint8_t rxPin, uint8_t txPin, uint8_t fMode ) {

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_CAN_BUS )) {

        printf( "Init Can Bus -> Node: %d, Rx: %d, Tx: %d, Mode: %d\n", canId, rxPin, txPin, fMode );
    }

    if (( rxPin == CDC::UNDEFINED_PIN ) || ( txPin == CDC::UNDEFINED_PIN )) return ( ERR_CAN_BUS_INIT );

    this -> canId = canId;

    cfg.mcSetupOK       = true;
    cfg.mcRxPin         = rxPin;
    cfg.mcTxPin         = txPin;
    cfg.mcRxCallback    = canBusEventCallback;
    cfg.mcSysClock      = CDC::getCpuFrequency( );
    cfg.mcPioNum        = 0;
    cfg.mcRxQueueSize   = RX_QUEUE_SIZE;

    cfg.mcRunOnCore1    =  (( fMode == CAN_BUS_LIB_PICO_PIO_125K_M_CORE ) ||
                            ( fMode == CAN_BUS_LIB_PICO_PIO_250K_M_CORE ) ||
                            ( fMode == CAN_BUS_LIB_PICO_PIO_500K_M_CORE ) ||
                            ( fMode == CAN_BUS_LIB_PICO_PIO_1000K_M_CORE ));

    switch ( fMode ) {

        case CAN_BUS_LIB_PICO_PIO_125K:
        case CAN_BUS_LIB_PICO_PIO_125K_M_CORE:   cfg.mcBitRate = 125000; break;

        case CAN_BUS_LIB_PICO_PIO_250K:
        case CAN_BUS_LIB_PICO_PIO_250K_M_CORE:   cfg.mcBitRate = 250000; break;

        case CAN_BUS_LIB_PICO_PIO_500K:
        case CAN_BUS_LIB_PICO_PIO_500K_M_CORE:   cfg.mcBitRate = 500000; break;

        case CAN_BUS_LIB_PICO_PIO_1000K:
        case CAN_BUS_LIB_PICO_PIO_1000K_M_CORE:  cfg.mcBitRate = 1000000; break;

        default: cfg.mcSetupOK = false;
    }

    if ( cfg.mcSetupOK ) {

        if ( cfg.mcRunOnCore1 ) multicore_launch_core1( canBusCore );
        else canBusCore( );
    }

    return (( cfg.mcSetupOK ) ? ALL_OK : ERR_CAN_BUS_INIT );
}

//----------------------------------------------------------------------------------------------------------
// "sendLcsMsg" will send a data packet. We are passed the message buffer and the message priority. The
// message length is encoded in the first message byte, which represents the LCS message opCode as well as
// the length of the message. The message has a certain initial priority. When we cannot send the message
// right away, the priority is raised. When we cannot send at the highest priority, the message send
// failed.
//
//----------------------------------------------------------------------------------------------------------
uint8_t LcsMsgBusCAN::sendLcsMsg ( uint8_t *msgBuf, uint8_t msgPri ) {

    can2040_msg msg;

    msg.id  = buildCanBusMsgHeader( canId, msgPri );
    msg.dlc = ( msgBuf[ 0 ] >> 5 ) + 1;

    for ( uint32_t i = 0; i < msg.dlc; i++ ) msg.data[ i ] = msgBuf[ i ];

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_CAN_BUS )) {

        printf( "CAN Send (TS: 0x%x)(Id: 0x%x, Pri: %d)(Data: ", CDC::getMillis( ), canId, msgPri );
        for ( int i = 0; i < msg.dlc; i++ ) printf( " 0x%x", msgBuf[ i ] );
        printf( ")\n" );
    }

```

```
    if ( can2040_transmit( &cBus, &msg ) != 0 ) {

        CDC::sleepMillis( TX_RETRY_TIMEOUT );

        if ( msgPri > MSG_PRI_VERY_HIGH ) return ( sendLcsMsg( msgBuf, msgPri - 1 ));
        else                              return ( ERR_CAN_MSG_SEND );

    } else return ( ALL_OK );
}

//------------------------------------------------------------------------------------------------------
// "receiveLcsMsg" will check for a CAN Bus message and if one is available fill the passed message buffer.
// With the "can2040" library CAN bus messages are received via a callback function, which will store the
// each received message in the local receiver queue.
//
// Besides receiving a message, there is the handling of CAN Id collisions. When we detect a non-zero length
// message with a Can Id that is our own, we have a collision and report an error. This could happen for
// example when a node hardware is connected to another layout. Both nodes will then stop and wait for
// manual resolution.
//
// In addition to message processing, we also need to react to RTR messages. We answer such a request with
// sending a zero length message response. Replying to such a message from other nodes results in a status
// return of "ERR_CAN_MSG_NO_MSG" on this call as no LCS message was actually received. This is also the
// case when the message queue is empty.
//
//------------------------------------------------------------------------------------------------------
uint8_t LcsMsgBusCAN::receiveLcsMsg( uint8_t *msgBuf ) {

    can2040_msg msg;

    if ( queue_try_remove( &rxQueue, &msg )) {

        if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_CAN_BUS )) {

            printf( "CAN Recv (TS: 0x%x)(Id: 0x%x, len: %d)(Data: ", CDC::getMillis( ), msg.id, msg.dlc );
            for ( uint32_t i = 0; i < msg.dlc; i++ ) printf( " 0x%x", msg.data[ i ] );
            printf( ")\n" );
        }

        bool      rtrFlag     = ( msg.id & 0x40000000 );
        bool      extFlag     = ( msg.id & 0x80000000 );
        uint16_t  remoteCanId = (( extFlag ) ? ( msg.id & 0x3FFF ) : ( msg.id & 0x7F ));

        if (( remoteCanId == canId ) && ( msg.dlc > 0 )) {

            return ( ERR_CAN_ID_COLLISION );
        }
        else if ( rtrFlag ) {

            msg.id          = canId;
            msg.dlc         = 0;
            msg.data32[ 0 ] = 0;
            msg.data32[ 1 ] = 0;

            can2040_transmit( &cBus, &msg );
            return ( ERR_CAN_MSG_NO_MSG );
        }
        else {

            memcpy( msgBuf, msg.data, msg.dlc );
            return ( ALL_OK );
        }
    }
    else return ( ERR_CAN_MSG_NO_MSG );
}

}; // namespace LCS
```

```
//-----------------------------------------------------------------------------------------------------------
//
// LCS Runtime library - Non volatile storage based on the M24LCxxx chip family
//
//-----------------------------------------------------------------------------------------------------------
// This file implements the LCS runtime library non-volatile memory. The hardware is the AA24xxx chip family,
// which offers an I2C protocol based chip with various capacities. They all share the same pin layout and
// command structure.
//
// In addition we also support the M24C04 chip, which is used on the extension boards as a configuration
// storage. This chip will however be replaced by 24AA32, a 4K chip of the same chip family as the other
// chips on the controller board.
//
//-----------------------------------------------------------------------------------------------------------
//
// LCS Core library - Non volatile storage based on the M24LCxxx chip family
// 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 (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 files.
//
//-----------------------------------------------------------------------------------------------------------
#include "LcsRuntimeLib.h"
#include "LcsRtLibInt.h"

//-----------------------------------------------------------------------------------------------------------
// Externals.
//
//-----------------------------------------------------------------------------------------------------------
namespace LCS {

    extern uint16_t debugMask;
};

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

using namespace LCS;

//-----------------------------------------------------------------------------------------------------------
// Definitions for the M24LCxxx chips page size and total size. The chips have a pageSize which is the unit
// updated in case of a write. A write cannot across a page boundary and must be split into several writes
// if necessary. Reads do not have this problem. Al chips have the same I2C address root which is "1010".
// There are three address lines A2, A1 and A0, which are used to select a chips. Up to eight chips can be
// addressed on a single I2C bus.
//
// The pageSizes on the chip are a multiple of 32bytes. For now, we use this size as the common denominator.
// Block handling and chipSize page handling are nicely taken care of this way. The downside is however that
// a write will update the chip page up to four times for a pageSize of 128. However, since the chips have
// more than a million write cycles and we rarely write large chunks of data, this will hopefully not be an
// issue in the near future.
//
// ??? the M24C04 is to be phased out ... we do not use that chip anymore...
//-----------------------------------------------------------------------------------------------------------
const uint16_t  BUFFER_BLOCK_SIZE           = 16; // ??? until we remove the M24C04 chip.

const uint16_t  M24LC32_PAGE_SIZE           = 32;
const uint32_t  M24LC32_MAX_SIZE            = 4096;

const uint16_t  M24LC64_PAGE_SIZE           = 32;
const uint32_t  M24LC64_MAX_SIZE            = 8192;

const uint16_t  M24LC128_PAGE_SIZE          = 64;
const uint32_t  M24LC128_MAX_SIZE           = 16384;

const uint16_t  M24LC256_PAGE_SIZE          = 64;
const uint32_t  M24LC256_MAX_SIZE           = 32768;

const uint16_t  M24LC512_PAGE_SIZE          = 128;
const uint32_t  M24LC512_MAX_SIZE           = 65536;

const uint16_t  M24C04_PAGE_SIZE            = 16;
const uint32_t  M24C04_MAX_SIZE             = 512;

const uint8_t   NVM_I2C_ADR_ROOT            = 0b1010000;
const uint8_t   EXT_I2C_ADR_ROOT            = 0b1010000;

const uint8_t   NVM_WRITE_DELAY             = 0x05;

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

```
// Runtime NVM sizes. The runtime map has a maximum of 8Kb. The maximum size of a NVM chip is 64Kb.  The
// maximum size for an extension board NVM chip is 4Kb.
//
//----------------------------------------------------------------------------------------------------------
const uint32_t  NVM_RUNTIME_MAP_SIZE        = 0x2000;
const uint32_t  NVM_MAX_NVM_SIZE            = 0x10000;
const uint32_t  NVM_MAX_EXT_SIZE            = 0x1000;

//----------------------------------------------------------------------------------------------------------
// Module global data. A LCS node board has two NVM channels. The "NVM" channel refers to the NVM chip on
// main controller board. The "EXT" channel is the I2C bus that reaches out the the extension boards. On
// each extension board there is again a small NVM chip with configuration data. Besides the hardware pins
// there are the sizes of the chips.
//
// There is no easy way to determine the size of the actual chip. By convention, the extension board NVM
// chip has a fixed 4 Kbytes. The NVM chip on the main controller board is at least 8Kbytes, which is the
// architected runtime data structures size. The maximum size is 64 Kbytes. All the chips are from a hardware
// perspective identical. When we start a node, the nodeMap structure, i.e. the first few hundred bytes,
// contains a field that holds the actual size configured for the chip on the particular board. The difference
// between the runtime map size and the particular NVM chip maximum is considered "user NVM space" which the
// firmware can use as needed.
//
//----------------------------------------------------------------------------------------------------------
uint32_t    nodeNvmSize                     = 0;
uint32_t    extNvmSize                      = 0;

uint8_t     nvmSclPin                       = CDC::UNDEFINED_PIN;
uint8_t     nvmSdaPin                       = CDC::UNDEFINED_PIN;

uint8_t     extSclPin                       = CDC::UNDEFINED_PIN;
uint8_t     extSdaPin                       = CDC::UNDEFINED_PIN;

//----------------------------------------------------------------------------------------------------------
// "testNvmChipMemorySize" will check the NVM chip for its size. Since the chip itself has no way of telling
// its memory capacity, we need to go a rather cumbersome way. For each possible size, read the last byte,
// store a new value there, read it again. If the values match, it is a valid memory location. Don't forget
// to restore the previous value. If we are not successful, try the next smaller size. Currently, the LCS
// hardware uses The chip family M24LCxxx with sizes of 4, 8, 16, 32 and 64Kbytes.
//
// ??? not tested yet ...
//----------------------------------------------------------------------------------------------------------
uint32_t testNvmChipMemorySize( uint8_t sclPin, uint8_t i2cAdr ) {

    uint32_t    nvmSize         = M24LC512_MAX_SIZE;
    uint32_t    testAdr         = nvmSize - 1;
    uint8_t     originalValue   = 0;
    uint8_t     testValue       = 0xab;
    uint8_t     tmpValue        = 0;
    uint8_t     tmpBuf[ 3 ]     = { 0 };
    uint8_t     rStat           = ALL_OK;

    while ( nvmSize >= M24LC32_MAX_SIZE ) {

        tmpBuf[ 0 ] = testAdr >> 8 & 0xFF;
        tmpBuf[ 1 ] = testAdr &0xFF;

        rStat = CDC::i2cWrite( sclPin, i2cAdr, tmpBuf, 2, true );
        if ( rStat == ALL_OK ) rStat = CDC::i2cRead( sclPin, i2cAdr, &originalValue, 1 );
        if ( rStat != ALL_OK ) return( ERR_NVM_CHIP_SIZE_DETECT );

        tmpBuf[ 2 ] = testValue;

        rStat = CDC::i2cWrite( sclPin, i2cAdr, tmpBuf, sizeof( tmpBuf ));
        if ( rStat == ALL_OK ) {

            CDC::sleepMillis( NVM_WRITE_DELAY );

            rStat = CDC::i2cWrite( sclPin, i2cAdr, tmpBuf, 2, true );
            if ( rStat == ALL_OK ) rStat = CDC::i2cRead( sclPin, i2cAdr, &tmpValue, 1 );
            if ( rStat == ALL_OK ) {

                if ( tmpValue == testValue ) {

                    rStat = CDC::i2cWrite( sclPin, i2cAdr, tmpBuf, sizeof( tmpBuf ));
                    CDC::sleepMillis( NVM_WRITE_DELAY );
                    return( nvmSize );
                }
                else {

                    nvmSize = nvmSize / 2;
                    testAdr = nvmSize - 1;
                }
            }
        }
        else return( ERR_NVM_CHIP_SIZE_DETECT );
    }

    return( nvmSize );
}

//----------------------------------------------------------------------------------------------------------
// Each NVM chip has certain size. This function will round the size to the next lower chip memory size.
// We expect however that the programmer used the correct size, so this is done just in case. The lowest
// value is the 4Kb chip.
//
//----------------------------------------------------------------------------------------------------------
uint32_t roundNvmMaxSize( uint16_t chipSize ) {

    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 ];

```

```
        rStat = CDC::i2cWrite( sclPin, i2cAdr, dataBuf, len + 2 );
    }

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

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

    return ( rStat );
}

//----------------------------------------------------------------------------------------------------------
// "nvmGetBytes" reads a set of data bytes from the memory. Although read operations do not have a page
// boundary issue, we stick to the concept to read within page boundaries as we may one day use more than
// chip to build NVMs and then we have no problems with crossing chip boundaries.
//
//----------------------------------------------------------------------------------------------------------
uint8_t nvmGetBytes( 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( "nvmGetBytes: scl: %d, i2c: 0x%x, ofs: 0x%x, bufAdr: %p, len: %d\n",
                sclPin, i2cAdr, ofs, (uint32_t) buf, len );
    }

    uint32_t nvmSize = (( sclPin == nvmSclPin ) ? nodeNvmSize : extNvmSize );
    if ( ofs + len > nvmSize ) return ( ERR_NVM_SIZE_EXCEEDED );

    uint32_t  bytesLeft     = len;
    uint32_t  pageBytesLeft = BUFFER_BLOCK_SIZE - ofs % BUFFER_BLOCK_SIZE;

    while ( bytesLeft > pageBytesLeft ) {

        rStat = nvmGetBytesFromPage( sclPin, i2cAdr, ofs + len - bytesLeft, buf + len - bytesLeft, pageBytesLeft );
        if ( rStat != ALL_OK ) break;

        bytesLeft       -= pageBytesLeft;
        pageBytesLeft   = BUFFER_BLOCK_SIZE;
    }

    if (( rStat == ALL_OK ) && ( bytesLeft > 0 )) {

        rStat = nvmGetBytesFromPage( sclPin, i2cAdr, ofs + len - bytesLeft, buf + len - bytesLeft, bytesLeft );
    }

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

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

    return( rStat );
}

//----------------------------------------------------------------------------------------------------------
// "nvmPutBytes" transmits a set of data bytes to the memory. We cannot write across the internal NVM page
// boundary and also across a chip boundary. This routine will split the data to write only within one page
// in a given write cycle.
//
// There is a quirk with figuring out that a chip is ready for the next write instruction. The data sheet
// suggest a writing of one byte to see of the chip acknowledges. If not it is still in a write operation.
// This approach does not seem to work with the PICO i2c libraries. So, we will go the "slow" way of giving
// the chip the time to complete the write cycle before issuing another one. Since we do not often write
// to the NVM, the slow mode is perhaps acceptable for now.
//
//----------------------------------------------------------------------------------------------------------
uint8_t nvmPutBytes( 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( "nvmPutBytes: scl: %d, i2c: 0x%x, ofs: 0x%x, buf: %p, len: %d\n", sclPin, i2cAdr, ofs, buf, len );
     }

    uint32_t nvmSize = (( sclPin == nvmSclPin ) ? nodeNvmSize : extNvmSize );
    if ( ofs + len > nvmSize ) return ( ERR_NVM_SIZE_EXCEEDED );

    uint32_t  bytesLeft     = len;
    uint32_t  pageBytesLeft = BUFFER_BLOCK_SIZE - ofs % BUFFER_BLOCK_SIZE;

    while ( bytesLeft > pageBytesLeft ) {

        rStat = nvmPutBytesInPage( sclPin, i2cAdr, ofs + len - bytesLeft, buf + len - bytesLeft, pageBytesLeft );
        if ( rStat != ALL_OK ) break;

        bytesLeft       -= pageBytesLeft;
        pageBytesLeft   = BUFFER_BLOCK_SIZE;

        CDC::sleepMillis( NVM_WRITE_DELAY );
    }

    if (( rStat == ALL_OK ) && ( bytesLeft > 0 )) {

       rStat = nvmPutBytesInPage( sclPin, i2cAdr, ofs + len - bytesLeft, buf + len - bytesLeft, bytesLeft );
       CDC::sleepMillis( NVM_WRITE_DELAY );
    }

```

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

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

    return( rStat );
}

//------------------------------------------------------------------------------------------------------
// "nvmClearArea" wipes out an area of the NVM chip. To speed up the writing, we fill a local buffer with
// the value and then write blocks at a time.
//
//------------------------------------------------------------------------------------------------------
uint8_t nvmClearArea( uint8_t sclPin, uint8_t i2cAdr, uint32_t ofs, uint32_t len, uint8_t val ) {

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

        printf( "nvmClearArea: scl: %d, i2c: 0x%x, ofs: 0x%x, len: %d, val: %d\n", sclPin, i2cAdr, ofs, len, val );
    }

    uint8_t     tmpBuf[ BUFFER_BLOCK_SIZE ];
    uint8_t     rStat   = ALL_OK;
    uint32_t    nvmSize = (( sclPin == nvmSclPin ) ? nodeNvmSize : extNvmSize );
    uint32_t    limit   = ofs + len;

    if ( ofs + len > nvmSize ) return ( ERR_NVM_SIZE_EXCEEDED );

    for ( int i = 0; i < BUFFER_BLOCK_SIZE; i ++ ) tmpBuf[ i ] = val;

    while ( len > BUFFER_BLOCK_SIZE ) {

        rStat = nvmPutBytes( sclPin, i2cAdr, ofs, tmpBuf, sizeof( tmpBuf ));
        if ( rStat != ALL_OK ) break;

        ofs += BUFFER_BLOCK_SIZE;
        len -= BUFFER_BLOCK_SIZE;
    }

    if (( rStat == ALL_OK ) && ( len > 0 )) {

        rStat = nvmPutBytes( sclPin, i2cAdr, ofs, tmpBuf, len );
    }

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

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

    return( rStat );
}

}; // namespace


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

//------------------------------------------------------------------------------------------------------
// "configNvm" will setup the module local variables. We copy the I2C hardware pins and the NVM related data
// from the CDC descriptors. The CDC descriptor also contains the configured sizes for the NVM chips.
//
//------------------------------------------------------------------------------------------------------
uint8_t configNvm( CDC::CdcConfigDesc *ci ) {

    nvmSclPin     = ci -> NVM_I2C_SCL_PIN;
    nvmSdaPin     = ci -> NVM_I2C_SDA_PIN;
    extSclPin     = ci -> EXT_I2C_SCL_PIN;
    extSdaPin     = ci -> EXT_I2C_SDA_PIN;
    nodeNvmSize   = ci -> NODE_NVM_SIZE;
    extNvmSize    = ci -> EXT_NVM_SIZE;

    if ( nodeNvmSize > NVM_MAX_NVM_SIZE )   nodeNvmSize = NVM_MAX_NVM_SIZE;
    if ( extNvmSize > NVM_MAX_EXT_SIZE )    extNvmSize  = NVM_MAX_EXT_SIZE;

    return( ALL_OK );
}

//------------------------------------------------------------------------------------------------------
// Controller Board Runtime Map access routines. The runtime map occupies the first 8 Kbytes of the main
// controller NVM chip. There are routines for getting and setting a word as well as routines to read and
// write a buffer. All access routines are prefixed with "rt".
//
//------------------------------------------------------------------------------------------------------
uint8_t rtNvmPutWord( uint32_t ofs, uint16_t word ) {

    return( nvmPutBytes( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, (uint8_t *) &word, sizeof( uint16_t )));
}

uint8_t rtNvmGetWord( uint32_t ofs, uint16_t *word ) {

    return( nvmGetBytes( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, (uint8_t *) word, sizeof( uint16_t )));
}

uint8_t rtNvmPutBytes( uint32_t ofs, uint8_t *buf, uint32_t len ) {

    return( nvmPutBytes( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, buf, len ));
```

```
}

uint8_t rtNvmGetBytes( uint32_t ofs, uint8_t *buf, uint32_t len ) {

    return( nvmGetBytes( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, buf, len ));
}

uint8_t rtNvmClearArea( uint32_t ofs, uint32_t len, uint8_t val ) {

    return( nvmClearArea( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, len, val ));
}

uint32_t rtNvmGetSize( ) {

    return( NVM_RUNTIME_MAP_SIZE );
}

//----------------------------------------------------------------------------------------------------
// Extension Board Map access routines. These routines access the NVM on the extension board. The I2C address
// is formed by the chip common I2C address plus the address bits of the chip to select the chip on the
// particular extension board. Similar to the runtime NVM access routines, there are routines for getting
// and setting a word as well as routines to read and  write a buffer. All access routines are prefixed with
// "ext".
//
//----------------------------------------------------------------------------------------------------
uint8_t extNvmPutWord( uint8_t boardId, uint32_t ofs, uint16_t word ) {

    uint8_t i2cAdr = EXT_I2C_ADR_ROOT + (( boardId % MAX_EXT_BOARD_MAP_ENTRIES ) << 1 );
    return( nvmPutBytes( extSclPin, i2cAdr, ofs, (uint8_t *) &word, sizeof( uint16_t )));
}

uint8_t extNvmGetWord( uint8_t boardId, uint32_t ofs, uint16_t *word ) {

    uint8_t i2cAdr = EXT_I2C_ADR_ROOT + (( boardId % MAX_EXT_BOARD_MAP_ENTRIES ) << 1 );
    return( nvmGetBytes( extSclPin, i2cAdr, ofs, (uint8_t *) word, sizeof( uint16_t )));
}

uint8_t extNvmPutBytes( uint8_t boardId, uint32_t ofs, uint8_t *buf, uint32_t len ) {

    uint8_t i2cAdr = EXT_I2C_ADR_ROOT + (( boardId % MAX_EXT_BOARD_MAP_ENTRIES ) << 1 );
    return( nvmPutBytes( extSclPin, i2cAdr, ofs, buf, len ));
}

uint8_t extNvmGetBytes( uint8_t boardId, uint32_t ofs, uint8_t *buf, uint32_t len ) {

    uint8_t i2cAdr = EXT_I2C_ADR_ROOT + (( boardId % MAX_EXT_BOARD_MAP_ENTRIES ) << 1 );
    return( nvmGetBytes( extSclPin, i2cAdr, ofs, buf, len ));
}

uint8_t extNvmClearArea( uint8_t boardId, uint32_t ofs, uint32_t len, uint8_t val ) {

    uint8_t i2cAdr = EXT_I2C_ADR_ROOT + (( boardId % MAX_EXT_BOARD_MAP_ENTRIES ) << 1 );
    return( nvmClearArea( extSclPin, i2cAdr, ofs, len, val ));
}

uint32_t extNvmGetSize( ) {

    return( extNvmSize );
}

//----------------------------------------------------------------------------------------------------
// Controller Board User Map access routines. The area between the main controller NVM chip runtime area and
// the chips hardware maximum size is the memory area available for the firmware programmer. Again, there
// are routines for getting and setting a word as well as routines to read and  write a buffer. All access
// routines are prefixed with "nvm".
//
//----------------------------------------------------------------------------------------------------
uint8_t usrNvmPutWord( uint32_t ofs, uint16_t word ) {

    ofs = ofs + NVM_USER_MAP_START;
    return( nvmPutBytes( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, (uint8_t *) &word, sizeof( uint16_t )));
}

uint8_t usrNvmGetWord( uint32_t ofs, uint16_t *word ) {

    ofs = ofs + NVM_USER_MAP_START;
    return( nvmGetBytes( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, (uint8_t *) word, sizeof( uint16_t )));
}

uint8_t usrNvmPutBytes( uint32_t ofs, uint8_t *buf, uint32_t len ) {

    ofs = ofs + NVM_USER_MAP_START;
    return( nvmPutBytes( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, buf, len ));
}

uint8_t usrNvmGetBytes( uint32_t ofs, uint8_t *buf, uint32_t len ) {

    ofs = ofs + NVM_USER_MAP_START;
    return( nvmGetBytes( nvmSclPin, NVM_I2C_ADR_ROOT + 0, ofs, buf, len ));
}

uint32_t usrNvmGetSize( ) {

    return ( nodeNvmSize - NVM_USER_MAP_START );
}

}; // 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";

```

```
uint16_t roundup( uint16_t elements, uint16_t alignSize ) {

    return ((( elements + alignSize - 1 ) / alignSize ) * alignSize );
}

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 );
}

//----------------------------------------------------------------------------------------------------
// "buildDefaultNodeMap" build a nodeMap with the default values from the declaration structure. The default
// nodeMap is used for initializing the memory runtime node map for formatting a new or corrupted runtime NVM.
//
//----------------------------------------------------------------------------------------------------
void buildDefaultNodeMap( LcsNodeMap *nMap ) {

    LcsNodeMap tmp;

    snprintf( tmp.name, MAX_NODE_NAME_SIZE, "Node" );
    tmp.nodeUID = CDC::createUid( );

    *nMap = tmp;
}

//----------------------------------------------------------------------------------------------------
// "buildDefaultPortMap" will initialize the portMap data structure on MEM or NVM, It is just an array of
// portMap entries. Each port will get a default name.
//
//----------------------------------------------------------------------------------------------------
void buildDefaultPortMap( LcsPortMap *pMap ) {

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

        LcsPortMapEntry pEntry;

        snprintf( pEntry.name, MAX_PORT_NAME_SIZE, "Port-%d", i + 1 );
        pMap -> map[ i ] = pEntry;
    }
}

//----------------------------------------------------------------------------------------------------
// "buildDefaultEventMap" initializes the event map on MEM or NVM. It is just an array of eventMap entries.
//
//----------------------------------------------------------------------------------------------------
void buildDefaultEventMap( LcsEventMap *eMap ) {

    LcsEventMapEntry e;
    for ( uint16_t i = 0; i < MAX_EVENT_MAP_ENTRIES; i++ ) eMap -> map[ i ] = e;
}

//----------------------------------------------------------------------------------------------------
// "buildDefaultNodeData" initializes the node data map on MEM or NVM. It is just an array of variables. We
// clear them out.
//
//----------------------------------------------------------------------------------------------------
void buildDefaultNodeData( LcsNodeData *nData ) {

    memset( nData -> map, 0, MAX_NODE_DATA_BLOCKS * MAX_ATTR_MAP_ENTRIES * sizeof( uint16_t));
}

//----------------------------------------------------------------------------------------------------
// "buildNvmRuntimeStructures" initializes a new or corrupt runtime NVM with default data. After successful
// completion, we will have a valid runtime map.
//
// ??? CDC map business ???
//----------------------------------------------------------------------------------------------------
uint8_t buildNvmRuntimeStructures( ) {

    uint8_t rStat;

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

    rtNvmClearArea( NVM_NODE_MAP_START, NVM_RUNTIME_AREA_SIZE );

    // ??? build default CDC map ?

    buildDefaultNodeMap( &nodeMap );
    buildDefaultPortMap( &portMap );
    buildDefaultEventMap( &eventMap );
    buildDefaultNodeData( &nodeData );

    rStat = rtNvmPutBytes( NVM_NODE_MAP_START, (uint8_t *) &nodeMap, sizeof( LcsNodeMap ));
    if ( rStat == ALL_OK ) rtNvmPutBytes( NVM_PORT_MAP_START, (uint8_t *) &portMap, sizeof( LcsPortMap ));
```

```
    if ( rStat == ALL_OK ) rtNvmPutBytes( NVM_EVENT_MAP_START, (uint8_t *) &eventMap, sizeof( LcsEventMap ));
    if ( rStat == ALL_OK ) rtNvmPutBytes( NVM_NODE_DATA_START, (uint8_t *) &nodeData, sizeof( LcsNodeData ));

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

    return( rStat );
}

//----------------------------------------------------------------------------------------------------------
// "buildDefaultBoardDesc" initializes the extension board structure. An extension board has a header
// structure identical to the nodeMap structure. In addition, there is the driver data area.
//
//----------------------------------------------------------------------------------------------------------
void buildDefaultBoardDesc( LcsDrvBoardDesc *bDesc ) {

    LcsNvmHeader tmp;

    bDesc -> head = tmp;
    memset( bDesc -> driverData, 0, MAX_DRV_DATA_SIZE * sizeof( uint16_t ));
}

//----------------------------------------------------------------------------------------------------------
// "buildDefaultDrvMap" initializes the driver map memory structure.
//
//----------------------------------------------------------------------------------------------------------
void buildDefaultDrvMap( LcsDrvMap *drv ) {

    LcsDrvBoardDesc initDesc;

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

        drvMap.map[ i ].flags       = 0;
        drvMap.map[ i ].drvFunc     = nullptr;
        drvMap.map[ i ].extBoard    = initDesc;
    }
}

//----------------------------------------------------------------------------------------------------------
// "buildNvmRuntimeStructures" initializes a new or corrupt runtime NVM with default data. However, the write
// will only succeed when we have the board write enabled. Otherwise an error is returned.
//
//----------------------------------------------------------------------------------------------------------
uint8_t buildNvmExtBoardStructure( uint8_t boardId ) {

    uint8_t     rStat;
    LcsDrvEntry entry;

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) {

        printf( "buildNvmExtBoardStructure for board: %d\n", boardId );
    }

    rStat = extNvmClearArea( boardId, 0, sizeof( LcsDrvEntry ));
    if ( rStat == ALL_OK ) extNvmPutBytes( boardId, 0, (uint8_t *) &entry, sizeof( entry ));

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) {

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

    return( rStat ); // ??? own error constant ?
}

}; // namespace


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

//----------------------------------------------------------------------------------------------------------
// When the node is powered on, the very first thing to do is to setup the CDC library and setup the "active"
// and "ready" LED pins used by the board. The pins need to to be configured. We also make a call to to
// initialize the CDC. Note that this may have been done before, when for example the firmware programmer
// wants to use the HW before calling any library setup code. It is no problem to call the CDC init routine
// several times.
//
// There are two basic modes. The first is when we have a console connected. We will prompt and wait for a
// start command. There are several options for starting a node. The easiest is "R" which just starts the
// node. The "D" command will start with debugging enabled. We will set the setup debug flags to check any
// issues during the startup phase. Finally, there is there "F" command, which will format the NVM runtime
// area. However, all that is happening in this routine is to set these options to be executed at the right
// place in the setup sequence.
//
// The second mode is when there no console connected. In this case, Debug is disabled and we just setup
// the node. This mode should be the normal case for all the nodes in a layout.
//
// Perhaps one day, this routine could be enhanced to allow commands to pile up the start options followed
// by the final start command to get the show going. especially the debug mask would be a candidate.
//
//----------------------------------------------------------------------------------------------------------
uint8_t initCdcLayer( CDC::CdcConfigDesc *ci ) {

    const uint32_t CONSOLE_TIMEOUT = 1024 * 1024 * 4;

    cdcMap.cfg = *ci;
```

```

    CDC::init( ci );

    if ( ci -> READY_LED_PIN != CDC::UNDEFINED_PIN ) CDC::configureDio( ci -> READY_LED_PIN, CDC::OUT );
    if ( ci -> ACTIVE_LED_PIN != CDC::UNDEFINED_PIN ) CDC::configureDio( ci -> ACTIVE_LED_PIN, CDC::OUT );

    if ( CDC::isConsoleConnected( )) {

        CDC::writeDio( ci -> READY_LED_PIN, true );

        while ( true ) {

            printf( "=>" );

            char ch = CDC::getConsoleChar( CONSOLE_TIMEOUT );

            if (( ch == 'R' ) || ( ch == 'r' )) {

                printf( "Starting - normal mode\n" );

                debugMask       &= ~ DBG_CONFIG;
                startOptions    = NOPT_NIL;
                return( ALL_OK );
            }
            else if (( ch == 'D' ) || ( ch == 'd' )) {

                 printf( "Starting - debug mode\n" );

                debugMask       = DBG_CONFIG | DBG_SETUP | DBG_EVENTS;
                startOptions    = NOPT_NIL;
                return( ALL_OK );
            }
            else if (( ch == 'F' ) || ( ch == 'f' )) {

                printf( "Starting - format mode\n" );

                debugMask       &= ~ DBG_CONFIG;

                debugMask       = DBG_CONFIG | DBG_SETUP | DBG_NVM_ACCESS;

                startOptions    = NOPT_FORMAT_RUNTIME;
                return( ALL_OK );
            }
            else if ( ch == '?' ) {

                printf( "Setup options:\n" );
                printf( "r, R -> start the node with debug initially disabled\n" );
                printf( "d, D -> start the node with \"setup\" debug options enabled\n" );
                printf( "f, F -> start the node with a newly formatted runtime map\n" );
            }
            else printf( "\n" );
        }
    }
    else {

        debugMask       &= ~ DBG_CONFIG;
        startOptions    = NOPT_NIL;
        return ( ALL_OK );
    }
}

//---------------------------------------------------------------------------------------------------------------
// The NVM library functions will work after this routine. We first set up the I2C channels, which are the
// heart of any internal board communication. After the I2C channels are initialized, we will configure the
// NVM library. If all is OK, we can talk to all NVMs on the boards making up the node.
//
// ??? should we assume a "architectural" setting of the IC2 channels and not rely on CDC map ?
//---------------------------------------------------------------------------------------------------------------
uint8_t initNvmChannels( CDC::CdcConfigDesc *ci ) {

   if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) {

        printf( "initNvmChannels: nvmSCL: %d, nvmSDA: %d, extSCL: %d, extSDA: %d\n",
                ci -> NVM_I2C_SCL_PIN, ci -> NVM_I2C_SDA_PIN, ci -> EXT_I2C_SCL_PIN, ci -> EXT_I2C_SDA_PIN );
    }

    uint8_t rStat;

    if (( ci -> NVM_I2C_SCL_PIN != CDC::UNDEFINED_PIN ) && ( ci -> NVM_I2C_SDA_PIN != CDC::UNDEFINED_PIN )) {

        rStat = CDC::configureI2C( ci -> NVM_I2C_SCL_PIN , ci -> NVM_I2C_SDA_PIN );
        if ( rStat != ALL_OK ) return( rStat );
    }

    if (( ci -> EXT_I2C_SCL_PIN != CDC::UNDEFINED_PIN ) && ( ci -> EXT_I2C_SDA_PIN != CDC::UNDEFINED_PIN )) {

        rStat = CDC::configureI2C( ci -> EXT_I2C_SCL_PIN , ci -> EXT_I2C_SDA_PIN );
        if ( rStat != ALL_OK ) return( rStat );
    }

    rStat = configNvm( ci );

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

    return ( rStat );
}

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

```
// Next is CAN bus setup. The message bus is the central communication mechanism. If we can also get it up
// early we could use it not only for configurations and operations but perhaps for remote troubleshooting.
//
// ??? should we assume a "architectural" setting of the CAN channel and not rely on CDC map ?
//----------------------------------------------------------------------------------------------------------
uint8_t initCanBus( CDC::CdcConfigDesc *ci ) {

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

    msgBus = new LcsMsgBusCAN( );

    uint8_t rStat = msgBus -> init( 0, ci -> CAN_BUS_RX_PIN, ci -> CAN_BUS_TX_PIN, ci -> CAN_BUS_CTRL_MODE );
    if ( rStat != ALL_OK ) {

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

        rStat = ERR_CAN_SETUP;
    }

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

    return ( rStat );
}

//----------------------------------------------------------------------------------------------------------
// "setupNodeMap" sets up the nodeMap. It is the first routine after all the basic hardware settings is in
// place. Unless the start options tell us to just format a new runtime area, we read in the nodeMap from
// the node NVM. A quick check of the magic words and the the nodeMap size field will tell us whether this
// nodeMap was initialized before. If this is not the case, we must assume a corrupt nodeMap or a new board.
// The runtime area data structures are created with default values and written to the NVM.
//
// In any case, the follow-on setup routines can assume a valid data structure to work from and just read
// the NVM as normal. If this routine has an error it should be considered as a fatal error.
//
// ??? check the sizes of the maps, load CDC map if we only have a board type ?
//----------------------------------------------------------------------------------------------------------
uint8_t setupNodeMap( LcsConfigDesc *cfg ) {

    uint8_t rStat = ALL_OK;

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

    if ( startOptions & NOPT_FORMAT_RUNTIME ) {

        rStat = buildNvmRuntimeStructures( );
    }

    rStat = rtNvmGetBytes( NVM_NODE_MAP_START, (uint8_t *) &nodeMap, sizeof( LcsNodeMap ));
    if ( rStat != ALL_OK ) return( rStat );

     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) {

        uint16_t *ptr = (uint16_t *) &nodeMap;

        printf( "NodeMap Head: " );
        for ( int i = 0; i < 16; i++ ) printf( "0x%x ", ptr[ i ] );
        printf( "\n" );
     }

    if (( nodeMap.head.magicWord1 != NVM_MWORD_1 ) ||
        ( nodeMap.head.magicWord2 != NVM_MWORD_2 ) ||
        ( nodeMap.nodeMapSize != sizeof( LcsNodeMap ))) {

            // ??? check the sizes of the maps ???

        if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
            printf( "setupNodeMap: invalid header, re-format\n" );

        rStat = buildNvmRuntimeStructures( );
    }

    nodeMap.nodeOptions = cfg -> options;

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

    return ( rStat );
}

//----------------------------------------------------------------------------------------------------------
// "setupPortMap" will read the port data the NVM port map data area into the memory counterpart.
//
//----------------------------------------------------------------------------------------------------------
uint8_t setupPortMap( ) {

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

    uint8_t rStat = rtNvmGetBytes( NVM_PORT_MAP_START, (uint8_t *) &portMap, sizeof( LcsPortMap ));

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

    return ( rStat );
}

//----------------------------------------------------------------------------------------------------------
// "setupNodeDataMap" will read the node data blocks.
```

```
//
//-----------------------------------------------------------------------------------------------------
uint8_t setupNodeDataMap( ) {

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

    uint8_t rStat = rtNvmGetBytes( NVM_NODE_DATA_START, (uint8_t *) &nodeData.map, sizeof( nodeData.map ));

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

    return ( rStat );
}

//-----------------------------------------------------------------------------------------------------
// "setupCdcMap" will read the CDC descriptor from the NVM.
//
// ??? we should read it from the NVM, which implies that we need a way to handle vanilla boards...
// ??? we should also have a file with CDC descriptors for all boards we currently have...
// ??? if we detect a board with the vanilla board type, we can set the correct board type and the
// reboot...
//-----------------------------------------------------------------------------------------------------
uint8_t setupCdcMap( CDC::CdcConfigDesc *cdcConfig ) {

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

    uint8_t rStat = ALL_OK;


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

    return ( rStat );
}

//-----------------------------------------------------------------------------------------------------
// The event map stores all event/port pairs this node is interested to process. The map is a sorted map and
// there is a high water mark, so that we only read up to the last used entry in the map. Just like other
// data structures we could just read in all entries. However, this is a large map. It would be better to
// just read up to the HWM, if the HWM is valid. If this is not the case, we have to assume that there are
// bigger issues with the event map. In this case we will read the entire map entry by entry, add used
// entries, i.e. entries with a non-NIL event ID to the memory map. After reading all entries, the newly
// created event map is written back to the NVM place.
//
//-----------------------------------------------------------------------------------------------------
uint8_t setupEventMap( ) {

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) {

        printf( "setupEventMap, entries: %d, HWM: %d\n", nodeMap.eventMapEntries, nodeMap.eventMapHwm );
    }

    uint8_t rStat = ALL_OK;

    if ( nodeMap.eventMapHwm <= MAX_EVENT_MAP_ENTRIES ) {

        for ( uint16_t i = 0; i < nodeMap.eventMapHwm; i++ ) {

            rStat = rtNvmGetBytes(  NVM_EVENT_MAP_START + i * sizeof( LcsEventMapEntry),
                                    (uint8_t *) &eventMap.map[ i ],
                                    sizeof( LcsEventMapEntry ));
        }
    }
    else {

        LcsEventMapEntry e;
        for ( uint16_t i = 0; i < nodeMap.eventMapEntries; i++ ) eventMap.map[ i ] = e;

        nodeMap.eventMapHwm = 0;
        for ( uint16_t i = 0; i < nodeMap.eventMapEntries; i++ ) {

            LcsEventMapEntry eventEntry;

            rStat = rtNvmGetBytes(  NVM_EVENT_MAP_START + i * sizeof(LcsEventMapEntry),
                                    (uint8_t *) &eventEntry,
                                    sizeof(LcsEventMapEntry));

            if (( rStat == ALL_OK ) && ( eventEntry.eventId != NIL_EVENT_ID )) {

                addEvent( eventEntry.eventId, eventEntry.portId );
            }
        }

        rStat = syncEventMap( );
    }

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

    return ( rStat );
}

//-----------------------------------------------------------------------------------------------------
// The user map is the additional NVM storage that the chip set offers beyond the area allocated for the
// system. Since we have no idea what the user is doing, we do nothing for now. It is just a placeholder.
//
//-----------------------------------------------------------------------------------------------------
uint8_t setupUserMap( ) {

```

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

    uint8_t rStat = ALL_OK;

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

    return ( ALL_OK );
}

//----------------------------------------------------------------------------------------------------------
// "setupCallbackMap" initializes the callback map. We expect the user to register their callbacks between
// the runtime init and runtime start routine.
//
//----------------------------------------------------------------------------------------------------------
uint8_t setupCallbackMap( ) {

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

    uint8_t rStat = ALL_OK;

    callbackMap.lcsMsgCallback      = nullptr;
    callbackMap.dccMsgCallback      = nullptr;
    callbackMap.cmdLineCallback     = nullptr;

    callbackMap.initCallback        = nullptr;
    callbackMap.resetCallback       = nullptr;
    callbackMap.pfailCallback       = nullptr;

    callbackMap.eventCallback       = nullptr;
    callbackMap.reqCallback         = nullptr;
    callbackMap.repCallback         = nullptr;

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

    return( rStat );
}

//----------------------------------------------------------------------------------------------------------
// "setupTaskMap" initializes the task map. A user can register routines that are executed on a periodic
// basis.
//
//----------------------------------------------------------------------------------------------------------
uint8_t setupTaskMap( ) {

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

    uint8_t rStat = ALL_OK;

    nodeMap.taskMapEntries  = MAX_TASK_MAP_ENTRIES;
    nodeMap.taskMapHwm      = 0;

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

        taskMap.map[ i ].task       = nullptr;
        taskMap.map[ i ].interval   = 0;
        taskMap.map[ i ].timeStamp  = 0;
    }

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

    return( rStat );
}

//----------------------------------------------------------------------------------------------------------
// "setupPendingReqMap" initializes the pending request map. Currently, we do not use a HWM approach, but
// just use all entries when searching the map.
//
//----------------------------------------------------------------------------------------------------------
uint8_t setupPendingReqMap( ) {

    uint8_t rStat = ALL_OK;

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

    nodeMap.pendingMapEntries   = MAX_PENDING_REQ_MAP_ENTRIES;
    nodeMap.pendingMapHwm       = MAX_PENDING_REQ_MAP_ENTRIES;

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

    return( rStat );
}

//----------------------------------------------------------------------------------------------------------
// "setupDrvFuncMap" initializes the driver function label map. This table is used when we need to find the
// driver for an extension board type.
//
// ??? we could pre register all known drivers... a user could still register a new one and also overwrite
// a pre-registered driver with a new func label.
//----------------------------------------------------------------------------------------------------------
uint8_t setupDrvFuncMap( ) {

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

    uint8_t rStat = ALL_OK;

```

```
    nodeMap.drvFuncMapEntries = MAX_DRV_TYPES;
    nodeMap.drvFuncMapHwm     = MAX_DRV_TYPES;

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

    return( rStat );
}

//----------------------------------------------------------------------------------------------------------
// "setupDrvMap" initializes the driver map. For each possible extension board, up to four, there is an
// entry in this map.
//
//----------------------------------------------------------------------------------------------------------
uint8_t setupDrvMap( ) {

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

    uint8_t rStat = ALL_OK;

    nodeMap.pendingMapEntries   = MAX_PENDING_REQ_MAP_ENTRIES;
    nodeMap.pendingMapHwm       = MAX_PENDING_REQ_MAP_ENTRIES;

    buildDefaultDrvMap( &drvMap );

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

    return( rStat );
}

//----------------------------------------------------------------------------------------------------------
// The runtime library will one day perhaps a set of internal functions to execute periodically. Right now,
// this routine will do nothing.
//
//----------------------------------------------------------------------------------------------------------
uint8_t registerInternalTasks( ) {

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

    uint8_t rStat = ALL_OK;


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

    return( rStat );
}

//----------------------------------------------------------------------------------------------------------
// With the node properly initialized, it is time to see whether we have connected extension boards. An
// extension board has also a small NVM on the board that will tell us what the board type is and keep driver
// configuration data. There can be up to four boards, numbered from 0 to 3. The runtime has a driver map
// with four extension descriptor entries. After initializing the driver map, we attempt to read from each
// extension NVM on an extension board. If the read fails, there is no board at that location, which will
// mark in the flags field. The drvMap HWM will tell us how many board we actually found. Note that the
// boards are by hardware always connected in the order 0,1,2 and 3. If for example only two boards are
// connected, the HWM would be 2.
//
//----------------------------------------------------------------------------------------------------------
uint8_t detectExtensionBoards( ) {

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

    uint8_t rStat = ALL_OK;

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

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

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

        rStat = extNvmGetBytes( i, 0, (uint8_t *) &drvEntry -> extBoard, sizeof( LcsDrvBoardDesc ));
        if ( rStat == ALL_OK ) {

            uint16_t *ptr = (uint16_t *) &drvEntry -> extBoard;

            if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) {

                printf( "Extension Board Desc Head: " );
                for ( int j = 0; j < 16; j++ ) printf( "0x%x ", ptr[ j ] );
                printf( "\n" );
            }

            nodeMap.nodeFlags |= NFLAGS_EXT_PRESENT;
            nodeMap.drvMapHwm ++;

            drvEntry -> flags |= BF_EXT_BOARD_PRESENT;

            if (( drvEntry -> extBoard.head.magicWord1 == NVM_MWORD_1 ) &&
                ( drvEntry -> extBoard.head.magicWord2 == NVM_MWORD_2 )) {

                drvEntry -> flags |= BF_EXT_BOARD_VALID;

                if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
                    printf( "detectExtensionBoard, boardId: %d -> valid\n", i );
            }
            else  {
```

```
                if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
                    printf( "detectExtensionBoard, boardId: %d -> inValid\n", i );
            }
        }
        else {

            if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP ))
                printf( "detectExtensionBoard, boardId: %d, rStat: %d\n", i, rStat );
        }
    }

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

    return ( ALL_OK );
}

//-----------------------------------------------------------------------------------------------------------
// "lookupDrvFunc" searches the driver function label map. It is called when we setup the extension board.
// When the type is unknown, a nullptr is returned.
//
//-----------------------------------------------------------------------------------------------------------
LcsDrvReqFunc lookupDrvFunc( uint16_t drvType ) {

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

        if ( drvFuncMap.map[ i ].drvType == drvType ) return( drvFuncMap.map[ i ].drvFunc );
    }

    return( nullptr );
}

//-----------------------------------------------------------------------------------------------------------
// For all detected extension boards, we will first check that the board descriptor at slot "n" is there and
// that the board descriptor is reasonable. If so, the board type is used to load the respective driver.
// Note that during normal operations we cannot manipulate the NVM, as it is read protected. The jumper on
// the extension board needs to be removed for this. When removed, the extension board NVM can be written to
// with commands from the runtime.
//
// When the driver function is not pre registered, we do not have a function to set in the driver map.
// In this case the driver is not usable yet. The problem is that we can only make driver registration
// calls after LcsInitRuntime. But the all driver boards have been detected. There are two ways. For driver
// know already, we "pre-register" them. For any other driver, the driver function registration routine will
// check the driver table for the driver type and the patch the function label. All this should takes place
// before the final call to "startRuntime".
//
//-----------------------------------------------------------------------------------------------------------
uint8_t setupExtensionBoards( ) {

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

    uint8_t rStat = ALL_OK;

    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 )) {

            LcsDrvReqFunc drvFunc = lookupDrvFunc( drvEntry -> extBoard.head.boardType );

            if ( drvFunc != nullptr ) {

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

                if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) {

                    printf( "Driver setup, type: %d, stat: %d\n",
                        drvEntry -> extBoard.head.boardType, drvEntry -> lastErr );
                }
            }
            else {

                if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) {

                    printf( "Driver setup, type not found: %d\n", drvEntry -> extBoard.head.boardType );
                }
            }
        }
    }

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

    return ( rStat );
}

//-----------------------------------------------------------------------------------------------------------
// Driver function registration. There is a simple table which maintains extension boards types and the
// driver function form them. If the type is already registered, we just overwrite the function signature.
// Otherwise we find a free entry and use it.
//
// The driver function registration can only be called after the initialization of the LCS runtime. By then
// the extension boards are however already detected, and if there are no drivers pre-registered no driver
// function was registered. The extension board is therefore not ready and was also not reseted. Consequently,
// 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.
```

```
//
// ??? read NVM to MEM.
// ??? would a node reset clear any outstanding requests ? or do we need to inform potential waiters ?
// ??? would it just drop all periodic tasks ? who registers them again ?
// ??? how do we make sure we only cover ports that are used ?
// ??? or would we need to be a bit more sensible what reset mean ?
//----------------------------------------------------------------------------------------------------------
uint8_t resetNode( uint16_t npId ) {

    uint8_t rStat = ALL_OK;

    if ( callbackMap.resetCallback != nullptr ) {

         // ??? load MEM from NVM....

        if ( callbackMap.resetCallback != nullptr ) {

            rStat = callbackMap.resetCallback( buildNpId( nodeMap.nodeId, 0  ));
        }


    }

    if ( rStat == ALL_OK ) {

        for ( uint8_t i = 1; i <= MAX_PORT_MAP_ENTRIES; i++ ) {

            // ??? load MEM from NVM....

            if ( callbackMap.resetCallback != nullptr ) {

                rStat = callbackMap.resetCallback( buildNpId( nodeMap.nodeId, i ));
            }
        }
    }

    if ( rStat == ALL_OK ) {

        // ??? reset the drivers...
    }

    return ( rStat );
}

//----------------------------------------------------------------------------------------------------------
// The LCS library has a set of configuration parameters. Currently this is only the "options" field. In the
// future there may be more fields. This routines returns a config structure with reasonable defaults set.
//
//----------------------------------------------------------------------------------------------------------
LcsConfigDesc getConfigDefault( ) {

    LcsConfigDesc cfg;

    cfg.options = 0;

    return( cfg );
}

//----------------------------------------------------------------------------------------------------------
// "initRuntime" is the routine that takes a controller board and initializes the whole show. It is the very
// first thing to call in a node firmware program. There is a lot to do. First, the CDC layer is initialized.
// NVM and CanBus follow. An error in this stage will result in a fatal error, we are not able to set up a
// valid runtime.
//
// If the HW setup worked, we are ready to read in the nodeMap. A nodeMap can be valid or not. It is defined
// as a map with valid "magic" words and reasonable values for the other fields. In case of an invalid
// nodeMap, a new default map is created and written back to the NVM. An invalid nodeMap could result from
// erroneous writes to NVM locations or simply a brand new HW board. If all is OK, we have a valid basic
// nodeMap that we can work from.
//
// The setup of the portMap follows. The flag field contains dynamic flags that are always reseted on node
// start or reset. Other fields in a map are read in from the NVM first and set to a default state this way.
//
// The eventMap initialization is a bit special, in that it is a rather large map and potentially only a
// portion is used. There is an eventMao high water mark field in the nodeMap that will tell how many entries
// are actually used in the event map. Adding increases, deleting decreases the high water mark. Note that
// the eventMap is a sorted map. Every time we insert or remove the eventMap is rebuilt. Instead of
// immediately updating the NVM storage, a dedicated command will SYNC between the MEM and the NVM eventMap.
//
// Next, we will set up the pending request map, callback function and task map. They are just memory data
// areas to be initialized. Up to here an error detected will result in a fatal error. If a console is
// connected the error messages are listed for analysis.
//
// If all is OK so far, extension boards are located, and if there are any, their initialization follows.
// First, we try to detect any. For all detected entries we validate the extension board NVM header and
// set the driver for a valid header found. The driver data area is copied to its memory counter part.
// All drivers are ready by then.
//
// The overall logic of the startup routine code below is that if there is a fault, the follow on steps are
// simply  skipped and the node is put into the FAIL state. Note that we still are able to access the node
// via the USB console and one day also via diagnostic LCS messages. The idea is to allow the correct
// configuration  of the nodeMap, so that we can restart with a correct nodeMap.
//
// ??? how do we deal wit PFAIL restarts ?
// ??? we could have also callbacks for the "restart" case ? or pass to init a flag...
//----------------------------------------------------------------------------------------------------------
uint8_t initRuntime( LcsConfigDesc *lcsConfig, CDC::CdcConfigDesc *cdcConfig ) {

    uint8_t rStat = ALL_OK;
```

```

    if ( rStat == ALL_OK )  rStat = initCdcLayer( cdcConfig );
    if ( rStat != ALL_OK )  CDC::fatalErrorMsg((char *) "Fatal: CDC Setup failed", 1, rStat );

    CDC::writeDio( cdcConfig -> READY_LED_PIN, false );
    CDC::writeDio( cdcConfig -> ACTIVE_LED_PIN, true );

    if ( rStat == ALL_OK )  rStat = initCanBus( cdcConfig );
    if ( rStat == ALL_OK )  rStat = initNvmChannels( cdcConfig );
    if ( rStat != ALL_OK )  CDC::fatalErrorMsg((char *) "Fatal: CAN bus or NVM Setup failed", 2, rStat );

    if ( rStat == ALL_OK )  rStat = setupNodeMap( lcsConfig );
    if ( rStat == ALL_OK )  rStat = setupCdcMap( cdcConfig );
    if ( rStat == ALL_OK )  rStat = setupPortMap( );
    if ( rStat == ALL_OK )  rStat = setupNodeDataMap( );
    if ( rStat == ALL_OK )  rStat = setupEventMap( );
    if ( rStat == ALL_OK )  rStat = setupUserMap( );
    if ( rStat == ALL_OK )  rStat = setupCallbackMap( );
    if ( rStat == ALL_OK )  rStat = setupTaskMap( );
    if ( rStat == ALL_OK )  rStat = setupPendingReqMap( );
    if ( rStat == ALL_OK )  rStat = setupDrvFuncMap( );
    if ( rStat == ALL_OK )  rStat = setupDrvMap( );
    if ( rStat == ALL_OK )  rStat = registerInternalTasks( );
    if ( rStat != ALL_OK )  CDC::fatalErrorMsg((char *) "Node setup Setup failed", 3, rStat );

    if ( rStat == ALL_OK )  rStat = detectExtensionBoards( );
    if ( rStat == ALL_OK )  rStat = setupExtensionBoards( );
    if ( rStat != ALL_OK )  printf( "Extension boards setup Setup failed, stat: %d\n", rStat );

    if ( rStat == ALL_OK )  nodeMap.nodeState = NS_INIT;
    else                    nodeMap.nodeState = NS_FAIL;

    if ( rStat == ALL_OK )  {

        CDC::writeDio( cdcConfig -> READY_LED_PIN, true );
        CDC::writeDio( cdcConfig -> ACTIVE_LED_PIN, false );
    }

    if ( debugMask & ( DBG_CONFIG && DBG_SETUP )) printf( "init LCS runtime, status: %d \n", rStat );
    return ( rStat );
}

//----------------------------------------------------------------------------------------------------------
// "startRuntime" is the main routine of the node activity processing. All it does is to call the node
// state machine.
//
//-----------------------------------------------------------------------------------------------------------
void startRuntime( ) {

     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_SETUP )) printf( "Start LCS runtime\n");

     handleNodeState( );
}

}; // 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.
//
//------------------------------------------------------------------------------------------------------------
```

```
uint8_t addToPendingReqMap( uint16_t npId, uint32_t timeoutVal = 0 ) {

    uint32_t ts = CDC::getMillis( );

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

        if ( pendingReqMap.map[ i ].npId == 0 ) {

            pendingReqMap.map[ i ].npId = npId;
            pendingReqMap.map[ i ].reqTimeoutTs = (( timeoutVal != 0 ) ? ts + timeoutVal : timeoutVal );
            return ( ALL_OK );
        }
    }

    return ( ERR_PENDING_REQ_MAP_FULL );
}

//----------------------------------------------------------------------------------------------------
// "removeFromPendingReqMap" removes an entry from the pending reply buffer. If the entry is not found, we
// received a reply for a request that we do not know. Right now, we just ignore this error.
//
//----------------------------------------------------------------------------------------------------
uint8_t removeFromPendingReqMap( uint16_t npId ) {

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

        if ( pendingReqMap.map[ i ].npId == npId ) pendingReqMap.map[ i ].npId = NIL_NODE_ID;
    }

    return ( ALL_OK );
}

//----------------------------------------------------------------------------------------------------
// "searchPendingReqMap" searches the pending request buffer for a matching node. We just return a boolean
// answer whether the entry is there or not.
//
//----------------------------------------------------------------------------------------------------
bool searchPendingReqMap( uint16_t npId ) {

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

        if ( pendingReqMap.map[ i ].npId == npId ) return ( true );
    }

    return ( false );
}

}; // namespace


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

//----------------------------------------------------------------------------------------------------
// "setupMsgBus" is called during node initialization to setup the LCS message bus interface. Right now,
// only the CAN bus is supported. We first create the CAN Bus object and then call its initialization routine.
// The canId and nodeId are identical. We ensure through LCS configuration that the nodeIds are unique.
//
//
// ??? how do we configure the CAN control mode ?
//----------------------------------------------------------------------------------------------------
uint8_t setupMsgBus( ) {

    uint8_t     rStat           = ALL_OK;
    uint16_t    canBusCtrlMode  = CAN_BUS_LIB_PICO_PIO_125K_M_CORE;
    uint8_t     canBusTxPin     = 0;
    uint8_t     canBusRxPin     = 0;

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_MSG_BUS )) {

        printf( "setupMsgBus -> %d:%d:%d%d\n", nodeMap.nodeId, canBusRxPin, canBusTxPin, canBusCtrlMode );
    }

    switch ( canBusCtrlMode ) {

        case CAN_BUS_LIB_PICO_PIO_125K:
        case CAN_BUS_LIB_PICO_PIO_250K:
        case CAN_BUS_LIB_PICO_PIO_500K:
        case CAN_BUS_LIB_PICO_PIO_1000K:
        case CAN_BUS_LIB_PICO_PIO_125K_M_CORE:
        case CAN_BUS_LIB_PICO_PIO_250K_M_CORE:
        case CAN_BUS_LIB_PICO_PIO_500K_M_CORE:
        case CAN_BUS_LIB_PICO_PIO_1000K_M_CORE: {

            msgBus = new LcsMsgBusCAN( );
            rStat = (( LcsMsgBusCAN *) msgBus ) -> init( nodeMap.nodeId, canBusRxPin, canBusTxPin, canBusCtrlMode );

        } break;

        default: rStat = ERR_CAN_SETUP;
    }

    if ( rStat != ALL_OK ) {

         if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_MSG_BUS ))
            printf( "setup CAN Bus failed: %d\n", rStat );
```

```
        return ( ERR_CAN_SETUP );
    }
     else {

         if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_MSG_BUS )) printf( " -> OK\n" );
        return ( ALL_OK );
    }
}

//----------------------------------------------------------------------------------------------------
// The primary task of the receive function is to receive an LCS messages and pass them to the respective
// handler method. In order to not always check whether a valid message was processed, this routine will
// always return a valid message opCode. The "LCS_NO_MSG" pseudo message is used to indicate that something
// else happened and no further message processing is required. We also maintain a request / reply map to
// keep track of outstanding requests transparently to the caller.
//
// ??? should we have a pre-filter for message ID and nodeId match ? this would be perhaps useful, when we
// run CAN bus on two cores on the pico. Then the checking would run on the interrupt handler processor.
// However, the pending request map is then accessed by two cores and we need to sync the access.
//----------------------------------------------------------------------------------------------------
uint8_t receiveLcsMsg( uint8_t *msg ) {

    int rStat = msgBus -> receiveLcsMsg( msg );

    if ( rStat == ALL_OK )  {

         if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_MSG_BUS )) {

             printf( "Can Msg Received (OpCode): 0x%x\n", msg[ 0 ] );
        }

        if (( msg[ 0 ] == LCS_OP_NODE_REP ) || ( msg[ 0 ] == LCS_OP_ACK ) || ( msg[ 0 ] == LCS_OP_ERR )) {

             uint16_t nodeId = (( msg[1] << 8 ) + msg[2] ) >> 4;

            if ( searchPendingReqMap( nodeId )) {

                removeFromPendingReqMap( nodeId );
                return ( msg[ 0 ] );

            } else return ( LCS_OP_NO_MSG );

        } else return ( msg[ 0 ] );

    } else return ( LCS_OP_NO_MSG );
}

//----------------------------------------------------------------------------------------------------
// A simple helper to print an LCS message.
//
//----------------------------------------------------------------------------------------------------
void printLcsMsg( uint8_t *msg ) {

    printf( "LCS MSG: op: %d, data: ", msg[ 0 ] & 0x1F );
    for ( int i = 0; i < ( msg[ 0 ] >> 5 ) + 1; i ++ ) printf( "0x%x ", msg[ i ] );
    printf( "\n" );
}

//----------------------------------------------------------------------------------------------------
// "processPendingReqMapTimeouts" is part of the periodic processing of the node. It will check wether any
// requests waiting for a reply have timed out. In this case, we should invoke the reply callback with an
// error code.
//
//----------------------------------------------------------------------------------------------------
void processPendingReqMapTimeouts( ) {

    uint32_t ts = CDC::getMillis( );

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

        if ( pendingReqMap.map[ i ].reqTimeoutTs != 0 ) {

            if ( pendingReqMap.map[ i ].reqTimeoutTs < ts ) {

                if ( callbackMap.repCallback != nullptr ) {

                    callbackMap.repCallback( pendingReqMap.map[ i ].npId, 0, 0, 0, ERR_REQ_TIMEOUT );
                }

                // ??? clear entry ?

            }
        }
    }
}

//----------------------------------------------------------------------------------------------------
// Some messages are requests that expect a reply. We maintain a pending request map which keeps track of
// outstanding requests.
//
//----------------------------------------------------------------------------------------------------
uint8_t sendTimedReq( uint16_t npId, uint8_t *msg, uint8_t msgPri, uint32_t timeout ) {

    if ( addToPendingReqMap( npId ) == ALL_OK ) {

        return ( msgBus -> sendLcsMsg( msg, msgPri ));
    }
    else return ( ERR_NODE_OUTSTANDING_REQ_LIMIT );
```

```
}

//----------------------------------------------------------------------------------------------------------
// LCB message send routines. They all follow the same pattern. There is a method for each message opcode,
// which maps the input parameters to the byte array and then send it. Straightforward. For messages that are
// a part of a request / reply pair, the requesting messages will also add the requesting nodeId to the
// pending request map. This way we know that there is an outstanding request. The receiving message handler
// will clear the entry upon the matching receipt.
//
//----------------------------------------------------------------------------------------------------------
uint8_t sendCfg( uint16_t npId ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_CFG };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );

    return( sendTimedReq( npId, msgBuf,  MSG_PRI_HIGH , 0 ));
}

uint8_t sendOps( uint16_t npId ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_OPS };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );

    return( sendTimedReq( npId, msgBuf,  MSG_PRI_HIGH , 0 ));
}

uint8_t sendReset( uint16_t npId ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_RESET };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );

    return ( sendTimedReq( npId, msgBuf, MSG_PRI_HIGH, 0 ));
}

uint8_t sendBusOn( ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_BON };
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_VERY_HIGH ));
}

uint8_t sendBusOff( ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_BOF };
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_VERY_HIGH ));
}

uint8_t sendErr( uint16_t npId, uint8_t errCode, uint8_t arg1, uint8_t arg2 ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_ERR };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );
    msgBuf[ 3 ] = errCode;
    msgBuf[ 4 ] = arg1;
    msgBuf[ 5 ] = arg2;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
}

uint8_t sendPing( uint16_t npId ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_PING };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
}

uint8_t sendAck( uint16_t npId ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_ACK };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
}

uint8_t sendReqNodeId( uint16_t npId, uint32_t nodeUID, uint8_t flags ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_REQ_NID };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );
    msgBuf[ 3 ] = ( nodeUID & 0xFF000000 ) >> 24;
    msgBuf[ 4 ] = ( nodeUID & 0x00FF0000 ) >> 16;
    msgBuf[ 5 ] = ( nodeUID & 0x0000FF00 ) >> 8;
    msgBuf[ 6 ] = ( nodeUID & 0x000000FF );
    msgBuf[ 7 ] = flags;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
}

uint8_t sendRepNodeId( uint16_t npId, uint32_t nodeUID ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_REP_NID };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );
    msgBuf[ 3 ] = ( nodeUID & 0xFF000000 ) >> 24;
    msgBuf[ 4 ] = ( nodeUID & 0x00FF0000 ) >> 16;
    msgBuf[ 5 ] = ( nodeUID & 0x0000FF00 ) >> 8;
    msgBuf[ 6 ] = ( nodeUID & 0x000000FF );
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
```

```
}

uint8_t sendSetNodeId( uint16_t npId, uint32_t nodeUID ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_SET_NID };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );
    msgBuf[ 3 ] = ( nodeUID & 0xFF000000 ) >> 24;
    msgBuf[ 4 ] = ( nodeUID & 0x00FF0000 ) >> 16;
    msgBuf[ 5 ] = ( nodeUID & 0x0000FF00 ) >> 8;
    msgBuf[ 6 ] = ( nodeUID & 0x000000FF );
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
}

uint8_t sendNodeIdCollision( uint16_t npId, uint32_t nodeUID ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_NCOL };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );
    msgBuf[ 3 ] = ( nodeUID & 0xFF000000 ) >> 24;
    msgBuf[ 4 ] = ( nodeUID & 0x00FF0000 ) >> 16;
    msgBuf[ 5 ] = ( nodeUID & 0x0000FF00 ) >> 8;
    msgBuf[ 6 ] = ( nodeUID & 0x000000FF );
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_HIGH ));
}

uint8_t sendGetNode( uint16_t npId, uint8_t item, uint16_t val1, uint16_t val2 ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_NODE_GET };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );
    msgBuf[ 3 ] = item;
    msgBuf[ 4 ] = highByte( val1 );
    msgBuf[ 5 ] = lowByte( val1 );
    msgBuf[ 6 ] = highByte( val2 );
    msgBuf[ 7 ] = lowByte( val2 );
    return( sendTimedReq( npId, msgBuf,  MSG_PRI_NORMAL, 0 ));
}

uint8_t sendSetNode( uint16_t npId, uint8_t item, uint16_t val1, uint16_t val2 ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_NODE_PUT };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );
    msgBuf[ 3 ] = item;
    msgBuf[ 4 ] = highByte( val1 );
    msgBuf[ 5 ] = lowByte( val1 );
    msgBuf[ 6 ] = highByte( val2 );
    msgBuf[ 7 ] = lowByte( val2 );
    return( sendTimedReq( npId, msgBuf,  MSG_PRI_NORMAL, 0 ));
}

uint8_t sendReqNode( uint16_t npId, uint8_t item, uint16_t val1, uint16_t val2 ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_NODE_REQ };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );
    msgBuf[ 3 ] = item;
    msgBuf[ 4 ] = highByte( val1 );
    msgBuf[ 5 ] = lowByte( val1 );
    msgBuf[ 6 ] = highByte( val2 );
    msgBuf[ 7 ] = lowByte( val2 );
    return( sendTimedReq( npId, msgBuf,  MSG_PRI_LOW, 0 ));
}

uint8_t sendRepNode( uint16_t npId, uint8_t item, uint16_t val1, uint16_t val2 ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_NODE_REP };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );
    msgBuf[ 3 ] = item;
    msgBuf[ 4 ] = highByte( val1 );
    msgBuf[ 5 ] = lowByte( val1 );
    msgBuf[ 6 ] = highByte( val2 );
    msgBuf[ 7 ] = lowByte( val2 );
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
}

uint8_t sendEventOn( uint16_t npId, uint16_t eventId ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_EVT_ON };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );
    msgBuf[ 3 ] = highByte( eventId );
    msgBuf[ 4 ] = lowByte( eventId );

     if ( nodeId( npId ) == nodeMap.nodeId ) {

        handleMsgEvent( msgBuf );
        return( ALL_OK );
    }
    else return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
}

uint8_t sendEventOff( uint16_t npId, uint16_t eventId ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_EVT_OFF };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );
```

```
    msgBuf[ 3 ] = highByte( eventId );
    msgBuf[ 4 ] = lowByte( eventId );

    if ( nodeId( npId ) == nodeMap.nodeId ) {

        handleMsgEvent( msgBuf );
        return( ALL_OK );
    }
    else return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
}

uint8_t sendEvent( uint16_t npId, uint16_t eventId, uint16_t arg ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_EVT };
    msgBuf[ 1 ] = highByte( npId );
    msgBuf[ 2 ] = lowByte( npId );
    msgBuf[ 3 ] = highByte( eventId );
    msgBuf[ 4 ] = lowByte( eventId );
    msgBuf[ 5 ] = highByte( arg );
    msgBuf[ 6 ] = lowByte( arg );

    if ( nodeId( npId ) == nodeMap.nodeId ) {

        handleMsgEvent( msgBuf );
        return( ALL_OK );
    }
    else return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
}

uint8_t sendTrackOn( ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_TON };
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_HIGH ));
}

uint8_t sendTrackOff( ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_TOF };
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_HIGH ));
}

uint8_t sendEstop( ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_ESTP };
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_VERY_HIGH ));
}

uint8_t sendReqLoc( uint16_t locAdr, uint8_t flags ) {

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

uint8_t sendRelLoc( uint8_t sId ) {

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

uint8_t sendRepLoc( uint8_t sId, uint16_t locAdr, uint8_t spDir, uint8_t fn1, uint8_t fn2, uint8_t fn3  ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_REP_LOC };
    msgBuf[ 1 ] = sId;
    msgBuf[ 2 ] = highByte( locAdr );
    msgBuf[ 3 ] = lowByte( locAdr );
    msgBuf[ 4 ] = spDir;
    msgBuf[ 5 ] = fn1;
    msgBuf[ 6 ] = fn2;
    msgBuf[ 7 ] = fn3;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendLocConsist( uint8_t sId, uint8_t consId, uint8_t flags ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_SET_LCON };
    msgBuf[ 1 ] = sId;
    msgBuf[ 2 ] = consId;
    msgBuf[ 3 ] = flags;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendQueryLoc( uint8_t sId ) {

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

uint8_t sendKeepLoc( uint8_t sId ) {

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

```

```
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 ) {
```

```
    uint8_t msgBuf[ 8 ] = { LCS_OP_SEND_DCC3 };
    msgBuf[ 1 ] = arg1;
    msgBuf[ 2 ] = arg2;
    msgBuf[ 3 ] = arg3;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendDccPacket( uint8_t arg1, uint8_t arg2, uint8_t arg3, uint8_t arg4 ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_SEND_DCC4 };
    msgBuf[ 1 ] = arg1;
    msgBuf[ 2 ] = arg2;
    msgBuf[ 3 ] = arg3;
    msgBuf[ 4 ] = arg4;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendDccPacket( uint8_t arg1, uint8_t arg2, uint8_t arg3, uint8_t arg4, uint8_t arg5 ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_SEND_DCC5 };
    msgBuf[ 1 ] = arg1;
    msgBuf[ 2 ] = arg2;
    msgBuf[ 3 ] = arg3;
    msgBuf[ 4 ] = arg4;
    msgBuf[ 5 ] = arg5;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendDccPacket( uint8_t arg1, uint8_t arg2, uint8_t arg3, uint8_t arg4, uint8_t arg5, uint8_t arg6 ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_SEND_DCC6 };
    msgBuf[ 1 ] = arg1;
    msgBuf[ 2 ] = arg2;
    msgBuf[ 3 ] = arg3;
    msgBuf[ 4 ] = arg4;
    msgBuf[ 5 ] = arg5;
    msgBuf[ 6 ] = arg6;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_NORMAL ));
}

uint8_t sendDccAck( ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_DCC_ACK };
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
}

uint8_t sendDccErr( uint8_t errCode, uint8_t arg1, uint8_t arg2 ) {

    uint8_t msgBuf[ 8 ] = { LCS_OP_DCC_ERR };
    msgBuf[ 1 ] = errCode;
    msgBuf[ 2 ] = arg1;
    msgBuf[ 3 ] = arg2;
    return ( msgBus -> sendLcsMsg( msgBuf, MSG_PRI_LOW ));
}

}; // 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 );
```

```
    }

    //-------------------------------------------------------------------------------------------------------
    // "writeAttrMem" stores a value to a 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 writeAttrMem( uint8_t block, uint8_t item, uint16_t arg ) {

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

    //------------------------------------------------------------------------------------------------------
    // "readAttrNvm" gets an attribute from the NVM storage. We read the value from NVM, store it in the MEM
    // counterpart and then return it. For the NVM access, the byte offset into the storage needs to be
    // computed. As an internal function, we expect a valid block and item argument.
    //
    //-------------------------------------------------------------------------------------------------------
    uint8_t readAttrNvm( uint8_t block, uint8_t item, uint16_t *arg ) {

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

            printf( "readAttrNvm: block: 0x%x, item: %d\n", block, item );
        }

        uint16_t index  = item - IR_ATTR_NVM_RANGE_START;
        uint16_t ofs    = NVM_NODE_DATA_START + (( block * MAX_ATTR_MAP_ENTRIES ) + index  ) * sizeof( uint16_t );

        printf( "Ofs: 0x%x\n", ofs );

        uint8_t rStat = rtNvmGetWord( ofs, &nodeData.map[ block ][ index ] );

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

        *arg = (( rStat == ALL_OK ) ? nodeData.map[ block ][ index ] : 0 );
        return ( rStat );
    }

    //------------------------------------------------------------------------------------------------------
    // "writeAttrNvm" stores an attribute to the NVM storage. We first update the corresponding MEM attribute
    // and then write the value to NVM storage. For the NVM access, the byte offset into the storage needs to
    // be computed. As an internal function, we expect a valid block and item argument.
    //
    //------------------------------------------------------------------------------------------------------
    uint8_t writeAttrNvm( uint8_t block, uint8_t item, uint16_t arg ) {

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

            printf( "readAttrNvm: block: 0x%x, item: %d\n", block, item );
        }

        uint16_t index  = item - IR_ATTR_NVM_RANGE_START;
        uint16_t ofs    = NVM_NODE_DATA_START + (( block * MAX_ATTR_MAP_ENTRIES ) + index  ) * sizeof( uint16_t );

        printf( "Ofs: 0x%x\n", ofs );

        nodeData.map[ block ][ index ] = arg;
        return ( rtNvmPutWord( ofs, arg ));
    }

    //------------------------------------------------------------------------------------------------------
    // User callback function invocation routine. Items 64 to 127 are user defined items. We will simply
    // invoke a previously registered callback passing the arguments.
    //
    //------------------------------------------------------------------------------------------------------
    uint8_t invokeUserItemCallback( uint8_t portId, uint8_t item, uint16_t *arg1, uint16_t *arg2 ) {

        if ( callbackMap.reqCallback != nullptr ) {

            return ( callbackMap.reqCallback( portId, item, arg1, arg2 ));
        }
        else return( ERR_INVALID_ITEM_ID );
    }

} // namespace


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

//----------------------------------------------------------------------------------------------------------
// "nodeGet" will lookup a value from the node, port or the attribute data map. The "npId" argument contains
// the node and port Id. However, we will only use the portId portion, which represents the block index.
//
//----------------------------------------------------------------------------------------------------------
uint8_t nodeGet( uint16_t npId, uint8_t item, uint16_t *arg1, uint16_t *arg2 ) {

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

        printf( "nodeGet: 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_ATTR_MEM_RANGE_START, IR_ATTR_MEM_RANGE_END )) {

        return ( readAttrMem( portId( npId ), item, arg1 ));
    }
    else if ( isInRangeU( item, IR_ATTR_NVM_RANGE_START, IR_ATTR_NVM_RANGE_END )) {

        return ( readAttrNvm( portId( npId ), item, arg1 ));
    }
    else {

        switch ( item ) {

            case ITEM_ID_DEBUG_MASK: {

                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
                *arg1 = debugMask;
                return( ALL_OK );
            }

            case ITEM_ID_OPTIONS: {

                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );

                *arg1 = nodeMap.nodeOptions;
                return ( ALL_OK );
            }

            case ITEM_ID_FLAGS: {

                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );

                if ( portId( npId ) == 0 )   *arg1 = nodeMap.nodeFlags;
                else                         *arg1 = portMap.map[ portId( npId ) - 1 ].flags;

                return ( ALL_OK );
            }

            case ITEM_ID_VERSION: {

                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );

                *arg1 = nodeMap.nodeSwVersion;
                return ( ALL_OK );
            }

            case ITEM_ID_TYPE: {

                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );

                if ( portId( npId ) == 0 )   *arg1 = nodeMap. nodeType;
                else                         *arg1 = portMap.map[ portId( npId ) - 1 ].type;

                return ( ALL_OK );
            }

            case ITEM_ID_CONTROLLER_FAMILY: {

                 if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );

                *arg1 = nodeMap.head.controllerFamily;
                return ( ALL_OK );
            }

            case ITEM_ID_NODE_ID: {

                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );

                *arg1 = nodeMap.nodeId;
                return ( ALL_OK );
            }

            case ITEM_ID_NODE_UID: {

                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
                if ( arg2 == nullptr ) return( ERR_INVALID_ATTR_ARG );

                *arg1 = nodeMap.nodeUID >> 16;
                *arg2 = nodeMap.nodeUID & 0xFFFF;
                return ( ALL_OK );
            }

            case ITEM_ID_RESTART_COUNT: {

                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );

                *arg1 = nodeMap.nodeRestartCnt;
                return ( ALL_OK );
            }

            case ITEM_ID_PORT_MAP_ENTRIES: {

                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );

                *arg1 = nodeMap.portMapEntries;
                return ( ALL_OK );
            }

            case ITEM_ID_EVENT_MAP_ENTRIES: {

```

```
                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );

                *arg1 = nodeMap.eventMapEntries;
                return ( ALL_OK );
            }

            case ITEM_ID_ATTR_MAP_ENTRIES: {

                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );

                *arg1 = MAX_ATTR_MAP_ENTRIES;
                return ( ALL_OK );
            }

            case ITEM_ID_GET_EVENT_MAP_ENTRY: {

                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
                if ( arg2 == nullptr ) return( ERR_INVALID_ATTR_ARG );

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

            case ITEM_ID_NAME_1: {

                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
                if ( arg2 == nullptr ) return( ERR_INVALID_ATTR_ARG );

                if ( portId( npId ) == 0 ) {

                    *arg1 = ((uint16_t) ( nodeMap.name[ 0 ] << 8  ) | nodeMap.name[ 1 ] );
                    *arg2 = ((uint16_t) ( nodeMap.name[ 2 ] << 8  ) | nodeMap.name[ 3 ] );
                }
                else {

                    LcsPortMapEntry *pEntry = &portMap.map[ portId( npId ) -1 ];

                    *arg1 = ((uint16_t) ( pEntry -> name[ 0 ] << 8  ) | pEntry -> name[ 1 ] );
                    *arg2 = ((uint16_t) ( pEntry -> name[ 2 ] << 8  ) | pEntry -> name[ 3 ] );
                }

                return ( ALL_OK );
            }

            case ITEM_ID_NAME_2: {

                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
                if ( arg2 == nullptr ) return( ERR_INVALID_ATTR_ARG );

                if ( portId( npId ) == 0 ) {

                    *arg1 = ((uint16_t) ( nodeMap.name[ 4 ] << 8  ) | nodeMap.name[ 5 ] );
                    *arg2 = ((uint16_t) ( nodeMap.name[ 6 ] << 8  ) | nodeMap.name[ 7 ] );
                }
                else {

                    LcsPortMapEntry *pEntry = &portMap.map[ portId( npId ) - 1 ];

                    *arg1 = ((uint16_t) ( pEntry -> name[ 4 ] << 8  ) | pEntry -> name[ 5 ] );
                    *arg2 = ((uint16_t) ( pEntry -> name[ 6 ] << 8  ) | pEntry -> name[ 7 ] );
                }

                return ( ALL_OK );
            }

            case ITEM_ID_NAME_3: {

                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
                if ( arg2 == nullptr ) return( ERR_INVALID_ATTR_ARG );

                if ( portId( npId ) == 0 ) {

                    *arg1 = ((uint16_t) ( nodeMap.name[ 8 ] << 8  ) | nodeMap.name[ 9 ] );
                    *arg2 = ((uint16_t) ( nodeMap.name[ 10 ] << 8  ) | nodeMap.name[ 11 ] );
                }
                else {

                    LcsPortMapEntry *pEntry = &portMap.map[ portId( npId ) - 1 ];

                    *arg1 = ((uint16_t) ( pEntry -> name[ 8 ] << 8  ) | pEntry -> name[ 9 ] );
                    *arg2 = ((uint16_t) ( pEntry -> name[ 10 ] << 8  ) | pEntry -> name[ 11 ] );
                }

                return ( ALL_OK );
            }

            case ITEM_ID_NAME_4: {

                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );
                if ( arg2 == nullptr ) return( ERR_INVALID_ATTR_ARG );

                if ( portId( npId ) == 0 ) {

                    *arg1 = ((uint16_t) ( nodeMap.name[ 12 ] << 8  ) | nodeMap.name[ 13 ] );
                    *arg2 = ((uint16_t) ( nodeMap.name[ 14 ] << 8  ) | nodeMap.name[ 15 ] );
                }
                else {

                    LcsPortMapEntry *pEntry = &portMap.map[ portId( npId ) - 1 ];

```

```
                    *arg1 = ((uint16_t) ( pEntry -> name[ 12 ] << 8  ) | pEntry -> name[ 13 ] );
                    *arg2 = ((uint16_t) ( pEntry -> name[ 14 ] << 8  ) | pEntry -> name[ 15 ] );
                }

                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;

            case ITEM_ID_EVENT_DELAY_TICKS: {

                if ( arg1 == nullptr ) return( ERR_INVALID_ATTR_ARG );

                *arg1 = portMap.map[ portId( npId ) - 1 ].eventDelayTime;
                return ( ALL_OK );
            }

            default: return ( ERR_INVALID_ITEM_ID );
        }
    }
}

//--------------------------------------------------------------------------------------------------------
// "nodePut" will write a value to the node, port or the attribute data map. The "npId" argument contains
// the node and port Id. However, we will only use the portId portion.
//
//--------------------------------------------------------------------------------------------------------
uint8_t nodePut( uint16_t npId, uint8_t item, uint16_t val1, uint16_t val2 ) {

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

        printf( "nodePut: 0x%x:%d:%d:%d\n", npId, item, val1, val2  );
    }

   if ( isInRangeU( item, IR_ATTR_MEM_RANGE_START, IR_ATTR_MEM_RANGE_END )) {

        return ( writeAttrMem( portId( npId ), item, val1 ));
    }
    else if ( isInRangeU( item, IR_ATTR_NVM_RANGE_START, IR_ATTR_NVM_RANGE_END )) {

        return ( writeAttrNvm( portId( npId ), item, val1 ));
    }
    else {

        switch ( item ) {

            case ITEM_ID_DEBUG_MASK: {

                if ( CDC::isConsoleConnected( ))     debugMask = val1 | DBG_CONFIG;
                else                                 debugMask = val1 & ~ DBG_CONFIG;

                return( ALL_OK );
            }

            case ITEM_ID_VERSION: {

                nodeMap.nodeSwVersion = val1;
                return( rtNvmPutWord( NVM_NODE_MAP_START + offsetof( LcsNodeMap, nodeSwVersion ), val1 ));
            }

            case ITEM_ID_OPTIONS: {

                nodeMap.nodeOptions = val1;
                return( rtNvmPutWord( NVM_NODE_MAP_START + offsetof( LcsNodeMap, nodeOptions ), val1 ));
            }

            case ITEM_ID_FLAGS: {

                nodeMap.nodeFlags = val1;
                return( rtNvmPutWord( NVM_NODE_MAP_START + offsetof( LcsNodeMap, nodeFlags ), val1 ));
            }

            case ITEM_ID_NODE_ID: {

                nodeMap.nodeId = val1;
                return( rtNvmPutWord( NVM_NODE_MAP_START + offsetof( LcsNodeMap, nodeId ), nodeMap.nodeId ));
            }

            case ITEM_ID_TYPE: {

                if ( portId( npId ) == 0 ) {

                    nodeMap.nodeType = lowByte( val1 );
                    return( rtNvmPutWord( NVM_NODE_MAP_START + offsetof( LcsNodeMap, nodeType ), val1 ));
                }
                else {

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

                    uint16_t ofs =  NVM_PORT_MAP_START +
                                    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.
//
```

```
//----------------------------------------------------------------------------------------------------
uint8_t addToMemEventMap( uint16_t eventId, uint16_t portId ) {

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_EVENTS )) {

        printf( "Add to MEM Event Map: %d : %d\n", eventId, portId );
    }

    if ( searchEvent( eventId, portId ) >= 0 )    return ( ALL_OK );
    if ( nodeMap.eventMapHwm >= MAX_EVENT_MAP_ENTRIES )  return ( ERR_EVENT_MAP_FULL );

    uint16_t index = nodeMap.eventMapHwm;

    if ( nodeMap.eventMapHwm > 0 ) {

        while (( index > 0 ) && ( compareEventEntry( &eventMap.map[ index - 1 ], eventId, portId ) > 0 )) {

            eventMap.map[ index ] = eventMap.map[ index - 1 ];
            index --;
        }
    }

    eventMap.map[ index ].eventId = eventId;
    eventMap.map[ index ].portId  = portId;
    nodeMap.eventMapHwm++;

    return ( ALL_OK );
}

//----------------------------------------------------------------------------------------------------
// "removeFromMemEventMap" removes an entry from the memory event map. The sorted order is maintained.
//
//----------------------------------------------------------------------------------------------------
uint8_t removeFromMemEventMap( uint16_t eventId, uint16_t portId ) {

     if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_EVENTS )) {

        printf( "Remove from MEM Event Map: %d : %d \n", eventId, portId );
    }

    int index = searchEvent( eventId, portId );

    if ( index >= 0 ) {

        nodeMap.eventMapHwm--;

        for ( uint16_t i = index; i < nodeMap.eventMapHwm; i++ )
            eventMap.map[ i ] = eventMap.map[ i + 1 ];
    }

    return ( ALL_OK );
}

} // namespace

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

//----------------------------------------------------------------------------------------------------
// The "addEvent" routine will add an eventId/portId combination to the event map if not already there. If
// the portId parameter is a NIL_PORT_ID, the event is added to all portMap entries.
//
//----------------------------------------------------------------------------------------------------
uint8_t addEvent( uint16_t eventId, uint16_t portId ) {

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_EVENTS )) {

        printf( "Add Event: event: %d, port: %d\n", eventId, portId );
    }

    int rStat = ALL_OK;

    if ( ! isInRangeU( eventId, MIN_EVENT_ID, MAX_EVENT_ID )) return ( ERR_INVALID_EVENT_ID );
    if ( portId >  MAX_PORT_ID ) return ( ERR_INVALID_PORT_ID );

    if ( portId == NIL_PORT_ID ) {

        for ( uint8_t p = 1; p <= MAX_PORT_MAP_ENTRIES; p++ ) {

            rStat = addToMemEventMap( eventId, p );
            if ( rStat != ALL_OK ) break;
        }
    }
    else {

        rStat = addToMemEventMap( eventId, portId );
    }

    return ( rStat );
}

//----------------------------------------------------------------------------------------------------
// The "removeEvent" routine will remove an event Id / port Id from the MEM event map. If the port ID is
// NIL_PORT_ID, all port map entries matching event Id are removed.
//
//----------------------------------------------------------------------------------------------------
```

```
uint8_t removeEvent( uint16_t eventId, uint16_t portId ) {

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_EVENTS )) {

        printf( "Remove Event: %d : %d\n", eventId, portId );
    }

    int rStat = ALL_OK;

    if ( ! isInRangeU( eventId, MIN_EVENT_ID, MAX_EVENT_ID )) return ( ERR_INVALID_EVENT_ID );

    if ( portId == NIL_PORT_ID ) {

        for ( uint16_t p = 1; p <= MAX_PORT_MAP_ENTRIES; p++ ) {

            rStat = removeFromMemEventMap( eventId, p );
            if ( rStat != ALL_OK ) break;
        }
    }
    else if ( isInRangeU( portId, MIN_PORT_ID, MAX_PORT_ID )) {

        rStat = removeFromMemEventMap( eventId, portId );
    }
    else rStat = ERR_INVALID_PORT_ID;

    return ( rStat );
}

//-----------------------------------------------------------------------------------------------------
// The event search function performs a binary search of the event map using the event Id and the port Id.
// If the port Id is NIL, a matching entry with lowest portId is returned. All eventMap entries with the
// same eventId follow. If the entry cannot be found, a -1 is returned.
//
//-----------------------------------------------------------------------------------------------------
int searchEvent( uint16_t eventId, uint16_t portId ) {

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_EVENTS )) {

        printf( "Search Event: %d : %d", eventId, portId );
    }

    int   res    = -1;
    int   low    = 0;
    int   high   = nodeMap.eventMapHwm - 1;

    if ( portId == NIL_PORT_ID ) {

        while ( low <= high ) {

            int mid = low + ( high - low + 1 ) / 2;

            if      ( eventMap.map[ mid ].eventId < eventId ) low  = mid + 1;
            else if ( eventMap.map[ mid ].eventId > eventId ) high = mid - 1;
            else if ( eventMap.map[ mid ].eventId == eventId ) {

                res   = mid;
                high  = mid - 1;
            }
        }
    }
    else {

        while ( low <= high ) {

            int mid = low + ( high - low ) / 2;
            int tst = compareEventEntry( &eventMap.map[ mid ], eventId, portId );

            if      ( tst < 0 ) low   = mid + 1;
            else if ( tst > 0 ) high  = mid - 1;
            else {

                res = mid;
                break;
            }
        }
    }

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_EVENTS )) printf( "-> %d\n", res );
    return ( res );
}

//-----------------------------------------------------------------------------------------------------
// "syncEventMap" will write back the sorted MEM event map. We only write up to the HWM mark, which points
// right after the last element in the sorted MEM event map. The idea is that all adds and removes are done
// on the MEM event map and a SYNC control call will flush the sorted MEM event map to NVM.
//
//-----------------------------------------------------------------------------------------------------
uint8_t syncEventMap( ) {

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_EVENTS )) printf( "sync EventMap \n" );

    uint8_t rStat =  rtNvmPutBytes( NVM_EVENT_MAP_START,
                                    (uint8_t *) eventMap.map,
                                    nodeMap.eventMapHwm * sizeof( LcsEventMapEntry ));

    if ( rStat == ALL_OK ) {

        uint32_t ofs = NVM_NODE_MAP_START + offsetof( LcsNodeMap, eventMapHwm );
        rStat = rtNvmPutWord( ofs, nodeMap.eventMapHwm );
```

```
    }

    return ( rStat );
}

//-----------------------------------------------------------------------------------------------------------
// "getMemEmapEntry" returns the eventId and portId pair from the MEM event map. It is used by the console
// command interface and also the LCS message handler to obtain that data. The index starts at 0.
//
//-----------------------------------------------------------------------------------------------------------
uint8_t getMemEmapEntry( uint16_t index, uint16_t *evId, uint16_t *pId ) {

    if (( debugMask & DBG_CONFIG ) && ( debugMask & DBG_EVENTS )) {

        printf( "Get Emap Entry: %d\n", index );
    }

    if ( index <  nodeMap.eventMapHwm ) {

        *evId = eventMap.map[ index ].eventId;
        *pId  = eventMap.map[ index ].portId;
        return ( ALL_OK );
    }
    else return ( ERR_INVALID_EVENT_MAP_INDEX );
}

};
```

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

```
        }

        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 ) {

                    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" );
    }
}

//------------------------------------------------------------------------------------------------------------
// Routines to list contents of the various memory areas. Right now, we just dump out hex data. It would be
// nice to show formatted data. Perhaps one day...
//
//------------------------------------------------------------------------------------------------------------
void printSummary( ) {

    printf( "LCS Node: \"" );
    for ( uint8_t i = 0; i < MAX_NODE_NAME_SIZE; i++ ) {

        if ( nodeMap.name[ i ] != 0 ) printf( "%c", nodeMap.name[ i ] );
    }

    printf( "\"\n" );
    printf( "LCS Library Version: %d.%d\n", nodeMap.nodeSwVersion >> 8, nodeMap.nodeSwVersion & 0xFF );
}

void dumpMemNodeMap( ) {

    printf( "MEM Node Map: \n\n" );
    dumpMemData((uint16_t *) &nodeMap, sizeof( LcsNodeMap ), 8, true);
    printf( "\n" );
}

void dumpMemCdcMap( ) {

    printf( "MEM CDC Map: \n\n" );
    // dumpMemData((uint16_t *) &nodeMap, sizeof( LcsNodeMap ));
    printf( "\n" );
}

void dumpMemPortMap( ) {

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

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

        printf( "Port %d:\n", i + 1 );
        dumpMemData((uint16_t *) &portMap.map[ i ], sizeof( LcsPortMapEntry ), 8, true );
        printf( "\n" );
    }
}

void dumpMemNodeData( ) {

    printf( "MEM Node Data: \n\n" );

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

        printf( "Port %d:\n", i );
        dumpMemData((uint16_t *) &nodeData.map[ i ], MAX_ATTR_MAP_ENTRIES * sizeof( uint16_t ));
        printf( "\n" );
    }
}

void dumpMemEventMap( ) {

    printf( "MEM Event Map (Size: %d, Hwm: %d): \n\n", nodeMap.eventMapEntries, nodeMap.eventMapHwm );
    dumpMemData((uint16_t *) &eventMap, sizeof( LcsEventMap ));
    printf( "\n" );
}

void dumpMemPendingReqMap( ) {

    printf( "MEM Pending Req Map: (Size: %d, Hwm: %d) \n\n", nodeMap.pendingMapEntries, nodeMap.pendingMapHwm );
    dumpMemData((uint16_t *) &pendingReqMap, sizeof( LcsPendingReqMap ));
    printf( "\n" );
}

void dumpMemCallbackMap( ) {

    printf( "MEM Callback Map: \n\n" );
    dumpMemData((uint16_t *) &callbackMap, sizeof( LcsCallbackMap ));
    printf( "\n" );
```

```
}

void dumpMemTaskMap( ) {

    printf( "MEM Task Map: (Size: %d, Hwm: %d) \n\n", nodeMap.taskMapEntries, nodeMap.taskMapHwm );
    dumpMemData((uint16_t *) &taskMap, sizeof( LcsTaskMap ));
    printf( "\n" );
}

void dumpMemDrvFuncMap( ) {

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

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

        LcsDrvFuncEntry *entry = &drvFuncMap.map[ i ];
        printf( "%d: Type: %d, Func: %p\n", i, entry -> drvType, entry -> drvFunc );
    }

     printf( "\n" );
}

void dumpMemDrvMap( ) {

    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 );

        dumpMemData(( uint16_t*) &drvMap.map[ i ].extBoard, sizeof( LcsDrvBoardDesc ), 8, true );
        printf( "\n" );
    }

     printf( "\n" );
}

void dumpMemRuntimeArea( ) {

    printf( "MEM Area Dump: \n\n" );
    dumpMemNodeMap( );
    dumpMemCdcMap( );
    dumpMemPortMap( );
    dumpMemEventMap( );
    dumpMemPendingReqMap( );
    dumpMemTaskMap( );
    dumpMemCallbackMap( );
    dumpMemDrvFuncMap( );
    dumpMemDrvMap( );
    printf( "\n" );
}

//----------------------------------------------------------------------------------------------------------
// Routines to list contents of the various NVM areas. Right now, we just dump out hex data. It would be
// nice to show formatted data. Perhaps one day...
//
//----------------------------------------------------------------------------------------------------------
void dumpNvmNodeMap( ) {

    printf( "NVM Node Map Dump: \n\n" );
    dumpNvmData( NVM_NODE_MAP_START, sizeof( LcsNodeMap ), 8, true );
    printf( "\n" );
}

void dumpNvmCdcMap( ) {

    printf( "MEM CDC Map Dump: \n\n" );
    dumpNvmData( NVM_CDC_MAP_START, sizeof( CDC::CdcConfigDesc ));
    printf( "\n" );
}

void dumpNvmPortMap( ) {

    printf( "NVM Port Map Dump: \n\n" );

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

        uint32_t ofs = NVM_PORT_MAP_START + ( i * sizeof( LcsPortMapEntry ));

        printf( "Port %d, NVM ofs: 0x%04x \n", i + 1, ofs );
        dumpNvmData( ofs, sizeof( LcsPortMapEntry ), 8, true );
        printf( "\n" );
    }

    printf( "\n" );
}

void dumpNvmNodeData( ) {

    printf( "NVM Port Map Dump: \n\n" );

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

        uint32_t ofs = NVM_NODE_DATA_START + ( i * MAX_ATTR_MAP_ENTRIES  * sizeof( uint16_t ));

        printf( "Node data block: %d, NVM ofs: 0x%04x \n", i, ofs );
```

```
        dumpNvmData( ofs, MAX_ATTR_MAP_ENTRIES  * sizeof( uint16_t ));
        printf( "\n" );
    }

    printf( "\n" );
}

void dumpNvmEventMap( ) {

    printf( "NVM Node Event Dump: \n\n" );
    dumpNvmData( NVM_EVENT_MAP_START, sizeof( LcsEventMap ));
    printf( "\n" );
}

void dumpNvmRuntimeArea( ) {

    printf( "NVM Runtime Area Dump: \n\n" );
    dumpNvmData( 0, NVM_RUNTIME_AREA_SIZE , 8, true );
    printf( "\n" );
}

void dumpNvmDrvData( uint16_t boardId ) {

    if ( boardId >= MAX_EXT_BOARD_MAP_ENTRIES ) {

        printf( "Invalid board ID\n" );
        return;
    }

    if ( drvMap.map[ boardId ].flags & BF_EXT_BOARD_PRESENT ) {

        printf( "NVM Driver Data( board: %d ): \n\n", boardId );
        dumpExtNvmData( boardId, 0, sizeof( LcsDrvBoardDesc ));
        printf( "\n" );
    }
    else printf( "No board found \n" );
}

void dumpNvmUserArea( ) {

    printf( "NVM Area Dump: \n\n" );
    dumpNvmData( NVM_USER_MAP_START, usrNvmGetSize( ), 8, true );
    printf( "\n" );
}

//----------------------------------------------------------------------------------------------------------
// Print memory structures in a formatted way. Note that not all memory structures are printed. Some of the
// maps contain dynamic data, which changed rapidly. There is no point in showing that kind of data.
//
//----------------------------------------------------------------------------------------------------------
void printMemNodeMap( ) {

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

/*
    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;

    uint16_t        nodeState                       = NS_NIL;
    uint16_t        nodeOptions                     = 0;
    uint16_t        nodeFlags                       = 0;
    uint16_t        nodeId                          = NIL_NODE_ID;
    uint32_t        nodeUID                         = 0L;
    uint16_t        nodeType                        = NIL_NODE_TYPE;
    uint16_t        nodeSwVersion                   = 0;
    uint16_t        nodeSwPatchLevel                = 0;
    uint16_t        nodeRestartCnt                  = 0;
    uint32_t        nodeSystemTime                  = 0;
    uint16_t        nodeMapSize                     = sizeof( LcsNodeMap );
    char            name[ MAX_NODE_NAME_SIZE ]      = { 0 };

    uint16_t        nvmNodeMapOfs                   = NVM_NODE_MAP_START;
    uint16_t        nvmCdcMapOfs                    = NVM_CDC_MAP_START;
    uint16_t        nvmPortMapOfs                   = NVM_PORT_MAP_START;
    uint16_t        nvmNodeDataOfs                  = NVM_NODE_DATA_START;
    uint16_t        nvmEventMapOfs                  = NVM_EVENT_MAP_START;
    uint16_t        nvmUserMapOfs                   = NVM_USER_MAP_START;
    uint32_t        nvmMemSize                      = NVM_RUNTIME_AREA_SIZE;

    uint16_t        portMapEntries                  = MAX_PORT_MAP_ENTRIES;
    uint16_t        portMapHwm                      = 0;

    uint16_t        eventMapEntries                 = MAX_EVENT_MAP_ENTRIES;
    uint16_t        eventMapHwm                     = 0;

    uint16_t        taskMapEntries                  = MAX_TASK_MAP_ENTRIES;
    uint16_t        taskMapHwm                      = 0;

    uint16_t        pendingMapEntries               = MAX_PENDING_REQ_MAP_ENTRIES;
    uint16_t        pendingMapHwm                   = 0;

    uint16_t        drvFuncMapEntries               = MAX_DRV_TYPES;
    uint16_t        drvFuncMapHwm                   = 0;

```

```
    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;
```

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

        uint8_t msg[ 8 ] = { LCS_OP_CFG };
        handleMsgLcsMgt( msg );
    }
    else {

        uint8_t ret = sendCfg( tmpNpId );
        if ( ret != ALL_OK ) errorStatusMsg((char *) "Remote Node send error", ret );
    }
}

//----------------------------------------------------------------------------------------------------------------
// "o" switches the nodes to OPS mode. For a local node command, we construct the LCS_OP_OPS message payload
// data and invoke the msg handler for switching the node mode. For any other node, we will just send a LCS
// message.
//
//    o [ npId ]
//
//----------------------------------------------------------------------------------------------------------------
void switchToOperationsCommand( char *s ) {

    int npId = NIL_NODE_ID;

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

    uint16_t tmpNpId = (uint16_t) npId;

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

        uint8_t msg[ 8 ] = { LCS_OP_OPS };
        handleMsgLcsMgt( msg );
    }
    else {

        uint8_t ret = sendOps( tmpNpId );
        if ( ret != ALL_OK ) errorStatusMsg((char *) "Remote Node send error", ret );
    }
}

//----------------------------------------------------------------------------------------------------------------
// "a" adds an eventId / portId to the event map. If the portId is omitted, every port of the node will be
// registered for the event. For a non-local npId we will send a message.
//
//      a npId eventId [ portId ]
//
//      npId      - the node and port Id for which the event is added.
//      eventId   - the eventId.
//      portId    - the port number.
//
//----------------------------------------------------------------------------------------------------------------
void enterEventCommand( char *s ) {

    int  npId      = NIL_NODE_ID;
    int  eventId   = NIL_EVENT_ID;
    int  portId    = NIL_PORT_ID;

    if ( sscanf( s, "%i %i %i ", &npId, &eventId, &portId ) < 2 ) return( errorArgList( ));

    uint16_t tmpNpId    = (uint16_t) npId;
    uint16_t tmpEvent   = (uint16_t) eventId;
    uint16_t tmpPort    = (uint16_t) portId;

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

        uint8_t ret = nodeReq((uint16_t) tmpNpId, ITEM_ID_ADD_EVENT_MAP_ENTRY, &tmpEvent, &tmpPort );
       if ( ret != ALL_OK ) errorStatusMsg((char *) "Node enter event error", ret );
    }
    else {

        uint8_t ret = sendReqNode( nodeId( tmpNpId ), ITEM_ID_ADD_EVENT_MAP_ENTRY, tmpEvent, tmpPort );
        if ( ret != ALL_OK ) errorStatusMsg((char *) "Remote Node send error", ret );
    }
}

//----------------------------------------------------------------------------------------------------------------
// "r"  removes a eventId / portId combination from the event map. If the portId is omitted, all eventMap
// entries with the eventId are removed.
//
//      r npId eventId [ portId ]
//
//      npId      - the node and port Id for which the event is added.
//      eventId   - the eventId.
//      portId    - the port number.
//
//----------------------------------------------------------------------------------------------------------------
void removeEventCommand( char *s ) {

    int  npId      = NIL_NODE_ID;
    int  eventId   = NIL_EVENT_ID;
    int  portId    = NIL_PORT_ID;

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

    uint16_t tmpNpId    = (uint16_t) npId;
    uint16_t tmpEvent   = (uint16_t) eventId;
    uint16_t tmpPort    = (uint16_t) portId;

```

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

        int ret = nodeReq( tmpNpId, ITEM_ID_DEL_EVENT_MAP_ENTRY, &tmpEvent, &tmpPort );
        if ( ret != ALL_OK ) errorStatusMsg((char *) "Node remove event error", ret );
    }
    else {

        uint8_t ret = sendReqNode( tmpNpId, ITEM_ID_DEL_EVENT_MAP_ENTRY, tmpEvent, tmpPort );
        if ( ret != ALL_OK ) errorStatusMsg((char *) "Remote Node send error", ret );
    }
}

//-------------------------------------------------------------------------------------------------------------
// "f" searches the event map for the eventId / portId combination and returns the index if found. If the
// portId is omitted, the first event map entry with the matching eventId is returned. This is a local
// command and cannot be called from a remote node.
//
//      f eventId [ portId ]
//
//      eventId   - the eventId.
//      portId    - the port number.
//
//-------------------------------------------------------------------------------------------------------------
void findEventCommand( char *s ) {

    int  eventId   = NIL_EVENT_ID;
    int  portId    = NIL_PORT_ID;

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

    uint16_t tmpEvent   = (uint16_t) eventId;
    uint16_t tmpPort    = (uint16_t) portId;

    int ret = searchEvent( tmpEvent, tmpPort );
    printf( "Event map index: %d", ret );
}

//-------------------------------------------------------------------------------------------------------------
// "e" will send an event. We will broadcast a message and also simulates receiving an event on the local
// node. Sending to ourselves is also quite useful for debugging event callback handlers.
//
//    e mode npId eventId [ arg ]
//
//    mode      - 0 - ON, 1 - OFF, 2 - DATA
//    npId      - the sending node / port Id
//    eventId   - the event Id
//    arg       - optional data argument for the event.
//
//-------------------------------------------------------------------------------------------------------------
void sendEventCommand( char *s ) {

    uint8_t msg[ 8 ]    = { };
    int     npId        = NIL_NODE_ID;
    int     eventId     = NIL_EVENT_ID;
    int     mode        = 0;
    int     arg         = 0;
    int     len         = 0;
    uint8_t ret         = ALL_OK;

    len = sscanf( s, "%i %i %i %i", &mode, &npId, &eventId, &arg );

    uint16_t tmpNpId    = (uint16_t) npId;
    uint16_t tmpEvent   = (uint16_t) eventId;
    uint16_t tmpArg     = (uint16_t) arg;

    if ( len < 3 ) return( errorArgList( ));

    msg[ 0 ] = 0;
    msg[ 1 ] = highByte( tmpNpId );
    msg[ 2 ] = lowByte( tmpNpId );
    msg[ 3 ] = highByte( tmpEvent );
    msg[ 4 ] = lowByte( tmpEvent );
    msg[ 5 ] = highByte( tmpArg );
    msg[ 6 ] = lowByte( tmpArg );
    msg[ 7 ] = 0;

    if ( mode == 0 ) {

        msg[ 0 ] = LCS_OP_EVT_ON;
        ret = sendEventOn( tmpNpId, tmpEvent );
    }
    else if ( mode == 0 ) {

        msg[ 0 ] = LCS_OP_EVT_OFF;
        ret = sendEventOn( tmpNpId, tmpEvent );
    }
    else if ( mode == 2 ) {

        msg[ 0 ] = LCS_OP_EVT;
        ret = sendEvent( tmpNpId, tmpEvent, tmpArg );
    }

    if ( ret != ALL_OK ) errorStatusMsg((char *) "Send event error", ret );
}

//-------------------------------------------------------------------------------------------------------------
// "g" handles the node/port attribute query command. If the node is our node, we call the local access
// routines. Otherwise we send a message.
//
```

```
//    <!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( ));
```

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

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

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

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

//---------------------------------------------------------------------------------------------------------------
// "B" broadcasts a LCS message. Mainly used for debugging purposes. Although most commands in the LCS
// console interface can also send messages to other nodes, not all messages are covered. This command sends
// any kind of message, even undefined ones.
//
//    B byte1 [ byte2 ... byte8 ]
//
//    byte1 .. byte8   - the packet data in hexadecimal
//
//---------------------------------------------------------------------------------------------------------------
void broadcastLcsMsgCommand( char *s ) {

    int     inBuf[ 8 ]  = { 0 };
    uint8_t b[ 8 ]      = { 0 };
    uint8_t nBytes  = sscanf( s, "%i %i %i %i %i %i %i %i",
                              inBuf, inBuf + 1, inBuf + 2, inBuf + 3, inBuf + 4, inBuf + 5, inBuf + 6, inBuf + 7 );

    if ( nBytes >= 1 && nBytes <= 8 ) {

        for ( int i = 0; i < 8; i++ ) b[ i ] == (uint8_t) inBuf[ i ];

        uint8_t ret = msgBus -> sendLcsMsg( b );
        if ( ret != ALL_OK ) errorStatusMsg((char *) "Can Bus send error", ret );
    }
    else errorArgList( );
}

//---------------------------------------------------------------------------------------------------------------
// "G" sends a GET request to a driver. The commands will typically work on the MEM image of the driver data.
// We will use the same idea of item ranges for MEM and NVM, except that the NVM range will work only if the
// extension board is write-enabled.
//
//    G board item
//
//    board - the extension board the driver handles.
//    item  - the driver specific item which is the requested operation.
//
//---------------------------------------------------------------------------------------------------------------
void drvGetCommand( char *s ) {

    int  boardId    = 0;
    int  item       = 0;
    int  arg        = 0;
    int  ret        = ALL_OK;

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

    uint16_t tmpBoard   = (uint8_t) boardId;
    uint8_t  tmpItem    = (uint8_t)  item;
    uint16_t tmpArg     = (uint16_t) arg;

    ret = drvGet( tmpBoard, tmpItem, &tmpArg );

    if ( ret != ALL_OK ) errorStatusMsg((char *) "Driver GET error", ret );
    else printf( "Board: %d, item: %d, arg: 0x%x\n", tmpBoard, tmpItem, tmpArg );
}

//---------------------------------------------------------------------------------------------------------------
// "P" sends a PUT request to a driver. The commands will typically work on the MEM image of the driver data.
// We will use the same idea of item ranges for MEM and NVM, except that the NVM range will work only if the
// extension board is write-enabled.
//
//    P board item arg
//
//    board - the extension board the driver handles.
//    item  - the driver specific item which is the requested operation.
//    arg   - the data argument to the driver.
//
//---------------------------------------------------------------------------------------------------------------
void drvPutCommand( char *s ) {

    int  boardId    = 0;
    int  item       = 0;
    int  val        = 0;
    int  ret        = ALL_OK;

    if ( sscanf( s, "%i %i %i", &boardId, &item, &val ) < 3 ) return( errorArgList( ));

    uint16_t tmpBoard   = (uint8_t) boardId;
    uint8_t  tmpItem    = (uint8_t)  item;
```

```
    uint16_t tmpVal     = (uint16_t) val;

    ret = drvPut( tmpBoard, tmpItem, tmpVal );

    if ( ret != ALL_OK ) errorStatusMsg((char *) "Driver PUT error", ret );
    else printf( "Board: %d, item: %d, arg: 0x%x\n", tmpBoard, tmpItem, tmpVal );
}

//----------------------------------------------------------------------------------------------------------
// "R" sends a REQ request to a driver.
//
//    R board item arg1 [ arg 2 ]
//
//    board - the extension board the driver handles.
//    item  - the driver specific item which is the requested operation.
//    arg1  - the first argument to the driver.
//    arg2  - the optional second argument to the driver and also output from the driver.
//
//----------------------------------------------------------------------------------------------------------
void drvReqCommand( char *s ) {

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

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

    uint16_t tmpBoard   = (uint8_t) boardId;
    uint8_t  tmpItem    = (uint8_t)  item;
    uint16_t tmpArg1     = (uint16_t) arg1;
    uint16_t tmpArg2     = (uint16_t) arg2;

    ret = drvReq( tmpBoard, tmpItem, &tmpArg1, &tmpArg2 );

    if ( ret != ALL_OK ) errorStatusMsg((char *) "Driver REQ error", ret );
    else printf( "Board: %d, item: %d, arg1: 0x%x, arg2: 0x%x\n", tmpBoard, tmpItem, tmpArg1, tmpArg2 );
}

//----------------------------------------------------------------------------------------------------------
// "s" lists status information. The level argument specifies the what and the detail level.
//
//    s [ level ]
//
//    returns:  NONE.
//
//----------------------------------------------------------------------------------------------------------
void listStatusCommand( char *s ) {

    int level = 0;

    if ( sscanf( s, " %i", &level ) > 0 ) {

        switch ( level ) {

            case 0:     printSummary( );             break;

            case 1:     dumpMemNodeMap( );           break;
            case 2:     dumpNvmCdcMap( );            break;
            case 3:     dumpMemPortMap( );           break;
            case 4:     dumpMemNodeData( );          break;
            case 5:     dumpMemEventMap( );          break;
            case 6:     dumpMemPendingReqMap( );     break;
            case 7:     dumpMemTaskMap( );           break;
            case 8:     dumpMemCallbackMap( );       break;
            case 9:     dumpMemDrvFuncMap( );        break;
            case 10:    dumpMemDrvMap( );            break;
            case 11:    dumpMemRuntimeArea( );       break;

            case 21:    dumpNvmNodeMap( );           break;
            case 22:    dumpNvmCdcMap( );            break;
            case 23:    dumpNvmPortMap( );           break;
            case 24:    dumpNvmNodeData( );          break;
            case 25:    dumpNvmEventMap( );          break;
            case 26:    dumpNvmRuntimeArea( );       break;

            case 31:    printMemNodeMap( );          break;
            case 32:    printMemCdcMap( );           break;
            case 33:    printMemPortMap( );          break;
            case 35:    printMemEventMap( );         break;
            case 38:    printMemDrvMap( );           break;

            case 40:    dumpNvmDrvData( 0 );         break;
            case 41:    dumpNvmDrvData( 1 );         break;
            case 42:    dumpNvmDrvData( 2 );         break;
            case 43:    dumpNvmDrvData( 3 );         break;

            case 50:    listDevicesI2C( );           break;

            default: printf( "Unknown help option, use '?' for help\n" );
        }
    }
    else printSummary( );
}

//----------------------------------------------------------------------------------------------------------
// "?" lists core library help information. We just list the available commands and a short description.
//
```

```
//    ?
//
//------------------------------------------------------------------------------------------------------------
void listCoreLibHelpCommand( ) {

    printf( "\nCommands: \n" );
    printf( "c [ npId ] - enter config mode\n" );
    printf( "o [ npId ] - enter operations mode\n" );

    printf( "a npId eventId [ npId ] - add an event to the event tab\n" );
    printf( "d npId eventId [ npId ] - remove an event from the event tab\n" );
    printf( "e npId eventId mode [ arg ] - simulate sending an event ( mode: 0 - ON, 1 - OFF, 2 - EVT\n" );
    printf( "f eventId [ npId ] - search an event on the event tab\n" );

    printf( "g npId item - gets a node attribute\n" );
    printf( "p npId item val1 [ val2 ] - puts a node attribute\n" );
    printf( "r npId item val1 [ val2 ] - request a node function\n" );

    printf( "B byte1 [ byte2 ... byte8 ] - broadcast a raw LCS message\n" );

    printf( "G board item - send a GET request to an extension board n\n" );
    printf( "P board item arg - send a PUT request to an extension board n\n" );
    printf( "R board item [ arg1 [ arg2 ]] - send a REQ request to an extension board n\n" );

    printf( "s [ level ] - list status, default is summary\n" );
    printf( "              " " -   0  - summary\n" );
    printf( "              " " -   1  - MEM Node Map\n" );
    printf( "              " " -   2  - MEM CDC Map\n" );
    printf( "              " " -   3  - MEM Port Map\n" );
    printf( "              " " -   4  - MEM Node Data\n" );
    printf( "              " " -   5  - MEM Event Map\n" );
    printf( "              " " -   6  - MEM Pending Request Map\n" );
    printf( "              " " -   7  - MEM Task Map\n" );
    printf( "              " " -   8  - MEM Callback Map\n" );
    printf( "              " " -   9  - MEM Driver Function Map\n" );
    printf( "              " " -  10  - MEM Driver Map\n" );
    printf( "              " " -  11  - MEM Runtime Area\n" );

    printf( "              " " -  21  - NVM Node Map\n" );
    printf( "              " " -  22  - NVM CDC Map\n" );
    printf( "              " " -  23  - NVM Port Map\n" );
    printf( "              " " -  24  - NVM Node Data\n" );
    printf( "              " " -  25  - NVM Event Map\n" );
    printf( "              " " -  26  - NVM Runtime Area\n" );

    printf( "              " " -  31  - MEM Node Map formatted\n" );
    printf( "              " " -  32  - MEM CDC Map formatted\n" );
    printf( "              " " -  33  - NVM Port Map formatted\n" );
    printf( "              " " -  35  - MEM Event Map formatted\n" );
    printf( "              " " -  38  - MEM Driver Map formatted\n" );

    printf( "              " " -  40  - NVM Extension Board 0\n" );
    printf( "              " " -  41  - NVM Extension Board 1\n" );
    printf( "              " " -  42  - NVM Extension Board 2\n" );
    printf( "              " " -  43  - NVM Extension Board 3\n" );

    printf( "              " " -  50  - I2C Devices\n" );
}

//------------------------------------------------------------------------------------------------------------
// "setupSerialCommand" initializes the serial interface. We use the PICO USB as console IO. The CDC lib
// contains functions for reading and writing to the console.
//
//------------------------------------------------------------------------------------------------------------
uint8_t setupSerialCommand( ) {

    return ( CDC::configureConsoleIO( ));
}

//------------------------------------------------------------------------------------------------------------
// "handleSerialCommand" reads characters from the console. The command interpreter is a simple character
// based input. Note that this routine is called as part of the runtime loop. Consequently, it cannot not
// block for IO. The interface is designed in a way that it assembles the character input when there are
// characters until a carriage return is received. The first character is the command. If it is not a
// command we know and there is a callback, the callback gets his chance to handle the input string.
// Since we are pretty basic on a character by character basis, we also add a bit of luxury and echo back
// the  what was typed in as well as process the backspace character.
//
//------------------------------------------------------------------------------------------------------------
uint8_t handleSerialCommand( ) {

    char c;

    while (( c = CDC::getConsoleChar( )) > 0 ) {

        switch( c ) {

            case '\r': {

                printf( "\n" );

                if ( strlen( commandBuf) > 0 ) {

                    switch ( commandBuf[ 0 ] ) {

                        case 'C': switchToConfigCommand( commandBuf + 1 );        break;
                        case 'O': switchToOperationsCommand( commandBuf + 1 );    break;

                        case 'a': enterEventCommand( commandBuf + 1 );          break;
```

```
                        case 'd': removeEventCommand( commandBuf + 1 );          break;
                        case 'f': findEventCommand( commandBuf + 1 );            break;
                        case 'e': sendEventCommand( commandBuf + 1 );            break;

                        case 'g': getNodeCommand( commandBuf + 1 );              break;
                        case 'p': putNodeCommand( commandBuf + 1 );              break;
                        case 'r': reqNodeCommand( commandBuf + 1 );              break;

                        case 'B': broadcastLcsMsgCommand( commandBuf + 1 );      break;

                        case 'G': drvGetCommand( commandBuf + 1 );               break;
                        case 'P': drvPutCommand( commandBuf + 1 );               break;
                        case 'R': drvReqCommand( commandBuf + 1 );               break;

                        case 's': listStatusCommand( commandBuf + 1 );           break;
                        case '?': listCoreLibHelpCommand( );                     break;

                        default: {

                            if ( callbackMap.cmdLineCallback != nullptr ) {

                                    callbackMap.cmdLineCallback( commandBuf );
                                }
                            else printf( "<Unknown command, use '?' for help>" );
                        }
                    }
                }

                commandBuf[ 0 ] = '\0';
                printf( "->");

            } break;

            case '\b': {

                printf( "\b \b" );
                if ( strlen( commandBuf ) > 0 ) commandBuf[ strlen( commandBuf ) - 1 ] = '\0';

            } break;

            default: {

                printf( "%c", c );
                if ( strlen( commandBuf ) < MAX_COMMAND_LINE_SIZE ) strncat( commandBuf, &c, 1 );
            }
        }
    }

    return( ALL_OK );
}

}; // 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 {

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

```
//
//-----------------------------------------------------------------------------------------------------
uint8_t drvInit( ) {

    return ( ALL_OK );
}

//-----------------------------------------------------------------------------------------------------
// "drvGet" returns a value from the driver data array.
//
//-----------------------------------------------------------------------------------------------------
uint8_t drvGet( uint8_t boardId, uint8_t item, uint16_t *arg ) {

    if ( boardId >= MAX_EXT_BOARD_MAP_ENTRIES )                return ( ERR_INVALID_BOARD_ID );
    if ( ! drvMap.map[ boardId ].flags & BF_EXT_BOARD_PRESENT ) return( ERR_INVALID_BOARD_ID );

    if ( isInRangeU( item, IR_ATTR_MEM_RANGE_START, IR_ATTR_MEM_RANGE_END )) {

        *arg = drvMap.map[ boardId ].extBoard.driverData[ item - IR_ATTR_MEM_RANGE_START ];
        return( ALL_OK );
    }
    else if ( isInRangeU( item, IR_ATTR_NVM_RANGE_START, IR_ATTR_NVM_RANGE_END )) {

        uint32_t ofs =  offsetof( LcsDrvBoardDesc, driverData ) +
                        (( item - IR_ATTR_NVM_RANGE_START ) * sizeof( uint16_t ));

       if ( extNvmGetWord( boardId, ofs, arg ) != ALL_OK ) return( ERR_DRV_GET_ERR );
       return( ALL_OK );
    }
    else if ( item == ITEM_ID_BOARD_VERSION ) {

        *arg = drvMap.map[ boardId ].extBoard.head.boardVersion;
        return( ALL_OK );
    }
    else if ( item == ITEM_ID_TYPE ) {

        *arg = drvMap.map[ boardId ].extBoard.head.boardType;
        return( ALL_OK );
    }
    else return( ERR_INVALID_ITEM_ID );
}

//-----------------------------------------------------------------------------------------------------
// "drvPut" sets a value in the driver data array. Note that this is during normal operations only the MEM
// portion. The NVM chip on the extension board is write disabled after initial configuration. When the
// jumper on the board is taken out, writing to the NVM is enabled. The PUT and GET routines can be called
// for a board that has no driver associated yet. This way, for example, the driver type and other initial
// data can be set.
//
//
// ??? should we restart the extension board after setting the driver type ?
//-----------------------------------------------------------------------------------------------------
uint8_t drvPut(uint8_t boardId, uint8_t item, uint16_t arg ) {

    if ( boardId >= MAX_EXT_BOARD_MAP_ENTRIES )                return( ERR_INVALID_BOARD_ID );
    if ( ! drvMap.map[ boardId ].flags & BF_EXT_BOARD_PRESENT ) return( ERR_INVALID_BOARD_ID );

     if ( isInRangeU( item, IR_ATTR_MEM_RANGE_START, IR_ATTR_MEM_RANGE_END )) {

        drvMap.map[ boardId ].extBoard.driverData[ item - IR_ATTR_MEM_RANGE_START ] = arg;
        return( ALL_OK );
    }
    else if ( isInRangeU( item, IR_ATTR_NVM_RANGE_START, IR_ATTR_NVM_RANGE_END )) {

        uint32_t ofs =  offsetof( LcsDrvBoardDesc, driverData ) +
                        (( item - IR_ATTR_NVM_RANGE_START ) * sizeof( uint16_t ));

       if ( extNvmPutWord( boardId, ofs, arg ) != ALL_OK ) return( ERR_DRV_PUT_ERR );
       return( ALL_OK );
    }
    else if ( item == ITEM_ID_BOARD_VERSION ) {

        uint8_t rStat = ALL_OK;

        rStat = extNvmPutWord( boardId, offsetof( LcsDrvBoardDesc, head.boardVersion ), arg );
        if ( rStat == ALL_OK ) drvMap.map[ boardId ].extBoard.head.boardVersion = arg;
        return( rStat );
    }
    else if ( item == ITEM_ID_TYPE ) {

        uint8_t rStat = ALL_OK;

        rStat = extNvmPutWord( boardId, offsetof( LcsDrvBoardDesc, head.boardType ), arg );
        if ( rStat == ALL_OK ) drvMap.map[ boardId ].extBoard.head.boardType = arg;
        return( rStat );
    }
    else return( ERR_INVALID_ITEM_ID );
}

//-----------------------------------------------------------------------------------------------------
// "drvReq" is the entry point to an extension board. For each extension board type there is driver function.
// This function is called when we access that extension board. Note that the REQ call will only work when
// there is a board with a driver associated. There is however the case that the header area is a new area
// or an invalid area. We have a board detected bit could not setup the driver for it. The "ITEM_ID_FORMAT"
// item is used to setup the extension board NVM. It works without checking for a valid driver.
//
// The PUT and GET routines can be called for a board that has no driver associated yet. This way, for
// example, the driver type and other initial data can be set.
//
```

```
//-----------------------------------------------------------------------------------------------------------
uint8_t drvReq( uint8_t boardId, uint8_t item, uint16_t *arg1, uint16_t *arg2 ) {

    if ( boardId >= MAX_EXT_BOARD_MAP_ENTRIES )                 return ( ERR_INVALID_BOARD_ID );
    if ( ! drvMap.map[ boardId ].flags & BF_EXT_BOARD_PRESENT ) return( ERR_INVALID_BOARD_ID );

    if ( item == ITEM_ID_FORMAT ) {

            LcsDrvEntry entry;

            entry.flags = BF_EXT_BOARD_PRESENT | BF_EXT_BOARD_VALID;

            uint8_t rStat = buildDrvBoardDescArea( boardId );
            if ( rStat == ALL_OK ) drvMap.map[ boardId ] = entry;

            return( rStat );
    }
    else {

        if ( drvMap.map[ boardId ].drvFunc != nullptr ) {

            return( drvMap.map[ boardId ].drvFunc( boardId - 1, item, arg1, arg2 ));
        }
        else return( ERR_EXT_BOARD_NOT_VALID );
    }
}

} // 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 ) {
```

```
  callbackMap.dccMsgCallback = functionId;
}

void registerCmdCallback( LcsCmdCallback functionId ) {

  callbackMap.cmdLineCallback = functionId;
}

void registerEventCallback( LcsEventCallback functionId ) {

  callbackMap.eventCallback = functionId;
}

void registerInitCallback( LcsInitCallback functionId ) {

    callbackMap.initCallback = functionId;
}

void registerResetCallback( LcsInitCallback functionId ) {

    callbackMap.resetCallback = functionId;
}

void registerPfailCallback( LcsInitCallback functionId ) {

    callbackMap.pfailCallback = functionId;
}

void registerReqCallback( LcsReqCallback functionId ) {

    callbackMap.reqCallback = functionId;
}

void registerRepCallback( LcsRepCallback functionId ) {

    callbackMap.repCallback = functionId;
}

//----------------------------------------------------------------------------------------------------------
// The core library features a very simple periodic task system. This routine adds a task callback to the
// pTaskMap. We only add entries, never remove them. A high water mark is used to record the highest entry
// used, so that processing will not run through empty entries.
//
// ??? perhaps this needs to be reworked to use HW driven timers.
//----------------------------------------------------------------------------------------------------------
uint8_t registerTaskCallback( LcsTaskCallback task, uint32_t interval ) {

    if ( nodeMap.taskMapHwm < MAX_TASK_MAP_ENTRIES ) {

        taskMap.map[ nodeMap.taskMapHwm ].task       = task;
        taskMap.map[ nodeMap.taskMapHwm ].interval   = interval;
        taskMap.map[ nodeMap.taskMapHwm ].timeStamp  = CDC::getMillis( );
        nodeMap.taskMapHwm ++;
        return ( ALL_OK );

    } else return ( ERR_TASK_MAP_SIZE_EXCEEDED );
}

//-----------------------------------------------------------------------------------------------------------
// "handleNodePortEvents" will be called for processing inbound port events on each loop iteration. Note that
// it does not matter where the events came from, i.e. whether another node sends an event or the event was
// created by a firmware call on this node. The event callback can be delayed with a timer value.
//
//-----------------------------------------------------------------------------------------------------------
void handleNodePortEvents( ) {

    if ( callbackMap.eventCallback == nullptr ) return;

    uint32_t ts = CDC::getMillis( );

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

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

        if (( pPtr -> flags & PF_PORT_ENABLED                 ) &&
            ( pPtr -> flags & PF_PORT_EVENT_HANDLING_ENABLED  ) &&
            ( pPtr -> flags & PF_EVENT_PENDING                )) {

            if ( ts > pPtr -> eventTimeStamp ) {

                callbackMap.eventCallback(  pPtr -> eventNodeId,
                                            pPtr -> eventId,
                                            pPtr -> eventAction,
                                            pPtr -> eventValue );
            }

            pPtr -> flags &= ~ PF_EVENT_PENDING;
        }
    }
}

//-----------------------------------------------------------------------------------------------------------
// "handlePeriodicTasks" is called from the core library main processing loop. The idea is that there is a
// lot of periodic processing that needs to be one by any firmware implementation. Instead of the firmware
// developer writing its own handler, there is a crude method that just samples the timestamps and interval
// and triggers the callback then the interval is reached. Note that this is not very accurate from a timing
// perspective but will do for simple periodic processing.
//
```

```
//----------------------------------------------------------------------------------------------------------
void handlePeriodicTasks( ) {

    uint32_t ts = CDC::getMillis( );

    for ( int i = 0; i < nodeMap.taskMapHwm; i++ ) {

        LcsPTaskMapEntry *thisEntry = &taskMap.map[ i ];

        if ( ts > thisEntry -> timeStamp  ) {

            if ( thisEntry -> task != nullptr ) thisEntry -> task( );
            thisEntry -> timeStamp = ts + thisEntry -> interval;
        }
    }
}

//----------------------------------------------------------------------------------------------------------
// "handleMsgRepNid" handles the message that the configuring node sends to our node in response to a nodeId
// setup request. If the UID matches, the message is for our node and we update our nodeId accordingly in
// MEM and NVM. The next node state is OPERATE.
//
//----------------------------------------------------------------------------------------------------------
void handleMsgRepNid( uint8_t *msg ) {

    uint16_t nodeId   = ( msg[1] << 8 ) + msg[2];
    uint32_t nodeUID  = ((uint32_t) msg[3] << 24 ) +
                        ((uint32_t) msg[4] << 16 ) +
                        ((uint32_t) msg[5] << 8 ) +
                        msg[6];

    if ( nodeUID == nodeMap.nodeUID ) {

        if ( nodeMap.nodeId != nodeId ) {

            nodeMap.nodeId = nodeId;
            uint8_t rStat = rtNvmPutWord( NVM_NODE_MAP_START + offsetof( LcsNodeMap, nodeId ), nodeId );
        }

        nodeMap.nodeState = NS_OPERATE;
    }
}

//----------------------------------------------------------------------------------------------------------
// LCS management deals with messages concerning the general LCS management. If there is a callback defined
// it will be invoked. Then the node state is changed accordingly. Most updates are just to the MEM nodeMap.
// In addition, the READY and ACTIVITY LEDs are set.
//
//----------------------------------------------------------------------------------------------------------
void handleMsgLcsMgt( uint8_t *msg ) {

    switch ( msg[ 0 ] ) {

        case LCS_OP_OPS: {

            nodeMap.nodeState = NS_OPERATE;
            if ( callbackMap.lcsMsgCallback != nullptr ) callbackMap.lcsMsgCallback( msg );

        } break;

        case LCS_OP_CFG: {

            nodeMap.nodeState = NS_CONFIG;
            if ( callbackMap.lcsMsgCallback != nullptr ) callbackMap.lcsMsgCallback( msg );

        } break;

        case LCS_OP_BON: {

            CDC::writeDio( cdcMap.cfg.READY_LED_PIN, true );
            nodeMap.nodeState = NS_OPERATE;
            if ( callbackMap.lcsMsgCallback != nullptr ) callbackMap.lcsMsgCallback( msg );

        } break;

        case LCS_OP_BOF: {

            CDC::writeDio( cdcMap.cfg.READY_LED_PIN, false );
            nodeMap.nodeState = NS_HALTED;
            if ( callbackMap.lcsMsgCallback != nullptr ) callbackMap.lcsMsgCallback( msg );

        } break;

        case LCS_OP_NCOL: {

            CDC::writeDio( cdcMap.cfg.READY_LED_PIN, false );
            CDC::writeDio( cdcMap.cfg.ACTIVE_LED_PIN, true );
            nodeMap.nodeState = NS_COLLISION;
            if ( callbackMap.lcsMsgCallback != nullptr ) callbackMap.lcsMsgCallback( msg );

        } break;

        case LCS_OP_RESET: {

            uint16_t npId = (( msg[1] << 8 ) + msg[2] );
            uint8_t rStat = resetNode( npId );

            if ( rStat == ALL_OK ) LCS::sendAck( npId );
            else                   LCS::sendErr( npId, rStat, 0, 0 );
```

```

        } break;

        case LCS_OP_SET_NID: {

            uint16_t nodeId   = ( msg[1] << 8 ) + msg[2];
            uint32_t nodeUID  = ((uint32_t) msg[3] << 24 ) + ((uint32_t) msg[4] << 16 ) +
                                ((uint32_t) msg[5] << 8 ) + msg[6];

            if ( nodeUID == nodeMap.nodeUID ) {

                if ( nodeMap.nodeState == NS_CONFIG ) {

                    if ( nodeId != nodeMap.nodeId ) nodeMap.nodeId = nodeId;
                    uint8_t rStat = rtNvmPutWord( NVM_NODE_MAP_START + offsetof( LcsNodeMap, nodeId ), nodeId );

                    sendAck( nodeId );
                }
                else sendErr( nodeId, ERR_NODE_NOT_CONFIG_STATE, 0, 0 );
            }

        } break;
    }
}

//----------------------------------------------------------------------------------------------------------
// "handleMsgGetNode" processes an incoming GET message for a node or port attribute. We construct the
// reply message with the requested data.
//
//----------------------------------------------------------------------------------------------------------
void handleMsgGetNode( uint8_t *msg ) {

    uint16_t npId =  (( msg[1] << 8 ) + msg[2] );

    if ( nodeId( npId ) == nodeMap.nodeId ) {

        uint8_t   item  = msg[3];
        uint16_t  arg1  = ( msg[4] << 8 ) + msg[5];
        uint16_t  arg2  = ( msg[6] << 8 ) + msg[7];
        uint8_t   ret   = nodeGet( npId, item, &arg1, &arg2 );

        if ( ret == ALL_OK )  sendRepNode( npId, item, arg1, arg2 );
        else                  sendErr( npId, ret, 0, 0 );
    }
}

//----------------------------------------------------------------------------------------------------------
// "handleMsgPutNode" processes an incoming PUT message for a node or port attribute. We update the data
// and send a confirmation.
//
//----------------------------------------------------------------------------------------------------------
void handleMsgPutNode( uint8_t *msg ) {

    uint16_t npId =  (( msg[1] << 8 ) + msg[2] );

    if ( nodeId( npId ) == nodeMap.nodeId ) {

        uint8_t   item  = msg[3];
        uint16_t  arg1  = ( msg[4] << 8 ) + msg[5];
        uint16_t  arg2  = ( msg[6] << 8 ) + msg[7];
        uint8_t   ret   = nodePut( npId, item, arg1, arg2 );

        if ( ret == ALL_OK )  sendAck( npId );
        else                  sendErr( npId, ret, 0, 0 );
    }
}

//----------------------------------------------------------------------------------------------------------
// "handleMsgRepNode" processes the answer to a previously sent node query. The incoming message will only
// result in an action when we have an outstanding request for that node. That is, this handler will only
// be called when the we passed the outstanding reply map check done before. All reply messages are routed
// to this one callback. It is up to the firmware programmer to analyze for what and whom the reply really
// is.
//
//----------------------------------------------------------------------------------------------------------
void handleMsgRepNode( uint8_t *msg ) {

    uint16_t  npId    = (( msg[1] << 8 ) + msg[2] );
    uint8_t   item    = msg[3];
    uint16_t  arg1    = ( msg[4] << 8 ) + msg[5];
    uint16_t  arg2    = ( msg[6] << 8 ) + msg[7];

    if ( callbackMap.repCallback != nullptr ) callbackMap.repCallback( npId, item, arg1, arg2, ALL_OK );
}

//----------------------------------------------------------------------------------------------------------
// "handleMsgReqNode" processes an incoming request for a node or port. The REQ message request will result
// in invoking the register firmware callback. We send a confirmation message.
//
//----------------------------------------------------------------------------------------------------------
void handleMsgReqNode( uint8_t *msg ) {

    uint16_t npId = (( msg[1] << 8 ) + msg[2] );

    if ( nodeId( npId ) == nodeMap.nodeId ) {

        uint8_t   item  = msg[3];
        uint16_t  arg1  = ( msg[4] << 8 ) + msg[5];
        uint16_t  arg2  = ( msg[6] << 8 ) + msg[7];
```

```
        uint8_t   ret   = nodeReq( npId, item, &arg1, &arg2 );

        if ( ret == ALL_OK )  sendAck( npId );
        else                  sendErr( npId, ret, 0, 0 );
    }
}

//------------------------------------------------------------------------------------------------------------
// "handleMsgEvent" deals with the event messages for inbound ports. For all matching events in the eventMap,
// the respective port map entry fields are set. The searchEvent function will return us the first matching
// entry in the sorted event map, if any. From there, we just hop along the eventMap until the eventId does
// not match any longer. All we do in this routine is to record the event data and the optional future time
// stamp when the event should result in a callback. The actual event processing is done in the port event
// processing routine, which will manage the timely invocation of the event callbacks.
//
// Note that we also are called from the event sending routine because another port on our node could be
// interested in this event. It is up to the firmware programmer to ensure that a port does send itself an
// event and may trigger an infinite loop.
//
//------------------------------------------------------------------------------------------------------------
void handleMsgEvent( uint8_t *msg ) {

    uint16_t  eventId = ( msg[3] * 256 ) + msg[4];
    int       index   = searchEvent( eventId );

    if ( index >= 0 ) {

        uint8_t   opCode            = msg[0];
        uint16_t  nodeId            = ( msg[1] * 256 ) + msg[2];
        uint16_t  eventData         = ( msg[5] * 256 ) + msg[6];
        uint8_t   eventAction       = PEA_EVENT_IDLE;
        uint32_t  ts                = CDC::getMillis( );

        switch ( opCode ) {

            case LCS_OP_EVT_ON:   eventAction = PEA_EVENT_ON;   break;
            case LCS_OP_EVT_OFF:  eventAction = PEA_EVENT_OFF;  break;
            case LCS_OP_EVT:      eventAction = PEA_EVENT_EVT;  break;
        }

        while (( index < nodeMap.eventMapHwm ) && ( eventMap.map[ index ].eventId == eventId )) {

            LcsPortMapEntry *pPtr = &portMap.map[ index ];

            if (( pPtr -> flags & PF_PORT_ENABLED                  ) &&
                ( pPtr -> flags & PF_PORT_EVENT_HANDLING_ENABLED   )) {

                pPtr -> eventNodeId     = nodeId;
                pPtr -> eventId         = eventId;
                pPtr -> eventAction     = eventAction;
                pPtr -> eventValue      = eventData;
                pPtr -> eventTimeStamp  = ts + ( pPtr -> eventDelayTime * EVENT_DELAY_TICK_MILLIS );
                pPtr -> flags           |= PF_EVENT_PENDING;
            }

            index++;
        }
    }
}

//------------------------------------------------------------------------------------------------------------
// We received a DCC subsystem message. These messages are handler solely by firmware, which is typically
// the base station, a handheld, or a decoder alike device. All we do is to pass the message to the call
// back routine. One day, we could decode the message a bit more and invoke more specialized callback.
//
//------------------------------------------------------------------------------------------------------------
void handleMsgDccMgt( uint8_t *msg ) {

    if ( callbackMap.dccMsgCallback != NULL ) callbackMap.dccMsgCallback( msg );
}

//------------------------------------------------------------------------------------------------------------
// Node state INIT. This is the first state after the initial library setup. The runtime init call created
// all memory areas and initialized the data structures. After a successful init call, the state is INIT and
// the firmware programmer can register the necessary callback functions and do other firmware specific work.
// Eventually, the runtime loop method is called. If the "init" option is set, the node init and port init
// callback routine will be invoked. If the nodeId validation option is set, the node will request a nodeId
// and enter the state SETUP. Otherwise the next state is OPERATE.
//
// ??? idea: a port init call cold return a status that says "this port is not used". This way we could
// set a better HWM.
//------------------------------------------------------------------------------------------------------------
void handleNodeStateInit( ) {

    if ( ! ( nodeMap.nodeOptions & NOPT_SKIP_NODE_INIT_STEP )) {

        if ( callbackMap.initCallback != nullptr ) {

            if ( callbackMap.initCallback ) callbackMap.initCallback( nodeMap.nodeId << 4 );
        }

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

            if ( callbackMap.initCallback ) callbackMap.initCallback(( nodeMap.nodeId << 4 ) | i + 1 );

            portMap.map[ i ].flags |= PF_PORT_ENABLED;
            portMap.map[ i ].flags |= PF_PORT_EVENT_HANDLING_ENABLED;
        }
    }
```

```

    if ( ! ( nodeMap.nodeOptions & NOPT_SKIP_NODE_ID_CONFIG )) {

        sendReqNodeId( nodeMap.nodeId, nodeMap.nodeUID, 0 );
        timerVal  = CDC::getMillis( );

        nodeMap.nodeState = NS_REGISTER;

    } else nodeMap.nodeState = NS_OPERATE;
}

//-------------------------------------------------------------------------------------------------------
// Node State FAIL. This is the state after the node startup failed.
//
// ??? would we ever come here ? if the INIT failed, the firmware programmer should do what ?
//-------------------------------------------------------------------------------------------------------
void handleNodeStateFail( ) {

    handleSerialCommand( );

}

//-------------------------------------------------------------------------------------------------------
// Node State Power FAIL. This is the state after when the node starts up after a power fail. We have this
// state so that the firmware programmer can take some recovery action before the power goes away. x
//
//-------------------------------------------------------------------------------------------------------
void handleNodeStatePfail( ) {

    if ( callbackMap.pfailCallback != nullptr ) {

        callbackMap.pfailCallback( nodeMap.nodeId << 4 );
    }

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

        callbackMap.pfailCallback(( nodeMap.nodeId << 4 ) | i + 1 );
    }

    handleSerialCommand( );
}

//-------------------------------------------------------------------------------------------------------
// Node State REGISTER. This is the state after the INIT state when a nodeId setup was requested. We are
// waiting for a nodeId reply message. If there is a timely reply message, we will handle the message reply
// and the node state will advance. If there is no timely reply, we will resubmit the request.
//
//-------------------------------------------------------------------------------------------------------
void handleNodeStateRegister( ) {

    uint8_t msg[ MAX_LCS_MSG_SIZE ];

    switch ( msgBus -> receiveLcsMsg( msg )) {

        case LCS_OP_REP_NID: handleMsgRepNid( msg );  break;
        case LCS_OP_RESET:   handleMsgLcsMgt( msg );  break;

        default: {

            if (( CDC::getMillis( ) - timerVal ) > NODE_SETUP_RETRY_TIMER_VAL_MS ) {

                sendReqNodeId( nodeMap.nodeId, nodeMap.nodeUID, 0 );
                timerVal = CDC::getMillis( );
             }
        }
    }

    handleSerialCommand( );
}

//-------------------------------------------------------------------------------------------------------
// Node State COLLISION. This is the state after the node receiver routine detected a nodeId collision. We
// will stay in this state and only react to RESET and SET_NID messages.
//
//-------------------------------------------------------------------------------------------------------
void handleNodeStateCollision( ) {

    uint8_t msg[ MAX_LCS_MSG_SIZE ];

    switch ( msgBus -> receiveLcsMsg( msg )) {

        case LCS_OP_RESET:
        case LCS_OP_SET_NID:  handleMsgLcsMgt( msg ); break;
    }

    handleSerialCommand( );
}

//-------------------------------------------------------------------------------------------------------
// Node State HALTED. The LCS communication bus was halted for all nodes. Note that the bus is still there,
// just not active. We just listen to the BON or RESET message to get going again.
//
//-------------------------------------------------------------------------------------------------------
void handleNodeStateHalted( ) {

    uint8_t msg[ MAX_LCS_MSG_SIZE ];

    switch ( msgBus -> receiveLcsMsg( msg )) {

```

```
        case LCS_OP_BON:
        case LCS_OP_RESET: handleMsgLcsMgt( msg ); break;
    }
}

//-----------------------------------------------------------------------------------------------------------
// Node State CONFIG. A node can be placed into configuration state. We process any LCS message, handle the
// periodic tasks registered and port events that may have been received. Note that we just listen to messages
// valid for that mode and invoke the respective handler. All other messages are ignored.
//
//-----------------------------------------------------------------------------------------------------------
void handleNodeStateConfig( ) {

    uint8_t msg[ MAX_LCS_MSG_SIZE ];

    switch ( msgBus -> receiveLcsMsg( msg )) {

        case LCS_OP_OPS:
        case LCS_OP_RESET:
        case LCS_OP_BON:
        case LCS_OP_BOF:
        case LCS_OP_ACK:
        case LCS_OP_ERR:
        case LCS_OP_SET_NID:
        case LCS_OP_NCOL:           handleMsgLcsMgt( msg );                break;

        case LCS_OP_NODE_GET:       handleMsgGetNode( msg );               break;
        case LCS_OP_NODE_REP:       handleMsgRepNode( msg );               break;
        case LCS_OP_NODE_REQ:       handleMsgReqNode( msg );               break;
    }

    handlePeriodicTasks( );
    handleNodePortEvents( );
    handleSerialCommand( );
}

//-----------------------------------------------------------------------------------------------------------
// Node State OPERATIONS. Most of the time the node state is in operations mode. We process any LCS message,
// handle the periodic tasks registered and port events that may have been received. Note that we just listen
// to messages valid for that mode and invoke the respective handler. All other messages are ignored.
//
//-----------------------------------------------------------------------------------------------------------
void handleNodeStateOperations( ) {

    uint8_t msg [ MAX_LCS_MSG_SIZE ];

    switch ( msgBus -> receiveLcsMsg( msg )) {

        case LCS_OP_CFG:
        case LCS_OP_RESET:
        case LCS_OP_BON:
        case LCS_OP_BOF:
        case LCS_OP_ACK:
        case LCS_OP_ERR:
        case LCS_OP_REQ_NID:
        case LCS_OP_NCOL:           handleMsgLcsMgt( msg );              break;

        case LCS_OP_NODE_GET:       handleMsgGetNode( msg );             break;
        case LCS_OP_NODE_PUT:       handleMsgPutNode( msg );             break;
        case LCS_OP_NODE_REQ:       handleMsgReqNode( msg );             break;
        case LCS_OP_NODE_REP:       handleMsgRepNode( msg );             break;

        case LCS_OP_EVT_ON:
        case LCS_OP_EVT_OFF:
        case LCS_OP_EVT:            handleMsgEvent( msg );               break;

        case LCS_OP_REQ_LOC:
        case LCS_OP_REL_LOC:
        case LCS_OP_REP_LOC:
        case LCS_OP_SET_LCON:
        case LCS_OP_KEEP_LOC:
        case LCS_OP_SET_LSPD:
        case LCS_OP_SET_LMOD:
        case LCS_OP_LOC_FON:
        case LCS_OP_LOC_FOF:

        case LCS_OP_SET_CVM:
        case LCS_OP_REQ_CVS:
        case LCS_OP_REP_CVS:
        case LCS_OP_SET_CVS:

        case LCS_OP_TON:
        case LCS_OP_TOF:
        case LCS_OP_ESTP:

        case LCS_OP_SEND_DCC3:
        case LCS_OP_SEND_DCC4:
        case LCS_OP_SEND_DCC5:
        case LCS_OP_SEND_DCC6:

        case LCS_OP_DCC_ACK:
        case LCS_OP_DCC_ERR:        handleMsgDccMgt( msg );                break;
    }

    handlePeriodicTasks( );
    handleNodePortEvents( );
    handleSerialCommand( );
}

```

```
//------------------------------------------------------------------------------------------------------------
// "handleNodeState" is the main routine of the node activity processing. It is the method called after all
// setup is done. Running in a loop, the primary function is to handle the activities according to the node
// state. The run loop also processes the serial commands. Note that this function will not return.
//
//------------------------------------------------------------------------------------------------------------
void handleNodeState( ) {

    while ( true ) {

        switch ( nodeMap.nodeState ) {

            case NS_INIT:       handleNodeStateInit( );       break;
            case NS_FAIL:       handleNodeStateFail( );       break;
            case NS_REGISTER:   handleNodeStateRegister( );   break;
            case NS_COLLISION:  handleNodeStateCollision( );  break;
            case NS_HALTED:     handleNodeStateHalted( );     break;
            case NS_CONFIG:     handleNodeStateConfig( );     break;
            case NS_OPERATE:    handleNodeStateOperations( ); break;
            default: ;
        }
    }
}

}; // 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;
```

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

    // or use GET in all constants

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

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

    // thresholds

    // eventID to send for events ?

};

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

const uint16_t  MAX_CAB_SESSIONS                = 64;

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

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

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

    uint16_t  options                       = SM_OPT_DEFAULT_SETTING;

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

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

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

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

    public:

    LcsBaseStationDccTrack( );

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

    uint16_t                    getFlags( );
    uint16_t                    getOptions( );

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

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

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

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

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

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

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

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

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

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

    void                        printDccTrackConfig( );
    void                        printDccTrackStatus( );

    void                        enableLog( bool arg );
    void                        beginLog( );
    void                        endLog( );
    void                        printLog( );

    void                        writeLogData( uint8_t id, uint8_t *buf, uint8_t len );
```

```
    void                        writeLogId( uint8_t id );
    void                        writeLogTs( );
    void                        writeLogVal( uint8_t valId, uint16_t val );

    private:

    uint16_t                    options                       = DT_OPT_DEFAULT_SETTING;
    volatile uint16_t           flags                         = DT_F_DEFAULT_SETTING;

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

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

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

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

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

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

    uint32_t                    totalPwrSamplesTaken          = 0;
    uint32_t                    lastPwrSampleTimeStamp        = 0;

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

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

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

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

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

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

    public:

    static void                 startDccProcessing( );

};

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

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

```

```
};

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

  public:

    LcsBaseStationLocoSession( );

    uint8_t setupSessionMap(

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

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

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

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

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

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

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

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

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

    void                       printSessionMapConfig( );
    void                       printSessionMapInfo( );

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

    private:

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

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

    private:

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

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

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

};

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

```

```
    public:

    LcsBaseStationMsgInterface( );

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

    void handleLcsMsg( uint8_t *msg );

    private:

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

};

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

    public:

    LcsBaseStationCommand( );

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

    void handleSerialCommand( char *s );

    private:

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

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

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

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

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

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

    void printDccLogCommand( char *s );

    private:

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

#endif
```

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

using namespace LCS;

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

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

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

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

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

    return ( ALL_OK );
}

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

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

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

        switch ( s[ charIndex ] ) {

            case '<': {

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

            } break;

            case '>': {

                switch ( cmdStr[ 0 ] ) {

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

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

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

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

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

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

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

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

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

                    case '?': printHelpCmd( ); break;

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

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

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

                charIndex ++;

            } break;

            default: {

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

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

    uint16_t  cabId = NIL_CAB_ID;
    uint8_t   sId   = 0;

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

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

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

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

    uint8_t sId = NIL_LOCO_SESSION_ID;

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

    int ret = locoSessions -> releaseSession( sId );

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    uint8_t sId = locoSessions -> getSessionIdByCabId( cabId );

    if ( sId == NIL_LOCO_SESSION_ID ) return;

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

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

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

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

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

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

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

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

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

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

    uint16_t  cvId          = NIL_DCC_CV_ID;
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

void LcsBaseStationCommand::turnPowerOffAllCmd( ) {

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

void LcsBaseStationCommand::turnPowerOnMainCmd( ) {

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

void LcsBaseStationCommand::turnPowerOnProgCmd( ) {

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

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

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

    uint8_t option = 0;

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

        switch ( option ) {

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

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

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

    uint8_t opt = 0;

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

        switch ( opt ) {

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

            case 9: {

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

            } break;

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

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

    printConfiguration( );
}

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

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

```
}

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

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

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

    locoSessions -> printSessionMapInfo( );
}

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

    mainTrack -> printDccTrackStatus( );
}

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

    progTrack -> printDccTrackStatus( );
}

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

    int opt = -1;

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

    printf( "<a " );

    switch ( opt ) {

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

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

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

    printf( ">" );
}

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

```

    int opt = -1;

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

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

    switch ( opt ) {

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

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

        case 20: {

            uint8_t buf[ 16 ];

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

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

        } break;

        default: ;
    }

    printf( ">" );
}

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

    printf( "\nCommands:\n" );

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

#define DEBUG_WAVE_FORM 0

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

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

using namespace LCS;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

const uint8_t railComDecode[256] = {

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

uint16_t digitValueToMilliAmp( uint16_t digitValue, uint16_t digitsPerAmp ) {

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

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

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

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

    uint8_t followUpMain = DCC_SIG_FOLLOW_UP_NONE;
    uint8_t followUpProg = DCC_SIG_FOLLOW_UP_NONE;

    timeLeftMainTrack -= timeToInterrupt;
    timeLeftProgTrack -= timeToInterrupt;

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

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

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

    CDC::setRepeatingTimerLimit( timeToInterrupt * TICK_IN_MICROSECONDS );

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

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

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

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

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

} // timerCallback

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

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

    CDC::startRepeatingTimer( TICK_IN_MICROSECONDS );
}

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

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

void printLogVal( uint16_t index ) {

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

void printLogData( uint16_t index, uint8_t len ) {

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

uint8_t printLogEntry( uint16_t index ) {

    if ( index < LOG_BUF_SIZE ) {

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

        switch ( logEntryId ) {

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

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

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

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

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

    if ( logActive ) {

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

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

void writeLogId( uint8_t id ) {

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

void writeLogTs( ) {

    if ( logActive ) {

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

void writeLogVal( uint8_t valId, uint16_t val ) {

    if ( logActive ) {

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

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

    logEnabled = arg;
    logActive  = false;
}

void beginLog( ) {

    if ( logEnabled ) {

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

void endLog( ) {

    if ( logActive ) {

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

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

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

    if ( logEnabled ) {

        if ( ! logActive ) {

            if ( logBufIndex > 0 ) {

                printf( "\n" );

                uint16_t entryIndex  = 0;
                uint8_t  entryLen    = 0;

                while ( entryIndex < logBufIndex ) {

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

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

}; // namespace


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


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

    initDccTrackProcessing( );
}

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

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

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

        flags = DT_F_CONFIG_ERROR;
        return ( ERR_DCC_PIN_CONFIG );
    }

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

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

        flags = DT_F_CONFIG_ERROR;
        return ( ERR_DCC_TRACK_CONFIG );
    }

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

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

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

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

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

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

    CDC::onTimerEvent( timerCallback );

    if ( options & DT_OPT_SERVICE_MODE_TRACK ) {

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

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

    if ( trackDesc -> options & DT_OPT_CUTOUT ) {

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

    if ( trackDesc -> options & DT_OPT_RAILCOM ) {

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

            flags = DT_F_CONFIG_ERROR;
            return ( ERR_DCC_TRACK_CONFIG );
        }
    }

    return ( ALL_OK );
}

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

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

    volatile uint8_t  *timeToInterrupt,
    uint8_t           *followUpAction

    ) {

    switch ( signalState ) {

        case DCC_SIG_CUTOUT_START: {

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

        } break;

        case DCC_SIG_CUTOUT_1: {

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

        } break;

        case DCC_SIG_CUTOUT_2: {

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

        } break;

        case DCC_SIG_CUTOUT_3: {

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

            if ( flags & DT_F_RAILCOM_MODE_ON ) {

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

        } break;
```

```
        case DCC_SIG_CUTOUT_END: {

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

        } break;

        case DCC_SIG_START_BIT: {

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

        } break;

        case DCC_SIG_TEST_BIT: {

            if ( currentBit ) {

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

                if ( postambleSent >= postambleLen ) {

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

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

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

            *timeToInterrupt  = TICKS_58_MICROS;

        } break;

        case DCC_SIG_ZERO_SECOND_HALF: {

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

        } break;

        default: {

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

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

    if ( preambleSent < preambleLen ) {

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

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

        if ( bitsSent == 9 ) {

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

        currentBit = true;
        postambleSent ++;
```

```
    }
}

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

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

    if ( activeBufPtr -> repeat > 0 ) {

        activeBufPtr -> repeat --;

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

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

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

        if ( flags & DT_F_SERVICE_MODE_ON ) {

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

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

    dccPacketsSend ++;
}

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

    CDC::startUartRead( uartRxPin );
}

void LcsBaseStationDccTrack::stopRailComIO( ) {

    CDC::stopUartRead( uartRxPin );
}

uint8_t LcsBaseStationDccTrack::handleRailComMsg( ) {

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

    writeLogData( LOG_DCC_RCM, railComMsgBuf, railComBufIndex );

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

        uint8_t dataByte = railComDecode[ railComMsgBuf[ i ]];

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

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

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

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

        railComMsgBuf[ i ] = dataByte;
    }

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

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

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

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

        uint8_t i = 0;

        do {

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

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

        return ( i );

    } else return ( 0 );
}

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

    switch ( trackState ) {

        case DCC_TRACK_POWER_START1: {

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

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

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

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

        }  break;

        case DCC_TRACK_POWER_START2: {

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

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

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

        } break;

        case DCC_TRACK_POWER_ON: {

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

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

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

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

            if ( flags & DT_F_POWER_OVERLOAD ) {

                overloadEventCount  ++;

                if ( overloadEventCount > overloadEventThreshold ) {

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

                        printf( "Overload detected: " );

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

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

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

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

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

        }  break;

        case DCC_TRACK_POWER_OVERLOAD: {

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

                overloadRestartCount ++;

                if ( overloadRestartCount > overloadRestartThreshold ) {

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

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

                    trackState = DCC_TRACK_POWER_STOP1;
                }
                else trackState = DCC_TRACK_POWER_START1;
            }

        }  break;

        case DCC_TRACK_POWER_STOP1: {

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

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

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

        }  break;

        case DCC_TRACK_POWER_STOP2: {

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

        } break;

        case DCC_TRACK_POWER_OFF: {

        } break;
    }
}

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

    return ( flags );
}

uint16_t LcsBaseStationDccTrack::getOptions( ) {

    return ( options );
}

uint32_t LcsBaseStationDccTrack::getDccPacketsSend( ) {

    return ( dccPacketsSend );
}

uint32_t LcsBaseStationDccTrack::getPwrSamplesTaken( ) {

    return ( totalPwrSamplesTaken );
}

uint16_t LcsBaseStationDccTrack::getPwrSamplesPerSec( ) {

    return ( pwrSamplesPerSec );
}

bool LcsBaseStationDccTrack::isPowerOn( ) {

    return ( flags & DT_F_POWER_ON );
}

bool LcsBaseStationDccTrack::isPowerOverload( ) {

    return ( flags & DT_F_POWER_OVERLOAD );
}

bool LcsBaseStationDccTrack::isServiceModeOn( ) {

    return ( flags & DT_F_SERVICE_MODE_ON );
}

bool LcsBaseStationDccTrack::isCutoutOn( ) {

    return ( flags & DT_F_CUTOUT_MODE_ON );
}

bool LcsBaseStationDccTrack::isRailComOn( ) {

    return ( flags & DT_F_RAILCOM_MODE_ON );
}

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

    trackState = DCC_TRACK_POWER_START1;
}

void LcsBaseStationDccTrack::powerStop( ) {

    trackState = DCC_TRACK_POWER_STOP1;
}

void LcsBaseStationDccTrack::serviceModeOn( ) {

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

void LcsBaseStationDccTrack::serviceModeOff( ) {

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

```

void LcsBaseStationDccTrack::cutoutOn( ) {

    if ( ! ( options & DT_OPT_SERVICE_MODE_TRACK )) {

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

void LcsBaseStationDccTrack::cutoutOff( ) {

    if ( ! ( options & DT_OPT_SERVICE_MODE_TRACK )) {

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

void LcsBaseStationDccTrack::railComOn( ) {

    if ( ! ( options & DT_OPT_SERVICE_MODE_TRACK )) {

        flags |= DT_F_CUTOUT_MODE_ON | DT_F_RAILCOM_MODE_ON;
    }
}

void LcsBaseStationDccTrack::railComOff( ) {

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

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

    return ( limitCurrentMilliAmp );
}

uint16_t LcsBaseStationDccTrack::getActualCurrent( ) {

    return ( digitValueToMilliAmp( actualCurrentDigitValue, digitsPerAmp ));
}

uint16_t LcsBaseStationDccTrack::getInitCurrent( ) {

    return ( initCurrentMilliAmp );
}

uint16_t LcsBaseStationDccTrack::getMaxCurrent( ) {

    return ( maxCurrentMilliAmp );
}

void LcsBaseStationDccTrack::setLimitCurrent( uint16_t val ) {

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

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

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

    uint32_t res = 0;

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

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

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

```
    if ( flags & DT_F_MEASUREMENT_ON ) {

        actualCurrentDigitValue = CDC::readAdc( sensePin );

        totalPwrSamplesTaken ++;

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

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

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

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

    uint16_t sum = 0;

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

        highWaterMarkDigitValue = 0;

        loadPacket( resetDccPacketData, 2, 0 );

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

            printf( "%d ", highWaterMarkDigitValue );
        }

        sum += highWaterMarkDigitValue;
    }

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

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

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

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

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

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

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

        highWaterMarkDigitValue = 0;

        loadPacket( resetDccPacketData, 2, 0 );

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

            printf( "%d ", highWaterMarkDigitValue );
        }

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

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

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

        return ( true );
        }
    }

```

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

        printf( ") -> FAILED" );
    }

    return ( false );
}

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

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

    while ( flags & DT_F_DCC_PACKET_PENDING );

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

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

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

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

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

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

    logEnabled = arg;
    logActive  = false;
}

void LcsBaseStationDccTrack::beginLog( ) {

    if ( logEnabled ) {

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

void LcsBaseStationDccTrack::endLog( ) {

    if ( logActive ) {

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

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

    if ( logActive ) {

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

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

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

```

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

void LcsBaseStationDccTrack::writeLogTs( ) {

    if ( logActive ) {

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

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

    if ( logActive ) {

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

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

    if ( logEnabled ) {

        if ( ! logActive ) {

            if ( logBufIndex > 0 ) {

                printf( "\n" );

                uint16_t entryIndex  = 0;
                uint8_t  entryLen    = 0;

                while ( entryIndex < logBufIndex ) {

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

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

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

    printf( "DccTrack Config: " );

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

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

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

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

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

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

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

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

```

    printf( "DccTrack: " );

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

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

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

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

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

using namespace LCS;

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

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

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

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

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

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

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

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

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

bool validCabId( uint16_t cabId ) {

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

bool validCvId( uint16_t cvId ) {

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

bool validFunctionId( uint8_t fId ) {

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

bool validFunctionGroupId( uint8_t fGroup ) {

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

bool validDccPacketlen( uint8_t len ) {

```

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

bool validDccPacketRepeatCnt( uint8_t nRepeat ) {

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

uint8_t lowByte( uint16_t arg ) {

    return( arg & 0xFF );
}

uint8_t highByte( uint16_t arg ) {

    return( arg >> 8 );
}

uint8_t bitRead( uint8_t arg, uint8_t pos ) {

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

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

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

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

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

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

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

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

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

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

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

uint8_t dccFunctionBitToGroup( uint8_t fNum ) {

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

}; // namespace

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

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

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

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

    LcsBaseStationSessionMapDesc  *sessionMapDesc,
    LcsBaseStationDccTrack        *mainTrack,
    LcsBaseStationDccTrack        *progTrack

    ) {

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

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

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

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

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

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

    return ( ALL_OK );
}

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

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

    switch ( mode ) {

        case LSM_NORMAL: {

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

            smePtr -> flags |= SME_SPDIR_ONLY_REFRESH;

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

        case LSM_STEAL: {

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

        } break;

        case LSM_SHARED: {

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

        } break;

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

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

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

    deallocateSessionEntry( smePtr );
    return ( ALL_OK );
}

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

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

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

    return ( ERR_NOT_IMPLEMENTED );
}

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

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

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

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

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

        refreshSessionEntry( sessionMapNextRefresh );

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

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

    // ??? introduce a return status ?

    if ( smePtr -> cabId != NIL_CAB_ID ) {

        if ( flags & SM_F_KEEP_ALIVE_CHECKING ) {

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

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

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

                deallocateSessionEntry( smePtr );
            }
        }

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

        else {

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

            if ( smePtr -> nextRefreshStep == 0 ) {

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

```

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

            uint8_t fGroup = smePtr -> nextRefreshStep;

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

            }
        }
    }
}

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

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

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

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

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

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

uint16_t LcsBaseStationLocoSession::getOptions( ) {

    return ( options );
}

uint16_t LcsBaseStationLocoSession::getFlags( ) {

    return ( flags );
}

uint8_t LcsBaseStationLocoSession::getSessionMapHwm( ) {

    return ( sessionMapHwm - sessionMap );
}

uint32_t LcsBaseStationLocoSession::getSessionKeepAliveInterval( ) {

    return ( refreshAliveTimeOutVal );
}

uint8_t LcsBaseStationLocoSession::getActiveSessions( ) {

    uint8_t sessionCnt = 0;

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

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

    return ( sessionCnt );
}

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

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

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

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

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

```

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

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

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

    if ( smePtr -> flags & SME_COMBINED_REFRESH ) {

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

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

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

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

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

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

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

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

    uint8_t fGroup = dccFunctionBitToGroup( fNum );

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

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

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

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

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

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

    uint8_t pBuf[ MAX_DCC_PACKET_SIZE];
    uint8_t pLen = 0;

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

    switch ( fGroup - 1 ) {

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

```

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

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

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

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

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

    uint8_t   pBuf[ MAX_DCC_PACKET_SIZE ];
    uint8_t   pLen = 0;

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

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

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

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

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

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

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

    uint8_t pBuf[ MAX_DCC_PACKET_SIZE ];
    uint8_t pLen = 0;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

    SessionMapEntry *freePtr = lookupSessionEntry( NIL_CAB_ID );

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

    if ( freePtr != nullptr ) {

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

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

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

    return ( freePtr );
}

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

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

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

            do {

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

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

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

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

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

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

    SessionMapEntry *smePtr = sessionMap;

    while ( smePtr < sessionMapHwm ) {

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

    return ( nullptr );
}

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

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

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

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

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

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

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

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

    printf( "Session Map Info\n" );

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

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

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

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

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

     printf( "\n" );
}

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

  if ( smePtr != nullptr ) {

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

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

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

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

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

```

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

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

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

using namespace LCS;

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

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

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

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

```

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

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

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

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

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

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

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

    lcsConfig.options                  |= NOPT_SKIP_NODE_ID_CONFIG;
}

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

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

  for ( int i = 0; i < msgLen; i++ ) printf( "0x%x ", msg[i] );
  printf( "\n" );
}

uint8_t printStatus (uint8_t status ) {

  printf( "Status: " );
  if ( status == LCS::ALL_OK ) printf( "OK\n" );
  else printf ( "FAILED: %d\n", status );
  return ( status );
}

//--------------------------------------------------------------------------------------------------------
// The node and port initialization callback.
//
// ??? when we know what ports we actually need / use, disable the rest of the ports.
//--------------------------------------------------------------------------------------------------------
uint8_t lcsInitCallback( uint16_t npId ) {

    switch ( npId & 0xF ) {

        case 0:     printf( "Node Init Callback: 0x%x\n", npId >> 4     ); break;
        default:    printf( "Port Init Callback: 0x%x\n", npId &  0xF   );
    }

    return( ALL_OK );
}

//--------------------------------------------------------------------------------------------------------
// The node or port reset callback.
//
//--------------------------------------------------------------------------------------------------------
uint8_t lcsResetCallback( uint16_t npId ) {

    switch ( npId & 0xF ) {

        case 0:     printf( "Node Reset Callback: 0x%x\n", npId >> 4     ); break;
        default:    printf( "Port Reset Callback: 0x%x\n", npId &  0xF   );
    }

    return( ALL_OK );
}

//--------------------------------------------------------------------------------------------------------
// The node or port power fail callback.
//
//--------------------------------------------------------------------------------------------------------
uint8_t lcsPfailCallback( uint16_t npId ) {

    switch ( npId & 0xF ) {

        case 0:     printf( "Node Power Fail Callback: 0x%x\n", npId >> 4     ); break;
        default:    printf( "Port Power Fail Callback: 0x%x\n", npId &  0xF   );
    }

```

```
    return( ALL_OK );
}

//-----------------------------------------------------------------------------------------------------
// The base station has also a command line interpreter. The callback is invoked by the core library when
// there is a command that it does not handle.
//
//-----------------------------------------------------------------------------------------------------
uint8_t lcsCmdCallback( char *cmdLine ) {

    serialCmd.handleSerialCommand( cmdLine );
    return( ALL_OK );
}

//-----------------------------------------------------------------------------------------------------
// Other LCS message callbacks. All we do is to list their invocation. ( for now )
//
//-----------------------------------------------------------------------------------------------------
uint8_t lcsMsgCallback( uint8_t *msg ) {

    printf( "MsgCallback: ", msg  );

    for ( int i = 0; i < 8; i++ ) printf( "0x%2x ");
    printf( "\n" );
    return( ALL_OK );
}

//------------------------------------------------------------------------------------------------------
// The LCS core library ends in a loop that manages its internal workings, invoking the callbacks where
// needed. One set of callbacks are the periodic tasks. The base station needs to periodically run the DCC
// track state machine for power consumption measurement and so on.  Another periodic task is to refresh the
// active locomotive session entries.
//
//------------------------------------------------------------------------------------------------------
uint8_t bsMainTrackCallback( ) {

    mainTrack.runDccTrackStateMachine( );
    return( ALL_OK );
}

uint8_t bsProgTrackCallback( ) {

    progTrack.runDccTrackStateMachine( );
    return( ALL_OK );
}

uint8_t bsRefreshActiveSessionCallback( ) {

    locoSessions.refreshActiveSessions( );
    return( ALL_OK );
}

//------------------------------------------------------------------------------------------------------
// When the base station node receives a request with an item defined in the user item range or the base
// station itself issues such a request, the defined callback is invoked.
//
//------------------------------------------------------------------------------------------------------
uint8_t lcsReqCallback( uint8_t npId, uint8_t item, uint16_t *arg1, uint16_t *arg2 ) {

    printf( "REQ callback: npId: 0x%x, item: %d", npId, item );
    if ( arg1 != nullptr ) printf( ", arg1: %d, ", *arg1 ); else printf( ", arg1: null" );
    if ( arg2 != nullptr ) printf( ", arg2: %d, ", *arg2 ); else printf( ", arg2: null" );
    return( ALL_OK );
}

//------------------------------------------------------------------------------------------------------
// When the base station gets a reply message for a request previously sent, this callback is invoked.
//
//------------------------------------------------------------------------------------------------------
uint8_t lcsRepCallback( uint8_t npId, uint8_t item, uint16_t arg1, uint16_t arg2, uint8_t ret ) {

    printf( "REP callback: npId: 0x%x, item: %d, arg1: %d, arg2: %d, ret: %d ", npId, item , arg1, arg2, ret );
    return( ALL_OK );
}

//------------------------------------------------------------------------------------------------------
// For any event on the LCS system that the base station is interested in, this callback is invoked.
//
//------------------------------------------------------------------------------------------------------
uint8_t lcsEventCallback( uint16_t npId, uint16_t eId, uint8_t eAction, uint16_t eData ) {

    printf( "Event: npId: 0x%x, eId: %d, eAction: %d, eData: %d\n", npId, eId, eAction, eData );
    return( ALL_OK );
}

//-----------------------------------------------------------------------------------------------------
// Init the Runtime.
//
//-----------------------------------------------------------------------------------------------------
uint8_t initLcsRuntime( ) {

    setupConfigInfo( );

    uint8_t rStat = LCS::initRuntime( &lcsConfig, &cdcConfig );
    printf( "LCS Base Station\n" );

    CDC::printConfigInfo( &cdcConfig );

    printStatus( rStat );
```

```
    return( rStat );
}

//-----------------------------------------------------------------------------------------------------------
// This routine initializes the Loco Session Map Object.
//
//-----------------------------------------------------------------------------------------------------------
uint8_t setupLocoSessions( ) {

  LcsBaseStationSessionMapDesc sessionDesc;

  sessionDesc.options     = SM_OPT_ENABLE_REFRESH;
  sessionDesc.maxSessions = 16;

  printf( "Setup Session Map -> " );
  return ( printStatus( locoSessions.setupSessionMap( &sessionDesc, &mainTrack, &progTrack )));
}

//-----------------------------------------------------------------------------------------------------------
// This routine initializes the MAIN track object.
//
// ??? define constants such as: SENSE_0R1_OPAMP_11 to set the milliVolts per Amp.
//-----------------------------------------------------------------------------------------------------------
int setupDccTrackMain( ) {

  LcsBaseStationTrackDesc mainTrackDesc;

  mainTrackDesc.options                     = DT_OPT_RAILCOM | DT_OPT_CUTOUT;

  mainTrackDesc.enablePin                   = cdcConfig.DIO_PIN_6;
  mainTrackDesc.dccSigPin1                  = cdcConfig.DIO_PIN_2;
  mainTrackDesc.dccSigPin2                  = cdcConfig.DIO_PIN_3;
  mainTrackDesc.sensePin                    = cdcConfig.ADC_PIN_0;
  mainTrackDesc.uartRxPin                   = cdcConfig.UART_RX_PIN_1;

  mainTrackDesc.initCurrentMilliAmp         = 500;
  mainTrackDesc.limitCurrentMilliAmp        = 1500;
  mainTrackDesc.maxCurrentMilliAmp          = 2000;
  mainTrackDesc.milliVoltPerAmp             = 100 * 11;  // ??? opAmp has Factor eleven ...
  mainTrackDesc.startTimeThresholdMillis    = 1000;
  mainTrackDesc.stopTimeThresholdMillis     = 500;
  mainTrackDesc.overloadTimeThresholdMillis = 500;
  mainTrackDesc.overloadEventThreshold      = 10;
  mainTrackDesc.overloadRestartThreshold    = 5;

  printf( "Setup MAIN track -> " );
  return ( printStatus( mainTrack.setupDccTrack( &mainTrackDesc )));
}

//-----------------------------------------------------------------------------------------------------------
// This routine initializes the PROG track object.
//
// ??? define constants such as: SENSE_0R1_OPAMP_11 to set the milliVolts per Amp.
//-----------------------------------------------------------------------------------------------------------
uint8_t setupDccTrackProg( ) {

  LcsBaseStationTrackDesc progTrackDesc;

  progTrackDesc.options                     = DT_OPT_SERVICE_MODE_TRACK;

  progTrackDesc.enablePin                   = cdcConfig.DIO_PIN_7;
  progTrackDesc.dccSigPin1                  = cdcConfig.DIO_PIN_4;
  progTrackDesc.dccSigPin2                  = cdcConfig.DIO_PIN_5;
  progTrackDesc.sensePin                    = cdcConfig.ADC_PIN_1;
  progTrackDesc.uartRxPin                   = cdcConfig.UART_RX_PIN_2;

  progTrackDesc.initCurrentMilliAmp         = 500;
  progTrackDesc.limitCurrentMilliAmp        = 500;
  progTrackDesc.maxCurrentMilliAmp          = 1000;
  progTrackDesc.milliVoltPerAmp             = 100 * 11;  // ??? opAmp has Factor eleven ...
  progTrackDesc.startTimeThresholdMillis    = 1000;
  progTrackDesc.stopTimeThresholdMillis     = 500;
  progTrackDesc.overloadTimeThresholdMillis = 500;
  progTrackDesc.overloadEventThreshold      = 10;
  progTrackDesc.overloadRestartThreshold    = 5;

  printf( "Setup PROG track -> " );
  return ( printStatus( progTrack.setupDccTrack( &progTrackDesc )));
}

//-----------------------------------------------------------------------------------------------------------
// The base station has also a command interpreter, primarily for the DCC++ commands.
//
//-----------------------------------------------------------------------------------------------------------
uint8_t setupSerialCommand( ) {

  printf( "Setup Serial Command -> " );
  return ( printStatus( serialCmd.setupSerialCommand( &locoSessions, &mainTrack, &progTrack )));
}

//-----------------------------------------------------------------------------------------------------------
// The LCS message interface is initialized in the LCS core library. This routine will set up the receiver
// handler for incoming LCS message that concern the base station.
//
//-----------------------------------------------------------------------------------------------------------
uint8_t setupMsgInterface( ) {

  printf( "Setup LCS Msg Interface -> " );
  return ( printStatus( msgInterface.setupLcsMsgInterface( &locoSessions, &mainTrack, &progTrack )));
```

```
}

//-----------------------------------------------------------------------------------------------------
// After the initial setup of the runtime library, the callback are registered.
//
//-----------------------------------------------------------------------------------------------------
uint8_t registerCallbacks( ) {

    printf( "Registering Callbacks\n" );

    registerLcsMsgCallback( lcsMsgCallback );
    registerCmdCallback( lcsCmdCallback );
    registerInitCallback( lcsInitCallback );
    registerResetCallback( lcsResetCallback );
    registerPfailCallback( lcsPfailCallback );
    registerReqCallback( lcsReqCallback );
    registerRepCallback( lcsRepCallback );
    registerEventCallback( lcsEventCallback );
    registerTaskCallback( bsMainTrackCallback, MAIN_TRACK_STATE_TIME_INTERVAL );
    registerTaskCallback( bsProgTrackCallback, PROG_TRACK_STATE_TIME_INTERVAL );
    registerTaskCallback( bsRefreshActiveSessionCallback, SESSION_REFRESH_TASK_INTERVAL );

    return( ALL_OK );
}

//-----------------------------------------------------------------------------------------------------
// Fire up the base station. First all base station modules are initialized. If this is OK, the DCC tack
// signal generation is enabled, i.e. the interrupt driven DCC packet broadcasting starts. Finally, the
// track power is turned on and we give control to the LCS runtime for processing events and requests.
//
//-----------------------------------------------------------------------------------------------------
uint8_t startBaseStation( ) {

    uint8_t rStat = ALL_OK;

    if ( rStat == ALL_OK ) rStat = setupSerialCommand( );
    if ( rStat == ALL_OK ) rStat = setupMsgInterface( );
    if ( rStat == ALL_OK ) rStat = setupLocoSessions( );
    if ( rStat == ALL_OK ) rStat = setupDccTrackMain( );
    if ( rStat == ALL_OK ) rStat = setupDccTrackProg( );

    if ( rStat == ALL_OK ) {

        LcsBaseStationDccTrack::startDccProcessing( );

        mainTrack.powerStart( );
        progTrack.powerStart( );

        // ??? bracket so that it is not printed when no console...
        mainTrack.printDccTrackStatus( );
        progTrack.printDccTrackStatus( );
        printf( "Ready...\n" );

        startRuntime( );
  }

  return( ALL_OK );
}

//-----------------------------------------------------------------------------------------------------
// The main program. Setup the runtime, register the callbacks, and get the show on the road.
//
//-----------------------------------------------------------------------------------------------------
int main( ) {

    uint8_t rStat = ALL_OK;

    if ( rStat == ALL_OK ) rStat = initLcsRuntime( );
    if ( rStat == ALL_OK ) rStat = registerCallbacks( );
    if ( rStat == ALL_OK ) return( startBaseStation( ));
}
```

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
```

```
//      DBG_BC_SETUP                    -   show the setup steps
//      DBG_BC_LCS_MSG_INTERFACE        -   show the incoming LCS messages
//      DBG_BC_TRACK_POWER_MGMT         -   show the track power measurement data
//      DBG_BC_RAILCOM                  -   show the RailCom activity
//
// The way to use these flags is for example:
//
//      if (( debugMask & DBG_BC_CONFIG ) && ( debugMask & DBG_BC_SESSION ))
//
//----------------------------------------------------------------------------------------------------------
enum BlockControllerDebugFlags : uint16_t {

    DBG_BC_CONFIG                   = 1 << 15,       // DEBUG enabled
    DBG_BC_SETUP                    = 1 << 1,        // show the setup steps
    DBG_BC_LCS_MSG_INTERFACE        = 1 << 2,        // show the incoming LCS messages
    DBG_BC_TRACK_POWER_MGMT         = 1 << 3,        // show the track power measurement data
    DBG_BC_RAILCOM                  = 1 << 4         // show the RailCom activity
};

//----------------------------------------------------------------------------------------------------------
// The base station items for nodeInfo and nodeControl calls .... tbd
//
// ??? the are mapped in the MEM / NVM range as well as in the USER range.
// ??? how to do it consistently and understandably ?
//----------------------------------------------------------------------------------------------------------
enum BlockControllerItems : uint8_t {

    BC_ITEM_SET_TRACK_STATE          = 64,

    BC_ITEM_INIT_CURRENT_VAL         = 140,
    BC_ITEM_LIMIT_CURRENT_VAL        = 141,
    BC_ITEM_MAX_CURRENT_VAL          = 142,
    BC_ITEM_ACTUAL_CURRENT_VAL       = 143,

    // thresholds
    // eventID to send for events ?
};

//----------------------------------------------------------------------------------------------------------
// 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 BlockControllerErrors : uint8_t {

    BLOCK_CONTROLLER_ERR_BASE        = 128,

    ERR_MSG_INTERFACE_SETUP          = BLOCK_CONTROLLER_ERR_BASE + 10,
    ERR_DCC_TRACK_CONFIG             = BLOCK_CONTROLLER_ERR_BASE + 11,
    ERR_PIN_CONFIG                   = BLOCK_CONTROLLER_ERR_BASE + 12,
    ERR_TRACK_CONFIG                 = BLOCK_CONTROLLER_ERR_BASE + 13,

    ERR_NVM_HW_SETUP                 = BLOCK_CONTROLLER_ERR_BASE + 15,
    ERR_PIO_HW_SETUP                 = BLOCK_CONTROLLER_ERR_BASE + 16
};

//----------------------------------------------------------------------------------------------------------
// Setup options to set for the DCC track. They are set when the track object is created.
//
//  DT_OPT_SERVICE_MODE_TRACK   - The track is a PROG track.
//  DT_OPT_RAILCOM              - The track support Railcom detection.
//
//----------------------------------------------------------------------------------------------------------
enum BlockControllerTrackOptions : uint16_t {

    BT_OPT_DEFAULT_SETTING      = 0,
    BT_OPT_RAILCOM              = 1 << 1
};

//----------------------------------------------------------------------------------------------------------
// The block track object has a set of flags to indicate its current status.
//
//  DT_F_POWER_ON              - The track is under power.
//  DT_F_POWER_OVERLOAD        - An overload situation was detected.
//  DT_F_MEASUREMENT_ON        - The power measurement is enabled.
//  DT_F_SERVICE_MODE_ON       - The track is currently in service mode, i.e. is a PROG track.
//  DT_F_CUTOUT_MODE_ON        - The track has the cutout generation enabled.
//  DT_F_RAILCOM_MODE_ON       - The track has the railcom detect enabled.
//  DT_F_RAILCOM_MSG_PENDING   - If railcom is enabled, a received datagram is indicated.
//  DT_F_CONFIG_ERROR          - The passed configuration descriptor has invalid options configured.
//
//----------------------------------------------------------------------------------------------------------
enum TrackFlags : uint16_t {

    BT_F_DEFAULT_SETTING      = 0,
    BT_F_POWER_ON             = 1 << 0,
    BT_F_POWER_OVERLOAD       = 1 << 1,
    BT_F_MEASUREMENT_ON       = 1 << 2,
    BT_F_CONFIG_ERROR         = 1 << 15
};

//----------------------------------------------------------------------------------------------------------
// The following constants are for the current consumption RMS measurement. The idea is to record the measured
// ADC values in a circular buffer, every time a certain amount of milliseconds has passed. This work is done
// by the DCC track state machine as part of the power on state.
//
//----------------------------------------------------------------------------------------------------------
const uint8_t   PWR_SAMPLE_BUF_SIZE               = 64;
const uint32_t  PWR_SAMPLE_TIME_INTERVAL_MILLIS   = 16;
```

```

//-------------------------------------------------------------------------------------------------------
// The track state machine runs at a time interval.
//
//-------------------------------------------------------------------------------------------------------
const uint32_t TRACK_STATE_TIME_INTERVAL  = 10;

//-------------------------------------------------------------------------------------------------------
// A block track can be in four states.
//
//-------------------------------------------------------------------------------------------------------
enum BlockTrackMode : uint16_t {

    BT_MODE_OFF        = 0,
    BT_MODE_PWM_FWD    = 1,
    BT_MODE_PWM_REV    = 2,
    BT_MODE_DCC        = 3
};

//---------------------------------------------------------------------------------------------------------
// The block controller can contain up to four blocks. Each block track is described by the LcsBlockDesc
// descriptor. There are the hardware pins sel1Pin1, selPin2, sensePin and uartRxPin. In addition there are
// the limits for current consumption values, all specified in milliAmps. The initial current sets the current
// consumption limit after the track is turned on. The limit current consumption specifies the actual
// configured value that is checked for a track current overload situation. The maximum current defines what
// current the power module should never exceed. For the measurements to work, the power module needs to
// deliver a voltage that corresponds to the current drawn on the track. The value is measured in milliVolt
// per Ampere drawn. Finally, there are threshold times for managing the track overload and restart
// capability.
//
//---------------------------------------------------------------------------------------------------------
struct LcsBlockTrackDesc {

    uint16_t    options;
    uint8_t     selPin1                         = CDC::UNDEFINED_PIN;
    uint8_t     selPin2                         = CDC::UNDEFINED_PIN;
    uint8_t     sensePin                        = CDC::UNDEFINED_PIN;

    uint16_t    pwmFrequency                    = 70;
    uint16_t    initialTrackMode                = BT_MODE_OFF;
    uint16_t    initialTrackSpeed               = 0;

    uint16_t    initCurrentMilliAmp             = 0;
    uint16_t    limitCurrentMilliAmp            = 0;
    uint16_t    maxCurrentMilliAmp              = 0;
    uint16_t    milliVoltPerAmp                 = 0;

    uint16_t    startTimeThresholdMillis        = 0;
    uint16_t    stopTimeThresholdMillis         = 0;
    uint16_t    overloadTimeThresholdMillis     = 0;
    uint16_t    overloadEventThreshold          = 0;
    uint16_t    overloadRestartThreshold        = 0;
};

//---------------------------------------------------------------------------------------------------------
// The "LcsBlockTrack" manages the track of a block. This primarily the power management and control of the
// H-Bridge settings. There is one object per track block. At the heart of the object is a state machine that
// is executed very often for measuring the power consumption and overload detection logic. The tack can
// operate in digital or analog mode. In digital mode, the DCC signal from the LCS bus is routed though to
// the H-Bridge, in analog mode a PWM signal is used to set the H-Bridge emitting a PWM signal with a
// positive or negative voltage.
//
//---------------------------------------------------------------------------------------------------------
struct LcsBlockTrack {

    public:

    LcsBlockTrack( );

    uint8_t                     setupBlockTrack( LcsBlockTrackDesc* trackDesc );
    uint8_t                     setTrackState( uint16_t state );
    uint8_t                     setTrackMode( uint16_t mode, uint8_t speed = 0 );

    uint16_t                    getFlags( );
    uint16_t                    getOptions( );

    void                        runTrackStateMachine( );

    void                        powerStart( );
    void                        powerStop( );
    bool                        isPowerOn( );
    bool                        isPowerOverload( );

    void                        setLimitCurrent( uint16_t val );
    uint16_t                    getLimitCurrent( );
    uint16_t                    getActualCurrent( );
    uint16_t                    getInitCurrent( );
    uint16_t                    getMaxCurrent( );
    uint16_t                    getRMSCurrent( );

    void                        checkOverload( );
    void                        powerMeasurement( );

    uint32_t                    getPwrSamplesTaken( );
    uint16_t                    getPwrSamplesPerSec( );

    void                        printTrackConfig( );
    void                        printTrackStatus( );

```

```
    private:

    uint16_t                    options                         = BT_OPT_DEFAULT_SETTING;
    volatile uint16_t           flags                           = BT_F_DEFAULT_SETTING;

    volatile uint16_t           trackState                      = 0;
    volatile uint16_t           trackMode                       = 0;
    volatile uint16_t           trackSpeed                      = 0;
    volatile uint32_t           trackTimeStamp                  = 0;
    volatile uint8_t            overloadEventCount              = 0;
    volatile uint8_t            overloadRestartCount            = 0;

    uint8_t                     selPin1                         = CDC::UNDEFINED_PIN;
    uint8_t                     selPin2                         = CDC::UNDEFINED_PIN;
    uint8_t                     sensePin                        = CDC::UNDEFINED_PIN;

    uint16_t                    pwmFrequency                    = 0;
    uint16_t                    initialTrackMode                = 0;
    uint16_t                    initialTrackSpeed               = 0;
    uint16_t                    initCurrentMilliAmp             = 0;
    uint16_t                    limitCurrentMilliAmp            = 0;
    uint16_t                    maxCurrentMilliAmp              = 0;

    uint16_t                    startTimeThreshold              = 0;
    uint16_t                    stopTimeThreshold               = 0;
    uint16_t                    overloadTimeThreshold           = 0;
    uint16_t                    overloadEventThreshold          = 0;
    uint16_t                    overloadRestartThreshold        = 0;

    uint16_t                    milliVoltPerAmp                 = 0;
    uint16_t                    digitsPerAmp                    = 0;
    volatile uint16_t           actualCurrentDigitValue         = 0;
    volatile uint16_t           highWaterMarkDigitValue         = 0;
    volatile uint16_t           limitCurrentDigitValue          = 0;

    volatile uint32_t           totalPwrSamplesTaken            = 0;
    uint32_t                    lastPwrSampleTimeStamp          = 0;

    uint32_t                    lastPwrSamplePerSecTaken        = 0;
    uint32_t                    lastPwrSamplePerSecTimeStamp    = 0;
    uint32_t                    pwrSamplesPerSec                = 0;

    uint8_t                     pwrSampleBufIndex                       = 0;
    uint16_t                    pwrSampleBuf[ PWR_SAMPLE_BUF_SIZE ]     = { 0 };

};

//----------------------------------------------------------------------------------------------------------
// "LcsOccDetect" manages an Occupancy detector extension board. The track power output of a block controller
// track is routed to an extension board which implements a set of current detectors. The extension board is
// access via the extension I2C bus.
//
//----------------------------------------------------------------------------------------------------------
struct LcsOccDetect {

    public:

    LcsOccDetect( );

    uint8_t getOccDetectMask( uint16_t *mask );

    private:

    // ??? need to remember the extension board ID.

};

//----------------------------------------------------------------------------------------------------------
// "LcsSignal" manages a signal. A block has a signal for each direction to indicate the state of the next
// block in a route.
//
//----------------------------------------------------------------------------------------------------------
struct LcsSignalControl {

    public:

    LcsSignalControl( );


    private:

    // ??? need to remember the extension board ID.

};

//----------------------------------------------------------------------------------------------------------
// "LcsTurnout" manages the optional turnouts at the end of a block.
//
//
//----------------------------------------------------------------------------------------------------------
struct LcsTurnoutControl {

    public:

    LcsTurnoutControl( );

    private:

    // ??? need to remember the extension board ID.
```

```
};

//-----------------------------------------------------------------------------------------------------------
// "LcsRailComDetect" manages the optional RailCom interface for the block.
//
//-----------------------------------------------------------------------------------------------------------
struct LcsRailComDetect {

    public:

    LcsRailComDetect( );

    private:

    // ??? need to remember the extension board ID.
};

//-----------------------------------------------------------------------------------------------------------
// "LcsBlockControl" manages a block. A block consists mainly of the tack itself and the optional elements
// detectors, signal and turnouts. The block logic, i.e. what to do when the next block is occupied, is
// handled here.
//
//
// ??? runs the block logic
// ??? how to assign occ detect an signals to the block ?
// ??? should message handling be a separate part ?
// ??? can we build the control logic in such a way that it is configurable via ITEMs ?
//-----------------------------------------------------------------------------------------------------------
struct LcsBlockControl {

    LcsBlockControl(  );

    uint8_t handleLcsRequest( uint8_t *msg );

    private:

    // ??? handles to detect, signal and turnout object.

};

//-----------------------------------------------------------------------------------------------------------
// A LCS block controller node can host up to four blocks. This object is the main object that manages the
// blocks on the node.
//
// ??? the node descriptor is an array of block descriptors. They are kept in the NVM ?
// ??? manages the LCS messages and forwards them to the target block.
//-----------------------------------------------------------------------------------------------------------
struct LcsBlockControllerNode {

    public:

    LcsBlockControllerNode( );

    // ??? setup
    // ??? targets for the LCS callbacks ?

    private:

    uint16_t    options     = 0;
    uint16_t    flags       = 0;
    uint16_t    hwm         = 0;

    LcsBlockControl map[ 4 ];
};

#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...

//--------------------------------------------------------------------------------------------------------
```

```
// The node and port initialization callback.
//
// ??? when we know what ports we actually need / use, disable the rest of the ports.
// ??? the number of ports / blocks should be note in the block descriptor.
// ??? invoke the configured block reset method in the block controller logic object...
//-----------------------------------------------------------------------------------------------
uint8_t lcsInitCallback( uint16_t npId ) {

    switch ( npId & 0xF ) {

        case 0:     printf( "Node Init Callback: 0x%x\n", npId >> 4     ); break;
        default:    printf( "Port Init Callback: 0x%x\n", npId &  0xF   );
    }

    return( ALL_OK );
}

//-----------------------------------------------------------------------------------------------
// The node or port reset callback.
//
// ??? invoke the configured block reset method in the block controller logic object...
//-----------------------------------------------------------------------------------------------
uint8_t lcsResetCallback( uint16_t npId ) {

    switch ( npId & 0xF ) {

        case 0:     printf( "Node Reset Callback: 0x%x\n", npId >> 4     ); break;
        default:    printf( "Port Reset Callback: 0x%x\n", npId &  0xF   );
    }

    return( ALL_OK );
}

//-----------------------------------------------------------------------------------------------
// The node or port power fail callback.
//
// ??? invoke the configured block pfail method in the block controller logic object...
//-----------------------------------------------------------------------------------------------
uint8_t lcsPfailCallback( uint16_t npId ) {

    switch ( npId & 0xF ) {

        case 0:     printf( "Node Power Fail Callback: 0x%x\n", npId >> 4     ); break;
        default:    printf( "Port Power Fail Callback: 0x%x\n", npId &  0xF   );
    }

    return( ALL_OK );
}

//-----------------------------------------------------------------------------------------------
// Other LCS message callbacks. All we do is to list their invocation. ( for now )
//
// ??? this should go out ....
//-----------------------------------------------------------------------------------------------
uint8_t lcsMsgCallback( uint8_t *msg ) {

    printf( "MsgCallback: ", msg  );
    printLcsMsg( msg );
    return( ALL_OK );
}

//-------------------------------------------------------------------------------------------------
// When the base station node receives a request with an item defined in the user item range or the base
// station itself issues such a request, the defined callback is invoked.
//
// ??? pass to the block controller logic...
//-------------------------------------------------------------------------------------------------
uint8_t lcsReqCallback( uint8_t npId, uint8_t item, uint16_t *arg1, uint16_t *arg2 ) {

    printf( "REQ callback: npId: 0x%x, item: %d", npId, item );
    if ( arg1 != nullptr ) printf( ", arg1: %d, ", *arg1 ); else printf( ", arg1: null" );
    if ( arg2 != nullptr ) printf( ", arg2: %d, ", *arg2 ); else printf( ", arg2: null" );

    switch( item ) {

        case 64: {

            uint16_t port = npId & 0xF;

            if      ( port == 1 ) block1 -> setTrackMode( *arg1 & 0xFF, *arg2 & 0xFF );
            else if ( port == 2 ) block2 -> setTrackMode( *arg1 & 0xFF, *arg2 & 0xFF );

        } break;

        default: {

        }
    }

    return( ALL_OK );
}

//-------------------------------------------------------------------------------------------------
// When the base station gets a reply message for a request previously sent, this callback is invoked.
//
// ??? pass to the block that requested...
//-------------------------------------------------------------------------------------------------
uint8_t lcsRepCallback( uint8_t npId, uint8_t item, uint16_t arg1, uint16_t arg2, uint8_t ret ) {

```

```
    printf( "REP callback: npId: 0x%x, item: %d, arg1: %d, arg2: %d, ret: %d ", npId, item , arg1, arg2, ret );
    return( ALL_OK );
}

//----------------------------------------------------------------------------------------------------------------
// For any event on the LCS system that the block controller is interested in, this callback is invoked.
//
// ??? what events to listen to ? where are they configured/set ?
//----------------------------------------------------------------------------------------------------------------
uint8_t lcsEventCallback( uint16_t npId, uint16_t eId, uint8_t eAction, uint16_t eData ) {

    printf( "Event: npId: 0x%x, eId: %d, eAction: %d, eData: %d\n", npId, eId, eAction, eData );
    return( ALL_OK );
}

//----------------------------------------------------------------------------------------------------------------
// We need to run the track state machines on a periodic basis.
//
//
// ??? we actually need an array of track machines ?
// ??? or should we register each one individually ?
//----------------------------------------------------------------------------------------------------------------
uint8_t trackStateMachine( ) {

    if ( block1 != nullptr ) block1 -> runTrackStateMachine( );
    if ( block2 != nullptr ) block2 -> runTrackStateMachine( );
    return( ALL_OK );
}

//--------------------------------------------------------------------------------------------------------------
// Init the Runtime.
//
//--------------------------------------------------------------------------------------------------------------
uint8_t initLcsRuntime( ) {

    printf( "LCS Block Controller\n" );
    printf( "initLcsRuntime\n" );

    setupConfigInfo( );

    uint8_t rStat = initRuntime( &lcsConfig, &cdcConfig );

    CDC::printConfigInfo( &cdcConfig );
    return( printStatus( rStat ));
}

//--------------------------------------------------------------------------------------------------------------
// After the initial setup of the runtime library, the callback are registered.
//
//--------------------------------------------------------------------------------------------------------------
uint8_t registerCallbacks( ) {

    printf( "Registering Callbacks\n" );

    registerLcsMsgCallback( lcsMsgCallback );
    registerInitCallback( lcsInitCallback );
    registerResetCallback( lcsResetCallback );
    registerPfailCallback( lcsPfailCallback );
    registerReqCallback( lcsReqCallback );
    registerRepCallback( lcsRepCallback );
    registerEventCallback( lcsEventCallback );
    registerTaskCallback( trackStateMachine, TRACK_STATE_TIME_INTERVAL );

    return( printStatus( ALL_OK ));
}

//--------------------------------------------------------------------------------------------------------------
// Fire up the base station. First all base station modules are initialized. If this is OK, the DCC tack
// signal generation is enabled, i.e. the interrupt driven DCC packet broadcasting starts. Finally, the
// track power is turned on and we give control to the LCS runtime for processing events and requests.
//
//--------------------------------------------------------------------------------------------------------------
uint8_t startBlockController( ) {

    printf( "Start Block controller\n" );

    uint8_t rStat = ALL_OK;

    block1 = new LcsBlockTrack( );
    block2 = new LcsBlockTrack( );

    printf( "Configure Block 1\n" );
    rStat = block1 -> setupBlockTrack( &block1Desc );
    if ( rStat != ALL_OK ) printStatus( rStat );

    printf( "Configure Block 2\n" );
    rStat = block1 -> setupBlockTrack( &block2Desc );
    if ( rStat != ALL_OK ) printStatus( rStat );

    #if 0
    // ??? for the quick test ....
    printf( "Configure PWM pins\n" );
    rStat = CDC::configurePwm( cdcConfig.PWM_PIN_0, 70 );
    if ( rStat != ALL_OK ) printStatus( rStat );

    rStat = CDC::configurePwm( cdcConfig.PWM_PIN_1, 70 );
    if ( rStat != ALL_OK ) printStatus( rStat );

    rStat = CDC::configurePwm( cdcConfig.PWM_PIN_2, 70 );
```

```
    if ( rStat != ALL_OK ) printStatus( rStat );

    rStat = CDC::configurePwm( cdcConfig.PWM_PIN_3, 70 );
    if ( rStat != ALL_OK ) printStatus( rStat );

    printf( "Set output to pins\n" );
    rStat = CDC::writePwm(cdcConfig.PWM_PIN_0, 255 );
    if ( rStat != ALL_OK ) printStatus( rStat );
    rStat = CDC::writePwm(cdcConfig.PWM_PIN_1, 255 );
    if ( rStat != ALL_OK ) printStatus( rStat );
    rStat = CDC::writePwm(cdcConfig.PWM_PIN_2, 0 );
    if ( rStat != ALL_OK ) printStatus( rStat );
    rStat = CDC::writePwm(cdcConfig.PWM_PIN_3, 120 );
    if ( rStat != ALL_OK ) printStatus( rStat );
    #endif

    printf( "Block 1 Config:" );
    if ( block1 != nullptr ) block1 -> printTrackConfig( );

    printf( "Block 2 Config:" );
    if ( block2 != nullptr ) block2 -> printTrackConfig( );

    if ( rStat == ALL_OK ) {

        printf( "Ready...\n" );
        startRuntime( );
    }

    return( ALL_OK );
}

//------------------------------------------------------------------------------------------------------------
// The main program. Setup the runtime, register the callbacks, and get the show on the road.
//
//------------------------------------------------------------------------------------------------------------
int main( ) {

    uint8_t rStat = ALL_OK;

    if ( rStat == ALL_OK ) rStat = initLcsRuntime( );
    if ( rStat == ALL_OK ) rStat = registerCallbacks( );
    if ( rStat == ALL_OK ) return( startBlockController( ));
}
```

```
//--------------------------------------------------------------------------------------------------------------
//
// 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 ) {

```

```
    return (( val >= lower ) && ( val <= upper ));
}

//-----------------------------------------------------------------------------------------------------------
// Conversion functions between milliAmps and digit values as report4de by the analog to digital converter
// hardware. For a better precision, the formula uses 32 bit computation and stores the result back in a
// 16 bit quantity.
//
//-----------------------------------------------------------------------------------------------------------
uint16_t milliAmpToDigitValue( uint16_t milliAmp, uint16_t digitsPerAmp ) {

    #if 0
    uint32_t mA = milliAmp;
    uint32_t dPA = digitsPerAmp;
    return (( uint16_t ) ( mA * dPA / 1000 ));
    #endif

    return ((uint16_t) ((((uint32_t) milliAmp ) * ((uint32_t) digitsPerAmp )) / 1000 ));
}

uint16_t digitValueToMilliAmp( uint16_t digitValue, uint16_t digitsPerAmp ) {

    #if 0
    uint32_t dV = digitValue;
    uint32_t dPA = digitsPerAmp;
    return ((uint16_t)( dV * 1000 / dPA ));
    #endif

    return ((uint16_t) ((((uint32_t) digitValue ) * 1000 ) / ((uint32_t) digitsPerAmp )));
}

}; // namespace


//===========================================================================================================
//===========================================================================================================
//
// Object part.
//
//===========================================================================================================
//===========================================================================================================


//-----------------------------------------------------------------------------------------------------------
// Object instance section. The DccTrack constructor. Nothing to do so far.
//
//-----------------------------------------------------------------------------------------------------------
LcsBlockTrack::LcsBlockTrack( ) { }


//-----------------------------------------------------------------------------------------------------------
// "setupDccTrack" performs the setup tasks for the DCC track.  We will configure the hardware, the DCC
// packet options such as preamble and postamble length, the initial state machine state current consumption
// limit and load the initial packet into the active buffer. There is quite a list of parameters and options
// that can be set. This routine does the following checking:
//
//     - the pins used in the CDC layer must be a pair ( for atmega controllers ).
//     - the sensePin must be an analog input pin.
//     - if the track is a service track, cutout and RailCom are not supported.
//     - if RailCom is set, Cutout must be set too.
//     - the initial current limit consumption setting must be less than the current limit setting.
//     - the current limit setting must be less than the maximum current limit setting.
//
// Once the DCC track object is initialized, the last thing to do is to remember the object instance in the
// file static variables. This is necessary for the interrupt handlers to work. If any of the checks fails,
// the flag field will have the error bit set.
//
//-----------------------------------------------------------------------------------------------------------
uint8_t LcsBlockTrack::setupBlockTrack( LcsBlockTrackDesc* trackDesc ) {

    if ((  trackDesc -> selPin1      == CDC::UNDEFINED_PIN ) ||
        (  trackDesc -> selPin2      == CDC::UNDEFINED_PIN ) ||
        (  trackDesc -> sensePin     == CDC::UNDEFINED_PIN )) {

        flags = BT_F_CONFIG_ERROR;
        return ( ERR_PIN_CONFIG );
    }

    if (( trackDesc -> initCurrentMilliAmp   > trackDesc -> limitCurrentMilliAmp )                    ||
        ( trackDesc -> limitCurrentMilliAmp > trackDesc -> maxCurrentMilliAmp )                       ||
        ( trackDesc -> startTimeThresholdMillis > MAX_START_TIME_THRESHOLD_MILLIS )                   ||
        ( trackDesc -> stopTimeThresholdMillis > MAX_STOP_TIME_THRESHOLD_MILLIS )                     ||
        ( trackDesc -> overloadTimeThresholdMillis > MAX_OVERLOAD_TIME_THRESHOLD_MILLIS )             ||
        ( trackDesc -> overloadEventThreshold > MAX_OVERLOAD_EVENT_COUNT )                            ||
        ( trackDesc -> overloadRestartThreshold > MAX_OVERLOAD_RESTART_COUNT )
        ) {

        flags = BT_F_CONFIG_ERROR;
        return ( ERR_TRACK_CONFIG );
    }

    trackState                  = TRACK_POWER_OFF;
    flags                       = BT_F_DEFAULT_SETTING;
    options                     = trackDesc -> options;
    selPin1                     = trackDesc -> selPin1;
    selPin2                     = trackDesc -> selPin2;
    sensePin                    = trackDesc -> sensePin;
    pwmFrequency                = trackDesc -> pwmFrequency;
    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.
//
```

```
//  TRACK_POWER_START1    - this is the first state of a start sequence. When the track should be powered
//                          on, the first activity is to set the status flags and enable the power module.
//                          We set the power module current consumption to the initial limit configured.
//                          The next state is TRACK_POWER_START2.
//
//  TRACK_POWER_START2    - we stay in this state until the threshold time has passed. Once the threshold
//                          is reached, the current consumption limit is set to the configured limit.
//                          Then we move on to TRACK_POWER_ON.
//
//  TRACK_POWER_ON        - this is the state when power is on and things are running normal. An overload
//                          situation is set by the current measurement routines through setting the
//                          overload status flag. We make sure that we have seen a couple of overloads
//                          in a row before taking action which is to turn power off and set the
//                          TRACK_POWER_OVERLOAD state. Otherwise we stay in this state.
//
//  TRACK_POWER_OVERLOAD  - with power turned off, we stay in this state until the threshold time has
//                          passed. If passed, the overload restart count is incremented and checked for
//                          its threshold. If reached, we have tried to restart several times and failed.
//                          The track state becomes TRACK_POWER_STOP1, something is wrong on the track.
//                          If not, we move on to TRACK_POWER_START1.
//
//  TRACK_POWER_STOP1     - this state initiates a shutdown sequence. We disable the power module, set
//                          status flags and advance to the TRACK_POWER_STOP2 state.
//
//  TRACK_POWER_STOP2     - we stay in this state until the configured threshold has passed. Then we move
//                          on to TRACK_POWER_OFF. The key reason for this time delay is to implement
//                          the requirement that track turned off and perhaps switched to another mode,
//                          should be powerless for one second. Switch track modes becomes simply a matter
//                          of stopping and then starting again.
//
//  TRACK_POWER_OFF       - the track is disabled. We just stay in this state until the state is set to
//                          a different state from outside.
//
// During the power on state, we append the actual current measurement value to a circular buffer when the
// time interval for this kind of measurement has passed. The idea is to measure the samples at a more or
// less constant interval rate and compute the power consumption RMS value from the data in the buffer when
// requested. In the interest of minimizing the controller load, the calculation is done in digit values
// the result is presented in then in milliAmps.
//
//---------------------------------------------------------------------------------------------------------
void LcsBlockTrack::runTrackStateMachine( ) {

    switch ( trackState ) {

        case TRACK_POWER_START1: {

            // ??? do we need a way to check for overload during this initial phase, just like we do when ON ?

            trackTimeStamp          = CDC::getMillis( );
            flags                   |= BT_F_POWER_ON;
            flags                   &= ~BT_F_POWER_OVERLOAD;
            flags                   &= ~BT_F_MEASUREMENT_ON;
            limitCurrentDigitValue  = milliAmpToDigitValue( initCurrentMilliAmp, digitsPerAmp );

            setTrackMode( initialTrackMode, 0 );
            trackState = TRACK_POWER_START2;

        }  break;

        case TRACK_POWER_START2: {

            if (( CDC::getMillis( ) - trackTimeStamp ) > startTimeThreshold ) {

                highWaterMarkDigitValue = 0;
                actualCurrentDigitValue = 0;
                overloadRestartCount    = 0;
                overloadEventCount      = 0;
                flags                   |= BT_F_POWER_ON | BT_F_MEASUREMENT_ON;
                limitCurrentDigitValue  = milliAmpToDigitValue( limitCurrentMilliAmp, digitsPerAmp );

                trackState = TRACK_POWER_ON;
            }

        } break;

        case TRACK_POWER_ON: {

            if (( CDC::getMillis( ) - lastPwrSampleTimeStamp ) > PWR_SAMPLE_TIME_INTERVAL_MILLIS ) {

                pwrSampleBuf[ pwrSampleBufIndex % TRACK_POWER_ON ] = actualCurrentDigitValue;
                pwrSampleBufIndex ++;
                lastPwrSampleTimeStamp = CDC::getMillis( );
            }

            if (( CDC::getMillis( ) - lastPwrSamplePerSecTimeStamp ) > 1000 ) {

                pwrSamplesPerSec           = totalPwrSamplesTaken - lastPwrSamplePerSecTaken;
                lastPwrSamplePerSecTaken   = totalPwrSamplesTaken;
                lastPwrSamplePerSecTimeStamp = CDC::getMillis( );
            }

            if ( flags & BT_F_POWER_OVERLOAD ) {

                overloadEventCount  ++;

                if ( overloadEventCount > overloadEventThreshold ) {

                    if (( debugMask & DBG_BC_CONFIG ) && ( debugMask & DBG_BC_TRACK_POWER_MGMT )) {

```

```
                        printf( "Overload detected: " );

                        #if 0
                        printf( "(hwm(mA): %d : limit(mA): %d )\n",
                                digitValueToMilliAmp( highWaterMarkDigitValue, digitsPerAmp ),
                                digitValueToMilliAmp( limitCurrentDigitValue, digitsPerAmp ));
                        #else
                        printf( "(hwm(dVal): %d  : limit(dVal): %d )\n", highWaterMarkDigitValue, limitCurrentDigitValue );
                        #endif
                    }

                    trackTimeStamp  = CDC::getMillis( );
                    flags           |= BT_F_POWER_OVERLOAD;
                    flags           &= ~BT_F_POWER_ON;
                    flags           &= ~BT_F_MEASUREMENT_ON;

                    setTrackMode( BT_MODE_OFF );
                    trackState = TRACK_POWER_OVERLOAD;
                }
            }

        }  break;

        case TRACK_POWER_OVERLOAD: {

            if ( CDC::getMillis( ) - trackTimeStamp > overloadTimeThreshold ) {

                overloadRestartCount ++;

                if ( overloadRestartCount > overloadRestartThreshold ) {

                    if (( debugMask & DBG_BC_CONFIG ) && ( debugMask & DBG_BC_TRACK_POWER_MGMT )) {

                        printf( "Overload restart failed, Cnt:%d\n", overloadRestartCount );
                    }

                    trackState = TRACK_POWER_STOP1;
                }
                else trackState = TRACK_POWER_START1;
            }

        }  break;

        case TRACK_POWER_STOP1: {

            trackTimeStamp  = CDC::getMillis( );
            flags           &= ~BT_F_POWER_ON;
            flags           &= ~BT_F_POWER_OVERLOAD;
            flags           &= ~BT_F_MEASUREMENT_ON;

            setTrackMode( BT_MODE_OFF );
            trackState = TRACK_POWER_STOP2;

        }  break;

        case TRACK_POWER_STOP2: {

            if ( CDC::getMillis( ) - trackTimeStamp > stopTimeThreshold ) trackState = TRACK_POWER_OFF;

        } break;

        case TRACK_POWER_OFF: {

        } break;
    }
}

//----------------------------------------------------------------------------------------------------
// Some getter functions. Straightforward.
//
//----------------------------------------------------------------------------------------------------
uint16_t LcsBlockTrack::getFlags( ) {

    return ( flags );
}

uint16_t LcsBlockTrack::getOptions( ) {

    return ( options );
}

uint32_t LcsBlockTrack::getPwrSamplesTaken( ) {

    return ( totalPwrSamplesTaken );
}

uint16_t LcsBlockTrack::getPwrSamplesPerSec( ) {

    return ( pwrSamplesPerSec );
}

bool LcsBlockTrack::isPowerOn( ) {

    return ( flags & BT_F_POWER_ON );
}

bool LcsBlockTrack::isPowerOverload( ) {

    return ( flags & BT_F_POWER_OVERLOAD );
```

```
}

//----------------------------------------------------------------------------------------------------------
// Track power management functions. The actual state of track power is kept in the track status field
// and can be queried or set by setting the respective flag. Starting and stopping track power is done by
// setting the respective START or STOP state.
//
//----------------------------------------------------------------------------------------------------------
void LcsBlockTrack::powerStart( ) {

    trackState = TRACK_POWER_START1;
}

void LcsBlockTrack::powerStop( ) {

    trackState = TRACK_POWER_STOP1;
}

//----------------------------------------------------------------------------------------------------------
// Power Consumption Management. There are two key values. The first is the actual current consumption as
// measured by the ADC hardware on each ZERO DCC bit. This value is used to do the power overload checking.
// The second value is the high water mark built from these measurements. This values is used for the DCC
// decoder programming logic. The high water mark will be set to zero before collecting measurements. All
// measurement values are actually ADC digit values for performance reason. Only on limit setting and external
// data access are these values converted from and to milliAmps.
//
//----------------------------------------------------------------------------------------------------------
uint16_t LcsBlockTrack::getLimitCurrent( ) {

    return ( limitCurrentMilliAmp );
}

uint16_t LcsBlockTrack::getActualCurrent( ) {

    return ( digitValueToMilliAmp( actualCurrentDigitValue, digitsPerAmp ));
}

uint16_t LcsBlockTrack::getInitCurrent( ) {

    return ( initCurrentMilliAmp );
}

uint16_t LcsBlockTrack::getMaxCurrent( ) {

    return ( maxCurrentMilliAmp );
}

void LcsBlockTrack::setLimitCurrent( uint16_t val ) {

    if      ( val < initCurrentMilliAmp )  val = initCurrentMilliAmp;
    else if ( val > maxCurrentMilliAmp  )  val = maxCurrentMilliAmp;

    limitCurrentMilliAmp    = val;
    limitCurrentDigitValue  = milliAmpToDigitValue( val, digitsPerAmp );
}

//----------------------------------------------------------------------------------------------------------
// The "getRMSCurrent" function returns the power consumption based on the samples taken and stored in the
// sample buffer. The function computes the square root of the sum of the squares of the array elements. The
// result is returned in milliAmps. Note that our measurement is based on unsigned 16-bit quantities that come
// from the controller ADC hardware. We compute the RMS based on 16-bit unsigned integers, which compared
// to floating point computation is not really precise. However, for our purpose to just show a rough power
// consumption, the error should be not a big issue. We will not use RMS values for power overload detection
// or decoder ACK detection.
//
//----------------------------------------------------------------------------------------------------------
uint16_t LcsBlockTrack::getRMSCurrent( ) {

    uint32_t res = 0;

    for ( uint8_t i = 0; i < PWR_SAMPLE_BUF_SIZE; i++ ) res += pwrSampleBuf[ i ] * pwrSampleBuf[ i ];

    return ( digitValueToMilliAmp( sqrt( res / PWR_SAMPLE_BUF_SIZE ), digitsPerAmp ));
}

//----------------------------------------------------------------------------------------------------------
// This function is called whenever we want to measure the power consumption. Typically this routine will be
// called from an timer or interrupt handler.
//
//----------------------------------------------------------------------------------------------------------
void LcsBlockTrack::powerMeasurement( ) {

    if ( flags & BT_F_MEASUREMENT_ON ) {

        actualCurrentDigitValue = CDC::readAdc( sensePin );

        totalPwrSamplesTaken ++;

        if ( actualCurrentDigitValue > highWaterMarkDigitValue ) highWaterMarkDigitValue = actualCurrentDigitValue;
        if ( actualCurrentDigitValue > limitCurrentDigitValue ) flags |= BT_F_POWER_OVERLOAD;
    }
}

//----------------------------------------------------------------------------------------------------------
// Print out the DCC Track configuration data. For debugging purposes.
//
//----------------------------------------------------------------------------------------------------------
void LcsBlockTrack::printTrackConfig( ) {

```

```
    printf( "DccTrack Config: " );

    printf( "Config options: ( 0x%x ) -> ", flags );
    if ( options & BT_OPT_RAILCOM ) printf( "Railcom " );
    printf( "\n" );

    printf( "Sel1 Pin: %d, Sel2 Pin: %d, Sensor Pin: %d\n", selPin1, selPin2, sensePin );

    printf( "Initial Block State: %d, speed: %d\n", initialTrackMode, initialTrackSpeed );

    printf( "Current Initial(mA): %d Current Limit(mA): %d Current Max(mA): %d\n",
            getInitCurrent( ), getLimitCurrent( ), getMaxCurrent( ));

    printf( "milliVoltPerAmp: %d\n", milliVoltPerAmp );
    printf( "digitsPerAmp: %d\n", digitsPerAmp );
    printf( "Limit Digit Value: %d\n", limitCurrentDigitValue );
}

//----------------------------------------------------------------------------------------------------------
// Print out the DCC Track status.
//
//----------------------------------------------------------------------------------------------------------
void LcsBlockTrack::printTrackStatus( ) {

    printf( ", Track Status: ( 0x%x ) -> ", flags );

    if ( flags & BT_F_POWER_ON          ) printf( "PowerOn " );
    if ( flags & BT_F_POWER_OVERLOAD    ) printf( "PowerOverload " );
    if ( flags & BT_F_MEASUREMENT_ON    ) printf( "PowerMeasOn " );
    if ( flags & BT_F_CONFIG_ERROR      ) printf( "ConfigError " );
    printf( "\n" );

    printf( "Total Power Samples: %d\n", totalPwrSamplesTaken );
    printf( "Power Samples per Sec: %d\n", pwrSamplesPerSec );
    printf( "Power consumption (RMS): %d\n", getRMSCurrent( ));
    printf( "\n" );
}
```

```
//-------------------------------------------------------------------------------------------------------------
//
// LCS 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,
```

```

    INV,    INV,    INV,    RSV1,   INV,    0x2B,   0x30,   INV,    // c
    INV,    0x2A,   0x2F,   INV,    0x31,   INV,    INV,    INV,

    INV,    0x29,   0x2E,   INV,    0x2D,   INV,    INV,    INV,    // d
    0x2C,   INV,    INV,    INV,    INV,    INV,    INV,    INV,

    INV,    RSV3,   0x28,   INV,    0x27,   INV,    INV,    INV,    // e
    0x26,   INV,    INV,    INV,    INV,    INV,    INV,    INV,

    ACK,    INV,    INV,    INV,    INV,    INV,    INV,    INV,    // f
    INV,    INV,    INV,    INV,    INV,    INV,    INV,    INV,
};

//-----------------------------------------------------------------------------------------------------------
// Railcom datagrams are sent from a mobile or a stationary decoder.
//
//-----------------------------------------------------------------------------------------------------------
enum railComDatagramType : uint8_t {

    RX_DG_TYPE_UNDEFINED  = 0,
    RC_DG_TYPE_MOB        = 1,
    RC_DG_TYPE_STAT       = 2
};

//-----------------------------------------------------------------------------------------------------------
// Each mobile decoder railcom datagram will start with an ID field of four bits. Channel one will use only
// the ADR_HIG and ADR_LOW Ids. All IDs can be used for channel 2. Since decoders answer on channel one
// for each DCC packet they receive, here is a good chance that channel 1 will contains nonsense data. This
// is different for channel two, where only the addressed decoder explicitly answers. To decide whether
// a railcom message is valid, you should perhaps ignore channel 1 data and just check channel 2 for this
// purpose. A RC datagram starts with the 4-bit ID and an 8 to 32bit payload.
//
//      RC_DG_MOB_ID_POM       ( 0  )  - 12bit
//      RC_DG_MOB_ID_ADR_HIGH  ( 1  )  - 12bit
//      RC_DG_MOB_ID_ADR_LOW   ( 2  )  - 12bit
//      RC_DG_MOB_ID_APP_EXT   ( 3  )  - 18bit
//      RC_DG_MOB_ID_APP_DYN   ( 7  )  - 18bit
//      RC_DG_MOB_ID_XPOM_1    ( 8  )  - 36bit
//      RC_DG_MOB_ID_XPOM_2    ( 9  )  - 36bit
//      RC_DG_MOB_ID_XPOM_3    ( 10 )  - 36bit
//      RC_DG_MOB_ID_XPOM_4    ( 11 )  - 36bit
//      RC_DG_MOB_ID_TEST      ( 12 )  - ignore
//      RC_DG_MOB_ID_SEARCH    ( 14 )  - 48bit
//
// A datagram with the ID 14 is a DDC-A datagram and all 8 datagram bytes are combined to an 48bit datagram.
// A datagram packet can also contain more than one datagram. For example there could be two 18-bit length
// datagram in one packet or 3 12-bit packets and so on. Finally, unused bytes in channel two could contain
// an ACK to fill them up.
//
//-----------------------------------------------------------------------------------------------------------
enum railComDatagramMobId : uint8_t {

    RC_DG_MOB_ID_POM        = 0,
    RC_DG_MOB_ID_ADR_HIGH   = 1,
    RC_DG_MOB_ID_ADR_LOW    = 2,
    RC_DG_MOB_ID_APP_EXT    = 3,
    RC_DG_MOB__IDAPP_DYN    = 7,
    RC_DG_MOB_ID_XPOM_1     = 8,
    RC_DG_MOB_ID_XPOM_2     = 9,
    RC_DG_MOB_ID_XPOM_3     = 10,
    RC_DG_MOB_ID_XPOM_4     = 11,
    RC_DG_MOB_ID_TEST       = 12,
    RC_DG_MOB_ID_SEARCH     = 14
};

//-----------------------------------------------------------------------------------------------------------
// Similar to the mobile decode, a stationary decoder datagram will start an ID field of four bits. Stationary
// decoders also define a datagram with "SRQ" and no ID field to request service from the base station.
//
// ??? to fill in ...
//
//      RC_DG_STAT_ID_SRQ       ( 0  )  - 12bit
//      RC_DG_STAT_ID_POM       ( 1  )  - 12bit
//      RC_DG_STAT_ID_STAT1     ( 4  )  - 12bit
//      RC_DG_STAT_ID_TIME      ( 5  )  - xxbit
//      RC_DG_STAT_ID_ERR       ( 6  )  - xxbit
//      RC_DG_STAT_ID_XPOM_1    ( 8  )  - 36bit
//      RC_DG_STAT_ID_XPOM_2    ( 9  )  - 36bit
//      RC_DG_STAT_ID_XPOM_3    ( 10 )  - 36bit
//      RC_DG_STAT_ID_XPOM_4    ( 11 )  - 36bit
//      RC_DG_STAT_ID_TEST      ( 12 )  - ignore
//
//-----------------------------------------------------------------------------------------------------------
enum railComDatagramStatId : uint8_t {

    RC_DG_STAT_ID_SRQ       = 0,
    RC_DG_STAT_ID_POM       = 1,
    RC_DG_STAT_ID_STAT1     = 4,
    RC_DG_STAT_ID_TIME      = 5,
    RC_DG_STAT_ID_ERR       = 6,
    RC_DG_STAT_ID_DYN       = 7,
    RC_DG_STAT_ID_XPOM_1    = 8,
    RC_DG_STAT_ID_XPOM_2    = 9,
    RC_DG_STAT_ID_XPOM_3    = 10,
    RC_DG_STAT_ID_XPOM_4    = 11,
    RC_DG_STAT_ID_TEST      = 12
};

```

```
//--------------------------------------------------------------------------------------------------------
// The RailCom buffer size. During the cutout period up to eight bytes of raw data are sent by the decoder if
// the Railcom option is enabled.
//
//--------------------------------------------------------------------------------------------------------
const uint8_t   RAILCOM_BUF_SIZE = 8;



struct RailCom {



    void                        startRailComIO( );
    void                        stopRailComIO( );
    uint8_t                     handleRailComMsg( );
    uint8_t                     getRailComMsg( uint8_t *buf, uint8_t bufLen );

};




//--------------------------------------------------------------------------------------------------------
// Railcom. If the cutout period and the RailCom feature is enabled, the signal state machine will also start
// and stop the UART reader for RailCom data. The final message is then to handle that message. In the cutout
// period, a decoder sends 8 data bytes. They are divided into two channels, 2bytes and another 6 bytes. The
// bytes themselves are encoded such that each byte has four bits set, i.e. a hamming weight of 4. The first
// channel is used to just send the locomotive address when the decoder is addressed. The second channel is
// used only when the decoder is explicitly addressed via a CV operation command to provide the answer to the
// request.
//
// The received datagrams are also recorded in the DCC_LOG, if enabled.
//
// ??? under construction....
// ??? we could store the last loco address in some global variable.
// ??? we could store the channel 2 datagram in the corresponding session.
// ??? still, both pieces of data needs to go somewhere before the next message is received...
//--------------------------------------------------------------------------------------------------------
void LcsBaseStationDccTrack::startRailComIO( ) {

    CDC::startUartRead( uartRxPin );
}

void LcsBaseStationDccTrack::stopRailComIO( ) {

    CDC::stopUartRead( uartRxPin );
}

uint8_t LcsBaseStationDccTrack::handleRailComMsg( ) {

    railComBufIndex = CDC::getUartBuffer( uartRxPin, railComMsgBuf, sizeof( railComMsgBuf ));

    writeLogData( LOG_DCC_RCM, railComMsgBuf, railComBufIndex );

    for ( uint8_t i = 0; i < railComBufIndex; i++ ) {

        uint8_t dataByte = railComDecode[ railComMsgBuf[ i ]];

        if      ( dataByte == ACK ) ;
        else if ( dataByte == NACK ) ;
        else if ( dataByte == BUSY ) ;
        else if ( dataByte < 64 ) {

            // ??? valid
            // ??? a railCom message can have multiple datagrams
            // we would need to handle each datagram, one at a time or fill them into a kind of structure
            // that has a slot for the up to maximum 4 datagrams per railCom cutout period.
        }
        else {

            // ??? invalid packet ... if this is channel2, discard the entire message.
        }

        railComMsgBuf[ i ] = dataByte;
    }

    flags &= ~ DT_F_RAILCOM_MSG_PENDING;
    return ( ALL_OK );
}

// ??? not very useful, but good for debugging and initial testing .... and it works like a champ :-)

uint8_t LcsBaseStationDccTrack::getRailComMsg( uint8_t *buf, uint8_t bufLen ) {

    if (( railComBufIndex > 0 ) && ( bufLen > 0 )) {

        uint8_t i = 0;

        do {

            buf[ i ] = railComMsgBuf[ i ];
            i++;

        } while (( i < railComBufIndex ) && ( i < bufLen ));

        return ( i );

    } else return ( 0 );
```

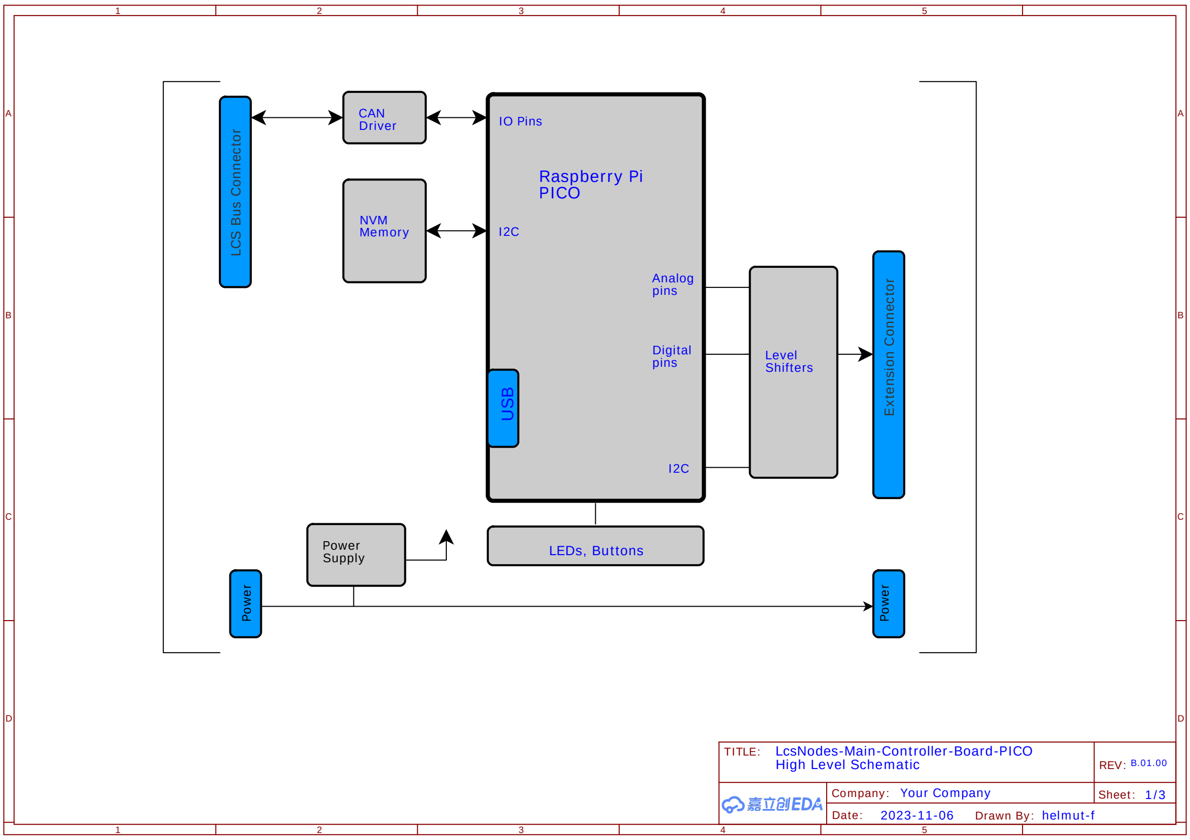
```
}

#endif
```

B Tests

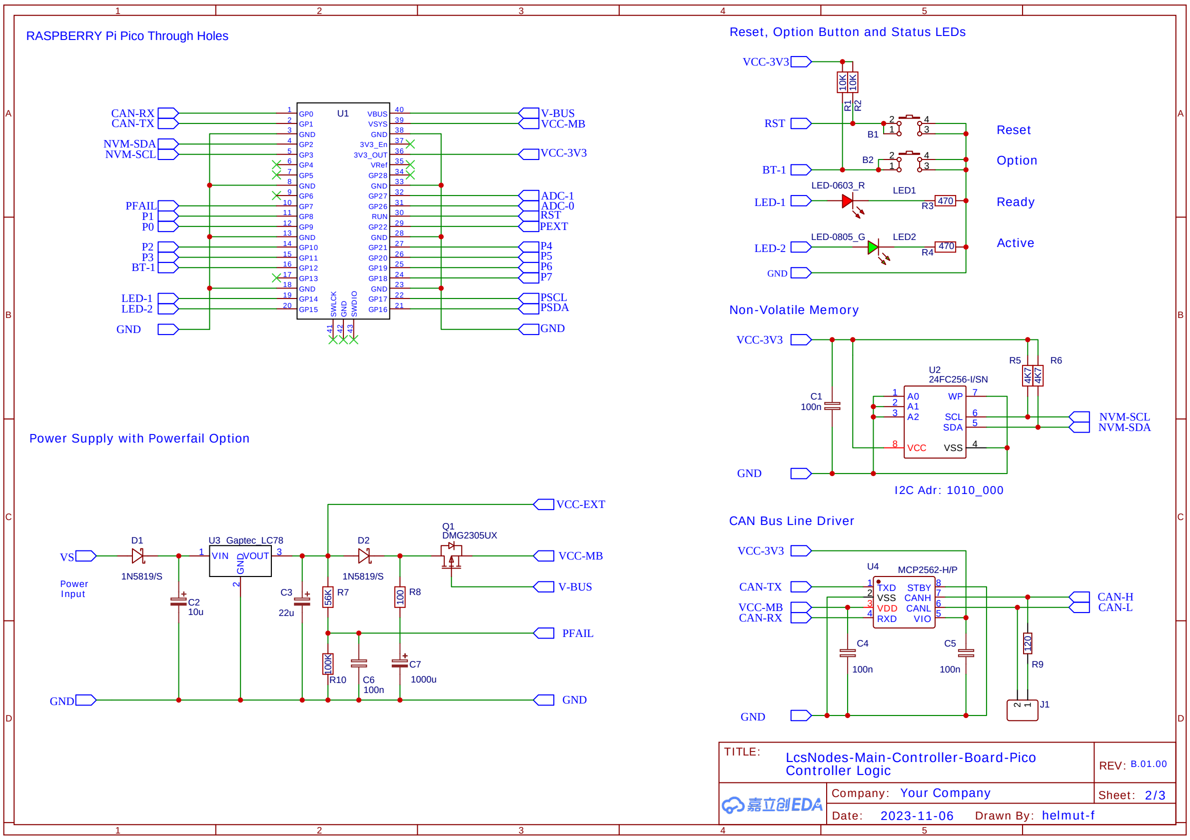
B.1 Schematics

float barrier command to ensure that text stays close to the picture but no text from after the picture.

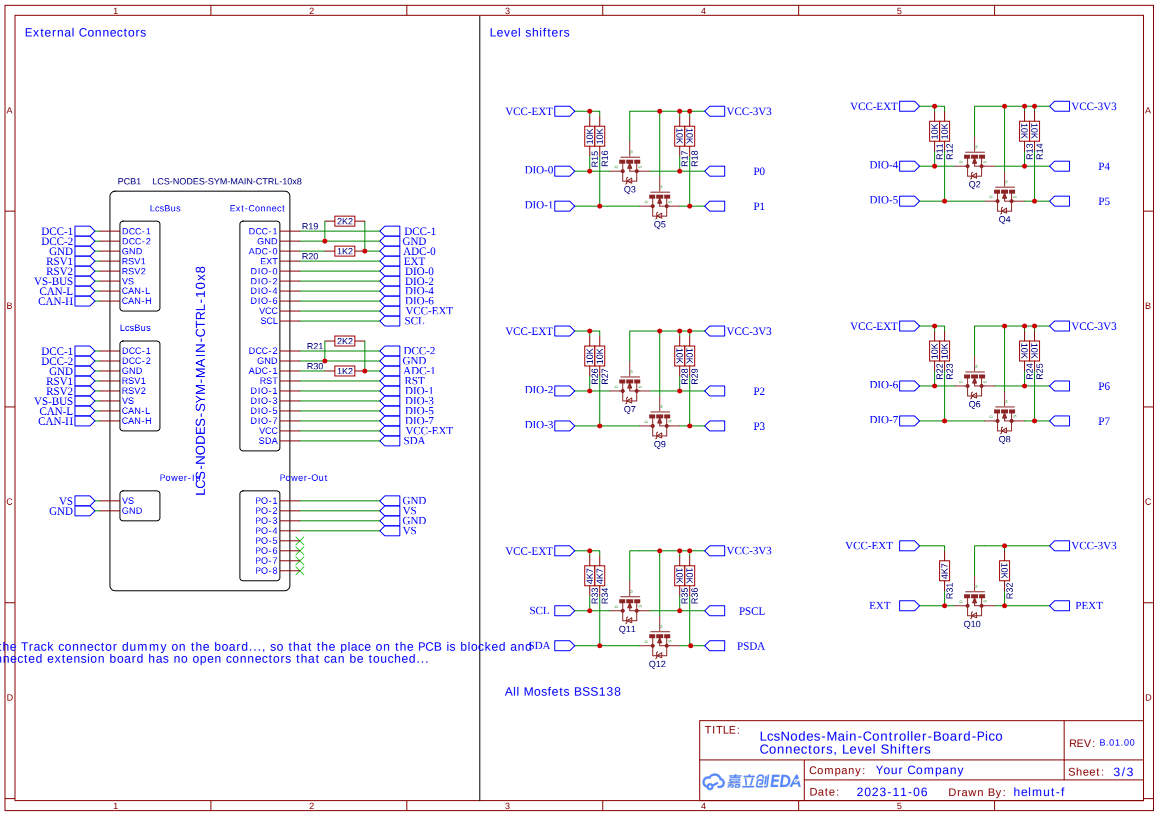
B.1.1 part 1



B.1.2 part 2


RASPBERRY Pi Pico Through Holes
Reset, Option Button and Status LEDs
Reset
Option
Ready
Active
Non-Volatile Memory
I2C Adr: 1010_000
Power Supply with Powerfail Option
CAN Bus Line Driver
TITLE: LcsNodes-Main-Controller-Board-Pico Controller Logic
REV: B.01.00
Company: Your Company
Sheet: 2/3
Date: 2023-11-06
Drawn By: helmut-f

B.1.3 part 3



B.2 Lists

B.2.1 A simple list

- First bullet point
- Second bullet point
- Third bullet point

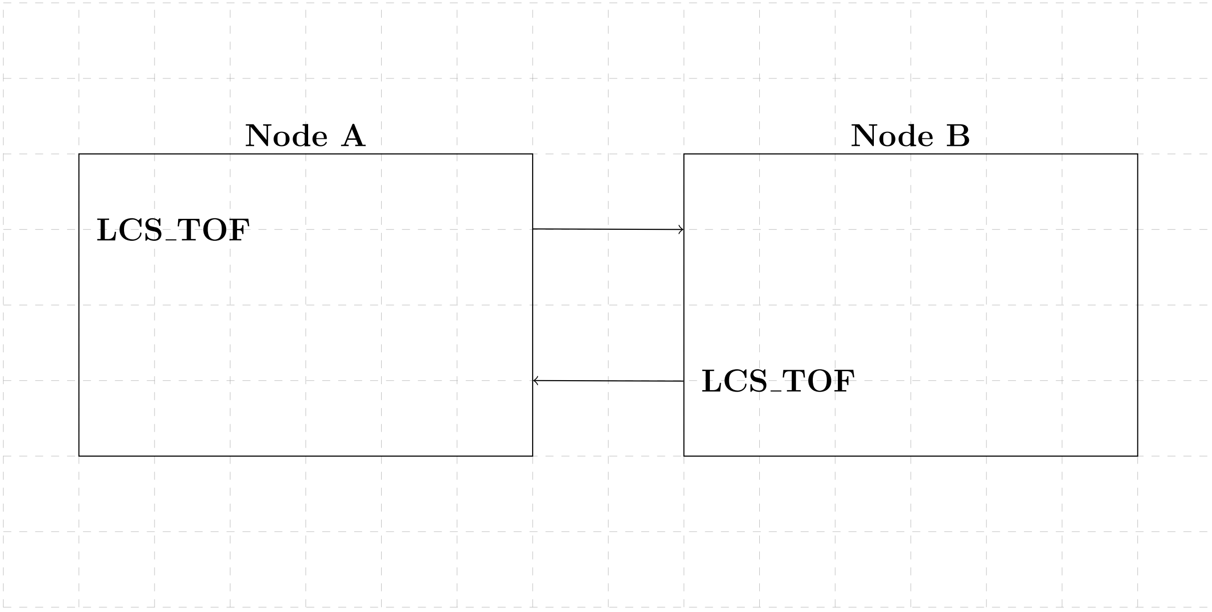
B.2.2 An instruction word layout

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.

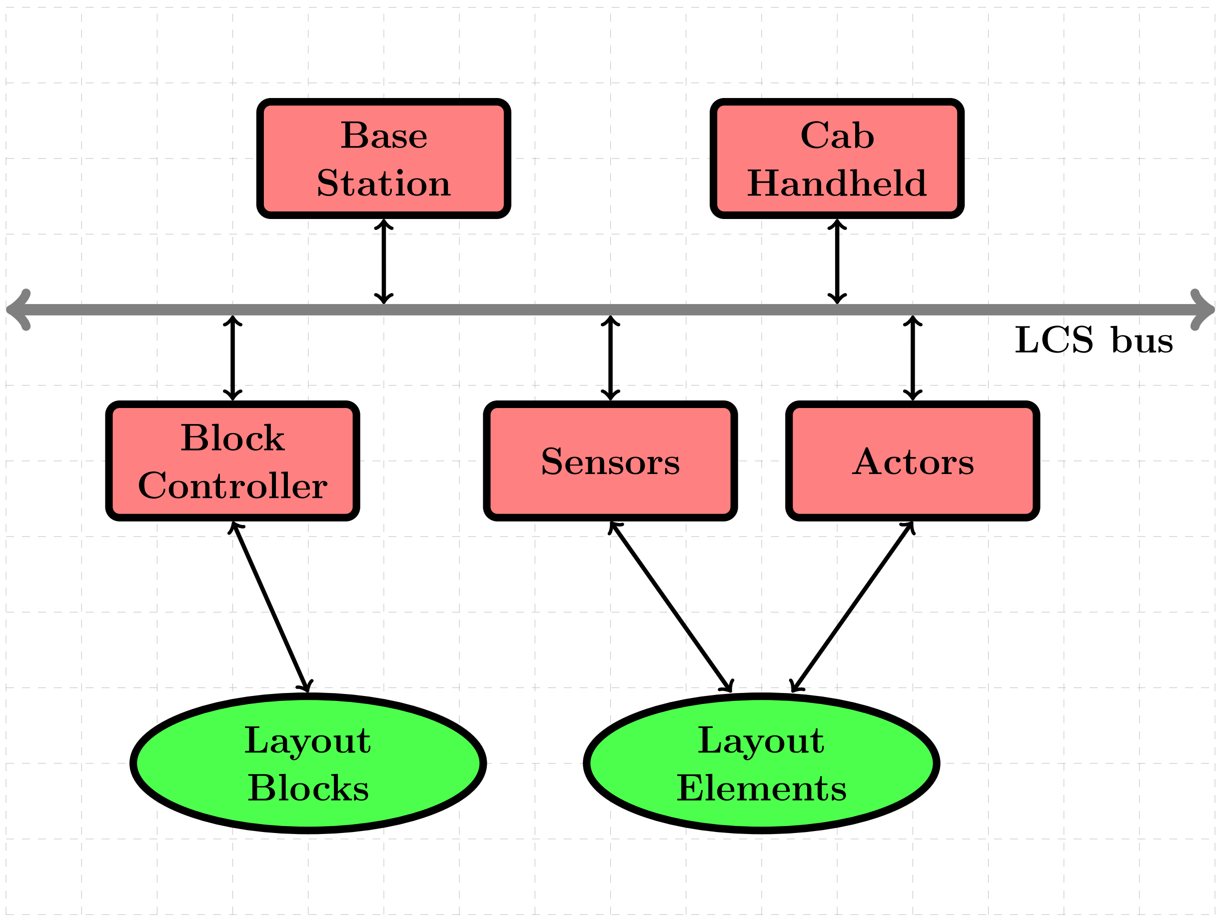


B.4 Split rectangle

We would need the split rectangle for the runtime area maps....

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