```
/**          Printed to PDF by using notepad++
 *
 * @copyright &copy; 2010 - 2020, Fraunhofer-Gesellschaft zur Foerderung der
 *  angewandten Forschung e.V. All rights reserved.
 *
 * BSD 3-Clause License
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 * 1.  Redistributions of source code must retain the above copyright notice,
 *     this list of conditions and the following disclaimer.
 * 2.  Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in the
 *     documentation and/or other materials provided with the distribution.
 * 3.  Neither the name of the copyright holder nor the names of its
 *     contributors may be used to endorse or promote products derived from
 *     this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *
 * We kindly request you to use one or more of the following phrases to refer
 * to foxBMS in your hardware, software, documentation or advertising
 * materials:
 *
 * &Prime;This product uses parts of foxBMS&reg;&Prime;
 *
 * &Prime;This product includes parts of foxBMS&reg;&Prime;
 *
 * &Prime;This product is derived from foxBMS&reg;&Prime;
 *
 */

/**
 * @file    ltc.c
 * @author  foxBMS Team
 * @date    01.09.2015 (date of creation)
 * @ingroup DRIVERS
 * @prefix  LTC
 *
 * @brief   Driver for the LTC monitoring chip.
 *
 */

```

```
53  /*================== Includes ============================================*/
54  #include "ltc.h"
55
56  #include "database.h"
57  #include "diag.h"
58  #include "ltc_pec.h"
59  #include "os.h"
60  #include "slaveplausibility.h"
61
62  /*================== Macros and Definitions ==============================*/
63
64  /**
65   * TI port expander register addresses
66   *
67   */
68
69  #define LTC_PORT_EXPANDER_TI_INPUT_REG_ADR    0x00
70  #define LTC_PORT_EXPANDER_TI_OUTPUT_REG_ADR   0x01
71  #define LTC_PORT_EXPANDER_TI_CONFIG_REG_ADR   0x03
72
73  /**
74   * LTC COMM definitions
75   */
76
77  #define LTC_ICOM_START              0x60
78  #define LTC_ICOM_STOP               0x10
79  #define LTC_ICOM_BLANK              0x00
80  #define LTC_ICOM_NO_TRANSMIT        0x70
81  #define LTC_FCOM_MASTER_ACK         0x00
82  #define LTC_FCOM_MASTER_NACK        0x08
83  #define LTC_FCOM_MASTER_NACK_STOP   0x09
84
85  #define LTC_MAX_SUPPORTED_CELLS         12
86
87  /**
88   * Saves the last state and the last substate
89   */
90  #define LTC_SAVELASTSTATES()    ltc_state.laststate = ltc_state.state; \
91                                  ltc_state.lastsubstate = ltc_state.substate
92
93  /*================== Constant and Variable Definitions ===================*/
94
95  static uint8_t ltc_used_cells_index = 0;
96
97  static DATA_BLOCK_CELLVOLTAGE_s ltc_cellvoltage;
98  static DATA_BLOCK_CELLTEMPERATURE_s ltc_celltemperature;
99  static DATA_BLOCK_MINMAX_s ltc_minmax;
100 static DATA_BLOCK_BALANCING_FEEDBACK_s ltc_balancing_feedback;
101 static DATA_BLOCK_USER_MUX_s ltc_user_mux;
102 static DATA_BLOCK_BALANCING_CONTROL_s ltc_balancing_control;
103 static DATA_BLOCK_SLAVE_CONTROL_s ltc_slave_control;
104 static DATA_BLOCK_ALLGPIOVOLTAGE_s ltc_allgpiovoltage;
```

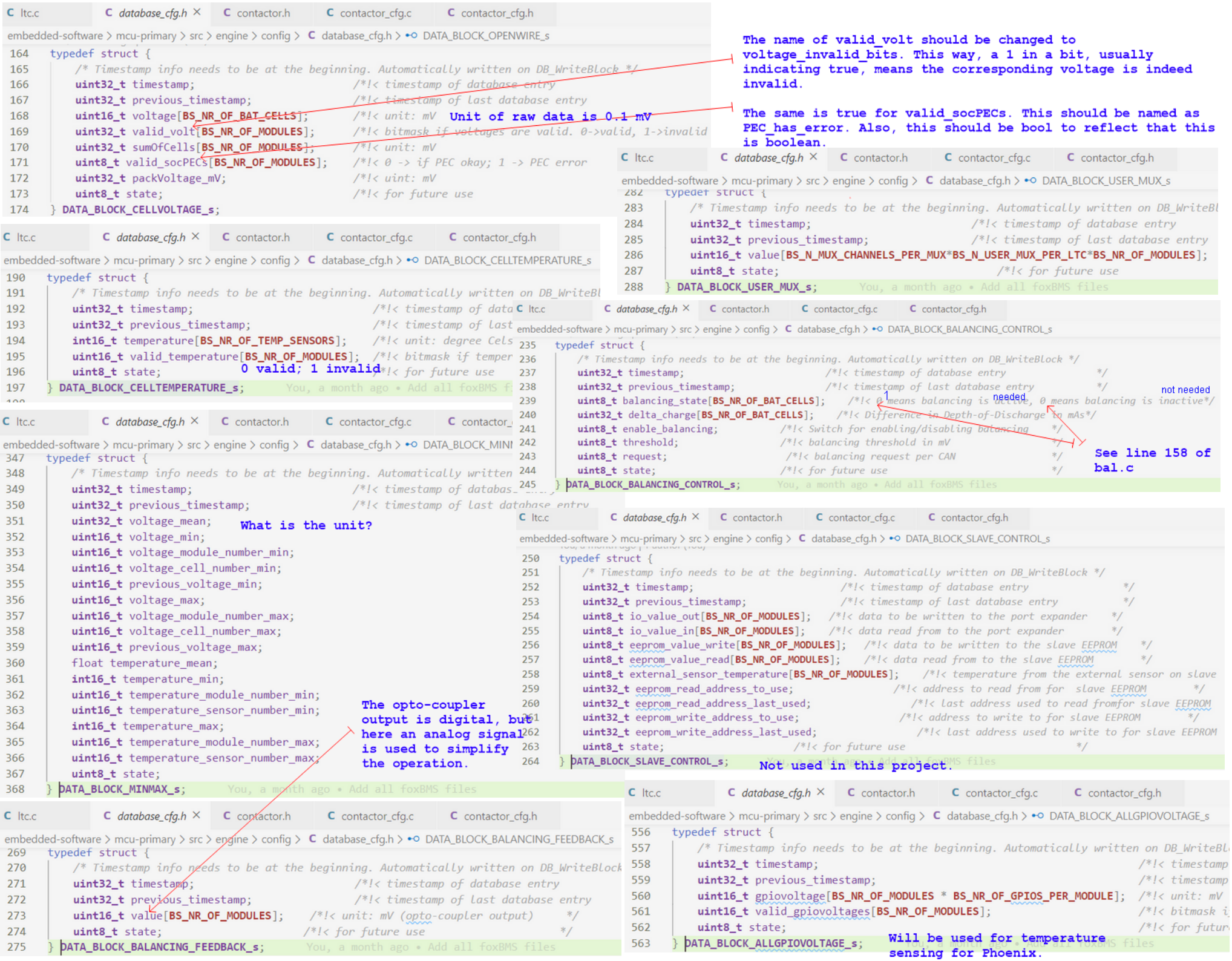
These ports are not used in the Phoenix battery module, but they can be useful for others. For details on the devices, see http://www.ti.com/interface/i2c/general-purpose-ios-gpios/products.html

LTC comm is not used in the Phoenix battery module. They are useful to control the above port expnaders or analog multiplexers. Analog multiplexers can be used in the next version of battery module where multiple thermisters can be installed on the battery signal board.

This should be defined in some configuration file.

This can be written as a common function with one parameter, the pointer of "ltc_state" here. This way, this function can be called in all modules.

Data structures are given on the next page.


ltc.c
database_cfg.h
contactor.h
contactor_cfg.c
contactor_cfg.h
embedded-software > mcu-primary > src > engine > config > database_cfg.h > DATA_BLOCK_OPENWIRE_s
typedef struct {
/* Timestamp info needs to be at the beginning. Automatically written on DB_WriteBlock */
uint32_t timestamp; /*!< timestamp of database entry
uint32_t previous_timestamp; /*!< timestamp of last database entry
uint16_t voltage[BS_NR_OF_BAT_CELLS]; /*!< unit: mV
uint32_t valid_volt[BS_NR_OF_MODULES]; /*!< bitmask if voltages are valid. 0->valid, 1->invalid
uint32_t sumOfCells[BS_NR_OF_MODULES]; /*!< unit: mV
uint8_t valid_socPECs[BS_NR_OF_MODULES]; /*!< 0 -> if PEC okay; 1 -> PEC error
uint32_t packVoltage_mV; /*!< uint: mV
uint8_t state; /*!< for future use
} DATA_BLOCK_CELLVOLTAGE_s;
Unit of raw data is 0.1 mV
The name of valid_volt should be changed to voltage_invalid_bits. This way, a 1 in a bit, usually indicating true, means the corresponding voltage is indeed invalid.
The same is true for valid_socPECs. This should be named as PEC_has_error. Also, this should be bool to reflect that this is boolean.
embedded-software > mcu-primary > src > engine > config > database_cfg.h > DATA_BLOCK_USER_MUX_s
typedef struct {
/* Timestamp info needs to be at the beginning. Automatically written on DB_WriteBl
uint32_t timestamp; /*!< timestamp of database entry
uint32_t previous_timestamp; /*!< timestamp of last database entry
uint16_t value[BS_N_MUX_CHANNELS_PER_MUX*BS_N_USER_MUX_PER_LTC*BS_NR_OF_MODULES];
uint8_t state; /*!< for future use
} DATA_BLOCK_USER_MUX_s;
You, a month ago • Add all foxBMS files
embedded-software > mcu-primary > src > engine > config > database_cfg.h > DATA_BLOCK_CELLTEMPERATURE_s
typedef struct {
/* Timestamp info needs to be at the beginning. Automatically written on DB_WriteBl
uint32_t timestamp; /*!< timestamp of data
uint32_t previous_timestamp; /*!< timestamp of last
int16_t temperature[BS_NR_OF_TEMP_SENSORS]; /*!< unit: degree Cels
uint16_t valid_temperature[BS_NR_OF_MODULES]; /*!< bitmask if temper
uint8_t state; /*!< for future use
} DATA_BLOCK_CELLTEMPERATURE_s;
You, a month ago • Add all foxBMS f
0 valid; 1 invalid
embedded-software > mcu-primary > src > engine > config > database_cfg.h > DATA_BLOCK_BALANCING_CONTROL_s
typedef struct {
/* Timestamp info needs to be at the beginning. Automatically written on DB_WriteBlock */
uint32_t timestamp; /*!< timestamp of database entry */
uint32_t previous_timestamp; /*!< timestamp of last database entry */
uint8_t balancing_state[BS_NR_OF_BAT_CELLS]; /*!< 0 means balancing is active, 0 means balancing is inactive*/
uint32_t delta_charge[BS_NR_OF_BAT_CELLS]; /*!< Difference in Depth-of-Discharge in mAs*/
uint8_t enable_balancing; /*!< Switch for enabling/disabling balancing */
uint8_t threshold; /*!< balancing threshold in mV */
uint8_t request; /*!< balancing request per CAN */
uint8_t state; /*!< for future use */
} DATA_BLOCK_BALANCING_CONTROL_s;
You, a month ago • Add all foxBMS files
1
needed
not needed
See line 158 of bal.c
embedded-software > mcu-primary > src > engine > config > database_cfg.h > DATA_BLOCK_MINMAX_s
typedef struct {
/* Timestamp info needs to be at the beginning. Automatically written
uint32_t timestamp; /*!< timestamp of databas
uint32_t previous_timestamp; /*!< timestamp of last database entry
uint32_t voltage_mean;
uint16_t voltage_min;
uint16_t voltage_module_number_min;
uint16_t voltage_cell_number_min;
uint16_t previous_voltage_min;
uint16_t voltage_max;
uint16_t voltage_module_number_max;
uint16_t voltage_cell_number_max;
uint16_t previous_voltage_max;
float temperature_mean;
int16_t temperature_min;
uint16_t temperature_module_number_min;
uint16_t temperature_sensor_number_min;
int16_t temperature_max;
uint16_t temperature_module_number_max;
uint16_t temperature_sensor_number_max;
uint8_t state;
} DATA_BLOCK_MINMAX_s;
You, a month ago • Add all foxBMS files
What is the unit?
The opto-coupler output is digital, but here an analog signal is used to simplify the operation.
embedded-software > mcu-primary > src > engine > config > database_cfg.h > DATA_BLOCK_SLAVE_CONTROL_s
typedef struct {
/* Timestamp info needs to be at the beginning. Automatically written on DB_WriteBlock */
uint32_t timestamp; /*!< timestamp of database entry */
uint32_t previous_timestamp; /*!< timestamp of last database entry */
uint8_t io_value_out[BS_NR_OF_MODULES]; /*!< data to be written to the port expander */
uint8_t io_value_in[BS_NR_OF_MODULES]; /*!< data read from to the port expander */
uint8_t eeprom_value_write[BS_NR_OF_MODULES]; /*!< data to be written to the slave EEPROM */
uint8_t eeprom_value_read[BS_NR_OF_MODULES]; /*!< data read from to the slave EEPROM */
uint8_t external_sensor_temperature[BS_NR_OF_MODULES]; /*!< temperature from the external sensor on slave
uint32_t eeprom_read_address_to_use; /*!< address to read from for slave EEPROM */
uint32_t eeprom_read_address_last_used; /*!< last address used to read fromfor slave EEPROM
uint32_t eeprom_write_address_to_use; /*!< address to write to for slave EEPROM */
uint32_t eeprom_write_address_last_used; /*!< last address used to write to for slave EEPROM
uint8_t state; /*!< for future use */
} DATA_BLOCK_SLAVE_CONTROL_s;
Not used in this project.
embedded-software > mcu-primary > src > engine > config > database_cfg.h > DATA_BLOCK_BALANCING_FEEDBACK_s
typedef struct {
/* Timestamp info needs to be at the beginning. Automatically written on DB_WriteBlock
uint32_t timestamp; /*!< timestamp of database entry
uint32_t previous_timestamp; /*!< timestamp of last database entry
uint16_t value[BS_NR_OF_MODULES]; /*!< unit: mV (opto-coupler output) */
uint8_t state; /*!< for future use */
} DATA_BLOCK_BALANCING_FEEDBACK_s;
You, a month ago • Add all foxBMS files
embedded-software > mcu-primary > src > engine > config > database_cfg.h > DATA_BLOCK_ALLGPIOVOLTAGE_s
typedef struct {
/* Timestamp info needs to be at the beginning. Automatically written on DB_WriteBl
uint32_t timestamp; /*!< timestamp
uint32_t previous_timestamp; /*!< timestamp
uint16_t gpiovoltage[BS_NR_OF_MODULES * BS_NR_OF_GPIOS_PER_MODULE]; /*!< unit: mV
uint16_t valid_gpiovoltages[BS_NR_OF_MODULES]; /*!< bitmask i
uint8_t state; /*!< for futur
} DATA_BLOCK_ALLGPIOVOLTAGE_s;
Will be used for temperature sensing for Phoenix.

```
105  static DATA_BLOCK_OPENWIRE_s ltc_openwire;
106  static uint16_t ltc_openwire_pup_buffer[BS_NR_OF_BAT_CELLS];
107  static uint16_t ltc_openwire_pdown_buffer[BS_NR_OF_BAT_CELLS];
108  static int32_t ltc_openwire_delta[BS_NR_OF_BAT_CELLS];
109
110  static LTC_ERRORTABLE_s LTC_ErrorTable[LTC_N_LTC];  /* init in LTC_ResetErrorTable-function */
111
112
113  static LTC_STATE_s ltc_state = {
114      .timer                    = 0,
115      .statereq                 = LTC_STATE_NO_REQUEST,
116      .state                    = LTC_STATEMACH_UNINITIALIZED,
117      .substate                 = 0,
118      .laststate                = LTC_STATEMACH_UNINITIALIZED,
119      .lastsubstate             = 0,
120      .adcModereq               = LTC_ADCMODE_FAST_DCP0,
121      .adcMode                  = LTC_ADCMODE_FAST_DCP0,
122      .adcMeasChreq             = LTC_ADCMEAS_UNDEFINED,
123      .adcMeasCh                = LTC_ADCMEAS_UNDEFINED,
124      .numberOfMeasuredMux      = 32,
125      .triggerentry             = 0,
126      .ErrRetryCounter          = 0,
127      .ErrRequestCounter        = 0,
128      .VoltageSampleTime        = 0,
129      .muxSampleTime            = 0,
130      .commandDataTransferTime  = 3,
131      .commandTransferTime      = 3,
132      .gpioClocksTransferTime   = 3,
133      .muxmeas_seqptr           = NULL_PTR,
134      .muxmeas_seqendptr        = NULL_PTR,
135      .muxmeas_nr_end           = 0,
136      .first_measurement_made   = FALSE,
137      .ltc_muxcycle_finished    = E_NOT_OK,
138      .check_spi_flag           = FALSE,
139      .balance_control_done      = FALSE,
140  };
141
142  static const uint8_t ltc_cmdDummy[1]={0x00};
143  static const uint8_t ltc_cmdWRCFG[4]={0x00, 0x01, 0x3D, 0x6E};      Discharge control is performaed using this command.
144  static const uint8_t ltc_cmdWRCFG2[4]={0x00, 0x24, 0xB1, 0x9E};        WRCFGB, for LTC6813 only
145
146  static const uint8_t ltc_cmdRDCVA[4] = {0x00, 0x04, 0x07, 0xC2};
147  static const uint8_t ltc_cmdRDCVB[4] = {0x00, 0x06, 0x9A, 0x94};
148  static const uint8_t ltc_cmdRDCVC[4] = {0x00, 0x08, 0x5E, 0x52};
149  static const uint8_t ltc_cmdRDCVD[4] = {0x00, 0x0A, 0xC3, 0x04};
150  static const uint8_t ltc_cmdRDCVE[4] = {0x00, 0x09, 0xD5, 0x60};      RDCVE and RDCVF are for LTC6813 only
151  static const uint8_t ltc_cmdRDCVF[4] = {0x00, 0x0B, 0x48, 0x36};
152  static const uint8_t ltc_cmdWRCOMM[4] = {0x07, 0x21, 0x24, 0xB2};
153  static const uint8_t ltc_cmdSTCOMM[4] = {0x07, 0x23, 0xB9, 0xE4};     STart I2C/SPI Comm of LTC681x
154  static const uint8_t ltc_cmdRDCOMM[4] = {0x07, 0x22, 0x32, 0xD6};
155  static const uint8_t ltc_cmdRDAUXA[4] = {0x00, 0x0C, 0xEF, 0xCC};
156  static const uint8_t ltc_cmdRDAUXB[4] = {0x00, 0x0E, 0x72, 0x9A};
```

The contents of this table is assigned in the LTC_RX_PECCheck function starting at Line 2679.

cansignal_cfg.h | cansignal_cfg.c | bms.c | ltc.c | ltc.h

mcu-common > src > module > ltc > ltc.h > LTC_ERRORTABLE_s

```
67  typedef struct {
68      uint8_t PEC_valid;  /*!<    */ Boolean
69      uint8_t mux0;       /*!<    */
70      uint8_t mux1;       /*!<    */
71      uint8_t mux2;       /*!<    */
72      uint8_t mux3;       /*!<    */
73  } LTC_ERRORTABLE_s;      You, a month ago • Add all foxBMS files
```

ltc.c | database_cfg.h | contactor.h | contactor_cfg.c | contactor_cfg.h

embedded-software > mcu-primary > src > engine > config > database_cfg.h > DATA_BLOCK_OPENWIRE_s

```
179  typedef struct {
180      /* Timestamp info needs to be at the beginning. Automatically written on DB_WriteBlock */
181      uint32_t timestamp;                 /*!< timestamp of database entry                    */
182      uint32_t previous_timestamp;        /*!< timestamp of last database entry               */
183      uint8_t openwire[BS_NR_OF_MODULES * (BS_NR_OF_BAT_CELLS_PER_MODULE+1)];   /*!< 1 -> open wire, 0 -> everything ok
184      uint8_t state;                      /*!< for future use                                 */
185  } DATA_BLOCK_OPENWIRE_s;      You, a month ago • Add all foxBMS files
```

```
157 static const uint8_t ltc_cmdRDAUXC[4] = {0x00, 0x0D, 0x64, 0xFE};     RDAUXC and RDAUXD are for LTC6813 only
158 static const uint8_t ltc_cmdRDAUXD[4] = {0x00, 0x0F, 0xF9, 0xA8};
159
160 /* static const uint8_t ltc_cmdMUTE[4] = {0x00, 0x28, 0xE8, 0x0E};                    !< MUTE discharging via S pins */
161 /* static const uint8_t ltc_cmdUNMUTE[4] = {0x00, 0x29, 0x63, 0x3C};                  !< UN-MUTE discharging via S
    pins */                                                  Cannot find this in the command code table of LTC6811
162
163 /* LTC I2C commands */
164 /* static const uint8_t ltc_I2CcmdDummy[6] = {0x7F, 0xF9, 0x7F, 0xF9, 0x7F, 0xF9};       !< dummy command (no
    transmit) */                                                  The slave I2C related code should be in a separate file,
165                                                  How to understand the following 6 byte code?  such as ltc_i2c.c, to reduce clutter.
166 static const uint8_t ltc_I2CcmdTempSens0[6] = {0x69, 0x08, 0x00, 0x09, 0x7F, 0xF9};  /*!< sets the internal data
    pointer of the temperature sensor (address 0x48) to 0x00 */
167 static const uint8_t ltc_I2CcmdTempSens1[6] = {0x69, 0x18, 0x0F, 0xF0, 0x0F, 0xF9};  /*!< reads two data bytes from
    the temperature sensor */
168
169 static const uint8_t ltc_I2CcmdPortExpander1[6] = {0x64, 0x18, 0x0F, 0xF9, 0x7F, 0xF9};  /*!< reads one data byte
    from the port expander */
170
171 /* Cells */  Different modes for the ADCV commands:
172 static const uint8_t ltc_cmdADCV_normal_DCP0[4] = {0x03, 0x60, 0xF4, 0x6C};        /*!< All cells, normal mode,
    discharge not permitted (DCP=0)    */
173 static const uint8_t ltc_cmdADCV_normal_DCP1[4] = {0x03, 0x70, 0xAF, 0x42};        /*!< All cells, normal mode,
    discharge permitted (DCP=1)        */
174 static const uint8_t ltc_cmdADCV_filtered_DCP0[4] = {0x03, 0xE0, 0xB0, 0x4A};      /*!< All cells, filtered mode,
    discharge not permitted (DCP=0)  */
175 static const uint8_t ltc_cmdADCV_filtered_DCP1[4] = {0x03, 0xF0, 0xEB, 0x64};      /*!< All cells, filtered mode,
    discharge permitted (DCP=1)      */
176 static const uint8_t ltc_cmdADCV_fast_DCP0[4] = {0x02, 0xE0, 0x38, 0x06};          /*!< All cells, fast mode,
    discharge not permitted (DCP=0)      */
177 static const uint8_t ltc_cmdADCV_fast_DCP1[4] = {0x02, 0xF0, 0x63, 0x28};          /*!< All cells, fast mode,
    discharge permitted (DCP=1)          */
178 static const uint8_t ltc_cmdADCV_fast_DCP0_twocells[4] = {0x02, 0xE1, 0xb3, 0x34}; /*!< Two cells (1 and 7), fast
    mode, discharge not permitted (DCP=0) */
179
180 /* GPIOs  */  Different modes for the ADAX commands:
181 static const uint8_t ltc_cmdADAX_normal_GPIO1[4] = {0x05, 0x61, 0x58, 0x92};      /*!< Single channel, GPIO 1, normal
    mode   */
182 static const uint8_t ltc_cmdADAX_filtered_GPIO1[4] = {0x05, 0xE1, 0x1C, 0xB4};    /*!< Single channel, GPIO 1,
    filtered mode */
183 static const uint8_t ltc_cmdADAX_fast_GPIO1[4] = {0x04, 0xE1, 0x94, 0xF8};        /*!< Single channel, GPIO 1, fast
    mode     */
184 static const uint8_t ltc_cmdADAX_normal_GPIO2[4] = {0x05, 0x62, 0x4E, 0xF6};      /*!< Single channel, GPIO 2, normal
    mode   */
185 static const uint8_t ltc_cmdADAX_filtered_GPIO2[4] = {0x05, 0xE2, 0x0A, 0xD0};    /*!< Single channel, GPIO 2,
    filtered mode */
186 static const uint8_t ltc_cmdADAX_fast_GPIO2[4] = {0x04, 0xE2, 0x82, 0x9C};        /*!< Single channel, GPIO 2, fast
    mode     */
187 static const uint8_t ltc_cmdADAX_normal_GPIO3[4] = {0x05, 0x63, 0xC5, 0xC4};      /*!< Single channel, GPIO 3, normal
    mode   */
188 static const uint8_t ltc_cmdADAX_filtered_GPIO3[4] = {0x05, 0xE3, 0x81, 0xE2};    /*!< Single channel, GPIO 3,
    filtered mode */
```

```
189 static const uint8_t ltc_cmdADAX_fast_GPIO3[4] = {0x04, 0xE3, 0x09, 0xAE};        /*!< Single channel, GPIO 3, fast
    mode     */
190 /* static const uint8_t ltc_cmdADAX_normal_GPIO4[4] = {0x05, 0x64, 0x62, 0x3E};      !< Single channel, GPIO 4,
    normal mode   */
191 /* static const uint8_t ltc_cmdADAX_filtered_GPIO4[4] = {0x05, 0xE4, 0x26, 0x18};    !< Single channel, GPIO 4,
    filtered mode */
192 /* static const uint8_t ltc_cmdADAX_fast_GPIO4[4] = {0x04, 0xE4, 0xAE, 0x54};        !< Single channel, GPIO 4, fast
    mode     */
193 /* static const uint8_t ltc_cmdADAX_normal_GPIO5[4] = {0x05, 0x65, 0xE9, 0x0C};      !< Single channel, GPIO 5,
    normal mode   */
194 /* static const uint8_t ltc_cmdADAX_filtered_GPIO5[4] = {0x05, 0xE5, 0xAD, 0x2A};    !< Single channel, GPIO 5,
    filtered mode */
195 /* static const uint8_t ltc_cmdADAX_fast_GPIO5[4] = {0x04, 0xE5, 0x25, 0x66};        !< Single channel, GPIO 5, fast
    mode     */
196 static const uint8_t ltc_cmdADAX_normal_ALLGPIOS[4] = {0x05, 0x60, 0xD3, 0xA0};   /*!< All channels, normal
    mode             */
197 static const uint8_t ltc_cmdADAX_filtered_ALLGPIOS[4] = {0x05, 0xE0, 0x97, 0x86}; /*!< All channels, filtered
    mode           */
198 static const uint8_t ltc_cmdADAX_fast_ALLGPIOS[4] = {0x04, 0xE0, 0x1F, 0xCA};     /*!< All channels, fast
    mode              */
199
200 /* Open-wire */      Why ltc2 below?
201 static const uint8_t ltc2_BC_cmdADOW_PUP_normal_DCP0[4] = {0x03, 0x68, 0x1C, 0x62};    /*!< Broadcast, Pull-up
    current, All cells, normal mode, discharge not permitted (DCP=0)    */
202 static const uint8_t ltc2_BC_cmdADOW_PDOWN_normal_DCP0[4] = {0x03, 0x28, 0xFB, 0xE8};  /*!< Broadcast, Pull-down
    current, All cells, normal mode, discharge not permitted (DCP=0) */
203 static const uint8_t ltc2_BC_cmdADOW_PUP_filtered_DCP0[4] = {0x03, 0xE8, 0x1C, 0x62};    /*!< Broadcast, Pull-up
    current, All cells, filtered mode, discharge not permitted (DCP=0)    */
204 static const uint8_t ltc2_BC_cmdADOW_PDOWN_filtered_DCP0[4] = {0x03, 0xA8, 0xFB, 0xE8};  /*!< Broadcast, Pull-down
    current, All cells, filtered mode, discharge not permitted (DCP=0) */
205
206                                                  (4+(8*LTC_N_LTC))
207 static uint8_t ltc_RXPECbuffer[LTC_N_BYTES_FOR_DATA_TRANSMISSION];
208 static uint8_t ltc_TXPECbuffer[LTC_N_BYTES_FOR_DATA_TRANSMISSION];
209 static uint8_t ltc_TXBuffer[LTC_N_BYTES_FOR_DATA_TRANSMISSION_DATA_ONLY];
210                                                  (0+(6*LTC_N_LTC))
211 static uint8_t ltc_TXBufferClock[4+9];
212 static uint8_t ltc_TXPECBufferClock[4+9];
213                       ???
214
215 /*================== Function Prototypes ==================================*/
216 /* Init functions */
217 static STD_RETURN_TYPE_e LTC_Init(void);          Line 2296
218 static void LTC_Initialize_Database(void);        Line 282
219 static STD_RETURN_TYPE_e LTC_CheckConfiguration(void); Line 316. Should be named as LTC_CheckCellConfiguration
220
221 static void LTC_SaveBalancingFeedback(uint8_t *DataBufferSPI_RX);
222 static void LTC_Get_BalancingControlValues(void);
223 static void LTC_StateTransition(LTC_STATEMACH_e state, uint8_t substate, uint16_t timer_ms);   Line 342
224 static void LTC_CondBasedStateTransition(STD_RETURN_TYPE_e retVal, DIAG_CH_ID_e diagCode, uint8_t state_ok, uint8_t
    substate_ok, uint16_t timer_ms_ok, uint8_t state_nok, uint8_t substate_nok, uint16_t timer_ms_nok);     Line 365
225
```

```
226  static STD_RETURN_TYPE_e LTC_BalanceControl(uint8_t registerSet);        See Line 2361
227  static void LTC_ResetErrorTable(void);
228
229  static STD_RETURN_TYPE_e LTC_StartVoltageMeasurement(LTC_ADCMODE_e adcMode, LTC_ADCMEAS_CHAN_e  adcMeasCh);  See Line 2533
230  static STD_RETURN_TYPE_e LTC_StartGPIOMeasurement(LTC_ADCMODE_e adcMode, LTC_ADCMEAS_CHAN_e  adcMeasCh);     See Line 2576
231  static STD_RETURN_TYPE_e LTC_StartOpenWireMeasurement(LTC_ADCMODE_e adcMode, uint8_t PUP);
232
233  static uint16_t LTC_Get_MeasurementTCycle(LTC_ADCMODE_e adcMode, LTC_ADCMEAS_CHAN_e  adcMeasCh);
234  static void LTC_SaveRXtoVoltagebuffer(uint8_t registerSet, uint8_t *rxBuffer);
235  static void LTC_SaveRXtoGPIOBuffer(uint8_t registerSet, uint8_t *rxBuffer);    See Line 2137
236
237  static STD_RETURN_TYPE_e LTC_RX_PECCheck(uint8_t *DataBufferSPI_RX_with_PEC);
238  static STD_RETURN_TYPE_e LTC_RX(uint8_t *Command, uint8_t *DataBufferSPI_RX_with_PEC);
239  static STD_RETURN_TYPE_e LTC_TX(uint8_t *Command, uint8_t *DataBufferSPI_TX, uint8_t *DataBufferSPI_TX_with_PEC);   See Line 2789
240  static void LTC_SetMUXChCommand(uint8_t *DataBufferSPI_TX, uint8_t mux, uint8_t channel);
241  static uint8_t LTC_SendEEPROMReadCommand(uint8_t *DataBufferSPI_TX, uint8_t *DataBufferSPI_TX_with_PEC, uint8_t step);
242  static void LTC_SetEEPROMReadCommand(uint8_t step, uint8_t *DataBufferSPI_TX);
243  static void LTC_EEPROMSaveReadValue(uint8_t *rxBuffer);
244  static uint8_t LTC_SendEEPROMWriteCommand(uint8_t *DataBufferSPI_TX, uint8_t *DataBufferSPI_TX_with_PEC, uint8_t step);
245  static void LTC_SetEEPROMWriteCommand(uint8_t step, uint8_t *DataBufferSPI_TX);
246  static uint8_t LTC_SetMuxChannel(uint8_t *DataBufferSPI_TX, uint8_t *DataBufferSPI_TX_with_PEC, uint8_t mux, uint8_t
     channel);
247  static uint8_t LTC_SetPortExpander(uint8_t *DataBufferSPI_TX, uint8_t *DataBufferSPI_TX_with_PEC);
248  static void LTC_PortExpanderSaveValues(uint8_t *rxBuffer);
249  static void LTC_TempSensSaveTemp(uint8_t *rxBuffer);
250  static uint8_t LTC_SetPortExpanderDirection_TI(LTC_PORT_EXPANDER_TI_DIRECTION_e direction, uint8_t *DataBufferSPI_TX,
     uint8_t *DataBufferSPI_TX_with_PEC);
251  static uint8_t LTC_SetPortExpander_Output_TI(uint8_t *DataBufferSPI_TX, uint8_t *DataBufferSPI_TX_with_PEC);
252  static uint8_t LTC_GetPortExpander_Input_TI(uint8_t step, uint8_t *DataBufferSPI_TX, uint8_t
     *DataBufferSPI_TX_with_PEC);
253  static void LTC_PortExpanderSaveValues_TI(uint8_t *rxBuffer);
254
255  static STD_RETURN_TYPE_e LTC_I2CClock(uint8_t *DataBufferSPI_TX, uint8_t *DataBufferSPI_TX_with_PEC);
256  static STD_RETURN_TYPE_e LTC_Send_I2C_Command(uint8_t *DataBufferSPI_TX, uint8_t *DataBufferSPI_TX_with_PEC, uint8_t
     *cmd_data);
257
258  static uint8_t LTC_I2CCheckACK(uint8_t *DataBufferSPI_RX, int mux);
259
260  static void LTC_SaveMuxMeasurement(uint8_t *DataBufferSPI_RX, LTC_MUX_CH_CFG_s  *muxseqptr);    temp voltage --> temperature
261
262
263  static uint32_t LTC_GetSPIClock(void);
264  static void LTC_SetTransferTimes(void);
265
266  static LTC_RETURN_TYPE_e LTC_CheckStateRequest(LTC_STATE_REQUEST_e statereq);
267
268  static STD_RETURN_TYPE_e LTC_TimerElapsedAndSPITransmitOngoing(uint16_t timer);
269
270  /*================== Function Implementations ============================*/
271
272  /*================== Public functions ====================================*/
273   Some other functions are defined in ltc_cfg.h. (They should have been defined in the ltc.h file.)
```

```c
/*================== Static functions ====================================*/
/**
 * @brief   in the database, initializes the fields related to the LTC drivers.
 *
 * This function loops through all the LTC-related data fields in the database
 * and sets them to 0. It should be called in the initialization or re-initialization
 * routine of the LTC driver.
 */
static void LTC_Initialize_Database(void) {
    uint16_t i = 0;
                Total number of cells in the battery pack
    for (i=0; i < BS_NR_OF_BAT_CELLS; i++) {
        ltc_openwire_pup_buffer[i] = 0;
        ltc_openwire_pdown_buffer[i] = 0;
        ltc_openwire_delta[i] = 0;
    }

    for (i=0; i < (LTC_N_MUX_CHANNELS_PER_MUX*LTC_N_USER_MUX_PER_LTC*BS_NR_OF_MODULES); i++) {
        ltc_user_mux.value[i] = 0;
    }
    ltc_user_mux.previous_timestamp = 0;
    ltc_user_mux.timestamp = 0;
    ltc_user_mux.state = 0;

    The above are local ones.

    /* Read database entries to initialize all local copies with 0 */
    DB_ReadBlock(&ltc_cellvoltage, DATA_BLOCK_ID_CELLVOLTAGE);
    DB_ReadBlock(&ltc_celltemperature, DATA_BLOCK_ID_CELLTEMPERATURE);
    DB_ReadBlock(&ltc_minmax, DATA_BLOCK_ID_MINMAX);
    DB_ReadBlock(&ltc_balancing_feedback, DATA_BLOCK_ID_BALANCING_FEEDBACK_VALUES);
    DB_ReadBlock(&ltc_balancing_control, DATA_BLOCK_ID_BALANCING_CONTROL_VALUES);
    DB_ReadBlock(&ltc_slave_control, DATA_BLOCK_ID_SLAVE_CONTROL);
    DB_ReadBlock(&ltc_openwire, DATA_BLOCK_ID_OPEN_WIRE);
}   Need to see why not doing a ltc_allGPIO? Not used?


/**
 * @brief   function to check configuration of config struct
 *          ltc_voltage_input_used
 *
 * @return  E_OK if valid configuration detected, otherwise E_NOT_OK
 */
STD_RETURN_TYPE_e LTC_CheckConfiguration(void) {        Should be called as LTC_CheckCellConfiguration4EachModule
    STD_RETURN_TYPE_e retval = E_OK;
    uint8_t configuredCells = 0;

    /* Iterate over struct to check configuration of LTC input struct */
    for (uint8_t i = 0; i < BS_MAX_SUPPORTED_CELLS; i++) {
        if (ltc_voltage_input_used[i] == 1) {          Difined in ltc_cfg.c to be 1 for a used cell.
            configuredCells++;
        }
    }
```

```
326
327         if (configuredCells != BS_NR_OF_BAT_CELLS_PER_MODULE) {
328             retval = E_NOT_OK;
329         }
330
331         return retval;
332     }
333
334
335     /**
336      * @brief   function for setting LTC_Trigger state transitions
337      *
338      * @param  state:    state to transition into
339      * @param  substate: substate to transition into
340      * @param  timer_ms: transition into state, substate after timer elapsed
341      */
342     static void LTC_StateTransition(LTC_STATEMACH_e state, uint8_t substate, uint16_t timer_ms) {
343             ltc_state.state = state;
344             ltc_state.substate = substate;
345             ltc_state.timer = timer_ms;
346     }
347
348     /**
349      * @brief   condition-based state transition depending on retVal
350      *
351      * If retVal is E_OK, after timer_ms_ok is elapsed the LTC statemachine will
352      * transition into state_ok and substate_ok, otherwise after timer_ms_nok the
353      * statemachine will transition to state_nok and substate_nok. Depending on
354      * value of retVal the corresponding diagnosis entry will be called.
355      *
356      * @param  retVal:       condition to determine if statemachine will transition into ok or nok states
357      * @param  diagCode:     symbolic IDs for diagnosis entry, called with DIAG_EVENT_OK if retVal is E_OK,
     DIAG_EVENT_NOK otherwise
358      * @param  state_ok      state to transition into if retVal is E_OK
359      * @param  substate_ok:  substate to transition into if retVal is E_OK
360      * @param  timer_ms_ok:  transition into state_ok, substate_ok after timer_ms_ok elapsed
361      * @param  state_nok:    state to transition into if retVal is E_NOT_OK
362      * @param  substate_nok: substate to transition into if retVal is E_NOT_OK
363      * @param  timer_ms_nok: transition into state_nok, substate_nok after timer_ms_nok elapsed
364      */
365     static void LTC_CondBasedStateTransition(STD_RETURN_TYPE_e retVal, DIAG_CH_ID_e diagCode, uint8_t state_ok, uint8_t
    substate_ok, uint16_t timer_ms_ok, uint8_t state_nok, uint8_t substate_nok, uint16_t timer_ms_nok) {
366         if ((retVal != E_OK)) {
367             DIAG_Handler(diagCode, DIAG_EVENT_NOK, 0);
368             LTC_StateTransition(state_nok, substate_nok, timer_ms_nok);
369         } else {
370             DIAG_Handler(diagCode, DIAG_EVENT_OK, 0);
371             LTC_StateTransition(state_ok, substate_ok, timer_ms_ok);
372         }
373     }
374
375     /**
```

```
376   * @brief   stores the measured voltages in the database.
377   *
378   * This function loops through the data of all modules in the LTC daisy-chain that are
379   * stored in the LTC_CellVoltages buffer and writes them in the database.
380   * At each write iteration, the variable named "state" and related to voltages in the
381   * database is incremented.
382   *
383   */
384  extern void LTC_SaveVoltages(void) {        LTC_SaveCellVoltages
385      uint16_t i = 0;
386      uint16_t j = 0;
387      uint16_t min = UINT16_MAX;
388      uint16_t max = 0;
389      uint32_t mean = 0;
390      uint32_t sum = 0;
391      uint8_t module_number_min = 0;
392      uint8_t module_number_max = 0;
393      uint8_t cell_number_min = 0;
394      uint8_t cell_number_max = 0;
395      uint16_t nrValidCellVoltages = 0;
396      STD_RETURN_TYPE_e retval_PLminmax = E_NOT_OK;          PL stands for plausibility
397      STD_RETURN_TYPE_e retval_PLspread = E_NOT_OK;
398      STD_RETURN_TYPE_e result = E_NOT_OK;
399
400      /* Perform min/max voltage plausibility check */
401      retval_PLminmax = PL_CheckVoltageMinMax(&ltc_cellvoltage);
402                                                  valid_volt == 0 means the voltage is valid. See the
403      /* Use only valid cell voltages for calculating mean voltage */   notes after line 104. See alos lines 2097 and 2100.
404      for (i=0; i < BS_NR_OF_MODULES; i++) {
405          for (j=0; j < BS_NR_OF_BAT_CELLS_PER_MODULE; j++) {
406              if ((ltc_cellvoltage.valid_volt[i] & (0x01 << j)) == 0) {
407                  /* Cell voltage is valid -> use this voltage for subsequent calculations */
408                  nrValidCellVoltages++;
409                  sum += ltc_cellvoltage.voltage[i*(BS_NR_OF_BAT_CELLS_PER_MODULE)+j];
410                  if (ltc_cellvoltage.voltage[i*(BS_NR_OF_BAT_CELLS_PER_MODULE)+j] < min) {
411                      min = ltc_cellvoltage.voltage[i*(BS_NR_OF_BAT_CELLS_PER_MODULE)+j];
412                      module_number_min = i;
413                      cell_number_min = j;
414                  }
415                  if (ltc_cellvoltage.voltage[i*(BS_NR_OF_BAT_CELLS_PER_MODULE)+j] > max) {
416                      max = ltc_cellvoltage.voltage[i*(BS_NR_OF_BAT_CELLS_PER_MODULE)+j];
417                      module_number_max = i;
418                      cell_number_max = j;
419                  }
420              }
421          }
422      }
423
424      ltc_cellvoltage.packVoltage_mV = sum;
425
426      /* Prevent division by 0, if all cell voltages are invalid */
427      if (nrValidCellVoltages > 0) {
```

```
428             mean = sum/nrValidCellVoltages;
429         }
430 
431         /* Perform voltage spread plausibility check */
432         retval_PLspread = PL_CheckVoltageSpread(&ltc_cellvoltage, mean);
433 
434         /* Set flag if plausibility error detected */
435         if ((retval_PLminmax == E_OK) && (retval_PLspread == E_OK)) {
436             result = E_OK;
437         }
438         DIAG_checkEvent(result, DIAG_CH_PLAUSIBILITY_CELL_VOLTAGE, 0);
439 
440         ltc_cellvoltage.state++;
441         ltc_minmax.state++;
442         ltc_minmax.voltage_mean = mean;
443         ltc_minmax.previous_voltage_min = ltc_minmax.voltage_min;
444         ltc_minmax.voltage_min = min;
445         ltc_minmax.voltage_module_number_min = module_number_min;
446         ltc_minmax.voltage_cell_number_min = cell_number_min;
447         ltc_minmax.previous_voltage_max = ltc_minmax.voltage_max;
448         ltc_minmax.voltage_max = max;
449         ltc_minmax.voltage_module_number_max = module_number_max;
450         ltc_minmax.voltage_cell_number_max = cell_number_max;
451 
452         DB_WriteBlock(&ltc_cellvoltage, DATA_BLOCK_ID_CELLVOLTAGE);
453         DB_WriteBlock(&ltc_minmax, DATA_BLOCK_ID_MINMAX);
454     }
455 
456     /**
457      * @brief   stores the measured temperatures and the measured multiplexer feedbacks in the database.
458      *
459      * This function loops through the temperature and multiplexer feedback data of all modules
460      * in the LTC daisy-chain that are stored in the LTC_MultiplexerVoltages buffer and writes
461      * them in the database.
462      * At each write iteration, the variables named "state" and related to temperatures and multiplexer feedbacks
463      * in the database are incremented.
464      *
465      */
466     extern void LTC_SaveTemperatures(void) {
467         uint16_t i = 0;
468         uint16_t j = 0;
469         int16_t min = INT16_MAX;
470         int16_t max = INT16_MIN;
471         int32_t sum = 0;
472         float mean = 0;
473         uint8_t module_number_min = 0;
474         uint8_t module_number_max = 0;
475         uint8_t sensor_number_min = 0;
476         uint8_t sensor_number_max = 0;
477         STD_RETURN_TYPE_e retval_PL  = E_NOT_OK;
478         uint16_t nrValidTemperatures = 0;
479 
```

```
480         /* Perform plausibility check */
481         retval_PL = PL_CheckTempMinMax(&ltc_celltemperature);
482         /* Set flag if plausibility error detected */
483         DIAG_checkEvent(retval_PL, DIAG_CH_PLAUSIBILITY_CELL_TEMP, 0);
484
485         for (i=0; i < BS_NR_OF_MODULES; i++) {
486             for (j=0; j < BS_NR_OF_TEMP_SENSORS_PER_MODULE; j++) {
487                 if ((ltc_celltemperature.valid_temperature[i] & (0x01 << j)) == 0) {
488                     /* Cell voltage is valid -> use this voltage for subsequent calculations */
489                     nrValidTemperatures++;
490                     sum += ltc_celltemperature.temperature[i*(BS_NR_OF_TEMP_SENSORS_PER_MODULE)+j];
491                     if (ltc_celltemperature.temperature[i*(BS_NR_OF_TEMP_SENSORS_PER_MODULE)+j] < min) {
492                         min = ltc_celltemperature.temperature[i*(BS_NR_OF_TEMP_SENSORS_PER_MODULE)+j];
493                         module_number_min = i;
494                         sensor_number_min = j;
495                     }
496                     if (ltc_celltemperature.temperature[i*(BS_NR_OF_TEMP_SENSORS_PER_MODULE)+j] > max) {
497                         max = ltc_celltemperature.temperature[i*(BS_NR_OF_TEMP_SENSORS_PER_MODULE)+j];
498                         module_number_max = i;
499                         sensor_number_max = j;
500                     }
501                 }
502             }
503         }
504
505         /* Prevent division by 0, if all temperatues are invalid */
506         if (nrValidTemperatures > 0) {
507             mean = sum/nrValidTemperatures;
508         }
509
510         ltc_celltemperature.state++;
511         ltc_minmax.state++;
512         ltc_minmax.temperature_mean = mean;
513         ltc_minmax.temperature_min = min;
514         ltc_minmax.temperature_module_number_min = module_number_min;
515         ltc_minmax.temperature_sensor_number_min = sensor_number_min;
516         ltc_minmax.temperature_max = max;
517         ltc_minmax.temperature_module_number_max = module_number_max;
518         ltc_minmax.temperature_sensor_number_max = sensor_number_max;
519         DB_WriteBlock(&ltc_celltemperature, DATA_BLOCK_ID_CELLTEMPERATURE);
520         DB_WriteBlock(&ltc_minmax, DATA_BLOCK_ID_MINMAX);
521     }
522
523     /**
524      * @brief   stores the measured GPIOs in the database.
525      *
526      * This function loops through the data of all modules in the LTC daisy-chain that are
527      * stored in the ltc_allgpiovoltage buffer and writes them in the database.
528      * At each write iteration, the variable named "state" and related to voltages in the
529      * database is incremented.
530      *
531      */
```

```
532  extern void LTC_SaveAllGPIOMeasurement(void) {
533      ltc_allgpiovoltage.state++;
534      DB_WriteBlock(&ltc_allgpiovoltage, DATA_BLOCK_ID_ALLGPIOVOLTAGE);
535  }
536
537  /**
538   * @brief   stores the measured balancing feedback values in the database.
539   *
540   * This function stores the global balancing feedback value measured on GPIO3 of the LTC into the database
541   *
542   */
543  static void LTC_SaveBalancingFeedback(uint8_t *DataBufferSPI_RX) {
544      uint16_t i = 0;
545      uint16_t val_i = 0;
546
547      for (i=0; i < LTC_N_LTC; i++) {
548          val_i = DataBufferSPI_RX[8+1*i*8] | (DataBufferSPI_RX[8+1*i*8+1] << 8);     /* raw value, GPIO3 */
549                                  lower byte                                  higher byte
550              ltc_balancing_feedback.value[i] = val_i;
551      }
552
553      ltc_balancing_feedback.state++;
554      DB_WriteBlock(&ltc_balancing_feedback, DATA_BLOCK_ID_BALANCING_FEEDBACK_VALUES);
555  }
556
557
558  /**
559   * @brief   gets the balancing orders from the database.
560   *
561   * This function gets the balancing control from the database. Balancing control
562   * is set by the BMS. The LTC driver only executes the balancing orders.
563   */
564  static void LTC_Get_BalancingControlValues(void) {
565      DB_ReadBlock(&ltc_balancing_control, DATA_BLOCK_ID_BALANCING_CONTROL_VALUES);
566  }
567
568
569
570  /**
571   * @brief   re-entrance check of LTC state machine trigger function
572   *
573   * This function is not re-entrant and should only be called time- or event-triggered.
574   * It increments the triggerentry counter from the state variable ltc_state.
575   * It should never be called by two different processes, so if it is the case, triggerentry
576   * should never be higher than 0 when this function is called.
577   *
578   *
579   * @return  retval  0 if no further instance of the function is active, 0xff else
580   *
581   */
582  uint8_t LTC_CheckReEntrance(void) {
583      uint8_t retval = 0;
```

We can intercept the GPIO measurements here to see if they match up the voltages of the voltage diviters.

GPIO3 is used in our system for temperature reading.

The first 8 should be replaced by GPIO3_index = 4 + (3-1) * 2 here. 4 is the number of bytes of command, 3 GPIO3, and 2 is the number of bytes for each data.

Read from the database, which is written on line 139 of bal.c

```
584
585         OS_TaskEnter_Critical();
586         if (!ltc_state.triggerentry) {
587             ltc_state.triggerentry++;
588         } else {
589             retval = 0xFF;    /* multiple calls of function */
590         }
591         OS_TaskExit_Critical();
592
593         return (retval);
594     }
595
596     /**
597      * @brief   gets the current state request.
598      *
599      * This function is used in the functioning of the LTC state machine.
600      *
601      * @return  retval  current state request, taken from LTC_STATE_REQUEST_e
602      */
603     extern LTC_STATE_REQUEST_e LTC_GetStateRequest(void) {
604         LTC_STATE_REQUEST_e retval = LTC_STATE_NO_REQUEST;
605
606         OS_TaskEnter_Critical();
607         retval    = ltc_state.statereq;
608         OS_TaskExit_Critical();
609
610         return (retval);
611     }
612
613     /**
614      * @brief   gets the current state.
615      *
616      * This function is used in the functioning of the LTC state machine.
617      *
618      * @return  current state, taken from LTC_STATEMACH_e
619      */
620     extern LTC_STATEMACH_e LTC_GetState(void) {
621         return (ltc_state.state);
622     }
623
624     /**
625      * @brief   transfers the current state request to the state machine.
626      *
627      * This function takes the current state request from ltc_state and transfers it to the state machine.
628      * It resets the value from ltc_state to LTC_STATE_NO_REQUEST
629      *
630      * @param   *busIDptr       bus ID, main or backup (deprecated)
631      * @param   *adcModeptr     LTC ADCmeasurement mode (fast, normal or filtered)
632      * @param   *adcMeasChptr   number of channels measured for GPIOS (one at a time for multiplexers or all five GPIOs)
633      *
634      * @return  retVal          current state request, taken from LTC_STATE_REQUEST_e
635      *
```

```
636     */
637    LTC_STATE_REQUEST_e LTC_TransferStateRequest(uint8_t *busIDptr, LTC_ADCMODE_e *adcModeptr, LTC_ADCMEAS_CHAN_e
       *adcMeasChptr) {                                          Not used in the function?
638        LTC_STATE_REQUEST_e retval = LTC_STATE_NO_REQUEST;
639
640        OS_TaskEnter_Critical();
641        retval    = ltc_state.statereq;
642        *adcModeptr = ltc_state.adcModereq;
643        *adcMeasChptr = ltc_state.adcMeasChreq;
644        ltc_state.statereq = LTC_STATE_NO_REQUEST;
645        OS_TaskExit_Critical();
646
647        return (retval);
648    }
649
650    LTC_RETURN_TYPE_e LTC_SetStateRequest(LTC_STATE_REQUEST_e statereq) {
651        LTC_RETURN_TYPE_e retVal = LTC_STATE_NO_REQUEST;
652
653        OS_TaskEnter_Critical();
654        retVal = LTC_CheckStateRequest(statereq);
655
656        if (retVal == LTC_OK || retVal == LTC_BUSY_OK || retVal == LTC_OK_FROM_ERROR) {
657                ltc_state.statereq   = statereq;
658            }
659        OS_TaskExit_Critical();
660
661        return (retVal);
662    }                    Function defintions resume at Line 1967.
663
664    void LTC_Trigger(void) {      This function ends at line 1962.
665        STD_RETURN_TYPE_e retVal = E_OK;
666        LTC_STATE_REQUEST_e statereq = LTC_STATE_NO_REQUEST;
667        uint8_t tmpbusID = 0;   This variable is never used (assigned a value other than 0) in this function/file.
668        LTC_ADCMODE_e tmpadcMode = LTC_ADCMODE_UNDEFINED;                    temporary adc mode
669        LTC_ADCMEAS_CHAN_e tmpadcMeasCh = LTC_ADCMEAS_UNDEFINED;
670
671        /* Check re-entrance of function */
672        if (LTC_CheckReEntrance())
673            return;
674
675        DIAG_SysMonNotify(DIAG_SYSMON_LTC_ID, 0);        /* task is running, state = ok */
676
677        if (ltc_state.check_spi_flag == FALSE) {
678            if (ltc_state.timer) {
679                if (--ltc_state.timer) {
680                    ltc_state.triggerentry--;
681                    return;    /* handle state machine only if timer has elapsed */
682                }
683            }
684        } else {
685            if (SPI_IsTransmitOngoing() == TRUE) {
686                if (ltc_state.timer) {
```

```
687                 if (--ltc_state.timer) {
688                     ltc_state.triggerentry--;
689                     return;    /* handle state machine only if timer has elapsed */
690                 }
691             }
692         }
693     }
694
695     switch (ltc_state.state) {
696         /*************************UNINITIALIZED*********************************/
697         case LTC_STATEMACH_UNINITIALIZED:
698             /* waiting for Initialization Request */
699             statereq = LTC_TransferStateRequest(&tmpbusID, &tmpadcMode, &tmpadcMeasCh);
700             if (statereq == LTC_STATE_INIT_REQUEST) {
701                 LTC_SAVELASTSTATES();
702                 LTC_StateTransition(LTC_STATEMACH_INITIALIZATION, LTC_ENTRY_UNINITIALIZED, LTC_STATEMACH_SHORTTIME);
703                 ltc_state.adcMode = tmpadcMode;
704                 ltc_state.adcMeasCh = tmpadcMeasCh;
705             } else if (statereq == LTC_STATE_NO_REQUEST) {
706                 /* no actual request pending */
707             } else {
708                 ltc_state.ErrRequestCounter++;   /* illegal request pending */
709             }
710             break;
711
712         /*************************INITIALIZATION*********************************/
713         case LTC_STATEMACH_INITIALIZATION:
714
715             LTC_SetTransferTimes();    Should be named as LTC_SetSPITransferTimes()
716             ltc_state.muxmeas_seqptr = ltc_mux_seq.seqptr;
717             ltc_state.muxmeas_nr_end = ltc_mux_seq.nr_of_steps;
718             ltc_state.muxmeas_seqendptr = ((LTC_MUX_CH_CFG_s *)ltc_mux_seq.seqptr)+ltc_mux_seq.nr_of_steps;  /* last
                sequence + 1 */
719
720             if (ltc_state.substate == LTC_ENTRY_INITIALIZATION) {
721                 LTC_SAVELASTSTATES();
722                 retVal = LTC_SendWakeUp();        /* Send dummy byte to wake up the daisy chain */
723                 LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
724                                              LTC_STATEMACH_INITIALIZATION, LTC_RE_ENTRY_INITIALIZATION,
                                                 LTC_STATEMACH_DAISY_CHAIN_FIRST_INITIALIZATION_TIME,
725                                              LTC_STATEMACH_INITIALIZATION, LTC_ENTRY_INITIALIZATION,
                                                 LTC_STATEMACH_SHORTTIME);
726
727             } else if (ltc_state.substate == LTC_RE_ENTRY_INITIALIZATION) {
728                 LTC_SAVELASTSTATES();
729                 retVal = LTC_SendWakeUp();  /* Send dummy byte again to wake up the daisy chain */
730                 LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
731                                              LTC_STATEMACH_INITIALIZATION, LTC_START_INIT_INITIALIZATION,
                                                 LTC_STATEMACH_DAISY_CHAIN_SECOND_INITIALIZATION_TIME,
732                                              LTC_STATEMACH_INITIALIZATION, LTC_RE_ENTRY_INITIALIZATION,
                                                 LTC_STATEMACH_SHORTTIME);
733
```

```
734             } else if (ltc_state.substate == LTC_START_INIT_INITIALIZATION) {
735                 retVal = LTC_Init();  /* Initialize main LTC loop */
736                 ltc_state.lastsubstate = ltc_state.substate;
737                 DIAG_checkEvent(retVal, DIAG_CH_LTC_SPI, 0);
738                 LTC_StateTransition(LTC_STATEMACH_INITIALIZATION, LTC_EXIT_INITIALIZATION,
                    ltc_state.commandDataTransferTime);
739
740             } else if (ltc_state.substate == LTC_EXIT_INITIALIZATION) {
741             /* in daisy-chain mode, there is no confirmation of the initialization */
742                 LTC_SAVELASTSTATES();
743                 LTC_Initialize_Database();
744                 LTC_ResetErrorTable();
745                 LTC_StateTransition(LTC_STATEMACH_INITIALIZED, LTC_ENTRY_INITIALIZATION, LTC_STATEMACH_SHORTTIME);
746             }
747             break;
748
749         /*****************************INITIALIZED*************************************/
750         case LTC_STATEMACH_INITIALIZED:
751             retVal = LTC_CheckConfiguration();      Should be called LTC_CheckCellConfiguration()
752             LTC_SAVELASTSTATES();
753             /* Stay in INITIALIZED state if configuration error detected */
754             LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_CONFIG,
755                     LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME,
756                     LTC_STATEMACH_INITIALIZED, LTC_ENTRY_INITIALIZATION, LTC_STATEMACH_SHORTTIME);
757             break;
758                                         (Cell Voltage)
759         /*****************************START MEASUREMENT*******************************/
760         case LTC_STATEMACH_STARTMEAS:
761                                                         = LTC_ADCMODE_NORMAL_DCP0
762             ltc_state.adcMode = LTC_VOLTAGE_MEASUREMENT_MODE;
763             ltc_state.adcMeasCh = LTC_ADCMEAS_ALLCHANNEL;
764
765             ltc_state.check_spi_flag = FALSE;
766             retVal = LTC_StartVoltageMeasurement(ltc_state.adcMode, ltc_state.adcMeasCh);
767             LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
768                                          LTC_STATEMACH_READVOLTAGE, LTC_READ_VOLTAGE_REGISTER_A_RDCVA_READVOLTAGE,
                                             (ltc_state.commandTransferTime + LTC_Get_MeasurementTCycle(ltc_state.adcMode,
                                             ltc_state.adcMeasCh)),
769                                          LTC_STATEMACH_READVOLTAGE, LTC_READ_VOLTAGE_REGISTER_A_RDCVA_READVOLTAGE,
                                             LTC_STATEMACH_SHORTTIME);
770             break;
771                                            Cell
772         /*****************************READ VOLTAGE************************************/
773         case LTC_STATEMACH_READVOLTAGE:
774
775             if (ltc_state.substate == LTC_READ_VOLTAGE_REGISTER_A_RDCVA_READVOLTAGE) {
776                 ltc_state.check_spi_flag = TRUE;
777                 SPI_SetTransmitOngoing();
778                 retVal = LTC_RX((uint8_t*)(ltc_cmdRDCVA), ltc_RXPECbuffer);
779                 LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
780                         LTC_STATEMACH_READVOLTAGE, LTC_READ_VOLTAGE_REGISTER_B_RDCVB_READVOLTAGE,
                            (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
```

```
781                     LTC_STATEMACH_READVOLTAGE, LTC_READ_VOLTAGE_REGISTER_B_RDCVB_READVOLTAGE,
                        LTC_STATEMACH_SHORTTIME);
782             break;
783
784         } else if (ltc_state.substate == LTC_READ_VOLTAGE_REGISTER_B_RDCVB_READVOLTAGE) {
785             retVal = LTC_RX_PECCheck(ltc_RXPECbuffer);      Check PEC of Group A
786             DIAG_checkEvent(retVal, DIAG_CH_LTC_PEC, 0);
787             LTC_SaveRXtoVoltagebuffer(0, ltc_RXPECbuffer); Save results of Group A
788
789             SPI_SetTransmitOngoing();
790             retVal = LTC_RX((uint8_t*)(ltc_cmdRDCVB), ltc_RXPECbuffer);
791             LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
792                     LTC_STATEMACH_READVOLTAGE, LTC_READ_VOLTAGE_REGISTER_C_RDCVC_READVOLTAGE,
                        (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
793                     LTC_STATEMACH_READVOLTAGE, LTC_READ_VOLTAGE_REGISTER_C_RDCVC_READVOLTAGE,
                        LTC_STATEMACH_SHORTTIME);
794             break;
795
796         } else if (ltc_state.substate == LTC_READ_VOLTAGE_REGISTER_C_RDCVC_READVOLTAGE) {
797             retVal = LTC_RX_PECCheck(ltc_RXPECbuffer);
798             DIAG_checkEvent(retVal, DIAG_CH_LTC_PEC, 0);
799             LTC_SaveRXtoVoltagebuffer(1, ltc_RXPECbuffer);
800
801             SPI_SetTransmitOngoing();
802             retVal = LTC_RX((uint8_t*)(ltc_cmdRDCVC), ltc_RXPECbuffer);
803             LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
804                                          LTC_STATEMACH_READVOLTAGE, LTC_READ_VOLTAGE_REGISTER_D_RDCVD_READVOLTAGE,
                                             (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
805                                          LTC_STATEMACH_READVOLTAGE, LTC_READ_VOLTAGE_REGISTER_D_RDCVD_READVOLTAGE,
                                             LTC_STATEMACH_SHORTTIME);
806             break;
807
808         } else if (ltc_state.substate == LTC_READ_VOLTAGE_REGISTER_D_RDCVD_READVOLTAGE) {
809             retVal = LTC_RX_PECCheck(ltc_RXPECbuffer);
810             DIAG_checkEvent(retVal, DIAG_CH_LTC_PEC, 0);
811             LTC_SaveRXtoVoltagebuffer(2, ltc_RXPECbuffer);
812
813             SPI_SetTransmitOngoing();
814             retVal = LTC_RX((uint8_t*)(ltc_cmdRDCVD), ltc_RXPECbuffer);
815             if (BS_MAX_SUPPORTED_CELLS > 12) {
816                 LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
817                         LTC_STATEMACH_READVOLTAGE, LTC_READ_VOLTAGE_REGISTER_E_RDCVE_READVOLTAGE,
                            (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
818                         LTC_STATEMACH_READVOLTAGE, LTC_READ_VOLTAGE_REGISTER_E_RDCVE_READVOLTAGE,
                            LTC_STATEMACH_SHORTTIME);
819             } else {
820                 LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
821                         LTC_STATEMACH_READVOLTAGE, LTC_EXIT_READVOLTAGE,
                            (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
822                         LTC_STATEMACH_READVOLTAGE, LTC_EXIT_READVOLTAGE, LTC_STATEMACH_SHORTTIME);
823             }
824             break;
```

```
825
826             } else if (ltc_state.substate == LTC_READ_VOLTAGE_REGISTER_E_RDCVE_READVOLTAGE) {
827                 retVal = LTC_RX_PECCheck(ltc_RXPECbuffer);
828                 DIAG_checkEvent(retVal, DIAG_CH_LTC_PEC, 0);
829                 LTC_SaveRXtoVoltagebuffer(3, ltc_RXPECbuffer);
830
831                 SPI_SetTransmitOngoing();
832                 retVal = LTC_RX((uint8_t*)(ltc_cmdRDCVE), ltc_RXPECbuffer);
833                  if (BS_MAX_SUPPORTED_CELLS > 15) {
834                     LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
835                             LTC_STATEMACH_READVOLTAGE, LTC_READ_VOLTAGE_REGISTER_F_RDCVF_READVOLTAGE,
                             (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
836                             LTC_STATEMACH_READVOLTAGE, LTC_READ_VOLTAGE_REGISTER_F_RDCVF_READVOLTAGE,
                             LTC_STATEMACH_SHORTTIME);
837                 } else {
838                     LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
839                             LTC_STATEMACH_READVOLTAGE, LTC_EXIT_READVOLTAGE,
                             (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
840                             LTC_STATEMACH_READVOLTAGE, LTC_EXIT_READVOLTAGE, LTC_STATEMACH_SHORTTIME);
841                 }
842                 break;
843
844             } else if (ltc_state.substate == LTC_READ_VOLTAGE_REGISTER_F_RDCVF_READVOLTAGE) {
845                 retVal = LTC_RX_PECCheck(ltc_RXPECbuffer);
846                 DIAG_checkEvent(retVal, DIAG_CH_LTC_PEC, 0);
847                 LTC_SaveRXtoVoltagebuffer(4, ltc_RXPECbuffer);
848
849                 SPI_SetTransmitOngoing();
850                 retVal = LTC_RX((uint8_t*)(ltc_cmdRDCVF), ltc_RXPECbuffer);
851                 LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
852                         LTC_STATEMACH_READVOLTAGE, LTC_EXIT_READVOLTAGE,
                         (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
853                         LTC_STATEMACH_READVOLTAGE, LTC_EXIT_READVOLTAGE, LTC_STATEMACH_SHORTTIME);
854                 break;
855
856             } else if (ltc_state.substate == LTC_EXIT_READVOLTAGE) {
857                 retVal = LTC_RX_PECCheck(ltc_RXPECbuffer);
858                 DIAG_checkEvent(retVal, DIAG_CH_LTC_PEC, 0);
859                 if (BS_MAX_SUPPORTED_CELLS == 12) {
860                     LTC_SaveRXtoVoltagebuffer(3, ltc_RXPECbuffer);
861                 } else if (BS_MAX_SUPPORTED_CELLS == 15) {
862                     LTC_SaveRXtoVoltagebuffer(4, ltc_RXPECbuffer);
863                 } else if (BS_MAX_SUPPORTED_CELLS == 18) {
864                     LTC_SaveRXtoVoltagebuffer(5, ltc_RXPECbuffer);
865                 }
866
867                 /* Switch to different state if read voltage state is reused
868                  * e.g. open-wire check...                                    */
869                 if (ltc_state.reusageMeasurementMode == LTC_NOT_REUSED) {
870                     LTC_SaveVoltages();
871                     LTC_StateTransition(LTC_STATEMACH_MUXMEASUREMENT, LTC_STATEMACH_MUXCONFIGURATION_INIT,
                     LTC_STATEMACH_SHORTTIME);
```

```
872             } else if (ltc_state.reusageMeasurementMode == LTC_REUSE_READVOLT_FOR_ADOW_PUP) {
873                 LTC_StateTransition(LTC_STATEMACH_OPENWIRE_CHECK, LTC_READ_VOLTAGES_PULLUP_OPENWIRE_CHECK,
                    LTC_STATEMACH_SHORTTIME);
874             } else if (ltc_state.reusageMeasurementMode == LTC_REUSE_READVOLT_FOR_ADOW_PDOWN) {
875                 LTC_StateTransition(LTC_STATEMACH_OPENWIRE_CHECK, LTC_READ_VOLTAGES_PULLDOWN_OPENWIRE_CHECK,
                    LTC_STATEMACH_SHORTTIME);
876             }
877             ltc_state.check_spi_flag = FALSE;
878         }
879         break;
880
881
882     /*************************MULTIPLEXED MEASUREMENT CONFIGURATION***********/
883     case LTC_STATEMACH_MUXMEASUREMENT:
884
885         if (ltc_state.substate == LTC_STATEMACH_MUXCONFIGURATION_INIT) {
886             ltc_state.adcMode = LTC_GPIO_MEASUREMENT_MODE;
887             ltc_state.adcMeasCh = LTC_ADCMEAS_SINGLECHANNEL_GPIO1;
888
889             if (ltc_state.muxmeas_seqptr >= ltc_state.muxmeas_seqendptr) {
890                 /* last step of sequence reached (or no sequence configured) */
891
892                 ltc_state.muxmeas_seqptr = ltc_mux_seq.seqptr;
893                 ltc_state.muxmeas_nr_end = ltc_mux_seq.nr_of_steps;
894                 ltc_state.muxmeas_seqendptr = ((LTC_MUX_CH_CFG_s *)ltc_mux_seq.seqptr)+ltc_mux_seq.nr_of_steps;
                    /* last sequence + 1 */
895
896                 LTC_SaveTemperatures();
897
898                 if (LTC_IsFirstMeasurementCycleFinished() == FALSE) {
899                     LTC_SetFirstMeasurementCycleFinished();
900                 }
901             }
902
903             ltc_state.check_spi_flag = TRUE;
904             SPI_SetTransmitOngoing();
905             retVal = LTC_SetMuxChannel(ltc_TXBuffer, ltc_TXPECbuffer,
906                                        ltc_state.muxmeas_seqptr->muxID,   /* mux */
907                                        ltc_state.muxmeas_seqptr->muxCh   /* channel */);
908             if (retVal != E_OK) {
909                 DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
910                 ++ltc_state.muxmeas_seqptr;
911                 LTC_StateTransition(LTC_STATEMACH_MUXMEASUREMENT, LTC_STATEMACH_MUXCONFIGURATION_INIT,
                    LTC_STATEMACH_SHORTTIME);
912             } else {
913                 DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
914                 LTC_StateTransition(LTC_STATEMACH_MUXMEASUREMENT, LTC_SEND_CLOCK_STCOMM_MUXMEASUREMENT_CONFIG,
                    (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT));
915             }
916             break;
917
918         } else if (ltc_state.substate == LTC_SEND_CLOCK_STCOMM_MUXMEASUREMENT_CONFIG) {
```

Reuse is for the reuse of the reading code for the cell voltage data. Very nice approach!

This is only for GPIO1, which is connected to the temp sensors in foxBMS slave board

Defined in line 466. Saves the local values of the temperature to database.

There is a saving of minmax values. This should be in a separate function.

```
919                 if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
920                     DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
921                 } else {
922                     DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
923                 }
924
925                 SPI_SetTransmitOngoing();
926                 retVal = LTC_I2CClock(ltc_TXBufferClock, ltc_TXPECBufferClock);
927                 if (LTC_GOTO_MUX_CHECK == TRUE) {
928                     LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
929                                                  LTC_STATEMACH_MUXMEASUREMENT,
                                                    LTC_READ_I2C_TRANSMISSION_RESULT_RDCOMM_MUXMEASUREMENT_CONFIG,
                                                    (ltc_state.gpioClocksTransferTime+LTC_TRANSMISSION_TIMEOUT),
930                                                  LTC_STATEMACH_MUXMEASUREMENT,
                                                    LTC_READ_I2C_TRANSMISSION_RESULT_RDCOMM_MUXMEASUREMENT_CONFIG,
                                                    LTC_STATEMACH_SHORTTIME);;
931                 } else {
932                     LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
933                                                  LTC_STATEMACH_MUXMEASUREMENT, LTC_STATEMACH_MUXMEASUREMENT,
                                                    (ltc_state.gpioClocksTransferTime+LTC_TRANSMISSION_TIMEOUT),
934                                                  LTC_STATEMACH_MUXMEASUREMENT, LTC_STATEMACH_MUXMEASUREMENT,
                                                    LTC_STATEMACH_SHORTTIME);
935                 }
936                 break;
937
938             } else if (ltc_state.substate == LTC_READ_I2C_TRANSMISSION_RESULT_RDCOMM_MUXMEASUREMENT_CONFIG) {
939                 if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
940                     DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
941                 } else {
942                     DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
943                 }
944
945                 SPI_SetTransmitOngoing();
946                 retVal = LTC_RX((uint8_t*)ltc_cmdRDCOMM, ltc_RXPECbuffer);
947                 LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
948                                              LTC_STATEMACH_MUXMEASUREMENT,
                                                LTC_READ_I2C_TRANSMISSION_CHECK_MUXMEASUREMENT_CONFIG,
                                                ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT,
949                                              LTC_STATEMACH_MUXMEASUREMENT,
                                                LTC_READ_I2C_TRANSMISSION_CHECK_MUXMEASUREMENT_CONFIG,
                                                LTC_STATEMACH_SHORTTIME);
950                 break;
951
952             } else if (ltc_state.substate == LTC_READ_I2C_TRANSMISSION_CHECK_MUXMEASUREMENT_CONFIG) {
953                 if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
954                     DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
955                 } else {
956                     DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
957                 }
958
959                 retVal = LTC_RX_PECCheck(ltc_RXPECbuffer);
960                 DIAG_checkEvent(retVal, DIAG_CH_LTC_PEC, 0);
```

State

Substate

It's confusing to use the same name for a state and a substate.

```
961
962                 /* if CRC OK: check multiplexer answer on i2C bus */
963                 retVal = LTC_I2CCheckACK(ltc_RXPECbuffer, ltc_state.muxmeas_seqptr->muxID);
964                 DIAG_checkEvent(retVal, DIAG_CH_LTC_MUX, 0);
965                 LTC_StateTransition(LTC_STATEMACH_MUXMEASUREMENT, LTC_STATEMACH_MUXMEASUREMENT,
                    LTC_STATEMACH_SHORTTIME);
966                 break;
967
968             } else if (ltc_state.substate == LTC_STATEMACH_MUXMEASUREMENT) {
969                 if (ltc_state.muxmeas_seqptr->muxCh == 0xFF) {
970                     /* actual multiplexer is switched off, so do not make a measurement and follow up with next step
                        (mux configuration) */
971                     ++ltc_state.muxmeas_seqptr;         /*  go further with next step of sequence
972                                                             ltc_state.numberOfMeasuredMux not decremented, this does
                                                            not count as a measurement */
973                     LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
974                     break;
975                 } else {
976                     if (LTC_GOTO_MUX_CHECK == FALSE) {
977                         if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
978                             DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
979                         } else {
980                             DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
981                         }
982                     }
983
984                     ltc_state.check_spi_flag = FALSE;
985                     /* user multiplexer type -> connected to GPIO2! */
986                     if (ltc_state.muxmeas_seqptr->muxID == 1 || ltc_state.muxmeas_seqptr->muxID == 2) {
987                         retVal = LTC_StartGPIOMeasurement(ltc_state.adcMode, LTC_ADCMEAS_SINGLECHANNEL_GPIO2);
988                     } else {
989                         retVal = LTC_StartGPIOMeasurement(ltc_state.adcMode, LTC_ADCMEAS_SINGLECHANNEL_GPIO1);
990                     }
991                 }
992                 LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
993                                              LTC_STATEMACH_MUXMEASUREMENT, LTC_STATEMACH_READMUXMEASUREMENT,
                                                (ltc_state.commandTransferTime +
                                                LTC_Get_MeasurementTCycle(ltc_state.adcMode,
                                                LTC_ADCMEAS_SINGLECHANNEL_GPIO2)),  /*  wait, ADAX-Command */
994                                              LTC_STATEMACH_MUXMEASUREMENT, LTC_STATEMACH_READMUXMEASUREMENT,
                                                LTC_STATEMACH_SHORTTIME);
995                 break;
996
997             } else if (ltc_state.substate == LTC_STATEMACH_READMUXMEASUREMENT) {
998                 ltc_state.check_spi_flag = TRUE;
999
1000                SPI_SetTransmitOngoing();
1001                retVal = LTC_RX((uint8_t*)(ltc_cmdRDAUXA), ltc_RXPECbuffer);
1002                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1003                                             LTC_STATEMACH_MUXMEASUREMENT, LTC_STATEMACH_STOREMUXMEASUREMENT,
                                                ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT,
1004                                             LTC_STATEMACH_MUXMEASUREMENT, LTC_STATEMACH_STOREMUXMEASUREMENT,
```

```
                                                LTC_STATEMACH_SHORTTIME);
1005                    break;
1006
1007                } else if (ltc_state.substate == LTC_STATEMACH_STOREMUXMEASUREMENT) {
1008                    if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1009                        DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1010                    } else {
1011                        DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1012                    }
1013
1014                    retVal = LTC_RX_PECCheck(ltc_RXPECbuffer);
1015                    DIAG_checkEvent(retVal, DIAG_CH_LTC_PEC, 0);
1016                    LTC_SaveMuxMeasurement(ltc_RXPECbuffer, ltc_state.muxmeas_seqptr);   See line 2001. Temp voltage is converted to
1017                                                                                         temperature value here.
1018                    ++ltc_state.muxmeas_seqptr;
1019
1020                    if (ltc_state.balance_control_done == TRUE) {
1021                        statereq = LTC_TransferStateRequest(&tmpbusID, &tmpadcMode, &tmpadcMeasCh);
1022                        if (statereq == LTC_STATE_USER_IO_WRITE_REQUEST) {
1023                            LTC_StateTransition(LTC_STATEMACH_USER_IO_CONTROL, LTC_USER_IO_SET_OUTPUT_REGISTER,
                                LTC_STATEMACH_SHORTTIME);
1024                            ltc_state.balance_control_done = FALSE;
1025                        } else if (statereq == LTC_STATE_USER_IO_READ_REQUEST) {
1026                            LTC_StateTransition(LTC_STATEMACH_USER_IO_FEEDBACK, LTC_USER_IO_READ_INPUT_REGISTER,
                                LTC_STATEMACH_SHORTTIME);
1027                            ltc_state.balance_control_done = FALSE;
1028                        } else if (statereq == LTC_STATE_USER_IO_WRITE_REQUEST_TI) {
1029                            LTC_StateTransition(LTC_STATEMACH_USER_IO_CONTROL_TI, LTC_USER_IO_SET_DIRECTION_REGISTER_TI,
                                LTC_STATEMACH_SHORTTIME);
1030                            ltc_state.balance_control_done = FALSE;
1031                        } else if (statereq == LTC_STATE_USER_IO_READ_REQUEST_TI) {
1032                            LTC_StateTransition(LTC_STATEMACH_USER_IO_FEEDBACK_TI, LTC_USER_IO_SET_DIRECTION_REGISTER_TI,
                                LTC_STATEMACH_SHORTTIME);
1033                            ltc_state.balance_control_done = FALSE;
1034                        } else if (statereq == LTC_STATE_EEPROM_READ_REQUEST) {
1035                            LTC_StateTransition(LTC_STATEMACH_EEPROM_READ, LTC_EEPROM_READ_DATA1, LTC_STATEMACH_SHORTTIME);
1036                        } else if (statereq == LTC_STATE_EEPROM_WRITE_REQUEST) {
1037                            LTC_StateTransition(LTC_STATEMACH_EEPROM_WRITE, LTC_EEPROM_WRITE_DATA1,
                                LTC_STATEMACH_SHORTTIME);
1038                            ltc_state.balance_control_done = FALSE;
1039                        } else if (statereq == LTC_STATE_TEMP_SENS_READ_REQUEST) {
1040                            LTC_StateTransition(LTC_STATEMACH_TEMP_SENS_READ, LTC_TEMP_SENS_SEND_DATA1,
                                LTC_STATEMACH_SHORTTIME);
1041                            ltc_state.balance_control_done = FALSE;
1042                        } else if (statereq == LTC_STATEMACH_BALANCEFEEDBACK_REQUEST) {
1043                            LTC_StateTransition(LTC_STATEMACH_BALANCEFEEDBACK, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1044                            ltc_state.balance_control_done = FALSE;
1045                        }   else if (statereq == LTC_STATE_OPENWIRE_CHECK_REQUEST) {
1046                            LTC_StateTransition(LTC_STATEMACH_OPENWIRE_CHECK, LTC_REQUEST_PULLUP_CURRENT_OPENWIRE_CHECK,
                                LTC_STATEMACH_SHORTTIME);
1047                            /* Send ADOW command with PUP two times */
1048                            ltc_state.resendCommandCounter = LTC_NMBR_REQ_ADOW_COMMANDS;
```

```
1049                        ltc_state.balance_control_done = FALSE;
1050                    } else {
1051                        LTC_StateTransition(LTC_STATEMACH_BALANCECONTROL, LTC_CONFIG_BALANCECONTROL,
                            LTC_STATEMACH_SHORTTIME);
1052                        ltc_state.balance_control_done = TRUE;
1053                    }
1054                } else {
1055                    LTC_StateTransition(LTC_STATEMACH_BALANCECONTROL, LTC_CONFIG_BALANCECONTROL,
                        LTC_STATEMACH_SHORTTIME);
1056                    ltc_state.balance_control_done = TRUE;
1057                }
1058
1059                break;
1060            }
1061
1062            break;
1063
1064        /*************************BALANCE CONTROL*********************************/
1065        case LTC_STATEMACH_BALANCECONTROL:
1066
1067            if (ltc_state.substate == LTC_CONFIG_BALANCECONTROL) {
1068                ltc_state.check_spi_flag = TRUE;
1069                SPI_SetTransmitOngoing();
1070                retVal = LTC_BalanceControl(0);
1071                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1072                        LTC_STATEMACH_BALANCECONTROL, LTC_CONFIG2_BALANCECONTROL,
                            (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
1073                        LTC_STATEMACH_BALANCECONTROL, LTC_CONFIG2_BALANCECONTROL, LTC_STATEMACH_SHORTTIME);
1074                break;
1075
1076            } else if (ltc_state.substate == LTC_CONFIG2_BALANCECONTROL) {
1077                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1078                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1079                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1080                    break;
1081                } else {
1082                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1083                }
1084
1085                if (BS_NR_OF_BAT_CELLS_PER_MODULE > 12) {
1086                    SPI_SetTransmitOngoing();
1087                    retVal = LTC_BalanceControl(1);
1088                    LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1089                            LTC_STATEMACH_BALANCECONTROL, LTC_CONFIG2_BALANCECONTROL_END,
                                ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT,
1090                            LTC_STATEMACH_BALANCECONTROL, LTC_CONFIG2_BALANCECONTROL_END, LTC_STATEMACH_SHORTTIME);
1091                } else {
1092                    /* 12 cells, balancing control finished */
1093                    ltc_state.check_spi_flag = FALSE;
1094                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1095                }
1096
```

```
1097                break;
1098
1099            } else if (ltc_state.substate == LTC_CONFIG2_BALANCECONTROL_END) {
1100                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1101                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1102                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1103                    break;
1104                } else {
1105                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1106                }
1107                /* More than 12 cells, balancing control finished */
1108                ltc_state.check_spi_flag = FALSE;
1109                LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1110
1111                break;
1112            }
1113            break;
1114
1115            /*****************************START MEASUREMENT*****************************/
1116            case LTC_STATEMACH_ALLGPIOMEASUREMENT:
1117
1118                ltc_state.adcMode = LTC_GPIO_MEASUREMENT_MODE;
1119                ltc_state.adcMeasCh = LTC_ADCMEAS_ALLCHANNEL;
1120
1121                ltc_state.check_spi_flag = FALSE;
1122                retVal = LTC_StartGPIOMeasurement(ltc_state.adcMode, ltc_state.adcMeasCh);
1123                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1124                                             LTC_STATEMACH_READALLGPIO, LTC_READ_AUXILIARY_REGISTER_A_RDAUXA,
                                             (ltc_state.commandTransferTime +
                                             LTC_Get_MeasurementTCycle(ltc_state.adcMode, ltc_state.adcMeasCh)),
1125                                         LTC_STATEMACH_ALLGPIOMEASUREMENT, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
                                             /* TODO: @koffel here same state is kept if error occurs */
1126                break;
1127
1128        /*************************READ ALL GPIO VOLTAGE**********************************/
1129        case LTC_STATEMACH_READALLGPIO:
1130
1131            if (ltc_state.substate == LTC_READ_AUXILIARY_REGISTER_A_RDAUXA) {
1132                ltc_state.check_spi_flag = TRUE;
1133                SPI_SetTransmitOngoing();
1134                retVal = LTC_RX((uint8_t*)(ltc_cmdRDAUXA), ltc_RXPECbuffer);
1135                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1136                                             LTC_STATEMACH_READALLGPIO, LTC_READ_AUXILIARY_REGISTER_B_RDAUXB,
                                             (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
1137                                         LTC_STATEMACH_READALLGPIO, LTC_READ_AUXILIARY_REGISTER_B_RDAUXB,
                                             LTC_STATEMACH_SHORTTIME);
1138                break;
1139
1140            } else if (ltc_state.substate == LTC_READ_AUXILIARY_REGISTER_B_RDAUXB) {
1141                retVal = LTC_RX_PECCheck(ltc_RXPECbuffer);
1142                DIAG_checkEvent(retVal, DIAG_CH_LTC_PEC, 0);
1143                LTC_SaveRXtoGPIOBuffer(0, ltc_RXPECbuffer);
```

The state goes back to cell voltage measurement. we need to change it to go down to the All GPIO measurement.

All GPIO

Read the data from auxilary register Group A

```
1144
1145                SPI_SetTransmitOngoing();       Read the data from auxilary register Group B
1146                retVal = LTC_RX((uint8_t*)(ltc_cmdRDAUXB), ltc_RXPECbuffer);
1147
1148                if (BS_MAX_SUPPORTED_CELLS > 12) {    If true, the chip must be LTC 6813, which has 8 GPIOs. Hence, we need to read auxilary register Groups C/D.
1149                    LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1150                                                 LTC_STATEMACH_READALLGPIO, LTC_READ_AUXILIARY_REGISTER_C_RDAUXC,
                                                     (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
1151                                                 LTC_STATEMACH_READALLGPIO, LTC_READ_AUXILIARY_REGISTER_C_RDAUXC,
                                                     LTC_STATEMACH_SHORTTIME);
1152                } else {
1153                    LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1154                                                 LTC_STATEMACH_READALLGPIO, LTC_EXIT_READAUXILIARY_ALLGPIOS,
                                                     (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
1155                                                 LTC_STATEMACH_READALLGPIO, LTC_EXIT_READAUXILIARY_ALLGPIOS,
                                                     LTC_STATEMACH_SHORTTIME);
1156                }
1157                break;
1158
1159            } else if (ltc_state.substate == LTC_READ_AUXILIARY_REGISTER_C_RDAUXC) {
1160                retVal = LTC_RX_PECCheck(ltc_RXPECbuffer);
1161                DIAG_checkEvent(retVal, DIAG_CH_LTC_PEC, 0);
1162                LTC_SaveRXtoGPIOBuffer(1, ltc_RXPECbuffer);
1163
1164                SPI_SetTransmitOngoing();
1165                retVal = LTC_RX((uint8_t*)(ltc_cmdRDAUXC), ltc_RXPECbuffer);
1166                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1167                                             LTC_STATEMACH_READALLGPIO, LTC_READ_AUXILIARY_REGISTER_D_RDAUXD,
                                                 (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
1168                                             LTC_STATEMACH_READALLGPIO, LTC_READ_AUXILIARY_REGISTER_D_RDAUXD,
                                                 LTC_STATEMACH_SHORTTIME);
1169                break;
1170
1171            } else if (ltc_state.substate == LTC_READ_AUXILIARY_REGISTER_D_RDAUXD) {
1172                retVal = LTC_RX_PECCheck(ltc_RXPECbuffer);
1173                DIAG_checkEvent(retVal, DIAG_CH_LTC_PEC, 0);
1174                LTC_SaveRXtoGPIOBuffer(2, ltc_RXPECbuffer);
1175
1176                SPI_SetTransmitOngoing();
1177                retVal = LTC_RX((uint8_t*)(ltc_cmdRDAUXD), ltc_RXPECbuffer);
1178                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1179                                             LTC_STATEMACH_READALLGPIO, LTC_EXIT_READAUXILIARY_ALLGPIOS,
                                                 (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
1180                                             LTC_STATEMACH_READALLGPIO, LTC_EXIT_READAUXILIARY_ALLGPIOS,
                                                 LTC_STATEMACH_SHORTTIME);
1181                break;
1182
1183            } else if (ltc_state.substate == LTC_EXIT_READAUXILIARY_ALLGPIOS) {
1184                retVal = LTC_RX_PECCheck(ltc_RXPECbuffer);
1185                DIAG_checkEvent(retVal, DIAG_CH_LTC_PEC, 0);
1186
1187                if (BS_MAX_SUPPORTED_CELLS == 12) {
```

```
1188                    LTC_SaveRXtoGPIOBuffer(1, ltc_RXPECbuffer);
1189                } else if (BS_MAX_SUPPORTED_CELLS > 12) {
1190                    LTC_SaveRXtoGPIOBuffer(3, ltc_RXPECbuffer);
1191                }
1192                                                    We need to stop here and go back to the Cell voltage measurement.
1193                LTC_SaveAllGPIOMeasurement();       ltc_state.check_spi_flag = FALSE;
1194                                                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1195                if (ltc_state.balance_control_done == TRUE) {
1196                    statereq = LTC_TransferStateRequest(&tmpbusID, &tmpadcMode, &tmpadcMeasCh);
1197                    if (statereq == LTC_STATE_USER_IO_WRITE_REQUEST) {
1198                        LTC_StateTransition(LTC_STATEMACH_USER_IO_CONTROL, LTC_USER_IO_SET_OUTPUT_REGISTER,
                            LTC_STATEMACH_SHORTTIME);
1199                        ltc_state.balance_control_done = FALSE;
1200                    } else if (statereq == LTC_STATE_USER_IO_READ_REQUEST) {
1201                        LTC_StateTransition(LTC_STATEMACH_USER_IO_FEEDBACK, LTC_USER_IO_READ_INPUT_REGISTER,
                            LTC_STATEMACH_SHORTTIME);
1202                        ltc_state.balance_control_done = FALSE;
1203                    } else if (statereq == LTC_STATE_USER_IO_WRITE_REQUEST_TI) {
1204                        LTC_StateTransition(LTC_STATEMACH_USER_IO_CONTROL_TI, LTC_USER_IO_SET_DIRECTION_REGISTER_TI,
                            LTC_STATEMACH_SHORTTIME);
1205                        ltc_state.balance_control_done = FALSE;
1206                    } else if (statereq == LTC_STATE_USER_IO_READ_REQUEST_TI) {
1207                        LTC_StateTransition(LTC_STATEMACH_USER_IO_FEEDBACK_TI, LTC_USER_IO_SET_DIRECTION_REGISTER_TI,
                            LTC_STATEMACH_SHORTTIME);
1208                        ltc_state.balance_control_done = FALSE;
1209                    } else if (statereq == LTC_STATE_EEPROM_READ_REQUEST) {
1210                        LTC_StateTransition(LTC_STATEMACH_EEPROM_READ, LTC_EEPROM_READ_DATA1, LTC_STATEMACH_SHORTTIME);
1211                        ltc_state.balance_control_done = FALSE;
1212                    } else if (statereq == LTC_STATE_EEPROM_WRITE_REQUEST) {
1213                        LTC_StateTransition(LTC_STATEMACH_EEPROM_WRITE, LTC_EEPROM_WRITE_DATA1,
                            LTC_STATEMACH_SHORTTIME);
1214                        ltc_state.balance_control_done = FALSE;
1215                    } else if (statereq == LTC_STATE_TEMP_SENS_READ_REQUEST) {
1216                        LTC_StateTransition(LTC_STATEMACH_TEMP_SENS_READ, LTC_TEMP_SENS_SEND_DATA1,
                            LTC_STATEMACH_SHORTTIME);
1217                        ltc_state.balance_control_done = FALSE;
1218                    } else if (statereq == LTC_STATEMACH_BALANCEFEEDBACK_REQUEST) {
1219                        LTC_StateTransition(LTC_STATEMACH_BALANCEFEEDBACK, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1220                        ltc_state.balance_control_done = FALSE;
1221                    }  else if (statereq == LTC_STATE_OPENWIRE_CHECK_REQUEST) {
1222                        LTC_StateTransition(LTC_STATEMACH_OPENWIRE_CHECK, LTC_REQUEST_PULLUP_CURRENT_OPENWIRE_CHECK,
                            LTC_STATEMACH_SHORTTIME);
1223                        /* Send ADOW command with PUP two times */
1224                        ltc_state.resendCommandCounter = LTC_NMBR_REQ_ADOW_COMMANDS;
1225                        ltc_state.balance_control_done = FALSE;
1226                    } else {
1227                        LTC_StateTransition(LTC_STATEMACH_BALANCECONTROL, LTC_CONFIG_BALANCECONTROL,
                            LTC_STATEMACH_SHORTTIME);
1228                        ltc_state.balance_control_done = TRUE;
1229                    }
1230                } else {
1231                    LTC_StateTransition(LTC_STATEMACH_BALANCECONTROL, LTC_CONFIG_BALANCECONTROL,
```

```
                    LTC_STATEMACH_SHORTTIME);
1232                ltc_state.balance_control_done = TRUE;
1233            }
1234        }
1235
1236        break;
1237
1238
1239    /*************************BALANCE FEEDBACK*********************************/
1240    case LTC_STATEMACH_BALANCEFEEDBACK:
1241
1242         if (ltc_state.substate == LTC_ENTRY) {
1243            ltc_state.adcMode = LTC_ADCMODE_NORMAL_DCP0;
1244            ltc_state.adcMeasCh = LTC_ADCMEAS_SINGLECHANNEL_GPIO3;
1245
1246            ltc_state.check_spi_flag = FALSE;
1247            retVal = LTC_StartGPIOMeasurement(ltc_state.adcMode, ltc_state.adcMeasCh);
1248            LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1249                                         LTC_STATEMACH_BALANCEFEEDBACK, LTC_READ_FEEDBACK_BALANCECONTROL,
                                             (ltc_state.commandDataTransferTime +
                                             LTC_Get_MeasurementTCycle(ltc_state.adcMode, ltc_state.adcMeasCh)),
1250                                         LTC_STATEMACH_BALANCEFEEDBACK, LTC_READ_FEEDBACK_BALANCECONTROL,
                                             LTC_STATEMACH_SHORTTIME);
1251            break;
1252
1253        } else if (ltc_state.substate == LTC_READ_FEEDBACK_BALANCECONTROL) {
1254            ltc_state.check_spi_flag = TRUE;
1255            SPI_SetTransmitOngoing();
1256            retVal = LTC_RX((uint8_t*)ltc_cmdRDAUXA, ltc_RXPECbuffer);  /* read AUXA register */
1257            LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1258                    LTC_STATEMACH_BALANCEFEEDBACK, LTC_SAVE_FEEDBACK_BALANCECONTROL,
                        ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT,
1259                    LTC_STATEMACH_BALANCEFEEDBACK, LTC_SAVE_FEEDBACK_BALANCECONTROL, LTC_STATEMACH_SHORTTIME);
1260
1261        } else if (ltc_state.substate == LTC_SAVE_FEEDBACK_BALANCECONTROL) {
1262            if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1263                DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1264                LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1265                break;
1266            } else {
1267                DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1268            }
1269
1270            if (LTC_RX_PECCheck(ltc_RXPECbuffer) != E_OK) {
1271                DIAG_Handler(DIAG_CH_LTC_PEC, DIAG_EVENT_NOK, 0);
1272            } else {
1273                DIAG_Handler(DIAG_CH_LTC_PEC, DIAG_EVENT_OK, 0);
1274                LTC_SaveBalancingFeedback(ltc_RXPECbuffer);
1275            }
1276            LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1277            break;
1278        }
```

```
1279            break;
1280
1281        /*************************BOARD TEMPERATURE SENSOR*******************************/
1282        case LTC_STATEMACH_TEMP_SENS_READ:
1283
1284            if (ltc_state.substate == LTC_TEMP_SENS_SEND_DATA1) {
1285                ltc_state.check_spi_flag = TRUE;
1286                SPI_SetTransmitOngoing();
1287                retVal = LTC_Send_I2C_Command(ltc_TXBuffer, ltc_TXPECbuffer, (uint8_t*)ltc_I2CcmdTempSens0);
1288
1289                if (retVal != E_OK) {
1290                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1291                    ++ltc_state.muxmeas_seqptr;
1292                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1293                } else {
1294                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1295                    LTC_StateTransition(LTC_STATEMACH_TEMP_SENS_READ, LTC_TEMP_SENS_SEND_CLOCK_STCOMM1,
                        ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT);
1296                }
1297
1298                break;
1299
1300            } else if (ltc_state.substate == LTC_TEMP_SENS_SEND_CLOCK_STCOMM1) {
1301                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1302                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1303                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1304                    break;
1305                } else {
1306                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1307                }
1308
1309                SPI_SetTransmitOngoing();
1310                retVal = LTC_I2CClock(ltc_TXBufferClock, ltc_TXPECBufferClock);
1311                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1312                                             LTC_STATEMACH_TEMP_SENS_READ, LTC_TEMP_SENS_READ_DATA1,
                                             (ltc_state.gpioClocksTransferTime+LTC_TRANSMISSION_TIMEOUT),
1313                                             LTC_STATEMACH_TEMP_SENS_READ, LTC_TEMP_SENS_READ_DATA1,
                                             LTC_STATEMACH_SHORTTIME);
1314                break;
1315
1316            } else if (ltc_state.substate == LTC_TEMP_SENS_READ_DATA1) {
1317                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1318                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1319                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1320                    break;
1321                } else {
1322                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1323                }
1324
1325                SPI_SetTransmitOngoing();
1326                retVal = LTC_Send_I2C_Command(ltc_TXBuffer, ltc_TXPECbuffer, (uint8_t*)ltc_I2CcmdTempSens1);
1327
```

```
1328                if (retVal != E_OK) {
1329                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1330                    ++ltc_state.muxmeas_seqptr;
1331                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1332                } else {
1333                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1334                    LTC_StateTransition(LTC_STATEMACH_TEMP_SENS_READ, LTC_TEMP_SENS_SEND_CLOCK_STCOMM2,
                        (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT));
1335                }
1336
1337                break;
1338
1339            } else if (ltc_state.substate == LTC_TEMP_SENS_SEND_CLOCK_STCOMM2) {
1340                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1341                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1342                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1343                    break;
1344                } else {
1345                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1346                }
1347
1348                SPI_SetTransmitOngoing();
1349                retVal = LTC_I2CClock(ltc_TXBufferClock, ltc_TXPECBufferClock);
1350                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1351                                             LTC_STATEMACH_TEMP_SENS_READ,
                                             LTC_TEMP_SENS_READ_I2C_TRANSMISSION_RESULT_RDCOMM,
                                             (ltc_state.gpioClocksTransferTime+LTC_TRANSMISSION_TIMEOUT),
1352                                         LTC_STATEMACH_TEMP_SENS_READ,
                                             LTC_TEMP_SENS_READ_I2C_TRANSMISSION_RESULT_RDCOMM, LTC_STATEMACH_SHORTTIME);
1353                break;
1354            }  else if (ltc_state.substate == LTC_TEMP_SENS_READ_I2C_TRANSMISSION_RESULT_RDCOMM) {
1355                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1356                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1357                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1358                    break;
1359                } else {
1360                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1361                }
1362
1363                SPI_SetTransmitOngoing();
1364                retVal = LTC_RX((uint8_t*)ltc_cmdRDCOMM, ltc_RXPECbuffer);
1365                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1366                                             LTC_STATEMACH_TEMP_SENS_READ, LTC_TEMP_SENS_SAVE_TEMP,
                                             ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT,
1367                                         LTC_STATEMACH_TEMP_SENS_READ, LTC_TEMP_SENS_SAVE_TEMP,
                                             LTC_STATEMACH_SHORTTIME);
1368                break;
1369
1370            } else if (ltc_state.substate == LTC_TEMP_SENS_SAVE_TEMP) {
1371                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1372                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1373                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
```

```
1374                    break;
1375                } else {
1376                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1377                }
1378
1379                if (LTC_RX_PECCheck(ltc_RXPECbuffer) != E_OK) {
1380                    DIAG_Handler(DIAG_CH_LTC_PEC, DIAG_EVENT_NOK, 0);
1381                } else {
1382                    DIAG_Handler(DIAG_CH_LTC_PEC, DIAG_EVENT_OK, 0);
1383                    LTC_TempSensSaveTemp(ltc_RXPECbuffer);
1384                }
1385
1386                LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1387                break;
1388            }
1389            break;
1390
1391        /*************************WRITE TO PORT EXPANDER IO***********/
1392        case LTC_STATEMACH_USER_IO_CONTROL:
1393
1394            if (ltc_state.substate == LTC_USER_IO_SET_OUTPUT_REGISTER) {
1395                ltc_state.check_spi_flag = TRUE;
1396                SPI_SetTransmitOngoing();
1397                retVal = LTC_SetPortExpander(ltc_TXBuffer, ltc_TXPECbuffer);
1398
1399                if (retVal != E_OK) {
1400                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1401                    ++ltc_state.muxmeas_seqptr;
1402                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1403                } else {
1404                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1405                    LTC_StateTransition(LTC_STATEMACH_USER_IO_CONTROL, LTC_SEND_CLOCK_STCOMM_MUXMEASUREMENT_CONFIG,
                        (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT));
1406                }
1407                break;
1408
1409            } else if (ltc_state.substate == LTC_SEND_CLOCK_STCOMM_MUXMEASUREMENT_CONFIG) {
1410                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1411                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1412                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1413                    break;
1414                } else {
1415                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1416                }
1417
1418                ltc_state.check_spi_flag = FALSE;
1419                retVal = LTC_I2CClock(ltc_TXBufferClock, ltc_TXPECBufferClock);
1420                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1421                                             LTC_STATEMACH_STARTMEAS, LTC_ENTRY, ltc_state.gpioClocksTransferTime,
1422                                             LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1423                break;
1424            }
```

```
1425            break;
1426
1427        /*************************READ FROM PORT EXPANDER IO***********/
1428        case LTC_STATEMACH_USER_IO_FEEDBACK:
1429
1430            if (ltc_state.substate == LTC_USER_IO_READ_INPUT_REGISTER) {
1431                ltc_state.check_spi_flag = TRUE;
1432                SPI_SetTransmitOngoing();
1433                retVal = LTC_Send_I2C_Command(ltc_TXBuffer, ltc_TXPECbuffer, (uint8_t*)ltc_I2CcmdPortExpander1);
1434
1435                if (retVal != E_OK) {
1436                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1437                    ++ltc_state.muxmeas_seqptr;
1438                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1439                } else {
1440                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1441                    LTC_StateTransition(LTC_STATEMACH_USER_IO_FEEDBACK, LTC_USER_IO_SEND_CLOCK_STCOMM,
                        ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT);
1442                }
1443
1444                break;
1445
1446            } else if (ltc_state.substate == LTC_USER_IO_SEND_CLOCK_STCOMM) {
1447                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1448                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1449                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1450                    break;
1451                } else {
1452                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1453                }
1454
1455                ltc_state.check_spi_flag = FALSE;
1456                retVal = LTC_I2CClock(ltc_TXBufferClock, ltc_TXPECBufferClock);
1457                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1458                                             LTC_STATEMACH_USER_IO_FEEDBACK,
                                                 LTC_USER_IO_READ_I2C_TRANSMISSION_RESULT_RDCOMM,
                                                 ltc_state.gpioClocksTransferTime,
1459                                             LTC_STATEMACH_USER_IO_FEEDBACK,
                                                 LTC_USER_IO_READ_I2C_TRANSMISSION_RESULT_RDCOMM, LTC_STATEMACH_SHORTTIME);
1460                break;
1461
1462                } else if (ltc_state.substate == LTC_USER_IO_READ_I2C_TRANSMISSION_RESULT_RDCOMM) {
1463                    if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1464                        DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1465                        LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1466                        break;
1467                    } else {
1468                        DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1469                    }
1470
1471                    SPI_SetTransmitOngoing();
1472                    retVal = LTC_RX((uint8_t*)ltc_cmdRDCOMM, ltc_RXPECbuffer);
```

```
1473                    LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1474                                                 LTC_STATEMACH_USER_IO_FEEDBACK, LTC_USER_IO_SAVE_DATA,
                                                     ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT,
1475                                                 LTC_STATEMACH_USER_IO_FEEDBACK, LTC_USER_IO_SAVE_DATA,
                                                     LTC_STATEMACH_SHORTTIME);
1476                    break;
1477
1478                } else if (ltc_state.substate == LTC_USER_IO_SAVE_DATA) {
1479                    if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1480                        DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1481                        LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1482                        break;
1483                    } else {
1484                        DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1485                    }
1486
1487                    if (LTC_RX_PECCheck(ltc_RXPECbuffer) != E_OK) {
1488                        DIAG_Handler(DIAG_CH_LTC_PEC, DIAG_EVENT_NOK, 0);
1489                    } else {
1490                        DIAG_Handler(DIAG_CH_LTC_PEC, DIAG_EVENT_OK, 0);
1491                        LTC_PortExpanderSaveValues(ltc_RXPECbuffer);
1492                    }
1493
1494                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1495                    break;
1496                }
1497
1498            break;
1499
1500        /*****************************WRITE TO TI PORT EXPANDER IO***********/
1501        case LTC_STATEMACH_USER_IO_CONTROL_TI:
1502
1503            if (ltc_state.substate == LTC_USER_IO_SET_DIRECTION_REGISTER_TI) {
1504                ltc_state.check_spi_flag = TRUE;
1505                SPI_SetTransmitOngoing();
1506                retVal = LTC_SetPortExpanderDirection_TI(LTC_PORT_EXPANDER_TI_OUTPUT, ltc_TXBuffer, ltc_TXPECbuffer);
1507                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1508                    LTC_STATEMACH_USER_IO_CONTROL_TI, LTC_USER_IO_SEND_CLOCK_STCOMM_TI, LTC_STATEMACH_SHORTTIME,
1509                    LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1510                break;
1511
1512            } else if (ltc_state.substate == LTC_USER_IO_SEND_CLOCK_STCOMM_TI) {
1513                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1514                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1515                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1516                    break;
1517
1518                } else {
1519                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1520                }
1521
1522                ltc_state.check_spi_flag = FALSE;
```

```
1523                retVal = LTC_I2CClock(ltc_TXBufferClock, ltc_TXPECBufferClock);
1524                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1525                        LTC_STATEMACH_USER_IO_CONTROL_TI, LTC_USER_IO_SET_OUTPUT_REGISTER_TI,
                            ltc_state.gpioClocksTransferTime,
1526                        LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1527                break;
1528
1529            } else if (ltc_state.substate == LTC_USER_IO_SET_OUTPUT_REGISTER_TI) {
1530                ltc_state.check_spi_flag = TRUE;
1531                SPI_SetTransmitOngoing();
1532                retVal = LTC_SetPortExpander_Output_TI(ltc_TXBuffer, ltc_TXPECbuffer);
1533                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1534                        LTC_STATEMACH_USER_IO_CONTROL_TI, LTC_USER_IO_READ_I2C_TRANSMISSION_RESULT_RDCOMM_TI_SECOND,
                            ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT,
1535                        LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1536                break;
1537
1538            } else if (ltc_state.substate == LTC_USER_IO_READ_I2C_TRANSMISSION_RESULT_RDCOMM_TI_SECOND) {
1539                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1540                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1541                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1542                    break;
1543                } else {
1544                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1545                }
1546
1547                ltc_state.check_spi_flag = FALSE;
1548                retVal = LTC_I2CClock(ltc_TXBufferClock, ltc_TXPECBufferClock);
1549                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1550                     LTC_STATEMACH_STARTMEAS, LTC_ENTRY, ltc_state.gpioClocksTransferTime,
1551                     LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1552                break;
1553            }
1554            break;
1555
1556        /*************************READ TI PORT EXPANDER IO***********/
1557        case LTC_STATEMACH_USER_IO_FEEDBACK_TI:
1558
1559            if (ltc_state.substate == LTC_USER_IO_SET_DIRECTION_REGISTER_TI) {
1560                ltc_state.check_spi_flag = TRUE;
1561                SPI_SetTransmitOngoing();
1562                retVal = LTC_SetPortExpanderDirection_TI(LTC_PORT_EXPANDER_TI_INPUT, ltc_TXBuffer, ltc_TXPECbuffer);
1563                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1564                    LTC_STATEMACH_USER_IO_FEEDBACK_TI, LTC_USER_IO_SEND_CLOCK_STCOMM_TI, LTC_STATEMACH_SHORTTIME,
1565                    LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1566                break;
1567
1568            } else if (ltc_state.substate == LTC_USER_IO_SEND_CLOCK_STCOMM_TI) {
1569                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1570                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1571                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1572                    break;
```

```
1573
1574                } else {
1575                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1576                }
1577
1578                ltc_state.check_spi_flag = FALSE;
1579                retVal = LTC_I2CClock(ltc_TXBufferClock, ltc_TXPECBufferClock);
1580                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1581                        LTC_STATEMACH_USER_IO_FEEDBACK_TI, LTC_USER_IO_READ_INPUT_REGISTER_TI_FIRST,
                        ltc_state.gpioClocksTransferTime,
1582                        LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1583                break;
1584
1585            } else if (ltc_state.substate == LTC_USER_IO_READ_INPUT_REGISTER_TI_FIRST) {
1586                ltc_state.check_spi_flag = TRUE;
1587                SPI_SetTransmitOngoing();
1588                retVal = LTC_GetPortExpander_Input_TI(0, ltc_TXBuffer, ltc_TXPECbuffer);
1589                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1590                        LTC_STATEMACH_USER_IO_FEEDBACK_TI, LTC_USER_IO_READ_I2C_TRANSMISSION_RESULT_RDCOMM_TI_SECOND,
                        ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT,
1591                        LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1592                break;
1593
1594            } else if (ltc_state.substate == LTC_USER_IO_READ_I2C_TRANSMISSION_RESULT_RDCOMM_TI_SECOND) {
1595                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1596                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1597                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1598                    break;
1599                } else {
1600                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1601                }
1602
1603                ltc_state.check_spi_flag = FALSE;
1604                retVal = LTC_I2CClock(ltc_TXBufferClock, ltc_TXPECBufferClock);
1605                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1606                    LTC_STATEMACH_USER_IO_FEEDBACK_TI, LTC_USER_IO_READ_INPUT_REGISTER_TI_SECOND,
                    ltc_state.gpioClocksTransferTime,
1607                    LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1608                break;
1609            } else if (ltc_state.substate == LTC_USER_IO_READ_INPUT_REGISTER_TI_SECOND) {
1610                ltc_state.check_spi_flag = TRUE;
1611                SPI_SetTransmitOngoing();
1612                retVal = LTC_GetPortExpander_Input_TI(1, ltc_TXBuffer, ltc_TXPECbuffer);
1613                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1614                    LTC_STATEMACH_USER_IO_FEEDBACK_TI, LTC_USER_IO_READ_I2C_TRANSMISSION_RESULT_RDCOMM_TI_THIRD,
                    ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT,
1615                    LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1616                break;
1617
1618            } else if (ltc_state.substate == LTC_USER_IO_READ_I2C_TRANSMISSION_RESULT_RDCOMM_TI_THIRD) {
1619                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1620                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
```

```
1621                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1622                    break;
1623                } else {
1624                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1625                }
1626
1627                ltc_state.check_spi_flag = FALSE;
1628                retVal = LTC_I2CClock(ltc_TXBufferClock, ltc_TXPECBufferClock);
1629                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1630                    LTC_STATEMACH_USER_IO_FEEDBACK_TI, LTC_USER_IO_READ_I2C_TRANSMISSION_RESULT_RDCOMM_TI_FOURTH,
                        ltc_state.gpioClocksTransferTime,
1631                    LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1632                break;
1633            } else if (ltc_state.substate == LTC_USER_IO_READ_I2C_TRANSMISSION_RESULT_RDCOMM_TI_FOURTH) {
1634                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1635                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1636                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1637                    break;
1638                } else {
1639                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1640                }
1641
1642                SPI_SetTransmitOngoing();
1643                retVal = LTC_RX((uint8_t*)ltc_cmdRDCOMM, ltc_RXPECbuffer);
1644                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1645                    LTC_STATEMACH_USER_IO_FEEDBACK_TI, LTC_USER_IO_SAVE_DATA_TI,
                        ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT,
1646                    LTC_STATEMACH_USER_IO_FEEDBACK_TI, LTC_USER_IO_SAVE_DATA_TI, LTC_STATEMACH_SHORTTIME);
1647                break;
1648
1649            } else if (ltc_state.substate == LTC_USER_IO_SAVE_DATA_TI) {
1650                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1651                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1652                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1653                    break;
1654                } else {
1655                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1656                }
1657
1658                if (LTC_RX_PECCheck(ltc_RXPECbuffer) != E_OK) {
1659                    DIAG_Handler(DIAG_CH_LTC_PEC, DIAG_EVENT_NOK, 0);
1660                } else {
1661                    DIAG_Handler(DIAG_CH_LTC_PEC, DIAG_EVENT_OK, 0);
1662                    LTC_PortExpanderSaveValues_TI(ltc_RXPECbuffer);
1663                }
1664
1665                LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1666                break;
1667            }
1668
1669            break;
1670
```

```
1671        /*************************EEPROM READ*********************************/
1672        case LTC_STATEMACH_EEPROM_READ:
1673
1674            if (ltc_state.substate == LTC_EEPROM_READ_DATA1) {
1675                ltc_state.check_spi_flag = TRUE;
1676                SPI_SetTransmitOngoing();
1677                retVal = LTC_SendEEPROMReadCommand(ltc_TXBuffer, ltc_TXPECbuffer, 0);
1678
1679                if (retVal != E_OK) {
1680                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1681                    ++ltc_state.muxmeas_seqptr;
1682                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1683                } else {
1684                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1685                    LTC_StateTransition(LTC_STATEMACH_EEPROM_READ, LTC_EEPROM_SEND_CLOCK_STCOMM1,
                        ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT);
1686                }
1687
1688                break;
1689
1690            } else if (ltc_state.substate == LTC_EEPROM_SEND_CLOCK_STCOMM1) {
1691                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1692                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1693                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1694                    break;
1695                } else {
1696                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1697                }
1698
1699                SPI_SetTransmitOngoing();
1700                retVal = LTC_I2CClock(ltc_TXBufferClock, ltc_TXPECBufferClock);
1701                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1702                                             LTC_STATEMACH_EEPROM_READ, LTC_EEPROM_READ_DATA2,
                                             (ltc_state.gpioClocksTransferTime+LTC_TRANSMISSION_TIMEOUT),
1703                                             LTC_STATEMACH_EEPROM_READ, LTC_EEPROM_READ_DATA2, LTC_STATEMACH_SHORTTIME);
1704                break;
1705
1706            } else if (ltc_state.substate == LTC_EEPROM_READ_DATA2) {
1707                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1708                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1709                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1710                    break;
1711                } else {
1712                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1713                }
1714
1715                SPI_SetTransmitOngoing();
1716                retVal = LTC_SendEEPROMReadCommand(ltc_TXBuffer, ltc_TXPECbuffer, 1);
1717                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1718                                             LTC_STATEMACH_EEPROM_READ, LTC_EEPROM_SEND_CLOCK_STCOMM2,
                                             (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
1719                                             LTC_STATEMACH_EEPROM_READ, LTC_EEPROM_SEND_CLOCK_STCOMM2,
```

```
                                        LTC_STATEMACH_SHORTTIME);
1720                    break;
1721
1722                } else if (ltc_state.substate == LTC_EEPROM_SEND_CLOCK_STCOMM2) {
1723                    if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1724                        DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1725                        LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1726                        break;
1727                    } else {
1728                        DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1729                    }
1730
1731                    SPI_SetTransmitOngoing();
1732                    retVal = LTC_I2CClock(ltc_TXBufferClock, ltc_TXPECBufferClock);
1733                    LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1734                                                 LTC_STATEMACH_EEPROM_READ, LTC_EEPROM_READ_I2C_TRANSMISSION_RESULT_RDCOMM,
                                                 (ltc_state.gpioClocksTransferTime+LTC_TRANSMISSION_TIMEOUT),
1735                                                 LTC_STATEMACH_EEPROM_READ, LTC_EEPROM_READ_I2C_TRANSMISSION_RESULT_RDCOMM,
                                                 LTC_STATEMACH_SHORTTIME);
1736                    break;
1737                }  else if (ltc_state.substate == LTC_EEPROM_READ_I2C_TRANSMISSION_RESULT_RDCOMM) {
1738                    if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1739                        DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1740                        LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1741                        break;
1742                    } else {
1743                        DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1744                    }
1745
1746                    SPI_SetTransmitOngoing();
1747                    retVal = LTC_RX((uint8_t*)ltc_cmdRDCOMM, ltc_RXPECbuffer);
1748                    LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1749                                                 LTC_STATEMACH_EEPROM_READ, LTC_EEPROM_SAVE_READ,
                                                 (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
1750                                                 LTC_STATEMACH_EEPROM_READ, LTC_EEPROM_SAVE_READ, LTC_STATEMACH_SHORTTIME);
1751                    break;
1752
1753                } else if (ltc_state.substate == LTC_EEPROM_SAVE_READ) {
1754                    if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1755                        DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1756                        LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1757                        break;
1758                    } else {
1759                        DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1760                    }
1761
1762                    if (LTC_RX_PECCheck(ltc_RXPECbuffer) != E_OK) {
1763                        DIAG_Handler(DIAG_CH_LTC_PEC, DIAG_EVENT_NOK, 0);
1764                    } else {
1765                        DIAG_Handler(DIAG_CH_LTC_PEC, DIAG_EVENT_OK, 0);
1766                        LTC_EEPROMSaveReadValue(ltc_RXPECbuffer);
1767                    }
```

```
1768                LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1769                break;
1770            }
1771
1772            break;
1773
1774        /*************************EEPROM READ*********************************/
1775        case LTC_STATEMACH_EEPROM_WRITE:
1776
1777            if (ltc_state.substate == LTC_EEPROM_WRITE_DATA1) {
1778                ltc_state.check_spi_flag = TRUE;
1779                SPI_SetTransmitOngoing();
1780                retVal = LTC_SendEEPROMWriteCommand(ltc_TXBuffer, ltc_TXPECbuffer, 0);
1781
1782                if (retVal != E_OK) {
1783                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1784                    ++ltc_state.muxmeas_seqptr;
1785                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1786                } else {
1787                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1788                    LTC_StateTransition(LTC_STATEMACH_EEPROM_WRITE, LTC_EEPROM_SEND_CLOCK_STCOMM3,
                        (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT));
1789                }
1790
1791                break;
1792
1793            } else if (ltc_state.substate == LTC_EEPROM_SEND_CLOCK_STCOMM3) {
1794                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1795                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1796                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1797                    break;
1798                } else {
1799                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1800                }
1801
1802                SPI_SetTransmitOngoing();
1803                retVal = LTC_I2CClock(ltc_TXBufferClock, ltc_TXPECBufferClock);
1804                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1805                                             LTC_STATEMACH_EEPROM_WRITE, LTC_EEPROM_WRITE_DATA2,
                                             (ltc_state.gpioClocksTransferTime+LTC_TRANSMISSION_TIMEOUT),
1806                                             LTC_STATEMACH_EEPROM_WRITE, LTC_EEPROM_WRITE_DATA2, LTC_STATEMACH_SHORTTIME);
1807                break;
1808
1809            } else if (ltc_state.substate == LTC_EEPROM_WRITE_DATA2) {
1810                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1811                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1812                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1813                    break;
1814                } else {
1815                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1816                }
1817
```

```
1818                SPI_SetTransmitOngoing();
1819                retVal = LTC_SendEEPROMWriteCommand(ltc_TXBuffer, ltc_TXPECbuffer, 1);
1820                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1821                                             LTC_STATEMACH_EEPROM_WRITE, LTC_EEPROM_SEND_CLOCK_STCOMM4,
                                             (ltc_state.commandDataTransferTime+LTC_TRANSMISSION_TIMEOUT),
1822                                             LTC_STATEMACH_EEPROM_WRITE, LTC_EEPROM_SEND_CLOCK_STCOMM4,
                                             LTC_STATEMACH_SHORTTIME);
1823                break;
1824
1825            } else if (ltc_state.substate == LTC_EEPROM_SEND_CLOCK_STCOMM4) {
1826                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1827                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1828                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1829                    break;
1830                } else {
1831                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1832                }
1833
1834                SPI_SetTransmitOngoing();
1835                retVal = LTC_I2CClock(ltc_TXBufferClock, ltc_TXPECBufferClock);
1836                LTC_CondBasedStateTransition(retVal, DIAG_CH_LTC_SPI,
1837                                             LTC_STATEMACH_EEPROM_WRITE, LTC_EEPROM_FINISHED,
                                             (ltc_state.gpioClocksTransferTime+LTC_TRANSMISSION_TIMEOUT),
1838                                             LTC_STATEMACH_EEPROM_WRITE, LTC_EEPROM_FINISHED, LTC_STATEMACH_SHORTTIME);
1839                break;
1840            }  else if (ltc_state.substate == LTC_EEPROM_FINISHED) {
1841                if (ltc_state.timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
1842                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1843                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1844                    break;
1845                } else {
1846                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1847                }
1848                LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1849                break;
1850            }
1851
1852            break;
1853
1854        /*************************OPEN-WIRE CHECK*****************************/
1855        case LTC_STATEMACH_OPENWIRE_CHECK:
1856            if (ltc_state.substate == LTC_REQUEST_PULLUP_CURRENT_OPENWIRE_CHECK) {
1857                /* Run ADOW command with PUP = 1 */
1858                ltc_state.adcMode = LTC_OW_MEASUREMENT_MODE;
1859                ltc_state.check_spi_flag = FALSE;
1860                                                                    PUP = 1
1861                retVal = LTC_StartOpenWireMeasurement(ltc_state.adcMode, 1);
1862                if (retVal == E_OK) {
1863                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1864                    LTC_StateTransition(LTC_STATEMACH_OPENWIRE_CHECK, LTC_REQUEST_PULLUP_CURRENT_OPENWIRE_CHECK,
                    (ltc_state.commandDataTransferTime + LTC_Get_MeasurementTCycle(ltc_state.adcMode,
                    LTC_ADCMEAS_ALLCHANNEL)));
                                                Take the pullup measurement one more time.
```

```
1865                    ltc_state.resendCommandCounter--;
1866
1867                    /* Check how many retries are left */
1868                    if (ltc_state.resendCommandCounter == 0) {
1869                        /* Switch to read voltage state to read cell voltages */
1870                        LTC_StateTransition(LTC_STATEMACH_READVOLTAGE, LTC_READ_VOLTAGE_REGISTER_A_RDCVA_READVOLTAGE,
                            (ltc_state.commandDataTransferTime + LTC_Get_MeasurementTCycle(ltc_state.adcMode,
                            LTC_ADCMEAS_ALLCHANNEL)));
1871                        /* Reuse read voltage register */
1872                        ltc_state.reusageMeasurementMode = LTC_REUSE_READVOLT_FOR_ADOW_PUP;
1873                    }
1874                } else {
1875                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1876                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1877                }
1878            } else if (ltc_state.substate == LTC_READ_VOLTAGES_PULLUP_OPENWIRE_CHECK) {
1879                /* Previous state: Read voltage -> information stored in voltage buffer */
1880                ltc_state.reusageMeasurementMode = LTC_NOT_REUSED;
1881
1882                /* Copy data from voltage struct into open-wire struct */
1883                for (uint16_t i = 0; i < BS_NR_OF_BAT_CELLS; i++) {
1884                    ltc_openwire_pup_buffer[i] = ltc_cellvoltage.voltage[i];
1885                }
1886
1887                /* Set number of ADOW retries - send ADOW command with pull-down two times */
1888                ltc_state.resendCommandCounter = LTC_NMBR_REQ_ADOW_COMMANDS;
1889                LTC_StateTransition(LTC_STATEMACH_OPENWIRE_CHECK, LTC_REQUEST_PULLDOWN_CURRENT_OPENWIRE_CHECK,
                    LTC_STATEMACH_SHORTTIME);
1890
1891            } else if (ltc_state.substate == LTC_REQUEST_PULLDOWN_CURRENT_OPENWIRE_CHECK) {
1892                /* Run ADOW command with PUP = 0 */
1893                ltc_state.adcMode = LTC_OW_MEASUREMENT_MODE;
1894                ltc_state.check_spi_flag = FALSE;
1895
1896                retVal = LTC_StartOpenWireMeasurement(ltc_state.adcMode, 0);
1897                if (retVal == E_OK) {
1898                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_OK, 0);
1899                    LTC_StateTransition(LTC_STATEMACH_OPENWIRE_CHECK, LTC_REQUEST_PULLDOWN_CURRENT_OPENWIRE_CHECK,
                        (ltc_state.commandDataTransferTime + LTC_Get_MeasurementTCycle(ltc_state.adcMode,
                        LTC_ADCMEAS_ALLCHANNEL)));
1900                    ltc_state.resendCommandCounter--;
1901
1902                    /* Check how many retries are left */
1903                    if (ltc_state.resendCommandCounter == 0) {
1904                        /* Switch to read voltage state to read cell voltages */
1905                        LTC_StateTransition(LTC_STATEMACH_READVOLTAGE, LTC_READ_VOLTAGE_REGISTER_A_RDCVA_READVOLTAGE,
                            (ltc_state.commandDataTransferTime + LTC_Get_MeasurementTCycle(ltc_state.adcMode,
                            LTC_ADCMEAS_ALLCHANNEL)));
1906                        /* Reuse read voltage register */
1907                        ltc_state.reusageMeasurementMode = LTC_REUSE_READVOLT_FOR_ADOW_PDOWN;
1908                    }
1909                } else {
```

```
1910                    DIAG_Handler(DIAG_CH_LTC_SPI, DIAG_EVENT_NOK, 0);
1911                    LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1912                }
1913            } else if (ltc_state.substate == LTC_READ_VOLTAGES_PULLDOWN_OPENWIRE_CHECK) {   Comes from Line 875
1914                /* Previous state: Read voltage -> information stored in voltage buffer */
1915                ltc_state.reusageMeasurementMode = LTC_NOT_REUSED;
1916
1917                /* Copy data from voltage struct into open-wire struct */
1918                for (uint16_t i = 0; i < BS_NR_OF_BAT_CELLS; i++) {
1919                    ltc_openwire_pdown_buffer[i] = ltc_cellvoltage.voltage[i];
1920                }
1921                LTC_StateTransition(LTC_STATEMACH_OPENWIRE_CHECK, LTC_PERFORM_OPENWIRE_CHECK, LTC_STATEMACH_SHORTTIME);
1922            } else if (ltc_state.substate == LTC_PERFORM_OPENWIRE_CHECK) {
1923                /* Perform actual open-wire check */
1924                for (uint8_t m = 0; m < BS_NR_OF_MODULES; m++) {
1925                    /* Open-wire at C0: cell_pup(0) == 0 */
1926                    if (ltc_openwire_pup_buffer[0 + (m*BS_NR_OF_BAT_CELLS_PER_MODULE)] == 0) {
1927                        ltc_openwire.openwire[0 + (m*(BS_NR_OF_BAT_CELLS_PER_MODULE))] = 1;
1928                    }
1929                    /* Open-wire at Cmax: cell_pdown(BS_NR_OF_BAT_CELLS_PER_MODULE-1) == 0 */
1930                    if (ltc_openwire_pdown_buffer[((BS_NR_OF_BAT_CELLS_PER_MODULE-1) +
                    (m*BS_NR_OF_BAT_CELLS_PER_MODULE))] == 0) {
1931                        ltc_openwire.openwire[BS_NR_OF_BAT_CELLS_PER_MODULE + (m*BS_NR_OF_BAT_CELLS_PER_MODULE)] = 1;
1932                    }
1933                }
1934
1935                /* Take difference between pull-up and pull-down measurement */
1936                for (uint16_t i = 1; i < BS_NR_OF_BAT_CELLS; i++) {
1937                    ltc_openwire_delta[i] = (int32_t)(ltc_openwire_pup_buffer[i] - ltc_openwire_pdown_buffer[i]);
1938                }
1939
1940                /* Open-wire at C(N): delta cell(n+1) < -400mV */
1941                for (uint8_t m = 0; m < BS_NR_OF_MODULES; m++) {
1942                    for (uint8_t c = 1; c < BS_NR_OF_BAT_CELLS_PER_MODULE-1; c++) {
1943                        if (ltc_openwire_delta[c + (m*BS_NR_OF_BAT_CELLS_PER_MODULE)] < -400) {
1944                            ltc_openwire.openwire[c + (m*BS_NR_OF_BAT_CELLS_PER_MODULE)] = 1;
1945                        }
1946                    }
1947                }
1948
1949                /* Write database entry */
1950                DB_WriteBlock(&ltc_openwire, DATA_BLOCK_ID_OPEN_WIRE);
1951                /* Start new measurement cycle */
1952                LTC_StateTransition(LTC_STATEMACH_STARTMEAS, LTC_ENTRY, LTC_STATEMACH_SHORTTIME);
1953            }
1954            break;
1955
1956        /*************************DEFAULT*************************/
1957        default:
1958            break;
1959    }
1960
```

```
1961        ltc_state.triggerentry--;        /* reentrance counter */
1962    }
1963
1964
1965
1966    /**
1967     * @brief   saves the multiplexer values read from the LTC daisy-chain.
1968     *
1969     * After a voltage measurement was initiated on GPIO 1 to read the currently selected
1970     * multiplexer voltage, the results is read via SPI from the daisy-chain.
1971     * This function is called to store the result from the transmission in a buffer.
1972     *
1973     * @param   *DataBufferSPI_RX   buffer containing the data obtained from the SPI transmission
1974     * @param   muxseqptr           pointer to the multiplexer sequence, which configures the currently selected
         multiplexer ID and channel
1975     */
1976    static void LTC_SaveMuxMeasurement(uint8_t *rxBuffer, LTC_MUX_CH_CFG_s  *muxseqptr) {
1977        uint16_t i = 0;
1978        uint16_t val_ui = 0;
1979        int16_t temperature = 0;
1980        uint8_t sensor_idx = 0;
1981        uint8_t ch_idx = 0;
1982        uint32_t bitmask = 0;
1983
1984        /* pointer to measurement Sequence of Mux- and Channel-Configurations (1,0xFF)...(3,0xFF),(0,1),...(0,7)) */
1985        if (muxseqptr->muxCh == 0xFF)
1986            return; /* Channel 0xFF means that the multiplexer is deactivated, therefore no measurement will be made and
                saved*/
1987
1988        /* user multiplexer type -> connected to GPIO2! */
1989        if (muxseqptr->muxID == 1 || muxseqptr->muxID == 2) {
1990            for (i=0; i < LTC_N_LTC; i++) {
1991                if (muxseqptr->muxID == 1)
1992                    ch_idx = 0 + muxseqptr->muxCh;    /* channel index 0..7 */
1993                else
1994                    ch_idx = 8 + muxseqptr->muxCh;    /* channel index 8..15 */
1995
1996                if (ch_idx < LTC_N_USER_MUX_PER_LTC*LTC_N_MUX_CHANNELS_PER_MUX) {
1997                    val_ui =*((uint16_t *)(&rxBuffer[6+1*i*8]));        /* raw values, all mux on all LTCs */
1998                    ltc_user_mux.value[i*LTC_N_MUX_CHANNELS_PER_MUX*LTC_N_USER_MUX_PER_LTC+ch_idx] =
                        (uint16_t)(((float)(val_ui))*100e-6f*1000.0f);  /* Unit -> in V -> in mV */
1999                }                                          is equivalent to * 0.1
2000            }
2001        } else {
2002            /* temperature multiplexer type -> connected to GPIO1! */
2003            for (i=0; i < LTC_N_LTC; i++) {
2004                val_ui = *((uint16_t *)(&rxBuffer[4+i*8]));     Little endian.
2005                /* GPIO voltage in 100uV -> * 0.1 ----  conversion to V from mV * 0.001 ----- -> 0.0001 */
2006                temperature = (int16_t)LTC_Convert_MuxVoltages_to_Temperatures((float)(val_ui)*0.0001f);        /* Unit
                    Celsius */          This function is defined in the ltc_fig.c file located in \mcu-primary\src\module\config
2007                sensor_idx = ltc_muxsensortemperatur_cfg[muxseqptr->muxCh];
2008                /* if wrong configuration: exit and write nothing */
```

```
2009                if (sensor_idx >= BS_NR_OF_TEMP_SENSORS_PER_MODULE)
2010                    return;
2011                /* Set bitmask for valid flags */
2012                bitmask |= 1 < sensor_idx;
2013                /* Check LTC PEC error */
2014                if (LTC_ErrorTable[i].PEC_valid == TRUE) {
2015                    bitmask = ~bitmask;  /* negate bitmask to only validate flags of this cell voltage */
2016                    ltc_celltemperature.valid_temperature[i] &= bitmask;
2017                    ltc_celltemperature.temperature[i*(BS_NR_OF_TEMP_SENSORS_PER_MODULE)+sensor_idx] = temperature;
2018                } else {
2019                    ltc_celltemperature.valid_temperature[i] |= bitmask;
2020                }
2021            }
2022        }
2023    }
2024
2025
2026
2027    /**
2028     * @brief   saves the voltage values read from the LTC daisy-chain.
2029     *
2030     * After a voltage measurement was initiated to measure the voltages of the cells,
2031     * the result is read via SPI from the daisy-chain.
2032     * There are 6 register to read _(A,B,C,D,E,F) to get all cell voltages.
2033     * Only one register can be read at a time.
2034     * This function is called to store the result from the transmission in a buffer.
2035     *
2036     * @param   registerSet    voltage register that was read (voltage register A,B,C,D,E or F)
2037     * @param   *rxBuffer      buffer containing the data obtained from the SPI transmission
2038     * @param   PEC_valid      tells the functions if the PEC is valid or not, if not, manage indices but do not store
2039     *
2040     */
2041    static void LTC_SaveRXtoVoltagebuffer(uint8_t registerSet, uint8_t *rxBuffer) {
2042        uint16_t i = 0;
2043        uint16_t j = 0;
2044        uint16_t i_offset = 0;
2045        uint16_t voltage_index = 0;
2046        uint16_t val_ui = 0;
2047        uint16_t voltage = 0;
2048        uint8_t incrementations = 0;
2049        uint32_t bitmask = 0;
2050
2051        if (registerSet == 0) {
2052        /* RDCVA command -> voltage register group A */
2053            i_offset = 0;
2054        } else if (registerSet == 1) {
2055        /* RDCVB command -> voltage register group B */
2056            i_offset = 3;
2057        } else if (registerSet == 2) {
2058        /* RDCVC command -> voltage register group C */
2059            i_offset = 6;
2060        } else if (registerSet == 3) {
```

```
if (registerSet < 6) {
    i_offset = 3 * registerSet;
} else {
    return;
}
```

```
2061        /* RDCVD command -> voltage register group D */
2062            i_offset = 9;
2063        } else if (registerSet == 4) {
2064        /* RDCVD command -> voltage register group E (only for 15 and 18 cell version) */
2065            i_offset = 12;
2066        } else if (registerSet == 5) {
2067        /* RDCVD command -> voltage register group F (only for 18 cell version) */
2068            i_offset = 15;
2069        } else {
2070            return;
2071        }
2072
2073        /* Calculate bitmask for valid flags */
2074        bitmask |= 0x07 << i_offset;    /* 0x07: three voltages in each register */
2075
2076        /* reinitialize index counter at begin of cycle */
2077        if (i_offset == 0) {
2078            ltc_used_cells_index = 0;
2079        }
2080
2081        /* Retrieve data without command and CRC*/
2082        for (i=0; i < LTC_N_LTC; i++) {
2083            incrementations = 0;
2084
2085            /* parse all three voltages (3 * 2bytes) contained in one register */
2086            for (j=0; j < 3; j++) {
2087                /* index considering maximum number of cells */
2088                voltage_index = j+i_offset;
2089
2090                if ((ltc_voltage_input_used[voltage_index] == 1) && (ltc_used_cells_index <
                    BS_NR_OF_BAT_CELLS_PER_MODULE)) {
2091                    val_ui = *((uint16_t *)(&rxBuffer[4+2*j+i*8]));                    We just need to have the unit of 0.1 mV!
2092                    voltage = (uint16_t)(((float)(val_ui))*100e-6f*1000.0f);           /* Unit V -> in mV */
2093                    /* Check PEC for every LTC in the daisy-chain */
2094                    if (LTC_ErrorTable[i].PEC_valid == TRUE) {
2095                        ltc_cellvoltage.voltage[ltc_used_cells_index+i*(BS_NR_OF_BAT_CELLS_PER_MODULE)] = voltage;
2096                        bitmask = ~bitmask;  /* negate bitmask to only validate flags of this voltage register */
2097                        ltc_cellvoltage.valid_volt[(i/LTC_NUMBER_OF_LTC_PER_MODULE)] &= bitmask;
2098                    } else {                This division can lead to a big problem when this number is large, say, 3, since the array is defined as if ONE word for each module.
2099                        /* PEC_valid == FALSE: Invalidate only flags of this voltage register */
2100                        ltc_cellvoltage.valid_volt[(i/LTC_NUMBER_OF_LTC_PER_MODULE)] |= bitmask;
2101                    }
2102
2103                    ltc_used_cells_index++;
2104                    incrementations++;
2105
2106                    if (ltc_used_cells_index > BS_NR_OF_BAT_CELLS_PER_MODULE) {
2107                        return;
2108                    }
2109                }
2110            }
2111            /* restore start value for next module */
```

```
2112  #pragma GCC diagnostic push
2113          /* This warning is allowed for the edge case
2114           * LTC_N_LTC == 1
2115           */
2116  #pragma GCC diagnostic ignored "-Wtype-limits"
2117          if (i < LTC_N_LTC-1) {                Why?
2118  #pragma GCC diagnostic pop
2119              ltc_used_cells_index -= incrementations;
2120          }
2121      }
2122  }
2123
2124  /**
2125   * @brief   saves the GPIO voltage values read from the LTC daisy-chain.
2126   *
2127   * After a voltage measurement was initiated to measure the voltages on all GPIOs,
2128   * the result is read via SPI from the daisy-chain. In order to read the result of all GPIO measurements,
2129   * it is necessary to read auxiliary register A and B.
2130   * Only one register can be read at a time.
2131   * This function is called to store the result from the transmission in a buffer.
2132   *
2133   * @param   registerSet    voltage register that was read (auxiliary register A, B, C or D)
2134   * @param   *rxBuffer      buffer containing the data obtained from the SPI transmission
2135   *
2136   */
2137  static void LTC_SaveRXtoGPIOBuffer(uint8_t registerSet, uint8_t *rxBuffer) {
2138      uint16_t i = 0;
2139      uint8_t i_offset = 0;
2140      uint32_t bitmask = 0;
2141
2142      if (registerSet == 0) {
2143      /* RDAUXA command -> GPIO register group A */
2144          i_offset = 0;
2145          bitmask = 0x07 << i_offset;  /* 0x07: three temperatures in this register */
2146          /* Retrieve data without command and CRC*/
2147          for (i = 0; i < LTC_N_LTC; i++) {
2148              /* Check if PEC is valid */
2149              if (LTC_ErrorTable[i].PEC_valid == TRUE) {
2150                  bitmask = ~bitmask; /* negate bitmask to only validate flags of this voltage register */
2151                  ltc_allgpiovoltage.valid_gpiovoltages[i] &= bitmask;~bitmask;
2152                  /* values received in 100uV -> divide by 10 to convert to mV */
2153                  ltc_allgpiovoltage.gpiovoltage[0 + i_offset + BS_NR_OF_GPIOS_PER_MODULE*i]= *((uint16_t
                    *)(&rxBuffer[4+i*8]))/10;   This is better than the floating point operation in line 2092.
2154                  ltc_allgpiovoltage.gpiovoltage[1 + i_offset + BS_NR_OF_GPIOS_PER_MODULE*i]= *((uint16_t
                    *)(&rxBuffer[6+i*8]))/10;   Again, these is no need to change the unit to mV. 0.1 mV is just fine.
2155                  ltc_allgpiovoltage.gpiovoltage[2 + i_offset + BS_NR_OF_GPIOS_PER_MODULE*i]= *((uint16_t
                    *)(&rxBuffer[8+i*8]))/10;
2156              } else {
2157                  ltc_allgpiovoltage.valid_gpiovoltages[i] |= bitmask;
2158              }
2159          }
2160      } else if (registerSet == 1) {
```

```
2161            /* RDAUXB command -> GPIO register group B */
2162            i_offset = 3;
2163            bitmask = 0x03 << i_offset;  /* 0x03: two temperatures in this register */
2164            /* Retrieve data without command and CRC*/
2165            for (i = 0; i < LTC_N_LTC; i++) {
2166                /* Check if PEC is valid */
2167                if (LTC_ErrorTable[i].PEC_valid == TRUE) {
2168                    bitmask = ~bitmask; /* negate bitmask to only validate flags of this voltage register */
2169                    ltc_allgpiovoltage.valid_gpiovoltages[i] &= bitmask;
2170                    /* values received in 100uV -> divide by 10 to convert to mV */
2171                    ltc_allgpiovoltage.gpiovoltage[0 + i_offset + BS_NR_OF_GPIOS_PER_MODULE*i]= *((uint16_t
                    *)(&rxBuffer[4+i*8]))/10;    GPIO4
2172                    ltc_allgpiovoltage.gpiovoltage[1 + i_offset + BS_NR_OF_GPIOS_PER_MODULE*i]= *((uint16_t
                    *)(&rxBuffer[6+i*8]))/10;    GPIO5
2173                } else {
2174                    ltc_allgpiovoltage.valid_gpiovoltages[i] |= bitmask;
2175                }
2176            }
2177        } else if (registerSet == 2) {     This is for LTC6813.
2178            /* RDAUXC command -> GPIO register group C, for 18 cell version */
2179            i_offset = 5;
2180            bitmask = 0x07 << i_offset;  /* 0x07: three temperatures in this register */
2181            /* Retrieve data without command and CRC*/
2182            for (i = 0; i < LTC_N_LTC; i++) {
2183                /* Check if PEC is valid */
2184                if (LTC_ErrorTable[i].PEC_valid == TRUE) {
2185                    bitmask = ~bitmask; /* negate bitmask to only validate flags of this voltage register */
2186                    ltc_allgpiovoltage.valid_gpiovoltages[i] &= bitmask;
2187                    /* values received in 100uV -> divide by 10 to convert to mV */
2188                    ltc_allgpiovoltage.gpiovoltage[0 + i_offset + BS_NR_OF_GPIOS_PER_MODULE*i]= *((uint16_t
                    *)(&rxBuffer[4+i*8]))/10;
2189                    ltc_allgpiovoltage.gpiovoltage[1 + i_offset + BS_NR_OF_GPIOS_PER_MODULE*i]= *((uint16_t
                    *)(&rxBuffer[6+i*8]))/10;
2190                    ltc_allgpiovoltage.gpiovoltage[2 + i_offset + BS_NR_OF_GPIOS_PER_MODULE*i]= *((uint16_t
                    *)(&rxBuffer[8+i*8]))/10;
2191                } else {
2192                    ltc_allgpiovoltage.valid_gpiovoltages[i] |= bitmask;
2193                }
2194            }
2195        } else if (registerSet == 3) {
2196            /* RDAUXD command -> GPIO register group D, for 18 cell version */
2197            i_offset = 8;
2198            bitmask = 0x01 << i_offset;  /* 0x01: one temperature in this register */
2199            /* Retrieve data without command and CRC*/
2200            for (i = 0; i < LTC_N_LTC; i++) {
2201                /* Check if PEC is valid */
2202                if (LTC_ErrorTable[i].PEC_valid == TRUE) {
2203                    bitmask = ~bitmask; /* negate bitmask to only validate flags of this voltage register */
2204                    ltc_allgpiovoltage.valid_gpiovoltages[i] &= bitmask;
2205                    /* values received in 100uV -> divide by 10 to convert to mV */
2206                    ltc_allgpiovoltage.gpiovoltage[0 + i_offset + BS_NR_OF_GPIOS_PER_MODULE*i]= *((uint16_t
                    *)(&rxBuffer[4+i*8]))/10;
```

```
2207                } else {
2208                    ltc_allgpiovoltage.valid_gpiovoltages[i] |= bitmask;
2209                }
2210            }
2211        } else {
2212            return;
2213        }
2214    }
2215
2216
2217    /**
2218     * @brief   checks if the multiplexers acknowledged transmission.
2219     *
2220     * The RDCOMM command can be used to read the answer of the multiplexers to a
2221     * I2C transmission.
2222     * This function determines if the communication with the multiplexers was
2223     * successful or not.
2224     * The array LTC_ErrorTable is updated to locate the multiplexers that did not
2225     * acknowledge transmission.
2226     *
2227     * @param   *DataBufferSPI_RX    data obtained from the SPI transmission
2228     * @param   mux                  multiplexer to be addressed (multiplexer ID)
2229     *
2230     * @return  mux_error            0 is there was no error, 1 if there was errors
2231     */
2232    static uint8_t LTC_I2CCheckACK(uint8_t *DataBufferSPI_RX, int mux) {
2233        uint8_t mux_error = E_OK;
2234        uint16_t i = 0;
2235
2236        for (i=0; i < BS_NR_OF_MODULES; i++) {
2237            if (mux == 0) {
2238                if ((DataBufferSPI_RX[4+1+LTC_NUMBER_OF_LTC_PER_MODULE*i*8] & 0x0F) != 0x07) {  /* ACK = 0xX7 */
2239                    if (LTC_DISCARD_MUX_CHECK == FALSE) {
2240                        LTC_ErrorTable[i].mux0 = 1;
2241                    }
2242                    mux_error = E_NOT_OK;
2243                } else {
2244                    LTC_ErrorTable[i].mux0 = 0;
2245                }
2246            }
2247            if (mux == 1) {
2248                if ((DataBufferSPI_RX[4+1+LTC_NUMBER_OF_LTC_PER_MODULE*i*8] & 0x0F) != 0x27) {
2249                    if (LTC_DISCARD_MUX_CHECK == FALSE) {
2250                        LTC_ErrorTable[i].mux1 = 1;
2251                    }
2252                    mux_error = E_NOT_OK;
2253                } else {
2254                    LTC_ErrorTable[i].mux1 = 0;
2255                }
2256            }
2257            if (mux == 2) {
2258                if ((DataBufferSPI_RX[4+1+LTC_NUMBER_OF_LTC_PER_MODULE*i*8] & 0x0F) != 0x47) {
```

```
2259                    if (LTC_DISCARD_MUX_CHECK == FALSE) {
2260                        LTC_ErrorTable[i].mux2 = 1;
2261                    }
2262                    mux_error = E_NOT_OK;
2263                } else {
2264                    LTC_ErrorTable[i].mux2 = 0;
2265                }
2266            }
2267            if (mux == 3) {
2268                if ((DataBufferSPI_RX[4+1+LTC_NUMBER_OF_LTC_PER_MODULE*i*8] & 0x0F) != 0x67) {
2269                    if (LTC_DISCARD_MUX_CHECK == FALSE) {
2270                        LTC_ErrorTable[i].mux3 = 1;
2271                    }
2272                    mux_error = E_NOT_OK;
2273                } else {
2274                    LTC_ErrorTable[i].mux3 = 0;
2275                }
2276            }
2277        }
2278
2279        if (LTC_DISCARD_MUX_CHECK == TRUE) {
2280            return 0;
2281        } else {
2282            return mux_error;
2283        }
2284    }
2285
2286
2287
2288    /*
2289     * @brief   initialize the daisy-chain.
2290     *
2291     * To initialize the LTC6804 daisy-chain, a dummy byte (0x00) is sent.
2292     *
2293     * @return  retVal  E_OK if dummy byte was sent correctly by SPI, E_NOT_OK otherwise
2294     *
2295     */
2296    static STD_RETURN_TYPE_e LTC_Init(void) {
2297        STD_RETURN_TYPE_e statusSPI = E_NOT_OK;
2298        STD_RETURN_TYPE_e retVal = E_OK;
2299
2300        uint8_t PEC_Check[6];
2301        uint16_t PEC_result = 0;
2302        uint16_t i = 0;
2303
2304
2305        /* set REFON bit to 1 */
2306        /* data for the configuration */
2307        for (i=0; i < LTC_N_LTC; i++) {
2308            /* FC = disable all pull-downs, REFON = 1, DTEN = 0, ADCOPT = 0 */
2309            ltc_TXBuffer[0+(1*i)*6] = 0xFC;
2310            ltc_TXBuffer[1+(1*i)*6] = 0x00;
```

```
2311            ltc_TXBuffer[2+(1*i)*6] = 0x00;
2312            ltc_TXBuffer[3+(1*i)*6] = 0x00;
2313            ltc_TXBuffer[4+(1*i)*6] = 0x00;
2314            ltc_TXBuffer[5+(1*i)*6] = 0x00;
2315        }
2316
2317        /* now construct the message to be sent: it contains the wanted data, PLUS the needed PECs */
2318        ltc_TXPECbuffer[0] = ltc_cmdWRCFG[0];
2319        ltc_TXPECbuffer[1] = ltc_cmdWRCFG[1];
2320        ltc_TXPECbuffer[2] = ltc_cmdWRCFG[2];
2321        ltc_TXPECbuffer[3] = ltc_cmdWRCFG[3];
2322
2323        for (i=0; i < LTC_N_LTC; i++) {
2324            PEC_Check[0] = ltc_TXPECbuffer[4+i*8] = ltc_TXBuffer[0+i*6];
2325            PEC_Check[1] = ltc_TXPECbuffer[5+i*8] = ltc_TXBuffer[1+i*6];
2326            PEC_Check[2] = ltc_TXPECbuffer[6+i*8] = ltc_TXBuffer[2+i*6];
2327            PEC_Check[3] = ltc_TXPECbuffer[7+i*8] = ltc_TXBuffer[3+i*6];
2328            PEC_Check[4] = ltc_TXPECbuffer[8+i*8] = ltc_TXBuffer[4+i*6];
2329            PEC_Check[5] = ltc_TXPECbuffer[9+i*8] = ltc_TXBuffer[5+i*6];
2330
2331            PEC_result = LTC_pec15_calc(6, PEC_Check);
2332            ltc_TXPECbuffer[10+i*8]=(uint8_t)((PEC_result>>8)&0xff);
2333            ltc_TXPECbuffer[11+i*8]=(uint8_t)(PEC_result&0xff);
2334        }  /* end for */
2335
2336        statusSPI = LTC_SendData(ltc_TXPECbuffer);
2337
2338        if (statusSPI != E_OK) {
2339            retVal = E_NOT_OK;
2340        }
2341
2342        retVal = statusSPI;
2343
2344        return retVal;
2345    }
2346
2347
2348
2349
2350    /*
2351     * @brief   sets the balancing according to the control values read in the database.
2352     *
2353     * To set balancing for the cells, the corresponding bits have to be written in the configuration register.
2354     * The LTC driver only executes the balancing orders written by the BMS in the database.
2355     *
2356     * @param registerSet   Register Set, 0: cells 1 to 12 (WRCFG), 1: cells 13 to 15/18 (WRCFG2)
2357     *
2358     * @return              E_OK if dummy byte was sent correctly by SPI, E_NOT_OK otherwise
2359     *
2360     */
2361    static STD_RETURN_TYPE_e LTC_BalanceControl(uint8_t registerSet) {
2362        STD_RETURN_TYPE_e retVal = E_OK;
```

A more efficient code block (removing vairable PEC_Check):

```
for (int j = 0; j < 6; j++) {
    ltc_TXPECbuffer[4+j+i*8] = ltc_TXBuffer[j+i*6];
}

PEC_result = LTC_pec15_cal(6, &ltc_TXBuffer[i*6]);
...
```

To make sure the data is not changed in the LTC_pec15_cal function, the data can be declared as uint8_t const*

Higer byte of PEC first.

```
2363
2364        uint16_t i = 0;
2365        uint16_t j = 0;
2366
2367        LTC_Get_BalancingControlValues();
2368
2369        if (registerSet == 0) {  /* cells 1 to 12, WRCFG */
2370            for (j=0; j < BS_NR_OF_MODULES; j++) {
2371                i = BS_NR_OF_MODULES-j-1;    Data for the last module sent first.
2372
2373                /* FC = disable all pull-downs, REFON = 1 (reference always on), DTEN off, ADCOPT = 0 */
2374                ltc_TXBuffer[0+(i)*6] = 0xFC;
2375                ltc_TXBuffer[1+(i)*6] = 0x00;
2376                ltc_TXBuffer[2+(i)*6] = 0x00;
2377                ltc_TXBuffer[3+(i)*6] = 0x00;
2378                ltc_TXBuffer[4+(i)*6] = 0x00;
2379                ltc_TXBuffer[5+(i)*6] = 0x00;
2380                                                                    Can be done using a loop.
2381                if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+0] == 1) {
2382                    ltc_TXBuffer[4+(i)*6]|=0x01;
2383                }
2384                if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+1] == 1) {
2385                    ltc_TXBuffer[4+(i)*6]|=0x02;
2386                }
2387                if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+2] == 1) {
2388                    ltc_TXBuffer[4+(i)*6]|=0x04;
2389                }
2390                if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+3] == 1) {
2391                    ltc_TXBuffer[4+(i)*6]|=0x08;
2392                }
2393                if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+4] == 1) {
2394                    ltc_TXBuffer[4+(i)*6]|=0x10;
2395                }
2396                if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+5] == 1) {
2397                    ltc_TXBuffer[4+(i)*6]|=0x20;
2398                }
2399                if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+6] == 1) {
2400                    ltc_TXBuffer[4+(i)*6]|=0x40;
2401                }
2402                if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+7] == 1) {
2403                    ltc_TXBuffer[4+(i)*6]|=0x80;
2404                }
2405                if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+8] == 1) {
2406                    ltc_TXBuffer[5+(i)*6]|=0x01;
2407                }
2408                if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+9] == 1) {
2409                    ltc_TXBuffer[5+(i)*6]|=0x02;
2410                }
2411                if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+10] == 1) {
2412                    ltc_TXBuffer[5+(i)*6]|=0x04;
2413                }
2414                if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+11] == 1) {
```

```
2415                    ltc_TXBuffer[5+(i)*6]|=0x08;
2416                }
2417            }
2418            retVal = LTC_TX((uint8_t*)ltc_cmdWRCFG, ltc_TXBuffer, ltc_TXPECbuffer);
2419        } else if (registerSet == 1) {  /* cells 13 to 15/18 WRCFG2 */
2420            for (j=0; j < BS_NR_OF_MODULES; j++) {
2421                i = BS_NR_OF_MODULES-j-1;
2422
2423                /* 0x0F = disable pull-downs on GPIO6-9 */
2424                ltc_TXBuffer[0+(i)*6] = 0x0F;
2425                ltc_TXBuffer[1+(i)*6] = 0x00;
2426                ltc_TXBuffer[2+(i)*6] = 0x00;
2427                ltc_TXBuffer[3+(i)*6] = 0x00;
2428                ltc_TXBuffer[4+(i)*6] = 0x00;
2429                ltc_TXBuffer[5+(i)*6] = 0x00;
2430
2431                if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+12] == 1) {
2432                    ltc_TXBuffer[0+(i)*6] |= 0x10;
2433                }
2434                if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+13] == 1) {
2435                    ltc_TXBuffer[0+(i)*6] |= 0x20;
2436                }
2437                if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+14] == 1) {
2438                    ltc_TXBuffer[0+(i)*6] |= 0x40;
2439                }
2440                if (BS_NR_OF_BAT_CELLS_PER_MODULE > 15) {
2441                    if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+15] == 1) {
2442                        ltc_TXBuffer[0+(i)*6] |= 0x80;
2443                    }
2444                    if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+16] == 1) {
2445                        ltc_TXBuffer[1+(i)*6] |= 0x01;
2446                    }
2447                    if (ltc_balancing_control.balancing_state[j*(BS_NR_OF_BAT_CELLS_PER_MODULE)+17] == 1) {
2448                        ltc_TXBuffer[1+(i)*6] |= 0x02;
2449                    }
2450                }
2451            }
2452            retVal = LTC_TX((uint8_t*)ltc_cmdWRCFG2, ltc_TXBuffer, ltc_TXPECbuffer);
2453        } else {
2454            return E_NOT_OK;
2455        }
2456        return retVal;
2457    }
2458
2459
2460    /*
2461     * @brief   resets the error table.
2462     *
2463     * This function should be called during initialization or before starting a new measurement cycle
2464     *
2465     */
2466    static void LTC_ResetErrorTable(void) {
```

```
2467        uint16_t i = 0;
2468
2469        for (i=0; i < LTC_N_LTC; i++) {
2470            LTC_ErrorTable[i].PEC_valid = 0;
2471            LTC_ErrorTable[i].mux0 = 0;
2472            LTC_ErrorTable[i].mux1 = 0;
2473            LTC_ErrorTable[i].mux2 = 0;
2474            LTC_ErrorTable[i].mux3 = 0;
2475        }
2476    }
2477
2478
2479    /**
2480     * @brief   brief missing
2481     *
2482     * Gets the measurement time needed by the LTC chip, depending on the measurement mode and the number of channels.
2483     * For all cell voltages or all 5 GPIOS, the measurement time is the same.
2484     * For 2 cell voltages or 1 GPIO, the measurement time is the same.
2485     * As a consequence, this function is used for cell voltage and for GPIO measurement.
2486     *
2487     * @param   adcMode     LTC ADCmeasurement mode (fast, normal or filtered)
2488     * @param   adcMeasCh   number of channels measured for GPIOS (one at a time for multiplexers or all five GPIOs)
2489     *                      or number of cell voltage measured (2 cells or all cells)
2490     *
2491     * @return  retVal      measurement time in ms
2492     */
2493    static uint16_t LTC_Get_MeasurementTCycle(LTC_ADCMODE_e adcMode, LTC_ADCMEAS_CHAN_e  adcMeasCh) {
2494        uint16_t retVal = LTC_STATEMACH_MEAS_ALL_NORMAL_TCYCLE;  /* default */
2495
2496        if (adcMeasCh == LTC_ADCMEAS_ALLCHANNEL) {
2497            if (adcMode == LTC_ADCMODE_FAST_DCP0 || adcMode == LTC_ADCMODE_FAST_DCP1) {
2498                retVal = LTC_STATEMACH_MEAS_ALL_FAST_TCYCLE;
2499            } else if (adcMode == LTC_ADCMODE_NORMAL_DCP0 || adcMode == LTC_ADCMODE_NORMAL_DCP1) {
2500                retVal = LTC_STATEMACH_MEAS_ALL_NORMAL_TCYCLE;
2501            } else if (adcMode == LTC_ADCMODE_FILTERED_DCP0 || adcMode == LTC_ADCMODE_FILTERED_DCP1) {
2502                retVal = LTC_STATEMACH_MEAS_ALL_FILTERED_TCYCLE;
2503            }
2504        } else if (adcMeasCh == LTC_ADCMEAS_SINGLECHANNEL_GPIO1 || adcMeasCh == LTC_ADCMEAS_SINGLECHANNEL_GPIO2
2505                || adcMeasCh == LTC_ADCMEAS_SINGLECHANNEL_GPIO3 || adcMeasCh == LTC_ADCMEAS_SINGLECHANNEL_GPIO4
2506                || adcMeasCh == LTC_ADCMEAS_SINGLECHANNEL_GPIO5 || adcMeasCh == LTC_ADCMEAS_SINGLECHANNEL_TWOCELLS) {
2507            if (adcMode == LTC_ADCMODE_FAST_DCP0 || adcMode == LTC_ADCMODE_FAST_DCP1) {
2508                retVal = LTC_STATEMACH_MEAS_SINGLE_FAST_TCYCLE;
2509            } else if (adcMode == LTC_ADCMODE_NORMAL_DCP0 || adcMode == LTC_ADCMODE_NORMAL_DCP1) {
2510                retVal = LTC_STATEMACH_MEAS_SINGLE_NORMAL_TCYCLE;
2511            } else if (adcMode == LTC_ADCMODE_FILTERED_DCP0 || adcMode == LTC_ADCMODE_FILTERED_DCP1) {
2512                retVal = LTC_STATEMACH_MEAS_SINGLE_FILTERED_TCYCLE;
2513            }
2514        } else {
2515            retVal = LTC_STATEMACH_MEAS_ALL_NORMAL_TCYCLE;
2516        }
2517
2518        return retVal;
```

```
2519  }
2520
2521
2522  /**
2523   * @brief   tells the LTC daisy-chain to start measuring the voltage on all cells.
2524   *
2525   * This function sends an instruction to the daisy-chain via SPI, in order to start voltage measurement for all cells.
2526   *
2527   * @param   adcMode     LTC ADCmeasurement mode (fast, normal or filtered)
2528   * @param   adcMeasCh   number of cell voltage measured (2 cells or all cells)
2529   *
2530   * @return  retVal      E_OK if dummy byte was sent correctly by SPI, E_NOT_OK otherwise
2531   *
2532   */                                Cell
2533  static STD_RETURN_TYPE_e LTC_StartVoltageMeasurement(LTC_ADCMODE_e adcMode, LTC_ADCMEAS_CHAN_e adcMeasCh) {
2534      STD_RETURN_TYPE_e retVal = E_OK;
2535
2536      if (adcMeasCh == LTC_ADCMEAS_ALLCHANNEL) {
2537          if (adcMode == LTC_ADCMODE_FAST_DCP0) {
2538              retVal = LTC_SendCmd(ltc_cmdADCV_fast_DCP0);
2539          } else if (adcMode == LTC_ADCMODE_NORMAL_DCP0) {
2540              retVal = LTC_SendCmd(ltc_cmdADCV_normal_DCP0);
2541          } else if (adcMode == LTC_ADCMODE_FILTERED_DCP0) {
2542              retVal = LTC_SendCmd(ltc_cmdADCV_filtered_DCP0);
2543          } else if (adcMode == LTC_ADCMODE_FAST_DCP1) {
2544              retVal = LTC_SendCmd(ltc_cmdADCV_fast_DCP1);
2545          } else if (adcMode == LTC_ADCMODE_NORMAL_DCP1) {
2546              retVal = LTC_SendCmd(ltc_cmdADCV_normal_DCP1);
2547          } else if (adcMode == LTC_ADCMODE_FILTERED_DCP1) {
2548              retVal = LTC_SendCmd(ltc_cmdADCV_filtered_DCP1);
2549          } else {
2550              retVal = E_NOT_OK;
2551          }
2552      } else if (adcMeasCh == LTC_ADCMEAS_SINGLECHANNEL_TWOCELLS) {
2553          if (adcMode == LTC_ADCMODE_FAST_DCP0) {
2554              retVal = LTC_SendCmd(ltc_cmdADCV_fast_DCP0_twocells);
2555          } else {
2556              retVal = E_NOT_OK;
2557          }
2558      } else {
2559          retVal = E_NOT_OK;
2560      }
2561      return retVal;
2562  }
2563
2564
2565  /**
2566   * @brief   tells LTC daisy-chain to start measuring the voltage on GPIOS.
2567   *
2568   * This function sends an instruction to the daisy-chain via SPI to start the measurement.
2569   *
2570   * @param   adcMode     LTC ADCmeasurement mode (fast, normal or filtered)
```

```
2571     * @param   adcMeasCh   number of channels measured for GPIOS (one at a time, typically when multiplexers are used,
         or all five GPIOs)
2572     *
2573     * @return  retVal      E_OK if dummy byte was sent correctly by SPI, E_NOT_OK otherwise
2574     *
2575     */
2576    static STD_RETURN_TYPE_e LTC_StartGPIOMeasurement(LTC_ADCMODE_e adcMode, LTC_ADCMEAS_CHAN_e  adcMeasCh) {
2577        STD_RETURN_TYPE_e retVal;
2578
2579        if (adcMeasCh == LTC_ADCMEAS_ALLCHANNEL) {
2580            if (adcMode == LTC_ADCMODE_FAST_DCP0 || adcMode == LTC_ADCMODE_FAST_DCP1) {
2581                retVal = LTC_SendCmd(ltc_cmdADAX_fast_ALLGPIOS);
2582            } else if (adcMode == LTC_ADCMODE_FILTERED_DCP0 || adcMode == LTC_ADCMODE_FILTERED_DCP1) {
2583                retVal = LTC_SendCmd(ltc_cmdADAX_filtered_ALLGPIOS);
2584            } else {
2585                /*if (adcMode == LTC_ADCMODE_NORMAL_DCP0 || adcMode == LTC_ADCMODE_NORMAL_DCP1)*/
2586                retVal = LTC_SendCmd(ltc_cmdADAX_normal_ALLGPIOS);
2587            }
2588        } else if (adcMeasCh == LTC_ADCMEAS_SINGLECHANNEL_GPIO1) {
2589            /* Single Channel */
2590            if (adcMode == LTC_ADCMODE_FAST_DCP0 || adcMode == LTC_ADCMODE_FAST_DCP1) {
2591                retVal = LTC_SendCmd(ltc_cmdADAX_fast_GPIO1);
2592            } else if (adcMode == LTC_ADCMODE_FILTERED_DCP0 || adcMode == LTC_ADCMODE_FILTERED_DCP1) {
2593                retVal = LTC_SendCmd(ltc_cmdADAX_filtered_GPIO1);
2594            } else {
2595                /*if (adcMode == LTC_ADCMODE_NORMAL_DCP0 || adcMode == LTC_ADCMODE_NORMAL_DCP1)*/
2596
2597                retVal = LTC_SendCmd(ltc_cmdADAX_normal_GPIO1);
2598            }
2599        } else if (adcMeasCh == LTC_ADCMEAS_SINGLECHANNEL_GPIO2) {
2600            /* Single Channel */
2601            if (adcMode == LTC_ADCMODE_FAST_DCP0 || adcMode == LTC_ADCMODE_FAST_DCP1) {
2602                retVal = LTC_SendCmd(ltc_cmdADAX_fast_GPIO2);
2603            } else if (adcMode == LTC_ADCMODE_FILTERED_DCP0 || adcMode == LTC_ADCMODE_FILTERED_DCP1) {
2604                retVal = LTC_SendCmd(ltc_cmdADAX_filtered_GPIO2);
2605            } else {
2606                /*if (adcMode == LTC_ADCMODE_NORMAL_DCP0 || adcMode == LTC_ADCMODE_NORMAL_DCP1)*/
2607
2608                retVal = LTC_SendCmd(ltc_cmdADAX_normal_GPIO2);
2609            }
2610        } else if (adcMeasCh == LTC_ADCMEAS_SINGLECHANNEL_GPIO3) {
2611            /* Single Channel */
2612            if (adcMode == LTC_ADCMODE_FAST_DCP0 || adcMode == LTC_ADCMODE_FAST_DCP1) {
2613                retVal = LTC_SendCmd(ltc_cmdADAX_fast_GPIO3);
2614            } else if (adcMode == LTC_ADCMODE_FILTERED_DCP0 || adcMode == LTC_ADCMODE_FILTERED_DCP1) {
2615                retVal = LTC_SendCmd(ltc_cmdADAX_filtered_GPIO3);
2616            } else {
2617                /*if (adcMode == LTC_ADCMODE_NORMAL_DCP0 || adcMode == LTC_ADCMODE_NORMAL_DCP1)*/
2618
2619                retVal = LTC_SendCmd(ltc_cmdADAX_normal_GPIO3);
2620            }
2621        } else {
```

```
2622            retVal = E_NOT_OK;
2623        }
2624
2625        return retVal;
2626    }
2627
2628
2629    /**
2630     * @brief   tells LTC daisy-chain to start measuring the voltage on GPIOS.
2631     *
2632     * This function sends an instruction to the daisy-chain via SPI to start the measurement.
2633     *
2634     * @param   adcMode     LTC ADCmeasurement mode (fast, normal or filtered)
2635     * @param   PUP         pull-up bit for pull-up or pull-down current (0: pull-down, 1: pull-up)
2636     *
2637     * @return  retVal      E_OK if command was sent correctly by SPI, E_NOT_OK otherwise
2638     *
2639     */
2640    static STD_RETURN_TYPE_e LTC_StartOpenWireMeasurement(LTC_ADCMODE_e adcMode, uint8_t PUP) {
2641        STD_RETURN_TYPE_e retval = E_NOT_OK;
2642        if (PUP == 0) {
2643            /* pull-down current */
2644            if (adcMode == LTC_ADCMODE_NORMAL_DCP0) {
2645                retval = LTC_SendCmd(ltc2_BC_cmdADOW_PDOWN_normal_DCP0);
2646            } else if (adcMode == LTC_ADCMODE_FILTERED_DCP0) {
2647                retval = LTC_SendCmd(ltc2_BC_cmdADOW_PDOWN_filtered_DCP0);
2648            } else {
2649                retval = E_NOT_OK;
2650            }
2651        } else if (PUP == 1) {
2652            /* pull-up current */
2653            if (adcMode == LTC_ADCMODE_NORMAL_DCP0) {
2654                retval = LTC_SendCmd(ltc2_BC_cmdADOW_PUP_normal_DCP0);
2655            } else if (adcMode == LTC_ADCMODE_FILTERED_DCP0) {
2656                retval = LTC_SendCmd(ltc2_BC_cmdADOW_PUP_filtered_DCP0);
2657            } else {
2658                retval = E_NOT_OK;
2659            }
2660        }
2661        return retval;
2662    }
2663
2664
2665
2666    /**
2667     * @brief   checks if the data received from the daisy-chain is not corrupt.
2668     *
2669     * This function computes the PEC (CRC) from the data received by the daisy-chain.
2670     * It compares it with the PEC sent by the LTCs.
2671     * If there are errors, the array LTC_ErrorTable is updated to locate the LTCs in daisy-chain
2672     * that transmitted corrupt data.
2673     *
```

```
2674    * @param   *DataBufferSPI_RX_with_PEC   data obtained from the SPI transmission
2675    *
2676    * @return  retVal                       E_OK if PEC check is OK, E_NOT_OK otherwise
2677    *
2678    */
2679   static STD_RETURN_TYPE_e LTC_RX_PECCheck(uint8_t *DataBufferSPI_RX_with_PEC) {
2680       uint16_t i = 0;
2681       STD_RETURN_TYPE_e retVal = E_OK;
2682       uint8_t PEC_TX[2];
2683       uint16_t PEC_result = 0;
2684       uint8_t PEC_Check[6] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
2685
2686       /* check all PECs and put data without command and PEC in DataBufferSPI_RX (easier to use) */
2687       for (i=0; i < LTC_N_LTC; i++) {
2688           PEC_Check[0] = DataBufferSPI_RX_with_PEC[4+i*8];
2689           PEC_Check[1] = DataBufferSPI_RX_with_PEC[5+i*8];
2690           PEC_Check[2] = DataBufferSPI_RX_with_PEC[6+i*8];
2691           PEC_Check[3] = DataBufferSPI_RX_with_PEC[7+i*8];
2692           PEC_Check[4] = DataBufferSPI_RX_with_PEC[8+i*8];
2693           PEC_Check[5] = DataBufferSPI_RX_with_PEC[9+i*8];
2694
2695           PEC_result = LTC_pec15_calc(6, PEC_Check);
2696           PEC_TX[0]=(uint8_t)((PEC_result>>8)&0xff);
2697           PEC_TX[1]=(uint8_t)(PEC_result&0xff);
2698
2699           /* if calculated PEC not equal to received PEC */
2700           if ((PEC_TX[0] != DataBufferSPI_RX_with_PEC[10+i*8]) || (PEC_TX[1] != DataBufferSPI_RX_with_PEC[11+i*8])) {
2701               /* update error table of the corresponding LTC only if PEC check is activated */
2702               if (LTC_DISCARD_PEC == FALSE) {
2703                   LTC_ErrorTable[i].PEC_valid = FALSE;
2704               }
2705               retVal = E_NOT_OK;
2706
2707           } else {
2708               /* update error table of the corresponding LTC */
2709               LTC_ErrorTable[i].PEC_valid = TRUE;
2710           }
2711       }
2712
2713       if (LTC_DISCARD_PEC == TRUE) {
2714           return E_OK;
2715       } else {
2716           return (retVal);
2717       }
2718   }
2719
2720
2721   /**
2722    * @brief   send command to the LTC daisy-chain and receives data from the LTC daisy-chain.
2723    *
2724    * This is the core function to receive data from the LTC6804 daisy-chain.
2725    * A 2 byte command is sent with the corresponding PEC. Example: read configuration register (RDCFG).
```

```
2726    * Only command has to be set, the function calculates the PEC automatically.
2727    * The data send is:
2728    * 2 bytes (COMMAND) 2 bytes (PEC)
2729    * The data received is:
2730    * 6 bytes (LTC1) 2 bytes (PEC) + 6 bytes (LTC2) 2 bytes (PEC) + 6 bytes (LTC3) 2 bytes (PEC) + ... + 6 bytes
        (LTC{LTC_N_LTC}) 2 bytes (PEC)
2731    *
2732    * The function does not check the PECs. This has to be done elsewhere.
2733    *
2734    * @param   *Command                    command sent to the daisy-chain
2735    * @param   *DataBufferSPI_RX_with_PEC  data to sent to the daisy-chain, i.e. data to be sent + PEC
2736    *
2737    * @return  statusSPI                   E_OK if SPI transmission is OK, E_NOT_OK otherwise
2738    *
2739    */
2740   static STD_RETURN_TYPE_e LTC_RX(uint8_t *Command, uint8_t *DataBufferSPI_RX_with_PEC) {
2741       STD_RETURN_TYPE_e statusSPI = E_OK;
2742       uint16_t i = 0;
2743
2744       /* DataBufferSPI_RX_with_PEC contains the data to receive.
2745          The transmission function checks the PECs.
2746          It constructs DataBufferSPI_RX, which contains the received data without PEC (easier to use). */
2747
2748       for (i=0; i < LTC_N_BYTES_FOR_DATA_TRANSMISSION; i++) {
2749           ltc_TXPECbuffer[i] = 0x00;
2750       }
2751
2752       ltc_TXPECbuffer[0] = Command[0];
2753       ltc_TXPECbuffer[1] = Command[1];
2754       ltc_TXPECbuffer[2] = Command[2];
2755       ltc_TXPECbuffer[3] = Command[3];
2756
2757       statusSPI = LTC_ReceiveData(ltc_TXPECbuffer, DataBufferSPI_RX_with_PEC);
2758
2759       if (statusSPI != E_OK) {
2760           return E_NOT_OK;
2761       } else {
2762           return E_OK;
2763       }
2764   }
2765
2766
2767
2768   /**
2769    * @brief   sends command and data to the LTC daisy-chain.
2770    *
2771    * This is the core function to transmit data to the LTC6804 daisy-chain.
2772    * The data sent is:
2773    * COMMAND + 6 bytes (LTC1) + 6 bytes (LTC2) + 6 bytes (LTC3) + ... + 6 bytes (LTC{LTC_N_LTC})
2774    * A 2 byte command is sent with the corresponding PEC. Example: write configuration register (WRCFG).
2775    * THe command has to be set and then the function calculates the PEC automatically.
2776    * The function calculates the needed PEC to send the data to the daisy-chain. The sent data has the format:
```

```
2777    * 2 byte-COMMAND (2 bytes PEC) + 6 bytes (LTC1) (2 bytes PEC) + 6 bytes (LTC2) (2 bytes PEC) + 6 bytes (LTC3) (2
        bytes PEC) + ... + 6 bytes (LTC{LTC_N_LTC}) (2 bytes PEC)
2778    *
2779    * The function returns 0. The only way to check if the transmission was successful is to read the results of the
        write operation.
2780    * (example: read configuration register after writing to it)
2781    *
2782    * @param   *Command                    command sent to the daisy-chain
2783    * @param   *DataBufferSPI_TX           data to be sent to the daisy-chain
2784    * @param   *DataBufferSPI_TX_with_PEC  data to sent to the daisy-chain, i.e. data to be sent + PEC (calculated by
        the function)
2785    *
2786    * @return                             E_OK if SPI transmission is OK, E_NOT_OK otherwise
2787    *
2788    */
2789   static STD_RETURN_TYPE_e LTC_TX(uint8_t *Command, uint8_t *DataBufferSPI_TX, uint8_t *DataBufferSPI_TX_with_PEC) {
2790       uint16_t i = 0;
2791       STD_RETURN_TYPE_e statusSPI = E_NOT_OK;
2792       uint16_t PEC_result = 0;
2793       uint8_t PEC_Check[6] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
2794
2795       /*  DataBufferSPI_TX contains the data to send.
2796           The transmission function calculates the needed PEC.
2797           With it constructs DataBufferSPI_TX_with_PEC.
2798           It corresponds to the data effectively received/sent. */
2799       for (i=0; i < LTC_N_BYTES_FOR_DATA_TRANSMISSION; i++) {
2800           DataBufferSPI_TX_with_PEC[i] = 0x00;
2801       }
2802
2803       DataBufferSPI_TX_with_PEC[0] = Command[0];
2804       DataBufferSPI_TX_with_PEC[1] = Command[1];
2805       DataBufferSPI_TX_with_PEC[2] = Command[2];
2806       DataBufferSPI_TX_with_PEC[3] = Command[3];
2807
2808       /* Calculate PEC of all data (1 PEC value for 6 bytes) */
2809       for (i=0; i < LTC_N_LTC; i++) {
2810           PEC_Check[0] = DataBufferSPI_TX_with_PEC[4+i*8] = DataBufferSPI_TX[0+i*6];
2811           PEC_Check[1] = DataBufferSPI_TX_with_PEC[5+i*8] = DataBufferSPI_TX[1+i*6];
2812           PEC_Check[2] = DataBufferSPI_TX_with_PEC[6+i*8] = DataBufferSPI_TX[2+i*6];
2813           PEC_Check[3] = DataBufferSPI_TX_with_PEC[7+i*8] = DataBufferSPI_TX[3+i*6];
2814           PEC_Check[4] = DataBufferSPI_TX_with_PEC[8+i*8] = DataBufferSPI_TX[4+i*6];
2815           PEC_Check[5] = DataBufferSPI_TX_with_PEC[9+i*8] = DataBufferSPI_TX[5+i*6];
2816
2817           PEC_result = LTC_pec15_calc(6, PEC_Check);
2818           DataBufferSPI_TX_with_PEC[10+i*8]=(uint8_t)((PEC_result>>8)&0xff);
2819           DataBufferSPI_TX_with_PEC[11+i*8]=(uint8_t)(PEC_result&0xff);
2820       }
2821
2822       statusSPI = LTC_SendData(DataBufferSPI_TX_with_PEC);
2823
2824       if (statusSPI != E_OK) {
2825           return E_NOT_OK;
```

```
2826        } else {
2827            return E_OK;
2828        }
2829    }
2830
2831    /**
2832     * @brief   configures the data that will be sent to the LTC daisy-chain to configure multiplexer channels.
2833     *
2834     * This function does not sent the data to the multiplexer daisy-chain. This is done
2835     * by the function LTC_SetMuxChannel(), which calls LTC_SetMUXChCommand()..
2836     *
2837     * @param   *DataBufferSPI_TX      data to be sent to the daisy-chain to configure the multiplexer channels
2838     * @param   mux                    multiplexer ID to be configured (0,1,2 or 3)
2839     * @param   channel                multiplexer channel to be configured (0 to 7)
2840     *
2841     */
2842    static void LTC_SetMUXChCommand(uint8_t *DataBufferSPI_TX, uint8_t mux, uint8_t channel) {
2843        uint16_t i = 0;
2844
2845        for (i=0; i < LTC_N_LTC; i++) {
2846    #if SLAVE_BOARD_VERSION == 2
2847
2848            /* using ADG728 */
2849            uint8_t address = 0x4C | (mux % 3);
2850            uint8_t data = 1 << (channel % 8);
2851            if (channel == 0xFF) {  /* no channel selected, output of multiplexer is high impedance */
2852                data = 0x00;
2853            }
2854
2855    #else
2856
2857            /* using LTC1380 */
2858            uint8_t address = 0x48 | (mux % 4);
2859            uint8_t data = 0x08 | (channel % 8);
2860            if (channel == 0xFF) {  /* no channel selected, output of multiplexer is high impedance */
2861                data = 0x00;
2862            }
2863
2864    #endif
2865
2866            DataBufferSPI_TX[0 + i * 6] = LTC_ICOM_START | (address >> 3);        /* 0x6 : LTC6804: ICOM START from
            Master */
2867            DataBufferSPI_TX[1 + i * 6] = LTC_FCOM_MASTER_NACK | (address << 5);
2868            DataBufferSPI_TX[2 + i * 6] = LTC_ICOM_BLANK | (data >> 4);
2869            DataBufferSPI_TX[3 + i * 6] = LTC_FCOM_MASTER_NACK_STOP | (data << 4);
2870            DataBufferSPI_TX[4 + i * 6] = LTC_ICOM_NO_TRANSMIT;         /* 0x1 : ICOM-STOP */
2871            DataBufferSPI_TX[5 + i * 6] = 0x00;         /* 0x0 : dummy (Dn) */
2872                                                        /* 9: MASTER NACK + STOP (FCOM) */
2873        }
2874    }
2875
2876
```

```
2877
2878    /**
2879     * @brief   sends data to the LTC daisy-chain to read EEPROM on slaves.
2880     *
2881     *
2882     * @param        *DataBufferSPI_TX              data to be sent to the daisy-chain to configure the multiplexer
        channels
2883     * @param        *DataBufferSPI_TX_with_PEC     data to be sent to the daisy-chain to configure the multiplexer
        channels, with PEC (calculated by the function)
2884     * @param        step                           first or second stage of read process (0 or 1)
2885     * @param        address                        read address (18 bits)
2886     *
2887     * @return       E_OK if SPI transmission is OK, E_NOT_OK otherwise
2888     */
2889    static uint8_t LTC_SendEEPROMReadCommand(uint8_t *DataBufferSPI_TX, uint8_t *DataBufferSPI_TX_with_PEC, uint8_t step) {
2890        STD_RETURN_TYPE_e statusSPI = E_NOT_OK;
2891
2892        /* send WRCOMM to send I2C message to choose channel */
2893        LTC_SetEEPROMReadCommand(step, DataBufferSPI_TX);
2894        statusSPI = LTC_TX((uint8_t*)ltc_cmdWRCOMM, DataBufferSPI_TX, DataBufferSPI_TX_with_PEC);
2895
2896        if (statusSPI != E_OK) {
2897            return E_NOT_OK;
2898        } else {
2899            return E_OK;
2900        }
2901    }
2902
2903
2904
2905    /**
2906     * @brief   configures the data that will be sent to the LTC daisy-chain to read EEPROM on slaves.
2907     *
2908     * @param   step                    first or second stage of read process (0 or 1)
2909     * @param   *DataBufferSPI_TX       data to be sent to the daisy-chain
2910     * @param   address                 read address (18 bits)
2911     *
2912     */
2913    static void LTC_SetEEPROMReadCommand(uint8_t step, uint8_t *DataBufferSPI_TX) {
2914        uint16_t i = 0;
2915        uint32_t address = 0;
2916        uint8_t address0 = 0;
2917        uint8_t address1 = 0;
2918        uint8_t address2 = 0;
2919
2920        DB_ReadBlock(&ltc_slave_control, DATA_BLOCK_ID_SLAVE_CONTROL);
2921
2922        address = ltc_slave_control.eeprom_read_address_to_use;
2923
2924        address &= 0x3FFFF;
2925        address0 = address>>16;
2926        address1 = (address&0xFFFF)>>8;
```

```
2927        address2 = address&0xFF;
2928
2929        if (step == 0) {
2930            for (i=0; i < LTC_N_LTC; i++) {
2931                DataBufferSPI_TX[0 + i * 6] = LTC_ICOM_START | (0x0A);         /* 0x6 : LTC6804: ICOM START from Master */
2932                DataBufferSPI_TX[1 + i * 6] = LTC_FCOM_MASTER_NACK | (((address0 & 0x03) << 5) | 0x00);
2933                DataBufferSPI_TX[2 + i * 6] = LTC_ICOM_BLANK | (address1 >> 4);
2934                DataBufferSPI_TX[3 + i * 6] = LTC_FCOM_MASTER_NACK | (address1 << 4);
2935                DataBufferSPI_TX[4 + i * 6] = LTC_ICOM_BLANK | (address2 >> 4);
2936                DataBufferSPI_TX[5 + i * 6] = LTC_FCOM_MASTER_NACK | (address2 << 4);
2937            }
2938
2939        } else {  /* step == 1 */
2940            for (i=0; i < LTC_N_LTC; i++) {
2941                DataBufferSPI_TX[0 + i * 6] = LTC_ICOM_START | (0x0A);         /* 0x6 : LTC6804: ICOM START from Master */
2942                DataBufferSPI_TX[1 + i * 6] = LTC_FCOM_MASTER_NACK | (((address0 & 0x03) << 5) | 0x10);
2943                DataBufferSPI_TX[2 + i * 6] = LTC_ICOM_BLANK | 0x0F;
2944                DataBufferSPI_TX[3 + i * 6] = LTC_FCOM_MASTER_NACK_STOP | 0xF0;
2945                DataBufferSPI_TX[4 + i * 6] = LTC_ICOM_NO_TRANSMIT | 0x00;
2946                DataBufferSPI_TX[5 + i * 6] = LTC_FCOM_MASTER_NACK_STOP | 0x00;
2947            }
2948        }
2949    }
2950
2951
2952    /**
2953     * @brief   saves the read values of the external EEPROMs read from the LTC daisy-chain.
2954     *
2955     *
2956     * @param   *rxBuffer      buffer containing the data obtained from the SPI transmission
2957     *
2958     */
2959    static void LTC_EEPROMSaveReadValue(uint8_t *rxBuffer) {
2960        uint16_t i = 0;
2961
2962        DB_ReadBlock(&ltc_slave_control, DATA_BLOCK_ID_SLAVE_CONTROL);
2963
2964        for (i=0; i < LTC_N_LTC; i++) {
2965            ltc_slave_control.eeprom_value_read[i] = (rxBuffer[6+i*8] << 4)|((rxBuffer[7+i*8] >> 4));
2966        }
2967
2968        ltc_slave_control.eeprom_read_address_last_used = ltc_slave_control.eeprom_read_address_to_use;
2969        ltc_slave_control.eeprom_read_address_to_use = 0xFFFFFFFF;
2970
2971        DB_WriteBlock(&ltc_slave_control, DATA_BLOCK_ID_SLAVE_CONTROL);
2972    }
2973
2974
2975
2976    /**
2977     * @brief   sends data to the LTC daisy-chain to write EEPROM on slaves.
2978     *
```

```
2979     *
2980     * @param        *DataBufferSPI_TX              data to be sent to the daisy-chain to configure the multiplexer
         channels
2981     * @param        *DataBufferSPI_TX_with_PEC     data to be sent to the daisy-chain to configure the multiplexer
         channels, with PEC (calculated by the function)
2982     * @param        step                           first or second stage of read process (0 or 1)
2983     * @param        address                        read address (18 bits)
2984     *
2985     * @return       E_OK if SPI transmission is OK, E_NOT_OK otherwise
2986     */
2987    static uint8_t LTC_SendEEPROMWriteCommand(uint8_t *DataBufferSPI_TX, uint8_t *DataBufferSPI_TX_with_PEC, uint8_t
         step) {
2988        STD_RETURN_TYPE_e statusSPI = E_NOT_OK;
2989
2990        /* send WRCOMM to send I2C message to write EEPROM */
2991        LTC_SetEEPROMWriteCommand(step, DataBufferSPI_TX);
2992        statusSPI = LTC_TX((uint8_t*)ltc_cmdWRCOMM, DataBufferSPI_TX, DataBufferSPI_TX_with_PEC);
2993
2994        if (statusSPI != E_OK) {
2995            return E_NOT_OK;
2996        } else {
2997            return E_OK;
2998        }
2999    }
3000
3001
3002
3003    /**
3004     * @brief   configures the data that will be sent to the LTC daisy-chain to write EEPROM on slaves.
3005     *
3006     * @param   step                    first or second stage of read process (0 or 1)
3007     * @param   *DataBufferSPI_TX       data to be sent to the daisy-chain
3008     * @param   address                 read address (18 bits)
3009     *
3010     */
3011    static void LTC_SetEEPROMWriteCommand(uint8_t step, uint8_t *DataBufferSPI_TX) {
3012        uint16_t i = 0;
3013        uint32_t address = 0;
3014        uint8_t data = 0;
3015        uint8_t address0 = 0;
3016        uint8_t address1 = 0;
3017        uint8_t address2 = 0;
3018
3019        DB_ReadBlock(&ltc_slave_control, DATA_BLOCK_ID_SLAVE_CONTROL);
3020
3021        address = ltc_slave_control.eeprom_write_address_to_use;
3022
3023        address &= 0x3FFFF;
3024        address0 = address>>16;
3025        address1 = (address&0xFFFF)>>8;
3026        address2 = address&0xFF;
3027
```

```
3028        if (step == 0) {
3029            for (i=0; i < LTC_N_LTC; i++) {
3030                DataBufferSPI_TX[0 + i * 6] = LTC_ICOM_START | (0x0A);         /* 0x6 : LTC6804: ICOM START from Master */
3031                DataBufferSPI_TX[1 + i * 6] = LTC_FCOM_MASTER_NACK | (((address0 & 0x03) << 5) | 0x00);
3032                DataBufferSPI_TX[2 + i * 6] = LTC_ICOM_BLANK | (address1 >> 4);
3033                DataBufferSPI_TX[3 + i * 6] = LTC_FCOM_MASTER_NACK | (address1 << 4);
3034                DataBufferSPI_TX[4 + i * 6] = LTC_ICOM_BLANK | (address2 >> 4);
3035                DataBufferSPI_TX[5 + i * 6] = LTC_FCOM_MASTER_NACK | (address2 << 4);
3036            }
3037
3038        } else {  /* step == 1 */
3039            for (i=0; i < LTC_N_LTC; i++) {
3040                data = ltc_slave_control.eeprom_value_write[i];
3041
3042                DataBufferSPI_TX[0 + i * 6] = LTC_ICOM_BLANK | (data >> 4);         /* 0x6 : LTC6804: ICOM START from
                    Master */
3043                DataBufferSPI_TX[1 + i * 6] = LTC_FCOM_MASTER_NACK_STOP | (data << 4);
3044                DataBufferSPI_TX[2 + i * 6] = LTC_ICOM_NO_TRANSMIT | 0x00;
3045                DataBufferSPI_TX[3 + i * 6] = LTC_FCOM_MASTER_NACK_STOP | 0x00;
3046                DataBufferSPI_TX[4 + i * 6] = LTC_ICOM_NO_TRANSMIT | 0x00;
3047                DataBufferSPI_TX[5 + i * 6] = LTC_FCOM_MASTER_NACK_STOP | 0x00;
3048            }
3049
3050            ltc_slave_control.eeprom_write_address_last_used = ltc_slave_control.eeprom_write_address_to_use;
3051            ltc_slave_control.eeprom_write_address_to_use = 0xFFFFFFFF;
3052
3053            DB_WriteBlock(&ltc_slave_control, DATA_BLOCK_ID_SLAVE_CONTROL);
3054        }
3055    }
3056
3057
3058    /**
3059     * @brief   sends data to the LTC daisy-chain to configure multiplexer channels.
3060     *
3061     * This function calls the function LTC_SetMUXChCommand() to set the data.
3062     *
3063     * @param        *DataBufferSPI_TX              data to be sent to the daisy-chain to configure the multiplexer
        channels
3064     * @param        *DataBufferSPI_TX_with_PEC     data to be sent to the daisy-chain to configure the multiplexer
        channels, with PEC (calculated by the function)
3065     * @param        mux                    multiplexer ID to be configured (0,1,2 or 3)
3066     * @param        channel                multiplexer channel to be configured (0 to 7)
3067     *
3068     * @return       E_OK if SPI transmission is OK, E_NOT_OK otherwise
3069     */
3070    static uint8_t LTC_SetMuxChannel(uint8_t *DataBufferSPI_TX, uint8_t *DataBufferSPI_TX_with_PEC, uint8_t mux, uint8_t
        channel) {
3071        STD_RETURN_TYPE_e statusSPI = E_NOT_OK;
3072
3073        /* send WRCOMM to send I2C message to choose channel */
3074        LTC_SetMUXChCommand(DataBufferSPI_TX, mux, channel);
3075        statusSPI = LTC_TX((uint8_t*)ltc_cmdWRCOMM, DataBufferSPI_TX, DataBufferSPI_TX_with_PEC);
```

```
3076
3077        if (statusSPI != E_OK) {
3078            return E_NOT_OK;
3079        } else {
3080            return E_OK;
3081        }
3082    }
3083
3084
3085
3086    /**
3087     * @brief   sends data to the LTC daisy-chain to communicate via I2C
3088     *
3089     * This function initiates an I2C signal sent by the LTC6804 on the slave boards
3090     *
3091     * @param        *DataBufferSPI_TX              data to be sent to the daisy-chain to configure the EEPROM
3092     * @param        *DataBufferSPI_TX_with_PEC     data to be sent to the daisy-chain to configure the EEPROM, with PEC
        (calculated by the function)
3093     * @param        cmd_daa                        command data to be sent
3094     *
3095     * @return       E_OK if SPI transmission is OK, E_NOT_OK otherwise
3096     */
3097    static STD_RETURN_TYPE_e LTC_Send_I2C_Command(uint8_t *DataBufferSPI_TX, uint8_t *DataBufferSPI_TX_with_PEC, uint8_t
        *cmd_data) {
3098        STD_RETURN_TYPE_e statusSPI = E_NOT_OK;
3099
3100        uint16_t i = 0;
3101
3102        for (i=0; i < BS_NR_OF_MODULES; i++) {
3103            DataBufferSPI_TX[0+i*6] = cmd_data[0];
3104            DataBufferSPI_TX[1+i*6] = cmd_data[1];
3105
3106            DataBufferSPI_TX[2+i*6] = cmd_data[2];
3107            DataBufferSPI_TX[3+i*6] = cmd_data[3];
3108
3109            DataBufferSPI_TX[4+i*6] = cmd_data[4];
3110            DataBufferSPI_TX[5+i*6] = cmd_data[5];
3111        }
3112
3113        /* send WRCOMM to send I2C message */
3114        statusSPI = LTC_TX((uint8_t*)ltc_cmdWRCOMM, DataBufferSPI_TX, DataBufferSPI_TX_with_PEC);
3115
3116        if (statusSPI != E_OK) {
3117            return E_NOT_OK;
3118        } else {
3119            return E_OK;
3120        }
3121    }
3122
3123
3124    /**
3125     * @brief   saves the temperature value of the external temperature sensors read from the LTC daisy-chain.
```

```
3126     *
3127     * This function saves the temperature value received from the external temperature sensors
3128     *
3129     * @param   *rxBuffer      buffer containing the data obtained from the SPI transmission
3130     *
3131     */
3132    static void LTC_TempSensSaveTemp(uint8_t *rxBuffer) {
3133        uint16_t i = 0;
3134        uint8_t temp_tmp[2];
3135        uint16_t val_i = 0;
3136
3137        DB_ReadBlock(&ltc_slave_control, DATA_BLOCK_ID_SLAVE_CONTROL);
3138
3139        for (i=0; i < LTC_N_LTC; i++) {
3140            temp_tmp[0] = (rxBuffer[6+i*8] << 4) | ((rxBuffer[7+i*8] >> 4));
3141            temp_tmp[1] = (rxBuffer[8+i*8] << 4) | ((rxBuffer[9+i*8] >> 4));
3142            val_i = (temp_tmp[0] << 8) | (temp_tmp[1]);
3143            val_i = val_i>>8;
3144            ltc_slave_control.external_sensor_temperature[i] = val_i;
3145        }
3146
3147        DB_WriteBlock(&ltc_slave_control, DATA_BLOCK_ID_SLAVE_CONTROL);
3148    }
3149
3150    /**
3151     * @brief   sends data to the LTC daisy-chain to control the user port expander
3152     *
3153     * This function sends a control byte to the register of the user port expander
3154     *
3155     * @param        *DataBufferSPI_TX              data to be sent to the daisy-chain to configure the multiplexer
         channels
3156     * @param        *DataBufferSPI_TX_with_PEC     data to be sent to the daisy-chain to configure the multiplexer
         channels, with PEC (calculated by the function)
3157     *
3158     * @return       E_OK if SPI transmission is OK, E_NOT_OK otherwise
3159     */
3160    static uint8_t LTC_SetPortExpander(uint8_t *DataBufferSPI_TX, uint8_t *DataBufferSPI_TX_with_PEC) {
3161        STD_RETURN_TYPE_e statusSPI = E_NOT_OK;
3162
3163        uint16_t i = 0;
3164        uint8_t output_data = 0;
3165
3166        DB_ReadBlock(&ltc_slave_control, DATA_BLOCK_ID_SLAVE_CONTROL);
3167
3168        for (i=0; i < BS_NR_OF_MODULES; i++) {
3169            output_data = ltc_slave_control.io_value_out[BS_NR_OF_MODULES-1-i];
3170
3171            DataBufferSPI_TX[0+i*6] = LTC_ICOM_START | 0x04;     /* 6: ICOM0 start condition, 4: upper nibble of PCA8574
             address */
3172            DataBufferSPI_TX[1+i*6] = 0 | LTC_FCOM_MASTER_NACK;  /* 0: lower nibble of PCA8574 address + R/W bit, 8:
             FCOM0 master NACK */
3173
```

```
3174            DataBufferSPI_TX[2+i*6] = LTC_ICOM_BLANK | (output_data>>4);  /* 0: ICOM1 blank, x: upper nibble of PCA8574
                data register (0 == pin low) */
3175            DataBufferSPI_TX[3+i*6] = (uint8_t)(output_data << 4) | LTC_FCOM_MASTER_NACK_STOP;  /* x: lower nibble of
                PCA8574 data register, 9: FCOM1 master NACK + STOP */
3176
3177            DataBufferSPI_TX[4+i*6] = LTC_ICOM_NO_TRANSMIT;  /* 7: no transmission, F: dummy data */
3178            DataBufferSPI_TX[5+i*6] = 0;  /* F: dummy data, 9: FCOM2 master NACK + STOP */
3179        }
3180
3181        /* send WRCOMM to send I2C message */
3182        statusSPI = LTC_TX((uint8_t*)ltc_cmdWRCOMM, DataBufferSPI_TX, DataBufferSPI_TX_with_PEC);
3183
3184        if (statusSPI != E_OK) {
3185            return E_NOT_OK;
3186        } else {
3187            return E_OK;
3188        }
3189    }
3190
3191    /**
3192     * @brief   saves the received values of the external port expander read from the LTC daisy-chain.
3193     *
3194     * This function saves the received data byte from the external port expander
3195     *
3196     * @param   *rxBuffer      buffer containing the data obtained from the SPI transmission
3197     *
3198     */
3199    static void LTC_PortExpanderSaveValues(uint8_t *rxBuffer) {
3200        uint16_t i = 0;
3201        uint8_t val_i;
3202
3203        DB_ReadBlock(&ltc_slave_control, DATA_BLOCK_ID_SLAVE_CONTROL);
3204
3205        /* extract data */
3206        for (i=0; i < LTC_N_LTC; i++) {
3207            val_i = (rxBuffer[6+i*8] << 4) | ((rxBuffer[7+i*8] >> 4));
3208            ltc_slave_control.io_value_in[i] = val_i;
3209        }
3210
3211        DB_WriteBlock(&ltc_slave_control, DATA_BLOCK_ID_SLAVE_CONTROL);
3212    }
3213
3214
3215    /**
3216     * @brief   sends data to the LTC daisy-chain to control the user port expander from TI
3217     *
3218     * This function sends a control byte to the register of the user port expander from TI
3219     *
3220     * @param        *DataBufferSPI_TX              data to be sent to the daisy-chain to configure the multiplexer
        channels
3221     * @param        *DataBufferSPI_TX_with_PEC     data to be sent to the daisy-chain to configure the multiplexer
        channels, with PEC (calculated by the function)
```

```
3222     *
3223     * @return       E_OK if SPI transmission is OK, E_NOT_OK otherwise
3224     */
3225    static uint8_t LTC_SetPortExpanderDirection_TI(LTC_PORT_EXPANDER_TI_DIRECTION_e direction, uint8_t *DataBufferSPI_TX,
         uint8_t *DataBufferSPI_TX_with_PEC) {
3226        STD_RETURN_TYPE_e statusSPI = E_NOT_OK;
3227
3228        uint16_t i = 0;
3229
3230        DB_ReadBlock(&ltc_slave_control, DATA_BLOCK_ID_SLAVE_CONTROL);
3231
3232        for (i=0; i < BS_NR_OF_MODULES; i++) {
3233            DataBufferSPI_TX[0+i*6] = LTC_ICOM_START | 0x4;      /*upper nibble of TCA6408A address */
3234            DataBufferSPI_TX[1+i*6] = (uint8_t)( ( LTC_PORTEXPANDER_ADR_TI << 1) << 4) | LTC_FCOM_MASTER_NACK;  /* 0:
            lower nibble of TCA6408A address + R/W bit */
3235
3236            DataBufferSPI_TX[2+i*6] = LTC_ICOM_BLANK | (LTC_PORT_EXPANDER_TI_CONFIG_REG_ADR >> 4);  /* upper nibble of
            TCA6408A configuration register address */
3237            DataBufferSPI_TX[3+i*6] = (uint8_t)(LTC_PORT_EXPANDER_TI_CONFIG_REG_ADR << 4) | LTC_FCOM_MASTER_NACK;  /*
            lower nibble of TCA6408A configuration register address */
3238
3239            DataBufferSPI_TX[4+i*6] = LTC_ICOM_BLANK | (direction >> 4);  /* upper nibble of TCA6408A configuration
            register data */
3240            DataBufferSPI_TX[5+i*6] = (uint8_t)(direction << 4) | LTC_FCOM_MASTER_NACK_STOP;  /* lower nibble of TCA6408A
            configuration register data */
3241        }
3242
3243        /* send WRCOMM to send I2C message */
3244        statusSPI = LTC_TX((uint8_t*)ltc_cmdWRCOMM, DataBufferSPI_TX, DataBufferSPI_TX_with_PEC);
3245
3246        if (statusSPI != E_OK) {
3247            return E_NOT_OK;
3248        } else {
3249            return E_OK;
3250        }
3251    }
3252
3253
3254    /**
3255     * @brief   sends data to the LTC daisy-chain to control the user port expander from TI
3256     *
3257     * This function sends a control byte to the register of the user port expander from TI
3258     *
3259     * @param        *DataBufferSPI_TX              data to be sent to the daisy-chain to configure the multiplexer
         channels
3260     * @param        *DataBufferSPI_TX_with_PEC     data to be sent to the daisy-chain to configure the multiplexer
         channels, with PEC (calculated by the function)
3261     *
3262     * @return       E_OK if SPI transmission is OK, E_NOT_OK otherwise
3263     */
3264    static uint8_t LTC_SetPortExpander_Output_TI(uint8_t *DataBufferSPI_TX, uint8_t *DataBufferSPI_TX_with_PEC) {
3265        STD_RETURN_TYPE_e statusSPI = E_NOT_OK;
```

```
3266
3267        uint16_t i = 0;
3268        uint8_t output_data = 0;
3269
3270        DB_ReadBlock(&ltc_slave_control, DATA_BLOCK_ID_SLAVE_CONTROL);
3271
3272        for (i=0; i < BS_NR_OF_MODULES; i++) {
3273            output_data = ltc_slave_control.io_value_out[BS_NR_OF_MODULES-1-i];
3274
3275            DataBufferSPI_TX[0+i*6] = LTC_ICOM_START | 0x4;      /* upper nibble of TCA6408A address */
3276            DataBufferSPI_TX[1+i*6] = (uint8_t)( ( LTC_PORTEXPANDER_ADR_TI << 1) << 4) | LTC_FCOM_MASTER_NACK;  /* 0:
                lower nibble of TCA6408A address + R/W bit */
3277
3278            DataBufferSPI_TX[2+i*6] = LTC_ICOM_BLANK | (LTC_PORT_EXPANDER_TI_OUTPUT_REG_ADR >> 4);  /* upper nibble of
                TCA6408A output register address */
3279            DataBufferSPI_TX[3+i*6] = (uint8_t)(LTC_PORT_EXPANDER_TI_OUTPUT_REG_ADR << 4) | LTC_FCOM_MASTER_NACK;  /*
                lower nibble of TCA6408A output register address */
3280
3281            DataBufferSPI_TX[4+i*6] = LTC_ICOM_BLANK | (output_data >> 4);  /* upper nibble of TCA6408A output register */
3282            DataBufferSPI_TX[5+i*6] = (uint8_t)(output_data << 4) | LTC_FCOM_MASTER_NACK_STOP;  /* lower nibble of
                TCA6408A output register */
3283        }
3284
3285        /* send WRCOMM to send I2C message */
3286        statusSPI = LTC_TX((uint8_t*)ltc_cmdWRCOMM, DataBufferSPI_TX, DataBufferSPI_TX_with_PEC);
3287
3288        if (statusSPI != E_OK) {
3289            return E_NOT_OK;
3290        } else {
3291            return E_OK;
3292        }
3293    }
3294
3295    /**
3296     * @brief   sends data to the LTC daisy-chain to control the user port expander from TI
3297     *
3298     * This function sends a control byte to the register of the user port expander from TI
3299     *
3300     * @param        *DataBufferSPI_TX              data to be sent to the daisy-chain to configure the multiplexer
        channels
3301     * @param        *DataBufferSPI_TX_with_PEC     data to be sent to the daisy-chain to configure the multiplexer
        channels, with PEC (calculated by the function)
3302     *
3303     * @return       E_OK if SPI transmission is OK, E_NOT_OK otherwise
3304     */
3305    static uint8_t LTC_GetPortExpander_Input_TI(uint8_t step, uint8_t *DataBufferSPI_TX, uint8_t
        *DataBufferSPI_TX_with_PEC) {
3306        STD_RETURN_TYPE_e statusSPI = E_NOT_OK;
3307
3308        uint16_t i = 0;
3309
3310        if (step == 0) {
```

```
3311            for (i=0; i < BS_NR_OF_MODULES; i++) {
3312                DataBufferSPI_TX[0+i*6] = LTC_ICOM_START | 0x4;     /* upper nibble of TCA6408A address */
3313                DataBufferSPI_TX[1+i*6] = (uint8_t)( ( LTC_PORTEXPANDER_ADR_TI << 1) << 4) | LTC_FCOM_MASTER_NACK;  /*
                    lower nibble of TCA6408A address + R/W bit */
3314
3315                DataBufferSPI_TX[2+i*6] = LTC_ICOM_BLANK | (LTC_PORT_EXPANDER_TI_INPUT_REG_ADR >> 4);  /* upper nibble of
                    TCA6408A input register address */
3316                DataBufferSPI_TX[3+i*6] = (uint8_t)(LTC_PORT_EXPANDER_TI_INPUT_REG_ADR << 4) | LTC_FCOM_MASTER_NACK;  /*
                    x: lower nibble of TCA6408A input register address */
3317
3318                DataBufferSPI_TX[4+i*6] = LTC_ICOM_NO_TRANSMIT;  /* no transmission */
3319                DataBufferSPI_TX[5+i*6] = 0;  /* dummy data */
3320            }
3321
3322        } else {
3323            DB_ReadBlock(&ltc_slave_control, DATA_BLOCK_ID_SLAVE_CONTROL);
3324
3325            for (i=0; i < BS_NR_OF_MODULES; i++) {
3326                DataBufferSPI_TX[0+i*6] = LTC_ICOM_START | 0x4;     /* upper nibble of TCA6408A address */
3327                DataBufferSPI_TX[1+i*6] = (uint8_t)( ( (LTC_PORTEXPANDER_ADR_TI << 1) | 1) << 4) | LTC_FCOM_MASTER_NACK;
                    /* lower nibble of TCA6408A address + R/W bit */
3328
3329                DataBufferSPI_TX[2+i*6] = LTC_ICOM_BLANK | 0x0F;  /* upper nibble slave data, master pulls bus high */
3330                DataBufferSPI_TX[3+i*6] = LTC_FCOM_MASTER_NACK  | 0xF0;  /* lower nibble slave data, master pulls bus
                    high */
3331
3332                DataBufferSPI_TX[4+i*6] = LTC_ICOM_NO_TRANSMIT;  /* no transmission */
3333                DataBufferSPI_TX[5+i*6] = 0;  /* dummy data */
3334            }
3335        }
3336
3337        /* send WRCOMM to send I2C message */
3338        statusSPI = LTC_TX((uint8_t*)ltc_cmdWRCOMM, DataBufferSPI_TX, DataBufferSPI_TX_with_PEC);
3339
3340        if (statusSPI != E_OK) {
3341            return E_NOT_OK;
3342        } else {
3343            return E_OK;
3344        }
3345    }
3346
3347    /**
3348     * @brief   saves the received values of the external port expander from TI read from the LTC daisy-chain.
3349     *
3350     * This function saves the received data byte from the external port expander from TI
3351     *
3352     * @param   *rxBuffer      buffer containing the data obtained from the SPI transmission
3353     *
3354     */
3355    static void LTC_PortExpanderSaveValues_TI(uint8_t *rxBuffer) {
3356        uint16_t i = 0;
3357        uint8_t val_i;
```

```
3358
3359        DB_ReadBlock(&ltc_slave_control, DATA_BLOCK_ID_SLAVE_CONTROL);
3360
3361        /* extract data */
3362        for (i=0; i < LTC_N_LTC; i++) {
3363            val_i = (rxBuffer[6+i*8] << 4) | ((rxBuffer[7+i*8] >> 4));
3364            ltc_slave_control.io_value_in[i] = val_i;
3365        }
3366
3367        DB_WriteBlock(&ltc_slave_control, DATA_BLOCK_ID_SLAVE_CONTROL);
3368    }
3369
3370    /**
3371     * @brief   sends 72 clock pulses to the LTC daisy-chain.
3372     *
3373     * This function is used for the communication with the multiplexers via I2C on the GPIOs.
3374     * It send the command STCOMM to the LTC daisy-chain.
3375     *
3376     * @param   *DataBufferSPI_TX           data to be sent to the daisy-chain, set to 0xFF
3377     * @param   *DataBufferSPI_TX_with_PEC  data to be sent to the daisy-chain  with PEC (calculated by the function)
3378     *
3379     * @return  statusSPI                   E_OK if clock pulses were sent correctly by SPI, E_NOT_OK otherwise
3380     *
3381     */
3382    static STD_RETURN_TYPE_e LTC_I2CClock(uint8_t *DataBufferSPI_TX, uint8_t *DataBufferSPI_TX_with_PEC) {
3383        uint16_t i = 0;
3384        STD_RETURN_TYPE_e statusSPI = E_NOT_OK;
3385
3386        for (i=0; i < 4+9; i++) {
3387            DataBufferSPI_TX_with_PEC[i] = 0xFF;
3388        }
3389
3390        DataBufferSPI_TX_with_PEC[0] = ltc_cmdSTCOMM[0];
3391        DataBufferSPI_TX_with_PEC[1] = ltc_cmdSTCOMM[1];
3392        DataBufferSPI_TX_with_PEC[2] = ltc_cmdSTCOMM[2];
3393        DataBufferSPI_TX_with_PEC[3] = ltc_cmdSTCOMM[3];
3394
3395        statusSPI = LTC_SendI2CCmd(DataBufferSPI_TX_with_PEC);
3396
3397        return statusSPI;
3398    }
3399
3400
3401
3402    /**
3403     * @brief   gets the frequency of the SPI clock.
3404     *
3405     * This function reads the configuration from the SPI handle directly.
3406     *
3407     * @return    frequency of the SPI clock
3408     */
3409    static uint32_t LTC_GetSPIClock(void) {
```

```
3410        uint32_t SPI_Clock = 0;
3411
3412        if (LTC_SPI_INSTANCE == SPI2 || LTC_SPI_INSTANCE == SPI3) {
3413            /* SPI2 and SPI3 are connected to APB1 (PCLK1) */
3414            /* The prescaler setup bits LTC_SPI_PRESCALER corresponds to the bits 5:3 in the SPI_CR1 register */
3415            /* Reference manual p.909 */
3416            /* The shift by 3 puts the bits 5:3 to the first position */
3417            /* Division are made by powers of 2 which corresponds to shifting to the right */
3418            /* Then 0 corresponds to divide by 2, 1 corresponds to divide by 4... so 1 has to be added to the value of
                the configuration bits */
3419
3420            SPI_Clock = HAL_RCC_GetPCLK1Freq()>>((LTC_SPI_PRESCALER>>3)+1);
3421        }
3422
3423        if (LTC_SPI_INSTANCE == SPI1 || LTC_SPI_INSTANCE == SPI4 || LTC_SPI_INSTANCE == SPI5 || LTC_SPI_INSTANCE == SPI6) {
3424            /* SPI1, SPI4, SPI5 and SPI6 are connected to APB2 (PCLK2) */
3425            /* The prescaler setup bits LTC_SPI_PRESCALER corresponds to the bits 5:3 in the SPI_CR1 register */
3426            /* Reference manual p.909 */
3427            /* The shift by 3 puts the bits 5:3 to the first position */
3428            /* Division are made by powers of 2 which corresponds to shifting to the right */
3429            /* Then 0 corresponds to divide by 2, 1 corresponds to divide by 4... so 1 has to be added to the value of
                the configuration bits */
3430
3431            SPI_Clock = HAL_RCC_GetPCLK2Freq()>>((LTC_SPI_PRESCALER>>3)+1);
3432        }
3433
3434        return SPI_Clock;
3435    }
3436
3437
3438
3439
3440    /**
3441     * @brief   sets the transfer time needed to receive/send data with the LTC daisy-chain.
3442     *
3443     * This function gets the clock frequency and uses the number of LTCs in the daisy-chain.
3444     *
3445     */
3446    static void LTC_SetTransferTimes(void) {
3447        uint32_t transferTime_us = 0;
3448        uint32_t SPI_Clock = 0;
3449
3450        SPI_Clock = LTC_GetSPIClock();
3451
3452        /* Transmission of a command and data */
3453        /* Multiplication by 1000*1000 to get us */
3454        transferTime_us = (8*1000*1000)/(SPI_Clock);
3455        transferTime_us *= LTC_N_BYTES_FOR_DATA_TRANSMISSION;
3456        transferTime_us = transferTime_us + SPI_WAKEUP_WAIT_TIME;
3457        ltc_state.commandDataTransferTime = (transferTime_us/1000)+1;
3458
3459        /* Transmission of a command */
```

```
3460        /* Multiplication by 1000*1000 to get us */
3461        transferTime_us = ((4)*8*1000*1000)/(SPI_Clock);
3462        transferTime_us = transferTime_us + SPI_WAKEUP_WAIT_TIME;
3463        ltc_state.commandTransferTime = (transferTime_us/1000)+1;
3464
3465        /* Transmission of a command + 9 clocks */
3466        /* Multiplication by 1000*1000 to get us */
3467        transferTime_us = ((4+9)*8*1000*1000)/(SPI_Clock);
3468        transferTime_us = transferTime_us + SPI_WAKEUP_WAIT_TIME;
3469        ltc_state.gpioClocksTransferTime = (transferTime_us/1000)+1;
3470    }
3471
3472
3473
3474    /**
3475     * @brief   checks the state requests that are made.
3476     *
3477     * This function checks the validity of the state requests.
3478     * The resuls of the checked is returned immediately.
3479     *
3480     * @param   statereq    state request to be checked
3481     *
3482     * @return              result of the state request that was made, taken from LTC_RETURN_TYPE_e
3483     */
3484    static LTC_RETURN_TYPE_e LTC_CheckStateRequest(LTC_STATE_REQUEST_e statereq) {
3485        if (ltc_state.statereq == LTC_STATE_NO_REQUEST) {
3486            /* init only allowed from the uninitialized state */
3487            if (statereq == LTC_STATE_INIT_REQUEST) {
3488                if (ltc_state.state == LTC_STATEMACH_UNINITIALIZED) {
3489                    return LTC_OK;
3490                } else {
3491                    return LTC_ALREADY_INITIALIZED;
3492                }
3493            }
3494
3495            return LTC_OK;
3496
3497        } else {
3498            return LTC_REQUEST_PENDING;
3499        }
3500    }
3501
3502    static STD_RETURN_TYPE_e LTC_TimerElapsedAndSPITransmitOngoing(uint16_t timer) {
3503        STD_RETURN_TYPE_e retVal = E_NOT_OK;
3504        if (timer == 0 && SPI_IsTransmitOngoing() == TRUE) {
3505            retVal = E_OK;
3506        }
3507        return retVal;
3508    }
3509
3510
3511
```

```
3512  /**
3513   * @brief   gets the measurement initialization status.
3514   *
3515   * @return  retval  TRUE if a first measurement cycle was made, FALSE otherwise
3516   *
3517   */
3518  extern uint8_t LTC_IsFirstMeasurementCycleFinished(void) {
3519      uint8_t retval = FALSE;
3520
3521      OS_TaskEnter_Critical();
3522      retval    = ltc_state.first_measurement_made;
3523      OS_TaskExit_Critical();
3524
3525      return (retval);
3526  }
3527
3528  /**
3529   * @brief   sets the measurement initialization status.
3530   *
3531   * @return  none
3532   *
3533   */
3534  extern void LTC_SetFirstMeasurementCycleFinished(void) {
3535      OS_TaskEnter_Critical();
3536      ltc_state.first_measurement_made = TRUE;
3537      OS_TaskExit_Critical();
3538  }
3539
3540
3541
3542  /**
3543   * @brief   gets the measurement initialization status.
3544   *
3545   * @return  retval  TRUE if a first measurement cycle was made, FALSE otherwise
3546   *
3547   */
3548  extern STD_RETURN_TYPE_e LTC_GetMuxSequenceState(void) {
3549      STD_RETURN_TYPE_e retval = FALSE;
3550
3551      retval    = ltc_state.ltc_muxcycle_finished;
3552
3553      return (retval);
3554  }
3555
```