```
/**
 *
 * @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    bms.c
 * @author  foxBMS Team
 * @date    21.09.2015 (date of creation)
 * @ingroup ENGINE
 * @prefix  BMS
 *
 * @brief   bms driver implementation
 */


```

```c
/*================== Includes =============================================*/
#include "bms.h"

#include "bal.h"
#include "batterycell_cfg.h"
#include "batterysystem_cfg.h"
#include "database.h"
#include "diag.h"
#include "interlock.h"
#include "ltc_cfg.h"
#include "meas.h"
#include "os.h"
#include "plausibility.h"


/*================== Macros and Definitions ===============================*/

/**
 * Saves the last state and the last substate
 */
#define BMS_SAVELASTSTATES()    bms_state.laststate = bms_state.state; \
                                bms_state.lastsubstate = bms_state.substate;

/*================== Constant and Variable Definitions ====================*/

/**
 * contains the state of the contactor state machine
 */
static BMS_STATE_s bms_state = {
    .timer             = 0,
    .statereq          = BMS_STATE_NO_REQUEST,
    .state             = BMS_STATEMACH_UNINITIALIZED,
    .substate          = BMS_ENTRY,
    .currentFlowState  = BMS_RELAXATION,
    .laststate         = BMS_STATEMACH_UNINITIALIZED,
    .lastsubstate      = 0,
    .triggerentry      = 0,
    .ErrRequestCounter = 0,
    .initFinished      = E_NOT_OK,
    .restTimer_ms      = BS_RELAXATION_PERIOD_MS,
    .counter           = 0,
};

static DATA_BLOCK_CELLVOLTAGE_s bms_tab_cellvolt;
static DATA_BLOCK_CURRENT_SENSOR_s bms_tab_cur_sensor;
static DATA_BLOCK_MINMAX_s bms_tab_minmax;
static DATA_BLOCK_OPENWIRE_s bms_ow_tab;
static DATA_BLOCK_SOF_s bms_tab_sof;


/*================== Function Prototypes ==================================*/

```

Need to update BMS_STATEMACH_e to add Engine related states similar to the change of CONT_STATEMACH_e.

Also need to update BMS_STATEMACH_SUB_e to add Engine related state.

```
105  static BMS_RETURN_TYPE_e BMS_CheckStateRequest(BMS_STATE_REQUEST_e statereq);
106  static BMS_STATE_REQUEST_e BMS_GetStateRequest(void);
107  static BMS_STATE_REQUEST_e BMS_TransferStateRequest(void);
108  static uint8_t BMS_CheckReEntrance(void);
109  static uint8_t BMS_CheckCANRequests(void);
110  static STD_RETURN_TYPE_e BMS_CheckAnyErrorFlagSet(void);
111  static void BMS_UpdateBatsysState(DATA_BLOCK_CURRENT_SENSOR_s *curSensor);
112  static void BMS_GetMeasurementValues(void);
113  static void BMS_CheckVoltages(void);
114  static void BMS_CheckTemperatures(void);
115  static void BMS_CheckCurrent(void);
116  static void BMS_CheckSlaveTemperatures(void);
117  static void BMS_CheckOpenSenseWire(void);
118
119  /*================== Function Implementations ============================*/
120
121  /**
122   * @brief   re-entrance check of SYS state machine trigger function
123   *
124   * @details This function is not re-entrant and should only be called time- or event-triggered. It
125   *          increments the triggerentry counter from the state variable ltc_state. It should never
126   *          be called by two different processes, so if it is the case, triggerentry should never
127   *          be higher than 0 when this function is called.
128   *
129   * @return  retval  0 if no further instance of the function is active, 0xff else
130   */
131  static uint8_t BMS_CheckReEntrance(void) {
132      uint8_t retval = 0;
133      OS_TaskEnter_Critical();
134      if (!bms_state.triggerentry) {
135          bms_state.triggerentry++;
136      } else {
137          retval = 0xFF;  /* multiple calls of function */
138      }
139      OS_TaskExit_Critical();
140      return (retval);
141  }
142
143  /**
144   * @brief   gets the current state request.
145   *
146   * @details This function is used in the functioning of the SYS state machine.
147   *
148   * @return  current state request, taken from BMS_STATE_REQUEST_e
149   */
150  static BMS_STATE_REQUEST_e BMS_GetStateRequest(void) {
151      BMS_STATE_REQUEST_e retval = BMS_STATE_NO_REQUEST;
152
153      OS_TaskEnter_Critical();
154      retval    = bms_state.statereq;
155      OS_TaskExit_Critical();
156
```

Line 215

```
157     return (retval);
158 }
159
160
161 BMS_STATEMACH_e BMS_GetState(void) {
162     return (bms_state.state);
163 }
164
165
166 STD_RETURN_TYPE_e BMS_GetInitializationState(void) {
167     return (bms_state.initFinished);
168 }
169
170
171 /**
172  * @brief   transfers the current state request to the state machine.
173  *
174  * @details This function takes the current state request from cont_state and transfers it to th
175  *          state machine. It resets the value from cont_state to BMS_STATE_NO_REQUEST
176  *
177  * @return  retVal          current state request, taken from BMS_STATE_REQUEST_e
178  */
179 static BMS_STATE_REQUEST_e BMS_TransferStateRequest(void) {
180     BMS_STATE_REQUEST_e retval = BMS_STATE_NO_REQUEST;
181
182     OS_TaskEnter_Critical();
183     retval    = bms_state.statereq;
184     bms_state.statereq = BMS_STATE_NO_REQUEST;
185     OS_TaskExit_Critical();
186     return (retval);
187 }
188
189
190
191 BMS_RETURN_TYPE_e BMS_SetStateRequest(BMS_STATE_REQUEST_e statereq) {
192     BMS_RETURN_TYPE_e retVal = BMS_STATE_NO_REQUEST;
193
194     OS_TaskEnter_Critical();
195     retVal = BMS_CheckStateRequest(statereq);
196
197     if (retVal == BMS_OK) {
198             bms_state.statereq = statereq;
199     }
200     OS_TaskExit_Critical();
201
202     return (retVal);
203 }
204
205 /**
206  * @brief   checks the state requests that are made.
207  *
208  * @details This function checks the validity of the state requests. The results of the checked is
```

```
209     *          returned immediately.
210     *
211     * @param   statereq    state request to be checked
212     *
213     * @return  result of the state request that was made, taken from BMS_RETURN_TYPE_e
214     */
215    static BMS_RETURN_TYPE_e BMS_CheckStateRequest(BMS_STATE_REQUEST_e statereq) {
216        if (statereq == BMS_STATE_ERROR_REQUEST) {
217            return BMS_OK;
218        }
219
220        if (bms_state.statereq == BMS_STATE_NO_REQUEST) {
221            /* init only allowed from the uninitialized state */
222            if (statereq == BMS_STATE_INIT_REQUEST) {
223                if (bms_state.state == BMS_STATEMACH_UNINITIALIZED) {
224                    return BMS_OK;
225                } else {
226                    return BMS_ALREADY_INITIALIZED;
227                }
228            } else {
229                return BMS_ILLEGAL_REQUEST;
230            }
231        } else {
232            return BMS_REQUEST_PENDING;
233        }
234    }
235
236    void BMS_Trigger(void) {
237        BMS_STATE_REQUEST_e statereq = BMS_STATE_NO_REQUEST;
238        CONT_STATEMACH_e contstate = CONT_STATEMACH_UNDEFINED;
239        DATA_BLOCK_SYSTEMSTATE_s systemstate = {0};
240        uint32_t timestamp = OS_getOSSysTick();
241        static uint32_t nextOpenWireCheck = 0;
242
243        DIAG_SysMonNotify(DIAG_SYSMON_BMS_ID, 0);  /* task is running, state = ok */
244
245        if (bms_state.state != BMS_STATEMACH_UNINITIALIZED) {
246            BMS_GetMeasurementValues();
247            BMS_UpdateBatsysState(&bms_tab_cur_sensor);
248            BMS_CheckVoltages();
249            BMS_CheckTemperatures();
250            BMS_CheckCurrent();
251            BMS_CheckSlaveTemperatures();
252            BMS_CheckOpenSenseWire();
253
254            /* Plausibility check */
255            // Commented out by JHL
256            // PL_CheckPackvoltage(&bms_tab_cellvolt, &bms_tab_cur_sensor);
257        }
258        /* Check re-entrance of function */
259        if (BMS_CheckReEntrance()) {
260            return;
```

Can only have BMS_STATE_NO/INIT_REQUEST; See Lines 276/281. For this reason, the name of this function should have been named as BMS_CheckNo_InitStateRequest.

Line 722. Get the measurement values from the database.

~~Change this to measure the GPIOs and then the temperature.~~

We can uncomment it later when we set up the system correctly.

```
261         }
262
263         if (bms_state.timer) {
264             if (--bms_state.timer) {
265                 bms_state.triggerentry--;
266                 return;     /* handle state machine only if timer has elapsed */
267             }
268         }
269
270         /****Happens every time the state machine is triggered*************/
271         switch (bms_state.state) {
272             /*************************UNINITIALIZED*********************************/
273             case BMS_STATEMACH_UNINITIALIZED:
274                 /* waiting for Initialization Request */
275                 statereq = BMS_TransferStateRequest();
276                 if (statereq == BMS_STATE_INIT_REQUEST) {
277                     BMS_SAVELASTSTATES();
278                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
279                     bms_state.state = BMS_STATEMACH_INITIALIZATION;
280                     bms_state.substate = BMS_ENTRY;
281                 } else if (statereq == BMS_STATE_NO_REQUEST) {
282                     /* no actual request pending */
283                 } else {
284                     bms_state.ErrRequestCounter++;  /* illegal request pending */
285                 }
286                 break;
287
288
289             /*************************INITIALIZATION*********************************/
290             case BMS_STATEMACH_INITIALIZATION:
291                 BMS_SAVELASTSTATES();
292
293                 bms_state.timer = BMS_STATEMACH_LONGTIME_MS;
294                 bms_state.state = BMS_STATEMACH_INITIALIZED;
295                 bms_state.substate = BMS_ENTRY;
296
297                 break;
298
299             /*************************INITIALIZED*********************************/
300             case BMS_STATEMACH_INITIALIZED:
301                 BMS_SAVELASTSTATES();
302                 bms_state.initFinished = E_OK;
303                 bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
304                 bms_state.state = BMS_STATEMACH_IDLE;
305                 bms_state.substate = BMS_ENTRY;
306                 break;
307
308             /*************************IDLE*********************
309             case BMS_STATEMACH_IDLE:
310                 BMS_SAVELASTSTATES();
311
312                 if (bms_state.substate == BMS_ENTRY) {
```

cansignal_cfg.h | cansignal_cfg.c | database_cfg.h | bms.c | contactor.c

mcu-primary > src > engine > config > database_cfg.h > DATA_BLOCK_SYSTEMSTATE_s

```
546 typedef struct {
547     /* Timestamp info needs to be at the beginning. Automatically written on DB_WriteBlock
548     uint32_t timestamp;                 /*!< timestamp of database entry
549     uint32_t previous_timestamp;        /*!< timestamp of last database entry
550     uint8_t bms_state;                  /*!< system state (e.g., standby, normal)
551 } DATA_BLOCK_SYSTEMSTATE_s;      You, a month ago • Add all foxBMS files
```

```
313                 DB_ReadBlock(&systemstate, DATA_BLOCK_ID_SYSTEMSTATE);
314                 systemstate.bms_state = BMS_STATEMACH_IDLE;
315                 DB_WriteBlock(&systemstate, DATA_BLOCK_ID_SYSTEMSTATE);
316                 bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
317                 bms_state.substate = BMS_CHECK_ERROR_FLAGS;
318                 break;
319             } else if (bms_state.substate == BMS_CHECK_ERROR_FLAGS) {
320                 if (BMS_CheckAnyErrorFlagSet() == E_NOT_OK) {
321                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
322                     bms_state.state = BMS_STATEMACH_ERROR;
323                     bms_state.substate = BMS_ENTRY;
324                     break;
325                 } else {
326                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
327                     bms_state.substate = BMS_CHECK_STATE_REQUESTS;
328                     break;
329                 }
330             } else if (bms_state.substate == BMS_CHECK_STATE_REQUESTS) {
331                 if (BMS_CheckCANRequests() == BMS_REQ_ID_STANDBY) {
332                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
333                     bms_state.state = BMS_STATEMACH_STANDBY;
334                     bms_state.substate = BMS_ENTRY;
335                     break;
336                 } else {
337                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
338                     bms_state.substate = BMS_CHECK_ERROR_FLAGS;
339                     break;
340                 }
341             }
342             break;
343
344
345         /*****************************STANDBY*************************************/
346         case BMS_STATEMACH_STANDBY:
347             BMS_SAVELASTSTATES();
348
349             if (bms_state.substate == BMS_ENTRY) {
350                 BAL_SetStateRequest(BAL_STATE_ALLOWBALANCING_REQUEST);
351 #if BUILD_MODULE_ENABLE_CONTACTOR == 1
352                 CONT_SetStateRequest(CONT_STATE_STANDBY_REQUEST);
353 #endif /* BUILD_MODULE_ENABLE_CONTACTOR == 1 */
354 #if BUILD_MODULE_ENABLE_ILCK == 1
355                 ILCK_SetStateRequest(ILCK_STATE_CLOSE_REQUEST);
356 #endif /* BUILD_MODULE_ENABLE_ILCK == 1 */
357 #if LTC_STANDBY_PERIODIC_OPEN_WIRE_CHECK == TRUE
358                 nextOpenWireCheck = timestamp + LTC_STANDBY_OPEN_WIRE_PERIOD_ms;
359 #endif /* LTC_STANDBY_PERIODIC_OPEN_WIRE_CHECK == TRUE */
360                 bms_state.timer = BMS_STATEMACH_MEDIUMTIME_MS;
361                 bms_state.substate = BMS_CHECK_ERROR_FLAGS_INTERLOCK;
362                 DB_ReadBlock(&systemstate, DATA_BLOCK_ID_SYSTEMSTATE);
363                 systemstate.bms_state = BMS_STATEMACH_STANDBY;
364                 DB_WriteBlock(&systemstate, DATA_BLOCK_ID_SYSTEMSTATE);
```

The first bms_state here can be renamed as bms_specific_state or bms_current_state

Checking if there state request for state transition.

Contactor goes to Standby state

Only update time during the entry

These three lines appeared a couple of times in the code. It is a good idea to change these lines to a function named BMS_UpdateBMS_State_inSystemState(newState)

```
365                 break;
366             } else if (bms_state.substate == BMS_CHECK_ERROR_FLAGS_INTERLOCK) {
367                 if (BMS_CheckAnyErrorFlagSet() == E_NOT_OK) {
368                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
369                     bms_state.state = BMS_STATEMACH_ERROR;
370                     bms_state.substate = BMS_ENTRY;
371                     break;
372                 } else {
373                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
374                     bms_state.substate = BMS_INTERLOCK_CHECKED;
375                     break;
376                 }
377             } else if (bms_state.substate == BMS_INTERLOCK_CHECKED) {
378                 bms_state.timer = BMS_STATEMACH_VERYLONGTIME_MS;
379                 bms_state.substate = BMS_CHECK_ERROR_FLAGS;
380                 break;
381             } else if (bms_state.substate == BMS_CHECK_ERROR_FLAGS) {
382                 if (BMS_CheckAnyErrorFlagSet() == E_NOT_OK) {
383                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
384                     bms_state.state = BMS_STATEMACH_ERROR;
385                     bms_state.substate = BMS_ENTRY;
386                     break;
387                 } else {
388                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
389                     bms_state.substate = BMS_CHECK_STATE_REQUESTS;
390                     break;
391                 }
392             } else if (bms_state.substate == BMS_CHECK_STATE_REQUESTS) {
393                 if (BMS_CheckCANRequests() == BMS_REQ_ID_NORMAL) {
394                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
395                     bms_state.state = BMS_STATEMACH_PRECHARGE;
396                     bms_state.substate = BMS_ENTRY;
397                     break;
398                 } else if
399                 if (BMS_CheckCANRequests() == BMS_REQ_ID_CHARGE) {
400                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
401                     bms_state.state = BMS_STATEMACH_CHARGE_PRECHARGE;
402                     bms_state.substate = BMS_ENTRY;
403                     break;                                                  Need to add the ENGINE state request here
404                 } else {    Other requests
405 #if LTC_STANDBY_PERIODIC_OPEN_WIRE_CHECK == TRUE                    timestamp is obtained at each entrance of the trigger function.
406                     if (nextOpenWireCheck <= timestamp) {
407                         MEAS_Request_OpenWireCheck();
408                         nextOpenWireCheck = timestamp + LTC_STANDBY_OPEN_WIRE_PERIOD_ms;
409                     }
410 #endif /* LTC_STANDBY_PERIODIC_OPEN_WIRE_CHECK == TRUE */
411                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;       Check open wire in the steady state of Standby
412                     bms_state.substate = BMS_CHECK_ERROR_FLAGS;
413                     break;
414                 }
415             }
416             break;
```

```
417                                 (Normal Precharge)                                  Need to borrow the code from contactor.c here
418         /*****************************PRECHARGE************************************/
419         case BMS_STATEMACH_PRECHARGE:
420             BMS_SAVELASTSTATES();
421
422             if (bms_state.substate == BMS_ENTRY) {
423                 BAL_SetStateRequest(BAL_STATE_NOBALANCING_REQUEST);
424                 DB_ReadBlock(&systemstate, DATA_BLOCK_ID_SYSTEMSTATE);
425                 systemstate.bms_state = BMS_STATEMACH_PRECHARGE;
426                 DB_WriteBlock(&systemstate, DATA_BLOCK_ID_SYSTEMSTATE);
427 #if BUILD_MODULE_ENABLE_CONTACTOR == 1
428                 CONT_SetStateRequest(CONT_STATE_NORMAL_REQUEST);        This is the true state request for contactor control.
429 #endif
430                 bms_state.substate = BMS_CHECK_ERROR_FLAGS;
431                 bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
432                 break;
433             } else if (bms_state.substate == BMS_CHECK_ERROR_FLAGS) {
434                 if (BMS_CheckAnyErrorFlagSet() == E_NOT_OK) {
435                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
436                     bms_state.state = BMS_STATEMACH_ERROR;
437                     bms_state.substate = BMS_ENTRY;
438                     break;
439                 } else {
440                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
441                     bms_state.substate = BMS_CHECK_STATE_REQUESTS;
442                     break;
443                 }
444             } else if (bms_state.substate == BMS_CHECK_STATE_REQUESTS) {
445                 if (BMS_CheckCANRequests() == BMS_REQ_ID_STANDBY) {
446                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
447                     bms_state.state = BMS_STATEMACH_STANDBY;
448                     bms_state.substate = BMS_ENTRY;
449                     break;
450                 } else {
451                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
452 #if BUILD_MODULE_ENABLE_CONTACTOR == 1
453                     bms_state.substate = BMS_CHECK_CONTACTOR_NORMAL_STATE;
454 #else
455                     bms_state.state = BMS_STATEMACH_NORMAL;
456                     bms_state.substate = BMS_ENTRY;
457 #endif
458                     break;
459                 }
460 #if BUILD_MODULE_ENABLE_CONTACTOR == 1
461             } else if (bms_state.substate == BMS_CHECK_CONTACTOR_NORMAL_STATE) {
462                 contstate = CONT_GetState();
463                 if (contstate == CONT_STATEMACH_NORMAL) {
464                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
465                     bms_state.state = BMS_STATEMACH_NORMAL;
466                     bms_state.substate = BMS_ENTRY;
467                     break;
468                 } else if (contstate == CONT_STATEMACH_ERROR) {
```

```
469                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
470                     bms_state.state = BMS_STATEMACH_ERROR;
471                     bms_state.substate = BMS_ENTRY;
472                     break;
473                 } else {
474                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
475                     bms_state.substate = BMS_CHECK_ERROR_FLAGS;
476                 }
477 #endif
478             }
479             break;
480 
481         /*************************NORMAL*************************************/
482         case BMS_STATEMACH_NORMAL:
483             BMS_SAVELASTSTATES();
484 
485             if (bms_state.substate == BMS_ENTRY) {
486 #if LTC_NORMAL_PERIODIC_OPEN_WIRE_CHECK == TRUE
487                 nextOpenWireCheck = timestamp + LTC_NORMAL_OPEN_WIRE_PERIOD_ms;
488 #endif /* LTC_NORMAL_PERIODIC_OPEN_WIRE_CHECK == TRUE */
489                 DB_ReadBlock(&systemstate, DATA_BLOCK_ID_SYSTEMSTATE);
490                 systemstate.bms_state = BMS_STATEMACH_NORMAL;
491                 DB_WriteBlock(&systemstate, DATA_BLOCK_ID_SYSTEMSTATE);
492                 bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
493                 bms_state.substate = BMS_CHECK_ERROR_FLAGS;
494                 break;
495             } else if (bms_state.substate == BMS_CHECK_ERROR_FLAGS) {
496                 if (BMS_CheckAnyErrorFlagSet() == E_NOT_OK) {
497                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
498                     bms_state.state = BMS_STATEMACH_ERROR;
499                     bms_state.substate = BMS_ENTRY;
500                     break;
501                 } else {
502                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
503                     bms_state.substate = BMS_CHECK_STATE_REQUESTS;
504                     break;
505                 }
506             } else if (bms_state.substate == BMS_CHECK_STATE_REQUESTS) {
507                 if (BMS_CheckCANRequests() == BMS_REQ_ID_STANDBY) {
508                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
509                     bms_state.state = BMS_STATEMACH_STANDBY;
510                     bms_state.substate = BMS_ENTRY;
511                     break;
512                 } else {
513 #if LTC_NORMAL_PERIODIC_OPEN_WIRE_CHECK == TRUE
514                     if (nextOpenWireCheck <= timestamp) {
515                         MEAS_Request_OpenWireCheck();
516                         nextOpenWireCheck = timestamp + LTC_NORMAL_OPEN_WIRE_PERIOD_ms;
517                     }
518 #endif /* LTC_NORMAL_PERIODIC_OPEN_WIRE_CHECK == TRUE */
519                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
520                     bms_state.substate = BMS_CHECK_ERROR_FLAGS;
```

If CONT is not in the normal state yet, the BMS state machine will be repeating here.

Only update time during the entry

Currently, we can only go to Standby from Normal.

Check open wire in the steady state of Standby

```
521                     break;
522                 }
523             }
524             break;
525
526             /*************************CHARGE_PRECHARGE*********************************/
527             case BMS_STATEMACH_CHARGE_PRECHARGE:
528                 BMS_SAVELASTSTATES();
529
530                 if (bms_state.substate == BMS_ENTRY) {
531                     BAL_SetStateRequest(BAL_STATE_NOBALANCING_REQUEST);
532                     DB_ReadBlock(&systemstate, DATA_BLOCK_ID_SYSTEMSTATE);
533                     systemstate.bms_state = BMS_STATEMACH_CHARGE_PRECHARGE;
534                     DB_WriteBlock(&systemstate, DATA_BLOCK_ID_SYSTEMSTATE);
535 #if BUILD_MODULE_ENABLE_CONTACTOR == 1
536                     CONT_SetStateRequest(CONT_STATE_CHARGE_REQUEST);
537 #endif
538                     bms_state.substate = BMS_CHECK_ERROR_FLAGS;
539                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
540                     break;
541                 } else if (bms_state.substate == BMS_CHECK_ERROR_FLAGS) {
542                     if (BMS_CheckAnyErrorFlagSet() == E_NOT_OK) {
543                         bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
544                         bms_state.state = BMS_STATEMACH_ERROR;
545                         bms_state.substate = BMS_ENTRY;
546                         break;
547                     } else {
548                         bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
549                         bms_state.substate = BMS_CHECK_STATE_REQUESTS;
550                         break;
551                     }
552                 } else if (bms_state.substate == BMS_CHECK_STATE_REQUESTS) {
553                     if (BMS_CheckCANRequests() == BMS_REQ_ID_STANDBY) {
554                         bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
555                         bms_state.state = BMS_STATEMACH_STANDBY;
556                         bms_state.substate = BMS_ENTRY;
557                         break;
558                     } else {
559                         bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
560 #if BUILD_MODULE_ENABLE_CONTACTOR == 1
561                         bms_state.substate = BMS_CHECK_CONTACTOR_CHARGE_STATE;
562 #else
563                         bms_state.state = BMS_STATEMACH_CHARGE;
564                         bms_state.substate = BMS_ENTRY;
565 #endif
566                         break;
567                     }
568 #if BUILD_MODULE_ENABLE_CONTACTOR == 1
569                 } else if (bms_state.substate == BMS_CHECK_CONTACTOR_CHARGE_STATE) {
570                     contstate = CONT_GetState();
571                     if (contstate == CONT_STATEMACH_CHARGE) {
572                         bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
```

```
573                         bms_state.state = BMS_STATEMACH_CHARGE;
574                         bms_state.substate = BMS_ENTRY;
575                         break;
576                     } else if (contstate == CONT_STATEMACH_ERROR) {
577                         bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
578                         bms_state.state = BMS_STATEMACH_ERROR;
579                         bms_state.substate = BMS_ENTRY;
580                         break;
581                     } else {
582                         bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
583                         bms_state.substate = BMS_CHECK_ERROR_FLAGS;
584                     }
585 #endif
586                 }
587                 break;
588
589
590
591
592             /*****************************CHARGE*************************************/
593             case BMS_STATEMACH_CHARGE:
594                 BMS_SAVELASTSTATES();
595
596                 if (bms_state.substate == BMS_ENTRY) {
597 #if LTC_CHARGE_PERIODIC_OPEN_WIRE_CHECK == TRUE
598                     nextOpenWireCheck = timestamp + LTC_CHARGE_OPEN_WIRE_PERIOD_ms;      Only update time during the entry
599 #endif /* LTC_CHARGE_PERIODIC_OPEN_WIRE_CHECK == TRUE */
600                     DB_ReadBlock(&systemstate, DATA_BLOCK_ID_SYSTEMSTATE);
601                     systemstate.bms_state = BMS_STATEMACH_CHARGE;
602                     DB_WriteBlock(&systemstate, DATA_BLOCK_ID_SYSTEMSTATE);
603                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
604                     bms_state.substate = BMS_CHECK_ERROR_FLAGS;
605                     break;
606                 } else if (bms_state.substate == BMS_CHECK_ERROR_FLAGS) {
607                     if (BMS_CheckAnyErrorFlagSet() == E_NOT_OK) {
608                         bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
609                         bms_state.state = BMS_STATEMACH_ERROR;
610                         bms_state.substate = BMS_ENTRY;
611                         break;
612                     } else {
613                         bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
614                         bms_state.substate = BMS_CHECK_STATE_REQUESTS;
615                         break;
616                     }
617                 } else if (bms_state.substate == BMS_CHECK_STATE_REQUESTS) {
618                     if (BMS_CheckCANRequests() == BMS_REQ_ID_STANDBY) {
619                         bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
620                         bms_state.state = BMS_STATEMACH_STANDBY;
621                         bms_state.substate = BMS_ENTRY;
622                         break;
623                     } else {
624 #if LTC_CHARGE_PERIODIC_OPEN_WIRE_CHECK == TRUE
```

```
625                     if (nextOpenWireCheck <= timestamp) {
626                         MEAS_Request_OpenWireCheck();
627                         nextOpenWireCheck = timestamp + LTC_CHARGE_OPEN_WIRE_PERIOD_ms;
628                     }
629 #endif /* LTC_CHARGE_PERIODIC_OPEN_WIRE_CHECK == TRUE */
630                         bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
631                         bms_state.substate = BMS_CHECK_ERROR_FLAGS;
632                         break;
633                     }
634                 }
635                 break;
636
637         /**************************ERROR*************************************/
638         case BMS_STATEMACH_ERROR:
639             BMS_SAVELASTSTATES();
640
641             if (bms_state.substate == BMS_ENTRY) {
642                 BAL_SetStateRequest(BAL_STATE_NOBALANCING_REQUEST);
643 #if BUILD_MODULE_ENABLE_CONTACTOR == 1
644                 CONT_SetStateRequest(CONT_STATE_ERROR_REQUEST);
645 #endif
646                 bms_state.timer = BMS_STATEMACH_VERYLONGTIME_MS;
647 #if BUILD_MODULE_ENABLE_ILCK == 1
648                 bms_state.substate = BMS_OPEN_INTERLOCK;
649 #else
650                 bms_state.substate = BMS_CHECK_ERROR_FLAGS;
651 #endif
652                 DB_ReadBlock(&systemstate, DATA_BLOCK_ID_SYSTEMSTATE);
653                 systemstate.bms_state = BMS_STATEMACH_ERROR;
654                 DB_WriteBlock(&systemstate, DATA_BLOCK_ID_SYSTEMSTATE);
655                 break;
656 #if BUILD_MODULE_ENABLE_ILCK == 1
657             } else if (bms_state.substate == BMS_OPEN_INTERLOCK) {
658                 ILCK_SetStateRequest(ILCK_STATE_OPEN_REQUEST);
659                 nextOpenWireCheck = timestamp + LTC_ERROR_OPEN_WIRE_PERIOD_ms;
660                 bms_state.timer = BMS_STATEMACH_VERYLONGTIME_MS;
661                 bms_state.substate = BMS_CHECK_ERROR_FLAGS;
662                 break;
663 #endif
664             } else if (bms_state.substate == BMS_CHECK_ERROR_FLAGS) {
665                 if (BMS_CheckAnyErrorFlagSet() == E_NOT_OK) {
666                     /* we stay already in requested state */
667                     if (nextOpenWireCheck <= timestamp) {
668                         /* Perform open-wire check periodically */
669                         MEAS_Request_OpenWireCheck();
670                         nextOpenWireCheck = timestamp + LTC_ERROR_OPEN_WIRE_PERIOD_ms;
671                     }
672                 } else {
673                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
674                     bms_state.substate = BMS_CHECK_STATE_REQUESTS;
675                     break;
676                 }
```

Check open wire in the steady state of Standby

Need to add the states for Engine Precharge and Engine.

```
677             } else if (bms_state.substate == BMS_CHECK_STATE_REQUESTS) {
678                 if (BMS_CheckCANRequests() == BMS_REQ_ID_STANDBY) {
679 #if BUILD_MODULE_ENABLE_ILCK == 1
680                     ILCK_SetStateRequest(ILCK_STATE_CLOSE_REQUEST);
681                     bms_state.substate = BMS_CHECK_INTERLOCK_CLOSE_AFTER_ERROR;
682 #else
683                     bms_state.state = BMS_STATEMACH_STANDBY;
684                     bms_state.substate = BMS_ENTRY;
685 #endif
686                     bms_state.timer = BMS_STATEMACH_MEDIUMTIME_MS;
687                     break;
688                 } else {
689                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
690                     bms_state.substate = BMS_CHECK_ERROR_FLAGS;
691                     break;
692                 }
693 #if BUILD_MODULE_ENABLE_ILCK == 1
694             } else if (bms_state.substate == BMS_CHECK_INTERLOCK_CLOSE_AFTER_ERROR) {
695                 if (ILCK_GetInterlockFeedback() == ILCK_SWITCH_ON) {
696                     /* TODO: check */
697                     BAL_SetStateRequest(BAL_STATE_ALLOWBALANCING_REQUEST);
698                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
699                     bms_state.state = BMS_STATEMACH_STANDBY;
700                     bms_state.substate = BMS_ENTRY;
701                     break;
702                 } else {
703                     bms_state.timer = BMS_STATEMACH_SHORTTIME_MS;
704                     bms_state.substate = BMS_CHECK_ERROR_FLAGS;
705                     break;
706                 }
707 #endif
708             }
709             break;
710         default:
711             break;
712     }  /* end switch (bms_state.state) */
713 
714     bms_state.triggerentry--;
715     bms_state.counter++;
716 }
717 
718 /*================== Static functions ==================================*/
719 /*
720  * @brief   Get latest database entries for static module variables
721  */
722 static void BMS_GetMeasurementValues(void) {
723     DB_ReadBlock(&bms_tab_cellvolt, DATA_BLOCK_ID_CELLVOLTAGE);
724     DB_ReadBlock(&bms_tab_cur_sensor, DATA_BLOCK_ID_CURRENT_SENSOR);
725     DB_ReadBlock(&bms_ow_tab, DATA_BLOCK_ID_OPEN_WIRE);
726     DB_ReadBlock(&bms_tab_minmax, DATA_BLOCK_ID_MINMAX);
727 #if MEAS_TEST_CELL_SOF_LIMITS == TRUE
728     /* Database entry only needed if current is checked against SOF values */
```

```
729         DB_ReadBlock(&bms_tab_sof, DATA_BLOCK_ID_SOF);
730     #endif /* MEAS_TEST_CELL_SOF_LIMITS == TRUE */
731     }
732
733     /*
734      * @brief   Checks the state requests made to the BMS state machine
735      *
736      * @details Checks of the state request in the database and sets this value as return value
737      *
738      * @return  requested state
739      */
740     static uint8_t BMS_CheckCANRequests(void) {
741         uint8_t retVal = BMS_REQ_ID_NOREQ;
742         DATA_BLOCK_STATEREQUEST_s request;
743
744         DB_ReadBlock(&request, DATA_BLOCK_ID_STATEREQUEST);
745
746         if (request.state_request == BMS_REQ_ID_STANDBY) {
747             retVal = BMS_REQ_ID_STANDBY;
748         } else if (request.state_request == BMS_REQ_ID_NORMAL) {
749             retVal = BMS_REQ_ID_NORMAL;
750         }
751
752     #if BS_SEPARATE_POWERLINES == 1
753         else if (request.state_request == BMS_REQ_ID_CHARGE) { /* NOLINT(readability/braces) */
754             retVal = BMS_REQ_ID_CHARGE;
755         }
756     #endif /*  BS_SEPARATE_POWERLINES == 1 */
757                                                                 Need to add the Engine request here.
758         return retVal;
759     }
760
761     /**
762      * @brief   checks the abidance by the safe operating area
763      *
764      * @details verify for cell voltage measurements (U), if minimum and maximum values are out of range
765      */
766     static void BMS_CheckVoltages(void) {
767         uint16_t vol_max = bms_tab_minmax.voltage_max;
768         uint16_t vol_min = bms_tab_minmax.voltage_min;
769         DIAG_RETURNTYPE_e retvalUndervoltMSL = DIAG_HANDLER_RETURN_ERR_OCCURRED;
770
771         if (vol_max >= BC_VOLTMAX_MOL) {
772             /* Over voltage maximum operating limit violated */
773             DIAG_Handler(DIAG_CH_CELLVOLTAGE_OVERVOLTAGE_MOL, DIAG_EVENT_NOK, 0);
774             if (vol_max >= BC_VOLTMAX_RSL) {
775                 /* Over voltage recommended safety limit violated */
776                 DIAG_Handler(DIAG_CH_CELLVOLTAGE_OVERVOLTAGE_RSL, DIAG_EVENT_NOK, 0);
777                 if (vol_max >= BC_VOLTMAX_MSL) {
778                     /* Over voltage maximum safety limit violated */
779                     DIAG_Handler(DIAG_CH_CELLVOLTAGE_OVERVOLTAGE_MSL, DIAG_EVENT_NOK, 0);
780                 }
```

```
781             }
782         }
783         if (vol_max < BC_VOLTMAX_MSL) {
784             /* over voltage maximum safety limit NOT violated */
785             DIAG_Handler(DIAG_CH_CELLVOLTAGE_OVERVOLTAGE_MSL, DIAG_EVENT_OK, 0);
786             if (vol_max < BC_VOLTMAX_RSL) {
787                 /* over voltage recommended safety limit NOT violated */
788                 DIAG_Handler(DIAG_CH_CELLVOLTAGE_OVERVOLTAGE_RSL, DIAG_EVENT_OK, 0);
789                 if (vol_max < BC_VOLTMAX_MOL) {
790                     /* over voltage maximum operating limit NOT violated */
791                     DIAG_Handler(DIAG_CH_CELLVOLTAGE_OVERVOLTAGE_MOL, DIAG_EVENT_OK, 0);
792                 }
793             }
794         }
795 
796         if (vol_min <= BC_VOLTMIN_MOL) {
797             /* Under voltage maximum operating limit violated */
798             DIAG_Handler(DIAG_CH_CELLVOLTAGE_UNDERVOLTAGE_MOL, DIAG_EVENT_NOK, 0);
799             if (vol_min <= BC_VOLTMIN_RSL) {
800                 /* Under voltage recommended safety limit violated */
801                 DIAG_Handler(DIAG_CH_CELLVOLTAGE_UNDERVOLTAGE_RSL, DIAG_EVENT_NOK, 0);
802                 if (vol_min <= BC_VOLTMIN_MSL) {
803                     /* Under voltage maximum safety limit violated */
804                     retvalUndervoltMSL = DIAG_Handler(DIAG_CH_CELLVOLTAGE_UNDERVOLTAGE_MSL, DIAG_EVENT_NOK, 0);
805 
806                     /* If under voltage flag is set and deep-discharge voltage is violated */
807                     if ((retvalUndervoltMSL == DIAG_HANDLER_RETURN_ERR_OCCURRED) &&
808                             (vol_min <= BC_VOLT_DEEP_DISCHARGE)) {
809                         DIAG_Handler(DIAG_CH_DEEP_DISCHARGE_DETECTED, DIAG_EVENT_NOK, 0);
810                     }
811                 }
812             }
813         }
814         if (vol_min > BC_VOLTMIN_MSL) {
815             /* under voltage maximum safety limit NOT violated */
816             DIAG_Handler(DIAG_CH_CELLVOLTAGE_UNDERVOLTAGE_MSL, DIAG_EVENT_OK, 0);
817             if (vol_min > BC_VOLTMIN_RSL) {
818                 /* under voltage recommended safety limit NOT violated */
819                 DIAG_Handler(DIAG_CH_CELLVOLTAGE_UNDERVOLTAGE_RSL, DIAG_EVENT_OK, 0);
820                 if (vol_min > BC_VOLTMIN_MOL) {
821                     /* under voltage maximum operating limit NOT violated */
822                     DIAG_Handler(DIAG_CH_CELLVOLTAGE_UNDERVOLTAGE_MOL, DIAG_EVENT_OK, 0);
823                 }
824             }
825         }
826     }
827 
828 
829     /**
830      * @brief   checks the abidance by the safe operating area
831      *
832      * @details verify for cell temperature measurements (T), if minimum and maximum values are out of range
```

```
 */
static void BMS_CheckTemperatures(void) {
    int16_t temp_min = bms_tab_minmax.temperature_min;
    int16_t temp_max = bms_tab_minmax.temperature_max;

    /* Over temperature check */
    if (BMS_GetBatterySystemState() == BMS_DISCHARGING) {
        /* Discharge */
        if (temp_max >= BC_TEMPMAX_DISCHARGE_MOL) {
            /* Over temperature maximum operating limit violated*/
            DIAG_Handler(DIAG_CH_TEMP_OVERTEMPERATURE_DISCHARGE_MOL, DIAG_EVENT_NOK, 0);
            if (temp_max >= BC_TEMPMAX_DISCHARGE_RSL) {
                /* Over temperature recommended safety limit violated*/
                DIAG_Handler(DIAG_CH_TEMP_OVERTEMPERATURE_DISCHARGE_RSL, DIAG_EVENT_NOK, 0);
                if (temp_max >= BC_TEMPMAX_DISCHARGE_MSL) {
                    /* Over temperature maximum safety limit violated */
                    DIAG_Handler(DIAG_CH_TEMP_OVERTEMPERATURE_DISCHARGE_MSL, DIAG_EVENT_NOK, 0);
                }
            }
        }
        if (temp_max < BC_TEMPMAX_DISCHARGE_MSL) {
            /* over temperature maximum safety limit NOT violated */
            DIAG_Handler(DIAG_CH_TEMP_OVERTEMPERATURE_DISCHARGE_MSL, DIAG_EVENT_OK, 0);
            if (temp_max < BC_TEMPMAX_DISCHARGE_RSL) {
                /* over temperature recommended safety limit NOT violated */
                DIAG_Handler(DIAG_CH_TEMP_OVERTEMPERATURE_DISCHARGE_RSL, DIAG_EVENT_OK, 0);
                if (temp_max < BC_TEMPMAX_DISCHARGE_MOL) {
                    /* over temperature maximum operating limit NOT violated */
                    DIAG_Handler(DIAG_CH_TEMP_OVERTEMPERATURE_DISCHARGE_MOL, DIAG_EVENT_OK, 0);
                }
            }
        }

    } else {
        /* Charge/Relaxation/At rest */
        if (temp_max >= BC_TEMPMAX_CHARGE_MOL) {
            /* Over temperature maximum operating limit violated */
            DIAG_Handler(DIAG_CH_TEMP_OVERTEMPERATURE_CHARGE_MOL, DIAG_EVENT_NOK, 0);
            if (temp_max >= BC_TEMPMAX_CHARGE_RSL) {
                /* Over temperature recommended safety limit violated */
                DIAG_Handler(DIAG_CH_TEMP_OVERTEMPERATURE_CHARGE_RSL, DIAG_EVENT_NOK, 0);
                /* Over temperature maximum safety limit violated */
                if (temp_max >= BC_TEMPMAX_CHARGE_MSL) {
                    DIAG_Handler(DIAG_CH_TEMP_OVERTEMPERATURE_CHARGE_MSL, DIAG_EVENT_NOK, 0);
                }
            }
        }
        if (temp_max < BC_TEMPMAX_CHARGE_MSL) {
            /* over temperature maximum safety limit NOT violated */
            DIAG_Handler(DIAG_CH_TEMP_OVERTEMPERATURE_CHARGE_MSL, DIAG_EVENT_OK, 0);
            if (temp_max < BC_TEMPMAX_CHARGE_RSL) {
                /* over temperature recommended safety limit NOT violated */
```

```
885                 DIAG_Handler(DIAG_CH_TEMP_OVERTEMPERATURE_CHARGE_RSL, DIAG_EVENT_OK, 0);
886                 if (temp_max < BC_TEMPMAX_CHARGE_MOL) {
887                     /* over temperature maximum operating limit NOT violated*/
888                     DIAG_Handler(DIAG_CH_TEMP_OVERTEMPERATURE_CHARGE_MOL, DIAG_EVENT_OK, 0);
889                 }
890             }
891         }
892     }
893 
894     /* Under temperature check */
895     if (BMS_GetBatterySystemState() == BMS_DISCHARGING) {
896         /* Discharge */
897         if (temp_min <= BC_TEMPMIN_DISCHARGE_MOL) {
898             /* Under temperature maximum operating limit violated */
899             DIAG_Handler(DIAG_CH_TEMP_UNDERTEMPERATURE_DISCHARGE_MOL, DIAG_EVENT_NOK, 0);
900             if (temp_min <= BC_TEMPMIN_DISCHARGE_RSL) {
901                 /* Under temperature recommended safety limit violated*/
902                 DIAG_Handler(DIAG_CH_TEMP_UNDERTEMPERATURE_DISCHARGE_RSL, DIAG_EVENT_NOK, 0);
903                 if (temp_min <= BC_TEMPMIN_DISCHARGE_MSL) {
904                     /* Under temperature maximum safety limit violated */
905                     DIAG_Handler(DIAG_CH_TEMP_UNDERTEMPERATURE_DISCHARGE_MSL, DIAG_EVENT_NOK, 0);
906                 }
907             }
908         }
909         if (temp_min > BC_TEMPMIN_DISCHARGE_MSL) {
910             /* under temperature maximum safety limit NOT violated */
911             DIAG_Handler(DIAG_CH_TEMP_UNDERTEMPERATURE_DISCHARGE_MSL, DIAG_EVENT_OK, 0);
912             if (temp_min > BC_TEMPMIN_DISCHARGE_RSL) {
913                 /* under temperature recommended safety limit NOT violated */
914                 DIAG_Handler(DIAG_CH_TEMP_UNDERTEMPERATURE_DISCHARGE_RSL, DIAG_EVENT_OK, 0);
915                 if (temp_min > BC_TEMPMIN_DISCHARGE_MOL) {
916                     /* under temperature maximum operating limit NOT violated*/
917                     DIAG_Handler(DIAG_CH_TEMP_UNDERTEMPERATURE_DISCHARGE_MOL, DIAG_EVENT_OK, 0);
918                 }
919             }
920         }
921     } else {
922         /* Charge/Relaxation/At rest */
923         if (temp_min <= BC_TEMPMIN_CHARGE_MOL) {
924             /* Under temperature maximum operating limit violated */
925             DIAG_Handler(DIAG_CH_TEMP_UNDERTEMPERATURE_CHARGE_MOL, DIAG_EVENT_NOK, 0);
926             if (temp_min <= BC_TEMPMIN_CHARGE_RSL) {
927                 /* Under temperature recommended safety limit violated */
928                 DIAG_Handler(DIAG_CH_TEMP_UNDERTEMPERATURE_CHARGE_RSL, DIAG_EVENT_NOK, 0);
929                 if (temp_min <= BC_TEMPMIN_CHARGE_MSL) {
930                     /* Under temperature maximum safety limit violated */
931                     DIAG_Handler(DIAG_CH_TEMP_UNDERTEMPERATURE_CHARGE_MSL, DIAG_EVENT_NOK, 0);
932                 }
933             }
934         }
935         if (temp_min > BC_TEMPMIN_CHARGE_MSL) {
936             /* under temperature maximum safety limit NOT violated */
```

```
937                 DIAG_Handler(DIAG_CH_TEMP_UNDERTEMPERATURE_CHARGE_MSL, DIAG_EVENT_OK, 0);
938                 if (temp_min > BC_TEMPMIN_CHARGE_RSL) {
939                     /* under temperature recommended safety limit NOT violated */
940                     DIAG_Handler(DIAG_CH_TEMP_UNDERTEMPERATURE_CHARGE_RSL, DIAG_EVENT_OK, 0);
941                     if (temp_min > BC_TEMPMIN_CHARGE_MOL) {
942                         /* under temperature maximum operating limit NOT violated*/
943                         DIAG_Handler(DIAG_CH_TEMP_UNDERTEMPERATURE_CHARGE_MOL, DIAG_EVENT_OK, 0);
944                     }
945                 }
946             }
947         }
948     }
949
950
951
952     /**
953      * @brief   checks the abidance by the safe operating area
954      *
955      * @details verify for cell current measurements (I), if minimum and maximum values are out of range
956      */
957     static void BMS_CheckCurrent(void) {
958         int32_t i_current = bms_tab_cur_sensor.current;
959         uint32_t i_current_abs = 0;
960         BMS_CURRENT_FLOW_STATE_e i_dir = BMS_GetBatterySystemState();
961         if (i_current < 0) {
962             i_current_abs = - i_current;
963         } else {
964             i_current_abs = i_current;
965         }
966
967         /* initialize variables with default values */
968         uint32_t batsys_charge_limit_msl = 0;
969         DIAG_CH_ID_e batsys_charge_limit_diag_msl = DIAG_CH_OVERCURRENT_CHARGE_PL0_MSL;
970         uint32_t batsys_discharge_limit_msl = 0;
971         DIAG_CH_ID_e batsys_discharge_limit_diag_msl = DIAG_CH_OVERCURRENT_DISCHARGE_PL0_MSL;
972         uint32_t batsys_charge_limit_rsl = 0;
973         DIAG_CH_ID_e batsys_charge_limit_diag_rsl = DIAG_CH_OVERCURRENT_CHARGE_PL0_RSL;
974         uint32_t batsys_discharge_limit_rsl = 0;
975         DIAG_CH_ID_e batsys_discharge_limit_diag_rsl = DIAG_CH_OVERCURRENT_DISCHARGE_PL0_RSL;
976         uint32_t batsys_charge_limit_mol = 0;
977         DIAG_CH_ID_e batsys_charge_limit_diag_mol = DIAG_CH_OVERCURRENT_CHARGE_PL0_MOL;
978         uint32_t batsys_discharge_limit_mol = 0;
979         DIAG_CH_ID_e batsys_discharge_limit_diag_mol = DIAG_CH_OVERCURRENT_DISCHARGE_PL0_MOL;
980
981         /* get active power line */
982         CONT_POWER_LINE_e powerline = CONT_GetActivePowerLine();
983
984         /* set limits for batterysystem according to current power line */
985         if (powerline == CONT_POWER_LINE_0) {
986             batsys_charge_limit_msl = BS_CURRENTMAX_CHARGE_PL0_MSL_mA;
987             batsys_charge_limit_diag_msl = DIAG_CH_OVERCURRENT_CHARGE_PL0_MSL;
988             batsys_discharge_limit_msl = BS_CURRENTMAX_DISCHARGE_PL0_MSL_mA;
```

```
989             batsys_discharge_limit_diag_msl = DIAG_CH_OVERCURRENT_DISCHARGE_PL0_MSL;
990
991             batsys_charge_limit_rsl = BS_CURRENTMAX_CHARGE_PL0_RSL_mA;
992             batsys_charge_limit_diag_rsl = DIAG_CH_OVERCURRENT_CHARGE_PL0_RSL;
993             batsys_discharge_limit_rsl = BS_CURRENTMAX_DISCHARGE_PL0_RSL_mA;
994             batsys_discharge_limit_diag_rsl = DIAG_CH_OVERCURRENT_DISCHARGE_PL0_RSL;
995
996             batsys_charge_limit_mol = BS_CURRENTMAX_CHARGE_PL0_MOL_mA;
997             batsys_charge_limit_diag_mol = DIAG_CH_OVERCURRENT_CHARGE_PL0_MOL;
998             batsys_discharge_limit_mol = BS_CURRENTMAX_DISCHARGE_PL0_MOL_mA;
999             batsys_discharge_limit_diag_mol = DIAG_CH_OVERCURRENT_DISCHARGE_PL0_MOL;
1000    #if BS_SEPARATE_POWERLINES == 1
1001        } else if (powerline == CONT_POWER_LINE_1) {
1002            batsys_charge_limit_msl = BS_CURRENTMAX_CHARGE_PL1_MSL_mA;
1003            batsys_charge_limit_diag_msl = DIAG_CH_OVERCURRENT_CHARGE_PL1_MSL;
1004            batsys_discharge_limit_msl = BS_CURRENTMAX_DISCHARGE_PL1_MSL_mA;
1005            batsys_discharge_limit_diag_msl = DIAG_CH_OVERCURRENT_DISCHARGE_PL1_MSL;
1006
1007            batsys_charge_limit_rsl = BS_CURRENTMAX_CHARGE_PL1_RSL_mA;
1008            batsys_charge_limit_diag_rsl = DIAG_CH_OVERCURRENT_CHARGE_PL1_RSL;
1009            batsys_discharge_limit_rsl = BS_CURRENTMAX_DISCHARGE_PL1_RSL_mA;
1010            batsys_discharge_limit_diag_rsl = DIAG_CH_OVERCURRENT_DISCHARGE_PL1_RSL;
1011
1012            batsys_charge_limit_mol = BS_CURRENTMAX_CHARGE_PL1_MOL_mA;
1013            batsys_charge_limit_diag_mol = DIAG_CH_OVERCURRENT_CHARGE_PL1_MOL;
1014            batsys_discharge_limit_mol = BS_CURRENTMAX_DISCHARGE_PL1_MOL_mA;
1015            batsys_discharge_limit_diag_mol = DIAG_CH_OVERCURRENT_DISCHARGE_PL1_MOL;
1016    #endif
1017        } else {
1018            /* this is a configuration error, assume safe default */
1019            batsys_charge_limit_msl = BS_CS_THRESHOLD_NO_CURRENT_mA;
1020            batsys_charge_limit_diag_msl = DIAG_CH_OVERCURRENT_PL_NONE;
1021            batsys_discharge_limit_msl = BS_CS_THRESHOLD_NO_CURRENT_mA;
1022            batsys_discharge_limit_diag_msl = DIAG_CH_OVERCURRENT_PL_NONE;
1023
1024            batsys_charge_limit_rsl = BS_CS_THRESHOLD_NO_CURRENT_mA;
1025            batsys_charge_limit_diag_rsl = DIAG_CH_OVERCURRENT_PL_NONE;
1026            batsys_discharge_limit_rsl = BS_CS_THRESHOLD_NO_CURRENT_mA;
1027            batsys_discharge_limit_diag_rsl = DIAG_CH_OVERCURRENT_PL_NONE;
1028
1029            batsys_charge_limit_mol = BS_CS_THRESHOLD_NO_CURRENT_mA;
1030            batsys_charge_limit_diag_mol = DIAG_CH_OVERCURRENT_PL_NONE;
1031            batsys_discharge_limit_mol = BS_CS_THRESHOLD_NO_CURRENT_mA;
1032            batsys_discharge_limit_diag_mol = DIAG_CH_OVERCURRENT_PL_NONE;
1033        }
1034
1035        /* check limits of battery system */
1036        if (i_dir == BMS_CHARGING) {
1037            /* Charge */
1038            if (i_current_abs >= batsys_charge_limit_mol) {
1039                /* Over current maximum operating limit of batsys violated */
1040                DIAG_Handler(batsys_charge_limit_diag_mol, DIAG_EVENT_NOK, 0);
```

```
1041            if (i_current_abs >= batsys_charge_limit_rsl) {
1042                /* Over current recommended safety limit of batsys violated */
1043                DIAG_Handler(batsys_charge_limit_diag_rsl, DIAG_EVENT_NOK, 0);
1044                if (i_current_abs >= batsys_charge_limit_msl) {
1045                    /* Over current maximum safety limit of batsys violated */
1046                    DIAG_Handler(batsys_charge_limit_diag_msl, DIAG_EVENT_NOK, 0);
1047                }
1048            }
1049        }
1050        if (i_current_abs < batsys_charge_limit_msl) {
1051            /* Over current maximum safety limit of batsys NOT violated */
1052            DIAG_Handler(batsys_charge_limit_diag_msl, DIAG_EVENT_OK, 0);
1053            if (i_current_abs < batsys_charge_limit_rsl) {
1054                /* Over current recommended safety limit of batsys NOT violated */
1055                DIAG_Handler(batsys_charge_limit_diag_rsl, DIAG_EVENT_OK, 0);
1056                if (i_current_abs < batsys_charge_limit_mol) {
1057                    /* Over current maximum operating limit of batsys NOT violated */
1058                    DIAG_Handler(batsys_charge_limit_diag_mol, DIAG_EVENT_OK, 0);
1059                }
1060            }
1061        }
1062    } else if (i_dir == BMS_DISCHARGING) {
1063        /* Discharge */
1064        if (i_current_abs >= batsys_discharge_limit_mol) {
1065            /* Over current maximum operating limit of batsys violated */
1066            DIAG_Handler(batsys_discharge_limit_diag_mol, DIAG_EVENT_NOK, 0);
1067            if (i_current_abs >= batsys_discharge_limit_rsl) {
1068                /* Over current recommended safety limit of batsys violated */
1069                DIAG_Handler(batsys_discharge_limit_diag_rsl, DIAG_EVENT_NOK, 0);
1070                if (i_current_abs >= batsys_discharge_limit_msl) {
1071                    /* Over current maximum safety limit of batsys violated */
1072                    DIAG_Handler(batsys_discharge_limit_diag_msl, DIAG_EVENT_NOK, 0);
1073                }
1074            }
1075        }
1076        if (i_current_abs < batsys_discharge_limit_msl) {
1077            /* Over current maximum safety limit of batsys NOT violated */
1078            DIAG_Handler(batsys_discharge_limit_diag_msl, DIAG_EVENT_OK, 0);
1079            if (i_current_abs < batsys_discharge_limit_rsl) {
1080                /* Over current recommended safety limit of batsys NOT violated */
1081                DIAG_Handler(batsys_discharge_limit_diag_rsl, DIAG_EVENT_OK, 0);
1082                if (i_current_abs < batsys_discharge_limit_mol) {
1083                    /* Over current maximum operating limit of batsys NOT violated */
1084                    DIAG_Handler(batsys_discharge_limit_diag_mol, DIAG_EVENT_OK, 0);
1085                }
1086            }
1087        }
1088    } else {
1089        /* BS_CURRENT_NO_CURRENT -> no violations */
1090        DIAG_Handler(DIAG_CH_OVERCURRENT_PL_NONE, DIAG_EVENT_OK, 0);
1091        DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_PL0_MSL, DIAG_EVENT_OK, 0);
1092        DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_PL0_RSL, DIAG_EVENT_OK, 0);
```

```
1093            DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_PL0_MOL, DIAG_EVENT_OK, 0);
1094            DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_PL1_MSL, DIAG_EVENT_OK, 0);
1095            DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_PL1_RSL, DIAG_EVENT_OK, 0);
1096            DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_PL1_MOL, DIAG_EVENT_OK, 0);
1097            DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_PL0_MSL, DIAG_EVENT_OK, 0);
1098            DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_PL0_RSL, DIAG_EVENT_OK, 0);
1099            DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_PL0_MOL, DIAG_EVENT_OK, 0);
1100            DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_PL1_MSL, DIAG_EVENT_OK, 0);
1101            DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_PL1_RSL, DIAG_EVENT_OK, 0);
1102            DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_PL1_MOL, DIAG_EVENT_OK, 0);
1103        }
1104
1105        /* check limits of cells */
1106    #if MEAS_TEST_CELL_SOF_LIMITS == TRUE
1107        if (i_dir == BMS_CHARGING) {
1108            /* Charge */
1109            if (i_current_abs >= bms_tab_sof.continuous_charge_MOL) {
1110                /* Over current maximum operating limit violated */
1111                DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_CELL_MOL, DIAG_EVENT_NOK, 0);
1112                if (i_current_abs >= bms_tab_sof.continuous_charge_RSL) {
1113                    /* Over current recommended safety limit violated */
1114                    DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_CELL_RSL, DIAG_EVENT_NOK, 0);
1115                    if (i_current_abs >= bms_tab_sof.continuous_charge_MSL) {
1116                        /* Over current maximum safety limit violated */
1117                        DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_CELL_MSL, DIAG_EVENT_NOK, 0);
1118                    }
1119                }
1120            }
1121            if (i_current_abs < bms_tab_sof.continuous_charge_MSL) {
1122                /* over current maximum safety limit NOT violated */
1123                DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_CELL_MSL, DIAG_EVENT_OK, 0);
1124                if (i_current_abs < bms_tab_sof.continuous_charge_RSL) {
1125                    /* over current recommended safety limit NOT violated */
1126                    DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_CELL_RSL, DIAG_EVENT_OK, 0);
1127                    if (i_current_abs < bms_tab_sof.continuous_charge_MOL) {
1128                        /* over current maximum operating limit NOT violated */
1129                        DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_CELL_MOL, DIAG_EVENT_OK, 0);
1130                    }
1131                }
1132            }
1133        } else if (i_dir == BMS_DISCHARGING) {
1134            /* Discharge */
1135            if (i_current_abs >= bms_tab_sof.continuous_discharge_MOL) {
1136                /* Over current maximum operating limit violated */
1137                DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_CELL_MOL, DIAG_EVENT_NOK, 0);
1138                if (i_current_abs >= bms_tab_sof.continuous_discharge_RSL) {
1139                    /* Over current recommended safety limit violated */
1140                    DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_CELL_RSL, DIAG_EVENT_NOK, 0);
1141                    if (i_current_abs >= bms_tab_sof.continuous_discharge_MSL) {
1142                        /* Over current error */
1143                        DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_CELL_MSL, DIAG_EVENT_NOK, 0);
1144                    }
```

```
1145                }
1146            }
1147
1148            if (i_current_abs < bms_tab_sof.continuous_discharge_MSL) {
1149                /* over current maximum safety limit NOT violated */
1150                DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_CELL_MSL, DIAG_EVENT_OK, 0);
1151                if (i_current_abs < bms_tab_sof.continuous_discharge_RSL) {
1152                    /* over current recommended safety limit NOT violated */
1153                    DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_CELL_RSL, DIAG_EVENT_OK, 0);
1154                    if (i_current_abs < bms_tab_sof.continuous_discharge_MOL) {
1155                        /* over current maximum operating limit NOT violated */
1156                        DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_CELL_MOL, DIAG_EVENT_OK, 0);
1157                    }
1158                }
1159            }
1160        } else {
1161            /* BS_CURRENT_NO_CURRENT -> no check needed if no current is floating */
1162        }
1163    #else /* MEAS_TEST_CELL_SOF_LIMITS == FALSE */
1164        if (i_dir == BMS_CHARGING) {
1165            /* Charge */
1166            if (i_current_abs >= BC_CURRENTMAX_CHARGE_MOL) {
1167                /* Over current maximum operating limit of cells violated */
1168                DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_CELL_MOL, DIAG_EVENT_NOK, 0);
1169                if (i_current_abs >= BC_CURRENTMAX_CHARGE_RSL) {
1170                    /* Over current recommended safety limit of cells violated */
1171                    DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_CELL_RSL, DIAG_EVENT_NOK, 0);
1172                    if (i_current_abs >= BC_CURRENTMAX_CHARGE_MSL) {
1173                        /* Over current maximum safety limit of cells violated */
1174                        DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_CELL_MSL, DIAG_EVENT_NOK, 0);
1175                    }
1176                }
1177            }
1178            if (i_current_abs < BC_CURRENTMAX_CHARGE_MSL) {
1179                /* Over current maximum safety limit of cells NOT violated */
1180                DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_CELL_MSL, DIAG_EVENT_OK, 0);
1181                if (i_current_abs < BC_CURRENTMAX_CHARGE_RSL) {
1182                    /* Over current recommended safety limit of cells NOT violated */
1183                    DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_CELL_RSL, DIAG_EVENT_OK, 0);
1184                    if (i_current_abs < BC_CURRENTMAX_CHARGE_MOL) {
1185                        /* Over current maximum operating limit of cells NOT violated */
1186                        DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_CELL_MOL, DIAG_EVENT_OK, 0);
1187                    }
1188                }
1189            }
1190        } else if (i_dir == BMS_DISCHARGING) {
1191            /* Discharge */
1192            if (i_current_abs >= BC_CURRENTMAX_DISCHARGE_MOL) {
1193                /* Over current maximum operating limit of cells violated */
1194                DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_CELL_MOL, DIAG_EVENT_NOK, 0);
1195                if (i_current_abs >= BC_CURRENTMAX_DISCHARGE_RSL) {
1196                    /* Over current recommended safety limit of cells violated */
```

```
1197                    DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_CELL_RSL, DIAG_EVENT_NOK, 0);
1198                    if (i_current_abs >= BC_CURRENTMAX_DISCHARGE_MSL) {
1199                        /* Over current maximum safety limit of cells violated */
1200                        DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_CELL_MSL, DIAG_EVENT_NOK, 0);
1201                    }
1202                }
1203            }
1204            if (i_current_abs < BC_CURRENTMAX_DISCHARGE_MSL) {
1205                /* Over current maximum safety limit of cells NOT violated */
1206                DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_CELL_MSL, DIAG_EVENT_OK, 0);
1207                if (i_current_abs < BC_CURRENTMAX_DISCHARGE_RSL) {
1208                    /* Over current recommended safety limit of cells NOT violated */
1209                    DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_CELL_RSL, DIAG_EVENT_OK, 0);
1210                    if (i_current_abs < BC_CURRENTMAX_DISCHARGE_MOL) {
1211                        /* Over current maximum operating limit of cells NOT violated */
1212                        DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_CELL_MOL, DIAG_EVENT_OK, 0);
1213                    }
1214                }
1215            }
1216        } else {
1217            /* BS_CURRENT_NO_CURRENT -> no violations */
1218            DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_CELL_MSL, DIAG_EVENT_OK, 0);
1219            DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_CELL_RSL, DIAG_EVENT_OK, 0);
1220            DIAG_Handler(DIAG_CH_OVERCURRENT_CHARGE_CELL_MOL, DIAG_EVENT_OK, 0);
1221            DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_CELL_MSL, DIAG_EVENT_OK, 0);
1222            DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_CELL_RSL, DIAG_EVENT_OK, 0);
1223            DIAG_Handler(DIAG_CH_OVERCURRENT_DISCHARGE_CELL_MOL, DIAG_EVENT_OK, 0);
1224        }
1225    #endif /* MEAS_TEST_CELL_SOF_LIMITS == TRUE */
1226    }
1227
1228    /**
1229     * @brief   FOR FUTURE COMPATIBILITY; DUMMY FUNCTION; DO NOT USE
1230     *
1231     * @details FOR FUTURE COMPATIBILITY; DUMMY FUNCTION; DO NOT USE
1232     */
1233    static void BMS_CheckSlaveTemperatures(void) {
1234        /* TODO: to be implemented */
1235    }
1236
1237
1238    /**
1239     * @brief   Check for any open voltage sense wire
1240     */
1241    static void BMS_CheckOpenSenseWire(void) {
1242        uint8_t openWireDetected = 0;
1243
1244        /* Iterate over all modules */
1245        for (uint8_t m = 0; m < BS_NR_OF_MODULES; m++) {
1246            /* Iterate over all voltage sense wires: cells per module + 1 */
1247            for (uint8_t wire = 0; wire < (BS_NR_OF_BAT_CELLS_PER_MODULE + 1); wire++) {
1248                /* open wire detected */
```

```
1249                if (bms_ow_tab.openwire[wire + m*(BS_NR_OF_BAT_CELLS_PER_MODULE + 1) == 1]) {
1250                    openWireDetected++;
1251
1252                    /* Add additional error handling here */
1253                }
1254            }
1255        }
1256        /* Set error if open wire detected */
1257        if (openWireDetected == 0u) {
1258            DIAG_Handler(DIAG_CH_OPEN_WIRE, DIAG_EVENT_OK, 0);
1259        } else {
1260            DIAG_Handler(DIAG_CH_OPEN_WIRE, DIAG_EVENT_NOK, 0);
1261        }
1262    }
1263
1264
1265    /**
1266     * @brief   Checks the error flags
1267     *
1268     * @details Checks all the error flags from the database and returns an error if at least one is set.
1269     *
1270     * @return  E_OK if no error flag is set, otherwise E_NOT_OK
1271     */
1272    static STD_RETURN_TYPE_e BMS_CheckAnyErrorFlagSet(void) {
1273        STD_RETURN_TYPE_e retVal = E_OK;  /* is set to E_NOT_OK if error detected */
1274        DATA_BLOCK_ERRORSTATE_s error_flags;
1275        DATA_BLOCK_MSL_FLAG_s msl_flags;
1276
1277        DB_ReadBlock(&error_flags, DATA_BLOCK_ID_ERRORSTATE);
1278        DB_ReadBlock(&msl_flags, DATA_BLOCK_ID_MSL);
1279
1280        /* Check maximum safety limit flags */
1281        if (msl_flags.over_current_charge_cell    == 1 ||
1282            msl_flags.over_current_charge_pl0     == 1 ||
1283            msl_flags.over_current_charge_pl1     == 1 ||
1284            msl_flags.over_current_discharge_cell == 1 ||
1285            msl_flags.over_current_discharge_pl0  == 1 ||
1286            msl_flags.over_current_discharge_pl1  == 1 ||
1287            msl_flags.over_voltage                == 1 ||
1288            msl_flags.under_voltage               == 1 ||
1289            msl_flags.over_temperature_charge     == 1 ||
1290            msl_flags.over_temperature_discharge  == 1 ||
1291            msl_flags.under_temperature_charge    == 1 ||
1292            msl_flags.under_temperature_discharge == 1) {
1293            /* error detected */
1294            retVal = E_NOT_OK;
1295        }
1296
1297        /* Check system error flags */
1298        if (error_flags.currentOnOpenPowerline    == 1 ||
1299            error_flags.deepDischargeDetected     == 1 ||
1300            error_flags.main_plus                 == 1 ||
```

```
1301            error_flags.main_minus                == 1 ||
1302            error_flags.precharge                 == 1 ||
1303            error_flags.charge_main_plus          == 1 ||
1304            error_flags.charge_main_minus         == 1 ||
1305            error_flags.charge_precharge          == 1 ||
1306            error_flags.fuse_state_normal         == 1 ||
1307            error_flags.fuse_state_charge         == 1 ||
1308            error_flags.interlock                 == 1 ||
1309            error_flags.crc_error                 == 1 ||
1310            error_flags.mux_error                 == 1 ||
1311            error_flags.spi_error                 == 1 ||
1312            error_flags.ltc_config_error          == 1 ||
1313            error_flags.currentsensorresponding   == 1 ||
1314            error_flags.open_wire                 == 1 ||
1315    #if BMS_OPEN_CONTACTORS_ON_INSULATION_ERROR == TRUE
1316            error_flags.insulation_error          == 1 ||
1317    #endif /* BMS_OPEN_CONTACTORS_ON_INSULATION_ERROR */
1318            error_flags.can_timing_cc             == 1 ||
1319            error_flags.can_timing                == 1) {
1320            /* error detected */
1321            retVal = E_NOT_OK;
1322        }
1323
1324        return retVal;
1325    }
1326
1327
1328    /**
1329     * @brief   Updates battery system state variable depending on measured/recent
1330     *          current values
1331     *
1332     * @param   curSensor   recent measured values from current sensor
1333     */
1334    static void BMS_UpdateBatsysState(DATA_BLOCK_CURRENT_SENSOR_s *curSensor) {
1335        if (POSITIVE_DISCHARGE_CURRENT == TRUE) {
1336            /* Positive current values equal a discharge of the battery system */
1337            if (curSensor->current >= BS_REST_CURRENT_mA) {
1338                bms_state.currentFlowState = BMS_DISCHARGING;
1339                bms_state.restTimer_ms = BS_RELAXATION_PERIOD_MS;
1340            } else if (curSensor->current <= -BS_REST_CURRENT_mA) {
1341                bms_state.currentFlowState = BMS_CHARGING;
1342                bms_state.restTimer_ms = BS_RELAXATION_PERIOD_MS;
1343            } else {
1344                /* Current below rest current: either battery system is at rest
1345                 * or the relaxation process is still ongoing */
1346                if (bms_state.restTimer_ms == 0) {
1347                    /* Rest timer elapsed -> battery system at rest */
1348                    bms_state.currentFlowState = BMS_AT_REST;
1349                } else {
1350                    bms_state.restTimer_ms--;
1351                    bms_state.currentFlowState = BMS_RELAXATION;
1352                }
```

```
1353            }
1354        } else {
1355            /* Negative current values equal a discharge of the battery system */
1356            if (curSensor->current <= -BS_REST_CURRENT_mA) {
1357                bms_state.currentFlowState = BMS_DISCHARGING;
1358                bms_state.restTimer_ms = BS_RELAXATION_PERIOD_MS;
1359            } else if (curSensor->current >= BS_REST_CURRENT_mA) {
1360                bms_state.currentFlowState = BMS_CHARGING;
1361                bms_state.restTimer_ms = BS_RELAXATION_PERIOD_MS;
1362            } else {
1363                /* Current below rest current: either battery system is at rest
1364                 * or the relaxation process is still ongoing */
1365                if (bms_state.restTimer_ms == 0) {
1366                    /* Rest timer elapsed -> battery system at rest */
1367                    bms_state.currentFlowState = BMS_AT_REST;
1368                } else {
1369                    bms_state.restTimer_ms--;
1370                    bms_state.currentFlowState = BMS_RELAXATION;
1371                }
1372            }
1373        }
1374    }
1375
1376
1377    BMS_CURRENT_FLOW_STATE_e BMS_GetBatterySystemState(void) {
1378        return bms_state.currentFlowState;
1379    }
1380
```