```
/**
 *
 * @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    diag.c
 * @author  foxBMS Team
 * @date    09.11.2015 (date of creation)
 * @ingroup ENGINE
 * @prefix  DIAG
 *
 * @brief   Diagnosis driver implementation
 *
 * This diagnose module is responsible for error handling and reporting.
 * Reported errors are logged into the global database and can be reviewed
```

```
 * on user request.
 */

/*================== Includes ============================================*/
#include "diag.h"
#if BUILD_MODULE_ENABLE_CONTACTOR == 1
#include "contactor.h"
#endif
#include "com.h"
#include "os.h"
#if BUILD_MODULE_ENABLE_NVRAM == 1
#include "nvramhandler.h"
#endif
#include "rtc.h"
#include "stdio.h"

extern int _write(int fd, char *ptr, int len);

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

/*================== Constant and Variable Definitions ===================*/
static DIAG_s diag;
static DIAG_DEV_s  *diag_devptr;
static uint32_t diagsysmonTimestamp = 0;
static uint8_t diag_locked = 0;

DIAG_SYSMON_NOTIFICATION_s diag_sysmon[DIAG_SYSMON_MODULE_ID_MAX];
DIAG_SYSMON_NOTIFICATION_s diag_sysmon_last[DIAG_SYSMON_MODULE_ID_MAX];

uint32_t diag_sysmon_cnt[DIAG_SYSMON_MODULE_ID_MAX];

DIAG_ERROR_ENTRY_s MEM_BKP_SRAM diag_memory[DIAG_FAIL_ENTRY_LENGTH];
DIAG_ERROR_ENTRY_s MEM_BKP_SRAM *diag_entry_wrptr;
DIAG_ERROR_ENTRY_s MEM_BKP_SRAM *diag_entry_rdptr;

DIAG_CONTACTOR_ERROR_ENTRY_s MEM_BKP_SRAM diagContactorErrorMemory[DIAG_FAIL_ENTRY_CONTACTOR_LENGTH];
DIAG_CONTACTOR_ERROR_ENTRY_s MEM_BKP_SRAM *diagContactorError_entry_wrptr;
DIAG_CONTACTOR_ERROR_ENTRY_s MEM_BKP_SRAM *diagContactorError_entry_rdptr;

DIAG_FAILURECODE_s diag_fc;

/*================== Function Prototypes =================================*/
static void DIAG_Reset(void);
static uint8_t DIAG_EntryWrite(uint8_t eventID, DIAG_EVENT_e event, uint32_t item_nr);

/*================== Function Implementations ============================*/

/**
 * @brief   DIAG_Reset resets/initializes all needed structures/buffers.
 *
 * This function gets called during initialization of the diagnose module.
 * It clears memory and counters used by diag later on.
```

```
 */

static void DIAG_Reset(void) {
    uint32_t i;
    uint32_t *u32ptr = (uint32_t*)(&diag_memory[0]);

    diag_locked = 1;

    /* Delete memory */
    for (i = 0; i < (sizeof(diag_memory))/4; i++) {
        *u32ptr++ = 0;
    }

    /* Reset counter */
    for (i = 0; i < sizeof(diag.entry_cnt); i++) {
        diag.entry_cnt[i] = 0;
    }

    /* Set pointer to beginning of buffer */
    diag_entry_wrptr = diag_entry_rdptr = &diag_memory[0];
    diag.errcnttotal = 0;

    /* Set pointer to beginning of buffer */
    u32ptr = (uint32_t*)(&diagContactorErrorMemory[0]);

    /* Delete memory */
    for (i = 0; i < (sizeof(diagContactorErrorMemory))/4; i++) {
        *u32ptr++ = 0;
    }

    /* Set pointer to beginning of buffer */
    diagContactorError_entry_wrptr = diagContactorError_entry_rdptr = &diagContactorErrorMemory[0];
    diag_locked = 0;
}


STD_RETURN_TYPE_e DIAG_Init(DIAG_DEV_s *diag_dev_pointer, STD_RETURN_TYPE_e bkpramValid) {
    STD_RETURN_TYPE_e retval = E_OK;
    uint8_t c = 0;
    uint8_t id_nr = DIAG_ID_MAX;
    uint32_t tmperr_Check[(DIAG_ID_MAX+31)/32];

    diag_devptr = diag_dev_pointer;

    diag.state = DIAG_STATE_UNINITIALIZED;
    uint16_t checkfail = 0;

    if ((diag_entry_rdptr < &diag_memory[0]) || (diag_entry_rdptr >= &diag_memory[DIAG_FAIL_ENTRY_LENGTH])) {
        checkfail |= 0x01;
    }

    if ((diag_entry_wrptr < &diag_memory[0]) || (diag_entry_wrptr >= &diag_memory[DIAG_FAIL_ENTRY_LENGTH])) {
```

```
157             checkfail |= 0x02;
158         }
159 
160         if (bkpramValid == E_NOT_OK) {
161             checkfail |= 0x04;
162         }
163 
164         if ((diagContactorError_entry_rdptr < &diagContactorErrorMemory[0]) ||
165                 (diagContactorError_entry_rdptr >= &diagContactorErrorMemory[DIAG_FAIL_ENTRY_CONTACTOR_LENGTH])) {
166             checkfail |= 0x08;
167         }
168 
169         if ((diagContactorError_entry_wrptr < &diagContactorErrorMemory[0]) ||
170                 (diagContactorError_entry_wrptr >= &diagContactorErrorMemory[DIAG_FAIL_ENTRY_CONTACTOR_LENGTH])) {
171             checkfail |= 0x10;
172         }
173 
174 
175         if (checkfail) {
176             DIAG_Reset();
177         }
178 
179         /* Fill lookup table id2ch */
180         for (c = 0; c < diag_dev_pointer->nr_of_ch; c++) {
181             id_nr = diag_dev_pointer->ch_cfg[c].id;
182             if (id_nr < DIAG_ID_MAX) {
183                 diag.id2ch[id_nr] = c;      /* e.g. diag.id2ch[DIAG_ID_90] = configured channel index */
184             } else {
185                 /* Configuration error -> set retval to E_NOT_OK */
186                 checkfail |= 0x20;
187                 retval = E_NOT_OK;
188             }
189         }
190 
191         for (int i = 0; i < (DIAG_ID_MAX+31)/32; i++) {
192             tmperr_Check[i] = 0;
193         }
194 
195         /* Fill enable array err_enableflag */
196         for (int i = 0; i < diag_dev_pointer->nr_of_ch; i++) {
197             if (diag_dev_pointer->ch_cfg[i].state == DIAG_DISABLED) {
198                 /* Disable diagnosis entry */
199                 tmperr_Check[diag_dev_pointer->ch_cfg[i].id/32] |= 1 << (diag_dev_pointer->ch_cfg[i].id % 32);
200             }
201         }
202 
203         /* take over configured error enable masks*/
204         for (c = 0; c < (DIAG_ID_MAX+31)/32; c++) {
205             diag.err_enableflag[c] = ~tmperr_Check[c];
206         }
207 
208         diag.state = DIAG_STATE_INITIALIZED;
```

```
209
210         if (checkfail) {
211             /* make first entry after DIAG_Reset() */
212             (void)(DIAG_Handler(DIAG_CH_BKPDIAG_FAILURE, DIAG_EVENT_NOK, checkfail));
213         }
214         return retval;
215     }
216
217
218     void DIAG_PrintErrors(void) {
219         /* FIXME if read once, rdptr is on writeptr, therefore in the next call the errors aren't
220          * printed again. Maybe tmp save rdptr and set again at end of function. But when is the
221          * diag memory cleared then? */
222
223         if (diag_entry_rdptr == diag_entry_wrptr) {
224             printf("no new entries in DIAG\r\n");
225         } else {
226             printf("DIAG error entries:\r\n");
227             printf("Date and Time:      Error Code/Item   Status     Description\r\n");
228         }
229         uint8_t c = 0;
230         while (diag_entry_rdptr != diag_entry_wrptr && c < 7) {
231             if (diag_entry_rdptr >= &diag_memory[DIAG_FAIL_ENTRY_LENGTH]) {
232                 diag_entry_rdptr = &diag_memory[0];
233             }
234
235
236             printf("%02d.%02d.20%02d - %02d:%02d:%02d     ", diag_entry_rdptr->DD, diag_entry_rdptr->MM,
                diag_entry_rdptr->YY,
237               diag_entry_rdptr->hh, diag_entry_rdptr->mm, diag_entry_rdptr->ss);
238
239             printf("%02d / 0x%08x      ", diag_entry_rdptr->event_id, diag_entry_rdptr->event_id);
240
241             if (diag_entry_rdptr->event == DIAG_EVENT_OK)
242                 printf("cleared     ");
243             else if (diag_entry_rdptr->event == DIAG_EVENT_NOK)
244                 printf("occurred    ");
245             else
246                 printf("reset       ");
247
248
249             printf("%s\r\n", diag_devptr->ch_cfg[diag.id2ch[diag_entry_rdptr->event_id]].description);
250
251             diag_entry_rdptr++;
252             c++;
253         }
254         /* More entries in diag buffer */
255         if (diag_entry_rdptr != diag_entry_wrptr)
256             printf("Please repeat command. Additional error entries in DIAG buffer available!\r\n");
257     }
258
259     #if BUILD_MODULE_ENABLE_CONTACTOR == 1
```

```
260  void DIAG_PrintContactorInfo(void) {
261      /* FIXME if read once, rdptr is on writeptr, therefore in the next call the errors aren't
262       * printed again. Maybe tmp save rdptr and set again at end of function. But when is the
263       * diag memory cleared then? */
264
265      DIAG_CONTACTOR_s diagContactor;
266
267      NVM_Get_contactorcnt(&diagContactor);
268
269      printf("Contactor switching entries:");
270      printf("\r\n");
271
272      for (uint8_t i = 0; i < BS_NR_OF_CONTACTORS; i++) {
273          printf("Contactor %02d\r\n", i);
274          printf("Opening switches: %04x\r\n", diagContactor.cont_switch_opened[i]);
275          printf("Closing switches: %04x\r\n", diagContactor.cont_switch_closed[i]);
276          printf("Opening switches hard at current: %04x\r\n", diagContactor.cont_switch_opened_hard_at_current[i]);
277          printf("\r\n\n");
278          printf("\r\n");
279      }
280
281      printf("\r\n");
282      if (diagContactorError_entry_rdptr == diagContactorError_entry_wrptr) {
283          printf("no entries in Contactor");
284          printf("\r\n");
285      } else {
286          printf("Contactor error entries:");
287          printf("\r\n");
288          printf("Date and Time       Contactor     Opening current");
289          printf("\r\n");
290      }
291
292
293
294      while (diagContactorError_entry_rdptr != diagContactorError_entry_wrptr) {
295          if (diagContactorError_entry_rdptr >= &diagContactorErrorMemory[DIAG_FAIL_ENTRY_CONTACTOR_LENGTH]) {
296              diagContactorError_entry_rdptr = &diagContactorErrorMemory[0];
297          }
298          printf("%02d.%02d.%02d - %02d:%02d:%02d     ", diagContactorError_entry_rdptr->DD,
            diagContactorError_entry_rdptr->MM,
299           diagContactorError_entry_rdptr->YY, diagContactorError_entry_rdptr->hh, diagContactorError_entry_rdptr->mm,
300           diagContactorError_entry_rdptr->ss);
301
302          printf("%02d         ", diagContactorError_entry_rdptr->contactor);
303          printf("%fmA\r\n", (double)diagContactorError_entry_rdptr->openingCurrent);
304
305          diagContactorError_entry_rdptr++;
306      }
307
308      /* Reset counter for open errors */
309      diagContactor.errcntreported = 0;
310  }
```

```
311  #endif
312
313  /**
314   * @brief DIAG_EntryWrite adds an error entry.
315   *
316   * This function adds an entry to the error buffer.
317   * It provides some functionality to prevent duplicates from being logged.
318   * Multiple occurring error doesn't get logged anymore after they reached a
319   * pre-defined error count.
320   *
321   * @param  eventID:   ID of entry
322   * @param  event:     OK, NOK or RESET
323   * @param  item_nr:   item number of event
324   *
325   * @return 0xFF if event is logged, otherwise 0
326   */
327  static uint8_t DIAG_EntryWrite(uint8_t eventID, DIAG_EVENT_e event, uint32_t item_nr) {
328      uint8_t ret_val = 0;
329      uint8_t c;
330      RTC_Time_s currTime;
331      RTC_Date_s currDate;
332
333      if (diag_locked) {
334          return ret_val;    /* only locked when clearing the diagnosis memory */
335      }
336
337      if (diag.entry_event[eventID] == event) {
338          /* same event of same error type already recorded before -> ignore until event toggles */
339          return ret_val;
340      }
341      if ((diag.entry_event[eventID] == DIAG_EVENT_OK) && (event ==  DIAG_EVENT_RESET)) {
342          /* do record DIAG_EVENT_RESET-event only if last event was an error (re-initialization) */
343          /* meaning: DIAG_EVENT_RESET-event at first time call or after DIAG_EVENT_OK-event will not be recorded */
344          return ret_val;
345      }
346
347      if (++diag.entry_cnt[eventID] > DIAG_MAX_ENTRIES_OF_ERROR) {
348          /* this type of error has been recorded too many times -> ignore to avoid filling buffer with same
             failurecodes */
349          diag.entry_cnt[eventID] = DIAG_MAX_ENTRIES_OF_ERROR;
350          return ret_val;
351      }
352
353      if (diag_entry_wrptr >= &diag_memory[DIAG_FAIL_ENTRY_LENGTH]) {
354          diag_entry_wrptr = &diag_memory[0];
355      }
356
357      /* now record failurecode */
358      ret_val = 0xFF;
359      RTC_getTime(&currTime);
360      RTC_getDate(&currDate);
361
```

```
362         diag_entry_wrptr->YY = currDate.Year;
363         diag_entry_wrptr->MM = currDate.Month;
364         diag_entry_wrptr->DD = currDate.Date;
365         diag_entry_wrptr->hh = currTime.Hours;
366         diag_entry_wrptr->mm = currTime.Minutes;
367         diag_entry_wrptr->ss = currTime.Seconds;
368
369         diag_entry_wrptr->event_id = eventID;        /* Error Code 0... 4x32-1 */
370         diag_entry_wrptr->item    = item_nr;         /*  */
371         diag_entry_wrptr->event   = (uint8_t)event;  /* DIAG_EVENT_OK, DIAG_EVENT_NOK, DIAG_EVENT_RESET */
372
373         diag_entry_wrptr->Val0 = diag_fc.Val0;
374         diag_entry_wrptr->Val1 = diag_fc.Val1;
375         diag_entry_wrptr->Val2 = diag_fc.Val2;
376         diag_entry_wrptr->Val3 = diag_fc.Val3;
377         ++diag_entry_wrptr;
378
379         ++diag.errcntreported;          /* counts of (new) diagnosis entry records which is still not been read by
    external Tool */
380                                         /* which will reset this value to 0 after having read all new entries which means
                                            <acknowledged by user> */
381         ++diag.errcnttotal;             /* total counts of diagnosis entry records */
382
383         diag.entry_event[eventID] = event;
384         c = (uint8_t) diag.errcntreported;
385
386         fprintf(stderr, "New Error entry! (%03d): Error Code/Item %03d/0x%08x ", c, eventID, (unsigned int)item_nr);
387
388         fprintf(stderr, "%s", diag_devptr->ch_cfg[diag.id2ch[eventID]].description);
389
390         if (event == DIAG_EVENT_OK) {
391             fprintf(stderr, " cleared\r\n");
392         } else if (event == DIAG_EVENT_NOK) {
393             fprintf(stderr, " occured\r\n");
394         } else {
395             /* DIAG_EVENT_RESET */
396             fprintf(stderr, " reset\r\n");
397         }
398
399         return ret_val;
400     }
401
402     DIAG_RETURNTYPE_e DIAG_Handler(DIAG_CH_ID_e diag_ch_id, DIAG_EVENT_e event, uint32_t item_nr) {
403         uint32_t ret_val = DIAG_HANDLER_RETURN_UNKNOWN;
404         uint32_t *u32ptr_errCodemsk, *u32ptr_warnCodemsk;
405         uint16_t  *u16ptr_threshcounter;
406         uint16_t cfg_threshold;
407         uint16_t err_enable_idx;
408         uint32_t err_enable_bitmask;
409
410         DIAG_TYPE_RECORDING_e recordingenabled;
411
```

```
412     if (diag.state == DIAG_STATE_UNINITIALIZED) {
413         return (DIAG_HANDLER_RETURN_NOT_READY);
414     }
415
416     if (diag_ch_id >= DIAG_ID_MAX) {
417         return (DIAG_HANDLER_RETURN_WRONG_ID);
418     }
419
420     if ((diag_ch_id == DIAG_CH_CONTACTOR_DAMAGED) || (diag_ch_id == DIAG_CH_CONTACTOR_OPENING) ||
421             (diag_ch_id == DIAG_CH_CONTACTOR_CLOSING)) {
422         return (DIAG_HANDLER_INVALID_TYPE);
423     }
424     err_enable_idx      = diag_ch_id/32;        /* array index of diag.err_enableflag[..] */
425     err_enable_bitmask  = 1 << (diag_ch_id%32);   /* bit number (mask) of diag.err_enableflag[idx] */
426
427
428     u32ptr_errCodemsk   = &diag.errflag[err_enable_idx];
429     u32ptr_warnCodemsk  = &diag.warnflag[err_enable_idx];
430     u16ptr_threshcounter = &diag.occurrence_cnt[diag_ch_id];
431     cfg_threshold       = diag_devptr->ch_cfg[diag.id2ch[diag_ch_id]].thresholds;
432     recordingenabled    = diag_devptr->ch_cfg[diag.id2ch[diag_ch_id]].enablerecording;
433
434     if (event == DIAG_EVENT_OK) {
435         if (diag.err_enableflag[err_enable_idx] & err_enable_bitmask) {
436             /* if (((*u16ptr_threshcounter) == 0) && (*u32ptr_errCodemsk == 0)) */
437             if (((*u16ptr_threshcounter) == 0)) {
438                 /* everything ok, nothing to be handled */
439             } else if ((*u16ptr_threshcounter) > 1) {
440                 (*u16ptr_threshcounter)--;   /* Error did not occur, decrement Error-Counter */
441             } else if ((*u16ptr_threshcounter) == 1) {
442                 /* else if ((*u16ptr_threshcounter) <= 1) */
443                 /* Error did not occur, now decrement to zero and clear Error- or Warning-Flag and make recording if
                    enabled */
444                 *u32ptr_errCodemsk &= ~err_enable_bitmask;      /* ERROR:   clear corresponding bit in errflag[idx] */
445                 *u32ptr_warnCodemsk &= ~err_enable_bitmask;     /* WARNING: clear corresponding bit in warnflag[idx] */
446                 (*u16ptr_threshcounter) = 0;
447                 /* Make entry in error-memory (error disappeared) */
448                 if (recordingenabled == DIAG_RECORDING_ENABLED)
449                     DIAG_EntryWrite(diag_ch_id, event, item_nr);
450
451                 /* Call callback function and reset error */
452                 diag_ch_cfg[diag.id2ch[diag_ch_id]].callbackfunc(diag_ch_id, DIAG_EVENT_RESET);
453             }
454         }
455         ret_val = DIAG_HANDLER_RETURN_OK; /* Function does not return an error-message! */
456     } else if (event == DIAG_EVENT_NOK) {
457         if (diag.err_enableflag[err_enable_idx] & err_enable_bitmask) {
458             if ((*u16ptr_threshcounter) < cfg_threshold) {
459                 (*u16ptr_threshcounter)++;   /* error-threshold not exceeded yet, increment Error-Counter */
460                 ret_val = DIAG_HANDLER_RETURN_OK; /* Function does not return an error-message! */
461             } else if ((*u16ptr_threshcounter) == cfg_threshold) {
462                 /* Error occured AND error-threshold exceeded */
```

```
463                 (*u16ptr_threshcounter)++;
464                 *u32ptr_errCodemsk |= err_enable_bitmask;      /* ERROR:   set corresponding bit in errflag[idx] */
465                 *u32ptr_warnCodemsk &= ~err_enable_bitmask;    /* WARNING: clear corresponding bit in warnflag[idx] */
466
467                 /* Make entry in error-memory (error occurred) */
468                 if (recordingenabled == DIAG_RECORDING_ENABLED) {
469                     DIAG_EntryWrite(diag_ch_id, event, item_nr);
470                 }
471
472                 /* Call callback function and set error */
473                 diag_ch_cfg[diag.id2ch[diag_ch_id]].callbackfunc(diag_ch_id, DIAG_EVENT_NOK);
474                 /* Function returns an error-message! */
475                 ret_val = DIAG_HANDLER_RETURN_ERR_OCCURRED;
476             } else if (((*u16ptr_threshcounter) > cfg_threshold)) {
477                 /* error-threshold already exceeded, nothing to be handled */
478                 ret_val = DIAG_HANDLER_RETURN_ERR_OCCURRED;
479             }
480         } else {
481             /* Error occured BUT NOT enabled by mask */
482             *u32ptr_errCodemsk &= ~err_enable_bitmask;        /* ERROR:   clear corresponding bit in errflag[idx] */
483             *u32ptr_warnCodemsk |= err_enable_bitmask;        /* WARNING: set corresponding bit in warnflag[idx] */
484             ret_val = DIAG_HANDLER_RETURN_WARNING_OCCURRED;   /* Function returns an error-message! */
485         }
486     } else if (event == DIAG_EVENT_RESET) {
487         if (diag.err_enableflag[err_enable_idx] & err_enable_bitmask) {
488             /* clear counter, Error-, Warning-Flag and make recording if enabled */
489             *u32ptr_errCodemsk &= ~err_enable_bitmask;      /* ERROR:   clear corresponding bit in errflag[idx] */
490             *u32ptr_warnCodemsk &= ~err_enable_bitmask;     /* WARNING: clear corresponding bit in warnflag[idx] */
491             (*u16ptr_threshcounter) = 0;
492             if (recordingenabled == DIAG_RECORDING_ENABLED)
493                 DIAG_EntryWrite(diag_ch_id, event, item_nr);      /* Make entry in error-memory (error disappeared)
                    if error was recorded before */
494         }
495         ret_val = DIAG_HANDLER_RETURN_OK; /* Function does not return an error-message! */
496     }
497
498
499     return (ret_val);
500 }
501
502
503 STD_RETURN_TYPE_e DIAG_checkEvent(STD_RETURN_TYPE_e cond,
504                                   DIAG_CH_ID_e diag_ch_id,
505                                   uint32_t item_nr) {
506     STD_RETURN_TYPE_e retVal = E_NOT_OK;
507
508     if (cond == E_OK) {
509         DIAG_Handler(diag_ch_id, DIAG_EVENT_OK, item_nr);
510     } else {
511         DIAG_Handler(diag_ch_id, DIAG_EVENT_NOK, item_nr);
512     }
513
```

```
514         return retVal;
515     }
516
517     #if BUILD_MODULE_ENABLE_CONTACTOR == 1
518     /**
519      * @brief DIAG_ContHandler provides generic contactor switching handling, based on configuration.
520      *
521      * This function does all the handling based on the user defined configuration.
522      * It needs to get called in every occurrence where a contactor is either opened or closed.
523      * According to its return value further treatment is left to the calling module itself.
524      *
525      * @param eventID     switching event that occurred
526      * @param cont_nr     contactor that switches
527      * @param openingCur  current flow during contactor opening
528      *
529      * @return  DIAG_HANDLER_RETURN_ERR_OCCURRED if hard opening threshold is reached,\n
530      *          DIAG_HANDLER_RETURN_OK if normal opening/closing of contactor occurred or threshold not reached,\n
531      *          DIAG_HANDLER_INVALID_TYPE if called with wrong eventID,\n
532      *          DIAG_HANDLER_INVALID_DATA if no opening current is passed in case of hard opening
533      */
534     DIAG_RETURNTYPE_e DIAG_ContHandler(DIAG_CH_ID_e eventID, uint32_t cont_nr, float* openingCur) {
535         DIAG_RETURNTYPE_e retVal = DIAG_HANDLER_RETURN_UNKNOWN;
536         uint8_t updateNVRAM = 0;
537
538         if (diag_locked)
539             return retVal;    /*  only locked when clearing the diagnosis memory */
540         DIAG_CONTACTOR_s diagContactor;
541         NVM_Get_contactorcnt(&diagContactor);
542         if (eventID == DIAG_CH_CONTACTOR_OPENING) {
543             diagContactor.cont_switch_opened[cont_nr]++;
544             retVal = DIAG_HANDLER_RETURN_OK;
545         } else if (eventID == DIAG_CH_CONTACTOR_CLOSING) {
546             diagContactor.cont_switch_closed[cont_nr]++;
547             retVal = DIAG_HANDLER_RETURN_OK;
548         } else if (eventID == DIAG_CH_CONTACTOR_DAMAGED) {
549             if (NULL_PTR == openingCur) {
550                 retVal = DIAG_HANDLER_INVALID_DATA;
551             } else {
552                 RTC_Time_s currTime;
553                 RTC_Date_s currDate;
554
555                 updateNVRAM = 1;
556
557                 diagContactor.cont_switch_opened_hard_at_current[cont_nr]++;
558
559                 if (diagContactor.cont_switch_opened_hard_at_current[cont_nr] >= CONT_NUMBER_OF_BAD_COUNTINGS)
560                     retVal = DIAG_HANDLER_RETURN_ERR_OCCURRED;
561                 else
562                     retVal = DIAG_HANDLER_RETURN_OK;
563
564                 if (diagContactorError_entry_wrptr >= &diagContactorErrorMemory[DIAG_FAIL_ENTRY_CONTACTOR_LENGTH]) {
565                     diagContactorError_entry_wrptr = &diagContactorErrorMemory[0];
```

```
566                 }
567
568                 /* Write error entry */
569
570                 /* Get time and date */
571                 RTC_getTime(&currTime);
572                 RTC_getDate(&currDate);
573
574                 /* Set time and date */
575                 diagContactorError_entry_wrptr->YY = currDate.Year;
576                 diagContactorError_entry_wrptr->MM = currDate.Month;
577                 diagContactorError_entry_wrptr->DD = currDate.Date;
578                 diagContactorError_entry_wrptr->hh = currTime.Hours;
579                 diagContactorError_entry_wrptr->mm = currTime.Minutes;
580                 diagContactorError_entry_wrptr->ss = currTime.Seconds;
581
582                 diagContactorError_entry_wrptr->contactor = (uint8_t)cont_nr;
583                 diagContactorError_entry_wrptr->openingCurrent =  *openingCur;
584
585                 diagContactorError_entry_wrptr++;
586
587                 printf("new Contactor error entry! currently %02d error entrys\r\n", diagContactor.errcntreported);
588             }
589         } else {
590             retVal = DIAG_HANDLER_INVALID_TYPE;
591         }
592         if ((DIAG_HANDLER_RETURN_ERR_OCCURRED == retVal) || (DIAG_HANDLER_RETURN_OK == retVal)) {
593             /* Write new value in nvram buffer */
594             NVM_Set_contactorcnt(&diagContactor);
595             if (updateNVRAM == 1) {
596                 /* Update NVRAM only if contactor opened hard at current */
597                 NVRAM_setWriteRequest(NVRAM_BLOCK_ID_CONT_COUNTER);
598             }
599         }
600         return retVal;
601     }
602     #endif
603
604     /**
605      * @brief overall system monitoring
606      *
607      * checks notifications (state and timestamps) of all system-relevant tasks or functions
608      * all checks should be customized corresponding to its timing and state requirements
609      */
610     void DIAG_SysMon(void) {
611         DIAG_SYSMON_MODULE_ID_e module_id;
612         uint32_t localTimer = OS_getOSSysTick();
613         if (diagsysmonTimestamp == localTimer) {
614             return;
615         }
616         diagsysmonTimestamp = localTimer;
617
```

```
618         /* check modules */
619         for (module_id = 0; module_id < DIAG_SYSMON_MODULE_ID_MAX; module_id++) {
620             if ((diag_sysmon_ch_cfg[module_id].type == DIAG_SYSMON_CYCLICTASK) &&
621                (diag_sysmon_ch_cfg[module_id].state == DIAG_ENABLED)) {
622                 if (diag_sysmon[module_id].timestamp -  diag_sysmon_last[module_id].timestamp <  1) {
623                     /* module not running */
624                     if (++diag_sysmon_cnt[module_id] >= diag_sysmon_ch_cfg[module_id].threshold) {
625                         /* @todo configurable timeouts ! */
626                         if (diag_sysmon_ch_cfg[module_id].enablerecording == DIAG_RECORDING_ENABLED) {
627                             DIAG_Handler(DIAG_CH_SYSTEMMONITORING_TIMEOUT, DIAG_EVENT_NOK, module_id);
628                         }
629 #if BUILD_MODULE_ENABLE_CONTACTOR == 1
630                         if (diag_sysmon_ch_cfg[module_id].handlingtype == DIAG_SYSMON_HANDLING_SWITCHOFFCONTACTOR) {
631                             /* system not working trustfully, switch off contactors! */
632                             CONT_SwitchAllContactorsOff();
633                         }
634 #endif
635                         diag_sysmon_cnt[module_id] = 0;
636 
637                         /* @todo: call callback function if error occurred */
638                         diag_sysmon_ch_cfg[module_id].callbackfunc(module_id);
639                     }
640                 } else {
641                     /* module running */
642                     diag_sysmon_cnt[module_id] = 0;
643 
644                     if (diag_sysmon[module_id].state != 0) {
645                         /* check state of module */
646                         /* @todo: do something now! */
647                     }
648                 }
649             } else {
650                 /* if Sysmon type != cyclic task (not used at the moment) */
651             }
652             diag_sysmon_last[module_id] = diag_sysmon[module_id];      /*save last values for next check*/
653         }
654     }
655 
656 
657     void DIAG_SysMonNotify(DIAG_SYSMON_MODULE_ID_e module_id, uint32_t state) {
658         if (module_id < DIAG_SYSMON_MODULE_ID_MAX) {
659             taskENTER_CRITICAL();
660             diag_sysmon[module_id].timestamp = OS_getOSSysTick();
661             diag_sysmon[module_id].state = state;
662             taskEXIT_CRITICAL();
663         }
664     }
665 
666 
667     void DIAG_configASSERT(void) {
668     #ifdef STM32F4
669         uint32_t lr_register;
```

```
670         uint32_t sp_register;
671         __ASM volatile("mov %0, r14" : "=r" (lr_register));
672         __ASM volatile("mov %0, r13" : "=r" (sp_register));
673 
674         lr_register = lr_register & 0xFFFFFFFE;     /* mask out LSB as this only is indicates thumb instruction */
675         diag_fc.Val0 = sp_register;                 /* actual stack pointer */
676         diag_fc.Val1 = lr_register;                 /* report instruction address where this function has been called */
677         diag_fc.Val2 = *(uint32_t*)(sp_register + 0x1C);        /* return address of callers context (one above caller) */
678         DIAG_Handler(DIAG_CH_CONFIGASSERT, DIAG_EVENT_NOK, 0);
679     #endif
680 
681         while (1) {
682             /* TODO: explain why infinite loop */
683         }
684     }
685 
```