

```

1  /**
2  *
3  *  @copyright &copy; 2010 - 2020, Fraunhofer-Gesellschaft zur Foerderung der
4  *  angewandten Forschung e.V. All rights reserved.
5  *
6  *  BSD 3-Clause License
7  *  Redistribution and use in source and binary forms, with or without
8  *  modification, are permitted provided that the following conditions are met:
9  *  1. Redistributions of source code must retain the above copyright notice,
10 *  this list of conditions and the following disclaimer.
11 *  2. Redistributions in binary form must reproduce the above copyright
12 *  notice, this list of conditions and the following disclaimer in the
13 *  documentation and/or other materials provided with the distribution.
14 *  3. Neither the name of the copyright holder nor the names of its
15 *  contributors may be used to endorse or promote products derived from
16 *  this software without specific prior written permission.
17 *
18 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
19 *  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
20 *  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
21 *  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
22 *  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
23 *  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
24 *  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
25 *  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
26 *  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
27 *  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
28 *  POSSIBILITY OF SUCH DAMAGE.
29 *
30 *  We kindly request you to use one or more of the following phrases to refer
31 *  to foxBMS in your hardware, software, documentation or advertising
32 *  materials:
33 *
34 *  &Prime;This product uses parts of foxBMS&reg;&Prime;;
35 *
36 *  &Prime;This product includes parts of foxBMS&reg;&Prime;;
37 *
38 *  &Prime;This product is derived from foxBMS&reg;&Prime;;
39 *
40 */
41
42 /**
43 *  @file    sys.c
44 *  @author  foxBMS Team
45 *  @date    21.09.2015 (date of creation)
46 *  @ingroup ENGINE
47 *  @prefix  SYS
48 *
49 *  @brief   Sys driver implementation
50 */
51
52

```

```

53  /*===== Includes =====*/
54  #include "sys.h"
55
56  #include "bal.h"
57  #include "bms.h"
58  #include "cansignal.h"
59  #include "contactor.h"
60  #include "diag.h"
61  #include "interlock.h"
62  #include "isoguard.h"
63  #include "meas.h"
64  #include "rtc.h"
65  #include "sox.h"
66  #include "FreeRTOS.h"
67  #include "task.h"
68
69  /*===== Macros and Definitions =====*/
70
71  /**
72   * Saves the last state and the last substate
73   */
74  #define SYS_SAVELASTSTATES()    sys_state.laststate = sys_state.state; \
75                                sys_state.lastsubstate = sys_state.substate
76
77  /*===== Constant and Variable Definitions =====*/
78
79  /**
80   * contains the state of the contactor state machine
81   *
82   */
83  static SYS_STATE_s sys_state = {
84      .timer                = 0,
85      .statereq              = SYS_STATE_NO_REQUEST,
86      .state                 = SYS_STATEMACH_UNINITIALIZED,
87      .substate              = SYS_ENTRY,
88      .laststate             = SYS_STATEMACH_UNINITIALIZED,
89      .lastsubstate          = 0,
90      .triggerentry          = 0,
91      .ErrRequestCounter     = 0,
92  };
93
94  /*===== Function Prototypes =====*/
95
96  static SYS_RETURN_TYPE_e SYS_CheckStateRequest(SYS_STATE_REQUEST_e statereq);
97  static SYS_STATE_REQUEST_e SYS_GetStateRequest(void);
98  static SYS_STATE_REQUEST_e SYS_TransferStateRequest(void);
99  static uint8_t SYS_CheckReEntrance(void);
100
101  /*===== Function Implementations =====*/
102
103  /**
104   * @brief re-entrance check of SYS state machine trigger function

```

```

105  *
106  * This function is not re-entrant and should only be called time- or event-triggered.
107  * It increments the triggerentry counter from the state variable ltc_state.
108  * It should never be called by two different processes, so if it is the case, triggerentry
109  * should never be higher than 0 when this function is called.
110  *
111  *
112  * @return  retval  0 if no further instance of the function is active, 0xff else
113  *
114  */
115  static uint8_t SYS_CheckReEntrance(void) {
116      uint8_t retval = 0;
117
118      taskENTER_CRITICAL();
119      if (!sys_state.triggerentry) {
120          sys_state.triggerentry++;
121      } else {
122          retval = 0xFF; /* multiple calls of function */
123      }
124      taskEXIT_CRITICAL();
125
126      return retval;
127  }
128
129
130
131
132  /**
133   * @brief  gets the current state request.
134   *
135   * This function is used in the functioning of the SYS state machine.
136   *
137   * @return  retval  current state request, taken from SYS_STATE_REQUEST_e
138   */
139  static SYS_STATE_REQUEST_e SYS_GetStateRequest(void) {
140      SYS_STATE_REQUEST_e retval = SYS_STATE_NO_REQUEST;
141
142      taskENTER_CRITICAL();
143      retval = sys_state.statereq;
144      taskEXIT_CRITICAL();
145
146      return (retval);
147  }
148
149
150  SYS_STATEMACH_e SYS_GetState(void) {
151      return (sys_state.state);
152  }
153
154
155  /**
156   * @brief  transfers the current state request to the state machine.

```

```

157  *
158  * This function takes the current state request from #sys_state and transfers it to the state machine.
159  * It resets the value from #sys_state to #SYS_STATE_NO_REQUEST
160  *
161  * @return  retVal          current state request, taken from #SYS_STATE_REQUEST_e
162  *
163  */
164  static SYS_STATE_REQUEST_e SYS_TransferStateRequest(void) {
165      SYS_STATE_REQUEST_e retval = SYS_STATE_NO_REQUEST;
166
167      taskENTER_CRITICAL();
168      retval = sys_state.statereq;
169      sys_state.statereq = SYS_STATE_NO_REQUEST;
170      taskEXIT_CRITICAL();
171
172      return (retval);
173  }
174
175
176
177  SYS_RETURN_TYPE_e SYS_SetStateRequest(SYS_STATE_REQUEST_e statereq) {
178      SYS_RETURN_TYPE_e retVal = SYS_ILLEGAL_REQUEST;
179
180      taskENTER_CRITICAL();
181      retVal = SYS_CheckStateRequest(statereq);
182
183      if (retVal == SYS_OK) {
184          sys_state.statereq = statereq;
185      }
186      taskEXIT_CRITICAL();
187
188      return (retVal);
189  }
190
191
192
193  /**
194   * @brief   checks the state requests that are made.
195   *
196   * This function checks the validity of the state requests.
197   * The results of the checked is returned immediately.
198   *
199   * @param   statereq      state request to be checked
200   *
201   * @return  result of the state request that was made, taken from SYS_RETURN_TYPE_e
202   */
203  static SYS_RETURN_TYPE_e SYS_CheckStateRequest(SYS_STATE_REQUEST_e statereq) {
204      SYS_RETURN_TYPE_e retVal = SYS_ILLEGAL_REQUEST;
205      if (statereq == SYS_STATE_ERROR_REQUEST) {
206          retVal = SYS_OK;
207      } else {
208          if (sys_state.statereq == SYS_STATE_NO_REQUEST) {

```

```

209         /* init only allowed from the uninitialized state */
210         if (statereq == SYS_STATE_INIT_REQUEST) {
211             if (sys_state.state == SYS_STATEMACH_UNINITIALIZED) {
212                 retval = SYS_OK;
213             } else {
214                 retval = SYS_ALREADY_INITIALIZED;
215             }
216         } else {
217             retval = SYS_ILLEGAL_REQUEST;
218         }
219     } else {
220         retval = SYS_REQUEST_PENDING;
221     }
222 }
223 return retval;
224 }
225
226
227 void SYS_Trigger(void) {
228     /* STD_RETURN_TYPE_e retVal=E_OK; */
229     SYS_STATE_REQUEST_e statereq = SYS_STATE_NO_REQUEST;
230     ILCK_STATEMACH_e ilckstate = ILCK_STATEMACH_UNDEFINED;
231     STD_RETURN_TYPE_e constate = E_NOT_OK;
232     STD_RETURN_TYPE_e balInitState = E_NOT_OK;
233     STD_RETURN_TYPE_e bmsstate = E_NOT_OK;
234
235
236     DIAG_SysMonNotify(DIAG_SYSMON_SYS_ID, 0); /* task is running, state = ok */
237     /* Check re-entrance of function */
238     if (SYS_CheckReEntrance()) {
239         return;
240     }
241
242     if (sys_state.timer) {
243         if (--sys_state.timer) {
244             sys_state.triggerentry--;
245             return; /* handle state machine only if timer has elapsed */
246         }
247     }
248
249     /*****Happens every time the state machine is triggered*****/
250
251
252     switch (sys_state.state) {
253         /******UNINITIALIZED******/
254         case SYS_STATEMACH_UNINITIALIZED:
255             /* waiting for Initialization Request */
256             statereq = SYS_TransferStateRequest();
257             if (statereq == SYS_STATE_INIT_REQUEST) {
258                 SYS_SAVELASTSTATES();
259                 sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
260                 sys_state.state = SYS_STATEMACH_INITIALIZATION;

```

```

261         sys_state.substate = SYS_ENTRY;
262     } else if (statereq == SYS_STATE_NO_REQUEST) {
263         /* no actual request pending */
264     } else {
265         sys_state.ErrRequestCounter++; /* illegal request pending */
266     }
267     break;
268 /*****INITIALIZATION*****/
269 case SYS_STATEMACH_INITIALIZATION:
270
271     SYS_SAVELASTSTATES();
272     /* Initializations done here */
273
274     /* Send CAN boot message directly on CAN */
275     SYS_SendBootMessage(1);
276
277     /* Check if undervoltage MSL violation was detected before reset */
278     if (RTC_DEEP_DISCHARGE_DETECTED == 1) {
279         /* Error detected */
280         DIAG_Handler(DIAG_CH_DEEP_DISCHARGE_DETECTED, DIAG_EVENT_NOK, 0);
281     } else {
282         DIAG_Handler(DIAG_CH_DEEP_DISCHARGE_DETECTED, DIAG_EVENT_OK, 0);
283     }
284
285     sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
286     sys_state.state = SYS_STATEMACH_INITIALIZED;
287     sys_state.substate = SYS_ENTRY;
288     break;
289
290 /*****INITIALIZED*****/
291 case SYS_STATEMACH_INITIALIZED:
292     SYS_SAVELASTSTATES();
293     sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
294 #if BUILD_MODULE_ENABLE_ILCK == 1
295     sys_state.state = SYS_STATEMACH_INITIALIZE_INTERLOCK;
296 #elif BUILD_MODULE_ENABLE_CONTACTOR == 1
297     sys_state.state = SYS_STATEMACH_INITIALIZE_CONTACTORS;
298 #else
299     sys_state.state = SYS_STATEMACH_INITIALIZE_BALANCING;
300 #endif
301     sys_state.substate = SYS_ENTRY;
302     break;
303
304 #if BUILD_MODULE_ENABLE_ILCK == 1
305 /*****INITIALIZE INTERLOCK*****/
306 case SYS_STATEMACH_INITIALIZE_INTERLOCK:
307     SYS_SAVELASTSTATES();
308
309     if (sys_state.substate == SYS_ENTRY) {
310         ILCK_SetStateRequest(ILCK_STATE_INIT_REQUEST);
311         sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
312         sys_state.substate = SYS_WAIT_INITIALIZATION_INTERLOCK;

```

```

313         sys_state.InitCounter = 0;
314         break;
315     } else if (sys_state.substate == SYS_WAIT_INITIALIZATION_INTERLOCK) {
316         ilckstate = ILCK_GetState();
317         if (ilckstate == ILCK_STATEMACH_WAIT_FIRST_REQUEST) {
318             ILCK_SetStateRequest(ILCK_STATE_OPEN_REQUEST);
319             sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
320 #if BUILD_MODULE_ENABLE_CONTACTOR == 1
321             sys_state.state = SYS_STATEMACH_INITIALIZE_CONTACTORS;
322 #else
323             sys_state.state = SYS_STATEMACH_INITIALIZE_BALANCING;
324 #endif
325             sys_state.substate = SYS_ENTRY;
326             break;
327         } else {
328             if (sys_state.InitCounter > (100/SYS_TASK_CYCLE_CONTEXT_MS)) {
329                 sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
330                 sys_state.state = SYS_STATEMACH_ERROR;
331                 sys_state.substate = SYS_ILCK_INIT_ERROR;
332                 break;
333             }
334             sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
335             sys_state.InitCounter++;
336             break;
337         }
338     }
339
340     break;
341 #endif
342
343 #if BUILD_MODULE_ENABLE_CONTACTOR == 1
344     /*****INITIALIZE CONTACTORS*****/
345     case SYS_STATEMACH_INITIALIZE_CONTACTORS:
346         SYS_SAVELASTSTATES();
347
348         if (sys_state.substate == SYS_ENTRY) {
349             CONT_SetStateRequest(CONT_STATE_INIT_REQUEST);
350
351             sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
352             sys_state.substate = SYS_WAIT_INITIALIZATION_CONT;
353             sys_state.InitCounter = 0;
354             break;
355         } else if (sys_state.substate == SYS_WAIT_INITIALIZATION_CONT) {
356             contstate = CONT_GetInitializationState();
357             if (contstate == E_OK) {
358                 sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
359                 sys_state.state = SYS_STATEMACH_INITIALIZE_BALANCING;
360                 sys_state.substate = SYS_ENTRY;
361                 break;
362             } else {
363                 if (sys_state.InitCounter > (100/SYS_TASK_CYCLE_CONTEXT_MS)) {
364                     sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;

```

```

365         sys_state.state = SYS_STATEMACH_ERROR;
366         sys_state.substate = SYS_CONT_INIT_ERROR;
367         break;
368     }
369     sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
370     sys_state.InitCounter++;
371     break;
372 }
373 }
374
375 break;
376 #endif
377
378 /*****INITIALIZE BALANCING*****/
379 case SYS_STATEMACH_INITIALIZE_BALANCING:
380     SYS_SAVELASTSTATES();
381     if (sys_state.substate == SYS_ENTRY) {
382         BAL_SetStateRequest(BAL_STATE_INIT_REQUEST);
383         sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
384         sys_state.substate = SYS_WAIT_INITIALIZATION_BAL;
385         sys_state.InitCounter = 0;
386         break;
387     } else if (sys_state.substate == SYS_WAIT_INITIALIZATION_BAL) {
388         balInitState = BAL_GetInitializationState();
389         if (BALANCING_DEFAULT_INACTIVE == TRUE) {
390             BAL_SetStateRequest(BAL_STATE_GLOBAL_DISABLE_REQUEST);
391         } else {
392             BAL_SetStateRequest(BAL_STATE_GLOBAL_ENABLE_REQUEST);
393         }
394         if (balInitState == E_OK) {
395             sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
396             sys_state.state = SYS_STATEMACH_INITIALIZE_ISOGUARD;
397             sys_state.substate = SYS_ENTRY;
398             break;
399         } else {
400             if (sys_state.InitCounter > (100/SYS_TASK_CYCLE_CONTEXT_MS)) {
401                 sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
402                 sys_state.state = SYS_STATEMACH_ERROR;
403                 sys_state.substate = SYS_BAL_INIT_ERROR;
404                 break;
405             }
406             sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
407             sys_state.InitCounter++;
408             break;
409         }
410     }
411     break;
412
413 /***** Initialize Isoguard *****/
414 case SYS_STATEMACH_INITIALIZE_ISOGUARD:
415
416 #if BUILD_MODULE_ENABLE_ISOGUARD == 1

```



```

417         ISO_Init();
418     #endif
419         sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
420         sys_state.state = SYS_STATEMACH_FIRST_MEASUREMENT_CYCLE;
421         sys_state.substate = SYS_ENTRY;
422         break;
423
424     /*****START FIRST MEAS CYCLE*****/
425     case SYS_STATEMACH_FIRST_MEASUREMENT_CYCLE:
426         SYS_SAVELASTSTATES();
427         if (sys_state.substate == SYS_ENTRY) {
428             MEAS_StartMeasurement();
429             sys_state.InitCounter = 0;
430             sys_state.substate = SYS_WAIT_FIRST_MEASUREMENT_CYCLE;
431         } else if (sys_state.substate == SYS_WAIT_FIRST_MEASUREMENT_CYCLE) {
432             if (MEAS_IsFirstMeasurementCycleFinished() == TRUE) {
433                 MEAS_Request_OpenWireCheck();
434                 sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
435                 if (CURRENT_SENSOR_PRESENT == TRUE)
436                     sys_state.state = SYS_STATEMACH_CHECK_CURRENT_SENSOR_PRESENCE;
437                 else
438                     sys_state.state = SYS_STATEMACH_INITIALIZE_MISC;
439                 sys_state.substate = SYS_ENTRY;
440                 break;
441             } else {
442                 if (sys_state.InitCounter > (100/SYS_TASK_CYCLE_CONTEXT_MS)) {
443                     sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
444                     sys_state.state = SYS_STATEMACH_ERROR;
445                     sys_state.substate = SYS_MEAS_INIT_ERROR;
446                     break;
447                 } else {
448                     sys_state.timer = SYS_STATEMACH_MEDIUMTIME_MS;
449                     sys_state.InitCounter++;
450                     break;
451                 }
452             }
453         }
454         break;
455
456     /*****CHECK CURRENT SENSOR PRESENCE*****/
457     case SYS_STATEMACH_CHECK_CURRENT_SENSOR_PRESENCE:
458         SYS_SAVELASTSTATES();
459
460         if (sys_state.substate == SYS_ENTRY) {
461             sys_state.InitCounter = 0;
462             CANS_Enable_Periodic(TRUE);
463             #if CURRENT_SENSOR_ISABELLENHUETTE_TRIGGERED
464                 /* If triggered mode is used, CAN trigger message needs to
465                  * be transmitted and current sensor response has to be
466                  * received afterwards. This may take some time, therefore
467                  * delay has to be increased.
468                  */

```

```

469         sys_state.timer = SYS_STATEMACH_LONGTIME_MS;
470     #else /* CURRENT_SENSOR_ISABELLENHUETTE_TRIGGERED */
471         sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
472     #endif /* CURRENT_SENSOR_ISABELLENHUETTE_TRIGGERED */
473     sys_state.substate = SYS_WAIT_CURRENT_SENSOR_PRESENCE;
474     } else if (sys_state.substate == SYS_WAIT_CURRENT_SENSOR_PRESENCE) {
475         if (CANS_IsCurrentSensorPresent() == TRUE) {
476             SOF_Init();
477             if (CANS_IsCurrentSensorCCPresent() == TRUE) {
478                 SOC_Init(TRUE);
479             } else {
480                 SOC_Init(FALSE);
481             }
482             sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
483             sys_state.state = SYS_STATEMACH_INITIALIZE_MISC;
484             sys_state.substate = SYS_ENTRY;
485             break;
486         } else {
487             if (sys_state.InitCounter > (100/SYS_TASK_CYCLE_CONTEXT_MS)) {
488                 sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
489                 sys_state.state = SYS_STATEMACH_ERROR;
490                 sys_state.substate = SYS_CURRENT_SENSOR_PRESENCE_ERROR;
491                 break;
492             } else {
493                 sys_state.timer = SYS_STATEMACH_MEDIUMTIME_MS;
494                 sys_state.InitCounter++;
495                 break;
496             }
497         }
498     }
499     break;
500
501     /*****INITIALIZED_MISC*****/
502     case SYS_STATEMACH_INITIALIZE_MISC:
503         SYS_SAVELASTSTATES();
504
505         if (CURRENT_SENSOR_PRESENT == FALSE) {
506             CANS_Enable_Periodic(TRUE);
507             SOC_Init(FALSE);
508         }
509
510         sys_state.timer = SYS_STATEMACH_MEDIUMTIME_MS;
511         sys_state.state = SYS_STATEMACH_INITIALIZE_BMS;
512         sys_state.substate = SYS_ENTRY;
513         break;
514
515     /*****INITIALIZE BMS*****/
516     case SYS_STATEMACH_INITIALIZE_BMS:
517         SYS_SAVELASTSTATES();
518
519         if (sys_state.substate == SYS_ENTRY) {
520             BMS_SetStateRequest(BMS_STATE_INIT_REQUEST);

```

```

521         sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
522         sys_state.substate = SYS_WAIT_INITIALIZATION_BMS;
523         sys_state.InitCounter = 0;
524         break;
525     } else if (sys_state.substate == SYS_WAIT_INITIALIZATION_BMS) {
526         bmsstate = BMS_GetInitializationState();
527         if (bmsstate == E_OK) {
528             sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
529             sys_state.state = SYS_STATEMACH_RUNNING;
530             sys_state.substate = SYS_ENTRY;
531             break;
532         } else {
533             if (sys_state.InitCounter > (100/SYS_TASK_CYCLE_CONTEXT_MS)) {
534                 sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
535                 sys_state.state = SYS_STATEMACH_ERROR;
536                 sys_state.substate = SYS_BMS_INIT_ERROR;
537                 break;
538             }
539             sys_state.timer = SYS_STATEMACH_SHORTTIME_MS;
540             sys_state.InitCounter++;
541             break;
542         }
543     }
544     break;
545
546     /*****RUNNNIG*****/
547     case SYS_STATEMACH_RUNNING:
548         SYS_SAVELASTSTATES();
549         sys_state.timer = SYS_STATEMACH_LONGTIME_MS;
550         break;
551
552     /*****ERROR*****/
553     case SYS_STATEMACH_ERROR:
554         SYS_SAVELASTSTATES();
555         CANS_Enable_Periodic(TRUE);
556         sys_state.timer = SYS_STATEMACH_LONGTIME_MS;
557         break;
558     /*****DEFAULT CASE*****/
559     default:
560         /* This default case should never be entered.
561          * If we actually enter this case, it means that an
562          * unrecoverable error has occurred. Therefore the program
563          * will trap.
564          */
565         configASSERT(0);
566         break;
567 } /* end switch (sys_state.state) */
568 sys_state.triggerentry--;
569 }
570

```