

```

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    can.c
44 *  @author  foxBMS Team
45 *  @date    12.07.2015 (date of creation)
46 *  @ingroup DRIVERS
47 *  @prefix  CAN
48 *
49 *  @brief   Driver for the CAN module
50 *
51 *  Implementation of the CAN Interrupts, initialization, buffers,
52 *  receive and transmit interfaces.

```

```

53  *
54  */
55
56  /*===== Includes =====*/
57  #include "can.h"
58
59  /*===== Macros and Definitions =====*/
60  #define ID_16BIT_FIFO0      (0U)
61  #define ID_16BIT_FIFO1      (1U)
62  #define ID_32BIT_FIFO0      (2U)
63  #define ID_32BIT_FIFO1      (3U)
64  #define MSK_16BIT_FIFO0      (4U)
65  #define MSK_16BIT_FIFO1      (5U)
66  #define MSK_32BIT            (6U)
67
68  /*===== Constant and Variable Definitions =====*/
69  uint8_t canNode0_listenonly_mode = 0;
70  uint8_t canNode1_listenonly_mode = 0;
71
72  #if CAN_USE_CAN_NODE0
73  #if CAN0_USE_TX_BUFFER
74  CAN_TX_BUFFERELEMENT_s can0_txbufferelements[CAN0_TX_BUFFER_LENGTH];
75  CAN_TX_BUFFER_s can0_txbuffer = {
76      .length = CAN0_TX_BUFFER_LENGTH,
77      .buffer = &can0_txbufferelements[0],
78  };
79  #endif /* CAN0_USE_TX_BUFFER */
80
81  #if CAN0_USE_RX_BUFFER
82  CAN_RX_BUFFERELEMENT_s can0_rxbufferelements[CAN0_RX_BUFFER_LENGTH];
83  CAN_RX_BUFFER_s can0_rxbuffer = {
84      .length = CAN0_RX_BUFFER_LENGTH,
85      .buffer = &can0_rxbufferelements[0],
86  };
87  #endif /* CAN0_USE_RX_BUFFER */
88
89  #if CAN0_BUFFER_BYPASS_NUMBER_OF_IDS > 0    This is for Rx.
90  uint8_t can0_fastLinkIndex[CAN0_BUFFER_BYPASS_NUMBER_OF_IDS];    /* Link Table for bufferBypassing */
91  #endif
92
93  CAN_ERROR_s CAN0_errorStruct = {
94      .canError = HAL_CAN_ERROR_NONE,
95      .canErrorCounter = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
96  };
97  #endif /* CAN_USE_CAN_NODE0 */
98
99  #if CAN_USE_CAN_NODE1
100  #if CAN1_USE_TX_BUFFER
101  CAN_TX_BUFFERELEMENT_s can1_txbufferelements[CAN1_TX_BUFFER_LENGTH];
102  CAN_TX_BUFFER_s can1_txbuffer = {
103      .length = CAN1_TX_BUFFER_LENGTH,
104      .buffer = &can1_txbufferelements[0],

```

```

105 };
106 #endif /* CAN1_USE_TX_BUFFER */
107
108 #if CAN1_USE_RX_BUFFER
109 CAN_RX_BUFFERELEMENT_s can1_rxbufferelements[CAN1_RX_BUFFER_LENGTH];
110 CAN_RX_BUFFER_s can1_rxbuffer = {
111     .length = CAN1_RX_BUFFER_LENGTH,
112     .buffer = &can1_rxbufferelements[0],
113 };
114 #endif /* CAN1_USE_RX_BUFFER */
115
116 #if CAN1_BUFFER_BYPASS_NUMBER_OF_IDS > 0
117 uint8_t can1_fastLinkIndex[CAN1_BUFFER_BYPASS_NUMBER_OF_IDS]; /* Link Table for bufferBypassing */
118 #endif
119
120 CAN_ERROR_s CAN1_errorStruct = {
121     .canError = HAL_CAN_ERROR_NONE,
122     .canErrorCounter = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
123 };
124 #endif /* CAN_USE_CAN_NODE1 */
125
126 /* *****
127  * Dummies for filter initialization and message reception
128  * *****
129
130 CAN_FilterTypeDef sFilterConfig = {
131     /* No need to insert here something */
132     .FilterActivation = ENABLE, /* enable the filter */
133 };
134
135 /*===== Function Prototypes =====*/
136 /* Inits */
137 static uint8_t CAN_GetNextID(CAN_MSG_RX_TYPE_s* can_RxMsgs, uint8_t numberOfRxIDs, uint8_t startIndex,
138     uint8_t filterCase);
139 static uint8_t CAN_NumberOfNeededFilters(CAN_MSG_RX_TYPE_s* can_RxMsgs, uint8_t* numberOfDifferentIDs, uint32_t*
140     error);
141 static uint32_t CAN_InitFilter(CAN_HandleTypeDef* ptrHcan, CAN_MSG_RX_TYPE_s* can_RxMsgs, uint8_t numberOfRxMsgs);
142
143 /* Interrupts */
144 static void CAN_TxCpltCallback(CAN_NodeTypeDef_e canNode);
145 static void CAN_RxMsg(CAN_NodeTypeDef_e canNode, CAN_HandleTypeDef* ptrHcan, uint8_t FIFONumber);
146
147 /* Buffer/Interpreter */
148 static STD_RETURN_TYPE_e CAN_BufferBypass(CAN_NodeTypeDef_e canNode, uint32_t msgID, uint8_t* rxData, uint8_t DLC,
149     uint8_t RTR);
150 static STD_RETURN_TYPE_e CAN_InterpreterReceivedMsg(CAN_NodeTypeDef_e canNode, uint32_t msgID, uint8_t* data, uint8_t
151     DLC,
152     uint8_t RTR);
153
154 /*===== Function Implementations =====*/
155 /* *****

```

The screenshot shows an IDE with four tabs: `cansignal_cfg.h`, `stm32f4xx_hal_can.h`, `cansignal_cfg.c`, and `can_cfg.c`. The active file is `stm32f4xx_hal_can.h`, showing the definition of `CAN_FilterTypeDef` starting at line 150. The structure includes fields like `FilterScale`, `FilterActivation`, and `SlaveStartFilterBank` with associated comments. A red arrow points from the `CAN_HandleTypeDef` parameter in the main code to the `CAN_FilterTypeDef` definition in the header file.

```

155 * Initialization
156 *****
157
158 uint32_t CAN_Init(void) {
159     uint32_t retval = 0;
160
161     #if CAN_USE_CAN_NODE0
162         /* DeInit CAN0 handle */
163         if (HAL_CAN_DeInit(&hcan0) != HAL_OK) {
164             /* Error deinitializing handle -> set error bit */
165             retval |= STD_ERR_BIT_0;
166         }
167
168         /* Init CAN0-handle */
169         if (HAL_CAN_Init(&hcan0) != HAL_OK) {
170             /* Error intializing handle -> set error bit */
171             retval |= STD_ERR_BIT_1;
172         }
173
174         /* Configure CAN0 hardware filter */
175         retval |= CAN_InitFilter(&hcan0, &can0_RxMsgs[0], can_CAN0_rx_length);
176
177         /* Check if more rx messages are bypassed than received */
178         #pragma GCC diagnostic push
179         /* configurations might exist that use this comparison */
180         #pragma GCC diagnostic ignored "-Wtype-limits"
181         if (CAN0_BUFFER_BYPASS_NUMBER_OF_IDS > can_CAN0_rx_length) {
182             #pragma GCC diagnostic pop
183             retval |= STD_ERR_BIT_7;
184         }
185
186         /* Enable CAN0 message receive interrupt FIFO0 */
187         if (HAL_CAN_ActivateNotification(&hcan0, CAN_IT_RX_FIFO0_MSG_PENDING) != HAL_OK) {
188             retval |= STD_ERR_BIT_8;
189         }
190         hcan0.State = HAL_CAN_STATE_READY;
191         /* Enable CAN0 message receive interrupt FIFO1 */
192         if (HAL_CAN_ActivateNotification(&hcan0, CAN_IT_RX_FIFO1_MSG_PENDING) != HAL_OK) {
193             retval |= STD_ERR_BIT_9;
194         }
195         /* Enable CAN0 Transmit mailbox empty interrupt */
196         if (HAL_CAN_ActivateNotification(&hcan0, CAN_IT_TX_MAILBOX_EMPTY) != HAL_OK) {
197             retval |= STD_ERR_BIT_10;
198         }
199
200         /* Debug freeze
201         /* set DBF bit to 0 for CAN activity while in debug mode */ DBF = 0 ==> CAN works during debug.
202         CLEAR_BIT(hcan0.Instance->MCR, CAN_MCR_DBF);
203
204         #if CAN_USE_STANDBY_CONTROL == 1
205             IO_WritePin(CAN_0_TRANS_STANDBY_CONTROL, IO_PIN_SET);
206         #endif /* CAN_USE_STANDBY_CONTROL == 1 */

```

```

C cansignal_cfg.h C can_cfg.h C stm32f4xx_hal_can.h
mcu-hal > STM32F4xx_HAL_Driver > Inc > C stm32f4xx_hal_can.h >
You, a month ago | 1 author (You)
typedef struct __CAN_HandleTypeDef
225 {
226     CAN_HandleTypeDef *Instance;
227     CAN_InitTypeDef Init;
228     __IO HAL_CAN_StateTypeDef State;
229     __IO uint32_t ErrorCode;
230 } CAN_HandleTypeDef;

```

```

C cansignal_cfg.h C can_cfg.h X C stm32f4xx_hal_can.h C cansignal_cfg.c
mcu-primary > src > driver > config > C can_cfg.h > CAN_MSG_RX_TYPE_s
297 typedef struct CAN_MSG_RX_TYPE {
298     uint32_t ID; /*!< message ID */
299     uint32_t mask; /*!< mask or 0x0000 to select list mode */
300     uint8_t DLC; /*!< data length */
301     uint8_t RTR; /*!< rtr bit */
302     uint32_t fifo; /*!< selected CAN hardware (CAN_FILTER_FIFO0 or CAN_FILTER_FIFO1) */
303     STD_RETURN_TYPE_e (*func)(uint32_t ID, uint8_t*, uint8_t, uint8_t);
304 } CAN_MSG_RX_TYPE_s;

```

```

C cansignal_cfg.h C can_cfg.c X C can_cfg.h C stm32f4xx_hal_can.h C cansignal_cfg.c C can.c
mcu-primary > src > driver > config > C can_cfg.c > can0_RxMsgs
453 /* Bypassed messages are --- ALSO --- to be configured here. See further down for bypass ID setting! */
454 CAN_MSG_RX_TYPE_s can0_RxMsgs[] = {
455     { 0x120, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< state request */
456     { CAN_ID_SOFTWARE_RESET_MSG, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< software reset */
457     #ifdef CURRENT_SENSOR_ISABELLENHUETTE_TRIGGERED
458     { 0x35C, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< current sensor I */
459     { 0x35D, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< current sensor U1 */
460     { 0x35E, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< current sensor U2 */
461     { 0x35F, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< current sensor U3 */
462     { 0x525, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< current sensor T in cyclic mode */
463     { 0x526, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< current sensor Power in cyclic mode */
464     { 0x527, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< current sensor C-C in cyclic mode */
465     { 0x528, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< current sensor E-C in cyclic mode */
466     #else /* CURRENT_SENSOR_ISABELLENHUETTE_CYCLIC */
467     { 0x521, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< current sensor I in cyclic mode */
468     { 0x522, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< current sensor U1 in cyclic mode */
469     { 0x523, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< current sensor U2 in cyclic mode */
470     { 0x524, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< current sensor U3 in cyclic mode */
471     { 0x525, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< current sensor T in cyclic mode */
472     { 0x526, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< current sensor Power in cyclic mode */
473     { 0x527, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< current sensor C-C in cyclic mode */
474     { 0x528, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< current sensor E-C in cyclic mode */
475     #endif /* CURRENT_SENSOR_ISABELLENHUETTE_TRIGGERED */
476     { 0x100, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< debug message */
477     { 0x777, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< request SW version */
478     { 0x121, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL }, /*!< engine request */
479 };

```

This should have been called
can0_RxMsgHeader since it is
not the true message.

It seems that FIFO 1 is never used.

```

C cansignal_cfg.h C can_cfg.c X C can_cfg.h C stm32f4xx_hal_can.h C cansignal_cfg.c
mcu-primary > src > driver > config > C can_cfg.c > can_CAN0_rx_length
487
488 const uint8_t can_CAN0_rx_length = sizeof(can0_RxMsgs)/sizeof(can0_RxMsgs[0]);

```

```

207     /* Start CAN */
208     HAL_CAN_Start(&hcan0);
209 #endif /* CAN_USE_CAN_NODE0 */
210
211
212 #if CAN_USE_CAN_NODE1
213     /* DeInit CAN1 handle */
214     if (HAL_CAN_DeInit(&hcan1) != HAL_OK) {
215         /* Error deinitializing handle -> set error bit */
216         retval |= STD_ERR_BIT_11;
217     }
218
219     /* Init CAN1-handle */
220     if (HAL_CAN_Init(&hcan1) != HAL_OK) {
221         /* Error intializing handle -> set error bit */
222         retval |= STD_ERR_BIT_12;
223     }
224
225     /* Configure CAN1 hardware filter */
226     retval |= CAN_InitFilter(&hcan1, &can1_RxMsgs[0], can_CAN1_rx_length);
227
228     /* Check if more RX messages are bypassed than received */
229 #pragma GCC diagnostic push
230     /* configurations might exist that use this comparison */
231 #pragma GCC diagnostic ignored "-Wtype-limits"
232     if (CAN1_BUFFER_BYPASS_NUMBER_OF_IDS > can_CAN1_rx_length) {
233 #pragma GCC diagnostic pop
234         retval |= STD_ERR_BIT_13;
235     }
236
237     /* Enable CAN1 message receive interrupt FIFO0 */
238     if (HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO0_MSG_PENDING) != HAL_OK) {
239         retval |= STD_ERR_BIT_14;
240     }
241     /* Enable CAN1 message receive interrupt FIFO1 */
242     hcan1.State = HAL_CAN_STATE_READY;
243     if (HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO1_MSG_PENDING) != HAL_OK) {
244         retval |= STD_ERR_BIT_15;
245     }
246     /* Enable CAN1 Transmit mailbox empty interrupt */
247     if (HAL_CAN_ActivateNotification(&hcan1, CAN_IT_TX_MAILBOX_EMPTY) != HAL_OK) {
248         retval |= STD_ERR_BIT_16;
249     }
250
251     /* set DBF bit to 0 for CAN activity while in debug mode */
252     CLEAR_BIT(hcan1.Instance->MCR, CAN_MCR_DBF);
253
254 #if CAN_USE_STANDBY_CONTROL == 1
255     IO_WritePin(CAN_1_TRANS_STANDBY_CONTROL, IO_PIN_SET);
256 #endif /* CAN_USE_STANDBY_CONTROL == 1 */
257
258     /* Start CAN */

```

```

259     HAL_CAN_Start(&hcan1);
260 #endif /* CAN_USE_CAN_NODE1 */
261     return retval;
262 }
263
264 /**
265  * @brief Initializes message filtering
266  * @retval 0: if no error occurred, otherwise error code
267  */
268 static uint32_t CAN_InitFilter(CAN_HandleTypeDef* ptrHcan, CAN_MSG_RX_TYPE_s* can_RxMsgs, uint8_t numberOfRxMsgs) {
269     /* Contains the occurrence of the different filter cases */ (Used for numberOfDifferentIDs)
270     * [0] - ID List mode 16bit routed on FIFO0
271     * [1] - ID List mode 16bit routed on FIFO1
272     * [2] - ID List mode 32bit routed on FIFO0
273     * [3] - ID List mode 32bit routed on FIFO1
274     * [4] - Mask mode 16bit routed on FIFO0
275     * [5] - Mask mode 16bit routed on FIFO1
276     * [6] - Mask mode 32bit
277     uint8_t numberOfDifferentIDs[7] = { 0, 0, 0, 0, 0, 0, 0 };
278     static uint8_t filterNumber = 0; /* Number of the filter to be initialized */
279     uint32_t retval = 0;
280
281     /* Calculate number of needed filter banks */ See Lines 459 to 557
282     uint8_t numberNeededFilters = CAN_NumberOfNeededFilters(can_RxMsgs, &numberOfDifferentIDs[0], &retval);
283     numberNeededFilters--; /* Decrement by one because IS_CAN_FILTER_BANK_DUAL checks filter bank numbers starting
284     with 0 */
285     if (IS_CAN_FILTER_BANK_DUAL(numberNeededFilters)) {
286         uint8_t j = 0; /* Counts the number of initialized instances per case */
287         uint8_t posRxMsgs = 0; /* Iterator for can_RxMsgs[] */
288         uint8_t numberRegistersUsed = 0; /* Counts how many register space is already used in each filter bank (max.
289         64bit) */
290         uint8_t caseID = 0; /* indicates the actual filter mode that will be initialized */
291         True for hcan0
292         if ((ptrHcan->Instance == CAN2) {
293             /* Set start slave bank filter */
294             sFilterConfig.FilterBank = filterNumber;
295         }
296         for (caseID = 0; caseID < 2u; caseID++) {
297             /* ID List mode 16bit routed on FIFO0 or FIFO1 */
298
299             if (numberOfDifferentIDs[caseID] > 0U) {
300                 j = 0;
301                 while (j < numberOfDifferentIDs[caseID]) {
302                     /* Until all IDs in that filter case are treated */
303                     See Lines 569 to 599
304                     posRxMsgs = CAN_GetNextID(can_RxMsgs, numberOfRxMsgs, posRxMsgs, caseID); /* Get array position
305                     of next ID */
306
307                     switch (numberRegistersUsed) {
308                         case 0: /* 1st ID per filter bank */
309                             sFilterConfig.FilterIdHigh = ((can_RxMsgs[posRxMsgs].ID << 5)
310 | can_RxMsgs[posRxMsgs].RTR << 4);

```

mcu-hal\STM32F4xx_HAL_Driver\Inc\stm32f4xx_hal_can.h:
735: #define IS_CAN_FILTER_BANK_DUAL(BANK) ((BANK) <= 27U)

stm32f4xx_hal_can.h C:\bms\phxbms\embedded-software\mcu-hal\STM32F4xx_HAL_Driver\Inc - Definitions (1)

```

#define IS_CAN_PRESCALER(PRESCALER) (((PRESCALER) >= 1U) && ((PRESCALER) <= 1024U))
#define IS_CAN_FILTER_ID_HALFWORD(HALFWORD) ((HALFWORD) <= 0xFFFFU)
#define IS_CAN_FILTER_BANK_DUAL(BANK) ((BANK) <= 27U)
#define IS_CAN_FILTER_BANK_SINGLE(BANK) ((BANK) <= 13U)

```

Dual filter bank—28 filter banks.

```

308         j++;
309         break;
310
311     case 1: /* 2nd ID */
312         sFilterConfig.FilterIdLow = ((can_RxMsgs[posRxMsgs].ID << 5)
313             | can_RxMsgs[posRxMsgs].RTR << 4);
314         j++;
315         break;
316
317     case 2: /* 3rd ID */
318         sFilterConfig.FilterMaskIdHigh = ((can_RxMsgs[posRxMsgs].ID << 5)
319             | can_RxMsgs[posRxMsgs].RTR << 4);
320         j++;
321         break;
322
323     case 3: /* 4th ID */
324         sFilterConfig.FilterMaskIdLow = ((can_RxMsgs[posRxMsgs].ID << 5)
325             | can_RxMsgs[posRxMsgs].RTR << 4);
326         j++;
327         break;
328     }
329     numberRegistersUsed = j % 4U; /* space for 4 IDs a 16 bit in one filter bank */
330     if ((numberRegistersUsed == 0 && j > 1U) || (j == numberOfDifferentIDs[caseID])) {
331         /* all registers in filter bank used OR no more IDs in that case */
332         sFilterConfig.FilterMode = CAN_FILTERMODE_IDLIST;
333         sFilterConfig.FilterScale = CAN_FILTERSCALE_16BIT;
334         if (caseID == ID_16BIT_FIFO0) {
335             sFilterConfig.FilterFIFOAssignment = CAN_FILTER_FIFO0;
336         } else if (caseID == ID_16BIT_FIFO1) {
337             sFilterConfig.FilterFIFOAssignment = CAN_FILTER_FIFO1;
338         }
339         sFilterConfig.FilterBank = filterNumber;
340         HAL_CAN_ConfigFilter(ptrHcan, &sFilterConfig); /* initialize filter bank */
341         filterNumber++; /* increment filter number */
342     }
343     posRxMsgs++; /* increment array position to find next valid ID */
344 }
345 posRxMsgs = 0; /* reset variables for next case */
346 numberRegistersUsed = 0;
347 }
348 }
349 for (caseID = 2; caseID < 6U; caseID++) {
350     /* ID List mode 32bit routed on FIFO0 or FIFO1; Mask mode 16bit routed on FIFO0 or FIFO1 */
351     j = 0;
352     if (numberOfDifferentIDs[caseID] > 0U) {
353         while (j < numberOfDifferentIDs[caseID]) {
354             /* Until all IDs in that filter case are treated */
355
356             posRxMsgs = CAN_GetNextID(can_RxMsgs, numberOfRxMsgs, posRxMsgs, caseID); /* Get array position
357             of next ID */
358             switch (numberRegistersUsed) {

```

```

359     case 0: /* first 32bit per filter bank */
360         if (caseID == ID_32BIT_FIFO0 || caseID == ID_32BIT_FIFO1) { /* list mode 32bit */
361             sFilterConfig.FilterIdHigh = ((can_RxMsgs[posRxMsgs].ID << 3) >> 16); /* 1 << 2 is
362             for setting IDE bit to receive extended identifiers */
363             sFilterConfig.FilterIdLow = (uint16_t)((can_RxMsgs[posRxMsgs].ID << 3) | 1 << 2
364             | can_RxMsgs[posRxMsgs].RTR << 1);
365         } else if (caseID == MSK_16BIT_FIFO0 || caseID == MSK_16BIT_FIFO1) { /* mask mode
366         16bit */
367             sFilterConfig.FilterIdHigh = ((can_RxMsgs[posRxMsgs].ID << 5)
368             | can_RxMsgs[posRxMsgs].RTR << 4);
369             sFilterConfig.FilterMaskIdHigh = can_RxMsgs[posRxMsgs].mask;
370             sFilterConfig.FilterIdLow = 0x0000; /* set second register to 0xFFFF, */
371             sFilterConfig.FilterMaskIdLow = 0xFFFF; /* otherwise all messages would be received */
372         }
373         j++;
374         break;
375     case 1: /* second 32bit per filter bank */
376         if (caseID == ID_32BIT_FIFO0 || caseID == ID_32BIT_FIFO1) { /* list mode 32bit */
377             sFilterConfig.FilterMaskIdHigh = ((can_RxMsgs[posRxMsgs].ID << 3) >> 16); /* 1 << 2
378             is for setting IDE bit to receive extended identifiers */
379             sFilterConfig.FilterMaskIdLow = (uint16_t)((can_RxMsgs[posRxMsgs].ID << 3) | 1 << 2
380             | can_RxMsgs[posRxMsgs].RTR << 1);
381         } else if (caseID == MSK_16BIT_FIFO0 || caseID == MSK_16BIT_FIFO1) { /* mask mode
382         16bit */
383             sFilterConfig.FilterIdLow = ((can_RxMsgs[posRxMsgs].ID << 5)
384             | can_RxMsgs[posRxMsgs].RTR << 4);
385             sFilterConfig.FilterMaskIdLow = can_RxMsgs[posRxMsgs].mask;
386         }
387         j++;
388         break;
389     }
390     numberRegistersUsed = j % 2; /* Space for two IDs a 32bit or two mask a 16bit */
391     if ((numberRegistersUsed == 0 && j > 1U) || (j == numberOfDifferentIDs[caseID])) {
392         /* all registers in filter bank used OR no more IDs in that case */
393         if (caseID == ID_32BIT_FIFO0 || caseID == ID_32BIT_FIFO1) {
394             sFilterConfig.FilterMode = CAN_FILTERMODE_IDLIST;
395             sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
396             if (caseID == ID_32BIT_FIFO0)
397                 sFilterConfig.FilterFIFOAssignment = CAN_FILTER_FIFO0;
398             else
399                 sFilterConfig.FilterFIFOAssignment = CAN_FILTER_FIFO1;
400         } else if (caseID == MSK_16BIT_FIFO0 || caseID == MSK_16BIT_FIFO1) {
401             sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
402             sFilterConfig.FilterScale = CAN_FILTERSCALE_16BIT;
403             if (caseID == MSK_16BIT_FIFO0)
404                 sFilterConfig.FilterFIFOAssignment = CAN_FILTER_FIFO0;
405             else
406                 sFilterConfig.FilterFIFOAssignment = CAN_FILTER_FIFO1;
407         }
408         sFilterConfig.FilterBank = filterNumber;
409         HAL_CAN_ConfigFilter(ptrHcan, &sFilterConfig); /* initialize filter bank */

```



```

407         filterNumber++;        /* increment filter number */
408     }
409     posRxMsgs++;        /* increment array position to find next valid ID */
410 }
411 posRxMsgs = 0;        /* reset variables for next case */
412 numberRegistersUsed = 0;
413 }
414 }
415 j = 0;
416 if (numberOfDifferentIDs[MSK_32BIT] > 0U) {
417     /* Mask mode 32bit */
418
419     while (j < numberOfDifferentIDs[MSK_32BIT]) { /* Get array position of next ID */
420         /* Until all IDs in that filter case are treated */
421         posRxMsgs = CAN_GetNextID(can_RxMsgs, numberOfRxMsgs, posRxMsgs, MSK_32BIT);
422
423         sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
424         sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
425         sFilterConfig.FilterIdHigh = ((can_RxMsgs[posRxMsgs].ID << 3) >> 16); /* 1 << 2 is for setting IDE
426         bit to receive extended identifiers */
427         sFilterConfig.FilterIdLow = (uint16_t)((can_RxMsgs[posRxMsgs].ID << 3) | 1 << 2
428         | can_RxMsgs[posRxMsgs].RTR << 1);
429         sFilterConfig.FilterMaskIdHigh = can_RxMsgs[posRxMsgs].mask >> 16;
430         sFilterConfig.FilterMaskIdLow = (uint16_t)(can_RxMsgs[posRxMsgs].mask);
431         sFilterConfig.FilterFIFOAssignment = can_RxMsgs[posRxMsgs].fifo;
432         sFilterConfig.FilterBank = filterNumber;
433         HAL_CAN_ConfigFilter(ptrHcan, &sFilterConfig);
434         filterNumber++;
435         posRxMsgs++;
436         j++;
437     }
438 } else {
439     /* Too many filterbanks needed! Check the value of CAN_NUMBER_OF_FILTERBANKS */
440     /* If correct, try to reduce the IDs through masks or optimize used filter bank space. */
441     /* Number of different filter cases can be evaluated in numberOfDifferentIDs[]. One */
442     /* filter bank can filter as many messages as followed: */
443     /* 4 IDs in list mode 16bit */
444     /* 2 IDs in list mode 32bit and mask mode 16bit */
445     /* 1 ID in 32bit mask mode */
446     retval |= STD_ERR_BIT_6;
447 }
448 return retval;
449 }
450
451 /**
452  * @brief Returns the number of filters that have to be initialized
453  *
454  * @param can_RxMsgs: pointer to receive message struct
455  * @param numberOfDifferentIDs: pointer to array, where to store the specific number of different IDs
456  *
457  * @retval number of needed filters

```

```

458  */
459  static uint8_t CAN_NumberOfNeededFilters(CAN_MSG_RX_TYPE_s* can_RxMsgs, uint8_t* numberOfDifferentIDs, uint32_t*
error) {
460      static uint8_t retVal = 0;          /* static so save the number of filters from CAN0 and add to the ones from CAN1 */
461      uint16_t can_rx_length = 0;
462
463      if (can_RxMsgs == &can0_RxMsgs[0]) {
464          can_rx_length = can_CAN0_rx_length;
465      } else if (can_RxMsgs == &can1_RxMsgs[0]) {
466          can_rx_length = can_CAN1_rx_length;
467      } else {
468          can_rx_length = 0;
469      }
470
471      for (uint8_t i = 0; i < can_rx_length; i++) {
472          #if (CAN0_BUFFER_BYPASS_NUMBER_OF_IDS > 0) && (CAN_USE_CAN_NODE0 == 1)
473              (This is defined as 1)
474              if (can_RxMsgs == &can0_RxMsgs[0]) {
475                  /* Set buffer bypass IDs link table */
476                  for (uint8_t k = 0; k < CAN0_BUFFER_BYPASS_NUMBER_OF_IDS; k++) {
477                      if (can_RxMsgs[i].ID == can0_bufferBypass_RxMsgs[k]) {
478                          /* bypass ID == ID in message receive struct */
479
480                          can0_fastLinkIndex[k] = i;    /* set for can_bufferBypass_RxMsgs[k] link to array index */
481                          break;
482                      }
483                  }
484              }
485          #endif /* (CAN0_BUFFER_BYPASS_NUMBER_OF_IDS > 0) && (CAN_USE_CAN_NODE0 == 1) */
486          #if (CAN1_BUFFER_BYPASS_NUMBER_OF_IDS > 0) && (CAN_USE_CAN_NODE1 == 1)
487
488              if (can_RxMsgs == &can1_RxMsgs[0]) {
489                  /* Set buffer bypass IDs link table */
490                  for (int k = 0; k < CAN1_BUFFER_BYPASS_NUMBER_OF_IDS; k++) {
491                      if (can1_RxMsgs[i].ID == can1_bufferBypass_RxMsgs[k]) {
492                          /* bypass ID == ID in message receive struct */
493
494                          can1_fastLinkIndex[k] = i;    /* set for can1_bufferBypass_RxMsgs[k] link to array index */
495                          break;
496                      }
497                  }
498              }
499          #endif /* (CAN1_BUFFER_BYPASS_NUMBER_OF_IDS > 0) && (CAN_USE_CAN_NODE1 == 1) */
500          if (can_RxMsgs[i].mask == 0 && IS_CAN_STDID(can_RxMsgs[i].ID)) {
501              /* ID List mode 16bit */
502
503              if (can_RxMsgs[i].fifo == CAN_FILTER_FIFO0) {
504                  numberOfDifferentIDs[ID_16BIT_FIFO0]++;
505              } else if (can_RxMsgs[i].fifo == CAN_FILTER_FIFO1) {
506                  numberOfDifferentIDs[ID_16BIT_FIFO1]++;
507              } else {
508                  /* Invalid FIFO selection; check can_RxMsgs[i].fifo value */

```

```

509         *error |= STD_ERR_BIT_2;
510     }
511 } else if ((can_RxMsgs[i].mask == 0) && IS_CAN_EXTID(can_RxMsgs[i].ID)) {
512     /* ID List mode 32bit */
513
514     if (can_RxMsgs[i].fifo == CAN_FILTER_FIFO0) {
515         numberOfDifferentIDs[ID_32BIT_FIFO0]++;
516     } else if (can_RxMsgs[i].fifo == CAN_FILTER_FIFO1) {
517         numberOfDifferentIDs[ID_32BIT_FIFO1]++;
518     } else {
519         /* Invalid FIFO selection; check can_RxMsgs[i].fifo value */
520         *error |= STD_ERR_BIT_3;
521     }
522 } else if ((can_RxMsgs[i].mask > 0) && IS_CAN_STDID(can_RxMsgs[i].ID)) {
523     /* Mask mode 16bit */
524
525     if (can_RxMsgs[i].fifo == CAN_FILTER_FIFO0) {
526         numberOfDifferentIDs[MSK_16BIT_FIFO0]++;
527     } else if (can_RxMsgs[i].fifo == CAN_FILTER_FIFO1) {
528         numberOfDifferentIDs[MSK_16BIT_FIFO1]++;
529     } else {
530         /* Invalid FIFO selection; check can_RxMsgs[i].fifo value */
531         *error |= STD_ERR_BIT_4;
532     }
533 } else if ((can_RxMsgs[i].mask > 0U) && IS_CAN_EXTID(can_RxMsgs[i].ID)) {
534     /* Mask mode 32bit */
535
536     numberOfDifferentIDs[MSK_32BIT]++;
537 } else {
538     /* Invalid ID > IS_CAN_EXTID; check can_RxMsgs[i].ID value */
539     *error |= STD_ERR_BIT_5;
540     break;
541 }
542 }
543 for (uint8_t i = 0; i < 2U; i++) {
544     if (numberOfDifferentIDs[i] > 0U) {
545         retVal += (numberOfDifferentIDs[i] + 2) / 4; /* 4 IDs per filter; rounding up */
546     }
547 }
548 for (uint8_t i = 2; i < 6U; i++) {
549     if (numberOfDifferentIDs[i] > 0U) {
550         retVal += (numberOfDifferentIDs[i] + 1) / 2; /* 2 IDs per filter; rounding up */
551     }
552 }
553 if (numberOfDifferentIDs[MSK_32BIT] > 0U) {
554     retVal += numberOfDifferentIDs[6]; /* 1 ID per filter */
555 }
556 return retVal;
557 }
558
559 /**
560  * @brief Returns the next index of wished filter ID setting in CAN_MSG_RX_TYPE_t can_RxMsgs[CAN_NUMBER_OF_RX_IDS]

```

can_cfg.c	C can.c	X	C can_cfg.h	C stm32f4
cu-common > src > driver > can > C can.c > MSK_16BIT_FIFO0				
59	/*===== Macros and Definitions =====			
60	#define ID_16BIT_FIFO0		(0U)	
61	#define ID_16BIT_FIFO1		(1U)	
62	#define ID_32BIT_FIFO0		(2U)	
63	#define ID_32BIT_FIFO1		(3U)	
64	#define MSK_16BIT_FIFO0		(4U)	You,
65	#define MSK_16BIT_FIFO1		(5U)	
66	#define MSK_32BIT		(6U)	

```

561  *
562  * @param can_RxMsgs: pointer to receive message struct
563  * @param numberOfRxIDs: count of that type of receive message in can_RxMsgs struct
564  * @param startIndex: index where to start searching
565  * @param filterCase: specifies the object what will be found
566  *
567  * @retval returns index
568  */
569  static uint8_t CAN_GetNextID(CAN_MSG_RX_TYPE_s* can_RxMsgs, uint8_t numberOfRxIDs, uint8_t startIndex,
570                              uint8_t filterCase) {
571      uint8_t retVal = 0;
572      uint8_t i = startIndex;
573      while (i < numberOfRxIDs) {
574          if ((filterCase == ID_16BIT_FIFO0 && can_RxMsgs[i].mask == 0U) && IS_CAN_STDID(can_RxMsgs[i].ID) &&
575              (can_RxMsgs[i].fifo == CAN_FILTER_FIFO0)) {
576              retVal = i;
577              break;
578          } else if ((filterCase == ID_16BIT_FIFO1 && (can_RxMsgs[i].mask == 0U) && IS_CAN_STDID(can_RxMsgs[i].ID) &&
579              (can_RxMsgs[i].fifo == CAN_FILTER_FIFO1)) {
580              retVal = i;
581              break;
582          } else if ((filterCase == ID_32BIT_FIFO0 && (can_RxMsgs[i].mask == 0U) && !IS_CAN_STDID(can_RxMsgs[i].ID) &&
583              IS_CAN_EXTID(can_RxMsgs[i].ID) && (can_RxMsgs[i].fifo == CAN_FILTER_FIFO0)) {
584              retVal = i;
585              break;
586          } else if ((filterCase == ID_32BIT_FIFO1 && (can_RxMsgs[i].mask == 0U) && !IS_CAN_STDID(can_RxMsgs[i].ID) &&
587              IS_CAN_EXTID(can_RxMsgs[i].ID) && (can_RxMsgs[i].fifo == CAN_FILTER_FIFO1)) {
588              retVal = i;
589              break;
590          } else if ((filterCase == MSK_16BIT_FIFO0 && (can_RxMsgs[i].mask > 0U) && IS_CAN_STDID(can_RxMsgs[i].ID) &&
591              (can_RxMsgs[i].fifo == CAN_FILTER_FIFO0)) {
592              retVal = i;
593              break;
594          } else if ((filterCase == MSK_16BIT_FIFO1 && (can_RxMsgs[i].mask > 0U) && IS_CAN_STDID(can_RxMsgs[i].ID) &&
595              (can_RxMsgs[i].fifo == CAN_FILTER_FIFO1)) {
596              retVal = i;
597              break;
598          } else if ((filterCase == MSK_32BIT && (can_RxMsgs[i].mask > 0U) && !IS_CAN_STDID(can_RxMsgs[i].ID) &&
599              IS_CAN_EXTID(can_RxMsgs[i].ID)) {
600              retVal = i;
601              break;
602          }
603          i++;
604      }
605      return retVal;
606  }
607
608  /* *****
609  * Interrupt handling
610  ***** */
611
612  /**

```

```

606     * @brief   Transmission Mailbox 0 complete callback.
607     * @param   hcan pointer to a CAN_HandleTypeDef structure that contains
608     *           the configuration information for the specified CAN.
609     * @retval  None
610     */
611 void HAL_CAN_TxMailbox0CompleteCallback(CAN_HandleTypeDef *hcan) {
612 #if CAN0_USE_TX_BUFFER
613     if (hcan->Instance == CAN2) {
614         CAN_TxCpltCallback(CAN_NODE0);
615     }
616 #endif /* CAN0_USE_TX_BUFFER */
617 #if CAN1_USE_TX_BUFFER
618     /* No need for callback, if no buffer is used */
619     if (hcan->Instance == CAN1) {
620         /* Transmission complete callback */
621         CAN_TxCpltCallback(CAN_NODE1);
622     }
623 #endif /* CAN1_USE_TX_BUFFER */
624 }
625
626 /**
627  * @brief   Transmission Mailbox 1 complete callback.
628  * @param   hcan pointer to a CAN_HandleTypeDef structure that contains
629  *           the configuration information for the specified CAN.
630  * @retval  None
631  */
632 void HAL_CAN_TxMailbox1CompleteCallback(CAN_HandleTypeDef *hcan) {
633 #if CAN0_USE_TX_BUFFER
634     if (hcan->Instance == CAN2) {
635         CAN_TxCpltCallback(CAN_NODE0);
636     }
637 #endif /* CAN0_USE_TX_BUFFER */
638 #if CAN1_USE_TX_BUFFER
639     /* No need for callback, if no buffer is used */
640     if (hcan->Instance == CAN1) {
641         /* Transmission complete callback */
642         CAN_TxCpltCallback(CAN_NODE1);
643     }
644 #endif /* CAN1_USE_TX_BUFFER */
645 }
646
647 /**
648  * @brief   Transmission Mailbox 2 complete callback.
649  * @param   hcan pointer to a CAN_HandleTypeDef structure that contains
650  *           the configuration information for the specified CAN.
651  * @retval  None
652  */
653 void HAL_CAN_TxMailbox2CompleteCallback(CAN_HandleTypeDef *hcan) {
654 #if CAN0_USE_TX_BUFFER
655     if (hcan->Instance == CAN2) {
656         CAN_TxCpltCallback(CAN_NODE0);
657     }

```

The next three TxMailboxnCompleteCallback functions can be wrappers of the same common function.

```

658 #endif /* CAN0_USE_TX_BUFFER */
659 #if CAN1_USE_TX_BUFFER
660     /* No need for callback, if no buffer is used */
661     if (hcan->Instance == CAN1) {
662         /* Transmission complete callback */
663         CAN_TxCpltCallback(CAN_NODE1);
664     }
665 #endif /* CAN1_USE_TX_BUFFER */
666 }
667
668 /**
669  * @brief Rx FIFO 0 message pending callback.
670  * @param hcan pointer to a CAN_HandleTypeDef structure that contains
671  *         the configuration information for the specified CAN.
672  * @retval None
673  */
674 void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan) {
675     /* Call callback function to interpret or save RX message in buffer */
676     if (hcan->Instance == CAN2) {
677         CAN_RxMsg(CAN_NODE0, hcan, CAN_RX_FIFO0);          /* change towards HAL_CAN_IRQHandler */
678     }
679     if (hcan->Instance == CAN1) {
680         CAN_RxMsg(CAN_NODE1, hcan, CAN_RX_FIFO0);          /* change towards HAL_CAN_IRQHandler */
681     }
682 }
683
684 /**
685  * @brief Rx FIFO 1 message pending callback.
686  * @param hcan pointer to a CAN_HandleTypeDef structure that contains
687  *         the configuration information for the specified CAN.
688  * @retval None
689  */
690 void HAL_CAN_RxFifo1MsgPendingCallback(CAN_HandleTypeDef *hcan) {
691     /* Call callback function to interpret or save RX message in buffer */
692     if (hcan->Instance == CAN2) {
693         CAN_RxMsg(CAN_NODE0, hcan, CAN_RX_FIFO1);          /* change towards HAL_CAN_IRQHandler */
694     }
695     if (hcan->Instance == CAN1) {
696         CAN_RxMsg(CAN_NODE1, hcan, CAN_RX_FIFO1);          /* change towards HAL_CAN_IRQHandler */
697     }
698 }
699
700 /**
701  * @brief Error CAN callback.
702  * @param hcan pointer to a CAN_HandleTypeDef structure that contains
703  *         the configuration information for the specified CAN.
704  * @retval None
705  */
706 void HAL_CAN_ErrorCallback(CAN_HandleTypeDef *hcan) {
707     CAN_ERROR_s* errorStruct = NULL_PTR;
708
709     if (hcan->Instance == CAN1) {

```

```

710 #if CAN_USE_CAN_NODE1 == 1
711     errorStruct = &CAN1_errorStruct;
712 #endif /* CAN_USE_CAN_NODE1 == 1 */
713 } else {
714 #if CAN_USE_CAN_NODE0 == 1
715     errorStruct = &CAN0_errorStruct;
716 #endif /* CAN_USE_CAN_NODE0 == 1 */
717 }
718
719 /* Check Error Warning flag and set error codes */
720 if (errorStruct != NULL_PTR) {
721     /* Check Error Warning Flag */
722     if ((hcan->ErrorCode & HAL_CAN_ERROR_EWG) != 0) {
723         /* This bit is set by hardware when the warning limit has been
724          * reached (Receive Error Counter or Transmit Error Counter>=96
725          * until error counter 127 write error frames dominant on can bus
726          * increment error occurrence of error warning state */
727         errorStruct->canErrorCounter[0]++;
728     }
729
730     /* Check Error Passive Flag */
731     if ((hcan->ErrorCode & HAL_CAN_ERROR_EPV) != 0) {
732         /* This bit is set by hardware when the Error Passive limit has
733          * been reached (Receive Error Counter or Transmit Error Counter
734          * > 127) write error frames recessive on can bus increment error
735          * occurrence of error passive state */
736         errorStruct->canErrorCounter[1]++;
737     }
738
739     /* Check Bus-Off Flag */
740     if ((hcan->ErrorCode & HAL_CAN_ERROR_BOF) != 0) {
741         /* This bit is set by hardware when it enters the bus-off state.
742          * The bus-off state is entered on TEC overflow, greater than 255
743          * increment error occurrence of bus-off state */
744         errorStruct->canErrorCounter[2]++;
745     }
746
747     /* Check stuff error flag */
748     if ((hcan->ErrorCode & HAL_CAN_ERROR_STF) != 0) {
749         /* When five consecutive bits of the same level have been
750          * transmitted by a node, it will add a sixth bit of the opposite
751          * level to the outgoing bit stream. The receivers will remove this
752          * extra bit. This is done to avoid excessive DC components on the
753          * bus, but it also gives the receivers an extra opportunity to
754          * detect errors: if more than five consecutive bits of the same
755          * level occurs on the bus, a Stuff Error is signaled. */
756         errorStruct->canErrorCounter[3]++;
757     }
758
759     if ((hcan->ErrorCode & HAL_CAN_ERROR_FOR) != 0) {
760         /* FORM ERROR --- Some parts of the CAN message have a fixed format,
761          * i.e. the standard defines exactly what levels must occur and
762          * when. (Those parts are the CRC Delimiter, ACK Delimiter, End of
763          * Frame, and also the Intermission, but there are some extra
764          * special error checking rules for that.) If a CAN controller

```

```

762         * detects an invalid value in one of these fixed fields, a Form
763         * Error is signaled. */
764     errorStruct->canErrorCounter[4]++;
765 }
766 if ((hcan->ErrorCode & HAL_CAN_ERROR_ACK) != 0) {
767     /* ACKNOWLEDGMENT ERROR --- All nodes on the bus that correctly
768     * receives a message (regardless of their being interested of its
769     * contents or not) are expected to send a dominant level in the
770     * so-called Acknowledgement Slot in the message. The transmitter
771     * will transmit a recessive level here. If the transmitter can
772     * detect a dominant level in the ACK slot, an Acknowledgement
773     * Error is signaled. */
774     errorStruct->canErrorCounter[5]++;
775 }
776 if ((hcan->ErrorCode & HAL_CAN_ERROR_BR) != 0) {
777     /* BIT RECESSIVE ERROR --- Each transmitter on the CAN bus monitors
778     * (i.e. reads back) the transmitted signal level. If the bit level
779     * actually read differs from the one transmitted, a Bit Error (No
780     * bit error is raised during the arbitration process.) */
781     errorStruct->canErrorCounter[6]++;
782 }
783 if ((hcan->ErrorCode & HAL_CAN_ERROR_BD) != 0) {
784     /* BIT DOMINANT ERROR --- Each transmitter on the CAN bus monitors
785     * (i.e. reads back) the transmitted signal level. If the bit level
786     * actually read differs from the one transmitted, a Bit Error (No
787     * bit error is raised during the arbitration process.) */
788     errorStruct->canErrorCounter[7]++;
789 }
790 if ((hcan->ErrorCode & HAL_CAN_ERROR_CRC) != 0) {
791     /* CRC ERROR --- Each message features a 15-bit Cyclic Redundancy
792     * Checksum (CRC), and any node that detects a different CRC in the
793     * message than what it has calculated itself will signal an CRC
794     * Error. */
795     errorStruct->canErrorCounter[8]++;
796 }
797 if ((hcan->ErrorCode & HAL_CAN_ERROR_RX_FOV0) != 0) {
798     /* Rx FIFO0 overrun error */
799     errorStruct->canErrorCounter[9]++;
800 }
801 if ((hcan->ErrorCode & HAL_CAN_ERROR_RX_FOV1) != 0) {
802     /* Rx FIFO1 overrun error */
803     errorStruct->canErrorCounter[10]++;
804 }
805 if ((hcan->ErrorCode & HAL_CAN_ERROR_TX_ALST0) != 0) {
806     /* TxMailbox 0 transmit failure due to arbitration lost */
807     errorStruct->canErrorCounter[11]++;
808 }
809 if ((hcan->ErrorCode & HAL_CAN_ERROR_TX_TERR0) != 0) {
810     /* TxMailbox 1 transmit failure due to transmit error */
811     errorStruct->canErrorCounter[12]++;
812 }
813 if ((hcan->ErrorCode & HAL_CAN_ERROR_TX_ALST1) != 0) {

```



```

814         /* TxMailbox 0 transmit failure due to arbitration lost */
815         errorStruct->canErrorCounter[13]++;
816     }
817     if ((hcan->ErrorCode & HAL_CAN_ERROR_TX_TERR1) != 0) {
818         /* TxMailbox 1 transmit failure due to transmit error */
819         errorStruct->canErrorCounter[14]++;
820     }
821     if ((hcan->ErrorCode & HAL_CAN_ERROR_TX_ALST2) != 0) {
822         /* TxMailbox 0 transmit failure due to arbitration lost */
823         errorStruct->canErrorCounter[15]++;
824     }
825     if ((hcan->ErrorCode & HAL_CAN_ERROR_TX_TERR2) != 0) {
826         /* TxMailbox 1 transmit failure due to transmit error */
827         errorStruct->canErrorCounter[16]++;
828     }
829     if ((hcan->ErrorCode & HAL_CAN_ERROR_TIMEOUT) != 0) {
830         /* Timeout error */
831         errorStruct->canErrorCounter[17]++;
832     }
833     if ((hcan->ErrorCode & HAL_CAN_ERROR_NOT_INITIALIZED) != 0) {
834         /* Peripheral not initialized */
835         errorStruct->canErrorCounter[18]++;
836     }
837     if ((hcan->ErrorCode & HAL_CAN_ERROR_NOT_READY) != 0) {
838         /* Peripheral not ready */
839         errorStruct->canErrorCounter[19]++;
840     }
841     if ((hcan->ErrorCode & HAL_CAN_ERROR_NOT_STARTED) != 0) {
842         /* Peripheral not started */
843         errorStruct->canErrorCounter[20]++;
844     }
845     if ((hcan->ErrorCode & HAL_CAN_ERROR_PARAM) != 0) {
846         /* Parameter error */
847         errorStruct->canErrorCounter[21]++;
848     }
849 }
850 }
851
852 /**
853  * @brief Transmission complete callback in non blocking mode
854  *
855  * @param canNode: canNode that transmitted a message
856  *
857  * @retval none (void)
858  */
859 static void CAN_TxCpltCallback(CAN_NodeTypeDef_e canNode) {
860     STD_RETURN_TYPE_e retVal = E_NOT_OK;
861     CAN_TX_BUFFER_s* can_txbuffer = NULL;
862
863     if (canNode == CAN_NODE0) {
864 #if CAN_USE_CAN_NODE0 == 1
865         can_txbuffer = &can0_txbuffer;

```

Lines 863 to 871 can be written in a function to hide the details. This is also good for code reuse.

```

866 #endif /* #if CAN_USE_CAN_NODE0 == 1 */
867 } else if (canNode == CAN_NODE1) {
868 #if CAN_USE_CAN_NODE1 == 1
869     can_txbuffer = &can1_txbuffer;
870 #endif /* CAN_USE_CAN_NODE1 == 1 */
871 }
872 /* Transmit buffer existing, check if message is ready for transmission */
873 if (can_txbuffer != NULL) {
874     /* No Error during start of transmission */
875     if ((can_txbuffer->ptrWrite == can_txbuffer->ptrRead)
876         && (can_txbuffer->buffer[can_txbuffer->ptrRead].newMsg == 0)) {
877         /* nothing to transmit, buffer is empty */
878         retVal = E_NOT_OK;
879     } else {
880         retVal = CAN_TxMsgBuffer(canNode);
881         if (retVal != E_OK) {
882             /* Error during transmission, retransmit message later */
883             retVal = E_NOT_OK;
884         }
885     }
886 } else {
887     /* no transmit buffer active */
888     retVal = E_NOT_OK;
889 }
890 }
891
892
893
894 /* *****
895  * Transmit message
896  * *****/
897
898 STD_RETURN_TYPE_e CAN_TxMsg(CAN_NodeTypeDef_e canNode, uint32_t msgID, uint8_t* ptrMsgData, uint32_t msgLength,
899     uint32_t RTR) {
900     STD_RETURN_TYPE_e retVal = E_NOT_OK;
901     CAN_TxHeaderTypeDef canMessage;
902     CAN_HandleTypeDef *ptrHcan;
903     uint32_t freeMailboxes = 0;
904     uint32_t *ptrMailbox = NULL_PTR;
905
906     if (canNode == CAN_NODE0) {
907         if (canNode0_listenonly_mode) {
908             ptrHcan = NULL;
909         } else {
910             /* Check if at least one mailbox is free */
911             freeMailboxes = HAL_CAN_GetTxMailboxesFreeLevel(&hcan0);
912             if (freeMailboxes != 0) {
913                 ptrHcan = &hcan0;
914             } else {
915                 ptrHcan = NULL;
916             }
917         }

```

```

918 } else if (canNode == CAN_NODE1) {
919     if (canNode1_listenonly_mode) {
920         ptrHcan = NULL;
921     } else {
922         /* Check if at least one mailbox is free */
923         freeMailboxes = HAL_CAN_GetTxMailboxesFreeLevel(&hcan1);
924         if (freeMailboxes != 0) {
925             ptrHcan = &hcan1;
926         } else {
927             ptrHcan = NULL;
928         }
929     }
930 } else {
931     ptrHcan = NULL;
932 }
933
934 if ((IS_CAN_STDID(msgID) || IS_CAN_EXTID(msgID)) && IS_CAN_DLC(msgLength) && ptrHcan != NULL) {
935     if (IS_CAN_STDID(msgID)) {
936         canMessage.StdId = msgID;
937         canMessage.IDE = CAN_ID_STD;
938     } else {
939         canMessage.ExtId = msgID;
940         canMessage.IDE = CAN_ID_EXT;
941     }
942     canMessage.DLC = msgLength;
943     canMessage.RTR = RTR;
944     canMessage.TransmitGlobalTime = DISABLE;
945
946     /* Copy message in TX mailbox and transmit it */
947     HAL_CAN_AddTxMessage(ptrHcan, &canMessage, ptrMsgData, ptrMailbox);
948 } else {
949     retVal = E_NOT_OK;
950 }
951 return retVal;
952 }

```

A better way is to write another wrapper function to hide ptrMailbox, which is not assigned a specific value.

```

953
954         Send to buffer
955 STD_RETURN_TYPE_e CAN_Send(CAN_NodeTypeDef_e canNode, uint32_t msgID, uint8_t* ptrMsgData, uint32_t msgLength,
956     uint32_t RTR) {
957     STD_RETURN_TYPE_e retVal = E_NOT_OK;
958     uint8_t tmptxbuffer_wr = 0;
959     CAN_TX_BUFFER_s* can_txbuffer = NULL_PTR;
960
961     if (canNode == CAN_NODE0) {
962 #if CAN_USE_CAN_NODE0 == 1
963         can_txbuffer = &can0_txbuffer;
964 #endif /* #if CAN_USE_CAN_NODE0 == 1 */
965     } else if (canNode == CAN_NODE1) {
966 #if CAN_USE_CAN_NODE1 == 1
967         can_txbuffer = &can1_txbuffer;
968 #endif /* #if CAN_USE_CAN_NODE1 == 1 */
969     }

```

```

970
971  /* Transmit buffer exisiting */
972  if (can_txbuffer != NULL_PTR) {
973      tmptxbuffer_wr = can_txbuffer->ptrWrite;
974
975      if (tmptxbuffer_wr == can_txbuffer->ptrRead) {
976          if (can_txbuffer->buffer[tmptxbuffer_wr].newMsg == 0) {
977              /* free buffer space for message */
978
979              can_txbuffer->ptrWrite++;
980              can_txbuffer->ptrWrite = can_txbuffer->ptrWrite % can_txbuffer->length;
981              retVal = E_OK;
982          } else {
983              /* buffer full */
984              retVal = E_NOT_OK;
985          }
986      } else {
987          can_txbuffer->ptrWrite++;
988          can_txbuffer->ptrWrite = can_txbuffer->ptrWrite % can_txbuffer->length;
989          retVal = E_OK;
990      }
991  } else {
992      retVal = E_NOT_OK;
993  }
994
995  if ((can_txbuffer != NULL_PTR) &&
996      (retVal == E_OK) &&
997      (IS_CAN_STDID(msgID) || IS_CAN_EXTID(msgID)) &&
998      (IS_CAN_DLC(msgLength))) {
999      /* if buffer free and valid CAN identifier */
1000
1001      if (IS_CAN_STDID(msgID)) {
1002          can_txbuffer->buffer[tmptxbuffer_wr].msg.StdId = msgID;
1003          can_txbuffer->buffer[tmptxbuffer_wr].msg.IDE = 0;
1004      } else {
1005          can_txbuffer->buffer[tmptxbuffer_wr].msg.ExtId = msgID;
1006          can_txbuffer->buffer[tmptxbuffer_wr].msg.IDE = 1;
1007      }
1008      can_txbuffer->buffer[tmptxbuffer_wr].newMsg = 1;
1009      can_txbuffer->buffer[tmptxbuffer_wr].msg.RTR = RTR;
1010      can_txbuffer->buffer[tmptxbuffer_wr].msg.DLC = msgLength; /* Data length of the frame that will be
transmitted */
1011      can_txbuffer->buffer[tmptxbuffer_wr].msg.TransmitGlobalTime = DISABLE;
1012
1013      /* copy message data in handle transmit structure */
1014      can_txbuffer->buffer[tmptxbuffer_wr].data[0] = ptrMsgData[0];
1015      can_txbuffer->buffer[tmptxbuffer_wr].data[1] = ptrMsgData[1];
1016      can_txbuffer->buffer[tmptxbuffer_wr].data[2] = ptrMsgData[2];
1017      can_txbuffer->buffer[tmptxbuffer_wr].data[3] = ptrMsgData[3];
1018      can_txbuffer->buffer[tmptxbuffer_wr].data[4] = ptrMsgData[4];
1019      can_txbuffer->buffer[tmptxbuffer_wr].data[5] = ptrMsgData[5];
1020      can_txbuffer->buffer[tmptxbuffer_wr].data[6] = ptrMsgData[6];

```

A for loop works better

```

1021         can_txbuffer->buffer[tmp_txbuffer_wr].data[7] = ptrMsgData[7];
1022
1023         retVal = E_OK;
1024     } else {
1025         retVal = E_NOT_OK;
1026     }
1027
1028     return retVal;
1029 }
1030
1031
1032 STD_RETURN_TYPE_e CAN_TxMsgBuffer(CAN_NodeTypeDef_e canNode) {
1033     STD_RETURN_TYPE_e retVal = E_NOT_OK;
1034     HAL_StatusTypeDef retHal = HAL_ERROR;
1035     CAN_TX_BUFFER_s* can_txbuffer = NULL;
1036     CAN_HandleTypeDef* ptrHcan = NULL;
1037     uint32_t freeMailboxes = 0;
1038     uint32_t *ptrMailbox = NULL_PTR;
1039
1040     if (canNode == CAN_NODE0) {
1041 #if CAN_USE_CAN_NODE0 == 1
1042         if (!canNode0_listenonly_mode) {
1043             can_txbuffer = &can0_txbuffer;
1044             ptrHcan = &hcan0;
1045         }
1046 #endif /* #if CAN_USE_CAN_NODE0 == 1 */
1047     } else if (canNode == CAN_NODE1) {
1048 #if CAN_USE_CAN_NODE1 == 1
1049         if (!canNode1_listenonly_mode) {
1050             can_txbuffer = &can1_txbuffer;
1051             ptrHcan = &hcan1;
1052         }
1053 #endif /* CAN_USE_CAN_NODE1 == 1 */
1054     }
1055     /* Check if at least one mailbox is free */
1056     freeMailboxes = HAL_CAN_GetTxMailboxesFreeLevel(&hcan0);
1057
1058     if ((can_txbuffer != NULL) && (freeMailboxes != 0)) {
1059         if ((can_txbuffer->ptrWrite == can_txbuffer->ptrRead)
1060             && (can_txbuffer->buffer[can_txbuffer->ptrRead].newMsg == 0)) {
1061             /* nothing to transmit, buffer is empty */
1062             retVal = E_NOT_OK;
1063         } else {
1064             /* Copy message into TX mailbox and transmit it */
1065             retHal = HAL_CAN_AddTxMessage(ptrHcan, &can_txbuffer->buffer[can_txbuffer->ptrRead].msg,
1066                 can_txbuffer->buffer[can_txbuffer->ptrRead].data, ptrMailbox);
1067
1068             if (retHal == HAL_OK) {
1069                 /* No Error during start of transmission */
1070                 can_txbuffer->buffer[can_txbuffer->ptrRead].newMsg = 0;    /* Msg is sent, set newMsg to 0, to allow
writing of new data in buffer space */
1071                 can_txbuffer->ptrRead++;

```

```

1071         can_txbuffer->ptrRead = can_txbuffer->ptrRead % can_txbuffer->length;
1072         retVal = E_OK;
1073     } else {
1074         retVal = E_NOT_OK;          /* Error during transmission, retransmit message later */
1075     }
1076 }
1077 } else {
1078     /* no transmit buffer active or TX mailboxes full */
1079     retVal = E_NOT_OK;
1080 }
1081 return retVal;
1082 }

```

```

1084 /* *****
1085  * Receive message
1086  ***** */

```

```

1088 /**
1089  * @brief  Receives CAN messages and stores them either in RxBuffer or in hcan
1090  *
1091  * @param  canNode:    canNode which received the message
1092  * @param  ptrHcan:    pointer to a CAN_HandleTypeDef structure that contains
1093  *                    the message information of the specified CAN.
1094  * @param  FIFONumber: FIFO in which the message has been received
1095  * @retval none (void)
1096  */

```

```

1097 static void CAN_RxMsg(CAN_NodeTypeDef_e canNode, CAN_HandleTypeDef* ptrHcan, uint8_t FIFONumber) {

```

```

1098     uint8_t bypassLinkIndex = 0;

```

```

1099     CAN_RX_BUFFERELEMENT_s tmpMsgBuffer;

```

```

1100     uint32_t msgID = 0;

```

```

1101
1102 #if CAN0_USE_RX_BUFFER || CAN1_USE_RX_BUFFER
1103     uint32_t* can_bufferbypass_rxmsgs = NULL;

```

```

1104     uint32_t bufferbypasslength = 0;

```

```

1105     CAN_RX_BUFFER_s* can_rxbuffer = NULL;

```

```

1106     CAN_MSG_RX_TYPE_s* can_rxmsgs = NULL;

```

```

1107     uint8_t* can_fastLinkIndex = NULL;

```

```

1108 #endif /* CAN0_USE_RX_BUFFER || CAN1_USE_RX_BUFFER */

```

```

1109
1110     /* Set pointer on respective RxBuffer */

```

```

1111     if (canNode == CAN_NODE1) {

```

```

1112 #if CAN1_USE_RX_BUFFER && CAN_USE_CAN_NODE1 == 1

```

```

1113     can_rxbuffer = &can1_rxbuffer;

```

```

1114 #if (CAN1_BUFFER_BYPASS_NUMBER_OF_IDS > 0u)

```

```

1115     can_rxmsgs = &can1_RxMsgs[0];

```

```

1116     can_bufferbypass_rxmsgs = &can1_bufferBypass_RxMsgs[0];

```

```

1117     bufferbypasslength = CAN1_BUFFER_BYPASS_NUMBER_OF_IDS;

```

```

1118     can_fastLinkIndex = &can1_fastLinkIndex[0];

```

```

1119 #endif /* (CAN1_BUFFER_BYPASS_NUMBER_OF_IDS > 0u) */

```

```

1120 #endif /* CAN1_USE_RX_BUFFER && CAN_USE_CAN_NODE1 == 1 */

```

```

1121     } else if (canNode == CAN_NODE0) {

```

```

1122 #if CAN0_USE_RX_BUFFER && CAN_USE_CAN_NODE0 == 1

```

```

C can_cfg.c  C can.h  X  C can.c  C can_cfg.h
mcu-common > src > driver > can > C can.h > CAN_RX_BUFFER_s
100 typedef struct CAN_RX_BUFFERELEMENT {
101     CAN_RxHeaderTypeDef msg;
102     uint8_t data[8];
103     uint8_t newMsg;
104 } CAN_RX_BUFFERELEMENT_s;

```

```

C can_cfg.c  C can.h  X  C can.c  C can_cfg.h
mcu-common > src > driver > can > C can.h > CAN_RX_BUFFER_s
112 typedef struct CAN_RX_BUFFER {
113     uint8_t ptrRead;
114     uint8_t ptrWrite;
115     uint8_t length;
116     CAN_RX_BUFFERELEMENT_s* buffer;
117 } CAN_RX_BUFFER_s;

```

```

C cansignal_cfg.h  C can_cfg.h  X  C stm32f4xx_hal_can.h  C cansignal_cfg.c  C c
mcu-primary > src > driver > config > C can_cfg.h > CAN_MSG_RX_TYPE_s
297 typedef struct CAN_MSG_RX_TYPE {
298     uint32_t ID; /*!< message ID */
299     uint32_t mask; /*!< mask or 0x0000 to select list mode */
300     uint8_t DLC; /*!< data length */
301     uint8_t RTR; /*!< rtr bit */
302     uint32_t fifo; /*!< selected CAN hardware (CAN_FILTER_FIFO0 or CAN_F
303     STD_RETURN_TYPE_e (*func)(uint32_t ID, uint8_t*, uint8_t, uint8_t);
304 } CAN_MSG_RX_TYPE_s;

```

```

1123     can_rxbuffer = &can0_rxbuffer;
1124 #if (CAN0_BUFFER_BYPASS_NUMBER_OF_IDS > 0U)
1125     can_rxmsgs = &can0_RxMsgs[0];           See Line 175.
1126     can_bufferbypass_rxmsgs = &can0_bufferBypass_RxMsgs[0];
1127     bufferbypasslength = CAN0_BUFFER_BYPASS_NUMBER_OF_IDS;
1128     can_fastLinkIndex = &can0_fastLinkIndex[0]; See Line 90.
1129 #endif /* (CAN0_BUFFER_BYPASS_NUMBER_OF_IDS > 0u) */
1130 #endif /* CAN0_USE_RX_BUFFER && CAN_USE_CAN_NODE0 == 1 */
1131 }
1132
1133 /* Get message ID */
1134 HAL_CAN_GetRxMessage(ptrHcan, FIFONumber, &tmpMsgBuffer.msg , &tmpMsgBuffer.data[0]);
1135
1136 #if (CAN1_BUFFER_BYPASS_NUMBER_OF_IDS > 0) || (CAN0_BUFFER_BYPASS_NUMBER_OF_IDS > 0)
1137     if (can_bufferbypass_rxmsgs != NULL) {
1138         /* only needed when messages are bypassed */
1139
1140         for (bypassLinkIndex = 0; bypassLinkIndex < bufferbypasslength; bypassLinkIndex++) {
1141             if ((tmpMsgBuffer.msg.StdId == can_bufferbypass_rxmsgs[bypassLinkIndex]) ||
1142                 (tmpMsgBuffer.msg.ExtId == can_bufferbypass_rxmsgs[bypassLinkIndex])) {
1143                 break;           Only match the ID in the received message with the ID in the Buffer_Bypass array.
1144             }
1145         }
1146         /* No match found in the previous step, which means that we need to put the data from
1147            tmpMsgBuffer to the buffer. */
1148     }
1149 #endif /* #if (CAN1_BUFFER_BYPASS_NUMBER_OF_IDS > 0) || (CAN0_BUFFER_BYPASS_NUMBER_OF_IDS > 0) */
1150     if (bypassLinkIndex >= bufferbypasslength && can_rxbuffer != NULL) {
1151         /* ##### Use buffer / Copy data in buffer ##### */
1152
1153 #if CAN0_USE_RX_BUFFER || CAN1_USE_RX_BUFFER
1154         /* NO NEED TO DISABLE INTERRUPTS, BECAUSE FUNCTION IS CALLED FROM ISR */
1155
1156         /* Set to 1 to mark message as new received. Set to 0 when reading message from buffer */
1157         can_rxbuffer->buffer[can_rxbuffer->ptrWrite].newMsg = 1;
1158
1159         /* Get message header */
1160         Note that msg is a struct and we can copy a struct in C directly.
1161         can_rxbuffer->buffer[can_rxbuffer->ptrWrite].msg = tmpMsgBuffer.msg;
1162
1163         /* Get the data field */
1164         for (uint8_t i = 0; i < can_rxbuffer->buffer[can_rxbuffer->ptrWrite].msg.DLC; i++) {
1165             can_rxbuffer->buffer[can_rxbuffer->ptrWrite].data[i] = tmpMsgBuffer.data[i];
1166         }
1167
1168         /* Increment write pointer */
1169         can_rxbuffer->ptrWrite++;
1170         can_rxbuffer->ptrWrite = can_rxbuffer->ptrWrite % can_rxbuffer->length;
1171         Bypass the buffer
1172     }
1173 #endif /* CAN0_USE_RX_BUFFER || CAN1_USE_RX_BUFFER */
1174     } else if (bypassLinkIndex < bufferbypasslength && can_rxmsgs != NULL && can_fastLinkIndex != NULL) {
1175         /* ##### Buffer active but bypassed ##### */
1176
1177 #if ((CAN1_BUFFER_BYPASS_NUMBER_OF_IDS > 0) || (CAN0_BUFFER_BYPASS_NUMBER_OF_IDS > 0)) && ((CAN_USE_CAN_NODE0 == 1)
1178     || (CAN_USE_CAN_NODE1 == 1)) This #if is buggy.
1179         /* call buffer bypass callback function */

```

```

1174     if (tmpMsgBuffer.msg.IDE == 0U) {
1175         if (can_rxmsgs[can_fastLinkIndex[bypassLinkIndex]].func != NULL) {
1176             can_rxmsgs[can_fastLinkIndex[bypassLinkIndex]].func(tmpMsgBuffer.msg.StdId,
1177                 tmpMsgBuffer.data,
1178                 tmpMsgBuffer.msg.DLC,
1179                 tmpMsgBuffer.msg.RTR);
1180         } else {
1181             /* No callback function defined */
1182             CAN_BufferBypass(canNode, tmpMsgBuffer.msg.StdId, tmpMsgBuffer.data, tmpMsgBuffer.msg.DLC,
1183                 tmpMsgBuffer.msg.RTR); See Line 1295
1184         }
1185     } else {
1186         if (can_rxmsgs[can_fastLinkIndex[bypassLinkIndex]].func != NULL) {
1187             can_rxmsgs[can_fastLinkIndex[bypassLinkIndex]].func(tmpMsgBuffer.msg.ExtId,
1188                 tmpMsgBuffer.data,
1189                 tmpMsgBuffer.msg.DLC,
1190                 tmpMsgBuffer.msg.RTR);
1191         } else {
1192             /* No callback function defined */
1193             CAN_BufferBypass(canNode, tmpMsgBuffer.msg.ExtId, tmpMsgBuffer.data, tmpMsgBuffer.msg.DLC,
1194                 tmpMsgBuffer.msg.RTR);
1195         }
1196     }
1197 }
1198 #endif /* ((CAN1_BUFFER_BYPASS_NUMBER_OF_IDS > 0) || (CAN0_BUFFER_BYPASS_NUMBER_OF_IDS > 0)) && ((CAN_USE_CAN_NODE0
1199 == 1) || (CAN_USE_CAN_NODE1 == 1)) */
1200 } else { This is the case never happens for this project.
1201     /* ##### Buffer not active ##### */
1202
1203     CAN_MSG_RX_TYPE_s* msgRXstruct;
1204     uint8_t length;
1205     if (canNode == CAN_NODE0) {
1206         msgRXstruct = &can0_RxMsgs[0];
1207         length = can_CAN0_rx_length;
1208     } else {
1209         msgRXstruct = &can1_RxMsgs[0];
1210         length = can_CAN1_rx_length;
1211     }
1212
1213     if (tmpMsgBuffer.msg.IDE == 0U) {
1214         msgID = tmpMsgBuffer.msg.StdId;
1215     } else {
1216         msgID = tmpMsgBuffer.msg.ExtId;
1217     }
1218
1219     /* Search for correct message in RX message array to check for callback */
1220     uint8_t rxMsg = 0;
1221     for (; rxMsg < length; rxMsg++) {
1222         if (msgRXstruct[rxMsg].ID == msgID) {
1223             break;
1224         }
1225     }

```



```

1223     /* Interpret received message */
1224     if (msgRXstruct[rxMsg].func != NULL) {
1225         msgRXstruct[rxMsg].func(msgID, &tmpMsgBuffer.data[0], tmpMsgBuffer.msg.DLC, tmpMsgBuffer.msg.RTR);
1226     } else {
1227         CAN_InterpretReceivedMsg(canNode, msgID, &tmpMsgBuffer.data[0], tmpMsgBuffer.msg.DLC,
1228                                 tmpMsgBuffer.msg.RTR);    See Line 1351
1229     }
1230 }
1231
1232     CAN_ReadReceiveBuffer
1233     STD_RETURN_TYPE_e CAN_ReceiveBuffer(CAN_NodeTypeDef_e canNode, Can_PduType* msg) {
1234         /* E_OK is returned, if buffer is empty and interpret function is called successful */
1235         STD_RETURN_TYPE_e retVal = E_NOT_OK;
1236
1237         #if CAN0_USE_RX_BUFFER || CAN1_USE_RX_BUFFER
1238             CAN_RX_BUFFER_s* can_rxbuffer = NULL;
1239
1240             #if CAN0_USE_RX_BUFFER && CAN_USE_CAN_NODE0 == 1
1241                 if (canNode == CAN_NODE0) {
1242                     can_rxbuffer = &can0_rxbuffer;
1243                 }
1244             #endif /* CAN0_USE_RX_BUFFER && CAN_USE_CAN_NODE0 == 1 */
1245             #if CAN1_USE_RX_BUFFER && CAN_USE_CAN_NODE1 == 1
1246                 if (canNode == CAN_NODE1) {
1247                     can_rxbuffer = &can1_rxbuffer;
1248                 }
1249             #endif /* CAN1_USE_RX_BUFFER && CAN_USE_CAN_NODE1 == 1 */
1250
1251             if (msg == NULL) {
1252                 /* null pointer to message data struct */
1253                 can_rxbuffer = NULL;
1254             }
1255
1256             if ((can_rxbuffer->ptrWrite != can_rxbuffer->ptrRead) &&
1257                 (can_rxbuffer->buffer[can_rxbuffer->ptrRead].newMsg == 1) &&
1258                 (can_rxbuffer != NULL)) {
1259                 /* buffer not empty -> read message */
1260                 if (can_rxbuffer->buffer[can_rxbuffer->ptrRead].msg.IDE == 1) {
1261                     /* Extended ID used */
1262                     msg->id = can_rxbuffer->buffer[can_rxbuffer->ptrRead].msg.ExtId;
1263                 } else {
1264                     msg->id = can_rxbuffer->buffer[can_rxbuffer->ptrRead].msg.StdId;
1265                 }
1266                 msg->dlc = can_rxbuffer->buffer[can_rxbuffer->ptrRead].msg.DLC;
1267
1268                 for (uint8_t i = 0; i < 8U; i++) {
1269                     msg->sdu[i] = can_rxbuffer->buffer[can_rxbuffer->ptrRead].data[i];
1270                 }
1271
1272                 /* Set to 0 to mark buffer entry as read. Set to 1 when writing message into buffer */
1273                 can_rxbuffer->buffer[can_rxbuffer->ptrRead].newMsg = 0;

```

```

C can_cfg.c  C can.c  C can_cfg.h x
mcu-primary > src > driver > config > C can_cfg.h > +o C
325 typedef struct CanPdu {
326     uint8_t sdu[8];
327     uint32_t id;
328     uint8_t dlc;
329 } Can_PduType; You, a month ago

```

```

C can_cfg.c  C can.h x  C can.c  C can_cfg.h
mcu-common > src > driver > can > C can.h > +o CAN_RX_BUFFER_s
112 typedef struct CAN_RX_BUFFER {
113     uint8_t ptrRead;
114     uint8_t ptrWrite;
115     uint8_t length;
116     CAN_RX_BUFFERELEMENT_s* buffer;
117 } CAN_RX_BUFFER_s; You, a month ago * Add all

```

```

C can_cfg.c  C can.h x  C can.c  C can_cfg.h
mcu-common > src > driver > can > C can.h > +o CAN_RX_BUFFER_s
100 typedef struct CAN_RX_BUFFERELEMENT {
101     CAN_RxHeaderTypeDef msg;
102     uint8_t data[8];
103     uint8_t newMsg;
104 } CAN_RX_BUFFERELEMENT_s;

```

```

1274
1275     /* Move to next buffer element */
1276     can_rxbuffer->ptrRead++;
1277     can_rxbuffer->ptrRead = can_rxbuffer->ptrRead % can_rxbuffer->length;
1278     retVal = E_OK;
1279 }
1280 #endif /* CAN0_USE_RX_BUFFER || CAN1_USE_RX_BUFFER */
1281     return retVal;
1282 }
1283
1284 /**
1285  * @brief   Receives a bypassed CAN message and interprets it
1286  *
1287  * @param   canNode: canNode on which the message has been received
1288  * @param   msgID:    message ID
1289  * @param   data:     pointer to the message data
1290  * @param   DLC:      length of received data
1291  * @param   RTR:      RTR bit of received message
1292  *
1293  * @retval  E_OK if interpreting was successful, otherwise E_NOT_OK
1294  */
1295 static STD_RETURN_TYPE_e CAN_BufferBypass(CAN_NodeTypeDef_e canNode, uint32_t msgID, uint8_t* rxData, uint8_t DLC,
1296     uint8_t RTR) {
1297     STD_RETURN_TYPE_e retVal = E_OK;
1298
1299     /* *****
1300      * Implement wished functionality of received messages here,
1301      *
1302      * if no callback function in CAN_MSG_RX_TYPE_s struct is defined
1303      ***** */
1304
1305     /* Perform SW reset */
1306     if (msgID == CAN_ID_SOFTWARE_RESET_MSG && DLC == 8) {
1307         uint8_t reset = 0;
1308
1309         /* CAN data = FF FF FF FF FF FF FF FF */
1310         for (uint8_t i = 0; i < DLC; i++) {
1311             if (rxData[i] != 0xFF) {
1312                 reset = 1;
1313             }
1314         }
1315         #if CAN_SW_RESET_WITH_DEVICE_ID == 1
1316         /* CAN data = MCU Device ID Byte [0] [1] [2] [3] [4] [5] [6] [7] */
1317         /* if (rxData[0] == (uint8_t)mcu_unique_deviceID.off0 && data[1] == (uint8_t)(mcu_unique_deviceID.off0 >> 8) &&
1318             rxData[2] == (uint8_t)(mcu_unique_deviceID.off0 >> 16) && rxData[3] ==
1319             (uint8_t)(mcu_unique_deviceID.off0 >> 24) &&
1320             rxData[4] == (uint8_t)mcu_unique_deviceID.off32 && rxData[5] == (uint8_t)(mcu_unique_deviceID.off32
1321             >> 8) &&
1322             rxData[6] == (uint8_t)(mcu_unique_deviceID.off32 >> 16) && rxData[7] ==
1323             (uint8_t)(mcu_unique_deviceID.off32 >> 24)) {
1324             reset = 1;
1325         }
1326     }

```

The screenshot shows a code editor with several files open: can_cfg.c, can.c, can_cfg.h, and stm32f4xx... The active file is can_cfg.h, showing a definition for CAN_SW_RESET_WITH_DEVICE_ID as 1. Comments above the definition describe the CAN data format as MCU Device ID bytes [0] through [7].

```

1323 */
1324
1325     if (rxData[0] == 0) {
1326         if ((CAN_CheckNodeID(&data[5]) == E_OK) || (CAN_CheckUniqueDeviceID(&data[1]) == E_OK) ||
            (CAN_CheckBroadcastID(&data[5]) == E_OK)) {
1327             reset = 1;
1328         }
1329     }
1330 #else /* CAN_SW_RESET_WITH_DEVICE_ID != 1 */
1331     reset = 1;           This line makes the operation in line 1312 useless.
1332 #endif /* CAN_SW_RESET_WITH_DEVICE_ID == 1 */
1333
1334     if (reset == 1)
1335         HAL_NVIC_SystemReset();
1336 }
1337 return retVal;
1338 }
1339
1340 /**
1341  * @brief   Interprets the received message
1342  *
1343  * @param   canNode: canNode on which the message has been received
1344  * @param   msgID:    message ID
1345  * @param   data:     pointer to the message data
1346  * @param   DLC:      length of received data
1347  * @param   RTR:      RTR bit of received message
1348  *
1349  * @return  E_OK if interpretation successful, otherwise E_NOT_OK
1350  */
1351 static STD_RETURN_TYPE_e CAN_InterpretReceivedMsg(CAN_NodeTypeDef_e canNode, uint32_t msgID, uint8_t* data, uint8_t
DLC,
1352     uint8_t RTR) {
1353     STD_RETURN_TYPE_e retVal = E_NOT_OK;
1354
1355     /* *****
1356      * Implement wished functionality of received messages here,
1357      *
1358      * if no callback function in CAN_MSG_RX_TYPE_s struct is defined
1359      *****/
1360     return retVal;
1361 }
1362
1363 /* *****
1364  * Sleep mode
1365  *****/
1366
1367 void CAN_SetSleepMode(CAN_NodeTypeDef_e canNode) {
1368     if (canNode == CAN_NODE0) {
1369         HAL_CAN_RequestSleep(&hcan0);
1370     } else if (canNode == CAN_NODE1) {
1371         HAL_CAN_RequestSleep(&hcan1);
1372     }

```

```
1373     }
1374     return;
1375 }
1376
1377
1378 void CAN_WakeUp(CAN_NodeTypeDef_e canNode) {
1379     if (canNode == CAN_NODE0) {
1380         HAL_CAN_WakeUp(&hcan0);
1381     } else if (canNode == CAN_NODE1) {
1382         HAL_CAN_WakeUp(&hcan1);
1383     }
1384     return;
1385 }
1386
```