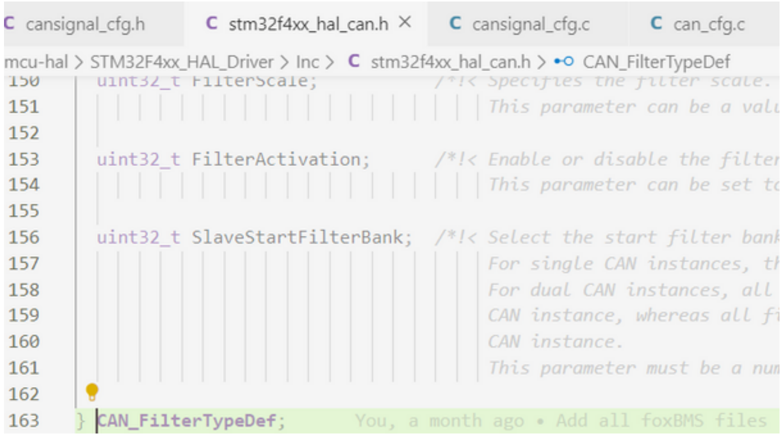
```c
/**
 *
 * @copyright &copy; 2010 - 2020, Fraunhofer-Gesellschaft zur Foerderung der
 *  angewandten Forschung e.V. All rights reserved.
 *
 * BSD 3-Clause License
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions 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    can.c
 * @author  foxBMS Team
 * @date    12.07.2015 (date of creation)
 * @ingroup DRIVERS
 * @prefix  CAN
 *
 * @brief   Driver for the CAN module
 *
 * Implementation of the CAN Interrupts, initialization, buffers,
 * receive and transmit interfaces.
 *
 */

/*=================== Includes ============================================*/
#include "can.h"

/*=================== Macros and Definitions ================================*/
#define ID_16BIT_FIFO0          (0U)
#define ID_16BIT_FIFO1          (1U)
#define ID_32BIT_FIFO0          (2U)
#define ID_32BIT_FIFO1          (3U)
#define MSK_16BIT_FIFO0         (4U)
#define MSK_16BIT_FIFO1         (5U)
#define MSK_32BIT               (6U)

/*=================== Constant and Variable Definitions ====================*/
uint8_t canNode0_listenonly_mode = 0;
uint8_t canNode1_listenonly_mode = 0;

#if CAN_USE_CAN_NODE0
#if CAN0_USE_TX_BUFFER
CAN_TX_BUFFERELEMENT_s can0_txbufferelements[CAN0_TX_BUFFER_LENGTH];
CAN_TX_BUFFER_s can0_txbuffer = {
    .length = CAN0_TX_BUFFER_LENGTH,
    .buffer = &can0_txbufferelements[0],
};
#endif /* CAN0_USE_TX_BUFFER */

#if CAN0_USE_RX_BUFFER
CAN_RX_BUFFERELEMENT_s can0_rxbufferelements[CAN0_RX_BUFFER_LENGTH];
CAN_RX_BUFFER_s can0_rxbuffer = {
    .length = CAN0_RX_BUFFER_LENGTH,
    .buffer = &can0_rxbufferelements[0],
};
#endif /* CAN0_USE_RX_BUFFER */

#if CAN0_BUFFER_BYPASS_NUMBER_OF_IDs > 0   This is for Rx.
uint8_t can0_fastLinkIndex[CAN0_BUFFER_BYPASS_NUMBER_OF_IDs];   /* Link Table for bufferBypassing */
#endif

CAN_ERROR_s CAN0_errorStruct = {
    .canError = HAL_CAN_ERROR_NONE,
    .canErrorCounter = { 0~~, 0, 0, 0, 0, 0, 0, 0, 0~~ },
};
#endif /* CAN_USE_CAN_NODE0 */

#if CAN_USE_CAN_NODE1
#if CAN1_USE_TX_BUFFER
CAN_TX_BUFFERELEMENT_s can1_txbufferelements[CAN1_TX_BUFFER_LENGTH];
CAN_TX_BUFFER_s can1_txbuffer = {
        .length = CAN1_TX_BUFFER_LENGTH,
        .buffer = &can1_txbufferelements[0],
};
#endif /* CAN1_USE_TX_BUFFER */

#if CAN1_USE_RX_BUFFER
CAN_RX_BUFFERELEMENT_s can1_rxbufferelements[CAN1_RX_BUFFER_LENGTH];
CAN_RX_BUFFER_s can1_rxbuffer = {
        .length = CAN1_RX_BUFFER_LENGTH,
        .buffer = &can1_rxbufferelements[0],
};
#endif /* CAN1_USE_RX_BUFFER */

#if CAN1_BUFFER_BYPASS_NUMBER_OF_IDs > 0
uint8_t can1_fastLinkIndex[CAN1_BUFFER_BYPASS_NUMBER_OF_IDs];    /* Link Table for bufferBypassing */
#endif

CAN_ERROR_s CAN1_errorStruct = {
    .canError = HAL_CAN_ERROR_NONE,
    .canErrorCounter = { ~~0, 0, 0, 0, 0, 0, 0, 0, 0~~ },
};
#endif /* CAN_USE_CAN_NODE1 */

/* ************************************************************
 *  Dummies for filter initialization and message reception
 *************************************************************/

CAN_FilterTypeDef sFilterConfig = {
        /* No need to insert here something */
        .FilterActivation = ENABLE,      /* enable the filter */
};
```


cansignal_cfg.h
stm32f4xx_hal_can.h ×
cansignal_cfg.c
can_cfg.c
mcu-hal > STM32F4xx_HAL_Driver > Inc > stm32f4xx_hal_can.h > CAN_FilterTypeDef

```
    uint32_t FilterScale;          /*!< Specifies the filter scale.
                                        This parameter can be a valu

    uint32_t FilterActivation;     /*!< Enable or disable the filter
                                        This parameter can be set to

    uint32_t SlaveStartFilterBank; /*!< Select the start filter bank
                                        For single CAN instances, th
                                        For dual CAN instances, all
                                        CAN instance, whereas all fi
                                        CAN instance.
                                        This parameter must be a num

} CAN_FilterTypeDef;     You, a month ago • Add all foxBMS files
```

```

/*================== Function Prototypes ===================================*/
/* Inits */
static uint8_t CAN_GetNextID(CAN_MSG_RX_TYPE_s* can_RxMsgs, uint8_t numberOfRxIDs, uint8_t startIndex,
        uint8_t filterCase);
static uint8_t CAN_NumberOfNeededFilters(CAN_MSG_RX_TYPE_s* can_RxMsgs, uint8_t* numberOfDifferentIDs, uint32_t*
error);
static uint32_t CAN_InitFilter(CAN_HandleTypeDef* ptrHcan, CAN_MSG_RX_TYPE_s* can_RxMsgs, uint8_t numberOfRxMsgs);

/* Interrupts */
static void CAN_TxCpltCallback(CAN_NodeTypeDef_e canNode);
static void CAN_RxMsg(CAN_NodeTypeDef_e canNode, CAN_HandleTypeDef* ptrHcan, uint8_t FIFONumber);

/* Buffer/Interpreter */
static STD_RETURN_TYPE_e CAN_BufferBypass(CAN_NodeTypeDef_e canNode, uint32_t msgID, uint8_t* rxData, uint8_t DLC,
        uint8_t RTR);
static STD_RETURN_TYPE_e CAN_InterpretReceivedMsg(CAN_NodeTypeDef_e canNode, uint32_t msgID, uint8_t* data, uint8_t
DLC,
        uint8_t RTR);

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

/* ***************************************
```

```
 *  Initialization
 ****************************************

uint32_t CAN_Init(void) {
    uint32_t retval = 0;

#if CAN_USE_CAN_NODE0
    /* DeInit CAN0 handle */
    if (HAL_CAN_DeInit(&hcan0) != HAL_OK) {
        /* Error deintializing handle -> set error bit */
        retval |= STD_ERR_BIT_0;
    }

    /* Init CAN0-handle */
    if (HAL_CAN_Init(&hcan0) != HAL_OK) {
        /* Error intializing handle -> set error bit */
        retval |= STD_ERR_BIT_1;
    }

    /* Configure CAN0 hardware filter */
    retval |= CAN_InitFilter(&hcan0, &can0_RxMsgs[0], can_CAN0_rx_length);
```

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

```
    /* Check if more rx messages are bypassed than received */
#pragma GCC diagnostic push
    /* configurations might exist that use this comparison */
#pragma GCC diagnostic ignored "-Wtype-limits"
    if (CAN0_BUFFER_BYPASS_NUMBER_OF_IDs > can_CAN0_rx_length) {
#pragma GCC diagnostic pop
        retval |= STD_ERR_BIT_7;
    }

    /* Enable CAN0 message receive interrupt FIFO0 */
    if (HAL_CAN_ActivateNotification(&hcan0, CAN_IT_RX_FIFO0_MSG_PENDING) != HAL_OK) {
        retval |= STD_ERR_BIT_8;
    }
    hcan0.State = HAL_CAN_STATE_READY;
    /* Enable CAN0 message receive interrupt FIFO1 */
```

It seems that FIFO 1 is never used.

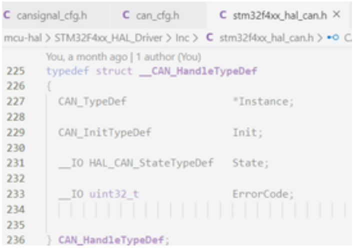
```
    if (HAL_CAN_ActivateNotification(&hcan0, CAN_IT_RX_FIFO1_MSG_PENDING) != HAL_OK) {
        retval |= STD_ERR_BIT_9;
    }
    /* Enable CAN0 Transmit mailbox empty interrupt */
    if (HAL_CAN_ActivateNotification(&hcan0, CAN_IT_TX_MAILBOX_EMPTY) != HAL_OK) {
        retval |= STD_ERR_BIT_10;
    }
```

Debug freeze

```
    /* set DBF bit to 0 for CAN activity while in debug mode */
```

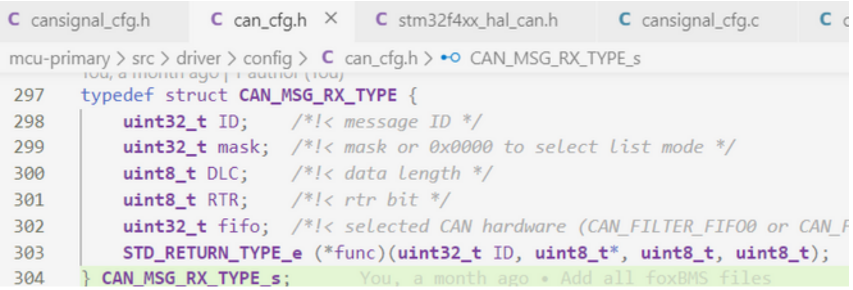
DBF = 0 ==> CAN works during debug.

```
    CLEAR_BIT(hcan0.Instance->MCR, CAN_MCR_DBF);

#if CAN_USE_STANDBY_CONTROL == 1
    IO_WritePin(CAN_0_TRANS_STANDBY_CONTROL, IO_PIN_SET);
#endif /* CAN_USE_STANDBY_CONTROL == 1 */
```



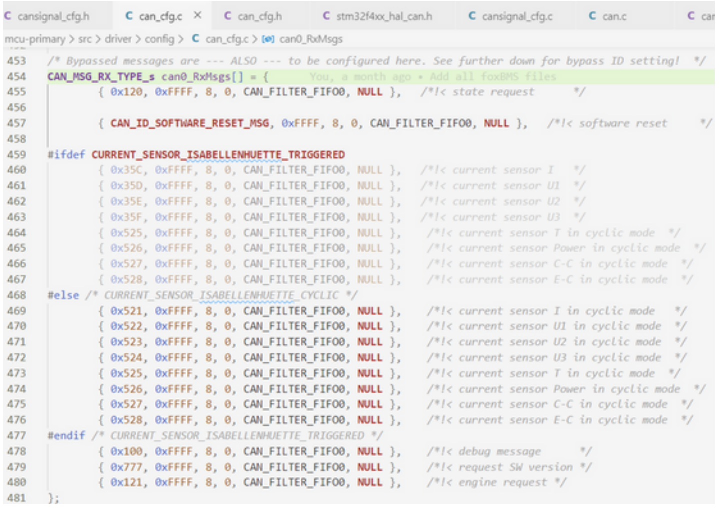
cansignal_cfg.h
can_cfg.h
stm32f4xx_hal_can.h
mcu-hal > STM32F4xx_HAL_Driver > Inc > stm32f4xx_hal_can.h > C
You, a month ago | 1 author (You)

```
typedef struct __CAN_HandleTypeDef
{
  CAN_TypeDef                 *Instance;

  CAN_InitTypeDef             Init;

  __IO HAL_CAN_StateTypeDef   State;

  __IO uint32_t               ErrorCode;


} CAN_HandleTypeDef;
```

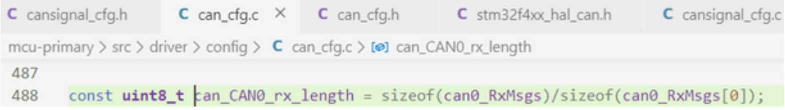


cansignal_cfg.h
can_cfg.h
stm32f4xx_hal_can.h
cansignal_cfg.c
mcu-primary > src > driver > config > can_cfg.h > CAN_MSG_RX_TYPE_s
You, a month ago | 1 author (You)

```
typedef struct CAN_MSG_RX_TYPE {
    uint32_t ID;    /*!< message ID */
    uint32_t mask;  /*!< mask or 0x0000 to select list mode */
    uint8_t DLC;    /*!< data length */
    uint8_t RTR;    /*!< rtr bit */
    uint32_t fifo;  /*!< selected CAN hardware (CAN_FILTER_FIFO0 or CAN_F
    STD_RETURN_TYPE_e (*func)(uint32_t ID, uint8_t*, uint8_t, uint8_t);
} CAN_MSG_RX_TYPE_s;      You, a month ago • Add all foxBMS files
```



```
/* Bypassed messages are --- ALSO --- to be configured here. See further down for bypass ID setting! */
CAN_MSG_RX_TYPE_s can0_RxMsgs[] = {      You, a month ago • Add all foxBMS files
        { 0x120, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },   /*!< state request      */

        { CAN_ID_SOFTWARE_RESET_MSG, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },   /*!< software reset    */

#ifdef CURRENT_SENSOR_ISABELLENHUETTE_TRIGGERED
        { 0x35C, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },   /*!< current sensor I   */
        { 0x35D, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },   /*!< current sensor U1  */
        { 0x35E, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },   /*!< current sensor U2  */
        { 0x35F, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },   /*!< current sensor U3  */
        { 0x525, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },    /*!< current sensor T in cyclic mode  */
        { 0x526, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },    /*!< current sensor Power in cyclic mode  */
        { 0x527, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },    /*!< current sensor C-C in cyclic mode  */
        { 0x528, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },    /*!< current sensor E-C in cyclic mode  */
#else /* CURRENT_SENSOR_ISABELLENHUETTE_CYCLIC */
        { 0x521, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },    /*!< current sensor I in cyclic mode   */
        { 0x522, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },    /*!< current sensor U1 in cyclic mode  */
        { 0x523, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },    /*!< current sensor U2 in cyclic mode  */
        { 0x524, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },    /*!< current sensor U3 in cyclic mode  */
        { 0x525, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },    /*!< current sensor T in cyclic mode  */
        { 0x526, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },    /*!< current sensor Power in cyclic mode  */
        { 0x527, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },    /*!< current sensor C-C in cyclic mode  */
        { 0x528, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },    /*!< current sensor E-C in cyclic mode  */
#endif /* CURRENT_SENSOR_ISABELLENHUETTE_TRIGGERED */
        { 0x100, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },    /*!< debug message      */
        { 0x777, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },    /*!< request SW version */
        { 0x121, 0xFFFF, 8, 0, CAN_FILTER_FIFO0, NULL },    /*!< engine request */
};
```



cansignal_cfg.h
can_cfg.c
can_cfg.h
stm32f4xx_hal_can.h
cansignal_cfg.c
mcu-primary > src > driver > config > can_cfg.c > can_CAN0_rx_length

```

const uint8_t can_CAN0_rx_length = sizeof(can0_RxMsgs)/sizeof(can0_RxMsgs[0]);
```

```
    /* Start CAN */
    HAL_CAN_Start(&hcan0);
#endif /* CAN_USE_CAN_NODE0 */


#if CAN_USE_CAN_NODE1
    /* DeInit CAN1 handle */
    if (HAL_CAN_DeInit(&hcan1) != HAL_OK) {
        /* Error deintializing handle -> set error bit */
        retval |= STD_ERR_BIT_11;
    }

    /* Init CAN1-handle */
    if (HAL_CAN_Init(&hcan1) != HAL_OK) {
        /* Error intializing handle -> set error bit */
        retval |= STD_ERR_BIT_12;
    }

    /* Configure CAN1 hardware filter */
    retval |= CAN_InitFilter(&hcan1, &can1_RxMsgs[0], can_CAN1_rx_length);

    /* Check if more RX messages are bypassed than received */
#pragma GCC diagnostic push
    /* configurations might exist that use this comparison */
#pragma GCC diagnostic ignored "-Wtype-limits"
    if (CAN1_BUFFER_BYPASS_NUMBER_OF_IDs > can_CAN1_rx_length) {
#pragma GCC diagnostic pop
        retval |= STD_ERR_BIT_13;
    }

    /* Enable CAN1 message receive interrupt FIFO0 */
    if (HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO0_MSG_PENDING) != HAL_OK) {
        retval |= STD_ERR_BIT_14;
    }
    /* Enable CAN1 message receive interrupt FIFO1 */
    hcan1.State = HAL_CAN_STATE_READY;
    if (HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO1_MSG_PENDING) != HAL_OK) {
        retval |= STD_ERR_BIT_15;
    }
    /* Enable CAN1 Transmit mailbox empty interrupt */
    if (HAL_CAN_ActivateNotification(&hcan1, CAN_IT_TX_MAILBOX_EMPTY) != HAL_OK) {
        retval |= STD_ERR_BIT_16;
    }

    /* set DBF bit to 0 for CAN activity while in debug mode */
    CLEAR_BIT(hcan1.Instance->MCR, CAN_MCR_DBF);

#if CAN_USE_STANDBY_CONTROL == 1
    IO_WritePin(CAN_1_TRANS_STANDBY_CONTROL, IO_PIN_SET);
#endif /* CAN_USE_STANDBY_CONTROL == 1 */

    /* Start CAN */
```

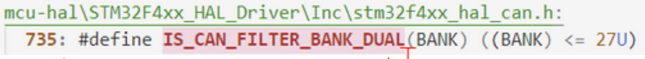
```
259         HAL_CAN_Start(&hcan1);
260     #endif /* CAN_USE_CAN_NODE1 */
261         return retval;
262     }
263 
264     /**
265      * @brief  Initializes message filtering
266      * @retval 0: if no error occurred, otherwise error code
267      */
268     static uint32_t CAN_InitFilter(CAN_HandleTypeDef* ptrHcan, CAN_MSG_RX_TYPE_s* can_RxMsgs, uint8_t numberOfRxMsgs) {
269         /* Contains the occurrence ~~of~~ 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 */
```

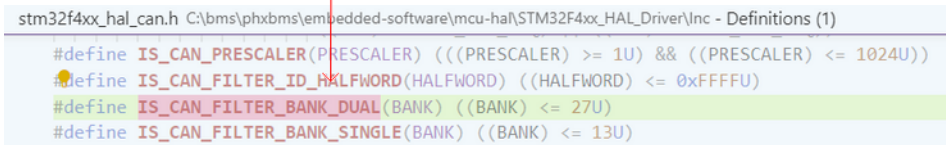
See Lines 459 to 557

```
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
            with 0 */
```



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

```
284         if (IS_CAN_FILTER_BANK_DUAL(numberNeededFilters)) {
285             uint8_t j = 0;  /* Counts the number of initialized instances per case */
286             uint8_t posRxMsgs = 0;  /* Iterator for can_RxMsgs[] */
287             uint8_t numberRegistersUsed = 0;  /* Counts how many register space is already used in each filter bank (max.
                64bit) */
288             uint8_t caseID = 0;  /* indicates the actual filter mode that will be initialized */
289 
```



```
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)
```

True for hcan0

```
290             if (ptrHcan->Instance == CAN2) {
291                 /* Set start slave bank filter */
292                 sFilterConfig.FilterBank = filterNumber;
293             }
294             for (caseID = 0; caseID < 2u; caseID++) {
295                 /* ID List mode 16bit routed on FIFO0 or FIFO1 */
296 
```

Dual filter bank-—28 filter banks.

```
297                 if (numberOfDifferentIDs[caseID] > 0U) {
298                     j = 0;
299                     while (j < numberOfDifferentIDs[caseID]) {
300                         /* Until all IDs in that filter case are treated */
301 
```

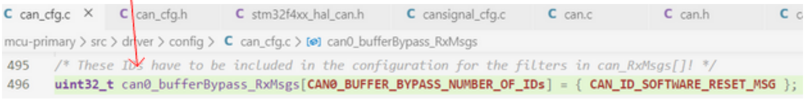
See Lines 569 to 599

```
302                         posRxMsgs = CAN_GetNextID(can_RxMsgs, numberOfRxMsgs, posRxMsgs, caseID);  /*  Get array position
                            of next ID */
303 
304                         switch (numberRegistersUsed) {
305                             case 0:  /* 1st ID per filter bank */
306                                 sFilterConfig.FilterIdHigh = ((can_RxMsgs[posRxMsgs].ID << 5)
307                                         | can_RxMsgs[posRxMsgs].RTR << 4);
```

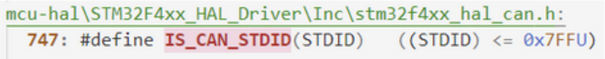
```c
                            j++;
                            break;

                        case 1:  /* 2nd ID */
                            sFilterConfig.FilterIdLow = ((can_RxMsgs[posRxMsgs].ID << 5)
                                    | can_RxMsgs[posRxMsgs].RTR << 4);
                            j++;
                            break;

                        case 2:  /* 3rd ID */
                            sFilterConfig.FilterMaskIdHigh = ((can_RxMsgs[posRxMsgs].ID << 5)
                                    | can_RxMsgs[posRxMsgs].RTR << 4);
                            j++;
                            break;

                        case 3:  /* 4th ID */
                            sFilterConfig.FilterMaskIdLow = ((can_RxMsgs[posRxMsgs].ID << 5)
                                    | can_RxMsgs[posRxMsgs].RTR << 4);
                            j++;
                            break;
                    }
                    numberRegistersUsed = j % 4U;    /* space for 4 IDs a 16 bit in one filter bank */
                    if ((numberRegistersUsed  ==  0 && j > 1U) || (j  ==  numberOfDifferentIDs[caseID])) {
                        /* all registers in filter bank used OR no more IDs in that case */
                        sFilterConfig.FilterMode = CAN_FILTERMODE_IDLIST;
                        sFilterConfig.FilterScale = CAN_FILTERSCALE_16BIT;
                        if (caseID == ID_16BIT_FIFO0) {
                            sFilterConfig.FilterFIFOAssignment = CAN_FILTER_FIFO0;
                        } else if (caseID == ID_16BIT_FIFO1) {
                            sFilterConfig.FilterFIFOAssignment = CAN_FILTER_FIFO1;
                        }
                        sFilterConfig.FilterBank = filterNumber;
                        HAL_CAN_ConfigFilter(ptrHcan, &sFilterConfig);    /* initialize filter bank */
                        filterNumber++;     /* increment filter number */
                    }
                    posRxMsgs++;    /* increment array position to find next valid ID */
                }
                posRxMsgs = 0;      /* reset variables for next case */
                numberRegistersUsed = 0;
            }
        }
        for (caseID = 2; caseID < 6U; caseID++) {
            /* ID List mode 32bit routed on FIFO0 or FIFO1; Mask mode 16bit routed on FIFO0 or FIFO1 */
            j = 0;
            if (numberOfDifferentIDs[caseID] > 0U) {
                while (j < numberOfDifferentIDs[caseID]) {
                    /* Until all IDs in that filter case are treated */

                    posRxMsgs = CAN_GetNextID(can_RxMsgs, numberOfRxMsgs, posRxMsgs, caseID);  /*  Get array position of next ID */

                    switch (numberRegistersUsed) {
```

```
                    case 0:  /* first 32bit per filter bank */
                        if (caseID  ==  ID_32BIT_FIFO0 || caseID  ==  ID_32BIT_FIFO1) {  /* list mode 32bit */
                            sFilterConfig.FilterIdHigh = ((can_RxMsgs[posRxMsgs].ID << 3) >> 16);  /* 1 << 2 is for setting IDE bit to receive extended identifiers */
                            sFilterConfig.FilterIdLow = (uint16_t)((can_RxMsgs[posRxMsgs].ID << 3) | 1 << 2
                                    | can_RxMsgs[posRxMsgs].RTR << 1);
                        } else if (caseID  ==  MSK_16BIT_FIFO0 || caseID  ==  MSK_16BIT_FIFO1) {  /* mask mode 16bit */
                            sFilterConfig.FilterIdHigh = ((can_RxMsgs[posRxMsgs].ID << 5)
                                    | can_RxMsgs[posRxMsgs].RTR << 4);
                            sFilterConfig.FilterMaskIdHigh = can_RxMsgs[posRxMsgs].mask;
                            sFilterConfig.FilterIdLow = 0x0000;      /* set second register to 0xFFFF, */
                            sFilterConfig.FilterMaskIdLow = 0xFFFF;  /* otherwise all messages would be received */
                        }
                        j++;
                        break;

                    case 1:  /* second 32bit per filter bank */
                        if (caseID  ==  ID_32BIT_FIFO0 || caseID  ==  ID_32BIT_FIFO1) {  /* list mode 32bit */
                            sFilterConfig.FilterMaskIdHigh = ((can_RxMsgs[posRxMsgs].ID << 3) >> 16);  /*  1 << 2 is for setting IDE bit to receive extended identifiers */
                            sFilterConfig.FilterMaskIdLow = (uint16_t)((can_RxMsgs[posRxMsgs].ID << 3) | 1 << 2
                                    | can_RxMsgs[posRxMsgs].RTR << 1);
                        } else if (caseID  ==  MSK_16BIT_FIFO0 || caseID  ==  MSK_16BIT_FIFO1) {  /* mask mode 16bit */
                            sFilterConfig.FilterIdLow = ((can_RxMsgs[posRxMsgs].ID << 5)
                                    | can_RxMsgs[posRxMsgs].RTR << 4);
                            sFilterConfig.FilterMaskIdLow = can_RxMsgs[posRxMsgs].mask;
                        }
                        j++;
                        break;
                }
                numberRegistersUsed = j % 2;    /* Space for two IDs a 32bit or two mask a 16bit */
                if ((numberRegistersUsed == 0 && j > 1U) || (j == numberOfDifferentIDs[caseID])) {
                    /* all registers in filter bank used OR no more IDs in that case */
                    if (caseID  ==  ID_32BIT_FIFO0 || caseID  ==  ID_32BIT_FIFO1) {
                        sFilterConfig.FilterMode = CAN_FILTERMODE_IDLIST;
                        sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
                        if (caseID  ==  ID_32BIT_FIFO0)
                            sFilterConfig.FilterFIFOAssignment = CAN_FILTER_FIFO0;
                        else
                            sFilterConfig.FilterFIFOAssignment = CAN_FILTER_FIFO1;
                    } else if (caseID  ==  MSK_16BIT_FIFO0 || caseID  ==  MSK_16BIT_FIFO1) {
                        sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
                        sFilterConfig.FilterScale = CAN_FILTERSCALE_16BIT;
                        if (caseID  ==  MSK_16BIT_FIFO0)
                            sFilterConfig.FilterFIFOAssignment = CAN_FILTER_FIFO0;
                        else
                            sFilterConfig.FilterFIFOAssignment = CAN_FILTER_FIFO1;
                    }
                    sFilterConfig.FilterBank = filterNumber;
                    HAL_CAN_ConfigFilter(ptrHcan, &sFilterConfig);    /* initialize filter bank */
```

```c
                        filterNumber++;     /* increment filter number */
                    }
                    posRxMsgs++;    /* increment array position to find next valid ID */
                }
                posRxMsgs = 0;      /* reset variables for next case */
                numberRegistersUsed = 0;
            }
        }
        j = 0;
        if (numberOfDifferentIDs[MSK_32BIT] > 0U) {
            /* Mask mode 32bit */

            while (j < numberOfDifferentIDs[MSK_32BIT]) {  /*  Get array position of next ID */
                /* Until all IDs in that filter case are treated */
                posRxMsgs = CAN_GetNextID(can_RxMsgs, numberOfRxMsgs, posRxMsgs, MSK_32BIT);

                sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK;
                sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT;
                sFilterConfig.FilterIdHigh = ((can_RxMsgs[posRxMsgs].ID << 3) >> 16);  /* 1 << 2 is for setting IDE bit to receive extended identifiers */
                sFilterConfig.FilterIdLow = (uint16_t)((can_RxMsgs[posRxMsgs].ID << 3) | 1 << 2
                        | can_RxMsgs[posRxMsgs].RTR << 1);
                sFilterConfig.FilterMaskIdHigh = can_RxMsgs[posRxMsgs].mask >> 16;
                sFilterConfig.FilterMaskIdLow = (uint16_t)(can_RxMsgs[posRxMsgs].mask);
                sFilterConfig.FilterFIFOAssignment = can_RxMsgs[posRxMsgs].fifo;
                sFilterConfig.FilterBank = filterNumber;
                HAL_CAN_ConfigFilter(ptrHcan, &sFilterConfig);
                filterNumber++;
                posRxMsgs++;
                j++;
            }
        }
    } else {
        /* Too many filterbanks needed! Check the value of CAN_NUMBER_OF_FILTERBANKS */
        /* If correct, try to reduce the IDs through masks or optimize used filter bank space. */
        /* Number of different filter cases can be evaluated in numberOfDifferentIDs[]. One */
        /* filter bank can filter as many messages as followed: */
        /* 4 IDs in list mode 16bit */
        /* 2 IDs in list mode 32bit and mask mode 16bit */
        /* 1 ID in 32bit mask mode */
        retval |= STD_ERR_BIT_6;
    }
    return retval;
}

/**
 * @brief  Returns the number of filters that have to be initialized
 *
 * @param can_RxMsgs:              pointer to receive message struct
 * @param numberOfDifferentIDs:    pointer to array, where to store the specific number of different IDs
 *
 * @retval number of needed filters
 */
static uint8_t CAN_NumberOfNeededFilters(CAN_MSG_RX_TYPE_s* can_RxMsgs, uint8_t* numberOfDifferentIDs, uint32_t* error) {
    static uint8_t retVal = 0;      /* static so save the number of filters from CAN0 and add to the ones from CAN1 */
    uint16_t can_rx_length = 0;

    if (can_RxMsgs  ==  &can0_RxMsgs[0]) {
        can_rx_length = can_CAN0_rx_length;
    } else if (can_RxMsgs  ==  &can1_RxMsgs[0]) {
        can_rx_length = can_CAN1_rx_length;
    } else {
        can_rx_length = 0;
    }

    for (uint8_t i = 0; i < can_rx_length; i++) {
#if (CAN0_BUFFER_BYPASS_NUMBER_OF_IDs > 0) && (CAN_USE_CAN_NODE0 == 1)
            (This is defined as 1)
        if (can_RxMsgs  ==  &can0_RxMsgs[0]) {
            /* Set buffer bypass IDs link table */
            for (uint8_t k = 0; k < CAN0_BUFFER_BYPASS_NUMBER_OF_IDs; k++) {
                if (can_RxMsgs[i].ID  ==  can0_bufferBypass_RxMsgs[k]) {
                    /* bypass ID  ==  ID in message receive struct */

                    can0_fastLinkIndex[k] = i;   /* set for can_bufferBypass_RxMsgs[k] link to array index */
                    break;
                }
            }
        }
#endif /* (CAN0_BUFFER_BYPASS_NUMBER_OF_IDs > 0) && (CAN_USE_CAN_NODE0 == 1) */
#if (CAN1_BUFFER_BYPASS_NUMBER_OF_IDs > 0) && (CAN_USE_CAN_NODE1 == 1)

        if (can_RxMsgs  ==  &can1_RxMsgs[0]) {
            /* Set buffer bypass IDs link table */
            for (int k = 0; k < CAN1_BUFFER_BYPASS_NUMBER_OF_IDs; k++) {
                if (can1_RxMsgs[i].ID  ==  can1_bufferBypass_RxMsgs[k]) {
                    /* bypass ID  ==  ID in message receive struct */

                    can1_fastLinkIndex[k] = i;   /* set for can1_bufferBypass_RxMsgs[k] link to array index */
                    break;
                }
            }
        }
#endif /* (CAN1_BUFFER_BYPASS_NUMBER_OF_IDs > 0) && (CAN_USE_CAN_NODE1 == 1) */
        if (can_RxMsgs[i].mask == 0 && IS_CAN_STDID(can_RxMsgs[i].ID)) {
            /* ID List mode 16bit */

            if (can_RxMsgs[i].fifo == CAN_FILTER_FIFO0) {
                numberOfDifferentIDs[ID_16BIT_FIFO0]++;
            } else if (can_RxMsgs[i].fifo == CAN_FILTER_FIFO1) {
                numberOfDifferentIDs[ID_16BIT_FIFO1]++;
            } else {
                /* Invalid FIFO selection; check can_RxMsgs[i].fifo value */
```

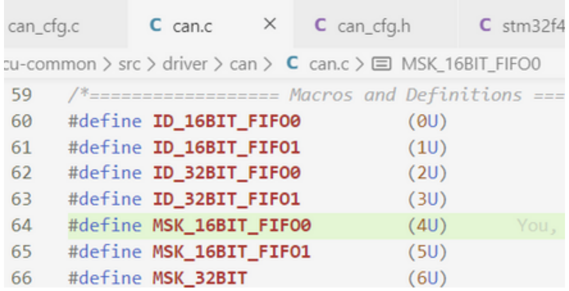

can_cfg.c
can_cfg.h
stm32f4xx_hal_can.h
cansignal_cfg.c
can.c
can.h
mcu-primary > src > driver > config > can_cfg.c > can0_bufferBypass_RxMsgs

```c
/* These IDs have to be included in the configuration for the filters in can_RxMsgs[]! */
uint32_t can0_bufferBypass_RxMsgs[CAN0_BUFFER_BYPASS_NUMBER_OF_IDs] = { CAN_ID_SOFTWARE_RESET_MSG };
```



```
mcu-hal\STM32F4xx_HAL_Driver\Inc\stm32f4xx_hal_can.h:
  747: #define IS_CAN_STDID(STDID)   ((STDID) <= 0x7FFU)
```

```c
                *error |= STD_ERR_BIT_2;
            }
        } else if ((can_RxMsgs[i].mask == 0) && IS_CAN_EXTID(can_RxMsgs[i].ID)) {
            /* ID List mode 32bit */

            if (can_RxMsgs[i].fifo == CAN_FILTER_FIFO0) {
                numberOfDifferentIDs[ID_32BIT_FIFO0]++;
            } else if (can_RxMsgs[i].fifo == CAN_FILTER_FIFO1) {
                numberOfDifferentIDs[ID_32BIT_FIFO1]++;
            } else {
                /* Invalid FIFO selection; check can_RxMsgs[i].fifo value */
                *error |= STD_ERR_BIT_3;
            }
        } else if (can_RxMsgs[i].mask > 0 && IS_CAN_STDID(can_RxMsgs[i].ID)) {
            /* Mask mode 16bit */

            if (can_RxMsgs[i].fifo == CAN_FILTER_FIFO0) {
                numberOfDifferentIDs[MSK_16BIT_FIFO0]++;
            } else if (can_RxMsgs[i].fifo == CAN_FILTER_FIFO1) {
                numberOfDifferentIDs[MSK_16BIT_FIFO1]++;
            } else {
                /* Invalid FIFO selection; check can_RxMsgs[i].fifo value */
                *error |= STD_ERR_BIT_4;
            }
        } else if ((can_RxMsgs[i].mask > 0U) && IS_CAN_EXTID(can_RxMsgs[i].ID)) {
            /* Mask mode 32bit */

            numberOfDifferentIDs[MSK_32BIT]++;
        } else {
            /* Invalid ID > IS_CAN_EXTID; check can_RxMsgs[i].ID value */
            *error |= STD_ERR_BIT_5;
            break;
        }
    }
    for (uint8_t i = 0; i < 2U; i++) {
        if (numberOfDifferentIDs[i] > 0U) {
            retVal += (numberOfDifferentIDs[i] + 2) / 4;  /* 4 IDs per filter; rounding up */
        }
    }
    for (uint8_t i = 2; i < 6U; i++) {
        if (numberOfDifferentIDs[i] > 0U) {
            retVal += (numberOfDifferentIDs[i] + 1) / 2;  /* 2 IDs per filter; rounding up */
        }
    }
    if (numberOfDifferentIDs[MSK_32BIT] > 0U) {
        retVal += numberOfDifferentIDs[6];                /* 1 ID per filter */
    }
    return retVal;
}

/**
 * @brief  Returns the next index of wished filter ID setting in CAN_MSG_RX_TYPE_t can_RxMsgs[CAN_NUMBER_OF_RX_IDs]
```



```c
/*================== Macros and Definitions ===
#define ID_16BIT_FIFO0          (0U)
#define ID_16BIT_FIFO1          (1U)
#define ID_32BIT_FIFO0          (2U)
#define ID_32BIT_FIFO1          (3U)
#define MSK_16BIT_FIFO0         (4U)        You,
#define MSK_16BIT_FIFO1         (5U)
#define MSK_32BIT               (6U)
```

```
 *
 * @param can_RxMsgs:     pointer to receive message struct
 * @param numberOfRxIDs:  count of that type of receive message in can_RxMsgs struct
 * @param startIndex:     index where to start searching
 * @param filterCase:     specifies the object what will be found
 *
 * @retval returns index
 */
static uint8_t CAN_GetNextID(CAN_MSG_RX_TYPE_s* can_RxMsgs, uint8_t numberOfRxIDs, uint8_t startIndex,
        uint8_t filterCase) {
    uint8_t retVal = 0;
    uint8_t i = startIndex;
```

There can be a hidden bug—if startIndex = 0, there is risk that retVal = 0 is ambiguous.

```
    while (i < numberOfRxIDs) {
        if ((filterCase  ==  ID_16BIT_FIFO0 && can_RxMsgs[i].mask == 0U) && IS_CAN_STDID(can_RxMsgs[i].ID) &&
        (can_RxMsgs[i].fifo == CAN_FILTER_FIFO0)) {
            retVal = i;
            break;
```

All the else if branches use the same operations; these branches can be combined using OR.

```
        } else if ((filterCase == ID_16BIT_FIFO1) && (can_RxMsgs[i].mask == 0U) && IS_CAN_STDID(can_RxMsgs[i].ID) &&
        (can_RxMsgs[i].fifo == CAN_FILTER_FIFO1)) {
            retVal = i;
            break;
        } else if ((filterCase == ID_32BIT_FIFO0) && (can_RxMsgs[i].mask == 0U) && !IS_CAN_STDID(can_RxMsgs[i].ID) &&
        IS_CAN_EXTID(can_RxMsgs[i].ID) && (can_RxMsgs[i].fifo == CAN_FILTER_FIFO0)) {
            retVal = i;
            break;
        } else if ((filterCase == ID_32BIT_FIFO1) && (can_RxMsgs[i].mask == 0U) && !IS_CAN_STDID(can_RxMsgs[i].ID) &&
        IS_CAN_EXTID(can_RxMsgs[i].ID) && (can_RxMsgs[i].fifo == CAN_FILTER_FIFO1)) {
            retVal = i;
            break;
        } else if ((filterCase == MSK_16BIT_FIFO0) && (can_RxMsgs[i].mask > 0U) && IS_CAN_STDID(can_RxMsgs[i].ID) &&
        (can_RxMsgs[i].fifo == CAN_FILTER_FIFO0)) {
            retVal = i;
            break;
        } else if ((filterCase == MSK_16BIT_FIFO1) && (can_RxMsgs[i].mask > 0U) && IS_CAN_STDID(can_RxMsgs[i].ID) &&
        (can_RxMsgs[i].fifo == CAN_FILTER_FIFO1)) {
            retVal = i;
            break;
        } else if ((filterCase == MSK_32BIT) && (can_RxMsgs[i].mask > 0U) && !IS_CAN_STDID(can_RxMsgs[i].ID) &&
        IS_CAN_EXTID(can_RxMsgs[i].ID)) {
            retVal = i;
            break;
        }
        i++;
    }
    return retVal;
}

/* ****************************************
 *  Interrupt handling
 *****************************************/

/**
```

```c
  * @brief  Transmission Mailbox 0 complete callback.
  * @param  hcan pointer to a CAN_HandleTypeDef structure that contains
  *         the configuration information for the specified CAN.
  * @retval None
  */
void HAL_CAN_TxMailbox0CompleteCallback(CAN_HandleTypeDef *hcan) {
#if CAN0_USE_TX_BUFFER
    if (hcan->Instance  ==  CAN2) {
        CAN_TxCpltCallback(CAN_NODE0);
    }
#endif /* CAN0_USE_TX_BUFFER */
#if CAN1_USE_TX_BUFFER
    /* No need for callback, if no buffer is used */
    if (hcan->Instance  ==  CAN1) {
        /* Transmission complete callback */
        CAN_TxCpltCallback(CAN_NODE1);
    }
#endif /* CAN1_USE_TX_BUFFER */
}

/**
  * @brief  Transmission Mailbox 1 complete callback.
  * @param  hcan pointer to a CAN_HandleTypeDef structure that contains
  *         the configuration information for the specified CAN.
  * @retval None
  */
void HAL_CAN_TxMailbox1CompleteCallback(CAN_HandleTypeDef *hcan) {
#if CAN0_USE_TX_BUFFER
    if (hcan->Instance  ==  CAN2) {
        CAN_TxCpltCallback(CAN_NODE0);
    }
#endif /* CAN0_USE_TX_BUFFER */
#if CAN1_USE_TX_BUFFER
    /* No need for callback, if no buffer is used */
    if (hcan->Instance  ==  CAN1) {
        /* Transmission complete callback */
        CAN_TxCpltCallback(CAN_NODE1);
    }
#endif /* CAN1_USE_TX_BUFFER */
}

/**
  * @brief  Transmission Mailbox 2 complete callback.
  * @param  hcan pointer to a CAN_HandleTypeDef structure that contains
  *         the configuration information for the specified CAN.
  * @retval None
  */
void HAL_CAN_TxMailbox2CompleteCallback(CAN_HandleTypeDef *hcan) {
#if CAN0_USE_TX_BUFFER
    if (hcan->Instance  ==  CAN2) {
        CAN_TxCpltCallback(CAN_NODE0);
    }
```

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

```c
#endif /* CAN0_USE_TX_BUFFER */
#if CAN1_USE_TX_BUFFER
    /* No need for callback, if no buffer is used */
    if (hcan->Instance  ==  CAN1) {
        /* Transmission complete callback */
        CAN_TxCpltCallback(CAN_NODE1);
    }
#endif /* CAN1_USE_TX_BUFFER */
}

/**
  * @brief  Rx FIFO 0 message pending callback.
  * @param  hcan pointer to a CAN_HandleTypeDef structure that contains
  *         the configuration information for the specified CAN.
  * @retval None
  */
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan) {
    /* Call callback function to interpret or save RX message in buffer */
    if (hcan->Instance  ==  CAN2) {
        CAN_RxMsg(CAN_NODE0, hcan, CAN_RX_FIFO0);        /* change towards HAL_CAN_IRQHandler */
    }
    if (hcan->Instance  ==  CAN1) {
        CAN_RxMsg(CAN_NODE1, hcan, CAN_RX_FIFO0);        /* change towards HAL_CAN_IRQHandler */
    }
}

/**
  * @brief  Rx FIFO 1 message pending callback.
  * @param  hcan pointer to a CAN_HandleTypeDef structure that contains
  *         the configuration information for the specified CAN.
  * @retval None
  */
void HAL_CAN_RxFifo1MsgPendingCallback(CAN_HandleTypeDef *hcan) {
    /* Call callback function to interpret or save RX message in buffer */
    if (hcan->Instance  ==  CAN2) {
        CAN_RxMsg(CAN_NODE0, hcan, CAN_RX_FIFO1);        /* change towards HAL_CAN_IRQHandler */
    }
    if (hcan->Instance  ==  CAN1) {
        CAN_RxMsg(CAN_NODE1, hcan, CAN_RX_FIFO1);        /* change towards HAL_CAN_IRQHandler */
    }
}

/**
  * @brief  Error CAN callback.
  * @param  hcan pointer to a CAN_HandleTypeDef structure that contains
  *         the configuration information for the specified CAN.
  * @retval None
  */
void HAL_CAN_ErrorCallback(CAN_HandleTypeDef *hcan) {
    CAN_ERROR_s* errorStruct = NULL_PTR;

    if (hcan->Instance == CAN1) {
```

```c
#if CAN_USE_CAN_NODE1 == 1
        errorStruct = &CAN1_errorStruct;
#endif /* CAN_USE_CAN_NODE1 == 1 */
    } else {
#if CAN_USE_CAN_NODE0 == 1
        errorStruct = &CAN0_errorStruct;
#endif /* CAN_USE_CAN_NODE0 == 1 */
    }

    /* Check Error Warning flag and set error codes */
    if (errorStruct != NULL_PTR) {
        /* Check Error Warning Flag */
        if ((hcan->ErrorCode & HAL_CAN_ERROR_EWG) != 0) {
            /* This bit is set by hardware when the warning limit has been
             * reached (Receive Error Counter or Transmit Error Counter>=96
             * until error counter 127 write error frames dominant on can bus
             * increment error occurrence of error warning state */
            errorStruct->canErrorCounter[0]++;
        }

        /* Check Error Passive Flag */
        if ((hcan->ErrorCode & HAL_CAN_ERROR_EPV) != 0) {
            /* This bit is set by hardware when the Error Passive limit has
             * been reached (Receive Error Counter or Transmit Error Counter
             * > 127) write error frames recessive on can bus increment error
             * occurrence of error passive state */
            errorStruct->canErrorCounter[1]++;
        }
        /* Check Bus-Off Flag */
        if ((hcan->ErrorCode & HAL_CAN_ERROR_BOF) != 0) {
            /* This bit is set by hardware when it enters the bus-off state.
             * The bus-off state is entered on TEC overflow, greater than 255
             * increment error occurrence of bus-off state */
            errorStruct->canErrorCounter[2]++;
        }
        /* Check stuff error flag */
        if ((hcan->ErrorCode & HAL_CAN_ERROR_STF) != 0) {
            /* When five consecutive bits of the same level have been
             * transmitted by a node, it will add a sixth bit of the opposite
             * level to the outgoing bit stream. The receivers will remove this
             * extra bit.This is done to avoid excessive DC components on the
             * bus, but it also gives the receivers an extra opportunity to
             * detect errors: if more than five consecutive bits of the same
             * level occurs on the bus, a Stuff Error is signaled. */
            errorStruct->canErrorCounter[3]++;
        }
        if ((hcan->ErrorCode & HAL_CAN_ERROR_FOR) != 0) {
            /* FORM ERROR --- Some parts of the CAN message have a fixed format,
             * i.e. the standard defines exactly what levels must occur and
             * when. (Those parts are the CRC Delimiter, ACK Delimiter, End of
             * Frame, and also the Intermission, but there are some extra
             * special error checking rules for that.) If a CAN controller
```

```
                 * detects an invalid value in one of these fixed fields, a Form
                 * Error is signaled. */
                errorStruct->canErrorCounter[4]++;
            }
            if ((hcan->ErrorCode & HAL_CAN_ERROR_ACK) != 0) {
                /* ACKNOWLEDGMENT ERROR --- All nodes on the bus that correctly
                 * receives a message (regardless of their being interested of its
                 * contents or not) are expected to send a dominant level in the
                 * so-called Acknowledgement Slot in the message. The transmitter
                 * will transmit a recessive level here. If the transmitter can
                 * detect a dominant level in the ACK slot, an Acknowledgement
                 * Error is signaled. */
                errorStruct->canErrorCounter[5]++;
            }
            if ((hcan->ErrorCode & HAL_CAN_ERROR_BR) != 0) {
                /* BIT RECESSIVE ERROR --- Each transmitter on the CAN bus monitors
                 * (i.e. reads back) the transmitted signal level. If the bit level
                 * actually read differs from the one transmitted, a Bit Error (No
                 * bit error is raised during the arbitration process.) */
                errorStruct->canErrorCounter[6]++;
            }
            if ((hcan->ErrorCode & HAL_CAN_ERROR_BD) != 0) {
                /* BIT DOMINANT ERROR --- Each transmitter on the CAN bus monitors
                 * (i.e. reads back) the transmitted signal level. If the bit level
                 * actually read differs from the one transmitted, a Bit Error (No
                 *  bit error is raised during the arbitration process.) */
                errorStruct->canErrorCounter[7]++;
            }
            if ((hcan->ErrorCode & HAL_CAN_ERROR_CRC) != 0) {
                /* CRC ERROR --- Each message features a 15-bit Cyclic Redundancy
                 * Checksum (CRC), and any node that detects a different CRC in the
                 * message than what it has calculated itself will signal an CRC
                 * Error. */
                errorStruct->canErrorCounter[8]++;
            }
            if ((hcan->ErrorCode & HAL_CAN_ERROR_RX_FOV0) != 0) {
                /* Rx FIFO0 overrun error */
                errorStruct->canErrorCounter[9]++;
            }
            if ((hcan->ErrorCode & HAL_CAN_ERROR_RX_FOV1) != 0) {
                /* Rx FIFO1 overrun error */
                errorStruct->canErrorCounter[10]++;
            }
            if ((hcan->ErrorCode & HAL_CAN_ERROR_TX_ALST0) != 0) {
                /* TxMailbox 0 transmit failure due to arbitration lost */
                errorStruct->canErrorCounter[11]++;
            }
            if ((hcan->ErrorCode & HAL_CAN_ERROR_TX_TERR0) != 0) {
                /* TxMailbox 1 transmit failure due to transmit error */
                errorStruct->canErrorCounter[12]++;
            }
            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  */
```

Transmit from a mailbox is complete

```
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 alos good for code reuse.

```c
#endif /* #if CAN_USE_CAN_NODE0 == 1 */
    } else if (canNode  ==  CAN_NODE1) {
#if CAN_USE_CAN_NODE1 == 1
        can_txbuffer = &can1_txbuffer;
#endif /* CAN_USE_CAN_NODE1 == 1 */
    }
    /* Transmit buffer existing, check if message is ready for transmission */
    if (can_txbuffer != NULL) {
        /* No Error during start of transmission */
        if ((can_txbuffer->ptrWrite  ==  can_txbuffer->ptrRead)
                && (can_txbuffer->buffer[can_txbuffer->ptrRead].newMsg == 0)) {
            /* nothing to transmit, buffer is empty */
            retVal = E_NOT_OK;
        } else {
            retVal = CAN_TxMsgBuffer(canNode);
            if (retVal  !=  E_OK) {
                /* Error during transmission, retransmit message later */
                retVal = E_NOT_OK;
            }
        }
    } else {
        /* no transmit buffer active */
        retVal = E_NOT_OK;
    }
}



/* ***************************************
 *  Transmit message
 ****************************************/

STD_RETURN_TYPE_e CAN_TxMsg(CAN_NodeTypeDef_e canNode, uint32_t msgID, uint8_t* ptrMsgData, uint32_t msgLength,
        uint32_t RTR) {
    STD_RETURN_TYPE_e retVal = E_NOT_OK;
    CAN_TxHeaderTypeDef canMessage;
    CAN_HandleTypeDef *ptrHcan;
    uint32_t freeMailboxes = 0;
    uint32_t *ptrMailbox = NULL_PTR;

    if (canNode  ==  CAN_NODE0) {
        if (canNode0_listenonly_mode) {
            ptrHcan = NULL;
        } else {
            /* Check if at least one mailbox is free */
            freeMailboxes = HAL_CAN_GetTxMailboxesFreeLevel(&hcan0);
            if (freeMailboxes != 0) {
                ptrHcan = &hcan0;
            } else {
                ptrHcan = NULL;
            }
        }
    } else if (canNode  ==  CAN_NODE1) {
        if (canNode1_listenonly_mode) {
            ptrHcan = NULL;
        } else {
            /* Check if at least one mailbox is free */
            freeMailboxes = HAL_CAN_GetTxMailboxesFreeLevel(&hcan1);
            if (freeMailboxes != 0) {
                ptrHcan = &hcan1;
            } else {
                ptrHcan = NULL;
            }
        }
    } else {
        ptrHcan = NULL;
    }

    if ((IS_CAN_STDID(msgID) || IS_CAN_EXTID(msgID)) && IS_CAN_DLC(msgLength) && ptrHcan != NULL) {
        if (IS_CAN_STDID(msgID)) {
            canMessage.StdId = msgID;
            canMessage.IDE = CAN_ID_STD;
        } else {
            canMessage.ExtId = msgID;
            canMessage.IDE = CAN_ID_EXT;
        }
        canMessage.DLC = msgLength;
        canMessage.RTR = RTR;
        canMessage.TransmitGlobalTime = DISABLE;

        /* Copy message in TX mailbox and transmit it */
        HAL_CAN_AddTxMessage(ptrHcan, &canMessage, ptrMsgData, ptrMailbox);
    } else {
        retVal = E_NOT_OK;
    }
    return retVal;
}
```

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

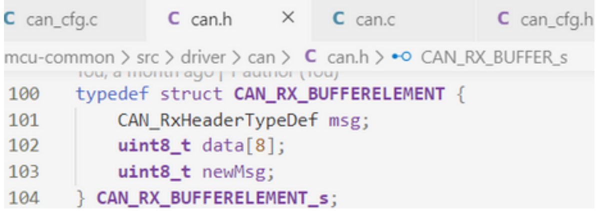
Send to buffer

```c
STD_RETURN_TYPE_e CAN_Send(CAN_NodeTypeDef_e canNode, uint32_t msgID, uint8_t* ptrMsgData, uint32_t msgLength,
        uint32_t RTR) {
    STD_RETURN_TYPE_e retVal = E_NOT_OK;
    uint8_t tmptxbuffer_wr = 0;
    CAN_TX_BUFFER_s* can_txbuffer = NULL_PTR;

    if (canNode  ==  CAN_NODE0) {
#if CAN_USE_CAN_NODE0 == 1
        can_txbuffer = &can0_txbuffer;
#endif /* #if CAN_USE_CAN_NODE0 == 1 */
    } else if (canNode  ==  CAN_NODE1) {
#if CAN_USE_CAN_NODE1 == 1
        can_txbuffer = &can1_txbuffer;
#endif /* #if CAN_USE_CAN_NODE1 == 1 */
    }

    /* Transmit buffer exisiting */
    if (can_txbuffer != NULL_PTR) {
        tmptxbuffer_wr = can_txbuffer->ptrWrite;

        if (tmptxbuffer_wr  ==  can_txbuffer->ptrRead) {
            if (can_txbuffer->buffer[tmptxbuffer_wr].newMsg  ==  0) {
                /* free buffer space for message */

                can_txbuffer->ptrWrite++;
                can_txbuffer->ptrWrite = can_txbuffer->ptrWrite % can_txbuffer->length;
                retVal = E_OK;
            } else {
                /* buffer full */
                retVal = E_NOT_OK;
            }
        } else {
            can_txbuffer->ptrWrite++;
            can_txbuffer->ptrWrite = can_txbuffer->ptrWrite % can_txbuffer->length;
            retVal = E_OK;
        }
    } else {
        retVal = E_NOT_OK;
    }

    if ((can_txbuffer != NULL_PTR) &&
        (retVal  ==  E_OK) &&
        (IS_CAN_STDID(msgID) || IS_CAN_EXTID(msgID)) &&
        (IS_CAN_DLC(msgLength))) {
        /* if buffer free and valid CAN identifier */

        if (IS_CAN_STDID(msgID)) {
            can_txbuffer->buffer[tmptxbuffer_wr].msg.StdId = msgID;
            can_txbuffer->buffer[tmptxbuffer_wr].msg.IDE = 0;
        } else {
            can_txbuffer->buffer[tmptxbuffer_wr].msg.ExtId = msgID;
            can_txbuffer->buffer[tmptxbuffer_wr].msg.IDE = 1;
        }
        can_txbuffer->buffer[tmptxbuffer_wr].newMsg = 1;
        can_txbuffer->buffer[tmptxbuffer_wr].msg.RTR = RTR;
        can_txbuffer->buffer[tmptxbuffer_wr].msg.DLC = msgLength;   /* Data length of the frame that will be transmitted */
        can_txbuffer->buffer[tmptxbuffer_wr].msg.TransmitGlobalTime = DISABLE;

        /* copy message data in handle transmit structure */
        can_txbuffer->buffer[tmptxbuffer_wr].data[0] = ptrMsgData[0];
        can_txbuffer->buffer[tmptxbuffer_wr].data[1] = ptrMsgData[1];
        can_txbuffer->buffer[tmptxbuffer_wr].data[2] = ptrMsgData[2];
        can_txbuffer->buffer[tmptxbuffer_wr].data[3] = ptrMsgData[3];
        can_txbuffer->buffer[tmptxbuffer_wr].data[4] = ptrMsgData[4];
        can_txbuffer->buffer[tmptxbuffer_wr].data[5] = ptrMsgData[5];
        can_txbuffer->buffer[tmptxbuffer_wr].data[6] = ptrMsgData[6];
```

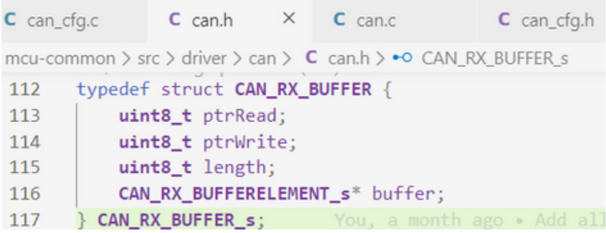
A for loop works better

```
        can_txbuffer->buffer[tmptxbuffer_wr].data[7] = ptrMsgData[7];

        retVal = E_OK;
    } else {
        retVal = E_NOT_OK;
    }

    return retVal;
}


STD_RETURN_TYPE_e CAN_TxMsgBuffer(CAN_NodeTypeDef_e canNode) {
    STD_RETURN_TYPE_e retVal = E_NOT_OK;
    HAL_StatusTypeDef retHal = HAL_ERROR;
    CAN_TX_BUFFER_s* can_txbuffer = NULL;
    CAN_HandleTypeDef* ptrHcan = NULL;
    uint32_t freeMailboxes = 0;
    uint32_t *ptrMailbox = NULL_PTR;

    if (canNode  ==  CAN_NODE0) {
#if CAN_USE_CAN_NODE0 == 1
        if (!canNode0_listenonly_mode) {
            can_txbuffer = &can0_txbuffer;
            ptrHcan = &hcan0;
        }
#endif /* #if CAN_USE_CAN_NODE0 == 1 */
    } else if (canNode  ==  CAN_NODE1) {
#if CAN_USE_CAN_NODE1 == 1
        if (!canNode1_listenonly_mode) {
            can_txbuffer = &can1_txbuffer;
            ptrHcan = &hcan1;
        }
#endif /* CAN_USE_CAN_NODE1 == 1 */
    }
    /* Check if at least one mailbox is free */
    freeMailboxes = HAL_CAN_GetTxMailboxesFreeLevel(&hcan0);

    if ((can_txbuffer != NULL) && (freeMailboxes != 0)) {
        if ((can_txbuffer->ptrWrite  ==  can_txbuffer->ptrRead)
                && (can_txbuffer->buffer[can_txbuffer->ptrRead].newMsg  ==  0)) {
            /* nothing to transmit, buffer is empty */
            retVal = E_NOT_OK;
        } else {
            /* Copy message into TX mailbox and transmit it */
            retHal = HAL_CAN_AddTxMessage(ptrHcan, &can_txbuffer->buffer[can_txbuffer->ptrRead].msg, can_txbuffer->buffer[can_txbuffer->ptrRead].data, ptrMailbox);

            if (retHal == HAL_OK) {
                /* No Error during start of transmission */
                can_txbuffer->buffer[can_txbuffer->ptrRead].newMsg = 0;    /* Msg is sent, set newMsg to 0, to allow writing of new data in buffer space */
                can_txbuffer->ptrRead++;
```

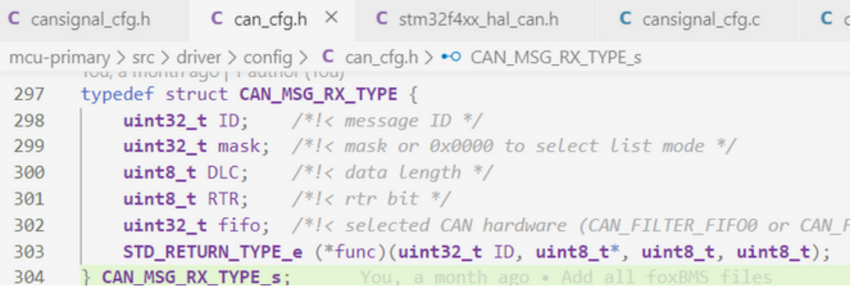
```c
                can_txbuffer->ptrRead = can_txbuffer->ptrRead % can_txbuffer->length;
                retVal = E_OK;
            } else {
                retVal = E_NOT_OK;        /* Error during transmission, retransmit message later */
            }
        }
    } else {
        /* no transmit buffer active or TX mailboxes full */
        retVal = E_NOT_OK;
    }
    return retVal;
}

/* ****************************************
 *  Receive message
 ****************************************/

/**
 * @brief  Receives CAN messages and stores them either in RxBuffer or in hcan
 *
 * @param  canNode:    canNode which received the message
 * @param  ptrHcan:    pointer to a CAN_HandleTypeDef structure that contains
 *                     the message information of the specified CAN.
 * @param  FIFONumber: FIFO in which the message has been received
 * @retval none (void)
 */
static void CAN_RxMsg(CAN_NodeTypeDef_e canNode, CAN_HandleTypeDef* ptrHcan, uint8_t FIFONumber) {
    uint8_t bypassLinkIndex = 0;
    CAN_RX_BUFFERELEMENT_s tmpMsgBuffer;
    uint32_t msgID = 0;

#if CAN0_USE_RX_BUFFER || CAN1_USE_RX_BUFFER
    uint32_t* can_bufferbypass_rxmsgs = NULL;
    uint32_t bufferbypasslength = 0;
    CAN_RX_BUFFER_s* can_rxbuffer = NULL;
    CAN_MSG_RX_TYPE_s* can_rxmsgs = NULL;
    uint8_t* can_fastLinkIndex = NULL;
#endif /* CAN0_USE_RX_BUFFER || CAN1_USE_RX_BUFFER */

    /* Set pointer on respective RxBuffer */
    if (canNode  ==  CAN_NODE1) {
#if CAN1_USE_RX_BUFFER && CAN_USE_CAN_NODE1 == 1
        can_rxbuffer = &can1_rxbuffer;
#if (CAN1_BUFFER_BYPASS_NUMBER_OF_IDs > 0u)
        can_rxmsgs = &can1_RxMsgs[0];
        can_bufferbypass_rxmsgs = &can1_bufferBypass_RxMsgs[0];
        bufferbypasslength = CAN1_BUFFER_BYPASS_NUMBER_OF_IDs;
        can_fastLinkIndex = &can1_fastLinkIndex[0];
#endif /* (CAN1_BUFFER_BYPASS_NUMBER_OF_IDs > 0u) */
#endif /* CAN1_USE_RX_BUFFER && CAN_USE_CAN_NODE1 == 1 */
    } else if (canNode  ==  CAN_NODE0) {
#if CAN0_USE_RX_BUFFER && CAN_USE_CAN_NODE0 == 1
```



```c
typedef struct CAN_RX_BUFFERELEMENT {
    CAN_RxHeaderTypeDef msg;
    uint8_t data[8];
    uint8_t newMsg;
} CAN_RX_BUFFERELEMENT_s;
```


C can_cfg.c
C can.h ×
C can.c
C can_cfg.h
mcu-common > src > driver > can > C can.h > CAN_RX_BUFFER_s
You, a month ago • Add all

```c
typedef struct CAN_RX_BUFFER {
    uint8_t ptrRead;
    uint8_t ptrWrite;
    uint8_t length;
    CAN_RX_BUFFERELEMENT_s* buffer;
} CAN_RX_BUFFER_s;
```

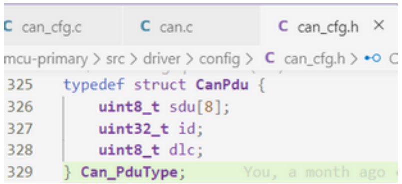

C cansignal_cfg.h
C can_cfg.h ×
C stm32f4xx_hal_can.h
C cansignal_cfg.c
mcu-primary > src > driver > config > C can_cfg.h > CAN_MSG_RX_TYPE_s
You, a month ago • Add all foxBMS files

```c
typedef struct CAN_MSG_RX_TYPE {
    uint32_t ID;    /*!< message ID */
    uint32_t mask;  /*!< mask or 0x0000 to select list mode */
    uint8_t DLC;    /*!< data length */
    uint8_t RTR;    /*!< rtr bit */
    uint32_t fifo;  /*!< selected CAN hardware (CAN_FILTER_FIFO0 or CAN_F
    STD_RETURN_TYPE_e (*func)(uint32_t ID, uint8_t*, uint8_t, uint8_t);
} CAN_MSG_RX_TYPE_s;
```

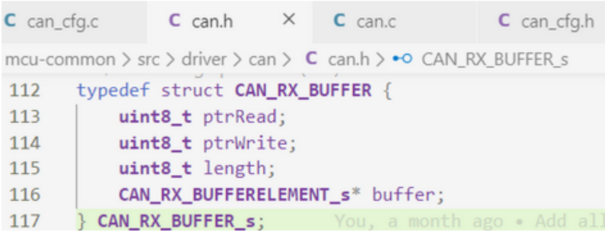
```
1123            can_rxbuffer = &can0_rxbuffer;
1124    #if (CAN0_BUFFER_BYPASS_NUMBER_OF_IDs > 0U)
1125            can_rxmsgs = &can0_RxMsgs[0];              See Line 175.
1126            can_bufferbypass_rxmsgs = &can0_bufferBypass_RxMsgs[0];
1127            bufferbypasslength = CAN0_BUFFER_BYPASS_NUMBER_OF_IDs;
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            }                       No match found in the previous step, which means that we need to put the data from
1146        }                           tmpMsgBuffer to the buffer.
1147    #endif /* #if (CAN1_BUFFER_BYPASS_NUMBER_OF_IDs > 0) || (CAN0_BUFFER_BYPASS_NUMBER_OF_IDs > 0) */
1148        if (bypassLinkIndex >= bufferbypasslength && can_rxbuffer != NULL) {
1149            /* ##### Use buffer / Copy data in buffer ##### */
1150
1151    #if CAN0_USE_RX_BUFFER || CAN1_USE_RX_BUFFER
1152            /* NO NEED TO DISABLE INTERRUPTS, BECAUSE FUNCTION IS CALLED FROM ISR */
1153
1154            /* Set to 1 to mark message as new received. Set to 0 when reading message from buffer */
1155            can_rxbuffer->buffer[can_rxbuffer->ptrWrite].newMsg = 1;
1156
1157            /* Get message header */                Note that msg is a struct and we can copy a struct in C directly.
1158            can_rxbuffer->buffer[can_rxbuffer->ptrWrite].msg = tmpMsgBuffer.msg;
1159
1160            /* Get the data field */
1161            for (uint8_t i = 0; i < can_rxbuffer->buffer[can_rxbuffer->ptrWrite].msg.DLC; i++) {
1162                can_rxbuffer->buffer[can_rxbuffer->ptrWrite].data[i] = tmpMsgBuffer.data[i];
1163            }
1164
1165            /* Increment write pointer */     Bypass the buffer
1166            can_rxbuffer->ptrWrite++;
1167            can_rxbuffer->ptrWrite = can_rxbuffer->ptrWrite % can_rxbuffer->length;
1168    #endif /* CAN0_USE_RX_BUFFER || CAN1_USE_RX_BUFFER */
1169        } else if (bypassLinkIndex < bufferbypasslength && can_rxmsgs != NULL && can_fastLinkIndex != NULL) {
1170            /* ##### Buffer active but bypassed ##### */
1171
1172    #if ((CAN1_BUFFER_BYPASS_NUMBER_OF_IDs > 0) || (CAN0_BUFFER_BYPASS_NUMBER_OF_IDs > 0)) && ((CAN_USE_CAN_NODE0 == 1)
        || (CAN_USE_CAN_NODE1 == 1))    This #if is buggy.
1173            /* 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,
                        tmpMsgBuffer.msg.RTR);      See Line 1295
1183                }
1184            } else {
1185                if (can_rxmsgs[can_fastLinkIndex[bypassLinkIndex]].func != NULL) {
1186                    can_rxmsgs[can_fastLinkIndex[bypassLinkIndex]].func(tmpMsgBuffer.msg.ExtId,
1187                                                                        tmpMsgBuffer.data,
1188                                                                        tmpMsgBuffer.msg.DLC,
1189                                                                        tmpMsgBuffer.msg.RTR);
1190                } else {
1191                    /* No callback function defined */
1192                    CAN_BufferBypass(canNode, tmpMsgBuffer.msg.ExtId, tmpMsgBuffer.data, tmpMsgBuffer.msg.DLC,
                        tmpMsgBuffer.msg.RTR);
1193                }
1194            }
1195    #endif /* ((CAN1_BUFFER_BYPASS_NUMBER_OF_IDs > 0) || (CAN0_BUFFER_BYPASS_NUMBER_OF_IDs > 0)) && ((CAN_USE_CAN_NODE0
        == 1) || (CAN_USE_CAN_NODE1 == 1)) */
1196        } else {     This is the case never happens for this project.
1197            /* ##### Buffer not active ##### */
1198
1199            CAN_MSG_RX_TYPE_s* msgRXstruct;
1200            uint8_t length;
1201            if (canNode == CAN_NODE0) {
1202                msgRXstruct = &can0_RxMsgs[0];
1203                length = can_CAN0_rx_length;
1204            } else {
1205                msgRXstruct = &can1_RxMsgs[0];
1206                length = can_CAN1_rx_length;
1207            }
1208
1209            if (tmpMsgBuffer.msg.IDE == 0U) {
1210               msgID = tmpMsgBuffer.msg.StdId;
1211            } else  {
1212                msgID = tmpMsgBuffer.msg.ExtId;
1213            }
1214
1215            /* Search for correct message in RX message array to check for callback */
1216            uint8_t rxMsg = 0;
1217            for (; rxMsg < length; rxMsg++) {
1218                if (msgRXstruct[rxMsg].ID == msgID) {
1219                    break;
1220                }
1221            }
1222
```

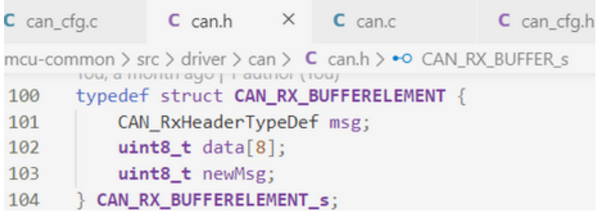
```
1223            /* Interpret received message */
1224            if (msgRXstruct[rxMsg].func != NULL) {
1225                msgRXstruct[rxMsg].func(msgID, &tmpMsgBuffer.data[0], tmpMsgBuffer.msg.DLC, tmpMsgBuffer.msg.RTR);
1226            } else {
1227                CAN_InterpretReceivedMsg(canNode, msgID, &tmpMsgBuffer.data[0], tmpMsgBuffer.msg.DLC,
                tmpMsgBuffer.msg.RTR);      See Line 1351
1228            }
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;
```



```
typedef struct CanPdu {
    uint8_t sdu[8];
    uint32_t id;
    uint8_t dlc;
} Can_PduType;          You, a month ago
```

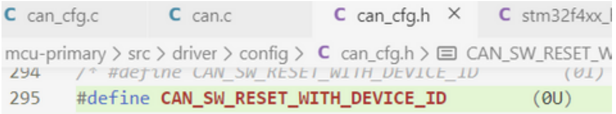


```
typedef struct CAN_RX_BUFFER {
    uint8_t ptrRead;
    uint8_t ptrWrite;
    uint8_t length;
    CAN_RX_BUFFERELEMENT_s* buffer;
} CAN_RX_BUFFER_s;      You, a month ago • Add all
```



```
typedef struct CAN_RX_BUFFERELEMENT {
    CAN_RxHeaderTypeDef msg;
    uint8_t data[8];
    uint8_t newMsg;
} CAN_RX_BUFFERELEMENT_s;
```

```c

        /* Move to next buffer element */
        can_rxbuffer->ptrRead++;
        can_rxbuffer->ptrRead = can_rxbuffer->ptrRead % can_rxbuffer->length;
        retVal = E_OK;
    }
#endif /* CAN0_USE_RX_BUFFER || CAN1_USE_RX_BUFFER */
    return retVal;
}

/**
 * @brief  Receives a bypassed CAN message and interprets it
 *
 * @param  canNode: canNode on which the message has been received
 * @param  msgID:   message ID
 * @param  data:    pointer to the message data
 * @param  DLC:     length of received data
 * @param  RTR:     RTR bit of received message
 *
 * @retval E_OK if interpreting was successful, otherwise E_NOT_OK
 */
static STD_RETURN_TYPE_e CAN_BufferBypass(CAN_NodeTypeDef_e canNode, uint32_t msgID, uint8_t* rxData, uint8_t DLC,
        uint8_t RTR) {
    STD_RETURN_TYPE_e retVal = E_OK;

    /* ***************************************************************
     *  Implement wished functionality of received messages here,
     *
     *  if no callback function in CAN_MSG_RX_TYPE_s struct is defined
     *****************************************************************/

    /* Perform SW reset */
    if (msgID == CAN_ID_SOFTWARE_RESET_MSG && DLC == 8) {
        uint8_t reset = 0;

        /* CAN data = FF FF FF FF FF FF FF FF */
        for (uint8_t i = 0; i < DLC; i++) {
            if (rxData[i] != 0xFF) {
                reset = 1;
            }
        }
#if CAN_SW_RESET_WITH_DEVICE_ID == 1
/*         CAN data = MCU Device ID Byte [0] [1] [2] [3] [4] [5] [6] [7] */
/*         if (rxData[0] == (uint8_t)mcu_unique_deviceID.off0 && data[1] == (uint8_t)(mcu_unique_deviceID.off0 >> 8) &&
                rxData[2] == (uint8_t)(mcu_unique_deviceID.off0 >> 16) && rxData[3] == (uint8_t)(mcu_unique_deviceID.off0 >> 24) &&
                rxData[4] == (uint8_t)mcu_unique_deviceID.off32 && rxData[5] == (uint8_t)(mcu_unique_deviceID.off32 >> 8) &&
                rxData[6] == (uint8_t)(mcu_unique_deviceID.off32 >> 16) && rxData[7] == (uint8_t)(mcu_unique_deviceID.off32 >> 24)) {
            reset = 1;
        }
```


C can_cfg.c
C can.c
C can_cfg.h ×
C stm32f4xx_l
mcu-primary > src > driver > config > C can_cfg.h > CAN_SW_RESET_W

```c
294    /* #define CAN_SW_RESET_WITH_DEVICE_ID         (01)
295    #define CAN_SW_RESET_WITH_DEVICE_ID         (0U)
```

```
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,                       Called in Line 1227.
1352          uint8_t RTR) {
1353      STD_RETURN_TYPE_e retVal = E_NOT_OK;
1354
1355
1356      /* ****************************************************************
1357       *  Implement wished functionality of received messages here,
1358       *
1359       *  if no callback function in CAN_MSG_RX_TYPE_s struct is defined
1360       ****************************************************************/
1361      return retVal;
1362  }
1363
1364  /* ****************************************
1365   *  Sleep mode
1366   ****************************************/
1367
1368  void CAN_SetSleepMode(CAN_NodeTypeDef_e canNode) {
1369      if (canNode  ==  CAN_NODE0) {
1370          HAL_CAN_RequestSleep(&hcan0);
1371      } else if (canNode  ==  CAN_NODE1) {
1372          HAL_CAN_RequestSleep(&hcan1);
```

```
    }
    return;
}


void CAN_WakeUp(CAN_NodeTypeDef_e canNode) {
    if (canNode  ==  CAN_NODE0) {
        HAL_CAN_WakeUp(&hcan0);
    } else if (canNode  ==  CAN_NODE1) {
        HAL_CAN_WakeUp(&hcan1);
    }
    return;
}

```