```
/**
 *
 * @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.h
 * @author  foxBMS Team
 * @date    12.07.2015 (date of creation)
 * @ingroup DRIVERS
 * @prefix  CAN
 *
 * @brief   Header for the driver for the CAN module
 *
 * Provides the interfaces for initialization, receive
 * and transmit handling
```

```
 *
 */

#ifndef CAN_H_
#define CAN_H_

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

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

#define CAN0_USE_TX_BUFFER       CAN0_USE_TRANSMIT_BUFFER
#define CAN0_USE_RX_BUFFER       CAN0_USE_RECEIVE_BUFFER

#define CAN0_TX_BUFFER_LENGTH    CAN0_TRANSMIT_BUFFER_LENGTH
#define CAN0_RX_BUFFER_LENGTH    CAN0_RECEIVE_BUFFER_LENGTH

#define CAN1_USE_TX_BUFFER       CAN1_USE_TRANSMIT_BUFFER
#define CAN1_USE_RX_BUFFER       CAN1_USE_RECEIVE_BUFFER

#define CAN1_TX_BUFFER_LENGTH    CAN1_TRANSMIT_BUFFER_LENGTH
#define CAN1_RX_BUFFER_LENGTH    CAN1_RECEIVE_BUFFER_LENGTH

typedef enum {
    CAN_ERROR_NONE = HAL_CAN_ERROR_NONE, /*!< No error              */
    CAN_ERROR_EWG = HAL_CAN_ERROR_EWG, /*!< EWG error             */
    CAN_ERROR_EPV = HAL_CAN_ERROR_EPV, /*!< EPV error             */
    CAN_ERROR_BOF = HAL_CAN_ERROR_BOF, /*!< BOF error             */
    CAN_ERROR_STF = HAL_CAN_ERROR_STF, /*!< Stuff error           */
    CAN_ERROR_FOR = HAL_CAN_ERROR_FOR, /*!< Form error            */
    CAN_ERROR_ACK = HAL_CAN_ERROR_ACK, /*!< Acknowledgment error */
    CAN_ERROR_BR = HAL_CAN_ERROR_BR, /*!< Bit recessive         */
    CAN_ERROR_BD = HAL_CAN_ERROR_BD, /*!< LEC dominant          */
    CAN_ERROR_CRC = HAL_CAN_ERROR_CRC, /*!< LEC transfer error   */
} CAN_ErrorTypeDef_e;

typedef enum {
    CAN_NODE1 = 0, /* CAN1 */
    CAN_NODE0 = 1, /* CAN0 */
} CAN_NodeTypeDef_e;

typedef struct CAN_ERROR {
    CAN_ErrorTypeDef_e canError;
    uint16_t canErrorCounter[23 - 1];  /* One slot for every error from CAN_Error_Code; */
                                       /* No space for NoError */
} CAN_ERROR_s;

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

Never used!

Boolean

Defined in stm32f4xx_hal_can.h

Why is this necessary? We only use a selected number errors, as defined in CAN_ErorTypeDef_e. Just wanted to be easy?

stm32f4xx_hal_can.h | cansignal_cfg.h | cansignal_cfg.c

mcu-hal > STM32F4xx_HAL_Driver > Inc > stm32f4xx_hal_can.h > __CAN

```
 */

/** @defgroup CAN_Error_Code CAN Error Code
  * @{
  */
#define HAL_CAN_ERROR_NONE            (0x00000000U)
#define HAL_CAN_ERROR_EWG             (0x00000001U)
#define HAL_CAN_ERROR_EPV             (0x00000002U)
#define HAL_CAN_ERROR_BOF             (0x00000004U)
#define HAL_CAN_ERROR_STF             (0x00000008U)
#define HAL_CAN_ERROR_FOR             (0x00000010U)
#define HAL_CAN_ERROR_ACK             (0x00000020U)
#define HAL_CAN_ERROR_BR              (0x00000040U)
#define HAL_CAN_ERROR_BD              (0x00000080U)
#define HAL_CAN_ERROR_CRC             (0x00000100U)
#define HAL_CAN_ERROR_RX_FOV0         (0x00000200U)
#define HAL_CAN_ERROR_RX_FOV1         (0x00000400U)
#define HAL_CAN_ERROR_TX_ALST0        (0x00000800U)
#define HAL_CAN_ERROR_TX_TERR0        (0x00001000U)
#define HAL_CAN_ERROR_TX_ALST1        (0x00002000U)
#define HAL_CAN_ERROR_TX_TERR1        (0x00004000U)
#define HAL_CAN_ERROR_TX_ALST2        (0x00008000U)
#define HAL_CAN_ERROR_TX_TERR2        (0x00010000U)
#define HAL_CAN_ERROR_TIMEOUT         (0x00020000U)
#define HAL_CAN_ERROR_NOT_INITIALIZED (0x00040000U)
#define HAL_CAN_ERROR_NOT_READY       (0x00080000U)
#define HAL_CAN_ERROR_NOT_STARTED     (0x00100000U)
#define HAL_CAN_ERROR_PARAM           (0x00200000U)

```

```
105
106 typedef struct CAN_TX_BUFFERELEMENT {
107     CAN_TxHeaderTypeDef msg;
108     uint8_t data[8];
109     uint8_t newMsg;
110 } CAN_TX_BUFFERELEMENT_s;
111
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;
118
119 typedef struct CAN_TX_BUFFER {
120     uint8_t ptrRead;
121     uint8_t ptrWrite;
122     uint8_t length;
123     CAN_TX_BUFFERELEMENT_s* buffer;
124 } CAN_TX_BUFFER_s;
125
126 /*================== Constant and Variable Definitions =======
127 /**
128  * @brief  CAN listen only transceiver mode of CAN node 0
129  */
130 extern uint8_t canNode0_listenonly_mode;
131
132 /**
133  * @brief  CAN listen only transceiver mode of CAN node 1
134  */
135 extern uint8_t canNode1_listenonly_mode;
136 /*================== Function Prototypes ==================================*/
137 /* Init */
138
139 /**
140  * @brief  Initializes CAN settings and message filtering
141  *
142  * @retval 0: if initialization successful, otherwise errorcode
143  */
144 extern uint32_t CAN_Init(void);
145
146 /* Interrupt handling */
147
148 /**
149  * @brief  Handles CAN TX interrupt request
150  * @param  ptrHcan: pointer to a CAN_HandleTypeDef structure that contains
151  *         the configuration information for the specified CAN.
152  * @retval none (void)
153  */
154 extern void CAN_TX_IRQHandler(CAN_HandleTypeDef* ptrHcan);
155
156 /**
```

Boolean

Pointer to read in a circular buffer?

stm32f4xx_hal_can.h | Search: can_buffe... | cansignal_cfg.h

mcu-hal > STM32F4xx_HAL_Driver > Inc > stm32f4xx_hal_can.h > CAN_RxHe

```
167   */
      You, a month ago | 1 author (You)
168 typedef struct
169 {
170   uint32_t StdId;    /*!< Specifies the standard identif
171                           This parameter must be a numbe
172
173   uint32_t ExtId;    /*!< Specifies the extended identif
174                           This parameter must be a numbe
175
176   uint32_t IDE;      /*!< Specifies the type of identifi
177                           This parameter can be a value
178
179   uint32_t RTR;      /*!< Specifies the type of frame fo
180                           This parameter can be a value
181
182   uint32_t DLC;      /*!< Specifies the length of the fr
183                           This parameter must be a numbe
184
185   FunctionalState TransmitGlobalTime; /*!< Specifies whe
186                                          of frame transmission, is sent
187                                          @note: Time Triggered Communic
188                                          @note: DLC must be programmed
189                                          This parameter can be set to E
190
191 } CAN_TxHeaderTypeDef;
```

A huge waste of memory with these type defines.

```
157   * @brief  Handles CAN RX interrupt request
158   *
159   * @param  canNode: canNode that received a message
160   * @param  ptrHcan: pointer to a CAN_HandleTypeDef structure that contains
161   *         the configuration information for the specified CAN.
162   *
163   * @retval None
164   */
165  extern void CAN_RX_IRQHandler(CAN_NodeTypeDef_e canNode, CAN_HandleTypeDef* ptrHcan);
166
167  /**
168   * @brief   Handles CAN error interrupt request
169   *
170   * @param   canNode:
171   * @param   ptrHcan: pointer to a CAN_HandleTypeDef structure that contains
172   *          the configuration information for the specified CAN.
173   *
174   * @retval  None
175   */
176  extern void CAN_Error_IRQHandler(CAN_NodeTypeDef_e canNode, CAN_HandleTypeDef* ptrHcan);
177
178  /* Transmit Message */
179
180  /**
181   * @brief  Transmits message directly on the CAN bus
182   *
183   * @param  canNode: canNode on which the message shall be transmitted
184   * @param  msgID:    ID of the message that will be transmitted
185   * @param  ptrMsgData:    pointer to the data that shall be transmitted
186   * @param  msgLength:    Specifies the data length
187   * @param  RTR: Specifies the type of frame for the message that will be transmitted.
188   *
189   * @retval E_OK if transmission successful, otherwise E_NOT_OK
190   */
191  extern STD_RETURN_TYPE_e CAN_TxMsg(CAN_NodeTypeDef_e canNode, uint32_t msgID, uint8_t* ptrMsgData,
192          uint32_t msgLength, uint32_t RTR);
193
194  /**
195   * @brief  Add message to transmit buffer, message will be transmitted shortly after.
196   *
197   *         ------------------------ IMPORTANT!!!! -------------------------------
198   *         Make sure that this function is not interrupted by the operating system
199   *         during its execution.
200   *
201   * @param  canNode: canNode on which the message shall be transmitted
202   * @param  msgID:    ID of the message that will be transmitted
203   * @param  ptrMsgData:    pointer to a uint8_t array that contains the message that will be transmitted
204   * @param  msgLength:    length of the message that will be transmitted
205   *                 This parameter can be a value of CAN_identifier_type.
206   * @param  RTR     Specifies the type of frame for the message that will be transmitted.
207   *                 This parameter can be a value of CAN_remote_transmission_request
208   *
```

```
209     * @retval E_OK if successful, E_NOT_OK if buffer is full or error occurred
210     */
211    extern STD_RETURN_TYPE_e CAN_Send(CAN_NodeTypeDef_e canNode, uint32_t msgID, uint8_t* ptrMsgData,
212            uint32_t msgLength, uint32_t RTR);
213
214    /**
215     * @brief  Transmits a can message from transmit buffer
216     *
217     *         ------------------------ IMPORTANT!!!! -------------------------------
218     *         Make sure that this function is not interrupted by the operating system
219     *         during its execution.
220     *
221     * @param canNode:  canNode on which the message shall be transmitted
222     *
223     * @retval E_OK if transmission successful, otherwise E_NOT_OK
224     */
225    extern STD_RETURN_TYPE_e CAN_TxMsgBuffer(CAN_NodeTypeDef_e canNode);
226
227    /* Read Message */
228
229    /**
230     * @brief  Reads a can message from RxBuffer
231     *
232     * @param canNode canNode on which a message has been received
233     * @param msg     message that has been received
234     *
235     * @retval E_OK if reception successful, if buffer empty or invalid pointer E_NOT_OK
236     */
237    extern STD_RETURN_TYPE_e CAN_ReceiveBuffer(CAN_NodeTypeDef_e canNode, Can_PduType* msg);
238
239    /* Sleep mode */
240
241    /**
242     * @brief  Set CAN to sleep mode
243     *
244     * @param canNode  canNode which shall be put to sleep mode
245     *
246     * @retval none (void)
247     */
248    extern void CAN_SetSleepMode(CAN_NodeTypeDef_e canNode);
249
250    /**
251     * @brief  Wake CAN up from sleep mode
252     *
253     * @param canNode  canNode which shall be waken up from sleep mode
254     *
255     * @retval none.
256     */
257    extern void CAN_WakeUp(CAN_NodeTypeDef_e canNode);
258
259    /*================== Function Implementations ============================*/
260
```

```
/* ************************************
 *  CAN Fault Confinement
 ************************************/

/* Fault confinement is a checking mechanism that makes it possible to distinguish between short disturbances
 (e.g. switching noise from a nearby power cable couples into the transmission media) and permanent failures
 (e.g. a node is malfunctioning and disturbs the bus).

 Manipulation of the error counters is asymmetric. On a successful transmission, or reception, of a message,
 the respective error counter is decremented if it had not been at zero. In the case of a transmit or receive
 error the counters are incremented, but by a value greater than the value they would be decrement by following
 a successful message transaction.

 If a node detects a local error condition (e.g. due to local conducted noise, application software, etc.),
 its resulting error flag (primary error flag) will subsequently cause all other nodes to respond with an error
 flag too (secondary error flags). It is important that a distinction is made between the nodes that detected an
 error first and the nodes which responded to the primary error flag. If a node transmits an active error frame,
 and it monitors a dominant bit after the sixth bit of its error flag, it considers itself as the node that has
 detected the error first. In the case where a node detects errors first too often, it is regarded as malfunctioning,
 and its impact to the network has to be limited. Therefore, a node can be in one of three possible error states:

 ERROR ACTIVE - Both of its error counters are less than 128. It takes part fully in bus communication and signals
 an error by transmission of an active error frame.This consists of sequence of 6 dominant bits followed
 by 8 recessive bits, all other nodes respond with the appropriate error flag, in response to the
 violation of the bit stuffing rule.

 ERROR PASSIVE - A node goes into error passive state if at least one of its error counters is greater than 127. It
 still takes part in bus activities, but it sends a passive error frame only, on errors. Furthermore,
 an error passive node has to wait an additional time (Suspend Transmission Field, 8 recessive bits after
 Intermission Field) after transmission of a message, before it can initiate a new data transfer. The
 primary passive error flag consists of 6 passive bits and thus is transparent on the bus and will
 not jam communications.

 BUS OFF - If the Transmit Error Counter of a CAN controller exceeds 255, it goes into the bus off state. It is
 disconnected from the bus (using internal logic) and does not take part in bus activities anymore. In order
 to reconnect the protocol controller, a so-called Bus Off recovery sequence has to be executed. This usually
 involves the re-initialization and configuration of the CAN controller by the host system, after which it
 will wait for 128 * 11 recessive bit times before it commences communication.

 */

/* ************************************
 *  CAN Error Confinement Rules
 ************************************/

/* REC: Receive Error Counter, TEC: Transmit Error Counter */

/* - When a receiver detects an error, the REC will be increased by 1, except when the detected error was a Bit Error
 during the sending of an Active error Flag or an Overload Flag.

 - When a receiver detects a dominant bit as the first bit after sending an Error Flag, the REC will be increased by 8.

```

```
313    - When a transmitter sends an Error Flag, the TEC is increased by 8.
314    Exception 1: If the transmitter is Error Passive and detects an ACK Error because of not detecting a dominant ACK
315    and does not detect a dominant bit while sending its Passive Error Flag.
316    Exception 2: If the transmitter sends an Error Flag because a Stuff Error occurred during arbitration, and should
317    have been recessive, and has been sent as recessive but monitored as dominant.
318
319    - If the transmitter detects a Bit Error while sending an Active Error Flag or an Overload Frame, the TEC is
       increased by 8.
320
321    - If a receiver detects a Bit Error while sending an Active Error Flag or an Overload Flag, the REC is increased by 8.
322
323    - Any node tolerates up to 7 consecutive dominant bits after sending an Active Error Flag, Passive Error Flag or
       Overload Flag.
324    After detecting the fourteenth consecutive dominant bit (in case of an Active Error Flag or an Overload Flag) or after
325    detecting the eighth consecutive dominant bit following a Passive Error Flag, and after each sequence of additional
       eight
326    consecutive dominant bits, ever y transmitter increases its TEC by 8 and every receiver increases its REC by 8.
327
328    - After successful transmission of a frame (getting ACK and no error until EOF is finished), the TEC is decreased by
       1 unless it was already 0.
329
330    - After the successful reception of a frame (reception without error up to the ACK Slot and the successful sending
       of the ACK bit),
331    the REC is decreased by 1, if it was between 1 and 127. If the REC was 0, it stays 0, and if it was greater than
       127, then it
332    will be set to a value between 119 and 127.
333
334    - A node is Error Passive when the TEC equals or exceeds 128, or when the REC equals or exceeds 128. An error
       condition
335    letting a node become Error Passive causes the node to send an Active Error Flag.
336
337    - A node is Bus Off when the TEC is greater than or equal to 256.
338
339    - An Error Passive node becomes Error Active again when both the TEC and the REC are less than or equal to 127.
340
341    - A node which is Bus Off is permitted to become Error Active (no longer Bus Off) with its error counters both set
       to 0
342    after 128 occurrence of 11 consecutive recessive bits have been monitored on the bus.
343    */
344
345   #endif /* CAN_H_ */
346
```