DELTAV

UZAY TEKNOLOJİLERİ A.Ş.

REFERENCE MANUEL

FOR CAN BUS PROTOCOL

AUTHOR : ORHAN AĞIRMAN

CREATED ON : 07.09.2021

CAN (CONTROL AREA NETWORK) BUS USING STM32-NUCLEO BOARDS

WHAT IS THE CAN-BUS?

- It was developed by Robert BOSCH in the 1980s to provide software-controlled data transfer from a cable instead of many cables in the automotive industry.
- CAN is the most widely used bus protocol in many fields from all kinds of vehicles to industrial products, especially automobiles.
- It is used in real-time applications where security is paramount.
- The application range is wide from high-speed networks to low-cost multi-wiring systems.
- CANBUS provides a maximum of 1Mbit/sec news data communication in application areas such as automobile electronics, smart motor control, robot control, smart sensors, elevators, machine control units, anti-slip systems, traffic signaling systems, smart buildings, and laboratory automation.
- While the communication speed is 1Mbit/sec at 40m, it drops to 40Kbit/sec at 1km distances.
- Unlike other protocols, CAN works message-based, not address-based. Each message has a unique ID number. Messages are transmitted with frames.

GENERAL FEATURES OF CAN BUS SYSTEM

- Message Priority, Lost Time Security
- Configuration Flexibility
- Multiple acceptance with synchronization: The same data can be received by many units.
- Ability to remove data density in the system
- Multi-master operation
- Error detection and generation of error related signals
- Automatic resending of the message at a time when the transmission line (BUS) of the message is empty, in case of an error in sending the message
- Ability to distinguish between temporary and permanent faults in the units and automatically turn off permanent faulty units.

CAN BUS is divided into three layers.

> Object Layer:

- To determine which message will be transferred,
- Deciding which message to receive at the transmission layer,
- To provide interface to hardware related application

> Transfer Layer:

- The main task of the transmission layer is the transfer protocol. For example, frame control, message prioritization, error checking, error signaling, and error shutdown.
- The transmission layer makes sure that the transmission line is free before sending a new message. It is also responsible for receiving data from the transmission line.
- It also considers some parameters of bit timing during data transfer for synchronous communication.
- All system components that communicate via CANBUS are called nodes.

> Physical Layer:

- The physical layer is the entire electrical part during data communication between nodes.

HOW CAN WORKS?

In the CANBUS system, all units have the right to send data to the transmission line with equal priority. This is called a multi-master operation.

All nodes always listen to the transmission line and try to catch the moment when the transmission line is empty. The node that sees the line empty sends its data. Although all nodes have equal priority to send messages, the truth of the matter is not. This is because CAN is a message priority system. In the Internet system PCs (wiring) are numbered, but in CANBUS messages are numbered, not nodes.

CAN uses CSMA/CD with bit priority structure as the method of accessing the communication medium. While this method ensures that messages do not collide, it limits the length of the communication line. Therefore, CAN nodes can be connected via a bus of 40m with a data communication rate of 1 Mbit/s and a bus of 1000m with a data rate of 40 Kbit/s.

The CAN system takes its secure communication power from the CMSA/CD (Carrier Serve Multiple Access With Collision Detection) structures. Role of CMSA/CD with an example;

In a CAN communication, each node has to listen to the transmission medium. If a node wants to transfer data to the transmission medium, it first listens to the road, and if the road is empty, it transmits its data to the road. Sometimes, due to reasons arising from long distances, while a node transmits data to the road, the other node at a distance does not understand that the message is on the way, so it leaves its own message assuming the road is empty. Thus, the collision occurs. It detects a collision in two nodes and delays data transfer requests in both nodes for a while. They then try to resend the data when the transmission medium is empty.

All data sent to CANBUS is received by all nodes. With the help of filters in each node, it receives messages that interest it and discards those that do not.

Generally, a CAN Network topology is shown below.

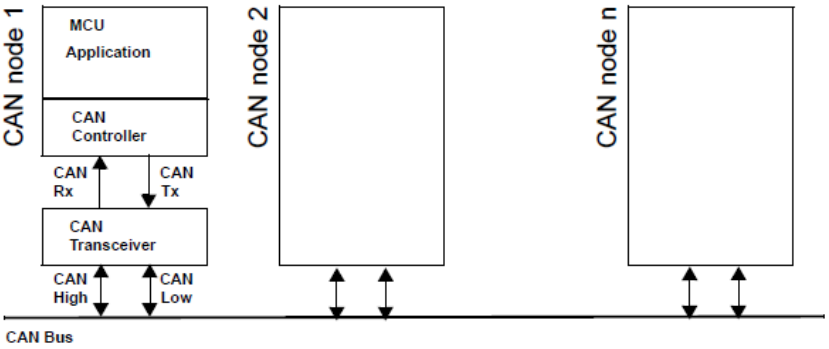

Figure 1: CAN Network topology

Two STM32F439ZIT Nucleo boards and one STM32F446ZET Nucleo board were used in the project. In the Nucleo boards used in the project, this CAN controller is included in the chip, but it can also be used as an external integrated. The CAN controller is directly connected to the CAN bus. This bus is a two-wire bus terminated by 120-ohm resistors on both sides.

As can be seen in Figure-1, CAN_Rx and CAN_Tx pins generate digital signals, while CAN_High and CAN_Low pins generate differential signals.

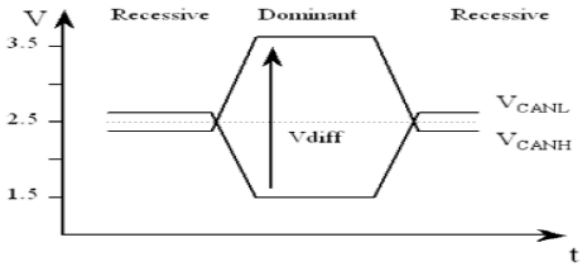

Figure 2: CAN_High & CAN_Low Signals

As seen in figure-2 above, the logic level of the line can take 2 different values and the level of these values is visible.

Logic 1 is called recessive, logic 0 is called dominant. The reason for this is that if 0 and 1 are written at the same time from different nodes on the line, 0 will override 1 against.

As a result of logic 0 dominating logic 1, messages with small message IDs take priority. When a node decides to send a message, the message is held until the path is free. Every node follows the path continuously. After the path is empty, the node starts sending the message by giving the start path sign.

The message reaches every node and the related nodes read and process the message. If more than one node starts to write a message to the path when the path is empty, the node that wrote the message with the low ID captures the path and the other nodes wait for the path to be free to resend by pulling out of the way.

If a node writes a message to the bus and reads a 0 when it writes a 1, it understands that another node has written a message to the bus and leaves the bus to it because its priority is high and it tries to send again when the path is empty.

An example is shown in the figure-3 below.

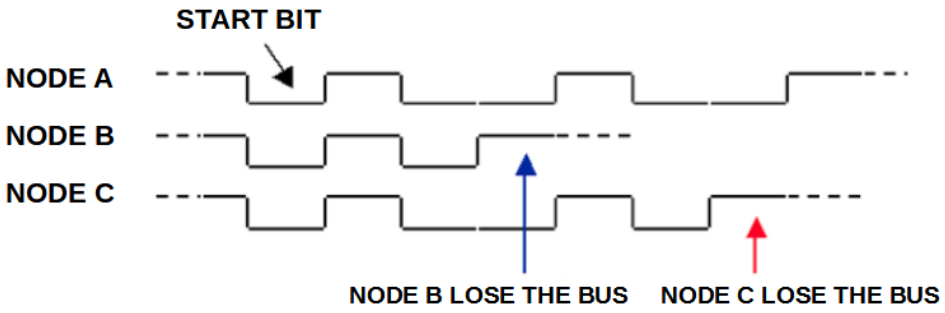

Figure 3: An example of Nodes priority for CAN BUS

THE CAN FRAME

In CAN systems, data is transmitted in packets. However, there are two types of packets and these packets have special names.

Those with 11-bit identifiers are called CAN2.0A, also known as STANDARD CAN, and those with 29-bit identifiers are called CAN2.0B, also known as EXTENDED CAN. The main difference between them is the number of messages to be defined.

- In standard CAN, 2^{11} = 2048 messages are defined.
- In Extended CAN, 2^{29} = 536 870 912 messages are defined.

The field where this information is kept is called the message-id field. The number here is taken into account in defining the message priority.

On the data path, messages are transmitted by dividing them into frames. There are two different types of messages.

These are Message Frame and Remote Transmit Request Frame.

The difference is that the Message Frames can carry data up to 8 bytes in length, while the data of a certain message is requested in the Remote Transmit Request Frames.

STANDARD FRAME:

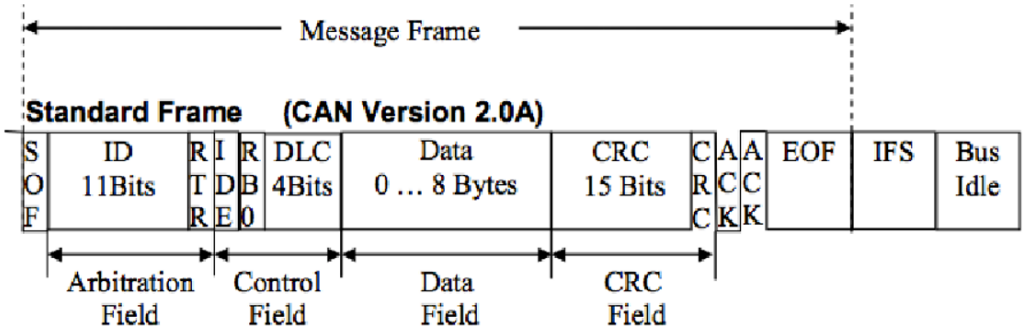

Figure 4: STANDARD FRAME

SOF	Start of Frame	DLC	Data Length Code	EOF	End of Frame
RTR	Remote Transmission Request	CRC	Cycle Redundancy check	RB0	Reserved Bit
IDE	Identifier extension	ACK	Receivers Acknowledgement	IFS	Interframe Space

Arbitration Field

Each frame starts with the SOF (Start of Frame) signal. This signal is 1-bit and dominant.

The SOF is followed by the 12-bit arbitration field. The first 11 bits are the message ID field and the messages are labeled with the ID value in this field.

The last bit in the arbitration field is called RTR and has a special meaning.

- If this bit is 0 (dominant), the sent frame is the message frame and the data field contains the data of the message defined in the ID field.
- If this bit is 1, the frame is the request frame and there is no data field. The data of the message determined by the value in the ID field of this frame is requested from the relevant nodes. The receiver who receives this frame reads the value in the ID field, understands what data to send, and sends it when the path is empty.

In this way, CAN protocol can work as master and slave. The first 7 bits in the ID field cannot be consecutively recessive.

Control Field

After the Arbitration Field comes the Control Field.

The first bit of this field is called IDE and is a dominant bit, indicating that this frame is a 2.0A frame with an 11-bit ID field.

This bit is followed by a one-bit unused reserved space. Next comes an area called 4-bit DLC. DLC field tells how many bytes of data are sent.

Data Field

The Control Field is followed by the Data Field. The Data Field can be up to 8 bytes.

CRC Field

The Data Field is followed by the CRC Field. This field is 16-bit and consists of 15-bit CRC (Cyclic Redundancy Check) information and a recessive CRC Delimiter bit. The CRC field is a value for understanding whether the data between the SOF field and the CRC field is correct.

The node sending the data calculates the 15-bit CRC value by performing some operations on the data and adds it to the frame.

When the receiving node receives the data, it performs the same operations as the sender and recalculates the CRC. If the received and sent CRC is consistent, the data was sent correctly. If at least 1 of the receiving nodes has received the data incorrectly, the data must be sent again.

ACK Field

The CRC field is followed by the ACK field. This field is 2 bits.

The sender sends its first bit recessively. If the data is received correctly by at least one receiver, the receiver writes the dominant bit to the path. This way the sender knows that the message has been received by at least one recipient. If the sender cannot read the dominant bit, it considers that there is an error due to the ACK signal and sends the data again.

The second bit of this field is called the ACK delimiter and is recessive.

If no information is received from any node during the Acknowledgment Information process, an Error Frame indicating that an error has occurred due to an ACK error is generated and the sender tries to resend.

If the sender has sent the Request Frame, the receiver also sends its reply to the sender at an idle moment of the transmission line.

EOF Field

The ACK field is followed by the 7-bit EOF field, indicating that the frame has been terminated. Lice in this area are recessive. Then comes the 3-bit IFS field to leave space between the frames and its bits are recessive. Thus, a message is sent and received in the basic frame.

EXTENDED FRAME:

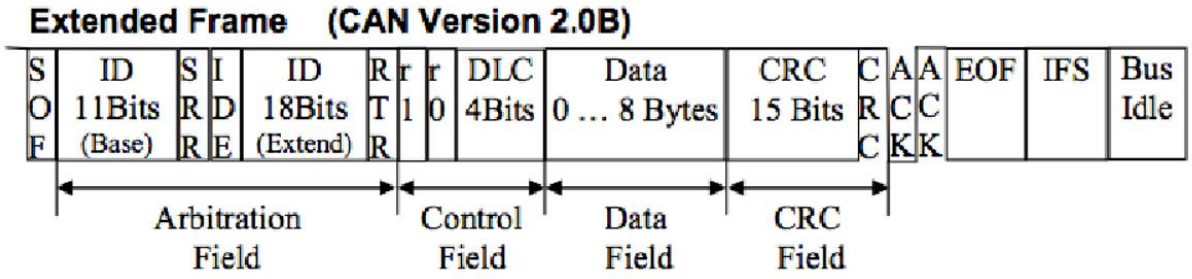

Figure 5: EXTENDED FRAME

In CAN2.0B, the message ID is 29 bits. It was developed with 2.0A in mind while developing CAN2.0B in terms of backward compatibility. Both protocols can work in the same way, but messages with 29-bit ID are not sent to 2.0A nodes.

Arbitration Field

The extended frame starts with a dominant SOF. Then comes the 11-bit ID Base field, as in 2.0A. Then the dominant RTR bit in 2.0 is replaced by the recessive SRR bit. Then comes the IDE bit, corresponding to the offset at 2.0A. The only difference is that in 2.0B this bit is recessive because messages with a 29-bit ID are transmitted. After the IDE bit comes to the second 18-bit ID Extended field. Then the dominant RTR bit comes in, indicating that this frame is a data frame.

Control Field

After the arbitration field comes the control field. The first two bits of the control field are reserved and not used. The last 4 bits form the DLC field and tell how many bytes of the data are sent.

Data Field

The Control Field is followed by the Data Field. The Data Field can be up to 8 bytes.

CRC Field

The Data Field is followed by the CRC Field. This field is 16-bit and consists of 15-bit CRC (Cyclic Redundancy Check) information and a recessive CRC Delimiter bit. The CRC field is a value for understanding whether the data between the SOF field and the CRC field is correct.

The node sending the data calculates the 15-bit CRC value by performing some operations on the data and adds it to the frame.

When the receiving node receives the data, it performs the same operations as the sender and recalculates the CRC. If the received and sent CRC is consistent, the data was sent correctly. If at least 1 of the receiving nodes has received the data incorrectly, the data must be sent again.

ACK Field

The CRC field is followed by the ACK field. This field is 2 bits.

The sender sends its first bit recessively. If the data is received correctly by at least one receiver, the receiver writes the dominant bit to the path. This way the sender knows that the message has been received by at least one recipient. If the sender cannot read the dominant bit, it considers that there is an error due to the ACK signal and sends the data again.

The second bit of this field is called the ACK delimiter and is recessive.

If no information is received from any node during the Acknowledgment Information process, an Error Frame indicating that an error has occurred due to an ACK error is generated and the sender tries to resend.

If the sender has sent the Request Frame, the receiver also sends its reply to the sender at an idle moment of the transmission line.

EOF Field

The ACK field is followed by the 7-bit EOF field, indicating that the frame has been terminated. Lice in this area are recessive. Then comes the 3-bit IFS field to leave space between the frames and its bits are recessive. Thus, a message is sent and received in the basic frame.

REQUEST FRAME:

Most of the time, the data bus works by sending the information that is read and formed. Sometimes nodes make requests. They do this with the request framework. The request framework has 2 differences. In these request frames, the RTR bit is recessive and request frames have no data field. The remaining parts are the same as the message frame.

ERROR FRAME:

An error frame is sent when an error occurs on the data path. The error frame consists of two fields. These fields are the error flags and the error delimiter field. There are two error flags, active and passive. The active or passive fault is generated according to the state of the bus.

THE CAN BIT TIME

Unlike most other serial protocols, in the CAN protocol, the bitrate is not set directly by setting up the baud rate pre-divider.

CAN hardware has a baud rate pre-divider, but it is used to generate a small time slot called quanta. The one-bit duration is divided into 3 parts.

The first part is the synchronization part and is fixedly one quantum long. The following segments are named Tseg1 and Tseg2 and their length can be adjusted by the user in quanta.

A one-bit period must be a minimum of 8 and a maximum of 25 quanta. The point at which the sent bit is received at the receiver is called the sampling point and is at the end of Tseg1.

By adjusting the ratio of Tseg1 and Tseg2, the sampling point can be shifted in one-bit time. The purpose of doing this is to ensure the stable operation of the system according to the length of the transmission line.

If we are using long transmission lines, the sampling point must be retracted. If our oscillator is not sensitive and has low precision, the sampling point is shifted forward. Additionally, receivers can be locked to the transmitter by adjusting their bit timings. This compensates for minor deviations in the bitrate of the transmitter.

Each bit is set by a user-adjustable variable called synchronous jump width, which takes values between 14 quanta times. The bit rate is calculated with the following relation (BRP=Baud Rate Prescaler)

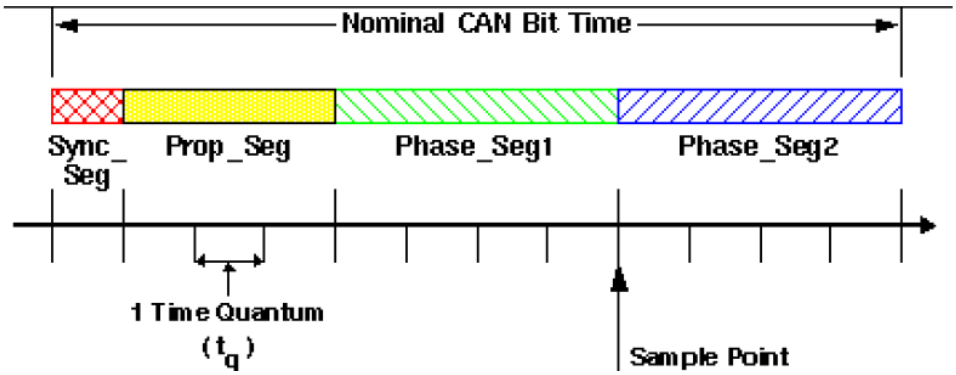

Figure 6: CAN Bit Time

The following formula can be used to set the duration of 1 time quantum.

$$\text{The duration of 1 time quantum} = \frac{Prescaler}{PCLK1}$$

For example, if the PLCK1 of 42 Mhz and the prescaler is 16, the duration of 1 time quantum will be 380.95 ns.

$$Prog\_Seg + Phase\_Seg1 = Bit\ Segment\ 1$$

$$Phase\_Seg2 = Bit\ Segment\ 2$$

Sync_Seg must be equal to 1 time quantum.

$$Time\ for\ one\ bit = Sync\_Seg + \ Bit\ Segment\ 1 + Bit\ Segment\ 2$$

The sample point can be found with the formula below.

$$Sample\ point\ \% = \frac{1 + \ Bit\ Segment\ 1}{1 + \ Bit\ Segment\ 1 + Bit\ Segment\ 2}\ x\ 100$$

For sample point 87.5 % is the preferred value used by CANopen and DeviceNet, 75 % is the default value for ARINC 825.

Baud Rate can be found using the formula below.

$$Baud\ Rate = \frac{PCLK1}{Prescaler * (1 + Bit\ Segment\ 1 + Bit\ Segment\ 2)}$$

CAN BUS supports baud rate up to 1 Mbit/s.

THE CAN ERROR PREVENTION

The CAN protocol has five error prevention methods. When an error occurs, the sender resends the data so that the processor does not need to intervene.

3 of the error prevention methods are at the frame level and 2 of them are at the bit level. The ones at the frame level are the frame format check, the CRC check, and the ACK (acknowledgment) check. The bitwise ones are the bit control and bit stuffing control.

The first method of error prevention at the framework level is, after receiving the data, the receiver checks the format of the data and compares it with the frame structure. If there is a missing field in the received data, the data is rejected and an error frame is left on the bus. This system ensures data retrieval in the correct format.

The second method of error prevention at the frame level is CRC checking. The CRC code is generated by processing the bits from the SOF bit to the beginning of the CRC bits. At the receiver, this CRC code is compared with the received data and it is tested whether the received bits are correct. After the format check, the receiver checks the bits and prevents messages that match the format but are wrong.

The final error check at the frame level is that the ACK message does not reach the sender. The ACK bit means that the receiver acknowledges the received message. After the sender sends the CRC bits, it sends the ACK bit recessively. At least one of the receivers is expected to override the recessive ACK bit on the line with the dominant bit. If the ACK bit has not reached the sender as a result of the timeout, it is interpreted as an error in the ACK bit. As a result, the sender fails and sends the same message again until it receives the ACK acknowledgment.

The first of the bitwise errors is the bit stuffing error. No clock pulses are sent between the sender and receiver. Instead, synchronization is achieved with the logic changes on the CANL and CANH lines on the bus.

As a result, the succession of more than 5 bits from the same logic level (dominant) means that the synchronization is broken and causes an error in the receiver. To prevent this, the sender continues the communication by sending a bit from the opposite level after the same 5 levels.

This event is illustrated in the figure below.

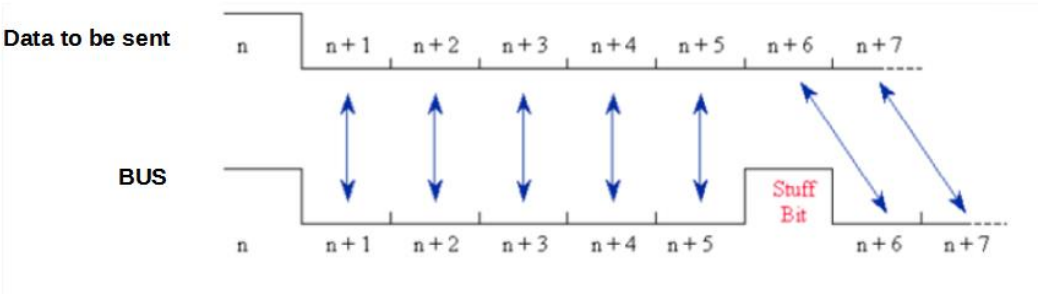

Figure 7: Stuffing Error

As a result, when a node wants to generate an error message at any time, it writes 6 dominant bits to the data bus and causes an error.

The other bitwise error is the bit check. When the bus is empty, the nodes take over the bus according to the IDs of the messages to send messages. Each node reads back the bit it wrote to the bus to see if there is a more important message than the one it sent. If there is a more important message, it pulls back and waits for the bus to clear. Thus, a node gains on the bus and continues to send the bits. It also reads back the bits it sends. If the node that wins the bus reads a different level than the one it sent, it generates an error.

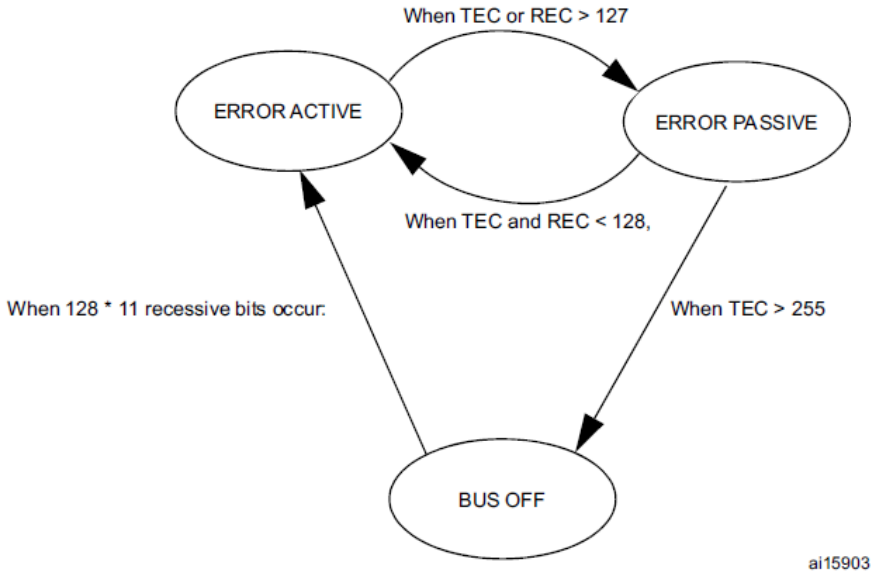

Figure 8: CAN Error State Diagram

CAN hardware switches between error state according to the errors that occur.

There are two error counters. They count errors generated and errors received in the sender. If any counter reaches a value greater than 127, the hardware enters passive fault mode. In this mode, it continues to respond to incoming error frames, but when it fails, it sends recessive bits instead of dominant bits. If the transmission error number exceeds 255, the hardware is turned off and the communication on the line is not interfered with or affected. To restart communication, the processor must intervene. These mechanisms are meant to keep the bus busy by repeatedly sending error messages when the node goes into a faulty state.

THE CAN IN STM32

STM32 series microcontrollers process incoming messages at the hardware level and only messages that pass the filter are forwarded to the mailbox. This will reduce the load on the processor and allow us to not pay attention to the extra interrupt. The basic extended CAN (bxCAN) bit is the CAN network interface. Supports CAN 2.0A and 2.0B versions. It is designed to handle large numbers of incoming messages with maximum efficiency and minimum CPU usage. It is also responsible for the priority conditions for the transfer of messages.

The working method of bxCAN is given below.

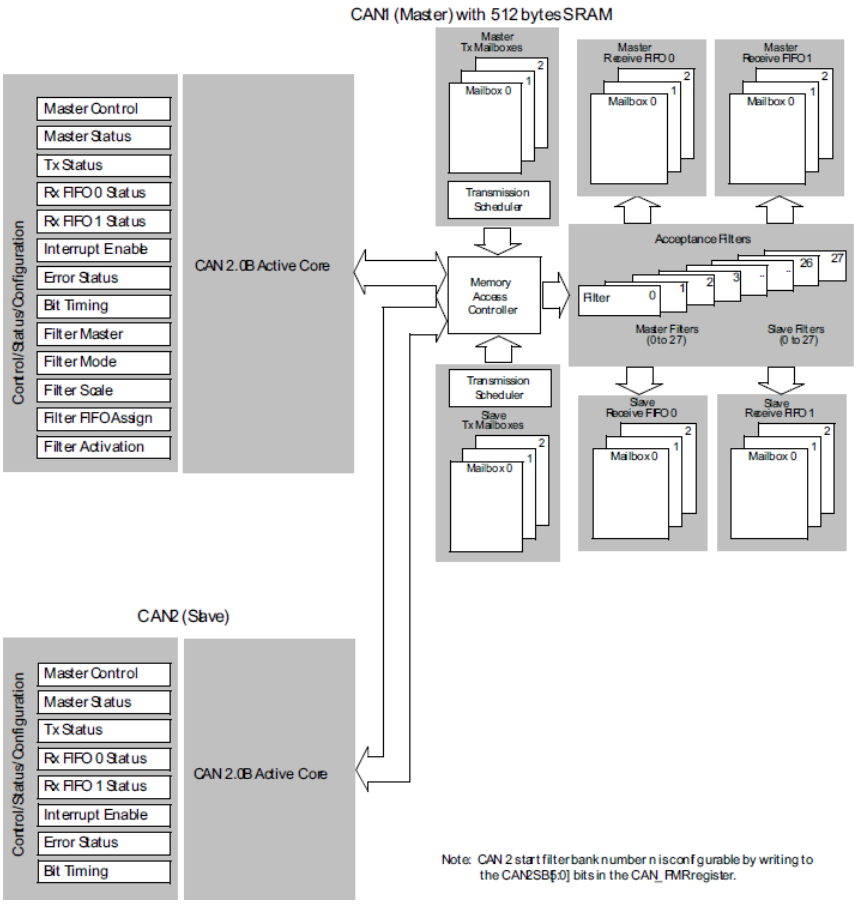

Figure 9: Dual CAN Block Diagram

Three mailboxes arranged as FIFO are provided for the reception of CAN messages.

FIFO is fully managed by hardware to save CPU load, simplify software and guarantee data consistency.

The application accesses messages stored in FIFO through the FIFO output mailbox. A received message is considered valid when received correctly according to the CAN protocol (no error until the last of the EOF field) and successfully passes descriptive filtering.

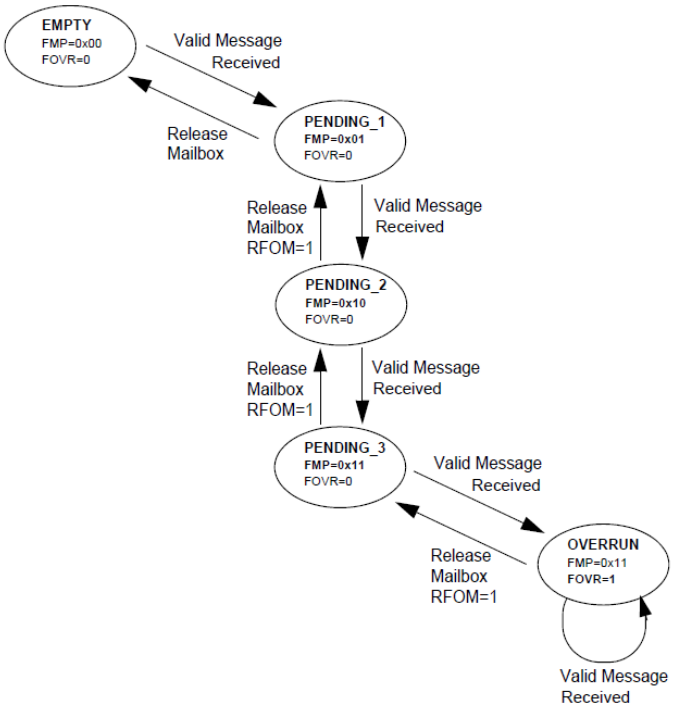

Figure 10: Receive FIFO States

THE CAN MESSAGE FILTERING

One of the conditions for receiving a message and sending it to the mailbox is to pass the message identifier through a filter. In the CAN protocol, the identifier of a message is not associated with the address of a node, but with the content of the message. As a result, a transmitter broadcasts its message to all receivers.

At message reception, a receiving node decides whether the software needs the message based on the identifier value.

If the message is required, it is copied to SRAM. Otherwise, the message should be discarded without interfering with the software. To fulfill this requirement, the bxCAN Controller provides the application with 28 configurable and scalable filter banks (0-27).

This hardware filtering saves CPU resources that would be required by the software to perform filtering. Each filter bank x (0-27) consists of two 32-bit registers CAN_FxR0 and CAN_FxR1.

Each filter bank can be scaled independently to optimize and adapt the filters to the needs of the applications.

Depending on the scale filter, the filter bank provides:

- STDID [10: 0], EXTID [17: 0], a 32-bit filter for IDE and RTR bits;
- Two 16-bit filters for STDID [10:0], RTR, IDE and EXTID [17:15] bits.

The IDE bit (Identifier Expansion Bit) means that the filter is designed for the extended message frame and the Remote Transmission Request (RTR) bit indicates the "Remote" message type. Additionally, filters can be configured in mask mode or list mode.

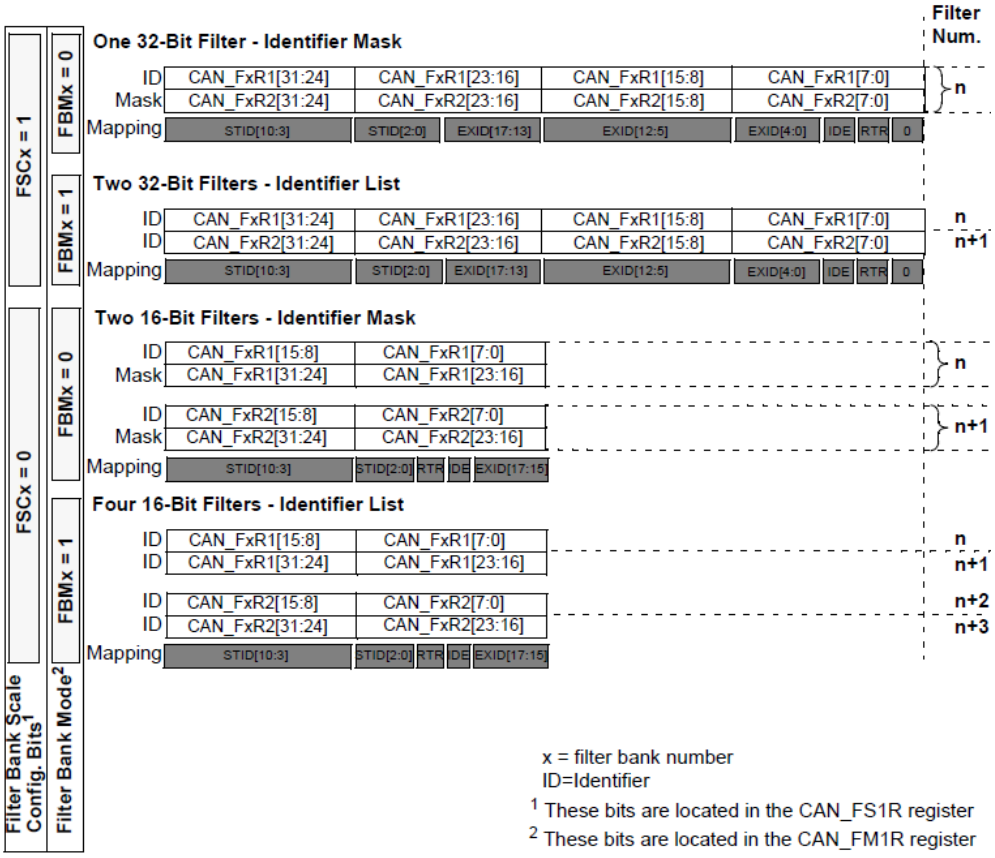

Figure 11: Filter bank scale configuration - register organization

The CAN-BUS unit contains firmware to perform this task, using the accept filter and mask value to filter out unwanted messages.

The filter mask is used to determine which bits in the descriptor of the received frame to compare.

- If a mask bit is set to zero (0x0000), the incoming ID bit is automatically received regardless of the filter bit.
- If a mask bit is set to one (0xFFFF), the corresponding ID bit is compared with the filter bit. If there is a match, it is accepted, otherwise, the frame is rejected.

For example, if we want to receive only frames containing the ID 00001234, we should set the mask to 1FFFFFFF.

– Filter: 0x00001234

– Mask: 0x1FFFFFFF

When a frame arrives, its ID is compared with the filter and all bits must match (ie all bits are compared one by one in order). Any frame that does not match the ID 00001234 is rejected.

For example, if we want to receive frames with IDs between 00001230 and 0000123F, we should set the mask to 1FFFFFF0.

– Filter: 0x00001230

– Mask: 0x1FFFFFF0

If we only want to accept frames between 00001230 and 00001237, we must set the mask to 1FFFFFF8.

– Filter 0x00001230

– Mask: 0x1FFFFFF8

When a frame arrives, its ID is compared with the filter and all bits except the 0bit must match, and also the 0bit must be between 0 and 8.

For example, if we want any incoming frame to be accepted, we should set the filter and mask to zero.

– Filter: 0x0000 / 0

– Mask: 0x0000 / 0

All frames are accepted at the end of the transaction.

CONFIGURATION OF STM32CUBEMX

First of all, the clock configuration of STM32 Nucleo-F439ZIT and F446ZET boards should be done as follows.

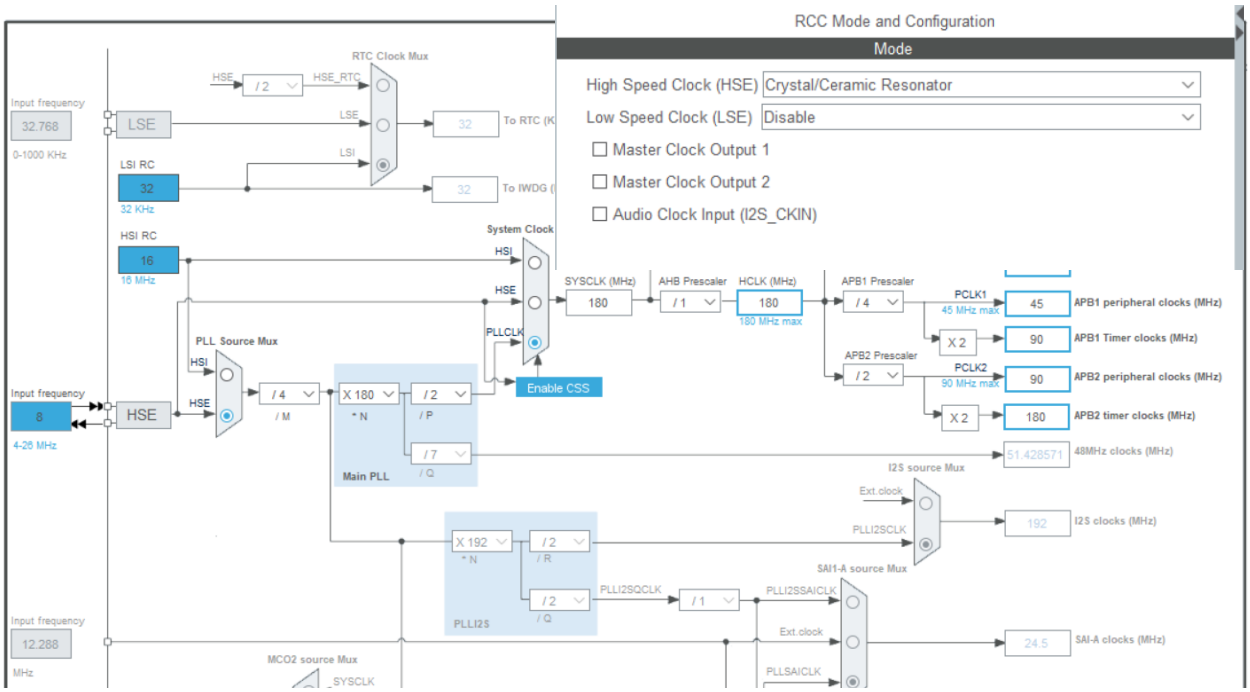

Figure 12: The clock configuration of STM32-F439ZIT and F446ZET boards

The CAN line is connected to the APB1 line on the Nucleo board. It is set to 45 Mhz, the highest speed we can get from the APB1 line.

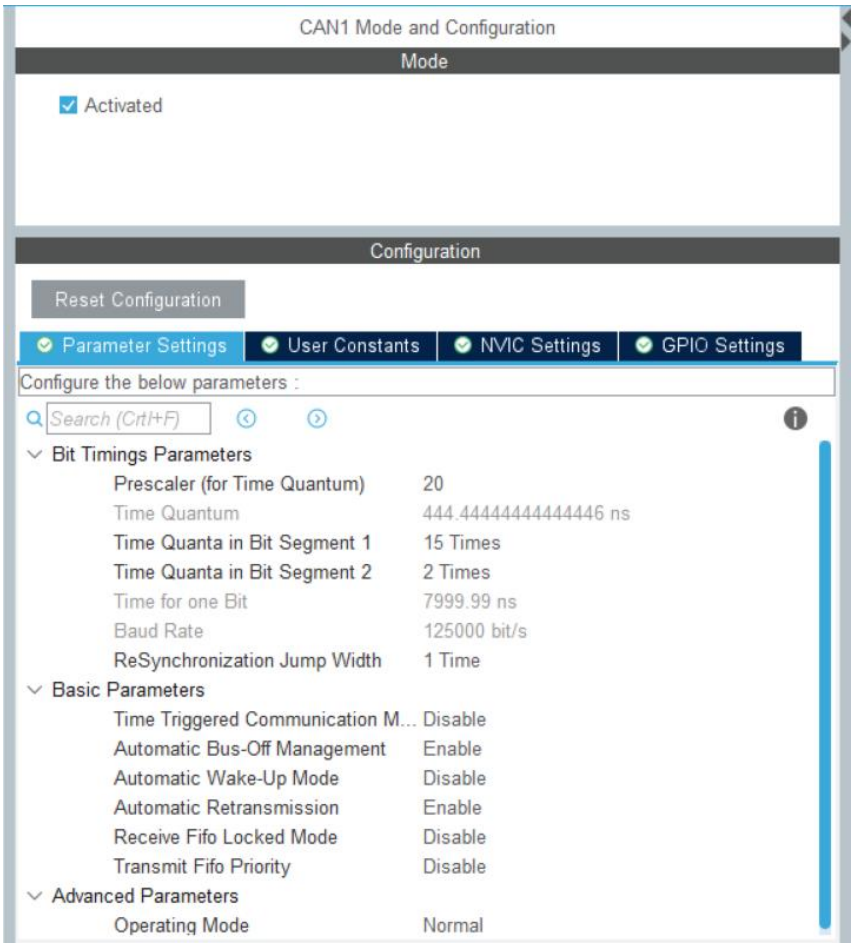

Figure 13: CAN1 Mode and Configuration

After the clock configuration are made, CAN1 is activated and Prescaler, Bit segment 1, and 2 are adjusted so that the baud rate is 125 kbit/s. The section on calculating the bit timing is mentioned in the previous section "THE CAN BIT TIME". "Automatic Bus-Off Management" is enabled in the simple parameters section. This feature ensures that the node that keeps the line busy by causing an error in the CAN line, turns off the node when the error counter reaches the limit value. More detailed information is explained in the section "THE CAN ERROR PREVENTION". "Automatic Retransmission" is enabled in the simple parameters section. This feature allows the sender to send the message frame again to the CAN line if the message is not received by any receiver. "Operating Mode" is selected as Normal. Other options, Loopback, Silent, Loopback combined Silent modes, can be used for testing the CAN line on a single Nucleo board. Detailed information about these test modes can be found in the datasheet of the board.

Since the user LEDs and the user button are used in the project, the necessary pin configuration should be made. The user button will be used as an interrupt, and in the "NVIC" section under the "System Core" tab, configurations are made as in the figure below. Note that the Priority Group is selected as "4-bits for pre-emption priority 0-bits for subpriority".

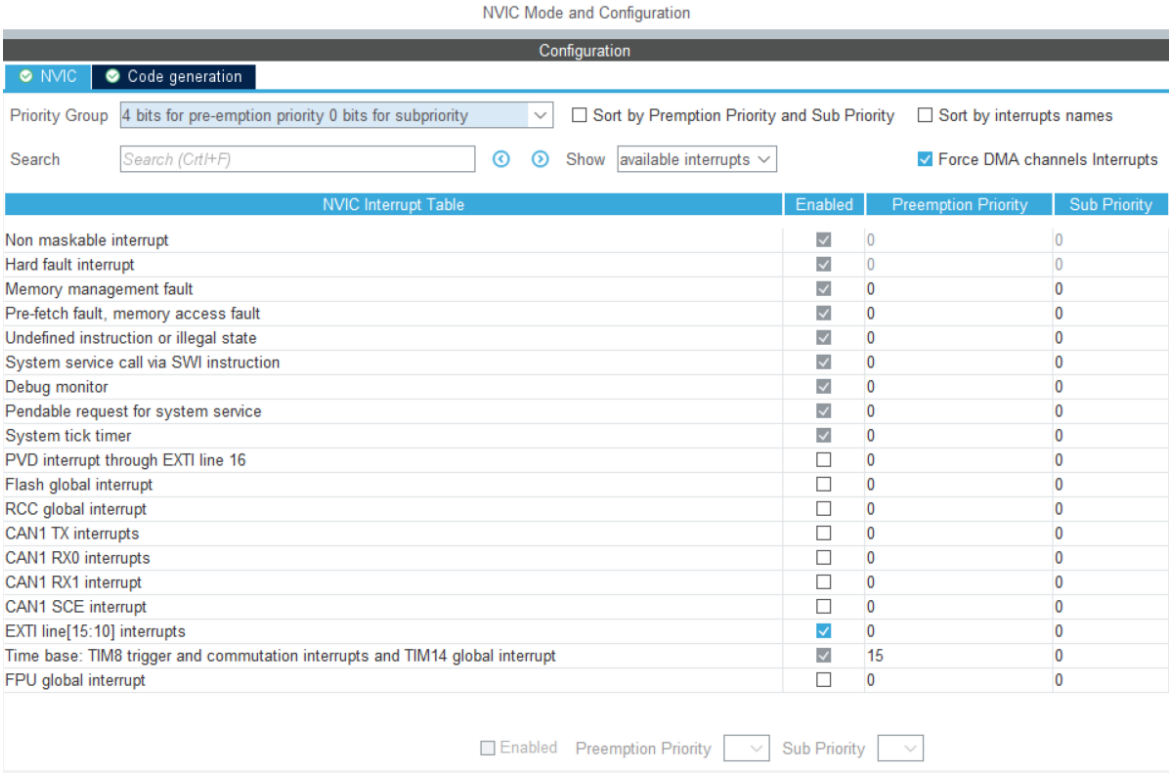

Figure 14: NVIC Mode and Configuration

DELTAV_CAN LIBRARY

After making the necessary configuration in STM32CUBEMX, the "deltav_can" library was created. The functions in the library are explained below with their purposes and parameters.

```
/**
  ******************************************************************************
  * @file    deltac_can.h
  * @Author  Orhan AGIRMAN
  * @brief   This file contains the headers of the deltav_can handlers.
  ******************************************************************************
  * @attention
  *
  * @verbatim
 ******************************************************************************
  */

#ifndef INC_DELTAV_CAN_H_
#define INC_DELTAV_CAN_H_

#ifdef __cplusplus
 extern "C" {
#endif

 /* Private variables ---------------------------------------------------------*/
CAN_TxHeaderTypeDef TxHeader;
CAN_RxHeaderTypeDef RxHeader;
CAN_FilterTypeDef sFilterConfig;
uint32_t pTxMailbox;
uint8_t RxData[8];

/* Exported functions prototypes ---------------------------------------------*/
void CAN_Tx(CAN_HandleTypeDef *hcan, uint32_t IDE, uint32_t StdId, uint32_t ExtId, uint32_t RTR, uint32_t DLC, uint8_t TxData[8]);
void CAN_Start(CAN_HandleTypeDef *hcan);
void CAN_Rx(CAN_HandleTypeDef *hcan, uint32_t RxFifo);
void CAN_FilterConfig(CAN_HandleTypeDef *hcan, uint32_t FilterIDE, uint32_t FilterActivation, uint32_t FilterBank, uint32_t FilterFIFOAssignment, uint32_t FilterMode, uint32_t FilterScale,
        uint32_t SlaveStartFilterBank, uint32_t FilterIdHigh, uint32_t FilterIdLow, uint32_t FilterMaskIdHigh, uint32_t FilterMaskIdLow);

#ifdef __cplusplus
}
#endif

#endif /* INC_DELTAV_CAN_H_ */

/************************ (C) COPYRIGHT DeltaV Uzay Teknolojileri A.S. *****END OF FILE****/
```

Figure 15: deltav_can header file

The functions in "deltav_can.c" are as follows.

```
69 void CAN_Start(CAN_HandleTypeDef *hcan){
70
71     if (HAL_CAN_Start(hcan) != HAL_OK)
72     {
73       Error_Handler();
74     }
75
76 }
```

Figure 16: CAN_Start Function

The “CAN_Start” function starts the CAN module. Retrieves an hcan pointer to a CAN_HandleTypeDef structure that contains configuration information for the CAN specified as a parameter. It does not return any value.

```
43 void CAN_Tx(CAN_HandleTypeDef *hcan, uint32_t IDE, uint32_t StdId, uint32_t ExtId, uint32_t RTR, uint32_t DLC, uint8_t TxData[8]){
44
45     TxHeader.IDE = IDE;
46     TxHeader.RTR = RTR;
47     TxHeader.DLC = DLC;
48     if(TxHeader.IDE == CAN_ID_STD){
49         TxHeader.StdId = StdId;
50         TxHeader.ExtId = 0;
51     }
52     else if(TxHeader.IDE == CAN_ID_EXT){
53         TxHeader.StdId = 0;
54         TxHeader.ExtId = ExtId;
55     }
56
57     if (HAL_CAN_AddTxMessage(hcan, &TxHeader, TxData, &pTxMailbox) != HAL_OK)
58     {
59       Error_Handler();
60     }
61 }
```

Figure 17: CAN_Tx Function

The “CAN_Tx” function provides CAN transmission operation. It takes 7 parameters. The parameters it takes, respectively, are

- hcan pointer to a CAN_HandleTypeDef structure that contains the configuration information for the specified CAN.
- The type of identifier for the message that will be transmitted. This parameter must be CAN_ID_STD or CAN_ID_EXT.
- Standard Id of the message that will be transmitted. (11 bits). This parameter must be a number between Min_Data = 0 and Max_Data = 0x7FF. If the IDE is selected as CAN_ID_EXT, it can be negligible.
- Extended Id of the message that will be transmitted. (29 bits). This parameter must be a number between Min_Data = 0 and Max_Data = 0x1FFFFFFF. If the IDE is selected as CAN_ID_STD, it can be negligible.
- The type of frame for the message that will be transmitted. This parameter must be CAN_RTR_DATA (Sending a Frame) or CAN_RTR_REMOTE (Requesting a Frame).

- The length of the frame that will be transmitted (0-8 Bytes). This parameter must be a number between Min_Data = 0 and Max_Data = 8.
- 8-Bytes Frame that will be transmitted. If the RTR is selected as CAN_RTR_REMOTE, it can be negligible.

It does not return any value.

```
85 void CAN_Rx(CAN_HandleTypeDef *hcan, uint32_t RxFifo){
86
87     if (HAL_CAN_GetRxMessage(hcan, RxFifo, &RxHeader, RxData) != HAL_OK)
88     {
89       Error_Handler();
90     }
91
92 }
```

Figure 18: CAN_Rx Function

The “CAN_Rx” function allows receiving the incoming frame. Takes 2 parameters.

- hcan pointer to a CAN_HandleTypeDef structure that contains the configuration information for the specified CAN.
- This parameter can be CAN Receive FIFO Number.

No value is returned.

```
134 void CAN_FilterConfig(CAN_HandleTypeDef *hcan, uint32_t FilterIDE, uint32_t FilterActivation, uint32_t FilterBank, uint32_t FilterFIFOAssignment, uint32_t FilterMode, uint32_t FilterScale,
135         uint32_t SlaveStartFilterBank, uint32_t FilterIdHigh, uint32_t FilterIdLow, uint32_t FilterMaskIdHigh, uint32_t FilterMaskIdLow){
136
137     sFilterConfig.FilterActivation = FilterActivation;
138     sFilterConfig.FilterBank = FilterBank;
139     sFilterConfig.FilterFIFOAssignment = FilterFIFOAssignment;
140     sFilterConfig.FilterMode = FilterMode;
141     sFilterConfig.FilterScale = FilterScale;
142     sFilterConfig.SlaveStartFilterBank = SlaveStartFilterBank;
143
144     if(FilterIDE == CAN_ID_STD){
145         sFilterConfig.FilterIdHigh = FilterIdHigh << 5;
146         sFilterConfig.FilterIdLow = FilterIdLow << 5;
147         sFilterConfig.FilterMaskIdHigh = FilterMaskIdHigh << 5;
148         sFilterConfig.FilterMaskIdLow = FilterMaskIdLow << 5;
149     }
150     else if(FilterIDE == CAN_ID_EXT && FilterScale == CAN_FILTERSCALE_32BIT){
151         sFilterConfig.FilterIdHigh = FilterIdHigh >> 13;
152         sFilterConfig.FilterIdLow = FilterIdLow << 3 | (0x0004);
153         sFilterConfig.FilterMaskIdHigh = FilterMaskIdHigh >> 13;
154         sFilterConfig.FilterMaskIdLow = FilterMaskIdLow << 3 | (0x0004);
155     }
156
157     if (HAL_CAN_ConfigFilter(hcan, &sFilterConfig) != HAL_OK)
158     {
159       Error_Handler();
160     }
161
162 }
```

Figure 19: CAN_FilterConfig Function

The “CAN_FilterConfig” function allows filtering the incoming frame. It takes 12 parameters. No value is returned.

- hcan pointer to a CAN_HandleTypeDef structure that contains the configuration information for the specified CAN.
- The type of identifier for the message that will be filtered. This parameter must be CAN_ID_STD or CAN_ID_EXT.
- Enable or disable the filter. This parameter must be CAN_FILTER_ENABLE or CAN_FILTER_DISABLE.
- Select the filter bank which will be initialized. For single CAN instance(14 dedicated filter banks), this parameter must be a number between Min_Data = 0 and Max_Data = 13. For dual CAN instances (28 filter banks shared), this parameter must be a number between Min_Data = 0 and Max_Data = 27.
- Select the FIFO (0 or 1) which will be assigned to the filter. This parameter must be CAN_FILTER_FIFO0 or CAN_FILTER_FIFO1.
- Specifies the filter mode to be initialized. This parameter must be CAN_FILTERMODE_IDMASK or CAN_FILTERMODE_IDLIST.
- Specifies the filter scale 16-bit or 32-bit. This parameter must be CAN_FILTERSCALE_16BIT or CAN_FILTERSCALE_32BIT
- Select the start filter bank for the slave CAN instance. For single CAN instances, this parameter is meaningless. For dual CAN instances, all filter banks with lower index are assigned to master CAN instance, whereas all filter banks with greater index are assigned to slave CAN instance. This parameter must be a number between Min_Data = 0 and Max_Data = 27.
- Specifies the filter identification number (MSBs for a 32-bit configuration, first one for a 16-bit configuration). This parameter must be a number between Min_Data = 0x0000 and Max_Data = 0xFFFF.
- Specifies the filter identification number (LSBs for a 32-bit. configuration, second one for a 16-bit configuration). This parameter must be a number between Min_Data = 0x0000 and Max_Data = 0xFFFF.
- Specifies the filter mask number or identification number, according to the mode (MSBs for a 32-bit configuration, first one for a 16-bit configuration). This parameter must be a number between Min_Data = 0x0000 and Max_Data = 0xFFFF.
- Specifies the filter mask number or identification number, according to the mode (LSBs for a 32-bit configuration, second one for a 16-bit configuration). This parameter must be a number between Min_Data = 0x0000 and Max_Data = 0xFFFF.

The following CAN related IRQ functions should be added to the "stm32f4xx_it.c" interrupt file.

```
/* USER CODE BEGIN 1 */
void CAN1_TX_IRQHandler(void){

    HAL_CAN_IRQHandler(&hcan1);


}
void CAN1_RX0_IRQHandler(void){

    HAL_CAN_IRQHandler(&hcan1);

}
void CAN1_RX1_IRQHandler(void){

    HAL_CAN_IRQHandler(&hcan1);

}
void CAN1_SCE_IRQHandler(void){

    HAL_CAN_IRQHandler(&hcan1);

}
/* USER CODE END 1 */
```

Figure 20: CAN1 IRQ Functions in "stm32f4xx_it.c"

Since the user button is used as an interrupt in this project, the relevant function is given below. Each time the user presses the button, the counter value increases and is sent to the CAN line.

```
/* USER CODE BEGIN EV */
extern CAN_HandleTypeDef hcan1;
extern uint8_t count;
extern uint8_t TxData[8];
/* USER CODE END EV */

void EXTI15_10_IRQHandler(void)
{
  /* USER CODE BEGIN EXTI15_10_IRQn 0 */

  /* USER CODE END EXTI15_10_IRQn 0 */
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
  /* USER CODE BEGIN EXTI15_10_IRQn 1 */
  if(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13)){
      while(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13));
      for(int i=0; i<2000000; i++);
      //HAL_Delay(100);
      count++;
      TxData[0]=count;
      CAN_Tx(&hcan1, CAN_ID_STD, 0x156, 0, CAN_RTR_DATA, 1, TxData);
        for(int i=0; i<10000000; i++);
        //  HAL_Delay(10);
    }
  /* USER CODE END EXTI15_10_IRQn 1 */
}
```

Figure 21: User-button Interrupt function in "stm32f4xx_it.c"

The callback functions of the CAN line are given below. The relevant ones of these functions can be used by adding them to the "main.c" file.

```
void HAL_CAN_TxMailbox0CompleteCallback(CAN_HandleTypeDef *hcan)
void HAL_CAN_TxMailbox1CompleteCallback(CAN_HandleTypeDef *hcan)
void HAL_CAN_TxMailbox2CompleteCallback(CAN_HandleTypeDef *hcan)
void HAL_CAN_TxMailbox0AbortCallback(CAN_HandleTypeDef *hcan)
void HAL_CAN_TxMailbox1AbortCallback(CAN_HandleTypeDef *hcan)
void HAL_CAN_TxMailbox2AbortCallback(CAN_HandleTypeDef *hcan)
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
void HAL_CAN_RxFifo0FullCallback(CAN_HandleTypeDef *hcan)
void HAL_CAN_RxFifo1MsgPendingCallback(CAN_HandleTypeDef *hcan)
void HAL_CAN_RxFifo1FullCallback(CAN_HandleTypeDef *hcan)
void HAL_CAN_SleepCallback(CAN_HandleTypeDef *hcan)
void HAL_CAN_WakeUpFromRxMsgCallback(CAN_HandleTypeDef *hcan)
void HAL_CAN_ErrorCallback(CAN_HandleTypeDef *hcan)
```

In this project, the following callback function runs when data comes to FIFO0. By using CAN_Rx function in the function, incoming data is received and processed and user LEDs are turned on.

```
276 void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan){
277
278     CAN_Rx(&hcan1,CAN_RX_FIFO0);
279
280         if(RxData[0] %3 == 1){
281             HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET);
282             HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7 | GPIO_PIN_14, GPIO_PIN_RESET);
283         }
284         else if(RxData[0] %3 == 2){
285             HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, GPIO_PIN_SET);
286             HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0 | GPIO_PIN_14, GPIO_PIN_RESET);
287         }
288         else if(RxData[0] %3 == 0){
289             HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_SET);
290             HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0 | GPIO_PIN_7, GPIO_PIN_RESET);
291         }
292
293
294 }
```

Figure 22: HAL_CAN_RxFifo0MsgPendingCallback function

In order for CAN interrupts to work, the following function must be called in int main(void) in the "main.c" file. Note that the parameter of the related callback function is given as a parameter, more detailed information can be found in the "stm32f4xx_hal_can.c" file.

```
95   //  Enable interrupts.
96   if (HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO0_MSG_PENDING) != HAL_OK)
97   {
98     Error_Handler();
99   }
```

Figure 23: HAL_CAN_ActivateNotification function

After configuring the CAN line in STM32CUBEMX, changes can be made from the function below.

```
169 static void MX_CAN1_Init(void)
170 {
171
172   /* USER CODE BEGIN CAN1_Init 0 */
173
174   /* USER CODE END CAN1_Init 0 */
175
176   /* USER CODE BEGIN CAN1_Init 1 */
177
178   /* USER CODE END CAN1_Init 1 */
179   hcan1.Instance = CAN1;
180   hcan1.Init.Prescaler = 20;
181   hcan1.Init.Mode = CAN_MODE_NORMAL;
182   hcan1.Init.SyncJumpWidth = CAN_SJW_1TQ;
183   hcan1.Init.TimeSeg1 = CAN_BS1_15TQ;
184   hcan1.Init.TimeSeg2 = CAN_BS2_2TQ;
185   hcan1.Init.TimeTriggeredMode = DISABLE;
186   hcan1.Init.AutoBusOff = ENABLE;
187   hcan1.Init.AutoWakeUp = ENABLE;
188   hcan1.Init.AutoRetransmission = DISABLE;
189   hcan1.Init.ReceiveFifoLocked = DISABLE;
190   hcan1.Init.TransmitFifoPriority = DISABLE;
191   if (HAL_CAN_Init(&hcan1) != HAL_OK)
192   {
193     Error_Handler();
194   }
195   /* USER CODE BEGIN CAN1_Init 2 */
196
197   /* USER CODE END CAN1_Init 2 */
198
199 }
```

Figure 24: MX_CAN1_Init function