AN0095
Application Note

AT32 MCU CAN Quick Start Guide

Introduction

The CAN (Controller Area Network) is a distributed serial communication protocol which realizes real-time and reliable data communication between nodes, compliant with the CAN 2.0 specification (2.0A and 2.0B). This application note introduces CAN protocol, AT32 MCU CAN application flow and examples of AT32 MCU CAN application.

Note: The corresponding code in this application note is developed on the basis of V2.x.x BSP provided by Artery. For other versions of BSP, please pay attention to the differences in usage.

Applicable products:

Part number	AT32Fxx
	AT32Lxx

Contents

1 Introduction to CAN 6

2 CAN protocol 7

2.1 CAN bus topology 7

2.2 CAN bus physical layer features 7

2.3 Frame type 8

2.4 Frame structure 10

2.5 Bit stuffing 10

2.6 Bit format 11

2.7 Synchronization mechanism 12

2.8 Arbitration mechanism 13

2.9 CAN error handling mechanism 14

2.9.1 Error type 14

2.9.2 Error state 15

3 AT32 CAN 16

3.1 Function overview 16

3.2 Message transmission 16

3.3 Message reception 17

3.4 Filter 19

3.5 CAN baud rate and sample point 21

3.5.1 Baud rate formula 21

3.5.2 Sample point formula 21

3.5.3 Baud rate configuration tool 22

4 Application case 1: CAN communication in normal mode 23

4.1 Function 23

4.2 Resources 23

4.3 Software design 23

4.4 Test result 27

5 Application case 2: CAN receive filter 28

5.1 Function 28

5.2 Resources 28

5.3 Software design 28

5.4 Test result 35

6 Application case 3: CAN debugging in loopback mode 36

6.1 Function 36

6.2 Resources 36

6.3 Software design 36

6.4 Test result 40

7 Revision history 41

List of tables

Table 1 CAN frame types 8
Table 2 Description of bit segments 11
Table 3 Sample point settings 21
Table 4 Document revision history 41

List of figures

Figure 1 CAN bus topology 7
Figure 2 CAN bus level characteristics 8
Figure 3 CAN frame structure 9
Figure 4 CAN standard data frame 10
Figure 5 Normal bit timing 12
Figure 6 Resynchronization jump 13
Figure 7 Arbitration mechanism 14
Figure 8 CAN node error states 15
Figure 9 AT32 CAN function overview 16
Figure 10 Message transmission process 17
Figure 11 Message reception process 19
Figure 12 32-bit identifier mask mode 20
Figure 13 32-bit identifier list mode 20
Figure 14 16-bit identifier mask mode 20
Figure 15 16-bit identifier list mode 20
Figure 16 CAN BitRate configuration tool 22
Figure 17 CAN level converter schematics diagram 23
Figure 18 CAN level converter schematics diagram 28
Figure 19 CAN loopback mode 36

1 Introduction to CAN

The Controller Area Network (CAN) is designed to efficiently process a large amount of messages with minimum CPU usage. It is a serial bus protocol created in 1986 by the German company BOSCH and ISO standardized with ISO11898 and ISO11519, which gain popularity in the automotive network domain in Europe. It has been recognized for high performance and reliability and widely used in industrial automation, ships, medical facilities and industrial equipment, etc.

Features of CAN protocol:

- **Multi-master control**

  When the bus is idle, all nodes can start sending messages. When multiple nodes start sending at the same time, the bus proceeds arbitration according to the identifier (ID), and the one with the lowest ID (highest priority) wins and gets the right to send while all other nodes (with lower priority) switch to a receiving mode. It should be noted that ID does not the node address but the priority of message to be sent.

- **Flexibility**

  As mentioned, there is no address information of nodes on the CAN bus; therefore, adding or removing a node does not affect the hardware and software of other nodes on the CAN bus.

- **High reliability**

  CAN protocol features are error detection, error notification, fault confinement and error handling. Every node on the CAN bus can detect an error (error detection) in a message. If any error is found, the discovering node will transmit an error frame to notify other nodes (error notification). Each node has an error counter internally that accumulates error count value every time an error is detected. When the accumulative error count value of one node is greater than 256, this fault node is disconnected from the CAN bus to avoid affecting other nodes (fault confinement). If the node detects an error when sending a message, the erroneous message will be retransmitted after the fault is cleared (error handling).

- **Fast and long-range communication**

  The rate can reach 1 Mbps (communication distance < 40m) and communication distance can reach 10km (rate < 5Kbps).

- **Multiple nodes**

  The number of nodes on a CAN bus is not limited theoretically, but the ultimate number is determined by the bus time delay and electrical load. Lowering or increasing the transmission rate can correspondingly increase or decrease the number of nodes.

With these features, the CAN is suitable for interconnection of industrial process monitoring devices and is recognized as one of the most ideal fieldbuses for industrial applications. The CAN protocol is ISO standardized with ISO11898 and ISO11519, in which ISO11898 is the CAN high-speed communication standard with a communication speed of 125 Kbps~1 Mbps and ISO11519-2 is a low-speed communication standard with a communication rate of 125 Kbps or less. In this application note, the ISO11898 standard is adopted and the communication rate is 1 Mbps.

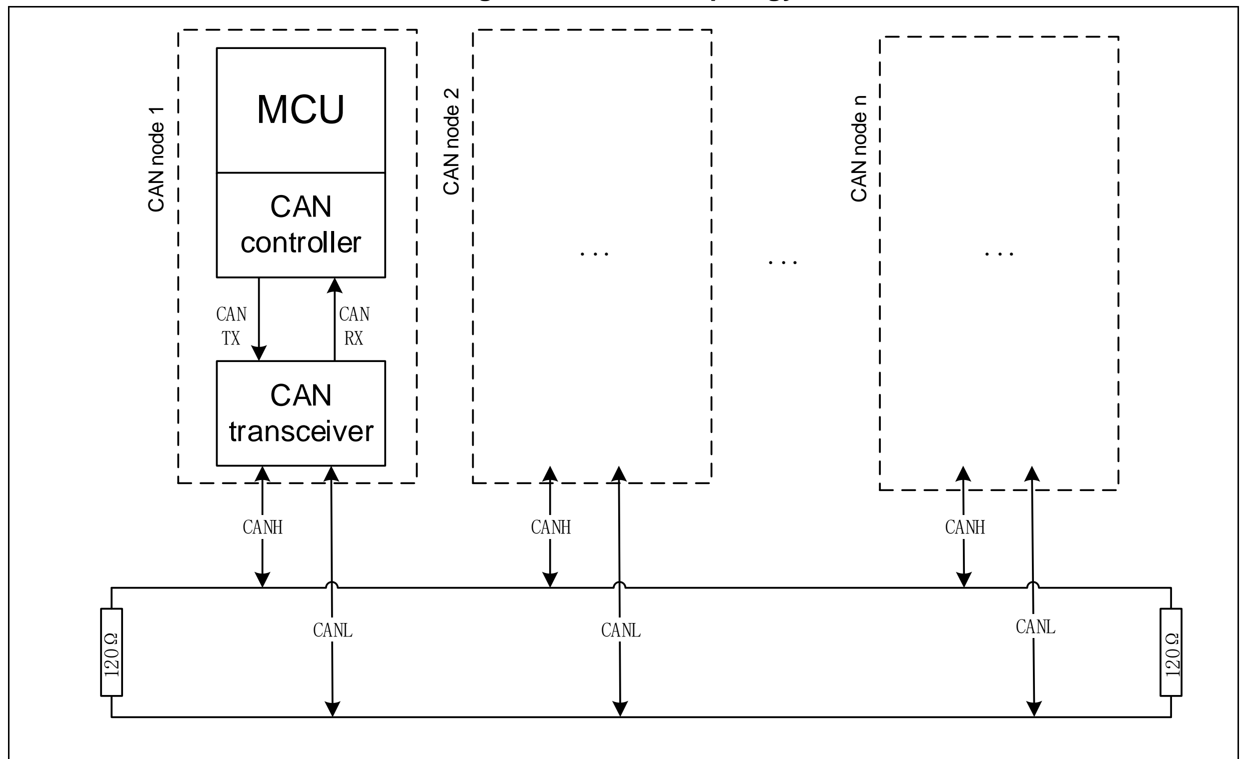
2 CAN protocol

This section mainly introduces the CAN bus topology, characteristics of physical layers, CAN frame types, frame structure, bit stuffing mechanism, bit format, synchronization mechanism, arbitration mechanism and error handling mechanism. For more details on CAN protocol, refer to BOSCH CAN specification.

2.1 CAN bus topology

As shown in Figure 1, the CAN bus consists of CANH and CANL lines. Each node is connected to CAN bus through a short branch line, and can transmit and receive data equally. In addition, there is a120 Ω terminal resistor at each end of CAN bus for impedance matching to reduce reflection.

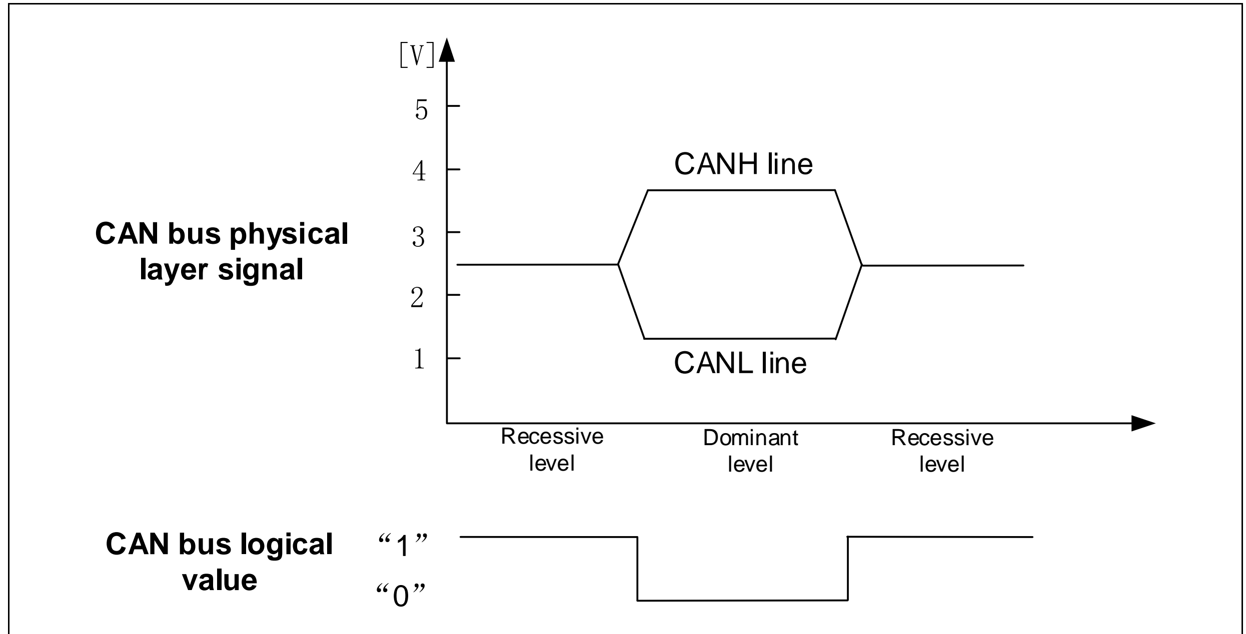
Figure 1 CAN bus topology



2.2 CAN bus physical layer features

As shown in Figure 2, the dominant level corresponds to logical “0” (voltage difference between CANH and CANL is about 2.5 V) and the recessive level corresponds to logical “1” (voltage difference between CANH and CANL is 0 V). On the CAN bus, the dominant level always overrides a recessive level, which means that the CAN bus level will be dominant is any number of nodes in the network output a dominant level, and the CAN bus level will only be recessive when all nodes in the network output a recessive level.

Figure 2 CAN bus level characteristics



2.3 Frame type

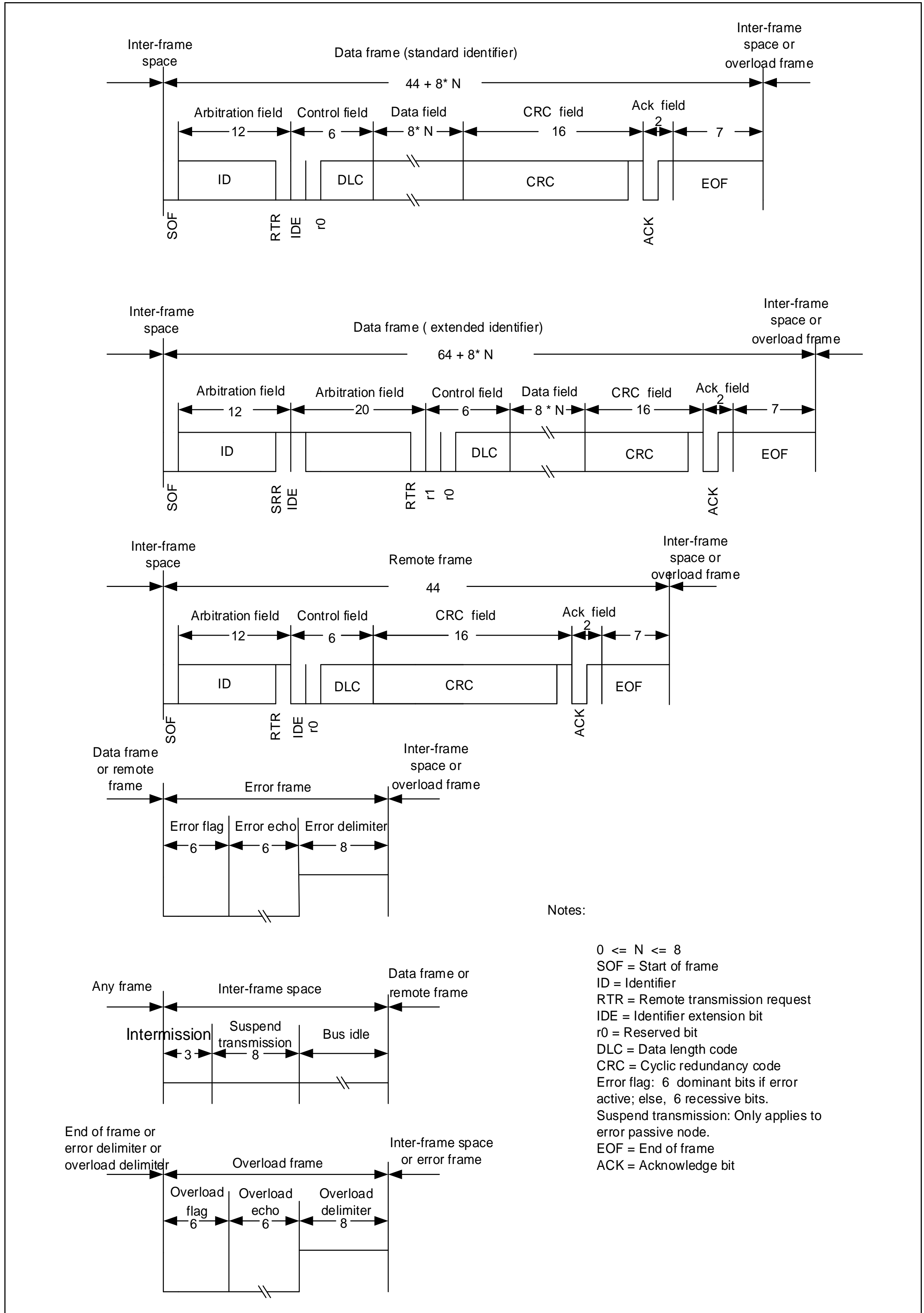
There are five types of frames in CAN protocol, as shown in Table 1. The data frame and remote frame are transmitted and received by users, while the error frame, overload frame and inter frame are transmitted by hardware of nodes on CAN bus according to the specific status, which is unnecessary to be or cannot be controlled by users.

Table 1 CAN frame types

Frame type	Description
Data frame	It carries data from transmitting node to receiving node.
Remote frame	It is sent by a node to request the transmission of a data frame with the same ID by another node.
Error frame	It is sent when error information is detected on any node.
Overload frame	It requests a delay between the preceding and subsequent data frame (or remote frames).
Inter frame	It is also known as inter-frame space, which is used to separate the above four types of frames.

The structure of each frame type is shown in Figure 3.

Figure 3 CAN frame structure



2.4 Frame structure

This section mainly introduces standard data frame. For other frame types, refer to Figure 3 for better understanding.

The CAN data frame is composed of seven fields as below:

SOF (Start of frame): It is dominant of 1-bit. The CAN bus level is recessive when the CAN bus is in an idle state; therefore, the dominant SOF bit marks the start of a message on the node.

Arbitration: It indicates the frame priority and contains message identifier and frame type (data/remote frame).

Control: It indicates the number of bytes, identifier format (standard/extended) and the reserved bit.

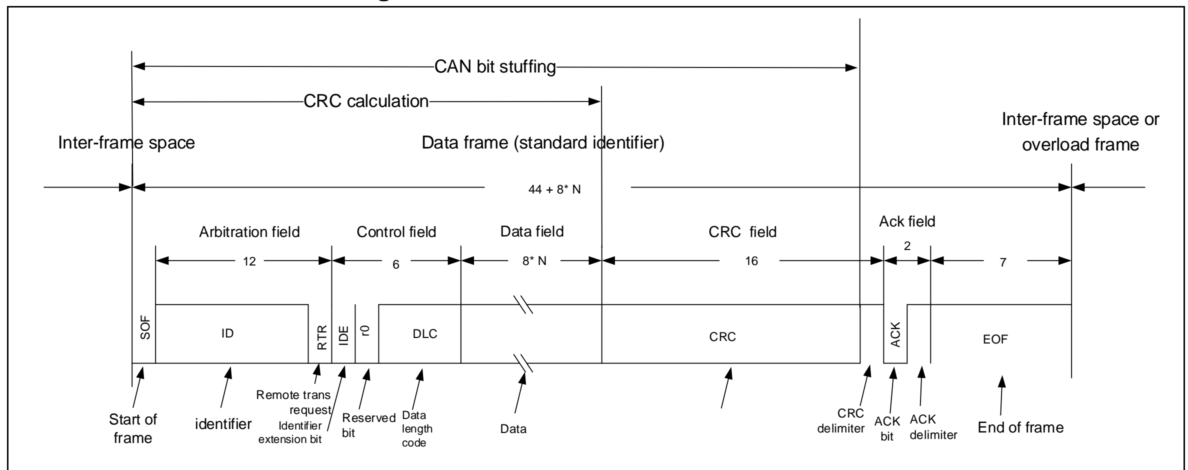
Data: Up to 8 bytes of data can be sent in one message (data length is determined by the DLC in the control field).

CRC (cyclic redundancy check): The transmitter calculates a check sum from the transmitted bits (stuff bit not included) and provides the result within the frame in the CRC field. The receivers use the same polynomial to calculate the check sum from the bits as seen on the bus-lines. The self-calculated check sum is compared with the received on. In case of a mismatch, the receiving node sends an error frame. If it matches, the frame is regarded as correctly received and then sent to the ACK field.

ACK (acknowledge): It contains the ACK SLOT and ACK DELIMITER. The transmitting node is always recessive in the ACK field. If no error is detected on the receiving node during receiving process, the ACK bit outputs 1-bit dominant level to notify the transmitting node that this frame is correctly received.

EOF (End of frame): It indicates the end of a data frame, and it is recessive of 7-bit.

Figure 4 CAN standard data frame



2.5 Bit stuffing

Since the CAN bus contains CANH/CANL lines only, without a CLK line for synchronization, it implements synchronization directly through the signal in data stream (refer to 2.7 Synchronization Mechanism). In addition, the bit stuffing mechanism is introduced to deal with the possible condition of no edge in data stream.

Every time 5 consecutive bits at the same level are found in the bit sequence sent on the bus, the CAN controller in the transmitting device(s) automatically inserts a stuff bit at the opposite value. For

example, the original data stream is “0000000111110001…” and the data stream sent to CAN bus after bit stuffing is “000001001111100001…”, in which the underlined bits are stuff bits.

Bit stuffing is added from the SOF to CRC field (CRC delimiter not included), as shown in Figure 4.

2.6 Bit format

One bit of AT32 CAN can be divided into the following 3 segments:

- Synchronization segment (SYNC_SEG)
- Bit segment 1, including PROP_SEG and PHASE_SEG1 in CAN standard, marked as BSEG1.
- Bit segment 2, i.e., PHASE_SEG2 in CAN standard, marked as BSEG2.

These segments consists of the smallest time unit (Time Quantum, Tq).

One bit is divided into three segments, and each segment consists of several Tq, which is called bit timing.

The number of Tq per bit / bit segment and bit timing can be set as required. Users can set the bit timing and Tq length to set CAN baud rate and sample point.

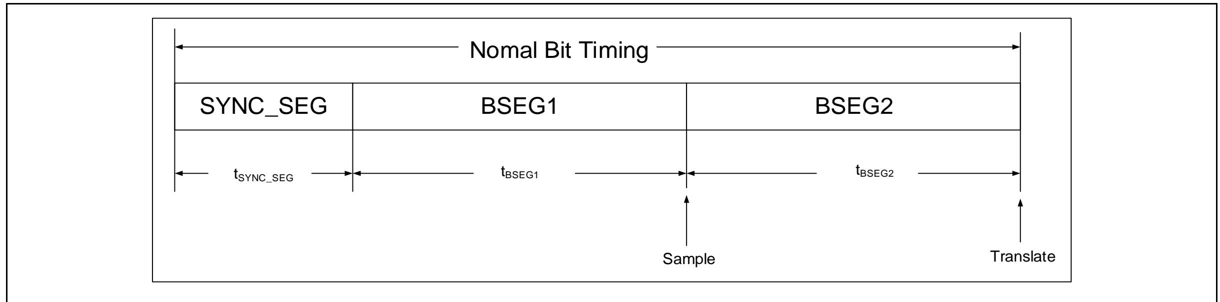
Table 2 lists each segment and configurable Tq of AT32 CAN.

Table 2 Description of bit segments

<table>
<tr><th>Segment (CAN standard)</th><th>Segment (AT32 CAN)</th><th>Description</th><th>Tq</th></tr>
<tr><td>Synchronization segment (SYNC_SEG)</td><td>Synchronization segment (SYNC_SEG)</td><td>It is used to synchronize all nodes connected to the CAN bus, and a signal edge is expected within this segment.</td><td>1Tq</td></tr>
<tr><td>Propagation segment (PROP SEG)</td><td rowspan="2">Bit segment 1 (BSEG1)</td><td>It is used to compensate for the physical delay times within the network, and its length is twice these delay times to compensate for input comparator and output driver.</td><td rowspan="2">1~16Tq</td></tr>
<tr><td>Phase segment 1 (PHASE SEG1)</td><td rowspan="2">It is used to compensate for phase error in the edge phase, and it may be lengthened or shortened through resynchronization jump.</td></tr>
<tr><td>Phase segment 2 (PHASE SEG2)</td><td>Bit segment 2 (BSEG2)</td><td>1~8Tq</td></tr>
</table>

As shown in Figure 5, one bit consists of a synchronization segment, bit segment 1 and bit segment 2. The sample point, i.e., the sampling point of receiving node, is at the junction of BSEG1 and BSEG2.

Figure 5 Normal bit timing



2.7 Synchronization mechanism

Hard synchronization

After a hard synchronization, the bit time is restarted with the end of synchronization segment. Thus hard synchronization forces the edge which has caused the hard synchronization to lie within the synchronization segment of the restarted bit time, as shown in Figure 6.

Resynchronization jump width

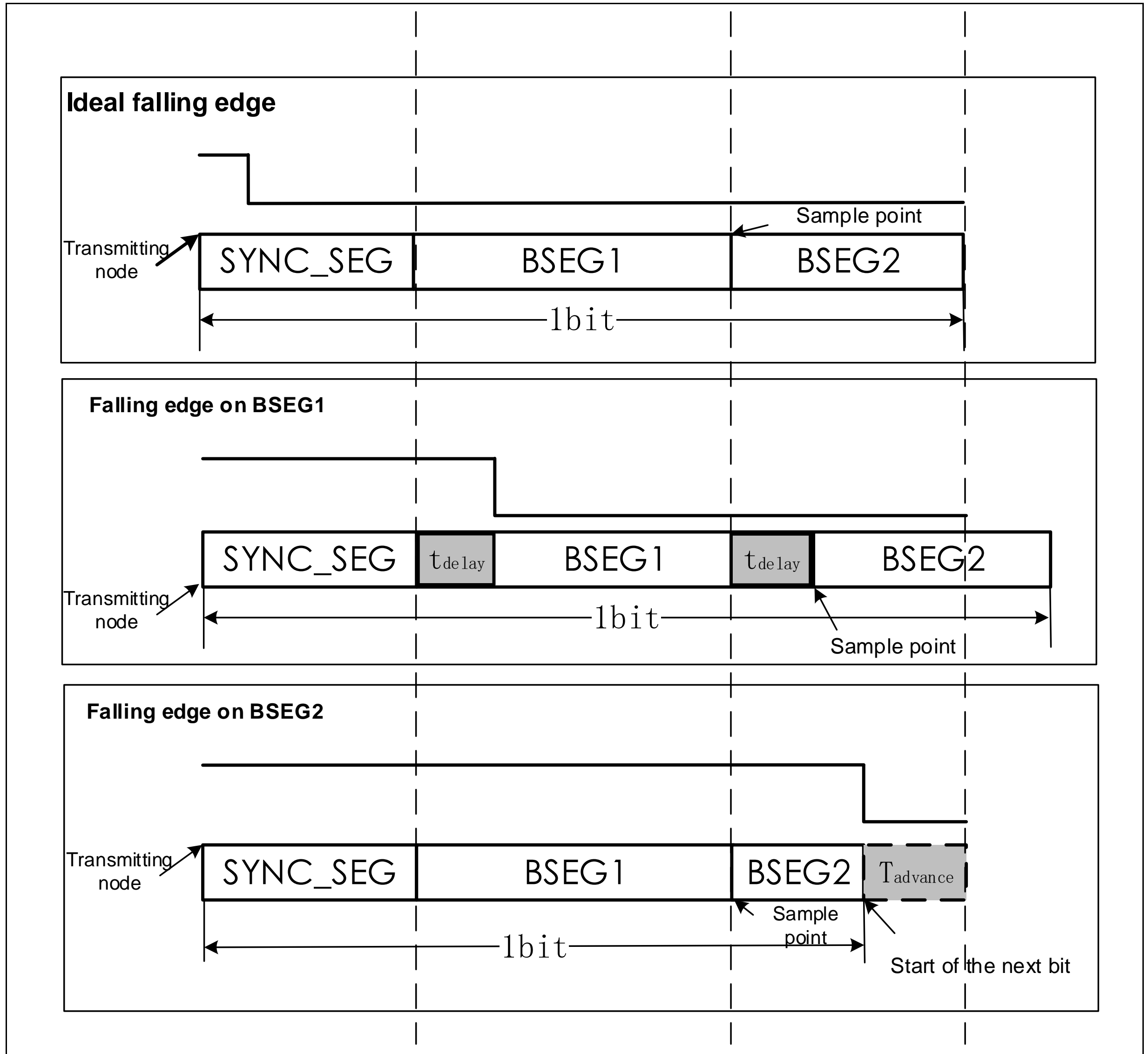
Resynchronization leads to lengthening of bit segment 1 or shortening of bit segment 2. The maximum bit lengthening and shortening are given by the resynchronization jump width (which should be set to 1~4 Tq).

As shown in Figure 6,

When a falling edge is detected on BSEG1, the BSEG1 lengthens for T_{delay} and the current bit lengthens for T_{delay}, in which the T_{delay}<= resynchronization jump width.

When a falling edge is detected on BSEG2, the BSEG2 shortens for $T_{advance}$ and the current bit shortens for $T_{advance}$, in which the $T_{advance}$<= resynchronization jump width.

Figure 6 Resynchronization jump

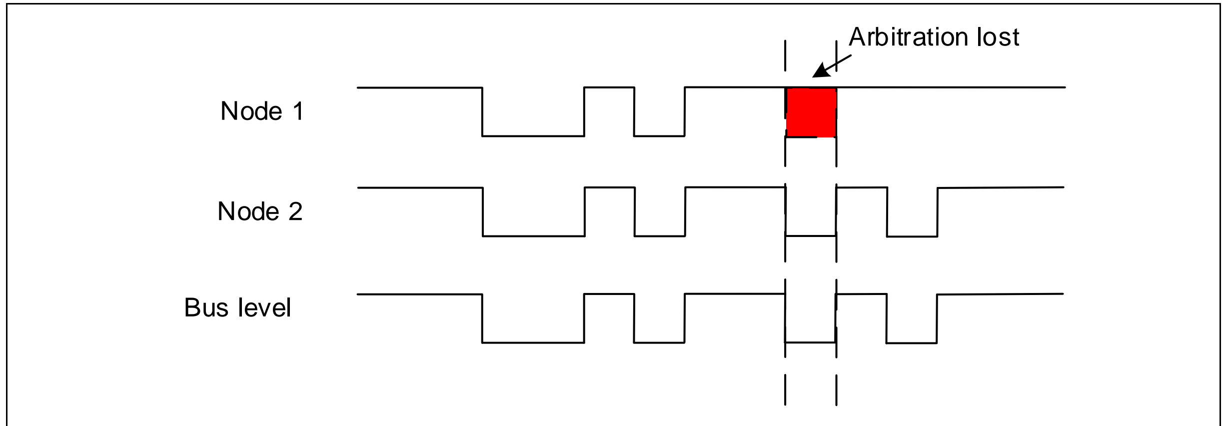


2.8 Arbitration mechanism

Whenever the bus is free, any node may start to transmit the message. If two or mode nodes start transmitting messages at the same time, the bus access conflict occurs, which can be resolved by the bitwise arbitration using the identifier. The mechanism of the arbitration guarantees that neither information nor time is lost. If a data frame and a remote frame with the same identifier are initiated at the same time, the data frame prevails over to the remote frame. During the arbitration, every transmitter compares the level of the bit transmitted with the level that is monitored on the bus. If these levels are equal, the node may continue to the send. When a recessive level is sent, but a dominant level is monitored, the node has lost arbitration and must withdraw without sending any further bits.

As shown in Figure 7, note 1 and node 2 transmit a message at the same time with the same identifier; from the red mark, node 1 starts transmitting recessive level “1” and node 2 starts transmitting dominant level “0”. At this time, node 2 wins the arbitration and keeps transmission, and the bus level is the same as that of node 2; while node 1 loses the arbitration and switches to a receiving mode in the next bit, with the subsequent transmitting pins remains at a recessive level.

Figure 7 Arbitration mechanism



2.9 CAN error handling mechanism

2.9.1 Error type

There are five types of error in CAN protocol.

- **Bit error**

  CAN nodes transmitting a message to the bus also monitor the bus and compare the transmitted level bit by bit with the corresponding level on the bus. A bit error is detected when the transmitted bit level differs from the monitored bus level. AT32 CAN bit error is classified into dominant bit error (transmitted bit is dominant while recessive bit is monitored) and recessive bit error (transmitted bit is recessive while dominant bit is monitored). A bit error may occur when CAN nodes are in a transmitting mode.
  However, there are exceptions that a dominant bit on the bus will not lead to a bit error when a recessive bit is transmitted during arbitration or during the ACK slot, and a node sending a passive error frame and detecting a dominant bit will not interpret this as a bit error.

- **Stuff error**

  A stuff error occurs whenever 6 consecutive bits of equal value are detected on the bus (in the bit stuffing region shown in Figure 4). A stuff error may occur when CAN nodes are in a receiving mode.

- **CRC error**

  CRC sequence contains the CRC checksum calculated by the transmitter (the receiver uses the same polynomial to calculate the checksum). If the calculated CRC is different from the received CRC, then the receiver signals it as CRC error. A CRC error may occur when CAN nodes are in a receiving mode.

- **Form error**

  A form error is detected when a fixed format field contains one or more dominant bits. For example, if a dominant bit is detected in the CRC delimiter /ACK delimiter, it is a form error. There is an exception that the dominant bit during the last bit at the end of frame of the receivers is not considered as a frame error. A form error may occur when the CAN node is in a receiving mode.

- **Acknowledgment error**

  As long as the monitored during the ACK SLOT bit is not "dominant", then the transmitter will detect an error response (acknowledgment error). An acknowledgment error may occur when

CAN node is in a transmitting mode.

2.9.2 Error state

When an error is detected on the CAN node, the transmit error counter (TEC[7:0]) /receive error counter (REC[7:0]) is increased by 1 or 8 (refer to BOSCH CAN protocol for details) according to the specific error state and type. After each time of correct transmission/receiving of a message, the TEC/REC is decreased by 1. Therefore, the TEC/REC value indicates the stability of CAN node and network. Depending on the TEC/REC value, a node can be in one of three states:

- **Error active**

In this state, node can participate in all CAN activity and raise "Active Error Flag" (6 dominant bits) when detecting errors. As shown in Figure 8, when TEC<128 and REC<128, a CAN node enters into error active state.

- **Error passive**

In this state, node can participate in data/remote frame receiving and transmitting, and raise "Passive Error Flag" (6 recessive bits) when detecting errors. As shown in Figure 8, when 255≥TEC>128 and 255≥REC>128, a CAN node enters into error passive state.

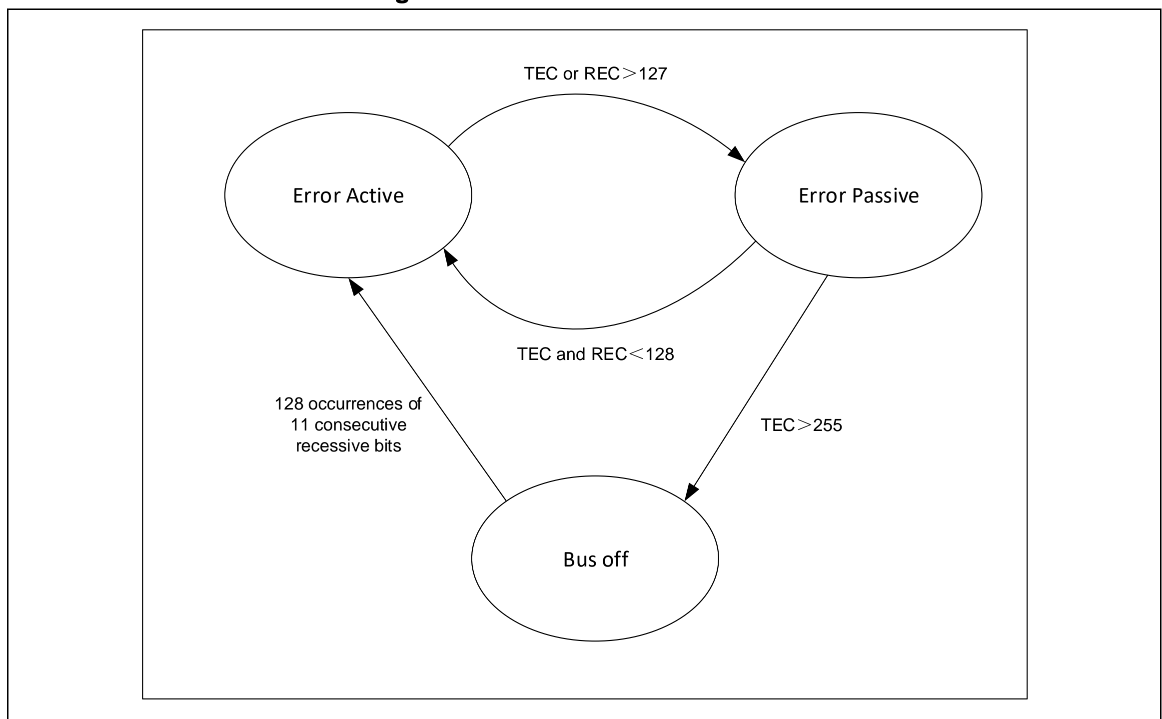
- **Bus-off**

In this state, node is switched off from the CAN bus and neither sends nor receives frames. As shown in Figure 8, when TEC>255, a CAN node enters into bus-off state.

AT32 CAN node bus-off management: AT32 CAN node recovers from bus-off state in the following conditions:

1) When the AEBOEN bit of CAN_MCTRL register is set to 0, the software requests to enter and then exit freeze mode, and then the CAN node waits for 128 occurrences of 11 consecutive recessive bits (detected on RX) in communication mode.
2) When AEBOEN bit is set to 1, the CAN node waits for 128 occurrences of 11 consecutive recessive bits (detected on RX) in communication mode.

Figure 8 CAN node error states



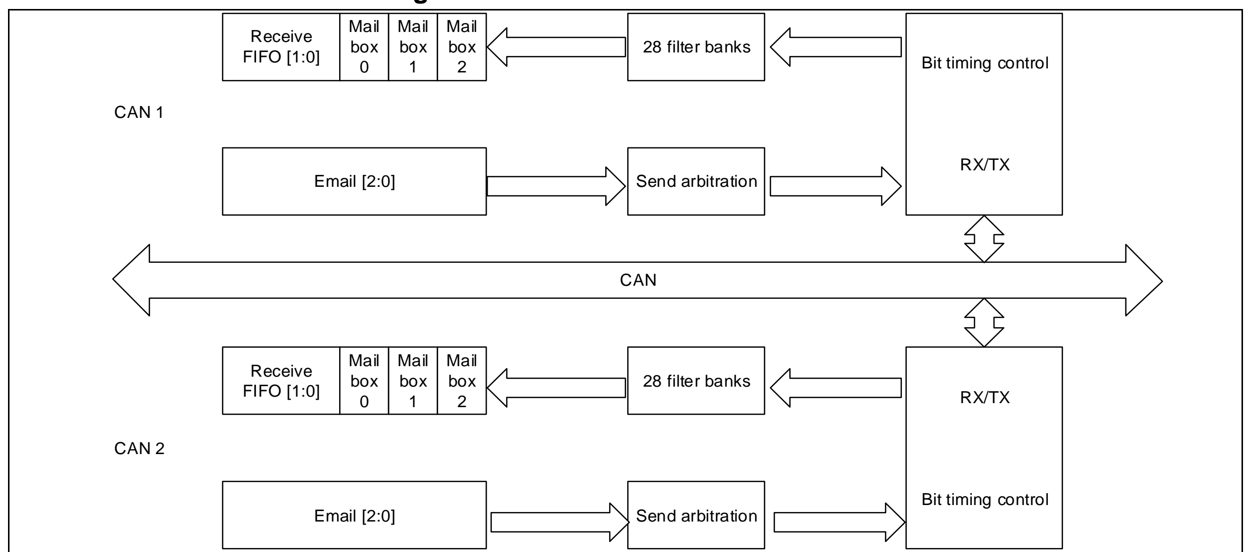
3 AT32 CAN

AT32 CAN is compliant with the CAN 2.0 specification (2.0A and 2.0B), with some functions and configurable options added based on the compatible standard CAN protocol. The main differences between CAN 2.0A and 2.0B are as follows: CAN 2.0A only supports 11-bit IDs (i.e., standard frames) while CAN 2.0B supports 11-bit/29-bit IDs (i.e., standard and extended frames).
This section mainly introduces the AT32 CAN structure and application method, AT32 CAN communication process including transmission, receiving, message filtering, baud rate and sample point settings. Refer to Reference Manual for details about error management and interrupt management, etc.

3.1 Function overview

As the number of nodes in the CAN network and the number of messages grows, an enhanced filtering mechanism is required to handle all types of messages in order to reduce the processing time of message reception. One FIFO scheme is used to ensure that the CPU can concentrate on application tasks for a long period of time without the loss of messages. In the meantime, the priority order of the messages to be transmitted is configured by hardware. Based on the above mentioned conditions, AT32 CAN controller provides 28 identifier filter banks (configurable bit width), two receive FIFOs (storing 3 complete messages each) and three transmit mailboxes with their transmit priority order defined by the transmit scheduler. The message transmitting and receiving process is totally managed by hardware, without occupying CPU.

Figure 9 AT32 CAN function overview



3.2 Message transmission

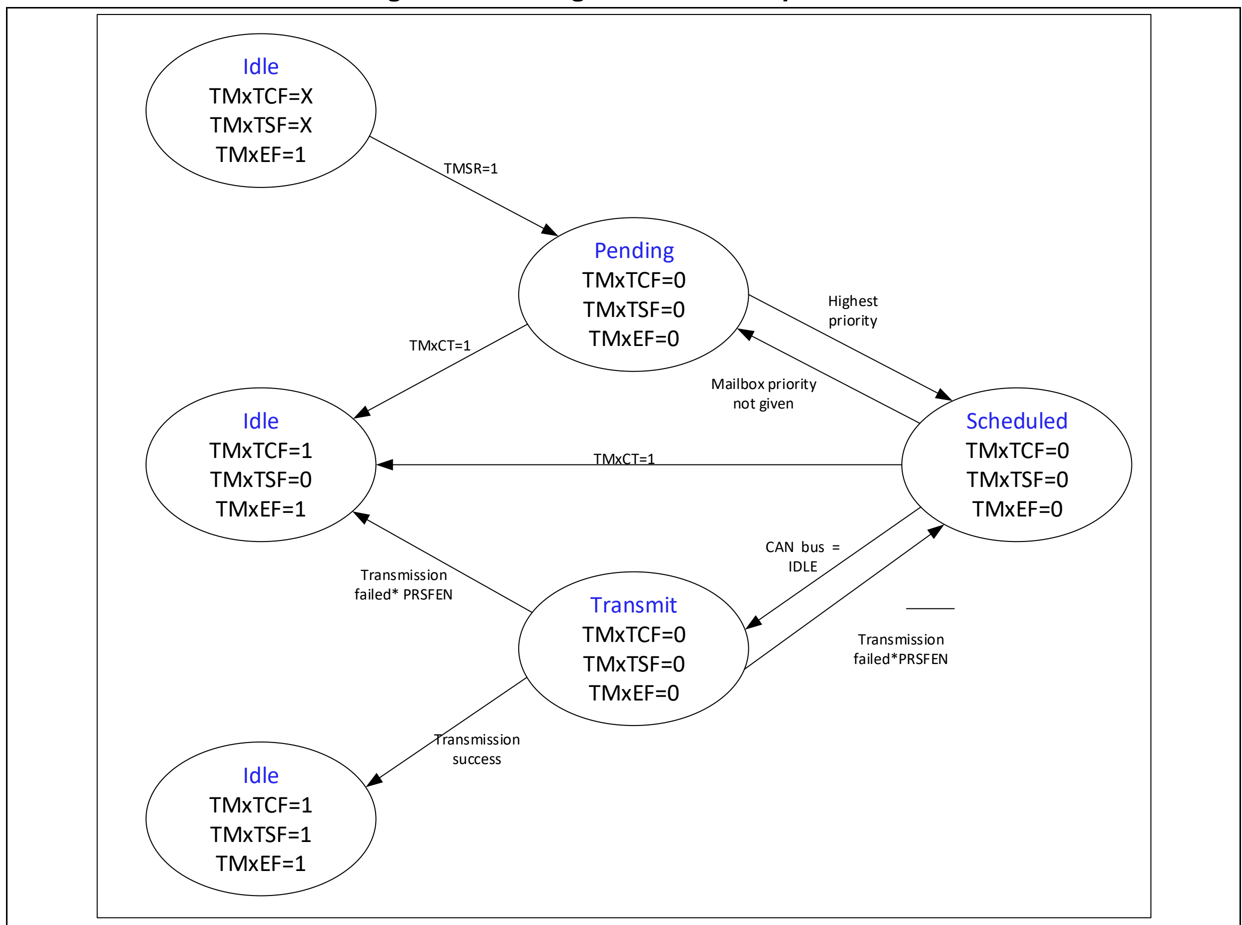
The message transmission process is shown in Figure 10, including the following steps.
Steps 1~3 are performed by the user, and steps 4~7 are completed by hardware automatically.

1) Program selects one empty mailbox (send mailbox empty flag by setting TMxEF=1);
2) Write the message to be sent into the corresponding empty mailbox, and the message content includes ID, frame type, data length and transmit data;
3) Request to send: Set TMSR=1 in the CAN_TMIx register;
4) Mailbox pending (wait until the priority is given);
5) Schedule transmission (wait until the CAN bus becomes free);

6) Transmit;
7) Mailbox becomes empty.

Note: Steps 1~7 are brief transmission procedures, and Figure 10 also demonstrates transmission cancelled, transmission failed, and automatic retransmission enabled / disenabled. Refer to message transmission in the Reference Manual for details.
Figure 10 shows the following flag bits and operation bits:
TMxTCF bit: Transmission complete flag (request to send/abort)
TMxTSF bit: Transmission success flag
TMxEF bit: Transmit mailbox empty flag
TMSR bit: Request to send
TMxCT bit: Abort sending
PRSFEN: Disable automatic retransmission (automatic retransmission disabled when PRSFEN =1; automatic retransmission enabled when PRSFEN=0)

Figure 10 Message transmission process



3.3 Message reception

The message reception process is shown in Figure 11 (empty and pending_1), including the following steps.
1) FIFO empty;
2) Valid message received;
3) Enter into “pending_1” state (there is one valid message in FIFO);
4) Read valid message: Read CAN_RFIx, CAN_RFCx, CAN_RFDTLx and CAN_RFDTHx

registers;

5) Release mailbox: Set RFxR=1 in the CAN_RFx register.

Note: Steps 4~3 are performed by the user, and steps 1~3 are completed by hardware automatically.

Valid message:

When a message is received correctly (no error occurs in the whole EOF field) and has passed identifier filtering, it is regarded as a valid message.

If the user does not participate in the message receiving process (i.e., neither read the valid message nor release mailbox), the hardware completes the following process:

1) Valid message received;
2) Enter into “pending_1” state (there is one valid message in FIFO);
3) Valid message received;
4) Enter into “pending_2” state (there are two valid messages in FIFO);
5) Valid message received;
6) Enter into “pending_3” state (there are three valid messages in FIFO);
7) Valid message received;
8) Enter into “overflow” state (there are three valid messages in FIFO and one message is lost, and the overflow flag is set).

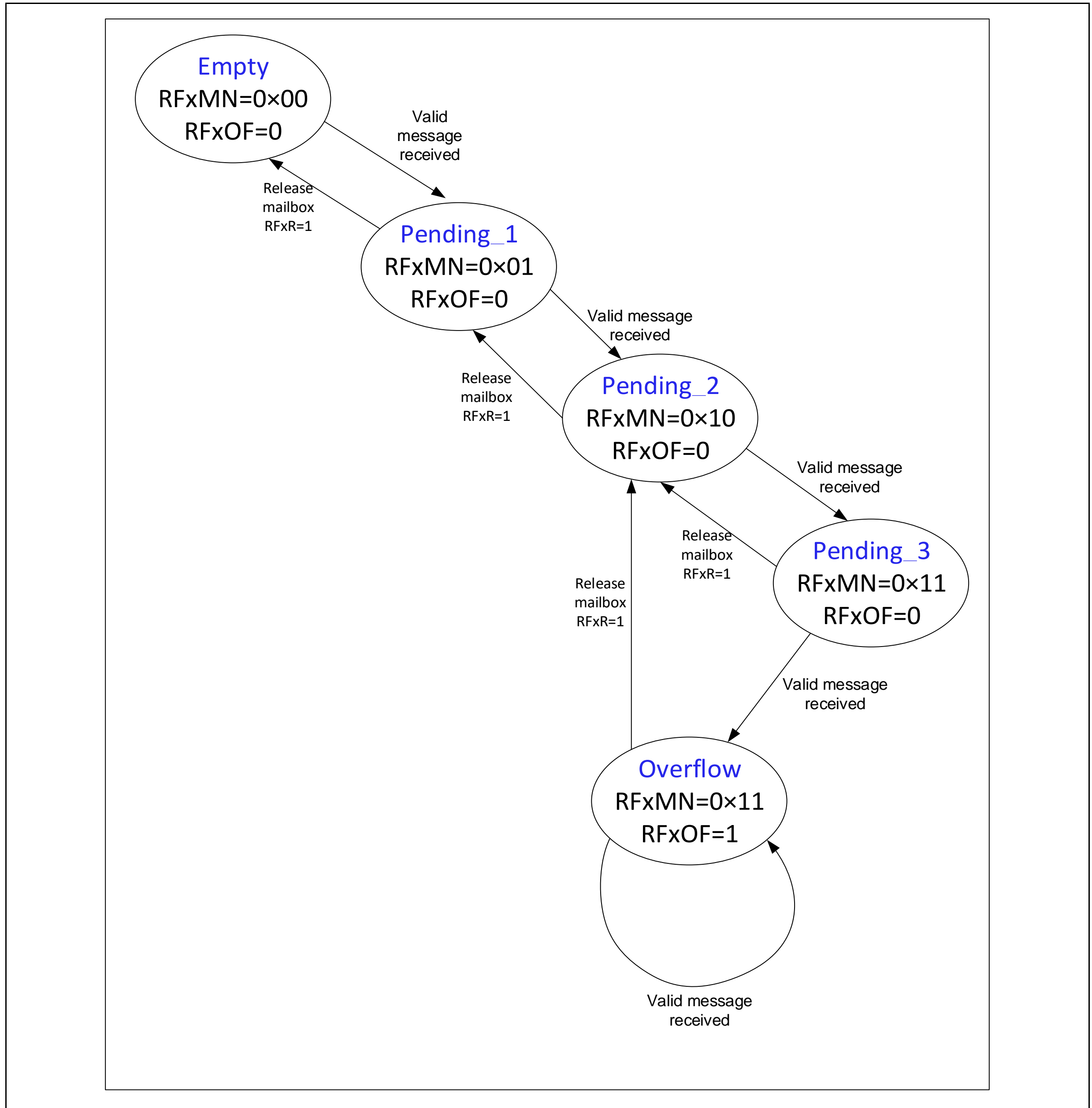
Figure 11 shows the following flag bits and operation bits:

RFxMN bit: Number of valid messages in FIFO (range: 0~3)

RFxOF bit: Overflow flag

RFxR bit: Release mailbox

Figure 11 Message reception process



3.4 Filter

In CAN protocol, the message ID does not represent the node address but is related to the message content. Therefore, the transmitter broadcasts the message to all receiver, and the node determines whether or not this message is required by software according to the ID value. If required, this message is stored in the receive FIFO, which can be obtained by users by reading the receive mailbox register; if not required, this message is discarded, without intervention by software. To meet these requirements, AT32 CAN controller provides 28 identifier filter banks (28 filter banks (0~27) for the AT32F435 series, and 14 filter banks (0~13) for the AT32F403A series; refer to the reference manual of the corresponding series) to receive messages required by software. The filtering process does not require software operation after the required ID is configured by the user.

Filter bit width

Each filter bank has two 32-bit registers (CAN_FiFB1 and CAN_FiFB2). The filter bit width can be configured as two 16 bits or one 32 bits, depending on the FBWSELx bit in the CAN_FBWCFG

register.

One 32-bit filter register CAN_FiFBx includes the SID[10:0], EID[17:0], IDT and RTR bits.

Two 16-bit filter register CAN_FiFBx includes the SID[10:0], IDT, RTR and EID[17:15] bits.

Filtering mode

The filter can be configured in identifier mask mode or in identifier list mode by setting the FMSELx bit in the CAN_FMCFG register. The mask mode is used to specify which bits must match the pre-programmed identifiers, and which bits do not need. In identifier list mode, the identifier must match the pre-programmed identifier.

The two modes can be used in conjunction with filter width to deliver four filtering modes below:

Figure 12 32-bit identifier mask mode

ID	CAN_FiFB1[31:21]	CAN_FiFB1[20:3]	CAN_FiFB1 [2:0]		
Mask	CAN_FiFB2[31:21]	CAN_FiFB2[20:3]	CAN_FiFB2 [2:0]		
Mapping	SID[10:0]	EID[17:0]	IDT	RTR	0

Figure 13 32-bit identifier list mode

ID	CAN_FiFB1[31:21]	CAN_FiFB1[20:3]	CAN_FiFB1 [2:0]		
ID	CAN_FiFB2[31:21]	CAN_FiFB2[20:3]	CAN_FiFB2 [2:0]		
Mapping	SID[10:0]	EID[17:0]	IDT	RTR	0

Figure 14 16-bit identifier mask mode

ID	CAN_FiFB1[15:5]	CAN_FiFB1[4:0]		
Mask	CAN_FiFB1[31:21]	CAN_FiFB1[20:16]		
ID	CAN_FiFB2[15:5]	CAN_FiFB2[4:0]		
Mask	CAN_FiFB2[31:21]	CAN_FiFB2[20:16]		
Mapping	SID[10:0]	RTR	IDT	EID[17:15]

Figure 15 16-bit identifier list mode

ID	CAN_FiFB1[15:8]	CAN_FiFB1[7:0]		
ID	CAN_FiFB1[31:24]	CAN_FiFB1[23:16]		
ID	CAN_FiFB2[15:8]	CAN_FiFB2[7:0]		
ID	CAN_FiFB2[31:24]	CAN_FiFB2[23:16]		
Mapping	SID[10:0]	RTR	IDT	EID[17:15]

For details about filter match number and priority rules, refer to the reference manual of the corresponding series. The filter configuration process is detailed in the example of CAN acceptance filter application.

3.5 CAN baud rate and sample point

The nominal bit time of the CAN bus consists of three parts as follows:

Synchronization segment (SYNC_SEG): This segment has one time unit (Tq) and its time duration is defined by the BRDIV[11:0] bit in the CAN_BTMG register.

Bit segment 1 (BSEG1): Its duration is between 1~16Tq, defined by the BTS1[3:0] bit in the CAN_BTMG register.

Bit segment 2 (BSEG2): Its duration is between 1~8Tq, defined by the BTS2[2:0] bit in the CAN_BTMG register.

Users can configure the CAN baud rate and sample point by setting the CAN timing register. Since the frequency of each node may be different, the baud rates and sampling points of all nodes on CAN bus are different (ideally consistent). In actual applications, users should first ensure that the baud rate is consistent and minimize the deviation of sample points, so that CAN bus can support mode nodes and longer line.

3.5.1 Baud rate formula

$$BaudRate = \frac{1}{\text{Nomal Bit Timimg}}$$

$$Nomal\ Bit\ Timing = t_{SYNC_SEG} + t_{BSEG1} + t_{BSEG2}$$

Where,

$$t_{SYNC_SEG} = 1 \text{ x } t_q$$
$$t_{BSEG1} = (1 + \text{BTS1}[3:0]) \text{ x } t_q$$
$$t_{BSEG2} = (1 + \text{BTS2}[2:0]) \text{ x } t_q$$
$$t_q = (1 + \text{BRDIV}[11:0]) \text{ x } t_{pclk}$$

For example, the BSP routine project\at_start_f437\examples\can\communication_mode:

APB clock: APB1_CLK = 144MHZ

CAN division: BRDIV = 12

At this time, $1\text{Tq} = \frac{1}{144\text{MHZ}/12} = \frac{1}{12}us$

Synchronization segment: SYNC_SEG = 1Tq (fixed)

Bit segment 1: BSEG1 = 8Tq (BTS1[3:0] = 7)

Bit segment 2: BSEG2 = 3Tq (BTS2[2:0] = 2)

Therefore, $Nomal\ Bit\ Timimg = 1\text{Tq} * (\text{SYNC}_{\text{SEG}} + \text{BSEG1} + \text{BSEG12}) = 1us$

$$BaudRate = \frac{1}{\text{Nomal Bit Timimg}} = \frac{1}{1\text{us}} = 1M\text{bps}$$

3.5.2 Sample point formula

$$sample\ point = \frac{\text{SYNC_SEG} + \text{BSEG1}}{\text{SYNC_SEG} + \text{BSEG1} + \text{BSEG2}}$$

Take the same example as above:

At this time, $sample\ point = \frac{1+8}{1+8+3} = 75\%$

The CAN protocol does not specifies sample points, and the suggested sample point settings are given in Table 3 below:

Table 3 Sample point settings

Baud rate	Sample point

>800Kbps	75%
>500Kbps	80%
<=500Kbps	87.5%

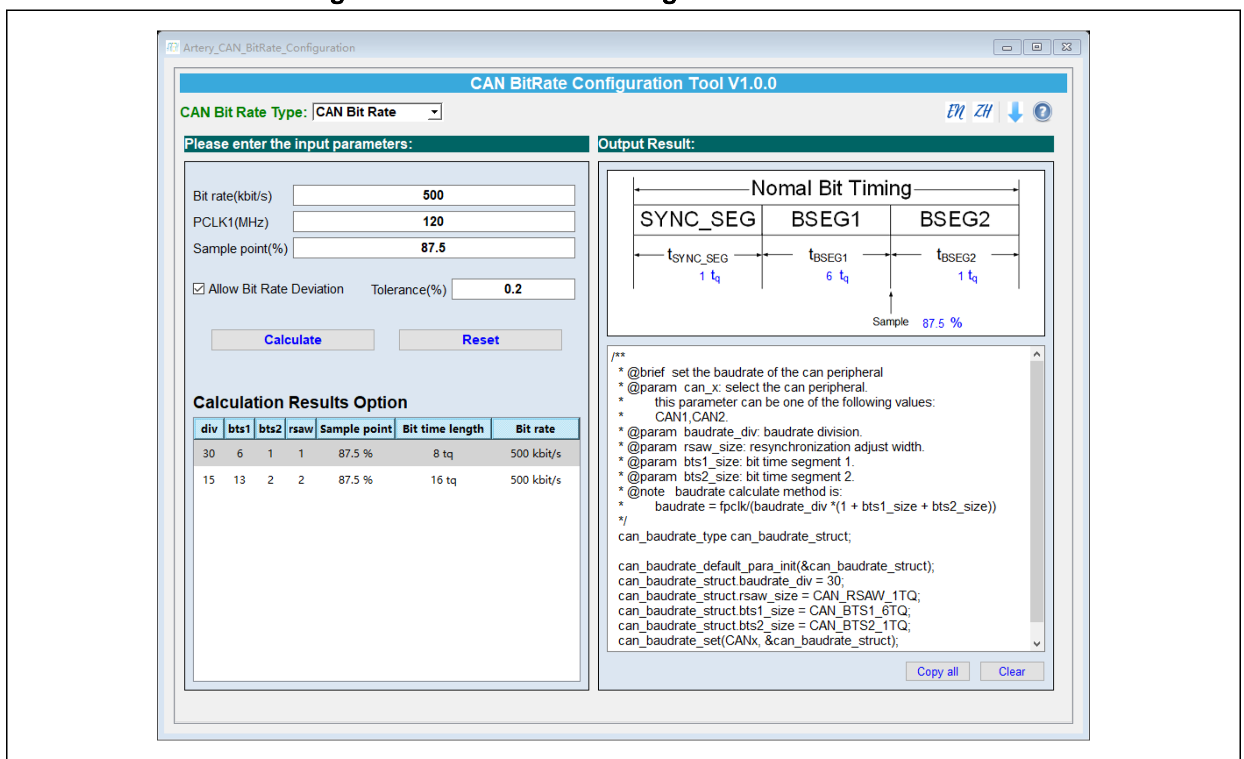
3.5.3 Baud rate configuration tool

For the convenience of baud rate configuration, this application note introduces the CAN BitRate Configuration specially developed by Artery.

Operation steps:

1) **Baud rate setting:** High-speed CAN offers baud rates of up to 1 M, and the frequently used baud rates are 125K, 250K, 333K, 500K and 1M. Users can set the baud rate as needed in the "Bit rate (Kbit/)" in Figure 16.
2) **CAN clock source frequency setting:** Refer to "PCLK1(MHZ)" in Figure 16.
3) **Sample point setting:** After the baud rate is set, a recommended sample point is filled in automatically. The sample point is kept as default if there is no specific requirement, or it can be adjusted as required. Refer to the "sample point (%)" in Figure 16.
4) **Baud rate deviation:** Tick the "allow bit rate deviation" only if there is no satisfactory calculation result. Since the baud rate error of nodes in the same CAN network will increase the probability of communication error, it is recommended to set the "tolerance" as small as possible. Refer to the "allow bit rate deviation" and "tolerance" in Figure 16.
5) **Baud rate configuration selection:** Multiple groups of results are obtained based on the above settings. Select one group of results, and the corresponding software code configuration is displayed on the right. Click on "copy all" to copy the corresponding codes.

Figure 16 CAN BitRate configuration tool



4 Application case 1: CAN communication in normal mode

Note: All projects are built around keil 5. If users want to use them in other compiling environments, please refer to AT32xxx_Firmware_Library_V2.x.x\project\at_start_xxx\templates (such as IAR6/7, keil 4/5) for a simple change.

4.1 Function

Realize communication between two CAN nodes.

4.2 Resources

1) Hardware

Two sets of AT-START BOARDs of the corresponding series + CAN level converters

Take the bsp demo as an example:

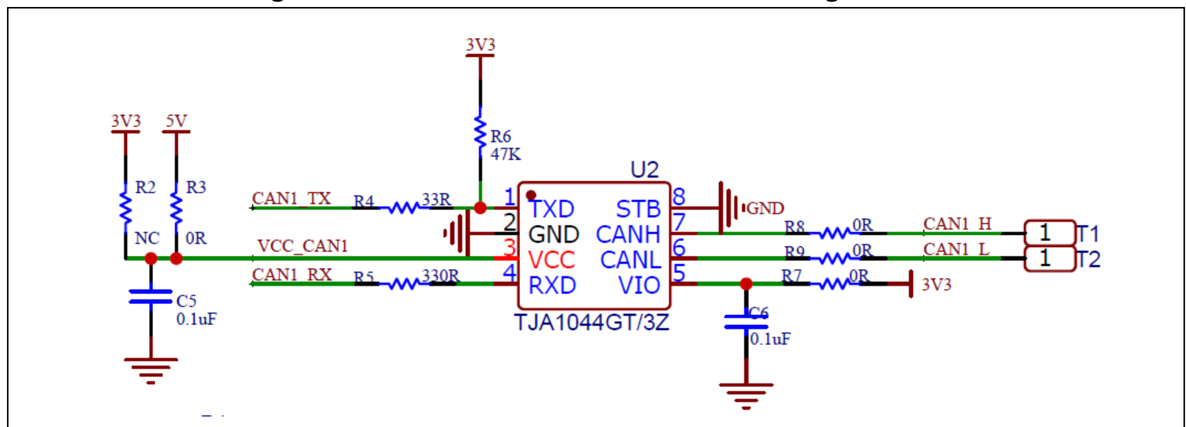
CAN1_TX(PB9) is connected to level converter TXD;

CAN1_RX(PB8) is connected to level converter RXD;

CANH and CANL are connected to the level converters respectively.

The hardware design of CAN level converter is shown below.

Figure 17 CAN level converter schematics diagram



2) Software

project\at_start_f435\examples\can\communication_mode

4.3 Software design

1) Configuration process

- Configure GPIO pins corresponding to CAN1 TX and RX pins
- Configure CAN basic settings
- Configure CAN baud rate
- Configure CAN filter
- Configure CAN interrupt

2) Code

- Main function

```
int main(void)
{
   system_clock_config(); /* Configure system clock */
   at32_board_init();/*AT-START BOARD basic settings, such as LED initialization, delay initialization, 
etc. */
   nvic_priority_group_config(NVIC_PRIORITY_GROUP_4); /*Configure interrupt group priority */
   can_gpio_config();/*Configure the GPIO corresponding to CAN1 */
   can_configuration(); /* Configure CAN, including CAN basic settings, baud rate, filter and interrupt */
   while(1)
   {
      can_transmit_data(); /* CAN1 transmits a data frame */
      at32_led_toggle(LED4); /* Toggle LED4 */
      delay_sec(1); /* delay for 1 second */
   }
}
```

- CAN configuration function

```
static void can_configuration(void)
{
   can_base_type can_base_struct;
   can_baudrate_type can_baudrate_struct;
   can_filter_init_type can_filter_init_struct;

   /* Enable the can clock */
   crm_periph_clock_enable(CRM_CAN1_PERIPH_CLOCK, TRUE); /* Enable CAN clock */

   /* CAN basic settings */
   can_default_para_init(&can_base_struct); /* Initialize CAN base configuration structure */
   can_base_struct.mode_selection = CAN_MODE_COMMUNICATE; /* CAN communication mode: 
normal mode */
   can_base_struct.ttc_enable = FALSE; /* CAN time triggered communication mode (timestamp): 
Disabled */
   can_base_struct.aebo_enable = TRUE; /* Automatic exit bus-off mode: Enabled */
   can_base_struct.aed_enable = TRUE; /* Automatic exit doze mode: Enabled */
   can_base_struct.prsf_enable = FALSE; /* Prohibit auto retransmission: Disabled (that is, auto 
retransmission enabled, as specified in CAN protocol) */
   can_base_struct.mdrsel_selection = CAN_DISCARDING_FIRST_RECEIVED; /* Message 
discarding rule when overflow: the previous message is discarded, and save the new incoming 
massage */
   can_base_struct.mmssr_selection = CAN_SENDING_BY_ID; /* Multiple message transmission 
priority: message with the smallest identifier is first transmitted */
   can_base_init(CAN1, &can_base_struct); /* Write the above base configurations into CAN master 
control register */

   /* CAN baud rate configuration
   can baudrate, set boudrate = pclk/(baudrate_div *(1 + bts1_size + bts2_size))
```

```
  pclk=144M;
  boudrate = 144/(12*(1+8+3))=1Mbps
   */
  can_baudrate_struct.baudrate_div = 12; /* CAN division: 12 */
  can_baudrate_struct.rsaw_size = CAN_RSAW_1TQ; /* CAN sync jump width: 1Tq */
  can_baudrate_struct.bts1_size = CAN_BTS1_8TQ; /* CAN bit segment 1: 8Tq */
  can_baudrate_struct.bts2_size = CAN_BTS2_3TQ; /* CAN bit segment 2: 3Tq */
  can_baudrate_set(CAN1, &can_baudrate_struct); /* Write the above base configurations into CAN
bit timing register */
  /* CAN filter configuration */
  can_filter_init_struct.filter_activate_enable = TRUE; /* Enable filter */
  can_filter_init_struct.filter_mode = CAN_FILTER_MODE_ID_MASK; /* Filter mode: identifier mask */
  can_filter_init_struct.filter_fifo = CAN_FILTER_FIFO0; /* Filter relation FIFO (FIFO1/FIFO1) select:
FIFO0 */
  can_filter_init_struct.filter_number = 0; /* Filter bank select (0~27): 0 */
  can_filter_init_struct.filter_bit = CAN_FILTER_32BIT; /* Filter bit width: 32 bits */
  can_filter_init_struct.filter_id_high = 0; /* Filter ID high: 0 */
  can_filter_init_struct.filter_id_low = 0; /* Filter ID low: 0 */
  can_filter_init_struct.filter_mask_high = 0; /* Filter identifier mask high: 0 (all IDs can pass)*/
  can_filter_init_struct.filter_mask_low = 0; /* Filter identifier mask low: 0 (all IDs can pass) */
  can_filter_init(CAN1, &can_filter_init_struct); /* Write the above base configurations into relevant
 CAN filter registers */

  /* CAN interrupt configuration */
  nvic_irq_enable(CAN1_SE_IRQn, 0x00, 0x00); /* Interrupt priority setting and enable: CAN1 status
change/error interrupt */
  nvic_irq_enable(CAN1_RX0_IRQn, 0x00, 0x00); /* Interrupt priority setting and enable: CAN1
FIFO0 receive interrupt */
  can_interrupt_enable(CAN1, CAN_RF0MIEN_INT, TRUE); /* CAN1 FIFO0 non-empty interrupt
enable: An interrupt is generated when FIFO0 receives a frame of valid data */
  can_interrupt_enable(CAN1, CAN_ETRIEN_INT, TRUE); /* Error type record interrupt enable: An
interrupt is generated when ETR[2:0] is not equal to 0 */
  can_interrupt_enable(CAN1, CAN_EOIEN_INT, TRUE); /* CAN1 error interrupt enable: Master
switch for all error interrupts */
}
```

- CAN transmit function

```
static void can_transmit_data(void)
{
  uint8_t transmit_mailbox;
  can_tx_message_type tx_message_struct;
  tx_message_struct.standard_id = 0x400; /* Set the to-be-transmitted data ID=0x400 */
  tx_message_struct.extended_id = 0;
  tx_message_struct.id_type = CAN_ID_STANDARD; /* To-be-transmitted data type
(standard/extended): standard data frame */
  tx_message_struct.frame_type = CAN_TFT_DATA; /* To-be-transmitted frame type (remote/data):
```

```
Data frame */
   tx_message_struct.dlc = 8; /* To-be-transmitted data length (0~8): 8 */
   tx_message_struct.data[0] = 0x11; /* 1st byte: 0x11 */
   tx_message_struct.data[1] = 0x22; /* 2nd byte: 0x22 */
   tx_message_struct.data[2] = 0x33; /* 3rd byte: 0x33 */
   tx_message_struct.data[3] = 0x44; /* 4th byte: 0x44 */
   tx_message_struct.data[4] = 0x55; /* 5th byte: 0x55 */
   tx_message_struct.data[5] = 0x66; /* 6th byte: 0x66 */
   tx_message_struct.data[6] = 0x77; /* 7th byte: 0x77 */
   tx_message_struct.data[7] = 0x88; /* 8th byte: 0x88 */
   transmit_mailbox = can_message_transmit(CAN1, &tx_message_struct); /* Write the above to-be-
transmitted message into the transmit mailbox and request to send */
   while(can_transmit_status_get(CAN1, (can_tx_mailbox_num_type)transmit_mailbox) !=
CAN_TX_STATUS_SUCCESSFUL); /* Wait for transmit mailbox transmission success (the
transmission success flag of the corresponding transmit mailbox is set) */
}
```

- CAN receive interrupt handler

```
void CAN1_RX0_IRQHandler(void)
{
   can_rx_message_type rx_message_struct;
   if(can_flag_get(CAN1,CAN_RF0MN_FLAG) != RESET) /* FIFO0 non-empty judgment (number of
message >0) */
   {
      can_message_receive(CAN1, CAN_RX_FIFO0, &rx_message_struct); /* Read a message,
including the ID, data length and data, etc. */
      if(rx_message_struct.standard_id == 0x400) /* Check whether the received message is in
standard frame format whose ID=0x400 */
         at32_led_toggle(LED2); /* If the received message is in standard frame format with ID=0x400,
toggle LED2 */
      else
         at32_led_toggle(LED3); /* If not, toggle LED3 */
   }
}
```

- GPIO configuration function

```
static void can_gpio_config(void)
{
   gpio_init_type gpio_init_struct;
   /* enable the gpio clock */
   crm_periph_clock_enable(CRM_GPIOB_PERIPH_CLOCK, TRUE); /* Enable the corresponding
GPIOB clock */
   gpio_default_para_init(&gpio_init_struct); /* Initialize GPIO configuration structure */

   /* configure the can tx, rx pin */
```

```
  gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER; /* GPIO drive strength: strong */
  gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL; /* FPIO output mode: push-pull output */
  gpio_init_struct.gpio_mode = GPIO_MODE_MUX; /* GPIO mode: multiplexed*/
  gpio_init_struct.gpio_pins = GPIO_PINS_9 | GPIO_PINS_8; /* GPIO pins: pin8 & pin9 */
  gpio_init_struct.gpio_pull = GPIO_PULL_NONE; /* GPIO pull configuration: no pull-up, no pull-down */
  gpio_init(GPIOB, &gpio_init_struct); /* Write the above configuration into the corresponding register */

  gpio_pin_mux_config(GPIOB, GPIO_PINS_SOURCE9, GPIO_MUX_9); /* Configure GPIOB_pin9 mux9 (CAN_TX) */
  gpio_pin_mux_config(GPIOB, GPIO_PINS_SOURCE8, GPIO_MUX_9); /* Configure GPIOB_pin8 mux9 (CAN_RX) */
```

4.4 Test result

- If the data transmission is correct, the LED2 lights on both AT-START BOARDs blink, indicating a message in standard data format with ID=0x400 is received, and LED4 keeps blinking, indicating the program is working properly.

5 Application case 2: CAN receive filter

Note: All projects are built around keil 5. If users want to use them in other compiling environments, please refer to AT32xxx_Firmware_Library_V2.x.x\project\at_start_xxx\templates (such as IAR6/7, keil 4/5) for a simple change.

5.1 Function

Realize message filtering; receive the required message and discard the undesired.

5.2 Resources

1) Hardware

Two sets of AT-START BOARDs of the corresponding series + CAN level converters

Take the bsp demo as an example:

CAN1_TX(PB9) is connected to level converter TXD;

CAN1_RX(PB8) is connected to level converter RXD;

CANH and CANL are connected to the level converters respectively.

The hardware design of CAN level converter is shown below.

Figure 18 CAN level converter schematics diagram

2) Software

project\at_start_f435\examples\can\ filter

5.3 Software design

1) Configuration process

- Configure pins corresponding to CAN1 TX and RX pins
- Configure CAN basic settings
- Configure CAN baud rate
- Configure CAN filter
- Configure CAN interrupt

2) Code

- Set IDs that can pass filtering

```
/* extended identifier */
#define FILTER_EXT_ID1                    ((uint32_t)0x18F5F100)
#define FILTER_EXT_ID2                    ((uint32_t)0x18F5F200)

/* standard identifier */
#define FILTER_STD_ID1                    ((uint16_t)0x04F6)
#define FILTER_STD_ID2                    ((uint16_t)0x04F7)
```

- Main function

```
int main(void)
{
  system_clock_config(); /* Configure system clock */
  at32_board_init(); /* Configure AT-START BOARD basic settings, such as LED initialization, delay
initialization, etc. */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4); /* Configure interrupt group priority */
  can_gpio_config(); /* Configure the GPIO corresponding to CAN1 */
  can_configuration(); /* Configure CAN, including CAN basic settings, baud rate, filter and interrupt */
can_transmit_data(); /* CAN1 transmits 4 frames of data (IDs: FILTER_EXT_ID1, FILTER_EXT_ID2,
FILTER_STD_ID1, FILTER_STD_ID2. */
  while(1)
  {
    if(test_result == 4)
    {
      at32_led_toggle(LED2); /* If 4 frames of data are received, toggle LED2/3/4 */
      at32_led_toggle(LED3);
      at32_led_toggle(LED4);
      delay_sec(1); /* delay for 1 second */
    }
  }
}
```

- CAN configuration function

```
static void can_configuration(void)
{
  can_base_type can_base_struct;
  can_baudrate_type can_baudrate_struct;
  can_filter_init_type can_filter_init_struct;

  /* enable the can clock */
  crm_periph_clock_enable(CRM_CAN1_PERIPH_CLOCK, TRUE); /* Enable CAN clock */

  /* CAN basic settings */
  can_default_para_init(&can_base_struct); /* Initialize CAN base configuration structure */
  can_base_struct.mode_selection = CAN_MODE_COMMUNICATE; /* CAN communication mode:
normal mode */
```

```
    can_base_struct.ttc_enable = FALSE; /* CAN time triggered communication mode (timestamp): Disabled */
    can_base_struct.aebo_enable = TRUE; /* Automatic exit bus-off mode: Enabled */
    can_base_struct.aed_enable = TRUE; /* Automatic exit doze mode: Enabled */
    can_base_struct.prsf_enable = FALSE; /* Prohibit auto retransmission: Disabled (that is, auto retransmission enabled, as specified in CAN protocol) */
    can_base_struct.mdrsel_selection = CAN_DISCARDING_FIRST_RECEIVED; /* Message discarding rule when overflow: the previous message is discarded, and save the new incoming massage */
    can_base_struct.mmssr_selection = CAN_SENDING_BY_ID; /* Multiple message transmission priority: message with the smallest identifier is first transmitted */
    can_base_init(CAN1, &can_base_struct); /* Write the above configuration into CAN master control register */

    /* CAN baud rate configuration:
    can baudrate, set boudrate = pclk/(baudrate_div *(1 + bts1_size + bts2_size))
    pclk=144M;
    boudrate = 144/(12*(1+8+3))=1Mbps
     */
    can_baudrate_struct.baudrate_div = 12; /* CAN division: 12 */
    can_baudrate_struct.rsaw_size = CAN_RSAW_1TQ; /* CAN sync jump width: 1Tq */
    can_baudrate_struct.bts1_size = CAN_BTS1_8TQ; /* CAN bit segment 1: 8Tq */
    can_baudrate_struct.bts2_size = CAN_BTS2_3TQ; /* CAN bit segment 2: 3Tq */
    can_baudrate_set(CAN1, &can_baudrate_struct); /* Write the above base configurations into CAN bit timing register */
    /* CAN filter configuration */
    /* can filter 0 config */
    can_filter_init_struct.filter_activate_enable = TRUE; /* Enable filter */
    can_filter_init_struct.filter_mode = CAN_FILTER_MODE_ID_LIST; /* Filter mode: identifier list mode */
    can_filter_init_struct.filter_fifo = CAN_FILTER_FIFO0; /* Filter relation FIFO (FIFO1/FIFO1): select FIFO0 */
    can_filter_init_struct.filter_number = 0; /* Filter bank select (0~27): 0 */
    can_filter_init_struct.filter_bit = CAN_FILTER_32BIT; /* Filter bit width: 32 bits */
    can_filter_init_struct.filter_id_high = (((FILTER_EXT_ID1 << 3) >> 16) & 0xFFFF); /* Configure the extended ID that can pass filtering: FILTER_EXT_ID1 */
    can_filter_init_struct.filter_id_low = ((FILTER_EXT_ID1 << 3) & 0xFFFF) | 0x04;
    can_filter_init_struct.filter_mask_high = ((FILTER_EXT_ID2 << 3) >> 16) & 0xFFFF; /* Configure the extended ID that can pass filtering: FILTER_EXT_ID2 */
    can_filter_init_struct.filter_mask_low = ((FILTER_EXT_ID2 << 3) & 0xFFFF) | 0x04;
    can_filter_init(CAN1, &can_filter_init_struct); /* Write the above configuration into relevant filter register */
    /* can filter 1 config */
    can_filter_init_struct.filter_activate_enable = TRUE; /* Enable filter */
    can_filter_init_struct.filter_mode = CAN_FILTER_MODE_ID_LIST; /* Filter mode: identifier list mode */
```

```
    can_filter_init_struct.filter_fifo = CAN_FILTER_FIFO0; /* Filter relation FIFO (FIFO1/FIFO1) select FIFO0 */
    can_filter_init_struct.filter_number = 1; /* Filter bank select (0~27):1 */
    can_filter_init_struct.filter_bit = CAN_FILTER_32BIT; /* Filter bit width: 32 bits */
    can_filter_init_struct.filter_id_high = FILTER_STD_ID1 << 5; /* Configure the standard ID that can pass filtering: FILTER_STD_ID1 */
    can_filter_init_struct.filter_id_low = 0;
    can_filter_init_struct.filter_mask_high = FILTER_STD_ID2 << 5; /* Configure the standard ID that can pass filtering: FILTER_STD_ID2 */
    can_filter_init_struct.filter_mask_low = 0;
    can_filter_init(CAN1, &can_filter_init_struct); /* Write the above configuration into relevant filter register */
    /* CAN interrupt configuration */
    nvic_irq_enable(CAN1_SE_IRQn, 0x00, 0x00); /* Interrupt priority setting and enable: CAN1 status change/error interrupt */
    nvic_irq_enable(CAN1_RX0_IRQn, 0x00, 0x00); /* Interrupt priority setting and enable: CAN1 FIFO0 receive interrupt */
    can_interrupt_enable(CAN1, CAN_RF0MIEN_INT, TRUE); /* CAN1 FIFO0 non-empty interrupt enable: An interrupt is generated when FIFO0 receives a frame of valid data */
    can_interrupt_enable(CAN1, CAN_ETRIEN_INT, TRUE); /* Error type record interrupt enable: An interrupt is generated when ETR[2:0] is not equal to 0 */
    can_interrupt_enable(CAN1, CAN_EOIEN_INT, TRUE); /* CAN1 error interrupt enable: Master switch for all error interrupts */
}
```

- CAN transmit function

```
static void can_transmit_data(void)
{
    uint8_t transmit_mailbox;
    can_tx_message_type tx_message_struct;

    /* transmit FILTER_STD_ID1 */
    tx_message_struct.standard_id = FILTER_STD_ID1;/* Set the to-be-transmit data ID= FILTER_STD_ID1 */
    tx_message_struct.extended_id = 0;
    tx_message_struct.id_type = CAN_ID_STANDARD; /* To-be-transmitted data type (standard/extended): standard data frame */
    tx_message_struct.frame_type = CAN_TFT_DATA; /* To-be-transmitted frame type (remote/data): data frame */
    tx_message_struct.dlc = 8; /* To-be-transmitted data length (0~8): 8 */
    tx_message_struct.data[0] = 0x11; /* 1st byte: 0x11 */
    tx_message_struct.data[1] = 0x22; /* 2nd byte: 0x22 */
    tx_message_struct.data[2] = 0x33; /* 3rd byte: 0x33 */
    tx_message_struct.data[3] = 0x44; /* 4th byte: 0x44 */
    tx_message_struct.data[4] = 0x55; /* 5th byte: 0x55 */
    tx_message_struct.data[5] = 0x66; /* 6th byte: 0x66 */
```

```
    tx_message_struct.data[6] = 0x77; /* 7th byte: 0x77 */
    tx_message_struct.data[7] = 0x88; /* 8th byte: 0x88 */
    transmit_mailbox = can_message_transmit(CAN1, &tx_message_struct); /* Write the above to-be-
transmitted message into the transmit mailbox and request to send */
    while(can_transmit_status_get(CAN1, (can_tx_mailbox_num_type)transmit_mailbox) !=
CAN_TX_STATUS_SUCCESSFUL); /* Wait for transmit mailbox transmission success (the
transmission success flag of the corresponding transmit mailbox is set) */

    /* transmit FILTER_STD_ID2 */
    tx_message_struct.standard_id = FILTER_STD_ID2;/* Set the to-be-transmit data ID=
FILTER_STD_ID2 */
    tx_message_struct.extended_id = 0;
    tx_message_struct.id_type = CAN_ID_STANDARD; /* To-be-transmitted data type
(standard/extended): standard data frame */
    tx_message_struct.frame_type = CAN_TFT_DATA; /* To-be-transmitted frame type (remote/data):
data frame */
    tx_message_struct.dlc = 8; /* To-be-transmitted data length (0~8): 8 */
    tx_message_struct.data[0] = 0x11; /* 1st byte: 0x11 */
    tx_message_struct.data[1] = 0x22; /* 2nd byte: 0x22 */
    tx_message_struct.data[2] = 0x33; /* 3rd byte: 0x33 */
    tx_message_struct.data[3] = 0x44; /* 4th byte: 0x44 */
    tx_message_struct.data[4] = 0x55; /* 5th byte: 0x55 */
    tx_message_struct.data[5] = 0x66; /* 6th byte: 0x66 */
    tx_message_struct.data[6] = 0x77; /* 7th byte: 0x77 */
    tx_message_struct.data[7] = 0x88; /* 8th byte: 0x88 */
    transmit_mailbox = can_message_transmit(CAN1, &tx_message_struct); /* Write the above to-be-
transmitted message into the transmit mailbox and request to send */
    while(can_transmit_status_get(CAN1, (can_tx_mailbox_num_type)transmit_mailbox) !=
CAN_TX_STATUS_SUCCESSFUL); /* Wait for transmit mailbox transmission success (the
transmission success flag of the corresponding transmit mailbox is set) */

    /* transmit FILTER_EXT_ID1 */
    tx_message_struct.standard_id = 0;
    tx_message_struct.extended_id = FILTER_EXT_ID1;/* Set the to-be-transmit extended data frame
ID= FILTER_EXT_ID1 */
    tx_message_struct.id_type = CAN_ID_EXTENDED; /* To-be-transmitted data type
(standard/extended): extended data frame */
    tx_message_struct.frame_type = CAN_TFT_DATA; /* To-be-transmitted frame type (remote/data):
data frame */
    tx_message_struct.dlc = 8; /* To-be-transmitted data length (0~8): 8 */
    tx_message_struct.data[0] = 0x11; /* 1st byte: 0x11 */
    tx_message_struct.data[1] = 0x22; /* 2nd byte: 0x22 */
    tx_message_struct.data[2] = 0x33; /* 3rd byte: 0x33 */
    tx_message_struct.data[3] = 0x44; /* 4th byte: 0x44 */
    tx_message_struct.data[4] = 0x55; /* 5th byte: 0x55 */
    tx_message_struct.data[5] = 0x66; /* 6th byte: 0x66 */
```

```
  tx_message_struct.data[6] = 0x77; /* 7th byte: 0x77 */
  tx_message_struct.data[7] = 0x88; /* 8th byte: 0x88 */
  transmit_mailbox = can_message_transmit(CAN1, &tx_message_struct); /* Write the above to-be-
transmitted message into the transmit mailbox and request to send */
  while(can_transmit_status_get(CAN1, (can_tx_mailbox_num_type)transmit_mailbox) !=
CAN_TX_STATUS_SUCCESSFUL); /* Wait for transmit mailbox transmission success (the
transmission success flag of the corresponding transmit mailbox is set) */

  /* transmit FILTER_EXT_ID2 */
  tx_message_struct.standard_id = 0;
  tx_message_struct.extended_id = FILTER_EXT_ID2;/* Set the to-be-transmit extended data frame
ID= FILTER_EXT_ID2 */
  tx_message_struct.id_type = CAN_ID_EXTENDED; /* To-be-transmitted data type
(standard/extended): extended data frame */
  tx_message_struct.frame_type = CAN_TFT_DATA; /* To-be-transmitted frame type (remote/data):
data frame */
  tx_message_struct.dlc = 8; /* To-be-transmitted data length (0~8): 8*/
  tx_message_struct.data[0] = 0x11; /* 1st byte: 0x11 */
  tx_message_struct.data[1] = 0x22; /* 2nd byte: 0x22 */
  tx_message_struct.data[2] = 0x33; /* 3rd byte: 0x33 */
  tx_message_struct.data[3] = 0x44; /* 4th byte: 0x44 */
  tx_message_struct.data[4] = 0x55; /* 5th byte: 0x55 */
  tx_message_struct.data[5] = 0x66; /* 6th byte: 0x66 */
  tx_message_struct.data[6] = 0x77; /* 7th byte: 0x77 */
  tx_message_struct.data[7] = 0x88; /* 8th byte: 0x88 */
  transmit_mailbox = can_message_transmit(CAN1, &tx_message_struct); /* Write the above to-be-
transmitted message into the transmit mailbox and request to send */
  while(can_transmit_status_get(CAN1, (can_tx_mailbox_num_type)transmit_mailbox) !=
CAN_TX_STATUS_SUCCESSFUL); /* Wait for transmit mailbox transmission success (the
transmission success flag of the corresponding transmit mailbox is set) */
}
```

■ CAN receive interrupt handler

```
void CAN1_RX0_IRQHandler(void)
{
  can_rx_message_type rx_message_struct;
  if(can_flag_get(CAN1,CAN_RF0MN_FLAG) != RESET) /* non-empty judgment (number of
message >0) */
  {
    if(test_result == 4)
    {
      test_result = 0;
    }
    can_message_receive(CAN1, CAN_RX_FIFO0, &rx_message_struct); /* Read a message,
including the ID, data length and data, etc. */
    if((rx_message_struct.id_type == CAN_ID_STANDARD) && (rx_message_struct.standard_id ==
```

```
FILTER_STD_ID1))
        test_result++; /* If a standard frame (ID= FILTER_STD_ID1) is received, the test_result flag is incremented by 1 */
      else if((rx_message_struct.id_type == CAN_ID_STANDARD) && (rx_message_struct.standard_id == FILTER_STD_ID2))
        test_result++;/* If a standard frame (ID= FILTER_STD_ID2) is received, the test_result flag is incremented by 1 */
      else if((rx_message_struct.id_type == CAN_ID_EXTENDED) && (rx_message_struct.extended_id == FILTER_EXT_ID1))
        test_result++;/* If an extended frame (ID= FILTER_EXT_ID1) is received, test_result flag is incremented by 1 */
      else if((rx_message_struct.id_type == CAN_ID_EXTENDED) && (rx_message_struct.extended_id == FILTER_EXT_ID2))
        test_result++;/* If an extended frame (ID= FILTER_EXT_ID2) is received, test_result flag is incremented by 1 */
  }
}
```

- GPIO configuration function

```
static void can_gpio_config(void)
{
  gpio_init_type gpio_init_struct;
  /* enable the gpio clock */
  crm_periph_clock_enable(CRM_GPIOB_PERIPH_CLOCK, TRUE); /* Enable the corresponding GPIOB clock */
  gpio_default_para_init(&gpio_init_struct); /* Initialize GPIO configuration structure */

  /* configure the can tx, rx pin */
  gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER; /* GPIO drive strength: strong */
  gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL; /* FPIO output mode: push-pull output */
  gpio_init_struct.gpio_mode = GPIO_MODE_MUX; /* GPIO mode: multiplexed */
  gpio_init_struct.gpio_pins = GPIO_PINS_9 | GPIO_PINS_8; /* GPIO pins: pin8 & pin9 */
  gpio_init_struct.gpio_pull = GPIO_PULL_NONE; /* GPIO pull configuration: no pull-up, no pull-down */
  gpio_init(GPIOB, &gpio_init_struct); /* Write the above configuration into the corresponding register */

  gpio_pin_mux_config(GPIOB, GPIO_PINS_SOURCE9, GPIO_MUX_9); /* Configure GPIOB_pin9 mux9 (CAN_TX) */
  gpio_pin_mux_config(GPIOB, GPIO_PINS_SOURCE8, GPIO_MUX_9); /* Configure GPIOB_pin8 mux9 (CAN_RX) */
```

5.4 Test result

- If the data transmission is correct, LED2/3/4 on AT-START BOARD toggle once, indicating four frames of data (ID=FILTER_EXT_ID1, FILTER_EXT_ID2, FILTER_STD_ID1, FILTER_STD_ID2) are received.

6 Application case 3: CAN debugging in loopback mode

Note: Note: All projects are built around keil 5. If users want to use them in other compiling environments, please refer to AT32xxx_Firmware_Library_V2.x.x\project\at_start_xxx\templates (such as IAR6/7, keil 4/5) for a simple change.

6.1 Function

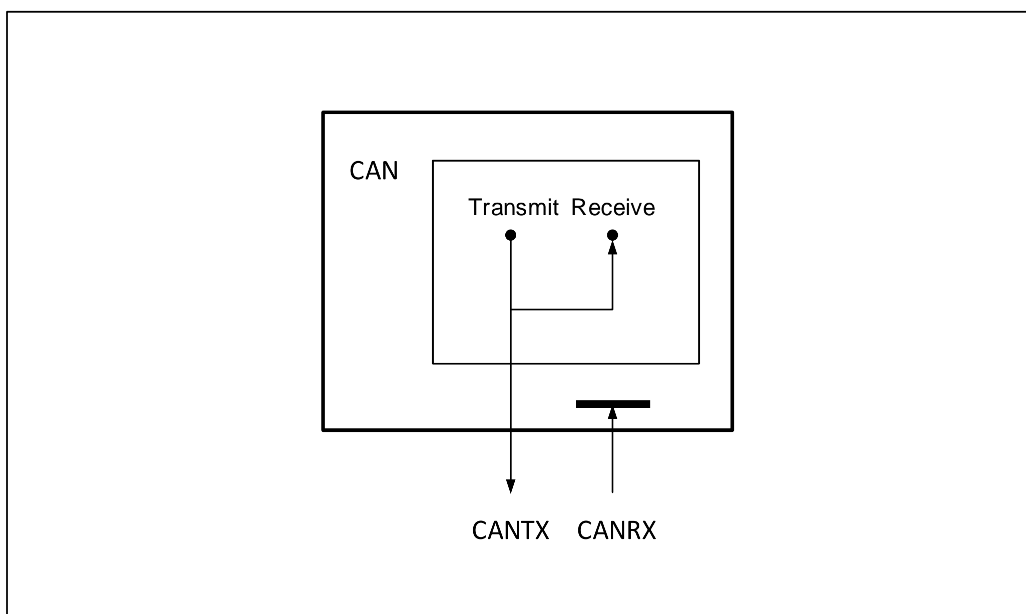
Realize CAN loopback communication.

Loopback mode:

The loopback mode is used for self-test. In this mode, CAN internally feed the TX output back to RX input, ignoring the actual state of CAN_RX pin. Therefore, it is not required to configure the corresponding GPIO pins. If the corresponding GPIO pins are configured, the transmitted message can be detected on CAN_TX, as shown in Figure 19. In addition, to avoid being affected by external factors, CAN core ignores validation errors (the dominant bit is not detected at the acknowledgement bit of data/remote frame) in loopback mode.

The hardware design of level converter (CAN loopback mode) is shown below.

Figure 19 CAN loopback mode



6.2 Resources

1) Hardware

   One AT-START BOARD of the corresponding series

2) Software

   project\at_start_f435\examples\can\loopback_mode

6.3 Software design

1) Configuration process

   - Configure GPIOs corresponding to the CAN1 TX and RX pins (not required in loopback mode)

- Configure CAN basic settings
- Configure CAN baud rate
- Configure CAN filter
- Configure CAN interrupt

2) Code

- Main function

```
int main(void)
{
  system_clock_config(); /* Configure system clock */
  at32_board_init(); /* Configure AT-START BOARD basic settings, such as LED initialization, delay
initialization, etc. */
  nvic_priority_group_config(NVIC_PRIORITY_GROUP_4); /* Configure interrupt group priority */
  can_gpio_config(); /* Configure the GPIO corresponding to CAN1 (not required in loopback mode; if
 it is configured, CAN_TX pin outputs data but CAN_RX pin does not receive external data, as shown
 in Figure 16)   */
  can_configuration(); /* Configure CAN, including CAN basic settings, baud rate, filter and interrupt */
  while(1)
  {
    can_transmit_data(); /* CAN1 transmits one standard data frame */
    at32_led_toggle(LED4); /* Toggle LED4 */
    delay_sec(1); /* delay for 1 second */
  }
}
```

- CAN configuration function

```
static void can_configuration(void)
{
  can_base_type can_base_struct;
  can_baudrate_type can_baudrate_struct;
  can_filter_init_type can_filter_init_struct;

  /* enable the can clock */
  crm_periph_clock_enable(CRM_CAN1_PERIPH_CLOCK, TRUE); /* Enable CAN clock */

  /* CAN basic configuration */
  can_default_para_init(&can_base_struct); /* Initialize CAN base configuration structure */
  can_base_struct.mode_selection = CAN_MODE_LOOPBACK; /* CAN mode: Loopback */
  can_base_struct.ttc_enable = FALSE; /* CAN time triggered communication mode (timestamp):
Disabled */
  can_base_struct.aebo_enable = TRUE; /* Automatic exit bus-off mode: Enabled */
  can_base_struct.aed_enable = TRUE; /* Automatic exit doze mode: Enabled */
  can_base_struct.prsf_enable = FALSE; /* Prohibit auto retransmission: Disabled (that is, auto
retransmission enabled, as specified in CAN protocol) */
  can_base_struct.mdrsel_selection = CAN_DISCARDING_FIRST_RECEIVED; /* Message
```

```
discarding rule when overflow: the previous message is discarded */
  can_base_struct.mmssr_selection = CAN_SENDING_BY_ID; /* Multiple message transmission
priority: message with the smallest identifier is first transmitted */
  can_base_init(CAN1, &can_base_struct); /* Write the above configuration into CAN master control
 register */

  /* CAN baud rate configuration:
  can baudrate, set boudrate = pclk/(baudrate_div *(1 + bts1_size + bts2_size))
  pclk=144M;
  boudrate = 144/(12*(1+8+3))=1Mbps
   */
  can_baudrate_struct.baudrate_div = 12; /* CAN division: 12 */
  can_baudrate_struct.rsaw_size = CAN_RSAW_1TQ; /* CAN sync jump width: 1Tq */
  can_baudrate_struct.bts1_size = CAN_BTS1_8TQ; /* CAN bit segment 1: 8Tq */
  can_baudrate_struct.bts2_size = CAN_BTS2_3TQ; /* CAN bit segment 2: 3Tq */
  can_baudrate_set(CAN1, &can_baudrate_struct); /* Write the above configuration into CAN bit
timing register */
  /* CAN filter configuration */
  can_filter_init_struct.filter_activate_enable = TRUE; /* Enable filter */
  can_filter_init_struct.filter_mode = CAN_FILTER_MODE_ID_MASK; /* Filter mode: identifier mask
mode */
  can_filter_init_struct.filter_fifo = CAN_FILTER_FIFO0; /* Filter relation FIFO (FIFO1/FIFO1): select
FIFO0 */
  can_filter_init_struct.filter_number = 0; /* Filter bank select (0~27): 0 */
  can_filter_init_struct.filter_bit = CAN_FILTER_32BIT; /* Filter bit width: 32 bits */
  can_filter_init_struct.filter_id_high = 0; /* Filter ID high: 0 */
  can_filter_init_struct.filter_id_low = 0; /* Filter ID low: 0 */
  can_filter_init_struct.filter_mask_high = 0; /* Filter identifier mask high: 0 (all IDs can pass) */
  can_filter_init_struct.filter_mask_low = 0; /* Filter identifier mask low: 0 (all IDs can pass) */
  can_filter_init(CAN1, &can_filter_init_struct); /* Write the above base configurations into relevant
CAN filter registers */

  /* CAN interrupt configuration */
  nvic_irq_enable(CAN1_SE_IRQn, 0x00, 0x00); /* Interrupt priority setting and enable: CAN1 status
change/error interrupt */
  nvic_irq_enable(CAN1_RX0_IRQn, 0x00, 0x00); /* Interrupt priority setting and enable: CAN1
FIFO0 receive interrupt */
  can_interrupt_enable(CAN1, CAN_RF0MIEN_INT, TRUE); /* CAN1 FIFO0 non-empty interrupt
enable: An interrupt is generated when FIFO0 receives a frame of valid data */
  can_interrupt_enable(CAN1, CAN_ETRIEN_INT, TRUE); /* Error type record interrupt enable: An
interrupt is generated when ETR[2:0] is not equal to 0*/
  can_interrupt_enable(CAN1, CAN_EOIEN_INT, TRUE); /* CAN1 error interrupt enable: Master
switch for all error interrupts */
}
```

- CAN transmit function

```
static void can_transmit_data(void)
{
   uint8_t transmit_mailbox;
   can_tx_message_type tx_message_struct;
   tx_message_struct.standard_id = 0x400; /* Set the to-be-transmit data ID=0x400 */
   tx_message_struct.extended_id = 0;
   tx_message_struct.id_type = CAN_ID_STANDARD; /* To-be-transmitted data type
(standard/extended): standard data frame */
   tx_message_struct.frame_type = CAN_TFT_DATA; /* To-be-transmitted frame type (remote/data):
data frame */
   tx_message_struct.dlc = 8; /* To-be-transmitted data length (0~8): 8 */
   tx_message_struct.data[0] = 0x11; /* 1st byte: 0x11 */
   tx_message_struct.data[1] = 0x22; /* 2nd byte: 0x22 */
   tx_message_struct.data[2] = 0x33; /* 3rd byte: 0x33 */
   tx_message_struct.data[3] = 0x44; /* 4th byte: 0x44 */
   tx_message_struct.data[4] = 0x55; /* 5th byte: 0x55 */
   tx_message_struct.data[5] = 0x66; /* 6th byte: 0x66 */
   tx_message_struct.data[6] = 0x77; /* 7th byte: 0x77 */
   tx_message_struct.data[7] = 0x88; /* 8th byte: 0x88 */
   transmit_mailbox = can_message_transmit(CAN1, &tx_message_struct); /* Write the above to-be-
transmitted message into the transmit mailbox and request to send */
   while(can_transmit_status_get(CAN1, (can_tx_mailbox_num_type)transmit_mailbox) !=
CAN_TX_STATUS_SUCCESSFUL); /* Wait for transmit mailbox transmission success (the
transmission success flag of the corresponding transmit mailbox is set) */
}
```

- CAN receive interrupt handler

```
void CAN1_RX0_IRQHandler(void)
{
   can_rx_message_type rx_message_struct;
   if(can_flag_get(CAN1,CAN_RF0MN_FLAG) != RESET) /* non-empty judgment (number of
message >0) */
   {
      can_message_receive(CAN1, CAN_RX_FIFO0, &rx_message_struct); /* Read a message,
including the ID, data length and data, etc. */
      if(rx_message_struct.standard_id == 0x400) /* Check whether the received message is in
standard frame format whose ID=0x400 */
         at32_led_toggle(LED2); /* If the received message is in standard frame format with ID=0x400,
toggle LED2 */
      else
         at32_led_toggle(LED3); /* If not, toggle LED3 */
   }
}
```

- GPIO configuration function

```
static void can_gpio_config(void)
{
  gpio_init_type gpio_init_struct;
  /* enable the gpio clock */
  crm_periph_clock_enable(CRM_GPIOB_PERIPH_CLOCK, TRUE); /* Enable the corresponding
GPIOB clock */
  gpio_default_para_init(&gpio_init_struct); /* Initialize GPIO configuration structure */

  /* configure the can tx, rx pin */
  gpio_init_struct.gpio_drive_strength = GPIO_DRIVE_STRENGTH_STRONGER; /* GPIO drive
strength: strong */
  gpio_init_struct.gpio_out_type = GPIO_OUTPUT_PUSH_PULL; /* FPIO output mode: push-pull
output */
  gpio_init_struct.gpio_mode = GPIO_MODE_MUX; /* GPIO mode: multiplexed */
  gpio_init_struct.gpio_pins = GPIO_PINS_9 | GPIO_PINS_8; /* GPIO pins: pin8 & pin9 */
  gpio_init_struct.gpio_pull = GPIO_PULL_NONE; /* GPIO pull configuration: no pull-up, no pull-down
*/
  gpio_init(GPIOB, &gpio_init_struct); /* Write the above configuration into the corresponding register
*/

  gpio_pin_mux_config(GPIOB, GPIO_PINS_SOURCE9, GPIO_MUX_9); /* Configure GPIOB_pin9
mux9 (CAN_TX) */
  gpio_pin_mux_config(GPIOB, GPIO_PINS_SOURCE8, GPIO_MUX_9); /* Configure GPIOB_pin8
mux9 (CAN_RX) */
```

6.4 Test result

- The LED2 on AT-START BOARD blinks, indicating a message in standard data format with ID=0x400 is received, and LED4 keeps blinking, indicating the program is working properly.

7 Revision history

Table 4 Document revision history

Date	Version	Revision note
2021.5.20	2.0.0	Initial release
2022.7.19	2.0.1	1. Modified the CAN baud rate calculation tool as the Artery_CAN_BitRate_Configuration.

IMPORTANT NOTICE – PLEASE READ CAREFULLY

Purchasers are solely responsible for the selection and use of ARTERY's products and services, and ARTERY assumes no liability whatsoever relating to the choice, selection or use of the ARTERY products and services described herein.

No license, express or implied, to any intellectual property rights is granted under this document. If any part of this document deals with any third party products or services, it shall not be deemed a license grant by ARTERY for the use of such third party products or services, or any intellectual property contained therein, or considered as a warranty regarding the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

Unless otherwise specified in ARTERY's terms and conditions of sale, ARTERY provides no warranties, express or implied, regarding the use and/or sale of ARTERY products, including but not limited to any implied warranties of merchantability, fitness for a particular purpose (and their equivalents under the laws of any jurisdiction), or infringement of any patent, copyright or other intellectual property right.

Purchasers hereby agrees that ARTERY's products are not designed or authorized for use in: (A) any application with special requirements of safety such as life support and active implantable device, or system with functional safety requirements; (B) any air craft application; (C) any automotive application or environment; (D) any space application or environment, and/or (E) any weapon application. Purchasers' unauthorized use of them in the aforementioned applications, even if with a written notice, is solely at purchasers' risk, and is solely responsible for meeting all legal and regulatory requirement in such use.

Resale of ARTERY products with provisions different from the statements and/or technical features stated in this document shall immediately void any warranty grant by ARTERY for ARTERY products or services described herein and shall not create or expand in any manner whatsoever, any liability of ARTERY.

© 2022 Artery Technology -All rights reserved