M16C/6N, M16C/1N, M16C/29, R8C/22,23 Groups

CAN Application Note

1. Summary

This document describes the procedure to be followed when performing CAN communication using the M16C/6N, M16C/1N, M16C/29, or R8C/22,23 group of microcomputers.

2. Application

This document applies to the M16C/6N, M16C/1N, M16C/29, or R8C/22,23 group of microcomputers (hereafter referred to respectively as 6N, 1N, 29, or R8C/22,23).
The 6N group is further divided into six subgroups: 6N4, 6N5, 6NK, 6NL, 6NM, and 6NN. The 6N5, 6NL, 6NN, 1N, 29, and R8C/22,23 have only CAN0, so that the descriptions relating to CAN1 do not apply to these groups.

3. Initial Settings

Before CAN communication can be performed, following settings must be made:

- CAN operation mode setting
- Baud rate setting
- Sampling count setting (Sampling counts are set to 1)
- Bit timing setting
- Acceptance filter setting

Furthermore, following programs must be created according to the system used:

- Program for finishing transmission by a CAN transmit-completed interrupt
- Program for finishing transmission by polling
- Program for finishing reception by a CAN receive-completed interrupt
- Program for finishing reception by polling
- Program for handling a CAN error interrupt
- Program for handling a CAN wakeup interrupt

3.1 CAN Bit Timing

In the CAN protocol, each bit in a communication frame is composed with four segments.
Figure 1 shows the segment composition of a bit and the sampling point in it.
Of these segments, the Propagation Time Segment (hereafter PTS), Phase Buffer Segment 1 (hereafter PBS1), and Phase Buffer Segment 2 (hereafter PBS2) are used to specify a sampling point, so that the timing with which each bit is sampled can be altered by changing the values of these segments.
The minimum unit in which this timing is set is referred to as Time Quantum (hereafter Tq), which is determined by the clock frequency supplied to the CAN module and the divide-by-N value of the baud rate prescaler.

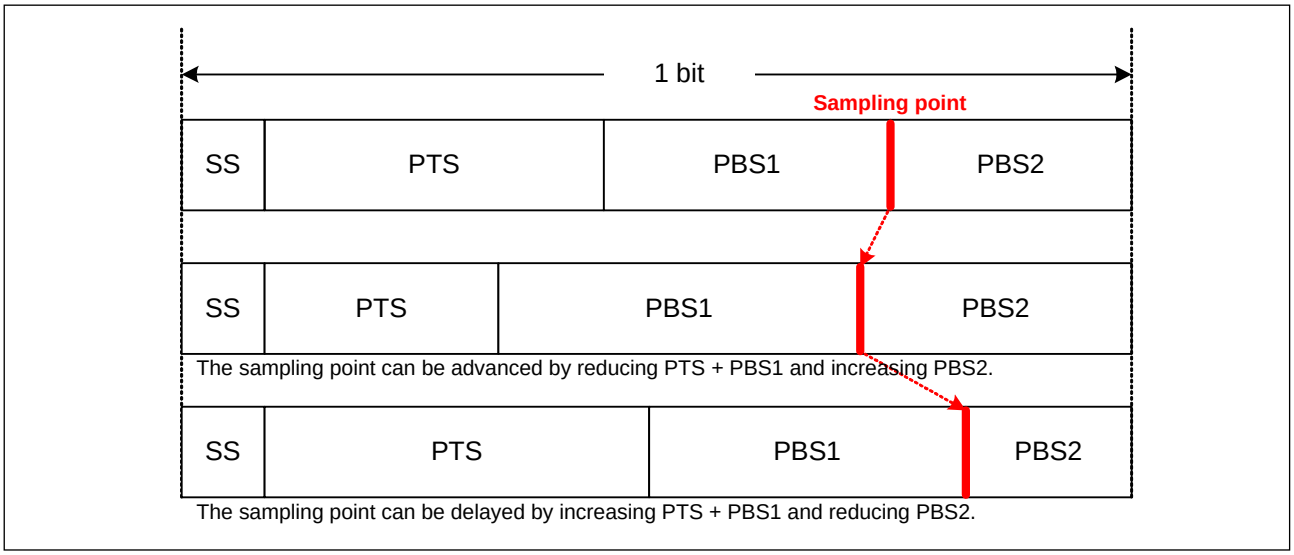


Figure 1 Segment composition of a bit and the sampling point

(1) SS: Synchronization Segment
This segment is used to synchronize bits by monitoring a recessive-to-dominant edge during the Interframe Space*1.

(2) PTS: Propagation Time Segment
This segment absorbs physical delays on the CAN network. A physical delay on the network is two times the sum total of a bus delay, input comparator delay, and output driver delay.

(3) PBS1, PBS2: Phase Buffer Segment 1, Phase Buffer Segment 2
These segments are used to correct a phase error*2 that occurs when bits are resynchronized.

(4) SJW: Resynchronization Jump Width
This is the maximum width by which bits gotten out of sync due to a phase error are corrected.

*1: Interframe Space
Comprised of Intermission, Suspend Transmission, and Bus Idle. During bus idle, all nodes can start transmission.
*2: Phase Error
Refers to out-of-sync caused by a drift in the oscillator frequency, etc. For details, refer to paragraph (2) in Section 3.2.

3.1.1 Bit Timing Conditions

The following describes how each segment is set and the limitations that apply to the segment setting.

(1) Each segment setting

- SS = Fixed to 1 Tq
- PTS = Set in the range 1 to 8 Tq's.
- PBS1 = Set in the range 2 to 8 Tq's.
- PBS2 = Set in the range 2 to 8 Tq's.
- SJW = Set in the range 1 to 4 Tq's.
- SS + PTS + PBS1 + PBS2 = 8 to 25 Tq's

(2) Limitations on PBS1 and PBS2

- PBS1 ≥ PBS2
- PBS1 ≥ SJW
- PBS2 ≥ 2 when SJW = 1
- PBS2 ≥ SJW when 2 ≤ SJW ≤ 4

3.2 How to Synchronize Bits

The communication method of CAN protocol is Non-Return to Zero (NRZ). No synchronizing signals are added to the beginning or end of each bit.

(1) Hardware synchronization (synchronization achieved when not sending or receiving messages)

When a recessive-to-dominant edge is detected in the interframe space, that point of time is recognized as the beginning of a bit (SS), based on which bits are synchronized. This is referred to as hardware synchronization.

Figure 2 shows the mechanism of hardware synchronization.

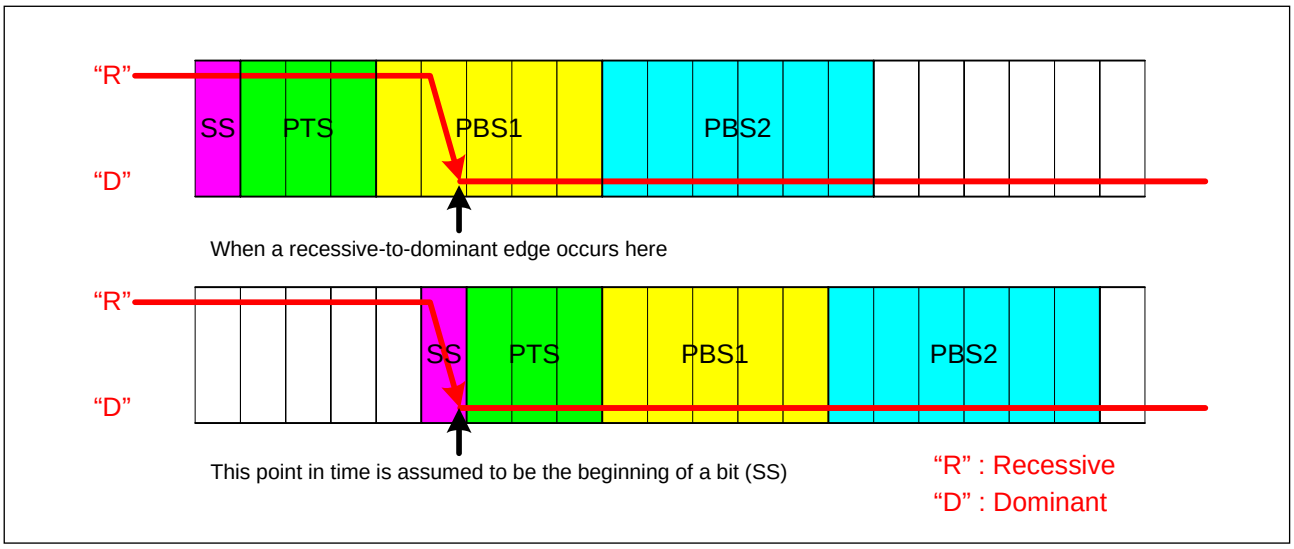


Figure 2 Mechanism of hardware synchronization

(2) Resynchronization (synchronization achieved when sending or receiving messages)

While sending or receiving a message, communication frames between some nodes may get out of sync due to a drift in the oscillator frequency or a delay in the transmission path. This is referred to as a phase error. If a bit gets out of sync, the length of the bit is dynamically corrected by adding the SJW value (or the SJW value if gotten out of sync more than the SJW value) to PBS1 or subtracting from PBS2. This is referred to as resynchronization.

Resynchronization is achieved by synchronizing bits with respect to only recessive-to-dominant edges as in the case of hardware synchronization.

Figure 3 and Figure 4 show the mechanism of resynchronization.

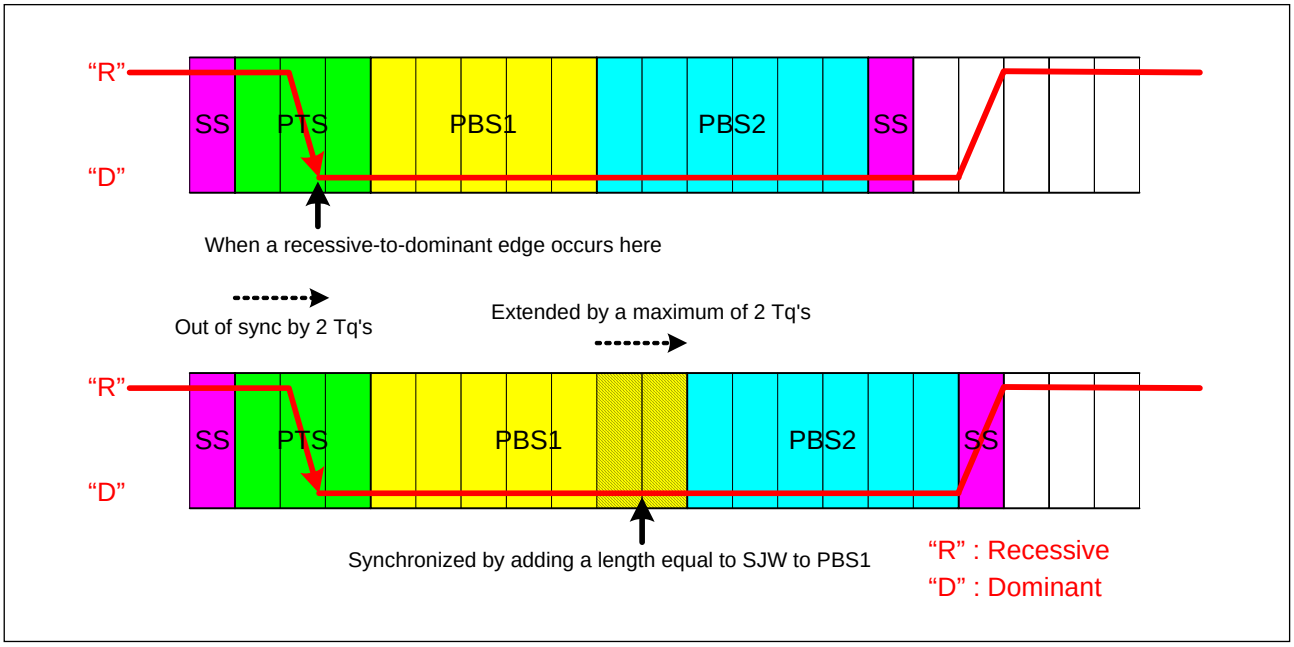


Figure 3 Resynchronization mechanism: when the R → D edge occurs in either the PTS or the PBS1 period (example for SJW = 2)

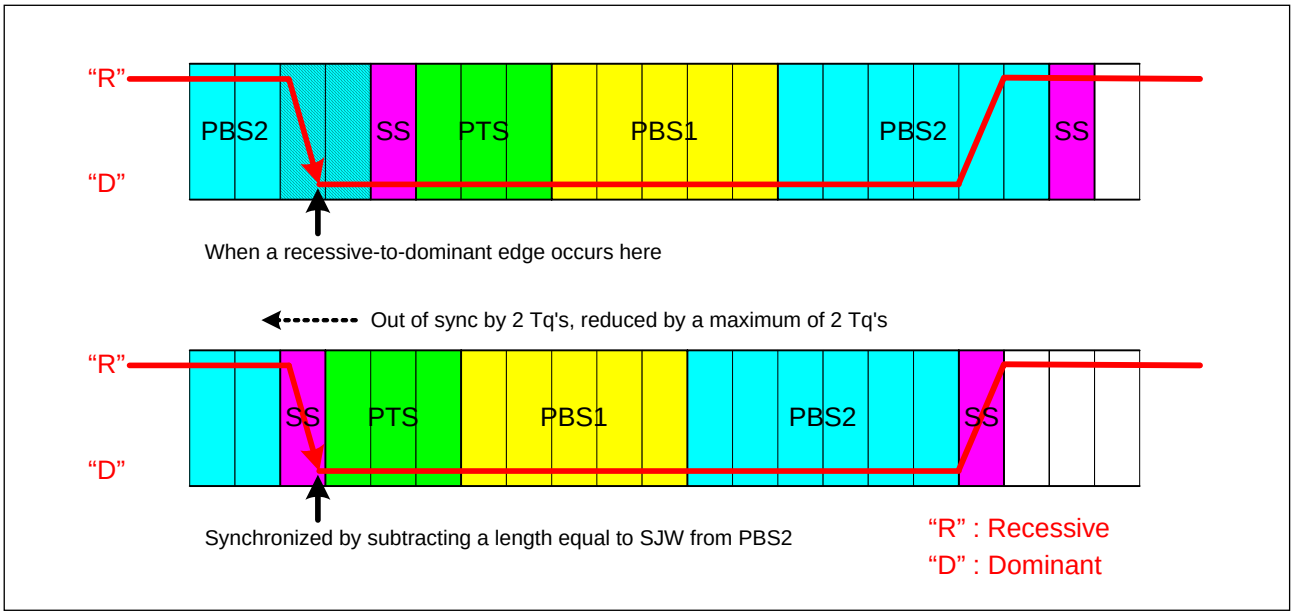


Figure 4 Resynchronization mechanism: when the R → D edge occurs in the PBS2 period (example for SJW = 2)

3.3 Baud Rate

The transfer speed is determined by f1, the divide-by-N value of the CAN module system clock, the divide-by-N value of the baud rate prescaler, and the number of Tq's in one bit.
Figure 5 shows a block diagram of the circuit that generates the CAN module system clock.
Table 1 shows a baud rate calculation formula and an example implementation. Table 2 shows an example of how to set the bit timing.

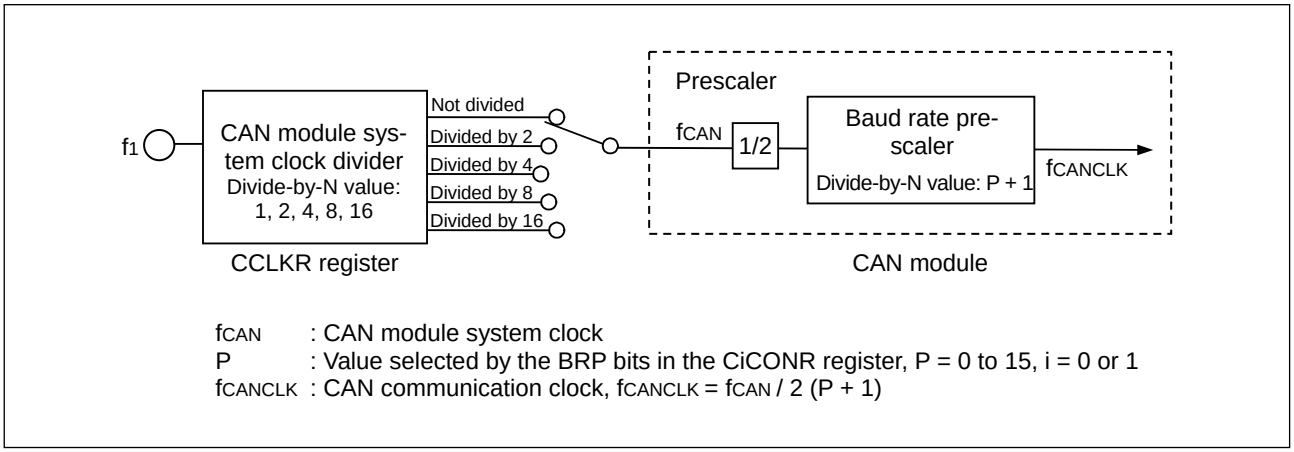


Figure 5 Block diagram of the circuit that generates the CAN module system clock

Table 1 Baud rate calculation formula and example implementation

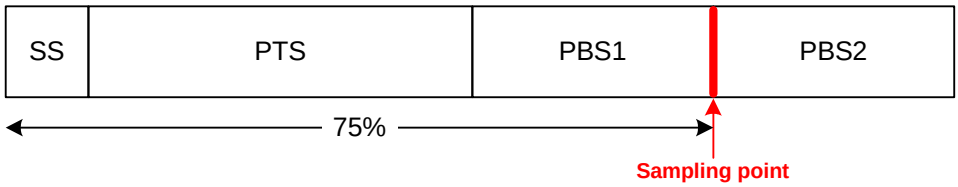
Baud rate calculation formula	$\frac{f_1}{2 \times f_{CAN}\text{ divide-by-N value}^{*1} \times \text{baud rate prescaler divide-by-N value}^{*2} \times \text{number of Tq's in one bit}}$				
Baud rate	24MHz*3	20MHz*4	16MHz	10MHz	8MHz
1Mbps	12Tq (1)	10Tq (1)	8Tq (1)	-	-
500kbps	12Tq (2) 24Tq (1)	10Tq (2) 20Tq (1)	8Tq (2) 16Tq (1)	10Tq (1) -	8Tq (1) -
125kbps	12Tq (8) 16Tq (6) 24Tq (4)	10Tq (8) 20Tq (4) -	8Tq (8) 16Tq (4) -	10Tq (4) 20Tq (2) -	8Tq (4) 16Tq (2) -
83.3kbps	12Tq (12) 16Tq (9) 24Tq (6)	10Tq (12) 20Tq (6) -	8Tq (12) 16Tq (6) -	10Tq (6) 20Tq (3) -	8Tq (6) 16Tq (3) -
33.3kbps	10Tq (30) 20Tq (15)	10Tq (30) 20Tq (15)	8Tq (30) 16Tq (15)	10Tq (15) -	8Tq (15) -

*1: f_{CAN} divide-by-N value = 1, 2, 4, 8, 16
f_{CAN} divide-by-N value: one that is selected by the CCLKR register.
*2: Baud rate prescaler divide-by-N value = P + 1 (P = 0-15)
P: value selected by the BRP bits in the CiCONR register (i = 0 or 1).
*3: The 6NK, 6NL, 6NM, and 6NN groups apply
*4: The 1N group does not apply.
*5: Shown in ( ) are the f_{CAN} divide-by-N value X baud rate prescaler divide-by-N value

Table 2 Example settings of bit timing

1 bit	Set value (Tq)					Sample point*1(%)
	SS	PTS	PBS1	PBS2	SJW	
8Tq	1	1	3	3	1	62.50
	1	3	2	2	1	75.00
10Tq	1	3	3	3	1	70.00
	1	5	2	2	1	80.00
16Tq	1	5	5	5	1	68.75
	1	7	4	4	1	75.00
20Tq	1	7	6	6	1	70.00
	1	5	7	7	1	65.00

*1: Sampling point (for the case where sampling point=75%)



3.4 Setting CAN Bit Timing and Baud rate

Figure 6 shows the procedure for setting the CAN bit timing and the baud rate.
These settings must always be performed during CAN configuration.
For details about the CAN configuration procedure, refer to Section 4.1.

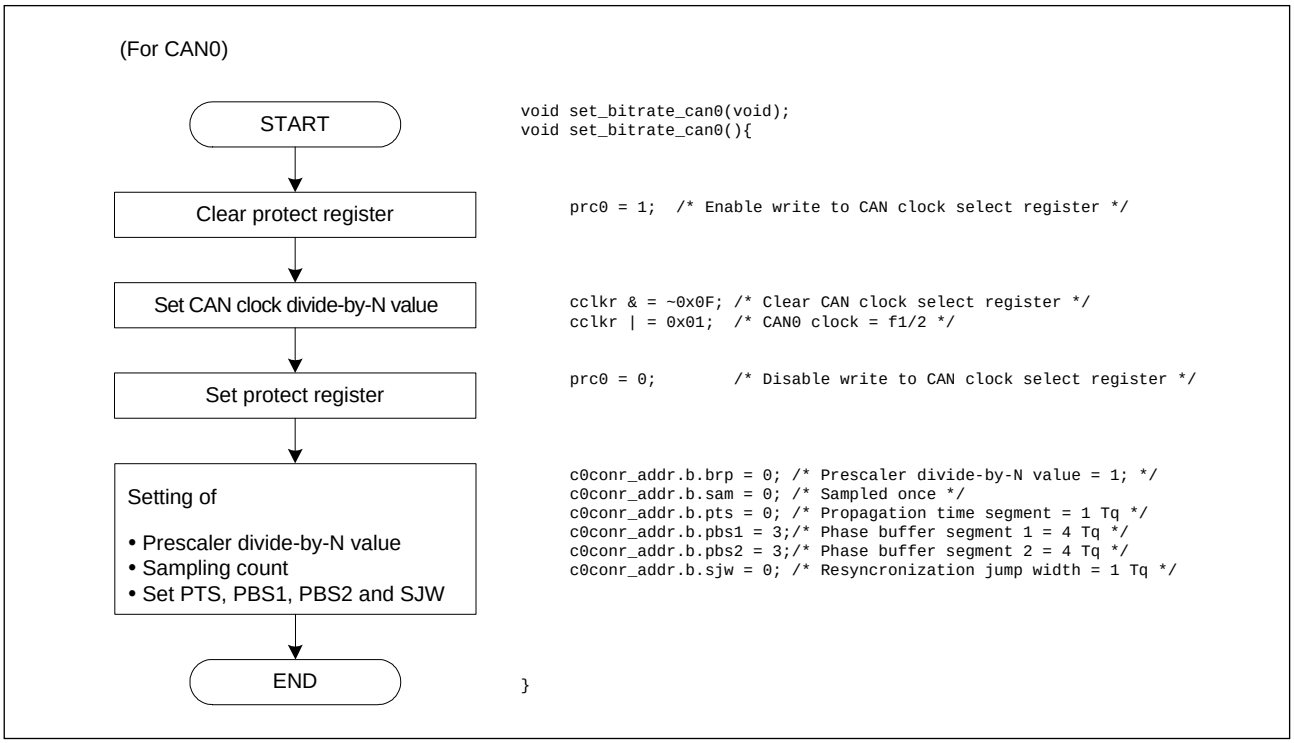


Figure 6 Procedure for setting the CAN bit timing and the baud rate

4. Transmitting/Receiving CAN Messages

CAN messages are transmitted / received following the three procedures described below.

(1) CAN configuration procedure

Set each relevant register during reset/initialization mode*1. These include the CAN control register, CAN bus timing control register, CAN clock select register, and mask register.

*1: The CAN message control register, CAN interrupt control register, and CAN extended ID register are automatically cleared when the CAN module goes from operating mode to reset/initialization mode. Note that these registers cannot be set during reset/initialization mode. Restore the CAN module to operating mode before setting these registers.

(2) Slot configuration procedure

To place any slot in transmit or receive mode, use the CAN message control register that is provided for each slot.

Table 3 shows the relationship between settings of the CAN message control register and the slot operation in transmit/receive modes.

(3) Data processing procedure

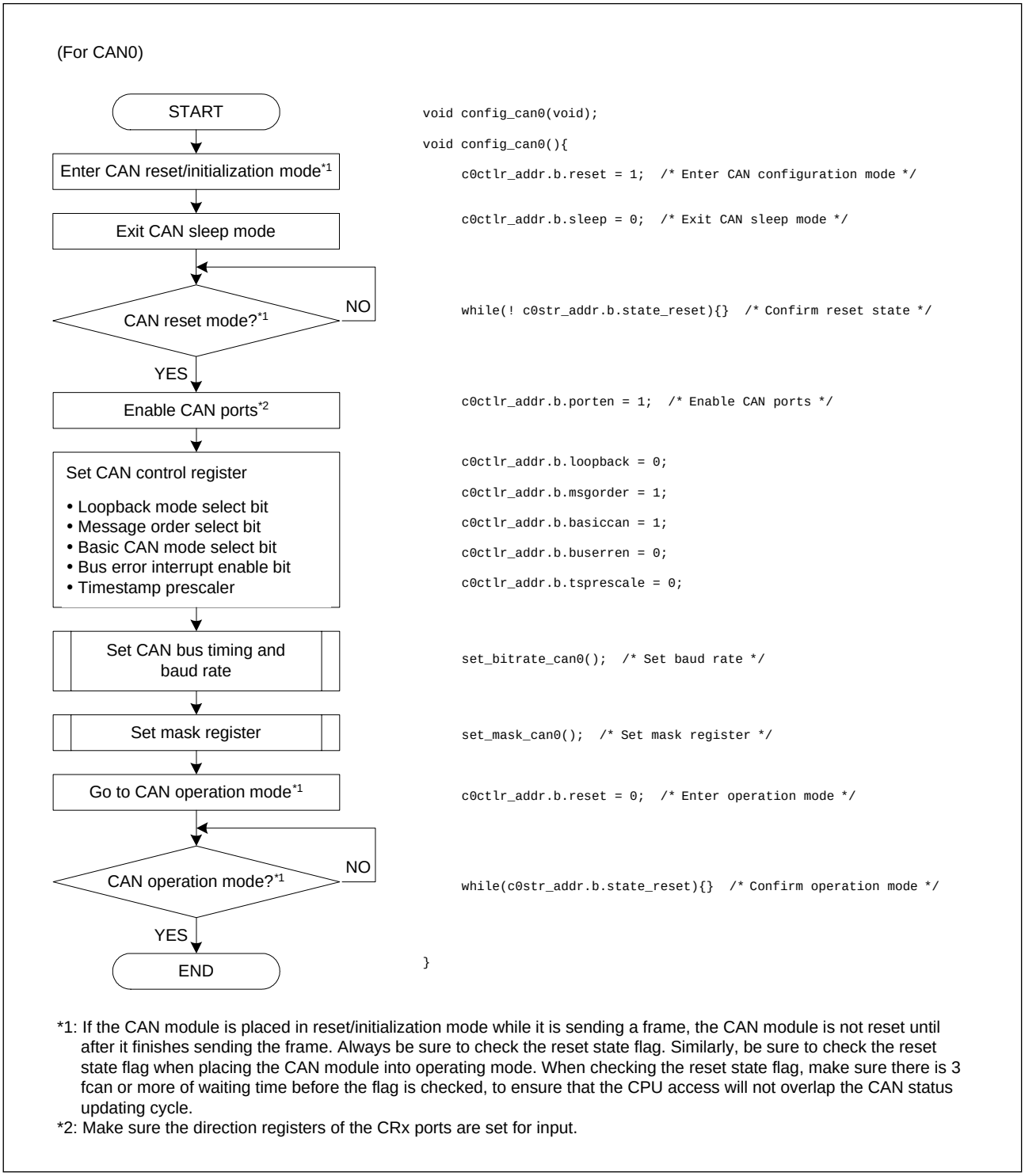
Perform message processing when the CAN module has finished sending or receiving a message normally.

Table 3 Relationship between CAN message control register settings and transmit/receive modes

C0MCTL/C1MCTL				Slot operation in transmit/receive modes
bit7 TrmReq	bit6 RecReq	bit5 Remote	bit4 RspLock	
0	0	-	-	Do not send or receive frames.
0	1	0	0	Receive a data frame (data frame receive mode)
1	0	1	0	Send a remote frame and when finished sending, receive a data frame. (Remote frame transmit/data frame receive mode)
1	0	0	0	Send a data frame (data frame transmit mode)
0	1	1	1/0*1	Receive a remote frame and when finished receiving, send a data frame. (Remote frame receive/data frame transmit mode) *1RspLock: 0: Send a data frame after receiving a remote frame as an automatic response. 1: Stand by without sending a data frame after receiving a remote frame. Clearing this bit causes the slot to send a data frame.

4.1 CAN Configuration

Figure 7 shows the CAN configuration procedure.



*1: If the CAN module is placed in reset/initialization mode while it is sending a frame, the CAN module is not reset until after it finishes sending the frame. Always be sure to check the reset state flag. Similarly, be sure to check the reset state flag when placing the CAN module into operating mode. When checking the reset state flag, make sure there is 3 fcan or more of waiting time before the flag is checked, to ensure that the CPU access will not overlap the CAN status updating cycle.
*2: Make sure the direction registers of the CRx ports are set for input.

Figure 7 CAN configuration procedure

4.2 Message Transmission

There are following two modes of message transmission:

- Data frame transmit mode
- Remote frame receive/data frame transmit mode

(1) Data frame transmit mode

If any slot is placed in data frame transmit mode, the data frame set in that slot can be transmitted.

(2) Remote frame receive/data frame transmit mode

If any slot is placed in remote frame receive/data frame transmit mode, the data set in that slot can be automatically transmitted after receiving a remote frame that has the same ID as set in the slot. In this case, the number of data bytes to be transmitted is determined by the DLC value in the received remote frame.

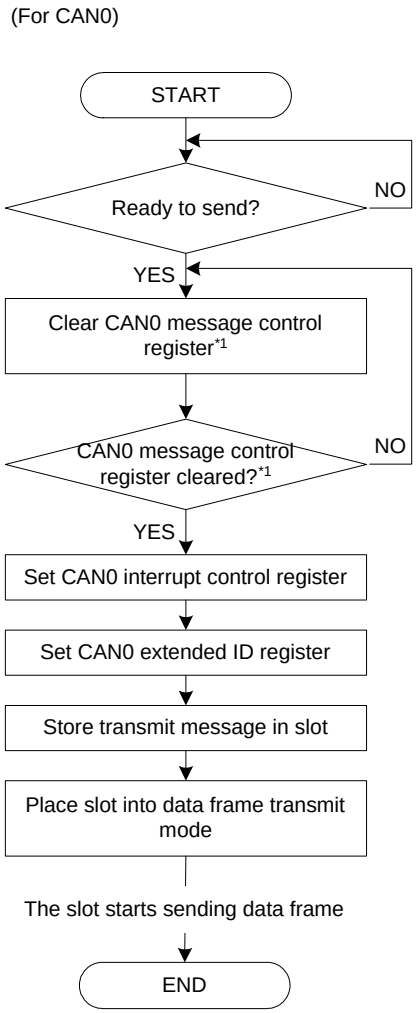
4.2.1 Data Frame Transmit Mode

(1) Transmit procedure

When any slot is placed in data frame transmit mode, the data frame set in that slot can be transmitted.

If two or more slots are placed in data frame transmit mode at the same time, the data frames in those slots are transmitted in order of slot numbers, beginning with the smallest.

Figure 8 shows the data frame transmit procedure.




```
typedef struct{
     unsigned short    id;         /* Standard ID */
     unsigned char     dlc;        /* Data length code */
     unsigned char     data[8];    /* Data */
}can_std_data_def;
void set_trm_std_dataframe_can0(unsigned short, can_std_data_def*);
void set_trm_std_dataframe_can0(
     unsigned short in_slot,
     can_std_data_def *in_trm_data)
{
     unsigned char lp_dlc;         /* Loop variable when storing data */
     while(c0mctl[in_slot].transmit.trmactive){}
                                   /* Check transmit ready state */



     while(c0mctl[in_slot].byte ! = 0x00){
                                   /* Check CAN0 message control register */
           c0mctl[in_slot].byte = 0x00;
                                   /* Clear CAN0 message control register */
}



     c0icr_addr.word &=  ~(0x0001<<in_slot);
                                   /* Set CAN0 interrupt control register */


     c0idr_addr.word &= ~(0x0001<<in_slot);
                                   /* Set CAN0 extended ID register  */
     c0slot[in_slot].ba.sidh = ((in_trm_data->id)>>6) & 0x1F;
                                   /* Set SID10-6 */
     c0slot[in_slot].ba.sidl = (in_trm_data->id) & 0x3F; /* Set SID5-0 */
     c0slot[in_slot].ba.dlc = in_trm_data->dlc;          /* Set DLC */
     for(lp_dlc=0; lp_dlc<(in_trm_data->dlc); ++lp_dlc){ /* Set data */
           c0slot[in_slot].ba.data[lp_dlc] = in_trm_data->data[lp_dlc];
     }
     c0mctl[in_slot].byte = 0x80;  /* Place slot in data frame transmit mode */



}
```

*1: The CAN0 Message Control Register must be cleared temporarily before any slot can be newly set for message transmission after it finished sending a data frame normally. Always be sure to follow the procedure shown here.

Figure 8 Data frame transmit procedure

(2) Procedure for confirming that transmission has terminated successfully

There are two methods to confirm that any slot has finished sending a message successfully. One method does this by polling, and the other by means of an interrupt.

- When confirming by polling

Whether a slot has finished sending a message successfully, can be confirmed by polling the CAN message control register.

Figure 9 shows the procedure for confirming by polling that a slot has finished sending a message successfully.

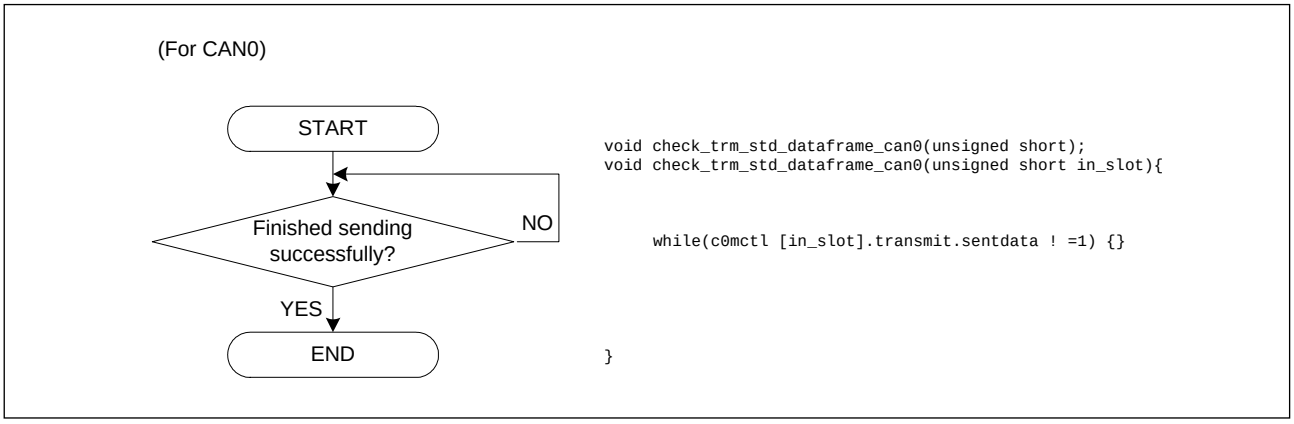
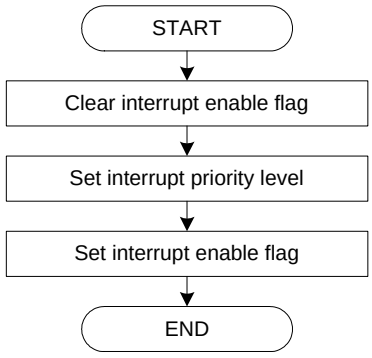


Figure 9 Procedure for confirming that transmission terminated successfully (by polling)

- When confirming by a CAN successful transmission interrupt

To use a CAN successful transmission interrupt for confirmation, enable the CAN successful transmission interrupt control register and then set the corresponding bit for each slot in the CAN interrupt control register to 1. This CAN interrupt control register is used in common for both CAN successful transmission and CAN successful reception interrupts. The CAN interrupt control register is automatically cleared when the CAN module goes to reset/initialization mode. Note that this register cannot be set during reset/initialization mode, and can only be set while the CAN module is in operating mode.
Figure 10 and Figure 11 show the procedure for setting the CAN successful transmission interrupt control register. Figure 12 shows the procedure for using a CAN successful transmission interrupt to confirm whether a slot has finished sending a message successfully.

(When setting the CAN0 successful transmission interrupt control register)

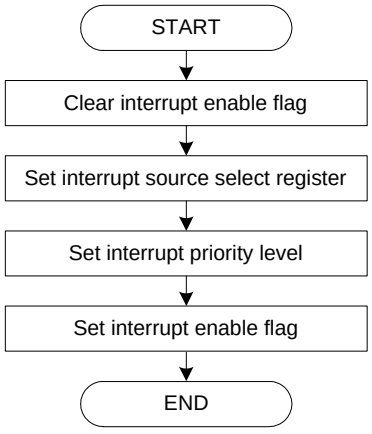



```
void enable_c0trmint(unsigned char);
void enable_c0trmint(unsigned char in_lvl){

     asm("FCLR I");  /* Disable interrupts */  *1

     c0trmic_addr.byte = in_lvl;
     asm("NOP");        *1/*2
     asm("NOP");        *1/*2

     asm("FSET I");    /* Enable interrupts */ *1

}
```

*1: asm() is an inline assemble description of the C compiler for the M16C family, NC30.
*2: The objective of two NOP instructions preceding one FSET instruction is to prevent the interrupt enable flag (I flag) from being set before a write to the interrupt control register for reasons of the instruction queue.

Figure 10 Procedure for setting the CAN0 successful transmission interrupt control register

(When setting the CAN1 successful transmission interrupt control register)

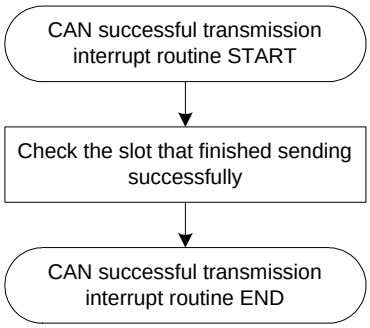



```
void enable_c1trmint(unsigned char);
void enable c1trmint(unsigned char in_lvl){

     asm("FCLR I");  /* Disable interrupts */  *1
     c1trmic_addr.byte = 0x00;
     ifsr16 = 0;     /* Select the source of interrupt */
     ifsr00 = 0;

     c1trmic_addr.byte = in_lvl;
     asm("NOP");        *1/*2
     asm("NOP");        *1/*2

     asm("FSET I");  /* Enable interrupts */   *1

}
```

*1: asm() is an inline assemble description of the C compiler for the M16C family, NC30.
*2: The objective of two NOP instructions preceding one FSET instruction is to prevent the interrupt enable flag (I flag) from being set before a write to the interrupt control register for reasons of the instruction queue.

Figure 11 Procedure for setting the CAN1 successful transmission interrupt control register

(For CAN0)




```
#pragma INTERRUPT fnc_c0trmic
void fnc_c0trmic(void);
void fnc_c0trmic(){

        if(c0mctl [0].transmit.sentdata = = 1){
                        /* Confirm that slot 0 finished sending successfully */
             /* Place here a user program after a successful transmission */
        }
        if(c0mctl [1].transmit.sentdata = = 1){
                        /* Confirm that slot 1 finished sending successfully */
             /* Place here a user program after a successful transmission */
        }
}
```

Figure 12 Procedure for confirming that a successful transmission (by a CAN successful transmission interrupt)

4.2.2 Remote Frame Receive/Data Frame Transmit Mode

When any slot is placed in remote frame receive/data frame transmit mode, the data set in that slot can be automatically transmitted after receiving a remote frame that has the same ID as set in the slot*1. In this case, the number of data bytes to be transmitted is determined by the DLC value in the received remote frame.

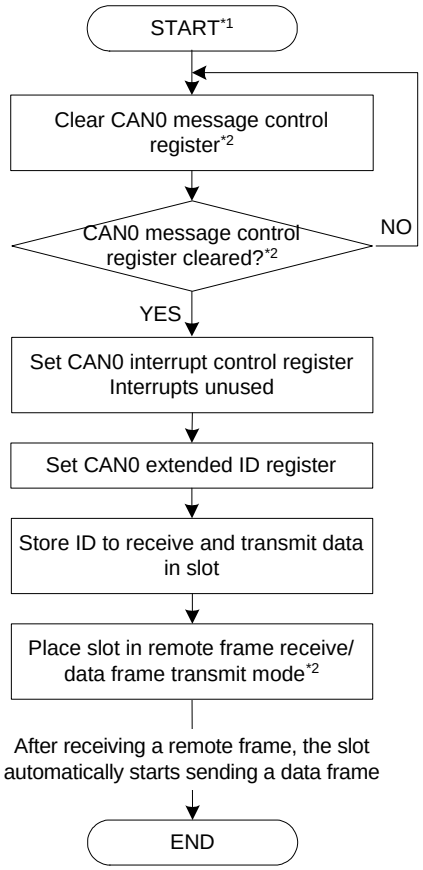
In remote frame receive/data frame transmit mode, it is possible to select whether or not to automatically respond to a received remote frame by setting the transmission/reception auto response lock mode select bit in the CAN message control register. (0: automatically respond; 1: not automatically respond.)

When selected to automatically respond, the slot automatically starts sending a data frame after receiving a remote frame. When selected not to automatically respond, set this bit to 0, and the slot will start sending a data frame after receiving a remote frame.

Figure 13 and Figure 14 show the remote frame receive/data frame transmit procedure.

*1: When operating in Basic CAN mode, slots 14 and 15 cannot be placed in remote frame receive/data frame transmit mode. For details, refer to Section 4.5.

(For CAN0)

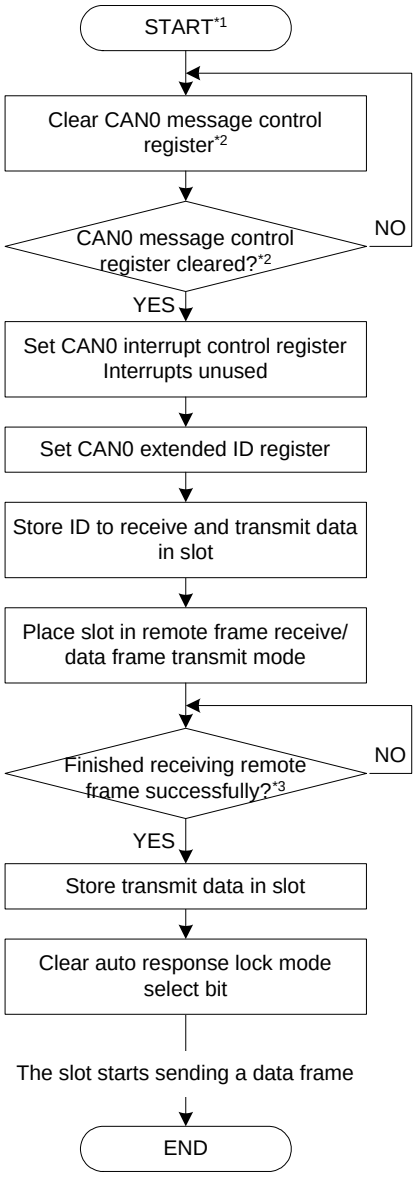



```
typedef struct{
     unsigned short id;            /* Standard ID */
     unsigned char data[8];        /* Data */
}can_rec_std_remote_def;

void set_rec_std_remoteframe_can0(unsigned short, can_rec_std_remote_def*);
void set_rec_std_remoteframe_can0
            (unsigned short in_slot, can_rec_std_remote_def *in_rec_remote)
{

     unsigned char lp_data;
     while(c0mctl[in_slot].byte ! = 0x00){
                        /* Check CAN0 message control register */
           c0mctl[in_slot].byte = 0x00;
                        /* Clear CAN0 message control register */
     }









     c0icr_addr.word &= ~(0x0001<<in_slot);
                         /* Set CAN0 interrupt control register */


     c0idr_addr.word &= ~(0x0001<<in_slot);
                         /* Set CAN0 extended ID register */

     c0slot[in_slot].ba.sidh = ((in_rec_remote->id)>>6) & 0x1F;
                        /* Set SID10-6 */
     c0slot[in_slot].ba.sidl = (in_rec_remote->id) & 0x3F;
                        /* Set SID5-0 */
     for(lp_data=0; lp_data<8; ++lp_data){     /* Set data */
           c0slot[in_slot].ba.data[lp_data] =
                                    in_rec_remote->data[lp_data];
     }
     asm("FCLR I");     *3/*4
     c0mctl [in_slot].byte = 0x70; *5
     c0mctl [in_slot].byte = 0x60
                        /* Set CAN0 message control register */
     asm("FSET I");     *3/*4



}
```

*1: Before making these transmission settings, confirm that there are no transmit or receive requests flagged for the slot.
*2: The CAN0 message control register must be cleared temporarily before any slot can be newly set for message transmission after it finished sending a data frame successfully. Always be sure to follow the procedure shown here.
*3: Make sure that no interrupts will be generated here.
*4: asm() is an inline assemble description of the C compiler for the M16C family, NC30.
*5: Due to optimization by the compiler, processing for the assignment of 70_{16} may be eliminated. After compiling, confirm that 70_{16} assignment processing has been executed.

Figure 13 Remote frame receive/data frame transmit procedure (for automatically responding to a received remote frame)

(For CAN0)




```
typedef struct{
     unsigned short id;            /* Standard ID */
     unsigned cha data[8];         /* Data */
}can_rec_std_remote_def;
void set_rec_std_remoteframe_can0(unsigned short, can_rec_std_remote_def*);
void set_rec_std_remoteframe_can0(
     unsigned short in_slot,
     can_rec_std_remote_def *in_rec_remote)
{

     unsigned char lp_data;
     while(c0mctl[in_slot].byte ! = 0x00){
                        /* Check CAN0 message control register */
       c0mctl[in_slot].byte = 0x00;
                        /* Clear CAN0 message control register */
     }



     c0icr_addr.word &= ~(0x0001<<in_slot);
                        /* Set CAN0 interrupt control register */


     0idr_addr.word &= ~(0x0001<<in_slot);
                        /* Set CAN0 extended ID register */

     c0slot[in_slot].ba.sidh = ((in_rec_remote->id)>>6) & 0x1F;
                        /* Set SID10-6 */
     c0slot[in_slot].ba.sidl = (in_rec_remote->id) & 0x3F;
                        /* Set SID5-0 */


     cc0mctl [in_slot].byte = 0x70;/* Set CAN0 message control register */



     while(c0mctl[in_slot].receive.remactive){}
     /* Confirm that the slot finished receiving remote frame successfully */


     /* Set data */
     for(lp_data=0; lp_data<c0slot[in_slot].ba.dlc; ++lp_data){
       c0slot[in_slot].ba.data[lp_data] = in_rec_remote->data[lp_data];
     }


     c0mctl[in_slot].receive.rsplock = 0;



}
```

*1: Before making these transmission settings, confirm that there are no transmit or receive requests flagged for the slot.
*2: The CAN0 message control register must be cleared temporarily before any slot can be newly set for message transmission after it finished sending a data frame successfully. Always be sure to follow the procedure shown here.
*3: A CAN successful reception interrupt may be used to confirm that the slot has finished receiving a remote frame.

Figure 14 Remote frame receive/data frame transmit procedure
(when not automatically responding to a received remote frame)

4.2.3 Abort Transmission

If two or more nodes started sending messages at the same time, the nodes whose messages have lower priority than the other may be lost in arbitration and abort the transmission. (They will re-send messages after the node whose message won the arbitration has finished sending.) Any nodes will not be able to finish sending a message successfully unless they win the arbitration, so that they will perpetually repeat a retransmission and cannot send a new message. To solve this problem, the CAN module has a function to cancel the message being retransmitted, which is referred to as the transmission abort function. This function may prove effective when it is desired to set a limit time for one message transmitted or there is an urgent high priority message to be transmitted.
Figure 15 shows an example application of the transmission abort function.
Abort transmission is executed by clearing the CAN message control register.

(1) Conditions under which a transmission abort request is effective

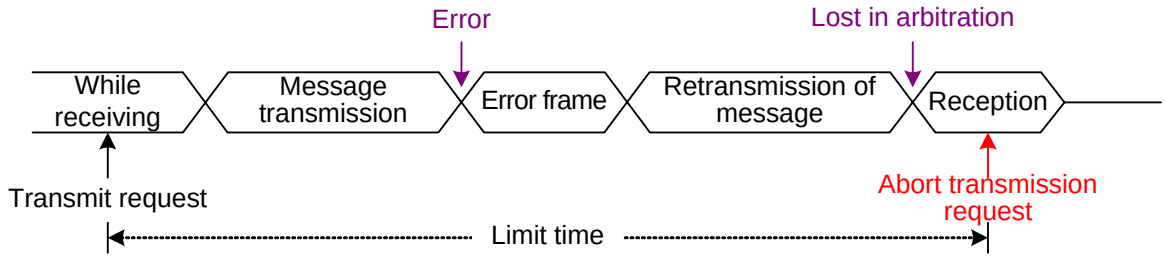
- When the node has been sending a data frame or a remote frame
  Abort transmission is effective only when the transmit slot setting bit in the CAN message control register = 1.
- When the node has been sending a data frame after receiving a remote frame
  Abort transmission is effective only when the receive slot setting bit in the CAN message control register = 1.

(2) Conditions under which abort transmission is executed

- When the message lost in arbitration (see Figure 16 and Figure 17).
- When an error occurred during message transmission (see Figure 16 and Figure 17).

(1) When it is desired to set a limit time for message transmission

It is possible to set a limit time for message transmission by requesting abort transmission a specified time after the node has started sending a message.



(2) When it is desired to send a high priority message

If abort transmission is executed while the node is sending a message, the message being transmitted is discarded when it loses in arbitration or an error in it is detected, allowing the node to send a high priority message.

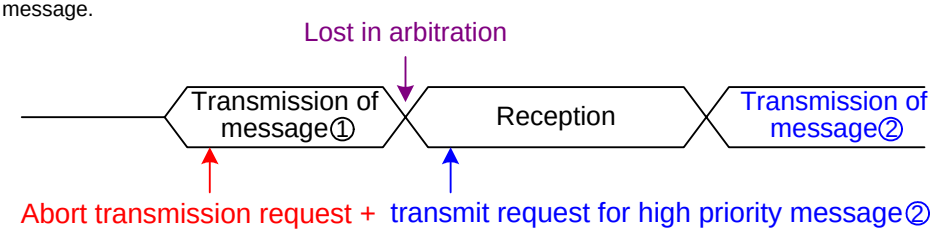
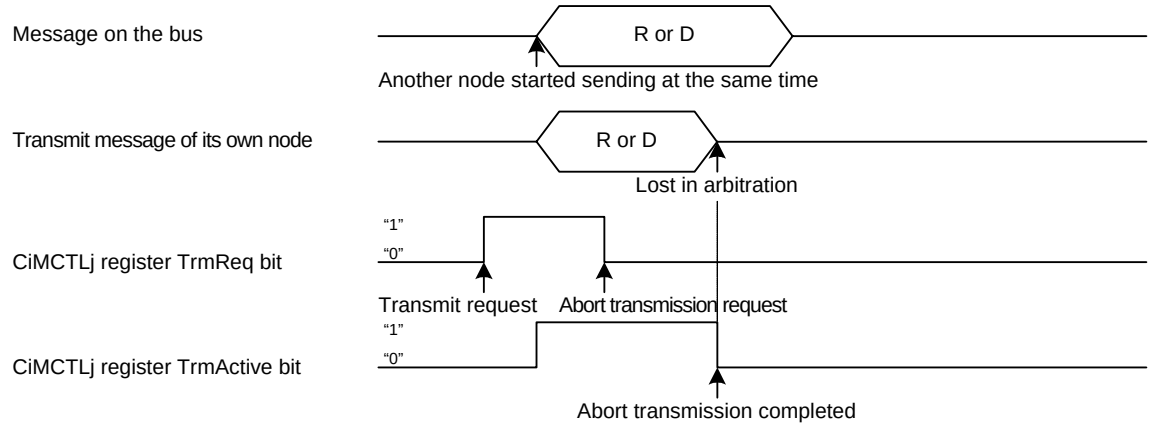
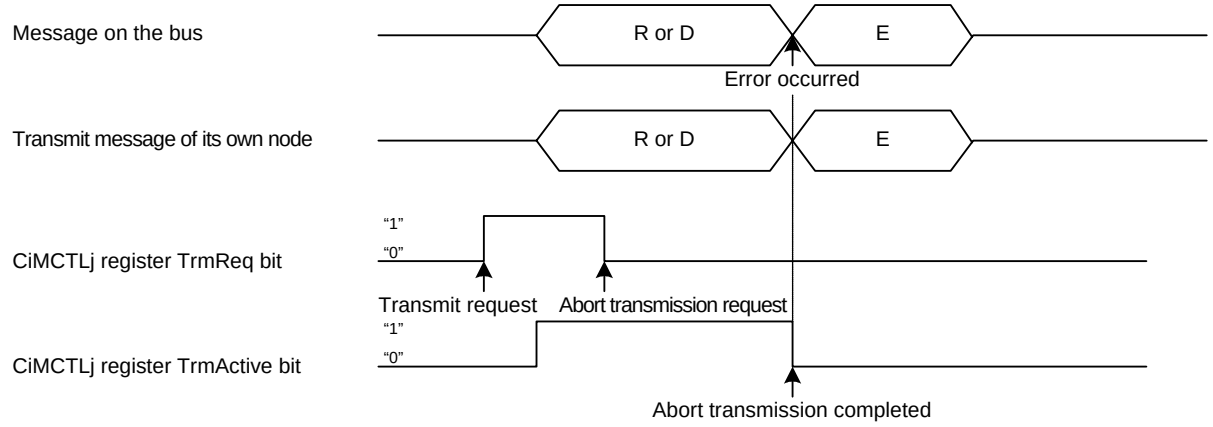


Figure 15 Example application of the transmission abort function

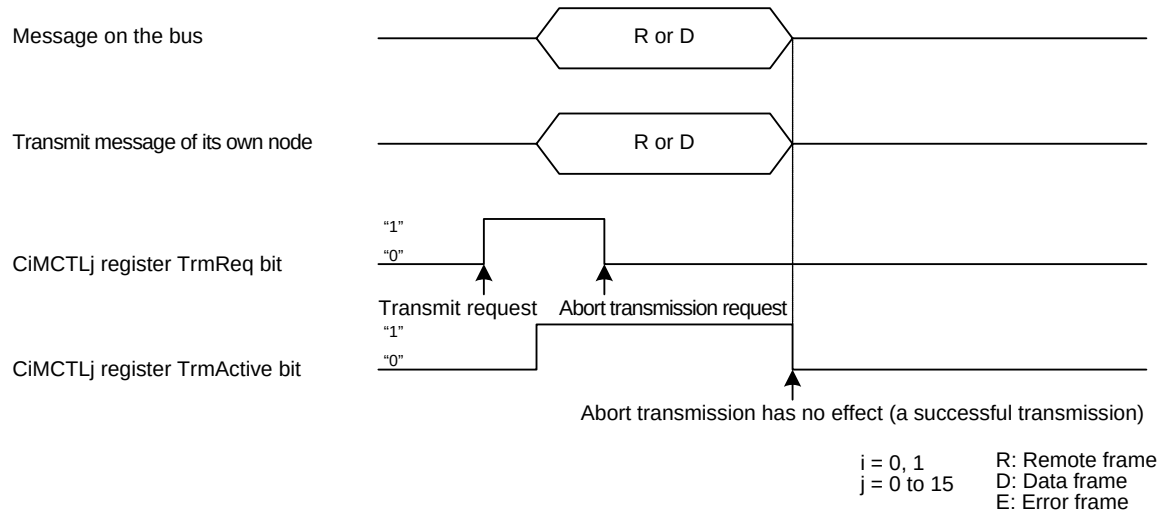
• When lost in arbitration



• When an error occurs



• When abortion of transmission has no effect (In case of a successful transmit)



i = 0, 1
j = 0 to 15

R: Remote frame
D: Data frame
E: Error frame

Figure 16 Abort transmission while sending a remote frame or data frame

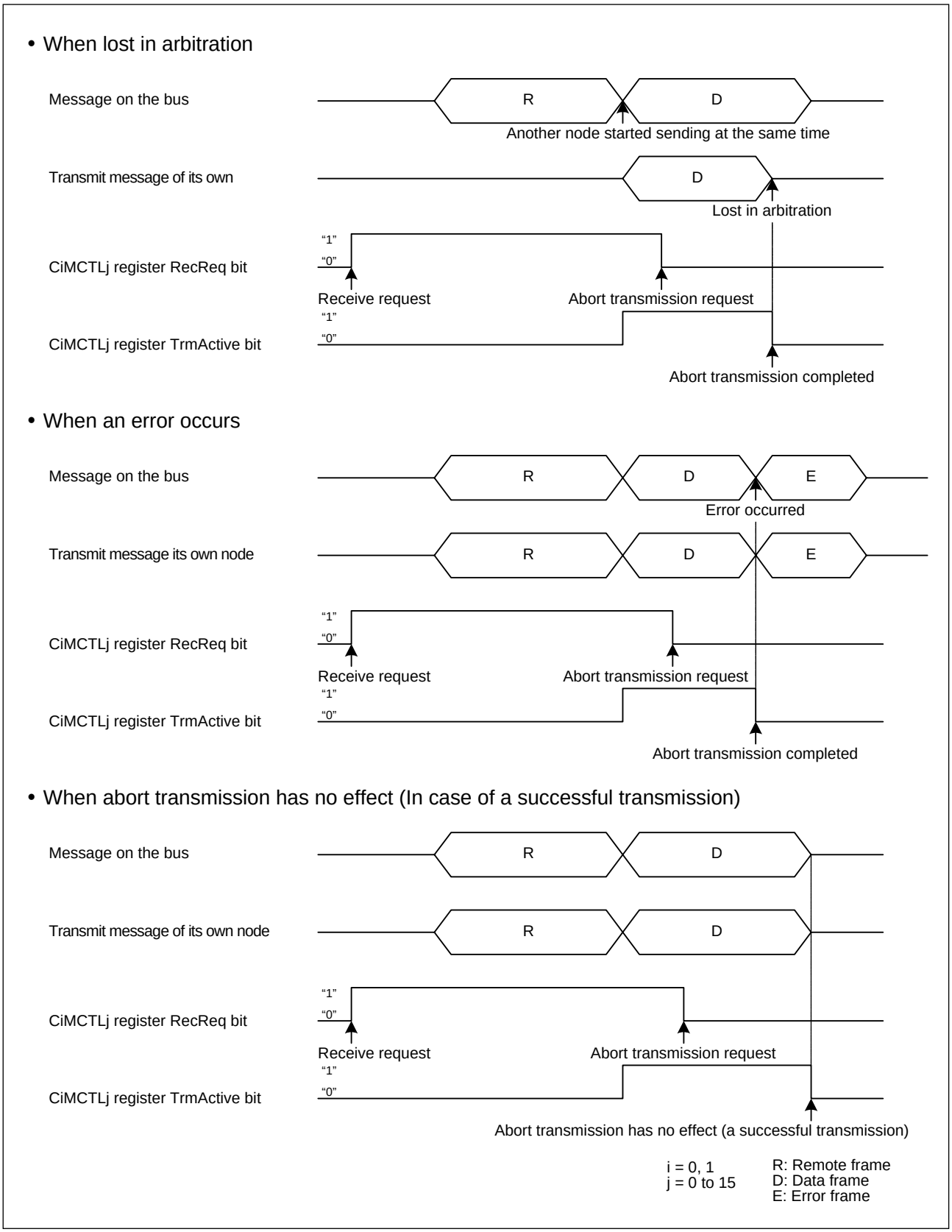
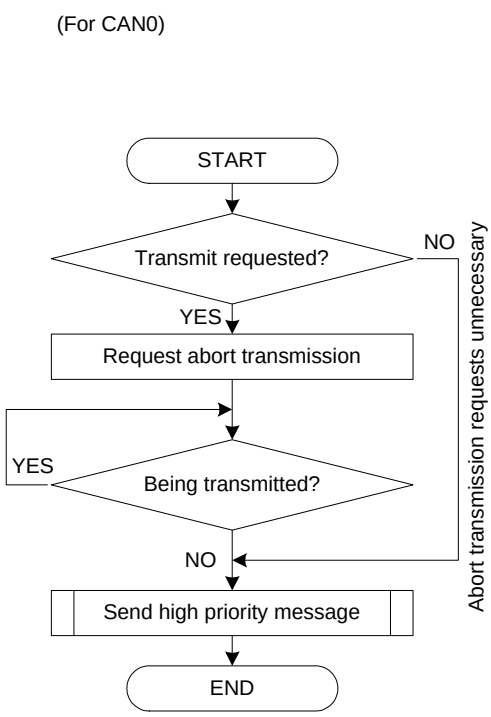


Figure 17 Abort transmission while sending a data frame after receiving a remote frame

(3) Abort transmission procedure

Figure 18 shows the procedure for abort transmission.




```
typedef struct{
     unsigned short     id;                    /* Standard ID */
     unsigned char      dlc;                   /* Data length code */
     unsigned char      data[8];    /* Data */
}can_std_data_def;
void abort_trm_can0(unsigned short, can_std_data_def*);
void abort_trm_can0(
     unsigned short in_slot,  /* Slot from which to send */
     can_std_data_def *in_trm_data)
                        /* Message to be sent after abortion */
{

     if(c0mctl[ in_slot ].transmit.trmreq
            || c0mctl[in_slot].receive.recreq){



            c0mctl[in_slot].byte = 0;  /* Transmission abort request */




            while(c0mctl[in_slot].transmit.trmactive){}
                        /* Confirm completion of transmission */
     }


     set_trm_std_dataframe_can0(in_slot, in_trm_data);
                        /* Send a message */


}
```

Figure 18 Abort transmission procedure

4.3 Receiving Messages

The CAN module has 16 slots per channel, of which unless the acceptance filter is used, the received message is stored in the slot with the smallest slot number of those that have been set for reception. If the acceptance filter is used, it is possible to select the message to receive. For details about the acceptance filter, refer to Section 6.
There are following two modes of message reception.

- Data frame receive mode
- Remote frame transmit/data frame receive mode

(1) Data frame receive mode

If any slot is placed in data frame receive mode, the data frame that has the same ID as set in that slot can be received.

(2) Remote frame transmit/data frame receive mode

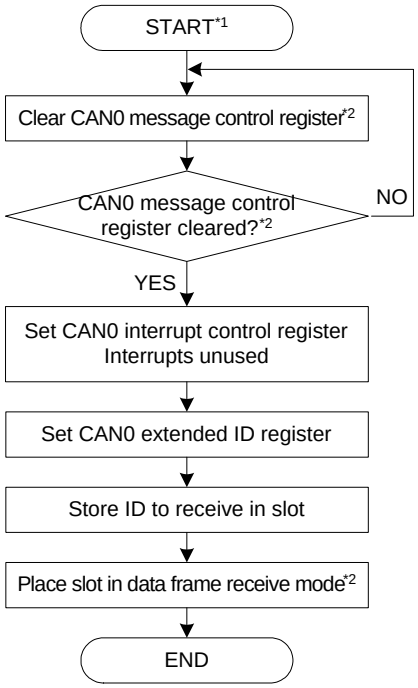
If any slot is placed in remote frame transmit/data frame receive mode, after a remote frame with its ID and DLC set in that slot is sent, the data frame that has the same ID as transmitted can be automatically received.

4.3.1 Data Frame Receive Mode

(1) Reception setup procedure

If the CAN module has been set to receive data frames with the same ID in two or more slots, the received message is always stored in the slot with the smallest slot number.

Figure 19 shows the procedure for setting up the CAN module to receive data frames.




```
void set_rec_std_dataframe_can0(unsigned short,
unsigned short);
void set_rec_std_dataframe_can0(
     unsigned short in_slot,
     unsigned short in_sid)
{


     while(c0mctl[in_slot].byte ! = 0x00){
                        /* Check CAN0 message control register */
           c0mctl[in_slot].byte = 0x00;
                        /* Clear CAN0 message control register */
     }









     c0icr_addr.word &= ~(0x0001<<in_slot);
                        /* Set CAN0 interrupt control register */


     c0idr_addr.word &= ~(0x0001<<in_slot);
                        /* Set CAN0 extended ID register */


     c0slot[in_slot].ba.sidh = (in_sid>>6) & 0x1F;/* Set SID10-6 */
     c0slot[in_slot].ba.sidl = in_sid & 0x3F;     /* Set SID5-0 */


     c0mctl[in_slot].byte = 0x40;
                        /* Place slot in data frame receive mode */


}
```

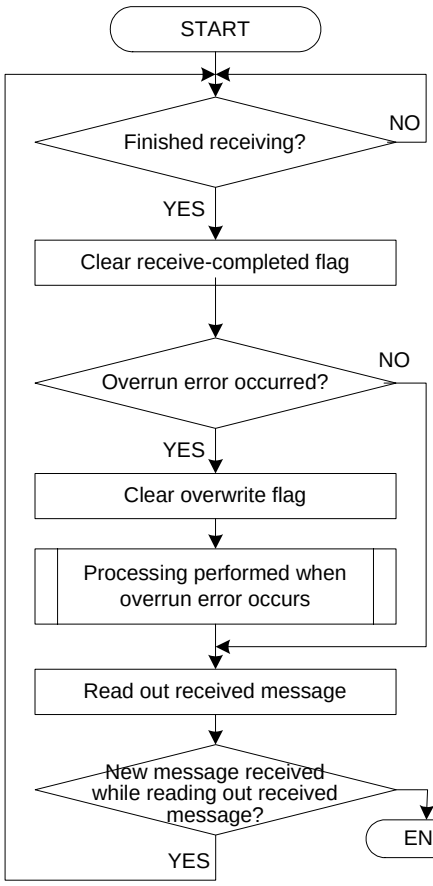
*1: Before making these receive settings, confirm that there are no transmit or receive requests flagged for the slot.
*2: The CAN0 message control register must be cleared temporarily before any slot can be newly set for message reception after it finished sending a data frame successfully. Always be sure to follow the procedure shown here.

Figure 19 Procedure for setting up to receive data frames

(2) Procedure for processing the received message

Figure 20 shows the procedure for processing the received message.

If any slot has finished receiving another message after it finished receiving successfully normally, the slot will be overwritten with the new message received. For details, refer to Section 4.4. Therefore, after the software has finished reading the received message out of the slot, it is necessary to confirm that the slot has not been overwritten during readout.




```
typedef struct{
     unsigned short    id;  /* Standard ID */
     unsigned char     dlc;  /* Data length code */
     unsigned char     data[8];  /* Data */
}can_std_data_def;
void get_message_can0(unsigned short, can_std_data_def *);
void get_message_can0(
     unsigned short in_slot,
     can_std_data_def *in_rec_data)

{
     unsigned char lp_dlc;

     while(1){
          /* Confirm completion of reception */
          while(c0mctl[in_slot].receive.invaldata){}



          c0mctl[in_slot].receive.newdata = 0;



          /* Check for overrun error */
          if(c0mctl[in_slot].receive.msglost){



            c0mctl[in_slot].receive.msglost = 0;



            lost_message_can0(in_slot);
                    /* Place a user program for overrun generation */
          }


          /* Read out message */
          in_rec_data->id = ((unsigned short)c0slot[in_slot].ba.sidh
                              << 6) + c0slot[in_slot].ba.sidl;
          in_rec_data->dlc = c0slot[in_slot].ba.dlc;
          for(lp_dlc=0; lp_dlc<c0slot[in_slot].ba.dlc; ++lp_dlc){
                    in_rec_data->data[lp_dlc] =
                              c0slot[in_slot].ba.data[lp_dlc];
          }
          /* Confirm that new messages have not been received */
          if(! c0mctl[in_slot].receive.newdata){
                    break;
          }
     }
}
```

Figure 20 Procedure for processing the received message

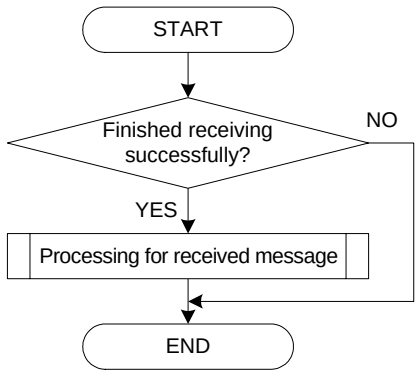
(3) Procedure for confirming that reception has terminated successfully

There are two methods to confirm that any slot has finished receiving a message successfully. One method does this by polling, and the other by means of an interrupt.

- When confirming by polling

Whether a slot has finished receiving a message successfully can be confirmed by polling the CAN message control register.

Figure 21 shows the procedure for confirming by polling that a slot has finished receiving a message successfully.



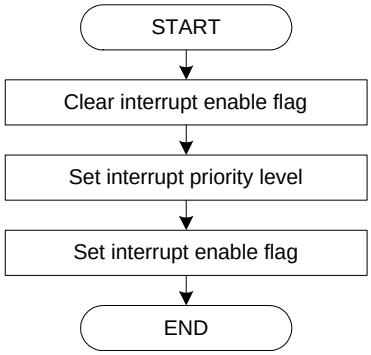

```
typedef struct{
     unsigned short  id;           /* Standard ID */
     unsigned char  dlc;           /* Data length code */
     unsigned char  data[8];       /* Data */
}can_std_data_def;
int check_rec_success_can0(unsigned short, can_std_data_def *);
int check_rec_success_can0(unsigned short in_slot, can_std_data_def
*in_rec_data)
{
     int result=0;

     /* Confirm that reception terminated normally */
     if(c0mctl [in_slot].receive.newdata = =1){


          /*  Read out the received message */
          get_message_can0(in_slot, in_rec_data);
          result = 1;
          }

     return result;
}
```

Figure 21 Procedure for confirming that reception successfully (by polling)

- When confirming by a CAN successful interrupt

To use a CAN successful interrupt for confirmation, enable the CAN successful interrupt control register and then set the corresponding bit for each slot in the CAN interrupt control register to 1. This CAN interrupt control register is shared by CAN successful transmission interrupt and CAN successful reception interrupt. The CAN interrupt control register is automatically cleared when the CAN module goes to reset/initialization mode. Note that this register cannot be set during reset/initialization mode, and can only be set while the CAN module is in operating mode. Figure 22 and Figure 23 show the procedure for setting the CAN successful reception interrupt control register. Figure 24 shows the procedure for using a CAN successful reception interrupt to confirm whether a slot has finished receiving a message successfully.

(When setting the CAN0 successful reception interrupt control register)

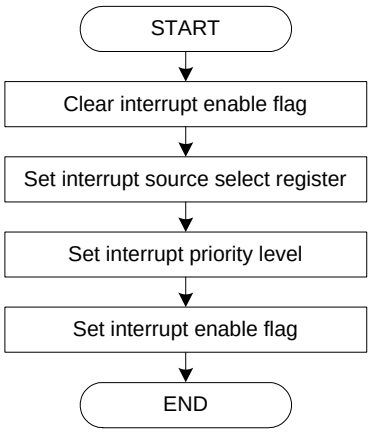



```
void enable_c0recint(unsigned char);
void enable_c0recint(unsigned char in_lvl){

     asm("FCLR I");  /* Disable interrupts */  *1

     c0recic_addr.byte = in_lvl;
     asm("NOP");  *1/*2
     asm("NOP");  *1/*2

     asm("FSET I");  /* Enable interrupts */  *1

}
```

*1: asm() is an inline assemble description of the C compiler for the M16C family, NC30.
*2: The objective of two NOP instructions preceding one FSET instruction is to prevent the interrupt enable flag (I flag) from being set before a write to the interrupt control register for reasons of the instruction queue.

Figure 22 Procedure for setting the CAN0 successful reception interrupt control register

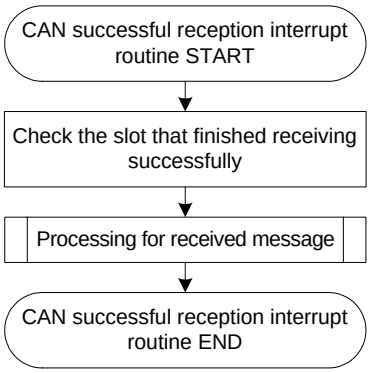
(When setting the CAN1 successful reception interrupt control register)




```
void enable_c1recint(unsigned char);
void enable_c1recint(unsigned char in_lvl){

     asm("FCLR I");  /* Disable interrupts */  *1

     ifsr17 = 0;     /* Select interrupt source */

     c1recic_addr.byte = in_lvl;
     asm("NOP");  *1/*2
     asm("NOP");  *1/*2

     asm("FSET I");  /* Enable interrupts */  *1

}
```

*1: asm() is an inline assemble description of the C compiler for the M16C family, NC30.
*2: The objective of two NOP instructions preceding one FSET instruction is to prevent the interrupt enable flag (I flag) from being set before a write to the interrupt control register for reasons of the instruction queue.

Figure 23 Procedure for setting the CAN1 successful reception interrupt control register

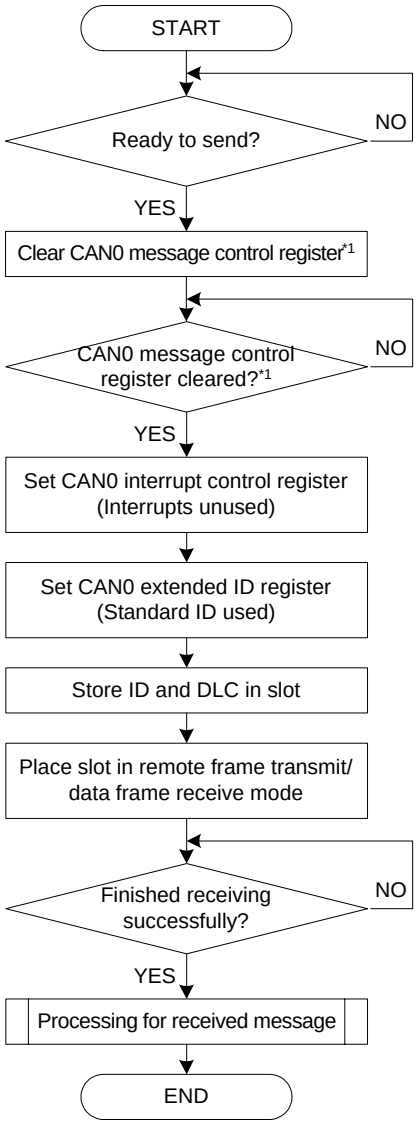



```
typedef struct{
     unsigned short  id;           /* Standard ID */
     unsigned char  dlc;           /* Data length code */
     unsigned char  data[8];       /* Data */
}can_std_data_def;
can_std_data_def rec_data;
#pragma INTERRUPT fnc_c0recic
void fnc_c0recic(void);
void fnc_c0recic(){
     if(c0mctl [14].receive.newdata ==1){
            get_message_can0(14, & rec_data);
                        /* Read received message out of slot 14 */
     }
     if(c0mctl [15].receive.newdata ==1){
            get_message_can0(15, & rec_data);
                        /* Read received message out of slot 15 */
     }

}
```

Figure 24 Procedure for confirming a successful transmission (by a CAN successful reception interrupt)

4.3.2 Remote Frame Transmit/Data Frame Receive Mode

If any slot is placed in remote frame transmit/data frame receive mode, after a remote frame with its ID and DLC set in that slot is sent, the data frame that has the same ID as transmitted can be automatically received. However, if a data frame that has the same ID as set in the slot is received, the remote frame is not transmitted.
Figure 25 shows the procedure for receiving a data frame after sending a remote frame, in which confirmation of reception is made by polling.




```
typedef struct{
     unsigned short id;/* Standard ID */
     unsigned char dlc;/* Data length code */
}can_trm_std_remote_def;
typedef struct{
     unsigned short id;            /* Standard ID */
     unsigned char dlc;            /* Data length code */
     unsigned char data[8];        /* Data */

}can_std_data_def;
void set_trm_std_remoteframe_can0
        (unsigned short, can_trm_std_remote_def*, can_std_data_def*);
void set_trm_std_remoteframe_can0
        (unsigned short in_slot, can_trm_std_remote_def *in trm_remote,
     can_std_data_def *in_rec_data);
{




     while(c0mctl[in_slot].transmit.trmactive){}
                        /* Check status to see if ready to send */
     while(c0mctl[in_slot].byte ! = 0x00){
                        /* Check CAN0 message control register */
           c0mctl[in_slot].byte = 0x00;
                        /* Clear CAN0 message control register */
     }



     c0icr_addr.word &= ~(0x0001<<in_slot);
                        /* Set CAN0 interrupt control register */



     c0idr_addr.word &= ~(0x0001<<in_slot);
                        /* Set CAN0 extended ID register */


     c0slot[in_slot].ba.sidh = ((in_trm_remote->id)>>6) & 0x1F;
                                                /* Set SID 10-6*/
     c0slot[in_slot].ba.sidl = (in trm_remote->id) & 0x3F; /* Set SID5-0*/
     c0slot[in_slot].ba.dlc = in trm_remote->dlc;          /* Set DLC */


     c0mctl[in_slot].byte = 0xa0;/* Set CAN0 message control register */




     while(check_rec_success_can0(in_slot)){}
                        /* Confirm that reception terminated successfully */



     get_message_can0(in_slot,& in_rec_data);
                        /* Read out received message */
     /* Place a user program for after a successful reception here */

}
```

*1: The CAN0 message control register must be cleared temporarily before any slot can be newly set for message reception after it finished sending a data frame successfully. Always be sure to follow the procedure shown here.

Figure 25 Procedure for receiving a data frame after sending a remote frame

4.4 CAN Overrun Error

If messages are received successively, the content of the slot is overwritten with the next message received.
If the successful reception flag is set to 0 by a program or the next CAN message is received successfully before the reception request for the slot is canceled, the overwrite flag is set to 1. In such a case, processing for the previous message to be retransmitted should be executed in programs on transmit and receive sides.
For details on how to process the received message, refer to Figure 20.

4.5 Basic CAN Mode

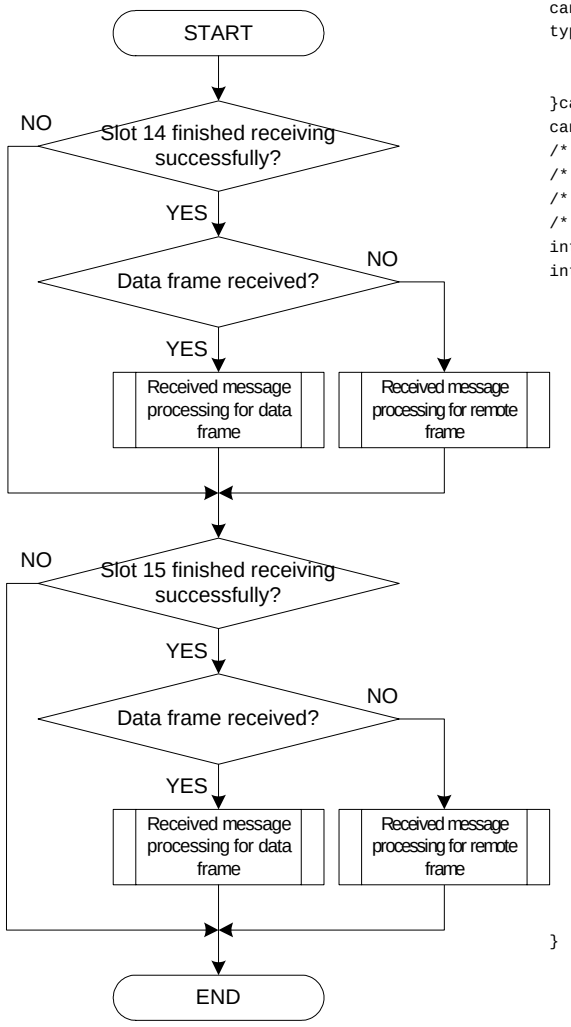
The CAN module has 16 slots per channel, each of which can be set for transmission or reception. This is referred to as normal operation mode.
In Basic CAN mode, slots 14 and 15 operate as receive slots, and slots 0 to 13 operate in normal operation mode. During this mode, the received messages are stored alternately in slots 14 and 15.
In normal operation mode, depending on how the CAN message control register is set, each slot can handle only one type of message at a time, either a data frame or a remote frame. In Basic CAN mode, however, slots 14 and 15 can receive both types of messages at the same time. Which type of message has been received can be determined by checking the remote frame transmission/reception status flag in the CAN message control register. The remote frame transmission/reception status flag is 0 when a data frame has been received and 1 when a remote frame has been received.
Basic CAN mode can be selected by setting the Basic CAN mode select bit in the CAN control register to 1 (= Basic CAN mode).

When using Basic CAN mode, observe the following precautions:

- Basic CAN mode can only be set during reset/initialization mode.
- Slots 14 and 15 must be set for reception.
- Slots 14 and 15 must have the same ID set in each. Also, the local mask register A and local mask register B for slots 14 and 15 must be set in the same way.
- Even during Basic CAN mode, if any slot receives a new message while it is receiving another message, the slot may be overwritten by the new message received.

Figure 26 shows the procedure for confirming that reception has terminated normally during Basic CAN mode.

(When CAN0 is set to Basic CAN mode)

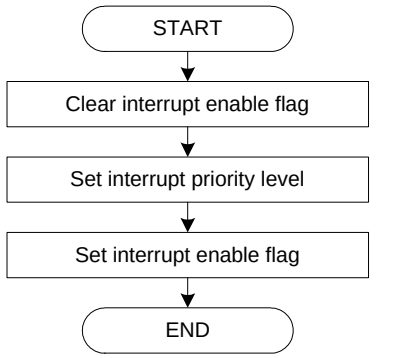



```
typedef struct{
     unsigned short id;            /* Standard ID */
     unsigned char dlc;            /* Data length code */
     unsigned char data[8];        /* Data */
}can_std_data_def;
can_std_data_def  rec_data[2];     /* Data frame reception */
typedef struct{
     unsigned short id;            /* Standard ID */
     unsigned char dlc;            /* Data length code */
}can_std_remote_def;
can_std_remote_def  rec_remote[2];
/* [0x0001]: received one dataframe */
/* [0x0002]: received two data frames */
/* [0x0010]: received one remoteframe */
/* [0x0020]: received two remoteframes */
int check_rec_success_basiccan_can0();
int check_rec_success_basiccan_can0(){
     char cnt_data = 0;   /* Counter for data frame reception */
     char cnt_remote = 0;/* Counter for remote frame reception */
     int result;
     /* Confirmation of mailbox 14 */
     if(c0mctl [14] receive.newdata = = 1){
            /* When data frame was received */
            if(! c0mctl[14].receive.remactive){
               get_dateframe_can0(14, &rec_data[cnt_data++]);
            }
            /* When remote frame was received */
            else{
               get_remoteframe_can0(14, &rec_remote[cnt_remote++]);
            }
     }
     /* Check slot 15 */
     if(c0mctl [15].receive.newdata = =1){
            /* When data frame was received */
            if(! c0mctl[15].receive.remactive){
               get_dataframe_can0(15, &rec_data[cnt_data++]);
            }
            /* When remote frame was received */
            else{
               get_remoteframe_can0(15,&rec_remote[cnt_remote++]);
            }
     }
     result = (cnt_remote<<4) + cnt_data;
     return result;
}
```

Figure 26 Procedure for confirming a successful receive in Basic CAN mode

5. CAN Errors

The CAN module has an interrupt facility for CAN errors, making it possible to confirm in this CAN error interrupt that a communication error has occurred.
If while a slot is sending or receiving the communication frame becomes erratic and an error is detected, the transmit error counter or receive error counter may increase depending on the transmit/receive status. When the transmit error counter or receive error counter exceeds 128, the CAN status changes from an error-active state to an error-passive state. When the transmit error counter exceeds 256, a bus-off state is entered.
If the bus error interrupt enable bit in the CAN control register is 1 (= bus error interrupt enabled), a CAN error interrupt is generated each time an error is detected.
If the bus error interrupt enable bit in the CAN control register is 0 (= bus error interrupt disabled), a CAN error interrupt is generated when an error-passive state or a bus-off state is entered.
Note that the bus error interrupt enable bit can only be set during CAN configuration. Before the CAN error interrupt can be used, the CAN error interrupt control register must be set.
Figure 27 shows the procedure for setting up the CAN error interrupt control register.




```
void enable_c01erric(unsigned char);
void enable_c01erric(unsigned char in_lvl){

     asm("FCLR I");  /* Disable interrupts */  *1

     c01erric_addr.byte = in_lvl;
     asm("NOP");        *1/*2
     asm("NOP");        *1/*2

     asm("FSET I");  /* Enable interrupts */   *1

}
```

*1: asm() is an inline assemble description of the C compiler for the M16C family, NC30.
*2: The objective of two NOP instructions preceding one FSET instruction is to prevent the interrupt enable flag (I flag) from being set before a write to the interrupt control register for reasons of the instruction queue.

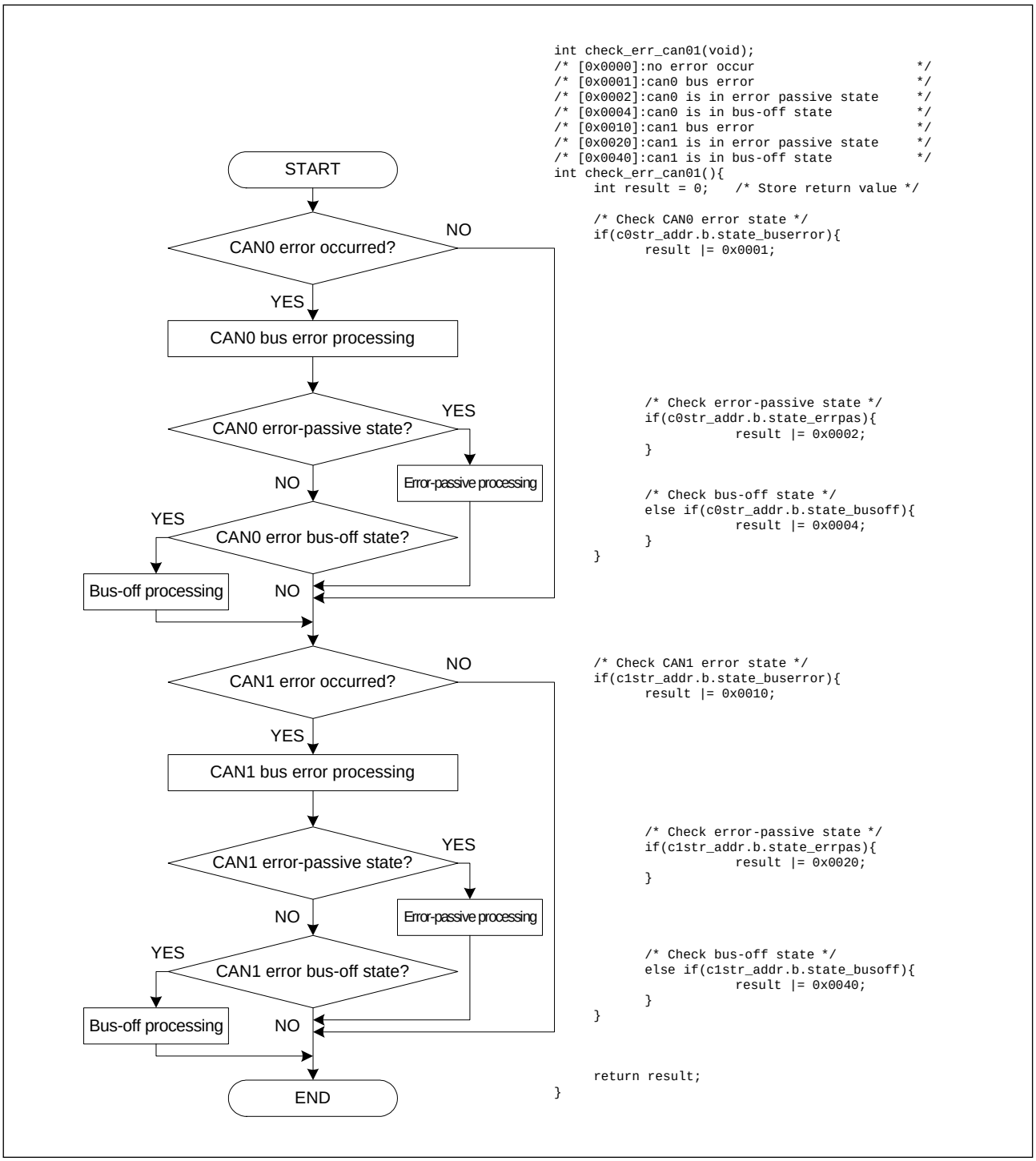
Figure 27 Procedure for setting up the CAN error interrupt control register

5.1 CAN Error Confirmation Procedure

There are two methods to confirm the CAN error state. One method does this by polling, and the other by means of an interrupt.

- Confirmation by polling

  The CAN error state can be confirmed by polling the bus-error state, error-passive state, and bus-off state flags in the CAN status register.
  Figure 28 shows the procedure for confirming the CAN error state by polling.




```
int check_err_can01(void);
/* [0x0000]:no error occur                      */
/* [0x0001]:can0 bus error                      */
/* [0x0002]:can0 is in error passive state      */
/* [0x0004]:can0 is in bus-off state            */
/* [0x0010]:can1 bus error                      */
/* [0x0020]:can1 is in error passive state      */
/* [0x0040]:can1 is in bus-off state            */
int check_err_can01(){
     int result = 0;    /* Store return value */

     /* Check CAN0 error state */
     if(c0str_addr.b.state_buserror){
          result |= 0x0001;

          /* Check error-passive state */
          if(c0str_addr.b.state_errpas){
                    result |= 0x0002;
          }

          /* Check bus-off state */
          else if(c0str_addr.b.state_busoff){
                    result |= 0x0004;
          }
     }

     /* Check CAN1 error state */
     if(c1str_addr.b.state_buserror){
          result |= 0x0010;

          /* Check error-passive state */
          if(c1str_addr.b.state_errpas){
                    result |= 0x0020;
          }

          /* Check bus-off state */
          else if(c1str_addr.b.state_busoff){
                    result |= 0x0040;
          }
     }

     return result;
}
```

Figure 28 Procedure for checking the CAN error state (by polling)

- Confirmation by means of a CAN error interrupt
  The CAN error interrupt becomes usable by setting up the CAN error interrupt control register to enable the interrupt. In this CAN error interrupt routine it is possible to check the CAN error state. This CAN error interrupt is shared by CAN0 and CAN1.
  Figure 29 shows the procedure for confirming the CAN error state by means of a CAN error interrupt.
  Here, the error state is checked in the CAN error interrupt routine and if the error state is confirmed to be a bus-off state, return from bus-off is executed.
  For details about return from bus-off, refer to Section 5.2.

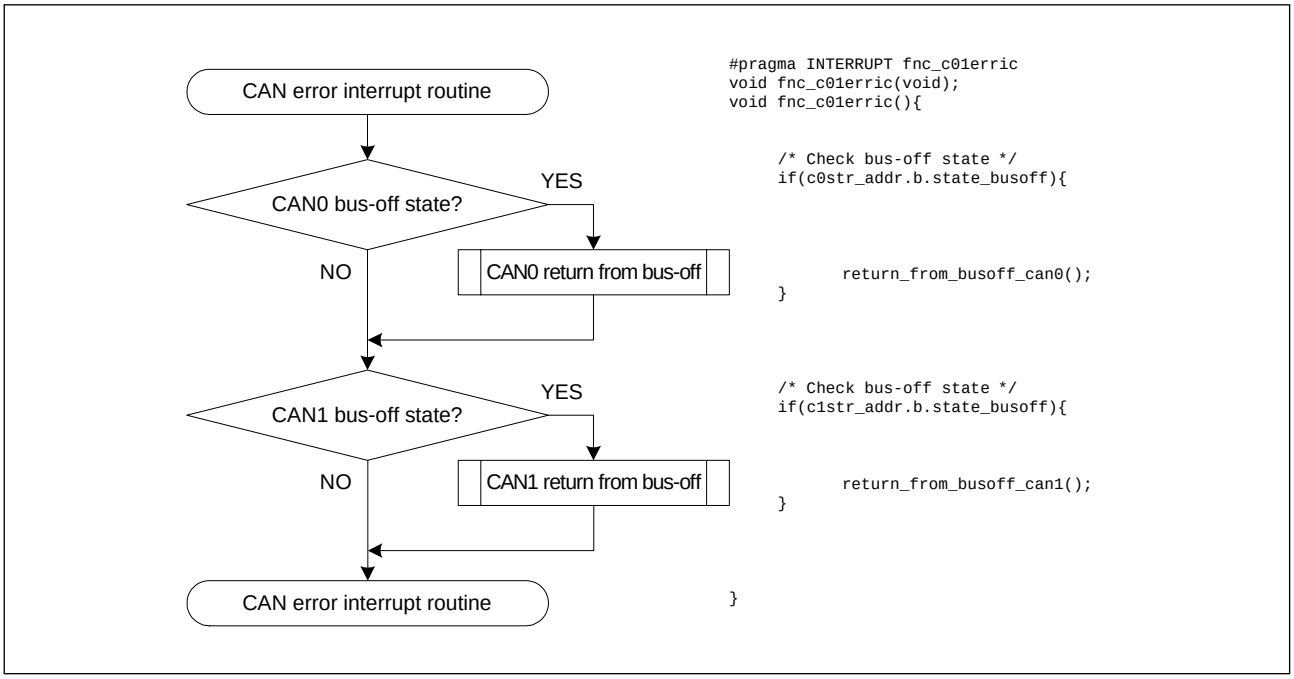
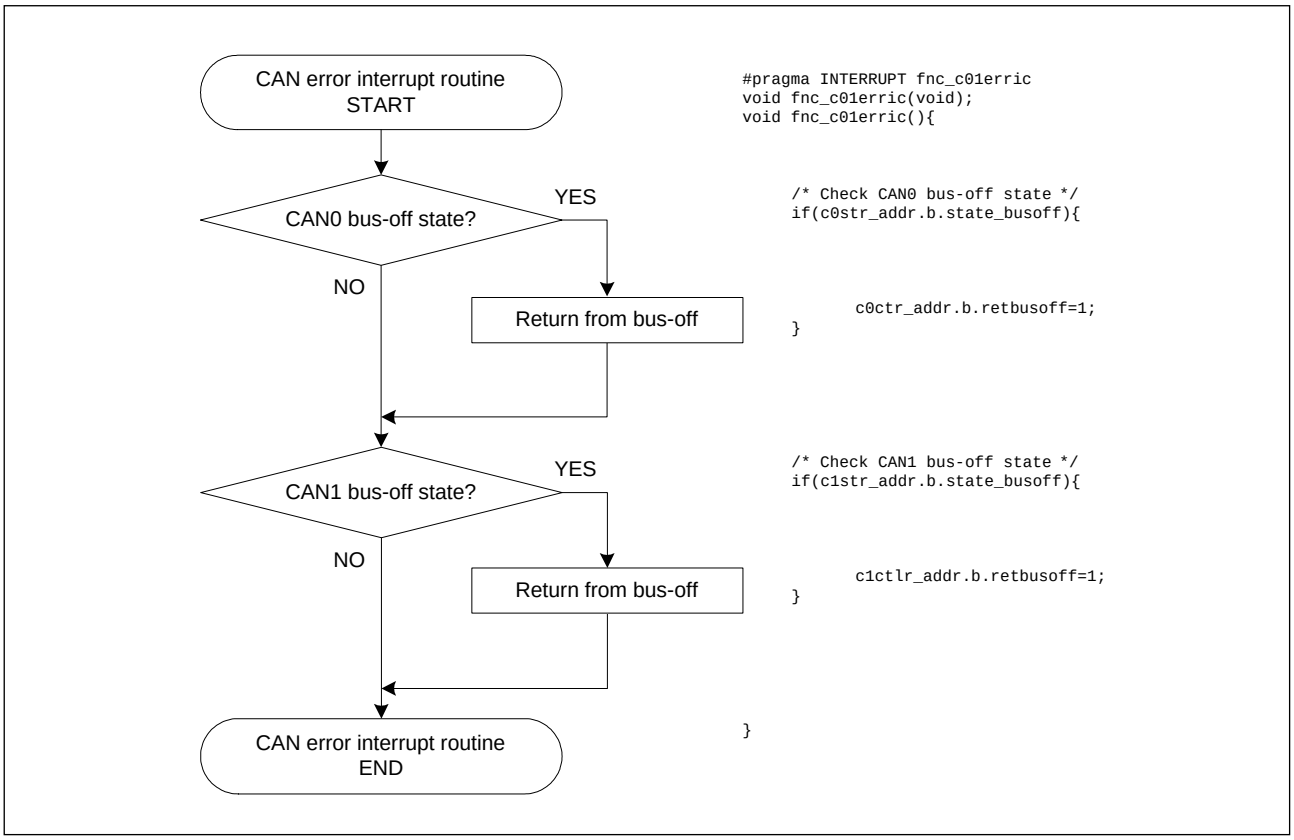


Figure 29 Procedure for confirming the CAN error state (by means of a CAN error interrupt)

5.2 Return from Bus-off Function

If the CAN status goes to a bus-off state, the CAN module becomes unable to transmit or receive. Before the CAN module can enter an error-active state where it can transmit and receive, it must wait until 11 consecutive recessive bits are detected on the bus as many as 128 times. However, if there is an urgent message to be sent, for example, the CAN module may need to be placed in an error-active state immediately. Therefore a function known as "Return from Bus-off" is provided that allows the CAN module to enter an error-active state without waiting for the above period.
The return from bus-off function is executed by setting the return from bus-off command bit in the CAN control register to 1 (= forcible return from bus-off). The return from bus-off command bit is automatically cleared after return from bus-off is executed. When return from bus-off is executed, note that CAN configuration does not need to be performed after an error-active state is entered.
The return from bus-off function is effective only when the CAN module is in a bus-off state. If an attempt is made to execute return from bus-off while the CAN module is in an error-active or an error-passive state, the attempt is ignored and the forcible return from bus-off Instruction bit is cleared immediately.
Figure 30 shows an example execution of the return from bus-off function.
Here, the CAN0 and CAN1 error states are checked in the CAN error interrupt routine and if the error state is confirmed to be a bus-off state, return from bus-off is executed.




```
#pragma INTERRUPT fnc_c01erric
void fnc_c01erric(void);
void fnc_c01erric(){

    /* Check CAN0 bus-off state */
    if(c0str_addr.b.state_busoff){

        c0ctr_addr.b.retbusoff=1;
    }

    /* Check CAN1 bus-off state */
    if(c1str_addr.b.state_busoff){

        c1ctlr_addr.b.retbusoff=1;
    }

}
```

Figure 30 Example execution of the return from bus-off function

6. Using the Acceptance Filter

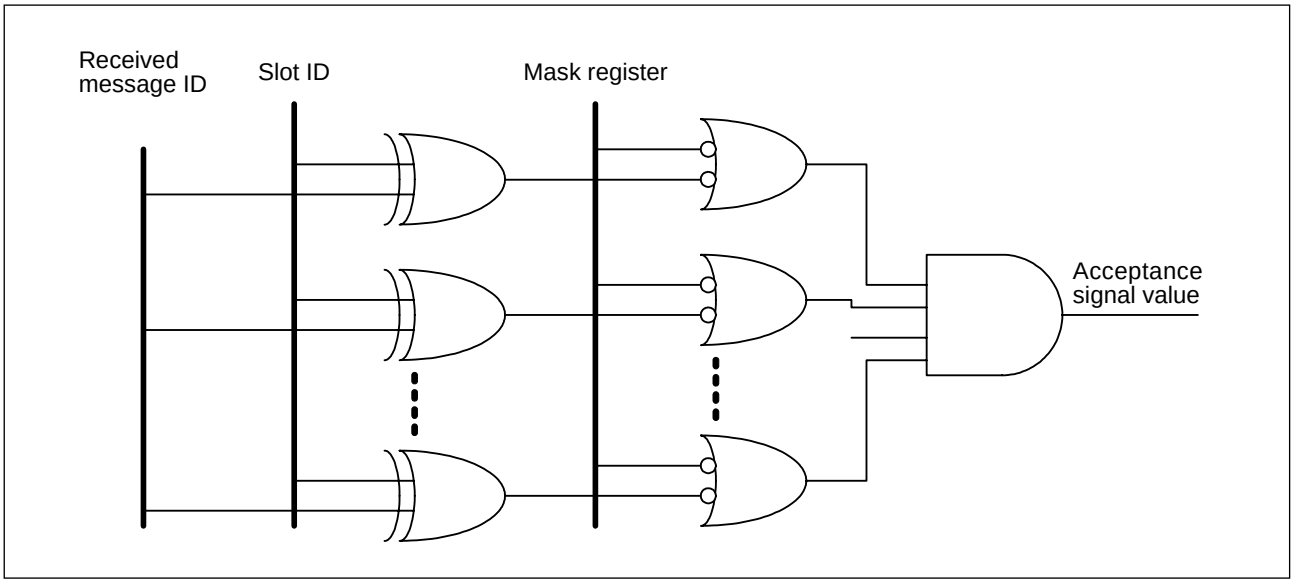
A function known as the acceptance filter is available that allows messages to be received or discarded in hardware.

6.1 Acceptance Filter (ACP)

The acceptance filter uses the global mask register (for slots 0 to 13), local mask register A (for slot 14), and local mask register B (for slot 15) for filtering of messages.

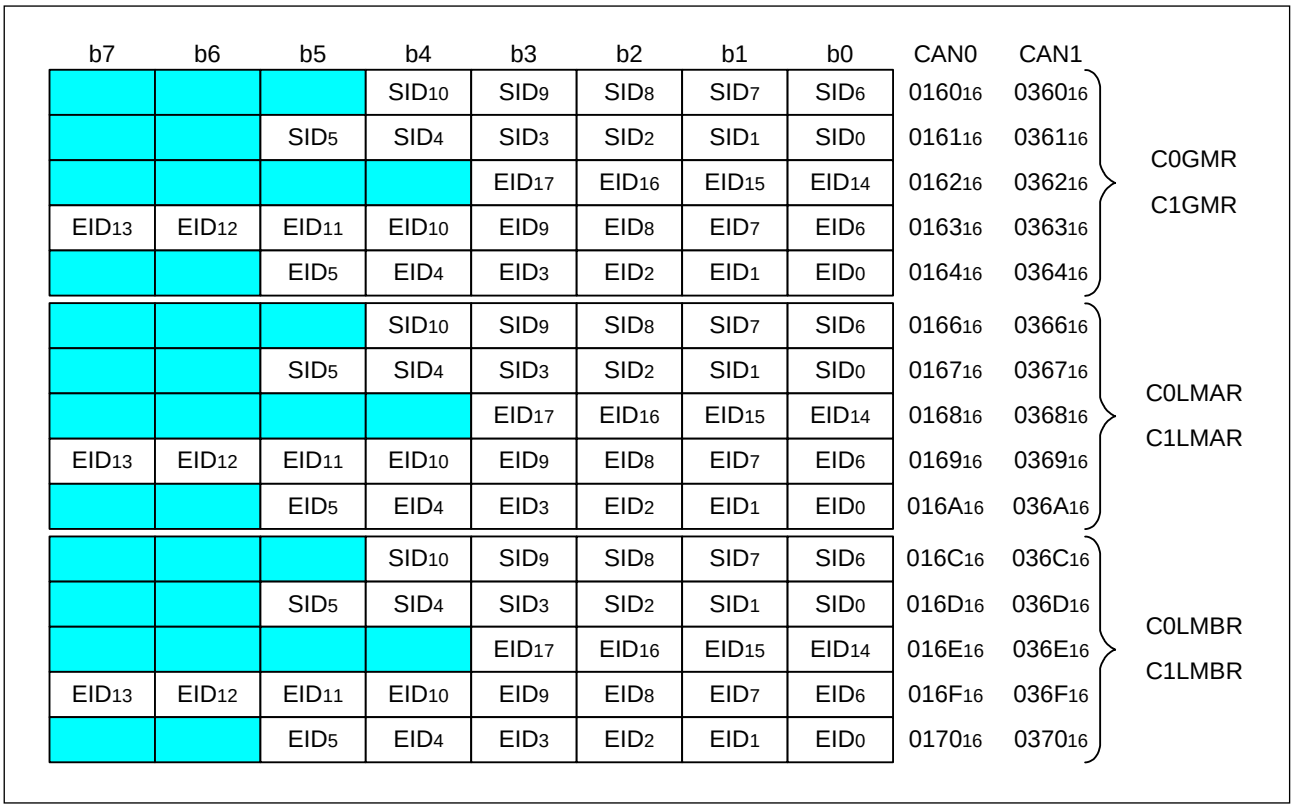
(1) Register structure of the acceptance filter

Figure 31 shows the structure of ID and the mask register. Figure 32 and Figure 33 show memory maps and bit assignments.



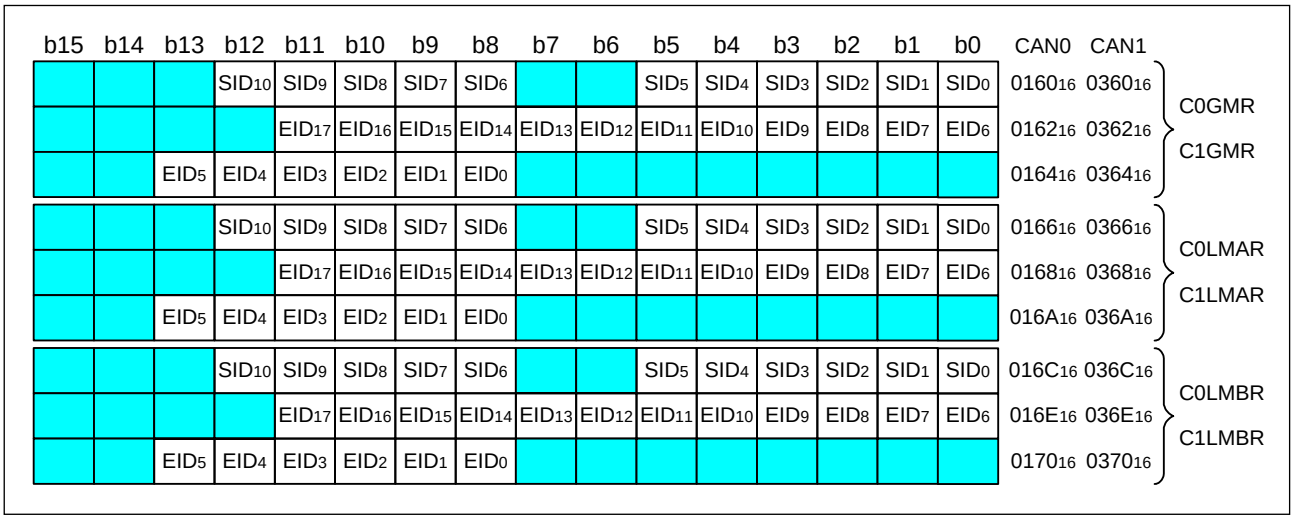
Slot ID	Set the value of the message ID to receive.
Mask register	0: Ignores the corresponding bit of the received message ID. 1: Compares the corresponding bit of the received message ID and the corresponding bit of the slot ID.
Acceptance value	0: Discards the message. 1: Receives the message.

Figure 31 Structure of ID and the mask register



b7	b6	b5	b4	b3	b2	b1	b0	CAN0	CAN1	
			SID10	SID9	SID8	SID7	SID6	016016	036016	C0GMR C1GMR
		SID5	SID4	SID3	SID2	SID1	SID0	016116	036116	
				EID17	EID16	EID15	EID14	016216	036216	
EID13	EID12	EID11	EID10	EID9	EID8	EID7	EID6	016316	036316	
		EID5	EID4	EID3	EID2	EID1	EID0	016416	036416	
			SID10	SID9	SID8	SID7	SID6	016616	036616	C0LMAR C1LMAR
		SID5	SID4	SID3	SID2	SID1	SID0	016716	036716	
				EID17	EID16	EID15	EID14	016816	036816	
EID13	EID12	EID11	EID10	EID9	EID8	EID7	EID6	016916	036916	
		EID5	EID4	EID3	EID2	EID1	EID0	016A16	036A16	
			SID10	SID9	SID8	SID7	SID6	016C16	036C16	C0LMBR C1LMBR
		SID5	SID4	SID3	SID2	SID1	SID0	016D16	036D16	
				EID17	EID16	EID15	EID14	016E16	036E16	
EID13	EID12	EID11	EID10	EID9	EID8	EID7	EID6	016F16	036F16	
		EID5	EID4	EID3	EID2	EID1	EID0	017016	037016	

Figure 32 Memory map and bit assignment at byte access*2



b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0	CAN0	CAN1	
			SID10	SID9	SID8	SID7	SID6			SID5	SID4	SID3	SID2	SID1	SID0	016016	036016	C0GMR
				EID17	EID16	EID15	EID14	EID13	EID12	EID11	EID10	EID9	EID8	EID7	EID6	016216	036216	C1GMR
		EID5	EID4	EID3	EID2	EID1	EID0									016416	036416	
			SID10	SID9	SID8	SID7	SID6			SID5	SID4	SID3	SID2	SID1	SID0	016616	036616	C0LMAR
				EID17	EID16	EID15	EID14	EID13	EID12	EID11	EID10	EID9	EID8	EID7	EID6	016816	036816	C1LMAR
		EID5	EID4	EID3	EID2	EID1	EID0									016A16	036A16	
			SID10	SID9	SID8	SID7	SID6			SID5	SID4	SID3	SID2	SID1	SID0	016C16	036C16	C0LMBR
				EID17	EID16	EID15	EID14	EID13	EID12	EID11	EID10	EID9	EID8	EID7	EID6	016E16	036E16	C1LMBR
		EID5	EID4	EID3	EID2	EID1	EID0									017016	037016	

Figure 33 Memory map and bit assignment at word access*3

*1: The global mask register, local mask register A, and local mask register B in the 1N group microcomputers are located at the CAN1 addresses.
*2: The registers are accessed byte wise when the message order select bit in the CAN control register = 1.
*3: The registers are accessed word wise when the message order select bit in the CAN control register = 0.

(2) Examples for using the acceptance filter

- Usage example 1

  Table 4 shows how each register should be set when slot 0 is to receive a standard data frame or standard remote frame with ID123_{16}.

Table 4 Acceptance filter usage example 1

		SID10-6	SID5-0	EID17-14	EID13-6	EID5-0
Slot 0		00100	100011	XXXX	XXXXXXXX	XXXXXX
Mask register	C0GMR	11111	111111	XXXX	XXXXXXXX	XXXXXX
Receive message	ID123_{16}	00100	100011			

- Usage example 2

  Table 5 shows how each register should be set when slot 0 is to receive two standard data frames or standard remote frames with ID122_{16} and ID123_{16}.

Table 5 Acceptance filter usage example 2

		SID10-6	SID5-0	EID17-14	EID13-6	EID5-0
Slot 0		00100	10001X	XXXX	XXXXXXXX	XXXXXX
Mask register	C0GMR	11111	111110	XXXX	XXXXXXXX	XXXXXX
Receive message	ID122_{16}	00100	100010			
	ID123_{16}	00100	100011			

- Usage example 3

  Table 6 shows how each register should be set when slot 0 is to receive an extended data frame or extended remote frame with ID12345678_{16}.

Table 6 Acceptance filter usage example 3

		SID10-6	SID5-0	EID17-14	EID13-6	EID5-0
Slot 0		10010	001101	0001	01011001	111000
Mask register	C0GMR	11111	111111	1111	11111111	111111
Receive message	ID12345678_{16}	10010	001101	0001	01011001	111000

6.2 Acceptance Filter Support Unit (ASU)

The acceptance filter support unit is a function to determine whether the receive ID is valid or not by means of a table search. To use this function, first register the ID to receive in the data table. Next, store the received ID in the CAN acceptance filter support register, read out the decoded received ID from the CAN acceptance filter support register, and check it by searching the table. The acceptance filter support unit can only be used for the IDs of standard frames.
The acceptance filter support unit will prove effective in the following cases:

- When the IDs to receive cannot be masked by the acceptance filter (Example: IDs to receive = 078_{16}, 087_{16}, 111_{16})
- When there are too many IDs to receive and filtering in software requires an excessive amount of time

6.2.1 Using the Acceptance Filter Support Unit

The following shows how to use the acceptance filter support unit when the IDs to receive are 000_{16}, $00D_{16}$, $6F3_{16}$, $6F4_{16}$, and $6FF_{16}$.

(1) Setting up the data table
In ROM or RAM, prepare a data table in which the IDs to receive are registered. The data table can be located at any desired address.
The data table should be comprised of the bits representing each ID to receive by arranging the value representing the 8 high-order bits (SID_{10-3}) in the vertical axis and the value representing the 3 low-order bits (SID_{2-0}) that has been decoded into 8-bit quantity in the horizontal axis, with the corresponding bits set to 1 and all other bits set to 0.

(2) Writing to the CAN acceptance filter support register
If a message is received by CAN0, write the received ID to the CAN0 acceptance filter support register; if received by CAN1, write the received ID to the CAN1 acceptance filter support register.

(3) Reading out of the CAN acceptance filter support register
Read the value representing the 8 high-order bits (SID_{10-3}) of the received ID and the value representing the 3 low-order bits that has been decoded into 8-bit quantity out of the CAN acceptance filter support register.

(4) Determining the validity of the received ID
Search the data table set in (1) for the value that compares equal to the value read out of the CAN acceptance filter support register in (3) to determine whether the received ID is valid.
Figure 34 shows the structure of the CAN acceptance filter support register. Figure 35 shows the structure of the data table. Figure 36 shows the content of the CAN acceptance filter support register when written to and read out.

b15								b8	b7							b0

Symbol	Address	When reset
C0AFS	0243₁₆,0242₁₆	Indeterminate
C1AFS	0245₁₆,0244₁₆	

Function	Set value	R	W
This register is used to write the standard ID of the received message. When accessed for read, the converted value of the standard ID is read from the register.	Standard ID	RW	

*1: The C0AFS register in the 1N group is located at the addresses 0245 16 and 0244 16.

Figure 34 Structure of the CAN acceptance filter support register

Example: For $6F4_{16}$

High-order 8 bits								Low-order 3 bits			
1	1	0	1	1	1	1	0	1	0	0	($6F4_{16}$)
D				E				3-8 conversion			
Vertical axis of the data table								Horizontal axis			

DE_{16}

Low-order 3 bits		
b2	b1	b0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Set bit 4 (horizontal axis) for XXDE (vertical axis) to 1.

8 bits							
b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0

In the same way, convert 000_{16}, $00D_{16}$, $6F3_{16}$, and $6FF_{16}$ before registering in the data table.

b7	b6	b5	b4	b3	b2	b1	b0	Address
007_{16} 0	006_{16} 0	005_{16} 0	004_{16} 0	003_{16} 0	002_{16} 0	001_{16} 0	000_{16} 1	Top
$00F_{16}$ 0	$00E_{16}$ 0	$00D_{16}$ 1	$00C_{16}$ 0	$00B_{16}$ 0	$00A_{16}$ 0	009_{16} 0	008_{16} 0	Top+01_{16}
$6F7_{16}$ 0	$6F6_{16}$ 0	$6F5_{16}$ 0	$6F4_{16}$ 1	$6F3_{16}$ 1	$6F2_{16}$ 0	$6F1_{16}$ 0	$6F0_{16}$ 0	Top+DE_{16}
$6FF_{16}$ 1	$6FE_{16}$ 0	$6FD_{16}$ 0	$6FC_{16}$ 0	$6FB_{16}$ 0	$6FA_{16}$ 0	$6F9_{16}$ 0	$6F8_{16}$ 0	Top+DF_{16}
$7FF_{16}$ 0	$7FE_{16}$ 0	$7FD_{16}$ 0	$7FC_{16}$ 0	$7FB_{16}$ 0	$7FA_{16}$ 0	$7F9_{16}$ 0	$7F8_{16}$ 0	Top+FF_{16}

1: Valid, 0: Invalid

Figure 35 Structure of the data table

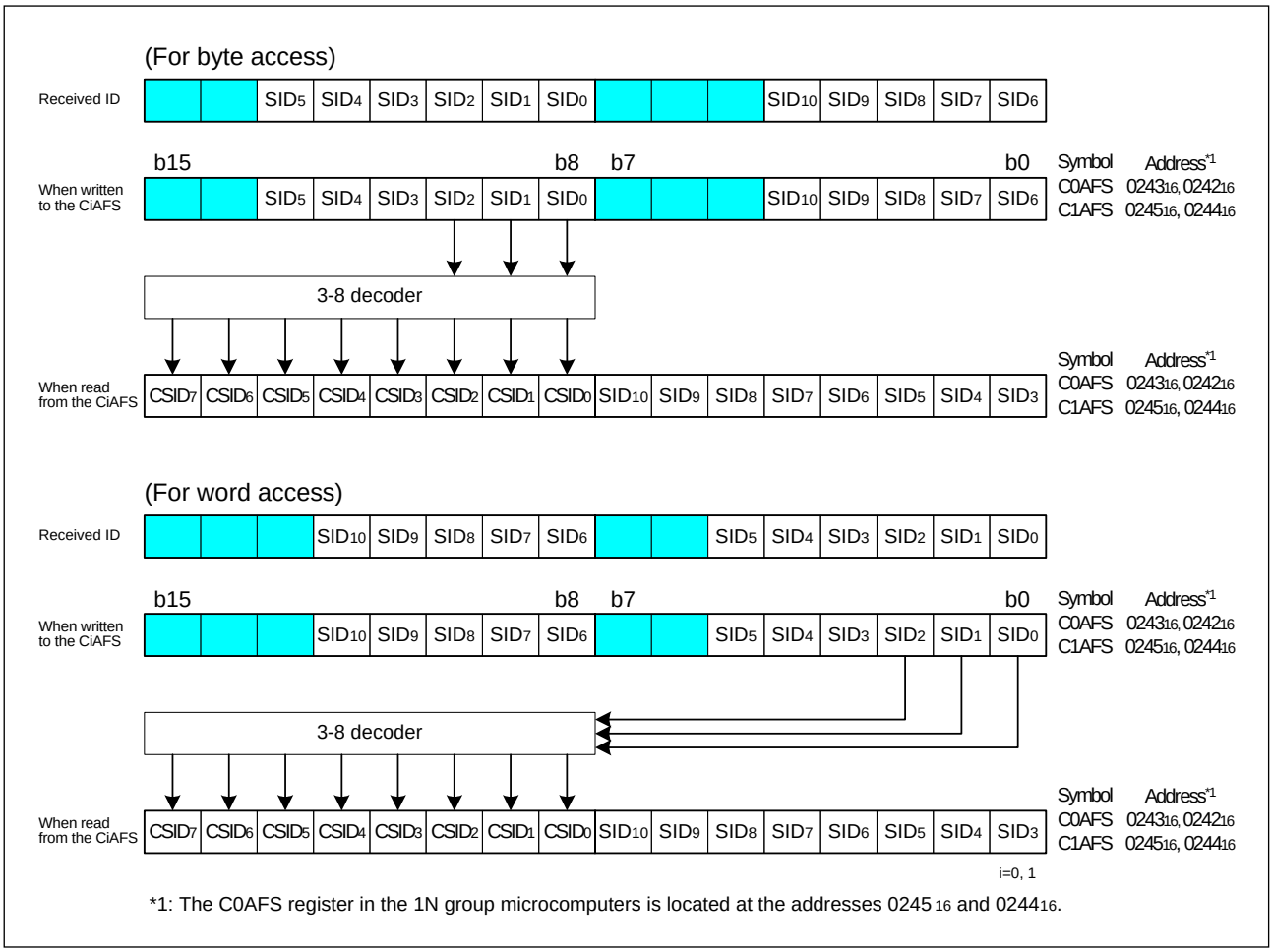
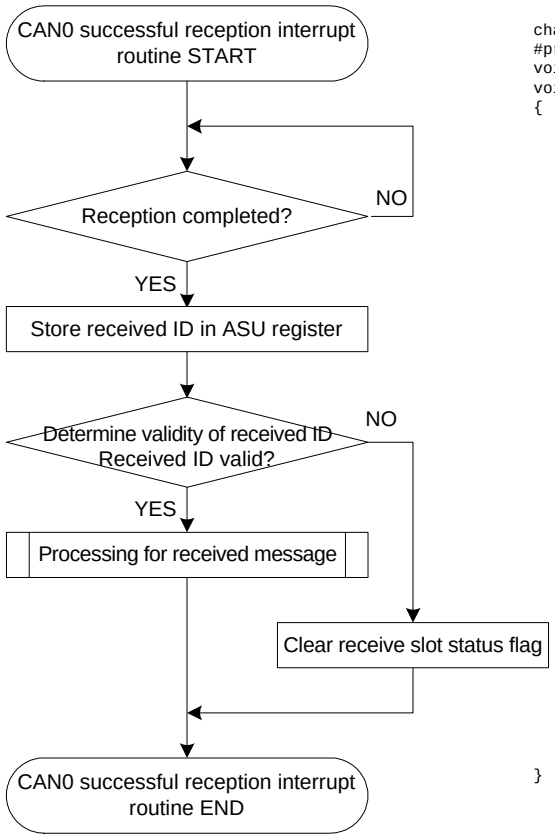


Figure 36 Content of the CAN acceptance filter support register when written to and read out

Figure 37 shows the procedure for using the acceptance filter support unit.
Here, the received ID is checked in a CAN successful reception interrupt routine to see if it is valid or not.




```
char tbl_valid_id[256];  /* Data table */
#pragma INTERRUPT fnc_c0recint
void fnc_c0recint(void);
void fnc_c0recint()
{
     unsigned short in_slot;


     /* Check whether reception is completed */

     for (in_slot=0; in_slot<16; in_slot ++){*1
          if (c0mctl [in_slot]. receive. newdata==1){
               c0afs_addr.word=c0slot0[in_slot].word[0];



               if ((tbl_valid_id[c0afs_addr.byte[0]]&
                    c0afs_addr.byte[1]) !=0){



                    get_message_can0(in_slot,&rec_data);
                    /* User program to be executed when
                       valid is located here */
               }

               else{
                    c0mctl[m_slot].receive.newdata=0 ;
               }

          }

     }

}
```

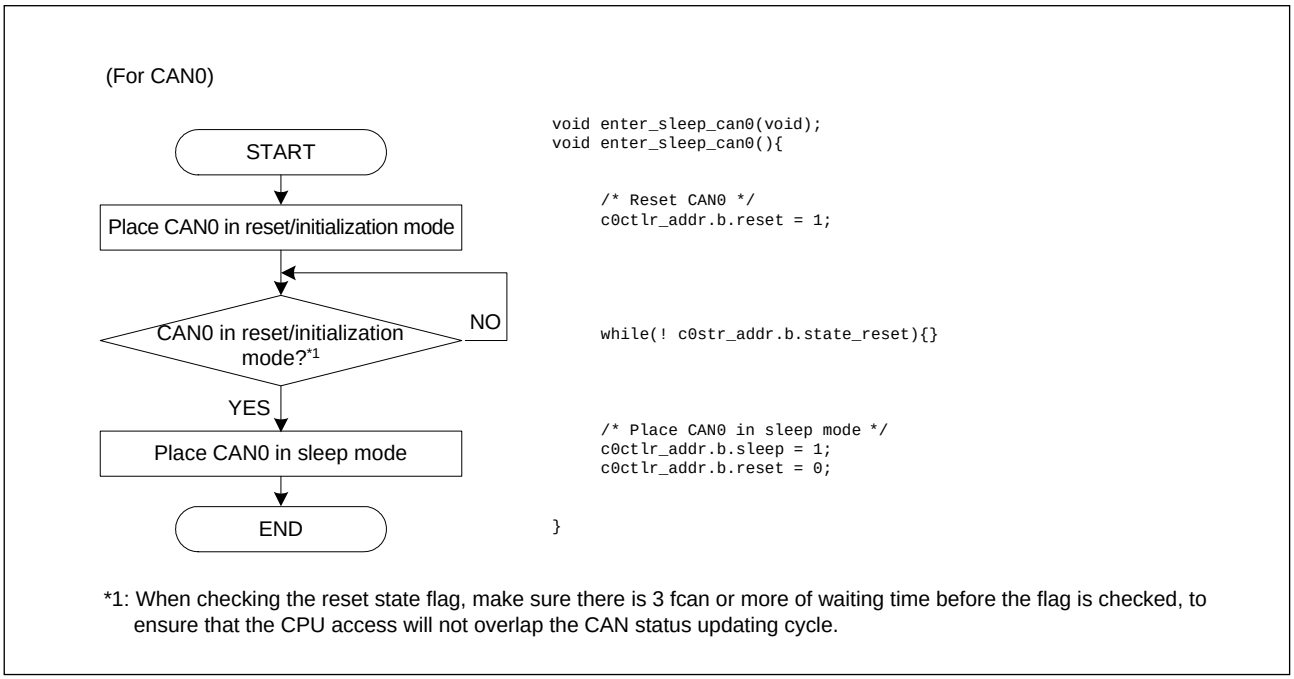
*1: Here, slots are checked in order of 0 to 16. When actually creating a program, however, only the reception-completed slots need to be checked.

Figure 37 Procedure for using the acceptance filter support unit

7. CAN Sleep and CAN Wakeup Operations

7.1 CAN Sleep Operation

When the CAN module is in sleep mode, the clock supplied to it is turned off. Therefore, the CAN module does not operate at all. When not using the CAN module, it is recommended that the CAN module be placed into sleep mode to reduce the amount of current consumed in the chip.
Before placing the CAN module into sleep mode, always be sure to reset it.
Figure 38 shows the procedure for placing the CAN module into sleep mode.
If while in this state the microcomputer is placed into wait or stop mode, the amount of current consumed in the chip can further be reduced.

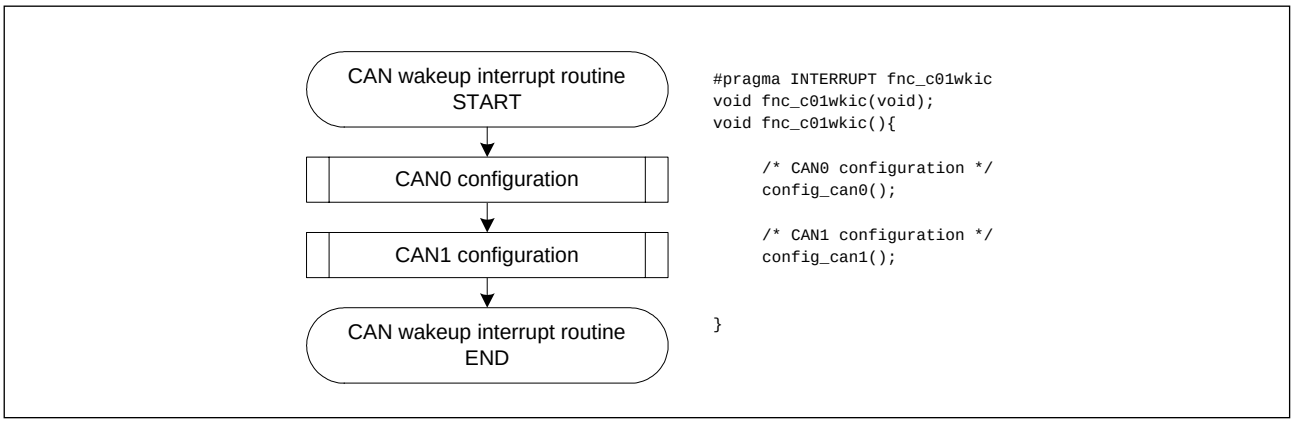


*1: When checking the reset state flag, make sure there is 3 fcan or more of waiting time before the flag is checked, to ensure that the CPU access will not overlap the CAN status updating cycle.

Figure 38 Procedure for placing the CAN module into sleep mode

7.2 CAN Wakeup Operation

While the CAN module is in sleep mode, it can be awaken from sleep mode by a CAN wakeup interrupt that is generated at the falling edge of the CAN receive line. CAN wakeup becomes usable by setting up the CAN wakeup interrupt control register to enable the interrupt.
The CAN wakeup interrupt is used for CAN0 and CAN1 in common. In the 6N4, 6NK, and 6NM groups, the CAN wakeup interrupt can be used for CAN0 and CAN1 individually by setting the IFSR02 bit in the interrupt request cause select register0 to 1. To use the CAN wakeup interrupt to reawaken the CAN module, perform CAN configuration in this interrupt routine.
Figure 39 shows the CAN wakeup procedure.
Here, the CAN modules of both CAN0 and CAN1 are reawaked from sleep mode in a CAN wakeup interrupt routine.




```
#pragma INTERRUPT fnc_c01wkic
void fnc_c01wkic(void);
void fnc_c01wkic(){

     /* CAN0 configuration */
     config_can0();

     /* CAN1 configuration */
     config_can1();


}
```

Figure 39 CAN wakeup procedure

8. Precautions Regarding the Sample Program

8.1 Symbol Notation of Each Register

The symbols representing each register in the sample program of this application note are based on the notation of Renesas standard C language SFR header files.

8.2 while Infinite Loop

To simplify the notation, some parts in the sample program are looped by a while statement. When actually creating a program, put a limit time on each while loop in order to enable the program to exit the loop when the time is over.
Figure 40 shows an example processing where a limit time is placed on a while loop.

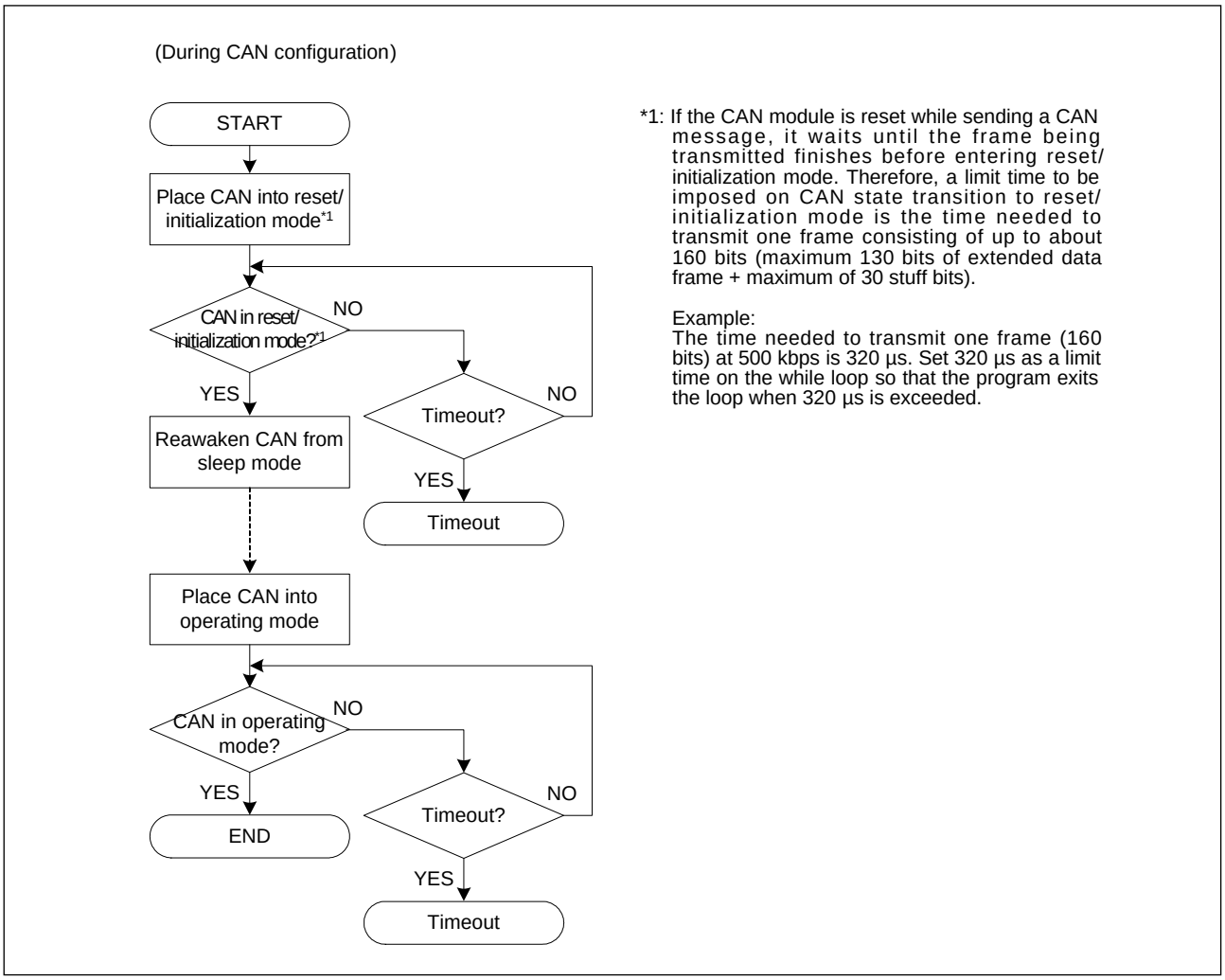


Figure 40 Example processing where a limit time is placed on a while loop

9. Web Site and Support Center

Renesas Technology Web Site
http://www.renesas.com

Where to contact for technical inquiries about CAN microcomputers
Customer Support Center: csc@renesas.com

DEVISION HISTORY	M16C/6N, M16C/1N, M16C/29, and R8C/22,23 Groups CAN Application Note

Rev.	Date	Description	
		Page	Summary
1.00	Nov 05, 2003	–	Unused version
2.00	Jul 05, 2004	–	First edition issued
2.01	Jul 28, 2004	1	The M16C/6NK, 6NL, 6NM, and 6NN groups were added to the application
		7	Tabled 1 was revised
		28	Figure 20 was revised
		44	Figure 34 *1 and 35 were revised
		45	Figure 36 was revised
		47	7.2 CAN Wakeup Operation was revised
2.02	Jun 08.2005	18	Figure 13 was revised
		19	Figure 14 was revised
		26	Figure 19 was revised
3.00	Oct.26.2005	1	The R8C/22,23 groups were added to the application

Keep safety first in your circuit designs!

1. Renesas Technology Corporation puts the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage.
   Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of nonflammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

1. These materials are intended as a reference to assist our customers in the selection of the Renesas Technology Corporation product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corporation or a third party.
2. Renesas Technology Corporation assumes no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
3. All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corporation without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor for the latest product information before purchasing a product listed herein.
   The information described here may contain technical inaccuracies or typographical errors.
   Renesas Technology Corporation assumes no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors.
   Please also pay attention to information published by Renesas Technology Corporation by various means, including the Renesas Technology Corporation Semiconductor home page (http://www.renesas.com).
4. When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corporation assumes no responsibility for any damage, liability or other loss resulting from the information contained herein.
5. Renesas Technology Corporation semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corporation or an authorized Renesas Technology Corporation product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
6. The prior written approval of Renesas Technology Corporation is necessary to reprint or reproduce in whole or in part these materials.
7. If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination.
   Any diversion or reexport contrary to the export control laws and regulations of Japan and/or the country of destination is prohibited.
8. Please contact Renesas Technology Corporation for further details on these materials or the products contained therein.