R8C, M16C, M32C, R32C, and SH RCAN-ET/TL MCUs

CAN Application Programming Interface

REU05B0063-0322
Rev.3.22
Jul 29, 2010

Table of Content

1. Abstract

This article introduces the Renesas CAN Application Programming Interface and explains how to use it to send, receive, and monitor data on the CAN bus. It explains briefly the use of the CAN peripheral's functionalities. As for the API source code, there are alternative ways to use it. For instance, you may want to write your own driver functions to tailor performance or behavior.

In the Appendix, more information on using the CAN peripheral is available and where characteristics of different MCUs are discussed.

This document deals with Full CAN and does not address the usage of Remote Frames.

2. Introduction

The CAN peripheral provides 16 or 32 mailboxes, depending on device, to read from and write to in order to communicate over CAN. These slots, or mailboxes, are message 'buffers' and will hold the CAN data frame until it is overwritten by another incoming frame, or rewritten by the application. Each mailbox can be configured dynamically to transmit or receive. Usually, most are configured to receive and a few to transmit.

The CAN API is available for CAN equipped MCUs in the families R32C, M32C, M16C, M16C, R8C, and also for the SH RCAN-ET MCUs SH7286, SH7137, SH7216, and SH7264. The RCAN_TL1 peripheral type of the SH7264 includes all registers of RCAN_ET, but add's additional registers not used in the API). Most, if not all, CAN MCUs are available on Renesas Starter Kit boards. Sample code and demonstrations using the API are available from Renesas.

3. Nomenclature

'Slot' is equivalent to the words 'mailbox', 'message box', or 'message buffer' and is the physical location where messages are stored inside the MCU's CAN peripheral.

Example source code is in the courier font:

```
c1recic = CAN_LVL;
```

4. CAN Basics

CAN was designed to provide reliable, error-free network communication for applications in which safety and real-time operation cannot be compromised. Its main attributes can be summarized as follows:

- High reliability and noise immunity
- Fewer connections
- Flexible architecture
- Error handling through peripherals
- Low wiring cost
- Scalability

The MCU and bus connectors need only two pins. Therefore, a CAN network is more reliable than other networking schemes that need more wires and connections. Adding new nodes is simple; just tap the bus wire at any point.

CAN is based on a “multiple master, multiple slave” topology. Message or Data Frames transmitted do not contain the addresses of either the transmitting node or of any intended receiving node. This means that any node can act as master or slave at any time. Messages can be broadcast, or sent between nodes depending on which nodes at a particular moment are listening to a certain ID. New nodes can be added without having to update others. Such design flexibility makes it practical for building intelligent, redundant, and easily reconfigured systems.

Bit rate determines the number of nodes that can be connected and cable length. Allowed CAN data bit rates are: 62.5, 125, 250, 500 Kbps and 1 Mbps. At the highest speed, the network can support 30 nodes on a 40-meter cable. At lower speeds, the network can support more than 100 nodes on a 1000-meter cable.

The basic building blocks of a CAN network are a CAN microcontroller, the firmware to run it, a CAN transceiver to drive and read the bus signal, and a physical bus media (2 wires). Choose a CAN MCUs with enough mailboxes to fit your applications.

5. The CAN Peripheral

The Protocol Controller of the CAN peripheral in your CAN MCU must be connected to a bus transceiver located outside the chip via the CAN Tx and Rx MCU pins. The Protocol Controller reads and writes to the peripherals slot (mailbox), depending on how they are configured. The configuration is done through the CAN Special Function Registers described in your MCU’s HW manual. As the registers in the CAN peripheral must be configured and read in the proper sequence to achieve useful communication, a CAN API greatly simplifies this – the API takes numerous tedious issues and does them for you.

After initializing the peripheral, all you need to do is use the receive and transmit API calls, and on a regular basis check for any CAN error states. If an error state is encountered, the application can just wait and monitor for the peripheral to recover, as the CAN peripheral takes itself on-line or off-line depending on its state. After a recovery is discovered, the application should restart.

6. CAN Communication Levels

The figure below shows the CAN communication layers, with the application layer at the top and the hardware layer at the bottom.

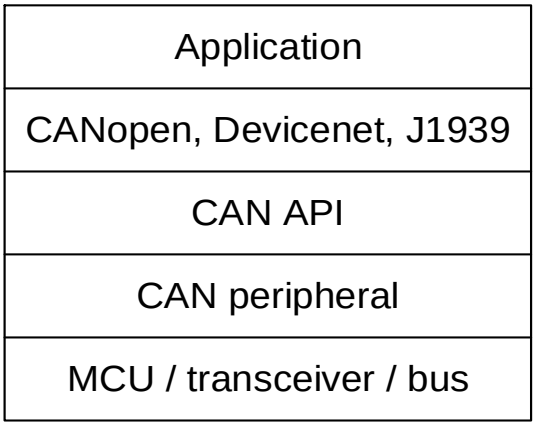

Figure 1. CAN communication layers

In this document we will not discuss any higher level protocols such as CANopen or DeviceNet. However there does exist a CANopen solution for Renesas CAN. Contact Renesas for more information.

7. The Mailbox (Slot)

When a CAN message is to be sent, it must first be written to a mailbox, or mailbox, by the application firmware. It will then be sent automatically as soon as the bus becomes idle, unless a message of lower ID is sent by another node. If a mailbox is configured to receive, the message is written to the mailbox by the Protocol Controller and must be copied by the user, using the API, to user memory area quickly to free the mailbox for the next message coming from the network.

The API calls will do all the writing to and from the mailbox for you. All you have to do is provide application data frame structures which the API functions can write incoming messages to and copy outgoing messages from. It is recommended to have *at least* one structure for outgoing messages, and one for incoming. For outgoing messages this could be a local variable (on the stack). For incoming messages, one for each mailbox is recommended. This CAN data frame structure, of type *can_std_frame_t,* is provided by the API header file and has the following structure:

```
typedef struct
{
    uint16_t id;
    uint8_t dlc;
    uint8_t data[8];
} can_std_frame_t;
```

Note that the timestamp is not included in this structure, but can easily be added.

Aside from CAN bus arbitration, priority is determined by mailbox number for both the transmit and receive operations. If two mailboxes have been set with the same CAN ID, the lowest mailbox number has the highest priority both in sending and receiving for M16C devices, and the highest has priority for SH RCAN-ET/TL MCUs. If two mailboxes are configured to receive the same ID, one mailbox will never receive a message.

8. The CAN API

The API is a set of functions that allow you to use CAN without having to commit attention to all the details of setting up the CAN peripheral, to be able to easily have your application communicate with other nodes on the network.

```
SET UP
void  R_CAN_Initial(void);

R_CAN_Initial calls these by default if necessary
void  R_CAN_SetBitrate(void);
void  R_CAN_SetMask(void);
void  R_CAN_EnablePort(void);          Only R32C, SH
void  R_CAN_DisablePort(void);         Only R32C, SH
void  R_CAN_Halt(void);                Only SH
void  R_CAN_ResumeAfterhalt(void);     Only SH

TRANSMIT
void                 R_CAN_SetTxStdData(const uint8_t mbx_nr, const can_std_frame_t* tx_dataframe_p);
void                 R_CAN_TxStdData(const uint8_t mbx_nr); Only R32C, SH
enum can_api_enum    R_CAN_CheckTxStdData(const uint8_t slot_nr);
void                 R_CAN_AbortTx(const uint16_t mbx_nr);

RECEIVE
void                 R_CAN_SetRxStdData(const uint8_t mbx_nr, const uint16_t sid);
enum can_api_enum    R_CAN_CheckRxStdData(const uint8_t mbx_nr, can_std_frame_t* frame_p);
enum can_api_enum    R_CAN_ReadStdCanMsg(const uint8_t mbx_nr, can_std_frame_t* frame_p);Only R32C, SH
enum can_api_enum    R_CAN_PollRxCAN(uint8_t slot_nr); Only R32C

ERRORS
uint8_t    R_CAN_CheckErr(void);
uint8_t    R_CAN_CheckSwTmo(void);
```

Figure 2. The CAN API functions.

The Set-up group of API functions is used to initialize the CAN peripheral. The first function, can_initial, will by default invoke the rest of the set up functions.

The Transmit functions are used to set up a mailbox to transmit and to check that it actually was sent successfully.

The Receive functions are used to set up a mailbox to receive and to retrieve a message from it.

The Error functions are for checking for errors on the bus and in the CAN peripheral.

The Additional functions are an extension of the original CAN API and vary depending on family and time of implementation of the API library. For example, the can_halt function and the trm_std_dataframe_can are additions made in the R32C and SH CAN APIs. The latter function causes an already set up for transmission mailbox to transmit a frame. The additional functions are not covered in this document as they vary between family but are explained in the source code.

This API eliminates the necessity to study and follow the details of setting up the peripheral, but it is recommended to have the Hardware manual as reference.

9. Setup, Send, and Receive Functions.

Here follows a description of the CAN setup and communication functions.

R_CAN_Initial

Initializes the CAN peripheral
Sets bitrate and masks, sets up mailbox defaults and interrupts

This function will by default invoke the rest of the initialization functions. It also sets the CAN interrupt levels. It will also call all other relevant set-up functions such as

- R_CAN_SetBitrate()
- R_CAN_SetMask()
- R_CAN_EnablePort()

Format

```
void R_CAN_Initial(void);
```

Parameters

-

Return Values

-

Properties

Prototyped in file r_can_api.h
Implemented in file r_can_api.c

Comments

This function sets the CAN operation to reset/initialization mode, and (optionally) the CAN interrupt levels. It also calls the other API functions to set up bit timing and baud rate, and the acceptance mask register default values. After all is set up, CAN is brought back to Operating mode and the physical CAN ports are enabled.

If anything other than default acceptance mask values (do not mask) or 500kbps bitrate are desired, this must be configured inside those APIs.

To prepare this function, you need to set the interrupt priority levels for receive, transmit and error interrupts. If your design polls mailboxes for messages instead of using interrupts, then comment out the interrupt level settings, or set them to 0. Make sure the global interrupt enable flag is set, so the interrupts are not masked.

Example of setting the CAN interrupt levels for M16C:

```
c0recic = CAN_INT_LVL;
c0trmic = CAN_INT_LVL;

/* If using error interrupts instead of polling for errors. Set to 0 if not
desired. */
c0erric = CAN0_ERR_INT_LVL;
```

R_CAN_SetBitrate

Sets the CAN bitrate (communication speed)

The baud rate and bit timing must always be set during the configuration process. It can be changed later on if reset mode is entered.

Format

```
void R_CAN_SetBitrate(void);
```

Parameters

-

Return Values

-

Properties

Prototyped in file r_can_api.h
Implemented in file r_can_api.c

Comments

Setting the baud rate or data speed on the CAN bus is among the more tedious tasks, as it requires some understanding of CAN bit timing and MCU frequency, as well as reading hardware manual figures and tables. The default bitrate setting of the API is 500kB, and unless the MCU clock or peripheral frequencies are changed, it is sufficient to just call the function.

One bit time is divided into a certain number of Time Quanta, Tq. The duration of one Time Quantum is equal to the period of the CAN system clock. Each bitrate register is then given a certain number of Tq of the total of Tq that make up one CAN bit period.

The Appendix has more information on setting the bitrate.

R_CAN_SetMask, R_CAN_SetCanSidMask

Sets the CAN ID Acceptance Masks

To accept only one ID, set mask to all ones. To accept all messages, set mask to all zeros. To accept a range of messages, set the corresponding ID bits to zero.

Format

```
void R_CAN_SetMask(void);
void R_CAN_SetCanSidMask(uint32_t sid_mask_val, uint16_t mask_reg_nr);
```

Parameters

R32C and RX only:

sid_mask_val: The mask value.

mask_reg_nr: Specifies which mask register to set.

Return Values

-

Properties

Prototyped in file r_can_api.h
Implemented in file r_can_api.c

Comments

Receive mailboxes use filters to determine which messages are of interest to them. Typically, those filters are implemented in software. The R_CAN_SetMask function enables mailboxes to accept a broader range of messages than just the single message ID that is set in the mailbox's ID field.

This function takes no arguments for reasons of backwards compatibility. The mask is changed by either using a global variable to set the masks (default M16C), or by having the API call a sub function (R32C, SH) that sets the mask with an argument. In both cases the API is used to invoke setting of the mask.

- Each '**0**' in the mask means "**mask this bit**", meaning don't look at that bit; accept anything.

- Each '**1**' means **check** if the CAN-ID bit in this position matches the CAN-ID of the mailbox.

How to set a mask

Lets say the CAN-IDs you want to receive in a mailbox is, using Standard IDs;

Hex representation	Bit representation
0x0700	0000011100000000b
0x0701	0000011100000001b
0x0702	0000011100000010b
0x0703	0000011100000011b
0x0704	0000011100000100b

The mailbox will only accept frames with an ID that matches the positions whose mask value is 1:

So if we want to accept all above, we set the mask as

111111111111000b, or 0xFFF8.

Only bit positions b15 (msb), to b3 (lsb) will be studied by the CAN module whether they match the receive ID of the mailbox.

If we then set a mailbox to receive ID 0x700 (0x700-0x707 will give the same result) it will accept IDs 0x700 to 0x707. 0x705 to 0x707 must later be ignored 'manually' by the application software.

If you want to receive scattered (or non-continuous) message-IDs, the mask may not be of much help as you must manually filter for desired messages. In this case one may use the Acceptance Filter Support Unit instead. This is faster than pure software filtering. See Appendix for more details.

R_CAN_SetTxStdData

Set up a mailbox to transmit

This API will wait until the mailbox finishes handling a prior frame, write to the given mailbox a given ID, data length and data frame payload, then set the mailbox to transmit mode and send a frame onto the bus.

Format

```
void R_CAN0_SetTxStdData(uint8_t mbx_nr,
                         can_std_frame_t* tx_dataframe_p);
```

Parameters

mbx_nr: The mailbox (slot) to use.
tx_dataframe_p: This argument of type can_std_data_def* is a pointer to a data frame structure in memory. It is an address to the data structure containing the ID, DLC (Data Length Code) and data that constitute the dataframe the mailbox will transmit.

Return Values

-

Properties

Prototyped in file r_can_api.h
Implemented in file r_can_api.c

Comments

The input variable of type can_std_frame_t* refers to a pointer to a data frame structure in application memory.

Usage example

```
#define MY_TX_SLOT7
typedef struct
{
    uint16_t id;
    uint8_t dlc;
    uint8_t data[8];
} can_std_frame_t;
can_std_frame_tmy_tx_dataframe;

my_tx_dataframe.id = 001;
my_tx_dataframe.dlc = 2;
my_tx_dataframe.data.data[0] = 0xAA;
my_tx_dataframe.data.data[1] = 0xBB;

/* Send my frame. */
R_CAN0_SetTxStdData(MY_TX_SLOT, &my_tx_dataframe);
```

R_CAN_CheckTxStdData

Check that a data frame has been sent

Check that specified mailbox made a successful transmission.

Format

```
enum can_rslt_enum   R_CAN_CheckTxStdData(const uint8_t  mbx_nr);
```

Parameters

mbx_nr: The mailbox or slot to use.

Return Values

0 if message is sent.
1 if not sent.

Properties

Prototyped in file r_can_api.h
Implemented in file r_can_api.c

Comments

This API function checks that a data frame was sent from the specified mailbox successfully. There are two methods for checking this:

1) *Polling*. Call the function regularly to poll whether a message from this mailbox has been sent.
2) *The CAN transmit interrupt*. When using transmit interrupts, R_CAN_CheckTxStdData is not necessary since CAN is supposed to dependably deliver the data. In the CAN transmit ISR, confirm that a mailbox caused a transmission interrupt, then flag your application that a successful transmission occurred from that mailbox.

Some versions of the API block the MCU for a short period if the message has not been sent (the length is adjustable with macro). This can easily be eliminated by removing the wait loop, and just keep the code that checks whether the respective mailbox's transmission complete flag is set or not.

R_CAN_SetRxStdData

Set up a mailbox to receive.

The API Sets up a given mailbox to receive dataframes with the given CAN ID. Incoming data frames with the same ID will be stored in the mailbox.

Format

```
void R_CAN_SetRxStdData(uint8_t mbx_nr, uint16_t sid);
```

Parameters

mbx_nr: The mailbox number to use.
sid: The standard CAN message ID that defines which frames the mailbox will receive.

Return Values

-

Properties

Prototyped in file r_can_api.h
Implemented in file r_can_api.c

Comments

The function will set up the mailbox for reception, set the receive identifier type and value, clear the relevant CAN control registers, and enable that mailbox's interrupt if interrupts are used.

Usage example

```
#define MY_RX_SLOT      8
#define SID_FAN_SPEED   0x11

/* Set mailbox 8 to receive. */
R_CAN_SetRxStdData(MY_RX_SLOT, SID_FAN_SPEED);
```

R_CAN_CheckRxStdData

Checks if a mailbox has received a message

The API checks if a given mailbox has received a message. If so, a copy of the mailbox's dataframe will be copied to the given structure.

R_CAN_PollRxCAN and *R_CAN_ReadCanMsg* can be called in succession as an alternative if your API has them.

Format

```
enum can_rslt_enum   R_CAN_CheckRxStdData(const uint8_t  mbx_nr,
                                          can_std_frame_t*     frame_p);
```

Parameters

mbx_nr: Which mailbox to read.
frame_p: The argument type can_std_data_def* refers to a pointer to a data frame structure in memory. It is an address to the data structure into which the function will place a copy of the mailbox's received CAN dataframe.

Return Values

CAN_DATA (1): A message is available and is copied to the structure pointed to by frame_p.
CAN_EMPTY (0): No new message available. Nothing is copied to the application.

Properties

Prototyped in file r_can_api.h
Implemented in file r_can_api.c

Comments

This function is only for backwards compatibility with earlier versions of this API. When a mailbox is set up to receive certain messages, it is important to determine when it has finished receiving successfully. There are two methods for doing this:

1) *Polling*. Call the API regularly to check for new messages. You can use *R_CAN_CheckRxStdData* instead of *R_CAN_PollRxCAN* and *R_CAN_ReadCanMsg* if your API has it.

2) Using the *CAN receive interrupt*. You can use *R_CAN_CheckRxStdData* instead of *R_CAN_PollRxCAN* and *R_CAN_ReadCanMsg* in the CAN receive interrupt routine if your API has it.

The function returns 1 if new data was found in the mailbox. It fetches the message and places a copy of the ID, DLC, and data bytes into the structure pointed to by frame_p. If the API returns 0, no action is taken.

Example:

```
#define MY_RX_SLOT   8
can_std_frame_t   my_rx_dataframe;

if (R_CAN_CheckRxStdData(MY_RX_SLOT, &my_rx_dataframe) == 1)
{
  /* Flag the application. */
  my_rx_dataframe_flag = 1;
}
```

R_CAN_PollRxCAN

Poll a mailbox to know if it has received data

The API checks if a given mailbox has received a message. If so, CAN_EMPTY (0) is returned, if not, CAN_DATA(1) is returned.

Use R_CAN_CheckRxStdData if your API does not have this function and you are using polled mode (R8C, M16C) .

This function is not used for SH. The RXPR0 is read to poll a mailbox whether it has received a frame.

Format

```
enum can_api_enum    R_CAN1_PollRxCAN(uint8_t  mbx_nr);
```

Parameters

mbx_nr: Which mailbox to check.
frame_p: The argument type can_std_data_def* refers to a pointer to a data frame structure in memory. It is an address to the data structure into which the function will place a copy of the mailbox's received CAN dataframe.

Return Values

CAN_DATA (1). For some devices API_OK (0)*:*	A message is available.
CAN_EMPTY(0). For some devices API_NOT_OK (1) :	No new message available.

Properties

Prototyped in file r_can_api.h
Implemented in file r_can_api.c

Comments

When a mailbox is set up to receive certain messages, it is important to determine when it has finished receiving successfully. There are two methods for doing this:

1) *Polling*. Call the API regularly to check for new messages.
2) Using the *CAN receive interrupt*. The API function can be called from the receive interrupt routine to retrieve the message.

The function returns API_OK if new data was found in the mailbox.

Example:

```
#define MY_RX_SLOT   8
can_std_data_def     my_rx_dataframe;

if (R_CAN_CheckRxStdData(MY_RX_SLOT, &my_rx_dataframe) == API_OK)
{
  /* Flag the application. */
  my_rx_dataframe_flag = 1;
}
```

R_CAN_ReadCanMsg

Read the CAN data frame content from a mailbox

The API checks if a given mailbox has received a message. If so, a copy of the mailbox's dataframe will be copied to the given structure.

This function is not used on the SH.

Format

```
void R_CAN_ReadCanMsg(uint8_t          mbx_nr,
                      can_std_frame_t  *frame_p);
```

Parameters

mbx_nr: Which mailbox to read.
frame_p: The argument type can_std_data_def* refers to a pointer to a data frame structure in memory. It is an address to the data structure into which the function will place a copy of the mailbox's received CAN dataframe.

Return Values

-

Properties

Prototyped in file r_can_api.h
Implemented in file r_can_api.c

Comments

This function is used to fetch the the message from a mailbox, either when using polled mode or from a CAN receive interrupt. Use *R_CAN_PollRxCAN()* first to check whether the mailbox has received a message.

Example:

```
#define MY_RX_SLOT   8
can_std_frame_t   my_rx_dataframe;

R_CAN_ReadCanMsg (my_rx_mailbox, &my_rx_dataframe) == 1)
{
  /* Flag the application. */
  my_rx_dataframe_flag = 1;
}
```

R_CAN_EnablePort

Configures the MCU and transceiver port pins

This function is responsible for configuring the MCU and transceiver port pins. Transceiver port pins such as Enable will vary depending on design. These port settings must be added/modified in this function.

Format

```
void R_CAN_EnablePort(void);
```

Parameters

-

Return Values

-

Properties

Prototyped in file r_can_api.h
Implemented in file r_can_api.c

Comments

Make sure this function is called before and after any default port set up function is used (e.g. ‘hwsetup’). You may discover when debugging that a hard reset on a node could cause other nodes to go into error mode. The reason may be that all ports were set as default output hi/low before CAN reconfigures the ports. For a brief period of time, the ports will then be output low and disrupt the CAN bus voltage level.

Change/add transceiver port pins according to your transceiver.

R_CAN_DisablePort

Isolates the MCUs CAN port pins from the bus

Only for SH, and R32C.

Format

```
void R_CAN_DisablePort(void);
```

Parameters

-

Return Values

-

Properties

Prototyped in file r_can_api.h
Implemented in file r_can_api.c

Comments

MCU CAN port pins will be disconnected from the CAN transceiver.

R_CAN_Halt

Set the CAN peripheral into Halt mode.

Only for SH.

Format

```
void R_CAN-DisablePort(void);
```

Parameters

-

Return Values

-

Properties

Prototyped in file r_can_api.h
Implemented in file r_can_api.c

Comments

The function waits until all mailboxes have finished transmitting before transitioning into HALT mode. To turn off communication immediately, disable the CAN bus with e.g. R_CAN_DisablePort().

R_CAN_ResumeAfterHalt

Reenables the CAN peripheral to communicate after a Halt.

Only for SH.

Format

```
void R_CAN_ResumeAfterHalt(void);
```

Parameters

-

Return Values

-

Properties

Prototyped in file r_can_api.h
Implemented in file r_can_api.c

Comments

-

10. CAN API Error Functions

CAN is designed to protect network communication in the event that any CAN network node becomes faulty. Every time the transmitter sees an Error flag, the Transmit Error Counter is increased, and when an error in a received frame is detected, the Receive Error Counter is increased. The Transmit and Receive Error Counters are respectively decreased with every successfully transmitted or received frame. In both the Error Active state (the normal operating state) and the Error Passive State, messages can be transmitted and received.

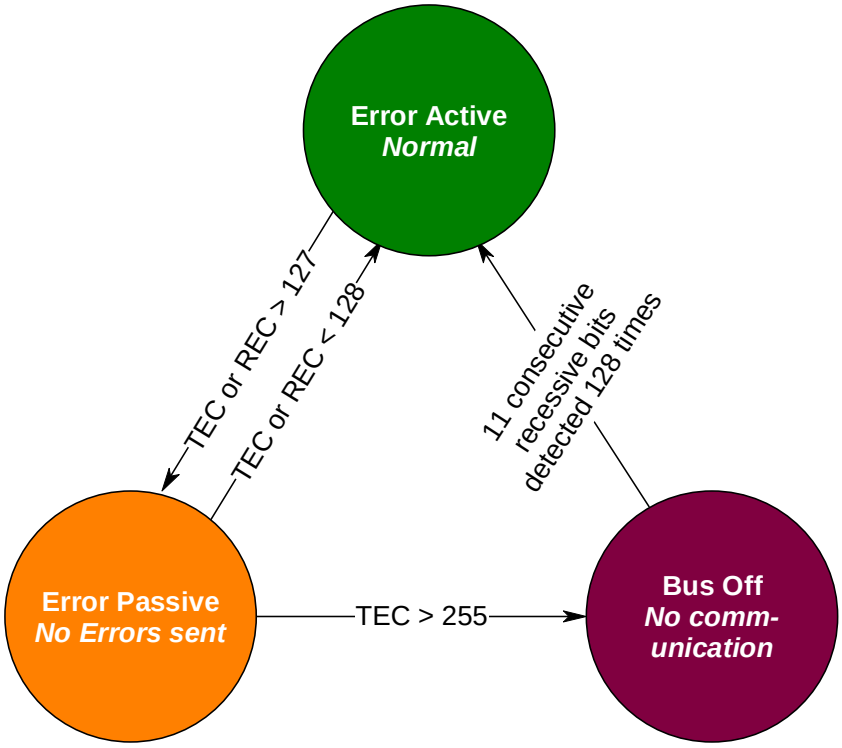

Figure 3. CAN Error States.

Error Active

When a node is in Error Active state it communicates with the bus normally. If the unit detects an error, it transmits an active Error flag. Once it counts 127 errors, it switches to the Error Passive state.

Error Passive

When either error counter exceeds 128, the CAN status for that node changes to state Error passive, and messages can still can be transmitted and received, but the node will not send Error frames. Error frames are invisible to the user and are taken care of by the peripheral silicon.

Bus Off

If the transmit error counter exceeds 255, the CAN node enters the Bus Off state. This prevents a faulty node from causing a bus failure. When serious problems cause a CAN node to enter the Bus Off state, no messages can be transmitted or received by that node until it detects 11 consecutive ‘recessive’ bits 128 times, or until the peripheral is reset.

When the application detects a recovery from Bus Off, the user should reinitialize all registers of the CAN module, and restart the application.

R_CAN_CheckErr

Check for bus errors.

The API checks the CAN status, or Error State, of the CAN peripheral.

Format

```
uint8_t R_CAN_CheckErr(void);
```

Parameters

-

Return Values

CAN_STATE_NO_ERROR (0):	No errors have occurred.
CAN_STATE_ERROR (1):	The node has sent at least one Error frame.
CAN_STATE_ERROR_PASSIVE (2):	The node has sent at least 127 Error frames for either the Transmit Error Counter, or the Receive Error Counter.
CAN_STATE_BUSOFF (4):	The node's Transmit Error Counter has surpassed 255 due to the node's failure to transmit correctly.

Properties

Prototyped in file r_can_api.h
Implemented in file r_can_api.c

Comments

The API checks the CAN status flags of the CAN peripheral and returns the status error code. It tells whether the node is in a functioning state or not and is used for *application* error handling.

It should be polled either routinely from the main loop, or via the CAN error interrupt. Since the peripheral automatically handles retransmissions and Error frames it is usually of no advantage to include an error interrupt routine.

If an error state is encountered, the application can just wait and monitor for the peripheral to recover, as the CAN peripheral takes itself on-line or off-line depending on its state. After a recovery is discovered, the application should restart.

10.1 Polling

Call the API regularly to check the CAN state.

Example:

```
uint8_t error_bus_status;

static void CheckCanBusState(void)
{
   can_std_frame_t err_tx_dataframe;

    /* Has the status register reached error passive or more? */
    error_bus_status = R_CAN_CheckErr();

   /* Tell user if CAN bus status changed. */
   if (error_bus_status != error_bus_status_prev)
   {
        switch (error_bus_status)
        {
            /* Error Active. */
            case CAN_STATE_NO_ERROR:
            case CAN_STATE_ERROR:
             /* There are very few (or no) errors. Tell user node is OK.
*/
                if (error_bus_status_prev > 0)
                /* Only report if there was a previous error. */
                {
                   /* SHOW USER. */
                DisplayString(LCD_LINE1,"Bus OK! ");
                   Delay(1000);
                }
                break;

            /* Error Passive. */
            case CAN_STATE_ERROR_PASSIVE:
            case CAN_STATE_ERROR_PASSIVE + CAN_STATE_ERROR:
            DisplayString(LCD_LINE1,"ErrPassv");
                Delay(1000);
                break;

            /* Bus Off. */
            case CAN_STATE_BUSOFF:
            case CAN_STATE_BUSOFF + CAN_STATE_ERROR:
          /* or higher */
            default:
            DisplayString(LCD_LINE1,"Bus Off ");
                /* SHOW USER */
                   Delay(1000);
               break;
        }
        error_bus_status_prev = error_bus_status;

        /* Transmit CAN bus status change. */
      err_tx_dataframe.id = NODE_NR_ID;
      err_tx_dataframe.dlc =  1;
      err_tx_dataframe.data[0] = error_bus_status;

        /* Send Error state change, in case channel is up. */
        R_CAN_SetTxStdData(TEST_TX_MBX, &err_tx_dataframe);
     }
}
```

In the main application, wait for the peripheral to return to normal:

```
if (error_bus_status < CAN_STATE_ERROR_PASSIVE)
{
  /* Initialize */
   DisplayString(LCD_LINE1, "Renesas ");
   DisplayString(LCD_LINE2, "CanDkit ");
   can_initial();

   /* Start CAN Application */
  InitCanApp();
   state = CAN_INIT;
}
```

10.2 Using CAN Error Interrupts.

The CAN error interrupt can be used to check the error state of the node, although polling with the API regularly is usually sufficient since low level error handling is performed automatically by the peripheral.

The API can be called from the error ISR to determine the error state, and then flag the application if a state transition has occurred. Most often the Transmit or Receive Error Counter will have just incremented.

Interrupts can be enabled separately for each of: A single error, transition to Error Passive, and transition to Bus Off. If the first of these, the CAN Error interrupt is enabled, an interrupt is generated each time an error is detected. Again, generating this interrupt is usually unnecessary as CAN handles errors on its own.

R_CAN_CheckSwTmo

Checks for bus errors in the CAN peripheral.

Checks if the CAN peripheral timed out, and puts the result in the parameter.

Format

```
uint8_t R_CAN_CheckSwTmo(void);
```

Parameters

-

Return Values

The global data variable *canblock_tmo_diag* has one bit flag for each while loop in all of the other APIs. The CAN driver API for some MCUs do not use certain flags as there is no while loop that could timeout.

CAN_SW_RST_ERR (0x01):	Timeout waiting for the peripheral to initialize.
CAN_SW_TRM_TMO (0x02):	Timeout waiting to transmit a frame.
CAN_SW_TRM_DATA_ERR (0x04):	Timeout waiting for the message mailbox to reset when transmitting.
CAN_SW_CHK_TRM_ERR (0x08):	Timeout waiting for a successful data frame transmission.
CAN_SW_SET_REC_ERR (0x10):	Timeout waiting for the message mailbox to reset when receiving.
CAN_SW_GET_ERR (0x20):	Timeout waiting for valid data when receiving.
CAN_SW_ABORT_ERR (0x40):	Timeout waiting for ongoing transmission to halt.
CAN_SW_OTHER_ERR (0x80):	Not used.

Properties

Prototyped in file r_can_api.h
Implemented in file r_can_api.c

Comments

Call this API regularly to verify that the chip is functioning properly. Its return value should be zero. The check is not necessary, as any faults will cause the node to go into a quiet error state, but it can help in determining the cause for error, as each bit in the return value are set individually due to a particular firmware timeout.

This function's main usage is when debugging a new/modified API.

Example

```
unsigned char  canblock_tmo_diag;

/* Check if can peripheral timed out */
canblock_tmo_diag  = R_CAN_CheckSwTmo();
if (canblock_tmo_diag)
{
  /* Take action */
  RED_LED = 1;
}
```

R_CAN_AbortTx

Discard a message that is in transmission status so that another message can be transmitted on the given mailbox.

The function checks to see whether the transmission is completed. If not, it sends a Transmit Abort request. After the Transmit Abort procedure is completed the mailbox is free to send a new message.

Format

```
void R_CAN_AbortTx(const uint16_t mbx_nr);
```

Parameters

-

Return Values

-

Properties

Prototyped in file r_can_api.h
Implemented in file r_can_api.c

Comments

When two or more mailboxes start transmitting at the same time, those with messages having lower priority lose the bus-arbitration process and stop transmitting. After transmission of the message that won the arbitration completes, a new negotiation for bus access is done. In certain cases, a low-priority message of a mailbox may lose the arbitration repeatedly, blocking the mailbox and making it impossible to transmit new messages. This can be prevented by making a transmit abort procedure available.

The abort function discards a message that is set up to be transmitted on the given mailbox so that the mailbox can be used for another purpose, e.g. so a more important message can be sent from that mailbox. This may be desired in the unlikely event of a bus being too heavily loaded so that the message set up for transmission is of such a low priority (high ID) that it repeatedly loses arbitration and is never sent. Again, this is unlikely in a properly designed system.

Usage example

```
#define MY_RX_SLOT   8

R_CAN_AbortTx(MY_RX_SLOT);
```

11. CAN Interrupt Service Functions

It is not necessary to use the CAN interrupts, but they can drive a timely sequencing of your firmware. You may prefer your solution to poll mailboxes for received and successfully transmitted messages. Using the CAN interrupts however gives a somewhat faster application response to events on the CAN bus.

The CAN interrupt logic generates different types of interrupts. The interrupts common for most MCUs are

- Receive Complete
- Transmit Complete
- CAN Error
- Transition to Error Passive
- Transition to Bus Off
- CAN wakeup

As a general rule, it's always advisable to keep interrupt code as short as possible. This will help to keep one interrupt from blocking other interrupts.

11.1 Receive Interrupt

The quickest way to verify that a data frame has been received successfully is to use an interrupt for confirmation — specifically, the CAN Successful interrupt. To do this, enable CAN Successful receive and transmit interrupts as well as a corresponding bit for each mailbox in the CAN interrupt control register. The register is cleared automatically when the CAN peripheral goes to reset/initialization mode, but cannot be set until it is in the operating mode.

The receive interrupt's duty shall be, at the very least, to inform the application that a frame is waiting to be read.

11.2 Transmit Interrupt

It is not necessary to include a Successful Data Frame Transmit interrupt routine in the code. CAN promises the delivery of a message unless the bus is overloaded or the node is in Bus Off mode. But using the CAN transmit interrupt is the fastest way to be notified that a message has indeed been sent successfully and therefore the quickest way to inform the main application to proceed with e.g. an application state machine.

The transmit interrupt's duty shall be at a minimum to check which mailbox (mailbox) finished sending and inform the application that the mailbox (mailbox) is ready to be used again.

11.3 Error Interrupts

These interrupts are actually several independently configurable interrupts: CAN Error interrupt, node transition to Error Passive, and node transition to Bus Off.

The error status given by the API *R_CAN_CheckErr* tells whether the node is in the normal Error Active state or not. This can be polled simply calling the API regularly. The peripheral automatically handles retransmissions of its frames and sending of Error frames. It is therefore it is not necessary to include an error interrupt routine.

Enabling this interrupt does allow for the most rapid detection of the transition to another error status level: Error Passive or — the most serious — Bus Off, and allow the application to take immediate action.

12. Example Code

There is plenty of example source code available for using the CAN API, including how to handle errors. The demonstrations "Streaming A-D" and "Playcatch Net Test" for both R8C/M16C/M32C/R32C and SH RCAN-ET/TL MCUs are available and demonstrate usage of the API.

"Flash-over-CAN" is also available for MCU's with internal flash, which allows reflashing a device in-field using a unique Device Unlock Code.

13. Appendix

Here, more device specific information is gathered, together with functionality which is not yet part of the API, but may migrate to become part of it in future releases of the API.

13.1 Interrupt Setup Checklist

All interrupts mentioned in chapter 11 are available on both SH and M16C family CAN devices. On the SH there are also interrupts for transition to Sleep/Reset mode, transmission of Overload frames, and mailbox overwrites among others.

Use this checklist to make sure interrupts are set up correctly, and if you are having problems getting the interrupts to trigger.

1) To tell the compiler to do a 'return from interrupt' the #pragma directive must be used to tell the compiler that the function is an ISR must be present in the file where the function is defined.

2) Is the ISR function declared as an interrupt with a #pragma directive?

3) Are interrupts enabled globally with e.g. the ENABLE_IRQ macro (or set_mask(n) intrinsic on SH parts) somewhere? If your interrupts does not occur, the flag may have been disabled (by e.g. DISABLE_IRQ). Check by doing this: At a point in the code where your interrupt is expected, set a breakpoint and check that the I-flag is '1'. Check the CPU flag registers for the I-flag.

4) Is the interrupt priority level for the interrupt set to a non-zero value? Check the main Interrupt chapter of the HW manual.

5) Is the respective mailbox's interrupt flag enabled?

6) Is the CAN interrupt mask register set to mask off the interrupt?

7) Make sure that the interrupt ISR function is *declared* in the vector area of your workspace. For SH this is under the statement
   #pragma section INTTBL

   For M16C, using the sect30.inc file this is under
   .org VECTOR_ADR

   Or if you are using a C-startup version of the code, use the pragma statement, e.g.
   #pragma interruptCan0TxInt(vect=96)

For the M16C, add */B* to the *#pragma* interrupt declaration in order to bank-switch to a separate register bank for interrupt context saving. This gives faster IRQ response as the content need not be saved to a stack.

13.2 Loopback (Self Test) - Test a Node without Bus

The Loopback function, or Self Test mode, allows you to check transmitted content without connecting to a bus. This can be useful for self diagnosis during debug.

For the M16C, set the loopback bit during initialization, prior to configuring a mailbox to receive those IDs you wish to test. Loopback mode of M16C cannot send acks.

For the SH RCAN-ET/TL, Loopback is named Test Mode 2. This makes it possible to receive messages sent by the node itself. Test Mode 1 is also for receiving the nodes own messages, but the node must be connected to a bus.

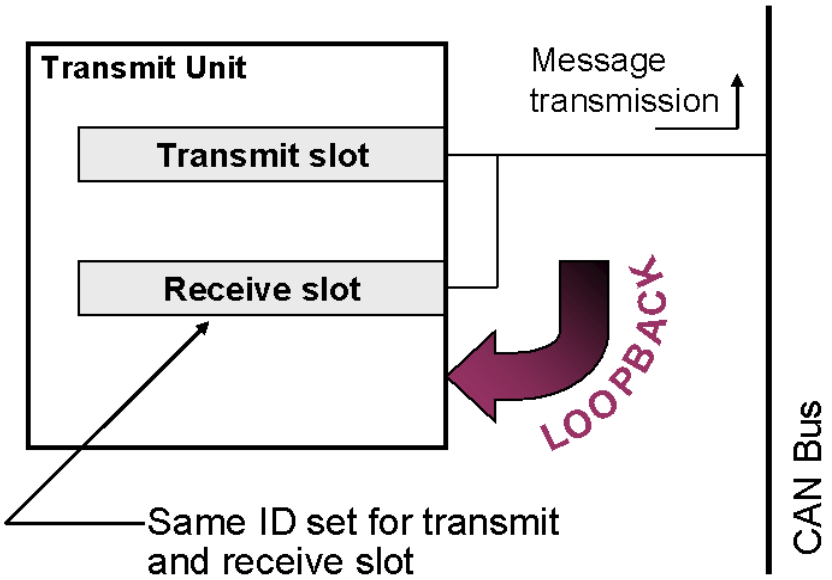

Figure 4. CAN Loopback, or Self Test mode. Loopback lets you test the functionality of a node without having a CAN bus connected.

13.3 Listen Only (Bus Monitoring) - Test a Node Without Affecting the Bus

In Listen Only, or Bus Monitoring, the node is quiet. When other nodes transmit, all units except the one in Listen Only mode will acknowledge the message or send Error frames etc.

"Listen Only" is useful for bringing up a *new* node that has been added to an existing CAN bus. The mode can be used to ensure that a CAN peripheral recently connected to the bus cannot transmit anything to the bus.

A common usage is to detect a bus's communication speed before letting the new unit go 'live'. "Listen Only" is *not* a part of the Bosch CAN specification, but is required by ISO-11898 for bitrate detection.

Caution: Mark entering listen only mode clearly in your code so you remember to disable Listen Only mode again! If you only have two nodes on the network and one of them goes into Listen Only, the other node will not get any acks and will eventually go Bus Off. Also, don't request any transmissions in Listen Only as that is not a correct behavior and the CAN module has not been designed for this.

13.4 Time Stamp

The timestamp function is available for M16C and SH parts equipped with RCAN-TL. It automatically writes the value of the on-chip time stamp counter to receive mailboxes when a message is received. By examining the counter values written to the mailboxes, you can determine the sequence in which a group of multiple messages was received.

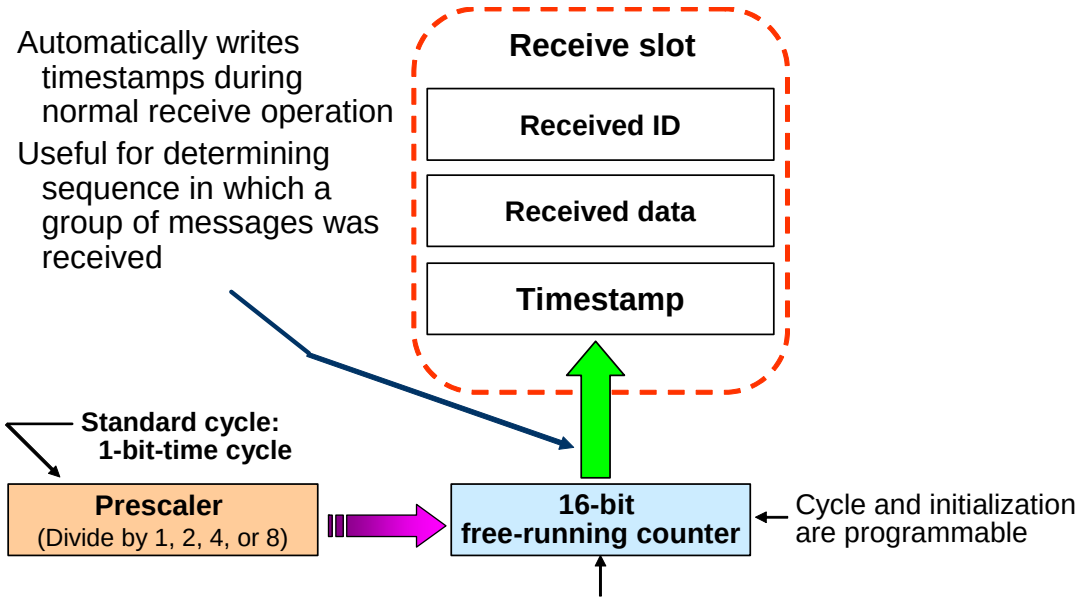

Figure 5. CAN Timestamp Counter

13.5 CAN Sleep Mode

13.5.1 M16C

CAN Sleep Mode

Entering the CAN sleep mode instantly stops the clock supply to the module and thereby reduces power dissipation. It will be awakened by a CAN wake-up interrupt (if configured) generated by a falling edge of the CAN receive line.

CAN Sleep mode is activated by setting the Sleep bit to “1” and the Reset bit to “0” in the C0CTLR register.

CAN Interface Sleep Mode

Entering the CAN Interface Sleep mode instantly stops the clock supply to the CAN module’s CPU interface, thereby further reducing power dissipation. When in this mode, CAN will no longer be awakened by the wake up interrupt.

CAN Interface Sleep mode should only be activated/deactivated when the CAN module is in CAN Sleep mode. It is done by setting the CCLK3 bit in the CCLKR register.

13.5.2 SH RCAN-ET/TL

SH RCAN-ET/TL offers the ability to enter Sleep Mode to lower power consumption. RCAN-ET/TL is required to be in Halt mode before requesting to enter in Sleep mode. When CAN bus activities are detected, RCAN-ET/TL automatically exits Sleep mode.

The Message Control register is used to set up Sleep mode. MCR5 enables or disables Sleep mode transition. MCR7 enables or disables the Auto-wake mode.

13.6 Acceptance Filter Support Unit

The Acceptance Filter Support Unit (AFSU) provides a means to achieve a faster response time than pure software filtering of message IDs. It is only available for some M16C CAN MCUs (e.g. M16C/xN).

Software filtering can be time consuming as the Standard ID bits are rearranged and not stored as a normal word in memory. Another problem could be that the acceptance mask may not be able to be set to receive the particular combination of messages you want. If you set the mask to accept all messages you may have to ‘waste’ time by checking a long list of the messages using software for each incoming ID. This manual filtering’ would also involve having all the IDs in a readable format. An efficient solution in such cases is to use the Acceptance Filter Support Unit.

To use it, one writes the CAN-ID as it is stored in the message box into the ASU. When read back from the ASU register it reads

Bit 0-7 = Table Address Search Info, 'ASI'

Bit 8-15 = Bit Search Information, 'BSI'. *SID0-3 has now been converted to a bit position to enable a faster table searches.* Use the output to search through a table.

The search table. A table must be prepared by the user to check whether an ID is of interest to the application. The firmware must search the table at each byte address *ASI* and bit position *BSI*. If a bit BSI-value is set in the user's table, the bit pattern matches the BSI pattern of the register which means the address is of interest to the node, and the frame should be processed by the application.

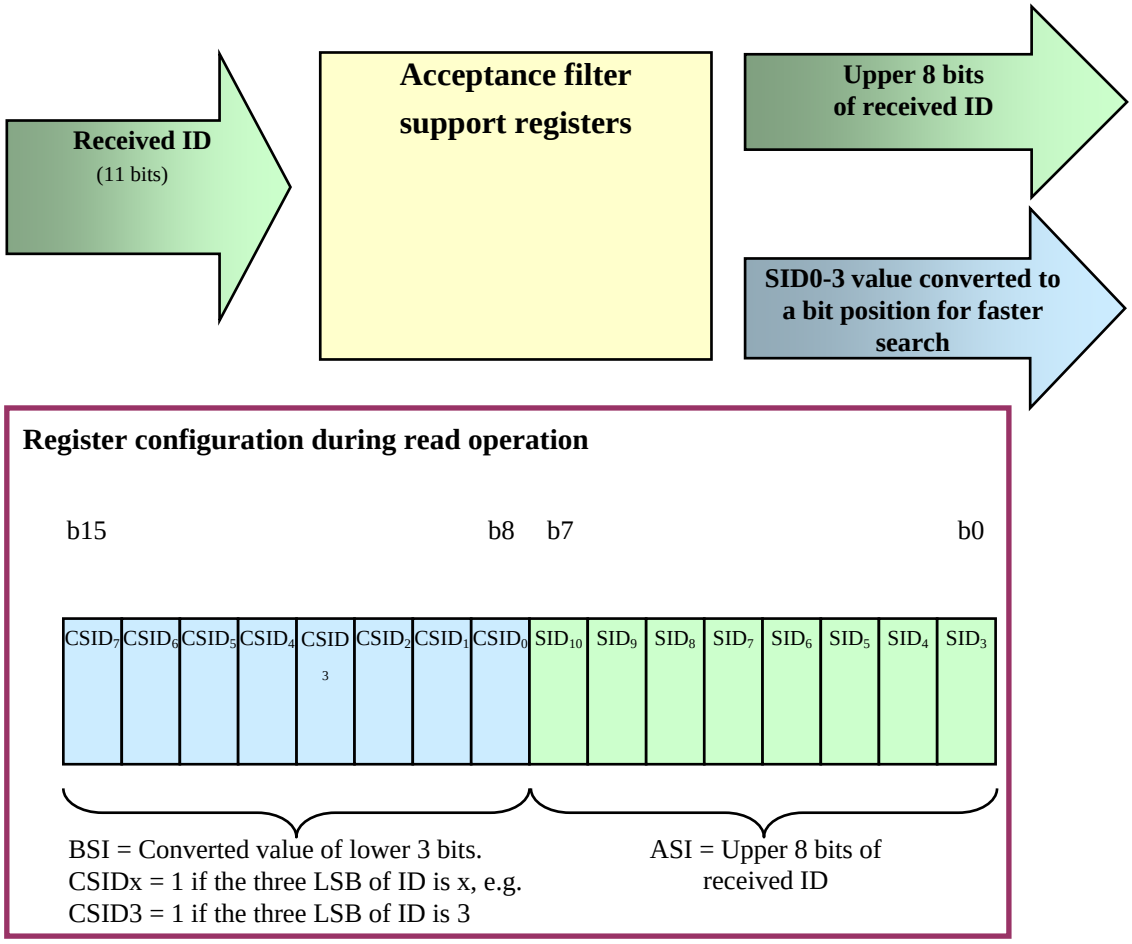

Figure 6. The AFU. When read, the representation in the register of the ID is of a format that enables a quick search through a table. This provides a faster response than a search through a 'normal' array of CAN IDs.

See REJ05B0276 "CAN Application Note" for more information on how to use the AFU.

13.7 Setting the Bit Rate

13.7.1 M16C

Bitrate =Xin / (2 * CAN_div_ratio * Prescaler * Tqtotal)

Tqtotal: We need a certain number to be able to tune the sampling points. For example acc. to HW-manual it must be 8-25.

CAN_div_ratio: CAN frequency division ratio is minimum 1.

Prescaler: Prescaler value is minimum 1.

13.7.2 Set Bitrate Worksheet for M16C

Bring up the HW manual's 'Examples of Bit-rate' table in the CAN chapter. With the frequency dividers and the CAN bit timing parameters you need to set the number of Time Quanta or Tq units you have per bit. This is determined by:

a. Table row value _____ kbaud (desired rate).

b. Table column; the MCU operating frequency "f1" = _____ MHz.

c. The table values are the total Time Quanta value to disperse among all the timing parameters. Examine Note 5 under the table and Figure 5 of the main Renesas CAN Application Note! There are usually two alternatives in each box. Pick one (the higher the Tq value, the more finely you can tune the communication sampling points).

_____Tq (____)

The first value we will call Total Tq. The sum of all timing units must add up to be Total Tq.
The value in parentheses is (fCAN div by N value * Baud rate prescaler value) or

(_____*_____) To get the value set out (in the brackets) we need:

cclkr = _____ fCAN is the frequency you will get after downscaling with cclkr.
brp = _____ fCANCLK is the frequency you will get after downscaling with brp.

See the HW-manual for what value to write to these registers.

Choose bit timing values according to the Total Tq value you have. The sum of the bit timing parameters must be equal to this. See Table 2 in the CAN Application Note for examples of values to choose. You must decrement most values in table by one, because e.g. '1' in the table corresponds to a '0' in the register.

Parameter	Value Table	Value for register (table value -1)	Comment
SAMPLES	1	0	Set 0 in code
SJW	1	0	
Total Tq = PTS + PBS1 + PBS2 + SS:			
PTS			
PBS1			
PBS2			
SS	1	-	No need to set in code

Total Tq: PTS+PBS1+PBS2+SS = ____ Tq. Check that it matches your choice from c. Change the values for

baud rate in the can01.h file. To set these values in the registers, see the C0CONR register settings from the HW manual.

The CAN clock select register CCLKR is protected from being overwritten by accident in case of program runaway. To change the value of this register, the protect function must be cancelled with the protect register PRCR. After the protect register is cleared, the CAN clock divide-by-N is set, i.e. a new value is written to CCLKR. Next, the protect register is set to protect again. Finally, the prescaler divide-by-N value, sampling count, and sampling points are all set.

13.7.3 Setting the Bitrate for SH RCAN-ET/TL

One CAN bus bit-time is $T_{CAN} = 1/f_{CAN}$. A Time quanta, **Tq**, is one bit-time of the CAN system clock, f_{CANclk}. This is *not* the CAN bit-time but the internal clock period of the CAN peripheral. This CAN system clock in turn is determined by (twice) the Baud Rate Pre-scaler value and the peripheral bus clock, f_{clk}, to create the CAN system clock.

Tq $= 1/f_{CANclk} =$ **2 * BRP * T_{clk} = 2 * BRP / f_{clk}.**

One CAN bit-time, T_{CAN}, is 8-25 Tq, depending on how we set the registers for PRSEG, PHSEG1, PHSEG2, and TSEG1:

PRSEG: Segment for compensating for physical delay between networks.

PHSEG1: Buffer segment for correcting phase drift (positive). (This segment is extended when synchronisation (resynchronisation) is established.)

PHSEG2: Buffer segment for correcting phase drift (negative). (This segment is shortened when synchronisation (resynchronisation) is established)

TSEG1: TSG1 + 1.

Finally, the RCAN-ET/TL Bit Rate Calculation is:

Bitrate = fclk / (2 * (BRP + 1) * (TSEG1 + TSEG2 + 1))

The Bit Configuration Registers BCR0 and BCR1 are used to set CAN bit timing parameters and the baud rate pre-scaler (BRP) for the CAN Interface. Follow the tables in the MCU HW-manual how to set the values for a particular bitrate.

Example: Desired bitrate 500 kbps, with a 50 MHz peripheral clock fclk.

With BRP = 4, fCLK = 50 MHz in the bitrate formula, last above, we get:

500 000 = 50 000 000/(2(4 + 1)(TSEG1 + TSEG2 + 1))

This gives

(TSEG1 + TSEG2 + 1) = 50 000 000 / (500 000*(2(4 + 1))

or

(TSEG1 + TSEG2 + 1) = 500/(5*(2(4 + 1))

or

(TSEG1 + TSEG2 + 1) = 10

We can now choose TSEG1 to be from 4 to 16 according to the TSEG1 and TSEG table of the HW manual. Let's choose 5 Tq, and so we set **TSEG1 = 4** (subtract one when setting register).

TSEG2 must be less than TSEG1 so we set TSEG2 to 4 Tq, or **TSEG2 = 3**.

In the code we then write

RCANET.BCR1.WORD = 0x4300;

Now, set **BRP to 4** which will give us a 10 peripheral bus clock periods in one Tq according to the BRP settings table.

RCANET.BCR0.WORD = 0x0004;

13.8 CAN FIFO

CAN FIFO buffering is available for the R32C. The Receive and Transmit FIFO size is four mailboxes each. If they are not used, the mailboxes are available as regular send and receive mailboxes. The FIFO can be used by polling or with interrupts. There is sample code available that demonstrates its usage with both.

14. More Information

Hardware manuals for the MCUs mentioned are available at http://www.renesas.com. Click on MPU/MCU and click on your MCU from the menus.

CAN Application Note REJ05B0276 for R8C and M16C MCUs. Download from http://documentation.renesas.com/eng/products/mpumcu/apn/rej05b0276_16ccanap.pdf

CAN Specification Version 2.0. 1991, Robert Bosch GmbH

Website and Support

Renesas Electronics Website
http://www.renesas.com/

Inquiries
http://www.renesas.com/inquiry

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Jun ‘06	—	First version
2.00	Dec ‘07	1 to 27	Second edition issued
2.02	Jun ‘08	15,23,25	Update recover from Bus off
2.04	Sep ‘08		Section 8.4; set_can_mask changed
2.05	Dec ‘08	1,25	Titles, R32C added.
3.00	Feb ‘09	All	Complete change for common Application note for M16C and SH RCAN-ET.
3.01	June ‘09	2.Chap 12.7. 28.	MCUs affected. Bitrate settings. Copyright text – bottom line.
3.02	June ‘09	5.12	Rewrote CAN sleep mode text.
3.03	Aug ‘09	All	Added SH7216
3.20	Nov ‘09	All	Updated to match coding standards. Added new functions R_CAN_EnablePort, R_CAN_TxStdData, and R_CAN_PollRxCAN.
3.21	Jan ‘09	All	- New table index. - Clarified R_CAN_CheckRxStdData, R_CAN_PollRxCAN and R_CAN_ReadCanMsg. - Added API functions R_CAN_EnablePort and R_CAN_DisablePort (for SH and R32C only). Added R_CAN_Halt and R_CAN_ResumeAfterHalt (for SH only).
3.22	Jul ‘10	All	- Updated Table of Contents; removed an extra copy of R_CAN_EnablePort() (there were 2 copies); - Minor clarifications throughout, including corrections to match the example code here to the latest API software. (function call names, variable names, parameter types, etc.) - Added references to API for SH7264 and RCAN-TL where applicable.

General Precautions in the Handling of MPU/MCU Products

The following usage notes are applicable to all MPU/MCU products from Renesas. For detailed usage notes on the products covered by this manual, refer to the relevant sections of the manual. If the descriptions under General Precautions in the Handling of MPU/MCU Products and in the body of the manual differ from each other, the description in the body of the manual takes precedence.

1. Handling of Unused Pins
   Handle unused pins in accord with the directions given under Handling of Unused Pins in the manual.
   — The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible. Unused pins should be handled as described under Handling of Unused Pins in the manual.
2. Processing at Power-on
   The state of the product is undefined at the moment when power is supplied.
   — The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the moment when power is supplied.
   In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the moment when power is supplied until the reset process is completed.
   In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the moment when power is supplied until the power reaches the level at which resetting has been specified.
3. Prohibition of Access to Reserved Addresses
   Access to reserved addresses is prohibited.
   — The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.
4. Clock Signals
   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.
   — When the clock signal is generated with an external resonator (or from an external oscillator) during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Moreover, when switching to a clock signal produced with an external resonator (or by an external oscillator) while program execution is in progress, wait until the target clock signal is stable.
5. Differences between Products
   Before changing from one product to another, i.e. to one with a different type number, confirm that the change will not lead to problems.
   — The characteristics of MPU/MCU in the same group but having different type numbers may differ because of the differences in internal memory capacity and layout pattern. When changing to products of different type numbers, implement a system-evaluation test for each of the products.

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.
   - "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.
   - "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.
   - "Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

RENESAS

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

Refer to "http://www.renesas.com/" for the latest and detailed information.

Renesas Electronics America Inc.
2880 Scott Boulevard Santa Clara, CA 95050-2554, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada
Tel: +1-905-898-5441, Fax: +1-905-898-3220

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

Renesas Electronics Europe GmbH
Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.
7th Floor, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100083, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 204, 205, AZIA Center, No.1233 Lujiazui Ring Rd., Pudong District, Shanghai 200120, China
Tel: +86-21-5877-1818, Fax: +86-21-6887-7858 / -7898

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2886-9318, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
7F, No. 363 Fu Shing North Road Taipei, Taiwan, R.O.C.
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
1 harbourFront Avenue, #06-10, keppel Bay Tower, Singapore 098632
Tel: +65-6213-0200, Fax: +65-6278-8001

Renesas Electronics Malaysia Sdn.Bhd.
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics Korea Co., Ltd.
11F., Samik Lavied' or Bldg., 720-2 Yeoksam-Dong, Kangnam-Ku, Seoul 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

Colophon 1.0