RX Family

CAN API Using Firmware Integration Technology

Introduction

The Renesas CAN Application Programming Interface enables you to send, receive, and monitor data on the CAN bus. This manual explains the usage of this API and some of the features of the CAN peripheral.

Bundled with this application note comes the CAN API driver source code files. Demonstration source code for the API is included in the download, the demo code essentially being in *can_api_demo.c,* and *switches.c*. The demo allows the user to press board switches to send CAN frames and to change demo receive and transmit CAN IDs.

Target Devices

The following is a list of devices that are currently supported by this API:

- RX64M Group
- RX71M Group
- RX65N, RX651 Groups
- RX66T Group
- RX72T Group
- RX72M Group

When using this application note with other Renesas MCUs, careful evaluation is recommended after making modifications to comply with the alternate MCU.

Target Compilers

- Renesas Electronics C/C++ Compiler Package for RX Family
- GCC for Renesas RX
- IAR C/C++ Compiler for Renesas RX

For details of the confirmed operation contents of each compiler, refer to “9.1 Confirmed Operation Environment.”

Contents

1. Overview

The RX CAN peripheral has 32 CAN mailboxes with which it can communicate on a CAN bus. The term ‘mailbox’, or in some literature ‘message box’ or ‘message buffer’ refers to the physical location where messages are stored inside the MCU’s CAN peripheral. In this document we will use the term ‘mailbox’. The mailboxes are message ‘buffers’ and will hold a CAN data frame until overwritten by either incoming data, or rewritten by the MCU.

Each mailbox can be configured dynamically to transmit or receive. Most are usually configured to receive and fewer to transmit, but this is completely flexible.

1.1 Basics

CAN was designed to provide extremely reliable communication for applications in which safety and real-time operation is a priority.

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 for a specific 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.

Main attributes of CAN may be listed as

- High reliability and noise immunity
- Error handling on silicon
- Two bus wires / node connection points - Low wiring cost
- Flexible architecture
- Easy to scale to large network

Complex stack software to take care of error handling at the low level is not needed since this takes place in silicon. Since the MCU bus connectors need only two pins, a CAN network is also at the physical level more reliable than networking schemes that need multiple bus connections. Adding new nodes is simple; just tap the bus wire at any point.

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.

1.2 Communication Layers

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

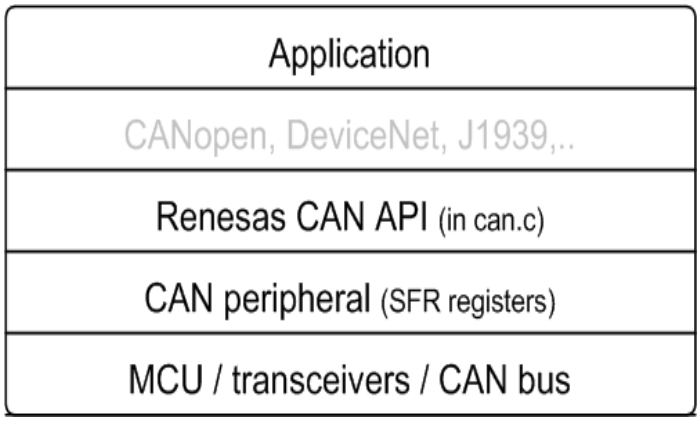

Figure 1. CAN physical and source code layers.

In this document we will not discuss any higher-level protocols such as CANopen or DeviceNet. (For some Renesas CAN MCUs there is a CANopen solution. Contact your sales representative.)

1.3 Physical Connection

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 Transmit (CTXn) and receive (CRXn) MCU pins.

1.4 The CAN Mailbox

The CAN Protocol Controller reads and writes to the CAN peripheral mailboxes. When a CAN message is to be sent, it must first be written to a 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 a 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_frame_t, is provided by the API header file and has the following structure:

```
typedef struct
{
    uint32_t   id;
    uint8_t    dlc;
    uint8_t    data[8];
} can_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 using the lowest mailbox number - except for SH (RCAN-ET) where the highest mailbox has priority. This is true for both transmit and receive operations. If two mailboxes have been set with the same CAN ID, the lowest mailbox number has the highest priority. Therefore, if two mailboxes are configured to receive with the same ID, one mailbox will never receive a message.

1.5 Extended CAN

To use extended ID, *FRAME_ID_MODE* in *r_can_rx_config.h* must be set. When Extended CAN is enabled, the API functions ending in ‘Xid’ can be called. These functions will automatically cause the ID field of the CAN mailbox to be formatted to use extended ID. In other words, the user need only call these Xid-functions, and the ID value passed in the *can_frame_t* structure will be sent as a 29-bit ID (instead of 11-bit).

2. API Information

The names of the APIs of the RSPI FIT module follow the Renesas API naming standard.

2.1 Hardware Requirements

This driver requires that your MCU supports the following peripheral:

- CAN Module (CAN)

2.2 Hardware Resource Requirements

This section details the hardware peripherals that this driver requires. Unless explicitly stated, these resources must be reserved for the driver, and cannot be used elsewhere in the application.

2.2.1 Peripheral Required

CAN Module (CAN)

2.2.2 Other Peripherals Used

The driver requires I/O port pins to be assigned for CAN bus receive and transmit signals. Assigned pins may not be used for GPIO.

The driver optionally uses GPIO port pins for Standby and Enable corresponding to each CAN channel.

2.3 Software Requirements

This driver is dependent upon the following FIT module:

- Renesas Board Support Package (r_bsp) v5.20 or higher

2.4 Limitations

- CAN FIFO operation not supported.

2.5 Supported Toolchain

This driver has been confirmed to work with the toolchain listed in 9.1 Confirmed Operation Environment.

2.6 Header Files

All API calls and their supporting interface definitions are located in “r_can_rx_if.h”.

Build-time configuration options are selected or defined in the file "r_can_rx_config.h”.

To reference CAN API elements in this FIT Module from your code include the following:

#include “r_can_rx_if.h”

2.7 Integer Types

This software uses ANSI C99. These types are defined in stdint.h.

2.8 Configuration

It will be necessary to make modifications to the *r_can_rx_config.h* file to customize the application for desired functionality. For example, there is the option of running in CAN polled mode or CAN interrupt mode. It is not recommended to change the *r_can_rx.c* file, which contains the Renesas CAN API driver function, but this may be merited to add some feature not available with the API.

If installing this software by using the “Smart Configurator” in e^2 studio, the configuration settings for this FIT module are made through the Smart Configurator “Components-> Property” view. Otherwise, *r_can_rx_config.h* can be edited manually using the following tables as a guide.

2.8.1 Interrupt vs. Polled Mode and CAN Interrupt Level

Set the method of checking CAN mailboxes for messages received and sent. If Interrupt mode is used, then also set the interrupt level for the channel

Define	Value	Meaning
USE_CAN_POLL	0 = Use interrupt, not polled 1 = Use polling, not interrupt	Define method of checking CAN mailboxes for messages received and sent.
CAN0_INT_LVL	Valid range = 0 to 15 (0 to disable)	Sets the CAN interrupt level for channel 0
CAN1_INT_LVL	Valid range = 0 to 15 (0 to disable)	Sets the CAN interrupt level for channel 1
CAN2_INT_LVL	Valid range = 0 to 15 (0 to disable)	Sets the CAN interrupt level for channel 2

2.8.2 Standard & Extended CAN IDs

Select what type of CAN ID type to enable in the driver, that is, usage of 11-bit Standard, or 29-bit Extended CAN IDs. The API can be set to STD_ID_MODE, EXT_ID_MODE, or MIXED_ID_MODE. If it is set to mixed mode, the whole API becomes available.

Define	Value	Meaning
FRAME_ID_MODE	STD_ID_MODE = 11-bit CAN ID. EXT_ID_MODE = 29-bit CAN ID. MIXED_ID_MODE = 11-bit and 29-bit IDs are both in use	STD_ID_MODE or EXT_ID_MODE enables only those API functions belonging to that ID mode. If it is set to mixed mode, the whole API becomes available.

Note: MIXED_ID_MODE must be used if there will be both Standard and Extended frames on the bus, otherwise unexpected data may result.

2.8.3 CAN Channel enabling and Pin Mapping

The CAN channel must be enabled here to be included in the driver build. Disabling a channel will remove some code from build. The CAN TX and Rx pins settings will affect the pin configuration, port mode, direction of the pins. There are only certain pins that can act as TX and Rx CAN pins.

Specify also where the CAN transceiver control pins are physically connected to the MCU. This is much more flexible as they are not specific to the CAN peripheral. General IO is used for this. Some transceivers may have other control pins for which the user will need to add own configuration code.

Define	Value	Meaning
CAN_USE_CAN0	0 = Disable 1 = Enable	Enables or disables the use of CAN channel 0.
CAN0_RX_PORT	Options for RX71M, RX64M, RX65N = P33, PD2 RX72T, RX66T = P22, PA1, PA7, PB6, PC6, PE0, PF3	Pins that can be assigned as CAN0 Receive pins. Availability depends on device package
CAN0_TX_PORT	Options for RX71M, RX64M, RX65N = P32, PD1 RX72T, RX66T = P23, PA0, PA6, PB5, PC5, PD7, PF2	Pins that can be assigned as CAN0 Transmit pins. Availability depends on device package
CAN_USE_CAN0_STANDBY_ENABLE_PINS	0 = Disable 1 = Enable	Enables or disables the use of CAN channel 0 standby and enable pins.
CAN0_TRX_STB_PORT	Port Number or Letter	GPIO output port value for the Standby signal

CAN0_TRX_STB_PIN	Port pin#	GPIO output pin number for the Standby signal
CAN0_TRX_STB_LVL	0 = active low 1 = active high	Standby signal active level
CAN0_TRX_ENABLE_PORT	Port Number or Letter	GPIO output port value for the Enable signal
CAN0_TRX_ENABLE_PIN	Port pin#	GPIO output pin number for the Enable signal
CAN0_TRX_ENABLE_LVL	0 = active low 1 = active high	Enable signal active level
CAN_USE_CAN1	0 = Disable 1 = Enable	Enables or disables the use of CAN channel 1.
CAN1_RX_PORT	Options for RX71M, RX64M, RX65N = P15, P55	Pins that can be assigned as CAN1 Receive pins.
CAN1_TX_PORT	Options for RX71M, RX64M, RX65N = P14, P54	Pins that can be assigned as CAN1 Transmit pins.
CAN_USE_CAN1_STANDBY_ENABLE_PINS	0 = Disable 1 = Enable	Enables or disables the use of CAN channel 1 standby and enable pins.
CAN1_TRX_STB_PORT	Port Number or Letter	GPIO output port value for the Standby signal
CAN1_TRX_STB_PIN	Port pin#	GPIO output pin number for the Standby signal
CAN1_TRX_STB_LVL	0 = active low 1 = active high	Standby signal active level
CAN1_TRX_ENABLE_PORT	Port Number or Letter	GPIO output port value for the Enable signal
CAN1_TRX_ENABLE_PIN	Port pin#	GPIO output pin number for the Enable signal
CAN1_TRX_ENABLE_LVL	0 = active low 1 = active high	Enable signal active level
CAN_USE_CAN2	0 = Disable 1 = Enable	Enables or disables the use of CAN channel 2.
CAN2_RX_PORT	RX71M, RX64M = P67	Pins that can be assigned as CAN2 Receive pins.
CAN2_TX_PORT	RX71M, RX64M = P66	Pins that can be assigned as CAN2 Transmit pins.
CAN_USE_CAN2_STANDBY_ENABLE_PINS	0 = Disable 1 = Enable	Enables or disables the use of CAN channel 2 standby and enable pins.
CAN2_TRX_STB_PORT	Port Number or Letter	GPIO output port value for the Standby signal
CAN2_TRX_STB_PIN	Port pin#	GPIO output pin number for the Standby signal
CAN2_TRX_STB_LVL	0 = active low 1 = active high	Standby signal active level
CAN2_TRX_ENABLE_PORT	Port Number or Letter	GPIO output port value for the Enable signal
CAN2_TRX_ENABLE_PIN	Port pin#	GPIO output pin number for the Enable signal
CAN2_TRX_ENABLE_LVL	0 = active low 1 = active high	Enable signal active level

2.8.4 Bitrate Settings

See API description for *R_CAN_SetBitrate*, and file *r_can_rx_config.h*.

Define	Value	Meaning
CAN_BRP	See HW manual	Baud rate prescaler division ratio
CAN_TSEG1	4 - 16	Time Segment 1 Control
CAN_TSEG2	2 - 8	Time Segment 2 Control
CAN_SJW	1 - 4	Resynchronization Jump Width Control

2.8.5 Max Register Poll Time

Maximum number of loops to poll a CAN register bit for expected value. If you are using polled mode, and If you wish to wait a certain time to check that a mailbox has received a frame, increase this value. This can be set to a very low value, but do not set to zero or the mailbox may not be checked at all.

Define	Value	Meaning
MAX_CANREG_POLLCYCLES	Integer value range > 0	Valid only for polling mode. Max loops to poll a CAN register bit for expected value. This can be set to a very low value, but do not set to zero or the mailbox may not be checked at all.

2.9 Code Size

The code size is based on optimization level 2 for size using the Renesas CCRX toolchain 3.01, GCC for Renesas RX 4.8.4 and IAR Embedded Workbench for Renesas RX 4.12.1. The ROM (code, constants, and preinitialized data) and RAM (preinitialized data, uninitialized data) sizes are determined by the build-time configuration options set in the module configuration header reference file for the device.

Build Settings	Area	Size (byte)		
		CCRX	GCC	IAR
Polled mode, Only channel 0 enabled CAN0 Standby/Enable pins not used	ROM	2497	5044	3413
Interrupt mode, Only channel 0 enabled CAN0 Standby/Enable pins not used	ROM	2940	5612	3973
Interrupt mode, 3 channels enabled All CAN Standby/Enable pins enabled	ROM	2982	5780	3983
All	RAM	36	36	36

2.10 Adding the CAN FIT Module to Your Project

This module must be added to each project in which it is used. Renesas recommends using “Smart Configurator” described in (1) or (3). However, “Smart Configurator” only supports some RX devices. Please use the methods of (2) or (4) for unsupported RX devices.

(1) Adding the FIT module to your project using “Smart Configurator” in e^2 studio.
By using the “Smart Configurator” in e^2 studio, the FIT module is automatically added to your project. Refer to “Renesas e^2 studio Smart Configurator User Guide (R20AN0451)” for details.

(2) Adding the FIT module to your project using “FIT Configurator” in e^2 studio.
By using the “FIT Configurator” in e^2 studio, the FIT module is automatically added to your project. Refer to “Adding Firmware Integration Technology Modules to Projects (R01AN1723)” for details.

(3) Adding the FIT module to your project using “Smart Configurator” on CS+
By using the “Smart Configurator Standalone version” in CS+, the FIT module is automatically added to your project. Refer to “Renesas e^2 studio Smart Configurator User Guide (R20AN0451)” for details.

(4) Adding the FIT module to your project in CS+
In CS+, please manually add the FIT module to your project. Refer to “Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)” for details.

3. 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.

CAN configuration and communication is accomplished via the CAN SFR 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 or off line depending on its state. After a recovery is discovered, the application should restart.

3.1 Summary

The following functions are included in this design:

Function Name	Description
R_CAN_Create()	Initializes CAN peripheral
R_CAN_PortSet()	Configures the MCU and transceiver port pins
R_CAN_Control()	Set CAN operating modes
R_CAN_SetBitrate()	Set the CAN bitrate (communication speed)
R_CAN_TxSet() and R_CAN_TxSetXid()	Set up a mailbox to transmit
R_CAN_Tx()	Starts message transmission onto the CAN bus
R_CAN_TxCheck()	Check for successful data frame transmission
R_CAN_TxStopMsg()	Stop a mailbox that has been asked to transmit a frame
R_CAN_RxSet() and R_CAN_RxSetXid()	Set up a mailbox to receive
R_CAN_RxPoll()	Checks if a mailbox has received a message
R_CAN_RxRead()	Read the CAN data frame content from a mailbox
R_CAN_RxSetMask()	Sets the CAN ID Acceptance Masks
R_CAN_CheckErr()	Check for bus errors

3.2 Return Codes

API Return Codes	Description
R_CAN_OK	Action completed successfully.
R_CAN_NOT_OK	Action did not complete successfully. Usually a more specific return code is used
R_CAN_SW_BAD_MBX	Bad mailbox number.
R_CAN_BAD_CH_NR	The channel number does not exist.
R_CAN_BAD_ACTION_TYPE	No such action type exists for this function.
R_CAN_MSGLOST	Message was overwritten or lost
R_CAN_NO_SENTDATA	No message was sent.
R_CAN_RXPOLL_TMO	Polling for received message timed out.
R_CAN_SW_WAKEUP_ERR	The CAN peripheral did not wake up from Sleep mode.
R_CAN_SW_SLEEP_ERR	The CAN peripheral did not enter Sleep mode
R_CAN_SW_HALT_ERR	The CAN peripheral did not enter Halt mode.
R_CAN_SW_RST_ERR	The CAN peripheral did not enter Reset mode.
R_CAN_SW_TSRC_ERR	Time Stamp error
R_CAN_SW_SET_TX_TMO	Waiting for previous transmission to finish timed out.
R_CAN_SW_SET_RX_TMO	Waiting for previous reception to complete timed out.
R_CAN_SW_ABORT_ERR	Wait for abort timed out.
R_CAN_MODULE_STOP_ERR	Whole CAN peripheral is in stop state (low power)

CAN BUS State Codes	Description
R_CAN_STATUS_ERROR_ACTIVE	Node status is normal.
R_CAN_STATUS_ERROR_PASSIVE	Node has sent at least 127 Error frames for either the Transmit Error Counter, or the Receive Error Counter
R_CAN_STATUS_BUSOFF	Node’s Transmit Error Counter has surpassed 255 due to the node’s failure to transmit correctly

3.3 R_CAN_Create

Initializes CAN peripheral - Sets user communication callback functions, configures CAN interrupts, sets bitrate, mailbox defaults, and enters CAN Operation Mode

This function sets the CAN interrupt levels and user callbacks. This function will also call R_CAN_SetBitrate() and sets the mask to default: not mask any frames.

Format

```
uint32_t R_CAN_Create(const uint32_t ch_nr,
                      void (*tx_cb_func)(void),
                      void (*rx_cb_func)(void),
                      void (*err_cb_func)(void));
```

Parameters

ch_nr

CAN channel to use (0-2 MCU dependent).

tx_cb_func

The name of a function in your application which will be called by the CAN driver when a mailbox has finished *transmitting*. If you are using polled mode, or do not want a callback for interrupt mode for some reason, specify NULL.

rx_cb_func

The name of a function in your application which will be called by the CAN driver when a mailbox has finished *receiving*. If you are using polled mode, or do not want a callback for interrupt mode for some reason, specify NULL.

err_cb_func

The name of a function in your application which will be called by the CAN driver when there is a *CAN error*. If you are using polled mode, or do not want a callback for interrupt mode for some reason, specify NULL.

Return Values

R_CAN_OK	*Action completed successfully.*
R_CAN_SW_BAD_MBX	*Bad mailbox number.*
R_CAN_BAD_CH_NR	*The channel number does not exist.*
R_CAN_SW_RST_ERR	*The CAN peripheral did not enter Reset mode.*
R_CAN_MODULE_STOP_ERR	*Whole CAN peripheral is in stop state (low power). Perhaps the PRCR register was not used to unlock the module stop register.*

See also *R_CAN_Control()* return values.

Properties

Prototyped in *r_can_rx_if.h*
Implemented in *r_can_rx.c*

Description

This function wakes the peripheral from CAN Sleep mode and puts it in CAN Reset mode. It configures the mailboxes with these default settings:

Overwrite an unread mailbox data when new frames arrive

Sets the device to use ID priority (normal CAN behavior, not the optional mailbox number priority).

Sets all mailboxes' masks invalid.

R_CAN_Create calls the R_CAN_SetBitrate function and configures CAN interrupts if USE_CAN_POLL is commented in *r_can_rx_config.h*.

Before returning, it clears all mailboxes, sets the peripheral into Operation mode, and clears any errors.

Example

```
#if USE_CAN_POLL
      api_status = R_CAN_Create(g_can_channel, NULL, NULL, NULL);
#else
      /* Using interrupts. */
      api_status = R_CAN_Create(g_can_channel, my_can_tx0_callback,
my_can_rx0_callback, my_can_err0_callback);
#endif
```

3.4 R_CAN_PortSet

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, and this function must then be modified. The function is also used to enter the CAN port test modes, such as Listen Only.

Format

```
uint32_t    R_CAN_PortSet(const uint32_t ch_nr, const uint32_t action_type);
```

Parameters

ch_nr

CAN channel to use (0-2 MCU dependent).

action_type

Port actions:

ENABLE	Enable the CAN port pins and the CAN transceiver.
DISABLE	Disable the CAN port pins and the CAN transceiver.
CANPORT_TEST_LISTEN_ONLY	Set to Listen Only mode. No ACKs or Error frames are sent.
CANPORT_TEST_0_EXT_LOOPBACK	Use external bus and loopback. Useful for initial debug. See separate test section.
CANPORT_TEST_1_INT_LOOPBACK	Only internal mailbox communication. Useful for initial debug. See separate test section.
CANPORT_RETURN_TO_NORMAL	Return to normal port usage.

Return Values

R_CAN_OK	*Action completed successfully.*
R_CAN_SW_BAD_MBX	*Bad mailbox number.*
R_CAN_BAD_CH_NR	*The channel number does not exist.*
R_CAN_BAD_ACTION_TYPE	*No such action type exists for this function.*
R_CAN_SW_HALT_ERR	*The CAN peripheral did not enter Halt mode.*
R_CAN_SW_RST_ERR	*The CAN peripheral did not enter Reset mode.*

See also R_CAN_Control() return values.

Properties

Prototyped in *r_can_rx_if.h*
Implemented in *r_can_rx.c*

Description

Unless Internal Loopback mode is used (for initial test and debug) make sure this function is called after any board default port set up function is used (e.g. ‘hwsetup’).

Observe that a stray output high/low on an MCU CAN port pin that was set by some other (default) board setup code could affect the bus negatively. You may discover 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. Such code should be removed, or else, for a brief period of time, the ports may be output low/high and disrupt the CAN bus voltage level.

You may have to change/add transceiver port pins according to your transceiver.

Example

```
/* Normal CAN bus usage. */
R_CAN_PortSet(0, ENABLE);
```

3.5 R_CAN_Control

Set CAN operating modes. Controls transition to CAN operating modes determined by the CAN Control register. For example, the Halt mode should be used to later configure a receive mailbox.

Format

```
uint32_t R_CAN_Control(const uint32_t ch_nr, const uint32_t action_type);
```

Parameters

ch_nr

CAN channel to use (0-2 MCU dependent).

action_type

Peripheral actions:

EXITSLEEP_CANMODE	Exit CAN Sleep mode, the default state when the peripheral starts up.
ENTERSLEEP_CANMODE	Enter CAN Sleep mode to save power.
RESET_CANMODE	Put the CAN peripheral into Reset mode.
HALT_CANMODE	Put the CAN peripheral into Halt mode. CAN peripheral is still connected to the bus, but stops communicating.
OPERATE_CANMODE	Put the CAN peripheral into normal Operation mode.

Return Values

R_CAN_OK	*Action completed successfully.*
R_CAN_SW_BAD_MBX	*Bad mailbox number.*
R_CAN_BAD_CH_NR	*The channel number does not exist.*
R_CAN_BAD_ACTION_TYPE	*No such action type exists for this function.*
R_CAN_SW_WAKEUP_ERR	*The CAN peripheral did not wake up from Sleep mode.*
R_CAN_SW_SLEEP_ERR	*The CAN peripheral did not enter Sleep mode.*
R_CAN_SW_HALT_ERR	*The CAN peripheral did not enter Halt mode.*
R_CAN_SW_RST_ERR	*The CAN peripheral did not enter Reset mode.*

See also R_CAN_PortSet() return values.

Properties

Prototyped in r_can_rx_if.h

Description

Other than calling this API to enter Halt mode, CAN mode transitions are called via the other API functions automatically. For example, the default mode when starting up is CAN Sleep mode. Use the API to switch to other operating modes, for example first ‘Exit Sleep’ followed by ‘Reset‘ to initialize the CAN registers for bitrate and interrupts, then enter ‘Halt’ mode to configure mailboxes.

Example

```
/* Normal CAN bus usage. */
result = R_CAN_Control(0, OPERATE_CANMODE); //Check that result is = R_CAN_OK.
```

3.6 R_CAN_SetBitrate

Set the CAN bitrate (communication speed). The baud rate and bit timing must always be set during the configuration process. It can be changed later if reset mode is entered.

Format

```
void R_CAN_SetBitrate(const uint32_t ch_nr);
```

Parameters

ch_nr

CAN channel to use (0-2 MCU dependent).

Return Values

None

Properties

Prototyped in r_can_rx_if.h

Description

Setting the baud rate or data speed on the CAN bus 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.

The bitrate is set via macros inside *r_can_rx_config.h*. Some calculations need to be done to set these macros. First some explanations. The CAN system clock, f_{canclk}, is the internal clock period of the CAN peripheral. This CAN system clock is determined by the CAN Baud Rate Prescaler value and the peripheral bus clock. One Time Quantum is equal to the period of the CAN clock.

One CAN bus bit-time is an integer sum of a number of Time Quanta, Tq. Each bitrate register is then given a certain number of Tq of the total number of Time Quanta that make up one CAN bit period, or Tqtot.

Formulas to calculate the bitrate register settings.

PCLK is the peripheral clock frequency, PCLKB.

fcan = PCLK or EXTAL

The prescaler scales the CAN peripheral clock down with a factor.

fcanclk = fcan/prescaler

One Time Quantum is one clock period of the CAN clock.

Tq =1/fcanclk

Tqtot is the total number of CAN peripheral clock cycles during one CAN bit time and is by the peripheral built by the sum of the “time segments” and “SS” which is always 1. In the code, Tqtot is shown to be

BSP_CFG_XTAL_HZ * BSP_CFG_PLL_MUL) / (CAN_BRP * BITRATE * BSP_CFG_PCKB_DIV)

Set these macros so that a Tqtot is found which is not larger than accepted by the CAN registers. See the HW-manual’s table of examples for bitrate settings.

Another restriction is:

Tqtot = TSEG1 + TSEG2 + SS (TSEG1 must be > TSEG2)

SS is always 1. SJW is often given by the bus administrator. Select 1 <= SJW <= 4.

See *r_can_rx_config.h* for details.

You can also use this Python code as an aid to change bit rate.

```
# Python 3.5.1. Simple python code to help calculate bitrate register settings.
If you don't have Python, just follow the code, you should see how to calculate
register settings by hand.

from fractions import Fraction
BITRATE = 500000

# Try a BRP. If TQTOT is too large for register settings, increase.
CAN_BRP = 4

# Limit on what is tolerated if TQTOT is not a whole integer.
# If it is not, it is impossible to get an exact baudrate.
# Value is not tested.
MAX_TQ_FRACTION_DEV = 0.1
XTAL_HZ = 12000000
PLL_MUL = 4             # Depending on part, these may not exist, and so be = 1.
PCKB_DIV = 2
TQTOT = (XTAL_HZ * PLL_MUL)/(CAN_BRP * BITRATE * PCKB_DIV)
print ("TQTOT is", round(TQTOT, 2), "=> Set TSEG1 larger than TSEG2, and SJW to
1, so that the sum of these is TQTOT.")
print ("=============")
```

Example

```
/* Set bitrate as defined by r_can_rx_config.h. */
R_CAN_SetBitrate(0);
```

3.7 R_CAN_TxSet and R_CAN_TxSetXid

Set up a mailbox to transmit. R_CAN_TxSet will write to a mailbox the specified ID, data length and data frame payload, then set the mailbox to transmit mode and send a frame onto the bus by calling R_CAN_Tx().

R_CAN_TxSetXid does the same, except if this function is used, the ID will be a 29-bit ID.

Format

```
uint32_t    R_CAN_TxSet(const uint32_t ch_nr, const uint32_t mbox_nr,
                        const can_frame_t* frame_p, const uint32_t frame_type);

uint32_t    R_CAN_TxSetXid(const uint32_t ch_nr, const uint32_t mbox_nr,
                           can_frame_t* frame_p, const uint32_t frame_type);
```

Parameters

ch_nr

CAN channel to use (0-2 MCU dependent).

mbox_nr

Mailbox to use.

frame_p

Pointer to a data frame structure in memory. It is an address to the data structure containing the ID, DLC and data that constitute the dataframe the mailbox will transmit.

frame_type

DATA_FRAME	Send a normal data frame.
REMOTE_FRAME	Send a remote data frame request.

Return Values

R_CAN_OK	*The mailbox was set up for transmission.*
R_CAN_SW_BAD_MBX	*Bad mailbox number.*
R_CAN_BAD_CH_NR	*The channel number does not exist.*
R_CAN_BAD_ACTION_TYPE	*No such action type exists for this function.*

Properties

Prototyped in r_can_rx_if.h

Description

This function first waits for any previous transmission of the specified mailbox to complete. It then interrupt disables the mailbox temporarily when setting up the mailbox: Sets the ID value for the mailbox, the Data Length Code indicated by frame_p, selects dataframe or remote frame request and finally copies the data frame payload bytes (0-7) into the mailbox. The mailbox is interrupt enabled again unless USE_CAN_POLL was defined. Finally R_CAN_Tx is called to deliver the message.

Example

```
#define MY_TX_SLOT  (7)
can_frame_t         my_tx_dataframe;

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

/* Send my frame. */
api_status = R_CAN_TxSet(0, MY_TX_SLOT, &my_tx_dataframe, DATA_FRAME);
```

3.8 R_CAN_Tx

Starts actual message transmission onto the CAN bus. This API will wait until the mailbox finishes handling a prior frame, then set the mailbox to transmit mode.

Format

```
uint32_t  R_CAN_Tx(const uint32_t ch_nr, const uint32_t mbox_nr);
```

Parameters

ch_nr
CAN channel to use (0-2 MCU dependent).
mbox_nr
Which CAN mailbox to use. (0-32)

Return Values

R_CAN_OK	*Set up to transmit was performed successfully.*
R_CAN_SW_BAD_MBX	*Bad mailbox number.*
R_CAN_BAD_CH_NR	*The channel number does not exist.*
R_CAN_SW_SET_TX_TMO	*Waiting for previous transmission to finish timed out.*
R_CAN_SW_SET_RX_TMO	*Waiting for previous reception to complete timed out.*

Properties

Prototyped in r_can_rx_if.h

Description

R_CAN_TxSet must have been called at least once for this mailbox after system start to set up the mailbox content, as this function only tells the mailbox to send its content.

Example

```
#define MY_TX_SLOT    (7)

/* Send mailbox content. This mailbox is presumed to have been set up to send
   some time in the past. */
R_CAN_Tx(0, MY_TX_SLOT);
```

3.9 R_CAN_TxCheck

Check for successful data frame transmission. Use to check a mailbox for a successful data frame transmission.

Format

```
uint32_t  R_CAN_TxCheck(const uint32_t ch_nr, const uint32_t mbox_nr);
```

Parameters

ch_nr

CAN channel to use (0-2 MCU dependent).

mbox_nr

Which CAN mailbox to use. (0-32)

Return Values

R_CAN_OK	*Transmission was completed successfully.*
R_CAN_SW_BAD_MBX	*Bad mailbox number.*
R_CAN_BAD_CH_NR	*The channel number does not exist.*
R_CAN_MSGLOST	*Message was overwritten or lost.*
R_CAN_NO_SENTDATA	*No message was sent.*

Properties

Prototyped in r_can_rx_if.h

Description

This function is only needed if an application needs to verify that a message has been transmitted for example so that it can progress a state machine, *or if messages are sent back-to-back*. With CAN's level of transport control built into the silicon, it can reasonably be assumed that once a mailbox has been asked to send with the API that the message will indeed be sent. Safest if of course to use this function after a transmission.

Example

```
/* TRANSMITTED a particular frame? */
api_status = R_CAN_TxCheck(0, CANBOX_TX);

if (api_status == R_CAN_OK)
{
      message_x_sent_flag = TRUE;   // Notify main application.
}
```

3.10 R_CAN_TxStopMsg

Stop a mailbox that has been asked to transmit a frame

Format

```
uint32_t R_CAN_TxStopMsg(const uint32_t ch_nr, const uint32_t mbox_nr);
```

Parameters

ch_nr
CAN channel to use (0-2 MCU dependent).
mbox_nr
Which CAN mailbox to use. (0-32)

Return Values

R_CAN_OK	*Action completed successfully.*
R_CAN_SW_BAD_MBX	*Bad mailbox number.*
R_CAN_BAD_CH_NR	*The channel number does not exist.*
R_CAN_SW_ABORT_ERR	*Waiting for an abort timed out.*

Properties

Prototyped in r_can_rx_if.h

Description

This function clears the mailbox control flags so that a transmission is stopped (TrmReq is set to 0.) A software counter then waits for an abort for a maximum period of time.

If the message was not stopped, R_CAN_SW_ABORT_ERR is returned. Note that the cause of this could be that the message was already sent.

Example

```
R_CAN_TxStopMsg(0, MY_TX_SLOT);
```

3.11 R_CAN_RxSet and R_CAN_RxSetXid

Set up a mailbox to receive.

R_CAN_RxSet: The API sets up a given mailbox to receive data frames with the given CAN 11-bit ID. Incoming data frames with the same ID will be stored in the mailbox.

R_CAN_RxSetXid: Does the same, except the ID will be a 29-bit ID.

Format

```
uint32_t    R_CAN_RxSet(const uint32_t ch_nr, const uint32_t mbox_nr,
                        const uint32_t sid, const uint32_t frame_type);

uint32_t    R_CAN_RxSetXid(const uint32_t ch_nr, const uint32_t mbox_nr,
                           uint32_t xid, const uint32_t frame_type);
```

Parameters

ch_nr
CAN channel to use (0-2 MCU dependent).
mbox_nr
Which CAN mailbox to use. (0-32)
sid
xid
The CAN ID which the mailbox should receive.
frame_type

DATA_FRAME	Send a normal data frame.
REMOTE_FRAME	Send a remote data frame request.

Return Values

R_CAN_OK	*Action completed successfully.*
R_CAN_SW_BAD_MBX	*Bad mailbox number.*
R_CAN_BAD_CH_NR	*The channel number does not exist.*
R_CAN_SW_SET_TX_TMO	*Waiting for previous transmission to finish timed out.*
R_CAN_SW_SET_RX_TMO	*Waiting for previous reception to complete timed out.*

Properties

Prototyped in r_can_rx_if.h

Description

The function will first wait for any previous transmission/reception to complete, then temporarily interrupt disable the mailbox. It sets the mailbox to the given standard ID value, and whether to receive normal CAN dataframes or remote frame requests.

Example

```
#define MY_RX_SLOT       (8)
#define SID_FAN_SPEED    0x10

R_CAN_RxSet(0, MY_RX_SLOT, SID_FAN_SPEED, DATA_FRAME);
```

3.12 R_CAN_RxPoll

Checks if a mailbox has received a message

Format

```
uint32_t  R_CAN_RxPoll(const uint32_t ch_nr, const uint32_t mbox_nr);
```

Parameters

ch_nr

CAN channel to use (0-2 MCU dependent).

mbox_nr

Which CAN mailbox to check (0-32).

Return Values

R_CAN_OK	*There is a message waiting.*
R_CAN_NOT_OK	*No message waiting or pending.*
R_CAN_RXPOLL_TMO	*Message pending but timed out.*
R_CAN_SW_BAD_MBX	*Bad mailbox number.*
R_CAN_BAD_CH_NR	*The channel number does not exist.*

Properties

Prototyped in r_can_rx_if.h

Description

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:

Polling. Call the API regularly to check for new messages. USE_CAN_POLL must be defined in the CAN configuration file. If there is a message use R_CAN_RxRead to fetch it.

Using the CAN receive interrupt (USE_CAN_POLL not defined): Use this API to check which mailbox received. Then notify the application.

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

Example

See example in R_CAN_RxRead().

3.13 R_CAN_RxRead

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 written to the given structure.

Format

```
uint32_t  R_CAN_RxRead(const uint32_t ch_nr, const uint32_t mbox_nr,
                       can_frame_t* const frame_p);
```

Parameters

ch_nr

CAN channel to use (0-2 MCU dependent).

mbox_nr

Which CAN mailbox to check (0-32).

frame_p

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

R_CAN_OK	*There is a message waiting.*
R_CAN_SW_BAD_MBX	*Bad mailbox number.*
R_CAN_BAD_CH_NR	*The channel number does not exist.*
R_CAN_MSGLOST	*Message was overwritten or lost.*

Properties

Prototyped in r_can_rx_if.h

Description

Use R_CAN_RxPoll() first to check whether the mailbox has received a message.

This function is used to fetch the message from a mailbox, either when using polled mode or from a CAN receive interrupt.

Example

```
#define MY_RX_SLOT    (8)
can_frame_t  my_rx_dataframe;

api_status = R_CAN_RxPoll(0, CANBOX_RX_DIAG);

if (api_status == R_CAN_OK)
{
    R_CAN_RxRead(0, CANBOX_RX_DIAG, &my_rx_dataframe);
}
```

3.14 R_CAN_RxSetMask

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_RxSetMask(const uint32_t ch_nr, const uint32_t mbox_nr,

                     const uint32_t mask_value);
```

Parameters

ch_nr

CAN channel to use (0-2 MCU dependent).

mbox_nr

Which mailbox to mask (0-32). Four mailboxes will be affected within its group.

mask_value

Mask value. (0-0x7FF)

Return Values

None

Properties

Prototyped in r_can_rx_if.h

Description

Receive mailboxes can use a mask to filter out one message, or expand receiving to a range of messages (CAN IDs). The mask enables this using the mailbox group's ID field. There is one mask for mailbox 0-3, one for 4-7, etc. Changing a mask will therefore affect the behavior of adjacent mailboxes.

- Each '0' in the mask means "mask this bit", or "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

Let's say that the range of CAN-IDs that you want to receive in a mailbox is 700-704$_h$. Using standard 11-bit IDs we then have the following Ids in hex and binary:

Hex representation	Bit representation
0x700	011100000000b
0x701	011100000001b
0x702	011100000010b
0x703	011100000011b
0x704	011100000100b

Normally, the mailbox will only accept frames whose ID matches the set receive ID, but if a bit position's MASK is 0, an ID bit of both 0 and 1 will be accepted. If we then want to accept all of above, we set the mask as

011111111000b, or 07F8$_h$.

The CAN receive filter will only look at bit positions b10 (MSB) to b3 (LSB), whether these match the receive ID of the mailbox.

If we then set one of the mailboxes belonging to above mask (they are grouped - four mailboxes per mask) to receive ID 0x700, that mailbox will accept all IDs from 0x700 to 0x707. (Setting the ID to 0x700-0x707 will give the same result.) Because of this, IDs 0x705 to 0x707 must later be ignored 'manually' by the application software.

Fast filtering of messages with Acceptance Filter Support

If you have used a mask to receive a broad range of message IDs, you must filter for the actual desired messages with firmware. To increase the speed of this search one may use the Acceptance Filter Support instead.

The Acceptance Filter Support Unit (ASU) provides a faster search compared to software filtering of messages using a mask (with the R_CAN_RxSetMask API). 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 reading back from the ASU register, the data word is used to search through a table. The data readout has the following parts.
Bit 0-7 = Table 'Address Search Info", ASI, SID10 -3.
Bit 8-15 = "Bit Search Information", BSI, SID0-3 has now been converted to a bit position to enable faster table searches.

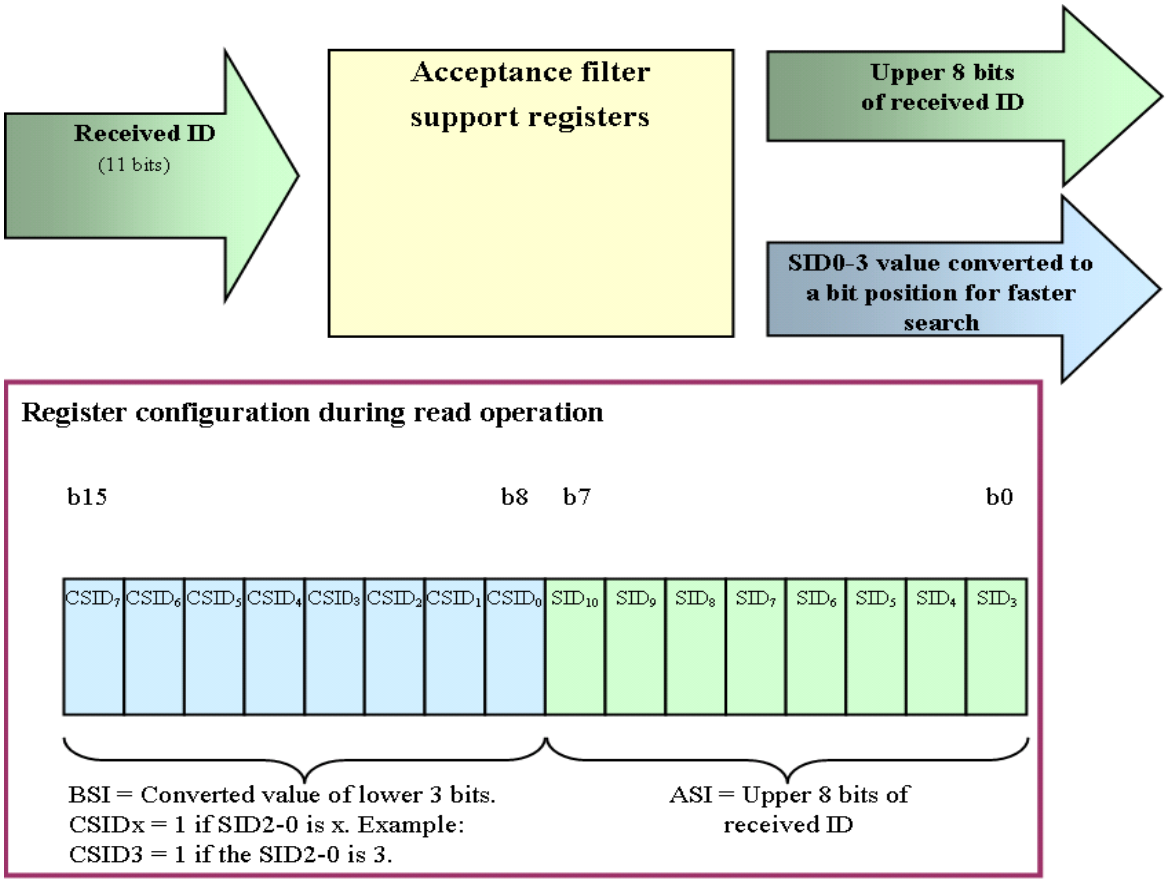

Figure 2. The Acceptance Filter Support Unit (ASU).

When read, the representation of the ID is formatted to enable a fast search through a table. This provides a faster response than a search through a 'normal' array of CAN IDs.

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.

See REJ05B0276 "CAN Application Note" for more information on how to use the ASU.
Download from www.renesas.com

3.15 R_CAN_CheckErr

Check for bus errors. The API checks the CAN status, or Error State, of the CAN peripheral.

Format

```
uint32_t  R_CAN_CheckErr(const uint32_t  ch_nr);
```

Parameters

ch_nr

CAN channel to use (0-2 MCU dependent).

Return Values

R_CAN_BAD_CH_NR	*The channel number does not exist.*
R_CAN_STATE_ERROR_ACTIVE	*CAN bus status is normal.*
R_CAN_STATE_ERROR_PASSIVE	*Node has sent at least 127 Error frames for either the Transmit Error Counter, or the Receive Error Counter.*
R_CAN_STATE_BUSOFF	*Node's Transmit Error Counter has surpassed 255 due to the node's failure to transmit correctly.*

Properties

Prototyped in "r_can_rx_if.h".

Description

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 or off line depending on its state. After a recovery is discovered, the application should restart.

Bus States

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.

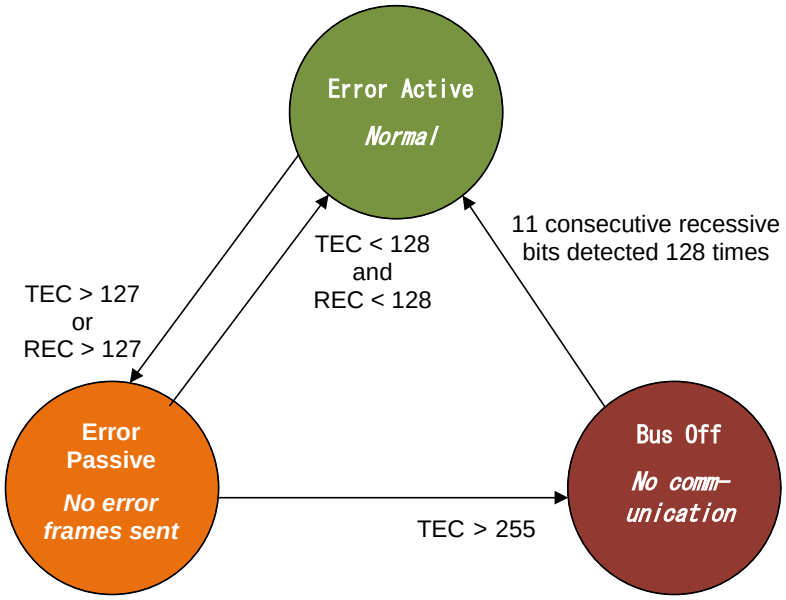

Figure 3. CAN bus error states.

(a) **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.

(b) **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.

(c) **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.

Using CAN Polling

Call the API regularly to check the CAN state for the application, so it does not try to communicate if the node is Bus Off. In the following, it is assumed that *handle_can_bus_state()* is called once every loop of the main application.

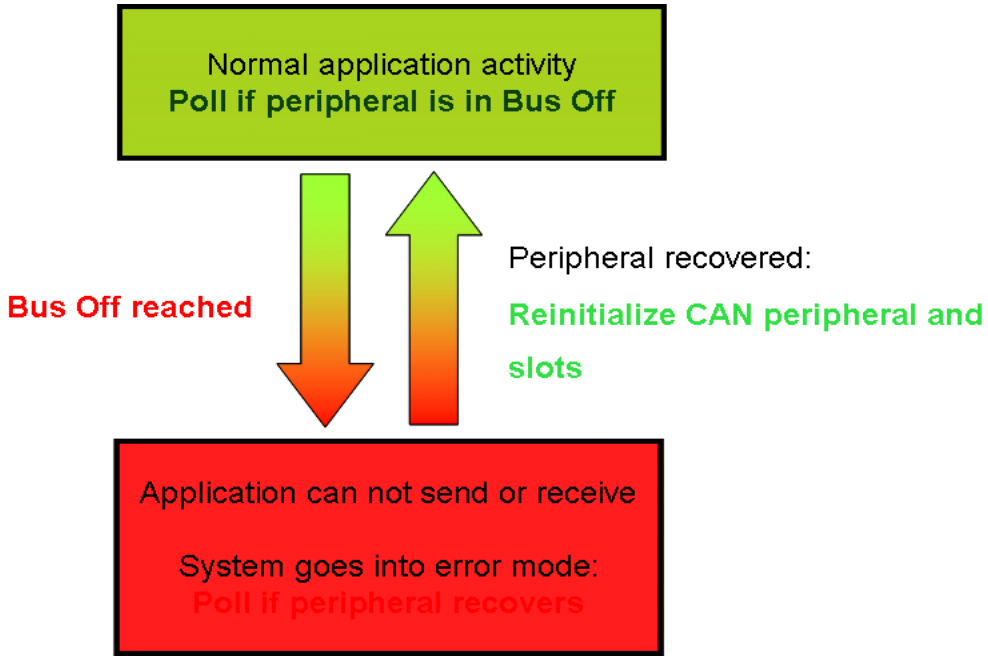

Figure 4. Handling recovery from Bus Off for the application.

The MCU detects recovery of the bus on its own. A node will automatically resume the normal Error Active state again after seeing 11 consecutive recessive bits on the bus 128 times. Note that the time a node spends in Bus Off could be very short, e.g. less than a millisecond.

Poll with the Check Error function once every cycle in the main routine what state the node is in (or use the CAN error interrupt). If the node has reached Bus Off a certain number of times within a certain time period, you may want to send a warning message, light an LED etc.

The minimum action required of a node if Bus Off is reached is shown above. Stop trying to communicate and poll the peripheral with the Check Error function to see when the peripheral has returned to the normal Error Active state. When the node has recovered, it is important to reinitialize the CAN peripheral and the application to make sure the slots are in a known state.

Example

See usage of *handle_can_bus_state*() in *can_api_demo.c*.

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 done 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.

4. Demo Projects

CAN demo projects are complete stand-alone programs. They include function main() that utilizes the module and its dependent modules.

4.1 Adding a Demo to a Workspace

Demo projects are found in the FITDemos subdirectory of the distribution file for this application note. To add a demo project to a workspace, select File>Import>General>Existing Projects into Workspace, then click “Next”. From the Import Projects dialog, choose the “Select archive file” radio button. “Browse” to the FITDemos subdirectory, select the desired demo zip file, then click “Finish”.

The demo CAN application code is in the *../src* directory, namely in files *can_api_demo.c* and *switches.c*.

To run the demo, import the e2studio project archive *r01an2472eu1xxxx_can.zip* into e² studio as explained below.

4.1.1 Import and Debug Project with e² studio

New workspace

Create an empty folder, where you want the workspace.

Start e² studio, and point to above folder when e² studio asks what workspace to open.

Click Workbench icon (bottom right in blue intro-screen).

Continue with next step below.

Existing workspace

Select Import.

Select General => Existing Projects into workspace. ("Create new projects from an archive file or directory.")

If the code is a zipped, previously exported archive:

Browse to the archive zip-file and select it.

If the code is an e² studio project directory with source code (with a .project file):

Browse to the root directory of the project. (The folder containing the “.project” file.) Make sure to check box "Copy project to workspace" if you want the code to be local to the workspace (where the .metadata directory is).

Click "Finish".

You have now imported this project into the workspace. You can go ahead and import other projects into the same workspace.

Run the code

Create a debug session, download and run the code.

4.1.2 Run Demo

Included in the package is a demonstration of using the CAN API, showing how to receive and transmit using the CAN API at 500 kbps. The demo can be run in polled mailbox mode, or with CAN receive and transmit interrupts.

The demo can physically be set up a few different ways:

Program two boards and connect them together over the CAN bus. Swap the CAN ID values TX_CANID_DEMO_INIT and RX_CANID_DEMO_INIT on one of the boards before programming and running the demo.

Use a CAN bus monitor, e.g. SysTec low-cost monitor 3204000, to send and receive frames to/from the demo.

With CANPORT_TEST_1_INT_LOOPBACK used in the R_CAN_PortSet API you can communicate internally, no external bus needed!

Remote frames can also be demonstrated if CAN interrupts are enabled.

Operation

The demo transmits and receives frames with the default CAN-IDs TX_CANID_DEMO_INIT and RX_CANID_DEMO_INIT. The demo starts up by sending NR_STARTUP_TEST_FRAMES test frames back-to-back as fast as possible. This has two purposes. 1) Check the bus link. 2) Demonstrate how messages are sent back-to-back as fast as possible.

User action

Press SW1 to send one CAN frame. To increment the TxID hold SW2 down and press SW3. The actual send command is invoked by the Sw1Func() function. To change RxID hold SW3 down and press SW2. The demo "action" can best be seen inside function *can_int_demo()* or *can_poll_demo()* depending on the setting of USE_CAN_POLL in *r_can_rx_config.h*.

Remote Frames

Besides demonstrating transmit and receive of standard CAN frames, the demo will also send remote frame responses for remote frame requests received by the mailbox at CAN-ID 50h (defined by REMOTE_TEST_ID in *can_api_demo.h*).

Set REMOTE_DEMO_ENABLE to 1 in *can_api_demo.c* to add this feature to the demo.

The demo requires interrupt mode; that is, USE_CAN_POLL set to 0 in the CAN API config-file. Remote requests must come from an outside source, e.g. the CAN monitor mentioned above. This external CAN source must be set to send remote frame requests to CAN-ID 50h.

4.2 The Renesas Debug Console

Enabling trace data from the E1/E20 to the e² studio Debug Console allows you to output data from your application in real-time. This means you have the ability to use printf() statements in C to send trace strings to the standard output. Standard output will in this case be the E1/E20 debug register.

To use this set BSP_CFG_IO_LIB_ENABLE to 1 in *../r_config/r_bsp_config.h.*

The macro should automatically enable code in order to make the Debug Console available, but there are certain actions you must take.

1. Make sure *INIT_IOLIB()* is called. See *resetprog.c*.

The code in *lowlvl.c* should contain functions *charput* and *charget* so that E1/E20 debug registers are used for the lowest level I/O processing. *charput* for example must contain

```
/* Wait for transmit buffer to be empty */
while(0 != (E1_DBG_PORT.DBGSTAT & TXFL0EN));
```

Include <*stdio.h*> in any files where you wish to use printf-statements.

To **any file** where *printf()* is called, add

```
#if BSP_CFG_IO_LIB_ENABLE
   #include <stdio.h>
#endif
```

In e² studio, depending on version, it may be necessary to add the Debug Console window by clicking on both icons “1/0” and “Pin Console” as shown below. **Both must be on so the print buffer in E1/E20 can be emptied and not block code execution.**

Figure 5. Buttons to control the Debug Console.

Press the I/O button for the console in e² studio again if the console seems unresponsive. If nothing is printed, press the Clear icon a few times. (The icon partially concealed by the red border.)

5. Test Modes

There are test modes that may be useful for example during product development. There are two loopback modes “Internal” and “External”, and a Listen only mode.

5.1 Loopback

With loopback modes, the node will itself also receive any messages it sends if a mailbox is configured to receive the same message. This can be useful for testing an application, or self-diagnosis during application debug.

5.1.1 Internal - Test node without CAN bus

Internal Loopback mode, or Self Test mode, allows you to communicate via the CAN mailboxes without connecting to a bus. The node acknowledges its own data with the ACK bit in the data frame. The node also stores its own transmitted messages into a receive mailbox if it was configured for that CAN ID. This is normally not possible.

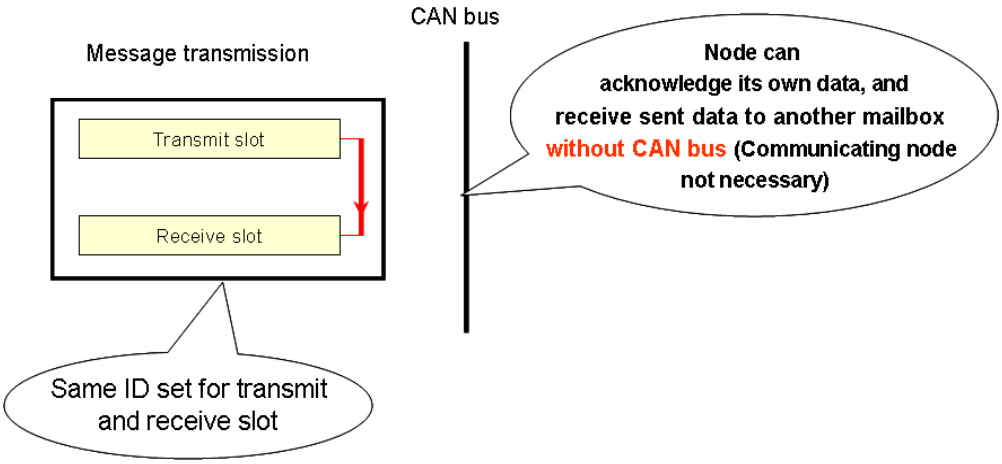

Figure 6.

CAN Internal Loopback mode lets you test the functionality of a node without having a CAN bus connected.

Internal Loopback can be convenient when testing as this mode allows the CAN controller to run without sending CAN errors due to no ACKs received when the node is alone on the bus, it acknowledges transmitted frames itself.

5.1.2 External - Test node on bus

External Loopback is like Internal Loopback with the differences that there must be a CAN bus connected to the node, and that the messages is also transmitted onto the bus. Just like internal loopback, a sent message is acknowledged by the node itself so the node can be alone on the bus. This is an advantage as nodes can be tested standalone.

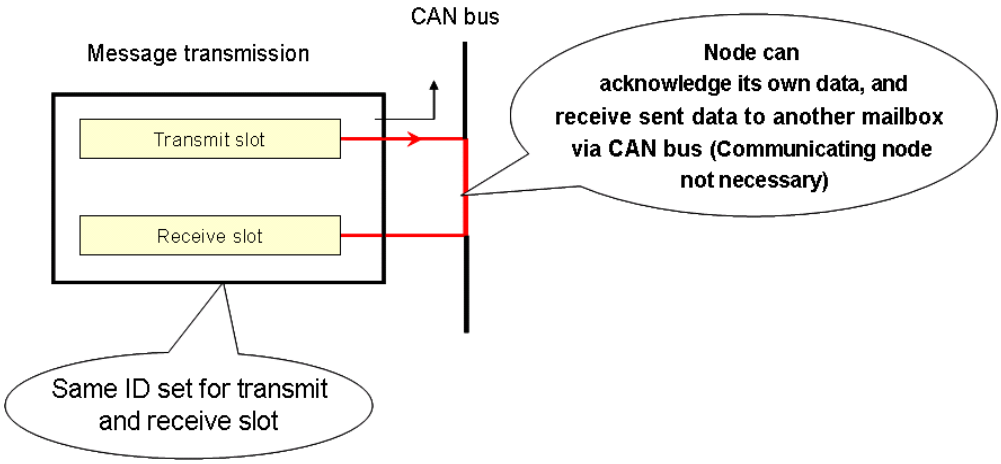

Figure 7. External Loopback.

The message is transmitted onto the CAN bus and can be received back on the same node. This is convenient when testing code and when a node is alone on the bus.

5.2 Listen Only = Bus Monitoring

In Listen Only mode, or Bus Monitoring, the node is quiet. A node in Listen Only mode will not acknowledge messages or send Error frames etc. This enables you to test your node without affecting bus traffic.

Caution:

1. Do not transmit frames from the Listen Only node. That is not a correct behavior, and the CAN module has not been designed for this.
2. If you only have two nodes on the network and one of them is Listen Only, the other node will not get any ACKs and will keep trying to send over and over.
3. Mark entering listen only mode clearly in your code so you remember to disable Listen Only mode again.

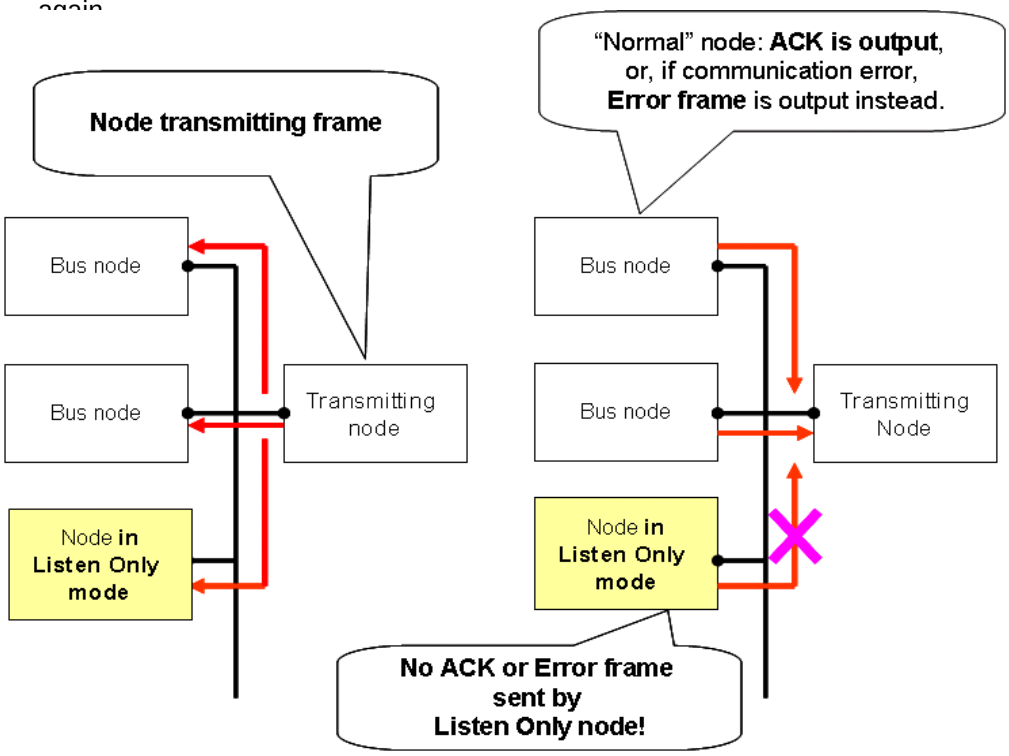

Figure 8.

A node in Listen Only mode will not acknowledge messages 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 for a recently connected node's application to ensure that frames have properly been received before going live.

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.

6. Time Stamp

The timestamp function captures the value of the on-chip time stamp to a mailbox when a message is received. By examining the time stamp you can for example determine the sequence of messages if they are spread out over multiple receive mailboxes. Time stamp reading is not done by the API, so you will have to poll the mailbox, and if the return value is R_CAN_OK (a message waiting) you can then go in and read the timestamp.

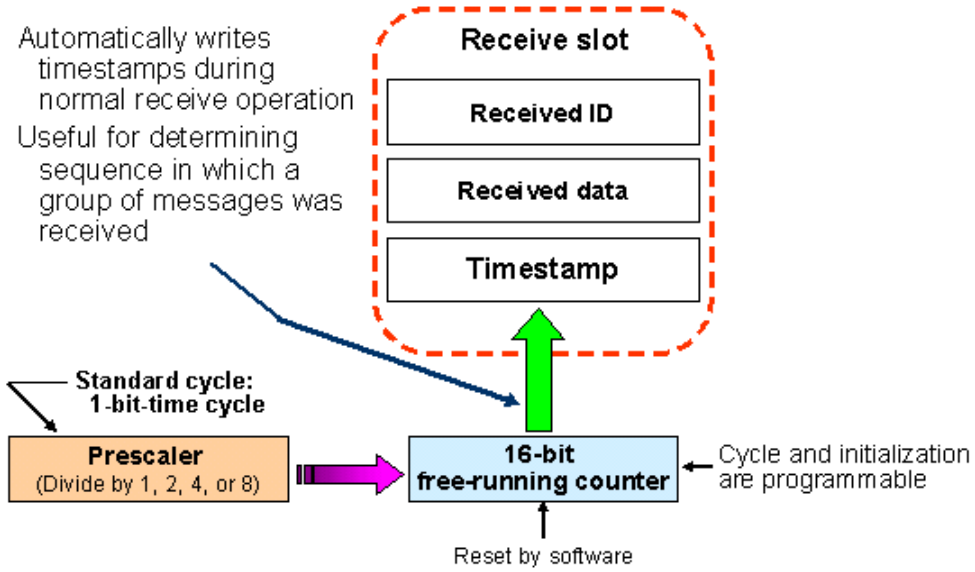

Figure 9. CAN Timestamp is available in each mailbox.

7. CAN Sleep Mode

The default mode after an MCU reset is CAN Sleep mode. Use the API to switch to other operating modes, see the R_CAN_Control API. Entering the CAN Sleep mode instantly stops the clock supply to the module and thereby reduces power dissipation. All registers remain unchanged when the CAN module enters CAN sleep mode.

8. CAN FIFO

CAN FIFO buffering is available in the RX MCUs that have CAN hardware, however FIFO mode is not supported in this software.

9. Appendices

9.1 Confirmed Operation Environment

This section describes confirmed operation environment for the CAN FIT module.

Table 9.1 Confirmed Operation Environment (Rev.3.10)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 7.5.0 IAR Embedded Workbench for Renesas RX 4.12.1
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.01.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 4.8.4.201902 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99 Linker option: The following user defined option should be added to the default settings of the integrated development environment, if “Optimize size (-Os)” is used: -Wl,--no-gc-sections This is to work around a GCC linker issue whereby the linker erroneously discard interrupt functions declared in FIT peripheral module
	IAR C/C++ Compiler for Renesas RX version 4.12.1 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.3.10
Board used	Renesas Starter Kit+ for RX72M (product No.: RTK5572Mxxxxxxxxxx)

Table 9.2 Confirmed Operation Environment (Rev.3.00)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 7.3.0 IAR Embedded Workbench for Renesas RX 4.10.1
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.01.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 4.8.4.201803 Compiler option: The following option is added to the default settings of the integrated development environment. -std=gnu99 Linker option: The following user defined option should be added to the default settings of the integrated development environment, if “Optimize size (-Os)” is used: -Wl,--no-gc-sections This is to work around a GCC linker issue whereby the linker erroneously discard interrupt functions declared in FIT peripheral module
	IAR C/C++ Compiler for Renesas RX version 4.10.1 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.3.00
Board used	Renesas Starter Kit+ for RX65N-2M (product No.: RTK50565Nxxxxxxxxx)

9.2 Troubleshooting

(1) Q: I have added the FIT module to the project and built it. Then I got the error: Could not open source file “platform.h”.

A: The FIT module may not be added to the project properly. Check if the method for adding FIT modules is correct with the following documents:

- Using CS+:

  Application note “Adding Firmware Integration Technology Modules to CS+ Projects (R01AN1826)”

- Using e^2 studio:

  Application note “Adding Firmware Integration Technology Modules to Projects (R01AN1723)”

When using this FIT module, the board support package FIT module (BSP module) must also be added to the project. Refer to the application note “Board Support Package Module Using Firmware Integration Technology (R01AN1685)”.

(2) Q: I have added the FIT module to the project and built it. Then I got the error: This MCU is not supported by the current r_can_rx module.

A: The FIT module you added may not support the target device chosen in your project. Check the supported devices of added FIT modules.

(3) Q: I have added the FIT module to the project and built it. Then I got an error for when the configuration setting is wrong.

A: The setting in the file “r_can_rx_config.h” may be wrong. Check the file “r_can_rx_config.h”. If there is a wrong setting, set the correct value for that. Refer to 2.8 Configuration for details.

Related Technical Updates

This module reflects the content of the following technical updates.

TN-RX*-A151A/E

Revision History

		Description	
Rev.	**Date**	**Page**	**Summary**
1.00	Nov 17, 2014	—	First release. 64M.
2.00	Feb 20, 2015	—	Added 71M.
2.01	Jul 01, 2015	21	- Introduction slightly revised. - Comments section under R_CAN_RxSetMask() slightly revised. - Source code: R_CAN_TxCheck() and R_CAN_TxStopMsg() in r_can_rx.c modified because of RX64M/71M UM 43.2.8 note "Bits SENTDATA and TRMREQ cannot be set to 0 simultaneously.”
2.02	Oct 30, 2015	5.3, 5.4, p9, p15.	Updates to code packaging (only) for FIT. R_CAN_Create(): Added arguments for interrupt callbacks. R_CAN_Tx(): Rephrased R_CAN_OK return case.
2.10	Mar 3, 2016	—	- 65N added. - Set IDE bit according to requested frame type for mixed ID mode only. - Change in R_CAN_RxRead() Mixed mode. - R_CAN_Control(). Cases EXITSLEEP_CANMODE and ENTERSLEEP_CANMODE, OPERATE_CANMODE.
2.11	Jan 30, 2017	All	Added 65N-2MB. Application note: - Comments from Japan review. - Added chapter “Using the Renesas Debug Console”. - R_CAN_SetBitrate() section rewritten and expanded. Code: - Added user level CAN error diagnostics code to can_api_demo.c. This is to aid user in bus problem diagnostics during development & test. This code is macro enabled by setting ERROR_DIAG to 1. - Removed all USE_LCD code. Using debug console (printf) instead. Added corresponding trace code to demo. - Function names changed_to_this_style(), except for API for legacy purpose. - Cleaned up code in Handle_can_bus_state(). - Fig. 4 text "TEC or REC > 127" changed to "TEC < 128 and REC <128". - 7.2. For Remote Frames, the value for USE_CAN_POLL corrected to "0".
2.12	Aug 15, 2017	22 6	- Text in Comments section of R_CAN_RxSetMask() adjusted. - ICU.GRPBE0.BIT changed for channels 1 and 2 in CAN_ERS_ISR(). (All were set to channel 0.) Text change in description of R_CAN_Create(). Removed reference to R_CAN_RxSetMask () and R_CAN_PortSet () calls.
2.13	Oct 26, 2018	1 All	Added RX66T as Target Device. Changed title to add new device.
2.14	Nov 16, 2018	All	Major revision of application note to updated template. All sections affected.
2.15	Jan 10, 2019	1	Revision changed. Added RX72T device.

Rev.	Date	Description Page	Summary
3.00	May.20.2019	—	Supported the following compilers: - GCC for Renesas RX - IAR C/C++ Compiler for Renesas RX
		6	2.3 Software Requirements Requires r_bsp v5.20 or higher
		9	Updated the section of 2.9 Code Size
		36	Table 9.1 Confirmed Operation Environment (Rev. 3.00) : Updated.
		37	Added 9.2 Troubleshooting
		38	Deleted the section of Website and Support.
		Program	Changed bellow for support GCC and IAR compiler: 1. Replaced evenaccess with the macro definition of BSP. 2. Replaced nop with the intrinsic functions of BSP. 3. Replaced the declaration of interrupt functions with the macro definition of BSP. Changed the processing to prevent register access contention between peripheral functions that occurs when using RTOS or when multiple interrupts are enabled. 1. Changed the setting process of the Interrupt Request Enable Bits (IEN) [Description] Changed the setting process of the Interrupt Request Enable Bits (IEN) to use R_BSP_InterruptRequestDisable, and R_BSP_InterruptRequestEnable in the API functions of BSP. 2. Changed the setting process of the Group Interrupt Request Enable Register (GENBL1) (RX64M, RX65N, RX66T, RX71M, and RX72T). [Description] Changed to perform the setting process of the Group Interrupt Request Enable Register (GENBL1) while interrupts are disabled.
3.10	Aug.15.19	1	Added support for RX72M
		9	Added code size corresponding to RX72M
		35	6.1 Confirmed Operation Environment: Added Table for Rev.3.10
		Program	Added support for RX72M.

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on

   The state of the product is undefined at the time 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 time 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 time 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 time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state

   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins

   Handle unused pins in accordance 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 the 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.
5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is 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. Additionally, 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.
6. Voltage application waveform at input pin

   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses

   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. 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 or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving 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, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, 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 and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall 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. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, 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.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/.