RX Family

CAN FD API Using Firmware Integration Technology

Introduction

The Renesas CAN FD (Controller Area Network with Flexible Data Rate) 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 FD peripheral.

Target Devices

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

- RX660 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 “7.1 Confirmed Operation Environment.”

Contents

1. Overview

The CAN FD module can be used to communicate over CAN networks, optionally using Flexible Data (CAN FD) to accelerate the data phase. A variety of message filtering and buffer options are available.

1.1 Basics

Features

- Compatibility
  - Send and receive CAN 2.0 and CAN FD frames on the same channel
  - Data transfer rate: Arbitration phase up to 1 Mbps. With FD, Data phase up to 8 Mbps
  - ISO 11898-1:2015 compliant
- Buffers
  - 32 global receive Message Buffers (RX MBs)
  - 2 global receive FIFOs (RX FIFOs)
  - 4 transmit Message Buffers (TX MBs) per channel
  - One common FIFO that can be configured as a receive FIFO or transmit FIFO
- Filtering
  - Up to 128 filter rules across both channels
  - Each rule can be individually configured to filter based on:
    - ID
    - Standard or Extended ID (IDE bit)
    - Data or Remote Frame (RTR bit)
    - ID/IDE/RTR mask
    - Minimum DLC (data length) value
- Interrupts
  - Configurable Global RX FIFO Interrupt
    - Configurable per FIFO
    - Interrupt at a certain depth or on every received message
  - Channel TX Interrupt
  - Global Error
    - DLC Check
    - Message Lost
    - FD Payload Overflow
  - Channel Error
    - Bus Error
    - Error Warning
    - Error Passive
    - Bus-Off Entry
    - Bus-Off Recovery
    - Overload
    - Bus Lock
    - Arbitration Loss
    - Transmission Aborted

1.1.1 Flexible Data (FD)

Flexible Data is an extension of the CAN protocol allowing for messages up to 64 bytes and higher data bitrates, among other features. The CAN FD driver supports the following:

- Sending and receiving FD messages
- Bitrate switching for data phase (up to 8 MHz)
- Manual and automatic setting of the error state (ESI) bit

To specify one or more of these options when transmitting set can_frame_t::options with combined

values from canfd_frame_options_t. Received messages will automatically have this field filled, if applicable.

```
#define CAN_FD_DATA_LENGTH_CODE (64)   //Data Length code for FD frame

/* Configure a frame to write 64 bytes with bitrate switching (BRS) enabled */
   g_canfd_tx_frame.id = CAN_EXAMPLE_ID;

   g_canfd_tx_frame.id_mode = CAN_ID_MODE_STANDARD;

   g_canfd_tx_frame.type = CAN_FRAME_TYPE_DATA;

   g_canfd_tx_frame.data_length_code = CAN_FD_DATA_LENGTH_CODE;

   g_canfd_tx_frame.options = CANFD_FRAME_OPTION_FD | CANFD_FRAME_OPTION_BRS;
```

Note

When using bitrate switching be sure to configure the Data Bitrate as desired in the "Smart Configurator".

1.1.2 Bit Rate Calculation

The bit rate of the CAN FD peripheral is manually set through the "Smart Configurator".

The CAN FD peripheral uses either PLL or the main oscillator as its clock source. To achieve an exact bitrate the CAN FD source clock or divisor may need to be adjusted to meet the criteria in the formula below:

```
Bitrate = canfd_clock_hz / ((time_segment_1 + time_segment_2 + 1) * prescaler)
```

For CAN FD, the possible values for each element are as follows:

Element	Min	Max (Nominal)	Max (Data)
Bitrate	-	1 Mbps	8 Mbps
Time Segment 1	2 Tq	256 Tq	32 Tq
Time Segment 2	2 Tq	128 Tq	16 Tq
Sync Jump Width	1 Tq	Time Segment 2	Time Segment 2
Prescaler	1	1024	256

Use the Components tab of the "Smart Configurator" to configure the CAN FD clock source/divisor as well as to set the frequency of PLL or the main oscillator.

The Sync Jump Width option specifies the maximum number of time quanta that the sample point may be delayed by to account for differences in oscillators on the bus. It should be set to a value between 1 and the configured Time Segment 2 value depending on the maximum permissible clock error.

The following relations between frequencies must apply if the CAN FD module is to be used.

- PCLKA: PCLKB = 2:1
- PCLKB ≥ CANFDCLK
- PCLKB ≥ CANFDMCLK

Formulas to calculate the bitrate register settings.

PCLK is the peripheral clock frequency, PCLKB.

fcan = PCLK or EXTAL

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

fcanclk = fcan/prescaler

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

Tq =1/fcanclk

Tqtot is the total number of CAN FD peripheral clock cycles during one CAN FD 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) / (CANFD_BRP * BITRATE * BSP_CFG_PCKB_DIV)

Set these macros so that a Tqtot is found which is not larger than accepted by the CANFD registers.

Note: CANFD_BRP defined in user program

BITRATE as expected bitrate

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.

Example calculate the bitrate register settings

CAN FD BITRATE Settings
Consult Section 33.4.1 "Initialization of CAN Clock, Bit Timing and Bit Rate" in the RX660 User's Manual (R01UH0937EJ) for details.

CCLKS is 0(running on PCLK which is PCLKB), that is,
FCANFD = PCLK = PCLKB.
CANFD_BRP = Bit Rate Prescaler.
FCANFDCLK = FCANFD / CANFD_BRP
P = value selected in BRP[9:0] bits in BCR (P = 0 to 1023). P + 1 = CANFD_BRP.
TQTOT = Nr CANFD clocks in one CANFD bit = FCANFDCLK/BITRATE.

With CCLKS = 0, and using r_bsp macros we get:
FCANFD = (BSP_CFG_XTAL_HZ * BSP_CFG_PLL_MUL / BSP_CFG_PCKB_DIV) (Eq. 1)
TQTOT = (FCANFD / (CANFD_BRP * BITRATE)) (Eq. 2)

Eq. (1) in (2):
TQTOT = (BSP_CFG_XTAL_HZ * BSP_CFG_PLL_MUL / BSP_CFG_PCKB_DIV)/(CANFD_BRP * BITRATE)), or
TQTOT = (BSP_CFG_XTAL_HZ * BSP_CFG_PLL_MUL) / (CANFD_BRP * BITRATE * BSP_CFG_PCKB_DIV) (Eq. 3)
Example: Desired bit rate 500 kbps.
Try CANFD_BRP = 4. Equation 3:
TQTOT = (24000000 * 10) / (4 * 500000 * 4) = 30. This is too large. TQTOT can be max 25.
Try CANFD_BRP = 5.
TQTOT =
(BSP_CFG_XTAL_HZ * BSP_CFG_PLL_MUL) / (CANFD_BRP * BITRATE * BSP_CFG_PCKB_DIV)
= (24000000 * 10) / (5 * 500000 * 4) = ***24***
TQTOT = 24 = TSEG1 + TSEG2 + SS:
Try:
SS = 1 Tq always.
TSEG1 = 15 Tq
TSEG2 = 8 Tq
==========
SUM = 24

1.1.3 Error Handing

The CAN FD peripheral provides two types of error interrupts: Channel and Global. As the names imply, each channel has its own Channel Error interrupt but there is only one Global Error interrupt. Only the configured channel will receive callbacks for Global Errors.
Error interrupt callbacks will pass either CAN_EVENT_ERR_CHANNEL or CAN_EVENT_ERR_GLOBAL in the can_callback_args_t::event field. A second field, can_callback_args_t::error, provides the actual error code as canfd_error_t. Cast to this enum to retrieve the error condition.

1.1.4 DLC Checking

When DLC Checking is enabled messages are checked against the destination.minimum_dlc value of each AFL rules. If the data length of a message is less than this value, the message will be rejected.

When DLC checking is set to "DLC Replacement Enable" in the “Smart Configurator” any data in excess of the minimum DLC setting will be truncated and the DLC value for the frame will be set to match.

1.1.5 FD Payload Overflow

When an FD message is received with a DLC larger than the destination buffers an FD Payload Overflow interrupt is thrown (if configured). When Payload Overflow is set to "Truncate" the message will still be accepted but only data up to the buffer capacity will be preserved. The DLC value is unchanged in this case; any data beyond this value in the can_frame_t::data array should not be used.

1.2 Communication Layers

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

Application
Renesas CAN FD API
CAN FD peripheral
MCU/transceivers/CANbus

1.3 Using the FIT CAN FD module

1.3.1 Using FIT CAN FD module in C++ project

For C++ project, add FIT CAN FD module interface header file within extern “C”{}:

```
Extern “C”
{
  #include “r_smc_entry.h”
  #include “r_canfd_rx_if.h”
}
```

1.4 Physical Connection

The Protocol Controller of the CAN FD peripheral in your CAN FD MCU must be connected to a bus transceiver located outside the chip via the CAN FD Transmit (CTXn) and receive (CRXn) MCU pins.

1.5 The CAN FD Buffer

Buffers

The CAN FD driver provides three types of buffers: Transmit Message Buffers (TX MBs), Receive Message Buffers (RX MBs) and FIFO Buffers. The total number of FIFO buffers is three (two receive FIFOs (RX FIFOs) + one common FIFO).

TX Message Buffers

TX MBs is used for transmission only. Refer to the hardware manual for your device for information on which TX MBs are available.

Note

The CAN FD peripheral continually scans TX MBs for new data. Depending on the provided clock it may be possible to write to multiple TX MBs before transmission begins. In this case, messages will be sent in the priority specified by the Transmission Priority option in the “Smart Configurator”*.*

RX Message Buffers

RX MBs are for reception only and may only hold one message at a time.
No interrupts are provided for RX MBs in this software. Use R_CANFD_InfoGet and R_CANFD_Read to poll and read them, respectively.

RX FIFOs

RX FIFOs provide interrupt-driven queue functionality for receiving messages. 2 RX FIFOs are available. All FIFOs have the following capabilities:

- Up to 64 bytes payloads
- Up to 48 message capacity

Once an interrupt is fired it will continue to fire until the FIFO is emptied, and all messages have been passed to user code via the callback. When using the threshold interrupt mode, a FIFO can be checked for data and read between interrupts by calling R_CANFD_InfoGet and R_CANFD_Read, respectively.

RX Buffer Pool

The RAM allocated to the receive message buffers and FIFO buffers is limited to 16 messages (1216 bytes) when the payload size is set to 64 bytes. Do not configure the receive message buffers and FIFO buffers that exceed this maximum limit. CAN FD module does not have the function to check the validity of the configuration.

Limitations

Developers should be aware of the following limitations when using CAN FD:

- RX MBs interrupt is available in the RX MCUs that have CAN FD hardware; however it is not supported in this software. To use them in an application one of the following is recommended: Use R_CANFD_InfoGet to determine if any RX MBs have received data, then use R_CANFD_Read to obtain it.
- The CAN FD peripheral has a limited amount of buffer pool RAM available for allocating RX MBs and FIFO stages. See the RX Buffer Pool section above for more information.
- When switching modes with R_CANFD_ModeTransition a delay of up to several CAN frames may be incurred. Consult Section 33.3.3.2 "Timing of Channel Mode Change" in the RX660 User's Manual (R01UH0937EJ) for details.

Message Filtering (Acceptance Filter List)

To filter messages to the desired message buffer or FIFO the CAN FD peripheral uses an Acceptance Filter List (AFL). Each entry in the AFL provides a rule to check a message against along with destination and other filtering information. When a message is received the CAN FD peripheral internally checks against every configured AFL rule for the channel. If a match is found the message is transferred to the destination(s) specified in the rule. See structure of an AFL entry at canfd_afl_entry_t below:

```
/** AFL Entry */
typedef struct st_canfd_afl_entry_t
{
    uint32_t id                 : 29; ///< ID to match against
    uint32_t rs                 : 1;
    can_frame_type_t frame_type : 1;  ///< Frame type (Data or Remote)
    can_id_mode_t    id_mode    : 1;  ///< ID mode (Standard or Extended)

    uint32_t mask_id            : 29; ///< ID Mask
    uint32_t rs1                : 1;
    uint32_t mask_frame_type    : 1;  ///< Only accept frames with the
configured frame type
    uint32_t mask_id_mode       : 1;  ///< Only accept frames with the
configured ID mode

    canfd_minimum_dlc_t minimum_dlc     : 4; ///< Minimum DLC value to accept
(valid when DLC Check is enabled)
    uint32_t rs2                        : 4;
    canfd_rx_mb_t rx_buffer             : 8; ///< RX Message Buffer to receive
messages accepted by this rule
    uint32_t rs3                        : 16;
    canfd_rx_fifo_t fifo_select_flags;   ///< RX FIFO(s) to receive messages
accepted by this rule
} canfd_afl_entry_t;
```

For an example configuration refer to the AFL Example below.

AFL Example

The below is an example Acceptance Filter List (AFL) declaration with one rule.

```
/* Acceptance filter array parameters
CANFD_CFG_AFL_CH0_RULE_NUM = 1 */
/* Acceptance filter array parameters */
#define CANFD_FILTER_ID (0x00001000)
#define MASK_ID                 (0x0FFFF000)
#define MASK_ID_MODE    (1)
#define ZERO                        (0U) //Array Index value

const canfd_afl_entry_t p_canfd0_afl[CANFD_CFG_AFL_CH0_RULE_NUM] =
{
    /* Accept a message with Extended ID 0x1000-0x1FFF */
    /* Specify the ID, ID type and frame type to accept. */
    {
      CANFD_FILTER_ID,
      0,
      CAN_FRAME_TYPE_DATA,
      CAN_ID_MODE_EXTENDED,
      MASK_ID,
```

```
      0,
      ZERO,
      MASK_ID_MODE,
      (canfd_minimum_dlc_t)ZERO,
      0,
      CANFD_RX_MB_0,
      0,
      CANFD_RX_FIFO_0
      },
};

void main(void)
{
    g_canfd0_extended_cfg.p_afl = p_canfd0_afl;
    err = R_CANFD_Open(&g_canfd0_ctrl, &g_canfd0_cfg);
}
```

Consult Section 33.5 "Filtering Using Acceptance Filter List (AFL)" in the RX660 User's Manual (R01UH0937EJ) for details.

2. API Information

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

2.1 Hardware Requirements

This driver requires that your MCU supports the following peripheral:

- CAN FD Module (CAN FD)

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 FD Module (CAN FD)

2.2.2 Other Peripherals Used

The driver requires I/O port pins to be assigned for CAN FD 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 FD channel.

2.3 Software Requirements

This driver is dependent upon the following FIT module:

- Renesas Board Support Package (r_bsp) v7.20 or higher

2.4 Limitations

2.4.1 RAM Location Limitations

In FIT, if a value equivalent to NULL is set as the pointer argument of an API function, error might be returned due to parameter check. Therefore, do not pass a NULL equivalent value as pointer argument to an API function.
The NULL value is defined as 0 because of the library function specifications. Therefore, the above phenomenon would occur when the variable or function passed to the API function pointer argument is located at the start address of RAM (address 0x0). In this case, change the section settings or prepare a dummy variable at the top of the RAM so that the variable or function passed to the API function pointer argument is not located at address 0x0.
In the case of the CCRX project (e2 studio V7.5.0), the RAM start address is set as 0x4 to prevent the variable from being located at address 0x0. In the case of the GCC project (e2 studio V7.5.0) and IAR project (EWRX V4.12.1), the start address of RAM is 0x0, so the above measures are necessary.
The default settings of the section may be changed due to the IDE version upgrade. Please check the section settings when using the latest IDE.

2.5 Supported Toolchain

This driver has been confirmed to work with the toolchain listed in 7.1 Confirmed Operation Environment.

2.6 Interrupt Vector

When CAN TX and CAN RX interrupts are used, make sure the respective interrupt are mapped to a software configurable interrupt. This can be done in “r_bsp_interrupt_config.h”

2.7 Header Files

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

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

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

#include “r_canfd_rx_if.h”

2.8 Integer Types

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

2.9 Configuration

It will be necessary to make modifications to the *r_canfd_rx_config.h* file to customize the application for desired functionality. It is not recommended to change the *r_canfd_rx.c* file, which contains the Renesas CAN FD API driver function, but this may be merited to add some features 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_canfd_rx_config.h* can be edited manually using the following tables as a guide.

Configuration options in r_canfd_rx_config.h	
CANFD_CFG_PARAM_CHECKING_ENABLE (BSP_CFG_PARAM_CHECKING_ENABLE)	1: Parameter checking is included in the build. 0: Parameter checking is omitted from the build. Setting this #define to BSP_CFG_PARAM_CHECKING_ENABLE utilizes the system default setting.
CANFD_CFG_AFL_CH0_RULE_NUM 32	Number of acceptance filter list rules dedicated to Channel 0. Any value (0~32) Default value is 32.
CANFD_CFG_FD_PROTOCOL_EXCEPTION 0	Select whether to enter the protocol exception handling state when a RES bit is sampled recessive as defined in ISO 11898-1. (0) = Enabled (ISO 11898-1) (default) (R_CANFD_GFDCFG_PXEDIS_Msk) = Disabled
CANFD_CFG_GLOBAL_ERR_SOURCES 0x3	Select which errors should trigger an interrupt. (0x3) (default) (R_CANFD_GCR_DEIE_Msk | 0x3) (R_CANFD_GCR_MLIE_Msk | 0x3) (R_CANFD_GCR_POIE_Msk | 0x3) (R_CANFD_GCR_DEIE_Msk | R_CANFD_GCR_MLIE_Msk | 0x3) (R_CANFD_GCR_DEIE_Msk | R_CANFD_GCR_POIE_Msk | 0x3) (R_CANFD_GCR_MLIE_Msk | R_CANFD_GCR_POIE_Msk | 0x3) (R_CANFD_GCR_DEIE_Msk | R_CANFD_GCR_MLIE_Msk | R_CANFD_GCR_POIE_Msk | 0x3)

Configuration options in r_canfd_rx_config.h	
CANFD_CFG_TX_PRIORITY (R_CANFD_GCFG_TPRI_Msk)	Select how messages should be prioritized for transmission. In either case, lower numbers indicate higher priority. (0) = Message ID (R_CANFD_GCFG_TPRI_Msk) = Buffer Number (default)
CANFD_CFG_DLC_CHECK 0	When enabled received messages will be rejected if their DLC field is less than the value configured in the associated AFL rule. If 'DLC Replacement Enable' is selected and a message passes the DLC check the DLC field is set to the value in the associated AFL rule and any excess data is discarded. (0) = Disabled (default) (R_CANFD_GCFG_DCE_Msk) = Enabled (R_CANFD_GCFG_DCE_Msk | R_CANFD_GCFG_DRE_Msk) = DLC Replacement Enable
CANFD_CFG_FD_OVERFLOW 0	Configure whether received messages larger than the destination buffer should be truncated or rejected. (0) = Reject (default) (R_CANFD_GCFG_TPRI_Msk) = Truncate
CANFD_CFG_CANFDCLK_SOURCE 0	Configure the CAN FD Clock source to be either PLL (default) or crystal direct. (0) = PLL (default) (1) = Crystal direct
CANFD_CFG_RXMB_NUMBER 0	Number of message buffers available for reception. As there is no interrupt for message buffer reception it is recommended to use RX FIFOs instead. Set this value to 0 to disable RX Message Buffers. Any value (0~32) Default value is 0.

Configuration options in r_canfd_rx_config.h	
CANFD_CFG_RXMB_SIZE 0	Payload size for all RX Message Buffers. (0) = 8 bytes (default) (1) = 12 bytes (2) = 16 bytes (3) = 20 bytes (4) = 24 bytes (5) = 32 bytes (6) = 48 bytes (7) = 64 bytes
CANFD_CFG_GLOBAL_ERR_IPL 12	This interrupt is fired for each of the error sources selected below. Any value (0) ~ (15) Default value is (12)
CANFD_CFG_RX_FIFO_IPL 12	Selects whether to include parameter checking in the code. BSP_CFG_PARAM_CHECKING_ENABLE = Default (BSP). Any value (0) ~ (15) Default value is (12).
CANFD_CFG_RXFIFO0_INT_THRESHOLD 3U	Set the interrupt threshold value for RX FIFO 0. This setting is only applicable when the Interrupt Mode is set to 'At Threshold Value'. (0U) = 1/8 full (1U) = 1/4 full (2U) = 3/8 full (3U) = 1/2 full (default) (4U) = 5/8 full (5U) = 3/4 full (6U) = 7/8 full (7U) = full
CANFD_CFG_RXFIFO0_DEPTH 3	Select the number of stages for RX FIFO 0. (1) = 4 stages (2) = 8 stages (3) = 16 stages (default) (4) = 32 stages (5) = 48 stages
CANFD_CFG_RXFIFO0_PAYLOAD 7	Select the message payload size for RX FIFO 0. (0) = 8 bytes (1) = 12 bytes (2) = 16 bytes (3) = 20 bytes (4) = 24 bytes (5) = 32 bytes (6) = 48 bytes (7) = 64 bytes (default)

Configuration options in r_canfd_rx_config.h	
CANFD_CFG_RXFIFO0_INT_MODE ((R_CANFD_RFCR_RFIE_Msk | R_CANFD_RFCR_RFIM_Msk))	Set the interrupt mode for RX FIFO 0. Threshold mode will only fire an interrupt each time an incoming message crosses the threshold value set below. (0) = Disabled (R_CANFD_RFCR_RFIE_Msk) = At Threshold Value (R_CANFD_RFCR_RFIE_Msk | R_CANFD_RFCR_RFIM_Msk) = Every Frame (default)
CANFD_CFG_RXFIFO0_ENABLE 1	Enable or disable RX FIFO 0. (0) = Disabled (1) = Enabled (default)
CANFD_CFG_RXFIFO1_INT_THRESHOLD 3U	Set the interrupt threshold value for RX FIFO 1. This setting is only applicable when the Interrupt Mode is set to 'At Threshold Value'. (0U) = 1/8 full (1U) = 1/4 full (2U) = 3/8 full (3U) = 1/2 full (default) (4U) = 5/8 full (5U) = 3/4 full (6U) = 7/8 full (7U) = full
CANFD_CFG_RXFIFO1_DEPTH 3	/* Select the number of stages for RX FIFO 1. (1) = 4 stages (2) = 8 stages (3) = 16 stages (default) (4) = 32 stages (5) = 48 stages
CANFD_CFG_RXFIFO1_PAYLOAD 7	Select the message payload size for RX FIFO 1. (0) = 8 bytes (1) = 12 bytes (2) = 16 bytes (3) = 20 bytes (4) = 24 bytes (5) = 32 bytes (6) = 48 bytes (7) = 64 bytes (default)

Configuration options in r_canfd_rx_config.h	
CANFD_CFG_RXFIFO1_INT_MODE ((R_CANFD_RFCR_RFIE_Msk | R_CANFD_RFCR_RFIM_Msk))	Set the interrupt mode for RX FIFO 1. Threshold mode will only fire an interrupt each time an incoming message crosses the threshold value set below. (0) = Disabled (R_CANFD_RFCR_RFIE_Msk) = At Threshold Value (R_CANFD_RFCR_RFIE_Msk | R_CANFD_RFCR_RFIM_Msk) = Every Frame (default)
CANFD_CFG_RXFIFO1_ENABLE 0	Enable or disable RX FIFO 0. (0) = Disabled (default) (1) = Enabled
CANFD0_EXTENDED_CFG_TXMB0_TXI_ENABLE 0ULL CANFD0_EXTENDED_CFG_TXMB1_TXI_ENABLE 0ULL CANFD0_EXTENDED_CFG_TXMB2_TXI_ENABLE 0ULL CANFD0_EXTENDED_CFG_TXMB3_TXI_ENABLE 0ULL	Select TX Message buffers should trigger an interrupt when transmission is complete. Disabled = 0ULL (default) Enabled = (1ULL << 0)
CANFD0_EXTENDED_CFG_WARNING_ERROR_INTERRUPTS 0U	Select Error Warning interrupt sources to enable. Disabled = 0ULL (default) Enabled = R_CANFD_CHCR_EWIE_Msk
CANFD0_EXTENDED_CFG_PASSING_ERROR_INTERRUPTS 0U	Select error passive interrupt sources to enable. Disabled = 0U (default) Enabled = R_CANFD_CHCR_EPIE_Msk
CANFD0_EXTENDED_CFG_BUS_OFF_ENTRY_ERROR_INTERRUPTS 0U	Select which channel bus-Off Entry error interrupt sources to enable. Disabled = 0U (default) Enabled = R_CANFD_CHCR_BOEIE_Msk
CANFD0_EXTENDED_CFG_BUS_OFF_RECOVERY_ERROR_INTERRUPTS 0U	Select channel bus-Off Recovery error interrupt sources to enable. Disabled = 0U (default) Enabled = R_CANFD_CHCR_BORIE_Msk
CANFD0_EXTENDED_CFG_OVERLOAD_ERROR_INTERRUPTS 0U	Select channel overload error interrupt sources to enable. Disabled = 0U (default) Enabled = R_CANFD_CHCR_OLIE_Msk
CANFD0_CFG_IPL 12	This interrupt is fired for each of the error sources selected below. Any value (0) ~ (15) Default value is (12).

Configuration options in r_canfd_rx_config.h	
CANFD0_BIT_TIMING_CFG_BRP 1	Specify clock divisor for nominal bitrate. Any value (1~1024) Default value is (1).
CANFD0_BIT_TIMING_CFG_TSEG1 29	Select the Time Segment 1 value. Check module usage notes for how to calculate this value. Any value (2~256) Default value is (29).
CANFD0_BIT_TIMING_CFG_TSEG2 10	Select the Time Segment 2 value. Check module usage notes for how to calculate this value. Any value (2~128) Default value is (10).
CANFD0_BIT_TIMING_CFG_SJW 4	Select the Synchronization Jump Width value. Check module usage notes for how to calculate this value. Any value (1~128) Default value is (4).
CANFD0_DATA_TIMING_CFG_BRP 1	Specify clock divisor for data bitrate. Any value (1~1024) Default value is (1).
CANFD0_DATA_TIMING_CFG_TSEG1 5	Select the Time Segment 1 value. Check module usage notes for how to calculate this value. Any value (2~32) Default value is (5).
CANFD0_DATA_TIMING_CFG_TSEG2 2	Select the Time Segment 2 value. Check module usage notes for how to calculate this value. Any value (2~16) Default value is (2).
CANFD0_DATA_TIMING_CFG_SJW 1	Select the Synchronization Jump Width value. Check module usage notes for how to calculate this value. Any value (1~16) Default value is (1).
CANFD0_EXTENDED_CFG_DELAY_COMPENSATION 1	When enabled the CAN FD module will automatically compensate for any transceiver or bus delay between transmitted and received bits. When manually supplying bit timing values with delay compensation enabled be sure the data prescaler is 2 or smaller for correct operation. (0) = Disabled (1) = Enabled (default) Default value is (1).

2.10 Interfaces and Instances

This section describes structures in r_canfd_rx/inc

2.10.1 CAN interface

This section describes structures in r_canfd_rx/inc/ r_can_api.h

The CAN interface provides common features and interaction methods of different implementations of CAN drivers. These common features and interaction methods allow upper layer caller function to be able to swap in and out different CAN driver modules which provide the same features. In this Application Note, CAN interface is implemented by CAN FD

CAN interface supports following features:
- Full-duplex CAN communication
- Generic CAN parameter setting
- Interrupt driven transmit/receive processing
- Callback function support with returning event code
- Hardware resource locking during a transaction

Implemented by:
- Controller Area Network - Flexible Data (r_canfd)

Data Structures

struct	can_info_t
struct	can_bit_timing_cfg_t
struct	can_frame_t
struct	can_callback_args_t
struct	can_cfg_t
struct	can_api_t
struct	can_instance_t

Enumerations

enum	can_event_t
enum	can_operation_mode_t
enum	can_test_mode_t
enum	can_id_mode_t
enum	can_frame_type_t

Typedefs

typedef void	can_ctrl_t

♦ **can_info_t**

struct can_info_t		
CAN status info		
Data Fields		
uint32_t	status	Useful information from theCAN status register.
uint32_t	rx_mb_status	RX Message Buffer New Data flags.
uint32_t	rx_fifo_status	RX FIFO Empty flags.
uint8_t	error_count_transmit	Transmit error count.
uint8_t	error_count_receive	Receive error count.
uint32_t	error_code	Error code, cleared after reading.

♦ **can_bit_timing_cfg_t**

struct can_bit_timing_cfg_t		
CAN bit rate configuration.		
Data Fields		
uint32_t	baud_rate_prescaler	Baud rate prescaler. Validvalues: 1 - 1024.
uint32_t	time_segment_1	Time segment 1 control.
uint32_t	time_segment_2	Time segment 2 control.
uint32_t	synchronization_jump_width	Synchronization jump width.

♦ **can_frame_t**

struct can_frame_t		
CAN data Frame		
Data Fields		
uint32_t	id	CAN ID.

can_id_mode_t	id_mode	Standard or Extended ID (IDE).
can_frame_type_t	type	Frame type (RTR).
uint8_t	data_length_code	CAN Data Length Code (DLC).
uint32_t	options	Implementation-Specific options
uint8_t	data[CAN_DATA_BUFFER_LENG TH]	CAN data.

◆ **can_callback_args_t**

struct can_callback_args_t			
CAN callback parameter definition			
Data Fields			
uint32_t		channel	Device channel number.
can_event_t		event	Event code.
uint32_t		error	Error code.
union	uint32_t	mailbox	Mailbox number of interrupt source.
	uint32_t	buffer	Buffer number of interrupt source.
can_frame_t *		p_frame	DEPRECATED Pointer to the received frame.
void const *		p_context	Context provided to user during callback
can_frame_t		frame	Received frame data.

◆ **can_cfg_t**

struct can_cfg_t	
CAN Configuration	
Data Fields	
uint32_t	channel
	CAN channel.
can_bit_timing_cfg_t *	p_bit_timing

	CAN bit timing.
void(*	p_callback)(can_callback_args_t *p_args)
	Pointer to callback function.
void const *	p_context
	User defined callback context.
void const *	p_extend
	CAN hardware dependent configuration.
uint8_t	ipl
	Error/Transmit/Receive interrupt priority.

♦ **can_api_t**

struct can_api_t
Shared Interface definition for CAN
Data Fields
fsp_err_t(*open)(can_ctrl_t *const p_ctrl, can_cfg_t const *const p_cfg)
fsp_err_t(*write)(can_ctrl_t *const p_ctrl, uint32_t buffer_number, can_frame_t *const p_frame)
fsp_err_t(*read)(can_ctrl_t *const p_ctrl, uint32_t buffer_number, can_frame_t *const p_frame)
fsp_err_t(*close)(can_ctrl_t *const p_ctrl)
fsp_err_t(*modeTransition)(can_ctrl_t *const p_api_ctrl, can_operation_mode_toperation_mode, can_test_mode_t test_mode)

fsp_err_t(*infoGet)(can_ctrl_t *const p_ctrl, can_info_t *const p_info)

fsp_err_t(*callbackSet)(can_ctrl_t *const p_api_ctrl, void(*p_callback)(can_callback_args_t *), void const *constp_context, can_callback_args_t *const p_callback_memory)

Field Documentation

♦ **open**

fsp_err_t(* can_api_t::open) (can_ctrl_t *const p_ctrl, can_cfg_t const *const p_cfg)

Open function for CAN device

Implemented as
R_CANFD_Open()

Parameters

[in]	p_ctrl	Pointer to the CAN control block. Must be declared by user.
[in]	can_cfg_t	Pointer to CAN configuration structure. All elements of this structure must be set by user

♦ **write**

fsp_err_t(* can_api_t::write) (can_ctrl_t *const p_ctrl, uint32_t buffer_number, can_frame_t *const p_frame)

Write function for CAN device

Implemented as

R_CANFD_Write()

Parameters

[in]	p_ctrl	Pointer to the CAN control block.
[in]	buffer_number	Buffer number (mailbox or message buffer) to write to.

[in]	p_frame	Pointer for frame of CAN ID, DLC, data and frame type to write.

♦ **read**

fsp_err_t(* can_api_t::read) (can_ctrl_t *const p_ctrl, uint32_t buffer_number, can_frame_t *const p_frame)

Read function for CAN device

Implemented as
R_CANFD_Read()

Parameters

[in]	p_ctrl	Pointer to the CAN control block.
[in]	buffer_number	Message buffer (number) to read from.
[in]	p_frame	Pointer to store the CAN ID, DLC, data and frame type.

♦ **close**

fsp_err_t(* can_api_t::close) (can_ctrl_t *const p_ctrl)

Close function for CAN device

Implemented as
R_CANFD_Close()
Parameters

[in]	p_ctrl	Pointer to the CAN control block.

♦ **modeTransition**

fsp_err_t(* can_api_t::modeTransition) (can_ctrl_t *const p_api_ctrl, can_operation_mode_t operation_mode, can_test_mode_t test_mode)

Mode Transition function for CAN device

Implemented as
R_CANFD_ModeTransition()
Parameters

[in]	p_api_ctrl	Pointer to the CAN control block.
[in]	operation_mode	Destination CAN operation state.
[in]	test_mode	Destination CAN test state.

♦ infoGet

fsp_err_t(* can_api_t::infoGet) (can_ctrl_t *const p_ctrl, can_info_t *const p_info)

Get CAN channel info.

Implemented as

R_CANFD_InfoGet()

Parameters

[in]	p_ctrl	Handle for channel (pointer to channel control block)
[out]	p_info	Memory address to return channel specific data to.

♦ callbackSet

fsp_err_t(* can_api_t::callbackSet) (can_ctrl_t *const p_api_ctrl, void(*p_callback)(can_callback_args_t *), void const *const p_context, can_callback_args_t *constp_callback_memory)

Specify callback function and optional context pointer and working memory pointer.

Implemented as

R_CANFD_CallbackSet()

Parameters

[in]	p_ctrl	Control block set in can_api_t::open call.
[in]	p_callback	Callback function to register
[in]	p_context	Pointer to send to callback function

[in]	p_working_memory	Pointer to volatile memory where callback structure can be allocated. Callback arguments allocated here are only valid during the callback.

♦ can_instance_t

struct can_instance_t		
This structure encompasses everything that is needed to use an instance of this interface.		
can_ctrl_t *	p_ctrl	Pointer to the control structure for this instance.
can_cfg_t const *	p_cfg	Pointer to the configuration structure for this instance.
can_api_t const *	p_api	Pointer to the API structure for this instance.

♦ can_ctrl_t

typedef void can_ctrl_t

CAN control block. Allocate an instance specific control block to pass into the CAN FD API calls.
Implemented as

- canfd_instance_ctrl_t

♦ can_event_t

enum can_event_t	
CAN event codes	
Enumerator	
CAN_EVENT_ERR_WARNING	Error Warning event.
CAN_EVENT_ERR_PASSIVE	Error Passive event.
CAN_EVENT_ERR_BUS_OFF	Bus Off event.
CAN_EVENT_BUS_RECOVERY	Bus Off Recovery event.
CAN_EVENT_MAILBOX_MESSAGE_LOST	Mailbox has been overrun.

CAN_EVENT_ERR_BUS_LOCK	Bus lock detected (32 consecutive dominant bits).
CAN_EVENT_ERR_CHANNEL	Channel error has occurred.
CAN_EVENT_TX_ABORTED	Transmit abort event.
CAN_EVENT_RX_COMPLETE	Receive complete event.
CAN_EVENT_TX_COMPLETE	Transmit complete event.
CAN_EVENT_ERR_GLOBAL	Global error has occurred.
CAN_EVENT_TX_FIFO_EMPTY	Transmit FIFO is empty.

♦ can_operation_mode_t

enum can_operation_mode_t	
CAN Operation modes	
Enumerator	
CAN_OPERATION_MODE_NORMAL	CAN Normal Operation Mode.
CAN_OPERATION_MODE_RESET	CAN Reset Operation Mode.
CAN_OPERATION_MODE_HALT	CAN Halt Operation Mode.
CAN_OPERATION_MODE_SLEEP	CAN Sleep Operation Mode.
CAN_OPERATION_MODE_GLOBAL_OPERATION	CAN FD Global Operation Mode.
CAN_OPERATION_MODE_GLOBAL_RESET	CAN FD Global Reset Mode.
CAN_OPERATION_MODE_GLOBAL_HALT	CAN FD Global Halt Mode.
CAN_OPERATION_MODE_GLOBAL_SLEEP	CAN FD Global Sleep Mode.

♦ can_test_mode_t

enum can_test_mode_t
CAN Test modes
Enumerator

CAN_TEST_MODE_DISABLED	CAN Test Mode Disabled.
CAN_TEST_MODE_LISTEN	CAN Test Listen Mode.
CAN_TEST_MODE_LOOPBACK_EXTERNAL	CAN Test External Loopback Mode.
CAN_TEST_MODE_LOOPBACK_INTERNAL	CAN Test Internal Loopback Mode.
CAN_TEST_MODE_INTERNAL_BUS	CAN FD Internal CAN Bus Communication TestMode.

♦ **can_id_mode_t**

enum can_id_mode_t	
CAN ID modes	
Enumerator	
CAN_ID_MODE_STANDARD	Standard IDs of 11 bits used.
CAN_ID_MODE_EXTENDED	Extended IDs of 29 bits used.

♦ **can_frame_type_t**

enum can_frame_type_t	
CAN frame types	
Enumerator	
CAN_FRAME_TYPE_DATA	Data frame.
CAN_FRAME_TYPE_REMOTE	Remote frame.

2.10.2 CAN FD instance

This section describes structures in r_canfd_rx/inc/ r_canfd.h

CAN FD instance is one of the actual implementations of CAN interface. The CAN FD instance uses the enumerations, data structures, and API prototypes from the CAN interface

Data Structures

struct canfd_instance_ctrl_t

struct	canfd_afl_entry_t
struct	canfd_global_cfg_t
struct	canfd_extended_cfg_t

Enumerations

enum	canfd_frame_options_t
enum	canfd_error_t
enum	canfd_tx_mb_t
enum	canfd_rx_buffer_t
enum	canfd_rx_mb_t
enum	canfd_rx_fifo_t
enum	canfd_minimum_dlc_t

♦ canfd_instance_ctrl_t

struct canfd_instance_ctrl_t		
CAN FD Instance Control Block		
Data Fields		
can_cfg_t const	* p_cfg	Pointer to the configuration structure
uint32_t	open	Open status of channel
can_operation_mode_t	operation_mode	Can operation mode
can_test_mode_t	test_mode	Can test mode
void	(* p_callback)(can_callback_args_t *)	Pointer to callback
can_callback_args_t	* p_callback_memory	Pointer to optional callback argument memory
void const	* p_context	Pointer to context to be passed into callback function

♦ canfd_afl_entry_t

struct canfd_afl_entry_t
AFL Entry
Data Fields

uint32_t	id	ID to match against
can_frame_type_t	frame_type	Frame type (Data or Remote)
can_id_mode_t	id_mode	ID mode (Standard or Extended)
uint32_t	mask_id	ID Mask
uint32_t	mask_frame_type	Only accept frames with the configured frame type
uint32_t	mask_id_mode	Only accept frames with the configured ID mode
canfd_minimum_dlc_t	minimum_dlc	Minimum DLC value to accept (valid when DLC Check is enabled)
canfd_rx_mb_t	rx_buffer	RX Message Buffer to receive messages accepted by this rule
canfd_rx_fifo_t	fifo_select_flags	RX FIFO(s) to receive messages accepted by this rule

♦ canfd_global_cfg_t

struct canfd_global_cfg_t		
CAN FD Global Configuration		
Data Fields		
uint32_t	global_interrupts	Global control options (GCR register setting)
uint32_t	global_config	Global configuration options(GCFG register setting)
uint32_t	rx_fifo_config[2]	RX FIFO configuration (RFCRn register settings)
uint32_t	rx_mb_config	Number and size of RX Message buffers (RMCR register setting)
uint8_t	global_err_ipl	Global Error interrupt priority.
uint8_t	rx_fifo_ipl	RX FIFO interrupt priority.

♦ **canfd_extended_cfg_t**

struct canfd_extended_cfg_t
CAN FD Extended Configuration

Data Fields		
canfd_afl_entry_t const *	p_afl	AFL rules list.
uint32_t	txmb_txi_enable	Array of TX Message Bufferenable bits.
uint32_t	error_interrupts	Error interrupts enable bits.
can_bit_timing_cfg_t *	p_data_timing	FD Data Rate (when bitrate switching is used)
uint8_t	delay_compensation	FD Transceiver Delay Compensation (enable or disable)
canfd_global_cfg_t *	p_global_cfg	Global configuration (global error callback channel only)

♦ canfd_status_t

enum canfd_status_t	
CAN FD Status	
Enumerator	
CANFD_STATUS_RESET_MODE	Channel in Reset mode.
CANFD_STATUS_HALT_MODE	Channel in Halt mode.
CANFD_STATUS_SLEEP_MODE	Channel in Sleep mode.
CANFD_STATUS_ERROR_PASSIVE	Channel in error-passive state.
CANFD_STATUS_BUS_OFF	Channel in bus-off state.
CANFD_STATUS_TRANSMITTING	Channel is transmitting.
CANFD_STATUS_RECEIVING	Channel is receiving.
CANFD_STATUS_READY	Channel is ready for communication.
CANFD_STATUS_ESI	At least one CAN FD message was received with the ESI flag set.

♦ canfd_error_t

enum canfd_error_t	
CAN FD Error Code	
Enumerator	
CANFD_ERROR_CHANNEL_BUS	Bus Error.
CANFD_ERROR_CHANNEL_WARNING	Error Warning (TX/RX error count over 0x5F)
CANFD_ERROR_CHANNEL_PASSIVE	Error Passive (TX/RX error count over 0x7F)
CANFD_ERROR_CHANNEL_BUS_OFF_ENTRY	Bus-Off State Entry.
CANFD_ERROR_CHANNEL_BUS_OFF_RECOVERY	Recovery from Bus-Off State.
CANFD_ERROR_CHANNEL_OVERLOAD	Overload.
CANFD_ERROR_CHANNEL_BUS_LOCK	Bus Locked.
CANFD_ERROR_CHANNEL_ARBITRATION_LOSS	Arbitration Lost.
CANFD_ERROR_CHANNEL_STUFF	Stuff Error.
CANFD_ERROR_CHANNEL_FORM	Form Error.
CANFD_ERROR_CHANNEL_ACK	ACK Error.
CANFD_ERROR_CHANNEL_CRC	CRC Error.
CANFD_ERROR_CHANNEL_BIT_RECESSIVE	Bit Error (recessive) Error.
CANFD_ERROR_CHANNEL_BIT_DOMINANT	Bit Error (dominant) Error.
CANFD_ERROR_CHANNEL_ACK_DELIMITER	ACK Delimiter Error.
CANFD_ERROR_GLOBAL_DLC	DLC Error.
CANFD_ERROR_GLOBAL_MESSAGE_LOST	Message Lost.
CANFD_ERROR_GLOBAL_PAYLOAD_OVERFLOW	FD Payload Overflow.
CANFD_ERROR_GLOBAL_TXQ_OVERWRITE	TX Queue Message Overwrite.
CANFD_ERROR_GLOBAL_TXQ_MESSAGE_LOST	TX Queue Message Lost.

CANFD_ERROR_GLOBAL_CH0_SCAN_FAIL	Channel 0 RX Scan Failure.
CANFD_ERROR_GLOBAL_CH1_SCAN_FAIL	Channel 1 RX Scan Failure.
CANFD_ERROR_GLOBAL_CH0_ECC	Channel 0 ECC Error.
CANFD_ERROR_GLOBAL_CH1_ECC	Channel 1 ECC Error.

♦ **canfd_tx_mb_t**

enum canfd_tx_mb_t
CAN FD Transmit Message Buffer (TX MB)

♦ **canfd_rx_buffer_t**

enum canfd_rx_buffer_t
CAN FD Receive Buffer (MB + FIFO)

♦ **canfd_rx_mb_t**

enum canfd_rx_mb_t
CAN FD Receive Message Buffer (RX MB)

♦ **canfd_rx_fifo_t**

enum canfd_rx_fifo_t
CAN FD Receive FIFO (RX FIFO)

♦ **canfd_minimum_dlc_t**

enum canfd_minimum_dlc_t
CAN FD AFL Minimum DLC settings

♦ **canfd_frame_options_t**

enum canfd_frame_options_t
CAN FD Frame Options
Enumerator

CANFD_FRAME_OPTION_ERROR	Error state set (ESI).
CANFD_FRAME_OPTION_BRS	Bit Rate Switching (BRS) enabled.
CANFD_FRAME_OPTION_FD	Flexible Data frame (FDF).

2.11 Instance Structure

The CANFD source code created an instance structure to use this module:

It includes:

- A pointer(p_ctrl) to the control structure
- A pointer(p_cfg) to the configuration structure
- A pointer(p_api) to the instance API structure

The control, configuration, and instance API structure have been created with the default value in the file “r_canfd_data.c”.

Below is the instance structure(g_canfd0) which has been created for channel 0 with the control structure(g_canfd0_ctrl), the configuration structure(g_canfd0_cfg) and the instance API structure (g_canfd_on_canfd).

Example:

```
/* Instance structure to use CAN FD module channel 0. */
const can_instance_t g_canfd0 =
{
      .p_ctrl = &g_canfd0_ctrl,
      .p_cfg = &g_canfd0_cfg,
      .p_api = &g_canfd_on_canfd
};

canfd_instance_ctrl_t g_canfd0_ctrl;
can_cfg_t g_canfd0_cfg =
{
      .channel = 0,
      .p_bit_timing = &g_canfd0_bit_timing_cfg,
      .p_callback = NULL,
      .p_extend = &g_canfd0_extended_cfg,
      .p_context = NULL,
      .ipl = CANFD0_CFG_IPL,
 };

/* Config Nominal bit rate */
can_bit_timing_cfg_t g_canfd0_bit_timing_cfg =
{
      .baud_rate_prescaler = CANFD0_BIT_TIMING_CFG_BRP,
      .time_segment_1 = CANFD0_BIT_TIMING_CFG_TSEG1,
      .time_segment_2 = CANFD0_BIT_TIMING_CFG_TSEG2,
      .synchronization_jump_width = CANFD0_BIT_TIMING_CFG_SJW
};
canfd_extended_cfg_t g_canfd0_extended_cfg =
{
      .p_afl = NULL,
      .txmb_txi_enable = (CANFD0_EXTENDED_CFG_TXMB0_TXI_ENABLE
```

```
        | CANFD0_EXTENDED_CFG_TXMB1_TXI_ENABLE
        | CANFD0_EXTENDED_CFG_TXMB2_TXI_ENABLE
        | CANFD0_EXTENDED_CFG_TXMB3_TXI_ENABLE | 0ULL),
      .error_interrupts = (CANFD0_EXTENDED_CFG_WARNING_ERROR_INTERRUPTS
                | CANFD0_EXTENDED_CFG_PASSING_ERROR_INTERRUPTS
                | CANFD0_EXTENDED_CFG_BUS_OFF_ENTRY_ERROR_INTERRUPTS
                | CANFD0_EXTENDED_CFG_BUS_OFF_RECOVERY_ERROR_INTERRUPTS
                | CANFD0_EXTENDED_CFG_OVERLOAD_ERROR_INTERRUPTS | 0U),
      .p_data_timing = &g_canfd0_data_timing_cfg, .delay_compensation =
CANFD0_EXTENDED_CFG_DELAY_COMPENSATION,
      .p_global_cfg = &g_canfd_global_cfg,
};

/* Config data rate */
can_bit_timing_cfg_t g_canfd0_data_timing_cfg =
{
      .baud_rate_prescaler = CANFD0_DATA_TIMING_CFG_BRP,
      .time_segment_1 = CANFD0_DATA_TIMING_CFG_TSEG1,
      .time_segment_2 = CANFD0_DATA_TIMING_CFG_TSEG2,
      .synchronization_jump_width = CANFD0_DATA_TIMING_CFG_SJW
};
#ifndef CANFD_PRV_GLOBAL_CFG
#define CANFD_PRV_GLOBAL_CFG
canfd_global_cfg_t g_canfd_global_cfg =
{
      .global_interrupts = CANFD_CFG_GLOBAL_ERR_SOURCES,
      .global_config = (CANFD_CFG_TX_PRIORITY | CANFD_CFG_DLC_CHECK
        | ((1U == CANFD_CFG_CANFDCLK_SOURCE)? R_CANFD_GCFG_DLLCS_Msk: 0U)
        | CANFD_CFG_FD_OVERFLOW),
      .rx_mb_config = (CANFD_CFG_RXMB_NUMBER | (CANFD_CFG_RXMB_SIZE <<
R_CANFD_RMCR_PLS_Pos)),
      .global_err_ipl = CANFD_CFG_GLOBAL_ERR_IPL,
      .rx_fifo_ipl = CANFD_CFG_RX_FIFO_IPL,
      .rx_fifo_config =
      {
      ((CANFD_CFG_RXFIFO0_INT_THRESHOLD << R_CANFD_RFCR_RFITH_Pos)
            | (CANFD_CFG_RXFIFO0_DEPTH << R_CANFD_RFCR_FDS_Pos)
            | (CANFD_CFG_RXFIFO0_PAYLOAD << R_CANFD_RFCR_PLS_Pos) |
(CANFD_CFG_RXFIFO0_INT_MODE) | (CANFD_CFG_RXFIFO0_ENABLE)),
      ((CANFD_CFG_RXFIFO1_INT_THRESHOLD << R_CANFD_RFCR_RFITH_Pos)
            | (CANFD_CFG_RXFIFO1_DEPTH << R_CANFD_RFCR_FDS_Pos)
            | (CANFD_CFG_RXFIFO1_PAYLOAD << R_CANFD_RFCR_PLS_Pos) |
(CANFD_CFG_RXFIFO1_INT_MODE) | (CANFD_CFG_RXFIFO1_ENABLE)),
      },
};
#endif

/* CANFD function pointers   */
/* g_canfd_on_canfd in the file "r_canfd_rx.c" */
const can_api_t g_canfd_on_canfd =
{
    .open           = R_CANFD_Open,
    .close          = R_CANFD_Close,
    .write          = R_CANFD_Write,
    .read           = R_CANFD_Read,
    .modeTransition = R_CANFD_ModeTransition,
    .infoGet        = R_CANFD_InfoGet,
    .callbackSet    = R_CANFD_CallbackSet,
```

```
};
```

2.12 Code Size

The ROM (code and constants) and RAM (global data) sizes are determined by the build-time configuration options described in 2.9 Configuration. The table lists reference values when the C compiler's compile options are set to their default values, as described in 2.5 Supported Toolchains. The compile option default values are optimization level: 2, optimization type: for size, and data endianness: little-endian. The code size varies depending on the C compiler version and compile options.

Area	Build Settings	Size (byte)		
		CCRX	GCC	IAR
ROM	With Parameter Checking	2533	4260	3210
ROM	Without Parameter Checking	2115	3404	2546
RAM		136	128	4
Maximum stack usage		304		

2.13 Callback Functions

In this module, a callback function set up by the user is called when either of the following conditions is met.

(1) Global interrupts:

- Receive FIFO interrupt.
- Global error interrupt: DLC Error Detect, Message Lost Detect, Payload Overflow Detect.

(2) Channel interrupts:

- Channel transmit interrupt: Successful transmission interrupt.
- Channel error interrupt: Error Warning Detect, Error Passive Detect, Bus-Off Entry Detect, Bus-Off Recovery Detect, Overload Detect.

The callback function is set up by storing the address of the user function in the p_callback argument of g_canfd0_cfg structure. The default value of the p_callback argument is NULL. User can change it into the user function by changing the value of the p_callback argument.

See example below to change the value of the p_callback argument from NULL to User_callback:

```
void User_callback(can_callback_arg_t *g_args);

void main(void)
{
      g_canfd0_cfg.p_callback = User_callback;
      R_CANFD_Open(&g_canfd0_ctrl, &g_canfd0_cfg);
}

void User_callback(can_callback_arg_t *g_args)
{
      User_program();
}
```

2.14 Adding the CAN FD 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.

2.15 “for”, “while” and “do while” statements

In this module, “for”, “while” and “do while” statements (loop processing) are used in processing to wait for register to be reflected and so on. For these loop processing, comments with “WAIT_LOOP” as a keyword are described. Therefore, if user incorporates fail-safe processing into loop processing, user can search the corresponding processing with “WAIT_LOOP”.

The following shows example of description.

```
while statement example:
/* WAIT_LOOP */
while(0 == SYSTEM.OSCOVFSR.BIT.PLOVF)
{
    /* The delay period needed is to make sure that the PLL has stabilized. */
}

for statement example:
/* Initialize reference counters to 0. */
/* WAIT_LOOP */
for (i = 0; i < BSP_REG_PROTECT_TOTAL_ITEMS; i++)
{
    g_protect_counters[i] = 0;
}

do while statement example:
/* Reset completion waiting */
do
{
    reg = phy_read(ether_channel, PHY_REG_CONTROL);
    count++;
} while ((reg & PHY_CONTROL_RESET) && (count < ETHER_CFG_PHY_DELAY_RESET)); /*
WAIT_LOOP */
```

3. API Functions

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

CAN FD configuration and communication are accomplished via the CAN FD SFR (Special Function Register) Registers described in the MCU's HW manual. As the registers in the CAN FD peripheral must be configured and read in the proper sequence to achieve useful communication, a CAN FD API greatly simplifies this. The API takes numerous tedious issues and does them for you.

After initializing the peripheral through the R_CANFD_Open function, all you need to do is use the receive(R_CANFD_Read) and transmit(R_CANFD_Write) API calls, and regularly check for any CAN FD error states. As well as you can close the CAN FD channel by the R_CANFD_Close function or switch to a different test mode through the R_CANFD_ModeTransition function

For details refer to below.

Summary

The following functions are included in this design:

Function Name	Description
R_CANFD_Open()	Open and configure the CAN FD channel for operation.
R_CANFD_Close()	Close the CAN FD channel.
R_CANFD_Write()	Write data to the CAN FD channel.
R_CANFD_Read()	Read data from a CAN FD Message Buffer or FIFO.
R_CANFD_ModeTransition()	Switch to a different channel, global or test mode.
R_CANFD_InfoGet()	Get CAN FD state and status information for the channel.
R_CANFD_CallbackSet()	Updates the user callback with the option to provide memory for the callback argument structure.

Return Codes

API Return Codes	Description
FSP_SUCCESS	Action completed successfully.
FSP_ERR_IP_CHANNEL_NOT_PRESENT	Requested channel does not exist on this device
FSP_ERR_ASSERTION	A critical assertion has failed
FSP_ERR_CAN_INIT_FAILED	Hardware initialization failed.
FSP_ERR_CLOCK_INACTIVE	Inactive clock specified as system clock.
FSP_ERR_CAN_TRANSMIT_NOT_READY	Transmit in progress.
FSP_ERR_INVALID_ARGUMENT	Invalid input parameter
FSP_ERR_INVALID_MODE	Unsupported or incorrect mode
FSP_ERR_NOT_OPEN	Requested channel is not configured or API not open
FSP_ERR_IN_USE	Channel/peripheral is running/busy
FSP_ERR_ALREADY_OPEN	Requested channel is already open in a different configuration
FSP_ERR_NO_CALLBACK_MEMORY	Non-secure callback memory not provided for non-secure callback
FSP_ERR_BUFFER_EMPTY	No data available in buffer

R_CANFD_Open

Open and configure the CAN FD channel for operation.

Format

```
fsp_err_t R_CANFD_Open (can_ctrl_t * const p_api_ctrl,
                        can_cfg_t const * const p_cfg);
```

Parameters

p_api_ctrl

Pointer to the CAN control block. Must be declared by user.
Consult Section 2.11 Instance Structure for details.

p_cfg

Pointer to CAN configuration structure. All elements of this structure must be set by user.
Consult Section 2.11 Instance Structure for details.

Return Values

FSP_SUCCESS	*Channel opened successfully.*
FSP_ERR_ALREADY_OPEN	*Driver already open.*
FSP_ERR_IN_USE	*Channel is already in use.*
FSP_ERR_IP_CHANNEL_NOT_PRESENT	*Channel does not exist on this MCU.*
FSP_ERR_ASSERTION	*A required pointer was NULL.*
FSP_ERR_CAN_INIT_FAILED	*The provided nominal or data bitrate is invalid.*
FSP_ERR_CLOCK_INACTIVE	*CAN FD source clock is disabled (PLL or PLL2).*

Properties

Prototyped in *r_canfd.h*
Implemented in *r_canfd_rx.c*

Description

Open and configure the CAN FD channel for operation.

Example

```
/* Initialize the CAN FD module */
    R_CANFD_Open(&g_canfd0_ctrl, &g_canfd0_cfg)
```

R_CANFD_Close

Close the CAN FD channel.

Format

```
fsp_err_t R_CANFD_Close (can_ctrl_t *const p_api_ctrl);
```

Parameters

p_api_ctrl

Pointer to the CAN control block.
Consult Section 2.11 Instance Structure for details.

Return Values

FSP_SUCCESS	*Channel closed successfully.*
FSP_ERR_NOT_OPEN	*Control block not open.*
FSP_ERR_ASSERTION	*Null pointer presented.*

Properties

Prototyped in *r_canfd.h*
Implemented in *r_canfd_rx.c*

Description

Close the CAN FD channel.

Example

```
/* Close the CAN FD module */
   R_CANFD_Close(&g_canfd0_ctrl);
```

R_CANFD_Write

Write data to the CAN FD channel.

Format

```
fsp_err_t R_CANFD_Write (can_ctrl_t *const p_api_ctrl,
                         uint32_t buffer,
                         can_frame_t *const p_frame);
```

Parameters

p_api_ctrl
Pointer to the CAN control block.
Consult Section 2.11 Instance Structure for details.

buffer
Buffer number (mailbox or message buffer) to write to.

p_frame
Pointer for frame of CAN ID, DLC, data and frame type to write.

Return Values

FSP_SUCCESS	*Operation succeeded.*
FSP_ERR_NOT_OPEN	*Control block not open.*
FSP_ERR_CAN_TRANSMIT_NOT_READY	*Transmit in progress, cannot write data at this time.*
FSP_ERR_INVALID_ARGUMENT	*Data length or buffer number invalid.*
FSP_ERR_INVALID_MODE	*An FD option was set on a non-FD frame.*
FSP_ERR_ASSERTION	*Null pointer presented*

Properties

Prototyped in *r_canfd.h*
Implemented in *r_canfd_rx.c*

Description

Write data to the CAN FD channel.

Example

```
#define CAN_BUFFER_NUMBER_0             (0U)                //buffer number
can_frame_t g_canfd_tx_frame;                          //CAN FD transmit frame

/* Fill tx frame data that is to be sent*/
   for( uint16_t j = 0; j < SIZE_8; j++)
   {
     g_canfd_tx_frame.data[j] = (uint8_t) (j + 1);
   }

/* Send data on the bus */
   err = R_CANFD_Write(&g_canfd0_ctrl, CAN_BUFFER_NUMBER_0, &g_canfd_tx_frame);
```

R_CANFD_Read

Read data from a CAN FD Message Buffer or FIFO.

Format

```
fsp_err_t R_CANFD_Read (can_ctrl_t *const p_api_ctrl, uint32_t buffer,
                        can_frame_t *const p_frame);
```

Parameters

p_api_ctrl

Pointer to the CAN control block.

Consult Section 2.11 Instance Structure for details.

buffer

Message buffer (number) to read from.

p_frame

Pointer to store the CAN ID, DLC, data and frame type.

Return Values

FSP_SUCCESS	*Operation succeeded.*
FSP_ERR_NOT_OPEN	*Control block not open.*
FSP_ERR_INVALID_ARGUMENT	*Buffer number invalid.*
FSP_ERR_ASSERTION	*p_api_ctrl or p_frame is NULL.*
FSP_ERR_BUFFER_EMPTY	*Buffer or FIFO is empty.*

Properties

Prototyped in *r_canfd.h*
Implemented in *r_canfd_rx.c*

Description

Read data from a CAN FD Message Buffer or FIFO.

Example

```
#define ZERO (0U)
can_frame_t g_canfd_rx_frame;

/* Read the input frame received */
   err = R_CANFD_Read(&g_canfd0_ctrl, ZERO, &g_canfd_rx_frame);
```

R_CANFD_ModeTransition

Switch to a different channel, global or test mode.

Format

```
fsp_err_t R_CANFD_ModeTransition (can_ctrl_t *const p_api_ctrl,
                                  can_operation_mode_t operation_mode,
                                  can_test_mode_t test_mode);
```

Parameters

p_api_ctrl

Pointer to the CAN control block.
Consult Section 2.11 Instance Structure for details.

operation_mode

Destination CAN FD operation state.

test_mode

Destination CAN FD test state.

Return Values

FSP_SUCCESS	*Operation succeeded.*
FSP_ERR_NOT_OPEN	*Control block not open.*
FSP_ERR_ASSERTION	*Null pointer presented.*
FSP_ERR_INVALID_MODE	*Cannot change to the requested mode from the current global mode.*

Properties

Prototyped in *r_canfd.h*
Implemented in *r_canfd_rx.c*

Description

Switch to a different channel, global or test mode.

Example

```
/* Switch to external loopback mode */
   R_CANFD_ModeTransition(&g_canfd0_ctrl, CAN_OPERATION_MODE_NORMAL,
(can_test_mode_t) CAN_TEST_MODE_LOOPBACK_EXTERNAL);
```

R_CANFD_InfoGet

Get CAN FD state and status information for the channel.

```
fsp_err_t R_CANFD_InfoGet (can_ctrl_t *const p_api_ctrl,
                           can_info_t *const p_info);
```

Parameters

p_api_ctrl

Handle for channel (pointer to channel control block)
Consult Section 2.11 Instance Structure for details.

p_info

Memory address to return channel specific data to.

Return Values

FSP_SUCCESS	*Operation succeeded.*
FSP_ERR_NOT_OPEN	*Control block not open.*
FSP_ERR_ASSERTION	*Null pointer presented.*

Properties

Prototyped in *r_canfd.h*
Implemented in *r_canfd_rx.c*

Description

Get CAN FD state and status information for the channel.

Example

```
#define RESET_VALUE             (0x00)
/* Variable to store rx frame status info*/
 can_info_t can_rx_info =
 {
  .error_code  = RESET_VALUE,
  .error_count_receive = RESET_VALUE,
  .error_count_transmit = RESET_VALUE,
  .rx_fifo_status = RESET_VALUE,
  .rx_mb_status = 1,
  .status = RESET_VALUE,
 };

/* Get CAN FD status*/
   R_CANFD_InfoGet(&g_canfd0_ctrl, &can_rx_info);
```

R_CANFD_CallbackSet

Updates the user callback with the option to provide memory for the callback argument structure.
Implements can_api_t::callbackSet..

Format

```
fsp_err_t R_CANFD_CallbackSet (can_ctrl_t *const p_api_ctrl,
                               void(*)(can_callback_args_t *) p_callback,
                               void const *const p_context,
                               can_callback_args_t *const p_callback_memory);
```

Parameters

p_api_ctrl
Control block set in can_api_t::open call.
Consult Section 2.11 Instance Structure for details.

p_callback
Callback function to register

p_context
Pointer to send to callback function

p_callback_memory
Pointer to volatile memory where callback structure can be allocated. Callback arguments allocated here are only valid during the callback.

Return Values

FSP_SUCCESS	*Callback updated successfully.*
FSP_ERR_ASSERTION	*A required pointer is NULL.*
FSP_ERR_NOT_OPEN	*The control block has not been opened.*
FSP_ERR_NO_CALLBACK_MEMORY	*p_callback is non-secure and p_callback_memory is either secure or NULL.*

Properties

Prototyped in *r_canfd.h*
Implemented in *r_canfd_rx.c*

Description

Updates the user callback with the option to provide memory for the callback argument structure.

Example

```
/* Config callback function */
   R_CANFD_CallbackSet(&g_canfd0_ctrl, canfd_callback, NULL, NULL);
```

Example

Basic Example

This is a basic example of minimal use of the CAN FD module in an application. It is implemented with classic CAN. If have a new message coming, the program will read it. Or the User can press sw2 to send a message to a CAN bus.

Note

It is recommended to use RX FIFOs for reception as there are no interrupts for RX message buffers in this software.

```
#define CAN_BUFFER_NUMBER_0            (0U)               //buffer number
#define ZERO (0U)
#define CAN_ID                    (0x1100) //ID of transmit frame
#define CAN_CLASSIC_FRAME_DATA_BYTES (8U) //Data Length code for classic frame
#define SIZE_8   (8u)
extern can_bit_timing_cfg_t g_canfd0_bit_timing_cfg; /* extern to change default
value */
can_frame_t g_canfd_tx_frame;                       //CAN FD transmit frame
can_frame_t g_canfd_rx_frame;
#define RESET_VALUE             (0x00)

/* Variable to store rx frame status info*/
 can_info_t can_rx_info =
 {
  .error_code  = RESET_VALUE,
  .error_count_receive = RESET_VALUE,
  .error_count_transmit = RESET_VALUE,
  .rx_fifo_status = RESET_VALUE,
  .rx_mb_status = 1,
  .status = RESET_VALUE,
 };
/* Acceptance filter array parameters
CANFD_CFG_AFL_CH0_RULE_NUM = 1 */
/* Acceptance filter array parameters */
#define CANFD_FILTER_ID (0x00001000)
#define MASK_ID                  (0x0FFFF000)
#define MASK_ID_MODE    (1)
#define ZERO                        (0U) //Array Index value
const canfd_afl_entry_t p_canfd0_afl[CANFD_CFG_AFL_CH0_RULE_NUM] =
{
    /* Accept a message with Extended ID 0x1000-0x1FFF */
    /* Specify the ID, ID type and frame type to accept. */
    {
      CANFD_FILTER_ID,
      0,
      CAN_FRAME_TYPE_DATA,
      CAN_ID_MODE_EXTENDED,
      MASK_ID,
      0,
      ZERO,
      MASK_ID_MODE,
      (canfd_minimum_dlc_t)ZERO,
      0,
      CANFD_RX_MB_0,
      0,
      CANFD_RX_FIFO_0
```

```
      },
};

void main(void)
{
    g_canfd0_extended_cfg.p_afl = p_canfd0_afl;
    /* Nominal rate: 1Mbps; DLL: 40M Hz. */
    g_canfd0_bit_timing_cfg.baud_rate_prescaler = 1;
    g_canfd0_bit_timing_cfg.synchronization_jump_width = 1;
    g_canfd0_bit_timing_cfg.time_segment_1 = 20;
    g_canfd0_bit_timing_cfg.time_segment_2 = 19;

    /* Fill tx frame data that is to be sent*/
    for( uint16_t j = 0; j < SIZE_8; j++)
    {
        g_canfd_tx_frame.data[j] = (uint8_t) (j + 1);
    }

    R_BSP_RegisterProtectDisable(BSP_REG_PROTECT_MPC);

    /* Set CRX0 pin */
    PORT3.PMR.BIT.B3 = 0U;
    PORT3.PDR.BIT.B3 = 0U;
    MPC.P33PFS.BYTE = 0x10U;
    PORT3.PMR.BIT.B3 = 1U;
    PORT3.PDR.BIT.B3 = 0U;

    /* Set CTX0 pin */
    PORT3.PMR.BIT.B2 = 0U;
    PORT3.PDR.BIT.B2 = 0U;
    MPC.P32PFS.BYTE = 0x10U;
    PORT3.PMR.BIT.B2 = 1U;
    PORT3.PDR.BIT.B2 = 1U;

    R_BSP_RegisterProtectEnable(BSP_REG_PROTECT_MPC);

    fsp_err_t err;
    /* Initialize the API. */
    err = R_CANFD_Open(&g_canfd0_ctrl, &g_canfd0_cfg);

while(1)
{
    /* Check whether having the new message... */
    can_read_operation();

    /* press sw2 to send a message to a CAN bus */
    read_switches();
}

/* Call sw2_func() when press sw2 */
void sw2_func(void)
{
    can_operation();
}/* end sw2_func() */

void can_operation(void)
{
```

```
    /* Update transmit frame parameters */
    g_canfd_tx_frame.id = CAN_ID;
    g_canfd_tx_frame.id_mode = CAN_ID_MODE_EXTENDED;
    g_canfd_tx_frame.type = CAN_FRAME_TYPE_DATA;

    /* Classic CAN 8 bytes */
    g_canfd_tx_frame.data_length_code = CAN_CLASSIC_FRAME_DATA_BYTES;
    g_canfd_tx_frame.options = ZERO;

    /* Transmission of data over classic CAN frame */
    can_write_operation(g_canfd_tx_frame);
}

static void can_write_operation(can_frame_t can_transmit_frame)
{
    fsp_err_t err = FSP_SUCCESS;

    /* Transmit the data from buffer #0 with tx_frame */
    err = R_CANFD_Write(&g_canfd0_ctrl, CAN_BUFFER_NUMBER_0,
&can_transmit_frame);
}

void can_read_operation(void)
{
    fsp_err_t err = FSP_SUCCESS;

    /* Get the status information for CAN FD transmission */
    err = R_CANFD_InfoGet(&g_canfd0_ctrl, &can_rx_info);

    /* Check if the data is received in FIFO */
    if(can_rx_info.rx_mb_status)
    {
        /* Read the input frame received */
        err = R_CANFD_Read(&g_canfd0_ctrl, ZERO, &g_canfd_rx_frame);
    }
}
```

Flexible Data

This example demonstrates sending an FD message with bitrate switching (Nominal rate = 1Mbps, Data rate = 8Mbps). If have a new message coming, the program will read it. Or the User can press switch 2 to send a message to a CAN bus

```
#define CAN_BUFFER_NUMBER_0            (0U)               //buffer number
#define ZERO (0U)
#define CAN_ID                    (0x1100) //ID of transmit frame
#define CAN_FD_DATA_LENGTH_CODE (64U)   //Data Length code for classic frame
#define SIZE_64      (64u)
extern can_bit_timing_cfg_t g_canfd0_bit_timing_cfg; /* extern to change default
value */
extern can_bit_timing_cfg_t g_canfd0_data_timing_cfg; /* extern to change
default value */
can_frame_t g_canfd_tx_frame;                       //CAN FD transmit frame
can_frame_t g_canfd_rx_frame;
#define RESET_VALUE             (0x00)

/* Variable to store rx frame status info*/
```

```
 can_info_t can_rx_info =
 {
  .error_code  = RESET_VALUE,
  .error_count_receive = RESET_VALUE,
  .error_count_transmit = RESET_VALUE,
  .rx_fifo_status = RESET_VALUE,
  .rx_mb_status = 1,
  .status = RESET_VALUE,
 };
/* Acceptance filter array parameters
CANFD_CFG_AFL_CH0_RULE_NUM = 1 */
/* Acceptance filter array parameters */
#define CANFD_FILTER_ID (0x00001000)
#define MASK_ID                 (0x0FFFF000)
#define MASK_ID_MODE    (1)
#define ZERO                        (0U) //Array Index value
const canfd_afl_entry_t p_canfd0_afl[CANFD_CFG_AFL_CH0_RULE_NUM] =
{
    /* Accept a message with Extended ID 0x1000-0x1FFF */
    /* Specify the ID, ID type and frame type to accept. */
    {
      CANFD_FILTER_ID,
      0,
      CAN_FRAME_TYPE_DATA,
      CAN_ID_MODE_EXTENDED,
      MASK_ID,
      0,
      ZERO,
      MASK_ID_MODE,
      (canfd_minimum_dlc_t)ZERO,
      0,
      CANFD_RX_MB_0,
      0,
      CANFD_RX_FIFO_0
      },
};

void main(void)
{
    g_canfd0_extended_cfg.p_afl = p_canfd0_afl;
    /* Nominal rate: 1Mbps; DLL: 40M Hz. */
    g_canfd0_bit_timing_cfg.baud_rate_prescaler = 1;
    g_canfd0_bit_timing_cfg.synchronization_jump_width = 1;
    g_canfd0_bit_timing_cfg.time_segment_1 = 20;
    g_canfd0_bit_timing_cfg.time_segment_2 = 19;

    /* Data rate: 8Mbps;  DLL: 40M Hz. */
    g_canfd0_data_timing_cfg.baud_rate_prescaler = 1;
    g_canfd0_data_timing_cfg.synchronization_jump_width = 1;
    g_canfd0_data_timing_cfg.time_segment_1 = 2;
    g_canfd0_data_timing_cfg.time_segment_2 = 2;

    /* Fill tx frame data that is to be sent*/
    for( uint16_t j = 0; j < SIZE_64; j++)
    {
        g_canfd_tx_frame.data[j] = (uint8_t) (j + 1);
    }
```

```
    R_BSP_RegisterProtectDisable(BSP_REG_PROTECT_MPC);

    /* Set CRX0 pin */
    PORT3.PMR.BIT.B3 = 0U;
    PORT3.PDR.BIT.B3 = 0U;
    MPC.P33PFS.BYTE = 0x10U;
    PORT3.PMR.BIT.B3 = 1U;
    PORT3.PDR.BIT.B3 = 0U;

    /* Set CTX0 pin */
    PORT3.PMR.BIT.B2 = 0U;
    PORT3.PDR.BIT.B2 = 0U;
    MPC.P32PFS.BYTE = 0x10U;
    PORT3.PMR.BIT.B2 = 1U;
    PORT3.PDR.BIT.B2 = 1U;

    R_BSP_RegisterProtectEnable(BSP_REG_PROTECT_MPC);

    fsp_err_t err;
    /* Initialize the API. */
    err = R_CANFD_Open(&g_canfd0_ctrl, &g_canfd0_cfg);

while(1)
{
    /* Check whether having the new message... */
    can_read_operation();

    /* press sw2 to send a message to a CAN bus */
    read_switches();
}

/* Call sw2_func() when press sw2 */
void sw2_func(void)
{
    canfd_operation();
}/* end sw2_func() */

void canfd_operation(void)
{

    /* Update transmit frame parameters */
    g_canfd_tx_frame.id = CAN_ID;
    g_canfd_tx_frame.id_mode = CAN_ID_MODE_EXTENDED;
    g_canfd_tx_frame.type = CAN_FRAME_TYPE_DATA;

    /* FD CAN 64bytes*/
    g_canfd_tx_frame.data_length_code = CAN_FD_DATA_LENGTH_CODE;
    g_canfd_tx_frame.options = CANFD_FRAME_OPTION_FD | CANFD_FRAME_OPTION_BRS;

    /* Transmission of data over FD CAN frame */
    can_write_operation(g_canfd_tx_frame);
}
void can_read_operation(void)
{
    fsp_err_t err = FSP_SUCCESS;

    /* Get the status information for CAN FD transmission */
    err = R_CANFD_InfoGet(&g_canfd0_ctrl, &can_rx_info);
```

```
    /* Check if the data is received in FIFO */
    if(can_rx_info.rx_mb_status)
    {
        /* Read the input frame received */
        err = R_CANFD_Read(&g_canfd0_ctrl, ZERO, &g_canfd_rx_frame);
    }
}
static void can_write_operation(can_frame_t can_transmit_frame)
{
    fsp_err_t err = FSP_SUCCESS;

    /* Transmit the data from buffer #0 with tx_frame */
    err = R_CANFD_Write(&g_canfd0_ctrl, CAN_BUFFER_NUMBER_0,
&can_transmit_frame);
}
```

4. Pin Setting

To use the CAN FD FIT module, assign input/output signals of the peripheral function to pins with the multi-function pin controller (MPC). The pin assignment is referred to as the “Pin Setting” in this document.

Please perform the pin setting after calling the R_CANFD_Open function.

When performing the pin setting in the e^2 studio, the Pin Setting feature of the FIT Configurator or the Smart Configurator can be used. When using the Pin Setting feature, a source file is generated according to the option selected in the Pin Setting window in the FIT Configurator or the Smart Configurator. Then pins are configured by calling the function defined in the source file. Refer to Table 4.1 Function Output by the FIT Configurator for details.

Table 4.1 Function Output by the Smart Configurator

MCU Used	Function to be Output	Remarks
All MCUs	R_CANFD_PinSet_CANFDx	x: Channel number

5. Demo Projects

Demo projects include function main() that utilizes the FIT module and its dependent modules (e.g. r_bsp). This FIT module includes the following demo projects.

5.1 Adding a Demo to a Workspace

Currently, sample program is not supported

5.2 The Renesas Debug Console

Currently, sample program is not supported

6. Test Modes

The CANFD module can be configured into test modes to allow testing of certain features. These features are provided only for special purposes and care must be taken when configuring the CANFD module in the test modes.
The test modes can be broadly split into two groups:

- Channel specific test modes
- Global test mode (the current source code does not support global test mode)

6.1 Channel Specific Test Mode

CAN FD channel can be configured into following test modes:

- Basic test mode
- Listen-only mode
- External loop back mode
- Internal loop back mode
- Restricted operation mode (the current source code does not support this test mode)

Use R_CANFD_ModeTransition to switch to a test mode.

6.1.1 Basic test mode

The basic test mode should be used when a particular test setting needs to be enabled other than when in listen-only and self-test modes.

6.1.2 Listen Only mode = 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 FD 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.

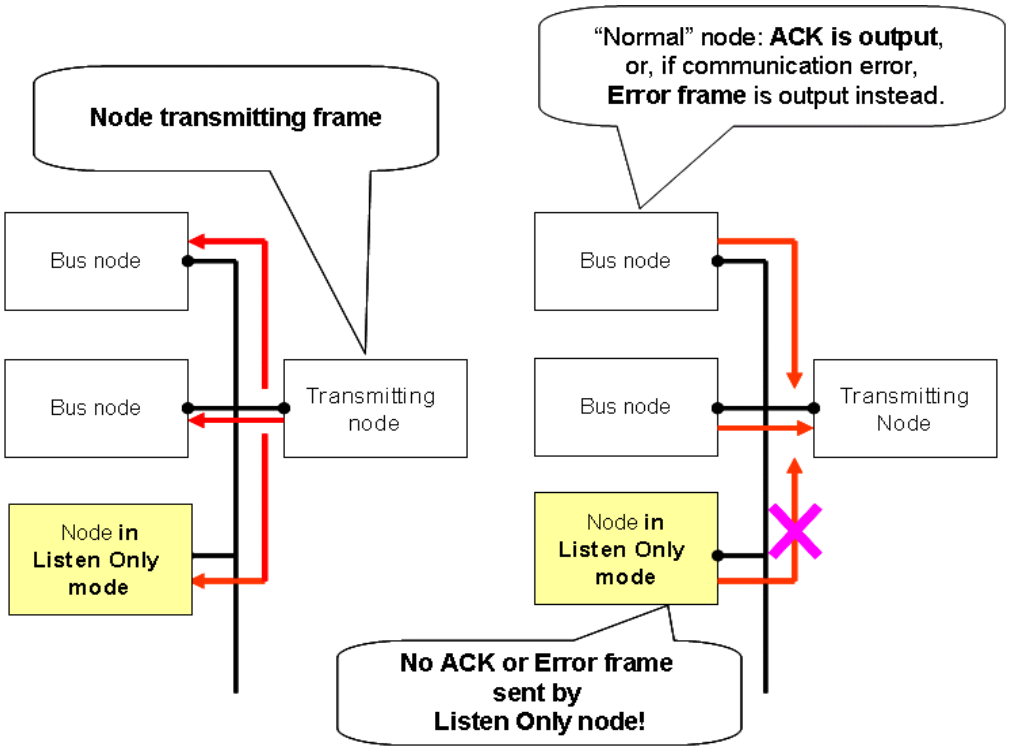

Figure 1.

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.

6.1.3 Loopback

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

6.1.3.1 Internal loopback mode - Test node without CAN bus

Internal Loopback mode, or Self-Test mode, allows you to communicate via the CAN FD buffers 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 buffer if it was configured for that CAN FD ID. This is normally not possible.

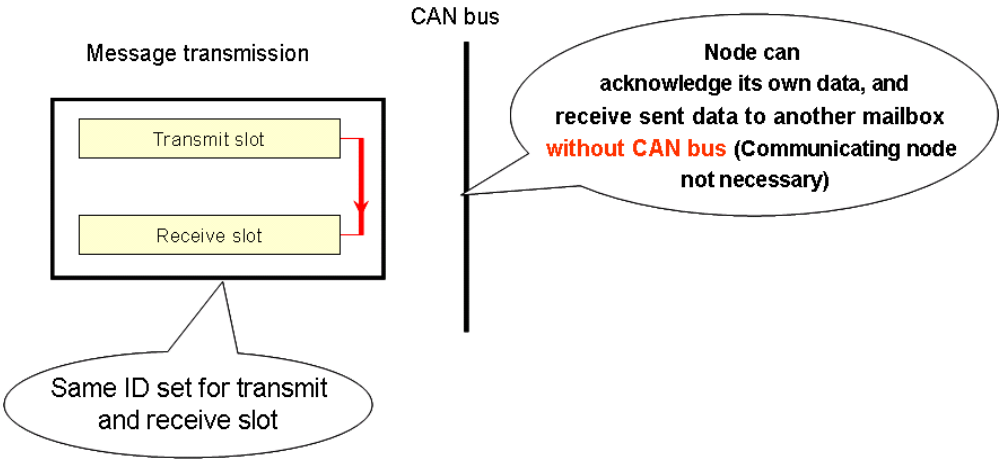

Figure 2.

CAN FD Internal Loopback mode let 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 FD controller to run without sending CAN FD errors due to no ACKs received when the node is alone on the bus, it acknowledges transmitted frames itself.

6.1.3.2 External loopback mode - 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.

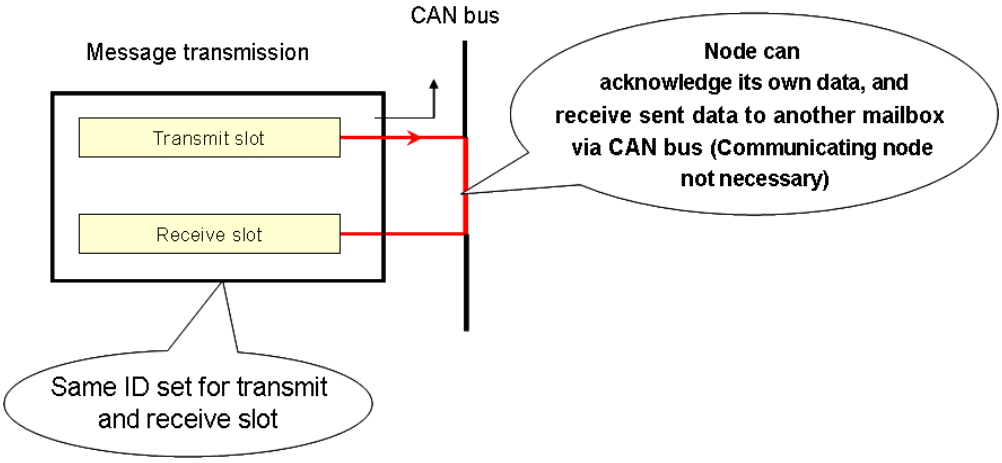

Figure 3. External Loopback.

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.

6.1.4 Restricted operation

The current source code does not support restricted operation mode.

6.2 Global test mode enable register

The current source code does not support global test mode.

7. Appendices

7.1 Confirmed Operation Environment

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

Table 7.1 Confirmed Operation Environment (Rev.1.00)

Item	Contents
Integrated development environment	Renesas Electronics e^2 studio Version 22.4.0 IAR Embedded Workbench for Renesas RX 4.20.3
C compiler	Renesas Electronics C/C++ Compiler Package for RX Family V3.04.00 Compiler option: The following option is added to the default settings of the integrated development environment. -lang = c99
	GCC for Renesas RX 8.3.0.202104 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 4.20.3 Compiler option: The default settings of the integrated development environment.
Endian	Big endian/little endian
Revision of the module	Rev.1.00
Board used	Renesas Starter Kit for RX660 (product number. RTK556609HC10000BJ)

7.2 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_canfd_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_canfd_rx_config.h” may be wrong. Check the file “r_canfd_rx_config.h”. If there is a wrong setting, set the correct value for that. Refer to 2.9 Configuration for details.

Related Technical Updates

This module reflects the content of the following technical updates.

None

Revision History

		Description	
Rev.	Date	Page	Summary
1.00	May.31.2022	—	First release.

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 be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. 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.
6. 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.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. 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.5.0-1 October 2020)

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/.