RZ/A1H Group

RSCAN Driver Module

R01AN2784EU0102
Rev. 1.02
Apr 4, 2016

Introduction

This document describes the API for the RSCAN driver for the RZ/A1. The driver supports all channels on the peripheral. Message transfers can be done using 1-message deep mailboxes, 16-message deep FIFOs, or any combination thereof.

NOTE: This driver has only had basic testing performed on it. This includes simple mailbox, FIFO (non-Gateway), interrupt, and Error Passive State detection and recovery operations.

Target Device

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

- RZ/A1H Group

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

Related Documents

- RZ/A1 Hardware User's Manual (R01UH0403EJ)

Contents

1. Overview

This driver provides support for all five channels of the RSCAN peripheral. A static configuration of mailboxes and FIFOs (boxes) is used to simplify the API design and its usage.

All mailboxes are one-message deep. There are 16 transmit mailboxes for each channel, and 16 receive mailboxes in total. The transmit mailboxes can optionally be configured for interrupt operation, whereas the receive mailboxes cannot. The transmit mailboxes do not accept a message for transmit until the previous message has been sent. The receive mailboxes always contain the most recent message received, overwriting the previous contents without an error condition being generated. There is no hardware interrupt option available.

All FIFOs are 16-messages deep. FIFOs are used for the sending and receiving of messages just like a mailbox. These can optionally be configured to be interrupt driven. Setting a receive FIFO to interrupt on every message received would behave similar to a receive mailbox with interrupt support.

There are two types of special FIFOs. One is a Gateway FIFO. This is used for bridging networks. It automatically retransmits every message it receives without CPU intervention (all done within peripheral hardware). The History FIFO logs all messages tagged in an R_CAN_SendMsg() call in the order they are sent. Note that any FIFO usage is optional and they are not required for normal operation.

The RSCAN hardware processes all messages transmitted on the bus, but uses Receive Rules to determine which messages to keep and which to ignore. A Receive Rule consists of two parts. The first part performs filtering on different parts of the message to see if the message should be kept. The second part specifies which box (receive mailbox or receive FIFO) to route the message to. After the hardware routes a message to a box, the function R_CAN_GetMsg() is used to read a message from the box.

There are two types of interrupts available- global interrupts and channel interrupts. The global interrupts indicate when a receive FIFO has received a message as well as when a global error occurs. These interrupts are enabled in the r_rscan_rz_config.h file. The driver detects the interrupt and calls a user callback function specified in R_CAN_Open() to process the particular event(s). The channel interrupts handle several transmit conditions as well as channel errors. These interrupts are also enabled in the r_rscan_rz_config.h file. The driver detects the interrupt and calls a user callback function specified in R_CAN_InitChan() to process the particular event(s).

By default, the following interrupts are enabled:

- RX, TX, or History FIFO threshold reached
- RX, TX, Gateway, or History FIFO overflow occurred
- Channel entered Error Passive state
- Channel entered Bus Off state
- Channel recovered from Bus Off state

The following sequence of function calls is used to setup the CAN:

```
R_CAN_Open();
R_CAN_InitChan();    // do for 1-5 channels
R_CAN_ConfigFIFO(); // do for 0 or more FIFOs
R_CAN_AddRxRule();  // do for 1-320 rules
```

Once the CAN is setup, the peripheral should enter normal communications mode or a test mode.

```
R_CAN_Control();    // Use CAN_CMD_SET_MODE_COMM or CAN_CMD_SET_MODE_TST_xxx
```

2. API Information

This Driver API follows the Renesas API naming standards.

2.1 Hardware Requirements

This driver utilizes the RSCAN peripheral.

2.2 Hardware Resource Requirements

In addition to the RSCAN peripheral, the driver requires:

- Two pins allocated for each CAN channel used

2.3 Software Requirements

This driver is dependent upon

- The R_INTC software provided with the RSK+RZA1H board

2.4 Limitations

Not all features of the peripheral are utilized. These include:

- Transmit queues
- Transmit complete interrupt on or off for each transmit mailbox (all on or off for all channels)
- Configurable depth transmit, receive, and gateway FIFOs (all fixed at 16 instead of configurable 1 to 128)
- Transmit by message ID priority (will be done by mailbox number, 0 being highest priority)
- Transmit FIFO interval transmission
- Transmit mirroring
- Filter on mirrored messages
- DLC substitution
- Multiple destinations for each received message (will fix at 1 destination; could be up to 8)
- Different methods of Bus Off recovery (will be ISO11898-1 compliant)
- Forcible return from Bus Off
- Different interrupt sources for each channel (same settings applied to all)
- Selection of protocol error flag accumulation vs first occurrence (will hard-code to accumulative for all channels)

2.5 Supported Toolchains

This driver is tested and working with the following toolchains:

- KPIT GNUARM-NONE-EABI Toolchain v14.02

2.6 Header Files

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

Build-time configuration options are set in the file "r_rscan_rz_config.h” (the default values are defined in the file “r_rscan_rz_config_reference.h”).

Both of these files should be included by the user’s application.

2.7 Integer Types

This project uses ANSI C99 “Exact width integer types” in order to make the code clearer and more portable. These types are defined in stdint.h.

2.8 Configuration Overview

Static configuration options for this driver are set by the user via the file r_rscan_rz_config.h.

Configuration options in *r_rscan_rz_config.h*		
Equate	**Default Value**	**Description**
CAN_CFG_PARAM_CHECKING_ENABLE	1	Setting to 0 removes parameter checking from the code. Setting to 1 includes parameter checking in the code.
CAN_CFG_CLOCK_SOURCE	0	If this equate is 0, the CAN clock source is ½ the peripheral clock speed (clkc). If this equate is 1, the source is the external CAN_CLOCK (clk_xincan).
CAN_CFG_INT_PRIORITY	5	Priority level for all CAN interrupts (0-31)
CAN_CFG_INT_RXFIFO_THRESHOLD	1	Setting to 0 disables interrupt when an RXFIFO threshold is reached. Setting to 1 enables interrupt. Requires FIFO to be initialized via R_CAN_ConfigFIFO(). CAN_EVT_RXFIFO_THRESHOLD is passed to the main callback function.
CAN_CFG_INT_DLC_ERR	0	Setting to 0 disables interrupt when a DLC error is detected. Setting to 1 enables interrupt. CAN_EVT_GLOBAL_ERR is passed to the main callback function.
CAN_CFG_INT_FIFO_OVFL	1	Setting to 0 disables interrupt when a TX, GW, or RX FIFO overflows. Setting to 1 enables interrupt. Requires FIFO to be initialized via R_CAN_ConfigFIFO(). CAN_EVT_ GLOBAL_ERR is passed to the main callback function.
CAN_CFG_INT_HIST_FIFO_OVFL	1	Setting to 0 disables interrupt when a History FIFO overflows. Setting to 1 enables interrupt. Requires FIFO to be initialized via R_CAN_ConfigFIFO(). CAN_EVT_ GLOBAL_ERR is passed to the main callback function.
CAN_CFG_INT_TXFIFO_THRESHOLD	1	Setting to 0 disables interrupt when a TXFIFO threshold is reached. Setting to 1 enables interrupt. Requires FIFO to be initializes via R_CAN_ConfigFIFO(). CAN_EVT_TRANSMIT is passed to the channel callback function.
CAN_CFG_INT_GWFIFO_RX_THRESHOLD	0	Setting to 0 disables interrupt when the GWFIFO receive threshold is reached. Setting to 1 enables interrupt. Requires FIFO to be initialized via R_CAN_ConfigFIFO().

		CAN_EVT_GATEWAY_RX is passed to the channel callback function.
CAN_CFG_INT_GWFIFO_TX_THRESHOLD	0	Setting to 0 disables interrupt when the GWFIFO transmit threshold is reached. Setting to 1 enables interrupt. Requires FIFO to be initialized via R_CAN_ConfigFIFO(). CAN_EVT_ TRANSMIT is passed to the channel callback function.
CAN_CFG_INT_HIST_FIFO_THRESHOLD	1	Setting to 0 disables interrupt when the HIST_FIFO threshold is reached. Setting to 1 enables interrupt. Requires FIFO to be initialized via R_CAN_ConfigFIFO(). CAN_EVT_ TRANSMIT is passed to the channel callback function.
CAN_CFG_INT_MBX_TX_COMPLETE	0	Setting to 0 disables interrupt when the mailbox completes transmission. Setting to 1 enables interrupt. CAN_EVT_ TRANSMIT is passed to the channel callback function.
CAN_CFG_INT_MBX_TX_ABORTED	0	Setting to 0 disables interrupt when the mailbox transmit is aborted. Setting to 1 enables interrupt. CAN_EVT_ TRANSMIT is passed to the channel callback function.
CAN_CFG_INT_BUS_ERROR	0	Setting to 0 disables interrupt when a bus error is detected. Setting to 1 enables interrupt. CAN_EVT_CHANNEL_ERR is passed to the channel callback function.
CAN_CFG_INT_ERR_WARNING	0	Setting to 0 disables interrupt when an error warning is detected. Setting to 1 enables interrupt. CAN_EVT_CHANNEL_ERR is passed to the channel callback function.
CAN_CFG_INT_ERR_PASSIVE	1	Setting to 0 disables interrupt when an error passive is detected. Setting to 1 enables interrupt. CAN_EVT_CHANNEL_ERR is passed to the channel callback function.
CAN_CFG_INT_BUS_OFF_ENTRY	1	Setting to 0 disables interrupt when a Bus Off error is detected. Setting to 1 enables interrupt. CAN_EVT_CHANNEL_ERR is passed to the channel callback function.
CAN_CFG_INT_BUS_OFF_RECOVERY	1	Setting to 0 disables interrupt when a Bus Off recovery is detected. Setting to 1 enables interrupt. CAN_EVT_CHANNEL_ERR is passed to the channel callback function.
CAN_CFG_INT_OVERLOAD_FRAME_TX	0	Setting to 0 disables interrupt when an overload is detected. Setting to 1 enables interrupt. CAN_EVT_CHANNEL_ERR is passed to the channel callback function.
CAN_CFG_INT_BUS_LOCK	0	Setting to 0 disables interrupt when a bus lock is detected. Setting to 1 enables interrupt. CAN_EVT_CHANNEL_ERR is passed to the channel callback function.

CAN_CFG_INT_ARB_LOST	0	Setting to 0 disables interrupt when arbitration loss is detected. Setting to 1 enables interrupt. CAN_EVT_CHANNEL_ERR is passed to the channel callback function.
CAN_CFG_NUM_RULES_CH0	0	Set to the number of receive rules for channel 0 (0-128). 320 max for system.
CAN_CFG_NUM_RULES_CH1	1	Set to the number of receive rules for channel 1 (0-128). 320 max for system.
CAN_CFG_NUM_RULES_CH2	0	Set to the number of receive rules for channel 2 (0-128). 320 max for system.
CAN_CFG_NUM_RULES_CH3	0	Set to the number of receive rules for channel 3 (0-128) 320 max for system.
CAN_CFG_NUM_RULES_CH4	0	Set to the number of receive rules for channel 4 (0-128) 320 max for system.

Table 1: Info about the configuration

2.9 Code Size

The code size is based on the default settings for the GNUARM-NONE-EABI compiler. **These code sizes include all interrupt handlers for all channels (17 ISRs).**

ROM and RAM code sizes		
	With Parameter Checking	**Without Parameter Checking**
RZ/A1	ROM: 15,344 bytes code	ROM: 13,612 bytes code
	RAM: 94 bytes	RAM: 94 bytes

Table 2: ROM and RAM code size

2.10 API Data Types

This section details the data types that are used with the driver's API functions.

2.10.1 Box IDs (mailboxes and FIFOs)

```
typedef enum e can box
{
    CAN_BOX_CH0_TXMBX_0       = (CAN_FLG_TXMBX | 0),
    CAN_BOX_CH0_TXMBX_1       = (CAN_FLG_TXMBX | 1),
    CAN_BOX_CH0_TXMBX_2       = (CAN_FLG_TXMBX | 2),
          :
    CAN_BOX_CH4_TXMBX_13      = (CAN_FLG_TXMBX | 77),
    CAN_BOX_CH4_TXMBX_14      = (CAN_FLG_TXMBX | 78),
    CAN_BOX_CH4_TXMBX_15      = (CAN_FLG_TXMBX | 79),

    CAN_BOX_RXMBX_0           = (CAN_FLG_RXMBX | 0),
    CAN_BOX_RXMBX_1           = (CAN_FLG_RXMBX | 1),
    CAN_BOX_RXMBX_2           = (CAN_FLG_RXMBX | 3),
          :
    CAN_BOX_RXMBX_13          = (CAN_FLG_RXMBX | 13),
    CAN_BOX_RXMBX_14          = (CAN_FLG_RXMBX | 14),
    CAN_BOX_RXMBX_15          = (CAN_FLG_RXMBX | 15),

    CAN_BOX_RXFIFO_0          = (CAN_FLG_FIFO | CAN_MASK_RXFIFO_0),
```

```
    CAN_BOX_RXFIFO_1          = (CAN_FLG_FIFO | CAN_MASK_RXFIFO_1),
    CAN_BOX_RXFIFO_2          = (CAN_FLG_FIFO | CAN_MASK_RXFIFO_2),
    CAN_BOX_RXFIFO_3          = (CAN_FLG_FIFO | CAN_MASK_RXFIFO_3),
    CAN_BOX_RXFIFO_4          = (CAN_FLG_FIFO | CAN_MASK_RXFIFO_4),
    CAN_BOX_RXFIFO_5          = (CAN_FLG_FIFO | CAN_MASK_RXFIFO_5),
    CAN_BOX_RXFIFO_6          = (CAN_FLG_FIFO | CAN_MASK_RXFIFO_6),
    CAN_BOX_RXFIFO_7          = (CAN_FLG_FIFO | CAN_MASK_RXFIFO_7),

    CAN_BOX_CH0_TXFIFO_0      = (CAN_FLG_FIFO | CAN_MASK_CH0_TXFIFO_0),
    CAN_BOX_CH0_TXFIFO_1      = (CAN_FLG_FIFO | CAN_MASK_CH0_TXFIFO_1),
    CAN_BOX_CH0_GWFIFO        = (CAN_FLG_FIFO | CAN_MASK_CH0_GWFIFO),
        :
    CAN_BOX_CH4_TXFIFO_0      = (CAN_FLG_FIFO | CAN_MASK_CH4_TXFIFO_0),
    CAN_BOX_CH4_TXFIFO_1      = (CAN_FLG_FIFO | CAN_MASK_CH4_TXFIFO_1),
    CAN_BOX_CH4_GWFIFO        = (CAN_FLG_FIFO | CAN_MASK_CH4_GWFIFO),

    CAN_BOX_CH0_HIST_FIFO     = (CAN_FLG_FIFO | CAN_MASK_CH0_HIST_FIFO),
    CAN_BOX_CH1_HIST_FIFO     = (CAN_FLG_FIFO | CAN_MASK_CH1_HIST_FIFO),
    CAN_BOX_CH2_HIST_FIFO     = (CAN_FLG_FIFO | CAN_MASK_CH2_HIST_FIFO),
    CAN_BOX_CH3_HIST_FIFO     = (CAN_FLG_FIFO | CAN_MASK_CH3_HIST_FIFO),
    CAN_BOX_CH4_HIST_FIFO     = (CAN_FLG_FIFO | CAN_MASK_CH4_HIST_FIFO)
} can_box_t;
```

2.10.2 R_CAN_Open() Data Types

```
typedef enum e_can_timestamp_src
{
    CAN_TIMESTAMP_SRC_CH0_BIT_CLK = 0,
    CAN_TIMESTAMP_SRC_CH1_BIT_CLK = 1,
    CAN_TIMESTAMP_SRC_CH2_BIT_CLK = 2,
    CAN_TIMESTAMP_SRC_CH3_BIT_CLK = 3,
    CAN_TIMESTAMP_SRC_CH4_BIT_CLK = 4,
    CAN_TIMESTAMP_SRC_HALF_PCLK   = 5,
    CAN_TIMESTAMP_SRC_END_ENUM
} can_timestamp_src_t;

typedef enum e_can_timestamp_div
{
    CAN_TIMESTAMP_DIV_1     = 0,
    CAN_TIMESTAMP_DIV_2     = 1,
    CAN_TIMESTAMP_DIV_4     = 2,
    CAN_TIMESTAMP_DIV_8     = 3,
    CAN_TIMESTAMP_DIV_16    = 4,
    CAN_TIMESTAMP_DIV_32    = 5,
    CAN_TIMESTAMP_DIV_64    = 6,
    CAN_TIMESTAMP_DIV_128   = 7,
    CAN_TIMESTAMP_DIV_256   = 8,
    CAN_TIMESTAMP_DIV_512   = 9,
    CAN_TIMESTAMP_DIV_1024  = 10,
    CAN_TIMESTAMP_DIV_2048  = 11,
    CAN_TIMESTAMP_DIV_4096  = 12,
    CAN_TIMESTAMP_DIV_8192  = 13,
    CAN_TIMESTAMP_DIV_16384 = 14,
    CAN_TIMESTAMP_DIV_32768 = 15,
    CAN_TIMESTAMP_DIV_END_ENUM
} can_timestamp_div_t;

typedef struct st_can_cfg
{
    can_timestamp_src_t     timestamp_src;
```

```
    can_timestamp_div_t     timestamp_div;
} can_cfg_t;
```

2.10.3 Callback function events

```
typedef enum e_can_cb_evt        // callback function events
{
    // Main Callback Events
    CAN_EVT_RXFIFO_THRESHOLD,    // RX FIFO threshold
    CAN_EVT_GLOBAL_ERR,          // RX, GW, or Hist FIFO overflow, or DLC error

    // Channel Callback Events
    CAN_EVT_TRANSMIT,            // mbx tx complete or aborted,
                                 //   tx or history FIFO threshold
    CAN_EVT_GWFIFO_RX_THRESHOLD, // GW FIFO rx threshold
    CAN_EVT_CHANNEL_ERR,
} can_cb_evt_t;
```

2.10.4 R_CAN_InitChan() Data Types

```
typedef struct st_can_bitrate
{
    uint16_t    prescaler;  // 1-1024
    uint8_t     tseg1;      // 4-16
    uint8_t     tseg2;      // 2-8
    uint8_t     sjw;        // 1-4
} can_bitrate_t;

/* Sample settings for 500kbps with 12MHz XTAL (1/2 pclk =30MHz; 0% baud err) */
#define CAN_RSK_12MHZXTAL_500KBPS_PRESCALER     4
#define CAN_RSK_12MHZXTAL_500KBPS_TSEG1         11  // TSEG1 + TSEG2 + SJW = 15
#define CAN_RSK_12MHZXTAL_500KBPS_TSEG2         3
#define CAN_RSK_12MHZXTAL_500KBPS_SJW           1
```

2.10.5 R_CAN_ConfigFIFO() Data Types

```
typedef enum e_can_fifo_threshold       // NOTE: History FIFO can only have a
{                                       //       threshold of 1 or 12
    CAN_FIFO_THRESHOLD_2    = 0,        // 1/8 of 16
    CAN_FIFO_THRESHOLD_4    = 1,        // 2/8 of 16
    CAN_FIFO_THRESHOLD_6    = 2,        // 3/8 of 16
    CAN_FIFO_THRESHOLD_8    = 3,        // 4/8 of 16
    CAN_FIFO_THRESHOLD_10   = 4,        // 5/8 of 16
    CAN_FIFO_THRESHOLD_12   = 5,        // 6/8 of 16
    CAN_FIFO_THRESHOLD_14   = 6,        // 7/8 of 16
    CAN_FIFO_THRESHOLD_FULL = 7,        // 8/8 of 16
    CAN_FIFO_THRESHOLD_1    = 8,        // every message
    CAN_FIFO_THRESHOLD_END_ENUM
} can_fifo_threshold_t;
```

2.10.6 R_CAN_AddRxRule() Data Types

```
typedef struct st_can_filter
{
    bool_t      check_ide;
    uint8_t     ide;
    bool_t      check_rtr;
```

```
    uint8_t     rtr;
    uint32_t    id;
    uint32_t    id_mask;
    uint8_t     min_dlc;
    uint16_t    label;          // 12-bit label
} can_filter_t;
```

2.10.7 R_CAN_SendMsg() Data Types

```
typedef struct st_can_txmsg
{
    uint8_t     ide;
    uint8_t     rtr;
    uint32_t    id;
    uint8_t     dlc;
    uint8_t     data[8];
    bool_t      one_shot;       // no retries on error; txmbx only
    bool_t      log_history;    // true if want to log
    uint8_t     label;          // 8-bit label for History FIFO
} can_txmsg_t;
```

2.10.8 R_CAN_GetMsg() Data Types

```
typedef struct st_can_rxmsg
{
    uint8_t     ide;
    uint8_t     rtr;
    uint32_t    id;
    uint8_t     dlc;
    uint8_t     data[8];
    uint16_t    label;          // 12-bit label from receive rule
    uint16_t    timestamp;
} can_rxmsg_t;
```

2.10.9 R_CAN_GetHistoryEntry() Data Types

```
typedef struct st_can_history
{
    can_box_t   box_id;         // box which sent message
    uint8 t     label;          // associated 8-bit label
} can_history_t;
```

2.10.10 R_CAN_GetStatusMask() Data Types

```
typedef enum e_can_stat
{
    CAN_STAT_FIFO_EMPTY,
    CAN_STAT_FIFO_THRESHOLD,
    CAN_STAT_FIFO_OVFL,         // bits reset after reading
    CAN_STAT_RXMBX_FULL,
    CAN_STAT_GLOBAL_ERR,        // DLC error bit is reset after reading
    CAN_STAT_CH_TXMBX_SENT,     // bits reset after reading
    CAN_STAT_CH_TXMBX_ABORTED,  // bits reset after reading
    CAN_STAT_CH_ERROR,          // bits reset after reading
    CAN_STAT_END_ENUM
} can_stat_t;
```

```
/* Returned mask values (multiple bits may be set at the same time)

/* CAN_STAT_CH_TXMBX_SENT, CAN_STAT_CH_TXMBX_ABORTED */
#define CAN_MASK_TXMBX_0            (0x0001)
#define CAN_MASK_TXMBX_1            (0x0002)
#define CAN_MASK_TXMBX_2            (0x0004)
         :
#define CAN_MASK_TXMBX_13           (0x2000)
#define CAN_MASK_TXMBX_14           (0x4000)
#define CAN_MASK_TXMBX_15           (0x8000)

/* CAN_STAT_RXMBX_FULL */
#define CAN_MASK_RXMBX_0            (0x0001)
#define CAN_MASK_RXMBX_1            (0x0002)
#define CAN_MASK_RXMBX_2            (0x0004)
         :
#define CAN_MASK_RXMBX_13           (0x2000)
#define CAN_MASK_RXMBX_14           (0x4000)
#define CAN_MASK_RXMBX_15           (0x8000)

/* CAN_STAT_FIFO_EMPTY, CAN_STAT_FIFO_THRESHOLD, CAN_STAT_FIFO_OVFL */
#define CAN_MASK_RXFIFO_0           (0x00000001)
#define CAN_MASK_RXFIFO_1           (0x00000002)
#define CAN_MASK_RXFIFO_2           (0x00000004)
#define CAN_MASK_RXFIFO_3           (0x00000008)
#define CAN_MASK_RXFIFO_4           (0x00000010)
#define CAN_MASK_RXFIFO_5           (0x00000020)
#define CAN_MASK_RXFIFO_6           (0x00000040)
#define CAN_MASK_RXFIFO_7           (0x00000080)
#define CAN_MASK_CH0_TXFIFO_0       (0x00000100)
#define CAN_MASK_CH0_TXFIFO_1       (0x00000200)
#define CAN_MASK_CH0_GWFIFO         (0x00000400)
         :
#define CAN_MASK_CH4_TXFIFO_0       (0x00100000)
#define CAN_MASK_CH4_TXFIFO_1       (0x00200000)
#define CAN_MASK_CH4_GWFIFO         (0x00400000)
#define CAN_MASK_CH0_HIST_FIFO      (0x00800000)
#define CAN_MASK_CH1_HIST_FIFO      (0x01000000)
#define CAN_MASK_CH2_HIST_FIFO      (0x02000000)
#define CAN_MASK_CH3_HIST_FIFO      (0x04000000)
#define CAN_MASK_CH4_HIST_FIFO      (0x08000000)

/* CAN_STAT_GLOBAL_ERR */
#define CAN_MASK_ERR_DLC            (0x0001)
#define CAN_MASK_ERR_GW_RX_OVFL     (0x0002)
#define CAN_MASK_ERR_HIST_OVFL      (0x0004)
#define CAN_MASK_ERR_FIFO_OVFL      (0x0006)

/* CAN_STAT_CH_ERROR */
#define CAN_MASK_ERR_PROTOCOL       (0x0001)
#define CAN_MASK_ERR_WARNING        (0x0002)
#define CAN_MASK_ERR_PASSIVE        (0x0004)
#define CAN_MASK_ERR_BUS_OFF_ENTRY  (0x0008)
#define CAN_MASK_ERR_BUS_OFF_EXIT   (0x0010)
#define CAN_MASK_ERR_OVERLOAD       (0x0020)
#define CAN_MASK_ERR_DOMINANT_LOCK  (0x0040)
#define CAN_MASK_ERR_ARB_LOST       (0x0080)
#define CAN_MASK_ERR_STUFF          (0x0100)
#define CAN_MASK_ERR_FORM           (0x0200)
#define CAN_MASK_ERR_ACK            (0x0400)
```

```
#define CAN_MASK_ERR_CRC            (0x0800)
#define CAN_MASK_ERR_RECESSIVE_BIT  (0x1000)
#define CAN_MASK_ERR_DOMINANT_BIT   (0x2000)
#define CAN_MASK_ERR_ACK_DELIMITER  (0x4000)
```

2.10.11 R_CAN_GetCountErr() Data Types

```
typedef enum e_can_count
{
    CAN_COUNT_RX_ERR,
    CAN_COUNT_TX_ERR,
    CAN_COUNT_END_ENUM
} can_count_t;
```

2.10.12 R_CAN_Control() Data Types

```
typedef enum e_can_cmd
{
    CAN_CMD_ABORT_TX,                       // argument: transmit mailbox id
    CAN_CMD_RESET_TIMESTAMP,
    CAN_CMD_SET_MODE_COMM,                  // start normal bus communications
    CAN_CMD_SET_MODE_TST_STANDARD,
    CAN_CMD_SET_MODE_TST_LISTEN,
    CAN_CMD_SET_MODE_TST_EXT_LOOPBACK,
    CAN_CMD_SET_MODE_TST_INT_LOOPBACK,
    CAN_CMD_SET_MODE_TST_INTERCHANNEL,
    CAN_CMD_END_ENUM
} can_cmd_t;
```

2.11 Return Values

API function return values. This enum is found in r_rscan_rz_if.h along with the API function declarations.

```
typedef enum e_can_err          // CAN API error codes
{
    CAN_SUCCESS=0,
    CAN_ERR_OPENED,             // Call to Open already made
    CAN_ERR_NOT_OPENED,         // Call to Open not yet made
    CAN_ERR_INIT_DONE,          // Call to InitChan already made for channel
    CAN_ERR_CH_NO_INIT,         // Channel not initialized
    CAN_ERR_INVALID_ARG,        // Invalid argument passed to function
    CAN_ERR_MISSING_CALLBACK,   // Callback func not provided and ints requested
    CAN_ERR_MAX_ONE_GWFIFO,     // Can only configure one GWFIFO
    CAN_ERR_MAX_RULES,          // Max configured rules already present
                                // (as specified in r_rscan_rz_config.h)
    CAN_ERR_BOX_FULL,           // Transmit mailbox or FIFO is full
    CAN_ERR_BOX_EMPTY,          // Receive mailbox or FIFO is full
    CAN_ERR_ILLEGAL_MODE        // Not in proper mode for request
} can_err_t;
```

3. API Functions

3.1 Summary

The following functions are included in this design:

Function	Description
R_CAN_Open()	Initializes the driver's internal structures and all of the receive mailboxes.
R_CAN_InitChan()	Sets the bit rate clock for the channel and initializes all of the transmit mailboxes.
R_CAN_ConfigFIFO()	Initializes a FIFO for usage. This function should not be called if FIFOs are not used.
R_CAN_AddRxRule()	Adds a receive rule to a channel. Specifies receive message filter and destination routing.
R_CAN_SendMsg()	Loads a message into a transmit mailbox or FIFO for transmission.
R_CAN_GetMsg()	Fetches a message from a receive mailbox or FIFO.
R_CAN_GetHistoryEntry()	Fetches a log entry from a transmit history FIFO.
R_CAN_GetStatusMask()	Returns a 32-bit mask based upon the status requested. Bit #defines have the form CAN_MASK_xxx.
R_CAN_GetCountFIFO()	Returns the number of messages in a FIFO.
R_CAN_GetCountErr()	Returns the number of transmit or receive errors.
R_CAN_Control()	Handles special operations and mode changes.
R_CAN_Close()	Removes power to the CAN peripheral and disables the associated interrupts.
R_CAN_GetVersion()	Returns the driver version number.

3.2 R_CAN_Open()

This function initializes the driver's internal structures and all of the receive mailboxes.

Format

```
can_err_t  R_CAN_Open(can_cfg_t  *p_cfg,
                      void      (* const p_callback)(can_cb_evt_t  event,
                                                     void          *p_args));
```

Parameters

p_cfg

Pointer to configuration structure. The element type definitions are provided in Section 2.10.1.

```
typedef struct st_can_cfg
{
    can_timestamp_src_t     ts_source;
    can_timestamp_div_t     ts_divisor;
} can_cfg_t;
```

p_callback

Optional pointer to main callback function. Must be present if interrupts are enabled in r_rscan_rz_config.h for RX FIFOs or global errors

event

First parameter for callback function. Specifies the interrupt source (see Section 2.10.3)

p_args

Second parameter for callback function (unused).

Return Values

CAN_SUCCESS:	*Successful*
CAN_ERR_OPENED:	*Call to Open already made*
CAN_ERR_INVALID_ARG:	*An element of the p_cfg structure contains an invalid value.*
CAN_ERR_MISSING_CALLBACK:	*A callback function was not provided and a main callback interrupt is enabled in config.h*

Properties

Prototyped in file "r_rscan_rz_if.h"

Description

This function initializes the driver's internal structures, applies clock to the peripheral, and sets the Global and Channel Modes to Reset. The timestamp is configured as per the p_cfg argument, and all receive mailboxes are initialized.

If interrupts are enabled in r_rscan_rz_config.h for receive FIFO thresholds, or DLC or FIFO overflow errors, a callback function must be provided here. Otherwise, NULL is entered.

Reentrant

No.

Example: Polling Configuration

```
    /* All main callback interrupt sources are set to 0 in r_rscan_rz_config.h
*/

    can_cfg_t    config;
    can_err_t    err;

    /* Configure timestamp and Open driver */
    config.timestamp_src = CAN_TIMESTAMP_SRC_CH1_BIT_CLK;
    config.timestamp_div = CAN_TIMESTAMP_DIV_1024;
```

```
    err = R_CAN_Open(&config, NULL);
```

Example: Interrupt Configuration

```
    /* 1+ main callback interrupt sources are set to 1 in r_rscan_rz_config.h */

    can_cfg_t    config;
    can_err_t    err;

    /* Configure timestamp and Open driver */
    config.timestamp_src = CAN_TIMESTAMP_SRC_CH1_BIT_CLK;
    config.timestamp_div = CAN_TIMESTAMP_DIV_1024;
    err = R_CAN_Open(&config, MyCallback);
```

```
/* Sample callback function */
void MyCallback(can_cb_evt_t event, void *p_args)
{
uint32_t   mask;
can_err_t  err;

    if (event == CAN_EVT_RXFIFO_THRESHOLD)
    {
        mask = R_CAN_GetStatusMask(CAN_STAT_FIFO_THRESHOLD, NULL, &err);

        /* check RXFIFOs in use */
        if (mask & CAN_MASK_RXFIFO_1)
        {
             /* read messages */
        }
    }
    else if (event == CAN_EVT_GLOBAL_ERR)
    {
        mask = R_CAN_GetStatusMask(CAN_STAT_GLOBAL_ERR, NULL, &err);

        if (mask & CAN_MASK_ERR_DLC)
        {
            /* handle DLC error */
        }

        if (mask & CAN_MASK_ERR_FIFO_OVFL)
        {
            mask = R_CAN_GetStatusMask(CAN_STAT_FIFO_OVFL, NULL, &err);

            /* check the RXFIFOs, GWFIFO, and HIST_FIFOs in use */
            if (mask & CAN_MASK_CH1_HIST_FIFO)
            {
                /* handle error */
            }
        }
    }
}
```

Special Notes:

None.

3.3 R_CAN_InitChan()

This function sets the bit rate clock for the channel and initializes all of the transmit mailboxes.

Format

```
can_err_t  R_CAN_InitChan(uint8_t        chan,
                          can_bitrate_t  *p_baud,
                          void   (* const p_chcallback)(uint8_t       chan,
                                                        can_cb_evt_t  event,
                                                        void          *p_args));
```

Parameters

chan

Channel to initialize (0-4).

p_baud

Pointer to bit rate structure. See Table 21.6 in the Hardware User's Manual for limitations on bit rate based upon the clock frequency and number of channels used. See Section 21.10.1.2 for bit time settings.

```
typedef struct st_can_bitrate
{
    uint16_t    prescaler;
    uint8_t     tseg1;
    uint8_t     tseg2;
    uint8_t     sjw;
} can_bitrate_t;
```

p_chcallback

Optional pointer to channel callback function. Must be present if interrupts are enabled in r_rscan_rz_config.h for TX mailboxes, TX FIFOs, History FIFOs, or bus errors.

channel

First parameter for channel callback function. Specifies the channel interrupt occurred on.

event

Second parameter for channel callback function. Specifies the interrupt source (see Section 2.10.3)

p_args

Third parameter for callback function (unused).

Return Values

CAN_SUCCESS:	*Successful*
CAN_ERR_ILLEGAL_MODE:	*Not in global reset mode (results from call to Open())*
CAN_ERR_INVALID_ARG:	*An invalid argument was provided*
CAN_ERR_MISSING_CALLBACK:	*A callback function was not provided and a channel interrupt is enabled in config.h*

Properties

Prototyped in file "r_rscan_rz_if.h"

Description

This function initializes all of the channel's transmit mailboxes, sets the bit rate, and enables interrupt sources for the channel as specified in the r_rscan_rz_config.h file. Default values for *p_baud* are provided in r_rscan_rz_if.h. See sections 21.10.2.1 - 21.10.2.2 in the RZ/A1 Hardware User's Manual for calculating Tq bit rate values.

If interrupts are enabled in r_rscan_rz_config.h for TX mailboxes, TX FIFOs, History FIFOs, or bus errors, a callback function must be provided here. Otherwise, NULL is entered.

Reentrant

Yes, for different channels.

Example: Polling Configuration

```
    /* All channel interrupt sources are set to 0 in r_rscan_rz_config.h */

    can_bitrate_t    baud;
    can_err_t        err;

    /* Initialize channel 1 */
    baud.prescaler = CAN_RSK_13MHZXTAL_125KBPS_PRESCALER;
    baud.tseg1 = CAN_RSK_13MHZXTAL_125KBPS_TSEG1;
    baud.tseg2 = CAN_RSK_13MHZXTAL_125KBPS_TSEG2;
    baud.sjw = CAN_RSK_13MHZXTAL_125KBPS_SJW;

    err = R_CAN_InitChan(CAN_CH1, &baud, NULL);
```

Example: Interrupt Configuration

```
    /* 1+ channel interrupt sources are set to 1 in r_rscan_rz_config.h */

    can_bitrate_t    baud;
    can_err_t        err;

    /* Initialize channel 1 */
    baud.prescaler = CAN_RSK_13MHZXTAL_125KBPS_PRESCALER;
    baud.tseg1 = CAN_RSK_13MHZXTAL_125KPS_TSEG1;
    baud.tseg2 = CAN_RSK_13MHZXTAL_125KPS_TSEG2;
    baud.sjw = CAN_RSK_13MHZXTAL_125KPS_SJW;

    err = R_CAN_InitChan(CAN_CH1, &baud, MyChanCallback);
```

```
/* Sample callback function template */
void MyChanCallback(uint8_t       chan,
                    can_cb_evt_t  event,
                    void          *p_args)
{
uint32_t   mask;
can_err_t  err;

    if (event == CAN_EVT_TRANSMIT)
    {
        mask = R_CAN_GetStatusMask(CAN_STAT_CH_TXMBX_SENT, chan, &err);

        /* check transmit mailboxes in use */
        if (mask & CAN_MASK_TXMBX_3)
        {
            /* do stuff */
        }

        mask = R_CAN_GetStatusMask(CAN_STAT_CH_TXMBX_ABORTED, chan, &err);

        /* check transmit mailboxes in use */
        if (mask & CAN_MASK_TXMBX_0)
        {
            /* do stuff */
        }

        mask = R_CAN_GetStatusMask(CAN_STAT_FIFO_THRESHOLD, NULL, &err);

        /* check transmit, gateway, and history FIFOs in use */
        if (mask & CAN_MASK_CH2_TXFIFO_1)
```

```
        {
            /* load next batch of messages for transmit */
        }
    }

    else if (event == CAN_EVT_GWFIFO_RX_THRESHOLD)
    {
        /* read gateway FIFO message if desired */
    }

    else if (event == CAN_EVT_CHANNEL_ERR)
    {
        mask = R_CAN_GetStatusMask(CAN_STAT_CH_ERROR, chan, &err);

        /* check individual errors if desired */
        if (mask & CAN_MASK_ERR_BUS_OFF_ENTRY)
        {
            /* handle error */
        }

        if (mask & CAN_MASK_ERR_BUS_OFF_EXIT)
        {
            /* handle recovery */
        }
    }
}
```

Special Notes:
None.

3.4 R_CAN_ConfigFIFO()

This function initializes a FIFO for usage. This function should not be called if FIFOs are not used.

Format

```
can_err_t  R_CAN_ConfigFIFO(can_box_t             fifo,
                            can_fifo_threshold_t  threshold,
                            can_box_t             txmbx);
```

Parameters

fifo_id

Box id for FIFO (see Section 2.10.1)

threshold

Number of messages needed in FIFO to set interrupt flag (see Section 2.10.5). Note that the only valid thresholds for the History FIFOs is 1 or 12 messages. All others may use 1, 2, 4, 6, 8, 10, 12, 14, or full (16).

txmbx

Box id for associated transmit mailbox (for transmit and gateway FIFOs only). This argument is ignored for receive and history FIFOs.

Return Values

CAN_SUCCESS:	*Successful*
CAN_ERR_ILLEGAL_MODE:	*Not in global reset mode (results from call to Open())*
CAN_ERR_CH_NO_INIT:	*Channel not initialized yet*
CAN_ERR_INVALID_ARG:	*An invalid argument was provided*
CAN_ERR_MAX_ONE_GWFIFO:	*Can only configure one gateway FIFO*

Properties

Prototyped in file “r_rscan_rz_if.h”

Description

FIFO usage is optional.

This function is used to activate a FIFO. All FIFOs are 16 entries deep. The transmit and gateway FIFOs must have associated with it a standard transmit mailbox. The number of the mailbox determines the priority of the FIFO when transmitting (mailbox 0 = highest priority; mailbox 15 = lowest).

Reentrant

Yes, for different FIFOs.

Example: RX FIFO

```
    can_err_t        err;

    /*
     * Set interrupt flag on every message received on RX FIFO 0.
     * Interrupt occurs if CAN_CFG_INT_RXFIFO_THRESHOLD is set to 1 in config.h.
     * Interrupt calls main callback function with CAN_EVT_RXFIFO_THRESHOLD.
     */
    err = R_CAN_ConfigFIFO(CAN_BOX_RXFIFO_0,
                           CAN_FIFO_THRESHOLD_1,
                           0);                      // unused field here
```

Example: TX FIFO

```
    can_err_t        err;

    /*
     * Associate mailbox 3 with TX FIFO 0 on channel 1.
```

```
    * Set interrupt flag when 4 messages remain in FIFO.
    * Interrupt occurs if CAN_CFG_INT_TXFIFO_THRESHOLD is set to 1 in config.h.
    * Interrupt calls channel callback function with CAN_EVT_TRANSMIT.
    */
   err = R_CAN_ConfigFIFO(CAN_BOX_CH1_TXFIFO_0,
                          CAN_FIFO_THRESHOLD_4,
                          CAN_BOX_CH1_TXMBX_3);
```

Example: History FIFO

```
 can_err_t        err;

/*
 * Set threshold to 12 for History FIFO on channel 2.
 * Interrupt occurs if CAN_CFG_INT_HIST_FIFO_THRESHOLD is set to 1 in config.h.
 * Interrupt calls channel callback function with CAN_EVT_TRANSMIT.
 */
 err = R_CAN_ConfigFIFO(CAN_BOX_CH2_HIST_FIFO,
                        CAN_FIFO_THRESHOLD_12,
                        0);                      // unused field here
```

Special Notes:

None.

3.5 R_CAN_AddRxRule()

This function adds a receive rule to a channel. Specifies receive message filter and destination routing.

Format

```
can_err_t  R_CAN_AddRxRule(uint8_t        chan,
                           can_filter_t   *p_filter,
                           can_box_t      dst_box);
```

Parameters

chan

Channel to apply rule to

p_filter

Pointer to rule information.

```
typedef struct st_can_filter
{
    bool_t      check_ide;
    uint8_t     ide;
    bool_t      check_rtr;
    uint8_t     rtr;
    uint32_t    id;
    uint32_t    id_mask;
    uint8_t     min_dlc;
    uint16_t    label;          // 12-bit label
} can_filter_t;
```

dst_box

Destination box (receive mailbox or receive FIFO) to route message to (see Section 2.10.1).

Return Values

CAN_SUCCESS:	*Successful*
CAN_ERR_ILLEGAL_MODE:	*Not in global reset mode (results from call to Open())*
CAN_ERR_CH_NO_INIT:	*Channel not initialized yet*
CAN_ERR_INVALID_ARG:	*An invalid argument was provided*
CAN_ERR_MAX_RULES:	*Max rules already present (as defined in r_rscan_rz_config.h, 128/channel, or 320 total)*

Properties

Prototyped in file “r_rscan_rz_if.h”

Description

This function is used to add a receive rule to a channel. There are two parts to this. The first part is specifying a filter as to which fields to inspect on received messages. The second part is to specify a destination to route the message to if it passes the filter test.

A “1” in the *id_mask* field indicates that the corresponding bit in a received message ID will be checked against the bit in the *id* field in this filter (see Examples).

The *label* field in the rule is optional. It is associated with each message that passes the filter. This may serve as a quick identification of a message when it is fetched from a receive box (mailbox or FIFO) using R_CAN_GetMsg()..

Reentrant

No.

Example 1: Match a range of messages

```
    can_filter_t  filter;
    can_err_t     err;

    /* Setup filter */
    filter.check_ide = TRUE;  // check the IDE field in message
```

```
    filter.ide = 0;           // 11-bit ID
    filter.check_rtr = FALSE; // do not check the RTR field in message
    filter.rtr = 0;           // (value does not matter here; not checking)
    filter.id = 0x040;        // message ID
    filter.id_mask = 0x7F0;   // messages with IDs of 0x040-0x04F are accepted
    filter.min_dlc = 4;       // message data must be at least four bytes long
    filter.label = 0x800;     // arbitrary label applied to msgs of this type


    /* Add rule to channel 1. Route filtered messages to receive mailbox 5. */
    err = R_CAN_AddRxRule(CAN_CH1, &filter, CAN_BOX_RXMBX_5);
```

Example 2: Exact match for message

```
    can_filter_t  filter;
    can_err_t     err;

    /* Setup filter */
    filter.check_ide = TRUE;  // check the IDE field in message
    filter.ide = 0;           // 11-bit ID
    filter.check_rtr = FALSE; // do not check the RTR field in message
    filter.rtr = 0;           // (value does not matter here; not checking)
    filter.id = 0x040;        // message ID
    filter.id_mask = 0x7FF;   // ID must match 0x040 exactly
    filter.min_dlc = 6;       // message data must be at least six bytes long
    filter.label = 0x700;     // arbitrary label applied to msgs of this type


    /* Add rule to channel 2. Route filtered messages to receive mailbox 4. */
    err = R_CAN_AddRxRule(CAN_CH2, &filter, CAN_BOX_RXMBX_4);
```

Special Notes:

Rules cannot be entered after entering communications mode.

3.6 R_CAN_Control()

This function handles special operations and mode changes.

Format

```
can_err_t  R_CAN_Control(can_cmd_t  cmd,
                         uint32_t   arg1);
```

Parameters

cmd

Specifies which command to run.

```
typedef enum e_can_cmd
{
    CAN_CMD_ABORT_TX,                   // argument: transmit mailbox id
    CAN_CMD_RESET_TIMESTAMP,
    CAN_CMD_SET_MODE_COMM,              // start normal bus communications
    CAN_CMD_SET_MODE_TST_STANDARD,
    CAN_CMD_SET_MODE_TST_LISTEN,
    CAN_CMD_SET_MODE_TST_EXT_LOOPBACK,
    CAN_CMD_SET_MODE_TST_INT_LOOPBACK,
    CAN_CMD_SET_MODE_TST_INTERCHANNEL,
    CAN_CMD_END_ENUM
} can_cmd_t;
```

arg1

Argument which is specific to command. Most commands do not require an argument.
For the command CAN_CMD_ABORT_TX, the argument is a transmit mailbox id (see Section 2.10.1).

Return Values

CAN_SUCCESS:	*Successful*
CAN_ERR_INVALID_ARG:	*An invalid argument was provided*
CAN_ERR_ILLEGAL_MODE:	*Changing to requested mode is illegal from current mode.*

Properties

Prototyped in file “r_rscan_rz_if.h”

Description

This function is used for resetting the timestamp counter, aborting transmission of mailbox messages, and changing the CAN mode.

The following sequence of function calls is used to setup the CAN:

```
R_CAN_Open();
R_CAN_InitChan();   // do for 1-5 channels
R_CAN_ConfigFIFO(); // do for 0 or more FIFOs
R_CAN_AddRxRule();  // do for 1-320 rules
```

Once the CAN is setup, the peripheral should enter normal communications mode or a test mode.

```
R_CAN_Control();    // Use CAN_CMD_SET_MODE_COMM or CAN_CMD_SET_MODE_TST_xxx
```

Note: If a Bus Off condition is detected on a channel, the channel enters Halt Mode and all communications cease. They cannot resume until after a Bus Off Recovery condition is detected and the application calls R_CAN_Control(CAN_CMD_SET_MODE_COMM).

Reentrant

Yes.

Example: Enter Normal Communications Mode

```
can_err_t  err;
```

```
    err = R_CAN_Control(CAN_CMD_SET_MODE_COMM, 0);
```

Example: Enter Inter-channel Communications Test Mode

```
    can_err_t  err;

    err = R_CAN_Control(CAN_CMD_SET_MODE_TST_INTERCHANNEL, 0);
```

Example: Abort Transmit

```
    can_err_t  err;

    /* Abort transmit on mailbox 6 on channel 1*/
    err = R_CAN_Control(CAN_CMD_ABORT_TX, CAN_BOX_CH1_TXMBX_6);
```

Special Notes:

Summary of different test modes:

- Standard Test Mode: Allows for CRC testing
- Listen-only Mode: Used for detecting communication speed. Cannot call R_CAN_SendMsg() in this mode.
- Internal Loopback Mode: Messages sent on a channel are handled as received messages and processed on that same channel. Here, the CAN transceiver is bypassed.
- Inter-channel Communications Mode: Same as Internal Loopback mode, only messages can be received from other local channels.
- External Loopback Mode: Same as Internal Loopback mode, only the transceiver is used.

3.7 R_CAN_SendMsg()

This function loads a message into a transmit mailbox or FIFO for transmission.

Format

```
can_err_t  R_CAN_SendMsg(can_box_t    box_id,
                         can_txmsg_t  *p_txmsg);
```

Parameters

box_id

Transmit box id (mailbox or FIFO; see Section 2.10.1)

p_msg

Pointer to message to send

```
typedef struct st_can_txmsg
{
    uint8_t     ide;
    uint8_t     rtr;
    uint32_t    id;
    uint8_t     dlc;
    uint8_t     data[8];
    bool_t      one_shot;       // no retries on error; txmbx only
    bool_t      log_history;    // true if want to log
    uint8_t     label;          // 8-bit label for History FIFO
} can_txmsg_t;
```

Return Values

CAN_SUCCESS:	*Successful*
CAN_ERR_INVALID_ARG:	*An invalid argument was provided*
CAN_ERR_BOX_FULL:	*Transmit mailbox or FIFO is full*
CAN_ERR_ILLEGAL_MODE:	*Cannot send message in current mode.*

Properties

Prototyped in file “r_rscan_rz_if.h”

Description

This function places a message into a 1-message deep transmit mailbox or 16-message deep transmit FIFO. If there is already a message waiting to send in the mailbox, or 16 messages already exist in the FIFO, CAN_ERR_BOX_FULL is returned immediately. If the box_id is for a transmit mailbox and interrupts are not enabled (CAN_CFG_INT_MBX_TX_COMPLETE is 0), this function blocks until the message is sent. If interrupts are enabled or the message is for a transmit FIFO, the function will return immediately after loading the message into the transmit registers.

Reentrant

Yes, for different boxes.

Example:

```
    can_txmsg_t   txmsg;
    can_err_t     err;

    /* Setup message */
    txmsg.ide = 0;                // ID field is 11-bits
    txmsg.rtr = 0;                // local message
    txmsg.id = 0x022;             // destination ID
    txmsg.dlc = 5;                // data length
    txmsg.data[0] = 'h';          // data…
    txmsg.data[1] = 'e';
    txmsg.data[2] = 'l';
    txmsg.data[3] = 'l';
```

```
    txmsg.data[4] = 'o';
    txmsg.one_shot = false;       // do normal retries on error
    txmsg.log_history = false;    // do not log in History FIFO
    txmsg.label = 0;              // (label ignored because not logging message)

    /*
     * Place message in transmit mailbox 2 on channel 1.
     * If transmit complete interrupt is not enabled, the function returns
     * after the message has been sent (assuming no error occurred).
     */
    err = R_CAN_SendMsg(CAN_BOX_CH1_TXMBX_2, &txmsg);
```

Special Notes:

None.

3.8 R_CAN_GetMsg()

This function fetches a message from a receive mailbox or FIFO.

Format

```
can_err_t  R_CAN_GetMsg(can_box_t    box_id,
                        can_rxmsg_t  *p_rxmsg);
```

Parameters

box_id

Receive box id (mailbox or FIFO; see Section 2.10.1)

p_rxmsg

Pointer to message buffer to load

```
typedef struct st_can_rxmsg
{
    uint8_t     ide;
    uint8_t     rtr;
    uint32_t    id;
    uint8_t     dlc;
    uint8_t     data[8];
    uint16_t    label;          // 12-bit label from receive rule
    uint16_t    timestamp;
} can_rxmsg_t;
```

Return Values

CAN_SUCCESS:	*Successful*
CAN_ERR_CH_NO_INIT:	*Channel not initialized yet*
CAN_ERR_INVALID_ARG:	*An invalid argument was provided*
CAN_ERR_BOX_EMPTY:	*No message available to fetch*

Properties

Prototyped in file “r_rscan_rz_if.h”

Description

This function loads the message from a receive mailbox or FIFO into the message buffer provided. If there are no messages in the box, this function does not block and returns a CAN_ERR_BOX_EMPTY.

Reentrant

Yes, for different boxes.

Example:

```
    can_rxmsg_t   rxmsg;
    can_err_t     err;

    /* Wait for message to appear in receive mailbox 3 */
    while (R_CAN_GetMsg(CAN_BOX_RXMBX_3, &rxmsg) == CAN_ERR_BOX_EMPTY)
        ;

    /* rxmsg contains message */
```

Special Notes:

None.

3.9 R_CAN_GetHistoryEntry()

This function fetches a log entry from a transmit history FIFO.

Format

```
can_err_t  R_CAN_GetHistoryEntry(can_box_t      box_id,
                                 can_history_t  *p_entry);
```

Parameters

box_id

Transmit history FIFO (see Section 2.10.1)

p_rxmsg

Pointer to entry buffer to load

```
typedef struct st_can_history
{
    can_box_t   box_id;   // box which sent message
    uint8_t     label;    // associated 8-bit label
} can_history_t;
```

Return Values

CAN_SUCCESS:	*Successful*
CAN_ERR_INVALID_ARG:	*An invalid argument was provided*
CAN_ERR_BOX_EMPTY:	*No entry available to fetch*

Properties

Prototyped in file “r_rscan_rz_if.h”

Description

An entry is added to the history FIFO each time an R_CAN_SendMsg() is called with the “log_history” in the argument structure is set to TRUE. This function loads a log entry from a transmit history FIFO into the entry buffer provided. If there are no entries in the FIFO, this function does not block and returns a CAN_ERR_BOX_EMPTY. The use of this feature is not required for normal operations.

Reentrant

Yes, for different boxes.

Example:

```
can_history_t   entry;
can_err_t       err;

/* Process all entries in transmit history FIFO for channel 1 */
while (R_CAN_GetMsg(CAN_BOX_CH1_TXHIST_FIFO, &entry) == CAN_SUCCESS)
{
    /* process entries here */
}
```

Special Notes:

None.

3.10 R_CAN_GetStatusMask()

This function returns a 32-bit mask based upon the status requested. Bit #defines have the form CAN_MASK_xxx.

Format

```
uint32_t  R_CAN_GetStatusMask(can_stat_t  type,
                              uint8_t     chan,
                              can_err_t   *p_err);
```

Parameters

type

Specifies which status to return.

```
typedef enum e_can_stat
{
    CAN_STAT_FIFO_EMPTY,
    CAN_STAT_FIFO_THRESHOLD,
    CAN_STAT_FIFO_OVFL,          // bits reset after reading
    CAN_STAT_RXMBX_FULL,
    CAN_STAT_GLOBAL_ERR,         // DLC error bit is reset after reading
    CAN_STAT_CH_TXMBX_SENT,      // bits reset after reading
    CAN_STAT_CH_TXMBX_ABORTED,   // bits reset after reading
    CAN_STAT_CH_ERROR,           // bits reset after reading
    CAN_STAT_END_ENUM
} can_stat_t;
```

chan

Specifies which channel to return status for. Applies only to `CAN_STAT_CH_xxx` requests.

p_err

Pointer to returned error code.

CAN_SUCCESS:	*Successful*
CAN_ERR_INVALID_ARG:	*An invalid argument was provided*

Return Values

32-bit box or error mask whose bit definitions have the form CAN_MASK_xxx and are defined in Section 2.10.10.

Properties

Prototyped in file “r_rscan_rz_if.h”

Description

This function returns a mask based upon the status type requested. All bit masks have the form CAN_MASK_xxx (see Section 2.10.10).

Reentrant

Yes.

Example

```
can_err_t   err;
can_rxmsg_t rxmsg;


/* Wait for a message to come in on any receive mailbox */
while (R_CAN_GetStatusMask(CAN_STAT_RXMBX_FULL, 0, &err) == 0)
    ;

/* Check if receive mailbox 15 is full */
if (R_CAN_GetStatusMask(CAN_STAT_RXMBX_FULL, 0, &err) & CAN_MASK_RXMBX_15)
{
    /* get message */
    R_CAN_GetMsg(CAN_BOX_RXMBX_15, &rxmsg);
}
```

Special Notes:
None.

3.11 R_CAN_GetCountFIFO()

This function returns the number of items in a FIFO.

Format

```
uint32_t  R_CAN_GetCountFIFO(can_box_t  box_id,
                             can_err_t  *p_err);
```

Parameters

box_id

Specifies which FIFO to check (see Section 2.10.1).

p_err

Pointer to returned error code.

CAN_SUCCESS:	*Successful*
CAN_ERR_INVALID_ARG:	*An invalid argument was provided*

Return Values

Number of items in the FIFO (0-16).

Properties

Prototyped in file “r_rscan_rz_if.h”

Description

This function returns the number of items in the FIFO specified by *box_id*. This function is not required for normal operations.

Reentrant

Yes.

Example

```
    uint32_t   cnt;
    can_err_t  err;

    /* Determine the number of messages in the History FIFO for channel 1 */
    cnt = R_CAN_GetCountFIFO(CAN_BOX_CH1_HIST_FIFO, &err);
```

Special Notes:

All FIFO usage is optional.

3.12 R_CAN_GetCountErr()

Returns the number of transmit or receive errors.

Format

```
uint32_t  R_CAN_GetCountErr(can_count_t  type,
                            uint8_t      chan,
                            can_err_t    *p_err);
```

Parameters

type

Specifies which status to return.

```
typedef enum e_can_count
{
    CAN_COUNT_RX_ERR,
    CAN_COUNT_TX_ERR,
    CAN_STAT_END_ENUM
} can_count_t;
```

chan

Specifies which channel to return error count for.

p_err

Pointer to returned error code.

CAN_SUCCESS:	*Successful*
CAN_ERR_INVALID_ARG:	*An invalid argument was provided*

Return Values

The number of errors detected.

Properties

Prototyped in file “r_rscan_rz_if.h”

Description

This function returns the number of receive or transmit errors on a channel based upon the count type requested.

Reentrant

Yes.

Example

```
uint32_t   rxcnt,txcnt;
can_err_t  err;

/* Get the number of errors detected on channel 2 */
rxcnt = R_CAN_GetCountErr(CAN_COUNT_RX_ERR, CAN_CH2, &err);
txcnt = R_CAN_GetCountErr(CAN_COUNT_TX_ERR, CAN_CH2, &err);
```

Special Notes:

This use of this function is optional. It can be used to detect the health of the network and how close the network is to entering the Error Passive state (128 errors) or Bus Off state (255 errors).

:

3.13 R_CAN_Close()

This function removes clock from the CAN peripheral and disables the associated interrupts.

Format

```
void  R_CAN_Close(void);
```

Parameters

None

Return Values

None

Properties

Prototyped in file “r_rscan_rz_if.h”

Description

This function halts all existing communications, disables all interrupts (if any), and shuts down the peripheral.

Reentrant

Yes, but no need to ever call more than once.

Example

```
    R_CAN_Close();
```

Special Notes:

None.

3.14 R_CAN_GetVersion()

This function returns the driver version number at runtime.

Format

```
uint32_t  R_CAN_GetVersion(void);
```

Parameters

None

Return Values

Version number.

Properties

Prototyped in file “r_rscan_rz_if.h”

Description

Returns the version of this module. The version number is encoded such that the top two bytes are the major version number and the bottom two bytes are the minor version number.

4. Demo Project

The CAN Driver demo program is written for channel 1 on the RSK+RZA1H board.

This program requires the connection of a CAN device (such as a sniffer) on channel 1 capable of receiving and sending messages. The program spins in a loop sending a hard-coded message then receiving one message at a time. The messages received must have an ID of 0x60-0x6F and contain at least 4 bytes of data.

The baud rate is set to 125Kbps.

This program can run using either mailboxes without interrupts or FIFOs with interrupts. The desired operation is configured by changing the value of USE_FIFOS in main.c to 0 for mailboxes or 1 for FIFOs.

The RSK board requires 0-ohm resistors in the following locations for proper CAN operation on channel 1: R104 (not R105) and R206 (not R207).

5. Website and Support

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

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

All trademarks and registered trademarks are the property of their respective owners.

Revision Record

		Description	
Rev.	**Date**	**Page**	**Summary**
1.00	Apr 23, 2015	—	Initial release
1.02	Apr 4, 2016	—	Fixed bug in channel-to-index conversion.

General Precautions in the Handling of MPU/MCU Products

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

1. 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 LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible. Unused pins should be handled as described under Handling of Unused Pins in the manual.

2. Processing at Power-on

   The state of the product is undefined at the moment when power is supplied.

   — The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the moment when power is supplied.
   In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the moment when power is supplied until the reset process is completed.
   In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the moment when power is supplied until the power reaches the level at which resetting has been specified.

3. Prohibition of Access to Reserved Addresses

   Access to reserved addresses is prohibited.

   — The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals

   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.

   — When the clock signal is generated with an external resonator (or from an external oscillator) during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Moreover, when switching to a clock signal produced with an external resonator (or by an external oscillator) while program execution is in progress, wait until the target clock signal is stable.

5. Differences between Products

   Before changing from one product to another, i.e. to a product with a different part number, confirm that the change will not lead to problems.

   — The characteristics of an MPU or MCU in the same group but having a different part number may differ in terms of the 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 of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
3. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from such alteration, modification, copy or otherwise misappropriation of Renesas Electronics product.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The recommended 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; and industrial robots etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; and safety equipment etc.
   Renesas Electronics products are neither intended nor 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 damages (nuclear reactor control systems, military equipment etc.). You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application for which it is not intended. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for which the product is not intended by Renesas Electronics.
6. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or 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. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You should not use Renesas Electronics products or technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. When exporting the Renesas Electronics products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, who distributes, disposes of, or otherwise places the product with a third party, to notify such third party in advance of the contents and conditions set forth in this document, Renesas Electronics assumes no responsibility for any losses incurred by you or third parties as a result of unauthorized use of Renesas Electronics products.
11. This document may not be reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

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

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

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

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

Renesas Electronics America Inc.
2801 Scott Boulevard Santa Clara, CA 95050-2549, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

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

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

Renesas Electronics (China) Co., Ltd.
Room 1709, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100191, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, P. R. China 200333
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited
Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.
13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.
Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.
No.777C, 100 Feet Road, HALII Stage, Indiranagar, Bangalore, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
12F., 234 Teheran-ro, Gangnam-Gu, Seoul, 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

© 2015 Renesas Electronics Corporation. All rights reserved.

Colophon 5.0