RX ファミリ

CAN FD モジュール Firmware Integration Technology

要旨

Renesas CAN FD (Controller Area Network with Flexible Data Rate) API (Application Programming Interface) を使って、CAN バス上のデータを送信、受信、監視できます。本ドキュメントでは、本 API の使用方法と CAN FD モジュールのいくつかの機能について説明します。

対象デバイス

本 API で現在サポートしているデバイスは以下の通りです。

- RX660 グループ

本アプリケーションノートを他のマイコンへ適用する場合、そのマイコンの仕様に合わせて変更し、十分評価してください。

対象コンパイラ

- ルネサスエレクトロニクス製 RX ファミリ用 C/C++コンパイラパッケージ
- GCC for Renesas RX
- IAR C/C++ Compiler for Renesas RX

各コンパイラの動作確認環境に関する詳細な内容は、「7.1 動作確認環境」を確認してください。

目次

1. 概要

CAN FD モジュールは、オプションで Flexible Data (CAN FD) を使ってデータフェーズを高速化した CAN ネットワーク上の通信に使用できます。さまざまなメッセージフィルタとバッファオプションが使用可能です。

1.1 基本情報

機能

- 互換性
  - CAN 2.0 フレームと CAN FD フレームを同じチャネルで送受信
  - データ転送速度：アービトレーションフェーズでは最大 1Mbps。FD を利用したデータフェーズでは最大 8Mbps
  - ISO 11898-1:2015 準拠
- バッファ
  - 32 個のグローバル受信メッセージバッファ (RX MB)
  - 2 個のグローバル受信 FIFO (RX FIFO)
  - 1 チャネルあたり 4 個の送信メッセージバッファ (TX MB)
  - 受信 FIFO または送信 FIFO として設定可能な共通 FIFO が 1 個
- フィルタリング
  - 最大 128 のフィルタルールを両チャネル共通で適用可能
  - 各ルールは以下に基づいて個別にフィルタに設定可能
    - ID
    - 標準 ID または拡張 ID（IDE ビット）
    - データフレームまたはリモートフレーム（RTR ビット）
    - ID、IDE、RTR マスク
    - 最小 DLC（データ長）値
- 割り込み
  - 設定可能なグローバル RX FIFO 割り込み
    - FIFO ごとに設定可能
    - 特定の深さでの割り込み、または受信したメッセージごとの割り込み
  - チャネル TX 割り込み
  - グローバルエラー
    - DLC チェック
    - メッセージロスト
    - FD ペイロードオーバーフロー
  - チャネルエラー
    - バスエラー
    - エラー警告
    - エラーパッシブ
    - バスオフ発生
    - バスオフ復帰
    - オーバーロード
    - バスロック
    - アービトレーションロス
    - 送受信中止

1.1.1 Flexible Data (FD)

Flexible Data は CAN プロトコルを拡張したものであり、他の機能に加え、最大 64 バイトのメッセージおよびデータビットレートの高速化を可能にします。CAN FD ドライバは以下に対応しています。

- FD メッセージの送受信
- データフェーズのビットレート切り替え（最大 8MHz）
- エラー状態 (ESI) ビットの手動設定と自動設定

送信時に上記のオプションを 1 つ以上指定するには、can_frame_t::options と canfd_frame_options_t の値を連結して設定してください。受信メッセージ内のこのフィールドには、可能ならば自動的に値が設定されます。

```
#define CAN_FD_DATA_LENGTH_CODE (64)   //FD フレームのデータ長コード

/* ビットレート切り替え (BRS) を有効にした状態で 64 バイト書き込むためにフレームを設定 */
   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;
```

注記
ビットレート切り替えを使用する場合は、希望のデータビットレートを Smart Configurator で設定してください。

1.1.2 ビットレートの計算

CAN FD モジュールのビットレートは、Smart Configurator で手動設定します。

CAN FD モジュールは、クロックソースとして PLL またはメイン発振器を使用します。正確なビットレートを求めるには、以下の計算式の基準に合わせて CAN FD ソースクロックまたは除数を調整する必要がある場合があります。

```
ビットレート = canfd_clock_hz / ((タイムセグメント 1 + タイムセグメント 2 + 1) * プリスケーラ)
```

CAN FD の場合、各要素に設定できる値は以下の通りです。

要素	最小	最大（公称）	最大（データ）
ビットレート	-	1Mbps	8Mbps
タイムセグメント 1	2Tq	256Tq	32Tq
タイムセグメント 2	2Tq	128Tq	16Tq
同期ジャンプ幅	1Tq	タイムセグメント 2	タイムセグメント 2
プリスケーラ	1	1024	256

Smart Configurator の［コンポーネント］タブを使って、CAN FD クロックソースまたは除数の設定と、PLL またはメイン発振器の周波数の設定をします。

同期ジャンプ幅オプションは、バス上の発振器間の差異に対応するために、サンプルポイントを遅延させることができる最大タイムクォンタム (Tq) 数を指定します。

この値は、最大許容クロック誤差に応じて、1～タイムセグメント 2 の設定値の範囲で設定する必要があります。

CAN FD モジュールを使用する場合は、各周波数が以下の関係である必要があります。

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

ビットレートレジスタを設定するための計算式

PCLK は周辺クロック周波数 (PCLKB) です。

fcan = PCLK または EXTAL

プリスケーラ値によって CAN FD 周辺クロックの周波数を下げます。

fcanclk = fcan/prescaler

1Tq は CAN FD クロックの 1 クロック周期です。

Tq = 1/fcanclk

Tqtot は、CAN FD の 1 ビット時間内の CAN FD 周辺クロック周期の総数であり、周辺クロックによって「時間セグメント」と「SS（常に 1）」の合計で構成されます。コードでの Tqtot は以下のようになります。

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

これらのマクロを設定して、Tqtot が CANFD レジスタで許容されている数値より大きくならないようにします。

注記：CANFD_BRP はユーザプログラム内で定義
BITRATE は予期されるビットレート
ハードウェアマニュアルのビットレート設定例の表を参照してください。

その他の制限を以下に示します。

Tqtot = TSEG1 + TSEG2 + SS　　(TSEG1＞TSEG2 であること)

SS は常に“1”です。多くの場合、同期ジャンプ幅 (SJW) はバス・アドミニストレータが提供します。“1≦SJW≦4”を選択してください。

ビットレートレジスタの設定の計算例

CAN FD BITRATE の設定

詳細については、「RX660 ユーザーズマニュアル (R01UH0937JJ)」のセクション「33.4.1 Initialization of CAN Clock, Bit Timing and Bit Rate（CAN クロック、ビットタイミング、ビットレートの初期化）」を参照してください。

CCLKS は 0（PCLK つまり PCLKB で動作）、言い換えると以下のようになります。

FCANFD = PCLK = PCLKB

CANFD_BRP = ビットレートプリスケーラ

FCANFDCLK = FCANFD / CANFD_BRP

P = BCR 内の BRP[9:0]ビットで選択した値 (P = 0～1023)。P + 1 = CANFD_BRP

TQTOT = 1 CANFD ビット内の Nr CANFD クロックの数 = FCANFDCLK/BITRATE

CCLKS = 0 の場合、r_bsp マクロを使用すると次の結果が得られます。

FCANFD = (BSP_CFG_XTAL_HZ * BSP_CFG_PLL_MUL / BSP_CFG_PCKB_DIV) （式 1）

TQTOT = (FCANFD / (CANFD_BRP * BITRATE)) （式 2）

式 1 を式 2 に代入：

TQTOT = (BSP_CFG_XTAL_HZ * BSP_CFG_PLL_MUL / BSP_CFG_PCKB_DIV)/(CANFD_BRP * BITRATE))、言い換えると

TQTOT = (BSP_CFG_XTAL_HZ * BSP_CFG_PLL_MUL) / (CANFD_BRP * BITRATE * BSP_CFG_PCKB_DIV) （式 3）

例： 希望するビットレートは 500kbps。
CANFD_BRP = 4 を試します。式 3 は以下のようになります。
TQTOT = (24000000 * 10) / (4 * 500000 * 4) = 30。これでは大きすぎます。TQTOT の最大値は 25 です。
CANFD_BRP = 5 を試します。
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:
次の値を試します。
SS = 1、Tq は常に以下の通りです。
TSEG1 = 15 Tq
TSEG2 = 8 Tq
==========
SUM = 24

1.1.3 エラーの処理

CAN FD モジュールには 2 種類のエラー割り込みがあります。チャネル割り込みとグローバル割り込みです。この名前からもわかるように、各チャネルには自身のチャネルエラー割り込みがありますが、グローバルエラー割り込みは 1 つしかありません。

グローバルエラーのコールバックを受信できるのは、設定済みのチャネルだけです。

エラー割り込みコールバックは、can_callback_args_t::event フィールドに CAN_EVENT_ERR_CHANNEL または CAN_EVENT_ERR_GLOBAL を渡します。第 2 フィールド (can_callback_args_t::error) には、実際のエラーコードが canfd_error_t として設定されます。この列挙型にキャストしてエラー状況を取得してください。

1.1.4 DLC チェック

DLC チェックを有効にすると、メッセージは各 AFL ルールの destination.minimum_dlc 値と照合されます。メッセージのデータ長がこの値未満の場合、そのメッセージは拒否されます。

Smart Configurator で DLC チェックを「DLC 置換有効」に設定した場合、DLC の最小設定値を超過したデータは切り捨てられ、当該フレームの DLC 値は一致に設定されます。

1.1.5 FD ペイロードオーバーフロー

宛先バッファより大きい DLC が設定された FD メッセージを受信すると、FD ペイロードオーバーフロー割り込みが発生します（設定していた場合）。ペイロードオーバーフローを「切り捨て」に設定している場合、メッセージは受信されますがバッファ容量までのデータしか保存されません。この場合、DLC 値は変わりません。can_frame_t::data 配列内のこの値を超えたデータは使用されません。

1.2 通信層

下図に CAN FD の通信層を示します。アプリケーション層が最上層、ハードウェア層が最下層となります。

アプリケーション
Renesas CAN FD API
CAN FD モジュール
マイコン／トランシーバ／CAN バス

1.3 FIT CAN FD モジュールを使用する

1.3.1 FIT CAN FD モジュールを C++プロジェクトで使用する

C++プロジェクトでは、FIT CAN FD モジュールのインタフェースヘッダファイルを extern “C”の宣言に追加してください。

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

1.4 物理接続

お使いの CAN FD MCU の CAN FD モジュールのプロトコルコントローラは、CAN FD の送信 MCU 端子 (CTXn)、および受信 MCU 端子 (CRXn) を介して、外部バスのトランシーバに接続する必要があります。

1.5 CAN FD バッファ

バッファ

CAN FD ドライバには、以下の 3 種類のバッファがあります。送信メッセージバッファ (TX MB)、受信メッセージバッファ (RX MB)、FIFO バッファ。FIFO バッファの総数は 3 個です（2 個の受信 FIFO (RX FIFO) + 1 個の共通 FIFO）。

TX メッセージバッファ

TX MB は送信専用です。TX MB を利用できる情報については、お使いのデバイスのハードウェアマニュアルを参照してください。

注記

CAN FD モジュールは、新データがあるかどうか継続的に TX MB をスキャンします。供給クロックによっては、送信開始前に複数の TX MB に書き込みが行われる可能性があります。この場合、

Smart Configurator の「送信の優先順位」オプションで指定した優先順位でメッセージが送信されます。

RX メッセージバッファ

RX MB は受信専用であり、一度に保持できるメッセージは 1 つだけです。

本ソフトウェアでは RX MB には割り込みがありません。割り込みのポーリングには R_CANFD_InfoGet を、読み出しには R_CANFD_Read を使用してください。

RX FIFO

RX FIFO には、メッセージ受信用に割り込み駆動型のキュー機能があります。2 個の RX FIFO が利用可能です。すべての FIFO には以下の特性があります。

- 最大 64 バイトのペイロード
- メッセージ容量は最大 48 件

ひとたび割り込みが発生すると、FIFO が空になり、全てのメッセージがコールバックを介してユーザーコードに渡されるまで割り込みが発生し続けます。しきい値割り込みモードを使用する場合、R_CANFD_InfoGet の呼び出しにより、FIFO 内のデータの有無のチェックが、R_CANFD_Read の呼び出しにより割り込み間の FIFO 読み出しが可能です。

RX バッファプール

ペイロードサイズが 64 バイトに設定されている場合、受信メッセージバッファおよび FIFO バッファに割り当てられた RAM は、16 メッセージ（1216 バイト）に制限されます。受信メッセージバッファと FIFO バッファを設定する場合は、この上限を超えないようにしてください。CAN FD モジュールには、この設定の妥当性をチェックする機能がありません。

制限事項

CAN FD の使用時、開発者は以下の制限に留意しなければなりません。

- RX MB 割り込みは CAN FD ハードウェアを持つ RX MCU で利用可能ですが、本ソフトウェアではこの割り込みは未サポートです。アプリケーションで使用するには、以下の実施を推奨します。R_CANFD_InfoGet を使用して RX MB が受信したデータの有無を判定してから、R_CANFD_Read を使用してデータを受信します。
- CAN FD モジュールには限られた容量のバッファプール RAM があり、RX MB と FIFO ステージの割り当てに使用できます。詳細については、前述の「RX バッファプール」セクションを参照してください。
- R_CANFD_ModeTransition を使ってモードを切り替えるとき、最大で CAN フレーム数個分の遅延が発生することがあります。詳細については、「RX660 ユーザーズマニュアル (R01UH0937JJ)」のセクション「33.3.3.2 Timing of Channel Mode Change（モード変更のタイミング）」を参照してください。

メッセージのフィルタリング（承認フィルタリスト）

メッセージをフィルタリングして必要なメッセージバッファまたは FIFO に入れるために、CAN FD モジュールは承認フィルタリスト (AFL) を使用します。AFL の各エントリには、メッセージのチェックに使用するルールが宛先やその他のフィルタリング情報とともに記載されています。メッセージが受信される

と、CAN FD モジュールは、メッセージとチャネルに設定した各 AFL ルールを内部で照合します。一致が見つかると、そのメッセージはルールで指定した宛先（複数指定可）に転送されます。以下に示す canfd_afl_entry_t の AFL エントリの構造体を参照してください。

```
/** AFL エントリ*/
typedef struct st_canfd_afl_entry_t
{
    uint32_t id              : 29; ///< 照合先 ID
    uint32_t rs              : 1;
    can_frame_type_t frame_type : 1;  ///< フレーム種別（データ／リモート）
    can_id_mode_t    id_mode    : 1;  ///< ID モード（標準／拡張）

    uint32_t mask_id          : 29; ///< ID マスク
    uint32_t rs1             : 1;
    uint32_t mask_frame_type    : 1;  ///< 設定されたフレーム種別のフレームのみ承認
    uint32_t mask_id_mode      : 1;  ///< 設定された ID モードのフレームのみ承認

    canfd_minimum_dlc_t minimum_dlc     : 4; ///< 承認する DLC の最小値（DLC チェックが有効の場合に有効）
    uint32_t rs2                    : 4;
    canfd_rx_mb_t rx_buffer           : 8; ///< このルールで承認されたメッセージを受信する RX メッセージバッファ
    uint32_t rs3                    : 16;
    canfd_rx_fifo_t fifo_select_flags;   ///< このルールで承認されたメッセージを受信する RX FIFO
} canfd_afl_entry_t;
```

設定例については、次の AFL 例を参照してください。

AFL 例

1 件のルールを設定した承認フィルタリスト (AFL) 宣言の例を以下に示します。

```
/* 承認フィルタ配列パラメータ
CANFD_CFG_AFL_CH0_RULE_NUM = 1 */
/* 承認フィルタ配列パラメータ */
#define CANFD_FILTER_ID (0x00001000)
#define MASK_ID              (0x0FFFF000)
#define MASK_ID_MODE    (1)
#define ZERO                    (0U) //配列のインデックス値

const canfd_afl_entry_t p_canfd0_afl[CANFD_CFG_AFL_CH0_RULE_NUM] =
{
    /* 拡張 ID が 0x1000～0x1FFF のメッセージを承認します */
    /* 承認する ID、ID 種別、フレーム種別を指定します。 */
    {
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);
}
```

詳細については、「RX660 ユーザーズマニュアル (R01UH0937JJ)」のセクション「33.5 Filtering Using Acceptance Filter List (AFL)（承認フィルタリスト（AFL）を使ったフィルタリング）」を参照してください。

2. API 情報

CAN FD FIT モジュールの各 API の名称はルネサス API 命名基準に従っています。

2.1 ハードウェア要件

本ドライバでは、ご使用になる MCU が以下の機能をサポートしている必要があります。

- CAN FD モジュール (CAN FD)

2.2 ハードウェアリソース要件

このセクションでは、本ドライバに必要なハードウェア周辺機能の詳細を説明します。明示的に記載していない限り、これらのリソースはドライバ専用に予約されており、アプリケーション内の他の場所では使用できません。

2.2.1 必要な周辺機能

CAN FD モジュール (CAN FD)

2.2.2 その他に使用する周辺機能

本ドライバでは、CAN FD バスの送受信信号用に I/O ポート端子を割り当てる必要があります。割り当てられた端子は、汎用入出力 (GPIO) には使用できません。

本ドライバは、オプションとして、各 CAN FD チャネルに対応するスタンバイ信号とイネーブル信号用に GPIO ポート端子を使います。

2.3 ソフトウェア要件

本ドライバは以下の FIT モジュールに依存しています。

- ルネサスボードサポートパッケージ (r_bsp) v7.20 以上

2.4 制限事項

2.4.1 RAM の配置に関する制限事項

FIT では、NULL と同等な値を API 関数のポインタ引数として設定すると、パラメータチェックによりエラーが返される場合があります。そのため、NULL と同等な値をポインタ引数として API 関数に渡さないでください。

ライブラリ関数の仕様のため、NULL 値は 0 と定義されています。そのため、API 関数のポインタ引数に渡す変数または関数が RAM の先頭アドレス（アドレス 0x0）に配置されていると、上記の現象が発生します。この場合、セクション設定を変更するか、RAM の先頭にダミー変数を用意して、API 関数のポインタ引数に渡した変数や関数がアドレス 0x0 に配置されないようにしてください。

CCRX プロジェクト (e^2 studio V7.5.0) の場合、アドレス 0x0 に変数が配置されることを防ぐために、RAM の先頭アドレスが 0x4 に設定されています。GCC プロジェクト (e^2 studio V7.5.0) および IAR プロジェクト (EWRX V4.12.1) の場合、RAM の先頭アドレスは 0x0 なので、上記の対策が必要です。

セクションのデフォルト設定は IDE のバージョンアップにより変更されることがあります。最新の IDE を使用される際は、セクション設定のご確認をお願いします。

2.5 サポートしているツールチェーン

本ドライバは、「7.1 動作確認環境」に示すツールチェーンで動作確認を行っています。

2.6 割り込みベクタ

CAN TX 割り込みおよび CAN RX 割り込みを使用する場合、それぞれの割り込みの選択型割り込みを設定してください。この設定は「r_bsp_interrupt_config.h」で行えます。

2.7 ヘッダファイル

すべての API 呼び出しとそれをサポートするインタフェース定義は「r_canfd.h」に記載しています。

ビルド時の設定オプションは r_canfd_rx_config.h ファイルで選択または定義されています。

お使いのコードから本 FIT モジュールの CAN FD API を参照するには、以下をインクルードしてください。

#include “r_canfd_rx_if.h”

2.8 整数型

本ソフトウェアは ANSI C99 を使用しています。これらの型は stdint.h で定義されています。

2.9 設定

必要な機能に合わせてアプリケーションをカスタマイズするために、r_canfd_rx_config.h ファイルの変更が必要な場合があります。ルネサス CAN FD API ドライバ関数が含まれている r_canfd_rx.c ファイルの変更は推奨しませんが、API で提供されない機能の追加でメリットがある場合があります。

e² studio の Smart Configurator を使用してこのソフトウェアをインストールする場合、この FIT モジュールの設定は Smart Configurator の［コンポーネント］→［プロパティ］ビューで行います。それ以外の場合、以降の表を目安として使用して手動で r_canfd_rx_config.h を編集できます。

r_canfd_rx_config.h の設定オプション	
CANFD_CFG_PARAM_CHECKING_ENABLE (BSP_CFG_PARAM_CHECKING_ENABLE)	1：パラメータチェックはビルドに含まれます。 0：パラメータチェックはビルドから除外されています。 この#define を BSP_CFG_PARAM_CHECKING_ENABLE に設定すると、システムのデフォルト設定が使用されます。
CANFD_CFG_AFL_CH0_RULE_NUM　32	チャネル 0 専用の承認フィルタリストルールの数 任意の値（0～32） デフォルト値は 32 です。
CANFD_CFG_FD_PROTOCOL_EXCEPTION　0	RES ビットが ISO 11898-1 で定義されたリセッシブとしてサンプリングされた場合にプロトコル例外処理状態に遷移するかどうかを選択します。 (0) = 有効 (ISO 11898-1) （デフォルト） (R_CANFD_GFDCFG_PXEDIS_Msk) = 無効
CANFD_CFG_GLOBAL_ERR_SOURCES　0x3	エラーを契機として割り込みを発生させるかどうかを選択します。 (0X3) （デフォルト） (R_CANFD_GCR_DEIE_Msk | 0x3) (R_CANFD_GCR_MEIE_Msk | 0x3) (R_CANFD_GCR_POIE_Msk | 0x3) (R_CANFD_GCR_DEIE_Msk | R_CANFD_GCR_MEIE_Msk | 0x3)

r_canfd_rx_config.h の設定オプション	
	(R_CANFD_GCR_DEIE_Msk | R_CANFD_GCR_POIE_Msk | 0x3) (R_CANFD_GCR_MEIE_Msk | R_CANFD_GCR_POIE_Msk | 0x3) (R_CANFD_GCR_DEIE_Msk | R_CANFD_GCR_MEIE_Msk | R_CANFD_GCR_POIE_Msk | 0x3)
CANFD_CFG_TX_PRIORITY (R_CANFD_GCFG_TPRI_Msk)	送信時のメッセージの優先順位のつけかたを選択します。いずれの場合も、番号が若いほうが優先されます。 (0) = メッセージ ID (R_CANFD_GCFG_TPRI_Msk) = バッファ番号（デフォルト）
CANFD_CFG_DLC_CHECK　0	有効に設定した場合、関連する AFL ルールに設定した値よりも受信メッセージの DLC フィールドが小さければ受信メッセージが拒否されます。 「DLC 置換有効」が選択され、メッセージが DLC チェックに合格した場合、DLC フィールドには関連する AFL ルールに設定した値が設定され、余剰データは破棄されます。 (0) = 無効（デフォルト） (R_CANFD_GCFG_DCE_Msk) = 有効 (R_CANFD_GCFG_DCE_Msk | R_CANFD_GCFG_DRE_Msk) = DLC 置換有効
CANFD_CFG_FD_OVERFLOW　0	宛先バッファより大きい受信メッセージを切り詰めるか拒否するかを設定します。 (0) = 拒否（デフォルト） (R_CANFD_GCFG_TPRI_Msk) = 切り詰める
CANFD_CFG_CANFDCLK_SOURCE　0	CAN FD クロックソースに PLL（デフォルト）またはクリスタルダイレクトを設定します。 (0) = PLL（デフォルト） (1) = クリスタルダイレクト
CANFD_CFG_RXMB_NUMBER　0	受信可能なメッセージバッファの数。メッセージバッファ受信用の割り込みがないので、代わりに RX FIFO を使用した受信をお勧めします。 この値を 0 に設定すると、RX メッセージバッファが無効化されます。 任意の値（0～32） デフォルト値は 0 です。

r_canfd_rx_config.h の設定オプション	
CANFD_CFG_RXMB_SIZE　0	すべての RX メッセージバッファのペイロードサイズ (0) = 8 バイト（デフォルト） (1) = 12 バイト (2) = 16 バイト (3) = 20 バイト (4) = 24 バイト (5) = 32 バイト (6) = 48 バイト (7) = 64 バイト
CANFD_CFG_GLOBAL_ERR_IPL　12	この割り込みは、以下で選択したエラー発生原因ごとに発生します。 任意の値（0～15） デフォルト値は (12) です。
CANFD_CFG_RX_FIFO_IPL　12	パラメータチェックをコードに含めるかどうかを選択します。 BSP_CFG_PARAM_CHECKING_ENABLE = デフォルト (BSP) 任意の値（0～15） デフォルト値は (12) です。
CANFD_CFG_RXFIFO0_INT_THRESHOLD　3U	RX FIFO 0 の割り込みしきい値を設定します。この設定は、割り込みモードを「しきい値」に設定した場合のみ適用できます。 (0U) = 1/8 フル (1U) = 1/4 フル (2U) = 3/8 フル (3U) = 1/2 フル（デフォルト） (4U) = 5/8 フル (5U) = 3/4 フル (6U) = 7/8 フル (7U) = フル
CANFD_CFG_RXFIFO0_DEPTH　3	RX FIFO 0 の段数を選択します。 (1) = 4 段 (2) = 8 段 (3) = 16 段（デフォルト） (4) = 32 段 (5) = 48 段
CANFD_CFG_RXFIFO0_PAYLOAD　7	RX FIFO 0 のメッセージペイロードサイズを選択します。 (0) = 8 バイト (1) = 12 バイト (2) = 16 バイト (3) = 20 バイト (4) = 24 バイト (5) = 32 バイト (6) = 48 バイト (7) = 64 バイト（デフォルト）

r_canfd_rx_config.h の設定オプション	
CANFD_CFG_RXFIFO0_INT_MODE ((R_CANFD_RFCR_RFIE_Msk | R_CANFD_RFCR_RFIM_Msk))	RX FIFO 0 の割り込みモードを設定します。しきい値モードでは、着信メッセージが以下で設定したしきい値を超えたときのみ割り込みが発生します。 (0) = 無効 (R_CANFD_RFCR_RFIE_Msk) = しきい値 (R_CANFD_RFCR_RFIE_Msk | R_CANFD_RFCR_RFIM_Msk) = フレーム毎（デフォルト）
CANFD_CFG_RXFIFO0_ENABLE　1	RX FIFO 0 を有効または無効にします。 (0) = 無効 (1) = 有効（デフォルト）
CANFD_CFG_RXFIFO1_INT_THRESHOLD　3U	RX FIFO 1 の割り込みしきい値を設定します。この設定は、割り込みモードを「しきい値」に設定した場合のみ適用できます。 (0U) = 1/8 フル (1U) = 1/4 フル (2U) = 3/8 フル (3U) = 1/2 フル（デフォルト） (4U) = 5/8 フル (5U) = 3/4 フル (6U) = 7/8 フル (7U) = フル
CANFD_CFG_RXFIFO1_DEPTH　3	RX FIFO 1 の段数を選択します。 (1) = 4 段 (2) = 8 段 (3) = 16 段（デフォルト） (4) = 32 段 (5) = 48 段
CANFD_CFG_RXFIFO1_PAYLOAD　7	RX FIFO 1 のメッセージペイロードサイズを選択します。 (0) = 8 バイト (1) = 12 バイト (2) = 16 バイト (3) = 20 バイト (4) = 24 バイト (5) = 32 バイト (6) = 48 バイト (7) = 64 バイト（デフォルト）
CANFD_CFG_RXFIFO1_INT_MODE ((R_CANFD_RFCR_RFIE_Msk | R_CANFD_RFCR_RFIM_Msk))	RX FIFO 1 の割り込みモードを設定します。しきい値モードでは、着信メッセージが以下で設定したしきい値を超えたときのみ割り込みが発生します。 (0) = 無効

r_canfd_rx_config.h の設定オプション	
	(R_CANFD_RFCR_RFIE_Msk) = しきい値 (R_CANFD_RFCR_RFIE_Msk | R_CANFD_RFCR_RFIM_Msk) = フレーム毎（デフォルト）
CANFD_CFG_RXFIFO1_ENABLE　0	RX FIFO 1 を有効または無効にします。 (0) = 無効（デフォルト） (1) = 有効
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	送信完了時に TX メッセージバッファが割り込みを発生させるかどうかを選択します。 無効 = 0ULL（デフォルト） 有効 = (1ULL << 0)
CANFD0_EXTENDED_CFG_WARNING_ERROR_INTERRUPTS 0U	有効化するエラー警告割り込み要因を選択します。 無効 = 0ULL（デフォルト） 有効 = R_CANFD_CHCR_EWIE_Msk
CANFD0_EXTENDED_CFG_PASSING_ERROR_INTERRUPTS 0U	有効化するエラーパッシブ割り込み要因を選択します。 無効 = 0U（デフォルト） 有効 = R_CANFD_CHCR_EPIE_Msk
CANFD0_EXTENDED_CFG_BUS_OFF_ENTRY_ERROR_INTERRUPTS　0U	有効化するチャネルバスオフ発生エラー割り込み要因を選択します。 無効 = 0U（デフォルト） 有効 = R_CANFD_CHCR_BOEIE_Msk
CANFD0_EXTENDED_CFG_BUS_OFF_RECOVERY_ERROR_INTERRUPTS　0U	有効化するチャネルバスオフ復帰エラー割り込み要因を選択します。 無効 = 0U（デフォルト） 有効 = R_CANFD_CHCR_BORIE_Msk
CANFD0_EXTENDED_CFG_OVERLOAD_ERROR_INTERRUPTS　0U	有効化するチャネルオーバーロードエラー割り込み要因を選択します。 無効 = 0U（デフォルト） 有効 = R_CANFD_CHCR_OLIE_Msk
CANFD0_CFG_IPL　12	この割り込みは、以下で選択したエラー発生原因ごとに発生します。 任意の値（0～15） デフォルト値は (12) です。
CANFD0_BIT_TIMING_CFG_BRP　1	公称ビットレートのクロック除数を指定します。 任意の値（1～1024） デフォルト値は (1) です。
CANFD0_BIT_TIMING_CFG_TSEG1　29	タイムセグメント 1 の値を選択します。この値の算出方法については、モジュール使用上の注意を確認してください。

r_canfd_rx_config.h の設定オプション	
	任意の値（2～256） デフォルト値は (29) です。
CANFD0_BIT_TIMING_CFG_TSEG2　10	タイムセグメント 2 の値を選択します。この値の算出方法については、モジュール使用上の注意を確認してください。 任意の値（2～128） デフォルト値は (10) です。
CANFD0_BIT_TIMING_CFG_SJW　4	同期ジャンプ幅の値を選択します。この値の算出方法については、モジュール使用上の注意を確認してください。 任意の値（1～128） デフォルト値は (4) です。
CANFD0_DATA_TIMING_CFG_BRP　1	データビットレートのクロック除数を指定します。 任意の値（1～1024） デフォルト値は (1) です。
CANFD0_DATA_TIMING_CFG_TSEG1　5	タイムセグメント 1 の値を選択します。この値の算出方法については、モジュール使用上の注意を確認してください。 任意の値（2～32） デフォルト値は (5) です。
CANFD0_DATA_TIMING_CFG_TSEG2　2	タイムセグメント 2 の値を選択します。この値の算出方法については、モジュール使用上の注意を確認してください。 任意の値（2～16） デフォルト値は (2) です。
CANFD0_DATA_TIMING_CFG_SJW　1	同期ジャンプ幅の値を選択します。この値の算出方法については、モジュール使用上の注意を確認してください。 任意の値（1～16） デフォルト値は (1) です。
CANFD0_EXTENDED_CFG_DELAY_COMPENSATION　1	有効に設定した場合、CAN FD モジュールは送信ビットと受信ビット間のすべてのトランシーバまたはバス遅延に対して自動補正を行います。 遅延補正を有効にした状態でビットタイミング値を手動で供給する場合は、正常動作のためにデータプリスケーラ値を 2 以下にしてください。 (0) = 無効 (1) = 有効（デフォルト） デフォルト値は (1) です。

2.10 インタフェースとインスタンス

このセクションでは、r_canfd_rx/inc の構造体を説明します。

2.10.1 CAN インタフェース

このセクションでは、r_canfd_rx/inc/ r_can_api.h の構造体を説明します。

CAN インタフェースは、異なる実装の CAN ドライバの共通機能と相互運用方法を提供します。これらの共通機能と相互運用方法により、同じ機能を持つ別々の CAN ドライバモジュールを上位層の呼び出し元関数でスワップイン／スワップアウトできます。このアプリケーションノートでは、CAN インタフェースは CAN FD により実装しています。

CAN インタフェースは以下の機能をサポートしています。

- 全二重 CAN 通信
- 汎用 CAN パラメータ設定
- 割り込み駆動型送受信処理
- イベントコードを返すコールバック関数
- トランザクション中のハードウェアリソースのロック

CAN インタフェースは、以下により実装します。

- Controller Area Network - Flexible Data (r_canfd)

データ構造体

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

列挙データ

enum	can_event_t
enum	can_operation_mode_t
enum	can_test_mode_t
enum	can_id_mode_t
enum	can_frame_type_t

Typedef

```
typedef void    can_ctrl_t
```

◆ can_info_t

struct can_info_t		
CAN ステータス情報		
データフィールド		
uint32_t	status	CAN ステータスレジスタからの有用な情報
uint32_t	rx_mb_status	RX メッセージバッファの新データフラグ
uint32_t	rx_fifo_status	RX FIFO の空フラグ
uint8_t	error_count_transmit	送信エラー数
uint8_t	error_count_receive	受信エラー数
uint32_t	error_code	エラーコード（読み出し後にクリアされます）

◆ can_bit_timing_cfg_t

struct can_bit_timing_cfg_t		
CAN ビットレート設定		
データフィールド		
uint32_t	baud_rate_prescaler	ボーレートプリスケーラ。有効な値：1～1024
uint32_t	time_segment_1	タイムセグメント 1 制御
uint32_t	time_segment_2	タイムセグメント 2 制御
uint32_t	synchronization_jump_width	同期ジャンプ幅

◆ can_frame_t

struct can_frame_t		
CAN データフレーム		
データフィールド		
uint32_t	id	CAN ID
can_id_mode_t	id_mode	標準 ID または拡張 ID (IDE)
can_frame_type_t	type	フレーム種別 (RTR)
uint8_t	data_length_code	CAN データ長コード (DLC)
uint32_t	options	実装固有のオプション
uint8_t	data[CAN_DATA_BUFFER_LENGTH]	CAN データ

◆ can_callback_args_t

struct can_callback_args_t			
CAN コールバックパラメータ定義			
データフィールド			
uint32_t		channel	デバイスチャネル番号
can_event_t		event	イベントコード
uint32_t		error	エラーコード
union	uint32_t	mailbox	割り込み発生原因のメールボックス番号

	uint32_t	buffer	割り込み発生原因のバッファ番号
can_frame_t *		p_frame	受信フレームへの非推奨ポインタ
void const *		p_context	コールバック中にユーザに提供するコンテキスト
can_frame_t		frame	受信したフレームデータ

◆ can_cfg_t

struct can_cfg_t	
CAN 設定	
データフィールド	
uint32_t	channel
	CAN チャネル
can_bit_timing_cfg_t *	p_bit_timing
	CAN ビットタイミング
void(*	p_callback)(can_callback_args_t *p_args)
	コールバック関数へのポインタ
void const *	p_context
	ユーザ定義のコールバックコンテキスト
void const *	p_extend
	CAN ハードウェア依存設定
uint8_t	ipl
	エラー／送信／受信割り込みの優先順位

◆ can_api_t

struct can_api_t
CAN の共有インタフェース定義
データフィールド
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_t operation_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 *const p_context, can_callback_args_t *const p_callback_memory)

フィールドドキュメンテーション

◆ open

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

CAN デバイス用のオープン関数

以下のように実装されます。
R_CANFD_Open()

パラメータ

[in]	p_ctrl	CAN 制御ブロックへのポインタ。ユーザが宣言する必要があります。
[in]	can_cfg_t	CAN 設定構造体へのポインタ。この構造体のすべての要素はユーザが設定する必要があります。

◆ write

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

CAN デバイス用の書き込み関数

以下のように実装されます。
R_CANFD_Write()

パラメータ

[in]	p_ctrl	CAN 制御ブロックへのポインタ
[in]	buffer_number	書き込み先のバッファ番号（メールボックスまたはメッセージバッファ）
[in]	p_frame	書き込み対象の CAN ID、DLC、データ、およびフレーム種別のフレームへのポインタ

◆ read

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

CAN デバイス用の読み込み関数

以下のように実装されます。
R_CANFD_Read()

パラメータ

[in]	p_ctrl	CAN 制御ブロックへのポインタ
[in]	buffer_number	読み出し対象のメッセージバッファ（番号）
[in]	p_frame	CAN ID、DLC、データ、およびフレーム種別を格納するポインタ

◆ close

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

CAN デバイス用のクローズ関数

以下のように実装されます。
R_CANFD_Close()

パラメータ

[in]	p_ctrl	CAN 制御ブロックへのポインタ

◆ 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)

CAN デバイス用のモード遷移関数

以下のように実装されます。
R_CANFD_ModeTransition()

パラメータ

[in]	p_api_ctrl	CAN 制御ブロックへのポインタ
[in]	operation_mode	宛先 CAN の動作状態
[in]	test_mode	宛先 CAN のテスト状態

◆ infoGet

fsp_err_t(* can_api_t::infoGet) (can_ctrl_t *const p_ctrl, can_info_t *const p_info)
CAN チャネル情報の取得

以下のように実装されます。
R_CANFD_InfoGet()

パラメータ

[in]	p_ctrl	チャネルの処理（チャネル制御ブロックへのポインタ）
[out]	p_info	チャネル固有データの返却先メモリアドレス

◆ 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 *const p_callback_memory)

コールバック関数、オプションのコンテキストポインタ、および作業用メモリポインタを指定します。

以下のように実装されます。
R_CANFD_CallbackSet()

パラメータ

[in]	p_ctrl	can_api_t::open 呼び出しに設定した制御ブロック
[in]	p_callback	登録対象のコールバック関数
[in]	p_context	コールバック関数に送信するポインタ
[in]	p_working_memory	コールバック構造体を割り当てることができる揮発性メモリへのポインタ。ここで割り当てられたコールバック引数は、コールバックの間だけ有効です。

◆ can_instance_t

struct can_instance_t

この構造体には、このインタフェースのインスタンスを使用するために必要なものが全て含まれています。

can_ctrl_t *	p_ctrl	このインスタンスの制御構造体へのポインタ
can_cfg_t const *	p_cfg	このインスタンスの設定構造体へのポインタ
can_api_t const *	p_api	このインスタンスの API 構造体へのポインタ

◆ can_ctrl_t

typedef void can_ctrl_t
CAN 制御ブロック。CAN FD API 呼び出しに渡すインスタンス固有の制御ブロックを割り当てます。 以下のように実装されます。 • canfd_instance_ctrl_t

◆ can_event_t

enum can_event_t	
CAN イベントコード	
列挙子	
CAN_EVENT_ERR_WARNING	エラー警告イベント
CAN_EVENT_ERR_PASSIVE	エラーパッシブイベント
CAN_EVENT_ERR_BUS_OFF	バスオフイベント
CAN_EVENT_BUS_RECOVERY	バスオフ復帰イベント
CAN_EVENT_MAILBOX_MESSAGE_LOST	メールボックスがオーバーランしています。
CAN_EVENT_ERR_BUS_LOCK	バスロック検出（連続した 32 のドミナントビット）
CAN_EVENT_ERR_CHANNEL	チャネルエラーが発生しました。
CAN_EVENT_TX_ABORTED	送信中止イベント
CAN_EVENT_RX_COMPLETE	受信完了イベント
CAN_EVENT_TX_COMPLETE	送信完了イベント
CAN_EVENT_ERR_GLOBAL	グローバルエラーが発生しました。
CAN_EVENT_TX_FIFO_EMPTY	送信 FIFO が空です。

◆ can_operation_mode_t

enum can_operation_mode_t	
CAN 動作モード	
列挙子	
CAN_OPERATION_MODE_NORMAL	CAN 正常動作モード
CAN_OPERATION_MODE_RESET	CAN リセット動作モード
CAN_OPERATION_MODE_HALT	CAN 停止動作モード
CAN_OPERATION_MODE_SLEEP	CAN スリープ動作モード
CAN_OPERATION_MODE_GLOBAL_OPERATION	CAN FD グローバル動作モード
CAN_OPERATION_MODE_GLOBAL_RESET	CAN FD グローバルリセットモード
CAN_OPERATION_MODE_GLOBAL_HALT	CAN FD グローバル Halt モード
CAN_OPERATION_MODE_GLOBAL_SLEEP	CAN FD グローバルスリープモード

◆ can_test_mode_t

enum can_test_mode_t
CAN テストモード
列挙子

CAN_TEST_MODE_DISABLED	CAN テストモード無効
CAN_TEST_MODE_LISTEN	CAN テストリッスンモード
CAN_TEST_MODE_LOOPBACK_EXTERNAL	CAN テスト外部ループバックモード
CAN_TEST_MODE_LOOPBACK_INTERNAL	CAN テスト内部ループバックモード
CAN_TEST_MODE_INTERNAL_BUS	CAN FD 内部 CAN バス通信テストモード

◆ can_id_mode_t

enum can_id_mode_t	
CAN ID モード	
列挙子	
CAN_ID_MODE_STANDARD	11 ビットの標準 ID を使用
CAN_ID_MODE_EXTENDED	29 ビットの拡張 ID を使用

◆ can_frame_type_t

enum can_frame_type_t	
CAN フレーム種別	
列挙子	
CAN_FRAME_TYPE_DATA	データフレーム
CAN_FRAME_TYPE_REMOTE	リモートフレーム

2.10.2 CAN FD インスタンス

このセクションでは、r_canfd_rx/inc/ r_canfd.h の構造体を説明します。

CAN FD インスタンスは CAN インタフェースの実際の実装の 1 つです。CAN FD インスタンスは CAN インタフェースの列挙データ、データ構造体、API プロトタイプを使用します。

データ構造体

```
struct    canfd_instance_ctrl_t
struct    canfd_afl_entry_t
struct    canfd_global_cfg_t
struct    canfd_extended_cfg_t
```

列挙データ

```
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 インスタンス制御ブロック		
データフィールド		
can_cfg_t const	* p_cfg	設定構造体へのポインタ
uint32_t	open	チャネルのオープン状態
can_operation_mode_t	operation_mode	CAN 動作モード
can_test_mode_t	test_mode	CAN テストモード
void	(* p_callback)(can_callback_args_t *)	コールバックへのポインタ
can_callback_args_t	* p_callback_memory	オプションのコールバック引数メモリへのポインタ
void const	* p_context	コールバック関数に渡すコンテキストへのポインタ

◆ canfd_afl_entry_t

struct canfd_afl_entry_t		
AFL エントリ		
データフィールド		
uint32_t	id	照合先 ID
can_frame_type_t	frame_type	フレーム種別（データ／リモート）
can_id_mode_t	id_mode	ID モード（標準／拡張）
uint32_t	mask_id	ID マスク
uint32_t	mask_frame_type	設定されたフレーム種別のフレームのみ承認
uint32_t	mask_id_mode	設定された ID モードのフレームのみ承認
canfd_minimum_dlc_t	minimum_dlc	承認する DLC の最小値（DLC チェックが有効の場合に有効）
canfd_rx_mb_t	rx_buffer	このルールで承認されたメッセージを受信する RX メッセージバッファ
canfd_rx_fifo_t	fifo_select_flags	このルールで承認されたメッセージを受信する RX FIFO

◆ canfd_global_cfg_t

struct canfd_global_cfg_t		
CAN FD グローバル設定		
データフィールド		
uint32_t	global_interrupts	グローバル制御オプション（GCR レジスタ設定）
uint32_t	global_config	グローバル設定オプション（GCFG レジスタ設定）
uint32_t	rx_fifo_config[2]	RX FIFO 設定（RFCRn レジスタ設定）
uint32_t	rx_mb_config	RX メッセージバッファの数とサイズ（RMCR レジスタ設定）

uint8_t	global_err_ipl	グローバルエラー割り込みの優先度
uint8_t	rx_fifo_ipl	RX FIFO 割り込みの優先度

◆ canfd_extended_cfg_t

struct canfd_extended_cfg_t		
CAN FD 拡張設定		
データフィールド		
canfd_afl_entry_t const *	p_afl	AFL ルール一覧
uint32_t	txmb_txi_enable	TX メッセージバッファイネーブルビットの配列
uint32_t	error_interrupts	エラー割り込みイネーブルビット
can_bit_timing_cfg_t *	p_data_timing	FD データ速度（ビットレート切り替え使用時）
uint8_t	delay_compensation	FD トランシーバ遅延補正（有効／無効）
canfd_global_cfg_t *	p_global_cfg	グローバル設定（グローバルエラーコールバックチャネルのみ）

◆ canfd_status_t

enum canfd_status_t	
CAN FD 状態	
列挙子	
CANFD_STATUS_RESET_MODE	チャネルはリセットモード
CANFD_STATUS_HALT_MODE	チャネルは Halt モード
CANFD_STATUS_SLEEP_MODE	チャネルはスリープモード
CANFD_STATUS_ERROR_PASSIVE	チャネルはエラーパッシブ状態
CANFD_STATUS_BUS_OFF	チャネルはバスオフ状態
CANFD_STATUS_TRANSMITTING	チャネルは送信中
CANFD_STATUS_RECEIVING	チャネルは受信中
CANFD_STATUS_READY	チャネルの通信準備が完了しました。
CANFD_STATUS_ESI	ESI フラグが設定された CAN FD メッセージを 1 件以上受信しました。

◆ canfd_error_t

enum canfd_error_t	
CAN FD エラーコード	
列挙子	
CANFD_ERROR_CHANNEL_BUS	バスエラー
CANFD_ERROR_CHANNEL_WARNING	エラー警告（TX/RX エラー数が 0x5F を超過）
CANFD_ERROR_CHANNEL_PASSIVE	エラーパッシブ（TX/RX エラー数が 0x7F を超過）
CANFD_ERROR_CHANNEL_BUS_OFF_ENTRY	バスオフ状態発生
CANFD_ERROR_CHANNEL_BUS_OFF_RECOVERY	バスオフ状態から復帰
CANFD_ERROR_CHANNEL_OVERLOAD	オーバーロード
CANFD_ERROR_CHANNEL_BUS_LOCK	バスがロックされました
CANFD_ERROR_CHANNEL_ARBITRATION_LOSS	アービトレーションロスト

CANFD_ERROR_CHANNEL_STUFF	スタッフエラー
CANFD_ERROR_CHANNEL_FORM	フォームエラー
CANFD_ERROR_CHANNEL_ACK	ACK エラー
CANFD_ERROR_CHANNEL_CRC	CRC エラー
CANFD_ERROR_CHANNEL_BIT_RECESSIVE	ビットエラー（リセッシブ）エラー
CANFD_ERROR_CHANNEL_BIT_DOMINANT	ビットエラー（ドミナント）エラー
CANFD_ERROR_CHANNEL_ACK_DELIMITER	ACK デリミタエラー
CANFD_ERROR_GLOBAL_DLC	DLC エラー
CANFD_ERROR_GLOBAL_MESSAGE_LOST	メッセージロスト
CANFD_ERROR_GLOBAL_PAYLOAD_OVERFLOW	FD ペイロードオーバーフロー
CANFD_ERROR_GLOBAL_TXQ_OVERWRITE	TX キューメッセージ上書き
CANFD_ERROR_GLOBAL_TXQ_MESSAGE_LOST	TX キューメッセージロスト
CANFD_ERROR_GLOBAL_CH0_SCAN_FAIL	チャネル 0 の RX スキャン失敗
CANFD_ERROR_GLOBAL_CH1_SCAN_FAIL	チャネル 1 の RX スキャン失敗
CANFD_ERROR_GLOBAL_CH0_ECC	チャネル 0 ECC エラー
CANFD_ERROR_GLOBAL_CH1_ECC	チャネル 1 ECC エラー

◆ canfd_tx_mb_t

enum canfd_tx_mb_t
CAN FD 送信メッセージバッファ (TX MB)

◆ canfd_rx_buffer_t

enum canfd_rx_buffer_t
CAN FD 受信バッファ (MB + FIFO)

◆ canfd_rx_mb_t

enum canfd_rx_mb_t
CAN FD 受信メッセージバッファ (RX MB)

◆ canfd_rx_fifo_t

enum canfd_rx_fifo_t
CAN FD 受信 FIFO (RX FIFO)

◆ canfd_minimum_dlc_t

enum canfd_minimum_dlc_t
CAN FD AFL の最小 DLC 設定

◆ canfd_frame_options_t

enum canfd_frame_options_t	
CAN FD フレームオプション	
列挙子	
CANFD_FRAME_OPTION_ERROR	エラー状態設定 (ESI)
CANFD_FRAME_OPTION_BRS	ビットレート切り替え (BRS) 有効
CANFD_FRAME_OPTION_FD	Flexible Data フレーム (FDF)

2.11 インスタンス構造体

本モジュールを使用するために、CANFD ソースコードでインスタンス構造体を作成しています。

内容は以下の通りです。

- 制御構造体へのポインタ (p_ctrl)
- 設定構造体へのポインタ (p_cfg)
- インスタンス API 構造体へのポインタ (p_api)

制御構造体、設定構造体、インスタンス API 構造体はデフォルト値で r_canfd_data.c ファイルに作成されています。

以下に、チャネル 0 用に作成したインスタンス構造体 (g_canfd0) を示します。この構造体には制御構造体 (g_canfd0_ctrl)、設定構造体 (g_canfd0_cfg)、インスタンス API 構造体 (g_canfd_on_canfd) が含まれます。

例：

```
/*CAN FD モジュールのチャネル 0 を使用するためのインスタンス構造体 */
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,
 };

/* 公称ビットレートの設定 */
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,
};
```

```
/* データ速度の設定*/
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 関数ポインタ   */
/* r_canfd_rx.c ファイル内の g_canfd_on_canfd */
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 コードサイズ

ROM（コードと定数）および RAM（グローバルデータ）のサイズは、「2.9 設定」に記載のビルド時の設定オプションによって決まります。C コンパイラの各コンパイルオプションを「2.5 サポートしているツールチェーン」の記載の通りにデフォルト値に設定した場合の参照値を以下の表に示します。コンパイルオプションのデフォルト値は、「最適化レベル：2、最適化タイプ：サイズ単位、データエンディアン：リトルエンディアン」です。コードサイズは、C コンパイラのバージョンとコンパイルオプションによって異なります。

領域	ビルド設定	サイズ（バイト）		
		CCRX	GCC	IAR
ROM	パラメータチェックあり	2533	4260	3210
ROM	パラメータチェックなし	2115	3404	2546
RAM		136	128	4
スタックの最大使用量		304		

2.13 コールバック関数

このモジュールでは、次のいずれかの条件が満たされたときに、ユーザーが設定したコールバック関数が呼び出されます。

（1）グローバル割り込み：

- 受信 FIFO 割り込みを検出したとき
- グローバルエラー割り込みを検出したとき （DLC エラーを検出、メッセージロストを検出、ペイロードオーバフローを検出 ）

（2）チャネル割り込み：

- 送信割り込みを検出したとき（送信成功割り込みを検出)
- チャネルエラー割り込みを検出したとき（エラーワーニングを検出、エラーパッシブを検出、バスオフ開始を検出、バスオフ復帰を検出、オーバロードを検出）

コールバック関数は、ユーザー関数のアドレスを g_canfd0_cfg 構造体の p_callback 引数に格納することによって設定されます。

p_callback 引数のデフォルト値は NULL です。ユーザーは、p_callback 引数の値を変更することにより、ユーザー関数に変更できます。

以下の例を参照して、p_callback 引数の値を NULL から User_callback に変更してください。

```
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 お使いのプロジェクトへの CAN FD FIT モジュールの追加方法

本モジュールは、使用するプロジェクトごとに追加する必要があります。ルネサスでは、Smart Configurator を使用した (1)、(3) の追加方法を推奨しています。ただし、Smart Configurator は一部の RX デバイスのみをサポートしています。未サポートの RX デバイスの場合は、(2) または (4) の方法を使用してください。

(1) e² studio の Smart Configurator を使用してプロジェクトに FIT モジュールを追加する場合

e² studio の Smart Configurator を使用すれば、お使いのプロジェクトに自動的にユーザ FIT モジュールが追加されます。詳細については、「RX スマート・コンフィグレータ ユーザーガイド: e² studio 編 (R20AN0451)」を参照してください。

(2) e² studio の FIT Configurator を使用してプロジェクトに FIT モジュールを追加する場合

e² studio の FIT Configurator を使用すれば、お使いのプロジェクトに自動的にユーザ FIT モジュールが追加されますす。詳細については、「RX ファミリ e² studio に組み込む方法 Firmware Integration Technology (R01AN1723)」を参照してください。

(3) CS+上で Smart Configurator を使用してプロジェクトに FIT モジュールを追加する場合

CS+上で Smart Configurator Standalone version を使用すれば、お使いのプロジェクトに自動的にユーザ FIT モジュールが追加されます。詳細については、「RX スマート・コンフィグレータ ユーザーガイド: e² studio 編 (R20AN0451)」を参照してください。

(4) CS+上でプロジェクトに FIT モジュールを追加する場合

CS+上で、お使いのプロジェクトに手動でユーザ FIT モジュールを追加します。詳細については、「RX ファミリ CS+に組み込む方法 Firmware Integration Technology (R01AN1826)」を参照してください。

2.15 for 文、while 文、do while 文について

本モジュールでは、レジスタの反映待ち処理等で for 文、while 文、do while 文（ループ処理）を使用しています。これらループ処理には、「WAIT_LOOP」をキーワードとしたコメントを記述しています。そのため、ループ処理にユーザがフェイルセーフの処理を組み込む場合は、「WAIT_LOOP」で該当の処理を検索できます。

以下に記述例を示します。

```
while 文の例：
/* WAIT_LOOP */
while(0 == SYSTEM.OSCOVFSR.BIT.PLOVF)
{
    /* PLL が安定したかどうかを確認するために必要な遅延時間 */
}

for 文の例：
/* 参照カウンタを 0 に初期化する。 */
/* WAIT_LOOP */
for (i = 0; i < BSP_REG_PROTECT_TOTAL_ITEMS; i++)
{
    g_protect_counters[i] = 0;
}

Do while 文の例：
/* リセット完了待ち */
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 関数

API とは一組の関数であり、これを使用することによって、CAN FD モジュールの細かい設定を気にせずに CAN FD を使用でき、ユーザアプリケーションとネットワーク上のノード間での通信が簡単に行えます。

CAN FD の設定と通信は CAN FD SFR（特殊関数レジスタ）レジスタを使って行います。詳細はお使いの MCU のユーザーズマニュアル ハードウェア編を参照してください。通信を成立させるには、CAN FD モジュールのレジスタを正しい順番で設定し、読み出す必要がありますが、CAN FD API を使えば、このような作業が簡単に行えます。この API は、数多くの煩雑な作業をユーザに代わって実行します。

R_CANFD_Open 関数でモジュールを初期化した後に必要な作業は、受信 (R_CANFD_Read) API 呼び出しと送信 (R_CANFD_Write) API 呼び出しの使用と、CAN FD のエラー状態の定期的な確認のみです。また、R_CANFD_Close 関数で CAN FD チャネルを閉じたり、R_CANFD_ModeTransition 関数で別のテストモードに切り替えたりもできます。

詳細については下記を参照してください。

ポイント

本 FIT モジュールには以下の API 関数があります。

関数名	説明
R_CANFD_Open()	動作用に CAN FD チャネルを開いて設定します。
R_CANFD_Close()	CAN FD チャネルを閉じます。
R_CANFD_Write()	CAN FD チャネルにデータを書き込みます。
R_CANFD_Read()	CAN FD メッセージバッファまたは FIFO からデータを読み出します。
R_CANFD_ModeTransition()	別のチャネル、グローバルモード、またはテストモードに切り替えます。
R_CANFD_InfoGet()	チャネルの CAN FD 状態とステータス情報を取得します。
R_CANFD_CallbackSet()	コールバック引数構造体のメモリを提供するオプションを使ってユーザコールバックを更新します。

戻り値

API の戻り値	説明
FSP_SUCCESS	処理が正常に完了しました。
FSP_ERR_IP_CHANNEL_NOT_PRESENT	要求されたチャネルはこのデバイスには存在しません。
FSP_ERR_ASSERTION	クリティカルアサーションに失敗しました。
FSP_ERR_CAN_INIT_FAILED	ハードウェア初期化に失敗しました。
FSP_ERR_CLOCK_INACTIVE	使用していないクロックがシステムクロックとして指定されました。
FSP_ERR_CAN_TRANSMIT_NOT_READY	送信中です。
FSP_ERR_INVALID_ARGUMENT	不正な入力パラメータです。
FSP_ERR_INVALID_MODE	未サポートまたは正しくないモードです。
FSP_ERR_NOT_OPEN	要求されたチャネルが設定されていないか、API がオープンしていません。
FSP_ERR_IN_USE	チャネルまたはモジュールが動作中または使用中です。
FSP_ERR_ALREADY_OPEN	要求されたチャネルは既に別の設定で開いています。
FSP_ERR_NO_CALLBACK_MEMORY	非安全コールバック用の非安全コールバックメモリが提供されていません。
FSP_ERR_BUFFER_EMPTY	利用可能なデータがバッファにありません。

R_CANFD_Open

動作用に CAN FD チャネルを開いて設定します。

形式

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

パラメータ

p_api_ctrl

CAN 制御ブロックへのポインタ。ユーザが宣言する必要があります。
詳細については、セクション「2.11 インスタンス構造体」を参照してください。

p_cfg

CAN 設定構造体へのポインタ。この構造体のすべての要素はユーザが設定する必要があります。
詳細については、セクション「2.11 インスタンス構造体」を参照してください。

戻り値

FSP_SUCCESS	チャネルが正常に開きました。
FSP_ERR_ALREADY_OPEN	ドライバは既に開いています。
FSP_ERR_IN_USE	チャネルは既に使用中です。
FSP_ERR_IP_CHANNEL_NOT_PRESENT	チャネルはこの MCU には存在しません。
FSP_ERR_ASSERTION	要求されたポインタは NULL でした。
FSP_ERR_CAN_INIT_FAILED	提供された公称ビットレートまはたデータビットレートが不正です。
FSP_ERR_CLOCK_INACTIVE	CAN FD ソースクロックが無効です（PLL または PLL2)。

プロパティ

r_canfd.h にプロトタイプ宣言されています。

r_canfd_rx.c で実装されます。

説明

動作用に CAN FD チャネルを開いて設定します。

例

```
/* CAN FD モジュールを初期化 */
    R_CANFD_Open(&g_canfd0_ctrl, &g_canfd0_cfg)
```

R_CANFD_Close

CAN FD チャネルを閉じます。

形式

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

パラメータ

p_api_ctrl

CAN 制御ブロックへのポインタ。
詳細については、セクション「2.11 インスタンス構造体」を参照してください。

戻り値

FSP_SUCCESS	チャネルが正常に閉じました。
FSP_ERR_NOT_OPEN	制御ブロックが開いていません。
FSP_ERR_ASSERTION	NULL ポインタがありました。

プロパティ

r_canfd.h にプロトタイプ宣言されています。

r_canfd_rx.c で実装されます。

説明

CAN FD チャネルを閉じます。

例

```
/* CAN FD モジュールを閉じる */
   R_CANFD_Close(&g_canfd0_ctrl);
```

R_CANFD_Write

CAN FD チャネルにデータを書き込みます。

形式

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

パラメータ

p_api_ctrl

CAN 制御ブロックへのポインタ。
詳細については、セクション「2.11 インスタンス構造体」を参照してください。

buffer

書き込み先のバッファ番号（メールボックスまたはメッセージバッファ）

p_frame

書き込み対象の CAN ID、DLC、データ、およびフレーム種別のフレームへのポインタ

戻り値

FSP_SUCCESS	処理に成功しました。
FSP_ERR_NOT_OPEN	制御ブロックが開いていません。
FSP_ERR_CAN_TRANSMIT_NOT_READY	送信中なので、データは書き込めません。
FSP_ERR_INVALID_ARGUMENT	データ長またはバッファ番号が不正です。
FSP_ERR_INVALID_MODE	非 FD フレームに FD オプションが設定されました。
FSP_ERR_ASSERTION	NULL ポインタがありました。

プロパティ

r_canfd.h にプロトタイプ宣言されています。

r_canfd_rx.c で実装されます。

説明

CAN FD チャネルにデータを書き込みます。

例

```
#define CAN_BUFFER_NUMBER_0          (0U)             //バッファ番号
can_frame_t g_canfd_tx_frame;                        //CAN FD 送信フレーム

/* 送信する tx フレームデータを設定 */
   for( uint16_t j = 0; j < SIZE_8; j++)
   {
     g_canfd_tx_frame.data[j] = (uint8_t) (j + 1);
   }

/* バス上でデータを送信 */
   err = R_CANFD_Write(&g_canfd0_ctrl, CAN_BUFFER_NUMBER_0, &g_canfd_tx_frame);
```

R_CANFD_Read

CAN FD メッセージバッファまたは FIFO からデータを読み出します。

形式

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

パラメータ

p_api_ctrl

CAN 制御ブロックへのポインタ。
詳細については、セクション「2.11 インスタンス構造体」を参照してください。

buffer

読み出し対象のメッセージバッファ（番号）

p_frame

CAN ID、DLC、データ、およびフレーム種別を格納するポインタ

戻り値

FSP_SUCCESS	処理に成功しました。
FSP_ERR_NOT_OPEN	制御ブロックが開いていません。
FSP_ERR_INVALID_ARGUMENT	不正なバッファ番号です。
FSP_ERR_ASSERTION	p_api_ctrl または p_frame が NULL です。
FSP_ERR_BUFFER_EMPTY	バッファまたは FIFO が空です。

プロパティ

r_canfd.h にプロトタイプ宣言されています。

r_canfd_rx.c で実装されます。

説明

CAN FD メッセージバッファまたは FIFO からデータを読み出します。

例

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

/* 受信した入力フレームの読み出し */
   err = R_CANFD_Read(&g_canfd0_ctrl, ZERO, &g_canfd_rx_frame);
```

R_CANFD_ModeTransition

別のチャネル、グローバルモード、またはテストモードに切り替えます。

形式

```
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);
```

パラメータ

p_api_ctrl

CAN 制御ブロックへのポインタ。
詳細については、セクション「2.11 インスタンス構造体」を参照してください。

operation_mode

宛先 CAN FD の動作状態

test_mode

宛先 CAN FD のテスト状態

戻り値

FSP_SUCCESS	処理に成功しました。
FSP_ERR_NOT_OPEN	制御ブロックが開いていません。
FSP_ERR_ASSERTION	NULL ポインタがありました。
FSP_ERR_INVALID_MODE	現在のグローバルモードから要求されたモードへの変更ができません。

プロパティ

r_canfd.h にプロトタイプ宣言されています。

r_canfd_rx.c で実装されます。

説明

別のチャネル、グローバルモード、またはテストモードに切り替えます。

例

```
/* 外部ループバックモードへの切り替え */
   R_CANFD_ModeTransition(&g_canfd0_ctrl, CAN_OPERATION_MODE_NORMAL,
(can_test_mode_t) CAN_TEST_MODE_LOOPBACK_EXTERNAL);
```

R_CANFD_InfoGet

チャネルの CAN FD 状態とステータス情報を取得します。

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

パラメータ

p_api_ctrl

チャネルの処理（チャネル制御ブロックへのポインタ）
詳細については、セクション「2.11 インスタンス構造体」を参照してください。

p_info

チャネル固有データの返却先メモリアドレス

戻り値

FSP_SUCCESS	処理に成功しました。
FSP_ERR_NOT_OPEN	制御ブロックが開いていません。
FSP_ERR_ASSERTION	NULL ポインタがありました。

プロパティ

r_canfd.h にプロトタイプ宣言されています。

r_canfd_rx.c で実装されます。

説明

チャネルの CAN FD 状態とステータス情報を取得します。

例

```
#define RESET_VALUE           (0x00)
/* rx フレームのステータス情報を格納する変数*/
 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,
 };

/*CAN FD 状態の取得*/
   R_CANFD_InfoGet(&g_canfd0_ctrl, &can_rx_info);
```

R_CANFD_CallbackSet

コールバック引数構造体のメモリを提供するオプションを使ってユーザコールバックを更新します。

can_api_t::callbackSet を実装します。

形式

```
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);
```

パラメータ

p_api_ctrl

can_api_t::open 呼び出しに設定した制御ブロック。
詳細については、セクション「2.11 インスタンス構造体」を参照してください。

p_callback

登録対象のコールバック関数

p_context

コールバック関数に送信するポインタ

p_callback_memory

コールバック構造体を割り当てることができる揮発性メモリへのポインタ。ここで割り当てられたコールバック引数は、コールバックの間だけ有効です。

戻り値

FSP_SUCCESS	コールバックの更新に成功しました。
FSP_ERR_ASSERTION	要求されたポインタが NULL です。
FSP_ERR_NOT_OPEN	制御ブロックがオープンしていません。
FSP_ERR_NO_CALLBACK_MEMORY	p_callback は非安全状態であり、p_callback_memory は安全状態または NULL です。

プロパティ

r_canfd.h にプロトタイプ宣言されています。

r_canfd_rx.c で実装されます。

説明

コールバック引数構造体のメモリを提供するオプションを使ってユーザコールバックを更新します。

例

```
/* コールバック関数の設定 */
   R_CANFD_CallbackSet(&g_canfd0_ctrl, canfd_callback, NULL, NULL);
```

例

基本例

アプリケーションで CAN FD モジュールを最小限使用する場合の基本例は以下の通りです。従来の CAN で実装されます。新規メッセージの着信があれば、プログラムが読み出します。もしくは、ユーザが sw2 を押下してメッセージを CAN バスに送信できます。

注記
本ソフトウェアには RX メッセージバッファ用の割り込みがないので、受信には RX FIFO を使うことをお勧めします。

```
#define CAN_BUFFER_NUMBER_0          (0U)             //バッファ番号
#define ZERO (0U)
#define CAN_ID                (0x1100) //送信フレームの ID
#define CAN_CLASSIC_FRAME_DATA_BYTES (8U) //従来のフレームのデータ長コード
#define SIZE_8   (8u)
extern can_bit_timing_cfg_t g_canfd0_bit_timing_cfg; /* extern によるデフォルト値の
変更 */
can_frame_t g_canfd_tx_frame;                    //CAN FD 送信フレーム
can_frame_t g_canfd_rx_frame;
#define RESET_VALUE           (0x00)

/* rx フレームのステータス情報を格納する変数*/
 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,
 };
/* 承認フィルタ配列パラメータ
CANFD_CFG_AFL_CH0_RULE_NUM = 1 */
/* 承認フィルタ配列パラメータ */
#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] =
{
    /* 拡張 ID が 0x1000～0x1FFF のメッセージを承認 */
    /* 承認する ID、ID 種別、フレーム種別を指定します。*/
    {
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;
    /* 公称レート：1Mbps; DLL：40MHz */
    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;

    /* 送信する tx フレームデータを設定 */
    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);

    /* CRX0 端子の設定 */
    PORT3.PMR.BIT.B3 = 0U;
    PORT3.PDR.BIT.B3 = 0U;
    MPC.P33PFS.BYTE = 0x10U;
    PORT3.PMR.BIT.B3 = 1U;
    PORT3.PDR.BIT.B3 = 0U;

    /* CTX0 端子の設定 */
    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;
    /* API の初期化*/
    err = R_CANFD_Open(&g_canfd0_ctrl, &g_canfd0_cfg);

while(1)
{
    /* 新規メッセージの有無を確認... */
    can_read_operation();

    /* sw2 を押下してメッセージを CAN バスに送信 */
    read_switches();
}

/* sw2 の押下時に sw2_func()を呼び出す */
void sw2_func(void)
{
    can_operation();
}/* sw2_func()の終了*/

void can_operation(void)
{

    /* 送信フレームパラメータの更新 */
    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;

    /* 従来の CAN 8 バイト */
    g_canfd_tx_frame.data_length_code = CAN_CLASSIC_FRAME_DATA_BYTES;
    g_canfd_tx_frame.options = ZERO;

    /* 従来の CAN フレームでデータを送信 */
    can_write_operation(g_canfd_tx_frame);
}

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

    /* tx_frame でバッファ#0 からデータを送信 */
    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;

    /* CAN FD 送信のステータス情報を取得 */
    err = R_CANFD_InfoGet(&g_canfd0_ctrl, &can_rx_info);

    /* FIFO 内に受信したデータがあるか確認 */
    if(can_rx_info.rx_mb_status)
    {
        /* 受信した入力フレームの読み出し */
        err = R_CANFD_Read(&g_canfd0_ctrl, ZERO, &g_canfd_rx_frame);
    }
}
```

Flexible Data

この例では、ビットレート切り替えあり（公称レート = 1Mbps、データレート = 8Mbps）で FD メッセージを送信しています。新規メッセージの着信があれば、プログラムが読み出します。もしくは、ユーザがスイッチ 2 を押下してメッセージを CAN バスに送信できます。

```
#define CAN_BUFFER_NUMBER_0          (0U)             //バッファ番号
#define ZERO (0U)
#define CAN_ID                (0x1100) //送信フレームの ID
#define CAN_FD_DATA_LENGTH_CODE (64U)   //従来のフレームのデータ長コード
#define SIZE_64     (64u)
extern can_bit_timing_cfg_t g_canfd0_bit_timing_cfg; /* extern によるデフォルト値の
変更 */
extern can_bit_timing_cfg_t g_canfd0_data_timing_cfg; /* extern によるデフォルト値の
変更 */
can_frame_t g_canfd_tx_frame;                   //CAN FD 送信フレーム
can_frame_t g_canfd_rx_frame;
#define RESET_VALUE           (0x00)

/* rx フレームのステータス情報を格納する変数*/
 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,
 };
/* 承認フィルタ配列パラメータ
CANFD_CFG_AFL_CH0_RULE_NUM = 1 */
/* 承認フィルタ配列パラメータ */
#define CANFD_FILTER_ID (0x00001000)
#define MASK_ID               (0x0FFFF000)
#define MASK_ID_MODE    (1)
#define ZERO                    (0U) //配列のインデックス値
const canfd_afl_entry_t p_canfd0_afl[CANFD_CFG_AFL_CH0_RULE_NUM] =
{
    /* 拡張 ID が 0x1000～0x1FFF のメッセージを承認 */
    /* 承認する ID、ID 種別、フレーム種別を指定します。 */
    {
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;
    /* 公称レート：1Mbps; DLL：40MHz */
    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;

    /* データレート：8Mbps;  DLL：40MHz */
    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;

    /* 送信する tx フレームデータを設定 */
    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);

    /* CRX0 端子の設定 */
    PORT3.PMR.BIT.B3 = 0U;
```

```
    PORT3.PDR.BIT.B3 = 0U;
    MPC.P33PFS.BYTE = 0x10U;
    PORT3.PMR.BIT.B3 = 1U;
    PORT3.PDR.BIT.B3 = 0U;

    /* CTX0 端子の設定 */
    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;
    /* API の初期化 */
    err = R_CANFD_Open(&g_canfd0_ctrl, &g_canfd0_cfg);

while(1)
{
    /* 新規メッセージの有無を確認... */
    can_read_operation();

    /* sw2 を押下してメッセージを CAN バスに送信 */
    read_switches();
}

/* sw2 の押下時に sw2_func()を呼び出す */
void sw2_func(void)
{
    canfd_operation();
}/* sw2_func()の終了 */

void canfd_operation(void)
{

    /* 送信フレームパラメータの更新 */
    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 64 バイト*/
    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;

    /* FD CAN フレームでデータを送信 */
    can_write_operation(g_canfd_tx_frame);
}
void can_read_operation(void)
{
    fsp_err_t err = FSP_SUCCESS;

    /* CAN FD 送信のステータス情報を取得 */
    err = R_CANFD_InfoGet(&g_canfd0_ctrl, &can_rx_info);

    /* FIFO 内に受信したデータがあるか確認 */
    if(can_rx_info.rx_mb_status)
    {
        /* 受信した入力フレームの読み出し */
        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;

    /* tx_frame でバッファ#0 からデータを送信 */
    err = R_CANFD_Write(&g_canfd0_ctrl, CAN_BUFFER_NUMBER_0,
&can_transmit_frame);
}
```

4. 端子設定

CAN FD FIT モジュールを使用する場合は、マルチファンクションピンコントローラ (MPC) で周辺機能の入出力信号を端子に割り当ててください。本ドキュメントでは、以降、端子の割り当ては「端子設定」と呼びます。

端子設定は、R_CANFD_Open 関数を呼び出した後に行ってください。

e^2 studio で端子設定を行う場合、FIT Configurator または Smart Configurator の端子設定機能が使用できます。端子設定機能の使用時、FIT Configurator または Smart Configurator の端子設定画面で選択したオプションに応じてソースファイルが生成されます。そのソースファイルで定義した関数を呼び出すことによって端子が設定されます。詳細については、「表 4.1　Smart Configurator が出力する関数」を参照してください。

表 4.1　Smart Configurator が出力する関数

使用マイコン	出力される関数	備考
すべてのマイコン	R_CANFD_PinSet_CANFDx	x：チャネル番号

5. デモプロジェクト

デモプロジェクトには main()関数が含まれ、この関数は FIT モジュールおよび FIT モジュールが依存するモジュール（例：r_bsp）を使用します。本 FIT モジュールには以下のデモプロジェクトが含まれます。

5.1 ワークスペースにデモを追加

デモプロジェクトは、本アプリケーションノートで提供されるファイルの FITDemos サブディレクトリにあります。ワークスペースにデモプロジェクトを追加するには、「ファイル」→「インポート」を選択し、「インポート」ダイアログから「一般」の「既存プロジェクトをワークスペースへ」を選択して「次へ」ボタンをクリックします。「インポート」ダイアログで「アーカイブ・ファイルの選択」ラジオボタンを選択し、「参照」ボタンをクリックして FITDemos サブディレクトリを開き、使用するデモの zip ファイルを選択して「完了」をクリックします。

CAN FD アプリケーションデモコードのファイルは、..¥src ディレクトリにある main.c、および switches.c です。

デモを実行するには、以下の説明に従って、圧縮 e² studio プロジェクト（*r01an6130esxxxx-rx-canfd.zip*）を e² studio にインポートします。

5.1.1 e2 studio でプロジェクトをインポートしてデバッグする

(a) 新規にワークスペースを作成する

ワークスペースを作成したい場所に空フォルダを作成します。

e2 studio を開始し、ワークスペースとして、上記で作成したフォルダを指定します。

Workbench アイコン（「ようこそ」ウィンドウの右下）をクリックします。

以下の手順を続けます。

(b) 既存のワークスペースを使用する

「インポート」を選択します。

「一般」→「既存プロジェクトをワークスペースへ」を選択します。または、アーカイブファイルかディレクトリから新しいプロジェクトを作成します。

- デモのコードがエクスポートして作成されたアーカイブ ZIP ファイルの場合、そのファイルを参照します。
- デモのコードがソースコード（.project ファイル）と一緒に e² studio プロジェクトのディレクトリにある場合、プロジェクトのルートディレクトリを参照します。コードをワークスペース（.metadata ディレクトリがある場所）に持ちたい場合、「プロジェクトをワークスペースにコピーする」を選択してください。

[終了]ボタンをクリックします。

アーティファクト名を$(ProjName)に変更します。「プロジェクト」→「プロパティ」→「C/C++ ビルド」→「設定」を選択します。こうすることで、プロジェクト名を変更した場合も、正しくビルドされます。

これでワークスペースにデモプロジェクトがインポートできました。同じワークスペースに別のプロジェクトをインポートすることもできます。

(c) コードを実行する

デバッグセッションを作成してダウンロードし、コードを実行します。

5.1.2　デモを実行する

パッケージには、それぞれ 1Mbps と 5Mbps の公称ビットレートとデータビットレートでデータを送受信するデモが含まれています。

デモはいくつかの異なる方法で物理的に設定できます。

2 つのボードにプログラムを書き込み、CAN バスでお互いを接続します。

フレームをデモと送受信するために、CAN FD バスモニタを使用します。例：Kvaser Leaf Pro HS v2

CAN_TEST_MODE_LOOPBACK_INTERNAL を使用すると、通信は内部で行われ、外部バスは不要となります。

(a)　動作説明

デモでは、デフォルトの TX-ID が CAN_ID、RX-ID が CANFD_FILTER_ID のフレームを送受信します。デモは canfd0_callback にコールバック関数を設定し、CAN FD モジュールの必要な I/O ピンを初期化することから始まります。次に、R_CANFD_Open が CAN FD モジュールをオープンするために呼び出されます。オープン処理が成功すると、プログラムはループ処理に入り、新しいメッセージまたはスイッチの押し下げを待ちます。

(b)　ユーザアクション

SW1 を押すと、内部ループバックテストモードで送受信を行います。

SW2 を押すと、CAN FD アナライザまたはほかのボードへメッセージを送信します。

5.2　Renesas デバッグコンソール

E1/E20 から e^2 studio のデバッグコンソールに対してトレースデータを有効にすると、ユーザアプリケーションからリアルタイムでデータを出力することができます。これによって、C 言語の printf()を使って、トレースした文字列を送信して、標準出力が可能になります。この場合、標準出力は E1/E20 デバッグレジスタになります。

これを行うには、../r_config/r_bsp_config.h の BSP_CFG_IO_LIB_ENABLE を“1”に設定します。

デバッグコンソールを有効にするために、マクロが自動的にコードを有効にします。そのためには以下の手順を行ってください。

1. INIT_IOLIB()が呼び出されていることを、resetprog.c で確認してください。
2. lowlvl.c 内のコードには charput および charget 関数を含む必要があります。これによって、最下レベルの入出力処理に E1/E20 デバッグレジスタが使用されます。
   例えば、charput には以下を含む必要があります。
```
/* 送信バッファが空になるのを待機 */
while(0 != (E1_DBG_PORT.DBGSTAT & TXFL0EN));
```
3. printf を使用したい場合、ファイルに<stdio.h>を記載してください。
   printf()を呼び出すファイルには以下を追加します。
```
#if BSP_CFG_IO_LIB_ENABLE
    #include <stdio.h>
#endif
```

4. e² studio にて、以下のように[Renesas デバッグ仮想コンソールの有効化／無効化]および[コンソールのピン留め]の両方をクリックして、「デバッグコンソール」ウィンドウを追加します。E1/E20 のプリントバッファを空にし、また、コードの実行がブロックされないようにするには、これらをオンにする必要があります。

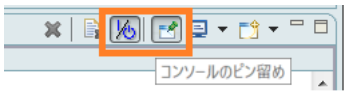

図 1　デバッグコンソールの制御ボタン。

5. 何もプリントされない場合、[Renesas デバッグ仮想コンソールのクリア]（[Renesas デバッグ仮想コンソールの有効化／無効化]アイコンの左のアイコン）を数回押します。

6. テストモード

特定機能をテストできるようにするために、CAN FD モジュールをテストモードに設定できます。これらの機能は特別な目的のためだけに提供されており、CAN FD モジュールをテストモードに設定する場合には注意が必要です。

テストモードは 2 つのグループに大別されます。

- チャネル別テストモード
- グローバルテストモード（現在のソースコードでは未サポート）

6.1 チャネル別テストモード

CAN FD チャネルは以下のテストモードに設定できます。

— 基本テストモード
— リッスンオンリモード
— 外部ループバックモード
— 内部ループバックモード
— 制限付き動作モード（現在のソースコードでは、このテストモードは未サポートです）

テストモードへの切り替えには R_CANFD_ModeTransition を使用してください。

6.1.1 基本テストモード

基本テストモードは、リッスンオンリモードおよびセルフテストモード以外で特定のテスト設定を有効にする必要がある場合に使用してください。

6.1.2 リッスンオンリモード（バスモニタ）

リッスンオンリモード、すなわちバスモニタモードでは、ノードは静止状態です。リッスンオンリモードでのノードは ACK メッセージやエラーフレームなどを送信しません。この方法では、バストラフィックに影響することなくノードをテストできます。

【注意】

1. リッスンオンリのノードからはフレームを送信しないでください。これは不正な動作であり、CAN FD モジュールでは対応していません。
2. ネットワーク上にあるノードが 2 つのみで、そのうち 1 つがリッスンオンリモードの場合、もう 1 つのノードは ACK を受信せずに送信を試行し続けます。
3. リッスンオンリモードへの遷移箇所をコード内で明確にして、再度リッスンオンリモードを無効にすることを忘れないようにしてください。

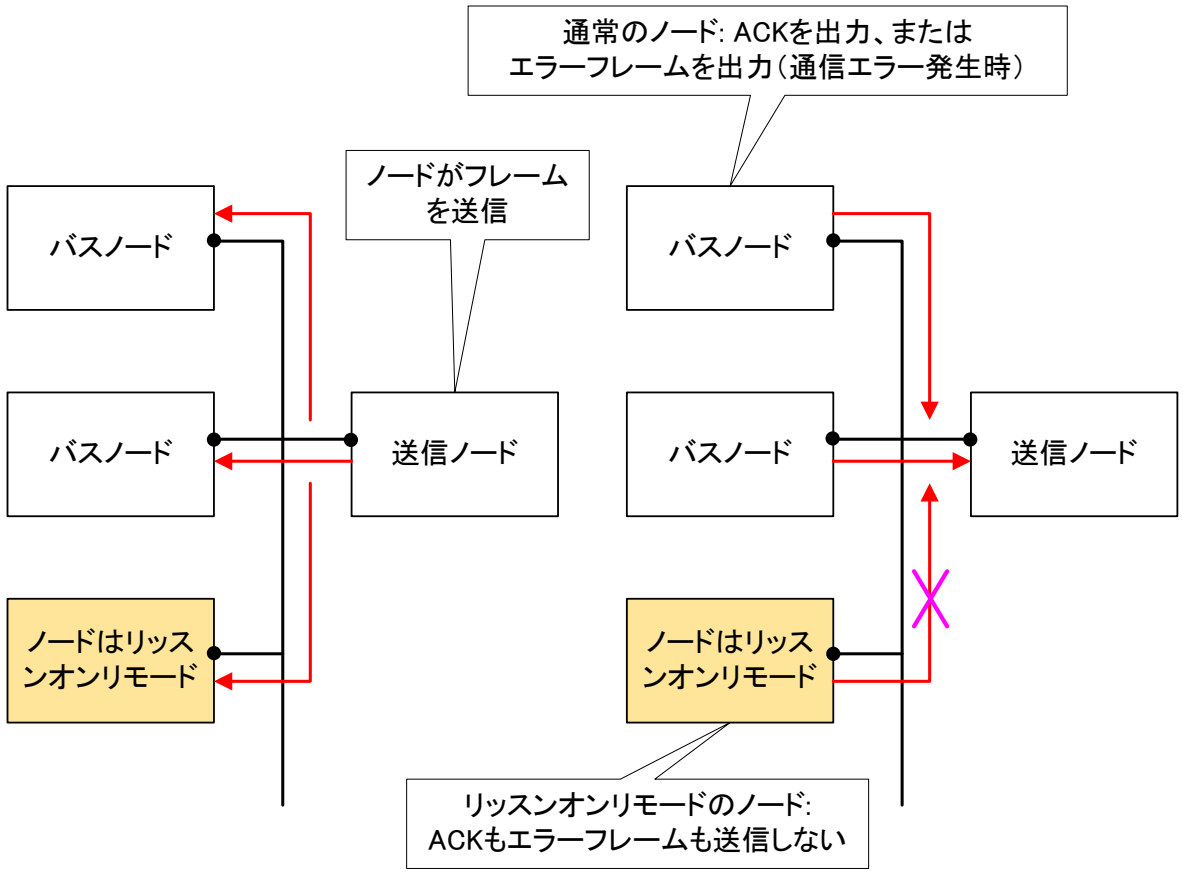

図 2. リッスンオンリモードのノード: ACK もエラーメッセージも送信しない

リッスンオンリモードでのノードは ACK メッセージやエラーフレームなどを送信しません。

リッスンオンリモードは、既存の CAN バスに追加した新規ノードを稼働する場合に有用です。実稼働開始前に、新たに接続されたノードのアプリケーションでこのモードを使用して、フレームが正常に受信されていることを確認できます。

一般的な使用例は、新規ユニットを実稼働する前のバスの通信速度の検出です。リッスンオンリモードは Bosch CAN 仕様ではありませんが、ビットレート検出においては ISO-11898 に準拠することが要求されます。

6.1.3　ループバック

ループバックモードでは、同じメッセージを受信するようにバッファを設定すると、ノードが送信したメッセージをそのノード自身で受信します。このモードは、アプリケーションのテストや、アプリケーションのデバッグ時の自己診断に有用です。

6.1.3.1　内部ループバックモード：CAN バスを介さずにノードをテストする

内部ループバックモード、すなわちセルフテストモードでは、バスに接続せずに CAN FD バッファを介して通信が行えます。ノードはデータフレームの ACK ビットを使って、自身のデータを認識します。また、受信バッファに同じ CAN FD ID が設定されていた場合、ノードは送信した自身のメッセージをその受信バッファに格納します。通常、このような動作は不可能です。

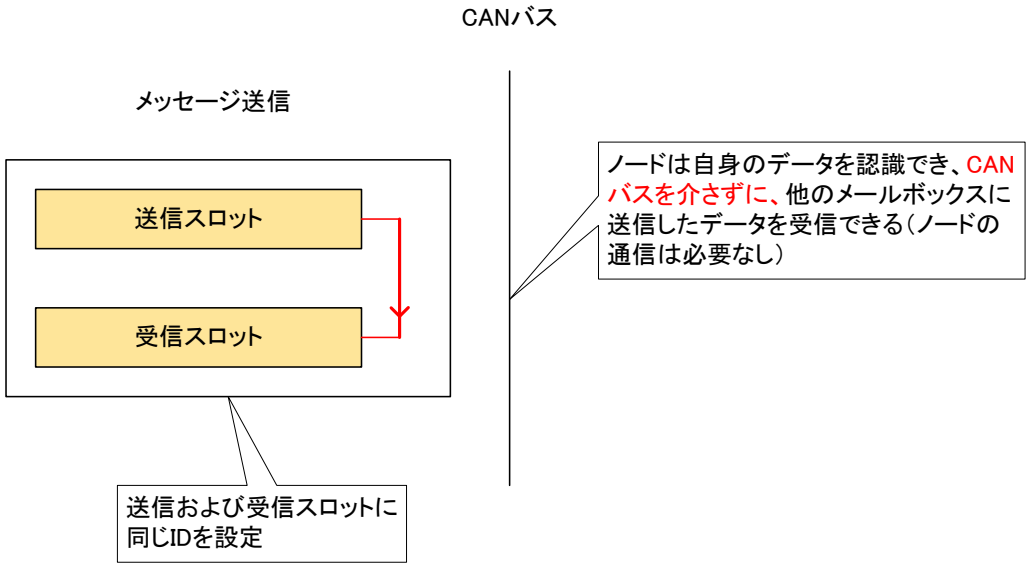

図 3. CAN 内部ループバックモード

CAN 内部ループバックモードでは、CAN バスを接続しない状態でノードの機能をテストできます。

内部ループバックはテスト時に有用です。このモードでは、バス上のノードが 1 つしかない場合、CAN FD コントローラが ACK 未受信による CAN FD エラーを送信せずに動作できるので、送信したフレームをノード自身で認識します。

6.1.3.2 外部ループバックモード：バス上でノードをテストする

外部ループバックは、ノードに接続された CAN バスが必要であり、メッセージもバスに送信されるという点を除いて、内部ループバックと同じです。内部ループバックと同様、送信されたメッセージはノード自身で認識されるので、ノードはバス上に 1 つで構いません。ノードを単体でテストできることが、この方法の利点です。

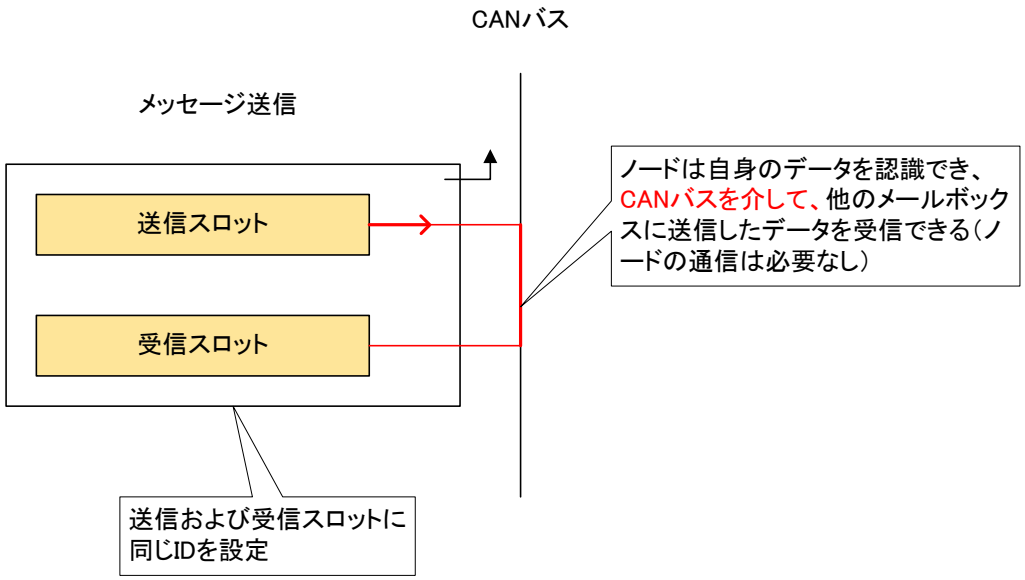

図 4　外部ループバック

メッセージは CAN バスに送信され、同じノードで受信できます。これは、コードのテスト時と、バス上のノードが 1 つだけの場合に有用です。

6.1.4 動作制限

現在のソースコードでは動作制限モードは未サポートです。

6.2 グローバルテストモード有効化レジスタ

現在のソースコードではグローバルテストモードは未サポートです。

7. 付録

7.1 動作確認環境

このセクションでは CAN FD FIT モジュールの動作確認用の環境について説明します。

表 7.1 動作確認環境 (Rev.1.20)

項目	内容
統合開発環境 (IDE)	ルネサスエレクトロニクス製 e^2 studio Version 22.7.0 IAR Embedded Workbench for Renesas RX 4.20.3
C コンパイラ	ルネサスエレクトロニクス製 RX ファミリ用 C/C++コンパイラパッケージ V3.04.00 コンパイルオプション：統合開発環境のデフォルト設定に以下のオプションを追加します。 -lang = c99
	GCC for Renesas RX 8.3.0.202104 コンパイルオプション：統合開発環境のデフォルト設定に以下のオプションを追加します。 -std=gnu99 リンクオプション：「Optimize size (サイズ最適化) (-Os)」を使用する場合、統合開発環境のデフォルト設定に以下のユーザ定義オプションを追加します。 -Wl,--no-gc-sections これは、FIT 周辺モジュール内で宣言された割り込み関数をリンカが誤って破棄してしまうという GCC リンカ問題を回避するための対策です。
	IAR C/C++ Compiler for Renesas RX version 4.20.3 コンパイルオプション：統合開発環境のデフォルト設定
エンディアン	ビッグエンディアン／リトルエンディアン
モジュールのリビジョン	Rev.1.20
使用ボード	Renesas Starter Kit for RX660 (product number. RTK556609HC10000BJ)

表 7.2　動作確認環境 (Rev.1.10)

項目	内容
統合開発環境 (IDE)	ルネサスエレクトロニクス製 e² studio Version 22.7.0 IAR Embedded Workbench for Renesas RX 4.20.3
C コンパイラ	ルネサスエレクトロニクス製 RX ファミリ用 C/C++コンパイラパッケージ V3.04.00 コンパイルオプション：統合開発環境のデフォルト設定に以下のオプションを追加します。 -lang = c99
	GCC for Renesas RX 8.3.0.202104 コンパイルオプション：統合開発環境のデフォルト設定に以下のオプションを追加します。 -std=gnu99 リンクオプション：「Optimize size (サイズ最適化) (-Os)」を使用する場合、統合開発環境のデフォルト設定に以下のユーザ定義オプションを追加します。 -Wl,--no-gc-sections これは、FIT 周辺モジュール内で宣言された割り込み関数をリンカが誤って破棄してしまうという GCC リンカ問題を回避するための対策です。
	IAR C/C++ Compiler for Renesas RX version 4.20.3 コンパイルオプション：統合開発環境のデフォルト設定
エンディアン	ビッグエンディアン／リトルエンディアン
モジュールのリビジョン	Rev.1.10
使用ボード	Renesas Starter Kit for RX660 (product number. RTK556609HC10000BJ)

表 7.3　動作確認環境 (Rev.1.00)

項目	内容
統合開発環境 (IDE)	ルネサスエレクトロニクス製 e² studio Version 22.4.0 IAR Embedded Workbench for Renesas RX 4.20.3
C コンパイラ	ルネサスエレクトロニクス製 RX ファミリ用 C/C++コンパイラパッケージ V3.04.00 コンパイルオプション：統合開発環境のデフォルト設定に以下のオプションを追加します。 -lang = c99
	GCC for Renesas RX 8.3.0.202104 コンパイルオプション：統合開発環境のデフォルト設定に以下のオプションを追加します。 -std=gnu99 リンクオプション：「Optimize size (サイズ最適化) (-Os)」を使用する場合、統合開発環境のデフォルト設定に以下のユーザ定義オプションを追加します。 -Wl,--no-gc-sections これは、FIT 周辺モジュール内で宣言された割り込み関数をリンカが誤って破棄してしまうという GCC リンカ問題を回避するための対策です。
	IAR C/C++ Compiler for Renesas RX version 4.20.3 コンパイルオプション：統合開発環境のデフォルト設定
エンディアン	ビッグエンディアン／リトルエンディアン
モジュールのリビジョン	Rev.1.00
使用ボード	Renesas Starter Kit for RX660 (product number. RTK556609HC10000BJ)

7.2 トラブルシューティング

(1) Q：本 FIT モジュールをプロジェクトに追加しましたが、ビルド実行すると、「Could not open source file "platform.h".（ソースファイル「platform.h」が開けません。）」というエラーが発生します。
A：FIT モジュールがプロジェクトに正しく追加されていない可能性があります。FIT モジュールの追加方法が正しいかどうかを以下のドキュメントで確認してください。

- CS+を使用している場合：
  アプリケーションノート「RX ファミリ CS+に組み込む方法 Firmware Integration Technology (R01AN1826)」
- e² studio を使用している場合：
  アプリケーションノート「RX ファミリ e² studio に組み込む方法 Firmware Integration Technology (R01AN1723)」

本 FIT モジュールを使用する場合、ボードサポートパッケージ FIT モジュール（BSP モジュール）もプロジェクトに追加する必要があります。アプリケーションノート「ボードサポートパッケージモジュール Firmware Integration Technology (R01AN1685)」を参照してください。

(2) Q：本 FIT モジュールをプロジェクトに追加しましたが、ビルド実行すると、「This MCU is not supported by the current r_canfd_rx module.（このマイコンは、現在の r_canfd_rx モジュールでは未サポートです。）」というエラーが発生します。
A：追加した FIT モジュールが、お使いのプロジェクトで選択したターゲットデバイスをサポートしていない可能性があります。追加した FIT モジュールのサポートデバイスを確認してください。

(3) Q：本 FIT モジュールをプロジェクトに追加しましたが、ビルド実行すると、コンフィグ設定が間違っている場合のエラーメッセージが発生します。
A：r_canfd_rx_config.h ファイルの設定が間違っている可能性があります。r_canfd_rx_config.h ファイルを確認してください。設定が間違っている場合、正しい値を設定してください。詳細については、「2.9 設定」を参照してください。

テクニカルアップデートの対応について

本モジュールは以下のテクニカルアップデートの内容を反映しています。

なし

改訂記録

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2022.05.31	−	初版発行
1.10	2022.06.28	48, 49, 50 55 プログラム	デモプロジェクトを更新 「7.1 動作確認環境」： Rev.1.10 に対応する表を追加。 デモプロジェクトを更新
1.20	2023.01.06	55 プログラム	「7.1 動作確認環境」： Rev.1.20 に対応する表を追加。 関数 canfd_channel_tx_isr()において、TXRF フラグがクリアされないのを修正。

製品ご使用上の注意事項

ここでは、マイコン製品全体に適用する「使用上の注意事項」について説明します。個別の使用上の注意事項については、本ドキュメントおよびテクニカルアップデートを参照してください。

1. 静電気対策
   CMOS 製品の取り扱いの際は静電気防止を心がけてください。CMOS 製品は強い静電気によってゲート絶縁破壊を生じることがあります。運搬や保存の際には、当社が出荷梱包に使用している導電性のトレーやマガジンケース、導電性の緩衝材、金属ケースなどを利用し、組み立て工程にはアースを施してください。プラスチック板上に放置したり、端子を触ったりしないでください。また、CMOS 製品を実装したボードについても同様の扱いをしてください。
2. 電源投入時の処置
   電源投入時は、製品の状態は不定です。電源投入時には、LSI の内部回路の状態は不確定であり、レジスタの設定や各端子の状態は不定です。外部リセット端子でリセットする製品の場合、電源投入からリセットが有効になるまでの期間、端子の状態は保証できません。同様に、内蔵パワーオンリセット機能を使用してリセットする製品の場合、電源投入からリセットのかかる一定電圧に達するまでの期間、端子の状態は保証できません。
3. 電源オフ時における入力信号
   当該製品の電源がオフ状態のときに、入力信号や入出力プルアップ電源を入れないでください。入力信号や入出力プルアップ電源からの電流注入により、誤動作を引き起こしたり、異常電流が流れ内部素子を劣化させたりする場合があります。資料中に「電源オフ時における入力信号」についての記載のある製品は、その内容を守ってください。
4. 未使用端子の処理
   未使用端子は、「未使用端子の処理」に従って処理してください。CMOS 製品の入力端子のインピーダンスは、一般に、ハイインピーダンスとなっています。未使用端子を開放状態で動作させると、誘導現象により、LSI 周辺のノイズが印加され、LSI 内部で貫通電流が流れたり、入力信号と認識されて誤動作を起こす恐れがあります。
5. クロックについて
   リセット時は、クロックが安定した後、リセットを解除してください。プログラム実行中のクロック切り替え時は、切り替え先クロックが安定した後に切り替えてください。リセット時、外部発振子（または外部発振回路）を用いたクロックで動作を開始するシステムでは、クロックが十分安定した後、リセットを解除してください。また、プログラムの途中で外部発振子（または外部発振回路）を用いたクロックに切り替える場合は、切り替え先のクロックが十分安定してから切り替えてください。
6. 入力端子の印加波形
   入力ノイズや反射波による波形歪みは誤動作の原因になりますので注意してください。CMOS 製品の入力がノイズなどに起因して、V_{IL}（Max.）から V_{IH}（Min.）までの領域にとどまるような場合は、誤動作を引き起こす恐れがあります。入力レベルが固定の場合はもちろん、V_{IL}（Max.）から V_{IH}（Min.）までの領域を通過する遷移期間中にチャタリングノイズなどが入らないように使用してください。
7. リザーブアドレス（予約領域）のアクセス禁止
   リザーブアドレス（予約領域）のアクセスを禁止します。アドレス領域には、将来の拡張機能用に割り付けられている リザーブアドレス（予約領域）があります。これらのアドレスをアクセスしたときの動作については、保証できませんので、アクセスしないようにしてください。
8. 製品間の相違について
   型名の異なる製品に変更する場合は、製品型名ごとにシステム評価試験を実施してください。同じグループのマイコンでも型名が違うと、フラッシュメモリ、レイアウトパターンの相違などにより、電気的特性の範囲で、特性値、動作マージン、ノイズ耐量、ノイズ幅射量などが異なる場合があります。型名が違う製品に変更する場合は、個々の製品ごとにシステム評価試験を実施してください。

ご注意書き

1. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。回路、ソフトウェアおよびこれらに関連する情報を使用する場合、お客様の責任において、お客様の機器・システムを設計ください。これらの使用に起因して生じた損害（お客様または第三者いずれに生じた損害も含みます。以下同じです。）に関し、当社は、一切その責任を負いません。
2. 当社製品または本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害またはこれらに関する紛争について、当社は、何らの保証を行うものではなく、また責任を負うものではありません。
3. 当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
4. 当社製品を組み込んだ製品の輸出入、製造、販売、利用、配布その他の行為を行うにあたり、第三者保有の技術の利用に関するライセンスが必要となる場合、当該ライセンス取得の判断および取得はお客様の責任において行ってください。
5. 当社製品を、全部または一部を問わず、改造、改変、複製、リバースエンジニアリング、その他、不適切に使用しないでください。かかる改造、改変、複製、リバースエンジニアリング等により生じた損害に関し、当社は、一切その責任を負いません。
6. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、各品質水準は、以下に示す用途に製品が使用されることを意図しております。
   標準水準：　コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット等
   高品質水準：輸送機器（自動車、電車、船舶等）、交通制御（信号）、大規模通信機器、金融端末基幹システム、各種安全制御装置等
   当社製品は、データシート等により高信頼性、Harsh environment 向け製品と定義しているものを除き、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（宇宙機器と、海底中継器、原子力制御システム、航空機制御システム、プラント基幹システム、軍事機器等）に使用されることを意図しておらず、これらの用途に使用することは想定していません。たとえ、当社が想定していない用途に当社製品を使用したことにより損害が生じても、当社は一切その責任を負いません。
7. あらゆる半導体製品は、外部攻撃からの安全性を 100%保証されているわけではありません。当社ハードウェア／ソフトウェア製品にはセキュリティ対策が組み込まれているものもありますが、これによって、当社は、セキュリティ脆弱性または侵害（当社製品または当社製品が使用されているシステムに対する不正アクセス・不正使用を含みますが、これに限りません。）から生じる責任を負うものではありません。当社は、当社製品または当社製品が使用されたあらゆるシステムが、不正な改変、攻撃、ウイルス、干渉、ハッキング、データの破壊または窃盗その他の不正な侵入行為（「脆弱性問題」といいます。）によって影響を受けないことを保証しません。当社は、脆弱性問題に起因しまたはこれに関連して生じた損害について、一切責任を負いません。また、法令において認められる限りにおいて、本資料および当社ハードウェア／ソフトウェア製品について、商品性および特定目的との合致に関する保証ならびに第三者の権利を侵害しないことの保証を含め、明示または黙示のいかなる保証も行いません。
8. 当社製品をご使用の際は、最新の製品情報（データシート、ユーザーズマニュアル、アプリケーションノート、信頼性ハンドブックに記載の「半導体デバイスの使用上の一般的な注意事項」等）をご確認の上、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他指定条件の範囲内でご使用ください。指定条件の範囲を超えて当社製品をご使用された場合の故障、誤動作の不具合および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めていますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は、データシート等において高信頼性、Harsh environment 向け製品と定義しているものを除き、耐放射線設計を行っておりません。仮に当社製品の故障または誤動作が生じた場合であっても、人身事故、火災事故その他社会的損害等を生じさせないよう、お客様の責任において、冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、お客様の機器・システムとしての出荷保証を行ってください。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
10. 当社製品の環境適合性等の詳細につきましては、製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。かかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器・システムに使用することはできません。当社製品および技術を輸出、販売または移転等する場合は、「外国為替及び外国貿易法」その他日本国および適用される外国の輸出管理関連法規を遵守し、それらの定めるところに従い必要な手続きを行ってください。
12. お客様が当社製品を第三者に転売等される場合には、事前に当該第三者に対して、本ご注意書き記載の諸条件を通知する責任を負うものといたします。
13. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを禁じます。
14. 本資料に記載されている内容または当社製品についてご不明な点がございましたら、当社の営業担当者までお問合せください。

注 1. 本資料において使用されている「当社」とは、ルネサス エレクトロニクス株式会社およびルネサス エレクトロニクス株式会社が直接的、間接的に支配する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

(Rev.5.0-1　2020.10)

本社所在地

〒135-0061 東京都江東区豊洲 3-2-24（豊洲フォレシア）

www.renesas.com

お問合せ窓口

弊社の製品や技術、ドキュメントの最新情報、最寄の営業お問合せ窓口に関する情報などは、弊社ウェブサイトをご覧ください。

www.renesas.com/contact/

商標について

ルネサスおよびルネサスロゴはルネサス エレクトロニクス株式会社の商標です。すべての商標および登録商標は、それぞれの所有者に帰属します。