RX600 シリーズ
CAN アプリケーションプログラミングインタフェース

R01AN0339JU0205
Rev.2.05
2018.02.07

要旨

本アプリケーションノートは、ルネサス CAN アプリケーションプログラミングインタフェースを紹介し、これを使用して CAN バス上でデータを送信、受信および監視する方法について説明します。また、CAN ペリフェラルの一部の機能についても簡単に説明します。

本アプリケーションノートには、CAN API ドライバソースコードファイル config_r_can_rapi.h、r_can_api.h、および r_can_api.c が付属しています。ドライバ関数を作成するには別の方法もあります。たとえば、特定の動作に合うように独自のドライバ関数を作成することができます。

RX CAN ペリフェラルには 32 の CAN メールボックスがあり、CAN と通信するために読み出しおよび書き込みをすることができます。これらのメールボックスはメッセージ「バッファ」であり、CAN データフレームが別の受信フレームによって上書きされるか、アプリケーションによって再び書き込まれるまで、CAN データフレームを保持します。各メールボックスは、送信または受信するように動的に設定することができます。通常、ほとんどのメールボックスを受信に設定し、少数のメールボックスを送信に設定します。

「メールボックス」または一部の文書における「メッセージボックス」または「メッセージバッファ」という用語は、MCU の CAN ペリフェラル内の、メッセージが格納される物理的位置を意味します。本書では、「メールボックス」という用語を使用します。

対象デバイス

このドライバは次のデバイスでサポートされています。

- RX62N、RX62T、RX62G グループ
- RX630、RX631、RX63N グループ

目次

1. 概要 3
1.1 CAN ペリフェラル 3
1.2 CAN 通信レベル 4
1.3 メールボックス 4
1.4 拡張 ID の使用 5

2. CAN API のプロジェクトへの追加 6
2.1 CAN Config ファイル 6
2.1.1 割り込みおよびポーリング 6
2.1.2 CAN 割り込み設定 6
2.1.3 レジスタをポーリングするための最大時間 6
2.1.4 トランシーバ制御端子のマッピング 6
2.1.5 標準/拡張 ID の使用 7
2.1.6 CAN I/O ポート設定 7
2.1.7 CAN ビットレート設定 7

3. CAN API 8
3.1 API リターンコード 9
3.2 CAN バスステータスリターンコード 9
3.3 R_CAN_Create 10
3.4 R_CAN_PortSet 11
3.5 R_CAN_Control 13
3.6 R_CAN_SetBitrate 15
3.7 R_CAN_TxSet および R_CAN_TxSetXid 17
3.8 R_CAN_Tx 19
3.9 R_CAN_TxCheck 20
3.10 R_CAN_TxStopMsg 21
3.11 R_CAN_RxSet および R_CAN_RxSetXid 22
3.12 R_CAN_RxPoll 23
3.13 R_CAN_RxRead 24
3.14 R_CAN_RxSetMask 25
3.15 R_CAN_CheckErr 28

4. CAN ポーリングの使用 30

5. CAN エラー割り込みの使用 33
5.1 CAN 割り込みチェックリスト 33

6. テストモード 34
6.1 内部ループバック-CAN バスを使用しないノードのテスト 34
6.2 外部ループバック - バス上の単独ノードのテスト 34
6.3 リスンオンリー（バス監視） - バスに影響を与えないノードのテスト 35

7. タイムスタンプ 36

8. CAN スリープモード 36

9. CAN FIFO 36

1. 概要

CAN は、安全性とリアルタイム動作が欠かせないアプリケーションに対して、信頼性が高くエラーのないネットワーク通信を提供するために設計されています。主な特長を以下に示します。

- 高い信頼性および雑音余裕度
- 接続箇所の低減
- 柔軟性の高いアーキテクチャ
- ペリフェラルを介したエラー処理
- 低い配線コスト
- 拡張性

MCU およびバスコネクタに必要なのは 2 つの端子のみです。このため、CAN ネットワークは、より多くの配線と接続が必要な他のネットワーキング方式よりも信頼性が高くなります。新しいノードの追加は簡単で、任意の点でバスワイヤをタップするだけです。

CAN は「複数マスタ、複数スレーブ」のトポロジに基づいています。送信されるメッセージまたはデータフレームには、送信ノードまたは目的の受信ノードのアドレスは含まれません。つまり、ノードはいつでもマスタまたはスレーブとして動作することができます。メッセージは、特定の時点でどのノードが特定の ID をリスンしているかによってブロードキャストされるか、ノード間で送信されます。新しいノードは、その他のノードを更新しないで追加することができます。このような柔軟な設計により、インテリジェントで、冗長性があり、簡単に再構成可能なシステムを作成することができます。

ビットレートによって接続可能なノード数とケーブル長が決まります。許容される CAN データビットレートは、62.5、125、250、500 Kbps、および 1 Mbps です。最高速度では、ネットワークは 40m ケーブルで 30 のノードをサポートできます。より低い速度では、ネットワークは 1000m ケーブルで 100 を超えるノードをサポートできます。

CAN ネットワークの基本的な構成要素は、CAN マイクロコントローラ、これを実行するファームウェア、バス信号を駆動し読み出すための CAN トランシーバ、および物理的バスメディア（2 本のワイヤ）です。アプリケーションに適した十分な数のメールボックスを備えた CAN MCU を選択します。

1.1 CAN ペリフェラル

CAN MCU 内の CAN ペリフェラルは、MCU の CAN Tx および Rx 端子を介してチップの外部にあるバストランシーバに接続しなければなりません。プロトコルコントローラは、構成方法よってペリフェラルメールボックスの読み出しおよび書き込みを行います。MCU のハードウェアマニュアルに記載されている CAN 特殊関数レジスタを介して構成が行われます。効果的な通信を実現するためには、CAN ペリフェラルのレジスタを正しい順序で構成して読み出す必要があり、CAN API はこれを大幅に簡略化します。つまり API が多数の難しい問題を処理し、解決します。

ペリフェラルを初期化した後にしなければならないのは、受信および送信 API 呼び出しを使用し、定期的に CAN のエラーステータスをチェックすることだけです。エラー状態が見つかり、アプリケーションがペリフェラルが回復するまで監視して待つ場合、CAN ペリフェラルはその状態に応じてオンラインまたはオフラインになります。回復が検出されると、アプリケーションは再起動します。

1.2 CAN 通信レベル

以下の図は、CAN 通信層を示しており、アプリケーション層が最上位、ハードウェア層が最下位になります。

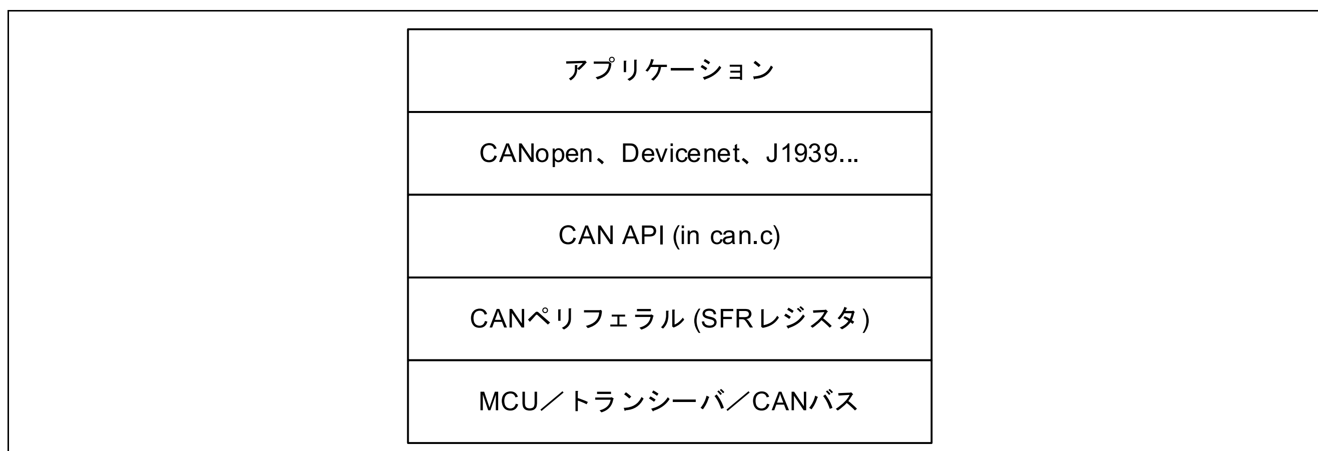

図1.1　CAN 物理層およびソースコード層

この文書では、CANopen や DeviceNet などのより高いレベルのプロトコルについては扱いません。（一部のルネサス CAN MCU では、CANopen ソリューションがあります。販売担当員にお問い合わせください。）

1.3 メールボックス

CAN メッセージを送信するときには、そのメッセージは最初にアプリケーションファームウェアによってメールボックスに書き込まなければなければなりません。その後、他のノードによってより低い ID のメッセージが送信されない限り、バスがアイドル状態になるとすぐにこのメッセージが自動的に送信されます。メールボックスが受信に設定されている場合は、メッセージはプロトコルコントローラによってメールボックスに書き込まれ、ユーザは API を使用して迅速にユーザメモリ領域にメッセージをコピーし、ネットワークから受信する次のメッセージのためにメールボックスを解放する必要があります。

API 呼び出しはメールボックスに対するすべての書き込みおよび呼び出しを実行します。ユーザが行う必要があるのは、API 関数が受信メッセージを書き込み、送信メッセージをコピーすることができるアプリケーションデータフレーム構造体を提供することだけです。送信メッセージに対して少なくとも 1 つの構造体、受信メッセージに対して 1 つの構造体を持つことをお勧めします。送信メッセージの場合、これはローカル変数となることがあります（スタック上）。受信メッセージについては、各メールボックスに 1 つとすることをお勧めします。この CAN データフレーム構造体（タイプ can_std_frame_t）は、API ヘッダファイルによって提供され、以下の構造を備えています。

```
typedef struct
{
    uint32_t    id;
    uint8_t  dlc;
    uint8_t  data[8];
} can_frame_t;
```

タイムスタンプはこの構造体には含まれていませんが、簡単に追加できます。

CAN バスアービトレーションとは別に、優先順位は最小のメールボックス番号により決定されます。ただし SH（RCAN-ET）の場合は、最大のメールボックスに優先順位が与えられます。これは、送信と受信の両方の動作に対して当てはまります。2 つのメールボックスに同じ CAN ID が設定された場合、最小のメールボックス番号が最も高い優先順位となります。このため、2 つのメールボックスが同じ ID で受信するように設定されている場合、その内の 1 つのメールボックスはメッセージを受信しません。

1.4 拡張 ID の使用

拡張 ID を使用するには、以下に説明するように FRAME_ID_MODE を設定する必要があります。

拡張 CAN を有効にすると、末尾が「Xid」の API 関数を呼び出すことができます。この関数は、拡張 ID を使用するように自動的に CAN メールボックスの ID フィールドをフォーマットします。つまり、ユーザがこの Xid 関数を呼び出すだけで、can_frame_t 構造体で渡した ID 値が（11 ビットではなく）29 ビットの ID として送信されるということです。

2. CAN API のプロジェクトへの追加

以下の手順に従って CAN API コードをプロジェクトに追加します。この手順は、フラッシュ API がすでにプロジェクトに追加されていることを前提にしています。

1. **r_can_api** ディレクトリ（このアプリケーションノートに付属）をプロジェクトのディレクトリにコピーします。
2. ファイル **r_can_api.c** をプロジェクトに追加します。
3. **r_can_api** ディレクトリにインクルードパスを追加します。
4. **r_can_api¥src** ディレクトリにインクルードパスを追加します。
5. **config_r_can_rapi.h** でミドルウェアを構成します。

2.1 CAN Config ファイル

実行方法に合わせてアプリケーションをカスタマイズするには、config_r_can_rapi.h ファイルに修正を加える必要がある場合があります。たとえば、CAN ポーリングモードで実行するか、CAN 割り込みモードで実行するかを選択するオプションがあります。これは、config_r_can_rapi.h を使用して選択することができます。

r_can_api.c ファイルは何も変更しないでください。このファイルには、お使いのルネサス MCU 用のルネサス CAN API 関数が含まれています。

2.1.1 割り込みおよびポーリング

(1) USE_CAN_POLL

マクロ USE_CAN_POLL は、送受信されたメッセージに対して CAN メールボックスがポーリングされるか否か、または CAN 割り込みを使用する必要があるか否かを決定します。一部のドライバのバージョンでは、モードを選択するためにこのマクロを組み込むか除外するかを決める必要があります。その他のバージョンでは、マクロを 0 または 1 に設定してください。使用しているバージョンについては、「config_r_can_rapi.h」のコメントを参照してください。

2.1.2 CAN 割り込み設定

(1) CAN0_INT_LVL

CAN 割り込みの優先レベルを設定します。

(2) USE_CAN_API_SEARCH（非推奨）

既存のコード例と混同しないようにするため最新のコードリリースには実装されていません。多くのメールボックスを設定していて、かつ大量の CAN トラフィックがある場合には、この機能の使用を検討してください。MCU のハードウェアマニュアルを参照してください。

2.1.3 レジスタをポーリングするための最大時間

(1) MAX_CANREG_POLLCYCLES

期待値を得るために CAN レジスタビットをポーリングする際の最大ループです。ポーリングモードを使用している場合で、ある一定の時間だけ待機してメールボックスがフレームを受信したことを確認したい場合に、この値を増やしてください。これは非常に小さな値に設定することができますが、ゼロには設定しないでください。ゼロに設定すると、メールボックスがまったくチェックされなくなる可能性があります。

2.1.4 トランシーバ制御端子のマッピング

(1) TRANSCEIVERSTANDBY, TRANSCEIVERENABLE

トランシーバ制御端子を接続する場所を指定します。一部のトランシーバは他の制御端子を持つ場合があります。この場合、これはユーザが設定する必要があります。

2.1.5 標準/拡張 ID の使用

(1) FRAME_ID_MODE

ドライバで有効にする CANID 型のタイプ、すなわち 11 ビット ID を使用するのか 29 ビット ID を使用するのかを選択します。FRAME_ID_MODE は、STD_ID_MODE、EXT_ID_MODE、または MIXED_ID_MODE に設定することができます。最初の 2 つの設定により、その ID モードに属する関数のみが有効になります。混合モードに設定した場合、API 全体が使用可能になります。1 つの ID タイプのみを使用するネットワークで使用する場合は、ドライバに混合モードを設定しないことをお勧めします。ネットワークの ID タイプが間違っているメッセージは、単に帯域幅を浪費するだけになります（いずれのノードも受信しません）。

2.1.6 CAN I/O ポート設定

CAN ペリフェラルに割り当てられた特定の IO ポートは、config_r_can_rapi.h で設定されます。

2.1.7 CAN ビットレート設定

以下の API R_CAN_SetBitrate およびファイル config_r_can_rapi.h を参照してください。

3. CAN API

API は CAN ペリフェラルの設定のすべての詳細を考慮することなく CAN を使用し、アプリケーションがネットワーク上の他のノードと簡単に通信できるようにする関数の集合です。

```
初期化、ポートとペリフェラルの制御
R_CAN_Create (uint32_t ch_nr);
R_CAN_PortSet (uint32_t ch_nr, uint32_t  action_type );
    [action_type = ENABLE, DISABLE]
R_CAN_Control (uint32_t ch_nr, uint32_t  action_type );
    [action_type = ENTERSLEEP_CANMODE, EXITSLEEP_CANMODE, RESET_CANMODE,
                   HALT_CANMODE, OPERATE_CANMODE, CANPORT_TEST_LISTEN_ONLY,
                   CANPORT_TEST_0_EXT_LOOPBACK, CANPORT_TEST_1_INT_LOOPBACK,
                   CANPORT_RETURN_TO_NORMAL]
R_CAN_SetBitrate (uint32_t ch_nr);

送信
R_CAN_TxSet (uint32_t ch_nr, uint32_t mbox_nr, can_std_frame_t* frame_p, uint32_t frame_type);
R_CAN_TxSetXid (uint32_t ch_nr, uint32_t mbox_nr, can_frame_t* frame_p, uint32_t frame_type);
R_CAN_Tx (uint32_t ch_nr, uint32_t mbox_nr);
R_CAN_TxCheck (uint32_t ch_nr, uint32_t mbox_nr);
R_CAN_TxStopMsg (uint32_t ch_nr, uint32_t mbox_nr);

受信
R_CAN_RxSet (uint32_t ch_nr, uint32_t mbox_nr, uint32_t stid,   uint32_t frame_type);
R_CAN_RxSetXid (uint32_t ch_nr, uint32_t mbox_nr, uint32_t xid, uint32_t frame_type);
R_CAN_RxRead (uint32_t ch_nr, uint32_t mbox_nr,  can_std_frame_t* frame_p);
R_CAN_RxPoll (uint32_t ch_nr, uint32_t mbox_nr);
R_CAN_RxSetMask (uint32_t ch_nr, uint32_t sid_mask_value,  uint32_t mask_reg_nr);

エラーチェック
R_CAN_CheckErr (uint32_t ch_nr);
```

図3.1　CAN API 関数

ハードウェアマニュアルが動作に関する絶対的な基準であることに注意してください。

図 3.1の最初のグループの関数（橙色）は、CAN ペリフェラルレジスタを初期化し、MCU CAN およびトランシーバポートを設定するために使用します。最初の関数 R_CAN_Create は、デフォルトでその他の初期化関数を呼び出します。

送信関数（青色）は、送信用メールボックスをセットアップし、正しく送信されたことを確認するために使用します。

受信関数（緑色）は、受信用メールボックスをセットアップし、そのメールボックスからメッセージを取得するために使用します。

エラー関数（赤色）は、ノードの CAN バスステータスをチェックします。

3.1 API リターンコード

R_CAN_OK	アクションが正常に終了しました。
R_CAN_NOT_OK	アクションが正常に終了しませんでした。 通常、より具体的なリターンコードが使われます。下記を参照してください。
R_CAN_SW_BAD_MBX	メールボックス番号が正しくありません。
R_CAN_BAD_CH_NR	チャネル番号が存在しません。
R_CAN_BAD_ACTION_TYPE	この関数にはそのようなアクションタイプはありません。
R_CAN_MSG_LOST	メッセージは上書きされたか、失われました。
R_CAN_NO_SENTDATA	メッセージは送信されませんでした。
R_CAN_RXPOLL_TMO	受信メッセージのポーリングはタイムアウトしました。
R_CAN_SW_WAKEUP_ERR	CAN ペリフェラルはスリープモードからウェイクアップしませんでした。
R_CAN_SW_SLEEP_ERR	CAN ペリフェラルはスリープモードになりませんでした。
R_CAN_SW_HALT_ERR	CAN ペリフェラルはホルトモードになりませんでした。
R_CAN_SW_RST_ERR	CAN ペリフェラルはリセットモードになりませんでした。
R_CAN_SW_TSRC_ERR	タイムスタンプエラー
R_CAN_SW_SET_TX_TMO	前の送信の完了待ちがタイムアウトしました。
R_CAN_SW_SET_RX_TMO	前の受信の完了待ちがタイムアウトしました。
R_CAN_SW_ABORT_ERR	アボート待ちがタイムアウトしました。
R_CAN_MODULE_STOP_ERR	CAN ペリフェラル全体が停止状態（低電力状態）です。おそらく、モジュール停止レジスタをアンロックするための PRCR レジスタが使用されませんでした。

3.2 CAN バスステータスリターンコード

R_CAN_STATUS_ERROR_ACTIVE	バスステータス：正常動作
R_CAN_STATUS_ERROR_PASSIVE	バスステータス：ノードは送信エラーカウンタまたは受信エラーカウンタに対して少なくとも 127 のエラーフレームを送信しました。
R_CAN_STATUS_BUSOFF	バスステータス：ノードの送信エラーカウンタはノードが正しく送信できなかったために 255 を超えました。

3.3 R_CAN_Create

CAN ペリフェラルを初期化し、ビットレートを設定し、CAN 割り込みを設定します。

この関数は、CAN ペリフェラルレジスタを使用する準備のため初期化を実行します。また、ビットレート、デフォルトのメールボックス設定、CAN 割り込みレベルも設定します。

フォーマット

```
uint32_t R_CAN_Create(const uint32_t ch_nr);
```

引数

ch_nr	使用する CAN バス。使用可能なチャネル数は製品によって異なります。

戻り値

R_CAN_OK	アクションが正常に完了しました。
R_CAN_SW_BAD_MBX	メールボックス番号が正しくありません。
R_CAN_BAD_CH_NR	チャネル番号が存在しません。
R_CAN_SW_RST_ERR	CAN ペリフェラルはリセットモードになりませんでした。
R_CAN_MODULE_STOP_ERR	CAN ペリフェラル全体が停止状態（低電力状態）です。おそらく、モジュール停止レジスタをアンロックするための PRCR レジスタが使用されませんでした。
R_CAN_SW_TSRC_ERR	タイムスタンプカウンタのカウントが 0 に達しました。

R_CAN_Control の戻り値も参照してください。

プロパティ

r_can_api.h にプロトタイプ化されます。

r_can_api.c に実装されます。

コメント

この関数はペリフェラルを CAN スリープモードから復帰させます。この関数は以下のデフォルト設定でメールボックスを構成します。

- 新しいフレームが到着したときに未読のメールボックスを上書きする
- ID 優先順位を使用するようにデバイスを設定する（通常の CAN 動作。オプションのメールボックス番号の優先順位ではない）
- すべてのメールボックスのマスクを無効に設定する

R_CAN_Create は、config_r_can_rapi.h で USE_CAN_POLL が無効化されている場合に R_CAN_SetBitrate 関数を呼び出して CAN 割り込みを設定します。

終了する前にすべてのメールボックスをクリアし、ペリフェラルを動作モードに設定して、エラーをクリアします。

例

```
/* Init CAN. */
api_status = R_CAN_Create(0);
```

3.4 R_CAN_PortSet

MCU およびトランシーバポート端子を設定します。

この関数は MCU およびトランシーバポート端子を設定します。Enable などのトランシーバポート端子は設計によって異なり、それに応じてこの関数を修正する必要があります。

この関数は、リスンオンリーなどの CAN ポートテストモードに移行する場合にも使用します。

フォーマット

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

引数

ch_nr	0, 1,2,3	使用する CAN バス。使用可能なチャネル数は製品によって異なります。1～4 チャネルが使用可能です。
action_type	ポートアクション:	
	ENABLE	CAN ポート端子と CAN トランシーバを有効にします。
	DISABLE	CAN ポート端子と CAN トランシーバを無効にします。
	CANPORT_TEST_LISTEN_ONLY	リスンオンリーモードに設定します。Ack またはエラーフレームは送信されません。6.3を参照してください。
	CANPORT_TEST_0_EXT_LOOPBACK	外部バスとループバックを使用します。*テストされていません。*
	CANPORT_TEST_1_INT_LOOPBACK	内部メールボックス通信のみ
	CANPORT_RETURN_TO_NORMAL	通常ポート使用に戻ります。

戻り値

R_CAN_OK	アクションが正常に完了しました。
R_CAN_SW_BAD_MBX	メールボックス番号が正しくありません。
R_CAN_BAD_CH_NR	チャネル番号が存在しません。
R_CAN_BAD_ACTION_TYPE	そのアクションタイプはこの関数には存在しません。
R_CAN_SW_HALT_ERR	CAN ペリフェラルはホルトモードになりませんでした。
R_CAN_SW_RST_ERR	CAN ペリフェラルはリセットモードになりませんでした。

R_CAN_Control の戻り値も参照してください。

プロパティ

r_can_api.h にプロトタイプ化されます。

r_can_api.c に実装されます。

コメント

デフォルトのポートセットアップ関数（たとえば「hwsetup」）を使用する前と後に必ずこの関数を呼び出してください。そうでないと、MCUCAN ポート端子の出力 high/low がバスに影響する場合があります。（デバッグ時に、ノードのハードリセットにより他のノードがエラーモードに移行する場合があります。その理由は、CAN がポートを再設定する前にすべてのポートがデフォルトの出力 high/low として設定されるためです。しばらくの間、ポートの出力が low になり、CAN バス電圧レベルが乱れることになります。）

トランシーバによってはトランシーバポート端子を変更または追加しなければならない場合があります。

例

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

3.5 R_CAN_Control

CAN 動作モードを設定します。

CAN 制御レジスタによって決まる CAN 動作モードへの遷移を制御します。たとえば、休止モードは後で受信メールボックスを設定する場合に使用します。

フォーマット

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

引数

ch_nr	0, 1,2,3	使用する CAN バス。使用可能なチャネル数は製品によって異なります。1～4 チャネルが使用可能です。
action_type	ペリフェラルアクション:	
	EXITSLEEP_CANMODE	CAN スリープモードを終了します。これはペリフェラルが起動するときのデフォルト状態です。8を参照してください。
	ENTERSLEEP_CANMODE	CAN スリープモードに入り、消費電力を低減します。
	RESET_CANMODE	CAN ペリフェラルをリセットモードにします。
	HALT_CANMODE	CAN ペリフェラルをリセットモードにします。CAN ペリフェラルはまだ接続されていますが、通信を停止します。
	OPERATE_CANMODE	CAN ペリフェラルを通常動作モードにします。

戻り値

R_CAN_OK	アクションが正常に完了しました。
R_CAN_SW_BAD_MBX	メールボックス番号が正しくありません。
R_CAN_BAD_CH_NR	チャネル番号が存在しません。
R_CAN_BAD_ACTION_TYPE	そのアクションタイプはこの関数には存在しません。
R_CAN_SW_WAKEUP_ERR	CAN ペリフェラルはスリープモードからウェイクアップしませんでした。
R_CAN_SW_SLEEP_ERR	CAN ペリフェラルはスリープモードになりませんでした。
R_CAN_SW_HALT_ERR	CAN ペリフェラルはホルトモードになりませんでした。
R_CAN_SW_RST_ERR	CAN ペリフェラルはリセットモードになりませんでした。

R_CAN_PortSet の戻り値も参照してください。

プロパティ

r_can_api.h にプロトタイプ化されます。

r_can_api.c に実装されます。

コメント

この API を呼び出してホルトモードに移行する以外に、他の API 関数を経由して CAN モード遷移が自動的に呼び出されます。たとえば、起動時のデフォルトのモードは CAN スリープモードです。API を使用して他の動作モードに切り替えます。たとえば、最初に「スリープを終了」し、次に「リセット」して、ビットレートおよび割り込みについて CAN レジスタを初期化し、「ホルト」モードに移行し、メールボックスを設定します。

例

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

3.6 R_CAN_SetBitrate

CAN ビットレート（通信速度）を設定します。

ボーレートおよびビットタイミングは設定プロセス時に必ず設定する必要があります。これは、リセットモードに移行すれば、後で変更することができます。

フォーマット

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

引数

ch_nr	0, 1,2,3	使用する CAN バス。1～4 チャネルが使用可能です。

戻り値

-

プロパティ

r_can_api.h にプロトタイプ化されます。

r_can_api.c に実装されます。

コメント

時間量 Tq は CAN システムクロック fcanclk の 1 ビット時間です。これは CAN ビット時間ではなく、CAN ペリフェラルの内部クロック期間です。この CAN システムクロックは、CAN システムクロックを作成するために、ボーレートプリスケーラ値（x2）およびペリフェラルバスクロックによって決定されます。

CAN バス上のボーレートやデータ速度を設定するには、CAN ビットタイミングと MCU 周波数を理解し、ハードウェアマニュアルの図と表を参照する必要があります。API のデフォルトのビットレート設定は 500kB で、MCU クロックまたはペリフェラルの周波数が変更されない限り、関数を呼び出すだけで済みます。

1 ビット時間はいくつかの時間クアンタ Tqtot に分割されます。1 時間クアンタは CAN クロックの期間と等しくなります。次に各ビットレートレジスタに特定の数の Tq、すなわち CAN ビットの 1 期間を構成する Tq の合計が指定されます。

ビットレートレジスタ設定値の計算式

PCLK はペリフェラルクロック周波数です。

fcan = PCLK

プリスケーラは係数を使用して CAN ペリフェラルクロックをスケールダウンします。

fcanclk = fcan/prescaler

1 時間クアンタは CAN クロックの 1 クロック期間です。

Tq = 1/fcanclk

Tqtot は CAN の 1 ビット時間の CAN ペリフェラルクロックサイクルの合計数であり、「時間セグメント」と常に 1 である「SS」の合計によって作成されるペリフェラルによるものです。

Tqtot = TSEG1 + TSEG2 + SS　　（TSEG1 > TSEG2 でなければならない）

したがってビットレートは次のようになります。

Bitrate = fcanclk/Tqtot

SS は常に 1 です。SJW はしばしばバス管理者によって指定されます。1 <= SJW <= 4 を選択します。

詳細については、CONFIG_R_CAN_RAPI.H を参照してください。

例

```
/* Set bitrate as defined in config_r_can_rapi.h. */
R_CAN_SetBitrate(0);
```

3.7 R_CAN_TxSet および R_CAN_TxSetXid

送信するメールボックスをセットアップします。

R_CAN_TxSet は、指定した ID、データ長、およびデータフレームペイロードをメールボックスに書き込んだ後、メールボックスを送信モードに設定し、R_CAN_Tx()を呼び出すことでバスにフレームを送信します。

R_CAN_TxSetXid も同じ機能ですが、この関数を使用した場合、ID は 29 ビット ID になります。

フォーマット

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

引数

ch_nr	0,1,2,3	使用する CAN バス。1～4 チャネルが使用可能です。
mbox_nr	0-32	使用するメールボックス
frame_p*		メモリ内のデータフレーム構造体を指すポインタです。これはメールボックスが送信するデータフレームを構成する ID、DLC、およびデータを含むデータ構造体へのアドレスです。
frame_type	DATA_FRAME	通常のデータフレームを送信します。
	REMOTE_FRAME	リモートデータフレーム要求を送信します。

戻り値

R_CAN_OK	送信用にメールボックスがセットアップされました。
R_CAN_SW_BAD_MBX	メールボックス番号が正しくありません。
R_CAN_BAD_CH_NR	チャネル番号が存在しません。
R_CAN_BAD_ACTION_TYPE	この関数にはそのようなアクションタイプはありません。

プロパティ

r_can_api.h にプロトタイプ化されます。

r_can_api.c に実装されます。

コメント

この関数は最初に指定したメールボックスの前の送信が完了するまで待ちます。その後、メールボックスをセットアップするときにメールボックスの割り込みを一時的に無効にします。すなわち、メールボックスの ID 値、frame_p で指定されたデータ長コードを設定し、データフレームまたはリモートフレーム要求を選択し、最後にデータフレームペイロードバイト（0～7）をメールボックスにコピーします。メールボックスは、USE_CAN_POLL が定義されていない限り、再び割り込みが有効になります。最後に R_CAN_Tx が呼び出され、メッセージが送信されます。

例

```
#define MY_TX_SLOT   7
can_std_frame_t      my_tx_dataframe;

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

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

3.8 R_CAN_Tx

CAN バスへの実際のメッセージ送信を開始します。

この API は、メールボックスが前のフレームの処理を終了するまで待ち、その後、メールボックスを送信モードに設定します。

フォーマット

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

引数

ch_nr	0,1,2,3	使用する CAN バス。1～4 チャネルが使用可能です。
mbox_nr	0-32	使用する CAN メールボックス

戻り値

R_CAN_OK	メールボックスは前に設定したメールボックスを送信するように設定されました。
R_CAN_SW_BAD_MBX	メールボックス番号が正しくありません。
R_CAN_BAD_CH_NR	チャネル番号が存在しません。
R_CAN_SW_SET_TX_TMO	前回の送信の終了待ちがタイムアウトしました。
R_CAN_SW_SET_RX_TMO	前回の受信の終了待ちがタイムアウトしました。

プロパティ

r_can_api.h にプロトタイプ化されます。

r_can_api.c に実装されます。

コメント

システムがメールボックスの内容のセットアップを開始した後、このメールボックスについて少なくとも 1 回は R_CAN_TxSet を呼び出しておく必要があります。この関数はメールボックスにその内容を送信するように指示するだけであるからです。

例

```
#define MY_TX_SLOT    7

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

3.9 R_CAN_TxCheck

データフレーム送信が成功したことを確認します。

メールボックスをチェックしてデータフレーム送信が成功したことを確認します。

フォーマット

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

引数

ch_nr	0,1,2,3	使用する CAN バス。1～4 チャネルが使用可能です。
mbox_nr	0-32	使用する CAN メールボックス

戻り値

R_CAN_OK	送信が正常に完了しました。
R_CAN_SW_BAD_MBX	メールボックス番号が正しくありません。
R_CAN_BAD_CH_NR	チャネル番号が存在しません。
R_CAN_MSGLOST	メッセージは上書きされたか、失われました。
R_CAN_NO_SENTDATA	メッセージは送信されませんでした。

プロパティ

r_can_api.h にプロトタイプ化されます。

r_can_api.c に実装されます。

コメント

この関数は、たとえばステートマシンを進行できるようにメッセージが送信されたことをアプリケーションが確認する必要がある場合、またはメッセージが戻された場合にのみ必要です。シリコンチップに組み込まれた CAN トランスポート制御レベルを使用して、メールボックスが API により要求されると、メッセージが送信されるとみなすことができます。当然ですが、この関数は送信後に使用するのが最も安全です。

例

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

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

3.10 R_CAN_TxStopMsg

フレーム送信を要求されたメールボックスを停止します。

フォーマット

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

引数

ch_nr	0,1,2,3	使用する CAN バス。1～4 チャネルが使用可能です。
mbox_nr	0-32	使用する CAN メールボックス

戻り値

R_CAN_OK	アクションが正常に完了しました。
R_CAN_SW_BAD_MBX	メールボックス番号が正しくありません。
R_CAN_BAD_CH_NR	チャネル番号が存在しません。
R_CAN_SW_ABORT_ERR	アボート待ちがタイムアウトしました。

プロパティ

r_can_api.h にプロトタイプ化されます。

r_can_api.c に実装されます。

コメント

この関数は、送信が停止するようにメールボックス制御フラグをクリアします（TrmReq が 0 に設定されます）。次にソフトウェアカウンタが最大時間アボートを待ちます。

メッセージが停止されなかった場合、R_CAN_SW_ABORT_ERR が返されます。この原因として、メッセージがすでに送信されたことが考えられます。

例

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

3.11 R_CAN_RxSet および R_CAN_RxSetXid

受信するメールボックスをセットアップします。

R_CAN_RxSet API は、指定された CAN ID を持つデータフレームを受信するように、指定されたメールボックスをセットアップします。同じ ID を持つ受信データフレームはメールボックスに格納されます。

R_CAN_RxSetXid も同じ機能ですが、この関数を使用した場合、ID は 29 ビット ID になります。

フォーマット

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

引数

ch_nr	0,1,2,3	使用する CAN バス。1～4 チャネルが使用可能です。
mbox_nr	0-32	使用する CAN メールボックス
sid	$0\text{-}7FF_h$	メールボックスが受信する標準 CANID
frame_type	DATA_FRAME	通常のデータフレームを送信します。
	REMOTE_FRAME	リモートデータフレーム要求を送信します。

戻り値

R_CAN_OK	アクションが正常に完了しました。
R_CAN_SW_BAD_MBX	メールボックス番号が正しくありません。
R_CAN_BAD_CH_NR	チャネル番号が存在しません。
R_CAN_SW_SET_TX_TMO	前の送信の終了待ちがタイムアウトしました。
R_CAN_SW_SET_RX_TMO	前の受信の終了待ちがタイムアウトしました。

プロパティ

r_can_api.h にプロトタイプ化されます。

r_can_api.c に実装されます。

コメント

関数は最初に、前の送信あるいは受信が完了するのを待ち、次にメールボックスの割り込みを一時的に無効にします。メールボックスを指定された標準 ID 値に設定し、通常の CAN データフレームを受信するのか、リモートフレーム要求を受信するのかを設定します。

例

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

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

3.12 R_CAN_RxPoll

メールボックスがメッセージを受信したかどうかをチェックします。

フォーマット

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

引数

ch_nr	0,1,2,3	使用する CAN バス。1～4 チャネルが使用可能です。
mbox_nr	0-32	チェックする CAN メールボックス

戻り値

R_CAN_OK	待ち状態のメッセージがあります。
R_CAN_NOT_OK	待ち状態か、保留中のメッセージはありません。
R_CAN_RXPOLL_TMO	保留中のメッセージがタイムアウトしました。
R_CAN_SW_BAD_MBX	メールボックス番号が正しくありません。
R_CAN_BAD_CH_NR	チャネル番号が存在しません。

プロパティ

r_can_api.h にプロトタイプ化されます。

r_can_api.c に実装されます。

コメント

特定のメッセージを受信するようにメールボックスがセットアップされた場合は、受信が正常に完了したときを判断することが重要です。これを行うには、次の 2 つの方法があります。

1. ポーリング。API を定期的に呼び出して新しいメッセージの有無をチェックします。CAN 設定ファイルの中で USE_CAN_POLL を定義する必要があります。メッセージがあれば、R_CAN_RxRead を使用してメッセージを取り出します。
2. CAN 受信割り込みの使用（USE_CAN_POLL は定義せず）:この API を使用してメールボックスの受信をチェックします。次にアプリケーションに通知します。

メールボックスで新しいデータが見つかった場合、関数は R_CAN_OK を返します。

例

R_CAN_RxRead の例を参照してください。

3.13 R_CAN_RxRead

メールボックスから CAN データフレームの内容を読み出します。

API は、指定されたメールボックスがメッセージを受信したかどうかをチェックします。受信した場合は、メールボックスのデータフレームのコピーが指定された構造体に書き込まれます。

フォーマット

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

引数

ch_nr	0,1,2,3	使用する CAN バス。1～4 チャネルが使用可能です。
mbox_nr	0-32	チェックする CAN メールボックス
frame_p	*	メモリ内のデータフレーム構造体を指すポインタを参照します。 これは関数がメールボックスの受信 CAN データフレームのコピーを格納するデータ構造体へのアドレスです。

戻り値

R_CAN_OK	待ち状態のメッセージがあります。
R_CAN_SW_BAD_MBX	メールボックス番号が正しくありません。
R_CAN_BAD_CH_NR	チャネル番号が存在しません。
R_CAN_MSGLOST	メッセージは上書きされたか、失われました。

プロパティ

r_can_api.h にプロトタイプ化されます。

r_can_api.c に実装されます。

コメント

最初に R_CAN_PollRxCAN()を使用してメールボックスがメッセージを受信したかどうかをチェックします。

この関数を使用すると、ポーリングモードの使用時または CAN 受信割り込みにより、メールボックスからメッセージを取り出すことができます。

例

```
#define MY_RX_SLOT    8
can_std_frame_t       my_rx_dataframe;

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

3.14 R_CAN_RxSetMask

CAN ID 受け入れマスクを設定します。

1 つの ID のみを受け入れるには、マスクをすべて 1 に設定します。すべてのメッセージを受け入れるには、マスクをすべて 0 に設定します。ある範囲のメッセージを受け入れるには、対応する ID ビットを 0 に設定します。

フォーマット

```
void R_CAN_RxSetMask(     const uint32_t    ch_nr,
                          const uint32_t    mbox_nr,
                          const uint32_t    sid_mask_value    );
```

引数

ch_nr	0,1,2,3	使用する CAN バス。1～4 チャネルが使用可能です。
mbox_nr	0-32	チェックする CAN メールボックス
sid_mask_value	0-7FFh	マスク値

戻り値

-

プロパティ

r_can_api.h にプロトタイプ化されます。

r_can_api.c に実装されます。

コメント

受信メールボックスは、マスクを使用して 1 つまたはある範囲のメッセージ CAN ID をフィルタすることができます。マスクにより、メールボックスは、メールボックスの ID フィールドに設定されたただ 1 つのメッセージ ID よりも広い範囲のメッセージを受け入れることができます。

メールボックス 0～3 について 1 つのマスク、4～7 について 1 つのマスクがあります。このため、マスクを変更すると、隣接するメールボックスの動作に大きな影響を与える点に注意してください。

- マスクの「0」は「このビットをマスクする」または「そのビットを見ない」を意味し、すべてを受け入れます。
- 「1」はこの位置の CAN-ID ビットがメールボックスの CAN-ID に一致するかどうかをチェックすることを意味します。

マスクの設定方法

メールボックスに受信する CAN-ID が、標準の 11 ビット ID を使用して 700～704_h であると仮定します。

16 進表示	ビット表示
0x700	11100000000b
0x701	11100000001b
0x702	11100000010b
0x703	11100000011b
0x704	11100000100b

メールボックスは、マスク値が 1 である位置と一致する ID を持つフレームのみを受け入れます。したがって上記のすべてを受け入れる場合は、マスクを次のように設定します。

11111111000b または $07F8_h$

CAN 受信フィルタはビット位置 b11（MSB）～b3（LSB）を調べて、これらのビットがメールボックスの受信 ID に一致するかどうかを確認します。

次に、ID0x700 を受信するようメールボックスを設定した場合（0x700～0x707 は同じ結果をもたらします）、ID0x700～0x707 を受け入れます。0x705～0x707 は、後でアプリケーションソフトウェアによって、手動で無視する必要があります。

受け入れフィルタサポートによるメッセージの高速フィルタリング

広範囲のメッセージ ID を受信するためにマスクを使用する場合、ファームウェアを使用して実際に必要なメッセージをフィルタリングする必要があります。この検索速度を高めるため、受け入れフィルタサポートユニットを代わりに使用することができます。

受け入れフィルタサポートユニット（ASU）を使うと、（R_CAN_RxSetMask API で）マスクを使用するときに受信したメッセージのソフトウェアフィルタリングと比べて、高速に検索を行うことができます。ソフトウェアフィルタリングでは、標準 ID ビットは再配置され、メモリ内の通常ワードとして格納されないので、時間がかかることがあります。他の問題は、受け入れマスクが、特定の組み合わせの希望するメッセージを受信するように設定できない場合があることです。すべてのメッセージを受け入れるようにマスクを設定した場合、各受信 ID ごとにソフトウェアを使用してメッセージの長いリストをチェックするので、時間を「無駄に」しなければならない場合があります。この手動フィルタリングでは、すべての ID を読み取り可能なフォーマットにすることも必要になります。この場合の効率的な解決策が、受け入れフィルタサポートユニットを使用することです。

これを使用するには、メッセージボックスに格納されるときに CAN-ID を ASU に書き込みます。ASU レジスタから値を読み戻すと次のようになります。

ビット 0～7=テーブルアドレス検索情報「ASI」

ビット 8～15=ビット検索情報「BSI」。SID0～3 はビット位置に変換され、高速テーブル検索が有効になります。出力を使用してテーブルを検索します。

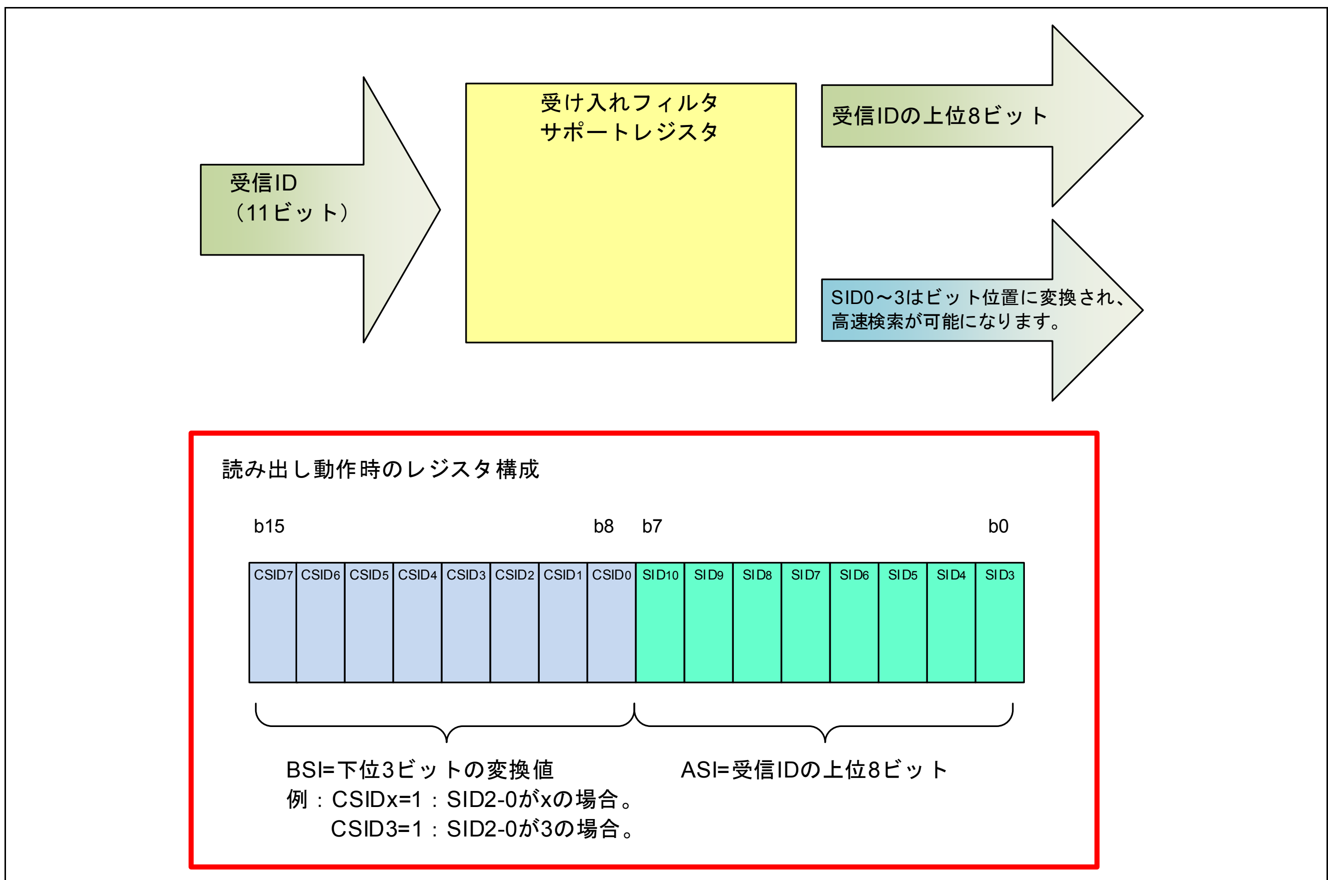

図3.2　受け入れフィルタサポートユニット（ASU）

読み取り時に、テーブルを高速に検索できるように ID の表記がフォーマットされます。これにより、「通常の」一連の CAN ID の検索よりも応答が速くなります。

検索テーブル。テーブルは、ID がアプリケーションに関連するかどうかをチェックするためにユーザが作成する必要があります。ファームウェアは各バイトアドレス ASI およびビット位置 BSI でテーブルを検索する必要があります。ビット BSI 値がユーザのテーブルに設定された場合、ビットパターンは、アドレスがノードと関連することを意味するレジスタの BSI パターンと一致し、フレームはアプリケーションによって処理されます。

ASU の使用方法の詳細については、REJ05B0276「CAN アプリケーションノート」を参照してください。（www.renesas.com からダウンロードしてください。）

3.15 R_CAN_CheckErr

バスエラーをチェックします。

API は、CAN ペリフェラルの CAN ステータスまたはエラー状態をチェックします。

フォーマット

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

パラメータ

ch_nr	0,1,2,3	使用する CAN バス。1〜4 チャネルが使用可能です。

戻り値

CAN_STATE_ERROR_ACTIVE	CAN バスステータス：正常動作
CAN_STATE_ERROR_PASSIVE	CAN バスステータス：ノードは送信エラーカウンタまたは受信エラーカウンタに対して少なくとも 127 のエラーフレームを送信しました。
CAN_STATE_BUSOFF	CAN バスステータス：ノードが正しく送信できなかったため、ノードの送信エラーカウンタが 255 を超えました。

プロパティ

r_can_api.h にプロトタイプ化されます。

r_can_api.c に実装されます。

コメント

API は CAN ペリフェラルの CAN ステータスフラグをチェックし、ステータスエラーコードを返します。これはノードが動作状態であるかどうかを通知し、アプリケーションエラー処理に使用されます。

メインループから定期的にポーリングするか、CAN エラー割り込みを経由してポーリングしなければなりません。ペリフェラルは再送信とエラーフレームを自動的に処理するので、エラー割り込みルーチンを組み込む利点はありません。

エラー状態が見つかり、アプリケーションがペリフェラルの回復を待って監視する場合、CAN ペリフェラルはその状態に応じてオンラインまたはオフラインになります。回復が検出されると、アプリケーションは再起動します。

バス状態

CAN は、CAN ネットワークのノードに障害が発生した場合にネットワーク通信を保護するように設計されています。トランスミッタがエラーフラグを検出するたびに、送信エラーカウンタが増加し、受信フレームのエラーが検出されると、受信エラーカウンタが増加します。送信エラーカウンタと受信エラーカウンタは、フレームが正しく送信または受信されるたびにそれぞれ減少します。エラーアクティブ状態（通常の動作状態）とエラーパッシブ状態の両方で、メッセージを送信および受信することができます。

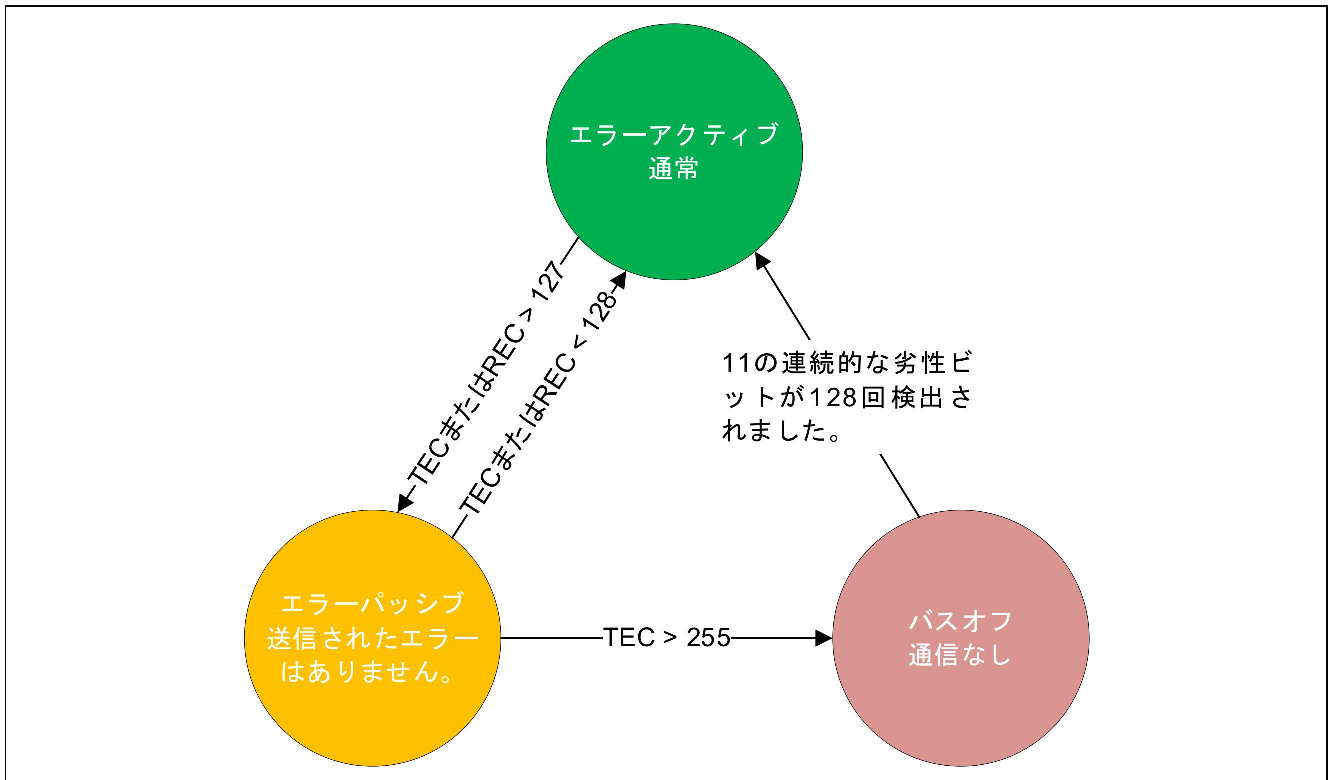

図3.3　CAN エラー状態

- エラーアクティブ

　ノードがエラーアクティブ状態の場合、バスと正常に通信します。ユニットがエラーを検出すると、アクティブエラーフラグを送信します。127 のエラーがカウントされると、エラーパッシブ状態に切り替わります。

- エラーパッシブ

　エラーカウンタのいずれかが 128 を超えると、そのノードの CAN ステータスはエラーパッシブ状態に変わります。メッセージは引き続き送信および受信できますが、ノードはエラーフレームを送信しません。エラーフレームはユーザには見えず、ペリフェラルシリコンチップによって処理されます。

- バスオフ

　送信エラーカウンタが 255 を超えた場合、CAN ノードはバスオフ状態に移行します。これにより、障害のあるノードによってバス障害が生じることが防止されます。深刻な問題によって CAN ノードがバスオフ状態に移行した場合、11 の連続する「リセッシブ」ビットが 128 回検出されるか、ペリフェラルがリセットされるまで、そのノードはメッセージを送信または受信できません。アプリケーションがバスオフ状態から回復したことを検出した場合、ユーザは CAN モジュールのすべてのレジスタを再初期化し、アプリケーションを再起動する必要があります。

4. CAN ポーリングの使用

API を定期的に呼び出してアプリケーションの CAN 状態をチェックし、ノードがバスオフ状態であれば、通信を試みません。以下では、メインアプリケーションのループごとに HandleCanBusState が 1 回呼び出されることを前提としています。

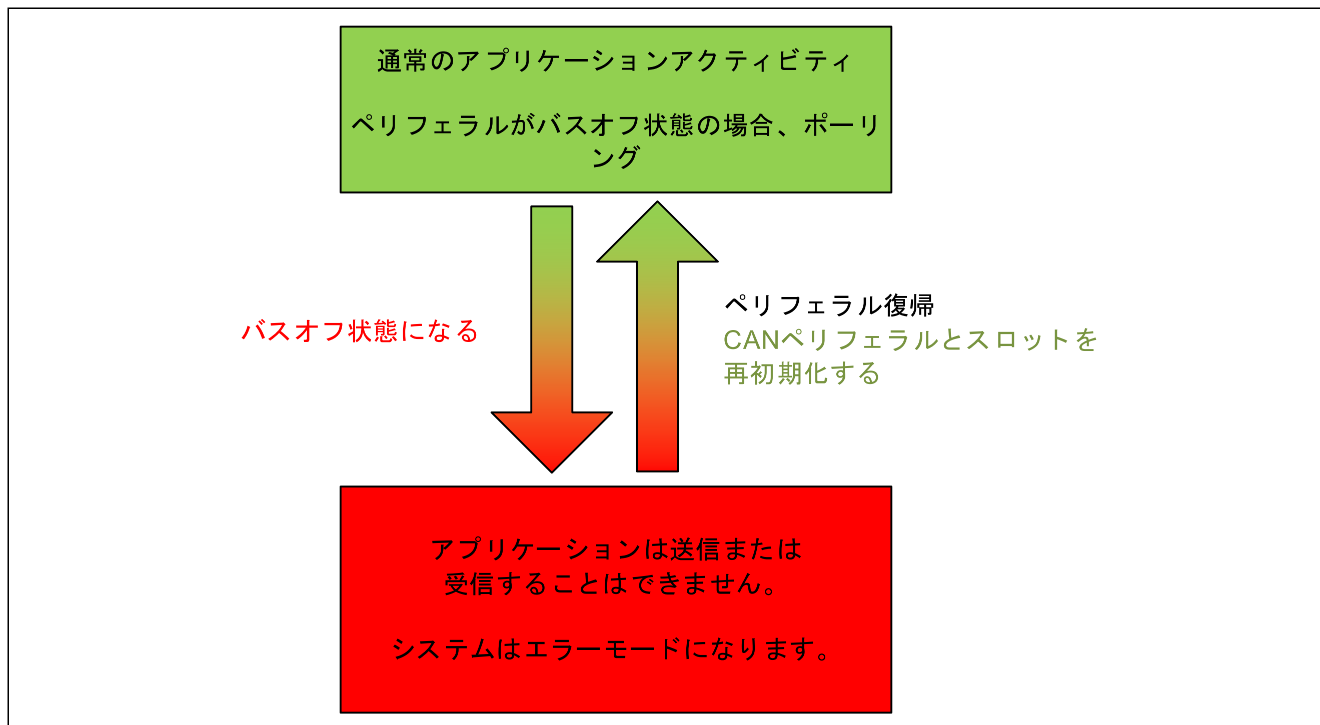

図4.1　アプリケーションのバスオフ状態からの回復処理（MCU はバスの回復を検出します）

ノードは、バス上に 11 の連続したリセッシブビットが 128 回検出されると、自動的に再び通常のエラーアクティブ状態に移行します。ノードがバスオフ状態である時間は非常に短い場合があることに注意してください（たとえば、1 ms 秒未満）。

ノードの状態に関係なくメインルーチンのサイクルごとに 1 回、チェックエラー関数を用いてポーリングします（または CAN エラー割り込みを使用します）。ノードが特定の期間内に特定の回数だけバスオフ状態に移行した場合、警告メッセージを送信したり、LED を点灯したりできます。

バスオフ状態に移行した場合にノードに必要な最低限のアクションを上に示しています。通信の試行を停止し、チェックエラー関数を使用してペリフェラルをポーリングしてペリフェラルが通常のエラーアクティブ状態に戻ったかどうかを確認します。ノードが回復したら、CAN ペリフェラルとアプリケーションを再初期化し、スロットが既知の状態であることを確認することが重要です。

例

```
uint8_t error_bus_status;
/******************************************************************************
Name:              HandleCanBusState
Parameters:        Bus number, 0 or 1.
Returns:           -
Description:       Check CAN peripheral bus state.
******************************************************************************/
static void HandleCanBusState(uint8_t  ch_nr)
{
    can_std_frame_t err_tx_dataframe;

    /* Has the status register reached error passive or more? */
    if (ch_nr == 0){
        error_bus_status[ch_nr] = R_CAN_CheckErr(0);
    }
    else {
        error_bus_status[ch_nr] = R_CAN1_CheckErr(1);
    }

    /* Tell user if CAN bus status changed. All Status bits are read only. */
    if (error_bus_status[ch_nr] != error_bus_status_prev[ch_nr]){
        switch (error_bus_status[ch_nr]){
            /* Error Active. */
            case R_CAN_STATUS_ERROR_ACTIVE:
                /* Only report if there was a previous error. */
                if (error_bus_status_prev[ch_nr] > R_CAN_STATUS_ERROR_ACTIVE){
                    if (ch_nr == 0){
                        DisplayString(LCD_LINE1, "bus0: OK");
                    }
                    else {
                        DisplayString(LCD_LINE2, "bus1: OK");// Show user
                    }
                    Delay(0x400000);
                }

                /* Restart if returned from Bus Off. */
                if (error_bus_status_prev[ch_nr] == R_CAN_STATUS_BUSOFF){
                    /* Restart CAN */
                    if (R_CAN_Create(0) != R_CAN_OK) {
                        app_err_nr |= APP_ERR_CAN_PERIPH;
                    }
                    /* Restart CAN demos even if only one channel failed. */
                    InitCanApp();
                }
            break;
            /* Error Passive. */
```

```
        case R_CAN_STATUS_ERROR_PASSIVE: //Continue to Bus off case
        /* Bus Off. */
        case R_CAN_STATUS_BUSOFF:
        default:
            /* The application should take note of the following state and
            Stop communication. */
            app_state[ch_nr] = R_CAN_STATUS_BUSOFF;
            if (ch_nr == 0){
                DisplayString(LCD_LINE1,"bus0:   ");
            }
            else {
                DisplayString(LCD_LINE2,"bus1:   ");
            }

            /* Show user */
            LcdShow2DigHex((uint8_t)error_bus_status[ch_nr], ch_nr*16 + 6);
            Delay(0x400000);
            nr_times_reached_busoff[ch_nr]++;
        break;
      }
   }
   error_bus_status_prev[ch_nr] = error_bus_status[ch_nr];
}/* end HandleCanBusState() */
```

5. CAN エラー割り込みの使用

低レベルエラー処理はペリフェラルによって行われるので、通常は API を使用して定期的にポーリングするだけで十分ですが、CAN エラー割り込みを使用してノードのエラー状態をチェックすることができます。

エラーISR から API を呼び出してエラー状態を決定し、状態遷移が生じた場合にアプリケーションにフラグを設定することができます。多くの場合、送信エラーカウンタまたは受信エラーカウンタがインクリメントされるだけです。

割り込みは、単一エラー、エラーパッシブへの遷移、およびバスオフへの遷移のそれぞれに対して個別に有効にすることができます。これらの割り込みの中の最初の CAN エラー割り込みを有効にした場合は、エラーが検出されるたびに割り込みが生成されます。繰り返しになりますが、CAN はエラーを独自に処理するので、この割り込みの生成は通常必要ありません。

5.1 CAN 割り込みチェックリスト

1. このチェックリストを使用して割り込みが正しくセットアップされ、割り込みを発生させることに問題があるかどうかを確認します。
2. #pragma ディレクティブを用いて ISR について「割り込みからの復帰」を行うようにコンパイラに指示します。これは関数が定義されているファイルに指定しなければなりません。例：
```
#pragma interrupt CAN0_RXM0_ISR(vect=VECT_CAN0_RXM0, enable)
void CAN0_RXM0_ISR(void)
{…
```
3. 割り込みは ENABLE_IRQ マクロなどを使用してグローバルに有効になりますか?
   割り込みが発生しない場合は、フラグが無効になっている可能性があります（たとえば、DISABLE_IRQ により）。以下のようにチェックします。すなわち、割り込みが発生することが期待されるコード内の場所で、ブレークポイントを設定し、I-flag が「1」であることを確認します。I-flag の CPU フラグレジスタをチェックします。
4. 割り込みの割り込み優先順位レベルがゼロ以外の値に設定されていますか?
   ハードウェアマニュアルの主要な割り込みの章を参照してください。
5. それぞれのメールボックスの割り込みフラグが有効にされましたか?
6. CAN 割り込みマスクレジスタが割り込みをマスキングするように設定されていますか?
7. CAN 割り込みをネストするには、つまり進行中の CAN 割り込みより先に実行されるように他の割り込みを有効にするには、上記の例のようにベクトル宣言に「enable」引数を追加します。
8. 割り込み ISR 関数がベクトルテーブルによって正しく指し示されていることを確認します。割り込みベースレジスタをチェックして、そこからのオフセットをカウントして、割り込みテーブル内の CAN 割り込み ISR アドレスを確認します。

6. テストモード

たとえば製品開発時などに役立つテストモードがあります。「内部」と「外部」の 2 つのループバックモードおよびリスンオンリーモードがあります。

6.1 内部ループバック-CAN バスを使用しないノードのテスト

内部ループバックモードまたはセルフテストモードでは、バスに接続することなく CAN メールボックスを介して通信することができます。これはアプリケーションのテストまたはアプリケーションデバッグ時の自己診断に役立ちます。

ノードはデータフレームの ACK ビットを使用してそのデータを確認します。ノードは、その CANID に対して設定されていた場合、その送信済みメッセージを受信メールボックスにも格納します。これは、通常では不可能です。

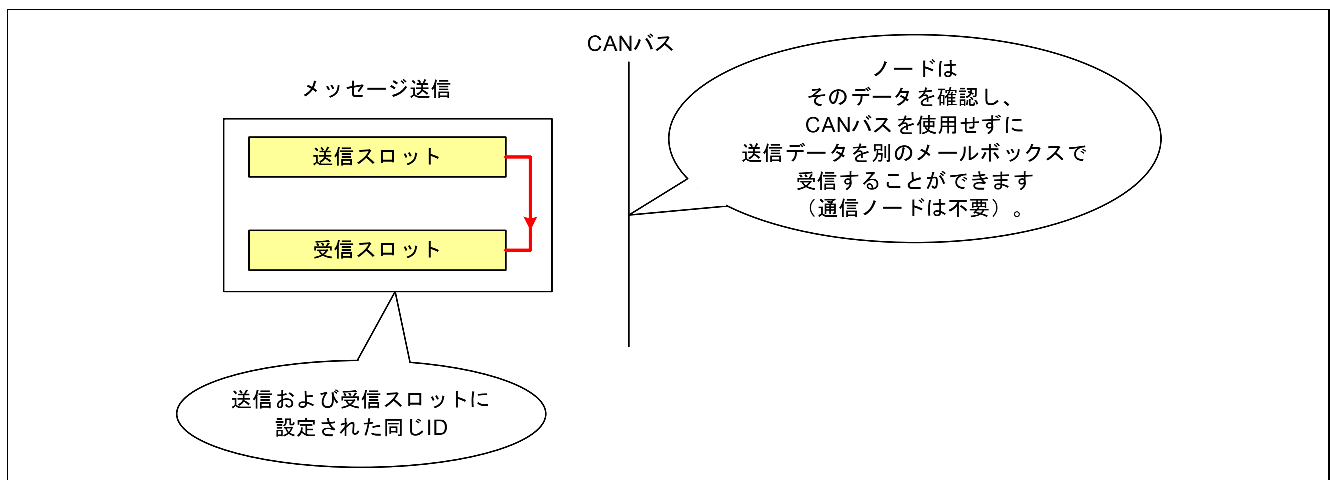

図6.1　CAN 内部ループバックモード

これにより、CAN バスを接続することなくノードの機能をテストすることができます。

内部ループバックは、テスト時に便利です。このモードでは、CAN コントローラはノードがバス上に単独であるときには Ack を受信しないので、CAN エラーを送信することなく動作できるからです。このモードで送信されたフレームを確認します。

6.2 外部ループバック - バス上の単独ノードのテスト

外部ループバックは内部ループバックと同様ですが、唯一の相違はノードに接続された CAN バスが必要であるということです。メッセージはノードによって確認されるので、ノードはバス上で単独であることができます。これはノードをスタンドアロンでテストできるので便利です。

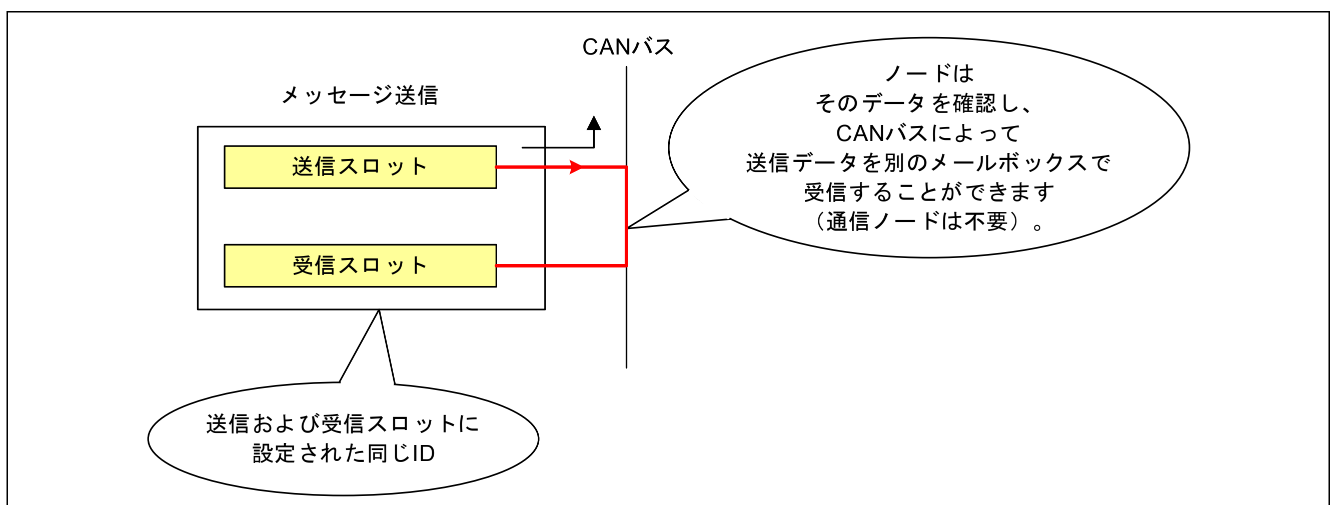

図6.2　外部ループバック

メッセージは CAN バスに送信され、同じノードに戻して受信することができます。これは、コードをテストし、ノードがバス上で単独である場合に便利です。

6.3 リスンオンリー（バス監視） - バスに影響を与えないノードのテスト

リスンオンリーまたはバス監視では、ノードは静的です。リスンオンリーモードのノードは、メッセージの確認やエラーフレームの送信などを行いません。

【注】 リスンオンリーモードに入ることをコードに明確にマークし、リスンオンリーモードを再び無効にすることを忘れないようにしてください。ネットワーク上にノードが 2 つだけあり、その 1 つがリスンオンリーに移行した場合、もう 1 つのノードは Ack を受け取らず、最終的にはバスオフ状態になります。また、リスンオンリーで送信を要求しないでください。これは正しい動作ではなく、CAN モジュールはそのように設計されていません。

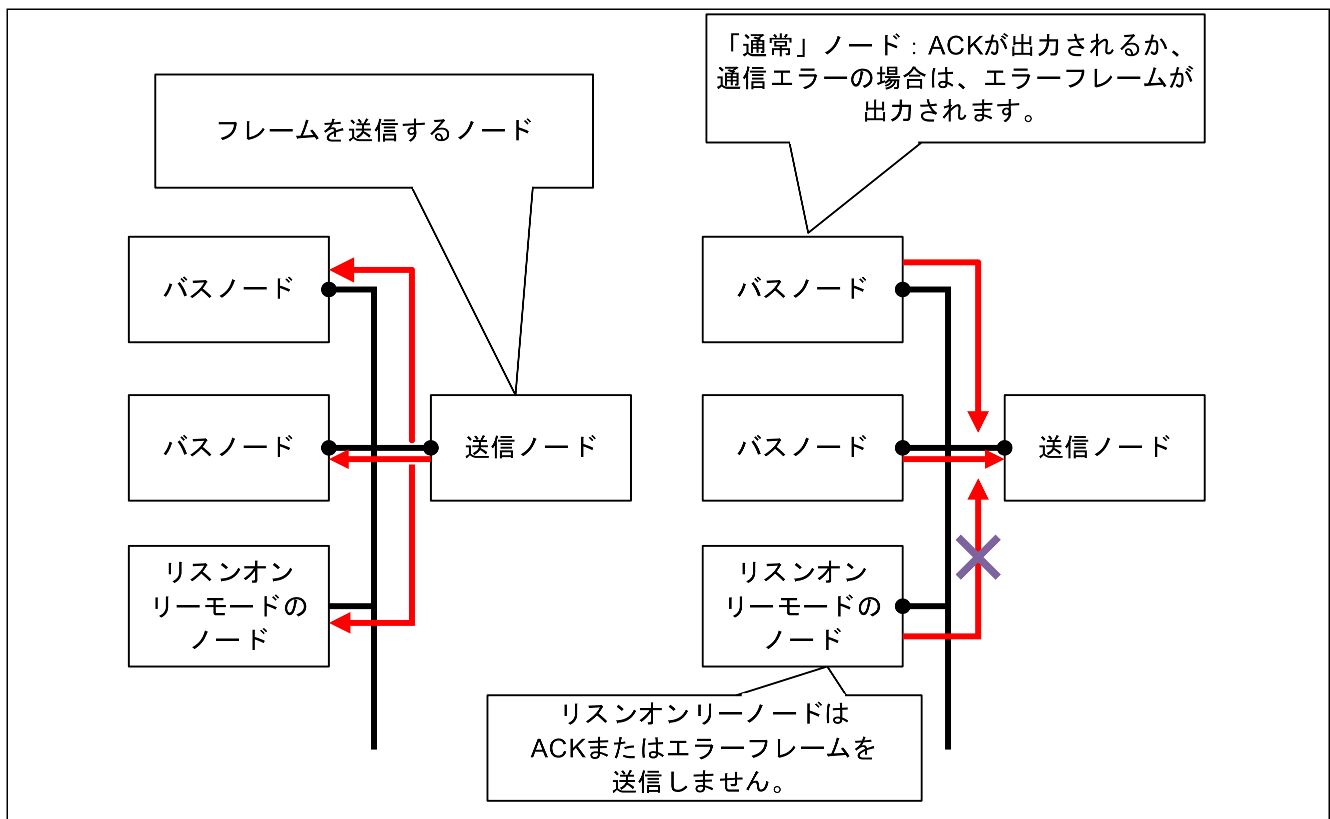

図6.3　リスンオンリーモード

リスンオンリーモードのノードは、メッセージの確認やエラーフレームの送信などを行いません。

リスンオンリーは既存の CAN バスに追加された新しいノードを起動するのに役立ちます。ライブになる前にフレームが正しく受信されるようにするために、このモードは新たに接続されたノードのアプリケーションのために使用することができます。

一般的な使い方は、新しいユニットを「ライブ」にする前にバスの通信速度を検出することです。リスンオンリーは Bosch CAN 仕様の一部ではありませんが、ビットレート検出のために ISO-11898 で要求されています。

7. タイムスタンプ

タイムスタンプ関数は、メッセージを受信するときにオンチップタイムスタンプの値をメールボックスに取り込みます。タイムスタンプを調べることにより、たとえば、メッセージの順序を決定するためにメッセージを複数の受信メールボックスに格納する場合、メッセージの順序を決定することができます。タイムスタンプの読み出しは API によって行われないので、メールボックスをポーリングする必要があり、戻り値が R_CAN_OK（メッセージ待ち）の場合、タイムスタンプを読み出すことができます。

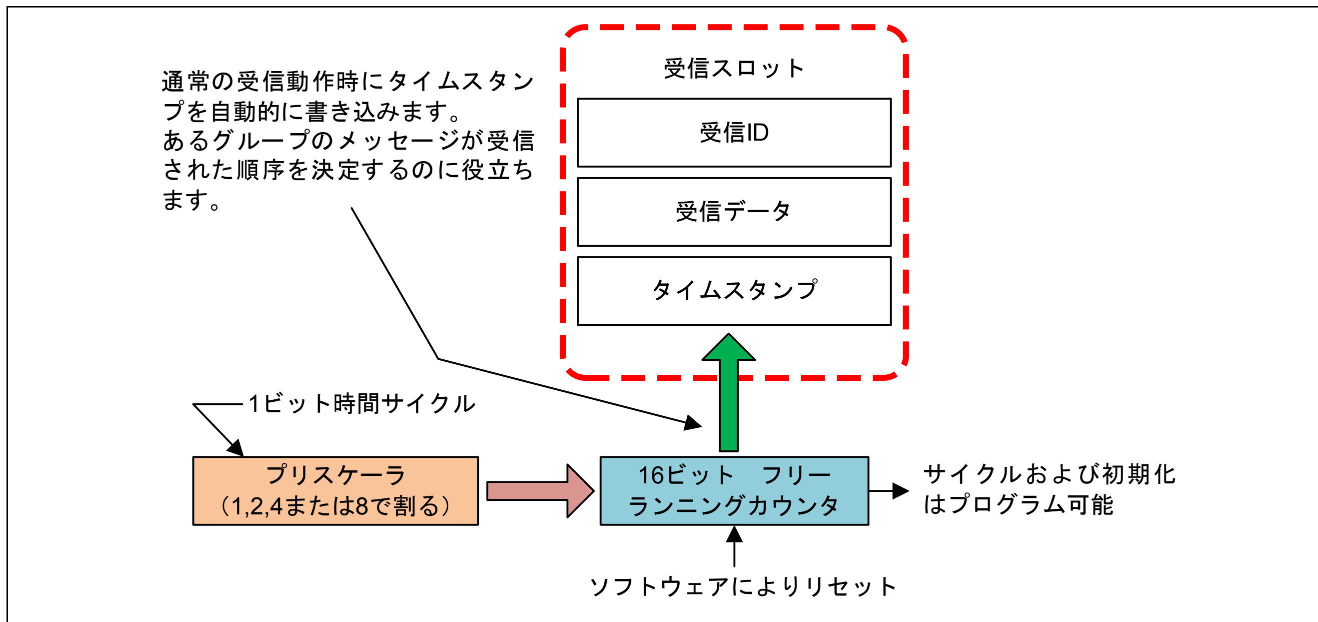

図7.1　CAN タイムスタンプ

CAN タイムスタンプは各メールボックスで使用可能です。

8. CAN スリープモード

MCU リセット後のデフォルトのモードは CAN スリープモードです。API を使用して他の動作モードに切り替えます。R_CAN_Control API を参照してください。CAN スリープモードに移行すると、モジュールに供給されるクロックが直ちに停止し、消費電力が低減されます。CAN モジュールが CAN スリープモードに移行しても、すべてのレジスタは変化しません。

9. CAN FIFO

CAN FIFO バッファリングは RX で使用可能です。送信または受信に 24 のメールボックスを設定することができます。FIFO はポーリングまたは割り込みに使用することができますが、本アプリケーションノートに付属するドライバはこの機能をサポートしていません。

ホームページとサポート窓口

ルネサス エレクトロニクスホームページ

http://japan.renesas.com/

お問合せ先

http://japan.renesas.com/inquiry

改訂記録	RX600 シリーズ　アプリケーションノート CAN アプリケーションプログラミングインタフェース

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2010.06.01	—	初版発行
1.10	2010.06.06		AN 番号を REU05B0145 から R01AN0339EU に変更。 RX62T と RX630 のプロジェクトを追加。
1.11	2011.10.02	4 6 全体	CAPI_CFG_CAN_ISR を CAPI_CFG_CANx_ISR に変更。 API リターンコード「R_CAN_NOT_OK」を追加。 「config_r_can_rap.h」を「config_r_can_rapi.h」に変更。
2.00	2011.01.10	1 全体 5	プロパティを RX600 シリーズに変更。 拡張 CAN を追加。 RX63N ワークスペースを追加。 今後廃止予定の（不要となる）マクロ - USE_CAN_API_SEARCH の下のテキストを変更。 - CAPI_CFG_CANx_ISR を削除。
2.01	2012.05.16 2012.10.10	全体 コード AN	RSKRX630 CAN アプリケーションノートのレビュー後に変更。 （v.1.10 をレビュー。多くはすでに修正済みのコメント） 62G RSK CAN API デモを追加。 すべての図に目を通して視認性を向上。見出しの第 3 章を変更。
2.02	2012.11.13	コード	スリープモードを終了する前に確実に休止モードに移行するように R_CAN_Control()スリープモードを変更。 標準と拡張の両方のフレームを扱うための can_frame_t
2.03	2013.03.23	コード	RSK63N SW1 を P32 から P02 に変更。
2.04	2016.06.07	コード	一部の事例で R_CAN_Control()呼び出しの引数として ch_nr の代わりに 0 を指定。
2.05	2018.02.07	コード	拡張 ID または混合 ID モード時にメールボックス IDE ビットが正しく使用されないエラーを修正。 デモプロジェクトを e^{2}studio へ移植。HEW サポートを削除。
		1	特別にサポートされるターゲットデバイスを特定。
		1	他の旧式 MCU に関する無関係な記述を削除。
		36	FIFO 処理がドライバによってサポートされない注意書きを追加。
		6、7	config_r_can_rapi.h 設定に変更を追加。
		10	R_CAN_Create()の記述を修正。
		全体	ヘッダーとフッターのルネサスロゴカラーを修正。 各章の構成を見直し、章・節・項番号を再設定。

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。

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. 未使用端子の処理
   【注意】未使用端子は、本文の「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。未使用端子は、本文「未使用端子の処理」で説明する指示に従い処理してください。
2. 電源投入時の処置
   【注意】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. リザーブアドレス（予約領域）のアクセス禁止
   【注意】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、将来の機能拡張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。
4. クロックについて
   【注意】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。
5. 製品間の相違について
   【注意】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、内部ROM、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、ノイズ輻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。お客様の機器・システムの設計において、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には、お客様の責任において行ってください。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、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、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。
5. 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。
6. 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。
7. 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。
8. 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。
9. 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。
10. 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。
11. 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。
12. 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.4.0-1 2017.11)

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口

http://www.renesas.com

※営業お問合せ窓口の住所は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス株式会社　〒135-0061　東京都江東区豊洲3-2-24（豊洲フォレシア）

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：https://www.renesas.com/contact/

© 2018 Renesas Electronics Corporation. All rights reserved.

Colophon 6.0