

RX ファミリ

ハイスピード I²C バスインタフェース(RIICHS)モジュール

Firmware Integration Technology

要旨

本アプリケーションノートでは、Firmware Integration Technology (FIT)を使用したハイスピード I²C バスインタフェースモジュール (RIICHS) について説明します。本モジュールは RIICHS を使用して、デバイス間で通信を行います。以降、本モジュールを RIICHS FIT モジュールと称します。

対象デバイス

- RX671 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あわせて変更し、十分評価してください。

対象コンパイラ

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

目次

1. 概要	4
1.1 RIICHS FIT モジュールとは	4
1.2 API の概要	5
1.3 RIICHS FIT モジュールの概要	6
1.3.1 RIICHS FIT モジュールの仕様	6
1.3.2 マスタ送信の処理	7
1.3.3 マスタ受信の処理	10
1.3.4 スレーブ送受信の処理	13
1.3.5 状態遷移図	17
1.3.6 状態遷移時の各フラグ	18
1.3.7 アービトレーションロスト検出機能	19
1.3.8 タイムアウト検出機能	20
2. API 情報	21
2.1 ハードウェアの要求	21
2.2 ソフトウェアの要求	21
2.3 サポートされているツールチェーン	21
2.4 使用する割り込みベクタ	21
2.5 ヘッダファイル	21
2.6 整数型	21
2.7 コンパイル時の設定	22
2.8 コードサイズ	23
2.9 引数	24
2.10 戻り値	25
2.11 コールバック関数	26
2.12 モジュールの追加方法	27
2.13 for 文、while 文、do while 文について	28
3. API 関数	29
R_RIICHS_Open()	29
R_RIICHS_MasterSend()	33
R_RIICHS_MasterReceive()	37
R_RIICHS_SlaveTransfer()	41
R_RIICHS_GetStatus()	46
R_RIICHS_Control()	49
R_RIICHS_Close()	51
R_RIICHS_GetVersion()	52
4. 端子設定	53
5. デモプロジェクト	54
5.1 riichs_mastersend_demo_rskrx671	54
5.2 riichs_masterreceive_demo_rskrx671	54

5.3	riichs_slavetransfer_demo_rskrx671.....	54
5.4	ワークスペースにデモを追加する	55
5.5	デモのダウンロード方法	55
6.	付録	56
6.1	通信方法の実現	56
6.1.1	制御時の状態	56
6.1.2	制御時のイベント	56
6.1.3	プロトコル状態遷移	57
6.1.4	プロトコル状態遷移表	61
6.1.5	プロトコル状態遷移登録関数	62
6.1.6	状態遷移時の各フラグの状態	62
6.2	割り込み発生タイミング	64
6.2.1	マスタ送信	64
6.2.2	マスタ受信	65
6.2.3	マスタ送受信	65
6.2.4	スレーブ送信	66
6.2.5	スレーブ受信	66
6.2.6	マルチマスタ通信（マスタ送信中の AL 検出後、スレーブ送信）	67
6.2.7	Hs モードでのマスタ送信	67
6.2.8	Hs モードでのマスタ受信	68
6.3	タイムアウトの検出、および検出後の処理	69
6.3.1	タイムアウト検出機能によるタイムアウト検出	69
6.3.2	タイムアウト検出後の対応方法	69
6.4	動作確認環境	71
6.5	トラブルシューティング	73
6.6	サンプルコード	74
6.6.1	1つのチャンネルで1つのスレーブデバイスに連続アクセスする場合の例	74
7.	参考ドキュメント	80
	テクニカルアップデートの対応について	81
	改訂記録	82

1. 概要

RIICHS FIT モジュールは、RIICHS を使用し、マスタデバイスとスレーブデバイスが送受信を行うための手段を提供します。RIICHS は NXP 社が提唱する I²C バス(Inter-IC-Bus)インタフェース方式に準拠しています。以下に本モジュールがサポートしている機能を列挙します。

- マスタ送信、マスタ受信、スレーブ送信、スレーブ受信に対応
- 複数のマスタがひとつのスレーブと調停を行いながら通信するマルチマスタ構成
- 通信モードはスタンダードモード、ファストモード、ファストモードプラス、ハイスピードモードに対応。スタンダードモードでは最大転送速度は 100kbps。ファストモードでは最大転送速度は 400kbps。ファストモードプラスでは最大転送速度は 1Mbps。ハイスピードモードでは最大転送速度は 3.4Mbps。

制限事項

本モジュールには以下の制限事項があります。

- (1) DMAC、DTC と組み合わせて使用することはできません。
- (2) RIICHS の NACK アービトレーションロスト機能に対応していません。
- (3) 10 ビットアドレスの送信に対応していません。
- (4) スレーブデバイス時、リスタートコンディションの受け付けに対応していません。
リスタートコンディション直後のアドレスで本モジュールを組み込んだデバイスのアドレスを指定しないでください。
- (5) 本モジュールは多重割り込みには対応していません。
- (6) コールバック関数内では R_RIICHS_GetStatus 関数以外の API 関数の呼び出しは禁止です。
- (7) 割り込みを使用するため、I フラグは“1”で使用してください。

1.1 RIICHS FIT モジュールとは

本モジュールは API として、プロジェクトに組み込んで使用します。本モジュールの組み込み方については、「2.12 モジュールの追加方法」を参照してください。

1.2 API の概要

表 1.1に本モジュールに含まれる API 関数を示します。

表 1.1 API 関数一覧

関数	関数説明
R_RIICHS_Open()	この関数は RIICHS FIT モジュールを初期化する関数です。この関数は他の API 関数を使用する前に呼び出される必要があります。
R_RIICHS_MasterSend()	マスタ送信を開始します。引数に合わせてマスタのデータ送信パターンを変更します。ストップコンディション生成まで一括で実施します。
R_RIICHS_MasterReceive()	マスタ受信を開始します。引数に合わせてマスタのデータ受信パターンを変更します。ストップコンディション生成まで一括で実施します。
R_RIICHS_SlaveTransfer()	スレーブ送受信を行う関数。 引数のパターンに合わせてデータ送受信パターンを変更します。
R_RIICHS_GetStatus()	本モジュールの状態を返します。
R_RIICHS_Control()	各コンディション出力、SDA 端子のハイインピーダンス出力、SCL クロックのワンショット出力、および RIICHS のモジュールリセットを行う関数です。主に通信エラー時に使用してください。
R_RIICHS_Close()	RIICHS の通信を終了し、使用していた RIICHS の対象チャネルを解放します。
R_RIICHS_GetVersion()	本モジュールのバージョンを返します。

1.3 RIICHS FIT モジュールの概要

1.3.1 RIICHS FIT モジュールの仕様

- 1) 本モジュールは、マスタ送信、マスタ受信、スレーブ送信、スレーブ受信 をサポートします。
 - マスタ送信では 4 種類の送信パターンが設定可能です。マスタ送信の詳細は1.3.2に示します。
 - マスタ受信では、マスタ受信とマスタ送受信の 2 種類の受信パターンが設定可能です。マスタ受信の詳細は1.3.3に示します。
 - スレーブ受信とスレーブ送信は、マスタから送信されるデータの内容によって、その後の動作を行います。スレーブ受信の詳細は、「1.3.4スレーブ送受信の処理」の「(1) スレーブ受信」に、スレーブ送信の詳細は、「1.3.4スレーブ送受信の処理」の「(2) スレーブ送信」に示します。
- 2) 割り込みは、スタートコンディション生成、スレーブアドレス送信／受信、データ送信／受信、NACK 検出、アービトレーションロスト検出、ストップコンディション生成のいずれかの処理が完了すると発生します。RIICHS の割り込み内で本モジュールの通信制御関数を呼び出し、処理を進めます。
- 3) 1つのチャンネル・バス上の複数かつアドレスが異なるスレーブデバイスを制御できます。ただし、通信中(スタートコンディション生成から、ストップコンディション生成完了までの期間)は、そのデバイス以外の通信はできません。図 1.1に複数スレーブデバイスの制御例を示します。

(例) ch0にスレーブデバイスAとBが接続されている場合

ST : スタートコンディション、SP : ストップコンディション

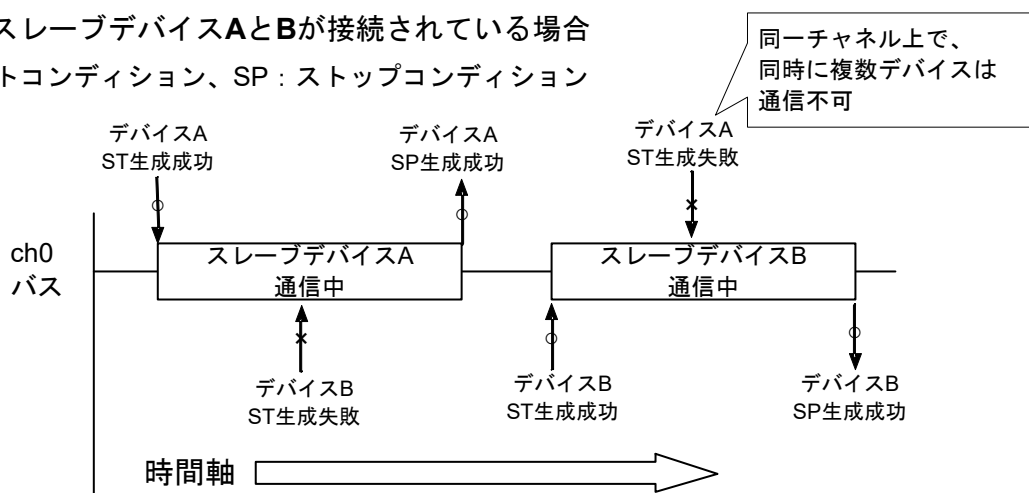


図 1.1 複数スレーブデバイスの制御例

1.3.2 マスタ送信の処理

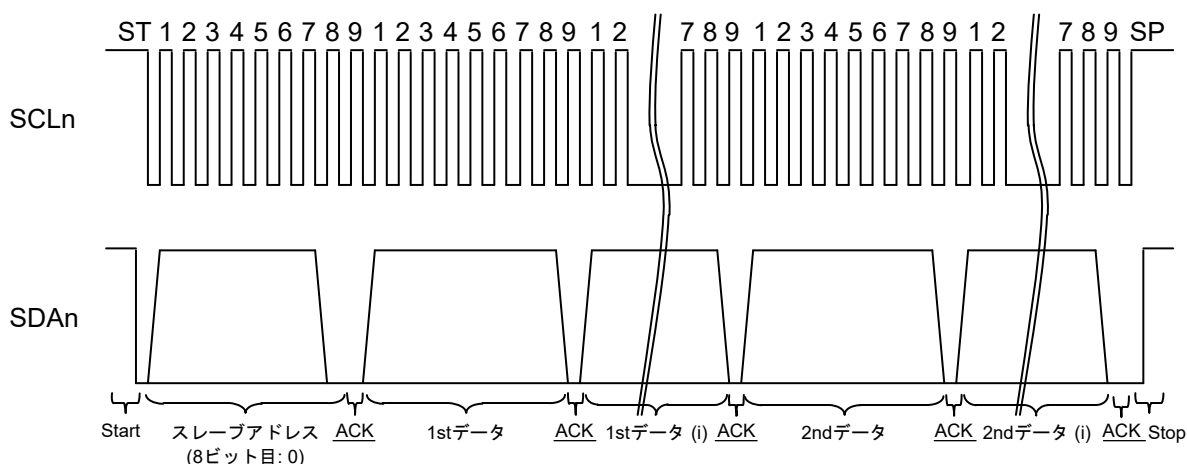
マスタデバイスとして、スレーブデバイスヘータを送信します。

本モジュールでは、マスタ送信は4種類の波形を生成できます。マスタ送信の際、引数とする I²C 通信情報構造体の設定値によってパターンを選択します。図 1.2～図 1.4に3種類の送信パターンを示します。また、I²C 通信情報構造体の詳細は、2.9を参照してください。

(1) パターン 1

マスタデバイスとして、2つのバッファのデータ(1st データと 2nd データ)をスレーブデバイスへ送信する機能です。

初めにスタートコンディション(ST)を生成し、次にスレーブデバイスのアドレスを送信します。このとき、8ビット目は転送方向指定ビットになりますので、データ送信時には“0”(Write)を送信します。次に 1st データを送信します。1st データとは、データ送信を行う前に、事前に送信したいデータがある場合に使用します。例えばスレーブデバイスが EEPROM の場合、EEPROM 内部のアドレスを送信することができます。次に 2nd データを送信します。2nd データがスレーブデバイスへ書き込むデータになります。データ送信を開始し、全データの送信が完了すると、ストップコンディション(SP)を生成してバスを解放します。



凡例:

n: チャンネル番号

ST: スタートコンディション生成

SP: ストップコンディション生成

ACK: Acknowledge“0”

※ 下線のある信号は、スレーブからマスタへのデータ送信です

図 1.2 マスタ送信(パターン 1)信号図

(2) パターン 2

マスタデバイスとして、1つのバッファのデータ(2nd データ)をスレーブデバイスへ送信する機能です。

スタートコンディション(ST)の生成からスレーブデバイスのアドレスを送信まではパターン 1 と同様に動作します。次に 1st データを送信せず、2nd データを送信します。全データの送信が完了すると、ストップコンディション(SP)を生成してバスを解放します。

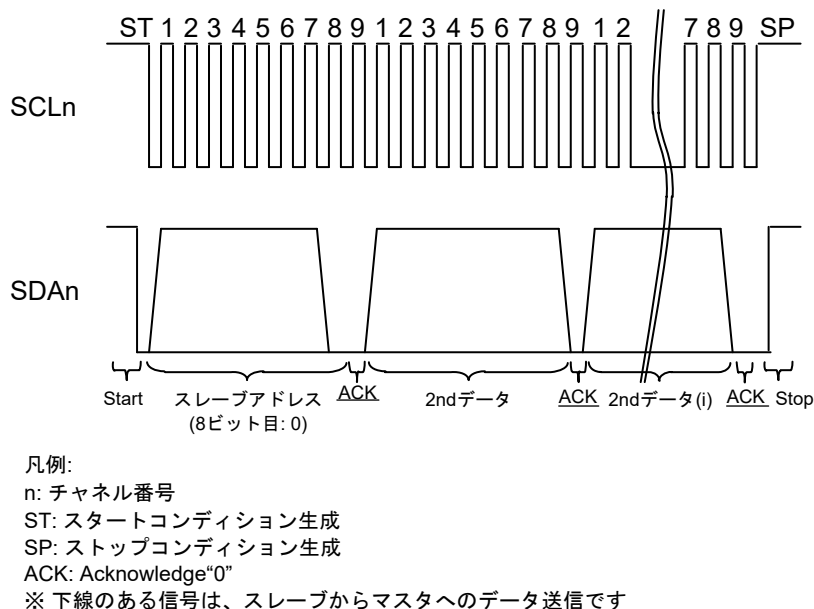


図 1.3 マスタ送信(パターン 2)信号図

(3) パターン 3

マスタデバイスとして、スレーブアドレスのみをスレーブデバイスへ送信する機能です。

スタートコンディション(ST)を生成から、スレーブアドレス送信まではパターン 1 と同様に動作します。スレーブアドレス送信後、1st データ/2nd データを設定していない場合、データ送信は行わず、ストップコンディション(SP)を生成してバスを解放します。

接続されているデバイスを検索する場合や、EEPROM 書き換え状態を確認する Acknowledge Polling を行う際に有効な処理です。

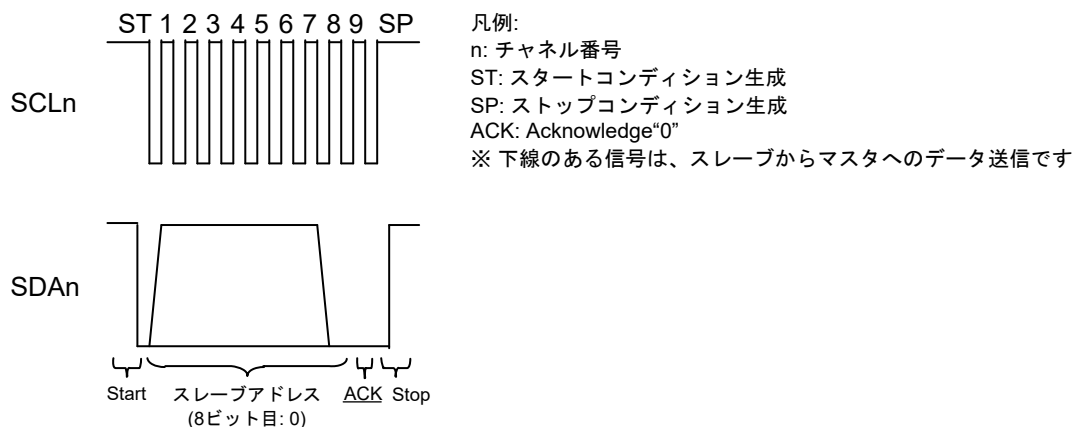


図 1.4 マスタ送信(パターン 3)信号図

図 1.5にマスタ受信を行う際の手順を示します。コールバック関数は、ストップコンディション生成後に呼ばれます。I²C 通信情報構造体メンバの CallBackFunc に関数名を指定してください。

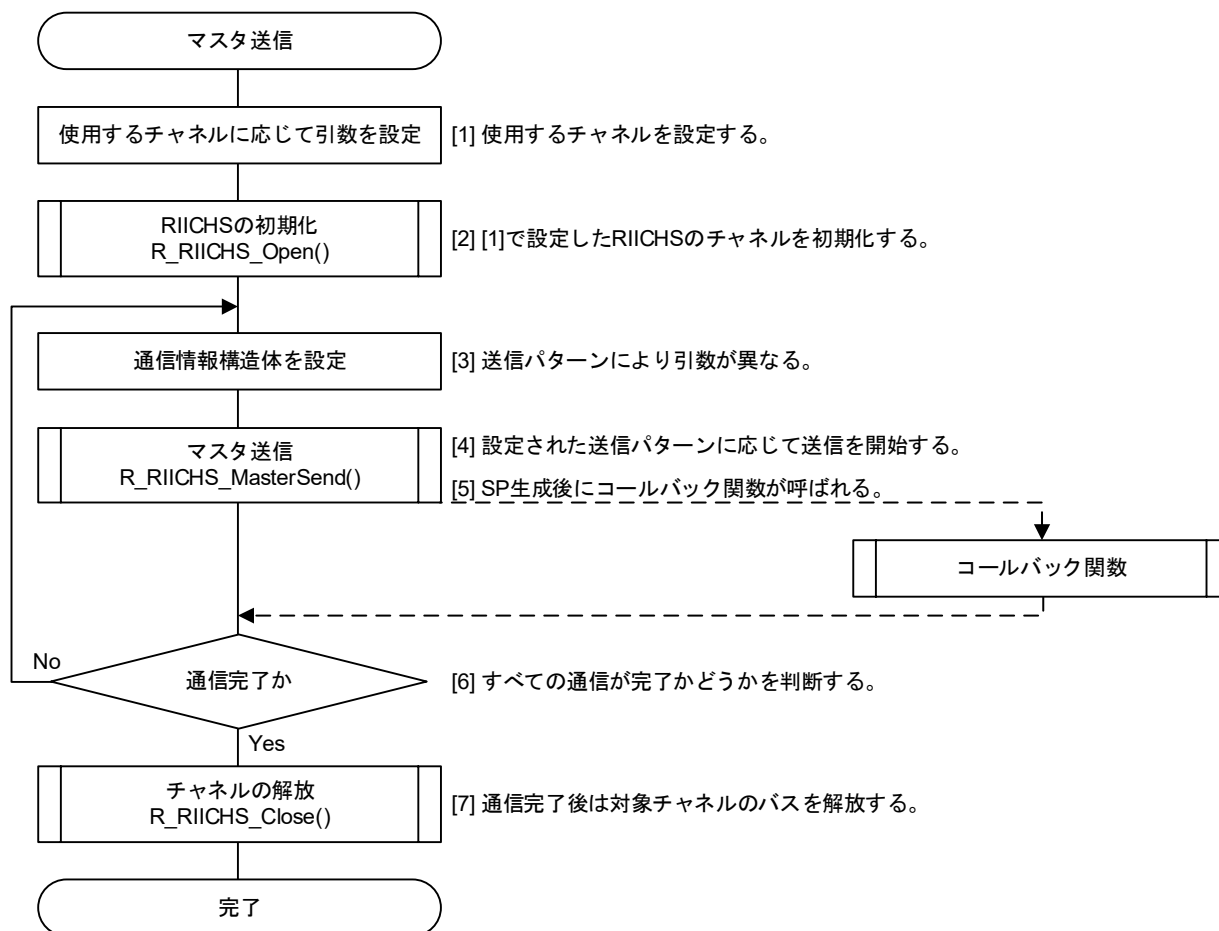


図 1.5 マスタ送信の処理例

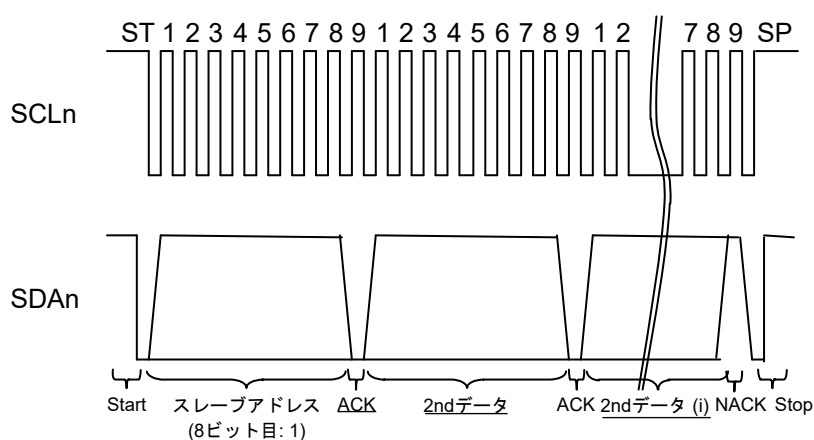
1.3.3 マスタ受信の処理

マスタデバイスとして、スレーブデバイスからデータを受信します。本モジュールでは、マスタ受信とマスタ送受信に対応しています。マスタ受信の際の引数とする I²C 通信情報構造体の設定値によってパターンを選択します。図 1.6～図 1.7に受信パターンを示します。また、I²C 通信情報構造体の詳細は、2.9を参照してください。

(1) マスタ受信

マスタデバイスとして、スレーブデバイスからデータを受信する機能です。

初めにスタートコンディション(ST)を生成し、次にスレーブデバイスのアドレスを送信します。このとき、8ビット目は転送方向指定ビットになりますので、データ受信時には“1”(Read)を送信します。次にデータ受信を開始します。受信中は、1バイト受信するごとに ACK を送信しますが、最終データ時のみ NACK を送信し、スレーブデバイスへ受信処理が完了したことを通知します。全データの受信が完了すると、ストップコンディション(SP)を生成してバスを解放します。



凡例:

n: チャネル番号

ST: スタートコンディション生成

NACK: Acknowledge "1"

SP: ストップコンディション生成

ACK: Acknowledge "0"

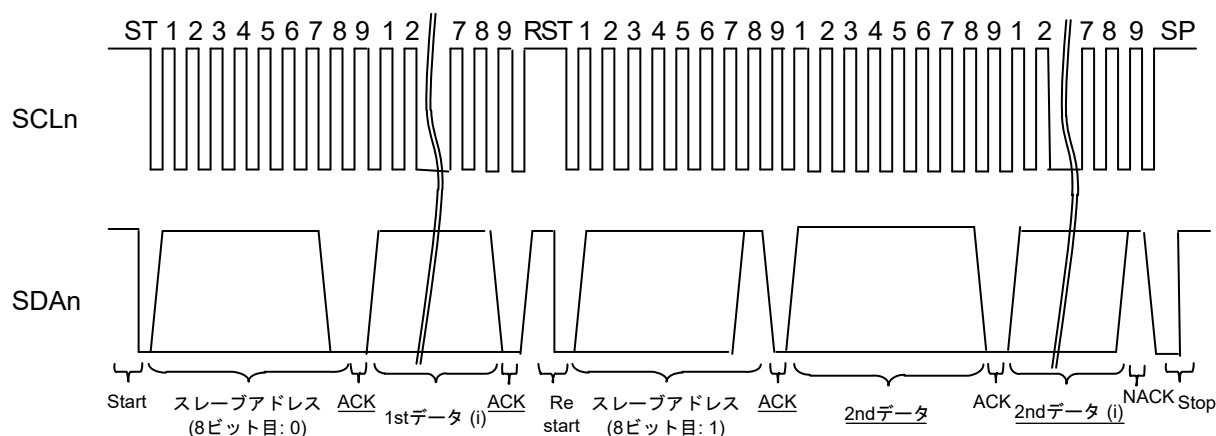
※ 下線のある信号は、スレーブからマスタへのデータ送信です

図 1.6 マスタ受信 信号図

(2) マスタ送受信

マスタデバイスとして、スレーブデバイスへデータを送信します。送信完了後、リスタートコンディションを生成し、スレーブデバイスからデータを受信する機能です。

初めにスタートコンディション(ST)を生成し、次にスレーブデバイスのアドレスを送信します。このとき、8ビット目の転送方向指定ビットには、“0” (Write)を送信します。次に1st データを送信します。データの送信が完了すると、リスタートコンディション(RST)を生成し、スレーブアドレスを送信します。このとき、転送方向指定ビットには、“1” (Read)を送信します。次にデータ受信を開始します。受信中は、1バイト受信するごとに ACK を送信しますが、最終データ時のみ NACK を送信し、スレーブデバイスへ受信処理が完了したことを通知します。全データの受信が完了すると、ストップコンディション(SP)を生成してバスを解放します。



凡例:

n: チャネル番号

ST: スタートコンディション生成

NACK: Acknowledge "1"

SP: ストップコンディション生成

ACK: Acknowledge "0"

RST: リスタートコンディション生成

※ 下線のある信号は、スレーブからマスタへのデータ送信です

図 1.7 マスタ送受信 信号図

図 1.8にマスタ受信を行う際の手順を示します。コールバック関数は、ストップコンディション生成後に呼ばれます。I²C 通信情報構造体メンバの CallBackFunc に関数名を指定してください。

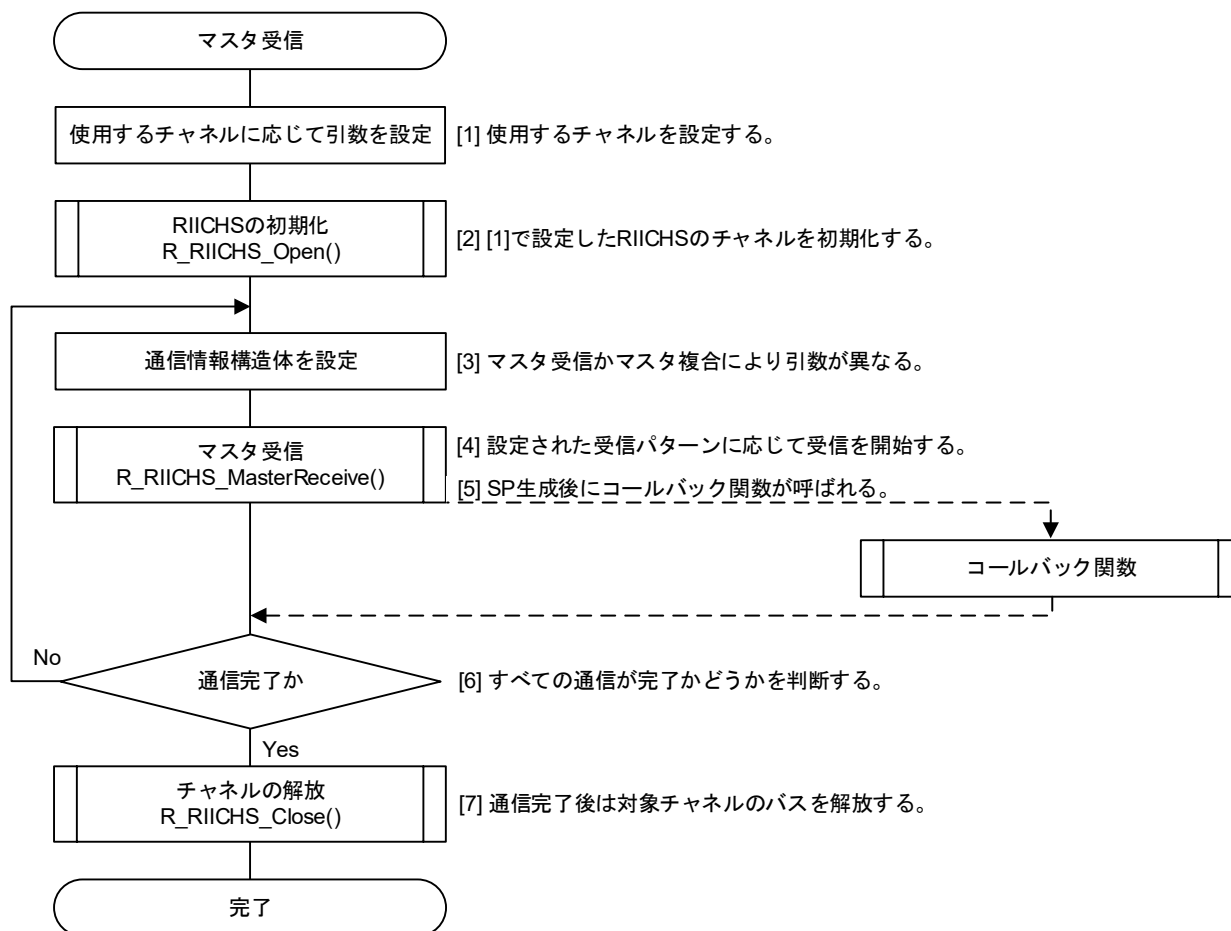


図 1.8 マスタ受信の処理例

マスタデバイスから送信されるデータを、スレーブデバイスとして受信します。

マスタデバイスが指定するスレーブアドレスが、関数 `R_RIICHS_Open()` の引数で設定したスレーブデバイスのスレーブアドレスと一致したとき、スレーブ送受信を開始します。スレーブアドレスの 8 ビット目(転送方向指定ビット)によって、本モジュールが自動的にスレーブ受信かスレーブ送信かを判断して処理を行います。

スレーブデバイスとして、マスタデバイスからのデータを受信する機能です。

マスタデバイスが生成したスタートコンディション(ST)を検出した後に、受信したスレーブアドレスが、自アドレスと一致し、かつスレーブアドレスの8ビット目(転送方向指定ビット)が“0”(Write)のとき、スレーブデバイスとして受信動作を開始します。最終データ(I²C 通信情報構造体に設定された受信データ数)を受信時は、NACKを返すことでマスタデバイスに必要なデータをすべて受信したことを通知します。図 1.9にスレーブ受信の信号図を示します。

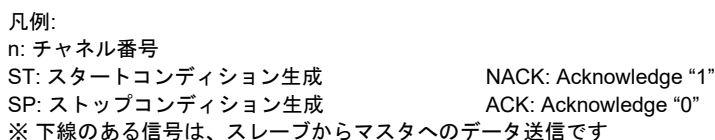


図 1.9 スレーブ受信 信号図

図 1.10にスレーブ受信を行う際の手順を示します。コールバック関数は、ストップコンディション検出後に呼ばれます。I²C 通信情報構造体メンバの CallBackFunc に関数名を指定してください。

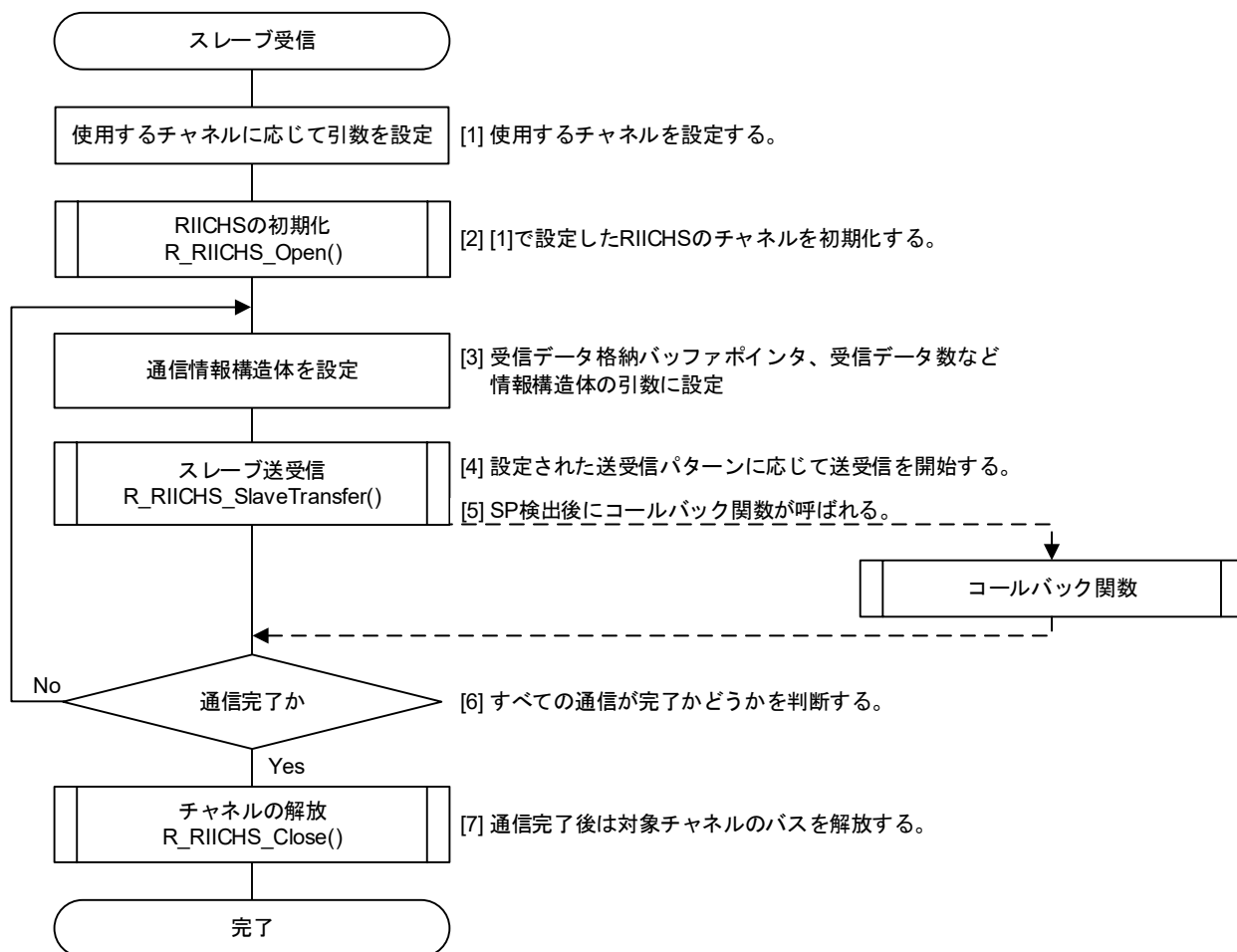


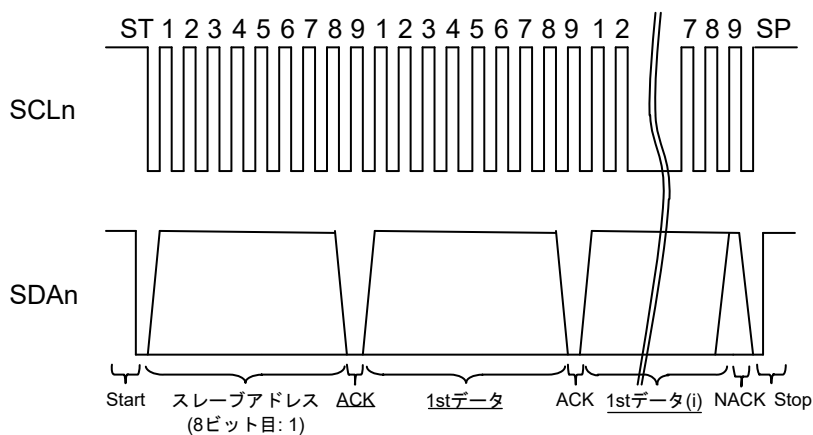
図 1.10 スレーブ受信の処理例

(2) スレーブ送信

スレーブデバイスとして、マスタデバイスヘータを送信する機能です。

マスタデバイスからのスタートコンディション(ST)検出後に、受信したスレーブアドレスが、自アドレスと一致し、かつスレーブアドレスの 8 ビット目(転送方向指定ビット)が“1”(Read)のとき、スレーブデバイスとして送信動作を開始します。設定したデータ数(I²C 通信情報構造体に設定した送信データ数)を超える送信データの要求があった場合、“0xFF”をデータとして送信します。ストップコンディション(SP)を検出するまでデータを送信します。

図 1.11にスレーブ送信の信号図を示します。



凡例:

n: チャネル番号

ST: スタートコンディション生成

NACK: Acknowledge "1"

SP: ストップコンディション生成

ACK: Acknowledge "0"

※ 下線のある信号は、スレーブからマスタへのデータ送信です

図 1.11 スレーブ送信 信号図

図 1.12にスレーブ送信を行う際の手順を示します。コールバック関数は、ストップコンディション検出後に呼ばれます。I²C 通信情報構造体メンバの CallBackFunc に関数名を指定してください。

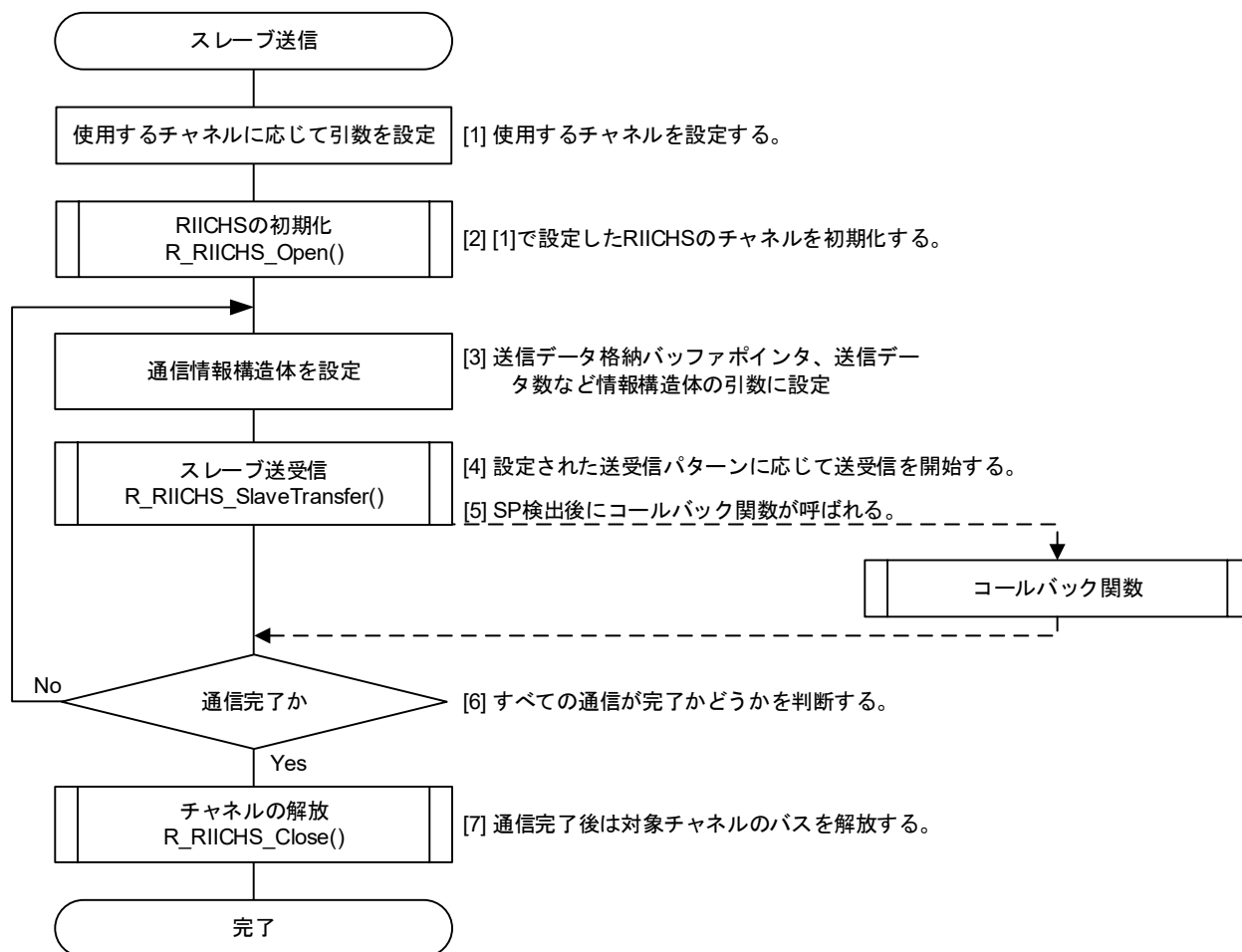


図 1.12 スレーブ送信の処理例

1.3.5 状態遷移図

本モジュールの状態遷移図を図 1.13に示します。

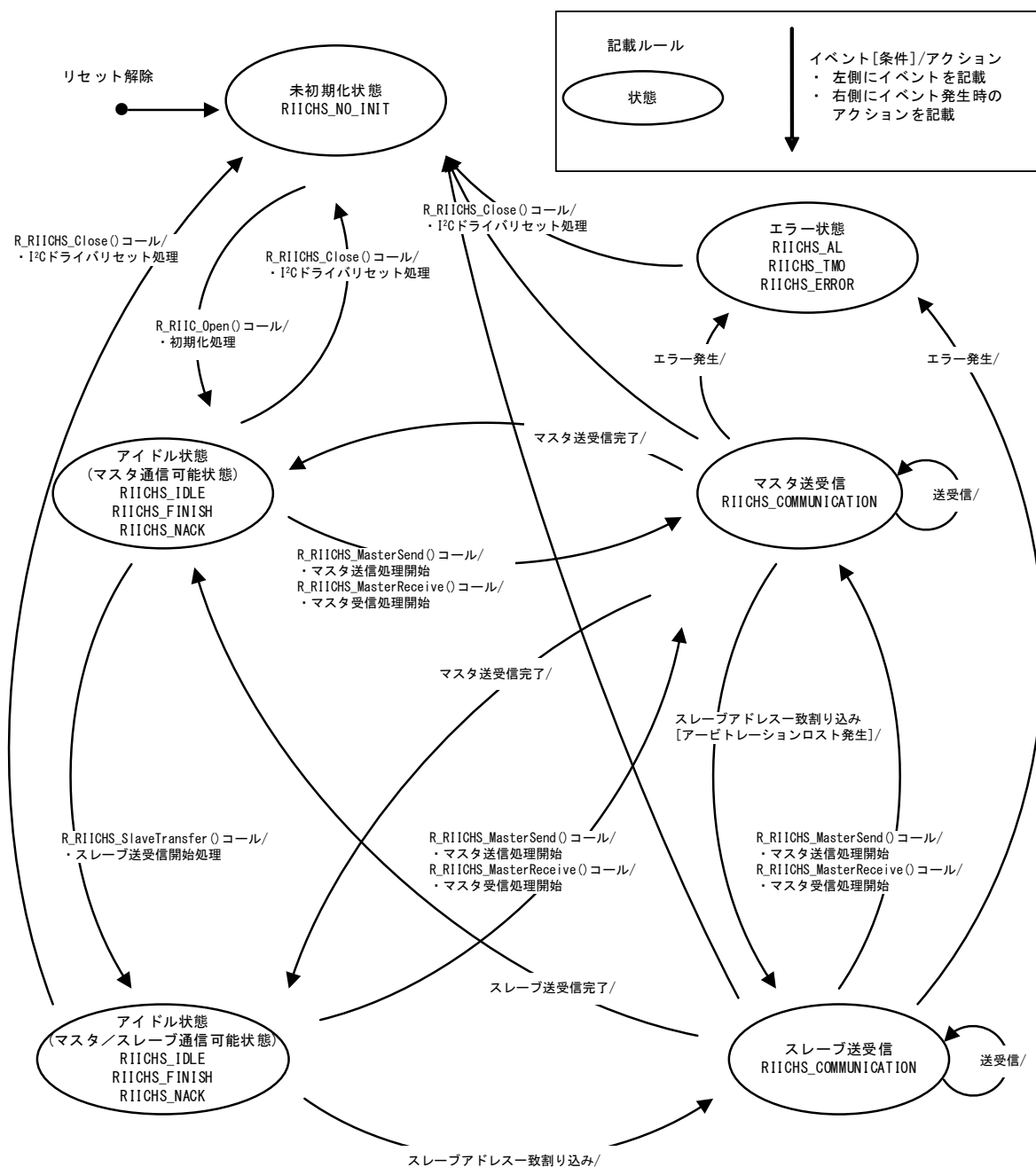


図 1.13 RIICHS FIT モジュールの状態遷移図

1.3.6 状態遷移時の各フラグ

I²C 通信情報構造体メンバには、デバイス状態フラグ(dev_sts)があります。デバイス状態フラグには、そのデバイスの通信状態が格納されます。また、このフラグにより、同一チャネル上の複数のスレーブデバイスの制御を行うことができます。表 1.2に状態遷移時のデバイス状態フラグの一覧を示します。

表 1.2 状態遷移時のデバイス状態フラグの一覧

状態	デバイス状態フラグ(dev_sts)
未初期化状態	RIICHS_NO_INIT
アイドル状態	RIICHS_IDLE RIICHS_FINISH RIICHS_NACK
通信中 (マスタ送信、マスタ受信、 スレーブ送信、スレーブ受信)	RIICHS_COMMUNICATION
アービトレーションロスト検出状態	RIICHS_AL
タイムアウト検出状態	RIICHS_TMO
エラー	RIICHS_ERROR

1.3.7 アービトレーションロスト検出機能

本モジュールは、以下に示すアービトレーションロストを検出することができます。なお、RIICHS は、以下に加えて、スレーブ送信時におけるアービトレーションロストの検出にも対応していますが、本モジュールは対応していません。

(1) バスビジー状態で、スタートコンディションを発行したとき

既に他のマスタデバイスがスタートコンディションを発行して、バスを占有している状態(バスビジー状態)でスタートコンディションを発行すると、本モジュールはアービトレーションロストを検出します。

(2) バスビジー状態ではないが、他のマスタより遅れてスタートコンディションを発行したとき

本モジュールは、スタートコンディションを発行するとき、SDA ラインを Low にしようとします。しかし、他のマスタデバイスがこれよりも早くスタートコンディションを発行した場合、SDA ライン上の信号レベルは、本モジュールが出力したレベルと一致しくなくなります。このとき、本モジュールはアービトレーションロストを検出します。

(3) スタートコンディションが同時に発行されたとき

複数のマスタデバイスが、同時にスタートコンディションを発行すると、それぞれのマスタデバイス上でスタートコンディションの発行が正常に終了したと判断されることがあります。その後、それぞれのマスタデバイスは通信を開始しますが、以下に示す条件が成立すると、本モジュールはアービトレーションロストを検出します。

a. それぞれのマスタデバイスが送信するデータが異なる場合

データ通信中、本モジュールは SDA ライン上の信号レベルと、本モジュールが出力したレベルを比較しています。そのため、スレーブアドレス送信を含むデータ送信中に、SDA ライン上と本モジュールが出力したレベルが一致しなくなると、本モジュールはその時点でアービトレーションロストを検出します。

b. それぞれのマスタデバイスが送信するデータは同じだが、データの送信回数が異なる場合

上記 a.に合致しない場合(スレーブアドレスおよび送信データが同じ)、本モジュールはアービトレーションを検出しませんが、それぞれのマスタデバイスがデータを送信する回数が異なる場合であれば、本モジュールはアービトレーションロストを検出します。

1.3.8 タイムアウト検出機能

本モジュールは、タイムアウト検出機能を有効にすることができます(デフォルト有効)。タイムアウト検出機能では SCL ラインが Low または High に固定されたまま一定時間以上経過したことを検知し、バスの異常状態を検出します。

タイムアウト検出機能は以下の期間で SCL ラインの Low 固定または High 固定のバスハングアップを検出します。

- (1) マスタモードで、バスビジー
- (2) スレーブモードで、自スレーブアドレス一致かつバスビジー
- (3) スタートコンディション発行要求中で、バスフリー

タイムアウト検出機能の有効/無効の設定方法については、「2.9 引数」を参照ください。

タイムアウト検出時の対応方法については、「6.3 タイムアウトの検出、および検出後の処理」を参照ください。

2. API 情報

本 FIT モジュールは、下記の条件で動作を確認しています。

2.1 ハードウェアの要求

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。

- RIICHS

2.2 ソフトウェアの要求

FIT モジュールは以下の FIT モジュールに依存しています。

- ボードサポートパッケージモジュール (r_bsp) Rev.6.10 以上

2.3 サポートされているツールチェーン

本 FIT モジュールは下記ツールチェーンで動作確認を行っています。詳細は、「6.4 動作確認環境」を参照ください。

- Renesas RX Toolchain v.3.06.00

2.4 使用する割り込みベクタ

(マクロ定義 RIICHS_CFG_CH0_INCLUDED が 1 の時)、R_RIICHS_MasterSend 関数、R_RIICHS_MasterReceive 関数、R_RIICHS_SlaveTransfer 関数を呼び出したとき、(引数のパラメータで指定したチャンネル番号のチャンネルに対応した)EEI 割り込み、RXI 割り込み、TXI 割り込み、TEI 割り込みが有効になります。表 2.1 に RIICHS FIT モジュールが使用する割り込みベクタを示します。

表 2.1 使用する割り込みベクター一覧

デバイス	割り込みベクタ
RX671	RXI0 割り込み[チャンネル 0] (ベクタ番号 : 118) TXI0 割り込み[チャンネル 0] (ベクタ番号 : 119) GROUPAL1 割り込み (ベクタ番号 : 113) <ul style="list-style-type: none">• TEI0 割り込み[チャンネル 0] (グループ割り込み要因番号 : 12)• EEI0 割り込み[チャンネル 0] (グループ割り込み要因番号 : 13)

2.5 ヘッドファイル

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_riichs_rx_if.h に記載しています。

2.6 整数型

このプロジェクトは 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。

2.7 コンパイル時の設定

本モジュールのコンフィギュレーションオプションの設定は、`r_riichs_rx_config.h`、`r_riichs_rx_pin_config.h`で行います。

オプション名および設定値に関する説明を、下表に示します。

Configuration options in <code>r_riichs_rx_config.h</code>	
RIICHS_CFG_PARAM_CHECKING_ENABLE ※デフォルト値は“1”	パラメータチェック処理をコードに含めるか選択できます。 “0”を選択すると、パラメータチェック処理をコードから省略できるため、コードサイズが削減できます。 “0”の場合、パラメータチェック処理をコードから省略します。 “1”の場合、パラメータチェック処理をコードに含めます。
RIICHS_CFG_CH0_INCLUDED (注 1) ※デフォルト値は“1”	該当チャネルを使用するかを選択できます。 該当チャネルを使用しない場合は“0”に設定してください。 “0”の場合、該当チャネルに関する処理をコードから省略します。 “1”の場合、該当チャネルに関する処理をコードに含めます。
RIICHS_CFG_PORT_SET_PROCESSING ※デフォルト値は“1”	R_RIICHS_CFG_RIICHS0_SCL0_PORT、 R_RIICHS_CFG_RIICHS0_SCL0_BIT、 R_RIICHS_CFG_RIICHS0_SDA0_PORT、 R_RIICHS_CFG_RIICHS0_SDA0_BIT で選択したポートを SCL、SDA 端子として使用するための設定処理をコードに含めるかを選択します。 “0”の場合、ポートの設定処理をコードから省略します。 “1”の場合、ポートの設定処理をコードに含めます。

注1. 該当チャネルをサポートしない対象デバイスでは本設定は無効です。

Configuration options in <code>r_riichs_rx_pin_config.h</code>	
R_RIICHS_CFG_RIICHS0_SCL0_PORT ※デフォルト値は'1'	SCL端子として使用するポートグループを選択します。 '0'~'J' (ASCIIコード)の範囲で設定してください。
R_RIICHS_CFG_RIICHS0_SCL0_BIT ※デフォルト値は'2'	SCL端子として使用する端子を選択します。 '0'~'7' (ASCIIコード)の範囲で設定してください。
R_RIICHS_CFG_RIICHS0_SDA0_PORT ※デフォルト値は'1'	SDA端子として使用するポートグループを選択します。 '0'~'J' (ASCIIコード)の範囲で設定してください。
R_RIICHS_CFG_RIICHS0_SDA0_BIT ※デフォルト値は'3'	SDA端子として使用する端子を選択します。 '0'~'7' (ASCIIコード)の範囲で設定してください。

2.8 コードサイズ

ROM (コードおよび定数) と RAM (グローバルデータ) のサイズは、ビルド時の「2.7 コンパイル時の設定」のコンフィギュレーションオプションによって決まります。掲載した値は、「2.3 サポートされているツールチェーン」の C コンパイラでコンパイルオプションがデフォルト時の参考値です。コンパイルオプションのデフォルトは最適化レベル：2、最適化のタイプ：サイズ優先、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やコンパイルオプションにより異なります。

下表の値は下記条件で確認しています。

モジュールリビジョン：r_riichs_rx rev1.00

コンパイラバージョン：Renesas Electronics C/C++ Compiler Package for RX Family V3.03.00

(統合開発環境のデフォルト設定に"-lang = c99"オプションを追加)

GCC for Renesas RX 8.03.00.202002

(統合開発環境のデフォルト設定に"-std=gnu99"オプションを追加)

IAR C/C++ Compiler for Renesas RX version 4.14.1

(統合開発環境のデフォルト設定)

コンフィギュレーションオプション：デフォルト設定

ROM、RAM およびスタックのコードサイズ								
デバイス	分類		使用メモリ					
			Renesas Compiler		GCC		IAR Compiler	
			パラメータ チェックあり	パラメータ チェックなし	パラメータ チェックあり	パラメータ チェックなし	パラメータ チェックあり	パラメータ チェックなし
RX671	ROM	1 チャンネル使用	10411 バイト	9936 バイト	19572 バイト	18804 バイト	14352 バイト	13698 バイト
	RAM	1 チャンネル使用	62 バイト		64 バイト		62 バイト	
	スタック (注1)		344 バイト		-		356 バイト	

注1. 割り込み関数の最大使用スタックサイズを含みます

2.9 引数

API 関数の引数である構造体を示します。この構造体は、API 関数のプロトタイプ宣言とともに `r_riic_rx_if.h` に記載されています。

構造体の内容は、通信中に参照、更新されます。このため、通信中(RIICHS_COMMUNICATION)に構造体の内容を書き換えしないでください。

```
typedef volatile struct
{
    uint8_t          rsv2;          /* 予約領域 */
    uint8_t          rsv1;          /* 予約領域 */
    riichs_ch_dev_status_t dev_sts; /* デバイス状態フラグ */
    uint8_t          ch_no;         /* 使用するデバイスのチャンネル番号 */
    riichs_callback_t callbackfunc; /* コールバック関数 */
    uint32_t          cnt2nd;        /* 2nd データカウンタ(バイト数) */
    uint32_t          cnt1st;        /* 1st データカウンタ(バイト数) */
    uint32_t *        p_data2nd;    /* 2nd データ格納バッファポインタ */
    uint32_t *        p_data1st;    /* 1st データ格納バッファポインタ */
    uint32_t *        p_slv_adr;    /* スレーブアドレスのバッファポインタ */
    double            scl_up_time;   /* SCLn の立ち上がり時間 */
    double            scl_down_time; /* SCLn の立ち下がり時間 */
    double            fs_scl_up_time; /* Hs モードに移行する前の SCLn の立ち上がり時間 */
    double            fs_scl_down_time; /* Hs モードに移行する前の SCLn の立ち下がり時間 */
    uint32_t          speed_kbps;    /* RIICHS の通信速度(kbps) */
    uint32_t          fs_speed_kbps; /* Hs モードに移行する前の RIICHS の通信速度(kbps) */
    uint32_t          bus_check_counter; /* ソフトウェアバスビジーチェックカウンタ */
    uint32_t          bus_free_time; /* ソフトウェアバスフリーカウンタ */
    uint16_t          slave_addr0;   /* スレーブアドレス 0 */
    uint16_t          slave_addr1;   /* スレーブアドレス 1 */
    uint16_t          slave_addr2;   /* スレーブアドレス 2 */
    riichs_addr_format_t slave_addr0_format; /* スレーブアドレス 0 のフォーマット */
    riichs_addr_format_t slave_addr1_format; /* スレーブアドレス 1 のフォーマット */
    riichs_addr_format_t slave_addr2_format; /* スレーブアドレス 2 のフォーマット */
    riichs_gca_t       gca_enable;    /* ジェネラルコールアドレス検出許可 */
    riichs_priority_t  rxi_priority;   /* RXI の割り込み優先レベル */
    riichs_priority_t  txi_priority;   /* TXI の割り込み優先レベル */
    riichs_priority_t  eei_priority;   /* EEI の割り込み優先レベル(TXI 及び RXI の割り込み優先レベル以上) */
    riichs_priority_t  tei_priority;   /* TEI の割り込み優先レベル(TXI 及び RXI の割り込み優先レベル以上) */
    riichs_master_arb_t master_arb;    /* マスタアービトレーションロスト検出許可 */
    riichs_filter_t     filter_stage;   /* デジタルノイズフィルタ段数 */
    riichs_timeout_t    timeout_enable; /* タイムアウト検出機能許可 */
    riichs_nack_detc_t  nack_detc_enable; /* NACK 検出許可 */
    riichs_arb_lost_t   arb_lost_enable; /* アービトレーションロスト検出許可 */
    riichs_counter_bit_t counter_bit;   /* タイムアウト検出時間選択 */
    riichs_low_count_t  l_count;        /* タイムアウト L カウント許可 */
    riichs_high_count_t h_count;        /* タイムアウト H カウント許可 */
    riichs_time_mode_t  timeout_mode;   /* タイムアウト検出モード選択 */
} riichs_info_t;
```

2.10 戻り値

API 関数の戻り値を示します。この列挙型は、API 関数のプロトタイプ宣言とともに `r_riic_rx_if.h` で記載されています。

```
typedef enum
{
    RIICHS_SUCCESS = 0U,           /* 関数の処理が正常に終了した場合 */
    RIICHS_ERR_LOCK_FUNC,         /* 他のモジュールで RIICHS が使用されている場合 */
    RIICHS_ERR_INVALID_CHAN,      /* 存在しないチャネルを指定した場合 */
    RIICHS_ERR_INVALID_ARG,       /* 不正な引数を設定した場合 */
    RIICHS_ERR_NO_INIT,           /* 未初期化状態の場合 */
    RIICHS_ERR_BUS_BUSY,          /* バスビジーの場合 */
    RIICHS_ERR_AL,                /* アービトレーションロスト検出状態で関数を呼び出した場合 */
    RIICHS_ERR_TMO,               /* タイムアウトを検出した場合 */
    RIICHS_ERR_OTHER              /* その他エラー */
} riichs_return_t;
```

2.11 コールバック関数

本モジュールでは、以下のいずれかの条件を満たし EEI 割り込み要求が発生したときに、ユーザが設定したコールバック関数を呼び出します。

- (1) 通信動作（マスタ送信、マスタ受信、マスタ送受信、スレーブ送信、スレーブ受信）が完了し、ストップコンディションを発行した。
- (2) 通信動作（マスタ送信、マスタ受信、マスタ送受信、スレーブ送信、スレーブ受信）中にタイムアウトを検出した。(注 1)

注1. タイムアウト検出機能の有効/無効の設定方法については、「2.9 引数」を参照ください。

コールバック関数は、「2.9 引数」に記載された構造体メンバ “callbackfunc” に、コールバック関数のアドレスを格納し、R_RIICHS_MasterSend 関数、R_RIICHS_MasterReceive 関数、R_RIICHS_SlaveTransfer 関数を呼び出したときに設定されます。

コールバック関数内では R_RIICHS_GetStatus 関数以外の API 関数の呼び出しは禁止です。

2.12 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、スマート・コンフィグレータを使用した(1)、(3)、(5)の追加方法を推奨しています。ただし、スマート・コンフィグレータは、一部の RX デバイスのみサポートしています。サポートされていない RX デバイスについては(2)、(4)の方法を使用してください。

- (1) e2 studio 上でスマート・コンフィグレータを使用して FIT モジュールを追加する場合 e2 studio のスマート・コンフィグレータを使用して、自動的にユーザプロジェクトに FIT モジュールを追加します。詳細は、アプリケーションノート「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 上で FIT コンフィグレータを使用して FIT モジュールを追加する場合 e2 studio の FIT コンフィグレータを使用して、自動的にユーザプロジェクトに FIT モジュールを追加することができます。詳細は、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)」を参。
- (3) CS+上でスマート・コンフィグレータを使用して FIT モジュールを追加する場合 CS+上で、スタンドアロン版スマート・コンフィグレータを使用して、自動的にユーザプロジェクトに FIT モジュールを追加します。詳細は、アプリケーションノート「RX スマート・コンフィグレータ ユーザーガイド: CS+編 (R20AN0470)」を参照してください。
- (4) CS+上で FIT モジュールを追加する場合 CS+上で、手動でユーザプロジェクトに FIT モジュールを追加します。詳細は、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)」を参照してください。
- (5) IAREW 上でスマート・コンフィグレータを使用して FIT モジュールを追加する場合スタンドアロン版スマート・コンフィグレータを使用して、自動的にユーザプロジェクトに FIT モジュールを追加します。詳細は、アプリケーションノート「RX スマート・コンフィグレータ ユーザーガイド: IAREW 編 (R20AN0535)」を参照してください。

2.13 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)
{
    /* The delay period needed is to make sure that the PLL has stabilized. */
}
```

for 文の例 :

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

do while 文の例 :

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

3. API 関数

R_RIICHS_Open()

この関数は RIICHS FIT モジュールを初期化する関数です。この関数は他の API 関数を使用する前に呼び出される必要があります。

Format

```
riichs_return_t R_RIICHS_Open(
    riichs_info_t * p_riichs_info /* 構造体データ */
)
```

Parameters

**p_riichs_info*

I²C 通信情報構造体のポインタ。

この構造体のうち、本関数で使用するメンバのみを以下に示します。この構造体の詳細については2.9を参照してください。

構造体の内容は、通信中に参照、更新されます。このため、通信中(RIICHS_COMMUNICATION)に構造体の内容を書き換えしないでください。

下記のうち、API 実行中に値が更新される引数には、“更新あり”と記載しています。

riichs_ch_dev_status_t	dev_sts;	/* デバイス状態フラグポインタ (更新あり) */
uint8_t	ch_no;	/* チャンネル番号 */
double	scl_up_time;	/* SCLn の立ち上がり時間 */
double	scl_down_time;	/* SCLn の立ち下がり時間 */
double	fs_scl_up_time;	/* Hs モードに移行する前の SCLn の立ち上がり時間 */
double	fs_scl_down_time;	/* Hs モードに移行する前の SCLn の立ち下がり時間 */
uint32_t	speed_kbps;	/* RIICHS の通信速度(kbps) */
uint32_t	fs_speed_kbps;	/* Hs モードに移行する前の RIICHS の通信速度(kbps) */
uint32_t	bus_check_counter;	/* ソフトウェアバスビジーチェックカウンタ */
uint32_t	bus_free_time;	/* ソフトウェアバスフリーカウンタ */
uint16_t	slave_addr0;	/* スレーブアドレス 0 */
uint16_t	slave_addr1;	/* スレーブアドレス 1 */
uint16_t	slave_addr2;	/* スレーブアドレス 2 */
riichs_addr_format_t	slave_addr0_format;	/* スレーブアドレス 0 のフォーマット */
riichs_addr_format_t	slave_addr1_format;	/* スレーブアドレス 1 のフォーマット */
riichs_addr_format_t	slave_addr2_format;	/* スレーブアドレス 2 のフォーマット */
riichs_gca_t	gca_enable;	/* ジェネラルコールアドレス検出許可 */
riichs_priority_t	rx_i_priority;	/* RXI の割り込み優先レベル */
riichs_priority_t	tx_i_priority;	/* TXI の割り込み優先レベル */
riichs_priority_t	eei_priority;	/* EEI の割り込み優先レベル(TXI 及び RXI の割り込み優先レベル以上) */
riichs_priority_t	tei_priority;	/* TEI の割り込み優先レベル(TXI 及び RXI の割り込み優先レベル以上) */
riichs_master_arb_t	master_arb;	/* マスタアービトレーションロスト検出許可 */
riichs_filter_t	filter_stage;	/* デジタルノイズフィルタ段数 */
riichs_timeout_t	timeout_enable;	/* タイムアウト検出機能許可 */
riichs_nack_detc_t	nack_detc_enable;	/* NACK 検出許可 */
riichs_arb_lost_t	arb_lost_enable;	/* アービトレーションロスト検出許可 */
riichs_counter_bit_t	counter_bit;	/* タイムアウト検出時間選択 */
riichs_low_count_t	l_count;	/* タイムアウト L カウント許可 */
riichs_high_count_t	h_count;	/* タイムアウト H カウント許可 */
riichs_time_mode_t	timeout_mode;	/* タイムアウト検出モード選択 */

Return Values

RIICHS_SUCCESS /* 問題なく処理が完了した場合 */
RIICHS_ERR_LOCK_FUNC /* 他のタスクがAPI をロックしている場合 */
RIICHS_ERR_INVALID_CHAN /* 存在しないチャンネルの場合 */
RIICHS_ERR_INVALID_ARG /* 不正な引数の場合 */
RIICHS_ERR_OTHER /* 現在の状態に該当しない不正なイベントが発生した場合 */

Properties

`r_riichs_rx_if.h` にプロトタイプ宣言されています。

Description

RIICHSの通信を開始するための初期設定をします。引数で指定したRIICHSのチャンネルを設定します。チャンネルの状態が“未初期化状態 (RIICHS_NO_INIT)”の場合、次の処理を行います。

- 状態フラグの設定
- ポートの入出力設定
- I²C 出力ポートの割り当て
- RIICHS のモジュールストップ状態の解除
- API で使用する変数の初期化
- RIICHS 通信で使用する RIICHS レジスタの初期化
- RIICHS 割り込みの禁止

Example

```
volatile riichs_return_t  ret;
riichs_info_t             iichs_info_m;

iichs_info_m.dev_sts = RIICHS_NO_INIT;
iichs_info_m.ch_no   = 0;
iichs_info_m.scl_up_time = 20E-9;
iichs_info_m.scl_down_time = 20E-9;
iichs_info_m.fs_scl_up_time = 20E-9;
iichs_info_m.fs_scl_down_time = 20E-9;
iichs_info_m.speed_kbps = 3400;
iichs_info_m.fs_speed_kbps = 400;
iichs_info_m.bus_check_counter = 1000;
iichs_info_m.bus_free_time = 5;
iichs_info_m.slave_addr0 = 0x0025;
iichs_info_m.slave_addr1 = 0x0000;
iichs_info_m.slave_addr2 = 0x0000;
iichs_info_m.slave_addr0_format = RIICHS_SEVEN_BIT_ADDR_FORMAT;
iichs_info_m.slave_addr1_format = RIICHS_ADDR_FORMAT_NONE;
iichs_info_m.slave_addr2_format = RIICHS_ADDR_FORMAT_NONE;
iichs_info_m.gca_enable = RIICHS_GCA_DISABLE;
iichs_info_m.rxi_priority = RIICHS_IPL_1;
iichs_info_m.txi_priority = RIICHS_IPL_1;
iichs_info_m.eei_priority = RIICHS_IPL_1;
iichs_info_m.tei_priority = RIICHS_IPL_1;
iichs_info_m.master_arb = RIICHS_MASTER_ARB_LOST_DISABLE;
iichs_info_m.filter_stage = RIICHS_DIGITAL_FILTER_0;
iichs_info_m.timeout_enable = RIICHS_TMO_ENABLE;
iichs_info_m.nack_detc_enable = RIICHS_NACK_DETC_ENABLE;
iichs_info_m.arb_lost_enable = RIICHS_ARB_LOST_ENABLE;
iichs_info_m.counter_bit = RIICHS_COUNTER_BIT16;
iichs_info_m.l_count = RIICHS_L_COUNT_ENABLE;
iichs_info_m.h_count = RIICHS_H_COUNT_ENABLE;
iichs_info_m.timeout_mode = RIICHS_TIMEOUT_MODE_ALL;
```

```
ret = R_RIICHS_Open(&iichs_info_m);
```

Special Notes:

引数の設定可能範囲は、下表を参照してください。

構造体メンバ	ユーザ設定可能範囲
scl_up_time	SCLn の立ち上がり時間を設定してください。
scl_down_time	SCLn の立ち下がり時間を設定してください。
fs_scl_up_time	Hs モードに移行する前の SCLn の立ち上がり時間を設定してください。
fs_scl_down_time	Hs モードに移行する前の SCLn の立ち下がり時間を設定してください。
speed_kbps	1~3400
fs_speed_kbps	0~1000 (注 1)
bus_check_counter	0000 0000h ~ 0000 ffffh
bus_free_time	0000 0000h ~ 0000 01ffh
slave_addr0	スレーブアドレス 0 を設定してください。(注 2)
slave_addr1	スレーブアドレス 1 を設定してください。(注 2)
slave_addr2	スレーブアドレス 2 を設定してください。(注 2)
slave_addr0_format	RIICHS_ADDR_FORMAT_NONE(スレーブアドレスの設定値は無効)、 RIICHS_SEVEN_BIT_ADDR_FORMAT(スレーブアドレスの設定値は設定値の下位 7 ビット) または RIICHS_TEN_BIT_ADDR_FORMAT(スレーブアドレスの設定値は設定値の下位 10 ビット)
slave_addr1_format	RIICHS_ADDR_FORMAT_NONE(スレーブアドレスの設定値は無効)、 RIICHS_SEVEN_BIT_ADDR_FORMAT(スレーブアドレスの設定値は設定値の下位 7 ビット) または RIICHS_TEN_BIT_ADDR_FORMAT(スレーブアドレスの設定値は設定値の下位 10 ビット)
slave_addr2_format	RIICHS_ADDR_FORMAT_NONE(スレーブアドレスの設定値は無効)、 RIICHS_SEVEN_BIT_ADDR_FORMAT(スレーブアドレスの設定値は設定値の下位 7 ビット) または RIICHS_TEN_BIT_ADDR_FORMAT(スレーブアドレスの設定値は設定値の下位 10 ビット)
gca_enable	RIICHS_GCA_ENABLE または RIICHS_GCA_DISABLE
rx_i_priority	RIICHS_IPL_1 ~ RIICHS_IPL_15
tx_i_priority	RIICHS_IPL_1 ~ RIICHS_IPL_15
eei_priority	RIICHS_IPL_1 ~ RIICHS_IPL_15 (注 3)
tei_priority	RIICHS_IPL_1 ~ RIICHS_IPL_15 (注 3)
master_arb	RIICHS_MASTER_ARB_LOST_ENABLE または RIICHS_MASTER_ARB_LOST_DISABLE
filter_stage	RIICHS_DIGITAL_FILTER_0 ~ RIICHS_DIGITAL_FILTER_16
timeout_enable	RIICHS_TMO_ENABLE または RIICHS_TMO_DISABLE
nack_detc_enable	RIICHS_NACK_DETC_ENABLE または RIICHS_NACK_DETC_DISABLE
arb_lost_enable	RIICHS_ARB_LOST_ENABLE または RIICHS_ARB_LOST_DISABLE
counter_bit	RIICHS_COUNTER_BIT6、RIICHS_COUNTER_BIT8、 RIICHS_COUNTER_BIT14、または RIICHS_COUNTER_BIT16
l_count	RIICHS_L_COUNT_ENABLE または RIICHS_L_COUNT_DISABLE
h_count	RIICHS_H_COUNT_ENABLE または RIICHS_H_COUNT_DISABLE
timeout_mode	RIICHS_TIMEOUT_MODE_ALL、RIICHS_TIMEOUT_MODE_BUSY、または RIICHS_TIMEOUT_MODE_FREE

注 1 : speed_kbps で"1000"を超える値を指定した場合、"1"以上を指定してください。

注 2 : slave_addr0_format~slave_addr2_format で設定したフォーマットによって設定可能範囲が変わります。

注 3 : rxi_priority 及び txi_priority で指定した優先レベルの値より低い値を設定しないでください。

R_RIICHS_MasterSend()

マスタ送信を開始します。引数に合わせてマスタのデータ送信パターンを変更します。ストップコンディション生成まで一括で実施します。

Format

```
riichs_return_t R_RIICHS_MasterSend(
    riichs_info_t * p_riichs_info /* 構造体データ */
)
```

Parameters

**p_riichs_info*

I²C 通信情報構造体のポインタ。引数によって、送信パターン(4パターンあります)を変更できます。各送信パターンの指定方法および引数の設定可能範囲は、「Special Notes」を参照ください。また、送信パターンの波形のイメージは「1.3.2 マスタ送信の処理」を参照ください。

この構造体のうち、本関数で使用するメンバのみを以下に示します。この構造体の詳細については2.9を参照してください。

構造体の内容は、通信中に参照、更新されます。このため、通信中(RIICHS_COMMUNICATION)に構造体の内容を書き換えしないでください。

スレーブアドレスを設定する際、1ビット左シフトせずに格納してください。

下記のうち、API 実行中に値が更新される引数には、“更新あり”と記載しています。

riichs_ch_dev_status_t	dev_sts;	/* デバイス状態フラグ(更新あり) */
uint8_t	ch_no;	/* チャンネル番号 */
riichs_callback	callbackfunc;	/* コールバック関数 */
uint32_t	cnt2nd;	/* 2nd データカウンタ(バイト数) (パターン1、2のみ更新あり) */
uint32_t	cnt1st;	/* 1st データカウンタ(バイト数) (パターン1のみ更新あり) */
uint32_t *	p_data2nd;	/* 2nd データ格納バッファポインタ */
uint32_t *	p_data1st;	/* 1st データ格納バッファポインタ */
uint32_t *	p_slv_adr;	/* スレーブアドレスのバッファポインタ */

Return Values

RIICHS_SUCCESS	/* 問題なく処理が完了した場合 */
RIICHS_ERR_INVALID_CHAN	/* 存在しないチャンネルの場合 */
RIICHS_ERR_INVALID_ARG	/* 不正な引数の場合 */
RIICHS_ERR_NO_INIT	/* 初期設定ができていない場合 (未初期化状態) */
RIICHS_ERR_BUS_BUSY	/* バスビジーの場合 */
RIICHS_ERR_AL	/* アービトレーションエラーが発生した場合 */
RIICHS_ERR_TMO	/* タイムアウトを検出した場合 */
RIICHS_ERR_OTHER	/* 現在の状態に該当しない不正なイベントが発生した場合 */

Properties

r_riichs_rx_if.h にプロトタイプ宣言されています。

Description

RIICHS のマスタ送信を開始します。引数で指定した RIICHS のチャンネル、送信パターンで送信します。チャンネルの状態が“アイドル状態”(RIICHS_IDLE、RIICHS_FINISH、RIICHS_NACK)の場合、次の処理を行います。

- 状態フラグの設定
- API で使用する変数の初期化
- RIICHS 割り込みの許可

ー スタートコンディションの生成

スタートコンディションの生成処理までが正常に終了した時、本関数は戻り値として RIICHS_SUCCESS を返します。

スタートコンディションの生成時に下記条件に該当した時、本関数は戻り値として RIICHS_ERR_BUS_BUSY を返します。(注 1)

- 内部のステータスビットが BUSY 状態である
- SCL、SDA ラインのいずれかが Low の状態である

送信の処理は、本関数が RIICHS_SUCCESS を返した後発生する割り込み処理の中で順次行われます。使用する割り込みは、「2.4 使用する割り込みベクタ」を参照ください。マスタ送信の割り込みの発生タイミングは、「6.2.1 マスタ送信」を参照ください。

送信終了でストップコンディションを発行した後、引数で指定したコールバック関数が呼び出されます。

送信が正常に完了したかどうかは、引数で指定したデバイス状態フラグ、またはチャネル状態フラグ g_riichs_ChStatus[]が"RIICHS_FINISH"になっているかどうかで確認することができます。

注1. SCL と SDA 端子が外部回路でプルアップされていない場合、SCL、SDA ラインのいずれかを Low の状態として検出し、RIICHS_ERR_BUS_BUSY を返すことがあります。

Example

```
/* for MasterSend(Pattern 1) */
#include <stddef.h>
#include "platform.h"
#include "r_riichs_rx_if.h"

riichs_info_t iichs_info_m;

void CallbackMaster(void);
void main(void);

void main(void)
{
    volatile riichs_return_t ret;

    uint8_t addr_eeprom[1]    = {0x53};
    uint8_t access_addr1[1]   = {0x00};
    uint8_t mst_send_data[5]  = {0x81,0x82,0x83,0x84,0x85};

    /* Sets IIC Information for sending pattern 1. */
    iichs_info_m.dev_sts = RIICHS_NO_INIT;
    iichs_info_m.ch_no = 0;
    iichs_info_m.callbackfunc = &CallbackMaster;
    iichs_info_m.cnt2nd = 3;
    iichs_info_m.cnt1st = 1;
    iichs_info_m.p_data2nd = mst_send_data;
    iichs_info_m.p_data1st = access_addr1;
    iichs_info_m.p_slv_adr = addr_eeprom;
    iichs_info_m.scl_up_time = 20E-9;
    iichs_info_m.scl_down_time = 20E-9;
    iichs_info_m.fs_scl_up_time = 20E-9;
    iichs_info_m.fs_scl_down_time = 20E-9;
    iichs_info_m.speed_kbps = 3400;
    iichs_info_m.fs_speed_kbps = 400;
    iichs_info_m.bus_check_counter = 1000;
    iichs_info_m.bus_free_time = 5;
```

```
iichs_info_m.slave_addr0 = 0x0025;
iichs_info_m.slave_addr1 = 0x0000;
iichs_info_m.slave_addr2 = 0x0000;
iichs_info_m.slave_addr0_format = RIICHS_SEVEN_BIT_ADDR_FORMAT;
iichs_info_m.slave_addr1_format = RIICHS_ADDR_FORMAT_NONE;
iichs_info_m.slave_addr2_format = RIICHS_ADDR_FORMAT_NONE;
iichs_info_m.gca_enable = RIICHS_GCA_DISABLE;
iichs_info_m.rxi_priority = RIICHS_IPL_1;
iichs_info_m.txi_priority = RIICHS_IPL_1;
iichs_info_m.eei_priority = RIICHS_IPL_1;
iichs_info_m.tei_priority = RIICHS_IPL_1;
iichs_info_m.master_arb = RIICHS_MASTER_ARB_LOST_DISABLE;
iichs_info_m.filter_stage = RIICHS_DIGITAL_FILTER_0;
iichs_info_m.timeout_enable = RIICHS_TMO_ENABLE;
iichs_info_m.nack_detc_enable = RIICHS_NACK_DETC_ENABLE;
iichs_info_m.arb_lost_enable = RIICHS_ARB_LOST_ENABLE;
iichs_info_m.counter_bit = RIICHS_COUNTER_BIT16;
iichs_info_m.l_count = RIICHS_L_COUNT_ENABLE;
iichs_info_m.h_count = RIICHS_H_COUNT_ENABLE;
iichs_info_m.timeout_mode = RIICHS_TIMEOUT_MODE_ALL;
/* RIICHS open */
ret = R_RIICHS_Open(&iichs_info_m);

/* RIICHS send start */
ret = R_RIICHS_MasterSend(&iichs_info_m);

if (RIICHS_SUCCESS == ret)
{
    while(RIICHS_FINISH != iichs_info_m.dev_sts);
}
else
{
    /* error */
}

/* RIICHS send complete */
while(1);
}

void CallbackMaster(void)
{
    volatile riichs_return_t ret;
    riichs_mcu_status_t iichs_status;

    ret = R_RIICHS_GetStatus(&iichs_info_m, &iichs_status);
    if(RIICHS_SUCCESS != ret)
    {
        /* R_RIICHS_GetStatus 関数のエラー処理 */
    }
    else
    {
        /* iichs_status のステータスフラグを確認して
           タイムアウト、アービトレーションロスト、NACK
           などが検出されていた場合の処理を記述 */
    }
}
```

Special Notes:

送信パターンごとの引数の設定可能範囲は、下表を参照してください。

構造体メンバ	ユーザ設定可能範囲		
	マスタ送信 パターン 1	マスタ送信 パターン 2	マスタ送信 パターン 3
*p_slv_adr	スレーブアドレスバッファポインタ	スレーブアドレスバッファ ポインタ	スレーブアドレスバッ ファポインタ
*p_data1st	[送信用]1st データバッファポインタ	FIT_NO_PTR (注 1)	FIT_NO_PTR (注 1)
*p_data2nd	[送信用]2nd データバッファポインタ	[送信用]2nd データ バッファポインタ	FIT_NO_PTR (注 1)
cnt1st	0000 0001h~FFFF FFFFh(注 2)	0	0
cnt2nd	0000 0001h~FFFF FFFFh(注 2)	0000 0001h~ FFFFFFFFh(注 2)	0
callbackfunc	使用する関数名を指定してください。	使用する関数名を指定して ください。	使用する関数名を指定し てください。
ch_no	00h~FFh	00h~FFh	00h~FFh
dev_sts	デバイス状態フラグ	デバイス状態ラグ	デバイス状態フラグ
rsv1,rsv2	予約領域(設定無効)	予約領域(設定無効)	予約領域(設定無効)

注 1：パターン 2、パターン 3 を使用する場合は、上表のとおり該当の構造体メンバに“FIT_NO_PTR”を入れてください。

注 2：“0” は設定しないでください。

R_RIICHS_MasterReceive()

マスタ受信を開始します。引数に合わせてマスタのデータ受信パターンを変更します。ストップコンディション生成まで一括で実施します。

Format

```
riichs_return_t R_RIICHS_MasterReceive(
    riichs_info_t * p_riichs_info /* 構造体データ */
)
```

Parameters

**p_riichs_info*

I²C 通信情報構造体のポインタ。引数の設定によって、マスタ受信かマスタ送受信を選択できます。マスタ受信およびマスタ送受信の指定方法と引数の設定可能範囲は「Special Notes」を参照ください。また、受信パターンの波形イメージは「1.3.3マスタ受信の処理」を参照ください。

この構造体のうち、本関数で使用するメンバのみを以下に示します。この構造体の詳細については2.9を参照してください。

構造体の内容は、通信中に参照、更新されます。このため、通信中(RIICHS_COMMUNICATION)に構造体の内容を書き換えしないでください。

スレーブアドレスを設定する際、1ビット左シフトせずに格納してください。

下記のうち、API 実行中に値が更新される引数には、“更新あり”と記載しています。

riichs_ch_dev_status_t	dev_sts;	/* デバイス状態フラグ(更新あり) */
uint8_t	ch_no;	/* チャンネル番号 */
riichs_callback	callbackfunc;	/* コールバック関数 */
uint32_t	cnt2nd;	/* 2nd データカウンタ(バイト数) (更新あり) */
uint32_t	cnt1st;	/* 1st データカウンタ(バイト数) (マスタ送受信のみ更新あり) */
uint32_t *	p_data2nd;	/* 2nd データ格納バッファポインタ */
uint32_t *	p_data1st;	/* 1st データ格納バッファポインタ */
uint32_t *	p_slv_adr;	/* スレーブアドレスのバッファポインタ */

Return Values

RIICHS_SUCCESS	/* 問題なく処理が完了した場合 */
RIICHS_ERR_INVALID_CHAN	/* 存在しないチャンネルの場合 */
RIICHS_ERR_INVALID_ARG	/* 不正な引数の場合 */
RIICHS_ERR_NO_INIT	/* 初期設定ができていない場合 (未初期化状態) */
RIICHS_ERR_BUS_BUSY	/* バスビジーの場合 */
RIICHS_ERR_AL	/* アービトレーションエラーが発生した場合 */
RIICHS_ERR_TMO	/* タイムアウトを検出した場合 */
RIICHS_ERR_OTHER	/* 現在の状態に該当しない不正なイベントが発生した場合 */

Properties

r_riichs_rx_if.h にプロトタイプ宣言されています。

Description

RIICHS のマスタ受信を開始します。引数で指定した RIICHS のチャンネル、受信パターンで受信します。チャンネルの状態が“アイドル状態” (RIICHS_IDLE、RIICHS_FINISH、RIICHS_NACK) の場合、次の処理を行います。

- － 状態フラグの設定
- － API で使用する変数の初期化
- － RIICHS 割り込みの許可
- － スタートコンディションの生成

スタートコンディションの生成処理までが正常に終了した時、本関数は戻り値として RIICHS_SUCCESS を返します。

スタートコンディションの生成時に下記条件に該当した時、本関数は戻り値として RIICHS_ERR_BUS_BUSY を返します。(注 1)

- 内部のステータスビットが BUSY 状態である
- SCL、SDA ラインのいずれかが Low の状態である

受信の処理は、本関数が RIICHS_SUCCESS を返した後発生する割り込み処理の中で順次行われます。

使用する割り込みは、「2.4 使用する割り込みベクタ」を参照ください。

マスタ受信の割り込みの発生タイミングは、「6.2.2 マスタ受信」を参照ください。

受信終了でストップコンディションを発行した後、引数で指定したコールバック関数が呼び出されます。

受信が正常に完了したかどうかは、引数で指定したデバイス状態フラグ、またはチャネル状態フラグ g_riichs_ChStatus[]が"RIICHS_FINISH"になっているかどうかで確認することができます。

注1. SCL と SDA 端子が外部回路でプルアップされていない場合、SCL、SDA ラインのいずれかを Low の状態として検出し、RIICHS_ERR_BUS_BUSY を返すことがあります。

Example

```
#include <stddef.h>
#include "platform.h"
#include "r_riichs_rx_if.h"

riichs_info_t    iichs_info_m;

void CallbackMaster(void);
void main(void);

void main(void)
{
    volatile riichs_return_t ret;

    uint8_t addr_eeprom[1]    = {0x53};
    uint8_t access_addr1[1]   = {0x00};
    uint8_t mst_store_area[5] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF};

    /* Sets IICHS Information. */
    iichs_info_m.dev_sts = RIICHS_NO_INIT;
    iichs_info_m.ch_no = 0;
    iichs_info_m.callbackfunc = &CallbackMaster;
    iichs_info_m.cnt2nd = 3;
    iichs_info_m.cnt1st = 1;
    iichs_info_m.p_data2nd = mst_store_area;
    iichs_info_m.p_data1st = access_addr1;
    iichs_info_m.p_slv_addr = addr_eeprom;
    iichs_info_m.scl_up_time = 20E-9;
    iichs_info_m.scl_down_time = 20E-9;
    iichs_info_m.fs_scl_up_time = 20E-9;
    iichs_info_m.fs_scl_down_time = 20E-9;
    iichs_info_m.speed_kbps = 3400;
    iichs_info_m.fs_speed_kbps = 400;
    iichs_info_m.bus_check_counter = 1000;
    iichs_info_m.bus_free_time = 5;
    iichs_info_m.slave_addr0 = 0x0025;
    iichs_info_m.slave_addr1 = 0x0000;
```

```
iichs_info_m.slave_addr2 = 0x0000;
iichs_info_m.slave_addr0_format = RIICHS_SEVEN_BIT_ADDR_FORMAT;
iichs_info_m.slave_addr1_format = RIICHS_ADDR_FORMAT_NONE;
iichs_info_m.slave_addr2_format = RIICHS_ADDR_FORMAT_NONE;
iichs_info_m.gca_enable = RIICHS_GCA_DISABLE;
iichs_info_m.rxi_priority = RIICHS_IPL_1;
iichs_info_m.txi_priority = RIICHS_IPL_1;
iichs_info_m.eei_priority = RIICHS_IPL_1;
iichs_info_m.tei_priority = RIICHS_IPL_1;
iichs_info_m.master_arb = RIICHS_MASTER_ARB_LOST_DISABLE;
iichs_info_m.filter_stage = RIICHS_DIGITAL_FILTER_0;
iichs_info_m.timeout_enable = RIICHS_TMO_ENABLE;
iichs_info_m.nack_detc_enable = RIICHS_NACK_DETC_ENABLE;
iichs_info_m.arb_lost_enable = RIICHS_ARB_LOST_ENABLE;
iichs_info_m.counter_bit = RIICHS_COUNTER_BIT16;
iichs_info_m.l_count = RIICHS_L_COUNT_ENABLE;
iichs_info_m.h_count = RIICHS_H_COUNT_ENABLE;
iichs_info_m.timeout_mode = RIICHS_TIMEOUT_MODE_ALL;

/* RIICHS open */
ret = R_RIICHS_Open(&iichs_info_m);

/* RIICHS receive start */
ret = R_RIICHS_MasterReceive(&iichs_info_m);

if (RIICHS_SUCCESS == ret)
{
    while(RIICHS_FINISH != iichs_info_m.dev_sts);
}
else
{
    /* error */
}

/* RIICHS receive complete */
while(1);
}

void CallbackMaster(void)
{
    volatile riichs_return_t ret;
    riichs_mcu_status_t      iichs_status;

    ret = R_RIICHS_GetStatus(&iichs_info_m, &iichs_status);
    if(RIICHS_SUCCESS != ret)
    {
        /* R_RIICHS_GetStatus 関数のエラー処理 */
    }
    else
    {
        /* iichs_status のステータスフラグを確認して
           タイムアウト、アービトレーションロスト、NACK
           などが検出されていた場合の処理を記述 */
    }
}
```

Special Notes:

受信パターンごとの引数の設定可能範囲は、下表を参照してください。

構造体メンバ	ユーザ設定可能範囲	
	マスタ受信	マスタ送受信
*p_slv_adr	スレーブアドレスバッファポインタ	スレーブアドレスバッファポインタ
*p_data1st	未使用(設定無効)	[送信用]1st データバッファポインタ
*p_data2nd	[受信用]2nd データバッファポインタ	[受信用]2nd データバッファポインタ
dev_sts	デバイス状態フラグ	デバイス状態フラグ
cnt1st(注 1)	0	0000 0001h~FFFF FFFFh
cnt2nd	0000 0001h~FFFF FFFFh(注 2)	0000 0001h~FFFF FFFFh(注 2)
callbackfunc	使用する関数名を指定してください。	使用する関数名を指定してください。
ch_no	00h~FFh	00h~FFh
rsv1,rsv2	予約領域(設定無効)	予約領域(設定無効)

注 1 : 1st データが “0” か “0 以外” かで受信パターンが決まります。

注 2 : “0” は設定しないでください。

R_RIICHS_SlaveTransfer()

スレーブ送受信を行います。引数のパターンに合わせてデータ送受信パターンを変更します。

Format

```
riichs_return_t R_RIICHS_SlaveTransfer (
    riichs_info_t *   p_riichs_info   /* 構造体データ */
)
```

Parameters

**p_riichs_info*

I²C 通信情報構造体のポインタ。引数の設定によって、スレーブ受信許可状態かスレーブ送信許可状態、またはその両方を選択できます。引数の設定可能範囲は「Special Notes」を参照ください。また、受信パターンの波形イメージは「図 1.9スレーブ受信 信号図」を、送信パターンの波形イメージは「図 1.11スレーブ送信 信号図」を参照ください。

この構造体のうち、本関数で使用するメンバのみを以下に示します。この構造体の詳細については2.9を参照してください。

構造体の内容は、通信中に参照、更新されます。このため、通信中(RIICHS_COMMUNICATION)に構造体の内容を書き換えしないでください。

下記のうち、API 実行中に値が更新される引数には、“更新あり”と記載しています。

riichs_ch_dev_status_t	dev_sts;	/* デバイス状態フラグ(更新あり) */
uint8_t	ch_no;	/* チャンネル番号 */
riichs_callback	callbackfunc;	/* コールバック関数 */
uint32_t	cnt2nd;	/* 2nd データカウンタ(バイト数) (スレーブ受信時のみ更新あり) */
uint32_t	cnt1st;	/* 1st データカウンタ(バイト数) (スレーブ送信時のみ更新あり) */
uint32_t *	p_data2nd;	/* 2nd データ格納バッファポインタ */
uint32_t *	p_data1st;	/* 1st データ格納バッファポインタ */

Return Values

RIICHS_SUCCESS	/* 問題なく処理が完了した場合 */
RIICHS_ERR_INVALID_CHAN	/* 存在しないチャンネルの場合 */
RIICHS_ERR_INVALID_ARG	/* 不正な引数の場合 */
RIICHS_ERR_NO_INIT	/* 初期設定ができていない場合 (未初期化状態) */
RIICHS_ERR_BUS_BUSY	/* バスビジーの場合 */
RIICHS_ERR_AL	/* アービトレーションエラーが発生した場合 */
RIICHS_ERR_TMO	/* タイムアウトを検出した場合 */
RIICHS_ERR_OTHER	/* 現在の状態に該当しない不正なイベントが発生した場合 */

Properties

r_riichs_rx_if.h にプロトタイプ宣言されています。

Description

RIICHS のスレーブ送信、またはスレーブ受信できる状態にします。マスタ通信中に本関数を呼び出した場合は、エラーとなります。引数で指定した RIICHS のチャンネルを設定します。チャンネルの状態が“アイドル状態 (RIICHS_IDLE、RIICHS_FINISH、RIICHS_NACK)”の場合、次の処理を行います。

- － 状態フラグの設定
- － API で使用する変数の初期化
- － RIICHS 通信で使用する RIICHS レジスタの初期化
- － RIICHS 割り込みの許可

ー スレーブアドレスの設定、スレーブアドレス一致割り込みの許可

スレーブアドレスの設定、スレーブアドレス一致割り込みの許可までが正常に終了した時、本関数は戻り値として RIICHS_SUCCESS を返します。

スレーブ送信、またはスレーブ受信の処理は、その後発生する割り込み処理の中で順次行われます。

使用する割り込みは、「2.4 使用する割り込みベクタ」を参照ください。

スレーブ送信の割り込みの発生タイミングは、「6.2.4 スレーブ送信」を参照ください。スレーブ受信の割り込みの発生タイミングは、「6.2.5 スレーブ受信」を参照ください。

スレーブ送信、またはスレーブ受信終了のストップコンディションを検出した後、引数で指定したコールバック関数が呼び出されます。

スレーブ受信が正常に完了したかどうかは、引数で指定したデバイス状態フラグ、またはチャネル状態フラグ g_riichs_ChStatus[]が"RIICHS_FINISH"になっているかどうかで確認することができます。スレーブ送信が正常に完了したかどうかは、引数で指定したデバイス状態フラグ、またはチャネル状態フラグ g_riichs_ChStatus[]が"RIICHS_FINISH"もしくは"RIICHS_NACK"になっているかどうかで確認することができます。マスタデバイスが最後の受信完了を NACK で通知する場合、"RIICHS_NACK"になります。

Example

```
#include <stddef.h>
#include "platform.h"
#include "r_riichs_rx_if.h"

riichs_info_t    iichs_info_m;

void CallbackMaster(void);
void CallbackSlave(void);
void main(void);

void main(void)
{
    volatile    riichs_return_t ret;
    riichs_info_t iichs_info_s;

    uint8_t addr_eeprom[1]    = {0x25};
    uint8_t access_addr1[1]   = {0x00};
    uint8_t mst_send_data[5]  = {0x81,0x82,0x83,0x84,0x85};
    uint8_t slv_send_data[5]  = {0x71,0x72,0x73,0x74,0x75};
    uint8_t mst_store_area[5] = {0xFF,0xFF,0xFF,0xFF,0xFF};
    uint8_t slv_store_area[5] = {0xFF,0xFF,0xFF,0xFF,0xFF};

    /* Sets IICHS Information for Master Send. */
    iichs_info_m.dev_sts = RIICHS_NO_INIT;
    iichs_info_m.ch_no = 0;
    iichs_info_m.callbackfunc = &CallbackMaster;
    iichs_info_m.cnt2nd = 3;
    iichs_info_m.cnt1st = 1;
    iichs_info_m.p_data2nd = mst_store_area;
    iichs_info_m.p_data1st = access_addr1;
    iichs_info_m.p_slv_adr = addr_eeprom;
    iichs_info_m.scl_up_time = 20E-9;
    iichs_info_m.scl_down_time = 20E-9;
    iichs_info_m.fs_scl_up_time = 20E-9;
    iichs_info_m.fs_scl_down_time = 20E-9;
    iichs_info_m.speed_kbps = 3400;
    iichs_info_m.fs_speed_kbps = 400;
    iichs_info_m.bus_check_counter = 1000;
    iichs_info_m.bus_free_time = 5;
    iichs_info_m.slave_addr0 = 0x0000;
    iichs_info_m.slave_addr1 = 0x0000;
    iichs_info_m.slave_addr2 = 0x0000;
    iichs_info_m.slave_addr0_format = RIICHS_SEVEN_BIT_ADDR_FORMAT;
    iichs_info_m.slave_addr1_format = RIICHS_ADDR_FORMAT_NONE;
    iichs_info_m.slave_addr2_format = RIICHS_ADDR_FORMAT_NONE;
    iichs_info_m.gca_enable = RIICHS_GCA_DISABLE;
    iichs_info_m.rxi_priority = RIICHS_IPL_1;
    iichs_info_m.txi_priority = RIICHS_IPL_1;
    iichs_info_m.eei_priority = RIICHS_IPL_1;
    iichs_info_m.tei_priority = RIICHS_IPL_1;
    iichs_info_m.master_arb = RIICHS_MASTER_ARB_LOST_DISABLE;
    iichs_info_m.filter_stage = RIICHS_DIGITAL_FILTER_0;
    iichs_info_m.timeout_enable = RIICHS_TMO_ENABLE;
    iichs_info_m.nack_detc_enable = RIICHS_NACK_DETC_ENABLE;
    iichs_info_m.arb_lost_enable = RIICHS_ARB_LOST_ENABLE;
    iichs_info_m.counter_bit = RIICHS_COUNTER_BIT16;
    iichs_info_m.l_count = RIICHS_L_COUNT_ENABLE;
```

```
iichs_info_m.h_count = RIICHS_H_COUNT_ENABLE;
iichs_info_m.timeout_mode = RIICHS_TIMEOUT_MODE_ALL;

/* Sets IIC Information for Slave Transfer. */
iichs_info_s.dev_sts = RIICHS_NO_INIT;
iichs_info_s.ch_no = 0;
iichs_info_s.callbackfunc = &CallbackSlave;
iichs_info_s.cnt2nd = 4;
iichs_info_s.cnt1st = 3;
iichs_info_s.p_data2nd = slv_store_area;
iichs_info_s.p_data1st = slv_send_data;
iichs_info_s.p_slv_adr = (uint8_t*)FIT_NO_PTR;
iichs_info_s.scl_up_time = 20E-9;
iichs_info_s.scl_down_time = 20E-9;
iichs_info_s.fs_scl_up_time = 20E-9;
iichs_info_s.fs_scl_down_time = 20E-9;
iichs_info_s.speed_kbps = 3400;
iichs_info_s.fs_speed_kbps = 400;
iichs_info_s.bus_check_counter = 1000;
iichs_info_s.bus_free_time = 5;
iichs_info_s.slave_addr0 = 0x0025;
iichs_info_s.slave_addr1 = 0x0000;
iichs_info_s.slave_addr2 = 0x0000;
iichs_info_s.slave_addr0_format = RIICHS_SEVEN_BIT_ADDR_FORMAT;
iichs_info_s.slave_addr1_format = RIICHS_ADDR_FORMAT_NONE;
iichs_info_s.slave_addr2_format = RIICHS_ADDR_FORMAT_NONE;
iichs_info_s.gca_enable = RIICHS_GCA_DISABLE;
iichs_info_s.rxi_priority = RIICHS_IPL_1;
iichs_info_s.txi_priority = RIICHS_IPL_1;
iichs_info_s.eei_priority = RIICHS_IPL_1;
iichs_info_s.tei_priority = RIICHS_IPL_1;
iichs_info_s.master_arb = RIICHS_MASTER_ARB_LOST_DISABLE;
iichs_info_s.filter_stage = RIICHS_DIGITAL_FILTER_0;
iichs_info_s.timeout_enable = RIICHS_TMO_ENABLE;
iichs_info_s.nack_detc_enable = RIICHS_NACK_DETC_ENABLE;
iichs_info_s.arb_lost_enable = RIICHS_ARB_LOST_ENABLE;
iichs_info_s.counter_bit = RIICHS_COUNTER_BIT16;
iichs_info_s.l_count = RIICHS_L_COUNT_ENABLE;
iichs_info_s.h_count = RIICHS_H_COUNT_ENABLE;
iichs_info_s.timeout_mode = RIICHS_TIMEOUT_MODE_ALL;

/* RIICHS open */
ret = R_RIICHS_Open(&iichs_info_m);

/* RIICHS slave transfer enable */
ret = R_RIICHS_SlaveTransfer(&iichs_info_s);

/* RIICHS master send start */
ret = R_RIICHS_MasterSend(&iichs_info_m);

while(1);
}

void CallbackMaster(void)
{
    volatile riichs_return_t ret;

    riichs_mcu_status_t    iichs_status;

    ret = R_RIICHS_GetStatus(&iichs_info_m, &iichs_status);
    if(RIICHS_SUCCESS != ret)
```

```

{
    /* R_RIICHS_GetStatus 関数のエラー処理 */
}
else
{
    /* iichs_status のステータスフラグを確認して
       タイムアウト、アービトレーションロスト、NACK
       などが検出されていた場合の処理を記述 */
}
}

void CallbackSlave(void)
{
    /* スレーブモードでのイベント発生時に必要な処理があれば記述 */
}

```

Special Notes:

受信パターンごとの引数の設定可能範囲は、下表を参照してください。

構造体メンバ	ユーザ設定可能範囲	
	スレーブ受信	スレーブ送信
*p_slv_adr	未使用(設定無効)	未使用(設定無効)
*p_data1st	(スレーブ送信用)	[送信用]1st データバッファポインタ(注 1)
*p_data2nd	[受信用]2nd データバッファポインタ(注 2)	(スレーブ受信用)
dev_sts	デバイス状態バッファフラグ	デバイス状態バッファフラグ
cnt1st	(スレーブ送信用)	0000 0001h~FFFF FFFFh
cnt2nd	0000 0001h~FFFF FFFFh	(スレーブ受信用)
callbackfunc	使用する関数名を指定してください。	使用する関数名を指定してください。
ch_no	00h~FFh	00h~FFh
rsv1,rsv2	予約領域(設定無効)	予約領域(設定無効)

注 1：スレーブ送信を使用する場合、設定してください。

システムとして、スレーブ送信を使用しない場合、“FIT_NO_PTR”を設定してください。

注 2：スレーブ受信を使用する場合、設定してください。

システムとして、スレーブ受信を使用しない場合、“FIT_NO_PTR”を設定してください。

R_RIICHS_GetStatus()

本モジュールの状態を返します。

Format

```

riichs_sts_flg_t R_RIICHS_GetStatus(
    riichs_info_t *      p_riichs_info /* 構造体データ */
    riichs_mcu_status_t * p_riichs_status /* RIICHS のステータス */
)

```

Parameters

**p_riichs_info*

I²C 通信情報構造体のポインタ。

この構造体のうち、本関数で使用するメンバのみを以下に示します。この構造体の詳細については2.9を参照してください。

構造体の内容は、通信中に参照、更新されます。このため、通信中(RIICHS_COMMUNICATION)に構造体の内容を書き換えしないでください。

下記のうち、API 実行中に値が更新される引数には、“更新あり”と記載しています。

```

riichs_ch_dev_status_t    dev_sts;      /* デバイス状態フラグ
                                         (ステータスが“RIICHS_AL”時、更新あり) */
uint8_t                   ch_no;        /* チャンネル番号 */

```

**p_riichs_status*

RIICHS のステータスを格納する変数のポインタ。

下記構造体で定義しているメンバで指定します。

typedef union

```

{
    uint32_t          LONG;
    struct
    {
        uint32_t      rsv:11;          /* reserve */
        uint32_t      AAS2:1;          /* Slave2 address detection flag */
        uint32_t      AAS1:1;          /* Slave1 address detection flag */
        uint32_t      AAS0:1;          /* Slave0 address detection flag */
        uint32_t      GCA :1;          /* Generalcall address detection flag */
        uint32_t      DID :1;          /* DeviceID address detection flag */
        uint32_t      HOA :1;          /* Host address detection flag */
        uint32_t      MST :1;          /* Master mode / Slave mode flag */
        uint32_t      TMO :1;          /* Time out flag */
        uint32_t      AL:1;             /* Arbitration lost detection flag */
        uint32_t      SP:1;             /* Stop condition detection flag */
        uint32_t      ST:1;             /* Start condition detection flag */
        uint32_t      RBUF:1;          /* Receive buffer status flag */
        uint32_t      SBUF:1;          /* Send buffer status flag */
        uint32_t      SCLO:1;          /* SCL pin output control status */
        uint32_t      SDAO:1;          /* SDA pin output control status */
        uint32_t      SCLI:1;          /* SCL pin level */
        uint32_t      SDAI:1;          /* SDA pin level */
        uint32_t      NACK:1;          /* NACK detection flag */
        uint32_t      TRS:1;           /* Send mode / Receive mode flag */
        uint32_t      BSY:1;           /* Bus status flag */
        uint32_t      HSMC:1;          /* Hs mode Master Code Detection flag */
    }BIT;
} riichs_mcu_status_t;

```

Return Values

RIICHS_SUCCESS /* 問題なく処理が完了した場合 */
 RIICHS_ERR_INVALID_CHAN /* 存在しないチャンネルの場合 */
 RIICHS_ERR_INVALID_ARG /* 不正な引数の場合 */

Properties

r_riichs_rx_if.h にプロトタイプ宣言されています。

Description

本モジュールの状態を返します。

引数で指定した RIICHS のチャンネルの状態を、レジスタの読み出し、端子レベルの読み出し、変数の読み出しなどにより取得し、32 ビットの構造体で戻り値として返します。

本関数では、RIICHS のアービトレーションロストフラグ、および NACK フラグを “0” にクリアします。ステータスが “RIICHS_AL” の場合、“RIICHS_FINISH” に更新します。

Example

```
volatile riichs_return_t ret;
riichs_info_t iichs_info_m;
riichs_mcu_status_t riichs_status;

iichs_info_m.ch_no = 0;

ret = R_RIICHS_GetStatus(&iichs_info_m, &riichs_status);
```

Special Notes:

以下にステータスフラグの配置を示します。

b31 to b21	b20	b19	b18	b17	b16
Reserved	Slave Address Detection			Event detection	Event detection
Reserved	Slave Address 0 Detection	Slave Address 1 Detection	Slave Address 2 Detection	General call detection	Device-ID detection
Rsv	AAS2	AAS1	AAS0	GCA	DID
Undefined	0: Not detected 1: Detected			0: Not detected 1: Detected	0: Not detected 1: Detected

b15	b14	b13	b12	b11	b10	b9	b8
Event detection	Mode flag	Event detection	Event detection	Event detection	Event detection	buffer status	buffer status
Host address detection	Master/slave mode	Timeout detection	Arbitration lost detection	Stop detection	Start detection	Receive buffer	Send buffer
HOA	MST	TMO	AL	SP	ST	RBUF	SBUF
0: Not detected 1: Detected	0: slave 1: master	0: Not detected 1: Detected	0: Not detected 1: Detected	0: Not detected 1: Detected	0: Not detected 1: Detected	0: no data 1: have data	0: have data 1: no data

b7	b6	b5	b4	b3	b2	b1	b0
Pin status		Pin level		Event detection	Mode flag	Bus state	Event detection
SCL pin control	SDA pin control	SCL pin level	SDA pin level	NACK detection	Transmit/Receive mode	Bus busy/ready	Master code detection
SCLO	SDAO	SCLI	SDAI	NACK	TRS	BSY	HSMC
0: Output low level 1: Output Hi-Z		0: Low level 1: High level		0: Not detected 1: Detected	0: Receive 1: Transmit	0: Idle 1: Busy	0: Not detected 1: Detected

R_RIICHS_Control()

各コンディション出力、SDA 端子のハイインピーダンス出力、SCL クロックのワンショット出力、および RIICHS のモジュールリセットを行う関数です。主に通信エラー時に使用してください。

Format

```
riichs_return_t R_RIICHS_Control(
    r_riichs_info_t *p_riichs_info /* 構造体データ */
    uint8_t ctrl_ptn /* 出力パターン */
)
```

Parameters

**p_riichs_info*

I²C 通信情報構造体のポインタ。

この構造体のうち、本関数で使用するメンバのみを以下に示します。この構造体の詳細については2.9を参照してください。

構造体の内容は、通信中に参照、更新されます。このため、通信中(RIICHS_COMMUNICATION)に構造体の内容を書き換えしないでください。

下記のうち、API 実行中に値が更新される引数には、“更新あり”と記載しています。

```
riichs_ch_dev_status_t dev_sts; /* デバイス状態フラグ (出力パターンに
                                “RIICHS_GEN_RESET” 指定時、更新あり) */
uint8_t ch_no; /* チャンネル番号 */
```

ctrl_ptn

出力パターンを設定します。

- 次の出力パターンは、同時指定が可能です。同時指定する場合は、“|” (OR)を用いてください。

- ・ “RIICHS_GEN_START_CON” と “RIICHS_GEN_STOP_CON” と “RIICHS_GEN_RESTART_CON” の、3 つ、または、いずれか 2 つの組み合わせで同時指定可能です。
- ・ “RIICHS_GEN_SDA_HI_Z” と “RIICHS_GEN_SCL_ONESHOT” の 2 つを同時指定可能です。

```
#define RIICHS_GEN_START_CON (uint8_t)(0x01) /* スタートコンディションの生成 */
#define RIICHS_GEN_STOP_CON (uint8_t)(0x02) /* ストップコンディションの生成 */
#define RIICHS_GEN_RESTART_CON (uint8_t)(0x04) /* リスタートコンディションの生成 */
#define RIICHS_GEN_SDA_HI_Z (uint8_t)(0x08) /* SDA 端子をハイインピーダンス出力 */
#define RIICHS_GEN_SCL_ONESHOT (uint8_t)(0x10) /* SCL クロックのワンショット出力 */
#define RIICHS_GEN_RESET (uint8_t)(0x20) /* RIICHS のモジュールリセット */
```

Return Values

```
RIICHS_SUCCESS /* 問題なく処理が完了した場合 */
RIICHS_ERR_INVALID_CHAN /* 存在しないチャンネルの場合 */
RIICHS_ERR_INVALID_ARG /* 不正な引数の場合 */
RIICHS_ERR_BUS_BUSY /* バスビジーの場合 */
RIICHS_ERR_AL /* アービトレーションエラーが発生した場合 */
RIICHS_ERR_OTHER /* 現在の状態に該当しない不正なイベントが発生した場合 */
```

Properties

r_riichs_rx_if.h にプロトタイプ宣言されています。

Description

RIICHS の制御信号を出力します。引数で指定した各コンディション出力、SDA 端子のハイインピーダンス出力、SCL クロックのワンショット出力、および RIICHS のモジュールリセットを行います。

Example

```
/* Outputs an extra SCL clock cycle after the SDA pin state is changed to a
high-impedance state. */
volatile riichs_return_t  ret;
riichs_info_t             iichs_info_m;

iichs_info_m.ch_no = 0;

ret = R_RIICHS_Control(&iichs_info_m, RIICHS_GEN_SDA_HI_Z |
RIICHS_GEN_SCL_ONESHOT);
```

Special Notes:

【出力パターンの SCL クロックのワンショット出力について】

マスタモード時、ノイズ等の影響でスレーブデバイスとの同期ズレが発生するとスレーブデバイスが SDA ラインを Low 固定状態にする場合があります(バスハングアップ)。この場合、SCL を 1 クロックずつ出力することでスレーブデバイスによる SDA ラインの Low 固定状態を解放させ、バス状態を復帰させることができます。

本モジュールでは、出力パターンに RIICHS_GEN_SCL_ONESHOT(SCL クロックのワンショット出力)を設定して R_RIICHS_Control 関数を呼び出すことにより、SCL を 1 クロック出力することができます。

R_RIICHS_Close()

RIICHS の通信を終了し、使用していた RIIC の対象チャネルを解放します。

Format

```
riichs_return_t R_RIICHS_Close(  
    riichs_info_t *    p_riichs_info    /* 構造体データ */  
)
```

Parameters

**p_riichs_info*

I²C 通信情報構造体のポインタ。

この構造体のうち、本関数で使用するメンバのみを以下に示します。この構造体の詳細については2.9を参照してください。

構造体の内容は、通信中に参照、更新されます。このため、通信中(RIICHS_COMMUNICATION)に構造体の内容を書き換えしないでください。

下記のうち、API 実行中に値が更新される引数には、“更新あり”と記載しています。

```
riichs_ch_dev_status_t    dev_sts;        /* デバイス状態フラグ (更新あり) */  
uint8_t                   ch_no;          /* チャネル番号 */
```

Return Values

```
RIICHS_SUCCESS            /* 問題なく処理が完了した場合 */  
RIICHS_ERR_INVALID_CHAN  /* 存在しないチャネルの場合 */  
RIICHS_ERR_INVALID_ARG   /* 不正な引数の場合 */
```

Properties

r_riichs_rx_if.h にプロトタイプ宣言されています。

Description

RIICHS 通信を終了するための設定をします。引数で指定した RIICHS のチャネルを無効にします。本関数では次の処理を行います。

- － RIICHS のモジュールストップ状態への遷移
- － I²C 出力ポートの開放
- － RIICHS 割り込みの禁止

再度通信を開始するには、R_RIICHS_Open (初期化関数)を呼び出す必要があります。通信中に強制的に停止した場合、その通信は保証しません。

Example

```
volatile riichs_return_t    ret;  
riichs_info_t               iichs_info_m;  
  
iichs_info_m.ch_no = 0;  
  
ret = R_RIICHS_Close(&iichs_info_m);
```

Special Notes:

なし

R_RIICHS_GetVersion()

本モジュールのバージョンを返します。

Format

uint32_t R_RIICHS_GetVersion(void)

Parameters

なし

Return Values

バージョン番号

Properties

r_riichs_rx_if.h にプロトタイプ宣言されています。

Description

本関数は、現在インストールされている RIICHS FIT モジュールのバージョンを返します。バージョン番号はコード化されています。最初の 2 バイトがメジャーバージョン番号、後の 2 バイトがマイナーバージョン番号です。例えば、バージョンが 4.25 の場合、戻り値は'0x00040019'となります。

Example

```
uint32_t    version;  
  
version = R_RIICHS_GetVersion();
```

Special Notes:

なし。

4. 端子設定

RIICHS FIT モジュールを使用するためには、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付ける（以下、端子設定と称す）必要があります。

RIICHS FIT モジュールは、コンフィグレーションオプションの RIICHS_CFG_PORT_SET_PROCESSING の設定により、R_RIICHS_Open 関数の中で端子設定するかを選択できます。

コンフィグレーションオプションの詳細は、「2.7 コンパイル時の設定」を参照ください。

e²studio の場合は「FIT Configurator」または「Smart Configurator」の端子設定機能を使用することができます。FIT Configurator、Smart Configurator の端子機能を使用すると、端子設定画面で選択した端子を使用することができます。選択した端子情報は r_riichs_pin_config.h に反映され、表 4.1 に示すマクロ定義の値が選択した端子に応じた値に上書きされます。RIICHS FIT モジュールでは、FIT Configurator を使用する場合、端子設定機能を有効にする関数が記述されたソースファイル（および“r_pincfg”フォルダ）は生成されません。

表 4.1 端子設定マクロ定義

選択したチャネル	選択した端子	マクロ定義
チャネル 0	SCL0 端子	R_RIICHS_CFG_RIICHS0_SCL0_PORT R_RIICHS_CFG_RIICHS0_SCL0_BIT
	SDA0 端子	R_RIICHS_CFG_RIICHS0_SDA0_PORT R_RIICHS_CFG_RIICHS0_SDA0_BIT

r_riichs_pin_config.h で選択した端子は、R_RIICHS_Open 関数呼び出し後、周辺機能端子として SCL 端子、SDA 端子となります。

周辺機能端子の割り付けは R_RIICHS_Close 関数が呼び出されると解除され、端子は汎用入出力端子（入力状態）になります。

なお、SCL 端子、SDA 端子は外付け抵抗でプルアップ処理を行ってください。

もし RIICHS_CFG_PORT_SET_PROCESSING の設定で本モジュール内の端子設定処理を使用しない場合は、R_RIICHS_Open 関数を呼び出した後、その他の API を呼び出す前にユーザ処理で使用する端子の端子設定を行ってください。

5. デモプロジェクト

デモプロジェクトはスタンドアロンプログラムです。デモプロジェクトには、FIT モジュールとそのモジュールが依存するモジュール（例：r_bsp）を使用する main()関数が含まれます。

開発環境の操作に関しては、e² studio を例に説明します。

5.1 riichs_mastersend_demo_rskrx671

説明：

RSKRX671（FIT モジュール “r_riichs_rx”）向けの RX671 マスタ送信を行うデモです。デモでは RIICHS FIT モジュールの API（r_riichs_rx_if.h に記載）を使って、マスタデバイスとして、スレーブデバイスヘータを送信します。マスタ送信終了で Main()関数によって、デバッグコンソールに出力します。

設定と実行：

1. サンプルコードをコンパイルしてダウンロードします。
2. ソフトウェアを実行します。PC が Main で停止した場合、F8 を押して再開します。
3. ブレークポイントを設定し、グローバル変数を確認します。

対応ボード：

- RSKRX671

5.2 riichs_masterreceive_demo_rskrx671

説明：

RSKRX671（FIT モジュール “r_riichs_rx”）向けの RX671 マスタ受信を行うデモです。デモでは RIICHS FIT モジュールの API（r_riichs_rx_if.h に記載）を使って、マスタデバイスとして、スレーブデバイスからデータを受信します。マスタ受信終了で Main()関数によって、受信したデータをデバッグコンソールに出力します。

対応ボード：

- RSKRX671

5.3 riichs_slavetransfer_demo_rskrx671

説明：

RSKRX671（FIT モジュール “r_riichs_rx”）向けの RX671 スレーブ送受信を行うデモです。デモでは RIICHS FIT モジュールの API（r_riichs_rx_if.h に記載）を使って、マスタデバイスから送信されるデータをスレーブデバイスとして受信、または、マスタデバイスの送信要求により、スレーブデバイスとしてデータを送信します。マスタ送受信終了で Main()関数によって、デバッグコンソールに出力します。

対応ボード：

- RSKRX671

5.4 ワークスペースにデモを追加する

デモプロジェクトは、e² studio のインストールディレクトリ内の 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。

5.5 デモのダウンロード方法

デモプロジェクトは、RX Driver Package には同梱されていません。デモプロジェクトを使用する場合は、個別に各 FIT モジュールをダウンロードする必要があります。「スマートブラウザ」の「アプリケーションノート」タブから、本アプリケーションノートを右クリックして「サンプル・コード（ダウンロード）」を選択することにより、ダウンロードできます。

6. 付録

6.1 通信方法の実現

本モジュールでは、スタートコンディション生成やスレーブアドレス送信などの処理を 1 つのプロトコルとして管理しており、このプロトコルを組み合わせることで通信を実現します。

6.1.1 制御時の状態

表 6.1 に、プロトコル制御を実現するための状態を定義します。

表 6.1 プロトコル制御のための状態一覧(enum r_riichs_api_status_t)

No	状態名	状態の定義
STS0	RIICHS_STS_NO_INIT	未初期化状態
STS1	RIICHS_STS_IDLE	アイドル状態(マスタ通信可能状態)
STS2	RIICHS_STS_IDLE_EN_SLV	アイドル状態(マスタ/スレーブ通信可能状態)
STS3	RIICHS_STS_ST_COND_WAIT	スタートコンディション検出待ち状態
STS4	RIICHS_STS_MASTER_CODE_WAIT	Hs モードマスタコード送信完了待ち状態
STS5	RIICHS_STS_SEND_SLVADR_W_WAIT	スレーブアドレス[Write]送信完了待ち状態
STS6	RIICHS_STS_SEND_SLVADR_R_WAIT	スレーブアドレス[Read]送信完了待ち状態
STS7	RIICSH_STS_SEND_DATA_WAIT	データ送信完了待ち状態
STS8	RIICHS_STS_RECEIVE_DATA_WAIT	データ受信完了待ち状態
STS9	RIICHS_STS_SP_COND_WAIT	ストップコンディション検出待ち状態
STS10	RIICHS_STS_AL	アービトレーションロスト状態
STS11	RIICHS_STS_TMO	タイムアウト検出状態

6.1.2 制御時のイベント

表 6.2 にプロトコル制御時に発生するイベントを定義します。割り込みだけでなく、本モジュールが提供する API 関数が呼び出された際も、イベントとして定義します。

表 6.2 プロトコル制御のためのイベント一覧(enum r_riichs_api_event_t)

No	イベント名	イベントの定義
EV0	RIICHS_EV_INIT	R_RIICHS_Open()呼び出し
EV1	RIICHS_EV_EN_SLV_TRANSFER	R_RIICHS_SlaveTransfer()呼び出し
EV2	RIICHS_EV_GEN_START_COND	R_RIICHS_MasterSend() または R_RIICHS_MasterReceive()呼び出し
EV3	RIICHS_EV_INT_START	EEI 割り込み発生 (割り込みフラグ : START) TEI 割り込み発生 (マスタコード(0000 1XXXb) 送信) EEI 割り込み発生 (割り込みフラグ : NACK)
EV4	RIICHS_EV_INT_ADD	TEI 割り込み発生、TXI 割り込み発生 (注 1)
EV5	RIICHS_EV_INT_SEND	TEI 割り込み発生、TXI 割り込み発生 (注 1)
EV6	RIICHS_EV_INT_RECEIVE	RXI 割り込み発生
EV7	RIICHS_EV_INT_STOP	EEI 割り込み発生 (割り込みフラグ : STOP)
EV8	RIICHS_EV_INT_AL	EEI 割り込み発生 (割り込みフラグ : AL)
EV9	RIICHS_EV_INT_NACK	EEI 割り込み発生 (割り込みフラグ : NACK)
EV10	RIICHS_EV_INT_TMO	EEI 割り込み発生 (割り込みフラグ : TMO)

注1. EV4 と EV5 の定義は通信動作と「6.1.1 制御時の状態」の状態により異なります。詳細は、「6.1.3 プロトコル状態遷移」を参照ください。

6.1.3 プロトコル状態遷移

本モジュールでは、提供する API 関数の呼び出し、または、I²C 割り込み発生をトリガに状態が遷移します。
図 6.1～図 6.4に各プロトコルの状態遷移を示します。

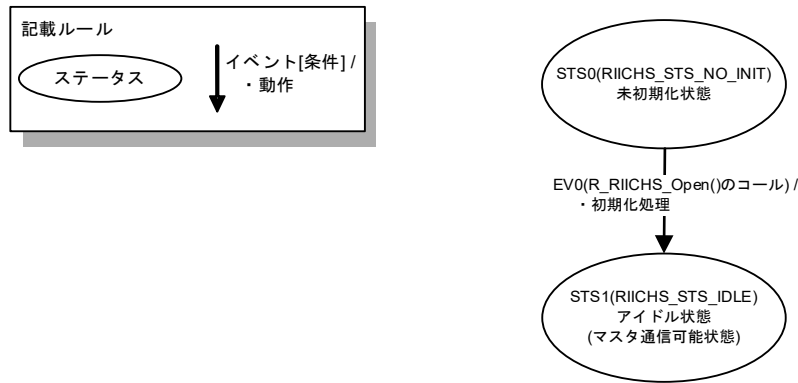


図 6.1 初期化処理（R_RIICHS_Open()呼び出し）時の状態遷移図

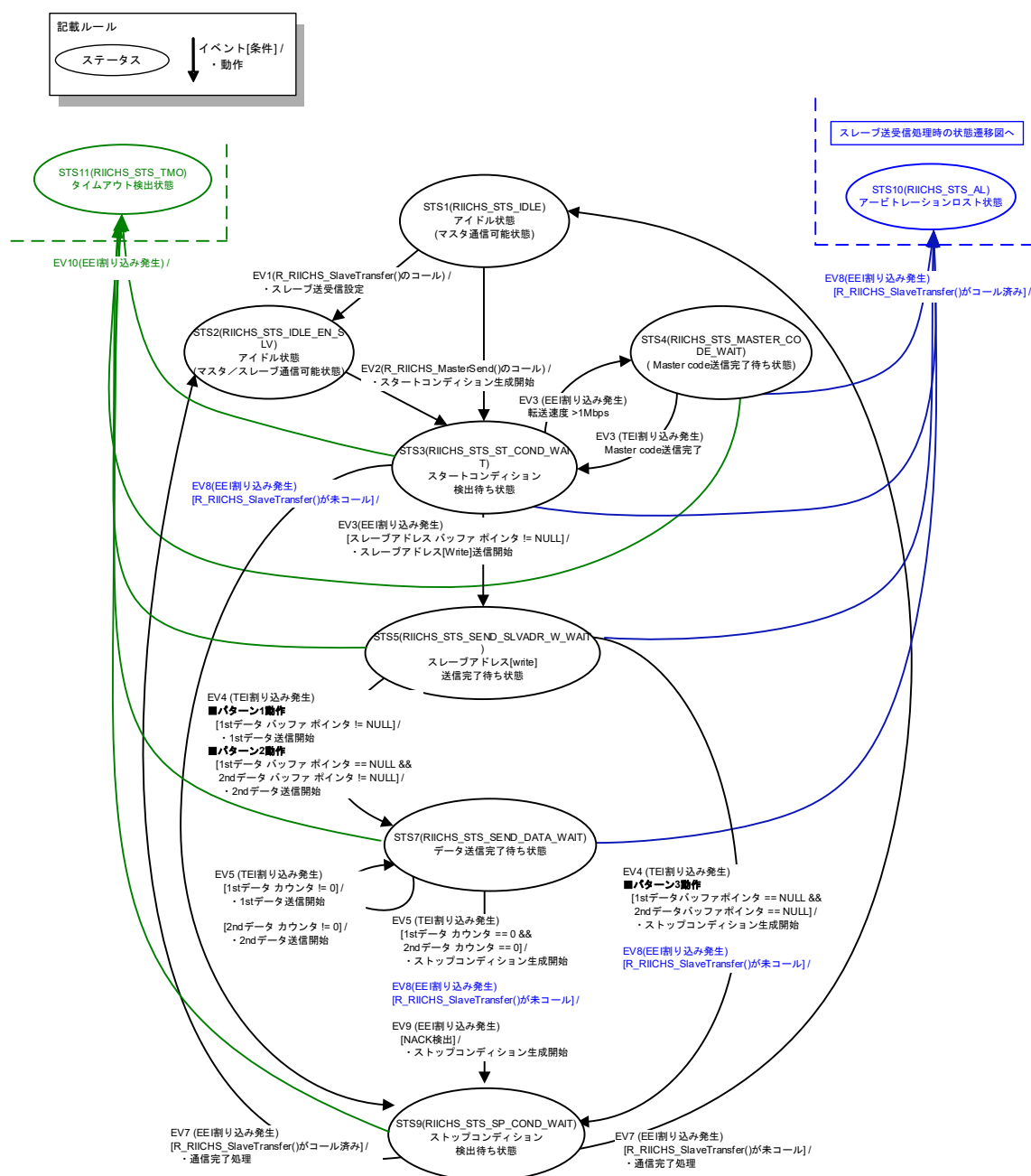


図 6.2 マスタ送信処理 (R_RIICHS_MasterSend()呼び出し) 時の状態遷移図

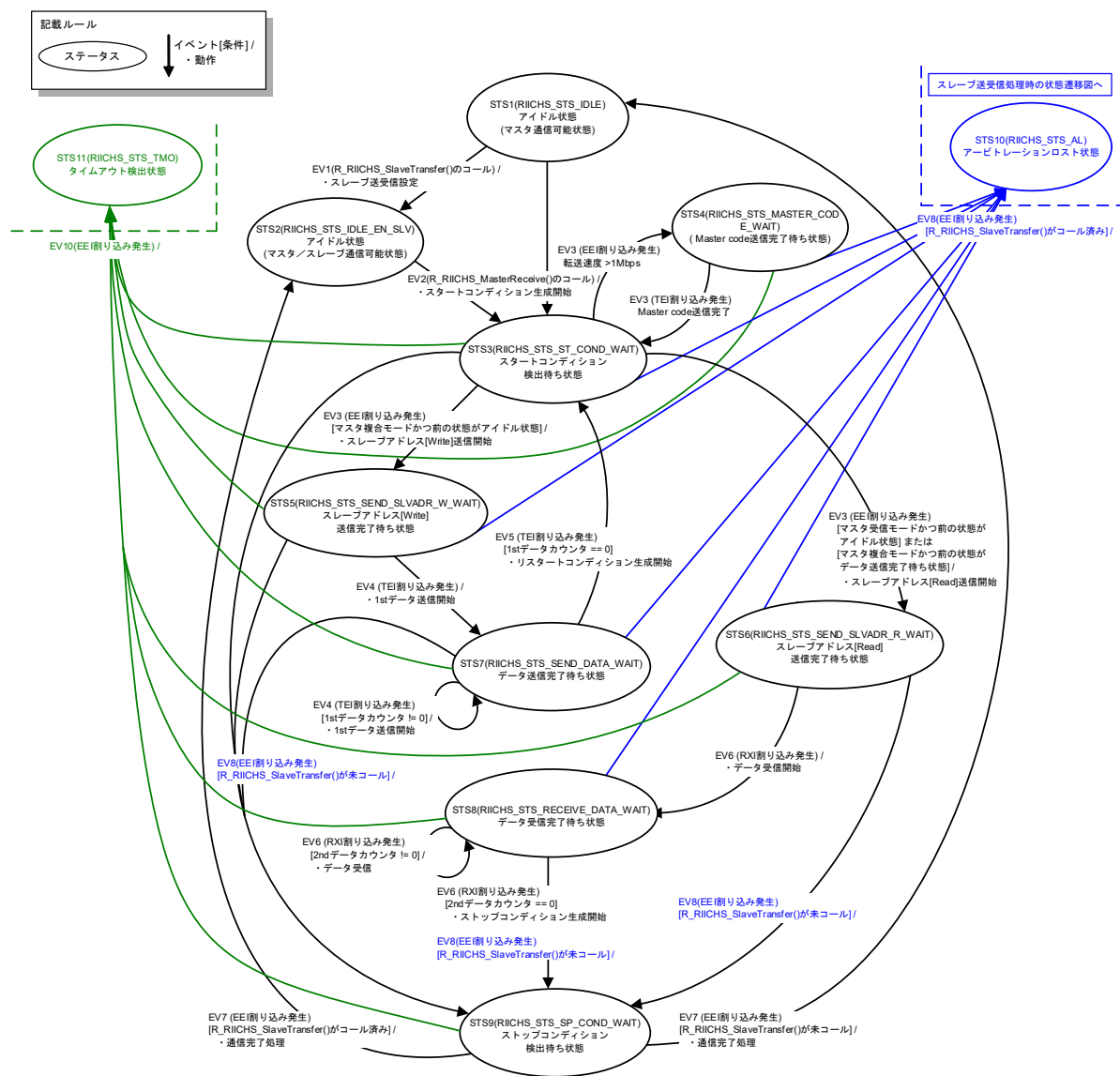


図 6.3 マスタ受信処理 (R_RIICHS_MasterReceive()呼び出し) 時の状態遷移図

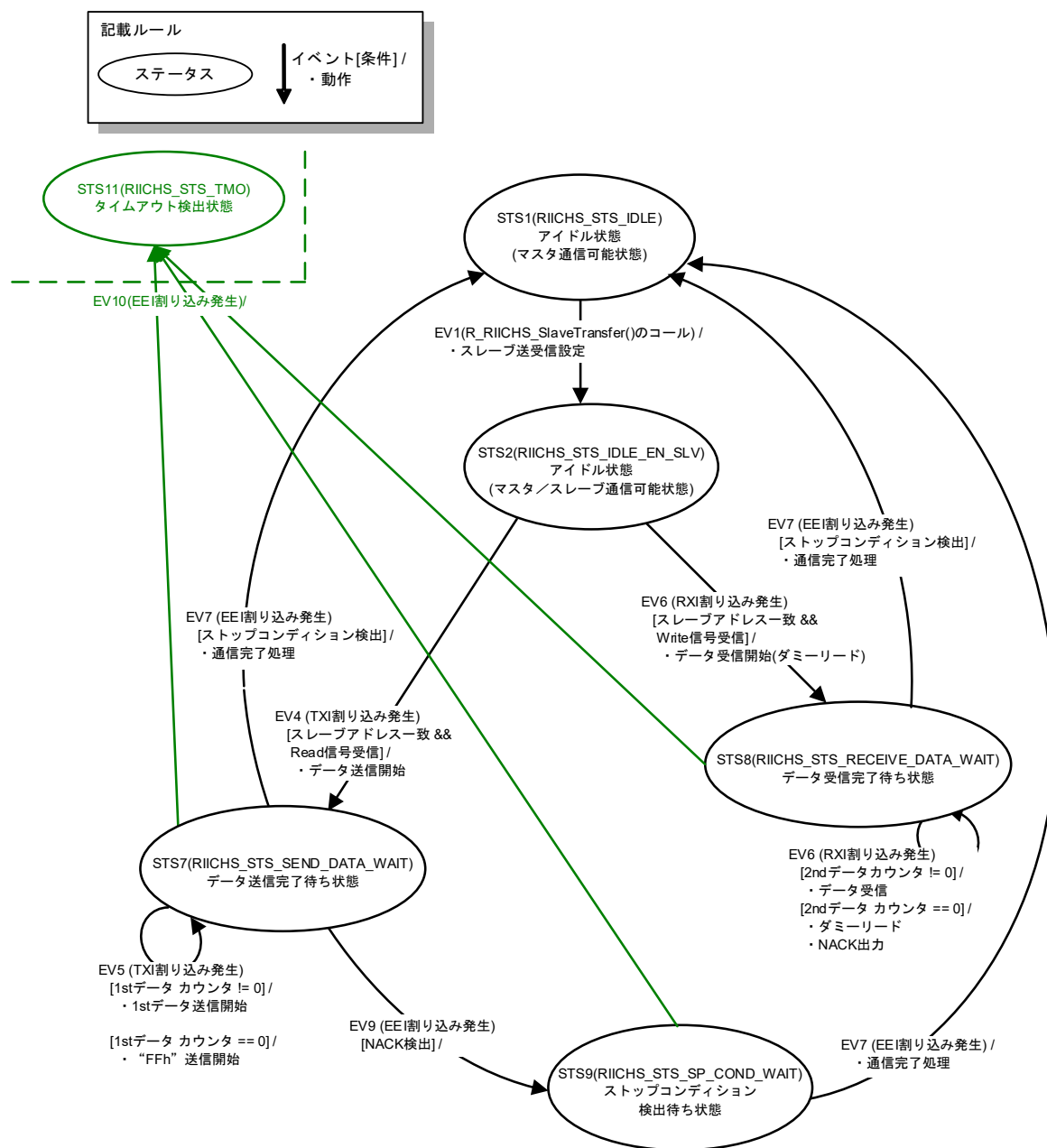


図 6.4 スレーブ送受信処理 (R_RIICHS_SlaveTransfer()呼び出し) 時の状態遷移図

6.1.4 プロトコル状態遷移表

表 6.1の各状態で、表 6.2のイベントが発生した際に動作する処理を、表 6.3の状態遷移表に定義します。

Func0~Func12については、表 6.4を参照してください。

表 6.3 プロトコル状態遷移表(gc_riichs_mtx_tbl [][])

状態	イベント	EV0	EV1	EV2	EV3	EV4	EV5	EV6	EV7	EV8	EV9	EV10
STS0	未初期化状態 【RIICHS_STS_NO_INIT】	Func0	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
STS1	アイドル状態 (マスタ通信可能状態) 【RIICHS_STS_IDLE】	ERR	Func10	Func1	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR
STS2	アイドル状態 (マスタ/スレーブ通信可能状態) 【RIICHS_STS_IDLE_EN_SLV】	ERR	ERR	Func1	ERR	Func4	ERR	Func4	ERR	ERR	ERR	ERR
STS3	スタートコンディション生成完了待ち状態 【RIICHS_STS_ST_COND_WAIT】	ERR	ERR	ERR	Func2	ERR	ERR	ERR	ERR	Func8	Func9	Func11
STS4	Hs モードマスタコード送信完了待ち状態 【RIICHS_STS_MASTER_CODE_WAIT】	ERR	ERR	ERR	Func12	ERR	ERR	ERR	ERR	Func8	ERR	Func11
STS5	スレーブアドレス[Write]送信完了待ち状態 【RIICHS_STS_SEND_SLVADR_W_WAIT】	ERR	ERR	ERR	ERR	Func3	ERR	ERR	ERR	Func8	Func9	Func11
STS6	スレーブアドレス[Read]送信完了待ち状態 【RIICHS_STS_SEND_SLVADR_R_WAIT】	ERR	ERR	ERR	ERR	ERR	ERR	Func3	ERR	Func8	Func9	Func11
STS7	データ送信完了待ち状態 【RIICHS_STS_SEND_DATA_WAIT】	ERR	ERR	ERR	ERR	ERR	Func5	ERR	ERR	Func8	Func9	Func11
STS8	データ受信完了待ち状態 【RIICHS_STS_RECEIVE_DATA_WAIT】	ERR	ERR	ERR	ERR	ERR	ERR	Func6	ERR	Func8	Func9	Func11
STS9	ストップコンディション生成完了待ち状態 【RIICHS_STS_SP_COND_WAIT】	ERR	ERR	ERR	ERR	ERR	ERR	ERR	Func7	ERR	Func9	Func11
STS10	アービトレーションロスト状態 【RIICHS_STS_AL】	ERR	ERR	ERR	ERR	ERR	Func5	Func6	Func7	ERR	ERR	ERR
STS11	タイムアウト検出状態 【RIICHS_STS_TMO】	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR	ERR

備考: ERR は RIICHS_ERR_OTHER を表します。ある状態で意図しないイベントが通知された場合には、すべてエラー処理を行います。

6.1.5 プロトコル状態遷移登録関数

表 6.4に状態遷移表に登録されている関数を定義します。

表 6.4 プロトコル状態遷移登録関数一覧

処理	関数名	概要
Func0	riichs_init_driver()	初期設定処理
Func1	riichs_generate_start_cond()	スタートコンディション生成処理(マスタ送信用)
Func2	riichs_after_gen_start_cond()	スタートコンディション生成後処理
Func3	riichs_after_send_slvadr()	スレーブアドレスが送信完了した後の処理
Func4	riichs_after_receive_slvadr()	受信したスレーブアドレスが一致した後の処理
Func5	riichs_write_data_sending()	データ送信処理
Func6	riichs_read_data_receiving()	データ受信処理
Func7	riichs_after_dtct_stop_cond()	通信完了処理
Func8	riichs_arbitration_lost()	アービトレーションロスト検出した時の処理
Func9	riichs_nack()	NACK 検出した時の処理
Func10	riichs_enable_slave_transfer()	スレーブ送受信有効
Func11	riichs_time_out()	タイムアウト検出時の処理
Func12	riichs_send_master_code_cond()	マスターコード(0000 1XXXb) 送信処理

6.1.6 状態遷移時の各フラグの状態

<各チャネル状態管理>

チャネル状態フラグ g_riichs_ChStatus[]により、1つのバス上に接続された複数スレーブデバイスの排他制御を行います。

本フラグは、各チャネルに対して1つ存在し、グローバル変数で管理します。本モジュールの初期化処理を完了し、対象バスで通信が行われていない場合、本フラグは

“RIICHS_IDLE/RIICHS_FINISH/RIICHS_NACK” (アイドル状態(通信可能))となり、通信が可能です。通信中の本フラグの状態は、“RIICHS_COMMUNICATION” (通信中)になります。通信開始時、必ず本フラグの確認を行うため、通信中に同一チャネル上の他デバイスの通信を開始しません。本フラグをチャネルごとに管理することで、複数チャネルの同時通信を実現します。

<各デバイス状態管理>

I²C 通信情報構造体メンバのデバイス状態フラグ dev_sts により、同一チャンネル上の複数のスレーブデバイスの制御を行うことができます。デバイス状態フラグには、そのデバイスの通信状態が格納されます。

表 6.5に状態遷移時の各フラグの状態を示します。

表 6.5 状態遷移時の各フラグの状態一覧

状態	チャンネル状態フラグ	デバイス状態フラグ (通信のデバイス)	I ² C プロトコルの動作モード	プロトコル制御の現状態
	g_riichs_ChStatus[]	I ² C 通信情報構造体 dev_sts	内部通信情報構造体 N_Mode	内部通信情報構造体 N_status
未初期化状態	RIICHS_NO_INIT	RIICHS_NO_INIT	RIICHS_MODE_NONE	RIICHS_STS_NO_INIT
アイドル状態 (マスタ通信可能状態)	RIICHS_IDLE RIICHS_FINISH RIICHS_NACK	RIICHS_IDLE RIICHS_FINISH RIICHS_NACK	RIICHS_MODE_NONE	RIICHS_STS_IDLE
アイドル状態 (マスタ/スレーブ通信可能状態)	RIICHS_IDLE	RIICHS_IDLE	RIICHS_MODE_S_READY	RIICHS_STS_IDLE_EN_SLV
通信中 (マスタ送信)	RIICHS_COMMUNICATION	RIICHS_COMMUNICATION	RIICHS_MODE_M_SEND	RIICHS_STS_ST_COND_WAIT RIICHS_STS_MASTER_CODE_WAIT RIICHS_STS_SEND_SLVADR_W_WAIT RIICHS_STS_SEND_DATA_WAIT RIICHS_STS_SP_COND_WAIT RIICHS_STS_AL RIICHS_STS_TMO
通信中 (マスタ受信)	RIICHS_COMMUNICATION	RIICHS_COMMUNICATION	RIICHS_MODE_M_RECEIVE	RIICHS_STS_ST_COND_WAIT RIICHS_STS_MASTER_CODE_WAIT RIICHS_STS_SEND_SLVADR_R_WAIT RIICHS_STS_RECEIVE_DATA_WAIT RIICHS_STS_SP_COND_WAIT RIICHS_STS_AL RIICHS_STS_TMO
通信中 (マスタ送受信)	RIICHS_COMMUNICATION	RIICHS_COMMUNICATION	RIICHS_MODE_M_SEND_RECEIVE	RIICHS_STS_ST_COND_WAIT RIICHS_STS_SEND_SLVADR_W_WAIT RIICHS_STS_SEND_SLVADR_R_WAIT RIICHS_STS_SEND_DATA_WAIT RIICHS_STS_RECEIVE_DATA_WAIT RIICHS_STS_SP_COND_WAIT RIICHS_STS_AL RIICHS_STS_TMO
通信中(スレーブ送信)	RIICHS_COMMUNICATION	RIICHS_COMMUNICATION	RIICHS_MODE_S_SEND	RIICHS_STS_SEND_DATA_WAIT RIICHS_STS_SP_COND_WAIT RIICHS_STS_TMO
通信中(スレーブ受信)	RIICHS_COMMUNICATION	RIICHS_COMMUNICATION	RIICHS_MODE_S_RECEIVE	RIICHS_STS_RECEIVE_DATA_WAIT RIICHS_STS_SP_COND_WAIT RIICHS_STS_TMO
アービトラクションロスト検出 状態	RIICHS_AL	RIICHS_AL		
タイムアウト検出状態	RIICHS_TMO	RIICHS_TMO		
エラー状態	RIICHS_ERROR	RIICHS_ERROR	-	-

6.2 割り込み発生タイミング

以下に本モジュールの割り込みタイミングを示します。

備考 ST : スタートコンディション

AD6-AD0 : スレーブアドレス

/W : 転送方向ビット “0” (Write)

R : 転送方向ビット “1” (Read)


/ACK : Acknowledge “0”

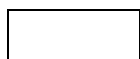
NACK : Acknowledge “1”

D7-D0 : データ

RST : リスタートコンディション

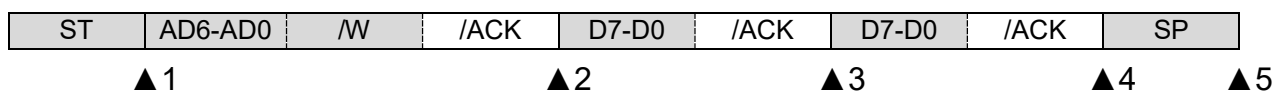
SP : ストップコンディショ

 マスタからスレーブへ

 スレーブからマスタへ

6.2.1 マスタ送信

1. パターン 1



▲1 : EEI (START) 割り込み・・・スタートコンディション検出

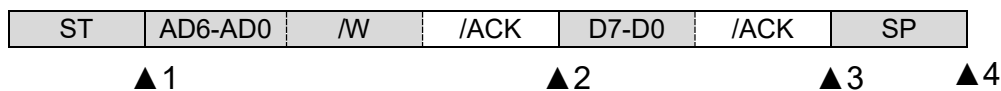
▲2 : TEI 割り込み・・・アドレス送信完了 (転送方向ビット : Write)

▲3 : TEI 割り込み・・・データ送信完了 (1st データ)

▲4 : TEI 割り込み・・・データ送信完了 (2nd データ)

▲5 : EEI (STOP) 割り込み・・・ストップコンディション検出

2. パターン 2



▲1 : EEI (START) 割り込み・・・スタートコンディション検出

▲2 : TEI 割り込み・・・アドレス送信完了 (転送方向ビット : Write)

▲3 : TEI 割り込み・・・データ送信完了 (2nd データ)

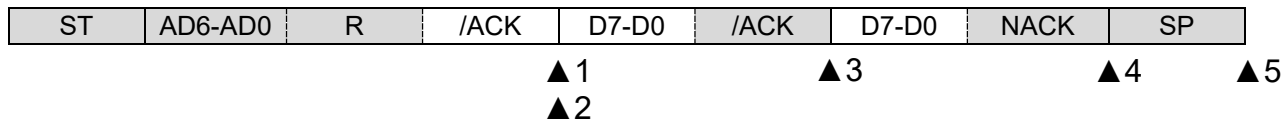
▲4 : EEI (STOP) 割り込み・・・ストップコンディション検出

ST	AD6-AD0	/W	/ACK	SP
	▲1		▲2	▲3

- Page 65 of 82

6.2.4 スレーブ送信

2 バイト送信時



▲1 : TXI 割り込み・・・アドレス受信一致（転送方向ビット : Read）

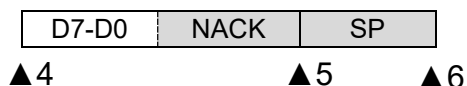
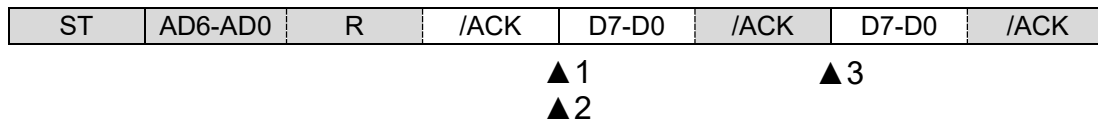
▲2 : TXI 割り込み・・・送信バッファ空

▲3 : TXI 割り込み・・・送信バッファ空

▲4 : EEI (NACK) 割り込み・・・NACK 検出

▲5 : EEI (STOP) 割り込み・・・ストップコンディション検出

3 バイト送信時



▲1 : TXI 割り込み・・・アドレス受信一致（転送方向ビット : Read）

▲2 : TXI 割り込み・・・送信バッファ空

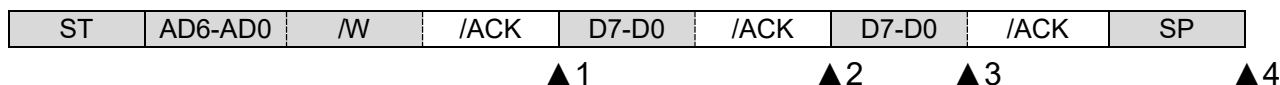
▲3 : TXI 割り込み・・・送信バッファ空

▲4 : TXI 割り込み・・・送信バッファ空

▲5 : EEI (NACK) 割り込み・・・NACK 検出

▲6 : EEI (STOP) 割り込み・・・ストップコンディション検出

6.2.5 スレーブ受信



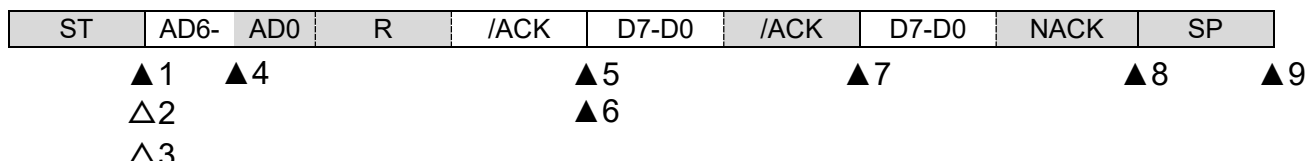
▲1 : RXI 割り込み・・・アドレス受信一致（転送方向ビット : Write）

▲2 : RXI 割り込み・・・最終データ-1 受信完了（2nd データ）

▲3 : RXI 割り込み・・・最終データ受信完了（2nd データ）

▲4 : EEI (STOP) 割り込み・・・ストップコンディション検出

6.2.6 マルチマスタ通信（マスタ送信中の AL 検出後、スレーブ送信）



▲1 : EEI (START) 割り込み・・・スタートコンディション検出

△2 : TXI 割り込み・・・スタートコンディション検出 ※処理なし

△3 : TXI 割り込み・・・送信バッファ空 ※処理なし

▲4 : EEI (AL) 割り込み・・・アービトレーションロスト検出

▲5 : TXI 割り込み・・・アドレス受信一致（転送方向ビット : Read）

▲6 : TXI 割り込み・・・送信バッファ空

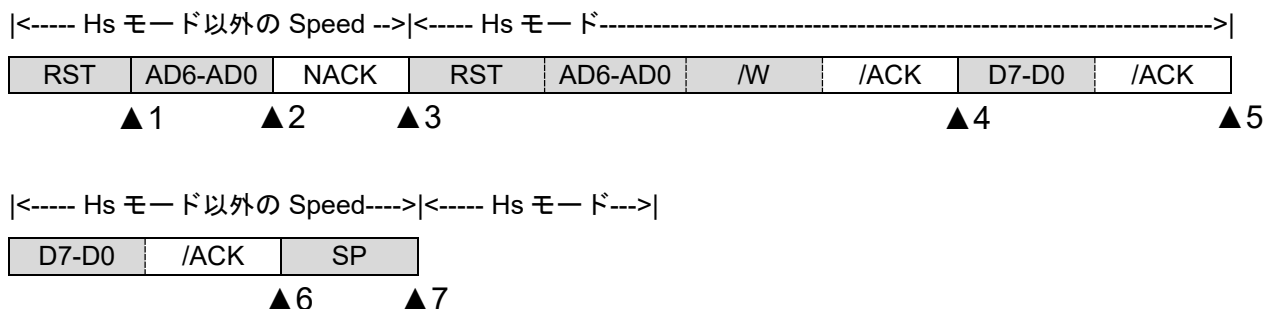
▲7 : TXI 割り込み・・・送信バッファ空

▲8 : EEI (NACK) 割り込み・・・NACK 検出

▲9 : EEI (STOP) 割り込み・・・ストップコンディション検出

6.2.7 Hs モードでのマスタ送信

1. パターン 1



▲1 : EEI (START) 割り込み・・・スタートコンディション検出

▲2 : TEI 割り込み・・・マスターコード(0000 1XXXb) 送信完了

▲3 : EEI (NACK) 割り込み・・・NACK 検出

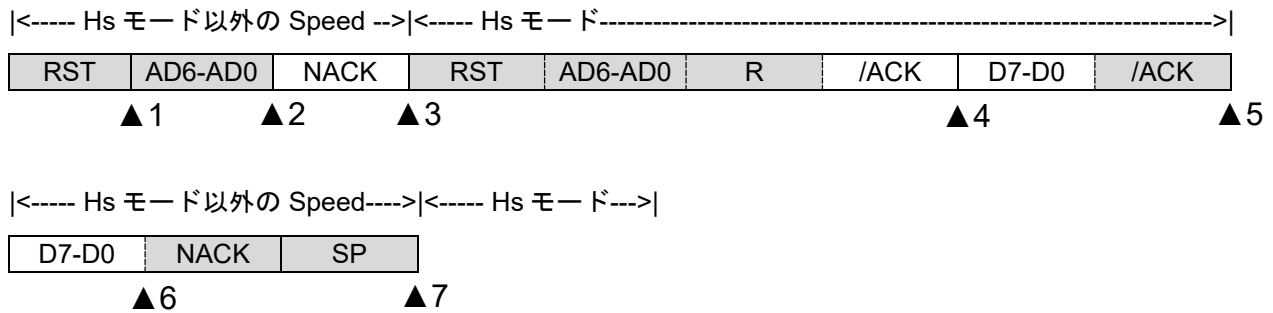
▲4 : TEI 割り込み・・・アドレス送信完了（転送方向ビット : Write）

▲5 : TEI 割り込み・・・データ送信完了（1st データ）

▲6 : TEI 割り込み・・・データ送信完了（2nd データ）

▲7 : EEI (STOP) 割り込み・・・ストップコンディション検出

6.2.8 Hs モードでのマスタ受信



- ▲1 : EEI (START) 割り込み・・・スタートコンディション検出
- ▲2 : TEI 割り込み・・・マスターコード(0000 1XXXb) 送信完了
- ▲3 : EEI (NACK) 割り込み・・・NACK 検出
- ▲4 : RXI 割り込み・・・アドレス送信完了 (転送方向ビット : Read)
- ▲5 : RXI 割り込み・・・最終データ-1 受信完了 (2nd データ)
- ▲6 : RXI 割り込み・・・最終データ受信完了 (2nd データ)
- ▲7 : EEI (STOP) 割り込み・・・ストップコンディション検出

6.3 タイムアウトの検出、および検出後の処理

6.3.1 タイムアウト検出機能によるタイムアウト検出

関数 `R_RIICHS_Open()` の引数の設定でタイムアウト検出機能を有効にした場合、コールバック関数内で `R_RIICHS_GetStatus` 関数を呼び出してください。

タイムアウト検出情報は `R_RIICHS_GetStatus` 関数の第 2 引数に設定した `riichs_mcu_status_t` 構造体変数の TMO ビットにより確認できます。

TMO ビットが“1”：タイムアウトを検出

TMO ビットが“0”：タイムアウト未検出

6.3.2 タイムアウト検出後の対応方法

タイムアウトが検出された場合は、いったん `R_RIICHS_Close` 関数を呼び出し、初期化処理の `R_RIICHS_Open` 関数から通信を再開する必要があります。

また、バスハングアップによりタイムアウトが検出される場合もあります。マスタモード時、ノイズ等の影響でスレーブデバイスとの同期ズレが発生するとスレーブデバイスが SDA ラインを Low 固定状態にする場合があります(バスハングアップ)。この状態ではストップコンディションは発行できないため、タイムアウトが検出されます。

バスハングアップから復帰するためには、SCL クロック追加出力機能を使用します。追加クロックを 1 クロックずつ出力することでスレーブデバイスによる SDA ラインの Low 固定状態を解放させ、バス状態を復帰させることができます。

追加クロックを 1 クロック出力するためには、`R_RIICHS_Control` 関数の第 2 引数に `RIICHS_GEN_SCL_ONESHOT` (SCL クロックのワンショット出力)を設定して `R_RIICHS_Control()` を呼び出してください。

また、SCL の端子状態は `R_RIICHS_GetStatus` 関数で確認できます。

SCL が High になるまで SCL クロックのワンショット出力を繰り返してください。

図 6.5 にタイムアウト検出と対応方法例(マスタ送信)を示します。

SCL クロック追加出力機能についての詳細は、各マイコンのユーザーズマニュアル ハードウェア編の RIICHS 章に記載されている「SCL クロック追加出力機能」を参照ください。

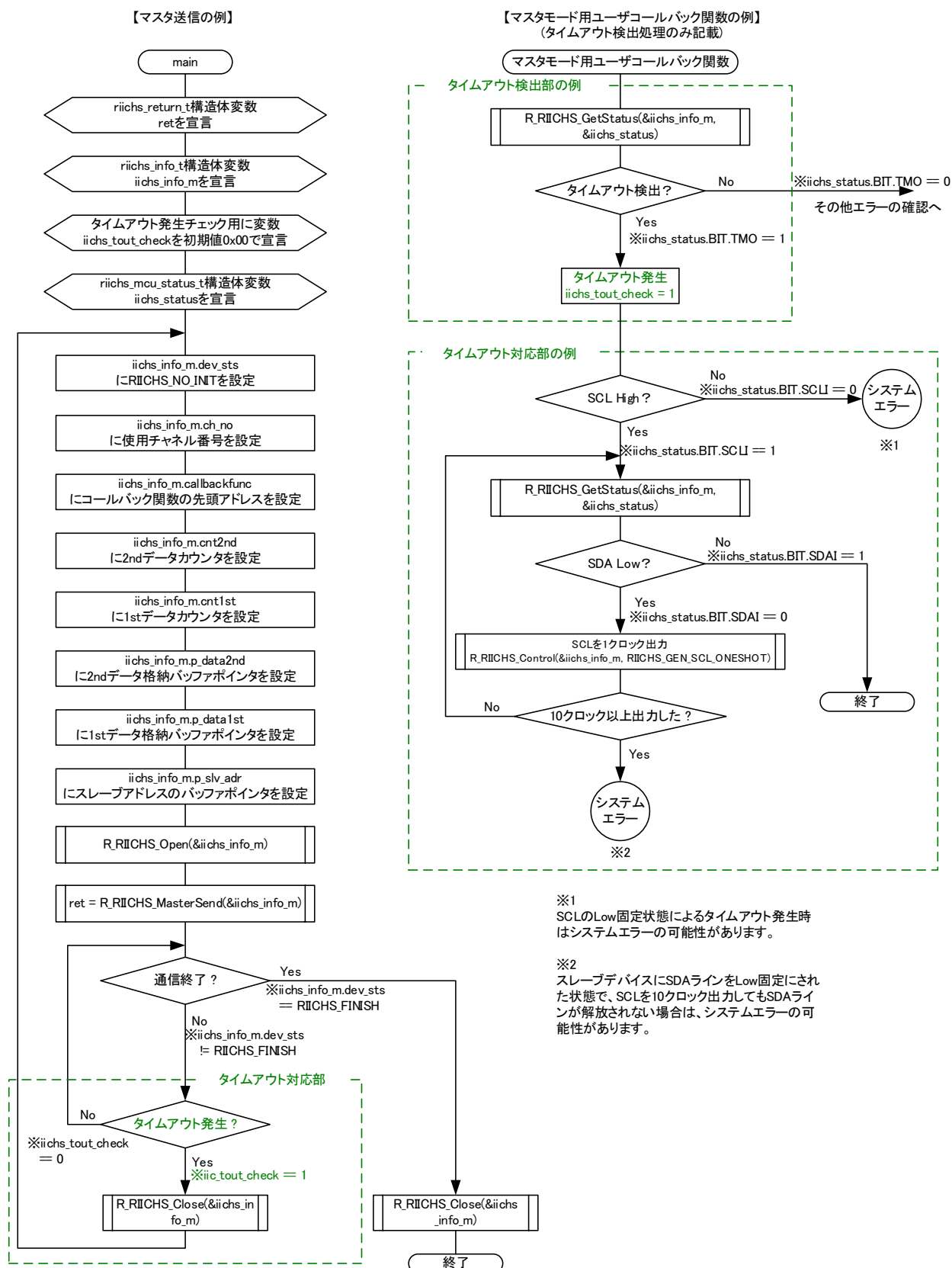


図 6.5 タイムアウト検出と対応方法例(マスタ送信)

6.4 動作確認環境

本モジュールの動作確認環境を以下に示します。

表 6.6 動作確認環境 (Rev.1.00)

項目	内容
統合開発環境	ルネサス エレクトロニクス製 e ² studio 2020-01 (21.1.0) IAR Embedded Workbench for Renesas RX 4.14.01
C コンパイラ	ルネサス エレクトロニクス製 C/C++ compiler for RX family V.3.03.00 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 -lang = c99 GCC for Renesas RX 8.03.00.202002 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 -std=gnu99 IAR C/C++ Compiler for Renesas RX version 4.14.01 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 RX671 (型名：RTK55671xxxxxxxxxx)

表 6.7 動作確認環境 (Rev.1.10)

項目	内容
統合開発環境	ルネサス エレクトロニクス製 e2 studio 2022-10 (22.10.0) 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 C/C++ compiler for RX family V.3.04.00 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 -lang = c99 GCC for Renesas RX 8.03.00.202204 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 -std=gnu99 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 RX671 (型名：RTK55671xxxxxxxxxx)

表 6.8 動作確認環境 (Rev.1.20)

項目	内容
統合開発環境	ルネサス エレクトロニクス製 e2 studio 2024-07 (24.7.0) IAR Embedded Workbench for Renesas RX 5.10.1
C コンパイラ	ルネサス エレクトロニクス製 C/C++ compiler for RX family V.3.06.00 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 -lang = c99
	GCC for Renesas RX 8.03.00.202405 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 -std=gnu99
	IAR C/C++ Compiler for Renesas RX version 5.10.1 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	なし

6.5 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"」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。プロジェクトへの追加方法をご確認ください。

- 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)」

また、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。BSP モジュールの追加方法は、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(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_riichs_rx module.」エラーが発生します。

A : 追加した FIT モジュールがユーザプロジェクトのターゲットデバイスに対応していない可能性があります。追加した FIT モジュールの対象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「ERROR - RIICHS_CFG_XXX_XXX - ...」エラーが発生します。

A : “r_riichs_rx_config.h” ファイルの設定値が間違っている可能性があります。
“r_riichs_rx_config.h” ファイルを確認して正しい値を設定してください。詳細は「2.7 コンパイル時の設定」を参照してください。

6.6 サンプルコード

6.6.1 1つのチャンネルで1つのスレーブデバイスに連続アクセスする場合の例

RIICHS の 1 つのチャンネルを使用し、1 つのスレーブデバイスに対して、連続アクセスする場合のサンプルコードを示します。

次の(1)~(6)の順に動作します。

- (1)RIICHS の ch0 を使用可能にするため、R_RIICHS_Open 関数を実行します。
- (2)EEPROM に 16 バイトのデータを書き込むため、R_RIICHS_MasterSend 関数を実行します。
- (3)EEPROM 書き込み完了を確認するために、R_RIICHS_MasterSend 関数を使用し、Acknowledge Polling を行います。
- (4)EEPROM から 16 バイトのデータを読み出すため、R_RIICHS_MasterReceive 関数を実行します。
- (5)書き込みデータと読み出しデータを比較します。
- (6)RIICHS の ch0 を RIICHS FIT モジュールから解放するため、R_RIICHS_Close 関数を実行します。

このサンプルコードは対象デバイスの Renesas Starter Kit で動作確認をしています。スレーブデバイスのアドレスは使用する EEPROM によって異なりますのでご注意ください。

```
#include <stddef.h>
#include "platform.h"
#include "r_riichs_rx_if.h"

/* EEPROM device code (fixed) */
#define EEPROM_DEVICE_CODE (0xA0)

/* Device address code (under 4 bit is A2 (Vss=0), A1 (Vcc=1), A0 (Vcc=1), and RW code)
   for hardware connection with EEPROM on RSK of the supported target device.
   Please change the following settings as necessary. */
#define EEPROM_DEVICE_ADDRESS_CODE (0x06)

/* E2PROM device address */
#define EEPROM_DEVICE_ADDRESS ((EEPROM_DEVICE_CODE | EEPROM_DEVICE_ADDRESS_CODE) >> 1)

/* variables */
static volatile riichs_return_t ret; /* Return value */
static riichs_info_t iichs_info_m; /* Structure data */

static uint8_t addr_eeprom[1] = { EEPROM_DEVICE_ADDRESS };
static uint8_t access_addr1[1] = { 0x00 };

/* This data is sent to the EEPROM when target device is the master device. */
static uint8_t master_send_data[16] =
{ 0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x8d, 0x8e, 0x8f };

/* This buffer stores data received from the slave device. */
static uint8_t master_store_area[16] =
{ 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF };

/* private functions */
static void callback_master (void);
static void eeprom_write (void);
static void acknowledge_polling (void);
static void eeprom_read (void);
```

図 6.6 1つのチャンネルで1つのスレーブデバイスに連続アクセスする例(1)

```

/*****
* Function Name: main
* Description   : The main loop
* Arguments    : none
* Return Value  : none
*****/
void main (void)
{
    uint8_t i = 0;

    /* Initialize */
    for (i = 0; i < 16; i++)
    {
        master_store_area[i] = 0xFF;
    }

    /* Set arguments for R_RIICHS_Open. */
    iichs_info_m.ch_no = 0; /* Channel number */
    iichs_info_m.dev_sts = RIICHS_NO_INIT; /* Device state flag (to be updated) */
    iichs_info_m.scl_up_time = 20E-9; /* Rise time of SCLn Line */
    iichs_info_m.scl_down_time = 20E-9; /* Fall time of SCLn Line */
    iichs_info_m.fs_scl_up_time = 20E-9; /* Rise time of SCLn Line before transition to Hs mode */
    iichs_info_m.fs_scl_down_time = 20E-9; /* Fall time of SCLn Line before transition to Hs mode */
    iichs_info_m.speed_kbps = 3400; /* RIICHS bps(kbps) */
    iichs_info_m.fs_speed_kbps = 400; /* RIICHS bps(kbps) before transition to Hs mode */
    iichs_info_m.bus_check_counter = 1000; /* software bus busy check counter */
    iichs_info_m.bus_free_time = 5; /* software bus free counter */
    iichs_info_m.slave_addr0 = 0x0025; /* Slave address 0 */
    iichs_info_m.slave_addr1 = 0x0000; /* Slave address 1 */
    iichs_info_m.slave_addr2 = 0x0000; /* Slave address 2 */
    iichs_info_m.slave_addr0_format = RIICHS_SEVEN_BIT_ADDR_FORMAT; /* Slave address 0 format */
    iichs_info_m.slave_addr1_format = RIICHS_ADDR_FORMAT_NONE; /* Slave address 1 format */
    iichs_info_m.slave_addr2_format = RIICHS_ADDR_FORMAT_NONE; /* Slave address 2 format */
    iichs_info_m.gca_enable = RIICHS_GCA_DISABLE; /* Disable General call address */
    iichs_info_m.rxi_priority = RIICHS_IPL_1; /* The priority level of the RXI */
    iichs_info_m.txi_priority = RIICHS_IPL_1; /* The priority level of the TXI */
    iichs_info_m.eei_priority = RIICHS_IPL_1; /* The priority level of the EEI */
    iichs_info_m.tei_priority = RIICHS_IPL_1; /* The priority level of the TEI */
    iichs_info_m.master_arb = RIICHS_MASTER_ARB_LOST_DISABLE; /* Disable Master Arbitration-Lost
Detection */
    iichs_info_m.filter_stage = RIICHS_DIGITAL_FILTER_0; /* digital noise filter stage */
    iichs_info_m.timeout_enable = RIICHS_TMO_ENABLE; /* Enable Timeout function */
    iichs_info_m.nack_detc_enable = RIICHS_NACK_DETC_ENABLE; /* Enable NACK Detection */
    iichs_info_m.arb_lost_enable = RIICHS_ARB_LOST_ENABLE; /* Enable Arbitration Lost */
    iichs_info_m.counter_bit = RIICHS_COUNTER_BIT16; /* 16 bit for the timeout detection time */
    iichs_info_m.l_count = RIICHS_L_COUNT_ENABLE; /* SCL line is held LOW when the timeout function
is enabled */
    iichs_info_m.h_count = RIICHS_H_COUNT_ENABLE; /* SCL line is held HIGH when the timeout function
is enabled */
    iichs_info_m.timeout_mode = RIICHS_TIMEOUT_MODE_ALL; /* Timeout Detection Mode */

    ret = R_RIICHS_Open(&iichs_info_m);
    if (RIICHS_SUCCESS != ret)
    {
        /* This software is for single master.
        Therefore, return value should be always 'RIICHS_SUCCESS'. */
        while (1)
        {
            R_BSP_NOP(); /* error */
        }
    }

    /* EEPROM Write (Master transfer) */
    eeprom_write();

    /* Acknowledge polling (Master transfer) */
    acknowledge_polling();

    /* EEPROM Read (Master transfer and Master receive) */
    eeprom_read();
}

```

図 6.7 1つのチャネルで1つのスレーブデバイスに連続アクセスする例(2)

```
/* Compare */
for (i = 0; i < 16; i++)
{
    if (master_store_area[i] != master_send_data[i])
    {
        /* Detected mismatch. */
        LED3 = LED_ON;
    }
    else
    {
        LED0 = LED_ON;
    }
}

ret = R_RIICHS_Close(&iichs_info_m);
if (RIICHS_SUCCESS != ret)
{
    /* This software is for single master.
       Therefore, return value should be always 'RIICHS_SUCCESS'. */
    while (1)
    {
        R_BSP_NOP();    /* error */
    }
}

while (1)
{
    /* do nothing */
}

} /* End of function main() */
```

図 6.8 1つのチャンネルで1つのスレーブデバイスに連続アクセスする例(3)

```

/*****
* Function Name: callback_master
* Description   : This function is sample of Master Mode callback function.
* Arguments    : none
* Return Value  : none
*****/
static void callback_master (void)
{
    riichs_mcu_status_t    iichs_status;

    ret = R_RIICHS_GetStatus(&iichs_info_m, &iichs_status);
    if (RIICHS_SUCCESS != ret)
    {
        /* This software is for single master.
           Therefore, return value should be always 'RIICHS_SUCCESS'. */
        while (1)
        {
            R_BSP_NOP();    /* error */
        }
    }
    else
    {
        /* Processing when a timeout, arbitration-lost, NACK,
           or others is detected by verifying the iichs_status flag. */
    }
} /* End of function callback_master() */

/*****
* Function Name: eeprom_write
* Description   : This function is sample of EEPROM write function using R_RIICHS_MasterSend.
* Arguments    : none
* Return Value  : none
*****/
static void eeprom_write (void)
{
    /* Set arguments for R_RIICHS_MasterSend. */
    iichs_info_m.p_slv_addr = addr_eeprom; /* Pointer to the slave address storage buffer */
    iichs_info_m.p_data1st = access_addr1; /* Pointer to the first data storage buffer */
    iichs_info_m.cnt1st = 1;                /* First data counter (number of bytes) (to be updated) */
    iichs_info_m.p_data2nd = master_send_data; /* Pointer to the second data storage buffer */
    iichs_info_m.cnt2nd = 16;              /* Second data counter (number of bytes) (to be updated) */

    /*
    iichs_info_m.callbackfunc = &callback_master; /* Callback function */

    /* Master send start. */
    ret = R_RIICHS_MasterSend(&iichs_info_m);
    if (RIICHS_SUCCESS == ret)
    {
        /* Waitting for R_RIICHS_MasterSend completed. */
        while (RIICHS_COMMUNICATION == iichs_info_m.dev_sts)
        {
            /* do nothing */
        }

        if (RIICHS_NACK == iichs_info_m.dev_sts)
        {
            /* Slave returns NACK. The slave address may not correct.
               Please check the macro definition value or hardware connection etc. */
            while (1)
            {
                R_BSP_NOP();    /* error */
            }
        }
    }
    else
    {
        /* This software is for single master.
           Therefore, return value should be always 'RIICHS_SUCCESS'. */
    }
}

```

図 6.9 1つのチャンネルで1つのスレーブデバイスに連続アクセスする例(4)

```

        while (1)
        {
            R_BSP_NOP();          /* error */

        }
    }

} /* End of function eeprom_write() */

/*****
 * Function Name: acknowledge_polling
 * Description   : This function is sample of Acknowledge Polling using R_RIICHS_MasterSend with
 *                 master send pattern 3.
 * Arguments      : none
 * Return Value   : none
 *****/
static void acknowledge_polling (void)
{
    do
    {
        /* Set arguments for R_RIICHS_MasterSend. */
        iichs_info_m.p_slv_adr = addr_eeprom;          /* Pointer to the slave address storage buffer */
        /*
        iichs_info_m.p_data1st = (uint8_t*) FIT_NO_PTR; /* Pointer to the first data storage buffer */
        */
        iichs_info_m.cnt1st = 0;                        /* First data counter (number of bytes) */
        iichs_info_m.p_data2nd = (uint8_t*) FIT_NO_PTR; /* Pointer to the second data storage buffer */
        /*
        iichs_info_m.cnt2nd = 0;                        /* Second data counter (number of bytes) */
        iichs_info_m.callbackfunc = &callback_master;  /* Callback function */

        /* Master send start. */
        ret = R_RIICHS_MasterSend(&iichs_info_m);
        if (RIICHS_SUCCESS == ret)
        {
            /* Waitting for R_RIICHS_MasterSend completed. */
            while (RIICHS_COMMUNICATION == iichs_info_m.dev_sts)
            {
                /* do nothing */
            }

            /* Slave returns NACK. Set retry interval. */
            if (RIICHS_NACK == iichs_info_m.dev_sts)
            {
                /* Waitting for retry interval 100us. */
                R_BSP_SoftwareDelay(100, BSP_DELAY_MICROSECS);
            }
        }
        else
        {
            /* This software is for single master.
            Therefore, return value should be always 'RIICHS_SUCCESS'. */
            while (1)
            {
                R_BSP_NOP();          /* error */
            }
        }
    } while (RIICHS_FINISH != iichs_info_m.dev_sts);
} /* End of function acknowledge_polling() */

```

図 6.10 1つのチャネルで1つのスレーブデバイスに連続アクセスする例(5)

```

/*****
* Function Name: eeprom_read
* Description   : This function is sample of EEPROM read function using R_RIICHS_MasterReceive.
* Arguments    : none
* Return Value  : none
*****/
static void eeprom_read (void)
{
    /* Set arguments for R_RIICHS_MasterReceive. */
    iichs_info_m.p_slv_adr = addr_eeprom;          /* Pointer to the slave address storage buffer */
    iichs_info_m.p_data1st = access_addr1;         /* Pointer to the first data storage buffer */
    iichs_info_m.cnt1st = 1;                       /* First data counter (number of bytes)(to be updated) */
    iichs_info_m.p_data2nd = master_store_area;    /* Pointer to the second data storage buffer */
    iichs_info_m.cnt2nd = 16;                      /* Second data counter (number of bytes)(to be updated) */

    /*
    iichs_info_m.callbackfunc = &callback_master; /* Callback function */

    /* Master send receive start. */
    ret = R_RIICHS_MasterReceive(&iichs_info_m);
    if (RIICHS_SUCCESS == ret)
    {
        /* Waitting for R_RIICHS_MasterSend completed. */
        while (RIICHS_COMMUNICATION == iichs_info_m.dev_sts)
        {
            /* do nothing */
        }

        if (RIICHS_NACK == iichs_info_m.dev_sts)
        {
            /* Slave returns NACK. The slave address may not correct.
            Please check the macro definition value or hardware connection etc. */
            while (1)
            {
                R_BSP_NOP();    /* error */
            }
        }
    }
    else
    {
        /* This software is for single master.
        Therefore, return value should be always 'RIICHS_SUCCESS'. */
        while (1)
        {
            R_BSP_NOP();    /* error */
        }
    }
} /* End of function eeprom_read() */

```

図 6.11 1つのチャンネルで1つのスレーブデバイスに連続アクセスする例(6)

7. 参考ドキュメント

ユーザーズマニュアル：ハードウェア

(最新版をルネサス エレクトロニクスホームページから入手してください。)

テクニカルアップデート／テクニカルニュース

(最新の情報をルネサス エレクトロニクスホームページから入手してください。)

ユーザーズマニュアル：開発環境

RX ファミリ C/C++コンパイラ CC-RX ユーザーズマニュアル (R20UT3248)

(最新版をルネサス エレクトロニクスホームページから入手してください。)

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。

改訂記録

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2021.06.30	—	初版発行
1.10	2022.12.21	71 プログラム	「6.4 動作確認環境」： Rev. 1. 10 に対応する表を追加。 riichs_bps_calc のエラー処理を修正。
1.20	2024.11.01	72 プログラム	「6.4 動作確認環境」： Rev. 1. 20 に対応する表を追加。 API 関数のコメントを Doxygen スタイルに変更。

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イやマガジンケース、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くと、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た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、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。