

RX ファミリ

簡易 I²C モジュール

Firmware Integration Technology

R01AN1691JJ0200

Rev.2.00

2016.10.1

要旨

本アプリケーションノートは、Firmware Integration Technology (FIT)を使用した簡易 I²C モジュールについて説明します。本モジュールはシリアルインタフェース(SCI)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 SCI (簡易 I²C モード) FIT モジュールと称します。

対象デバイス

以下は、この API によってサポートできるデバイスの一覧です。

- RX110、RX111、RX113 グループ
- RX130 グループ
- RX230、RX231、RX23T グループ
- RX24T グループ
- RX63N グループ
- RX64M グループ
- RX65N グループ
- RX71M 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。

関連ドキュメント

- Firmware Integration Technology ユーザーズマニュアル(R01AN1833)
- ボードサポートパッケージモジュール Firmware Integration Technology(R01AN1685)
- e²studio に組み込む方法 Firmware Integration Technology(R01AN1723)
- CS+に組み込む方法 Firmware Integration Technology(R01AN1826)

目次

1.概要	3
2.API 情報	16
3.API 関数	23
4.付録	38
5.サンプルコード	49
6.提供するモジュール	64
7.参考ドキュメント	64

1. 概要

SCI (簡易 I²C モード) FIT モジュールは、SCI を使用し、マスタデバイスがスレーブデバイスと送受信を行うための手段を提供します。SCI の簡易 I²C モードは、NXP 社が提唱する I²C バス(Inter-IC-Bus)インタフェース方式のシングルマスタモードに準拠しています。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。

- シングルマスタモードのみに対応(スレーブ送信、スレーブ受信には非対応)
- コンディショニング波形の生成
- 通信モードはスタンダードモードとファストモードに対応し、最大転送速度は 384kbps

制限事項

- DMAC、DTC と組み合わせて使用することはできません。
- 10 ビットアドレスの送信には対応していません。
- 多重割り込みには対応していません。
- コールバック関数内では R_SCI_IIC_GetStatus 関数以外の API 関数のコールは禁止です。
- 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 SCI (簡易 I²C モード) 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、2.10 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_SCI_IIC_Open()	この関数は SCI(簡易 I ² C モード) 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実行される必要があります。
R_SCI_IIC_MasterSend()	マスタ送信を開始します。引数に合わせて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_SCI_IIC_MasterReceive()	マスタ受信を開始します。引数に合わせて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_SCI_IIC_Close()	簡易 I ² C の通信を終了し、使用していた SCI の対象チャネルを解放します。
R_SCI_IIC_GetStatus()	本モジュールの状態を返します。
R_SCI_IIC_Control()	各コンディション出力、NACK 出力、SSCL クロックのワンショット出力、および本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_SCI_IIC_GetVersion()	本モジュールのバージョンを返します。

1.3 SCI (簡易 I²C モード) FIT モジュールの概要

1.3.1 SCI (簡易 I²C モード) FIT モジュールの仕様

- 1) 本モジュールでは、マスタ送信、マスタ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 マスタ送信の処理」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 マスタ受信の処理」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、データ受信、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。SCI(簡易 I²C モード)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) 本モジュールは、複数のチャンネルを制御することができます。また複数チャンネルを持つデバイスでは複数チャンネルを使用して同時に通信することができます。
- 4) 1つのチャンネル・バス上の複数の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。

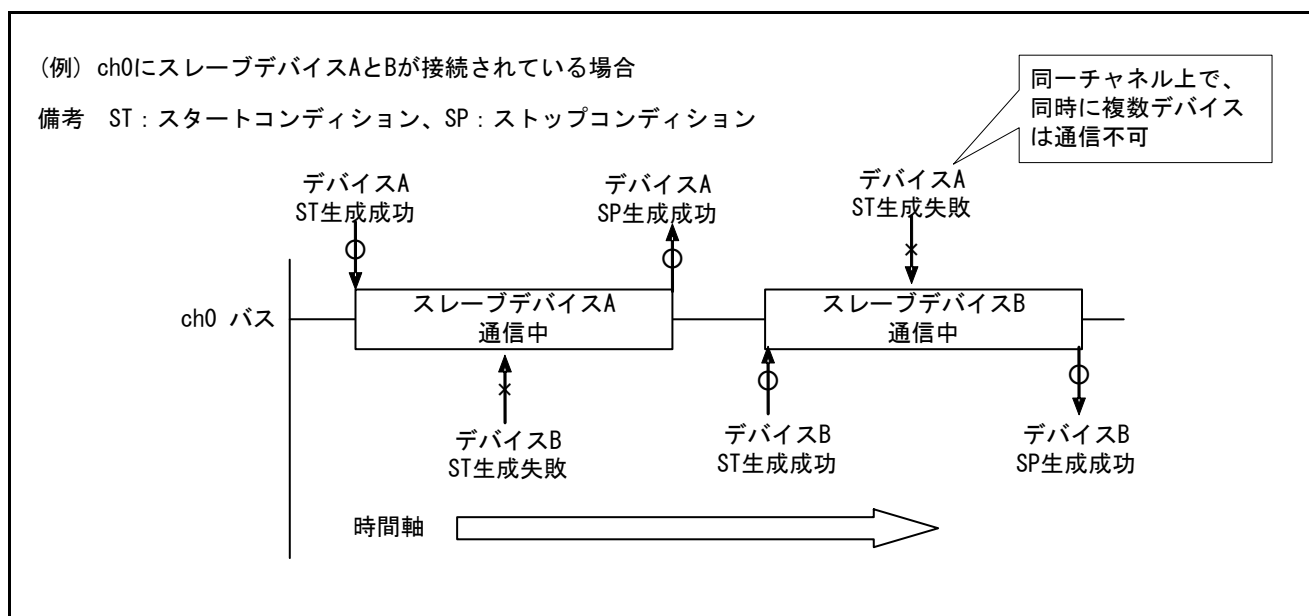


図 1.1 複数スレーブデバイス制御例

1.3.2 マスタ送信の処理

マスタデバイス (RX MCU) からスレーブデバイスへデータを送信します。

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。I²C 通信情報構造体の詳細は、「2.8 引数」を参照してください。図 1.2～図 1.5 に 4 種類の送信パターンを示します。

(1) パターン 1

マスタデバイス(RX MCU)からスレーブデバイスへデータを送信します。

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。

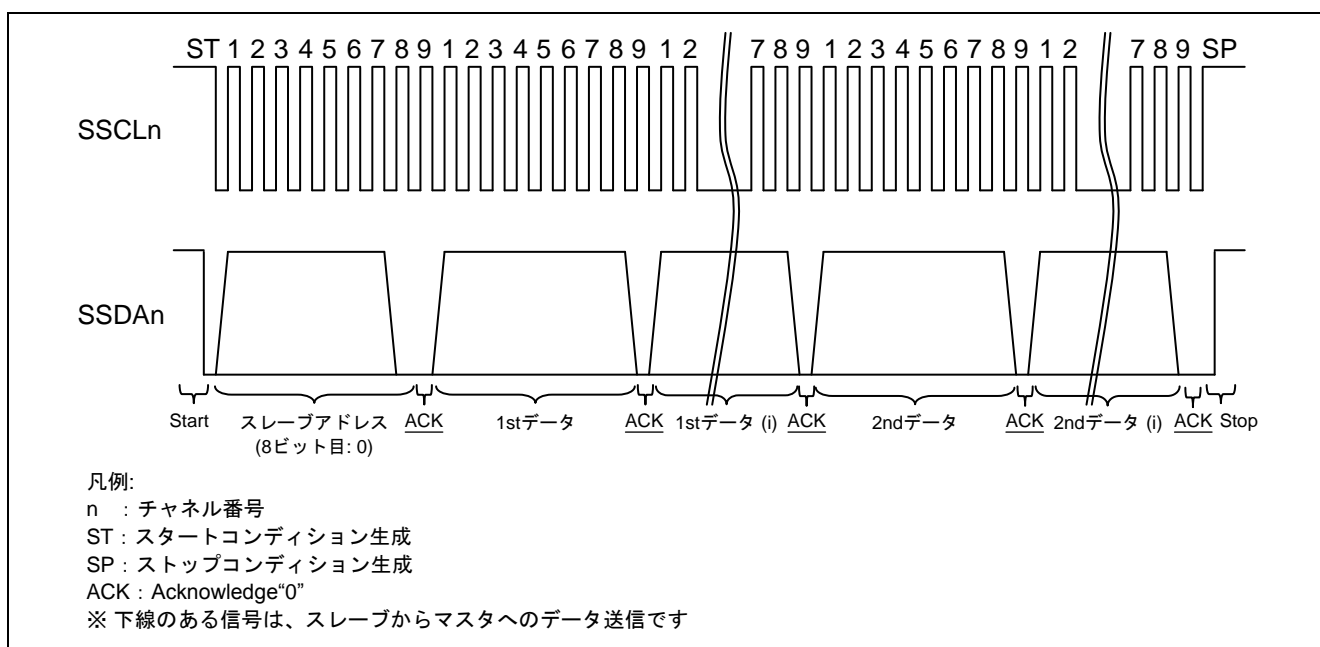


図 1.2 マスタ送信(パターン 1)信号図

(2) パターン 2

マスタ(RX MCU)からスレーブデバイスへデータを送信します。ただし、1st データを設定していない場合、1st データは送信しません。

スタートコンディション(ST)を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。

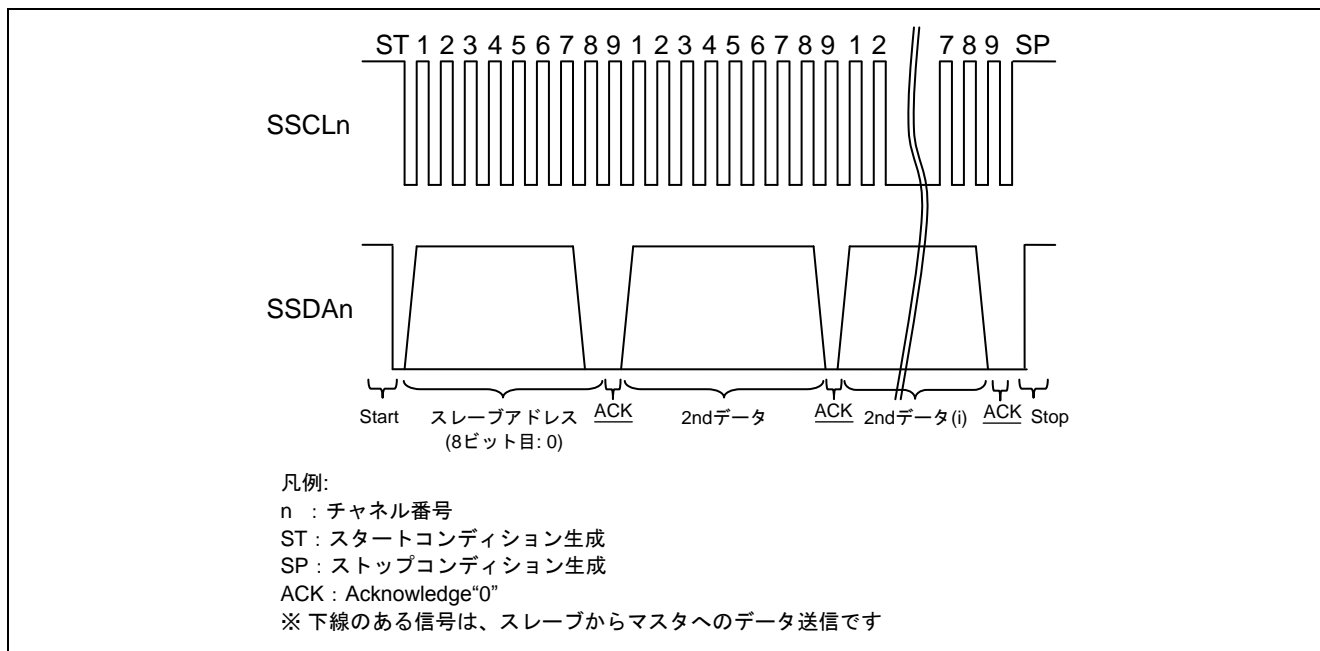


図 1.3 マスタ送信(パターン 2) 信号図

(3) パターン 3

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。

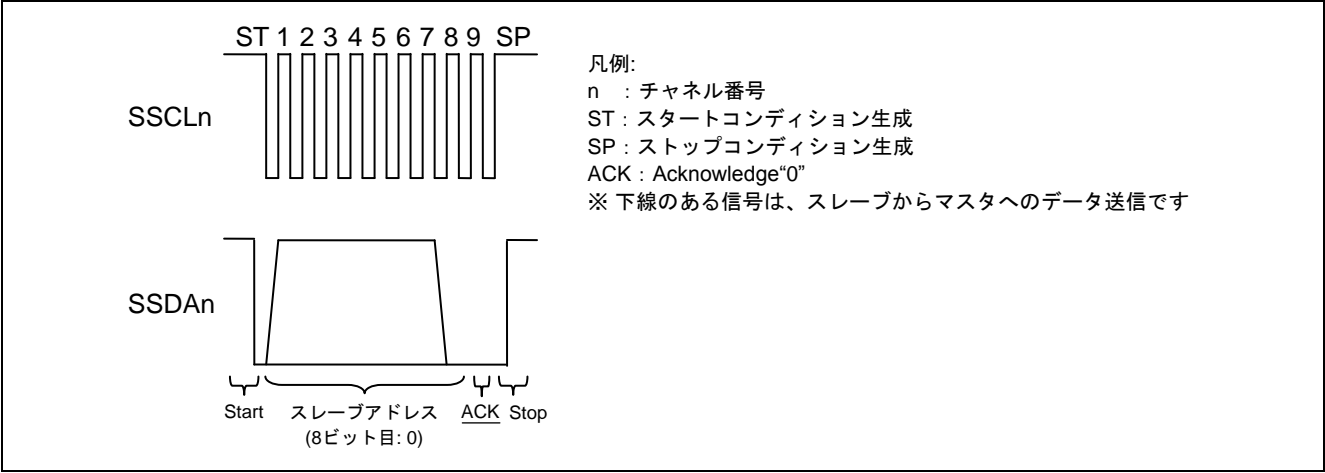


図 1.4 マスタ送信(パターン 3)信号図

(4) パターン 4

スタートコンディション(ST)を生成後、スレーブアドレスと 1st データ/2nd データを設定していない場合、スレーブアドレス送信とデータの送信は行わず、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。バス解放のみを行いたい場合に有効な処理です。

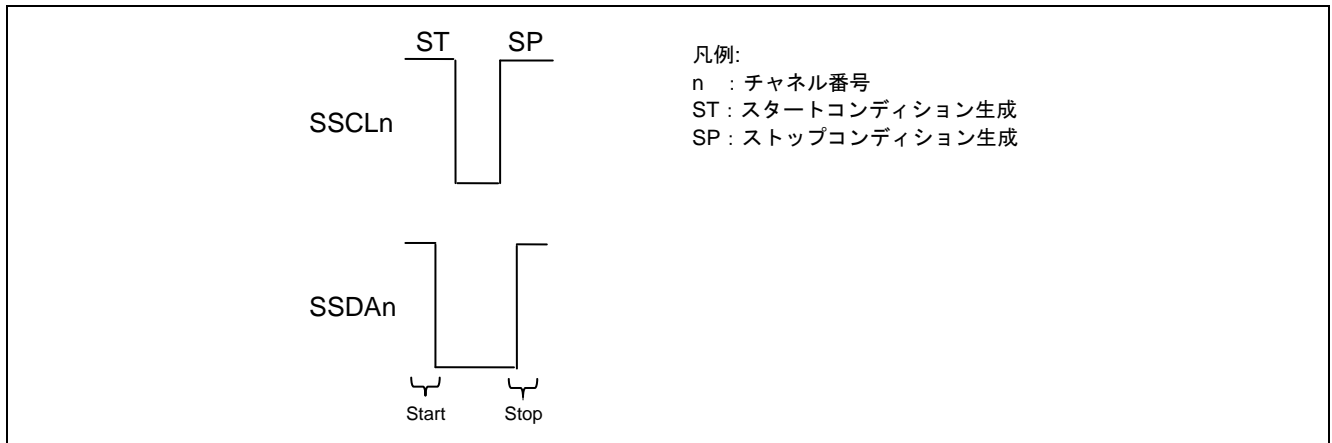


図 1.5 マスタ送信(パターン 4) 信号図

図 1.6 にマスタ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。

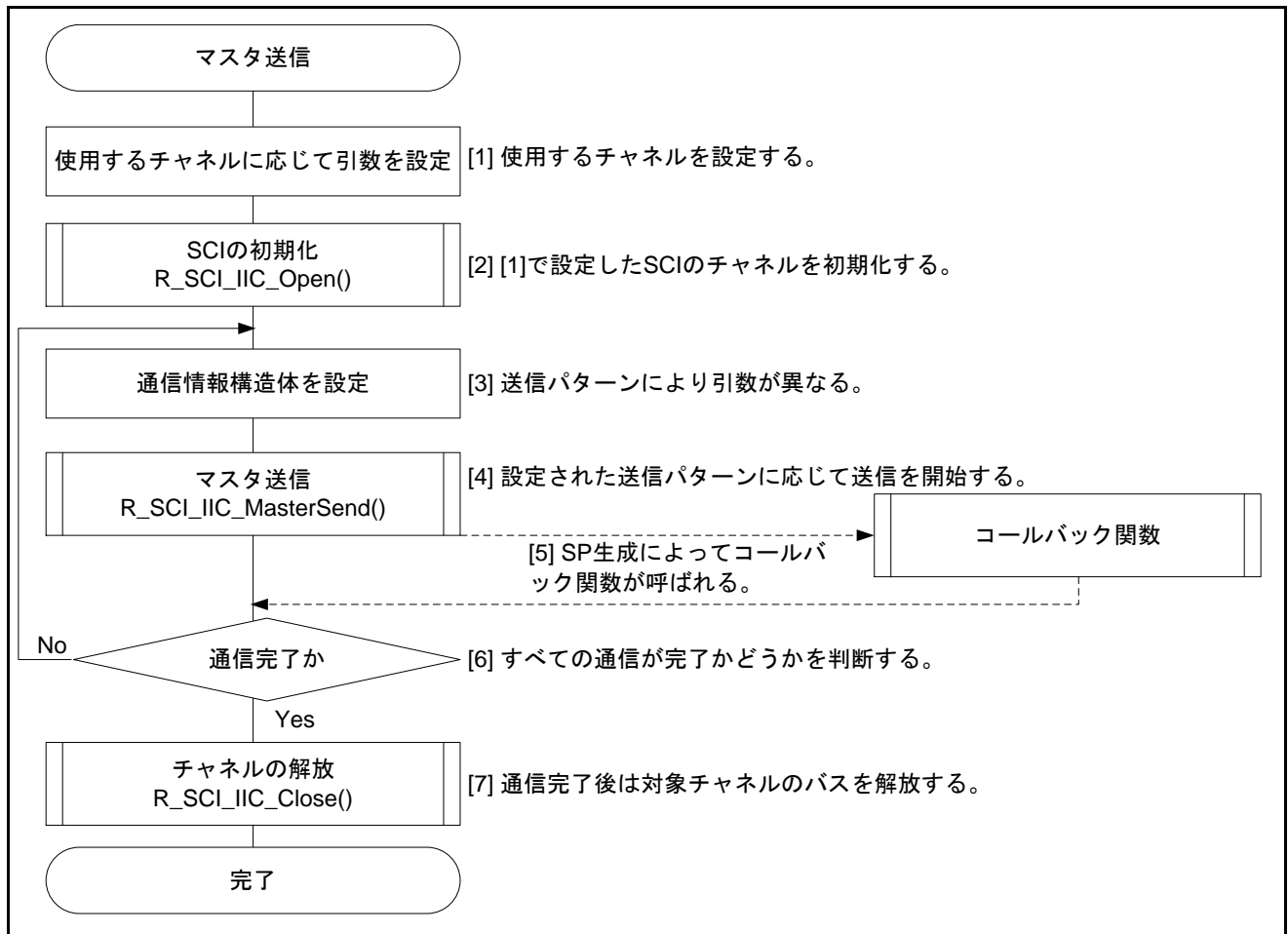


図 1.6 マスタ送信の処理例

1.3.3 マスタ受信の処理

マスタデバイス(RX MCU)は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。I²C 通信情報構造体の詳細は、「2.8 引数」を参照してください。図 1.7～図 1.8 に受信パターンを示します。

(1) マスタ受信

マスタ(RX MCU)はスレーブデバイスからデータを受信します。

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。

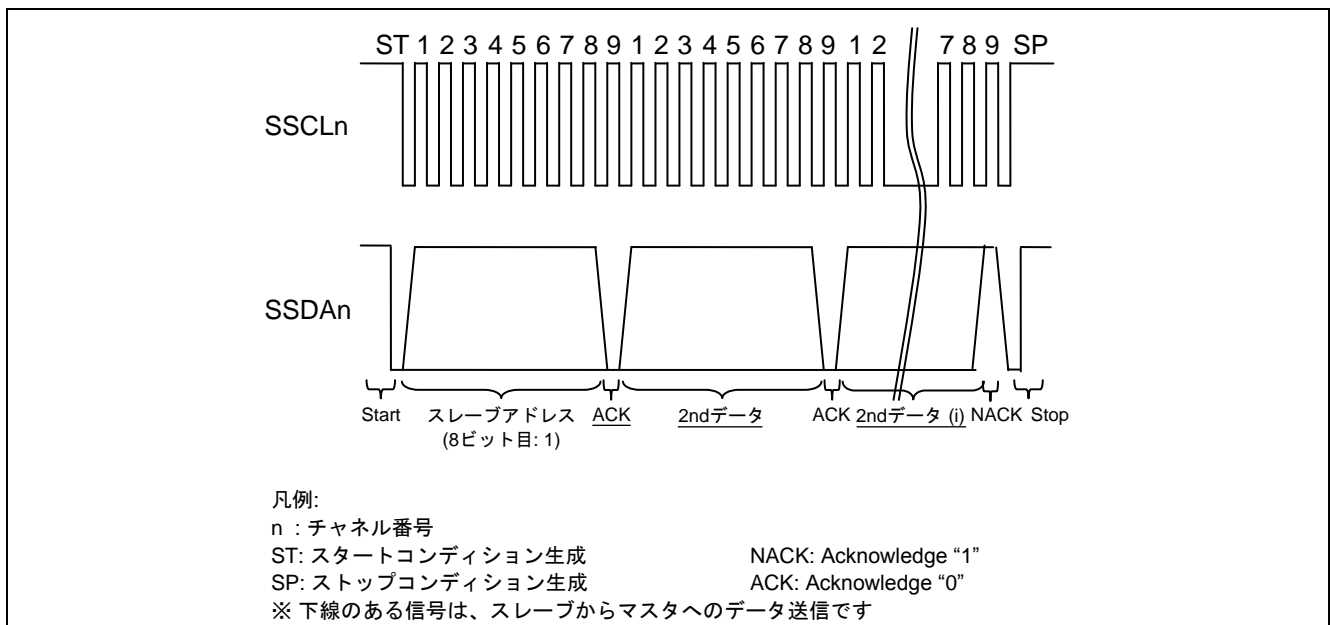


図 1.7 マスタ受信 信号図

(2) マスタ送受信

マスタ(RX MCU)からスレーブデバイスへデータを送信します(マスタ送信)。送信完了後、リスタートコンディションを生成し、転送方向を“1”(Read)に変更して、マスタはスレーブデバイスからデータを受信します(マスタ受信)。

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。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。

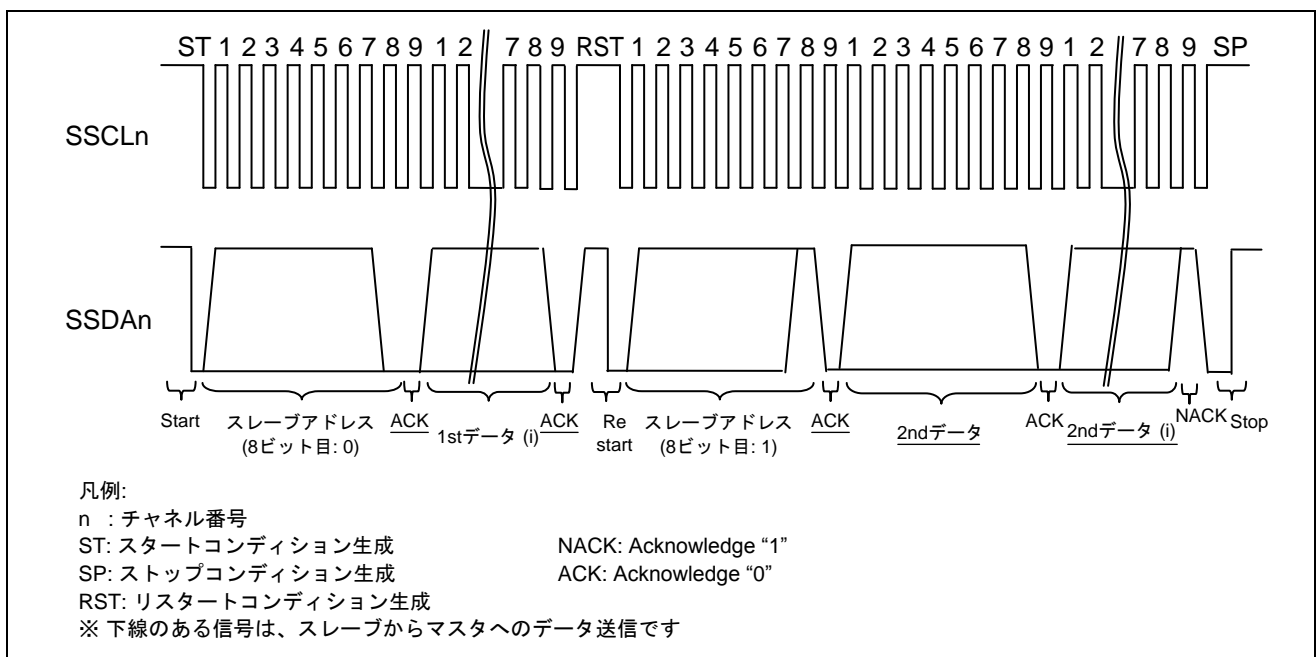


図 1.8 マスタ送受信 信号図

図 1.9 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。

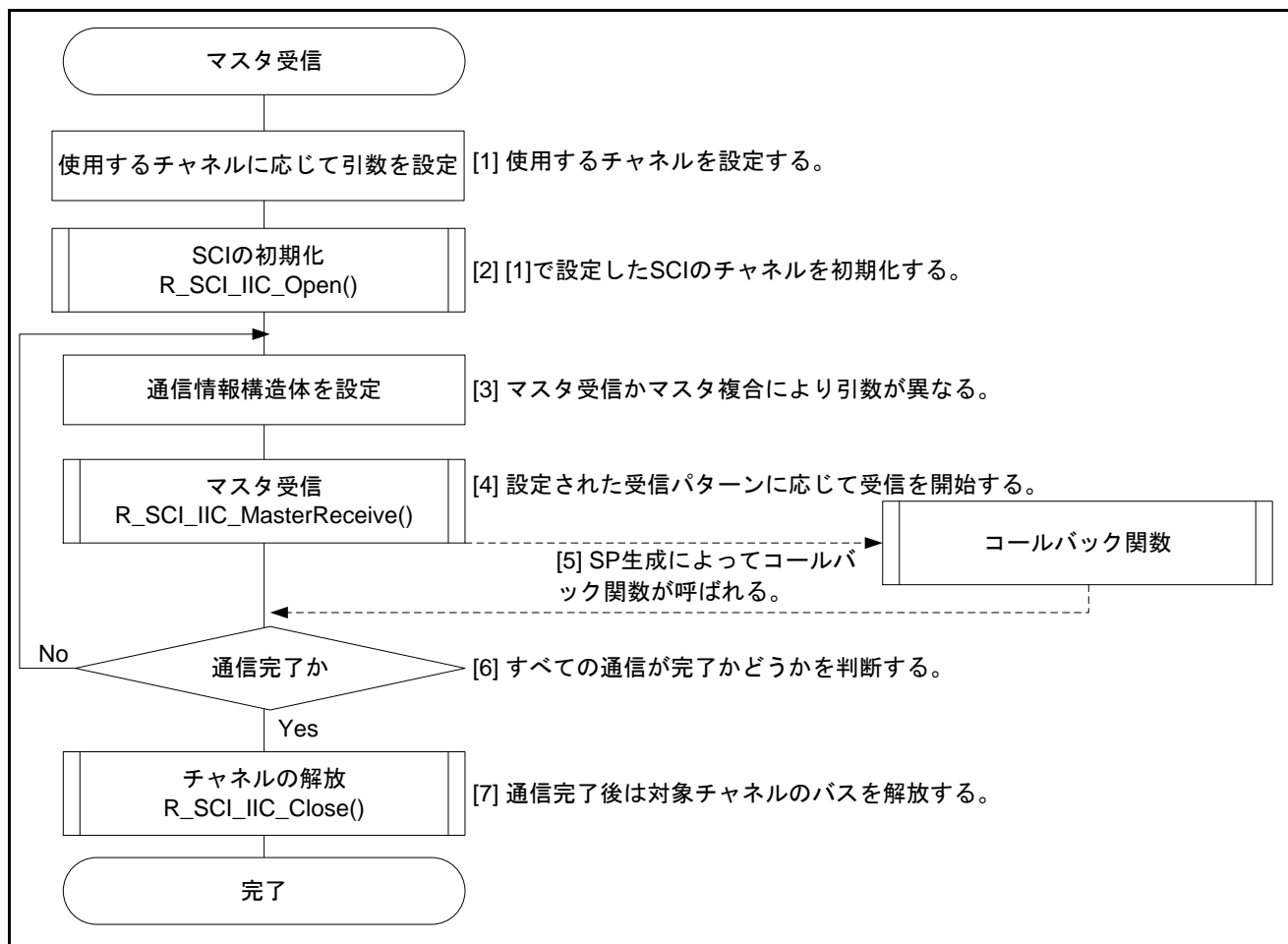


図 1.9 マスタ受信の処理例

1.3.4 状態遷移図

本モジュールを使用する場合、未初期化状態、アイドル状態、通信中のいずれかの状態になります。
図 1.10 に、状態遷移図を示します。

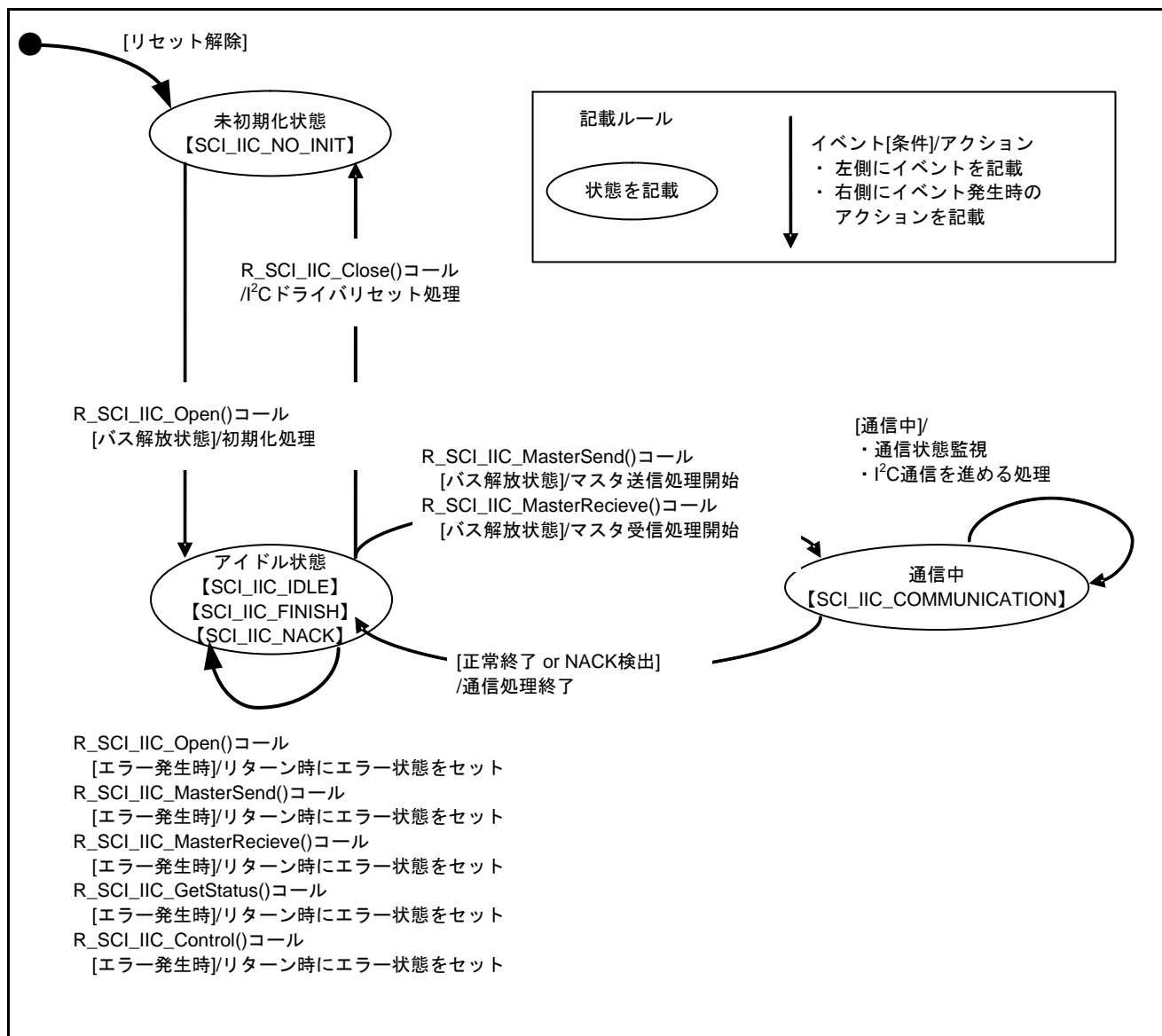


図 1.10 状態遷移図

1.3.5 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。表 1.3 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 1.3 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	SCI_IIC_NO_INIT
アイドル状態	SCI_IIC_IDLE SCI_IIC_FINISH SCI_IIC_NACK
通信中(マスタ送信)	SCI_IIC_COMMUNICATION
通信中(マスタ受信)	SCI_IIC_COMMUNICATION
通信中(マスタ送受信)	SCI_IIC_COMMUNICATION
エラー	SCI_IIC_ERROR

2. API 情報

本モジュールの API はルネサスの API の命名基準に従っています。

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。

- SCI

2.2 ソフトウェアの要求

このドライバは以下のパッケージに依存しています。

- r_bsp

2.3 サポートされているツールチェーン

このドライバは下記ツールチェーンで動作確認を行っています。

- Renesas RX Toolchain v.2.03.00 (RX65N グループ以外の対象デバイスグループ)
- Renesas RX Toolchain v.2.05.00 (RX65N グループ)

2.4 ヘッダファイル

すべての API 呼び出しと使用されるインタフェース定義は `r_sci_iic_rx_if.h` に記載しています。

2.5 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。

2.6 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_sci_iic_rx_config.h`、`r_sci_iic_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_sci_iic_rx_config.h</code> (1/2)	
SCI_IIC_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” 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。
SCI_IIC_CFG_CHI_INCLUDED <i>i</i> =0~12 ※ <i>i</i> = 1 のデフォルト値は “1” ※ <i>i</i> = 0, 2~12 のデフォルト値は “0”	該当チャネルを使用するかを選択できます。 “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。
SCI_IIC_CFG_CHI_BITRATE_BPS <i>i</i> =0~12 ※デフォルト値は全て “384000”	ビットレートを設定してください。 384000(384kbit/s)以下を設定してください。
SCI_IIC_CFG_CHI_INT_PRIORITY <i>i</i> =0~12 ※デフォルト値は全て “2”	コンディション割り込み、受信割り込み、送信空割り込み、送信完了割り込みの優先レベルを設定してください。 “1” ~ “15” の範囲で設定してください。
SCI_IIC_CFG_CHI_DIGITAL_FILTER <i>i</i> =0~12 ※デフォルト値は全て “1”	SSCL、SSDA 入力信号のノイズ除去機能を使用するか選択できます。 “0” の場合、ノイズ除去機能を無効にします。 “1” の場合、ノイズ除去機能を有効にします。
SCI_IIC_CFG_CHI_FILTER_CLOCK <i>i</i> =0~12 ※デフォルト値は全て “1”	デジタルノイズフィルタのサンプリングクロックを選択します。 “1” の場合、1分周のクロックをノイズフィルタに使用します。 “2” の場合、2分周のクロックをノイズフィルタに使用します。 “3” の場合、4分周のクロックをノイズフィルタに使用します。 “4” の場合、8分周のクロックをノイズフィルタに使用します。
SCI_IIC_CFG_CHI_SSDA_DELAY_SELECT <i>i</i> =0~12 ※デフォルト値は全て “18”	SSCL端子出力の立ち下がりに対するSSDA端子出力の遅延を選択します。“1” ~ “31” の範囲で設定してください。 デフォルト値は、内蔵ポーレートジェネレータのクロックソースであるPCLKの周波数が60MHzの場合を基準とした値になっています。 SSDAの遅延時間は内蔵ポーレートジェネレータのクロックソースに応じて遅延時間が増減します。 ビットレートを低速に設定した場合、もしくはPCLKの周波数を低速に設定した場合は、スタートコンディションにおいてSSDAの立下りタイミングがSSCLの立下りタイミングより後に発生することがあります。 システムに応じて、本設定値を見直してください。

Configuration options in *r_sci_iic_rx_config.h* (2/2)

SCI_IIC_CFG_BUS_CHECK_COUNTER <i>i</i> =0~12 ※デフォルト値は “1000”	簡易I ² C のAPI 関数のバスチェック処理時の、タイムアウトカウンタ (バス確認回数)を設定できます。 “0xFFFFFFFF” 以下の値を設定してください。 バスチェック処理は、 ・簡易I ² C 制御機能(R_SCI_IIC_Control 関数)を使用した 各コンディション生成処理実行後に行います。 バスチェック処理では、各コンディション生成処理実行後、タイムアウトカウンタをデクリメントします。“0” になるとタイムアウトと判断し、戻り値でエラー (Busy) を返します。 ※バスロックなどでロックされないようにするためのカウンタであるため、相手デバイスがSCL 端子を “L” ホールドする時間以上になるよう値を設定してください。 「タイムアウト時間(ns) \div (1 / ICLK(Hz)) * カウンタ値 * 10」
SCI_IIC_CFG_PORT_SETTING_PROCESSING ※デフォルト値は “1”	R_SCI_IIC_CFG_SCLi_SSCLi_PORT、 R_SCI_IIC_CFG_SCLi_SSCLi_BIT、 R_SCI_IIC_CFG_SCLi_SSDAi_PORT、 R_SCI_IIC_CFG_SCLi_SSDAi_BIT で選択したポートを SSCL、SSDA 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選択できます。 “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。

Configuration options in <i>r_sci_iic_rx_pin_config.h</i>	
R_SCI_IIC_CFG_SCIi_SSCLi_PORT <i>i</i> =0~12 ※ <i>i</i> = 0 のデフォルト値は'2' ※ <i>i</i> = 1 のデフォルト値は'1' ※ <i>i</i> = 2 のデフォルト値は'5' ※ <i>i</i> = 3 のデフォルト値は'2' ※ <i>i</i> = 4 のデフォルト値は'B' ※ <i>i</i> = 5 のデフォルト値は'C' ※ <i>i</i> = 6 のデフォルト値は'B' ※ <i>i</i> = 7 のデフォルト値は'9' ※ <i>i</i> = 8 のデフォルト値は'C' ※ <i>i</i> = 9 のデフォルト値は'B' ※ <i>i</i> = 10 のデフォルト値は'8' ※ <i>i</i> = 11 のデフォルト値は'7' ※ <i>i</i> = 12 のデフォルト値は'E'	SSCL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_SCI_IIC_CFG_SCIi_SSCLi_BIT <i>i</i> =0~12 ※ <i>i</i> = 0 のデフォルト値は'1' ※ <i>i</i> = 1 のデフォルト値は'5' ※ <i>i</i> = 2 のデフォルト値は'2' ※ <i>i</i> = 3 のデフォルト値は'5' ※ <i>i</i> = 4 のデフォルト値は'0' ※ <i>i</i> = 5 のデフォルト値は'2' ※ <i>i</i> = 6 のデフォルト値は'0' ※ <i>i</i> = 7 のデフォルト値は'2' ※ <i>i</i> = 8 のデフォルト値は'6' ※ <i>i</i> = 9 のデフォルト値は'6' ※ <i>i</i> = 10 のデフォルト値は'1' ※ <i>i</i> = 11 のデフォルト値は'6' ※ <i>i</i> = 12 のデフォルト値は'2'	SSCL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_SCI_IIC_CFG_SCIi_SSDAi_PORT <i>i</i> =0~12 ※ <i>i</i> = 0 のデフォルト値は'2' ※ <i>i</i> = 1 のデフォルト値は'1' ※ <i>i</i> = 2 のデフォルト値は'5' ※ <i>i</i> = 3 のデフォルト値は'2' ※ <i>i</i> = 4 のデフォルト値は'B' ※ <i>i</i> = 5 のデフォルト値は'C' ※ <i>i</i> = 6 のデフォルト値は'B' ※ <i>i</i> = 7 のデフォルト値は'9' ※ <i>i</i> = 8 のデフォルト値は'C' ※ <i>i</i> = 9 のデフォルト値は'B' ※ <i>i</i> = 10 のデフォルト値は'8' ※ <i>i</i> = 11 のデフォルト値は'7' ※ <i>i</i> = 12 のデフォルト値は'E'	SSDA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_SCI_IIC_CFG_SCIi_SSDAi_BIT <i>i</i> =0~12 ※ <i>i</i> = 0 のデフォルト値は'0' ※ <i>i</i> = 1 のデフォルト値は'6' ※ <i>i</i> = 2 のデフォルト値は'0' ※ <i>i</i> = 3 のデフォルト値は'3' ※ <i>i</i> = 4 のデフォルト値は'1' ※ <i>i</i> = 5 のデフォルト値は'3' ※ <i>i</i> = 6 のデフォルト値は'1' ※ <i>i</i> = 7 のデフォルト値は'0' ※ <i>i</i> = 8 のデフォルト値は'7' ※ <i>i</i> = 9 のデフォルト値は'7' ※ <i>i</i> = 10 のデフォルト値は'2' ※ <i>i</i> = 11 のデフォルト値は'7' ※ <i>i</i> = 12 のデフォルト値は'1'	SSDA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。

2.7 コードサイズ

本モジュールのコードサイズを下表に示します。RX100 シリーズ、RX200 シリーズ、RX600 シリーズから代表して 1 デバイスずつ掲載しています。

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「2.6 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。

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		備考
			パラメータチェック 処理あり	パラメータチェック 処理なし	
RX130	ROM	1 チャンネル使用	4358 バイト	4241 バイト	
		2 チャンネル使用	4506 バイト	4389 バイト	ROM サイズは次の計算式で計算することができます。1 チャンネル使用量 + (148 バイト × 追加チャンネル数)
	RAM	1 チャンネル使用	41 バイト		
		2 チャンネル使用	69 バイト		RAM サイズは次の計算式で計算することができます。1 チャンネル使用量 + (28 バイト × 追加チャンネル数)
	最大使用スタックサイズ		256 バイト		多重割り込み禁止のため 1 チャンネル使用時の最大値のみを記載しています。
RX231	ROM	1 チャンネル使用	4341 バイト	4224 バイト	
		2 チャンネル使用	4489 バイト	4372 バイト	ROM サイズは次の計算式で計算することができます。1 チャンネル使用量 + (148 バイト × 追加チャンネル数)
	RAM	1 チャンネル使用	41 バイト		
		2 チャンネル使用	69 バイト		RAM サイズは次の計算式で計算することができます。1 チャンネル使用量 + (28 バイト × 追加チャンネル数)
	最大使用スタックサイズ		232 バイト		多重割り込み禁止のため 1 チャンネル使用時の最大値のみを記載しています。
RX64M	ROM	1 チャンネル使用	4367 バイト	4250 バイト	
		2 チャンネル使用	4513 バイト	4396 バイト	ROM サイズは次の計算式で計算することができます。1 チャンネル使用量 + (146 バイト × 追加チャンネル数)
	RAM	1 チャンネル使用	41 バイト		
		2 チャンネル使用	69 バイト		RAM サイズは次の計算式で計算することができます。1 チャンネル使用量 + (28 バイト × 追加チャンネル数)
	最大使用スタックサイズ		240 バイト		多重割り込み禁止のため 1 チャンネル使用時の最大値のみを記載しています。

2.8 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_sci_iic_rx_if.h` で記載されています。

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(SCI_IIC_COMMUNICATION)に構造体の内容を書き換えないでください。

```
typedef struct
{
    uint8_t      rsv2;           /* 予約領域 */
    uint8_t      rsv1;           /* 予約領域 */
    sci_iic_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 */
    sci_iic_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) */
    uint32_t      cnt1st;         /* 1st データカウンタ(バイト数) */
    uint8_t *     p_data2nd;      /* 2nd データ格納バッファポインタ */
    uint8_t *     p_data1st;      /* 1st データ格納バッファポインタ */
    uint8_t *     p_slv_adr;      /* スレーブアドレスのバッファポインタ */
} sci_iic_info_t;
```

2.9 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_sci_iic_rx_if.h` で記載されています。

```
typedef enum                                /* 簡易 I2C バス API のステータスコード */
{
    SCI_IIC_SUCCESS,                       /* 問題なく処理が終了した場合 */
    SCI_IIC_ERR_LOCK_FUNC,                  /* 同じチャンネルに対して多重コールされた場合 */
    SCI_IIC_ERR_INVALID_CHAN,              /* 存在しないチャンネルの場合 */
    SCI_IIC_ERR_INVALID_ARG,               /* 不正な引数の場合 */
    SCI_IIC_ERR_NO_INIT,                   /* 未初期化状態 */
    SCI_IIC_ERR_BUS_BUSY,                  /* バスビジー 以下の場合に発生します。 */
                                           /* 通信中に初期化関数または各開始関数をコールした場合 */
                                           /* 同一チャンネル上の他のデバイスが通信中に、 */
                                           /* 各開始関数またはアドバンス関数をコールした場合 */
    SCI_IIC_ERR_OTHER                      /* その他エラー */
} sci_iic_return_t;
```

2.10 モジュールの追加方法

本モジュールは、**e² studio** で、使用するプロジェクトごとに追加する必要があります。

プロジェクトへの追加方法は、**FIT** プラグインを使用する方法と、手動で追加する方法があります。

FIT プラグインを使用すると、簡単にプロジェクトに **FIT** モジュールを追加でき、また、自動的にインクルードファイルパスが更新されます。このため、プロジェクトへ **FIT** モジュールを追加する際は、**FIT** プラグインの使用を推奨します。

FIT プラグインを使用して **FIT** モジュールを追加する方法は、アプリケーションノート「**e²studio** に組み込む方法(R01AN1723)」の「**3. FIT プラグインを使用して FIT モジュールをプロジェクトに追加する方法**」を参照してください。**FIT** プラグインを使用せず手動で **FIT** モジュールを追加する方法は、「**4. 手作業で FIT モジュールをプロジェクトに追加する方法**」を参照してください。

FIT モジュールを使用する場合、**BSP FIT** 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 FIT モジュールの詳細については、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。

3. API 関数

3.1 R_SCI_IIC_Open()

この関数は SCI(簡易 I²C モード) 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実行される必要があります。

Format

```
sci_iic_return_t R_SCI_IIC_Open(
    sci_iic_info_t    * p_sci_iic_info    /* 構造体データ */
)
```

Parameters

* *p_sci_iic_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 2.8 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(SCI_IIC_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。

```
sci_iic_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 */
```

Return Values

SCI_IIC_SUCCESS	/* 問題なく処理が完了した場合 */
SCI_IIC_ERR_LOCK_FUNC	/* 他のタスクが API をロックしている場合 */
SCI_IIC_ERR_INVALID_CHAN	/* 存在しないチャンネルの場合 */
SCI_IIC_ERR_INVALID_ARG	/* 不正な引数の場合 */
SCI_IIC_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_sci_iic_rx_if.h にプロトタイプ宣言されています。

Description

簡易 I²C バス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 SCI 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 (SCI_IIC_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
- －SCI 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
- －簡易 I²C バス通信で使用する SCI レジスタの初期化
- －SCI 割り込みの禁止

Reentrant

- 異なるチャンネルからリエントラントは可能です。

Example

```
volatile sci_iic_return_t ret;  
sci_iic_info_t          siic_info;  
  
siic_info.dev_sts = SCI_IIC_NO_INIT;  
siic_info.ch_no = 1;  
  
ret = R_SCI_IIC_Open(&siic_info);
```

Special Notes:

なし

3.2 R_SCI_IIC_MasterSend()

マスタ送信を開始します。引数に合わせて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

```
sci_iic_return_t R_SCI_IIC_MasterSend(
    sci_iic_info_t * p_sci_iic_info /* 構造体データ */
)
```

Parameters

* *p_sci_iic_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 2.8 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(SCI_IIC_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。

```
uint8_t * p_slv_adr; /* スレーブアドレスのバッファポインタ */
uint8_t * p_data1st; /* 1st データ格納バッファポインタ (更新あり) */
uint8_t * p_data2nd; /* 2nd データ格納バッファポインタ (更新あり) */
sci_iic_ch_dev_status_t dev_sts; /* デバイス状態フラグ (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 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) */
sci_iic_callback callbackfunc; /* コールバック関数 */
uint8_t ch_no; /* チャンネル番号 */
```

Return Values

```
SCI_IIC_SUCCESS /* 問題なく処理が完了した場合 */
SCI_IIC_ERR_INVALID_CHAN /* 存在しないチャンネルの場合 */
SCI_IIC_ERR_INVALID_ARG /* 不正な引数の場合 */
SCI_IIC_ERR_NO_INIT /* 初期設定ができていない場合 (未初期化状態) */
SCI_IIC_ERR_BUS_BUSY /* バスビジーの場合 */
SCI_IIC_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_sci_iic_rx_if.h にプロトタイプ宣言されています。

Description

簡易 I²C バスのマスタ送信を開始します。引数で指定した SCI 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”(SCI_IIC_IDLE)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
- － SCI 割り込みの許可
- － I²C のリセット解除
- － I²C 出力ポートの割り当て

ー スタートコンディションの生成

Reentrant

- 異なるチャネルからリエントラントは可能です。

Example

– Case1: 送信パターン 1

```
#include <stddef.h>          // NULL definition
#include "platform.h"
#include "r_sci_iic_rx_if.h"

void main(void);
void Callback_ch1(void);

void main(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_info_t          siic_info;

    uint8_t slave_addr_eeprom[1] = {0x50}; /* Slave address for EEPROM */
    uint8_t access_addr1[1]      = {0x00}; /* 1st data field */
    uint8_t send_data[5]         = {0x81, 0x82, 0x83, 0x84, 0x85};

    /* Sets IIC Information (Send pattern 1) */
    siic_info.p_slv_adr      = slave_addr_eeprom;
    siic_info.p_data1st      = access_addr1;
    siic_info.p_data2nd      = send_data;
    siic_info.dev_sts        = SCI_IIC_NO_INIT;
    siic_info.cnt1st         = 1;
    siic_info.cnt2nd         = 3;
    siic_info.callbackfunc    = &Callback_ch1;
    siic_info.ch_no          = 1;

    /* SCI open */
    ret = R_SCI_IIC_Open(&siic_info);
    /* Start Master Send */
    ret = R_SCI_IIC_MasterSend(&siic_info);
    while(1);
}

void Callback_ch1(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_mcu_status_t      iic_status;
    sci_iic_info_t            iic_info_ch;

    iic_info_ch.ch_no = 1;
    ret = R_SCI_IIC_GetStatus(&iic_info_ch, &iic_status);

    if (SCI_IIC_SUCCESS != ret)
    {
        /* R_SCI_IIC_GetStatus() 関数コールエラー処理 */
    }
    else
    {
        if (1 == iic_status.BIT.NACK)
        {
            /* iic_status のステータスフラグを確認して
             NACK が検出されていた場合の処理 */
        }
    }
}
```

```

    }
}
}

```

- Case2: 2つのスレーブデバイス(スレーブ 1、スレーブ 2)への連続送信

```

#include <stddef.h>          // NULL definition
#include "platform.h"
#include "r_sci_iic_rx_if.h"

void main(void);
void Callback_ch1(void);

void main(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_info_t      siic_info_slave1;
    sci_iic_info_t      siic_info_slave2;

    uint8_t slave_addr_eeprom[1] = {0x50}; /* Slave address for EEPROM */
    uint8_t slave_addr_m16c[1]   = {0x01}; /* Slave address for M16C */
    uint8_t write_addr_slave1[1] = {0x01}; /* 1st data field */
    uint8_t write_addr_slave2[1] = {0x02}; /* 1st data field */
    uint8_t data_area_slave1[5]  = {0x81, 0x82, 0x83, 0x84, 0x85};
    uint8_t data_area_slave2[5]  = {0x18, 0x28, 0x38, 0x48, 0x58};

    /* Sets 'Slave 1' Information (Send pattern 1) */
    siic_info_slave1.p_slv_adr   = slave_addr_eeprom;
    siic_info_slave1.p_data1st   = write_addr_slave1;
    siic_info_slave1.p_data2nd   = data_area_slave1;
    siic_info_slave1.dev_sts     = SCI_IIC_NO_INIT;
    siic_info_slave1.cnt1st      = 1;
    siic_info_slave1.cnt2nd      = 3;
    siic_info_slave1.callbackfunc = &Callback_ch1;
    siic_info_slave1.ch_no       = 1;

    /* SCI open */
    ret = R_SCI_IIC_Open(&siic_info_slave1);
    /* Start Master Send */
    ret = R_SCI_IIC_MasterSend(&siic_info_slave1);

    while((SCI_IIC_FINISH != siic_info_slave1.dev_sts) &&
          (SCI_IIC_NACK != siic_info_slave1.dev_sts));

    /* Sets 'Slave 2' Information (Send pattern 1) */
    siic_info_slave2.p_slv_adr   = slave_addr_m16c;
    siic_info_slave2.p_data1st   = write_addr_slave2;
    siic_info_slave2.p_data2nd   = data_area_slave2;
    siic_info_slave2.dev_sts     = SCI_IIC_NO_INIT;
    siic_info_slave2.cnt1st      = 1;
    siic_info_slave2.cnt2nd      = 3;
    siic_info_slave2.callbackfunc = &Callback_ch1;
    siic_info_slave2.ch_no       = 1;

    /* Start Master Send */
    ret = R_SCI_IIC_MasterSend(&siic_info_slave2);

```

アクセスするスレーブデバイスごとに情報構造体を書き換えることで複数のスレーブデバイスにアクセスできます。

```

while((SCI_IIC_FINISH != siic_info_slave2.dev_sts) &&
      (SCI_IIC_NACK != siic_info_slave2.dev_sts));
while(1);
}

void Callback_ch1(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_mcu_status_t      iic_status;
    sci_iic_info_t            iic_info_ch;

    iic_info_ch.ch_no = 1;
    ret = R_SCI_IIC_GetStatus(&iic_info_ch, &iic_status);

    if (SCI_IIC_SUCCESS != ret)
    {
        /* R_SCI_IIC_GetStatus() 関数コールエラー処理 */
    }
    else
    {
        if (1 == iic_status.BIT.NACK)
        {
            /* iic_status のステータスフラグを確認して
               NACK が検出されていた場合の処理 */
        }
    }
}

```

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	マスタ送信 パターン 4
*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	FIT_NO_PTR (注 1)
*p_data1st	1st データ格納元 アドレス	FIT_NO_PTR (注 1)	FIT_NO_PTR (注 1)	FIT_NO_PTR (注 1)
*p_data2nd	2nd データ (送信データ) バッファポインタ	2nd データ (送信データ) バッファポインタ	FIT_NO_PTR (注 1)	FIT_NO_PTR (注 1)
dev_sts	デバイス状態 フラグ	デバイス状態 フラグ	デバイス状態 フラグ	デバイス状態 フラグ
cnt1st	0000 0001h～ FFFF FFFFh (注 2)	0	0	0
cnt2nd	0000 0001h～ FFFF FFFFh (注 2)	0000 0001h～ FFFF FFFFh (注 2)	0	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。	使用する関数名を指定してください。
ch_no	00h～FFh	00h～FFh	00h～FFh	00h～FFh
rsv1,rsv2,rsv3	リザーブ (設定無効)	リザーブ (設定無効)	リザーブ (設定無効)	リザーブ (設定無効)

注 1：パターン 2、パターン 3、パターン 4 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。

3.3 R_SCI_IIC_MasterReceive()

マスタ受信を開始します。引数に合わせて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

```
sci_iic_return_t R_SCI_IIC_MasterReceive(
    sci_iic_info_t * p_sci_iic_info    /* 構造体データ */
)
```

Parameters

* p_sci_iic_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 2.8 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(SCI_IIC_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。

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

Return Values

SCI_IIC_SUCCESS	/* 問題なく処理が完了した場合 */
SCI_IIC_ERR_INVALID_CHAN	/* 存在しないチャンネルの場合 */
SCI_IIC_ERR_INVALID_ARG	/* 不正な引数の場合 */
SCI_IIC_ERR_NO_INIT	/* 初期設定ができていない場合 (未初期化状態) */
SCI_IIC_ERR_BUS_BUSY	/* バスビジーの場合 */
SCI_IIC_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_sci_iic_rx_if.h にプロトタイプ宣言されています。

Description

簡易 I²C バスのマスタ受信を開始します。引数で指定した SCI 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”(SCI_IIC_IDLE)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
- － SCI 割り込みの許可
- － I²C のリセット解除
- － I²C 出力ポートの割り当て
- － スタートコンディションの生成

Reentrant

- 異なるチャネルからリエントラントは可能です。

Example

```
#include <stddef.h>          // NULL definition
#include "platform.h"
#include "r_sci_iic_rx_if.h"

void main(void);
void Callback_ch1(void);

void main(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_info_t          siic_info;

    uint8_t slave_addr_eeprom[1] = {0x50}; /* Slave address for EEPROM */
    uint8_t access_addr1[1]       = {0x00}; /* 1st data field          */
    uint8_t store_area[5]         = {0xFF,0xFF,0xFF,0xFF,0xFF};

    /* Sets IIC Information (Ch1) */
    siic_info.p_slv_adr    = slave_addr_eeprom;
    siic_info.p_data1st    = access_addr1;
    siic_info.p_data2nd    = store_area;
    siic_info.dev_sts      = SCI_IIC_NO_INIT;
    siic_info.cnt1st       = 1;
    siic_info.cnt2nd       = 3;
    siic_info.callbackfunc = &Callback_ch1;
    siic_info.ch_no        = 1;

    /* SCI open */
    ret = R_SCI_IIC_Open(&siic_info);
    /* Start Master Receive */
    ret = R_SCI_IIC_MasterReceive(&siic_info);
    while(1);
}

void Callback_ch1(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_mcu_status_t      iic_status;
    sci_iic_info_t            iic_info_ch;

    iic_info_ch.ch_no = 1;
    ret = R_SCI_IIC_GetStatus(&iic_info_ch, &iic_status);

    if (SCI_IIC_SUCCESS != ret)
    {
        /* R_SCI_IIC_GetStatus() 関数コールエラー処理 */
    }
    else
    {
        {
            if (1 == iic_status.BIT.NACK)
            {
                /* iic_status のステータスフラグを確認して
                 NACK が検出されていた場合の処理 */
            }
        }
    }
}
```

```

    }
}
}

```

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	(設定無効)	1st データバッファポインタ
*p_data2nd	2nd (受信データ)格納先アドレス	2nd データ(受信データ) 格納先アドレス
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 (注 2)	0000 0001h ~FFFF FFFFh	0000 0001h ~FFFF FFFFh
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,rsv3	リザーブ (設定無効)	リザーブ (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。

3.4 R_SCI_IIC_Close()

簡易 I²C の通信を終了し、使用していた SCI の対象チャネルを解放します。

Format

```
sci_iic_return_t R_SCI_IIC_Close(  
    sci_iic_info_t * p_sci_iic_info    /* 構造体データ */  
)
```

Parameters

* p_sci_iic_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 2.8 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(SCI_IIC_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。

```
sci_iic_ch_dev_status_t dev_sts;    /* デバイスフラグポインタ (更新あり) */  
uint8_t ch_no;                      /* チャネル番号 */
```

Return Values

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

Properties

r_sci_iic_rx_if.h にプロトタイプ宣言されています。

Description

簡易 I²C バス通信を終了するための設定をします。引数で指定した SCI 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。

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

再度通信を開始するには、R_SCI_IIC_Open () (初期化関数)をコールする必要があります。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。

Reentrant

- 異なるチャネルからリエントラントは可能です。

Example

```
volatile sci_iic_return_t ret;  
sci_iic_info_t siic_info;  
  
siic_info.ch_no = 1;  
  
ret = R_SCI_IIC_Close(&siic_info);
```

Special Notes:

なし

3.5 R_SCI_IIC_GetStatus()

本モジュールの状態を返します。

Format

```
sci_iic_return_t R_SCI_IIC_GetStatus(
    sci_iic_info_t      * p_sci_iic_info /* 構造体データ */
    sci_iic_mcu_status_t * p_sci_iic_status /* 本モジュールのステータス */
);
```

Parameters

* *p_sci_iic_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 2.8 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(SCI_IIC_COMMUNICATION)に構造体の内容を書き換えしないでください。

```
uint8_t  ch_no; /* チャンネル番号 */
```

* *p_sci_iic_status*

I²C のステータスフラグを格納するアドレスです。引数が"FIT_NO_PTR"のときステータスは返しません。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 st_sci_iic_status_flag
    {
        uint32_t rsv :27 /* 予約ビット */
        uint32_t SCLI:1 /* SSCL ピンレベル */
        uint32_t SDAI:1 /* SSDA ピンレベル */
        uint32_t NACK :1; /* NACK 検出フラグ */
        uint32_t TRS :1; /* 送受信モードレベル */
        uint32_t BSY :1; /* バス状態フラグ */
    }BIT;
} sci_iic_mcu_status_t;
```

Return Values

```
SCI_IIC_SUCCESS /* 問題なく処理が完了した場合 */
SCI_IIC_ERR_INVALID_CHAN /* 存在しないチャンネルの場合 */
SCI_IIC_ERR_INVALID_ARG /* 不正な引数の場合 */
SCI_IIC_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_sci_iic_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 SCI 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。

Reentrant

- 異なるチャネルからリエントラントは可能です。

Example

```
volatile sci_iic_return_t ret;
sci_iic_info_t          siic_info;
sci_iic_mcu_status_t    iic_status;

siic_info.ch_no = 1

ret = R_SCI_IIC_GetStatus(&siic_info, &iic_status);
```

Special Notes:

以下にステータスフラグの配置を示します。

b31 - b16
Reserve
Reserve
Rsv
Always 0

b15 – b8
Reserve
Reserve
Rsv
Always 0

b7– b5	b4	b3	b2	b1	b0
Reserve	Pin Level		Event detection	Mode	Bus state
Reserve	SSCL pin level	SSDA pin Level	NACK detection	Send/Receive mode	Bus busy/ready
Rsv	SCLI	SDAI	NACK	TRS	BSY
Always 0	0:Low level 1:High level		0:Not detected 1:Detected	0:Receive 1:Transmit	0:Idle 1:Busy

3.6 R_SCI_IIC_Control()

各コンディション出力、NACK 出力、SSCL クロックのワンショット出力、および本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

```
sci_iic_return_t R_SCI_IIC_Control(
    r_sci_iic_info_t      * p_sci_iic_info /* 構造体データ */
    sci_iic_ctrl_ptn_t    ctrl_ptn       /* 出力パターン */
);
```

Parameters

* *p_sci_iic_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 2.8 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(SCI_IIC_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。

```
sci_iic_ch_dev_status_t dev_sts; /* デバイスフラグポインタ (更新あり) */
uint8_t ch_no; /* チャンネル番号 */
```

ctrl_ptn

生成波形パターンの設定をします。複数選択する場合は、"|" (OR) を用いてください。

- 複数選択が可能なパターン

・ 'SCI_IIC_GEN_START_CON', 'SCI_IIC_GEN_RESTART_CON', 'SCI_IIC_GEN_STOP_CON' の 3 つは同時指定可能です。

3 つを組み合わせて選択する場合は、上記順番で実施されます。

・ 'SCI_IIC_GEN_SSDA_HI_Z', 'SCI_IIC_GEN_SSCL_ONESHOT' の 2 つは同時指定可能です。

```
typedef uint8_t sci_iic_ctrl_ptn_t;
```

```
#define SCI_IIC_GEN_START_CON      (sci_iic_ctrl_ptn_t) (0x01)
/* スタートコンディションの生成 */
#define SCI_IIC_GEN_STOP_CON      (sci_iic_ctrl_ptn_t) (0x02)
/* ストップコンディションの生成 */
#define SCI_IIC_GEN_RESTART_CON   (sci_iic_ctrl_ptn_t) (0x04)
/* リスタートコンディションの生成 */
#define SCI_IIC_GEN_SSDA_HI_Z     (sci_iic_ctrl_ptn_t) (0x08)
/* SSDA 端子から Hi-z 出力 */
#define SCI_IIC_GEN_SSCL_ONESHOT (sci_iic_ctrl_ptn_t) (0x10)
/* SSCL クロックのワンショット出力 */
#define SCI_IIC_GEN_RESET         (sci_iic_ctrl_ptn_t) (0x20)
/* 簡易 I2C バスのモジュールリセット */
```

Return Values

```
SCI_IIC_SUCCESS /* 問題なく処理が完了した場合 */
SCI_IIC_ERR_INVALID_CHAN /* 存在しないチャンネルの場合 */
SCI_IIC_ERR_INVALID_ARG /* 不正な引数の場合 */
SCI_IIC_ERR_BUS_BUSY /* バスビジーの場合 */
SCI_IIC_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_sci_iic_rx_if.h にプロトタイプ宣言されています。

Description

簡易 I²C バス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、SSDA 端子から Hi-z 出力、SSCL クロックのワンショット出力、および簡易 I²C バスのモジュールリセットを行います。

Reentrant

- 異なるチャネルからリエントラントは可能です。

Example

```
volatile sci_iic_return_t ret;
sci_iic_info_t          siic_info;

siic_info.ch_no = 1;

/* Output an extra SSCL clock cycle after changes the SSDA pin in a high-
impedance state */
ret = R_SCI_IIC_Control(&siic_info, SCI_IIC_GEN_SSDA_HI_Z |
SCI_IIC_SSCL_ONESHOT);
```

Special Notes:

なし

3.7 R_SCI_IIC_GetVersion()

本モジュールのバージョンを返します。

Format

uint32_t R_SCI_IIC_GetVersion(void)

Parameters

なし

Return Values

バージョン番号

Properties

r_sci_iic_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 SCI(簡易 I²C モード) 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。

Reentrant

- 異なるチャネルからリエントラントは可能です。

Example

```
uint32_t version;
```

```
version = R_SCI_IIC_GetVersion();
```

Special Notes:

この関数は“#pragma inline”を使用してインライン化されています。

4. 付録

4.1 通信の実現方法

本 API では、スタートコンディション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。

4.1.1 API 動作の状態

表 4.1 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 4.1 プロトコル制御のための状態一覧 (enum sci_iic_api_status_t)

No	定数名	内容
STS0	SCI_IIC_STS_NO_INIT	未初期化状態
STS1	SCI_IIC_STS_IDLE	アイドル状態
STS2	SCI_IIC_STS_ST_COND_WAIT	スタートコンディション生成完了待ち状態
STS3	SCI_IIC_STS_SEND_SLVADR_W_WAIT	スレーブアドレス[Write]送信完了待ち状態
STS4	SCI_IIC_STS_SEND_SLVADR_R_WAIT	スレーブアドレス[Read]送信完了待ち状態
STS5	SCI_IIC_STS_SEND_DATA_WAIT	データ送信完了待ち状態
STS6	SCI_IIC_STS_RECEIVE_DATA_WAIT	データ受信完了待ち状態
STS7	SCI_IIC_STS_SP_COND_WAIT	ストップコンディション生成完了待ち状態

4.1.2 API 動作中のイベント

表 4.2 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インタフェースがコールされた際も、イベントとして定義します。

表 4.2 プロトコル制御のためのイベント一覧 (enum sci_iic_api_event_t)

No	イベント	イベントの定義
EV0	SCI_IIC_EV_INIT	sci_iic_init_driver() コール
EV1	SCI_IIC_EV_GEN_START_COND	sci_iic_generate_start_cond() コール
EV2	SCI_IIC_EV_INT_START	STI 割り込み発生(割り込みフラグ : START)
EV3	SCI_IIC_EV_INT_ADD	TXI 割り込み発生
EV4	SCI_IIC_EV_INT_SEND	TXI 割り込み発生
EV5	SCI_IIC_EV_INT_STOP	STI 割り込み発生(割り込みフラグ : STOP)
EV6	SCI_IIC_EV_INT_NACK	STI 割り込み発生(割り込みフラグ : NACK)

4.1.3 プロトコル状態遷移

本モジュールでは、提供インタフェースのコール、または、SCI(簡易 I²C モード)の割り込み発生をトリガに状態が遷移します。図 4.1～図 4.4 に各プロトコルの状態遷移を示します。

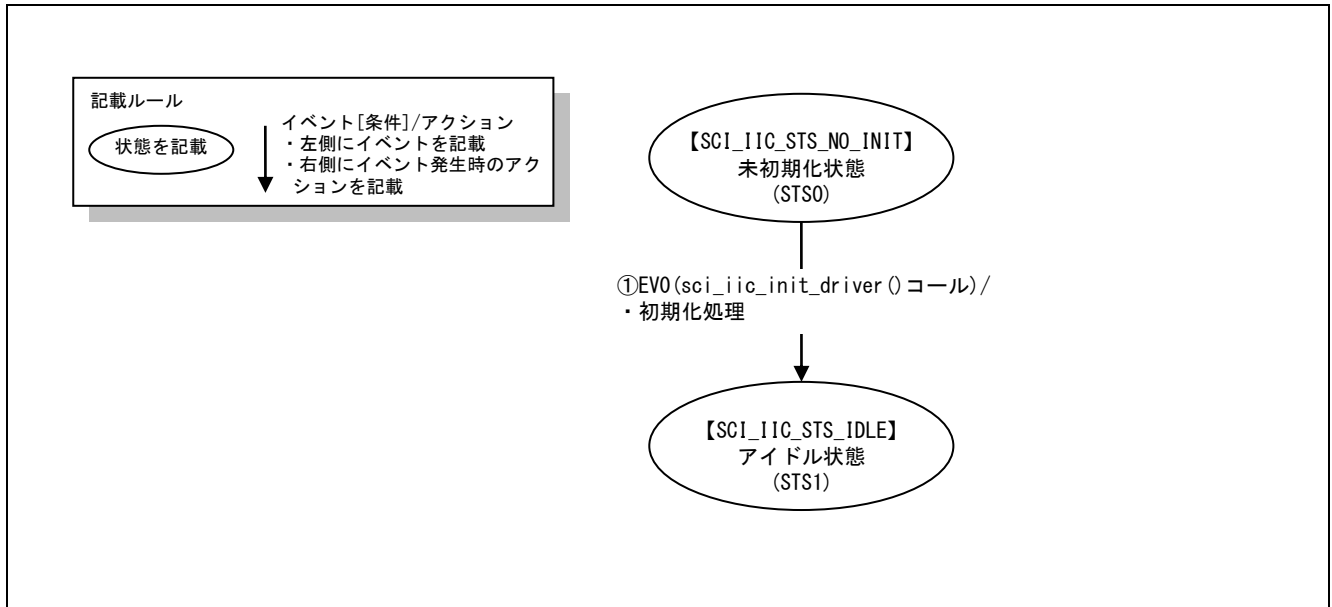


図 4.1 初期化状態遷移図

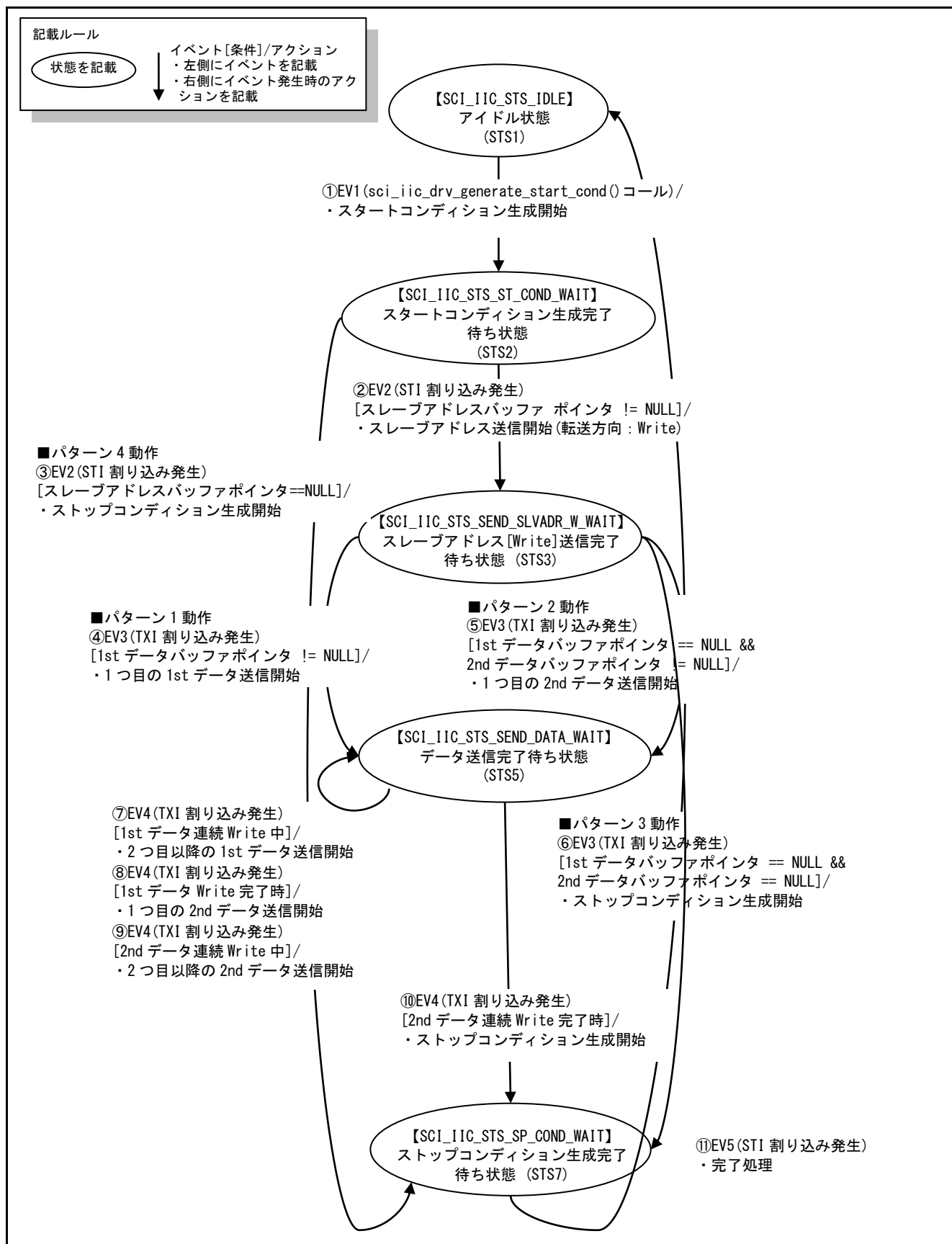


図 4.2 マスタ送信 状態遷移図

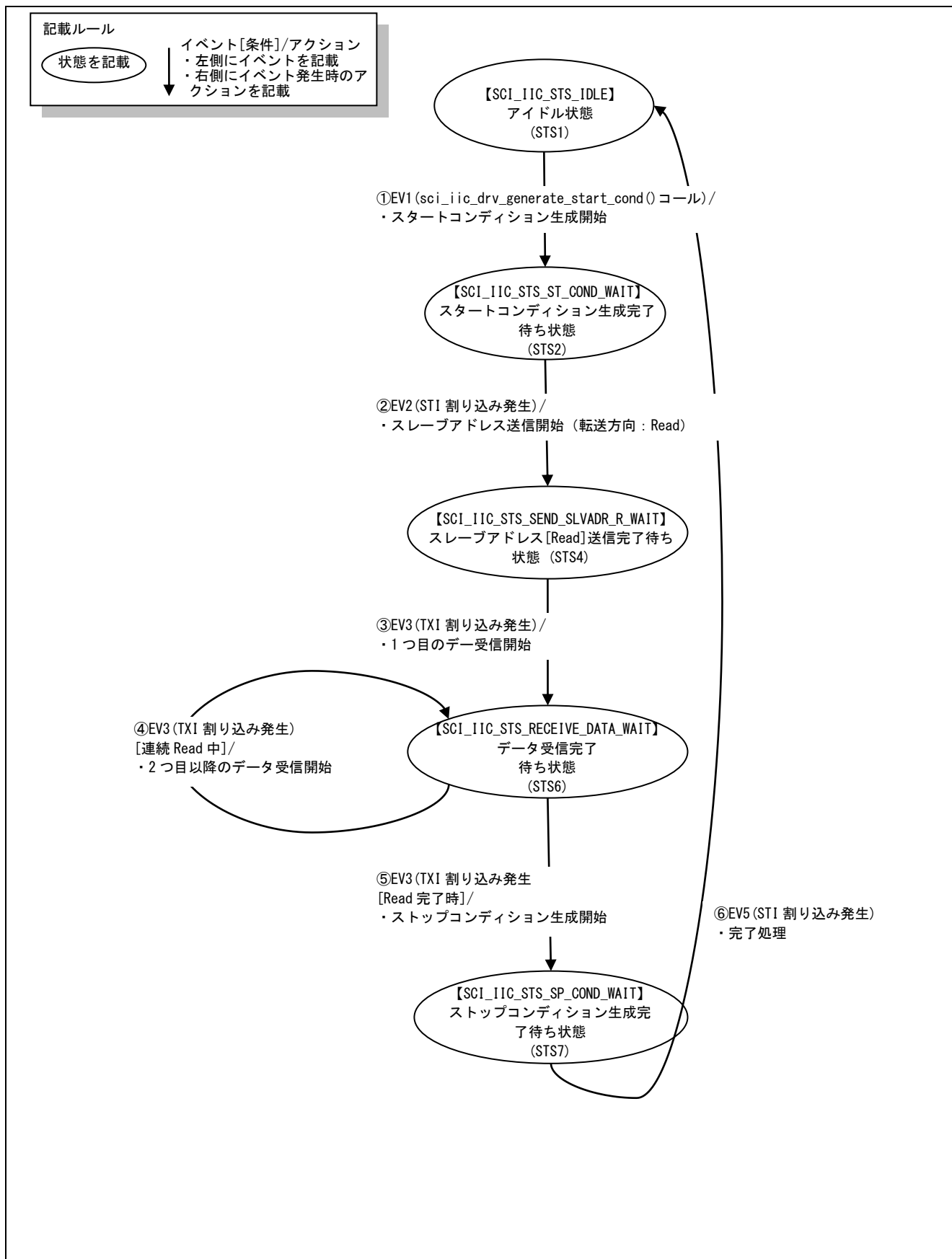


図 4.3 マスタ受信 状態遷移図

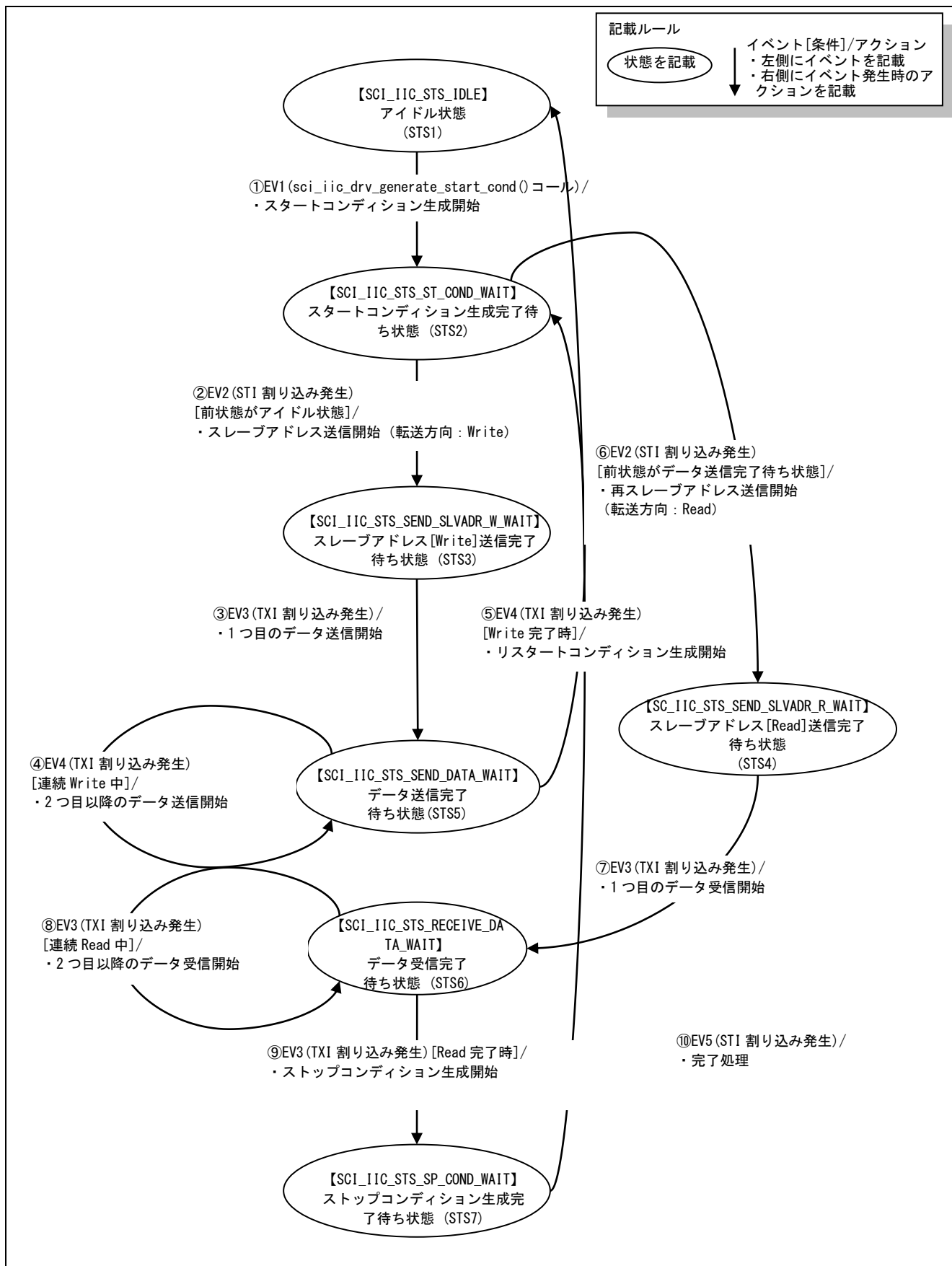


図 4.4 マスタ送受信 状態遷移図

4.1.4 プロトコル状態遷移表

表 4.1 の各状態で、表 4.2 のイベントが発生した際に動作する処理を、表 4.3 の状態遷移表に定義します。Func については表 4.4 を参照してください。

表 4.3 プロトコル状態遷移表(gc_sci_iic_mtx_tbl[][])

状態	イベント	EV0	EV1	EV2	EV3	EV4	EV5	EV6
STS0	未初期化状態 【SCI_IIC_STS_NO_INIT】	Func0	ERR	ERR	ERR	ERR	ERR	ERR
STS1	アイドル状態 【SCI_IIC_STS_IDLE】	ERR	Func1	ERR	ERR	ERR	ERR	ERR
STS2	スタートコンディション生成完了待ち状態 【SCI_IIC_STS_ST_COND_WAIT】	ERR	ERR	Func2	ERR	ERR	ERR	Func7
STS3	スレーブアドレス[Write]送信完了待ち状態 【SCI_IIC_STS_SEND_SLVADR_W_WAIT】	ERR	ERR	ERR	Func3	ERR	ERR	Func7
STS4	スレーブアドレス[Read]送信完了待ち状態 【SCI_IIC_STS_SEND_SLVADR_R_WAIT】	ERR	ERR	ERR	Func3	ERR	ERR	Func7
STS5	データ送信完了待ち状態 【SCI_IIC_STS_SEND_DATA_WAIT】	ERR	ERR	ERR	ERR	Func4	ERR	Func7
STS6	データ受信完了待ち状態 【SCI_IIC_STS_RECEIVE_DATA_WAIT】	ERR	ERR	ERR	Func5	ERR	ERR	Func7
STS7	ストップコンディション生成完了待ち状態 【SCI_IIC_STS_SP_COND_WAIT】	ERR	ERR	ERR	ERR	ERR	Func6	Func7

備考：ERR は SCI_IIC_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。

4.1.5 プロトコル状態遷移登録関数

表 4.4 に状態遷移表に登録されている関数を定義します。

表 4.4 プロトコル状態遷移登録関数一覧

処理	関数名	概要
Func0	sci_iic_init_driver()	初期設定処理
Func1	sci_iic_generate_start_cond()	スタートコンディション生成処理
Func2	sci_iic_after_gen_start_cond()	スタートコンディション生成後処理
Func3	sci_iic_after_send_slvadr()	スレーブアドレス送信完了後処理
Func4	sci_iic_write_data_sending()	データ送信処理
Func5	sci_iic_read_data_receiving()	データ受信処理
Func6	sci_iic_release()	通信完了処理
Func7	sci_iic_nack()	NACK エラー処理

4.1.6 状態遷移時の各フラグの状態

1) 各チャンネル状態管理

チャンネル状態フラグ `g_sci_iic_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、各チャンネルに対して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“`SCI_IIC_IDLE/SCI_IIC_FINISH/SCI_IIC_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、“`SCI_IIC_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、通信中に同一チャンネル上の他デバイスの通信を開始しません。本フラグをチャンネルごとに管理することで、複数チャンネルの同時通信を実現します。

2) 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 4.5 に状態遷移時の各フラグの状態を示します。

表 4.5 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_sci_iic_ChStatus[]	I ² C 通信情報構造体 dev_sts	内部通信情報構造体 api_Mode	内部通信情報構造体 api_N_status
未初期化状態	SCI_IIC_NO_INIT	SCI_IIC_NO_INIT	SCI_IIC_MODE_NONE	SCI_IIC_STS_NO_INIT
アイドル状態	SCI_IIC_IDLE SCI_IIC_FINISH SCI_IIC_NACK	SCI_IIC_IDLE SCI_IIC_FINISH SCI_IIC_NACK	SCI_IIC_MODE_NONE	SCI_IIC_STS_IDLE
通信中 (マスタ送信)	SCI_IIC _COMMUNICATION	SCI_IIC _COMMUNICATION	SCI_IIC_MODE_SEND	SCI_IIC_STS_ST_COND_WAIT
				SCI_IIC _STS_SEND_SLVADR_W_WAIT
				SCI_IIC_STS_SEND_DATA_WAIT
				SCI_IIC_STS_SP_COND_WAIT
通信中 (マスタ受信)	SCI_IIC _COMMUNICATION	SCI_IIC _COMMUNICATION	SCI_IIC_MODE_RECEIVE	SCI_IIC_STS_ST_COND_WAIT
				SCI_IIC _STS_SEND_SLVADR_R_WAIT
				SCI_IIC_STS_RECEIVE_DATA_WAIT
				SCI_IIC_STS_SP_COND_WAIT
通信中 (マスタ送受信)	SCI_IIC _COMMUNICATION	SCI_IIC _COMMUNICATION	SCI_IIC _MODE_SEND_RECEIVE	SCI_IIC_STS_ST_COND_WAIT
				SCI_IIC _STS_SEND_SLVADR_W_WAIT
				SCI_IIC _STS_SEND_SLVADR_R_WAIT
				SCI_IIC_STS_SEND_DATA_WAIT
				SCI_IIC_STS_RECEIVE_DATA_WAIT
エラー	SCI_IIC_ERROR	SCI_IIC_ERROR	-	SCI_IIC_STS_SP_COND_WAIT
				-

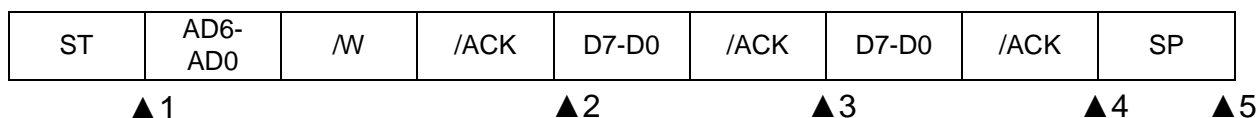
4.2 割り込み発生タイミング

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ン

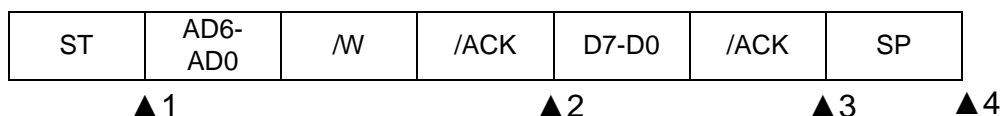
4.2.1 マスタ送信

(1) パターン 1



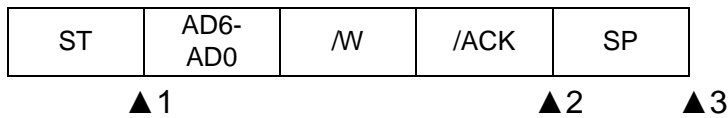
- ▲1 : STI(START)割り込み・・・スタートコンディション検出
- ▲2 : TXI 割り込み・・・アドレス送信完了(転送方向ビット : Write) ※1
- ▲3 : TXI 割り込み・・・データ送信完了(1st データ) ※1
- ▲4 : TXI 割り込み・・・データ送信完了(2nd データ) ※1
- ▲5 : STI(STOP)割り込み・・・ストップコンディション検出

(2) パターン 2



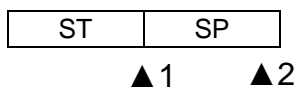
- ▲1 : STI(START)割り込み・・・スタートコンディション検出
- ▲2 : TXI 割り込み・・・アドレス送信完了(転送方向ビット : Write) ※1
- ▲3 : TXI 割り込み・・・データ送信完了(2nd データ) ※1
- ▲4 : STI(STOP)割り込み・・・ストップコンディション検出

(3) パターン 3



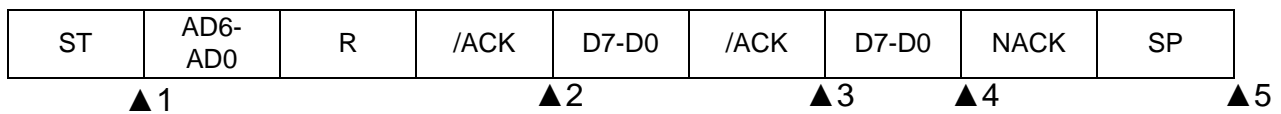
- ▲1: STI(START)割り込み・・・スタートコンディション検出
▲2: TXI 割り込み・・・アドレス送信完了(転送方向ビット: Write) ※1
▲3: STI(STOP)割り込み・・・ストップコンディション検出

(4) パターン 4



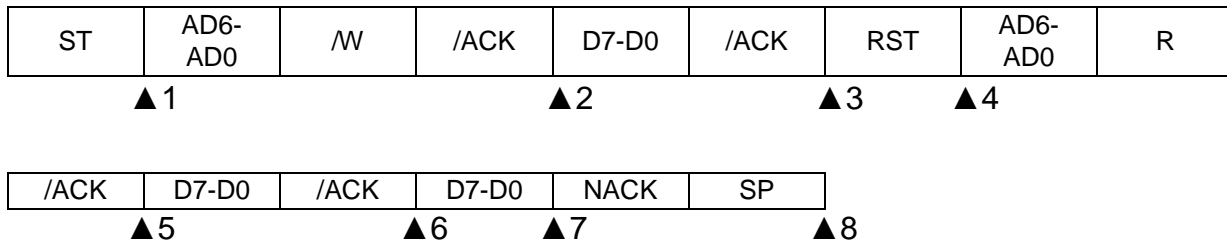
- ▲1: STI(START)割り込み・・・スタートコンディション検出
▲2: STI(STOP)割り込み・・・ストップコンディション検出

4.2.2 マスタ受信



- ▲1: STI(START)割り込み・・・スタートコンディション検出
▲2: TXI 割り込み・・・アドレス送信完了(転送方向ビット: Read) ※1
▲3: TXI 割り込み・・・最終データ1受信完了(2nd データ) ※1
▲4: TXI 割り込み・・・最終データ受信完了(2nd データ) ※2
▲5: STI(STOP)割り込み・・・ストップコンディション検出

4.2.3 マスタ送受信



▲1 : STI(START)割り込み・・・スタートコンディション検出

▲2 : TXI 割り込み・・・アドレス送信完了(転送方向ビット : Write) ※1

▲3 : TXI 割り込み・・・データ送信完了(1st データ) ※1

▲4 : STI(START)割り込み・・・リスタートコンディション検出

▲5 : TXI 割り込み・・・アドレス送信完了(転送方向ビット : Read) ※1

▲6 : TXI 割り込み・・・最終データ-1 受信完了(2nd データ) ※1

▲7 : TXI 割り込み・・・最終データ受信完了(2nd データ) ※2

▲8 : STI(STOP)割り込み・・・ストップコンディション検出

※1 : 9クロック目の立ち上がりエッジで要求の発生

※2 : 8クロック目の立ち上がりで要求の発生

5. サンプルコード

5.1 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

SCI の 1 つのチャンネルを簡易 I²C モードで使用し、1 つのスレーブデバイスに対して、書き込む場合のサンプルコードを示します。

次の(1)~(5)の順に動作します。

- (1) SCI の ch1 を簡易 I²C FIT モジュールで使用可能にするため、R_SCI_IIC_Open 関数を実行する。
- (2) デバイス A に 3 バイトのデータを書き込むため、R_SCI_IIC_MasterSend 関数を実行する。
- (3) 送信データの更新
- (4) デバイス A に 3 バイトのデータを書き込むため、R_SCI_IIC_MasterSend 関数を実行する。
- (5) SCI の ch1 を簡易 I²C FIT モジュールから解放するため、R_SCI_IIC_Close 関数を実行する。

```
#include <stddef.h> // NULL definition
#include "platform.h"
#include "r_sci_iic_rx_if.h"

/* NACK 検出時のリトライ回数の定義 */
#define RETRY_TMO 10

/* リトライ時の次の通信開始まで待つためのソフトウェアループ回数の定義 */
#define RETRY_WAIT_TIME 1000

/* 送信サイズ */
#define SEND_SIZE 3

/* サンプルコードのモード管理用定義 */
typedef enum
{
    IDLE = 0U,                /* アイドル中 */
    BUSY,                    /* I2C 通信中 */
    INITIALIZE,              /* 簡易 I2C FIT モジュールの初期設定 */
    DEVICE_A_WRITE,         /* デバイス A への書き込み */
    FINISH,                  /* 通信終了 */
    RETRY_WAIT_DEV_A_WR,    /* デバイス A への書き込みのリトライ待ち */
    ERROR                    /* エラー発生 */
} sample_mode_t;

/* サンプルコードのモード管理用変数 */
volatile uint8_t sample_mode;

/* リトライ回数管理用変数 */
uint32_t retry_cnt;

/* 送信回数管理用変数 */
uint8_t send_num = 0;

void main(void);
void Callback_deviceA(void);

void main(void)
{
    sci_iic_return_t ret;                /* API 関数の戻り値確認用 */
    volatile uint32_t retry_wait_cnt = 0; /* リトライ間隔調整用カウンタ */

    sci_iic_info_t iic_info_deviceA;    /* デバイス A 用情報構造体 */
    uint8_t slave_addr_deviceA[1] = {0x50}; /* デバイス A 用スレーブアドレス */
    uint8_t access_addr_deviceA[1] = {0x00}; /* デバイス A 用アクセス先アドレス */
    uint8_t send_data[6] = {0x81, 0x82, 0x83, 0x84, 0x85, 0x86}; /* 送信データ */
}
```

プログラムの説明で、
次の略称を使用しています。
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 5.1 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(1)

```

sample_mode = INITIALIZE;                                /* 次に"初期設定"を行う */

while(1)
{
    switch(sample_mode)
    {
        /* アイドル中 */
        case IDLE:
            /* Nothing to do */
            break;

        /* I2C 通信中 */
        case BUSY:
            /* Nothing to do */
            break;

        /* 簡易 I2C FIT モジュールの初期設定 */
        case INITIALIZE:
            /* デバイス A へ初回の通信か */
            if (0 == send_num)
            {
                /* ch1 は通信中か */
                if (SCI_IIC_COMMUNICATION == g_sci_iic_ChStatus[1])
                {
                    sample_mode = ERROR;                    /* 次に"エラー時の処理"を行う */
                }
                else
                {
                    /* デバイス A の情報構造体の設定 (送信パターン 1) */
                    iic_info_deviceA.p_slv_addr = slave_addr_deviceA;
                    iic_info_deviceA.p_data1st = access_addr_deviceA;
                    iic_info_deviceA.p_data2nd = send_data;
                    iic_info_deviceA.dev_sts = SCI_IIC_NO_INIT;
                    iic_info_deviceA.cnt1st = sizeof(access_addr_deviceA);
                    iic_info_deviceA.cnt2nd = SEND_SIZE;
                    iic_info_deviceA.callbackfunc = &Callback_deviceA;
                    iic_info_deviceA.ch_no = 1;
                }

                retry_cnt = 0;

                /* SCI open */
                ret = R_SCI_IIC_Open(&iic_info_deviceA);

                if (SCI_IIC_SUCCESS == ret)
                {
                    sample_mode = DEVICE_A_WRITE; /* 次に"デバイス A への書き込み"を行う */
                }
                else
                {
                    /* R_SCI_IIC_Open() 関数コール時のエラー処理 */
                    sample_mode = ERROR;            /* 次に"エラー時の処理"を行う */
                }
            }
            /* デバイス A へ連続した 2 回目以降の通信か */
            else if (1 <= send_num)
            {
                /* ch1 は通信中か */
                if (SCI_IIC_COMMUNICATION == g_sci_iic_ChStatus[1])
                {
                    sample_mode = ERROR;            /* 次に"エラー時の処理"を行う */
                }
                else
                {
                    /* デバイス A の情報構造体の設定 (マスタ送信パターン 1) */
                    access_addr_deviceA[0] = (access_addr_deviceA[0] + SEND_SIZE);
                    iic_info_deviceA.p_data1st = access_addr_deviceA;
                    iic_info_deviceA.p_data2nd = (send_data + (SEND_SIZE * send_num));
                    iic_info_deviceA.cnt1st = sizeof(access_addr_deviceA);
                    iic_info_deviceA.cnt2nd = SEND_SIZE;
                }
            }
        }
    }
}

```

アイドル中や I²C 通信中は、空処理でループする

チャンネルの状態をグローバル変数の g_sci_iic_ChStatus[] で確認できる

2 回目の送信を行うため、送信カウンタやポインタを初期化する

図 5.2 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(2)

```

        sample_mode = DEVICE_A_WRITE; /* 次に"デバイス A への書き込み"を行う */
    }
}
break;

/* デバイス A への書き込み */
case DEVICE_A_WRITE:
    retry_cnt = retry_cnt + 1;

    /* マスタ送信の開始 */
    ret = R_SCI_IIC_MasterSend(&iic_info_deviceA);

    if (SCI_IIC_SUCCESS == ret)
    {
        sample_mode = BUSY; /* 次に"I2C 通信中"となる */
    }
    else if (SCI_IIC_ERR_BUS_BUSY == ret)
    {
        sample_mode = RETRY_WAIT_DEV_A_WR; /* 次に"リトライ待ち"を行う */
    }
    else
    {
        /* R_SCI_IIC_MasterSend() 関数コール時のエラー処理 */
        sample_mode = ERROR; /* 次に"エラー時の処理"を行う */
    }
break;

/* デバイス A への書き込みのリトライ待ち */
case RETRY_WAIT_DEV_A_WR:
    retry_wait_cnt = retry_wait_cnt + 1;

    if (RETRY_TMO < retry_cnt)
    {
        retry_wait_cnt = 0;
        sample_mode = ERROR; /* 次に"エラー時の処理"を行う */
    }

    if (RETRY_WAIT_TIME < retry_wait_cnt)
    {
        retry_wait_cnt = 0;

        switch (sample_mode)
        {
            case RETRY_WAIT_DEV_A_WR:
                sample_mode = DEVICE_A_WRITE; /* 次に"デバイス A への書き込み"を行う */
                break;

            default:
                /* Nothing to do */
                break;
        }
    }
}
break;

```

通信相手デバイスが EEPROM の場合、書き込みコマンドを送信し書くと、EEPROM の書き込み処理が完了するまでは、NACK 応答となる
サンプルコードでは、ACK 応答になるまで通信開始のリトライ処理をしています

図 5.3 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)

```

/* 通信終了 */
case FINISH:
    /* SCI close */
    ret = R_SCI_IIC_Close(&iic_info_deviceA);

    if (SCI_IIC_SUCCESS == ret)
    {
        sample_mode = IDLE; /* 次に"アイドル中"となる */
    }
    else
    {
        /* R_SCI_IIC_Close() 関数コール時のエラー処理 */
        sample_mode = ERROR; /* 次に"エラー時の処理"を行う */
    }
    break;

/* エラー発生 */
case ERROR:
    /* Nothing to do */
    break;

default:
    /* Nothing to do */
    break;
}
}

void Callback_deviceA(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_mcu_status_t iic_status;
    sci_iic_info_t iic_info_ch;
    iic_info_ch.ch_no = 1;

    /* 簡易 I2C ステータスの取得 */
    ret = R_SCI_IIC_GetStatus(&iic_info_ch, &iic_status);

    if (SCI_IIC_SUCCESS != ret)
    {
        /* R_SCI_IIC_GetStatus() 関数コール時のエラー処理 */
        sample_mode = ERROR; /* 次に"エラー時の処理"を行う */
    }
    else
    {
        if (1 == iic_status.BIT.NACK)
        {
            /* iic_status のステータスフラグを確認して
             NACK が検出されていた場合の処理 */
            sample_mode = RETRY_WAIT_DEV_A_WR;
        }
        else
        {
            retry_cnt = 0;
            send_num++;
            if (1 >= send_num)
            {
                sample_mode = INITIALIZE; /* 次に"初期設定"を行う */
            }
            else
            {
                sample_mode = FINISH; /* 次に"通信終了"を行う */
            }
        }
    }
}
}

```

通信が終了した場合に R_SCI_IIC_Close 関数を呼び出すことで使用している SCI チャンネルを解放することができる
R_SCI_IIC_Close 関数は主に次の場合に呼び出してください

- ・低消費電力モードに移行する場合
- ・通信異常が発生した場合
- ・使用している SCI のチャンネルを解放したい場合

図 5.4 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)

5.2 1つのチャンネルで2つのスレーブデバイスにアクセスする場合の例

SCI の 1 つのチャンネルを簡易 I²C モードで使用し、2 つのスレーブデバイスに対して書き込みおよび読み出しを行う場合のサンプルコードを示します。

サンプルコードでは、アクセスするデバイスごとに、I²C 通信情報構造体を用意しています。

次の(1)~(4)の順に動作します。

- (1) SCI の ch1 を簡易 I²C FIT モジュールで使用可能にするため、R_SCI_IIC_Open 関数を実行する。
- (2) デバイス A に 3 バイトのデータを書き込むため、R_SCI_IIC_MasterSend 関数を実行する。
- (3) デバイス B から 3 バイトのデータを読み出すため、R_SCI_IIC_MasterReceive 関数を実行する。
- (4) SCI の ch1 を簡易 I²C FIT モジュールから解放するため、R_SCI_IIC_Close 関数を実行する。

```
#include <stddef.h> // NULL definition
#include "platform.h"
#include "r_sci_iic_rx_if.h"

/* NACK 検出時のリトライ回数の定義 */
#define RETRY_TMO 10

/* リトライ時の次の通信開始まで待つためのソフトウェアループ回数の定義 */
#define RETRY_WAIT_TIME 1000

/* 送信サイズ */
#define SEND_SIZE 3

/* 受信サイズ */
#define RECEIVE_SIZE 3

/* サンプルコードのモード管理用定義 */
typedef enum
{
    IDLE = 0U, /* アイドル中 */
    BUSY, /* I2C 通信中 */
    INITIALIZE, /* 簡易 I2C FIT モジュールの初期設定 */
    DEVICE_A_WRITE, /* デバイス A への書き込み */
    DEVICE_B_READ, /* デバイス B からの読み出し */
    FINISH, /* 通信終了 */
    RETRY_WAIT_DEV_A_WR, /* デバイス A への書き込みのリトライ待ち */
    RETRY_WAIT_DEV_B_RD, /* デバイス B からの読み出しのリトライ待ち */
    ERROR /* エラー発生 */
} sample_mode_t;

/* サンプルコードのモード管理用変数 */
volatile uint8_t sample_mode;

/* リトライ回数管理用変数 */
volatile uint32_t retry_cnt;

void main(void);
void Callback_deviceA(void);
void Callback_deviceB(void);

void main(void)
{
    volatile sci_iic_return_t ret; /* API 関数の戻り値確認用 */
    volatile uint32_t retry_wait_cnt = 0; /* リトライ間隔調整用カウンタ */

    sci_iic_info_t iic_info_deviceA; /* デバイス A 用情報構造体 */
    sci_iic_info_t iic_info_deviceB; /* デバイス B 用情報構造体 */
```

プログラムの説明で、
次の略称を使用しています。
ST: スタートコンディション
SP: ストップコンディション

通信するデバイスの数だけ、
情報構造体を宣言する

図 5.5 1つのチャンネルで2つのスレーブデバイスにアクセスする例(1)

```

uint8_t slave_addr_deviceA[1] = {0x51};      /* デバイス A 用スレーブアドレス */
uint8_t slave_addr_deviceB[1] = {0x52};      /* デバイス B 用スレーブアドレス */
uint8_t access_addr_deviceA[1] = {0x00};      /* デバイス A 用アクセス先アドレス */
uint8_t access_addr_deviceB[2] = {0x00,0x00}; /* デバイス B 用アクセス先アドレス */
uint8_t send_data[5]          = {0x81,0x82,0x83,0x84,0x85}; /* 送信データ */
uint8_t store_area[5]         = {0xFF,0xFF,0xFF,0xFF,0xFF}; /* 受信データ格納用 */

sample_mode = INITIALIZE;                      /* 次に"初期設定"を行う */

while(1)
{
    switch(sample_mode)
    {
        /* アイドル中 */
        case IDLE:
            /* Nothing to do */
            break;

        /* I2C 通信中 */
        case BUSY:
            /* Nothing to do */
            break;

        /* 簡易 I2C FIT モジュールの初期設定 */
        case INITIALIZE:
            /* チャンネル 1 は通信中か */
            if (SCI_IIC_COMMUNICATION == g_sci_iic_ChStatus[1])
            {
                sample_mode = ERROR;          /* 次に"エラー時の処理"を行う */
            }
            else
            {
                /* デバイス A の情報構造体の設定 (マスタ送信パターン 1) */
                iic_info_deviceA.p_slv_adr = slave_addr_deviceA;
                iic_info_deviceA.p_data1st = access_addr_deviceA;
                iic_info_deviceA.p_data2nd = send_data;
                iic_info_deviceA.dev_sts = SCI_IIC_NO_INIT;
                iic_info_deviceA.cnt1st = sizeof(access_addr_deviceA);
                iic_info_deviceA.cnt2nd = SEND_SIZE;
                iic_info_deviceA.callbackfunc = &Callback_deviceA;
                iic_info_deviceA.ch_no = 1;

                /* デバイス B の情報構造体の設定 (マスタ送受信) */
                iic_info_deviceB.p_slv_adr = slave_addr_deviceB;
                iic_info_deviceB.p_data1st = access_addr_deviceB;
                iic_info_deviceB.p_data2nd = store_area;
                iic_info_deviceB.dev_sts = SCI_IIC_NO_INIT;
                iic_info_deviceB.cnt1st = sizeof(access_addr_deviceB);
                iic_info_deviceB.cnt2nd = RECEIVE_SIZE;
                iic_info_deviceB.callbackfunc = &Callback_deviceB;
                iic_info_deviceB.ch_no = 1;
            }

            retry_cnt = 0;

            /* SCI open */
            ret = R_SCI_IIC_Open(&iic_info_deviceA);

            if (SCI_IIC_SUCCESS == ret)
            {
                sample_mode = DEVICE_A_WRITE; /* 次に"デバイス A への書き込み"を行う */
            }
            else
            {
                /* R_SCI_IIC_Open() 関数コール時のエラー処理 */
                sample_mode = ERROR;          /* 次に"エラー時の処理"を行う */
            }

            break;
    }
}

```

アイドル中や I²C 通信中は、
空処理でループする

チャンネルの状態をグローバル変数の
g_sci_iic_ChStatus[] で確認できる

SCI リソースの確保は、チャンネルごとに行うため、
R_SCI_IIC_Open 関数は 1 回だけ実行する
/* リトライ回数の初期化 */

図 5.6 1つのチャンネルで2つのスレーブデバイスにアクセスする例(2)

```

/* デバイス A への書き込み */
case DEVICE_A_WRITE:
    retry_cnt = retry_cnt + 1;

    /* マスタ送信の開始 */
    ret = R_SCI_IIC_MasterSend(&iic_info_deviceA);

    if (SCI_IIC_SUCCESS == ret)
    {
        sample_mode = BUSY; /* 次に"I2C 通信中"となる */
    }
    else if (SCI_IIC_ERR_BUS_BUSY == ret)
    {
        sample_mode = RETRY_WAIT_DEV_A_WR; /* 次に"リトライ待ち"を行う */
    }
    else
    {
        /* R_SCI_IIC_MasterSend() 関数コール時のエラー処理 */
        sample_mode = ERROR; /* 次に"エラー時の処理"を行う */
    }
break;

/* デバイス B からの読み出し */
case DEVICE_B_READ:
    retry_cnt = retry_cnt + 1;

    /* マスタ受信の開始 */
    ret = R_SCI_IIC_MasterReceive(&iic_info_deviceB);

    if (SCI_IIC_SUCCESS == ret)
    {
        sample_mode = BUSY; /* 次に"I2C 通信中"となる */
    }
    else if (SCI_IIC_ERR_BUS_BUSY == ret)
    {
        sample_mode = RETRY_WAIT_DEV_B_RD; /* 次に"リトライ待ち"を行う */
    }
    else
    {
        /* R_SCI_IIC_MasterReceive() 関数コール時のエラー処理 */
        sample_mode = ERROR; /* 次に"エラー時の処理"を行う */
    }
break;

/* デバイス A への書き込みのリトライ待ち */
/* デバイス B からの読み出しのリトライ待ち */
case RETRY_WAIT_DEV_A_WR:
case RETRY_WAIT_DEV_B_RD:
    retry_wait_cnt = retry_wait_cnt + 1;

    if (RETRY_TMO < retry_cnt)
    {
        retry_wait_cnt = 0;
        sample_mode = ERROR; /* 次に"エラー時の処理"を行う */
    }

    if (RETRY_WAIT_TIME < retry_wait_cnt)
    {
        retry_wait_cnt = 0;

        switch (sample_mode)
        {
            case RETRY_WAIT_DEV_A_WR:
                sample_mode = DEVICE_A_WRITE; /* 次に"デバイス A への書き込み"を行う */
                break;

            case RETRY_WAIT_DEV_B_RD:
                sample_mode = DEVICE_B_READ; /* 次に"デバイス B からの読み出し"を行う */
                break;
        }
    }

```

R_SCI_IIC_MasterSend 関数実行により、ST 生成から SP 生成まで FIT モジュールで処理される SP 出力後に設定したコールバック関数 (Callback_deviceA()) が呼び出される

R_SCI_IIC_MasterReceive 関数実行により、ST 生成から SP 生成まで FIT モジュールで処理される SP 出力後に設定したコールバック関数 (Callback_deviceB()) が呼び出される

通信相手デバイスが EEPROM の場合、書き込みコマンドを送信し書くと、EEPROM の書き込み処理が完了するまでは、NACK 応答となる
サンプルコードでは、ACK 応答になるまで通信開始のリトライ処理をしています

図 5.7 1つのチャンネルで2つのスレーブデバイスにアクセスする例(3)

```

        default:
            /* Nothing to do */
            break;
    }
}
break;

/* 通信終了 */
case FINISH:
    /* SCI close */
    ret = R_SCI_IIC_Close(&iic_info_deviceA);

    if (SCI_IIC_SUCCESS == ret)
    {
        sample_mode = IDLE;
    }
    else
    {
        /* R_SCI_IIC_Close() 関数コール時のエラー処理 */
        sample_mode = ERROR;
    }
    break;

/* エラー発生 */
case ERROR:
    /* Nothing to do */
    break;

default:
    /* Nothing to do */
    break;
}
}

void Callback_deviceA(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_mcu_status_t iic_status;
    sci_iic_info_t iic_info_ch;
    iic_info_ch.ch_no = 1;

    /* 簡易 I2C ステータスの取得 */
    ret = R_SCI_IIC_GetStatus(&iic_info_ch, &iic_status);

    if (SCI_IIC_SUCCESS != ret)
    {
        /* R_SCI_IIC_GetStatus() 関数コール時のエラー処理 */
        sample_mode = ERROR;
    }
    else
    {
        if (1 == iic_status.BIT.NACK)
        {
            /* iic_status のステータスフラグを確認して
             NACK が検出されていた場合の処理 */
            sample_mode = RETRY_WAIT_DEV_A_WR;
        }
        else
        {
            retry_cnt = 0;
            sample_mode = DEVICE_B_READ;
        }
    }
}

void Callback_deviceB(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_mcu_status_t iic_status;
    sci_iic_info_t iic_info_ch;
    iic_info_ch.ch_no = 1;

```

通信が終了した場合に R_SCI_IIC_Close 関数を呼び出すことで
使用している SCI チャンネルを解放することができる
R_SCI_IIC_Close 関数は主に次の場合に呼び出してください

- ・低消費電力モードに移行する場合
- ・通信異常が発生した場合
- ・使用している SCI のチャンネルを解放したい場合

/* 次に「アイドル中」となる */

/* 次に「エラー時の処理」を行う */

/* 次に「リトライ待ち」を行う */

/* 次に「デバイス B からの読み出し」を行う */

図 5.8 1つのチャンネルで2つのスレーブデバイスにアクセスする例(4)


```
/* 簡易 I2C ステータスの取得 */
ret = R_SCI_IIC_GetStatus(&iic_info_ch, &iic_status);

if (SCI_IIC_SUCCESS != ret)
{
    /* R_SCI_IIC_GetStatus() 関数コール時のエラー処理 */
    sample_mode = ERROR; /* 次に"エラー時の処理"を行う */
}
else
{
    if (1 == iic_status.BIT.NACK)
    {
        /* iic_status のステータスフラグを確認して
        NACK が検出されていた場合の処理 */
        sample_mode = RETRY_WAIT_DEV_B_RD; /* 次に"リトライ待ち"を行う */
    }
    else
    {
        retry_cnt = 0;
        sample_mode = FINISH; /* 次に"通信終了"を行う */
    }
}
}
```

図 5.9 1つのチャンネルで2つのスレーブデバイスにアクセスする例(5)

5.3 2つのチャンネルで2つのスレーブデバイスにアクセスする場合の例

SCI の 2 つのチャンネルを簡易 I²C モードで使用し、各チャンネルはそれぞれ異なるスレーブデバイスに対して書き込みおよび読み出しを行う場合のサンプルコードを示します。

サンプルコードでは、アクセスするデバイスごとに、I²C 通信情報構造体を用意しています。

次の(1)~(3)の順に動作します。

- (1) SCI の ch1 を簡易 I²C FIT モジュールで使用可能にするため、R_SCI_IIC_Open 関数を実行する。
SCI の ch5 を簡易 I²C FIT モジュールで使用可能にするため、R_SCI_IIC_Open 関数を実行する。
- (2) SCI の ch1 を使用し、デバイス A に 3 バイトのデータを書き込むため、R_SCI_IIC_MasterSend 関数を実行する。SCI の ch5 を使用し、デバイス B から 3 バイトのデータを読み出すため、R_SCI_IIC_MasterReceive 関数を実行する。
- (3) SCI の ch1 を簡易 I²C FIT モジュールから解放するため、R_SCI_IIC_Close 関数を実行する。SCI の ch5 を簡易 I²C FIT モジュールから解放するため、R_SCI_IIC_Close 関数を実行する。

```
#include <stddef.h> /* NULL definition */
#include "platform.h"
#include "r_sci_iic_rx_if.h"

/* NACK 検出時のリトライ回数の定義 */
#define RETRY_TMO 10

/* リトライ時の次の通信開始まで待つためのソフトウェアループ回数の定義 */
#define RETRY_WAIT_TIME 1000

/* 送信サイズ */
#define SEND_SIZE 3

/* 受信サイズ */
#define RECEIVE_SIZE 3

/* サンプルコードのモード管理用定義 */
typedef enum
{
    IDLE = 0U, /* アイドル中 */
    BUSY, /* I2C 通信中 */
    INITIALIZE, /* 簡易 I2C FIT モジュールの初期設定 */
    DEVICE_A_WRITE, /* デバイス A への書き込み */
    DEVICE_B_READ, /* デバイス B からの読み出し */
    FINISH, /* 通信終了 */
    RETRY_WAIT_DEV_A_WR, /* デバイス A への書き込みのリトライ待ち */
    RETRY_WAIT_DEV_B_RD, /* デバイス B からの読み出しのリトライ待ち */
    ERROR /* エラー発生 */
} sample_mode_t;

/* サンプルコードのモード管理用変数 */
volatile uint8_t sample_mode_ch1;
volatile uint8_t sample_mode_ch5;

/* リトライ回数管理用変数 */
volatile uint32_t retry_cnt_ch1;
volatile uint32_t retry_cnt_ch5;

void main(void);
void Callback_deviceA(void);
void Callback_deviceB(void);

void main(void)
{
```

プログラムの説明で、
次の略称を使用しています。
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 5.10 2つのチャンネルで2つのスレーブデバイスにアクセスする例(1)

```

volatile sci_iic_return_t  ret;                /* API 関数の戻り値確認用 */
volatile uint32_t retry_wait_cnt_ch1 = 0;     /* リトライ間隔調整用カウンタ */
volatile uint32_t retry_wait_cnt_ch5 = 0;     /* リトライ間隔調整用カウンタ */

sci_iic_info_t iic_info_deviceA;              /* デバイス A 用情報構造体 */
sci_iic_info_t iic_info_deviceB;              /* デバイス B 用情報構造体 */
uint8_t slave_addr_deviceA[1] = {0x50};      /* デバイス A 用スレーブアドレス */
uint8_t slave_addr_deviceB[1] = {0x50};      /* デバイス B 用スレーブアドレス */
uint8_t access_addr_deviceA[1] = {0x00};      /* デバイス A 用アクセス先アドレス */
uint8_t access_addr_deviceB[2] = {0x00, 0x00}; /* デバイス B 用アクセス先アドレス */
uint8_t send_data[5] = {0x81, 0x82, 0x83, 0x84, 0x85}; /* 送信データ */
uint8_t store_area[5] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF}; /* 受信データ格納用 */

sample_mode_ch1 = INITIALIZE;                  /* ch1 は、次に“初期設定”を行う */
sample_mode_ch5 = INITIALIZE;                  /* ch5 は、次に“初期設定”を行う */

while(1)
{
    switch(sample_mode_ch1)
    {
        /* アイドル中 */
        case IDLE:
            /* Nothing to do */
            break;

        /* I2C 通信中 */
        case BUSY:
            /* Nothing to do */
            break;

        /* 簡易 I2C FIT モジュールの初期設定 */
        case INITIALIZE:
            /* チャンネル 1 は通信中か */
            if (SCI_IIC_COMMUNICATION == g_sci_iic_ChStatus[1])
            {
                sample_mode_ch1 = ERROR;        /* ch1 は、次に“エラー時の処理”を行う */
            }
            else
            {
                /* デバイス A の情報構造体の設定 (マスタ送信パターン 1) */
                iic_info_deviceA.p_slv_addr = slave_addr_deviceA;
                iic_info_deviceA.p_data1st = access_addr_deviceA;
                iic_info_deviceA.p_data2nd = send_data;
                iic_info_deviceA.dev_sts = SCI_IIC_NO_INIT;
                iic_info_deviceA.cnt1st = sizeof(access_addr_deviceA);
                iic_info_deviceA.cnt2nd = SEND_SIZE;
                iic_info_deviceA.callbackfunc = &Callback_deviceA;
                iic_info_deviceA.ch_no = 1;
            }

            retry_cnt_ch1 = 0;                    /* リトライ回数の初期化 */

            /* SCI open */
            ret = R_SCI_IIC_Open(&iic_info_deviceA);

            if (SCI_IIC_SUCCESS == ret)
            {
                sample_mode_ch1 = DEVICE_A_WRITE; /* ch1 は、次に“デバイス A への書き込み”を行う */
            }
            else
            {
                /* R_SCI_IIC_Open() 関数コール時のエラー処理 */
                sample_mode_ch1 = ERROR;          /* ch1 は、次に“エラー時の処理”を行う */
            }
            break;

            /* デバイス A への書き込み */
            case DEVICE_A_WRITE:
                retry_cnt_ch1 = retry_cnt_ch1 + 1;
    }
}

```

アクセスするデバイスごとに情報構造体を用意する

異なるチャンネルはそれぞれ同時に動作できるためチャンネルごとにモードを管理する

アイドル中や I²C 通信中は、空処理でループする

チャンネルの状態をグローバル変数の g_sci_iic_ChStatus[] で確認できる

図 5.11 2つのチャンネルで2つのスレーブデバイスにアクセスする例(2)

```

/* マスタ送信の開始 */
ret = R_SCI_IIC_MasterSend(&iic_info_deviceA);

if (SCI_IIC_SUCCESS == ret)
{
    sample_mode_ch1 = BUSY; /* ch1 は、次に"I2C 通信中"となる */
}
else if (SCI_IIC_ERR_BUS_BUSY == ret)
{
    sample_mode_ch1 = RETRY_WAIT_DEV_A_WR; /* ch1 は、次に"リトライ待ち"を行う */
}
else
{
    /* R_SCI_IIC_MasterSend() 関数コール時のエラー処理 */
    sample_mode_ch1 = ERROR; /* ch1 は、次に"エラー時の処理"を行う */
}
break;
/* デバイス A への書き込みのリトライ待ち */
case RETRY_WAIT_DEV_A_WR:
    retry_wait_cnt_ch1 = retry_wait_cnt_ch1 + 1;

    if (RETRY_TMO < retry_cnt_ch1)
    {
        retry_wait_cnt_ch1 = 0;
        sample_mode_ch1 = ERROR; /* ch1 は、次に"エラー時の処理"を行う */
    }

    if (RETRY_WAIT_TIME < retry_wait_cnt_ch1)
    {
        retry_wait_cnt_ch1 = 0;

        switch (sample_mode_ch1)
        {
            case RETRY_WAIT_DEV_A_WR:
                sample_mode_ch1 = DEVICE_A_WRITE; /* ch1 は、次に"デバイス A への書き込み"を行う */

                break;

            default:
                /* Nothing to do */
                break;
        }
    }
}
break;

/* 通信終了 */
case FINISH:
    /* SCI close */
    ret = R_SCI_IIC_Close(&iic_info_deviceA);

    if (SCI_IIC_SUCCESS == ret)
    {
        sample_mode_ch1 = IDLE; /* ch1 は、次に"アイドル中"となる */
    }
    else
    {
        /* R_SCI_IIC_Close() 関数コール時のエラー処理 */
        sample_mode_ch1 = ERROR; /* ch1 は、次に"エラー時の処理"を行う */
    }
}
break;

/* エラー発生 */
case ERROR:
    /* Nothing to do */
    break;

default:
    /* Nothing to do */
    break;
}

```

この関数実行により、ST 生成から SP 生成まで FIT モジュールで処理される
SP 出力後に設定したコールバック関数(Callback_deviceA())が呼び出される

通信相手デバイスが EEPROM の場合、
書き込みコマンドを送信し書くと、
EEPROM の書き込み処理が
完了するまでは、NACK 応答となる
サンプルコードでは、ACK 応答になるまで
通信開始のリトライ処理をしています

通信が終了した場合に R_SCI_IIC_Close 関数を呼び出すことで
使用している SCI チャンネルを解放することができる
R_SCI_IIC_Close 関数は主に次の場合に呼び出してください

- ・低消費電力モードに移行する場合
- ・通信異常が発生した場合
- ・使用している SCI のチャンネルを解放したい場合

図 5.12 2つのチャンネルで2つのスレーブデバイスにアクセスする例(3)

```

switch(sample_mode_ch5)
{
    /* アイドル中 */
    case IDLE:
        /* Nothing to do */
        break;

    /* I2C 通信中 */
    case BUSY:
        /* Nothing to do */
        break;

    /* 簡易 I2C FIT モジュールの初期設定 */
    case INITIALIZE:
        /* チャンネル 5 は通信中か */
        if (SCI_IIC_COMMUNICATION == g_sci_iic_ChStatus[5])
        {
            sample_mode_ch5 = ERROR; /* ch5 は、次に「エラー時の処理」を行う */
        }
        else
        {
            /* デバイス B の情報構造体の設定 (マスタ送受信) */
            iic_info_deviceB.p_slv_adr = slave_addr_deviceB;
            iic_info_deviceB.p_data1st = access_addr_deviceB;
            iic_info_deviceB.p_data2nd = store_area;
            iic_info_deviceB.dev_sts = SCI_IIC_NO_INIT;
            iic_info_deviceB.cnt1st = sizeof(access_addr_deviceB);
            iic_info_deviceB.cnt2nd = RECEIVE_SIZE;
            iic_info_deviceB.callbackfunc = &Callback_deviceB;
            iic_info_deviceB.ch_no = 5;

            retry_cnt_ch5 = 0; /* リトライ回数の初期化 */

            /* SCI open */
            ret = R_SCI_IIC_Open(&iic_info_deviceB);

            if (SCI_IIC_SUCCESS == ret)
            {
                sample_mode_ch5 = DEVICE_B_READ; /* ch5 は、次に「デバイス B からの読み出し」を行う */
            }
            else
            {
                /* R_SCI_IIC_Open() 関数コール時のエラー処理 */
                sample_mode_ch5 = ERROR; /* ch5 は、次に「エラー時の処理」を行う */
            }

            break;

            case DEVICE_B_READ:
                retry_cnt_ch5 = retry_cnt_ch5 + 1;

                /* マスタ送受信の開始 */
                ret = R_SCI_IIC_MasterReceive(&iic_info_deviceB);

                if (SCI_IIC_SUCCESS == ret)
                {
                    sample_mode_ch5 = BUSY; /* ch5 は、次に「I2C 通信中」となる */
                }
                else if (SCI_IIC_ERR_BUS_BUSY == ret)
                {
                    sample_mode_ch5 = RETRY_WAIT_DEV_B_RD; /* ch5 は、次に「リトライ待ち」を行う */
                }
                else
                {
                    /* R_SCI_IIC_MasterReceive() 関数コール時のエラー処理 */
                    sample_mode_ch5 = ERROR; /* ch5 は、次に「エラー時の処理」を行う */
                }
                break;
        }
    }
}

```

アイドル中や I²C 通信中は、空処理でループする

チャンネルの状態をグローバル変数の g_sci_iic_ChStatus[]で確認できる

この関数実行により、ST 生成から SP 生成まで FIT モジュールで処理される SP 出力後に設定したコールバック関数(Callback_deviceB())が呼び出される

図 5.13 2つのチャンネルで2つのスレーブデバイスにアクセスする例(4)

```

/* デバイス B からの読み出しのリトライ待ち */
case RETRY_WAIT_DEV_B_RD:
    retry_wait_cnt_ch5 = retry_wait_cnt_ch5 + 1;

    if (RETRY_TMO < retry_cnt_ch5)
    {
        retry_wait_cnt_ch5 = 0;
        sample_mode_ch5 = ERROR; /* ch5 は、次に"エラー時の処理"を行う */
    }

    if (RETRY_WAIT_TIME < retry_wait_cnt_ch5)
    {
        retry_wait_cnt_ch5 = 0;

        switch (sample_mode_ch5)
        {
            case RETRY_WAIT_DEV_B_RD:
                sample_mode_ch5 = DEVICE_B_READ; /* ch5 は、次に"デバイス B からの読み出し"を行う */
                break;

            default:
                /* Nothing to do */
                break;
        }
    }
    break;

/* 通信終了 */
case FINISH:
    /* SCI close */
    ret = R_SCI_IIC_Close(&iic_info_deviceB);

    if (SCI_IIC_SUCCESS == ret)
    {
        sample_mode_ch5 = IDLE; /* ch5 は、次に"アイドル中"となる */
    }
    else
    {
        /* R_SCI_IIC_Close() 関数コール時のエラー処理 */
        sample_mode_ch5 = ERROR; /* ch5 は、次に"エラー時の処理"を行う */
    }
    break;

/* エラー発生 */
case ERROR:
    /* Nothing to do */
    break;

default:
    /* Nothing to do */
    break;
}

}

void Callback_deviceA(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_mcu_status_t iic_status;
    sci_iic_info_t iic_info_ch;
    iic_info_ch.ch_no = 1;

    /* 簡易 I2C ステータスの取得 */
    ret = R_SCI_IIC_GetStatus(&iic_info_ch, &iic_status);

    if (SCI_IIC_SUCCESS != ret)
    {
        /* R_SCI_IIC_GetStatus() 関数コール時のエラー処理 */
        sample_mode_ch1 = ERROR; /* ch1 は、次に"エラー時の処理"を行う */
    }
}

```

通信相手デバイスが EEPROM の場合、書き込みコマンドを送信し書くと、EEPROM の書き込み処理が完了するまでは、NACK 応答となるサンプルコードでは、ACK 応答になるまで通信開始のリトライ処理をしています

通信が終了した場合に R_SCI_IIC_Close 関数を呼び出すことで使用している SCI チャンネルを解放することができる
R_SCI_IIC_Close 関数は主に次の場合に呼び出してください

- ・低消費電力モードに移行する場合
- ・通信異常が発生した場合
- ・使用している SCI のチャンネルを解放したい場合

図 5.14 2つのチャンネルで2つのスレーブデバイスにアクセスする例(5)

```

else
{
    if (1 == iic_status.BIT.NACK)
    {
        /* iic_status のステータスフラグを確認して
           NACK が検出されていた場合の処理 */
        sample_mode_ch1 = RETRY_WAIT_DEV_A_WR; /* ch1 は、次に"リトライ待ち"を行う */
    }
    else
    {
        retry_cnt_ch1 = 0;
        sample_mode_ch1 = FINISH; /* ch1 は、次に"通信終了"を行う */
    }
}
}

void Callback_deviceB(void)
{
    volatile sci_iic_return_t ret;
    sci_iic_mcu_status_t iic_status;
    sci_iic_info_t iic_info_ch;
    iic_info_ch.ch_no = 5;

    /* 簡易 I2C ステータスの取得 */
    ret = R_SCI_IIC_GetStatus(&iic_info_ch, &iic_status);

    if (SCI_IIC_SUCCESS != ret)
    {
        /* R_SCI_IIC_GetStatus() 関数コール時のエラー処理 */
        sample_mode_ch5 = ERROR; /* ch5 は、次に"エラー時の処理"を行う */
    }
    else
    {
        if (1 == iic_status.BIT.NACK)
        {
            /* iic_status のステータスフラグを確認して
               NACK が検出されていた場合の処理 */
            sample_mode_ch5 = RETRY_WAIT_DEV_B_RD; /* ch5 は、次に"リトライ待ち"を行う */
        }
        else
        {
            retry_cnt_ch5 = 0;
            sample_mode_ch5 = FINISH; /* ch5 は、次に"通信終了"を行う */
        }
    }
}
}

```

図 5.15 2つのチャンネルで2つのスレーブデバイスにアクセスする例(6)

6. 提供するモジュール

提供するモジュールは、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。

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。

- なし

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>

お問い合わせ先

<http://japan.renesas.com/contact/>

改訂記録	RX ファミリ 簡易 I ² C モジュール Firmware Integration Technology
------	--

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2013.07.01	—	初版作成
1.10	2013.11.15	—	戻り値の変更
1.20	2014.07.01	3	FIT モジュールの RX100 シリーズ対応、ポート選択機能追加の改定により、ROM サイズ、RAM サイズ、スタックサイズを変更 「表 1.2 必要メモリサイズ」 <ul style="list-style-type: none"> ・ ROM : 4629 バイト → 4617 バイト ・ RAM : 300 バイト → 41 バイト ・ 最大使用ユーザスタック : 136 バイト → 140 バイト ・ 最大使用割り込みスタック : 160 バイト → 148 バイト
		13	FIT モジュールの RX100 シリーズ対応、ポート選択機能追加の改定時、動作確認を行ったツールチェーンバージョンを更新 「2.3 サポートされているツールチェーン」 Renesas RX Toolchain v.2.00 ↓ Renesas RX Toolchain v.2.01
		15	FIT モジュールのポート選択機能を追加したため、オプションの定義を追加
		17	「2.9 モジュールの追加方法」の説明を変更
		—	FIT モジュールが RX64M グループに対応
1.30	2014.09.22	1	「関連ドキュメント」の項目を追加
		3	「制限事項」に次の内容を追加 <ul style="list-style-type: none"> ・ 多重割り込みには対応していません。 ・ コールバック関数内では R_SCI_IIC_GetStatus 関数以外の API 関数のコールは禁止です。 ・ 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。
		4	ROM サイズ、RAM サイズ、スタックサイズを変更 ※RX111 グループ使用時のサイズを掲載 「表 1.2 必要メモリサイズ」 <ul style="list-style-type: none"> ・ ROM : 4617 バイト → 4518 バイト ・ RAM : 41 バイト → 41 バイト ・ 最大使用ユーザスタック : 140 バイト → 148 バイト ・ 最大使用割り込みスタック : 148 バイト → 152 バイト
		16	誤記の訂正 r_cgc_rx に依存していないため、「2.2 ソフトウェアの要求」のパッケージ依存から r_cgc_rx を削除
		17,18	対応チャネルの追加 <ul style="list-style-type: none"> ・ 対応チャネル : 1,5,12 → 0~9,12 コンフィギュレーションオプションの追加 <ul style="list-style-type: none"> ・ SCI_IIC_CFG_PORT_SETTING_PROCESSING →ポート設定処理の有効 / 無効の選択オプションを追加
		24,25,28	R_SCI_IIC_MasterSend 関数と R_SCI_IIC_MasterReceive 関数、R_SCI_IIC_GetStatus 関数の説明の「Example」に記載のサンプルプログラムについて、ヘッダファイルのインクルード、コールバック関数のコード追加など
		—	FIT モジュールが RX113 グループに対応
1.40	2014.12.01	—	FIT モジュールが RX71M グループに対応
1.50	2014.12.15	3	制限事項の追加 「DTC と組み合わせて使用することはできません。」 →「DMAC、DTC と組み合わせて使用することはできません。」
		20	「2.8 引数」に説明を追加

改訂記録	RX ファミリ 簡易 I ² C モジュール Firmware Integration Technology
------	--

Rev.	発行日	改訂内容	
		ページ	ポイント
1.50	2014.12.15	21~33	「3. API 関数」に説明を追加
		47~61	「5. サンプルコード」の章を追加
1.60	2015.2.27	—	FIT モジュールが RX63N グループに対応
		17,18	対応チャネルの追加 ・対応チャネル：0~9,12 → 0~12
		4	ROM サイズ、RAM サイズ、スタックサイズを変更 ※RX111 グループ使用時のサイズを掲載 「表 1.2 必要メモリサイズ」 ・ROM : 4518 バイト → 4297 バイト ・RAM : 41 バイト → 41 バイト ・最大使用ユーザスタック : 148 バイト → 124 バイト ・最大使用割り込みスタック : 152 バイト → 132 バイト
		プログラム	ソフトウェア不具合のため、簡易 I ² C FIT モジュールを改修 ■内容 内蔵ポーレートジェネレータのクロックソース(SMR レジスタの CKS ビット)およびビットレート(BRR レジスタ)の設定処理に誤りがあり、期待する設定値から乖離した値が設定される場合があります。 ■発生条件 簡易 I ² C FIT モジュール Rev.1.50 以前のバージョンで、RX64M または RX71M をご使用されており、次のいずれかの条件に該当したとき ・PLL 入力分周比(PLLCR レジスタの PLIDIV ビット)で 3 分周を選択している ・PLL 周波数通倍率(PLLCR レジスタの STC ビット)で小数第 1 位が 5 の倍率を選択している ■対策 簡易 I ² C FIT モジュール Rev1.60 以降をご使用ください。
1.70	2015.05.29	—	FIT モジュールが RX231 グループに対応
		—	FIT モジュールが RX130 グループ、RX230 グループ、RX23T グループに対応
1.80	2015.10.31	31	「3.5 R_SCI_IIC_GetStatus()」 「Format」を変更

改訂記録	RX ファミリ 簡易 I ² C モジュール Firmware Integration Technology
------	--

Rev.	発行日	改訂内容	
		ページ	ポイント
1.90	2016.03.04	—	FIT モジュールが RX24T グループに対応
		4	「表 1.2 必要メモリサイズ」の説明を変更
		17,19	「2.6 コンパイル時の設定」に r_sci_iic_rx_pin_config.h について説明を追記
		—	「マスタ複合」の表記を「マスタ送受信」に変更
		44	「表 4.5 状態遷移時の各フラグの状態一覧」 通信中(マスタ送受信)状態での I ² C プロトコルの動作モード 内部通信情報構造体 api_Mode のマクロ定義を変更
2.00	2016.10.1	—	FIT モジュールが RX65N グループに対応
		17	「2.6 コンパイル時の設定」 SCI_IIC_CFG_CHI_SSDA_DELAY_SELECT のデフォルト値を変更
		20	コードサイズの説明「表 1.2 必要メモリサイズ」から「2.7 コードサイズ」に変更。

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、お客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
2. 本資料に記載されている情報は、正確を期すため慎重に作成したのですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
3. 本資料に記載された製品デ・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に関し、当社は、何らの責任を負うものではありません。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. 当社製品を改造、改変、複製等しないでください。かかる改造、改変、複製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）、交通用信号機器、
防災・防犯装置、各種安全装置等

当社製品は、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（原子力制御システム、軍事機器等）に使用されることを意図しておらず、使用することはできません。たとえ、意図しない用途に当社製品を使用したことによりお客様または第三者に損害が生じて、当社は一切その責任を負いません。なお、ご不明点がある場合は、当社営業にお問い合わせください。

6. 当社製品をご使用の際は、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他の保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障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、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、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。お客様が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. 本資料に記載されている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。また、当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事情報に使用しないでください。当社製品または技術を輸出する場合は、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。
10. お客様の転売等により、本ご注意書き記載の諸条件に抵触して当社製品が使用され、その使用から損害が生じた場合、当社は何らの責任も負わず、お客様にてご負担して頂きますのでご了承ください。
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。

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とその総株主の議決権の過半数を直接または間接に保有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。



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：<http://japan.renesas.com/contact/>