RX ファミリ

簡易 I²C モジュール Firmware Integration Technology

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、RX24U グループ
- RX64M グループ
- RX65N、RX651 グループ
- RX66T グループ
- RX71M グループ
- RX72T 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

各コンパイラの動作確認内容については「6.3 動作確認環境詳細」を参照してください。

目次

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)
- コンディション波形の生成
- 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。
- SCI(簡易 I²C モード) FIT モジュールと SCI モジュール Firmware Integration Technology(R01AN1815)を組み合わせて使用するとき、同じチャネルを同時に使用することはできません。

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、2.11 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 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。
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 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。
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、複数のチャネルを制御することができます。また複数チャネルを持つデバイスでは複数チャ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。

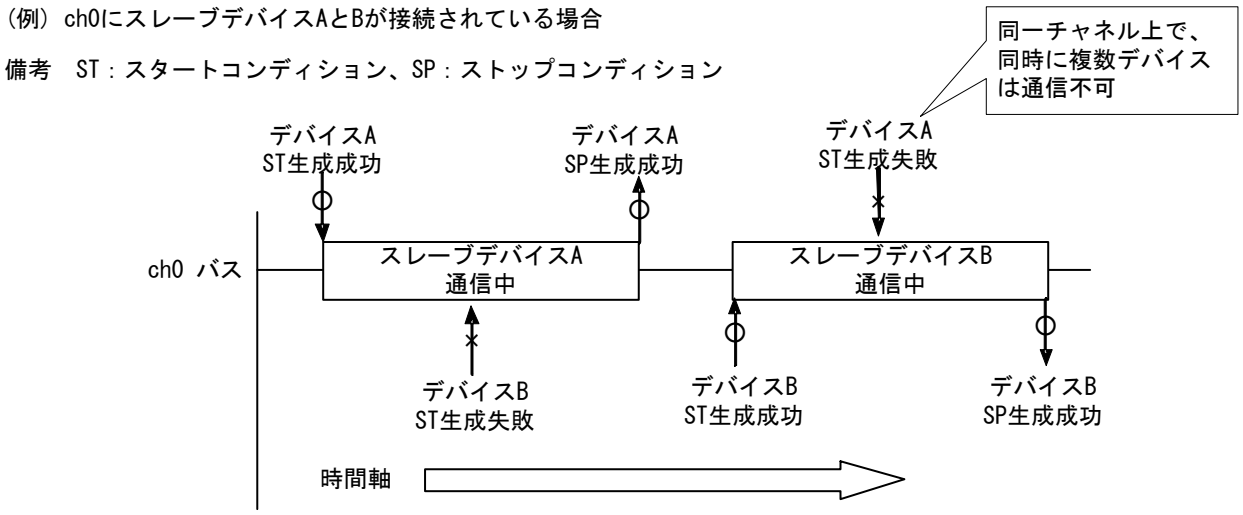


図 1.1 複数スレーブデバイス制御例

1.3.2　マスタ送信の処理

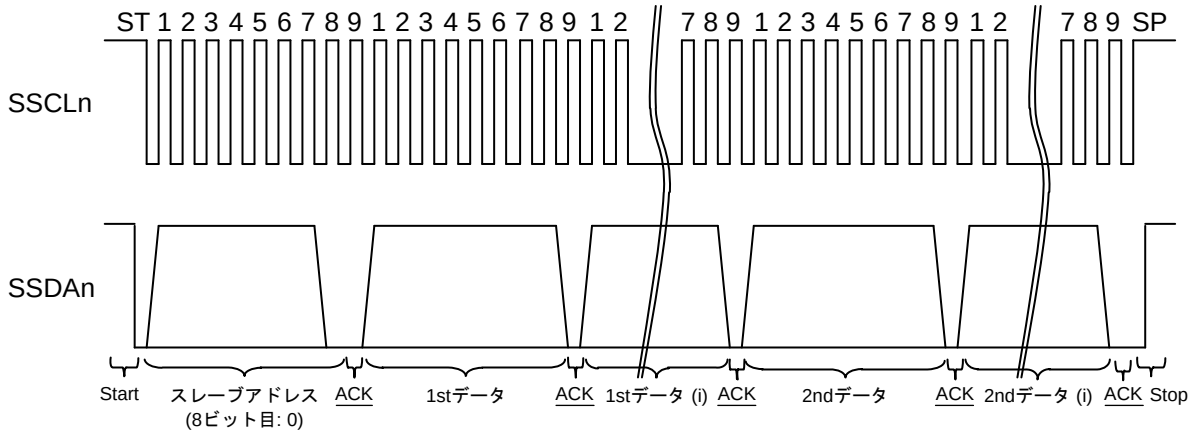
マスタデバイス (RX MCU) からスレーブデバイスへデータを送信します。

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、「2.9 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。



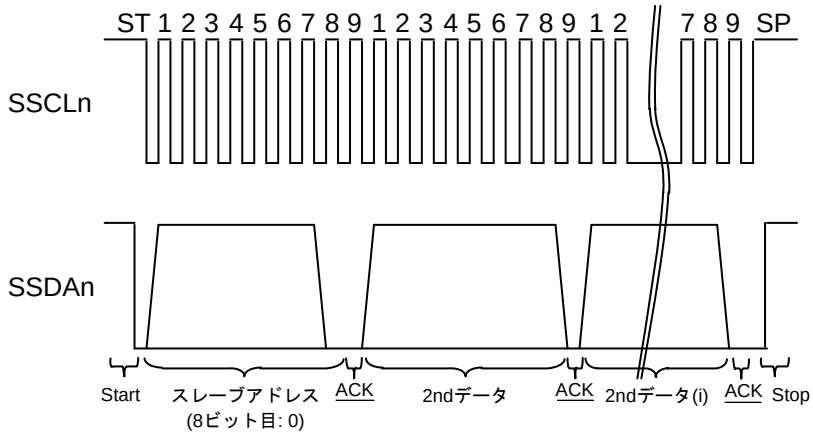
凡例:
n　：チャネル番号
ST：スタートコンディション生成
SP：ストップコンディション生成
ACK：Acknowledge“0”
※ 下線のある信号は、スレーブからマスタへのデータ送信です

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(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。



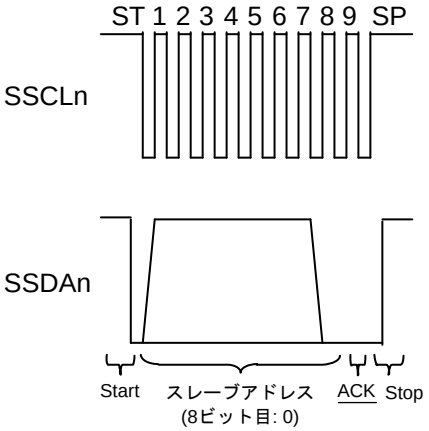
凡例:
n　：チャネル番号
ST：スタートコンディション生成
SP：ストップコンディション生成
ACK：Acknowledge“0”
※ 下線のある信号は、スレーブからマスタへのデータ送信です

図 1.3 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。

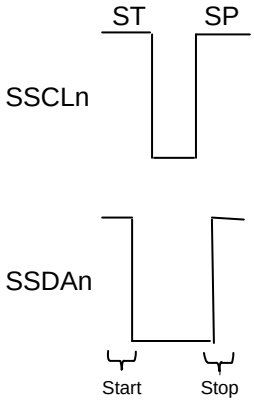


凡例:
n　：チャネル番号
ST：スタートコンディション生成
SP：ストップコンディション生成
ACK：Acknowledge“0”
※ 下線のある信号は、スレーブからマスタへのデータ送信です

図 1.4 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。



凡例:
n　：チャネル番号
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

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。

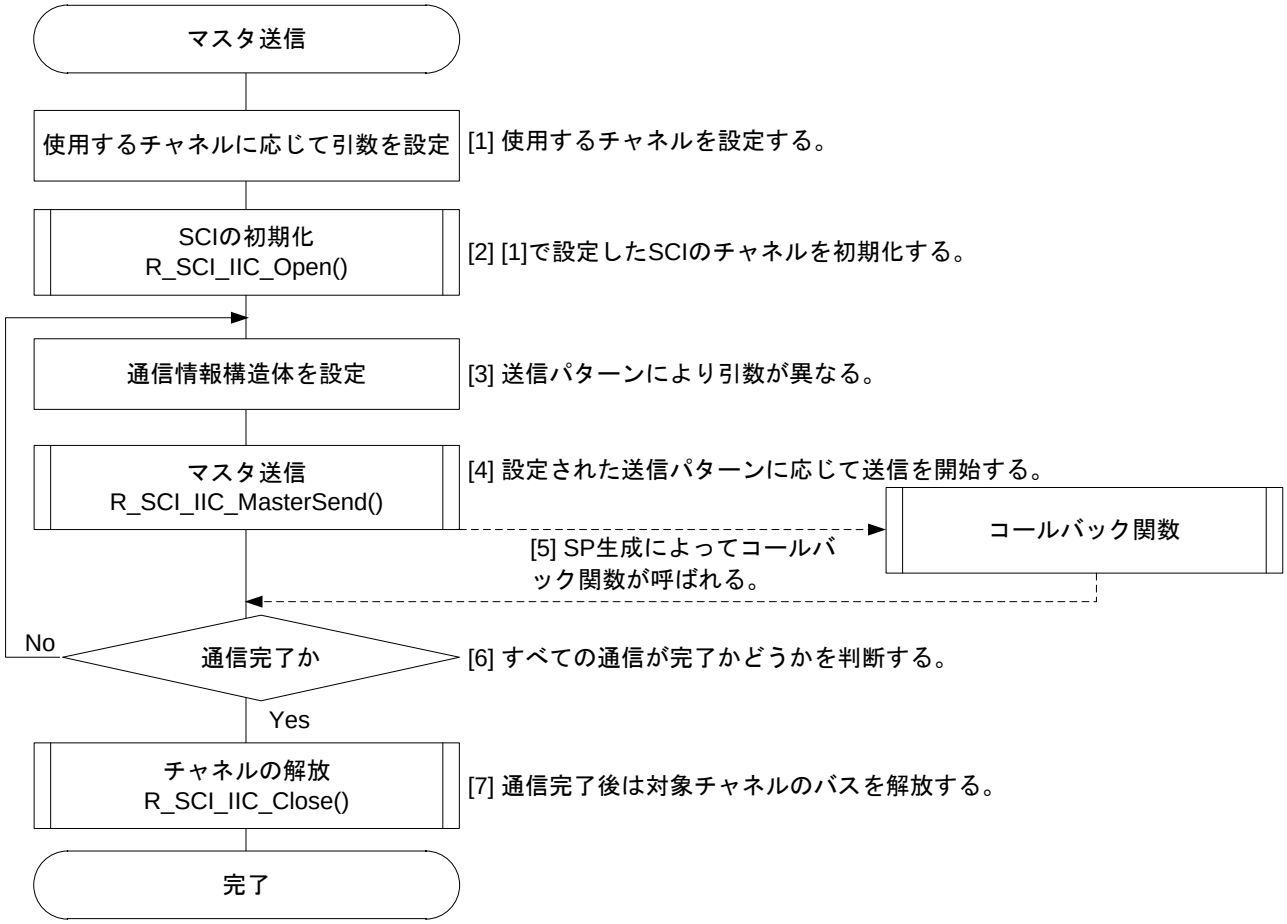


図 1.6 マスタ送信の処理例

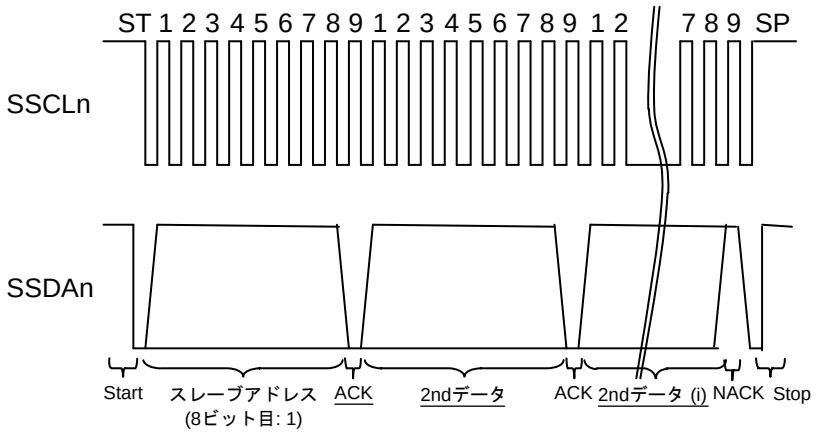
1.3.3　マスタ受信の処理

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 I^2C 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。I^2C 通信情報構造体の詳細は、「2.9 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。



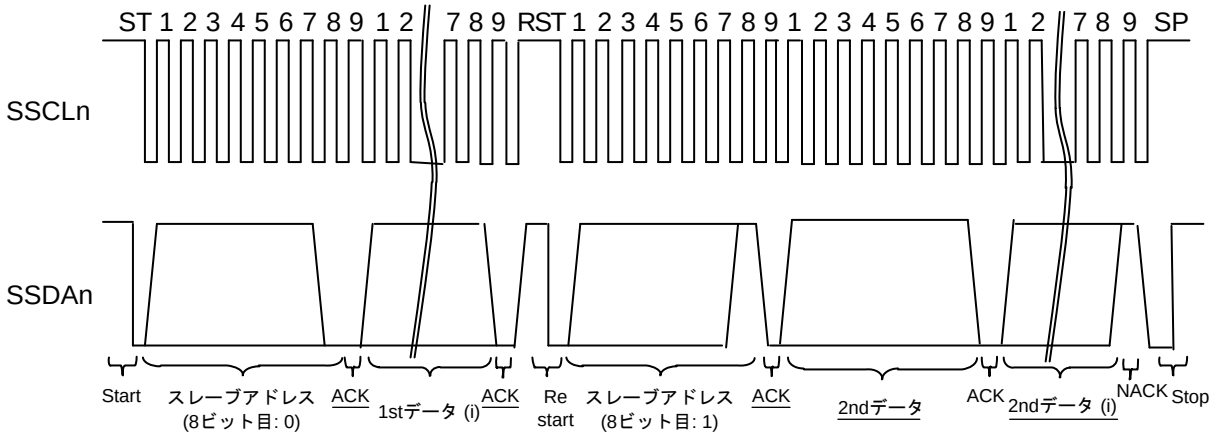
凡例:
n ：チャネル番号
ST: スタートコンディション生成　　NACK: Acknowledge “1”
SP: ストップコンディション生成　　ACK: Acknowledge “0”
※ 下線のある信号は、スレーブからマスタへのデータ送信です

図 1.7 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。



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 1.8 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。

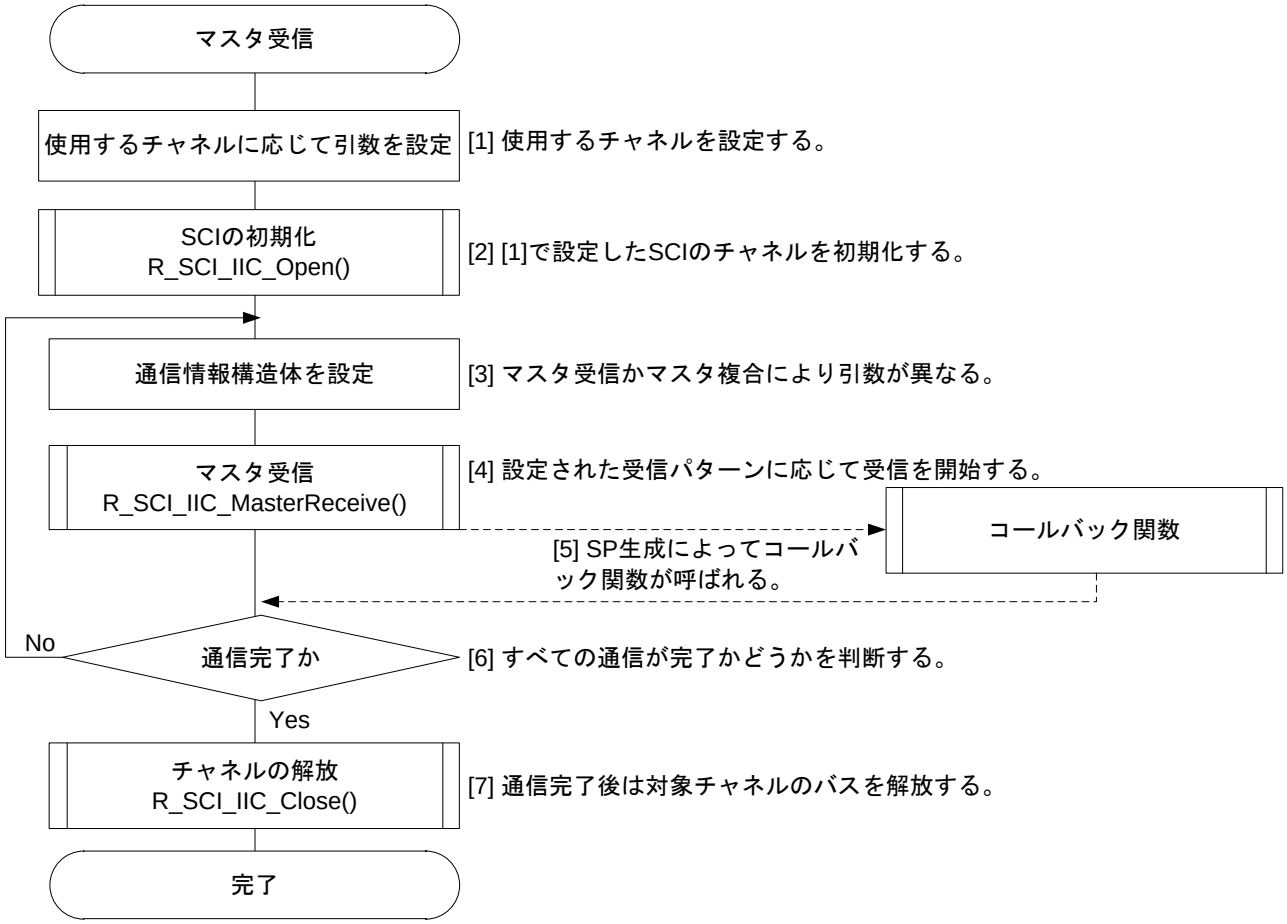


図 1.9 マスタ受信の処理例

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。

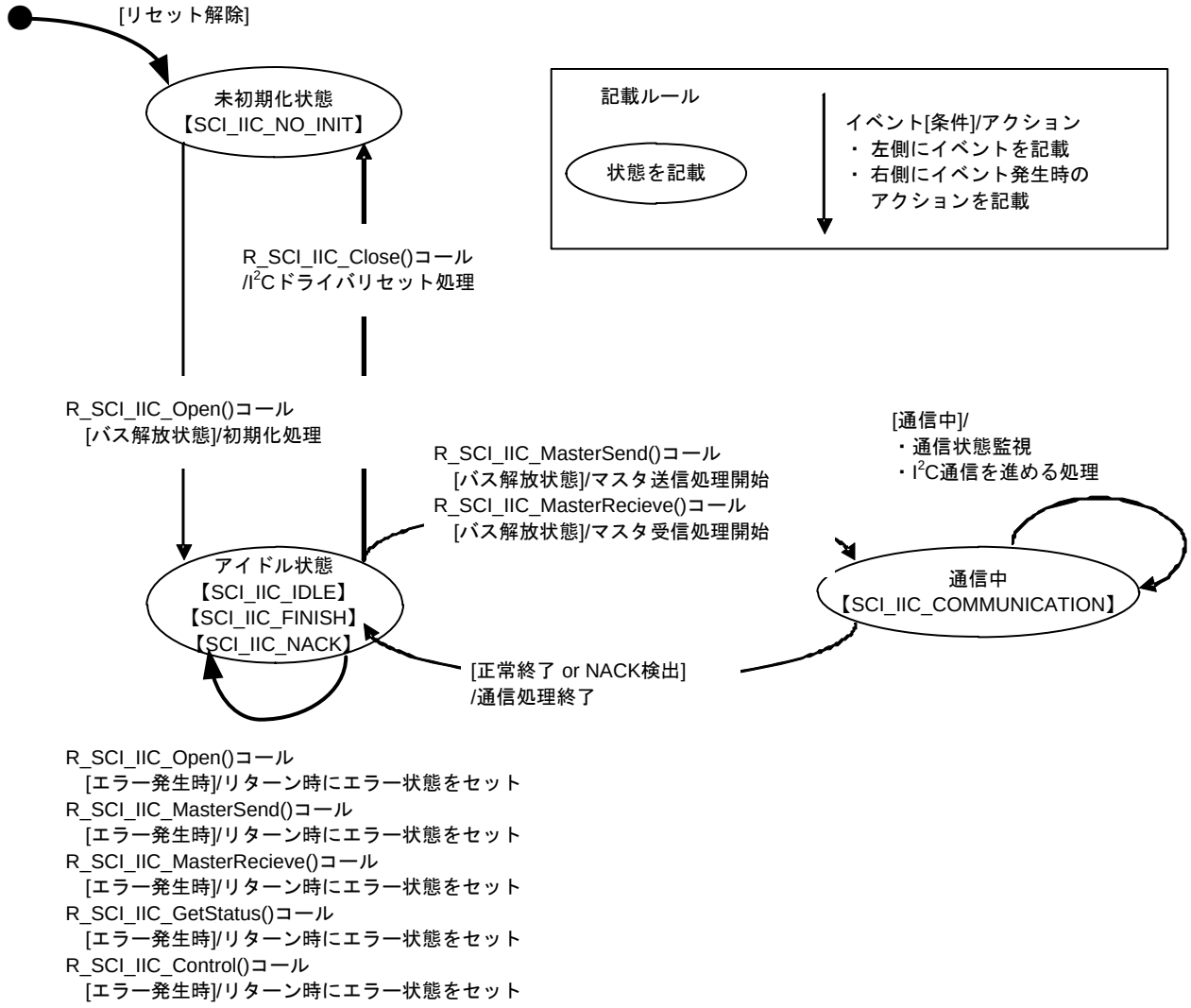


図 1.10 状態遷移図

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。

- ボードサポートパッケージモジュール (r_bsp) Rev.5.00 以上

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。詳細は、「6.3 動作確認環境詳細」を参照ください。

- Renesas RX Toolchain v.2.02.00
- Renesas RX Toolchain v.2.03.00
- Renesas RX Toolchain v.2.05.00
- Renesas RX Toolchain v.2.06.00
- Renesas RX Toolchain v.2.07.00
- Renesas RX Toolchain v.3.00.00
- Renesas RX Toolchain v.3.01.00

2.4　使用する割り込みベクタ

R_SCI_IIC_MasterSend 関数、R_SCI_IIC_MasterReceive 関数を実行したとき、引数のパラメータで指定したチャネル番号のチャネルに対応した TXI 割り込み、TEI 割り込みが有効になります。表 2.1～表 2.3 に SCI(簡易 I²C モード) 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覧(1)

デバイス	割り込みベクタ
RX110 RX111	TXI1 割り込み[チャネル 1]（ベクタ番号: 220） TEI1 割り込み[チャネル 1]（ベクタ番号: 221） TXI5 割り込み[チャネル 5]（ベクタ番号: 224） TEI5 割り込み[チャネル 5]（ベクタ番号: 225） TXI12 割り込み[チャネル 12]（ベクタ番号: 240） TEI12 割り込み[チャネル 12]（ベクタ番号: 241）
RX113 RX130 RX230 RX231	TXI0 割り込み[チャネル 0]（ベクタ番号: 216） TEI0 割り込み[チャネル 0]（ベクタ番号: 217） TXI1 割り込み[チャネル 1]（ベクタ番号: 220） TEI1 割り込み[チャネル 1]（ベクタ番号: 221） TXI5 割り込み[チャネル 5]（ベクタ番号: 224） TEI5 割り込み[チャネル 5]（ベクタ番号: 225） TXI6 割り込み[チャネル 6]（ベクタ番号: 228） TEI6 割り込み[チャネル 6]（ベクタ番号: 229） TXI8 割り込み[チャネル 8]（ベクタ番号: 232） TEI8 割り込み[チャネル 8]（ベクタ番号: 233） TXI9 割り込み[チャネル 9]（ベクタ番号: 236） TEI9 割り込み[チャネル 9]（ベクタ番号: 237） TXI12 割り込み[チャネル 12]（ベクタ番号: 240） TEI12 割り込み[チャネル 12]（ベクタ番号: 241）
RX23T	TXI1 割り込み[チャネル 1]（ベクタ番号: 220） TEI1 割り込み[チャネル 1]（ベクタ番号: 221） TXI5 割り込み[チャネル 5]（ベクタ番号: 224） TEI5 割り込み[チャネル 5]（ベクタ番号: 225）
RX24T	TXI1 割り込み[チャネル 1]（ベクタ番号: 220） TEI1 割り込み[チャネル 1]（ベクタ番号: 221） TXI5 割り込み[チャネル 5]（ベクタ番号: 224） TEI5 割り込み[チャネル 5]（ベクタ番号: 225） TXI6 割り込み[チャネル 6]（ベクタ番号: 228） TEI6 割り込み[チャネル 6]（ベクタ番号: 229）
RX24U	TXI1 割り込み[チャネル 1]（ベクタ番号: 220） TEI1 割り込み[チャネル 1]（ベクタ番号: 221） TXI5 割り込み[チャネル 5]（ベクタ番号: 224） TEI5 割り込み[チャネル 5]（ベクタ番号: 225） TXI6 割り込み[チャネル 6]（ベクタ番号: 228） TEI6 割り込み[チャネル 6]（ベクタ番号: 229） TXI8 割り込み[チャネル 8]（ベクタ番号: 232） TEI8 割り込み[チャネル 8]（ベクタ番号: 233） TXI9 割り込み[チャネル 9]（ベクタ番号: 236） TEI9 割り込み[チャネル 9]（ベクタ番号: 237） TXI11 割り込み[チャネル 11]（ベクタ番号: 252） TEI11 割り込み[チャネル 11]（ベクタ番号: 253）

表 2.2　使用する割り込みベクター覧(2)

デバイス	割り込みベクタ
RX64M RX71M	TXI0 割り込み[チャネル 0]（ベクタ番号: 59） TXI1 割り込み[チャネル 1]（ベクタ番号: 61） TXI2 割り込み[チャネル 2]（ベクタ番号: 63） TXI3 割り込み[チャネル 3]（ベクタ番号: 81） TXI4 割り込み[チャネル 4]（ベクタ番号: 83） TXI5 割り込み[チャネル 5]（ベクタ番号: 85） TXI6 割り込み[チャネル 6]（ベクタ番号: 87） TXI7 割り込み[チャネル 7]（ベクタ番号: 99） TXI12 割り込み[チャネル 12]（ベクタ番号: 117） GROUPBL0 割り込み（ベクタ番号: 110） ● TEI0 割り込み[チャネル 0]（グループ割り込み要因番号: 0） ● TEI1 割り込み[チャネル 1]（グループ割り込み要因番号: 2） ● TEI2 割り込み[チャネル 2]（グループ割り込み要因番号: 4） ● TEI3 割り込み[チャネル 3]（グループ割り込み要因番号: 6） ● TEI4 割り込み[チャネル 4]（グループ割り込み要因番号: 8） ● TEI5 割り込み[チャネル 5]（グループ割り込み要因番号: 10） ● TEI6 割り込み[チャネル 6]（グループ割り込み要因番号: 12） ● TEI7 割り込み[チャネル 7]（グループ割り込み要因番号: 14） ● TEI12 割り込み[チャネル 12]（グループ割り込み要因番号: 16）

表 2.3　使用する割り込みベクター覧(3)

デバイス	割り込みベクタ
RX65N RX651	TXI0 割り込み[チャネル 0]（ベクタ番号: 59） TXI1 割り込み[チャネル 1]（ベクタ番号: 61） TXI2 割り込み[チャネル 2]（ベクタ番号: 63） TXI3 割り込み[チャネル 3]（ベクタ番号: 81） TXI4 割り込み[チャネル 4]（ベクタ番号: 83） TXI5 割り込み[チャネル 5]（ベクタ番号: 85） TXI6 割り込み[チャネル 6]（ベクタ番号: 87） TXI7 割り込み[チャネル 7]（ベクタ番号: 99） TXI8 割り込み[チャネル 8]（ベクタ番号: 101） TXI9 割り込み[チャネル 9]（ベクタ番号: 103） TXI10 割り込み[チャネル 10]（ベクタ番号: 105） TXI11 割り込み[チャネル 11]（ベクタ番号: 115） TXI12 割り込み[チャネル 12]（ベクタ番号: 117） GROUPBL0 割り込み（ベクタ番号: 110） ● TEI0 割り込み[チャネル 0]（グループ割り込み要因番号: 0） ● TEI1 割り込み[チャネル 1]（グループ割り込み要因番号: 2） ● TEI2 割り込み[チャネル 2]（グループ割り込み要因番号: 4） ● TEI3 割り込み[チャネル 3]（グループ割り込み要因番号: 6） ● TEI4 割り込み[チャネル 4]（グループ割り込み要因番号: 8） ● TEI5 割り込み[チャネル 5]（グループ割り込み要因番号: 10） ● TEI6 割り込み[チャネル 6]（グループ割り込み要因番号: 12） ● TEI7 割り込み[チャネル 7]（グループ割り込み要因番号: 14） ● TEI12 割り込み[チャネル 12]（グループ割り込み要因番号: 16） GROUPBL1 割り込み（ベクタ番号: 111） ● TEI8 割り込み[チャネル 8]（グループ割り込み要因番号: 24） ● TEI9 割り込み[チャネル 9]（グループ割り込み要因番号: 26） GROUPAL0 割り込み（ベクタ番号: 112） ● TEI10 割り込み[チャネル 10]（グループ割り込み要因番号: 8） ● TEI11 割り込み[チャネル 11]（グループ割り込み要因番号: 12）

表 2.4　使用する割り込みベクター覧(4)

デバイス	割り込みベクタ
RX66T RX72T	TXI1 割り込み[チャネル 1]（ベクタ番号: 61） TXI5 割り込み[チャネル 5]（ベクタ番号: 85） TXI6 割り込み[チャネル 6]（ベクタ番号: 87） TXI8 割り込み[チャネル 8]（ベクタ番号: 101） TXI9 割り込み[チャネル 9]（ベクタ番号: 103） TXI11 割り込み[チャネル 11]（ベクタ番号: 115） TXI12 割り込み[チャネル 12]（ベクタ番号: 117） GROUPBL0 割り込み（ベクタ番号: 110） ● TEI1 割り込み[チャネル 1]（グループ割り込み要因番号: 2） ● TEI5 割り込み[チャネル 5]（グループ割り込み要因番号: 10） ● TEI6 割り込み[チャネル 6]（グループ割り込み要因番号: 12） ● TEI12 割り込み[チャネル 12]（グループ割り込み要因番号: 16） GROUPBL1 割り込み（ベクタ番号: 111） ● TEI8 割り込み[チャネル 8]（グループ割り込み要因番号: 24） ● TEI9 割り込み[チャネル 9]（グループ割り込み要因番号: 26） GROUPAL0 割り込み（ベクタ番号: 112） ● TEI11 割り込み[チャネル 11]（グループ割り込み要因番号: 12）

2.5　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。

2.6　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_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 *r_sci_iic_rx_config.h* (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=0～12 ※i = 0～12 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。 使用時に、使用するチャネルの定義値を”1”に変更してください。
SCI_IIC_CFG_CHi_BITRATE_BPS 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。 ビットレートは本設定値と RX ファミリ ボードサポートパッケージモジュール(BSP FIT モジュール) のクロックの設定の定義値を元に設定されます。使用する対象デバイスと BSP FIT モジュールのクロックの設定によっては、実際のビットレートが期待するビットレートと異なる場合があります。
SCI_IIC_CFG_CHi_INT_PRIORITY 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=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=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=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、内蔵ボ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の遅延時間は内蔵ボ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の立ち下がりタイミングがSSCLの立ち下が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=0～12 ※デフォルト値は“1000”	簡易I²C 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I²C 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) ≒ (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_SCIi_SSCLi_PORT、 R_SCI_IIC_CFG_SCIi_SSCLi_BIT、 R_SCI_IIC_CFG_SCIi_SSDAi_PORT、 R_SCI_IIC_CFG_SCIi_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。 この設定を“0”とした場合でも、上記の4つの定義は設定してください。

Configuration options in *r_sci_iic_rx_pin_config.h*	
R_SCI_IIC_CFG_SCIi_SSCLi_PORT i=0～12 ※i = 0 のデフォルト値は’2’ ※i = 1 のデフォルト値は’1’ ※i = 2 のデフォルト値は’5’ ※i = 3 のデフォルト値は’2’ ※i = 4 のデフォルト値は’B’ ※i = 5 のデフォルト値は’B’ ※i = 6 のデフォルト値は’B’ ※i = 7 のデフォルト値は’9’ ※i = 8 のデフォルト値は’C’ ※i = 9 のデフォルト値は’B’ ※i = 10 のデフォルト値は’8’ ※i = 11 のデフォルト値は’7’ ※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’～’K’ (ASCIIコード)の範囲で設定してください。
R_SCI_IIC_CFG_SCIi_SSCLi_BIT i=0～12 ※i = 0 のデフォルト値は’1’ ※i = 1 のデフォルト値は’5’ ※i = 2 のデフォルト値は’2’ ※i = 3 のデフォルト値は’5’ ※i = 4 のデフォルト値は’0’ ※i = 5 のデフォルト値は’1’ ※i = 6 のデフォルト値は’1’ ※i = 7 のデフォルト値は’2’ ※i = 8 のデフォルト値は’6’ ※i = 9 のデフォルト値は’6’ ※i = 10 のデフォルト値は’1’ ※i = 11 のデフォルト値は’6’ ※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=0～12 ※i = 0 のデフォルト値は’2’ ※i = 1 のデフォルト値は’1’ ※i = 2 のデフォルト値は’5’ ※i = 3 のデフォルト値は’2’ ※i = 4 のデフォルト値は’B’ ※i = 5 のデフォルト値は’B’ ※i = 6 のデフォルト値は’B’ ※i = 7 のデフォルト値は’9’ ※i = 8 のデフォルト値は’C’ ※i = 9 のデフォルト値は’B’ ※i = 10 のデフォルト値は’8’ ※i = 11 のデフォルト値は’7’ ※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’～’K’ (ASCIIコード)の範囲で設定してください。
R_SCI_IIC_CFG_SCIi_SSDAi_BIT i=0～12 ※i = 0 のデフォルト値は’0’ ※i = 1 のデフォルト値は’6’ ※i = 2 のデフォルト値は’0’ ※i = 3 のデフォルト値は’3’ ※i = 4 のデフォルト値は’1’ ※i = 5 のデフォルト値は’2’ ※i = 6 のデフォルト値は’2’ ※i = 7 のデフォルト値は’0’ ※i = 8 のデフォルト値は’7’ ※i = 9 のデフォルト値は’7’ ※i = 10 のデフォルト値は’2’ ※i = 11 のデフォルト値は’7’ ※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。

2.8　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「2.7 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_sci_iic_rx rev2.41

コンパイラバージョン: Renesas Electronics C/C++ Compiler Package for RX Family V3.01.00
(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 4.08.04.201803
(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 4.10.1
(統合開発環境のデフォルト設定)

コンフィグ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
RX130	ROM	1 チャネル使用	4365 バイト	4248 バイト	9269 バイト	9176 バイト	7823 バイト	7555 バイト
		2 チャネル使用	4513 バイト	4396 バイト	9444 バイト	9324 バイト	7960 バイト	7692 バイト
	RAM	1 チャネル使用	41 バイト		16 バイト		34 バイト	
		2 チャネル使用	69 バイト		16 バイト		50 バイト	
	スタック（注 1）		256 バイト		-		276 バイト	
RX231	ROM	1 チャネル使用	4357 バイト	4240 バイト	7720 バイト	7592 バイト	7825 バイト	7557 バイト
		2 チャネル使用	4505 バイト	4388 バイト	7868 バイト	7740 バイト	7962 バイト	7694 バイト
	RAM	1 チャネル使用	41 バイト		16 バイト		34 バイト	
		2 チャネル使用	69 バイト		16 バイト		50 バイト	
	スタック（注 1）		248 バイト		-		276 バイト	
RX64M	ROM	1 チャネル使用	4393 バイト	4296 バイト	7768 バイト	7640 バイト	7858 バイト	7630 バイト
		2 チャネル使用	4559 バイト	4442 バイト	7916 バイト	7788 バイト	7994 バイト	7766 バイト
	RAM	1 チャネル使用	41 バイト		16 バイト		36 バイト	
		2 チャネル使用	69 バイト		16 バイト		52 バイト	
	スタック（注 1）		248 バイト		-		312 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_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;
```

2.10　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_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;
```

2.11　モジュールの追加方法

本モジュールは、使用するプロジェクトごとに追加する必要があります。ルネサスでは、Smart Configurator を使用した(1)、(3)の追加方法を推奨しています。ただし、Smart Configurator は、一部の RX デバイスのみ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) e^2 studio 上で Smart Configurator を使用して FIT モジュールを追加する場合

e^2 studio の Smart Configurator 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「Renesas e^2 studio スマート・コンフィグレータ ユーザーガイド (R20AN0451)」を参照してください。

(2) e^2 studio 上で FIT Configurator を使用して FIT モジュールを追加する場合

e^2 studio の FIT Configurator 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 e^2 studio に組み込む方法 Firmware Integration Technology (R01AN1723)」を参照してください。

(3) CS+上で Smart Configurator を使用して FIT モジュールを追加する場合

CS+上で、スタンドアロン版 Smart Configurator 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「Renesas e^2 studio スマート・コンフィグレータ ユーザーガイド (R20AN0451)」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。

2.12　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

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

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 のチャネルを設定します。チャネルの状態が“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:
簡易 I²C バス通信を開始するための初期設定の中でビットレートを設定します。ビットレートは、「2.7 コンパイル時の設定」の設定値と BSP FIT モジュールのクロックの設定の定義値を元に設定されます。

使用する対象デバイスと BSP FIT モジュールのクロックの設定によっては、実際のビットレートが期待するビットレートと異なる場合があります。

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

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 */
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 のチャネル、送信パターンで送信します。チャネルの状態が“アイドル状態”(SCI_IIC_IDLE)の場合、次の処理を行います。

ー 状態フラグの設定
ー API で使用する変数の初期化
ー SCI 割り込みの許可
ー I²C のリセット解除
ー 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

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 SCI_IIC_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 SCI_IIC_ERR_BUS_BUSY を返します。(注 1)

- 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 SCI_IIC_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_sci_iic_ChStatus[ ]が”SCI_IIC_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、SCI_IIC_ERR_BUS_BUSY を返すことがあります。

Reentrant

- 異なるチャネルからリエントラントは可能です。

Example

```
- 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);

    if (SCI_IIC_SUCCESS == ret)
    {
        while(SCI_IIC_FINISH != siic_info.dev_sts);
    }
    else
    {
        /* error */
    }

    /* Master send complete */
    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	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_MasterRecive(
        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 2.9 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(SCI_IIC_COMMUNICATION)に構造体の内容を書き換えないでください。

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 */
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 のチャネル、受信パターンで受信します。チャネルの状態が“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

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 SCI_IIC_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 SCI_IIC_ERR_BUS_BUSY を返します。(注 1)

- 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 SCI_IIC_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、「6.2.2 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_sci_iic_ChStatus[ ]が”SCI_IIC_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、SCI_IIC_ERR_BUS_BUSY を返すことがあります。

Reentrant

- 異なるチャネルからリエントラントは可能です。

Example

```
#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);

    if (SCI_IIC_SUCCESS == ret)
```

```
    {
        while(SCI_IIC_FINISH != siic_info.dev_sts);
    }
    else
    {
        /* error */
    }

    /* Master receive complete */
    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 データ（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	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 2.9 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(SCI_IIC_COMMUNICATION)に構造体の内容を書き換えないでください。

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

```
uint8_t     ch_no;          /* チャネル番号 */
```

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

　　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 */
```

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:
なし。

4. 端子設定

SCI (簡易 I²C モード) 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。

SCI (簡易 I²C モード) FIT モジュールは、コンフィグレーションオプションの SCI_IIC_CFG_PORT_SETTING_PROCESSING の設定により、R_SCI_IIC_Open／R_SCI_IIC_MasterSend／R_SCI_IIC_MasterReceive／R_SCI_IIC_Close／R_SCI_IIC_Control 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。

e² studio 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_sci_iic_pin_config.h に反映され、表 4.1、表 4.2 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。SCI (簡易 I²C モード) 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(1)

選択したチャネル	選択した端子	マクロ定義
チャネル 0	SSCL0 端子	R_SCI_IIC_CFG_SCI0_SSCL0_PORT R_SCI_IIC_CFG_SCI0_SSCL0_BIT
	SSDA0 端子	R_SCI_IIC_CFG_SCI0_SSDA0_PORT R_SCI_IIC_CFG_SCI0_SSDA0_BIT
チャネル 1	SSCL1 端子	R_SCI_IIC_CFG_SCI1_SSCL1_PORT R_SCI_IIC_CFG_SCI1_SSCL1_BIT
	SSDA1 端子	R_SCI_IIC_CFG_SCI1_SSDA1_PORT R_SCI_IIC_CFG_SCI1_SSDA1_BIT
チャネル 2	SSCL2 端子	R_SCI_IIC_CFG_SCI2_SSCL2_PORT R_SCI_IIC_CFG_SCI2_SSCL2_BIT
	SSDA2 端子	R_SCI_IIC_CFG_SCI2_SSDA2_PORT R_SCI_IIC_CFG_SCI2_SSDA2_BIT
チャネル 3	SSCL3 端子	R_SCI_IIC_CFG_SCI3_SSCL3_PORT R_SCI_IIC_CFG_SCI3_SSCL3_BIT
	SSDA3 端子	R_SCI_IIC_CFG_SCI3_SSDA3_PORT R_SCI_IIC_CFG_SCI3_SSDA3_BIT
チャネル 4	SSCL4 端子	R_SCI_IIC_CFG_SCI4_SSCL4_PORT R_SCI_IIC_CFG_SCI4_SSCL4_BIT
	SSDA4 端子	R_SCI_IIC_CFG_SCI4_SSDA4_PORT R_SCI_IIC_CFG_SCI4_SSDA4_BIT
チャネル 5	SSCL5 端子	R_SCI_IIC_CFG_SCI5_SSCL5_PORT R_SCI_IIC_CFG_SCI5_SSCL5_BIT
	SSDA5 端子	R_SCI_IIC_CFG_SCI5_SSDA5_PORT R_SCI_IIC_CFG_SCI5_SSDA5_BIT
チャネル 6	SSCL6 端子	R_SCI_IIC_CFG_SCI6_SSCL6_PORT R_SCI_IIC_CFG_SCI6_SSCL6_BIT
	SSDA6 端子	R_SCI_IIC_CFG_SCI6_SSDA6_PORT R_SCI_IIC_CFG_SCI6_SSDA6_BIT

表 4.2 端子設定マクロ定義(2)

選択したチャネル	選択した端子	マクロ定義
チャネル 7	SSCL7 端子	R_SCI_IIC_CFG_SCI7_SSCL7_PORT R_SCI_IIC_CFG_SCI7_SSCL7_BIT
	SSDA7 端子	R_SCI_IIC_CFG_SCI7_SSDA7_PORT R_SCI_IIC_CFG_SCI7_SSDA7_BIT
チャネル 8	SSCL8 端子	R_SCI_IIC_CFG_SCI8_SSCL8_PORT R_SCI_IIC_CFG_SCI8_SSCL8_BIT
	SSDA8 端子	R_SCI_IIC_CFG_SCI8_SSDA8_PORT R_SCI_IIC_CFG_SCI8_SSDA8_BIT
チャネル 9	SSCL9 端子	R_SCI_IIC_CFG_SCI9_SSCL9_PORT R_SCI_IIC_CFG_SCI9_SSCL9_BIT
	SSDA9 端子	R_SCI_IIC_CFG_SCI9_SSDA9_PORT R_SCI_IIC_CFG_SCI9_SSDA9_BIT
チャネル 10	SSCL10 端子	R_SCI_IIC_CFG_SCI10_SSCL10_PORT R_SCI_IIC_CFG_SCI10_SSCL10_BIT
	SSDA10 端子	R_SCI_IIC_CFG_SCI10_SSDA10_PORT R_SCI_IIC_CFG_SCI10_SSDA10_BIT
チャネル 11	SSCL11 端子	R_SCI_IIC_CFG_SCI11_SSCL11_PORT R_SCI_IIC_CFG_SCI11_SSCL11_BIT
	SSDA11 端子	R_SCI_IIC_CFG_SCI11_SSDA11_PORT R_SCI_IIC_CFG_SCI11_SSDA11_BIT
チャネル 12	SSCL12 端子	R_SCI_IIC_CFG_SCI12_SSCL12_PORT R_SCI_IIC_CFG_SCI12_SSCL12_BIT
	SSDA12 端子	R_SCI_IIC_CFG_SCI12_SSDA12_PORT R_SCI_IIC_CFG_SCI12_SSDA12_BIT

r_sci_iic_pin_config.h で選択した端子は、R_SCI_IIC_MasterSend／R_SCI_IIC_MasterReceive／R_SCI_IIC_Control 関数呼び出し後、周辺機能端子として SSCL 端子、SSDA 端子となります。

周辺機能端子の割り付けは R_SCI_IIC_MasterSend／R_SCI_IIC_MasterReceive／R_SCI_IIC_Control 関数で実行した通信動作を完了するか、もしくは R_SCI_IIC_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、SSCL 端子、SSDA 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 SCI_IIC_CFG_PORT_SETTING_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_SCI_IIC_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 sciiic_send_demo_rskrx64m

説明:

RSKRX64M（FIT モジュール“r_sci_iic_rx”）向けの RX64M SCI (簡易 I2C モード)マスタ送信を行うデモです。デモでは SCI (簡易 I2C モード) FIT モジュールの API（r_sci_iic_rx_if.h に記載）を使って、マスタデバイスとして、スレーブデバイスへデータを送信します。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:

- RSKRX64M

5.2 sciiic_receive_demo_rskrx64m

説明:

RSKRX64M（FIT モジュール“r_sci_iic_rx”）向けの RX64M SCI (簡易 I2C モード)マスタ受信を行うデモです。デモでは SCI (簡易 I2C モード) FIT モジュールの API（r_sci_iic_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:

- RSKRX64M

5.3 sciiic_send_demo_rskrx231

説明:

RSKRX231（FIT モジュール“r_sci_iic_rx”）向けの RX231 SCI (簡易 I2C モード)マスタ送信を行うデモです。デモの内容は RX64M と同じです。

対応ボード:

- RSKRX231

5.4 sciiic_receive_demo_rskrx231

説明:

RSKRX231（FIT モジュール “r_sci_iic_rx”）向けの RX231 SCI (簡易 I²C モード)マスタ受信を行うデモです。デモの内容は RX64M と同じです。

対応ボード:

- RSKRX231

5.5 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。

5.6 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。

6.1.1 API 動作の状態

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 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

6.1.2 API 動作中のイベント

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 6.2 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)

6.1.3　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 6.1～図 6.4 に各プロトコルの状態遷移を示します。

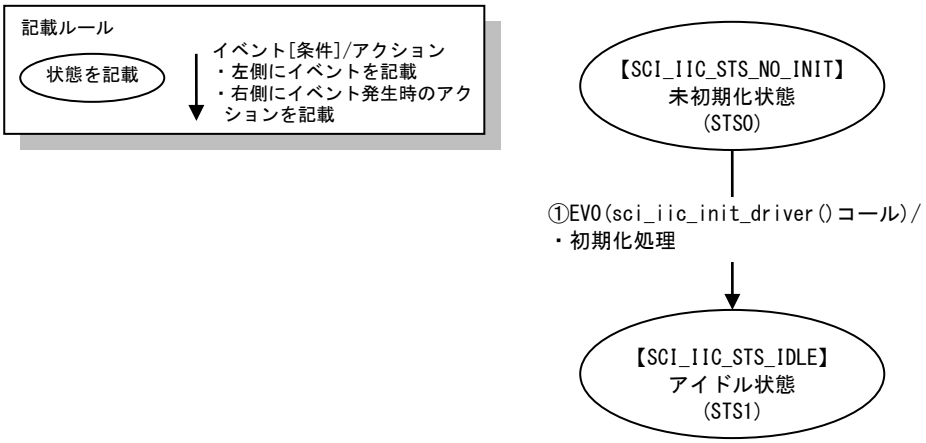


図 6.1 初期化状態遷移図

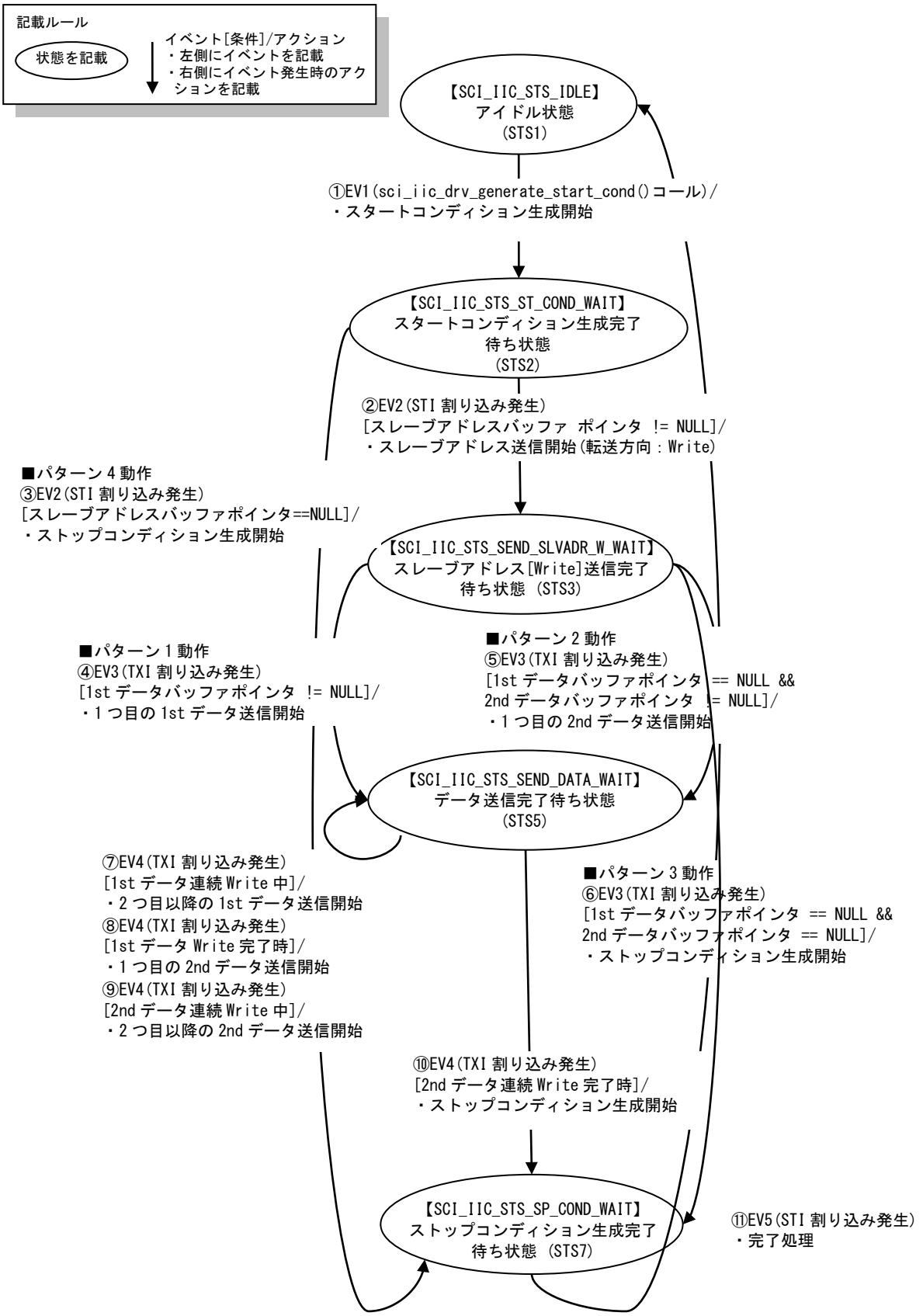


図 6.2 マスタ送信 状態遷移図

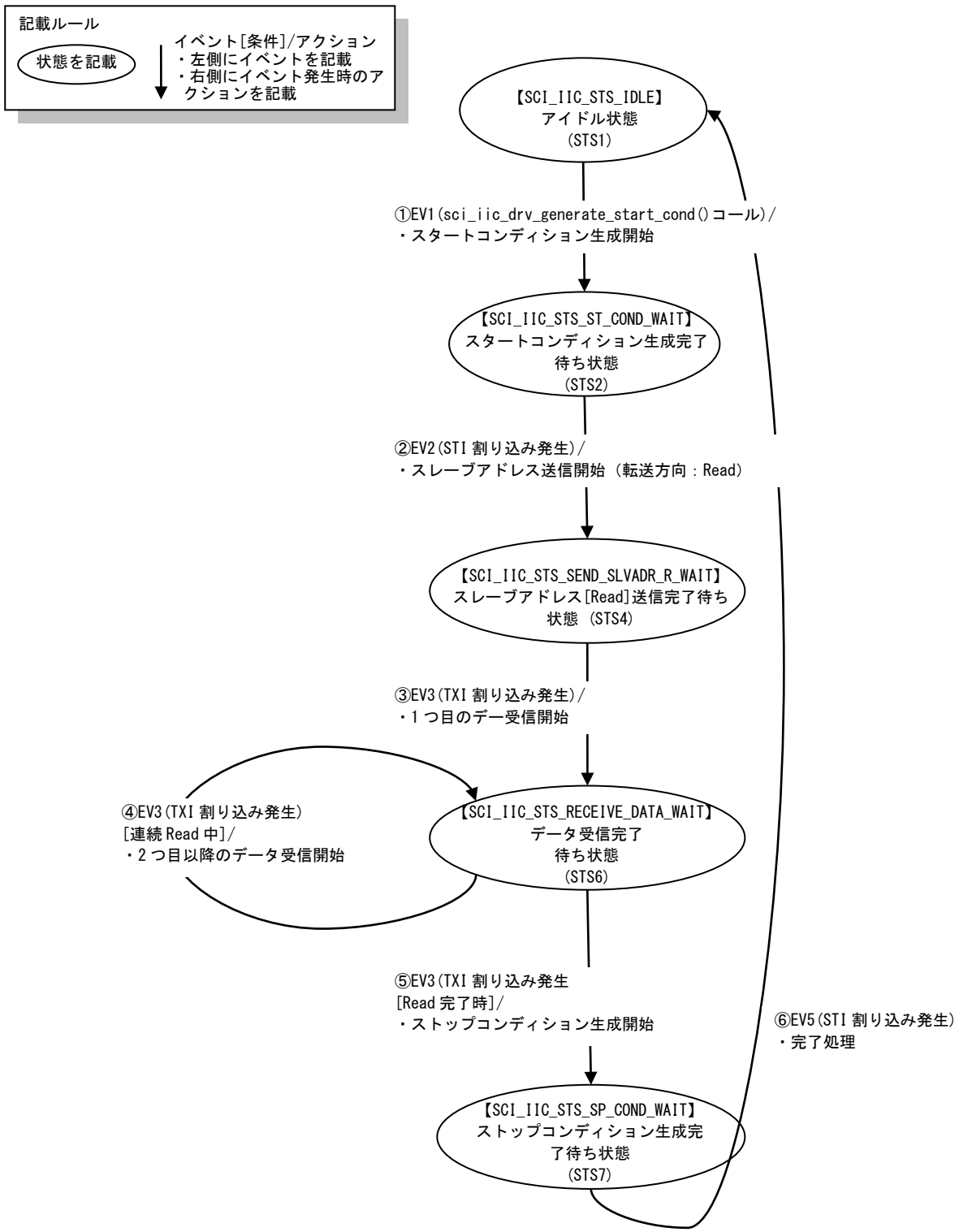


図 6.3 マスタ受信 状態遷移図

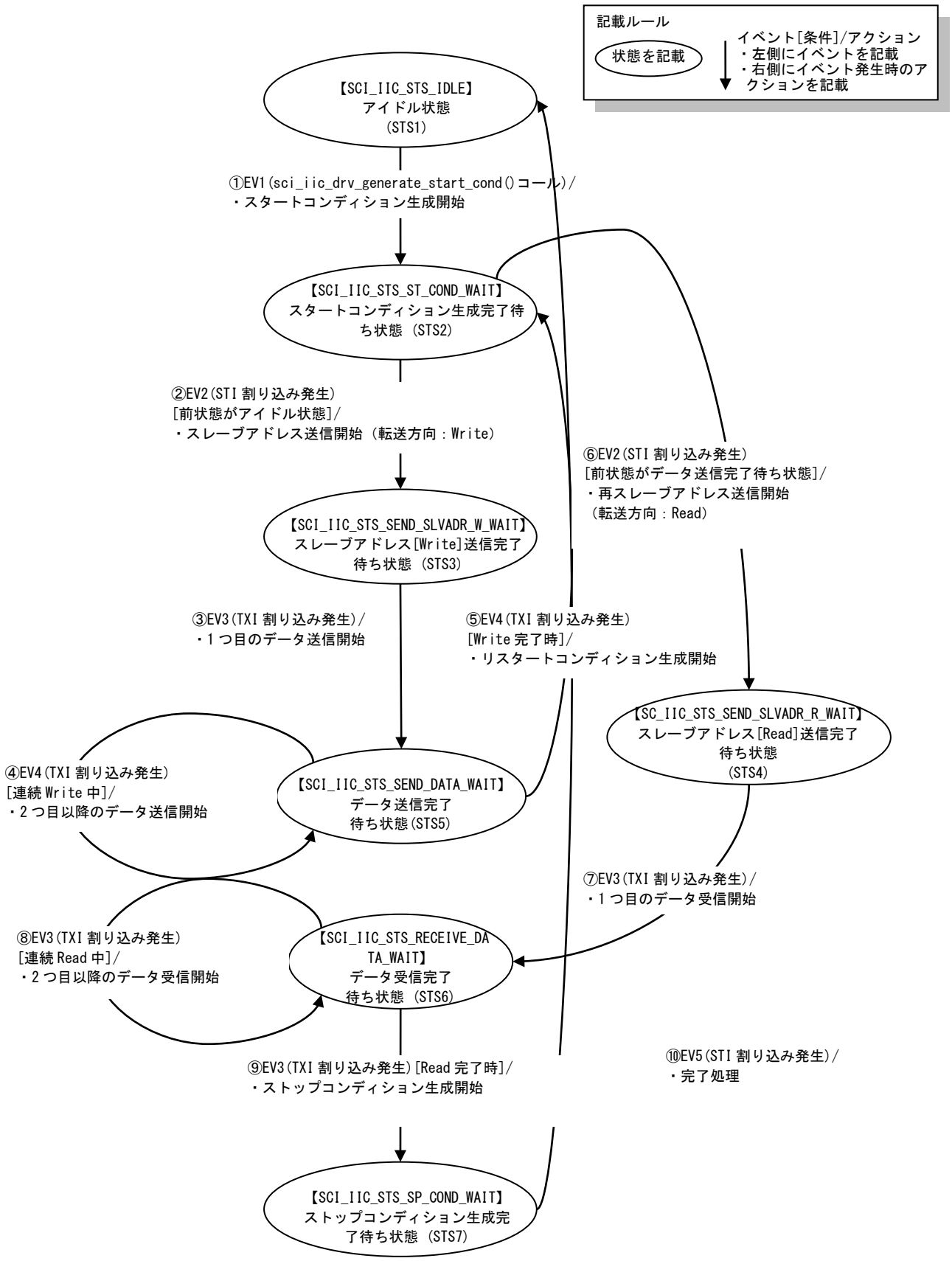


図 6.4 マスタ送受信 状態遷移図

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。Func 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_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。

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	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

6.1.6　状態遷移時の各フラグの状態

1）各チャネル状態管理

チャネル状態フラグ 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。

2) 各デバイス状態管理

I^2C 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	チャネル状態フラグ	デバイス状態フラグ (通信のデバイス)	I²C 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[]	I²C 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
				SCI_IIC _STS_SP_COND_WAIT
エラー	SCI_IIC _ERROR	SCI_IIC _ERROR	-	-

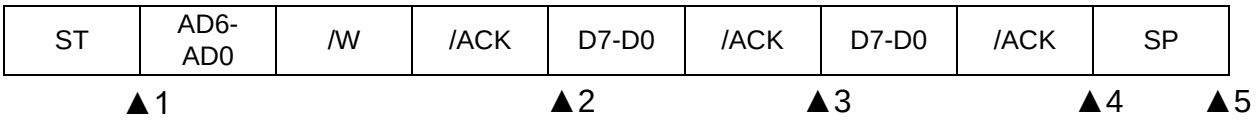
6.2　割り込み発生タイミング

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ン

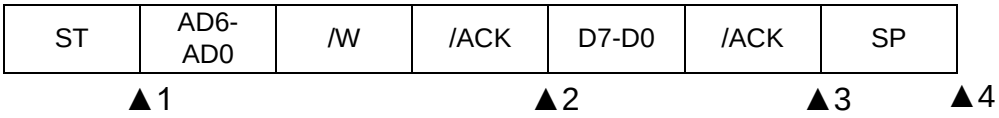
6.2.1　マスタ送信

(1)　パターン 1



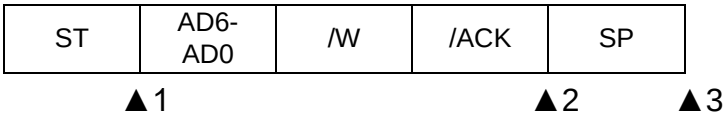
▲1：STI(START)割り込み・・・スタートコンディション検出
▲2：TXI 割り込み・・・アドレス送信完了(転送方向ビット：Write)　※1
▲3：TXI 割り込み・・・データ送信完了(1st データ)　※1
▲4：TXI 割り込み・・・データ送信完了(2nd データ)　※1
▲5：STI(STOP)割り込み・・・ストップコンディション検出

(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

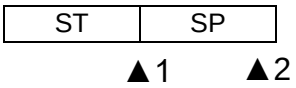


▲1：STI(START)割り込み・・・スタートコンディション検出

▲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

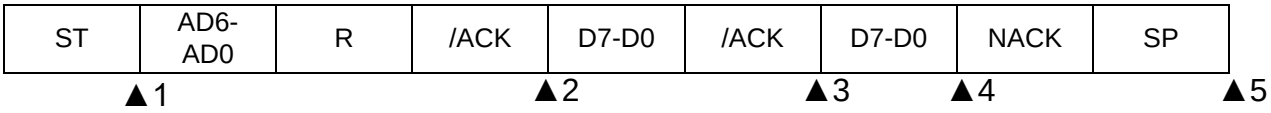
(4)　パターン 4



▲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

6.2.2　　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

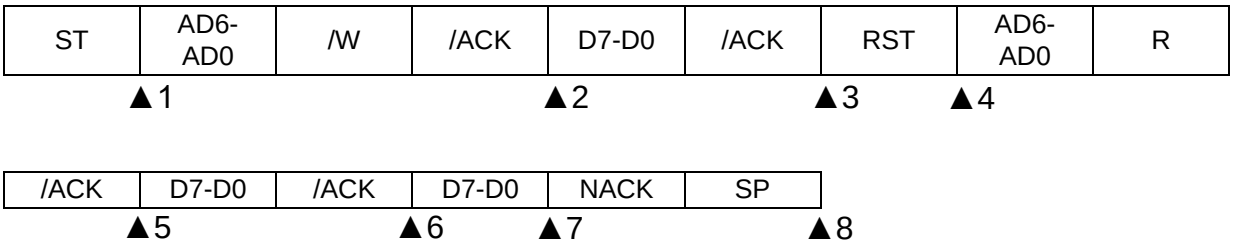
▲2：TXI 割り込み・・・アドレス送信完了(転送方向ビット：Read)　※1

▲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

6.2.3　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

6.3　動作確認環境詳細

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.1.60、Rev.1.70）

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 V3.1.2.09
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 V2.02.00
	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
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.1.60、Rev.1.70
使用ボード	Renesas Starter Kit for RX111（型名：R0K505111SxxxBE） Renesas Starter Kit for RX113（型名：R0K505113SxxxBE） Renesas Starter Kit for RX231（型名：R0K505231SxxxBE） Renesas Starter Kit+ for RX63N（型名：R0K50563NSxxxBE） Renesas Starter Kit+ for RX64M（型名：R0K50564MSxxxBE） Renesas Starter Kit+ for RX71M（型名：R0K50571MSxxxBE）

表 6.7 動作確認環境（Rev.1.80）

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 V4.0.2.008
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 V2.03.00
	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
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.1.80
使用ボード	Renesas Starter Kit for RX130（型名：R0K505113SxxxBE） Renesas Starter Kit for RX23T（型名：RTK500523TSxxxxxBE）

表 6.8 動作確認環境（Rev.1.90）

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 V4.1.0.018
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 V2.03.00
	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
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.1.90
使用ボード	Renesas Starter Kit for RX111（型名：R0K505111SxxxBE） Renesas Starter Kit for RX113（型名：R0K505113SxxxBE） Renesas Starter Kit for RX130（型名：RTK5005130SxxxxxBE） Renesas Starter Kit for RX231（型名：R0K505231SxxxBE） Renesas Starter Kit for RX23T（型名：RTK500523TSxxxxxBE） Renesas Starter Kit for RX24T（型名：RTK500524TSxxxxxBE） Renesas Starter Kit+ for RX63N（型名：R0K50563NSxxxBE） Renesas Starter Kit+ for RX64M（型名：R0K50564MSxxxBE） Renesas Starter Kit+ for RX71M（型名：R0K50571MSxxxBE）

表 6.9 動作確認環境(Rev.2.00)

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 V5.0.1.005
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 V2.05.00
	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
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.2.00
使用ボード	Renesas Starter Kit for RX111（型名：R0K505111SxxxBE） Renesas Starter Kit for RX130（型名：RTK5005130SxxxxxBE） Renesas Starter Kit for RX231（型名：R0K505231SxxxBE） Renesas Starter Kit for RX23T（型名：RTK500523TSxxxxxBE） Renesas Starter Kit for RX24T（型名：RTK500524TSxxxxxBE） Renesas Starter Kit+ for RX63N（型名：R0K50563NSxxxBE） Renesas Starter Kit+ for RX64M（型名：R0K50564MSxxxBE） Renesas Starter Kit+ for RX65N（型名：RTK500565NSxxxxxBE） Renesas Starter Kit+ for RX71M（型名：R0K50571MSxxxBE）

表 6.10 動作確認環境(Rev.2.20)

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 V6.0.0.001
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 V2.06.00 ルネサスエレクトロニクス製 C/C++ Compiler for RX Family V2.07.00
	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
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.2.20
使用ボード	Renesas Starter Kit for RX24U（型名：RTK500524USxxxxxBE） Renesas Starter Kit for RX130-512KB（型名：RTK5051308SxxxxxBE） Renesas Starter Kit+ for RX65N-2MB（型名：RTK50565N2SxxxxxBE）

表 6.11 動作確認環境(Rev.2.30)

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 V7.0.0
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 V3.00.00
	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
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.2.30
使用ボード	Renesas Starter Kit for RX66T（型名：RTK50566T0SxxxxxBE）

表 6.12 動作確認環境(Rev.2.31)

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 V7.1.0
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 V3.00.00
	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
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.2.31

表 6.13 動作確認環境(Rev.2.40)

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 V7.3.0
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 V3.01.00
	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
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.2.40
使用ボード	Renesas Starter Kit for RX72T（型名：RTK5572Txxxxxxxxx）

表 6.14 動作確認環境(Rev.2.41)

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 V7.3.0 IAR Embedded Workbench for Renesas RX 4.10.01
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.3.01.00 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 4.08.04.201803 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 4.10.01 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.2.41
使用ボード	Renesas Starter Kit+ for RX65N（型名：RTK500565Nxxxxxx）

6.4　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)」
- e^2 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 e^2 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_sci_iic_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 – SCI_IIC_CFG_xxx_xxx - …」エラーが発生します。

A：“r_sci_iic_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_sci_iic_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。

7. サンプルコード

7.1 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場合の例

SCI の 1 つのチャネルを簡易 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。

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

```
#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 */
```

図 6.1　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(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_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;
                }

                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 6.2　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)

```
                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;
```

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

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 6.3　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(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

図 6.4　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(4)

7.2　1 つのチャネルで 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。

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

```
#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 */
```

通信するデバイスの数だけ、
情報構造体を宣言する

図 6.5　1 つのチャネルで 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;                      /* 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 = 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、チャネルごとに行うため、R_SCI_IIC_Open 関数は 1 回だけ実行する

図 6.6　1 つのチャネルで 2 つのスレーブデバイスにアクセスする例(2)

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

```
/* 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;
```

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

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

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

```
/* 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;
```

図 6.7　1 つのチャネルで 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;                 /* 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;          /* 次に“リトライ待ち”を行う */
        }
        else
        {
            retry_cnt = 0;
            sample_mode = DEVICE_B_READ;                /* 次に“デバイス B からの読み出し”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 = 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

図 6.8　1 つのチャネルで 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 6.9　1 つのチャネルで 2 つのスレーブデバイスにアクセスする例(5)

7.3　2 つのチャネルで 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 つのチャネルを簡易 I²C モードで使用し、各チャ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。

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

```
#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)
{
```

図 6.10　2 つのチャネルで 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_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;
                }

                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

異なるチャネルはそれぞれ同時に動作できるため
チャ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 6.11　2 つのチャネルで 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

図 6.12　2 つのチャネルで 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 6.13　2 つのチャネルで 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

図 6.14　2 つのチャネルで 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 6.15　2 つのチャネルで 2 つのスレーブデバイスにアクセスする例(6)

8. 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

改訂記録	RX ファミリ 簡易 I²C 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」 ・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
1.30	2014.09.22	—	FIT モジュールが RX64M グループに対応
		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 ・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」 ・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 ・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 ・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
1.40	2014.12.01	—	FIT モジュールが RX113 グループに対応

改訂記録	RX ファミリ 簡易 I²C 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	—	FIT モジュールが RX71M グループに対応
		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	「2.9 引数」に説明を追加
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
		ﾌﾟﾛｸﾞﾗﾑ	ソフトウェア不具合のため、簡易 I²C FIT モジュールを改修 ■内容 内蔵ボ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 I²C 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 周波数逓倍率(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 I²C FIT モジュール Rev1.60 以降をご使用ください。
		ﾌﾟﾛｸﾞﾗﾑ	ソフトウェア不具合のため、簡易 I²C FIT モジュールを改修 ■内容 ビットレートを低速に設定した場合、プログラムが無限ループする場合があります。 ■発生条件 次の 2 つの条件に該当したとき ・簡易 I²C FIT モジュール Rev.1.50 以前のバージョンをご使用されている ・sci_iic_set_frequency 関数で演算した BRR レジスタの値が 255 より大きくなる場合 (PCLKB に対してビットレートが極端に遅い場合) 例：PCLKB が 60MHz のときにビットレートを 200bps 以下にしたとき PCLKB が 300kHz のときにビットレートを 1bps にしたとき ■対策 簡易 I²C FIT モジュール Rev1.60 以降をご使用ください。

改訂記録	RX ファミリ 簡易 I²C モジュール Firmware Integration Technology

Rev.	発行日	改訂内容	
		ページ	ポイント
1.70	2015.05.29	—	FIT モジュールが RX231 グループに対応
1.80	2015.10.31	—	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
		31	「3.5 R_SCI_IIC_GetStatus()」「Format」を変更
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	「2.7 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 5.5 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 I²C 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	「2.7 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「2.8 コードサイズ」に変更。
2.20	2017.08.31	—	「対象デバイス」に RX24U グループを追加
		—	RX65N-2MB 版に対応
		—	RX130-512KB 版に対応
		—	RX24T-512KB 版に対応
		1	「関連ドキュメント」にスマート・コンフィグレータ ユーザーガイドを追加
		17-19	「2.4 使用する割り込みベクタ 」を追加
		20	「2.7 コンパイル時の設定」 SCI_IIC_CFG_CHi_INCLUDED の説明に使用上の注意事項を追加
			SCI_IIC_CFG_CHi_BITRATE_BPS の説明にビットレートについての注意事項を追加
		21	SCI_IIC_CFG_PORT_SETTING_PROCESSING についての 注意事項を追加
		25	「2.11 モジュールの追加方法」を変更
		42-43	「4. 端子設定 」を追加
		55-56	「5.3 動作確認環境」を追加
		57	「5.5. トラブルシューティング」を追加
		73	「7. 提供するモジュール」削除
		ﾌﾟﾛｸﾞﾗﾑ	SCI_IIC_CFG_CH1_INCLUDED の初期値を変更
		ﾌﾟﾛｸﾞﾗﾑ	RX63N、RX64M、RX65N、RX71M の r_sci_iic_io_open 関数の駆動能力制御設定処理を修正

改訂記録	RX ファミリ 簡易 I²C モジュール Firmware Integration Technology

Rev.	発行日	改訂内容	
		ページ	ポイント
2.30	2018.09.20	—	「対象デバイス」に RX66T グループを追加
		16	「2.3 サポートされているツールチェーン」 ツールチェーン v.3.00.00 を追加
		20	「2.4 使用する割り込みベクタ」 RX66T の使用する割り込みベクタを追加
		24	「2.7 コンパイル時の設定」 R_SCI_IIC_CFG_SCIi_SSCLi_PORT の設定範囲の上限を’J’→’K’に変更 R_SCI_IIC_CFG_SCIi_SSDAi_PORT の設定範囲の上限を’J’→’K’に変更
		25	「2.8 コードサイズ」コードサイズ(Rev2.30)を変更
		28	「2.12 for 文、while 文、do while 文について」を追加
		47-48	「5. デモプロジェクト」章を追加
		—	「5. 付録」→「6. 付録」に変更 全文で「6.付録」関連番号は 5→6 に変更
		61	「表 6.11 動作確認環境(Rev.2.30)」を追加
2.31	2018.12.03	61	6.3 動作確認環境詳細 表 6.11 動作確認環境 (Rev.2.30)の使用ボードの誤記を修正。
		62	6.3 動作確認環境詳細 表 6.12 動作確認環境 (Rev.2.31)を追加。
		ﾌﾟﾛｸﾞﾗﾑ	FIT モジュールのサンプルプログラムをダウンロードするためのアプリケーションノートのドキュメント番号を xml ファイルに追加。
2.40	2019.02.20	—	「対象デバイス」に RX72T グループを追加
		1	「関連ドキュメント」に以下のドキュメント名を変更 RX ファミリ ボードサポートパッケージモジュール Firmware Integration Technology(R01AN1685) RX ファミリ e² studio に組み込む方法 Firmware Integration Technology(R01AN1723) RX ファミリ CS+ に組み込む方法 Firmware Integration Technology(R01AN1826)
		16	「2.3 サポートされているツールチェーン」 ツールチェーン v.3.01.00 を追加
		20	「2.4 使用する割り込みベクタ」 RX72T の使用する割り込みベクタを追加
		62	「表 6.13 動作確認環境(Rev.2.40)」を追加
2.41	2019.05.20	—	以下のコンパイラに対応 ・GCC for Renesas RX ・IAR C/C++ Compiler for Renesas RX
		1	「関連ドキュメント」を削除
		1	「対象コンパイラ」を追加
		16	「2.2 ソフトウェアの要求」　依存する r_bsp モジュールのリビジョンを追加
		25	「2.8 コードサイズ」を更新
		44	3.7　R_SCI_IIC_GetVersion に関数のインライン展開を削除
		62	「表 6.14 動作確認環境(Rev.2.41)」を追加
		ﾌﾟﾛｸﾞﾗﾑ	RX63N の更新終了に伴い、RX63N に関する処理を削除 「対象デバイス」に、RX63N を削除

製品ご使用上の注意事項

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

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

ご注意書き

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

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

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

(Rev.4.0-1　2017.11)

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。