User's Manual

16

RL78 ファミリ

フラッシュ・セルフ・プログラミング・ライブラリ Type01

日本リリース版

インストーラ名：RENESAS_RL78_FSL_T01_xVxx

16 ビット・シングルチップ・マイクロコントローラ

ルネサス エレクトロニクス
www.renesas.com

Rev.1.10　2023.12

ご注意書き

1. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。回路、ソフトウェアおよびこれらに関連する情報を使用する場合、お客様の責任において、お客様の機器・システムを設計ください。これらの使用に起因して生じた損害（お客様または第三者いずれに生じた損害も含みます。以下同じです。）に関し、当社は、一切その責任を負いません。
2. 当社製品または本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害またはこれらに関する紛争について、当社は、何らの保証を行うものではなく、また責任を負うものではありません。
3. 当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
4. 当社製品を組み込んだ製品の輸出入、製造、販売、利用、配布その他の行為を行うにあたり、第三者保有の技術の利用に関するライセンスが必要となる場合、当該ライセンス取得の判断および取得はお客様の責任において行ってください。
5. 当社製品を、全部または一部を問わず、改造、改変、複製、リバースエンジニアリング、その他、不適切に使用しないでください。かかる改造、改変、複製、リバースエンジニアリング等により生じた損害に関し、当社は、一切その責任を負いません。
6. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、各品質水準は、以下に示す用途に製品が使用されることを意図しております。

   標準水準：　コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット等

   高品質水準：輸送機器（自動車、電車、船舶等）、交通制御（信号）、大規模通信機器、金融端末基幹システム、各種安全制御装置等

   当社製品は、データシート等により高信頼性、Harsh environment 向け製品と定義しているものを除き、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（宇宙機器と、海底中継器、原子力制御システム、航空機制御システム、プラント基幹システム、軍事機器等）に使用されることを意図しておらず、これらの用途に使用することは想定していません。たとえ、当社が想定していない用途に当社製品を使用したことにより損害が生じても、当社は一切その責任を負いません。
7. あらゆる半導体製品は、外部攻撃からの安全性を 100%保証されているわけではありません。当社ハードウェア／ソフトウェア製品にはセキュリティ対策が組み込まれているものもありますが、これによって、当社は、セキュリティ脆弱性または侵害（当社製品または当社製品が使用されているシステムに対する不正アクセス・不正使用を含みますが、これに限りません。）から生じる責任を負うものではありません。当社は、当社製品または当社製品が使用されたあらゆるシステムが、不正な改変、攻撃、ウイルス、干渉、ハッキング、データの破壊または窃盗その他の不正な侵入行為（「脆弱性問題」といいます。）によって影響を受けないことを保証しません。当社は、脆弱性問題に起因しまたはこれに関連して生じた損害について、一切責任を負いません。また、法令において認められる限りにおいて、本資料および当社ハードウェア／ソフトウェア製品について、商品性および特定目的との合致に関する保証ならびに第三者の権利を侵害しないことの保証を含め、明示または黙示のいかなる保証も行いません。
8. 当社製品をご使用の際は、最新の製品情報（データシート、ユーザーズマニュアル、アプリケーションノート、信頼性ハンドブックに記載の「半導体デバイスの使用上の一般的な注意事項」等）をご確認の上、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他指定条件の範囲内でご使用ください。指定条件の範囲を超えて当社製品をご使用された場合の故障、誤動作の不具合および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めていますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は、データシート等において高信頼性、Harsh environment 向け製品と定義しているものを除き、耐放射線設計を行っておりません。仮に当社製品の故障または誤動作が生じた場合であっても、人身事故、火災事故その他社会的損害等を生じさせないよう、お客様の責任において、冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、お客様の機器・システムとしての出荷保証を行ってください。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
10. 当社製品の環境適合性等の詳細につきましては、製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。かかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器・システムに使用することはできません。当社製品および技術を輸出、販売または移転等する場合は、「外国為替及び外国貿易法」その他日本国および適用される外国の輸出管理関連法規を遵守し、それらの定めるところに従い必要な手続きを行ってください。
12. お客様が当社製品を第三者に転売等される場合には、事前に当該第三者に対して、本ご注意書き記載の諸条件を通知する責任を負うものといたします。
13. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを禁じます。
14. 本資料に記載されている内容または当社製品についてご不明な点がございましたら、当社の営業担当者までお問合せください。

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

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

(Rev.5.0-1　2020.10)

本社所在地

〒135-0061 東京都江東区豊洲 3-2-24（豊洲フォレシア）

www.renesas.com

お問合せ窓口

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

www.renesas.com/contact/

商標について

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

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。

このマニュアルの使い方

対 象 者　このユーザーズマニュアルは、RL78マイクロコントローラのフラッシュ・セルフ・プログラミング・ライブラリ　Type01の機能を理解し、それを用いたアプリケーション・システムを設計するユーザを対象としています。

対応MCU：以下のリストを参照してください。

日本語版：

マイコン対応セルフプログラミングライブラリ(日本リリース版)一覧(R20UT2741)、及び

RL78ファミリ セルフプログラミングライブラリ セルフRAMリスト(R20UT2943)

目　　的　このユーザーズマニュアルは、RL78マイクロコントローラのコード・フラッシュ・メモリの書き換えを行うために使用するフラッシュ・セルフ・プログラミング・ライブラリ　Type01の使用方法を理解していただくことを目的としています。

構　　成　このマニュアルは、大きく分けて次の内容で構成しています。

・概要説明
・プログラミング環境
・フラッシュ・セルフ・プログラミング実行中の割り込み
・セキュリティ設定
・ブート・スワップ機能
・フラッシュ関数

読 み 方　このマニュアルを読むにあたっては、電気、論理回路、マイクロコントローラの一般知識を必要とします。

□一通りの機能を理解しようとするとき

→目次に従って読んでください。

□関数の機能の詳細を知りたいとき

→このユーザーズマニュアルの**第6章　フラッシュ関数**を参照してください。

凡　　例

データ表記の重み	：左が上位桁、右が下位桁
アクティブ・ロウの表記	：$\overline{\times\times\times}$（端子、信号名称に上線）
注	：本文中につけた注の説明
注意	：気をつけて読んでいただきたい内容
備考	：本文の補足説明
数の表記	：2進数…××××または××××B 10進数…×××× 16進数…××××Hまたは 0x××××

目次

RL78 ファミリ
フラッシュ・セルフ・プログラミング・ライブラリ Type01

R01US0050JJ0110
Rev.1.10
2023.12.26

第1章　概　説

1.1　概　要

フラッシュ・セルフ・プログラミング・ライブラリは、RL78マイクロコントローラに搭載されたファームウエアを使用し、コード・フラッシュ・メモリ内のデータを書き換えるためのソフトウェアです。

フラッシュ・セルフ・プログラミング・ライブラリをユーザ・プログラムから呼び出すことにより、コード・フラッシュ・メモリの内容を書き換えることができるため、ソフトウェアの開発期間を大幅に短縮することが可能となります。

なお、本フラッシュ・セルフ・プログラミング・ライブラリ・ユーザーズマニュアルは、対象デバイスのマニュアルと合わせてご使用ください。

用　　語　このマニュアルで使用する用語について、その意味を次に示します。

・フラッシュ・セルフ・プログラミング
　ユーザ・プログラム自身によるコード・フラッシュ・メモリへの書き込み動作です。
・フラッシュ・セルフ・プログラミング・ライブラリ
　RL78マイクロコントローラが提供する機能によるコード・フラッシュ・メモリ操作のためのライブラリです。
　データ・フラッシュ・メモリへの操作は出来ません。
・フラッシュ環境
　コード・フラッシュ・メモリの操作が可能である状態です。通常のプログラムの実行とは異なる制限事項があります。また、データ・フラッシュ・メモリへの操作は出来なくなります。
・ブロック番号
　フラッシュ・メモリのブロックを示す番号です。消去、ブランク・チェック、ベリファイ（内部ベリファイ）の操作単位です。
・ブート・クラスタ
　ブート・スワップに使用するブート領域です。ブート・スワップ機能の有無に関しては対象となるRL78マイクロコントローラのユーザーズマニュアルを参照してください。
・内部ベリファイ
　フラッシュ・メモリへの書き込み後、フラッシュ・メモリのセルの信号レベルが適正であるかを確認することです。
　内部ベリファイでエラーとなった場合、そのデバイスは故障していると判断されます。ただし、内部ベリファイ・エラー後に、再度、データの消去→書き込み→内部ベリファイを実行し、正常終了した場合には、そのデバイスは正常であると判断します。
・FSL
　フラッシュ・セルフ・プログラミング・ライブラリの略称です。

・FSW

フラッシュ・シールド・ウインドウの略称です。

・フラッシュ関数

フラッシュ・セルフ・プログラミング・ライブラリを構成する関数のことです。

・シーケンサ

RL78マイクロコントローラにはフラッシュ・メモリ制御用の専用回路が搭載されています。本書ではこの回路のことをシーケンサと呼びます。

・BGO (バック・グラウンド・オペレーション)

シーケンサにフラッシュ・メモリの制御を任せる事によって、ユーザ・プログラムを動作させながら、フラッシュ・メモリの書き換えを実行できる状態のことです。

・ステータス・チェック

シーケンサを使用する場合、フラッシュ・メモリの制御を行うプログラムでシーケンサの状態(フラッシュ・メモリに対する制御の状態)を確認する処理が必要になります。このシーケンサの状態を確認する処理を本書ではステータス・チェックと呼びます。

・ROM化(プログラム)

RL78マイクロコントローラのフラッシュ・セルフ・プログラミングでは、制御方法によってはユーザ・プログラムやフラッシュ・セルフ・プログラミング・ライブラリをRAMに配置して処理を実行する必要があります。本書ではRAM上で動作させるために作成したプログラムを、コード・フラッシュ・メモリ等に配置して使用可能にする事をROM化と呼びます。

ROM化を行う場合は、開発ツール等の機能を使用する必要があります。

・EEPROMエミュレーション・ライブラリ

搭載されているフラッシュ・メモリへEEPROMのようにデータを格納させるための機能を提供するソフトウェア・ライブラリです。

・データ・フラッシュ・ライブラリ

データ・フラッシュ・メモリへの操作を行うためのソフトウェア・ライブラリです。

1.2　フラッシュ・セルフ・プログラミング・ライブラリの呼び出し方法

フラッシュ・セルフ・プログラミングを行うためにはフラッシュ・セルフ・プログラミングの初期化処理や、使用する機能に対応する関数をC言語、ならびにアセンブリ言語のどちらかでユーザ・プログラムから実行する必要があります。

また、フラッシュ・セルフ・プログラミング・ライブラリ Type01では、コード・フラッシュ・メモリを書き換える操作を行う場合、書き換え実行中はコード・フラッシュ・メモリが参照できなくなるため、使用方法によってはフラッシュ・セルフ・プログラミング・ライブラリの一部のセグメントやユーザ・プログラムをRAM上に配置する必要があります。

図1−1にフラッシュ・セルフ・プログラミングの状態遷移図を、図1−2にフラッシュ・セルフ・プログラミング・ライブラリを利用したコード・フラッシュ・メモリ書き換えフロー例を、図1−3にBGO（バック・グラウンド・オペレーション）時のコード・フラッシュ・メモリの書き換えフロー例を示します。

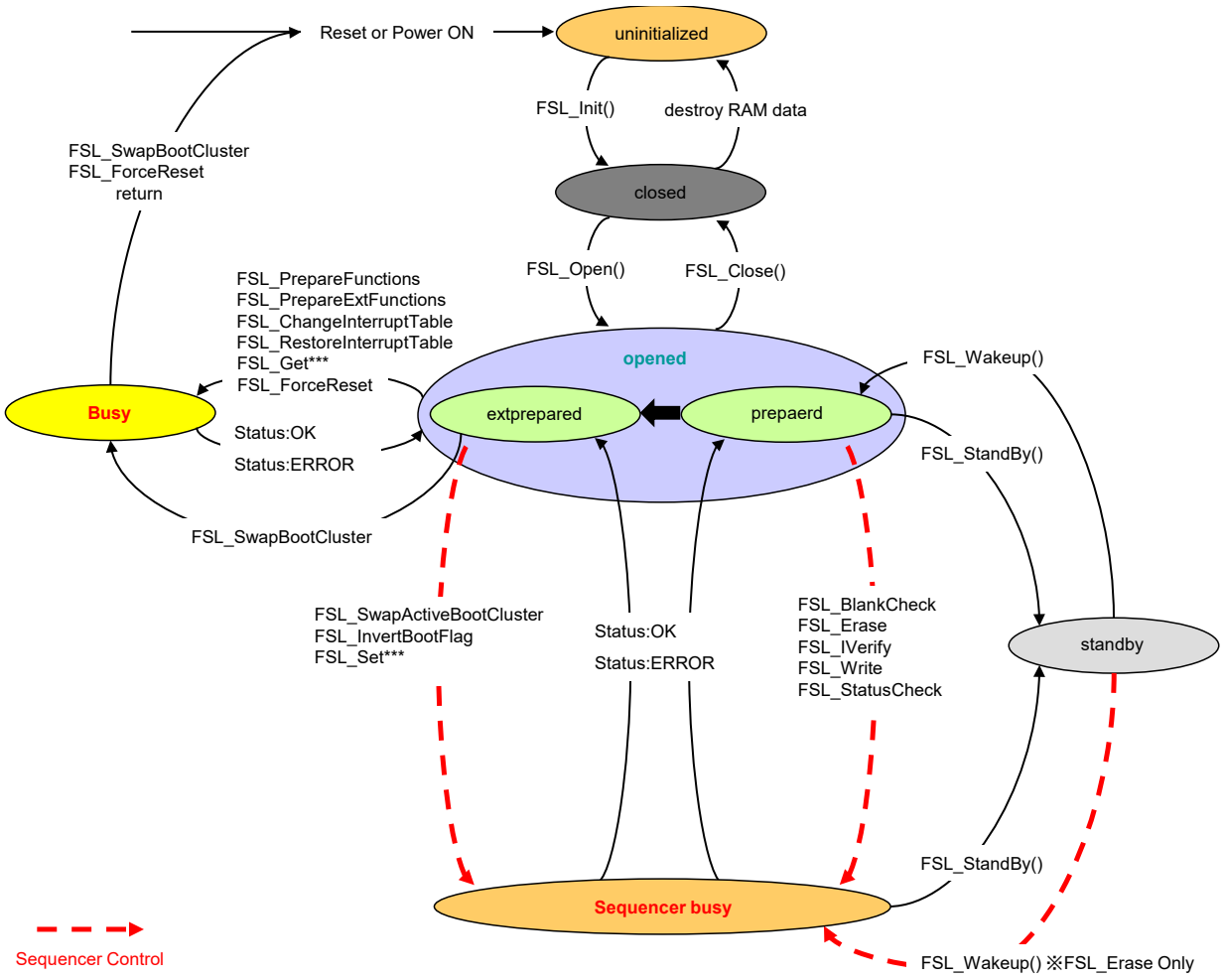


図 1−1　フラッシュ・セルフ・プログラミングの状態遷移図

【状態遷移図の概要】

フラッシュ・セルフ・プログラミング・ライブラリを使用してコード・フラッシュ・メモリを操作するためには、用意されている関数を順に実行し、処理を進める必要があります。

(1) uninitialized

Power ON、Reset時の状態です。また、EEPROMエミュレーション・ライブラリやデータ・フラッシュ・ライブラリを実行した場合もこの状態に遷移します。

(2) closed

FSL_Init関数を実行し、フラッシュ・セルフ・プログラミングを実行するためのデータを初期化した状態(コード・フラッシュ・メモリへの操作は停止状態)です。フラッシュ・セルフ・プログラミングを動作させた後にEEPROMエミュレーション・ライブラリやデータ・フラッシュ・ライブラリ、STOPモード、HALTモードを実行する場合は、opened状態からFSL_Closeを実行し、この状態に遷移させてください。

(3) opened

closed状態からFSL_Open関数を実行し、フラッシュ・セルフ・プログラミングが実行可能になった状態です。この状態をフラッシュ環境と呼びます。FSL_Closeを実行し、closed状態に遷移するまでの間はEEPROMエミュレーション・ライブラリやデータ・フラッシュ・ライブラリ、STOPモード、HALTモードは実行できません。

(4) prepared

opened状態からFSL_PrepareFunctions関数を実行し、書き込みや消去等のコード・フラッシュ・メモリへの操作が可能になった状態です。

(5) extprepared

opened状態からFSL_PrepareFunctions関数とFSL_PrepareExtFunctions関数を順番に実行し、セキュリティ・フラグの書き換えやブート・スワップ処理が実行可能になった状態です。

(6) busy

指定された処理を実行している状態です。実行関数と終了状況によっては遷移する状態が変わる場合もあります。

(7) sequencer busy

シーケンサを使用して指定された処理を実行している状態です。シーケンサの使用中はコード・フラッシュ・メモリを参照できなくなります。また、実行関数と終了状況によっては遷移する状態が変わる場合もあります。

(8) standby

FSL_StandBy関数によってフラッシュ・セルフ・プログラミングを一時停止している状態です。FSL_WakeUp関数によってフラッシュ・セルフ・プログラミングを再開します。FSL_Erase関数実行中に一時停止させた場合は、FSL_Erase関数の処理を再開します。

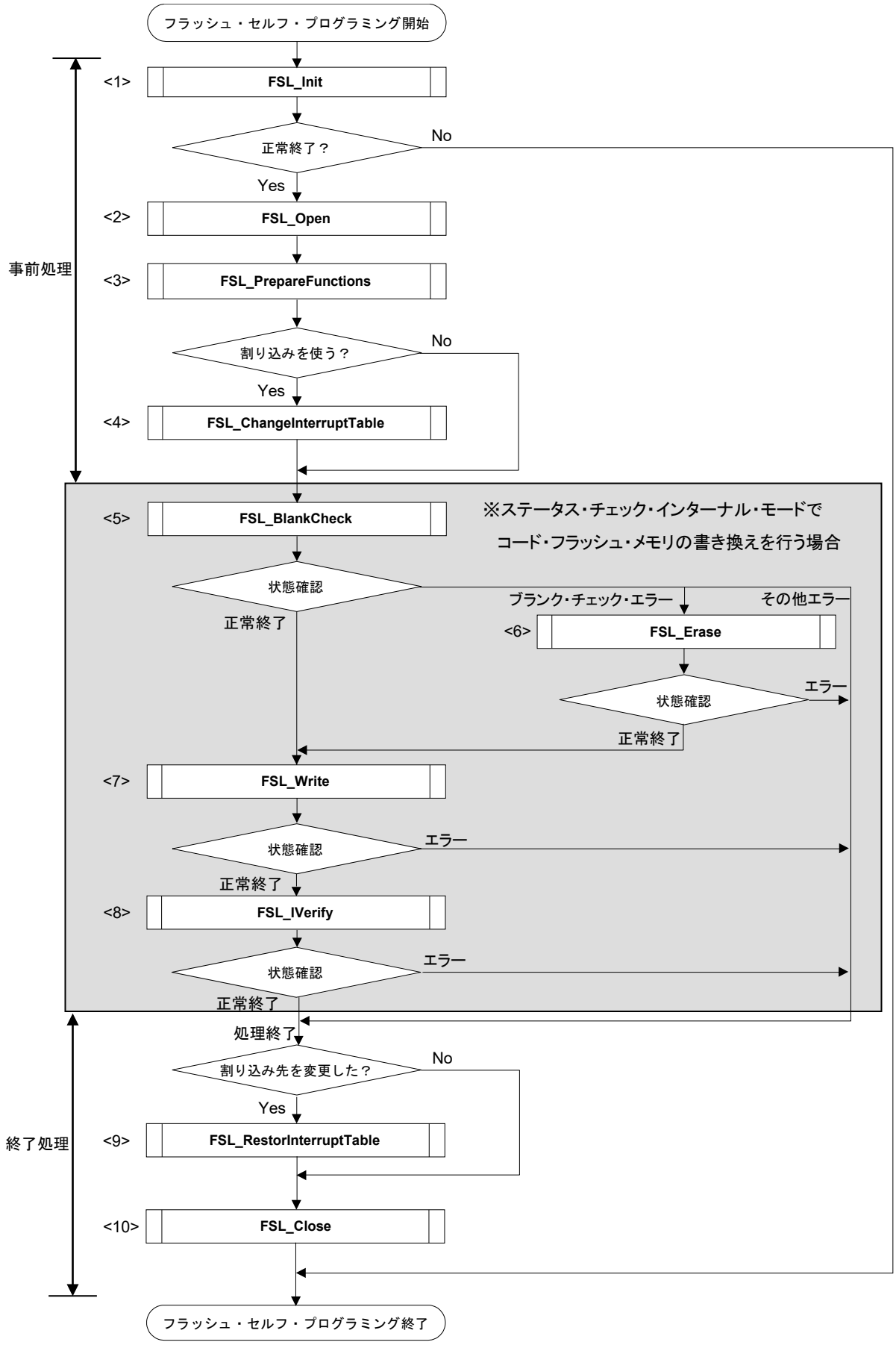


図 1−2　フラッシュ・セルフ・プログラミング（コード・フラッシュ・メモリの書き換え）のフロー例

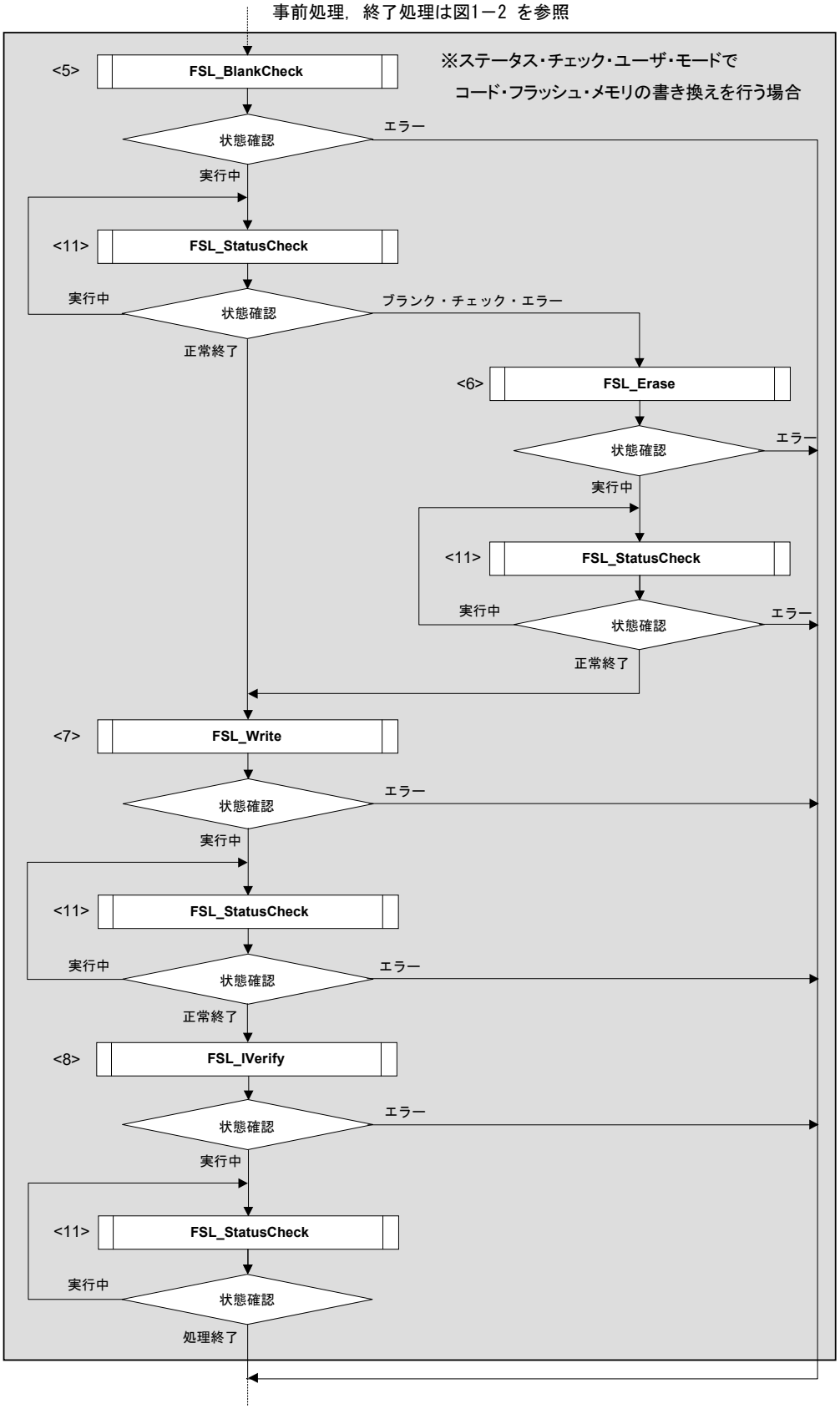


図 1－3　バック・グラウンド・オペレーション時のコード・フラッシュ・メモリ書き換えフロー例

<1>　FSL_Init：フラッシュ・セルフ・プログラミングで使用するRAMの初期化

FSL_Init関数を呼び出し、フラッシュ・セルフ・プログラミングで使用するRAMを初期化し、動作に必要なパラメータを設定します。

<2>　FSL_Open：フラッシュ環境の開始

FSL_Open関数を呼び出し、フラッシュ・セルフ・プログラミングを使用可能にします。

<3>　FSL_PrepareFunctions：準備処理

FSL_PrepareFunctions関数を呼び出し、フラッシュ・セルフ・プログラミングで使用する関数の準備を行います。
拡張関数を使用する場合は、FSL_PrepareExtFunctions関数も呼び出す必要があります。
FSL_PrepareFunctions関数、FSL_PrepareExtFunctions関数の詳細については、第6章 フラッシュ関数を参照してください。

<4>　FSL_ChangeInterruptTable：割り込みの受付をRAMへ変更

フラッシュ・セルフ・プログラミング実行中に割り込みが必要な場合は、FSL_ChangeInterruptTable関数を呼び出し、割り込み先をROMからRAMへ変更します。

<5>　FSL_BlankCheck：指定ブロック（1 Kバイト）のブランク・チェック

FSL_BlankCheck関数を呼び出し、指定ブロック（1 Kバイト）のブランク・チェック（書き込み可能な領域であることの確認）を行います。

<6>　FSL_Erase：指定ブロック（1 Kバイト）の消去

FSL_Erase関数を呼び出し、指定ブロック（1 Kバイト）の消去を行います。

<7>　FSL_Write：指定アドレスに対する1-64ワード（4-256バイト）のデータ書き込み

FSL_Write関数を呼び出し、指定アドレスに対する1-64ワード（4-256バイト）データの書き込みを行います。
指定ブロックの書き込みが一度で終了しない場合は、FSL_Write関数を複数回実行し、指定ブロックの書き込みを全て完了させ、次の処理に遷移します。また、書き込みはブランク状態か、もしくは消去済みの領域へのみ行えます。既に書き込みを行っている領域へ再度書き込みを行う事(上書き)はできません。

<8>　FSL_IVerify：指定ブロック（1 Kバイト）の内部ベリファイ[注]

FSL_IVerify関数を呼び出し、指定ブロック（1 Kバイト）に内部ベリファイを行います。

注. 内部ベリファイとは、フラッシュ・メモリのセルの信号レベルが適正であるかを確認することです。
データの比較による確認は行いません。

<9>　FSL_RestoreInterruptTable：割り込みの受付をROMへ戻す

<4>で割り込み先をRAMへ変更している場合はFSL_RestoreInterruptTable関数を呼び出し、割り込みの受付先をROMに戻します。

<10> FSL_Close：フラッシュ環境の終了

FSL_Close関数を呼び出し、フラッシュ・セルフ・プログラミングを終了します。FSL_Close関数は書き込み処理が全て終了するか、フラッシュ・セルフ・プログラミングを終了させる必要がある場合に実行します。

<11> FSL_StatusCheck：ステータス・チェック

ステータス・チェック・ユーザ・モードを使用する場合は、コード・フラッシュ・メモリの制御が終了するまで、ステータス・チェックを行う必要があります。

備考 1ワード = 4バイト

第2章　プログラミング環境

この章では、ユーザがフラッシュ・セルフ・プログラミング・ライブラリを使用してコード・フラッシュ・メモリの書き換えを行ううえで、必要なハードウェア環境とソフトウェア環境について説明します。

2.1　ハードウェア環境

RL78 マイクロコントローラのフラッシュ・セルフ・プログラミングはシーケンサを使用し、フラッシュの書き換え制御を実行します。また、シーケンサの制御中はコード・フラッシュ・メモリを参照できなくなります。そのため、割り込み注等、シーケンサ制御中にユーザ・プログラムを動作させる必要がある場合、コード・フラッシュ・メモリの消去や書き込み、セキュリティ・フラグの設定等を行う時に、フラッシュ・セルフ・プログラミング・ライブラリの一部のセグメントや、ユーザ・プログラムを RAM に配置して制御を行う必要があります。シーケンサ制御中にユーザ・プログラムを動作させる必要が無い場合は、フラッシュ・セルフ・プログラミング・ライブラリやユーザ・プログラムを ROM 上に配置して動作させる事が可能です。

図 2−1 にコード・フラッシュ・メモリの書き換え中の状態、図 2−2、図 2−3 にコード・フラッシュ・メモリの書き換えを行う場合のフラッシュ関数の実行例を示します。

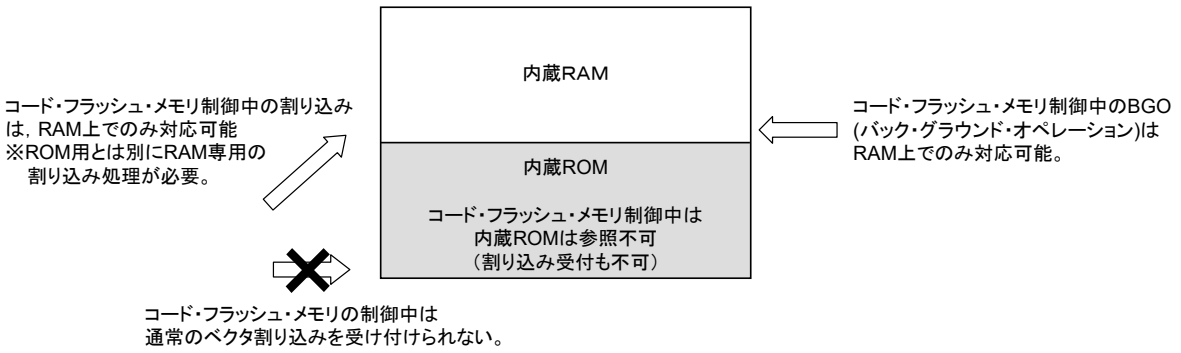


図 2−1　コード・フラッシュ・メモリ書き換え中の状態

注　一部の RL78 マイクロコントローラはフラッシュ・セルフ・プログラミング中の割り込みに対応していない場合があります。ご使用になる RL78 マイクロコントローラがフラッシュ・セルフ・プログラミング中の割り込みに対応しているかについては、対象となる RL78 マイクロコントローラのユーザーズマニュアルを参照してください。

・　RL78 マイクロコントローラのシーケンサに該当処理の実行要求を行ったのち、制御を直ちにユーザ・プログラムに戻します。コード・フラッシュ・メモリへの制御はシーケンサが行うため、コード・フラッシュ・メモリの書き換え中にもユーザ・プログラムを動作させることが可能です。この事を BGO(バック・グラウンド・オペレーション)と呼びます。このモードを使用する場合は、フラッシュ・セルフ・プログラミング・ライブラリ Type01 の初期化を実行する際に、ステータス・チェック・ユーザ・モードを選択します。

　ただし、シーケンサがコード・フラッシュ・メモリへの制御を行っている間は、コード・フラッシュ・メモリは参照できなくなります。そのため、コード・フラッシュ・メモリ操作中に動作するユーザ・プログラムや割り込みの分岐先と割り込み処理、フラッシュ・セルフ・プログラミング・ライブラリの一部のセグメントを RAM に配置する必要があります。

　コード・フラッシュ・メモリへの制御の結果については、ユーザ・プログラムからステータス・チェック関数(FSL_StatusCheck 関数)を呼び出し、コード・フラッシュ・メモリの制御状態を確認する処理が必要となります。

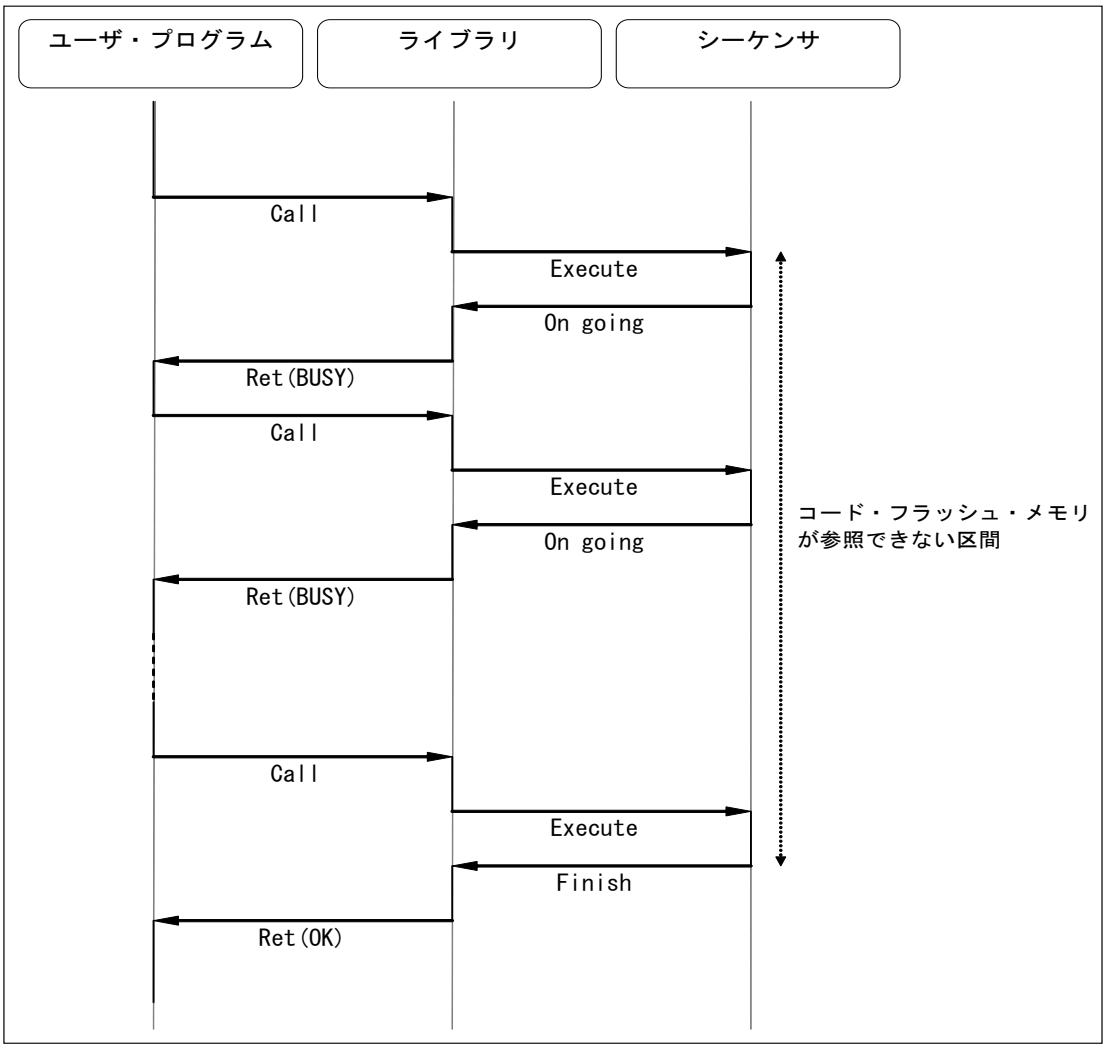


図 2−2　フラッシュの書き換え制御例 1 (書き換え中にユーザ･プログラムが動作する場合)

・　RL78 マイクロコントローラのシーケンサに該当処理の実行要求を行ったのち、シーケンサの該当処理が完了するまで制御をユーザ・プログラムに戻しません。コード・フラッシュ・メモリへの制御が完了した状態でユーザ・プログラムに戻るため、ユーザ・プログラムやフラッシュ・セルフ・プログラミングを ROM に配置することが可能となります。このモードを使用する場合は、フラッシュ・セルフ・プログラミング・ライブラリ Type01 の初期化を実行する際に、ステータス・チェック・インターナル・モードを選択します。

　ただし、コード・フラッシュ・メモリの制御中に割り込みを受け付ける必要がある場合は、割り込みの分岐先と割り込み処理を RAM に配置する必要があります。また、ROM に配置した場合は、一部のフラッシュ関数が使用できなくなります。フラッシュ関数の詳細については、第 6 章 フラッシュ関数を参照してください。

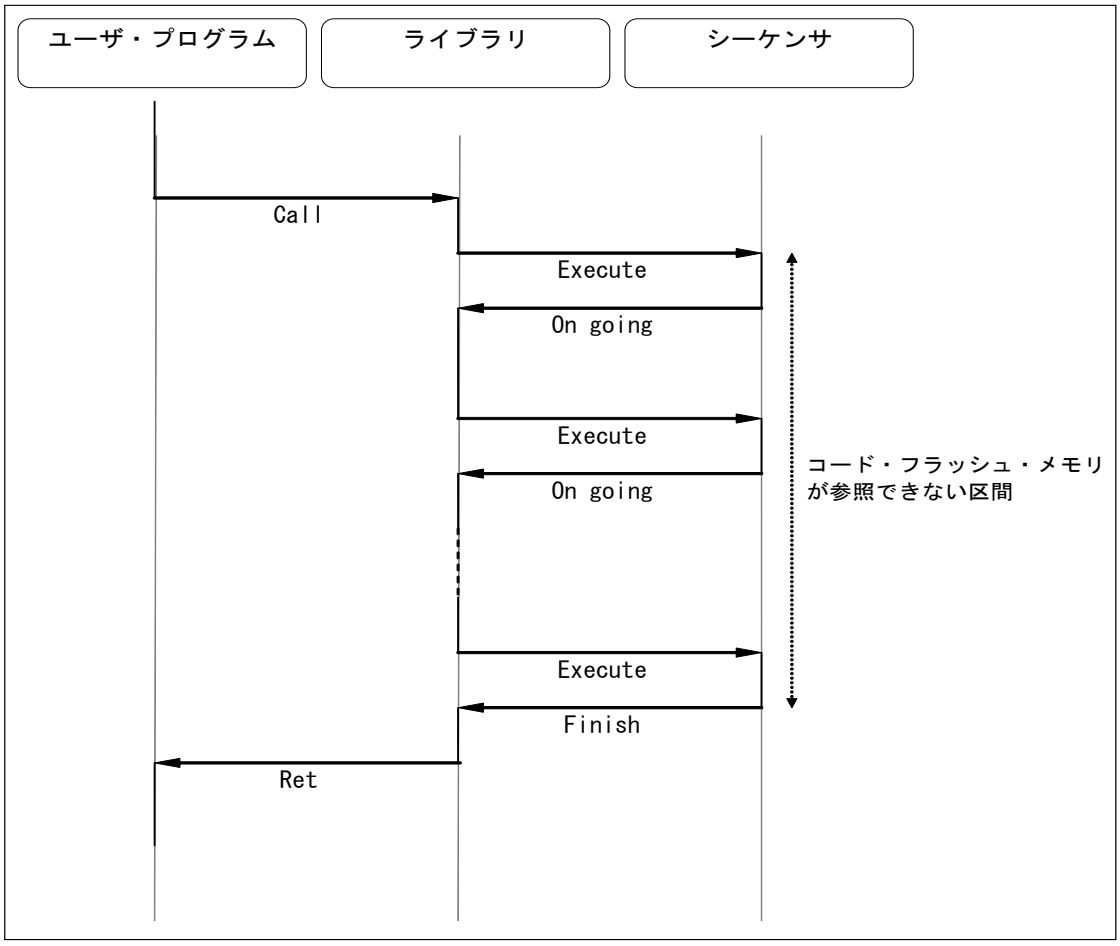


図 2−3　フラッシュの書き換え制御例 2 (書き換え中にユーザ・プログラムが動作しない場合)

2. 1. 1　初期設定

フラッシュ・セルフ・プログラミング・ライブラリを使用してコード・フラッシュ・メモリの書き換えを行う場合、以下の設定を行う必要があります。

（1）　高速オンチップ・オシレータの起動

フラッシュ・セルフ・プログラミング・ライブラリの使用中は高速オンチップ・オシレータを動作させておく必要があります。高速オンチップ・オシレータを停止させている場合は、使用前に起動させておいてください。

（2）　CPU の動作周波数[注 1]の設定

フラッシュ・セルフ・プログラミング・ライブラリ 内部のタイミング計算を行うため、初期化時に CPU の動作周波数を設定する必要があります。CPU の動作周波数の設定方法には関しては、FSL_Init()関数の説明を参照してください。

（3）フラッシュ書き換えモード[注 2]の設定

消去、および書き込み時のフラッシュ書き換えモードの設定を行うため、フラッシュ・セルフ・プログラミング・ライブラリの初期化時に、以下のフラッシュ書き換えモードを設定する必要があります。フラッシュ書き換えモードの設定方法に関しては FSL_Init()関数の説明を参照してください。

- フルスピード・モード
- ワイド・ボルテージ・モード

注　1. CPU の動作周波数はフラッシュ・セルフ・プログラミング・ライブラリ 内部のタイミング計算用のパラメータとして使用されます。本設定によって CPU の動作周波数が変わることはありません。また、高速オンチップ・オシレータの動作周波数ではありません。
2. フラッシュ書き換えモードの詳細については、対象となる RL78 マイクロコントローラのユーザーズマニュアルを参照してください。

2. 1. 2　ブロック

RL78 マイクロコントローラは、フラッシュ・メモリが 1 K バイト単位でブロック分割されています。フラッシュ・セルフ・プログラミングでは、このブロックを単位としてコード・フラッシュ・メモリに対し、消去処理、ブランク・チェック処理、ベリファイ（内部ベリファイ）処理を行います。これらのフラッシュ・セルフ・プログラミング・ライブラリを呼び出す際には、ブロック番号を指定します。

なお、ブート・クラスタ[注]は、ベクタ領域を含むエリアを書き換える際、電源の瞬断、書き換え中のリセットなどにより、ベクタ・テーブル・データ、プログラムの基本機能などが破壊され、ユーザ・プログラムが立ち上がらなくなることを防止するための領域です。詳細については、第 5 章　ブート・スワップ機能[注]を参照してください。

図 2−4 に、ブロック番号とブート・クラスタを示します。

注　本機能を使用するためには、ブート・スワップ機能に対応した RL78 マイクロコントローラが必要になります。ご使用になる RL78 マイクロコントローラがブート・スワップ機能に対応しているかは、対象となる RL78 マイクロコントローラのユーザーズマニュアルを参照してください。

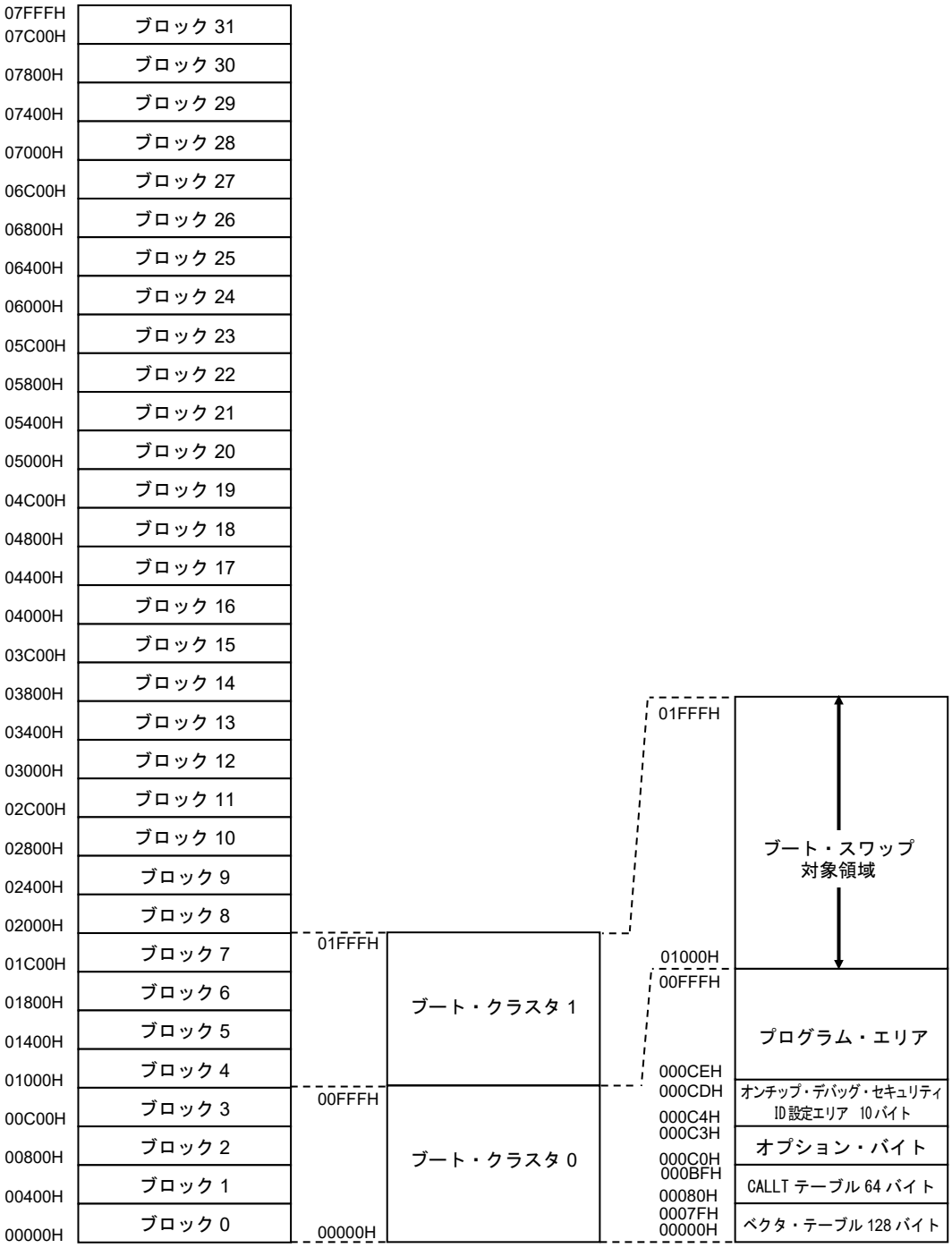


図 2−4　ブロック番号とブート・クラスタ例（RL78/G13:コード・フラッシュ・メモリが 32 K バイトの場合）

2. 1. 3　処理時間

この節ではフラッシュ・セルフ・プログラミング・ライブラリ Type01 の処理時間について記載します。フラッシュ関数は内部 ROM 領域（フラッシュ・メモリ）に配置した場合と内部 RAM 領域に配置した場合では、実行クロックが異なります。処理時間は ROM から実行した場合に対し、RAM から実行した場合、最大 2 倍程度になる場合があります。

ここで記載している関数の処理時間は、FSL_RCD セグメントを RAM で実行し、その他のセグメントを ROM で実行した場合の処理時間になります。各フラッシュ関数のセグメントについては、表 6-2 フラッシュ関数のセグメント一覧をご参照ください。

(1) フラッシュ関数処理時間（ステータス・チェック・ユーザ・モード）

フラッシュ関数がユーザ・プログラムから実行された後、処理を終了してユーザ・プログラムに戻るまでの時間です。ステータス・チェックのモード設定によってフラッシュ関数処理時間は異なります。本項ではステータス・チェック・ユーザ・モードにおけるフラッシュ関数処理時間を記載します。

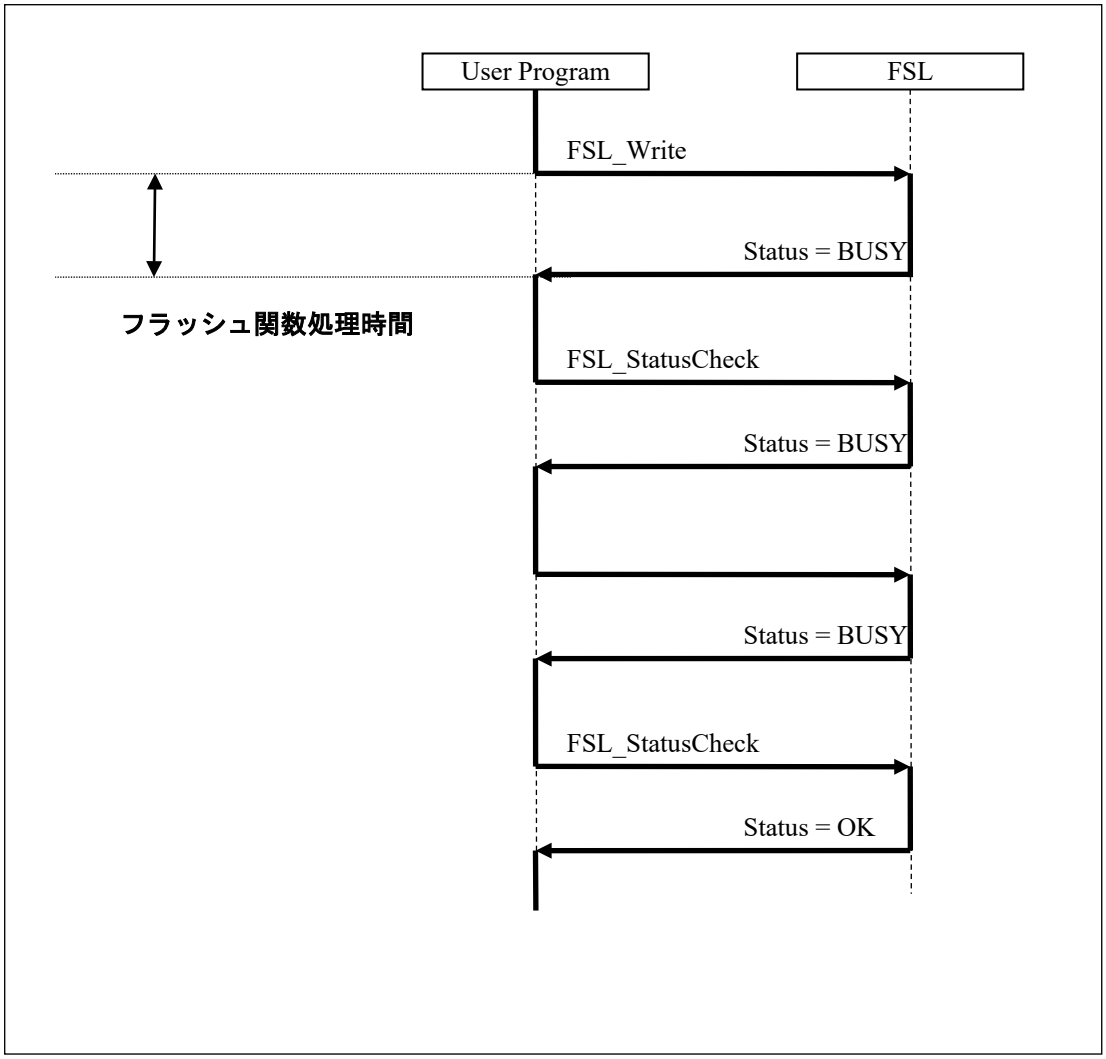


図 2−5　ステータス・チェック・ユーザ・モードにおけるフラッシュ関数処理時間の概念図

表 2−1 ステータス・チェック・ユーザ・モードのフラッシュ関数処理時間
（フルスピード・モード）

FSL_Functions		Max (μs)
FSL_Init		5021 / fCLK
FSL_Open		10 / fCLK
FSL_Close		10 / fCLK
FSL_PrepareFunctions		2484 / fCLK
FSL_PrepareExtFunctions		1259 / fCLK
FSL_ChangeInterruptTable		253 / fCLK
FSL_RestoreInterruptTable		229 / fCLK
FSL_BlankCheck		2069 / fCLK + 30
FSL_Erase		2192 / fCLK + 30
FSL_IVerify		2097 / fCLK + 30
FSL_Write		2451 / fCLK + 30
FSL_GetSecurityFlags		331 / fCLK
FSL_GetBootFlag		328 / fCLK
FSL_GetSwapState		206 / fCLK
FSL_GetBlockEndAddr		368 / fCLK
FSL_GetFlashShieldWindow		307 / fCLK
FSL_SwapBootCluster		419 / fCLK + 32
FSL_SwapActiveBootCluster		2316 / fCLK + 30
FSL_InvertBootFlag		2341 / fCLK + 30
FSL_SetBlockEraseProtectFlag		2347 / fCLK + 30
FSL_SetWriteProtectFlag		2346 / fCLK + 30
FSL_SetBootClusterProtectFlag		2347 / fCLK + 30
FSL_SetFlashShieldWindow		2141 / fCLK + 30
FSL_StatusCheck		1135 / fCLK + 50
FSL_StandBy	Erase	935 / fCLK + 31
	except Erase (in case of FSL_SetXXX are suported)	140367 / fCLK + 513844
	except Erase (in case of FSL_SetXXX are not suported)	76101 / fCLK + 35952
FSL_WakeUp	Suspended Erase	2144 / fCLK + 30
	except Erase	148 / fCLK
FSL_ForceReset		−
FSL_GetVersionString		10 / fCLK

備考. fCLK：CPU の動作周波数(例：20MHz 時の fCLK = 20)

表 2−2 ステータス・チェック・ユーザ・モードのフラッシュ関数処理時間
（ワイド・ボルテージ・モード）

FSL_Functions		Max (μs)
FSL_Init		5021 / f_{CLK}
FSL_Open		10 / f_{CLK}
FSL_Close		10 / f_{CLK}
FSL_PrepareFunctions		2484 / f_{CLK}
FSL_PrepareExtFunctions		1259 / f_{CLK}
FSL_ChangeInterruptTable		253 / f_{CLK}
FSL_RestoreInterruptTable		229 / f_{CLK}
FSL_BlankCheck		2068 / f_{CLK} + 30
FSL_Erase		2192 / f_{CLK} + 30
FSL_IVerify		2097 / f_{CLK} + 30
FSL_Write		2451 / f_{CLK} + 30
FSL_GetSecurityFlags		331 / f_{CLK}
FSL_GetBootFlag		328 / f_{CLK}
FSL_GetSwapState		206 / f_{CLK}
FSL_GetBlockEndAddr		368 / f_{CLK}
FSL_GetFlashShieldWindow		307 / f_{CLK}
FSL_SwapBootCluster		419 / f_{CLK} + 32
FSL_SwapActiveBootCluster		2316 / f_{CLK} + 30
FSL_InvertBootFlag		2341 / f_{CLK} + 30
FSL_SetBlockEraseProtectFlag		2347 / f_{CLK} + 30
FSL_SetWriteProtectFlag		2346 / f_{CLK} + 30
FSL_SetBootClusterProtectFlag		2347 / f_{CLK} + 30
FSL_SetFlashShieldWindow		2141 / f_{CLK} + 30
FSL_StatusCheck		1135 / f_{CLK} + 50
FSL_StandBy	Erase	935 / f_{CLK} + 44
	except Erase (in case of FSL_SetXXX are suported)	123274 / f_{CLK} + 538046
	except Erase (in case of FSL_SetXXX are not suported)	73221 / f_{CLK} + 69488
FSL_WakeUp	Suspended Erase	2144 / f_{CLK}+ 30
	except Erase	148 / f_{CLK}
FSL_ForceReset		−
FSL_GetVersionString		10 / f_{CLK}

備考. f_{CLK}：CPU の動作周波数(例：20MHz 時の f_{CLK} = 20)

（2）フラッシュ関数処理時間（ステータス・チェック・インターナル・モード）

本項ではステータス・チェック・インターナル・モードのフラッシュ関数処理時間を記載します。

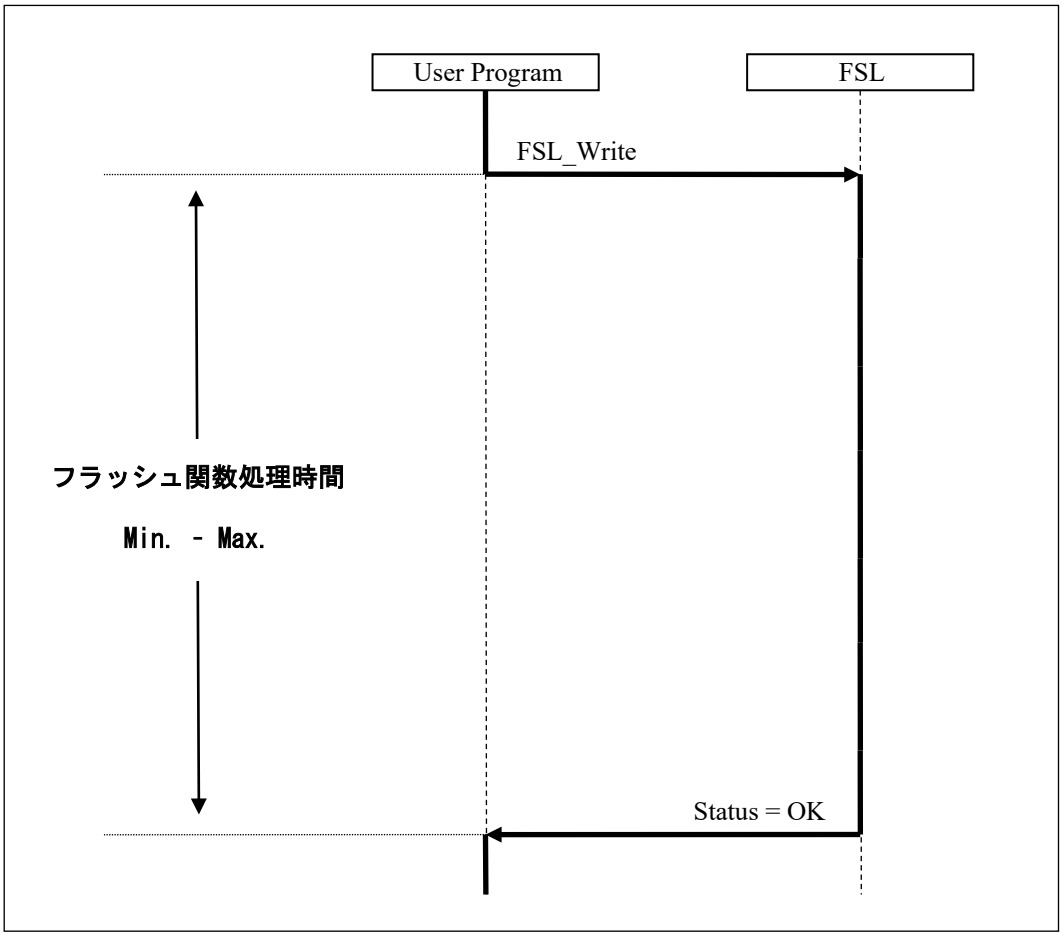


図 2−6　ステータス・チェック・インターナル・モードにおけるフラッシュ関数処理時間の概念図

表 2−3 ステータス・チェック・インターナル・モードのフラッシュ関数処理時間
（フルスピード・モード）

FSL_Functions	Min (μs)	Max (μs)
FSL_Init	−	5021 / fCLK
FSL_Open	−	10 / fCLK
FSL_Close	−	10 / fCLK
FSL_PrepareFunctions	−	2484 / fCLK
FSL_PrepareExtFunctions	−	1259 / fCLK
FSL_ChangeInterruptTable	−	253 / fCLK
FSL_RestoreInterruptTable	−	229 / fCLK
FSL_BlankCheck	3302 / fCLK + 84	4833 / fCLK + 164
FSL_Erase	4877 / fCLK + 163	73339 / fCLK + 255366
FSL_IVerify	−	10474 / fCLK + 1107
FSL_Write	3121 / fCLK + 66 + (595 / fCLK + 60) × W	3121 / fCLK + 66 + (1153 / fCLK + 561) × W
FSL_GetSecurityFlags	−	331 / fCLK
FSL_GetBootFlag	−	328 / fCLK
FSL_GetSwapState	−	206 / fCLK
FSL_GetBlockEndAddr	−	368 / fCLK
FSL_GetFlashShieldWindow	−	307 / fCLK
FSL_SwapBootCluster	−	419 / fCLK + 32
FSL_SwapActiveBootCluster	1938 / fCLK + 50	141314 / fCLK + 513862
FSL_InvertBootFlag	1565 / fCLK + 18	140940 / fCLK + 513830
FSL_SetBlockEraseProtectFlag	1571 / fCLK + 18	140946 / fCLK + 513830
FSL_SetWriteProtectFlag	1569 / fCLK + 18	140945 / fCLK + 513830
FSL_SetBootClusterProtectFlag	1571 / fCLK + 18	140946 / fCLK + 513830
FSL_SetFlashShieldWindow	1356 / fCLK + 18	140739 / fCLK + 513830
FSL_StatusCheck	−	−
FSL_StandBy	−	−
FSL_WakeUp	−	−
FSL_ForceReset	−	−
FSL_GetVersionString	−	10 / fCLK

備考 1.　fCLK：CPU の動作周波数(例：20MHz 時の fCLK = 20)

2.　W　：1 ワード＝4 バイト単位の書き込みデータ長（ 例：2 ワード/8 バイト指定時 W = 2 ）

表 2―4 ステータス・チェック・インターナル・モードのフラッシュ関数処理時間
（ワイド・ボルテージ・モード）

FSL_Functions	Min (μs)	Max (μs)
FSL_Init	―	5021 / fCLK
FSL_Open	―	10 / fCLK
FSL_Close	―	10 / fCLK
FSL_PrepareFunctions	―	2484 / fCLK
FSL_PrepareExtFunctions	―	1259 / fCLK
FSL_ChangeInterruptTable	―	253 / fCLK
FSL_RestoreInterruptTable	―	229 / fCLK
FSL_BlankCheck	3298 / fCLK + 124	4574 / fCLK + 401
FSL_Erase	4675 / fCLK + 401	64468 / fCLK + 266193
FSL_IVerify	―	7659 / fCLK + 7534
FSL_Write	3121 / fCLK + 66 +(591 / fCLK + 112) × W	3121 / fCLK+ 66 +(1108 / fCLK+ 1085) × W
FSL_GetSecurityFlags	―	331 / fCLK
FSL_GetBootFlag	―	328 / fCLK
FSL_GetSwapState	―	206 / fCLK
FSL_GetBlockEndAddr	―	368 / fCLK
FSL_GetFlashShieldWindow	―	307 / fCLK
FSL_SwapBootCluster	―	419 / fCLK + 32
FSL_SwapActiveBootCluster	1938 / fCLK + 50	124221 / fCLK + 538064
FSL_InvertBootFlag	1565 / fCLK + 18	123847 / fCLK + 538032
FSL_SetBlockEraseProtectFlag	1571 / fCLK + 18	123853 / fCLK + 538032
FSL_SetWriteProtectFlag	1569 / fCLK + 18	123852 / fCLK + 538032
FSL_SetBootClusterProtectFlag	1571 / fCLK + 18	123853 / fCLK + 538032
FSL_SetFlashShieldWindow	1356 / fCLK + 18	123646 / fCLK + 538032
FSL_StatusCheck	―	―
FSL_StandBy	―	―
FSL_WakeUp	―	―
FSL_ForceReset	―	―
FSL_GetVersionString	―	10 / fCLK

備考 1.　fCLK：CPU の動作周波数(例：20MHz 時の fCLK = 20)
　　2.　W　：1 ワード＝4 バイト単位の書き込みデータ長（ 例：2 ワード/8 バイト指定時 W = 2 ）

(3) FSL_StatusCheck (ステータス・チェック)推奨実行間隔

ステータス・チェック・ユーザ・モードで処理を行う場合、FSL_StatusCheck 関数でステータス・チェックを行いますが、シーケンサの制御が終了する前に FSL_StatusCheck 関数を実行しても意味がないため、各フラッシュ関数で実行している処理毎に、一定の間隔を空ける事でステータス・チェック処理の効率が向上します。また、FSL_Write 関数による書き込みは、4byte 毎にステータス・チェック処理によるトリガーが必要になるため、4byte 単位でのステータス・チェック処理が必要となります。

- ステータス・チェック・ユーザ・モードで 12byte の書き込みを行う場合、シーケンサは 4byte 毎に書き込みを行うため、4byte の書き込み終了時に FSL_StatusCheck 関数によって次の書き込みへのトリガーが必要となります。4byte 以降の書き込み処理が残っている状態で FSL_StatusCheck 関数を実行しない場合、次の書き込み処理に遷移できないため、書き込み処理が終了しません。

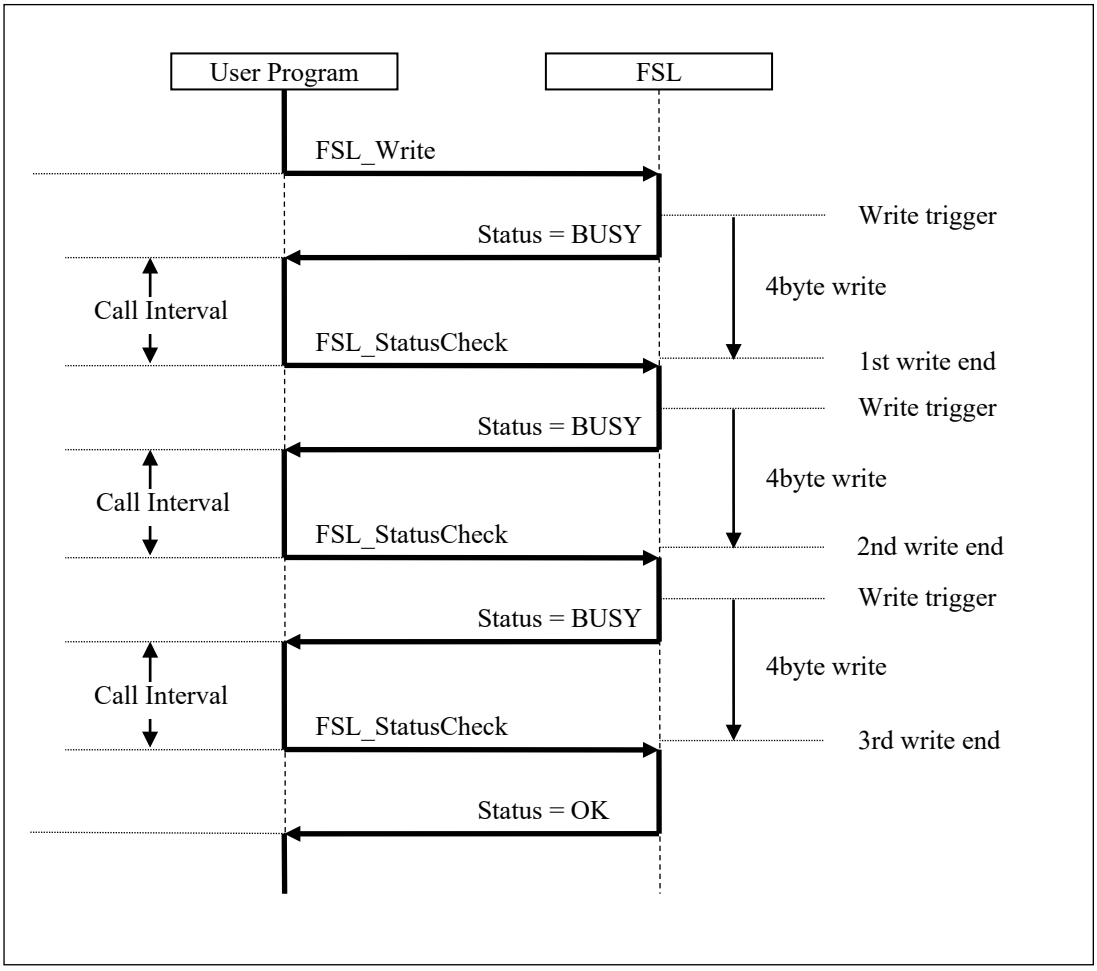


図 2－7　FSL_Write 関数のステータス・チェック推奨実行間隔の概念図(12byte 書き込みの場合)

・　ステータス・チェック・ユーザ・モードで FSL_Write 以外の処理を行う場合、シーケンサは全ての処理を終了するまで Busy 状態となり、FSL_StatusCheck 関数によるトリガーは必要ありません。

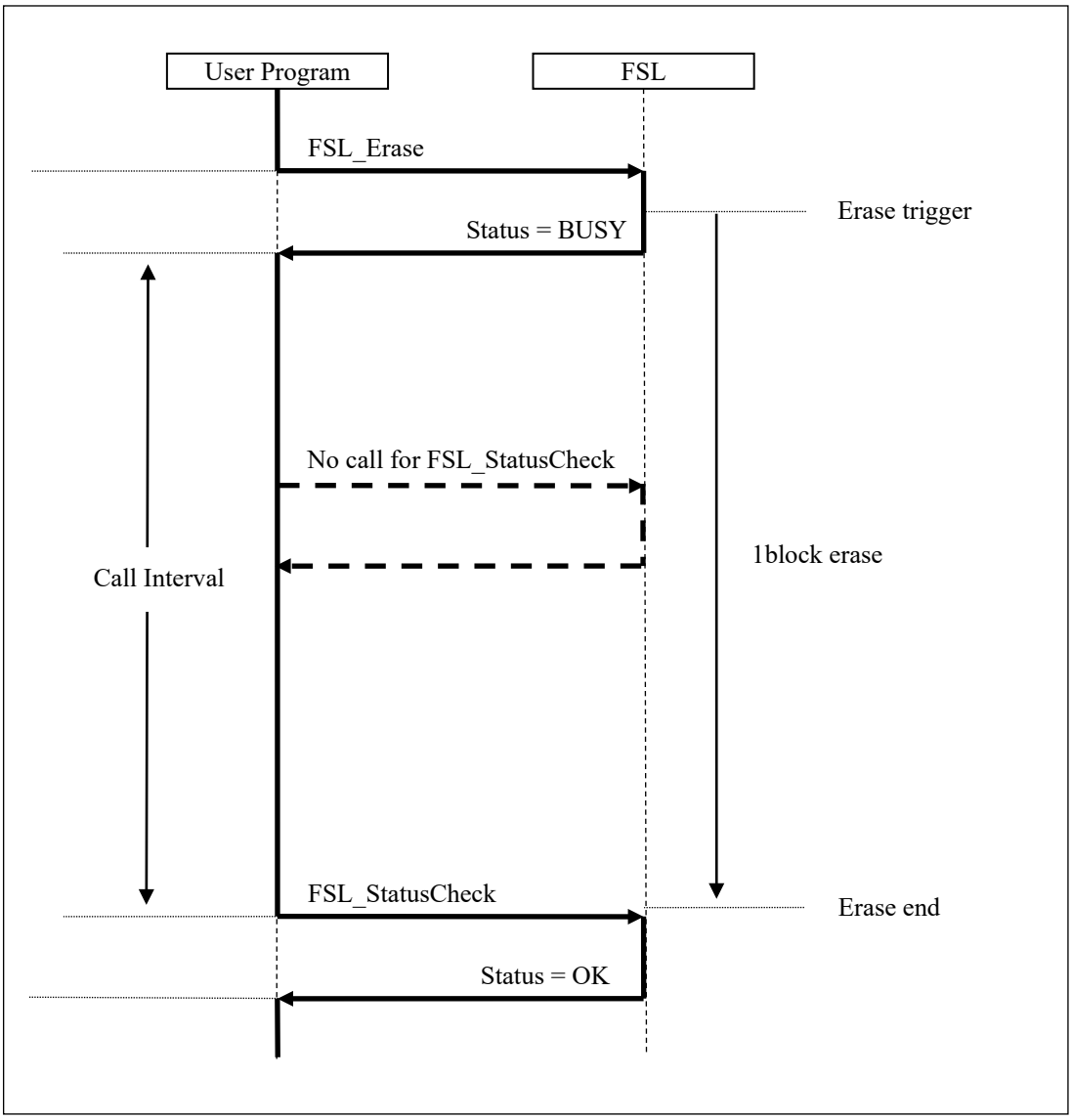


図 2－8　FSL_Write 関数以外のステータス・チェック推奨実行間隔の概念図(消去の場合)

表 2―5 ステータス・チェック・ユーザ・モードのステータスチェック推奨実行間隔
（フルスピード・モード）

FSL_Functions		Call Interval (μs)
FSL_Init		―
FSL_Open		―
FSL_Close		―
FSL_PrepareFunctions		―
FSL_PrepareExtFunctions		―
FSL_ChangeInterruptTable		―
FSL_RestoreInterruptTable		―
FSL_BlankCheck		1569 / fCLK + 98
FSL_Erase	case : block is blanked	1490 / fCLK + 97
	case : block is not blanked	3092 / fCLK + 6471
FSL_IVerify		7181 / fCLK+ 1041
FSL_Write 注		72 / fCLK + 60
FSL_GetSecurityFlags		―
FSL_GetBootFlag		―
FSL_GetSwapState		―
FSL_GetBlockEndAddr		―
FSL_GetFlashShieldWindow		―
FSL_SwapBootCluster		―
FSL_SwapActiveBootCluster		6431 / fCLK + 7053
FSL_InvertBootFlag		
FSL_SetBlockEraseProtectFlag		
FSL_SetWriteProtectFlag		
FSL_SetBootClusterProtectFlag		
FSL_SetFlashShieldWindow		
FSL_StatusCheck		―
FSL_StandBy		―
FSL_WakeUp	case : block is blanked	1490 / fCLK + 97
	case : block is not blanked	3092 / fCLK + 6471
FSL_ForceReset		―
FSL_GetVersionString		―

備考. fCLK：CPU の動作周波数(例：20MHz 時の fCLK = 20)

注.　FSL_Write 関数は 4 バイト毎の推奨間隔時間

表 2−6 ステータス・チェック・ユーザ・モードのステータスチェック推奨実行間隔
（ワイド・ボルテージ・モード）

FSL_Functions		Call Interval (μs)
FSL_Init		−
FSL_Open		−
FSL_Close		−
FSL_PrepareFunctions		−
FSL_PrepareExtFunctions		−
FSL_ChangeInterruptTable		−
FSL_RestoreInterruptTable		−
FSL_BlankCheck		1310 / fCLK + 335
FSL_Erase	case : block is blanked	1289 / fCLK + 335
	case : block is not blanked	2689 / fCLK + 6959
FSL_IVerify		4366 / fCLK + 7468
FSL_Write 注		67 / fCLK + 112
FSL_GetSecurityFlags		−
FSL_GetBootFlag		−
FSL_GetSwapState		−
FSL_GetBlockEndAddr		−
FSL_GetFlashShieldWindow		−
FSL_SwapBootCluster		−
FSL_SwapActiveBootCluster		5728 / fCLK + 8445
FSL_InvertBootFlag		
FSL_SetBlockEraseProtectFlag		
FSL_SetWriteProtectFlag		
FSL_SetBootClusterProtectFlag		
FSL_SetFlashShieldWindow		
FSL_StatusCheck		−
FSL_StandBy		−
FSL_WakeUp	case : block is blanked	1289 / fCLK + 335
	case : block is not blanked	2689 / fCLK + 6959
FSL_ForceReset		−
FSL_GetVersionString		−

備考. fCLK：CPU の動作周波数(例：20MHz 時の fCLK = 20)

注.　FSL_Write 関数は 4 バイト毎の推奨間隔時間

2.2　ソフトウェア環境

フラッシュ・セルフ・プログラミング・ライブラリ Type01 では、該当プログラムをユーザ領域に配置するため、使用するライブラリの容量のプログラム領域を消費し、フラッシュ・セルフ・プログラミング・ライブラリ Type01 自身は、CPU、スタック、データ・バッファを使用します。

★　また、フラッシュ・セルフ・プログラミング・ライブラリ Type01 は、CA78K0R コンパイラ用(V2.20)と CC-RL コンパイラ用(V2.21)があります。各表中では、CA78K0R コンパイラ用(V2.20)を「CA78」、CC-RL コンパイラ用(V2.21)を「CCRL」、LLVM コンパイラ用(V2.21)を「LLVM」と略す場合があります。

表 2−7 に、必要となるソフトウェア・リソースの一覧[注 1,2]、図 2−9、2−10 に RAM の配置イメージ例を示します。

★　**表 2−7　フラッシュ・セルフ・プログラミング・ライブラリ Type01 ソフトウェア・リソース**

項目	容量(バイト)		フラッシュ・セルフ・プログラミング・ライブラリ Type01 の使用領域[注 1,2]
	CA78	CCRL LLVM	
セルフ RAM[注 3]	0 ～ 1024[注 3]	0 ～ 1024[注 3]	RL78 ファミリ フラッシュ・セルフ・プログラミング・ライブラリ Type01 で使用するセルフ RAM 領域は デバイス毎に異なります。詳細は、『RL78 ファミリ セルフプログラミングライブラリ セルフ RAM リスト(R20UT2943)』を参照してください。
スタック[表 2-8]	MAX 46	MAX 50	セルフ RAM、FFE20H-FFEFFH 以外の RAM 領域に配置可能
データ・バッファ[表 2-9、注 4]	1 ～ 256	1 ～ 256	
ライブラリ関数の引数	0 ～ 8	0 ～ 8	
ライブラリ・サイズ 表 2-10、表 2-11	ROM:MAX 1252	ROM:MAX 1294	セルフ RAM、FFE20H-FFEFFH 以外のプログラム領域に配置可能
	RAM:0 ～ 447	RAM:0 ～ 468	セルフ RAM、FFE20H-FFEFFH、内蔵 ROM 以外のプログラム領域に配置可能

注 1.　『RL78 ファミリ セルフプログラミングライブラリ セルフ RAM リスト(R20UT2943)』に掲載の無い製品については、お問い合わせください。

2.　R5F10266 製品はセルフ・プログラミング機能に対応していません。

3.　フラッシュ・セルフ・プログラミング・ライブラリでワーク・エリアとして使用する領域を本書、及びリリース・ノートではセルフ RAM と呼びます。セルフ RAM はマッピングされず、フラッシュ・セルフ・プログラミング・ライブラリ実行時に自動的に使用される（以前のデータは破壊される）領域のため、ユーザ設定等は必要ありません。フラッシュ・セルフ・プログラミング・ライブラリを使用していない状態の場合は、通常の RAM 空間として使用できます。

4.　データ・バッファは、フラッシュ・セルフ・プログラミング・ライブラリ内部処理で使用するワーク領域、または FSL_Write 関数では設定するデータを配置する領域として使用します。必要となるサイズは、使用する関数によって異なります。

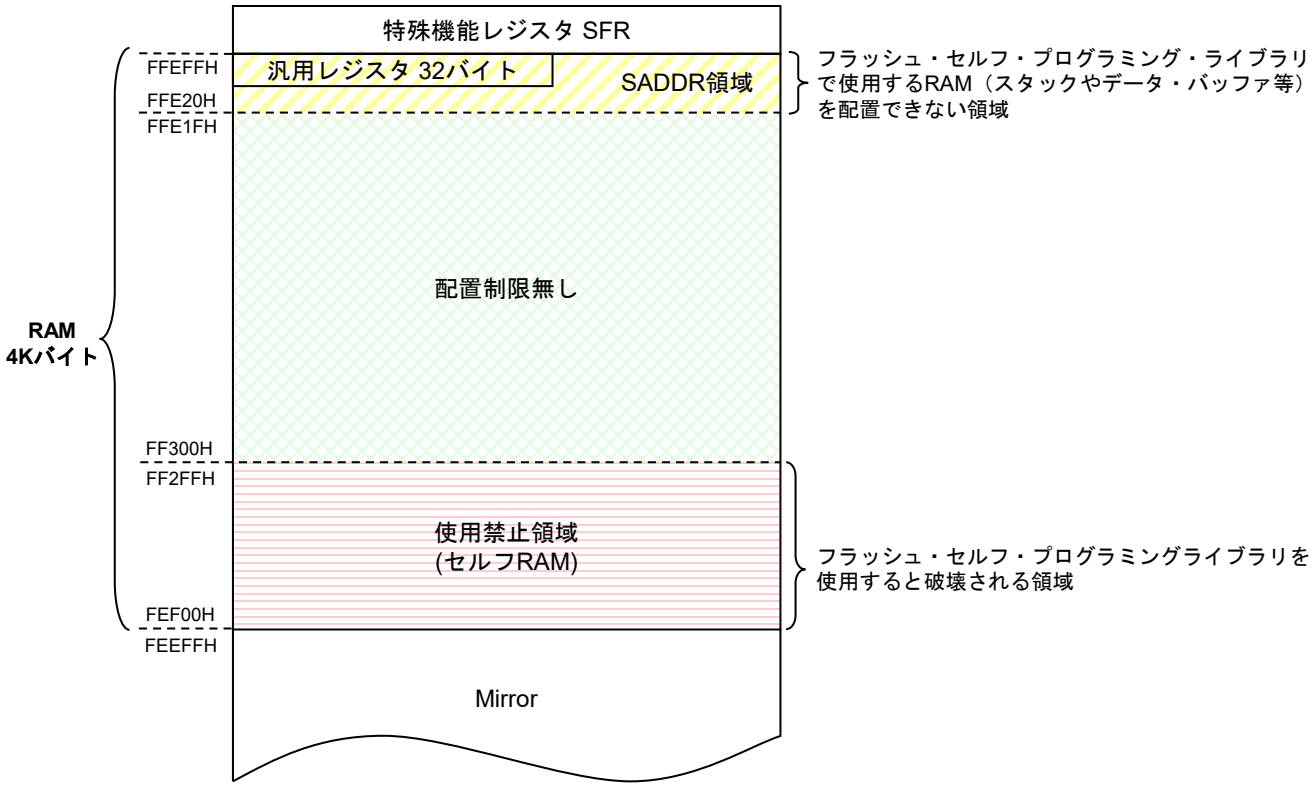


図 2－9　RAM の配置イメージ例 1 / セルフ RAM 有り（RL78/G13：RAM 4KB/ROM 64KB 製品）

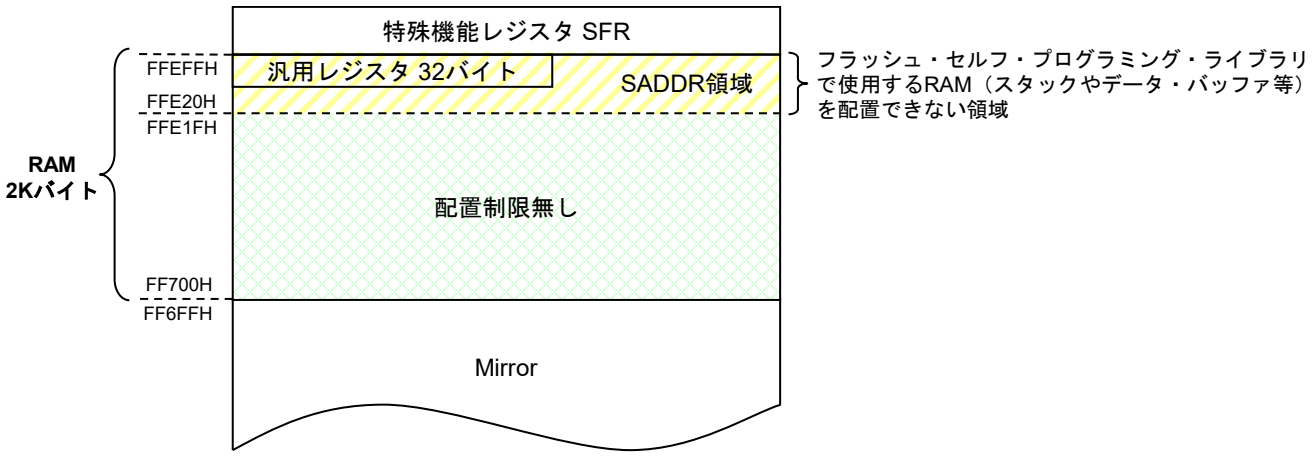


図 2－10　RAM の配置イメージ例 2 / セルフ RAM 無し（RL78/G13：RAM 2KB/ROM 32KB 製品）

★ 表 2-8　フラッシュ関数のスタック・サイズ

関数名	バイト		関数名	バイト	
	CA78	CCRL LLVM		CA78	CCRL LLVM
FSL_Init	40	44	FSL_GetBlockEndAddr	36	40
FSL_Open	0	2	FSL_GetFlashShieldWindow	46	50
FSL_Close	0	2	FSL_SwapBootCluster	38	40
FSL_PrepareFunctions	10	12	FSL_SwapActiveBootCluster	42	46
FSL_PrepareExtFunctions	10	12	FSL_InvertBootFlag	42	46
FSL_ChangeInterruptTable	30	32	FSL_SetBlockEraseProtectFlag	42	46
FSL_RestoreInterruptTable	30	32	FSL_SetWriteProtectFlag	42	46
FSL_BlankCheck	42	46	FSL_SetBootClusterProtectFlag	42	46
FSL_Erase	42	46	FSL_SetFlashShieldWindow	42	46
FSL_IVerify	42	46	FSL_StatusCheck	30	34
FSL_Write	42	46	FSL_StandBy	30	34
FSL_GetSecurityFlags	46	50	FSL_WakeUp	42	46
FSL_GetBootFlag	46	50	FSL_ForceReset	0	2
FSL_GetSwapState	36	40	FSL_GetVersionString	0	2

※ FSL 関数呼び出し元からの CALL 分のスタックは含まれておりません。

表 2-9　フラッシュ関数のデータ・バッファ使用サイズ

関数名	バイト	関数名	バイト
FSL_Init	0	FSL_GetBlockEndAddr	4
FSL_Open	0	FSL_GetFlashShieldWindow	4
FSL_Close	0	FSL_SwapBootCluster	0
FSL_PrepareFunctions	0	FSL_SwapActiveBootCluster	0
FSL_PrepareExtFunctions	0	FSL_InvertBootFlag	0
FSL_ChangeInterruptTable	0	FSL_SetBlockEraseProtectFlag	0
FSL_RestoreInterruptTable	0	FSL_SetWriteProtectFlag	0
FSL_BlankCheck	0	FSL_SetBootClusterProtectFlag	0
FSL_Erase	0	FSL_SetFlashShieldWindow	4
FSL_IVerify	0	FSL_StatusCheck	0
FSL_Write 注	4 ～ 256	FSL_StandBy	0
FSL_GetSecurityFlags	2	FSL_WakeUp	0
FSL_GetBootFlag	1	FSL_ForceReset	0
FSL_GetSwapState	1	FSL_GetVersionString	0

注.　FSL_Write 関数は書き込みサイズ分(ワード単位)の容量が必要です。

例：2 ワード（1 ワード = 4 バイト）分の書き込み = 2 × 4 = 8 バイト

・フラッシュ・セルフ・プログラミング・ライブラリのコード・サイズ

(1) 全て ROM に配置する場合のコード・サイズ

フラッシュ・セルフ・プログラミング・ライブラリを全て ROM に配置して使用する場合のコード・サイズです。RAM にコードを配置する必要はありませんが、使用方法に制限があり、一部の関数が使用できなくなります。詳細については **6. 2** フラッシュ関数のセグメントの項を参照してください。

★ 表 2-10　全て ROM に配置する場合のコード・サイズ

コード・サイズの条件	CA78K0R コンパイラ用 V2.20		CC-RL コンパイラ用 V2.21 LLVM コンパイラ用 V2.21	
	RAM 容量(バイト)	ROM 容量(バイト)	RAM 容量(バイト)	ROM 容量(バイト)
全ての関数を登録した場合のコード･サイズ ※一部の関数は使用できません。	0	1252	0	1294
以下の関数を使用した場合のコード･サイズ ・FSL_Init ・FSL_Open ・FSL_Close ・FSL_PrepareFunctions ・FSL_BlankCheck ・FSL_Erase ・FSL_IVerify ・FSL_Write ・FSL_StatusCheck	0	500	0	502

(2) 一部 RAM に配置する場合のコード・サイズ(BGO を使用する場合)

・　フラッシュ・セルフ・プログラミング中に BGO(バック・グラウンド・オペレーション)機能を使用する場合のコード・サイズです。BGO 機能を使用する場合は FSL_RCD セグメントを RAM に配置する必要があります。また､FSL_RCD セグメントを RAM に展開するためにはプログラムの ROM 化処理を行う必要があるため､別途 ROM 化用として FSL_RCD セグメント分の ROM 容量が必要となります。

★ 表 2-11 一部 RAM に配置する場合のコード・サイズ

コード・サイズの条件	CA78K0R コンパイラ用 V2.20		CC-RL コンパイラ用 V2.21 LLVM コンパイラ用 V2.21	
	RAM 容量(バイト)	ROM 容量(バイト)	RAM 容量(バイト)	ROM 容量(バイト)
全ての関数を登録した場合のコード･サイズ	447 (FSL_RCD)	805 + ROM 化が必要な容量 (447)	468 (FSL_RCD)	826 + ROM 化が必要な容量 (468)
以下の関数を使用した場合のコード･サイズ ・FSL_Init ・FSL_Open ・FSL_Close ・FSL_PrepareFunctions ・FSL_BlankCheck ・FSL_Erase ・FSL_IVerify ・FSL_Write ・FSL_StatusCheck	66 (FSL_RCD)	434 + ROM 化が必要な容量 (66)	88 (FSL_RCD)	502 + ROM 化が必要な容量 (88)

備考.　本表はフラッシュ・セルフ・プログラミング・ライブラリ分のコード・サイズのみについて記載しています。BGO を行う場合はユーザ・プログラムも RAM 上に配置する必要があるため、別途ユーザ・プログラム分のコード・サイズの容量も必要となります。また、ROM 化したプログラムを RAM に展開するプログラム分のコード・サイズも必要になります。ROM 化機能の詳細については使用する予定の開発ツール等のユーザーズマニュアルを参照してください。

2. 2. 1　セルフ RAM

フラッシュ・セルフ・プログラミング・ライブラリは、ワーク・エリアとして 1 K バイトの RAM 領域を使用する場合があります。この領域をセルフ RAM と呼びます。このセルフ RAM で使用するデータは、ライブラリ内で定義されているため、ユーザがデータを定義する必要はありません。

フラッシュ･セルフ･プログラミング･ライブラリ関数を呼び出すとセルフ RAM 領域のデータが書き換わります。

また、フラッシュ・セルフ・プログラミングで使用するセルフ RAM の領域はマイコンによって異なり、ユーザ RAM を使用する場合があります。

★　この場合、ユーザがユーザ RAM 上にセルフ RAM 領域の確保をする必要がありますので、リンク時にセルフ RAM 領域の確保(CA78K0R コンパイラではリンクディレクティブファイルで設定可能。CC-RL コンパイラではセクションを配置しないことで設定可能。LLVM ではリンカスクリプトファイルで設定可能)を行ってください。リンクディレクティブファイルでの設定方法については、リリース・ノートの『内蔵 RAM 領域の定義』の章をご参照ください。

2. 2. 2　レジスタ・バンク

フラッシュ・セルフ・プログラミング・ライブラリはユーザが選択しているレジスタ・バンクの汎用レジスタ、ES/CS レジスタ、 SP および PSW を使用します。

2. 2. 3　スタック、データ・バッファ

フラッシュ・セルフ・プログラミング・ライブラリは、シーケンサを使用してコード・フラッシュ・メモリへの書き込みを行いますが、事前の設定や制御を行うために CPU を使用します。このため、フラッシュ・セルフ・プログラミング・ライブラリを使用するためには、ユーザ・プログラムで指定されているスタックも必要となります。

★　備考　ユーザが指定するアドレスに、スタック、データ・バッファを配置するためには、CA78K0R コンパイラではリンク・ディレクティブを使用し、CC-RL コンパイラでは、セクションの配置をリンカオプションで設定します。また、LLVM コンパイラではリンカスクリプトファイルを使用します。

・スタック

ユーザ･プログラムで使用するスタックに加え、フラッシュ関数で必要となるスタックの容量を事前に確保し、フラッシュ・セルフ・プログラミング動作におけるスタック処理で、ユーザ使用 RAM が破壊されないように配置する必要があります。スタックの指定可能範囲は、セルフ RAM 及び FFE20H-FFEFFH 以外の内蔵 RAM となります。

・データ・バッファ

データ・バッファは、フラッシュ・セルフ・プログラミング・ライブラリ内部処理で使用するワーク領域、または FSL_Write 関数では設定するデータを配置する領域として使用します。

データ・バッファの先頭アドレスの指定可能範囲は、スタックと同様、セルフ RAM 及び FFE20H-FFEFFH 以外の内蔵 RAM となります。

2. 2. 4　フラッシュ・セルフ・プログラミング・ライブラリ

すべてのフラッシュ関数がリンクされるのではなく、使用するフラッシュ関数に限定してリンク注します。

・フラッシュ・セルフ・プログラミング・ライブラリのメモリ配置について
フラッシュ・セルフ・プログラミング・ライブラリでは使用する関数や変数にセグメントが割り当てられており、フラッシュ・セルフ・プログラミング・ライブラリで使用される領域を特定の場所に指定することができます。
詳しくは、6.2 フラッシュ関数のセグメントの項、またはインストーラに添付されている文書「リリース・ノート」を参照してください。

注　アセンブリ言語の場合は、インクルード・ファイルから使用しない関数を削除することで、使用するフラッシュ関数に限定してリンクできます。

2. 2. 5　プログラム領域

フラッシュ・セルフ・プログラミング・ライブラリ、及びフラッシュ・セルフ・プログラミング・ライブラリを使用するユーザ・プログラムを配置する領域です。

RL78 マイクロコントローラのフラッシュ・セルフ・プログラミングでは、シーケンサを使用してコード・フラッシュ・メモリの書き換えを行うため、コード・フラッシュ・メモリの書き換え中にユーザ・プログラムを動作させる事が可能です。（バック・グラウンド・オペレーション）

ただし、コード・フラッシュ・メモリの書き換え中はコード・フラッシュ・メモリに配置したプログラムが参照できなくなるため、使用方法によっては、ユーザ・プログラムやフラッシュ関数で使用する一部のセグメントを RAM 上に配置する必要があります。

詳しくは、第 6 章 フラッシュ関数の各項を参照してください。

2. 2. 6　プログラムの ROM 化

フラッシュ・セルフ・プログラミングを使用するユーザ・プログラムやライブラリを RAM 上に配置させて動作させる場合、対象となるプログラムを ROM 化してコード・フラッシュ・メモリ等に配置、フラッシュ・セルフ・プログラミングで使用する前にプログラムを RAM へ展開する必要があります。

RAM 上に配置したプログラムの ROM 化機能については、使用される開発ツール等に添付されているユーザーズマニュアル等を参照してください。

2. 3　プログラミング環境に関する注意事項

（1）　フラッシュ・セルフ・プログラミング実行中に EEPROM エミュレーション・ライブラリ、またはデータ・フラッシュ・ライブラリを実行しないで下さい。EEPROM エミュレーション・ライブラリ、またはデータ・フラッシュ・ライブラリを使用する場合は、必ず FSL_Close まで実行し、フラッシュ・セルフ・プログラミング・ライブラリを終了状態にする必要があります。
EEPROM エミュレーション・ライブラリ、またはデータ・フラッシュ・ライブラリ実行後にフラッシュ・セルフ・プログラミング・ライブラリを使用する場合は初期化関数(FSL_Init)から処理を行う必要があります。

（2）　フラッシュ・セルフ・プログラミング実行中に STOP 命令、及び HALT 命令は実行しないで下さい。STOP 命令、及び HALT 命令を実行する必要がある場合は、FSL_StandBy 関数によってフラッシュ・セルフ・プログラミングを一時停止させるか、FSL_Close 関数まで実行し、フラッシュ・セルフ・プログラミングを終了させてください。

（3）　ウォッチドック・タイマはセルフ・プログラミング実行中も停止しません。ステータス・チェック・インターナル・モードの場合、ウォッチドック・タイマ割り込み間隔を FSL_SetXXX、 FSL_SwapActiveBootCluster、FSL_InvertBootFlag の実行時間より短くしないでください。

（4）　フラッシュ・セルフ・プログラミングによるコード・フラッシュ・メモリ操作中はコード・フラッシュ・メモリを読み出せません。

（5）　フラッシュ関数で使用するデータ・バッファ(引数)やスタックを FFE20H(FE20H)以上のアドレスに配置しないでください。

（6）　フラッシュ・セルフ・プログラミング実行中に データ・トランスファ・コントローラ（DTC）を使用する場合は、DTC で使用する RAM 領域をセルフ RAM、および FFE20H(FE20H)以上のアドレスに配置しないでください。

（7）　フラッシュ・セルフ・プログラミングが終了するまで、フラッシュ・セルフ・プログラミングが使用する RAM 領域(セルフ RAM を含む)を破壊しないでください。

（8）　フラッシュ関数を割り込み処理内で実行しないで下さい。フラッシュ関数は多重実行に対応していないため、割り込み処理内で実行された場合は動作保証が出来ません。

（9）　OS 上でフラッシュ・セルフ・プログラミングを実行する場合は、複数のタスクからフラッシュ関数を実行しないで下さい。フラッシュ関数は多重実行に対応していないため、複数のタスクで実行された場合は動作保証が出来ません。

（10）　フラッシュ・セルフ・プログラミングを開始する前に高速オンチップ・オシレータを起動しておく必要があります（RL78 マイクロコントローラ内のハードウェアがフラッシュ書き換えの際に使用します）。

（11） CPU の動作周波数と初期化関数（FSL_Init）で設定する CPU の動作周波数値について、以下の点に注意してください。

- CPU の動作周波数を 4MHz 未満[注]で使用する場合は、1MHz, 2MHz, 3MHz を使用することができます。（1.5MHz のように整数値でない周波数は使用できません）初期化関数で設定する動作周波数値も 1, 2, 3 の整数値を設定してください。
- CPU の動作周波数を 4MHz 以上[注]で使用する場合は、小数点以下の数値を持つ周波数を使用することができます。ただし、初期化関数（FSL_Init）で設定する動作周波数は、小数点以下を切り上げた整数値を設定してください。（例：4.5MHz の場合は、初期化関数で 5 を設定してください）
- 高速オンチップ・オシレータの周波数ではありません（CPU の動作周波数に高速オンチップ・オシレータを使用しない場合）。

注. CPU の動作周波数の範囲については対象となる RL78 マイクロコントローラのユーザーズマニュアルを参照してください。

（12） フラッシュ・セルフ・プログラミング・ライブラリ関数で使用する引数(RAM)は一度初期化してください。初期化をしない場合、RAM パリティ・エラーが検出され、RL78 マイクロコントローラにリセットが発生する可能性があります。
RAM パリティ・エラーについては、対象となる RL78 マイクロコントローラのユーザーズマニュアルを参照してください。

（13） コード・フラッシュ・メモリへの書き込みはブランク状態か、もしくは消去済みの領域へのみ行えます。既に書き込みを行った領域に対し、消去を行わずに再度書き込み(上書き)を行う事はできません。消去を行わずに再度書き込みを行った場合、コード・フラッシュ・メモリへダメージを与える可能性があります。

（14） R5F10266 製品はフラッシュ・セルフ・プログラミング機能を使用できません。

（15） 一部の RL78 マイクロコントローラはフラッシュ・セルフ・プログラミング中の割り込みに対応していない場合があります。ご使用になる RL78 マイクロコントローラがフラッシュ・セルフ・プログラミング中の割り込みに対応しているかについては､対象となる RL78 マイクロコントローラのユーザーズマニュアルを参照してください。

（16） 一部の RL78 マイクロコントローラはブート･スワップ機能に対応していない場合があります｡ご使用になる RL78 マイクロコントローラがブート・スワップ機能に対応しているかについては、対象となる RL78 マイクロコントローラのユーザーズマニュアルを参照してください。

（17） 一部の RL78 マイクロコントローラはフラッシュ・セルフ・プログラミングによるセキュリティ設定機能に対応していない場合があります。ご使用になる RL78 マイクロコントローラがフラッシュ・セルフ・プログラミングによるセキュリティ設定機能に対応しているかについては、対象となる RL78 マイクロコントローラのユーザーズマニュアルを参照してください。

（18） フラッシュ・セルフ・ライブラリでは、セグメント FSL_BCD および FSL_BECD を 64KB 境界の最終アドレス (?FFFEH-?FFFFH) に配置しないでください。(対象のフラッシュ・セルフ・ライブラリは V2.20 以下です。V2.21 以降のフラッシュ・セルフ・ライブラリは改善済み、非対象です。)

（19）　フラッシュ・セルフ・プログラミング・ライブラリの各セグメント（FSL_FCD, FSL_FECD, FSL_RCD, FSL_BCD, FSL_BECD）は、64KB 境界を跨いで配置することができません。
必ず、64KB 境界を跨がないように配置してください。

（20）　CC-RL のアセンブラを使用する場合、16 進数の Prefix 表現(0x..)と Suffix 表現(..H)は混在できません。ユーザの環境に合わせて fsl.inc 内のシンボル定義を編集することで表現方法を指定してください。

fsl.inc

```
; __FSL_INC_BASE_NUMBER_SUFFIX .SET 1
```

シンボル"__FSL_INC_BASE_NUMBER_SUFFIX"を定義しない場合(初期状態)、Prefix 表現が選択されます。

fsl.inc

```
__FSL_INC_BASE_NUMBER_SUFFIX .SET 1
```

シンボル"__FSL_INC_BASE_NUMBER_SUFFIX"を定義する場合、Suffix 表現が選択されます。

第3章　フラッシュ・セルフ・プログラミング実行中の割り込み

3.1　概　要

フラッシュ環境でも、割り込み処理を使用することが可能[注]です。但しコード・フラッシュ・メモリを制御する時は、通常のユーザ・アプリケーションの割り込みベクタは使用できません。割り込みベクタ変更関数（FSL_ChangeInterruptTable）を使用して、割り込みベクタをRAM に設定しておく必要があります。また、割り込みルーチンもRAM 上に配置しておく必要があります。設定後は、全ての割り込み発生時にRAM 上の１つのベクタに分岐しますので、複数の割り込み要因があり、それぞれ異なる処理を実行したい場合は、割り込み要因を判別する必要があります。

コード・フラッシュ・メモリの書き換えが終了し、割り込みを元のベクタ状態に復元する場合は割り込みベクタ復元関数（FSL_RestoreInterruptTable）を使用し、割り込み先を元の状態に戻します。

注　一部のRL78マイクロコントローラはフラッシュ・セルフ・プログラミング中の割り込みに対応していない場合があります。ご使用になるRL78マイクロコントローラがフラッシュ・セルフ・プログラミング中の割り込みに対応しているかについては、対象となるRL78マイクロコントローラのユーザーズマニュアルを参照してください。

3.2　フラッシュ・セルフ・プログラミング実行中の割り込み

コード・フラッシュ・メモリ制御中の割り込みはコード・フラッシュ・メモリが参照できないため、通常の割り込みベクタでは割り込みを受け付ける事ができません。そのため、割り込みを受け付ける場合はRAM上で割り込みを受け付ける必要があります。

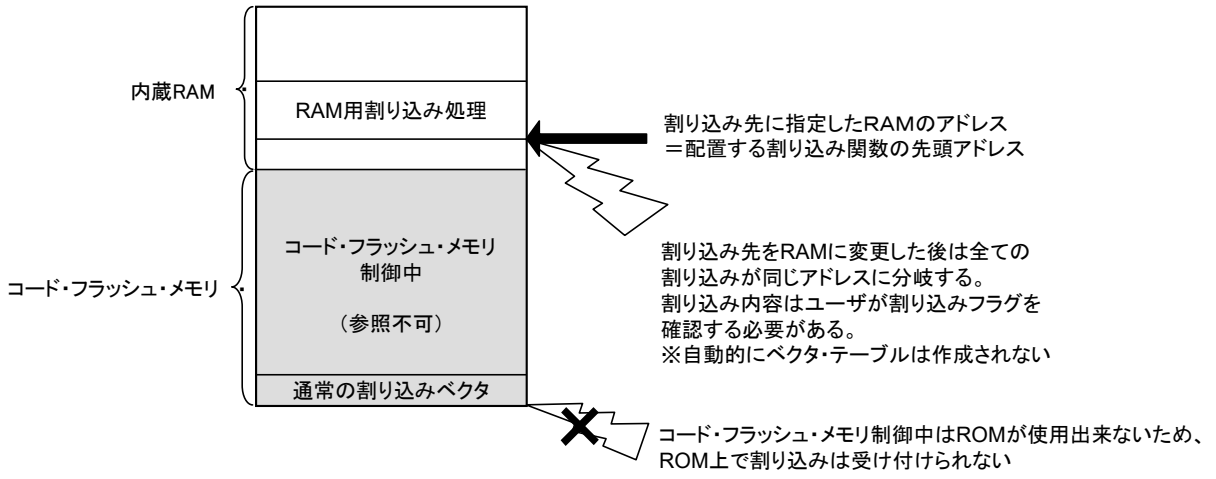


図 3−1　フラッシュ・セルフ・プログラミング実行中の割り込み

3. 3　割り込みに関する注意事項

- アプリケーションにおいて割り込みベクタ変更機能により割り込みベクタを変更する際は、切り替え手順の開始から完了までの間、割り込みを禁止にしてください。

- 割り込みベクタ・アドレスの変更先にFFE20H 以上の値を指定しないでください。

- フラッシュ・セルフ・プログラミング実行中の割り込み処理において、コード・フラッシュ・メモリ領域へのアクセスは禁止です。

- 割り込み処理において、フラッシュ関数の実行は禁止です。

- 割り込み処理で使用するレジスタを退避・復帰するようにしてください。

- RAM 上での割り込み発生時にSFR(割り込み要求フラグIF)を参照することにより割り込み要因を判別できますが、判別後は割り込み要求フラグをクリア(0 を設定)するようにしてください。

- RAM上における割り込み処理は、通常の割り込み応答時間より最大20クロック増加します。

- 割り込みベクタ変更機能で割り込み先を変更した場合、元の割り込みベクタに戻すためには、割り込みベクタ復元機能を実行する必要があります。割り込みベクタ復元機能を実行しなかった場合、フラッシュ・セルフ・プログラミングを終了しても割り込み先が変更された状態のままになります。

- 割り込みベクタ変更機能で割り込み先を変更した後リセットした場合、割り込み先は復元された状態で起動します。

- 一部のRL78マイクロコントローラはフラッシュ・セルフ・プログラミング中の割り込みに対応していない場合があります。ご使用になるRL78マイクロコントローラがフラッシュ・セルフ・プログラミング中の割り込みに対応しているかについては、対象となるRL78マイクロコントローラのユーザーズマニュアルを参照してください。

第4章　セキュリティ設定

コード・フラッシュ・メモリに書かれたユーザ・プログラムの書き換えを禁止するセキュリティ機能をサポート[注]しており、第三者によるプログラムの改ざん防止などに対応可能となっています。

なお、セキュリティ設定についての詳細は、対象デバイスのマニュアルを参照してください。

注　本機能を使用するためには、フラッシュ・セルフ・プログラミングによるセキュリティ設定機能に対応したRL78マイクロコントローラが必要になります。ご使用になるRL78マイクロコントローラがフラッシュ・セルフ・プログラミングによるセキュリティ設定機能に対応しているかについては、対象となるRL78マイクロコントローラのユーザーズマニュアルを参照してください。

4.1　セキュリティ・フラグ

フラッシュ・セルフ・プログラミング・ライブラリはセキュリティ・フラグを設定する関数を実装しています（この関数のAPIの詳細は第6章 フラッシュ関数を参照してください）。

セキュリティ・フラグを設定する関数	機　　能
FSL_SetBlockEraseProtectFlag	ブロック消去禁止フラグを禁止に設定
FSL_SetWriteProtectFlag	書き込み禁止フラグを禁止に設定
FSL_SetBootClusterProtectFlag	ブート領域(ブートクラスタ0)書き換え禁止フラグを禁止に設定

4.2　フラッシュ・シールド・ウインドウ機能

フラッシュ・セルフ・プログラミング実行中のセキュリティ機能の一つとして、フラッシュ・シールド・ウインドウ機能があります。フラッシュ・シールド・ウインドウ機能は、指定したウインドウ範囲以外の書き込みおよび消去を、フラッシュ・セルフ・プログラミング実行中のみ禁止にするセキュリティ機能です。ウインドウ範囲は、スタート・ブロックとエンド・ブロックを指定することで設定できます。ウインドウ範囲以外の領域は、フラッシュ・セルフ・プログラミング実行中には書き込み／消去禁止となります。

フラッシュ・シールド・ウインドウを設定する関数	機　　能
FSL_SetFlashShieldWindow	フラッシュ・シールド・ウインドウの設定

第5章　ブート・スワップ機能

5.1　概　要

ベクタ・テーブル・データ、プログラムの基本機能、およびフラッシュ・セルフ・プログラミング・ライブラリを配置している領域の書き換え中に、電源の瞬断、外部要因によるリセットの発生などにより書き換えが失敗した場合、書き換え中のデータが破壊され、その後のリセットによるユーザ・プログラムの再スタートや、再書き込みができなくなります。この問題を回避するための機能がブート・スワップ機能です。注

注　本機能を使用するためには、ブート・スワップ機能に対応したRL78マイクロコントローラが必要になります。ご使用になるRL78マイクロコントローラがブート・スワップ機能に対応しているかについては、対象となるRL78マイクロコントローラのユーザーズマニュアルを参照してください。

5.2　ブート・スワップ機能

ブート・スワップ機能では、ブート・プログラム領域であるブート・クラスタ0注とブート・スワップ対象領域であるブート・クラスタ1注を置換します。

書き換え処理を行う前に、あらかじめ新しいブート・プログラムをブート・クラスタ1に書き込んでおきます。このブート・クラスタ1とブート・クラスタ0をスワップし、ブート・クラスタ1をブート・プログラム領域にします。

これによって、ブート・プログラム領域の書き換え中に電源の瞬断が発生しても、次のリセット・スタートはブート・クラスタ1からブートを行うため、正常にプログラムが動作します。このあと、必要であれば、ブート・クラスタ0への消去や書き込み処理を行うことも可能です。

注　ブート・クラスタ0：ブート・プログラム領域
　　ブート・クラスタ1：ブート・スワップ対象領域

5. 3　ブート・スワップ手順

図5－1に、フラッシュ・セルフ・プログラミング・ライブラリを利用してブート・スワップを行うフロー例を示します。

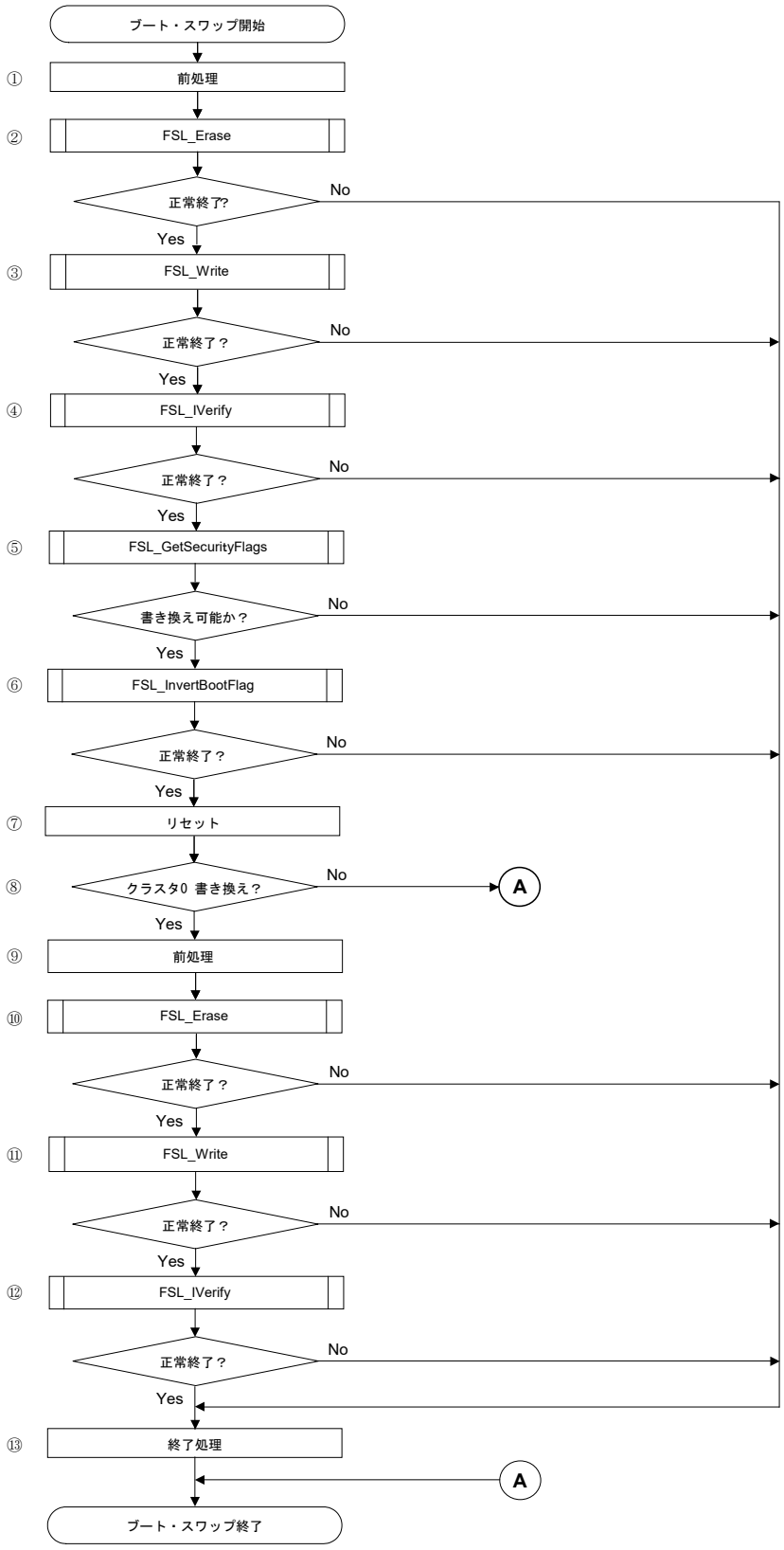


図 5－1　ブート・スワップのフロー例

① 前処理

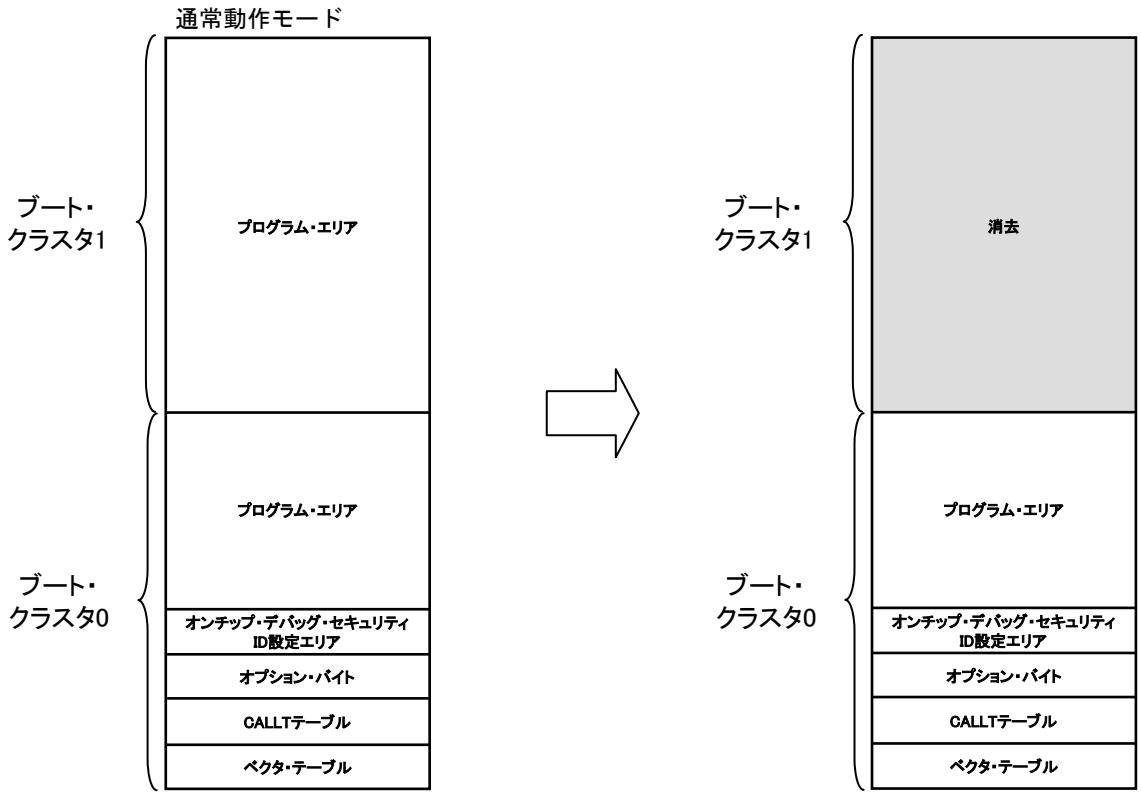
ブート・スワップの前処理

- ソフトウェア環境の設定（データ・バッファの確保など）
- フラッシュ・セルフ・プログラミングの初期化（FSL_Init関数の実行）
- フラッシュ環境の開始（FSL_Open関数の実行）
- フラッシュ関数の準備処理（FSL_PrepareFunctions関数の実行）
- フラッシュ関数（拡張関数）の準備処理（FSL_PrepareExtFunctions関数の実行）
- 書き換えプログラムのROM化を行っている場合はROM化コードのRAM展開処理

② ブート・クラスタ 1 の消去

FSL_Erase 関数の呼び出しにより、ブート・クラスタ1に含まれるブロックを全て消去します。

備考　FSL_Erase 関数はブロック単位で消去を行います。

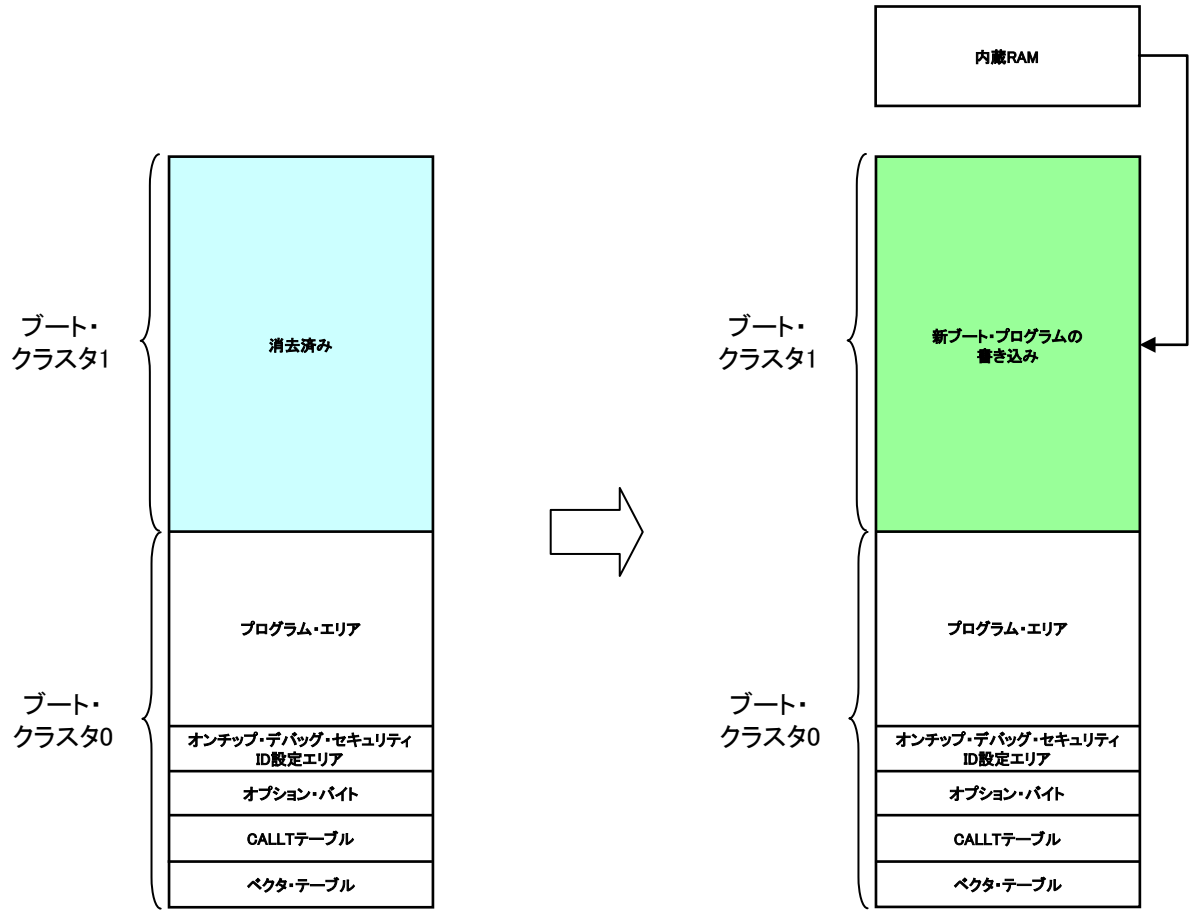


③　ブート・クラスタ 1 へ新しいブート・プログラムをコピー

FSL_Write 関数の呼び出しにより、新ブート・プログラム（ブート・スワップ処理後にブート・プログラム領域として配置したいプログラム）をブート・クラスタ1 に書き込みます。

備考 FSL_Write 関数はワード単位（1 ワード = 4 バイト、最大64 ワード（256 バイト））で書き込みを行います。

新ブート・プログラムを、外部I/F（3線式SIO、UARTなど）経由で内蔵RAMにダウン・ロードし、順次書き込みます。



④　ブート・クラスタ１のベリファイ

FSL_IVerify 関数の呼び出しにより、書き込みを行ったブート・クラスタ1のブロックを全てベリファイします。

備考　FSL_IVerify 関数はブロック単位でベリファイを行います。

⑤　ブート・スワップ・ビットの確認（推奨）

FSL_GetSecurityFlags 関数の呼び出しにより、セキュリティ・フラグ情報を取得し、ブート領域(ブート・クラスタ0)書き換え禁止フラグが1（許可）になっていることを確認します。

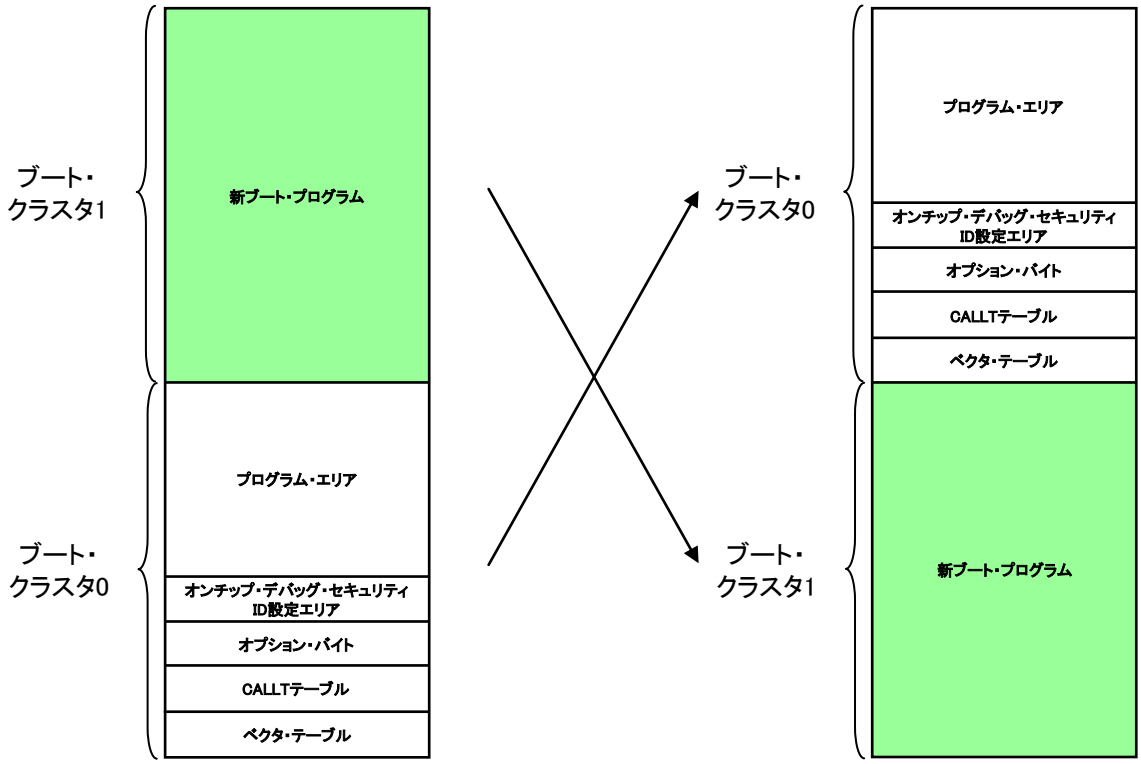
備考　(ブート・クラスタ0)書き換え禁止フラグが0（禁止）になっている場合、⑥のFSL_InvertBootFlag 関数の呼び出しにより、エラーとなります。

⑥　ブート・スワップ・ビットの設定

FSL_InvertBootFlag 関数を実行することにより、ブート・フラグの切り替えを行います。

⑦　イベントの発生

リセットを発生させることにより、ブート・クラスタ1 がブート・プログラム領域になります。



⑧　スワップ処理（ブート・クラスタ 1）の終了

②-⑦の操作により、ブート・クラスタ1 に対するスワップ処理が終了となります。

ブート・クラスタ 0 を書き換える必要がない場合は、そのまま終了してください。

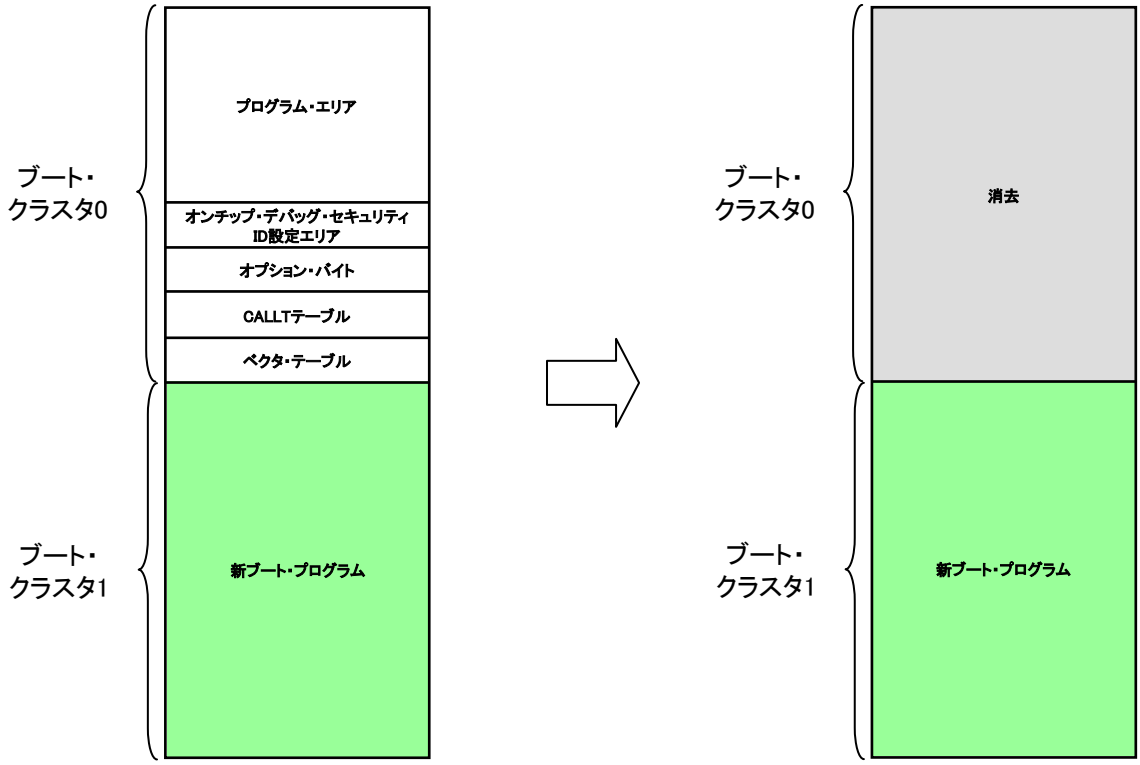
ブート・クラスタ 0 を書き換える必要がある場合は、⑨以降の処理を行ってください。

⑨　前処理

①と同様の処理を行います。

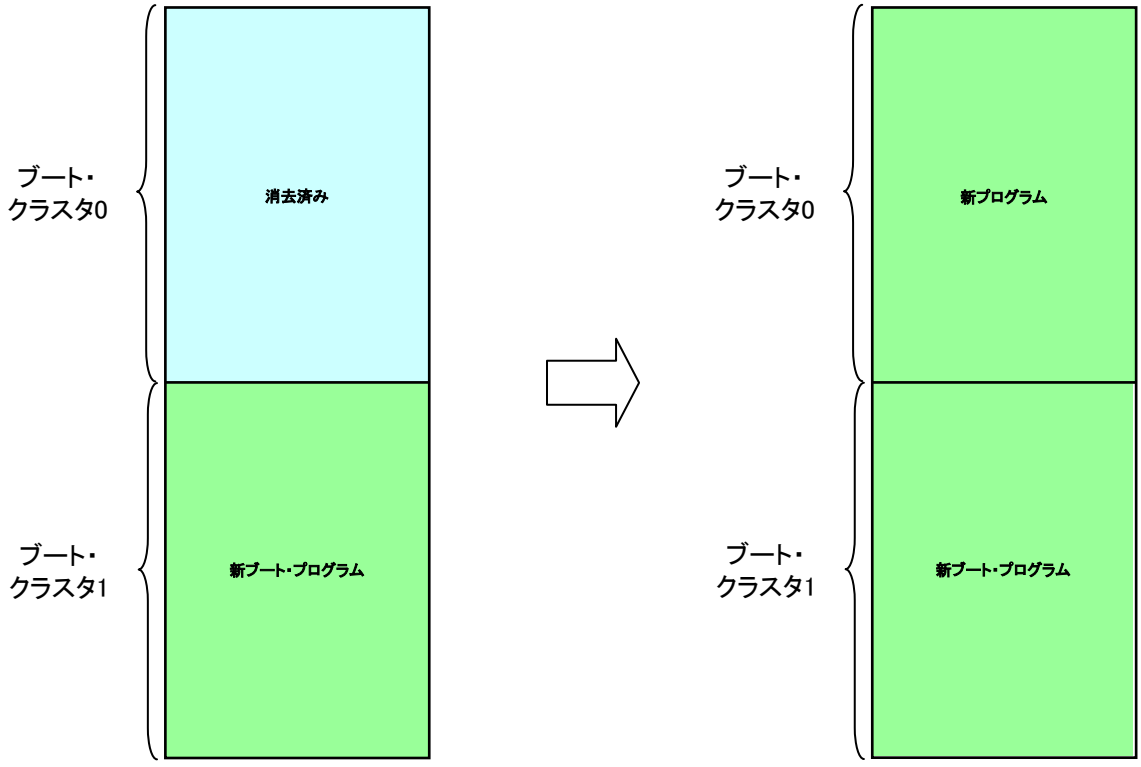
⑩　ブート・クラスタ 0 の消去

FSL_Erase 関数の呼び出しにより、ブート・クラスタ0のブロックを全て消去します。



⑪　ブート・クラスタ 0 への新しいプログラムの書き込み

FSL_Write 関数の呼び出しにより、新しいプログラムの内容をブート・クラスタ0に書き込みます。



⑫　ブート・クラスタ 0 のベリファイ

FSL_IVerify 関数の呼び出しにより、書き込みを行ったブート・クラスタ0のブロックを全てベリファイします。

⑬　終了処理

ブート・スワップの終了処理として、FSL_Close 関数を呼び出します。

5.4　ブート・スワップに関する注意事項

- ブート領域(ブート・クラスタ0)書き換え禁止フラグが0（禁止）になっている場合、ブート・スワップは実行できません。

- ブート・スワップを行なう関数は、スワップ実行後、関数を呼び出した領域へ戻ります。その為、関数を呼び出すプログラムは、スワップを行うブートクラスタ0,1には配置できません。

  備考　該当関数：FSL_SwapBootCluster、FSL_SwapActiveBootCluster

- ブート・スワップ機能を提供するフラッシュ関数には、それぞれ関数毎に注意事項があります。詳しくは、**第6章 フラッシュ関数**を参照してください。

第6章　フラッシュ関数

この章では、フラッシュ関数（フラッシュ・セルフ・プログラミング・ライブラリの関数）の詳細について説明しています。

6.1　フラッシュ関数の種類

フラッシュ・セルフ・プログラミング・ライブラリは、表6－1に示すフラッシュ関数で構成されています。

表 6－1　フラッシュ関数一覧

関数名	説　　明	基本関数[注]	G11対象[注]
FSL_Init	フラッシュ・セルフ・プログラミング環境の初期化	○	○
FSL_Open	フラッシュ環境の開始（フラッシュ・セルフ・プログラミングの開始宣言）	○	○
FSL_Close	フラッシュ環境の終了（フラッシュ・セルフ・プログラミングの終了宣言）	○	○
FSL_PrepareFunctions	フラッシュ関数の準備処理	○	○
FSL_PrepareExtFunctions	フラッシュ関数(拡張関数)の準備処理	-	○
FSL_ChangeInterruptTable	割り込みベクタの変更処理（割り込み先をROMからRAMに変更）	-	-
FSL_RestoreInterruptTable	割り込みベクタの復元処理（割り込み先をRAMからROMに変更）	-	-
FSL_BlankCheck	指定ブロックのブランク・チェック	○	○
FSL_Erase	指定ブロックの消去	○	○
FSL_IVerify	指定ブロックのベリファイ（内部ベリファイ）	○	○
FSL_Write	指定アドレスに対する1-64ワード・データの書き込み（1ワード = 4バイト）	○	○
FSL_GetSecurityFlags	セキュリティ情報の取得	-	-
FSL_GetBootFlag	ブート・フラグ情報の取得	-	○
FSL_GetSwapState	スワップ情報の取得	-	○
FSL_GetBlockEndAddr	指定したブロックの最終アドレスの取得	-	-
FSL_GetFlashShieldWindow	フラッシュ・シールド・ウインドウの開始ブロック番号と終了ブロック番号の取得	-	○
FSL_SwapBootCluster	ブート・スワップを実行し、リセット・ベクタの登録アドレスへジャンプ	-	○
FSL_SwapActiveBootCluster	ブート・フラグの現在値を反転し、ブート・スワップの実行	-	-
FSL_InvertBootFlag	ブート・フラグの現在値を反転	-	○
FSL_SetBlockEraseProtectFlag	ブロック消去禁止フラグを禁止に設定	-	-
FSL_SetWriteProtectFlag	書き込み禁止フラグを禁止に設定	-	-
FSL_SetBootClusterProtectFlag	ブート領域(ブート・クラスタ0)書き換え禁止フラグを禁止に設定	-	-
FSL_SetFlashShieldWindow	フラッシュ・シールド・ウインドウの開始ブロックと終了ブロックの設定	-	○
FSL_StatusCheck	ステータス・チェック処理	○	-
FSL_StandBy	フラッシュ・セルフ・プログラミングの一時停止処理	-	-
FSL_WakeUp	フラッシュ・セルフ・プログラミングの再開処理	-	-
FSL_ForceReset	使用中のマイクロコントローラをリセット	-	○
FSL_GetVersionString	フラッシュ・セルフ・プログラミング・ライブラリのバージョン取得処理	-	-

注.　RL78/G12,L12,G1G グループ製品のサポート対象関数は基本関数のみで、その他の関数はサポート対象外です。RL78/G11 グループ製品でのステータス・チェック・モードは、ステータス・チェック・インターナル・モードのみのサポートになります。

6. 2　フラッシュ関数のセグメント（セクション）

フラッシュ関数の実体は、各グループに分けられており、指定の領域に配置する必要があります。

★　このグループは、CA78K0Rコンパイラでメモリ配置する際にセグメントとして使用され、CC-RLコンパイラとLLVMコンパイラでは、セクションとして使用されます。

セグメント（セクション）は以下のように分けられています。

- FSL_FCD　：環境の初期化等を行う関数郡です。ROM/RAMどちらでも配置可能です。
- FSL_FECD　：セキュリティ情報等の読み出しを行う関数郡です。ROM/RAMどちらでも配置可能です。
- FSL_RCD　：フラッシュの書き換え動作に必要な関数郡です。RAMに配置可能です。ROMに配置する場合は使用制限[注]があります。
- FSL_BCD　：FSL_PrepareFunctions関数で使用する領域です。ROM/RAMどちらでも配置可能です。
- FSL_BECD　：FSL_PrepareExtFunctions関数で使用する領域です。ROM/RAMどちらでも配置可能です。

★　以降、CC-RLコンパイラ用、LLVMコンパイラ用フラッシュ・セルフ・プログラミング・ライブラリを使用する場合は、"セグメント"を"セクション"と読み替えてください。

表 6−2　フラッシュ関数のセグメント一覧

関数名	セグメント名	ROM配置	RAM配置
FSL_Init	FSL_FCD	○	○
FSL_Open	FSL_FCD	○	○
FSL_Close	FSL_FCD	○	○
FSL_PrepareFunctions	FSL_FCD / FSL_BCD	○	○
FSL_PrepareExtFunctions	FSL_FCD / FSL_BECD	○	○
FSL_ChangeInterruptTable	FSL_FCD	○	○
FSL_RestoreInterruptTable	FSL_FCD	○	○
FSL_BlankCheck	FSL_RCD	△注	○
FSL_Erase	FSL_RCD	△注	○
FSL_IVerify	FSL_RCD	△注	○
FSL_Write	FSL_RCD	△注	○
FSL_GetSecurityFlags	FSL_FECD	○	○
FSL_GetBootFlag	FSL_FECD	○	○
FSL_GetSwapState	FSL_FECD	○	○
FSL_GetBlockEndAddr	FSL_FECD	○	○
FSL_GetFlashShieldWindow	FSL_FECD	○	○
FSL_SwapBootCluster	FSL_RCD	△注	○
FSL_SwapActiveBootCluster	FSL_RCD	×	○
FSL_InvertBootFlag	FSL_RCD	△注	○
FSL_SetBlockEraseProtectFlag	FSL_RCD	△注	○
FSL_SetWriteProtectFlag	FSL_RCD	△注	○
FSL_SetBootClusterProtectFlag	FSL_RCD	△注	○
FSL_SetFlashShieldWindow	FSL_RCD	△注	○
FSL_StatusCheck	FSL_RCD	△注	○
FSL_StandBy	FSL_RCD	△注	○
FSL_WakeUp	FSL_RCD	△注	○
FSL_ForceReset	FSL_RCD	△注	○
FSL_GetVersionString	FSL_FCD	○	○

注　ROM上に配置して使用する場合は以下の使用制限があります。

・FSL_SwapActiveBootCluster()関数は使用しない。

・FSL_Init()関数で設定するステータス・チェック・モードの指定をステータス・チェック・インターナル・モードに設定する。

6. 3　割り込みとBGO（バック・グラウンド・オペレーション）

フラッシュ関数はシーケンサを使用しない処理と、シーケンサを使用する処理に分かれ、割り込みの受け付け方法に違いがあります。また、シーケンサを使用する処理はBGO（バック・グラウンド・オペレーション）が可能です。

以下にフラッシュ関数のシーケンサ制御の有無と割り込み受け付け領域の一覧を示します。

表 6−3　フラッシュ関数の割り込み受け付け領域とBGOの一覧

関数名	シーケンサ制御	割り込みの受付[注1]	BGO機能
FSL_Init	無し	ROM：可 RAM：可	無し
FSL_Open			
FSL_Close			
FSL_PrepareFunctions			
FSL_PrepareExtFunctions			
FSL_ChangeInterruptTable		不可	
FSL_RestoreInterruptTable			
FSL_BlankCheck	有り	ROM：不可 RAM：可	有り ※RAM上のみ[注2]
FSL_Erase			
FSL_IVerify			
FSL_Write			
FSL_GetSecurityFlags	無し	ROM：可 RAM：可	無し
FSL_GetBootFlag			
FSL_GetSwapState			
FSL_GetBlockEndAddr			
FSL_GetFlashShieldWindow			
FSL_SwapBootCluster		不可	
FSL_SwapActiveBootCluster	有り	ROM：不可 RAM：可	有り ※RAM上のみ[注2]
FSL_InvertBootFlag			
FSL_SetBlockEraseProtectFlag			
FSL_SetWriteProtectFlag			
FSL_SetBootClusterProtectFlag			
FSL_SetFlashShieldWindow			
FSL_StatusCheck[注3]			無し
FSL_StandBy[注3]			
FSL_WakeUp[注3]			
FSL_ForceReset	無し	ROM：可 RAM：可	
FSL_GetVersionString			

注 1.　関数実行中、あるいはシーケンサ制御中の割り込み受け付けの可否
　　ROM：通常のベクタ割り込み　／　RAM：RAM上での割り込み

2.　BGOを実行する場合は、ユーザ・プログラムと一部のライブラリをRAM上に配置する必要があります。

3.　シーケンサの状態確認や、ブロック消去中のシーケンサ制御を停止・再開させる関数のため、本関数の処理にはBGO機能はありません。

6.4　ステータス・チェック・モード

シーケンサを使用する事でバック・グラウンド・オペレーションが可能な関数は、コード・フラッシュ・メモリへの制御状態を確認するため、ステータス・チェックを行う必要があります。

ステータス・チェックのモードには以下の2種類があり、FSL_Init()関数で設定します。また、それぞれステータス・チェックの方法が異なります。

・ステータス・チェック・ユーザ・モード[注]

フラッシュ関数でシーケンサの制御設定を行った後、ユーザ・プログラムに戻ります。ユーザはシーケンサの状態をステータス・チェック関数（FSL_StatusCheck）で確認する必要がありますが、シーケンサの処理が終了するまでの間、ユーザ・プログラムを動作させる事が可能です。シーケンサ制御中に動作させるユーザ・プログラムや割り込みプログラムはRAM上に配置する必要があります。

・ステータス・チェック・インターナル・モード[注]

フラッシュ関数内でシーケンサの状態を確認し、シーケンサの処理が終了するまでの間、ユーザ・プログラムに戻りません。関数実行中（シーケンサ制御中）に割り込みを動作させる場合は、割り込みプログラムをRAM上に配置する必要があります。

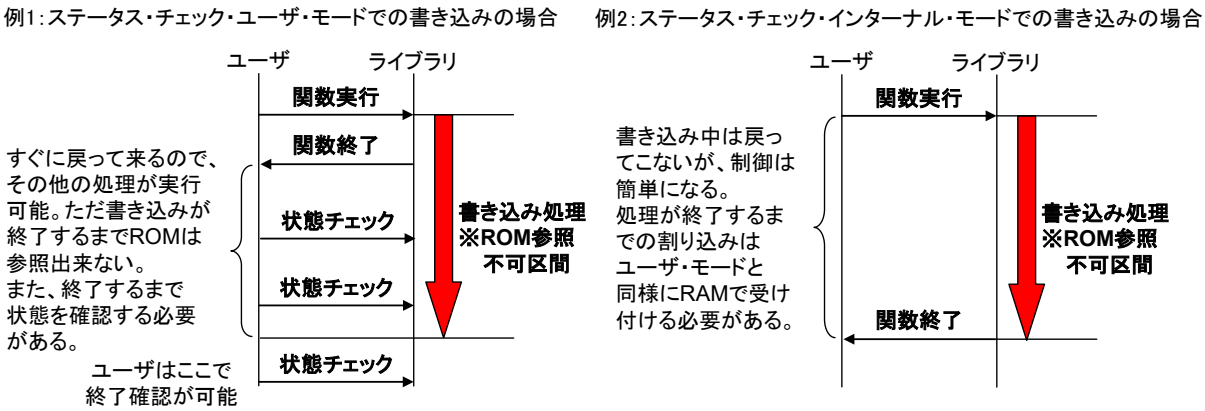


図 6−1　ステータス・チェック・モードの例

注　一部ROM上での配置に使用制限のあるフラッシュ関数のセグメント（FSL_RCD）、及びユーザ・プログラムをROM上に配置して使用する場合は、ステータス・チェック・インターナル・モードのみ使用可能です。

表 6−4　フラッシュ関数のステータス・チェック一覧

関数名	シーケンサ制御	ステータス・チェック
FSL_Init	無し	不要
FSL_Open		
FSL_Close		
FSL_PrepareFunctions		
FSL_PrepareExtFunctions		
FSL_ChangeInterruptTable		
FSL_RestoreInterruptTable		
FSL_BlankCheck	有り	必要
FSL_Erase		
FSL_IVerify		
FSL_Write		
FSL_GetSecurityFlags	無し	不要
FSL_GetBootFlag		
FSL_GetSwapState		
FSL_GetBlockEndAddr		
FSL_GetFlashShieldWindow		
FSL_SwapBootCluster		
FSL_SwapActiveBootCluster	有り	必要
FSL_InvertBootFlag		
FSL_SetBlockEraseProtectFlag		
FSL_SetWriteProtectFlag		
FSL_SetBootClusterProtectFlag		
FSL_SetFlashShieldWindow		
FSL_StatusCheck[注1]		不要
FSL_StandBy[注1]		
FSL_WakeUp[注1, 2]		
FSL_ForceReset	無し	
FSL_GetVersionString		

注 1.　ステータス・チェックを行う関数や、ブロック消去中のシーケンサ制御を停止・再開させる関数のため、本関数の処理にはステータス・チェックは必要ありません。

2.　ブロックの消去処理(FSL_Erase)を再開させる場合は、ブロックの消去状態を確認するためのステータスチェックが必要になります。

6. 4. 1　ステータス・チェック・ユーザ・モード

ステータス・チェック・ユーザ・モードではRAM上からのBGO（バック・グラウンド・オペレーション）が可能です。以下に処理毎の動作例を示します。

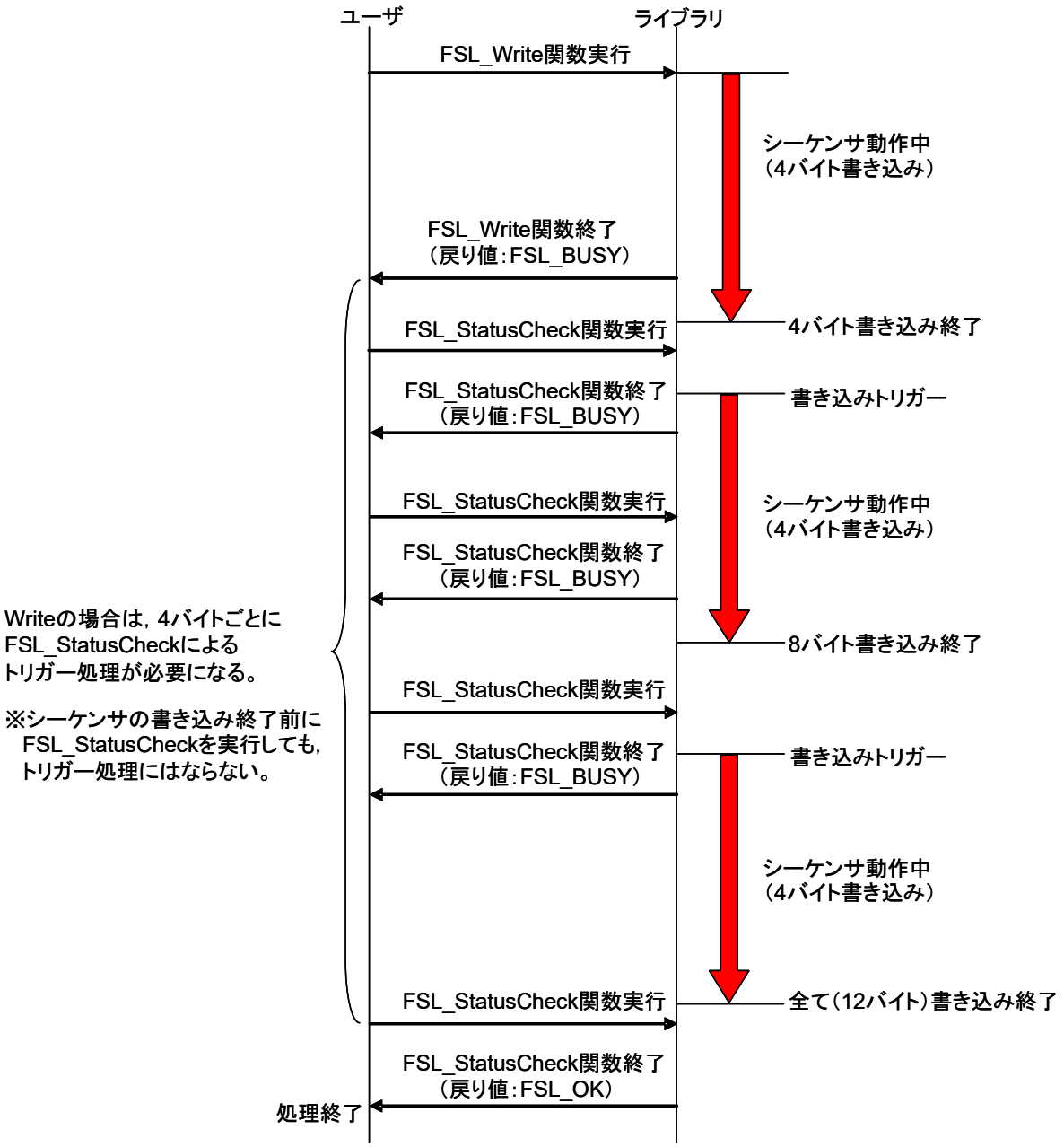


図 6－2　ステータス・チェック・ユーザ・モードの動作例1 (FSL_Write:12バイト書き込み時の例)

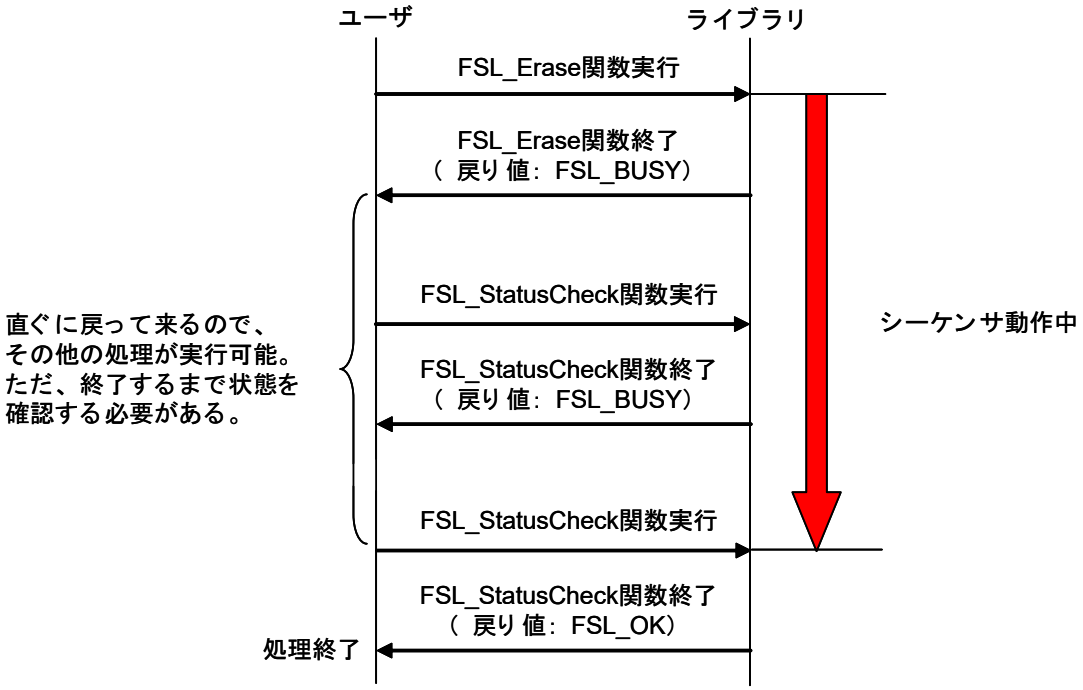


図 6−3　ステータス・チェック・ユーザ・モードの動作例2 (FSL_Write以外の例)

6. 5　フラッシュ・セルフ・プログラミングの一時停止

ステータス・チェック・ユーザ・モードでフラッシュ関数を実行している状態で、ブロック消去中にシーケンサの制御を一時停止させる必要がある場合、スタンバイ関数（FSL_StandBy）を使用する事で消去処理を一時中断させ、フラッシュ・セルフ・プログラミングを一時停止状態にすることが可能です。ブロック消去中以外の状態でスタンバイ関数を実行した場合は、事前に行った処理が終了するまで待ち、終了後に一時停止状態に移行します。

一時停止状態に移行するとコード・フラッシュ・メモリの制御ができなくなります。一時停止状態から復帰するためにはウェイク・アップ関数（FSL_WakeUp）を実行する必要があります。ブロック消去を中断していた場合は、一時停止状態を解除してブロック消去処理を再開します。その他の場合は一時停止状態の解除のみを行います。

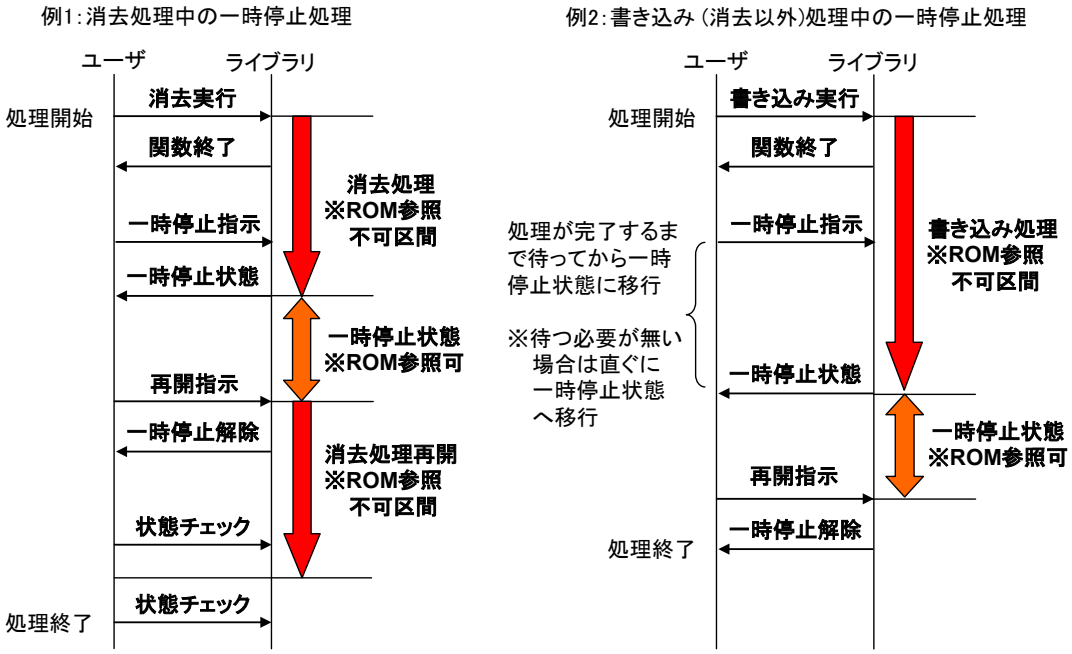


図 6−4　フラッシュ・セルフ・プログラミングの一時停止の例

表 6−5　スタンバイ関数の実行状態一覧

関数名	シーケンサ制御	スタンバイ関数を実行した場合の状態
FSL_Init	無し	使用不可
FSL_Open		
FSL_Close		
FSL_PrepareFunctions		
FSL_PrepareExtFunctions		
FSL_ChangeInterruptTable		
FSL_RestoreInterruptTable		
FSL_BlankCheck	有り	処理が終了するまで待ち、一時停止状態に移行
FSL_Erase		消去処理を一時中断し、一時停止状態に移行
FSL_IVerify		処理が終了するまで待ち、一時停止状態に移行
FSL_Write		
FSL_GetSecurityFlags	無し	使用不可
FSL_GetBootFlag		
FSL_GetSwapState		
FSL_GetBlockEndAddr		
FSL_GetFlashShieldWindow		
FSL_SwapBootCluster		
FSL_SwapActiveBootCluster	有り	処理が終了するまで待ち、一時停止状態に移行
FSL_InvertBootFlag		
FSL_SetBlockEraseProtectFlag		
FSL_SetWriteProtectFlag		
FSL_SetBootClusterProtectFlag		
FSL_SetFlashShieldWindow		
FSL_StatusCheck		使用不可
FSL_StandBy		
FSL_WakeUp		
FSL_ForceReset	無し	
FSL_GetVersionString		
フラッシュ関数 未実行状態		一時停止状態に移行

6.6　データ型、戻り値一覧

データ型は次のとおりです。

表 6−6　データ型一覧

定　　義	データ・タイプ	説　　明
fsl_u08	unsigned char	1バイト(8ビット)符号無し整数
fsl_u16	unsigned int	2バイト(16ビット)符号無し整数
fsl_u32	unsigned long int	4バイト(32ビット)符号無し整数

各戻り値の意味は次のとおりです。

表 6−7　戻り値一覧

定　　義	戻り値	説　　明
FSL_OK	0x00	正常終了
FSL_ERR_PARAMETER	0x05	パラメータ・エラー - 設定パラメータに誤りがある
FSL_ERR_PROTECTION	0x10	プロテクト・エラー - 対象領域はプロテクトされている
FSL_ERR_ERASE	0x1A	消去エラー - 対象領域の消去に失敗した
FSL_ERR_BLANKCHECK	0x1B	ブランク・チェック・エラー - 対象領域はブランク状態ではない
FSL_ERR_IVERIFY	0x1B	内部ベリファイ・エラー - 対象領域の内部ベリファイ処理中にエラーが発生した
FSL_ERR_WRITE	0x1C	書き込みエラー - 対象領域の書き込みに失敗した
FSL_ERR_FLOW	0x1F	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態
FSL_IDLE	0x30	アイドル状態 - フラッシュ・セルフ・プログラミングは実行されていない
FSL_SUSPEND	0x43	一時停止状態 - フラッシュ・セルフ・プログラミングは一時停止中
FSL_BUSY	0xFF	フラッシュ関数の実行開始、もしくは実行中 - フラッシュ関数の実行中

★　戻り値に使用する汎用レジスタは、RENESAS製 CA78K0R コンパイラとRENESAS製 CC-RL コンパイラ、およびLLVMコンパイラで異なります。

コンパイラ毎の戻り値と使用する汎用レジスタは、次のとおりです。

表 6−8　コンパイラ毎の戻り値一覧

	開発ツール	戻り値	
		C言語	アセンブリ言語
	RENESAS製 CA78K0R コンパイラ	fsl_u08	C (汎用レジスタ)
	RENESAS製 CC-RL コンパイラ	fsl_u08	A (汎用レジスタ)
★	LLVM コンパイラ	fsl_u08	A (汎用レジスタ)

6.7　フラッシュ関数の説明

フラッシュ関数について、次の形式で説明します。

フラッシュ関数名

【概　要】

この関数の機能概要を示します。

【書　式】

＜C言語＞

この関数をC言語で記述されたユーザ・プログラムから呼び出す際の書式を示します。

＜アセンブラ＞

この関数をアセンブリ言語で記述されたユーザ・プログラムから呼び出す際の書式を示します。

注　アセンブリ言語用のヘッダファイル(fsl.inc)はCA78K0Rコンパイラと CC-RLコンパイラ用にのみ用意しています。LLVMコンパイラでアセンブリ言語を使用する場合は、CC-RLコンパイラ用の「fsl.inc」を参考にFSL関数が返すステータスコードを定義したヘッダファイルを作成してください。

【事前設定】

この関数の事前設定を示します。

【機　能】

この関数の機能詳細と注意事項を示します。

【呼び出し後のレジスタ状態】

この関数を呼び出した後のレジスタの状態を示します。

【引　数】

この関数の引数を示します。

【戻り値】

この関数からの戻り値を示します。

【フロー】（FSL_SwapBootCluster()関数のみ）

この関数の内部フローを示します。

【操作例】（FSL_ChangeInterruptTable()，FSL_RestoreInterruptTable()関数のみ）

この関数を使用するための操作例を示します。

FSL_Init

【概　要】

フラッシュ・セルフ・プログラミング環境の初期化

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_Init( __far fsl_descriptor_t* descriptor_pstr)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_Init(const __far fsl_descriptor_t* descriptor_pstr)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_Init(const __far fsl_descriptor_t* descriptor_pstr)
                                    __attribute__ ((section ("FSL_FCD")))
```

＜アセンブラ＞

```
CALL !_FSL_Init または CALL !!_FSL_Init
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

・フラッシュ・セルフ・プログラミング・ライブラリ、もしくはデータ・フラッシュ・メモリを操作するためのプログラムやデータ・フラッシュ・ライブラリ、及びEEPROMエミュレーション・ライブラリが未実行、あるいは終了していること。

・高速オンチップ・オシレータを起動しておくこと。

【機　能】

・フラッシュ・セルフ・プログラミングで使用するセルフRAMの確保、および初期化を行います。セルフRAM[注1]が存在する場合は、フラッシュ・セルフ・プログラミングが終了するまでセルフRAMを使用しないで下さい。

・引数fsl_flash_voltage_u08でフラッシュ・セルフ・プログラミングのフラッシュ書き換えモード[注2]を定義します。

0x00　　：フルスピード・モード

上記以外　　：ワイド・ボルテージ・モード

・引数fsl_u08 fsl_frequency_u08にCPUの動作周波数を設定します。設定値はフラッシュ・セルフ・プログラミング・ライブラリ内部のタイミング・データの計算に使用します。[注3]

設定するCPUの動作周波数の値（fsl_frequency_u08）については、以下の点に注意してください。

- CPUの動作周波数を 4 MHz未満[注4]で使用する場合は、1 MHz, 2 MHz, 3 MHz を使用することができます。（1.5 MHz のように整数値でない周波数は使用できません）

  初期化関数で設定する動作周波数値も 1, 2, 3 の整数値を設定してください。

- CPUの動作周波数を 4 MHz以上[注4]で使用する場合は、小数点以下の数値を持つ周波数を使用することができます。ただし、初期化関数（FSL_Init）で設定する動作周波数は、小数点以下を切り上げた整数値を設定してください。（例：4.5MHz の場合は、初期化関数で 5 を設定してください）
- 高速オンチップ・オシレータの動作周波数ではありません。

・引数fsl_auto_status_check_u08でステータス・チェック・モードの設定を行います。[注5] ステータス・チェック・ユーザ・モードとステータス・チェック・インターナル・モードの違いについては、**2.1　ハードウェア環境**または**6.4　ステータス・チェック・モード**を参照してください。

0x00　　：ステータス・チェック・ユーザ・モード
上記以外　　：ステータス・チェック・インターナル・モード

注 1. セルフRAMに関してはインストーラに添付されている文書、「リリース・ノート」か、もしくは対象となるRL78マイクロコントローラのユーザーズマニュアルを参照してください。

2. フラッシュ書き換えモードの詳細については、対象となるRL78マイクロコントローラのユーザーズマニュアルを参照してください。

3. フラッシュ・セルフ・プログラミング・ライブラリ内部のタイミング計算に使用されるために必要なパラメータとなります。この設定によってCPUの動作周波数が変わる事はありません。

4. CPUの動作周波数の範囲については対象となるRL78マイクロコントローラのユーザーズマニュアルを参照してください。

5. FSL_RCDセグメントをROM上に配置する場合は、必ずステータス・チェック・インターナル・モードで使用してください。

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

引数の定義

引　　数	説　　明
__far fsl_descriptor_t* descriptor_pstr	フラッシュ・セルフ・プログラミング・ライブラリ Type01の初期設定値 （フラッシュ書き換えモード、CPUの動作周波数、ステータス・チェック・モード）

fsl_descriptor_t の定義

開発ツール	C言語（構造体の定義）	アセンブリ言語（変数の定義例）
RENESAS製 CA78K0R コンパイラ	typedef struct { fsl_u08 fsl_flash_voltage_u08; fsl_u08 fsl_frequency_u08; fsl_u08 fsl_auto_status_check_u08; } fsl_descriptor_t;	fsl_descriptor_str: DB fsl_flash_voltage_u08 DB fsl_frequency_u08 DB fsl_auto_status_check_u08
RENESAS製 CC-RL コンパイラ	typedef struct { fsl_u08 fsl_flash_voltage_u08; fsl_u08 fsl_frequency_u08; fsl_u08 fsl_auto_status_check_u08; } fsl_descriptor_t;	fsl_descriptor_str: .DB fsl_flash_voltage_u08 .DB fsl_frequency_u08 .DB fsl_auto_status_check_u08
★ LLVM コンパイラ	typedef struct { fsl_u08 fsl_flash_voltage_u08; fsl_u08 fsl_frequency_u08; fsl_u08 fsl_auto_status_check_u08; } fsl_descriptor_t;	コンパイラの仕様をご確認ください。

fsl_descriptor_t のパラメータ内容

引　数	説　明
fsl_u08 fsl_flash_voltage_u08	フラッシュ書き換えモードの設定
fsl_u08 fsl_frequency_u08	フラッシュ・セルフ・プログラミング実行中のCPU周波数
fsl_u08 fsl_auto_status_check_u08	ステータス・チェック・モードの設定

引数の設定内容

開発ツール	引数型／レジスタ	
	C言語	アセンブリ言語
RENESAS製 CA78K0R コンパイラ	__far fsl_descriptor_t *descriptor_pstr	AX(0-15), C(16-23) : 構造体引数の先頭アドレス（24bit）
RENESAS製 CC-RL コンパイラ	const __far fsl_descriptor_t *descriptor_pstr	DE(0-15), A(16-23) : 構造体引数の先頭アドレス（24bit）
★ LLVM コンパイラ	const __far fsl_descriptor_t *descriptor_pstr	DE(0-15), A(16-23) : 構造体引数の先頭アドレス（24bit）

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了 - 初期設定が完了した状態
0x05(FSL_ ERR_PARAMETER)	パラメータ・エラー - CPUの動作周波数の値が設定可能範囲外 - 高速オンチップ・オシレータが起動していない

FSL_Open

【概　要】

フラッシュ・セルフ・プログラミングの開始宣言（フラッシュ環境の開始）

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
void FSL_Open(void)
```

RENESAS製 CC-RL コンパイラ:

```
void __far FSL_Open(void)
```

★ LLVM コンパイラ:

```
void __far FSL_Open(void) __attribute__ ((section ("FSL_FCD")))
```

＜アセンブラ＞

```
CALL !_FSL_Open または CALL !!_FSL_Open
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させてください。

【機　能】

フラッシュ・セルフ・プログラミングの開始宣言（フラッシュ環境の開始）を行います。フラッシュ・セルフ・プログラミング操作の最初に、この関数を呼び出してください。

【呼び出し後のレジスタ状態】

レジスタは破壊されません。

【引　数】

なし

【戻り値】

なし

FSL_Close

【概　要】

フラッシュ・セルフ・プログラミングの終了宣言（フラッシュ環境の終了）

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
void FSL_Close(void)
```

RENESAS製 CC-RL コンパイラ:

```
void __far FSL_Close(void)
```

★ LLVM コンパイラ:

```
void __far FSL_Close(void) __attribute__ ((section ("FSL_FCD")))
```

＜アセンブラ＞

```
CALL !_FSL_Close または CALL !!_FSL_Close
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数を実行完了させてください。

【機　能】

フラッシュ・セルフ・プログラミングの終了宣言(フラッシュ環境の終了)を行います。コード・フラッシュ・メモリへの書き込み操作を終了し、通常動作モードに戻します。

【呼び出し後のレジスタ状態】

レジスタは破壊されません。

【引　数】

なし

【戻り値】

なし

FSL_PrepareFunctions

【概　要】

RAM実行が必要なフラッシュ関数(標準書き換え関数)を使用できるように準備

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
void FSL_PrepareFunctions( void )
```

RENESAS製 CC-RL コンパイラ:

```
void __far FSL_PrepareFunctions( void )
```

★ LLVM コンパイラ:

```
void __far FSL_PrepareFunctions(void) __attribute__ ((section ("FSL_FCD")))
```

＜アセンブラ＞

```
CALL !_FSL_PrepareFunctions または CALL !!_FSL_PrepareFunctions
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数を実行完了させてください。

【機　能】

以下の関数を使用できるように準備します。

- FSL_BlankCheck
- FSL_Erase
- FSL_Write
- FSL_IVerify
- FSL_StatusCheck
- FSL_StandBy
- FSL_WakeUp

【呼び出し後のレジスタ状態】

レジスタは破壊されません。

【引　数】

なし

【戻り値】

なし

FSL_PrepareExtFunctions

【概　要】

RAM実行が必要なフラッシュ関数(拡張機能関数)を使用できるようにします

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
void FSL_PrepareExtFunctions(void)
```

RENESAS製 CC-RL コンパイラ:

```
void __far FSL_PrepareExtFunctions(void)
```

★ LLVM コンパイラ:

```
void __far FSL_PrepareExtFunctions(void) __attribute__ ((section ("FSL_FCD")))
```

＜アセンブラ＞

```
CALL !_FSL_PrepareExtFunctions または CALL !!_FSL_PrepareExtFunctions
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数、およびFSL_PrepareFunctions関数を実行完了させてください。

【機　能】

以下の関数を使用できるように準備します。

- FSL_SwapBootCluster
- FSL_SwapActiveBootCluster
- FSL_InvertBootFlag
- FSL_SetBlockEraseProtectFlag
- FSL_SetWriteProtectFlag
- FSL_SetBootClusterProtectFlag
- FSL_SetFlashShieldWindow

【呼び出し後のレジスタ状態】

レジスタは破壊されません。

【引　数】

なし

【戻り値】

なし

FSL_ChangeInterruptTable

【概　要】

全ての割り込みの飛び先をRAMの指定アドレス上へ入るように変更

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
void FSL_ChangeInterruptTable(fsl_u16 fsl_interrupt_destination_u16)
```

RENESAS製 CC-RL コンパイラ:

```
void __far FSL_ChangeInterruptTable(fsl_u16 fsl_interrupt_destination_u16)
```

★ LLVM コンパイラ:

```
void __far FSL_ChangeInterruptTable(fsl_u16 fsl_interrupt_destination_u16)
                                    __attribute__ ((section ("FSL_FCD")))
```

＜アセンブラ＞

```
CALL !_FSL_ChangeInterruptTable または CALL !!_FSL_ChangeInterruptTable
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

なし

【機　能】

全ての割り込み機能の飛び先をRAMの指定アドレス上へ変更します。この関数実行後は割り込みが入っても、割り込みベクタ・テーブルには飛ばず、全て本関数で指定されたRAMのアドレス上へ入るようになります。

注意 1. 割り込みの種類はユーザ側で割り込みフラグを確認して判断する必要があります。また、本関数実行後は割り込みの種類をユーザ側で判断する必要があるため、割り込みフラグは自動的にクリアされません。割り込み種類の判断後、ユーザ側でクリア処理を行う必要があります。

2. 指定するRAMのアドレスをフラッシュ・セルフ・プログラミング実行中に使用制限のある領域に指定しないで下さい。フラッシュ関数が正常に動作できなくなる恐れがあります。

3. 割り込み変更先をROM側に設定することは出来ません。(FxxxxHのアドレス範囲のみ)

4. この関数で割り込み先を変更した場合、FSL_RestoreInterruptTable()関数で割り込み先を復元するか、リセット等を実行するまで、フラッシュ・セルフ・プログラミング終了後も割り込み先は、変更されたままになります。

5. この関数で割り込み先をRAMに変更する場合、開始から完了までの間は割り込み禁止にしてください。

【呼び出し後のレジスタ状態】

レジスタは破壊されません。

【引　数】

引数の定義

引　　数	説　　明
fsl_u16　fsl_interrupt_destination_u16	割り込み先のRAMアドレス（下位16bit：FxxxxH ）※上位bitは必要なし

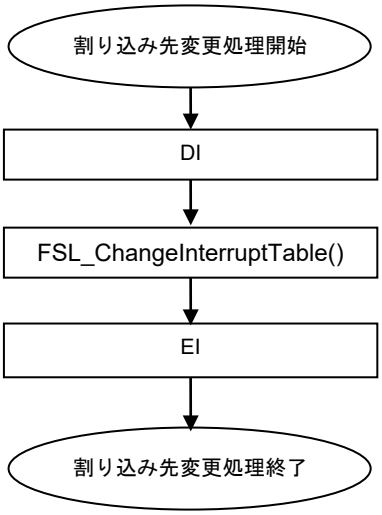
引数の設定内容

開発ツール	引数型／レジスタ	
	C言語	アセンブリ言語
RENESAS製 CA78K0R コンパイラ	fsl_u16 fsl_interrupt_destination_u16	AX(0-15)：RAMアドレス（下位16bit）
RENESAS製 CC-RL コンパイラ	fsl_u16 fsl_interrupt_destination_u16	AX(0-15)：RAMアドレス（下位16bit）
★ LLVM コンパイラ	fsl_u16 fsl_interrupt_destination_u16	AX(0-15)：RAMアドレス（下位16bit）

【戻り値】

なし

【操作例】



FSL_RestoreInterruptTable

【概　要】

RAMに変更された割り込みの飛び先を標準の割り込みベクタ・テーブルに戻す

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
void FSL_RestoreInterruptTable( void )
```

RENESAS製 CC-RL コンパイラ:

```
void __far FSL_RestoreInterruptTable( void )
```

★ LLVM コンパイラ:

```
void __far FSL_RestoreInterruptTable(void) __attribute__ ((section ("FSL_FCD")))
```

＜アセンブラ＞

```
CALL !_FSL_RestoreInterruptTable または CALL !!_FSL_RestoreInterruptTable
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

なし

【機　能】

RAMに変更された割り込みの飛び先を標準の割り込みベクタ・テーブルに戻します。

注意　1. FSL_ChangeInterruptTable()関数で割り込み先を変更している場合、この関数で割り込み先を復元しないと、リセット等を実行するまでフラッシュ・セルフ・プログラミング終了後も割り込み先は変更されたままになります。

2. この関数で割り込み先を標準の割り込みベクタに変更する場合、開始から完了までの間は割り込み禁止にしてください。

【呼び出し後のレジスタ状態】

レジスタは破壊されません。

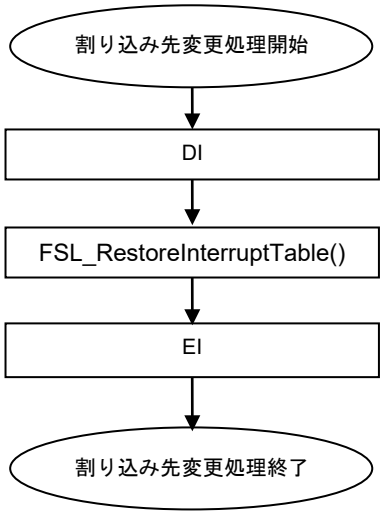
【引　数】

なし

【戻り値】

なし

【操作例】


割り込み先変更処理開始
DI
FSL_RestoreInterruptTable()
EI
割り込み先変更処理終了

FSL_BlankCheck

【概　要】

指定ブロックのブランク・チェック

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_BlankCheck(fsl_u16 block_u16)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_BlankCheck(fsl_u16 block_u16)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_BlankCheck(fsl_u16 block_u16)
                                        __attribute__ ((section ("FSL_RCD")))
```

＜アセンブラ＞

```
CALL !_FSL_BlankCheck または CALL !!_FSL_BlankCheck
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数、およびFSL_PrepareFunctions関数を実行完了させてください。また処理が終了する前に割り込みを受け付ける必要がある場合は、FSL_ChangeInterruptTable関数を使用して、割り込み先をRAMに変更してください。

【機　能】

指定したブロックのコード・フラッシュ・メモリが消去レベルにあることを確認します。
（データ・リードによる0xFFの確認では、消去レベルの確認は行えません）
エラーの場合はFSL_Erase関数を実行してください。
なお、FSL_Erase関数の実行が正常終了した場合には、ブランク・チェックは不要です。
指定したブロック番号が存在しない場合、パラメータ・エラー（0x05）を返します。

注意　以下の(1)、(2)の両方を満たす場合、本関数を内蔵ROM上に配置して使用することが可能です。

(1) FSL_Init関数のステータス・チェック・モードの指定をステータス・チェック・インターナル・モードに設定すること。

(2) 本関数の処理が終了するまでの間、"割り込みを使用しない"、または"内蔵ROM上での割り込みを禁止に設定"すること。(RAM上での割り込み受け付けは可能)

備考 1.　FSL_BlankCheck関数は、コード・フラッシュ・メモリのセルが十分なマージンを持って消去レベルを満たしているかを確認するものです。ブランク・チェック・エラーは、コード・フラッシュ・メモリに問題があることを示すものではありませんが、ブランク・チェック・エラー後は、消去処理を行ってから書き込みをしてください。

2.　ブランク・チェックは1ブロックのみ行うので、複数のブロックをブランク・チェックする場合は、この関数を複数回呼び出してください。

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

引数の定義

引　　数	説　　明
fsl_u16　block_u16	ブランク・チェックを行うブロックのブロック番号

引数の設定内容

開発ツール	引数型／レジスタ	
	C言語	アセンブリ言語
RENESAS製 CA78K0R コンパイラ	fsl_u16 block_u16	AX(0-15)：ブロック番号（16bit）
RENESAS製 CC-RL コンパイラ	fsl_u16 block_u16	AX(0-15)：ブロック番号（16bit）
★ LLVM コンパイラ	fsl_u16 block_u16	AX(0-15)：ブロック番号（16bit）

【戻り値】

状　　態	説　　明
0x00(FSL_OK)	正常終了[注1] - 指定したブロックはブランク状態
0x05(FSL_ERR_PARAMETER)	パラメータ・エラー - ブロック番号の指定が設定可能範囲外
0x1B(FSL_ERR_BLANKCHECK)	ブランク・チェック・エラー[注1] - 指定したブロックがブランク状態でない
0x1F(FSL_ERR_FLOW)	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない[注2] - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態[注2]
0xFF(FSL_BUSY)	本関数機能の実行開始[注2] - 本関数の実行が開始された (実行状態をFSL_StatusCheck関数により確認してください)

注 1.　ステータス・チェック・インターナル・モード時のみ

2.　ステータス・チェック・ユーザ・モード時のみ

FSL_Erase

【概　要】

指定ブロックの消去

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_Erase(fsl_u16 block_u16)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_Erase(fsl_u16 block_u16)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_Erase(fsl_u16 block_u16) __attribute__ ((section ("FSL_RCD")))
```

＜アセンブラ＞

```
CALL !_FSL_Erase または CALL !!_FSL_Erase
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数、およびFSL_PrepareFunctions関数を実行完了させてください。また処理が終了する前に割り込みを受け付ける必要がある場合は、FSL_ChangeInterruptTable関数を使用して、割り込み先をRAMに変更してください。

【機　能】

指定したブロックのコード・フラッシュ・メモリの内容を消去（0xFF）します。

注意　以下の(1)、(2)の両方を満たす場合、本関数を内蔵ROM上に配置して使用することが可能です。

(1) FSL_Init関数のステータス・チェック・モードの指定をステータス・チェック・インターナル・モードに設定すること。

(2) 本関数の処理が終了するまでの間、"割り込みを使用しない"、または"内蔵ROM上での割り込みを禁止に設定"すること。(RAM上での割り込み受け付けは可能)

備考　消去は1ブロックのみ行うので、複数のブロックを消去する場合は、この関数を複数回呼び出してください。

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

引数の定義

引　数	説　明
fsl_u16　block_u16	消去するブロックのブロック番号

引数の設定内容

開発ツール	引数型／レジスタ	
	C言語	アセンブリ言語
RENESAS製 CA78K0R コンパイラ	fsl_u16 block_u16	AX(0-15)：ブロック番号（16bit）
RENESAS製 CC-RL コンパイラ	fsl_u16 block_u16	AX(0-15)：ブロック番号（16bit）
★ LLVM コンパイラ	fsl_u16 block_u16	AX(0-15)：ブロック番号（16bit）

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了[注1]
0x05(FSL_ERR_PARAMETER)	パラメータ・エラー - ブロック番号の指定が設定可能範囲外
0x10(FSL_ERR_PROTECTION)	プロテクト・エラー - 指定ブロックがブート領域に含まれており、ブート領域書き換え許可フラグが禁止に設定されている - 指定したブロックがFSW設定領域外
0x1A(FSL_ERR_ERASE)	消去エラー[注1] - 消去処理中にエラーが発生した
0x1F(FSL_ERR_FLOW)	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない[注2] - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態[注2]
0xFF(FSL_BUSY)	本関数機能の実行開始[注2] - 本関数の実行が開始された (実行状態をFSL_StatusCheck関数により確認してください)

注 1.　ステータス・チェック・インターナル・モード時のみ
　 2.　ステータス・チェック・ユーザ・モード時のみ

FSL_IVerify

【概　要】

指定ブロックのベリファイ（内部ベリファイ）

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_IVerify(fsl_u16 block_u16)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_IVerify(fsl_u16 block_u16)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_IVerify(fsl_u16 block_u16) __attribute__ ((section ("FSL_RCD")))
```

＜アセンブラ＞

```
CALL !_FSL_IVerify または CALL !!_FSL_IVerify
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数、およびFSL_PrepareFunctions関数を実行完了させてください。また処理が終了する前に割り込みを受け付ける必要がある場合は、FSL_ChangeInterruptTable関数を使用して、割り込み先をRAMに変更してください。

【機　能】

指定されたブロックへの書き込みレベルを確認するためのベリファイを行います。

このベリファイは、指定されたブロックのコード・フラッシュ・メモリに書き込まれたデータが消去レベル（データ"1"）／書き込みレベル（データ"0"）にあることを確認するものです。

エラーの場合はFSL_Eraseを実行後、再度FSL_Writeにより書き込みを実行してください。

指定したブロック番号が存在しない場合、パラメータ・エラー（0x05）を返します。

注意 1.　データ書き込み後に、書き込みを行った範囲を含むブロックのベリファイ（内部ベリファイ）を実行しない場合、書き込んだデータは保証されません。

2.　内部ベリファイ・エラー後に、再度、データの消去→書き込み→内部ベリファイを実行し、正常終了した場合には、そのデバイスは正常であると判断します。

3.　以下の(1)、(2)の両方を満たす場合、本関数を内蔵ROM上に配置して使用することが可能です。

(1)　FSL_Init関数のステータス・チェック・モードの指定をステータス・チェック・インターナル・モードに設定すること。

(2)　本関数の処理が終了するまでの間、"割り込みを使用しない"、または"内蔵ROM上での割り込みを禁止に設定"すること。(RAM上での割り込み受け付けは可能)

備考　ベリファイは1ブロックのみ行うので、複数のブロックをベリファイする場合は、この関数を複数回呼び出してください。

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

引数の定義

引　　数	説　　明
fsl_u16　block_u16	ベリファイするブロック番号

引数の設定内容

開発ツール	引数型／レジスタ	
	C言語	アセンブリ言語
RENESAS製 CA78K0R コンパイラ	fsl_u16 block_u16	AX(0-15)：ブロック番号（16bit）
RENESAS製 CC-RL コンパイラ	fsl_u16 block_u16	AX(0-15)：ブロック番号（16bit）
★ LLVM コンパイラ	fsl_u16 block_u16	AX(0-15)：ブロック番号（16bit）

【戻り値】

状　　態	説　　明
0x00(FSL_OK)	正常終了[注1]
0x05(FSL_ERR_PARAMETER)	パラメータ・エラー - ブロック番号の指定が設定可能範囲外
0x1B(FSL_ERR_IVERIFY)	ベリファイ（内部ベリファイ）エラー[注1] - ベリファイ（内部ベリファイ）処理中にエラーが発生した
0x1F(FSL_ERR_FLOW)	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない[注2] - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態[注2]
0xFF(FSL_BUSY)	本関数機能の実行開始[注2] - 本関数の実行が開始された（実行状態をFSL_StatusCheck関数により確認してください）

注 1.　ステータス・チェック・インターナル・モード時のみ
　 2.　ステータス・チェック・ユーザ・モード時のみ

FSL_Write

【概　要】

指定アドレスに対する1-64ワード・データの書き込み (1ワード = 4バイト)

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_Write (__near fsl_write_t* write_pstr)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_Write (__near fsl_write_t* write_pstr)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_Write(__near fsl_write_t* write_pstr)
                                        __attribute__ ((section ("FSL_RCD")))
```

＜アセンブラ＞

```
CALL !_FSL_Write または CALL !!_FSL_Write
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

・この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数、およびFSL_PrepareFunctions関数を実行完了させてください。また処理が終了する前に割り込みを受け付ける必要がある場合は、FSL_ChangeInterruptTable関数を使用して、割り込み先をRAMに変更してください。

・この関数を呼び出す前に、コード・フラッシュ・メモリに書き込むデータをデータ・バッファに格納してください。

【機　能】

指定されたアドレスのコード・フラッシュ・メモリへの書き込みを行います。

書き込みを実施したブロックに対しては、書き込み後、必ずFSL_IVerifyを実行してください。

FSL_Write関数は、消去されたブロックに対してのみ実行してください。

一度に最大256バイト（4バイト単位）のデータの書き込みを行うことができます。

次の場合（指定されたワード数、アドレスが設定可能範囲外の場合）、パラメータ・エラー（0x05）を返します。

ワード数チェック

・0ワードの場合

・65ワード以上の場合

アドレス・チェック

・先頭アドレスから4バイト単位の設定になっていない場合

・書き込み終了アドレスがコード・フラッシュ・メモリの最終アドレスを越えた場合

注意 1.　データの書き込み後、書き込みを行った範囲を含むブロックのベリファイ（内部ベリファイ）を実行してください。実行しない場合、書き込んだデータは保証されません。

2.　以下の(1)、(2)の両方を満たす場合、本関数を内蔵ROM上に配置して使用することが可能です。

(1) FSL_Init関数のステータス・チェック・モードの指定をステータス・チェック・インターナル・モードに設定すること。

(2) 本関数の処理が終了するまでの間、"割り込みを使用しない"、または"内蔵ROM上での割り込みを禁止に設定"すること。(RAM上での割り込み受け付けは可能)

備考　256 バイトを超えるデータを書き込む場合は、この関数を複数回呼び出してください。

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

引数の定義

引　　数	説　　明
__near fsl_write_t* write_pstr	書き込みデータ設定値 （データ・バッファのアドレス、書き込み先アドレス、書き込みサイズ）

fsl_write_t の定義

★ 開発ツール	C言語（構造体の定義）	アセンブリ言語（変数の定義例）
RENESAS製 CA78K0R コンパイラ	typedef struct { fsl_u08 __near *fsl_data_buffer_p_u08; fsl_u32 fsl_destination_address_u32; fsl_u08 fsl_word_count_u08; } fsl_write_t;	fsl_write_str : fsl_data_buffer_p_u08: DS 2 fsl_destination_address_u32: DS 4 fsl_word_count_u08: DS 1
RENESAS製 CC-RL コンパイラ	typedef struct { fsl_u08 __near *fsl_data_buffer_p_u08; fsl_u32 fsl_destination_address_u32; fsl_u08 fsl_word_count_u08; } fsl_write_t;	fsl_write_str : fsl_data_buffer_p_u08: .DS 2 fsl_destination_address_u32: .DS 4 fsl_word_count_u08: .DS 1
LLVM コンパイラ	typedef struct { fsl_u08 __near *fsl_data_buffer_p_u08; fsl_u32 fsl_destination_address_u32; fsl_u08 fsl_word_count_u08; } fsl_write_t;	コンパイラの仕様をご確認ください。

fsl_write_t のパラメータ内容

引　　数	説　　明
fsl_u08 __near *fsl_data_buffer_p_u08	書き込むデータが入力されているバッファ領域の先頭アドレス（16bit）
fsl_u32 fsl_destination_address_u32	書き込み先の先頭アドレス（32bit）
fsl_u08 fsl_word_count_u08	書き込むデータ数（1-64：ワード単位）

引数の設定内容

	開発ツール	引数型／レジスタ	
		C言語	アセンブリ言語
	RENESAS製 CA78K0R コンパイラ	__near fsl_write_t* write_pstr	AX(0-15)：変数の先頭アドレス（16bit）
	RENESAS製 CC-RL コンパイラ	__near fsl_write_t* write_pstr	AX(0-15)：変数の先頭アドレス（16bit）
★	LLVM コンパイラ	__near fsl_write_t* write_pstr	AX(0-15)：変数の先頭アドレス（16bit）

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了[注1]
0x05(FSL_ERR_PARAMETER)	パラメータ・エラー - 開始アドレスが1ワード（4バイト）の倍数でない - 書き込みデータ数が0 - 書き込みデータ数が64ワードを越えている - 書き込み終了アドレス（開始アドレス＋（書き込みデータ数×4バイト）） がコード・フラッシュ・メモリ領域を越えている
0x10(FSL_ERR_PROTECTION)	プロテクト・エラー - 指定範囲にブート領域が含まれており、ブート領域書き換え許可フラグが 禁止に設定されている - 指定したブロックがFSW設定領域外
0x1C(FSL_ERR_WRITE)	書き込みエラー[注1] - 書き込み処理中にエラーが発生した
0x1F(FSL_ERR_FLOW)	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない[注2] - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態[注2]
0xFF(FSL_BUSY)	本関数機能の実行開始[注2] - 本関数の実行が開始された (実行状態をFSL_StatusCheck関数により確認してください)

注 1.　ステータス・チェック・インターナル・モード時のみ
　 2.　ステータス・チェック・ユーザ・モード時のみ

FSL_GetSecurityFlags

【概　要】

セキュリティ情報の取得

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_GetSecurityFlags(fsl_u08 __near *data_destination_pu08)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_GetSecurityFlags(fsl_u08 __near *data_destination_pu08)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_GetSecurityFlags(fsl_u08 __near *data_destination_pu08)
                                        __attribute__ ((section ("FSL_FECD")))
```

＜アセンブラ＞

```
CALL !_FSL_GetSecurityFlags または CALL !!_FSL_GetSecurityFlags
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、およびFSL_Open関数を実行完了させてください。

【機　能】

セキュリティ･フラグの情報を取得し、引数で指定されたデータ格納用のバッファに値を入力します。

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

引数の定義

引　数	説　明
fsl_u08 __near *data_destination_pu08	データ格納用バッファ - 1バイトの専用のデータ・バッファを確保してください

引数の設定内容

開発ツール	引数型／レジスタ	
	C言語	アセンブリ言語
RENESAS製 CA78K0R コンパイラ	__near *data_destination_pu08	AX(0-15)：バッファの先頭アドレス（16bit）
RENESAS製 CC-RL コンパイラ	__near *data_destination_pu08	AX(0-15)：バッファの先頭アドレス（16bit）
★ LLVM コンパイラ	__near *data_destination_pu08	AX(0-15)：バッファの先頭アドレス（16bit）

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了
0x1F(FSL_ERR_FLOW)	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない注 - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態注

注　ステータス・チェック・ユーザ・モード時のみ

セキュリティ・ビット情報

セキュリティ・ビットは引数で渡したデータ格納用バッファ(data_destination_pu08)に書き込まれます。

data_destination_pu08	説　明
ビット1 : 0b000000X0	ブート領域書き換え禁止フラグ　（0：禁止、1：許可）
ビット2 : 0b00000X00	ブロック消去禁止フラグ　（0：禁止、1：許可）
ビット4 : 0b000X0000	書き込み禁止フラグ　（0：禁止、1：許可）
その他のビット	1

FSL_GetBootFlag

【概　要】

ブート・フラグ情報の取得

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_GetBootFlag(fsl_u08 __near *data_destination_pu08)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_GetBootFlag(fsl_u08 __near *data_destination_pu08)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_GetBootFlag(fsl_u08 __near *data_destination_pu08)
                                        __attribute__ ((section ("FSL_FECD")))
```

＜アセンブラ＞

```
CALL !_FSL_GetBootFlag または CALL !!_FSL_GetBootFlag
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数を実行完了させてください。

【機　能】

ブート･クラスタ･フラグの情報を取得し、引数で指定されたデータ格納用のバッファに値を入力します。

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

引数の定義

引　数	説　明
fsl_u08 __near *data_destination_pu08	データ格納用バッファ - 1バイトの専用のデータ・バッファを確保してください

引数の設定内容

開発ツール	引数型／レジスタ	
	C言語	アセンブリ言語
RENESAS製 CA78K0R コンパイラ	__near *data_destination_pu08	AX(0-15)：バッファの先頭アドレス（16bit）
RENESAS製 CC-RL コンパイラ	__near *data_destination_pu08	AX(0-15)：バッファの先頭アドレス（16bit）
★ LLVM コンパイラ	__near *data_destination_pu08	AX(0-15)：バッファの先頭アドレス（16bit）

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了
0x1F(FSL_ERR_FLOW)	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない注 - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態注

注　　ステータス・チェック・ユーザ・モード時のみ

ブート・フラグ情報

ブート・フラグは引数で渡したデータ格納用バッファ(data_destination_pu08)に書き込まれます。

data_destination_pu08	説　明
0x00	リセット後、ブート・クラスタ0をブート領域(0000Hから)として起動する
0x01	リセット後、ブート・クラスタ1をブート領域(0000Hから)として起動する

備考　リセット前のブート領域のスワップ状態についてはFSL_GetSwapState関数の項を参照してください。
また、ブート領域のクラスタサイズは、デバイスにより異なります。デバイスのユーザーズマニュアルをご確認ください。
（例：RL78/G13のブート領域サイズは1クラスタ4KB、RL78/F13は8KB）

FSL_GetSwapState

【概　要】

スワップ状態の取得

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_GetSwapState(fsl_u08 __near *data_destination_pu08)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_GetSwapState(fsl_u08 __near *data_destination_pu08)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_GetSwapState(fsl_u08 __near *data_destination_pu08)
                                        __attribute__ ((section ("FSL_FECD")))
```

＜アセンブラ＞

```
CALL !_FSL_GetSwapState または CALL !!_FSL_GetSwapState
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数を実行完了させてください。

【機　能】

現在のブート･クラスタのスワップ状態を取得し、引数で指定されたデータ格納用のバッファに値を入力します。

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

引数の定義

引　　数	説　　明
fsl_u08 __near *data_destination_pu08	データ格納用バッファ - 1バイトの専用のデータ・バッファを確保してください

引数の設定内容

	開発ツール	引数型／レジスタ	
		C言語	アセンブリ言語
	RENESAS製 CA78K0R コンパイラ	__near *data_destination_pu08	AX(0-15)：バッファの先頭アドレス（16bit）
	RENESAS製 CC-RL コンパイラ	__near *data_destination_pu08	AX(0-15)：バッファの先頭アドレス（16bit）
★	LLVM コンパイラ	__near *data_destination_pu08	AX(0-15)：バッファの先頭アドレス（16bit）

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了
0x1F(FSL_ERR_FLOW)	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない注 - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態注

注　　ステータス・チェック・ユーザ・モード時のみ

ブート・スワップ状態

ブート・フラグは引数で渡したデータ格納用バッファ(data_destination_pu08)に書き込まれます。

data_destination_pu08	説　明
0x00	現在のブート領域(0000Hからの領域)はブート・クラスタ0
0x01	現在のブート領域(0000Hからの領域)はブート・クラスタ1

備考　リセット後のブート領域の状態についてはFSL_GetBootFlag関数の項を参照してください。
また、ブート領域のクラスタサイズは、デバイスにより異なります。デバイスのユーザーズマニュアルをご確認ください。
（例：RL78/G13のブート領域サイズは1クラスタ4KB、RL78/F13は8KB）

FSL_GetBlockEndAddr

【概　要】

指定したブロックの最終アドレスの取得

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_GetBlockEndAddr(__near fsl_getblockendaddr_t* getblockendaddr_pstr)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_GetBlockEndAddr(__near fsl_getblockendaddr_t*
                                                    getblockendaddr_pstr)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_GetBlockEndAddr(__near fsl_getblockendaddr_t*
                getblockendaddr_pstr) __attribute__ ((section ("FSL_FECD")))
```

＜アセンブラ＞

```
CALL !_FSL_GetBlockEndAddr または CALL !!_FSL_GetBlockEndAddr
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数を実行完了させてください。

【機　能】

引数で指定されたブロックの最終アドレスを取得し、データ格納用のバッファに値を入力します。

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

引数の定義

引　　数	説　　明
__near fsl_getblockendaddr_t* getblockendaddr_pstr	該当ブロックの最終アドレス取得用構造体へのポインタ

fsl_getblockendaddr_tの定義

★

開発ツール	C言語（構造体の定義）	アセンブリ言語（変数の定義例）
RENESAS製 CA78K0R コンパイラ	typedef struct { fsl_u32 fsl_destination_address_u32; fsl_u16 fsl_block_u16 } fsl_getblockendaddr_t;	fsl_getblockendaddr_str: fsl_destination_address_u32: DS 4 fsl_block_u16: DS 2
RENESAS製 CC-RL コンパイラ	typedef struct { fsl_u32 fsl_destination_address_u32; fsl_u16 fsl_block_u16 } fsl_getblockendaddr_t;	fsl_getblockendaddr_str: fsl_destination_address_u32: .DS 4 fsl_block_u16: .DS 2
LLVM コンパイラ	typedef struct { fsl_u32 fsl_destination_address_u32; fsl_u16 fsl_block_u16 } fsl_getblockendaddr_t;	コンパイラの仕様をご確認ください。

fsl_getblockendaddr_t のパラメータ内容

引　数	説　明
fsl_u32 fsl_destination_address_u32	最終アドレス格納用バッファ：出力値 - 4バイトの専用のデータ・バッファです。
fsl_u16 fsl_block_u16	ブロック番号：入力値

引数の設定内容

開発ツール	引数型／レジスタ	
	C言語	アセンブリ言語
RENESAS製 CA78K0R コンパイラ	__near fsl_getblockendaddr_t* getblockendaddr_pstr	AX(0-15)：変数の先頭アドレス（16bit）
RENESAS製 CC-RL コンパイラ	__near fsl_getblockendaddr_t* getblockendaddr_pstr	AX(0-15)：変数の先頭アドレス（16bit）
★ LLVM コンパイラ	__near fsl_getblockendaddr_t* getblockendaddr_pstr	AX(0-15)：変数の先頭アドレス（16bit）

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了
0x05(FSL_ ERR_PARAMETER)	パラメータ・エラー - ブロック番号の指定が指定可能範囲外
0x1F(FSL_ERR_FLOW)	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない注 - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態注

注　　ステータス・チェック・ユーザ・モード時のみ

FSL_GetFlashShieldWindow

【概　要】

フラッシュ・シールド・ウインドウの開始ブロック番号と終了ブロック番号の取得

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_GetFlashShieldWindow(__near fsl_fsw_t* getfsw_pstr);
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_GetFlashShieldWindow(__near fsl_fsw_t* getfsw_pstr);
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_GetFlashShieldWindow(__near fsl_fsw_t* getfsw_pstr)
                                        __attribute__ ((section ("FSL_FECD")))
```

＜アセンブラ＞

```
CALL !_FSL_GetFlashShieldWindow または CALL !!_FSL_GetFlashShieldWindow
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数を実行完了させてください。

【機　能】

フラッシュ･シールド・ウインドウの開始ブロックと終了ブロックを取得し、それぞれ引数で指定されたデータ・格納用バッファ、fsl_start_block_u16(開始ブロック)と、fsl_end_block_u16(終了ブロック)に値を入力します。

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

引数の定義

引　　数	説　　明
__near fsl_fsw_t* getfsw_pstr	FSW設定値取得用変数 （FSWの開始ブロック番号、FSWの終了ブロック番号）

fsl_fsw_t の定義

開発ツール	C言語（構造体の定義）	アセンブリ言語（変数の定義例）
RENESAS製 CA78K0R コンパイラ	typedef struct { fsl_u16 fsl_start_block_u16; fsl_u16 fsl_end_block_u16; } fsl_fsw_t;	getfsw_pstr: fsl_start_block_u16: DS 2 fsl_end_block_u16 : DS 2
RENESAS製 CC-RL コンパイラ	typedef struct { fsl_u16 fsl_start_block_u16; fsl_u16 fsl_end_block_u16; } fsl_fsw_t	getfsw_pstr: fsl_start_block_u16: .DS 2 fsl_end_block_u16 : .DS 2
LLVM コンパイラ	typedef struct { fsl_u16 fsl_start_block_u16; fsl_u16 fsl_end_block_u16; } fsl_fsw_t	コンパイラの仕様をご確認ください。

fsl_fsw_t のパラメータ内容

引　数	説　明
fsl_u16 fsl_start_block_u16;	FSWの開始ブロック格納用バッファ - 2バイトの専用のデータ・バッファを確保してください
fsl_u16 fsl_end_block_u16;	FSWの終了ブロック格納用バッファ - 2バイトの専用のデータ・バッファを確保してください

引数の設定内容

開発ツール	引数型／レジスタ	
	C言語	アセンブリ言語
RENESAS製 CA78K0R コンパイラ	__near fsl_fsw_t* getfsw_pstr	AX(0-15)：変数の先頭アドレス（16bit）
RENESAS製 CC-RL コンパイラ	__near fsl_fsw_t* getfsw_pstr	AX(0-15)：変数の先頭アドレス（16bit）
LLVM コンパイラ	__near fsl_fsw_t* getfsw_pstr	AX(0-15)：変数の先頭アドレス（16bit）

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了
0x1F(FSL_ERR_FLOW)	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない注 - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態注

注　　ステータス・チェック・ユーザ・モード時のみ

FSL_SwapBootCluster

【概　要】

ブート・スワップを実行し、スワップ後の領域のリセット・ベクタに登録されているアドレスへ移動

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_SwapBootCluster(void)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_SwapBootCluster(void)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_SwapBootCluster(void) __attribute__ ((section ("FSL_RCD")))
```

＜アセンブラ＞

```
CALL !_FSL_SwapBootCluster または CALL !!_FSL_SwapBootCluster
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数、およびFSL_PrepareExtFunctions関数を実行完了させてください。また処理が終了する前に割り込みを受け付ける必要がある場合は、FSL_ChangeInterruptTable関数を使用して、割り込み先をRAMに変更してください。

【機　能】

関数実行直後に割り込み禁止(DI)に設定、ブート・クラスタの入れ替えを実行し、入れ替えられた領域のリセット・ベクタに登録されているアドレスへ移動します。（RL78マイクロコントローラのリセット機能とは異なり、リセット・ベクタ・アドレスからのプログラム実行のみを行います。）

注意 1. ブート・スワップに対応していないRL78マイクロコントローラで本関数を実行しないで下さい。

2. スワップ実行前に、必ずオプション・バイトなどのスワップ後の動作に必要な設定情報を、スワップ先の領域に書き込みを行ってください。
3. この関数が正常に実行された場合は、入れ替えられたブート・クラスタのリセット・ベクタに登録されているアドレスに移動するため、この関数以降の処理は実行されません。
4. この関数ではブート・フラグの反転は行いません。リセットを実行した場合、ブート・クラスタはブート・フラグの設定に沿った状態となります。
5. FSL_ChangeInterruptTable関数で割り込み先を変更した状態で実行すると、リセット・ベクタに登録されているアドレスへ移動した後も、割り込みはFSL_ChangeInterruptTable関数で変更された領域へ入る状態に維持されます。元に戻った状態でリセット・ベクタに登録されているアドレスに移動したい場合、実行前に必ずFSL_RestoreInterruptTable関数を実行し、割り込み先を復元してください。

6. 以下の(1)、(2)の両方を満たす場合、本関数を内蔵ROM上に配置して使用することが可能です。
   (1) FSL_Init関数のステータス・チェック・モードの指定をステータス・チェック・インターナル・モードに設定すること。
   (2) 本関数の処理が終了するまでの間、"割り込みを使用しない"、または"内蔵ROM上での割り込みを禁止に設定"すること。(RAM上での割り込み受け付けは可能)
7. ブート・クラスタの入れ替えを行うため、ブート・クラスタ内に書き換えに必要なユーザ・プログラムやデータ、及びフラッシュ・セルフ・プログラミング・ライブラリ等を配置しないで下さい。

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】
なし

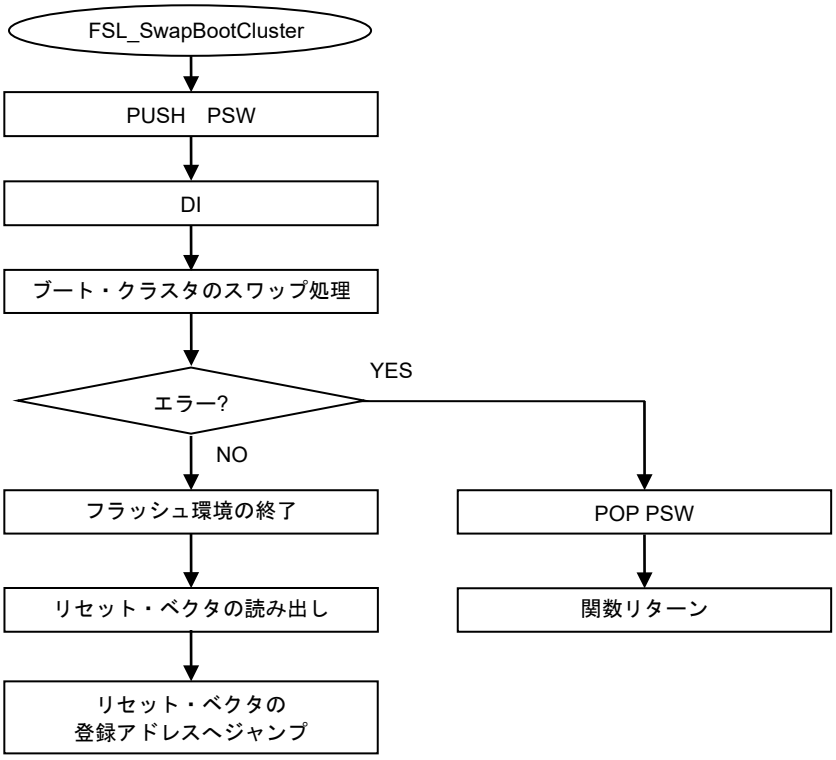
【戻り値】

状　態	説　明
0x10(FSL_ERR_PROTECTION)	プロテクト・エラー - ブート領域書き換え禁止状態で、ブート・スワップを実行しようとした
0x1F(FSL_ERR_FLOW)	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない注 - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態注

注.　ステータス・チェック・ユーザ・モード時のみ

備考. 正常終了の場合は、戻り値は確認できません。

【フロー】


FSL_SwapBootCluster
PUSH　PSW
DI
ブート・クラスタのスワップ処理
エラー?
YES
NO
フラッシュ環境の終了
リセット・ベクタの読み出し
リセット・ベクタの
登録アドレスへジャンプ
POP PSW
関数リターン

FSL_SwapActiveBootCluster

【概　要】

ブート・フラグの現在値を反転し、ブート・スワップの実行

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_SwapActiveBootCluster(void)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_SwapActiveBootCluster(void)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_SwapActiveBootCluster(void) __attribute__ ((section ("FSL_RCD")))
```

＜アセンブラ＞

```
CALL !_FSL_SwapActiveBootCluster または CALL !!_FSL_SwapActiveBootCluster
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数、およびFSL_PrepareExtFunctions関数を実行完了させてください。また処理が終了する前に割り込みを受け付ける必要がある場合は、FSL_ChangeInterruptTable関数を使用して、割り込み先をRAMに変更してください。

【機　能】

この関数を実行すると、ブート・フラグの現在値が反転し、ブート・クラスタの入れ替えも実行されます。

注意 1. ブート・スワップに対応していないRL78マイクロコントローラで本関数を実行しないで下さい。
2. スワップ実行前に、必ずオプション・バイトなどのスワップ後の動作に必要な設定情報を、スワップ先の領域に書き込みを行ってください。
3. リセット等を行わない状態でブート・クラスタの入れ替えを行うため、ブート・クラスタ内に書き換えに必要なユーザ・プログラムやデータ、及びフラッシュ・セルフ・プログラミング・ライブラリ等を配置しないで下さい。また、本関数を実行後にブート・クラスタ内のプログラムやデータを参照する必要がある場合は、ブート・クラスタの入れ替えが行われる事に配慮して使用してください。
4. 本関数はROM上から実行する事は出来ません。本関数を使用する場合は、FSL_RCDセグメントをRAM上に配置して実行してください。
5. 本関数実行後はROM上の割り込みベクタも変更されます。実行前から実行後を通してROM上の割り込み処理を使用する場合は、ROM上の割り込みベクタが動作中に切り替わる事に配慮して使用してください。
6. 本関数を使用する場合はFSL_RCDセグメントに含まれる関数をROM上に配置して使用することは出来ません。

【呼び出し後のレジスタ状態】

	開発ツール	戻り値	破壊レジスタ
	RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
	RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★	LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

なし

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了[注1]
0x10(FSL_ERR_PROTECTION)	プロテクト・エラー - すでに禁止が設定されているフラグを許可しようとした - ブート領域書き換え禁止状態で、ブート領域入れ替えフラグを変更しようとした
0x1A(FSL_ERR_ERASE)	消去エラー[注1] - 消去処理中にエラーが発生した
0x1B(FSL_ERR_IVERIFY)	内部ベリファイ・エラー[注1] - ベリファイ（内部ベリファイ）処理中にエラーが発生した
0x1C(FSL_ERR_WRITE)	書き込みエラー[注1] - 書き込み処理中にエラーが発生した
0x1F(FSL_ERR_FLOW)	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない[注2] - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態[注2]
0xFF(FSL_BUSY)	本関数機能の実行開始[注2] - 本関数の実行が開始された (実行状態をFSL_StatusCheck関数により確認してください)

注 1.　ステータス・チェック・インターナル・モード時のみ

2.　ステータス・チェック・ユーザ・モード時のみ

FSL_InvertBootFlag

【概　要】

ブート・フラグの現在値を反転

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_InvertBootFlag(void)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_InvertBootFlag(void)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_InvertBootFlag(void) __attribute__ ((section ("FSL_RCD")))
```

＜アセンブラ＞

```
CALL !_FSL_InvertBootFlag または CALL !!_FSL_InvertBootFlag
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数、およびFSL_PrepareExtFunctions関数を実行完了させてください。また処理が終了する前に割り込みを受け付ける必要がある場合は、FSL_ChangeInterruptTable関数を使用して、割り込み先をRAMに変更してください。

【機　能】

ブート・フラグの現在値を反転させます。リセット後、ブート・クラスタはブート・フラグの設定に沿った状態となります。

注意 1. ブート・スワップに対応していないRL78マイクロコントローラで本関数を実行しないで下さい。
2. 関数実行時にブート・クラスタの入れ替えを行う事はありません。
3. 下の(1)、(2)の両方を満たす場合、本関数を内蔵ROM上に配置して使用することが可能です。
(1) FSL_Init関数のステータス・チェック・モードの指定をステータス・チェック・インターナル・モードに設定すること。
(2) 本関数の処理が終了するまでの間、"割り込みを使用しない"、または"内蔵ROM上での割り込みを禁止に設定"すること。(RAM上での割り込み受け付けは可能)

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

なし

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了[注1]
0x10(FSL_ERR_PROTECTION)	プロテクト・エラー - すでに禁止が設定されているフラグを許可しようとした - ブート領域書き換え禁止状態で、ブート領域入れ替えフラグを変更しようとした
0x1A(FSL_ERR_ERASE)	消去エラー[注1] - 消去処理中にエラーが発生した
0x1B(FSL_ERR_IVERIFY)	内部ベリファイ・エラー[注1] - ベリファイ（内部ベリファイ）処理中にエラーが発生した
0x1C(FSL_ERR_WRITE)	書き込みエラー[注1] - 書き込み処理中にエラーが発生した
0x1F(FSL_ERR_FLOW)	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない[注2] - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態[注2]
0xFF(FSL_BUSY)	本関数機能の実行開始[注2] - 本関数の実行が開始された (実行状態をFSL_StatusCheck関数により確認してください)

注 1.　ステータス・チェック・インターナル・モード時のみ

2.　ステータス・チェック・ユーザ・モード時のみ

FSL_SetBlockEraseProtectFlag

【概　要】

ブロック消去禁止フラグを禁止に設定

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_SetBlockEraseProtectFlag(void)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_SetBlockEraseProtectFlag(void)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_SetBlockEraseProtectFlag(void)
                                    __attribute__ ((section ("FSL_RCD")))
```

＜アセンブラ＞

```
CALL !_FSL_SetBlockEraseProtectFlag または CALL !!_FSL_SetBlockEraseProtectFlag
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数、およびFSL_PrepareExtFunctions関数を実行完了させてください。また処理が終了する前に割り込みを受け付ける必要がある場合は、FSL_ChangeInterruptTable関数を使用して、割り込み先をRAMに変更してください。

【機　能】

ブロック消去禁止フラグを禁止に設定します。禁止に設定した場合、プログラマによるデバイスへのブロック消去ができなくなります。

注意　以下の(1)、(2)の両方を満たす場合、本関数を内蔵ROM上に配置して使用することが可能です。

(1)　FSL_Init関数のステータス・チェック・モードの指定をステータス・チェック・インターナル・モードに設定すること。

(2) 本関数の処理が終了するまでの間、"割り込みを使用しない"、または"内蔵ROM上での割り込みを禁止に設定"すること。(RAM上での割り込み受け付けは可能)

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

なし

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了[注1]
0x1A(FSL_ERR_ERASE)	消去エラー[注1] - 消去処理中にエラーが発生した
0x1B(FSL_ERR_IVERIFY)	内部ベリファイ・エラー[注1] - ベリファイ（内部ベリファイ）処理中にエラーが発生した
0x1C(FSL_ERR_WRITE)	書き込みエラー[注1] - 書き込み処理中にエラーが発生した
0x1F(FSL_ERR_FLOW)	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない[注2] - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態[注2]
0xFF(FSL_BUSY)	本関数機能の実行開始[注2] - 本関数の実行が開始された (実行状態をFSL_StatusCheck関数により確認してください)

注 1.　ステータス・チェック・インターナル・モード時のみ

2.　ステータス・チェック・ユーザ・モード時のみ

FSL_SetWriteProtectFlag

【概　要】

書き込み禁止フラグを禁止に設定

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_SetWriteProtectFlag(void)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_SetWriteProtectFlag(void)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_SetWriteProtectFlag(void) __attribute__ ((section ("FSL_RCD")))
```

＜アセンブラ＞

```
CALL !_FSL_SetWriteProtectFlag または CALL !!_FSL_SetWriteProtectFlag
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数、およびFSL_PrepareExtFunctions関数を実行完了させてください。また処理が終了する前に割り込みを受け付ける必要がある場合は、FSL_ChangeInterruptTable関数を使用して、割り込み先をRAMに変更してください。

【機　能】

書き込み禁止フラグを禁止に設定します。禁止に設定した場合、プログラマによるデバイスへの書き込みができません。

注意　以下の(1)、(2)の両方を満たす場合、本関数を内蔵ROM上に配置して使用することが可能です。

(1) FSL_Init関数のステータス・チェック・モードの指定をステータス・チェック・インターナル・モードに設定すること。

(2) 本関数の処理が終了するまでの間、"割り込みを使用しない"、または"内蔵ROM上での割り込みを禁止に設定"すること。(RAM上での割り込み受け付けは可能)

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

なし

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了[注1]
0x1A(FSL_ERR_ERASE)	消去エラー[注1] - 消去処理中にエラーが発生した
0x1B(FSL_ERR_IVERIFY)	内部ベリファイ・エラー[注1] - ベリファイ（内部ベリファイ）処理中にエラーが発生した
0x1C(FSL_ERR_WRITE)	書き込みエラー[注1] - 書き込み処理中にエラーが発生した
0x1F(FSL_ERR_FLOW)	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない[注2] - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態[注2]
0xFF(FSL_BUSY)	本関数機能の実行開始[注2] - 本関数の実行が開始された (実行状態をFSL_StatusCheck関数により確認してください)

注 1.　ステータス・チェック・インターナル・モード時のみ

2.　ステータス・チェック・ユーザ・モード時のみ

FSL_SetBootClusterProtectFlag

【概　要】

ブート領域書き換え禁止フラグを禁止に設定

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_SetBootClusterProtectFlag(void)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_SetBootClusterProtectFlag(void)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_SetBootClusterProtectFlag(void)
                                        __attribute__ ((section ("FSL_RCD")))
```

＜アセンブラ＞

```
CALL !_FSL_SetBootClusterProtectFlag または CALL !!_FSL_SetBootClusterProtectFlag
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数、およびFSL_PrepareExtFunctions関数を実行完了させてください。また処理が終了する前に割り込みを受け付ける必要がある場合は、FSL_ChangeInterruptTable関数を使用して、割り込み先をRAMに変更してください。

【機　能】

ブート領域書き換え禁止フラグを禁止に設定します。禁止に設定した場合、ブート・クラスタのスワップ、消去、書き込みはできません。

注意　以下の(1)、(2)の両方を満たす場合、本関数を内蔵ROM上に配置して使用することが可能です。

(1) FSL_Init関数のステータス・チェック・モードの指定をステータス・チェック・インターナル・モードに設定すること。

(2) 本関数の処理が終了するまでの間、"割り込みを使用しない"、または"内蔵ROM上での割り込みを禁止に設定"すること。(RAM上での割り込み受け付けは可能)

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

なし

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了[注1]
0x1A(FSL_ERR_ERASE)	消去エラー[注1] - 消去処理中にエラーが発生した
0x1B(FSL_ERR_IVERIFY)	内部ベリファイ・エラー[注1] - ベリファイ（内部ベリファイ）処理中にエラーが発生した
0x1C(FSL_ERR_WRITE)	書き込みエラー[注1] - 書き込み処理中にエラーが発生した
0x1F(FSL_ERR_FLOW)	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない[注2] - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態[注2]
0xFF(FSL_BUSY)	本関数機能の実行開始[注2] - 本関数の実行が開始された (実行状態をFSL_StatusCheck関数により確認してください)

注 1.　ステータス・チェック・インターナル・モード時のみ
　 2.　ステータス・チェック・ユーザ・モード時のみ

FSL_SetFlashShieldWindow

【概　要】

フラッシュ・シールド・ウインドウの設定

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_SetFlashShieldWindow(__near fsl_fsw_t* setfsw_pstr)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_SetFlashShieldWindow(__near fsl_fsw_t* setfsw_pstr)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_SetFlashShieldWindow(__near fsl_fsw_t* setfsw_pstr)
                                       __attribute__ ((section ("FSL_RCD")))
```

＜アセンブラ＞

```
CALL !_FSL_SetFlashShieldWindow または CALL !!_FSL_SetFlashShieldWindow
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数、FSL_PrepareFunctions関数、およびFSL_PrepareExtFunctions関数を実行完了させてください。また処理が終了する前に割り込みを受け付ける必要がある場合は、FSL_ChangeInterruptTable関数を使用して、割り込み先をRAMに変更してください。

【機　能】

フラッシュ・シールド・ウインドウを設定します。

注意　以下の(1)、(2)の両方を満たす場合、本関数を内蔵ROM上に配置して使用することが可能です。

(1) FSL_Init関数のステータス・チェック・モードの指定をステータス・チェック・インターナル・モードに設定すること。

(2) 本関数の処理が終了するまでの間、"割り込みを使用しない"、または"内蔵ROM上での割り込みを禁止に設定"すること。(RAM上での割り込み受け付けは可能)

備考　フラッシュ・セルフ・プログラミング実行中のセキュリティ機能として、フラッシュ・シールド・ウインドウ機能を搭載しています。
フラッシュ・セルフ・プログラミング実行中では、ウインドウとして指定した範囲内のコード・フラッシュ・メモリは、書き込みおよび消去が可能となり、指定範囲以外のコード・フラッシュ・メモリは、書き込みおよび消去が禁止となります。ただし、オンボード／オフボード・プログラミング時では、ウインドウとして指定した範囲以外のコード・フラッシュ・メモリも、書き込みおよび消去が可能となります。また、ウインドウとして指定した範囲とブート・クラスタ0の書き換え禁止領域が重なる場合は、ブート・クラスタ0の書き換え禁止が優先されます。

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

引数の定義

引　　数	説　　明
__near fsl_fsw_t* setfsw_pstr	FSW設定用変数 （FSWの開始ブロック番号、FSWの終了ブロック番号）

fsl_fsw_t の定義

★ 開発ツール	C言語（構造体の定義）	アセンブリ言語（変数の定義例）
RENESAS製 CA78K0R コンパイラ	typedef struct { 　fsl_u16 fsl_start_block_u16; 　fsl_u16 fsl_end_block_u16; } fsl_fsw_t;	fsl_fsw_str: 　fsl_start_block_u16:　DS 2 　fsl_end_block_u16:　DS 2
RENESAS製 CC-RL コンパイラ	typedef struct { 　fsl_u16 fsl_start_block_u16; 　fsl_u16 fsl_end_block_u16; } fsl_fsw_t;	fsl_fsw_str: 　fsl_start_block_u16:　.DS 2 　fsl_end_block_u16:　.DS 2
LLVM コンパイラ	typedef struct { 　fsl_u16 fsl_start_block_u16; 　fsl_u16 fsl_end_block_u16; } fsl_fsw_t;	コンパイラの仕様をご確認ください。

fsl_fsw_t のパラメータ内容

引　　数	説　　明
fsl_u16 fsl_start_block_u16;	FSWの開始ブロック格納用バッファ - 2バイトの専用のデータ・バッファを確保してください
fsl_u16 fsl_end_block_u16;	FSWの終了ブロック格納用バッファ - 2バイトの専用のデータ・バッファを確保してください

引数の設定内容

	開発ツール	引数型／レジスタ	
		C言語	アセンブリ言語
	RENESAS製 CA78K0R コンパイラ	__near fsl_fsw_t* setfsw_pstr	AX(0-15)：変数の先頭アドレス（16bit）
	RENESAS製 CC-RL コンパイラ	__near fsl_fsw_t* setfsw_pstr	AX(0-15)：変数の先頭アドレス（16bit）
★	LLVM コンパイラ	__near fsl_fsw_t* setfsw_pstr	AX(0-15)：変数の先頭アドレス（16bit）

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了[注1]
0x05(FSL_ERR_PARAMETER)	パラメータ･エラー - ブロック番号の指定が設定可能範囲外
0x1A(FSL_ERR_ERASE)	消去エラー[注1] - 消去処理中にエラーが発生した
0x1B(FSL_ERR_IVERIFY)	内部ベリファイ・エラー[注1] - ベリファイ（内部ベリファイ）処理中にエラーが発生した
0x1C(FSL_ERR_WRITE)	書き込みエラー[注1] - 書き込み処理中にエラーが発生した
0x1F(FSL_ERR_FLOW)	フローエラー - 直前に実行したフラッシュ関数の処理が終了していない[注2] - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態[注2]
0xFF(FSL_BUSY)	本関数機能の実行開始[注2] - 本関数の実行が開始された (実行状態をFSL_StatusCheck関数により確認してください)

注 1.　ステータス・チェック・インターナル・モード時のみ
　 2.　ステータス・チェック・ユーザ・モード時のみ

FSL_StatusCheck

【概　要】

フラッシュ関数の動作状態の確認

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_StatusCheck( void )
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_StatusCheck( void )
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_StatusCheck(void) __attribute__ ((section ("FSL_RCD")))
```

＜アセンブラ＞

```
CALL !_FSL_StatusCheck または CALL !!_FSL_StatusCheck
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

・この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数、およびFSL_PrepareFunctions関数を実行完了させてください。また処理が終了する前に割り込みを受け付ける必要がある場合は、FSL_ChangeInterruptTable関数を使用して、割り込み先をRAMに変更してください。

・この関数はステータス・チェック・ユーザ・モード時のみ使用可能です。

【機　能】

直前に実行したフラッシュ関数の動作開始・継続・状況確認をします。

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

なし

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了[注]
0x1A(FSL_ERR_ERASE)	消去エラー[注] - 消去処理中にエラーが発生した
0x1B(FSL_ERR_IVERIFY)	内部ベリファイ・エラー[注] - ベリファイ（内部ベリファイ）処理中にエラーが発生した
0x1B(FSL_ERR_BLANKCHECK)	ブランク・チェック・エラー[注] - 指定したブロックがブランク状態でない
0x1C(FSL_ERR_WRITE)	書き込みエラー[注] - 書き込み処理中にエラーが発生した
0x1F(FSL_ERR_FLOW)	フローエラー[注] - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態
0x30(FSL_ERR_IDLE)	未実行エラー[注] - 実行している処理が存在しない
0xFF(FSL_BUSY)	フラッシュ関数の実行中[注] - フラッシュ関数の実行中

注　　ステータス・チェック・ユーザ・モード時のみ

FSL_StandBy

【概　要】

消去処理（FSL_Erase）の中断、及びフラッシュ・セルフ・プログラミングの一時停止

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_StandBy(void)
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_StandBy(void)
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_StandBy(void) __attribute__ ((section ("FSL_RCD")))
```

＜アセンブラ＞

```
CALL !_FSL_StandBy または CALL !!_FSL_StandBy
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

・この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数、およびFSL_PrepareFunctions関数を実行完了させてください。

・この関数はステータス・チェック・ユーザ・モード時のみ使用可能です。

【機　能】

実行中の消去処理(FSL_Erase)を中断し、FSL_WakeUpを実行するまで消去処理(FSL_Erase)を停止状態にします。
また、本関数を実行すると、フラッシュ・セルフ・プログラミングが一時停止状態となり、FSL_WakeUpを実行するまでフラッシュ・セルフ・プログラミングを実行できなくなります。

注意　1. フラッシュ・セルフ・プログラミングの一時停止中はフラッシュ関数を実行する事ができなくなります。フラッシュ・セルフ・プログラミングを再開する場合は、FSL_WakeUp関数を実行する必要があります。
2. 戻り値がフローエラー以外は全て一時停止状態に移行します。

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

なし

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了[注1]
0x1A(FSL_ERR_ERASE)	消去エラー[注1] - 中断する前に消去処理でエラーが発生した
0x1B(FSL_ERR_BLANKCHECK)	ブランク・チェック・エラー[注1] - 中断する前にブランク・チェック処理でエラーが発生した
0x1B(FSL_ERR_IVERIFY)	内部ベリファイ・エラー[注1] - 中断する前にベリファイ（内部ベリファイ）処理中にエラーが発生した
0x1C(FSL_ERR_WRITE)	書き込みエラー[注1] - 中断する前に書き込み処理中にエラーが発生した
0x1F(FSL_ERR_FLOW)	フローエラー[注1]（一時停止状態にはなりません） - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態
0x30(FSL_ERR_IDLE)	未実行エラー[注1] - 実行している処理が存在しない
0x43(FSL_SUSPEND)	フラッシュ関数の一時停止中[注1、2] - 実行中のフラッシュ関数の機能は一時停止中になった。

注 1.　ステータス・チェック・ユーザ・モード時のみ
　 2.　消去処理の一時停止時のみ

FSL_WakeUp

【概　要】

一時停止状態を解除し、フラッシュ・セルフ・プログラミングを再開

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
fsl_u08 FSL_WakeUp( void )
```

RENESAS製 CC-RL コンパイラ:

```
fsl_u08 __far FSL_WakeUp( void )
```

★ LLVM コンパイラ:

```
fsl_u08 __far FSL_WakeUp(void) __attribute__ ((section ("FSL_RCD")))
```

＜アセンブラ＞

```
CALL !_FSL_WakeUp または CALL !!_FSL_WakeUp
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

・この関数の実行前に必ずFSL_Init関数を正常終了させた後、FSL_Open関数、およびFSL_PrepareFunctions関数を実行完了させてください。

・この関数はステータス・チェック・ユーザ・モード時のみ使用可能です。

【機　能】

一時停止状態を解除し、フラッシュ・セルフ・プログラミングを再開します。ブロック消去処理を中断していた場合は、ブロック消去処理を再開します。

【呼び出し後のレジスタ状態】

開発ツール	戻り値	破壊レジスタ
RENESAS製 CA78K0R コンパイラ	C (汎用レジスタ)	—
RENESAS製 CC-RL コンパイラ	A (汎用レジスタ)	—
★ LLVM コンパイラ	A (汎用レジスタ)	—

【引　数】

なし

【戻り値】

状　態	説　明
0x00(FSL_OK)	正常終了[注1]
0x1A(FSL_ERR_ERASE)	消去エラー[注1] - 再開した消去処理でエラーが発生した
0x1F(FSL_ERR_FLOW)	フローエラー[注1] - 事前設定で定義されている前提条件に違反している - フラッシュ・セルフ・プログラミングが一時停止状態ではない
0xFF(FSL_BUSY)	フラッシュ関数の再開[注1, 2] - フラッシュ関数の実行が再開された (実行状態をFSL_StatusCheck関数により確認してください)

注 1.　ステータス・チェック・ユーザ・モード時のみ

2.　消去処理の再開時のみ

FSL_ForceReset

【概　要】

★ 使用中のRL78マイクロコントローラをリセット

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
void FSL_ForceReset(void)
```

RENESAS製 CC-RL コンパイラ:

```
void __far FSL_ForceReset(void)
```

★ LLVM コンパイラ:

```
void __far FSL_ForceReset(void) __attribute__ ((section ("FSL_RCD")))
```

＜アセンブラ＞

```
CALL !_FSL_ForceReset または CALL !!_FSL_ForceReset
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

なし

【機　能】

0xFFの命令コードを実行し、使用中のRL78マイクロコントローラに内部リセットを発生させます。

注意 1. 使用中のRL78マイクロコントローラをリセットしますので、この関数以降の処理は実行されません。

2. E1, E2, E2 エミュレータ Lite, E20、またはIECUBE®を使用中に本関数を実行した場合、Breakが発生し、処理がとまります。Break発生以降は正常に動作できませんので、手動でリセットを実行してください。

3. 0xFFの命令コードによる内部リセット（不正命令の実行による内部リセット）については、対象となるRL78マイクロコントローラのユーザーズマニュアルを参照してください。

【呼び出し後のレジスタ状態】

レジスタは破壊されません。

【引　数】

なし

【戻り値】

なし

FSL_GetVersionString

【概　要】

フラッシュ・セルフ・プログラミング・ライブラリのバージョン取得

【書　式】

＜C言語＞

RENESAS製 CA78K0R コンパイラ:

```
__far fsl_u08* FSL_GetVersionString( void )
```

RENESAS製 CC-RL コンパイラ:

```
__far fsl_u08* __far FSL_GetVersionString( void )
```

★ LLVM コンパイラ:

```
__far fsl_u08 * __far FSL_GetVersionString(void)
                                        __attribute__ ((section ("FSL_FCD")))
```

＜アセンブラ＞

```
CALL !_FSL_GetVersionString または CALL !!_FSL_GetVersionString
```

備考　フラッシュ・セルフ・プログラミング・ライブラリを00000H-0FFFFHに配置する場合は“！”、それ以外の場合は“！！”で呼び出してください。

【事前設定】

なし

【機　能】

フラッシュ・セルフ・プログラミング・ライブラリのバージョン情報が入力されている先頭アドレスを取得。

【呼び出し後のレジスタ状態】

	開発ツール	戻り値	破壊レジスタ
	RENESAS製 CA78K0R コンパイラ	BC(0-15), DE(16-31)	—
	RENESAS製 CC-RL コンパイラ	DE(0-15), A(16-23)	—
★	LLVM コンパイラ	DE(0-15), A(16-23)	—

【引　数】

なし

【戻り値】

	データ型	説　明
★	__far fsl_u08*	・フラッシュ・セルフ・プログラミング・ライブラリのバージョン情報の格納先頭アドレス (far 領域) ・ライブラリのバージョン情報の各文字は ASCII コードです。 例：CC-RL コンパイラ用 フラッシュ・セルフ・プログラミング・ライブラリの Type01 の場合 "SRL78T01LyyyzGVxxx" バージョン情報：例：V221 → Ver.2.21 コンパイラ情報（５～６文字）：CA78K0R [例:RyyyG] CC-RL, LLVM [例:LyyyzG] Type 番号（３文字）：T01 → Type01 対象デバイスファミリ（４文字）：RL78 ライブラリ名（１文字）：’S’は FSL

付録A　改版履歴

A. 1　本版で改訂された主な箇所

箇　所	内　容	分類
全般		
－	LLVMコンパイラ用フラッシュ・セルフ・プログラミング・ライブラリの情報を新規追加	(b)
－	図番号とタイトルを図の上部から下部へ移動	(c)
第2章 プログラミング環境		
P24	LLVMコンパイラについての説明を追記 表 2-7にLLVMコンパイラ用のソフトウェアリソースについて追記	(c)
P26	表 2-8にLLVMコンパイラ使用時のスタック・サイズについて追記	(c)
P27	表 2-10, 表 2-11にLLVMコンパイラ使用時のコード・サイズについて追記	(c)
P28	LLVMコンパイラ使用時にセルフRAM領域、スタック領域を設定する場合の説明を追記	(c)
第6章 フラッシュ関数		
P44	LLVMコンパイラについての説明を追記	(c)
P54	LLVMコンパイラ使用時の戻り値に使用する汎用レジスタについて追記	(c)
P55	＜アセンブラ＞の説明内に"注"を追加	(c)
P56 – P110	各ライブラリ関数にLLVMコンパイラ用の戻り値、C言語書式、引数の定義と設定内容を追記	(c)

備考　表中の「分類」により、改訂内容を次のように区分しています。

（a）：誤記訂正，（b）：仕様（スペック含む）の追加／変更，（c）：説明，注意事項の追加／変更，

（d）：パッケージ，オーダ名称，管理区分の追加／変更，（e）：関連資料の追加／変更

A. 2　前版までの改版履歴

これまでの改版履歴を次に示します。なお、適用箇所は各版での章を示します。

(1/4)

版　数	内　容	適用箇所
Rev.1.05	誤記訂正	全般
	ZIPファイル名をインストーラ名に変更	表紙
	対応デバイスを修正	第6章 フラッシュ関数
	操作例：関数名をFSL_ChangeInterruptTable()からFSL_RestoreInterruptTable() に修正	
	対象となるエミュレータ名を追記	

(2/4)

版　数	内　容	適用箇所
Rev.1.04	CC-RLコンパイラ用フラッシュ・セルフ・ライブラリのユーザーズマニュアルと統合	全般
	RL78/G11グループ製品のサポート内容を追記	
	表2-5 FSL_Erase Call Intervalの誤記修正	第2章 プログラミング環境
	対応コンパイラについての説明を追記	
	表2-7にCC-RLコンパイラ用のソフトウェアリソースを追記	
	表2-8にCC-RLコンパイラ用のスタック・サイズを追記	
	表2-10,表2-11にCC-RLコンパイラ用のROM/RAM使用量を追記	
	CC-RLコンパイラ使用時のセルフRAM領域確保、ユーザ指定アドレスの設定方法を追記	
	高速オンチップ・オシレータを起動についての説明を追記	
	動作周波数の設定についての説明を追記	
	セグメント配置に関する制限事項に対象バージョンを追記	
	64KB境界に跨いで各セグメントが配置できないことを追記	
	CC-RLのアセンブラでの16進数表現の設定方法について追記	
	ブート・スワップを行う関数の配置制限を追記	第5章 ブート・スワップ機能
	表6-1にRL78/G11グループ製品の対象関数を追記	第6章 フラッシュ関数
	注1にRl78/G11グループ製品でサポートする ステータス・チェック・モードを記載	
	6.2の題名を変更(セクションを追加)、指定領域への配置について、説明を見直し、修正	
	CC-RLコンパイラでの各ライブラリ関数の戻り値の記述を追記	
	CC-RLコンパイラでの各ライブラリ関数のC言語書式を追記	
	CC-RLコンパイラでの各ライブラリ関数の引数の定義と設定内容を追記	
	各関数の戻り値、引数の記述誤記を訂正	
	ブート領域のアドレス説明を変更	
	FSL_SwapBootCluster関数の注意事項7を追記	

(3/4)

版　数	内　容	適用箇所
Rev.1.03	表紙に対応ZIPファイル名、リリース版を明記	全般
	対応デバイスを削除	
	対象MCUについてのリスト参照説明を追加	
	電圧モードの表記を削除。フラッシュ書き換えモードに表記統一	
	動作周波数の記載について、説明毎に表記方法が異なっていた部分をCPUの動作周波数に表記統一	
	ブート領域にブート・クラスタ0の記述を追加	
	図2－2にFSL_IVerifyの状態確認処理を追加	第1章 概 説
	フラッシュ関数をRAMで実行した場合の説明を追加	第2章 プログラミング環境
	フラッシュ書き換えモードをユーザからインターナルに誤記修正	
	FSL_BlankCheck Min時間の計算式を追加	
	表2-4の各関数処理時間の計算式を修正	
	ステータスチェックの関数名をPFDL_HandlerからFSL_StatusCheckに誤記修正	
	表2-7 セルフRAMの使用領域に記述を変更	
	表2-7 注1で問合せに関する注意事項を変更	
	表2-7 注4の記載を削除	
	図2-9 V2.10までの制限事項を削除	
	表2-8 フラッシュ関数のスタック・サイズに関する注意事項を追加	
	セルフRAMに関する説明を見直し、修正	
	(3)ウォッチドック・タイマで注意が必要な関数の説明を見直し、修正	
	(18) 64KB境界配置不可の注意事項を追加	
	表6-1に基本関数の欄を追加、注の位置、説明内容を変更	第6章 フラッシュ関数
	表題等 フラッシュ書き換えモードをユーザからインターナルに誤記修正	

(4/4)

版　数	内　容	適用箇所
Rev.1.02	フラッシュ・ライブラリのドキュメントの扱いをアプリケーション・ノート（旧版R01AN0350）から ユーザーズマニュアルへ変更	全般
	表紙に対応インストーラ、リリース版を明記	
	処理時間、並びにソフトウェアリソースの内容を使用上の留意点から本書に移動。また、それに伴って前述の内容に対する本書の参照先を変更	
	対応デバイスを追加	
	高速OCOの表記を削除。高速オンチップ・オシレータに表記統一	
	動作周波数の記載について、説明毎に表記方法が異なっていた部分をCPUの動作周波数に表記統一	
	図1－1にprepaerdからextprepaerdに遷移する状態を追加	第1章 概 説
	FSL_PrepareFunctions関数の記述を追加	
	説明全体で関数名を明記	
	上書きに関する注意事項を追加	
	内部ベリファイの説明を追加	
	FSL_Close関数実行時の状態説明を追加	
	割り込みに関する注意事項を追加	第2章 プログラミング環境
	フラッシュの書き換え制御例にモード選択の説明を追加	
	初期設定に関する説明を追加	
	処理時間に関する項目を追加（使用上の留意点から処理時間に関する記述を本書に移動）	
	リソースに関する項目を追加（使用上の留意点からリソースに関する記述を本書に移動）	
	高速オンチップ・オシレータの周波数に関する注意事項を追加	
	RAMパリティ・エラーに関する注意事項を追加	
	書き込み時の注意事項を追加	
	非対応製品（R5F10266）についての記述を追加	
	割り込み関する注意事項を追加	
	ブート・スワップに関する注意事項を追加	
	セキュリティ設定に関する注意事項を追加	
	割り込み関する注意事項を追加	第3章 フラッシュ・セルフ・プログラミング実行中の割り込み
	セキュリティ設定に関する注意事項を追加	第4章 セキュリティ設定
	構造体の定義の表、パラメータ内容の表の追加、引数の設定内容のアセンブラ覧に説明を追加	第6章 フラッシュ関数
	内部リセットに関する注意事項を追加	

RL78ファミリ ユーザーズマニュアル
フラッシュ・セルフ・プログラミング・ライブラリ Type01

発行年月日　2011年 3月 4日　Rev.1.00
2018年10月31日　Rev.1.05
2023年12月26日　Rev.1.10

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

RL78 ファミリ
フラッシュ・セルフ・プログラミング・ライブラリ Type01

ルネサス エレクトロニクス株式会社

R01US0050JJ0110