RX600 & RX200 シリーズ

RX 用シンプルフラッシュ API

R01AN0544JU0240
Rev.2.40
2013.07.01

要旨

フラッシュベースの RX600 および RX200 シリーズデバイスのユーザが、ユーザモードプログラミングを使用して書き換え機能をアプリケーションに容易に組み込むことができる、シンプルなアプリケーションプログラムインタフェース（API）が用意されました。ユーザモードプログラミングは、通常の動作モードで実行中に内蔵フラッシュメモリを書き換えるためのルネサス MCU の機能です。本アプリケーションノートでは、API の使用方法およびアプリケーションへの組み込み方法について説明しています。

この API ソースファイルはルネサス RX コンパイラのみに適合しています。

データフラッシュの消去されている領域を読み込んだ場合

このパッケージで、データフラッシュの消去されている領域を読み込んだ際の値が 0xFF と異なっているというご質問が多く寄せられています。この点に関してはセクション 3.10 をご参照ください。

対象デバイス

この API は次のデバイスで使用できます。

- RX610 グループ
- RX621、RX62N、RX62T、RX62G グループ
- RX630、RX631、RX63N、RX63T グループ
- RX210 グループ

目次

1. 概要

シンプルフラッシュ API は、オンチップフラッシュ領域の書き込みおよび消去をより容易にするためにユーザに提供されています。ROM とデータフラッシュ領域の両方がサポートされています。最も単純な形式の API を使用して消去および書き込み操作のブロックを実行することができます。「ブロッキング」という用語は、書き込みまたは消去関数が呼び出された場合に、この関数の処理が終了するまで、呼び出される前の状態に戻らないことを意味します。フラッシュ操作が実行中、ユーザはそのフラッシュ領域にアクセスすることはできません。フラッシュ領域にアクセスしようとすると、フラッシュ制御ユニットがエラー状態になります。このため、「ブロッキング」操作はフラッシュエラーの可能性を防止するために一部のユーザによって使用されています。しかし、ブロッキング操作が望ましくない場合もあります。たとえば、ユーザがデータフラッシュにデータを書き込んでいる場合、ROM を読み出すことができます。この場合、多くのユーザはアプリケーションが ROM で実行されているときに、データフラッシュの書き込みまたは消去をバックグラウンド（非ブロッキング）で実行するようにします。RX600 および RX200 シリーズ MCU では、この機能がサポートされており、シンプルフラッシュ API で使用可能です。ユーザは非ブロッキングの ROM 操作を行うこともできますが、アプリケーションコードは ROM 以外の領域に格納する必要があります。

1.1 特徴

シンプルフラッシュ API によってサポートされる特徴を以下に示します。

- ユーザ ROM の消去と書き込みのブロッキング
- ユーザ ROM の非ブロッキング、バックグラウンド操作、消去および書き込み
- データフラッシュの消去、書き込み、およびブランクチェックのブロッキング
- データフラッシュの非ブロッキング、バックグラウンド操作、消去、書き込み、およびブランクチェック
- フラッシュ操作が終了したときのためのコールバック関数（非ブロッキングの場合のみ）
- ROM から ROM への転送
- データフラッシュからデータフラッシュへの転送
- ロックビットプロテクト
- ロックビットのセット／読み出し

2. API の情報

ミドルウェアの API は、ルネサスの標準的な命名規則に準じています。

2.1 ハードウェア要件

このミドルウェアでは MCU が次の機能をサポートしていることが必要です。

- バックグランド動作が可能なフラッシュメモリ（すべての RX600 および RX200 シリーズ MCU がこの機能を持っています）
- フラッシュコントロールユニットに供給されているクロック速度が 4MHz 以上

2.2 ハードウェアリソース要件

このセクションではミドルウェアが必要とする周辺回路ハードウェアについて説明します。特に明記されていない限り、周辺回路は専らミドルウェアで使用され、これをユーザアプリケーションで利用することはできません。

2.2.1 フラッシュコントロールユニット（FCU）

FCU は内部フラッシュメモリの書き込みと消去を実行します。ミドルウェアでは FCU を利用しており、これをユーザが直接に操作してはいけません。

2.3 ソフトウェア要件

このソフトウェアは他のいずれのソフトウェアパッケージにも依存していません。

2.4 サポートされているツールチェイン

このミドルウェアは以下のツールチェインの環境でのみ試験と動作確認が行われています。

- ルネサス RX ツールチェイン v1.02

2.5 ヘッダファイル

このミドルウェアのプロジェクトコードとともに提供されている r_flash_api_rx_if.h ファイルをインクルードすることで、すべての API 呼び出しを行うことができます。

2.6 整数型のデータ

コード内容を明確にし移植性を増すために、このプロジェクトでは ANSI C99 で定義されている固定長整数（exact width integer types）を使用しています。これらの型は stdint.h ファイルで定義されています。

2.7 コンフィグレーションの概要

ミドルウェアのコンフィグレーションは提供されている r_flash_api_rx_config.h ファイル上で行われます。個々のコンフィグレーション項目はこのファイルにおけるマクロ定義の形で指定されます。次の表は各項目の詳細です。

表1　フラッシュ API コンフィグレーション項目

r_flash_api_rx_config.h にあるコンフィグレーションオプション	
FLASH_API_RX_CFG_ ENABLE_ROM_PROGRAMMING	定義された場合、ROM 書き込みが有効になり、この操作に必要なコードが RAM にコピーされます。未定義の場合、データフラッシュ操作のみが使用可能で、すべてのコードが ROM に格納されます。
FLASH_API_RX_CFG_FLASH_TO_FLASH	定義された場合、ROM から ROM への操作およびデータフラッシュからデータフラッシュへの操作が有効になります。有効な場合、フラッシュ API は書き込むデータを保持するための RAM バッファを必要とします。RAM バッファのサイズは、データフラッシュと ROM の書き込みサイズの間の最大バイト数です。
FLASH_API_RX_CFG_DATA_FLASH_ BGO	非ブロッキングデータフラッシュ操作を有効にします。有効にした場合、データフラッシュ操作はバックグランドで実行され、API 関数は操作が終了する前に返ります。無効にした場合は、API 関数はデータフラッシュ操作が完了するまで返りません。
FLASH_API_RX_CFG_ROM_BGO	非ブロッキング ROM 操作を有効にします。有効にした場合、ROM 操作はバックグランドで実行され、API 関数は操作が終了する前に返ります。無効にした場合は、API 関数は ROM 操作が完了するまで返りません。
FLASH_API_RX_CFG_FLASH_READY_IPL	BGO 操作が有効な場合に、フラッシュレディ割り込みに使用される割り込み優先レベルです。
FLASH_API_RX_CFG_ IGNORE_LOCK_BITS	定義された場合、ロックビットプロテクトは無視されます。未定義の場合、ロックビットプロテクトが使用され、ロックビットがセットされたブロックに対して書き込み／消去を実行しようとすると、操作は失敗します。
FLASH_API_RX_CFG_ COPY_CODE_BY_API	リセット後、API を利用する前にフラッシュ API を RAM にコピーしておかなければなりません。従来これは他の RAM セクションの初期化の際にコードをコピーするよう dbsct.c ファイルを修正することで行われてきました。今回の版では同じ役割を果たす R_FlashCodeCopy()関数が用意されました。R_FlashCodeCopy()関数を使用するには、このマクロのコメントを外します。従来の dbsct.c による手順を利用する場合には、このマクロをコメントとすることで関数を無効とします。

2.7.1 MCU 情報のコンフィグレーションについて

このミドルウェアの以前の版では、MCU に関する情報はユーザが入力することが必要でした。必要とされた情報の例には次のような項目が含まれます。

- 使用される MCU ファミリ（例えば RX62N など）
- ROM とデータフラッシュの大きさ
- FCU に供給されるクロックの速度

フラッシュ API のコードが r_bsp パッケージを利用するよう変更されたため、これらはフラッシュ API ミドルウェアの中では定義されなくなりました。r_bsp パッケージには個々の RX ボードのためのスタートアップコードと MCU 情報が格納されています。フラッシュ API は必要な情報を r_bsp パッケージのファイルから読み出します。ユーザには独自のボードを r_bsp パッケージに追加されることを強くお勧めします。ミドルウェアを構築する際の明確な基礎をこの先頭に置くことで、RX ミドルウェアの統合性が一層増すことになります。

2.7.2 DATA_FLASH_OPERATION_PIPL と ROM_OPERATION_PIPL について

RX 用シンプルフラッシュ API のバージョン 2.00 では、上の表にはない 2 個の#define がユーザ構成ファイルに存在しました。これらの定義はコードに見つかったバグのために削除されました。これらの定義はフラッシュ操作が呼び出された際に API が MCU の IPL を指定したレベルに設定するためのものでした。フラッシュ操作が完了すると API は IPL をフラッシュ操作が呼び出される前の値に戻します。この方法でフラッシュ操作中に何らかの割り込みが発生し、ROM またはデータフラッシュへのアクセス違反が発生することを容易に防ぐことができます。問題はフラッシュ操作の完了時に MCU の IPL を元に戻す際に生じました。フラッシュ操作が BGO を使用して実行されているときにはフラッシュレディ ISR の処理の中でフラッシュ操作を完了することになります。IPL を ISR の中で変更することは可能ですが、ISR から戻る時点で IPL をスタックから戻しているため、この変更は実質的に効果がありません。これはフラッシュ操作が完了したときに MCU の IPL が正しく戻されないことを意味します。この問題を解決するために、これらの定義は削除されました。このため、フラッシュ操作の間にアクセス違反の原因となる割り込みが発生しないよう、ユーザはより一層の注意を払う必要があります。

ユーザがこの機能を復元するには 2 つの方法があります。1 つ目は、ISR が呼び出された時点でスタックに格納された IPL の値を変更するコードを用意することです。スタック上にどれだけの変数が確保されているかや何個のレジスタが保存されているかによって格納されている IPL の位置が異なるため、注意しなければならないコードとなります。別の方法はフラッシュレディ割り込みを高速割り込みとすることです。この手法は IPL が常にバックアップ PSW レジスタに保存されていることになるため、より容易で安全なコードとなります。この手法の不利な点は別の割り込みのために高速割り込み機能を使用できなくなることです。

2.8 API のデータ構造

このセクションではミドルウェアの API 関数で使用されるデータの詳細を解説します。

2.8.1 フラッシュブロックのアドレス

ユーザは MCU メモリブロックに関連付けられているアドレスを得るために、必要ならば g_flash_BlockAddresses[] 配列を利用することができます。これらのアドレスはフラッシュの消去と書き込みのためのアドレス値で、その領域を読み出す際のアドレスではないことに注意してください。この相違点は、ROM の消去と書き込みの際のアドレスは最上位バイトの値が 0x00（例 0x00FF4000）であるのに対して、読み出し時の ROM アドレスでは最上位バイトが 0xFF（例 0xFFFF4000）となることです。配列から得られた ROM アドレスに対して 0xFF000000 を OR することで読み出しのための ROM アドレスを容易に得ることができます。データフラッシュを利用する際には、このようなアドレスの違いはありません。また、この配列はデフォルトでは ROM 領域に格納される定数配列ですので ROM 消去の際にこの配列を消去することがないように注意しなければなりません。

```
/* Data Structure #1 */
const uint32_t g_flash_BlockAddresses[86] = {
   0x00FFF000,  /* EB00 */
   0x00FFE000,  /* EB01 */
   0x00FFD000,  /* EB02 */
   0x00FFC000,  /* EB03 */
   ...
};
```

2.9　戻り値

API 関数が返す可能性があるいくつかの値を示します。これらの定義はすべて r_flash_api_rx_if.h に存在します。ミドルウェアの従来の版との互換性を保つため、戻り値の中には同じ値を持つものもあります。ただし、いずれの関数も、下記リスト中の同じ値を持つ 2 つの戻り値定義の両方を使用することはありません。

```
/**** Function Return Values ****/
/* Operation was successful */
#define FLASH_SUCCESS           (0x00)
/* Flash area checked was blank, making this 0x00 as well to keep existing
   code checking compatibility */
#define FLASH_BLANK             (0x00)
/* The address that was supplied was not on aligned correctly for ROM or DF */
#define FLASH_ERROR_ALIGNED     (0x01)
/* Flash area checked was not blank, making this 0x01 as well to keep existing
   code checking compatibility */
#define FLASH_NOT_BLANK         (0x01)
/* The number of bytes supplied to write was incorrect */
#define FLASH_ERROR_BYTES       (0x02)
/* The address provided is not a valid ROM or DF address */
#define FLASH_ERROR_ADDRESS     (0x03)
/* Writes cannot cross the 1MB boundary on some parts */
#define FLASH_ERROR_BOUNDARY    (0x04)
/* Flash is busy with another operation */
#define FLASH_BUSY              (0x05)
/* Operation failed */
#define FLASH_FAILURE           (0x06)
/* Lock bit was set for the block in question */
#define FLASH_LOCK_BIT_SET      (0x07)
/* Lock bit was not set for the block in question */
#define FLASH_LOCK_BIT_NOT_SET  (0x08)
```

2.10　プロジェクトにミドルウェアを加えるには

以下の手順によって、プロジェクトにこのミドルウェアを追加することができます。

1. ‘r_flash_api_rx’ ディレクトリ（このアプリケーションノートで提供されています）をプロジェクトのディレクトリにコピーします。
2. src¥r_flash_api_rx.c ファイルをプロジェクトに追加します。
3. ‘r_flash_api_rx’ ディレクトリをインクルードパスに含めます。
4. ‘r_flash_api_rx¥src’ ディレクトリをインクルードパスに追加します。
5. ‘ref’ ディレクトリからコンフィグレーションファイルの例 r_flash_api_rx_config_reference.h をプロジェクトのディレクトリにコピーし、ファイル名を r_flash_api_rx_config.h とします。
6. コピーされた r_flash_api_rx_config.h で対象システムに応じたコンフィグレーションを行います。
7. フラッシュ API を使用している各ソースファイルに r_flash_api_rx_if.h の#include を追加します。

次の手順は ROM への書き込みや消去を行おうとする場合にのみ必要とされます。データフラッシュのみを操作するときには、この手順は無視します。この手順に関する説明はセクション 2.12 に記されています。

8. ROM 領域に‘PFRAM’という名前のセクションを作ります。
9. RAM 領域に‘RPFRAM’という名前のセクションを作ります。
10. リンカの設定で‘FRAM’セクション内のコードが実際には RAM で実行されるよう指定します。
11. リセット後に忘れずにフラッシュ API コードが ROM から RAM にコピーします。これは R_FlashCodeCopy() 関数を呼び出すことで実行できます。

2.11　制約事項

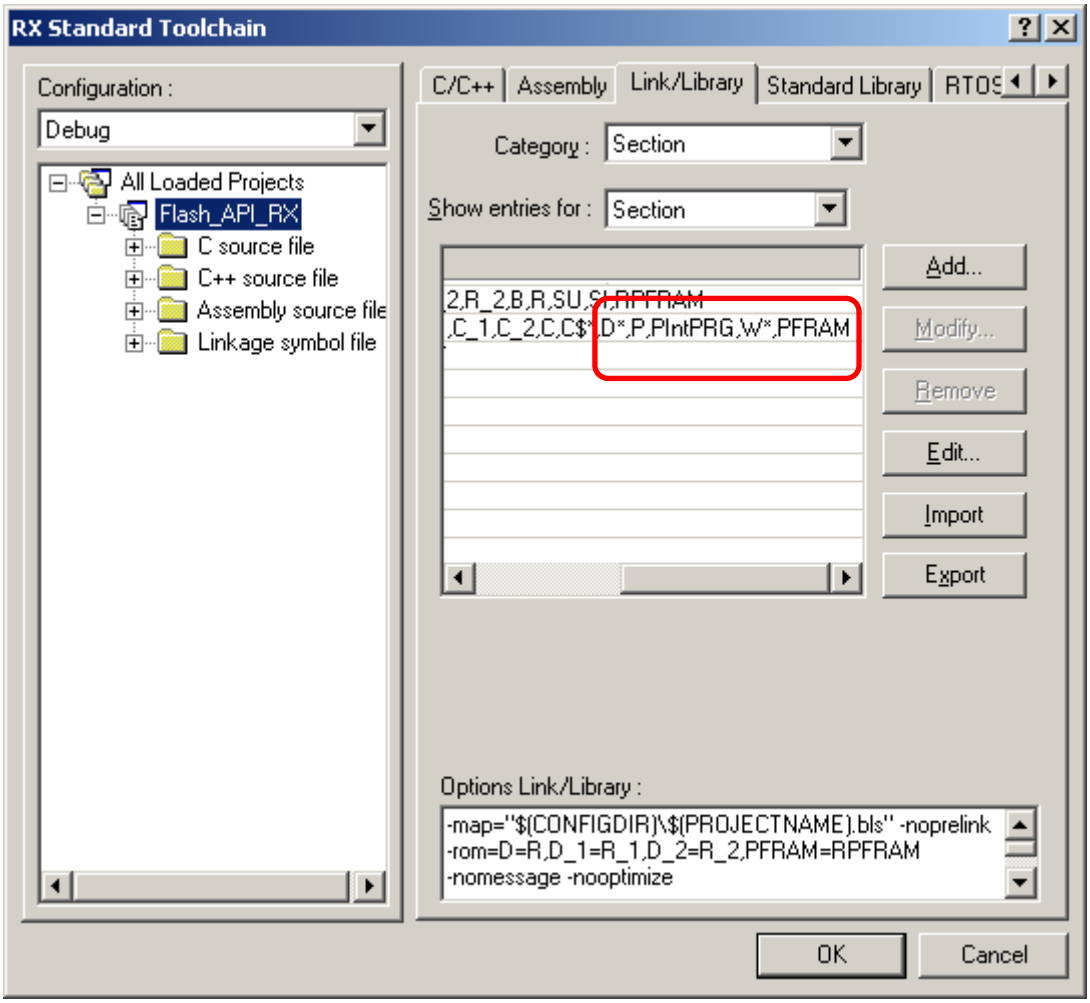
1. このコードはリエントラントではありませんが、同時に同じ関数を複数回呼び出すことに対しては保護されています。
2. ROM 操作中は ROM およびデータフラッシュのいずれにもアクセスできません。ROM BGO 操作を行うときには、コードが間違いなく RAM 上で実行されるよう注意してください。
3. DF の操作中は DF へのアクセスはできませんが、ROM は通常どおりアクセスできます。

2.12　フラッシュ API コードを RAM 上に置くには

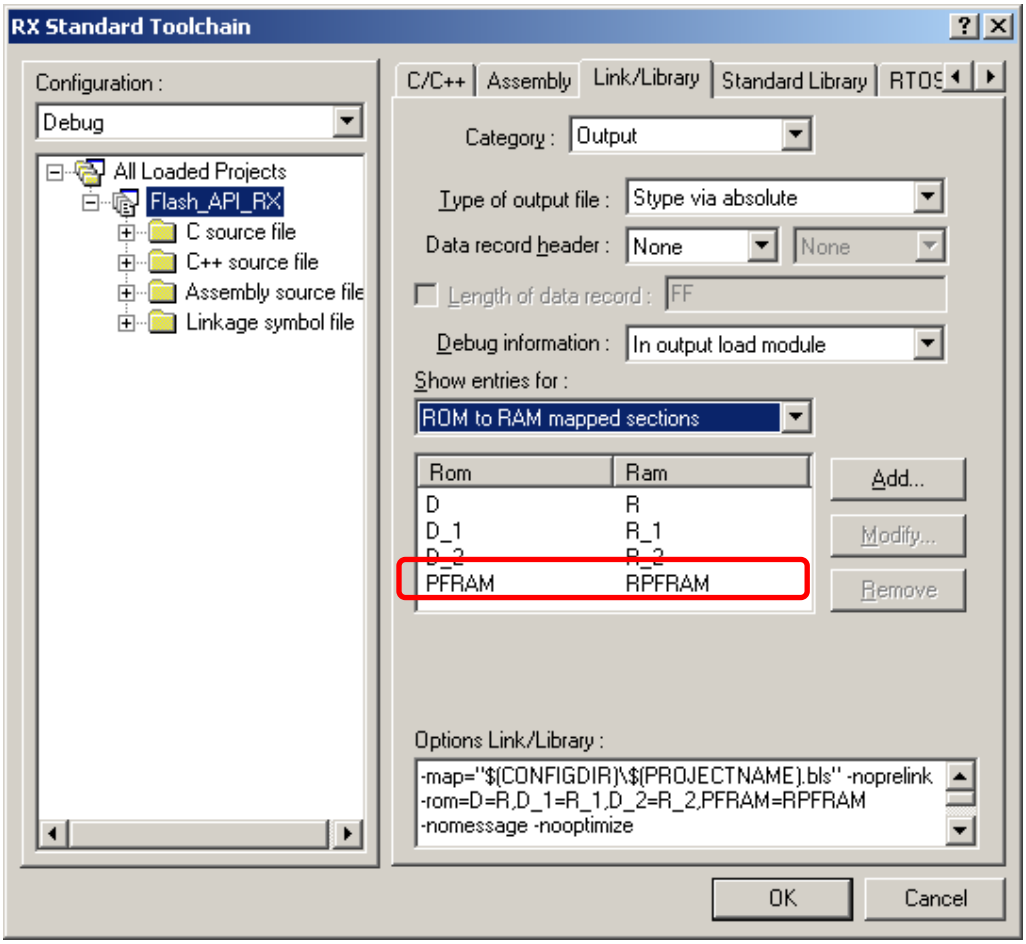
RX600 および RX200 シリーズ MPU では、ROM への再書き込みを行う API 関数を保持するセクションが RAM と ROM 上に必要となります。これは ROM からの読み出し、もしくはプログラムを ROM で実行しているときには FCU が ROM への書き込みや消去を行えないために必要とされます。RAM セクションはまたリセット 後に初期化されなければなりません。これは ROM の書き込みの際にのみ必要となることに注意してください。データフラッシュのみを書き換えるのであれば、これらの設定は必要ありませんが、r_flash_api_rx_config.h ファイルにある FLASH_API_RX_CFG_ENABLE_ROM_PROGRAMMING 項目を未定義としておかねばなりません。ROM の消去と書き込みを行う場合には、以下のステップに従って設定を行います。

HEW の場合:

1. RAM 領域に‘**RPFRAM**’という名前のセクションを新たに加えます。
2. ROM 領域に‘**PFRAM**’という名前のセクションを新たに加えます。



3. 下図のように ROM セクション PFRAM のアドレスを RAM セクションアドレス RPFRAM にマップするリンカオプションを追加します。



4. この段階で、適切なフラッシュ API コードを RAM に正しく配置するリンカオプションが設定されました。次に、リセット後にコードが ROM から RAM にコピーされることを確認しておきます。これが行われていないと、フラッシュ API 関数の呼び出しで、MCU は初期化されていない RAM にジャンプすることになります。コードを RAM にコピーするには以下の二通りの方法があります。

第一の方法は dbsct.c を編集することです。このファイルはリセット後に初期化を行う RAM 領域を指定する配列を持っています。以下のコード中の赤文字で示されているように、dbsct.c に RAM セクションのためのコードの初期化を追加します。（注意: 前の行にコンマを加えることを忘れないでください。）

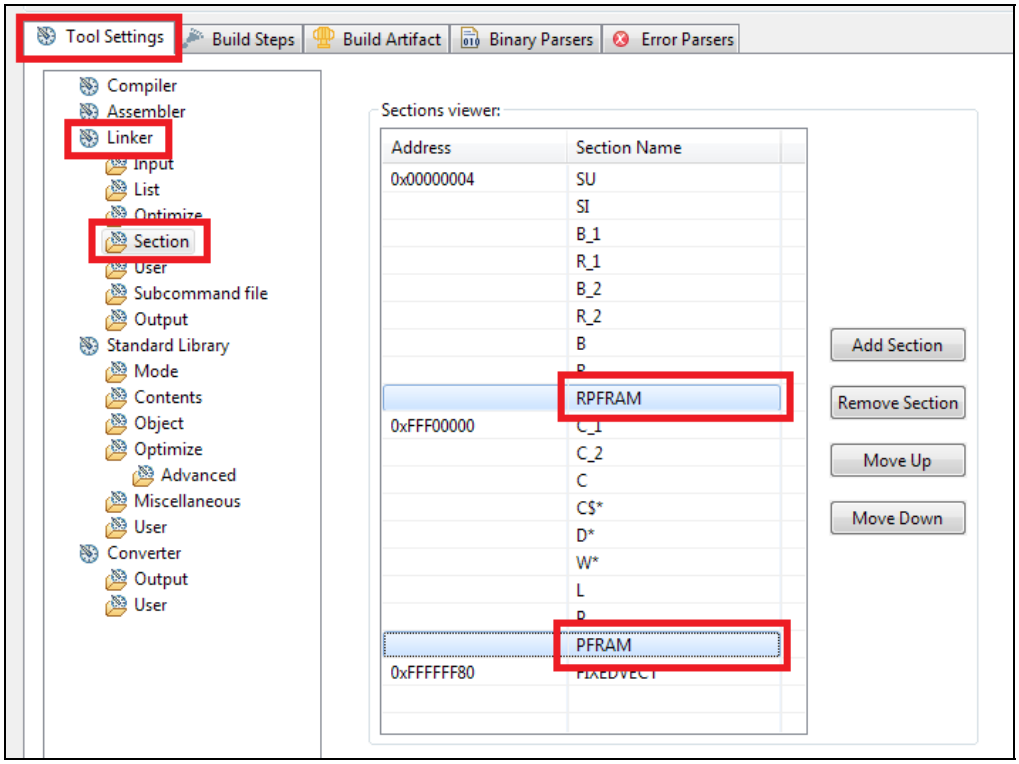
```
-- FILE [dbsct.c] --
#pragma section $DSEC
static const struct {
    _UBYTE *rom_s;  /* Initial address on ROM of initialization data section */
    _UBYTE *rom_e;  /* Final address on ROM of initialization data section */
    _UBYTE *ram_s;  /* Initial address on RAM of initialization data section */
}   DTBL[] = {
    { __sectop("D"), __secend("D"), __sectop("R") } ,
    { __sectop("PFRAM"), __secend("PFRAM"), __sectop("RPFRAM") }
};
```

RX 用シンプルフラッシュ API バージョン 2.20 以降、コードを RAM にコピーする API 関数 R_FlashCodeCopy() が用意されました。他のフラッシュ API 呼び出しを行う前に、この関数を呼び出します。この方法を利用するときには、r_flash_api_rx_config.h 内の COPY_CODE_BY_API マクロのコメントを外すことを忘れないで下さい。先の dbsct.c による方法を採る場合には、このマクロをコメントアウトし、R_FlashCodeCopy() 関数がコンパイルされないようにすることができます。

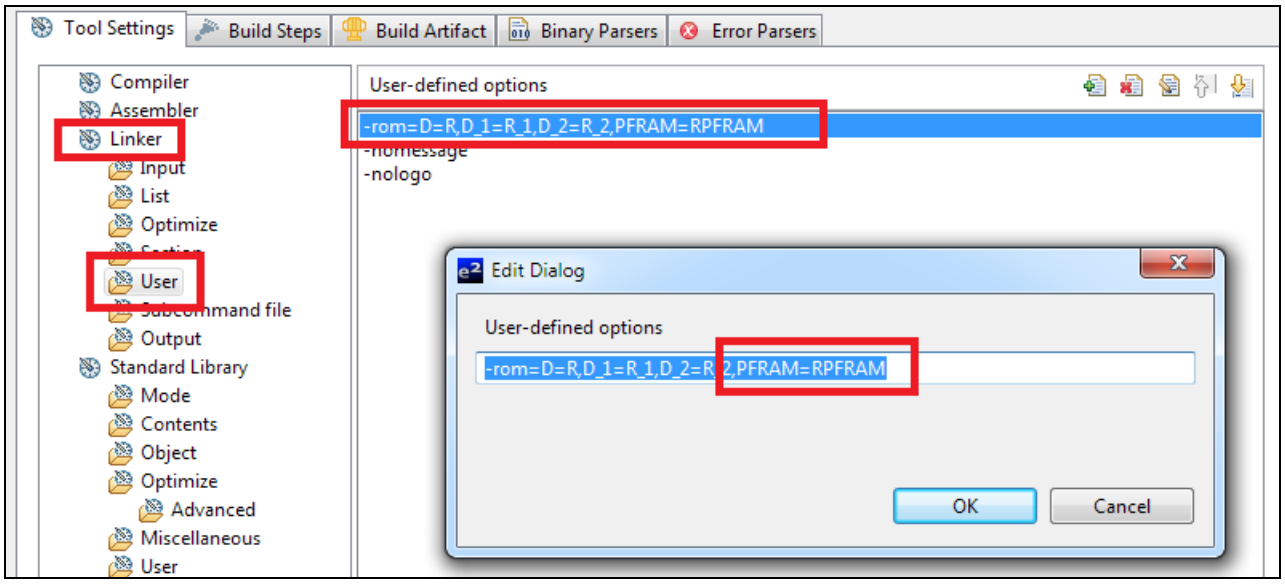
E2Studio の場合:

E2Studio でもリンカセクションと ROM から RAM へのマッピングを設定する同様の手順が必要とされます。

1. RAM 領域に‘**RPFRAM**’という名前のセクションを新たに加えます。
2. ROM 領域に‘**PFRAM**’という名前のセクションを新たに加えます。



3. 次の図のように‘-rom’のユーザ定義オプションに‘,PFRAM=RPFRAM’を加え、ROM セクション PFRAM のアドレスを RAM セクションアドレス RPFRAM にマップするリンカオプションを追加します。これは E2Studio の Tool Settings の中の Linker >> User セクションで実行します。



4. 先の HEW の場合の最終ステップの説明に従い、フラッシュ API を RAM にコピーするための設定を行います。

2.13 非ブロッキングバックグランド操作の利用

ROM またはデータフラッシュに対するバックグランド操作(BGO)が有効にされているとき、API 関数の呼び出しはブロックされず、フラッシュ操作が完了する前に制御が戻ります。この際、ユーザは、操作が完了するまで操作中のフラッシュ領域に対してアクセスを行わないよう注意を払わねばなりません。この領域が操作中にアクセスされると、FCU はエラー状態となり、フラッシュ操作は異常終了となります。

バックグランド操作が終了したときには、ユーザはコールバック関数によってこれを知ることができます。このシンプルフラッシュ API では操作が終了したことを知らせるために 3 種のコールバック関数が用意されています。ユーザはアプリケーションコードでこれらの関数を使用せねばなりません。実際の使用例に関してはアプリケーションノートで提供されている flash_api_rx_demo_main.c ファイルをご覧ください。3 個のコールバック関数は次のとおりです。

- void FlashEraseDone(void)
  — この関数はデータフラッシュもしくは ROM の消去が終了したときに呼び出されます。
- void FlashWriteDone(void)
  — この関数はデータフラッシュもしくは ROM への書き込みが終了したときに呼び出されます。
- void FlashBlankCheckDone(uint8_t result)
  — この関数はデータフラッシュのブランクチェックが完了したときに呼び出されます。ブロックが消去状態であれば‘result’パラメータとして‘FLASH_BLANK’が、消去された状態でないときには‘FLASH_NOT_BLANK’が渡されます。
  フラッシュエラーが発生したときのために、次のコールバック関数もあります。
- void FlashError(void)
  エラーが検出されたとき、フラッシュ API は FCU をリセットしますが、フラッシュ操作が正常に終了しなかったことをユーザに通知するためにこのコールバック関数が用意されています。

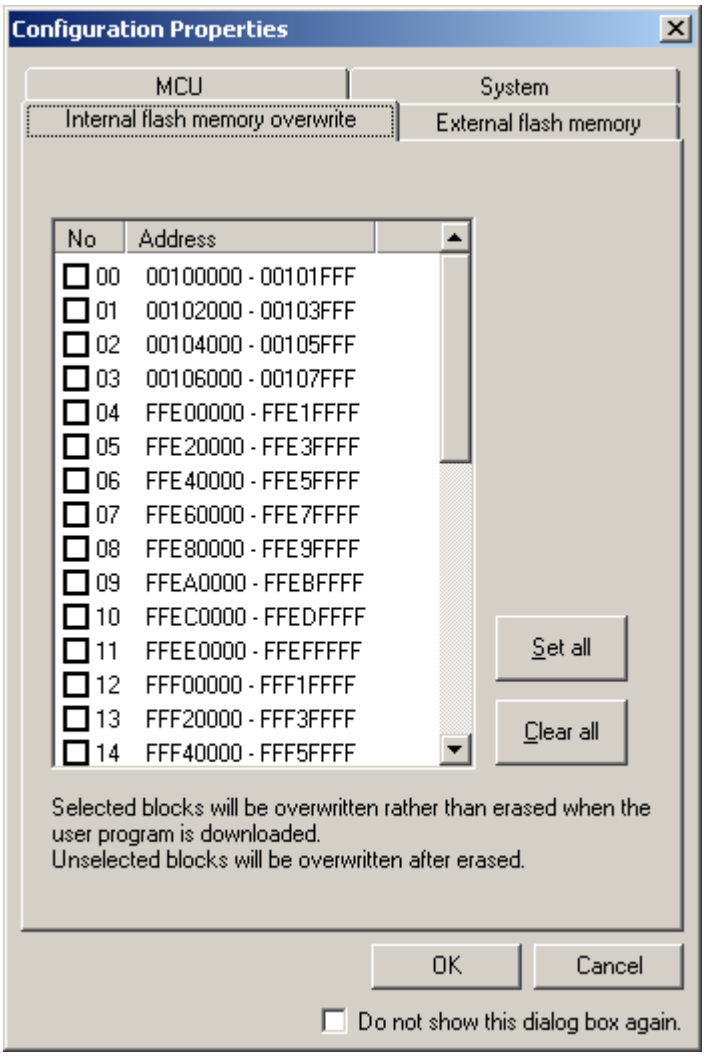
3. 使用の際の注意事項

3.1 HEW からのデバッグ

E1 および E20 を使用して、オンボードのフラッシュメモリおよびデータフラッシュメモリの書き込みや消去中にデバッグを行うことができます。この際には、RAM 上でプログラムを実行し新しいコードを書き込む何らかの手法を用いていない限り、ユーザプログラムを保持しているフラッシュブロックが消去されないように十分な注意を払うことが必要となります。

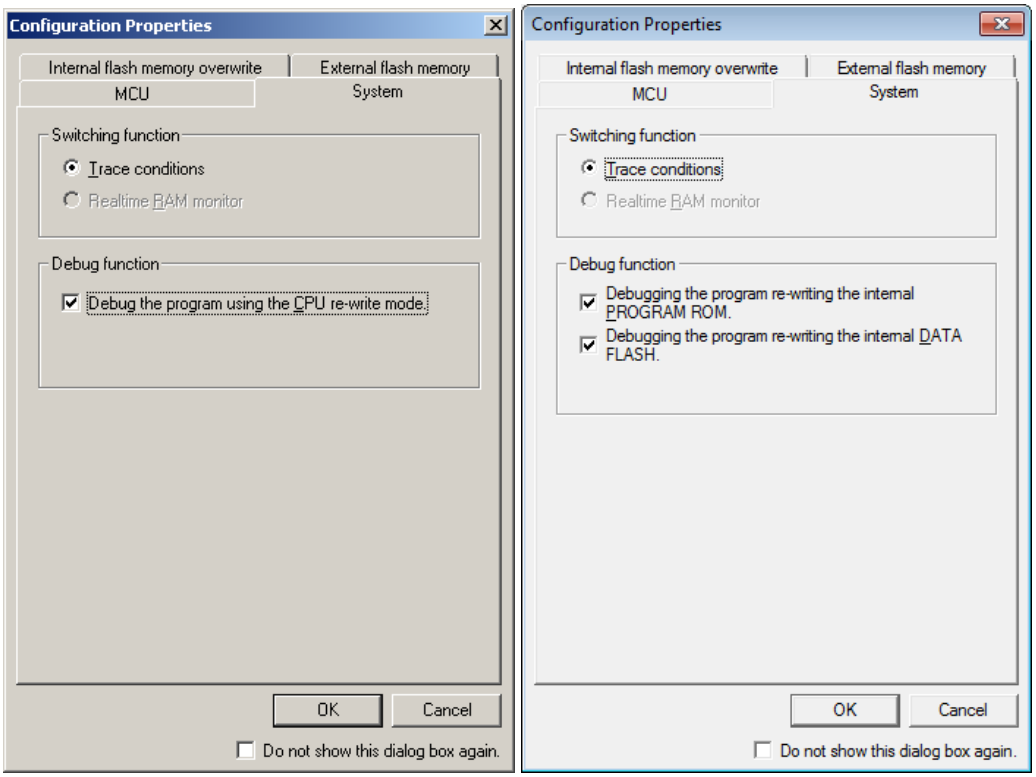
RX600 または RX200 シリーズのデバイスでは組み込まれたセキュリティ機能としてブートモードに入ったときに自動的にフラッシュメモリ全体を消去するため、フラッシュメモリに以前に書き込まれたデータを見るために FDT 書き込みソフトウェアを使用することはできません。

対象システムを HEW から一旦切り離し、再接続を試みたときには、デフォルトのデバッグ設定で E1/E20 と再接続したときにフラッシュメモリ全体が消去されてしまいます。フラッシュメモリの内容を保存するには、どのフラッシュブロックが消去ではなく上書きされるべきかを指示しなければなりません。これを指定するには‘Configuration Properties’ウィンドウの‘Internal flash memory overwrite’ タブの下で、消去ではなく上書きしたいフラッシュブロックの横のボックスにチェックを入れます。下の図がこの際の画面です。



3.2 HEW で消去／書き込みが行われたフラッシュメモリの表示

デフォルトのデバッガ設定では、HEW の Memory ウィンドウを使用して消去や書き込みが行われたフラッシュメモリの内容を見ることはできません。この理由は、デバッグセッションが始まる際に HEW がフラッシュの内容をキャッシュしており、消去や書き込みコマンドが終了した後もこれを更新していないためです。接続時に、CPU 書き換えコードを使用しており、フラッシュメモリの内容を更新しなければならないことを指定するオプションが用意されています。このオプションは E1/E20/JLink に接続したときに現れる ‘Configuration Properties’ ウィンドウにあります。インストールされているデバッガソフトウェアによって、異なるオプションが表示されることがあります。次の図では、表示されることがある異なる画面を示しています。まず System’ タブを選択します。以前のバージョンのデバッガをお使いのときには、左側の図のように、‘Debug the program using the CPU re-write mode’ の横のボックスにチェックを入れます。新しいバージョンのソフトウェアでは、右側の図のような画面が現れます。このときには書き込みや消去を行おうとするメモリ領域の横のボックスにチェックを入れます。両方の領域の書き込みと消去を行うときには、下の例のように両方のボックスをチェックします。これにより Memory ウィンドウでその時点のフラッシュメモリの内容を観察できるようになります。



3.3 ROM 領域の境界

RX600 および RX200 シリーズには複数の ROM 領域を持つ幾つかの MCU が含まれています。一例として 2MB の ROM を持つ RX63N は 4 個の ROM 領域（領域 0、1、2、および 3）を持っています。フラッシュブロックの書き込みが一度に行えるのは同じ領域内に限られており、ROM 領域の境界を超えた書き込みはできません。境界を越えて書き込みを行おうとすると、R_FlashWrite() 関数は書込み操作を行う前に、この状況が生じたことを通知するエラーコードを戻します。境界を越えた書き込みを行いたいときには、この点を確認し、書き込みを分割してまず境界までの書き込みを行い、次に境界から始まる書き込みを行うことが必要です。

どの ROM 領域が書込みと消去のために現在選択されているかは、FENTRY レジスタ内の FENTRY ビットによってコントロールされています。境界を越えた書き込みができない理由は、これらのビットは同時にいずれか 1 個のみがセット可能であることによるものです。どのビットがセットされるかは R_FlashWrite() 関数が呼び出されたときに自動的に管理されています。

3.4 データフラッシュ BGO 操作時の注意事項

データフラッシュ BGO を行うときには、ユーザ ROM や RAM、外部メモリにはアクセスすることができます。つまり、データフラッシュ操作中は、データフラッシュへのアクセスを行わないようにのみ注意すればよいことになります。データフラッシュにアクセスする割り込みに関しても注意が必要です。

3.5 ROM の BGO 操作時の注意事項

ROM の BGO 操作を行っているときには、外部メモリと RAM にアクセスすることができます。多くの場合、コードは ROM に格納されますので、BGO データフラッシュの操作の場合と比べてより一層の注意が必要とされます。API コードは ROM 操作が終了する前に制御を戻しますので、API 関数を呼び出すコードはユーザ ROM 以外の場所に置かれていなければなりません。もうひとつの重要な問題は再配置可能なベクタテーブルに関するものです。デフォルトでは、ベクタテーブルはユーザ ROM 内に常駐します。ROM 操作の途中で割り込みが発生すると、割り込み処理の開始アドレスをフェッチするために ROM へのアクセスが生じ、エラーが発生します。この状況を解消するには、ベクタテーブルと発生する可能性のある割り込みの処理ルーチンを ROM の外に再配置することが必要となります。また、可変ベクタテーブルのポインタレジスタ（INTB）を変更しなければなりません。この例は、アプリケーションノートに付属しているワークスペース例に示します。

3.6 割り込み

ROM またはデータフラッシュ領域は、各メモリ領域に対するフラッシュ操作が進行中はアクセスすることができません。つまり、フラッシュ操作中に割り込みを発生できるようにする際に注意しなければなりません。これらの注意事項は、BGO 操作が利用されているか否かにかかわらず適用されます。

3.7 データフラッシュのみを操作対象とする構成

フラッシュ API はデータフラッシュの操作のみが可能なように構成することができます。ROM の操作を必要としていないユーザにとっては、これは必要なコードと RAM 領域の節約という大きな利点があります。ROM を操作する可能性がなければ、フラッシュ API のコードを RAM 上に置く必要はなく、これにより RPFRAM と PFRAM セクションを設定する必要はなくなります。ROM の操作を行わないように構成するには、次の設定を行います。

1. r_flash_api_rx_config.h ファイルにある ENABLE_ROM_PROGRAMMING マクロをコメントアウトします。
2. 既に RPFRAM と PFRAM セクションが定義されているときには、これらを削除します。また設定されている ROM から RAM へのマッピング設定も削除します。これらの詳細はセクション 2.12 を参照してください。
3. ROM に対する操作を行わない構成のときには R_FlashCodeCopy()関数の呼び出しは必要なくなります。この関数を呼び出した場合でも関数は何も行わずに直ちに戻ります。

3.8 ユーザアプリケーション領域（ROM）全体の消去

ユーザアプリケーション領域全体を消去するにはいくつかの方法が考えられます。そのひとつは、フラッシュ API をユーザブート領域に置く方法です。この領域は通常はブートローダが置かれる場所で、シリアルブートモードでのみ消去が可能です。ユーザアプリケーションがブートモードで実行されることはありませんので、何らかのトラブルでユーザブート領域が消去される懸念はなくなります。ユーザブート領域を利用する場合にはいくつかのフラッシュ API 関数を RAM にコピーする手順をあわせて実行せねばなりません。また ROM やデータフラッシュに対する BGO を行う際には、リロケータブルベクタテーブルをユーザブート領域か RAM に移動しなければなりません。

もうひとつの手法は API 全体を RAM にコピーし、専らこの目的に RAM を利用するものです。これを行うときにはフラッシュアドレスを持つ配列を ROM ではなく RAM に置くようにフラッシュ API コードを修正しなければなりません。配列を RAM に置く設定は次の手順で行います。

1. src/targets/ ディレクトリにある、使用される MCU グループに対応したヘッダファイルを開きます。たとえば RX62N MCU をご使用の場合には src/targets/rx62n/r_flash_api_rx62n.h を開くことになります。
2. このファイルで配列 g_flash_BlockAddresses[]を探します。この配列に関しては配列の内容の定義と配列を extern と定義している 2 個の定義が存在するはずです。両方の定義からキーワード const を削除します。これにより配列は RAM に置かれます。

RAM のみを利用し ROM やデータフラッシュの BGO 操作を行っている場合には、これらの機能で FCU 割り込みが使用されているため、リロケータブルベクタテーブルも RAM に移動することが必要となります。

3.9　リセット直後のデータフラッシュの読み込み

リセット直後にはデータフラッシュの読み込み、書き込みや消去は行えません。これらの操作を可能にするには R_FlashDataAreaAccess()関数を呼び出す必要があります。詳細はこの API の説明（セクション 5.4）を参照してください。

3.10　データフラッシュの特定個所がブランクか（消去されているか）の確認

RX では読み出したデータの値を 0xFF と比べることでデータフラッシュの特定のアドレスがブランクか（消去されているか）を判定することはできません。RX のデータフラッシュでは 1 ビットに対して実際には 2 個のセルが使われています。（これに対し ROM では 1 ビットあたり 1 セルです。）これでビットごとに 4 値を表現することが可能ですが、このうちの 0、1 と「未定義」の 3 値が使用されています。RX ではデータフラッシュの消去されたビットには（0 や 1 ではなく）「未定義」という値が設定されており、読み出されたビット値をこのビットが消去されているかの判定に利用することはできません。データフラッシュのビットが書き込まれると、書き込まれた値が 0 か 1 かに応じて一方のセルの状態が必ず変化します。データフラッシュ上の特定のアドレスが消去されているか否かを判定するには R_FlashDataAreaBlankCheck()関数（セクション 5.5）を使用しなければなりません。この関数ではデータフラッシュの特定のアドレスがプログラムされた値を保持しているか否かの判定に RX のフラッシュ制御ユニット（FCU）のブランクチェック機能を使用しています。

3.11　フラッシュ API をユーザブート領域に配置する場合

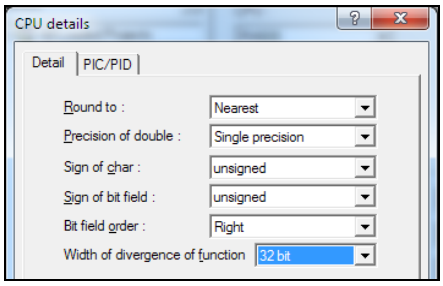
分岐命令で到達できる最大距離として、ルネサス RX ツールチェインはデフォルトでは 24 ビットを使用しています。この値は 16、24、32 ビットのいずれかを選択できます。この値が小さいほどコンパイルされたコードも小さくなります。これは、16 もしくは 24 ビットを選択することで、ルネサス RX ツールチェインに対してすべての分岐命令の分岐先が指定された範囲にとどまり、コンパイラが分岐先を指定するために 32 ビットを確保する必要がないことが保証されるためです。ツールチェインに対してこれが保証されることでコンパイラは分岐命令ごとに 24 ビットオフセットの場合には 1 バイトを、16 ビットオフセットの場合には 2 バイトを節約することができるようになります。

通常のアプリケーションでは 24 ビットが使用されますが、フラッシュ API でユーザブートモードを使用するときには 32 ビット分岐が必要となります。フラッシュ API は ROM の書き込みや消去の際にコードの一部を RAM に置くことが必要となります。ユーザブート領域の最終アドレスは 0xFF800000 であるのに対し、RAM の開始アドレスは 0x00000000 で、この間の距離は 0x800000 となります。この値は 24 ビットで表現できるように見えますが分岐オフセットは正負の値をとる整数値で 2 の補数により表現されているため指定できる範囲は各方向に対してこの半分となります。このため RAM 上の関数からユーザブート領域にある関数を呼び出すには 24 ビット分岐では到達できません。ルネサス RX ツールチェインで分岐命令の長さの設定を変更していないときには‘L2330 (E) Relocation size overflow’というエラーが発生します。

ルネサス RX ツールチェインで 32 ビット分岐を使用するには、次のような設定を行います。

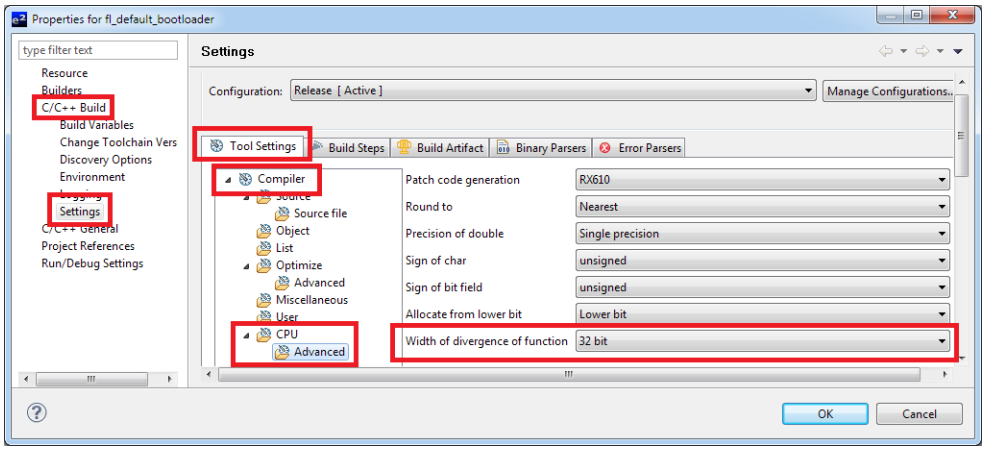
HEW の場合:

1. HEW で対象となるプロジェクトを開きます。
2. Build >> RX Standard Toolchain に進みます。
3. ‘RX Standard Toolchain’ウィンドウの右上にある右矢印を、‘CPU’タブが現れるまでクリックします。
4. ‘CPU’タブをクリックします。
5. ‘Details…’ボタンをクリックします。
6. 現れたウィンドウ（‘CPU details’）にある‘Width of divergence of function’項目で’32 bit’を選択します。



E2Studio の場合:

1. 対象のプロジェクトフォルダを右クリックし、‘Properties’を選択します。
2. ‘C/C++ Build’を展開し、‘Settings’をクリックします。
3. ‘Tool Settings’タブをクリックし、Compiler >> CPU >> Advanced を選択します。
4. ‘Width of divergence of function’項目で‘32 bit’を選択します。



4. ブートローダの実装

デバイスのメモリスペース全体の消去と書き込みを行うことのできるブートローダを作成するときには、コード全体をユーザブートフラッシュ領域に置くか、ブートローダアプリケーション全体を RAM に移動するかのいずれかをしなければなりません。現在実行中のフラッシュブロックを消去することはできませんので、コードをユーザフラッシュ領域に残しておくことはできません。

4.1 アプリケーション全体をユーザブート領域に移動する場合

いくつかの RX600 および RX200 シリーズのデバイスはユーザブート領域と称される独立したフラッシュ領域を持っています。ユーザアプリケーションのリセットベクタを使用する代わりに MCU がこの領域から起動するように設定することができます。このフラッシュ領域は MCU 上で動作しているユーザプログラムから書き込みや消去を行うことができません。このような特徴から、このフラッシュ領域はブートローダアプリケーションを格納するために便利です。この実装の際の手順と注意事項が以下に記されています。

- HEW のリンカ設定を、アプリケーションと割り込みベクタテーブル（IVT）がユーザブート領域に置かれるように変更します。この変更が行われていないと、デフォルトでコードと IVT は通常のユーザフラッシュ領域に置かれてしまいます。
- ユーザブートモードではリセットベクタの位置が 0xFFFFFFFC から 0xFF7FFFFC に移動されています。したがって、ブートローダアプリケーションでは、リセットベクタとして 0xFF7FFFFC をマークしてください。これは次の手順で実行できます。

1. リンカ設定で 0xFF7FFFFC にセクションを設定します。下記の例では BOOTVECT と名付けてあります。
2. 新しい C ソースファイルを用意しプロジェクトに追加し、関数ポインタの配列を加えます。下記の例ではリセット後に実行したい関数を BootLoader と名付けてあります。

```
#pragma section C BOOTVECT

void* const Boot_Vectors[] = {
    //0xFF7FFFFC is reset vector in User Boot Mode
    (void*) BootLoader,
}
```

4.2 ブートローダアプリケーションを RAM に移動する場合

使用する RX デバイスがユーザブート領域を持っていないときや、他の何らかの理由でユーザブート領域を使いたくないときには、ブートローダアプリケーションを RAM に移動することができます。この実装の際の手順と注意事項が次に記されています。

- HEW のリンカ設定で、RAM 領域にプログラムを実行しようとするセクションを設けます。このセクションに名前をつける際に先頭の文字を ‘F’ とし、このセクションが ‘固定 (Fixed)’ 領域セクションであることを示します。一例として、セクションを MY_APP_RAM と名付けたいときには、リンカ設定におけるセクション名を FMY_APP_RAM とします。
- RAM で実行されるコードには、これをロードし格納する ROM 領域が必要ですので、最初に HEW のリンカ設定でセクションを割り当てる必要があります。このセクションに名前をつける際に先頭の文字を ‘P’ とします。これは、このセクションが ‘プログラム(Program)’ 領域であることをツールチェインで示すために必要となります。一例として、セクションを MY_APP と名付けたいときには、リンカ設定におけるセクション名を ‘PMY_APP’ とします。
- RX リンカは、関数（およびデータ）に RAM のアドレスを割り当て、実際にはこれを ROM 領域に置くという特別なオプションを持っています。これは、アプリケーションプログラムがコードもしくはデータを、参照される前に格納されている ROM 領域から RAM 実行領域にコピーするという前提で行われます。コードは RAM に移動された後で絶対アドレスの参照がすべて対象の RAM アドレスに一致しています。また、他のソースモジュールがこのコードを参照する際には、ROM の格納アドレスではなく RAM アドレスが割り当てられています。これはリンカの ROM から RAM へのマッピングオプション “-rom=xxxx=yyyy” で実行されます。ここで、‘xxxx’ が事前に設定されていた ROM セクションを、‘yyyy’ が設定されていた RAM

セクションを示します。これはセクション 2.12 のステップ 3 に記されているように、HEW で設定することができます。このセクションの例では次のようになります。

-rom=PMY_APP=FMY_APP_RAM

- RAM 上のプログラムを実行する際には、まず、格納されている ROM アドレスから実行される RAM アドレスに実行可能なバイナリをコピーしなければなりません。下記はこれを実行する一例です。また、セクション 2.12 のステップ 4 に記されている dbset.c ファイルのコードにセクションの指定を追加しなければなりません。

```
unsigned char *scr;
unsigned char *dst;

src = (unsigned char *)(__sectop("PMY_APP"));
dst = (unsigned char *)(__sectop("FMY_APP_RAM"));
for( ; src < (unsigned char *)(__secend("PMY_APP")); src++, dst++)
{
    *dst = *src;
}
```

- コードを記述する際には、どの関数が RAM に再配置されるこの特別な ROM セクションであるかをリンカに指定する必要があります。それを実行するには、“#pragma section MY_APP” を関数の前に置きます。ここで ‘MY_APP’ の前の ‘P’ が削除されていることに注意してください。これは、実行可能なコードを保持する個々のセクションの先頭にコンパイラが自動的に ‘P’ を挿入するからです。‘#pragma section’ がソースファイルに使用されると、これ以降、ファイルの終わり、もしくは別の ‘#pragma section’ が現れるまでに記述されている関数やデータのすべてを指定されたセクションに格納されることに注意してください。次の ‘#pragma section’ がセクション名を持たないときには、これ以降はデフォルトのセクションが使用されます。セクション名を指定することも可能で、これ以降のコードやデータに対してそのセクション名が使用されます。詳細は RX ツールチェインマニュアルを参照してください。次はこの使用方法の例です。

```
#pragma section MY_APP
void function1( void )
{
    {THIS FUNCTION WILL BE PLACED IN ‘PMY_APP’}
}
void function2( void )
{
    {THIS FUNCTION WILL BE PLACED IN ‘PMY_APP’}
}
```

- 最後の 1 つの確認事項として、参照関数がすべて RAM に再配置されるセクションの一部となっていることを確認してください。誤ってコードが（すでに消去されている可能性がある）ROM 内の関数を呼び出す場合は、コードを移動し RAM から実行することは好ましいことではありません。場合によっては、コード全体で同じ機能を複数回実装するより 1 個のライブラリ関数を呼び出すほうがコードのサイズを小さくできるため、コンパイラの最適化によって、コードの効率を高めるために対象コードが共通の標準ライブラリを呼び出すことがあります。この一例として、アプリケーションコードにおける 32 ビットの乗算を実行する場合が考えられます。これはコンパイラが生成した出力を確認しない限り気づきにくいものです。これらのライブラリは（ユーザが設定した ROM から RAM へのセクションではなく）デフォルトの ROM ベースの領域に置かれるため、あるユーザの再書き込みコードは数ブロックを消去している間は正常に動作しますが、呼び出されているライブラリ関数が格納されているブロックが消去された時点でアプリケーションが最終的にクラッシュします。

5. API 関数

5.1 概要

この API パッケージでは次の関数が提供されています。

関数名	概要
R_FlashErase()	フラッシュブロック全体を消去します。
R_FlashEraseRange()	アドレスで指定された領域を消去します。この関数では少なくとも 1 個のフラッシュブロックが消去されます。
R_FlashWrite()	ROM もしくはデータフラッシュにデータを書き込みます。
R_FlashDataAreaAccess()	データフラッシュ領域の読み出し、書き換え、消去を許可します。
R_FlashDataAreaBlankCheck()	データフラッシュのアドレスで指定された領域もしくはブロックが消去されているかをチェックします。
R_FlashProgramLockBit()	ROM ブロックのロックビットをセットし、消去や書き込みを禁止します。
R_FlashReadLockBit()	ROM ブロックのロックビットの状態を読み出します。
R_FLashSetLockBitProtection()	ロックビットプロテクト機能の有効･無効を切り替えます。
R_FlashGetStatus()	フラッシュの現在の操作状態を返します。
R_FlashCodeCopy()	フラッシュ API コードを ROM セクションから RAM にコピーします。
R_FlashGetVersion()	この API パッケージのバージョンを返します。

5.2 R_FlashErase

この関数はフラッシュブロック全体を消去します。

フォーマット

```
uint8_t R_FlashErase(uint32_t block);
```

パラメータ

block

消去するブロックを指定します。この値は r_flash_api_rx_if.h ファイルで定義されています。ブロック名はデバイスのハードウェアマニュアルに記載されているものと同じフォーマットです。たとえば、RX610 ではアドレス 0xFFFFE000 にあるブロックはハードウェアマニュアルではブロック 0 と称されており、このパラメータとしては “BLOCK_0” と記述します。

戻り値

FLASH_SUCCESS:	操作が成功しました。（BGO 有効時には、操作が正常に開始されたことを示しています。）
FLASH_FAILURE:	操作が失敗しました。
FLASH_BUSY:	別のフラッシュ操作を実行中です。後で再試行してください。

プロパティ

“r_flash_api_rx_if.h” ファイルにプロトタイプ化されます。
“r_flash_api_rx.c” ファイルに実装されます。

説明

フラッシュメモリの 1 個のブロックを消去します。RX63x MCU 以降、いくつかの RX MCU はデータフラッシュで従来よりはるかに小さな消去ブロックを有しています。一例として RX630、RX631、RX63N では消去ブロックは 32 バイトです。つまり 32KB のデータフラッシュが 1024 ブロックに分割されていることになります。個々のブロックを定義する（例: BLOCK_DB0, BLOCK_DB1, …, BLOCK_DB1023）代わりにデータフラッシュのブロックは 2KB の仮想的なブロックにグループ化されています。個々の仮想ブロックは 64 個の実データフラッシュブロックから構成されます。これは、ユーザが従来と同じように大きなデータフラッシュ領域を容易に消去できるようにするためです。また、R_FlashEraseRange() を利用して、32 バイトの小さなブロックを単位としての消去を行うこともできます。

リエントラント

リエントラントではありませんが、関数を同時に複数回呼び出すことによるエラーを防ぐためのロックによって保護されています。

使用例

```
uint32_t loop;
uint8_t ret;

/* Search for record */
for (loop = 0; loop < NUM_BLOCKS_TO_ERASE; loop++)
{
    /* Erase block */
    ret = R_FlashErase(loop);

    /* Check for errors. */
    if (FLASH_SUCCESS != ret)
    {
        . . .
    }
}
```

注意事項

現在実行しているフラッシュブロックを消去してはいけません。データフラッシュのブロックを消去するときには事前に R_FlashDataAreaAccess() 呼び出しでデータフラッシュを変更可能にしておかねばなりません。

5.3　R_FlashEraseRange（RX610 と RX62x では使用できません）

この関数は、指定されたアドレスにあるフラッシュブロックの消去を開始し、指定されたバイト数が消去された時点で消去を終えます。

フォーマット

```
uint8_t R_FlashEraseRange(uint32_t start_addr, uint32_t bytes);
```

パラメータ

start_addr

消去を始めるアドレスを指定します。このアドレスはデータフラッシュ領域をさしており、消去境界上になければなりません。

bytes

消去するバイト数を指定します。この値はデータフラッシュの消去サイズの倍数でなければなりません。たとえば RX630 の場合にはデータフラッシュの消去サイズは 32 バイトですので、パラメータの値は 32、64、96... などとなります。

戻り値

FLASH_SUCCESS:	操作が成功しました。（BGO 動作時には、操作が正常に開始されたことを示しています。）
FLASH_FAILURE:	操作が失敗しました。
FLASH_BUSY:	別のフラッシュ操作を実行中です、後で再試行してください。
FLASH_ERROR_BYTES:	バイト数が消去サイズと一致しません。
FLASH_ERROR_ADDRESS:	アドレス値が無効、対象はデータフラッシュに限られています。

プロパティ

“r_flash_api_rx_if.h” ファイルにプロトタイプ化されます。
コードの本体は “r_flash_api_rx.c” ファイルに実装されます。

説明

1 個以上のデータフラッシュブロックを消去します。この関数は最初に RX63x に対して提供されました。この MCU は従来の RX600 MCU と比べ、はるかに小さなデータフラッシュ消去ブロックを持っています。データフラッシュブロックのために数多くの #define を使用する代わりに、この関数では消去する領域の指定にアドレスとバイト数を使用できます。

リエントラント

リエントラントではありませんが、関数を同時に複数回呼び出すことによるエラーを防ぐためのロックによって保護されています。

使用例

```
uint8_t ret;

/* Erase 64 bytes. */
ret = R_FlashEraseRange(address, 64);

/* Check for errors. */
if (FLASH_SUCCESS != ret)
{
    . . .
}
```

注意事項

- この関数は RX610 および RX62x MCU では使用できません。この理由は、これらの MCU ではデータフラッシュの消去セクタのサイズが大きいため、R_FlashErase() を使って個別の消去ができるからです。
- この関数はデータフラッシュブロックに対してのみ有効です。ROM ブロックの消去には使えません。
- この関数を使用するときには事前に R_FlashDataAreaAccess() 呼び出しでデータフラッシュを変更可能にしておかねばなりません。

5.4 R_FlashWrite

この関数はフラッシュにデータを書き込みます。

フォーマット

```
uint8_t R_FlashWrite( uint32_t   flash_addr,
                      uint32_t   buffer_addr,
                      uint16_t   bytes);
```

パラメータ

flash_addr

これは書き込むフラッシュまたはデータフラッシュ領域を指すポインタです。アドレスは書き込みライン境界でなければなりません。このパラメータに関する重要な制約については、以下の説明を参照してください。

buffer_addr

これはフラッシュに書き込むデータを含むバッファを指すポインタです。

bytes

buffer_addr バッファに含まれるバイト数です。この値は書き込むメモリ領域の書き込みサイズの倍数でなければなりません。このパラメータに関する重要な制約については、以下の注意事項を参照してください。

戻り値

FLASH_SUCCESS:	操作が成功しました。(BGO 動作時には、操作が正常に開始されたことを示しています。)
FLASH_FAILURE:	操作が失敗しました。
FLASH_BUSY:	別のフラッシュ操作を実行中です。後で再試行してください。
FLASH_ERROR_ALIGNED:	フラッシュアドレスが書き込み境界値ではありません。
FLASH_ERROR_BYTES:	指定されたバイト数が書き込みサイズの倍数ではありません。
FLASH_ERROR_ADDRESS:	無効なアドレスが入力されました。
FLASH_ERROR_BOUNDARY:	(ROM) ROM 領域の境界をまたぐ書き込みはできません。

プロパティ

“r_flash_api_rx_if.h” ファイルにプロトタイプ化されます。
コードの本体は “r_flash_api_rx.c” ファイルに実装されます。

説明

フラッシュメモリにデータを書き込みます。

書き込みを行うときには、書き込み境界から書き込みを始め、書き込むバイト数は書き込みサイズの倍数でなければなりません。この境界と書き込みサイズは使用する MCU と、ROM またはデータフラッシュのいずれに書き込みが行われるかによって異なります。最初の書き込み境界はフラッシュ領域の先頭にあり、個々の境界は書き込みサイズの倍数ごとに存在します。たとえば、書き込みラインサイズが 256 の場合、渡すフラッシュアドレスのビット B0～B7 がすべて 0 となる必要があります。

いくつかの RX MCU は、越えて書き込みを行うことができない ROM 領域境界を持っています（これは前記の書き込み境界とは異なります）。この位置を越えて書き込む場合は、書き込みを分割し最初に境界まで書き込み、次に境界から書き込むようにしなければなりません。この境界を越えて書き込もうとすると、書き込み操作を行う前に関数はエラーを返します。使用するデバイスの境界は、そのデバイスに対応した *r_flash_api_rx_private.h* ファイルの ROM_AREA_# の定義で確認することができます。

リエントラント

リエントラントではありませんが、関数を同時に複数回呼び出すことによるエラーを防ぐためのロックによって保護されています。

使用例

```
uint8_t ret;
uint8_t write_buffer[PROGRAM_SIZE] = “Hello World...”;

/* Write data to internal memory. */
ret = R_FlashWrite(address, (uint32_t)write_buffer, PROGRAM_SIZE);

/* Check for errors. */
if (FLASH_SUCCESS != ret)
{
    . . .
}
```

注意事項

個々の RX MCU の書き込みサイズは次の表のとおりです。

MCU	ROM 書き込みラインサイズ	データフラッシュ 書き込みラインサイズ
RX61x および RX62x グループ	256 バイト	8 または 128 バイト
RX63x グループ	128 バイト	2 バイト
RX210 グループ	2、8、または 128 バイト	2 または 8 バイト

データフラッシュのブロックを書き込むときには事前に R_FlashDataAreaAccess() を呼び出してデータフラッシュを変更可能にしておかねばなりません。

5.5　R_FlashDataAreaAccess

この関数はデータフラッシュ領域のアクセスまたは書き換えを許可します。**この関数が一度も呼び出されていないときにはデータフラッシュの読み出し、書き込み、消去はできません。**

フォーマット

```
void R_FlashDataAreaAccess(uint16_t read_en_mask,
                           uint16_t write_en_mask);
```

パラメータ

read_en_mask

これはビットマップ値で、各ビットは、どのデータブロックが MCU により読み出すことができるかを指定します。'0'はそのブロックがアクセスできないことを示し、'1'はアクセス可能であることを示します。ビット 0～3 はそれぞれデータブロック 0～3 に対応します。

write_en_mask

これはビットマップ値で、各ビットは、フラッシュ制御ユニット（FCU）が書き換え（消去／書き込み）可能なデータブロックを指定します。'0'はそのブロックの書き換えができないことを示し、'1' は書き換え可能なことを示します。ビット 0～3 はそれぞれデータブロック 0～3 に対応します。

戻り値

なし

プロパティ

“r_flash_api_rx_if.h” ファイルにプロトタイプ化されます。
コードの本体は “r_flash_api_rx.c” ファイルに実装されます。

説明

リセット後は、データフラッシュ領域は MCU が読み出すことができません。また、書き換えもできません。この関数は、読み出しまたは書き換えを可能とするブロックを選択するために使用します。この関数は、アプリケーションの最初に、1 回だけ設定します。

リエントラント

リエントラントではありませんが、関数を同時に複数回呼び出すことによるエラーを防ぐためのロックによって保護されています。

使用例

```
/* Enable reading, writing, and erasing of all data flash blocks. */
R_FlashDataAreaAccess(0xFFFF, 0xFFFF);
```

注意事項

なし

5.6　R_FlashDataAreaBlankCheck

データフラッシュ領域内の領域がブランクかどうかは、メモリの指定場所を単純に読み出すだけではチェックできないため、この関数を使用してチェックします。**ユーザがデータフラッシュを読み出して値を 0xFF と比較することでは、その場所がブランクか（消去されているか）否かを知ることができないため、この関数が必要となります。**

フォーマット

```
uint8_t R_FlashDataAreaBlankCheck(uint32_t address,
                                  uint8_t size);
```

パラメータ

address

ブランクチェックする領域のアドレスです。
パラメータ 'size' に 'BLANK_CHECK_8_BYTE'（RX610 と RX62x デバイスで使用可能）を指定した場合、これは 8 バイトアドレス境界に設定しなければなりません。
パラメータ 'size' に 'BLANK_CHECK_2_BYTE'（RX63x デバイスで使用可能）を指定した場合、これは 2 バイトアドレス境界に設定しなければなりません。
パラメータ 'size' に 'BLANK_CHECK_ENTIRE_BLOCK'（すべての RX600 および RX200 シリーズのデバイスで使用可能）を指定した場合、これは定義済みデータブロック番号（'BLOCK_DB0'、'BLOCK_DB1'、'BLOCK_DB2' または 'BLOCK_DB3'）またはデータフラッシュブロックのアドレスを設定しなければなりません。いずれかの機能が有効になります。

size

これは、チェックの対象サイズを 8 バイトとするか、2 バイトとするか、または 8KB ブロック全体とするかを指定します。これは、'BLANK_CHECK_2_BYTE'、'BLANK_CHECK_8_BYTE'、'BLANK_CHECK_ENTIRE_BLOCK' のいずれかに設定しなければなりません。

戻り値

FLASH_BLANK:	（2 もしくは 8 バイトチェック、または非 BGO）アドレスはブランクでした。 （ブロック全体および BGO）ブランクチェック操作が開始されました
FLASH_NOT_BLANK:	ブランクではありません。
FLASH_FAILURE:	操作が失敗しました。
FLASH_BUSY:	別のフラッシュ操作が実行中です。
FLASH_ERROR_ADDRESS:	無効なアドレスが入力されました。
FLASH_ERROR_BYTES:	正しくない size が指定されました。

プロパティ

“r_flash_api_rx_if.h” ファイルにプロトタイプ化されます。
コードの本体は “r_flash_api_rx.c” ファイルに実装されます。

説明

MCU のフラッシュ領域に書き込む前に、その領域はブランクにしておかなければなりません。RX600 および RX200 シリーズのデータフラッシュ領域のメモリでは、ユーザプログラム領域とは異なり、ブランク値 0xFF で表わされないので、フラッシュのセクションがブランクであるかを調べるために、追加の関数が必要とされます。

RX600 および RX200 シリーズのデバイスではブランク領域をチェックするために、小さな領域のチェックと大きな領域のチェックの、二通りの方法があります。小さな領域をチェックする方法では、チェック対象のバイト数はデータフラッシュの最小書き込みサイズ（RX610 および RX62x では 8 バイト、RX63x では 2 バイト）と同じです。大きな領域のチェックではデータフラッシュのブロック全体を一度にチェックします。なお、書き込みに先立ってセクションごとにこの関数を呼び出す必要はありません。この関数はアプリケーションプログラム作成時の補助として用意されたものです。

リエントラント

リエントラントではありませんが、関数を同時に複数回呼び出すことによるエラーを防ぐためのロックによって保護されています。

使用例

```
uint8_t ret;

/* Blank check an entire data flash block. */
ret = R_FlashDataAreaBlankCheck(address, BLANK_CHECK_ENTIRE_BLOCK);

/* Check result. */
if (FLASH_NOT_BLANK == ret)
{
    /* Block is not blank. */
    . . .
}
else if (FLASH_BLANK == ret)
{
    /* Block is blank. */
    . . .
}
```

注意事項

個々の RX MCU のブランクチェックサイズは次の表のとおりです。

MCU	ブランクチェックサイズ
RX610	8 バイトもしくはブロック全体（8KB）
RX62x	8 バイトもしくはブロック全体（2KB）
RX63x	2 バイトもしくはブロック全体（2KB）
RX210	2 バイトもしくはブロック全体（2KB）

5.7　R_FlashProgramLockBit

フラッシュブロックのロックビットをセットします。

フォーマット

```
uint8_t R_FlashProgramLockBit(uint32_t block);
```

パラメータ

block

ロックビットをセットする ROM 消去ブロックです。

戻り値

FLASH_SUCCESS:	操作が成功しました、ロックビットがセットされました。
FLASH_FAILURE:	操作が失敗しました。
FLASH_BUSY:	別のフラッシュ操作を実行中です、後で再試行してください。

プロパティ

“r_flash_api_rx_if.h” ファイルにプロトタイプ化されます。
コードの本体は “r_flash_api_rx.c” ファイルに実装されます。

説明

ROM の各ブロックにはロックビットが関連付けられます。ロックビットプロテクトが有効で、ロックビットが任意のブロックにセットされた場合、そのブロックを書き込みまたは消去することはできません。ブロックを消去または書き込もうとすると、その操作は無視されます。この関数は選択したフラッシュブロックのロックビットをセットします。ロックビットプロテクトを有効にするかどうかは API 関数の R_FlashSetLockBitProtection() によって制御されます。

リエントラント

リエントラントではありませんが、関数を同時に複数回呼び出すことによるエラーを防ぐためのロックによって保護されています。

使用例

```
uint8_t ret;

/* Enable lock bit protection (this is default out of reset) */
ret = R_FlashSetLockBitProtection(true);

/* Check for errors. */
if (FLASH_SUCCESS != ret)
{
    . . .
}

/* Program lock bits */
ret = R_FlashProgramLockBit(flash_block);

/* Check for errors. */
if (FLASH_SUCCESS != ret)
{
    . . .
}
```

注意事項

- ロックビットプロテクトが無効状態でフラッシュブロックが消去されたときには、そのブロックのロックビットはクリアされます。
- r_flash_api_rx_config.h ファイルで FLASH_API_RX_CFG_IGNORE_LOCK_BITS マクロが定義されているときには、この関数は使用できません。

5.8　R_FlashReadLockBit

フラッシュブロックのロックビットを読み出します。

フォーマット

```
uint8_t R_FlashReadLockBit(uint32_t block);
```

パラメータ

block

ロックビットの読み出しを行う ROM 消去ブロックです。

戻り値

FLASH_LOCK_BIT_SET:	ロックビットはセットされています。
FLASH_LOCK_BIT_NOT_SET:	ロックビットはセットされていません。
FLASH_FAILURE:	操作が失敗しました。
FLASH_BUSY:	別のフラッシュ操作を実行しています。

プロパティ

“r_flash_api_rx_if.h” ファイルにプロトタイプ化されます。
コードの本体は “r_flash_api_rx.c” ファイルに実装されます。

説明

ROM の各ブロックにはロックビットが関連付けられます。ロックビットプロテクトが有効で、ロックビットが任意のブロックにセットされた場合、そのブロックを書き込みまたは消去することはできません。ブロックを消去または書き込もうとすると、その操作は無視されます。この関数はフラッシュブロックのロックビットがセットされているかどうかを返します。ロックビットプロテクトを有効にするかどうかは API 関数の R_FlashSetLockBitProtection() によって制御されます。

リエントラント

リエントラントではありませんが、関数を同時に複数回呼び出すことによるエラーを防ぐためのロックによって保護されています。

使用例

```
uint8_t ret;

/* Program lock bits */
ret = R_FlashReadLockBit(flash_block);

/* Check result. */
if (FLASH_LOCK_BIT_SET == ret)
{
    /* Lock bit is set for this block. */
    . . .
}
else if (FLASH_LOCK_BIT_NOT_SET == ret)
{
    /* Lock bit was not set for this block. */
    . . .
}
```

注意事項

- ロックビットプロテクトが無効状態でフラッシュブロックが消去されたときには、そのブロックのロックビットはクリアされます。
- r_flash_api_rx_config.h ファイルで FLASH_API_RX_CFG_IGNORE_LOCK_BITS マクロが定義されているときには、この関数は使用できません。

5.9 R_FlashSetLockBitProtection

ロックビットプロテクトを有効または無効にします。

フォーマット

```
uint8_t R_FlashSetLockBitProtection(uint32_t lock_bit);
```

パラメータ

lock_bit

ロックビットプロテクトを有効にするか、無効にするかを指定するブール値です。'true' に設定された場合、ロックビットプロテクトが有効になります。'false' に設定された場合、ロックビットプロテクトは無効になります。

戻り値

FLASH_SUCCESS:	操作が成功しました。
FLASH_BUSY:	別のフラッシュ操作を実行しています。

プロパティ

“r_flash_api_rx_if.h” ファイルにプロトタイプ化されます。
コードの本体は “r_flash_api_rx.c” ファイルに実装されます。

説明

ROM の各ブロックにはロックビットが関連付けられます。ロックビットによるプロテクトが有効で、ロックビットが任意のブロックにセットされた場合、そのブロックを書き込みまたは消去することはできません。ブロックを消去または書き込もうとすると、その操作は無視されます。この関数は、ロックビットプロテクトが有効か否かをコントロールします。無効にした場合、ロックビットプロテクトが設定されているかどうかにかかわらず、フラッシュブロックはすべて書き込みおよび消去が可能となります。

リエントラント

リエントラントではありませんが、関数を同時に複数回呼び出すことによるエラーを防ぐためのロックによって保護されています。

使用例

```
uint8_t ret;

/* Enable lock bit protection (this is default out of reset) */
ret = R_FlashSetLockBitProtection(true);

/* Check for errors. */
if (FLASH_SUCCESS != ret)
{
    . . .
}
```

注意事項

- ロックビットプロテクトが無効状態でフラッシュブロックが消去されたときには、そのブロックのロックビットはクリアされます。
- r_flash_api_rx_config.h ファイルで FLASH_API_RX_CFG_IGNORE_LOCK_BITS マクロが定義されているときには、この関数は使用できません。

5.10　R_FlashGetStatus

フラッシュの現在の状態を返します。

フォーマット

```
uint8_t R_FlashGetStatus(void);
```

パラメータ

なし

戻り値

FLASH_SUCCESS:　フラッシュは使用できる状態です。
FLASH_BUSY:　フラッシュは別の操作でビジーです。

プロパティ

“r_flash_api_rx_if.h” ファイルにプロトタイプ化されます。
コードの本体は “r_flash_api_rx.c” ファイルに実装されます。

説明

この関数はフラッシュの現在の状態を返します。BGO 操作を使用している場合、この関数呼び出しを使用して最後のフラッシュ操作が終了したときを検出するためにポーリングすることができます。

リエントラント

リエントラントです。

使用例

```
uint8_t ret;

/* Blank check an entire data flash block. */
ret = R_FlashDataAreaBlankCheck(address, BLANK_CHECK_ENTIRE_BLOCK);

while( R_FlashGetStatus() == FLASH_BUSY )
{
    /* Wait for previous operation to finish. You could also stall this task
       and do some real work. */
}
```

注意事項

なし

5.11　R_FlashCodeCopy

フラッシュ API コードを ROM から RAM にコピーします。

フォーマット

```
void R_FlashCodeCopy(void);
```

パラメータ

なし

戻り値

なし

プロパティ

“r_flash_api_rx_if.h” ファイルにプロトタイプ化されます。
コードの本体は “r_flash_api_rx.c” ファイルに実装されます。

説明

ROM に対する書き込みや ROM の消去を行う際には、その API コードを ROM に常駐させることはできません。この関数はコードを ROM から RAM に転送します。

リエントラント

リエントラントです。

使用例

```
/* Transfer Flash API code to RAM so that we can program/erase ROM. */
R_FlashCodeCopy();

/* Flash API can now program/erase ROM. */
```

注意事項

- ROM ではなくデータフラッシュのみの書き込みや消去を行う場合には、フラッシュ API のコード全体を ROM に常駐させることができ、この関数を使う必要はありません。
- セクション 2.12 に記されている dbsct.c によるコピー手順が使われているときには、この関数を使う必要はありません。
- ROM への書き込みや消去が行われ、かつ dbsct.c によるコピー手順が利用されていないときにのみ、他の API 関数の呼び出しに先立ってこの関数が呼び出されねばなりません。この関数が最初に呼び出されない場合は、他のフラッシュ API 関数は初期化されていない RAM にジャンプします。

5.12　R_FlashGetVersion

この API パッケージのバージョンを返します。

フォーマット

```
uint32_t R_FlashGetVersion(void);
```

パラメータ

なし

戻り値

フラッシュ API のバージョン。

プロパティ

“r_flash_api_rx_if.h” ファイルにプロトタイプ化されます。
コードの本体は “r_flash_api_rx.c” ファイルに実装されます。

説明

この関数は実際に組み込まれているフラッシュ API のバージョンを戻り値として返します。戻り値は上位 2 バイトがバージョン番号の整数部を、下位 2 バイトが小数点以下を示します。たとえばバージョン 4.25 では戻り値の値は 0x00040019 です。

リエントラント

リエントラントです。

使用例

```
uint32_t cur_version;

/* Get version of installed Flash API. */
cur_version = R_FlashGetVersion();

/* Check to make sure version is new enough for this application’s use. */
if (MIN_VERSION > cur_version)
{
    /* This Flash API version is not new enough and does not have XXX feature
       that is needed by this application. Alert user. */
    ....
}
```

注意事項

- この関数は r_flash_api_rx.c ファイル内でインライン関数として定義されています。

6. プロジェクト例

このアプリケーションノートでは、API 機能のすべてを使用するプロジェクトの例が提供されています。flash_api_rx_demo_main.cファイルには main() 関数とデモコードが含まれています。

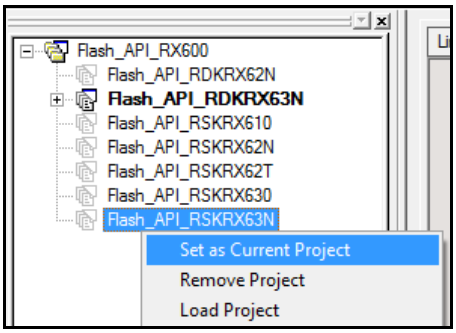
6.1 HEW ワークスペース

実例のワークスペースは複数の異なるプロジェクトから構成されており、各プロジェクトはそれぞれ異なる開発ボードに対応しています。例えば RSKRX62N、RSKRX630、YRDKRX63N などに対するプロジェクトが個別に用意されています。このミドルウェアは主要なボードをサポートするために r_bsp パッケージを使用していますので、プロジェクトはこれに沿った構成となっています。従って、各プロジェクトにはそれぞれのボードに対応したスタートアップファイルが備わっています。

特定のボードに対するデモンストレーションは以下の手順で実行することができます。

HEW の場合:

1. Flash_API_RX ワークスペースの中から、使用されるボードに対応するプロジェクトを選択します。このためには HEW でボードに対応するプロジェクト上で右クリックを行い、‘Set as Current Project’を選択します。



2. r_bsp フォルダにあるヘッダファイル platform.h で、使用するボードを選択します。一例として、RSK+RX63N を使用するときには下図にあるように RSKRX63N #include のコメントを外します。

```
/***********************************************************************************
DEFINE YOUR SYSTEM - UNCOMMENT THE INCLUDE PATH FOR THE PLATFORM YOU ARE USING.
***********************************************************************************/
/* RSKRX610 */
//#include "./board/rskrx610/r_bsp.h"

/* RSKRX62N */
//#include "./board/rskrx62n/r_bsp.h"

/* RSKRX62T */
//#include "./board/rskrx62t/r_bsp.h"

/* RDKRX62N */
//#include "./board/rdkrx62n/r_bsp.h"

/* RSKRX630 */
//#include "./board/rskrx630/r_bsp.h"

/* RSKRX63N */
#include "./board/rskrx63n/r_bsp.h"
```

この変更を行えば、プロジェクトをビルドしてデモを実行することができます。

ホームページとサポート窓口

ルネサス エレクトロニクスホームページ
http://japan.renesas.com/

お問合せ先
http://japan.renesas.com/inquiry

改訂記録	RX600 & RX200 シリーズ RX 用シンプルフラッシュ API

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2010.01.27	—	初版発行
1.20	2010.02.11	—	文章細部の変更、および割り込みの禁止に関する記述セクションの追加
1.30	2010.03.05	—	RTE からの指摘に基づく修正
1.40	2010.05.26	—	RX62x グループのサポートを追加するための変更
1.41	2010.06.11	—	誤植などの訂正
1.43	2011.02.18	—	消去確認（blank check）関数のパラメタ記述の更新
2.00	2011.04.27	—	バックグランド操作（BGO）、フラッシュからフラッシュへの転送、ロックビット保護の機能の追加
2.10	2011.07.11	—	RX630、RX631 および RX63N デバイスのサポートの追加。 DATA_FLASH_OPERATION_PIPL および ROM_OPERATION_PIPL 定義の削除と、その理由の記述セクションを追加。API に R_FlashEraseRange() 関数を追加。RX610 と RS63x デバイスに対応するための ROM 領域境界に関する記述の（セクション 3.4 としての）書き換え。
2.20	2012.03.27	—	新しい文書形式への移行。r_bsp パッケージの利用に関する既存の情報の再構成と新たな情報の追加。API への R_FlashCodeCopy() 関数の追加。
2.30	2012.09.12	—	F_FlashGetVersion()関数を API に追加。 コード内で r_bsp を自動的に認識するよう変更されたため r_bsp を利用しない際のマクロの構成を削除。 「データフラッシュのみを操作対象とする構成」、「ユーザアプリケーション領域（ROM）全体の消去」、「リセット直後のデータフラッシュの読み込み」、「データフラッシュの特定個所がブランクか（消去されているか）の確認」、「フラッシュ API をユーザブート領域に配置する場合」セクションの追加。 ブランクチェックのサイズの表を R_FlashDataAreaBlankCheck セクションに追加。
2.40	2013.07.01	—	RX210、RX62G および RX63T デバイスのサポートの追加。サポート対象に RX200 シリーズのデバイスが加わったため、アプリケーションノートの表題を「RX600 用シンプルフラッシュ API」から「RX 用シンプルフラッシュ API」に変更。 「データフラッシュの特定個所がブランクか（消去されているか）の確認」セクションの追加と、データフラッシュの消去されたデータ読み出しが 0xFF とならない理由の記述個所に関するコメントを先頭ページに付記。 API 関数の一覧表を API 関数セクションの先頭に追加。プロジェクトの実例のセクションを追加。

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、本文を参照してください。なお、本マニュアルの本文と異なる記載がある場合は、本文の記載が優先するものとします。

1. **未使用端子の処理**
   【注意】未使用端子は、本文の「未使用端子の処理」に従って処理してください。
   CMOS製品の入力端子のインピーダンスは、一般に、ハイインピーダンスとなっています。未使用端子を開放状態で動作させると、誘導現象により、LSI周辺のノイズが印加され、LSI内部で貫通電流が流れたり、入力信号と認識されて誤動作を起こす恐れがあります。未使用端子は、本文「未使用端子の処理」で説明する指示に従い処理してください。
2. **電源投入時の処置**
   【注意】電源投入時は，製品の状態は不定です。
   電源投入時には、LSIの内部回路の状態は不確定であり、レジスタの設定や各端子の状態は不定です。
   外部リセット端子でリセットする製品の場合、電源投入からリセットが有効になるまでの期間、端子の状態は保証できません。
   同様に、内蔵パワーオンリセット機能を使用してリセットする製品の場合、電源投入からリセットのかかる一定電圧に達するまでの期間、端子の状態は保証できません。
3. **リザーブアドレスのアクセス禁止**
   【注意】リザーブアドレスのアクセスを禁止します。
   アドレス領域には、将来の機能拡張用に割り付けられているリザーブアドレスがあります。これらのアドレスをアクセスしたときの動作については、保証できませんので、アクセスしないようにしてください。
4. **クロックについて**
   【注意】リセット時は、クロックが安定した後、リセットを解除してください。
   プログラム実行中のクロック切り替え時は、切り替え先クロックが安定した後に切り替えてください。
   リセット時、外部発振子（または外部発振回路）を用いたクロックで動作を開始するシステムでは、クロックが十分安定した後、リセットを解除してください。また、プログラムの途中で外部発振子（または外部発振回路）を用いたクロックに切り替える場合は、切り替え先のクロックが十分安定してから切り替えてください。
5. **製品間の相違について**
   【注意】型名の異なる製品に変更する場合は、事前に問題ないことをご確認ください。
   同じグループのマイコンでも型名が違うと、内部メモリ、レイアウトパターンの相違などにより、特性が異なる場合があります。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。

ご注意書き

1. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器・システムの設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因して、お客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
2. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
3. 本資料に記載された製品データ、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に関し、当社は、何らの責任を負うものではありません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
4. 当社製品を改造、改変、複製等しないでください。かかる改造、改変、複製等により生じた損害に関し、当社は、一切その責任を負いません。
5. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、
   各品質水準は、以下に示す用途に製品が使用されることを意図しております。
   標準水準： コンピュータ、OA機器、通信機器、計測機器、AV機器、
   家電、工作機械、パーソナル機器、産業用ロボット等
   高品質水準：輸送機器（自動車、電車、船舶等）、交通用信号機器、
   防災・防犯装置、各種安全装置等
   当社製品は、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（原子力制御システム、軍事機器等）に使用されることを意図しておらず、使用することはできません。たとえ、意図しない用途に当社製品を使用したことによりお客様または第三者に損害が生じても、当社は一切その責任を負いません。なお、ご不明点がある場合は、当社営業にお問い合わせください。
6. 当社製品をご使用の際は、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他の保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質および信頼性の向上に努めていますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害等を生じさせないよう、お客様の責任において、冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、お客様の機器・システムとしての出荷保証を行ってください。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
8. 当社製品の環境適合性等の詳細につきましては、製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制するRoHS指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
9. 本資料に記載されている当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器・システムに使用することはできません。また、当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途に使用しないでください。当社製品または技術を輸出する場合は、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。
10. お客様の転売等により、本ご注意書き記載の諸条件に抵触して当社製品が使用され、その使用から損害が生じた場合、当社は何らの責任も負わず、お客様にてご負担して頂きますのでご了承ください。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを禁じます。

注1. 本資料において使用されている「当社」とは、ルネサス エレクトロニクス株式会社およびルネサス エレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。
注2. 本資料において使用されている「当社製品」とは、注１において定義された当社の開発、製造製品をいいます。

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口 http://www.renesas.com

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

ルネサス エレクトロニクス販売株式会社　〒100-0004　千代田区大手町2-6-2（日本ビル） (03)5201-5307

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