RX600 & RX200 シリーズ

RX 用仮想 EEPROM

R01AN0724JU0170
Rev.1.70
2013.09.18

要旨

使用している MCU のデータフラッシュを EEPROM のように使用したいとお考えのユーザの方が多くおられます。この際の問題は、RX MCU の多くはデータフラッシュで 1 バイト単位の書き込みや消去が行えないことです。RX MCU がこの機能を提供していたとしても、フラッシュ疲労の均等化や記録の管理に関する問題が生じます。この問題を解消する一助とするために、仮想 EEPROM（以下 VEE と略します）プロジェクトが作成されました。VEE プロジェクトは次の機能を提供します。

- データフラッシュの実際の書き込み単位に関わらず、任意の大きさでデータを書き込むことができます。
- 疲労が均等化されることで、データフラッシュの寿命が延びます。
- 安全に読み書きを行う、使いやすい API インタフェースが提供されます。
- 容易に読み書きを行うための、レコード管理機能が組み込まれています。
- MCU のバックグランド動作機能を使用していますので、データフラッシュの動作が MCU のユーザアプリケーションの妨げとなることはありません。
- フラッシュ書き込みもしくは消去中のリセットや電源断が生じても、自動的に回復します。
- 書き込みサイズ、消去サイズ、ブロックサイズ、およびデータフラッシュのブロック数が異なるフラッシュに対応しています。
- 個々のユーザ独自の要求にあわせた高度な設定が行えます。

対象デバイス

この API は現時点で次のデバイスでサポートされています。

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

本アプリケーションノートを他のマイコンへ適用する場合、そのマイコンの仕様にあわせて変更し、十分評価してください。

目次

1.　概要 ... 3
1.1　バッググランド動作の利用 ... 3
1.2　レコード ... 3
1.3　データ管理 ... 3
1.4　VEE 動作の例 ... 4

2.　API の情報 ... 7
2.1　ハードウェア要件 ... 7
2.2　ハードウェアリソース要件 ... 7
2.2.1　データフラッシュ ... 7
2.3　ソフトウェア要件 ... 7
2.4　制約 ... 7
2.5　サポートされるツールチェイン ... 7
2.6　ヘッダファイル ... 7
2.7　整数型データ ... 7
2.8　コンフィグレーションの概要 ... 8
2.8.1　r_bsp パッケージの使用 ... 9
2.9　VEE セクタのコンフィグレーション ... 9
2.9.1　VEE セクタ数 ... 9
2.9.2　VEE レコードの VEE セクタへの割当て ... 9
2.9.3　VEE ブロックの配置 ... 10
2.9.4　VEE プロジェクトのデータコンフィグレーションのデータ構造 ... 10
2.10　API のデータ構造 ... 12
2.10.1　VEE レコード ... 12
2.11　API の Typedef 定義 ... 12
2.11.1　API の戻り値 ... 12
2.11.2　VEE の状態 ... 12
2.11.3　R_VEE_Control()関数で使用される VEE コマンド ... 13
2.12　MCU に依存する Typedef 定義 ... 13
2.13　VEE レコードにおけるオーバヘッド ... 13
2.14　VEE からのデータ読み出し ... 14
2.15　エラーの回復 ... 15
2.16　ユーザプロジェクトにミドルウェアを追加するには ... 15
2.17　フラッシュエラーの検出 ... 16

3.　API 関数 ... 17
3.1　概要 ... 17
3.2　R_VEE_Read ... 18
3.3　R_VEE_Write ... 19
3.4　R_VEE_Defrag ... 21
3.5　R_VEE_Erase ... 22
3.6　R_VEE_GetState ... 24
3.7　R_VEE_ReleaseState ... 25
3.8　R_VEE_GenerateCheck ... 26
3.9　R_VEE_Open ... 27
3.10　R_VEE_Control ... 28
3.11　R_VEE_GetVersion ... 29

4.　デモンストレーションプロジェクト ... 30
4.1　HEW ワークスペース ... 30
4.2　E2Studio プロジェクト ... 31

1. 概要

図 1.1 に示されているように、仮想 EEPROM (VEE) プロジェクトは、ルネサスが提供しているフラッシュ API の上に位置するソフトウェアレイヤです。

仮想EEPROM（ソフトウェア）

Flash API（ソフトウェア）

MCU（ハードウェア）

図1.1　プロジェクトレイヤ

1.1 バッググランド動作の利用

仮想 EEPROM では、使用される MCU がデータフラッシュでバックグランド動作（BGO）をサポートするハードウェアを有しており、フラッシュ API ソフトウェアが提供されていることが必要とされます。バックグランド動作では、フラッシュの操作が MCU の動作を妨げることがありません。BGO をサポートしていないシステムの場合、フラッシュの動作が開始されると、その動作が終了するまでユーザアプリケーションにはコントロールが戻りません。BGO をサポートしているシステムでは、フラッシュの動作が正常に開始された直後に、ユーザアプリケーションにコントロールが戻されます。動作が終了したときには、コールバック関数により通知されるか、ポーリングによってこれを知ることができます。VEE プロジェクトでは MPU の BGO 機能を利用しており、ユーザアプリケーションに占める時間がより小さくなります。

1.2 レコード

VEE にデータが書き込まれるときには、VEE レコードが使用されます。レコードは格納されたデータへのポインタに加えてユーザによって書き込まれる幾つかの情報を持っています。各レコードには希望する任意の大きさのデータを対応させることができます。ユーザが VEE に対してレコードを書き込むと、データとレコード情報が一緒に格納されます。個々のレコードは一意な ID を持っています。ユーザが前に書き込んだのと同じ ID を持つレコードを書き込んだときには、これが新しいレコードとして書き込まれ、以前のレコードは無効とされます。これは疲労の平均化のために行われています。また、VEE のコードには同一のデータを重複して書き込むような指示を無視する機能を持たせる設定項目が用意されています。レコードがどのように格納されているかはセクション1.4で解説されています。

1.3 データ管理

VEE プロジェクトでは MCU のデータフラッシュ領域は VEE セクタに分割されます。これは MCU の物理的なセクタに対応するものではありません。VEE プロジェクトでは少なくともひとつの VEE セクタが必要です。個々の VEE セクタは少なくとも 2 個の VEE ブロックで構成されます。各 VEE ブロックは 1 個以上の MCU のフラッシュブロックからなっています。何時の時点でも、1 個の VEE ブロックがその VEE セクタに格納される最新のデータを保持しています。

VEE ブロックが一杯になると、ブロック間で有効なデータがピンポンのように前後にやりとりされます。書き込みの際にその時点で使用されているブロックに余裕がないときにはデフラグが実行され、最新のデータがその VEE セクタ内の次の VEE ブロックに転送されます。データの転送が終わると、元の VEE ブロックを消去することができますので、新しい VEE ブロックが一杯になったときの対応が可能になります。このデータの移動でフラッシュ疲労の均等化が実現されます。

複数の VEE セクタを用意することで、ユーザはデータを区分けすることができます。この理由のひとつは、頻繁に書き込まれるデータと書き込みが稀なデータを分離することです。一例として、一日に一回書き込まれる大きなブロックデータと、1 分ごとに書き込まれる小さなデータがある状況を想定します。小さなブロックデータでは早い時点でカレントブロックが一杯になるため頻繁にデフラグが実行され、結果として変更のない大きなブロックデータの転送が必要となります。また、大きなブロックデータは VEE ブロックの利用可能なスペースで比較的大きな領域を占めるため、デフラグを発生させる頻度がより大きくなります。

ユーザはセクション1.2に記されているように、VEE レコードを書き込みます。レコードが仮想 EEPROM に書き込まれると、このデータはレコードが持つ一意な ID を利用して読み出すことができます。ユーザが新しいバージョンのレコードを書き込む際には、以前と同じ ID を使用します。データの大きさが以前と同じである必要はなく、この点には API が対応します。

1.4 VEE 動作の例

このセクションでは、VEE が 1 セクタ 2 ブロックの構成にセットアップされているとき、データがどのように管理されるかを説明します。先に述べられているように、VEE レコードは VEE ブロックに格納されます。VEE セクタを構成するには少なくとも 2 個のブロックが必要です。VEE の何時の時点でも有効な一意 ID に対して 1 個の有効なレコードが 1 個のみ存在します。同じ ID を持つレコードが書き込まれるとき、最新のレコードが有効なレコードとなり、以前に書き込まれたレコードは無視されるようになります。図 1.2は使用時に VEE がどのように見えるかを示しています。各レコードが VEE ブロック 0 で 2 回ずつ現れている点に注意してください。下側のレコード（この例では、レコードは下向きに順次格納されるものとします）のみが有効なものです。

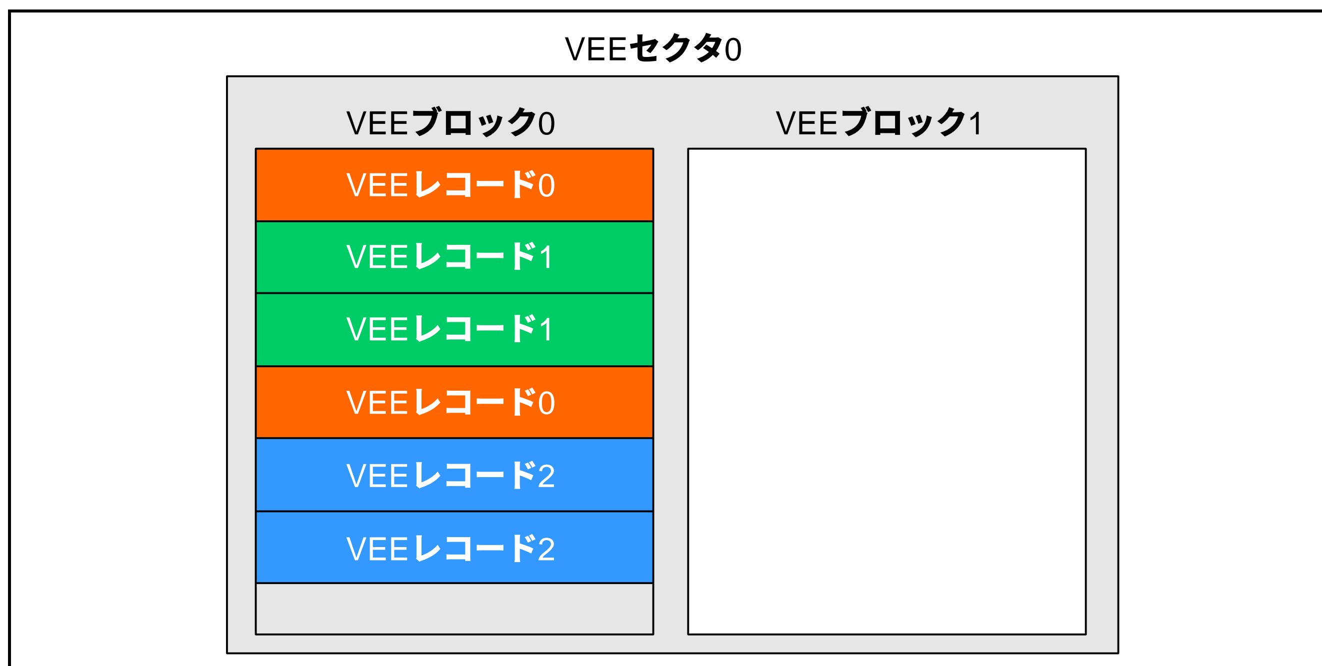


図1.2 ブロック 0 にレコードを格納します。

レコードに対して充分な空がない VEE ブロックに VEE 書き込みが要求されたときには、デフラグが必要となります。デフラグでは全ての有効なレコードを別の VEE ブロックに移動します。VEE レコード 1 を VEE ブロック 0 に書き込めないときの状況が図 1.3に示されています。 これにより全ての有効なレコードを VEE ブロック 1 に移動するデフラグ動作が開始されます。

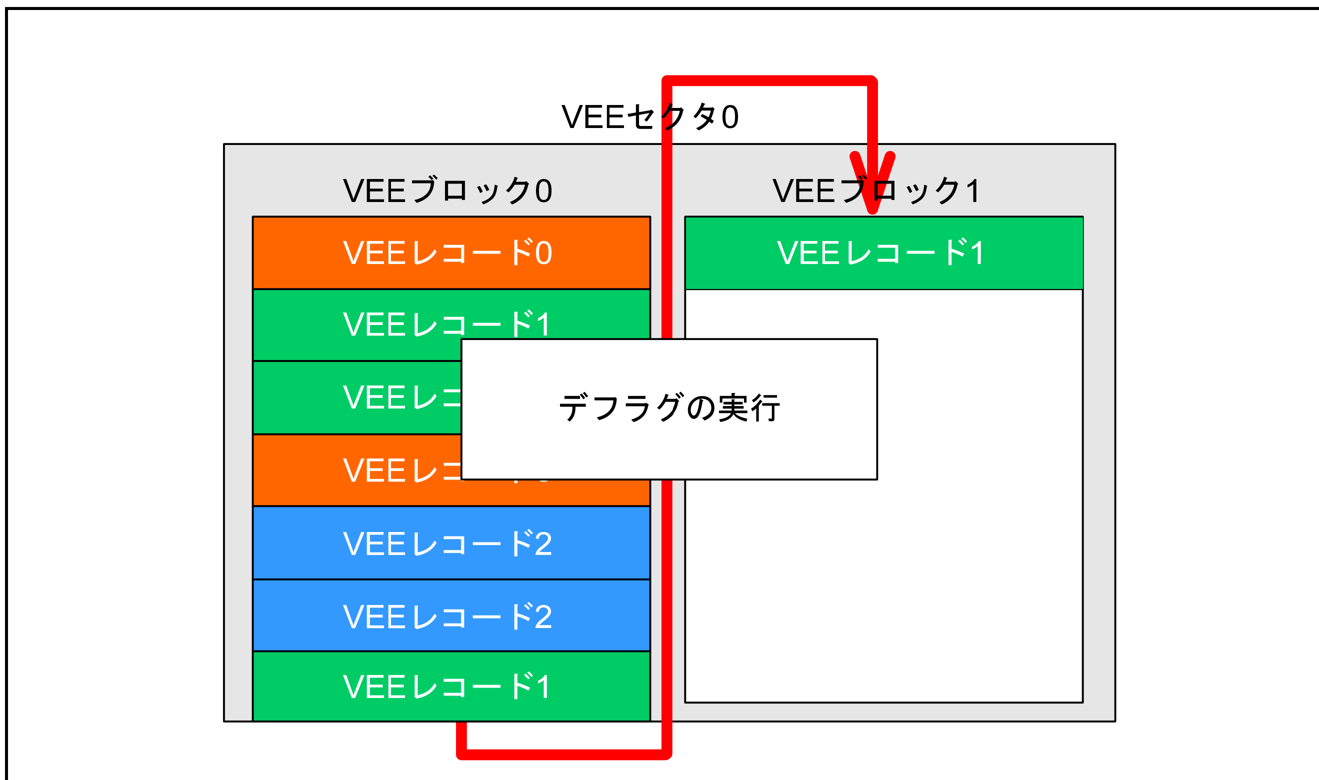


図1.3　デフラグ時にはレコードをブロック 1 に移します。

各レコードで有効なバージョンのみがコピーされていることに注意してください。古いレコードは無視され、後ほど消去されます。

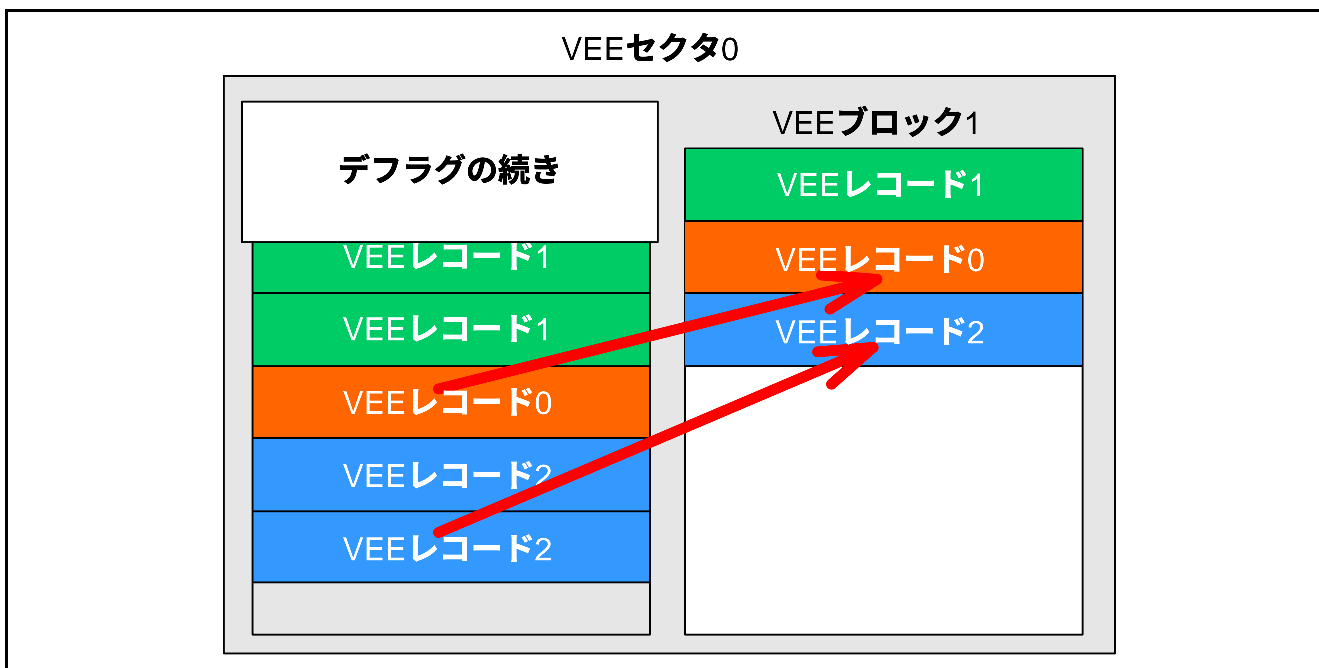


図1.4　デフラグでは有効なレコードのみをコピーします。

デフラグが終了すると、以前の VEE ブロックは消去されます。図 1.5はデフラグと消去が終了した後の VEE の状態を示しています。有効なレコードは全て VEE ブロック 1 に移されており、VEE ブロック 0 が消去されています。この時点では VEE ブロック 0 は消去されており、VEE ブロック 1 でデフラグが必要なときに直ちに使用できるようになっています。

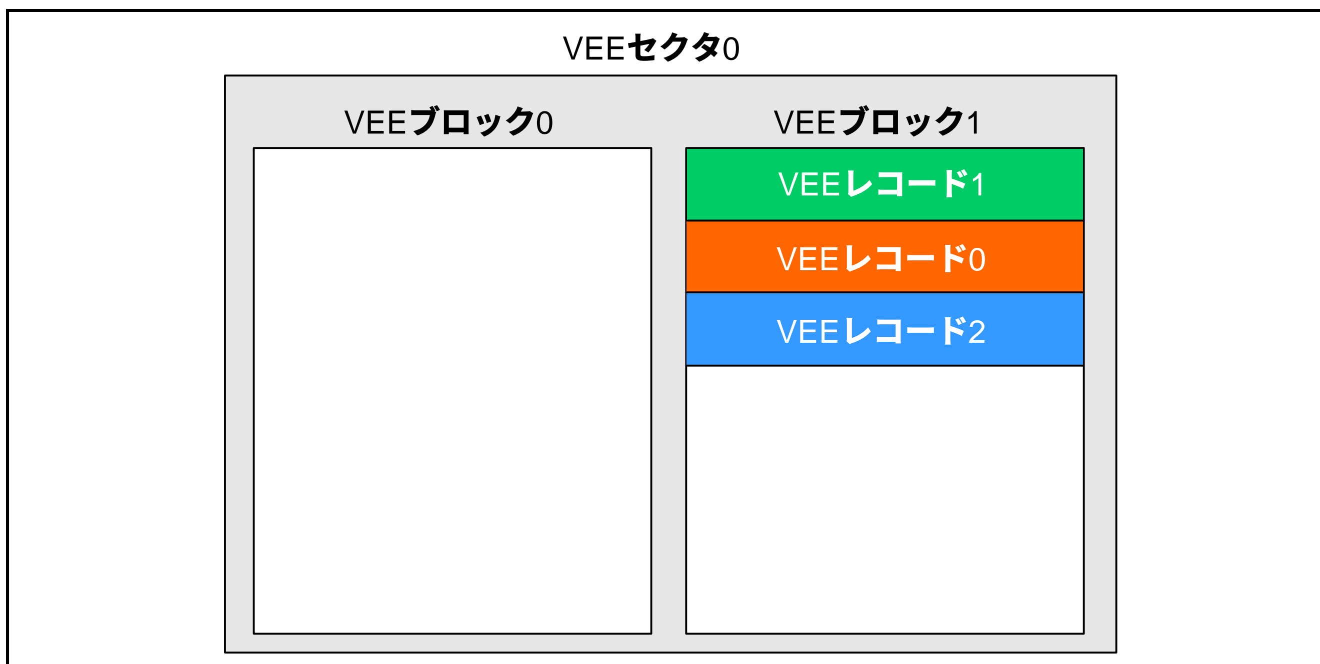


図1.5　デフラグ後にブロック 0 を消去します。

2. API の情報

VEE API はルネサス API ネーミング基準に準拠しています。

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（BGO）が可能なフラッシュメモリ

2.2 ハードウェアリソース要件

このセクションでは、このミドルウェアが必要とする個々の周辺回路について、その詳細を説明します。特に明記されていない限り、ここにあるリソース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から直に使用することはできません。

2.2.1 データフラッシュ

確実な動作のためには MCU のデータフラッシュ全体を VEE 動作のために確保することが必要です。これが不可能な場合には、データフラッシュ上の他の操作が VEE 動作に干渉しないことを保証することはユーザの責任となります。

2.3 ソフトウェア要件

このミドルウェアは次のパッケージに依存しています。

- RX600 用シンプルフラッシュ API (R01AN0544JU) v2.40、もしくはこれ以降のバージョン
- ルネサス FIT ボードサポートパッケージ (r_bsp) v2.00、もしくはこれ以降のバージョン

2.4 制約

フラッシュ API を利用して仮想 EEPROM の外でのデータフラッシュ操作を行うことは可能ですが、VEE 操作の妨げとならないように注意しなければなりません。一例として、フラッシュ API を利用してデータフラッシュへの書き込みを行い、その後、データフラッシュの書き込みが完了する前に VEE レコードを読み込もうとすると、フラッシュエラーが発生します。

2.5 サポートされるツールチェイン

このミドルウェアは次のツールチェインの下でテストされ、動作しています。

- ルネサス RX ツールチェイン v1.02

2.6 ヘッダファイル

全ての API 呼び出しは VEE プロジェクトコードとして提供されている 1 個のファイル r_vee_if.h をインクルードすることによって行われます。このヘッダファイルはユーザの VEE コンフィグレーション情報をもっている r_vee_user_config.h ファイルを参照しています。

2.7 整数型データ

このプロジェクトではコードをわかりやすく、移植性をより大きくするために ANSI C99 の"Exact width integer types"を採用しています。これらのデータ型は stdint.h で定義されています。

2.8 コンフィグレーションの概要

このセクションでは、r_vee_config.h ファイルにある定義と、それぞれの項目が VEE プロジェクトの動作にどのように影響するかについて説明します。

表2.1 コンフィグレーション定義項目の説明

r_vee_config.h ファイルあるコンフィグレーション項目	
VEE_NUM_SECTORS	この定義は使用される VEE セクタの数を指定します。この項目は r_vee_config_<target>_<df_size>.h ファイルにあるセクタ構成の選択を定義しています。例えば、データフラッシュ 32 KB の RX62N が 2 個の VEE セクタを持つ構成で使われるときには、2 セクタ VEE コンフィグレーションが r_vee_config_rx62x_32kb.h で定義されていなければなりません。
VEE_MAX_RECORD_ID	システムで使用される一意のレコード（ID）の数を指定します。ユニーク ID ごとにキャッシュ項目が用意されるため、余分に多くの数を指定することは勧められません。ここで与えた値と同じか、より大きな値のレコード ID を指定して VEE 操作を行おうとしたときには、API はエラーを返します。この定義の値としてはユニーク ID の最大の番号を指定しますが、実際のレコード番号の最大は定義された値 –1 となります。たとえばこのマクロとして 8 が定義されているときには、レコードの数は 8 個ですが、レコード ID は 0 から始まるため、最大のレコード ID は 7 となります。
VEE_IGNORE_DUPLICATE_WRITES	このオプションは VEE に既に格納されているレコードと同じレコード（レコードデータの一致）の書き込みを無視することを指示します。これは VEE のスペースを節約し、デフラグの回数を減少させますが、書き込みの際に既存レコードとの一致を調べるための余分な時間を要します。
VEE_CACHE_FILL_ALL	このコンフィグレーション値は一度にキャッシュを満たすか、一度に 1 レコードずつ埋めてゆくかの選択を可能とします。この定義が有効であれば、読み込み動作が指示されレコードがキャッシュ内に見つからないとき（リセット後の最初の読み出しなど）に、全てのレコードがサーチされキャッシュ全体が一度に埋められます。この定義がコメントアウトされている（定義されていない）ときには、要求されたレコードのみがサーチされます。これは、サーチに要する時間を分散するか、もしくは最初に全てを読み出しておくことでサーチ時間を不要とするかによって決められます。
VEE_USE_DEFAULT_CHECK_FUNCTIONS	このオプションはデフォルトの R_VEE_GenerateCheck() と vee_check_record()関数を使用するか、ユーザ独自の関数を使用するかを選択します。たとえば、デフォルトの静的なフラグチェックではなくチェックサムを使いたい場合には、この定義をコメントアウトし、独自の関数を用意します。このとき、関数名と引数はデフォルトの関数と同じものでなければなりません。

r_vee_config.h ファイルあるコンフィグレーション項目	
VEE_CALLBACK_FUNCTION	**このセクションは 2 つの目的に分かれています。第一は、VEE_CALLBACK_FUNCTION 定義が定義されているか否かで判断します。定義されていなければ、VEE プロジェクトでコールバックは使用されず、操作が終了したか否かはユーザがポーリングにより判断しなければなりません。第二に、これが定義されているときには、値としてユーザが作成したコールバック関数の名前を指定します。たとえば、VEE_CALLBACK_FUNCTION として MyCallback を定義しているときには、ユーザが MyCallback()関数をアプリケーション内に用意することで、VEE 操作の完了時にこの関数が呼び出されます。**

2.8.1　　r_bsp パッケージの使用

VEE ミドルウェアの v1.50 以降ではルネサス FIT ボードサポートパッケージ（r_bsp）を利用しています。r_bsp パッケージには個々の RX ボードに対応するスタートアップコードと MCU の情報が含まれています。VEE のコードは r_bsp パッケージのファイルから必要な MCU 情報を入手します。ユーザが独自のボードを使用される際にも、そのボードの情報を r_bsp パッケージに追加されることをお勧めします。RX ミドルウェアを構築する際の明示された基礎を用意することで、ミドルウェアの移植がより一層容易になります。

2.9　　VEE セクタのコンフィグレーション

このセクションでは r_vee_config_<target>_<df_size>.h ファイル（例えばデータフラッシュが 32 KB の RX62N を使う際のファイル名は r_vee_config_rx62x_32kb.h）を使用する VEE セクタコンフィグレーションに関する事項を説明します。このセクションにある定義とデータ構造により VEE プロジェクトで使用される VEE セクタ、VEE ブロック、および VEE レコードのコンフィグレーションが行われます。データ構造も r_vee_config_<target>_<df_size>.h で定義されていますが、これらが r_vee.c で宣言されるまでは実際の領域は割当てられません。

2.9.1　　VEE セクタ数

使用される VEE セクタの数は r_vee_config.h にある VEE_NUM_SECTORS の#define で指定されます。詳細はセクション2.8をご覧ください。

2.9.2　　VEE レコードの VEE セクタへの割当て

VEE レコードがどの VEE セクタに格納されるかの割当てはコンパイル時に g_vee_RecordLocaion[] 配列で指定されます。VEE の一意 ID ごとにこの配列に 1 個の要素が用意されます。VEE で幾つの一意 ID が使用されるかはセクション2.8で紹介された VEE_MAX_RECORD_ID によって決まります。この配列の要素の値は、レコードがどの VEE セクタに格納されるかを指定しています。以下は 2 個のセクタが存在し、最初の 4 レコードが VEE セクタ 0 に、後の 4 レコードが VEE セクタ 1 に格納される場合の設定例です。

```
const uint8_t g_vee_RecordLocations[VEE_MAX_RECORD_ID] = {
    0,  /* Record 0 will be in sector 0 */
    0,  /* Record 1 will be in sector 0 */
    0,  /* Record 2 will be in sector 0 */
    0,  /* Record 3 will be in sector 0 */
    1,  /* Record 4 will be in sector 1 */
    1,  /* Record 5 will be in sector 1 */
    1,  /* Record 6 will be in sector 1 */
    1,  /* Record 7 will be in sector 1 */
};
```

2.9.3 VEE ブロックの配置

いくつの VEE セクタが使用されるかが定義された後、ユーザはセクタ内の VEE ブロックがメモリのどこに置かれるかを決めなければなりません。このためには 2 個の配列が使用されます。提供されている実例ではデフォルトとして下の例にある配列名が使用されています。

最初の配列のタイプは g_vee_sect#_block_addresses[] という名前で、#にはセクタ番号が入ります。配列の各要素はこのセクタにある個々の VEE ブロックの開始アドレスを定めています。VEE セクタごとに、この配列が 1 個ずつ定義されます。

2 番目の配列は g_vee_sect#_df_blocks[][2] という名前で、この#もセクタ番号です。これは 2 次元配列で、各要素自体が配列となっています。各要素となる配列にはこの VEE ブロックを構成する MCU 上の先頭と最後のデータフラッシュブロックが格納されています。この配列も VEE セクタごとに 1 個ずつ定義されます。

下記の例では、次の設定のシステムを定義しています。

- 2 個の VEE セクタ。
- VEE セクタあたり 2 個の VEE ブロック。
- VEE ブロックあたり 4 個の MCU データフラッシュブロック。
- VEE セクタ 0 はセクタ 1 よりメモリの低位のアドレスに置かれます。

```
/* Sector 0 */
const uint32_t g_vee_sect0_block_addresses[] =
{
    0x100000, /* Start address of VEE Block 0 */
    0x102000  /* Start address of VEE Block 1 */
};

const uint16_t g_vee_sect0_df_blocks[][2] =
{
    {BLOCK_DB0, BLOCK_DB3}, /* Start & end DF blocks making up VEE Block 0 */
    {BLOCK_DB4, BLOCK_DB7}  /* Start & end DF blocks making up VEE Block 1 */
};
/* Sector 1 */
const uint32_t g_vee_sect1_block_addresses[] =
{
    0x104000, /* Start address of VEE Block 0 */
    0x106000  /* Start address of VEE Block 1 */
};
const uint16_t g_vee_sect1_df_blocks[][2] =
{
    {BLOCK_DB8,  BLOCK_DB11}, /* Start & end DF blocks making up VEE Block 0 */
    {BLOCK_DB12, BLOCK_DB15}  /* Start & end DF blocks making up VEE Block 1 */
};
```

データフラッシュブロックの#define 名（BLOCK_DB0 など）は RX 用シンプルフラッシュ API パッケージで定義されています。

2.9.4 VEE プロジェクトのデータコンフィグレーションのデータ構造

g_vee_Sectors 配列は VEE プロジェクトコードがシステムの VEE データコンフィグレーションの情報を得るために使用されるデータ構造です。各要素は VEE セクタを定めるもので、次の情報を保持しています。

- セクタの ID
- このセクタを構成する VEE ブロックの数
- このセクタのサイズ（バイト数）
- セクタ内の各 VEE ブロックの開始 MCU アドレス（セクション2.9.3を参照）
- VEE ブロックあたりの MCU データフラッシュブロック数
- 各 VEE ブロックの先頭および最終 MCU データフラッシュブロック（セクション2.9.3を参照）

下記はサイズの異なる 3 個の VEE セクタを持つ VEE プロジェクトにおけるデータ構造の例です。

```
const vee_sector_t g_vee_Sectors[ VEE_NUM_SECTORS ] =
{
    /* Sector 0 */
    {
        /* ID is 0 */
        0,
        /* There are 2 VEE Blocks in this sector */
        2,
        /* Size of each VEE Block */
        8192,
        /* Starting addresses for each VEE Block */
        (const uint32_t *)g_vee_sect0_block_addresses,
        /* Number of data flash blocks per VEE Block
           (End Block # - Start Block # + 1) */
        4,
        /* Start & end DF blocks making up VEE Blocks */
        g_vee_sect0_df_blocks
    }
    ,
    /* Sector 1 */
    {
        /* ID is 1 */
        1,
        /* There are 2 VEE Blocks in this sector */
        2,
        /* Size of each VEE Block */
        6144,
        /* Starting addresses for each VEE Block */
        (const uint32_t *)g_vee_sect1_block_addresses,
        /* Number of data flash blocks per VEE Block
           (End Block # - Start Block # + 1) */
        3,
        /* Start & end DF blocks making up VEE Blocks */
        g_vee_sect1_df_blocks
    }
    ,

    /* Sector 2 */
    {
        /* ID is 2 */
        2,
        /* There are 2 VEE Blocks in this sector */
        2,
        /* Size of each VEE Block */
        2048,
        /* Starting addresses for each VEE Block */
        (const uint32_t *)g_vee_sect2_block_addresses,
        /* Number of data flash blocks per VEE Block
           (End Block # - Start Block # + 1) */
        1,
        /* Start & end DF blocks making up VEE Blocks */
        g_vee_sect2_df_blocks
    }

    /* To add more sectors copy the one above and change the values */
};
```

2.10 API のデータ構造

2.10.1 VEE レコード

API 関数を使用して VEE との間で読み出しや書き込みを行う際には、VEE レコードデータ構造体でデータを渡します。この構造体は次のように r_vee_if.h で定義されています。

```
/* VEE Record Structure */
typedef struct
{
    /* Unique record identifier, cannot be 0xFF! */
    vee_var_data_t   ID;
    /* Number of bytes of data for this record */
    vee_var_data_t   size;
    /* Valid or error checking field */
    vee_var_data_t   check;
    /* Which VEE Block this record is located in, user does not set this */
    vee_var_data_t   block;
    /* Pointer to record data */
    uint8_t    far  * pData;
} vee_record_t;
```

2.11 API の Typedef 定義

2.11.1 API の戻り値

VEE API 関数から返される値は r_vee_if.h にある typedef で次のように定義されています。

```
/* Return values for functions */
typedef enum
{
    VEE_SUCCESS,
    VEE_FAILURE,
    VEE_BUSY,
    VEE_NO_ROOM,
    VEE_NOT_FOUND,
    VEE_ERROR_FOUND
} vee_return_values_t;
```

2.11.2 VEE の状態

VEE がとりうる状態は下記に示されており、この定義は r_vee_if.h で行われています。これらの状態のいずれかが R_VEE_GetState() 関数から返されます。

```
/* Defines the possible states of the VEE */
typedef enum
{
    VEE_READY,
    VEE_READING,
    VEE_WRITING,
    VEE_ERASING,
    VEE_DEFRAG,
    VEE_ERASE_AND_DEFRAG,
    VEE_WRITE_AND_DEFRAG,
    VEE_ERASE_AND_WRITE
} vee_states_t;
```

2.11.3 R_VEE_Control()関数で使用される VEE コマンド

R_VEE_Control() 関数を使用する際には実行するコマンドを与えなければなりません。この typedef では使用できるコマンドを定義しています。

```
/* VEE Record Structure */
typedef enum
{
    /* This command will reset the VEE even if it is in the middle of an
       operation. This should only be used when a flash error (e.g. data flash
       access during VEE operation) has occurred and you need to return the VEE
       to a working state. */
    VEE_CMD_RESET
} vee_command_t;
```

2.12 MCU に依存する Typedef 定義

VEE プロジェクトには、使用される MCU のデータフラッシュの特性によって変更される 2 個の typedef が存在します。これらは vee_var_min_t と vee_var_data_t で、使用されているプロジェクトのヘッダファイル r_vee.h の中にあります。

vee_var_min_t はデータフラッシュの最小書き込みサイズに設定されなればなりません。たとえば、RX62N グループでは 8 バイト単位の書き込みが可能なため uint64_t（8 バイト整数）が使用されます。RX63N では 2 バイト単位の書き込みが可能なため vee_var_min_t として uint16_t（2 バイト整数）が使用されます。

vee_var_data_t の typedef はセクション2.10にあるように VEE レコード構造体で使用されます。vee_var_data_t のサイズは vee_var_min_t と同じか、より大きくなければなりません。vee_var_data_t を大きくすると、VEE レコードあたりのオーバヘッドとなるバイト数が増加します。この詳細はセクション2.13に記されています。したがって vee_var_data_t としては、このシステムで使用できる最小のデータ型とするべきです。他方、必要な値の範囲（特にデータバイト数の指定）を満たすよう、vee_var_data_t をデータフラッシュの最小書き込みサイズより大きなサイズとすることが妥当な場合もあります。一例として、R8/38C では vee_var_data_t の typedef を uint8_t（1 バイト整数）とすることも可能ですが、この場合には VEE レコードにおけるデータの大きさは最大 255 バイト(0xFF)に制限されてしまいます。

次は R8C/3x グループの場合の例です。

```
/* Set size of vee_var_data_t to the minimum write size of MCU's data flash
   or larger. This is the size of the variables in a record structure. */
typedef uint16_t vee_var_data_t;
/* Set size of vee_var_min_t to the minimum write size of MCU's data flash */
typedef uint8_t vee_var_min_t;
```

2.13 VEE レコードにおけるオーバヘッド

VEE レコードにおけるオーバヘッドはセクション2.10にあるデータ構造 vee_record_t を見ればご理解いただけると思われます。格納されるユーザデータのほかに 4 個の vee_var_data_t 型の要素が同時に書き込まれます。つまり、VEE レコードのオーバヘッドは 4 * sizeof(*vee_var_data_t*) です。一例としてセクション2.12で R8C/3x に関するデフォルト定義で見られるように、vee_var_data_t が 2 バイトに設定されていますので、レコードごとのオーバヘッドは 8 バイトとなります。また RX62N では vee_var_data_t が 8 バイトに設定されていますので、レコードごとのオーバヘッドは 32 バイトとなります。

2.14　VEE からのデータ読み出し

VEE からレコードを読み出すには、API 関数 R_VEE_Read (vee_record_t * vee_temp) を使用します。指定されたデータがあれば、VEE は格納されているデータのデータフラッシュ上のアドレスを pData にセットします。ユーザはこのポインタを利用して実際のデータを読むことができます。忘れてはいけない重要な点は、リードを行った後、別のリード以外の他の API 関数が実行される前に必ず R_VEE_ReleaseState() 関数を呼び出さねばならないことです。これは確実な動作のために必要とされています。これが必要な理由は、BGO データフラッシュ操作を使用しているとき、データフラッシュへのアクセスを排他的に行う以外に、データが確実に読み出されることを保証する手段がないためです。次の例は、この手順を踏まないときに問題が起こりうることを示しています。

1. VEE レコード 0 に対する R_VEE_Read() を呼び出し、データフラッシュ内のデータのアドレスを得ます。
2. VEE レコード 0 のデータを読み出します。
3. VEE に VEE レコード 1 を書き込むために R_VEE_Write() を呼び出します。
4. R_VEE_Write() 関数は正常終了で戻り、BGO 機能により、書き込みがバックグランドで進行します。
5. 先に得られたアドレスを使用して、VEE レコード 0 のデータを再び読み出します。
6. VEE レコード 1 の書き込みが終了する前にデータフラッシュで読み出しを行おうとしたために、データフラッシュのアクセス違反エラーが発生します。

同じような状況は VEE の消去やデフラグでも生じる可能性があります。このような問題の発生を避けるために役立つ、安全のための予防策が備えられています。組み込まれている対策と同時に、ユーザも今何を行っているかに注意を払わねばなりません。たとえば、VEE にとって、データフラッシュの操作中にユーザが以前に入手した VEE レコードのデータポインタを利用することを防ぐ手立てはありません。ユーザがこのような状況を自ら避けるために、次のコマンドのいずれかの後にデータの読み出し、もしくは再読み出しを行う場合、常に R_VEE_Read() コマンドを使用するよう注意を払わねばなりません。

- R_VEE_Write()
- R_VEE_Erase()
- R_VEE_Defrag()

つまり、VEE レコード 0 を以前に読み込んでいたとしても、R_VEE_Write(ID=2)コマンドを発した後では、データを再び読み込む前に R_VEE_Read(ID=0)コマンドを使用しなければなりません。次の例は、どのように VEE からデータを読み込み、使用すべきかと、API 関数がどのように確実な動作を行おうとしているかを示しています。

操作	結果
R_VEE_Read(ID=1)	**正常終了**
R_VEE_Write(ID=2)	**異常終了、**VEE_BUSY **が返されます。**
R_VEE_Read(ID=2)	**正常終了**
前の R_VEE_Read(ID=1)**と** R_VEE_Read(ID=2)**によって得られたポインタを利用して、**VEE **レコード** 1 **および** 2 **のデータを参照。**	**正常終了**
R_VEE_Erase(…)	**異常終了、**VEE_BUSY **が返されます。**
R_VEE_ReleaseState()	**正常終了**
R_VEE_Erase(…)	**正常終了**
R_VEE_Write(ID=3)	**正常終了**
R_VEE_Read(ID=2)	**正常終了**
直前の R_VEE_Read(ID=2)**で得られたポインタを利用して、**VEE **レコード** 2 **のデータを参照。**	**正常終了**

2.15　エラーの回復

R_VEE_Write()もしくはR_VEE_Defrag() API関数が使用されるとき、APIは指定されたVEEセクタのエラーの有無を検査します。VEE動作中にリセットもしくは電源断が生じたときにエラーが発生します。VEEの書き込み動作中にリセットが起こると、データフラッシュ内のレコードデータは途中で中断されたようになります。VEEではVEEレコード構造体のcheck要素を使用し、書き込みが完了していないレコードをユーザが読み込むことがないように保護しています。check要素は最後に書き込まれ、VEEで読み出し操作が行われる際に検査されます。

VEEプロジェクトで組み込まれているエラー回復のメカニズムは、できるだけ多くのデータの回復を試みます。最後に書き込まれたデータが失われる場合があります。このときの状況は次のようになります。

1. VEEレコード0がR_VEE_Write() を使用して書き込まれます。
2. VEEレコード0を格納する充分なスペースがないため、デフラグが行われます。
3. デフラグは新しいVEEブロックに最初にVEEレコード0を書き込むことで開始されます。
4. VEEレコード0が書き込まれた後、デフラグが完了する前にリセットが発生します。
5. ユーザはR_VEE_Read()を使用してVEEレコード0を読み込もうとします。

この時点で、二通りの方策が考えられます。最初の選択肢は、リセット発生前にデフラグされた元のVEEブロックに存在するレコードを、最後に書き込んだVEEレコード0として返すという方法です。もうひとつの選択肢は、新しいVEEブロックにあるより新しいレコードを見つけるというものです。単純さと速度を優先する場合には、最初の選択肢を採ります。この際の主な問題点は、デフラグ状態が再び生じ、無効とされたVEEブロックが消去さるかも知れないため、より新しいレコードを格納している確実な場所が存在しないことです。レコードをRAMに格納することはできますが、これはユーザがVEEのために書き込むかもしれないと同時に通常はまず利用されないであろう、レコードの最大長のRAM領域を確保しなければなりません。この方法によっても、レコードはRAMに保存されているため、パワーダウンが再び発生するとレコードが失われてしまいます。このときレコードは完全に失われます。

R_VEE_Read() 関数はVEEセクタが破損しているかのチェックを行いません。これはもっともすばやくレコードを返すことができることを意味します。この点はパワーアップ時にデータをできるだけ早く必要としているユーザにとっては特に重要なことです。問題のVEEセクタに次の書き込みが要求されたときに、VEEはエラーの有無を検出し、新しいブロックを消去した後にデフラグ操作を再び開始します。

2.16　ユーザプロジェクトにミドルウェアを追加するには

下記の手順で、VEEコードをユーザのプロジェクトに追加することができます。この手順では、フラッシュAPIが既にユーザプロジェクトに組み込まれていることを想定しています。

1. r_veeディレクトリ（このアプリケーションノートに同梱されています）をユーザプロジェクトのディレクトリにコピーします。
2. ユーザプロジェクトにr_vee.cファイルを加えます。
3. ユーザプロジェクトに、使用されるMPU移植用のC言語のソースファイルをsrc/targets/ディレクトリから追加します。
    a. RX62xのときには、このファイルはr_vee_rx62x.cという名前で、r_vee/src/targets/rx62xフォルダにあります。
4. ユーザプロジェクトに、r_veeディレクトリへのインクルードパスを追加します。
5. ユーザプロジェクトに、r_vee/srcディレクトリへのインクルードパスを追加します。
6. コンフィグレーションファイルの見本r_vec_config_reference.hをrefフォルダからユーザプロジェクトにコピーし、r_vee_config.hに名前を変更します。
7. r_vee_config.hファイルを利用してミドルウェアのコンフィグレーションを行います。
8. VEE APIを利用しているすべてのソースファイルにr_vec_if.hへの#includeを追加します。

2.17 フラッシュエラーの検出

データフラッシュでエラーが発生したときには、コールバック関数 FlashError()を使用して警告がなされます。これはフラッシュ API が使用しているのと共通のコールバック関数です。フラッシュ API はコールバック関数を呼び出す前に MCU のフラッシュコントロールユニットをリセットしています。これはコールバック関数ですので、ユーザはアプリケーションの中で関数を用意しなければなりません。この関数のプロトタイプは次のようになります。

```
void FlashError(void);
```

3. API 関数

3.1 概要

この API には次の関数が含まれています。

関数名	概要
R_VEE_Read()	VEE からレコードを読み出します。
R_VEE_Write()	VEE にレコードを書き込みます。
R_VEE_Defrag()	その時点の VEE セクタのデフラグを行います。
R_VEE_Erase()	VEE セクタを消去します。
R_VEE_GetState()	その時点の VEE の処理状態を取得します。
R_VEE_ReleaseState()	この関数は VEE のリード状態を解除し、以後のリード以外の VEE 操作を可能とします。
R_VEE_GenerateCheck()	入力されるレコードの check 要素を生成します。
R_VEE_Open()	VEE のデータ構造体と内部状態を初期化します。
R_VEE_Control()	さまざまな操作のための拡張可能な VEE 関数です。
R_VEE_GetVersion()	VEE コードのバージョンを返します。

3.2 R_VEE_Read

VEE にある指定されたレコードを探します。

フォーマット

```
uint8_t R_VEE_Read(vee_record_t * vee_temp);
```

パラメータ

vee_temp 探す対象となるレコードのレコード構造体のポインタ。.

戻り値

VEE_SUCCESS:	正常終了、レコード構造体の要素には値がセットされます。
VEE_NOT_FOUND:	レコードが見つかりません。
VEE_BUSY:	他の VEE 操作が進行中です。後ほど改めて試みてください。
VEE_INVALID_INPUT:	引数として渡されたレコード構造体が有効でないデータを持っています。

プロパティ

プロトタイプは r_vee_if.h ファイルにあり、関数の実体は r_vee.c ファイルに実装されています。

説明

この関数は VEE からレコードを検索します。ユーザは要素 ID に見つけたいレコードの値を設定した VEE レコード構造体を渡します。VEE は最初に VEE キャッシュをサーチします。レコードがキャッシュ内に見つからないときには、データフラッシュ内をサーチします。データフラッシュに所定のレコードが見つかると、その後の読み出しのために、そのアドレスがキャッシュに格納されます。

リエントラント

リエントラントです。ただし、VEE の書き込み、デフラグ、消去操作が行われていない場合にのみ実行できます。これらのいずれかの操作が進行中の時には VEE_BUSY が戻されます。

使用例

```
vee_record_t example_record;

/* We want to find VEE Record 1 */
example_record.ID = 1;

/* Search VEE for record */
if (VEE_SUCCESS == R_VEE_Read(&example_record))
{
    /* Send data */
    for (loop = 0; loop < example_record.size; loop++)
    {
        TransmitByte(example_record.pData[loop]);
        ...
    }
}
```

注意:

VEE セクタの記録に誤りがあっても、この関数は回復作業を開始しません。たとえば、VEE の書き込み、消去、もしくはデフラグ中にリセットや電源断が生じた場合、VEE システムは破損した状態となることがあります。MCU の電源が回復して VEE の読み出しが要求されたとき、破損したシステムに対して読み出しが指示されます。この関数は破損したシステムを無視し、指定された ID を持つ最新の有効な VEE レコードを読み出そうとします。この関数がこのように働く理由は、リセット後にユーザがデータをできるだけ早く読み出すことを可能とするためです。仮にこの関数で回復処理を開始した場合には、VEE システムが修復されるまでアプリケーションがストール状態となってしまいます。

3.3 R_VEE_Write

VEE にレコードを書き込みます。

フォーマット

```
uint8_t R_VEE_Write(vee_record_t * vee_temp);
```

パラメータ

vee_temp　　書き込むレコードのレコード構造体へのポインタ。

戻り値

VEE_SUCCESS:	正常終了、書き込みが進行中です。
VEE_BUSY:	他の VEE 操作が進行中です。後ほど改めて試みてください。
VEE_NO_ROOM:	充分なスペースの余裕がありません。R_VEE_Erase()を呼ぶ必要があります。
VEE_FAILURE:	データフラッシュの操作が失敗しました。
VEE_INVALID_INPUT:	引数として渡されたレコード構造体が有効でないデータを持っています。

プロパティ

プロトタイプは r_vee_if.h ファイルにあり、関数の実体は r_vee.c ファイルに実装されています。

説明

この関数は VEE レコードをデータフラッシュに書き込むために使用されます。VEE レコード構造体を渡すときには、ユーザは構造体の次の要素に値を設定しなければなりません。

- ID
- size
- check
- pData

関数が VEE_SUCCESS を返したとき、レコードはまだ書き込まれていません。書き込みは進行中です。ユーザが VEE コールバック関数の使用を選択したときには、書き込みが終了したときにコールバック関数が呼ばれます。もしくは、ユーザは VEE 状態のポーリングのために R_VEE_GetState() 関数を利用することができます。レコードが書き込まれたとき、その後の検索をすばやく行うため、このレコードは自動的に VEE キャッシュに登録されます。

リエントラント

リエントラントではありません。ただし、関数を同時に複数回呼び出すことによる誤動作を避けるためのロック機構によって保護されています。

使用例

```
vee_record_t example_record;

/* Fill in data for VEE Record 1 */
example_record.ID = 1;
example_record.size = sizeof(record_data);
example_record.pData = &record_data[0];

/* Generate 'check' field */
R_VEE_GenerateCheck(&example_record);

/* Write record */
if (VEE_SUCCESS == R_VEE_Write(&example_record))
{
    ...
}
```

注意:

この関数は書き込み対象となる VEE セクタのエラーの有無を検査し、エラーが見つかったときには、その回復を試みます。復旧動作が必要とされた場合には API は VEE_BUSY を返します。このときユーザはこの書き込みを後ほど改めて要求しなければなりません。

3.4 R_VEE_Defrag

VEE のセクタのデフラグを行います。

フォーマット

```
uint8_t R_VEE_Defrag(uint8_t sector);
```

パラメータ

sector　デフラグされる VEE セクタの ID を指定します。

戻り値

VEE_SUCCESS:　正常終了、デフラグが進行中です。

VEE_BUSY:　他の VEE 操作が進行中です。後ほど改めて試みてください。

VEE_NOT_FOUND:　デフラグのための ACTIVE ブロックが見つかりません。

VEE_INVALID_INPUT:　セクタ ID の指定が有効ではありません。

プロパティ

プロトタイプは r_vee_if.h ファイルにあり、関数の実体は r_vee.c ファイルに実装されています。

説明

この関数はセクタをデフラグするために使用されます。デフラグは R_VEE_Write()が呼ばれ、アクティブな VEE ブロックに空きスペースがないときに自動的に実行されます。ユーザはアイドル時に強制的にデフラグを行うために、この関数を呼ぶことができます。これにより、多忙な「書き込み」時にデフラグが生じる可能性を減らすことができます。

関数が VEE_SUCCESS を返したときには、デフラグはまだ終了おらず、デフラグ処理を実行中です。ユーザが VEE コールバック関数の使用を選択したときには、デフラグが終了したときにコールバック関数が呼ばれます。もしくは、ユーザは VEE 状態のポーリングのために R_VEE_GetState() 関数を利用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。ただし、関数を同時に複数回呼び出すことによる誤動作を避けるためのロック機構によって保護されています。

使用例

```
uint8_t sector;

for (sector = 0; sector < VEE_NUM_SECTORS; sector++)
{
     /* Defrag sector */
    ret = R_VEE_Defrag(sector);

    /* Check result */
    if (VEE_SUCCESS == ret)
    {
        ...
    }

    /* Wait for defrag to finish */
    ...
}
```

注意:

この関数は対象となっている VEE セクタでエラーの有無を検査し、エラーが見つかったときには、その回復を試みます。復旧動作が必要とされた場合には API は VEE_BUSY を返します。このときユーザはこのデフラグ要求を後ほど改めて行わねばなりません。

3.5 R_VEE_Erase

VEE のセクタを消去します。

フォーマット

```
uint8_t R_VEE_Erase(uint8_t sector);
```

パラメータ

sector　　消去される VEE セクタの ID を指定します。

戻り値

VEE_SUCCESS:	正常終了、消去は進行中です。
VEE_BUSY:	他の VEE 操作が進行中です。後ほど改めて試みてください。
VEE_FAILURE:	データフラッシュの操作が失敗しました。
VEE_INVALID_INPUT:	セクタ ID の指定が有効ではありません。

プロパティ

プロトタイプは r_vee_if.h ファイルにあり、関数の実体は r_vee.c ファイルに実装されています。

説明

この関数は、VEE セクタにあるデータを消去するために使用されます。指定された VEE セクタにアクティブな VEE ブロックが見つからないときには VEE_SUCCESS が返されます。アクティブな VEE ブロックが見つかると、このブロックが消去されます。この関数は VEE ブロックが空であるか否かの判定に VEE ブロックのフラグを利用しており、VEE ブロックのメモリ領域全体をチェックしているわけではありません。

関数が VEE_SUCCESS を返したときには、消去はまだ終了おらず、消去処理が進行中です。ユーザが VEE コールバック関数の使用を選択したときには、消去が終了したときにコールバック関数が呼ばれます。もしくは、ユーザは VEE 状態のポーリングのために R_VEE_GetState() 関数を利用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。ただし、関数を同時に複数回呼び出すことによる誤動作を避けるためのロック機構によって保護されています。

使用例

```
uint8_t sector;

/* Erase all data from VEE */
for (sector = 0; sector < VEE_NUM_SECTORS; sector++)
{
     /* Erase sector */
    ret = R_VEE_Erase(sector);

    /* Check result */
    if (VEE_SUCCESS == ret)
    {
        ...
    }

    /* Wait for erase to finish */
    ...
}

/* VEE is empty */
```

注意:

VEE は、レコードが格納されている VEE セクタが消去されたとき、ユーザが自分のプログラム内に持っている VEE レコード構造体を無効化する手段をもっていません。これにより生じるエラーを避けるために、ユーザは VEE セクタが消去されたときにはデータポインタを使用してデータを読むことのないように注意しなけ

ればなりません。もしくは消去後には常に R_VEE_Read() 関数を使用することでデータが有効であることを確認する必要があります。

3.6 R_VEE_GetState

その時点の VEE の状態を返します。

フォーマット

```
vee_states_t R_VEE_GetState(void);
```

パラメータ

なし

戻り値

VEE の状態。VEE のとりうる状態に関する情報はセクション2.11.2を参照してください。

プロパティ

プロトタイプは r_vee_if.h ファイルにあり、関数の実体は r_vee.c ファイルに実装されています。

説明

この関数は、その時点の VEE の状態を返します。この関数を使用し VEE 操作が終了したことを判定するためのポーリングを行うことができます。

リエントラント

リエントラントです。

使用例

```
uint8_t sector;

/* Erase all data from VEE */
for (sector = 0; sector < VEE_NUM_SECTORS; sector++)
{
    /* Erase sector */
    ret = R_VEE_Erase(sector);

    /* Check result */
    if (VEE_SUCCESS == ret)
    {
        ...
    }

    while (VEE_READY != R_VEE_GetState())
    {
        /* Wait for erase to finish */
    }
}

/* VEE is empty */
```

注意:

なし

3.7 R_VEE_ReleaseState

読み出しが正常に行われた後、この関数は VEE の状態を VEE_READY に設定し、以後の VEE 操作を可能とします。

フォーマット

```
uint8_t R_VEE_ReleaseState(void);
```

パラメータ

なし

戻り値

VEE_SUCCESS: 正常終了、VEE の状態が VEE_READING から VEE_READY に解放されました。

VEE_FAILURE: 状態が VEE_READING の時にのみ状態を解放できます。

プロパティ

プロトタイプは r_vee_if.h ファイルにあり、関数の実体は r_vee.c ファイルに実装されています。

説明

この関数は VEE の状態を解放し、他の VEE 操作を可能とします。ユーザが R_VEE_Read() 関数を使用して VEE からレコードを読んだ後には、この関数を呼び出す必要があります。この必要性に関しては、セクション2.14を参照してください。この関数は読み出しが正常に終了した後にのみ呼び出すことができます。ユーザが VEE による書き込みやデフラグ、消去の後にこの関数を呼んだときには、関数は VEE_FAILURE を返します。

リエントラント

リエントラントです。

使用例

```
vee_record_t example_record;
uint8_t ret;

/* We want to find VEE Record 1 */
example_record.ID = 1;

/* Search VEE for record */
if (VEE_SUCCESS == R_VEE_Read(&example_record))
{
    /* Read data and use it */
    ...
}

/* Release state so other VEE operations can occur */
ret = R_VEE_ReleaseState();
```

注意:

なし

3.8 R_VEE_GenerateCheck

レコードの check フィールドを生成します。

フォーマット

```
uint8_t R_VEE_GenerageCheck(vee_record_t * record);
```

パラメータ

record　　check フィールド生成の対象となるレコードを持つ構造体のポインタを渡します。

戻り値

VEE_SUCCESS:　　正常終了、check フィールドに値が格納されています。

VEE_FAILURE:　　引数として渡されたレコード構造体が有効ではありません。

プロパティ

プロトタイプは r_vee_if.h ファイルで定義されています。

関数の実体は各 MCU のポートソースファイル（r_vee_rx62x.c など）に実装されています。

説明

この関数は指定されたレコードに対応する check フィールドの値を生成します。デフォルトでは VEE はチェック目的で単に定数値のフラグを使用しています。これはレコードの書き込みが正しく行われたか否かを VEE が確認する目的で使用されています。アプリケーションで CRC やチェックサムを利用する場合、必要に応じてこの関数を書き換えることができます。

リエントラント

リエントラントです。

使用例

```
vee_record_t example_record;

/* Fill in data for VEE Record 1 */
example_record.ID = 1;
example_record.size = sizeof(record_data);
example_record.pData = &record_data[0];

/* Generate 'check' field */
R_VEE_GenerateCheck(&example_record);

/* Write record */
if (VEE_SUCCESS == R_VEE_Write(&example_record))
{
    ...
}
```

注意:

ユーザが独自のエラーチェック（CRC など）を導入するためにこの関数を修正するときには、同時に vee_check_record() 関数も変更しなければなりません。R_VEE_GenerateCheck() 関数はユーザが check フィールドを生成するために利用されます。vee_check_record() 関数は VEE が内部的にレコードの破損をチェックするために使用します。vee_check_record() 関数が変更されていないと、すべてのレコードが破損していると認識される可能性があります。

3.9 R_VEE_Open

VEE が使用しているデータ構造と内部の状態を初期化します。

フォーマット

```
uint8_t R_VEE_Open(void);
```

パラメータ

なし

戻り値

VEE_SUCCESS:　　正常終了

プロパティ

プロトタイプは r_vee_if.h ファイルにあり、関数の実体は r_vee.c ファイルに実装されています。

説明

この関数は VEE 内部の状態を初期化します。また、VEE 内のレコードキャッシュも無効化されます。

リエントラント

リエントラントです。

使用例

```
/* Initialize the Virtual EEPROM */
R_VEE_Open();
```

注意:

なし

3.10 R_VEE_Control

さまざまな操作のための拡張可能な VEE 関数です。

フォーマット

```
uint8_t R_VEE_Control(vee_command_t command, void * pdata);
```

パラメータ

command　実行されるコマンドを指定します。

pdata　コマンドに渡されるデータ、コマンドから戻されるデータ、もしくはその双方。

戻り値

VEE_SUCCESS:　正常終了

VEE_BUSY:　他の VEE 操作が進行中です。後ほど改めて試みてください。

VEE_INVALID_INPUT:サポートされていないコマンド、もしくは適切でない入力データが渡されました。

プロパティ

プロトタイプは r_vee_if.h ファイルにあり、関数の実体は r_vee.c ファイルに実装されています。

説明

この関数は VEE 内部でさまざまな操作を実行するために使用されます。これらの操作は通常はユーティリティ関数で使用されるもので、拡張性を持たせるためにこの関数にまとめられています。この関数は実行すべき操作を指定するコマンドを持っています。もうひとつのパラメータはコマンドに対しての入力データ、出力データ、もしくはその両方として使用できます。使用可能なコマンドに関しては r_vee_if.h ファイルにある vee_command_t の typedef 定義をご覧ください。これらのオプションに関してはセクション2.11.3でも述べられています。

リエントラント

リエントラントです。

使用例

```
/* A data flash access violation occurred and the VEE is in a locked state.
   Reset the VEE to start recovery. */
if (VEE_SUCCESS == R_VEE_Control(VEE_CMD_RESET, (void *)FIT_NO_PTR))
{
    /* VEE has been reset. The next write or defrag of the VEE will start any
       needed recovery operations. */
    ...
}
```

注意:

FIT の r_bsp を使用している場合、pdata 引数が不要のコマンドでは上の使用例のように FIT_NO_PTR マクロを使用することが推奨されます。

3.11 R_VEE_GetVersion

使用されている VEE のバージョン情報を戻します。

フォーマット

```
uint32_t R_VEE_GetVersion(void);
```

パラメータ

なし

戻り値

VEE のバージョン情報

プロパティ

プロトタイプは r_vee_if.h ファイルにあり、関数の実体は r_vee.c ファイルに実装されています。

説明

この関数はインストールされているフラッシュ API のバージョンを戻します。バージョン番号は上位 2 バイトでメジャー番号を、下位 2 バイトでマイナー番号を示します。例えば、バージョン 4.25 であれば 0x00040019 が戻されます。

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 VEE. */
cur_version = R_VEE_GetVersion();

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

注意:

この関数は r_vee.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。

4. デモンストレーションプロジェクト

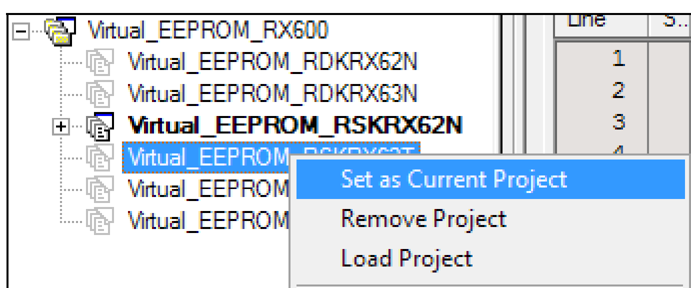
このアプリケーションノートには HEW と E2Studio の両方に対するデモンストレーションプロジェクトが含まれています。HEW の場合には、デモンストレーションは各ルネサス開発ボードのためのプロジェクトを含む完全な HEW ワークスペースの形のパッケージとなっています。E2Studio の場合には、各ルネサス開発ボードのそれぞれに対して、既存の E2Studio ワークスペースにインポート可能なジッププロジェクトが用意されています。VEE パッケージの今回のバージョンには以下のボードに対応するプロジェクトが含まれています。

- RSKRX62N
- RSKRX63N
- RDKRX63N
- RSKRX62G
- RSKRX63T_144PIN
- RSKRX62T
- RSKRX630
- RDKRX62N
- RSKRX63T_64PIN
- RSKRX210

4.1 HEW ワークスペース

このアプリケーションノートパッケージの HEW ワークスペースには、サポートされている個々のルネサス開発ボードのためのプロジェクトが用意されています。これらのプロジェクト間の相違は、VEE コードとデモコードで使用されているボードサポートコードのみです。プロジェクトは次の手順で選択します。

1. HEW ワークスペースを開きます。
2. ナビゲーションペイン(デフォルトでは左にあります)でロードしたいプロジェクトを右クリックし、次に Set as Current Project をクリックします。



3. VEE API コードとデモンストレーションのワークスペースでは、スタートアップコードとボードサポートコード、および MCU 情報を r_bsp パッケージから得ています。r_bsp パッケージは r_bsp フォルダにある plotform.h ヘッダファイルを通して容易にコンフィグレーションを行うことができます。r_bsp パッケージのコンフィグレーションでは、platform.h ファイルを開き、使用するボードに関する定義のコメントを外します。一例として、RSK+RX63N ボードでデモンストレーションを実行するには、./board/rskrx63n/r_bsp.h ファイルの#include のコメントを外し、同時に他のボードの#include がすべてコメントとしてあることを確認してください。

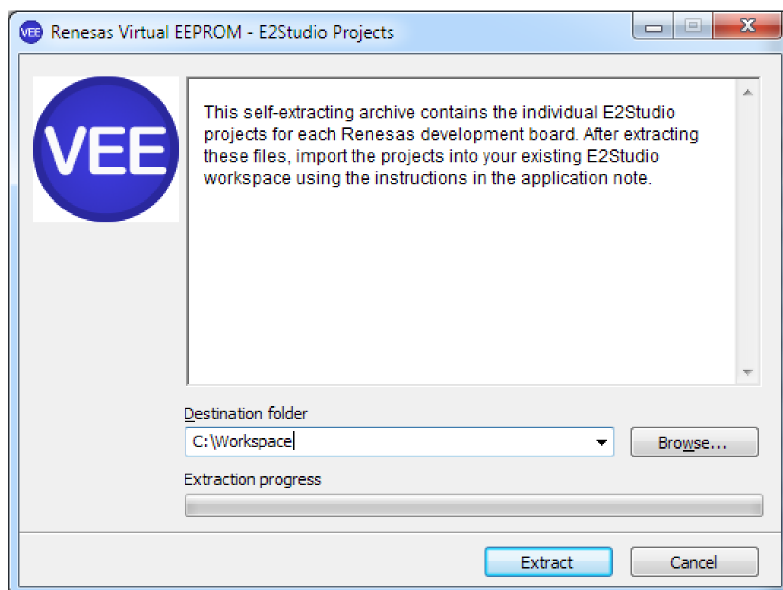
```
/***********************************************************************
DEFINE YOUR SYSTEM - UNCOMMENT THE INCLUDE PATH FOR THE PLATFORM YOU ARE USING.
***********************************************************************
/* RSKRX610 */
//#include "./board/rskrx610/r_bsp.h"

/* 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"
```

4. これで、デモンストレーションをビルドし、実行することができます。

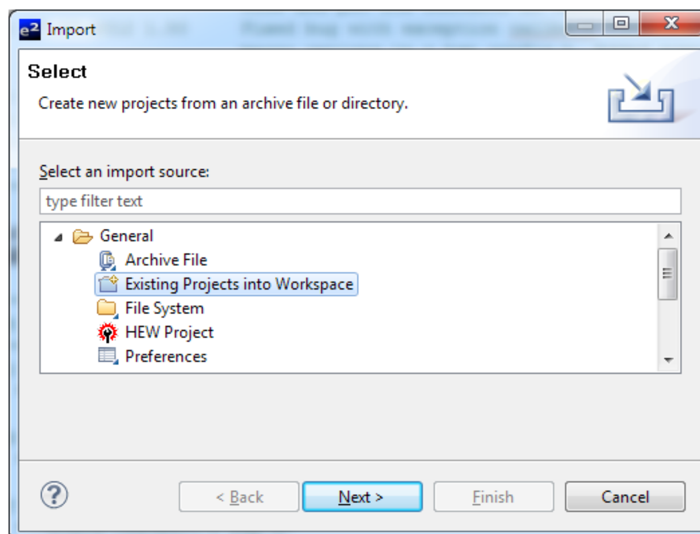
4.2　E2Studio プロジェクト

E2Studio では HEW と異なった方法でワークスペースを扱っており、プロジェクトは既存のアプリケーションの E2Studio ワークスペースに導入されなければなりません。特定の開発ボードでデモンストレーションを使用するには次の手順に従ってください。

1. E2Studio プロジェクトは自己展開ファイルの形でこのアプリケーションノートに同梱されています。最初にすべきことは、このアーカイブファイルの展開です。自己展開ファイル（Workspace¥e2studio ディレクトリにある*.exe ファイルです）をダブルクリックします。
2. プロジェクトを展開する場所を選択し、Extract をクリックします。

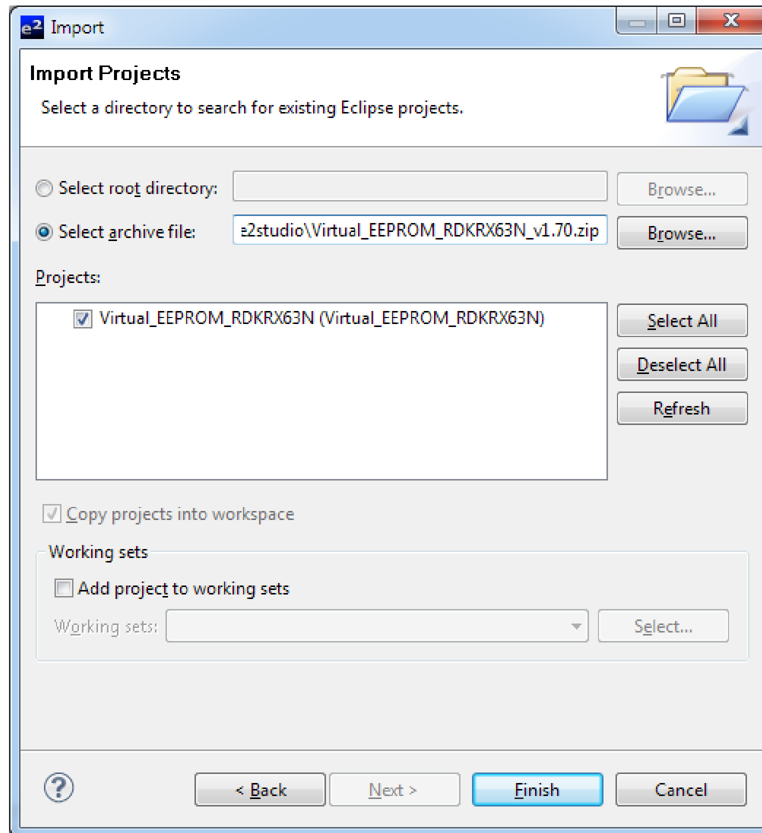


3. 導入先の E2Studio ワークスペースを開きます。
4. File >> Import（File メニューの中の Import）をクリックします。
5. General >> Existing Projects into Workspace を選択し、Next をクリックします。



6. Select archive file をクリックし、次に browse をクリックします。
7. E2Studio プロジェクトを展開するディレクトリを探し、使用される開発ボードに対応する zip ファイルを選択します。

8. 導入したいプロジェクトの横のボックスにチェックを入れ、Finish をクリックします。このスクリーンショットでは RDKRX63N プロジェクトが導入されています。



9. VEE API コードとデモンストレーションのワークスペースでは、スタートアップコードとボードサポートコード、および MCU 情報を r_bsp パッケージから得ています。r_bsp パッケージは r_bsp フォルダにある plotform.h ヘッダファイルを通して容易にコンフィグレーションを行うことができます。r_bsp パッケージのコンフィグレーションでは、platform.h ファイルを開き、使用するボードに関する定義のコメントを外します。一例として、RSK+RX63N ボードでデモンストレーションを実行するには、./board/rskrx63n/r_bsp.h ファイルの#include のコメントを外し、同時に他のボードの#include がすべてコメントとしてあることを確認してください。

10. これで、デモンストレーションをビルドし、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/contact/

改訂記録	RX600 & RX200 シリーズ　アプリケーションノート RX 用仮想 EEPROM

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2011.07.15	—	初版発行
1.50	2012.01.03	—	RX63x グループのサポートを追加。VEE セクタコンフィグレーション定義のプロパティの変更に対応するようドキュメントを改訂。最新のコーディング基準に合うようコードが更新されたことによる細部の変更。
1.60	2012.09.14	—	R_VEE_GenerateCheck()を API 関数に追加。FIT 仕様 v0.7 に準じてコードを更新。
1.70	2013.09.18	—	• FIT 仕様 v1.0 に準じてコードを更新。 • 「API 関数セクション」に「概要」のサブセクションを追加。 • 「ユーザプロジェクトにミドルウェアを追加するには」サブセクションを改訂。 • 「制約」サブセクションを追加。 • 「サポートされるツールチェイン」サブセクションを追加。 • R_VEE_Open()関数と R_VEE_Control()関数を追加し、「API 関数」セクションに記載。 • R_VEE_GetVersion()関数の記載。 • HEW と E2Studio のプロジェクトを使用するよう「デモワークスペース」セクションの記述を更新。

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下さい。

   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

http://www.renesas.com

■営業お問合せ窓口

※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/

© 2013 Renesas Electronics Corporation. All rights reserved.

Colophon 2.0