アプリケーション・ノート

NEC

USBファンクション ファームウエア・アップデート

78K0Rマイクロコントローラ編

対象デバイス

78K0Rマイクロコントローラ USBファンクション内蔵品

資料番号 U20291JJ1V0AN00
発行年月 March 2010 NS

〔メ　モ〕

目次要約

CMOSデバイスの一般的注意事項

① 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デバイス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。入力レベルが固定な場合はもちろん，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等が入らないようご使用ください。

② 未使用入力の処理

CMOSデバイスの未使用端子の入力レベルは固定してください。

未使用端子入力については，CMOSデバイスの入力に何も接続しない状態で動作させるのではなく，プルアップかプルダウンによって入力レベルを固定してください。また，未使用の入出力端子が出力となる可能性（タイミングは規定しません）を考慮すると，個別に抵抗を介してV_{DD}またはGNDに接続することが有効です。

資料中に「未使用端子の処理」について記載のある製品については，その内容を守ってください。

③ 静電気対策

MOSデバイス取り扱いの際は静電気防止を心がけてください。

MOSデバイス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，または導電性の緩衝材，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，MOSデバイス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。

④ 初期化以前の状態

電源投入時，MOSデバイスの初期状態は不定です。

電源投入時の端子の出力状態や入出力設定，レジスタ内容などは保証しておりません。ただし，リセット動作やモード設定で定義している項目については，これらの動作ののちに保証の対象となります。

リセット機能を持つデバイスの電源投入後は，まずリセット動作を実行してください。

⑤ 電源投入切断順序

内部動作および外部インタフェースで異なる電源を使用するデバイスの場合，原則として内部電源を投入した後に外部電源を投入してください。切断の際には，原則として外部電源を切断した後に内部電源を切断してください。逆の電源投入切断順により，内部素子に過電圧が印加され，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「電源投入切断シーケンス」についての記載のある製品については，その内容を守ってください。

⑥ 電源OFF時における入力信号

当該デバイスの電源がOFF状態の時に，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OFF時における入力信号」についての記載のある製品については，その内容を守ってください。

注意：本製品は，Silicon Storage Technology, Inc.からライセンスを受けたSuperFlash®を使用しています。

MINICUBE**は，**NEC**エレクトロニクス株式会社の登録商標です。**

Windows XP**および**Windows Vista**は，米国**Microsoft Corporation**の米国およびその他の国における登録商標または商標です。**

SuperFlash**は，米国**Silicon Storage Technology, Inc.**の米国，日本などの国における登録商標です。**

- 本資料に記載されている内容は2010年03月現在のもので，今後，予告なく変更することがあります。量産設計の際には最新の個別データ・シート等をご参照ください。
- 文書による当社の事前の承諾なしに本資料の転載複製を禁じます。当社は，本資料の誤りに関し，一切その責を負いません。
- 当社は，本資料に記載された当社製品の使用に関連し発生した第三者の特許権，著作権その他の知的財産権の侵害等に関し，一切その責を負いません。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。
- 本資料に記載された回路，ソフトウエ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。お客様の機器の設計において，回路，ソフトウエ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，一切その責を負いません。
- 当社は，当社製品の品質，信頼性の向上に努めておりますが，当社製品の不具合が完全に発生しないことを保証するものではありません。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。当社製品をお客様の機器にご使用の際には，当社製品の不具合の結果として，生命，身体および財産に対する損害や社会的損害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を行ってください。
- 当社は，当社製品の品質水準を「標準水準」，「特別水準」およびお客様に品質保証プログラムを指定していただく「特定水準」に分類しております。また，各品質水準は，以下に示す用途に製品が使われることを意図しておりますので，当社製品の品質水準をご確認ください。

  「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

  「特別水準」: 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，生命維持を目的として設計されていない医療機器

  「特定水準」: 航空機器，航空宇宙機器，海底中継機器，原子力制御システム，生命維持のための医療機器，生命維持のための装置またはシステム等

  当社製品のデータ・シート，データ・ブック等の資料で特に品質水準の表示がない場合は，標準水準製品であることを表します。意図されていない用途で当社製品の使用をお客様が希望する場合には，事前に当社販売窓口までお問い合わせください。

  注1. 本事項において使用されている「当社」とは，NECエレクトロニクス株式会社およびNEC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う。

  注2. 本事項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，製造製品をいう。

（M8E0909J）

はじめに

注　　意　**このアプリケーション・ノートで使用するサンプル・プログラムはあくまで参考用のものであり，当社がこの動作を保証するものではありません。**
サンプル・プログラムを使用する場合，ユーザのセット上で十分な評価をしたうえで使用してください。

対 象 者　**このアプリケーション・ノートは，**78K0R**マイクロコントローラの機能を理解し，それを用いたアプリケーション・システムを開発しようとするユーザを対象とします。**

目　　的　**このアプリケーション・ノートは，**78K0R**マイクロコントローラに内蔵の**USB**ファンクション・コントローラを使用するためのサンプル・ドライバの仕様をユーザに理解していただくことを目的とします。**

構　　成　**このアプリケーション・ノートは，大きく分けて次の内容で構成しています。**

- USB**ファンクション・ファームウエア・アップデートの概要**
- **プログラム構成の説明**
- **アプリケーションの使用方法**
- **サンプル・プログラムの応用**

読 み 方　**このマニュアルの読者には，電気，論理回路，およびマイクロコントローラに関する一般知識を必要とします。**

ハードウエア機能の詳細（特にレジスタ機能とその設定方法など），および電気的特性を知りたいとき
→別冊の対象78K0R**マイクロコントローラの　ユーザーズ・マニュアル　ハードウエア編を参照してください。**

命令機能の詳細を理解しようとするとき
→別冊の78K0R　**ユーザーズ・マニュアル　命令編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：~~xxx~~**（端子，信号名称に上線）**

メモリ・マップのアドレス：上部-上位，下部-下位

注　：本文中に付けた注の説明

注意：気を付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**10**進数**… XXXX

16**進数**　　… XXXXH**または**0xXXXX

2**のべき数を示す接頭語（アドレス空間，メモリ容量）：**

K（**キロ**）　… $2^{10} = 1024$

M（**メガ**）　… $2^{20} = 1024^2$

G（**ギガ**）　… $2^{30} = 1024^3$

データ・タイプ：ワード　…　32**ビット**

ハーフワード　…　16**ビット**

バイト　…　8**ビット**

関連資料　関連資料は暫定版の場合がありますが，この資料では「暫定」の表示をしておりません。あらかじめご了承ください。

78K0R/KC3-L, 78K0R/KE3-L**に関する資料**

資　料　名	資料番号
78K0R**マイクロコントローラ　ユーザーズ・マニュアル　命令編**	U17792J
78K0R/KC3-L, 78K0R/KE3-L　**ユーザーズ・マニュアル**	U19878J
78K0R**マイクロコントローラ**　16**ビット・シングルチップ・マイクロコントローラ　フラッシュ・セルフ・プログラミング・ライブラリ** Type02	U19193J

開発ツールに関する資料（ユーザーズ・マニュアル）

資　料　名		資料番号
QB-MINI2　**プログラミング機能付きオンチップ・デバック・エミュレータ**		U18371J
CC78K0R　Ver2.00　C**コンパイラ**	**操作編**	U18549J
	言語編	U18548J
RA78K0R　Ver1.20　**アセンブラ・パッケージ**	**操作編**	U18547J
	言語編	U18546J
SM+　**システム・シミュレータ**	**操作編**	U18010J
PM+　Ver.6.31　**プロジェクト・マネージャ**		U18416J
ID78K0R-QB　Ver.3.20　**統合デバッガ**		U17839J
PG-FP5　**フラッシュ・メモリ・プログラマ**		U18865J

備考1.　**スタータ・キット**（TK-78K0R/KE3L+USB)**は，テセラ・テクノロジー株式会社製です。詳細については，テセラ・テクノロジー株式会社にお問い合わせください。**

2. USB**規格の策定と管理は，**USB Implementers Forum（USB-IF）**という団体が行っています。**
Universal Serial Bus Class Definitions for Communications Devices**の詳細については，**USB-IF**の公式ウェブサイト**（www.usb.org）**を参照してください。**

目　　次

第1章　概　　要

1.1　本書の目的

このアプリケーション・ノートでは，フラッシュ・セルフ・プログラミング・ライブラリ（以降，セルフ・ライブラリ）を用いて，内蔵フラッシュ・メモリ上のデータを任意に書き換える処理（セルフ書き換え）の応用，およびUSBファンクション・コントローラのコミュニケーション・デバイス・クラス（以降，CDCと記します）を用いた処理の応用を，理解することを目的としています。

処理の詳細は，USBファンクション・ファームウエア・アップデート・サンプル・プログラムを例に述べていきます。

評価環境は、製品ごとに下記を使用します。

表1－1　マイコンと評価環境ターゲット・ボード一覧

対象CPU	78K0R/KC3-L 78K0R/KE3-L
評価環境 ターゲットボード	TK-78K0R/KE3L+USB

1. 2　USBファンクション・ファームウエア・アップデート概要

USBファンクション・ファームウエア・アップデート・サンプル・プログラムは，ホスト・マシン（以降，PCと記します）上のファイル転送アプリケーションで指定されたファイルを，USBでのシリアル通信で評価ボードに転送し，セルフ・ライブラリを使用してユーザ・プログラムのブート領域への書き込みや，メモリ上の任意の場所にデータを書き込めます。

USBファンクション・ファームウエア・アップデート・サンプル・プログラムは，次のプログラムで構成されます。

・ファームウエア・アップデート・プログラム

評価ボードに実装します。USBでのシリアル通信，およびセルフ書き換えを行います。

・ファイル転送アプリケーション

ホスト・マシン（PC）で動作し，指定ファイルをシリアル通信で評価ボードへ送ります。

・サンプル・ユーザ・プログラム

動作確認のためのHEXファイルです。

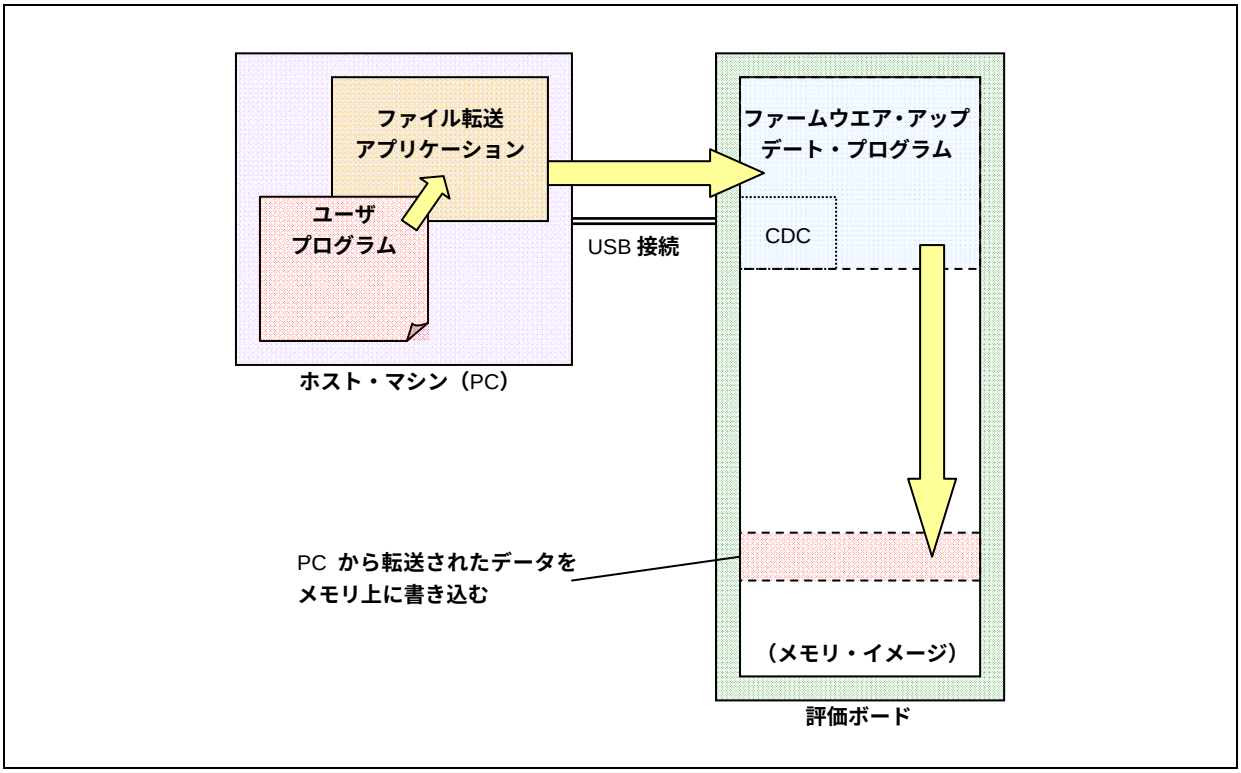
- TK-78K0R/KE3L+USB用

1秒タイマ・サンプル・プログラム　：　1秒ごとにLEDカウントアップ表示を行います。

1/10秒タイマ・サンプル・プログラム　：　1/10秒ごとにLEDカウントアップ表示を行います。

次に，プログラムのデータの流れを表します。

図1－1　USBファンクション・ファームウエア・アップデートのデータの流れ



評価ボードは，通常，起動すると書き込まれたユーザ・プログラムが動作しますが，特定条件下で起動（またはリセット）することで，ファームウエア・アップデート・プログラムが動作します。

1. 2. 1 特　　徴

USBファンクション・ファームウエア・アップデート・サンプル・プログラムには，次の特徴があります。

- ファームウエア・アップデート・プログラムは，内蔵フラッシュ・メモリを14ブロック（14 Kバイト）使用します。
- ユーザ・プログラムの書き換え可能なフォーマット（HEXファイル）は，モトローラSフォーマットとインテル拡張フォーマットです。
- メモリ上のアドレスを指定して，任意の領域にデータを書き込めます。
- ユーザ・プログラムは，すべての割り込みを使用できます。
- 使用する内部資源を表1－2に示します。

表1－2　ファームウエア・アップデート・プログラムで使用する内部資源

資源名	用途	サイズ（バイト）(※1)
		TK-78K0R/KE3L＋USB
ROM	ファームウェア・アップデート・プログラム （ブートストラップ，ライブラリ含む）	9K（※2： 12K）
	セルフ・プログラミング・ライブラリ	806（※2： 1K）
	ユーザ・リセット・ベクタ退避領域	1K
RAM	ファームウェア・アップデート・プログラム用	2.4K
	セルフ・プログラミング・ライブラリ用	1K
	スタック・エリア	230(※3)

※1： ログ出力機能OFF時

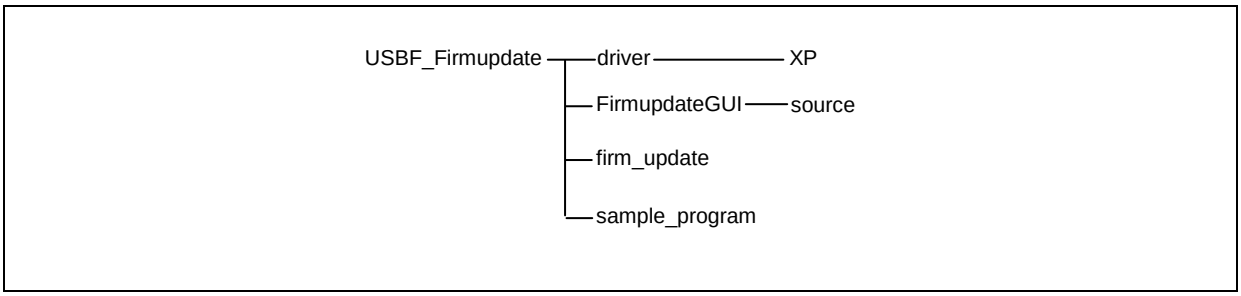
※2： ブロック確保サイズ

※3： 最大値

1. 2. 2 フォルダ構成

USBファンクション・ファームウエア・アップデート・サンプル・プログラムのフォルダ構成を示します。

図1－2 USBファンクション・ファームウエア・アップデート・サンプル・プログラムのフォルダ構成



次に，各フォルダの説明を示します。

（1）driver¥XP

Windows XP®用のCDCドライバが格納されているフォルダです。

xxxx_CDC_XP.inf：Windows XP用のCDCドライバ(xxxxには型番が入ります)

（2）FimupdateGUI

ファイル転送アプリケーションが格納されているフォルダです。

UsbfUpdate.exe ：ファイル転送アプリケーションの実行ファイル

UsbfUpdate.ini ：ファイル転送アプリケーションの設定ファイル

（3）FirmupdateGUI¥source

ファイル転送アプリケーションのソース・プログラムが格納されているフォルダです。**第4章 ファイル転送アプリケーションの解説**を参照してください。

（4）firm_update

ファームウエア・アップデート・プログラムが格納されているフォルダです。**第3章 ファームウエア・アップデート・プログラムの解説**を参照してください。

（5）sample_program

サンプル・ユーザ・プログラムが格納されているフォルダです。

- TK-78K0R/KE3L＋USB用

1sec_sample.hex ：1秒タイマ・カウント・プログラム

100msec_sample.hex ：1/10秒タイマ・カウント・プログラム

第2章　USBファンクション・ファームウエア・アップデートの実行

USBファンクション・ファームウエア・アップデート・サンプル・プログラムの実行方法について説明します。

ここではTK-78K0R/KE3L＋USBボードを用い、ユーザ・プログラムが，1秒タイマ・カウント・プログラム，次に1/10秒タイマ・カウント・プログラムの順番で書き換わることを確認します。

2. 1　動作環境

ハードウエア環境を次に示します。

○評価ボード	TK-78K0R/KE3L＋USB（テセラ・テクノロジー株式会社製）
○評価ボード搭載CPU	μPD78F1026（78K0R/KE3-L)
○インサーキット・エミュレータ	QB-MINI2（MINICUBE2®）
○USBケーブル	評価ボードとPC間でシリアル通信を行う
○PC	Windows XP搭載

ソフトウエア環境を次に示します。

○統合開発環境	PM+ V6.31
○コンパイラ	CC78K0R Ver1.20
	RA78K0R Ver1.20
○デバッガ	ID78K0R-QB V3.20

○USBファンクション・ファームウエア・アップデート・サンプル・プログラム一式

ファームウエア・アップデート・プログラム

ファイル転送アプリケーション

サンプル・ユーザ・プログラム：1秒タイマ・カウント・プログラム

1/10秒タイマ・カウント・プログラム

2. 2　サンプル・プログラムの実行

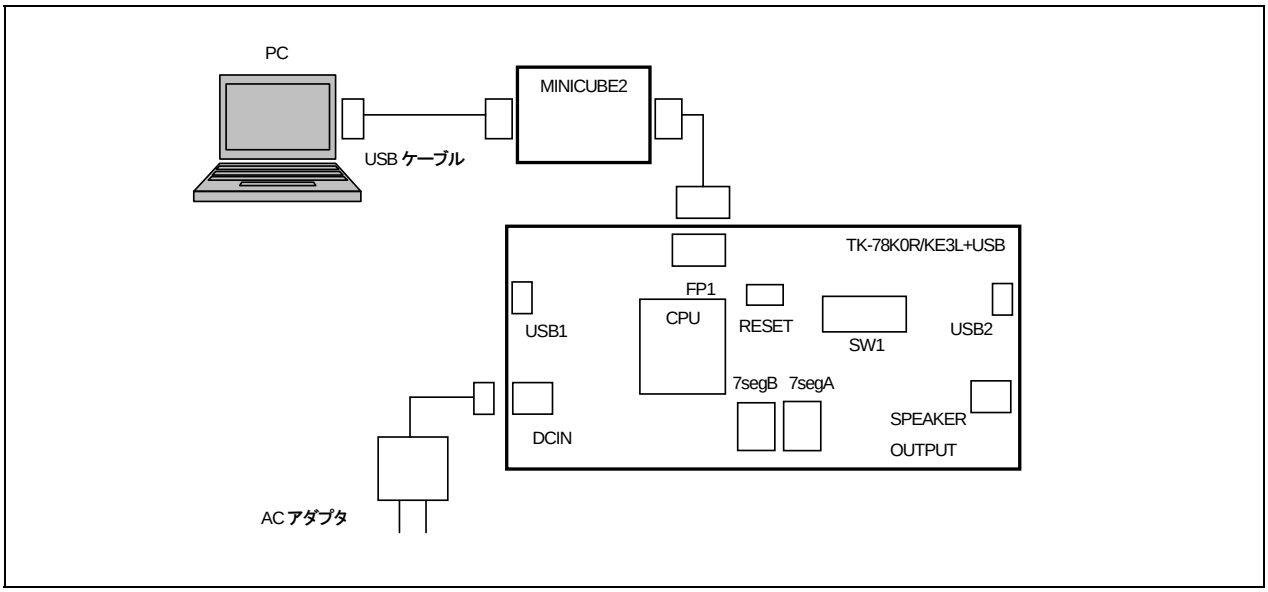
USBファンクション・ファームウエア・アップデート・サンプル・プログラムの動作環境を実行し，書き換え処理を行うときの手順を示します。

2. 2. 1　ファームウエア・アップデート・プログラムの実装

（1）ファームウエア・アップデート・プログラムを実装するために，MINICUBE２と評価ボードを接続します。

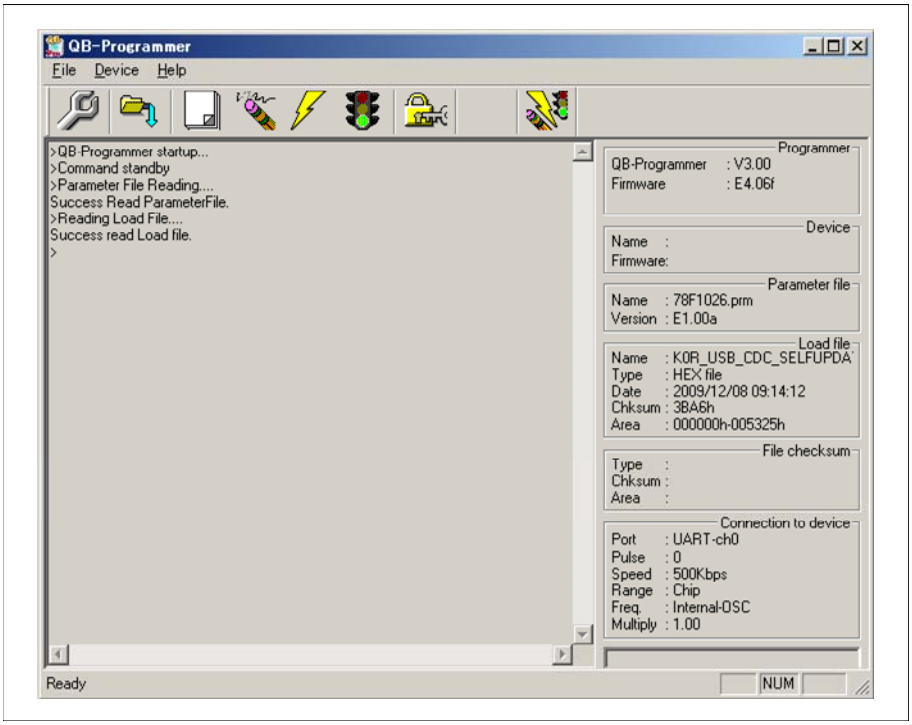
次にTK-78K0R/KE3L＋USBの接続図を示します。

図2－1　MINICUBE２と評価ボードの接続図



（2）QB-Programmerを起動し，「K0R_USB_CDC_selfupdate_sample.hex」を選択してファームウエア・アップデート・プログラムをターゲット・ボードにダウンロードします。

図2－2　ワークスペース・ファイルの指定

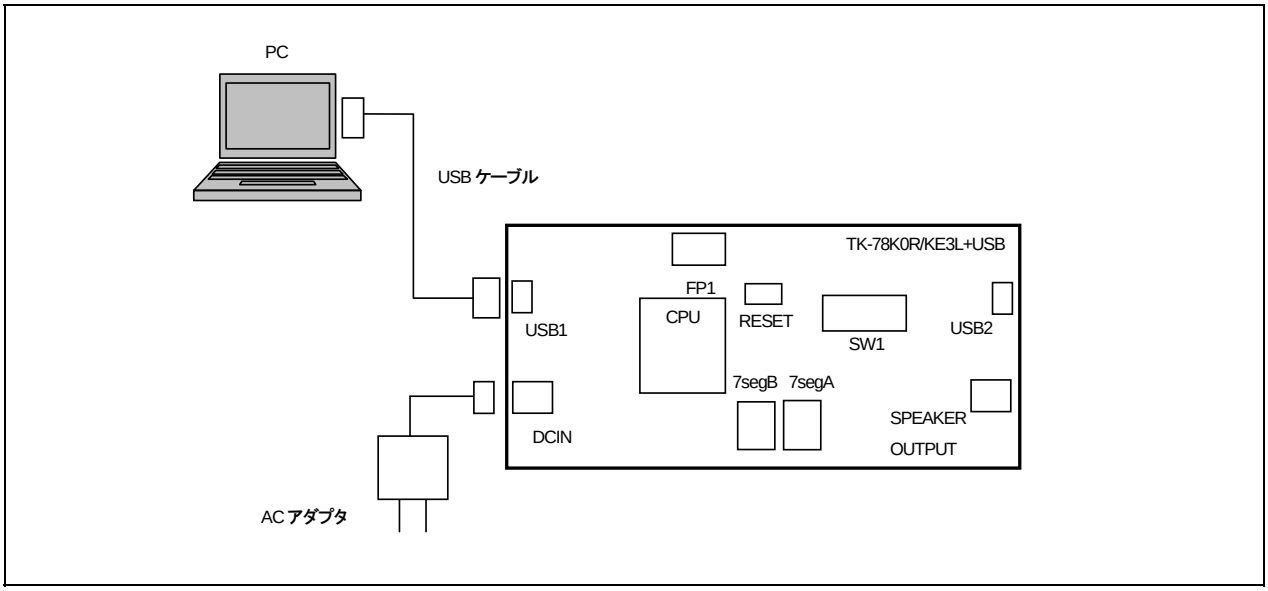


備考：　QB-Programmerについての詳細は、以下のURLを参照してください。
URL：http://www.necel.com/micro/ja/development/asia/minicube2/minicube2.html

2. 2. 2　書き換え処理の実行手順

（1）書き換え処理を実行するために，MINICUBE2を外し，PCと評価ボードをUSBケーブルで接続します。次にTK-78K0R/KE3L+USBの接続図を示します。

図2－3　PCと評価ボードの接続図



（2）評価ボード上のSW1のBit8をOnにして，リセット・ボタンを押してください。書き換えモードとなり，7SegA及び7SegBが「FF」と表示されて，PCからの転送データを待つ状態になります。

注意　初めてTK-78K0R/KE3L+USBとPCを接続し書き換えモードにした場合は，CDCドライバのインストールが必要になります。詳細については，2. 2. 3　CDC**ドライバのインストールを参照してください。**

(3) 評価ボードへ転送するサンプル・ユーザ・プログラムのHEXファイル（ここでは1秒タイマ・カウント・プログラムの“1sec_sample.hex”ファイルを指します）をPC側に用意します。PC上で，ファイル転送アプリケーションを起動します（図2－4参照）。

“Load File”ボタンをクリックして，対象とするHEXファイルを選択します（ファイルのパスは，《Path》のテキスト・ボックスへの直接入力，またはファイルのウインドウ上へのドラッグ＆ドロップでも，《Path》のテキスト・ボックスに表示できます）。

《Mode》は《Chip》を選択します。《COM》は，デバイス・マネージャを確認し，接続したUSBポートを選択してください。

注意　COMの番号は環境によって変わります。

図2－4　ファイル転送アプリケーションでファイルを選択

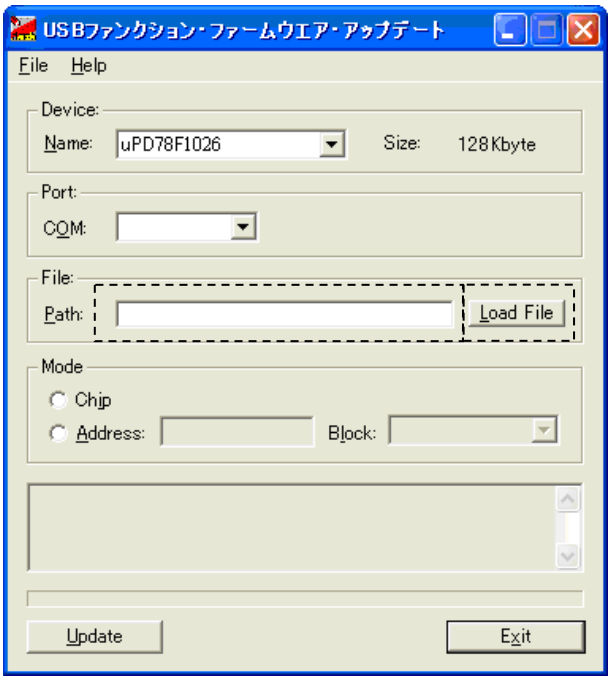
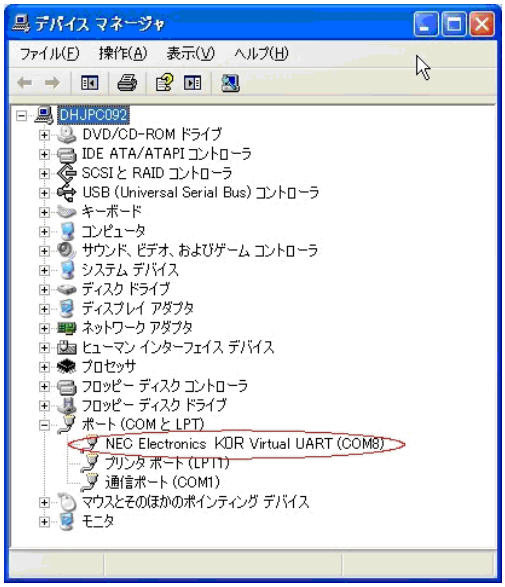


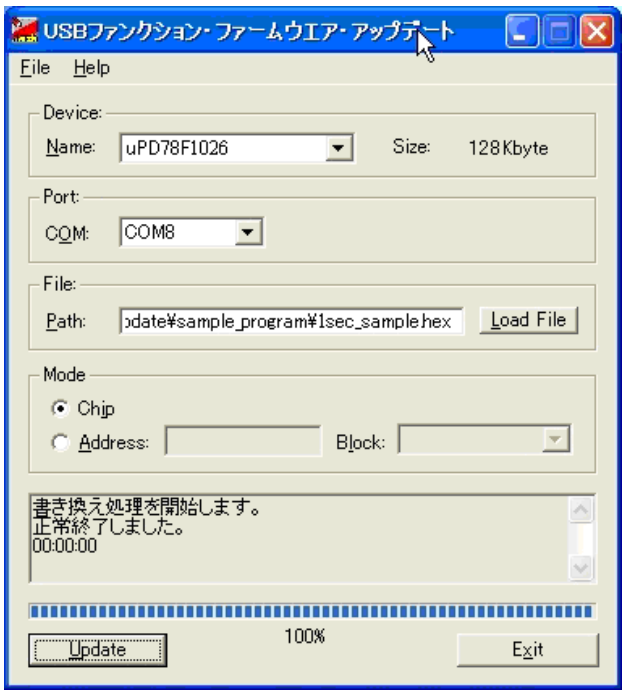
図2－5　デバイス・マネージャ



(4) ファイル転送アプリケーションの“Update”ボタンをクリックします。開始のメッセージが表示され，ファイルの転送処理，および書き換え処理が開始されます。

(5) 転送処理，および書き換え処理が終了すると，ファイル転送アプリケーションにより，終了メッセージが表示されます。これで一連の書き換え処理は終了です。

図2－6　書き換え処理終了1

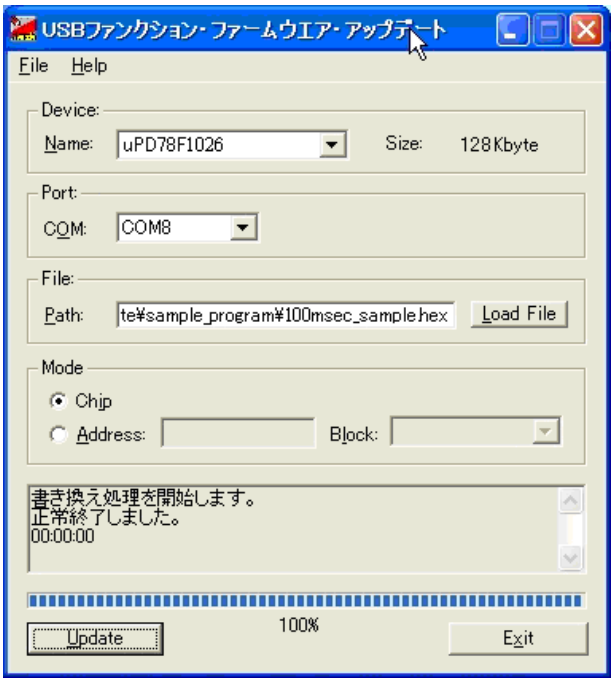


（6）評価ボード上のSW1のBit8をOffにして評価ボードをリセットします。書き込んだユーザ・プログラムが起動します。

1秒ごとにカウントアップを行い、７SegA及び7SegBにて表示を行います。

（7）ユーザ・プログラムを書き換えます。1/10秒タイマ・カウント・プログラム“100msec_sample.hex”を用意し，（4）から同様の手順で書き換えを行います。

図2－7　書き換え処理終了2



（8）評価ボード上のSW1のBit8をOffにして評価ボードをリセットします。書き込んだユーザ・プログラムが起動します。

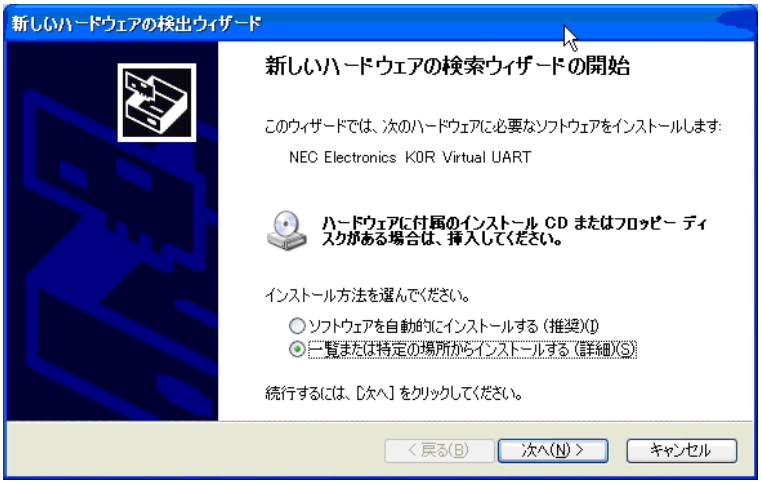
1/10秒ごとにカウントアップを行い、７SegA及び7SegBにて表示を行います。

2. 2. 3　CDCドライバのインストール

TK-78K0R/KE3L+USBを初めて書き換えモードにした場合は，PCにCDCドライバをインストールする必要があります。次にWindows XPを例としたCDCドライバのインストール手順を示します。

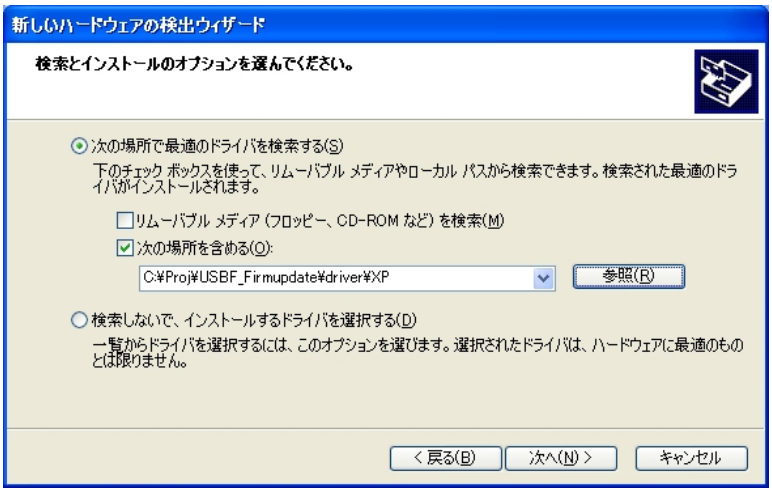
(1) 新しいハードウエアが検出され，＜新しいハードウエアの検出ウィザード＞ウインドウが表示されます。《一覧または特定の場所からインストールする（詳細）（S）》を選択し，“次へ（N）”をクリックください。

図2－8　新しいハードウエアの検索ウィザード



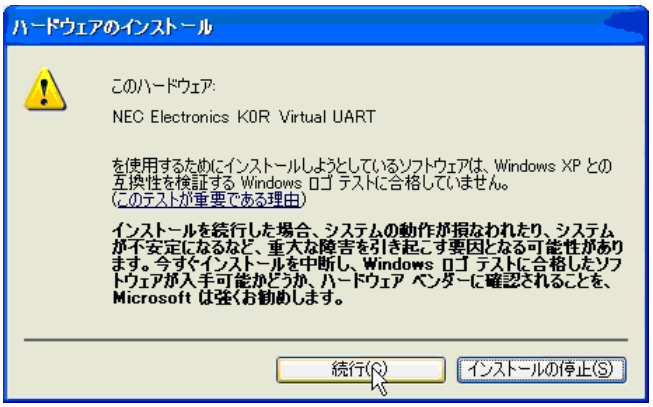
(2) 《次の場所で最適のドライバを検索する（S）》，《次の場所を含める（O）》を選択します。“参照（R）”をクリックして“xxxx_CDC_XP.inf”(xxxxは型番)の存在するフォルダを指定し，“次へ（N）”をクリックしてください。

図2－9　ドライバの場所の選択



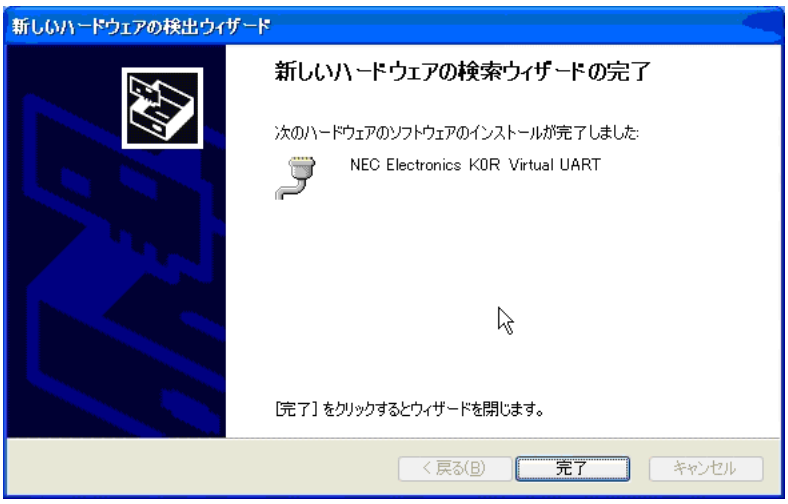
(3) 次のインストール確認画面が表示される場合は，“ 続行（C） ”をクリックしてください。

図2－10　インストール確認



(4) 次のウインドウが表示されたら，CDCドライバのインストールは完了です。 ”完了“ をクリックしてください。

図2－11　インストール完了



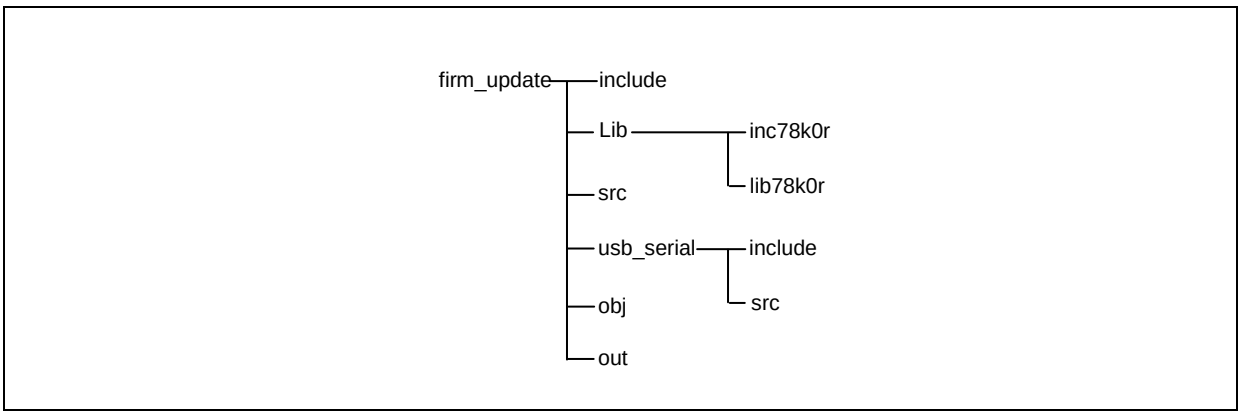
第3章　ファームウエア・アップデート・プログラムの解説

この章では，ファームウエア・アップデート・プログラムで使用している各ファイルについて説明します。

3. 1　ファイル・フォルダ構成

ファームウエア・アップデート・プログラムのソース・ファイルとフォルダ構成を示します。

図3－1　ファームウエア・アップデート・プログラムのフォルダ構成



3. 1. 1　firm_updateフォルダ

ファームウエア・アップデート・プログラムのフォルダ直下にはプロジェクト関連ファイルが格納されています。次に主なファイルを示します。

表3－1　ファームウエア・アップデート・プログラムのプロジェクト関連ファイル

ファイル名	説　　明
firm_update.prw	PM+ワークスペース・ファイル
firm_update.prj	PM+プロジェクト・ファイル
firm_update.pri	PM+プロジェクトのPRIファイル
firm_update.mak	メイク・ファイル
firm_update.dr	リンク・ディレクティブ・ファイル

3. 1. 2　firm_update¥include**フォルダ**

ファームウエア・アップデート・プログラムのヘッダ・ファイルを格納するフォルダです。

表3－2　ファームウエア・アップデート・プログラムのヘッダ・ファイル

ファイル名	説　　明
debug.h	ログ出力機能用ヘッダ・ファイル
fsl_user.h	セルフ・プログラミング・ライブラリ・ユーザ処理ヘッダ・ファイル
Types.h	データ・タイプ定義ヘッダ・ファイル

3. 1. 3　firm_update¥lib**フォルダ**

セルフ・ライブラリを格納するフォルダです。

表3－3　セルフ・ライブラリとヘッダ・ファイル

ファイル名	説　　明
inc78k0r¥fsl.h	セルフ・プログラミング・ライブラリ・ヘッダ・ファイル
lib78k0r¥fsl.lib	セルフ・プログラミング・ライブラリ

3. 1. 4　firm_update¥src**フォルダ**

ファームウエア・アップデート・プログラムのソース・ファイルを格納するフォルダです。

表3－4　ファームウエア・アップデート・プログラムのソース・ファイル

ファイル名	説　　明
main.c	C言語メイン関数記述ファイル
flash_update.c	セルフ・アップデート処理ファイル
fsl_user.c	セルフ・プログラミング・ライブラリ・ユーザ処理ファイル
debug.c	ログ出力機能用ファイル

3. 1. 5　firm_update¥usb_serialフォルダ

CDCのソース・ファイル，およびヘッダ・ファイルを格納します。

表3ー5　CDCプログラムのソース・ファイル

ファイル名	説　　明
usbf78k0r.c	78K0R USB ファンクション・ドライバ・ファイル
usbf78k0r_communication.c	CDCドライバファイル
usbf78k0r.h	78K0R USBドライバ・ヘッダ・ファイル
usbf78k0r_communication.c	CDCドライバ・ヘッダ・ファイル
usbcomm_desc.h	CDCディスクリプタ情報ファイル
errno.h	エラー番号定義ヘッダ・ファイル

3. 1. 6　firm_update¥objフォルダ

ファームウエア・アップデート・プログラムのオブジェクト・ファイルを格納するフォルダです。

3. 1. 7　firm_update¥outフォルダ

ファームウエア・アップデート・プログラムの実行可能なオブジェクト・ファイルとHEXファイルを格納するフォルダです。

表3ー6　CDCプログラムのオブジェクト・ファイル

ファイル名	説　　明
K0R_USB_CDC_selfupdate_sample.lmf	実行可能オブジェクト・ファイル
K0R_USB_CDC_selfupdate_sample.hex	ヘキサ・フォーマットの実行可能オブジェクト・ファイル

3. 2　メモリ・マップ

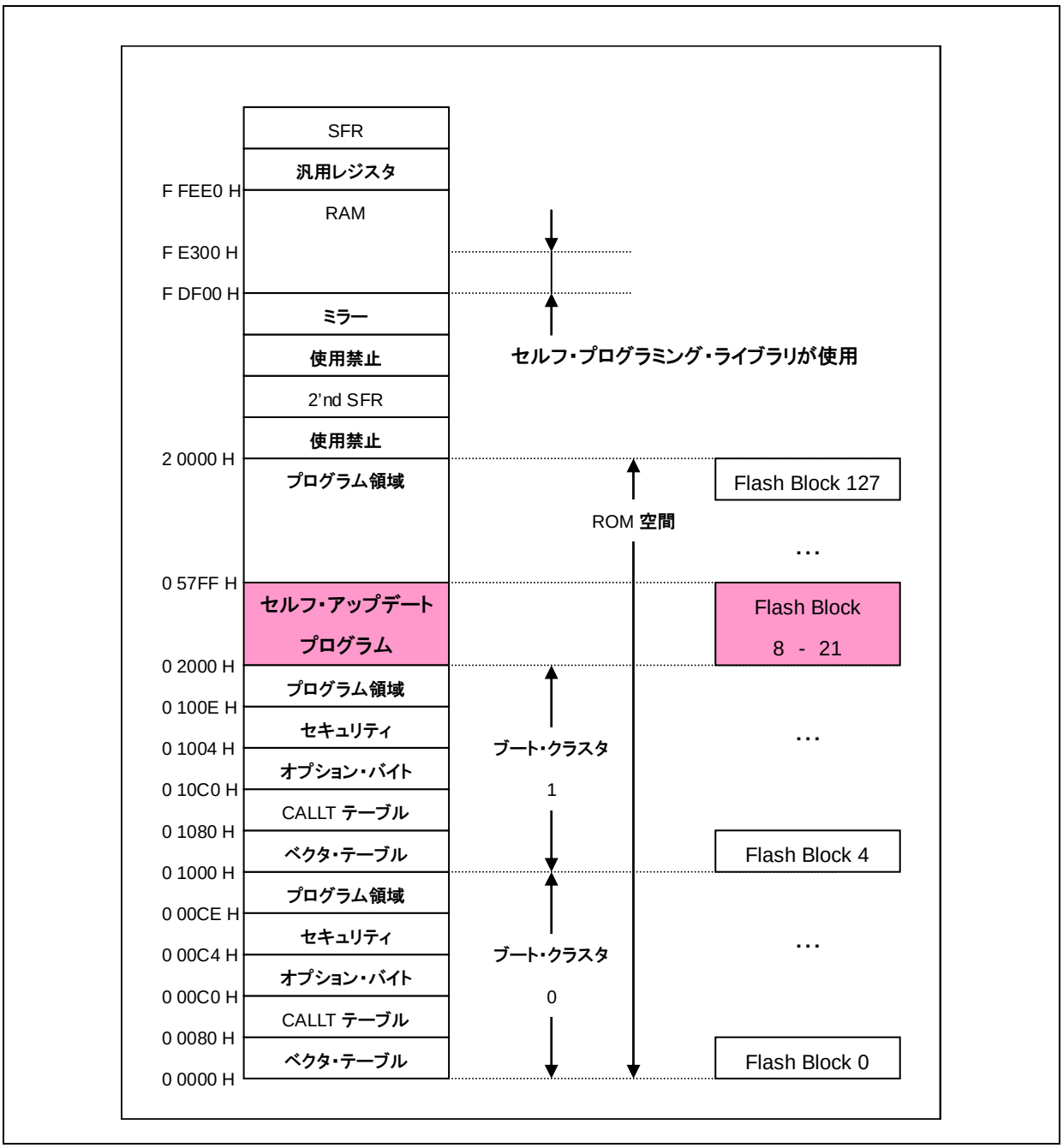
メモリ配置とリンク・ディレクティブ・ファイルの説明をします。

3. 2. 1　メモリ・マップ

下記は，セルフ・アップデート・プログラムのメモリ・マップです。ここでは78K0R/KE3-L(μPD78F1026)を例に説明します。

ブロックとは，セルフ・ライブラリが内蔵フラッシュ・メモリを書き換える単位です。

図3ー2　メモリ・マップ



3. 2. 2　リンク・ディレクティブ・ファイル（flash_update.dr）

リンク・ディレクティブ・ファイル（flash_update.dr）で領域の割り当てを行います（セグメントを定義し，マッピングを行います）。ここでは78K0R/KE3-L(μPD78F1026)を例に説明します。

ROM領域，RAM領域配置情報等をμPD78F1026（78K0R/KE3-L）の内蔵メモリ空間へ配置します。

（1）ROM領域の割り当て

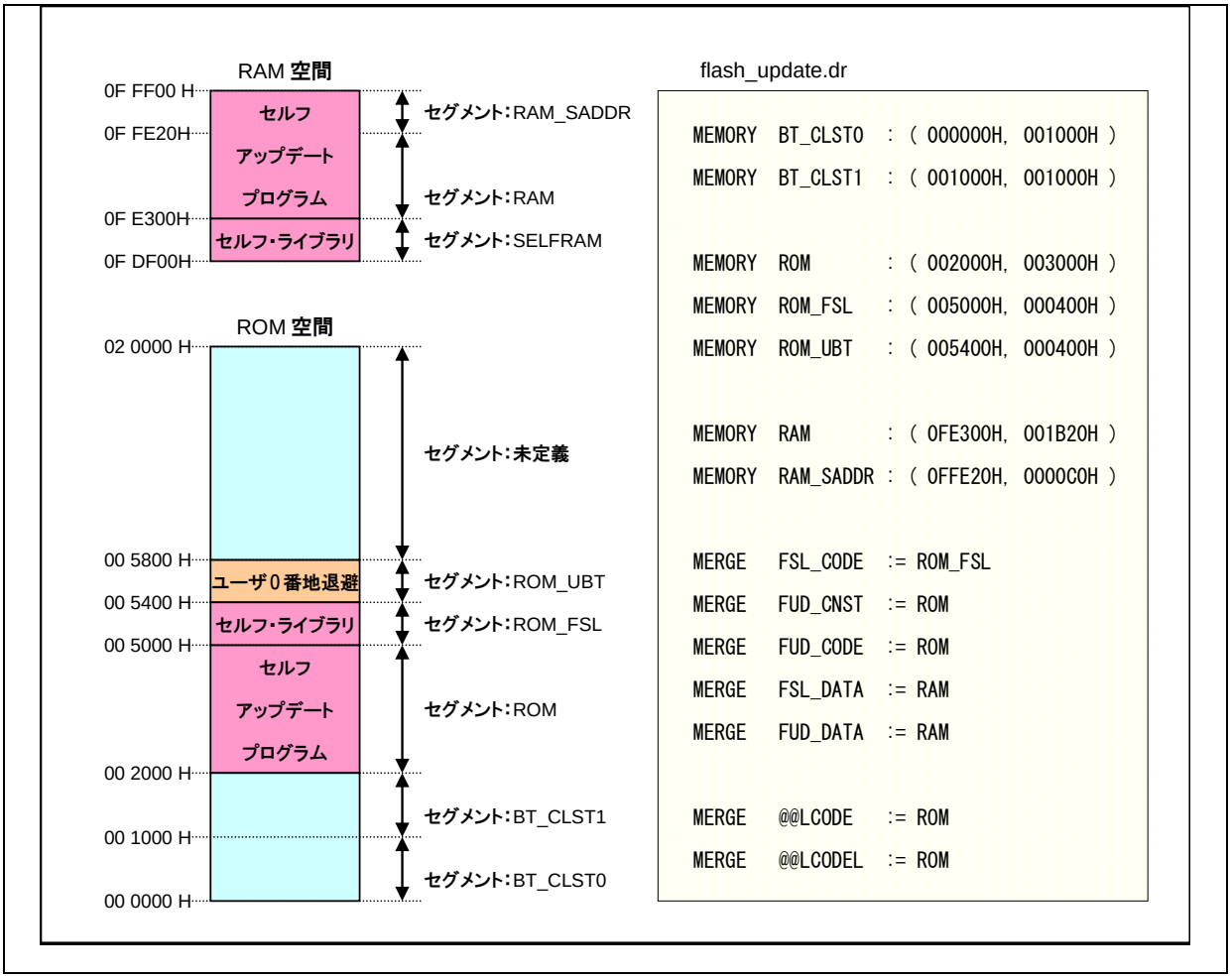
ファームウエア・アップデート・プログラムが使用するROM空間配置情報は，00 2000H-00 57FFHの14 Kバイトです。したがって，ユーザ・プログラムが使用するROM領域は，00 2000H-00 57FFHの14 Kバイト以外に配置する必要があります。

（2）RAM領域の割り当て

0F DF00H-0F FF00HがRAM空間となります。

セルフ・プログラミング・ライブラリは0F DF00H-0F E300Hおよび0F FE20H-0F FF00Hを使用します。しかしながら、ファームウェア・アップデート・プログラム実行時にのみ使用するので、ユーザ・プログラムはRAM領域配置に関して影響を受けません。

図3－3　リンク・ディレクティブ（ROM/RAM）



備考：　RAM領域名”SELFRAM”(0F DF00H-0F E2FFHは，セルフ・プログラミング・ライブラリ内で定義されています。

3. 3　ブート処理

ブート処理とは，78K0Rマイコンをリセット後，メイン関数（C言語記述：main（））を実行する前に実行するプログラムを指します。

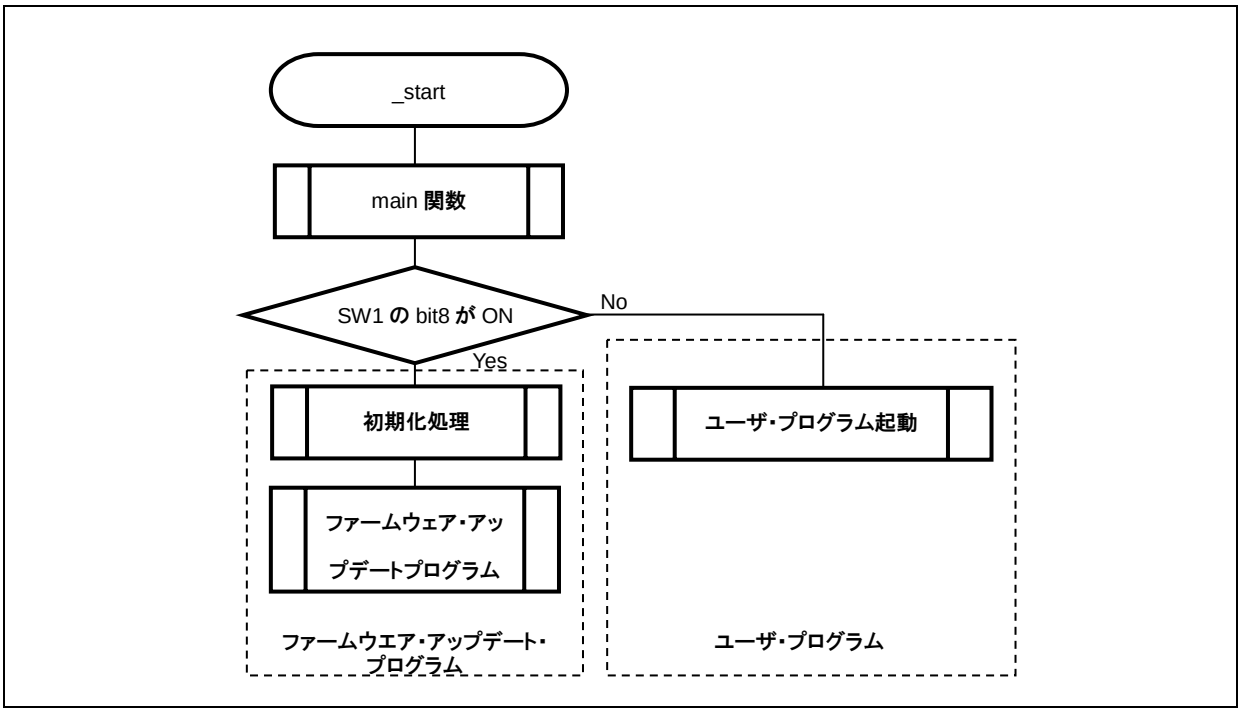
78K0Rマイコンでは，リセット後の初期化処理として，主に次のことを行います。

- スタック領域の確保とスタック・ポインタの設定
- main関数の引数領域の確保
- Data領域，スタック領域の初期化
- main関数への分岐

ブート処理の詳細については，CC78K0R　**ユーザーズ・マニュアル**を参照してください。

ファームウエア・アップデート・プログラムは，上記処理の後main関数へ来た場合に，ユーザ・プログラムへの分岐，ファームウェア・アップデートプログラム動作時に必要な78K0Rマイコンの初期化処理を行っています。次にブート処理の大まかな流れを示します。

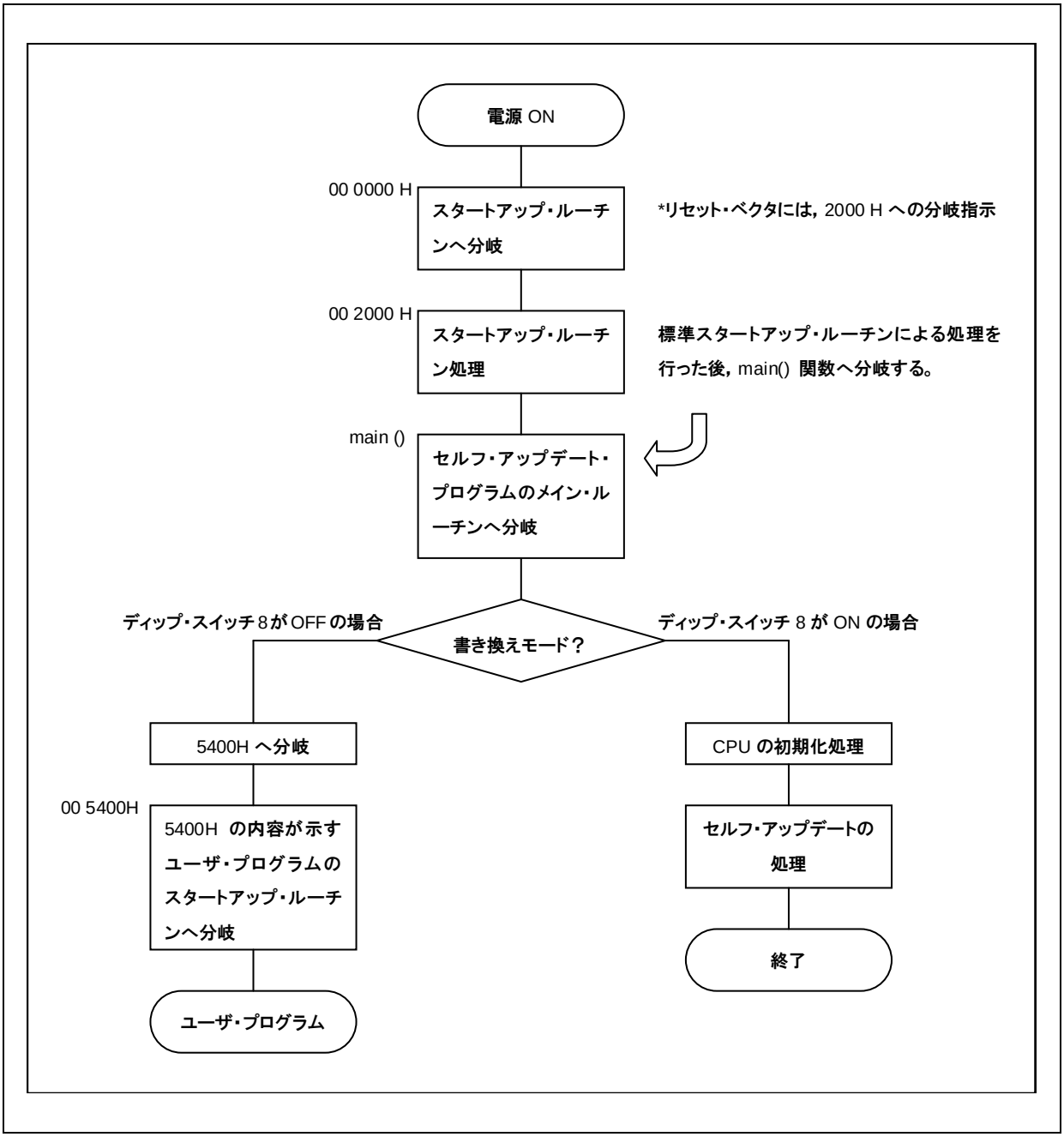
図3－4　ファームウエア・アップデート・プログラムのブート処理の流れ



3. 3. 1　電源投入時の動作フロー

次に，ファームウエア・アップデート・プログラムの電源投入時の動作フローについて説明します。

図3－5　電源投入時の動作フロー



ユーザ・プログラムが書き込まれていない（5400Hに分岐指示が無い場合も含む）状態で、ユーザ・プログラムを実行した場合の動作保証はありません。

評価ボードについては，TK-78K0R/KE3L+USBのユーザーズ・マニュアルを参照してください。

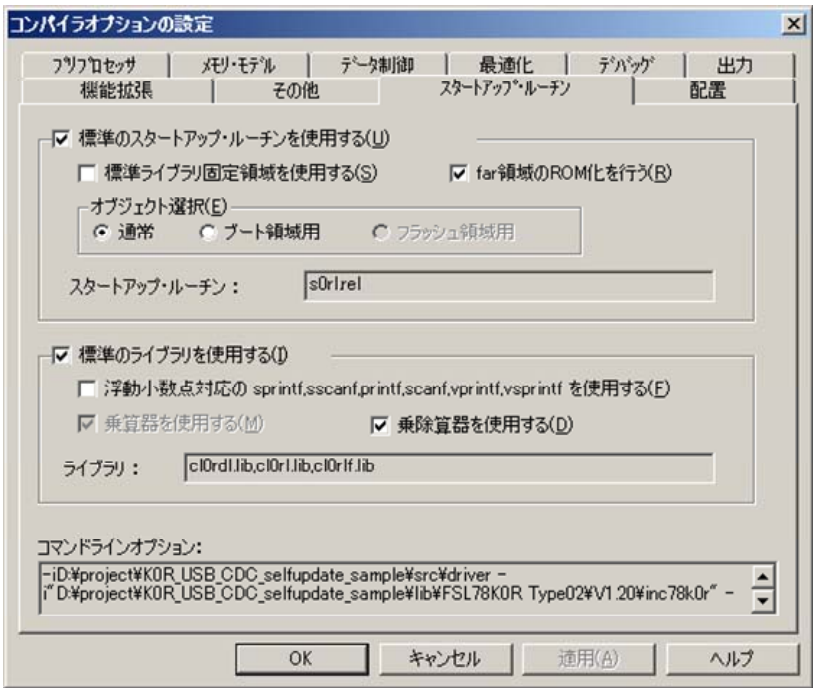
CPUについては，78K0R/KC3-L，78K0R/KE3-Lのユーザーズ・マニュアル　ハードウエア編を参照してください。

3. 3. 2　スタートアップ・ルーチン

セルフ・アップデート・プログラムのスタートアップ・ルーチンは，標準スタートアップ・ルーチンを使用しています。

PM+機能 ”コンパイラオプションの設定” により，スタートアップ・ルーチンを設定します。

図3－6　スタートアップ・ルーチンの選択



3. 3. 3　オプション・バイト

78K0R/Kx3-Lは，電源投入時またはリセットからの起動時に，自動的にオプション・バイトを参照し，指定された機能の設定を行います。その為，オプション・バイト空間には，予め設定を行っておく必要があります。

○ **ユーザ・オプション・バイト**（0 00C0H - 0 00C2H）
- **ウォッチドッグ・タイマの設定**
- **電源投入時のLVIの設定**
- **高速内蔵発振回路の周波数の設定**

○ **オンチップ・デバッグ・オプション・バイト**（0 00C3H）
- **オンチップ・デバッグに関する設定**
- **セキュリティに関する設定**

下表は，セルフ･アップデート・プログラムでの，各オプション・バイト空間への設定値となります。

表3－7　オプション・バイトの設定情報

オプション･バイト･アドレス	用途	設定値	設定内容
000C0H	ウォッチドッグ･タイマ機能	6EH	ウォッチドッグ･タイマ未使用
000C1H	高速内蔵発振回路の周波数 電源投入時のLVI	FAH	高速内蔵8MHまたは20MHz LVIデフォルト･スタート
000C2H	予約領域	FFH	-
000C3H	オンチップ･デバッグ	B5H	オンチップ･デバッグ許可 フラッシュ消去無し

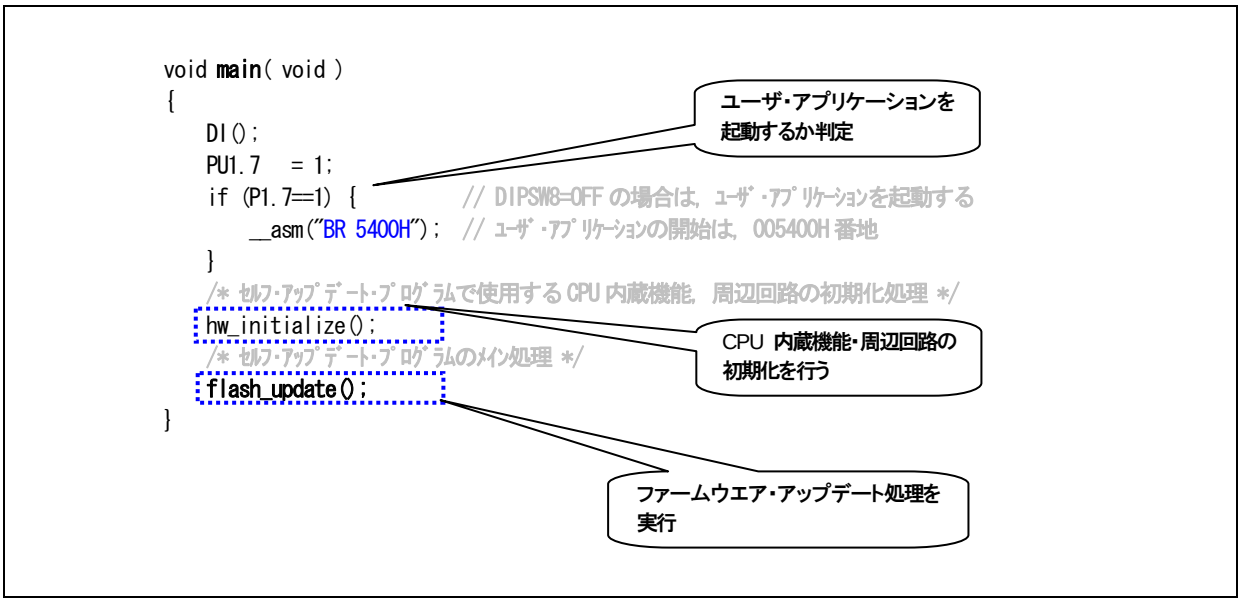
備考：　オプション・バイトに関しての詳細は，”78K0R/Kx3-L　ユーザーズ・マニュアル　第23章 オプション・バイト”　を参照して下さい。

3.4　メイン・ルーチン

ブート処理が終了すると，main関数へ分岐してメイン・ルーチンが実行されます。

メイン・ルーチンでは，CPU内蔵機能・周辺回路の初期化を行ったあと，ファームウエア・アップデート・プログラムを実行します。

図3－7　メイン・ルーチン



セルフ・アップデート・プログラムは，ユーザ・プログラムのスタート番地を 00 5400H 番地としています。セルフ・アップデートにより，ユーザ・プログラムの開始番地（ユーザ・プログラムでの 00 0000H 番地の配置情報 ）を， 00 5400H - 00 5403H に退避しています。

3. 4. 1　CPU内蔵機能・周辺回路の初期化処理（main.c）

フラッシュ・セルフ・アップデート・プログラムに入る前に、hw_initialize（）関数にてCPU内蔵機能の初期化・CPU周辺のハードウェア初期化を行います。

図3ー8　CPU・周辺回路の初期化

```
void hw_initialize( void )
{
    PMC    = 0x00;                  /* ﾐﾗｰﾘﾝｸﾞ                          */
    CMC    = 0x51;                  /* X1 発振/低消費 XT1 発振/fMX16MHz */
    OSTS   = 0x07;                  /* 発振安定時間 13.11ms             */
    CSC    = 0x00;                  /* X1/XT1 高速内蔵発振動作          */
    while (OSTC != 0xff);           /* 発振安定                         */
    OSMC   = 0x81;                  /* fCLK10MHz 以上                   */
    CKC    = 0x18;                  /* fCLK20MHz                        */
    DSCCTL = 0x00;                  /* 内蔵 20MHz を未使用              */
    PLLC   = 0x04;                  /* USBCK = fMX / 5 * 12             */
    UCKC   = 0x80;                  /* USB へ供給                       */
    while ((CKC & 0x20)!=0x20);     /* 高速ｼｽﾃﾑ                         */
    CSC    = 0x01;                  /* 高速内蔵 8MHz 停止               */

    以下省略
```

表3ー8　主な初期化内容

レジスタ	設定内容
PMC	00000H - 0FFFFH を F0000H – FFFFFH へミラー
CMC	高速システム・クロック：X1発振 (20MHz) サブシステム・クロック：XT1発振 (32.712KHz) より低消費電流モード
OSTS	発振安定時間 13.11ms
OSMC	CPU／周辺ハードウエア・クロックを高速システム・クロック (20MHz)で動作
CKC	CPU／周辺ハードウエア・クロックを高速システム・クロック (20MHz)
DSCCTL	20 MHz 高速内蔵発振クロックの動作停止
PLLC	USBクロックを, (20MHz / 5) に対して12逓倍
UCKC	USBクロック供給

3. 4. 2　セルフ・アップデート関数処理（flash_update.c）

セルフ・アップデート関数flash_update()**により、セルフ・アップデートの開始から終了までを処理します。**

図3ー9　flash_update()**関数**(1/5)

```
void flash_update( void )
{
    fsl_u08  fslRetCode;
    UINT32   first_addr;
    INT32    ret;
    INT32    lp;
    UINT8    code;

    per_7SegControl( LED_A , 0x0F );
    per_7SegControl( LED_B , 0x0F );

    /* FSL_GetActiveBootCluster()により，現在のｱｸﾃｨﾌﾞ･ｸﾗｽﾀ番号を取得します */
    FSL_Open();
    fslRetCode  = FSL_Init( fsl_data_buffer );
    fslRetCode  = FSL_ModeCheck();
    fslRetCode  = FSL_GetActiveBootCluster( fsl_data_buffer );
    FSL_Close();

    /* 使用するﾊﾞｯﾌｧ領域の初期化 */
    usbserial_init( serial_buf, SERIAL_BUF_SIZE );                ・・・①

    /* 内蔵 USB ﾌｧﾝｸｼｮﾝ･ｺﾝﾄﾛｰﾗの初期化(CDC) */
    usbf78k0r_init();                                             ・・・②

    /* CDC ｴﾆｭﾒﾚｰｼｮﾝを含んだ USB ﾒｯｾｰｼﾞ(開始ﾚｺｰﾄﾞ)処理を行います */
    ret  = recv_record();                                         ・・・③
    while (ret != RECORD_TYPE_START) {
        send_startres(RESPONSE_NAK);
        ret  = recv_record();
    }
```

① **セルフ・アップデート・プログラムが使用するバッファ・メモリを初期化します。**

② CPU**内蔵**USB**ファンクション・コントローラは，**usbf78k0r_init( ) **関数により**USB**ファンクション・コントローラ**(CDC) **としての初期化を行います。セルフ・アップデート・プログラムでは割り込みを使用しない為，**usbf78k0r_init( ) **関数内部では割り込み未使用の設定を行っています。**

③ USB **ファンクション・デバイス**（CDC）**としてのエニュメレーションを含んだ，最初の**USB **パケットの受信を行います。セルフ・アップデート・プログラムは，開始レコードが含まれるバルク・アウト転送を待ち続けます。**

図3ー10　flash_update()**関数**(2/5)

```
/* 開始ﾚｺｰﾄﾞに対する応答処理を行います */
send_startres( RESPONSE_ACK );
/* ﾃﾞｰﾀ･ﾚｺｰﾄﾞに対する処理を行います */
ret  = recv_record();                                          ・・・①
while (ret != RECORD_TYPE_DATA) {
    if (ret == RECORD_TYPE_START) {
        send_startres(RESPONSE_ACK);
    } else if (ret == RECORD_TYPE_END) {
        goto end;
    } else {
        send_datares(RESPONSE_NAK);
    }
    ret  = recv_record();
}
/* ﾃﾞｰﾀ･ﾚｺｰﾄﾞに含まれる､ｱﾄﾞﾚｽ情報の取得と､書き込み先ﾌﾞﾛｯｸを算出します */
flash_addr   = get_addr();                                     ・・・②
flash_block  = (UINT32)(flash_addr / FLASH_BLOCK_SIZE);        ・・・③

if (flash_block < 4) {                                         ・・・④
    /* ﾃﾞｰﾀ･ﾚｺｰﾄﾞがﾌﾞｰﾄ･ｸﾗｽﾀ０を含む場合の処理を開始します */
    /* ﾃﾞｰﾀ･ﾚｺｰﾄﾞをﾌﾞｰﾄ･ｸﾗｽﾀ１へ書き込みます */
    first_addr       = flash_addr;
    flash_buf.block  = flash_block + 4;
    flash_block      = recv_block();
    if (first_addr == 0) {
        /* ﾃﾞｰﾀ･ﾚｺｰﾄﾞがﾘｾｯﾄ･ﾍﾞｸﾀを含む場合､ﾕｰｻﾞ･ﾌﾟﾛｸﾞﾗﾑの
           ﾘｾｯﾄ･ﾍﾞｸﾀ情報をRAM領域に一時退避します */
        replace_apstart();                                     ・・・⑤
    }
```

① **開始レコードの受信完了後，データ・レコードのバルク・アウト転送を待ち続けます。**

② **受信したデータ・レコードから，フラッシュ・メモリの操作対照アドレス情報を抽出します。**

③ **同じく，抽出したアドレスからフラッシュ・メモリ・ブロック番号を算出します。**

④ **データ･レコード中にブート･クラスタ** 0(**フラッシュ･ブロック**0 - 3) **の情報が含まれる場合，対になるブート・クラスタ** 1(**フラッシュ･ブロック**4 - 7) **の領域に対して書き込みを行います。**

⑤ **データ・レコード中に，フラッシュ・ブロック**0 **の情報が含まれる場合，リセット・ベクタ情報のみを抽出し，一時**RAM **空間に退避します。ユーザ・プログラムにおけるリセット・ベクタの情報は，** 0000H - 0001H **番地に対しての書き込みを行いません。**

図3ー11　flash_update()**関数**(3/5)

```
while (flash_block < 4) {                                    ・・・①
    /* ﾃﾞｰﾀ･ﾚｺｰﾄﾞがﾌﾞｰﾄ･ｸﾗｽﾀ 0 を含んでいる場合は、
       ﾌﾞｰﾄ･ｸﾗｽﾀ 1 へ連続して書き込みを行います */
    if ( (ret=flashrom_write(&flash_buf)) != 0 ) {
        code    = ERROR_FLASH_WRITE;
        goto error;
    }
    flash_buf.block  = flash_block + 4;
    flash_block      = recv_block();
}
if (first_addr == 0) {
    /* ﾌﾞｰﾄ･ｽﾜｯﾌﾟが必要とされる場合､FSL_InvertBootFlag により
       ﾌﾞｰﾄ･ｸﾗｽﾀを示すﾌﾞｰﾄ･ﾌﾗｸﾞの情報を反転します */
    /* ﾌﾞｰﾄ･ｽﾜｯﾌﾟは次のﾘｾｯﾄの後となります */
    FSL_Open();
    if ( (fslRetCode=FSL_Init(fsl_data_buffer))!=0 ) {
        code  = ERROR_FLASH_WRITE;
        FSL_Close();
        goto error;
    }
    if ( (fslRetCode=FSL_ModeCheck())!=0 ) {
        code  = ERROR_FLASH_WRITE;
        FSL_Close();
        goto error;
    }
    if ( (fslRetCode=FSL_InvertBootFlag())!=0 ) {            ・・・②
        code  = ERROR_FLASH_WRITE;
        FSL_Close();
        goto error;
    }
    if ( (fslRetCode=FSL_GetActiveBootCluster(fsl_data_buffer))!=0 ) {
        code  = ERROR_FLASH_WRITE;
        FSL_Close();
        goto error;
    }
    FSL_Close();
}
```

① **データ･レコード中にブート･クラスタ 0 の情報が含まれている期間中は,ブート･クラスタ 1に連続した書き込みを行います。**

② **現ブート･クラスタ 1に対しての書き込みが完了した後，ブート・フラグの入れ替えを実施します。ブート・クラスタの入れ替え処理は，セルフ・プログラミング・ライブラリ**（FSL_InvertBootFlag()）**により，ブート・フラグの反転のみを行います。（ この時点では，ブート・スワップはしていません ）**

図3ー12　flash_update()**関数**(4/5)

```
        if ( (ret=flashrom_write(&flash_buf)) != 0 ) {
            code    = ERROR_FLASH_WRITE;
            goto error;
        }
        while (flash_block < 8) {                                    ・・・・①
            /* ﾃﾞｰﾀ･ﾚｺｰﾄﾞがﾌﾞｰﾄ･ｸﾗｽﾀ１を含んでいる場合は、
               ﾌﾞｰﾄ･ｸﾗｽﾀ０へ連続して書き込みを行います */
            flash_buf.block  = flash_block - 4;
            flash_block      = recv_block();
            if ( (ret=flashrom_write(&flash_buf)) != 0 ) {
                code    = ERROR_FLASH_WRITE;
                goto error;
            }
        }
        if (first_addr == 0) {
            if ( (ret=write_apstart()) != 0 ) {                      ・・・・②
                code  = ERROR_FLASH_WRITE;
                goto error;
            }
        }
    }
    while (flash_block <= WRITE_MAX_BLOCK) {                         ・・・・③
        /* ﾌﾞｰﾄ･ｸﾗｽﾀを含まないその他の領域に連続して書き込みを行います */
        flash_buf.block  = flash_block;
        flash_block      = recv_block();
        if ( (ret=flashrom_write(&flash_buf)) != 0 ) {
            code = ERROR_FLASH_WRITE;
            goto error;
        }
    }
```

①　ブート・クラスタ0に対する書き込みを行います。

②　既にRAMに退避されているユーザ・プログラムのリセット・ベクタを，退避先のフラッシュ・メモリ・アドレスへ書き込みます。

③　ブート･クラスタ以外の番地に対して，連続して書き込みを行います。

図3ー13　flash_update()**関数**(5/5)

```
    if (flash_block != RECEIVE_END_RECORD) {
        code = ERROR_INVALID_DATA;
        goto error;
    }
end:
    /* 以降はｾﾙﾌ･ｱｯﾌﾟﾃﾞｰﾄの終端処理となります */
    ret  = inrec.type;
    while (1) {                                                    ・・・・①
        if (ret == RECORD_TYPE_END) {                              ・・・・②
            send_endres(RESPONSE_ACK);
            per_7SegControl( LED_OFF , 0x0F );
            per_7SegControl( LED_OFF , 0x0F );
        } else if (ret == RECORD_TYPE_DATA) {
            send_datares(RESPONSE_ACK);
        } else if (ret == RECORD_TYPE_START) {
            send_startres(RESPONSE_ACK);
        } else {
            send_datares(RESPONSE_NAK);
        }
        ret  = recv_record();
    }
error:                                                             ・・・・③
    per_7SegControl( LED_A , 0x0E );
    per_7SegControl( LED_B , 0x0E );
    while (1) {
        send_errorres(code);
        ret  = recv_record();
    }
}
```

①　書き込み終了後に分岐する本ループは，抜け出す事はありません。

②　正常書き込み終了による実行パスです。以後はリセットを期待し続けます。

③　異常処理とします。

3. 4. 3　セルフ・アップデート内部関数

ここではflash_update()**関数内部で呼び出している関数について説明します。**

3.4.3.1 usbserial_init()

usbserial_init()**関数は，セルフ・アップデート・プログラムが使用するバッファ領域を初期化します。**

図3ー14　usbserial_init()

```
static void usbserial_init( UINT8 *buf , INT32 buf_len )
{
    INT32  lp;

    recv_buf       = buf;
    recv_buf_size  = buf_len;

    for ( lp=0;lp<sizeof(flash_buf.data);lp++ ) {
        flash_buf.data[lp]  = 0xff;
    }

    usbserial_clear_buffer();
}
```

3.4.3.2　recv_record()

recv_record( ) **関数は，受信したレコードが有効レコードであるかどうかを解析します。**

図3ー15　recv_record()

```
static INT32 recv_record( void )
{
    INT32   ret,i,chk;

    usbserial_clear_buffer();

    /* ﾚｺｰﾄﾞの受信を行います */
    ret  = usbserial_recv(&inrec.type, 1);
    ret  = usbserial_recv(&inrec.len, 1);
    if (inrec.len == 0) {
        ret  = -1;
        goto end;
    }
    chk  = inrec.len;

    /* ﾚｺｰﾄﾞ長分の USBﾊﾟｹｯﾄ･ﾃﾞｰﾀをｾﾙﾌ･ｱｯﾌﾟﾃﾞｰﾄ･ﾌﾟﾛｸﾞﾗﾑ用ﾊﾞｯﾌｧへ転送します */
    if (inrec.len > 1) {                                          ・・・・①
        ret    = usbserial_recv(inrec.data, inrec.len - 1);
        for (i = inrec.len - 2; i >= 0; i--) {
            chk += inrec.data[i];
        }
    }
    ret  = usbserial_recv(&inrec.sum, 1);
    chk    ^= 0xffff;
    chk    &= 0x00ff;
    if (chk != inrec.sum) {
        ret = -1;
        goto end;
    }
    ret    = inrec.type;
end:
    return ret;
}
```

① レコード長分データを，セルフ・アップデート・プログラムで使用する領域に転送します。

3.4.3.3　usbserial_recv()

usbserial_recv( ) **関数は，**USB**イベント（ 割り込み要因フラグ ）による，バルク・アウト転送の制御を行う** usbf78k0r_intusbf( ) **関数をポーリングします。**

引数として渡されたバイト長分のUSB**パケット・データの受信を行います。**

図3－16　usbserial_recv()

```
static INT32 usbserial_recv( UINT8 *data , INT32 len )
{
    INT32  num = 0;

    while (num < len) {
        while (recv_len == 0) {                                    ・・・・①
            /* 期待する受信ﾊﾞｲﾄ長に達するまで、INTUSB 割り込み処理同等の
               INTUSB 要因解析処理をﾎﾟｰﾘﾝｸﾞします*/
            usbf78k0r_intusbf0();                                  ・・・・②
        }
        data[num]    = recv_buf[recv_idx];                         ・・・・③
        recv_idx++;
        if (recv_idx >= recv_buf_size) {
            recv_idx -= recv_buf_size;
        }
        recv_len--;
        num++;
    }
    return num;
}
```

① recv_len **は** usbf78k0r_intusbf0( ) **関数で扱われます。**

② USB**イベントに対する処理を行います。エニュメレーションを含むコントロール転送，及びバルク転送に対する処理を行います。**

③ **受信したデータを，セルフ・アップデート・プログラム用バッファ領域を転送します。**

3.4.3.4　usb_read()

usb_read( ) **関数は，データ受信の契機で呼び出されます。受信されたデータを，**RAM**領域にコピーします。**

図3ー17　usb_read()

```
void usb_read( void )
{
    INT32  iidx;
    INT32  oidx;
    INT32  num;
    INT32  len;

    /* 本関数はUSBﾌｧﾝｸｼｮﾝ･ﾄﾞﾗｲﾊﾞへの提供関数です */
    /* Bulk Out転送に伴う、Bulk Out用ﾃﾞｰﾀ･ﾊﾞｯﾌｧへの蓄積と
       USB/Serial変換用ﾊﾞｯﾌｧへの転送を行います */

    len  = usbf78k0r_data_receive(bko1_buf, USERBUF_SIZE, C_BKOUT);
    num  = recv_buf_size - recv_len;
    if (num > (INT32)len) {
        num = (INT32)len;
    }
    oidx   = recv_idx + recv_len;
    iidx   = 0;
    recv_len  += num;
    while (num > 0) {
        if (oidx >= recv_buf_size) {
            oidx -= recv_buf_size;
        }
        recv_buf[oidx] = bko1_buf[iidx];
        num--;
        iidx++;
        oidx++;
    }
}
```

3.4.3.5　copy_block()

copy_block( ) **関数は，指示されるフラッシュ・メモリ・ブロックの全情報を**RAM**領域に退避させます。**

図3－18　copy_block()

```
static void copy_block( INT32 block )
{
    UINT16  *in;
    UINT16  *out;
    INT32   i;

    in  = (UINT16 *)(block * FLASH_BLOCK_SIZE);
    out = (UINT16 *)flash_buf.data;

    for ( i=FLASH_BLOCK_SIZE / 2 - 1 ; i >= 0 ; i-- ) {
        out[i]  = in[i];
    }
}
```

3.4.3.6　replace_apstart()

replace_apstart( ) **関数は,ユーザ・プログラムにおけるリセット・ベクタの情報を**RAM**空間へ退避させます。**

図3－19　replace_apstart()

```
static void replace_apstart( void )
{
    UINT16  *in  = (UINT16 *)0;
    UINT16  *out;

    out         = (UINT16 *)flash_buf.data;
    start_inst  = out[0];
    out[0]      = in[0];
}
```

3.4.3.7　write_apstart()

write_apstart( ) **関数は，**RAM**に退避されたユーザ・プログラムにおけるリセット・ベクタの情報を，ユーザ・プログラムの開始アドレス**（00 5400H）**に書き込みます。**

図3ー20　write_apstart()

```
static INT32 write_apstart( void )
{
    UINT16  *out;
    INT32   num      = 0;

    /* 00 5400H 番地から 1 ﾌﾞﾛｯｸ分を RAM 空間へ転送 */
    out              = (UINT16 *)flash_buf.data;                          ・・・・①
    flash_buf.block  = APSTART_ADDR / FLASH_BLOCK_SIZE;
    copy_block((INT32)flash_buf.block);
    flash_buf.data_length  = FLASH_BLOCK_SIZE;

    /* RAM に展開された 00 5400H 番地を含む 1 ﾌﾞﾛｯｸ領域に対して
       ﾕｰｻﾞ･ﾘｾｯﾄ･ﾍﾞｸﾀの情報を，退避済み領域から読み出し分岐命令を生成する*/
    out[0]  = (UINT16)start_inst;                                        ・・・・②
    out[0]  = 0xEC | (UINT16)((start_inst << 8) & 0xFF00);               ・・・・③
    out[1]  = ((UINT16)((start_inst >> 8) & 0xFF));
    num     = flashrom_write(&flash_buf);                                ・・・・④
    return  num;
}
```

① ユーザ・プログラム開始ブロックを全コピーします。

② ユーザ・プログラムでのリセット・ベクタ情報を退避先のRAM**領域から読み出します。**

③ 読み出した情報に分岐命令（BR）**を付加します。**

④ ユーザ・プログラム開始ブロックを書き換えます。

備考：　78K0R**におけるリセット・ベクタ・テーブルは分岐命令を必要としません。その為，ユーザ・プログラム開始ブロックを書き換える際は，分岐命令（ 上記リストでは**ECH)**を付加した上で書き込みを行っています。**

3.4.3.8　send_startres()

send_startres( ) **関数は，ホスト・マシン**（PC）**に開始レコードに対するレスポンスを送信します。**

図3－21　send_startres()

```
static void send_startres( UINT8 kbn )
{
    UINT16     sum;

    outrec[0]  = RECORD_TYPE_START;
    outrec[1]  = 3;
    outrec[2]  = kbn;
    outrec[3]  = inrec.data[0];
    sum        = outrec[1] + outrec[2] + outrec[3];
    sum        ^= 0xffff;
    sum        &= 0x00ff;
    outrec[4]  = (UINT8)sum;
    usbf78k0r_send_buf(outrec, 5);
}
```

3.4.3.9　send_datares()

send_datares( ) **関数は，ホスト・マシン**（PC）**にデータ･レコードに対するレスポンスを送信します。**

図3－22　send_datares()

```
static void send_datares( UINT8 kbn )
{
    INT32   i;
    UINT16  sum = 0;

    outrec[0]   = RECORD_TYPE_DATA;
    outrec[1]   = 6;
    outrec[2]   = kbn;
    outrec[3]   = inrec.data[0];
    outrec[4]   = inrec.data[1];
    outrec[5]   = inrec.data[2];
    outrec[6]   = inrec.data[3];
    for (i=6;i>0;i--) {
        sum     += outrec[i];
    }
    sum         ^= 0xffff;
    sum         &= 0x00ff;
    outrec[7]   = (UINT8)sum;
    usbf78k0r_send_buf(outrec, 8);
}
```

3.4.3.10　send_endres()

send_endres( ) **関数は，ホスト・マシン**（PC）**に終了レコードに対するレスポンスを送信します。**

図3ー23　send_endres()

```
static void send_endres( UINT8 kbn )
{
    UINT16     sum;

    outrec[0]  = RECORD_TYPE_END;
    outrec[1]  = 3;
    outrec[2]  = kbn;
    outrec[3]  = inrec.data[0];
    sum        = outrec[1] + outrec[2] + outrec[3];
    sum        ^= 0xffff;
    sum        &= 0x00ff;
    outrec[4]  = (UINT8)sum;
    usbf78k0r_send_buf(outrec, 5);
}
```

3.4.3.11　send_errorres()

send_endres( ) **関数は，ホスト・マシン**（PC）**に異常レコードを送信します。**

図3ー24　send_errorres()

```
static void send_errorres( UINT8 code )
{
    UINT16     sum;

    outrec[0]  = inrec.type;
    outrec[1]  = 3;
    outrec[2]  = RESPONSE_ERROR;
    outrec[3]  = code;
    sum        = outrec[1] + outrec[2] + outrec[3];
    sum        ^= 0xffff;
    sum        &= 0x00ff;
    outrec[4]  = (UINT8)sum;
    usbf78k0r_send_buf(outrec, 5);
}
```

3.4.3.12　recv_block()

recv_block( ) **関数は，1ブロック分の書き換えデータをセルフ・アップデート・プログラムで使用する**RAM**空間へ蓄積します。**

図3ー25　recv_block() (1/2)

```
static INT32 recv_block( void )
{
    INT32   ret;
    UINT32  in_addr;
    UINT32  out_addr;
    INT32   in_len;
    INT32   in_idx;
    INT32   out_idx;

    /* 該当のﾌﾗｯｼｭ･ﾌﾞﾛｯｸを RAM に全ｺﾋﾟｰする */
    copy_block(flash_block);

    flash_buf.data_length  = FLASH_BLOCK_SIZE;
    out_addr               = flash_block * FLASH_BLOCK_SIZE;
    do {
        if (out_addr == flash_addr) {
            out_idx  = 0;
            in_idx   = 4;
        } else if (out_addr > flash_addr) {
            in_idx   = out_addr - flash_addr + 4;
            out_idx  = 0;
        } else {
            out_idx  = flash_addr - out_addr;
            in_idx   = 4;
        }
        in_len       = inrec.len - 1;
        /* 1ﾚｺｰﾄﾞの終端またはﾌﾞﾛｯｸの終端まで読み出す */
        while (in_idx < in_len) {
            if (out_idx >= FLASH_BLOCK_SIZE) {
                /*ﾌﾞﾛｯｸの終端*/
                ret  = flash_block + 1;
                goto end;
            }
            flash_buf.data[out_idx] = inrec.data[in_idx];
            out_idx++;
            in_idx++;
        }
```

図3－26　recv_block() (2/2)

```
        /* ACK送信後に次のﾚｺｰﾄﾞを受信 */
        send_datares(RESPONSE_ACK);
        ret  = recv_record();

        /* 新たに受信したﾚｺｰﾄﾞを判定する */
        while (1) {
            if (ret == RECORD_TYPE_DATA) {
                in_addr  = get_addr();
                if (in_addr > flash_addr) {
                    break;
                }
                send_datares(RESPONSE_ACK);
            } else if (ret == RECORD_TYPE_END) {
                ret = RECEIVE_END_RECORD;
                goto end;
            } else if (ret == RECORD_TYPE_START) {
                send_startres(RESPONSE_ACK);
            } else {
                send_datares(RESPONSE_NAK);
            }
            ret  = recv_record();
        }
        flash_addr = in_addr;
        ret        = flash_addr / FLASH_BLOCK_SIZE;
    } while (ret == flash_block);
end:
    return ret;
}
```

3.4.3.13　flashrom_write()

flashrom_write( ) **関数は，1ブロック単位で該当するフラッシュ・メモリ・ブロックに対する書き込みを行います。**

図3ー27　flashrom_write() (1/2)

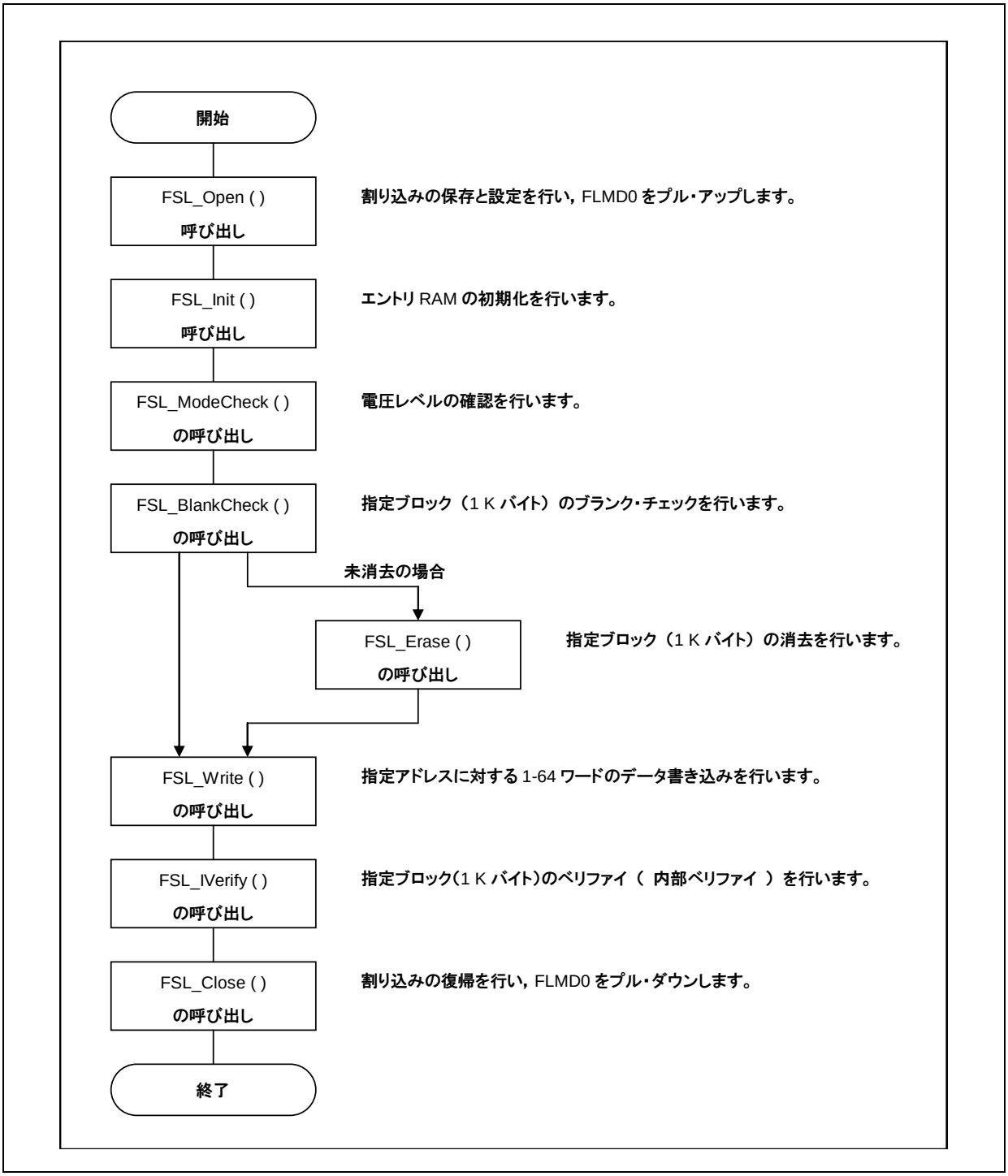
```
static fsl_u08 flashrom_write( struct flash_data *fd )
{
    fsl_u08  fslRetCode = 0;
    UINT32   wAdr;

    fd->pDataBuffer = fsl_data_buffer;
    /* ﾌﾗｯｼｭ･ｾﾙﾌ･ﾌﾟﾛｸﾞﾗﾐﾝｸﾞの開始宣言を行います */
    FSL_Open();
    do {
        /* 自信を書き換える事はNG */
        if ( (fd->block>=0x08) && (fd->block<0x15) ) {
            fslRetCode  = 1;
            break;
        }
        if ( (fslRetCode=FSL_Init( fd->pDataBuffer )) != FSL_OK ) {
            break;
        }
        if ( (fslRetCode=FSL_ModeCheck()) != FSL_OK ) {
            break;
        }
        if ( (fslRetCode=(fsl_u08)FSL_BlankCheck((fsl_u16)fd->block)) == FSL_ERR_BLANKCHECK ) {
            if ( (fslRetCode=FSL_Erase( (fsl_u16)fd->block )) != FSL_OK ) {
                break;
            }
        }
        /* 1ﾌﾞﾛｯｸ分を64ﾜｰﾄﾞ単位で書き換え開始 */
        wAdr  = (fd->block * FLASH_BLOCK_SIZE);
```

図3ー28　flashrom_write() (2/2)

```
        /* ｵﾌｾｯﾄ 0000H-00FFH を書き換え */
        memcpy( fd->pDataBuffer , fd->data , 256 );
        if ( (fslRetCode=FSL_Write( wAdr , 64 )) != FSL_OK ) {
            break;
        }
        /* ｵﾌｾｯﾄ 0100H-01FFH を書き換え */
        wAdr  = wAdr + 256;
        memcpy( fd->pDataBuffer , fd->data + 256 , 256 );
        if ( (fslRetCode=FSL_Write( wAdr , 64 )) != FSL_OK ) {
            break;
        }
        /* ｵﾌｾｯﾄ 0200H-02FFH を書き換え */
        wAdr  = wAdr + 256;
        memcpy( fd->pDataBuffer , fd->data + 256 + 256 , 256 );
        if ( (fslRetCode=FSL_Write( wAdr , 64 )) != FSL_OK ) {
            break;
        }
        /* ｵﾌｾｯﾄ 0300H-03FFH を書き換え */
        wAdr  = wAdr + 256;
        memcpy( fd->pDataBuffer , fd->data + 256 + 256 + 256 , 256 );
        if ( (fslRetCode=FSL_Write( wAdr , 64 )) != FSL_OK ) {
            break;
        }
        /* ﾌﾞﾛｯｸ書き換え終端として FSL_IVerify()を呼び出す */
        if ( (fslRetCode=FSL_IVerify( (fsl_u16)fd->block )) != FSL_OK ) {
            break;
        }
    } while(0);
    /* ﾌﾗｯｼｭ･ｾﾙﾌ･ﾌﾟﾛｸﾞﾗﾐﾝｸﾞの終了宣言を行います */
    FSL_Close();
    return fslRetCode;
}
```

下図に，フラッシュ・ブロック・ライト時の動作フローを示します。

図3－29　フラッシュ・メモリ書き込みフロー



備考：　78K0Rマイクロコントローラは，FLMD0端子のプルアップ機能とプルダウン機能を内蔵しており，FSL_Open ( ) と FSL_Close ( )の各関数によってFLMD0端子をハイ・レベルやロウ・レベルにする事が可能です。

FLMD0端子は，フラッシュ・セルフ・プログラミングの実行中はハイ・レベルにする必要があり，FLMD0端子がロウ・レベルの場合，フラッシュ・メモリの書き換え動作は行われません。

3.4.3.14　データ・タイプ

セルフ・アップデート・プログラムにおける，データ・タイプを下表に示します。
データ・タイプは，Types.h により定義されています。

表3－9　データ・タイプ

データ・タイプ	指定子	78K0Rにおける有効範囲
INT8	signed char	符号つき8ビット整数
INT16	signed short	符号つき16ビット整数
INT32	signed long	符号つき32ビット整数
UINT8	unsigned char	符号なし8ビット整数
UINT16	unsigned short	符号なし16ビット整数
UINT32	unsigned long	符号なし32ビット整数
STATUS	int	符号つき16ビット整数

3. 6　内蔵フラッシュ・メモリへの書き込み

ファームウエア・アップデート・プログラムは，内蔵フラッシュ・メモリの内容を書き換えることにより，ファームウエア，およびメモリ上の任意の領域を更新します。

ファームウエア・アップデート・プログラムでは，内蔵フラッシュ・メモリの書き込みを行うために，セルフ・ライブラリを使用します。セルフ・ライブラリは使用するデバイスに対応したものが必要で，本評価ボードではType02のセルフ・ライブラリを使用します。セルフ・ライブラリの詳細については，78K0Rマイクロコントローラ　ユーザーズ・マニュアル　フラッシュ・メモリ・セルフ・プログラミング・ライブラリ Type02 Ver.1.20を参照してください。

3. 6. 1　フラッシュ・メモリの書き込み処理

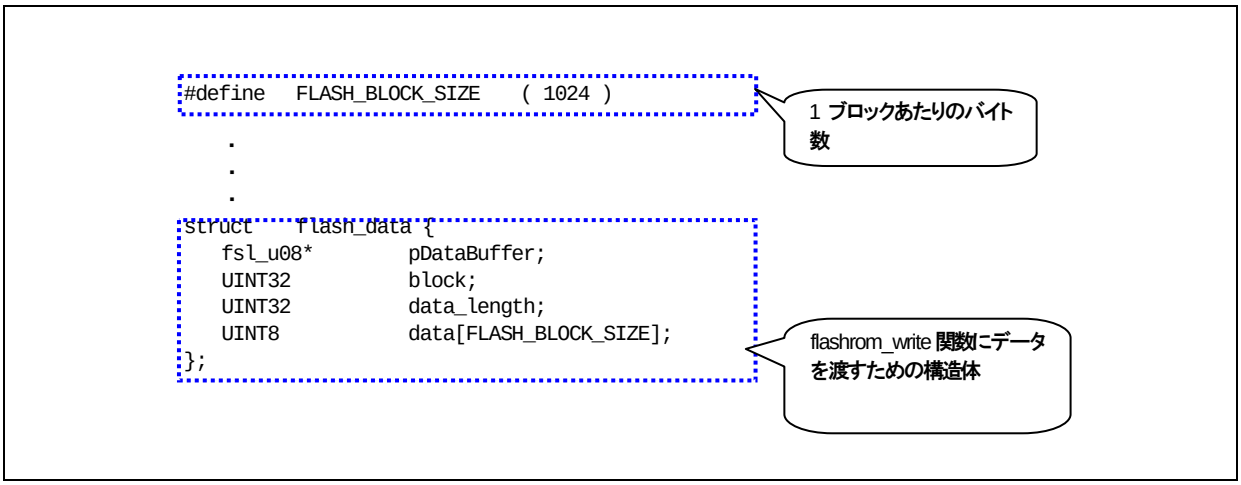
評価ボードで使用される製品の内蔵フラッシュ・メモリは、下記に示すブロックで構成されます。

- μPD78F1026（78K0R/KE3-L）　：　ブロック0-127

セルフ書き換えは，ブロック単位で内蔵フラッシュ・メモリへ書き込みを行います。

flash_update.cで定義しているflashrom_write関数は，内蔵フラッシュ・メモリの指定されたブロックへ書き込み処理を行います。書き込みを行うブロック，書き込むデータは，flash_data構造体の形式で指定します。flash_data構造体は，flash_update.cで宣言しています。

図3－30　flash_data構造体の定義



flash_data構造体のpDataBufferメンバはフラッシュ・セルフ・プログラミング・ライブラリ用バッファ，blockメンバは書き込みを行うブロックの番号，dataメンバは書き込むデータ，data_lengthメンバは書き込むデータのバイト数を指定します。

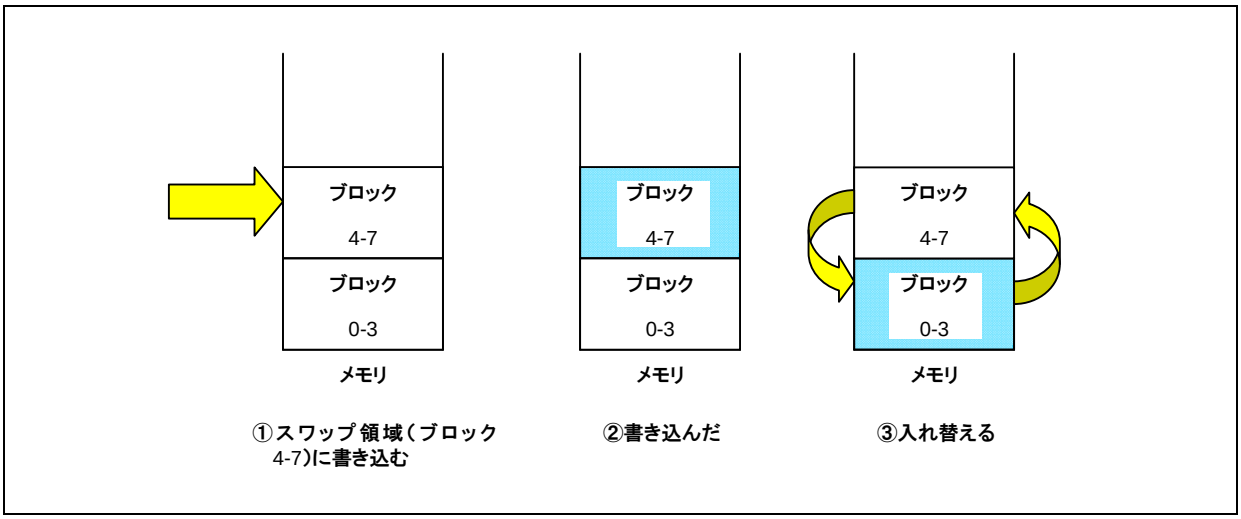
flashrom_write関数は，セルフ・ライブラリが提供しているフラッシュ関数を使用して内蔵フラッシュ・メモリへデータを書き込みます。

3. 6. 2　ブート・スワップ機能

対象としている製品は，ユーザ・プログラムを書き換えるときに，ブート領域の書き換え中に電源遮断等の事故が発生しても，ブート領域を保障し，正常に動作するためにブート・スワップ機能をサポートしています。

対象としている製品は，ブロック番号0-3とブロック番号4-7を入れ替えること（スワップ）ができます。

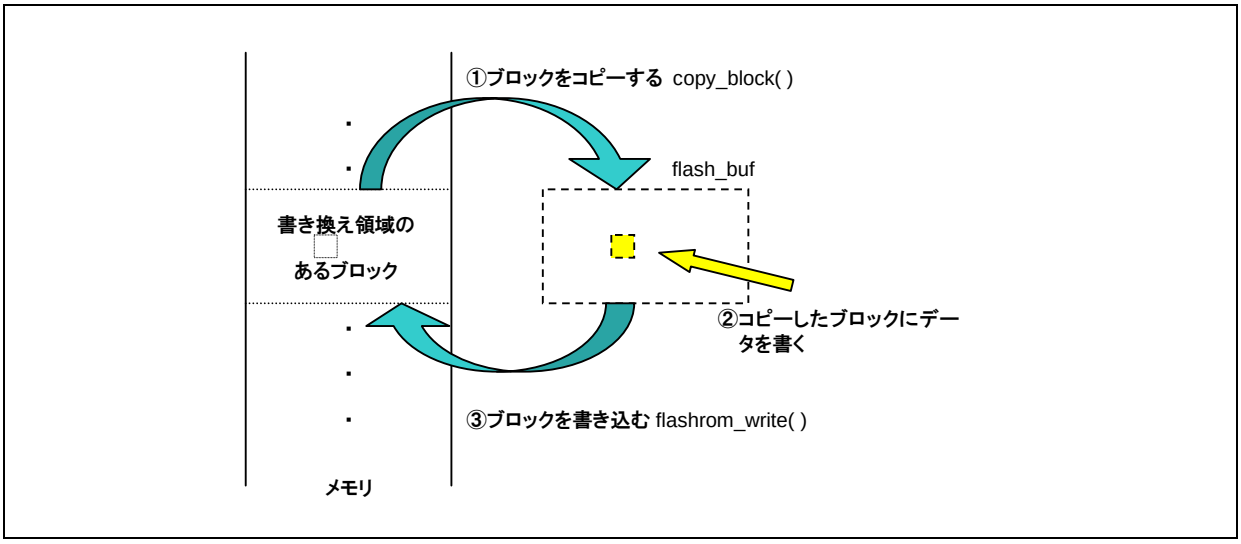
図3－31　ユーザ・プログラム書き換え時のブート・スワップ機能



3. 6. 3　ファームウエア・アップデート処理

内蔵フラッシュ・メモリを書き換える場合は，ブロック単位で行います。ファームウエア・アップデート・プログラムでは，書き換える領域のブロックをコピーし，コピーしたブロックを書き換えてから，そのブロックを書き込みます。これにより，1バイト単位の書き換えを実現します。

図3－32　書き換え処理イメージ



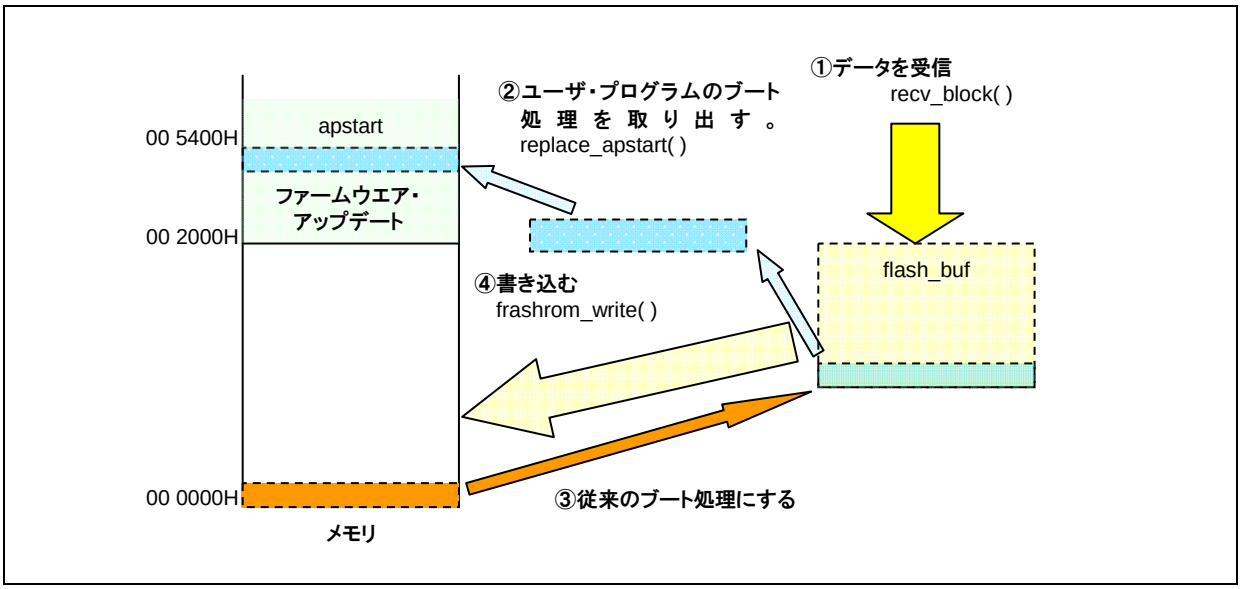
flash_update.cのflash_update関数で，PCからのデータ受信／応答，データの書き換えを行います。

3. 6. 4　ユーザ・プログラムの書き換え

ユーザ・プログラムを書き込むときは，apstartセクション内を変更し，起動時にユーザ・プログラムが動作するようにします。

受信したユーザ・プログラムのブート処理（リセット・セクション）を従来（ファームウエア・アップデート）のブート処理での分岐先（ユーザ・プログラムへジャンプするコード：apstartセクション）として取り出して書き込み，元のブート処理を従来（ファームウエア・アップデート）のブート処理に変更します。これによりブート処理部分を保持し，ファームウエア・アップデートの操作を再度実施できようになります。

図3－33　ユーザ・プログラム書き換え時のブート処理書き換えイメージ



起動処理では，最初にメイン関数処理へ移動し，その処理内で起動条件（SW8状態）を判定し，apstartへ移動し，そこからユーザ・プログラムの先頭へ移動します。

図3－34　ユーザ・プログラムへの分岐イメージ

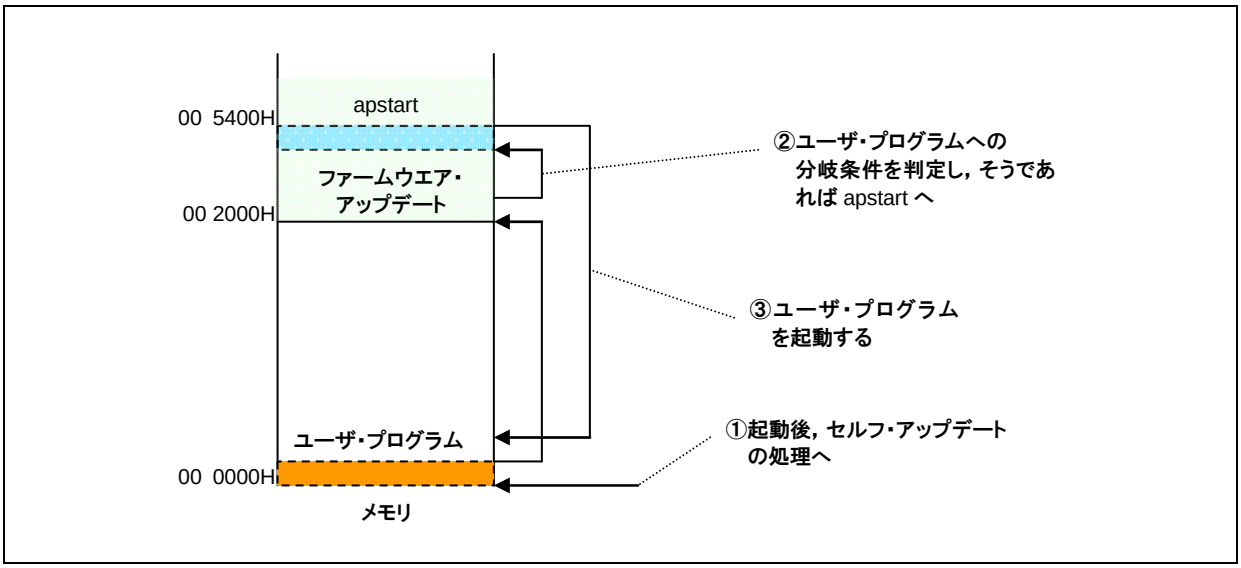
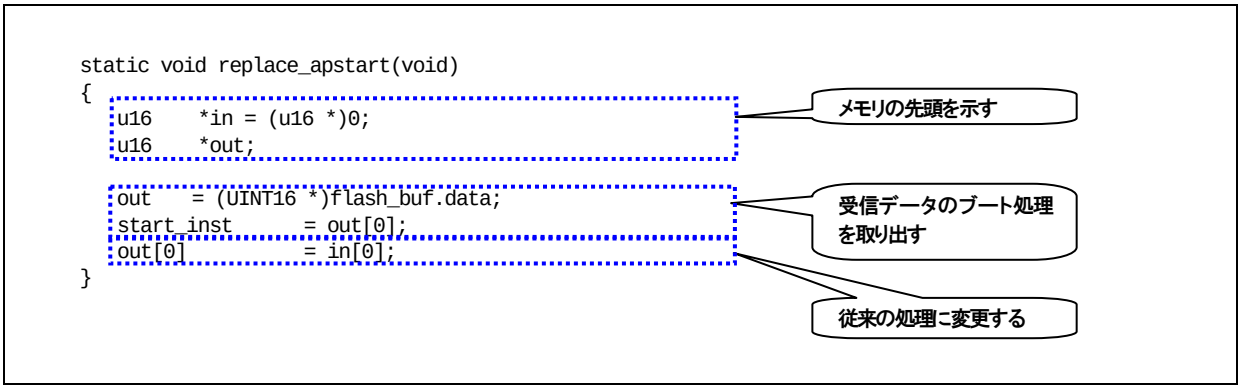
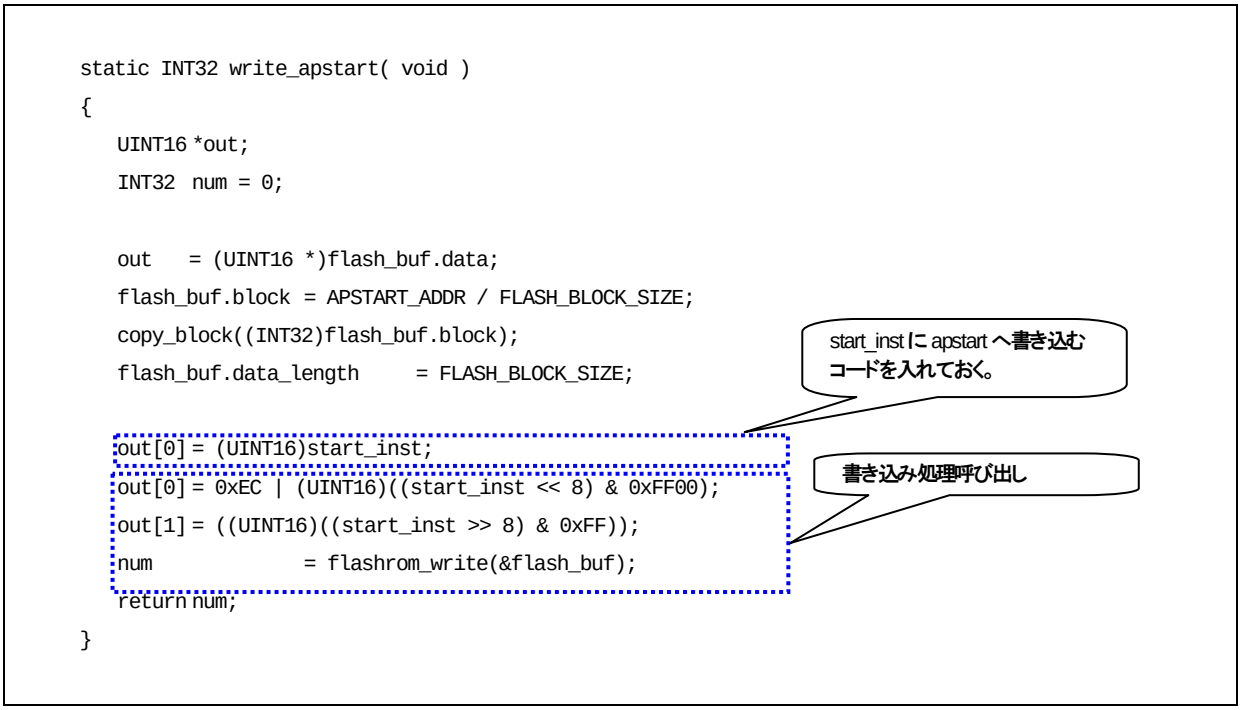


図3－35　ブート処理の変更



apstartへ書き込む処理を編集します。

図3－36　ブート処理の書き換え



3. 6. 5　データ受信処理

ファームウエア・アップデート・プログラムは，PCとのシリアル通信を行い，書き換えるデータを受信します。通信インタフェース仕様については，6. 1　書き換え通信インタフェース仕様を参照してください。

図3－37　1ブロック・データ受信

```
static INT32 recv_block( void )
{
        INT32           ret;
        UINT32          in_addr;
        UINT32          out_addr;
        INT32           in_len;
        INT32           in_idx;
        INT32           out_idx;

    copy_block(flash_block);
    flash_buf.data_length  = FLASH_BLOCK_SIZE;
    out_addr = flash_block * FLASH_BLOCK_SIZE;

    do {
        if (out_addr == flash_addr) {
                out_idx   = 0;
                in_idx    = 4;
        } else if (out_addr > flash_addr) {
                in_idx    = out_addr - flash_addr + 4;
                out_idx   = 0;
        } else {
                out_idx   = flash_addr - out_addr;
                in_idx    = 4;
        }
        in_len              = inrec.len - 1;
        while (in_idx < in_len) {
                if (out_idx >= FLASH_BLOCK_SIZE) {
                        ret       = flash_block + 1;
                        goto end;
                }
                flash_buf.data[out_idx] = inrec.data[in_idx];
                out_idx++;
                in_idx++;
        }
        send_datares(RESPONSE_ACK);
        ret       = recv_record();
        while (1) {
                if (ret == RECORD_TYPE_DATA) {
                        in_addr   = get_addr();
                        if (in_addr > flash_addr) {
                                break;
                        }
                        send_datares(RESPONSE_ACK);
                } else if (ret == RECORD_TYPE_END) {
                        ret = RECEIVE_END_RECORD;
                        goto end;
                } else if (ret == RECORD_TYPE_START) {
                        send_startres(RESPONSE_ACK);
                } else {
                        send_datares(RESPONSE_NAK);
                }
                ret       = recv_record();
        }
        flash_addr          = in_addr;
        ret                         = flash_addr / FLASH_BLOCK_SIZE;
    } while (ret == flash_block);
end:
        return ret;
}
```

指定のブロックをコピーする

1レコード分を受信する。
最後まで受信するか，ブロックがいっぱいになったらループを抜ける。

ACKを送信後
再度レコードを受信する

受信レコードを判定する

ブロックが変わったらループを抜ける

次に示すのは，1レコードを受信する処理です。

図3－38　1レコード受信

```
static INT32 recv_record( void )
{
        INT32           ret;
        INT32           i;
        UINT16          chk;

        /* Read record */
        usbserial_clear_buffer();
        ret= usbserial_recv(&inrec.type, 1);
        ret= usbserial_recv(&inrec.len, 1);
        if (inrec.len == 0) {
           ret          = -1;
           goto end;
        }
        chk= inrec.len;
        if (inrec.len > 1) {
           ret          = usbserial_recv(inrec.data, inrec.len - 1);
           for (i = inrec.len - 2; i >= 0; i--) {
                    chk += inrec.data[i];
           }
        }
        ret = usbserial_recv(&inrec.sum, 1);

        /* Check sum */
        chk^= 0xffff;
        chk&= 0x00ff;
        if (chk != inrec.sum) {
           ret = -1;
           goto end;
        }
        ret = inrec.type;
end:
        return ret;
}
```

レコード・タイプと長さを受信する

指定の長さに達するまでループする。
また，受信データはチェック・サム用に計上する。

チェック・サム

3. 7　CDC（Communication Device Class）

ファームウエア・アップデート・プログラムで使用されているCDC処理について記載します。

USBのCDC仕様については，”Universal Serial Bus Class Definitions for Communications Devices” Revision1.2を参照してください。

ファームウエア・アップデート・プログラムで使用されているCDCは，Abstract Control Modelで，対応するクラス・リクエストは次のようになります。

備考　USB規格の策定と管理は，USB Implementers Forum（USB-IF）という団体が行っています。
“Universal Serial Bus Class Definitions for Communications Devices” Revision1.2の詳細については，USB-IFの公式ウェブサイト（www.usb.org）を参照してください。

表3ー10　対応するクラス・リクエスト

クラス・リクエスト	内　　容
SendEncapsulatedCommand	コミュニケーション・クラス・インタフェースの制御プロトコルのフォーマットでコマンドを発行するためのリクエスト
GetEncapsulatedRespons	コミュニケーション・クラス・インタフェースの制御プロトコルのフォーマットで応答を要求するためのリクエスト
SetLineCoding	シリアル通信の通信フォーマットを指定するためのリクエスト
GetLineCoding	デバイス側の現在の通信フォーマット設定を取得するためのリクエスト
SetControlLineState	RS-232/V.24形式の制御信号

3. 7. 1　ポーリングによるEP監視処理

EP監視処理は，割り込みベクタではなく，ポーリングでEP（エンドポイント）の割り込みフラグを監視することで，EP0（コントロール転送用エンドポイント），EP2（バルク・アウト用エンドポイント）のFIFOにデータがあるかどうかを監視します。

次に，受信時におけるEP監視処理を示します。

図3－39　データ受信時のEP監視

```
static INT32 usbserial_recv( UINT8 *data , INT32 len )
{
        INT32     num       = 0;

        while (num < len) {
                while (recv_len == 0) {
                        usbf78k0r_intusbf0();
                }
                data[num] = recv_buf[recv_idx];
                recv_idx++;
                if (recv_idx >= recv_buf_size) {
                        recv_idx -= recv_buf_size;
                }
                recv_len--;
                num++;
        }
        return num;
}
```

データを受信するまで INTUSBF0 の割り込みを待つ。(EP0 / EP2)

受信データをバッファからコピー

3. 7. 2　INTUSBF0割り込み監視処理

EP0はコントロール転送用のエンドポイントです。ここでは，ハードウエアでは対応しない標準リクエスト，クラス・リクエスト，ベンダ・リクエストを監視します。またEP2はバルク・アウト用エンドポイントです。次にEP0・EP2の監視処理を示します。

図3－40　INTUSBF0割り込み監視処理(1/2)

```
void usbf78k0r_intusbf0(void)
{
   UINT8  request;
   volatile UINT16  tmpl = 0;
   volatile UINT16  tmph = 0;
   USB_SETUP  setup_data;

   if(UF0IS0 & C_RSUSPD){
         UF0IC0 = C_RSUSPDC;
         if(UF0EPS1 & C_RSUM){
                  if (usbf78k0r_rsmsus_flg != F_RESUME){
                           return;
                  }
                  usbf78k0r_rsmsus_flg = F_SUSPEND;

                  usbf78k0r_stop_mode();        /* Stop mode execution */
                  return;
         }
   }

   if (UF0IS0 & C_BUSRST) { /* Bus Reset has occurred. */
         UF0IC0 = C_BUSRSTC;
         usbf78k0r_busrst_flg = 1;     /* BUS Reset flag Setting */
         usbf78k0r_buff_init();        /* initialize buffer     */
   }

   if (UF0IS0 & C_SETRQ) {
         UF0IC0 = C_SETRQC;
         if ((UF0SET & C_SETCON) && (UF0MODS & C_CONF)) {
                  /* SET_CONFIGURATION request was automatically processed. */
                  usbf78k0r_busrst_flg = 0;     /* BUS Reset flag clear */
         }
   }
```

Resume またはSuspend 状態の判定

バス・リセットのチェック

UF0E0ST にデータがあったら割り込み要因をクリアする

図3－41　INTUSBF0割り込み監視処理(2/2)

```
    if (UF0IS1 & C_CPUDEC) {
          UF0IC1 = (UINT8)~C_PROT; /* interrupt clear */

          /* building of setup data */
          setup_data.RequstType = UF0E0ST;
          setup_data.Request    = UF0E0ST;
          tmpl = UF0E0ST;
          tmph = UF0E0ST;
          setup_data.Value      = ((tmpl & 0x00ff) | ((tmph << 8) & 0xff00));
          tmpl = UF0E0ST;
          tmph = UF0E0ST;
          setup_data.Index      = ((tmpl & 0x00ff) | ((tmph << 8) & 0xff00));
          tmpl = UF0E0ST;
          tmph = UF0E0ST;
          setup_data.Length     = ((tmpl & 0x00ff) | ((tmph << 8) & 0xff00));

          if (setup_data.RequstType & C_CLASS_REQUEST) {
                    if (setup_data.Index != C_IF0_DSC_bInterfaceNumber) {
                              usbf78k0r_sendstallEP0(); /* error */
                    }
                    usbf78k0r_classreq(&setup_data); /* Class Request Decode */
          }
          else {
                    usbf78k0r_standardreq(&setup_data); /* Standard Request Decode */
          }
    }

    if (UF0IS3 & C_BKODT) {
          UF0IC3 = ~C_BKODT;            /* interrupt clear                    */
//        usbf78k0r_rdata_flg = F_RCVDATA; /* data receiving flag setting */
          usb_read();
    }
    return;
}
```

リクエスト・データを構成する

クラス・リクエストの処理初期化の際に登録した関数を実行

標準リクエストの処理

BKO3DT にデータの受信があったら割り込み要因をクリア

データを取得

以下にINTUSBF0監視処理で呼び出す関数の説明をします。

(1) 標準リクエスト

標準リクエストでは，ディスクプリタを取得します。

図3－42　標準リクエストの処理

```
void usbf78k0r_standardreq(USB_SETUP *req_data)
{
        if (req_data->Request == C_GETDESC) { /* GetDescriptor[String/Class] */
                usbf78k0r_getdesc(req_data);
        }
        else { /* error */
                usbf78k0r_sendstallEP0();
        }
        return;
}
```

リクエストがディスクプリタなら取得する。そうでない場合はSTALL送信処理を行う。

標準リクエスト処理で呼び出されているディスクリプタ送信処理について、以下に示します。

図3－43　ディスクリプタ送信処理

```
void usbf78k0r_getdesc(USB_SETUP *req_data)
{
      UINT8  value;
      UINT8  *tmp;
      INT32  len;

      if ((req_data->Value & 0xff00) == C_STRDESC) { /* String Descriptor */
          value = (UINT8)(req_data->Value & 0xff);
          if (value >= (sizeof(T_USB_strings)/sizeof(T_USB_strings[0]))) {
                    /* EP0 STALL */
                    usbf78k0r_sendstallEP0();
                    return;
          }
          len = (INT32)(T_USB_strings[value][0]) & 0x000000ff;
          tmp = &(T_USB_strings[value][0]);
      }
      else { /* error */
          usbf78k0r_sendstallEP0();
          return;
      }

      if (req_data->Length < len) {
          len = (INT32)req_data->Length & 0x0000ffff;
      }
      usbf78k0r_send_EP0(tmp, len);

      return;
}
```

文字列によるディスクリプタを設定

長さを取得

EP0 から送信

ディスクリプタ・データ（USB_string）は，次のように定義されています。DSTR，USTR**はロケール設定，および**Unicode**設定のためのマクロです。**

図3－44　ディスクリプタ・データの定義

```
/* 0 : Language Code*/
DSTR(LangString, 2, (0x09,0x04));
/* 1 : Manufacturer*/
USTR(ManString, 19, ('N','E','C',' ','E','l','e','c','t','r','o','n','i','c','s',' ','C','o','.'));
/* 2 : Product*/
USTR(ProductString, 10, ('U','S','B',' ','C','o','m','D','r','v'));
/* 3 : Serial Number*/
USTR(SerialString,  10, ('0','_','9','8','7','6','5','4','3','2'));

unsigned char *USB_strings[]={LangString,ManString,ProductString,SerialString};
```

(2) クラス・リクエスト

クラス・リクエストの一覧を示します。リクエストにより次に示す関数が実行されます。

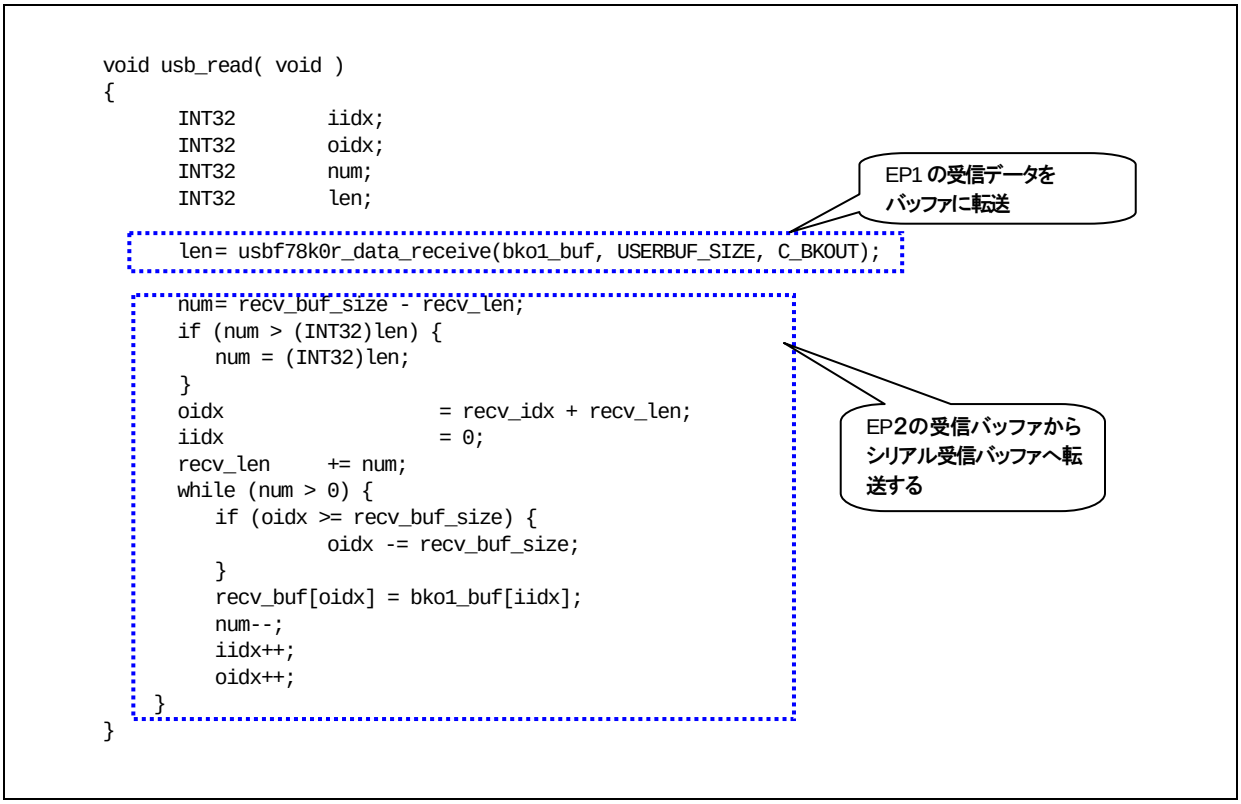
表3－11　クラス・リクエスト一覧

関数名	対応するリクエストと動作
usbf78k0r_send_encapsulated_command	SendEncapsulatedCommand ：EP0からデータを受信する
usbf78k0r_get_encapsulated_response	GetEncapsulatedRespons ：処理なし
usbf78k0r_set_line_coding	SetLineCoding ：EP0でUART通信用設定データを受信 ：EP0NULLパケット送信処理実行
usbf78k0r_get_line_coding	GetLineCoding ：UART通信用設定データをEP0から送信
usbf78k0r_set_control_line_state	SetControlLineState ：EP0NULLパケット送信処理実行
usbf78k0r_sendstallEP0	：STALL送信処理

(3) EP2受信処理

EP2の受信処理を示します。

図3ー45　EP2受信処理




```
void usb_read( void )
{
      INT32        iidx;
      INT32        oidx;
      INT32        num;
      INT32        len;

   len= usbf78k0r_data_receive(bko1_buf, USERBUF_SIZE, C_BKOUT);

   num= recv_buf_size - recv_len;
   if (num > (INT32)len) {
      num = (INT32)len;
   }
   oidx                  = recv_idx + recv_len;
   iidx                  = 0;
   recv_len     += num;
   while (num > 0) {
      if (oidx >= recv_buf_size) {
               oidx -= recv_buf_size;
      }
      recv_buf[oidx] = bko1_buf[iidx];
      num--;
      iidx++;
      oidx++;
  }
}
```

3. 7. 4　USBデータ送受信

次に，USBデータ送受信処理，NULLパケット送信処理，STALL送信処理を示します。

（1）USBデータ送信処理

USBデータ送信処理を以下に示します。

図3－46　データ送信処理（1/2）

```
INT32 usbf78k0r_data_send(UINT8* data, INT32 len, INT8 ep)
{
        INT8   data_end;
        INT8   mpkt_size;
        INT32  i;
        UINT32 addr;

        switch (ep) {
        case C_BKI1:
                addr     = UF0BI1_ADDRESS;
                data_end  = C_BKI1DED;
                mpkt_size = C_BKI1PK;
                break;
        case C_BKI2:
                addr     = UF0BI2_ADDRESS;
                data_end  = C_BKI2DED;
                mpkt_size = C_BKI2PK;
                break;
        case C_INT1:
                addr     = UF0INT1_ADDRESS;
                data_end  = C_IT1DED;
                mpkt_size = C_INT1PK;
                break;
        case C_INT2:
                addr     = UF0INT2_ADDRESS;
                data_end  = C_IT2DED;
                mpkt_size = C_INT2PK;
                break;
        default: /* error */
                return DEV_ERROR;
        }
```

EP1 用の書き込み用のレジスタ・アドレス、書き込み終了ビット，状態ビット，最大サイズをセット

EP3 用の書き込み用のレジスタ・アドレス，終了ビット，状態ビット，最大サイズをセット

EP7 用の書き込み用のレジスタ・アドレス，終了ビット，状態ビット，最大サイズをセット

EP8 用の書き込み用のレジスタ・アドレス，終了ビット，状態ビット，最大サイズをセット

図3－47　データ送信処理（2/2）

```
        if (len > mpkt_size) { /* below the Max Packet Size */
                len = mpkt_size;
        }
        for (i = 0; i < len; i++) {
                USBF78K0RREG_SET(addr, *data);
                data++;
        }
        if ((len < mpkt_size) || (mpkt_size < C_FIFOSIZE)) {
                UF0DEND |= data_end; /* Tx enable(short packet) */
        }
        return len;
}
```

最大サイズ内なら，指定のサイズまでを順次書き込みレジスタに書き，最後に終了ビットを立てる

(2) USBデータ受信処理

USBデータ受信処理を以下に示します。

図3－48　データ受信処理

```
INT32 usbf78k0r_data_receive(UINT8* data, INT32 len, INT8 ep)
{
        UINT32 addr;
        INT32  size;
        INT32  i;

        switch (ep) {
        case C_BKO1:
                addr = UF0BO1_ADDRESS;
                break;
        case C_BKO2:
                addr = UF0BO2_ADDRESS;
                break;
        default: /* error */
                return DEV_ERROR;
        }

        usbf78k0r_rdata_length(&size, ep); /* received data size */
        if (size <= len) {
                len = size;
                usbf78k0r_rdata_flg = F_NODATA; /* data receiving flag clear */
        }

        for (i = 0; i < len; i++) {
                *data = USBF78K0RREG_READ(addr);
                data++;
        }
        return len;
}
```

受信データが指定サイズと一致しているか確認を行う。

受信データをバッファに転送する

(3) EP0NULLパケット送信処理

EP0NULLパケット送信処理を以下に示します。

図3ー49　EP0NULLパケット送信

```
void usbf78k0r_sendnullEP0(void)
{
        if (UF0IS1 & C_PROT) { /* second request is executed */
                return;
        }

        UF0FIC0  = C_EP0WC; /* FIFO Clear              */
        UF0DEND |= C_E0DED; /* data send(Null Packet) */
        return;
}
```

FIFOをクリアし，送信処理を行う

(4) STALL送信処理

STALL送信処理を以下に示します。

図3ー50　STALL送信処理

```
void usbf78k0r_sendstallEP0(void)
{
        if (UF0IS1 & C_PROT) { /* second request is executed */
                return;
        }

        UF0SDS = C_SNDSTL; /* send STALL */
        return;
}
```

STALLハンドシェーク応答

第4章　ファイル転送アプリケーションの解説

この章では，PC上で動作するファイル転送アプリケーションについて記載します。

4. 1　開発環境

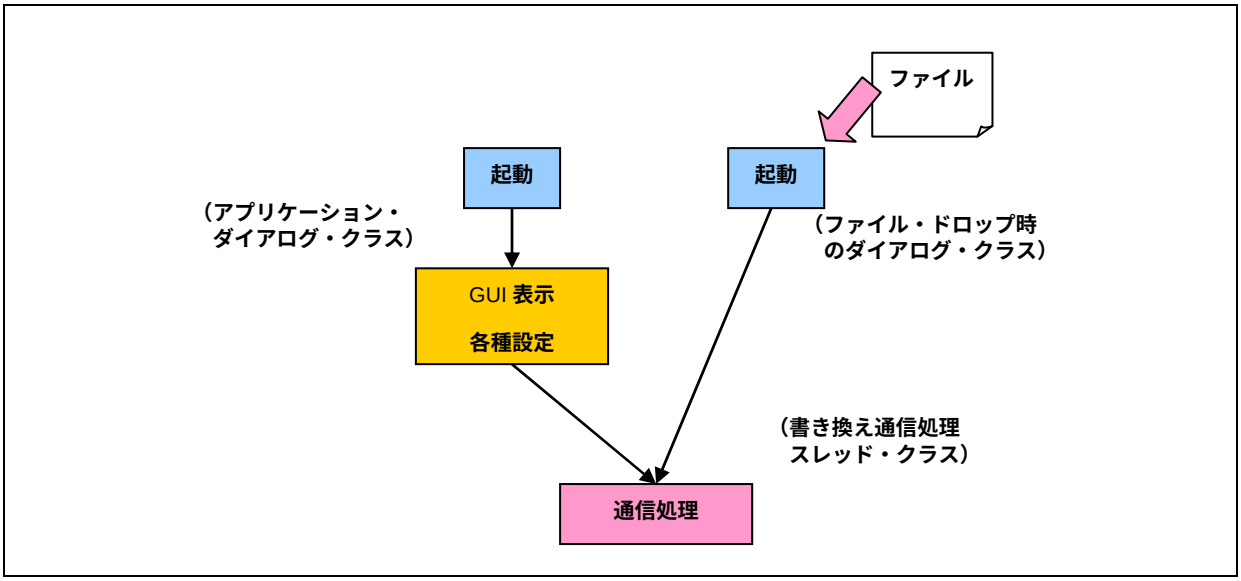
ファイル転送アプリケーションは，次に示す環境で構築されています。

OS　　：Windows XP
開発言語　：Microsoft Visual C++ 6.0（MFC）

4. 2　動作概要

ファイル転送アプリケーションは，起動時に引数として，書き換え対象のファイル名（およびオプション）を受け取ると，直接書き換え処理に移行します。ファイルの指定がない場合は，設定画面を表示します。

図4－1　ファイル転送アプリケーション動作概要



4. 3　ファイル構成

ファイル転送アプリケーションのファイル一覧を次に示します（主要なファイルのみを記載しています）。

表4－1　ファイル転送アプリケーションのファイル一覧

ファイル名	説　　明
FlashSelfRewriteGUI.dsw	ワークスペース・ファイル
FlashSelfRewriteGUI.dsp	プロジェクト・ファイル
FlashSelfRewriteGUI.clw	クラス・ウィザード用ファイル
FlashSelfRewriteGUI.rc	リソース・ファイル
FlashSelfRewriteGUI.cpp	アプリケーション・クラスの処理ファイル
FlashSelfRewriteGUI.h	アプリケーション・クラスの定義ファイル
FlashSelfRewriteGUIDlg.cpp	アプリケーションのダイアログ・クラスの処理ファイル
FlashSelfRewriteGUIDlg.h	アプリケーションのダイアログ・クラスの定義ファイル
FlashSelfRewriteGUIDrop.cpp	ファイル・ドロップ時のダイアログ・クラスの処理ファイル
FlashSelfRewriteGUIDrop.h	ファイル・ドロップ時のダイアログ・クラスの定義ファイル
CommandThread.cpp	書き換え通信処理スレッド・クラス処理ファイル
CommandThread.h	書き換え通信処理スレッド・クラス定義ファイル
CommonProc.cpp	共通処理クラス処理ファイル
CommonProc.h	共通処理クラス定義ファイル
SerialPort.cpp	シリアルCOMポート通信クラスの処理ファイル
SerialPort.h	シリアルCOMポート通信クラスの定義ファイル
Resource.h	リソース・ヘッダ・ファイル
UsbfUpdate.ini	アプリケーション動作設定ファイル

4. 3. 1　アプリケーション･クラス（FlashSelfRewriteGUI）

初回起動時に引数（オプション）をチェックし，ファイルが指定してあればファイル・ドロップ時のダイアログ・クラスを呼び，そうでなければ通常のダイアログ・クラスを呼び出します。

次にアプリケーションの起動オプションを示します。

表4－2　アプリケーション起動オプション

オプション	説　　明
/M [chip|address]	動作モードの指定 Chipかaddressを指定
/S nnnnnn	書き込み開始アドレスを16進数で指定
/C nn	接続COMポート番号の指定
filename	書き換え対象のファイル・パス

4. 3. 2　アプリケーション・ダイアログ・クラス（FlashSelfRewriteGUIDlg）

書き換え指定のダイアログ画面を表示します（第2章　USBファンクション・ファームウエア・アップデートの実行参照）。この画面で動作モード，書き換えアドレス，書き換えファイル，接続COMポートを指定します。また，画面表示の際に，アプリケーション動作設定ファイルを読み込み，先の指定がしてあればデフォルト値として画面に反映されます。

“Update” ボタンをクリックすると，書き換え通信処理スレッド・クラスが呼び出されます。

追加したメンバ変数を示します。

表4－3　アプリケーション・ダイアログ・クラスのメンバ変数の一覧

メンバ変数		説　明
型	メンバ名	
int	m_nCOM	接続するCOMポート番号
TCHAR	m_tcAppDir[_MAX_PATH]	アプリケーションの実行ディレクトリ
int	m_nCurTargetID	現在のターゲットのID
CString	m_strCurTarget	現在のターゲット名
CString	m_strCurDevice	現在のデバイス
CStringArray	m_arDeviceVal	デバイスのリスト
CStringArray	m_arDeviceText	デバイス名のリスト
int	m_nDevSize	現在のデバイスのROMサイズ
CWinThrread*	m_pCommandThread	スレッド・クラスのポインタ
BOOL	m_bExistThread	スレッドの動作状態
BOOL	m_bStartUp	初回起動を表す
CArray<int,int>	m_arBlockStart	先頭ブロック番号の配列
CArray<int,int>	m_arBlockEnd	末尾ブロック番号の配列
CArray<int,int>	m_arBlockUnit	1ブロックごとのバイト数の配列
COleDateTime	m_dtStart	書き換え処理開始日付
COleDateTime	m_dtEnd	書き換え処理終了日付

メンバ関数を次に示します。

表4－4　Read_DeviceInfo関数

関数名		Read_DeviceInfo
記述形式		BOOL Read_DeviceInfo (VOID)
機能		アプリケーション動作設定ファイルから情報を取得
入出力	入力	なし
	出力	TRUE(成功)／FALSE(失敗)

表4－5　Write_DeviceInfo関数

関数名		Write_DeviceInfo
記述形式		BOOL Write_DeviceInfo (VOID)
機能		アプリケーション動作設定ファイルを更新する
入出力	入力	なし
	出力	TRUE(成功)／FALSE(失敗)

表4ー6　Update_Message関数

関数名		Update_Message
記述形式		VOID Update_Message (LPCTSTR)
機能		**メッセージ表示欄にメッセージを表示する**
入出力	**入力**	**メッセージ文字列のポインタ**
	出力	**なし**

表4ー7　Get_BlockAddress関数

関数名		Get_BlockAddress
記述形式		DWORD Get_BlockAddress(int nBlk, EnBlockAddress opt)
機能		**指定したブロック番号のメモリ上のアドレスを返す**
入出力	**入力**	nBlk：**ブロック番号** Opt：START／END(**ブロックの先頭か末尾を現す**)
	出力	**メモリ上のアドレス**

表4ー8　Get_AddressBlock関数

関数名		Get_AddressBlock
記述形式		int Get_AddressBlock(DWORD dwAddress)
機能		**指定したアドレスがあるブロック番号を返す**
入出力	**入力**	dwAddress：**メモリ上のアドレス**
	出力	**ブロック番号**

表4ー9　Initialize_Device関数

関数名		Initialize_Device
記述形式		VOID Initialize_Device(VOID)
機能		**初期化処理**
入出力	**入力**	**なし**
	出力	**なし**

表4ー10　AppStatus関数

関数名		AppStatus
記述形式		VOID AppStatus(BOOL stu)
機能		**書き換え動作時の状態を設定する**
入出力	**入力**	stu：TRUE(**画面操作を有効にする**) FALSE(**画面操作を無効にする**)
	出力	**なし**

4. 3. 3　ファイル・ドロップ時のダイアログ・クラス（FlashSelfRewriteGUIDrop）

表示後，すぐに書き換え通信処理スレッド・クラスを呼び出し，書き換え処理を開始します。画面はプログレス・バーのみです。

追加したメンバ変数を示します（アプリケーション・ダイアログ・クラスと同じ部分は，省略しています）。

表4－11　ファイル・ドロップ時のダイアログ・クラスのメンバ変数の一覧

メンバ変数		説　明
型	メンバ名	
CString	m_strFileName	対象とするファイル・パス
EnMode	m_enMode	書き換えモード
DWORD	m_dwStartAddress	書き換え開始アドレス

追加したメンバ関数を次に示します。

表4－12　Execute関数

関数名		Execute
記述形式		VOID Execute(VOID)
機能		書き換え処理を実行します
入出力	入力	なし
	出力	なし

4. 3. 4　書き換え通信処理スレッド・クラス（CommandThread）

シリアルCOMポート通信クラスを使用して，ターゲットとなる評価ボードに接続し，指定のファイルをインタフェース仕様に沿って送受信を行います。ファイルがHEXファイルの場合は，その解析も行います。

追加したメンバ変数を示します（アプリケーション・ダイアログ・クラスと同じ部分は，省略しています）。

表4－13　書き換え通信処理スレッド・クラスのメンバ変数一覧

メンバ変数		説　明
型	メンバ名	
CDialog*	m_pAppDlg	呼び出し元のダイアログ・クラスのポインタ
CString	m_strAppDir	アプリケーションのあるディレクトリ
BOOL*	m_pbExistThread	スレッドの動作状況のポインタ
CSerialPort	m_Serial	シリアルCOMポート通信クラス
int	m_nCOM	接続するCOMポート番号
CString	m_strFileName	対象とするファイル・パス
EnMode	m_enMode	書き換えモード
DWORD	m_dwStartAddress	書き換え開始アドレス
DWORD	m_dwROMStartAddress	ROMの先頭アドレス
DWORD	m_dwROMEndAddress	ROMの末尾アドレス

追加したメンバ関数を次に示します。

表4ー14　Cal_CheckSum関数

関数名		Cal_CheckSum
記述形式		BYTE Cal_CheckSum(LPBYTE bytes, LONG size)
機能		チェック・サムを算出します
入出力	入力	bytes：データ列のポインタ size：データ列の長さ
	出力	チェック・サム算出値

表4ー15　Chage_strHex2Bibary関数

関数名		Change_strHex2Binary
記述形式		VOID Change_strHex2Binary(LPCSTR strHex, LPBYTE pbytes, LONG size)
機能		16進数であらわされた文字列をバイナリのデータ列に変換します
入出力	入力	strHex：16進数で表された文字列のポインタ pbyte：データ列の先頭ポインタ size：変換するデータの数
	出力	なし

表4ー16　Upsets_DWORD関数

関数名		Upsets_DWORD
記述形式		DWORD Upsets_DWORD(DWORD dwVal)
機能		DWORD型の値をバイトごとに反転する （ex.）　0xaabbccdd → 0xddccbbaa
入出力	入力	dwVal：反転するDWORDの値
	出力	反転された値

表4ー17　SET_StartRecord関数

関数名		SET_StartRecord
記述形式		VOID SET_StartRecord (LPVOID lpRecord)
機能		書き換え開始レコードを作製する
入出力	入力	lpRecord：レコード格納ポインタ
	出力	なし

表4ー18　SET_EndRecord関数

関数名		SET_EndRecord
記述形式		VOID SET_EndRecord (LPVOID lpRecord)
機能		書き換え終了レコードを作製する
入出力	入力	lpRecord：レコード格納ポインタ
	出力	なし

4. 3. 5　共通処理クラス（CommonProc）

共通で使用される処理を定義しています。

追加したメンバ関数を次に示します。

表4ー19　GetAppDir関数

関数名		GetAppDir
記述形式		static VOID GetAppDir(LPTSTR path, int sw = 0)
機能		**アプリケーションの実行アドレスを取得します。**
入出力	**入力**	path：**取得する文字列のポインタ** sw：0　**パスをそのまま取得** 1　**ショート・パスに変更したパスを取得**
	出力	**なし**

表4ー20　Change_Hex2Val関数

関数名		Change_Hex2Val
記述形式		static DWORD Change_Hex2Val(LPCSTR pHex)
機能		**1バイト（16進数2桁）で表された文字列を数値に変換する**
入出力	**入力**	pHex：**16進数2桁で表された文字列のポインタ**
	出力	**変換された値**

表4ー21　IsNumeric関数

関数名		IsNumeric
記述形式		static BOOL IsNumeric(LPCTSTR lpNum, LONG size, int type = 10)
機能		**数値チェック処理**
入出力	**入力**	lpNum：**数値を表した文字列のポインタ** size：**チェックする数値の桁数** type：10　10**進数としてチェックする** 16　16**進数としてチェックする**
	出力	TRUE(**数値であることを表す**)／FALSE(**数値ではないことを表す**)

表4ー22　IsExistFile関数

関数名		IsExistFile
記述形式		static BOOL IsExistFile(LPCTSTR lpszFileName, BOOL bDirectory = FALSE)
機能		**ファイルの存在チェック**
入出力	**入力**	lpszFileName：**確認するファイル・パス** bDirectory：FALSE(**ファイルをチェックする**) TRUE(**ディレクトリをチェックする**)
	出力	TRUE(**存在する**)／FALSE(**存在しない**)

4. 3. 6　シリアルCOMポート通信クラス（SerialPort）

このクラスを使用して，COMポートでのシリアル通信を行います。通信設定は固定で次のようになっています。

表4－23　シリアル通信設定値

設定項目	値
ボー・レート	115200 bps
データ・サイズ	8ビット
パリティ	なし
ストップ・ビット	1ビット
スタート・ビット	LSB
フロー制御	なし

追加したメンバ変数を次に示します。

表4－24　シリアルCOMポート通信クラスのメンバ変数一覧

メンバ変数		説　明
型	メンバ名	
HANDLE	m_hCom	接続時に取得するハンドル
DCB	m_Dcb	デバイス制御ブロック構造体
COMMTIMEOUTS	m_TimeoutSts	タイムアウト設定用の構造体
INT	m_nCOM	接続するポート番号

メンバ関数を次に示します。

表4－25　Port_Open関数

関数名		Port_Open
記述形式		LONG Port_Open(INT com)
機能		指定のCOMポートに接続します
入出力	入力	com：COMポート番号
	出力	0　接続成功 －1　接続失敗

表4－26　Port_Close関数

関数名		Port_Close
記述形式		VOID Port_Close(VOID)
機能		接続中のポートを切断します。
入出力	入力	なし
	出力	なし

表4ー27　Port_Write関数

関数名		Port_Write
記述形式		LONG Port_Write(LPCVOID buf, LONG cnt)
機能		シリアル通信によるデータの送信を行う。
入出力	入力	buf：送信データの列のポインタ cnt：送信データの長さ（バイト）
	出力	送信したバイト数。－1で送信の失敗。

表4ー28　Port_Read関数

関数名		Port_Read
記述形式		LONG Port_Read(LPVOID buf, LONG cnt)
機能		シリアル通信によるデータの受信を行う。
入出力	入力	buf：受信データを格納するデータ列のポインタ cnt：受信するデータの長さ（バイト）
	出力	受信したバイト数。－1で受信の失敗を表す。

表4ー29　Get_PortNumber関数

関数名		Get_PortNumber
記述形式		INT Get_PortNumber(VOID)
機能		接続中のポート番号を取得
入出力	入力	なし
	出力	接続中のポート番号

表4ー30　AutoScanCom関数

関数名		AutoScanCom
記述形式		INT AutoScanCom (LPCTSTR pszService, LPCTSTR pszInterface, INT nNo = 0)
機能		接続可能なCOMポートを検出
入出力	入力	pszService：COMポートが動作しているサービス名 pszInterface：インタフェース名 nNo：この番号以降を検索する
	出力	検出したCOMポート番号。見つからなかったら0が返る。

4. 3. 7　アプリケーション動作設定ファイル（UsbfUpdate.ini）

アプリケーション動作設定ファイルはiniファイル形式で，設定値の保持，またはデバイスの情報を保持します。このファイルはexeファイルと同一のフォルダに置きます。

次にiniファイル内の定義を示します。

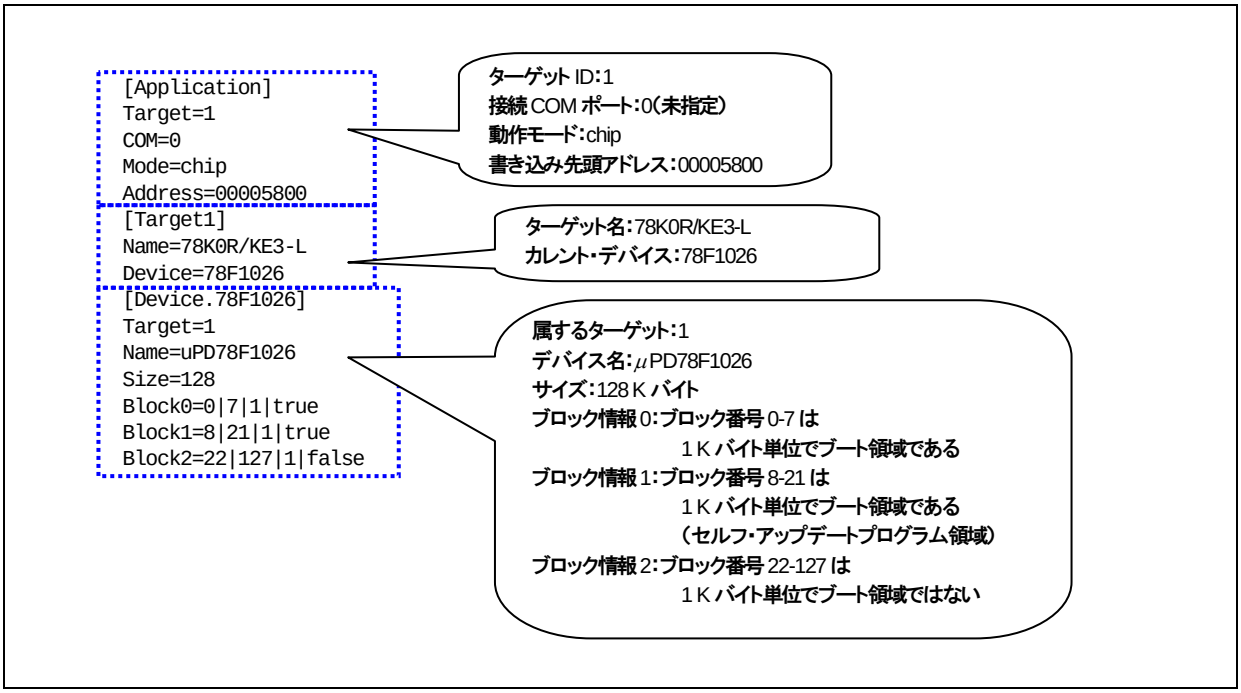
表4－31　アプリケーション動作設定ファイル説明（セクション）

セクション	説　明
Application	アプリケーションで設定中の値を表す
Tartget1	ターゲットIDを指す
Device.78F1026	デバイスの情報を表す 複数設定が可能

表4－32　アプリケーション動作設定ファイル内項目一覧

セクション	キー	値	説　明
Application	Target	1～	指定中のターゲットのID番号
	COM	1～20	接続中または接続するCOMポート番号
	Mode	chip/address	指定中の動作モードを表す chip：ブート・スワップを使用したユーザ・プログラムの書き換え address：指定アドレスからの書き換え
	Address	FFFFFFFF	書き込む先頭のアドレス（16進数）
Tartget1	Name	XXX	このターゲットの名前を表す
	Device	XXX	このターゲットで指定しているデバイス
Device.70F3769	Target	1～	このデバイスが属するターゲットID
	Name	XXX	このデバイスの名前
	Size	999	このデバイスのROMサイズ
	Block0	XXX|XXX|XXX|XXX	ブロックの情報。"|"で区切る 先頭ブロック番号｜最終ブロック番号｜1ブロックのサイズ（Kバイト）｜ブート領域かどうか Block1～とすることで複数指定可能

図4－2　アプリケーション動作設定iniファイル



4. 4　動作モード

動作モードの内容を次に示します。

（1）Chip

指定のファイルは，モトローラSフォーマット，およびインテル拡張フォーマットのHEXファイルでなければなりません。解析処理でそれ以外のフォーマットであればエラーとなります。書き込む先はメモリの先頭からなので，アドレスの指定は無視されます。

（2）Address

ファイルのイメージをそのまま転送し，指定のアドレスから書き込みます。

4.5　メッセージ表示

メッセージ表示欄に表示されるメッセージとその表示タイミングの一覧を次に示します。

表4－33　表示メッセージ一覧

	メッセージ	表示タイミング
1	書き換え処理を開始します	書き換え処理開始時
2	正常終了しました	書き換え処理正常終了時
3	ファイルを指定してください	書き換え処理時に指定のファイルが指定されていない。 または，指定ファイルが存在しない
4	モードを指定してください	書き換え処理時にモードが設定されていない
5	アドレスを正しく指定してください	Addressモードで書き換え処理時に，アドレスが正常に指定されていない。
6	COMポートを指定してください	COMポートが正しく指定されていない
7	ERR：ファイルオープンエラー	ファイルのオープンに失敗した
8	ERR：ファイルフォーマットエラー	Mode=Chipのときに，モトローラSフォーマット，インテル拡張フォーマット以外のファイルを指定した
9	ERR：COMポートnに接続できません。	COMポートnの接続に失敗した
10	ERR：データ送信エラー	データの送信に失敗した
11	ERR：データ受信エラー	データの受信に失敗した（リトライ3回も同様）
12	ERR：書き換え処理停止	評価ボード側からNAK（エラー）を受信した
13	ERR：ファイルサイズエラー	ファイル・サイズ・チェックの際にデータのサイズがROM領域をはみ出てしまう

第5章　ユーザ・プログラムの作成

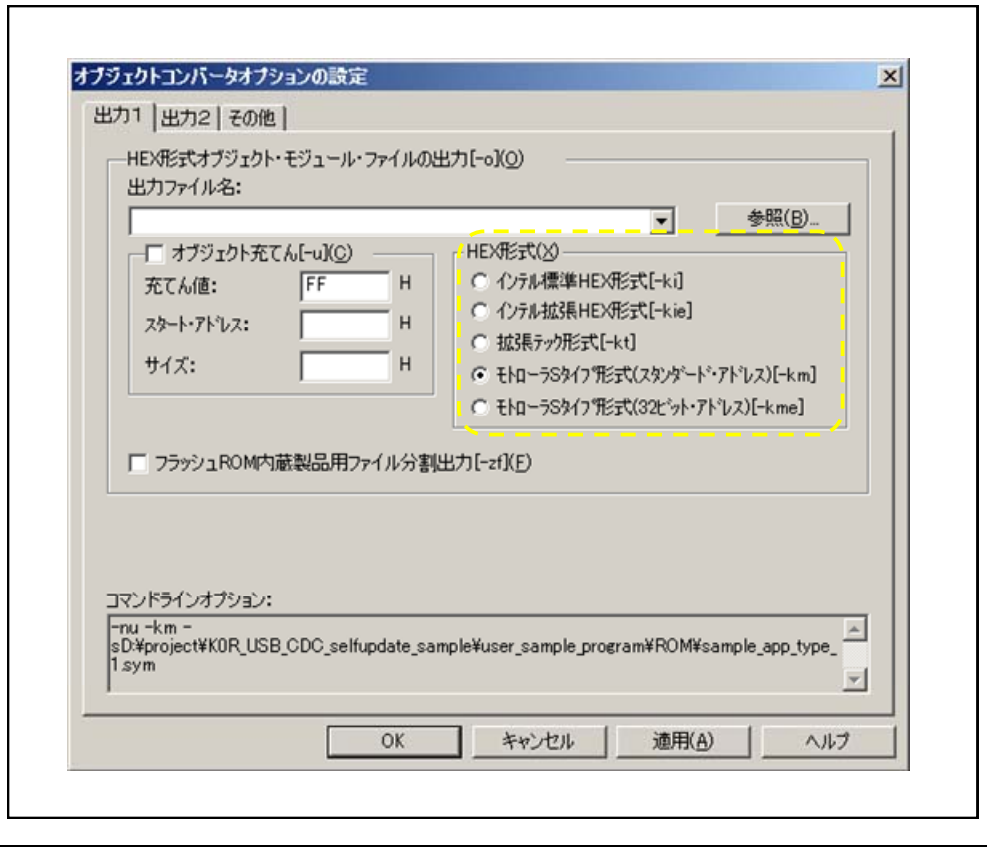
この章では，ユーザ・プログラムを作成するうえで留意すべき事項について説明します。

5. 1　PM+設定（HEXファイルのフォーマット設定）

USBファームウエア・アップデート・プログラム（ファイル転送アプリケーション）は，ユーザ・プログラムを書き換えるとき，モトローラ・タイプS（32ビット），モトローラ・タイプS（スタンダード），インテル拡張フォーマットのHEXファイルしか扱いません。＜オブジェクトコンバータオプションの設定＞ダイアログの《出力1》タブからフォーマットを指定してください。

例　《HEX形式（X）》に“モトローラタイプS（スタンダード・アドレス）[-km]”を選択します。

図5－1　HEXファイル・フォーマット指定例



5. 2　ブート処理（リセット・ベクタ・セクション）

セルフ・アップデート・プログラムは，リセット時のベクタ処理をメモリの先頭（00000000番地から）を前提に動作するため，作成するユーザ・プログラムも，リセットのセクションはメモリの先頭にしてください。

5. 3　リンク・ディレクティブ（ユーザ・プログラムの配置制限）

ユーザ・プログラムは、ファームウェア・アップデートプログラムの位置には配置できません。そのため，リンク・ディレクティブ・ファイルに記述するセグメントには，セルフ・アップデート・プログラムに配置されないようにアドレスを指定する必要があります（3. 2　**メモリ・マップ**参照）。

各製品でのファームウェア・アップデート・プログラムの位置は次のとおりです。

表5－1　ファームウエア・アップデート・プログラム配置アドレス

対象CPU	配置アドレス
78K0R/KC3-L, 78K0R/KE3-L	00 2000H

5. 4　セグメントの再定義

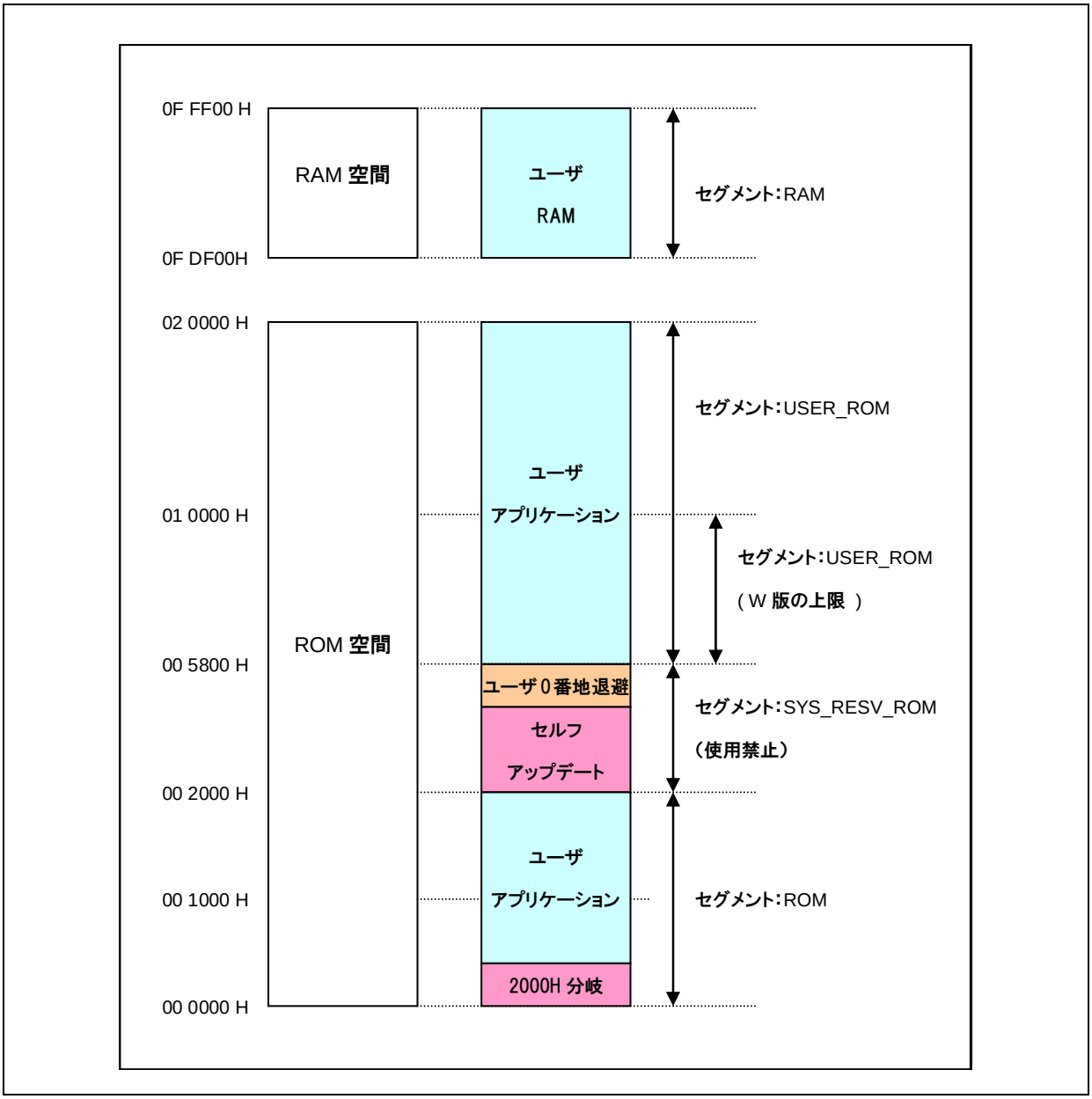
ユーザ・サンプル・プログラムのリンク・ディレクティブ・ファイルでは，内蔵フラッシュ・メモリ空間を次の通り再定義しています。

図5－2　セグメントの再定義

```
      MEMORY  ROM           : ( 000000H , 002000H )
      MEMORY  SYS_RESV_ROM  : ( 002000H , 003800H )    ; 使用禁止領域
;     MEMORY  USER_ROM      : ( 005800H , 01A800H )    ; V 版ｺﾝﾊﾟｲﾗ
      MEMORY  USER_ROM      : ( 005800H , 00A800H )    ; W 版ｺﾝﾊﾟｲﾗ
      MEMORY  RAM           : ( 0FDF00H , 002000H )
```

備考：　　無償版（W版 ）コンパイラ / アセンブラを使用している場合，64K空間にのみマッピング可能です。

図5－3　ユーザ・サンプル・プログラムのマッピング



5.4　セグメントの配置

ユーザ・サンプル・プログラムのリンク・ディレクティブ・ファイルでは，ROM 空間をメモリ･ディレクティブにより再定義した上で，ユーザ・サンプル・コードの配置セグメントを次の通り定義しています。

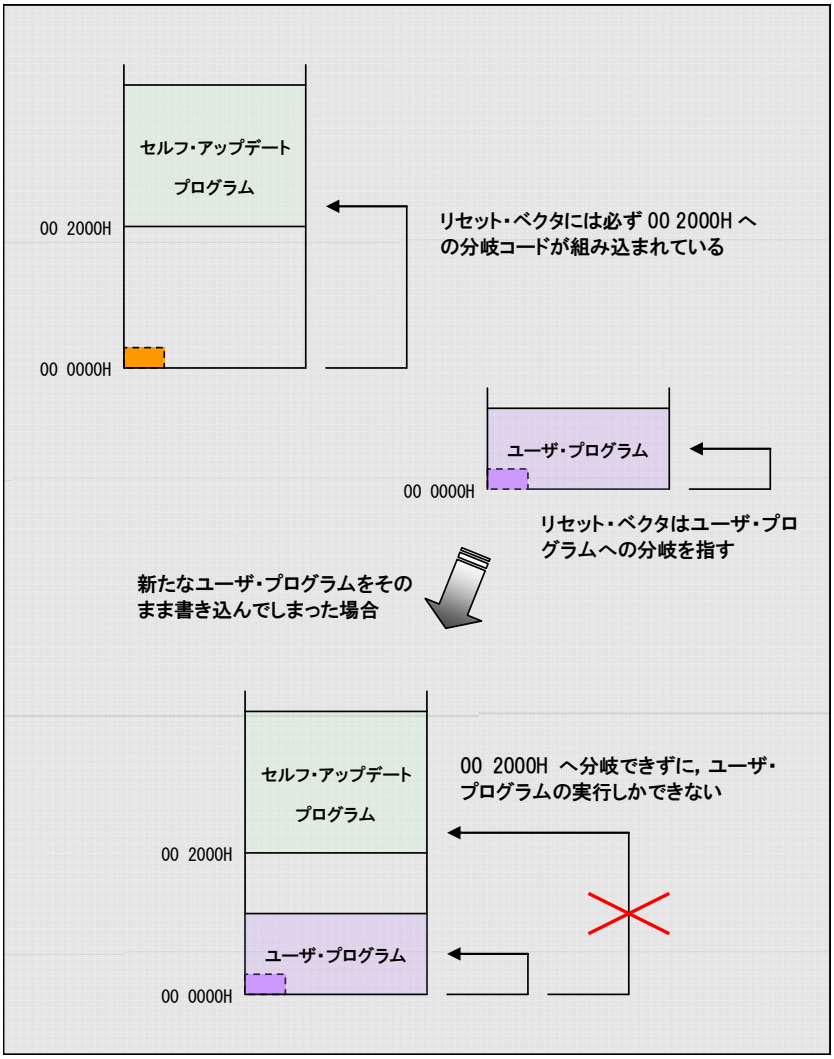
図5ー4　セグメント配置

```
MERGE   SMP_CODE      := ROM
MERGE   SMP_CNST      := ROM
MERGE   SMP_DATA      := RAM

MERGE   @@LCODE       := ROM
MERGE   @@LCODEL      := ROM
```

5. 5　ユーザ・プログラムの注意点

ユーザ・プログラムを書き込む際に，新たなリセット・ベクタの情報（ ユーザ・プログラムにおける，スタートアップ・ルーチンの配置アドレス ）を書き込んだ場合，下図の通り，以後のセルフ・アップデート・プログラムは起動できない事になります。

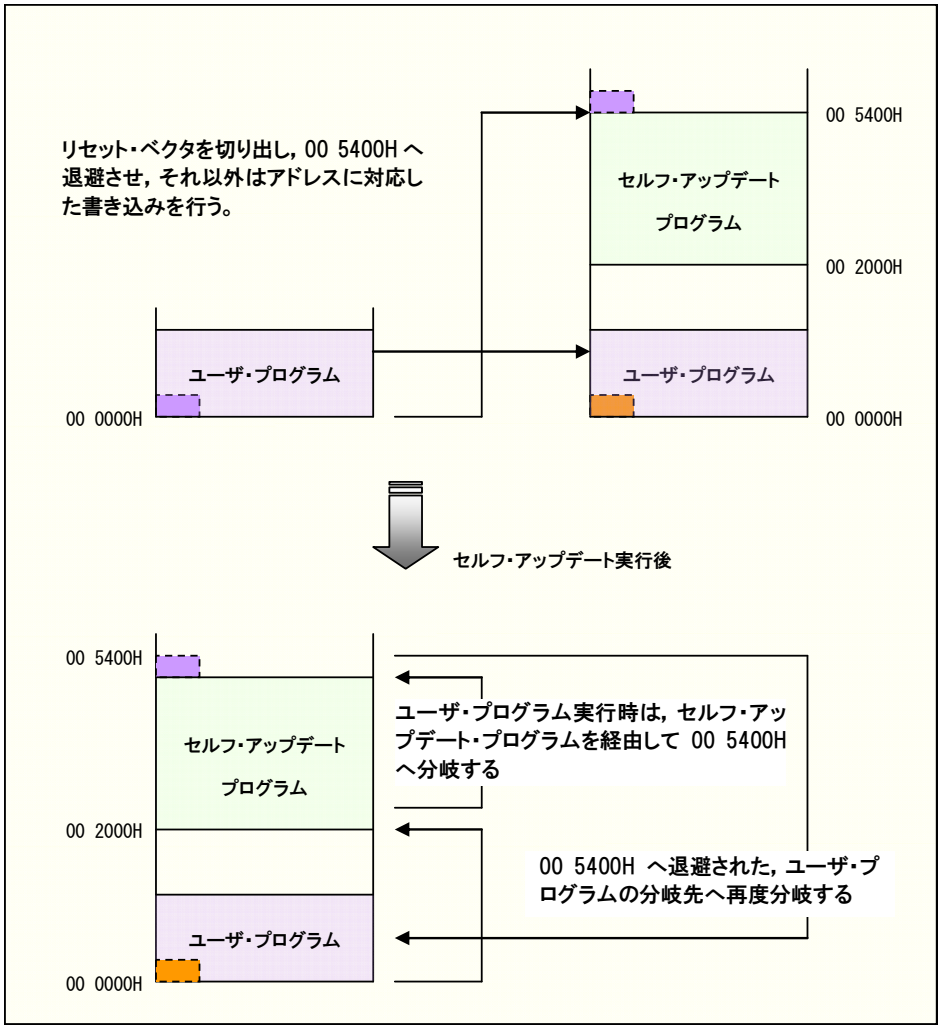
図5－5　リセット・ベクタの上書き



セルフ・アップデート・プログラムでは，リセット・ベクタの情報を固定（ 常に 00 2000Hを指す ）とし，ユーザ・プログラムにおけるリセット・ベクタ情報はフラッシュ・メモリの専用領域に退避させています。ユーザ・プログラムの実行時に退避させた情報を読み出す事により，セルフ・アップデートの実行後であってもユーザ・プログラムや，セルフ・アップデート・プログラムの再実行が可能になります。

下図は，退避処理について記したものです。

図5－6　リセット・ベクタの退避



78K0R におけるリセット・ベクタを含むベクタ・テーブル領域は， 00 0000H - 00 007FHの 128 バイトの領域として予約されています。ベクタ・テーブル領域には，リセットや各割り込み要求発生により分岐する時のプログラム・スタート・アドレスを格納しておきます。

表5－2　ベクタ・テーブル（1/2）

名称	ベクタ･アドレス
RESET / POC / LVI / WDT / TRAP	0000H
INTDBG	0002H
INTWDTI	0004H
INTLVI	0006H
INTP0	0008H
INTP1	000AH
INTP2	000CH
INTP4	0010H
INTP5	0012H
INTST3	0014H
INTSR3	0016H
INTSRE3	0018H
INTDMA0	001AH
INTDMA1	001CH
INTST0/INTCSI00	001EH
INTSR0/INTCSI01	0020H
INTSRE0	0022H
INTST1 / INTCSI10 / INTIIC10	0024H
INTSR1	0026H
INTSRE1	0028H
INTIICA	002AH
INTTM00	002CH
INTTM01	002EH
INTTM02	0030H
INTTM03	0032H
INTAD	0034H
INTRTC	0036H
INTRTCI	0038H
INTKR	003AH
INTST2 / INTCSI20 / INTIIC20	003CH
INTP6	003EH

表5－3　ベクタ・テーブル（2/2）

名称	ベクタ･アドレス
INTTM04	0042H
INTTM05	0044H
INTTM06	0046H
INTTM07	0048H
INTSR2	004AH
INTP8	004EH
INTP9	0050H
INTP10	0052H
INTP11	0054H
INTSRE2	005CH
INTUSB	005EH
INTRSUM	0060H
BRK	007EH

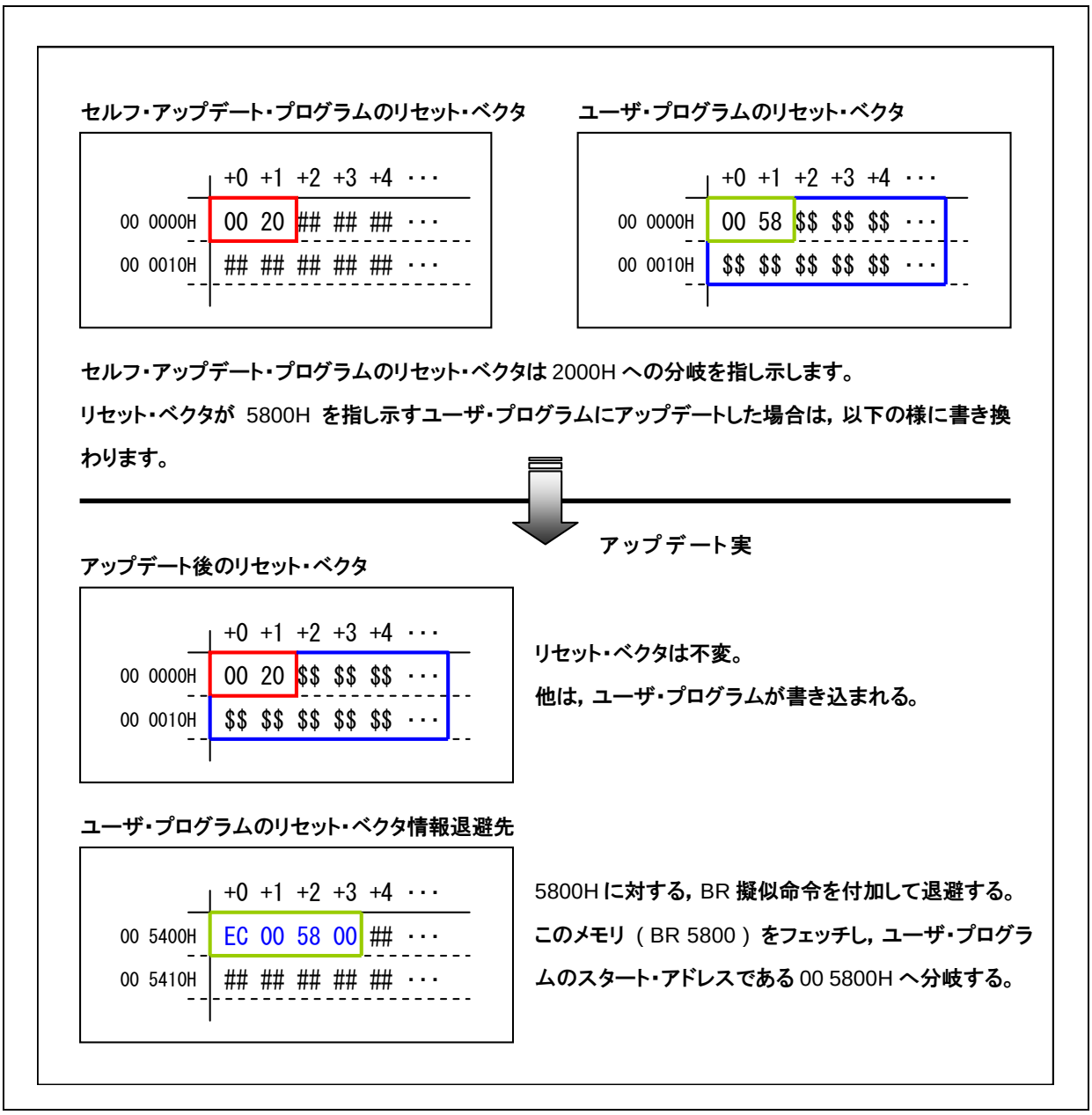
上表で示される通り，ベクタ・コードは2バイトであるので，割り込みの飛び先アドレスは 00 0000H - 00FFFFH の 64 K アドレスとなります。従って，セルフ・アップデート・プログラムは，通常の分岐では64K アドレス空間内にのみ配置可能です。

また，ベクタ・コードは2バイトであるので，分岐先のアドレスのみを配置します。(16ビット・アドレスのうち下位 8ビットが偶数アドレスに，上位 8ビットが奇数アドレスに格納)

その為，セルフ・アップデートにおけるユーザ・プログラムのリセット・ベクタ情報の退避とは，リセット・ベクタが指し示す分岐先のアドレスに対し，分岐命令を付加した上での退避を行う事，を指します。

下図は，ユーザ・プログラムにおいてリセット・ベクタの情報が 5800H を指し示す場合について，アップデート実行前と実行後のそれぞれのメモリ内容を示したものです。

図5－7　リセット・ベクタの退避



第６章　データ通信仕様

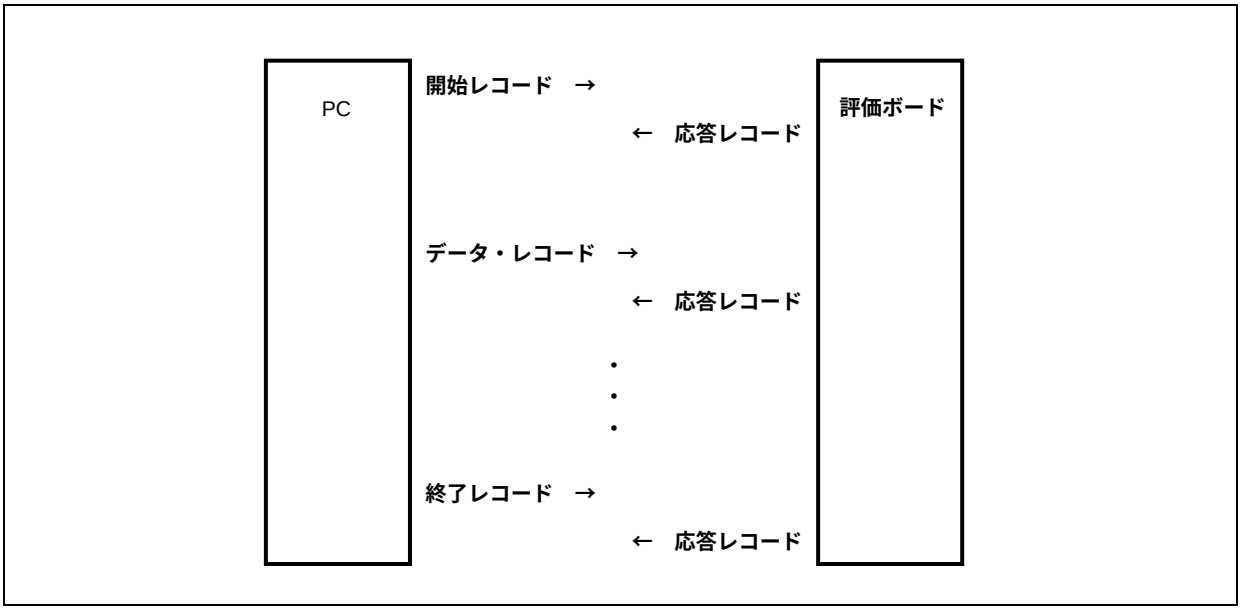
6. 1　書き換え通信インタフェース仕様

ファームウエア・アップデート・プログラムのPCと評価ボード間で通信する内容を示します。

6. 1. 1　通信データの構成

PCは最初に開始レコード，最後に終了レコードを送信します。フラッシュ・メモリに書き込むデータは，データ・レコードの形式で送信します。

図6－1　通信データ・シーケンス



6.1.2　PC側送信データ

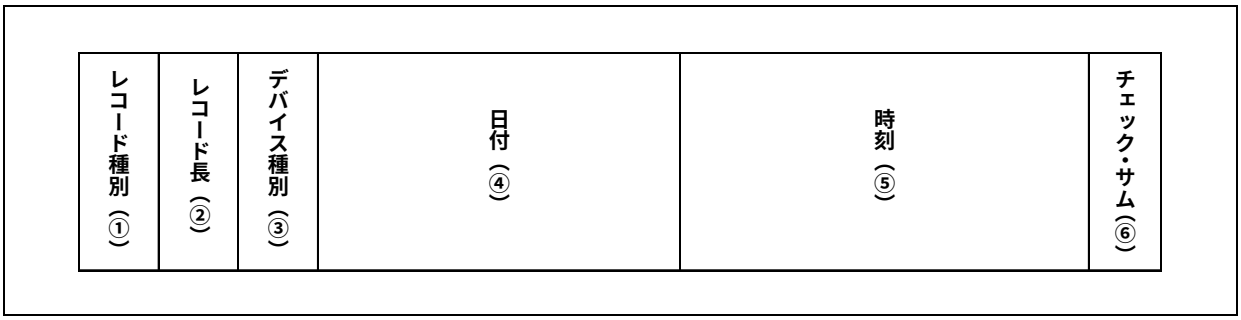
PC側は，開始レコード，データ・レコード，終了レコードを送信します。

各レコードは1レコードずつ送信し，応答レコードを受信するまで，次のレコードの送信は行いません。

(1) 開始レコード

書き換えの実行時に，最初に送信するレコードです。

図6－2　開始レコードの形式



①レコード種別

レコードの種類

1バイト

開始レコードのレコード種別は，0x00

②レコード長

デバイス種別以降のバイト数

1バイト

③デバイス種別

デバイスの種類

1バイト

④日付

現在の日付

年，月，日が各1バイト

年は西暦年の下2桁

⑤時刻

現在の時刻

時，分，秒が各1バイト

⑥チェック・サム

レコードのチェック・サム

1バイト

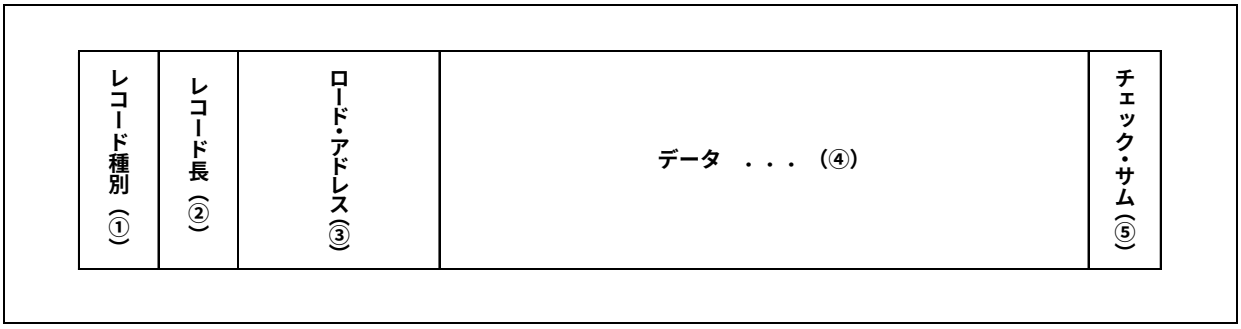
レコード長とデバイス種別と日付と時刻のチェック・サム

各バイトの値を加算した合計値の1の補数の下位8ビット

(2) データ・レコード

書き込むデータのレコードです。

図6－3　データ・レコードの形式



①レコード種別

レコードの種類

1バイト

データ・レコードのレコード種別は，0x0f

②レコード長

ロード・アドレス以降のバイト数

1バイト

③ロード・アドレス

フラッシュ・メモリのアドレス

4バイト

このアドレスからデータが書き込まれる

ロード・アドレスは，32ビット数値でリトル・エンディアンの形式

④データ

フラッシュ・メモリに書き込むデータ

1レコードあたり最大で256バイト

⑤チェック・サム

レコードのチェック・サム

1バイト

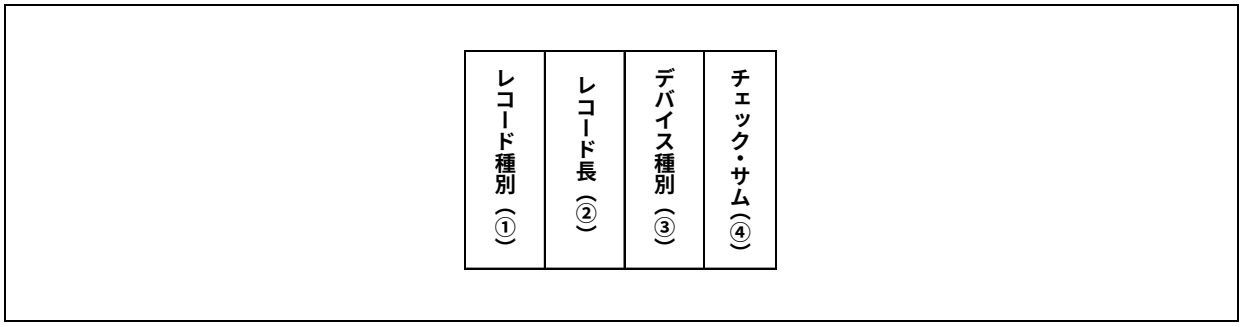
レコード長とロード・アドレスとデータのチェック・サム

各バイトの値を加算した合計値の1の補数の下位8ビット

(3) 終了レコード

すべてのデータを送信後，最後に送信するレコードです。

図6－4　終了レコードの形式



①レコード種別

レコードの種類

1バイト

終了レコードのレコード種別は，0xf0

②レコード長

デバイス種別以降のバイト数

1バイト

③デバイス種別

デバイスの種類

1バイト

④チェック・サム

レコードのチェック・サム

1バイト

レコード長とデバイス種別のチェック・サム

各バイトの値を加算した合計値の1の補数の下位8ビット

6.1.3　評価ボード側送信データ

評価ボードは，PCからのレコードに対して，応答レコードを送信します。

（1）応答レコード

図6－5　応答レコードの形式

①レコード種別

レコードの種類

1**バイト**

応答を返す対象のレコードのレコード種別

②レコード長

応答種別以降のバイト数

1**バイト**

③応答種別

応答種別

1**バイト**

以下の3**種類**

0x00 **：**ACK

0x0f **：**NAK**（再送要求）**

0xf0 **：**NAK**（エラー終了）**

④フィールド

エラーの場合は，エラー・コード1**バイト**

エラーでない場合は，レコード種別によって内容が異なる

開始レコード　　：デバイス種別

データ・レコード　：ロード・アドレス

終了レコード　　：デバイス種別

⑤チェック・サム

レコードのチェック・サム

1**バイト**

レコード長と応答種別とフィールドのチェック・サム

各バイトの値を加算した合計値の1**の補数の下位**8**ビット**

〔メ　モ〕

【発行】NEC**エレクトロニクス株式会社（**http://www.necel.co.jp/**）**
【問い合わせ先】http://www.necel.com/contact/ja/