お客様各位

カタログ等資料中の旧社名の扱いについて

2010年4月1日を以ってNECエレクトロニクス株式会社及び株式会社ルネサステクノロジが合併し、両社の全ての事業が当社に承継されております。従いまして、本資料中には旧社名での表記が残っておりますが、当社の資料として有効ですので、ご理解の程宜しくお願い申し上げます。

ルネサスエレクトロニクス ホームページ（http://www.renesas.com）

2010年4月1日
ルネサスエレクトロニクス株式会社

【発行】ルネサスエレクトロニクス株式会社（http://www.renesas.com）

【問い合わせ先】http://japan.renesas.com/inquiry

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報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、当社は、一切その責任を負いません。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。
3. 当社製品を改造、改変、複製等しないでください。
4. 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。
5. 輸出に際しては、「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。本資料に記載されている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の目的で使用しないでください。また、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に使用することができません。
6. 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。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「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。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されてい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生じた損害等に関し、当社は、一切その責任を負いません。なお、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。

   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

   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、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   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（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社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。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、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、社会的損害など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、機器またはシステムとしての出荷保証をお願いいたします。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、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、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。
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固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

注 1. 本資料において使用されている「当社」とは、ルネサスエレクトロニクス株式会社およびルネサスエレクトロニクス株式会社が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。

SuperH RISC engine C/C++ コンパイラパッケージ

アプリケーションノート：＜導入ガイド＞スタートアップルーチンガイド SH-3,SH-4,SH-4A 編

本ドキュメントでは、SuperH RISC engine C/C++ コンパイラ V.9 における High-performance Embedded Workshop（以下、HEW と略します）の生成ファイル、初期コーディング時の注意事項について説明します。

目次

1. サンプルプログラムの生成

1.1 プロジェクトジェネレータ設定

本書では、SH7750 を例にプロジェクトジェネレータ（HEW の[ファイル → 新規ワークスペース]メニューで起動）で下記手順に従って生成されたサンプルプログラムについて説明します。

(1) 新規ワークスペースの作成

プロジェクトタイプ“Application”を選択。

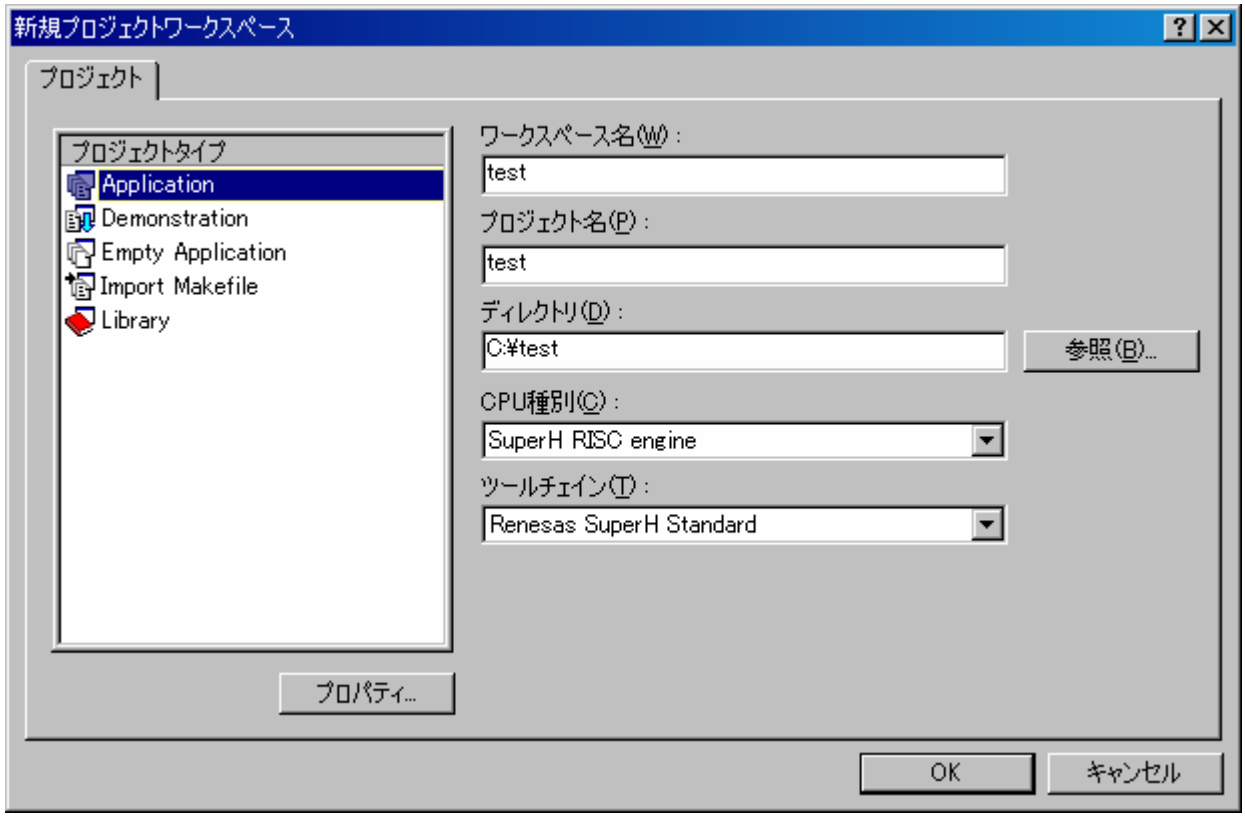

図 1-1

(2) CPU の選択

[CPU シリーズ]に“SH-4”
[CPU タイプ]に“SH7750”を選択

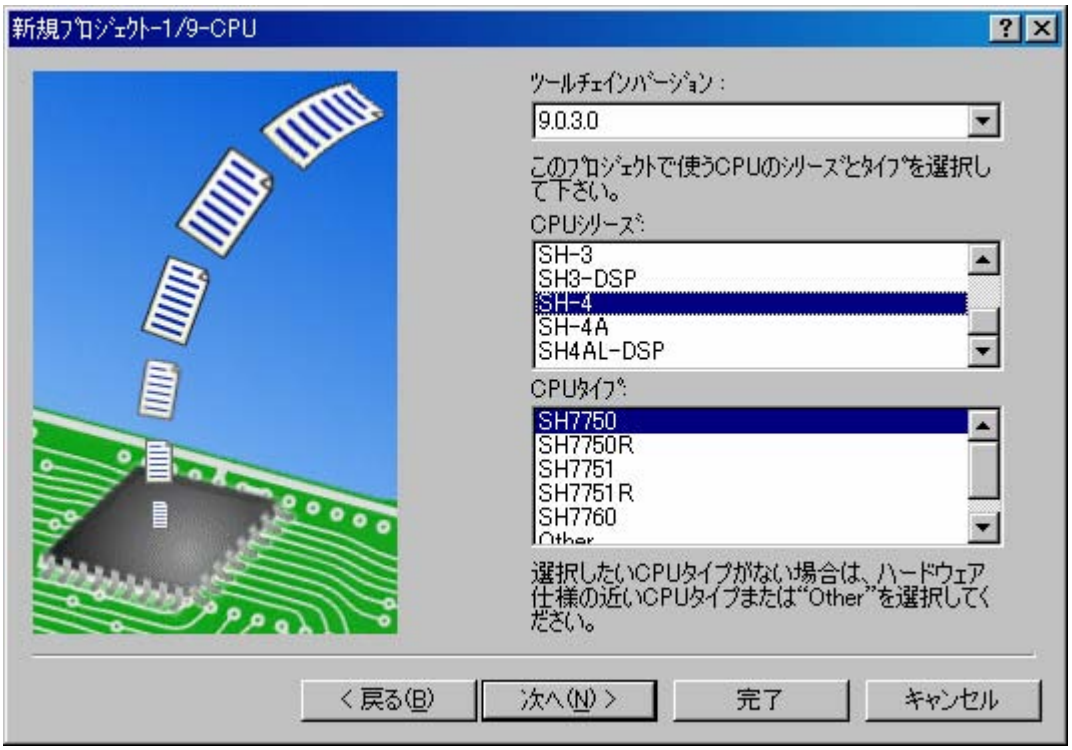

図 1-2

[補足]

- [CPU シリーズ]の設定は、SuperH RISC engine Standard Toolchain ダイアログ(以下、Toolchain ダイアログ)の CPU タブ に反映されます。
- [CPU タイプ]の設定は、intprg.src、vecttbl.src、iodefine.h、vect.inc の記述内容 及び 最適化リンケージエディタのメモリ配置設定に反映されます。選択したい CPU が無い場合には、DeviceUpdater を用いて CPU タイプの追加を行ってください。DeviceUpdater はルネサス WEB サイトよりダウンロード可能です。

(3) オプション設定

デフォルトのままで次に進む

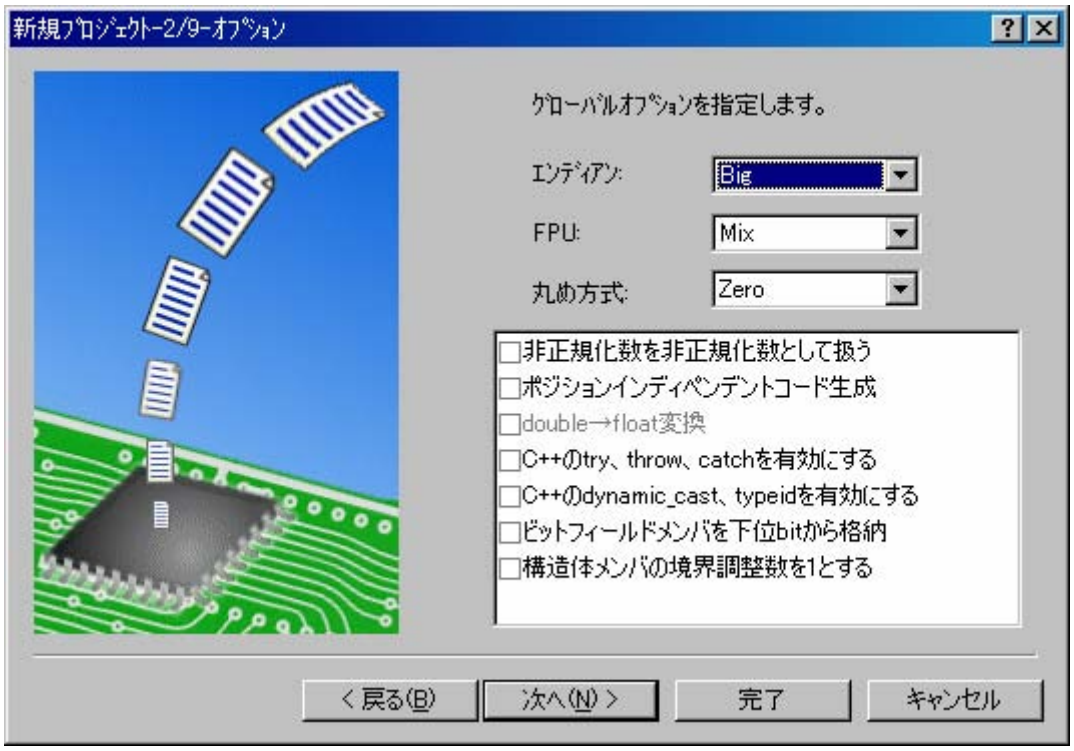

図 1-3

[補足]

- このダイアログの設定は、プロジェクト全体に共通で設定するオプションを指定します。設定項目は Toolchain ダイアログの CPU タブに反映されます。「(2)CPU の選択」の選択により、選択できる箇所が変わります。

(4) 生成ファイルの設定

“I/O ライブラリ使用”をチェック
“I/O ストリーム数”に“20”を指定

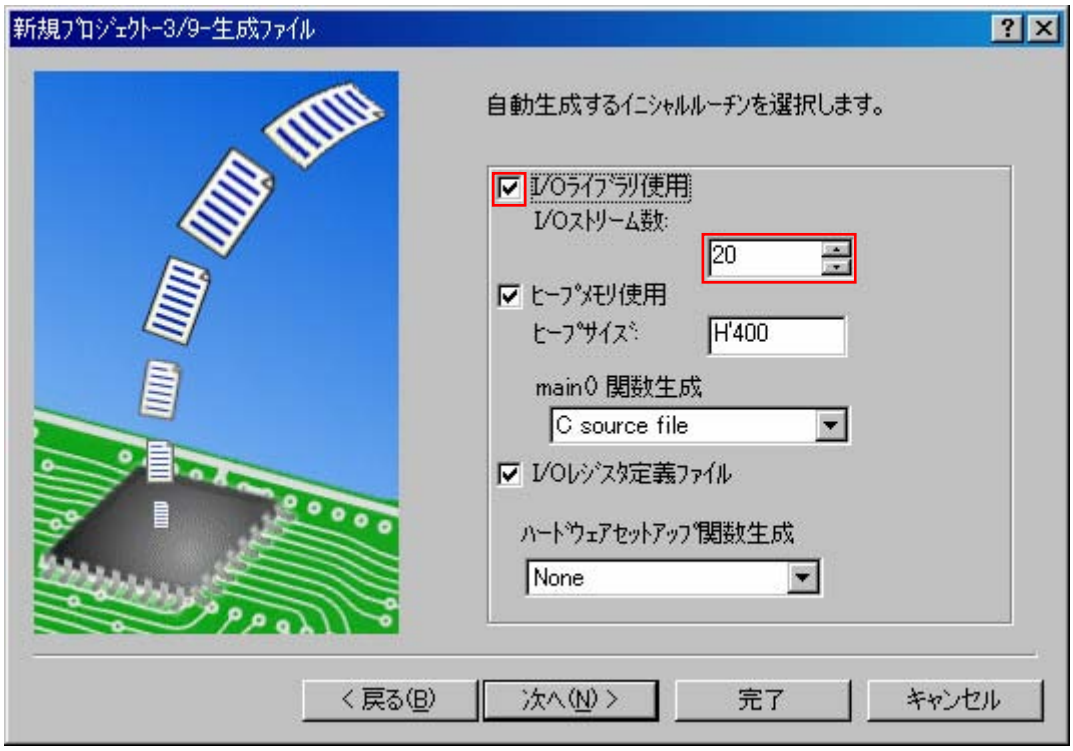

図 1-4

[補足]

- [I/O ライブラリ使用]にチェックを付けると、入出力関連の低水準インタフェースルーチン（open、close、write、read、lseek）及び 標準ライブラリの初期化プログラム（_INIT_IOLIB、_CLOSEALL）のサンプルプログラム（lowlvl.src、lowsrc.c、lowsrc.h）が生成されます。
- [I/O ストリーム数]の設定値は、lowsrc.h に反映されます。
- [ヒープメモリ使用]にチェックを付けると、メモリ管理関連の低水準インタフェースルーチン（sbrk）のサンプルプログラム（sbrk.h、sbrk.c）が生成されます。
- [ヒープサイズ]の設定値は、sbrk.h に反映されます。
- [main()関数生成]の設定により、メイン関数（C ソースファイルもしくは C++ソースファイル）及び、abort 関数の雛形が生成されます。
- [I/O レジスタ定義ファイル]のチェックを付けると、iodefine.h が生成されます。
- [ハードウェアセットアップ関数生成]の設定により、hwsetup.c、hwsetup.cpp、または、hwsetup.src が生成されます。ハードウェアセットアップ関数には、バスステートコントローラ（BSC）の初期化、シリアルの初期化等、ターゲットシステムに必要となるハードウェアの初期化処理を記述してください。なお、C/C++言語でプログラムを記述している場合は、いつスタックを使用するかを C/C++言語記述、コンパイルオプションでコントロールすることができません。そのため、SDRAM などの初期化が必要なメモリにスタック領域を確保している場合は、初期化前のメモリにアクセスを行ってしまう可能性があります。この場合は C 言語記述のプログラム実行前にアセンブリ言語を用いてメモリの初期化を行ってください。

(5) 標準ライブラリの設定

デフォルトのままで次に進む

図 1-5

[補足]

- このダイアログでは標準ライブラリ構築ツールで構築するライブラリを選択します。
- このダイアログでの設定は、Toolchain ダイアログの標準ライブラリタブに反映されます。

(6) スタック領域の設定

デフォルトのままで次に進む

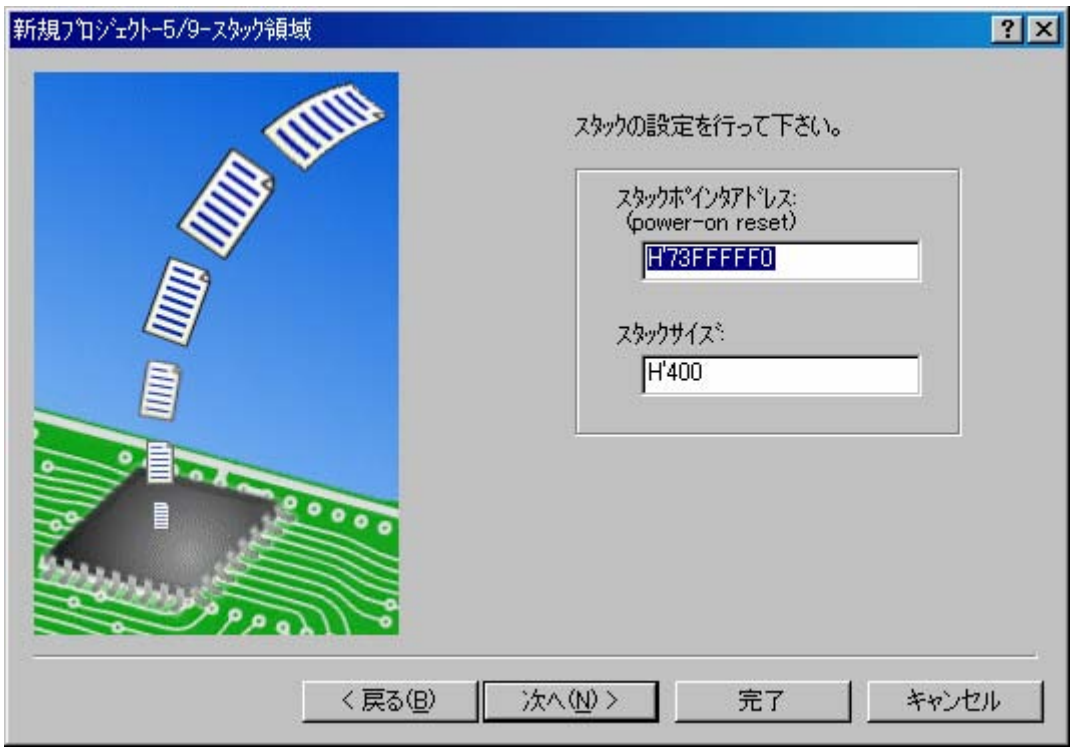

図 1-6

[補足]

- [スタックポインタアドレス]の設定は、最適化リンケージエディタの S セクションの設定に反映されます。
- [スタックサイズ]の設定は、stacksct.h に反映されます。
  ただし、「(7) ベクタの設定」で[ベクタテーブル定義]のチェックを外すと、stacksct.h は生成されません。

(7) ベクタの設定

デフォルトのままで次に進む

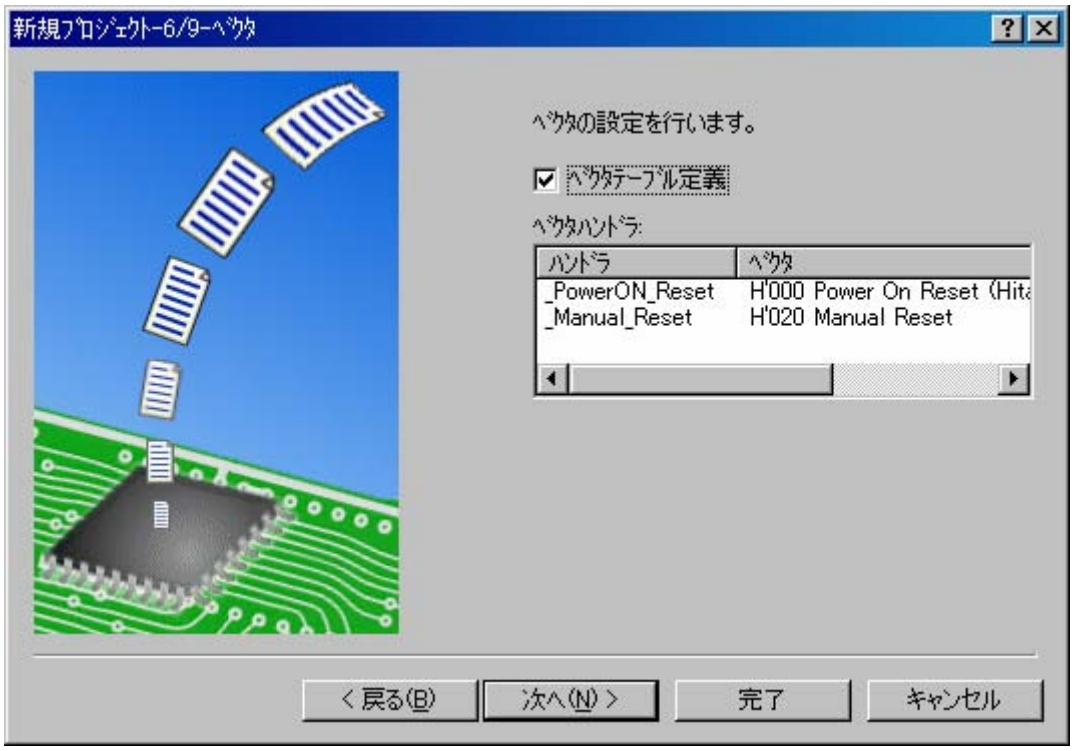

図 1-7

[補足]

- [ベクタテーブル定義]にチェックを付けると、env.inc、intprg.src、resetprg.c、stacksct.h、vecttbl.src、vect.inc、vhandler.src が生成されます。

(8) デバッガターゲットの設定

デフォルトのままで次に進む

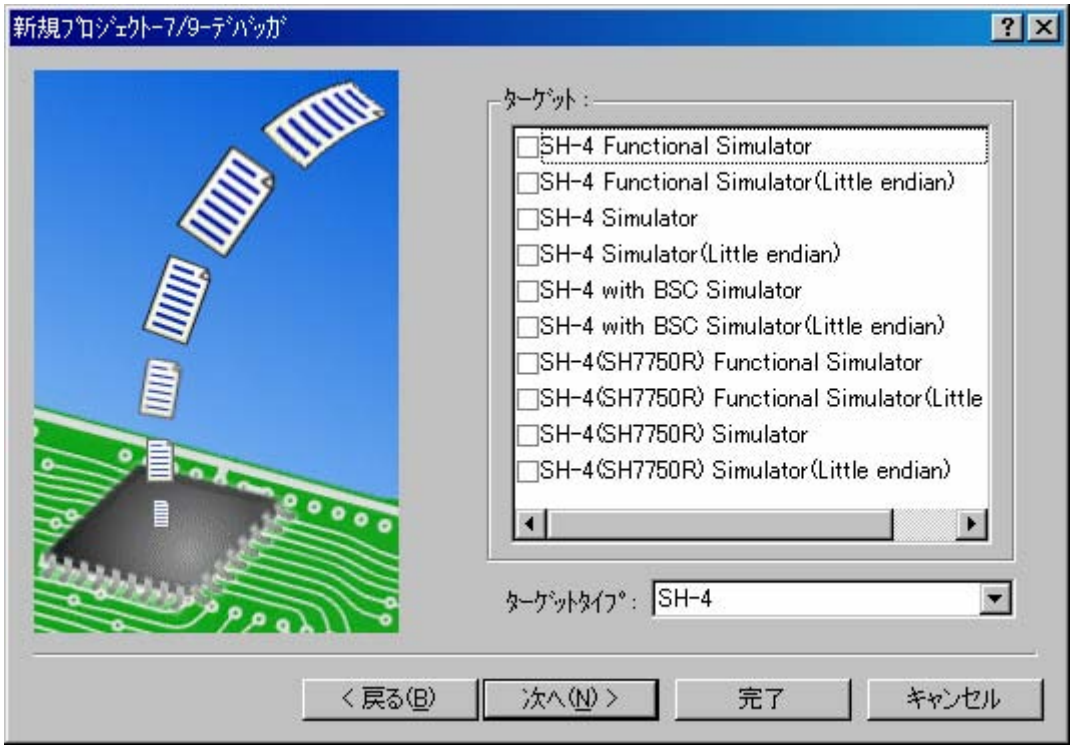

図 1-8

(9) 生成ファイル名の変更

完了を選択

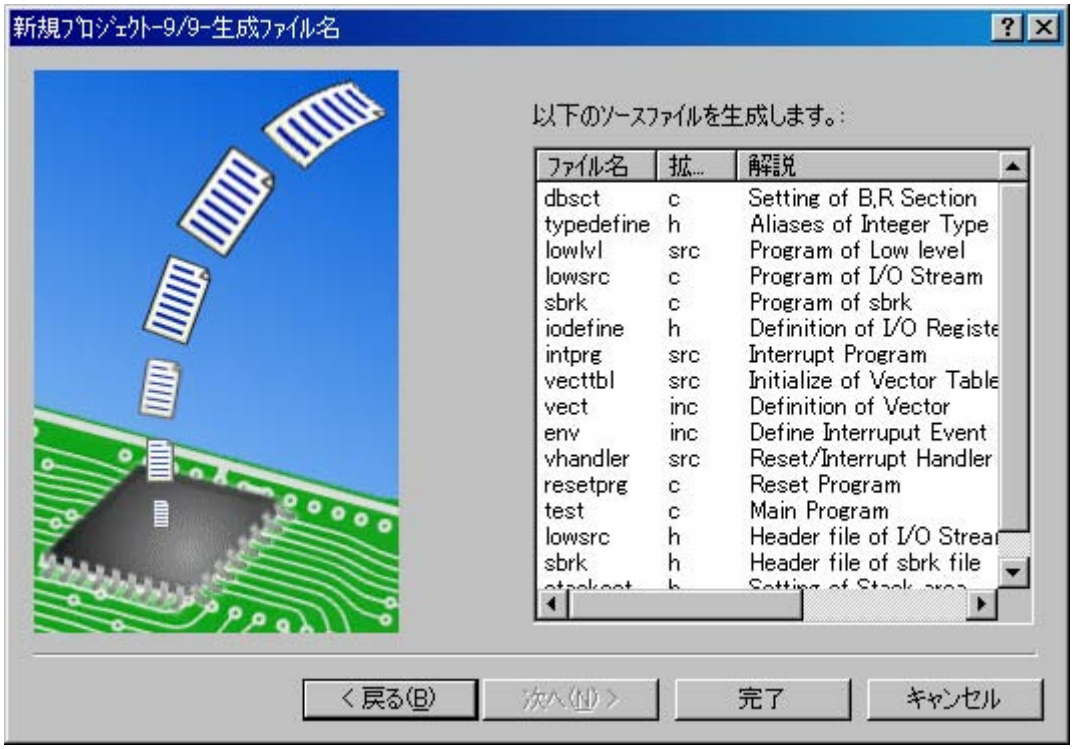

図 1-9

1.2 生成ファイル一覧

プロジェクトジェネレータで自動生成されたサンプルファイルは以下の通りです。

表 1-1 自動生成サンプルファイル一覧(1)

intprg.src	[割り込み関数] • 割り込み関数(ダミー)が定義されています。 • (7) [ベクタテーブル定義]の指定で生成されます。 詳しくは、3.4.例外処理ルーチン (intprg.src)ご参照ください。
lowlvl.src	[入出力低水準インタフェースルーチン] • 低水準インターフェースルーチン(write,read)から呼び出される、_charput、_charget が定義されています。 • 本プログラムはシミュレータでのみ動作します。 • (4)[I/O ライブラリ使用]の指定で生成されます。 詳しくは、5.2.入出力関連(lowlvl.src、lowsrc.c、lowsrc.h)をご参照ください。
vecttbl.src	[ベクタテーブル] • 例外処理ベクタテーブルが定義されています。 • (7) [ベクタテーブル定義]の指定で生成されます。 詳しくは、2.1.リセットハンドラ(vhandler.src、vecttbl.src、env.src、env.inc)をご参照ください。
vhandler.src	[例外処理ハンドラ] • 例外処理ハンドラが定義されています。 • (7) [ベクタテーブル定義]の指定で生成されます。 詳しくは、2.1.リセットハンドラ(vhandler.src、vecttbl.src、env.src、env.inc)を参照ください。
dbsct.c	[メモリ初期化対象の指定] • RAM の初期化及び ROM から RAM 領域への転送処理の対象が定義されています。 詳しくは、4.1.メモリ初期化関数_INITSCT(dbsct.c)をご参照ください。
lowsrc.c	[入出力低水準インタフェースルーチン] • 低水準インタフェースルーチン（write、read、open、close、lseek）が定義されています。 • 本プログラムは標準入出力関数のみ対応したシミュレータ向けプログラムです。 • (4)[I/O ライブラリ使用]の指定で生成されます。 詳しくは、5.2.入出力関連(lowlvl.src、lowsrc.c、lowsrc.h)をご参照ください。
resetprg.c	[リセット関数] • リセット関数(PowerON_Reset)が定義されています。 • (7) [ベクタテーブル定義]の指定で生成されます。 詳しくは、2.2.リセット関数 (resetprg.c)をご参照ください。
sbrk.c	[メモリ管理関連の低水準インタフェースルーチン] • メモリ管理関連の低水準インタフェースルーチン(sbrk)が定義されています。 • (4) [ヒープメモリ使用]の指定で生成されます。 詳しくは、5.1.メモリ管理関連(sbrk.c、sbrk.h)をご参照ください。
test.c (test.cpp)	[メインルーチン] • main 関数が定義されています。(C++言語使用時は abort 関数も定義されます) • (1)[プロジェクト名]で指定したファイル名称となります。
env.inc	[例外処理に関するレジスタのアドレス定義] • 例外事象レジスタ（EXPEVT）および割り込み事象レジスタ（INTEVT）が配置されるアドレスが定義されています。 詳しくは、2.1.リセットハンドラ(vhandler.src、vecttbl.src、env.src、env.inc)ご参照ください。

表 1-2自動生成サンプルファイル一覧(2)

lowsrc.h	[I/O 低レベル関数ヘッダ] • ファイルハンドラの数(= 同時に操作できるファイルの数)を指定する IOSTREAM マクロが定義されています。 • (4)[I/O ライブラリ使用]の指定で生成されます。 • (4)[I/O ストリーム数]の設定値が反映されます。 詳しくは、5.2.入出力関連(lowlvl.src、lowsrc.c、lowsrc.h)をご参照ください。
sbrk.h	[メモリ管理関連の低水準用ヘッダ] • ヒープ領域の全体サイズを指定する HEAPSIZE マクロが定義されています。 • (4) [ヒープメモリ使用]の指定で生成されます。 • (4) [ヒープサイズ]の設定値が反映されます。 詳しくは、5.1.メモリ管理関連(sbrk.c、sbrk.h)をご参照ください。
stacksct.h	[スタックセクションサイズヘッダ] • スタックセクションのサイズの定義がされています。 • (7) [ベクタテーブル定義]の指定で生成されます。 • (6) [スタックサイズ]の設定値が反映されます。 詳しくは、2.3.スタックサイズの設定(stacksct.h)をご参照ください。
typedefine.h	[型別名宣言ヘッダ] • 型の別名宣言がされています。
vect.inc	[ベクタテーブル用ヘッダ] • リセット関数、割り込み関数の原型宣言がされています。 • (7) [ベクタテーブル定義]の指定で生成されます。

2. リセット処理

HEW が生成するサンプルプログラムをパワーオンリセット後の動作に沿って説明します。

2.1 リセットハンドラ(vhandler.src、vecttbl.src、env.src、env.inc)

表 2-1に示す 5 つのリセット要因による例外が発生した場合、CPUは以下の動作を行います。

1. プログラムカウンタ(PC) に H'A0000000 を設定する。
2. 例外事象レジスタ（EXPEVT）に例外コードを設定する。
3. ステータスレジスタ（SR）のモードビット（MD）、レジスタバンクビット（RB）、例外／割り込みブロックビット（BL）を 1、FPU ディスエーブルビット（FD）を 0、割り込みマスクビット（I3～I0）に B'1111 を設定する。
4. ベクタベースレジスタ（VBR）に H'A0000000 を設定する。

表 2-1 例外一覧（リセット要因）

例外	例外コード	ベクタベース	_RESET_Vectors からのオフセット	例外処理ルーチン
パワーオンリセット	H'000	H'A0000000	H'000	_PowerON_Reset
マニュアルリセット	H'020	H'A0000000	H'004	_Manual_Reset
H-UDI	H'000	H'A0000000	H'000	_PowerON_Reset
命令 TLB	H'140	H'A0000000	H'028	_TBL_Reset
データ TLB	H'140	H'A0000000	H'028	_TBL_Reset

例外の要因は EXPEVT の値により判定します。サンプルプログラムでは、vhandler.src に定義されている“_ResetHandler”にて EXPEVT に設定された例外コードを参照し、それぞれの例外要因毎の処理関数にジャンプしています。
この例外判定処理を「リセットハンドラ」と呼び、例外要因毎の処理関数を「例外処理ルーチン」と呼びます。

[_ResetHandler の詳細]
(1) EXPEVT の値を読み込む。 …(a)、(b)
(2) EXPEVT の値から、 “_RESET_Vectors” からのオフセットを算出。（ EXPEVT / 8 ） …(c)、(d)
(3) “_RESET_Vectors” のアドレスに(2)を加算。 …(e)、(f)
(4) (3)が示すアドレスから例外処理ルーチンのアドレスを取得。 …(g)
(5) (4)で取得した例外処理ルーチンにジャンプ。 …(h)

```
                .include        "env.inc"
                .include        "vect.inc"

                .import         _RESET_Vectors
                .import         _INT_Vectors
                .import         _INT_MASK
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;       reset                                                           ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .section        RSTHandler,code
_ResetHandler:
                mov.l   #EXPEVT,r0              (a)
                mov.l   @r0,r0                  (b)
                shlr2   r0                      (c)
                shlr    r0                      (d)
                mov.l   #_RESET_Vectors,r1      (e)
                add     r1,r0                   (f)
                mov.l   @r0,r0                  (g)
                jmp     @r0                     (h)
                nop
```

リスト 2-1

[補足 1]
env.incにて、例外事象レジスタ（EXPEVT）のアドレスをシンボルEXPEVTに設定しています(リスト 2-2)。

```
EXPEVT:         .equ    H'FF000024
```

リスト 2-2

[補足 2]
“_RESET_Vectors”はvecttbl.srcに定義されています(リスト 2-3)。

例えば、パワーオンリセットが発生した時は、EXPEVT に例外コード H’000 が設定されます。そして、例外コード H’000 からオフセット値 0 が算出されるため、“_RESET_Vectors”の先頭にある_PowerON_Reset 関数にジャンプします。

```
        .include        "vect.inc"

        .section        VECTTBL,data
        .export         _RESET_Vectors

_RESET_Vectors:
;<<VECTOR DATA START (POWER ON RESET)>>
        ;H'000 Power On Reset (Hitachi-UDI RESET)
        .data.l  _PowerON_Reset
;<<VECTOR DATA END (POWER ON RESET)>>
;<<VECTOR DATA START (MANUAL RESET)>>
        ;H'020 Manual Reset
        .data.l  _Manual_Reset
;<<VECTOR DATA END (MANUAL RESET)>>
; Reserved
        .datab.l        8,H'00000000
;<<VECTOR DATA START (TBL RESET)>>
        ;H'140 TBL Reset (DATA TBL Reset)
        .data.l         _TBL_Reset
;<<VECTOR DATA END (TBL RESET)>>
```

リスト 2-3

[補足 3]
リセット要因例外が発生した後のPCは 0xA0000000 番地であるため、リセットハンドラは 0xA0000000 番地に配置する必要があります。 サンプルプログラムではリセットハンドラ(_ResetHandler) がRSTHandlerセクションに配置されているので、リンカのセクション設定にて、RSTHandlerセクションを 0xA0000000 番地に配置しています (図 2-1) 。

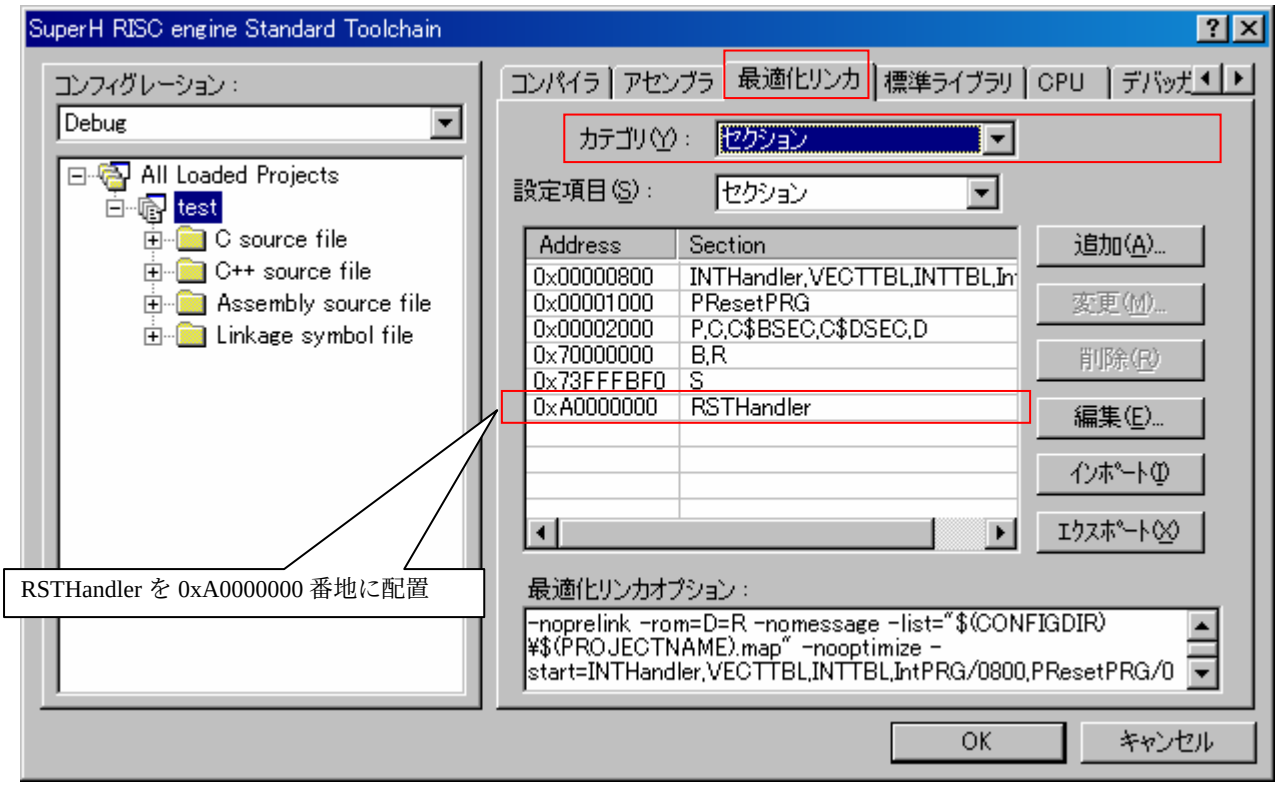

図 2-1

2.2 リセット関数 (resetprg.c)

パワーオンリセット時に呼び出される、PowerON_Reset の処理内容は以下の通りです。

C ソース	説明
`#include <machine.h>`	set_cr , set_vbr , sleep などの組み込み関数を使用する時は、machine.h を include します。
`#include <_h_c_lib.h>`	_INITSCT 関数を使用する時は _h_c_lib.h を include します
`//#include <stddef.h>`	errno を使用する時は、stddef.h を include します。
`//#include <stdlib.h>`	rand 関数を使用する時は、stdlib.h を include します。
`#include "typedefine.h"`	typedefine.h にて、型の別名宣言を行っています。
`#include "stacksct.h"`	#pragma stacksize の指定を行っています。
`#define SR_Init 0x000000F0`	ステータスレジスタ(SR)の設定値をマクロ定義しています。 SR の 4~7bit 目は割り込みマスクビット（I3～I0）で、H'F（B'1111）を設定し、割り込みマスクレベル 15(割り込み禁止)としています。
`#define INT_OFFSET 0x100UL`	リセットベクタテーブルのサイズをマクロ定義しています。ベクタベースレジスタ（VBR）設定処理でオフセット値として使用します。
`extern void INTHandlerPRG(void);`	関数 INTHandlerPRG の原型宣言を行っています。

`#ifdef __cplusplus`
`extern "C" {`
`#endif`
	C++言語の時に、extern "C" 宣言を行っています。

`Void PowerON_Reset(void);`	PowerON_Reset の原型宣言を行っています。
`Void Manual_Reset(void);`	Manual_Reset の原型宣言を行っています。
`Void main(void);`	main の原型宣言を行っています。

`#ifdef __cplusplus`
`}`
`#endif`	

`#ifdef __cplusplus`
`extern "C" {`
`#endif`	

`extern void INIT IOLIB(void);`	入出力関連標準ライブラリ初期化処理の原型宣言を行っています。
`extern void CLOSEALL(void);`	入出力関連標準ライブラリ終了関数の原型宣言を行っています。

`#ifdef __cplusplus`
`}`
`#endif`	

`//extern void srand(_UINT);`	rand 関数を使用する場合は srand の原型宣言を行います。
`//extern _SBYTE *_s1ptr;`	strtok 関数を使用する場合は変数 _s1ptr の宣言を有効にします。

`//#ifdef __cplusplus`
`//extern "C" {`
`//#endif`	

`//extern void HardwareSetup(void);`	HardwareSetup を呼び出す場合は原型宣言を行います。

`//#ifdef __cplusplus`
`//}`
`//#endif`	

C ソース	説明

`//#ifdef __cplusplus`
`//extern "C" {`
`//#endif`
`//extern void _CALL_INIT(void);`
	コンストラクタ呼び出し処理の原型宣言。グローバルクラスを使用する場合は有効にします。

`//extern void _CALL_END(void);`	デストラクタ呼び出し処理の原型宣言。グローバルクラスを使用する場合は有効にします。

`//#ifdef __cplusplus`
`//}`
`//#endif`	

`#pragma section ResetPRG`	PowerON_Reset 関数を PResetPRG セクションに配置します。
`#pragma entry PowerON_Reset`	PowerON_Reset 関数のエントリ関数指定。エントリ関数に指定するとレジスタの退避・回復コードを抑止することができます。 また、#pragma stacksize の指定があるため、PowerON_Reset 関数の先頭で R15 にスタックアドレスを設定するコードが生成されます。
`void PowerON_Reset(void)` `{`	
`set_vbr((void *)((_UINT *)& INTHandlerPRG - INT_OFFSET));`	ベクタベースレジスタ（VBR）設定処理を行います。 詳しくは、3.3.ベクタベースレジスタ(VBR)の設定(set_vbr関数)をご参照ください。
`INITSCT();`	メモリ処理化関数を呼び出します。 詳しくは、4.メモリ初期化をご参照ください。
`//  CALL_INIT();`	グローバルクラスオブジェクトのコンストラクタ呼び出し処理を行います。 詳しくは、6.C++言語を使用する上での注意(_CALL_INIT関数、CALL_END関数)をご参照ください
`_INIT_IOLIB();`	入出力関連の標準ライブラリの初期化を行います。 詳しくは、5.2.入出力関連(lowlvl.src、lowsrc.c、lowsrc.h)をご参照ください。
`//  errno=0;`	errno 初期化処理。 errno を使用する場合、有効にします。
`//  srand((_UINT)1);`	rand 関数を使用する場合、srand を呼び出して乱数表を初期化する必要があります。
`//  _s1ptr=NULL;`	strtok 関数を使用する場合、変数 _s1ptr の初期化が必要です。
`//  HardwareSetup();`	ハードウェア設定処理のダミー関数を呼び出します。
`set_cr(SR_Init);`	ステータスレジスタ(SR)の設定処理を行います。
`main();`	main 関数を呼び出します。
`_CLOSEALL();`	入出力関連の標準ライブラリの終了処理を行います。
`//  _CALL_END();`	デストラクタ呼び出し処理。 グローバルクラスを使用する場合、呼び出す必要があります。
`sleep();` `}`	sleep 命令を実行し sleep 状態に移行します。sleep 状態に移行することで、PowerON_Reset から抜け出さない様になっています。

`//#pragma entry Manual_Reset`
`void Manual_Reset(void)`
`{`
`}`
	マニュアルリセット関数(ダミー)

2.3 スタックサイズの設定(stacksct.h)

スタックポインタのアドレスは、ユーザプログラムにてR15 に設定する必要があります。サンプルプログラムでは、#pragma entry および #pragma stacksize の拡張機能を用いて、PowerON_Reset関数の先頭で設定処理が行われています(リスト 2-4)。

```
C ソースコード                          アセンブリコード
void PowerON_Reset(void)
                                        _PowerON_Reset:
                                                MOV.L       L12+2,R15   ; STARTOF S+SIZEOF S
{
    set_vbr((void *)((_UINT *)&
INTHandlerPRG - INT_OFFSET));
                                                MOV.L       L12+6,R6    ; _INTHandlerPRG
                                                MOV         #1,R1       ; H'00000001
            . . . .                                          . . . .
```

R15 にスタックアドレスを設定

リスト 2-4

stacksct.h（リスト 2-5）の#pragma stacksize指定により、サイズが 0x400 バイト分のスタック領域（Sセクション）がコンパイラにより確保されます。

スタックは上位アドレスから下位アドレスに向かって遡って使用されるため、Sセクションの先頭アドレスは(スタックポインタアドレス - スタックサイズ) とする必要があります。サンプルプロジェクトでは、スタックポインタアドレスに 0x73FFFFF0 を設定している(図 1-6)ため、最適化リンケージエディタのセクション配置で、 Sセクションの先頭アドレスを 0x73FFFBF0（0x73FFFFF0 – 0x400）としています(図 2-2)。

```
#pragma stacksize 0x400
```

リスト 2-5

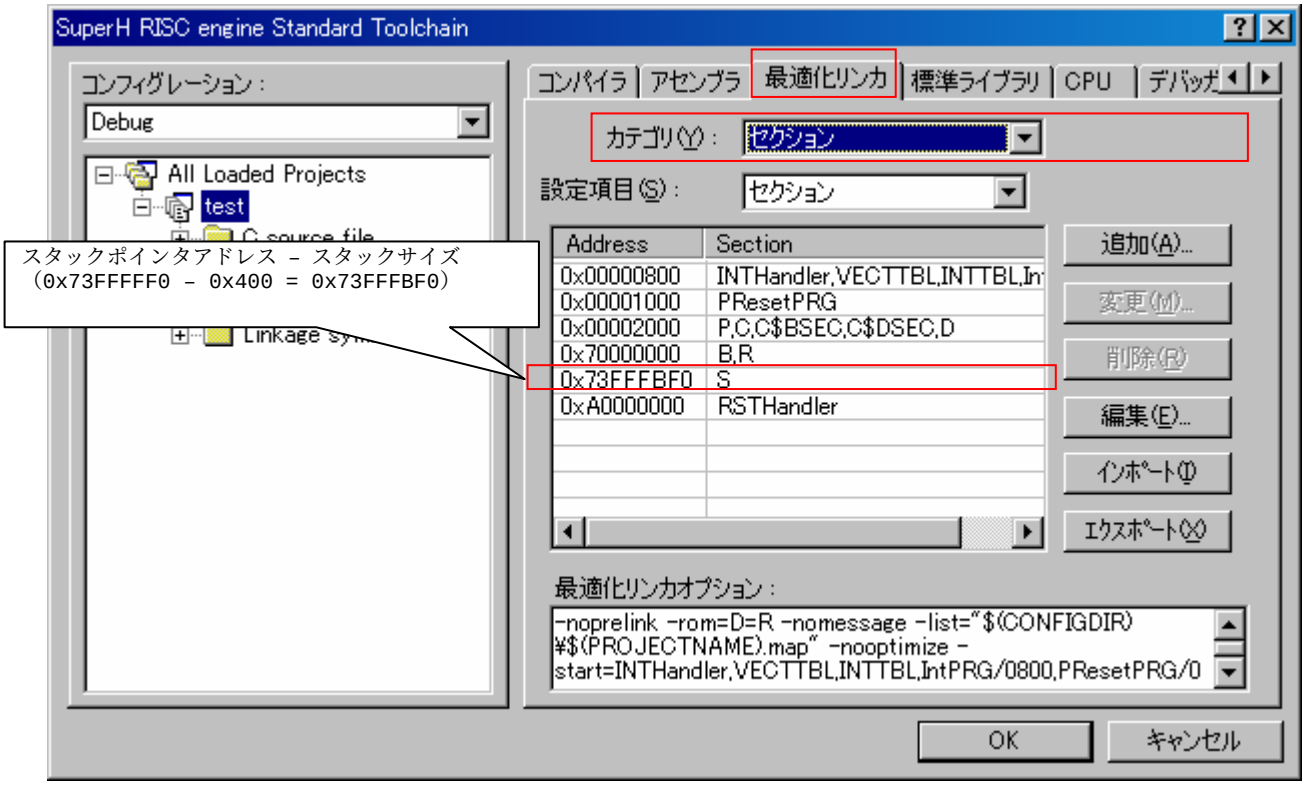

図 2-2

3. リセット以外の例外

リセット以外の例外には、一般例外、割り込みによる例外があります。リセット以外の例外が発生した場合ハードウェアでは以下の動作を行います。

1. ステータスレジスタ（SR）、プログラムカウンタ（PC）をそれぞれ退避ステータスレジスタ（SSR）、退避プログラムカウンタ（SPC）に退避します。
2. SR の BL ビット、MD ビット、RB ビットを 1 に設定し、PC に(VBR +例外要因毎のオフセット値)を設定します。これにより、バンクが切り替わり、処理モードが特権モードに変わり、割り込みがマスクされます。

- BL ビット
  BL ビットが 0 の場合は、例外、割り込みを受け付けます。BL ビットが 1 の場合、すべての割り込みをマクします。ユーザブレークを除く例外が発生した場合には、CPU の内部レジスタ、他のモジュールのレジスタは、マニュアルリセット後の状態になります。
- MD ビット
  MD ビットが 0 の場合、ユーザモードになります。1 の場合は特権モードになります。
- RB ビット
  RB ビットが 0 の場合、R0_BANK0～R7_BANK0 が、汎用レジスタ R0～R7 としてアクセスされます。
  RB ビットが 1 の場合、R0_BANK1～R7_BANK1 が、汎用レジスタ R0～R7 としてアクセスされます。

3.1 リセット以外の例外処理ハンドラ (vhandler.src、vecttbl.src、env.src)

リセット以外の例外が発生した場合、PC は(VBR +例外要因毎のオフセット値)に設定され、一般例外の場合は例外事象レジスタ(EXPEVT)に例外要因を設定し、割り込みの場合は割り込み事象レジスタ(INTEVT)に割り込み要因が設定されます。

例外要因に対するオフセット値と例外コード(EXPEVT/INTEVT)の一部を表 3-1に示します。

表 3-1 例外一覧(一部のみ掲載)

例外要因	オフセット	例外コード	例外処理ルーチン	index
データ TLB ミス例外（読み出し）	H'400	H'040	INT_TLBMiss_Load	0
データ TLB ミス例外（書き込み）	H'400	H'060	INT_TLBMiss_Store	1
初期ページ書き込み例外	H'100	H'080	INT_TLBInitial_Page	2
データ TLB 保護違反例外（読み出し）	H'400	H'0A0	INT_TLBProtect_Load	3
⋮				
無条件トラップ	H'100	H'160	INT_TRAPA	9
⋮				
ノンマスカブル割り込み	H'600	H'1C0	INT_NMI	12
命令実行後ユーザブレーク	H'100	H'1E0	INT_User_Break	13
外部割り込み 0	H'600	H'200	INT_Extern_0000	14
外部割り込み 1	H'600	H'220	INT_Extern_0001	15

サンプルプロジェクトでは vhandler.src に定義されている次の例外処理ハンドラが呼び出されます。

- 一般例外が発生した場合　：_INTHandlerPRG
- TLB ミス例外が発生した場合　：_TLBmissHandler
- 割り込みが発生した場合　：_IRQHandler

例外処理ハンドラはそれぞれの例外要因の処理関数(例外処理ルーチン)を呼び出し、例外処理ルーチンで処理が行われた後に再び例外処理ハンドラに戻り、例外処理ハンドラから通常処理に戻ります(図 3-1)。

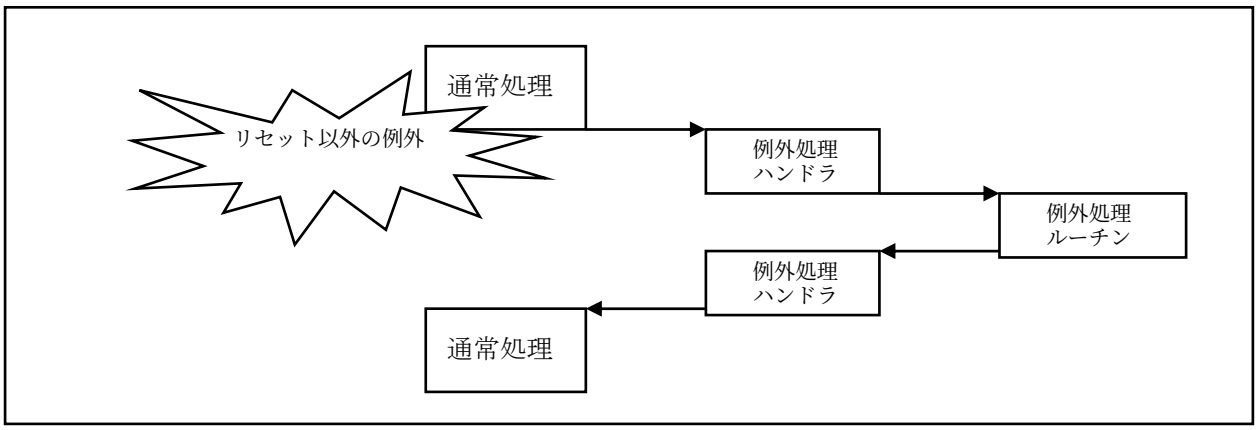

図 3-1

3.2 一般例外処理ハンドラ (_INTHandlerPRG)

例外処理ハンドラの説明として、一般例外処理ハンドラ_INTHandlerPRG(リスト 3-1) を例に説明します。
TLB ミス例外処理ハンドラ(_TLBmissHandler)、割り込みハンドラ(_IRQHandler) も同様の処理が行われます。
なお、割り込みハンドラで例外処理ルーチンのアドレスを算出する際には INTEVT が参照されます。

(補足)
例外処理ハンドラが呼び出される時は、SR が RB=1(バンク 1)、BL=1(割り込みマスク)となっています。

(1) レジスタ退避 …(a)
vhandler.src(リスト 3-2) に定義されているPUSH_EXP_BASE_REGマクロを呼び出し、汎用レジスタ、SSR、SPC、PR、FPSCRの退避を行います。

(2) 例外処理ルーチンアドレスの取得 …(b)
EXPEVT の値を取得し、 “_INT_Vectors” からのオフセットを求め（ (EXPEVT – 0x40) / 8 ）、例外処理ルーチンのアドレスを算出します。

(3) 割り込みマスクの取得 …(c)
EXPEVT の値より、INT_MASK（vecttbl.src に定義）からのオフセット値を算出(EXPEVT – 0x40) / 16）し、割り込み要因に応じた割り込みマスク値を取得します。

(4) 退避ステータスレジスタ（SSR）の設定 …(d)
現在のステータスレジスタ（SR）の値に対し、RB ビット、BL ビットを 0 クリアし、 (3)で取得した割り込みマスクを設定した値を SSR に設定します。

(5) 退避プログラムカウンタ（SPC）およびステータスレジスタ（SR）の設定 …(e)
(2)で取得した例外処理ルーチンのアドレスを SPC に設定し、 “__int_term” 関数のアドレスを PR に設定します。

(6) RTE 命令発行 …(f)
RTE 命令を実行します。 RTE 命令は SPC を PC に、SSR を SR に回復させ、SPC のアドレスに分岐します。
SPC には(5)の処理により例外処理ルーチンのアドレスが格納されているため、例外処理ルーチンに遷移します。
SSR には(4)の処理で求めたステータス値が格納されています。そのため、例外処理ルーチンに遷移した時は、バンク 0 に切り替わり、マスク値以上の割り込みが受け付け可能な状態となっています。例外処理ルーチン実行中に割り込みが受け付けられると、多重割り込みとなります。 (4)の処理で BL ビットを 0 クリアしないと、多重割り込みは禁止されます。

(7) 例外処理ルーチンからの復帰 …(g)

サンプルプログラムでは、例外処理ルーチンは通常関数で記述しています(#pragma interrupt は指定しないでください)。 そのため、例外処理ルーチンからはRTS 命令で復帰します。この時、PRには(5)の処理により “__int_term” のアドレスが格納されているため、 “__int_term” に遷移します。“__int_term” において、リスト 3-2のvhandler.srcに定義されているPOP_EXP_BASE_REGマクロを呼び出し、汎用レジスタ、SSR、SPC、PR、FPSCRを回復させます。最後に、RTE 命令で例外処理ハンドラから復帰します。

サンプルプログラムによる例外処理の状態遷移は図 3-2のとおりです。

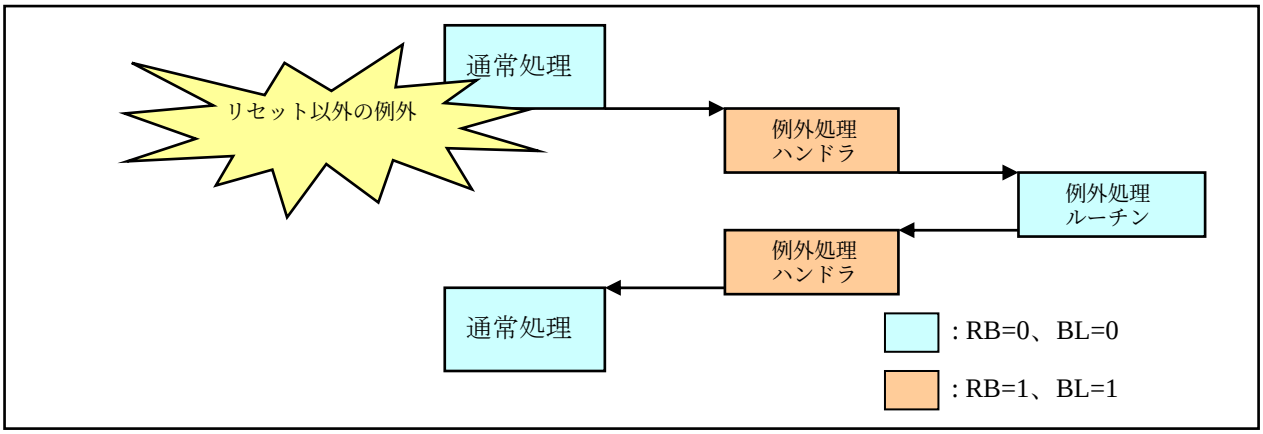

図 3-2

vhandler.src に定義された一般例外処理ハンドラ(INTHandlerPRG)のアセンブラソースは以下の通りです。

```
IMASKclr:       .equ    H'FFFFFF0F
RBBLclr:        .equ    H'CFFFFFFF
MDRBBLset:      .equ    H'70000000

                .import         _RESET_Vectors
                .import         _INT_Vectors
                .import         _INT_MASK

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;         exceptional interrupt                                         ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        .section    INTHandler,code
        .export      _INTHandlerPRG
_INTHandlerPRG:
_ExpHandler:
                PUSH_EXP_BASE_REG               (a)
;
                mov.l    #EXPEVT,r0             (b)
                mov.l    @r0,r1
                mov.l    #_INT_Vectors,r0
                add      #-(h'40),r1
                shlr2    r1
                shlr     r1
                mov.l    @(r0,r1),r3
;
                mov.l    #_INT_MASK,r0          (c)
                shlr2    r1
                mov.b    @(r0,r1),r1
                extu.b   r1,r1
;
                stc      sr,r0                  (d)
                mov.l    #(RBBLclr&IMASKclr),r2
                and      r2,r0
                or       r1,r0
                ldc      r0,ssr
;
                ldc.l    r3,spc                 (e)
                mov.l    #__int_term,r0
                lds      r0,pr
;
                rte                             (f)
                nop
;
                .pool
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;        Interrupt terminate                                            ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                .align   4
__int_term:                                     (g)
                mov.l    #MDRBBLset,r0
                ldc.l    r0,sr
                POP_EXP_BASE_REG
                rte
                nop
;
                .pool
;
```

リスト 3-1

[補足 1]
　例外処理リセットハンドラより呼び出されるレジスタの退避および回復を行うマクロPUSH_EXP_BASE_REG 、POP_EXP_BASE_REGはvhandler.srcで定義されています(リスト 3-2)。
　必要に応じて、浮動小数点レジスタの退避・回復を行って下さい。また、例外処理ルーチンを C 言語で記述し、コンパイラオプション macsave=0 を指定している場合は、MACH レジスタ、MACL レジスタも退避・回復する必要があります。

リスト 3-2に示したマクロでは、R0_BANK ～ R7_BANKをスタックへ退避（stc.l rn_bank,@-r15）、回復（ldc.l @r15+, rn_bank）しており、汎用レジスタ(R0～R7)の退避/回復は行っておりません。これは、割り込み受付け時、SR レジスタのRB ビットが自動的に 1 に設定されるため、例外発生前の汎用レジスタがバンクレジスタになっているためです。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;*              macro definition                                        *;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                .macro PUSH_EXP_BASE_REG
        stc.l   ssr,@-r15               ; save ssr
        stc.l   spc,@-r15               ; save spc
        sts.l   pr,@-r15                ; save context registers
        sts.l   fpscr,@-r15             ; save fpscr registers
        stc.l   r7_bank,@-r15
        stc.l   r6_bank,@-r15
        stc.l   r5_bank,@-r15
        stc.l   r4_bank,@-r15
        stc.l   r3_bank,@-r15
        stc.l   r2_bank,@-r15
        stc.l   r1_bank,@-r15
        stc.l   r0_bank,@-r15
                .endm
;
                .macro POP_EXP_BASE_REG
        ldc.l   @r15+,r0_bank           ; recover registers
        ldc.l   @r15+,r1_bank
        ldc.l   @r15+,r2_bank
        ldc.l   @r15+,r3_bank
        ldc.l   @r15+,r4_bank
        ldc.l   @r15+,r5_bank
        ldc.l   @r15+,r6_bank
        ldc.l   @r15+,r7_bank
        lds.l   @r15+,fpscr
        lds.l   @r15+,pr
        ldc.l   @r15+,spc
        ldc.l   @r15+,ssr
                .endm
```

リスト 3-2

[補足 2]
　SH7760 では一部の例外と割り込みが同じ例外コードとなっています(表＊＊)。　その為、サンプルプログラムではこれらの例外と割り込みを区別して処理できません。一般例外用の例外処理ハンドラ(_INTHandlerPRG)と割り込み用の例外処理ハンドラ(_IRQHandler)で別の関数テーブルを参照するなど、サンプルプログラムを変更してください。

表 3-2 SH7760 の例外コード(例外/割り込み)

例外要因	例外コード	例外処理ルーチン
一般 FPU 抑止例外	H'800	_INT_Illegal_FPU
スロット FPU 抑止例外	H'820	_INT_Illegal_slot_FPU
IRQ4	H'800	_INT_Illegal_FPU
IRQ5	H'820	_INT_Illegal_slot_FPU

3.3 ベクタベースレジスタ(VBR)の設定(set_vbr 関数)

VBRに任意のアドレスを設定する事により、リセット以外の例外処理ハンドラを任意のアドレスに配置する事ができます。VBRは組み込み関数のset_vbr関数を使用して設定する事ができます。サンプルプログラムでは、“INTHandlerPRG”の配置アドレスからVBRの設定値を算出しています(リスト 3-3)。 “INTHandlerPRG”はINTHandlerセクションの先頭に配置されているため、INTHandlerを任意のアドレスに配置することで、リセット以外の例外処理ハンドラを任意のアドレスに配置する事ができます。

```
resetprg.c
    #define  INT_OFFSET  0x100UL
                  ・・・
    set_vbr((void *)((_UINT *)&INTHandlerPRG - INT_OFFSET));
```

リスト 3-3

リセット以外の例外が発生した場合、(VBR +例外要因毎のオフセット値)にジャンプします。そのため、それぞれの例外処理ハンドラは表 3-3に示す位置に配置されている必要があります。サンプルプログラムでは、 “_INTHandlerPRG”を基点としたオフセット値を用いて、 “_TLBmissHandler” と “_IRQHandler” を配置しています(リスト 3-4)。

表 3-3 NTHandlerPRG からのオフセット値

例外種別	例外処理ハンドラ	VBR からのオフセット値	_INTHandlerPRG からのオフセット値
一般例外	_INTHandlerPRG	H’100	H’000
TLB **ミス例外**	_TLBmissHandler	H’400	H’300
割り込み	_IRQHandler	H’600	H’500

```
        .section    INTHandler,code
        .export     _INTHandlerPRG
_INTHandlerPRG:
      .
      .
      .
        .org    H'300
_TLBmissHandler:
      .
      .
      .
        .org    H'500
_IRQHandler:
      .
      .
      .
```

リスト 3-4

3.4 例外処理ルーチン (intprg.src)

リセット以外の例外では、例外処理ルーチンのダミー関数（_INT_TLBMiss_Load 関数、_INT_TLBMiss_Store関数など）がintprg.srcに定義されています（リスト 3-5）。

```
;H'040 TLB miss/invalid (load)
_INT_TLBMiss_Load
;H'060 TLB miss/invalid (store)
_INT_TLBMiss_Store
;H'080 Initial page write
_INT_TLBInitial_Page
      .
      .
      .
;H'820 Illegal slot FPU
_INT_Illegal_slot_FPU
    sleep
    nop
    .end
```

リスト 3-5

例外処理ルーチンを C 言語で記述したい場合は、ダミー関数をコメントアウトし、ダミー関数の先頭のアンダースコアを削除した関数名で C 言語関数を作成してください。 このとき、#pragma interrupt の指定は不要です。

例)

```
void INT_TLBMiss_Load(void)
{

}
```

リスト 3-6

4. メモリ初期化

サンプルプログラムでは標準ライブラリに含まれる_INITSCT 関数を呼び出しメモリの初期化を行います。__INITSCT 関数は次の初期化処理を行います。

- 初期化データ領域の初期化
- 未初期化データ領域の初期化

4.1 メモリ初期化関数_INITSCT(dbsct.c)

_INITSCT 関数を使用する場合には、<_h_c_lib.h>をインクルードし、標準ライブラリをリンクしてください。_INITSCT関数は、初期化データ領域の初期化対象をC$DSECセクションから、未初期化データ領域の初期化対象をC$BSECセクションから取得します。サンプルプログラムでは、dbsct.c(図 4-1)の構造体配列 “DTBL” に初期化データ領域の初期化処理の対象を、構造体配列 “BTBL” に未初期化データ領域の初期化処理の対象を定義しています。

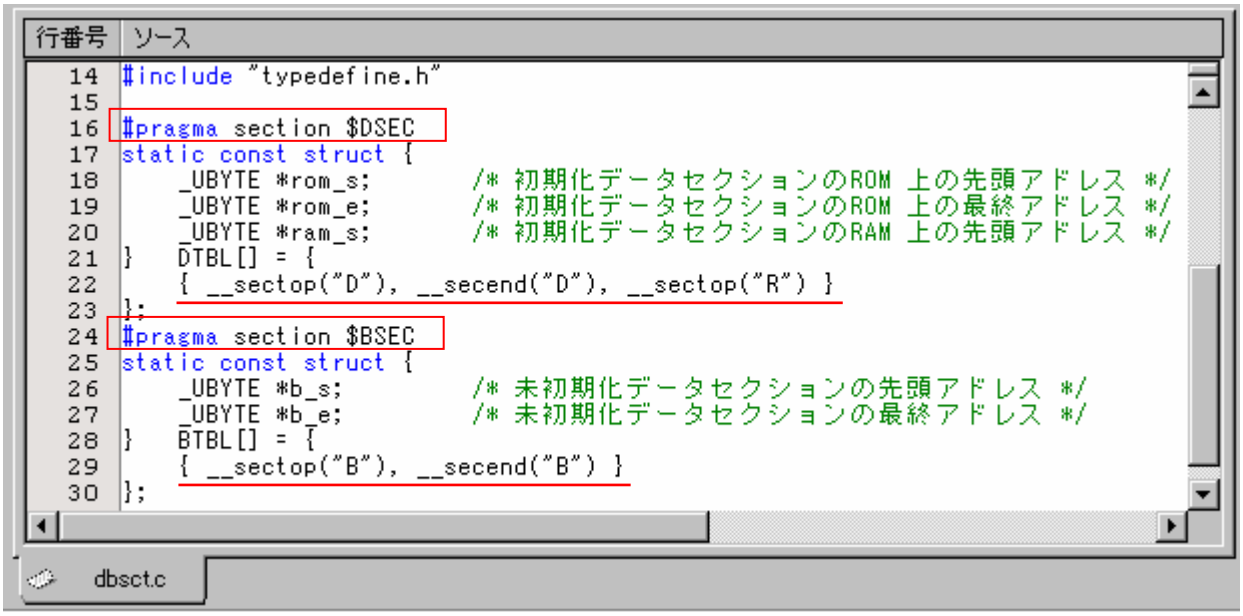

図 4-1

[初期化データ領域の初期化について]

初期化データは初期値を持つデータ(変数)です。初期値は ROM 領域に持つ必要がありますが、データはプログラム実行中に書き換えられる可能性があるため、RAM 領域に配置されている必要があります。__INITSCT 関数の初期化データ領域の初期化処理では、ROM 領域の初期値データを RAM 領域にコピーする処理を行います。また、ROM 領域に初期値を配置し、RAM 領域のアドレスでデータをアクセスするためには、リンカにて ROM 化支援オプションを指定する必要があります。(詳しくは、4.4.ROM化支援機能をご参照ください)

サンプルプロジェクトでは、dbsct.c の構造体配列 DTBL にて D セクションから R セクションへのデータのコピーを指定し、リンカにてROM化支援オプションの指定を行っています。(図 4-2)

[未初期化データ領域の初期化について]

C/C++言語では、初期値のない静的変数もしくは、初期値のない外部変数は０である必要があります。__INITSCT 関数の未初期化データ領域の初期化処理では、指定されたセクションを 0 クリアします。

サンプルプログラムでは、dbsct.c の構造体配列 BTBL にて B セクションの 0 クリアを指定しています。

4.2 D セクション以外の初期化データ領域が存在する場合

D セクション以外に初期化データ領域がある場合は構造体配列 “DTBL” に追加してください。
例えば、D1 セクションをR1 セクションにコピーする場合は、リスト 4-1の様に追加してください。また、ROM化支援オプションの指定も併せて行ってください。

```
#pragma section $DSEC
static const struct {
       _UBYTE  *rom_s
       _UBYTE  *rom_e
       _UBYTE  *ram_s
}   DTBL[] = {
    { __sectop("D"), __secend("D"), __sectop("R") },
    { __sectop("D1"),__secend("D1"),__sectop("R1")}
};
```

リスト 4-1

4.3 B セクション以外の未初期化データ領域が存在する場合

B セクション以外に未初期化データ領域がある場合には “BTBL” に追加してください。
例えば、B1 セクションを 0 クリアしたい場合は、リスト 4-2の様に追加してください。

```
#pragma section $DSEC
static const struct {
       _UBYTE  *b_s;    /*  未初期化データセクションの先頭アドレス  */
       _UBYTE  *b_e;    /*  未初期化データセクションの最終アドレス  */
}   BTBL[] = {
    { __sectop("B"), __secend("B") },
    { __sectop("B1"),__secend("B1")}
};
```

リスト 4-2

4.4 ROM 化支援機能

リンケージエディタの ROM 化支援機能を用いることにより、次の処理が行われます。

- ROM 上の初期化データ領域と同じ大きさの領域を RAM 上に確保します。
- 初期化データ領域に宣言したシンボルの参照が RAM 領域のアドレスを指すようにアドレス解決を自動的に行います。

次の手順でダイアログを表示し設定します。
Toolchain ダイアログ
→“最適化リンカ”タブを選択 → [カテゴリ]に“出力”を選択
→[オプション項目]に“ROM から RAM へマップするセッション”を選択

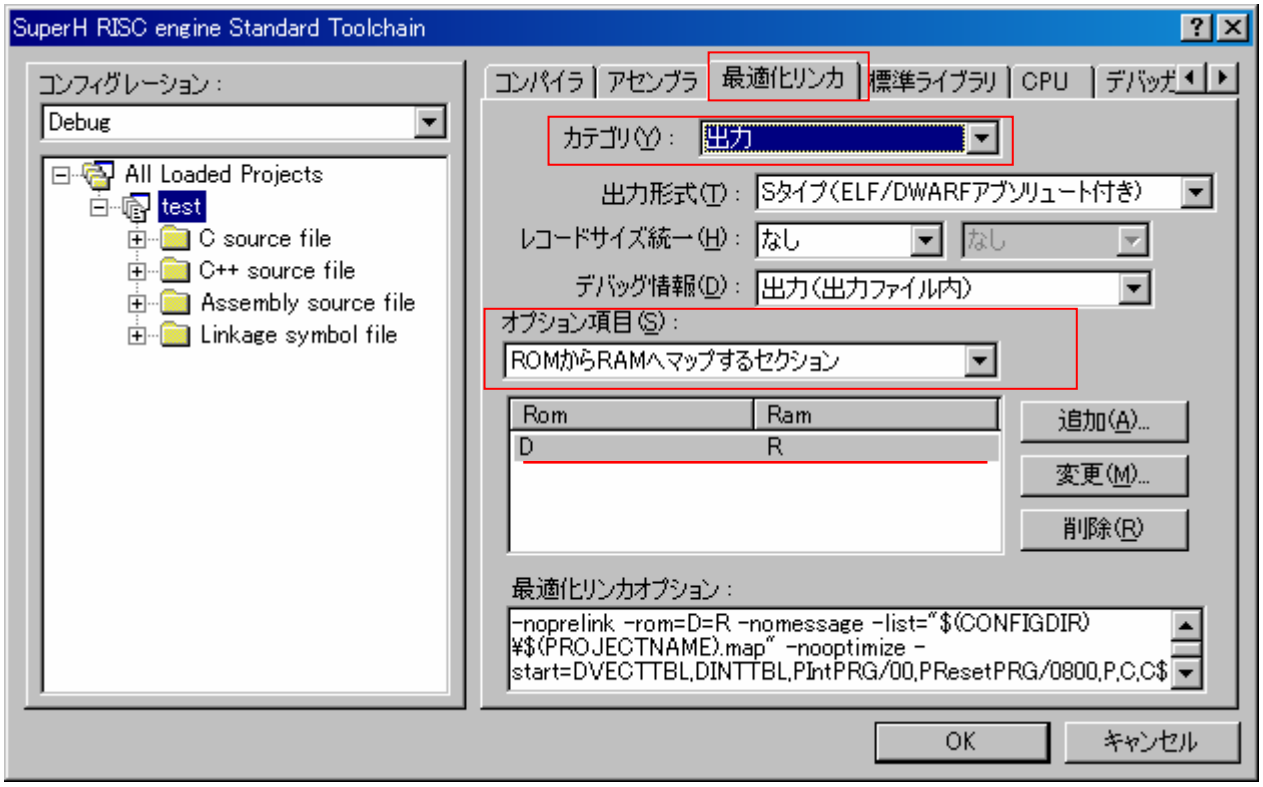

図 4-2

サンプルプロジェクトでは、ROM に D セクション、RAM に R セクションが指定されています。この指定により、リンク時に D セクションと同じ大きさの R セクションを RAM 上に確保し、初期化データ領域に宣言したシンボルの参照が RAM 領域の R セクションのアドレスを指すようにアドレス解決(リンケージ)をします。

5. 低水準インタフェースルーチンの設定

C/C++言語で開発をすると、標準入出力ライブラリ（fopen、printf、scanf など）、メモリ管理ライブラリ（malloc、free、new、delete）といった関数を使用する場合があります。しかし、これらはコンパイラで全ての機能を提供している訳ではありません。例えば、標準出力と言っても、その出力先は LCD、HDD、プリンタ、CD-R/RW など様々です。また、標準入力と言っても、DIP スイッチ、キーボード、マウス、携帯電話のボタン、タッチパネルなど様々です。また、それら機器により当然操作が異なります。従って、標準入出力、メモリ管理ライブラリの処理全てをコンパイラ側で提供することはできません。そこで標準入出力、メモリ管理ライブラリからは、低水準インタフェースルーチンと呼ばれる関数群を呼ぶようにしています。低水準インタフェースルーチンはユーザにて実装して頂く必要があります。低水準インタフェースルーチンには open、close、read、write、lseek、sbrk、errno_addr、wait_sem、signal_sem があります。

各ルーチンの仕様については、コンパイラユーザズマニュアル 9.2.2 実行環境の設定 (6) 低水準インタフェースルーチン をご参照ください。

5.1 メモリ管理関連(sbrk.c、sbrk.h)

HEWが生成するメモリ管理関連の低水準インタフェースルーチンのサンプル一覧を表 5-1に示します。

表 5-1 低水準インタフェースサンプル一覧(メモリ管理関連)

ソースファイル名	低水準インタフェース名	機能
sbrk.c	sbrk()	**ヒープメモリの確保関数** **引数で指定されたサイズ分のメモリを確保します。複数回呼び出された場合、下位アドレスから順に連続したメモリ領域を確保します。** HEAPSIZE **で定義されたサイズまでメモリの確保を行います。**
sbrk.h	HEAPSIZE	**ヒープ領域の全体サイズを指定する** HEAPSIZE **マクロが定義されています。**

[補足]

メモリ管理ライブラリ関数は sbrk 関数を呼び出してメモリの確保を行います。確保したメモリはライブラリ関数内で管理され、free、または delete 関数で解放された領域は再びヒープメモリとして再利用されます。sbrk 関数にメモリ確保を要求するサイズは_sbrk_size で指定されたサイズ毎です(デフォルト：1024)。確保したメモリが不足した場合は再度 sbrk 関数を呼び出します。ヒープメモリは確保、解放を繰り返すと空き領域サイズの合計は十分でも空き領域が小さな領域に分割しているために、新たに要求した大きなサイズの領域を確保できないという状況が生じることがあります。そのため、_sbrk_size = HEAPSIZE とし、1 回の sbrk 関数呼び出しでヒープメモリ領域を一括して取得する方法を推奨します。この取得方法を用いると、ヒープメモリの断片化が軽減され、ヒープ領域の管理処理の効率も向上します。

(例)

```
SBYTE  *sbrk(size_t size);
const size_t _sbrk_size = HEAPSIZE;     /* Specifies the minimum unit of      */
/*  コメントを外し、HEAPSIZEを初期値に設定する。  */
```

5.2 入出力関連(lowlvl.src、lowsrc.c、lowsrc.h)

HEWが生成する入出力関連の低水準インタフェースルーチンのサンプル一覧を表 5-2に示します。

表 5-2 低水準インタフェースサンプル一覧(入出力関連)

ソースファイル	低水準インタフェース名	機能
lowsrc.c	_INIT_IOLIB()	ファイルハンドラの初期化と標準入力(stdin)、標準出力(stdout)、標準エラー出力(stderr)用のファイルのオープンを行う関数。 標準入力、標準出力、標準エラー出力を使用しない場合は、これらのオープン処理を削除してください。 _INIT_IOLIB 関数以外の場所では、ファイルハンドラの操作は行わないでください。 ファイルハンドラのメンバ変数 _bufptr、_bufcnt、_bufbase、_buflen は、ファイルオープン後に setbuf 関数または setvbuf 関数を使用して設定してください。
lowsrc.c	_CLOSEALL()	閉じていないファイルを全て閉じる関数。
lowsrc.c	open()	ファイルオープン要求が標準入力／標準出力／標準エラー出力かの判別と、ファイルモードのチェックを行っています。 (サンプルプログラムでは、実際にファイルをオープンする処理は行っておりません)
lowsrc.c	close()	ファイル番号の範囲チェックとファイルモードのクリアを行っています。 ファイル番号が範囲エラーの場合は、エラーとして–1 を返します。
lowsrc.c	read()	ファイルモードのチェックを行った後、実際に文字を取得する charget 関数を要求された文字数回呼び出しする関数。エラーの場合は、–1 を返します
lowsrc.c	write()	ファイルモードのチェックを行った後、実際に文字を出力する charput 関数を要求された文字数回呼び出しする関数。エラーの場合は、–1 を返します。
lowsrc.c	lseek()	ダミー関数。HEW が生成する lseek 関数では、何も処理をしていません。
lowsrc.h	IOSTREAM	ファイルハンドラの数(= 同時に操作できるファイルの数)を指定するマクロ定義 ファイルハンドラの数を変更する場合には、IOSTREAM マクロを修正してください。 なお、HEW が生成した lowsrc.c では、_INIT_IOLIB 関数内で、標準入力(stdin)、標準出力(stdout)、標準エラー出力(stderr)の 3 つのファイルハンドラをオープンしています。従って、これらのオープン処理が有効になっている状況では、ユーザが使用できるファイルハンドラの数は、(IOSTREAM – 3)となります。
lowlvl.src	charget()	read()関数から呼ばれる文字入力関数。 シミュレータデバッガの I/O シミュレーションウィンドウからの文字入力を受け付けます。 本関数のアルゴリズムはシミュレータデバッガでのみ動作します。 実ターゲットでは動作できませんのでご注意ください。
lowlvl.src	charput()	write()関数から呼ばれる文字入力関数。 シミュレータデバッガの I/O シミュレーションウィンドウに文字を出力します。 本関数のアルゴリズムはシミュレータデバッガでのみ動作します。 実ターゲットでは動作できませんのでご注意ください。

6. C++言語を使用する上での注意(_CALL_INIT 関数、CALL_END 関数)

C++言語を使用して開発する際に、グローバルに宣言した変数を動的に初期化する時もしくはグローバルに宣言されたクラスオブジェクト（グローバルクラスオブジェクト）が存在する時は、_CALL_INIT 関数を事前に呼び出す必要があります。下記ソースプログラムにおいて、(a)、(b)がグローバルクラスオブジェクトです。

```
class A
{
...
};

A g_A;          ...(a)
A * g_pA;
static A s_A; ...(b)

void main()
{
    A a;
    A * p_a;
    static A s_a;
    g_pA = new A; delete g_pA;
    l_pA = new A; delete l_pA;
}
```

リスト 6-1

このクラスがコンストラクタを持っていた場合、そのクラスのメンバにアクセスする前に、既にコンストラクタが呼ばれている状況でなければなりません。例えば、下記 C++プログラムにおいて、(e)を実行する前に(c)が処理され、(d)のメンバ変数 a が 1 に初期化されていなければなりません。つまり、(c)のコンストラクタが呼ばれている必要があります。

```
class A
{
private:
    int a;
public:
    A(void) { a = 1; }            ...(c)
    int Get(void) { return a; }
};

A g_a;                            ...(d)

void main()
{
    int a = g_a.Get();            ...(e)
}
```

リスト 6-2

このコンストラクタ呼び出しのために、_CALL_INIT 関数が標準ライブラリとして準備されています。また、同様に、グローバルクラスオブジェクトのデストラクタを呼ぶための関数_CALL_END 関数も準備されています。_CALL_INIT 関数及び、_CALL_END 関数は、<_h_c_lib.h>に宣言されていますので、使用するソースファイル内で、<_h_c_lib.h>をインクルードします(f)。アプリケーションの開始前に_CALL_INIT 関数を呼び出し (g)、アプリケーションの終了後に _CALL_END 関数を呼び出してください(h)。

```
#include <_h_c_lib.h>     ...(f)

void PowerON_Reset_PC(void)
{
    _INITSCT();
    _CALL_INIT();         ...(g)

    main();

    _CALL_END();          ...(h)
    sleep();
}
```

リスト 6-3

また、コンストラクタ及びデストラクタを呼び出すための情報が C$INIT セクションに生成されています。このセクションはコンパイラにより自動的に生成されます。最適化リンケージエディタのメモリ配置設定により、C$INIT セクションを ROM 領域に配置してください。

7. C 言語による例外処理プログラムの記述方法

HEW のサンプルプログラムでは、例外処理ハンドラはアセンブラ言語により記述されています。C 言語でレジスタバンクを活用した例外処理ハンドラ、例外処理ルーチンを記述する場合は#pragma 拡張機能を使用します。

7.1 多重割り込みなし

多重割り込みを認めない場合の、C 言語による例外処理ハンドラ、例外処理ルーチンの記述方法を説明します。

■ 使用する#pragma 拡張機能

[例外処理ハンドラ]

- #pragma interrupt 関数名(bank)
  - RTE 命令で終了
  - レジスタの退避・回復規則
    SPC、SSR は退避・回復しない（sr_jsr 組み込み関数が使用されないため）
    R0-R7 は退避・回復しない
    上記以外は使用するレジスタのみ退避・回復を行う

[例外処理ルーチン]

- #pragma interrupt 関数名(rts)
  - RTS 命令で終了
  - レジスタの退避・回復規則
    SPC、SSR は退避・回復しない
    R0-R7 は退避・回復しない
    上記以外は使用するレジスタのみ退避・回復を行う

■ 例外処理フロー

前述の拡張機能を使用することで、以下の例外処理フローの例外処理ハンドラ、例外処理ルーチンを作成できます。

(1) 例外処理ハンドラ呼び出し前までのハードウェアの動作
例外発生後、例外発生時の PC を SPC、SR を SSR に退避させ、SR の RB を 1 に（BANK1 が汎用レジスタとして使用されます）、BL を 1 に設定（割り込み要求がマスクされます）し、例外要因に応じたアドレスに遷移します。

(2) 例外処理ハンドラ
例外処理ハンドラ内で使用する R0-R7 以外のレジスタを退避し、例外処理ルーチンを JSR 命令により呼び出します。例外処理ハンドラには、#pragma interrupt 関数名(bank) を指定します。

(3) 例外処理ルーチン
各例外要因に対する具体的な処理を記述します。例外処理ルーチンには、#pragma interrupt 関数名(rts) を指定します。例外処理ルーチンから例外処理ハンドラへは RTS 命令で復帰します。

(4) 例外処理ハンドラ
(2)で退避したレジスタを回復し、RTE 命令で例外処理から通常処理に復帰します。
RTE 命令を実行するとハードウェアにより、(1)で退避された SPC、SSR から SP,SR が回復されます。

■ 例外処理の処理フロー

C言語による例外処理ハンドラ、例外処理ルーチンの処理フローは
図 7-1の通りです。

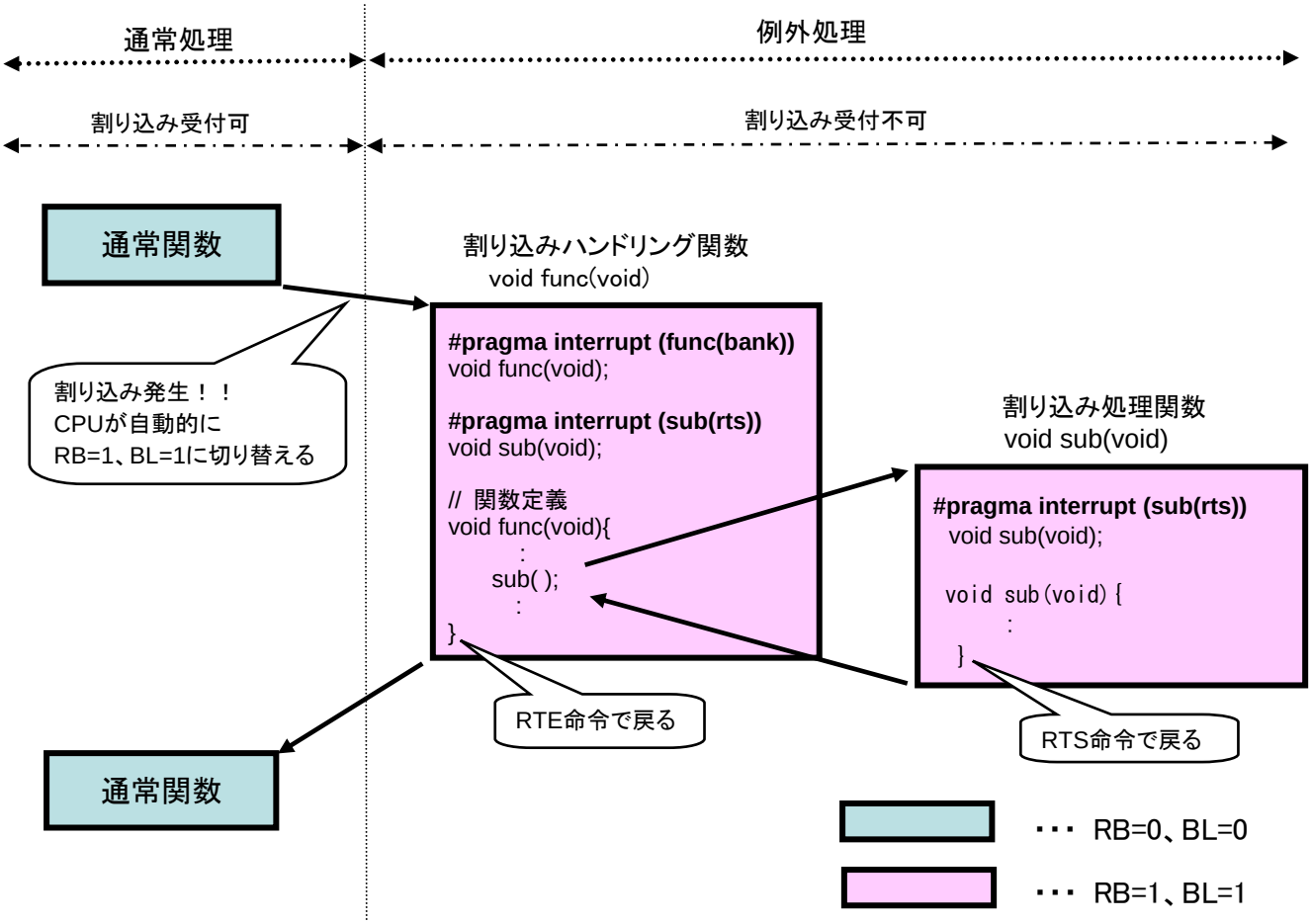

図 7-1

■ C ソースコード記述例

サンプルソースおよびそのアセンブリ展開コードを記載します。

サンプルソース

```
#include<machine.h>

extern void interrupt_handler(void);
extern void interrupt_routine(void);

#pragma interrupt interrupt_handler(bank)
#pragma interrupt interrupt_routine(rts)

/* ハンドリング関数 */
void interrupt_handler()
{
    /* 通常の関数呼び出し */
    interrupt_routine();
}

void interrupt_routine()
{
     /* 各種割り込み要因に対する処理を行う */
}
```

アセンブリ展開コード

```
_interrupt_handler:
         STS.L      PR,@-R15
         BSR        _interrupt_routine
         NOP
         LDS.L      @R15+,PR
         RTE
         NOP
_interrupt_routine:
         RTS
         NOP
```

リスト 7-1

7.2 多重割り込みあり

多重割り込みを認める場合の、C 言語による例外処理ハンドラ、例外処理ルーチンの記述方法を説明します。

■ 使用する拡張機能

[例外処理ハンドラ]

- #pragma interrupt 関数名(bank)
  - RTE 命令で終了
  - レジスタの退避・回復規則
    SPC、SSR を退避・回復する（sr_jsr 組み込み関数を使用するため）
    R0-R7 は退避・回復しない
    上記以外は使用するレジスタのみ退避・回復を行う

- 組み込み関数 void sr_jsr(void(*)(void)func, int imask)
  - 例外処理ルーチンの呼び出し関数
  - func：例外処理ルーチンのアドレス
  - imask：割り込みマスクビットの設定値
    imask が 1～15 の場合、マスクビットに imask を設定
    imask が 0 の場合、マスクビットは変更しない

[例外処理ルーチン]

- #pragma interrupt 関数名(sr_rts)
  - RTS 命令で終了
  - 出口処理で SR を RB=1、BL=1 に変更
  - レジスタの退避・回復規則
    SPC、SSR は退避・回復しない
    上記以外は使用するレジスタのみ退避・回復を行う

■ 例外処理フロー

前述の拡張機能を使用することで、以下の例外処理フローの例外処理ハンドラ、例外処理ルーチンを作成できます。

(1) 例外処理ハンドラ呼び出し前までのハードウェアの動作
例外発生後、例外発生時の PC を SPC、SR を SSR に退避させ、SR の RB を 1 に（BANK1 が汎用レジスタとして使用されます）、BL を 1 に設定（割り込み要求がマスクされます）し、例外要因に応じたアドレスに遷移します。

(2) 例外処理ハンドラ
例外処理ハンドラ内で使用する R0-R7 以外のレジスタ、SPC、SSR を退避します。組み込み関数 sr_jsr を用いて例外処理ルーチンを呼び出します。sr_jsr 関数を用いると、呼び出された例外処理ルーチンで RB=0（割り込み要求のマスクが解除されます）、BL=0（BANK0 が汎用レジスタとして使用され）、割り込みマスクレベル=imask となるようにコードが生成されます。 例外処理ハンドラには、#pragma interrupt 関数名(bank) を指定します。

(3) 例外処理ルーチン
各例外要因に対する具体的な処理を記述します。例外処理ルーチンには、#pragma interrupt 関数名(sr_rts) を指定します。例外処理ルーチンから例外処理ハンドラへ復帰する際には、SR を RB=1、BL=1 に変更します。
この例外処理ルーチンの実行中には、(2)で設定された imask よりも高いレベルの割り込みが受け付け可能となっています。

(4) 例外処理ハンドラ
(2)で退避したレジスタを回復し、RTE 命令で例外処理から通常処理に復帰します。
RTE 命令を実行するとハードウェアにより、(1)で退避された SPC、SSR から PC、SR が回復されます。

■ 例外処理の処理フロー(多重割り込みあり)

C言語による例外処理ハンドラ、例外処理ルーチンの処理フローは

図 7-2の通りです。

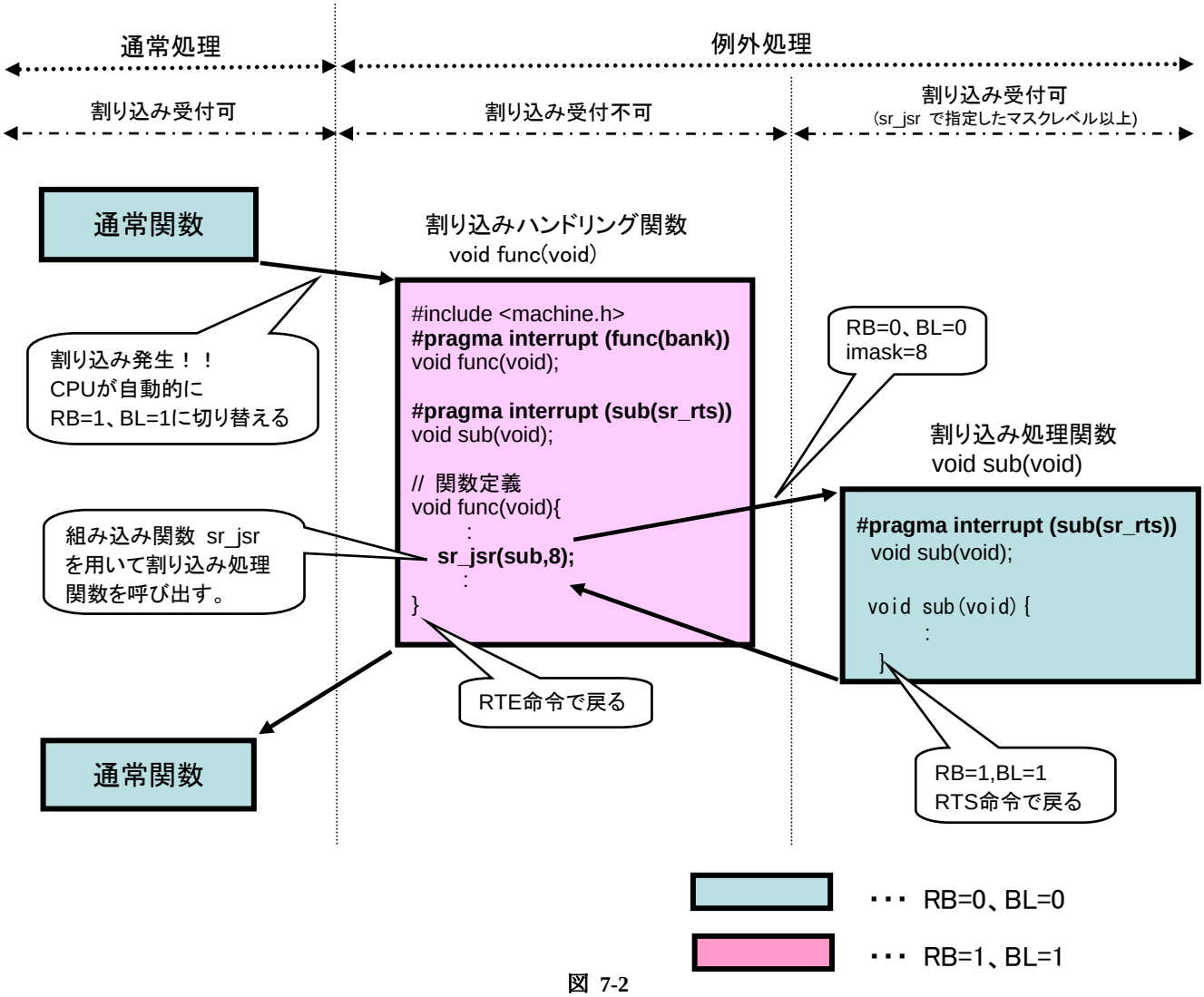

図 7-2

■ C ソースコード記述例

サンプルソースおよびそのアセンブリ展開コードを記載します。

サンプルプログラム

```
#include<machine.h>

extern void interrupt_handler(void);
extern void interrupt_routine(void);

#pragma interrupt interrupt_handler(bank)
#pragma interrupt interrupt_routine(sr_rts)

/* ハンドリング関数 */
void interrupt_handler()
{
    /* sr_jsr  に処理関数と
       割り込みマスクビット指定 */
    sr_jsr(interrupt_routine, 5);
}

void interrupt_routine()
{
    /* 各種割り込み要因に対する処理を行う */
}
```

アセンブリ展開コード

```
_interrupt_handler:
          MOV.L       R14,@-R15
          STS.L       PR,@-R15
          STC         SSR,@-R15
          STC         SPC,@-R15
          STC         SR,R4
          MOV.L       L12,R1      ; H'CFFFFF0F
          MOV         #80,R5      ; H'00000050
          MOV.L       L12+4,R14
                          ; _interrupt_routine
          AND         R1,R4
          OR          R5,R4
          LDC         R4,SR
          JSR         @R14
          NOP
          LDC         @R15+,SPC
          LDC         @R15+,SSR
          LDS.L       @R15+,PR
          MOV.L       @R15+,R14
          RTE
          NOP
_interrupt_routine:
          MOV.L       R0,@-R15
          MOV.L       R1,@-R15
          STC         SR,R0
          MOV.L       L12+8,R1    ; H'30000000
          OR          R1,R0
          MOV.L       @R15+,R1
          LDC         R0,SR
          RTS
          LDC.L       @R15+,R0_BANK
```

リスト 7-2

8. よくあるお問い合わせ

8.1 終了処理

■ 質問

メインルーチン(プロジェクト名.c) にある abort()はどのような時に使用する関数でしょうか？

■ 回答

abort 関数は C++言語で例外処理を使用する時に必ず必要となるルーチンです。(関数の定義が無いとリンク時にエラーとなります)

abort 関数が呼び出される時は例外が起きた時ですので、システムが暴走しないよう、sleep() などで終了処理を行ってください。

8.2 C++関数、C 関数の相互呼び出し

■ 質問

extern ”C” { と } で関数宣言が囲われていますが、何のために囲われているのでしょうか?

■ 回答

C++関数から C 関数を呼び出す場合は C++ソース内の C 関数の原型宣言に対して「extern "C"」宣言を指定する必要があります。C 関数から C++関数を呼び出す時は、C++ソース内の C++関数の原型宣言に対して「extern "C"」宣言を指定する必要があります。

C++言語では関数の多重定義が行えるため、同じ関数名で異なる関数が複数存在する可能性があります。そのため、コンパイラは内部的に関数名に引数の名前などを付加してシンボル名を管理しています。C 関数では多重定義がありませんので、このようなシンボル名の管理はされません。

C++関数に「extern "C"」宣言を行うと、シンボル名の管理方法が C 関数と同じとなります。この事により、C 関数、C++関数の相互呼び出しが可能となります。

なお、「extern "C"」を用いて宣言した C++関数は多重定義できませんので、ご注意ください。

- 「extern "C"」宣言を用いることにより C オブジェクトプログラムの関数を参照できます。

```
(C++プログラム)

extern "C" void CFUNC();
void main(void)
{
    X XCLASS;
    XCLASS.SetValue(10);

    CFUNC();
}
```

```
(Cプログラム)

extern void CFUNC();
void CFUNC()
{
    while(1)
    {
        a++;
    }
}
```

- 「extern "C"」宣言を用いることにより C++オブジェクトプログラムの関数を参照できます。

```
(Cプログラム)

void CFUNC()
{
    CPPFUNC();
}
```

```
(C++プログラム)

extern "C" void CPPFUNC();
void CPPFUNC(void)
{
    while(1)
    {
        a++;
    }
}
```

ホームページとサポート窓口<website and support,ws>

ルネサステクノロジホームページ

http://japan.renesas.com/

お問合せ先

http://japan.renesas.com/inquiry

csc@renesas.com

改訂記録<revision history,rh>

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2007.6.1	—	初版発行

安全設計に関するお願い

1. 弊社は品質、信頼性の向上に努めておりますが、半導体製品は故障が発生したり、誤動作する場合があります。弊社の半導体製品の故障又は誤動作によって結果として、人身事故、火災事故、社会的損害などを生じさせないような安全性を考慮した冗長設計、延焼対策設計、誤動作防止設計などの安全設計に十分ご留意ください。

本資料ご利用に際しての留意事項

1. 本資料は、お客様が用途に応じた適切なルネサス テクノロジ製品をご購入いただくための参考資料であり、本資料中に記載の技術情報についてルネサス テクノロジが所有する知的財産権その他の権利の実施、使用を許諾するものではありません。
2. 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他応用回路例の使用に起因する損害、第三者所有の権利に対する侵害に関し、ルネサス テクノロジは責任を負いません。
3. 本資料に記載の製品データ、図、表、プログラム、アルゴリズムその他全ての情報は本資料発行時点のものであり、ルネサス テクノロジは、予告なしに、本資料に記載した製品または仕様を変更することがあります。ルネサス テクノロジ半導体製品のご購入に当たりましては、事前にルネサス テクノロジ、ルネサス販売または特約店へ最新の情報をご確認頂きますとともに、ルネサス テクノロジホームページ(http://www.renesas.com)などを通じて公開される情報に常にご注意ください。
4. 本資料に記載した情報は、正確を期すため、慎重に制作したものですが万一本資料の記述誤りに起因する損害がお客様に生じた場合には、ルネサス テクノロジはその責任を負いません。
5. 本資料に記載の製品データ、図、表に示す技術的な内容、プログラム及びアルゴリズムを流用する場合は、技術内容、プログラム、アルゴリズム単位で評価するだけでなく、システム全体で十分に評価し、お客様の責任において適用可否を判断してください。ルネサス テクノロジは、適用可否に対する責任は負いません。
6. 本資料に記載された製品は、人命にかかわるような状況の下で使用される機器あるいはシステムに用いられることを目的として設計、製造されたものではありません。本資料に記載の製品を運輸、移動体用、医療用、航空宇宙用、原子力制御用、海底中継用機器あるいはシステムなど、特殊用途へのご利用をご検討の際には、ルネサス テクノロジ、ルネサス販売または特約店へご照会ください。
7. 本資料の転載、複製については、文書によるルネサス テクノロジの事前の承諾が必要です。
8. 本資料に関し詳細についてのお問い合わせ、その他お気付きの点がございましたらルネサス テクノロジ、ルネサス販売または特約店までご照会ください。