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-1,SH-2,SH-2A 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、SH7046 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。

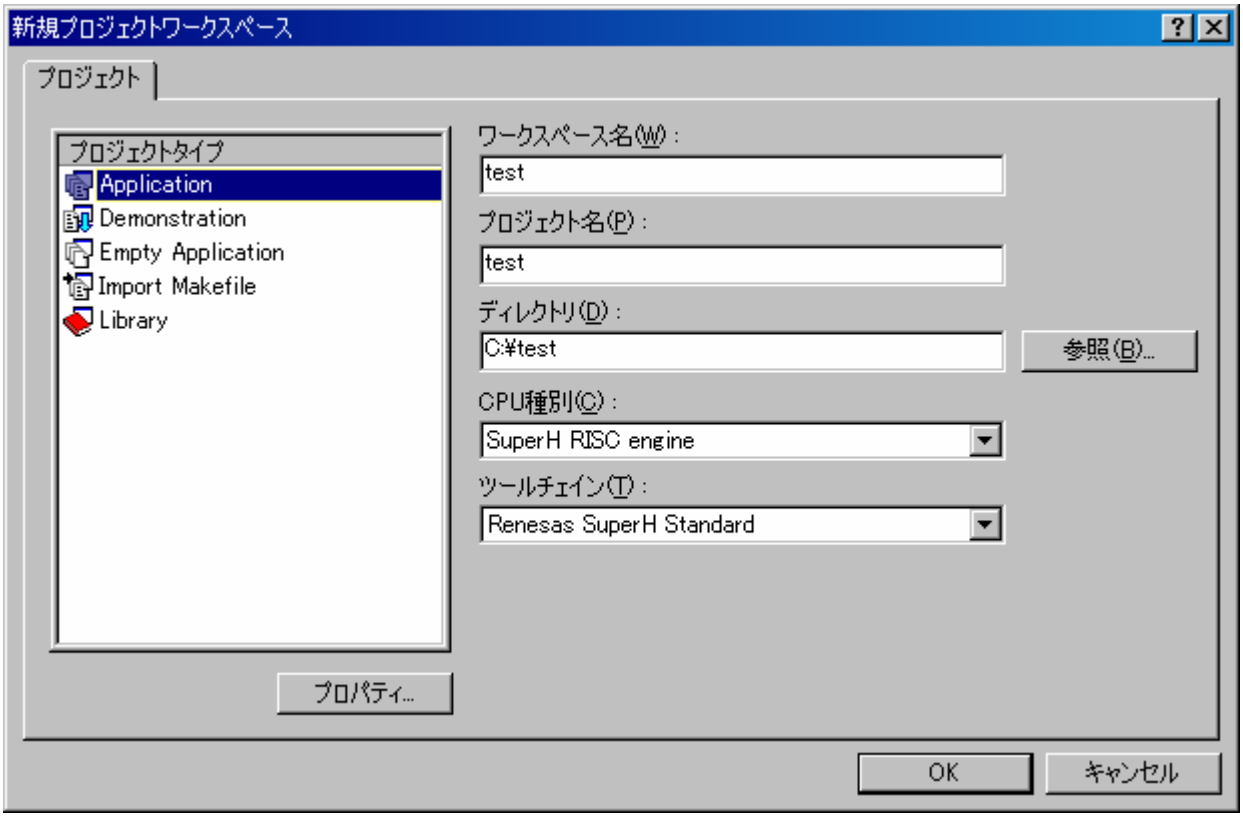

図 1-1

(2) CPU の選択

[CPU シリーズ]に“SH-2”
[CPU タイプ]に“SH7046”を選択

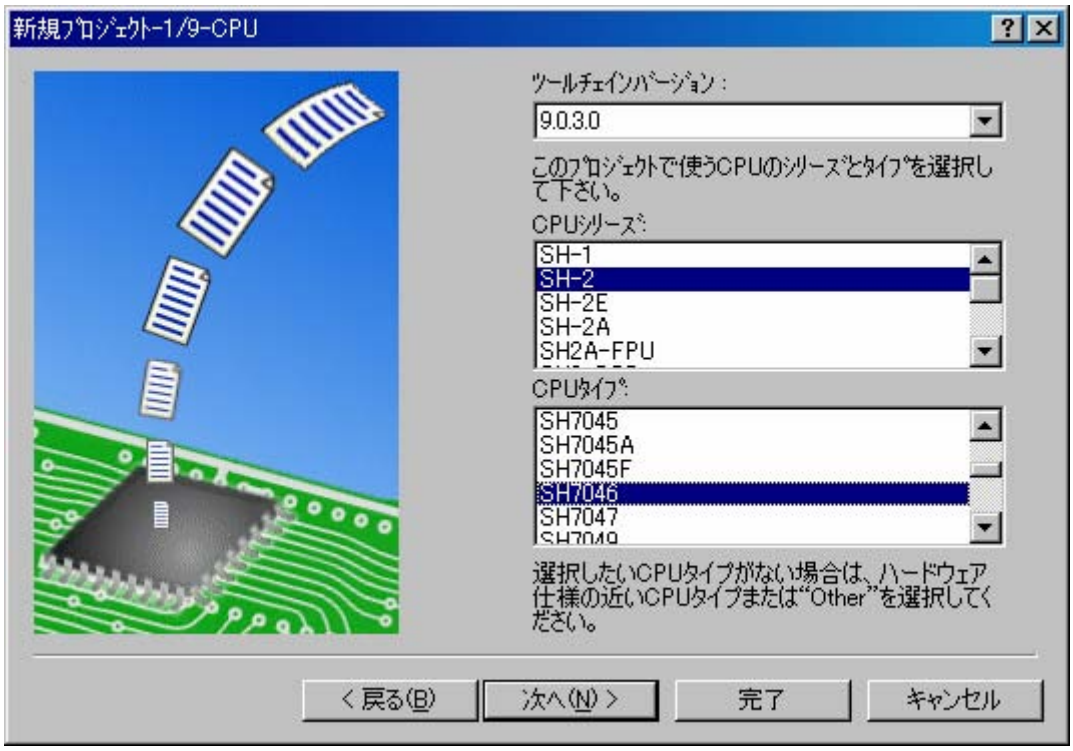

図 1-2

[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.c、vecttbl.c、iodefine.h、vect.h 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

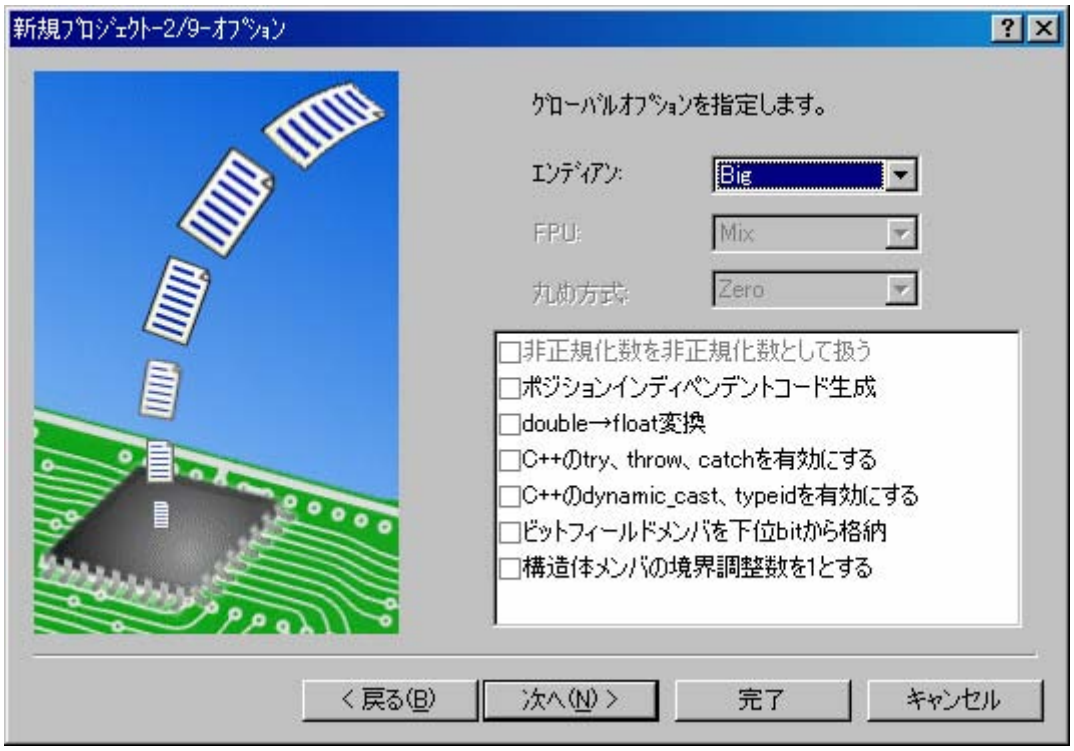

図 1-3

[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

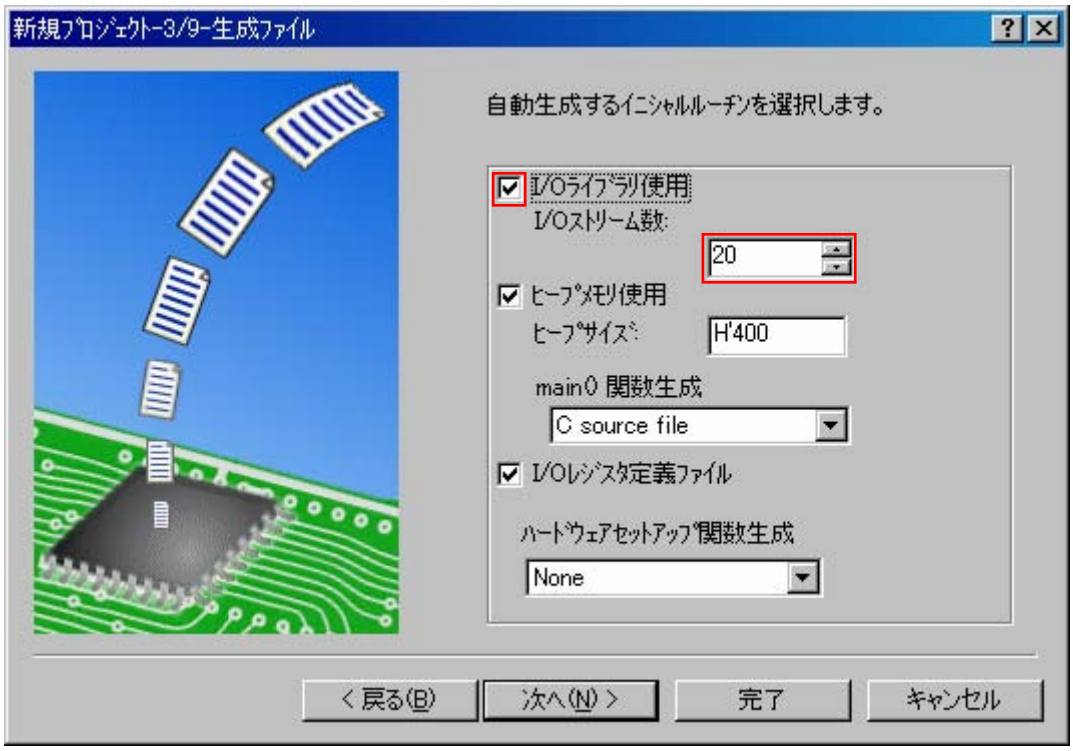

図 1-4

[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

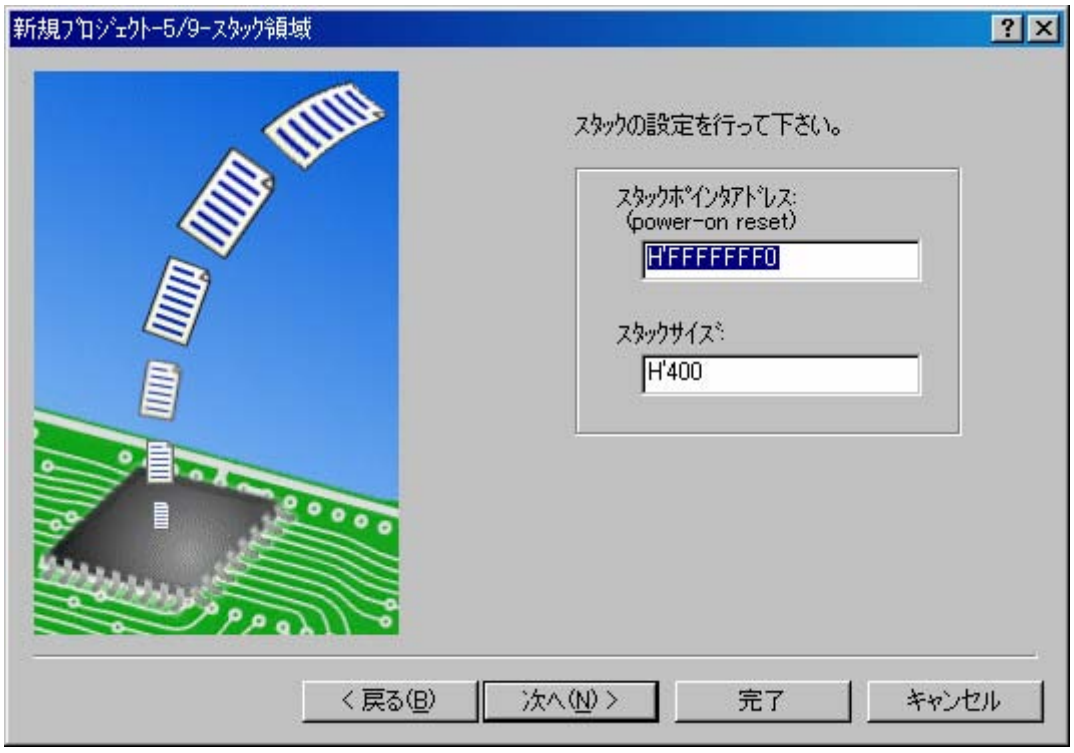

図 1-6

[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

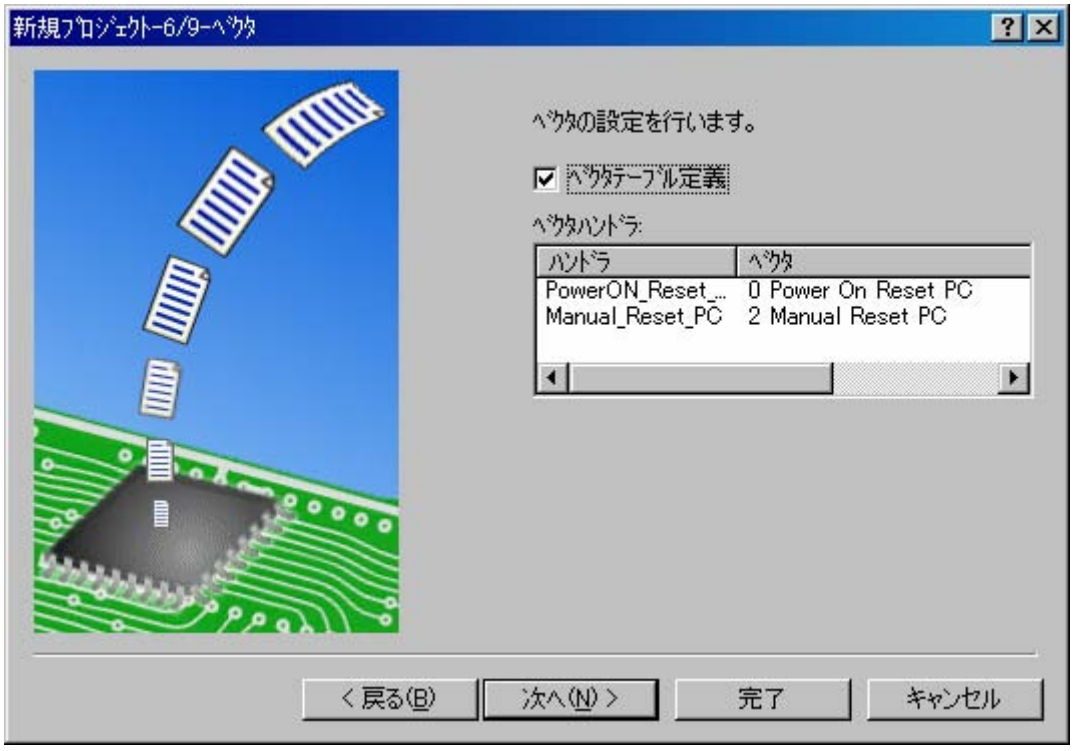

図 1-7

[補足]

・ [ベクタテーブル定義]にチェックを付けると、intprg.c、resetprg.c、stacksct.h、vect.c、vect.h 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

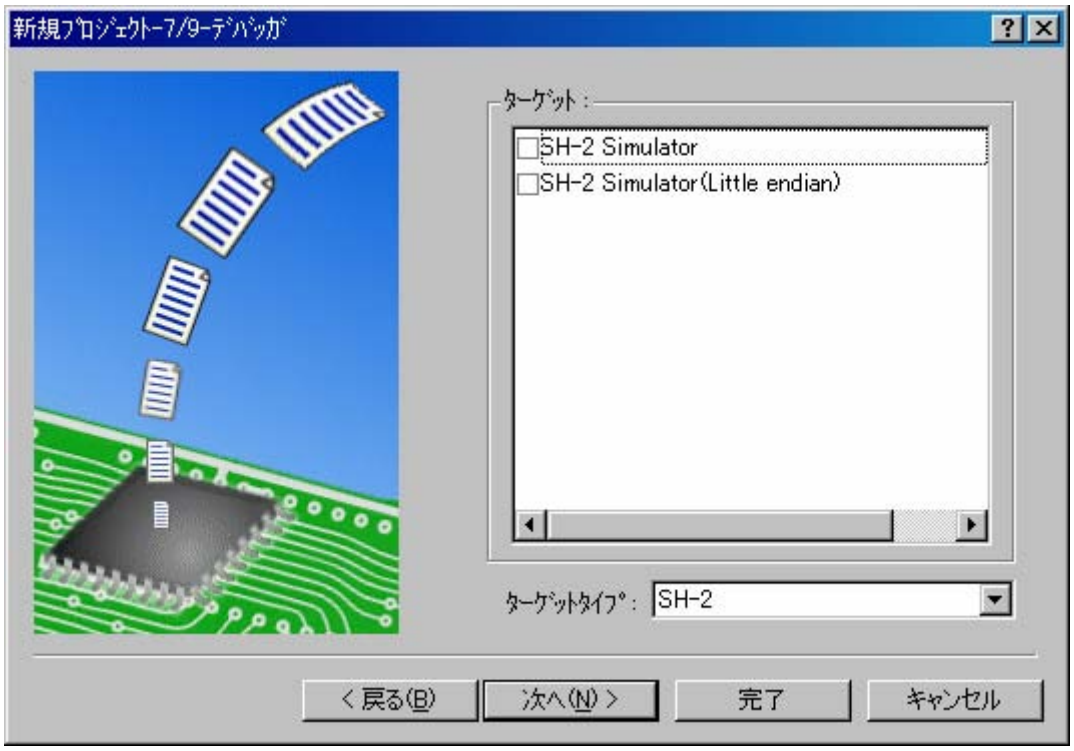

図 1-8

(9) 生成ファイル名の変更

完了を選択

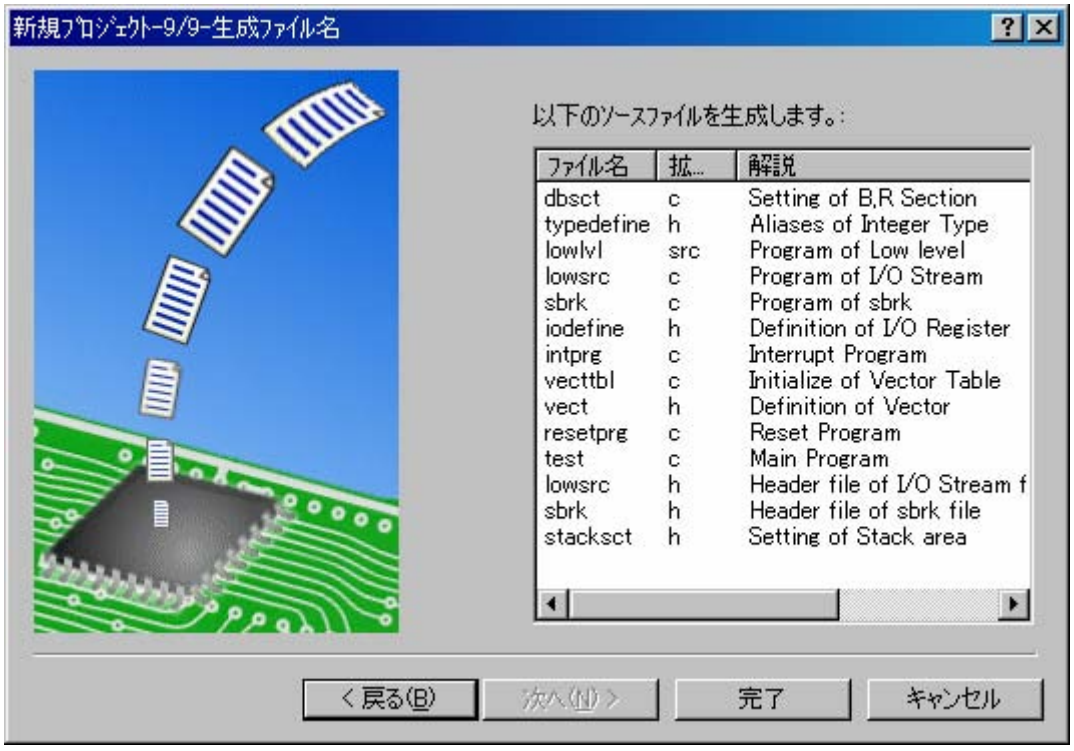

図 1-9

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)

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。**
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。**
intprg.c	[**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、**3.3.**例外処理関数** (intprg.c**、**vect.h)**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_PC)**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、**2.3.**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。**
vecttbl.c	[**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.**リセットベクタテーブル**(vecttbl.c)**をご参照ください。**
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。**

表 1-2自動生成サンプルファイル一覧(2)

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、2.2.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.h	[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。 • 割り込み関数に #pragma interrupt を指定しています。 • (7) [ベクタテーブル定義]の指定で生成されます。 詳しくは、3.3.例外処理関数 (intprg.c、vect.h)をご参照ください。

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 リセットベクタテーブル(vecttbl.c)

パワーオンリセットを行うと、CPU は以下の動作を行います。

1. プログラムカウンタ（PC）の初期値（実行開始アドレス）を例外処理ベクタテーブルから取り出す。
2. スタックポインタ（SP）の初期値を例外処理ベクタテーブルから取り出す。
3. ベクタベースレジスタ（VBR）を H'00000000 にクリアし、
   ステータスレジスタ（SR）の割り込みマスクビット（I3～I0）を H'F（B'1111）にセットする。
4. 例外処理ベクタテーブルから取り出した値をそれぞれ PC と SP に設定し、プログラムの実行を開始する。

例外処理ベクタテーブルとは、例外処理が発生した時に CPU が例外要因に応じたジャンプ先のアドレス情報を取得するデータテーブルの事です。リセット例外処理では、プログラムカウンタ（PC）とスタックポインタ（SP）の初期値を表 2-1 のアドレスから取得します。 そのため、これらの設定値はリセット前にあらかじめ設定されている必要があります。

表 2-1 例外処理ベクタテーブル（リセット要因）

例外要因		ベクタ番号	ベクタテーブルアドレス
パワーオンリセット	PC	0	H'00000000 ~ H'00000003
	SP	1	H'00000004 ~ H'00000007
マニュアルリセット	PC	2	H'00000008 ~ H'0000000B
	SP	3	H'0000000C ~ H'0000000F

サンプルプログラムでは、リセット要因用の例外処理ベクタテーブルとその他の例外処理用の例外処理ベクタテーブルを分けています。リセット要因用の例外処理ベクタテーブルはvecttbl.cに “RESET_Vectors” として定義（リスト 2-1）されています。

```
#pragma section VECTTBL          ...(a)

void *RESET_Vectors[] = {
//;<<VECTOR DATA START (POWER ON RESET)>>
//;0 Power On Reset PC
PowerON_Reset_PC,                ...(b)
//;<<VECTOR DATA END (POWER ON RESET)>>
// 1 Power On Reset SP
    __secend("S"),               ...(c)
//;<<VECTOR DATA START (MANUAL RESET)>>
//;2 Manual Reset PC
Manual_Reset_PC                  ...(d)
//;<<VECTOR DATA END (MANUAL RESET)>>
// 3 Manual Reset SP
    __secend("S")                ...(e)
};
```

リスト 2-1

[リスト 2-1 解説]

#pragma section VECTTBL 指定により、DVECTTBLセクションに配列：RESET_Vectors が配置されます。(a)
配列の第１要素には、パワーオンリセット関数（PowerON_Reset_PC）のアドレスを設定しています。(b)
第２要素には、Ｓセクションの終了アドレスを設定しています。(c)
第３要素には、マニュアルリセット関数（Manual_Reset_PC）のアドレスを設定しています。(d)
第４要素には、Ｓセクションの終了アドレスを設定しています。(e)
__secend はセクションアドレス演算子（関数ではありません）と呼ばれ、" "で囲まれたセクションの（終了アドレス+1）を取得できます。

配列：RESET_Vectors を 0 番地に配置することで、リセット要因用の例外処理ベクタテーブルとなります。
この配列を 0 番地へ配置するためには、DVECTTBLセクションをリンカのセクション設定にて、0 番地に配置する必要があります。サンプルプロジェクトでは、図 2-1の様に設定しています。

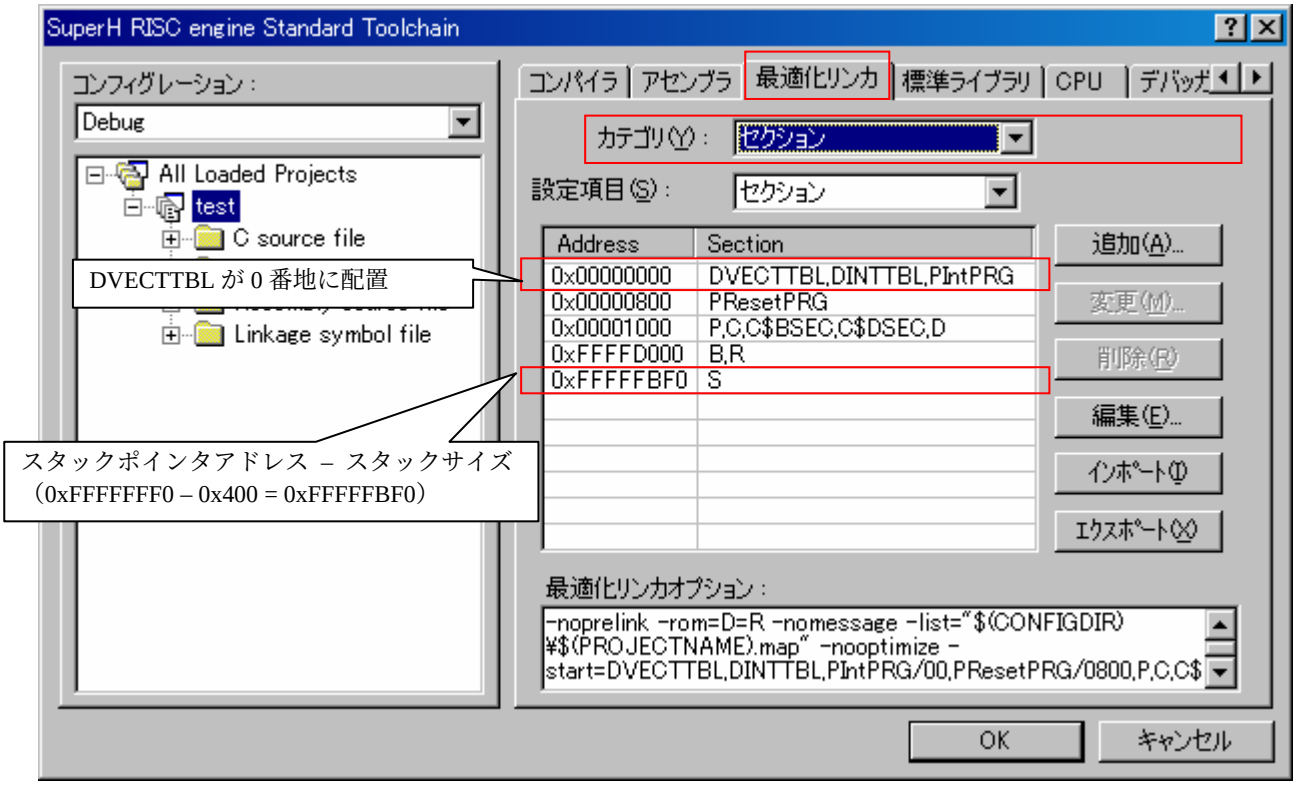
SuperH RISC engine Standard Toolchain
コンフィグレーション :
Debug
All Loaded Projects
test
C source file
Linkage symbol file
コンパイラ
アセンブラ
最適化リンカ
標準ライブラリ
CPU
デバッグ
カテゴリ(Y) :
セクション
設定項目(S) :
セクション
Address
Section
0x00000000
DVECTTBL,DINTTBL,PIntPRG
0x00000800
PResetPRG
0x00001000
P,C,C$BSEC,C$DSEC,D
0xFFFFD000
B,R
0xFFFFFBF0
S
追加(A)...
変更(M)...
削除(R)
編集(E)...
インポート(I)
エクスポート(X)
最適化リンカオプション :
-noprelink -rom=D=R -nomessage -list="$(CONFIGDIR)¥$(PROJECTNAME).map" -nooptimize -start=DVECTTBL,DINTTBL,PIntPRG/00,PResetPRG/0800,P,C,C$
OK
キャンセル
DVECTTBL が 0 番地に配置
スタックポインタアドレス – スタックサイズ
（0xFFFFFFF0 – 0x400 = 0xFFFFFBF0）

図 2-1

[リンカ最適化使用時の注意事項]

リンカの未参照シンボル削除の最適化を使用している場合は、ベクタテーブル(RESET_Vectors 、 INT_Vectors) までも適化によって削除されてしまいます。ベクタテーブルの削除を抑止するため、図 2-2の様に最適化リンカの “未参照シンボル削除抑止シンボル” で、ベクタテーブルのシンボルを指定する必要があります。

なお、シンボルを指定する時に、C/C++変数名、C 関数名はプログラム中での定義名先頭に “_” を付加する必要があります。また、C++関数の場合は引数列を含めたプログラム中の定義名をダブルクォーテーションで囲んで指定する必要があります(但し引数が void の場合は、"関数名()"で指定します)。

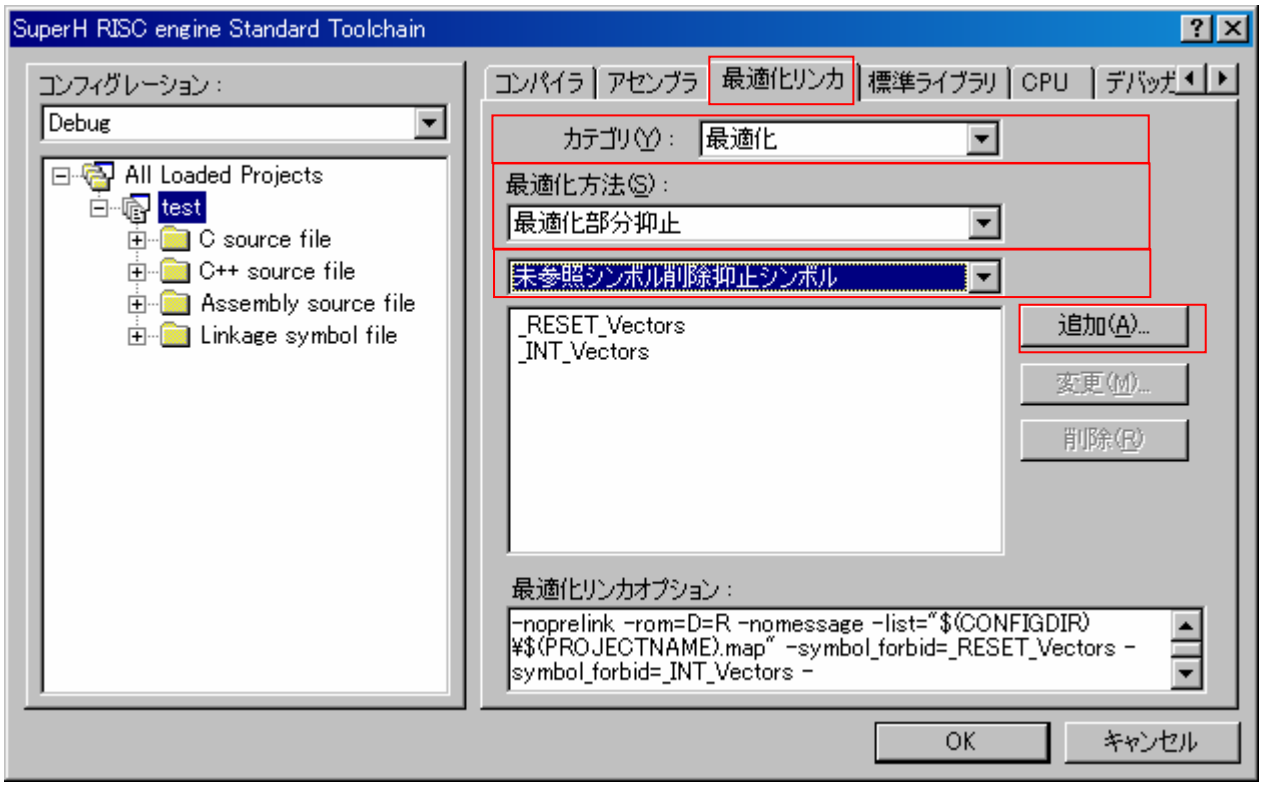

図 2-2

2.2 スタックサイズの設定(stacksct.h)

stacksct.h（リスト 2-2）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 0xFFFFFFF0 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 0xFFFFFBF0（0xFFFFFFF0 – 0x400）としています(図 2-1)。

```
#pragma stacksize 0x400
```

リスト 2-2

2.3 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、リセット関数 PowerON_Reset_PC 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 0x10`	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 _UINT INT_Vectors;`	INT_Vectors を参照します。

`#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_PC(void);`	PowerON_Reset_PC の原型宣言を行っています。
`void Manual_Reset_PC(void);`	Manual_Reset_PC 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`	リセット関数を PResetPRG セクションに配置します。
`#pragma entry PowerON_Reset_PC`	リセット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。
`void PowerON_Reset_PC(void)` `{`	
`set_vbr((void *)((_UBYTE *)& INT_Vectors - INT_OFFSET));`	ベクタベースレジスタ（VBR）設定処理を行います。 本レジスタを設定することで、任意のアドレスにリセット以外の例外処理ベクタテーブルを設定することができます。 詳しくは、3.2.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_PC から抜け出さない様になっています。

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

3. リセット以外の例外処理

リセット以外の例外要因には、アドレスエラー、割り込み、命令による例外があります。例外が発生した時は (VBR + ベクタテーブルアドレスオフセット) のアドレスから例外処理の開始アドレスを取り出し、例外処理を実行します。

3.1 リセット以外の例外処理ベクタテーブル(vecttbl.c)

例外処理ベクタテーブルには例外処理の開始アドレスを設定する必要があります。各例外要因には、それぞれ異なるベクタ番号とベクタテーブルアドレスオフセット(= VBRからのオフセット値) が割り当てられています。ベクタ番号とベクタテーブルアドレスオフセットを表 3-1に示します。

表 3-1 例外処理ベクタテーブル

例外要因		ベクタ番号	ベクタテーブルアドレスオフセット
パワーオンリセット	PC	0	H'00000000 ~ H'00000003
	SP	1	H'00000004 ~ H'00000007
マニュアルリセット	PC	2	H'00000008 ~ H'0000000B
	SP	3	H'0000000C ~ H'0000000F
一般不当命令		4	H'00000010 ~ H'00000013
（システム予約）		5	H'00000014 ~ H'00000017
スロット不当命令		6	H'00000018 ~ H'0000001B
（システム予約）		7	H'0000001C ~ H'0000001F
		8	H'00000020 ~ H'00000023
CPU **アドレスエラー**		9	H'00000024 ~ H'00000027
DTC **アドレスエラー**		10	H'00000028 ~ H'0000002B
割り込み	NMI	11	H'0000002C ~ H'0000002F
	ユーザブレーク	12	H'00000030 ~ H'00000033
（システム予約）		13	H'00000034 ~ H'00000037
		14	H'00000038 ~ H'0000003B
		15	H'0000003C ~ H'0000003F
		⋮	⋮
		31	H'0000007C ~ H'0000007F
トラップ命令（ユーザベクタ）		32	H'00000080 ~ H'00000083
		⋮	⋮
		63	H'000000FC ~ H'000000FF
割り込み	RQ0	64	H'00000100 ~ H'00000103
	IRQ1	65	H'00000104 ~ H'00000107
	IRQ2	66	H'00000108 ~ H'0000010B
	IRQ3	67	H'0000010C ~ H'0000010F
	システム予約	68	H'00000110 ~ H'00000113
	システム予約	69	H'00000114 ~ H'00000117
	システム予約	70	H'00000118 ~ H'0000011B
	システム予約	71	H'0000011C ~ H'0000011F
内蔵周辺モジュール		72	H'00000120 ~ H'00000123
		⋮	⋮
		255	H'000003FC ~ H'000003FF

サンプルプログラムでは、リセット以外の例外要因用の例外処理ベクタテーブルを任意のアドレスに割り付けられるようにするため、リセット要因用の例外処理ベクタテーブルとその他の例外処理用の例外処理ベクタテーブルを分けて定義しています。その他の例外処理用の例外処理ベクタテーブルはvecttbl.cに“INT_Vectors”として定義されています（図3-1）。

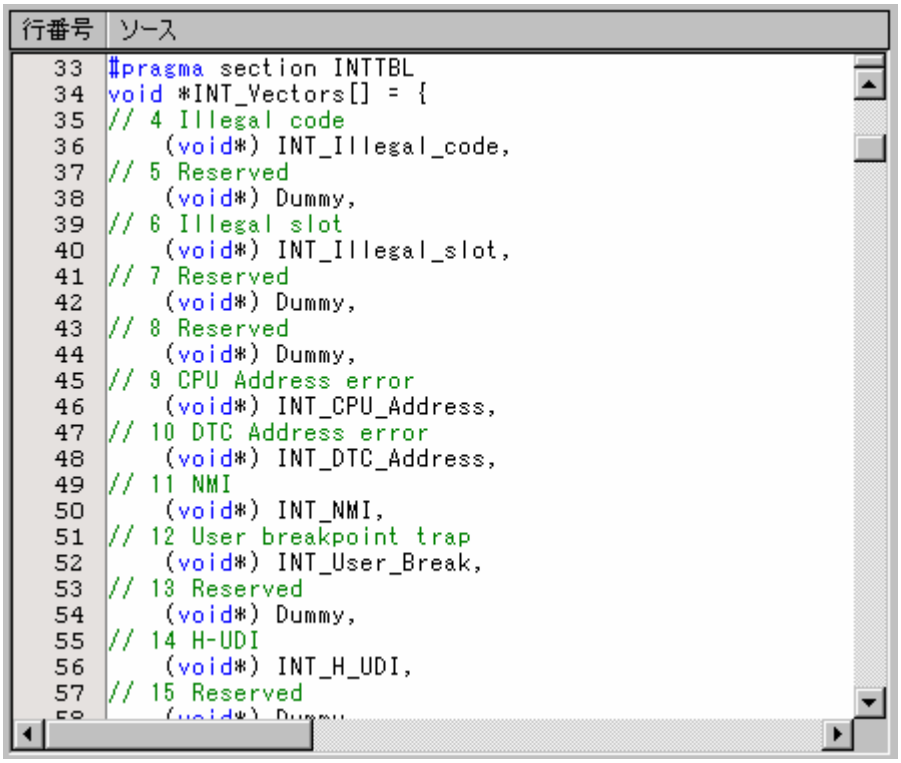

図 3-1

3.2 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ベクタテーブル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、“INT_Vectors”の配置アドレスから VBR の設定値を算出しています。“INT_Vectors” は#pragma section 指定により、DINTTBL セクションに指定されていますので、DINTTBL を任意のセクションに配置することで、“INT_Vectors” を任意のアドレスに配置することができます。

resetprg.c

```
#define INT_OFFSET 0x10
            . . .
set_vbr((void *)((_UBYTE *)&INT_Vectors - INT_OFFSET));
```

リスト 3-1

[VBR 設定値の算出方法]

リセット以外のベクタテーブル（INT_Vectors）は一般不当命令から始まっています。

そのため、“INT_Vectors” のアドレスから一般不当命令のオフセット値（INT_OFFSET(= 0x00000010)）を引くことで VBR の設定値を求めることができます。

3.3 例外処理関数 (intprg.c、vect.h)

リセット以外の例外処理（INT_Illegal_code関数、INT_Illegal_slot関数など）はintprg.cにダミー関数が定義されています（図 3-2）。

```
行番号 ソース
  16 #pragma section IntPRG
  17 // 4 Illegal code
  18 void INT_Illegal_code(void){/* sleep(); */}
  19 // 5 Reserved
  20
  21 // 6 Illegal slot
  22 void INT_Illegal_slot(void){/* sleep(); */}
  23 // 7 Reserved
  24
  25 // 8 Reserved
  26
  27 // 9 CPU Address error
  28 void INT_CPU_Address(void){/* sleep(); */}
  29 // 10 DTC Address error
  30 void INT_DTC_Address(void){/* sleep(); */}
  31 // 11 NMI
  32 void INT_NMI(void){/* sleep(); */}
```

図 3-2

これらの、リセット以外の例外処理の関数はvect.h（図 3-3）にて#pragma interrupt指定されています。#pragma interrupt指定が行われた関数は、コンパイラにより自動的に割り込み関数としてコード生成されます。 割り込み関数では、RTE命令による割り込みからの復帰、必要なレジスタの退避回復処理等が行われます。

```
行番号 ソース
  30 // 4 Illegal code
  31 #pragma interrupt INT_Illegal_code
  32 extern void INT_Illegal_code(void);
  33
  34 // 5 Reserved
  35
  36 // 6 Illegal slot
  37 #pragma interrupt INT_Illegal_slot
  38 extern void INT_Illegal_slot(void);
  39
  40 // 7 Reserved
  41
  42 // 8 Reserved
  43
  44 // 9 CPU Address error
  45 #pragma interrupt INT_CPU_Address
  46 extern void INT_CPU_Address(void);
```

図 3-3

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。

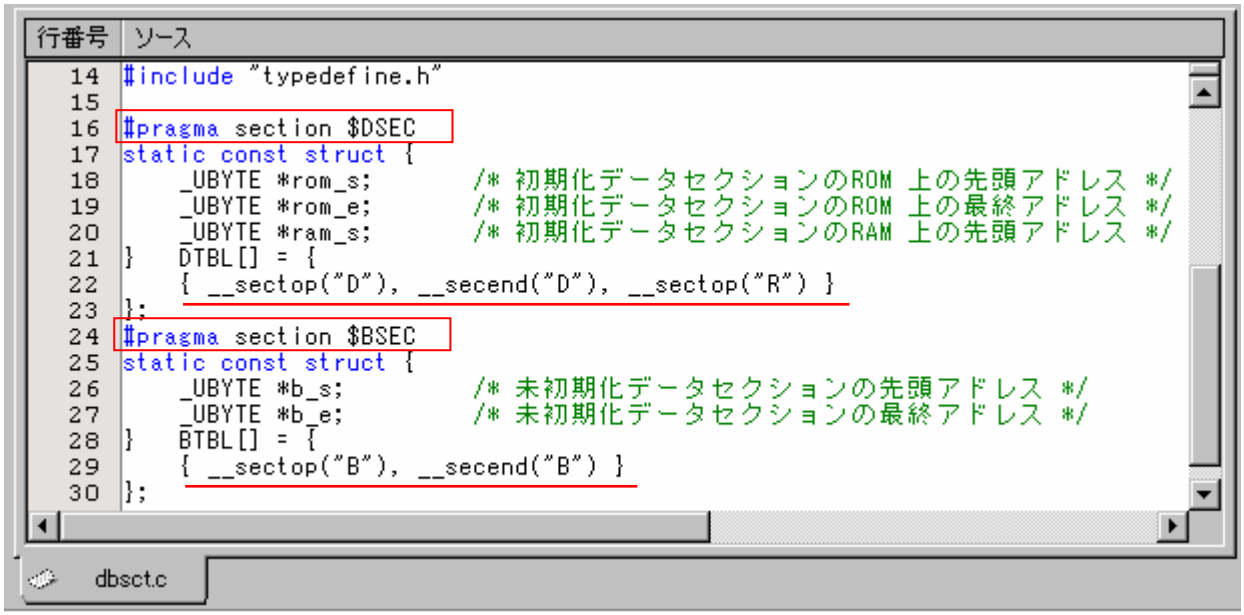

図 4-1

[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

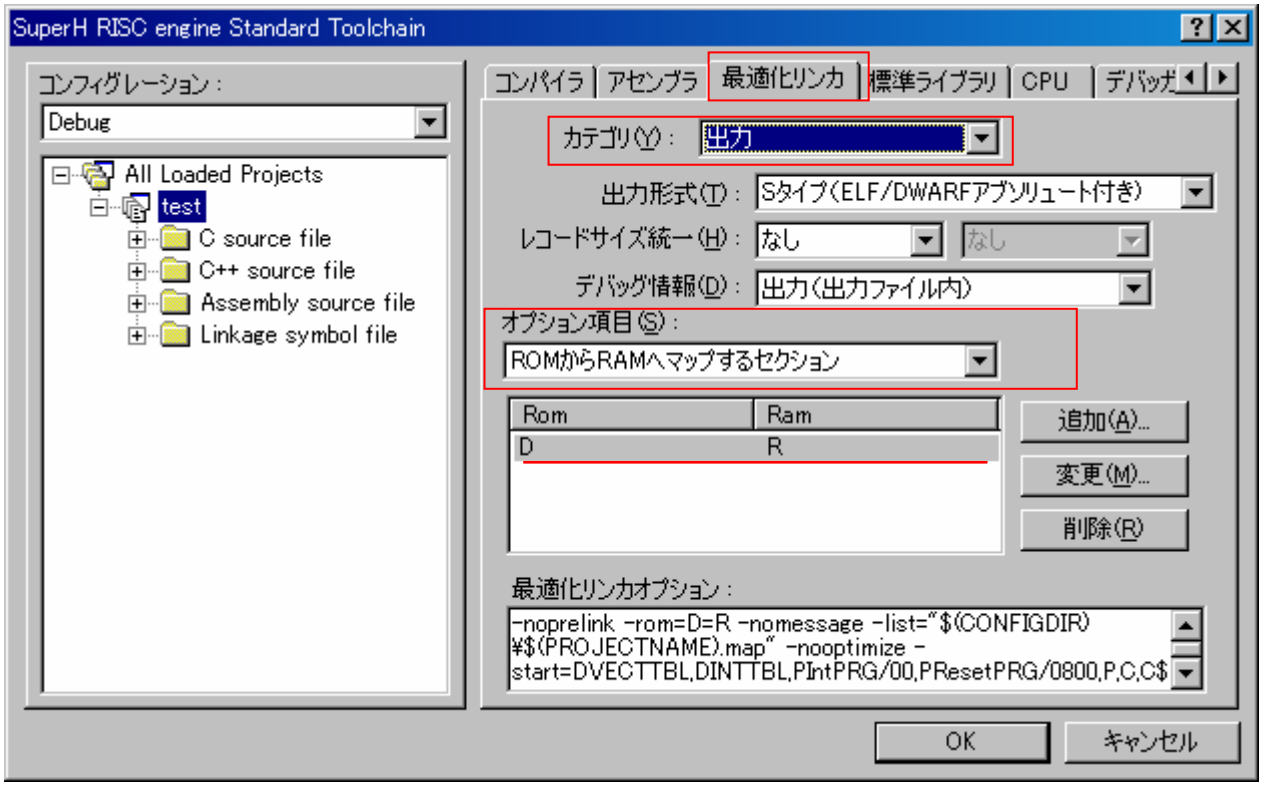

図 4-2

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. よくあるお問い合わせ

7.1 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。

7.2 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++関数を呼び出すときは、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。