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。

Application Note

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

ルネサスマイクロコンピュータ開発環境システム

ルネサス エレクトロニクス
www.renesas.com

Rev.7.00　2007.02

本資料ご利用に際しての留意事項

１．　本資料は、お客様に用途に応じた適切な弊社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弊社または第三者の知的財産権その他の権利の実施、使用を許諾または保証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など全ての情報の使用に起因する損害、第三者の知的財産権その他の権利に対する侵害に関し、弊社は責任を負いません。

３．　本資料に記載の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、あるいはその他軍事用途の目的で使用しないでください。また、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、それらの定めるところにより必要な手続を行ってください。

４．　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などの全ての情報は本資料発行時点のものであり、弊社は本資料に記載した製品または仕様等を予告なしに変更することがあります。弊社の半導体製品のご購入およびご使用に当たりましては、事前に弊社営業窓口で最新の情報をご確認頂きますとともに、弊社ホームページ(http://www.renesas.com)などを通じて公開される情報に常にご注意下さい。

５．　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、弊社はその責任を負いません。

６．　本資料に記載の製品データ、図、表などに示す技術的な内容、プログラム、アルゴリズムその他応用回路例などの情報を流用する場合は、流用する情報を単独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下さい。弊社は、適用可否に対する責任は負いません。

７．　本資料に記載された製品は、各種安全装置や運輸・交通用、医療用、燃焼制御用、航空宇宙用、原子力、海底中継用の機器・システムなど、その故障や誤動作が直接人命を脅かしあるいは人体に危害を及ぼすおそれのあるような機器・システムや特に高度な品質・信頼性が要求される機器・システムでの使用を意図して設計、製造されたものではありません（弊社が自動車用と指定する製品を自動車に使用する場合を除きます）。これらの用途に利用されることをご検討の際には、必ず事前に弊社営業窓口へご照会下さい。なお、上記用途に使用されたことにより発生した損害等について弊社はその責任を負いかねますのでご了承願います。

８．　第７項にかかわらず、本資料に記載された製品は、下記の用途には使用しないで下さい。これらの用途に使用されたことにより発生した損害等につきましては、弊社は一切の責任を負いません。
　１）生命維持装置。
　２）人体に埋め込み使用するもの。
　３）治療行為（患部切り出し、薬剤投与等）を行なうもの。
　４）その他、直接人命に影響を与えるもの。

９．　本資料に記載された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、実装条件およびその他諸条件につきましては、弊社保証範囲内でご使用ください。弊社保証値を越えて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、弊社はその責任を負いません。

１０．弊社は製品の品質および信頼性の向上に努めておりますが、特に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。弊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、誤動作防止設計などの安全設計（含むハードウエアおよびソフトウエア）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エアは、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致します。

１１．本資料に記載の製品は、これを搭載した製品から剥がれた場合、幼児が口に入れて誤飲する等の事故の危険性があります。お客様の製品への実装後に容易に本製品が剥がれることがなきよう、お客様の責任において十分な安全設計をお願いします。お客様の製品から剥がれた場合の事故につきましては、弊社はその責任を負いません。

１２．本資料の全部または一部を弊社の文書による事前の承諾なしに転載または複製することを固くお断り致します。

１３．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弊社営業窓口までご照会下さい。

はじめに

ルネサステクノロジ SuperH RISC engine ファミリは、高性能の演算処理を実現すると共に各種周辺機器を内蔵し、低消費電力で動作する機器組み込み用新世代シングルチップ RISC マイコンです。

本アプリケーションノートでは、ルネサステクノロジ SuperH RISC engine ファミリのこれらの機能・性能を活かした応用プログラムを「SuperH RISC engine C/C++コンパイラパッケージ V.9.00」を用いて効果的に作成する方法を説明します。

C/C++コンパイラの詳細な仕様については、「**SuperH RISC engine C/C++コンパイラ、アセンブラ、最適化リンケージエディタ ユーザーズマニュアル**」を参照してください。

〈アプリケーションノートの構成〉

本アプリケーションノートは、以下に示す 10 章と付録から構成されています。

第 1 章では、概説として、インストール方法や、プログラミング開発フローを説明します。

第 2 章では、サンプルを用いて、一通りデバッグまで行います。また、C 言語を用いたプログラム作成方法を説明しています。

第 3 章では、C 言語プログラムとアセンブリ言語プログラムを接続するとき、および C/C++コンパイラが生成したオブジェクトファイルに対してクロスソフトを使用するときの注意事項ならびに、SuperH RISC engine C/C++コンパイラの拡張機能と機器組み込み用ソフト特有の手法を説明しています。

第 4 章では、HEW のオプションを説明します。

第 5 章、第 6 章では、ルネサステクノロジ SuperH RISC engine ファミリの性能を活かすための C 言語プログラム作成方法を説明しています。

第 7 章では、HEW を用いた活用法を説明いたします。

第 8 章では、効率の良い C++プログラミング技法を示します。

第 9 章では、リンク時における便利なオプションの説明、またリンク時にモジュール間をまたいで横断的に最適化する最適化機能について説明いたします。

第 10 章では、ユーザから多く寄せられた質問についての回答を記載しています。

付録では、SuperH RISC engine C/C++コンパイラの各バージョンでの変更点を記載しています。

〈関連マニュアル〉

関連マニュアルは以下のとおりです。

- ルネサステクノロジ SuperH RISC engine ファミリ、各マイコン別ハードウェアマニュアル
- SuperH RISC engine High-performance Enbedded Worksyop3 ユーザーズマニュアル
- SuperH RISC engine　C/C++コンパイラ、アセンブラ、最適化リンケージエディタユーザーズマニュアル
- SuperH RISC engine High-performance Embedded Workshop3 チュートリアル

〈使用クロスソフトのバージョン〉

SuperH RISC engine C/C++コンパイラ V.9.00 を用いるためには、次の表で示すバージョンのクロスソフトを使用してください。

クロスソフト名	バージョン
SH シリーズ　クロスアセンブラ	7.00
H シリーズ　最適化リンケージエディタ	9.00
SH シリーズ　ライブラリ・ジェネレータ	3.00

〈本アプリケーションノートで使用する記号などの意味〉

［］	：省略できることを示します。
(RET)	：リターンキーの入力を示します。
△	：１つ以上の空白またはタブコードを示します。
abc	：太字の部分はユーザがキー入力する部分を示します。
〈〉	：この記号で囲まれた内容を指定することを示します。
…	：直前の項目を１回以上指定することを示します。
H'	：整数定数の先頭に" H' "がついているのは16進数です。
0x	：整数定数の先頭に" 0x "がついているのは16進数です。

UNIX は、X/Open カンパニーリミテッドがライセンスしている米国ならびに他の国における登録商標です。
MS-DOS は米国マイクロソフト社により管理されているオペレーティングシステムの名称です。
Microsoft® WindowsNT® operating system, Microsoft®,Windows®98 and Windows 2000 operating system, Microsoft® WindowsMe® operating system, Microsoft® WindowsXp® operating system は、米国 Microsoft Corporation の米国およびその他の国における登録商標です。
IBM PC は、米国 International Business Machines Corporation の登録商標です。

本アプリケーションノートは下記のように読まれることをおすすめします。

項番	状況	本アプリケーションノートの使用方法
1	SuperH RISC engine C/C++コンパイラを初めて使う。 (1) ロードモジュールを作成するためのコンパイラ、およびクロスソフトの使用方法を知りたい。 (2) SuperH RISC engine ファミリで動作するプログラムを作りたい。	(1) 「1.4 起動方法」でコンパイラの起動方法がわかります。「1.5 プログラム開発手順」にロードモジュール完成までに必要なクロスソフトの操作方法が説明されています。 (2) 「2.2～2.3 サンプルプログラムの紹介」にプログラムがあります。 これは機器組み込み用に必要な最低限のコンパイラ機能を説明するためのプログラムです。これを参考に簡単なプログラムを作り、シミュレータ・デバッガなどで動作を確認してください。コンパイラの他の機能は「3. コンパイラ」にあります。ロードモジュール作成がうまくいかない場合、「3.15 クロスソフト間の関連」も参照してください。
2	機器組み込み用プログラムを作成する。 (1) 他のマイコン用プログラムがあり、これを移植する。 (2) 新しく作成する。	(1) 「2.2～2.3 サンプルプログラムの紹介」と「3. コンパイラ」を読み、利用できる機能を見つけ出して、アセンブリ言語部分をC言語に書き換えられないかを検討してください。アセンブリ言語プログラムとの接続は「3.15.1 アセンブリ言語プログラムとの関連」を参考にしてください。 (2) まず「2.2～2.3 サンプルプログラムの紹介」を読み、プログラム作成の概略を理解してください。次に「3. コンパイラ」に進み、SuperH RISC engine C/C++コンパイラの拡張機能を習得してください。プログラム作成の際に「5. 効率の良いプログラミング技法」を参考にして、最初から効果的なプログラムになるようにしてください。
3	実行速度の向上、またはサイズの縮小を行う。	「5. 効率の良いプログラミング技法」を参考にして性能向上を行ってください。
4	プログラムがうまく動作しない。	関連項目の後ろにある「注意事項」、または「11. Q&A」の各項目の中に、該当するものがないか探してください。

目次

1. 概説 1-1

1.1 概要 1-1

1.2 特長 1-1

1.3 インストール方法 1-2

1.3.1 PC 版 1-2
1.3.2 UNIX 版 1-5

1.4 起動方法 1-10

1.4.1 統合化環境からの起動方法 1-10
1.4.2 コンパイラの起動方法 1-11

1.5 プログラム開発手順 1-13

2. プログラムの作成とデバッグまでの手順 2-1

2.1 プロジェクトの構築 2-1

2.1.1 シミュレータデバッガ用プロジェクトの作成 2-1

2.2 サンプルプログラムの紹介 (SH-1,SH-2,SH-2E,SH-2A,SH2A-FPU,SH2-DSP) 2-8

2.2.1 ベクタテーブルの作成 2-9
2.2.2 ヘッダファイルの作成 2-11
2.2.3 メイン処理部の作成 2-14
2.2.4 初期化部の作成 2-15
2.2.5 割り込み関数の作成 2-17
2.2.6 ロードモジュール用バッチファイルの作成 2-18
2.2.7 リンケージエディタのサブコマンドファイルの作成 2-18

2.3 サンプルプログラムの紹介(SH-3,SH3-DSP,SH-4,SH-4A,SH4AL-DSP) 2-19

2.3.1 割り込みハンドラの作成 2-19
2.3.2 ベクタテーブルの作成 2-24
2.3.3 ヘッダファイルの作成 2-28
2.3.4 初期化部の作成 2-29
2.3.5 メイン処理部、割り込み処理部の作成 2-31
2.3.6 ロードモジュール用バッチファイルの作成 2-31
2.3.7 リンケージエディタのサブコマンドファイルの作成 2-32

2.4 シミュレータデバッガを使用したデバッグ方法 2-33

2.4.1 コンフィグレーションの設定 2-33
2.4.2 メモリリソースの確保 2-34
2.4.3 サンプルプログラムのダウンロード 2-35
2.4.4 I/O シミュレーション の設定 2-36
2.4.5 トレース情報取得条件の設定 2-37
2.4.6 ステイタスウィンドウ 2-37
2.4.7 レジスタウィンドウ 2-38
2.4.8 トレース 2-38
2.4.9 ブレークポイントの確認 2-40
2.4.10 メモリ内容の確認 2-41

2.5 シミュレータデバッガでの標準入出力およびファイル入出力処理 2-42

3. コンパイラ 3-1

3.1 割り込み関数 3-1

3.1.1 割り込み関数の定義（オプションなし） 3-1
3.1.2 割り込み関数の定義（オプションあり） 3-6
3.1.3 ベクタテーブルの作成 3-9

3.2 組み込み関数 3-11
3.2.1 ステータスレジスタの設定／参照 3-16
3.2.2 ベクタベースレジスタの設定／参照 3-17
3.2.3 I/O レジスタへのアクセス（1） 3-18
3.2.4 I/O レジスタへのアクセス（2） 3-20
3.2.5 システム制御 3-21
3.2.6 積和演算（1） 3-22
3.2.7 積和演算（2） 3-24
3.2.8 64bit 乗算（1） 3-26
3.2.9 64bit 乗算（2） 3-27
3.2.10 データ上位・下位交換 3-28
3.2.11 システムコール 3-29
3.2.12 プリフェッチ命令 3-31
3.2.13 LDTLB 命令 3-32
3.2.14 NOP 命令 3-33
3.2.15 単精度浮動小数点演算 3-34
3.2.16 拡張レジスタのアクセス 3-40
3.2.17 DSP 命令 3-41
3.2.18 正弦・余弦 3-43
3.2.19 平方根の逆数 3-44
3.2.20 命令キャッシュ無効化 3-45
3.2.21 キャッシュブロック操作 3-46
3.2.22 命令キャッシュプリフェッチ 3-47
3.2.23 システム同期 3-48
3.2.24 T ビット参照・設定 3-49
3.2.25 連結レジスタの中央切り出し 3-50
3.2.26 キャリ付き加算 3-51
3.2.27 ボロー付き減算 3-52
3.2.28 符号反転 3-53
3.2.29 1 ビット除算 3-54
3.2.30 回転 3-56
3.2.31 シフト 3-57
3.2.32 飽和演算 3-58
3.2.33 TBR 設定・参照 3-59
3.3 インライン展開 3-60
3.3.1 関数のインライン展開 3-60
3.3.2 アセンブラ埋め込みインライン展開の記述方法 3-62
3.3.3 インラインアセンブラ関数サンプルプログラム 3-64
3.4 レジスタ指定 3-81
3.4.1 GBR ベース変数の指定 3-82
3.4.2 グローバル変数のレジスタ割り付け 3-84
3.5 レジスタ退避／回復の制御 3-86
3.6 16/20/28/32 ビットアドレス領域指定 3-89
3.7 セクション名指定 3-92
3.7.1 セクション名指定 3-92
3.7.2 セクション切り替え 3-93
3.8 エントリ関数の指定、SP の設定 3-94
3.9 ポジションインディペンデントコード 3-95
3.10 MAP 最適化 3-96
3.10.1 使用方法 3-96
3.10.2 外部変数アクセスコード改善例（1） 3-97
3.10.3 外部変数アクセスコード改善例（2） 3-97
3.10.4 外部変数アクセスコード改善例（3） 3-97
3.10.5 外部変数アクセスコード改善例（4） 3-98

3.11 オプション.......3-99
3.11.1 コード生成に関するオプション.......3-99
3.11.2 最適化リンケージに関するオプション.......3-100
3.11.3 標準ライブラリ構築に関するオプション.......3-101
3.12 SH-DSP の特長.......3-102
3.13 DSP ライブラリ.......3-106
3.13.1 概要.......3-106
3.13.2 データフォーマット.......3-107
3.13.3 効率.......3-108
3.13.4 高速フーリエ変換.......3-108
3.13.5 窓関数.......3-128
3.13.6 フィルタ.......3-131
3.13.7 畳み込みと相関.......3-150
3.13.8 その他.......3-157
3.14 DSP ライブラリの性能について.......3-175
3.15 クロスソフト間の関連.......3-180
3.15.1 アセンブリ言語プログラムとの関連.......3-180
3.15.2 最適化リンケージエディタとの関連.......3-191
3.15.3 シミュレータ・デバッガとの関連.......3-193
3.16 構造体の境界調整数の変更.......3-204
3.17 long long 型.......3-206
3.18 DSP-C 仕様.......3-207
3.18.1 固定小数点データ型.......3-207
3.18.2 メモリ修飾子.......3-209
3.18.3 飽和修飾子.......3-212
3.18.4 循環修飾子.......3-213
3.18.5 型変換.......3-214
3.18.6 算術変換.......3-216
3.19 MAP 最適化拡張オプション.......3-217
3.19.1 使用方法.......3-217
3.19.2 外部変数アクセスコード改善例(1).......3-217
3.19.3 外部変数アクセスコード改善例(2).......3-217
3.20 TBR 相対関数呼び出し.......3-218
3.21 GBR 相対論理演算命令生成.......3-223
3.22 register 宣言有効化.......3-225
3.23 変数の絶対アドレス指定.......3-227
3.24 最適化強化.......3-228
3.24.1 リテラルデータ改善（1）.......3-228
3.24.2 リテラルデータ改善（2）.......3-228
3.24.3 EXTU 抑止（1）.......3-229
3.24.4 EXTU 抑止（2）.......3-229
3.24.5 ビット演算改善（1）.......3-229
3.24.6 ビット演算改善（2）.......3-230
3.24.7 ビット演算改善（3）.......3-230
3.24.8 ビット演算改善（4）.......3-231
3.24.9 ビット演算改善（5）.......3-231
3.25 未初期化変数の出力順制御.......3-232
3.26 変数の配置指定.......3-233

4. HEW.......4-1

4.1 HEW2.0 以降のオプション指定方法.......4-1
4.1.1 C/C++コンパイラのオプション.......4-2
4.1.2 アセンブラのオプション.......4-13

4.1.3 最適化リンケージエディタのオプション ... 4-18
4.1.4 標準ライブラリ構築ツールのオプション ... 4-28
4.1.5 CPU オプション ... 4-36
4.2 統合化環境からのコンパイラバージョンの指定 ... 4-37

5. 効率の良いプログラミング技法 ... 5-1

5.1 データ指定 ... 5-3
5.1.1 局所変数（データサイズ） ... 5-4
5.1.2 大域変数（符号） ... 5-5
5.1.3 データサイズ（乗算） ... 5-6
5.1.4 データの構造化 ... 5-7
5.1.5 データの整合 ... 5-8
5.1.6 初期値と const 型 ... 5-9
5.1.7 局所変数と大域変数 ... 5-10
5.1.8 ポインタ変数の活用 ... 5-11
5.1.9 定数参照(1) ... 5-12
5.1.10 定数参照(2) ... 5-13
5.1.11 一定値になる変数(1) ... 5-14
5.1.12 一定値になる変数(2) ... 5-15
5.2 関数呼び出し ... 5-16
5.2.1 関数のモジュール化 ... 5-17
5.2.2 ポインタ変数による関数呼び出し ... 5-18
5.2.3 関数のインタフェース ... 5-20
5.2.4 テイルリカージョン ... 5-21
5.2.5 FSQRT,FABS 命令活用 ... 5-22
5.3 演算方法 ... 5-23
5.3.1 ループ内不変式の移動 ... 5-24
5.3.2 ループ回数の削減 ... 5-25
5.3.3 乗算／除算の使用 ... 5-26
5.3.4 公式の適用 ... 5-27
5.3.5 テーブルの活用 ... 5-28
5.3.6 条件式 ... 5-30
5.3.7 ロードストア削除 ... 5-31
5.4 分岐 ... 5-34
5.5 インライン展開 ... 5-36
5.5.1 関数のインライン展開 ... 5-37
5.5.2 アセンブラ埋め込みのインライン展開 ... 5-39
5.6 グローバルベースレジスタ（GBR） ... 5-41
5.6.1 グローバルベースレジスタ（GBR）を使ったオフセット参照 ... 5-41
5.6.2 グローバルベースレジスタ（GBR）領域の使い分け ... 5-43
5.7 レジスタ退避／回復の制御 ... 5-45
5.8 2 バイトアドレスの指定 ... 5-49
5.9 キャッシュの利用 ... 5-50
5.9.1 プリフェッチ命令 ... 5-50
5.9.2 タイリング ... 5-52
5.10 マトリックス演算 ... 5-55
5.11 ソフトパイプ ... 5-57
5.12 キャッシュメモリについて ... 5-59
5.13 SuperH シリーズのキャッシュ ... 5-61
5.14 キャッシュ活用のテクニック ... 5-63

6. 効率の良いプログラミング技法 (追加) ... 6-1

6.1 オプションの指定方法 ... 6-1
6.1.1 HEW 起動時のオプション (浮動小数点の設定) ... 6-1

6.1.2 最適化オプションの指定方法 (スピードとサイズ) 6-3
6.1.3 プログラムの互換性に注意が必要なオプション (関数インタフェース) 6-4
6.1.4 Volatile 宣言された変数の扱いに関するオプション (volatile 変数) 6-6
6.1.5 空ループ削除の抑止 6-11
6.1.6 Const 変数最適化の抑止 6-12
6.1.7 浮動小数点の実行効率に効果があるオプション 6-13
6.2 定数除算最適化 6-15
6.3 整数除算のサイズ 6-16
6.4 レジスタ宣言 6-17
6.5 構造体宣言のメンバオフセット 6-19
6.6 ビットフィールドの割り付け 6-20
6.7 ソフトウェアパイプライン (浮動小数点のテーブルサーチ) 6-21
6.8 データのアクセスサイズの保証 6-22
6.9 浮動小数点命令の活用 6-23

7. HEW の活用 7-1

7.1 ビルド編 7-2
7.1.1 自動生成ファイルの再生成と編集 7-2
7.1.2 メイクファイルの出力 7-4
7.1.3 メイクファイルの入力 7-5
7.1.4 カスタムプロジェクトタイプの作成 7-7
7.1.5 マルチ CPU 機能 7-11
7.1.6 ネットワーク機能 7-12
7.1.7 古い HEW から転用する場合 7-16
7.1.8 HIM システムから転用する場合 7-18
7.1.9 サポート CPU の追加 7-21
7.2 シミュレーション編 7-22
7.2.1 擬似割り込み 7-22
7.2.2 ブレークポイントの便利機能 7-23
7.2.3 カバレジ機能 7-27
7.2.4 ファイル入出力 7-30
7.2.5 デバッガターゲットの同期 7-32
7.2.6 タイマ使用方法 7-35
7.2.7 タイマ使用の具体例 7-37
7.2.8 デバッガターゲットの再登録 7-40
7.3 Call Walker 編 7-41
7.3.1 スタック情報ファイルの作成方法 7-41
7.3.2 Call Walker の起動 7-42
7.3.3 ファイルのオープンと Call Walker の画面 7-43
7.3.4 スタック情報の編集 7-47
7.3.5 アセンブラプログラムのスタック使用量 7-50
7.3.6 スタック情報のマージ（連結） 7-51
7.3.7 その他の機能 7-53

8. 効率の良い C++プログラミング技法 8-1

8.1 初期処理／後処理 8-2
8.1.1 グローバルクラスオブジェクトの初期処理と後処理 8-2
8.2 C++機能紹介 8-4
8.2.1 C 言語オブジェクトの参照方法 8-4
8.2.2 new,delete の実装方法 8-5
8.2.3 スタティックメンバ変数 8-7
8.3 オプション活用法 8-9
8.3.1 組み込み向け C++言語 8-9
8.3.2 実行時型情報 8-9

8.3.3 例外処理機能 8-12
8.3.4 プレリンカの起動抑止 8-12
8.4 C++記述のメリット・デメリット 8-13
8.4.1 コンストラクタ (1) 8-14
8.4.2 コンストラクタ (2) 8-16
8.4.3 デフォルトパラメータ 8-18
8.4.4 インライン展開 8-19
8.4.5 クラスメンバ関数 8-20
8.4.6 operator 演算子 8-22
8.4.7 関数のオーバロード 8-24
8.4.8 リファレンス型 8-26
8.4.9 スタティック関数 8-27
8.4.10 スタティックメンバ変数 8-30
8.4.11 匿名 union 8-32
8.4.12 仮想関数 8-33
9. 最適化リンケージエディタ 9-1
9.1 入出力オプション 9-2
9.1.1 入力オプション 9-2
9.1.2 出力オプション 9-5
9.2 リストオプション 9-7
9.2.1 シンボル情報表示 9-7
9.2.2 シンボル参照回数表示 9-8
9.2.3 クロスリファレンス情報表示 9-9
9.3 便利な機能 9-10
9.3.1 空きエリア出力指定 9-10
9.3.2 S タイプファイルの終端コード 9-14
9.3.3 デバッグ情報の圧縮 9-14
9.3.4 リンク時間の短縮 9-15
9.3.5 参照されない定義シンボルの通知 9-16
9.3.6 セクション内データの詰め込み配置 9-17
9.4 最適化機能 9-19
9.4.1 リンク時の最適化とは？ 9-19
9.4.2 定数／文字列の統合 9-20
9.4.3 未参照シンボルの削除 9-21
9.4.4 レジスタ退避・回復の最適化 9-22
9.4.5 共通コードの統合 9-24
9.4.6 分岐命令の最適化 9-26
9.4.7 最適化部分抑止 9-28
9.4.8 最適化結果の確認 9-29
10. MISRA C 10-1
10.1 MISRA C 10-1
10.1.1 MISRA C とは 10-1
10.1.2 ルールの例 10-1
10.1.3 合致マトリクス 10-2
10.1.4 ルール違反 10-2
10.1.5 MISRA C 準拠 10-3
10.2 SQMlint 10-3
10.2.1 SQMlint とは 10-3
10.2.2 使用方法 10-5
10.2.3 検査結果の確認方法 10-5
10.2.4 開発手順 10-6
10.2.5 対応コンパイラ 10-6
10.2.6 SH C/C++コンパイラでチェック可能なルール 10-7

11. Q & A 11-1

11.1 C/C++コンパイラ、アセンブラ 11-1

11.1.1 const 宣言 11-1
11.1.2 1 ビットデータの正しい判定方法 11-2
11.1.3 インストール 11-3
11.1.4 実行時ルーチンの仕様とスピード 11-4
11.1.5 SH シリーズオブジェクト互換性 11-10
11.1.6 稼働するホストマシンと OS について 11-11
11.1.7 C/ C++ソースレベルデバッグができない 11-11
11.1.8 インライン展開時にウォーニングが出る 11-13
11.1.9 コンパイル時に Function not optimized が出る 11-14
11.1.10 コンパイル時に compiler version mismatch が出る 11-14
11.1.11 コンパイル時に memory overflow が出る 11-15
11.1.12 インクルード指定の優先順位 11-15
11.1.13 コンパイルバッチファイル 11-16
11.1.14 プログラム内への日本語記述 11-16
11.1.15 データ割り付け Endian 11-17
11.1.16 #pragma inline_asm 使用時のアセンブル 11-18
11.1.17 特権モード 11-18
11.1.18 オブジェクトの生成について 11-19
11.1.19 #pragma gbr_base 指定機能について 11-19
11.1.20 漢字コードを含むプログラムのコンパイル 11-19
11.1.21 浮動小数点演算の速度 11-20
11.1.22 PIC オプションの使用方法 11-24
11.1.23 最適化によって、コードが大幅に削除されてしまう 11-26
11.1.24 デバッグ時にローカル変数の値が見えない 11-27
11.1.25 割り込み禁止／許可マクロ 11-28
11.1.26 SH-3 以降での割り込み関数 11-28
11.1.27 SH4 の浮動小数点演算結果 11-29
11.1.28 最適化オプションについて 11-29
11.1.29 関数の引数が正しく渡されない 11-29
11.1.30 不正動作になりやすいコーディングを調べたい 11-30
11.1.31 コメントの記述について 11-31
11.1.32 アセンブラを埋め込んだ場合のビルドの仕方 11-32
11.1.33 C++ 言語仕様についての機能 11-33
11.1.34 プリプロセッサ展開後のソースが見たい 11-34
11.1.35 ICE でうまく行くが実チップ上では暴走する 11-34
11.1.36 H8 マイコン用に開発した C プログラムの利用について 11-35
11.1.37 最適化により無限ループになる 11-36
11.1.38 DSP ライブラリの注意事項 11-38
11.1.39 DSP ライブラリの最大サンプリングデータ数 11-39
11.1.40 ビットフィールドのリードライト命令 11-40
11.1.41 割り込み処理を記述したい 11-42
11.1.42 プログラムを長時間実行すると一般不当命令例外が発生することがある 11-45
11.1.43 整数演算結果が期待値と異なる 11-46

11.2 リンケージエディタ 11-47

11.2.1 リンク時に UNDEFINED SYMBOL が出る 11-47
11.2.2 リンク時に RELOCATION SIZE OVERFLOW が出る 11-47
11.2.3 リンク時に SECTION ATTRIBUTE MISMATCH が出る 11-48
11.2.4 プログラムの RAM への転送実行 11-49
11.2.5 一部のアドレス領域のシンボルアドレスを FIX してリンクしたい 11-55
11.2.6 オーバレイの実現 11-56
11.2.7 未定義シンボルのエラー出力指定 11-57
11.2.8 S タイプファイルの出力形式の統一 11-57
11.2.9 出力ファイルの分割 11-57
11.2.10 Windows2000 での optlnksh.exe の実行 11-57

11.2.11 最適化リンケージエディタが出力するファイル形式 ... 11-58
11.2.12 プログラムサイズ（ROM,RAM）サイズの算出方法 ... 11-59
11.2.13 Section alignment mismatch が出力される ... 11-60
11.3 標準ライブラリ ... 11-61
11.3.1 リエントラントと標準ライブラリ ... 11-61
11.3.2 標準ライブラリで、リエントラントライブラリを使用したい ... 11-64
11.3.3 標準ライブラリが存在しない（SHC V6 以降） ... 11-64
11.3.4 標準ライブラリ構築時のウォーニング ... 11-65
11.3.5 ヒープ領域で使用するメモリのサイズ ... 11-66
11.3.6 ライブラリファイルを編集したい ... 11-67
11.4 HEW ... 11-69
11.4.1 ダイアログメニューが正しく表示されない ... 11-69
11.4.2 オブジェクトファイルのリンク順序 ... 11-69
11.4.3 MAP 最適化の指定方法 ... 11-71
11.4.4 プロジェクトファイルの除外 ... 11-72
11.4.5 プロジェクトファイルのデフォルトオプション指定 ... 11-73
11.4.6 メモリマップの変更方法 ... 11-74
11.4.7 HEW のネットワーク上での使用について ... 11-74
11.4.8 HEW で作成するファイル、ディレクトリ名の制限 ... 11-74
11.4.9 HEW エディタ、HDI での日本語表示フォントがおかしい ... 11-75
11.4.10 HIM から HEW への変換方法 ... 11-76
11.4.11 HEW のプロジェクト構築時に当該デバイスがない ... 11-77
11.4.12 古いコンパイラ(ツールチェイン)を最新の HEW に登録したい ... 11-78
付録 ... 付録-1
付録 A. 実行時ルーチン命名規則 ... 付録-1
付録 B. 追加機能について ... 付録-2
B.1 Ver.1.0 から Ver.2.0 への追加機能 ... 付録-2
B.2 Ver.2.0 から Ver.3.0 への追加機能 ... 付録-3
B.3 Ver.3.0 から Ver.4.1 への追加機能 ... 付録-6
B.4 Ver.4.1 から Ver.5.0 への追加機能 ... 付録-9
B.5 Ver.5.0 から Ver.5.1 への追加機能 ... 付録-10
B.6 Ver.5.1 から Ver.6.0 への追加機能 ... 付録-11
B.7 Ver.6.0 から Ver.7.0 への追加機能 ... 付録-13
B.8 Ver.7.0 から Ver.7.1 への追加機能 ... 付録-23
B.9 Ver.7.1 から Ver.8.0 への追加機能 ... 付録-33
B.10 Ver.8.0 から Ver.9.0 への追加機能 ... 付録-34
付録 C. バージョンアップにおける注意事項 ... 付録-36
C.1 プログラムの動作保証 ... 付録-36
C.2 旧バージョンとの互換性 ... 付録-37
付録 D. ASCII コード表 ... 付録-38

1. 概説

1.1 概要

SuperH RISC engine C/C++コンパイラは、機器組み込み用シングルチップ RISC マイコンルネサステクノロジ SuperH RISC engine ファミリの機能・性能を活かしたプログラムを、C 言語で効果的に作成できるようにしたコンパイラです。

本書では、この C/C++コンパイラを用いて応用プログラムを作成する手法を説明します。

1.2 特長

SuperH RISC engine C/C++コンパイラの特長を以下に示します。

(1) 豊富な機能

ルネサステクノロジ SuperH RISC engine ファミリの応用プログラムを効果的に作成できる次の機能があります。

- 割り込み関数やルネサステクノロジ SuperH RISC engine ファミリ専用の特殊命令の C 言語記述機能
- ポジションインディペンデントコード生成（SH-1 以外）
- 高速浮動小数点演算
- 最適化の実行速度優先、メモリ効率優先の選択

(2) 強力な最適化

RISC(Reduced Instruction Set Computer)タイプの命令セットを持つルネサステクノロジ SuperH RISC engine ファミリの性能を発揮できるように、次の最適化を実現しています。

- 局所変数のレジスタへの自動／最適割り付け
- 演算の強度軽減
- パイプライン最適化
- 定数の畳み込み
- 文字列の共有化
- 共通式／ループ不変式の削除
- 不要文の削除
- テールリカージョン最適化
- モジュール間最適化

このため､ルネサステクノロジ SuperH RISC engine ファミリのアーキテクチャを意識しないでプログラミングできます。

1.3 インストール方法

1.3.1 PC 版

Windows98/Me/2000/XP/NT 対応 SuperH RISC engine C/C++コンパイラの動作環境および Windows98/Me/2000/XP/NT 上に組み込むための手順を示します。

(1) 動作環境

- ホストコンピュータ：IBM PC 互換機
  （CPU：Windows98/Me/2000/XP/NT が動作するもの）
- OS：Windows98/Me/2000/XP/NT（日本語または英語）
- メモリ容量：最低 128MB、256MB 以上を推奨
- ハードディスク容量：空き容量 120MB 以上（フルインストール時）
- ディスプレイ：SVGA 以上
- I/O：CD-ROM ドライブ
- その他：マウスなどのポインティングデバイス

(2) PC へのインストール方法

統合化環境を PC に組み込むためには、［コントロールパネル］の［アプリケーションの追加と削除］でセットアップボタンをクリックしてください。以下、メッセージにしたがってインストールしてください。

(3) DOS プロンプトからコンパイラを使用する場合

Windows で DOS プロンプトからコマンドラインでコンパイラを使用する場合、環境変数の設定が必要となります。

■環境変数の説明

（a） 環境変数SHC_LIB
SuperH RISC engine C/C++コンパイラ本体の格納場所を示します。

（b） 環境変数SHC_TMP
SuperH RISC engine C/C++コンパイラが作業用のテンポラリファイルを作成するパスを指定します。この設定を省略することはできません。

（c） 環境変数SHC_INC
SuperH RISC engine C/C++コンパイラの標準ヘッダファイルを特定のパスから取り込む場合に指定します。このパスはカンマ(‘,’)で区切ることにより複数指定することができます。この設定がない場合はSHC_LIBから標準ヘッダファイルを取り込みます。

まず、DOS プロンプト起動時に必要な以下の内容のバッチファイルを作成します。すでに作成済みの方は、項目を付け加えてください。以下の内容は、ハードディスクドライブ C に統合化環境をインストールした場合の例です。
PATH の設定については、MS-DOS プロンプトの“set”コマンドで確認後、現在の PATH 設定に追加してください。
以下にバッチファイルの記述例を示します。

```
PATH C:¥Hew3¥Tools¥Renesas¥Sh¥9_0_0¥bin; %PATH%
SET SHC_LIB=C:¥Hew3¥Tools¥Renesas¥Sh¥9_0_0¥bin
SET SHC_TMP=C:¥tmp
SET SHC_INC=C:¥Hew3¥Tools¥Renesas¥Sh¥9_0_0¥include
```

次に、DOS プロンプトのプロパティで、［プログラム］タグの〈バッチファイル〉に下記のようにバッチファイルのパスを記入してください。

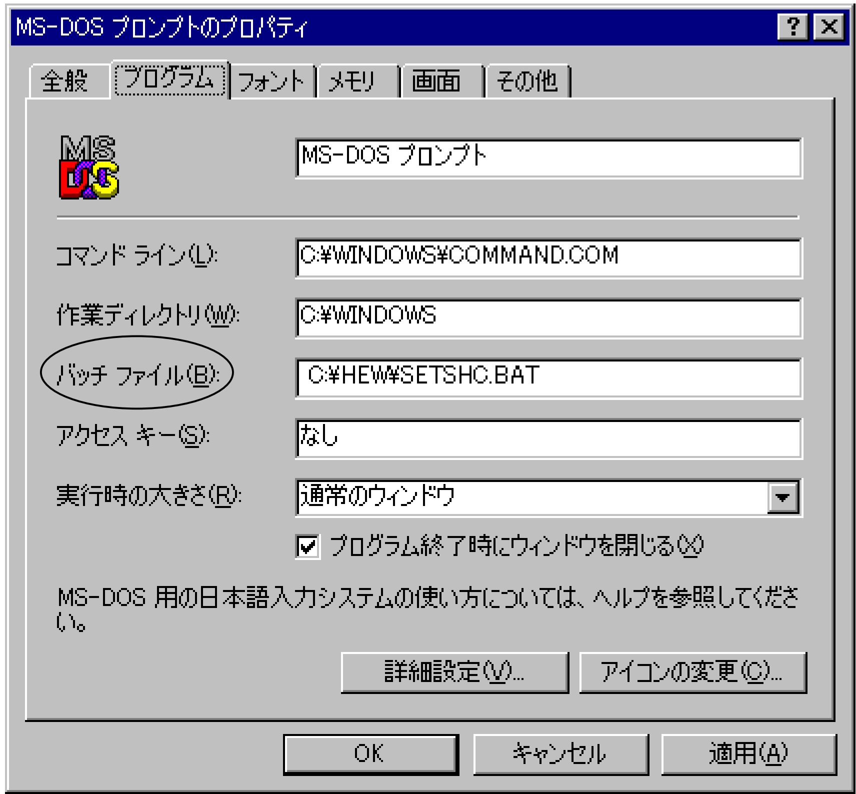


図 1.1　MS-DOS プロンプトのプロパティ(1)

以上の設定を終えた後に、MS-DOS プロンプトを開き直してください。

【注】　バッチを起動して「環境変数のための領域が足りません」というメッセージが出た場合、次のように設定を変更してください。

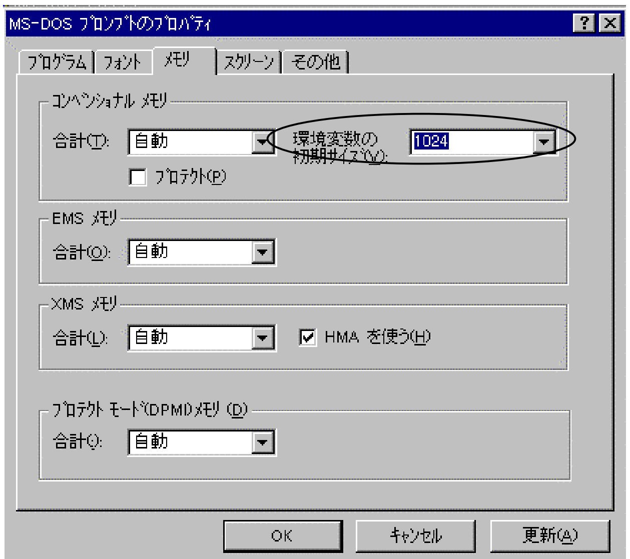


図 1.2 MS-DOS **プロンプトのプロパティ**(2)

MS-DOS プロンプトのプロパティの［メモリ］タグで〈環境変数の初期サイズ〉を自動から 1024 まで増やしてください。

この設定後も、MS-DOS プロンプトを開き直す必要があります。

RENESAS

1.3.2 UNIX 版

SuperH RISC engine C/C++コンパイラを UNIX システムにインストールするための手順を以下に示します。

【注】 インストールディレクトリに漢字・空白を使用しないでください。

(1) **記録媒体**

本コンパイラは、CD-ROM 1 枚で提供致します。

(2) **インストール方法**

ご使用のマシンへの組み込みは以下の手順で行ってください。なお、説明内の(RET)は[Enter]キーを示します。

(a) コンパイラ／シミュレータのインストール
コンパイラ／シミュレータのインストール手順を以下に示します。

(i) コンパイラ／シミュレータ用パスの作成
コンパイラの各ファイルを格納するパスを任意の名称で作成します。

```
% mkdir△コンパイラ／シミュレータ用パス名称 (RET)
```

(ii) CD-ROMのマウント
以下のようにCD-ROMをマウントします。自動的にマウントされる場合は以下のコマンドは必要ありません。

[Solaris の場合]

```
% mount△-r△-F△hsfs△/dev/dsk/c0t6d0s2△/cdrom (RET)
```

[HP-UX の場合]

```
% mount△/dev/dsk/c201d2s0△/cdrom (RET)
```

(iii) コンパイラ／シミュレータのコピー
作成パスに移動して、提供 CD-ROM から(i)で作成したパスにSuperH RISC engine C/C++コンパイラ／シミュレータのソフトウェア一式を解凍します。

[Solaris の場合]

```
% cd△コンパイラ／シミュレータ用パス名称 (RET)
% tar△-xvf△/cdrom/sh_c_sim_pack_sparc/Program.tar (RET)
```

[HP-UX の場合]

```
% cd△コンパイラ／シミュレータ用パス名称 (RET)
% tar△-xvf△/cdrom/Program.tar (RET)
```

(iv) 環境の設定
以下のように環境変数、パス指定を行います。（**には適当な指定を行います。）環境変数についての詳細は**「SuperH RISC engine C/C++コンパイラ、アセンブラ、最適化リンケージエディタユーザーズマニュアル」**をご覧ください。

```
% setenv△SHC_LIB△コンパイラ／シミュレータ用パス名称 (RET)
% setenv△SHC_INC△コンパイラ／シミュレータ用パス名称 (RET)
% setenv△SHC_TMP△/usr/tmp (RET)
% setenv△SHCPU△SH** (RET)
% setenv△HLNK_TMP△/usr/tmp (RET)
% setenv△HLNK_LIBRARY1△コンパイラ／シミュレータ用パス名称/******.lib (RET)
% setenv△HLNK_LIBRARY2△コンパイラ／シミュレータ用パス名称/******.lib (RET)
```

(v) CD-ROMをアンマウントします。

```
% umount△/cdrom (RET)
```

(b) シミュレータのインストール
Ver8より前のバージョンにて、UNIX版シミュレータをインストールする手順は以下のとおりです。

(i) CD-ROMのマウント
README.TXTを参照してCD-ROMをマウントしてください。
README.TXTに書かれている方法で、クロスソフトウェアをコピーしている場合は、コピーしたディレクトリに移動し、<u>(iii)インストーラの起動</u>の説明に移行してください。

(ii) CD-ROM上のtarfileよりインストーラを読み込みます。
（CD-ROMドライバの装置名は、/cdromとしています。）

```
tar△xvf△/cdrom/program.tar△cas_install (RET)
```

(iii) インストーラを起動します。

```
cas_install (RET)
```

(iv) インストーラ起動直後の表示
インストーラ起動直後の表示を以下に示します。

```
Installation of the cycle-accurate simulator starts. Input parameters according to the messages.
```

(v) CPUの選択
使用するシミュレータのCPU選択します。（例では10のSH4を選択しています。）
SH4またはSH2DSPを選択した場合は<u>(vi)協調検証ツールの選択</u>の説明に移行してください。
SH4またはSH2DSP以外を選択した場合は
<u>(vii)定義ファイルインストールディレクトリ名入力</u>の説明に移行してください。

```
Target
CPU (1:SH1,2:SH2,3:SH3,4:SH3E,5:SHDSP,6:SH2E,7:SH4BSC,8:SH3DSP,9:SHDSPC,10:SH4,11:SH2DSP)
: 10
```

(vi) 協調検証ツールの選択
(v)でSH4またはSH2DSPを選択した場合、使用するシミュレータの協調検証ツール名を選択します。
（例では１のSeamlessを選択しています。）
協調検証ツールを使用しない場合は、Noを選択してください。
なお、(v)でSH2DSPを選択した場合は、Ealgeiは選択できません。

```
SH-4:Please select Co-Velification Tool( 1:Seamless,2:Eaglei,3:No)
: 1
```

(vii) 定義ファイルインストールディレクトリ名入力
定義ファイルをインストールするディレクトリを入力します。「（ ）」の中の表示は、デフォルトの情報を示しています。デフォルトの情報は、以下の規則で生成します。
〈カレントディレクトリ〉＋ "/df_CSDSH"

デフォルトのディレクトリ名のままで良い場合、(RET)を入力してください。
ディレクトリ名を入力する場合は、絶対パスでも、相対パスでも指定可能です。
（例では、(RET)を入力しています。）

```
Directory name for the definition files(/export/home1/cas/cassh3sim/df_CSDSH)
: (RET)
```

(viii) CD-ROMドライバのあるホストマシン名入力
CD-ROMドライバのあるホストマシン名を入力します。デフォルトの情報は、起動ホストのホスト名を表示しています。起動ホストのCD-ROMドライバからインストールする場合には、(RET)を入力してください。
ネットワーク上の別なホストのCD-ROMドライバからインストールするときは、そのホスト名を入力してください。
ただし、ネットワーク上の別なホストのCD-ROMドライバからインストールする場合には、リモートシェルでログイン可能になっていること（/etc/hosts.equiv および$HOME/.rhostsファイルが設定されている。）を前提としています。
リモートシェルの環境設定などについては、起動マシンのマニュアルを参照してください。
起動ホストのCD-ROMドライバからインストールする場合には、(v) tarfile名入力の説明に移行してください。
ネットワーク上の別なホストのCD-ROMドライバからインストールする場合には、(ix) CD-ROMドライバのあるホストマシンのログイン名称入力の説明に移行してください。
（例では、CD-ROMドライバのある別ホスト名を sp3 としています。）

```
Host name connected to a tape driver(sparc2): sp3 (RET)
```

(ix) CD-ROMドライバのあるホストマシンのログイン名称入力
CD-ROMドライバのあるホストマシンのログイン名称を入力します。当メッセージは、別ホストのCD-ROMドライバからインストールする際に表示します。
（例では、CD-ROMドライバのあるホストマシンのログイン名称を remote としています。）

```
Login name of host connected to a tape driver:remote (RET)
```

(x) tarfile名入力
tarfile名を入力します。デフォルトは、HP9000:/dev/rmt/0m、SPARC:/dev/rmt/0としています。
（CD-ROMドライバの装置名は、/cdromとしています。）

```
Tape driver name(/dev/rmt/0): /cdrom/simulator.tar(RET)
```

(xi) 定義ファイルインストール前の(RET)キー入力
定義ファイルのメディアがCD-ROMドライバにマウントされているのを確認して、(RET)を入力してください。

```
Input return,after setting the tape including the definition files to the tape driver.
(RET)
```

(xii) 本体インストール選択入力
インタフェースソフト本体をインストールするかどうかを入力します。
インストールする場合には、yを入力し、インストールしない場合には、nを入力します。
nを入力した場合は、(xv)セットアップファイルインストール選択入力の説明に移行してください。
（例では、yを入力しています。）

```
Do you install the main files?(y/n):y(RET)
```

(xiii) 本体インストールディレクトリ名入力
インタフェースソフト本体をインストールするディレクトリを入力します。
デフォルトの情報は以下の規則で生成します。
〈カレントディレクトリ〉＋ "/main"

ディレクトリ名を入力する場合は、絶対パスでも、相対パスでも指定可能です。
（例では、(RET)を入力しています。）

```
Directory name for the main files(/export/home1/cas/cassh3sim/main)
: (RET)
```

(xiv) 本体インストール前の(RET)キー入力
インタフェースソフト本体のメディアがCD-ROMドライバにマウントされているのを確認して、(RET)を入力してください。

```
Input return,after setting the tape including the main files to the tape driver. (RET)
```

(xv) セットアップファイルインストール選択入力
セットアップサンプルファイルをコピーするかどうかを入力します。
コピーする場合には、yを入力し、コピーしない場合には、nを入力します。
（例では、yを入力しています。その後インストールファイル名を表示します。）

```
Do you copy the setup files to current directory?(y/n):y(RET)
```

(xvi) パスおよび環境変数設定選択入力
パスおよび環境変数の設定をシェルスクリプトに追加するかどうかを入力します。
ｙを入力するとインストーラは、環境変数"SHELL"からログインシェル種別を判断し、環境変数"HOME"のディレクトリ直下の任意（**表1.1参照**）のシェルスクリプトファイルのバックアップをとり、パス、および環境変数をセットします。
ただし、以下の仕様でセットします。
本体をインストールしなかった場合（(xii)本体インストール選択入力参照）は、パスの設定を行いません。
(vii)定義ファイルインストールディレクトリ名入力、および(xiii)本体インストールディレクトリ名入力にて、相対パスで指定した場合は、入力した情報でパス、および環境変数を設定しますのでインストーラを起動したディレクトリ以外では作業できません。

nを入力すると(xviii)インストール完了メッセージに移行し、本インストーラを終了します。

表 1.1　シェルごとの使用ファイル名

No.	シェル名	対象スクリプト ファイル名	バックアップ ファイル名
1	ボーンシェル(sh)	.profile	.profile.bak
2	Cシェル(csh)	.cshrc	.cshrc.bak
3	コーンシェル(ksh)	.profile	.profile.bak

（例では、yを入力しています。その後、対象のシェルスクリプトを表示します。）

```
Do you append the path list and the environment variables in shell script?(y/n):y(RET)
/export/home1/cas/.cshrc
```

(xvii) バックアップファイルオーバライト選択入力
シェルスクリプトバックアップ時に、バックアップファイル名と同じファイルが存在している場合、当メッセージを表示します。オーバライトするかどうか入力してください。
（例のログインシェルは、Cシェルです。）

```
Do you overwrite the backup file(.cshrc.bak)?(y/n):y(RET)
```

(xviii) インストール完了メッセージ
すべてのインストールが完了すると以下のメッセージを表示して、終了します。

```
Installation of the cycle-accurate simulator completed.
```

(c) Acrobat® Readerのインストール
マニュアルはWindows上から参照できます。このためにマニュアルを参照するためのソフトウェア(Acrobat® Reader)をWindows98/Me/2000/XP/NTが動作しているパーソナルコンピュータにインストールしてください。

Acrobat® Reader copyright © 1987-2001 Adobe Systems Incorporated. All rights reserved.
AdobeおよびAcrobatはアドビシステムズ社の商標で特定の法域で登録されています。

以下の手順でインストールを実行します。インストールは、実行中のアプリケーションをあらかじめ終了させてから実行してください。
(i) 提供CD-ROMをCD-ROMドライブに挿入します。（以下、仮にDドライブとします）
(ii) Windows®スタートメニューの［ファイル名を指定して実行...］をクリックします。
(iii) CD-ROMの[PDF_Read¥Japanese]ディレクトリにあるACROBEADER51_JPN.EXE (日本語版)または[PDF_Read¥English] ディレクトリにあるACROBEADER51_ENU.EXE (英語版)を［ファイル名を指定して実行］ダイアログボックスで指定し（例D:¥PDF_Read¥Japanese¥ ACROBEADER51_JPN.EXE）、[OK]をクリックします。
(iv) 画面に表示されるインストールの指示に従います。

1.4　起動方法

1.4.1　統合化環境からの起動方法

統合化環境のインストーラは、インストール正常終了時、Windows のスタートメニューのプログラムフォルダの下に Renesas High-performance Embedded Workshop という名称のフォルダを作成し、そのフォルダ内に統合化環境の実行プログラムである統合化環境などの各ショートカットを登録します。なお、スタートメニューの表示内容は、ツールのインストール状況により異なる場合があります。

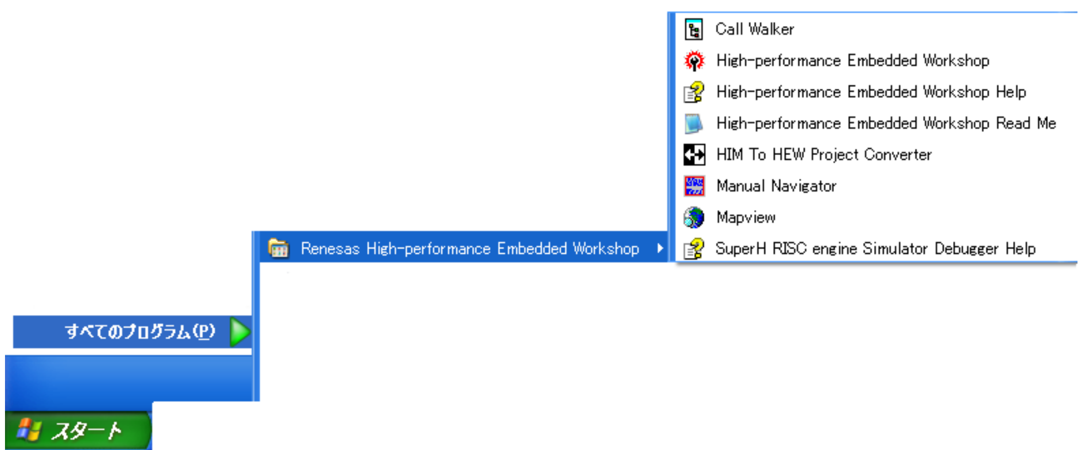


図 1.3　**スタートメニューによる統合化環境起動**

このスタートメニューで、統合化環境をクリックすると起動メッセージを表示し、引き続きようこそ!ダイアログボックス（**図** 1.4）が表示されます。

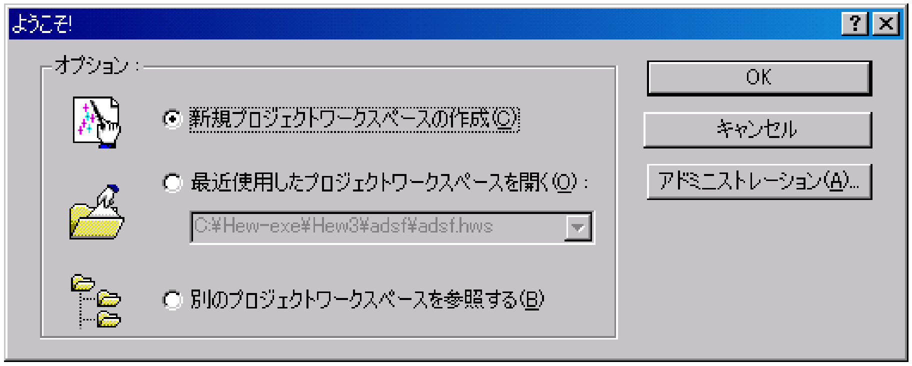


図 1.4　**ようこそ!　ダイアログボックス**

統合化環境を初めて使用する場合や、新たにプロジェクトを作成して作業を開始する場合は、［新規プロジェクトワークスペースの作成］を選択して[OK]をクリックしてください。また、すでに作成したプロジェクトで作業する場合は、[最近使用したプロジェクトワークスペース]または、［別のプロジェクトワークスペースを参照する］を選択して[OK]をクリックしてください。なお、どちらを選択した場合でも、［アプリケーションの終了］をクリックすると統合化環境は終了します。また、［アドミニストレーション...］をクリックすると、統合化環境で使うシステムツールの登録や削除ができます。

1.4.2 コンパイラの起動方法

本節では、SuperH RISC engine C/C++コンパイラの起動方法とその使用例について説明します。コンパイラオプションについては、「SuperH RISC engine **シリーズ C/C++コンパイラ、アセンブラ、 最適化リンケージエディタ ユーザーズマニュアル**」を参照してください。PC 版をご使用の場合は、**操作説明書**を参照してください。

表 1.2 **コンパイル条件判別表**

コマンド	**オプション**	**コンパイル対象ファイルの拡張子**	**コンパイル条件**
shcpp	**任意**	**任意**	**C++でコンパイル**
shc	-lang=c	**任意**	**Cでコンパイル**
	-lang=cpp		**C++でコンパイル**
	-lang **オプションの指定なし**	*.c	**Cでコンパイル**
		*.cpp, *.cc, *.cp, *.CC	**C++でコンパイル**

コマンド shc は、C プログラム、C++プログラムをそれぞれ、lang オプションまたは、プログラムファイル名の拡張子に従い、C コンパイル*1、C++コンパイル*1 します。コマンド shcpp は、C プログラム、C++プログラムに関係なく、C++コンパイルします。**表** 1.2 にコンパイル条件判別表を示します。

【注】 *1 C **コンパイルとは、プログラムを** C **言語の文法に基づいてコンパイルすることを意味しています。**C++**コンパイルとは、**C++**言語の文法に基づいてコンパイルすることを意味しています。**

以下、コンパイラの基本的な操作方法を説明します。

(1) **プログラムのコンパイル**

Cソースプログラム「test.c」をコンパイルします。

```
shc△test.c (RET)
```

C++ソースプログラム「test.cpp」をコンパイルします。

```
shc△test.cpp(RET)
shcpp△test.cpp(RET)
```

(2) **コマンド入力形式、コンパイラオプションの表示**

標準出力画面上にコマンドの入力形式、コンパイラオプションの一覧を表示します。

```
shc (RET)
shcpp(RET)
```

(3) **オプション指定方法**

オプション(debug、listfile、showなど)の前に － を付加し、複数のオプションを指定するときはスペース(△)で区切ります。PC版では、DOSプロンプトで － のかわりに / を使用することもできます。
複数のサブオプションを指定するときはコンマ(,)で区切って指定します。

```
shc△-debug△-listfile△-show=noobject,expansion△test.c  (RET)
```

PC版では、さらに括弧（ ）でくくって指定することもできます。

```
shc△/debug△/listfile△/show=(noobject,expansion)△test.c  (RET)
```

(4) 複数の C/C++プログラムのコンパイル

複数のC/C++プログラムを一度にコンパイルできます。以下に、Cソースプログラムをコンパイルする例を示します。

例 1 複数プログラムの指定方法

```
shc△test1.c△test2.c (RET)
```

例 2 オプションの指定(C ソースプログラムすべてに有効なオプション指定例)

```
shc△-listfile△test1.c△test2.c (RET)
```

「test1.c」、「test2.c」ともlistfileオプションが有効となります。

例 3 オプションの指定(プログラムごとに有効なオプション指定例)

```
shc△test1.c△test2.c△-listfile  (RET)
```

listfile オプションは「test2.c」だけに対して有効になります。プログラムごとのオプション指定は、ソースプログラム全体に対するオプション指定よりも優先されます。

【注意事項】

(1) **コンパイラをインストールしても起動できない場合は、次の点をまず再確認してください。**

- 環境変数"PATH"が C/C++コンパイラのあるディレクトリへ設定されているか。
- 環境変数"SHC_LIB"が C/C++コンパイラ本体のあるディレクトリへ設定されているか。
  環境変数"SHC_LIB"は、C/C++コンパイラ本体があるディレクトリを指定する役割があります。
  したがって、C/C++コンパイラ本体のファイル一式は同一ディレクトリに入れておかないとコンパイラは動作しません。

(2) **コンパイラは、shc,shcpp のコマンドの使い分けでもコンパイル時の文法を決定しますが、shc のコマンド使用時でもファイルの拡張子やオプションによりC++コンパイルを行います。**

1.5 プログラム開発手順

C/C++言語プログラムの開発手順を**図** 1.5 に示します。網掛け部分は、「SuperH RISC engine C/C++**コンパイラパッケージ**」として提供するソフトウェアを示します。

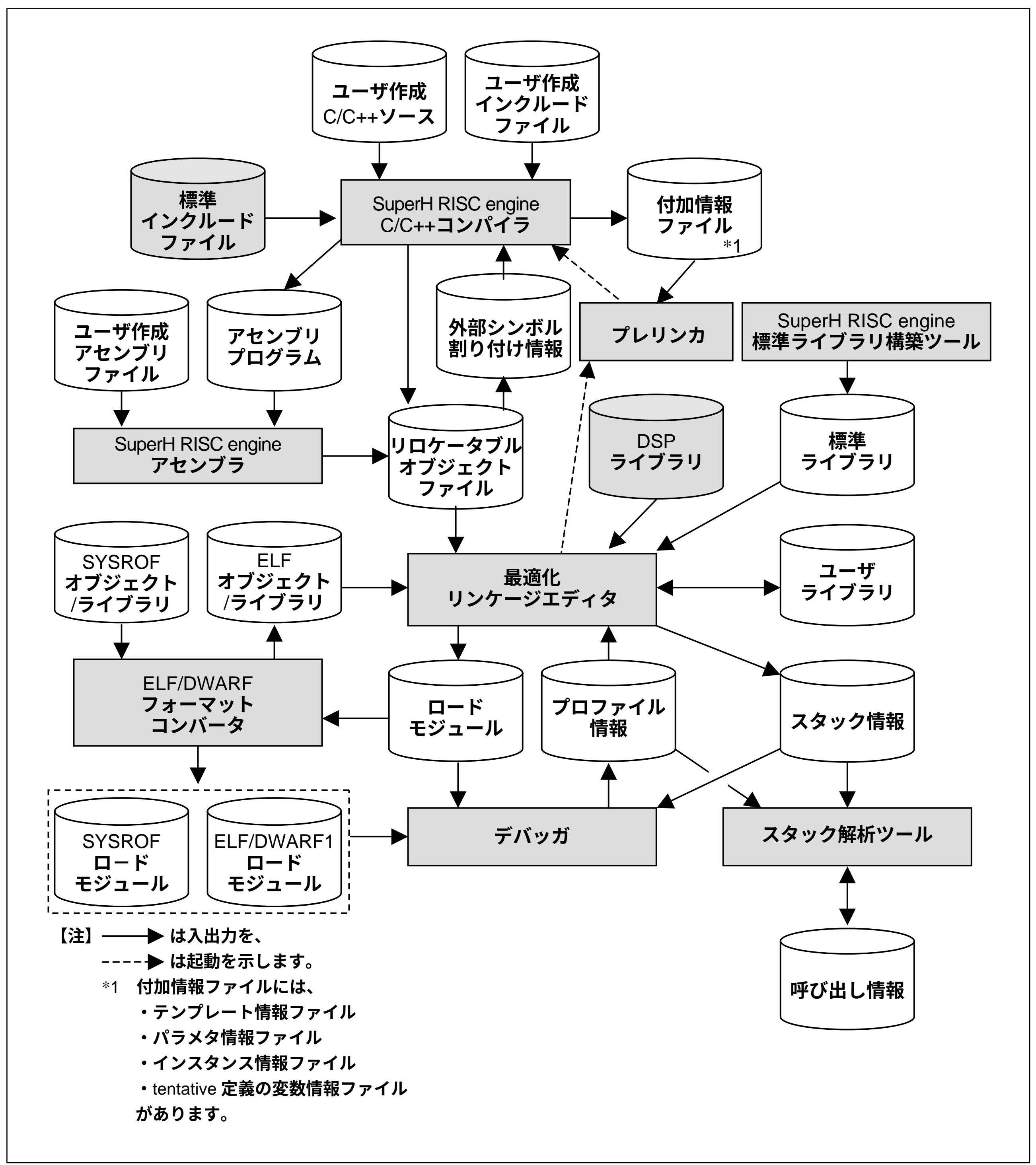


図 1.5 **プログラム開発手順**

ソースファイル on_motor.c を例にとり、プログラム開発手順を説明します。なお、各クロスソフトの使用方法の詳細は各クロスソフトのユーザーズマニュアルを参照してください。

(1) ソースファイルの作成

エディタを用いてソースファイルを作成します。

(2) リロケータブルオブジェクトファイルの生成

コンパイラを起動し、ソースファイルをコンパイルします。

```
shc△on_motor.c (RET)
```

on_motor.obj という名前のデバッグ情報のない最適化されたリロケータブルオブジェクトファイルが生成されます。リストファイルを生成するには、listfile オプションを指定してください。

(3) ロードモジュールファイルの生成

下記のようにライブラリファイル sensor.lib を取り込みリンケージエディタを起動すると、実行可能な on_motor.abs という名前のロードモジュールファイルが生成されます。

```
optlnk△-noopt△-subcommand = link.sub (RET)
```

lnk.sub の内容は以下のとおりです。

```
Sdebug
input on_motor
library sensor.lib
Exit
```

なお、リロケータブルオブジェクトファイルがデバッグ情報付きであっても、リンク時に debug オプションを省略すると、ロードモジュールファイルではデバッグ情報は出力されませんので注意してください。

(4) S タイプ形式ファイルの出力

ROM ライタを用いて EPROM に書き込む場合には、lnk.sub を下記のように記述します。

```
Form=stype
Sdebug
input on_motor
library sensor.lib
Exit
```

on_motor.mot という名前の S タイプ形式ロードモジュールファイルが生成されます。

2. プログラムの作成とデバッグまでの手順

2.1 プロジェクトの構築

2.1.1 シミュレータデバッガ用プロジェクトの作成

(1) プロジェクトの指定

“ようこそ!”ダイアログボックスで［新規プロジェクトワークスペースの作成］を選択して[OK]をクリックすると、新しいワークスペースとプロジェクト作成用の”新規プロジェクトワークスペース”ダイアログボックス（図 2.1）を表示します。このダイアログボックスでワークスペース名（新規作成時はプロジェクト名もデフォルトで同名です）や CPU の種類、プロジェクトのタイプなどを設定します。

たとえば、［ワークスペース名］にワークスペース名として“ tutorial ”と入力すると、［プロジェクト名］も”tutorial”になり、［ディレクトリ］も“ C:¥Hew3¥tutorial ”となります。プロジェクト名を変更する場合は、［プロジェクト名］に直接入力し、ワークスペースとして使用するディレクトリを変更する場合は、［参照...］をクリックしてディレクトリを選択するか、直接［ディレクトリ］に入力してください。

ここでは、左側のプロジェクトタイプに[Demonstration]を指定します。

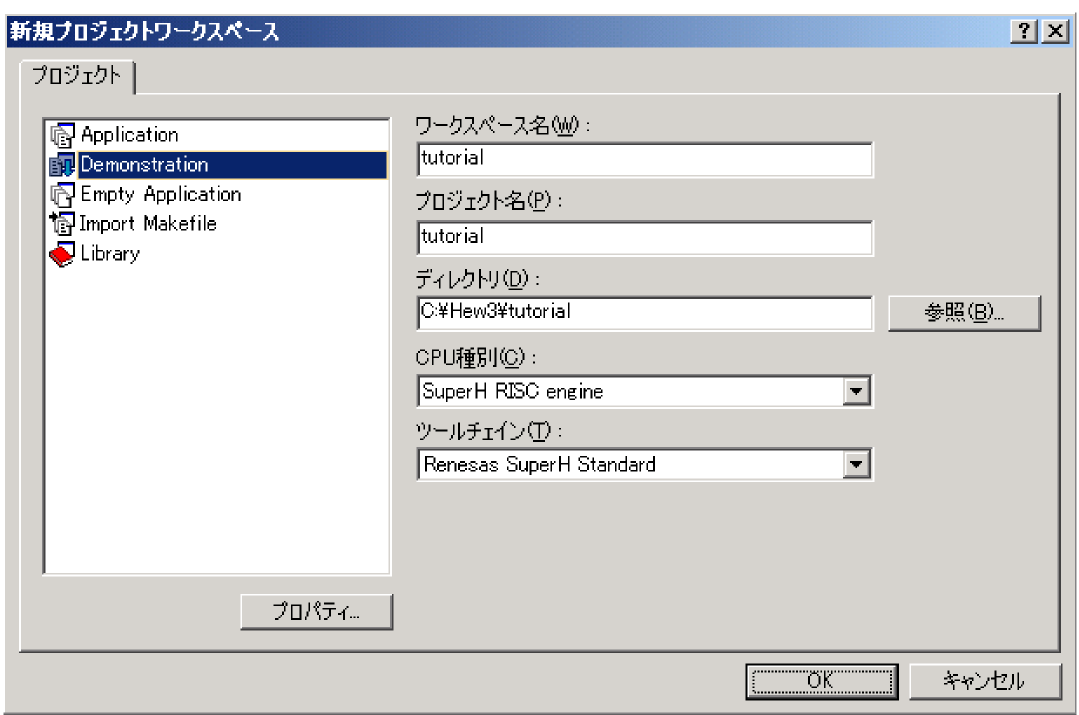


図 2.1 新規プロジェクトワークスペースダイアログボックス

(2) CPU **の選択**

“新規プロジェクトワークスペース”ダイアログボックスで[OK]をクリックすると、プロジェクトジェネレータを起動します。最初に使用する CPU を選択します。使用する CPU の種類（［CPU タイプ］）は CPU のシリーズ（［CPU シリーズ］）ごとに分類しています。［CPU シリーズ］および［CPU タイプ］の選択により、生成するファイルが異なるので、開発するプログラムの対象となる CPU タイプを選択してください。選択したい CPU タイプがない場合は、ハードウェア仕様の近い CPU タイプまたは“ Other ”を選択してください。

- ［次へ(N)>］をクリックすると、次の画面を表示します。
- ［<戻る(B)］をクリックすると、この画面を表示する前の画面またはダイアログボックスを表示します。
- ［完了］をクリックすると、“ Summary ”ダイアログボックスが開きます。
- ［キャンセル］をクリックすると、“ 新規プロジェクトワークスペース ”ダイアログボックスに戻ります。

［<戻る(B)］、［次へ(N)>］、［完了］および［キャンセル］の機能は、このウィザードダイアログボックスで共通の機能です。

ここでは、[CPU シリーズ]で“ SH-1 ”を選択（**図** 2.2）して、［次へ(N)>］をクリックしてください。

demonstration を選択した場合、CPU タイプの選択はできません。

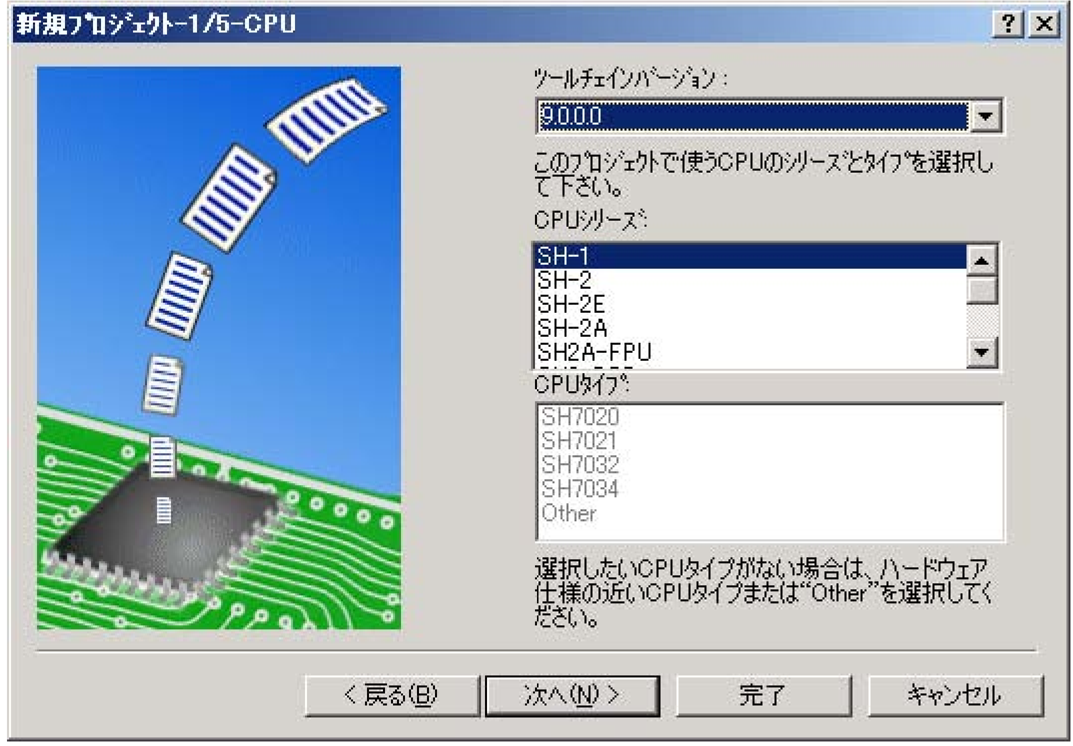


図 2.2 New Project Step 1 **ダイアログボックス**

(3) **オプションの選択**

Step1 画面で［次へ(N)>］をクリックすると、**図** 2.3 に示す画面を表示します。この画面で、全プロジェクトファイルで共通のオプションを設定します。オプション設定項目は、Step1 画面で選択した CPU シリーズに併せて設定変更ができるようになっています。また、プロジェクト作成後にオプションを変更する場合は、HEW の［オプション-> SuperH RISC engine Standard Toolchain］の CPU タブで変更できます。

ここでは、設定を変更しないで［次へ(N)>］をクリックします。クリックすると、Step3 の画面を表示します。

図 2.3 New Project Step 2 **ダイアログボックス**

(4)　デバッガターゲットの設定

Step2 画面で［次へ(N)>］をクリックすると、**図** 2.4 に示す画面を表示します。この画面で、デバッガターゲットを設定します。［ターゲット］から使用するデバッガターゲットを選択（チェック）してください。デバッガターゲットは、未選択でも複数選択してもかまいません。

ここでは、“ SH-1 Simulator ”を選択し、［次へ(N)>］をクリックします。

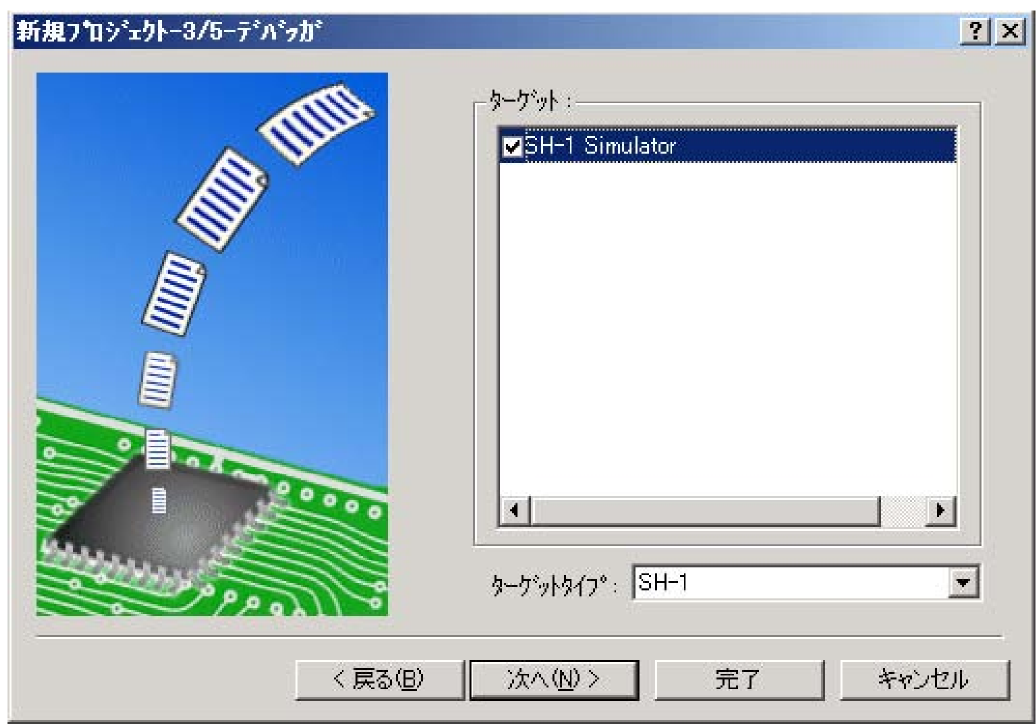


図 2.4　New Project Step 3 **ダイアログボックス**

(5)　**デバッガオプションの設定**

Step3 画面で［次へ(N)>］をクリックすると、**図** 2.5 に示す画面を表示します。この画面で、選択したデバッガターゲットのオプションを設定します。

HEW はデフォルトで、“ elease ”と“ Debug ”の 2 つのコンフィグレーションを作成しますが、デバッガターゲットを選択すると、選択したターゲット用のコンフィグレーションも作成します（ターゲット名を含んだ略称となります）。このコンフィグレーション名は、[Configuration name:]で変更できます。また、デバッガターゲットのオプションを、[Detail options:]で表示します。変更する場合は、[Item]を選択して[Modify]をクリックしてください。なお、変更できない項目の場合、[Item]を選択しても[Modify]はグレーのままです。

ここでは、設定を変更しないで［次へ(N)>］をクリックします。

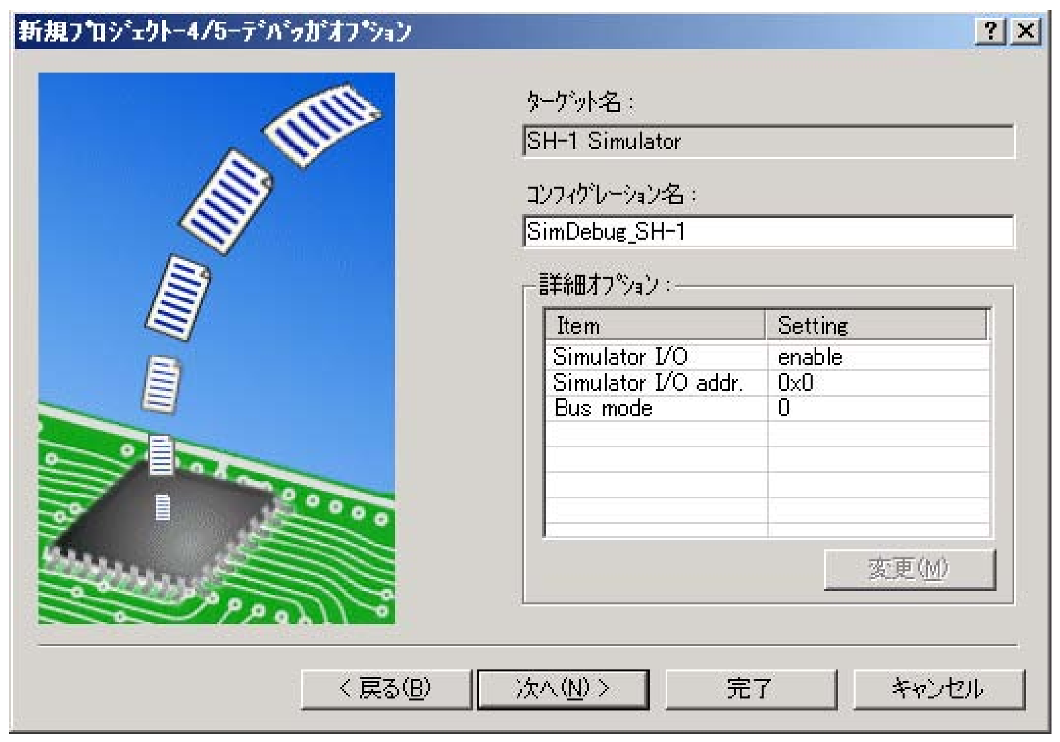


図 2.5　New Project Step 4 **ダイアログボックス**

(6) 設定確認（Summary ダイアログボックス）

Step4 画面で［次へ(N)>］をクリックすると**図** 2.6 に示す画面を表示します。この画面で、生成するプロジェクトのソースファイル情報を表示します。確認後、[完了]をクリックしてください。

図 2.6 の画面で［完了］をクリックすると、プロジェクトジェネレータは、生成するプロジェクトに関する情報をSummary ダイアログボックス（**図** 2.7）で表示しますので、確認後、[OK]をクリックしてください。

なお、[Generate Readme.txt as a summary file in the project directory]をチェックすると、Summary ダイアログボックスで表示したプロジェクトの情報を、”Readme.txt”という名称のテキストファイルでプロジェクトディレクトリに保存します。

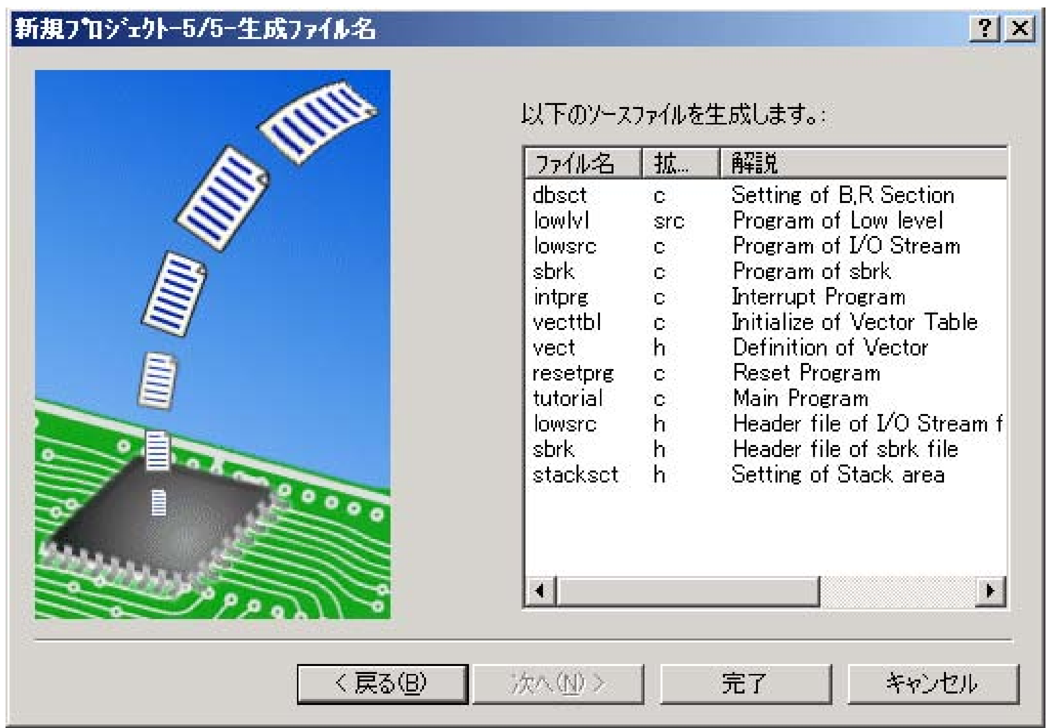


図 2.6 New Project Step 5 **ダイアログボックス**

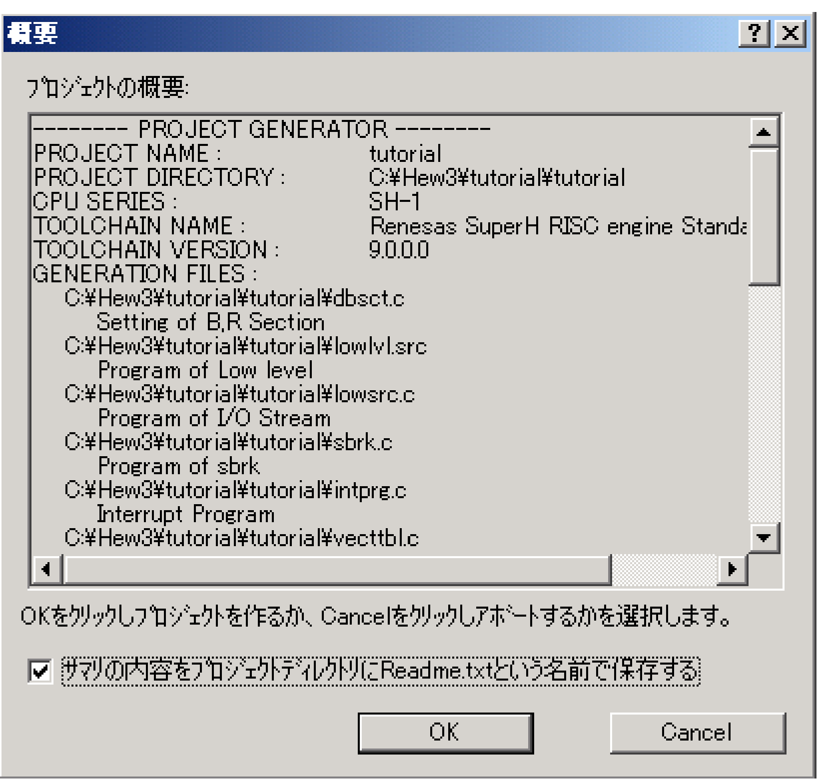


図 2.7 Summary ダイアログボックス

(7) その他

Project Type で demonstration を選択した場合、シミュレータデバッグ時に使用できる低水準ライブラリサンプルが組み込まれます。組み込まれるファイルは次のとおりです。

- lowlvl.src　（標準入出力サンプルアセンブラリスト）
- lowsrc.c　（低水準ライブラリソースファイル）
- lowsrc.h　（低水準ライブラリヘッダファイル）

2.2 サンプルプログラムの紹介 (SH-1,SH-2,SH-2E,SH-2A,SH2A-FPU,SH2-DSP)

本節では、**図** 2.8 に示す構成のサンプルプログラムを用いて、プログラム作成の実際を説明します。開発環境は**表** 2.1 に示します。

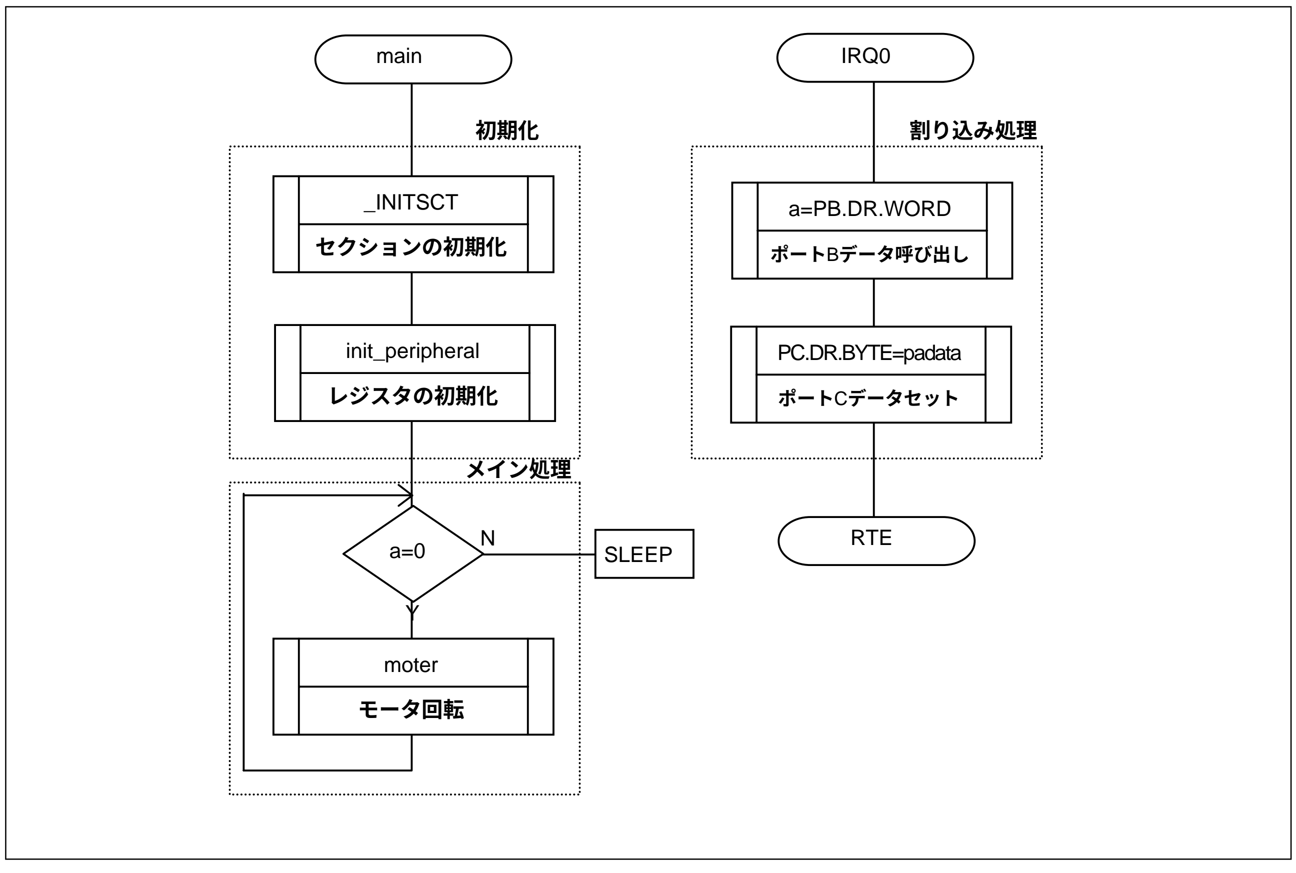


図 2.8 **サンプルプログラムの流れ**

表 2.1 **サンプルプログラム開発環境**

OS	UNIX
CPU	SH-1

2.2.1　ベクタテーブルの作成

ベクタテーブル作成プログラムを**図** 2.9 に示します。ベクタテーブル作成方法の詳細は、「3.1.3　**ベクタテーブルの作成**」を参照してください。

アセンブリ言語で**図** 2.9 と同じ内容を記述すると、**図** 2.10 のようになります。

```
/**************************************************************/
/*                 file name "vect.c"                         */
/**************************************************************/
      extern void main(void);
      extern void inv_inst(void);
      extern void IRQ0(void);

      void (* const vec_table[])(void)={
            main,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
            0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
            inv_inst, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
            0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
            IRQ0
      };
```

図 2.9　**ベクタテーブル作成プログラム（C 言語版）**

SH-１のベクタテーブルは**表** 2.2 のようになっています。

パワーオンリセットにより関数 main が起動します。このときスタックポインタは０に設定されます。

関数 inv_inst の先頭アドレスをベクタ番号 32 に、関数 IRQ0 の先頭アドレスをベクタ番号 64 に設定します。これらは、それぞれユーザベクタと外部割り込みの先頭ベクタ番号です。

表 2.2　**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
:		:	:
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	IRQ0	64	H'00000100～H'00000103
	:	:	:

```
.SECTION    VECT,DATA,ALIGN=4
.IMPORT     _main
.IMPORT     _inv_inst
.IMPORT     _IRQ0
.DATA.L     _main                    ;_main の先頭アドレスを番号 0 のベクタに設定
.DATA.L     H'0000000                ;SP の初期値をベクタ番号 1 に設定
.ORG        H'0080
.DATA.L     _inv_inst                ;_inv_inst の先頭アドレスを番号 32 のベクタに設定
.ORG        H'0100
.DATA.L     _IRQ0                    ;_IRQ0 の先頭アドレスを番号 64 のベクタに設定
.END
```

図 2.10　ベクタテーブル作成プログラム（アセンブリ言語版）

C 言語プログラムの外部名は、アセンブリ言語プログラムにおいては、先頭に"_"を付加します。

2.2.2 ヘッダファイルの作成

サンプルプログラムで共通に使用するヘッダファイルを**図 2.27** に示します。IPRA などの I/O ポートの定義を行うことにより、I/O ポートを変数のように名前でアクセスできるようにします。

```
/************************************************************************/
/*                    file name "7032.h" (抜粋)                         */
/************************************************************************/
/************************************************************************/
/*                    Definitions of I/O Registers                      */
/************************************************************************/
struct st_intc {                                    /* struct INTC      */
               union {                              /*    IPRA          */
                        unsigned short  WORD;       /* Word  Access     */
                     struct {                       /* Bit   Access     */
                           unsigned char  UU:4;     /*    IRQ0          */
                           unsigned char  UL:4;     /*    IRQ1          */
                           unsigned char  LU:4;     /*    IRQ2          */
                           unsigned char  LL:4;     /*    IRQ3          */
                           }  BIT;                  /*                  */
                     }  IPRA;                       /*                  */
               union {                              /*    IPRB          */
                     unsigned short  WORD;          /* Word  Access     */
                     struct {                       /* Bit   Access     */
                             unsigned char  UU:4;   /*    IRQ4          */
                             unsigned char  UL:4;   /*    IRQ5          */
                             unsigned char  LU:4;   /*    IRQ6          */
                             unsigned char  LL:4;   /*    IRQ7          */
                            }         BIT;          /*                  */
                        }         IPRB;             /*                  */
};                                                  /*                  */
#define INTC    (*(volatile struct st_intc      *)0x5FFFF84)
                                                    /* INTC  Address    */

/************************************************************************/
/*                              Timer registers                         */
/************************************************************************/
struct st_itu0 {                                    /* struct ITU0      */
      union {                                       /*    TCR           */
            unsigned char BYTE;                     /* Byte  Access     */
            struct {                                /* Bit   Access     */
                     unsigned char      :1;         /*                  */
                     unsigned char CCLR :2;         /*    CCLR          */
                     unsigned char CKEG :2;         /*    CKEG          */
                     unsigned char TPSC :3;         /*    TPSC          */
                   }        BIT;                    /*                  */
            }           TCR;                        /*                  */
};                                                  /*                  */
   #define ITU0    (*(volatile struct st_itu0      *)0x5FFFF04)
                                                    /* ITU0  Address    */

/************************************************************************/
/*                              PORT registers                          */
/************************************************************************/
```

```
struct st_pa {                                          /*  struct   PA     */
      union {                                           /*   PADR           */
            unsigned short     WORD;                    /*  Word Access     */
            struct {                                    /*  Byte Access     */
                     unsigned char   H;                 /*   High           */
                     unsigned char   L;                 /*   Low            */
                   }        BYTE;                       /*                  */
                     struct {                           /*  Bit  Access     */
                              unsigned char  B15 :1;/*   Bit 15        */
                              unsigned char  B14 :1;/*   Bit 14        */
                              unsigned char  B13 :1;/*   Bit 13        */
                              unsigned char  B12 :1;/*   Bit 12        */
                              unsigned char  B11 :1;/*   Bit 11        */
                              unsigned char  B10 :1;/*   Bit 10        */
                              unsigned char  B9  :1;/*   Bit  9        */
                              unsigned char  B8  :1;/*   Bit  8        */
                              unsigned char  B7  :1;/*   Bit  7        */
                              unsigned char  B6  :1;/*   Bit  6        */
                              unsigned char  B5  :1;/*   Bit  5        */
                              unsigned char  B4  :1;/*   Bit  4        */
                              unsigned char  B3  :1;/*   Bit  3        */
                              unsigned char  B2  :1;/*   Bit  2        */
                              unsigned char  B1  :1;/*   Bit  1        */
                              unsigned char  B0  :1;/*   Bit  0        */
                           }         BIT;               /*                  */
                     }            DR;                   /*                  */
};                                                      /*                  */
#define PB         (*(volatile struct st_pa    *)0x5FFFFC2)
                                                        /*  PB   Address    */

struct st_pc {                                          /*  struct PC       */
         union {                                        /*   PCDR           */
               unsigned char      BYTE;                 /*  Byte Access     */
               struct {                                 /*  Bit  Access     */
                        unsigned char  B7  :1;          /*   Bit 7          */
                        unsigned char  B6  :1;          /*   Bit 6          */
                        unsigned char  B5  :1;          /*   Bit 5          */
                        unsigned char  B4  :1;          /*   Bit 4          */
                        unsigned char  B3  :1;          /*   Bit 3          */
                        unsigned char  B2  :1;          /*   Bit 2          */
                        unsigned char  B1  :1;          /*   Bit 1          */
                        unsigned char  B0  :1;          /*   Bit 0          */
                      }    BIT;                         /*                  */
               }        DR;                             /*                  */
};                                                      /*                  */
#define PC      (*(volatile struct st_pc *)0x5FFFFD1)
                                                        /* PC    Address    */

/************************************************************************/
/*                         file name "sample.h"                         */
/************************************************************************/
/************************************************************************/
/*                            Timer registers                           */
/************************************************************************/
   struct   tcsr   {                                 /*                  */
```

```
short OVF    :1;                /* TCSR struct OVF bit*/
short WTIT   :1;                /* WTIT bit           */
short        :3;                /* work area          */
short CKS2   :1;                /* CKS2 bit           */
short CKS1   :1;                /* CKS1 bit           */
short        :9;                /* work area          */
};                                                  /*                      */
#define  TCSR_FRT  (*(volatile unsigned short *)0x5FFFFB8)
/*                     */
#define  TCSR__FRT (*(volatile struct tcsr *)0x5FFFFB8)
/*                     */


extern void motor( void );                       /* motor module       */
extern void _INITSCT( void );
/* section initialize module   */
extern void init_peripheral(void);
  /* peripheral initialize module*/
```

図 2.11 ヘッダファイル

RENESAS

2.2.3 メイン処理部の作成

メイン処理プログラムを**図** 2.12 に示します。パワーオンリセットにより起動される関数 main と割り込み発生まで呼ばれ続ける関数 motor を定義します。

```
/**********************************************************************/
/*                        file name "sample.c"                        */
/**********************************************************************/
   #include "7032.h"
   #include "sample.h"
   #include <machine.h>                    /* 組み込み関数 sleep を定義       */
   const short padata=0x3;                 /* C セクション                    */
   short a=0;                              /* D セクション                    */
   int work;                               /* B セクション                    */

/**********************************************************************/
/*                           main module                              */
/**********************************************************************/
   void main( void )
   {
         _INITSCT();                       /* 各セクションの初期化            */
         init_peripheral();
         while(!a)    motor();
         sleep();
   }

/**********************************************************************/
/*                           motor module                             */
/**********************************************************************/
   void motor( void )                 /*割り込みが発生するまで呼び出される*/
   {
            :
            :
         return;
   }
```

図 2.12 **メイン処理プログラム**

関数 main では、_INITSCT と init_peripheral を呼び出し、セクション初期化と内部レジスタの初期化を行います。その後、大域変数 a の値が変更されるのを待ちます。その間、関数 motor が常時呼び出されます。a の値が 0 以外になれば、低消費電力状態に入ります。

2.2.4 初期化部の作成

セクション初期化で使用する外部名の値を設定するアセンブリ言語プログラムを**図** 2.13 に、セクション初期化とレジスタ初期化を行う C 言語プログラムを**図** 2.14 に示します。

```
;*****************************************************************
;                   file name "sct.src"                          *
;*****************************************************************
                             .SECTION B,DATA,ALIGN=4
                             .SECTION R,DATA,ALIGN=4
                             .SECTION D,DATA,ALIGN=4
          ; セクションを追加する場合にはここに記述する

                             .SECTION C,DATA,ALIGN=4
          __B_BGN:           .DATA.L  (STARTOF B)
          __B_END:           .DATA.L  (STARTOF B)+(SIZEOF B)
          __D_BGN:           .DATA.L  (STARTOF R)
          __D_END:           .DATA.L  (STARTOF R)+(SIZEOF R)
          __D_ROM:           .DATA.L  (STARTOF D)

                             .EXPORT  __B_BGN
                             .EXPORT  __B_END
                             .EXPORT  __D_BGN
                             .EXPORT  __D_END
                             .EXPORT  __D_ROM
                             .END
```

図 2.13 **初期化プログラム（アセンブリ言語部分）**

B セクションと D セクションの先頭アドレス、最終アドレスを定義します。

C/C++コンパイラではコンパイル時に section オプションでセクション名を指定しないと、それぞれ次のような名称が割り付けられます。

プログラム領域セクション	:P
定数領域セクション	:C
初期化データ領域セクション	:D
未初期化データ領域セクション	:B

R セクションは、リンケージエディタの ROM 化支援機能を用いて ROM 上の初期化データ領域を複写する RAM 上の領域を示しています。リンケージエディタの ROM 化支援機能については、「3.15.2 (1) ROM **化支援機能**」を参照してください。

STARTOF は、"STARTOF 〈セクション名〉"という書式により、セクション集合の先頭アドレスを求める演算子です。
SIZEOF は、"SIZEOF〈セクション名〉"という書式により、セクション集合のサイズをバイト単位で求める演算子です。

```
/*********************************************************************/
/*                         file name "init.c"                        */
/*********************************************************************/
   #include "7032.h"
   #include "sample.h"
/*********************************************************************/
/*                    section initialize module                      */
/*********************************************************************/
   extern int *_B_BGN,*_B_END,*_D_BGN,*_D_END,*_D_ROM;

   void _INITSCT(void)
   {
          register int *p,*q;

          for (p=_B_BGN; p<_B_END; p++)
                 *p=0;

          for (p=_D_BGN; q=_D_ROM, p<_D_END; p++,q++)
                 *p=*q;
   }

/********************************************************************* /
/*                  peripheral initialize module                     * /
/********************************************************************* /
   void init_peripheral(void)
   {
          INTC.IPRA.WORD = 0x3000;             /*  IPRA 初期化      */
          ITU0.TCR.BYTE = 0x02;                /*  TCR0 初期化      */
          TCSR_FRT = 0x5A01;                   /*  TCSR 初期化      */
          PB.DR.WORD = 0x80;                   /*  PORT 初期化      */
   }
```

図 2.14 初期化プログラム（C 言語部分）

セクション初期化モジュール_INITSCT では、sct.src で指定されたセクションアドレスに基づいて B セクションのゼロクリア、ROM 上の初期化データ領域の RAM 上への複写を行います。型指定子に int を用いていますが、サイズが 4n バイト以外のときには、char を使用してください。

内部レジスタ初期化モジュール init_peripheral では、それぞれ次の設定を行います。

- 割り込み優先レベル設定レジスタ A で IRQ0 の割り込み優先レベルを 3 に設定
- タイマコントロールレジスタ 0 で 16 ビットインテグレーテッドタイマパルスユニットのタイマカウンタ 0 のクリア禁止、立ち上がりエッジでカウント、内部クロックをφ／4 でカウントに設定
- ウォッチドックタイマのタイマカウンタを 0x01 に設定
- ポート B に 0x80 を設定

2.2.5 割り込み関数の作成

割り込み関数を**図** 2.15 に示します。外部割り込み処理関数 IRQ0 とトラップ命令関数 inv_inst を定義します。

```
/************************************************************************/
/*                          file name "int.c"                           */
/************************************************************************/
   #include "7032.h"
   #include "sample.h"
   Extern const short padata;                      /* Cセクション   */
   extern short a;                                 /* Dセクション   */
   extern int work;                                /* Bセクション   */
   #pragma interrupt(IRQ0, inv_inst)

/************************************************************************/
/*                          interrupt module IRQ0                       */
/************************************************************************/
   void IRQ0(void)
   {
         a = PB.DR.WORD;
         PC.DR.BYTE = padata;
   }

/************************************************************************/
/*                       interrupt module inv_inst                      */
/************************************************************************/
   void inv_inst(void)
   {
         return;
   }
```

図 2.15 **割り込み関数**

関数 IRQ0 では、IRQ0 外部割り込みが発生すると大域変数 a に PB.DR.WORD (0x80)を設定します。これにより、CPU は低消費電力状態に入ります。

2.2.6 ロードモジュール用バッチファイルの作成

Sタイプ形式ロードモジュール(sample.mot)を作成するためのバッチファイルを**図** 2.16 に示します。

```
shc△-debug△sample.c△init.c△int.c
#C 言語プログラムのコンパイル
asmsh△sct.src△-debug
#アセンブリ言語プログラムのアセンブル
shc△-debug△-section=c=VECT△vect.c
#ベクタテーブル作成プログラムのコンパイル
optlnk△-noopt△-subcommand=rom.sub
#サブコマンドファイルを用いてリンク
rm△*.obj
#中間ファイルの削除
```

図 2.16 ロードモジュール作成バッチファイル

vect.c は単独ファイルでコンパイルし、オプション section=VECT を付けて他の初期化データ領域と異なるセクションになるようにします。そして、リンク時にアドレス 0 から割り付けます。

2.2.7 リンケージエディタのサブコマンドファイルの作成

ロードモジュール作成の際に使用するリンケージエディタのサブコマンドファイル（ファイル名:rom.sub）を**図** 2.17 に示します。

```
Sdebug
input     sample,init,int,vect,sct
                        ;  入力ファイルを指定
library    /user/unix/SHCV5.0/shclib.lib
                        ;  標準ライブラリを指定
output     sample.abs   ;  出力ファイル名を指定
rom       D=R           ;  ROM 化支援オプションを指定
start     VECT/0,P,C,D/0400,R,B/F0000000
                        ;  各セクションの先頭アドレスを指定
                        ;  セクション VECT を 0 番地から割り付け
                        ;  セクション P,C,D を H '400 番地から順に割り付け
                        ;  セクション R,B を F0000000 番地から順に割り付け
form      s             ;  S タイプ形式を指定
list      sample.map    ;  メモリマップ情報の出力を指定
Exit
```

図 2.17 リンケージエディタのサブコマンドファイル

2.3 サンプルプログラムの紹介(SH-3,SH3-DSP,SH-4,SH-4A,SH4AL-DSP)

ここでは、SH7708を例にとり、サンプルプログラムを紹介します。ここで紹介するプログラムの趣旨は、「RESETからmain()までのプログラム」です。CPUを動作させるため行う、最低限のプログラム例を紹介します。

2.3.1 割り込みハンドラの作成

SH-3,SH3-DSP,SH-4,SH-4A,SH4AL-DSPではSH-1,SH-2,SH-2E,SH-2A,SH2A-FPU,SH2-DSPと異なり、基本的に割り込み発生時のベクタ制御をソフトウェアで記述する必要があります。

SH-3の割り込みは、大きく分けて、リセット、例外、割り込みの3つの原因によりPC（プログラムカウンタ）が固定のアドレスにセットされます。よって、それぞれのアドレスに割り込みの要因判定と、各要因ごとの割り込み処理への分岐を割り込みハンドラとして記述しなければなりません。

具体的に各ハンドラを説明します。ここでは、VBR（ベクタベースレジスタ）は、H'00000000固定、MMU（メモリマネジメントユニット）未使用として例を示します。

(1) リセットハンドラ(H'00000000番地)

パワーオン、マニュアルリセット時は、PCは、H'A0000000にセットします。物理アドレスがH'00000000とH'A0000000が同一であるため、H'00000000にプログラムを配置します。ここでは、

- EXEVT判定による、例外事象の判定
- ベクタテーブルからの処理ルーチンの呼び出し

を行います。この処理を**図**2.18に示します。

```
;**********************************************************************
;   file name "reset.src"
;**********************************************************************
;  SH7708 Reset handler Routine
          .IMPORT      _vecttbl
          .IMPORT      _stacktbl
          .SECTION  VECT,CODE,LOCATE＝H’0
__reset:
;*************************************************************
;      You should initialize the stack RAM area by BSC
;       before set the stack pointer "R15"
;*************************************************************
; exsample )  AREA1 (CS1) -> STACK RAM
; AREA1
;    Bus size ->16bit
;    D23-D16  ->not PORT
;    wait 3 state
; BCR2>> PORTEN:A1SZ0:A1SZ0
;            0:    1   :0
;     >> BCR2=0x3fff8
          MOV.L     BSCR2,R0
          MOV.L     #H'3fff8,R1
          MOV.W     R1,@R0
;    WCR2>> A1-2W1:A1-2W0
;            1:    1
;       >> WCR2=0xffff
          MOV.L     WCR2,R0
          MOV.L     #H'ffff,R1
          MOV.W     R1,@R0
;*************************************************************
          MOV.L     VECTadr,R1
          MOV.L     STACKadr,R2
          MOV.L     EXPEVT,R0
          MOV.L     @R0,R0
          CMP/EQ    #0,R0        ;POWER ON RESET
          BT        PON_RESET
          CMP/EQ    #H'20,R0
          BT        MANUAL_RESET
          ; if( EXPEVT != RESET)
          ; while(1);
LOOP
          BRA       LOOP
          NOP
PON_RESET
```

```
          MOV.L     @(0,R1),R1   ;set function
          MOV.L     @(0,R2),R15  ;set stack pointer
          JMP       @R1
          NOP
MANUAL_RESET
          MOV.L     @(4,R1),R1   ;set function
          MOV.L     @(4,R2),R15  ;set stack pointer
          JMP       @R1
          NOP
;
          .ALIGN       4
VECTadr   .DATA.L      _vecttbl
STACKadr  .DATA.L      _stacktbl
EXPEVT    .DATA.L      H'ffffffd4
BSCR2     .DATA.L      H'ffffff62
WCR2      .DATA.L      H'ffffff66
          .END
```

図 2.18　セットハンドラプログラム

(2)　一般例外処理ハンドラ（VBR+H'100）

- EXPEVT から例外の要因コードを読み取ります。
- ベクタテーブルから、この要因の処理関数（ベクタ関数）を読み取ります。
- ターミネートルーチンをセットします。
- ベクタ関数にジャンプします。

このとき、ベクタ関数へのジャンプのために RTE 命令を使用しています。また、ベクタ関数からの戻りがターミネートルーチンになるように、ベクタ関数にジャンプする直前に PR レジスタの値を変更しています。ベクタ関数処理中の PR がターミネートルーチンであるため、ベクタ関数は RTS で復帰する必要があります。そのため、ベクタ関数を定義する際には「#pragma interrupt」を使用しないでください。

(3)　VBR+H'400　TLB ミス例外ハンドラ

MMU は未使用のため、ここは記述しません。

(4)　VBR+H'600　割り込みハンドラ

- INTEVT から割り込みの要因コードを読み取ります。
- ベクタテーブルから、この要因の処理関数（ベクタ関数）を読み取ります。
- 割り込みマスクテーブルから、この要因の割り込みマスクレベルをセットします。
- ターミネートルーチンをセットします。
- ベクタ関数にジャンプします。

この処理は、基本的に一般例外ハンドラと同じで、ベクタ関数からの戻りはターミネートルーチンになります。

```
;*****************************************************************
;   FILE        :vhandler.src
;*****************************************************************
      .include "env.inc"
      .include "vect.inc"

IMASKclr:    .equu  H'FFFFFF0F
RBBLclr:     .equ  H'CFFFFFFF
MDRBBLset:   .equ  H'70000000
      .import      _RESET_Vectors
      .import      _INT_Vectors
      .import      _INT_MASK
;*****************************************************************
;*          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
       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+,pr
       ldc.l  @r15+,spc
       ldc.l  @r15+,ssr
             .endm
;*****************************************************************
;      reset
;*****************************************************************
      .section  RSTHandler,code
_ResetHandler:
            mov.l #EXPEVT,r0
            mov.l @r0,r0
            shlr2 r0
            shlr  r0
            mov.l #_RESET_Vectors,r1
            add   r1,r0
            mov.l @r0,r0
            jmp   @r0
            nop
;*****************************************************************
;      exceptional interrupt
;*****************************************************************
      .section  INTHandler,code
      .export   INTHandlerPRG
INTHandlerPRG:
_ExpHandler:
            PUSH_EXP_BASE_REG
;
            mov.l #EXPEVT,r0          ; set event address
            mov.l @r0,r1              ; set exception code
            mov.l #_INT_Vectors,r0    ; set vector table address
            add   #-(h'40),r1         ; exception code - h'40
            shlr2 r1
            shlr     r1
            mov.l @(r0,r1),r3         ; set interrupt function addr
;
```

```
            mov.l  #_INT_MASK,r0      ; interrupt mask table addr
            shlr2  r1
            mov.b  @(r0,r1),r1        ; interrupt mask
            extu.b r1,r1
;
            stc    sr,r0              ; save sr
            mov.l  #(RBBLclr&IMASKclr),r2   ; RB,BL,mask clear data
            and    r2,r0              ; clear mask data
            or     r1,r0              ; set interrupt mask
            ldc    r0,ssr             ; set current status
;
            ldc.l  r3,spc
            mov.l  #__int_term,r0     ; set interrupt terminate
            lds    r0,pr
;
            rte
            nop
;
            .pool
;
;********************************************************************
;      Interrupt terminate
;********************************************************************
            .align 4
__int_term:
            mov.l   #MDRBBLset,r0  ; set MD,BL,RB
            ldc.l  r0,sr           ;
            POP_EXP_BASE_REG
            rte                    ; return
            nop
;
            .pool
;
;********************************************************************
;      TLB miss interrupt
;********************************************************************
       .org  H'300
_TLBmissHandler:
            PUSH_EXP_BASE_REG
;
            mov.l  #EXPEVT,r0         ; set event address
            mov.l  @r0,r1             ; set exception code
            mov.l  #_INT_Vectors,r0   ; set vector table address
            add    #-(h'40),r1        ; exception code - h'40
            shlr2  r1
            shlr   r1
            mov.l  @(r0,r1),r3        ; set interrupt function addr
;
            mov.l  #_INT_MASK,r0      ; interrupt mask table addr
            shlr2  r1
            mov.b  @(r0,r1),r1        ; interrupt mask
            extu.b r1,r1
;
            stc    sr,r0              ; save sr
            mov.l  #(RBBLclr&IMASKclr),r2   ; RB,BL,mask clear data
            and    r2,r0              ; clear mask data
            or     r1,r0              ; set interrupt mask
            ldc    r0,ssr             ; set current status
;
            ldc.l  r3,spc
            mov.l  #__int_term,r0     ; set interrupt terminate
            lds    r0,pr
;
            rte
            nop
;
            .pool
;
;********************************************************************
;      IRQ
;********************************************************************
       .org  H'500
_IRQHandler:
```

```
            PUSH_EXP_BASE_REG
;
            mov.l  #INTEVT,r0          ; set event address
            mov.l  @r0,r1              ; set exception code
            mov.l  #_INT_Vectors,r0    ; set vector table address
            add    #-(h'40),r1         ; exception code - h'40
            shlr2  r1
            shlr   r1
            mov.l  @(r0,r1),r3         ; set interrupt function addr
;
            mov.l  #_INT_MASK,r0       ; interrupt mask table addr
            shlr2  r1
            mov.b  @(r0,r1),r1         ; interrupt mask
            extu.b r1,r1
;
            stc    sr,r0               ; save sr
            mov.l  #(RBBLclr&IMASKclr),r2    ; RB,BL,mask clear data
            and    r2,r0               ; clear mask data
            or     r1,r0               ; set interrupt mask
            ldc    r0,ssr              ; set current status
;
            ldc.l  r3,spc
            mov.l  #__int_term,r0      ; set interrupt terminate
            lds    r0,pr
;
            rte
            nop
;
            .pool
            .end
```

図 2.19　割り込みハンドラプログラム

【注】 インクルードファイルの"env.inc"と"vect.inc"は HEW で SH3 のプロジェクト作成時に自動生成されます。

2.3.2 ベクタテーブルの作成

(1) **ベクタテーブル** <vect.c>

ここでは、ベクタテーブルの記述、割り込み優先度テーブル、TRAPA 関数テーブルを示します。ここでは、各要因の名前を登録し、実際のユーザ作成の関数名は、ヘッダファイル vect7708.h で記述します。

```
/****************************************************************/
/*                FILE NAME "vect.c"                            */
/****************************************************************/
#include "vect7708.h"

/****************************************************************/
/*                ALLOCATE STACK AREA                           */
/****************************************************************/
#pragma section STK   /* SECTION name "BSTK" */
long stack[STACK_SIZE];
#pragma section
/****************************************************************/
/*                 ALLOCATE DEFINITION TABLE                     */
/****************************************************************/
const void *stacktbl[]={
   STACK_PON,
   STACK_MANUAL
};
/****************************************************************/
/* ALLOCATE VECTOR TABLE   (EXPEVT or INTEVT CODE H'000-H'5a0)  */
/****************************************************************/
void (*const vecttbl[])(void) = {
                      /* EVT KIND     CODE     REG    */
   RESET_PON,         /* PON RESET    H'000    EXPEVT*/
   RESET_MANUAL,      /* MANUAL RESET H'020    EXPEVT*/
   TLB_MISS_READ,     /* TLB MISS(R)  H'040    EXPEVT*/
   TLB_MISS_WRITE,    /* TLB MISS(W)  H'060          */
   TLB_1ST_PAGE,      /*              H'080          */
   TLB_PROTECT_READ,  /*              H'0a0          */
   TLB_PROTECT_WRITE, /*              H'0c0          */
   ADR_ERROR_WRITE,   /*              H'0e0          */
   ADR_ERROR_WRITE,   /*              H'100          */
   RESERVED,          /*              H'120    ------*/
   RESERVED,          /*              H'140    ------*/
   TRAP,              /*              H'160 (with TRA)*/
   ILLEGAL_INST,      /*              H'180    EXPEVT*/
   ILLEGAL_SLOT,      /*              H'1a0    EXPEVT*/
   NMI,               /*              H'1c0    INTEVT*/
   USER_BREAK,        /*              H'1e0    EXPEVT*/
   IRQ15,             /*              H'200    INTEVT*/
   IRQ14,             /*              H'220    INTEVT*/
   IRQ13,             /*              H'240    INTEVT*/
   IRQ12,             /*              H'260    INTEVT*/
   IRQ11,             /*              H'280    INTEVT*/
   IRQ10,             /*              H'2a0    INTEVT*/
   IRQ9,              /*              H'2c0    INTEVT*/
   IRQ8,              /*              H'2e0    INTEVT*/
   IRQ7,              /*              H'300    INTEVT*/
   IRQ6,              /*              H'320    INTEVT*/
   IRQ5,              /*              H'340    INTEVT*/
   IRQ4,              /*              H'360    INTEVT*/
   IRQ3,              /*              H'380    INTEVT*/
   IRQ2,              /*              H'3a0    INTEVT*/
   IRQ1,              /*              H'3c0    INTEVT*/
   RESERVED,          /*              H'3e0    ------*/
   TMU0_TUNI0,        /*              H'400    INTEVT*/
   TMU1_TUNI1,        /*              H'420    INTEVT*/
   TMU2_TUNI2,        /*              H'440    INTEVT*/
   TMU2_TICPI2,       /*              H'460    INTEVT*/
   RTC_ATI,           /*              H'480    INTEVT*/
   RTC_PRI,           /*              H'4a0    INTEVT*/
   RTC_CUI,           /*              H'4c0    INTEVT*/
   SCI_ERI,           /*              H'4e0    INTRVT*/
   SCI_RXI,           /*              H'500    INTRVT*/
   SCI_TXI,           /*              H'520    INTRVT*/
```

```
   SCI_TEI,            /*              H'540     INTRVT*/
   WDT_ITI,            /*              H'560     INTEVT*/
   REF_RCMI,           /*              H'580     INTEVT*/
   DEF_RPVI,           /*              H'5a0     INTEVT*/
   RESERVED
};
/***************************************************************/
/* ALLOCATE INTERRUPT PRIORITY TABLE   INTEVT H'1c0-H'5a0       */
/***************************************************************/
const char imasktbl[]={
   15<<4,              /*     NMI level 16(IMASK=0-15)     */
   IP_RESERVED,        /*      ---------------             */
   15<<4,              /*     IRQ15 (IRL0000)              */
   14<<4,              /*     IRQ14 (IRL0001)              */
   13<<4,              /*     IRQ13 (IRL0010)              */
   12<<4,              /*     IRQ12 (IRL0011)              */
   11<<4,              /*     IRQ11 (IRL0100)              */
   10<<4,              /*     IRQ10 (IRL0101)              */
    9<<4,              /*     IRQ9  (IRL0110)              */
    8<<4,              /*     IRQ8  (IRL0111)              */
    7<<4,              /*     IRQ7  (IRL1000)              */
    6<<4,              /*     IRQ6  (IRL1001)              */
    5<<4,              /*     IRQ5  (IRL1010)              */
    4<<4,              /*     IRQ4  (IRL1011)              */
    3<<4,              /*     IRQ3  (IRL1100)              */
    2<<4,              /*     IRQ2  (IRL1101)              */
    1<<4,              /*     IRQ1  (IRL1110)              */
   IP_RESERVED,        /*     ---------------              */
   IP_TMU0,            /*     TMU0 TUNI0                   */
   IP_TMU1,            /*     TMU1 TUNI1                   */
   IP_TMU2,            /*     TNU2 TUNI2                   */
   IP_TMU2,            /*          TICPI2                  */
   IP_RTC,             /*     RTC  ATI                     */
   IP_RTC,
   IP_RTC,
   IP_SCI,             /*     SCI  ERI                     */
   IP_SCI,
   IP_SCI,
   IP_SCI,
   IP_WDT,             /*     WDT  ITI                     */
   IP_REF,             /*     REF  RCMI                    */
   IP_REF,             /*     REF  ROVI                    */
   IP_RESERVED
};

void (*const trap tbl[])(void)={
   TRAPA_0,
   TRAPA_1,
   TRAPA_2,
   TRAPA_3,
   TRAPA_4,
   TRAPA_5,
   TRAPA_6,
   TRAPA_7,
   TRAPA_8,
   TRAPA_9,
   TRAPA_10,
   TRAPA_11,
   TRAPA_12,
   TRAPA_13,
   TRAPA_14,
   TRAPA_15
};
```

図 2.20 ベクタテーブル

RENESAS

(2) ベクタ関数登録 <vect7708.h>

ここでは、実際のユーザ定義の関数名などを設定します。割り込み処理関数を追加する場合はここを変更します。
ここでの処理は、

- スタックサイズの定義
- 各要因のベクタ関数名の定義
- 割り込み優先度の設定（これは、IPRA,IPRB に設定する値）

となります。使用しないベクタは、ここでは halt という関数を定義しています。ユーザ関数自体の定義は、#pragma interrupt 宣言を使って割り込み関数として定義する必要があります。また、関数を登録した場合は、その関数の extern 宣言も必ずこのファイルで行ってください。

```
/*****************************************************************/
/*                FILE NAME  "vect7708.h"                        */
/*****************************************************************/


/*****************************************************************/
/*                  STACK SIZE definition                        */
/*****************************************************************/
#define STACK_SIZE       (0x4096/4)   /* 4096 byte */
#define STACK_PON        (&stack[STACK_SIZE])
#define STACK_MANUAL     (&stack[STACK_SIZE])
extern long stack[];

/*****************************************************************/
/*                RESET FUNCTION definition                      */
/*****************************************************************/
#define       RESET_PON      init      /* PON RESET H'000     EXPEVT*/

#define       RESET_MANUAL   init      /* MANUAL RESET H'020  EXPEVT*/
/******************************************************************/
/*                INTERRUPT FUNCTION definition                   */
/******************************************************************/
#define   TLB_MISS_READ      halt      /* TLB MISS(R)  H'040  EXPEVT*/
#define   TLB_MISS_WRITE     halt      /* TLB MISS(W)  H'060  EXPEVT*/
#define   TLB_1ST_PAGE       halt      /*           H'080     EXPEVT*/
#define   TLB_PROTECT_READ   halt      /*           H'0a0     EXPEVT*/
#define   TLB_PROTECT_WRITE  halt      /*           H'0c0     EXPEVT*/
#define   ADR_ERROR_WRITE    halt      /*           H'0e0     EXPEVT*/
#define   ADR_ERROR_WRITE    halt      /*           H'100     EXPEVT*/
/*#define RESERVED           halt */   /*           H'120     ------*/
/*#define RESERVED           halt */   /*           H'140     ------*/
#define   TRAP               _trap     /*           H'160 (with TRA)*/
#define   ILLEGAL_INST       halt      /*           H'180     EXPEVT*/
#define   ILLEGAL_SLOT       halt      /*           H'1a0     EXPEVT*/
#define   NMI                halt      /*           H'1c0     INTEVT*/
#define   USER_BREAK         halt      /*           H'1e0     EXPEVT*/
#define   IRQ15              irq15     /*           H'200     INTEVT*/
#define   IRQ14              halt      /*           H'220     INTEVT*/
#define   IRQ13              halt      /*           H'240     INTEVT*/
#define   IRQ12              halt      /*           H'260     INTEVT*/
#define   IRQ11              halt      /*           H'280     INTEVT*/
#define   IRQ10              halt      /*           H'2a0     INTEVT*/
#define   IRQ9               halt      /*           H'2c0     INTEVT*/
#define   IRQ8               halt      /*           H'2e0     INTEVT*/
#define   IRQ7               halt      /*           H'300     INTEVT*/
#define   IRQ6               halt      /*           H'320     INTEVT*/
#define   IRQ5               halt      /*           H'340     INTEVT*/
#define   IRQ4               halt      /*           H'360     INTEVT*/
#define   IRQ3               halt      /*           H'380     INTEVT*/
#define   IRQ2               halt      /*           H'3a0     INTEVT*/
#define   IRQ1               halt      /*           H'3c0     INTEVT*/
/*#define RESERVED           halt  */  /*           H'3e0     ------*/
#define   TMU0_TUNI0         halt      /*           H'400     INTEVT*/
#define   TMU1_TUNI1         halt      /*           H'420     INTEVT*/
#define   TMU2_TUNI2         halt      /*           H'440     INTEVT*/
#define   TMU2_TICPI2        halt      /*           H'460     INTEVT*/
#define   RTC_ATI            halt      /*           H'480     INTEVT*/
#define   RTC_PRI            halt      /*           H'4a0     INTEVT*/
#define   RTC_CUI            halt      /*           H'4c0     INTEVT*/
```

```
#define   SCI_ERI            halt      /*          H'4e0     INTRVT*/
#define   SCI_RXI            halt      /*          H'500     INTRVT*/
#define   SCI_TXI            halt      /*          H'520     INTRVT*/
#define   SCI_TEI            halt      /*          H'540     INTRVT*/
#define   WDT_ITI            halt      /*          H'560     INTEVT*/
#define   REF_RCMI           halt      /*          H'580     INTEVT*/
#define   DEF_RPVI           halt      /*          H'5a0     INTEVT*/
#define   RESERVED           halt
extern void init(void);
extern void halt(void);
extern void _trap(void);
extern void irq15(void);

/*********************************************************************/
/*                 INTERRUPT MASK definition                         */
/*********************************************************************/
#define IP_TMU0       (0<<4)
#define IP_TMU1       (0<<4)
#define IP_TMU2       (0<<4)
#define IP_RTC        (0<<4)
#define IP_SCI        (0<<4)
#define IP_WDT        (0<<4)
#define IP_REF        (0<<4)
#define IP_RESERVED   (15<<4)
/*********************************************************************/
/*                  IPRA,IPRB definition                              */
/*********************************************************************/
#define WORD_IPRA  ((IP_TMU0<<12)|(IP_TMU1<<8)|(IP_TMU2<<4)|IP_RTC)
#define WORD_IPRB  ((IP_WDT<<12)|(IP_REF<<8)|(IP_SCI<<4)|0)
extern void set_ip(void);
extern long stack[];

/*********************************************************************/
/*                 TRAPA system call definition                       */
/*********************************************************************/
#define TRAPA_0    halt
#define TRAPA_1    halt
#define TRAPA_2    halt
#define TRAPA_3    halt
#define TRAPA_4    halt
#define TRAPA_5    halt
#define TRAPA_6    halt
#define TRAPA_7    halt
#define TRAPA_8    halt
#define TRAPA_9    halt
#define TRAPA_10   halt
#define TRAPA_11   halt
#define TRAPA_12   halt
#define TRAPA_13   halt
#define TRAPA_14   halt
#define TRAPA_15   halt  /*#15(#0F) should be Exception routine(Illegal use )*/
```

図 2.21 ベクタ関数名定義

2.3.3 ヘッダファイルの作成

サンプルプログラムで共通に使用するヘッダファイルを示します。

```
/************************************************************************/
/*                  file name "7700s.h" (抜粋)                          */
/************************************************************************/
struct st_intc {                                        /*  struct INTC */
            union {                                     /*    ICR       */
                 unsigned short WORD;                   /*  Byte Access */
                 struct {                               /*  Bit  Access */
                       unsigned short NMIL  :1;         /*    NMIL      */
                       unsigned short       :6;         /*              */
                       unsigned short NMIE  :1;         /*    NMIE      */
                       }      BIT;                      /*              */
                 }         ICR;                         /*              */
            union {                                     /*    IPRA      */
                 unsigned short WORD;                   /*  Word Access */
                 struct {                               /*  Bit  Access */
                       unsigned short UU:4;             /*    IRQ0      */
                       unsigned short UL:4;             /*    IRQ1      */
                       unsigned short LU:4;             /*    IRQ2      */
                       unsigned short LL:4;             /*    IRQ3      */
                       }      BIT;                      /*              */
                 }         IPRA;                        /*              */
            union {                                     /*    IPRB      */
                 unsigned short WORD;                   /*  Word Access */
                 struct {                               /*  Bit  Access */
                       unsigned short UU:4;             /*    IRQ4      */
                       unsigned short UL:4;             /*    IRQ5      */
                       unsigned short LU:4;             /*    IRQ6      */
                       unsigned short LL:4;             /*    IRQ7      */
                       }      BIT;                      /*              */
                 }         IPRB;                        /*              */
            char           wk1[234];                    /*              */
            unsigned int   TRA;                         /*    TRA       */
            unsigned int   EXPEVT;                      /*    EXPEVT    */
            unsigned int   INTEVT;                      /*    INTEVT    */
};                                                      /*              */

union un_ccr {                                          /*  union CCR   */
           unsigned int      LONG;                      /*  Long Access */
           struct {                                     /*  Bit  Access */
                 unsigned int      :26;                 /*              */
                 unsigned int RA   :1;                  /*    RA        */
                 unsigned int      :1;                  /*    0         */
                 unsigned int CF   :1;                  /*    CF        */
                 unsigned int CB   :1;                  /*    CB        */
                 unsigned int WT   :1;                  /*    WT        */
                 unsigned int CE   :1;                  /*    CE        */
                 }          BIT;                        /*              */
};                                                      /*              */
#define SCI   (*(volatile struct st_sci   *)0xFFFFFE80)   /* SCI  Address */
#define TMU   (*(volatile struct st_tmu   *)0xFFFFFE90)   /* TMU  Address */
#define TMU0  (*(volatile struct st_tmu0  *)0xFFFFFE94)   /* TMU0 Address */
#define TMU1  (*(volatile struct st_tmu0  *)0xFFFFFEA0)   /* TMU1 Address */
#define TMU2  (*(volatile struct st_tmu2  *)0xFFFFFEAC)   /* TMU2 Address */
#define RTC   (*(volatile struct st_rtc   *)0xFFFFFEC0)   /* RTC  Address */
#define INTC  (*(volatile struct st_intc  *)0xFFFFFEE0)   /* INTC Address */
#define BSC   (*(volatile struct st_bsc   *)0xFFFFFF60)   /* BSC  Address */
#define CPG   (*(volatile struct st_cpg   *)0xFFFFFF80)   /* CPG  Address */
#define UBC   (*(volatile struct st_ubc   *)0xFFFFFF90)   /* UBC  Address */
#define MMU   (*(volatile struct st_mmu   *)0xFFFFFFE0)   /* MMU  Address */
#define CCR   (*(volatile union  un_ccr   *)0xFFFFFFEC)   /* CCR  Address */
```

図 2.22 ヘッダファイル

RENESAS

2.3.4 初期化部の作成

リセット後、BSC の設定と、ポインタの設定を行い、初期化関数へ処理が移ります。

初期化関数では、割り込み優先度の設定、セクションの初期化などを行い、実際のユーザ関数の先頭へ処理を移します。

(1) **初期化関数** <init.c,cntrl.h>

- 割り込み優先度の設定
- キャッシュのフラッシュ
- キャッシュオン
- セクションの初期化
- 割り込みマスクの設定
- ユーザ関数への分岐

を行います。

```
/*****************************************************************/
/*                  file name "cntrl.h"                          */
/*****************************************************************/
#include <machine.h>
#include "7700s.h"
/*****************************************************************/
/*                  control BL ,MD bit                           */
/*****************************************************************/
#define BLoff()    set_cr((get_cr()&0xefffffff))
#define BLon()     set_cr((get_cr()|0x10000000))
#define USRmode()  set_cr((get_cr()|0x40000000))
/*****************************************************************/
/*                   cache control                               */
/*****************************************************************/
#define CacheON()     (CCR.BIT.CE=1)
#define CacheOFF()    (CCR.BIT.CE=0)
#define CacheFLASH()  (CCR.BIT.CF=1)
```

図 2.23 マクロ定義プログラム

```
/*****************************************************************/
/*                     file name "init.c"                        */
/*****************************************************************/
#include <machine.h>
#include "cntrl.h"
void init(void)
{
   set_ip();
   CacheOFF();
   CacheFLASH();
   CacheON();
   BLoff();        /* BLOCK BIT OFF          */

   _INITSCT();     /* section initialize     */
   set_imask(0);   /* interrput priority 0   */

   main();         /* User main() routine    */

   halt();         /* halt()                 */

}
```

図 2.24 初期化プログラム(1)

(a) 割り込み優先度の設定 <ipr.c>

vect7708.h で定義した、各割り込み要因の割り込み優先度を、実際に IPRA、IPRB に設定します。

```
/***********************************************************************/
/*                    file name "ipr.c"                                */
/***********************************************************************/
#include "7700s.h"
#include "vect7708.h"
void set_ip(void)
{
   INTC.IPRA.WORD=WORD_IPRA;
   INTC.IPRB.WORD=WORD_IPRB;
}
```

図 2.25 割り込み優先度設定プログラム

(b) セクションの初期化 <sect.src, initsct.c>

RAM に割り当てるセクションの初期化を行います。

未初期化データ B セクションは、0クリアします。初期値ありデータセクションは、ROM 上の D セクションから、RAM 上の R セクションへデータをコピーします。(initsct.c)

また、セクションの先頭アドレス、サイズを取得するためにアセンブラ記述が必要です。(sct.src)

```
;**********************************************************************
;                     file name "sct.src"
;**********************************************************************
          .SECTION  B,DATA,ALIGN=4
          .SECTION  R,DATA,ALIGN=4
          .SECTION  D,DATA,ALIGN=4
;  If other section are existed , Insert here ".SECTION XXX",
          .SECTION  C,DATA,ALIGN=4
__B_BGN:  .DATA.L      (STARTOF B)
__B_END:  .DATA.L      (STARTOF B)+(SIZEOF B)
__D_BGN:  .DATA.L      (STARTOF R)
__D_END:  .DATA.L      (STARTOF R)+(SIZEOF R)
__D_ROM:  .DATA.L      (STARTOF D)
          .EXPORT      __B_BGN
          .EXPORT      __B_END
          .EXPORT      __D_BGN
          .EXPORT      __D_END
          .EXPORT      __D_ROM
          .END
```

図 2.26 セクション定義プログラム

```
/***********************************************************************/
/*               file name "initsct.c"                                 */
/***********************************************************************/
extern int *_B_BGN,*_B_END,*_D_BGN,*_D_END,*_D_ROM;

  void _INITSCT(void)
  {
   register int *p, *q;
   for (p=_B_BGN; p<_B_END; p++){
      *p=0;
   }
   for (p=_D_BGN, q=_D_ROM; p<_D_END; p++, q++){
      *p=*q;
   }
}
```

図 2.27 セクション初期化プログラム

2.3.5　メイン処理部、割り込み処理部の作成

main()、halt()、irq15()の関数を作成すれば、上記プログラムはリンク可能になります。

```
void main(void)
{
   /* ユーザプログラムの記述 */
}
#pragma interrupt(halt, irq15)

void halt(void)
{
   while(1);     /* 異常処理時のルーチン          */
                 /* ここでは無限ループとしている   */
}
void irq15(void)
{
                 /* IRQ15 の処理プログラム       */
}
```

図 2.28　メインプログラム

2.3.6　ロードモジュール用バッチファイルの作成

S タイプ形式ロードモジュール(sample.mot)を作成するためのバッチファイルを図 2.29 に示します。

```
shc△-debug△-cpu=sh3△vect.c△init.c△ipr.c△initsct.c△main.c
                                #C 言語プログラムのコンパイル
asmsh△sct.src△-debug△-cpu=sh3
asmsh△intr.src△-debug△-cpu=sh3
asmsh△reset.src△-debug△-cpu=sh3
                                #アセンブリ言語プログラムのアセンブル
optlnk△-noopt△-subcommand=lnk.sub
                                #サブコマンドファイルを用いてリンク
rm△*.obj
                                #中間ファイルの削除
```

図 2.29　ロードモジュール作成バッチファイル

2.3.7 リンケージエディタのサブコマンドファイルの作成

ロードモジュール作成の際に使用するリンケージエディタのサブコマンドファイル（ファイル名:lnk.sub）**図** 2.30 に示します。

```
sdebug
input      vect, init, ipr, initsct, main, intr, sct, reset
                              ;  入力ファイルを指定
Library    /user/unix/SHCV50/shc3npb.lib
                              ;  標準ライブラリを指定
output     sample.abs         ;  出力ファイル名を指定
rom        D=R                ;  ROM 化支援オプションを指定
start      P,C,D/10000,R,B,BSTK/04000000
                              ;  各セクションの先頭アドレスを指定
                              ;  セクション VECT は絶対アドレスセクションのため
                              ;  アドレス指定しない。（0 番地に割り付け）
                              ;  セクション P,C,D を H'10000 番地から順に割り付ける
                              ;  セクション R,B を H’04000000 地から順に割り付ける
form       s                  ;  S タイプ形式を指定
list       sample.map         ;  メモリマップ情報の出力を指定
Exit
```

図 2.30　リンケージエディタのサブコマンドファイル

2.4　シミュレータデバッガを使用したデバッグ方法

「**2.1.1 シミュレータデバッガ用プロジェクトの作成**」で作成したプロジェクトを使用し、シミュレータデバッガを実行します。

2.4.1　コンフィグレーションの設定

［オプション］メニューから［ビルドの構成...］を選択し、利用可能な環境を表示してください。**図** 2.32 の画面で、使用する環境を選択してください。この場合は、現在のコンフィグレーションを[SimDebug SH-1]と選択します。

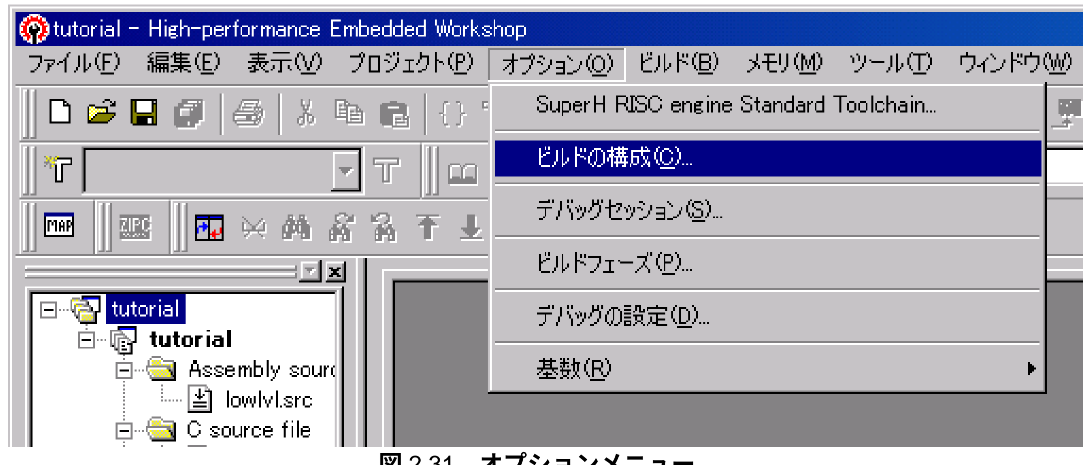


図 2.31　**オプションメニュー**

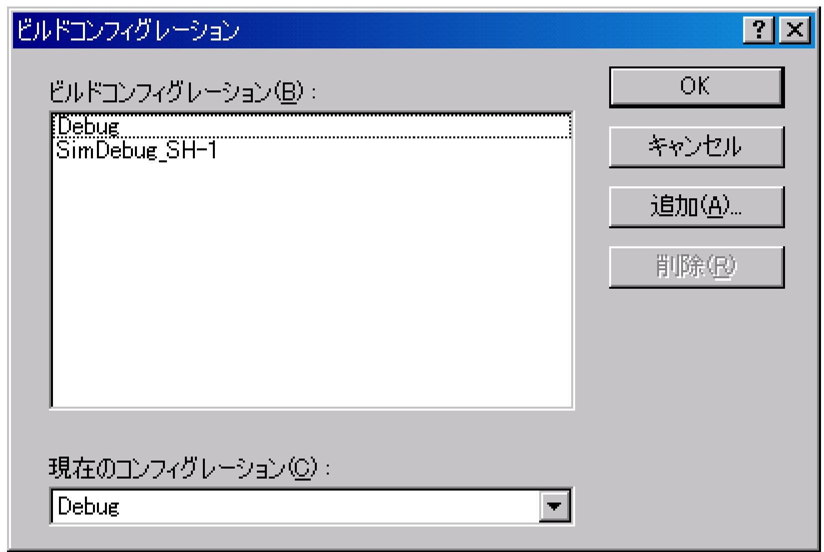


図 2.32　**ビルドコンフィグレーションダイアログボックス**

2.4.2 メモリリソースの確保

開発しているアプリケーションを動作させるためにメモリリソースの確保が必要です。デモンストレーションプロジェクトでは、自動的にメモリリソースを確保しますので、設定を確認してください。

［オプション］メニューから［シミュレータ-> メモリリソース...］を選択し、現在のメモリリソースを表示してください。

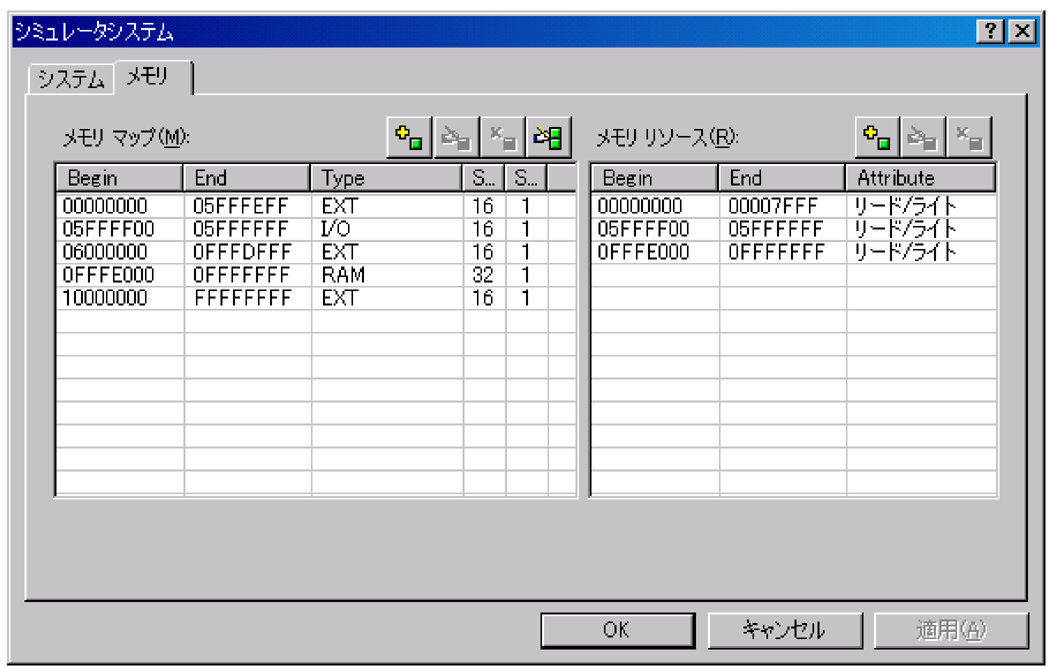


図 2.33 シミュレータ-> シミュレータシステムダイアログボックス

プログラム領域として H'00000000 から H'00007FFF を、スタック領域として H'0FFFE000 から H'0FFFFFFF を読み出し／書き込み可能領域として確保しています。

[OK]ボタンをクリックしてダイアログボックスを閉じてください。

メモリリソースは、SuperH RISC engine Standard Toolchain ダイアログボックスの[Simulator]タブでも参照／変更ができます。お互いの変更は反映されます。

2.4.3　サンプルプログラムのダウンロード

デモンストレーションプロジェクトでは、自動的にダウンロードするサンプルプログラムを設定しますので、設定を確認してください。

［オプション］メニューから［デバッグの設定...］を選択して、デバッグの設定ダイアログボックスを開いてください。

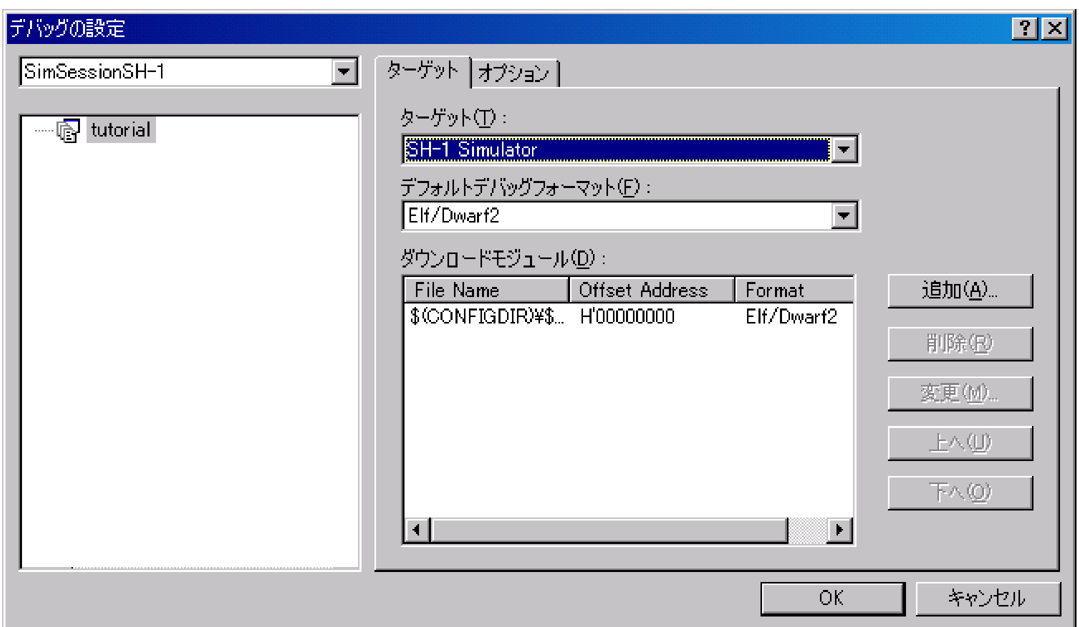


図 2.34　デバッグの設定 ダイアログボックス

［ダウンロードモジュール］に設定しているファイルがダウンロードするファイルです。

[OK]ボタンをクリックしてデバッグの設定 ダイアログボックスを閉じてください。

［デバッグ］メニューから［ダウンロード-> All Download Modules］を選択して、サンプルプログラムをダウンロードしてください。

RENESAS

2.4.4 I/O シミュレーション の設定

デモンストレーションプロジェクトでは、自動的に I/O シミュレーションを設定しますので、設定を確認してください。

［オプション］メニューから［シミュレータ->システム］を選択して、シミュレータシステムダイアログボックスを開いてください。

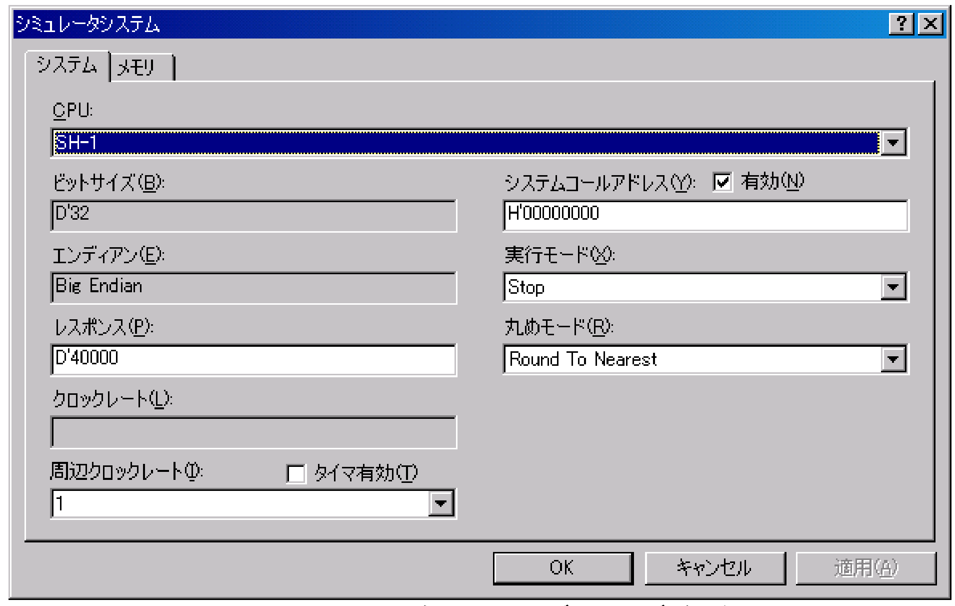


図 2.35 シミュレータシステム ダイアログボックス

［システムコールアドレス］の［有効］にチェックがあることを確認してください。

[OK]ボタンをクリックして I/O シミュレーションを有効にしてください。

［表示］メニューから［CPU->I/O シミュレーション］を選択して、I/O シミュレーションウィンドウを開いてください。

I/O シミュレーションウィンドウを開かなければ、I/O シミュレーションが有効になりません。

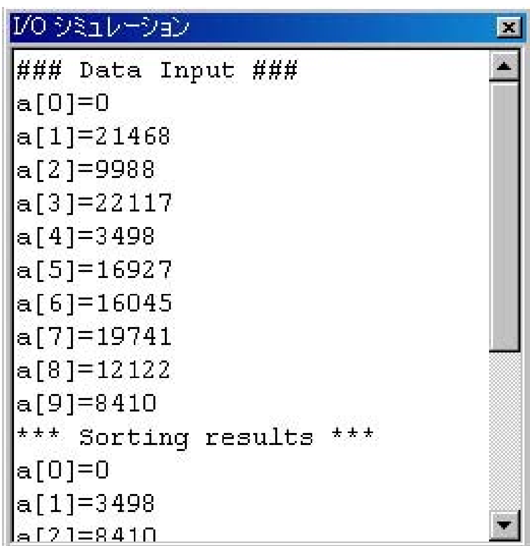


図 2.36 I/O シミュレーションウィンドウ

2.4.5　トレース情報取得条件の設定

［表示］メニューから［コード->トレース］を選択して、トレースウィンドウを開いてください。さらに、トレースウィンドウ上で右クリックしてポップアップメニューを表示し、［設定...］を選択してください。

以下のトレース取得ダイアログボックスを表示します。

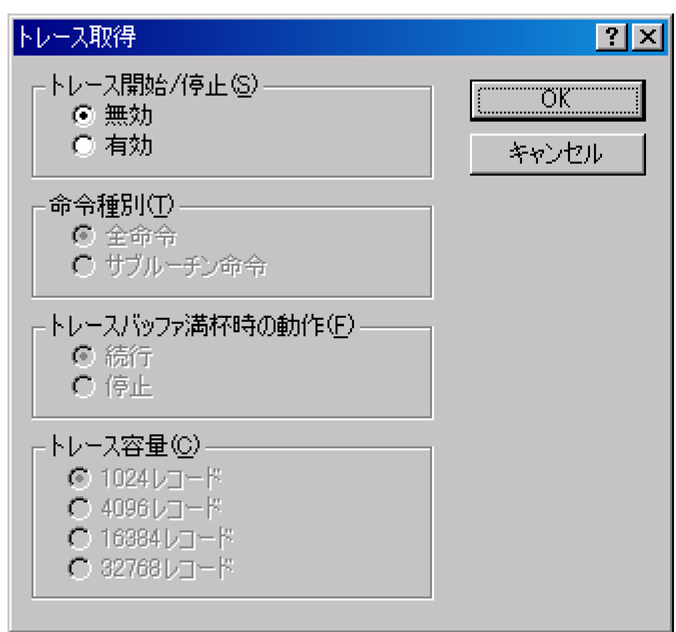


図 2.37　トレース取得ダイアログボックス

トレース取得ダイアログボックスの［トレース開始/停止］を［有効］に設定し、[OK]ボタンをクリックしてトレース情報取得を有効にしてください。

2.4.6　ステイタスウィンドウ

ステイタスウィンドウで停止要因が確認できます。

［表示］メニューから［CPU->ステイタス］を選択して、ステイタスウィンドウを開いてください。さらに、ステイタスウィンドウのうち[Platform]シートを表示してください。

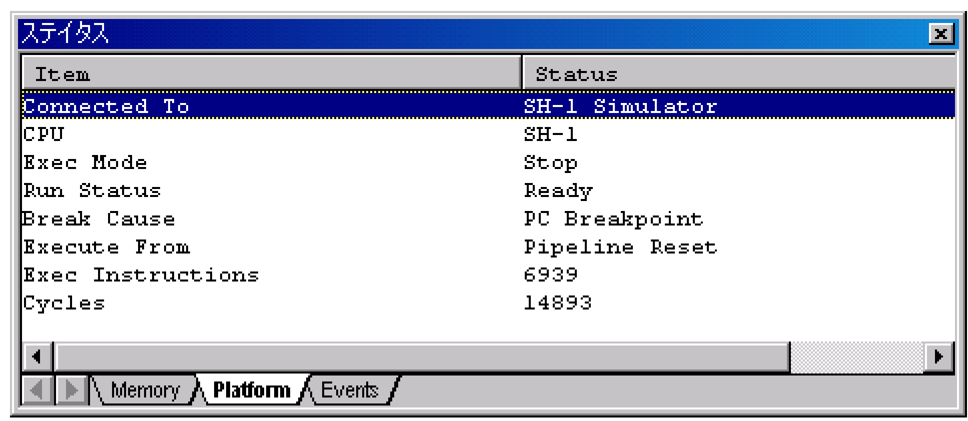


図 2.38　表示-> CPU->ステイタスウィンドウ

2.4.7 レジスタウィンドウ

レジスタウィンドウでレジスタの値が確認できます
［表示］メニューから［CPU->レジスタ］を選択してください。

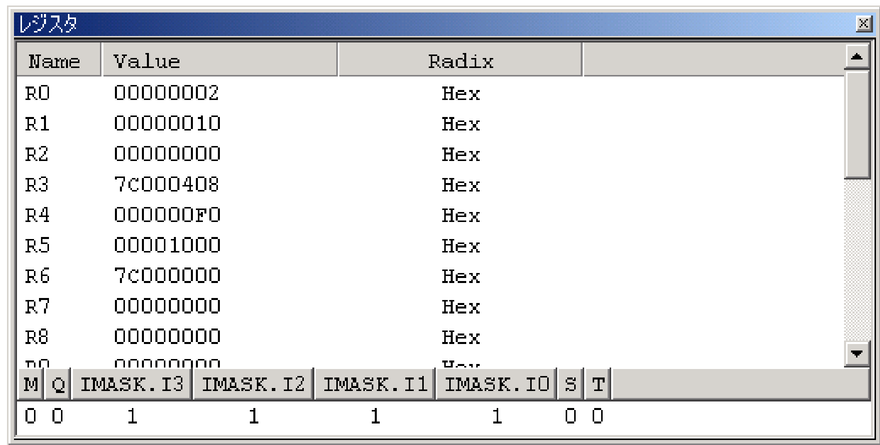


図 2.39 表示-> CPU->レジスタウィンドウ

2.4.8 トレース

(1) トレースバッファの使い方

トレースバッファを使って、命令実行の履歴を知ることができます。
［表示］メニューから［コード->トレース］を選択して、トレースウィンドウを開いてください。ウィンドウの最上部までスクロールアップしてください。

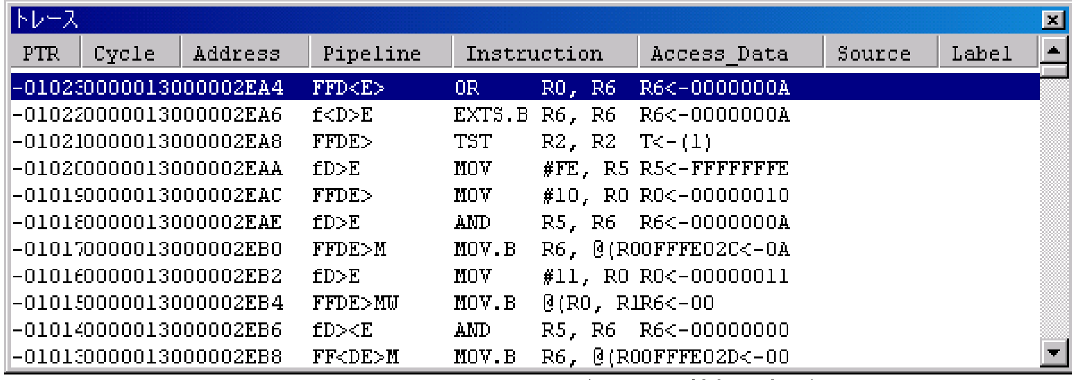


図 2.40 トレースウィンドウ（トレース情報の表示）

(2)　トレースサーチ

最初に、トレースウィンドウ上で右クリックしてポップアップメニューを表示し、［検索...］を選択して、トレース検索ダイアログボックスを開いてください。

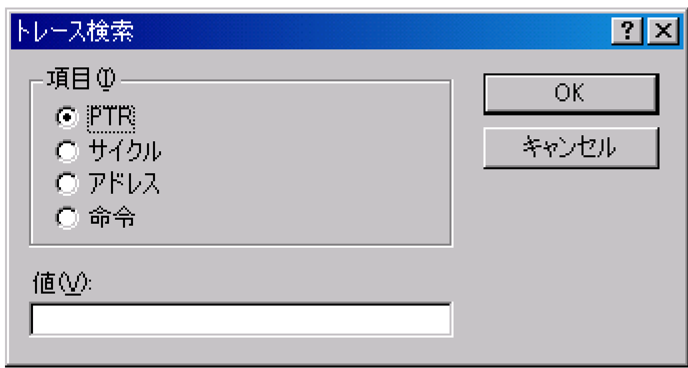


図 2.41　トレース検索ダイアログボックス

サーチ項目［項目］とサーチ内容［値］を設定して[OK]ボタンをクリックすると、トレースサーチを実行します。該当するトレース情報があった場合、該当する最初の行を強調表示します。同じサーチ内容［値］でトレースサーチを続ける場合は、トレースウィンドウ上で右クリックしてポップアップメニューを表示し、［次を検索］を選択してください。次に該当する行を強調表示します。

```
トレース
PTR   Cycle   Address   Pipeline   Instruction   Access_Data   Source   Label
-0092600000131000117C   FFDE>       CMP/HI R2, R13 T<-(1)
-0092500000131000117E   fD>E        BF     00001196T(1)
-00924000001310001180   FFDE>       MOV    R14, R2 R2<-0FFFE199
-00923000001310001182   fD>E        ADD    #01, R14R14<-0FFFE19A
-00922000001310001184   FFDE>MW     MOV.B  @R2, R4 R4<-20
-00921000001310001186   fD><E       EXTS.B R4, R4  R4<-00000020
-00920000001320001188   FF<DE>MMW   MOV.L  @(000001R3<-00001000
-0091900000132000118A   fD><<E      JSR    @R3      PC<-00001000
-0091800000132000118C   FF<<-D>E    NOP
-00917000001320001000   FFDE>MMW    MOV.L  @(000000R0<-0FFFE000            charput
-00916000001320001002   fD><<EM>>   MOV.B  R4, @R0 0FFFE000<-20
```

図 2.42　トレースウィンドウ

RENESAS

2.4.9 ブレークポイントの確認

プログラムに設定したすべてのブレークポイントのリストをイベントポイントウィンドウで確認することができます。
［表示］メニューから［コード->イベントポイント］を選択してください。

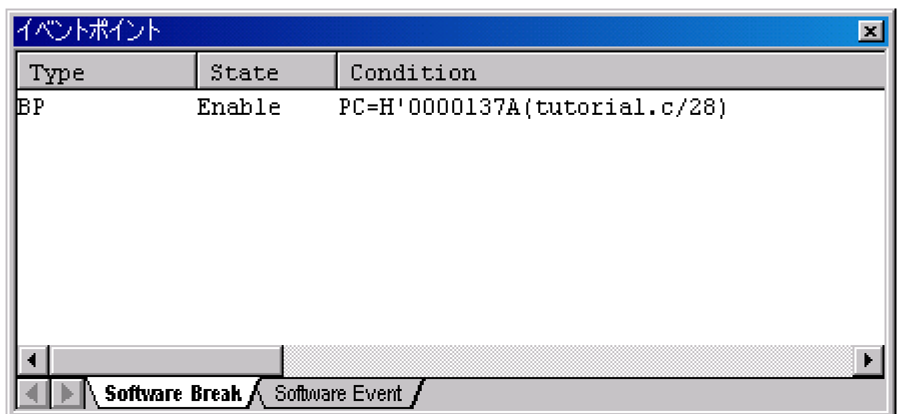


図 2.43 イベントポイントウィンドウ

イベントポイントウィンドウによって、ブレークポイントの設定、新しいブレークポイントの定義、およびブレイクポイントの表示ができます。
イベントポイントウィンドウを閉じてください。

2.4.10　メモリ内容の確認

メモリブロックの内容をメモリウィンドウで確認することができます。たとえば、バイトサイズで main 列に対応したメモリを確認する場合の手順を以下に示します。

［表示］メニューから［CPU->メモリ…］を選択し、［先頭アドレス］フィールドにメモリ領域の開始アドレス、［終了アドレス］フィールドに終了アドレスを入力してください。

図 2.44　表示形式ダイアログボックス

[OK]ボタンをクリックして、指定したメモリ領域を示すメモリウィンドウを開いてください。

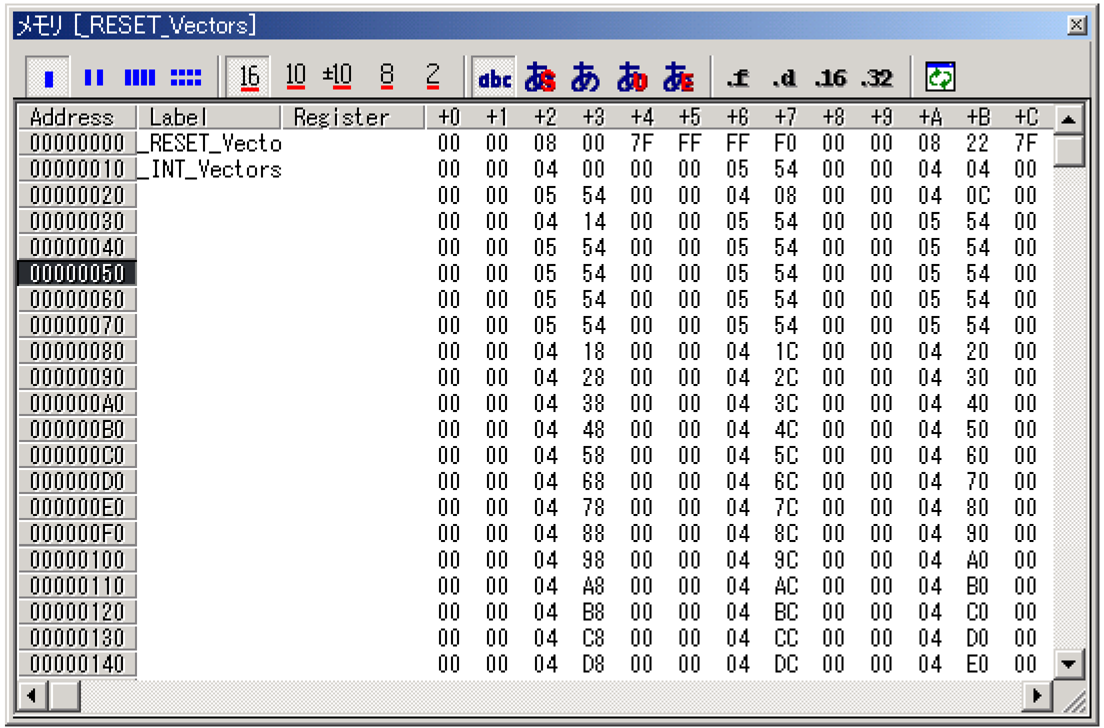


図 2.45　メモリウィンドウ

2.5 シミュレータデバッガでの標準入出力およびファイル入出力処理

シミュレータ・デバッガでは、デバッグ対象プログラムから標準入出力およびファイル入出力を行うことができます。入出力機能を利用する場合は、必ず I/O シミュレーション ウィンドウをオープンしておいてください。サポートしている入出力処理は、以下のとおりです。

表 2.3 入出力機能一覧

番号	機能コード	機能名	内容
1	H'21	GETC	標準入力からの１バイト入力
2	H'22	PUTC	標準出力への１バイト出力
3	H'23	GETS	標準入力からの１行入力
4	H'24	PUTS	標準出力への１行出力
5	H'25	FOPEN	ファイルのオープン
6	H'06	FCLOSE	ファイルのクローズ
7	H'27	FGETC	ファイルからの１バイト入力
8	H'28	FPUTC	ファイルへの１バイト出力
9	H'29	FGETS	ファイルからの１行入力
10	H'2A	FPUTS	ファイルへの１行出力
11	H'0B	FEOF	エンドオブファイルのチェック
12	H'0C	FSEEK	ファイルポインタの移動
13	H'0D	FTELL	ファイルポインタの現在位置を得る

この機能を実現するためには、まず入出力用の特定の位置をシミュレータシステム ダイアログボックスの［システムコールアドレス］で指定し、［有効］をチェック後、デバッグ対象プログラムを実行します。

シミュレータ・デバッガでは、デバッグ対象プログラムの命令を実行中に、指定した位置へのサブルーチン分岐命令（BSR、JSR、BSRF）すなわちシステムコール命令を検出すると、R0、R1 の内容をパラメータとして入出力処理を行います。

したがって、システムコールを行う前にデバッグ対象プログラムの中で次の設定をしておきます。

- R0 レジスタ：**表** 2.3 に示す機能コード

MSB **１バイト**	**１バイト**		LSB
H'01	**機能コード**	- - -	- - -

- R1 レジスタ：パラメータブロックのアドレス
  （パラメータブロックの内容は各機能の説明を参照してください。）

MSB ･･･ LSB
パラメータブロックのアドレス

- パラメータブロックおよび入出力バッファ領域の確保

なお、パラメータブロックの各パラメータにアクセスする場合は、該当するパラメータのサイズで入出力処理が終了すると、システムコール命令の次の命令からシミュレーションを再開します。

【注】 JSR、BSR、BSRF **命令をシステムコール命令として実行すると、**JSR、BSR、BSRF **の次命令はスロット命令ではなく、通常の命令として実行します。このため、スロット命令と通常命令で実行結果に違いが発生する命令をシステムコール用** JSR、BSR、BSRF **命令の次命令には記述しないでください。**

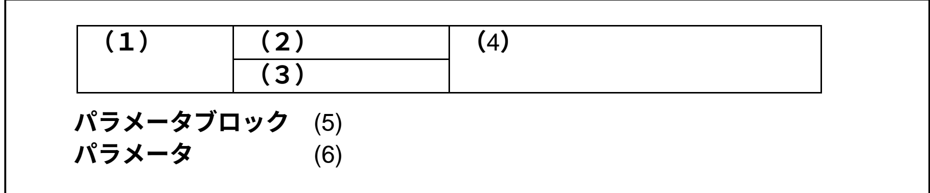


図 2.46 入出力機能の説明形式

各項目の内容は、以下のとおりです。

(1) 表2.3に対応する番号
(2) 機能名
(3) 機能コード
(4) 入出力の機能
(5) 入出力のパラメータブロック
(6) 入出力のパラメータ

1	GETC	標準入力からの 1 バイト入力
	H'21	

【パラメータブロック】

	1 バイト	1 バイト
+0	入力バッファ先頭アドレス	
+2		

【パラメータ】 ・入力バッファ先頭アドレス(入力)
入力データを書き込むバッファの先頭アドレス

2	PUTC	標準出力への1バイト出力
	H'22	

【パラメータブロック】

	1バイト	1バイト
+0	出力バッファ先頭アドレス	
+2		

【パラメータ】 ・出力バッファ先頭アドレス(入力)
出力データを格納しているバッファの先頭アドレス

3	GETS	標準入力からの1行入力
	H'23	

【パラメータブロック】

	1バイト	1バイト
+0	入力バッファ先頭アドレス	
+2		

【パラメータ】 ・入力バッファ先頭アドレス(入力)
入力データを書き込むバッファの先頭アドレス

4	PUTS	標準出力への1行出力
	H'24	

【パラメータブロック】

	1バイト	1バイト
+0	出力バッファ先頭アドレス	
+2		

【パラメータ】 ・出力バッファ先頭アドレス（入力）
出力データを格納しているバッファの先頭アドレス

5	FOPEN	ファイルのオープン
	H'25	

FOPENによってファイルをオープンすると、ファイル番号を返します。以後のファイル入出力、ファイルクローズなどでは、このファイル番号を用います。同時にオープンできる最大ファイル数は256です。

【パラメータブロック】

	1バイト	1バイト
+0	実行結果	ファイル番号
+2	オープンモード	未使用
+4	ファイル名先頭アドレス	
+6		

【パラメータ】

・実行結果（出力）
0 正常終了
–1 エラー
・ファイル番号（出力）
オープン処理以降のファイルアクセスで使用する番号
・オープンモード（入力）
H'00 "r"
H'01 "w"
H'02 "a"
H'03 "r+"
H'04 "w+"
H'05 "a+"
H'10 "rb"
H'11 "wb"
H'12 "ab"
H'13 "r+b"
H'14 "w+b"
H'15 "a+b"
各モードの内容は以下のとおりです。
"r" 読み出し用にオープンする。
"w" 空ファイルを書き込み用にオープンする。
"a" ファイルの最後から書き込み用にオープンする。
"r+" 読み出し，書き込み用にオープンする。
"w+" 空ファイルを読み出し，書き込み用にオープンする。
"a+" 読み出し追加用にオープンする。
"b" バイナリモードでオープンする。
・ファイル名先頭アドレス（入力）
ファイル名を格納している領域の先頭アドレス

6	FCLOSE	ファイルのクローズ
	H'06	

【パラメータブロック】

	1バイト	1バイト
+0	実行結果	ファイル番号

【パラメータ】

・実行結果（出力）
0 正常終了
–1 エラー
・ファイル番号（入力）
ファイルオープン時に返す番号

7	FGETC H'27	ファイルから 1 バイトのデータ読み出し

【パラメータブロック】

	1バイト	1バイト
+0	実行結果	ファイル番号
+2	未使用	
+4 +6	入力バッファ先頭アドレス	

【パラメータ】

・実行結果（出力）
0 正常終了
–1 エラー
・ファイル番号（入力）
ファイルオープン時に返す番号
・入力バッファ先頭アドレス（入力）
入力データを書き込むバッファの先頭アドレス

8	FPUTC H'28	ファイルへ 1 バイトのデータ書き込み

【パラメータブロック】

	1バイト	1バイト
+0	実行結果	ファイル番号
+2	未使用	
+4 +6	出力バッファ先頭アドレス	

【パラメータ】

・実行結果（出力）
0 正常終了
–1 エラー
・ファイル番号（入力）
ファイルオープン時に返す番号
・入力バッファ先頭アドレス（入力）
出力データを格納しているバッファの先頭アドレス

9	FGETC H'29	ファイルからの文字列データの読み出し

【パラメータブロック】

	1バイト	1バイト
+0	実行結果	ファイル番号
+2	未使用	
+4 +6	入力バッファ先頭アドレス	

【パラメータ】
・実行結果（出力）
0 正常終了
–1 EOF 検出
・ファイル番号（入力）
ファイルオープン時に返す番号
・バッファサイズ（入力）
データを格納する領域のサイズ
（バイト単位で最大 256 バイトまで）
・入力バッファ先頭アドレス（入力）
入力データを格納しているバッファの先頭アドレス

10	FPUTC H'29	**ファイルへの文字列データ書き込み**

【パラメータブロック】

	1バイト	**1バイト**
+0	**実行結果**	**ファイル番号**
+2	**未使用**	
+4	**入力バッファ先頭アドレス**	
+6		

【パラメータ】
・実行結果（出力）
0 正常終了
–1 エラー
・ファイル番号（入力）
ファイルオープン時に返す番号
・出力バッファ先頭アドレス（入力）
出力データを格納しているバッファの先頭アドレス

11	FEOF H'0B	**エンドオブファイルのチェック**

【パラメータブロック】

	1バイト	**1バイト**
+0	**実行結果**	**ファイル番号**

【パラメータ】
・実行結果（出力）
0 EOF ではない
–1 EOF 検出
・ファイル番号（入力）
ファイルオープン時に返す番号

12	FSEEK H'0C	**指定位置にファイルポインタを移動**

【パラメータブロック】

	1バイト	**1バイト**
+0	**実行結果**	**ファイル番号**
+2	**ディレクション**	**未使用**
+4	**オフセット**	
+6		

【パラメータ】 ・実行結果（出力）
0 正常終了
–1 エラー
・ファイル番号（入力）
ファイルオープン時に返す番号
・ディレクション（入力）
0 オフセットはファイルの先頭からのバイト数
1 オフセットは現在のファイルポインタからのバイト数
2 オフセットはファイルの最後尾からのバイト数
・オフセット（入力）
ディレクションで指定した位置からのバイト数

13	FTEEL	**ファイルポインタの現在位置を調査**
	H'0D	

【パラメータブロック】

	1バイト	**1バイト**
+0	**実行結果**	**ファイル番号**
+2	**未使用**	
+4	**オフセット**	
+6		

【パラメータ】 ・実行結果（出力）
0 正常終了
–1 エラー
・ファイル番号（入力）
ファイルオープン時に返す番号
・オフセット（出力）
現在のファイルポインタの位置
（ファイル先頭からのバイト数）

以下に標準入力（キーボード）から１文字入力および１文字出力する例を示します。

```
;----------------------------------------------------------------------
;
;   FILE       :lowlvl.src
;   DATE       :Tue, Mar 05, 2002
;   DESCRIPTION :Program of Low level
;   CPU TYPE   :
;
;   This file is generated by Renesas Project Generator (Ver.3.0).
;
;----------------------------------------------------------------------

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;                       lowlvl.src
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;           SH-series simulator debugger interface routine
;                      -Input/output one character-
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
             .EXPORT    _charput
             .EXPORT    _charget
SIM_IO:   .EQU                                  H'0000       ;Specifies TRAP_ADDRESS

             .SECTION   P, CODE, ALIGN=4

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;  _charput: One character output
;         C program interface: charput(char)
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

_charput:
        MOV.L      O_PAR,R0         ; Sets output buffer address to R0
        MOV.B      R4,@R0           ; Sets output charcter to buffer
        MOV.L      #O_PAR,R1        ; Sets parameter block address to R1
        MOV.L      #H'01220000,R0    ; Specifies function code (PUTC)
        MOV.W      #SIM_IO,R2       ; Sets system call address to R2
        JSR        @R2
             NOP
             RTS
             NOP

        .ALIGN     4
O_PAR:                              ; Parameter block
        .DATA.L    OUT_BUF
```

```
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; _charget: One character input
;     C program interface: char charget(void)
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

        .ALIGN     4
_charget:
        MOV.L      #I_PAR,R1        ; Sets parameter block address to R1
        MOV.L      #H'01210000,R0    ; Specifies function code (GETC)
        MOV.W      #SIM_IO,R2       ; Sets system call address to R2
        JSR        @R2
             NOP
        MOV.L      I_PAR,R0         ; Sets input buffer address to R0
        MOV.B      @R0,R0          ; Returns input data
             RTS
             NOP

        .ALIGN     4
I_PAR:                             ; Parameter block
        .DATA.L    IN_BUF

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;                             I/O buffer definition
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

             .SECTION   B,DATA,ALIGN=4

OUT_BUF:
        .RES.L     1            ; Output buffer
IN_BUF:
        .RES.L     1            ; Input buffer

  .END
```

3. コンパイラ

3.1 割り込み関数

3.1.1 割り込み関数の定義（オプションなし）

■説明

プリプロセッサ制御文（#pragma）を用いて割り込み関数を C 言語で作成することができます。

"#pragma interrupt"で宣言された関数は、関数処理の前後で関数内で使用する全レジスタ（ただし、グローバルベースレジスタ GBR、ベクタベースレジスタ VBR は除く）を退避／回復します。このため、割り込まれる関数は割り込みに対処する処理を用意しておく必要がありません。

【書式】

```
#pragma interrupt  (〈関数名〉[,〈関数名〉...])
```

■使用例

割り込み関数 handler1 を宣言します。この関数は、割り込まれた関数のスタックを引き継いで使用して動作し、処理終了後、RTE 命令で復帰します。

＜GBR、VBR を退避／回復しない場合＞

C言語コード

```
#pragma interrupt(handler1)        /* 割り込み関数の宣言   */

void  handler1(void)
 {
    :                              /* 割り込み関数の処理   */
    :
 }
```

アセンブリ言語展開コード

```
      .EXPORT    _handler1
      .SECTION   P,CODE,ALIGN=4
_handler1:                          ; function: handler1
                  :                 ; 処理内で使用するレジスタの退避
                  :                 ; 割り込み関数の処理
                  :                 ; 処理内で使用したレジスタの回復
                 RTE
                 NOP
                 .END
```

＜GBR、VBR を退避／回復させる場合＞

C言語コード

```
#pragma interrupt(handler1)

void handler1(void)
{
 void** save_vbr;                  /* VBR 退避領域        */
 void* save_gbr;                   /* GBR 退避領域        */

 save_vbr = get_vbr();             /* VBR の退避          */
 save_gbr = get_gbr();             /* GBR の退避          */
    :                              /* 割り込み関数の処理 */
    :
 set_vbr(save_vbr);                /* VBR の回復          */
 set_gbr(save_gbr);                /* GBR の回復          */
}
```

アセンブリ言語展開コード

```
         .EXPORT     _handler1
         .SECTION    P,CODE,ALIGN=4
_handler1:                                   ; function: handler1
                                             ; frame size=8
         MOV.L       R5,@-R15
         STC         GBR,R5                  ; GBR の退避
         MOV.L       R4,@-R15
         STC         VBR,R4                  ; VBR の退避
           :                                 ; 処理内で使用するレジスタの退避
           :                                 ; 割り込み関数の処理
           :                                 ; 処理内で使用するレジスタの回復
         LDC         R4,VBR                  ; VBR の回復
         LDC         R5,GBR                  ; GBR の回復
         MOV.L       @R15+,R4
         MOV.L       @R15+,R5
         RTE
         NOP
         .END
```

■注意事項

(1) 割り込み関数の返すデータ型はvoidのみです。

例)

```
#pragma interrupt(f1, f2)         /* 割り込み関数の宣言     */
 void  f1(void){･･･}              /* 割り込み関数 f1 の定義 */
 int   f2(void){･･･}              /* 割り込み関数 f2 の定義 */
```

割り込み関数f1の定義は正しいのですが、割り込み関数f2の定義はエラーとなります。

(2) 割り込み関数の定義に指定できる記憶クラス指定子はexternだけです。staticを指定してもexternとして処理します。

(3) 割り込み関数として宣言した関数を通常関数として呼び出すことはできません。割り込み関数として宣言した関数を通常関数から呼び出した場合、実行時の動作は保証されません。

例)

- test1.c **ファイル内容**

```
#pragma interrupt(f1)               /* 割り込み関数の宣言      */
 void f1(void){･･･}                 /* 割り込み関数 f1 の定義  */
 int     f2(){f1();}
```

- test2.c **ファイル内容**

```
 f3(){ f1(); }
```

test1.cファイルにおいては、関数f2でエラーとなります。test2.cファイルにおいては、関数f3でエラーにはなりませんが、関数f1はextern int f1( )と解釈され、実行時の動作は不定になります。

(4) SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSPでは割り込み時の動作がSH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSPの場合と異なり、割り込みハンドラが必要になります。次に割り込みハンドラの例を示します。

SH-3用割り込みハンドラ例

```
;*******************************************************************
;   FILE        :vhandler.src
;*******************************************************************
      .include "env.inc"
      .include "vect.inc"

IMASKclr:    .equ  H'FFFFFF0F
RBBLclr:     .equ  H'CFFFFFFF
MDRBBLset:   .equ  H'70000000
      .import     _RESET_Vectors
      .import     _INT_Vectors
      .import     _INT_MASK
;*******************************************************************
;*           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
       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+,pr
       ldc.l  @r15+,spc
       ldc.l  @r15+,ssr
             .endm
;*******************************************************************
;     reset
;*******************************************************************
      .section  RSTHandler,code
_ResetHandler:
            mov.l #EXPEVT,r0
            mov.l @r0,r0
            shlr2 r0
            shlr  r0
            mov.l #_RESET_Vectors,r1
            add   r1,r0
            mov.l @r0,r0
            jmp   @r0
            nop
;*******************************************************************
;     exceptional interrupt
;*******************************************************************
      .section  INTHandler,code
      .export   INTHandlerPRG
INTHandlerPRG:
_ExpHandler:
            PUSH_EXP_BASE_REG
;
            mov.l #EXPEVT,r0          ; set event address
            mov.l @r0,r1              ; set exception code
            mov.l #_INT_Vectors,r0    ; set vector table address
            add   #-(h'40),r1         ; exception code - h'40
            shlr2 r1
            shlr  r1
```

```
            mov.l @(r0,r1),r3        ; set interrupt function addr
;
            mov.l #_INT_MASK,r0      ; interrupt mask table addr
            shlr2 r1
            mov.b @(r0,r1),r1        ; interrupt mask
            extu.b r1,r1
;
            stc   sr,r0              ; save sr
            mov.l #(RBBLclr&IMASKclr),r2   ; RB,BL,mask clear data
            and   r2,r0              ; clear mask data
            or    r1,r0              ; set interrupt mask
            ldc   r0,ssr             ; set current status
;
            ldc.l r3,spc
            mov.l #__int_term,r0     ; set interrupt terminate
            lds   r0,pr
;
            rte
            nop
;
            .pool
;
;*******************************************************************
;     Interrupt terminate
;*******************************************************************
            .align 4
__int_term:
            mov.l  #MDRBBLset,r0  ; set MD,BL,RB
            ldc.l r0,sr           ;
            POP_EXP_BASE_REG
            rte                   ; return
            nop
;
            .pool
;
;*******************************************************************
;     TLB miss interrupt
;*******************************************************************
      .org  H'300
_TLBmissHandler:
            PUSH_EXP_BASE_REG
;
            mov.l #EXPEVT,r0         ; set event address
            mov.l @r0,r1             ; set exception code
            mov.l #_INT_Vectors,r0   ; set vector table address
            add   #-(h'40),r1        ; exception code - h'40
            shlr2 r1
            shlr  r1
            mov.l @(r0,r1),r3        ; set interrupt function addr
;
            mov.l #_INT_MASK,r0      ; interrupt mask table addr
            shlr2 r1
            mov.b @(r0,r1),r1        ; interrupt mask
            extu.b r1,r1
;
            stc   sr,r0              ; save sr
            mov.l #(RBBLclr&IMASKclr),r2   ; RB,BL,mask clear data
            and   r2,r0              ; clear mask data
            or    r1,r0              ; set interrupt mask
            ldc   r0,ssr             ; set current status
;
            ldc.l r3,spc
            mov.l #__int_term,r0     ; set interrupt terminate
            lds   r0,pr
;
            rte
            nop
;
            .pool
;
;*******************************************************************
;     IRQ
;*******************************************************************
      .org  H'500
```

```
_IRQHandler:
            PUSH_EXP_BASE_REG
;
            mov.l  #INTEVT,r0          ; set event address
            mov.l  @r0,r1              ; set exception code
            mov.l  #_INT_Vectors,r0    ; set vector table address
            add    #-(h'40),r1         ; exception code - h'40
            shlr2  r1
            shlr   r1
            mov.l  @(r0,r1),r3         ; set interrupt function addr
;
            mov.l  #_INT_MASK,r0       ; interrupt mask table addr
            shlr2  r1
            mov.b  @(r0,r1),r1         ; interrupt mask
            extu.b r1,r1
;
            stc    sr,r0               ; save sr
            mov.l  #(RBBLclr&IMASKclr),r2    ; RB,BL,mask clear data
            and    r2,r0               ; clear mask data
            or     r1,r0               ; set interrupt mask
            ldc    r0,ssr              ; set current status
;
            ldc.l  r3,spc
            mov.l  #__int_term,r0      ; set interrupt terminate
            lds    r0,pr
;
            rte
            nop
;
            .pool
            .end
```

【注】　SH-3 用割り込みハンドラ例のテーブル部において対応するアドレスがないところを空けて用してください。このとき、ベクタ関数へのジャンプのために RTE 命令を使用しています。また、ベクタ関数からの戻りがターミネートルーチンであるため、ベクタ関数は RTS で復帰する必要があります。

そのため、ベクタ関数を定義する際には「#pragma interrrupt」を使用しないでください。インクルードファイルの"env.inc"と"vect.inc"は HEW で SH3 のプロジェクト作成時に自動生成されます。

本リストに示した PUSH_EXP_BASE_REG、POP_EXP_BASE_REG マクロでは、R0～R7 のバンクレジスタをスタックへ退避（stc.l rn_bank, @-R15）、回復（ldc.l @R15+, rn_bank）しており、通常の MOV 命令でアクセス可能なレジスタ内容の退避は行っておりません。

SH-3,SH3-DSP,SH-4,SH-4A,SH4AL-DSP では、割り込み受け付け時、SR レジスタの RB ビットが 1 に設定されます。そのため、本マクロをそのまま割り込みハンドラの最初に使用した場合には、上記退避命令により退避されるレジスタは、R0_BANK0～R7_BANK0 となり、R0_BANK1～R7_BANK1 レジスタは退避されません。したがって、RB=1 でプログラムを実行していて割り込みを受け付け、割り込みハンドラの先頭で上記マクロを用いてレジスタ退避を行っても、R0_BANK1～R7_BANK1 が退避されないため、割り込み受け付け前のレジスタ内容が破壊されてしまいます。

本マクロを使用する場合には、割り込み受け付け前のプログラムを RB=0 で実行して頂くか、R0_BANK1～R7_BANK1 の退避を行うようにマクロを修正してください。

3.1.2 割り込み関数の定義（オプションあり）

■説明

割り込み関数定義のオプションには「スタック切り替え指定」と、「トラップ命令リターン指定」および「レジスタバンク指定」があります。

「スタック切り替え指定」により、外部割り込み発生の際、スタックポインタを指定アドレスに切り替え、このスタックを利用して割り込み関数を動作させます。復帰時にはスタックポインタを割り込み発生前に戻します(**図**3.1)。本指定で割り込まれる関数のスタックには、割り込み関数用スタックを余分に確保しておく必要がなくなります。

「トラップ命令リターン指定」により、復帰を TRAPA 命令で行います。指定しない場合は RTE 命令で復帰します。

SH-2A、SH2A-FPU は、割り込み処理に伴うレジスタの退避、回復を高速に行うためのレジスタバンクを内蔵しています。

「レジスタバンク指定」により、レジスタバンクの使用を前提としたレジスタ退避・回復コードを生成します。

具体的には、バンク対象レジスタ（R0～R14、GBR、MACH、MACL、PR）の退避は割り込み例外発生時に自動的に行うことになるので、割り込み関数側での退避コードの生成を抑止します。

また、バンク対象レジスタの回復は RESBANK 命令で行うことになります。

【書式】

```
#pragma interrupt (〈関数名〉[(〈割り込み仕様〉)][,〈関数名〉[(〈割り込み仕様〉)]...])
```

表 3.1　**割り込み仕様一覧**

項番	項目	形式	オプション	指定内容
1	スタック 切り替え指定	sp＝	{ <変数> | &<変数> | <定数> | <変数>+<定数> | &<変数>+<定数> }	新しいスタックのアドレスを変数または定数で指定 <変数>　：変数(ポインタ型) &<変数>：変数(オブジェクト型)のアドレス <定数>　：定数値
2	トラップ命令 リターン指定	tn＝	〈定数〉	終了を TRAPA 命令で指定 〈定数〉　：定数値（トラップベクタ番号）
3	レジスタバンク指定	resbank	なし	以下のレジスタについて、レジスタ退避コードの出力抑止 R0～R14、GBR、MACH、MACL、PR tn 指定がない場合は、RTE 命令の直前に RESBANK 命令を生成

■使用例

使用例 1

割り込み関数 handler2 を宣言します。この関数は、配列 STK をスタックとして使用し（**図** 3.1）、処理終了後、“ TRAPA#63 ” 命令で復帰します。

C言語コード

```
extern  int  STK[100];
int  *ptr  = STK + 100;
#pragma interrupt(handler2(sp=ptr, tn=63))
                                        /* 割り込み関数の宣言      */
void  handler2(void)
{
        :                               /* 割り込み関数の処理記述 */
        :

}
```

アセンブリ言語展開コード

```
          .IMPORT     _STK
          .EXPORT     _ptr
          .EXPORT     _handler2
          .SECTION    P,CODE,ALIGN=4
_handler2:                              ;  function: handler2
                                        ;  frame size=4
          MOV.L       R0,@-R15
          MOV.L       L217,R0           ; _ptr
          MOV.L       @R0,R0
          MOV.L       R15,@-R0
          MOV         R0,R15
            :                           ;  処理内で使用するレジスタの退避
            :                           ;  割り込み関数処理
            :                           ;  処理内で使用したレジスタの回復
          MOV.L       @R15+,R15
          MOV.L       @R15+,R0
          TRAPA       #63
L217:
          .DATA.L     _ptr
          .SECTION    D,DATA,ALIGN=4
_ptr:                                          ;  static: ptr
          .DATA.L     H'00000190+_STK
          .END
```

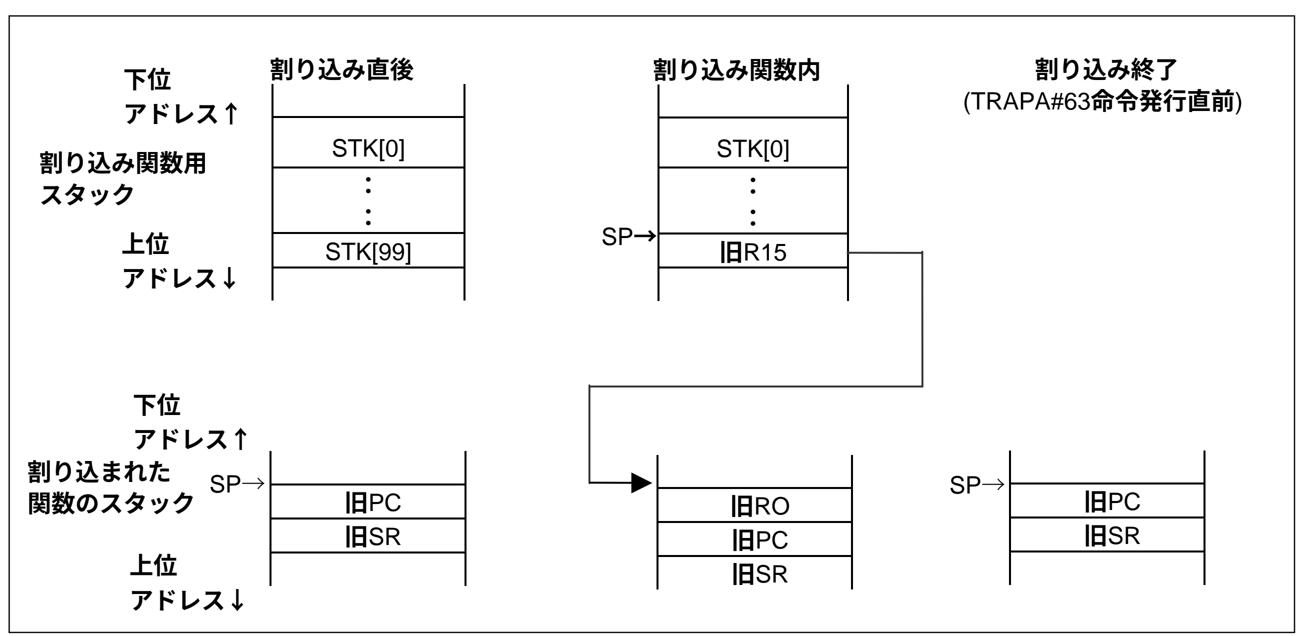


図 3.1　割り込み関数によるスタック使用例

使用例２

C言語コード

```
#pragma interrupt(handler(resbank))
double flag1;
int  flag2;
void handler()
{
         flag1 = 1;
         flag2 = 1;
}
```


RENESAS

アセンブリ言語展開コード(-cpu=sh2afpu指定時)

```
 _handler:
                                        ; 処理内で使用しているバンク対象レジスタ
                                        ; の退避コードは出力しません
          FMOV.S     FR8,@-R15          ; 処理内で使用しているバンク対象外の
          FMOV.S     FR9,@-R15          ; レジスタの退避を行います。
          MOVA       L11,R0
          MOV        #1,R4
          FMOV.S     @R0+,FR8
          MOV.L      L11+8,R1
          FMOV.S     @R0,FR9
          MOV.L      L11+12,R6
          FMOV.S     FR9,@-R1
          FMOV.S     FR8,@-R1
          MOV.L      R4,@R6
          FMOV.S     @R15+,FR9          ; 処理内で使用しているバンク対象外の
          FMOV.S     @R15+,FR8          ; 回復を行います。
          RESBANK                       ; バンク対象レジスタの回復を行います。
          RTE
          NOP
```

■注意事項

（1） resbank指定はcpu=sh2a|sh2afpu指定時のみ有効です。

（2） resbank指定関数の割り込み発生前にレジスタバンク使用を許可してください。

（3） resbank、tnとも指定した場合は、レジスタ退避コード、RESBANK命令とも生成しません。この場合は、トラップルーチン側でRESBANK 命令を生成するようにしてください。

（4） resbank指定を行った関数から回復するときは、#pragma global_registerで指定した変数の値は、割り込み処理中に書き換えた場合も割り込み前の値になります。

3.1.3 ベクタテーブルの作成

■説明

ベクタテーブルは、次のように設定することによりＣ言語で作成できます。

(1) ベクタテーブル用の配列を用意し、これの各要素に例外処理関数のポインタを指定します。

(2) このファイルのコンパイル後、ベクタテーブルの先頭アドレスを指定してリンクします。

■使用例

C言語コード:vect_table.c

```
extern    void  reset(void);       /* パワーオンリセット処理関数             */
extern    void  warm_reset(void);  /* マニュアルリセット処理関数             */
extern    void  irq0(void);        /* IRQ0 割り込み処理関数                  */
extern    void  irq1(void);        /* IRQ1 割り込み処理関数                  */
       :
       :

void(* const vect_table[])(void) = {
          reset,                   /* パワーオンリセット時の開始アドレス     */
          0,                       /* パワーオンリセット時のスタックポインタ*/
          warm_reset,              /* マニュアルリセット時の開始アドレス     */
          0,                       /* マニュアルリセット時のスタックポインタ*/
           :
           :
          irq0,                    /* ベクタ番号 64                          */
          irq1,                    /* ベクタ番号 65                          */
           :
           :
};
```

バッチファイル

```
shc△-section=c=VECT△vect_table
shc△reset△warm_reset△irq0△irq1 …
optlnk△-noopt△-subcommand=link.sub
```

link.subの内容は以下のとおりです。

```
sdebug
input vect_table
input reset
input warm_reset
input irq0,irq1
      ⋮
output sample.abs
start VECT/0,P,C,D/0400,B/0F000000
exit
```

vect_table.c をコンパイルすることにより、初期化データセクション（VECT）のみのリロケータブルオブジェクトファイル vect_table.obj が生成されます。

セクション VECT を先頭アドレス H'0 番地に指定して他のファイルと共にリンクし、ロードモジュールファイル sample.abs を得ます。

アセンブリ言語展開コード:vect_table.src

```
          .IMPORT     _reset
          .IMPORT     _warm_reset
          .IMPORT     _irq0
          .IMPORT     _irq1
          .EXPORT     _vect_table
          .SECTION    VECT,DATA,ALIGN=4
_vect_table:                                   ; static: vect_table
          .DATA.L     _reset
          .DATA.L     H'00000000
          .DATA.L     _warm_reset
          .DATA.L     H'00000000
               :
               :
          .DATA.L     _irq0,_irq1
               :
               :
          .END
```

■注意事項

SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP では割り込み時の動作が SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP の場合と異なり、ベクタテーブルではなく、割り込みハンドラが必要になります。

ベクタテーブルは確定した絶対アドレスに割り付ける必要があるので、ここでは単独ファイルにして作成しましたが、セクション切り替え機能を用いることにより、他のモジュールと同一ファイルにすることができます。詳細については「3.7　**セクション名指定**」を参照してください。

3.2 組み込み関数

SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP、SH-3、SH3-DSP、SH-4、SH-4A、および SH4AL-DSP 固有の命令を C 言語で記述できるように**表 3.2** に示す組み込み関数を用意しています。なお、組み込み関数使用の際は、標準ヘッダファイル"machine.h"または"smachine.h"や "umachine.h"をインクルードする必要があります。

SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP の実行モードに対応し<machine.h>の内容を次のように分割しています。

(1) 〈machine.h〉　組み込み関数全体
(2) 〈smachine.h〉　特権モードでのみ、使用可能な組み込み関数
(3) 〈umachine.h〉　(2)以外の組み込み関数

表 3.2 組み込み関数機能一覧（1）

項番	項目	機能	使用可能 CPU	使用できる実行モード	参照
1	ステータスレジスタ（SR）	SR の設定	全 CPU	特権モードのみ	3.2.1
2		SR の参照			
3		割り込みマスクの設定			
4		割り込みマスクの参照			
5	ベクタベースレジスタ（VBR）	VBR の設定	全 CPU	特権モードのみ	3.2.2
6		VBR の参照			
7	グローバルベースレジスタ（GBR）	GBR の設定	全 CPU	制限なし	3.2.3
8		GBR の参照			3.2.4
9		GBR ベースのバイト参照			
10		GBR ベースのワード参照			
11		GBR ベースのロングワード参照			
12		GBR ベースのバイト設定			
13		GBR ベースのワード設定			
14		GBR ベースのロングワード設定			
15		GBR ベースのバイト AND			
16		GBR ベースのバイト OR			
17		GBR ベースのバイト XOR			
18		GBR ベースのバイト TEST			
19	システム制御	SLEEP 命令	全 CPU	特権モードのみ	3.2.5
20		TAS 命令		制限なし	
21		TRAPA 命令			

表 3.2　組み込み関数機能一覧（2）

項番	項目	機能	使用可能 CPU	使用できる実行モード	参照
22	積和演算	ワード積和演算	全 CPU	制限なし	3.2.6
23		ロングワード積和演算	CPU=sh1 を除く		
24		リングバッファ対応ワード積和演算	全 CPU	制限なし	3.2.7
25		リングバッファ対応ロングワード積和演算	CPU=sh1 を除く		
26	64bit 乗算	符号付き 64bit 乗算の上位 32bit	CPU=sh1 を除く	制限なし	3.2.8
27		符号付き 64bit 乗算の下位 32bit			
28		符号なし 64bit 乗算の上位 32bit	CPU=sh1 を除く	制限なし	3.2.9
29		符号なし 64bit 乗算の下位 32bit			
30	データ上位・下位交換	SWAP.B 命令	全 CPU	制限なし	3.2.10
31		SWAP.W 命令			
32		4 バイトデータの上位・下位交換			
33	システムコール	システムコール実行	全 CPU	制限なし	3.2.11
34	プリフェッチ命令	プリフェッチ命令	CPU=sh2a,sh2afpu,sh3,sh3dsp,sh4,sh4a,sh4aldsp 指定時のみ	制限なし	3.2.12
35	LDTLB 命令	LDTLB 展開	CPU=sh3,sh3dsp,sh4,sh4a,sh4aldsp 指定時のみ	特権モードのみ	3.2.13
36	NOP 命令	NOP 展開	全 CPU	制限なし	3.2.14
37	浮動小数点ユニット	FPSCR の設定	CPU=sh2e,sh2afpu,sh4,sh4a 指定時のみ	制限なし	3.2.15
38		FPSCR の参照			
39	単精度浮動小数点ベクタ演算	FIPR 命令	CPU=sh4,sh4a 指定時のみ	制限なし	
40		FTRV 命令			
41		４次元ベクタの４×４行列による変換と４次元ベクタとの和			
42		４次元ベクタの４×４行列による変換と４次元ベクタとの差			
43		４次元ベクタの和	CPU=sh2afpu,sh4,sh4a 指定時のみ		
44		４次元ベクタの差			
45		４×４行列の乗算	CPU=sh4,sh4a 指定時のみ		
46		４×４行列の乗算と和			
47		４×４行列の乗算と差			
48	拡張レジスタのアクセス	拡張レジスタへのロード	CPU=sh4,sh4a 指定時のみ	制限なし	3.2.16
49		拡張レジスタからのストア			

RENESAS

表 3.2 組み込み関数機能一覧（3）

項番	項目	機能	使用可能 CPU	使用できる実行モード	参照
50	DSP 命令	絶対値	sh2dsp, sh3dsp,sh4aldsp および dspc を 指定した場合	制限なし	3.2.17
51		MSB 検出			
52		算術シフト			
53		丸め演算			
54		ビットパターンコピー			
55		モジューロアドレッシング設定		特権モードのみ	
56		モジューロアドレッシング解除			
57		CS ビット(DSR レジスタ)設定		制限なし	
58	正弦・余弦	正弦・余弦の計算	CPU=sh4a 指定時のみ	制限なし	3.2.18
59	平方根の逆数	平方根の逆数	CPU=sh4a 指定時のみ	制限なし	3.2.19
60	命令キャッシュの無効化	命令キャッシュブロックの無効化	CPU=sh4a,sh4aldsp 指定時のみ	制限なし	3.2.20
61	キャッシュブロック操作	キャッシュブロックの無効化	CPU=sh4a,sh4aldsp 指定時のみ	制限なし	3.2.21
62		キャッシュブロックのパージ			
63		キャッシュブロックの書き戻し			
64	命令キャッシュプリフェッチ	命令キャッシュブロックのプリフェッチ	CPU=sh4a,sh4aldsp 指定時のみ	制限なし	3.2.22
65	システム同期	データ操作の同期	CPU=sh4a,sh4aldsp 指定時のみ	制限なし	3.2.23

RENESAS

表 3.2　組み込み関数機能一覧（4）

項番	項目	機能	使用可能 CPU	使用できる実行モード	参照
66	T ビット参照・設定	T ビットの参照	全 CPU	制限なし	3.2.24
67		T ビットのクリア			
68		T ビットのセット			
69	連結レジスタの中央切り出し	連結した 64 ビットの内容から中央の 32 ビットを切り出し	全 CPU	制限なし	3.2.25
70	キャリ付き加算	２つのデータと T ビットを加算し、キャリを T ビットに反映	全 CPU	制限なし	3.2.26
71		２つのデータと T ビットを加算し、キャリを参照			
72		２つのデータを加算し、キャリを T ビットに反映			
73		２つのデータを加算し、キャリを参照			
74	ボロー付き減算	data1 から data2 と T ビットを減算し、ボローを T ビットに反映	全 CPU	制限なし	3.2.27
75		data1 から data2 と T ビットを減算し、ボローを参照			
76		data1 から data2 を減算し、ボローを T ビットに反映			
77		data1 から data2 を減算し、ボローを参照			
78	符号反転	0 から data と T ビットを減算し、ボローを T ビットに反映	全 CPU	制限なし	3.2.28
79	1 ビット除算	data1/data2 の 1 ステップ除算を行い、結果を T ビットに反映	全 CPU	制限なし	3.2.29
80		data1/data2 の符号付き除算の初期設定をし、T ビットを参照			
81		符号なし除算の初期設定			

表 3.2　組み込み関数機能一覧（5）

項番	項目	機能	使用可能 CPU	使用できる実行モード	参照
82	回転	データを 1 ビット左方向に回転し、オペランドの外に出たビットを T ビットに反映	全 CPU	制限なし	3.2.30
83		データを 1 ビット右方向に回転し、オペランドの外に出たビットを T ビットに反映			
84		データを 1 ビット左方向に T ビットを含めて回転し、オペランドの外に出たビットを T ビットに反映			
85		データを 1 ビット右方向に T ビットを含めて回転し、オペランドの外に出たビットを T ビットに反映			
86	シフト	データを 1 ビット左シフトし、オペランドの外へ出たビットを T ビットに反映	全 CPU	制限なし	3.2.31
87		データを論理的に 1 ビット右シフトし、オペランドの外へ出たビットを T ビットに反映			
88		データを算術的に 1 ビット右シフトし、オペランドの外へ出たビットを T ビットに反映			
89	飽和演算	符号付き 1 バイトデータ飽和演算	cpu=sh2a,sh2afpu 指定時のみ	制限なし	3.2.32
90		符号付き 2 バイトデータ飽和演算			
91		符号なし 1 バイトデータ飽和演算			
92		符号なし 2 バイトデータ飽和演算			
93	TBR 設定・参照	TBR に data を設定	cpu=sh2a,sh2afpu 指定時のみ	制限なし	3.2.33
94		TBR の値を参照			

3.2.1 ステータスレジスタの設定／参照

■説明

ステータスレジスタの設定／参照について、**表** 3.3 に示す関数を用意しています。

表 3.3 **ステータスレジスタ用組み込み関数一覧**

項番	項目	書式	説明
1	ステータスレジスタの設定	void set_cr(int cr)	ステータスレジスタにcr(32bit)を設定
2	ステータスレジスタの参照	int get_cr(void)	ステータスレジスタを参照
3	割り込みマスクの設定	void set_imask(int mask)	割り込みマスク(4bit)にmask(4bit)を設定
4	割り込みマスクの参照	int get_imask(void)	割り込みマスク(4bit) を参照

■使用例

関数 func1 は、割り込みマスクを最高(15)に設定することで外部割り込みを禁止して処理を行います。処理終了後、元の割り込みマスクレベルに復帰させて終了します。

C言語コード

```
#include <machine.h>

void  func1(void)
{
    int   mask;                          /* 割り込みマスクレベルの保存場所     */

    mask = get_imask();                  /* 割り込みマスクレベルを保存         */
    set_imask(15);                       /* 割り込みマスクレベルを 15 に設定   */
         :                               /* 割り込みを禁止して処理を実行       */
         :
    set_imask(mask);                     /* 割り込みマスクレベルの復帰         */
}
```

アセンブリ言語展開コード

```
          .EXPORT     _func1
          .SECTION     P,CODE,ALIGN=4
_func1:                                       ; function: func1
                                              ; frame size=0
          MOV.W       L216,R3                 ; H'FF0F
          STC         SR,R0
          SHLR2       R0
          SHLR2       R0
          AND         #15,R0
          MOV         R0,R4
          STC         SR,R0
          AND         R3,R0
          OR          #240,R0
          LDC         R0,SR
            :
            :
          MOV         R4,R0
          AND         #15,R0
          SHLL2       R0
          SHLL2       R0
          STC         SR,R2
          MOV         R3,R1
          AND         R1,R2
          OR          R2,R0
          LDC         R0,SR
          RTS
          NOP
L216:
          .DATA.W     H'FF0F
          .END
```

3.2.2 ベクタベースレジスタの設定／参照

■説明

ベクタベースレジスタの設定／参照について、**表** 3.4 に示す関数を用意しています。

表 3.4 ベクタベースレジスタ用組み込み関数一覧

項番	項目	書式	説明
1	ベクタベースレジスタの設定	void set_vbr(void **base)	ベクタベースレジスタ に **base(32bit)を設定
2	ベクタベースレジスタの参照	void **get_vbr(void)	ベクタベースレジスタを参照

■使用例

リセットによりベクタベースレジスタ（VBR）は0に初期化されます。ベクタテーブルを0番地以外から開始している場合、マニュアルリセット時の開始アドレス（H'00000008)に次の関数を設定して、システム開始時にマニュアルリセットすれば、設定ベクタテーブルを使用した例外処理が実行できます。

C言語コード

```
#include <machine.h>
#define VBR  0x0000FC00              /* ベクタテーブルの先頭アドレス   */

void  warm_reset(void)
{
   set_vbr((void **)VBR);
                                     /* ベクタベースレジスタをベクタ   */
                                     /* テーブルの先頭アドレスに設定   */
}
```

アセンブリ言語展開コード

```
    .EXPORT      _warm_reset
    .SECTION     P,CODE,ALIGN=4
_warm_reset:                            ; function: warm_reset
                                        ; frame size=0
    MOV.L        L215,R3                ; H'0000FC00
    LDC          R3,VBR
    RTS
    NOP
L215:
    .DATA.L      H'0000FC00
    .END
```

■注意事項

ベクタベースレジスタの変更は、ベクタテーブルを設定してから行ってください。順序を逆にすると、ベクタテーブルを設定している間に外部割り込みが発生した場合にシステムダウンを起こします。

3.2.3 I/O レジスタへのアクセス（1）

■説明

I/O レジスタへアクセスするためのグローバルベースレジスタ（GBR）操作について、**表** 3.5 に示す関数を用意しています。

表 3.5 グローバルベースレジスタ用組み込み関数一覧

項番	項目	書式	説明
1	グローバルベースレジスタの設定	void set_gbr(void *base)	グローバルベースレジスタに *base(32bit)を設定
2	グローバルベースレジスタの参照	int *get_gbr(void)	グローバルベースレジスタを参照
3	グローバルベースレジスタベースのバイト参照	unsigned char gbr_read_byte(int offset)	グローバルベースレジスタ相対 offset のバイトデータ(8bit)を参照
4	グローバルベースレジスタベースのワード参照	unsigned short gbr_read_word(int offset)	グローバルベースレジスタ相対 offset のワードデータ(16bit)を参照
5	グローバルベースレジスタベースのロングワード参照	unsigned long gbr_read_long(int offset)	グローバルベースレジスタ相対 offset のロングワードデータ(32bit)を参照
6	グローバルベースレジスタベースのバイト設定	void gbr_write_byte(int offset,unsigned char data)	グローバルベースレジスタ相対 offset の data(8bit)を設定
7	グローバルベースレジスタベースのワード設定	void gbr_write_word(int offset,unsigned short data)	グローバルベースレジスタ相対 offset の data(16bit)を設定
8	グローバルベースレジスタベースのロングワード設定	void gbr_write_long(int offset,unsigned long data)	グローバルベースレジスタ相対 offset の data(32bit)を設定
9	グローバルベースレジスタベースのバイト AND	void gbr_and_byte(int offset,unsigned char mask)	グローバルベースレジスタ相対 offset のバイトデータと mask の AND をとり、offset に設定
10	グローバルベースレジスタベースのバイト OR	void gbr_or_byte(int offset,unsigned char mask)	グローバルベースレジスタ相対 offset のバイトデータと mask の OR をとり、offset に設定
11	グローバルベースレジスタベースのバイト XOR	void gbr_xor_byte(int offset,unsigned char mask)	グローバルベースレジスタ相対 offset のバイトデータと mask の XOR をとり、offset に設定
12	グローバルベースレジスタベースのバイト TEST	void gbr_tst_byte(int offset,unsigned char mask)	グローバルベースレジスタ相対 offset のバイトデータを mask と AND をとり、その値を０と比較判定する。結果をＴビットにセット

【注】
(1) base は、アクセスサイズがワードのときには２の倍数、アクセスサイズがロングワードのときには 4 の倍数に設定してください。
(2) offset は、項番 3～8 のときには定数でなければなりません。offset の指定可能範囲は、アクセスサイズがバイトのときには+255 バイト、アクセスサイズがワードのときには+510 バイト、アクセスサイズがロングワードのときには+1020 バイトまでです。
(3) mask は定数でなければなりません。mask の指定可能範囲は 0～+255 です。
(4) グローバルベースレジスタはコントロールレジスタですので、関数入口／出口での値の退避／回復を C/C++コンパイラは実行しません。グローバルベースレジスタの値を変更する際は、ユーザが関数入口／出口での値の退避／回復を行ってください。
(5) gbr=auto を指定した場合、本関数は使用できません。(Ver.7 以降)

■使用例

SH-1 内蔵の 16 ビットインテグレーテッドタイマパルスユニットを使用したタイマドライバの例です。

C言語コード

```
#include <machine.h>
#define IOBASE 0x05fffec0       /* I/O ベースアドレス              */
#define TSR   (0x05ffff07 - IOBASE)
                                /* タイマステータスフラグレジスタの  */
                                /* オフセットアドレス               */
#define TSRCLR (unsigned char)0xf8
                                /* タイマステータスフラグレジスタの  */
                                /* クリア値                        */

void  tmrhdr(void)
{
   void     *gbrsave;           /* グローバルベースレジスタの値の    */
                                /* 保存場所                        */

   gbrsave = get_gbr();         /* グローバルベースレジスタの値を保存*/
   set_gbr((void *)IOBASE);
                                /* グローバルレジスタに I/O ベース   */
                                /* アドレスを設定                   */

   gbr_read_byte(TSR);          /* タイマステータスフラグレジスタを  */
                                /* クリアするためにダミーリード      */
   gbr_and_byte(TSR, TSRCLR);
                                /* タイマステータスフラグレジスタの  */
                                /* コンペアマッチフラグをクリア      */

   set_gbr(gbrsave);            /* グローバルベースレジスタの値の復帰*/
}
```

アセンブリ言語展開コード

```
        .EXPORT     _tmrhdr
        .SECTION    P,CODE,ALIGN=4
_tmrhdr:                    ; function: tmrhdr
                            ; frame size=4
        ADD         #-4,R15
        STC         GBR,R1
        MOV.L       L11,R5      ; H'05FFFEC0
        MOV.L       R1,@R15
        LDC         R5,GBR
        MOV.B       @(71,GBR),R0
        MOV         #71,R0      ; H'00000047
        AND.B       #248,@(R0,GBR)
        MOV.L       @R15,R2
        LDC         R2,GBR
        RTS
        ADD         #4,R15
L11:
        .DATA.L     H'05FFFEC0
```

3.2.4 I/O レジスタへのアクセス（2）

■使用例

標準ライブラリ offsetof を利用することにより、グローバルベースレジスタ相対 offset をあらかじめ計算しておく必要がなくなります。

C言語コード

```
#include <stddef.h>
#include <machine.h>
struct IOTBL{
   char  cdata1;           /* offset  0              */
   char  cdata2;           /* offset  1              */
   char  cdata3;           /* offset  2              */
   short sdata1;           /* offset  4              */
   int   idata1;           /* offset  8              */
   int   idata2;           /* offset 12              */
} table;

void  f(void)
{
    void     *gbrsave;     /* グローバルベースレジスタの値の    */
                           /* 保存場所                         */
    gbrsave = get_gbr();   /* グローバルベースレジスタの値を保存 */
    set_gbr(&table);       /* グローバルベースレジスタに        */
                           /* table 先頭アドレスを設定          */
             :
             :
    gbr_and_byte(offsetof(struct IOTBL, cdata2), 0x10);
                           /* table.cdata2 の値と 0x10 の AND */
                           /* をとって table.cdata2 に設定     */
             :
             :

    set_gbr(gbrsave);      /* グローバルベースレジスタの値の復帰 */
}
```

アセンブリ言語展開コード

```
    .EXPORT     _table
    .EXPORT     _f
    .SECTION    P,CODE,ALIGN=4
_f:                                     ; function: f
                                        ; frame size=0
    MOV.L       L217+2,R3               ; _table
    MOV         #1,R0
    STC         GBR,R4
    LDC         R3,GBR
       :
       :
    AND.B       #16,@(R0,GBR)
       :
       :
    RTS
    LDC         R4,GBR
L217:
    .RES.W      1
    .DATA.L     _table
    .SECTION    B,DATA,ALIGN=4
_table:                                   ; static: table
    .RES.L      4
    .END
```

3.2.5 システム制御

■説明

ルネサステクノロジ SuperH RISC engine ファミリ専用の特殊命令について、**表** 3.6 に示す関数を用意しています。

表 3.6 特殊命令用組み込み関数一覧

項番	項目	書式	説明
1	SLEEP 命令	void sleep(void)	SLEEP 命令に展開
2	TAS 命令	int tas(char *addr)	TAS.B @addr に展開
3	TRAPA 命令	int trapa(int trap_no)	TRAPA #trap_no に展開
4	OS システムコールの実現	-	3.2.11 参照
5	PREF 命令	-	3.2.12 参照

【注】 (1) 表中の trap_no は定数でなければなりません。
(2) trapa 組み込み関数は、C 言語プログラムから割り込み関数を起動します。
呼び出される関数は、割り込み関数として作成してください。

■使用例

SLEEP 命令を発行し、CPU を低消費電力状態にします。

低消費電力状態では、CPU の内部状態を保持して直後の命令の実行を停止し、割り込み要求の発生を待ちます。割り込みが発生すると、低消費電力状態から抜けます。

C言語コード

```
#include <machine.h>

void  func(void)
{
        :
        :
   sleep();                          /* SLEEP 命令を発行します */
        :
        :
}
```

アセンブリ言語展開コード

```
       .EXPORT     _func
       .SECTION    P,CODE,ALIGN=4
_func:                                    ; function: func
                                          ; frame size=0
       SLEEP
       RTS
       NOP
       .END
```

3.2.6 積和演算（1）

■説明

積和演算について、**表** 3.7 に示す関数を用意しています。

表 3.7 積和演算用組み込み関数一覧

項番	項 目	書 式	説 明
1	ワード積和演算	int macw(short *ptr1, short *ptr2,unsigned int count)	ワードデータ*ptr1（16bit）とワードデータ*ptr2（16bit）の count 数積和演算
2	ロングワード積和演算	int macl(int *ptr1, int *ptr2,unsigned int count)	ロングワードデータ*ptr1（32bit）とロングワードデータ*ptr2（32bit）の count 数積和演算

ワード積和関数 macw は SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP、SH-3、SH3-DSP、SH-4、SH-4A および SH4L-DSP の全 cpu でサポートしていますが、ロングワード積和関数 macl は SH-1 以外でサポートしています。

積和演算組み込み関数は引数チェックを行いません。積和演算するデータのテーブルは双方とも、ワード積和関数のときには 2 バイト、ロングワード積和関数のときには 4 バイトで境界調整してください。

■使用例

積和演算を行います。積和演算を実行する回数が 32 回以下であれば MAC 命令の繰り返しで実現し、33 回以上または回数が変数のときには MAC 命令のループで実現します。

C言語コード

```
#include <machine.h>
short a[SIZE];
short b[SIZE];

Void  func(void)
{
        int x;
               :
               :
        x = macw(a,b,SIZE);
               :
               :
}
```

```
a[0]        *    b[0]        +
a[1]        *    b[1]        +
a[2]        *    b[2]        +
  :                :         +
a[SIZE-2]   *    b[SIZE-2]   +
a[SIZE-1]   *    b[SIZE-1]
```

アセンブリ言語展開コード

・case of size ≦ 32 ：MAC命令の繰り返しで実現

```
       .EXPORT   _func
       .SECTION  P,CODE,ALIGN=4
_func:                              ; function: func
                                       ; frame size=8
       STS.L       MACH,@-R15
       STS.L       MACL,@-R15
       MOV.L       L218+2,R3           ; _b
       CLRMAC
       MOV.L       L218+6,R2           ; _a
       MAC.W       @R2+,@R3+
         :                             ; SIZE 分繰り返し
         :
       STS         MACL,R0
       LDS.L       @R15+,MACL
       RTS
       LDS.L       @R15+,MACH
L218:
       .RES.W      1
       .DATA.L     _b
       .DATA.L     _a
```

・case of size > 32または変数：MAC命令のループで実現

```
        .EXPORT      _func
        .SECTION     P,CODE,ALIGN=4
_func:                                    ; function: func
                                          ; frame size=8
        STS.L        MACH,@-R15
        MOV          #33,R3
        STS.L        MACL,@-R15
        TST          R3,R3
        CLRMAC
        BT           L218
        MOV.L        L220+2,R2            ; _b
        SHLL         R3
        MOV.L        L220+6,R1            ; _a
        ADD          R1,R3
L219:
        MAC.W        @R1+,@R2+
        CMP/HI       R1,R3
        BT           L219
L218:
        STS          MACL,R0
        LDS.L        @R15+,MACL
        RTS
        LDS.L        @R15+,MACH
L220:
        .RES.W       1
        .DATA.L      _b
        .DATA.L      _a
```

3.2.7 積和演算（2）

■説明

リングバッファ対応積和演算について、表 3.8 に示す関数を用意しています。

表 3.8 リングバッファ対応積和演算用組み込み関数一覧

項番	項 目	書 式	説 明
1	リングバッファ対応ワード積和演算	int macwl(short *ptr1, short *ptr2,unsigned int count, unsigned int mask)	ワードデータ*ptr1（16bit）とマスク mask で指定されるワードデータ*ptr2（16bit）の count 数積和演算
2	リングバッファ対応ロングワード積和演算	int macll(int *ptr1, int *ptr2,unsigned int count, unsigned int mask)	ロングワードデータ*ptr1（32bit）とマスク mask で指定されるロングワードデータ*ptr2（32bit）の count 数積和演算

リングバッファ対応ワード積和関数 macwl は SH-1、SH-2、SH-2E、SH-2A、SH2A-FPU、SH2-DSP、SH-3、SH3-DSP、SH-4、SH-4A および SH4AL-DSP の全 CPU でサポートしていますが、リングバッファ対応ロングワード積和関数 macll は SH-1 以外でサポートしています。

リングバッファ対応積和演算組み込み関数は引数チェックを行いません。第 1 引数はワード積和関数のときには 2 バイト、ロングワード積和関数のときには 4 バイト、第 2 引数はリングバッファマスクの 2 倍のサイズで境界調整してください。

■使用例

リングバッファ対応積和演算を行います。第 2 引数はリングバッファのサイズの 2 倍で境界調整する必要があるので別ファイルにします。

C言語ソースコード：macwl.c

```
#include <machine.h>

short a[SIZE];
extern short b[];

void  func(void)
{
      int x;
           :
           :
       x = macwl(a,b,SIZE,~0x10);

           :
           :
}
```

```
     a[0]          *   b[0]   +
     a[1]          *   b[1]   +
      :                 :     +
     a[7]          *   b[7]   +
     a[8]          *   b[0]   +
     a[9]          *   b[1]   +
      :                 :     +
     a[15]         *   b[7]   +
      :                 :     +
   a[SIZE-8]       *   b[0]   +
   a[SIZE-7]       *   b[1]   +
      :                 :     +
   a[SIZE-1]       *   b[7]
```

アセンブリ言語展開コード：macwl.src

```
          .IMPORT   _b
          .EXPORT   _a
          .EXPORT   _func
          .SECTION  P,CODE,ALIGN=4
_func:                                       ; function: func
                                             ; frame size=8
          STS.L          MACH,@-R15
          MOV            #33,R3
          STS.L          MACL,@-R15
          TST            R3,R3
          CLRMAC
          BT             L218
          MOV.L          L220+2,R1           ; _b
          SHLL           R3
          MOV.L          L220+6,R4           ; _a
          MOV            #-17,R2
          ADD            R4,R3
```

```
L219:
          MAC.W           @R4+,@R1+
          AND             R2,R1
          CMP/HI          R4,R3
          BT              L219
L218:
          STS             MACL,R0
          LDS.L           @R15+,MACL
          RTS
          LDS.L           @R15+,MACH
L220:
          .RES.W          1
          .DATA.L         _b
          .DATA.L         _a
          .SECTION        B,DATA,ALIGN=4
_a:                                                    ; static: a
          .RES.W          33
          .END
```

3.2.8　　64bit 乗算（1）

符号付き 64bit 乗算について、**表** 3.9 に示す関数を用意しています。

表 3.9　**符号付き** 64bit **乗算用組み込み関数一覧**

項番	項　目	書　式	説　明
1	符号付き 64bit 乗算の上位 32bit	long dmuls_h(long data1, long data2)	符号付き 32 ビット×符号付き３２ビット→符号付き 64 ビットの乗算を行い、結果の上位 32 ビットを返す
2	符号付き 64bit 乗算の下位 32bit	long dmuls_l(long data1, long data2)	符号付き 32 ビット×符号付き３２ビット→符号付き 64 ビットの乗算を行い、結果の下位 32 ビットを返す

符号付き 64bit 乗算は、SH-1 以外でサポートしています。

■使用例

C言語コード

```
#include <machine.h>

extern long data1,data2;
extern long result;
void  main(void)
{
    result = dmuls_h(data1,data2);     /*  符号付き 64bit 乗算を行います */
}
```

アセンブリ言語展開コード

```
         .IMPORT     _result
         .IMPORT     _data1
         .IMPORT     _data2
         .EXPORT     _main
         .SECTION     P,CODE,ALIGN=4
_main:                        ; function: main
                             ; frame size=8
         STS.L       MACL,@-R15
         STS.L       MACH,@-R15
         MOV.L       L11+2,R2   ; _data1
         MOV.L       L11+6,R5   ; _data2
         MOV.L       @R2,R6
         MOV.L       @R5,R2
         DMULS.L     R6,R2
         MOV.L       L11+10,R6  ; _result
         STS        MACH,R2
         MOV.L       R2,@R6
         LDS.L       @R15+,MACH
         RTS
         LDS.L       @R15+,MACL
L11:
         .RES.W      1
         .DATA.L     _data1
         .DATA.L     _data2
         .DATA.L     _result
         .END
```

3.2.9 64bit 乗算（2）

符号なし 64bit 乗算について、**表** 3.10 に示す関数を用意しています。

表 3.10　**符号なし** 64bit **乗算用組み込み関数一覧**

項番	項　目	書　式	説　明
1	**符号なし** 64bit **乗算の上位** 32bit	long dmulu_h(long data1, long data2)	**符号なし** 32 **ビット×符号なし３２ビット→符号なし** 64 **ビットの乗算を行い、結果の上位** 32 **ビットを返す**
2	**符号なし** 64bit **乗算の下位** 32bit	long dmulu_l(long data1, long data2)	**符号なし** 32 **ビット×符号なし３２ビット→符号なし** 64 **ビットの乗算を行い、結果の下位** 32 **ビットを返す**

符号なし 64bit 乗算は、SH-1 以外でサポートしています。

■使用例

C言語コード

```
#include <machine.h>

extern long data1,data2;
extern long result;
void  main(void)
{
    result = dmulu_h(data1,data2);     /*  符号付き 64bit 乗算を行います */
}
```

アセンブリ言語展開コード

```
         .IMPORT    _result
         .IMPORT    _data1
         .IMPORT    _data2
         .EXPORT    _main
         .SECTION    P,CODE,ALIGN=4
_main:                         ; function: main
                              ; frame size=8
         STS.L      MACL,@-R15
         STS.L      MACH,@-R15
         MOV.L      L11+2,R2   ; _data1
         MOV.L      L11+6,R5   ; _data2
         MOV.L      @R2,R6
         MOV.L      @R5,R2
         DMULU.L    R6,R2
         MOV.L      L11+10,R6  ; _result
         STS        MACH,R2
         MOV.L      R2,@R6
         LDS.L      @R15+,MACH
         RTS
         LDS.L      @R15+,MACL
L11:
         .RES.W     1
         .DATA.L    _data1
         .DATA.L    _data2
         .DATA.L    _result
         .END
```

3.2.10 データ上位・下位交換

データ上位・下位交換について、表 3.11 に示す関数を用意しています。

表 3.11 リングバッファ対応積和演算用組み込み関数一覧

項番	項　目	書　式	説　明
1	SWAP.B 命令	unsigned short swapb(unsigned short data)	２バイトデータの上位・下位 1 バイトを交換
2	SWAP.W 命令	unsigned long swapw(unsigned long data)	４バイトデータの上位・下位２バイトを交換
3	４バイトデータの上位・下位交換	unsigned long end_cnvl(unsigned long data)	４バイトデータを１バイトごとに上位・下位を逆順に並べる

■使用例

C言語コード

```
#include <machine.h>

extern unsigned short data;
extern unsigned short result;
void  main(void)
{
    result = swapb(data);
           /*data = 0x1234 の場合 result =0x3412 になります。*/
}
```

アセンブリ言語展開コード

```
         .IMPORT     _result
         .IMPORT     _data
         .EXPORT     _main
         .SECTION    P,CODE,ALIGN=4
_main:                           ; function: main
                                 ; frame size=0
         MOV.L       L11,R6      ; _data
         MOV.W       @R6,R2
         SWAP.B      R2,R6
         MOV.L       L11+4,R2    ; _result
         RTS
         MOV.W       R6,@R2
L11:
         .DATA.L     _data
         .DATA.L     _result
         .END
```

3.2.11 システムコール

■説明

システムコールを C 言語プログラムから発行できる組み込み関数の書式を下記に示します。なお、システムコールへの引数個数は 0〜4 で可変です。

ただし、TRAPA で直接 C の関数を呼び出し、復帰を RTE とすることはできません。

実際には、trapa_svc（C 組み込み関数）を使用しベクタテーブルに登録するハンドラ関数（これはアセンブラで記述してください）を作成して、ここで R0 機能コード判定して各ルーチンをコールしてください。

このアセンブルルーチンは RTE で復帰してください。

【書式】

```
ret=trapa_svc(int trap_no, int code,
 [type1 p1[, type2 p2[, type3 p3[, type4 p4]]]])
```

```
 trap_no          ：トラップ番号（定数で指定）
 code             ：機能コード、R0 に割り当てられます
 p1               ：第 1 引数、R4 に割り当てられます
 p2               ：第 2 引数、R5 に割り当てられます
 p3               ：第 3 引数、R6 に割り当てられます
 p4               ：第 4 引数、R7 に割り当てられます
 type1〜type4     ：引数の型は、汎整数型（[unsigned]char,
                    [unsigned]short,[unsigned]int,
                    [unsigned]long）、またはポインタ型です
```

■使用例

本関数を用いてトラップ番号#63 で指定できる OS のシステムコールを発行します。

C言語コード

```
#include <machine.h>
#define  SIG_SEM  0xffc8

void  main(void)
{
     :
     :
   trapa_svc(63, SIG_SEM, 0x05);
     :
     :
}
```

アセンブリ言語展開コード

```
          .EXPORT      _main
          .SECTION     P,CODE,ALIGN=4
_main:       :                           ; function: main
             :                           ; frame size=0
          MOV.L        L215+2,R0         ; H'0000FFC8
          MOV          #5,R4
          TRAPA        #63
             :
             :
          RTS
          NOP
L215:
          .RES.W       1
          .DATA.L      H'0000FFC8
          .END
```

ベクタテーブルの登録

```
void(*const vect[])(void)={
       :
       :
    HDR,.......                    /* トラップ 63 のベクタに HDR を登録 */
       :
}   ;
```

ハンドラ（アセンブラで記述）

```
        .IMPORT  _func
HDR:
                                       ;PR,R1～R7 の退避
                                       ;R0 機能コード判定で呼び出す関数を選択する
;              :
         MOV.L      label+2,R0
         JSR        @R0
                                       ; ->R4～R7 が破壊されなければ func に
                                       ; 正しいパラメータが渡されます
         NOP
;              :
                                       ; PR,R1～R7 の回復
         RTE                           ; ->例外からの復帰
                                       ; func のリターン値 R0 がそのまま
                                       ; trapa_svc のリターン値となる
          NOP
label:
          .RES.W       1
          .DATA.L      _func
          .END
```

3.2.12 プリフェッチ命令

■説明

SH-2A、SH2A-FPU、SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP でキャッシュのプリフェッチを行う組み込み関数の書式を下記に示します。なお、本組み込み関数は"-cpu=sh2a"、"-cpu=sh2afpu"、"-cpu=sh3"、"-cpu=sh3dsp"、"-cpu=sh4"、"-cpu=sh4a"、"-cpu=sh4aldsp" 指定時のみ有効です。

【書式】

```
void prefetch(void *p1)
p1  :  プリフェッチを行うアドレス
```

■使用例

C 言語コード

```
#include <umachine.h>
int a[1200];

f()
{
   int *pa = a;
        :
   prefetch(pa+8);
        :
}
```

アセンブリ言語展開コード

```
_f:                              ; function: f
           :
      ADD    #32,R6
      PREF   @R6
           :
```

3.2.13 LDTLB 命令

■説明

SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP で LDTLB 展開を行う組み込み関数の書式を下記に示します。なお、本組み込み関数は"-cpu=sh3"、"-cpu=sh3dsp"、"-cpu=sh4"、"-cpu=sh4a"、"-cpu=sh4aldsp" 指定時のみ有効です。

【書式】

```
void ldtlb(void)
```

■使用例

C 言語コード

```
 #include <machine.h>

void main(void)
 {
    ldtlb();
 }
```

アセンブリ言語展開コード

```
         .EXPORT     _main
         .SECTION    P,CODE,ALIGN=4
_main:                         ; function: main
                             ; frame size=0
         LDTLB
         NOP
         NOP
         RTS
         NOP
         .END
```

3.2.14 NOP 命令

■説明

NOP 命令に展開する組み込み関数の書式を下記に示します。

【書式】

```
void nop(void)
```

■使用例

C 言語コード

```
 #include <machine.h>

void  main(void)
{
    int a;
 if (a){
    nop();
 }
}
```

アセンブリ言語展開コード

```
         .EXPORT     _main
         .SECTION    P,CODE,ALIGN=4
_main:                         ; function: main
                             ; frame size=0
         TST        R2,R2
         BT         L12
         NOP
         RTS
         NOP
L12:
         RTS
         NOP
         .END
```

3.2.15 単精度浮動小数点演算

■説明

単精度浮動小数点に関する組み込み関数が SH-4 から追加になりました。演算の一覧表を**表 3.12** に示します。なお、浮動小数点ユニットの本組み込み関数は"-cpu=sh2e"、"-cpu=sh2afpu"、"-cpu=sh4"、"-cpu=sh4a"指定時のみ有効です。また、単精度浮動小数点ベクタ演算の本組み込み関数は"-cpu=sh4"、"-cpu=sh4a" 指定時のみ有効です。ただし、add4()、sub4()は、"-cpu=sh2afpu"指定時も有効です。

表 3.12 単精度浮動小数点の演算一覧

項番	項目	書式	説明
1	浮動小数点 ユニット	void set_fpscr(int cr)	FPSCR に cr(32 ビット)を設定する。
2		int get_fpscr()	FPSCR を参照する。
3	単精度浮動小数点 ベクタ演算	float fipr(float vect1[4], float vect2[4])	2 つのベクタの内積を求める。
4		float ftrv(float vec1[4], float vec2[4])	vec1(ベクタ)をあらかじめ ld_ext()でロードされた tbl(4×4 行列)で変換した結果を vec2(ベクタ)に格納する。
5		void ftrvadd(float vec1[4], float vec2[4], float vec3[4])	vec1(ベクタ)をあらかじめ ld_ext()でロードされた tbl(4×4 行列)で変換した結果と vec2(ベクタ)の和を vec3(ベクタ)に格納する。
6		void ftrvsub(float vec1[4], float vec2[4], float vec3[4])	vec1 (ベクタ)をあらかじめ ld_ext()でロードされた tbl(4×4 行列)で変換した結果と vec2(ベクタ)の差を vec3(ベクタ)に格納する。
7		void add4(float vec1[4], float vec2[4], float vec3[4])	vec1 (ベクタ)と vec2(ベクタ)の和を vec3(ベクタ)に格納する。
8		void sub4(float vec1[4], float vec2[4], float vec3[4])	vec1(ベクタ)と vec2(ベクタ)の差を vec3(ベクタ)に格納する。
9		void mtrx4mul(float mat1[4][4], float mat2[4][4])	mat1 (4×4 行列)をあらかじめ ld_ext()でロードした tbl(4×4 行列)で変換した結果を mat2 に格納する。
10		void mtrx4muladd(float mat1[4][4], float mat2[4][4], float mat3[4][4])	mat1(4×4 行列)をあらかじめ ld_ext()でロードした tbl(4×4 行列)で変換した結果と mat2(4×4 行列)の和を mat3(4×4 行列)に格納する。
11		void mtrx4mulsub(float mat1[4][4], float mat2[4][4], float mat3[4][4])	mat1(4×4 行列)をあらかじめ ld_ext()でロードした tbl(4×4 行列)で変換した結果と mat2(4×4 行列)との差を mat3 に格納する。

■使用例

4×4 行列の掛け算を行います。

一方の行列データをld_ext関数でロードしておく必要があります。

C 言語コード

```
#include<machine.h>
float table[4][4] ={{1.0,0.0,0.0,0.0},{0.0,1.0,0.0,0.0},
              {0.0,0.0,1.0,0.0},{0.0,0.0,0.0,1.0}} ;
float data1[4][4] ={{11.0,12.0,13.0,14.0},{15.0,16.0,17.0,18.0},
                   {11.0,12.0,13.0,14.0},{15.0,16.0,17.0,18.0}} ;
float data2[4][4] ={{0.0,0.0,0.0,0.0},{0.0,0.0,0.0,0.0},
                 {0.0,0.0,0.0,0.0},{0.0,0.0,0.0,0.0}} ;

void main()
{
 ld_ext(table) ;
 mtrx4mul(data1,data2) ;
}
```

アセンブリ言語展開コード

```
         .EXPORT     _table
         .EXPORT     _data1
         .EXPORT     _data2
         .EXPORT     _main
         .SECTION    P,CODE,ALIGN=4
_main:                          ; function: main
                               ; frame size=0
         MOV.L      L11+2,R2   ; _table
         MOV.L      L11+6,R6   ; _data2
         FRCHG
         FMOV.S     @R2+,FR0
         FMOV.S     @R2+,FR1
         FMOV.S     @R2+,FR2
         FMOV.S     @R2+,FR3
         FMOV.S     @R2+,FR4
         FMOV.S     @R2+,FR5
         FMOV.S     @R2+,FR6
         FMOV.S     @R2+,FR7
         FMOV.S     @R2+,FR8
         FMOV.S     @R2+,FR9
         FMOV.S     @R2+,FR10
         FMOV.S     @R2+,FR11
         FMOV.S     @R2+,FR12
         FMOV.S     @R2+,FR13
         FMOV.S     @R2+,FR14
         FMOV.S     @R2+,FR15
         FRCHG
         ADD        #-64,R2
         MOV.L      L11+10,R2  ; _data1
         ADD        #16,R6
         FMOV.S     @R2+,FR0
         FMOV.S     @R2+,FR1
         FMOV.S     @R2+,FR2
         FMOV.S     @R2+,FR3
         FTRV       XMTRX,FV0
         FMOV.S     FR3,@-R6
         FMOV.S     FR2,@-R6
         FMOV.S     FR1,@-R6
         FMOV.S     FR0,@-R6
         FMOV.S     @R2+,FR0
         ADD        #32,R6
         FMOV.S     @R2+,FR1
         FMOV.S     @R2+,FR2
         FMOV.S     @R2+,FR3
         FTRV       XMTRX,FV0
         FMOV.S     FR3,@-R6
         FMOV.S     FR2,@-R6
         FMOV.S     FR1,@-R6
         FMOV.S     FR0,@-R6
```

```
         FMOV.S     @R2+,FR0
         ADD        #32,R6
         FMOV.S     @R2+,FR1
         FMOV.S     @R2+,FR2
         FMOV.S     @R2+,FR3
         FTRV       XMTRX,FV0
         FMOV.S     FR3,@-R6
         FMOV.S     FR2,@-R6
         FMOV.S     FR1,@-R6
         FMOV.S     FR0,@-R6
         FMOV.S     @R2+,FR0
         ADD        #32,R6
         FMOV.S     @R2+,FR1
         FMOV.S     @R2+,FR2
         FMOV.S     @R2+,FR3
         FTRV       XMTRX,FV0
         FMOV.S     FR3,@-R6
         FMOV.S     FR2,@-R6
         FMOV.S     FR1,@-R6
         RTS
         FMOV.S     FR0,@-R6
L11:
         .RES.W     1
         .DATA.L    _table
         .DATA.L    _data2
         .DATA.L    _data1
         .SECTION   D,DATA,ALIGN=4
_table:                         ; static: table
         .DATA.L    H'3F800000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'3F800000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'3F800000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'3F800000
_data1:                         ; static: data1
         .DATA.L    H'41300000
         .DATA.L    H'41400000
         .DATA.L    H'41500000
         .DATA.L    H'41600000
         .DATA.L    H'41700000
         .DATA.L    H'41800000
         .DATA.L    H'41880000
         .DATA.L    H'41900000
         .DATA.L    H'41300000
         .DATA.L    H'41400000
         .DATA.L    H'41500000
         .DATA.L    H'41600000
         .DATA.L    H'41700000
         .DATA.L    H'41800000
         .DATA.L    H'41880000
         .DATA.L    H'41900000
_data2:                         ; static: data2
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
```

```
        .DATA.L     H'00000000
        .DATA.L     H'00000000
        .DATA.L     H'00000000
        .DATA.L     H'00000000
        .END
```

■使用例

ベクトルと行列の掛け算を行います。

行列データをld_extにてロードしておく必要があります。

C言語コード

```
#include<machine.h>
float table[4][4]={{1.0,2.0,3.0,4.0},{5.0,6.0,7.0,8.0},
                   {8.0,7.0,6.0,5.0},{4.0,3.0,2.0,1.0}} ;
float data1[4]   = {11.0,12.0,13.0,14.0} ;
float data2[4]  = {0.0,0.0,0.0,0.0} ;

void main()
{
    ld_ext(table) ;
    ftrv(data1,data2) ;
}
```

アセンブリ言語展開コード

```
        .EXPORT     _table
        .EXPORT     _data1
        .EXPORT     _data2
        .EXPORT     _main
        .SECTION    P,CODE,ALIGN=4
_main:                          ; function: main
                                ; frame size=0
        MOV.L       L11+2,R2    ; _table
        FRCHG
        FMOV.S      @R2+,FR0
        FMOV.S      @R2+,FR1
        FMOV.S      @R2+,FR2
        FMOV.S      @R2+,FR3
        FMOV.S      @R2+,FR4
        FMOV.S      @R2+,FR5
        FMOV.S      @R2+,FR6
        FMOV.S      @R2+,FR7
        FMOV.S      @R2+,FR8
        FMOV.S      @R2+,FR9
        FMOV.S      @R2+,FR10
        FMOV.S      @R2+,FR11
        FMOV.S      @R2+,FR12
        FMOV.S      @R2+,FR13
        FMOV.S      @R2+,FR14
        FMOV.S      @R2+,FR15
        FRCHG
        ADD         #-64,R2
        MOV.L       L11+6,R2    ; _data1
        FMOV.S      @R2+,FR0
        FMOV.S      @R2+,FR1
        FMOV.S      @R2+,FR2
        FMOV.S      @R2+,FR3
        MOV.L       L11+10,R2   ; _data2
        FTRV        XMTRX,FV0
        ADD         #16,R2
        FMOV.S      FR3,@-R2
        FMOV.S      FR2,@-R2
        FMOV.S      FR1,@-R2
        RTS
        FMOV.S      FR0,@-R2
L11:
        .RES.W      1
        .DATA.L     _table
        .DATA.L     _data1
        .DATA.L     _data2
        .SECTION    D,DATA,ALIGN=4
```

```
_table:                          ; static: table
         .DATA.L    H'3F800000
         .DATA.L    H'40000000
         .DATA.L    H'40400000
         .DATA.L    H'40800000
         .DATA.L    H'40A00000
         .DATA.L    H'40C00000
         .DATA.L    H'40E00000
         .DATA.L    H'41000000
         .DATA.L    H'41000000
         .DATA.L    H'40E00000
         .DATA.L    H'40C00000
         .DATA.L    H'40A00000
         .DATA.L    H'40800000
         .DATA.L    H'40400000
         .DATA.L    H'40000000
         .DATA.L    H'3F800000
_data1:                          ; static: data1
         .DATA.L    H'41300000
         .DATA.L    H'41400000
         .DATA.L    H'41500000
         .DATA.L    H'41600000
_data2:                          ; static: data2
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .DATA.L    H'00000000
         .END
```

・2 つのベクトルの内積を求めます。

C 言語コード

```
#include<machine.h>
float ret = 0;
float data1[]={1.0,2.0,3.0,4.0} ;
float data2[]={11.0,12.0,13.0,14.0} ;

void main()
{
  ret = fipr (data1,data2) ;
}
```

アセンブリ言語展開コード

```
         .EXPORT    _ret
         .EXPORT    _data1
         .EXPORT    _data2
         .EXPORT    _main
         .SECTION   P,CODE,ALIGN=4
_main:                          ; function: main
                              ; frame size=0
         MOV.L      L11,R2     ; _data1
         FMOV.S     @R2+,FR0
         FMOV.S     @R2+,FR1
         FMOV.S     @R2+,FR2
         FMOV.S     @R2+,FR3
         MOV.L      L11+4,R2   ; _data2
         FMOV.S     @R2+,FR4
         FMOV.S     @R2+,FR5
         FMOV.S     @R2+,FR6
         FMOV.S     @R2+,FR7
         MOV.L      L11+8,R2   ; _ret
         FIPR       FV4,FV0
         RTS
         FMOV.S     FR3,@R2
L11:
         .DATA.L    _data1
         .DATA.L    _data2
         .DATA.L    _ret
         .SECTION   D,DATA,ALIGN=4
_ret:                           ; static: ret
         .DATA.L    H'00000000
```

```
_data1:                         ; static: data1
         .DATA.L    H'3F800000
         .DATA.L    H'40000000
         .DATA.L    H'40400000
         .DATA.L    H'40800000
_data2:                         ; static: data2
         .DATA.L    H'41300000
         .DATA.L    H'41400000
         .DATA.L    H'41500000
         .DATA.L    H'41600000
         .END
```

■注意事項

(1) 単精度浮動小数点ベクタ演算組み込み関数は、SH-4、SH-4Aのみ有効です。
（add4()、sub4()はSH2A-FPUでも有効です。）

(2) ベクタ演算組み込み関数は、割り込み関数で使用するとき以下の点に注意してください。
組み込み関数ld_ext(float[4][4])とst_ext(float[4][4])は、浮動小数点ステータス制御レジスタ(FPSCR)の浮動小数点レジスタバンクビット(FR)を変更して拡張レジスタにアクセスするため、割り込み関数内で、組み込み関数ld_ext(float[4][4])とst_ext(float[4][4])を使用しているときには、ベクタ演算組み込み関数の前後で割り込みマスクを変更してください。以下に例を示します。

例)

```
#pragma interrupt (intfunc)
void intfunc(){
   ...
   ld_ext();
   ...
}
void normfunc(){
   ...
   int maskdata=get_imask();    /*割り込みマスクの退避*/
   set_imask(15);               /*割り込みマスクの設定*/
   ld_ext(mat1);
   ftrv(vec1,vec2);
   set_imask(maskdata);         /*割り込みマスクの回復*/
   ...
}
```

(3) 組み込み関数mtrx4mul、mtrx4muladd、mtrx4mulsubは4×4行列の演算のため行列A×行列Bと行列B×行列Aの結果は必ずしも一致しません。

例)

```
extern float matA[][];
extern float matB[][];
int judge(){
   float data1[4][4], data2[4][4];
   set_imask(15);
   ld_ext(matA);
   mtrx4mul(matB,data1);     /* data1 = matB×matA */
   ld_ext(matB);
   mtrx4mul(matA,data2);     /* data2 = matA×matB */
   ....  /*このときのdata1[][]とdata2[][]の各要素は必ずしも一致しません*/
}
```

3.2.16 拡張レジスタのアクセス

■説明

拡張レジスタのアクセスについて、**表** 3.13 に示す関数を用意しています。

表 3.13　**拡張レジスタのアクセス用組み込み関数一覧**

項番	項目	書式	説明
1	拡張レジスタへのアクセス	void ld_ext(float mat[4][4])	tbl(4×4 行列)を拡張レジスタにロードする。 例 extern float tbl[4][4]; このとき ld_ext(tbl)は、tbl の内容を拡張レジスタにロードする。
2		void st_ext(float mat[4][4])	拡張レジスタ の内容を tbl(4×4 行列)にストアする。 例 extern float tbl[4][4]; このとき st_ext(tbl)は、拡張レジスタの内容を tbl にストアする。

【注】 (1) 拡張レジスタアクセス組み込み関数は SH-4、SH-4A のみ有効です。

(2) 本関数を割り込み関数内で使用する祭には割り込みマスクの変更が必要です。詳細は前項「3.2.15 単精度浮動小数点演算の注意事項(2)」を参照してください。

3.2.17 DSP 命令

■説明

DSP 命令について、**表** 3.14 に示す関数を用意しています。

表 3.14 DSP 命令用組み込み関数一覧

項番	項目	書式	説明
1	絶対値	long __fixed pabs_lf (long __fixed data) long __accum pabs_la (long __accum data)	絶対値を求めます 絶対値を求めた結果、リターン値の型(pabs_lf()は long __fixed 型, pabs_la()は､long __accum 型)の値として表現できない場合の動作は保証されません
2	MSB 検出	__fixed pdmsb_lf (long __fixed data) __fixed pdmsb_la (long __accum data)	MSB 検出します(データを正規化するためのシフト量を求めます)
3	算術シフト	long __fixed psha_lf (long __fixed data, int count) long __accum psha_la (long __accum data, int count)	算術シフトを行います count 指定可能範囲は、-32〜+32 です。 正の値を指定した場合は、左シフト. 負の値を指定した場合は、その絶対値分だけ右シフトを行います。 範囲外の値を指定した場合の動作は保証されません。
4	丸め誤差	__accum rndtoa (long __accum data) __fixed rndtof (long __fixed data)	丸め誤差を行います
5	ビットパターンコピー	long __fixed long_as_lfixed (long data) long lfixed_as_long (long __fixed data)	ビットパターンコピーを行います (汎用レジスタ ⇔ DSP レジスタ間のコピー)
6	モジュロアドレッシング設定	void set_circ_x (__X__circ __fixed array[], size_t size) void set_circ_y (__Y__circ __fixed array[], size_t size)	モジュロアドレッシングの設定を行います
7	モジュロアドレッシング解除	void clr_circ(void)	モジュロアドレッシングの解除を行います (SR の右から 10、11bit 目をゼロクリアします)
8	CS ビット(DSR レジスタ)設定	void set_cs (unsigned int mode)	CS ビットの設定を行います mode=0：キャリ／ボローmode mode=1：負値 mode mode=2：ゼロ値 mode mode=3：オーバフローmode mode=4：符号付大 mode mode=5：符号付以上 mode

■使用例

C 言語コード

```
#include <machine.h>
__circ __X __fixed input[4] = {0.0r, 0.25r, 0.5r, 0.25r};
 __Y __fixed output[8];

void  main(void)
```

```
{
        int i;
        set_circ_x(input, sizeof(input)); /* モジュロアドレッシング設定 */
        for(i=0; i < 8; ii++){
        utput[i] = input[i];
  }
        clr_circ(); /* モジュロアドレッシングの解除 */
}
```

アセンブリ言語展開コード

```
        .EXPORT     _output
        .EXPORT     _input
        .EXPORT     _main
        .SECTION    P,CODE,ALIGN=4
_main:                          ; function: main
                                ; frame size=0
        MOV.L       L13+4,R5    ; _input
        EXTU.W      R5,R2
        MOV         R5,R6
        ADD         #6,R6
        SHLL16      R6
        ADD         R6,R2
        LDC         R2,MOD
        STC         SR,R2
        MOV.W       L13,R4      ; H'F3FF
        AND         R4,R2
        MOV         R2,R6
        MOV         #4,R2       ; H'00000004
        SHLL8       R2
        OR          R2,R6
        LDC         R6,SR
        MOV         #8,R2       ; H'00000008
        MOV.L       L13+8,R6    ; _output
L11:
        MOVX.W      @R5+,X1
        DT          R2
        PCOPY       X1,A0
        BF/S        L11
        MOVY.W      A0,@R6+
        STC         SR,R2
        AND         R4,R2
        LDC         R2,SR
        RTS
        NOP
L13:
        .DATA.W     H'F3FF
        .RES.W      1
        .DATA.L     _input
        .DATA.L     _output
        .SECTION    $XD,DATA,ALIGN=4
_input:                         ; static: input
        .DATA.W     H'0000,H'2000,H'4000,H'2000
        .SECTION    $YB,DATA,ALIGN=4
_output:                        ; static: output
        .RES.W      8
        .END
```

3.2.18 正弦・余弦

■説明

angle で指定された角度から、正弦・余弦の近似値を計算し、その結果を sinv, cosv の指す領域に設定します。

【書式】

```
void fsca(long angle, float * sinv, float * cosv)
```

■使用例

C 言語コード

```
#include <machine.h>
long angle = (45<<16)/360;  /* 45 度 */
float  * sinv;
float  * cosv;
void  main(void)
{
   fsca(angle, sinv, cosv);
}
```

アセンブリ言語展開コード

```
        .EXPORT    _sinv
        .EXPORT    _cosv
        .EXPORT    _angle
        .EXPORT    _main
        .SECTION   P,CODE,ALIGN=4
_main:                          ; function: main
                               ; frame size=0
        MOV.L      L11+2,R6   ; _angle
        MOV.L      @R6,R2
        MOV.L      L11+6,R6   ; _sinv
        LDS        R2,FPUL
        FSCA       FPUL,DR0
        MOV.L      @R6,R2
        MOV.L      L11+10,R6  ; _cosv
        FMOV.S     FR0,@R2
        MOV.L      @R6,R2
        RTS
        FMOV.S     FR1,@R2
L11:
        .RES.W     1
        .DATA.L    _angle
        .DATA.L    _sinv
        .DATA.L    _cosv
        .SECTION   D,DATA,ALIGN=4
_angle:                         ; static: angle
        .DATA.L    H'00002000
        .SECTION   B,DATA,ALIGN=4
_sinv:                          ; static: sinv
        .RES.L     1
_cosv:                          ; static: cosv
        .RES.L     1
        .END
```

3.2.19 平方根の逆数

■説明

平方根の逆数の近似値を計算し、求めます。

【書式】

```
float fsrra(float data)
```

■使用例

C言語コード

```
#include <machine.h>
float  data;
float result;
void  main(void)
{
   result=fsrra(data);
}
```

アセンブリ言語展開コード

```
         .EXPORT    _data
         .EXPORT    _result
         .EXPORT    _main
         .SECTION   P,CODE,ALIGN=4
_main:                        ; function: main
                             ; frame size=0
         MOV.L      L11,R2     ; _data
         FMOV.S     @R2,FR9
         MOV.L      L11+4,R2   ; _result
         FSRRA      FR9
         RTS
         FMOV.S     FR9,@R2
L11:
         .DATA.L    _data
         .DATA.L    _result
         .SECTION   B,DATA,ALIGN=4
_data:                        ; static: data
         .RES.L     1
_result:                      ; static: result
         .RES.L     1
         .END
```

3.2.20 命令キャッシュ無効化

■説明

命令キャッシュの無効化を行います。

【書式】

```
void icbi(void *p)
```

■使用例

C言語コード

```
#include <machine.h>
extern int *p;
void  main(void)
{
   icbi(p);
}
```

アセンブリ言語展開コード

```
        .IMPORT     _p
        .EXPORT     _main
        .SECTION    P,CODE,ALIGN=4
_main:                       ; function: main
                             ; frame size=0
        MOV.L      L11+2,R6   ; _p
        MOV.L      @R6,R2
        ICBI       @R2
        RTS
        NOP
L11:
        .RES.W      1
        .DATA.L     _p
        .END
```


RENESAS

3.2.21 キャッシュブロック操作

■説明

キャッシュブロックの操作を行います。

【書式】

void ocbi(void *p)	キャッシュブロックの無効化
void ocbp(void *p)	キャッシュブロックのパージ
void ocbwb(void *p)	キャッシュブロックの書き戻し

■使用例

C言語コード

```
#include <machine.h>
extern  int *p;
void  main(void)
{
   ocbi(p);
}
```

アセンブリ言語展開コード

```
        .IMPORT     _p
        .EXPORT     _main
        .SECTION    P,CODE,ALIGN=4
_main:                          ; function: main
                               ; frame size=0
        MOV.L       L11+2,R6   ; _p
        MOV.L       @R6,R2
        OCBI        @R2
        RTS
        NOP
L11:
        .RES.W      1
        .DATA.L     _p
        .END
```

3.2.22 命令キャッシュプリフェッチ

■説明

32 バイト境界で始まる 32 バイトの命令ブロックを命令キャッシュに読み込みます。

【書式】

```
void prefi(void *p)
```

■使用例

C 言語コード

```
#include <machine.h>
void *pa;
void  main(void)
{
   prefi(pa);
}
```

アセンブリ言語展開コード

```
        .EXPORT    _pa
        .EXPORT    _main
        .SECTION   P,CODE,ALIGN=4
_main:                       ; function: main
                            ; frame size=0
        MOV.L      L11+2,R6   ; _pa
        MOV.L      @R6,R2
        PREFI      @R2
        RTS
        NOP
L11:
        .RES.W     1
        .DATA.L    _pa
        .SECTION   B,DATA,ALIGN=4
_pa:                         ; static: pa
        .RES.L     1
        .END
```

3.2.23 システム同期

■説明

SYNCO 命令に展開します。

SYNCO 命令は、データ操作の同期に使用します。SYNCO 命令を実行すると SYNCO 命令より前のデータ操作を完了してから SYNCO 命令より後の命令を開始します。

【書式】

```
void synco(void)
```

■使用例

C 言語コード

```
#include <machine.h>

void  main(void)
{
   synco();
}
```

アセンブリ言語展開コード

```
        .EXPORT     _main
        .SECTION    P,CODE,ALIGN=4
_main:                        ; function: main
                             ; frame size=0
        SYNCO
        RTS
        NOP
        .END
```

3.2.24 T ビット参照・設定

■説明

T ビットの設定・参照について、表 3.15 に示す関数を用意しています。

表 3.15 T ビット用組み込み関数一覧

項番	項目	書式	説明
1	T ビットの参照	int movt(void)	SR レジスタの T ビットの値を参照
2	T ビットのクリア	void clrt(void)	SR レジスタの T ビットをクリア
3	T ビットのセット	void sett(void)	SR レジスタの T ビットをセット

■使用例

SR レジスタの T ビットの値を参照することができます。参照した値は、0 または 1 となります。

C 言語コード

```
#include <machine.h>
int sr_t;
void func(void)
{
   sr_t = movt();
}
```

アセンブリ言語展開コード

```
_func:
        MOVT       R2
        MOV.L      L11,R6     ; _sr_t
        RTS
        MOV.L      R2,@R6
```

3.2.25 連結レジスタの中央切り出し

■説明

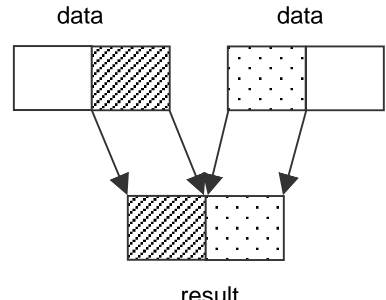
2 つの 32 ビットデータを連結した 64 ビットの内容から中央の 32 ビットを切り出します。

【書式】

```
unsigned long xtrct(unsigned long data1, unsigned long data2)
```

■使用例

data1 と data2 を連結して中央 32 ビットを切り出します。



C 言語コード

```
#include <machine.h>
unsigned long result, data1, data2;
void main(void)
{
    result = xtrct(data1,data2);
}
```

アセンブリ言語展開コード

```
_main:
        MOV.L      L11,R1     ; _data2
        MOV.L      L11+4,R4   ; _data1
        MOV.L      @R1,R2
        MOV.L      @R4,R5
        MOV.L      L11+8,R6   ; _result
        XTRCT      R5,R2
        RTS
        MOV.L      R2,@R6
```

3.2.26 キャリ付き加算

■説明

キャリ付き加算について、**表** 3.16 に示す関数を用意しています。

表 3.16　キャリ付き加算用組み込み関数一覧

項番	項　目	書　式	説　明
1	キャリ付き加算	long addc(long data1, long data2)	2 つのデータとＴビットを加算し、キャリをＴビットに反映
2		int ovf_addc(long data1, long data2)	2 つのデータとＴビットを加算し、キャリを参照
3		long addv(long data1, long data2)	2 つのデータを加算し、キャリをＴビットに反映
4		int ovf_addv(long data1, long data2)	2 つのデータを加算し、キャリを参照

■使用例

data1 と data2 とＴビットを加算し、キャリをＴビットに反映します。

Ｃ言語コード

```
#include <machine.h>
long result, data1, data2;
void main(void)
{
   result = addc(data1,data2); /* result = data1 + data2 + Ｔビット */
}
```

アセンブリ言語展開コード

```
_main:
        MOV.L      L11,R1     ; _data1
        MOV.L      L11+4,R2   ; _data2
        MOV.L      @R1,R4
        MOV.L      @R2,R2
        MOV.L      L11+8,R5   ; _result
        ADDC       R4,R2
        RTS
        MOV.L      R2,@R5
```

■注意事項

(1)　addc関数およびovf_addc関数はＴビットの内容を参照しますので、直前に比較やシフトなどを記述した場合、その演算結果がＴビットに反映されるため、本関数の動作が正しく行われない場合があります。

3.2.27 ボロー付き減算

■説明

ボロー付き減算について、**表** 3.17 に示す関数を用意しています。

表 3.17 ボロー付き減算用組み込み関数一覧

項番	項　目	書　式	説　明
1	**ボロー付き減算**	long subc(long data1, long data2)	data1 **から** data2 **と** T **ビットを減算し、ボローを** T **ビットに反映**
2		int unf_subc(long data1, long data2)	data1 **から** data2 **と** T **ビットを減算し、ボローを参照**
3		long subv(long data1, long data2)	data1 **から** data2 **を減算し、ボローを** T **ビットに反映**
4		int unf_subv(long data1, long data2)	data1 **から** data2 **を減算し、ボローを参照**

■使用例

data1 から data2 とＴビットを減算し、ボローをＴビットに反映します。

C 言語コード

```
#include<machine.h>
long result, data1, data2;
void main(void)
{
   result = subc(data1,data2); /* result = data1 - data2 - Ｔビット */
}
```

アセンブリ言語展開コード

```
  _main:
          MOV.L      L11,R1     ; _data1
          MOV.L      L11+4,R4   ; _data2
          MOV.L      @R1,R6
          MOV.L      @R4,R5
          MOV.L      L11+8,R1   ; _result
          SUBC       R5,R6
          RTS
          MOV.L      R6,@R1
```

■注意事項

（1） subc関数およびunf_subc関数はＴビットの内容を参照しますので、直前に比較やシフトなどを記述した場合、その演算結果がＴビットに反映されるため、本関数の動作が正しく行われない場合があります。

3.2.28 符号反転

■説明

0 から data と T ビットを減算し、ボローを T ビットに反映します。

【書式】

```
long negc(long data)
```

■使用例

0 から data と T ビットを減算し、ボローを T ビットに反映します。

C 言語コード

```
#include <machine.h>
long result, data;
void main(void)
{
    result = negc(data); /* result = 0 - data - Tビット */
}
```

アセンブリ言語展開コード

```
_main:
        MOV.L       L11,R1     ; _data
        MOV.L       L11+4,R5   ; _result
        MOV.L       @R1,R4
        NEGC        R4,R2
        RTS
        MOV.L       R2,@R5
```

3.2.29 1 ビット除算

■説明

1 ビット除算について、**表 3.18** に示す関数を用意しています。

表 3.18 1 ビット除算用組み込み関数一覧

項番	項 目	書 式	説 明
1	1 ビット除算	unsigned long div1(unsigned long data1, unsigned long data2)	data1/data2 の 1 ステップ除算を行い、結果を T ビットに反映
2		int div0s(long data1, long data2)	data1/data2 の符号付き除算の初期設定をし、T ビットを参照
3		void div0u(void)	符号なし除算の初期設定

■使用例

１ビット除算を繰り返し実行することで商を求めることができます。以下は、
d1(32bit) ÷ d2(16bit) = ret(16bit) の符号なし除算をする場合の例です。

C 言語コード

```
#include <machine.h>
unsigned long data1,data2;
unsigned long result;
void main(void)
{
    unsigned long d1,d2;

    d1 = data1;
    d2 = data2;
    d2 <<= 16;   /* 除数を上位 16bit とし、下位 16bit を 0 とする */
    div0u();              /* 符号なし除算の初期値設定 */
    d1 = div1(d1, d2);    /* １ステップ除算を 16 回繰り返す */
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    result = rotcl(d1);    /* rotcl は、ROTCL 命令を生成する組み込み関数です */
                  /* 最後のステップ除算結果である T ビットを商に反映します */
}
```

アセンブリ言語展開コード

```
_main:
        MOV.L      L11,R1     ; _data2
        MOV.L      @R1,R5
        SHLL16     R5
        DIV0U
        MOV.L      L11+4,R2   ; _data1
        MOV.L      @R2,R2
        DIV1       R5,R2
        DIV1       R5,R2
        DIV1       R5,R2
        DIV1       R5,R2
        MOV        R2,R6
        DIV1       R5,R6
        MOV        R6,R2
        DIV1       R5,R2
        DIV1       R5,R2
        DIV1       R5,R2
        DIV1       R5,R2
        DIV1       R5,R2
        DIV1       R5,R2
        MOV        R2,R6
        DIV1       R5,R6
        MOV        R6,R2
        DIV1       R5,R2
        DIV1       R5,R2
        DIV1       R5,R2
        DIV1       R5,R2
        ROTCL      R2
        MOV.L      L11+8,R4   ; _result
        RTS
        MOV.L      R2,@R4
```

■注意事項

（1） div1関数を繰り返し使用することで除算をすることができますが、繰り返す間はMビット、Qビット、Tビットを書き換えないでください。（比較やシフトによってもTビットを書き換えます）

（2） 関数の直前で必ずdiv0s()またはdiv0u()を使用してMビット、Qビット、Tビットの初期化をしてください。

3.2.30 回転

■説明

回転について、**表 3.19** に示す関数を用意しています。

表 3.19 回転用組み込み関数一覧

項番	項　目	書　式	説　明
1	回転	unsigned long rotl(unsigned long data)	データを 1 ビット左方向に回転し、オペランドの外へ出たビットを T ビットに反映
2		unsigned long rotr(unsigned long data)	データを 1 ビット右方向に回転し、オペランドの外へ出たビットを T ビットに反映
3		unsigned long rotcl(unsigned long data)	データを 1 ビット左方向に T ビットを含めて回転し、オペランドの外へ出たビットを T ビットに反映
4		unsigned long rotcr(unsigned long data)	データを 1 ビット右方向に T ビットを含めて回転し、オペランドの外へ出たビットを T ビットに反映

■使用例

data を左に 1 ビット回転し、オペランド外に出たビットを T ビットに反映します。

C言語コード

```
#include <machine.h>
unsigned long result, data;
void main(void)
{
   result = rotl(data);
}
```

アセンブリ言語展開コード

```
_main:
        MOV.L       L11,R2      ; _data
        MOV.L       L11+4,R6    ; _result
        MOV.L       @R2,R2
        ROTL        R2          ; ROTL 命令を生成します。
        RTS
        MOV.L       R2,@R6
```

■注意事項

rotcl 関数および rotcr 関数は T ビットの内容を参照しますので、直前に比較やシフトなどを記述した場合、その演算結果が T ビットに反映されるため、本関数の動作が正しく行われない場合があります。

3.2.31 シフト

■説明

シフトについて、**表** 3.20 に示す関数を用意しています。

表 3.20 **シフト用組み込み関数一覧**

項番	項 目	書 式	説 明
1	**シフト**	unsigned long shll(unsigned long data)	**データを 1 ビット左シフトし、オペランドの外へ出たビットを T ビットに反映**
2		unsigned long shlr(unsigned long data)	**データを論理的に 1 ビット右シフトし、オペランドの外へ出たビットを T ビットに反映**
3		long shar(long data)	**データを算術的に 1 ビット右シフトし、オペランドの外へ出たビットを T ビットに反映**

■使用例

data を左に１ビットシフトし、オペランド外に出たビットをＴビットに反映します。

Ｃ言語コード

```
#include <machine.h>
unsigned long result, data;
void main(void)
{
   result = shll(data);
}
```

アセンブリ言語展開コード

```
_main:
        MOV.L      L11,R2     ; _data
        MOV.L      L11+4,R6   ; _result
        MOV.L      @R2,R2
        SHLL       R2         ; SHLL 命令を生成します。
        RTS
        MOV.L      R2,@R6
```

3.2.32 飽和演算

■説明

飽和演算について、**表** 3.21 に示す関数を用意しています。

表 3.21 **飽和演算用組み込み関数一覧**

項番	項　目	書　式	説　明
1	符号付き 1 バイトデータ飽和演算	long clipsb(long data)	データが-128～127 の範囲内の場合はその値を、範囲外の場合は上限値もしくは下限値を設定
2	符号付き 2 バイトデータ飽和演算	long clipsw(long data)	データが-32768～32767 の範囲内の場合はその値を、範囲外の場合は上限値もしくは下限値を設定
3	符号なし 1 バイトデータ飽和演算	unsigned long clipub(unsigned long data)	データが 0～255 の範囲内の場合はその値を、範囲外の場合は上限値を設定
4	符号なし 2 バイトデータ飽和演算	unsigned long clipuw(unsigned long data)	データが 0～65535 の範囲内の場合はその値を、範囲外の場合は上限値を設定

本組み込み関数は、"-cpu=sh2a" または、"-cpu=sh2afpu"指定時のみ有効です。

■使用例

data が-128～127 の範囲内の場合はその値を、範囲外の場合は上限値もしくは下限値を設定します。

C 言語コード

```
#include <machine.h>
long result, data;
void main(void)
{
    result = clipsb(data);  /* result は、-128～127 の範囲の値になります */
}
```

アセンブリ言語展開コード

```
_main:
        MOV.L      L11,R2     ; _data
        MOV.L      @R2,R2
        MOV.L      L11+4,R6   ; _result
        CLIPS.B   R2
        RTS
        MOV.L      R2,@R6
```

3.2.33 TBR 設定・参照

■説明

ジャンプテーブルベースレジスタ(TBR)の設定・参照について、表 3.22 に示す関数を用意しています。

表 3.22 TBR 用組み込み関数一覧

項番	項目	書式	説明
1	TBR 設定	void set_tbr(void *data)	TBR に data を設定
2	TBR 参照	void *get_tbr(void)	TBR の値を参照

本組み込み関数は、"-cpu=sh2a" または、"-cpu=sh2afpu"指定時のみ有効です。

■使用例

TBR にデータを設定します。
TBR 相対関数呼び出しをするときに生成するジャンプテーブルを TBR に設定するときに使用します。

C 言語コード

```
#include <machine.h>
void main(void){
   set_tbr(__sectop("$TBR"));   /* $TBR セクションの先頭を TBR に設定する */
}
```

アセンブリ言語展開コード

```
_main:
        MOV.L      L11,R2   ; STARTOF $TBR
        RTS
        LDC        R2,TBR
L11:
        .DATA.L     STARTOF $TBR
```

3.3 インライン展開

3.3.1 関数のインライン展開

■説明

関数のインライン展開機能は、プログラムの実行速度の向上を行う場合に使います。通常、関数の呼び出しは一連の処理がある部分に分岐して処理させる形を取りますが、本機能は、関数呼び出し位置に関数の処理を展開し、分岐部分の命令を削除して速度向上を行うものです。特にループ内などで呼ばれる関数を展開すると大きな効果が期待できます。

関数のインライン展開機能には、以下の二種類があります。

(1) **自動インライン展開**

コンパイル時に"-speed"オプションを指定すると関数の自動インライン展開機能が働き､小さな関数を自動的に展開します。また､関数の自動インライン展開機能をより細かく制御するために"-inline"オプションで展開する関数の大きさを指定することができます。Ver.6 以前では、関数の大きさはノード数（宣言部を除く変数、演算子などの語句の数）で指定します（"-inline"オプションのデフォルト値は 20）。Ver.7 以降では、プログラムサイズが何%増加するまでインライン展開を行うかを指定できます。

【書式】

```
shc -speed [-inline=<数値>]・・・
```

(2) **制御文によるインライン展開**

#pragma inline 文でインライン展開したい関数を指定します。

【書式】

```
#pragma inline (<関数名> [,<関数名>・・・] )
```

■使用例

ループ内で呼び出している関数をインライン展開します。

(1) **自動インライン展開**

次のプログラムを"-speed"オプションを付けてコンパイルすると、f がインライン展開されます。

C 言語コード

```
extern int *z;
int  f (int p1, int p2)    /* 展開される関数 */
{
   if (p1 > p2)
       return p1;
   else if (p1 < p2)
       return p2;
   else
       return 0;
}

void  g (int *x, int *y, int count)
{
   for ( ; count>0; count--, z++, x++, y++)
         *z = f(*x, *y);
}
```

(2) **インライン展開**

#pragma inline で指定された f1 および f2 がインライン展開されます。

C 言語コード

```
int  v,w,x,y;
#pragma  inline(f1,f2)     /* インライン展開する関数の指定   */
```

```
int  f1(int a, int b)      /* 展開される関数              */
{
   return (a+b)/2;
}
int  f2(int  c, int  d)    /* 展開される関数              */
{
   return (c-d)/2;
}
void  g ()
{
   int i;
   for(i=0;i<100;i++){
      if(f1(x,y) == f2(v,w))
         sleep();
   }
}
```

■注意事項

（1） #pragma inlineは、関数本体の定義の前に指定してください。

（2） #pragma inlineで指定した関数に対しても、外部定義を生成しますので、複数ファイルでインクルードされるファイル中にインライン展開関数の実体を記述する場合は、必ず関数の宣言にstaticを指定してください。

（3） 以下の関数はインライン展開しません。

- 可変パラメータを持つ関数
- 関数内でパラメータのアドレスを参照している関数
- 実引数と仮引数の数と型が一致していない関数
- アドレスを介して呼び出される関数

（4） SH-1以外でキャッシュを搭載した場合、インライン展開するとキャッシュミスにより速度が向上しない場合があります。

（5） 当機能を使用した場合､関数の呼び出し位置に同じコードが展開されるためプログラムサイズが増加する傾向にあります。実行速度とプログラムサイズのバランスを考えて使用してください。

3.3.2 アセンブラ埋め込みインライン展開の記述方法

■説明

C 言語でサポートしていない CPU 命令を使用したい場合や、C 言語で記述するよりも、アセンブラで記述し性能を向上したい場合があります。このような場合、アセンブリ言語で記述し、C プログラムと結合する方法がありますが、SuperH RISC engine C/C++コンパイラではアセンブラ埋め込みインライン展開機能を使って、C ソースプログラムと混在させることができます。

アセンブリ言語で記述したコードを C 言語の関数と同じ要領で記述し、その関数の前に"#pragma inline_asm"で、記述された関数がアセンブラで書かれた関数であることを宣言しておくと、コンパイラはその関数の呼び出し位置にアセンブラコードを展開します。

なお、関数間のインタフェースは C/C++コンパイラの生成規則に従ってください。C/C++コンパイラはレジスタ R4～R7 にパラメタ値を格納し、R0 にリターン値が格納されているものとしてコードを生成します。SH-2E、SH2A-FPU、SH-4 および SH-4A のとき単精度浮動小数点型のリターン値は FR0、SH2A-FPU、SH-4、SH-4A のとき倍精度浮動小数点型のリターン値は、DR0 に設定してください。

【書式】

```
#pragma inline_asm（<関数名> [,<関数名>・・・] ）
```

■使用例

上位バイトと下位バイトの置換が頻繁にあり、性能上鍵となる場合、バイトスワップの関数をアセンブラで記述し、埋め込みインライン展開します。

C 言語コード

```
#pragma inline_asm(swap)      /* 展開するアセンブラ関数の指定                 */
short swap(short p1)          /* 性能を向上したい関数をアセンブラで記述する     */
{
        EXTU.W     R4,R0      ; clear upper word
        SWAP.B     R0,R2      ; swap with R0 lower word
        CMP/GT     R2,R0      ; if (R2 < R0)
        BT            ?0001      ; then goto ?0001
        NOP                   ;
        MOV           R2,R0      ; return R2
?0001:                        ; local label ラベルはローカルラベルを使用する
                             ;
}

void f (short *x, short *y, int i)
{
       for ( ; i > 0; i--, x++, y++)
          *y = swap(*x);  /* C の関数呼び出しと同様に記述する              */
}
```

アセンブリ言語展開コード（一部）

```
_f:
        MOV.L      R14,@-R15
        MOV        R6,R14
        MOV.L      R13,@-R15
        CMP/PL     R14
        MOV.L      R12,@-R15
        MOV        R5,R13
        MOV        R4,R12
        BT         L224
        MOV.L      L225,R3            ; L221
        JMP        @R3
        NOP
L224:
L222:
        MOV.W      @R12,R4
        BRA        L223
        NOP
L225:
        .DATA.L    L221
L223:
        EXTU.W     R4,R0
```

```
         SWAP.B      R0,R2
         CMP/GT      R2,R0
         BT          ?0001
         NOP
         MOV         R2,R0
?0001:
         .ALIGN      4
         MOV.W       R0,@R13
         ADD         #-1,R14
         ADD         #2,R12
         ADD         #2,R13
         CMP/PL      R14
         BF          L226
         MOV.L       L227+2,R3       ; L222
         JMP         @R3
         NOP
L226:
L221:
         MOV.L       @R15+,R12
         MOV.L       @R15+,R13
         RTS
         MOV.L       @R15+,R14
L227:
         .RES.W      1
         .DATA.L     L222
```

■注意事項

(1) #pragma inline_asmは、関数本体の定義の前に指定してください。

(2) #pragma inline_asmで指定した関数に対しても、外部定義を生成しますので、複数ファイルでインクルードされるファイル中にインライン展開関数の実体を記述する場合は、必ず関数の宣言にstaticを指定してください。

(3) アセンブラ記述の中でラベルを使用する場合、必ずローカルラベルを使用してください。

(4) アセンブラ記述関数でR8からR15（ただし、SH-2Eの場合はFR12～FR15、SH2A-FPU、SH-4、SH-4Aの場合はFR12～FR15、DR12～DR14も含む）のレジスタを使用する場合は、アセンブラ記述関数の先頭と最後でこれらレジスタの退避／回復が必要です。詳細は、「3.15.1 (2) (c) **レジスタに関する規則**」を参照してください。

(5) アセンブラ記述関数の最後にRTSを記述しないでください。

(6) 本機能を使用する際は、オブジェクト形式指定オプション"-code=asmcode"を用いてコンパイルしてください。

(7) 本機能を使った場合、Cソースレベルのデバッグに制限を受けます。

(8) C言語プログラムとアセンブリ言語プログラムとの間で関数呼び出しを行う場合の詳細については、「3.15.1 (2) **関数呼び出しのインタフェース**」を参照してください。

(9) Cプログラムとアセンブリプログラムの結合については「3.15.1 **アセンブリ言語プログラムとの関連**」を参照してください。

3.3.3 インラインアセンブラ関数サンプルプログラム

C 言語では記述すると効率が落ちるようなプログラム、C 言語では記述できないプログラムは普通アセンブラ言語による記述を行いますが、インラインアセンブラ関数を使うことによって、C 言語のような記述を行うことが可能です。

■インラインアセンブラ関数のメリット

C言語の関数として、アセンブラ記述関数を定義できます。
直接アセンブラ命令を埋め込むことができ、一般のアセンブラ関数の呼び出しによる、サブルーチンコール／リターンのオーバヘッドはありません。

■インラインアセンブラ関数のデメリット

コンパイル時に、一度アセンブラソースプログラムを出力する必要があります。
これにより、デバッグ時に、Cのローカル変数が参照できなくなります。
（コンパイル時に、 コンパイルオプション –code=asmcodeを指定し、その後アセンブラを起動する必要があります。Cコンパイル時、アセンブル時の両方に、-debugのオプションをつけることにより、C言語ソースレベルのステップ実行は可能になります。）

■インラインアセンブラの活用ノウハウ

インラインアセンブラ関数は、以下の方法で、ヘッダファイル化して使用することをお勧めします。

- 関数はスタティック宣言する。
- ラベルはローカルラベルを使用する。
- アセンブラがリテラルプールを自動生成する命令は書かない。
- 定義の最後に RTS（リターン）命令は記述しない。

【書式】

```
/* Inline function definition */
/* FILE:  inlasm.h            */
#pragma inline_asm(rev4b)
static  unsigned long rev4b(unsigned long p)
/* 関数はstaticで宣言           */
{
     ; 定義中のコメントは アセンブラの;（セミコロン）を使用する
            SWAP.W    R4,R0
            SWAP.B    R0,R0
     ; 最後にRTS命令は記述しない
}
#pragma inline_asm(ovf)
static  unsigned long ovf()
{
?LABEL001            ; インラインアセンブラ関数内ではローカルラベルを使用する
                     ; ローカルラベル： ‘?’で始まり16文字以内
     MOV    R4,R0
         :
     CMP/EQ    #1,R0
     BT        ?LBABEL001
}
#pragma inline_asm(ovfadd)
#ifdef  NG_INLINEASM
/* 正しくないインラインアセンブラの定義 */
static  unsigned long ovfadd()
{
         :
     MOV.L     #H’f0000000,R0
     ; このような記述はアセンブラがリテラルプールを自動生成する
     ; この場合命令が正しく展開されない場合がある
     ; この関数の範囲外に、リテラルプールが生成されると、アラインが崩れる
}
#else
/* 正しいインラインアセンブラの定義 */
static  unsigned long ovfadd()
{
            :
     MOV.L     #H’f0000000,R0
         ; .POOL制御命令をこのインラインアセンブラ定義内に記述する必要がある
         ; この場合命令は正しく展開される。
     .POOL
         ; この.POOLにより、ここにリテラルプールが展開される
```

```
        ; このプログラムの実際のコード展開イメージは以下のようになる
        ;       :
        ;       MOV.L     Lxxx,R0
        ;       BRA       Lyyy
        ;       NOP
        ; Lxxx .DATA.L    H'f0000000
        ; Lyyy
}
#end
```

以下に、ここで紹介するインラインアセンブラを示します。
なお、Ver.8 以降、64 ビット演算を行う場合、Ver.8 でサポートした long long 型、unsigned long long 型を使用することができます。

- 64 ビット加算
- 64 ビット減算
- 64 ビット乗算
- ビットローテート
- エンディアン変換
- 積和演算
- オーバフローチェック

64 ビットの演算では以下のヘッダを使用しています。

```
"longlong.h"

typedef struct{
      unsigned long H;
      unsigned long L;
   }longlong;
```

(1) 64 ビット加算

C 言語の整数型には、64 ビットデータはないため、C 言語での処理は冗長になります。そこで 64 ビット演算を効率よく使用するためのインラインアセンブラを示します。

(i) 64ビットデータの加算

【書式】 longlong addll(longlong a,longlong b)
【引数】 a：64 ビットデータ
b：64 ビットデータ
【戻り値】 longlong ：64 ビットデータ
【内容】 a と b を加算し、結果を戻す

```
#include <stdio.h>
#include "longlong.h"

#pragma inline_asm(addll)
static longlong addll(longlong a,longlong b)
{
   MOV     @(0,R15),R0        ;戻り値の構造体 c の先頭アドレスをセット
   MOV     @(4,R15),R1        ;第 1 パラメータセット(a.H)
   MOV     @(8,R15),R2        ;                    (a.L)
   MOV     @(12,R15),R3       ;第 2 パラメータセット(b.H)
   MOV     @(16,R15),R4       ;                    (b.L)
   CLRT                       ;T ビットクリア
   ADDC    R4,R2              ;下位 32 ビットの加算,キャリの有無
   ADDC    R3,R1              ;上位 32 ビットの加算+キャリ
   MOV     R1,@(0,R0)         ;戻り値セット(c.H)
   MOV     R2,@(4,R0)         ;            (c.L)
}

void main(void)
{
   longlong a,b,c;
```

```
    a.H=0xefffffff;
    a.L=0xffffffff;
    b.H=0x10000000;
    b.L=0x00000000;
    c=addll(a,b);
    printf("addll = %8X %08X  ¥n",c.H,c.L);
}
```

(ii) 64ビットデータの加算（アドレス指定）

【書式】 void addllp(longlong *pa,longlong *pb,longlong *pc)
【引数】 pa：64 ビットデータのアドレス
pb：64 ビットデータのアドレス
pc：結果格納用変数のアドレス
【戻り値】 なし
【内容】 pa と pb を加算し、結果を pc に返す

```
#include <stdio.h>
#include "longlong.h"

#pragma inline_asm(addllp)
static void addllp(longlong *pa,longlong *pb,longlong *pc)
{
    MOV      @(0,R5),R0   ;(pa->H)を R0 にセット
    MOV      @(4,R5),R1   ;(pa->L)を R1 にセット
    MOV      @(0,R6),R2   ;(pb->H)を R2 にセット
    MOV      @(4,R6),R3   ;(pb->L)を R3 にセット
    CLRT                  ;T ビットクリア
    ADDC     R3,R1        ;下位 32 ビットの加算,キャリの有無
    ADDC     R2,R0        ;上位 32 ビットの加算+キャリ
    MOV      R0,@(0,R4)   ;R0 を(pc->H)にセット
    MOV      R1,@(4,R4)   ;R1 を(pc->L)にセット
}

void main(void)
{
    longlong a,b,c;
    longlong *pa,*pb,*pc;

    b.H=0x10000000;
    b.L=0x00000000;
    c.H=0xefffffff;
    c.L=0xffffffff;

    pa=&a;
    pb=&b;
    pc=&c;
    addllp(pa,pb,pc);
    printf("addllp = %8x %08x ¥n",pa->H,pa->L);
}
```

(iii) 64ビットデータの加算（アドレス指定混在）

【書式】 void addtoll(longlong *pa,longlong b)
【引数】 *pa：64 ビットデータのアドレス
b ：64 ビットデータ
【戻り値】 なし
【内容】 pa の指定するデータと b を加算し、結果を pa に返す

```
#include <stdio.h>
#include "longlong.h"

#pragma inline_asm(addtoll)
static void addtoll(longlong *pa,longlong b)
{
    MOV      @(0,R4),R0   ;(pa->H)を R0 にセット
    MOV      @(4,R4),R1   ;(pa->L)を R1 にセット
    MOV      @(0,R15),R2  ;(b.H)を R2 にセット
    MOV      @(4,R15),R3  ;(b.L)を R3 にセット
    CLRT                  ;T ビットクリア
    ADDC     R3,R1        ;(pa->L)+(b.L)、キャリの有無
    ADDC     R2,R0        ;(pa->H)+(b.H)+キャリ
    MOV   R0,@(0,R4)      ;R0 を(pa->H)にセット
    MOV   R1,@(4,R4)      ;R1 を(pa->L)にセット
```

```
}

void main(void)
{
    longlong *pa,b,c;

    b.H=0x10000000;
    b.L=0x00000000;
    c.H=0xefffffff;
    c.L=0xffffffff;

    pa=&c;
    addtoll(pa,b);
    printf("addtoll = %8x %08x ¥n",pa->H,pa->L);
}
```

(iv) 64ビットデータと32ビットデータの加算

【書式】 void addtoll32(longlong *pa,long b)
【引数】 *pa：64 ビットデータのアドレス
 b ：32 ビットデータ
【戻り値】 なし
【内容】 pa の指定するデータと b を加算し、結果を pa の指すアドレスに戻す

```
#include <stdio.h>
#include "longlong.h"

#pragma inline_asm(addtoll32)
static void addtoll32(longlong *pa,long b)
{
    MOV      @(0,R4),R0   ;(pa->H)を R0 にセット
    MOV      @(4,R4),R1   ;(pa->L)を R1 にセット
    CLRT                  ;T ビットクリア
    ADDC     R5,R1        ;(pa->L)+b,キャリの有無
    MOVT     R3           ;キャリを R3 にセット
    ADD      R3,R0        ;(pa->H)+キャリ
    MOV      R0,@(0,R4)   ;R0 を(pa->H)にセット
    MOV      R1,@(4,R4)   ;R1 を(pa->L)にセット
}

void main(void)
{
    longlong *pa,c;
    long b;
    b=0x00000001;
    c.H=0xefffffff;
    c.L=0xffffffff;

    pa=&c;
    addtoll32(pa,b);
    printf("addlltoll32 = %8x %08x  ¥n",pa->H,pa->L);
}
```

(2) 64 ビット減算

(i) 64ビットデータの減算

【書式】 longlong subll(longlong a,longlong b)
【引数】 a：64 ビットデータ
 b：64 ビットデータ
【戻り値】 longlong 型 ：64 ビットデータ
【内容】 a から b を減算し結果を戻す

```
#include <stdio.h>
#include "longlong.h"

#pragma inline_asm(subll)
static longlong subll(longlong a,longlong b)
{
    MOV    @(0,R15),R0      ;戻り値のアドレスを R0 にセット
    MOV    @(4,R15),R1      ;(a.H)を R1 にセット
    MOV    @(8,R15),R2      ;(a.L)を R2 にセット
    MOV    @(12,R15),R3     ;(b.H)を R3 にセット
    MOV    @(16,R15),R4     ;(b.L)を R4 にセット
    CLRT                    ;T ビットクリア
```


RENESAS

```
    SUBC    R4,R2             ;(a.L)-(b.L)、ボローの有無
    SUBC    R3,R1             ;(a.H)-(b.H)-ボロー
    MOV       R1,@(0,R0)      ;R1を(c.H)にセット
    MOV       R2,@(4,R0)      ;R2を(c.L)にセット
}

void main(void)
{
    longlong a,b,c;

    a.H=0xffffffff;
    a.L=0xffffffff;
    b.H=0xffffffff;
    b.L=0xffffffff;
    c=subll(a,b);
    printf("subll = %x %08x  ¥n",c.H,c.L);
}
```

(ii) 64ビットデータの減算（アドレス指定混在）

【書式】　void subtoll(longlong *pa,longlong b)
【引数】　*pa：64ビットデータのアドレス
　　　　　 b ：64ビットデータ
【戻り値】　なし
【内容】　paの指すデータからbを減算し結果をpaに戻す

```
#include <stdio.h>
#include "longlong.h"

#pragma inline_asm(subtoll)
static void subtoll(longlong *pa,longlong b)
{
    MOV       @(0,R4),R0      ;(pa->H)をR0にセット
    MOV       @(4,R4),R1      ;(pa->L)をR1にセット
    MOV       @(0,R15),R2     ;(b.H)をR2にセット
    MOV       @(4,R15),R3     ;(b.L)をR3にセット
    CLRT                      ;Tビットクリア
    SUBC      R3,R1           ;(a.L)-(b.L)、ボローの有無
    SUBC      R2,R0           ;(a.H)-(b.H)-ボロー
    MOV       R0,@(0,R4)      ;R0を(pa->H)にセット
    MOV       R1,@(4,R4)      ;R1を(pa->L)にセット
}

void main(void)
{
    longlong *pa,b,c;

    b.H=0xffffffff;
    b.L=0xffffffff;
    c.H=0xffffffff;
    c.L=0xffffffff;
    pa=&c;
    subtoll(pa,b);
    printf("addtoll = %8x %08x ¥n",pa->H,pa->L);
}
```

(iii) 64ビットデータと32ビットデータの減算

【書式】　void subtoll32(longlong *pa,long b)
【引数】　*pa:64ビットデータのアドレス
　　　　　 b :32ビットデータ
【戻り値】　なし
【内容】　paの指定するデータからbを減算し結果をpaに返す

```
#include <stdio.h>
#include "longlong.h"

#pragma inline_asm(subtoll32)
static void subtoll32(longlong *pa,long b)
{
    MOV     @(0,R4),R0  ;(pa->H)をR0にセット
    MOV     @(4,R4),R1  ;(pa->L)をR1にセット
    CLRT                ;Tビットクリア
    SUBC    R5,R1       ;(pa->L)-b、ボローの有無
    MOVT      R3           ;ボローをR3にセット
```

```
    SUB       R3,R0         ;(pa->H)-ボロー
    MOV       R0,@(0,R4)    ;R0を(pa->H)にセット
    MOV       R1,@(4,R4)    ;R1を(pa->L)にセット
}

void main(void)
{
    longlong *pa,c;
    unsigned long b;
    pa=&c;

    c.H=0xf0000000;
    c.L=0x00000000;

    b=0x00000001;

    subtoll32(pa,b);
    printf("subll = %8x %08x  ¥n",pa->H,pa->L);
}
```

(3) 64 ビット乗算

(i) 64ビットデータの乗算

【書式】 longlong mulll(longlong a,longlong b)
【引数】 a ：64 ビットデータ
　　　　 b ：64 ビットデータ
【戻り値】 longlong ：64 ビットデータ
【内容】 a と b を乗算し、結果を戻す

```
#include <stdio.h>
#include "longlong.h"

#pragma inline_asm(mulll)
static longlong mulll(longlong a,longlong b)
{
    MOV       @(4,R15),R0     ;(a.H)を R0 にセット
    MOV       @(8,R15),R1     ;(a.L)を R1 にセット
    MOV       @(12,R15),R2    ;(b.H)を R2 にセット
    MOV       @(16,R15),R3    ;(b.L)を R3 にセット
    MUL.L      R0,R3          ;(a.H)*(b.L)
    STS       MACL,R0         ;結果を代入（下位 32 ビット）
    MUL.L     R2,R1           ;(a.L)*(b.H)
    STS       MACL,R2         ;結果を代入（下位 32 ビット）
    ADD       R2,R0           ;
    DMULU     R1,R3           ;(a.L)*(b.L)
    STS       MACH,R1         ;結果を代入（上位 32 ビット）
    STS       MACL,R3         ;結果を代入（下位 32 ビット）
    ADD       R1,R0           ;
    MOV       @(0,R15),R4     ;
    MOV       R0,@(0,R4)      ;R0 を(c.H)にセット
    MOV       R3,@(4,R4)      ;R3 を(c.L)にセット
}

void main(void)
{
    longlong a,b,c;
    a.H=0x7fffffff;
    a.L=0xffffffff;
    b.H=0x00000000;
    b.L=0x00000002;

    c=mulll(a,b);
    printf("mulll = %8x %08x  ¥n",c.H,c.L);
}
```

(ii) 64ビットデータの乗算（アドレス指定）

【書式】 void multoll(longlong *pa,longlong b)
【引数】 pa ：64ビットデータアドレス
　　　　 b ：64ビットデータ
【戻り値】 なし
【内容】 paアドレスの指す値とbを乗算し、結果をpaの指すアドレスに戻す

```
#include <stdio.h>
```

```
#include "longlong.h"

#pragma inline_asm(multoll)
static void multoll(longlong *pa,longlong b)
{
    MOV     @(0,R4),R0      ;(pa->H)をR0にセット
    MOV     @(4,R4),R5      ;(pa->L)をR5にセット
    MOV     @(4,R15),R1     ;(b.L)をR1にセット
    MUL     R0,R1           ;(pa->H)*(b.L)
    STS     MACL,R3         ;
    DMULU   R5,R1           ;(pa->L)*(b.L)
    STS     MACH,R0         ;結果を代入（上位32ビット）
    STS     MACL,R1         ;結果を代入（下位32ビット）
    ADD     R3,R0           ;
    MOV     R0,@(0,R4)      ;R0を(pa->H)にセット
    MOV     R1,@(4,R4)      ;R1を(pa->L)にセット

}

void main(void)
{
    longlong *pa,b,c;

    c.H=0x0000ffff;
    c.L=0xffff0000;
    b.H=0x00000000;
    b.L=0x00010000;

    pa=&c;
    multoll(pa,b);
    printf("multoll = %8x %08x ¥n",pa->H,pa->L);
}
```

(iii) 64ビットデータと符号なし32ビットデータの乗算

【書式】 void multoll32(longlong *pa,unsigned long b)
【引数】 *pa：64ビットデータのアドレス
b ：符号なし32ビットデータ
【戻り値】 なし
【内容】 paの指す値とbを乗算し、結果をpaの指すアドレスに戻す

```
#include <stdio.h>
#include "longlong.h"

#pragma inline_asm(multoll32)
static void multoll32(longlong *pa,unsigned long b)
{
    MOV    @(0,R4),R0      ;(pa->H)をR0にセット
    MOV    @(4,R4),R1      ;(pa->L)をR1にセット
    ADDC   R5,R1           ;(pa->L)+b,キャリの有無
    MOVT   R3              ;キャリをR3にセット
    ADD    R3,R0           ;(pa->H)+キャリ
    MOV    R0,@(0,R4)      ;R0を(pa->H)にセット
    MOV    R1,@(4,R4)      ;R1を(pa->L)にセット

void main(void)
{
    longlong *pa,c;
    unsigned long b;

    b=0xffffff00;
    c.H=0x00000000;
    c.L=0x00000100;
    pa=&c;
    multoll32(pa,b);
    printf("mulltoll32 = %8x %08x ¥n",pa->H,pa->L);
}
```

(iv) 符号なし32ビットデータの乗算

【書式】 longlong mul64(unsigned long a,unsigned long b)
【引数】 a: 符号なし32ビットデータ
b: 符号なし32ビットデータ

【戻り値】 longlong: 64ビットデータ
【内容】 aとbを乗算し、結果を戻す

```
#include <stdio.h>
#include "longlong.h"

#pragma inline_asm(mul64)
static longlong mul64(unsigned long a,unsigned long b)
{
       MOV      @(0,R15),R0     ;cのアドレスをR0にセット
       DMULU    R4,R5           ;a*b
       STS      MACH,R1         ;結果を代入（上位32ビット）
       MOV      R1,@(0,R0)      ;R1を(c.H)にセット
       STS      MACL,R2         ;結果を代入（下位32ビット）
       MOV      R2,@(4,R0)      ;R2を(c.L)にセット
}

void main(void)
{
    longlong c;
    unsigned long a,b;

a=0xffffffff;
    b=0x10000000;
    c=mul64(a,b);
    printf("mul64 = %8x %08x  ¥n",c.H,c.L);
}
```

(v) 符号付き32ビットデータの乗算

【書式】 longlong mul64s(signed long a,signed long b)
【引数】 a: 32ビットデータ
b: 32ビットデータ
【戻り値】 longlong: 64ビットデータ
【内容】 aとbを乗算し、結果を戻す

```
#include <stdio.h>
#include "longlong.h"

#pragma inline_asm(mul64s)
static longlong mul64s(signed long a,signed long b)
{
       MOV      @(0,R15),R0     ;cのアドレスをR0にセット
       DMULS    R4,R5           ;a*b（符号付き）
       STS      MACH,R1         ;結果を代入（上位32ビット）
       MOV      R1,@(0,R0)      ;R1を(c.H)にセット
       STS      MACL,R2         ;結果を代入（下位32ビット）
       MOV      R2,@(4,R0)      ;R2を(c.L)にセット
}

void main(void)
{
    longlong c;
    signed long a,b;
    a=-1;
    b=1;
    c=mul64s(a,b);
    printf("mul64s = %8x %08x ¥n",c.H,c.L);
}
```

(4) ビットローテート

(i) 8ビットデータの左1ビットローテート

【書式】 short rot8l(unsigned long a)
【引数】 a：符号なし 8 ビットデータ
【戻り値】 short ：8 ビットデータ
【内容】 a を左に 1 ビット、ローテーションして値を戻す

```
#include <stdio.h>

#pragma inline_asm(rot8l)
unsigned char rot8l(unsigned char a)
{

    ROTL    R4          ;左に 1 ビットシフト
```

```
    MOV     R4,R0     ;
    SHLR8   R0        ;右に 8 ビットシフト
    OR      R4,R0     ;
}

void main(void)
{

    unsigned char a;

    a=0x12;
    a=rot8l(a);
    printf(" rot8l %x ¥n",a);

}
```

(ii) 8ビットデータの左nビットローテート

【書式】 short rot8ln(unsigned char a,int n)
【引数】 a：符号なし 8 ビットデータ
n：シフト数
【戻り値】 short ：8 ビットデータ
【内容】 a を左に n ビット、ローテーションして値を戻す

```
#include <stdio.h>

#pragma inline_asm(rot8ln)
unsigned char rot8ln(unsigned char a,int n)
{

    MOV    #0,R1     ;カウンタ用レジスタをセット
?LOOP:
    ROTL    R4       ;左に 1 ビットシフト
    MOV     R4,R2    ;
    SHLR8   R2       ;右に 8 ビットシフト
    ADD     #1,R1    ;カウンタ+1
    CMP/EQ R1,R5     ;R1==R5 ならば T=1
    BF      ?LOOP    ;T!=1 ならば分岐
    OR      R2,R4    ;分岐の前に実行
    MOV     R4,R0    ;戻り値をセット
}

void main(void)
{

    unsigned char a,b;
    int n;

    a=0x12;
    n=4;
    b=rot8ln(a,n);
    printf(" b: %x ¥n",b);
}
```

(iii) 8ビットデータの右1ビットローテート

【書式】 short rot8r(unsigned char a)
【引数】 a: 符号なし 8 ビットデータ
【戻り値】 short: 8 ビットデータ
【内容】 a を右に 1 ビット、ローテーションして値を戻す

```
#pragma inline_asm(rot8r)
unsigned char rot8r(unsigned char a)
{
    ROTR      R4       ;右に 1 ビットシフト
    MOV       R4,R0    ;
    SHLR16    R4       ;右に 16 ビットシフト
    SHLR8     R4       ;右に 8 ビットシフト
    OR        R4,R0    ;
}

void main(void)
{
    unsigned char a;
```

```
    a=0x12;
    a=rot8r(a);
    printf(" rot8r %x ¥n",a);

}
```

(iv) 8ビットデータの右nビットローテート

【書式】 short rot8rn(unsigned char a,int n)
【引数】 a：符号なし 8 ビットデータ
　　　　 n：シフト数
【戻り値】 short ：8 ビットデータ
【内容】 a を右に n ビット、ローテーションして値を戻す

```
#include <stdio.h>

#pragma inline_asm(rot8rn)
unsigned char rot8rn(unsigned char a,int n)
{

    MOV       #0,R1      ;カウンタ用レジスタをセット
?LOOP:
    ROTR      R4         ;右に 1 ビットシフト
    MOV       R4,R2      ;
    SHLR16    R2         ;右に 16 ビットシフト
    SHLR8     R2         ;右に 8 ビットシフト
    ADD       #1,R1      ;カウンタ+1
    CMP/EQ    R1,R5      ;R1==R5 ならば T=1
    BF        ?LOOP      ;T!=1 ならば分岐
    OR        R2,R4      ;分岐する前に実行
    MOV       R4,R0      ;戻り値をセット
}

void main(void)
{

    unsigned char a,b;
    int n;

    a=0x12;
    n=4;
    b=rot8rn(a,n);
    printf(" rot8rn %x ¥n",b);

}
```

(v) 16ビットデータの左1ビットローテート

【書式】 short rot16l(unsigned short a)
【引数】 a: 符号なし 16 ビットデータアドレス
【戻り値】 short: 16 ビットデータ
【内容】 a を左に 1 ビット、ローテーションして値を戻す

```
#pragma inline_asm(rot16l)
unsigned short rot16l(unsigned short a)
{
       ROTL      R4       ;左に 1 ビットシフト
       MOV       R4,R0    ;
       SHLR16    R0       ;右に 16 ビットシフト
       OR        R4,R0    ;
}

void main(void)
{
    unsigned short a,b;
    a=0x1234;
    b=rot16l(a);
    printf(" rot16l = %x ¥n",b);
}
```

(vi) 16ビットデータの左nビットローテート

【書式】 short rot16ln(unsigned short a,int n)
【引数】 a：符号なし 16 ビットデータアドレス

```
            n：シフト数
【戻り値】  short ：16 ビットデータ
【内容】    a を左に n ビット、ローテーションして値を戻す
#include <stdio.h>

#pragma inline_asm(rot16ln)
unsigned short rot16ln(unsigned short a,int n)
{
    MOV     #0,R1       ;カウンタ用レジスタをセット
?LOOP:
    ROTL    R4          ;左に 1 ビットシフト
    MOV     R4,R2       ;
    SHLR16  R2          ;右に 16 ビットシフト
    ADD     #1,R1       ;カウンタ+1
    CMP/EQ  R1,R5       ;R1==R5 ならば T=1
    BF      ?LOOP       ;T!=1 ならば分岐
    OR      R2,R4       ;
    MOV     R4,R0       ;戻り値をセット
}

void main(void)
{

    unsigned short a,b;
    int n;

    a=0x1234;
    n=8;
    b=rot16ln(a,n);
    printf("rot16ln = %x ¥n",b);

}
```

(vii) 16ビットデータの右1ビットローテート

```
【書式】    short rot16r(unsigned short a)
【引数】    a：符号なし 16 ビットデータアドレス
【戻り値】  short ：16 ビットデータ
【内容】    a を右に 1 ビット、ローテーションして値を戻す

#include <stdio.h>

#pragma inline_asm(rot16r)
unsigned short rot16r(unsigned short a)
{
    ROTR      R4        ;右に 1 ビットシフト
    MOV       R4,R0     ;
    SHLR16    R0        ;右に 16 ビットシフト
    OR        R4,R0     ;

}

void main(void)
{

    unsigned short a,b;
    a=0x1234;

    b=rot16r(a);

    printf("rot16r = %x ¥n",b);

}
```

(viii) 16ビットデータの右nビットローテート

```
【書式】    short rot16rn(unsigned short a,int n)
【引数】       a：符号なし 16 ビットデータアドレス
               n：シフト数
【戻り値】  short ：16 ビットデータ
【内容】       a を右に n ビット、ローテーションして値を戻す

#include <stdio.h>
```

```
#pragma inline_asm(rot16rn)
unsigned short rot16rn(unsigned short a,int n)
{
   MOV       #0,R1     ;カウンタ用レジスタをセット
?LOOP:
   ROTR      R4        ;右に 1 ビットシフト
   MOV       R4,R2     ;
   SHLR16  R2          ;右に 16 ビットシフト
   ADD       #1,R1     ;カウンタ+1
   CMP/EQ  R1,R5       ;R1==R5 ならば T=1
   BF        ?LOOP     ;T!=1 ならば分岐
   OR        R2,R4     ;
   MOV       R4,R0     ;戻り値をセット

}

void main(void)
{

   unsigned short a,b;
   int n;

   a=0x1234;
   n=8;
   b=rot16rn(a,n);
   printf("rot16rn %x ¥n",b);

}
```

(ix) 32ビットデータの左1ビットローテート

【書式】 short rot32l(unsigned long a)
【引数】 a：符号なし 32 ビットデータアドレス
【戻り値】 short ：32 ビットデータ
【内容】 a を左に 1 ビット、ローテーションして値を戻す

```
#include <stdio.h>

#pragma inline_asm(rot32l)
unsigned long rot32l(unsigned long a)
{
   ROTL    R4        ;左に 1 ビットシフト
   MOV       R4,R0     ;戻り値をセット
}

void main(void)
{
   unsigned long a;

   a=0x12345678;

   a=rot32l(a);

   printf(" rot32l %8x ¥n",a);

}
```

(x) 32ビットデータの左nビットローテート

【書式】 short rot32ln(unsigned long a,int b)
【引数】 a：符号なし 32 ビットデータアドレス
n：シフト数
【戻り値】 short ：32 ビットデータ
【内容】 a を左に n ビット、ローテーションして値を戻す

```
#include <stdio.h>

#pragma inline_asm(rot32ln)
unsigned long rot32ln(unsigned long a,int b)
{
   MOV       #0,R1     ;カウンタ用レジスタをセット
?LOOP:
   ROTL      R4        ;右に 1 ビットシフト
   ADD       #1,R1     ;カウンタ+1
```

```
    CMP/EQ    R1,R5     ;R1==R5 ならば T=1
    BF        ?LOOP     ;T!=1 ならば分岐
    MOV       R4,R0     ;戻り値をセット
}

void main(void)
{
    unsigned long a;
    int b;

    a=0x12345678;
    b=16;

    a=rot32ln(a,b);
    printf(" rot32ln %8x ¥n",a);

}
```

(xi) 32ビットデータの右1ビットローテート

【書式】 short rot32r(unsigned long a)
【引数】 a：符号なし 32 ビットデータアドレス
【戻り値】 short ：32 ビットデータ
【内容】 a を右に 1 ビット、ローテーションして値を戻す

```
#include <stdio.h>

#pragma inline_asm(rot32r)
unsigned long rot32r(unsigned long a)
{
    ROTR   R4        ;右に 1 ビットシフト
    MOV    R4,R0     ;戻り値をセット
}


void main(void)
{
    unsigned long a,b;

    a=0x12345678;

    b=rot32r(a);

    printf(" rot32r %8x ¥n",b);

}
```

(xii) 32ビットデータの右nビットローテート

【書式】 short rot32rn(unsigned long a,int b)
【引数】 a：符号なし 32 ビットデータアドレス
　　　　 n：シフト数
【戻り値】 short ：32 ビットデータ
【内容】 a を右に n ビット、ローテーションして値を戻す

```
#include <stdio.h>

#pragma inline_asm(rot32rn)
unsigned long rot32rn(unsigned long a,int b)
{
    MOV       #0,R1     ;カウンタ用レジスタをセット
?LOOP:
    ROTR      R4        ;右に 1 ビットシフト
    ADD       #1,R1     ;カウンタ+1
    CMP/EQ    R1,R5     ;R1==R5 ならば T=1
    BF        ?LOOP     ;T!=1 ならば分岐
    MOV       R4,R0     ;戻り値をセット
}

void main(void)
{
    unsigned long a;
    int b;

    a=0x12345678;
```

```
    b=16;

    a=rot32rn(a,b);
    printf("rot32rn %8x ¥n",b);

}
```

(5) エンディアン変換

(i) 上位16ビット下位16ビットを入れ替える

【書式】 unsigned long swap(unsigned long a)
【引数】 a：符号なし 32 ビットデータ
【戻り値】 unsigned long ：符号なし 32 ビットデータ
【内容】 a の上位 16 ビットと下位 16 ビットを入れ替える

```
#include <stdio.h>

#pragma inline_asm(swap)
static unsigned long swap(unsigned long a)
{

    SWAP.W  R4,R0   ;R4 の上位 16 ビットと下位 16 ビットを入れ替える

}

void main(void)
{

    unsigned long a,b;
    a=0xaaaabbbb;

    b=swap(a);
    printf("b: %8x ¥n",b);

}
```

(ii) 上位16ビット下位16ビットの対称的な入れ替え

【書式】 unsigned long swapbit(unsigned long a)
【引数】 a：符号なし 32 ビットデータ
【戻り値】 unsigned long ：符号なし 32 ビットデータ
【内容】 上位 16 ビットと下位 16 ビットを 1 つずつ交換する

```
32bit  →   1bit  ,  1bit  →  32bit
31bit  →   2bit  ,  2bit  →  31bit
                 :
18bit  →  15bit  , 15bit  →  18bit
17bit  →  16bit  , 16bit  →  17bit
```

```
#include <stdio.h>

#pragma inline_asm(swapbit)
static unsigned long swapbit(unsigned long a)
{
    MOV       #0,R0     ;カウンタ用レジスタのセット
?LOOP:
    ROTCL     R4        ;左にローテート
    ROTCR     R1        ;右にローテート
    ADD       #1,R0     ;カウンタ+1
    CMP/EQ    #32,R0    ;32==R0 ならば T=1
    BF        ?LOOP     ;T!=1 ならば分岐する
    NOP                 ;
    MOV       R1,R0     ;戻り値をセットする

}


void main(void)
{

    unsigned long a,b;
```

```
        a=0x1234;
        b=swapbit(a);

        printf("b: %8x ¥n",b);

    }
```

(iii) エンディアン変換

【書式】 unsigned long swapbyte(unsigned long a)
【引数】 a：符号なし 32 ビットデータ
【戻り値】 unsigned long ：符号なし 32 ビットデータ
【内容】 a をエンディアン変換し、結果を戻す

```
#include <stdio.h>

#pragma inline_asm(swapbyte)
static unsigned long swapbyte(unsigned long a)
{

    SWAP.B  R4,R4   ;bit0～bit7 の 8bit と bit8～bit15 の 8bit を入れ替える
    SWAP.W  R4,R4   ;上位 16bit と下位 16bit を入れ替える
    SWAP.B  R4,R0   ;bit0～bit7 の 8bit と bit8～bit15 の 8bit を入れ替える

}

void main(void)
{

    unsigned long a,b;
    a=0xaabbccdd;

    b=swapbyte(a);
    printf("b: %8x ¥n",b);

}
```

(6) 積和演算

(i) 32ビットデータの配列の積和演算

【書式】 long macl32h(long *pa,long *pb,int size)
【引数】 *pa ：32 ビットデータ配列の先頭アドレス
*pb ：32 ビットデータ配列の先頭アドレス
size：配列の個数
【戻り値】 long ：32 ビットデータ
【内容】 データ配列*pa と*pb を積和演算する。
演算結果の 64 ビットデータの上位 32 ビットを戻す

```
#include <stdio.h>

#pragma inline_asm(macl32h)
static long macl32h(long *pa,long *pb,int size)
{
    MOV       #0,R1         ;カウンタ用レジスタをセット
    CLRMAC                  ;MAC の値を初期化
?LOOP:
    MAC.L     @R4+,@R5+     ;積和計算
    ADD       #1,R1         ;カウンタ+1
    CMP/EQ    R1,R6         ;R1==R6 ならば T=1
    BF        ?LOOP         ;T!=1 ならば分岐
    NOP                     ;
    STS       MACH,R0       ;結果を代入
}
void main(void)
{
    int size=3;
    long c;
    long pa[3]={0x0000f000,0x000f0000,0x00f00000};
    long pb[3]={0x00000100,0x00001000,0x00010000};

    c=macl32h(pa,pb,size);
    printf("macl32h = %8x ¥n",c);

}
```

(ii) 符号なしビットデータ配列の積和演算

【書式】 longlong macl64(long *pa,long *pb,int size)
【引数】 *pa ：32 ビットデータ配列の先頭アドレス
*pb ：32 ビットデータ配列の先頭アドレス
size：配列の個数
【戻り値】 longlong ：64 ビットデータ
【内容】 32 ビットデータ配列*pa と*pb を積和演算する。
演算結果を戻す

```
#include <stdio.h>
#include "longlong.h"

#pragma inline_asm(macl64)
static longlong macl64(long *pa,long *pb,int size)
{
    MOV       #0,R0              ;カウンタ用レジスタセット
    MOV       @(0,R15),R1        ;第 1 パラメータのアドレスをセット
    CLRMAC                       ;MAC レジスタを初期化
?LOOP:
    MAC.L   @R4+,@R5+            ;積和演算、計算後アドレスを進める
    ADD       #1,R0              ;カウンタ+1
    CMP/EQ  R0,R6                ;R0==R6 ならば T=1
    BF        ?LOOP              ;T!=1 ならば分岐する
    NOP                          ;
    STS       MACH,R2            ;積和演算の上位 32bit の結果をセット
    MOV       R2,@(0,R1)         ;
    STS       MACL,R3            ;積和演算の下位 32bit の結果をセット
    MOV       R3,@(4,R1)         ;
}

void main(void)
{
    longlong c;
    int size=3;
    long *pa,*pb;

    long pa[3]={0x0000f000,0x000f0000,0x00f00000};
    long pb[3]={0x00000100,0x00001000,0x00010000};

    c=macl64(pa,pb,size);
    printf("macl64 = %8X %08X  ¥n",c.H,c.L);

}
```

(7) オーバフローチェック

(i) 32 ビットデータ加算のオーバフローチェック

【書式】 long addovf(long a,long b)
【引数】 a：加算用 32 ビットデータ
b：加算用 32 ビットデータ
【戻り値】 long：32 ビットデータ
【内容】 a と b を加算し、結果を返す
結果がオーバフローしていれば最大値（7FFFFFFF）を返す
結果がアンダフローしていれば最小値（80000000）を返す
ただし、判定は符号ビットの変化で判定

```
#include <stdio.h>
#pragma inline_asm(addovf)
static long addovf(long a,long b)
{
    ADDV      R4,R5         ;符号付き加算、符号ビットの変化で、T ビットをセット
    BF        ?RETURN       ;T==0 で分岐
    MOV       #0,R1         ;
    CMP/GT    R4,R1         ;R1>R4 であれば T ビットをセット
    BF        ?OVER         ;T==0 で分岐
    NOP                     ;
    MOV.L     ?DATA+4,R5    ;R5 に(H'7FFFFFFF)をセット
    BRA       ?RETURN       ;RETURN に分岐
    NOP                     ;

?OVER:
    MOV.L     ?DATA,R5      ;R5 に(H'80000000)をセット
```

```
?RETURN:
    MOV       R5,R0         ;R5 を R0 にセット
    BRA       ?OWARI        ;OWARI に分岐
    NOP                     ;

?DATA:
    .ALIGN    4
    .RES.L    1
    .DATA.L   H'7FFFFFFF
    .DATA.L   H'80000000

?OWARI:

}

void main(void)
{
    long a,b,c;
    a=0x3000000;
    b=0x2000000;

    c=addovf(a,b);
    printf("c: %x ¥n",c);

}
```

3.4 レジスタ指定

外部変数へのアクセスが多いモジュールの実行速度を向上したい場合があります。このような場合、頻繁にアクセスするデータを、グローバルベースレジスタ（GBR）を用い相対アドレシングモードで参照する GBR ベース変数の指定機能を使用します。GBR 参照にした変数は$G0,$G1 セクションに割り付けられ GBR に格納されている$G0 セクションの先頭アドレスからのオフセットで参照されます。そのため、アドレスをロードして参照するコードよりもコンパクトで高速なコードに展開されるので、実行速度及び ROM 効率の向上に役立ちます。

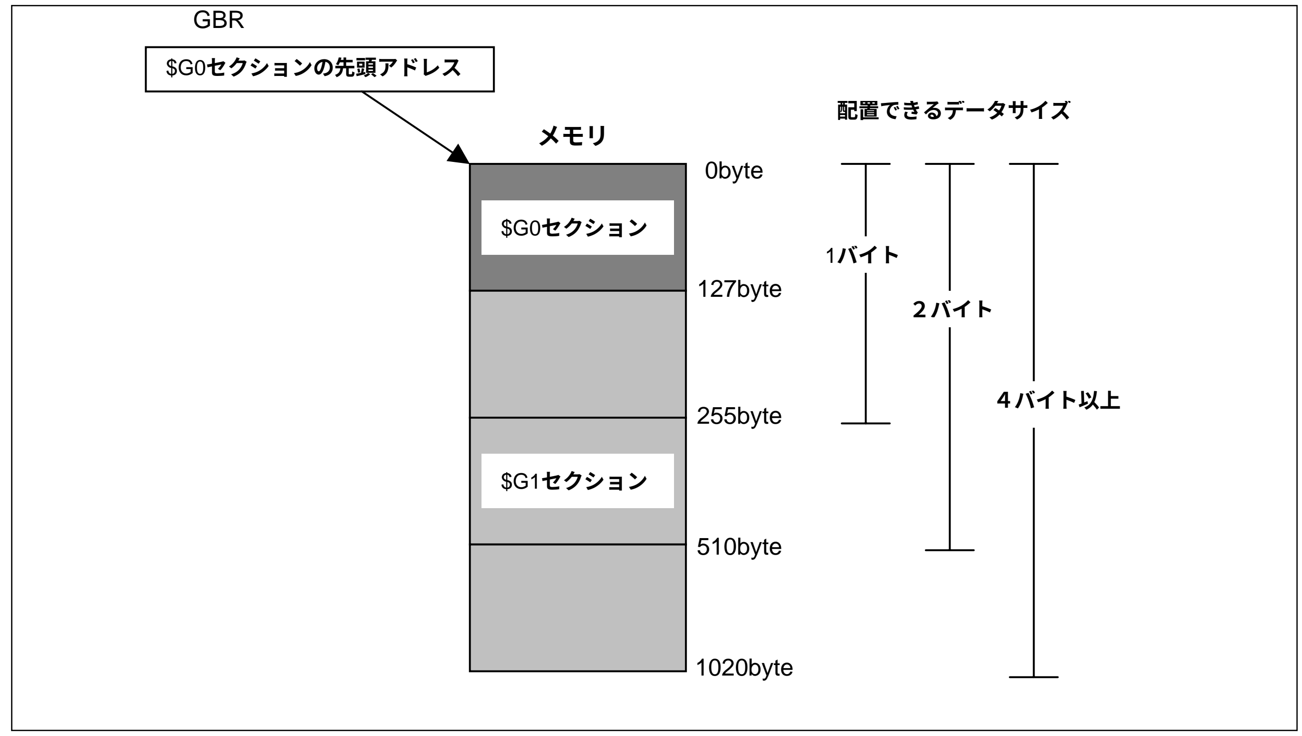


図 3.2 GBR ベース変数参照

3.4.1 GBR ベース変数の指定

■説明

外部変数を GBR ベースの参照にするには、プリプロセッサ制御文で行います。

"#pragma gbr_base"は、変数が GBR の指すアドレスからオフセット 0～127 バイトにあることを指定します。ここで指定した変数は、セクション"$G0"に割り付けられます。

"#pragma gbr_base1"は、変数が GBR の指すアドレスからのオフセットが、char 型、 unsigned char 型の場合は最大 255 バイト、short 型、unsigned short 型の場合は最大 510 バイト、int 型、unsigned int 型、long 型、unsigned long 型、float 型、double 型の場合は最大 1020 バイトであることを指定します。ここで指定した変数は、セクション"$G1"に割り付けられます。

【書式】

```
#pragma gbr_base （<変数名> [,<変数名>・・・] ）
#pragma gbr_base1 （<変数名> [,<変数名>・・・] ）
```

■使用例

C 言語コード

```
#pragma gbr_base(a1,b1,c1)
#pragma gbr_base1(a2,b2,c2)
char a1,a2;
short b1,b2;
long c1,c2;
void f()
{
      a1 = a2;
      b1 = b2;
      c1 = c2;
}
```

アセンブリ言語展開コード

```
_f:
        MOV.B      @(_a2-(STARTOF $G0),GBR),R0
        MOV.B      R0,@(_a1-(STARTOF $G0),GBR)
        MOV.W      @(_b2-(STARTOF $G0),GBR),R0
        MOV.W      R0,@(_b1-(STARTOF $G0),GBR)
        MOV.L      @(_c2-(STARTOF $G0),GBR),R0
        RTS
        MOV.L      R0,@(_c1-(STARTOF $G0),GBR)
```

GBR ベース変数を使用するには、あらかじめ GBR レジスタに$G0 セクションの先頭アドレスを設定しておくことが必要です。以下にその例を示します。

初期化プログラム（アセンブリ言語部分）

```
              :
           .SECTION  $G0,DATA,ALIGN=4
              :
__G_BGN:   .DATA.L   (STARTOF $G0)      ;$G0 セクションの先頭
              :                         ; アドレスを設定
           .EXPORT   __G_BGN
              :
            .END
```

初期化プログラム（C 言語部分）

```
#include <machine.h>
extern int *_G_BGN;
void _INITSCT()           /* main 関数の前に実行される関数            */
{
       :
   set_gbr(_G_BGN);       /* GBR レジスタに$G0 セクションの先頭を設定  */
       :
}
```

■注意事項

本機能を使用する場合は、次の注意事項に従ってください。

（1） プログラム実行開始時に、GBRを$G0セクションの先頭アドレスに設定してください。

（2） $G1セクションは、リンク時に$G0セクションの直後に必ず配置してください。また、#pragma gbr_base1のみ使用する場合でも、必ず$G0セクションは作成してください。

（3） セクション$G0のリンク後の合計サイズが128バイトを超えた場合、または、セクション$G1内に、"#pragma gbr_base1"の説明の各データ型に示した以上のオフセットを持つデータがある場合、動作を保証しません。

（4） （2），（3）の制約を満たしていないと正しく動作しませんので、リンク時に出力されるマップリストで確認してください。

（5） 特に頻繁にアクセスされるデータ、ビット演算が行われるデータは、なるべく$G0セクションに割り付けてください。$G1セクションより$G0セクションに割り付けられたデータにアクセスするほうが実行速度、サイズともに効率の良いオブジェクトが生成されます。

（6） "#pragma gbr_base"および"#pragma gbr_base1"で指定された変数は、変数宣言された順番に各セクションに割り付けられます。異なるサイズの変数が交互に宣言されるとデータサイズが増えますので注意してください。

（7） gbr=autoを指定した場合、#pragma gbr_base、#pragma gbr_base1指定は無効になります。(Ver.7以降)

3.4.2 グローバル変数のレジスタ割り付け

■説明

〈変数名〉で指定されたグローバル変数に、〈レジスタ名〉で指定したレジスタを割り付けます。

【書式】

```
#pragma global_register (<変数名>=<レジスタ名>,...)
```

■使用例

C 言語コード

```
#pragma global_register(x=R13,y=R14)

int    x;
char   *y;

func1()
{
   x++;
}

func2()
{
   *y=0;
}

func(int a)
{
   x = a;
   func1();
   func2();
}
```

アセンブリ言語展開コード

```
        .EXPORT     _func1
        .EXPORT     _func2
        .EXPORT     _func
        .SECTION    P,CODE,ALIGN=4
_func1:                            ; function: func1
                                   ; frame size=0
        RTS
        ADD         #1,R13
_func2:                            ; function: func2
                                   ; frame size=0
        MOV         #0,R3
        RTS
        MOV.B       R3,@R14
_func:                             ; function: func
                                   ; frame size=4
        STS.L       PR,@-R15
        BSR         _func1
        MOV         R4,R13
        BRA         _func2
        LDS.L       @R15+,PR
        .SECTION    B,DATA,ALIGN=4
        .END
```

■注意事項

(1) グローバル変数で、単純型またはポインタ型の変数に使用できます。また、"-double=float"オプションを指定した場合を除き、double型の変数は指定できません。（SH2A-FPU、SH-4,SH-4Aを除く）

(2) 指定可能なレジスタは、R8～R14,FR12～FR15（SH-2E,SH2A-FPU,SH-4,SH-4Aの場合）、DR12～DR14（SH2A-FPU,SH-4,SH-4Aの場合）です。

(3) 初期値の設定はできません。また、アドレスの参照もできません。

(4) 指定された変数の、リンク先からの参照は保証されません。

(5) 静的データメンバの指定は可能ですが、非静的データメンバの指定は不可能です。

■ FR12～FR15に設定可能な変数の型

(i)　SH-2Eの場合

・float型変数

・double型変数(double=floatオプション指定)

(ii)　SH2A-FPU,SH-4,SH-4Aの場合

・float型変数(fpu=doubleオプション指定なし)

・double型変数(fpu=singleオプション指定)

■ DR12～DR15に設定可能な変数の型

(i)　SH2A-FPU,SH-4,SH-4Aの場合

・float型変数(fpu=doubleオプション指定)

・double型変数(fpu=singleオプション指定なし)

3.5 レジスタ退避／回復の制御

■説明

関数呼び出ししか処理のない関数などから呼び出される関数において､レジスタの退避／回復を行わないようにし､実行速度を向上したい場合があります。このような場合、レジスタの退避／回復をきめ細かく制御するプリプロセッサ制御文#pragma noregsave, #pragma noregalloc,#pragma regsave を使用します。

（1） #pragma noregsaveは、関数の出入口で汎用レジスタの退避／回復を行わないことを指定します。
（2） #pragma noregallocは、関数の出入口で汎用レジスタの退避／回復を行わず、関数呼び出しを超えてレジスタ変数用レジスタ（R8～R14）を割り付けないオブジェクトを生成します。
（3） #pragma regsaveは、関数の出入口で汎用レジスタのうちR8～R14を全て退避／回復し、レジスタ変数用レジスタ（R8～R14）を割り付けないオブジェクトを生成します。
（4） #pragma regsaveと#pragma noregallocは同一関数に対して重複指定できます。重複指定した場合、関数の出入口でレジスタ変数用レジスタ（R8～R14）をすべて退避／回復し、関数呼び出しを超えてレジスタ変数用レジスタを割り付けないオブジェクトを生成します。

【書式】

```
#pragma noregsave(<関数名> [,<関数名>・・・])
#pragma noregalloc(<関数名> [,<関数名>・・・])
#pragma regsave(<関数名> [,<関数名>・・・])
```

■使用例

レジスタの退避／回復を削除または削除できる可能性のある状況を以下に示します。

使用例 1

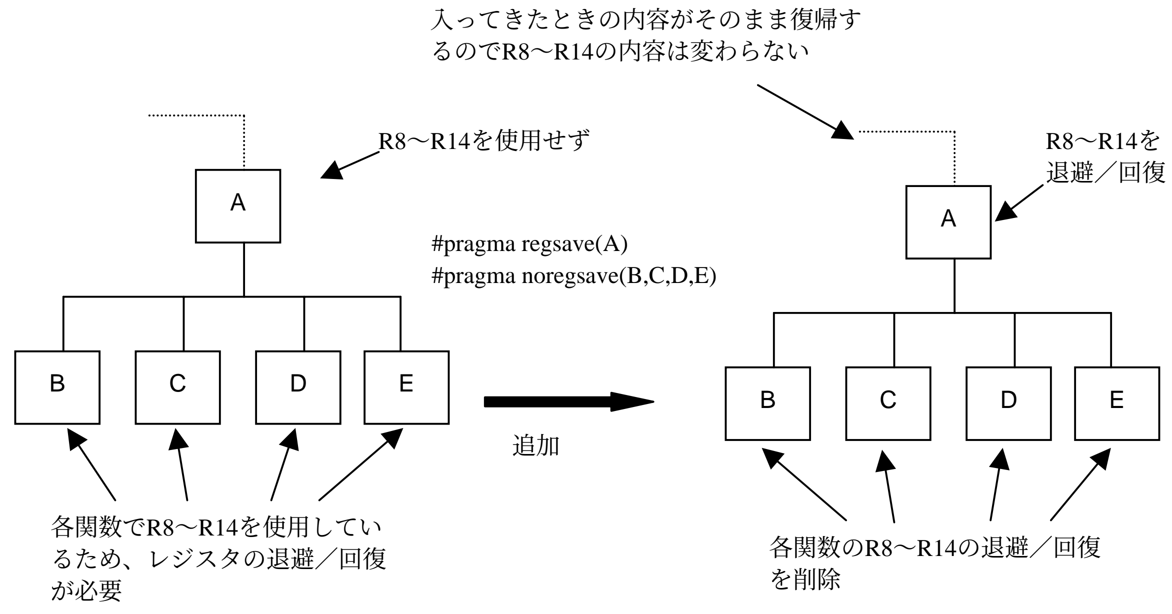
パワーオン時起動される関数で R8～R14 のレジスタを使用している場合などは､レジスタの退避／回復を行う必要がないため、"#pragma noregsave"を指定することにより、オブジェクトサイズおよび実行速度を向上できます。

使用例 2

呼び出し元へ戻らず低消費電力モードにする関数で R8～R14 のレジスタを使用している場合などは､レジスタの退避／回復を行う必要がないため、"#pragma noregsave"を指定することにより、オブジェクトサイズおよび実行速度を向上できます。

使用例 3

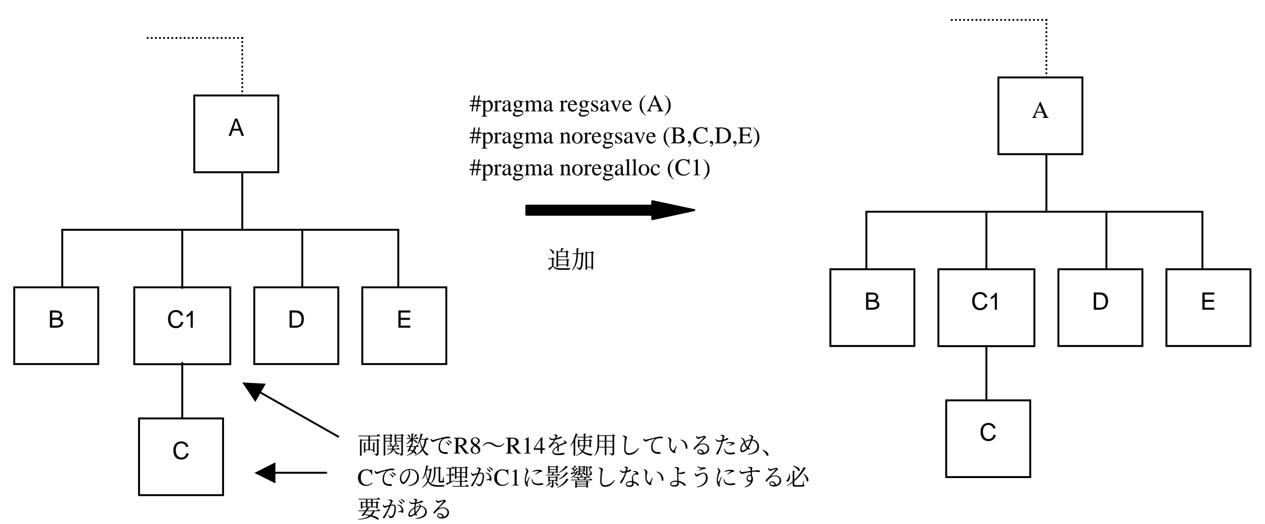
関数 A でレジスタ R8～R14 が割り付けられず、関数 B,C,D,E で R8～R14 が割り付けられている場合、関数 B,C,D,E の出入口で、R8～R14 の退避／回復を行うオブジェクトが生成されます。関数 A では R8～R14 を使用していないため、関数 A から呼び出される関数で退避／回復しなくとも影響がありませんが、関数 A を呼び出した関数で使用している場合がありますので、関数 A の出入口で退避／回復を行い、関数 A から呼び出される各関数では退避／回復を行わないようにできます。



使用例 4

使用例 3 と同様な呼び出し関係で、関数 C,C1 とも R8～R14 のレジスタを使用している場合、関数 C1 で R8～R14 を関数 C の呼び出しを超えて使用しないようにする必要があります。

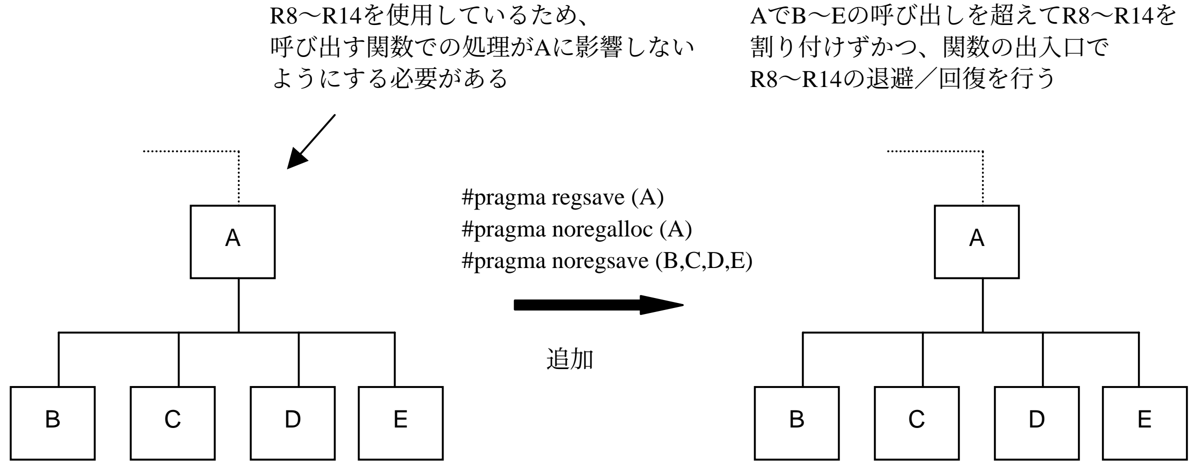
このような場合、関数 C1 を"#pragma noregalloc"で指定して R8～R14 を関数呼び出しを超えて割り付けないように指示すれば、関数 C を"#pragma noregsave"で指定することができるようになります。



RENESAS

使用例 5

使用例 3 と同様な呼び出し関係で、関数 A においても R8～R14 のレジスタを使用している場合、関数 A で R8～R14 を関数 B,C,D,E の呼び出しを超えて使用しないようにする必要があります。このような場合、関数 A を"#pragma regsave"と"#pragma noregalloc"で重複指定します。"#pragma regsave"と"#pragma noregalloc"を重複指定すると、R8～R14 の退避／回復を関数の出入口で行い、かつ R8～R14 を関数呼び出しを超えて割り付けないコードを出力するので、関数 B,C,D,E を"#pragma noregsave"で指定することができるようになります。



■注意事項

下記以外の方法で"#pragma noregsave"を指定した関数を呼び出した場合の結果は保証されませんので注意が必要です。

(1) 他の関数から呼び出されない、最初に起動する関数として使用する。
(2) "#pragma regsave"を指定した関数から呼び出す。
(3) "#pragma regsave"を指定した関数から、さらに"#pragma noregalloc"を指定した関数を介して呼び出す。

3.6 16/20/28/32 ビットアドレス領域指定

■説明

プリプロセッサ制御文を用いて外部参照される変数や関数のアドレスが 16/20/28/32 ビットであることをコンパイラに指示することができます。

コンパイラは、"#pragma abs16"で宣言された識別子は 16 ビットで表現できるアドレスとし、通常、32 ビット分割り付けられるアドレスの格納領域を 16 ビット分のみ割り付けるようにします。このようにして、オブジェクトサイズを小さくすることにより ROM 効率を向上させることができます。

また、設計時に複数の関数で参照される変数や関数などを、優先的に 16 ビットで表現できるアドレスに置かれるようなメモリ配置にしておけば、本機能を効果的に使うことができます。

SuperH RISC engine C/C++コンパイラ Ver.4.1 より、16 ビットアドレス変換指定がオプション化されました。このオプションにより一括指定することもできます。詳細は「**付録 B. 追加機能について**」を参照してください。

また、Ver.9.0 より SH-2A、SH2A-FPU において 20/28 ビットアドレス領域指定ができます。オプションによる一括指定も可能です。

【書式】

＜プリプロセッサ制御文＞

```
#pragma abs16 (<識別子> [,<識別子>・・・] )
#pragma abs20 (<識別子> [,<識別子>・・・] )
#pragma abs28 (<識別子> [,<識別子>・・・] )
#pragma abs32 (<識別子> [,<識別子>・・・] )
          識別子：変数名 | 関数名
```

＜オプション＞

```
abs16 = { program | const | data | bss | run | all }[,…]
abs20 = { program | const | data | bss | run | all }[,…]
abs28 = { program | const | data | bss | run | all }[,…]
abs32 = { program | const | data | bss | run | all }[,…]
          省略時解釈は、abs32=all
```

■使用例

使用例１

外部アクセスの変数および関数のアドレスを 16 ビットにします。

C 言語コード

```
#pragma abs16 (x,y,z)
extern int x();
int y;
long z;
f()
{
    z = x() + y;
}
```

アセンブラ言語展開コード

```
_f:
        STS.L    PR,@-R15
        MOV.W    L218,R3       ;x のアドレスのロード
        JSR      @R3
        NOP
        MOV.W    L218+2,R3     ;y のアドレスのロード
        MOV.L    @R3,R2
        MOV.W    L218+4,R1     ;z のアドレスのロード
        ADD      R2,R0
        LDS.L    @R15+,PR
        RTS
        MOV.L    R0,@R1
L218:
        .DATA.W  _x
        .DATA.W  _y
        .DATA.W  _z
```

使用例２
　外部アクセスの変数および関数のアドレスを 20 ビットにします。

C 言語コード

```
#pragma abs20 (x,y,z)
extern int x();
int y;
long z;
f()
{
    z = x() + y;
}
```

アセンブラ言語展開コード

```
_f:
        STS.L      PR,@-R15
        MOVI20     #_x,R2         ; x のアドレスロード
        JSR/N      @R2
        MOVI20     #_y,R5         ; y のアドレスロード
        MOV.L      @R5,R1
        MOVI20     #_z,R4         ; z のアドレスロード
        ADD        R1,R0
        LDS.L      @R15+,PR
        RTS
        MOV.L      R0,@R4
_y:
        .RES.L     1
_z:
        .RES.L     1
```

使用例３
　外部アクセスの変数および関数のアドレスを 28 ビットにします。

C 言語コード

```
#pragma abs28 (x,y,z)
extern int x();
int y;
long z;
f()
{
    z = x() + y;
}
```

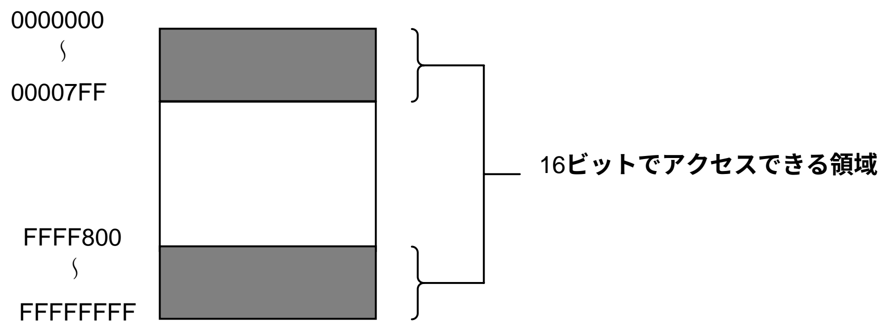
アセンブラ言語展開コード

```
_f:
        STS.L      PR,@-R15
        MOVI20S    #_x+H'80,R2    ; x のアドレスロード
        ADD        #Low _x,R2
        JSR/N      @R2
        MOVI20S    #_y+H'80,R5    ; y のアドレスロード
        ADD        #Low _y,R5
        MOV.L      @R5,R1
        MOVI20S    #_z+H'80,R4    ; z のアドレスロード
        ADD        #Low _z,R4
        ADD        R1,R0
```

```
        LDS.L       @R15+,PR
        RTS
        MOV.L       R0,@R4
_y:
        .RES.L      1
_z:
        .RES.L      1
```

■注意事項

（1） abs16/20/28/32指定をした変数および関数はセクション切り替え機能で別セクションにし、リンク時に指定ビットで表現できるアドレスになるようにセクションを配置してください。指定ビットアドレスで表現されるアドレスに配置されていないとリンク時にエラーとなります。



各指定ごとの配置可能なアドレス範囲を下表に示します。

表 3.23 アドレス範囲

#pragma/オプション	アドレス範囲	
	下位	上位
abs16	0x00000000	0x00007FFF
	0xFFFF8000	0xFFFFFFFF
abs20	0x00000000	0x0007FFFF
	0xFFF80000	0xFFFFFFFF
abs28	0x00000000	0x07FFFF7F*
	0xF8000000	0xFFFFFFFF
abs32	0x00000000	0xFFFFFFFF

【注】*0x07FFFF7F となることに注意。

（2） コンパイル時にポジションインディペンデントコードの生成を指定していると、関数アドレスは指定ビットで生成されません。

3.7 セクション名指定

1 つのシステムで同一属性のセクションを別々のアドレスに割り付けたい場合（たとえば、あるモジュールを外付け RAM に、別のモジュールを内蔵 RAM に割り付けたい場合など）、分割したいセクションそれぞれに名称をつけ、リンク時にそれぞれのセクションに配置したいアドレスを指定する方法をとります。SuperH RISC engine C/C++コンパイラではセクション名称を指定する方法として、2 種類の方法が用意されています。以下に複数のモジュールに別々のセクション名を指定する方法を示します。なお、説明中の例ではモジュール f,g,h、データ a,b をそれぞれ f,h,a と、g,b に分割する場合を想定しています。

3.7.1 セクション名指定

SuperH RISC engine C/C++コンパイラでは、コンパイル時に"-section"オプションを指定することによりオブジェクトのセクション名を指定することができます。この機能を利用して、分割したいモジュールおよびデータどうしを別々のファイルにまとめ、コンパイル時に異なるセクション名を指定し、リンク時にそれぞれのスタートアドレスが指定できるようにします。

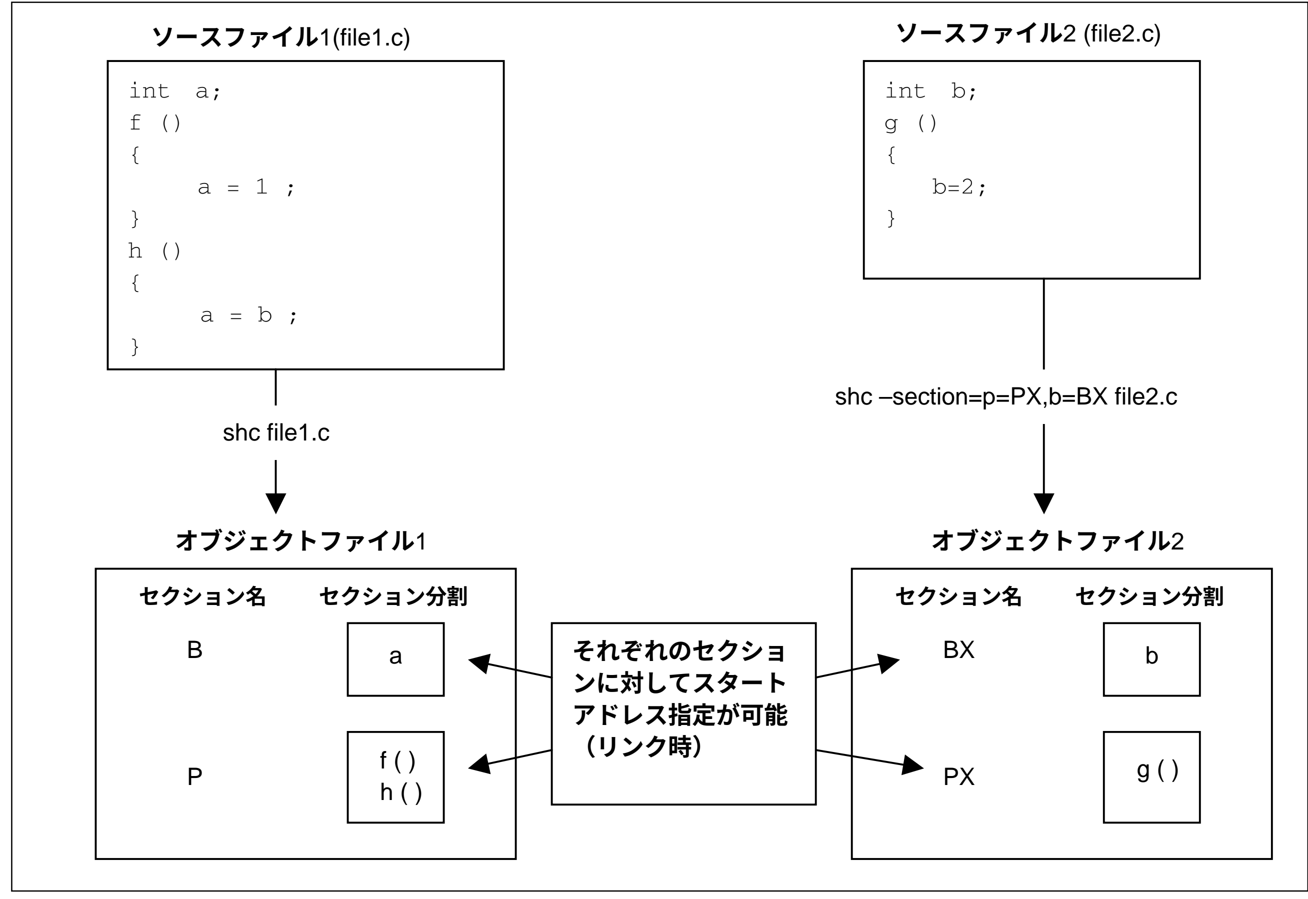


図 3.3 セクション名の指定方法

3.7.2 セクション切り替え

"-section"オプションでは、セクション名称をファイル単位でしか指定できませんでしたが、"#pragma section"を用いることにより、1つのファイル内の同一属性のセクション名称を切り替えて、よりきめ細かなメモリ割り付けが可能となります。本機能により、「**3.7.1 セクション名指名**」で示したセクション分割も１つのファイルに記述することができます。この機能を用いた例を**図** 3.4 に示します。

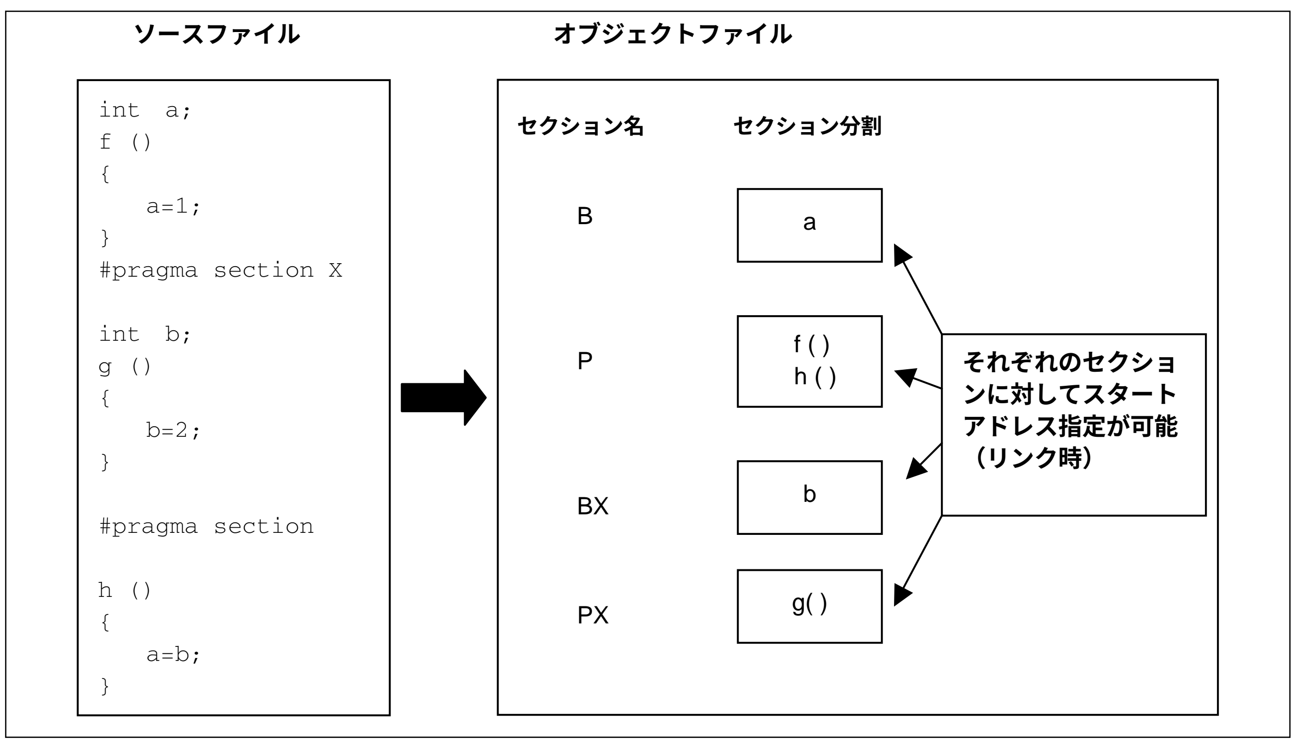


図 3.4　**セクション切り替え方法**

この図では、"#pragma section X"を指定することにより、この行から"#pragma section"で指定された行までのプログラム領域セクションの名称は"PX"に、未初期データセクションの名称は"BX"になります。

"#pragma section"の指定によりデフォルトのセクション名に戻ります。

3.8 エントリ関数の指定、SP の設定

■説明

〈関数名〉で指定した関数をエントリ関数として扱います。エントリ関数ではレジスタの退避・回復コードを一切作成しません。CPU が SH-3,SH3-DSP,SH-4,SH-4A,SH4AL-DSP の場合では、sp=<定数>指定、もしくは#pragma stacksize 宣言があると、関数先頭でスタックポインタの初期設定コードを出力します。

【書式】

```
#pragma entry (<関数名>=<(sp=<定数>))
```

■使用例

C 言語コード

```
  例 1
#pragma entry INIT(sp=0x10000)
void INIT() {
    :
}

  例 2
#pragma stacksize 100
#pragma entry INIT
void INIT() {
    :
}
```

アセンブリ言語展開コード

```
  例 1

 .SECTION P, CODE
_INIT:
 MOV.L L1, R15
    :
L1: .DATA.L H'00010000
  :
例 2
 .SECTION S, STACK
 .RES.B 100
 .SECTION P, CODE
_INIT:
 MOV.L L1, R15
    :
L1: .DATA.L STARTOF S + SIZEOF S
    :
```

■注意事項

#pragma entry 指定は、関数の宣言前に行ってください。エントリ関数はロードモジュール全体の 2 つまでしか指定できません。<定数>は必ず 4 の倍数を指定してください。

cpu=sh1|sh2|sh2e|sh2a|sh2afpu|sh2dsp を指定した場合、sp=<定数>指定は無効になります。

3.9 ポジションインディペンデントコード

実行速度を向上させるために、起動時に ROM 上のコードを RAM 上に移動し、RAM 上で動作させる場合があります。これを実現するには、プログラムが任意のアドレスへロード可能になっている必要があります。このようになっているコードをポジションインディペンデントコードと呼びます。

SuperH RISC engine C/C++コンパイラではコンパイル時のコマンドラインオプションに"pic=1"を指定することにより、ポジションインディペンデントコードを生成できます。

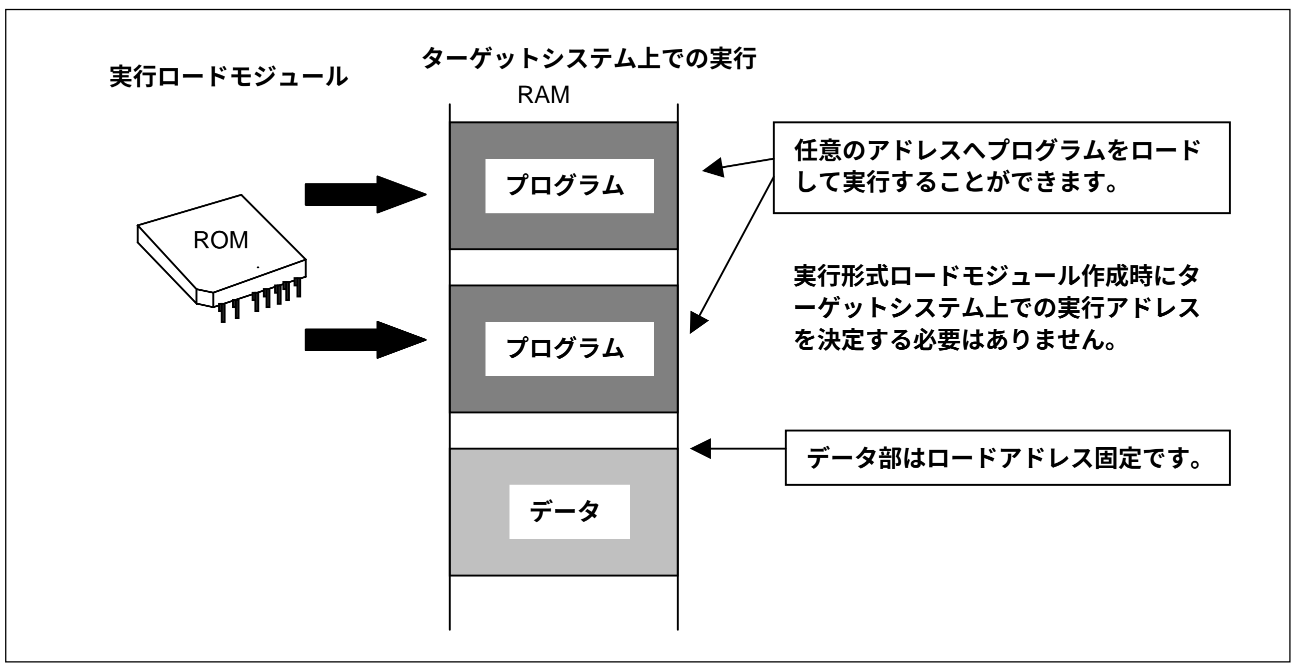


図 3.5 ポジションインディペンデントコード

【注】
(1) **本機能は SH-1 では使用できません。**
(2) **本機能はデータセクションに対しては適用できません。**
(3) **ポジションインディペンデントコードとして実行する場合には、関数のアドレスを初期値として指定することはできません。**
例）

```
extern int f();
int (*fp)() = f;
```

関数 f のアドレスは RAM 上にロードされるまで確定しないので、この場合、動作は保証されません。

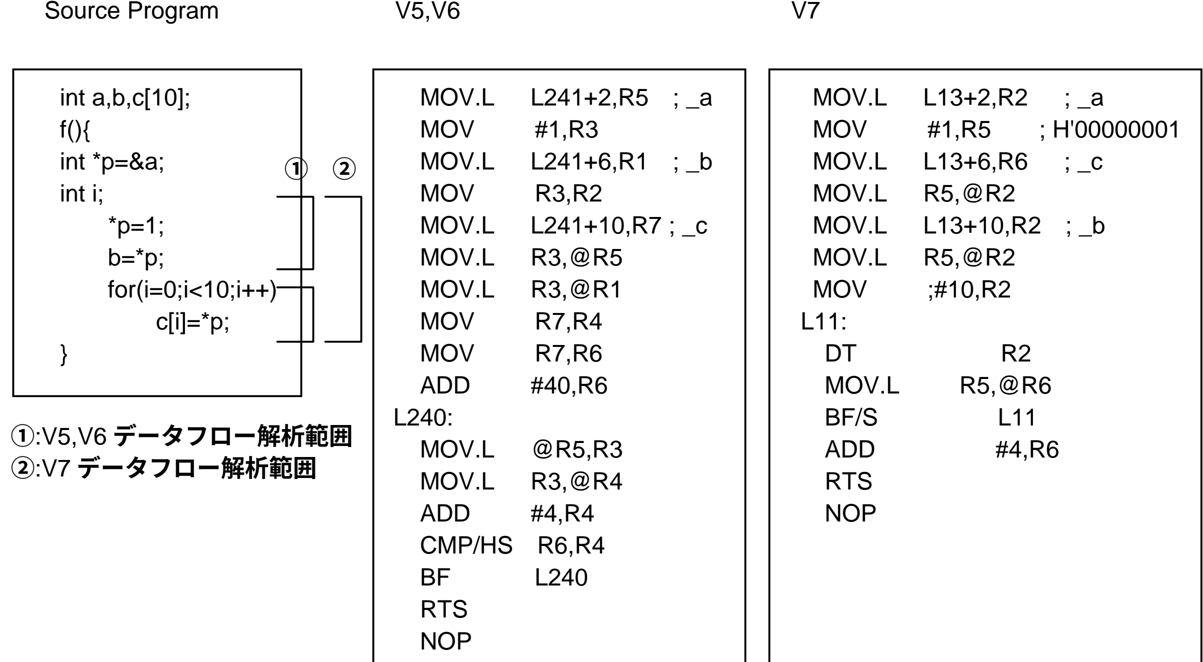
(4) **本機能を使用する場合は、ポジションインディペンデントコード対応の標準ライブラリを使用してください。ライブラリの構築についての詳細は「SuperH RISC engine C/C++コンパイラ、アセンブラ、最適化リンケージエディタ ユーザーズマニュアル」を参照してください。**

RENESAS

3.10 MAP 最適化

スーパーコンピュータ向け最新の最適化処理を適用し、ポインタや外部変数の別名解析、制御文を含めたデータフロー解析により、さらに強力な最適化を実現しています。

最適化例



3.10.1 使用方法

コンパイル、リンクによって確定したシンボル割り付けアドレスを使用して再コンパイルします。
これにより、割り付けアドレスに依存した最適化をコンパイラで実現できます。

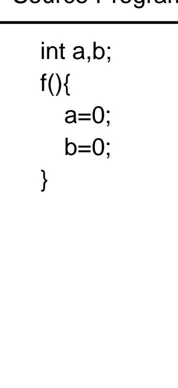
使用方法：
1 回目のコンパイル、リンク：
通常オプションでコンパイル
-map=<file>.bls オプション付きでリンク -> <file>.bls を出力

2 回目のコンパイル、リンク：
-map=<file>.bls オプション付きでコンパイル
通常オプションでリンク

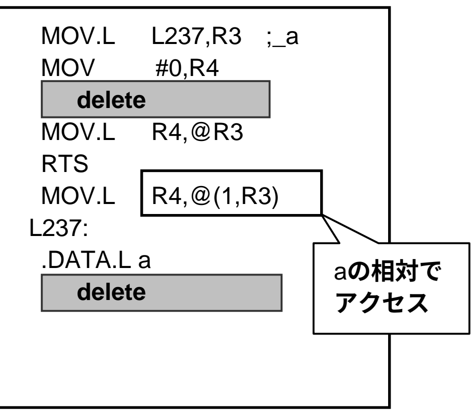
3.10.2 外部変数アクセスコード改善例（1）

変数割り付け順序を意識して、連続して割り付けられる変数を同一レジスタの相対でアクセスします。

Source Program　　V5,V6　　V7(MAP 指定時)




```
MOV.L    L237,R3   ;_a
MOV      #0,R4
MOV.L    L237+4,R2;_b
MOV.L    R4,@R3
RTS
MOV.L    R4,@R2
L237:
 .DATA.L a
 .DATA.L b
```



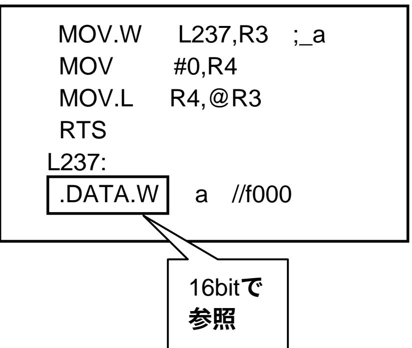
3.10.3 外部変数アクセスコード改善例（2）

0～7FFF,FFFF8000～FFFFFFFF に割り付けられた変数を 16bit リテラルでアクセスします。

Source Program　　V5,V6　　V7(MAP 指定時)

```
int a;
f(){
   a=0;
}
```

```
MOV.L    L237,R3   ;_a
MOV      #0,R4
MOV.L    R4,@R3
RTS
L237:
 .DATA.L a   //ffff0000
```

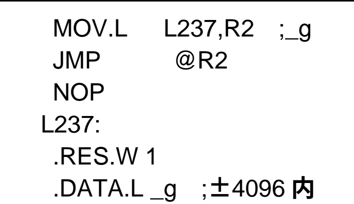


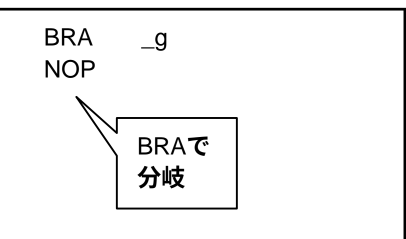
3.10.4 外部変数アクセスコード改善例（3）

分岐先アドレスを意識して、BSR/BRA で分岐します。

Source Program　　V5,V6　　V7(MAP 指定時)

```
extern g();
f(){
   g();
}
```





3.10.5 外部変数アクセスコード改善例（4）

gbr=auto オプション（デフォルト）時、外部変数アクセスのベースとして、GBR を使用します。

Source Program

```
int a[101];
int b;

void f()
{
   int i;

   for( i=0;
   i<100;
   i++ )
      a[i] = 0;

   a[50] = b;
}
```

V5,V6

```
_f:
     MOV.L      L239+4,R7    ; _a
     MOV        #0,R5
     MOV.W      L239,R6       ; H'0190
     MOV        R7,R4
     ADD        R7,R6
L238:
     MOV.L      R5,@R4
     ADD        #4,R4
     CMP/HS     R6,R4
     BF      L238
     MOV.L      L239+8,R2    ;
H'0C8+_a
     MOV.L      L239+12,R1 ; _b
     MOV.L      @R1,R3
     RTS
     MOV.L      R3,@R2
L239:
     .DATA.W   H'0190
     .DATA.W   0
     .DATA.L   _a
     .DATA.L   H'000000C8+_a
     .DATA.L   _b
```

V7(MAP 指定時)

```
_f:
     STC        GBR,@-R15
     MOV.L      L13+2,R0     ; _a
     LDC        R0,GBR
     MOV        #100,R6      ;
     STC        GBR,R2
     MOV        #0,R5
L11:
     ADD        #-1,R6
     MOV.L      R5,@R2
     TST        R6,R6
     ADD        #4,R2
     BF      L11
     MOV.L      @(404,GBR),R0
     MOV.L      R0,@(200,GBR)
     RTS
     LDC        @R15+,GBR
L13:
     .RES.W    1
     .DATA.L   _a
```

GBR相対で参照

3.11 オプション

3.11.1 コード生成に関するオプション

SuperH RISC engine C/C++コンパイラでは、コード生成の方針をユーザが選択できるようにするため、以下のオプションを設けています。

表 3.24 コード生成に関するオプション

オプション	説　明
-SPeed	速度優先のコードを生成します。
-SIze	サイズの縮小を優先してコードを生成します。
-Goptimize	モジュール間最適化用付加情報を出力します。
-MAP	最適化リンケージエディタが生成する外部シンボル割付情報を元にベースアドレスを設定し、外部アクセスをベースアドレス相対で行うコードを生成します。 gbr=auto を指定した場合、条件によりベースアドレスを GBR レジスタに設定し、外部アクセスを GBR 相対で行うコードを生成します。
-GBr	gbr=auto を指定した場合、条件によりコンパイラが自動的に GBR 相対論理演算コードを生成します。gbr=auto かつ MAP=＜ファイル名＞を指定した場合は、条件により GBR にベースアドレスを設定し、外部変数アクセスを GBR 相対で行うコードを生成します。
-CAse	Switch 分のコード展開方式を指定します。case =ifthen を指定した場合、switch 文を if_then 方式で展開します。この方式は switch 文に含まれる case ラベルの数に比例してオブジェクトコードサイズが増大します。 case=table を指定した場合、switch 文をテーブル方式で展開します。この方式は、switch 文に含まれる case ラベルの数に比例して定数領域に確保されるジャンプテーブルのサイズは増えますが、実行速度は常に一定です。本オプション省略時は、いずれかの展開方式をコンパイラが自動的に判断します。
-SHIft	shift=inline を指定した場合、すべてのシフト演算を命令展開します。 shift=runtime をしていした場合、展開する命令が多い場合は実行時ルーチン呼び出しになります。
-BLOckcopy	blockcopy=inline を指定した場合、すべてのメモリ間転送コードを命令展開します。 blockcopy=runtime を指定した場合、転送サイズが大きい場合は実行時ルーチン呼び出しになります。
-INLine	関数の自動インライン展開を行うかどうかを指定します。 inline オプションを指定した場合、自動インライン展開を行います。サイズ指定が可能です。
-DIvision	プログラム中の整数型除算、剰余算の方式を選択します。 division=cpu=inline を指定した場合、定数除算は乗算に変換してインライン展開し、変数除算は DIV１命令による実行時ルーチンを選択します。
-Macsave	MACH、MACL レジスタを関数呼び出しの前後で保証するかどうかを指定します。 ０を指定した場合、関数の呼び出し前後で MACH、MACL レジスタを保証しません。

3.11.2 最適化リンケージに関するオプション

SuperH RISC engine C/C++最適化リンケージエディタでは、最適化リンケージの方針をユーザが選択できるようにするため、以下のオプションを設けています。

表 3.25 リンケージに関するオプション

オプション	サブオプション	説　明
-OPtimize	-	モジュール間最適化実行有無を指定します。
	-SPeed	オブジェクトスピード低下を招く可能性のある最適化以外を実行します。ptimize=string_unify,symbol_delete,variable_access,register,branch と同じです。
	-SAFe	変数や関数の属性によって制限される可能性のある最適化以外を実行します。optimize =string_unify,register,branch と同じです。
	-Branch	プログラムの配置情報に基づいて、分岐命令サイズを最適化します。他の最適化項目を実行すると、指定の有無にかかわらず実行します。
	-Register	関数呼び出し関係を解析し、レジスタの再割り付けおよび冗長なレジスタ退避・回復コードを削除します。
	-STring_unify	const 属性を持つ定数に対し、同一値定数を統合します。 const 属性を持つ定数には次のものが含まれます。 ・C/C++プログラム中で const 宣言した変数 ・文字列データの初期値 ・リテラル定数
	-SYmbol_delete	一度も参照のない変数／関数を削除します。
	-Variable_access	8/16 ビット絶対アドレッシングモードでアクセス可能な領域にアクセス回数の多い変数を割り当てます。
	-SAMe_code	複数の同一命令列をサブルーチン化します。
	-Function_call	0～0xFF の範囲に空きがあれば、アクセス回数の多い関数のアドレスを割り付けます。
-SMAESize	-	共通コード統合最適化で、最適化対象となる最小コードサイズを指定します。
-PROfile	-	プロファイル情報ファイルを指定します。モジュール間最適化で、動的情報に基づいた最適化を実行することができます。
-CAchesize	-	キャッシュサイズおよびキャッシュラインサイズを指定します。profile オプションを指定した場合に、分岐命令最適化で使用します。
-SYmbol_forbid	-	未参照シンボル削除最適化を抑止します。
-SAMECode_forbid	-	共通コード統合最適化を抑止します。
-Variable_forbid	-	短絶対アドレッシングモード活用最適化を抑止します。
-FUnction_forbid	-	間接アドレッシングモード活用最適化を抑止します。
-Absolute_forbid	-	アドレス＋サイズの範囲の最適化を抑止します。

3.11.3 標準ライブラリ構築に関するオプション

SuperH RISC engine C/C++標準ライブラリ構築ツールでは、標準ライブラリ構築時の最適化方針をユーザが選択できるようにするため、以下のオプションを設けています。

表 3.26 標準ライブラリ構築最適化に関するオプション

オプション	説明
-SPeed	速度優先のコードを生成します。
-SIze	サイズの縮小を優先してコードを生成します。
-Goptimize	モジュール間最適化用付加情報を出力します。
-MAP	最適化リンケージエディタが生成する外部シンボル割り付け情報を元にベースアドレスを設定し、外部アクセスをベースアドレス相対で行なうコードを生成します。 gbr=auto を指定した場合、条件によりベースアドレスを GBR レジスタに設定し、外部アクセスを GBR 相対で行うコードを生成します。
-GBr	gbr=auto を指定した場合、条件によりコンパイラが自動的に GBR 相対論理演算コードを生成します。gbr=auto かつ MAP=＜ファイル名＞を指定した場合は、条件により GBR にベースアドレスを設定し、外部変数アクセスを GBR 相対で行うコードを生成します。
-CAse	Switch 分のコード展開方式を指定します。case =ifthen を指定した場合、switch 文を if_then 方式で展開します。この方式は switch 文に含まれる case ラベルの数に比例してオブジェクトコードサイズが増大します。 case=table を指定した場合、switch 文をテーブル方式で展開します。この方式は、switch 文に含まれる case ラベルの数に比例して定数領域に確保されるジャンプテーブルのサイズは増えますが、実行速度は常に一定です。本オプション省略時は、いずれかの展開方式をコンパイラが自動的に判断します。
-SHIft	shift=inline を指定した場合、すべてのシフト演算を命令展開します。 shift=runtime を指定した場合、展開する命令が多い場合は実行時ルーチン呼び出しになります。
-BLOckcopy	blockcopy=inline を指定した場合、すべてのメモリ間転送コードを命令展開します。 blockcopy=runtime を指定した場合、転送サイズが大きい場合は実行時ルーチン呼び出しになります。
-INLine	関数の自動インライン展開を行うかどうかを指定します。 inline オプションを指定した場合、自動インライン展開を行います。サイズ指定が可能です。

3.12 SH-DSP の特長

SH-DSP コアは 16 ビット固定小数点を扱う、

積和演算

繰り返し処理

に最適な DSP ユニットを搭載しているので、マルチメディア演算に必要な JPEG 処理、音声処理、フィルタ処理などを高速に実行することができます。

従来の SH コア（**図 3.6** は SH-1 コアの例）では、パイプラインのとおり、乗算器の動作時間の 3 サイクルが積和演算の性能を決定してしまいます。また、仮に乗算器の動作時間を 1 サイクルに改善しても命令データ転送によるパイプラインのストールがおこり、長期平均時間は 2.5 サイクルになってしまいます。

SH-DSP コアでは、DSP ユニットの動作時間の１サイクル化とデータバス用に XY のバスを設けたことにより、積和演算の１サイクル化を実現しています（**図 3.7**）。この場合、長期平均時間も１サイクルとなります。

コード例

```
clrmac
mac.w @r4+,@r5+
mac.w @r4+,@r5+
mac.w @r4+,@r5+
mac.w @r4+,@r5+
rts
sts     macl,r0
```

IF	**：命令フェッチ(32ビット)**
if	**：命令フェッチ (バスサイクルなし)**
ID	**：デコード**
EX	**：実行／アドレス計算**
MA	**：メモリアクセス**
mul	**：乗算器動作**
WB	**：ライトバック**

パイプライン動作例

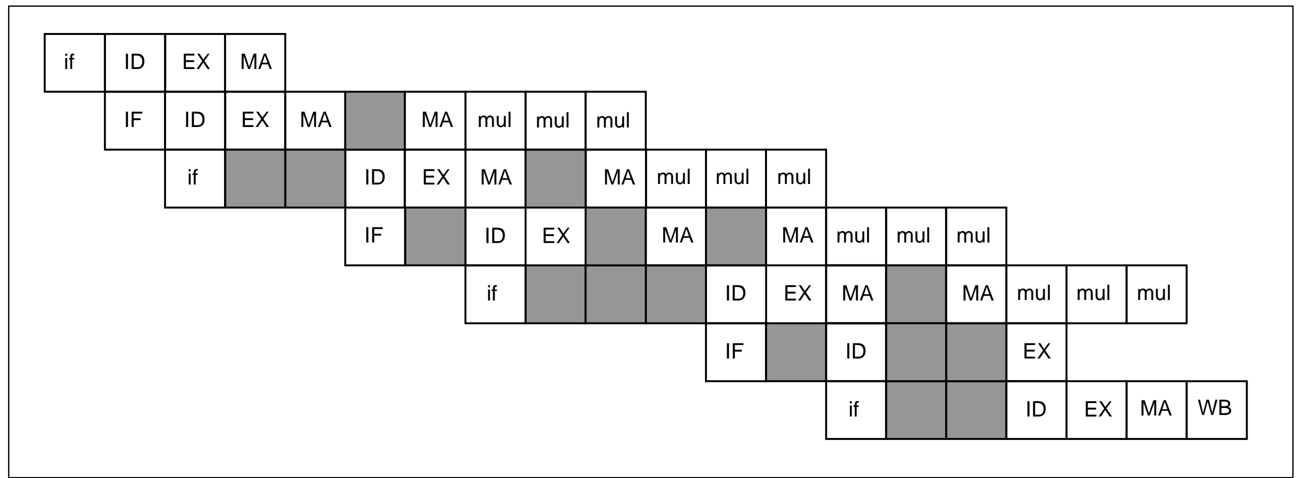
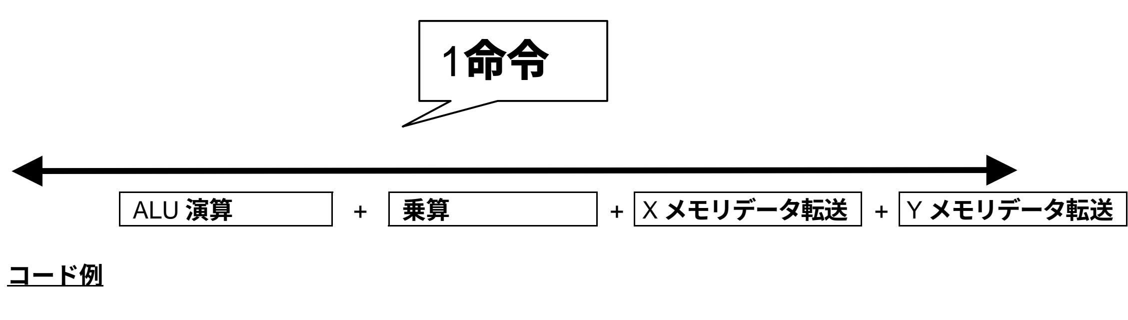


図 3.6 SH **コアの積和命令**



コード例

命令 1			MOVX.W@R4+,X0	MOVY.W@R6+,Y0
命令 2		PMULSX0,Y0,M0	MOVX.W@R4+,X1	MOVY.W@R6+,Y1
命令 3	PADD A0,M0,A0	PMULSX1,Y1,M1	MOVX.W@R4+,X0	MOVYW@R6+,Y0
命令 4	PADD A0,M1,A0	PMULS X0,Y0,M0	MOVX.W@R4+,X1	MOVYW@R6+,Y1

パイプライン動作例

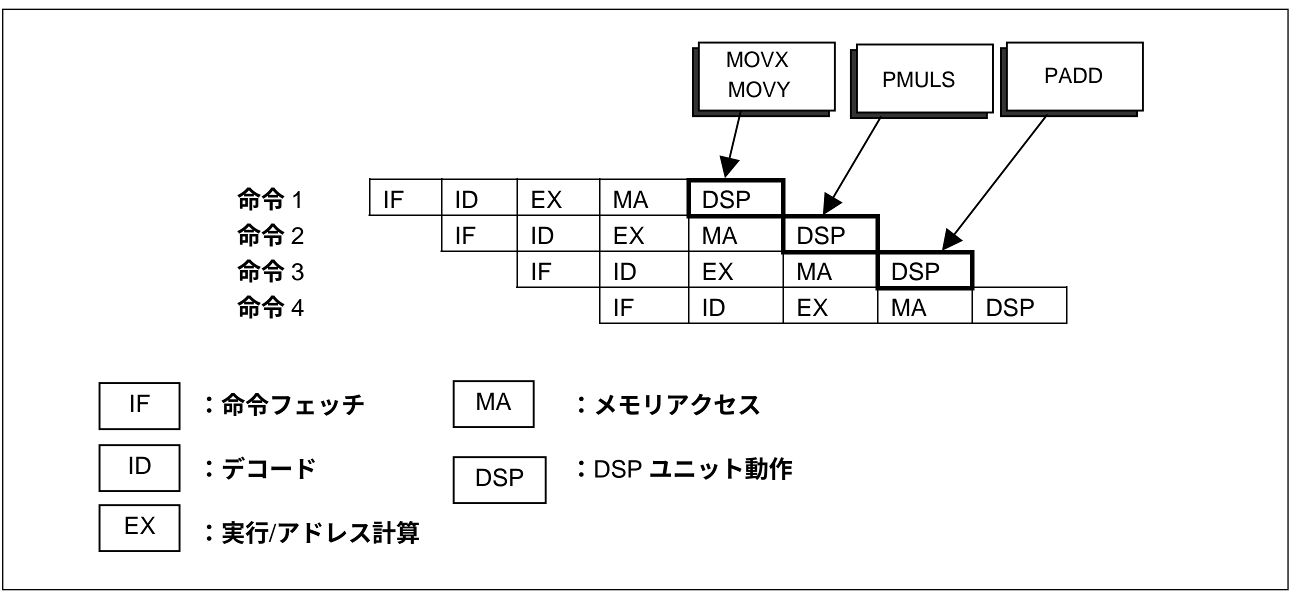


図 3.7　SH-DSP コアの積和命令の例

また、SH-DSP コアでは、繰り返し処理によるパイプラインの乱れを低減するためのハードウェア機構が搭載されています。

従来の SH コアでは、ループ処理には条件分岐命令を使用します。条件分岐命令はパイプラインの乱れを発生させ、処理にオーバヘッドをだしてしまいます。

SH-DSP コアでは、このループ処理によるパイプラインの乱れをゼロにするゼロオーバヘッドの機構があります。あらかじめ、ループの開始終了のアドレスとループ回数をセットしておくだけで、条件分岐の処理を行わずにループ処理が完了します。ソフト的にクリティカルな処理はループ処理になっている場合が多く、ソフト処理の高速化に有効なハードウェア機構です。

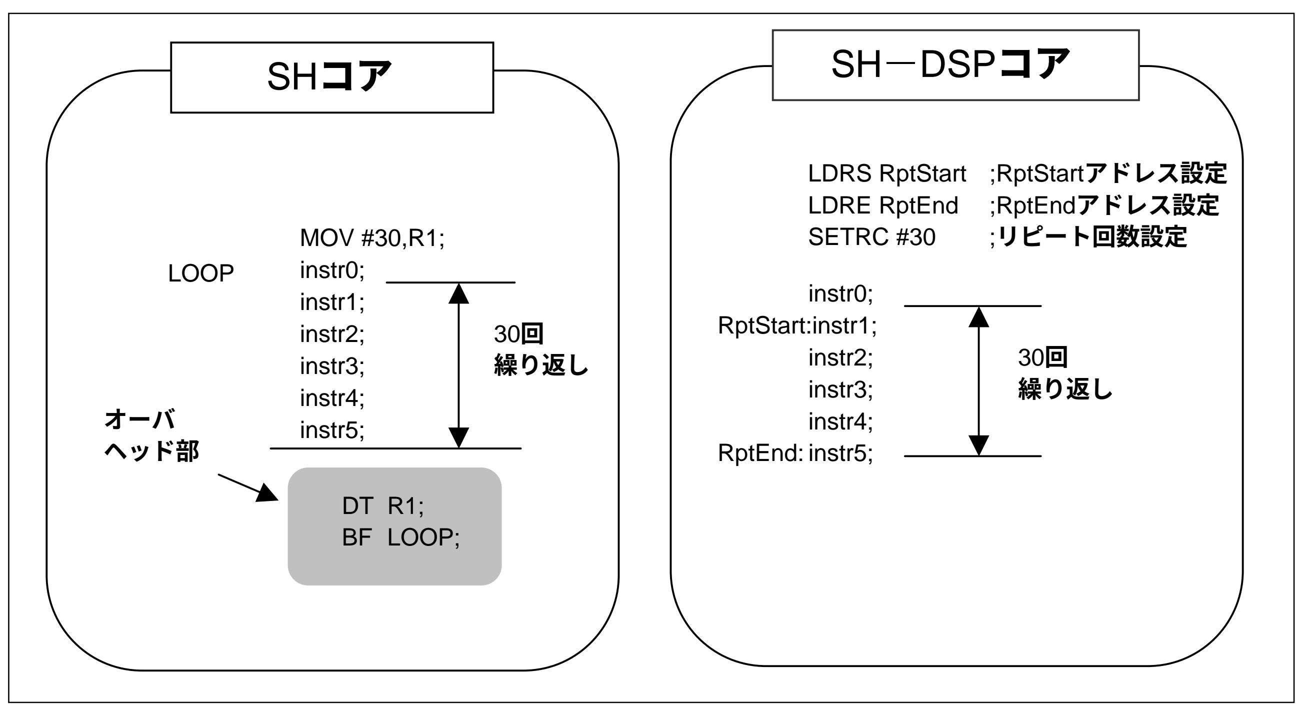


図 3.8 繰り返し処理

SH-DSP コアは、**図** 3.9 に示すように実行条件の判定、ALU 演算、符号付き乗算、X メモリアクセス、Y メモリアクセスの 5 つの命令を並行して実行することができます。これらの命令を組み合わせることによって、さまざまな積和演算の処理を高速に行うことができるわけです。

実行条件	ALU **演算**	**符号付き乗算**	X **メモリ・アクセス**	Y **メモリ・アクセス**
	PADD PSUB	PMULS	MOVX.W NOPX	MOVY.W NOPY
――：**無条件** DCT：DC=1 DCF：DC=0	PADD PCLR PNEG PLDS PCOPY PSTS PINC PSUB PDEC PSHA PSHL PAND POR PXOR PDMSB			
	PADDC PMULS PSUBC PSHA# PCMP PSHL# PRND			

図 3.9 DSP **命令（並列命令）**

3.13 DSP ライブラリ

3.13.1 概要

SH2-DSP および SH3-DSP(以降、併せて単に SH-DSP とよぶこととします)で使用できるデジタル信号処理(DSP)ライブラリについて説明します。本ライブラリは標準的な DSP 関数を含んでおり、単独または連続的に使用することによって、DSP 演算を行うことができます。

本ライブラリは以下の関数を含んでいます。

- 高速フーリエ変換
- 窓関数
- フィルタ
- 畳み込みと相関
- その他

本ライブラリ関数は高速フーリエ変換とフィルタを除いてリエントラントです。

本ライブラリを使用するときは、**表** 3.27 に示すファイルをインクルードしてください。また、**表** 3.28 に示すように、CPU およびコンパイルオプションに対応するライブラリをリンクしてください。

本ライブラリを呼び出した際、関数が正常に終了した場合は EDSP_OK を、異常があった場合は EDSP_BAD_ARG もしくは EDSP_NO_HEAP をリターン値として返します。リターン値の詳細については各関数の説明を参照してください。

表 3.27 DSP **ライブラリ用のインクルードファイル**

ライブラリの種類	内容	インクルードファイル
DSP ライブラリ	DSP 演算を行うライブラリです。	<ensigdsp.h>
		<filt_ws.h>*1

【注】 *1 フィルタ関数を使用する場合、ユーザプログラム内で１回のみインクルードしてください。

表 3.28 DSP **ライブラリ一覧**

CPU	オプション		ライブラリ名
SH2-DSP	-pic=0		shdsplib.lib
	-pic=1		shdsppic.lib
SH3-DSP SH4AL-DSP	-pic=0	-endian=big	sh3dspnb.lib
	-pic=1	-endian=big	sh3dsppb.lib
	-pic=0	-endian=little	sh3dspnl.lib
	-pic=1	-endian=little	sh3dsppl.lib

3.13.2 データフォーマット

本ライブラリはデータを符号付き16ビット固定小数点数として扱います。符号付き16ビット固定小数点数は**図**3.10(a)に示すように、小数点が最上位ビット(MSB)の右側に固定されたデータフォーマットとなっており、-1～$1-2^{-15}$の範囲の値を表現できます。

本ライブラリでは、データの受け渡しはshort型のデータフォーマットを使用します。したがって、C/C++プログラムから本ライブラリを使用する場合、データを符号付き16ビット固定小数点数で表現する必要があります。

(例)　+0.5は符号付き16ビット固定小数点数で表現するとH'4000です。したがってライブラリ関数に渡すshort型実引数はH'4000となります。

本ライブラリ内部の演算では、符号付き32ビット固定小数点数と符号付き40ビット固定小数点数も使用します。符号付き32ビット固定小数点数は**図**3.10(b)に示すデータフォーマットとなっており､-1～$1-2^{-31}$の範囲の値を表現できます。符号付き40ビット固定小数点数は**図**3.10(c)に示すように8ビットのガードビットが付加されたデータフォーマットとなっており、-2^{8}～$2^{8}-2^{-31}$の範囲の値を表現できます。

符号付き16ビット固定小数点数の乗算結果は符号付き32ビット固定小数点数で保持します。DSP命令を用いた固定小数点乗算では、H'8000×H'8000の場合だけオーバフローが発生することに注意してください。また乗算結果の最下位ビット(LSB)は常に0になります。乗算結果を次の演算に使用する場合、上位16ビットを取り出し、符号付き16ビット固定小数点数に変換します。このときアンダフローや精度低下が発生する可能性があります。

本ライブラリの積和演算では、加算結果を符号付き40ビット固定小数点数で保持します。加算のときにオーバフローが発生しないように注意してください。

演算の際、オーバフローが発生すると正しい結果が得られません。オーバフローを防ぐためには、係数や入力データをスケーリングする必要があります。本ライブラリには、スケーリングの機能が組み込まれています。スケーリングの詳細については各関数の説明を参照してください。

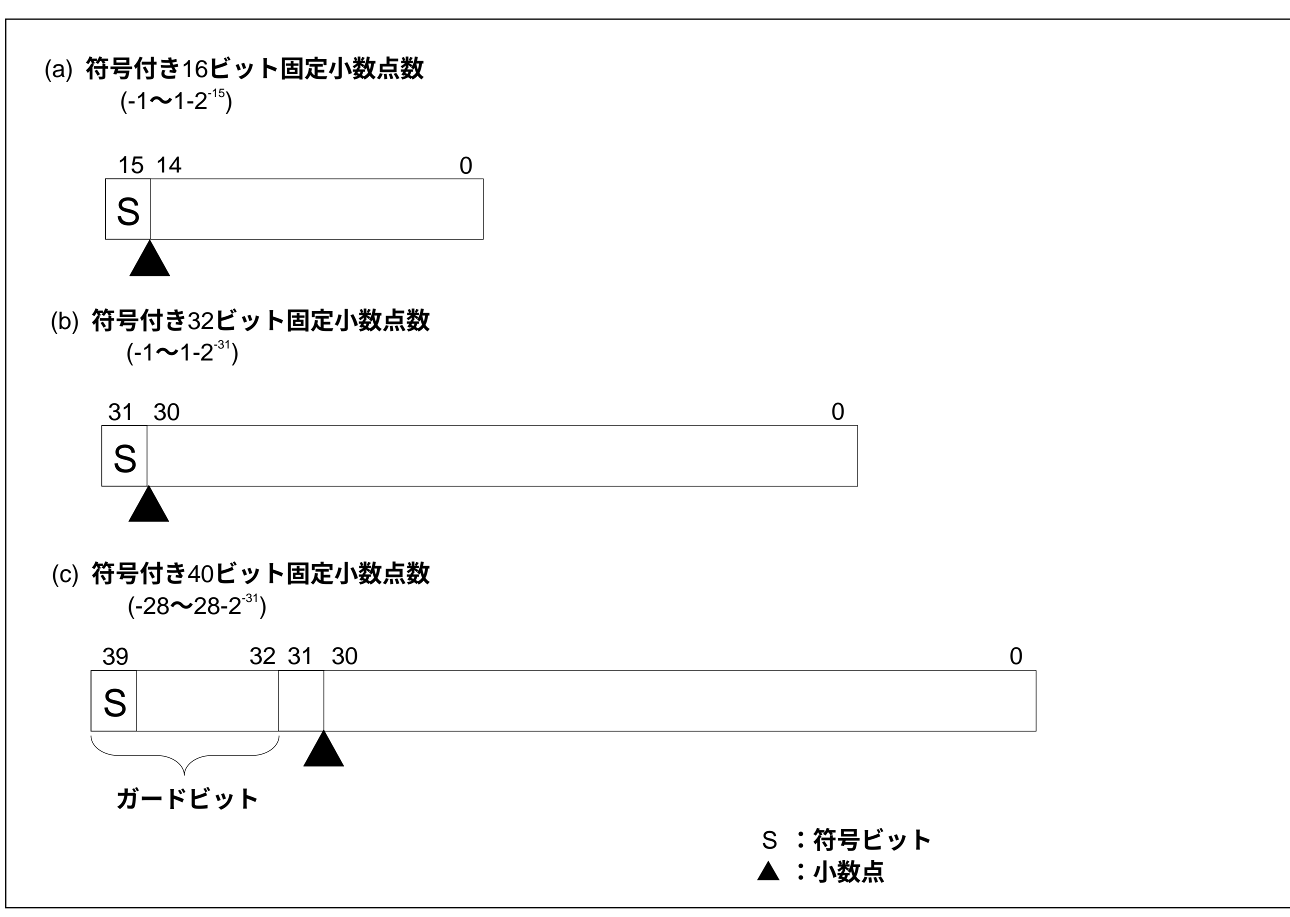


図3.10　データフォーマット

3.13.3 効率

本ライブラリ関数は、SH-DSP 上で高速に実行するように最適化しています。

ライブラリを効率よく活用するために、開発するシステムのメモリマップを決める際には、できるだけ以下の 2 つの推奨事項に従ってください。

プログラムコードセグメントは、1サイクルでの32ビットリードをサポートしているメモリに配置する。

データセグメントは、1サイクルでの16(または32)ビットリード・ライトをサポートしているメモリに配置する。

使用するマイコンが､ライブラリコードとデータを配置するのに十分な容量の 32 ビットメモリを内蔵している場合は、その 32 ビットメモリに配置するのが最適です。その他のメモリを使用しなければならない場合は、可能な限り上記の推奨事項に従ってください。

3.13.4 高速フーリエ変換

(1) 関数一覧

表 3.29 DSP ライブラリ関数一覧(高速フーリエ変換)

項番	項目	関数名	説明
1	not-in-place 複素数 FFT	FftComplex	not-in-place 複素数 FFT を実行します。
2	not-in-place 実数 FFT	FftReal	not-in-place 実数 FFT を実行します。
3	not-in-place 複素数逆 FFT	IfftComplex	not-in-place 複素数逆 FFT を実行します。
4	not-in-place 実数逆 FFT	IfftReal	not-in-place 実数逆 FFT を実行します。
5	in-place 複素数 FFT	FftInComplex	in-place 複素数 FFT を実行します。
6	in-place 実数 FFT	FftInReal	in-place 実数 FFT を実行します。
7	in-place 複素数逆 FFT	IfftInComplex	in-place 複素数逆 FFT を実行します。
8	in-place 実数逆 FFT	IfftInReal	in-place 実数逆 FFT を実行します。
9	対数絶対値	LogMagnitude	複素数データを対数絶対値に変換します。
10	FFT 回転係数生成	InitFft	FFT 回転係数を生成します。
11	FFT 回転係数解放	FreeFft	FFT 回転係数の格納に使用したメモリを解放します。

【注】 not-in-place、in-place については「(5) FFT 構造」を参照してください。

これらの関数は、ユーザが定義したスケーリングを使って、順方向高速フーリエ変換と逆方向高速フーリエ変換を実行します。

順方向フーリエ変換は以下の式で定義されます。

$$y_n = 2^{-s}\sum_{n=0}^{N} e^{-2j\pi n/N} \cdot x_n$$

ここで、s はスケーリングが行われるステージの数、N はデータ数を示しています。

逆方向フーリエ変換は以下の式で定義されます。

$$y_n = 2^{-s}\sum_{n=0}^{N} e^{2j\pi n/N} \cdot x_n$$

スケーリングについては「(4) **スケーリング**」を参照してください。

(2) 複素数データ配列フォーマット

FFT および IFFT の複素数データ配列は、実数を X メモリに、虚数を Y メモリに配置します。ただし、実数 FFT の出力データと実数 IFFT の入力データの配置は異なっています。実数、虚数を格納する配列をそれぞれ x,y とすると、x[0]には DC 成分の実数成分が入り、y[0]には DC 成分の虚数成分ではなく Fs/2 成分の実数成分が入ります(DC 成分と Fs/2 成分はどちらも実数で、虚数成分は 0 です)。

(3) 実数データ配列フォーマット

FFT および IFFT の実数データ配列フォーマットには、以下の 3 種類があります。

単一の配列に格納し、任意のメモリブロックに配置。

単一の配列に格納し、Xメモリに配置。

2つの配列に分けて格納。それぞれの配列のサイズはN/2で、配列の前半はXメモリに配置し、後半はYメモリに配置。

FftReal は 1 番目の指定方法のみです。IfftReal、FftInReal および IfftInReal は 2 番目か 3 番目の方法をユーザが選択します。

(4) スケーリング

基数 2 の FFT は各ステージで信号強度が倍になり、ピーク信号振幅も倍になります。そのため、高強度信号を変換する際にオーバフローが発生することがありますが、各ステージで信号を 1/2 にすることにより(これをスケーリングといいます)オーバフローを防ぐことができます。しかし、スケーリングしすぎると不要な量子化雑音が発生する可能性があります。

オーバフローや量子化雑音とスケーリングの最適なバランスは入力信号の特性に大きく依存します。スペクトルが大きなピークを持つ信号はオーバフローを防ぐために最大スケーリングが必要になりますが､インパルス信号ではスケーリングの必要はほとんどありません。

すべてのステージでスケーリングするのが最も安全な方法です。入力データが強度 2^{30} 未満であれば、この方法でオーバフローを防ぐことができます。本ライブラリでは、各ステージごとにスケーリングを行うかどうかを指定できます。したがって､スケーリング指定を精密に行うことによって､オーバフローと量子化雑音の影響を最小限に抑えることができます。

スケーリングの方法を指定するために、各 FFT 関数の引数に scale が含まれています。scale は最下位ビットから 1 ビットずつが各ステージに対応しています。対応する scale のビットが 1 に設定されているすべてのステージで、2 の除算を実行します。

本ライブラリは実行速度を上げるために基数 4 の FFT を使用しています。scale は最下位ビットから 2 ビットずつが各ステージに対応しています。どちらか 1 ビットが 1 に設定されていれば、2 の除算を実行します。両方が 1 に設定されていれば 4 の除算を実行します。つまり、2 つの基数 2 の FFT ステージが 1 つの基数 4 の FFT ステージに置き換えられたのと同じことになります。しかし、基数 2 の FFT よりも基数 4 の FFT の方が量子化雑音の発生する可能性があります。

以下に scale の例を示します。

scale = H’FFFFFFFF(またはsize-1)はすべての基数2のFFTステージでスケーリングを行います。すべての入力データの強度が2^{30}未満であれば、オーバフローは発生しません。

scale = H’55555555は1つおきの基数2のFFTステージでスケーリングを行います。

scale = 0はスケーリングを行いません。

これらの scale の値は、ensigdsp.h で EFFTALLSCALE(H’FFFFFFFF)、EFFTMIDSCALE(H’55555555)、EFFTNOSCALE(0)と定義されています。

(5) FFT 構造

本ライブラリの FFT 構造には not-in-place FFT と in-place FFT の 2 種類があります。

not-in-place FFT では、入力データを RAM から取り出し、FFT を実行し、出力結果を RAM のユーザが指定した別の場所に格納します。

一方 in-place FFT では、入力データを RAM から取り出し、FFT を実行し、出力結果を RAM の同じ場所に格納します。この方法を用いると FFT の実行時間は増加しますが、使用メモリスペースが削減できます。

入力データを FFT 関数のほかにも使用する場合は、not-in-place FFT を使用してください。また、メモリスペースを節約したい場合は、in-place FFT を使用してください。

(6) 各関数の説明

(a) not-in-place 複素数 FFT

■説明

【書式】
```
int FftComplex (short op_x[], short op_y[], const short  ip_x[],
                const short ip_y[], long size, long scale)
```

【引数】

op_x[]	出力データの実数成分
op_y[]	出力データの虚数成分
ip_x[]	入力データの実数成分
ip_y[]	入力データの虚数成分
size	FFT のサイズ
scale	スケーリング指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 4

・size が 2 の累乗ではありません
・size > max_fft_size

【内容】　複素数高速フーリエ変換を実行します。

【備考】　本関数は not-in-place で行いますので、入力配列と出力配列を別々に用意してください。
複素数データ配列の配置については「(2) **複素数データ配列フォーマット**」を参照してください。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(4) **スケーリング**」を参照してください。
scale は下位 $\log_2$(size)ビットを使用します。
本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
```

インクルードヘッダ

```
#define MAX_FFT_SAMP  64
#define MIN_CFFT_SIZE   4
long ip_scale=0xffffffff;
long size = MIN_CFFT_SIZE;


#pragma section X
short ip_x[MAX_FFT_SAMP];
short op_x[MAX_FFT_SAMP];
#pragma section Y
short ip_y[MAX_FFT_SAMP];
short op_y[MAX_FFT_SAMP];
#pragma section

/* サイクル数測定用データ */
#define TWOPI 6.283185307 /* data */

void main()
{
    int i,j;
    long n_samp;

    n_samp=MAX_FFT_SAMP; /* data */

    for (j = 0; j < n_samp; j++){
        ip_x[j] = cos(j * TWOPI/n_samp) * 8188;
        ip_y[j] = sin(j * TWOPI/n_samp) * 8188;
    }
    if(InitFft(n_samp) != EDSP_OK){
        printf("Initfft != err end");
    }
    if(FftComplex(op_x,op_y,ip_x,ip_y,n_samp,EFFTALLSCALE) != EDSP_OK){
        printf("FftComplex error¥n");
    }
    FreeFft();

    for(i=0;i<n_samp;i++){
        printf("[%d] op_x=%d  op_y=%d  ¥n",i,op_x[i],op_y[i]);
    }
}
```

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義します。

FFT用データ作成

FFTの初期化関数
データ数で初期化を行います。必須です。
データ数はFFTのデータサイズと同じで2のべき乗であることが必要です。

FFT計算に使用したテーブルのfreeを行います。
これを行わないとメモリを無駄に使用してしまいます。
次に同じデータ数でFFTを行うのであれば、FreeFftを行わずにそのままFFTの関数を使用します。

(b) not-in-place 実数 FFT

■説明

【書式】
```
int FftReal (short op_x[], short op_y[], const short ip[],
             long size, long scale)
```

【引数】

op_x[]	正の出力データの実数成分
op_y[]	正の出力データの虚数成分
ip[]	実数入力データ
size	FFT のサイズ
scale	スケーリング指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 8 ・size が 2 の累乗ではありません ・size > max_fft_size

【内容】 実数高速フーリエ変換を実行します。

【備考】 op_x と op_y には size/2 の正の出力データが格納されます。負の出力データは正の出力データの共役複素数です。また、0 と $F_s/2$ での出力データの値は実数なので、$F_s/2$ での実数出力は op_y[0]に格納されます。
本関数は not-in-place で行いますので、入力配列と出力配列を別々に用意してください。
複素数と実数データ配列の配置については「(2) **複素数データ配列フォーマット**」「(3) **実数データ配列フォーマット**」を参照してください。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(4) **スケーリング**」を参照してください。
scale は下位 log_2(size)ビットを使用します。
本関数はリエントラントではありません。

RENESAS

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define   VLEN  64
#define   TWOPI 6.28318530717959

/* global data declarations */
#pragma section X
short output_x[VLEN];
#pragma section Y
short output_y[VLEN];
#pragma section

void main()
{
    short i;
    int k;
    short input[VLEN];
    short output[VLEN];

/* generate two sinusoids */
    k = VLEN / 8;
    for (i = 0; i < VLEN; i++)
       input[i] = floor(16383 * cos(TWOPI * k * i / VLEN) + 0.5);
    k = VLEN * 3 / 8;
    for (i = 0; i < VLEN; i++)
        input[i] += floor(16383 * cos(TWOPI * k * i / VLEN) + 0.5);

/* do FFT */
    if (InitFft(VLEN) != EDSP_OK)
        printf("InitFft problem¥n");
    if (FftReal(output_x, output_y, input, VLEN, EFFTALLSCALE) != EDSP_OK)
        printf("FftReal problem¥n");
    FreeFft();
}
```

インクルードヘッダ

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義します。

FFT用データ作成部

FFTの初期化関数
データ数で初期化を行います。必須です。データ数はFFTのデータサイズと同じで2のべき乗であることが必要です。

FFT計算に使用したテーブルのfreeを行います。
これを行わないとメモリを無駄に使用してしまいます。
次に同じデータ数でFFTを行うのであれば、FreeFftを行わずにそのままFFTの関数を使用します。

(c) not-in-place **複素数逆** FFT

■説明

【書式】

```
int IfftComplex (short op_x[], short op_y[], const short ip_x[],
                 const short ip_y[], long size, long scale)
```

【引数】

op_x[]	出力データの実数成分
op_y[]	出力データの虚数成分
ip_x[]	入力データの実数成分
ip_y[]	入力データの虚数成分
size	逆 FFT のサイズ
scale	スケーリング指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 4 ・size が 2 の累乗ではありません ・size > max_fft_size

【内容】 複素数逆高速フーリエ変換を実行します。

【備考】 本関数は not-in-place で行いますので、入力配列と出力配列を別々に用意してください。
複素数データ配列の配置については「(2) **複素数データ配列フォーマット**」を参照してください。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(4) **スケーリング**」を参照してください。
scale は下位 $\log_2$(size) ビットを使用します。
本関数はリエントラントではありません。

RENESAS

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */

long ip_scale=8188;

#pragma section X
short ipi_x[MAX_IFFT_SIZE];          /* input array */
short opi_x[MAX_IFFT_SIZE];          /* normal output array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
short opi_y[MAX_IFFT_SIZE];
#pragma section

void main()
{
    int i,j;
    long scale;
    long max_size;

    max_size=MAX_IFFT_SIZE;/* data */

    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
        ipi_y[j] = sin(j * TWOPI/max_size) * ip_scale;
    }
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error end  ¥n");
    }
    else {
        if(FftInComplex(ipi_x, ipi_y, max_size,EFFTALLSCALE) != EDSP_OK){
            printf("FftInComplex err  end  ¥n");
        }
        for (j = 0; j < max_size; j++){
            opi_x[j]=0;
            opi_y[j]=0;
        }
        if(IfftComplex(opi_x, opi_y, ipi_x, ipi_y, max_size,
                       EFFTALLSCALE)!= EDSP_OK){
            printf("IfftComplex err  end  ¥n");
        }
        for (j = 0; j < max_size; j++){
           printf("[%d]  opi_x=%d  op_y=%d ¥n",j, opi_x[j],opi_y[j]);
        }
        FreeFft();
    }
}
```

インクルードヘッダ

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義します。

FFT用データ作成部（FftComplexを実行するためのデータです。）

FFTの初期化関数
データ数で初期化を行います。必須です。データ数はFFTのデータサイズと同じで2のべき乗であることが必要です。

いったんFFT計算を行ってその結果を逆FFT関数の入力値とするための処理なので通常は不要です。

FFT計算に使用したテーブルのfreeを行います。これを行わないとメモリを無駄に使用してしまいます。
次に同じデータ数でFFTを行うのであれば、FreeFftを行わずにそのままFFTの関数を使用します。

(d) not-in-place **実数逆** FFT

■説明

【書式】
```
int IfftReal (short op_x[], short scratch_y[], const short ip_x[],
              const short ip_y[], long size, long scale,
              int op_all_x)
```

【引数】

op_x[]	実数出力データ
scratch_y[]	スクラッチメモリまたは実数出力データ
ip_x[]	正の入力データの実数成分
ip_y[]	正の入力データの虚数成分
size	逆 FFT のサイズ
scale	スケーリング指定
op_all_x	出力データの配置指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 8 ・size が 2 の累乗ではありません ・size > max_fft_size ・op_all_x ≠ 0 または 1

【内容】 実数逆高速フーリエ変換を実行します。

【備考】 ip_x と ip_y には size/2 の正の入力データを格納してください。負の入力データは正の入力データの共役複素数です。また、0 と $F_s/2$ での入力データの値は実数なので、$F_s/2$ での実数入力は ip_y[0] に格納してください。
出力データのフォーマットは op_all_x で指定します。op_all_x=1 の場合、全出力データは op_x に格納されます。op_all_x=0 の場合、最初の size/2 の出力データは op_x に格納され、残りの size/2 の出力データは scratch_y に格納されます。
本関数は not-in-place で行いますので、入力配列と出力配列を別々に用意してください。
複素数と実数データ配列の配置については「(2) **複素数データ配列フォーマット**」「(3) **実数データ配列フォーマット**」を参照してください。
ip_x、ip_y はそれぞれ size/2 のデータを格納してください。op_x は op_all_x の値によって、size または size/2 のデータが格納されます。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(4) **スケーリング**」を参照してください。
scale は下位 $\log_2$(size) ビットを使用します。
本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */

long ip_scale=8188;

#pragma section X
short ipi_x[MAX_IFFT_SIZE];          /* input array */
short opi_x[MAX_IFFT_SIZE];          /* normal output array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
short opi_y[MAX_IFFT_SIZE];
#pragma section

void main()
{
    int i,j;
    long scale;
    long max_size;

    max_size=MAX_IFFT_SIZE;/* data */

    for (j = 0; j < max_size; j++){
       ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
    }

    if (InitFft(max_size) != EDSP_OK){
       printf("InitFft error end  ¥n");
    }
    else {
       if(FftInReal(ipi_x, ipi_y, max_size,EFFTALLSCALE,1) != EDSP_OK){
          printf("FftInReal err  end  ¥n");
       }

if(IfftReal(opi_x, opi_y, ipi_x, ipi_y, max_size, EFFTALLSCALE,1)!=
   EDSP_OK){
          printf("IfftReal err end  ¥n");
       }
       for (j = 0; j < max_size; j++){
         printf("[%d]  opi_x=%d  op_y=%d ¥n",j, opi_x[j],opi_y[j]);
       }

       FreeFft();
    }
}
```

インクルードヘッダ

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義します。

FFT用データ作成部（FftRealを計算するためのデータです。）

FFTの初期化関数
データ数で初期化を行います。必須です。データ数はFFTのデータサイズと同じで2のべき乗であること。逆FFTにも必要です。

FFT計算に使用したテーブルのfreeを行います。これを行わないとメモリを無駄に使用してしまいます。
次に同じデータ数でFFTを行うのであれば、FreeFftを行わずにそのままFFTの関数を使用します。

いったんFFT計算を行ってその結果を逆FFT関数の入力値とするための処理なので通常は不要です。

(e) in-place **複素数** FFT

■説明

【書式】
```
int FftInComplex (short data_x[], short data_y[], long size,
                  long scale)
```

【引数】

data_x[]	入出力データの実数成分
data_y[]	入出力データの虚数成分
size	FFT のサイズ
scale	スケーリング指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 4 ・size が 2 の累乗ではありません ・size > max_fft_size

【内容】 in-place 複素数高速フーリエ変換を実行します。

【備考】 複素数データ配列の配置については「(2) **複素数データ配列フォーマット**」を参照してください。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(4) **スケーリング**」を参照してください。
scale は下位 $\log_2$(size) ビットを使用します。
本関数はリエントラントではありません。

RENESAS

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>

#define MAX_FFT_SAMP  64
#define TWOPI 6.283185307 /* data */
long ip_scale=0xffffffff;

#pragma section X
short ip_x[MAX_FFT_SAMP];
#pragma section Y
short ip_y[MAX_FFT_SAMP];
#pragma section

void main()
{
    int i,j;
    long max_size;
    long n_samp;

    n_samp=MAX_FFT_SAMP;
    max_size=n_samp;/* data */

    for (j = 0; j < n_samp; j++){
        ip_x[j] = cos(j * TWOPI/n_samp) * ip_scale;
        ip_y[j] = sin(j * TWOPI/n_samp) * ip_scale;
    }
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error¥n");
    }
    if(FftInComplex(ip_x, ip_y, n_samp,EFFTALLSCALE ) != EDSP_OK){
        printf("FftInComplex error¥n");
    }
    FreeFft();

    for(i=0;i<max_size;i++){
        printf("[%d] ip_x=%d  ip_y=%d  ¥n",i,ip_x[i],ip_y[i]);
    }
}
```

インクルードヘッダ

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義します。

FFT用データ作成部

FFTの初期化関数
データ数で初期化を行います。必須です。データ数はFFTのデータサイズと同じで2のべき乗でなければなりません。

FFT計算に使用したテーブルのfreeを行います。
これを行わないとメモリを無駄に使用してしまいます。
次に同じデータ数でFFTを行うのであれば、FreeFftを行わずにそのままFFTの関数を使用します。

(f) in-place **実数** FFT

■説明

【書式】 `int FftInReal (short data_x[], short data_y[], long size, long scale, int ip_all_x)`

【引数】

data_x[]	入力時は実数データ、出力時は正の出力データの実数成分
data_y[]	入力時は実数データまたは未使用、出力時は正の出力データの虚数成分
size	FFT のサイズ
scale	スケーリング指定
ip_all_x	入力データの配置指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 8 ・size が 2 の累乗ではありません ・size > max_fft_size ・ip_all_x ≠ 0 または 1

【内容】 in-place 実数高速フーリエ変換を実行します。

【備考】 入力データのフォーマットは、ip_all_x で指定します。ip_all_x=1 の場合、全入力データは data_x から取り出します。ip_all_x=0 の場合、前半の size/2 の入力データは data_x から、後半の size/2 の入力データは data_y から取り出します。
本関数実行後、data_x と data_y には size/2 の正の出力データが格納されます。負の出力データは正の出力データの共役複素数です。また、0 と $F_S/2$ での出力データの値は実数なので、$F_S/2$ での実数出力は data_y[0]に格納されます。
複素数と実数データ配列の配置については「(2) **複素数データ配列フォーマット**」「(3) **実数データ配列フォーマット**」を参照してください。
data_y は size/2 のデータを格納します。data_x は ip_all_x の値によって size または size/2 のデータを格納します。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(4) **スケーリング**」を参照してください。
scale は下位 $\log_2$(size) ビットを使用します。
本関数はリエントラントではありません。

RENESAS

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>

#define MAX_FFT_SAMP  64
#define TWOPI 6.283185307 /* data */

long ip_scale=8188;
/*long ip_scale=0xffffffff;*/

#pragma section X
short ip_x[MAX_FFT_SAMP];
#pragma section Y
short ip_y[MAX_FFT_SAMP];
#pragma section


void main()
{
    int i,j;
    long max_size;
    long n_samp;
    int ip_all_x;

    n_samp=MAX_FFT_SAMP;
    max_size=n_samp;/* data */

    for (j = 0; j < n_samp; j++){
        ip_x[j] = cos(j * TWOPI/n_samp) * ip_scale;
        ip_y[j] = 0;
    }
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error¥n");
    }

    ip_all_x = 1;
    if(FftInReal(ip_x, ip_y, n_samp,EFFTALLSCALE ,ip_all_x) != EDSP_OK){
        printf("FftInReal error¥n");
    }

    FreeFft();

    for(i=0;i<max_size;i++){
        printf("[%d] ip_x=%d  ip_y=%d  ¥n",i,ip_x[i],ip_y[i]);
    }
}
```

インクルードヘッダ

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義します。

FFT用データ作成部

FFTの初期化関数
データ数で初期化を行います。必須です。
データ数はFFTのデータサイズと同じで2のべき乗である必要があります。

FFT計算に使用したテーブルのfreeを行います。
これを行わないとメモリを無駄に使用してしまいます。
次に同じデータ数でFFTを行うのであれば、FreeFftを行わずにそのままFFTの関数を使用します。

(g) in-place **複素数逆** FFT

■説明

【書式】

```
int IfftInComplex (short data_x[], short data_y[], long size,
                   long scale)
```

【引数】

data_x[]　　入出力データの実数成分
data_y[]　　入出力データの虚数成分
size　　逆 FFT のサイズ
scale　　スケーリング指定

【戻り値】

EDSP_OK　　成功
EDSP_BAD_ARG　　以下のいずれかの場合です
・size < 4
・size が 2 の累乗ではありません
・size > max_fft_size

【内容】

in-place 複素数逆高速フーリエ変換を実行します。

【備考】

複素数データ配列の配置については「(2) 複素数データ配列フォーマット」を参照してください。
本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。
スケーリング指定については「(4) **スケーリング**」を参照してください。
scale は下位 $\log_2$(size)ビットを使用します。
本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */

long ip_scale=8188;

#pragma section X
short ipi_x[MAX_IFFT_SIZE];          /* input array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
#pragma section

void main()
{
    int i,j;
    long scale;
    long max_size;

    max_size=MAX_IFFT_SIZE;/* data */

    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
        ipi_y[j] = sin(j * TWOPI/max_size) * ip_scale;
    }

    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error end  ¥n");
    }
    else {
        if(FftInComplex(ipi_x, ipi_y, max_size,EFFTALLSCALE) != EDSP_OK){
            printf("FftInComplex err  end  ¥n");
        }

        if(IfftInComplex(ipi_x, ipi_y, max_size,EFFTALLSCALE) != EDSP_OK){
            printf("IfftInComplex err  end  ¥n");
        }
        for (j = 0; j < max_size; j++){
           printf("[%d]  ipi_x=%d  ip_y=%d ¥n",j, ipi_x[j],ipi_y[j]);
        }

        FreeFft();
    }
}
```

インクルードヘッダ

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義します。

FFT用データ作成部（FftInComplexの入力値となるデータ）

FFTの初期化関数
データ数で初期化を行います。必須です。データ数はFFTのデータサイズと同じで2のべき乗でなければなりません。逆FFTにも必要です。

いったんFFT計算を行ってその結果を逆FFT関数の入力値とするための処理なので通常は不要です。

FFT計算に使用したテーブルのfreeを行います。これを行わないとメモリを無駄に使用してしまいます。
次に同じデータ数でFFTを行うのであれば、FreeFftを行わずにそのままFFTの関数を使用します。

(h) in-place 実数逆 FFT

■説明

【書式】

```
int IfftInReal (short data_x[], short data_y[],
                long size, long scale, int op_all_x)
```

【引数】

data_x[]	入力時は正の入力データの実数成分、出力時は実数データ
data_y[]	入力時は正の入力データの虚数成分、出力時は実数データまたは未使用
size	逆 FFT のサイズ
scale	スケーリング指定
op_all_x	出力データの配置指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 8 ・size が 2 の累乗ではありません ・size > max_fft_size ・op_all_x ≠ 0 または 1

【内容】 in-place 実数逆高速フーリエ変換を実行します。

【備考】

data_x と data_y には size/2 の正の入力データを格納してください。負の入力データは正の入力データの共役複素数です。また、0 と $F_s/2$ での入力データの値は実数なので、$F_s/2$ での実数入力は data_y[0] に格納してください。

出力データのフォーマットは op_all_x で指定します。op_all_x=1 の場合、全出力データは data_x に格納されます。op_all_x=0 の場合、前半の size/2 の出力データは data_x に格納され、後半の size/2 の出力データは data_y に格納されます。

複素数と実数データ配列の配置については**「(2) 複素数データ配列フォーマット」「(3) 実数データ配列フォーマット」**を参照してください。

data_y は size/2 のデータを格納します。data_x は、op_all_x の値によって size または size/2 のデータが格納されます。

本関数を呼び出す前に InitFft を呼び出して、回転係数と max_fft_size を初期化してください。

スケーリング指定については**「(4) スケーリング」**を参照してください。

scale は下位 $\log_2$(size) ビットを使用します。

本関数はリエントラントではありません。

RENESAS

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */

long ip_scale=8188;

#pragma section X
short ipi_x[MAX_IFFT_SIZE];         /* input array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
#pragma section

void main()
{
   int i,j;
   long scale;
   long max_size;

   max_size=MAX_IFFT_SIZE;/* data */

   for (j = 0; j < max_size; j++){
      ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
   }

   if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error end  ¥n");
   }
   else {
      if(FftInReal(ipi_x, ipi_y, max_size,EFFTALLSCALE,1) != EDSP_OK){
         printf("FftInReal err  end  ¥n");
      }
      if(IfftInReal(ipi_x, ipi_y, max_size, EFFTALLSCALE,1) != EDSP_OK){
         printf("IfftInReal err  end  ¥n");
      }
      for (j = 0; j < max_size; j++){
        printf("[%d]  ipi_x=%d  ip_y=%d ¥n",j, ipi_x[j],ipi_y[j]);
      }

      FreeFft();
   }
}
```

(i) 対数絶対値

■説明

【書式】
```
int LogMagnitude (short output[], const short ip_x[],
                  const short ip_y[], long no_elements,
                  float fscale)
```

【引数】

output[]	実数出力 z
ip_x[]	入力の実数成分 x
ip_y[]	入力の虚数成分 y
no_elements	出力データ数 N
fscale	出力スケーリング係数

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・no_elements > 32767 ・$|fscale| \geq 2^{15}/(10\log_{10}2^{31})$

【内容】 複素数入力データの対数絶対値をデシベル単位で計算し、スケーリング結果を出力配列に書き込みます。

【備考】 $z(n)=10 fscale \cdot \log_{10}(x(n)^2+y(n)^2) \quad 0 \leq n < N$
複素数データ配列の配置については「(2) **複素数データ配列フォーマット**」を参照してください。

■使用例

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */

long ip_scale=8188;

#pragma section X
short ipi_x[MAX_IFFT_SIZE];          /* input array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
#pragma section

void main()
{
    int i,j;
    long scale;
    long max_size;
    short output[MAX_IFFT_SIZE];

    max_size=MAX_IFFT_SIZE;/* data */

    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
    }

    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error end  ¥n");
    }
    else {
        if(FftInReal(ipi_x, ipi_y, max_size,EFFTALLSCALE,1) != EDSP_OK){
            printf("FftInReal err  end  ¥n");
        }
        if(LogMagnitude(output, ipi_x,ipi_y, max_size/2, 2) != EDSP_OK){
            printf("LogMagnitude err  end  ¥n");
        }
        for (j = 0; j < max_size/2; j++){
          printf("[%d]  output=%d  ¥n",j, output[j]);
        }

        FreeFft();
    }

}
```

インクルードヘッダ

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義します。

FFT用データ作成部

FFTの関数
LogMagnitude関数で使用するデータを作成します。

FFT計算に使用したテーブルのfreeを行います。
これを行わないとメモリを無駄に使用してしまいます。次に同じデータ数でFFTを行うのであれば、FreeFftを行わずにそのままFFTの関数を使用します。
LogMagnitudeには直接関係ありません。

(j) 回転係数生成

■説明

【書式】 `int InitFft (long max_size)`

【引数】 max_size 必要になる FFT の最大サイズ

【戻り値】
EDSP_OK 成功
EDSP_NO_HEAP malloc で確保できるメモリスペースが不十分
EDSP_BAD_ARG 以下のいずれかの場合です
・max_size < 2
・max_size が 2 の累乗ではありません
・max_size > 32768

【内容】 FFT 関数で使用する回転係数（1/4 サイズ）を生成します。

【備考】 回転係数は malloc によって確保されるメモリに格納されます。
回転係数が生成されると max_fft_size グローバル変数が更新されます。max_fft_size は FFT の最大許容サイズを示します。
本関数は最初の FFT 関数を呼び出す前に必ず一度呼び出してください。
max_size は 8 以上としてください。
回転係数は max_size で指定した変換サイズで生成されます。max_size より小さいサイズの FFT 関数を実行したときも同じ回転係数を使用します。
回転係数のアドレスは内部変数内に格納されています。ここはユーザプログラムでアクセスしないでください。
本関数はリエントラントではありません。

(k) 回転係数開放

■説明

【書式】 `void FreeFft (void)`

【引数】 なし

【戻り値】 なし

【内容】 回転係数の格納に使用したメモリを解放します。

【備考】 max_fft_size グローバル変数を 0 にします。FreeFft を実行した後再び FFT 関数を実行するときには、その前に必ず InitFft を実行してください。
本関数はリエントラントではありません。

3.13.5 窓関数

(1) 関数一覧

表 3.30 DSP ライブラリ関数一覧(窓関数)

項番	項目	関数名	説明
1	ブラックマン窓	GenBlackman	ブラックマン窓を生成します。
2	ハミング窓	GenHamming	ハミング窓を生成します。
3	ハニング窓	GenHanning	ハニング窓を生成します。
4	三角窓	GenTriangle	三角窓を生成します。

(2) 各関数の説明

(a) ブラックマン窓

■説明

【書式】 `int GenBlackman (short output[], long win_size)`

【引数】
output[] 出力データ W(n)
win_size 窓サイズ N

【戻り値】
EDSP_OK 成功
EDSP_BAD_ARG win_size ≤ 1

【内容】 ブラックマン窓を生成し、output に出力します。

【備考】 実際のデータにこの窓をかけるときは VectorMult を使用します。
使用する関数を以下に示します。

$$W(n) = \left(2^{15} - 1\right)\left[0.42 - 0.5\cos\left(\frac{2\pi n}{N}\right) + 0.08\cos\left(\frac{4\pi n}{N}\right)\right] \qquad 0 \le n < N$$

■使用例

```
#include <stdio.h>      } インクルードヘッダ
#include <ensigdsp.h>
#define MAXN 10

void main()
{
    int i;
    long len;
    short output[MAXN];

    len=MAXN ;
    if(GenBlackman(output, len) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<len;i++){
        printf("output=%d ¥n",output[i]);
    }
}
```

(b) ハミング窓

■説明

【書式】 int GenHamming (short output[], long win_size)

【引数】
output[] 出力データ W(n)
win_size 窓サイズ N

【戻り値】
EDSP_OK 成功
EDSP_BAD_ARG win_size ≤ 1

【内容】 ハミング窓を生成し、output に出力します。

【備考】 実際のデータにこの窓をかけるときは VectorMult を使用します。
使用する関数を以下に示します。

$$W(n) = \left(2^{15} - 1\right)\left[0.54 - 0.46\cos\left(\frac{2\pi n}{N}\right)\right] \quad 0 \le n < N$$

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#define MAXN  10

void main()
{

   int i;
   long len;
   short output[MAXN];

   len=MAXN ;
   if(GenHamming(output, len) != EDSP_OK){
      printf("EDSP_OK not returned¥n");
   }
   for(i=0;i<len;i++){
      printf("output=%d ¥n",output[i]);
   }
}
```

インクルードヘッダ

(c) ハニング窓

■説明

【書式】 int GenHanning (short output[], long win_size)

【引数】
output[] 出力データ W(n)
win_size 窓サイズ N

【戻り値】
EDSP_OK 成功
EDSP_BAD_ARG win_size ≤ 1

【内容】 ハニング窓を生成し、output に出力します。

【備考】 実際のデータにこの窓をかけるときは VectorMult を使用します。
使用する関数を以下に示します。

$$W(n) = \left(\frac{2^{15} - 1}{2}\right)\left[1 - \cos\left(\frac{2\pi n}{N}\right)\right] \quad 0 \le n < N$$

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define MAXN  10

void main()
{
    int i;
    long len;
    short output[MAXN];

    len=MAXN ;
    if(GenHanning(output, len) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<len;i++){
        printf("output=%d ¥n",output[i]);
    }

}
```

インクルードヘッダ

(d) 三角窓

■説明

【書式】 `int GenTriangle (short output[], long win_size)`

【引数】
output[]　　出力データ W(n)
win_size　　窓サイズ N

【戻り値】
EDSP_OK　　成功
EDSP_BAD_ARG　　win_size ≤ 1

【内容】 三角窓を生成し、output に出力します。

【備考】 実際のデータにこの窓をかけるときは VectorMult を使用します。
使用する関数を以下に示します。

$$W(n) = \left(2^{15} - 1\right)\left[1 - \left|\frac{2n - N + 1}{N + 1}\right|\right] \qquad 0 \le n < N$$

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define MAXN 10
void main()
{
    int i;
    long len;
    short output[MAXN];

    len=MAXN ;
    if(GenTriangle(output, len) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<len;i++){
        printf("output=%d ¥n",output[i]);
    }
}
```

インクルードヘッダ

3.13.6 フィルタ

(1) 関数一覧

表 3.31 DSP ライブラリ関数一覧(フィルタ)

項番	項目	関数名	説明
1	FIR	Fir	有限インパルス応答フィルタ処理を実行します。
2	単一データ用 FIR	Fir1	単一データ用有限インパルス応答フィルタ処理を実行します。
3	IIR	Iir	無限インパルス応答フィルタ処理を実行します。
4	単一データ用 IIR	Iir1	単一データ用無限インパルス応答フィルタ処理を実行します。
5	倍精度 IIR	Diir	倍精度無限インパルス応答フィルタ処理を実行します。
6	単一データ用倍精度 IIR	Diir1	単一データ用倍精度無限インパルス応答フィルタ処理を実行します。
7	適応 FIR	Lms	適応 FIR フィルタ処理を実行します。
8	単一データ用適応 FIR	Lms1	単一データ用適応 FIR フィルタ処理を実行します。
9	FIR 作業領域割り付け	InitFir	FIR フィルタ用に作業領域を割り付けます。
１０	IIR 作業領域割り付け	InitIir	IIR フィルタ用に作業領域を割り付けます。
１１	倍精度 IIR 作業領域割り付け	InitDIir	DIIR フィルタ用に作業領域を割り付けます。
１２	適応 FIR 作業領域割り付け	InitLms	LMS フィルタ用に作業領域を割り付けます。
１３	FIR 作業領域解放	FreeFir	InitFir で割り付けられた作業領域を解放します。
１４	IIR 作業領域解放	FreeIir	InitIir で割り付けられた作業領域を解放します。
１５	倍精度 IIR 作業領域解放	FreeDIir	InitDIir で割り付けられた作業領域を解放します。
１６	適応 FIR 作業領域解放	FreeLms	InitLms で割り付けられた作業領域を解放します。

【注】 本関数のいずれかを使用する場合、ユーザプログラム内で 1 回のみ filt_ws.h をインクルードしてください。

(2) 係数のスケーリング

フィルタ処理を行うと飽和、または量子化雑音が発生する可能性があります。これらはフィルタ係数のスケーリングを行うことによって最小限に抑えることができます。しかし、飽和と量子化雑音の影響をよく考えてスケーリングを行わなければなりません。係数が大きすぎると飽和が、小さすぎると量子化雑音が発生する可能性があります。

FIR(有限インパルス応答)フィルタの場合、以下の式が成り立つようにフィルタ係数を設定すれば飽和は起こりません。

coeff[i] ≠ H’8000 (すべての i について)

$\sum |coeff| < 2^{24}$

res_shift = 24

coeff はフィルタ係数、res_shift は出力で行われる右シフトのビット数です。

しかし、多くの入力信号の場合、もっと小さい res_shift の値(またはもっと大きな coeff の値)を使用しても飽和する可能性は少なく、量子化雑音も大幅に削減できます。また入力値に H’8000 が含まれている可能性があれば、すべての coeff の値は H’8001～H’7FFF の範囲になるように設定してください。

IIR(無限インパルス応答)フィルタは再帰的な構造になっています。そのため上述したようなスケーリング方法は適していません。

LMS(最小 2 乗平均)適応フィルタは FIR フィルタと同様です。しかし、係数を適応するときに飽和を引き起こす場合があります。その場合は、係数に H’8000 を含まないように設定してください。

(3) 作業領域

デジタルフィルタでは、ある処理から次の処理へ保持しておかなければならない情報があります。これらの情報は、最小オーバーヘッドでアクセスすることができるメモリに格納します。本ライブラリでは、Y-RAM 領域を作業領域として使用します。作業領域はフィルタ処理を実行する前に Init 関数を呼び出して初期化してください。

作業領域メモリはライブラリ関数によってアクセスされます。なお、ユーザプログラムから作業領域を直接アクセスしないでください。

(4) メモリの使用

SH-DSP を効率よく使うために、フィルタ係数は X メモリに配置してください。入出力データは任意のメモリセグメン

トに配置することができます。
　フィルタ係数は#pragma section 命令を用いて X メモリに配置してください。
　各フィルタは Init 関数を用いてグローバルバッファから作業領域を割り付けます。グローバルバッファは Y メモリに配置します。

(5) 各関数の説明

(a) FIR

■説明

【書式】
```
int Fir (short output[], const short input[], long no_samples,
         const short coeff[], long no_coeffs, int res_shift,
         short *workspace)
```

【引数】

output[]	出力データ y
input[]	入力データ x
no_samples	入力データの数 N
coeff[]	フィルタ係数 h
no_coeffs	係数の数(フィルタの長さ)K
res_shift	各出力に適用される右シフト
workspace	作業領域へのポインタ

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_samples < 1 ・no_coeffs ≤ 2 ・res_shift < 0 ・res_shift > 25

【内容】　有限インパルス応答(FIR)フィルタ処理を実行します。

【備考】　最新の入力データは作業領域に保持されます。input をフィルタ処理した結果は output に書き込まれます。

$$y(n) = \left[\sum_{k=0}^{K-1} h(k)\, x(n-k)\right] \cdot 2^{-res_shift}$$

積和演算の結果は 39 ビットで保持されます。出力 y(n)は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。
係数のスケーリングについては「(2) **係数のスケーリング**」を参照してください。
本関数を呼び出す前に InitFir を呼び出し、フィルタの作業領域を初期化してください。
output に input と同じ配列を指定した場合、input は上書きされます。
本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
```

インクルードヘッダ

```
#define NFN 8 /* number of functions */
#define FIL_COUNT   32  /* データ数 */
#define N   32

#pragma section X
static short coeff_x[FIL_COUNT];
#pragma section

short data[FIL_COUNT] =  {
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,};

short coeff[8] =  {
        0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000,};

void main()
{
   short *work, i;
   short output[N];
   int nsamp, ncoeff, rshift;

   /* copy coeffs into X RAM */
   for(i=0;i<NFN;i++) {
      coeff_x[i] = coeff[i];/* 係数設定 */
   }
   for (i = 0; i < N; output[i++] = 0) ;
   ncoeff = NFN;/* 係数の数設定 */
   nsamp = FIL_COUNT;/* サンプル数設定 */
   rshift = 12;

   if (InitFir(&work, ncoeff) != EDSP_OK){
      printf("Init Problem¥n");
   }
   if(Fir(output, data, nsamp, coeff_x, ncoeff, rshift, work) != EDSP_OK){
      printf("Fir Problem¥n");
   }
   if (FreeFir(&work, ncoeff) != EDSP_OK){
      printf("Free Problem¥n");
   }
   for(i=0;i<nsamp;i++){
      printf("#%2d output:%6d ¥n",i,output[i]);
   }

}
```

フィルタ係数をXメモリ上に設定します。
フィルタ計算のワークエリアとしてライブラリの内部で使用しているのでYメモリは使用しないでください。

フィルタ係数をXメモリ上の変数に設定します。

フィルタの初期化
(1)ワークエリアアドレス
(2)係数の数
Fir関数発行の前に必須です。
Yメモリをワークエリアとして（係数の数）*2+8バイト使用します。

Fir計算に使用したワークエリアの解放。
Fir使用後は必ず行います。
この関数を発行しないと、メモリが無駄に使用されます。

(b) 単一データ用 FIR

■説明

【書式】
```
int Fir1 (short *output, short input, const short coeff[],
          long no_coeffs, int res_shift, short *workspace)
```

【引数】

output	出力データ y(n)へのポインタ
input	入力データ x(n)
coeff[]	フィルタ係数 h
no_coeffs	係数の数(フィルタの長さ)K
res_shift	各出力に適用される右シフト
workspace	作業領域へのポインタ

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_coeffs ≤ 2 ・res_shift < 0 ・res_shift > 25

【内容】 単一データ用に有限インパルス応答(FIR)フィルタ処理を実行します。

【備考】 最新の入力データは作業領域に保持されます。input をフィルタ処理した結果は*output に書き込まれます。

$$y(n) = \left[\sum_{k=0}^{K-1} h(k)\, x(n-k)\right] \cdot 2^{-res_shift}$$

積和演算の結果は 39 ビットで保持されます。出力 y(n)は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。

係数のスケーリングについては「(2) **係数のスケーリング**」を参照してください。

関数を呼び出す前に InitFir を呼び出し、フィルタの作業領域を初期化してください。

本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>

#define NFN 8  /* number of functions */
#define MAXSH 25
#define N   32

#pragma section X
static short coeff_x[NFN];
#pragma section

short data[32] =  {
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400};

short coeff[8] =  {
        0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000};

void main()
{
    short *work, i;
    short output[N];
    int ncoeff, rshift;

    /* copy coeffs into X RAM */
    for(i=0;i<NFN;i++) {
        coeff_x[i] = coeff[i];/* 係数設定 */
    }
    for (i = 0; i < N; output[i++] = 0) ;
    rshift = 12;
    ncoeff = NFN;/* 係数の数設定 */

    if (InitFir(&work, NFN) != EDSP_OK){
        printf("Init Problem¥n");
    }
    for(i=0;i<N;i++) {
        if(Fir1(&output[i], data[i], coeff_x, ncoeff, rshift, work) !=
        EDSP_OK){
            printf("Fir1 Problem¥n");
        }
        printf(" output[%d]=%d ¥n",i,output[i]);
    }
    if (FreeFir(&work, NFN) != EDSP_OK){
        printf("Free Problem¥n");
    }
}
```

インクルードヘッダ（#include の3行）

フィルタ係数をXメモリ上に設定します。フィルタ計算のワークエリアとしてライブラリの内部で使用しているのでYメモリは使用しないでください。（static short coeff_x[NFN];）

フィルタ係数をXメモリ上の変数に設定します。（coeff_x[i] = coeff[i];）

フィルタの初期化
(1)**ワークエリアアドレス**
(2)**係数の数**
Fir1**関数発行の前に必須です。**Y**メモリをワークエリアとして（係数の数）***2+8**バイト使用します。**（InitFir）

Fir1**はデータ数**1**の**Fir**関数と同じです。**
Fir1**を複数回発行する場合には、**InitFir**と**FreeFir**は最初と最後の**1**回ずつ発行してください。**

(c) IIR

■説明

【書式】 `int Iir (short output[], const short input[], long no_samples, const short coeff[], long no_sections, short *workspace)`

【引数】

output[]	出力データ y_{K-1}
input[]	入力データ x_0
no_samples	入力データの数 N
coeff[]	フィルタ係数
no_sections	2 次フィルタセクションの数 K
workspace	作業領域へのポインタ

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_samples < 1 ・no_sections < 1 ・$a_{0k} < 0$ ・$a_{0k} > 16$

【内容】 無限インパルス応答(IIR)フィルタ処理を実行します。

【備考】 フィルタは、バイカッドという 2 次フィルタを K 個縦列に接続した構成になっています。各バイカッドの出力で付加的なスケーリングが行われます。フィルタ係数は符号付き 16 ビット固定小数点数で指定します。

各バイカッドの出力は以下の式で与えられます。

$d_k(n)=[a_{1k}d_k(n-1)+a_{2k}d_k(n-2)+2^{15}x(n)]\cdot 2^{-15}$

$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)]\cdot 2^{-a0k}$

k 番目のセクションの入力 $x_k(n)$は、前のセクションの出力 $y_{k-1}(n)$です。最初のセクション(k=0)の入力は input から読み込まれます。最後のセクション(k=K-1)の出力は output に書き込まれます。

coeff は係数を以下の順序に設定してください。

a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}

a_{0k} 項は k 番目のバイカッドの出力で行われる右シフトのビット数です。

各バイカッドでは飽和演算を 32 ビットで行います。各バイカッドの出力は 15 ビットまたは a_{0k} ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。

本関数を呼び出す前に InitIir を呼び出し、フィルタの作業領域を初期化してください。

output に input と同じ配列を指定した場合、input は上書きされます。

本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>

#define K  4
#define NUMCOEF  (6*K)
#define N 50

#pragma section X
static short coeff_x[NUMCOEF];
#pragma section

static short coeff[24] = {15, 19144, -7581,  5301, 10602,  5301,
                          15, -1724,-23247, 13627, 27254, 13627,
                          15, 19144, -7581,  5301, 10602,  5301,
                          15, -1724,-23247, 13627, 27254, 13627};


static short input[50] = {32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000 };

void main()
{
   short *work, i;
   short output[N];

   for(i=0;i<NUMCOEF;i++) {
      coeff_x[i] = coeff[i];
   }
   if (InitIir(&work, K) != EDSP_OK){
      printf("Init Problem¥n");
   }
   if (Iir(output, input, N, coeff_x, K, work) != EDSP_OK){
      printf("EDSP_OK not returned¥n");
   }
   if (FreeIir(&work, K) != EDSP_OK){
      printf("Free Problem¥n");
   }
   for(i=0;i<N;i++){
      printf("#%2d output:%6d ¥n",i,output[i]);
   }
}
```

インクルードヘッダ

フィルタ係数をXメモリ上に設定します。
フィルタ計算のワークエリアとしてライブラリの内部で使用しているのでYメモリは使用しないでください。

フィルタ係数は6個を1セクションとして設定してください、セクションの先頭要素は右シフト数でフィルタ係数ではありません。

フィルタ係数をXメモリ上の変数に設定します。

フィルタの初期化
(1)ワークエリアアドレス
(2)フィルタセクションの数
Iir関数発行の前に必須です。
Yメモリをワークエリアとして((フィルタセクションの数)*2*2)バイト使用します。

Iir計算に使用したワークエリアの解放。
Iir使用後は必ず行ってください。
この関数を発行しないと、メモリが無駄に使用されてしまいます。

(d) 単一データ用 IIR

■説明

【書式】
```
int Iir1 (short *output, short input, const short coeff[],
          long no_sections, short *workspace)
```

【引数】

output	出力データ y_{K-1}(n)へのポインタ
input	入力データ x_0(n)
coeff[]	フィルタ係数
no_sections	2 次フィルタセクションの数 K
workspace	作業領域へのポインタ

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_sections < 1 ・$a_{ok} < 0$ ・$a_{ok} > 16$

【内容】 単一データ用に無限インパルス応答(IIR)フィルタ処理を実行します。

【備考】 フィルタは、バイカッドという 2 次フィルタを K 個縦列に接続した構成になっています。各バイカッドの出力で付加的なスケーリングが行われます。フィルタ係数は符号付き 16 ビット固定小数点数で指定します。

各バイカッドの出力は以下の式で与えられます。

$d_k(n)=[a_{1k}d_k(n-1)+a_{2k}d_k(n-2)+2^{15}x(n)]\cdot 2^{-15}$

$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)]\cdot 2^{-a0k}$

k 番目のセクションの入力 $x_k(n)$は、前のセクションの出力 $y_{k-1}(n)$です。最初のセクション(k=0)の入力は input から読み込まれます。最後のセクション(k=K-1)の出力は output に書き込まれます。

coeff は係数を以下の順序に設定してください。

a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}

a_{0k} 項は k 番目のバイカッドの出力で行われる右シフトのビット数です。

各バイカッドでは飽和演算を 32 ビットで行います。各バイカッドの出力は 15 ビットまたは a_{0k} ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。

本関数を呼び出す前に InitIir を呼び出し、フィルタの作業領域を初期化してください。

本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>

#define K   4
#define NUMCOEF  (6*K)
#define N   50

#pragma section X
static short coeff_x[NUMCOEF];
#pragma section

static short coeff[24] = {15, 19144, -7581,  5301, 10602,  5301,
                          15, -1724,-23247, 13627, 27254, 13627,
                          15, 19144, -7581,  5301, 10602,  5301,
                          15, -1724,-23247, 13627, 27254, 13627};

static short input[50] = {32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000,
                 32000, 32000, 32000, 32000, 32000 };

short keisu[5]={ 1,2,20,4,5 };

void main()
{
   short *work, i;
   short output[N];

   for(i=0;i<NUMCOEF;i++) {
      coeff_x[i] = coeff[i];
    }

   if (InitIir(&work, K) != EDSP_OK){
      printf("Init Problem¥n");
   }
   for(i=0;i<N;i++){
      if (Iir1(&output[i], input[i], coeff_x, K, work) != EDSP_OK){
         printf("EDSP_OK not returned¥n");
      }
      printf("output[%d]:%d ¥n" ,i,output[i]);
   }
   if (FreeIir(&work, K) != EDSP_OK){
      printf("Free Problem¥n");
   }
}
```

インクルードヘッダ

フィルタ係数をXメモリ上に設定します。
フィルタ計算のワークエリアとしてライブラリの内部で使用しているのでYメモリは使用しないでください。

フィルタ係数は6個を1セクションとして設定し、セクションの先頭要素は右シフト数でフィルタ係数ではありません。

フィルタ係数をXメモリ上の変数に設定します。

フィルタの初期化
(1)ワークエリアアドレス
(2)フィルタセクションの数
Iir1関数発行の前に必須です。Yメモリをワークエリアとして（フィルタセクションの数）*2*2バイト使用します。

Iir1はデータ数1のIir関数と同じです。
Iir1を複数回発行する場合には、InitIirとFreeIirは最初と最後の1回ずつ発行します。

(e) 倍精度 IIR

■説明

【書式】

```
int DIir (short output[], const short input[], long no_samples,
         const long coeff[], long no_sections, long *workspace)
```

【引数】

output[]	出力データ y_{K-1}
input[]	入力データ x
no_samples	入力データの数 N
coeff[]	フィルタ係数
no_sections	2 次フィルタセクションの数 K
workspace	作業領域へのポインタ

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_samples < 1 ・no_sections < 1 ・$a_{0k} < 3$ ・k < K-1 で $a_{0k} > 32$ ・k = K-1 で $a_{0k} > 48$

【内容】 倍精度無限インパルス応答フィルタ処理を実行します。

【備考】 フィルタは、バイカッドという 2 次フィルタを K 個縦列に接続した構成になっています。各バイカッドの出力で付加的なスケーリングが行われます。フィルタ係数は符号付き 32 ビット固定小数点数で指定します。

各バイカッドの出力は、以下の方程式で与えられます。

$$d_k(n)=[a_{1k}d_k(n-1)+a_{2k}d_k(n-2)+2^{29}x(n)]\cdot 2^{-31}$$

$$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)]\cdot 2^{-a0k}\cdot 2^{2}$$

k 番目のセクションの入力 $x_k(n)$は、前のセクションの出力 $y_{k-1}(n)$です。最初のセクション(k=0)の入力は、input を 16 ビット左シフトした値が読み込まれます。最後のセクション(k=K-1)の出力は output に書き込まれます。

coeff は係数を以下の順序に設定してください。

a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}

a_{0k} 項は k 番目のバイカッドの出力で行われる右シフトのビット数です。

DIir は、フィルタ係数を 16 ビット値ではなく、32 ビット値で指定するという点で Iir と異なっています。積和演算の結果は 64 ビットで保持されます。中間ステージの出力は、a_{0k} ビット右シフトした結果の下位 32 ビットが取り出されます。オーバフローしたときは正または負の最大値となります。最終ステージでは、a_{0K-1} ビット右シフトした結果の下位 16 ビットが取り出されます。なお、オーバフローしたときは正または負の最大値となります。

本関数を呼び出す前に InitDIir を呼び出し、フィルタの作業領域を初期化してください。

遅延ノード $d_k(n)$は、30 ビットの値に丸められ、オーバフローしたときは正または負の最大値となります。

DIir は符号付き 32 ビット固定小数点数で係数を指定して使用してください。このとき、a_{0k} は k＜K-1 のときは 31、k=K-1 のときは 47 に設定してください。

DIir より Iir の方が実行速度は速いので、倍精度計算の必要がなければ Iir を使用してください。

output に input と同じ配列を指定した場合、input は上書きされます。

本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <filt_ws.h>
#include <ensigdsp.h>

#define K  5
#define NUMCOEF  (6*K)
#define N  50

#pragma section X
static long coeff_x[NUMCOEF];
#pragma section

static long coeff[60] =
   {31,1254686956, -496866304, 347415747, 694831502, 347415746,
    31,-113001278,-1523568505, 893094203,1786188388, 893094206,
    31,1254686956, -496866304, 347415747, 694831502, 347415746,
    31,-113001278,-1523568505, 893094203,1786188388, 893094206,
    47,1254686956, -496866304, 347415747, 694831502, 347415746};

static short input[100] = {
                      32000, 32000, 32000, 32000, 32000,
                      32000, 32000, 32000, 32000, 32000,
                      32000, 32000, 32000, 32000, 32000,
                      32000, 32000, 32000, 32000, 32000,
                      32000, 32000, 32000, 32000, 32000,
                      32000, 32000, 32000, 32000, 32000,
                      32000, 32000, 32000, 32000, 32000,
                      32000, 32000, 32000, 32000, 32000,
                      32000, 32000, 32000, 32000, 32000,
                      32000, 32000, 32000, 32000, 32000 };

void main()
{
   short i;
   short output[N];
   long *work;
   long nsamp;

   for(i=0;i<NUMCOEF;i++)
      coeff_x[i] = coeff[i];
   if(InitDIir(&work,K) != EDSP_OK){
      printf("InitDIir Problem¥n");
   }
   if(DIir(output, input, N, coeff_x, K, work) != EDSP_OK){
      printf("DIir Problem¥n");
   }
   if(FreeDIir(&work, K) != EDSP_OK){
      printf("FreeDIir Problem¥n");
   }
   for(i=0;i<N;i++){
      printf("output[%d]=%d¥n",i,output[i]);
   }
}
```

インクルードヘッダ

フィルタ係数をXメモリ上に設定します。
フィルタ計算のワークエリアとしてライブラリの内部で使用しているのでYメモリは使用しないでください。

フィルタ係数は6個を1セクションとして設定します。セクションの先頭要素は右シフト数でフィルタ係数ではありません。

右シフト数は
最後のセクション以外31。
最後のセクションのみ47。

フィルタ係数をXメモリ上の変数に設定します。

フィルタの初期化
(1)ワークエリアアドレス
(2)フィルタセクションの数
DIir関数発行の前に必須です。
Yメモリをワークエリアとして（フィルタセクションの数）*4*2バイト使用します。

DIir計算に使用したワークエリアの解放。DIir使用後は必ず行ってください。この関数を発行しないと、メモリが無駄に使用されてしまいます。

(f) 単一データ用倍精度 IIR

■説明

【書式】
```
int DIir1  (short output[], const short input[], long no_samples,
        const long coeff[], long no_sections, long *workspace)
```

【引数】

output	出力データ $y_{K-1}(n)$ へのポインタ
input	入力データ $x_0(n)$
coeff[]	フィルタ係数
no_sections	2 次フィルタセクションの数 K
workspace	作業領域へのポインタ

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_sections < 1 ・a_{0k} < 3 ・k < K-1 で a_{0k} > 32 ・k = K-1 で a_{0k} > 48

【内容】 単一データ用に倍精度無限インパルス応答フィルタ処理を実行します。

【備考】 フィルタはバイカッドという 2 次フィルタを K 個縦列に接続した構成になっています。各バイカッドの出力で付加的なスケーリングが行われます。フィルタ係数は符号付き 32 ビット固定小数点数で指定します。
各バイカッドの出力は、以下の方程式で与えられます。

$d_k(n)=[a_{1k}d_k(n-1)+a_{2k}d_k(n-2)+2^{29}x(n)]\cdot 2^{-31}$

$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)]\cdot 2^{-a0k}\cdot 2^{2}$

k 番目のセクションの入力 $x_k(n)$ は、前のセクションの出力 $y_{k-1}(n)$ です。最初のセクション(k=0)への入力は、input を 16 ビット左シフトした値が読み込まれます。最後のセクション(k=K-1)からの出力は output に書き込まれます。
coeff は係数を以下の順序に設定してください。
a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}
a_{0k} 項は k 番目のバイカッドの出力で行われる右シフトのビット数です。
DIir1 は、フィルタ係数を 16 ビット値ではなく、32 ビット値で指定するという点で Iir1 と異なっています。積和演算の結果は 64 ビットで保持されます。中間ステージの出力は、a_{0k} ビット右シフトした結果の下位 32 ビットが取り出されます。オーバフローしたときは、正または負の最大値となります。
最終ステージでは、a_{0K-1} ビット右シフトした結果の下位 16 ビットが取り出されます。なお、オーバフローしたときは正または負の最大値となります。
本関数を呼び出す前に InitDIir を呼び出し、フィルタの作業領域を初期化してください。
遅延ノード $d_k(n)$ は、30 ビットの値に丸められ、オーバフローしたときは正または負の最大値となります。
DIir1 は符号付き 32 ビット固定小数点数で係数を指定して使用してください。このとき、a_{0k} は k<K-1 のときは 31、k=K-1 のときは 47 に設定してください。
DIir1 より Iir1 の方が実行速度は速いので、倍精度計算の必要がなければ Iir1 を使用してください。
本関数はリエントラントではありません。

RENESAS

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>

#define K  5
#define NUMCOEF  (6*K)
#define N   50

#pragma section X
static long coeff_x[NUMCOEF];
#pragma section

static long coeff[60] =
 {31,1254686956, -496866304, 347415747, 694831502, 347415746,
  31,-113001278,-1523568505, 893094203,1786188388, 893094206,
  31,1254686956, -496866304, 347415747, 694831502, 347415746,
  31,-113001278,-1523568505, 893094203,1786188388, 893094206,
  47,1254686956, -496866304, 347415747, 694831502, 347415746};

static short input[N]   = {32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000 };

void main()
{
    short i;
    short output[N];
    long *work;

    for(i=0;i<NUMCOEF;i++)
        coeff_x[i] = coeff[i];

    if(InitDIir(&work, K) != EDSP_OK){
        printf("Init Problem¥n");
    }
    for(i=0;i<N;i++){
        if(DIir1(&output[i], input[i], coeff_x, K, work) !=EDSP_OK){
            printf("DIir1 error¥n");
        }
        printf("output[%d]:%d ¥n" ,i,output[i]);
    }
    if(FreeDIir(&work, K) != EDSP_OK){
        printf("Free DIir error¥n");
    }
}
```

インクルードヘッダ

フィルタ係数は6個を1セクションとして設定します。セクションの先頭要素は右シフト数でフィルタ係数ではありません。

フィルタ係数をXメモリ上に設定します。
フィルタ計算のワークエリアとしてライブラリの内部で使用しているのでYメモリは使用しないでく

右シフト数は
最後のセクション以外31。
最後のセクションのみ47。

フィルタ係数をXメモリ上の変数に設定します。

フィルタの初期化
(1)ワークエリアアドレス
(2)フィルタセクションの数
DIir1関数発行の前に必須です。Yメモリをワークエリアとして（フィルタセクションの数）*4*2バイト使用します。

DIir1はデータ数1のDIir関数と同じです。
DIir1を複数回発行する場合には、InitDIirとFreeDIirは最初と最後の1回ずつで良い。

(g) 適応 FIR

■説明

【書式】
```
int Lms  (short output[], const short input[],
          const short ref_output[], long no_samples,
          short coeff[], long no_coeffs, int res_shift,
          short conv_fact, short  *workspace)
```

【引数】

output[]	出力データ y
input[]	入力データ x
ref_output[]	所望の出力値 d
no_samples	入力データの数 N
coeff[]	適応フィルタ係数 h
no_coeffs	係数の数 K
res_shift	各出力に適用される右シフト
conv_fact	収束係数 2μ

	workspace	作業領域へのポインタ
【戻り値】	EDSP_OK	成功
	EDSP_BAD_ARG	以下のいずれかの場合です ・no_samples < 1 ・no_coeffs ≤ 2 ・res_shift < 0 ・res_shift > 25

【内容】 最小 2 乗平均アルゴリズム(LMS)を使って、実数適応 FIR フィルタ処理を実行します。

【備考】 FIR フィルタは以下の式で定義されます。

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k)\ x(n-k)\right] \cdot 2^{-res_shift}$$

積和演算の結果は 39 ビットで保持されます。出力 y(n) は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。

フィルタ係数の更新は、Widrow-Hoff アルゴリズムを使用します。

$h_{n+1}(k)=h_n(k)+2\mu e(n)x(n-k)$

ここで e(n) は所望する信号と実際の出力の誤差です。

$e(n)=d(n)-y(n)$

$2\mu e(n)x(n-k)$ の計算では、16 ビット×16 ビットの乗算を 2 回行います。どちらの乗算結果とも上位 16 ビットが保持され、オーバフローしたときは正または負の最大値となります。更新した係数の値が H'8000 になると、積和演算でオーバフローが発生する可能性があります。係数の値は H'8001～H'7FFF の範囲内になるように設定してください。

係数のスケーリングについては「(2) **係数のスケーリング**」を参照してください。係数は LMS フィルタによって適応させるので、最も安全なスケーリングは係数を 256 個未満にし、res_shift を 24 に設定する方法です。

conv_fact は通常正に設定してください。また H'8000 には設定しないでください。

本関数を呼び出す前に InitLms を呼び出し、フィルタを初期化してください。

output に input または ref_output と同じ配列を指定した場合、input または ref_output は上書きされます。

本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>

#define K   8
#define N   40
#define TWOMU 32767
#define RSHIFT 15
#define MAXSH 25

#pragma section X
static short coeff_x[K];
#pragma section
short data[N] =  {
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400};

short coeff[K] =  {
        0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000};

static short ref[N] = { -107, -143, 998, 1112, -5956,
                        -10781, 239, 13655, 11202, 2180,
                        -687, -2883, -7315, -6527, 196,
                        4278, 3712, 3367, 4101, 2703,
                        591, 695, -1061, -5626, -4200,
                        3585, 9285, 11796, 13416, 12994,
```

インクルードヘッダ

フィルタ係数をXメモリ上に設定します。フィルタ計算のワークエリアとしてライブラリの内部で使用しているのでYメモリは使用しないでください。

```
                         10231, 5803, -449, -6782, -11131,
                         -10376, -2968, 2588, -1241, -6133};

void main()
{
    short *work, i, errc;
    short output[N];
    short twomu;
    int nsamp, ncoeff, rshift;

    /* copy coeffs into X RAM */

    for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
    nsamp = 10;
    ncoeff = K;
    rshift = RSHIFT;
    twomu = TWOMU;

    for (i = 0; i < N; output[i++] = 0) ;
    ncoeff = K;/* 係数の数設定 */
    nsamp = N;/* サンプル数設定 */

    for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
    if (InitLms(&work, K) != EDSP_OK){
        printf("Init Problem¥n");
    }
    if(Lms(output, data, ref, nsamp, coeff_x, ncoeff, RSHIFT,TWOMU, work) !=
                                                                 EDSP_OK){
        printf("Lms Problem¥n");
    }
    if (FreeLms(&work, K) != EDSP_OK){
        printf( "Free Problem¥n");
    }
    for (i = 0; i < N; i++){
        printf("#%2d output:%6d ¥n",i,output[i]);
    }
}
```

フィルタ係数をXメモリ上の変数に設定します。

フィルタの初期化
(1)ワークエリアアドレス
(2)係数の数
LMS関数発行の前に必須です。
Yメモリをワークエリアとして（係数の数）*2+8
バイト使用します。

LMS計算に使用したワークエリアの解放。LMS使用後は必ず行ってください。この関数を発行しないと、メモリが無駄に使用されてしまいます。

(h) 単一データ用適応 FIR

■説明

【書式】
```
int Lms1  (short *output, short input, short ref_output,
           short coeff[],long no_coeffs, int res_shift,
           short conv_fact, short *workspace)
```

【引数】

output	出力データ y(n)へのポインタ
input	入力データ x(n)
ref_output	所望の出力値 d(n)
coeff[]	適応フィルタ係数 h
no_coeffs	係数の数 K
res_shift	各出力に適用される右シフト
conv_fact	収束係数 2μ
workspace	作業領域へのポインタ

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_coeffs ≤ 2 ・res_shift < 0 ・res_shift > 25

【内容】 最小 2 乗平均アルゴリズム(LMS)を使って、単一データ用に実数適応 FIR フィルタ処理を実行します。

【備考】 FIR フィルタは以下の式で定義されます。

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k)x(n-k)\right] \cdot 2^{-res_shift}$$

積和演算の結果は 39 ビットで保持されます。出力 y(n)は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。

フィルタ係数の更新は Widrow-Hoff アルゴリズムを使用します。

$h_{n+1}(k)=h_n(k)+2\mu e(n)x(n-k)$

ここで e(n)は所望する信号と実際の出力の誤差です。

$e(n)=d(n)-y(n)$

$2\mu e(n)x(n-k)$の計算では、16 ビット×16 ビットの乗算を 2 回行います。どちらの乗算とも、上位 16 ビットが保持され、オーバフローしたときは正または負の最大値となります。更新した係数の値が H'8000 になると、積和演算でオーバフローが発生する可能性があります。係数の値は H'8001～H'7FFF の範囲内になるように設定してください。

係数のスケーリングについては「(2) **係数のスケーリング**」を参照してください。係数は LMS フィルタによって適応させるので、最も安全なスケーリングは係数を 256 個未満にし、res_shift を 24 に設定する方法です。

conv_fact は通常正に設定してください。また H'8000 には設定しないでください。

本関数を呼び出す前に InitLms を呼び出し、フィルタを初期化してください。

本関数はリエントラントではありません。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>      } インクルードヘッダ
#include <filt_ws.h>

#define K   8
#define N   40
#define TWOMU 32767
#define RSHIFT 15
#define MAXSH 25

#pragma section X
static short coeff_x[K];    ← Xメモリに配置する変数はpragma sectionでセクション内に定義します。
#pragma section

short data[N] =  {
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
        0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400};

short coeff[K] =  {
        0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000};

static short ref[N] = { -107, -143, 998, 1112, -5956,
                        -10781, 239, 13655, 11202, 2180,
                        -687, -2883, -7315, -6527, 196,
                        4278, 3712, 3367, 4101, 2703,
                        591, 695, -1061, -5626, -4200,
                        3585, 9285, 11796, 13416, 12994,
                        10231, 5803, -449, -6782, -11131,
                        -10376, -2968, 2588, -1241, -6133};
void main()
{
    short *work, i, errc;
    short output[N];
    short twomu;
    int nsamp, ncoeff, rshift;

    /* copy coeffs into X RAM */

    for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];    ← フィルタ係数をXメモリ上の変数に設定します。
    }
    nsamp = 10;
```

```
    ncoeff = K;
    rshift = RSHIFT;
    twomu = TWOMU;
    for (i = 0; i < N; output[i++] = 0) ;



    ncoeff = K;/* 係数の数設定 */
    nsamp = N;/* サンプル数設定 */

    for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
    if (InitLms(&work, K) != EDSP_OK){
        printf("Init Problem¥n");
    }
    for(i=0;i<nsamp;i++){
        if(Lms1(&output[i], data[i], ref[i], coeff_x, ncoeff, RSHIFT, TWOMU,
                                                        work) != EDSP_OK){
            printf("Lms1 Problem¥n");
        }
    }
    if (FreeLms(&work, K) != EDSP_OK){
        printf( "Free Problem¥n");
    }
    for (i = 0; i < N; i++){
        printf("#%2d output:%6d ¥n",i,output[i]);
    }
}
```

フィルタの初期化
(1)ワークエリアアドレス
(2)係数の数
LMS1関数発行の前に必須です。
Yメモリをワークエリアとして
（係数の数）*2+8
バイト使用します。

LMS1計算に使用したワークエリアの解放。
LMS1使用後は必ず行ってください。
この関数を発行しないと、メモリが無駄に使用されてしまいます。

(i) FIR 作業領域割り付け

■説明

【書式】 `int InitFir (short **workspace, long no_coeffs)`

【引数】

workspace	作業領域へのポインタへのポインタ
no_coeffs	係数の数 K

【戻り値】

EDSP_OK	成功
EDSP_NO_HEAP	workspace の使用できるメモリスペースが不十分
EDSP_BAD_ARG	no_coeffs ≤ 2

【内容】 Fir と Fir1 で使用する作業領域を割り付けます。

【備考】 すでに入力されているデータは 0 に初期化されます。
Fir、Fir1、Lms および Lms1 だけが InitFir で割り付けられた作業領域を操作することができます。
ユーザプログラムから作業領域を直接アクセスしないでください。
本関数はリエントラントではありません。

(j) IIR 作業領域割り付け

■説明

【内容】 `int InitIir (short **workspace, long no_sections)`

【引数】

workspace　　作業領域へのポインタへのポインタ
no_sections　　2 次フィルタセクションの数 K

【戻り値】

EDSP_OK　　成功
EDSP_NO_HEAP　　workspace の使用できるメモリスペースが不十分
EDSP_BAD_ARG　　no_sections < 1

【内容】 Iir と Iir1 で使用する作業領域を割り付けます。

【備考】 すでに入力されているデータは 0 に初期化されます。
Iir と Iir1 だけが InitIir で割り付けられた作業領域を操作することができます。ユーザプログラムから作業領域を直接アクセスしないでください。
本関数はリエントラントではありません。

(k) 倍精度 IIR 作業領域割り付け

■説明

【書式】 `int InitDIir (long **workspace, long no_sections)`

【引数】

workspace　　作業領域へのポインタへのポインタ
no_sections　　2 次フィルタセクションの数 K

【戻り値】

EDSP_OK　　成功
EDSP_NO_HEAP　　workspace の使用できるメモリスペースが不十分
EDSP_BAD_ARG　　no_sections < 1

【内容】 DIir と DIir1 で使用する作業領域を割り付けます。

【備考】 すでに入力されているデータは 0 に初期化されます。
DIir と DIir1 だけが InitDIir で割り付けられた作業領域を操作することができます。
本関数はリエントラントではありません。

(l) 適応 FIR 作業領域割り付け

■説明

【書式】 `int InitLms (short **workspace, long no_coeffs)`

【引数】

workspace　　作業領域へのポインタへのポインタ
no_coeffs　　係数の数 K

【戻り値】

EDSP_OK　　成功
EDSP_NO_HEAP　　workspace の使用できるメモリスペースが不十分
EDSP_BAD_ARG　　no_coeffs ≤ 2

【内容】 Lms と Lms1 で使用する作業領域を割り付けます。

【備考】 すでに入力されているデータは 0 に初期化されます。
Fir、Fir1、Lms および Lms1 だけが InitLms で割り付けられた作業領域を操作することができます。
ユーザプログラムから作業領域を直接アクセスしないでください。
本関数はリエントラントではありません。

(m) FIR 作業領域開放

■説明

【書式】 `int FreeFir (short **workspace, long no_coeffs)`

【引数】
workspace　作業領域へのポインタへのポインタ
no_coeffs　係数の数 K

【戻り値】
EDSP_OK　成功
EDSP_BAD_ARG　no_coeffs ≤ 2

【内容】 InitFir で割り付けられた作業領域を解放します。

【備考】 本関数はリエントラントではありません。

(n) IIR 作業領域開放

■説明

【書式】 `int FreeIir (short **workspace, long no_sections)`

【引数】
workspace　作業領域へのポインタへのポインタ
no_sections　2 次フィルタセクションの数 K

【戻り値】
EDSP_OK　成功
EDSP_BAD_ARG　no_sections < 1

【内容】 InitIir で割り付けられた作業領域を解放します。

【備考】 本関数はリエントラントではありません。

(o) 倍精度 IIR 作業領域開放

■説明

【書式】 `int FreeDIir (long **workspace, long no_sections)`

【引数】
workspace　作業領域へのポインタへのポインタ
no_sections　2 次フィルタセクションの数 K

【戻り値】
EDSP_OK　成功
EDSP_BAD_ARG　no_section ≤ 2

【内容】 InitDIir で割り付けられた作業領域メモリを解放します。

【備考】 本関数はリエントラントではありません。

(p) 適応 FIR 作業領域開放

■説明

【書式】 `int FreeLms (short **workspace, long no_coeffs)`

【引数】
workspace　作業領域へのポインタへのポインタ
no_coeffs　係数の数 K

【戻り値】
EDSP_OK　成功
EDSP_BAD_ARG　no_coeffs < 1

【内容】 InitLms で割り付けられた作業領域メモリを解放します。

【備考】 本関数はリエントラントではありません。

3.13.7 畳み込みと相関

(1) 関数一覧

表 3.32 DSP ライブラリ関数一覧(畳み込み)

項番	項目	関数名	説明
1	完全畳み込み	ConvComplete	2 つの配列の完全な畳み込みを計算します。
2	周期的畳み込み	ConvCyclic	2 つの配列の周期的な畳み込みを計算します。
3	部分的畳み込み	ConvPartial	2 つの配列の部分的な畳み込みを計算します。
4	相関	Correlate	2 つの配列の相関を計算します。
5	周期的相関	CorrCyclic	2 つの配列の周期的な相関を計算します。

これらの関数を使用する際は、2 つの入力配列のうち 1 つは X メモリに、もう 1 つは Y メモリに配置してください。出力配列はどのメモリに配置してもかまいません。

(2) 各関数の説明

(a) 完全畳み込み

■説明

【書式】
```
int ConvComplete  (short output[], const short ip_x[],
                   const short ip_y[], long x_size,
                   long y_size, int res_shift)
```

【引数】

output[]	出力 z
ip_x[]	入力 x
ip_y[]	入力 y
x_size	ip_x のサイズ X
y_size	ip_y のサイズ Y
res_shift	各出力に適用される右シフト

【戻り値】

EDSP_OK 成功

EDSP_BAD_ARG 以下のいずれかの場合です
- ・x_size < 1
- ・y_size < 1
- ・res_shift < 0
- ・res_shift > 25

【内容】 2 つの入力配列 x,y を完全に畳み込み、結果を出力配列 z に書き出します。

【備考】

$$z(m) = \left[\sum_{i=0}^{X-1} x(i)\, y(m-i)\right] \cdot 2^{-res_shift} \qquad 0 \le m < X+Y-1$$

入力配列外のデータは 0 として読み込まれます。

ip_x は X メモリに、ip_y は Y メモリに、output は任意のメモリに配置してください。

また、配列 output のサイズは、(xsize+ysize-1)以上確保しておく必要があります。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#define NX   8
#define NY   8
#define NOUT  NX+NY-1

#pragma section X
static short datx[NX];
#pragma section Y
static short daty[NY];
#pragma section

short w1[5] = {-1, -32768, 32767, 2, -3, };
short x1[5] = {1, 32767, -32767, -32767, -2, };

void main()
{
    short i;
    short output[NOUT];
    int xsize, ysize, rshift;

    /* copy data into X and Y RAM */
    for(i=0;i<NX;i++){
        datx[i] = w1[i%5];
    }
    for(i=0;i<NY;i++){
        daty[i] = x1[i%5];
    }
    xsize = NX;
    ysize = NY;
    rshift = 15;

    if(ConvComplete(output, datx, daty, xsize, ysize, rshift) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }

    for(i=0;i<NX;i++){
        printf("#%3d  dat_x:%6d  dat_y:%6d ¥n",i,datx[i],daty[i]);
    }
    for(i=0;i<NOUT;i++){
        printf("#%3d  output:%d ¥n",i,output[i]);
    }

}
```


(b) 周期的畳み込み

■説明

【書式】 `int ConvCyclic (short output[], const short ip_x[], const short ip_y[], long size, int res_shift)`

【引数】

output[]	出力 z
ip_x[]	入力 x
ip_y[]	入力 y
size	配列のサイズ N
res_shift	各出力に適用される右シフト

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 1 ・res_shift < 0 ・res_shift > 25

【内容】 2 つの入力配列 x,y を周期的に畳み込み、結果を出力配列 z に書き出します。

【備考】

$$z(m) = \left[\sum_{i=0}^{N-1} x(i)\, y\left(|m - i + N|_N \right)\right] \cdot 2^{-res_shift} \qquad 0 \le m < N$$

ここで、$|i|_N$ は剰余(i % N)を意味します。
ip_x は X メモリに、ip_y は Y メモリに、output は任意のメモリに配置してください。
また、配列 output のサイズは、size 以上確保しておく必要があります。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
short x2[5] = {1, 32767, -32767, -32767, -2, };
short w2[5] = {-1, -32768, 32767, 2, -3, };

#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section

void main()
{
    short i;
    short output[N];
    int size, rshift;

    /* copy data into X and Y RAM */
    for(i=0;i<N;i++){
        datx[i] = w2[i];
        daty[i] = x2[i];
    }
    size = N ;
    rshift = 15;
    if(ConvCyclic(output, datx, daty, size, rshift) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }

    for(i=0;i<N;i++){
        printf("#%2d  ip_x:%6d  ip_y:%6d  output:%6d ¥n",i,datx[i],daty[i],
                                                              output[i]);
    }
}
```


(c) **部分的畳み込み**

■説明

【書式】
```
int ConvPartial (short output[], const short ip_x[],
                 const short ip_y[],long x_size, long y_size,
                 int res_shift)
```

【引数】

output[]	出力 z
ip_x[]	入力 x
ip_y[]	入力 y
x_size	ip_x のサイズ X
y_size	ip_y のサイズ Y
res_shift	各出力に適用される右シフト

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・x_size < 1 ・y_size < 1 ・res_shift < 0 ・res_shift > 25

【内容】　本関数は2つの入力配列x,yを畳み込み、結果を出力配列zに書き出します。

【備考】　入力配列外のデータから引き出された出力は含まれていません。

$$z(m) = \left[\sum_{i=0}^{A-1} a(i)\ b(m + A - 1 - i)\right] \cdot 2^{-res_shift} \qquad 0 \le m \le |A-B|$$

ただし、配列の個数はa < bで、Aはaのサイズ、Bはbのサイズです。
入力配列外のデータは0として読み込まれます。
ip_xはXメモリに、ip_yはYメモリに、outputは任意のメモリに配置してください。
また、配列outputのサイズは、(|xsize-ysize|+1)以上確保しておく必要があります。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#define   NX    5
#define   NY    5
short x3[5] = {1, 32767, -32767, -32767, -2, };
short w3[5] = {-1, -32768, 32767, 2, -3, };

#pragma section X
static short datx[NX];
#pragma section Y
static short daty[NY];
#pragma section

void main()
{
    short i;
    short output[NY+NX];
    int ysize, xsize, rshift;

    /* copy data into X and Y RAM */
    for(i=0;i<NX;i++){
        datx[i] = w3[i];
    }
    for(i=0;i<NY;i++){
        daty[i] = x3[i];
    }

    xsize = NX;
    ysize = NY;
    rshift = 15;
    if(ConvPartial(output, datx, daty, xsize, ysize, rshift) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<NX;i++){
        printf("ip_x=%d  ¥n",datx[i]);
    }
    for(i=0;i<NY;i++){
        printf("ip_y=%d  ¥n",daty[i]);
    }
    for(i=0;i<(NY+NX);i++){
        printf("output=%d ¥n",output[i]);
    }
}
```

インクルードヘッダ

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義します。

畳み込み計算に使用するデータ設定

(d) 相関

■説明

【書式】

```
int Correlate  (short output[], const short ip_x[],
                const short ip_y[], long x_size, long y_size,
                long no_corr, int x_is_larger, int res_shift)
```

【引数】

output[]	出力 z
ip_x[]	入力 x
ip_y[]	入力 y
x_size	ip_x のサイズ X
y_size	ip_y のサイズ Y
no_corr	計算する相関の数 M
x_is_larger	X=Y のときの配列指定
res_shift	各出力に適用される右シフト

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・x_size < 1 ・y_size < 1 ・no_corr < 1 ・res_shift < 0 ・res_shift > 25 ・x_is_larger ≠ 0 または 1

【内容】 2 つの入力配列 x,y の相関を求め、結果を出力配列 z に書き出します。

【備考】 以下の式では配列の個数は a < b で、A は a のサイズとします。
x_is_larger=0 とすると x を a とし、x_is_larger=1 とすると x を b とします。
a 配列より b 配列が少ない場合の動作は保証致しません。
入力配列 x,y の大小と x_is_larger は矛盾のないように設定をお願いします。

$$z(m) = \left[\sum_{i=0}^{A-1} a(i)\, b(i+m)\right] \cdot 2^{-res_shift} \qquad 0 \le m < M$$

A < X + M となっても差し支えありません。この場合、入力配列外のデータは 0 を使用します。
res_shift=0 は通常の整数計算に、res_shift=15 は小数計算に相当します。
ip_x は X メモリに、ip_y は Y メモリに、output は任意のメモリに配置してください。
また、配列 output のサイズは、no_corr 以上確保しておく必要があります。

■使用例

```
#include <stdio.h>        } インクルードヘッダ
#include <ensigdsp.h>

#define NY   5
#define NX   5
#define M    4
#define MAXM  NX+NY
short x4[5] = {1, 32767, -32767, -32767, -2, };
short w4[5] = {-1, -32768, 32767, 2, -3, };

#pragma section X
static short datx[NX];
#pragma section Y
static short daty[NY];
#pragma section

void main()
{
    short i;
    int ysize, xsize, ncorr, rshift;
    short output[MAXM];
    int x_is_larger;

    /* copy data into X and Y RAM */
    for(i=0;i<NX;i++){
        datx[i] = w4[i%5];
    }
    for(i=0;i<NY;i++){
        daty[i] = x4[i%5];
    }

    /* test working of stack */
    ysize = NY;
    xsize = NX;
    ncorr = M;
    rshift = 15;

    x_is_larger=0;
    for (i = 0; i < MAXM; output[i++] = 0);

    if (Correlate(output, datx, daty, xsize, ysize, ncorr,x_is_larger,rshift)
                                                            != EDSP_OK){
      printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<MAXM;i++){
        printf("[%d]:output=%d¥n",i,output[i]);
    }
}
```

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義します。

計算に使用するデータ設定

(e) 周期的相関

■説明

【書式】 `int CorrCyclic (short output[], const short ip_x[], const short ip_y[], long size, int reverse, int res_shift)`

【引数】

output[]	出力 z
ip_x[]	入力 x
ip_y[]	入力 y
size	配列のサイズ N
reverse	反転フラグ
res_shift	各出力に適用される右シフト

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・size < 1 ・res_shift < 0

・res_shift > 25
・reverse ≠ 0 または 1

【内容】　周期的に配列 x,y の相関を求め、結果を出力配列 z に書き出します。

【備考】

$$z(m) = \left[\sum_{i=0}^{N-1} x(i)\, y\left(\left|i + m\right|_N\right)\right] \cdot 2^{-res_shift} \qquad 0 \le m < N$$

ここで、$|i|_N$ は剰余(i % N)を意味します。reverse=1 の場合、出力のデータは反転され、実際の計算は以下のようになります。

$$z(m) = \left[\sum_{i=0}^{N-1} y(i)\, x\left(\left|i + m\right|_N\right)\right] \cdot 2^{-res_shift} \qquad 0 \le m < N$$

ip_x は X メモリに、ip_y は Y メモリに、output は任意のメモリに配置してください。
また、配列 output のサイズは、size 以上確保しておく必要があります。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5

short x5[5] = {1, 32767, -32767, -32767, -2, };
short w5[5] = {-1, -32768, 32767, 2, -3, };

#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section

void main()
{
    short i;
    short output[N];
    int size, rshift;
    int reverse;
    int result;

    /* TEST CYCLIC CORRELATION OF X WITH Y */
    reverse=0;
    /* copy data into X and Y RAM */
    for(i=0;i<N;i++){
        datx[i] = w5[i];
        daty[i] = x5[i];
    }

    /* test working of stack */
    size = N;
    rshift = 15;

    if (CorrCyclic(output, datx, daty, size, reverse, rshift) != EDSP_OK){
        printf("EDSP_OK not returned - this one¥n");
    }

    for(i=0;i<N;i++){
        printf("output[%d]=%d¥n",i,output[i]);
    }
}
```

3.13.8 その他

(1) 関数一覧

表 3.33 DSP ライブラリ関数一覧(その他)

項番	項目	関数名	説明
1	H'8000→H'8001 置換	Limit	H'8000 のデータを H'8001 に置き換えます。
2	X メモリ→Y メモリコピー	CopyXtoY	配列を X メモリから Y メモリにコピーします。
3	Y メモリ→X メモリコピー	CopyYtoX	配列を Y メモリから X メモリにコピーします。
4	X メモリへコピー	CopyToX	配列を指定した場所から X メモリにコピーします。
5	Y メモリへコピー	CopyToY	配列を指定した場所から Y メモリにコピーします。
6	X メモリからコピー	CopyFromX	配列を X メモリから指定した場所にコピーします。
7	Y メモリからコピー	CopyFromY	配列を Y メモリから指定した場所にコピーします。
8	白色ガウス雑音	GenGWnoise	白色ガウス雑音を生成します。
9	マトリックス乗算	MatrixMult	2 つのマトリックスの乗算をします。
10	乗算	VectorMult	2 つのデータの乗算をします。
11	平均 2 乗値	MsPower	2 乗平均強度を求めます。
12	平均	Mean	平均を求めます。
13	平均と偏差	Variance	平均と偏差を求めます。
14	最大値	MaxI	整数配列の最大値を求めます。
15	最小値	MinI	整数配列の最小値を求めます。
16	最大絶対値	PeakI	整数配列の最大絶対値を求めます。

(2) 各関数の説明

(a) H'8000→H'8001 置換

■説明

【書式】 `int Limit (short data[], long no_elements, int data_is_x)`

【引数】
data[] データ配列
no_elements データ数
data_is_x データ配置指定

【戻り値】
EDSP_OK 成功
EDSP_BAD_ARG 以下のいずれかの場合です
・no_elements < 1
・data_is_x ≠ 0 または 1

【内容】 値が H'8000 の入力データを H'8001 に置き換えます。これにより、DSP 命令の固定小数点乗算の際にオーバフローが発生しないようにします。

【備考】 この処理を行っても積和演算の加算でオーバフローが発生する可能性はあります。
data_is_x=1 のときは data は X メモリに、data_is_x=0 のときは Y メモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section

void main()
{
   short i;
   int size;
   int src_x;

   /* copy data into X and Y RAM */
   for(i=0;i<N;i++) {
      datx[i] = dat[i%4];
      daty[i] = dat[i%4];
      printf("BEFORE NO %d datx daty :%d:%d ¥n",i,datx[i], daty[i]);
   }

   size = N;
   src_x = 1;

   if (Limit(datx, size, src_x) != EDSP_OK){
      printf( "EDSP_OK not returned¥n");
   }

   src_x = 0;
   if (Limit(daty, size, src_x) != EDSP_OK){
      printf( "EDSP_OK not returned¥n");
   }

   for(i=0;i<N;i++) {
      printf("After NO %d datx daty :%d:%d¥n",i,datx[i], daty[i]);
   }
}
```

インクルードヘッダ

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義します。

データ設定

Xメモリを使用する場合

Yメモリを使用する場合

(b) X メモリ→Y メモリコピー

■説明

【書式】 `int CopyXtoY (short op_y[], const short ip_x[], long n)`

【引数】

op_y[]	出力配列
ip_x[]	入力配列
n	データ数

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	n < 1

【内容】 配列を ip_x から op_y へコピーします。

【備考】 ip_x は X メモリに、op_y は Y メモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};

#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section

void main()
{
    int i;

    for(i=0;i<N;i++){
        daty[i]=0;
        datx[i]=dat[i%4];
    }

    if(CopyXtoY(daty, datx, N) != EDSP_OK){
        printf("CopyXtoY Problem¥n");
    }
    printf("no_elements:%d ¥n",N);
    for(i=0;i<N;i++){
        printf("#%2d  op_x:%6d  ip_y:%6d ¥n",i,datx[i],daty[i]);
    }
}
```

インクルードヘッダ

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義します。

データ設定

(c) Y メモリ→X メモリコピー

■説明

【書式】 int CopyYtoX (short op_x[], const short ip_y[], long n)

【引数】
op_x[] 出力配列
ip_y[] 入力配列
n データ数

【戻り値】
EDSP_OK 成功
EDSP_BAD_ARG n < 1

【内容】 配列を ip_y から op_x へコピーします。

【備考】 ip_y は Y メモリ、op_x は X メモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  5
static short dat[N] = { -32768, 32767, -32768, 0,3};

#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section

void main()
{
    int i;

    for(i=0;i<N;i++){
        daty[i]=dat[i];
    }

    if(CopyYtoX(datx, daty, N)!= EDSP_OK){
     printf("CopyYtoX error!¥n");
    }

    printf("no_elements %d ¥n",N);
    for(i=0;i<N;i++){
        printf("#%2d  po_x:%6d  ip_y:%6d ¥n",i,datx[i],daty[i]);
    }
}
```

インクルードヘッダ

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義する。

データ設定

(d) X メモリへコピー

■説明

【書式】 `int CopyToX (short op_x[], const short input[], long n)`

【引数】

op_x[]	出力配列
input[]	入力配列
n	データ数

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	n < 1

【内容】 配列 input を op_x へコピーします。

【備考】 op_x は X メモリに、input は任意のメモリに配置してください。

■使用例

```
#include <stdio.h>        } インクルードヘッダ
#include <ensigdsp.h>

#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};

#pragma section X
static short datx[N];     ← Xメモリに配置する変数はpragma sectionでセクション内に定義する。
#pragma section

void main()
{
    int   i;
    short data[N];

    for(i=0;i<N;i++){     ← データ設定
       data[i]=dat[i];
    }
    if(CopyToX(datx, data, N) !=EDSP_OK){
        printf("CopyToX Problem¥n");
    }
    printf("no_elements %d¥n",N);
    for(i=0;i<N;i++){
        printf("#%2d  op_x:%6d  input:%6d ¥n",i,datx[i],data[i]);
    }
}
```

(e) Y メモリへコピー

■説明

【書式】 `int CopyToY (short op_y[], const short input[], long n)`

【引数】

op_y[]	出力配列
input[]	入力配列
n	データ数

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	n < 1

【内容】 配列 input を op_y へコピーします。

【備考】 op_y は Y メモリに、input は任意のメモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};

#pragma section Y
static short daty[N];
#pragma section

void main()
{
    int i;
    short data[N] ;

    for(i = 0; i < N; i++){
        data[i] = dat[i%4] ;
    }
    if(CopyToY(daty, data, N) != EDSP_OK){
        printf("CopyToY Problem¥n");
    }
    printf("no_elements %ld ¥n",N);
    for(i = 0; i < N; i++){
        printf("#%2d  op_y:%6d  input:%6d ¥n",i,daty[i],data[i]);
    }
}
```

インクルードヘッダ

Yメモリに配置する変数はpragma sectionでセクション内に定義する。

データ設定

(f) X メモリからコピー

■説明

【書式】 `int CopyFromX (short output[], const short ip_x[], long n)`

【引数】
output[]	出力配列
ip_x[]	入力配列
n	データ数

【戻り値】
EDSP_OK	成功
EDSP_BAD_ARG	n < 1

【内容】 配列 ip_x を output へコピーします。

【備考】 ip_x は X メモリに、output は任意のメモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
static short out_dat[N] ;

#pragma section X
static short datx[N];
#pragma section

void main()
{
    int i;

    for(i=0;i<N;i++){
        datx[i]=dat[i];
    }
    if(CopyFromX(out_dat,datx, N) != EDSP_OK){
        printf("CopyFromX Problem¥n");
    }
    for(i=0;i<N;i++){
        printf("#%3d output:%6d  ip_x:%6d ¥n",i,out_dat[i],datx[i]);
    }
    printf("no_elements:%ld¥n",N);
}
```

インクルードヘッダ

Xメモリに配置する変数はpragma sectionでセクション内に定義する。

データ設定

(g) Y メモリからコピー

■説明

【書式】 `int CopyFromY (short output[], const short ip_y[], long n)`

【引数】
output[] 出力配列
ip_y[] 入力配列
n データ数

【戻り値】
EDSP_OK 成功
EDSP_BAD_ARG n < 1

【内容】 配列 ip_y を output へコピーします。

【備考】 ip_y は Y メモリに、output は任意のメモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
static short out_dat[N] ;

#pragma section Y
static short daty[N];
#pragma section

void main()
{
    int i;

    for(i=0;i<N;i++){
        daty[i]=dat[i];
    }
    if(CopyFromY(out_dat,daty, N)!= EDSP_OK){
         printf("CopyFormY Problem¥n");
    }
    printf("no_elements:%d ¥n",N);
    for(i=0;i<N;i++){
        printf("#%2d  output:%6d  ip_y:%6d ¥n",i,out_dat[i],daty[i]);
    }
}
```

インクルードヘッダ

Yメモリに配置する変数はpragma sectionでセクション内に定義する。

データ設定

(h) 白色ガウス雑音

■説明

【書式】 `int GenGWnoise (short output[], long no_samples, float variance)`

【引数】
- `output[]` 白色雑音データの出力
- `no_samples` 出力データ数
- `variance` ノイズ分布の偏差σ^2

【戻り値】
- `EDSP_OK` 成功
- `EDSP_BAD_ARG` 以下のいずれかの場合です
  - ・`no_samples < 1`
  - ・`variance ≤ 0.0`

【内容】 平均が 0 で、ユーザが指定した偏差を持つ白色ガウス雑音を生成します。

【備考】 出力データは 2 つ 1 組で生成されます。1 組の出力データを生成するために `rand` 関数を使用し、x の 2 乗合計が 1 未満になる組が求められるまで-1～1 の間で 1 組の乱数γ_1、γ_2を生成します。そして、1 組の出力データo_1、o_2が以下の式で計算されます。

$$o_1 = \sigma\gamma_1\sqrt{-2\ln(x)/x}$$

$$o_2 = \sigma\gamma_2\sqrt{-2\ln(x)/x}$$

データ数を奇数に設定した場合、最後の組の 2 番目のデータは破棄されます。
本関数が呼び出している標準ライブラリの `rand` 関数はリエントラントではないので、生成される乱数γ_1、γ_2の順番が常に同じになるとは限りません。しかし、生成される白色雑音o_1、o_2の特性に影響を及ぼすことはありません。
本関数は浮動小数点演算を使用しています。浮動小数点演算は処理速度が遅くなるので、本関数は評価用として使うことをおすすめします。

■使用例

```
#include <stdio.h>       } インクルードヘッダ
#include <ensigdsp.h>

#define MAXG  4.5   /* approx. saturating level for N(0,1) random variable */
#define N_SAMP  10  /* number of samples generated in a frame             */

void main()
{
    short out[N_SAMP];
    float var;
    int   i;

    var = 32768 / MAXG * 32768 / MAXG;
    if(GenGWnoise(out, N_SAMP, var) !=EDSP_OK){
        printf("GenGWnoise Problem¥n");
    }
    for(i=0;i<N_SAMP;i++){
        printf("#%2d  out:%6d  ¥n",i,out[i]);
    }
}
```

(i) マトリックス乗算

■説明

【書式】 `int MatrixMult (void *op_matrix, const void *ip_x, const void *ip_y, long m, long n, long p, int x_first, int res_shift)`

【引数】

op_matrix	出力の第一データへのポインタ
ip_x	入力 x の第一データへのポインタ
ip_y	入力 y の第一データへのポインタ
m	マトリックス 1 の行数
n	マトリックス 1 の列数、マトリックス 2 の行数
p	マトリックス 2 の列数
x_first	マトリックス乗算の順番指定
res_shift	各出力に適用される右シフト

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・m,n,または p < 1 ・res_shift < 0 ・res_shift > 25 ・x_first ≠ 0 または 1

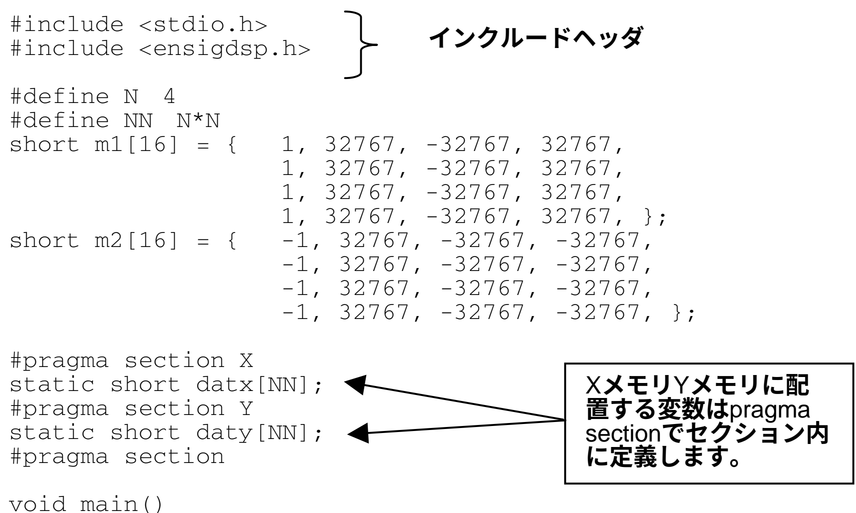
【内容】 2 つのマトリックス x,y の乗算を行い、結果を op_matrix に配置します。

【備考】 x_first=1 の場合、x•y を計算します。このとき、ip_x は m×n、ip_y は n×p、op_matrix は m×p となります。
x_first=0 の場合、y•x を計算します。このとき、ip_y は m×n、ip_x は n×p、op_matrix は m×p となります。
積和演算の結果は 39 ビットで保持されます。出力 y(n) は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。なお、オーバフローしたときは正または負の最大値となります。
各マトリックスは通常の C 様式(行優先順)で配置されます。

$$\begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \end{pmatrix}$$

任意の配列サイズを指定できるようにするために、配列パラメタは void*で指定します。これらのパラメタは short 変数を指すようにしてください。
入力配列 ip_x,ip_y と出力配列 op_matrix は別々に用意してください。
ip_x は X メモリに、ip_y は Y メモリに、op_matrix は任意のメモリに配置してください。

■使用例




```
#include <stdio.h>
#include <ensigdsp.h>

#define N  4
#define NN  N*N
short m1[16] = {   1, 32767, -32767, 32767,
                   1, 32767, -32767, 32767,
                   1, 32767, -32767, 32767,
                   1, 32767, -32767, 32767, };
short m2[16] = {   -1, 32767, -32767, -32767,
                   -1, 32767, -32767, -32767,
                   -1, 32767, -32767, -32767,
                   -1, 32767, -32767, -32767, };

#pragma section X
static short datx[NN];
#pragma section Y
static short daty[NN];
#pragma section

void main()
```

```
{
    short i, j;
    short output[NN];
    int m, n, p, rshift, x_first;
    long sum;

    for (i = 0; i < NN; output[i++] = 0) ;
    /* copy data into X and Y RAM */
    for(i=0;i<NN;i++) {
        datx[i] = m1[i%16];
        daty[i] = m2[i%16];          ← データ設定
    }

    m = n = p = N;
    rshift = 15;
    x_first = 1;

    if (MatrixMult(output, datx, daty, m, n, p, x_first, rshift) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    for(i=0;i<NN;i++) {
        printf("output[%d]=%d¥n",i,output[i]);
    }

}
```

(j) 乗算

■説明

【書式】 `int VectorMult (short output[], const short ip_x[], const short ip_y[],long no_elements, int res_shift)`

【引数】

output[]	出力
ip_x[]	入力 1
ip_y[]	入力 2
no_elements	データ数
res_shift	各出力に適用される右シフト

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・res_shift < 0 ・res_shift > 16

【内容】 ip_x,ip_y から 1 つずつデータを取り出して乗算を行い、結果を output に配置します。

【備考】 出力は res_shift ビット右シフトした結果の下位 16 ビットを取り出したものとなります。
なお、オーバフローしたときは正または負の最大値となります。
本関数はデータの乗算を行います。内積を計算する場合は m(マトリックス 1 の行数)と p(マトリックス 2 の列数)を 1 に設定して MatrixMult を使用してください。
ip_x は X メモリに、ip_y は Y メモリに、output は任意のメモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  4
#define RSHIFT  15
short y[4] = {1, 32767, -32767, 32767, };
short x[4] = {-1, 32767, -32767, -32767, };

#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section

void main()
{
    short i, n ;
    short output[N];
    int size, rshift;


    /* copy data into X and Y RAM */
    for(i=0;i<N;i++) {
        datx[i] = x[i];
        daty[i] = y[i];
    }

    size = N;
    rshift = RSHIFT;
    for (i = 0; i < N; output[i++] = 0) ;
    if (VectorMult(output, datx, daty, size, rshift) != EDSP_OK) {
        printf("EDSP_OK not returned¥n");
    }

    for(i=0;i<N;i++){
        printf("#%2d  output:%6d  ip_x:%6d  ip_y:%6d ¥n",i,output[i],datx[i],daty[i]);
    }

}
```

インクルードヘッダ

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義します。

データ設定

(k) 平均 2 乗値

■説明

【書式】 `int MsPower (long *output, const short input[], long no_elements, int src_is_x)`

【引数】

output	出力へのポインタ
input[]	入力 x
no_elements	データ数 N
src_is_x	データ配置指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・src_is_x ≠ 0 または 1

【内容】 入力データの平均 2 乗値を求めます。

【備考】

$$平均2乗値 = \frac{1}{N}\sum_{i=0}^{N-1} x(i)^2$$

除算結果は最も近い整数値に丸められます。
演算の結果は 63 ビットで保持されます。no_elements が 2^{32} 以上の場合、オーバフローが発生することがあります。
src_is_x=1 のときは input は X メモリに、src_is_x=0 のときは Y メモリに配置してください。
output は任意のメモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N  5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};

#pragma section X
static short   datx[N];
#pragma section Y
static short   daty[N];
#pragma section

void main()
{
    int    i;
    long   output[1];
    int    src_x;

    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
    src_x = 1;
    if (MsPower(output, datx, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("MsPower:x=%d¥n",output[0]);

    src_x = 0;
    if (MsPower(output, daty, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("MsPower:y=%d¥n",output[0]);
}
```


(l) 平均値

■説明

【書式】 `int Mean (short *mean, const short input[], long no_elements, int src_is_x)`

【引数】
mean	input の平均 $\overline{x}$ へのポインタ
input[]	入力 x
no_elements	データ数 N
src_is_x	データ配置指定

【戻り値】
EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・src_is_x ≠ 0 または 1

【内容】 入力データの平均値を求めます。

【備考】 $\overline{x} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)$

除算結果は最も近い整数値に丸められます。
演算結果は 32 ビットで保持されます。no_elements が 2^{16}-1 よりも大きい場合、オーバフローが発生することがあります。
src_is_x=1 のときは input は X メモリに、src_is_x=0 のときは Y メモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};

#pragma section X
static short   datx[N];
#pragma section Y
static short   daty[N];
#pragma section

void main()
{
   short    i,output[1];
   int      size;
   int      src_x;
   int      flag = 1;

   /* copy data into X and Y RAM */
   for (i = 0; i < N; i++) {
      datx[i] = dat[i];
      daty[i] = dat[i];
   }
   /* test working of stack */
   src_x = 1;
   if (Mean(output, datx, N, src_x) != EDSP_OK){
      printf("EDSP_OK not returned¥n");
   }
   printf("Mean:x=%d¥n",output[0]);

   src_x = 0;
   if (Mean(output, daty, N, src_x) != EDSP_OK){
      printf("EDSP_OK not returned¥n");
   }
   printf("Mean:y=%d¥n",output[0]);
}
```

インクルードヘッダ

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義する。

Xメモリを使用する場合
src_x=1

Yメモリを使用する場合
src_x=0

(m) 平均と偏差

■説明

【書式】 `int Variance (long *variance, short *mean, const short input[], long no_elements, int src_is_x)`

【引数】

variance	入力の偏差σ^2へのポインタ
mean	データの平均$\bar{x}$へのポインタ
input[]	入力x
no_elements	データ数N
src_is_x	データ配置指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・src_is_x ≠ 0 または 1

【内容】 input の平均と偏差を求めます。

【備考】

$$\bar{x} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)$$

$$\sigma^2 = \frac{1}{N}\sum_{i=0}^{N-1} x(i)^2 - \bar{x}^2$$

除算結果は最も近い整数値に丸められます。
$\bar{x}$ は 32 ビットで保持されます。また、オーバフローのチェックはしません。
no_elements が 2^{16}-1 よりも大きい場合、オーバフローが発生することがあります。
σ^2 は 63 ビットで保持されます。オーバフローのチェックはしません。
src_is_x=1 のときは input は X メモリに、src_is_x=0 のときは Y メモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N 5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};

#pragma section X
static short   datx[N];
#pragma section Y
static short   daty[N];
#pragma section

void main()
{
    long       size,var[1];
    short      mean[1];
    int        i ;
    int        src_x;

    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }

    /* test working of stack */
    size = N;
    src_x = 1;
    if (Variance(var, mean, datx, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Variance:%d  mean:%d ¥n ",var[0],mean[0]);

    src_x = 0;
    if (Variance(var, mean, daty, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned¥n");
    }
    printf("Variance:%d  mean:%d ¥n ",var[0],mean[0]);
}
```

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義する。

データ設定

Xメモリを使用する場合
src_x=1

Yメモリを使用する場合
src_x=0

(n) 最大値

■説明

【書式】 `int MaxI (short **max_ptr, short input[], long no_elements, int src_is_x)`

【引数】

max_ptr	最大データへのポインタへのポインタ
input[]	入力
no_elements	データ数
src_is_x	データ配置指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・src_is_x ≠ 0 または 1

【内容】 配列 input の最大値を検索して、そのアドレスを max_ptr に返します。

【備考】 複数のデータが同じ最大値を持つ場合、input の先頭に最も近いデータのアドレスが返されます。
src_is_x=1 のときは input は X メモリに、src_is_x=0 のときは Y メモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
static short dat[131] = {-16384, -32767, 32767, 14877, 8005};

#pragma section X
static short   datx[N];
#pragma section Y
static short   daty[N];
#pragma section

void main()
{
    short    *outp,**outpp;
    int      size,i;
    int      src_x;

 /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
       datx[i] = dat[i];
       daty[i] = dat[i];
    }

    /* MAXI */
    size = N;
    outpp = &outp;
    src_x = 1;

    if (MaxI(outpp, datx, size, src_x) != EDSP_OK){
       printf("EDSP_OK not returned¥n");
    }
    printf("Max:x = %d¥n",**outpp);

    src_x = 0;
    if (MaxI(outpp, daty, size, src_x) != EDSP_OK){
       printf("EDSP_OK not returned¥n");
    }
    printf("Max:y = %d¥n",**outpp);
}
```

インクルードヘッダ

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義する。

データ設定

Xメモリを使用する場合
src_x=1

Yメモリを使用する場合
src_x=0

(o) 最小値

■説明

【書式】
```
int MinI  (short **min_ptr, short input[], long no_elements,
           int src_is_x)
```

【引数】

min_ptr	最小データへのポインタへのポインタ
input[]	入力
no_elements	データ数
src_is_x	データ配置指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・src_is_x ≠ 0 または 1

【内容】 配列 input の最小値を検索して、そのアドレスを min_ptr に返します。

【備考】 複数のデータが同じ最小値を持つ場合、input の先頭に最も近いデータのアドレスが返されます。
src_is_x=1 のときは input は X メモリに、src_is_x=0 のときは Y メモリに配置してください。

■使用例

```
#include <stdio.h>        } インクルードヘッダ
#include <ensigdsp.h>

#define N 10
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};

#pragma section X
static short   datx[N];
#pragma section Y
static short   daty[N];
#pragma section

void main()
{
   short    *outp,**outpp;
   int      size,i;
   int      src_x;

   /* copy data into X and Y RAM */
   for (i = 0; i < N; i++) {
      datx[i] = dat[i];
      daty[i] = dat[i];
   }

   /* MINI */
   /* test working of stack */
   size = N;
   outpp = &outp;

   src_x = 1;
   if (MinI(outpp, datx, size, src_x) != EDSP_OK){
      printf("EDSP_OK not returned¥n");
   }
   printf("Min:x=%d¥n",**outpp);

   src_x = 0;
   if (MinI(outpp, daty, size, src_x) != EDSP_OK){
      printf("EDSP_OK not returned¥n");
   }
   printf("Min:y=%d¥n",**outpp);
}
```

XメモリYメモリに配置する変数はpragma sectionでセクション内に定義する。

データ設定

Xメモリを使用する場合
src_x=1

Yメモリを使用する場合
src_x=0

(p) 最大絶対値

■説明

【書式】 int PeakI (short **peak_ptr, short input[], long no_elements, int src_is_x)

【引数】

peak_ptr	最大絶対値データへのポインタへのポインタ
input[]	入力
no_elements	データ数
src_is_x	データ配置指定

【戻り値】

EDSP_OK	成功
EDSP_BAD_ARG	以下のいずれかの場合です ・no_elements < 1 ・src_is_x ≠ 0 または 1

【内容】 配列 input の最大絶対値を検索して、そのアドレスを peak_ptr に返します。

【備考】 複数のデータが同じ最大絶対値を持つ場合、input の先頭に最も近いデータのアドレスが返されます。src_is_x=1 のときは input は X メモリに、src_is_x=0 のときは Y メモリに配置してください。

■使用例

```
#include <stdio.h>
#include <ensigdsp.h>

#define N 5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};

#pragma section X
static short   datx[N];
#pragma section Y
static short   daty[N];
#pragma section

void main()
{
    short    *outp,**outpp;
    int      size,i;
    int      src_x;

    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }

    size = N;
    outpp = &outp;

    src_x = 1;
    if (PeakI(outpp, datx, size, src_x) != EDSP_OK)
    {
        printf("EDSP_OK not returned¥n");
    }
    printf("Peak:x=%d¥n",**outpp);

    src_x = 0;
    if (PeakI(outpp, daty, size, src_x) != EDSP_OK)
    {
        printf("EDSP_OK not returned¥n");
    }
    printf("Peak:y=%d¥n",**outpp);
}
```

3.14 DSP ライブラリの性能について

(1) DSP ライブラリの実行サイクル数について

DSP ライブラリの実行サイクル数は以下のとおりです。
測定条件はエミュレータ(SH-DSP,60MHz)にて測定、プログラムセクションは X-ROM または Y-ROM に割り付けました。

表 3.34 DSP ライブラリの実行サイクル数一覧表（1）

分類	DSP ライブラリ関数名	実行サイクル数 (Cycle)	備考
高速フーリエ変換	FftComplex	29,330	サイズ: 256 スケーリング: 0xFFFFFFFF
	FftReal	25,490	
	IfftComplex	30,380	
	IfftReal	29,240	
	FftInComplex	26,540	
	FftInreal	25,260	
	IfftInComplex	27,590	
	IfftInReal	27,470	
	LogMagnitude	1,778,290	
	InitFft	3,116,640	
	FreeFft	780	
フィルタ	Fir	23,010	係数の数 : 64 データの数 : 200 収束係数 2μ = 32767
	Fir1	280	
	Lms	97,710	
	Lms1	790	
	InitFir	1,400	
	InitLms	1,400	
	FreeFir	90	
	FreeLms	90	
	Iir	23,530	データ数 : 200 フィルタセクションの数 : 5
	Iir1	360	
	DIir	309,010	
	DIir1	1,860	
	InitIir	280	
	InitDIir	280	
	FreeIir	90	
	FreeDIir	270	
窓関数	GenBlackman	789,950	データ数 : 100
	GenHamming	418,330	
	GenHanning	447,250	
	GenTriangle	744,220	
畳み込み	ConvComplete	21,890	データ数 : 100
	ConvCyclic	14,790	
	ConvPartial	370	
	Correlate	11,930	
	CorrCyclic	15,790	

表 3.34 DSP ライブラリの実行サイクル数一覧表(2)

分類	DSP ライブラリ関数名	実行サイクル数 (Cycle)	備考
その他	Limit	480	データ数 : 100
	CopyXtoY	130	
	CopyYtoX	130	
	CopyToX	1,270	
	CopyToY	1,270	
	CopyFromX	1,320	
	CopyFromY	1,320	
	GenGWnoise	2,878,410	
	MatrixMult	2,337,460	
	VectorMult	1,500	
	MsPower	370	
	Mean	270	
	Variance	820	
	MaxI	540	
	MinI	520	
	PeakI	740	

(2) **C 言語と DSP ライブラリのソースコードの比較**

FFT 関数の一部(バタフライ計算を行っている部分)の関数について C 言語で書かれたものと DSP ライブラリのソースを示します。

DSP ライブラリでは movx,movy,padd などの DSP 特有の命令を使用することで性能 Up を実現しています。

C 言語のソースコード

```
void R4add(short *arp, short *brp, short *aip, short *bip, int grpinc, int numgrp) {
     short  tr,ti;
     int    grpind;

     for(grpind=0;grpind<numgrp;grpind++) {
              tr =  *brp;
              ti =  *bip;
              *brp = sub(*arp,ti);
              *bip = add(*aip,tr);
              *arp = add(*arp,ti);
              *aip = sub(*aip,tr);
              arp += grpinc;
              aip += grpinc;
              brp += grpinc;
              bip += grpinc;
     }
}
```

DSP ライブラリのソースコード

```
_R4add:

        MOV.L   Ix,@-R15
        MOV.L   Iy,@-R15

        MOV.L   @(2*4,R15),Ix
        SHLL    Ix
        MOV     Ix,Iy
        MOV.L   @(3*4,R15),R1

        REPEAT  r4alps,r4alpe
        ADD     #-1,R1
        SETRC   R1
                                      movx.w  @ar,X0      movy.w @bi,Y0
        padd    X0,Y0,A0
        psub    X0,Y0,A1              movx.w  @br,X0      movy.w @ai,Y0
        padd    X0,Y0,A0              movx.w  A0,@ar+Ix
```

```
         pneg    X0,X0                   movx.w  A1,@br+Ix
         padd    X0,Y0,A1                movy.w A0,@bi+Iy
         movx.w  @ar,X0                  movy.w @bi,Y0
         .ALIGN 4
r4alps   padd    X0,Y0,A0                movy.w A1,@ai+Iy
         psub    X0,Y0,A1                movx.w  @br,X0     movy.w @ai,Y0
         padd    X0,Y0,A0                movx.w  A0,@ar+Ix
                         pneg   X0,X0    movx.w  A1,@br+Ix
         padd    X0,Y0,A1                movy.w A0,@bi+Iy
r4alpe                                   movx.w  @ar,X0     movy.w @bi,Y0
                                         movy.w A1,@ai+Iy
         MOV.L   @R15+,Iy
         RTS
         MOV.L   @R15+,Ix
```

(3) FFT の各関数についての性能

フーリエ変換の各関数は以下のように分類されます。

表 3.35 高速フーリエ変換

	Not-in-place 方式	In-place 方式
複素数フーリエ変換	FftComplex	FftInComplex
実数フーリエ変換	FftReal	FftReal

表 3.36 逆高速フーリエ変換

	Not-in-place 方式	In-place 方式
複素数フーリエ変換	IfftComplex	IfftInComplex
実数フーリエ変換	IfftReal	IfftInReal

■ **in-place 方式と not-in-place 方式の違いについて**

in-place 方式は入力データとの配列をそのまま出力データの配列として使用する方式です。したがって入力データは出力データによって上書きされるため保存されません。

not-in-place 方式は入力データと出力データを別々に用意して関数呼び出しを行う方式です。入力データと出力データが別々なので関数を呼び出した後も入力データはそのまま保存されます。

in-place 方式と not-in-place 方式の性能上の差はほとんどないのでメモリの使用量に従って使用する関数を決定してください。

in-place 方式では not-in-place 方式に比べて使用するメモリ量が半分で済みます。

■ **スケーリングについて**

FFT 計算の各段階において、計算は積和の形式で実行されるためオーバフローが起きやすくなっています。オーバフローが起きるとすべて最大値または最小値となるため計算結果を正しく評価することができません。

スケーリングはそのオーバフローを防ぐために FFT 計算の各段階において、どの程度 2 で割る（右シフトを行う）かを表す目安です。

表 3.37 スケーリング値とその特長

スケーリング値	特長
FFTNOSCALE	まったくシフトしない。オーバフローが起きやすい
EFFTMIDSCALE	段階の 1 つおきにシフトする。
EFFTALLSCALE	すべての段階でシフト処理を行う。オーバフローが起きにくい。

スケーリングの性能に及ぼす影響は大きくありません。したがって、スケーリングを決定する場合は性能ではなく、データの特性から決定してください。

(4) フィルタの各関数について

■ **Fir と Lms の使用方法**

Firフィルタと Lmsフィルタの係数の数とサイクル数の関係を**図**3.11に示します。

Lmsの計算速度は適応アルゴリズムを使用しているため、Firよりも遅くなります。データの波形が安定しているシステムでは、Lmsはフィルタ係数を決定するために使用し、決定した後はFirフィルタに置き換えてください。

データのスケーリングのために、右シフトの数を指定することができます。SH-DSPライブラリの内部で積和演算を行いますので、データによってはオーバフローが起こります。その場合は右シフト数を適当に変更して、出力される値を参考にしながら決定してください。

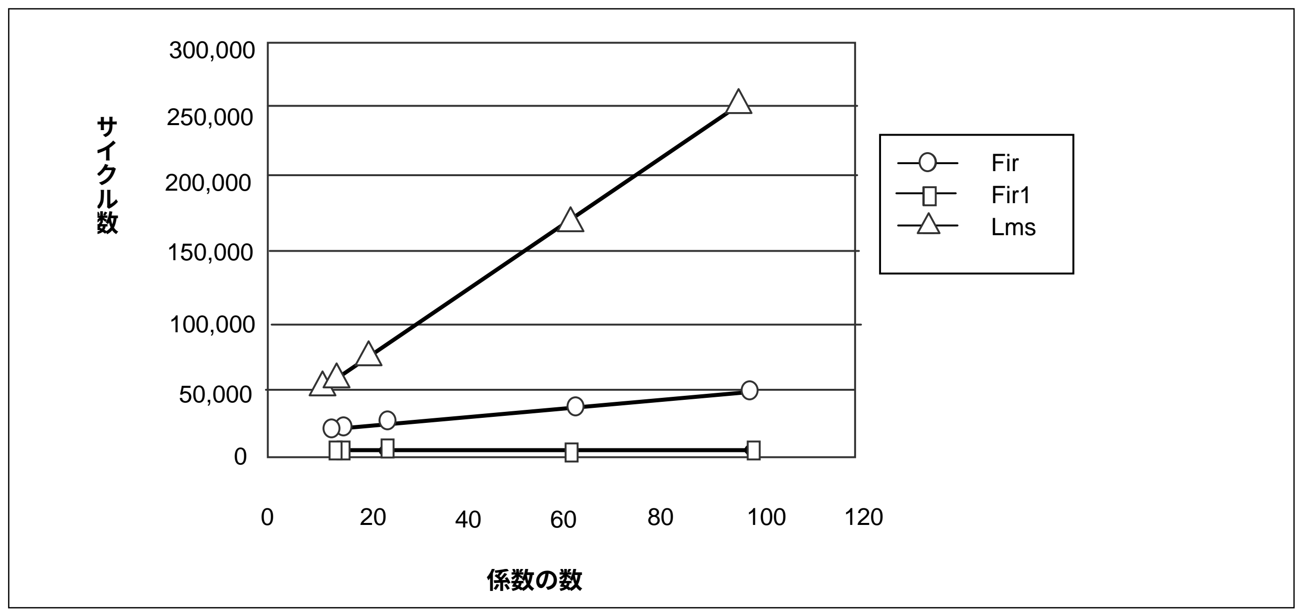


図 3.11 **係数の数とサイクル数の関係**

■ Iir と DIir について

性能を優先する場合は DIir ではなく、Iir を使用してください。SH-DSP ライブラリの内部で積和演算を行いますのでデータによってはオーバフローが起こります。その場合には右シフト数を適当に変更して、出力される値を見ながら決定してください。
データのスケーリングのために、右シフトの数を指定することができます。ただし、右シフト数の指定方法はフィルタ係数の配列の一部として設定します。詳細は「3.13.6(5)(c)IIR,(e)倍精度 IIR」を参照してください。

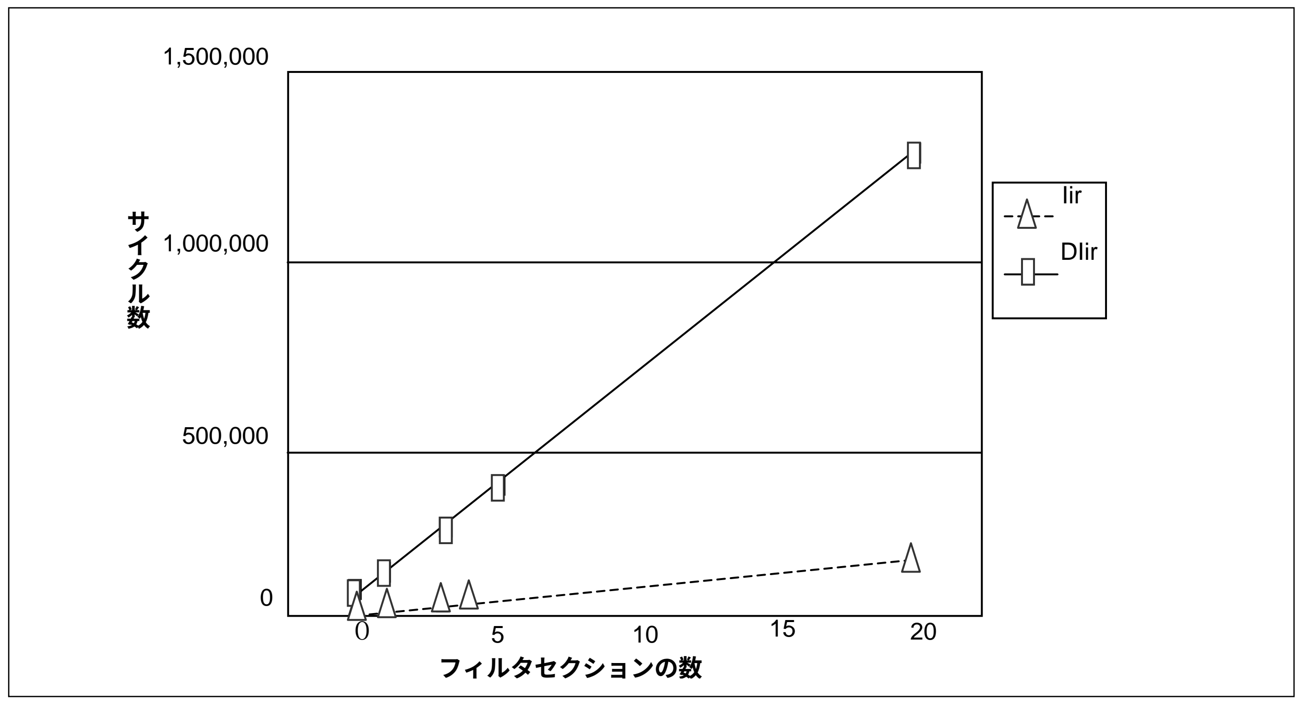


図 3.12　フィルタセクションの数とサイクル数の関係

■ フィルタ関数の使い分け

Fir フィルタは直線位相応答の特長を持ち、常に安定しているので位相の歪みが許されないオーディオ、ビデオなどのアプリケーションに適しています。一方、Iir フィルタはフィードバックを持つフィルタであり、Fir よりも少ない係数で結果を得ることができ、高速なので、時間制約のあるような場合に適しています。しかし、Iir フィルタは不安定になることがあるので使用に際しては十分注意が必要です。

3.15 クロスソフト間の関連

3.15.1 アセンブリ言語プログラムとの関連

SuperH RISC engine C/C++コンパイラは、ルネサステクノロジ SuperH RISC engine ファミリ専用の特殊命令までサポートしているため、ほとんどのプログラムは C 言語で記述できます。しかし、性能を追求する場合、鍵になる部分をアセンブリ言語で記述し、C 言語プログラムと接続する必要があります。

本節では、C 言語プログラムとアセンブリ言語プログラムの接続時に注意すべき次の事項について概説します。

- 外部名の相互参照方法
- 関数呼び出しのインタフェース

詳細については、「SuperH RISC engine **C/C++コンパイラ、アセンブラ、最適化リンケージエディタ ユーザーズマニュアル**」を参照してください。

(1) 外部名の相互参照方法

(a) アセンブリ言語プログラムの外部定義名を C 言語プログラムで参照する方法

C 言語プログラムからアセンブリ言語プログラムの外部定義名を参照するには、次のようにします。

- アセンブリ言語プログラムで先頭に"_"を付けたシンボル名（32 文字以内）を".EXPORT"または".GLOBAL"アセンブラ制御命令を用いて外部定義宣言します。
- C 言語プログラムでは"extern"記憶クラス指定子を用いて先頭に"_"がないシンボル名を外部参照宣言します。

アセンブリ言語プログラム（定義する側）

```
      .EXPORT          _a , _b
      .SECTION   D, DATA, ALIGN=4
_a : .DATA.L      1
_b : .DATA.L      1
      .END
```

C 言語プログラム（参照する側）

```
extern   int  a , b;
 f ()
 {
       a+=b;
 }
```

図 3.13 アセンブリ言語プログラムの外部定義名を C 言語プログラムから参照する例

(b) C 言語プログラムの外部定義名をアセンブリ言語プログラムで参照する方法

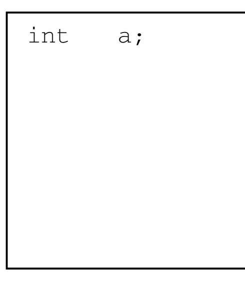
C 言語プログラムにとっての外部定義名とは次のものです。

- 大域変数であって、かつ static 記憶クラスでないもの
- extern 記憶クラスで宣言されている変数名
- static 記憶クラスを指定されていない関数名

アセンブリ言語プログラムから C 言語プログラムの外部定義名を参照するには、次のようにします。

- C 言語プログラムでシンボル名（先頭に_がない）を外部定義（グローバル変数）します。
- アセンブリ言語プログラムでは".IMPORT "または".GLOBAL "アセンブラ制御命令を用いて先頭に"_"を付加したシンボル名を外部参照宣言します。

C 言語プログラム（定義する側）



```
int    a;
```

アセンブリ言語プログラム（参照する側）

```
           .IMPORT    _a
           .SECTION   P,CODE,
ALIGN=2
           MOV.L          A_a, R1
           MOV.L          @R1, R0
           ADD            #1,R0
           RTS
           MOV.L          R0,@R1
A_a:    .DATA.L    _a
           .END
```

図 3.14 C 言語プログラムの外部定義名をアセンブリ言語プログラムから参照する例

【注】 関数名、静的データメンバから生成する外部名は、C++コンパイルのとき一定の規則で変換を行っています。コンパイラが生成した外部名を知る必要があるときは、コンパイラオプション code=asm または、listfile にてコンパイラが生成する外部名を参照してください。また、C++の関数を「extern "C"」を付与して関数定義を行えば、外部名は C の関数と同様の生成規則になります。ただし、その関数を多重定義できなくなります。

(2) 関数呼び出しのインタフェース

C 言語プログラムとアセンブリ言語プログラムとの間で相互に関数呼び出しを行うとき、アセンブリ言語プログラム側で守るべき規則には、次の 4 つがあげられます。

- （i）スタックポインタに関する規則
- （ii）スタックフレームの割り付け／解放に関する規則
- （iii）レジスタに関する規則
- （iv）引数とリターン値の設定／参照に関する規則

ここでは、(i)～(iii)までを説明します。(iv)については、「3.15.1 (3)**引数とリターン値の設定／参照**」を参照してください。

(a) スタックポインタに関する規則

スタックポインタの指すアドレスよりも下位（0 番地の方向）のスタック領域に、有効なデータを格納しないでください。スタックポインタより下位アドレスに格納されたデータは、割り込み処理で破壊される可能性があります。

(b) スタックフレームの割り付け／解放に関する規則

関数呼び出しが行なわれた時点（JSR または BSR 命令の実行直後）では、スタックポインタは呼び出し関数側で使用したスタックの最下位アドレスを指しています。この領域より上位アドレス（H'FFFFFFFF 番地の方向）のデータの割り付け／設定は呼び出し側の関数の役目です。

関数のリターン時は、呼び出された関数で確保した領域を解放してから、通常 RTS 命令を用いて呼び出し関数へ返ります。これより上位アドレスの領域（リターン値アドレスおよび引数の領域）は、呼び出し側の関数で解放します。

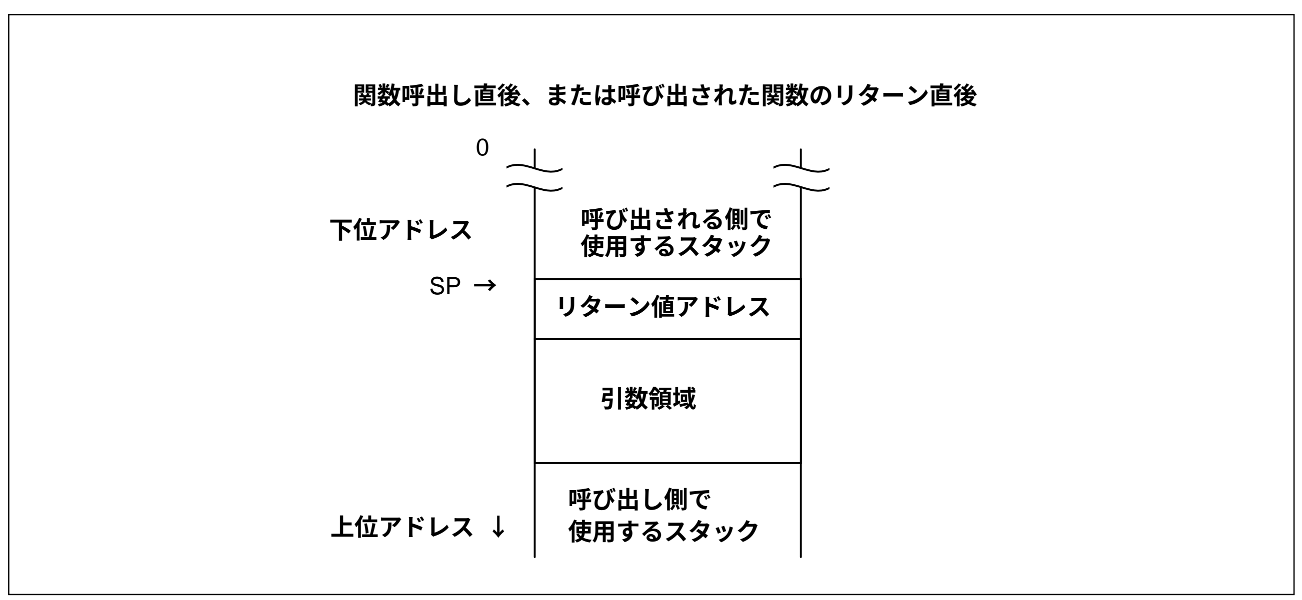


図 3.15 **スタックフレームの割り付け／解放**

(c) レジスタに関する規則

C/C++コンパイラが関数呼び出し前後において、値を保証するレジスタと保証しないレジスタがあります。レジスタの保証規則を**表** 3.38 に示します。

表 3.38　C **言語プログラムでの関数呼び出し前後のレジスタ保証規則**

項番	項目	対象レジスタ	アセンブリ言語プログラミングにおける注意点
1	保証しない レジスタ	R0～R7 FR0～FR11*1 DR0～DR10*2 FPUL*1,*2,FPSCR*1,*2 A0*3, A0G*3, A1*3,A1G*3, M0*3, M1*3, X0*3,X1*3, Y0*3, Y1*3, DSR*3, MOD*3, RS*3, RE*3	**関数呼び出し時に対象レジスタに有効な値があれば、呼び出し側で値を退避する。呼び出される側の関数では退避せずに使用できる。ただし、**fpscr=safe **を指定した場合は**FPSCR **は保証するレジスタ。**
2	保証する レジスタ	R8～R15 MACH,MACL,PR FR12～FR15*1 DR12～DR14*2	**対象レジスタのうち関数内で使用するレジスタの値を退避し、リターン時に回復する。ただし、**macsave = 0**オプション指定時は**MACH、MACL**は保証しないレジスタ。また、**gbr=auto **を指定した場合は**GBR **は保証するレジスタ。**

【注】 *1　SH-2E,SH2A-FPU,SH-4,SH-4A **の単精度浮動小数点用レジスタです。**
*2　SH2A-FPU,SH-4,SH-4A **の倍精度浮動小数点用レジスタです。**
*3　SH2-DSP,SH3-DSP,SH4AL-DSP **の** DSP **レジスタです。**

C言語プログラムとアセンブリ言語プログラムの関数の接続は、次のようにしてください。

(i)　**アセンブリ言語関数を** C **言語プログラムから呼び出す場合**

- 対象アセンブリ言語関数が別のモジュールを呼び出しているときには、アセンブリ言語関数の入口で PR レジスタの値のスタックへの退避、出口でスタックからの回復を行ってください。
- アセンブリ言語関数内で R8～R15、MACH、MACL のレジスタを使用するときには、使用前にレジスタ値のスタックへの退避、使用後にスタックからの回復を行ってください。
- アセンブリ言語関数へどのように引数が渡されるかについては「3.15.1 (3) **引数とリターン値の設定／参照**」を参照してください。

(ii)　C **言語関数をアセンブリ言語プログラムから呼び出す場合**

- R0～R7 レジスタに有効な値があれば、C 言語関数呼び出し前に空きレジスタまたはスタックへ値を退避してください。
- アセンブリ言語関数へどのようにリターン値が渡されるかについては「3.15.1 (3) **引数とリターン値の設定／参照**」を参照してください。

C言語関数 f からアセンブリ言語関数 g を呼び出し、さらにアセンブリ言語関数 g から C 言語関数 h を呼び出している例を**図** 3.16 に示します。

C言語関数 f

```
extern void
   g( );

f( )
{
      g( );
}
```

アセンブリ言語関数 g

```
          .EXPORT        _g            関数gの外部定義宣言
          .IMPORT        _h            関数hの外部参照宣言
          .SECTION       P, CODE,
ALIGN=2                                PRレジスタ値の保存
_g :   STS.L             PR ,@-R15     関数gでの使用レジスタの保存
     MOV.L           R14,@-R15
     MOV.L           R13,@-R15
       :                               関数hでの使用レジスタの保存
          MOV.L             R2,@R15
          MOV.L             R1,@R15    関数hの呼び出し
     MOV.L           L_h,R0
     JSR                 @R0
     NOP
       :                               関数gでの使用レジスタの回復
     MOV.L           @R15+,R13
     MOV.L           @R15+,R14
     RTS                               PRレジスタ値の回復
     LDS.L           @R15+,PR

L_h :  .DATA.L       _h
       .END
```

C 言語関数 h

```
h( )
{
      :
      :
}
```

図 3.16　C 言語プログラム関数とアセンブリ言語プログラム関数の相互呼び出し例

(3)　引数とリターン値の設定／参照

C/C++コンパイラの引数とリターン値の設定／参照に関する規則は、関数宣言において個々の引数とリターン値の型が明示的に宣言されているかどうかによって異なります。C 言語プログラムで引数とリターン値の型を明示的に宣言するには、関数の原型宣言を用います。

以下の説明では、まず C 言語プログラムでの引数とリターン値に対する一般的な規則について述べたあと、引数の割り付け領域と割り付け方、およびリターン値の設定場所について述べます。

(a)　C 言語プログラムでの引数とリターン値に対する一般的な規則

(i)　引数の渡し方

引数の値を、必ずレジスタまたはスタック上の引数の割り付け領域にコピーしたあとで関数を呼び出します。呼び出し側の関数では、リターン後に引数の割り付け領域を参照することはありませんので、呼び出された側の関数で引数の値を変更しても呼び出し側の処理は直接には影響を受けません。

(ii)　型変換の規則

引数を渡す場合、またはリターン値を返す場合、自動的に型変換を行うことがあります。
型変換の規則について**表** 3.39 に示します。

表 3.39　型変換の規則

型変換	変換方法
型の宣言された引数の型変換	原型宣言によって型が宣言されている引数は、宣言された型に変換します。
型の宣言されていない引数の型変換	原型宣言によって型が宣言されていない引数は、次の規則に従って変換します。 ・char 型、unsigned char 型、short 型、unsigned short 型の引数は、int 型に変換します。 ・float 型の引数は、double 型に変換します。 ・上記以外の型は変換しません。
リターン値の型変換	リターン値は、その関数の返す型に変換します。

例 1)　原型宣言により型が宣言されている例

```
long f();
long f()
{
      float x;
            :
      return x;
}
```

リターン値 x は、原型宣言に従って long 型に変換されます。

例 2)　原型宣言により型が宣言されていない例 1

```
void p(int,...);

long f()
{
     char c;
          :
     p(1.0, c);
          :
}
```

第 1 引数は、対応する引数の型が int 型なので、int 型に変換されます。
第 2 引数は、対応する引数の型がないので、int 型に変換されます。

例 3)　原型宣言により型が宣言されていない例 2

原型宣言によって引数の型を宣言していない場合、正しく引数が渡されるように呼び出される側と呼び出す側で同じ型に指定してください。型があっていない場合は、動作が保証されません。

```
void f(x)
float x;
{
        :
}

void main()
{
     float x;
     f(x);
}
```

この例では、関数 f の引数の原型宣言がないため、関数 main の側で呼び出すときに引数 x を double 型に変換します。一方、関数 f の側では引数を float 型として宣言しています。このため、正しく引数を受け渡しすることはできません。原型宣言によって引数の型を宣言するか、関数 f の側の引数宣言を double 型にする必要があります。

原型宣言によって正しく引数の型を宣言すると次のようになります。

```
void f(float x)
{
        ⋮
}

void main()
{
     float x;
     f(x);
}
```

(b) C 言語プログラムでの引数の割り付け方

引数は、レジスタに割り付ける場合とレジスタに割り付けられないときスタック上の引数領域に割り付ける場合があります。引数の割り付け領域を**図** 3.17 に、引数割り付け領域の一般規則を**表** 3.40 にそれぞれ示します。

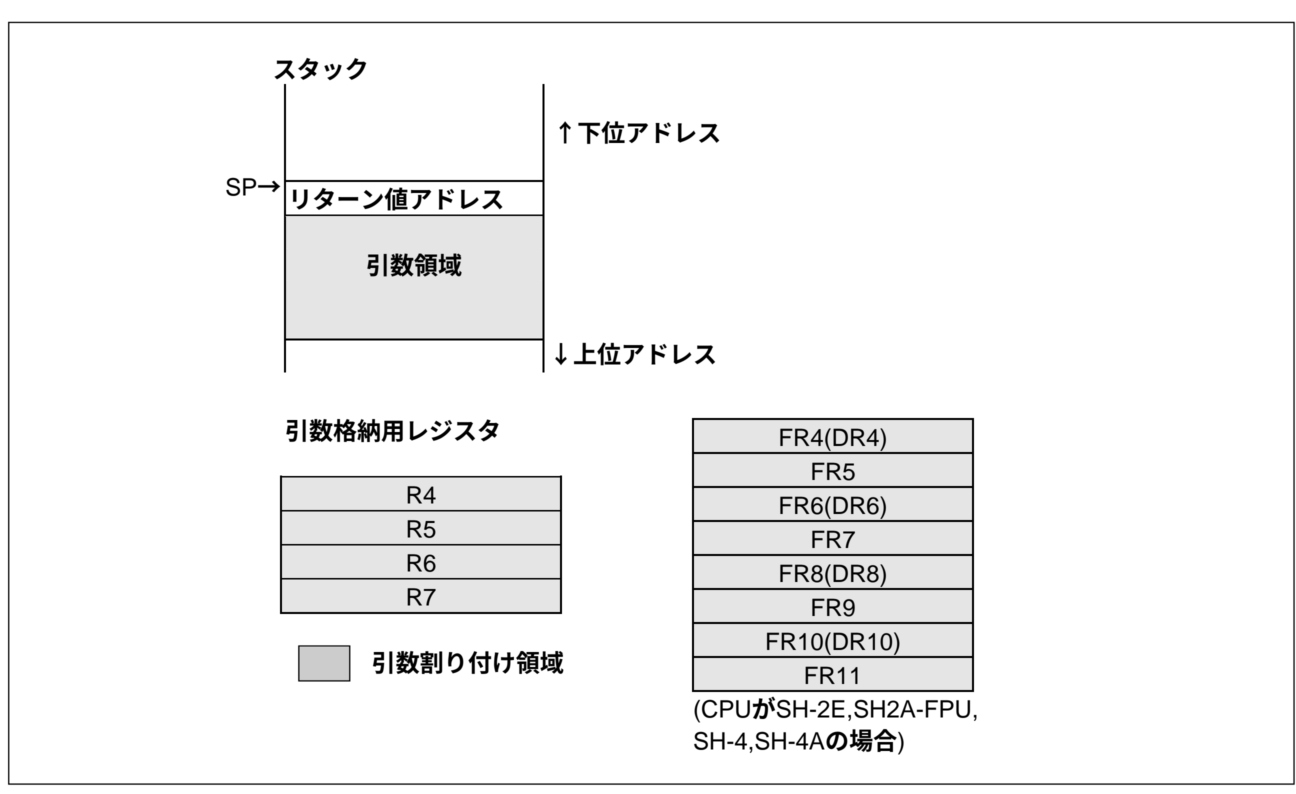


図 3.17 C 言語プログラムでの引数割り付け領域

表 3.40 C 言語プログラムでの引数割り付けの一般規則

<table>
<tr><th colspan="3">割り付け規則</th></tr>
<tr><th colspan="2">レジスタで渡される引数</th><th rowspan="2">スタックで渡される引数</th></tr>
<tr><th>引数格納用
レジスタ</th><th>対象の型</th></tr>
<tr><td>R4〜R7</td><td>char, unsigned char, bool,
short, unsigned short,
int, unsigned int,
long, unsigned long,
float(CPU が SH-2E,SH2A-FPU,
SH-4,SH-4A 以外の場合),ポインタ,
データメンバへのポインタ,リファレンス</td><td rowspan="3">(1)引数の型がレジスタ渡しの対象の型以外のもの
(2)プロトタイプ宣言により可変個の引数を持つ関数として宣言しているもの*3
(3)他の引数がすでに R4〜R7 に割り付いている場合
(4)他の引数がすでに FR4(DR4)〜FR11(DR10)に割り付いている場合
(5) long long,unsigned long long 型の引数
(6) _ _fixed 型,long _ _fixed 型,_ _accum 型,long _ _accum 型の引数</td></tr>
<tr><td>FR4〜FR11*1</td><td>SH-2E のとき
・引数が float 型
・引数が double 型かつ
double=float オプション指定
SH2A-FPU,SH-4,SH-4A のとき
・引数型が float 型かつ
fpu=double オプション指定なし
・引数型が double 型かつ
fpu=single オプション指定</td></tr>
<tr><td>DR4〜DR10*2</td><td>SH2A-FPU,SH-4,SH-4A のとき
・引数型が double 型かつ
fpu=single オプション指定なし
・引数型が float 型かつ
fpu=double オプション指定</td></tr>
</table>

【注】 *1 SH-2E,SH2A-FPU,SH-4,SH-4A の単精度浮動小数点用のレジスタです。
*2 SH2A-FPU,SH-4,SH-4A の倍精度浮動小数点用レジスタです。
*3 プロトタイプ宣言により可変個の引数を持つ関数として宣言している場合、宣言の中で対応する型のない引数およびその直前の引数はスタックに割り付けます。

例)

```
  int f2(int, int, int, int,...);
f2(a, b, c, x, y, z)
{
              :
}
```

第 4 引数までは通常、レジスタに割り付けますが、ここでは x,y,z もスタックに割り付けます。

(i) **引数格納用レジスタへの割り付け**
引数格納用レジスタには、ソースプログラムの宣言の順に番号の小さいレジスタから割り付けます。引数格納用レジスタの割り付け例を例1に示します。

(ii) **スタック上の引数領域への割り付け**
スタック上の引数領域には、ソースプログラム上で宣言した順に下位アドレスから割り付けます。引数格納用スタックの割り付け例を例2〜例8に示します。

【構造体、共用体型引数に関する注意】
構造体、共用体型の引数を設定する場合は、その型の境界調整にかかわらず 4 バイト境界に割り付けられ、しかもその領域として 4 の倍数バイトの領域が使用されます。これは、SuperH マイコンのスタックポインタが 4 バイト単位に変化するためです。

例 1) **レジスタの対象の型である引数は、宣言順にレジスタ** R4〜R7 **に割り付けます。**

```
int f(char,short,int,float);
    :
f(1,2,3,4.0);
    :
```

R4	保証しない	1
R5	保証しない	2
R6	3	
R7	4.0	

例 2) **レジスタに割り付けることができなかった引数は、スタックに割り付けます。また、引数の型が（**(unsigned) char **型、または、**(unsigned) short **型でスタック上の引数領域に割り付けられる場合、**4 **バイトに拡張して割り付けれらます。**

```
int f(int,short,long,float,char);
    :
f(1,2,3,4.0,5);
    :
```

R4	1	
R5	保証しない	2
R6	3	
R7	4.0	

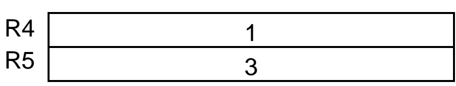
↑下位アドレス

引数領域（スタック）	保証しない	5

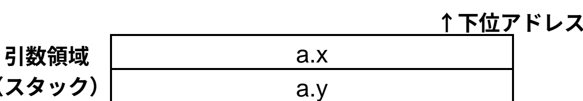
↓上位アドレス

例 3) **レジスタに割り付けられない型の引数は、スタックに割り付けます。**

```
struct s{int x,y;}a;
int f(int,struct s,int);
    :
f(1,a,3);
    :
```

R4
R5


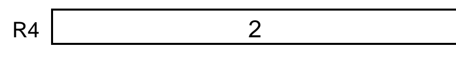
↑下位アドレス

引数領域（スタック）


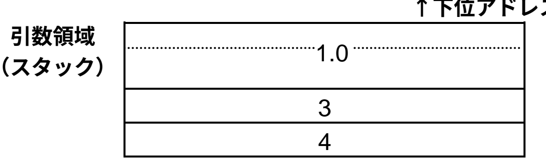
↓上位アドレス

例 4) **プロトタイプ宣言により可変個の引数を持つ関数として宣言している場合、対応する型のない引数およびその直前の引数は、宣言順にスタックに割り付けます。**

```
int f(double, int, int...)
    :
f(1.0, 2, 3, 4)
    :
```

R4


↑下位アドレス

引数領域（スタック）


↓上位アドレス

RENESAS

例 5) **原型宣言がない場合**

→ char 型は int 型に、float 型は double 型に拡張して割り付ける

```
int f ( )
char  a ;
float  b;

      f (a ,b)
```

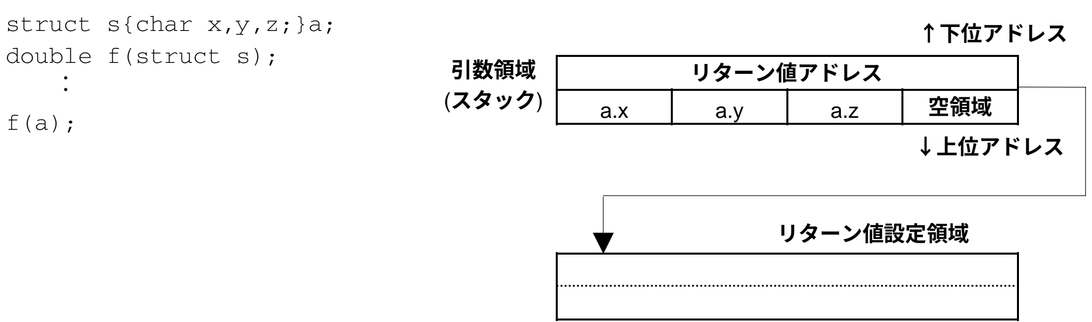
R4 a

↑下位アドレス

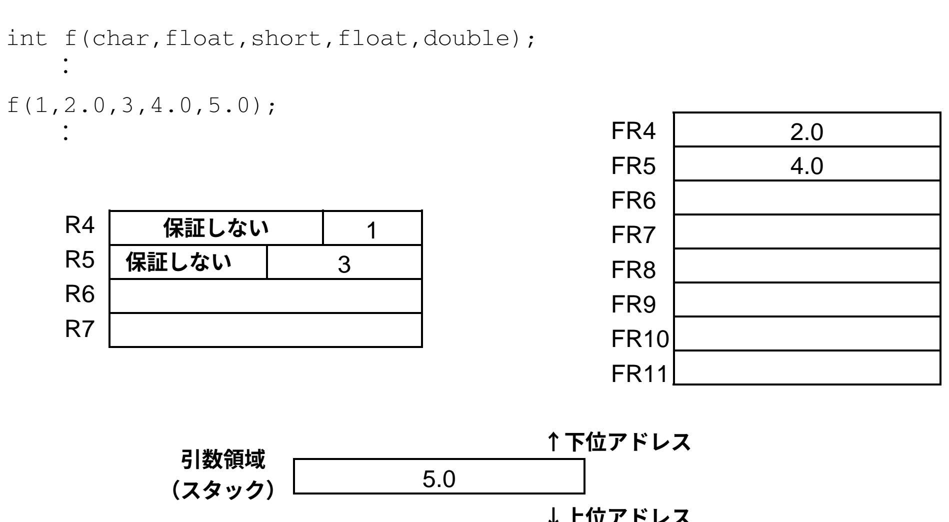
引数領域（スタック） b

↓上位アドレス

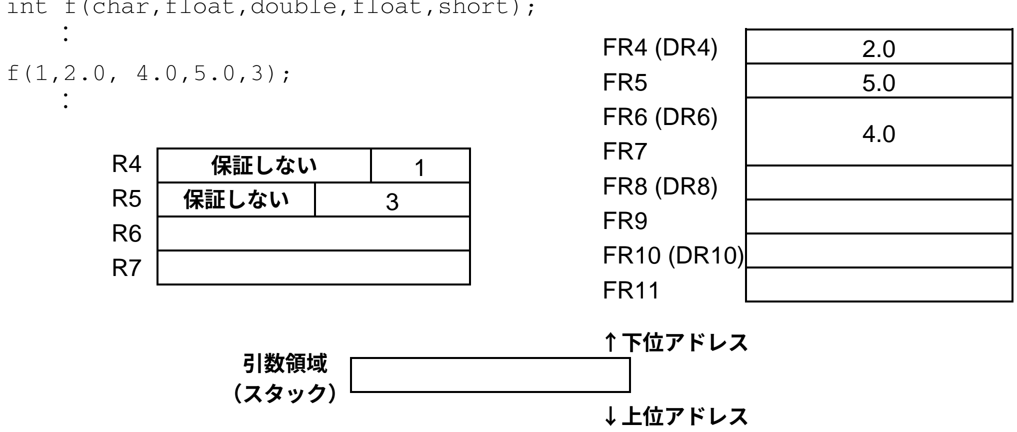
例 6) **関数の返す型が** 4 **バイトを超える場合またはクラスの場合、引数領域の直前にリターン値アドレスを設定します。また、クラスのサイズが** 4 **の倍数バイトでないとき、空領域が生じます。**



例 7) CPU **が** SH-2E **の場合、**float **型の引数は** FPU **レジスタに割り付けられます。**



例 8) CPU **が** SH2A-FPU,SH-4,SH-4A **かつ**-fpu **オプション指定なしの場合、**float/double **型の引数は** FPU **レジスタに割り付けられます。**



(c) **C 言語プログラムでのリターン値の設定場所**

関数のリターン値の型により、リターン値をレジスタに設定する場合とスタックに設定する場合があります。リターン値の型と設定場所の関係は**表** 3.41 を参照してください。

関数のリターン値をスタックに設定する場合、リターン値はリターン値アドレスの指す領域に設定します。呼び出す側では、引数領域のほかにリターン値設定領域を確保し、そのアドレスをリターン値アドレスに設定してから関数を呼び出します（**図** 3.18 **参照**）。関数のリターン値が void 型の場合、リターン値を設定しません。

表 3.41 **C 言語プログラムでのリターン値の型と設定場所**

No	リターン値の型	リターン値の設定場所
1	(singed) char, unsigned char, (singed) short, unsigned short, (singed) int, unsigned int, long, unsigned long, float, **ポインタ**, bool, **リファレンス、データメンバへのポインタ**	R0 : 32 **ビット** (signed) char,unsigned char **の上位** 3 **バイト、**(signed) short, unsigned short **の上位** 2 **バイトの内容は保証しません。**(**ただし、**-rtnext **オプション指定時は**(signed) char,(singed) short **型は符号拡張、**unsigned char,unsigned short **型はゼロ拡張を行います。**) FR0 : 32 **ビット** (1)SH-2E **のとき** ・**リターン値が** float **型** ・**リターン値が** double **型かつ** double=float **オプション指定** (2)SH2A-FPU,SH-4,SH-4A **のとき** ・**リターン値が** float **型かつ** fpu=double **オプション指定なし** ・**リターン値が浮動小数点型かつ** fpu=single **オプション指定**
2	double, long double **構造体、共用体、クラス型、** **関数メンバへのポインタ**	**リターン値設定領域**(**メモリ**) DR0:64 **ビット** SH2A-FPU,SH-4,SH-4A **のとき** ・**リターン値が** double **型かつ** fpu=single **オプション指定なし** ・**リターン値が浮動小数点型かつ** fpu=double **オプション指定**
3	(signed)long long,unsigned long long	**リターン値設定領域（メモリ）**
4	_ _fixed,long _ _fixed, _ _accum, long __accum	**リターン値設定領域（メモリ）**

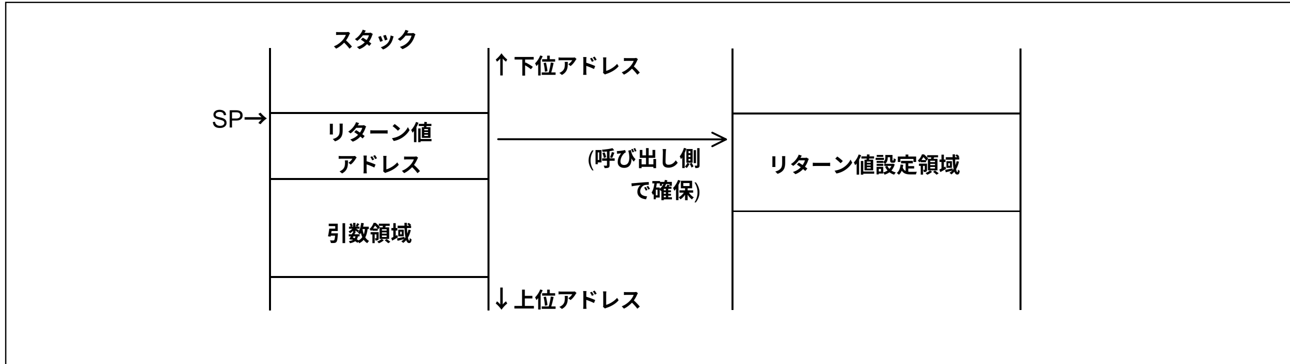


図 3.18　C 言語プログラムでリターン値をスタックに設定する場合のリターン値の設定領域

3.15.2 最適化リンケージエディタとの関連

(1) ROM 化支援機能

ロードモジュールを ROM に書き込む際、初期化データ領域も ROM に書き込まれてしまいます。しかし、データ操作は RAM 上で行わなければならないので、起動時に初期化データ領域の ROM から RAM への複写が必要です。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、この処理を容易に実行できます。

ROM 化支援機能を使用するには、リンク時にオプション"ROM=D=R"（D：ROM 上の初期化データ領域のセクション名、R：RAM 上の初期化データ領域のセクション名）を指定します。

ROM 化支援機能では、次の事項が行われます。

(a) **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。図 3.19 にメモリへの二重割り付け方を示します。**

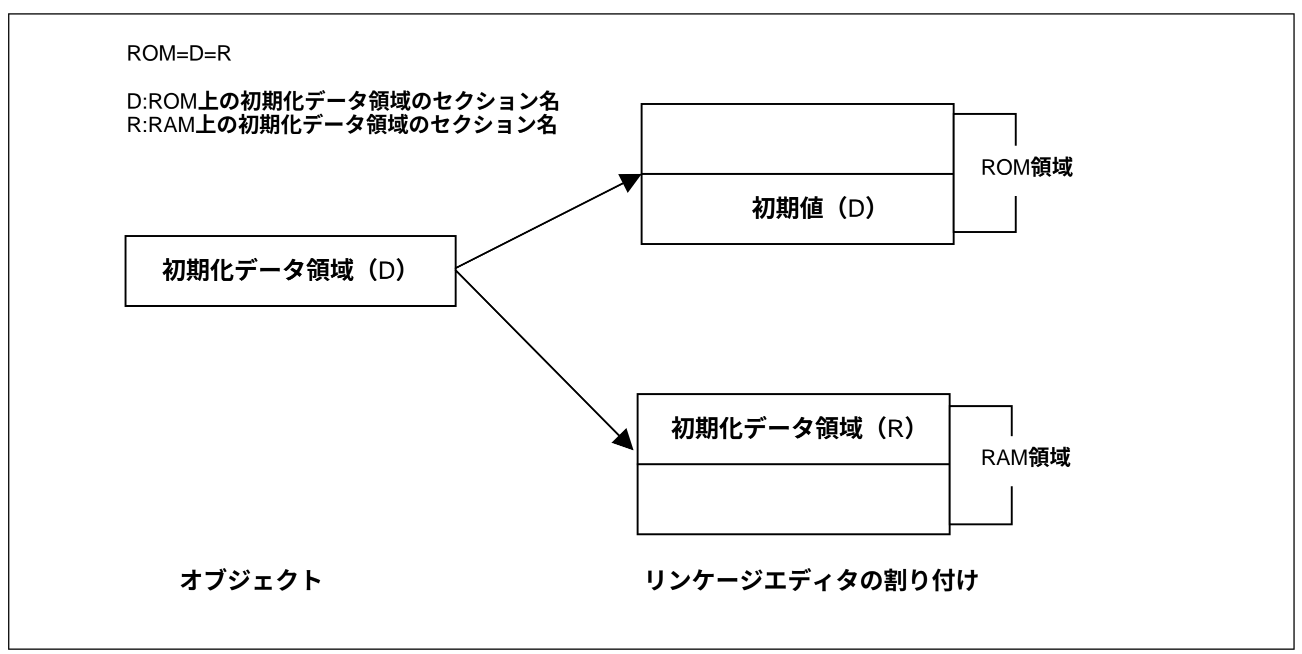


図 3.19 ROM 化支援機能によるメモリ割り付け

(b) **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。**

ユーザはスタートアップルーチンに ROM 上のデータを RAM 上に複写する処理を組み込んでおきます。例については「2.2.4 **初期化部の作成**」を参照してください。

ROM 化支援機能の詳細については、「SuperH RISC engine C/C++**コンパイラ、アセンブラ、最適化リンケージエディタユーザーズマニュアル**」を参照してください。

なお、本機能は H シリーズリンケージエディタ Ver.4 以降でサポートしています。

(2) リンク時の注意事項

C/C++コンパイラが生成したリロケータブルオブジェクトファイルをリンクした際、出力されるエラーメッセージへの対処方法を**表 3.42** に示します。

表 3.42 リンク時のエラーメッセージへの対処方法

項番	現象	確認内容	対処方法
1	リンク時にエラーNo. L1100 (314)* cannot found section が出力される。	リンケージエディタの start オプションにおいて、コンパイラ出力のセクション名を大文字で指定しているか。	正しいセクション名を指定してください。
2	リンク時にエラーNo. L1160 (105)* undifined external symbol が出力される。	C/C++プログラムとアセンブリプログラム間で変数を相互参照している場合、アセンブリプログラム内で下線を付加しているか。	正しい変数名で参照してください。
		C/C++プログラムで C ライブラリ関数を使用していないか。	リンク時に入力ライブラリとして標準ライブラリを指定してください。
		未定義参照シンボル名が__で始まっていないか。 （標準ライブラリ中の実行時ルーチンを使用しています）	
		C ライブラリ関数の標準入出力ライブラリを使用していないか。	低水準インタフェースルーチンを作成してリンクしてください。
3	C/C++ソースレベルデバッグができない。	コンパイル、リンク時に debug オプションを指定したか。	コンパイル、リンク時に debug オプションを指定してください。リンク時に sdebug オプションを指定した場合は、デバッガにデバッグ情報ファイルもロードしてください。
		リンケージエディタの Ver.5.3 以上を使用しているか。	リンケージエディタの Ver.5.3 以上を使用してください。
4	リンク時に、エラーNo. L2330 (108)* relocation size overflow が出力される。	GBR ベース変数の指定で、指定した変数のオフセットは制限内におさまっているか。	制限を超えるデータに対し、#pragma gbr_base/ gbr_base1 宣言を削除してください。
5	リンク時に、エラーNo. L2300 (104)* duplicate symbol が出力される。	同じ名称の変数または関数を複数のファイル内で外部定義していないか。	名前を変更するかまたは static を指定してください。
		複数のファイルでインクルードされるヘッダファイル内で変数または関数を外部定義していないか。 (#pragma inline/ inline_asm 指定した関数でも同様です)	static を指定してください。

【注】*括弧の前はリンカ Ver.7 以降、括弧内は Ver.6 までのエラー番号を示します。

3.15.3 シミュレータ・デバッガとの関連

ロードモジュールをシミュレータ・デバッガを用いて実行した場合、"MEMORY ACCESS ERROR"を発生する可能性があります。安全のため、下記の回避方法のどちらかを適用してください。

(a) シミュレータ・デバッガ使用時も実機と同様のメモリをマッピングする（必ず1つのセクションの総バイト数は4の倍数になる）。

(b) リンク時、Pセクションを除くすべてのセクションの後に下記のアセンブリ言語プログラムから作成するダミーセクションをリンクする。

アセンブリ言語プログラム

```
.SECTION DM,DUMMY,ALIGN=1
.RES.B    3
.END
```

リンク時の結合例

コマンドオプションの場合

```
-START=P,C,DM/0400,B,DM,D,DM/01000000
```

サブコマンドファイルの場合

```
START P,C,DM/0400,B,DM,D,DM/01000000
```

シミュレータ・デバッガを用いてソースレベルデバッグを行う際の注意事項を以下に示します。

（a）リンケージエディタは、Ver.6.0以上を使用してください。

（b）コンパイル時には"-debug"オプションをリンク時には"sdebug"オプションを指定してください。

（c）関数内で当該関数のローカルシンボルを参照できないことがあります。

（d）1行のソース行に複数のステートメントを記述した場合、1つのステートメントしか表示できません。

（e）最適化により消失したソース行のデバッグはできません。

（f）最適化により行の入れ替えなどが発生するため、プログラムの実行順序や逆アセンブル表示がソースリストの記述順序とは異なる場合があります。

例）

C 言語プログラム

```
    12 for (i=0; i<6; i++)
    13 {
    14      j = i+1;
    15     j++;
    16 }
17 j++;
```

シミュレータ・デバッガでの逆アセンブル表示

```
    14 j = i+1;
    12 for (i=0; i<6; i++)
    17      j++;
```

（g）for文、while文はループ文の入口と出口で2回逆アセンブル表示を行うことがあります。

(1) プロファイル機能

(a) Profile 機能の使用方法

(i) スタック情報ファイル

Profile 機能は、最適化リンカ(Ver.7.0 以降)が出力するスタック情報ファイル（拡張子".SNI"）を読み込むことができます。このファイルには、ソースファイル上の(静的な)関数呼び出し関係の情報が入っています。HEW がスタック情報ファイルを読み込むことで、ユーザアプリケーションが未実行（Profile データの測定を行う前）でも、関数の呼び出し関係を表示できるようになります。(ただし、"Show Only Executed Function"をチェックしている場合を除きます。)

HEW がスタック情報ファイルを読み込まない場合、Profile 機能で表示するデータは、Profile データ測定中に実行された関数についてのみになります。

スタック情報ファイルを読み込むかどうかは、Load Program ダイアログボックスの Load Stack Information file (SNI file)チェックボックスの ON/OFF で切り替えることができます。

リンカでスタック情報ファイルを生成するには、HEW のリンカオプション指定ダイアログの”Other”シートにある”Stack information output”をチェックしてください。その後、Build してください。

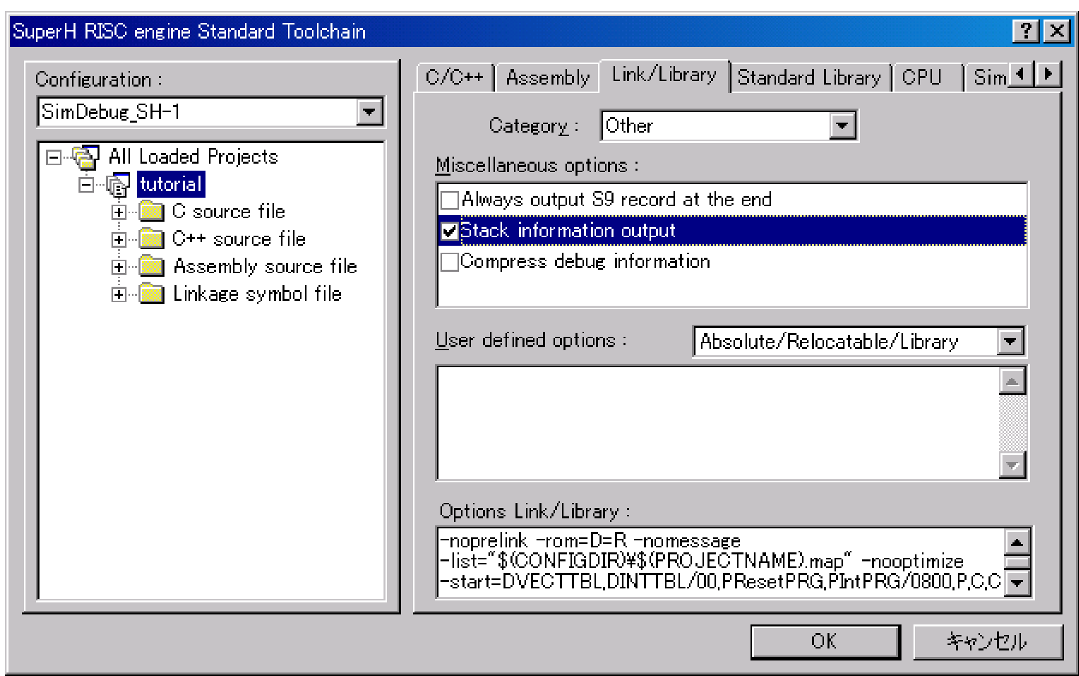


図 3.20　Category:[other]ダイアログボックス

(ii)　**プロファイル情報ファイル**

Profile 情報ファイルを作成するためには、アプリケーションプログラムのプロファイルデータを測定後に、Profile-List(または Tree, Chart)ウィンドウの Pop-up メニュー”Output Profile Information File…”を選択し、ファイル名を指定します。

Profile 情報ファイルには、関数の実行回数とグローバル変数のアクセス回数の情報が入っています。最適化リンカ(Ver.7.0 以降)は、Profile 情報ファイルを読み込み、関数および変数の配置を実際のプログラム動作状況にあわせた配置に最適化する機能を持っています。

RENESAS

プロファイル情報ファイルをリンカに入力するには、HEW のリンカオプション指定ダイアログの Optimize シートにある”Include Profile:”をボックスチェックして、プロファイル情報ファイル名を指定してください。

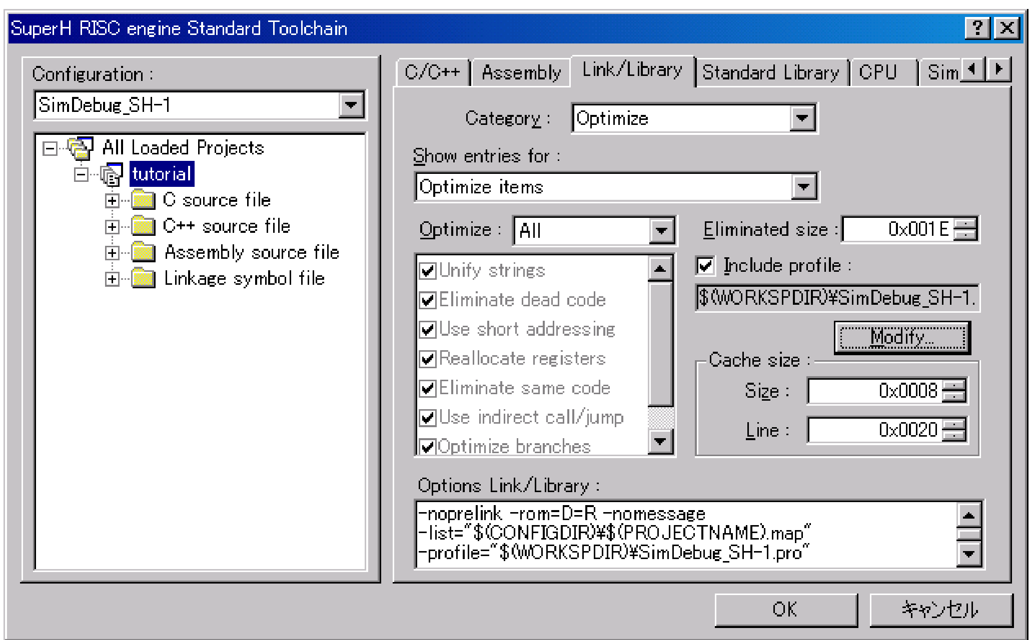


図 3.21　Category:[optimize]ダイアログボックス

(iii)　profile **ウィンドウ**

[表示->パフォーマンス->プロファイル]を選択し、Profile ウィンドウをオープンします。本メニューアイテムはロードモジュールをロード後に有効表示されます

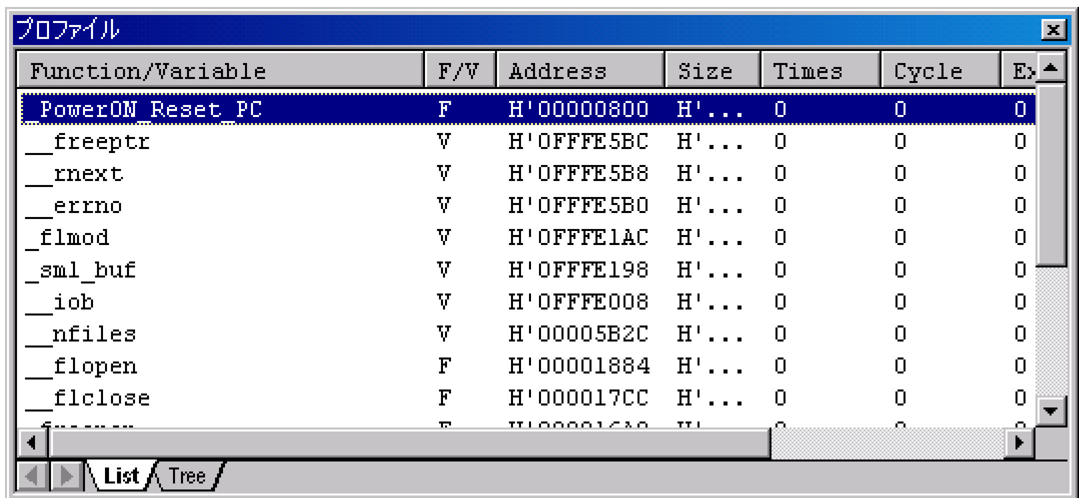


図 3.22　プロファイルウィンドウ

(iv) **プロファイルウィンドウメニュー**

プロファイルウィンドウの Pop-up メニュー"有効"を選択します。（メニューにチェックマークが付きます。）

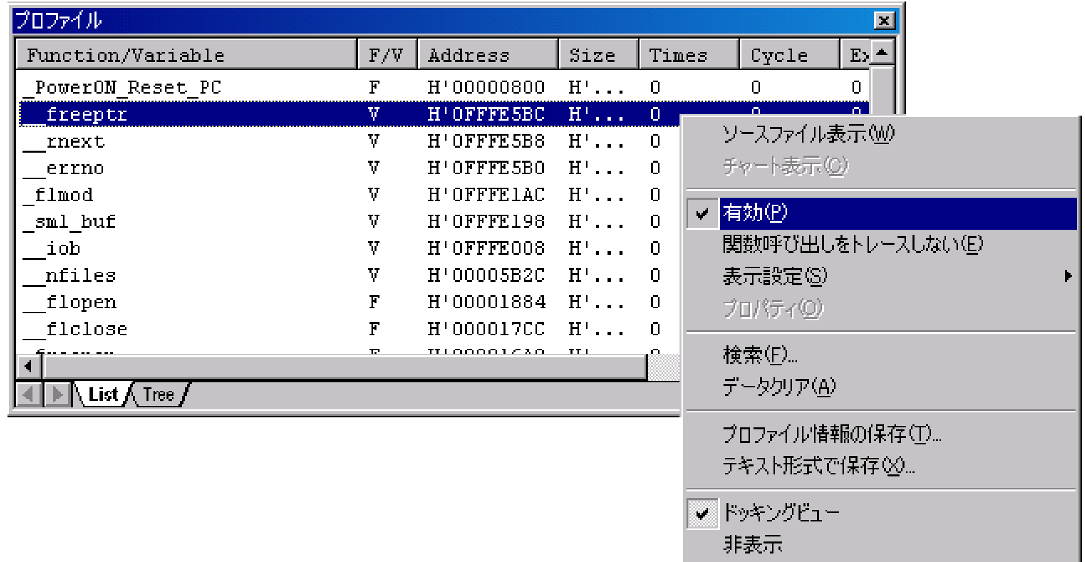


図 3.23　プロファイルウィンドウメニュー(有効)

(v) **プロファイル測定を停止したい条件で、**Breakpoint **を設定します。(特に停止条件を設定しなくても、手動で停止することができます。)**

(vi) (v)**で設定した停止条件が成立するか、マニュアルブレーク、その他の要因で実行が停止すると、プロファイルウィンドウに測定結果が表示される。**

(vii) **プロファイル情報ファイルを作成する場合は、**Pop-up **メニューの" プロファイル情報の保存**..."**を選択します。**

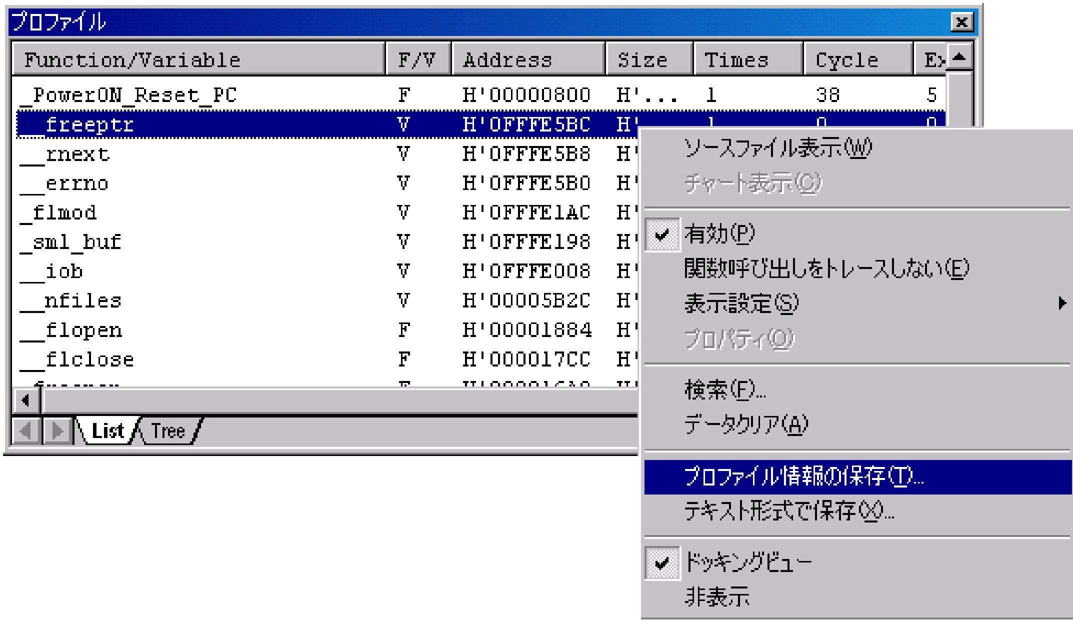


図 3.24　プロファイルウィンドウメニュー(プロファイル情報の保存...)

(b) **注意事項**

(i) アプリケーションプログラムの実行サイクル数をProfile機能で測定した場合のデータには 誤差があります。Profile機能では、アプリケーションプログラム全体の中で各関数が占める実行時間の比率を調べることができますが、より厳密に関数の実行サイクル数を測定したい場合には、Performance Analysis機能を使用してください。

(ii) デバッグ情報がないロードモジュールでプロファイル情報の測定を行った場合、関数名が表示されない場合があります。

(iii) スタック情報ファイル(拡張子".SNI")はロードモジュールファイル(拡張子".ABS")と同一のディレクトリに置いてある必要があります。

(iv) 測定結果の蓄積はできません。

(v) 測定結果の編集はできません。

(c) **機能概要**

Profile 機能は、アプリケーションプログラムの実行パフォーマンスを関数単位に測定します。アプリケーションプログラム中の性能劣化の原因となっている場所および要因を調査することができます。

(i) Profile **ウィンドウ**

Profile ウィンドウには、"List"タブと"Tree"タブがあります。

- List **タブ**

関数とグローバル変数をリスト表示し、各関数／変数の Profile データを表示します。

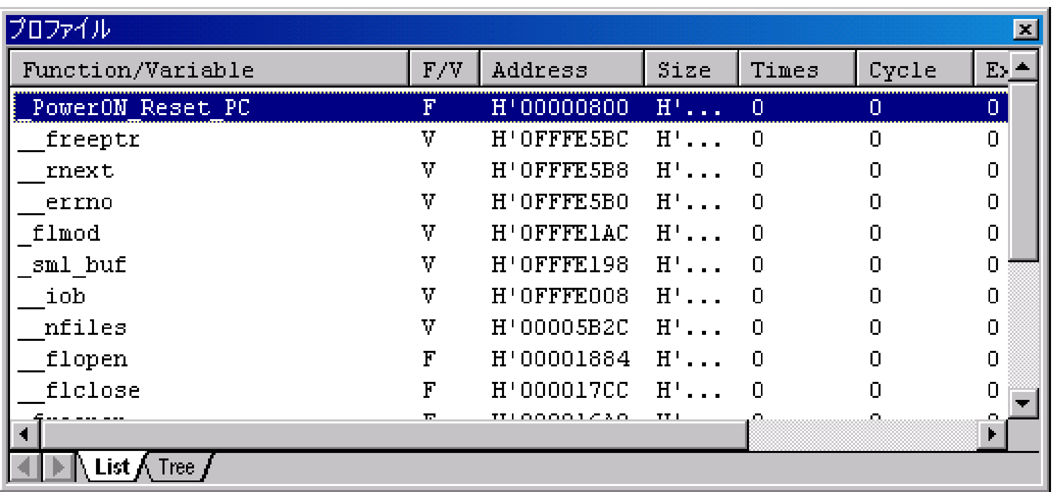


図 3.25 List **タブ**

RENESAS

• Tree タブ

関数の呼び出し関係を表示し、各呼び出し位置における Profile データを表示します。

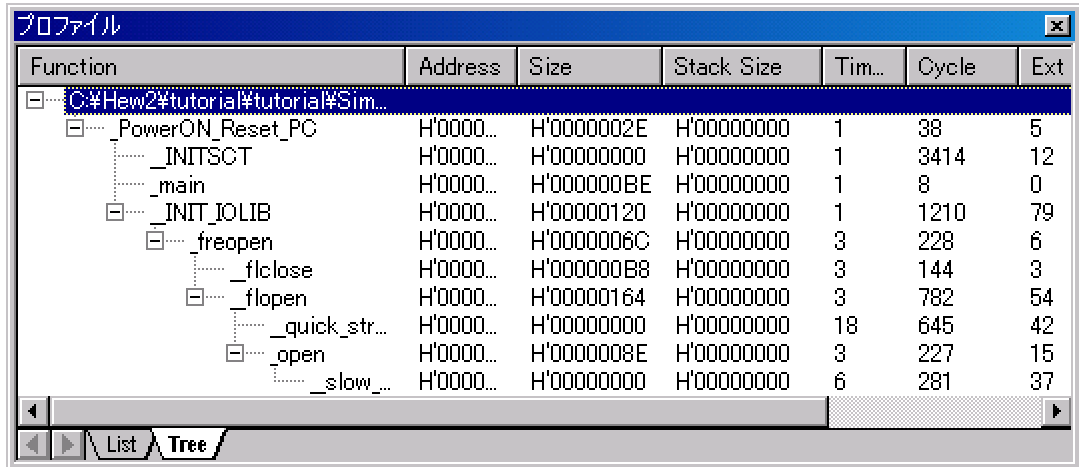


図 3.26 プロファイル-Tree ウィンドウ

• Profile-Chart ウィンドウ

Profile-Chart ウィンドウは、特定の関数に着目した関数の呼び出し関係を表示します。本ウィンドウは、着目する関数を中心に表示し、その左側には着目した関数を呼び出した関数、右側には、着目している関数が呼び出した関数を、それぞれ表示します。また、各呼び出しが行われた回数も表示します。

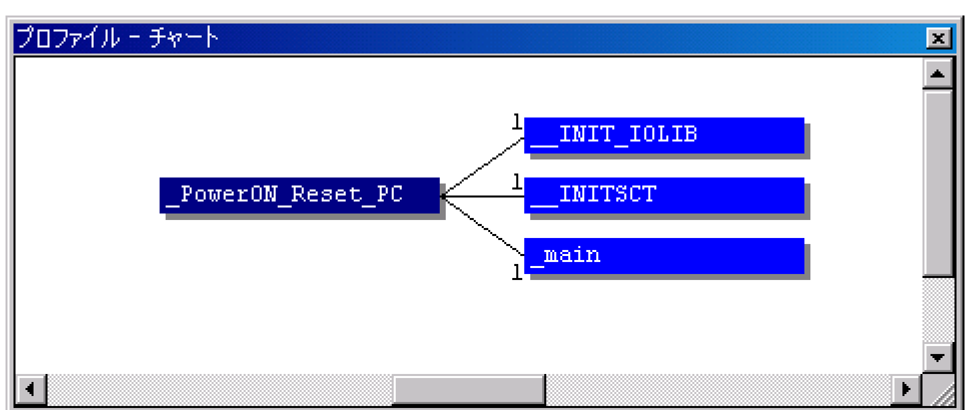


図 3.27 プロファイル-チャートウィンドウ

(ii) **表示データの種類および用途**

• Address

機能：関数(グローバル変数)のアドレスを表示します。
用途：関数が配置されているメモリ上の位置を知ることができます。アドレス順にソート表示することにより、メモリ上の配置イメージで関数とグローバル変数を並べることができます。
備考：ソート表示はListタブのみ可能です。

• Size

機能：関数(グローバル変数)のサイズを表示します。
用途：サイズ順にソート表示すれば、サイズが小さくて頻繁に呼び出されている関数を見つけることができます。そのような関数があればinline関数にすることで、関数呼び出しのオーバヘッドを減らせる場合があります。
また、キャッシュ内蔵マイコンをご使用の場合、サイズの大きい関数を実行すると、更新されるキャッシ

ュのサイズが大きくなります。このような、キャッシュミスの原因となり得る関数が頻繁に呼び出されていないかを容易に確認できます。

備考：ソート表示はListタブのみ可能です。

- Stack Size

機能：関数が使用するスタックのサイズを表示します。

用途：関数呼び出しのネストが深い場合、関数呼び出し経路をたどり、その経路上の全関数のスタックサイズを合計することで、おおよそのスタック使用量を見積もれます。

備考：Treeタブでのみ表示します。
関数が使用するスタックのサイズは、スタック情報ファイルに設定されています。スタック情報ファイルを読み込まない場合はすべて0を表示します。また、その場合に出力したプロファイル情報ファイル(拡張子".PRO")をスタック解析ツール(Hitachi Call Walker)に入力しても、正しい値は表示されません。

- Times

機能：関数が呼び出された回数(変数がアクセスされた回数)を表示します。

用途：呼び出し（アクセス）回数順にソート表示すれば、頻繁に呼び出されている関数や頻繁にアクセスされている変数を容易に調べることができます。

備考：ソート表示はListタブのみ可能です。

- **その他**

ターゲットプラットフォームにより、さまざまなデータを測定できます。お使いの Target(シミュレータ/エミュレータ)マニュアルを参照してください。

(iii)　表示設定

Pop-up メニューの"Setting…"を選択すると、Setting リストを表示します。このリストで表示をカスタマイズすることにより、問題個所の検出が容易になります。

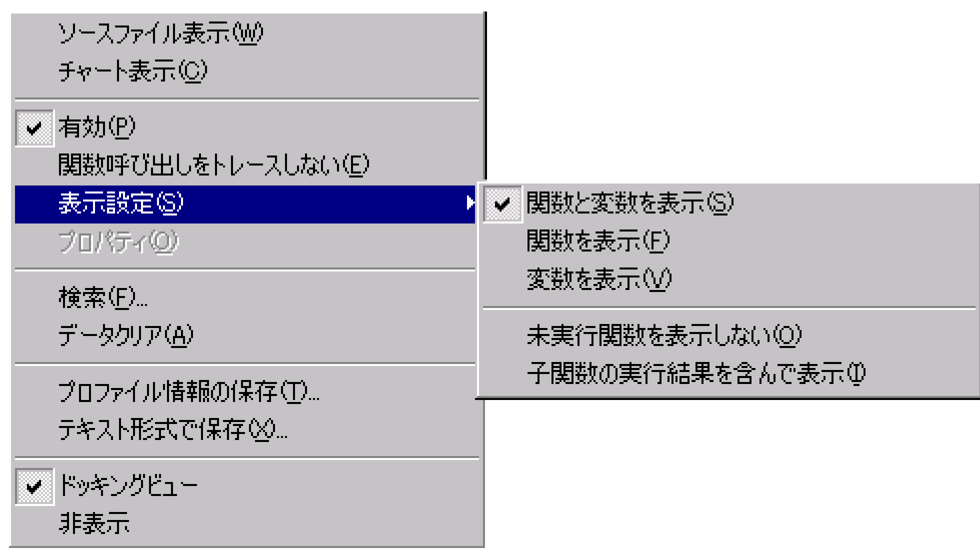


図 3.28　プロファイルウィンドウ Pop-up メニュー

• **関数と変数を表示**

機能：ウィンドウに関数と変数の情報を表示します。

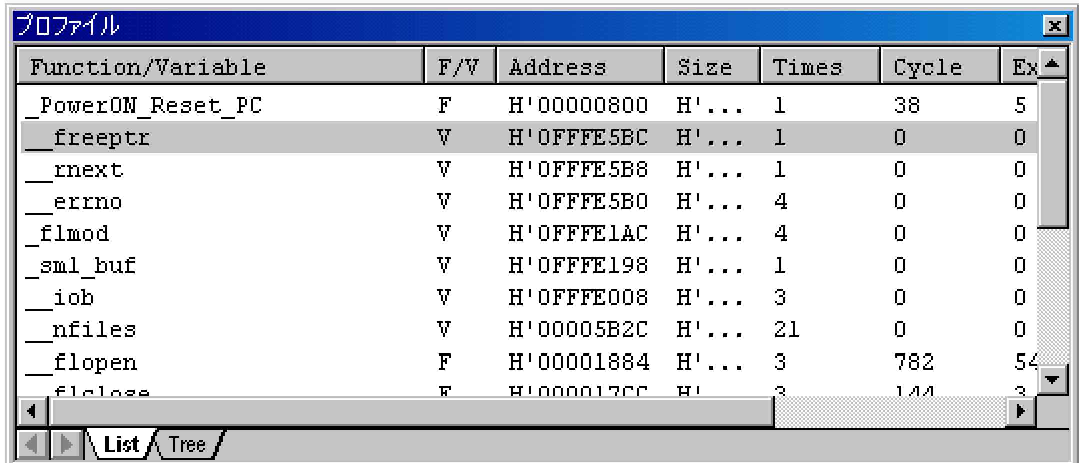


図 3.29　profile ウィンドウ(Show Function/Variables)

• **関数を表示**

機能：ウィンドウに関数の情報を表示します。

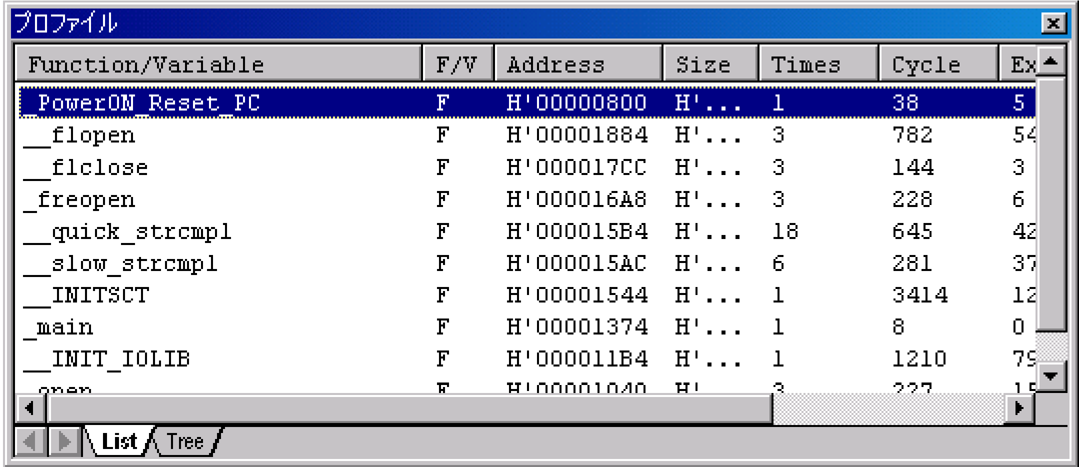


図 3.30　プロファイルウィンドウ(関数を表示)

- **変数を表示**

  機能：ウィンドウに変数の情報を表示します。

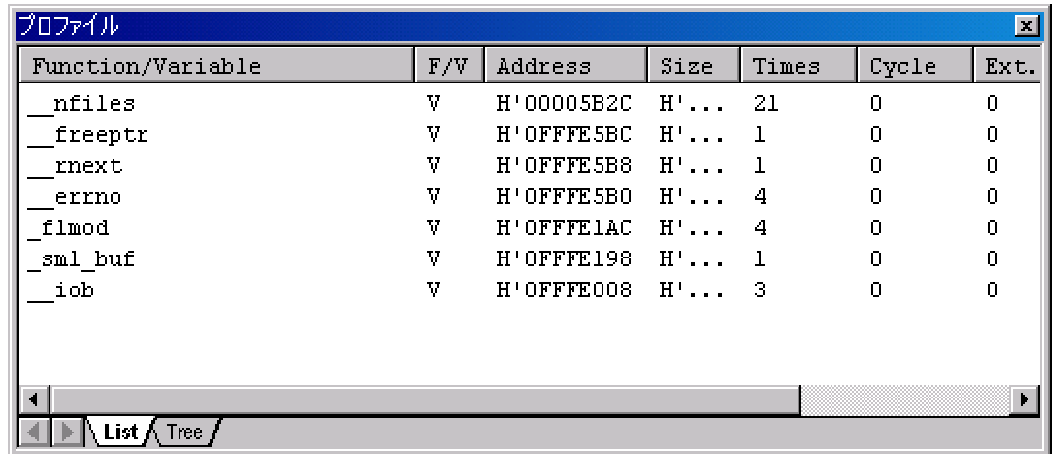


図 3.31　プロファイルウィンドウ(変数を表示)

- **未実行関数を表示しない チェックボックス**

機能：Profileデータ測定中に実行した関数(アクセスされた変数)のみを表示するか、または未実行の関数(未アクセスの変数)も表示するかを指定します。

用途：データ測定時に実行した関数のみを表示するので、測定対象外の関数の情報は表示されなくなり、表示が見易くなります。

また、全関数を表示すれば、全関数の中で、データ測定時に実行された関数が占める割合を容易に知ることができ、データ測定時のアプリケーションプログラム実行内容の満足度を容易に判断することができます。

備考：スタック情報ファイルを読み込んでいない場合は、本指定の内容にかかわらず、常に実行した関数(アクセスされた変数)だけを表示します。

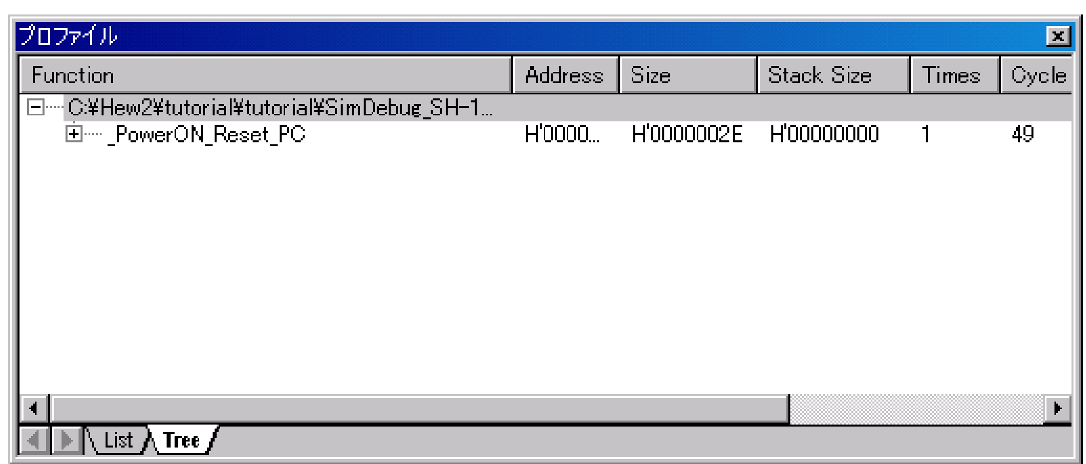
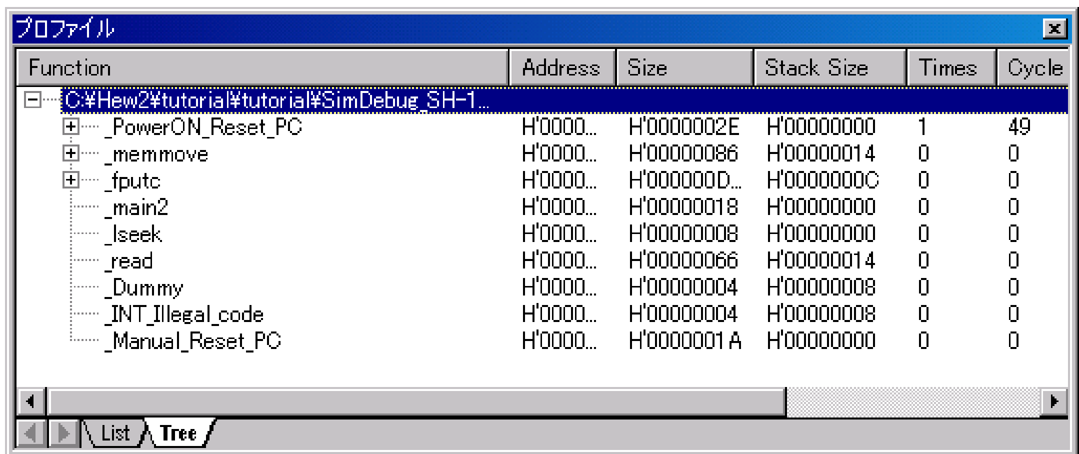


図 3.32　未実行関数を表示しないチェックボックスをチェックしたときの表示例

RENESAS

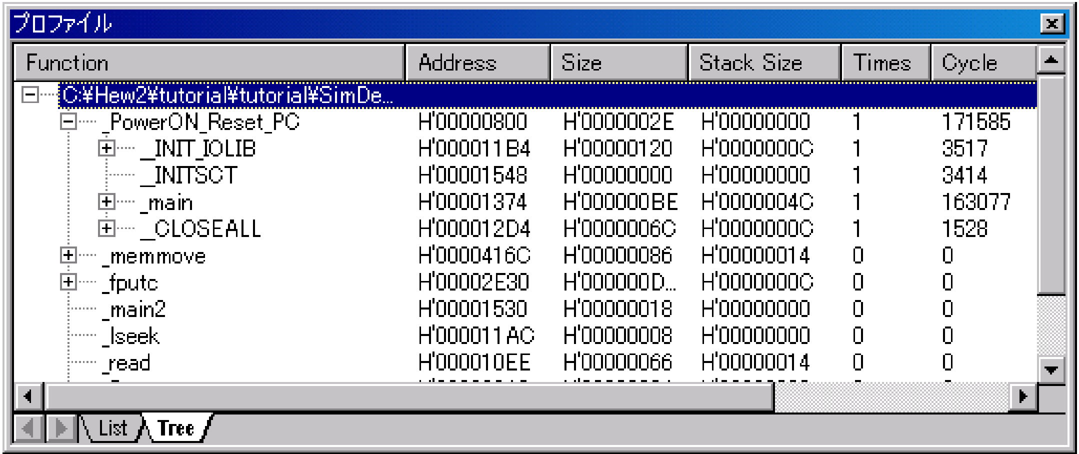


Function	Address	Size	Stack Size	Times	Cycle
C:¥Hew2¥tutorial¥tutorial¥SimDebug_SH-1...					
_PowerON_Reset_PC	H'0000...	H'0000002E	H'00000000	1	49
_memmove	H'0000...	H'00000086	H'00000014	0	0
_fputc	H'0000...	H'000000D...	H'0000000C	0	0
_main2	H'0000...	H'00000018	H'00000000	0	0
_lseek	H'0000...	H'00000008	H'00000000	0	0
_read	H'0000...	H'00000066	H'00000014	0	0
_Dummy	H'0000...	H'00000004	H'00000008	0	0
_INT_Illegal_code	H'0000...	H'00000004	H'00000008	0	0
_Manual_Reset_PC	H'0000...	H'0000001A	H'00000000	0	0

図 3.33　未実行関数を表示しないチェックボックスのチェックをはずしたときの表示例

- **子関数の実行結果を含んで表示 チェックボックス**

機能：測定結果のデータとして表示する値に、その子関数のデータを含むかどうかを指定します。
用途：たとえばSH1 SimulatorでCycleを測定している場合、本チェックボックスをチェックすると、”Cycle”の表示値はその関数が呼び出されてから、その関数からリターンするまでのサイクル数(その関数が呼び出している子関数のサイクル数も加算)になります。処理時間の割合をモジュール単位に調べる場合に有効です。
備考：Address, Size, Stack Size, Timesの各カラムに表示する値は変化しません。



Function	Address	Size	Stack Size	Times	Cycle
C:¥Hew2¥tutorial¥tutorial¥SimDe...					
_PowerON_Reset_PC	H'00000800	H'0000002E	H'00000000	1	171585
_INIT_IOLIB	H'000011B4	H'00000120	H'0000000C	1	3517
_INITSCT	H'00001548	H'00000000	H'00000000	1	3414
_main	H'00001374	H'000000BE	H'0000004C	1	163077
_CLOSEALL	H'000012D4	H'0000006C	H'0000000C	1	1528
_memmove	H'0000416C	H'00000086	H'00000014	0	0
_fputc	H'00002E30	H'000000D...	H'0000000C	0	0
_main2	H'00001530	H'00000018	H'00000000	0	0
_lseek	H'000011AC	H'00000008	H'00000000	0	0
_read	H'000010EE	H'00000066	H'00000014	0	0

図 3.34　子関数の実行結果を含んで表示 チェックボックスをチェックしたときの表示例

RENESAS

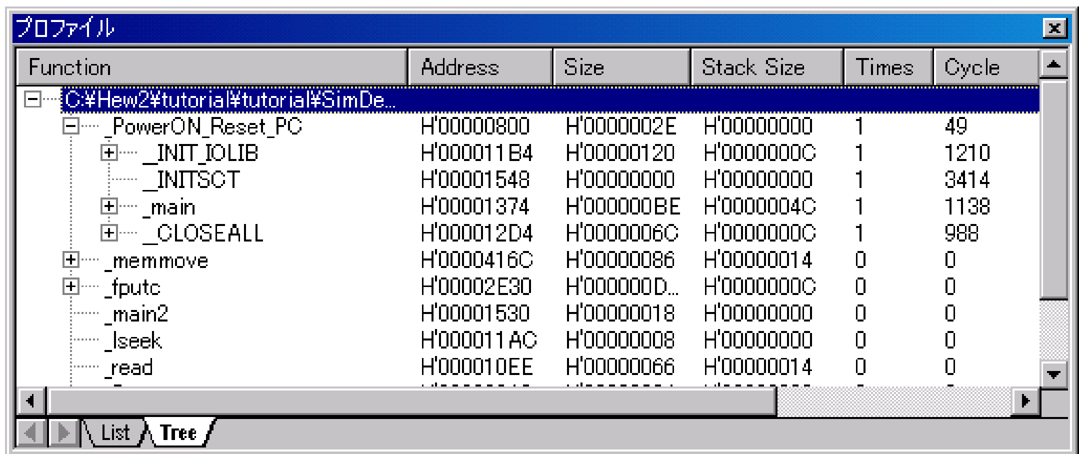


図 3.35　子関数の実行結果を含んで表示 チェックボックスのチェックをはずしたときの表示例

(iv)　Column の設定

Profile ウィンドウの表示カラム上で右クリックして Pop-up メニューを表示します。

機能：ウィンドウに表示する情報を選択できます。

用途：必要な情報だけを表示し、ウィンドウを見易くできます。

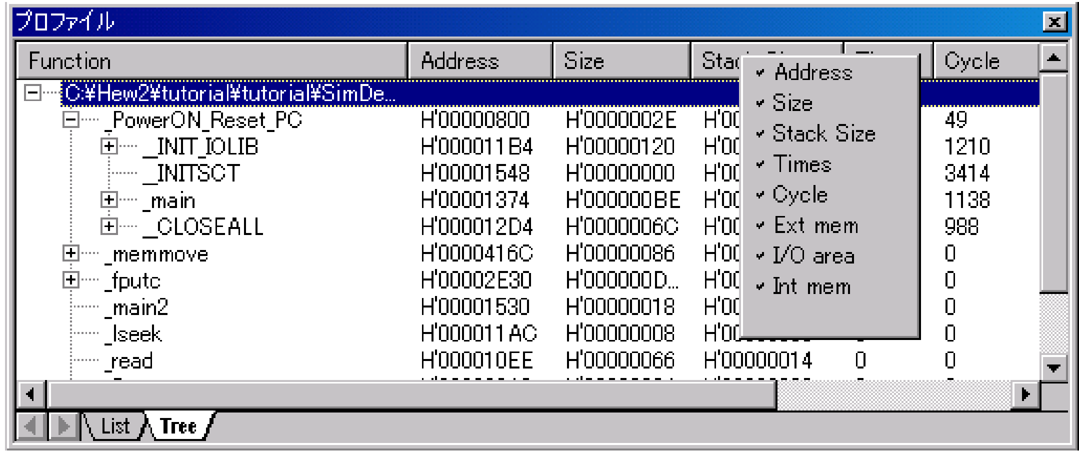


図 3.36　プロファイルウィンドウ Pop-Up ウィンドウ

RENESAS

3.16 構造体の境界調整数の変更

■説明

構造体の境界調整数を pack オプション(-pack={1 | 4 }、または#pragma pack 拡張子(pack 1 | pack 4 | unpack)を使用し、構造体の境界調整数を変更することができます。

オプションと#pragma 拡張子の両方が指定された場合には、拡張子の指定を優先します。

構造体、共用体、クラスの境界調整数は、メンバの最大の境界調整数と同じになります。

省略時解釈は、pack=4 です。

これらの指定による境界調整数は次のようになります。

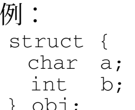
表 3.43 pack オプション指定時の構造体、共用体、クラスメンバの境界調整数

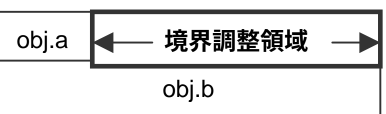
指定	pack=1	pack=4	指定なし
[unsigned]char	1	1	1
[unsigned]short、_ _fixed	1	2	2
[unsigned]int, [unsigned]long, [unsigned]long long, long, _ _fixed, _ _accum, long, _ _accum, 浮動少数点型, ポインタ型	1	4	4
境界調整数が 1 の構造体、共用体、クラス	1	1	1
境界調整数が 2 の構造体、共用体、クラス	1	2	2
境界調整数が 4 の構造体、共用体、クラス	1	4	4

■構造体データの割り付け方

(1) 構造体型の各メンバを割り付ける場合、そのメンバのデータ型の境界調整数にあわせるために直前のメンバとの間に空き領域が生じる場合があります。

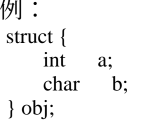
例：

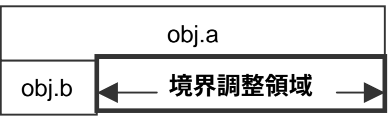

```
struct {
  char  a;
  int   b;
} obj;
```



(2) 構造体が4バイトの境界調整数を持ち、最後のメンバが1,2,3バイト目で終わっている場合、その次のバイトも含めて構造体型の領域として扱います。

例：

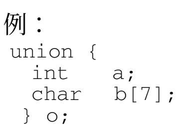

```
struct {
    int    a;
    char    b;
} obj;
```

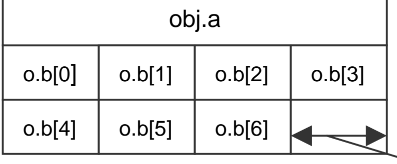


■共用体データの割り付け方

(1) 共用体が4バイトの境界調整数を持ち、メンバのサイズの最大値が4の倍数バイトでない場合、4の倍数になるまで残りのバイトも含めて共用体型の領域として扱います。

例：


```
union {
  int    a;
  char   b[7];
 } o;
```

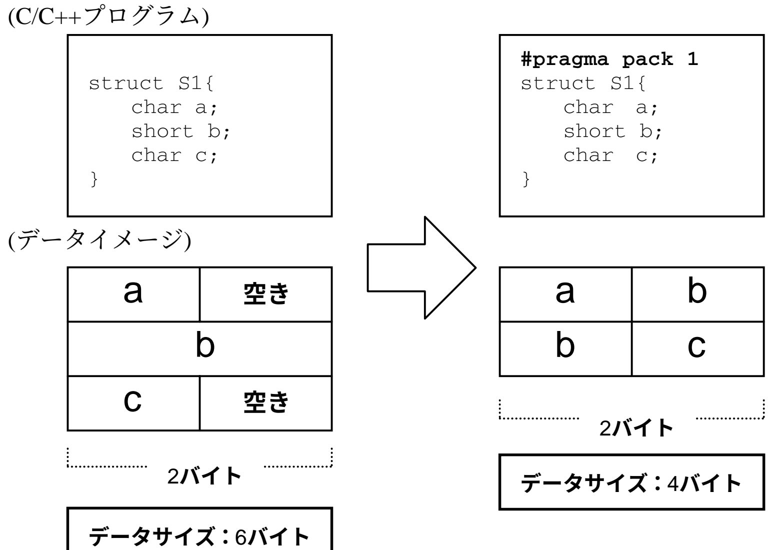


■境界調整数変更のイメージ

#pragma pack 1 を指定すると、以下のように 1 バイトデータ以外も奇数アドレスに割り付けることができるため、境界調整のための空白が入らない可能性があります。

よって、データサイズが小さくなる可能性があります。

(C/C++プログラム)

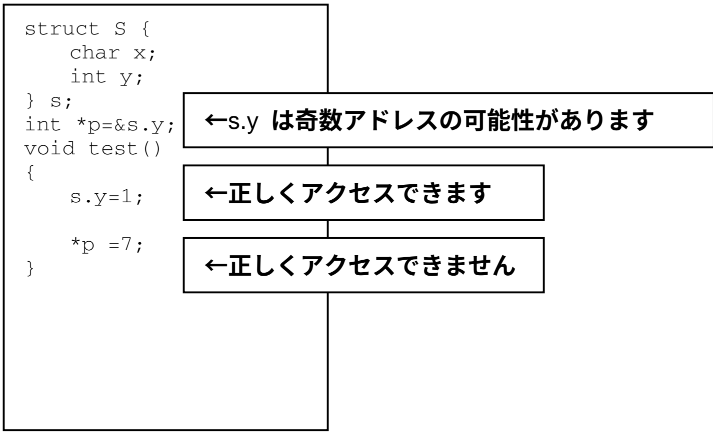


■注意事項

境界調整数を 1 とした場合、各メンバへのアクセスは byte アクセスの展開となります。また、メンバはポインタを用いてアクセスすることはできません。

データサイズが小さくなるので、データをブロック転送する場合などで有効ですが、境界調整数を１に変更するとワード、ロングワードの構造体メンバを１バイトずつアクセスするためコードが増大します。

(C/C++プログラム)



3.17 long long 型

■説明

long long, unsigned long long 型のデータ型をサポートします。

有符号整数については long long、無符号整数については unsigned long long と書きます。

型が long long の整数定数を作成する場合、整数値の後ろに接尾語 LL をつけ、型が unsigned long long の整数定数の場合は、整数値の後ろに接尾語 ULL を付けます。

表 3.44 整数型とその値の範囲

型	値の範囲	データサイズ
char	-128 ～ 127	1 byte
signed char	-128 ～ 127	1 byte
unsigned char	0 ～ 255	1 byte
short	-32768 ～ 32767	2 byte
unsigned short	0 ～ 65535	2 byte
int	-2147483648 ～ 2147483647	4 byte
unsigned int	0 ～ 4294967295	4 byte
long	-2147483648 ～ 2147483647	4 byte
unsigned long	0 ～ 4294967295	4 byte
long long	-9223372036854775808 ～ 9223372036754775807	8 byte
unsigned long long	0 ～ 18446744073709551615	8 byte

3.18 DSP-C 仕様

■説明

DSP-C 言語をサポートします。

SuperH RISC engine C/C++コンパイラのコンパイルオプション「dspc」を指定した場合に有効となります。

3.18.1 固定小数点データ型

従来使われてきた整数型で代用する方法ではなく、固定小数点データ型を用いることで、小数値をそのまま記述することができます。

また、SuperH RISC engine C/C++コンパイラでは固定小数点データ型の使用状況により、適切な DSP 演算命令を生成します。固定小数点データ型の内部表現を**表** 3.45 に示します。

表 3.45 **固定小数点データ型の内部表現**

型	Size (メモリ上の Size)	境界調整数 (byte)	データの範囲		定数添字
			最小値	最大値	
_ _fixed	16 bit (16 bit)	2	-1.0	$1.0-2^{-15}$ (0.999969482421875)	r
long _ _fixed	32 bit (32 bit)	4	-1.0	$1.0-2^{-31}$ (0.9999999995343387126922607421875)	R
_ _accum	24 bit (32 bit)	4	-256.0	$256.0-2^{-15}$ (255.999969482421875)	a
long _ _accum	40 bit (64 bit)	4	-256.0	$256.0-2^{-31}$ (255.99999999953433871269226074218 75)	A

■注意事項

(1) _ _accumおよびlong _ _accum型の場合、メモリ格納は右詰めとなり、先頭部分は符号拡張されます。

例) (_ _accum)128.5a ⇒ 00 40 40 00

例) (long _ _accum)(-256.0A) ⇒ FF FF FF 80 00 00 00 00

(2) 従来方法とDSP-Cの比較

C言語関数[従来方法]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8
short input[NUM] = {0x1000, 0x2000,
                    0x4000, 0x6000,
                    0xf000, 0xe000,
                    0xc000, 0xa000};
short result[NUM];
void func(void)
{
    int i;
    for (i = 0; i < NUM; i++) {
    result[i] = input[i] + 0x1000;
    }
}
void main(void)
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
    printf("%f¥n",
result[i]/32768.0);
    }
}
```

[DSP-C の場合]

```
// -cpu=sh3dsp -dspc
#include <stdio.h>
#define NUM 8
__fixed input[8] = { 0.125r, 0.25r, 0.5r,
                     0.75r,-0.125r,
                     -0.25r, -0.5r,
                     -0.75r};
__fixed result[NUM];
void func()
{
    int i;
    for (i = 0; i < NUM; i++) {
    result[i] = input[i] + 0.125r;
    }
}
void main(void)
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
    printf("%r¥n", result[i]);
    }
}
```

(3) 積和演算の例

整数型で代用した場合、積の桁をあわせる必要がありますが、固定小数点データ型は必要ありません。

C言語関数[従来方法]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8
short x_input[NUM] = {0x1000, 0x2000,
0x4000, 0x6000, 0xf000, 0xe000,
0xc000, 0xa000};
short y_input[NUM] = {0x1000, 0x2000,
0x4000, 0x6000, 0xf000, 0xe000,
0xc000, 0xa000};
int result;
int func(short *x_input, short
*y_input)
{
    int i;
    int temp = 0;
    for (i = 0; i< NUM ;i++) {
        temp += (x_input[i] *
y_input[i]) >> 15;
    }
    return (temp);
}
void main()
{
    result = func(x_input, y_input);
    printf("%f¥n", result/32768.0);
}
```

[DSP-C の場合]

```
// -cpu=sh3dsp -dspc -fixed_noround
#include <stdio.h>
#define NUM 8
__X __fixed x_input[NUM] = { 0.125r,
  0.25r,  0.5r,  0.75r, -0.125r,
  -0.25r, -0.5r, -0.75r};
__Y __fixed y_input[NUM] = { 0.125r,
0.25r,  0.5r,  0.75r,  -0.125r,
-0.25r, -0.5r, -0.75r};
__accum result;
void func(__accum *result_p,
         __X __fixed *x_input,
         __Y __fixed *y_input)
{
    int i;
    __accum temp = 0.0a;
    for (i = 0; i< NUM ;i++) {
        temp += x_input[i] * y_input[i];
    }
    *result_p = temp;
}
void main()
{
    func(&result, x_input, y_input);
    printf("%a¥n", result);
}
```

3.18.2 メモリ修飾子

変数にX／Yメモリ修飾子を付加することで､通常のメモリアクセス命令よりも効率の良いX／Yメモリ専用アクセス命令の生成を促します。

X／Yメモリへの格納を明示的に指定するために、以下の修飾子を使用します。

__X　：Xメモリにデータを格納
__Y　：Yメモリにデータを格納

SuperH RISC engine C/C++コンパイラは、__X/__Yメモリ修飾子の付いたオブジェクトを**表** 3.46に示すセクションに出力します。これらのセクションは、リンク時にそれぞれX／Yメモリに割り付けるようにしてください。

表 3.46　メモリ修飾子の仕様

名称	セクション	内容
定数領域	$XC	const 型のデータ　(X メモリに格納)
	$YC	const 型のデータ　(X メモリに格納)
初期化データ領域	$XD	初期値のあるデータ　(X メモリに格納)
	$YD	初期値のあるデータ　(Y メモリに格納)
未初期化データ領域	$XB	初期値のないデータ　(X メモリに格納)
	$YB	初期値のないデータ　(Y メモリに格納)

ただし､X／YメモリがRAM上にのみ存在する場合があります｡この場合のROM化の際は注意する必要があります。

■使用例

(1) __X/__Yメモリ修飾子使用時のメモリ格納例

```
_ _X  int        a;    // Xメモリに格納
int  _ _X        b;    // Xメモリに格納
_ _Y  int      * c;    // Yメモリ上のint型へのポインタ(メモリは不定)
int   _ _Y     * d;    // Yメモリ上のint型へのポインタ(メモリは不定)
int   *_ _Y      e;    // int型へのポインタ(Yメモリに格納)
_ _X int *_ _Y    f;   // Xメモリ上のint型へのポインタ(Yメモリに格納)
```

(2) ROM上の定数領域、初期化データ領域をX／Y RAMへコピー

リンク時にはROM上に置き、プログラムの実行開始時にX／Y RAM上にコピーします。

したがって、最適化リンケージエディタの rom オプションを用いて、ROM上とX／Y RAMに、二重に領域をとる必要があります。

リンク時サブコマンド例)

```
rom=$XC=XC,$XD=XD,$YC=YC,$YD=YD
start P,C,D,$XC,$XD,$YC,$YD/400,$XB,XC,XD/05007000,$YB,YC,YD/05017000
```

また ROM から X／Y RAM へのコピーは、標準ライブラリ関数_INITSCT()を使うと容易にできます。

（使用例）_INITSCT()

```
#include <_h_c_lib.h>
void PowerON_Reset(void)
{
    _INITSCT();
    main();
    sleep();
}

#pragma section $DSEC
static const struct {
    void *rom_s;
    void *rom_e;
    void *ram_s;
} DTBL[] = { {__sectop("$XC"), __secend("$XC"), __sectop("XC")},
         {__sectop("$XD"), __secend("$XD"), __sectop("XD")},
         {__sectop("$YC"), __secend("$YC"), __sectop("YC")},
         {__sectop("$YD"), __secend("$YD"), __sectop("YD")}};
#pragma section
```

（3）定数領域、初期化データ領域を使用しない

_ _X／_ _Y メモリ修飾子を付けたオブジェクトに対して const 指定、初期化データを付けないようにすれば、ROM 上と X／Y RAM 上に二重に領域を取る必要はありません。

たとえば、以下の例のように動的に初期化するようにすることで、初期化データをなくすことができます。

（使用例）

```
#define NUM 8
__X __fixed x_input[NUM];
__Y __fixed y_input[NUM];
__fixed x_input[NUM] = {  0.125r,   0.25r,  0.5r,  0.75r,
                           -0.125r,  -0.25r, -0.5r, -0.75r};
__fixed y_input[NUM] = {  0.125r,   0.25r,  0.5r,  0.75r,
                           -0.125r,  -0.25r, -0.5r, -0.75r};
void xy_init()
{
      int i;

      for (i = 0; i< NUM; i++) {

          x_input[i] = x_init[i];
          y_input[i] = y_init[i];
      }
}

void main()
{
       xy_init();
          :
          :
}
```


RENESAS

(4) 従来方法と DSP-C の比較

C言語関数[従来方法]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8
short x_input[NUM] = {0x1000, 0x2000,
                 0x4000, 0x6000,0xf000,
                 0xe000, 0xc000, 0xa000};
short y_input[NUM] = {0x2000, 0x4000,
                 0xe000, 0xf000,
    0x6000, 0x2000, 0xe000, 0xf000};
short result[NUM];
void func(void)
{
    int i;
    for (i = 0; i < NUM; i++) {
        result[i]=x_input[i]-y_input[i];
    }
}

void main(void)
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
        printf("%f¥n", result[i]/32768.0);
    }
}
```

[DSP-C の場合]

```
// -cpu=sh3dsp -dspc
#include <stdio.h>
#define NUM 8
__X __fixed x_input[NUM] = { 0.125r,
              0.25r,  0.5r,  0.75r,
              -0.125r, -0.25r, -0.5r,
              -0.75r};
__Y __fixed y_input[NUM] = {0.25r,
            0.5r,  -0.25r, -0.125r,
      0.75r, 0.25r, -0.25r, -0.125r};
__fixed result[NUM];
void func(void)
{
    int i;
    for (i = 0; i < NUM; i++) {
        result[i] = x_input[i] -
y_input[i];
    }
}

void main(void)
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
        printf("%r¥n", result[i]);
    }
}
```

3.18.3 飽和修飾子

演算結果がオーバフローした場合にその結果を最大値／最小値に置き換える飽和演算を DSP-C では飽和修飾子をつけるだけで可能にします。

飽和演算を指定するために以下の修飾子を使用します。

```
_ _sat
```

飽和修飾子は、_ _fixed 型、long _ _fixed 型データのみに使用することができます。それ以外の型に指定した場合は、エラーになります。

なお、式の中に１つ以上の飽和修飾子(_ _sat 指定)指定されたデータがあれば、飽和演算を行います。

■使用例

(1) _ _sat指定例

```
_ _fixed       a;
_ _sat  _ _fixed b;
_ _fixed       c;

a = -0.75r ;
b = -0.75r ;
c = a + b  ; // c = -1.0r になります。
```

(2) 従来方法とDSP-Cの比較

C言語関数[従来方法]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8
short x_input[NUM] = {0x1000, 0x2000, 0x4000,
   0x6000, 0xf000, 0xe000, 0xc000, 0xa000};
short y_input[NUM] = {0x1000, 0x2000, 0x4000,
    0x6000, 0xf000, 0xe000, 0xc000, 0xa000};
short result[NUM];
void func(void)
{
    int i;
    int temp;
    for (i = 0; i < NUM; i++) {
        temp = x_input[i] + y_input[i];
        if (temp > 32767) {
            temp = 32767;
        }
        else if (temp < -32768) {
        temp = -32768;
    }
    result[i] = temp;
    }
}
void main(void)
{
    int i;
    func();
     :
```

[DSP-C の場合]

```
// -cpu=sh3dsp -dspc
#include <stdio.h>
#define NUM 8
__sat __X __fixed x_input[NUM] = { 0.125r,
0.25r,  0.5r,  0.75r,
  -0.125r, -0.25r, -0.5r, -0.75r};
__sat __Y __fixed y_input[NUM] = { 0.125r,
0.25r,  0.5r,  0.75r,
-0.125r, -0.25r, -0.5r, -0.75r};
__fixed result[NUM];

void func(void)
{
    int i;
    for (i = 0; i < NUM; i++) {
        result[i] = x_input[i] +
y_input[i];
    }
}
void main(void)
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
        printf("%r¥n", result[i]);
    }
}
```

3.18.4 循環修飾子

モジュロアドレッシングを指定するために、以下の修飾子を使用します。

_ _circ

モジュロアドレッシングは、メモリ修飾子(_ _X／_ _Y)指定のある_ _fixed 型の１次元配列およびポインタのみに指定することができます。それ以外の条件で使用した場合はエラーになります。

■使用例

（1） 従来方法とDSP-Cの比較

C言語関数[従来方法]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8
#define BUFFER_SIZE 4
short x_input[NUM] = {0x1000, 0x2000, 0x4000,
0x6000,, 0xf000, 0xe000, 0xc000, 0xa000};
short y_input[BUFFER_SIZE] = {0x2000, 0x4000,
0x2000, 0x1000};
short result[NUM];

void func()
{
    int i;
    for (i = 0; i < NUM; i++) {
      result[i] = x_input[i] +
      y_input[i%(BUFFER_SIZE)];
    }
}

void main()
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
      printf("%f¥n", result[i]/32768.0);
    }
}
```

[DSP-C の場合]

```
// -cpu=sh3dsp -dspc
#include <stdio.h>
#include <machine.h>
#define NUM 8
#define BUFFER_SIZE 4
__X __fixed x_input[NUM] = { 0.125r,  0.25r,
    0.5r,  0.75r, -0.125r, -0.25r, -0.5r,
    -0.75r};
__circ __Y __fixed y_input[BUFFER_SIZE] =
    {0.25r, 0.5r, 0.25r, 0.125r};
__fixed result[NUM];

void func()
{
    int i;
    set_circ_y(y_input, sizeof(y_input));
    for (i = 0; i < NUM; i++) {
      result[i] = x_input[i] + y_input[i];
    }
    clr_circ();
}

void main()
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
      printf("%r¥n", result[i]);
    }
}
```

■注意事項

（1） モジュロアドレッシングは、組み込み関数 set_circ_x()またはset_circ_y()とclr_circ()の間にある１次元配列およびポインタが対象となります。

（2） 複数の配列を同時にモジュロアドレッシング指定をした場合、および上記組み込み関数の間以外で_ _circ指定のある配列またはポインタを参照した場合、動作は保証しません。

（3） 負方向へのモジュロアドレッシングを指定した場合、動作は保証しません。

（4） モジュロアドレッシングを行うデータは、リンク時にアドレスの上位16bitが同じになるように配置する必要があります。なお、配列内容の直接参照はできません。

（5） 以下のいずれかに該当する場合、動作は保証しません。(ウォーニングを出力する場合もあり)

optimize=0を指定している

_ _circポインタをローカル変数以外に指定している

_ _circポインタにvolatileを指定している

_ _circポインタの更新のみで参照がない

組み込み関数 set_circ_xまたはset_circ_yとclr_circの間に関数呼び出しがある

3.18.5 型変換

型変換の規則を**表** 3.47 に示します。

表 3.47 **型変換の規則**

変換	仕様
_ _fixed -> long _ _fixed	下位 16bit ゼロクリア。 値は変わらない。
_ _accum -> long _ _accum	
long _ _fixed -> _ _fixed	下位 16bit 切り捨て。 小数の精度が落ちる。
long _ _accum -> _ _accum	
_ _fixed -> _ _accum	上位 8bit 符号拡張。 値は変わらない。
long _ _fixed -> long _ _accum	
_ _fixed -> long _ _accum	上位 8bit 符号拡張。 下位 16bit ゼロクリア。 値は変わらない。
long _ _fixed -> _ _accum	上位 8bit 符号拡張。 下位 16bit は切り捨て。 小数の精度が落ちる。
_ _accum -> _ _fixed	上位 8bit 切り捨て。 9bit 目を符号 bit とする。 整数部分が 0 の時は値は不変。
long _ _accum -> long _ _fixed	
_ _accum -> long _ _fixed	
long _ _accum -> _ _fixed	上位 8bit 切り捨て。 下位 16bit 切り捨て。 9bit 目を符号 bit とする。整数部分が 0 のときは値は不変。 小数の精度が落ちる。
_ _fixed ->符号付き整数型	-1.0r, -1.0R の場合は、-1, それ以外の場合は 0。
long _ _fixed ->符号付き整数型	
_ _accum ->符号付き整数型	小数部切り捨て。 変換後の値は,-256～255 の間の整数。
long _ _accum ->符号付き整数型	
_ _fixed ->符号なし整数型	-1.0r, -1.0R の場合は変換後の型の最大値、 それ以外の場合は 0。
long _ _fixed ->符号なし整数型	
_ _accum ->符号なし整数型	小数部切り捨て。 正の値の場合、変換後の値は、0～255 の間の整数。 負の値の場合、(変換前の値+1+変換後の型の最大値)。
long _ _accum ->符号なし整数型	
符号付き整数型-> _ _fixed	変換前の最上位 bit を変換後の最上位 bit にする。 その他の bit はすべて 0。
符号付き整数型->long _ _fixed	
符号付き整数型-> _ _accum	値の下位 9bit を整数部とする。 小数部は 0 とする。
符号付き整数型->long _ _accum	
符号なし整数型-> _ _fixed	変換後のすべての bit を 0 とする。
符号なし整数型->long _ _fixed	
符号なし整数型-> _ _accum	値の下位 9bit を整数部とする。 小数部は 0 とする。
符号なし整数型->long _ _accum	
固定小数点 ->浮動小数点	変換後の型で表現可能な値は、元の値と同じ。 表現できない場合は、最も近い値に丸める。
浮動小数点 ->固定小数点	小数部分は固定小数点 -> 浮動小数点変換と同じ。 整数部分は浮動小数点 -> 整数変換と同じ。 整数部分が固定小数点の表現可能範囲内の場合は、 そのままの値を保つようにする。 範囲外の場合はあふれた部分の最下位 bit を符号 bit とする。また、変換後の型に飽和処理指定があっても飽和処理は行わない。

■注意事項

（1） (long)_ _fixed ⇔ 整数型

(long)_ _fixed型で表現できる整数は、０と－１のみとなります。
このため、この変換を行うことで情報落ちが発生してしまいます。

（2） (long)_ _accum ⇔ 整数型

(long)_ _accum型は、－256～255の間の整数型を表現することができ、この範囲の整数型であれば、変換後も情報を保持します。
ただし、符号なし整数型への変換の場合、負の値はオーバフローとなりますので注意が必要です。
整数型のみを必要とする演算が続く場合は、整数型に変換した方が、性能向上になる場合もあります。

(3) ビットパターンコピー

ビットパターンをコピーする場合、代入演算子などで行いますと型変換されてしまい、期待どおりの結果を得られません。この場合は、組み込み関数(long_as_lfixed, lfixed_as_long)を使用してください。

3.18.6 算術変換

演算の 2 つのオペランドの型が異なる場合、**図** 3.37 の上側にある型にあわせて演算します。

また、**図** 3.37 で関係が上下間にない型(整数型⇔固定小数点, _ _accum⇔long_ _fixed)間の演算を記述した場合は、エラーとなります。このような場合、明示的にキャストで型変換する必要があります。ただし、結果の値が保証される範囲にて、効率などの面から上記変換ルールに従わないで演算する場合もあります。

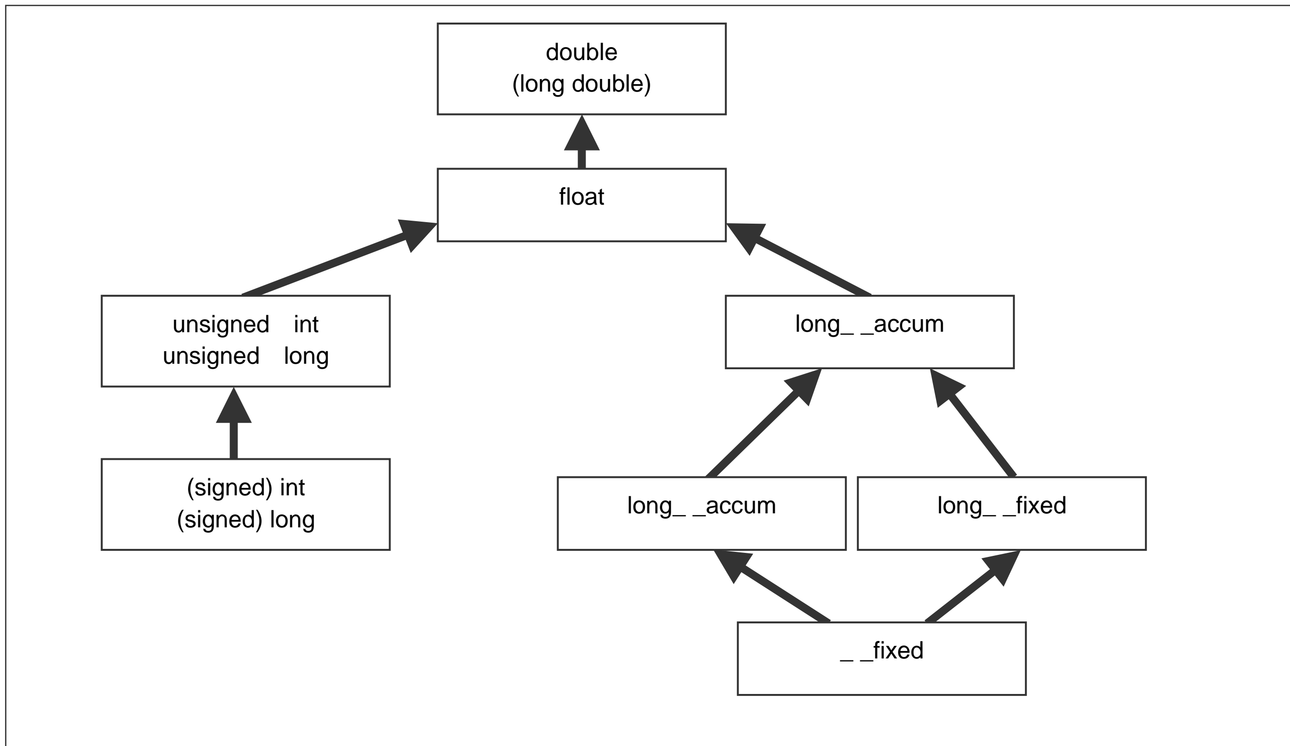


図 3.37 **算術変換の規則**

3.19 MAP 最適化拡張オプション

■説明

リンクにより確定するシンボル割り付けアドレスの情報を使用せずに MAP 最適化を行います。再コンパイルの必要はありません。

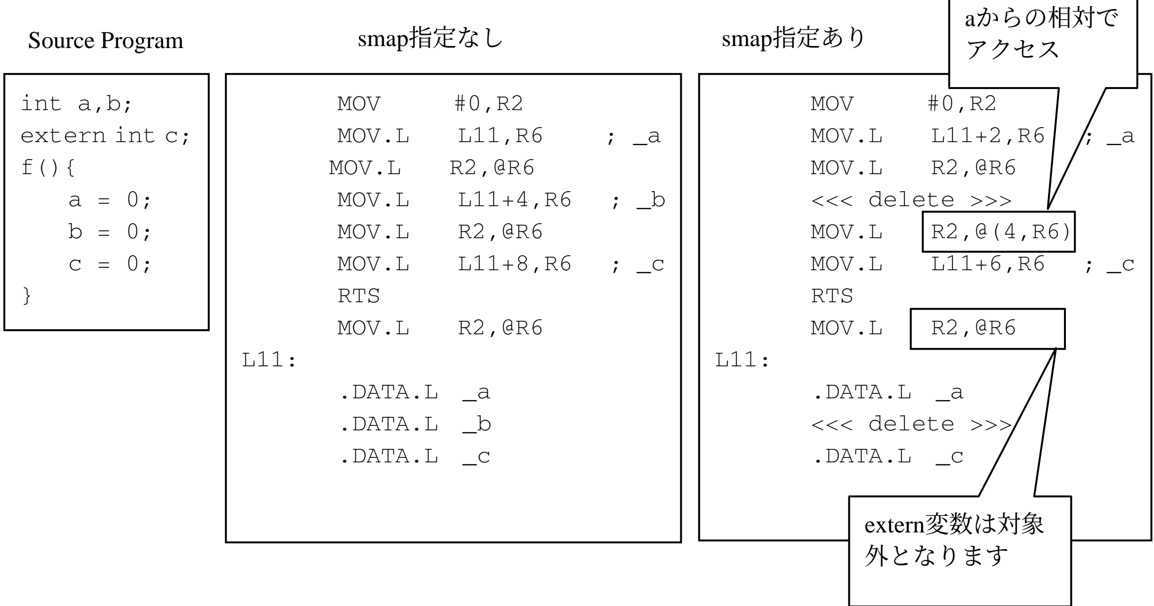
ただし、ファイル内で定義された静的変数のみが最適化の対象のため、extern 変数は、最適化の対象外となります。

3.19.1 使用方法

コンパイル時に"-smap"オプションを指定します。

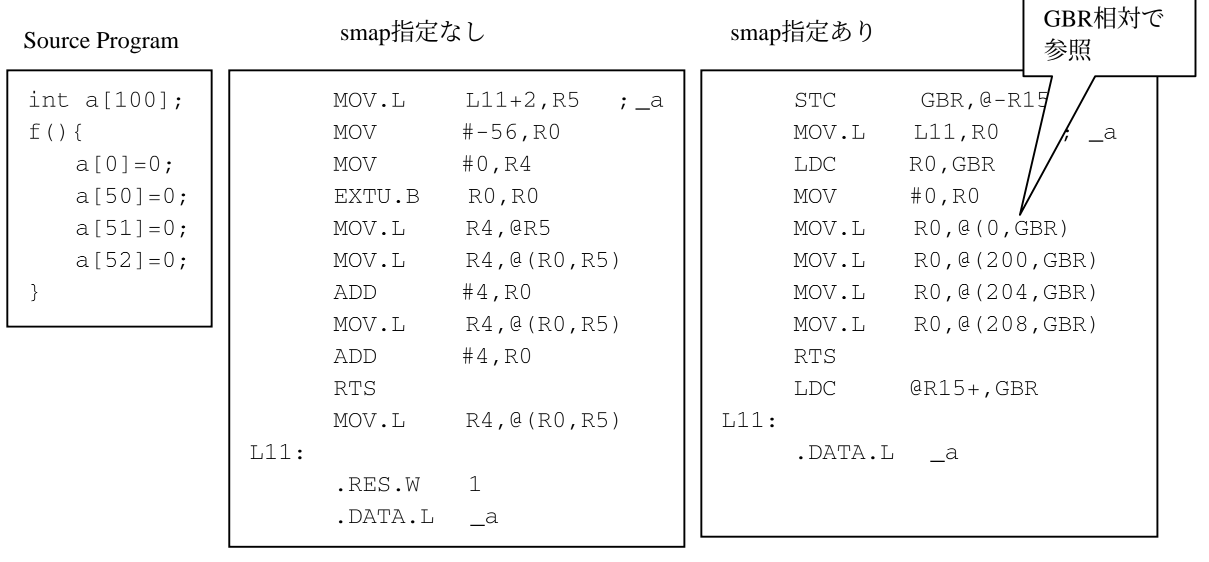
3.19.2 外部変数アクセスコード改善例(1)

同一セクションの変数割り付け順序を意識して、連続して割り付けられる変数を同一レジスタの相対でアクセスします。



3.19.3 外部変数アクセスコード改善例(2)

gbr=auto オプション（デフォルト）指定時、外部変数アクセスのベースとして、GBR を使用します。



3.20 TBR 相対関数呼び出し

■説明

SH-2A、SH2A-FPU で、ジャンプテーブルベースレジスタ（TBR）を使用した、テーブル参照サブルーチンコール命令による関数呼び出しを実現します。

TBR との相対値は､TBR をベースアドレスとするジャンプテーブル中のオフセットで､$TBR セクション先頭から、呼び出す関数のアドレスデータのラベルまでの距離となります。

"-tbr"オプション指定により、すべての関数呼び出しを TBR 相対呼び出しの対象とすることができます。また、プリプロセッサ制御文"#pragma tbr"を用いることで、各関数ごとに指定ができます。

TBR 相対関数呼び出しをする場合、$TBR セクションの先頭アドレスを TBR に設定する必要があります。

標準ライブラリを TBR 相対で関数呼び出ししたい場合は以下の手順を踏む必要があります。

(1) システムインクルードファイル"tbr.h"にTBR相対で関数呼び出ししたいライブラリを#pragma tbr指定で記述する。
(2) 標準ライブラリ作成ツールlbgshでライブラリを作成する。
(3) TBR相対で関数呼び出ししたいライブラリをコールしているソースで"tbr.h"を#includeで取り込む。

【書式】

＜オプション＞

```
-tbr[=<セクション名>]
```

＜プリプロセッサ制御文＞

```
#pragma tbr (<関数名>[(sn=<セクション名>|ov=<オフセット値>)][,...])
```

<オフセット値>は 0～1020 の範囲の 4 の倍数

■使用例

使用例 1

TBR 相対関数呼び出し指定がされた場合、コンパイラは、関数の呼び出しを TBR 相対呼び出しとし、ファイル中で定義された関数に対して、関数のアドレスデータとそのラベルからなるジャンプテーブルを生成します。

ジャンプテーブル中の関数アドレスデータのラベル名は、"$_"+<関数名> となります。また、static 関数については、"$_"+"_$"+<関数名> となります。

C 言語コード

```
/* -cpu=sh2a -size -tbr */
#include <machine.h>
void f1(){}
void f2(){}
static void f3(){}

main()
{
    set_tbr(__sectop("$TBR")); /* $TBRセクションの先頭をTBRに設定する */
    f1();
    f2();
    f3();
}
```

アセンブリ言語展開コード

```
_main:
        STS.L      PR,@-R15
        MOV.L      L14+2,R2   ; STARTOF $TBR
        LDC        R2,TBR
        JSR/N      @@($_f1-(STARTOF $TBR),TBR)    ; TBR相対関数呼び出し
        JSR/N      @@($_f2-(STARTOF $TBR),TBR)    ; TBR相対関数呼び出し
        JSR/N      @@($__$f3-(STARTOF $TBR),TBR)  ; TBR相対関数呼び出し
```

```
        LDS.L       @R15+,PR
        RTS/N
L14:
        .RES.W      1
        .DATA.L     STARTOF $TBR
        .SECTION    $TBR,DATA,ALIGN=4           ; TBR相対ジャンプテーブル
$_f1:
        .DATA.L     _f1                         ; 関数アドレスデータ
$_f2:
        .DATA.L     _f2                         ; 関数アドレスデータ
$_main:
        .DATA.L     _main                       ; 関数アドレスデータ
$__$f3:
        .DATA.L     __$f3                       ; static関数アドレスデータ
```

使用例２

"-tbr" オプションで、すべての関数を対象にする方法のほかに、"#pragma tbr"を用いることで、各関数ごとに指定ができます。

<関数名>で指定された関数の呼び出しが、TBR 相対呼び出しとなります。

"sn=<セクション名>" を指定した場合、関数アドレスデータは、"$TBR"＋<セクション名> のセクションに生成されます。

"ov=<オフセット値>" を指定した場合、TBR 相対値は、<オフセット値>の値になります。

C 言語コード

```
/* -cpu=sh2a -size */
#pragma tbr (f1(sn=X))
#pragma tbr (f2(ov=0))
f1(){}
f2(){}
main(){
f1();
f2();
}
```

アセンブリ言語展開コード

```
   _main:
        STS.L       PR,@-R15
        JSR/N       @@($_f1-(STARTOF $TBRX),TBR)
        JSR/N       @@(0,TBR)           ; TBR相対値は、0
        LDS.L       @R15+,PR
        RTS/N
        .SECTION    $TBRX,DATA,ALIGN=4   ; セクション名 "$TBRX"
$_f1:
        .DATA.L     _f1
                                        ; "ov=<オフセット値>"を指定した関数(f2)
                                        ; に対する関数アドレスデータは生成
                                        ; されません。（使用例３参照）
```

使用例３

"ov=<オフセット値>"が指定された関数に対する、TBR 相対ジャンプテーブル中の関数アドレスデータは、ユーザ自身で作成する必要があります。

また、同一ファイル内に関数定義がない場合は、関数定義側で同一の指定を行うか、TBR 相対ジャンプテーブル中の関数アドレスデータをユーザ自身で作成する必要があります。

C 言語コード

```
/* -cpu=sh2a */
#pragma tbr (func1(ov=0))   /* ジャンプテーブル内オフセット０を指定 */
#pragma tbr (func2(ov=4))   /* ジャンプテーブル内オフセット４を指定 */
#pragma tbr (func3(ov=8))   /* ジャンプテーブル内オフセット８を指定 */
extern void func1();
extern void func2();
extern void func3();

#pragma tbr (func4(sn=NEW))/* ジャンプテーブルのセクションに"$TBRNEW"を指定 */
#pragma tbr (func5(sn=NEW))
#pragma tbr (func6(sn=NEW))
extern void func4();
extern void func5();
extern void func6();

#include<machine.h>
void main()
{
    set_tbr(__sectop("$TBR"));    /* TBRに"$TBR"セクション先頭を設定する。*/
    func1();
    func2();
    func3();
    set_tbr(__sectop("$TBRNEW"));/* テーブルを "$TBRNEW" に切り替える。 */
    func4();
    func5();
    func6();
}
```

アセンブリ言語展開コード

```
_main:
         STS.L      PR,@-R15
         MOV.L      L11+2,R1   ; STARTOF $TBR
         LDC        R1,TBR
         JSR/N      @@(0,TBR)
         JSR/N      @@(4,TBR)
         JSR/N      @@(8,TBR)
         MOV.L      L11+6,R4   ; STARTOF $TBRNEW
         LDC        R4,TBR
         JSR/N      @@($_func4-(STARTOF $TBRNEW),TBR)
         JSR/N      @@($_func5-(STARTOF $TBRNEW),TBR)
         JSR/N      @@($_func6-(STARTOF $TBRNEW),TBR)
         LDS.L      @R15+,PR
         RTS/N
L11:
        .RES.W     1
         .DATA.L    STARTOF $TBR
         .DATA.L    STARTOF $TBRNEW
```

TBR 相対関数呼び出しを指定する場合、TBR の設定が必要になりますが、関数呼び出しをするときに参照する関数アドレスは、ジャンプテーブル上のデータを参照するので、データサイズを小さくすることができます。

以下は、TBR を使用しない場合の関数呼び出しコードです。

```
_main:
         STS.L      PR,@-R15
         MOV.L      L11,R1    ; _func1
```

```
        JSR/N      @R1
        MOV.L      L11+4,R4   ; _func2
        JSR/N      @R4
        MOV.L      L11+8,R5   ; _func3
        JSR/N      @R5
        MOV.L      L11+12,R6  ; _func4
        JSR/N      @R6
        MOV.L      L11+16,R7  ; _func5
        JSR/N      @R7
        MOV.L      L11+20,R2  ; _func6
        JMP        @R2
        LDS.L      @R15+,PR
L11:
        .DATA.L    _func1
        .DATA.L    _func2
        .DATA.L    _func3
        .DATA.L    _func4
        .DATA.L    _func5
        .DATA.L    _func6
```

アセンブリ言語コード（ジャンプテーブル１）

"pragma tbr"の"ov=<オフセット値>"の指定にあわせて、"$TBR"セクションに関数アドレスデータのジャンプテーブルを作成してください。

```
        .SECTION   $TBR,DATA,ALIGN=4   ;
        .DATA.L    _func1   ; ジャンプテーブル内オフセットを０とする。
        .DATA.L    _func2   ;                 オフセットを４とする。
        .DATA.L    _func3   ;                 オフセットを８とする。
```

アセンブリ言語コード（ジャンプテーブル２）

同一ファイル内に関数定義がない場合は、関数定義側で同一の指定を行うか、下記のようなジャンプテーブルを作成してください。

```
        .EXPORT    $_func4
        .EXPORT    $_func5
        .EXPORT    $_func6
        .SECTION   $TBRNEW,DATA,ALIGN=4
$_func4:                          ; ラベル名は、"$_"+<関数名> とする。
        .DATA.L    _func4         ; 関数アドレスデータ。
$_func5:
        .DATA.L    _func5
$_func6:
        .DATA.L    _func6
```

使用例４

CPU が SH-2A で printf を TBR 相対で関数呼び出しする場合

（1）tbr.hに"#pragma tbr printf"の記述を追加する。

```
     :
#if (defined(_SH2A) || defined(_SH2AFPU)) && !defined(_PIC)
     :
#pragma tbr printf     // ← 追加
     :
#endif  /* #if (defined(_SH2A) || defined(_SH2AFPU)) && !defined(_PIC) */
     :
```

（2）TBR相対テーブルを含んだ標準ライブラリを作成し、利用する。

```
lbgsh -cpu=sh2a
```

（3）printfを使っているプログラムにtbr.hをインクルードする記述を追加する

```
#include <tbr.h>       // ← 追加
#include <stdio.h>

main()
{
    printf("tbr¥n");
}
```

■注意事項

（1） BSR命令による関数呼び出しが可能な場合には、TBR相対呼び出しとなりません。ただし、"-size" オプション指定時は、TBR相対呼び出しとなります。
（2） "-cpu=sh2a" または、"-cpu=sh2afpu" オプション以外を指定した場合、TBR相対関数呼び出し指定は無効になります。
（3） "-pic=1" オプションを指定した場合には、関数の絶対アドレスが決定しないため、TBR相対関数呼び出し指定は無効になります。
（4） "-section" オプションで、セクション名にジャンプテーブルのセクション名である"$TBR"を指定した場合、オブジェクト実行時の動作は保障されません。
（5） 指定できる関数の数は、プログラム全体で、各セクション255個までです。
（6） "sn=<セクション名>"と"ov=<オフセット値>は、同じ関数に対して、同時に指定できません。
（7） 同一の関数に対して以下の#pragma 拡張子を同時に指定した場合はエラーとなります。

```
#pragma interrupt
#pargma inline
#pragma inline_asm
#pragma entry
```

3.21 GBR 相対論理演算命令生成

■説明

"-gbr=user" オプションを指定したとき、"-logic_gbr" オプションを指定することにより、"#pragma gbr_base" で指定された GBR ベース変数以外の外部変数に対する演算にも、GBR 相対の論理演算命令を使用します。

【書式】

```
-logic_gbr
```

■使用例

C 言語コード

```
char a,b,c;
main(){
    a &= 0x0f;
    b |= 0x01;
    c ^= 0x01;
}
```

アセンブリ言語展開コード（-gbr_user 指定あり、-logic_gbr 指定なし）

```
        MOV.L     L11+2,R6   ; _a
        MOV.B     @R6,R0
        AND       #15,R0
        MOV.B     R0,@R6
        MOV.L     L11+6,R6   ; _b
        MOV.B     @R6,R0
        OR        #1,R0
        MOV.B     R0,@R6
        MOV.L     L11+10,R6  ; _c
        MOV.B     @R6,R0
        XOR       #1,R0
        RTS
        MOV.B     R0,@R6
L11:
        .RES.W    1
        .DATA.L   _a
        .DATA.L   _b
        .DATA.L   _c
```

アセンブリ言語展開コード（-gbr_user 指定あり、-logic_gbr 指定あり）

```
        MOV.L     L11+2,R0   ; _a-(STARTOF $G0)
        AND.B     #15,@(R0,GBR)              ; GBR相対演算命令
        MOV.L     L11+6,R0   ; _b-(STARTOF $G0)
        OR.B      #1,@(R0,GBR)               ; GBR相対演算命令
        MOV.L     L11+10,R0  ; _c-(STARTOF $G0)
        RTS
        XOR.B     #1,@(R0,GBR)               ; GBR相対演算命令
L11:
        .RES.W    1
        .DATA.L   _a-(STARTOF $G0)
        .DATA.L   _b-(STARTOF $G0)
        .DATA.L   _c-(STARTOF $G0)
```

GBR をベースとするには、"#pragma gbr_base" を使用するときと同様に、あらかじめ GBR を$G0 セクションの先頭アドレスを設定しておく必要があります。

■注意事項

（1） "-logic_gbr" オプション指定時には、必ず $G0セクションを配置してください。

（2） "-gbr=user" オプションを指定しない場合、"-logic_gbr" オプション指定は無効になります。

3.22 register 宣言有効化

■説明

コンパイラは register 宣言の有無にかかわらず、コンパイラ内の解析結果に基づいた順序で変数にレジスタを割り付けます。

"-enable_register"オプションを指定すると register 宣言のある変数に優先的にレジスタを割り付けるようにします。

【書式】

-enable_register

■使用例

C 言語コード

```
int sum[10],input1[10],input2[10];
int b;

void func()
{
register int a = 0;
    int i;

    while(b) {
      a++;
      for (i = 0; i < 10; i++) {
          sum[i] = input1[i] + input2[i];
      }
      b--;
    }

    printf("%d¥n",a); // 'a'の値はR5経由でprintfに渡されるので
                       // 'a'にR5が割り付けられると都合が良い。
}
```

アセンブリ言語展開コード（-enable_register 指定なし）

```
_func:
        MOV.L     R12,@-R15
        MOV.L     R13,@-R15
        MOV.L     R14,@-R15
        MOV.L     L16+2,R12
        MOV       #0,R13      ; 優先度の高い別の変数にR5を割り付けたため、
                              ; 変数aにはR13を割り付けている
               :         :
        MOV.L     L16+22,R2   ; _printf
        MOV.L     R14,@R12
        MOV       R13,R5      ; 変数a(R13)の値をR5にコピーする
        MOV.L     @R15+,R14
        MOV.L     @R15+,R13
        JMP       @R2         ; printf()をコール
        MOV.L     @R15+,R12
```

アセンブリ言語展開コード（-enable_register 指定あり）

```
_func:
        MOV.L     R12,@-R15
        MOV.L     R13,@-R15
        MOV.L     R14,@-R15
```

```
MOV.L      L16,R12    ; _b
MOV        #0,R5      ; 変数aの優先度が上がったのでR5を割り付けている
    :          :
MOV.L      L16+20,R2  ; _printf
MOV.L      R13,@R12
MOV.L      @R15+,R14
MOV.L      @R15+,R13
JMP        @R2        ; printf()をコール
MOV.L      @R15+,R12
```

■注意事項

レジスタに割り付けられなかった場合は、インフォメーションメッセージ C0102 (I) Register is not allocated to "変数名" in "関数名"を出力します。ただし、引数がレジスタに割り付けられなかった場合は、本メッセージは出力しません。

3.23 変数の絶対アドレス指定

■説明

プリプロセッサ制御文を用いて外部参照される変数を絶対アドレス指定することができます。コンパイラは"#pramga address"で宣言された変数を対応付けられた絶対アドレスに割り付けるようにします。この機能で特定のアドレスに割り付けられた I/O などを変数を通じて容易にアクセスすることを実現できます。

【書式】

`#pragma address` （<変数名>＝<アドレス値>[,<変数名>＝<アドレス値> ・・・] ）

■使用例

変数 io を絶対アドレス 0x100 番地に割り付けます。

C 言語コード

```
#pragma address (io=0x100)
int io;
f()
{
    io = 10;
}
```

アセンブリ言語展開コード

```
_func:
        MOV        #1,R2
        SHLL8      R2
        MOV        #10,R6
        RTS
        MOV.L      R6,@R2
        .SECTION   $ADDRESS$B100,DATA,LOCATE=H'100
_io:
        .RES.L     1
```

■注意事項

（1） #pragma address 指定は、変数の宣言前に行ってください。
（2） 複合型のメンバ、もしくは変数以外を指定した場合はエラーとなります。
（3） 境界調整数2の変数、構造体に奇数アドレスを指定した場合、境界調整数4の変数、構造体に4の倍数以外のアドレスを指定した場合は、エラーとなります。
（4） #pragma address を同一の変数に対して複数回指定した場合はエラーとなります。
（5） 異なる変数に対して同一アドレスを指定した場合、もしくは変数のアドレスが重なった場合はエラーとなります。
（6） 同一の変数に対して以下の#pragma 拡張子を同時に指定した場合はエラーとなります。

```
#pragma section
#pragma abs16/abs20/abs28/abs32
#pragma gbr_base/gbr_base1
#pragma global_register
```

3.24 最適化強化

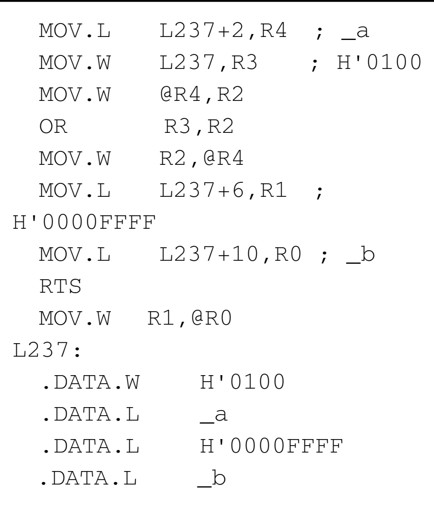
3.24.1 リテラルデータ改善（1）

定数データ最適化を強化しました。

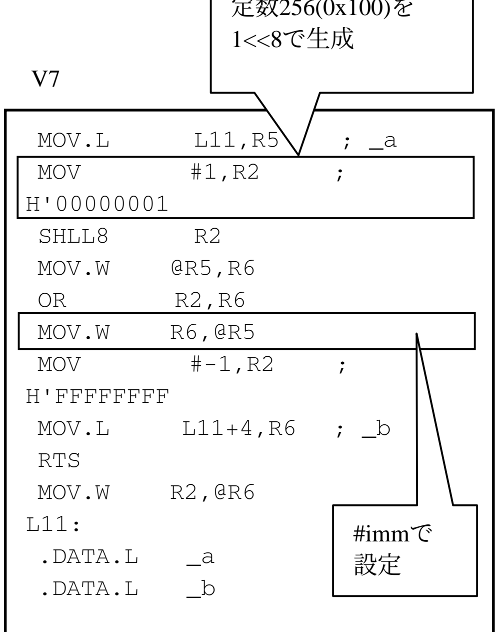
Source Program

```
unsigned
short a,b;
f(){
  a|=0x100;
  b=0xffff;
}
```

V5,V6



V7



3.24.2 リテラルデータ改善（2）

2 byte 以上の定数値を再利用します。

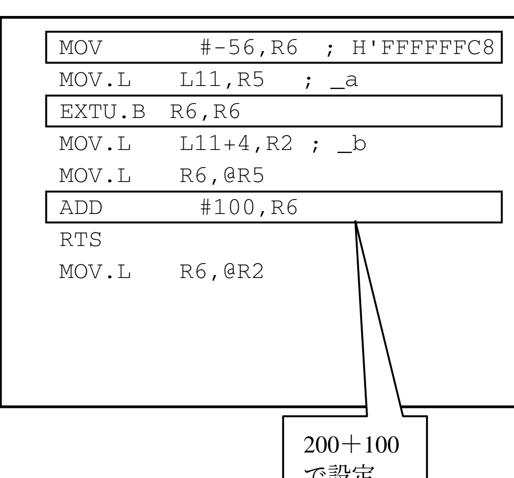
Source Program

```
unsigned
short a,b;
f(){
  a=200;
  b=300;
}
```

V5,V6

```
  MOV.W   L237,R3    ; H'00C8
  MOV.L   L237+6,R2  ; _a
  MOV.W   L237+2,R1  ; H'012C
  MOV.L   R3,@R2
  MOV.L   L237+10,R0 ; _b
  RTS
  MOV.L   R1,@R0
L237:
  .DATA.W H'00C8 ; 200
  .DATA.W H'012C ; 300
```

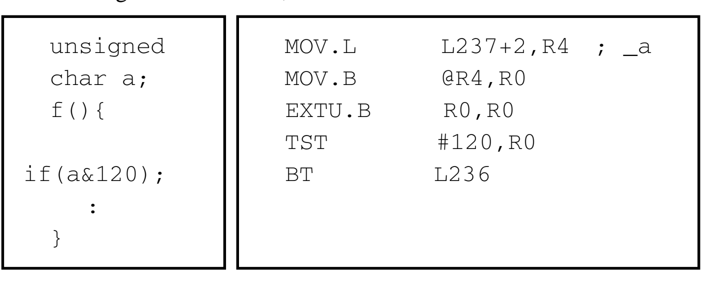
V7

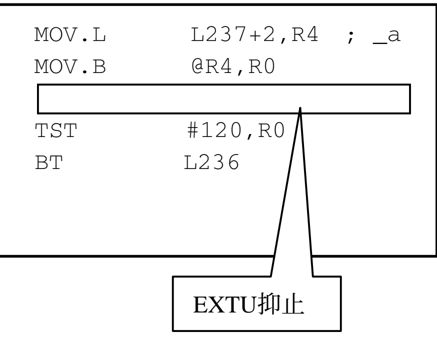


3.24.3 EXTU 抑止（1）

条件式中の論理積結果に対する EXTU を抑止します。

Source Program　　V5,V6　　V 7

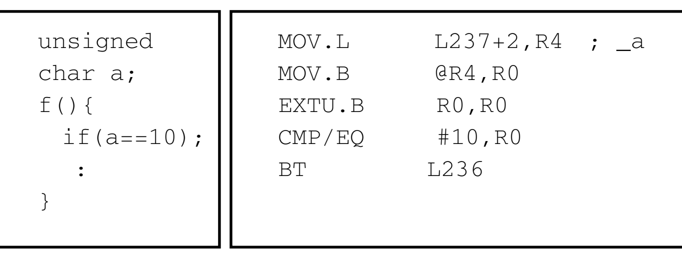


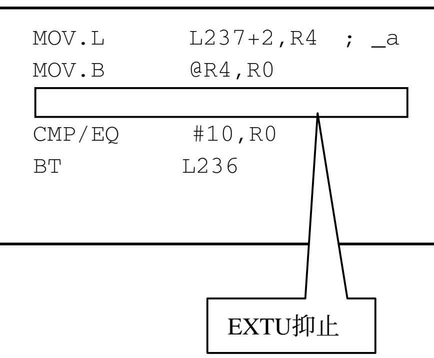


3.24.4 EXTU 抑止（2）

定数との比較時の EXTU を抑止します。

Source Program　　V5,V6　　V 7

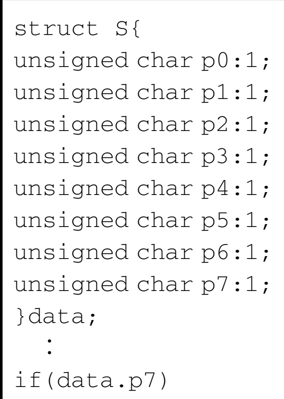


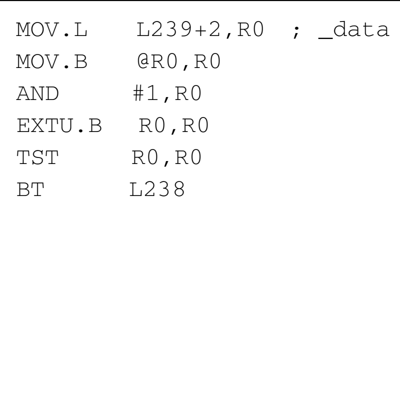


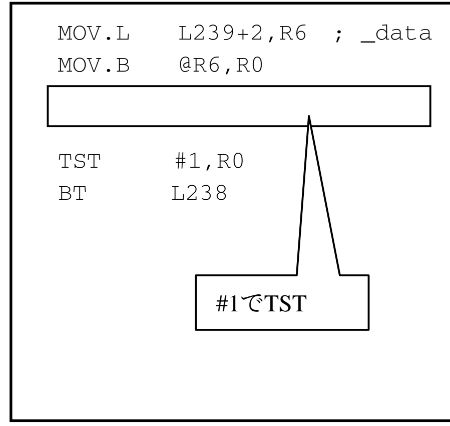
3.24.5 ビット演算改善（1）

1 bit データの比較コードを改善

Source Program　　V5,V6　　V 7







3.24.6 ビット演算改善（2）

1 bit データの代入コードを改善

Source Program

```
struct S{
unsigned char p0:1;
unsigned char p1:1;
unsigned char p2:1;
unsigned char p3:1;
unsigned char p4:1;
unsigned char p5:1;
unsigned char p6:1;
unsigned char p7:1;
  }data1,data2;
    :
  data1.p7=data2.p6
;
```

V5,V6

```
 STS.L      PR,@-R15
 MOV.L      L239+4,R0  ; _data2
 MOV.L      L239+8,R2  ; _data1
 MOV.B      @R0,R0
 MOV.W      L239,R1    ; H'0701
 TST        #2,R0
 MOV.L      L239+12,R3 ; __bfsbu
 MOVT       R0
 ADD        #-1,R0
 JSR        @R3
 NEG        R0,R0
 LDS.L      @R15+,PR
 RTS
 NOP
L239:
 .DATA.W    H'0701
 .DATA.L    __bfsbu0
```

V 7

```
 MOV.L   L14+2,R6  ; _data2
 MOV.B   @R6,R0
 MOV.L   L14+6,R6  ; _data1
 TST      #2,R0
 MOV.B   @R6,R0
 BF       L12
 BRA     L13
 AND     #254,R0
L12:
 OR      #1,R0
L13:
 RTS
 MOV.B   R0,@R6
```

実行時ルーチンを使用せず
インラインで展開

3.24.7 ビット演算改善（3）

bit フィールドの論理演算コードを改善

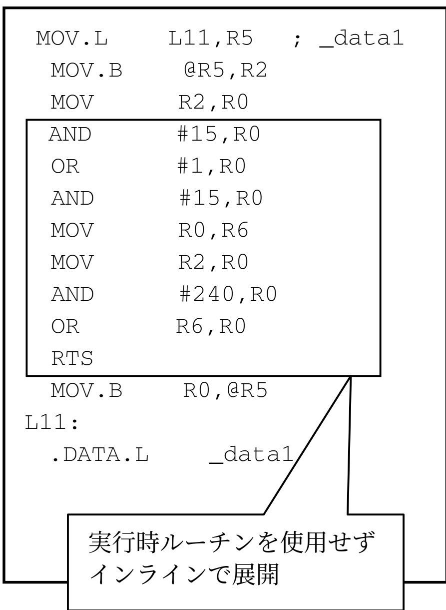
Source Program

```
struct S{
unsigned char
p0:4;
unsigned char
p1:4;
}data1;
  :
data1.p1|=1;
```

V5,V6

```
 STS.L      PR,@-R15
 MOV.L    L236+4,R0  ; _data1
 MOV.L    L236+4,R2  ; _data1
 MOV.B    @R0,R0
 MOV.W    L236,R1    ; H'0404
 AND      #15,R0
 MOV.L    L236+8,R3  ; __bfsbu
 JSR       @R3
 OR        #1,R0
 LDS.L      @R15+,PR
 RTS
 NOP
L236:
 .DATA.W  H'0404
 .DATA.W  0
 .DATA.L   _data1
 .DATA.L   __bfsbu
```

V 7



```
MOV.L     L11,R5   ; _data1
 MOV.B     @R5,R2
 MOV       R2,R0
 AND       #15,R0
 OR        #1,R0
 AND       #15,R0
 MOV       R0,R6
 MOV       R2,R0
 AND       #240,R0
 OR        R6,R0
 RTS
 MOV.B     R0,@R5
L11:
 .DATA.L     _data1
```

3.24.8 ビット演算改善（4）

同一 bit フィールドの連続判定処理を改善

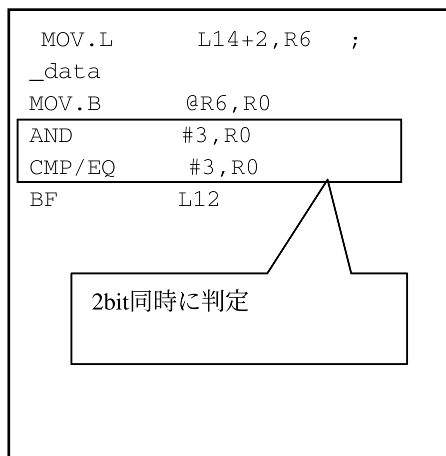
Source Program

```
struct S{
unsigned char p0:1;
unsigned char p1:1;
unsigned char p2:1;
unsigned char p3:1;
unsigned char p4:1;
unsigned char p5:1;
unsigned char p6:1;
unsigned char p7:1;
}data;
   :
if(data.p7==1 &&
     data.p6==1)
```

V5,V6

```
MOV.L       L239+2,R4  ; _data
MOV         R4,R0
MOV.B       @R0,R0
AND         #1,R0
CMP/EQ      #1,R0
BF          L238
MOV         R4,R0
MOV.B       @R0,R0
TST         #2,R0
MOVT        R0
ADD         #-1,R0
NEG         R0,R0
CMP/EQ      #1,R0
BF          L238
```

V 7



3.24.9 ビット演算改善（5）

同一 bit フィールドのへの連続代入の改善

Source Program

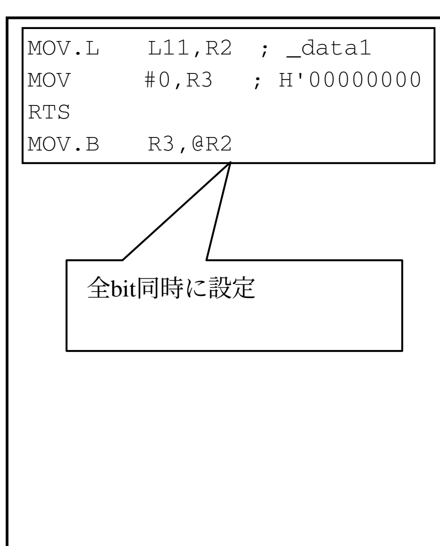
```
struct S{
unsigned char p0:1;
unsigned char p1:1;
unsigned char p2:1;
unsigned char p3:1;
unsigned char p4:1;
unsigned char p5:1;
unsigned char p6:1;
unsigned char p7:1;
}data;
   :
data.p0=0;
data.p1=0;

:
data.p7=0;
```

V5,V6

```
MOV.L       L240,R4    ; _data1
MOV.B       @R4,R0
AND         #127,R0
MOV.B       R0,@R4
MOV.B       @R4,R0
AND         #191,R0
MOV.B       R0,@R4
   :
MOV.B       @R4,R0
AND         #254,R0
RTS
MOV.B       R0,@R4
```

V 7



RENESAS

3.25 未初期化変数の出力順制御

■説明

-bss_order オプションを使用することにより未初期化変数を宣言順、または定義順に割り付けることができます。

-bss_order=declaration を指定すると宣言順に未初期化変数が割り付けられ、-bss_order=definition を指定すると定義順に未初期化変数が割り付けられます。本オプション省略時解釈は、-bss_order=declaration です。

【書式】

```
-bss_order={ declaration | definition }
```

■使用例

C 言語コード

```
extern int a1;
extern int a2;
int a3;
extern int a4;
int a5;
int a2;
int a1;
int a4;
```

アセンブリ言語展開コード

＜bss_order=declaration 指定時＞

```
          .SECTION B,DATA,ALIGN=4
_a1:
          .RES.L 1
_a2:
          .RES.L 1
_a3:
          .RES.L 1
_a4:
          .RES.L 1
_a5:
          .RES.L 1
```

＜bss_order=definition 指定時＞

```
            .SECTION B,DATA,ALIGN=4
_a3:
            .RES.L 1
_a5:
            .RES.L 1
_a2:
            .RES.L 1
_a1:
            .RES.L 1
_a4:
            .RES.L 1
```

■注意事項

`stuff` オプションを指定した場合、常に `bss_order=definition` が有効になります。

3.26 変数の配置指定

■説明

-stuff オプションを使用することで変数を境界調整数別のセクションに配置することができます。これによりパディングがなくなりメモリを節約することができます。

-stuff オプションではセクション種別を指定することができます。指定したセクション種別に属する変数をデータサイズに応じて境界調整数４のセクション、２のセクション、１のセクションに配置します。セクション種別を省略した場合はすべての変数が対象になります。

各セクション内のデータは定義順に出力されます。（bss_order=declaration 指定は無効）。

-nostuff を指定した場合は全ての変数を境界調整数４のセクションに配置します。

各セクション内のデータは C,D セクションは定義順、B セクションは bss_order に従います。

本オプション省略時解釈は nostuff です。

表 3.48 変数のサイズとセクション名の関係

	セクション種別	変数のサイズ（Byte）		
		4n	4n+2	2n+1
const 変数	const	C$4	C$2	C$1
初期値あり変数	data	D$4	D$2	D$1
初期値なし変数	bss	B$4	B$2	B$1

【書式】

```
-stuff [=<セクション種別>[,...]]
-nostuff
<セクション種別>：{ Bss | Data | Const }
```

■使用例

C言語コード

```
int a;
char b=0;
const short c=0;
struct {
char x;
char y;
} ST;
```

アセンブリ言語展開コード

```
            .SECTION C$2,DATA,ALIGN=2
_c:
            .DATA.W  H'0000
            .SECTION D$1,DATA,ALIGN=1
_b:
            .DATA.B  H'00
            .SECTION B$4,DATA,ALIGN=4
_a:
            .RES.L  1
            .SECTION B$2,DATA,ALIGN=2
_sr:
            .RES.B  2
```

■注意事項

#pragma gbr_base|gbr_base1 または#pragma global_register を指定した変数は、本オプションの対象外になります。

4. HEW

4.1 HEW2.0 以降のオプション指定方法

ビルドメニューからオプションを指定することができます。統合化環境からのオプション指定方法を示します。ビルドメニューから SuperH RISC engine Standard Toolchain を選択します。

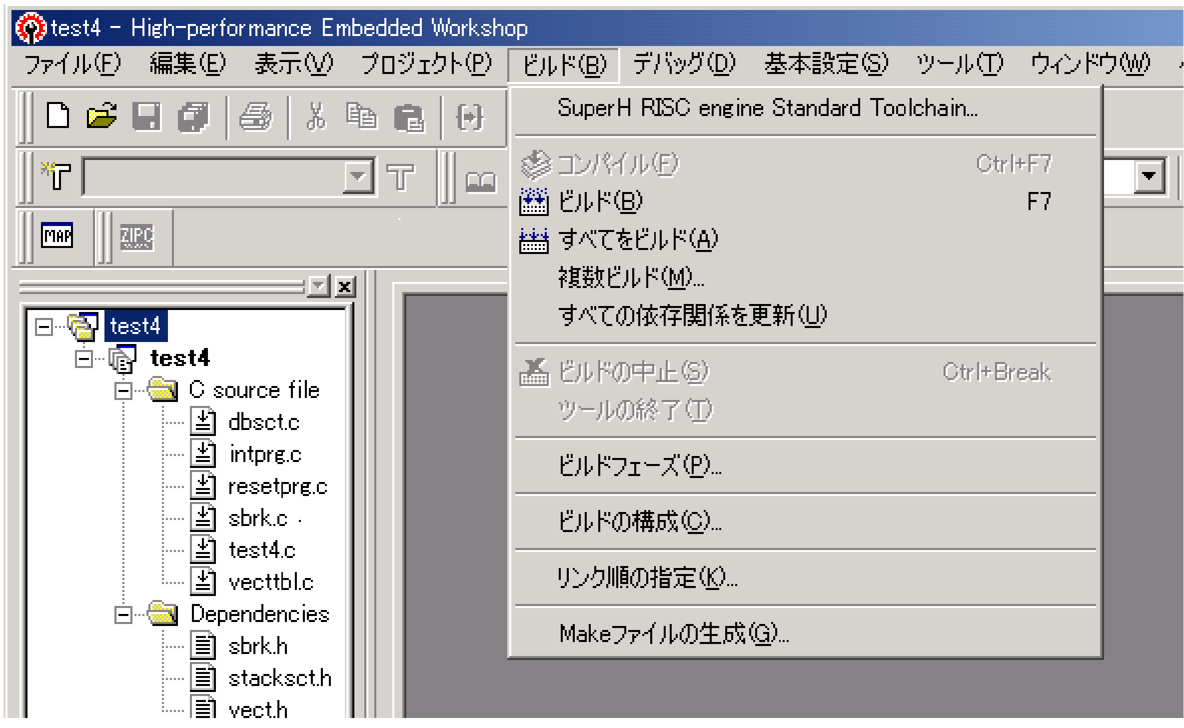


図 4.1 HEW ビルドメニュー

4.1.1 C/C++コンパイラのオプション

SuperH RISC engine Standard Toolchain ダイアログボックスからコンパイラタブを選択します。

(1) カテゴリ:[ソース]

表 4.1 カテゴリ:[ソース]の項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
オプション項目：	
インクルードファイルディレクトリ	Include = <**パス名**>[,…]
デフォルトインクルードファイル	PREInclude = <**ファイル名**>[,…]
マクロ定義	DEFine = <sub>[,…] <sub> : <**マクロ名**> [= <**文字列**>]
インフォメーションメッセージ	MEssage
メッセージレベル	CHAnge_message = <sub>[,…] <sub>:<level> [= <n>[-m],…] <level>:{Information | Warning | Error }
ファイル間インライン展開ディレクトリ	FILE_INLINE_PATH = <**パス名**>[,…]
インフォメーションレベルメッセージの表示	NOMEssage [= <**エラー番号**> [- <**エラー番号**>[,…]]

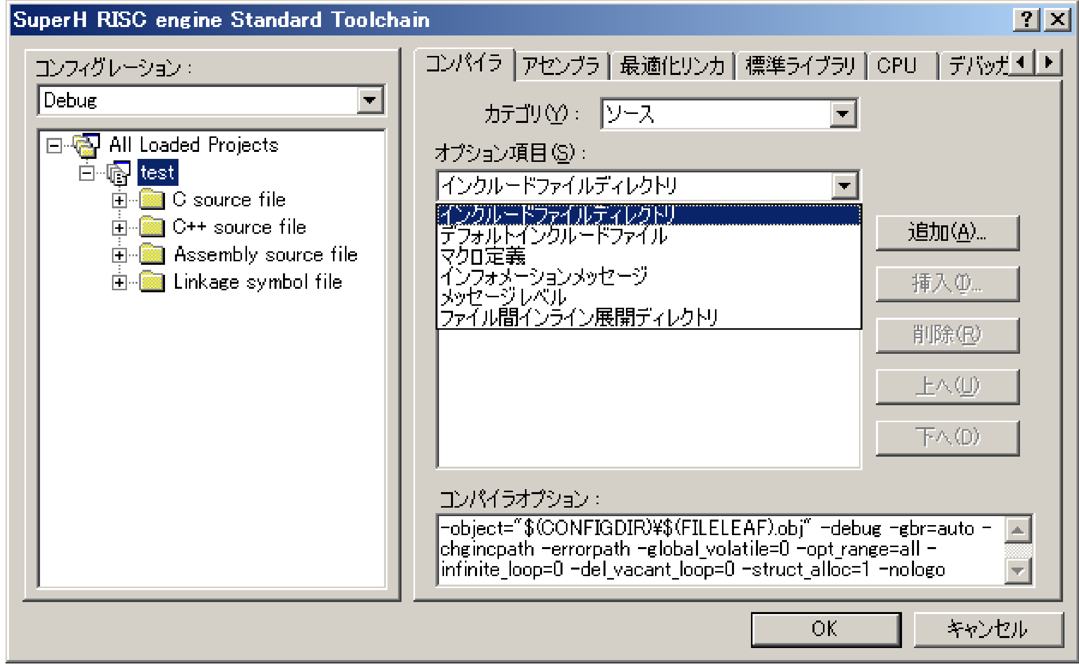


図 4.2 カテゴリ:[ソース] のダイアログボックス

(2) カテゴリ:[オブジェクト]

表 4.2 カテゴリ:[オブジェクト]の項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
出力ファイル形式：	
機械語プログラム (*.obj)	Code = Machinecode
アセンブリプログラム (*.src)	Code = Asmcode
プリプロセッサ 展開プログラム (*.p/*.pp)	PREProcessor [= <**ファイル名**>]
プリプロセッサ 展開プログラム (#line **出力抑止**)	NOLINe
デバッグ情報出力：	DEBug / NODEBug
オブジェクト出力ディレクトリ：	OBjectfile = <**ファイル名**>

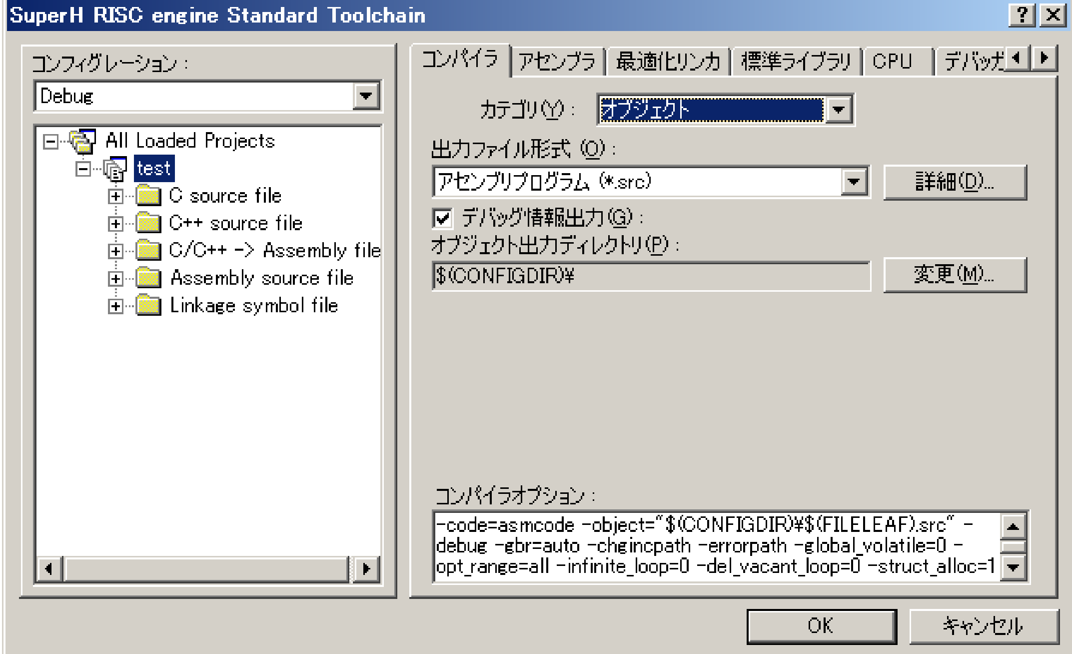


図 4.3 カテゴリ:[オブジェクト]のダイアログボックス

RENESAS

[詳細…]をクリックすると、”Object details”ダイアログボックスが表示されます。

(a) **詳細タブ**

表 4.3 項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
セクション： **プログラム領域** (P) **定数領域** (C) **初期化データ領域** (D) **未初期化データ領域** (B)	SEction = <sub>[,…] <sub> : Program = <**セクション名**> <sub> : Const = <**セクション名**> <sub> : Data = <**セクション名**> <sub> : Bss = <**セクション名**> **省略時：**(p=P, c=C, d=D, b=B)
テンプレート生成：	Template = { None | Static | Used | ALl | AUto }
文字列データ格納：	STring = { Const | Data }
除算方式選択：	DIvision = Cpu [= { Inline | Runtime }]
浮動小数点レジスタ退避・回避抑止	IFUnc
ラベルの 16 / 32 **バイト整合：**	ALIGN16 ALIGN32 NOALign

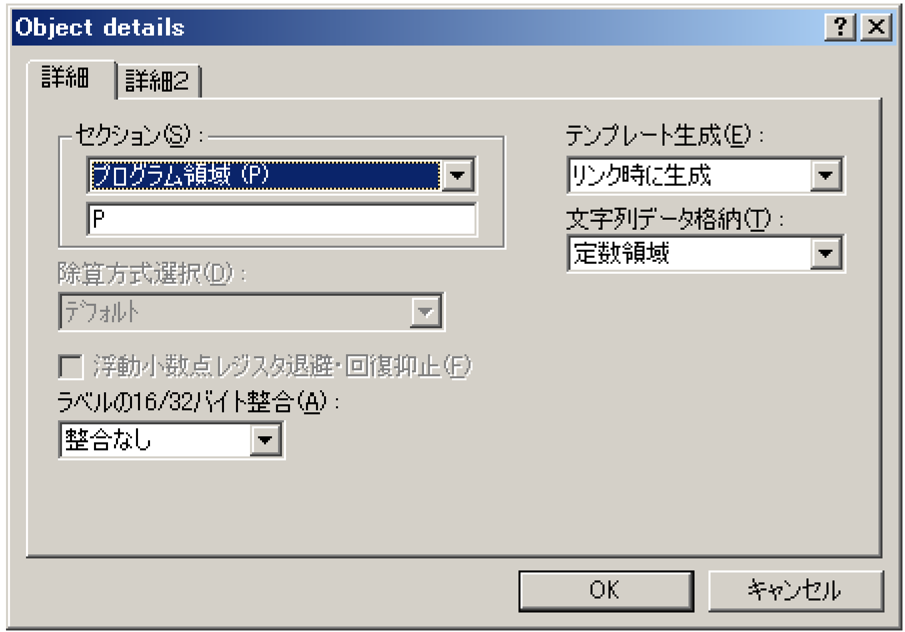


図 4.4 詳細タブのダイアログボックス

(b) **詳細 2 タブ**

表 4.4 項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
アドレス宣言：	<ABS> = <sub>[,…] <ABS>: { ABs16 | ABS20 | ABS28 | ABS32 } <sub>: { Program | Const | Data | Bss | Run | All }
TBR **指定：**	TBR[=<**セクション名**>]
変数の配置指定：	STUff=<sub>[,...] <sub>: {Bss|Data|Const}
未初期化変数の出力順：	BS**s**_order=<sub> <sub>: {DEClaration|DEFinition}

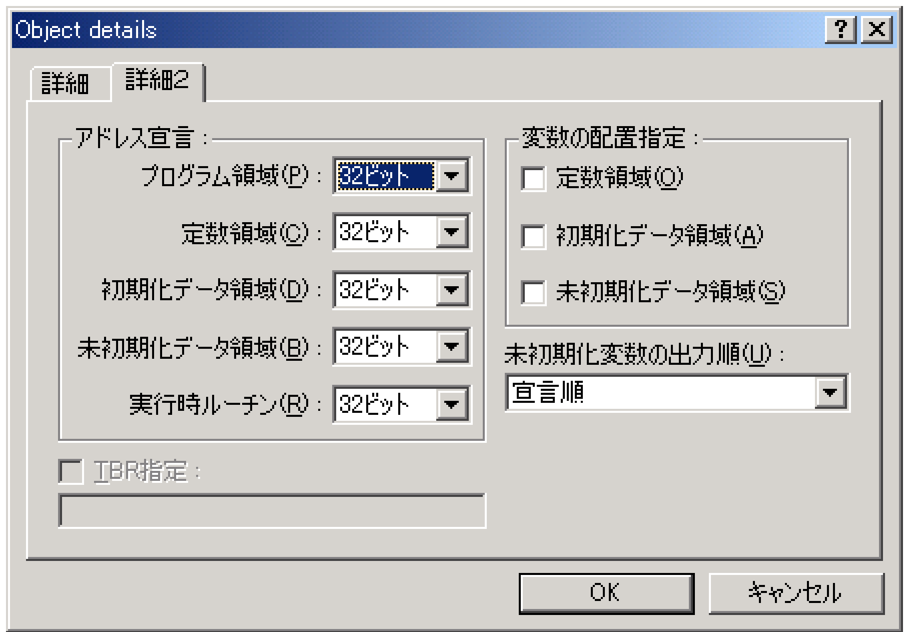


図 4.5 詳細 2 タブのダイアログボックス

(3) **カテゴリ:[リスト]**

表 4.5 カテゴリ:[リスト]の項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
コンパイルリスト出力	Listfile [= <**ファイル名**>] / NOListfile
タブサイズ ：	SHow = <sub>[,…] <sub> : Tab = { 4 | 8 }
リスト内容： **オブジェクトプログラムのリストを出力** **統計情報のリストを出力** **ソースリストを出力** **インクルード展開後のリストを出力** **マクロ展開後のリストを出力**	SHow = <sub>[,…] <sub> : Object / NOObject <sub> : STatistics / NOSTatistics <sub> : SOurce / NOSOurce <sub> : Include / NOInclude <sub> : Expansion / NOExpansion

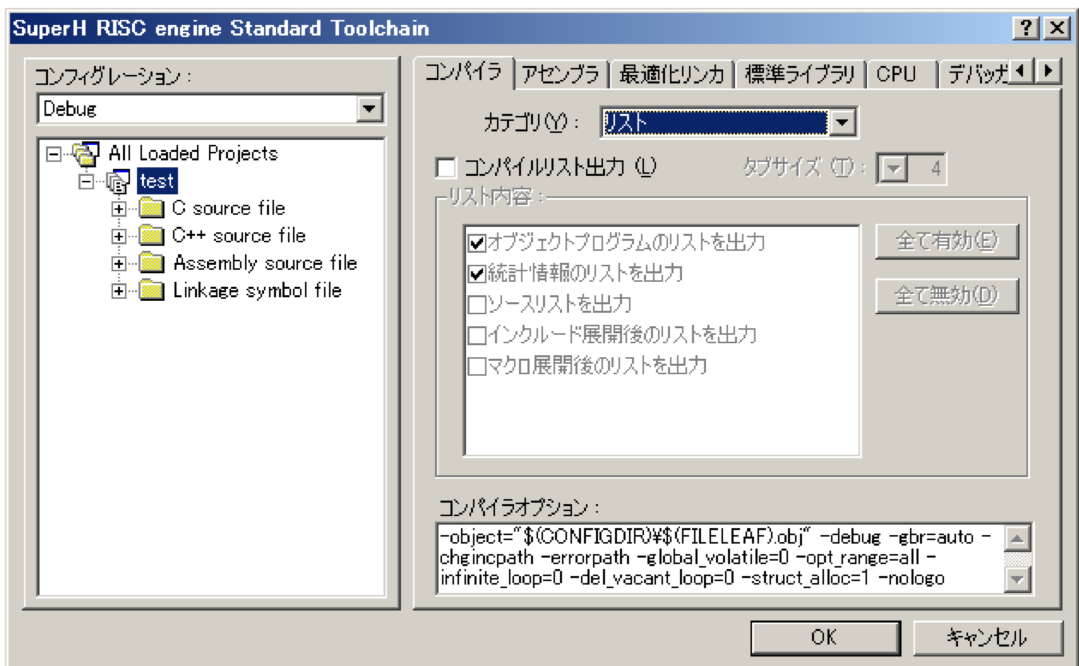


図 4.6 カテゴリ:[リスト]のダイアログボックス

-nolist と-show オプションでは-nolist オプションが優先して評価されます。

RENESAS

(4) カテゴリ:[最適化]

表 4.6 カテゴリ:[最適化]の項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
最適化	OPtimize = 1 / OPtimize = 0
最適化方法： 　スピード優先 　サイズ優先 　サイズ＆スピード	 SPeed SIze NOSPeed
モジュール間最適化	Goptimize
外部変数アクセス最適化：	MAP = <ファイル名>
GBR 相対アクセス最適化：	GBr = { Auto | User }
非境界調整データ転送：	Unaligned = { Inline | Runtime }
switch 文展開：	CAse = { Ifthen | Table }
シフト演算展開：	SHIft = { Inline | Runtime }
転送コード展開：	BLOckcopy = { Inline | Runtime }

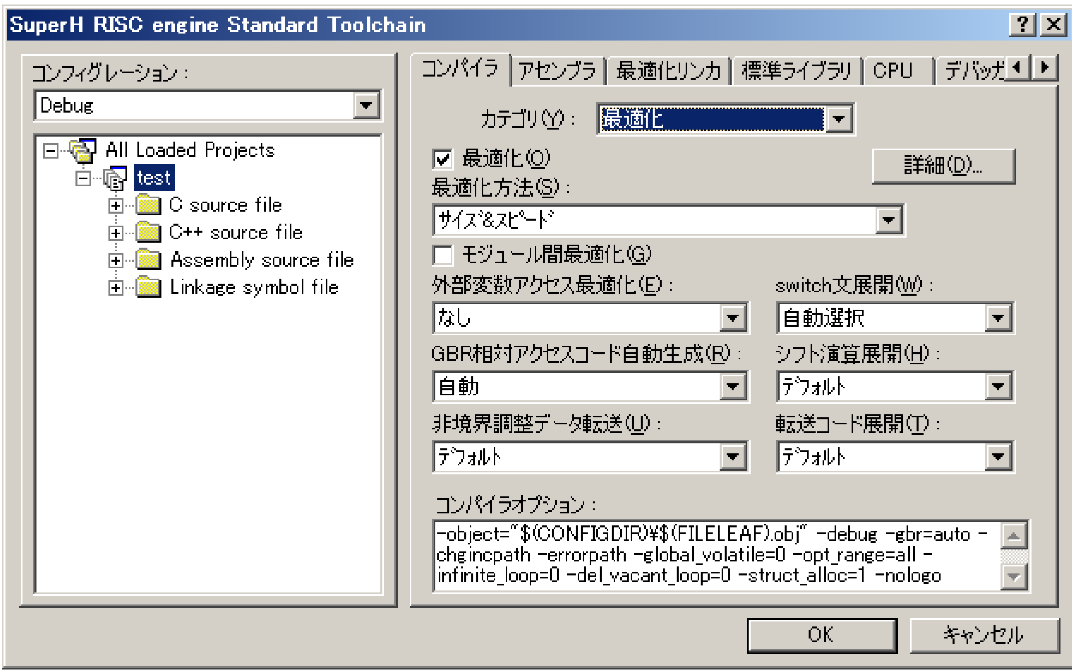


図 4.7 カテゴリ:[最適化]のダイアログボックス

RENESAS

[詳細…]をクリックすると、”Optimize details”ダイアログボックスが表示されます。
V.7.0.06 で追加されたオプションはここで指定します。

(a) インライン展開タブ

表 4.7 項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
インライン展開	
インライン展開ファイル：	FILe_inline = <**ファイル名**>[,…]
自動インライン展開ファイル：	INLine[=<**数値**>] / NOINLine

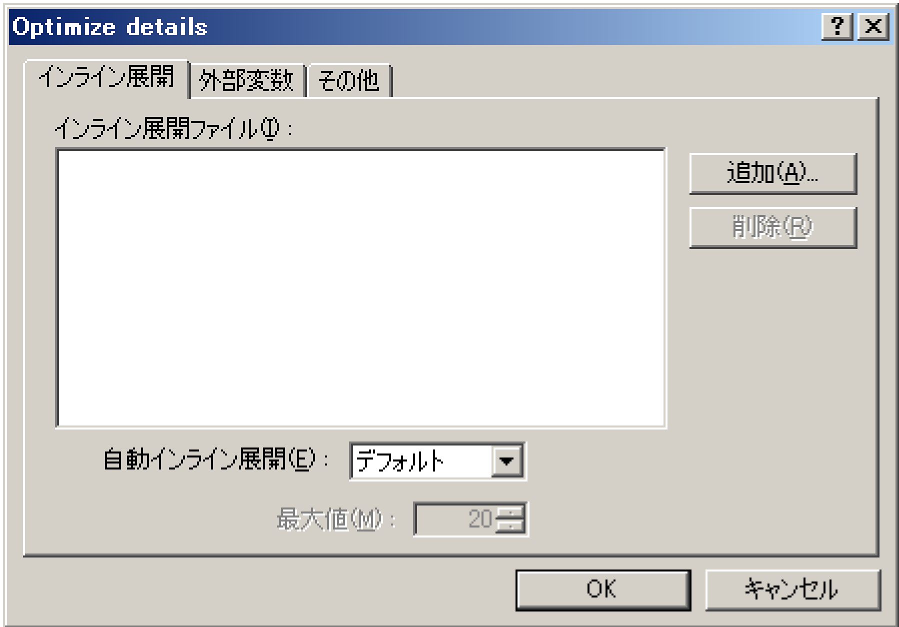


図 4.8 インライン展開タブのダイアログボックス

(b) **外部変数タブ**

表 4.8　項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
レベル:	-
設定項目：	
外部変数の volatile 化	GLOBAL_Volatile = 1 / GLOBAL_Volatile = 0
無限ループ前の外部変数への代入式削除	INFinite_loop = 1 / INFinite_loop = 0
外部変数への最適化範囲　：	OPT_Range = { All | NOLoop | NOBlock }
外部変数へのレジスタ割り付け　：	
抑止	GLOBAL_Alloc = 0
割り付け	GLOBAL_Alloc = 1
デフォルト	-
外部変数の定数伝播　：	
抑止	CONST_Var_propagate = 0
定数伝播	CONST_Var_propagate = 1
デフォルト	-
命令並べ替え：	
抑止	SChedule = 0
有効	SChedule = 1
デフォルト	-

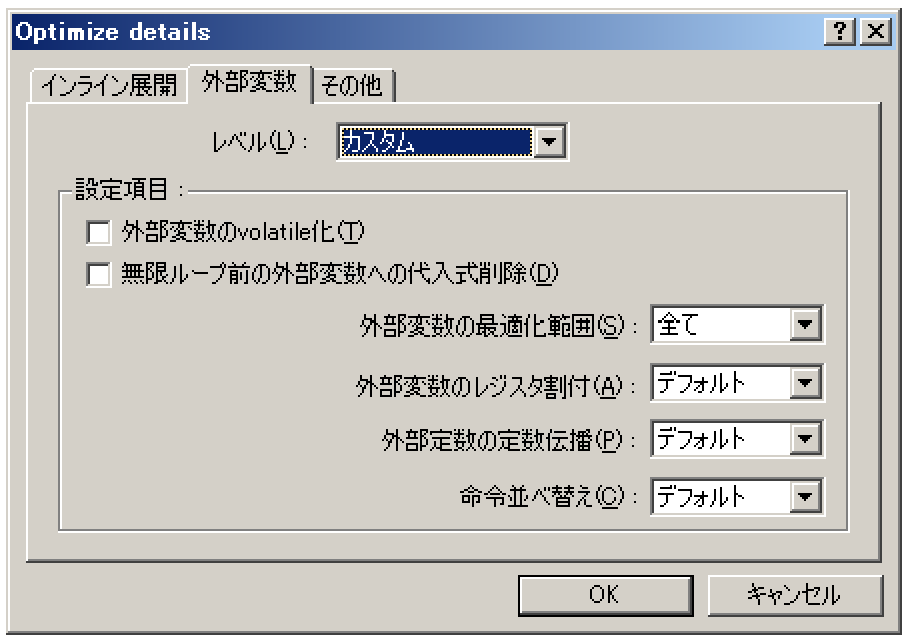


図 4.9　外部変数タブのダイアログボックス

Levelを設定することにより、外部変数に対する最適化を一括制御することができます。

Level1

外部変数の最適化をすべて抑止する。

gloal_volatile = 1
opt_range = noblock
infinite_loop = 0
global_alloc = 0
const_var_propagate = 0
schedule = 0

Level2

volatile指定のない外部変数を分岐（ループを含む）を超えない範囲で最適化する。

gloal_volatile = 0
opt_range = noblock
infinite_loop = 0
global_alloc = 0
const_var_propagate = 0
schedule = 1

Level3

volatile指定のない外部変数をすべて最適化対象とする。

gloal_volatile = 0
opt_range = all
infinite_loop = 0
global_alloc = 1
const_var_propagate = 1
schedule = 1

Custom

外部変数に対する最適化をプログラムにあわせてユーザが指定する。

(c) **その他タブ**

表 4.9 **項目名とコンパイラオプションの対応表**

ダイアログボックス	オプション
空ループ削除	DEL_vacant_loop = 1 / DEL_vacant_loop = 0
ループ最大展開数：	MAX_unroll = <数値> : 1-32
定数ロードの命令展開：	
インライン	CONST_Load = Inline
リテラル	CONST_Load = Literal
デフォルト	-
構造体／共用体メンバのレジスタ割り付け	STRUCT_Alloc = 1 / STRUCT_Alloc = 0
ソフトウェアパイプライニング	SOftpipe

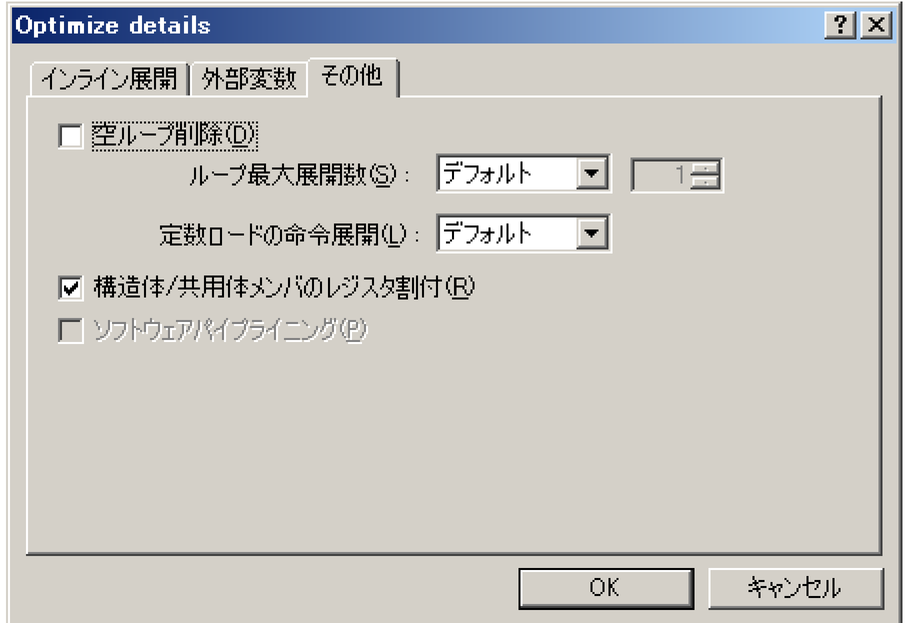


図 4.10 **その他タブのダイアログボックス**

(5) **カテゴリ:[その他]**

表 4.10 カテゴリ:[その他]の項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
その他のオプション：	
EC++ 言語に基づいたチェック	ECpp
DSP-C 言語に基いたチェック	DSpc
コメント(/＊＊/)のネクストを許す	COMment = Nest / COMment = NONest
MAC レジスタを保証する	Macsave = 1 / Macsave = 0
SSR / SPC退避・回復	SAve_cont_reg = 0 / SAve_cont_reg = 1
返却値の拡張を行う	RTnext / NORTnext
ループ展開	LOop / NOLOop
浮動小数点定数除算の乗算化	APproxdiv
SH7055 不具合回避	PAtch = 7055
FPSCR レジスタの切り替え	FPScr = Safe / FPScr = Aggressive
ループ判定式の最適化抑止	Volatile_loop
列挙型サイズの自動選択	AUto_enum
浮動小数点→固定小数点変換	FIXED_Const
1.0→__fixed 型最大値変換	FIXED_Max
__fixed 型乗算結果の型変換省略	FIXED_Noround
DSP 拡張リピートループ	REPeat
Register 指定変数の優先レジスタ割り付け	ENAble_register
ANSI 準拠対応拡張	STRIct_ansi
整数除算の浮動小数点除算置き換え	FDIv

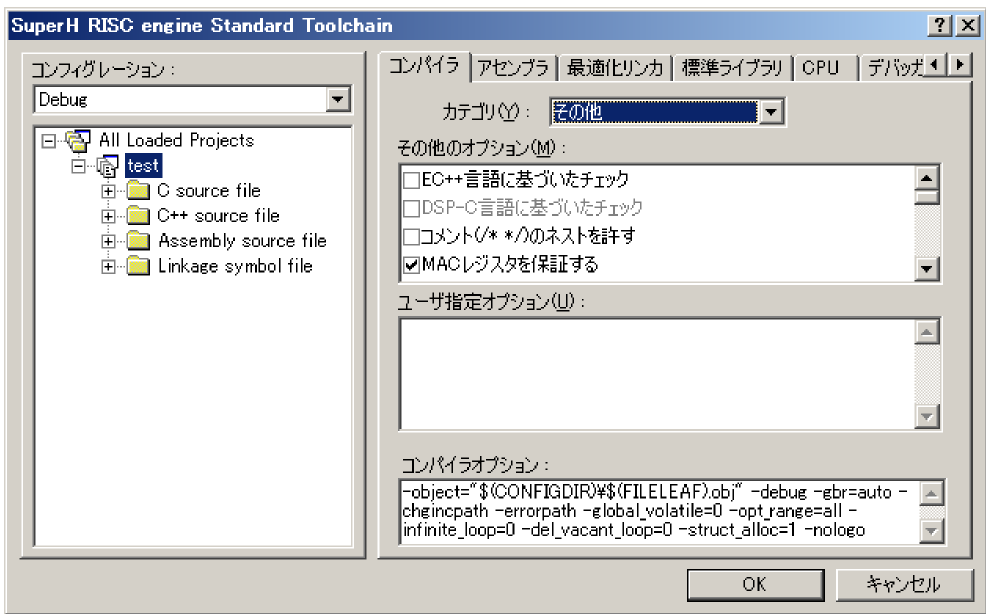


図 4.11 カテゴリ:[その他]のダイアログボックス

RENESAS

4.1.2 アセンブラのオプション

SuperH RISC engine Standard Toolchain ダイアログボックスからアセンブラタブを選択します。

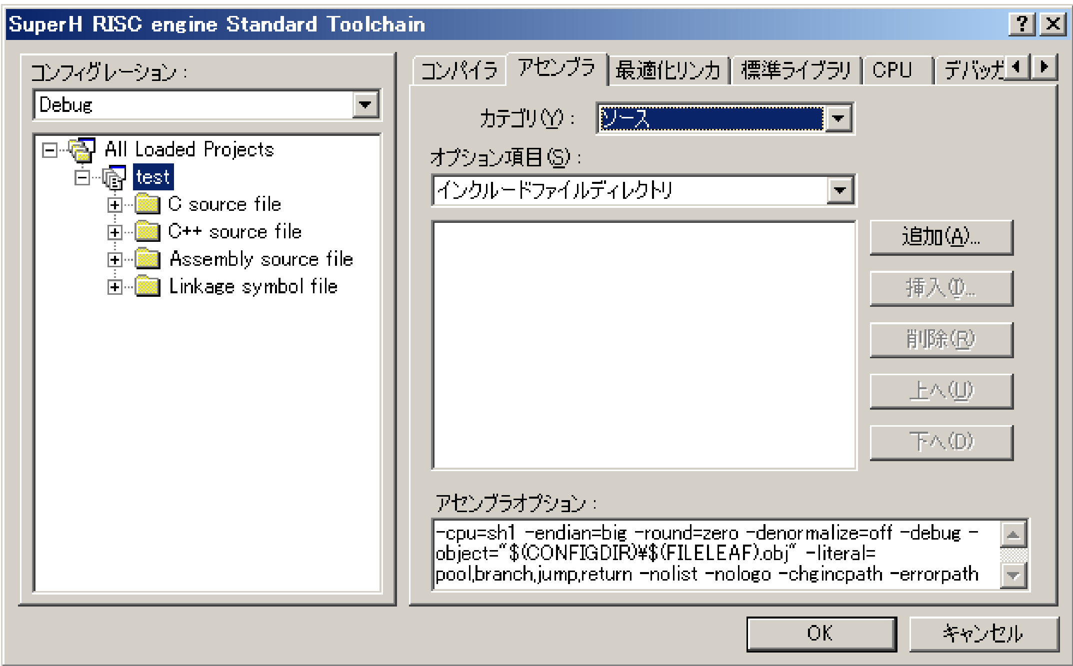


図 4.12 アセンブラタブのダイアログボックス

(1) カテゴリ:[ソース]

表 4.11 カテゴリ:[ソース]の項目名とアセンブラオプションの対応表

ダイアログボックス	オプション
オプション項目：	
インクルードファイルディレクトリ	Include = <**パス名**>[, …]
シンボル定義	DEFine = <sub>[, …] <sub> : <**置換シンボル**> = "<**文字列**>"
プロプロセッサ変数定義	ASsignA = <sub>[, …] <sub> : <**変数名**> = <**整数定数**> ASsignC = <sub>[, …] <sub> : <**変数名**> = "<**文字列**>"

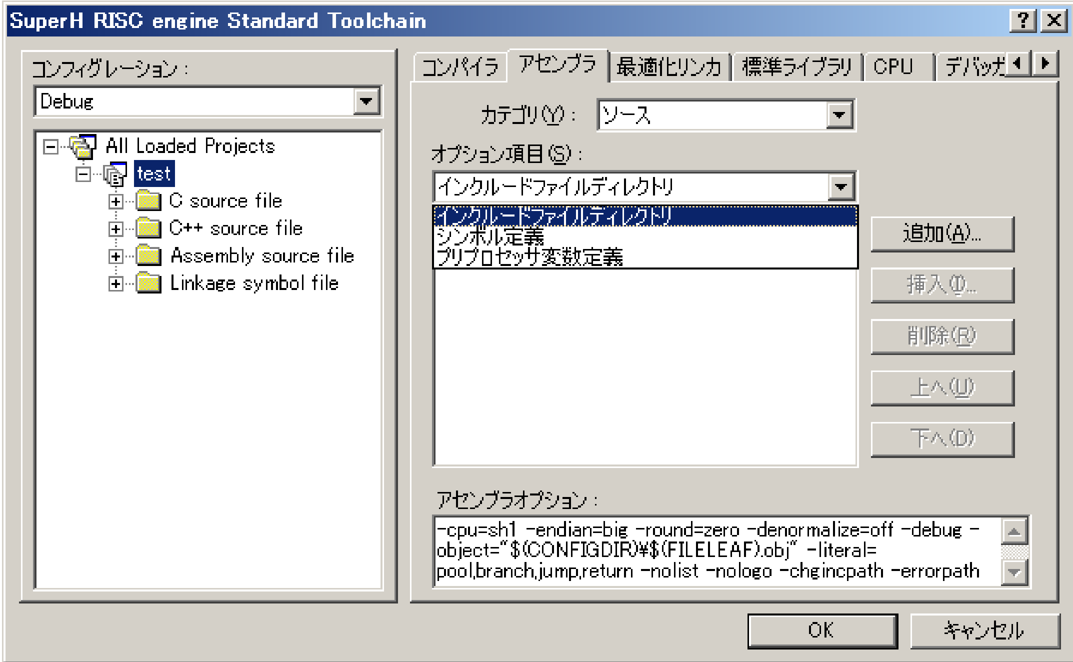


図 4.13 カテゴリ:[ソース]のダイアログボックス

(2) カテゴリ:[オブジェクト]

表 4.12 カテゴリ:[オブジェクト]の項目名とアセンブラオプションの対応表

ダイアログボックス	オプション
デバッグ情報」出力： **デフォルト** **デッバグ情報出力あり** **デバッグ情報出力なし**	 - Debug NODebug
プロセッサ展開結果出力	EXPand [= <**出力ファイル名**>]
リテラルテーブル展開結果出力： .POOL directive BRA, BRAF JMP RTS, RTE	LITERAL = <point>[,…] <point> : Pool <point> : Branch <point> : Jump <point> : Return
未解決シンボルサイズ指定：	DIspsize={ 4 | 12 }
オブジェクト出力ディレクトリ：	Object [= <**出力ファイル名**>] / NOObject

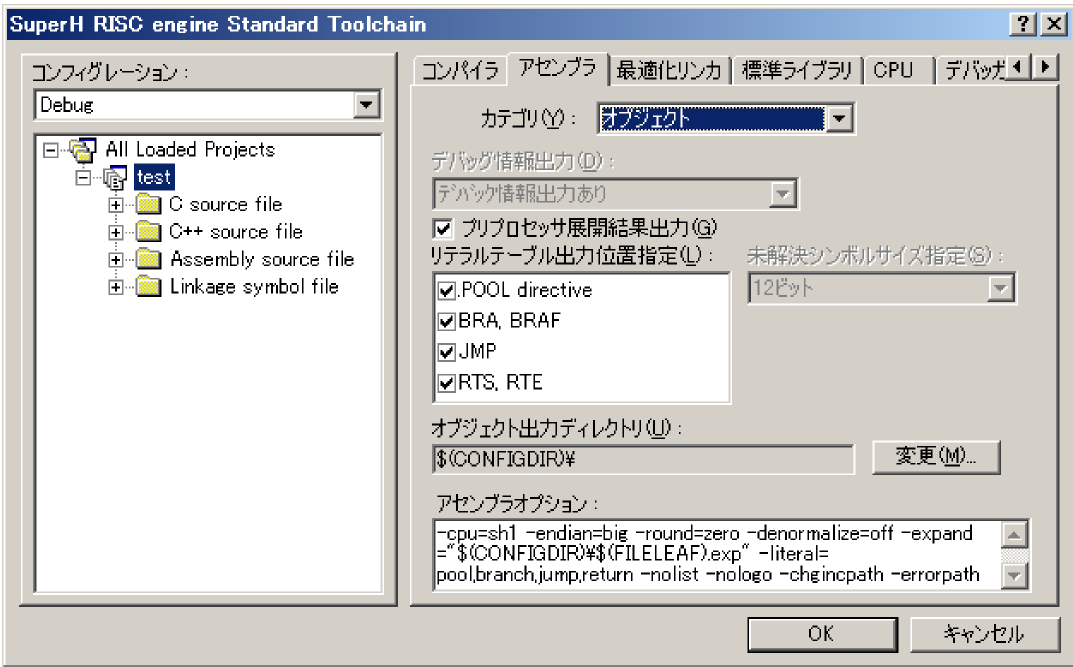


図 4.14 カテゴリ:[オブジェクト]のダイアログボックス

(3) **カテゴリ:[リスト]**

表 4.13 **カテゴリ:[リスト]の項目名とアセンブラオプションの対応表**

ダイアログボックス	オプション
アセンブルリスト出力	LISt [= <**出力ファイル名**>] / NOLISt
ソースプログラム：	
デフォルト	-
表示	SOurce
非表示	NOSOurce
クロスリファレンス：	
デフォルト	-
表示	CRoss_reference
非表示	NOCRoss_reference
セクション：	
デフォルト	-
表示	SEction
非表示	NOSEction
ソースプロプログラムリスト部分出力：	
Contents	
Default / Shown / Not shown	- / SHow [= <item>[,…]] / NOSHow [= <item>[,…]]
Status	
条件つき不成立	<item> : CONditionals
定義	<item> : Definitions
コール	<item> : CAlls
展開	<item> : Expansions
オブジェクトコード	<item> : CODe
タブサイズ	<item> : TAB = { 4 / 8 }

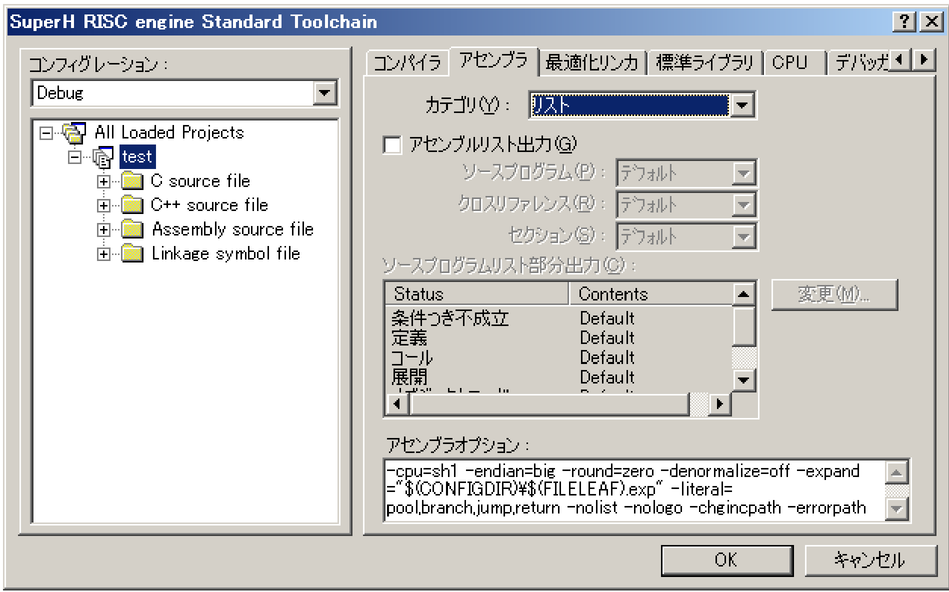


図 4.15 **カテゴリ:[リスト]のダイアログボックス**

RENESAS

(4) **カテゴリ:[その他]**

表 4.14 カテゴリ:[その他]の項目名とアセンブラオプションの対応表

ダイアログボックス	オプション
その他のオプション：	
リテラルプール自動生成機能をサイズ選択モードに指定	AUTO_literal
未定義外部参照シンボル情報の出力抑止	Exclude / NOExclude
特権モード命令チェックを指定	CHKMd
LDTLB命令チェックを指定	CHKTlb
キャッシュ関連命令チェックを指定	CHKCache
DSP関連命令チェックを指定	CHKDsp
FPU関連命令チェックを指定	CHKFpu
FDATAの8byte ALIGNチェックを指定	CHKAlign8

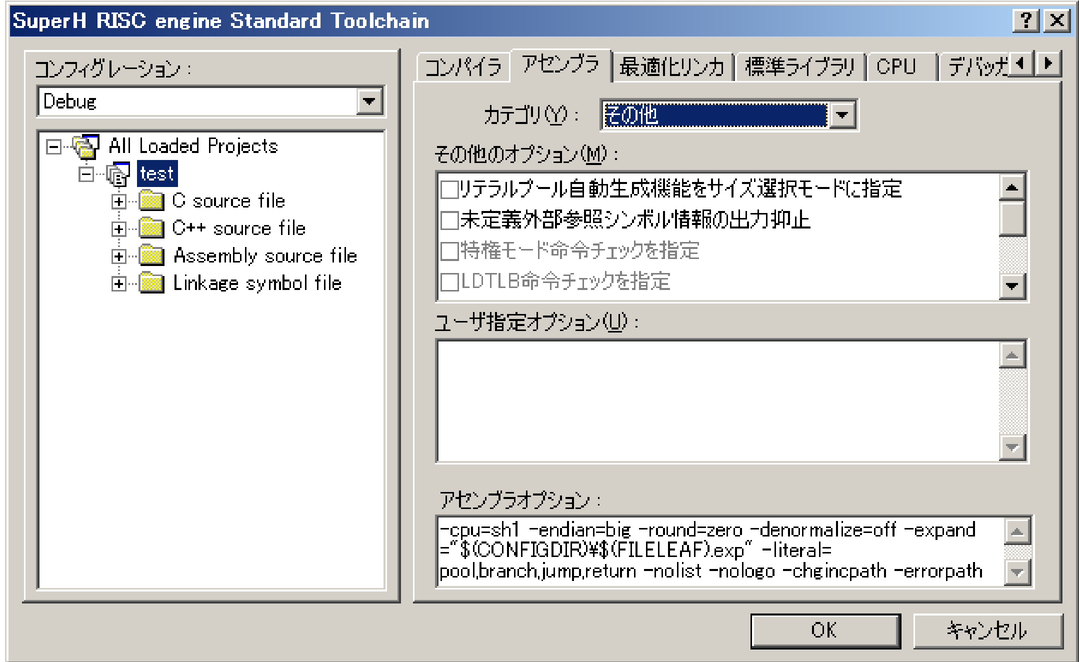


図 4.16 カテゴリ:[その他]のダイアログボックス

4.1.3 最適化リンケージエディタのオプション

SuperH RISC engine Standard Toolchain ダイアログボックスから最適化リンカタブを選択します。

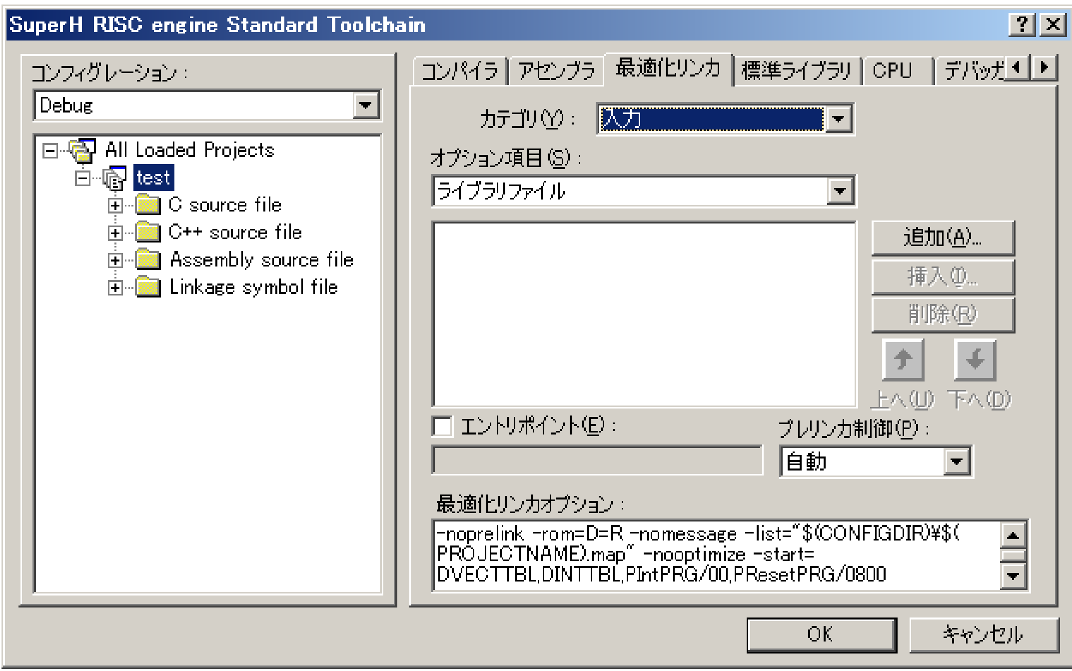


図 4.17　最適化リンカタブダイアログボックス

(1) **カテゴリ:[入力]**

表 4.15 カテゴリ:[入力]の項目名と最適化リンケージエディタオプションの対応表

ダイアログボックス	オプション
オプション項目： **ライブラリファイル** **リロケータブルファイル /** **オブジェクトファイル***1 **バイナリファイル** **シンボル定義**	 LIBrary = <ファイル名>[,...] Input = <sub> [{,|△}...] <sub> : <ファイル名>[(<モジュール名>[,...])] Binary = <sub>[,...] <sub> : <ファイル名> (<セクション名> [:<境界調整数>] [,<シンボル名>]) DEFine = <sub>[,...] <sub> : <シンボル名> = { <シンボル名> |<数値> }
エントリポイント：	ENTry = { <シンボル名> | <アドレス> }
プレリンカ制御： **自動** **不使用** **使用**	 NOPRElink NOPRElink -
*1 **プロジェクトに登録されているファイルについては明示的に追加する必要はなくコンパイル／アセンブルしないオブジェクトなどをリンクする場合に指定する。**	

図 4.18 カテゴリ:[入力]のダイアログボックス

(2) カテゴリ:[出力]

表 4.16 カテゴリ:[出力]の項目名と最適化リンケージエディタオプションの対応表

ダイアログボックス	オプション
出力形式：	
アブソリュート(ELF/DWARF)	FOrm = Absolute
アブソリュート(SYSROF)	FOrm = Absolute
リロケータブル	FOrm = Relocate
システムライブラリ	FOrm = Library = S
ユーザライブラリ	FOrm = Library = U
Hex(ELF/DWAR アブソリュート付き)	FOrm = Hexadecimal
S タイプ(ELF/DWAR アブソリュート付き)	FOrm = Stype
バイナリ(ELF/DWAR アブソリュート付き)	FOrm = Binary
レコードサイズ統一：	REcord = { H16 | H20 | H32 | S1 | S2 | S3 }
デバッグ情報：	
なし	NODEBug
出力(出力ファイル内)	DEBug
デバッグ情報ファイル(*.dbg)出力	SDebug
オプション項目：	
出力ファイル	
ROM から RAM へマップするセクション	ROm = <sub>[,…] <sub> : <ROM セクション名> =<RAM セクション名>
出力ファイルの分割	OUtput = <sub>[,…] <sub> : <ファイル名>[=<出力範囲>] <出力範囲> : { <先頭アドレス> - <終了アドレス> | <セクション名>[: …] }
空エリア出力指定	SPace = [<数値>]
メッセージ出力指定	NOMessage [= <sub>[,…]] / Message <sub> : <エラー番号> [- <エラー番号>]
外部シンボル割り付け情報ファイル出力	MAp [= <ファイル名>]

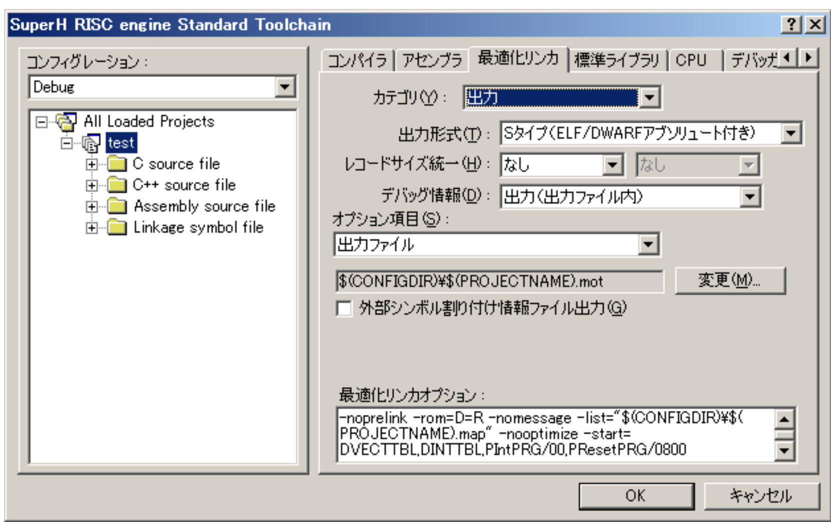


図 4.19 カテゴリ:[出力]のダイアログボックス

(3) カテゴリ:[リスト]

表 4.17 カテゴリ:[リスト]の項目名と最適化リンケージエディタオプションの対応表

ダイアログボックス	オプション
リンケージリスト出力	LISt [= <ファイル名>] / -
リスト内容： シンボル情報 参照回数 セクション情報 クロスリファレンス情報	SHow [= <sub>[,...]] <sub> : SYmbol <sub> : Reference <sub> : SEction <sub> : Xreference

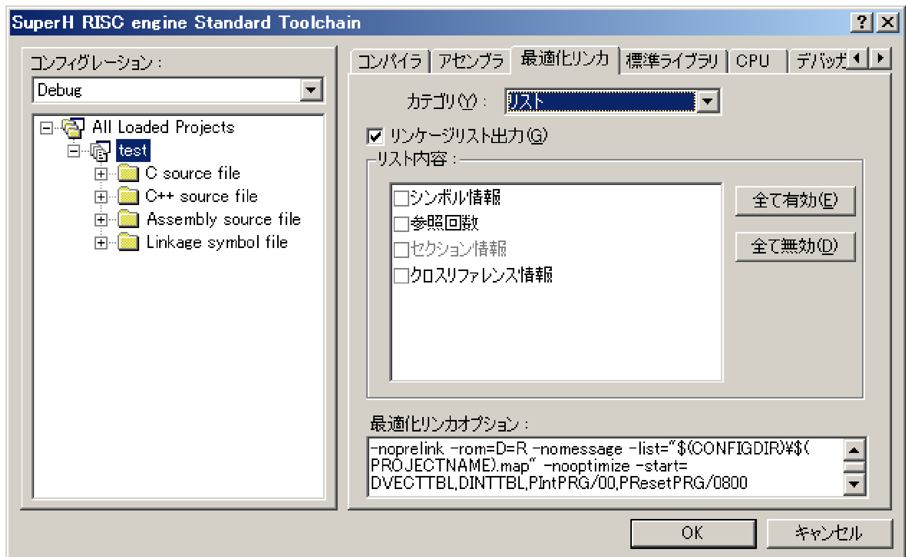


図 4.20 カテゴリ:[リスト]のダイアログボックス

(4) **カテゴリ:[最適化]**

表 4.18 カテゴリ:[最適化]の項目名と最適化リンケージエディタオプションの対応表

ダイアログボックス	オプション
最適化方法：	
最適化設定	
設定：	OPtimize [= <sub>[,...]]
全項目	<sub> : STring_unify,SYmbol_delete, Variable_access,Register, SAMe_code,SHort_format, Function_call,Branch,
スピード重視	<sub> : SPeed
安全な最適化	<sub> : SAFe
カスタム	**下記を任意指定**
定数／文字列の情報	<sub> : STring_unify
未参照シンボルの削除	<sub> : SYmbol_delete
レジスタ退避・回復の最適化	<sub> : Register
共通コードの統合	<sub> : SAMe_code
分岐命令の最適化	<sub> : Branch
最適化なし	NOOPtimize
統合サイズ：	SAMESize = <**サイズ**> (**省略時：**sames=1e)
プロファイル情報：	PROfile = <**ファイル名**>
キャッシュサイズ：	CAchesize = Size = <**サイズ**>, Align = <**ラインサイズ**> (**省略時：**ca=s=8,a=20)
最適化方法：	
最適化部分抑止	
未参照シンボル削除抑止シンボル	SYmbol_forbid = <**シンボル名**>[,...]
共通コード統合抑止シンボル	SAMECode_forbid = <**関数名**>[,...]
最適化抑止アドレス範囲	Absolute_forbid = <**アドレス**> [+ <**サイズ**>] [,...]

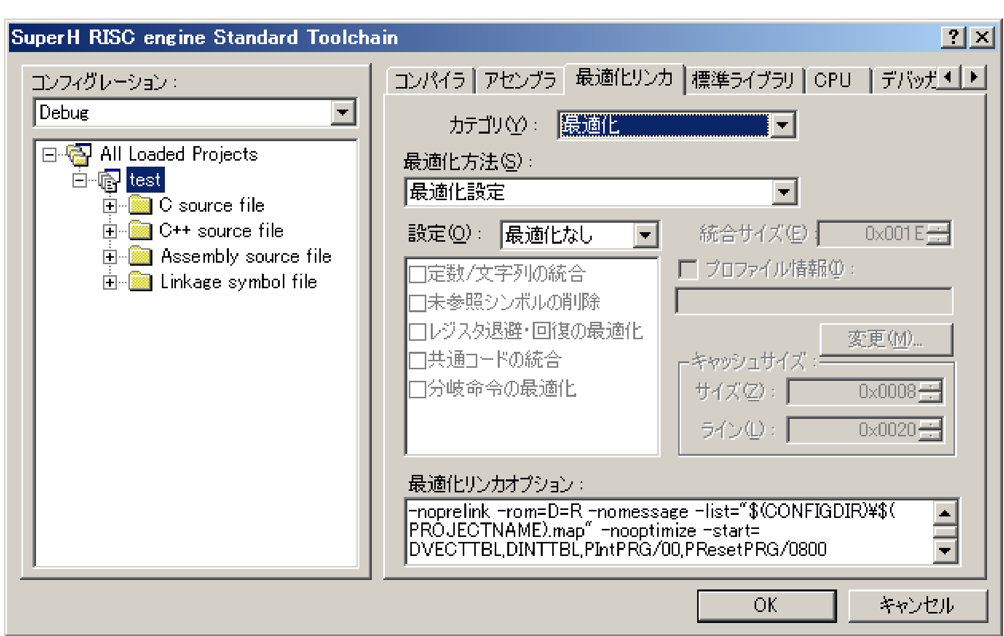


図 4.21 カテゴリ:[最適化]のダイアログボックス

RENESAS

(5) カテゴリ:[セクション]

表 4.19 カテゴリ:[セクション]の項目名と最適化リンケージエディタオプションの対応表

ダイアログボックス	オプション
設定項目：	
セクション	STAR t = <sub>[,...] <sub> : <**セクション名**> [{: | ,} <**セクション名**>[,...]] [/<**アドレス**>]
シンボルアドレスファイル	FSymbol = <**セクション名**>[,...]

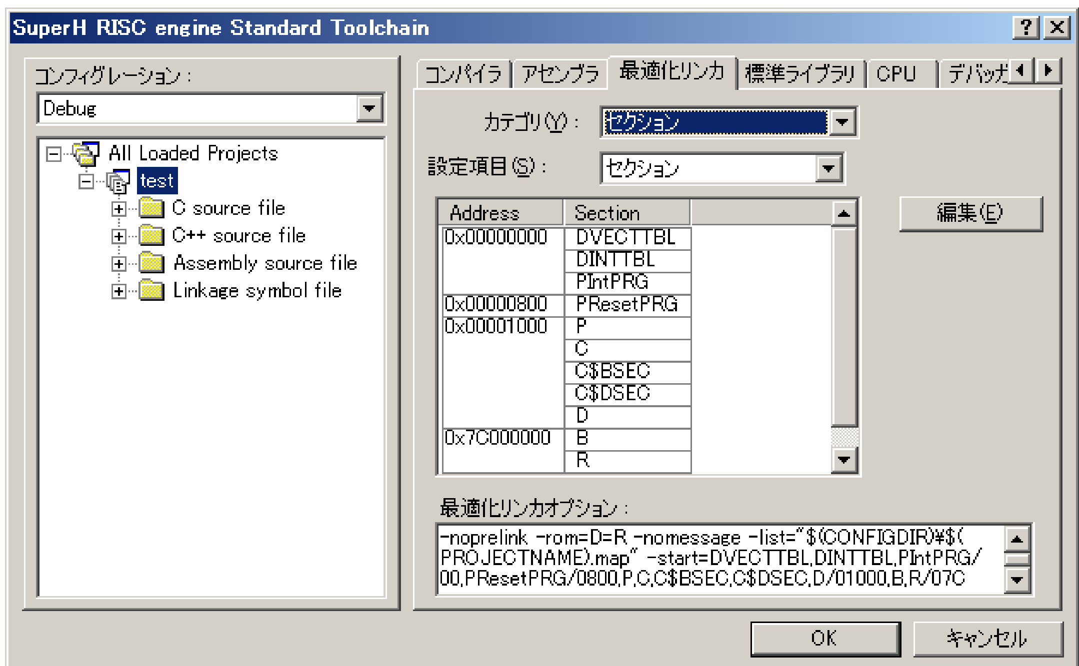


図 4.22 カテゴリ:[セクション]のダイアログボックス

[編集]ボタンでセクションの割り付けを新しく指定することができます。
[追加]ボタンでセクション名、アドレスを追加することができます。
[変更]ボタンで既に指定されたセクション名、アドレスを編集することができます。
[複数割付]ボタンで複数のセクションを同一アドレスに割り付けることができます。
[削除]ボタンですでに指定されたセクションを削除することができます。
セクションの並び順は[上]／[下]ボタンで変更することができます。

前ページのダイアログボックスの内容をリンケージエディタのサブコマンドファイルに記述した場合

```
    START DVECTTBL,DINTTBL,PIntPRG
    START PResetPRG/800
    START P,C,C$BSEC,C$DSEC,D/1000
    START RAM_sct1:RAM_sct2/F00000
                START B,R/7F000000
                START Stack/7FFFFBF0
```

図4.22上で見えている部分

RAM_sct1 と RAM_sct2 は同一セクションに割り付けられます。

【注】リンケージエディタ用サブコマンドファイルの詳しい作成方法は、「SuperH RISC engine C/C++コンパイラ、アセンブラ、最適化リンケージエディタユーザーズマニュアル」を参照してください。

(6) **カテゴリ:[ベリファイ]**

表 4.20 カテゴリ:[ベリファイ]の項目名と最適化リンケージエディタオプションの対応表

ダイアログボックス	オプション
CPU 情報：	-
チェックなし	
チェック	CPu = {<cpu情報ファイル名> | <メモリ種別> = <アドレス範囲>[,…]} <メモリ種別> = {ROm | RAm | XROm | XRAm | YROm | YRAm} <アドレス範囲> : <先頭アドレス> - <終了アドレス>
CPU 情報ファイルを指定してチェック	CPu = <cpu情報ファイル名>

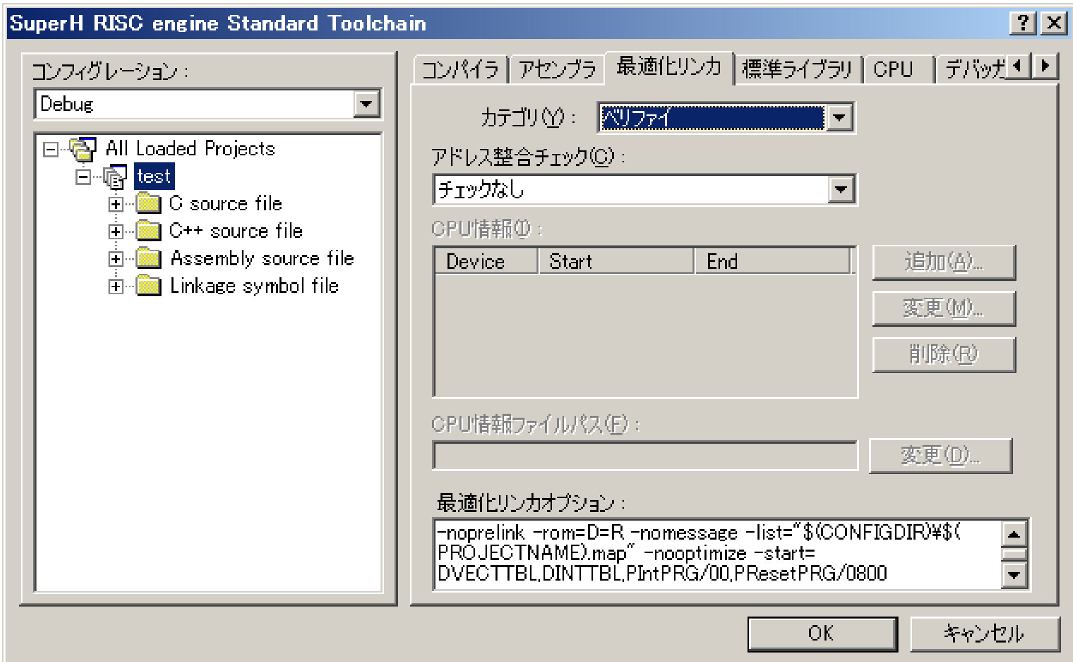


図 4.23 カテゴリ:[ベリファイ]のダイアログボックス

(7) **カテゴリ**: [**その他**]

表 4.21 **カテゴリ**:[**その他**]**の項目名と最適化リンケージエディタオプションの対応表**

ダイアログボックス	オプション
その他のオプション：	
S9 **レコードを終端に出力**	S9
スタック上右方ファイル(sni)**出力**	STACk
デバッグ情報圧縮	COmpress / NOCOmpress
入力ファイルロード時のメモリ使用量削減	MEMory = [Hight | Low]
ユーザ指定オプション：	
アブソリュート／リロケータブル／ライブラリ	
HEX/S **タイプ／バイナリ**	

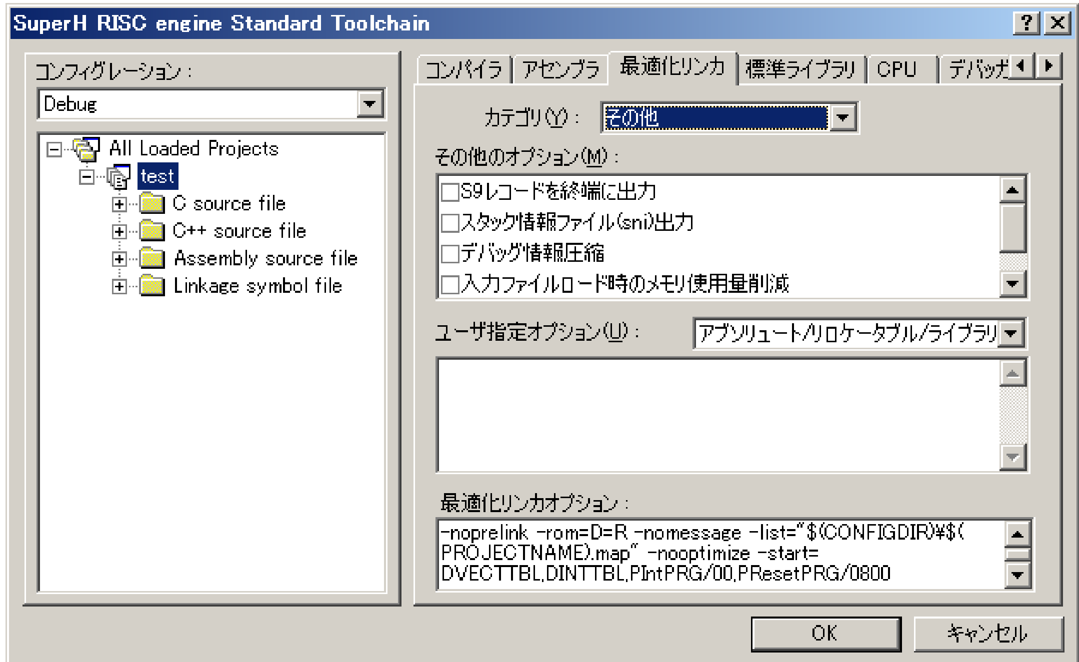


図 4.24 **カテゴリ**:[**その他**]**のダイアログボックス**

RENESAS

(8)　カテゴリ:[サブコマンドファイル]

表 4.22　カテゴリ:[サブコマンドファイル]の項目名と最適化リンケージエディタオプションの対応表

ダイアログボックス	オプション
サブコマンドファイルを指定	SUbcommand = <ファイル名>

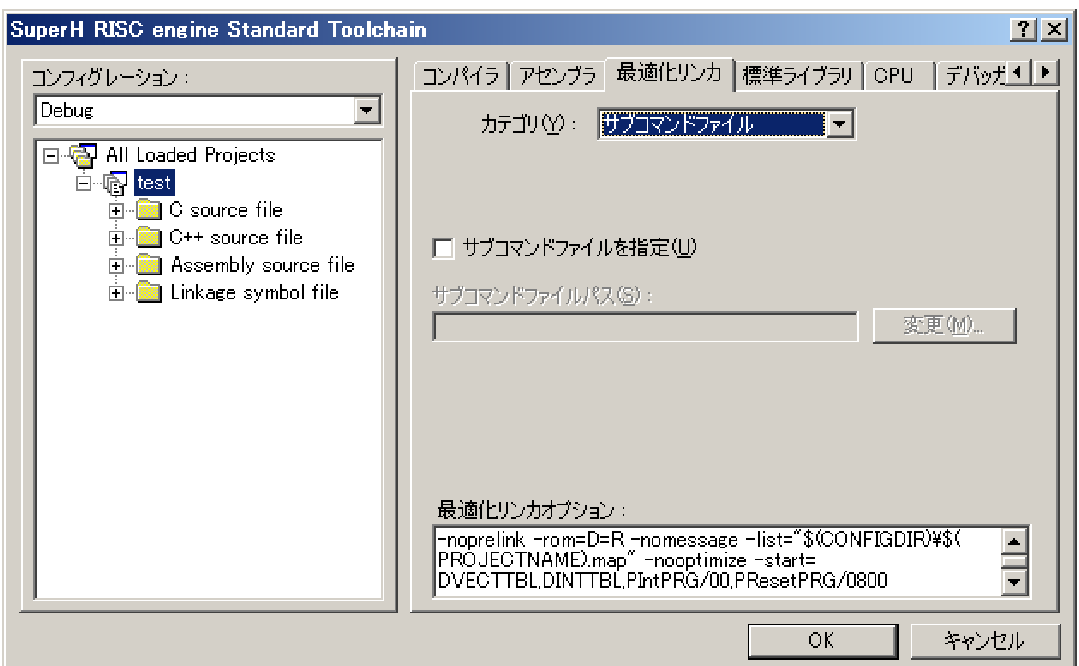


図 4.25　カテゴリ:[サブコマンドファイル]のダイアログボックス

4.1.4 標準ライブラリ構築ツールのオプション

SuperH RISC engine Standard Toolchain ダイアログボックスから標準ライブラリタブを選択します。

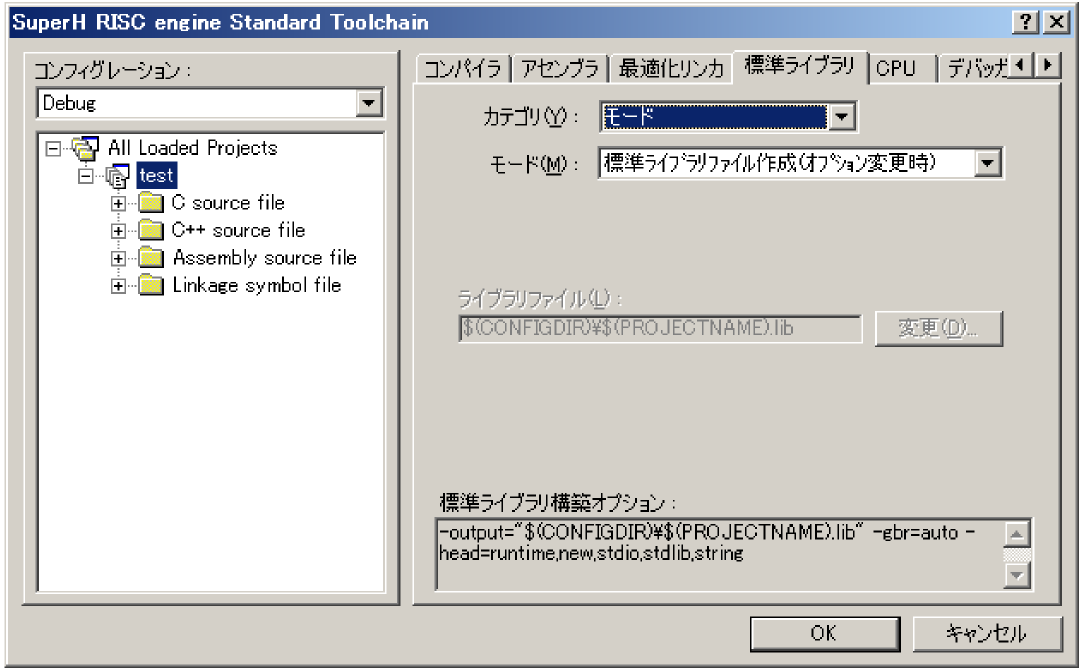


図 4.26 標準ライブラリタブのダイアログボックス

(1)　**カテゴリ:[モード]**

表 4.23　カテゴリ:[モード]の項目名と機能の対応表

ダイアログボックス	機能
モード：	
標準ライブラリファイル作成(常に作成)	最新の標準ライブラリ作成
標準ライブラリファイル作成(オプション変更時)	オプション変更時、最新の標準ライブラリ作成
既存標準ライブラリファイル指定	既存の標準ライブラリをリンク
標準ライブラリファイル指定なし	標準ライブラリをリンクしない

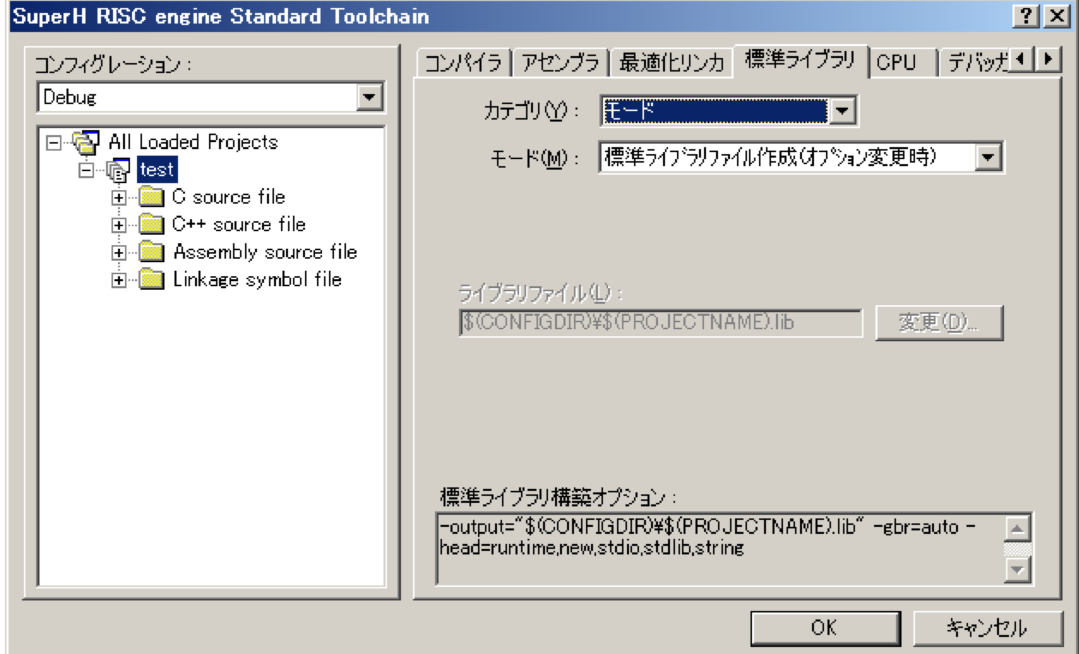


図 4.27　カテゴリ:[モード]ダイアログボックス

(2)　**カテゴリ:[標準ライブラリ]**

表 4.24　カテゴリ:[標準ライブラリ]の項目名と標準ライブラリ構築ツールオプションの対応表

ダイアログボックス	オプション
カテゴリ：	Head = <sub>[,….]
runtime：実行時ルーチン	<sub> : RUNTIME
new：メモリ操作用ライブラリ	<sub> : NEW
ctype.h：文字操作用ライブラリ	<sub> : CTYPE
math.h：数値計算用ライブラリ	<sub> : MATH
mathf.h：数値計算用ライブラリ (各関数は float 型の引数を受け取り、float 型の値を返す)	<sub> : MATHF
stdarg.h：可変個の実引数アクセス用ライブラリ	<sub> : STDARG
stdio.h：入出力用ライブラリ	<sub> : STDIO
stdlib.h：標準処理用ライブラリ	<sub> : STDLIB
string.h：文字列操作用ライブラリ	<sub> : STRING
ios(EC++)：ストリーム入出用クラスライブラリ	<sub> : IOS
complex(EC++)：複素数計算用クラスライブラリ	<sub> : COMPREX
string(EC++)：文字列操作用ライブラリ(EC++用)	<sub> : CPPSTRING

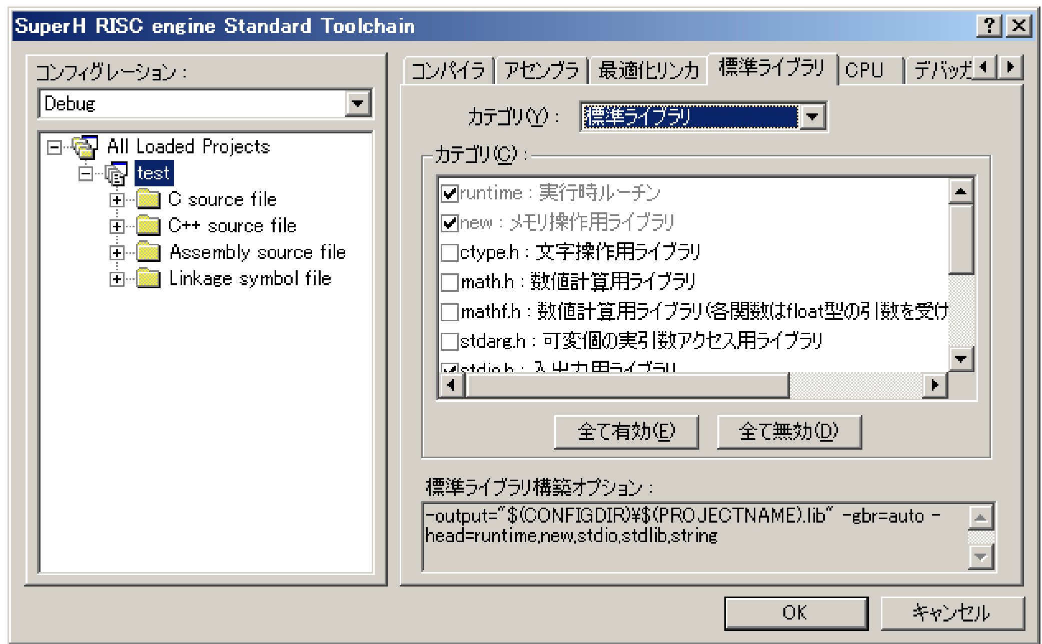


図 4.28　カテゴリ:[標準ライブラリ]のダイアログボックス

(3)　**カテゴリ:[オブジェクト]**

表 4.25　**カテゴリ:[オブジェクト]の項目名とオプションの対応表**

ダイアログボックス	オプション
簡易入出力関数	NOFLoat
リエントラントライブラリを作成	REent

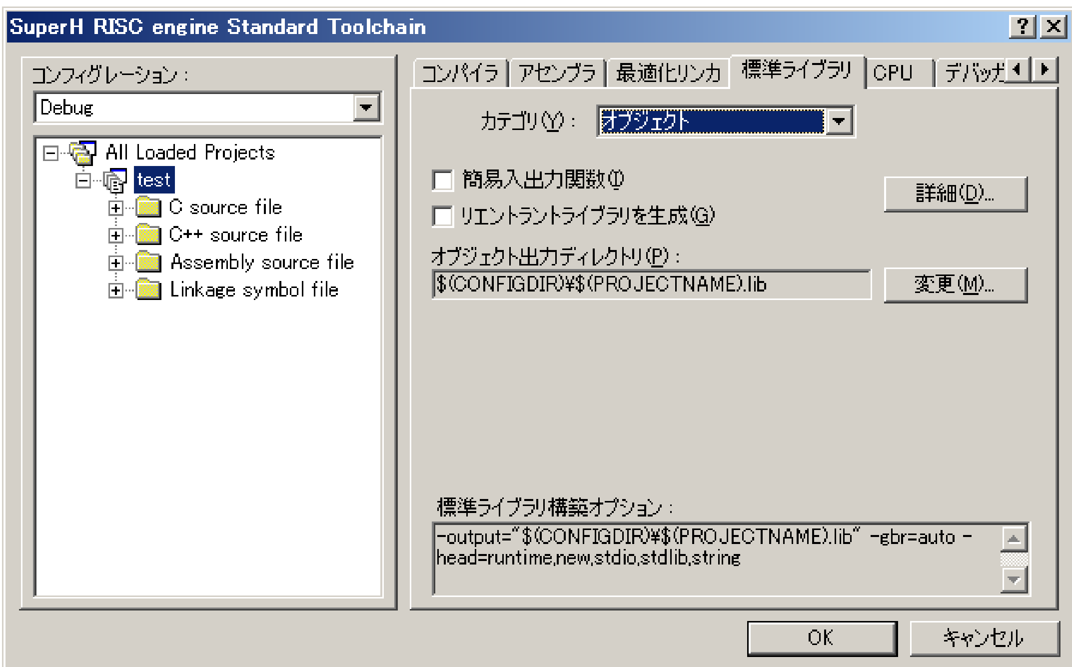


図 4.29　**カテゴリ:[オブジェクト]のダイアログボックス**

[詳細…]をクリックすると、”Object details”ダイアログボックスが表示されます。

(a) **詳細タブ**

表 4.26 項目名とオプションの対応表

ダイアログボックス	オプション
セクション： **プログラム領域**(P) **定数領域**(C) **初期化データ**(D) **未初期化データ領域**(B)	SEction = <sub>[,…] <sub> : Program = <**セクション名**> <sub> : Const = <**セクション名**> <sub> : Data = <**セクション名**> <sub> : Bss = <**セクション名**> **省略時：**(p=P, c=C, d=D, b=B)
文字列データ格納：	STring = { Const | Data }
除算方式選択：	DIvision = Cpu [= { Inline | Runtime }]
浮動小数点レジスタ退避・回避抑止	IFUnc
ラベルの 16 / 32 **バイト整合：**	ALIGN16 ALIGN32 NOALign
NOFLoat, REent：**標準ライブラリ構築ツールオプション** **それ以外：コンパイラオプション**	

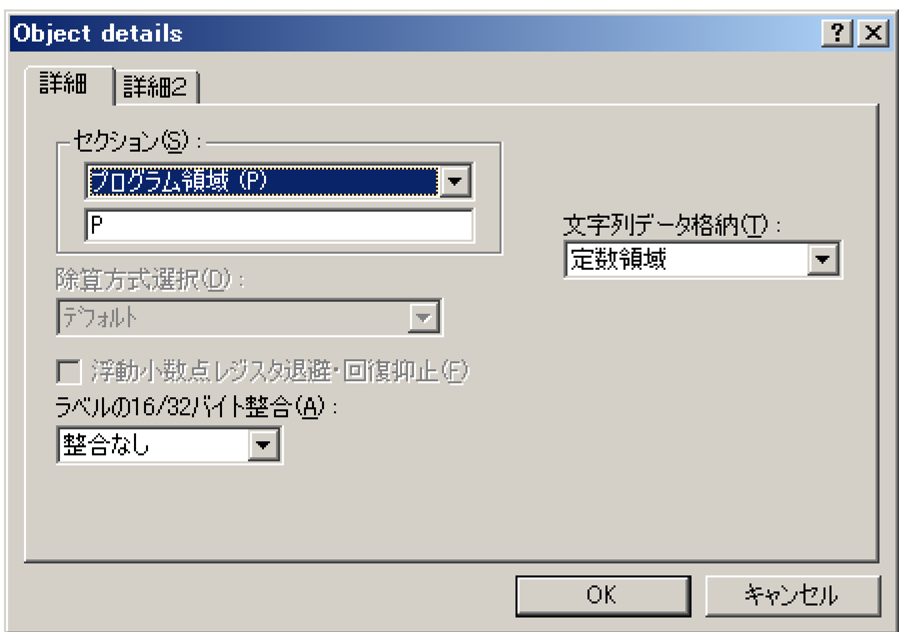


図 4.30 詳細タブのダイアログボックス

(b)　**詳細 2 タブ**

表 4.27　項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
アドレス宣言：	<ABS> = <sub>[,…] <ABS>: { ABs16 | ABS20 | ABS28 | ABS32} <sub>: { Program | Const | Data | Bss | Run | All }
TBR **指定：**	TBR[=<**セクション名**>]

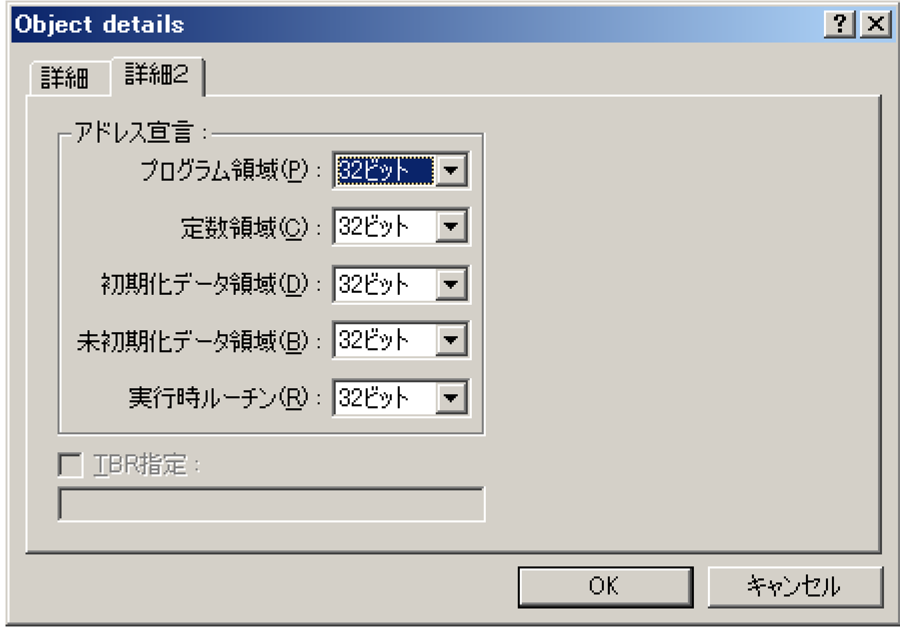


図 4.31　詳細 2 タブのダイアログボックス

RENESAS

(4) **カテゴリ:[最適化]**

表 4.28 カテゴリ:[最適化]の項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
最適化	OPtimize = 1 / OPtimize = 0
最適化方法： 　**スピード優先** 　**サイズ優先** 　**サイズ＆優先**	 SPeed SIze NOSPeed
モジュール間最適化	Goptimize
GBR **相対アクセス最適化：**	GBr = { Auto | User }
非境界調整データ転送：	Unaligned = { Inline | Runtime }
自動インライン展開ファイル：	INLine[=<**数値**>] / NOINLine
switch **文展開：**	CAse = { Ifthen | Table }
シフト演算展開：	SHIft = { Inline | Runtime }
転送コード展開：	BLOckcopy = { Inline | Runtime }

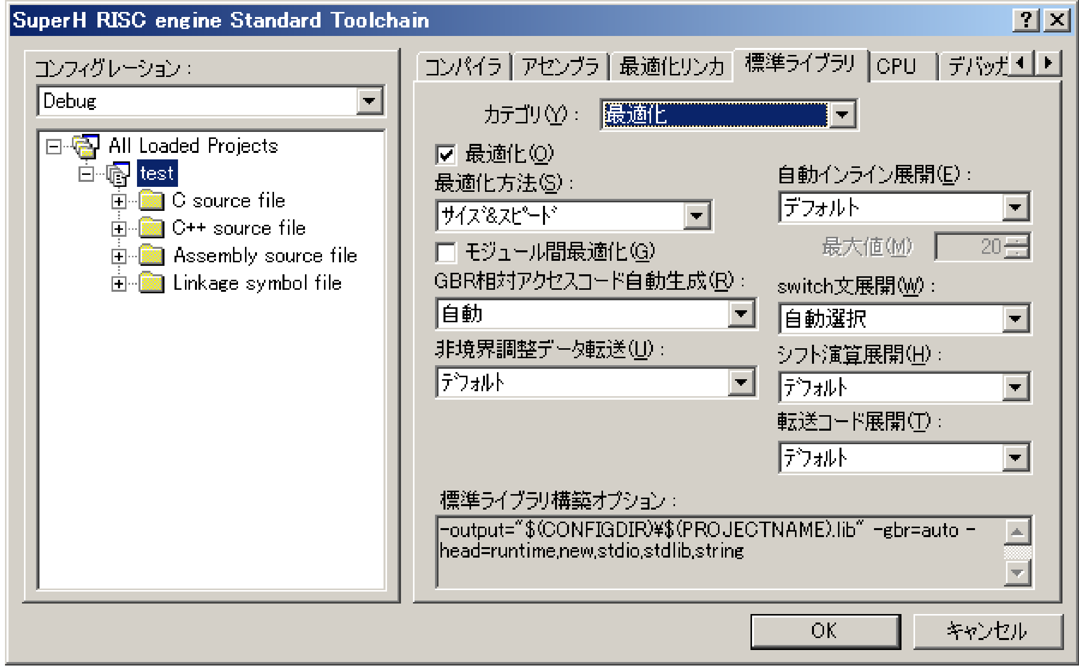


図 4.32 カテゴリ:[最適化]のダイアログボックス

(5) **カテゴリ:[その他]**

表 4.29 カテゴリ:[その他]の項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
その他のオプション：	
EC++ 言語に基づいたチェック	ECpp
DSP-C 言語に基いたチェック	DSpc
SSR / SPC 退避・回復	SAve_cont_reg = { 0 / 1 }
返却値の拡張を行う	RTnext / NORTnext
ループ展開	LOop / NOLOop
浮動小数点定数除算の乗算化	APproxdiv
SH7055 不具合回避	PAtch = 7055
FPSCR レジスタの切り替え	FPScr = Safe / FPScr = Aggressive
ループ判定式の最適化抑止	Volatile_loop
列挙型サイズの自動選択	AUto_enum
浮動小数点→固定小数点変換	FIXED_Const
1.0→__fixed 型最大値変換	FIXED_Max
__fixed 型乗算結果の型変換省略	FIXED_Noround
DSP 拡張リピートループ	REPeat
Register 指定変数の優先レジスタ割り付け	ENAble_register
ANSI 準拠対応拡張	STRIct__ansi
整数除算の浮動小数点除算置き換え	FDIv

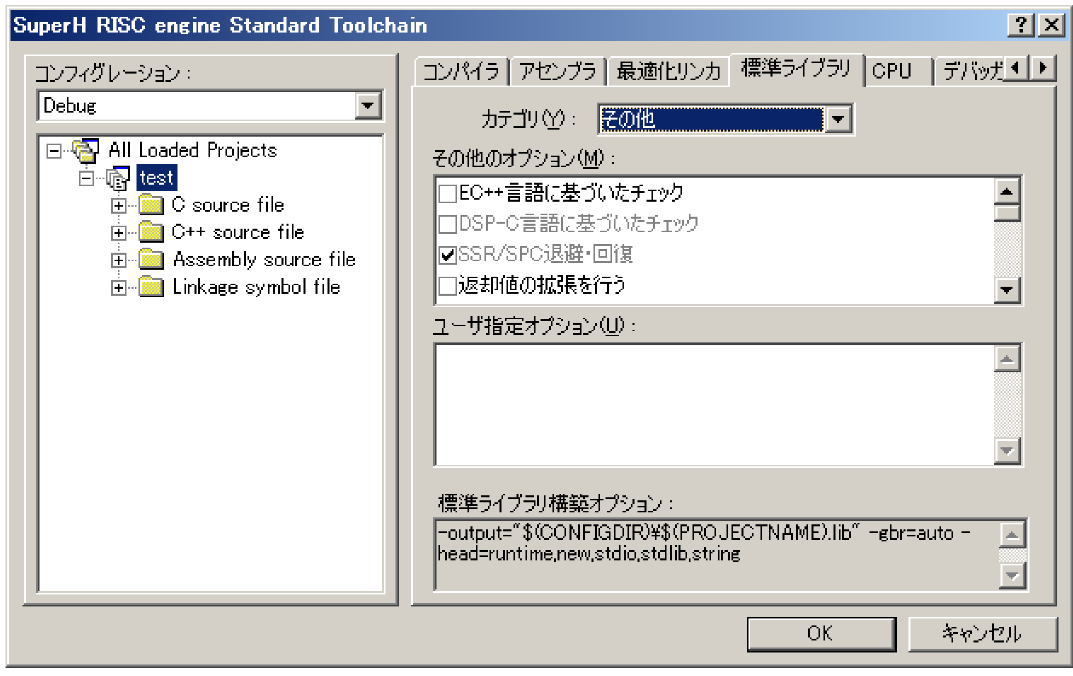


図 4.33 カテゴリ:[その他]のダイアログボックス

4.1.5 CPU オプション

SuperH RISC engine Standard Toolchain ダイアログボックスから CPU タブを選択します。

表 4.30 [CPU]タブの項目名とコンパイラオプションの対応表

ダイアログボックス	オプション
CPU：	CPu = { SH1 | SH2 | SH2E | SH2A | SH2AFPU | SH2DSP | SH3 | SH3DSP | SH4 | SH4A | SH4ALDSP }
除算方式選択：	DIvision = { Cpu = [= { Inline | Runtion}] | Peripheral | Nomask }
Endian 選択：	ENdian = { Big | Little }
浮動小数点演算モード：	Fpu = { Single | Double }
丸め方式：	Round = { Zero | Nearest }
非正規化数を非正規化数として扱う	DENormalize = ON / DENormalize = OFF
ポジションインディペンデントコード生成	Pic = 1 / Pic = 0
double→float 変換	DOuble = Float
ビットフィールドメンバを下位 bit から格納	Bit_order = { Left | Right }
構造体メンバの境界調整数を 1 とする	PACK = 1 / PACK = 4
C++の try、throw、catch を有効にする	EXception / NOEXception
C++の dynamic_cast、typeid を有効にする	RTTI = ON / RTTI = OFF

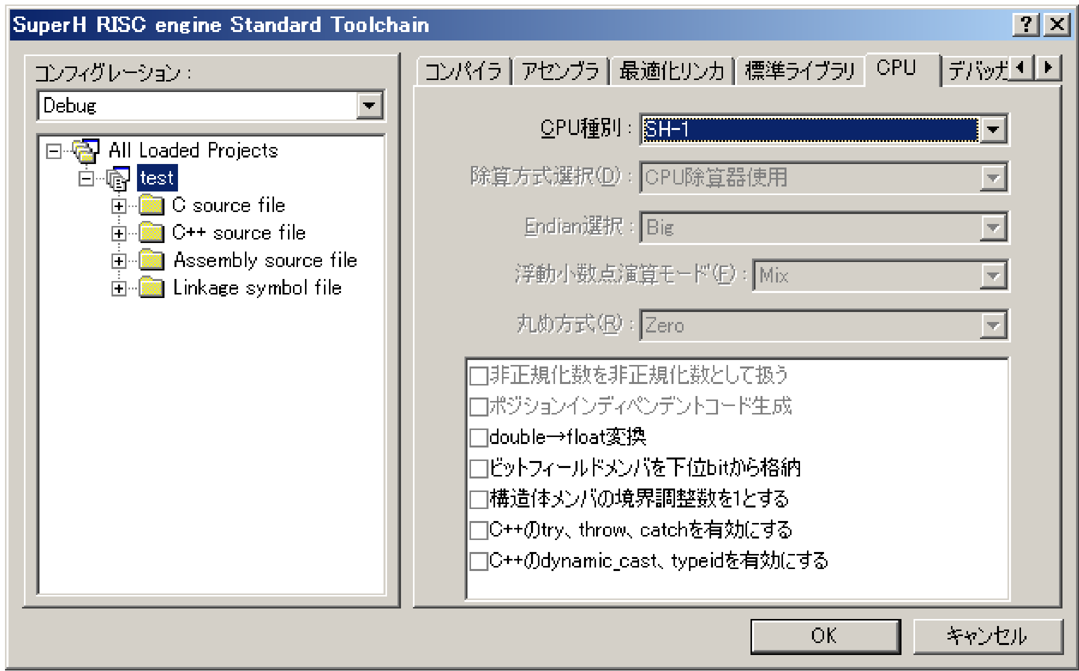


図 4.34 [CPU]タブのダイアログボックス

4.2 統合化環境からのコンパイラバージョンの指定

統合化環境でのコンパイラバージョンの指定方法について説明します。バージョンの指定は統合化環境をアップグレードすることで可能になります。

旧バージョン（例 HEW1.1,SH5.1B など）で作成したワークスペースを新バージョン（例 HEW3.01,SH9.0）でオープン時、以下のダイアログボックスを表示します。

(1) アップグレード対象プロジェクトの確認

アップグレード対象のプロジェクト名を確認します。

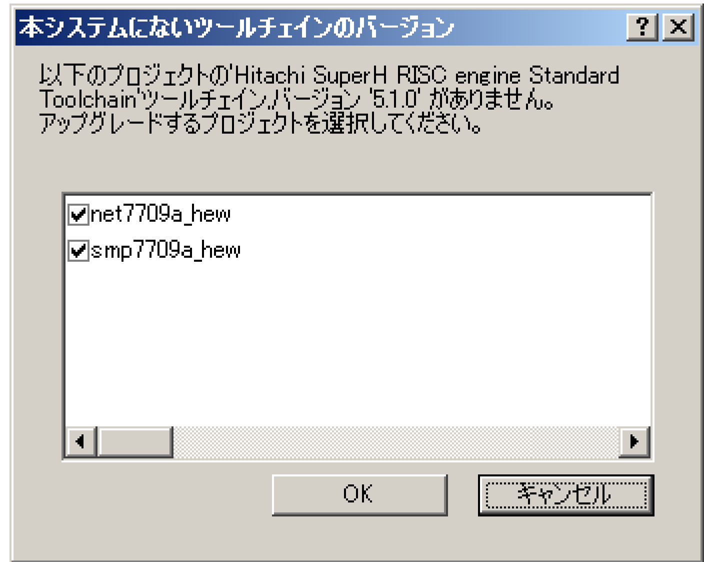


図 4.35 High-performance Embedded Workshop

RENESAS

(2) バージョンの指定

アップグレードできるバージョンを選択します。

図 4.36 ツールチェインのバージョンの変更ダイアログボックス

(3) 確認メッセージ

C/C++コンパイラ Ver7.1 以降、出力するオブジェクトのファイル形式が ELF/DWARF のみをサポートしています。アップグレード時に、ELF/DWARF フォーマットに変換するため、現在使用中のデバック環境が ELF/DWARF フォーマットをサポートしていない場合は、アップグレード後に、デバック環境がサポートするフォーマット形式に変換してください。

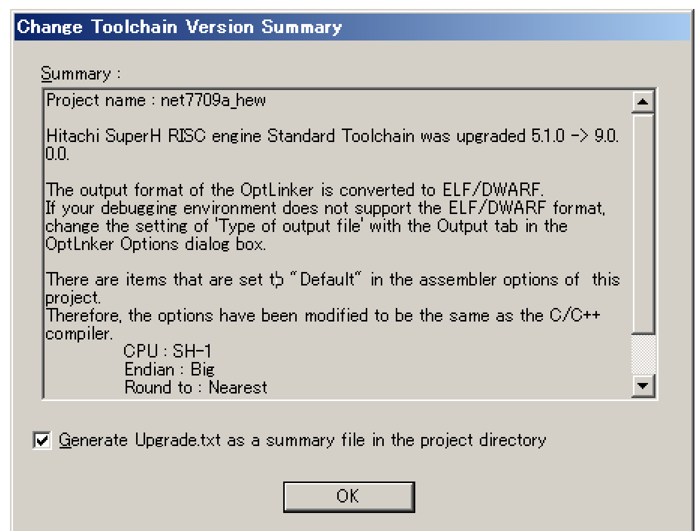


図 4.37 確認メッセージダイアログ

RENESAS

5. 効率の良いプログラミング技法

SuperH RISC engine C/C++コンパイラは最適化を行っていますが、プログラミングの工夫により一層の性能向上が可能です。

本章では、効果的なプログラム作成のために、ユーザに試みて頂きたい手法について記述します。

プログラムの評価基準には、実行速度が速いこととサイズが小さいことの 2 種類があります。

SuperH RISC engine C/C++コンパイラでは、最適化を実行速度優先で行うことができます。このときには、コンパイルオプションに”-speed”を指定してください。

効果的なプログラムを作成するための原則を以下に示します。

（1） 実行速度向上の原則
実行頻度の高い文、複雑な文で実行速度は決まるので、これらの処理を把握して、重点的に改良してください。

（2） サイズ縮小の原則
プログラムサイズ縮小のためには、類似処理の共通化、複雑な関数の見直しを行ってください。

コンパイラの最適化のため、実行速度が机上で検討したときとは異なる結果になることがあります。さまざまな手法を駆使し、実際にコンパイラで実行して確認しながら性能追求を進めてください。

本章のアセンブリ言語展開コードは

shc△〈C 言語ファイル〉△-code＝asmcode△-cpu=sh2

のコマンドラインで取得しています。ただし、-cpu オプションにより、アセンブリ言語展開コードが SH-1、SH-2、SH-2E、SH-3、および SH-4 で異なる場合があります。今後、コンパイラの改善などにより、アセンブリ言語展開コードは変わる可能性があります。

本章のコードサイズ、実行速度は、SH-1、SH-2、SH-2A、SH2A-FPU、SH-2E、SH2-DSP（SH7065）、SH-3、SH3-DSP 、SH-4、SH-4A、および SH4AL-DSP で測定しています。コンパイル時の CPU オプションを**表** 5.1 に示します。

表 5.1　CPU **オプション一覧**

項番	CPU **種別**	CPU **オプション**
1	SH-1	-cpu=sh1
2	SH-2	-cpu=sh2
3	SH-2A	-cpu=sh2a
4	SH2A-FPU	-cpu=sh2afpu
5	SH-2E	-cpu=sh2e
6	SH2-DSP(SH7065)	-cpu=sh2
7	SH-3	-cpu=sh3
8	SH3-DSP	-cpu=sh3
9	SH-4	-cpu=sh4△-fpu=single
10	SH-4A	-cpu=sh4△-fpu=single
11	SH4AL-DSP	-cpu=sh4aldsp

SH-2A、SH2A-FPU、SH3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP の測定に関しては、一部を除いてキャッシュミスを考慮していません。外部メモリへのアクセスサイクル数は１として測定しています。

効率の良いプログラミング技法の一覧を**表5.2**に示します。

表5.2 効率の良いプログラミング技法一覧

項番	項目	ROM 効率	RAM 効率	実行速度	参照
1	局所変数（データサイズ）	○	-	○	5.1.1
2	大域変数（符号）	○	-	○	5.1.2
3	データサイズ（乗算）	○	-	○	5.1.3
4	データの構造化	○	-	○	5.1.4
5	データの整合	-	○	-	5.1.5
6	初期値と const 型	-	○	-	5.1.6
7	局所変数と大域変数	○	-	○	5.1.7
8	ポインタ変数の活用	○	-	○	5.1.8
9	定数参照（1）	○	-	-	5.1.9
10	定数参照（2）	○	-	-	5.1.10
11	一定値になる変数（1）	-	-	-	5.1.11
12	一定値になる変数（2）	-	-	-	5.1.12
13	関数のモジュール化	○	-	○	5.2.1
14	ポインタ変数による関数呼び出し	○	-	○	5.2.2
15	関数のインタフェース	-	○	○	5.2.3
16	テールリカージョン	○	-	○	5.2.4
17	FSQRT,FABS 命令活用	○	-	○	5.2.5
18	ループ内不変式の移動	-	-	○	5.3.1
19	ループ回数の削減	×	-	○	5.3.2
20	乗算／除算の使用	-	-	-	5.3.3
21	公式の適用	-	-	○	5.3.4
22	テーブルの活用	○	-	○	5.3.5
23	条件式	○	-	○	5.3.6
24	ロードストア削除	○	-	○	5.3.7
25	分岐	○	-	○	5.4
26	関数のインライン展開	×	-	○	5.5.1
27	アセンブラ埋め込みのインライン展開	-	-	○	5.5.2
28	グローバルベースレジスタ（GBR）のオフセット参照	○	-	○	5.6.1
29	グローバルベースレジスタ（GBR）の領域の使い分け	○	-	○	5.6.2
30	レジスタ退避／回復の制御	○	-	○	5.7
31	2 バイトアドレスの指定	○	-	-	5.8
32	プリフェッチ命令	-	-	○	5.9.1
33	タイリング	×	-	○	5.9.2
34	マトリックス演算	○	-	○	5.10
35	ソフトパイプ	-	-	○	5.11

【注】表中の○、×は以下の意味を示します。
○…性能向上に効果あり
×…性能低下の可能性あり

5.1 データ指定

データに関して考慮すべき事項を**表** 5.3 に示します。

表 5.3 **データ指定における注意事項**

項目	注意点	参照
データ型指定子、型修飾子	・データサイズを縮小しようとすると、プログラムサイズが増大する場合があります。データは用途を考えて型宣言してください。 ・符号あり／なしによりプログラムサイズが変わることがあるので、選択時に注意してください。 ・プログラム内で値が不変な初期化データの場合、const 演算子を付けておくと使用メモリ量の節約になります。	5.1.1～5.1.3 5.1.6
データの整合	・データ領域に無駄なエリアを生じないように割り付けてください。	5.1.5
構造体の定義／参照	・頻繁に参照／変更するデータは構造体にして、ポインタ変数を用いることによりプログラムサイズを縮小できる場合があります。 ・ビットフィールドを使用すると、データサイズを縮小できます。	5.1.4
局所変数と大域変数	・局所変数の方が効率がよいので、局所変数として使用できるものは大域変数として宣言しないで必ず局所変数として宣言してください。	5.1.7
ポインタ型の活用	・配列型を用いたプログラムは、ポインタ型を用いて書き直せないか検討してください。	5.1.8
内蔵 ROM/RAM の活用	・外部メモリに比べ内蔵メモリへのアクセスは速いので、共通変数は内蔵メモリへ格納するようにしてください。	-

RENESAS

5.1.1 局所変数（データサイズ）

■ポイント

局所変数のサイズは 4 バイトでとると、ROM 効率と実行速度を向上できる場合があります。

■説明

ルネサステクノロジ SuperH RISC engine ファミリの汎用レジスタは 4 バイトであるため、処理の基本は 4 バイトです。このため、1 バイト／2 バイトの局所変数を用いた演算があると、4 バイトに型変換するコードが付け加わります。1 バイト／2 バイトで十分な変数でも 4 バイトでとっておくとプログラムサイズが小さくなり、実行速度を向上できる場合があります。

■使用例

1 から 10 までの総和を求めます。

改善前ソースコード

```
int f(void)
{
   char a = 10;
   int  c = 0;

   for ( ; a > 0; a-- )
        c += a;

   return(c);
}
```

改善前アセンブリ展開コード

```
_f:
        MOV         #10,R4
        MOV         #0,R5
L217:
        EXTS.B      R4,R3
        ADD         R3,R5
        ADD         #-1,R4
        EXTS.B      R4,R2
        CMP/PL      R2
        BT          L217
        RTS
        MOV         R5,R0
```

改善後ソースコード

```
int f(void)
{
   long a = 10;
   int  c = 0;

   for ( ; a > 0; a-- )
        c += a;

   return(c);
}
```

改善後アセンブリ展開コード

```
_f:
        MOV         #10,R4
        MOV         #0,R5
L217:
        ADD         R4,R5
        ADD         #-1,R4
        CMP/PL      R4
        BT          L217
        RTS
        MOV         R5,R0
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	18	16	73	63
SH-2	18	16	73	63
SH-2A	16	14	62	52
SH2A-FPU	16	14	62	52
SH-2E	18	16	73	63
SH2-DSP(SH7065)	18	16	73	63
SH-3	18	16	73	63
SH3-DSP	18	16	73	63
SH-4	18	16	64	54
SH-4A	18	16	54	44
SH4AL-DSP	18	16	54	44

5.1.2 大域変数（符号）

■ポイント

式中に大域変数の型変換が含まれる場合、整数の型が signed でも unsigned でもよいときには signed で宣言すると、ROM 効率と実行速度を向上できます。

■説明

ルネサステクノロジ SuperH RISC engine ファミリでは、メモリから MOV 命令で 1 バイト／2 バイトのデータを転送するとき、unsigned のデータでは EXTU 命令を付加します。このため、unsigned 型整数の方が signed 型整数よりも効率が悪くなります。

■使用例

変数 c に変数 a と変数 b の和を代入します。

改善前ソースコード

```
unsigned short  a;
unsigned short  b;
int             c;
void f(void)
{
    c = b + a;
}
```

改善前アセンブリ展開コード

```
_f:
        MOV.L       L11,R1
        MOV.L       L11+4,R2
        MOV.W       @R1,R5
        EXTU.W      R5,R4
        MOV.L       L11+8,R5
        MOV.W       @R5,R7
        EXTU.W      R7,R7
        ADD         R7,R4
        RTS
        MOV.L       R4,@R2
L11:
        .DATA.L     _b
        .DATA.L     _c
        .DATA.L     _a
```

改善後ソースコード

```
short a;
short b;
int    c;
void f(void)
{
    c = b + a;
}
```

改善後アセンブリ展開コード

```
_f:
        MOV.L       L11,R1
        MOV.L       L11+4,R4
        MOV.W       @R1,R5
        MOV.W       @R4,R7
        MOV.L       L11+8,R2
        ADD         R7,R5
        RTS
        MOV.L       R5,@R2
L11:
        .DATA.L     _b
        .DATA.L     _a
        .DATA.L     _c
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	32	28	15	11
SH-2	32	28	15	11
SH-2A	32	28	8	8
SH2A-FPU	32	28	15	11
SH-2E	32	28	15	11
SH2-DSP(SH7065)	32	28	15	11
SH-3	32	28	15	11
SH3-DSP	32	28	15	13
SH-4	32	28	13	9
SH-4A	32	28	10	8
SH4AL-DSP	32	28	10	8

5.1.3 データサイズ（乗算）

■ポイント

乗算では、被乗数／乗数を[unsigned]char または[unsigned]short で宣言すると実行速度を向上できます。

■説明

SH-2、SH-2A、SH2AL-DSP、SH-2E、SH2-DSP、SH-3、SH-3DSP、SH-4、SH-4A および SH4AL-DSP の乗算は、被乗数／乗数が 1 バイト／2 バイトの場合、MULS.W/MULU.W 命令に展開されますが、4 バイトの場合、MUL.L 命令に展開されます。

SH-1 の乗算は、被乗数／乗数が 1 バイト／2 バイトの場合、MULS/MULU 命令に展開されますが、4 バイトの場合、ランタイムライブラリが呼び出されます。

■使用例

変数 a と b の積を求めて返します。

【注】コンパイルオプションは –cpu=sh1 の例です。

改善前ソースコード

```
int f( long a, long b )
{
  return( a * b );
}
```

改善前アセンブリ展開コード

```
_f:
        MOV.L      L11,R2
        MOV        R5,R1
        JMP        @R2
        MOV        R4,R0
L11:
        .DATA.L    _muli
```

改善後ソースコード

```
int f( short a, short b )
{
  return( a * b );
}
```

改善後アセンブリ展開コード

```
_f:
        STS.L      MACL,@-R15
        MULS       R5,R4
        STS        MACL,R0
        RTS
        LDS.L      @R15+,MACL
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	12	10	23	8

【注】a=1,b=2 の場合

5.1.4 データの構造化

■ポイント

関連するデータを構造体で宣言すると、実行速度を向上できる場合があります。

■説明

同一関数の中で何度も参照している場合、ベースアドレスがレジスタに割り付けられれば構造体の方が効率がよくなります。また、引数として渡す場合も効率が向上します。頻繁にアクセスするデータは構造体の先頭に集めると効果的です。

データを構造化すると、データの表現を変更するようなチューニングが容易になります。

■使用例

変数 a, b, c に数値を代入します。

改善前ソースコード

```
int a, b, c;
void f(void)
{
    a = 1;
    b = 2;
    c = 3;
}
```

改善後ソースコード

```
struct s{
    int a;
    int b;
    int c;
} s1;
void f(void)
{
    register struct s *p=&s1;

    p->a = 1;
    p->b = 2;
    p->c = 3;
}
```

改善前アセンブリ展開コード

```
_f:
          MOV.L      L11,R7
          MOV        #1,R1
          MOV.L      R1,@R7
          MOV.L      L11+4,R1
          MOV.L      L11+8,R2
          MOV        #2,R4
          MOV        #3,R5
          MOV.L      R4,@R1
          RTS
          MOV.L      R5,@R2
L11:
          .DATA.L    _a
          .DATA.L    _b
          .DATA.L    _c
```

改善後アセンブリ展開コード

```
_f:
          MOV.L      L11,R2
          MOV        #1,R1
          MOV        #2,R4
          MOV        #3,R5
          MOV.L      R1,@R2
          MOV.L      R4,@(4,R2)
          RTS
          MOV.L      R5,@(8,R2)
L11:
          .DATA.L    _s1
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	32	20	12	9
SH-2	32	20	12	9
SH-2A	32	20	9	6
SH2A-FPU	32	20	9	6
SH-2E	32	20	12	9
SH2-DSP(SH7065)	32	20	12	9
SH-3	32	20	14	10
SH3-DSP	32	20	15	11
SH-4	32	20	8	7
SH-4A	32	20	10	8
SH4AL-DSP	32	20	10	8

5.1.5 データの整合

■ポイント

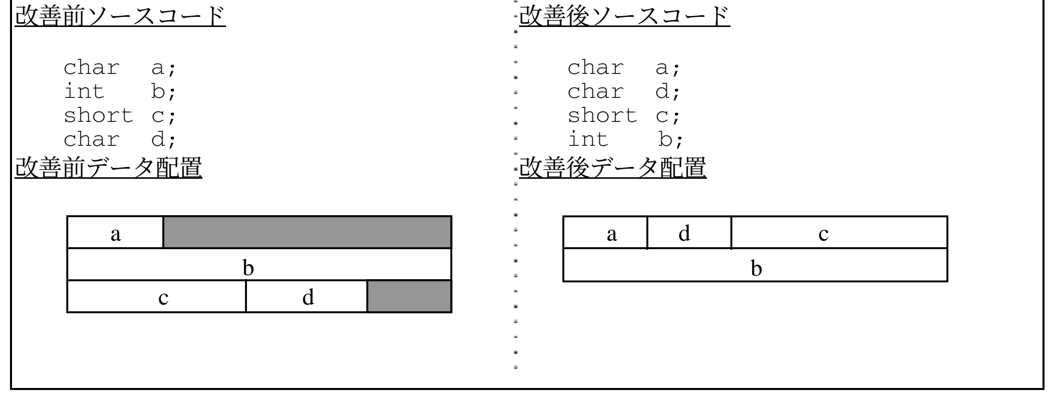
データの宣言順序を交換することにより、RAM 容量を削減できる場合があります。

■説明

大きさの異なる型の変数を宣言する場合は、同じ大きさの型の変数をまとめて宣言してください。これにより、データの整合によるデータ領域の空きが最小になります。

■使用例

全部で 8 バイトのデータを配置します。



5.1.6 初期値と const 型

■ポイント

値の変更がない初期値は、const 型で宣言してください。

■説明

初期化データは、通常、起動時に ROM エリアから RAM エリアに転送して、RAM エリアを使って処理を行います。このため、プログラム内で値が不変な初期化データの場合、確保した RAM エリアが無駄になります。初期化データに const 演算子を付けておくと、起動時の RAM エリアへの転送が抑止され、使用メモリ量の節約になります。

また、初期値は変更しない、というルールでプログラムを作成すると、ROM 化が容易になります。

■使用例

5 個の初期化データを設定します。

改善前ソースコード	改善後ソースコード
`char a[] =` `      {1, 2, 3, 4, 5};`	`const  char a[] =` `      {1, 2, 3, 4, 5};`
初期値を ROM から RAM へ転送して処理を行います	ROM 上の初期値を使用して処理を行います

5.1.7 局所変数と大域変数

■ポイント

一時変数、ループのカウンタなど、局所的に用いる変数は、関数の中で局所変数として宣言すると実行速度を向上できます。

■説明

局所変数として使用できるものは、大域変数として宣言しないで必ず局所変数として宣言してください。大域変数は、関数呼び出しやポインタ操作によって値が変化してしまう可能性があるので、大域的最適化の対象にはなりません。

局所変数を使用すると次の利点があります。

a. アクセスコストが安い。

b. レジスタに割り付けられる可能性がある。

c. 最適化の対象になる。

■使用例

10 回ループさせます。

改善前ソースコード

```
int i;

void f(void)
{
  for ( i = 0; i < 10; i++ );
}
```

改善前アセンブリ展開コード

```
_f:
        MOV.L      L218+2,R4
        MOV        #0,R3
        MOV        #10,R5
        BRA        L216
        MOV.L      R3,@R4
L217:
        MOV.L      @R4,R1
        ADD        #1,R1
        MOV.L      R1,@R4
L216:
        MOV.L      @R4,R3
        CMP/GE     R5,R3
        BF         L217
        RTS
        NOP
L218:
        .DATA.W    0
        .DATA.L    _i
```

改善後ソースコード

```
void f(void)
{
  int i;

  for ( i = 0; i < 10; i++ );
}
```

改善後アセンブリ展開コード

```
_f:
        MOV        #10,R4
L216:
        DT         R4
        BF         L216
        RTS
        NOP
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	20	12	54	52
SH-2	20	10	45	42
SH-2A	20	8	42	42
SH2A-FPU	20	8	42	42
SH-2E	20	10	45	42
SH2-DSP(SH7065)	20	10	54	51
SH-3	20	10	45	42
SH3-DSP	20	10	53	51
SH-4	20	10	34	33
SH-4A	20	10	25	23
SH4AL-DSP	20	10	50	46

5.1.8 ポインタ変数の活用

■ポイント

配列型を用いたプログラムはポインタ型を用いて書き直すと、実行速度を向上できる場合があります。（Ver.6）

■説明

配列参照 a[i]は、a[0]のアドレスに i 番目要素のアドレスを加算したコードが生成されます。ポインタ変数を用いれば、変数や演算の数を削減できる場合があります。

■使用例

配列の合計を求めます。

改善前ソースコード

```
int f1( int data[], int count )
{
      int ret = 0, i;

      for (i = 0; i < count; i++)
            ret += data[i]*i;
      return ret;
}
```

改善後ソースコード

```
int f2( int *data, int count )
{
      int ret = 0, i;

      for (i = 0; i < count; i++)
            ret += *data++ *i;
      return ret;
}
```

改善前アセンブリ展開コード

```
_f1:
        STS.L       MACL,@-R15
        MOV         #0,R7
        CMP/PL      R5
        BF/S        L219
        MOV         R7,R6
L220:
        MOV         R6,R0
        SHLL2       R0
        MOV.L       @(R0,R4),R3
        MUL.L       R6,R3
        ADD         #1,R6
        STS         MACL,R3
        CMP/GE      R5,R6
        BF/S        L220
        ADD         R3,R7
L219:
        MOV         R7,R0
        RTS
        LDS.L       @R15+,MACL
```

改善後アセンブリ展開コード

```
_f2:
        STS.L       MACL,@-R15
        MOV         #0,R7
        CMP/PL      R5
        BF/S        L221
        MOV         R7,R6
L222:
        MOV.L       @R4+,R3
        MUL.L       R6,R3
        ADD         #1,R6
        STS         MACL,R3
        CMP/GE      R5,R6
        BF/S        L222
        ADD         R3,R7
L221:
        MOV         R7,R0
        RTS
        LDS.L       @R15+,MACL
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	40	36	318	318
SH-2	34	30	179	159
SH-2E	34	30	178	158
SH2-DSP(SH7065)	34	30	207	187
SH-3	34	30	149	129
SH3-DSP	34	30	168	148
SH-4	34	30	117	97

【注】ただし、サイクル数は、count=10 での値

5.1.9 定数参照(1)

■ポイント

イミディエイト値は、できる限り 1 バイトで表現できるようにしておくとコードサイズを縮小できます。

■説明

1 バイトのイミディエイト値を使用すると、コード内に埋め込まれます。これに対し、2 バイトまたは 4 バイトのイミディエイト値を用いると、いったんメモリ上に置き、アクセスする形式になります。

■使用例

変数 I にイミディエイト値を代入します。

ソースコード(1)

```
int i
void f(void)
{
   i= 0x10000;
}
```

アセンブリ展開コード(1)

```
_f:
        MOV        #1,R2
        MOV.L      L12+2,R6
        SHLL16     R2
        RTS
        MOV.L      R2,@R6
L12:
        .RES.W     1
        .DATA.L    _i
```

ソースコード(2)

```
int i
void f(void)
{
   i= 0x01;
}
```

アセンブリ展開コード(2)

```
_f:
        MOV.L      L12+2,R6
        MOV        #1,R2
        RTS
        MOV.L      R2,@R6
L12:
        .RES.W     1
        .DATA.L    _i
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	ソースコード(1)	ソースコード(2)	ソースコード(1)	ソースコード(2)
SH-1	16	12	7	5
SH-2	16	12	7	5
SH-2A	16	12	4	4
SH2A-FPU	16	12	4	4
SH-2E	16	12	7	5
SH2-DSP(SH7065)	16	12	7	6
SH-3	16	12	7	5
SH3-DSP	16	12	6	6
SH-4	16	12	5	4
SH-4A	16	12	5	4
SH4AL-DSP	16	12	5	4

5.1.10 定数参照(2)

■ポイント

定数を用いた演算式をまとめても生成コードは増大しません。

■説明

定数の畳み込み機能があります。定数を式で表現してもコンパイル時に計算するので、生成コードには反映しません。

■使用例

変数aに定数を代入します。

改善前ソースコード

```
#define MASK1 0x1000
#define MASK2 0x10

int a = 0xffffffff;

void f(void)
{
    int x;

   x = MASK1;
   x |= MASK2;
   a &= x;
}
```

改善後ソースコード

```
#define MASK1 0x1000
#define MASK2 0x10

int a = 0xffffffff;

void f(void)
{
    a &= MASK1 | MASK2;
}
```

改善前アセンブリ展開コード

```
_f:
        MOV.W      L217,R4
        MOV.L      L217+4,R5
        MOV.L      @R5,R3
        AND        R4,R3
        RTS
        MOV.L       R3,@R5
L217:
        .DATA.W    H'1010
        .DATA.W     0
        .DATA.L    _a
```

改善後アセンブリ展開コード

```
_f:
        MOV.L      L216+4,R4
        MOV.W      L216,R3
        MOV.L      @R4,R2
        AND        R3,R2
        RTS
        MOV.L       R2,@R4
L216:
        .DATA.W    H'1010
        .DATA.W     0
        .DATA.L    _a
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	20	20	9	9
SH-2	20	20	9	9
SH-2A	20	20	7	7
SH2A-FPU	20	20	7	7
SH-2E	20	20	9	9
SH2-DSP(SH7065)	20	20	9	9
SH-3	20	20	9	9
SH3-DSP	20	20	9	9
SH-4	20	20	8	8
SH-4A	20	20	6	6
SH4AL-DSP	20	20	6	6

5.1.11 一定値になる変数(1)

■ポイント

変数が一定値になる場合、定数として扱うので、あらかじめ計算しておかなくてもメモリ効率と実行速度は変わりません。

■説明

定数になる変数にも定数の畳み込み機能が働き、この変数の値をトレースし、定数計算を行います。このため、読みやすくソースコードを記述しても生成コードは増大することはありません。

■使用例

変数 rc の結果によりリターン値を変えます。

変数値をあらかじめ計算しておく

ソースコード(1)

```
#define ERR     -1
#define NORMAL  0

int f(void)
{
   int rc, code;

   rc = 0;
   code = NORMAL;
   return( code );
}
```

アセンブリ展開コード(1)

```
_f:
        RTS
        MOV        #0,R0
```

C コンパイラに計算させる

ソースコード(2)

```
#define ERR     -1
#define NORMAL  0

int f(void)
{
   int rc, code;

   rc = 0;
   if ( rc ) code = ERR;
      else code = NORMAL;
   return( code );
}
```

アセンブリ展開コード(2)

```
_f:
        RTS
        MOV        #0,R0
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	ソースコード(1)	ソースコード(2)	ソースコード(1)	ソースコード(2)
SH-1	4	4	3	3
SH-2	4	4	3	3
SH-2A	4	4	4	4
SH2A-FPU	4	4	4	4
SH-2E	4	4	3	3
SH2-DSP(SH7065)	4	4	3	3
SH-3	4	4	3	3
SH3-DSP	4	4	3	3
SH-4	4	4	3	3
SH-4A	4	4	2	2
SH4AL-DSP	4	4	2	2

5.1.12 一定値になる変数(2)

■ポイント

変数が一定値になる場合、定数として扱うので、あらかじめ計算しておかなくてもメモリ効率と実行速度は変わりません。

■説明

定数になる変数にも定数の畳み込み機能が働き、この変数の値をトレースし、定数計算を行います。このため、読みやすくソースコードを記述しても生成コードは増大することはありません。

■使用例

変数 a と c の積を求め、変数 b に代入します。

変数値をあらかじめ計算しておく

ソースコード(1)

```
int f(void)
{
    int a, b;

    a = 3;
    b = 15;
    return b;
}
```

上記のアセンブリ展開コード(1)

```
_f:
        RTS
        MOV         #15,R0
```

C コンパイラに計算させる

ソースコード(2)

```
int f(void)
{
    int a, b, c;

    a = 3;
    c = 5;
    b = c * a;
    return b;
}
```

上記のアセンブリ展開コード(2)

```
_f:
        RTS
        MOV         #15,R0
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	ソースコード(1)	ソースコード(2)	ソースコード(1)	ソースコード(2)
SH-1	4	4	3	3
SH-2	4	4	3	3
SH-2E	4	4	4	4
SH-2A	4	4	4	4
SH2A-FPU	4	4	3	3
SH2-DSP(SH7065)	4	4	3	3
SH-3	4	4	3	3
SH3-DSP	4	4	3	3
SH-4	4	4	3	3
SH-4A	4	4	2	2
SH4AL-DSP	4	4	2	2

5.2 関数呼び出し

関数呼び出しに関して考慮すべき事項を**表** 5.4 に示します。

表 5.4 関数呼び出しにおける注意事項

項目	注意点	参照
関数位置	・関連の深い関数は１ファイルにまとめてください。	5.2.1
インタフェース	・引数がすべてレジスタに割り付くように（４個まで）引数の数を厳選してください。 ・引数が多い場合、構造体にしてポインタで渡してください。	5.2.3
関数分割	・非常に大きな関数では各種の最適化が効果的に行われない場合があります。テールリカージョンと呼ぶ機能を利用して、最適化が効果的に実行される大きさの関数にまで分割してください。	5.2.4
マクロへの置換	・関数呼び出しが多数ある場合、マクロにすれば実行速度を向上できます。ただし、マクロにするとプログラムサイズが増大するので、状況により選択してください。	-

RENESAS

5.2.1 関数のモジュール化

■ポイント

関連の深い関数は 1 ファイルにまとめることにより実行速度を向上できます。

■説明

異なるファイルにある関数を呼び出す場合、JSR 命令に展開されますが、同一ファイル内の関数呼び出しでは、呼び出し範囲が近いと BSR 命令に展開され、高速かつコンパクトなオブジェクトが生成されます。

また、モジュール化によって、チューンアップ時の修正が容易になります。

■使用例

関数 f から関数 g を呼び出します。

改善前ソースコード

```
extern g(void);
int f(void)
{
    g();
}
```

改善後ソースコード

```
int g(void)
{
}

int f(void)
{
    g();
}
```

改善前アセンブリ展開コード

```
_f:
        MOV.L     L216+2,R3
        JMP       @R3
        NOP
L216:
        .DATA.W     0
        .DATA.L   _g
```

改善後アセンブリ展開コード

```
_g:
        RTS
        NOP
_f:
        BRA       _g
        NOP
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	12	4	8	6
SH-2	12	4	8	6
SH-2A	12	4	8	6
SH2AL-DSP	12	4	8	6
SH-2E	12	4	8	6
SH2-DSP(SH7065)	12	4	9	6
SH-3	12	4	8	6
SH3-DSP	12	4	9	6
SH-4	12	4	8	5
SH-4A	12	4	5	4
SH4AL-DSP	12	4	5	4

■備考

BSR 命令で呼び出せる範囲は±4096 バイト（±2048 命令）です。

ファイルのサイズが大きくなりすぎると BSR を有効に使用できなくなります。

このような場合、頻繁に呼び合う関数を BSR 命令で呼び出せる位置に置くことをお勧めします。

5.2.2 ポインタ変数による関数呼び出し

■ポイント

switch 文による分岐の代わりにテーブルを用いることで実行速度を向上できます。

■説明

switch 文の各 case の処理がほぼ同じ場合は、テーブルを使用できないか検討してください。

■使用例

変数 a の値により呼び出し関数を変えます。

改善前ソースコード

```
extern void nop(void);
extern void stop(void);
extern void play(void);
void f(int a)
{
   switch (a)
    {
    case 0:
            nop(); break;
    case 1:
            stop(); break;
    case 2:
            play(); break;
    }
}
```

改善前アセンブリ展開コード

```
_f:
        MOV         R4,R0
        CMP/EQ      #0,R0
        BT          L220
        CMP/EQ      #1,R0
        BT          L221
        CMP/EQ      #2,R0
        BT          L222
        BRA         L223
        NOP
L220:
        MOV.L       L224,R3
        JMP         @R3
        NOP
L221:
        MOV.L       L224+4,R3
        JMP         @R3
        NOP
L222:
        MOV.L       L224+8,R3
        JMP         @R3
        NOP

L223:
        RTS
        NOP
L224:
        .DATA.L    _nop
        .DATA.L    _stop
        .DATA.L    _play
```

改善後ソースコード

```
extern void nop(void);
extern void stop(void);
extern void play(void);

static int (*key[3])() =
               {nop, stop, play};
void f(int a)
{
     (*key[a])();
}
```

改善後アセンブリ展開コード

```
_f:
        MOV.L      R4,@-R15
        MOV        R4,R3
        MOV.L      L241+2,R0
        SHLL2      R3
        MOV.L      @(R0,R3),R3
        JMP        @R3
        ADD        #4,R15
L241:
        .DATA.W     0
        .DATA.L     _$key
        .SECTION    D,DATA,ALIGN=4
_$key:
        .DATA.L     _nop,_stop,_play
```

■改善前後のコードサイズと実行速度

CPU 種別	**コードサイズ** [byte]		**実行速度** [cycle]	
	改善前	**改善後**	**改善前**	**改善後**
SH-1	52	16	15	11
SH-2	52	16	15	11
SH-2A	52	16	14	10
SH2A-FPU	52	16	14	10
SH-2E	52	16	15	11
SH2-DSP(SH7065)	52	16	16	12
SH-3	52	16	15	11
SH3-DSP	52	16	16	13
SH-4	52	16	13	10
SH-4A	52	16	10	8
SH4AL-DSP	52	16	10	8

5.2.3 関数のインタフェース

■ポイント

関数の引数を工夫することにより RAM 容量を削減でき、実行速度も向上できます。

「3.15.1 (2) **関数呼び出しのインタフェース**」を参照してください。

■説明

引数がすべてレジスタに乗るように（4 個まで）引数の数を厳選してください。引数が多い場合は、構造体にしてポインタで渡してください。引数がレジスタに乗れば、呼び出し、関数の出入り口の処理が簡単になります。また、スタック領域も節約できます。

なお、レジスタは R0～R3 がワークレジスタ、R4～R7 が引数用、R8～R14 が局所変数用です。

SH-2E,SH-4,SH-4A においては、浮動小数点レジスタで浮動小数点数を扱います。レジスタは FR0～FR3 がワークレジスタ、FR4～FR11 が引数用、FR12～FR14 が局所変数用です。

■使用例

関数 f の引数が引数用レジスタ個数よりも多く 5 個あります。

改善前ソースコード

```
int f(int, int, int, int, int);

void g(void)
{
      f(1, 2, 3, 4, 5);
}
```

改善後ソースコード

```
struct b{
     int a, b, c, d, e;
} b1 = {1, 2, 3, 4, 5};

int f(struct b *p);

void g(void)
{
     f(&b1);
}
```

改善前アセンブリ展開コード

```
_g:
        STS.L      PR,@-R15
        MOV        #5,R3
        MOV.L      L216+2,R2
        MOV        #4,R7
        MOV.L      R3,@-R15
        MOV        #3,R6
        MOV        #2,R5
        JSR        @R2
        MOV        #1,R4
        ADD        #4,R15
        LDS.L      @R15+,PR
        RTS
        NOP
L216:
        .DATA.W      0
        .DATA.L    _f
```

改善後アセンブリ展開コード

```
_g:
        MOV.L      L217,R4
        MOV.L      L217+4,R3
        JMP        @R3
        NOP
L217:
        .DATA.L    _b1
        .DATA.L    _f
```

■改善前後のコードサイズと実行速度

CPU 種別	**コードサイズ** [byte]		**実行速度** [cycle]	
	改善前	**改善後**	**改善前**	**改善後**
SH-1	32	16	17	7
SH-2	32	16	20	10
SH-2A	28	16	17	9
SH2A-FPU	28	16	17	9
SH-2E	32	16	20	10
SH2-DSP(SH7065)	32	16	28	14
SH-3	32	16	22	10
SH3-DSP	32	16	25	15
SH-4	32	16	18	10
SH-4A	32	16	15	6
SH4AL-DSP	32	16	15	6

5.2.4　テイルリカージョン

■ポイント

大きな関数を、関数の末尾で次々に関数を呼び出すように細かくモジュール分けしても実行速度を損ないません。

■説明

関数 funk1( )から呼び出されている関数 funk2( )において、関数 funk3( )を呼び出した場合、BSR 命令／JSR 命令で関数 funk3( )へ移行し、通常は関数 funk3( )の処理終了後 RTS 命令により関数 funk2( )へ戻り、さらに関数 funk2( )の処理終了後 RTS 命令により関数 funk1( )へ戻ります。（**図 5.1 左図**）

ここで、関数 funk2( )の末尾において関数 funk3( )を呼び出している場合には、BRA 命令／JMP 命令で関数 funk3( )へ移行し、関数 funk3( )の処理終了後、直接 RTS 命令により関数 funk1( )へ戻すことができます。（**図 5.1 右図**）この機能をテイルリカージョンと呼びます。

非常に大きなモジュールでは各種の最適化が効果的に行われない場合があります。本機能を利用して、最適化が効果的に実行される大きさのモジュールにまで分割することにより、性能を向上できます。

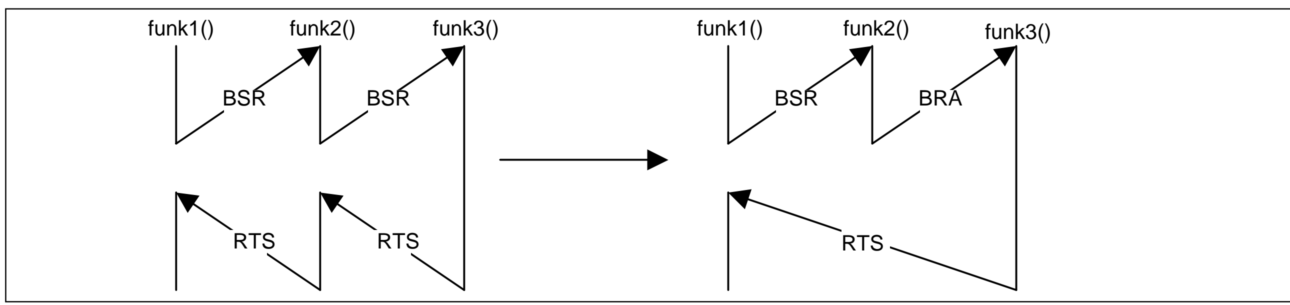
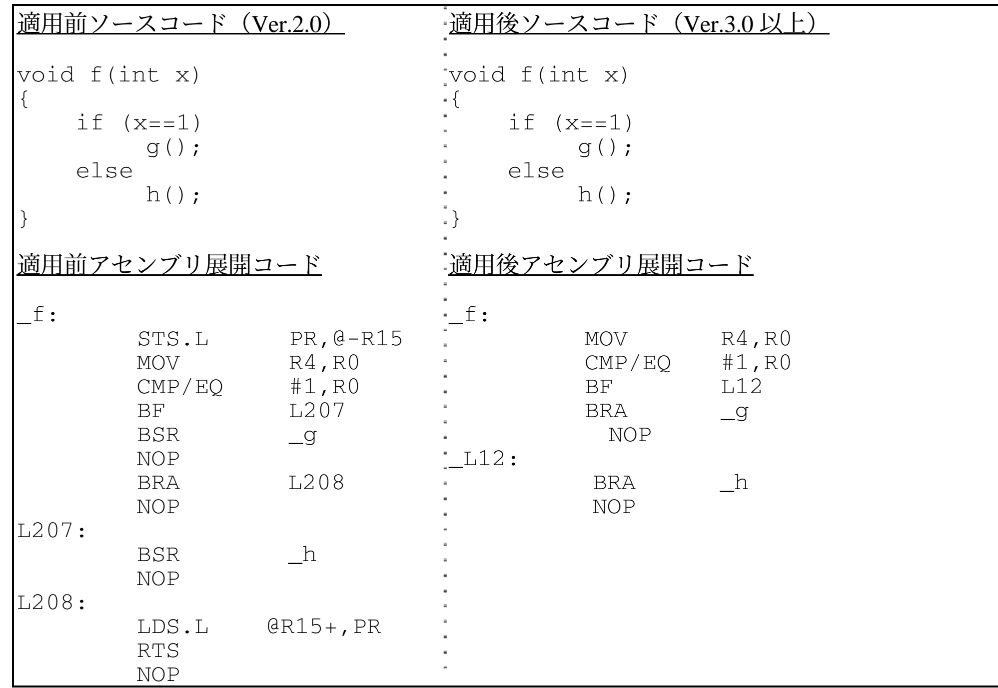


図 5.1　テイルリカージョン

■使用例

関数 f から関数 g と h を呼び出しています。g と h から戻るときには f を経由せずに f を呼び出している関数に直接戻ります。



適用前ソースコード（Ver.2.0）

```
void f(int x)
{
    if (x==1)
        g();
    else
        h();
}
```

適用前アセンブリ展開コード

```
_f:
        STS.L     PR,@-R15
        MOV       R4,R0
        CMP/EQ    #1,R0
        BF        L207
        BSR       _g
        NOP
        BRA       L208
        NOP
L207:
        BSR       _h
        NOP
L208:
        LDS.L    @R15+,PR
        RTS
        NOP
```

適用後ソースコード（Ver.3.0 以上）

```
void f(int x)
{
    if (x==1)
        g();
    else
        h();
}
```

適用後アセンブリ展開コード

```
_f:
          MOV       R4,R0
          CMP/EQ    #1,R0
          BF        L12
          BRA       _g
            NOP
_L12:
           BRA      _h
           NOP
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	適用前	適用後	適用前	適用後
SH-2	24	14	14	6

【注】 x=2 の場合

RENESAS

5.2.5 FSQRT,FABS 命令活用

■ポイント

数学関数 sqrt,fabs をライブラリコールしていませんか。
SH-4 命令セットには、FSQRT,FABS 命令があります。

■説明

fabs（浮動小数点数の絶対値）は数学関数ライブラリですが、関数アドレスを持たないようなプログラムではライブラリである必要がないため、直接 FABS 命令を使用します。

ただし、そのためには、<math.h> または、<mathf.h>をインクルードしている必要があります。

これらがないと、コンパイラは fabs を一般の関数としてコールするためライブラリコールとなり性能を低下させます。

また、ユーザがマクロ定義することも不要ですので注意してください。

＜マクロ例＞

```
#define fabs(a)   ((a)>=0?0:(-(a)))/* FABS 命令には展開されません */
```

■使用例

<math.h>をインクルードしない場合（ライブラリコール）とインクルードした場合（FABS 命令展開）の違い。

【注】コンパイルオプションは`-cpu=sh4△-fpu=single`の例です。

fabsf()を使用する場合は <mathf.h>のインクルードが必要です。

改善前ソースコード

```
float a,b;
f()
{
    :
    :
  b=fabs(a);
    :
}
```

改善前アセンブリ展開コード

```
_f:
        STS.L     PR,@-R15
        MOV.L     L12,R6
        MOV.L     L12+4,R1
        JSR       @R1
        FMOV.S    @R6,FR4
        MOV       R0,R4
        LDS       R4,FPUL
        FLOAT     FPUL,FR8
        MOV.L     L12+8,R5
        LDS.L     @R15+,PR
        RTS
        FMOV.S    FR8,@R5

L12:
        .DATA.L   _a
        .DATA.L   _fabs
        .DATA.L   _bW
```

改善後ソースコード

```
#include <math.h>
float a,b;
f()
{
    :
    :
  b=fabs(a);
    :
}
```

改善後アセンブリ展開コード

```
_f:
        MOV.L     L12,R1
        MOV.L     L12+8,R4
        FMOV.S    @R1,FR9
        FABS      FR9
        RTS
        FMOV.S    FR9,@R4
_L12:
        .DATA.L   _a
        .DATA.L   _fabs
        .DATA.L   _bW
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH2A-FPU	68	40	49	21
SH-2E	36	20	33	9
SH-4	36	20	29	8
SH-4A	36	20	22	6

5.3 演算方法

演算方式に関して考慮すべき事項を**表 5.5** に示します。

表 5.5 演算方式における注意事項

項目	注意点	参照
不変式／共通式の統合／移動	■関数内で共通に使用している部分式の一時変数への置換を検討してください。 ・for 文内で使用する不変式を for 文外に出してください。	5.3.1
ループ回数の削減	■ループ条件が同一または類似しているループ文のマージを検討してください。 ・ループの展開を試みてください。	5.3.2
演算方式の工夫	同じ演算はまとめて演算回数を削減してください。	5.3.3
公式の適用	■数学の公式を適用することにより演算回数を削減できないかを検討してください。	5.3.4
高速なアルゴリズムの利用	■配列におけるクイックソートのような計算時間が少なくてすむアルゴリズムを検討してください。	-
テーブルの活用	■switch 文の各 case の処理がほぼ同じ場合は、テーブルを使用できないか検討してください。 ■あらかじめ演算した結果をテーブルに代入しておき、演算結果 が必要になった際、テーブルの値を参照することで実行速度を向上させる手法があります。ただし、この手法は、ROM 容量の増大になるので、必要実行速度と余裕 ROM 要領との兼ね合いで選択してください。	5.3.5
条件式	定数との比較は 0 で行うと効率の良いコードが生成されます。	5.3.6
ロードストア削除	メモリアクセス（ロード、ストア）命令を削減することで、実行サイクルを減少させます。	5.3.7

RENESAS

5.3.1 ループ内不変式の移動

■ポイント

ループ内で値が変更されない式は、ループ開始前に計算すると実行速度を向上できます。（Ver.6）

■説明

ループ内で値が変更されない式をループ開始前に計算すると、毎回の計算が省略でき、実行命令数を低減できます。

■使用例

配列 a[ ]に配列要素 b[5]を代入します。

改善前ソースコード

```
extern int a[100], b[100];
void f(void)
{
     int i,j;

     j = 5;
     for ( i=0; i < 100; i++)
       a[i] = b[j];
}
```

改善後ソースコード

```
extern int a[100], b[100];
void f(void)
{
     int i,j,t;

     j = 5;
     for ( i=0, t=b[j];i < 100; i++)
       a[i] = t;
}
```

改善前アセンブリ展開コード

```
_f:
         MOV.L       L240+4,R5
         MOV         R5,R4
         MOV.W       L240,R6
         ADD         R5,R6
         MOV.L       L240+8,R5
L239:
         MOV.L       @R5,R3
         MOV.L       R3,@R4
         ADD         #4,R4
         CMP/HS      R6,R4
         BF          L239
         RTS
         NOP
L240:
         .DATA.W     H'0190
         .DATA.W     0
         .DATA.L     _a
         .DATA.L     H'00000014+_b
```

改善後アセンブリ展開コード

```
_f:
         MOV.L       L241+4,R5
         MOV.L       @R5,R5
         MOV.L       L241+8,R7
         MOV         R7,R4
         MOV.W       L241,R6
         ADD         R7,R6
L240:
         MOV.L       R5,@R4
         ADD         #4,R4
         CMP/HS      R6,R4
         BF          L240
         RTS
         NOP
L241:
         .DATA.W     H'0190
         .DATA.W     0
         .DATA.L     H'00000014+_b
         .DATA.L     _a
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	36	36	809	611
SH-2	36	36	809	611
SH-2E	36	36	809	611
SH2-DSP(SH7065)	36	36	908	611
SH-3	36	36	909	711
SH3-DSP	36	36	1008	711
SH-4	36	36	608	407

5.3.2 ループ回数の削減

■ポイント

ループを展開すると、実行速度は大幅に向上できます。

■説明

ループの展開は特に内側のループが有効です。ループの展開によりプログラムサイズは増大するので、プログラムサイズを犠牲にしても実行速度を向上させたい場合に適用してください。

■使用例

配列 a[ ]を初期化します。

改善前ソースコード

```
extern int a[100];
void f(void)
{
     int i;

     for ( i = 0; i < 100; i++)
          a[i] = 0;
}
```

改善後ソースコード

```
extern int a[100];
void f(void)
{
     int i;

     for ( i = 0; i < 100; i+=2)
     {
          a[i] = 0;
          a[i+1] = 0;
     }
}
```

改善前アセンブリ展開コード

```
_f:
         MOV.L      L238+2,R7
         MOV        #0,R5
         MOV.W      L238,R6
         MOV        R7,R4
         ADD        R7,R6
L237:
         MOV.L      R5,@R4
         ADD        #4,R4
         CMP/HS     R6,R4
         BF         L237
         RTS
         NOP
L238:
         .DATA.W    H'0190
         .DATA.L    _a
```

改善後アセンブリ展開コード

```
_f:
         MOV.L      L239+2,R7
         MOV        #0,R5
         MOV.W      L239,R0
         MOV        R7,R6
         ADD        #4,R6
         MOV        R7,R4
         ADD        R7,R0
L238:
         MOV.L      R5,@R4
         MOV.L      R5,@R6
         ADD        #8,R4
         CMP/HS     R0,R4
         BF/S       L238
         ADD        #8,R6
         RTS
         NOP
L239:
         .DATA.W    H'0190
         .DATA.L    _a
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	24	28	805	455
SH-2	24	24	506	356
SH-2A	20	24	403	253
SH-2A-FPU	20	24	403	253
SH-2E	24	24	506	356
SH2-DSP(SH7065)	24	24	605	605
SH-3	24	24	606	407
SH3-DSP	24	24	705	503
SH-4	24	24	305	204
SH-4A	24	24	405	255
SH4AL-DSP	24	24	405	255

5.3.3 乗算／除算の使用

■ポイント

乗算／除算とシフト演算のどちらを適用するかに迷うときは、乗算／除算を使用してください。

■説明

まずは、プログラムを読みやすく記述してみてください。乗算／除算は、乗数／除数と被乗数／被除数が符号なしの場合、コンパイラの最適化によりシフト演算の組み合わせに置換されます。

■使用例

乗算／除算を実行します。

ソースコード（乗算）

```
unsigned int a;

int f(void)
{
    return(a*4);
}
```

上記のアセンブリ展開コード

```
_f:
        MOV.L      L217,R3
        MOV.L      @R3,R0
        RTS
        SHLL2      R0
L217:
        .DATA.L    _a
```

ソースコード（除算）

```
unsigned int b;

int f(void)
{
    return(b/2);
}
```

上記のアセンブリ展開コード

```
_f:
        MOV.L      L217,R3
        MOV.L      @R3,R0
        RTS
        SHLR       R0
L217:
        .DATA.L    _b
```

5.3.4 公式の適用

■ポイント

数学の公式を適用することにより演算回数を削減できれば、実行速度を向上できます。

■説明

数学の公式によっては解析的には簡単になりますが、算術的には適用すると演算回数が増加することがありますので注意してください。

■使用例

1 から n までの総和を求めます。

改善前ソースコード

```
int f( long n )
{
    int i, s;

    for (s = 0, i = 1;
          i <= n; i++)
          s += i;
    return( s );
}
```

改善後ソースコード

```
int f( long n )
{
        return( n*(n+1) >> 1 );
}
```

改善前アセンブリ展開コード

```
_f:
        MOV         #1,R5
        CMP/GT      R4,R5
        BT/S        L218
        MOV         #0,R6
L219:
        ADD         R5,R6
        ADD         #1,R5
        CMP/GT      R4,R5
        BF          L219
L218:
        RTS
        MOV         R6,R0
```

改善後アセンブリ展開コード

```
_f:
        STS.L       MACL,@-R15
        MOV         R4,R0
        ADD         #1,R0
        MUL.L       R4,R0
        STS         MACL,R0
        SHAR        R0
        RTS
        LDS.L       @R15+,MACL
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	18	24	609	31
SH-2	18	16	609	21
SH-2A	16	12	608	10
SH2A-FPU	16	12	608	10
SH-2E	18	16	609	14
SH2-DSP(SH7065)	18	16	710	15
SH-3	18	16	609	18
SH3-DSP	18	16	710	18
SH-4	18	16	507	14
SH-4A	18	16	407	8
SH4AL-DSP	18	16	407	8

【注】サイクル数は n=100 の場合

5.3.5 テーブルの活用

■ポイント

switch 文による分岐の代わりにテーブルを用いることで実行速度を向上できます。

■説明

switch 文の各 case の処理がほぼ同じ場合は、テーブルを使用できないか検討してください。

■使用例

変数 i の値により変数 ch に代入する文字定数を変えます。

<table>
<tr><td>

改善前ソースコード

```
char f (int i)
{
    char ch;

    switch (i)
    {
       case 0:
           ch = ‘a’; break;
       case 1:
           ch = ‘x’; break;
       case 2:
           ch = ‘b’; break;
    }
    return (ch);
}
```

</td><td>

改善後ソースコード

```
char chbuf[] = { ‘a’, ‘x’, ‘b’ };

char f(int i)
{
      return (chbuf[i]);
}
```

</td></tr>
<tr><td>

改善前アセンブリ展開コード

```
_f:
        MOV         R4,R0
        CMP/EQ      #0,R0
        BT          L218
        CMP/EQ      #1,R0
        BT          L219
        CMP/EQ      #2,R0
        BT          L220
        BRA         L221
        NOP
L218:
        BRA         L221
        MOV         #97,R4
L219:
        BRA         L221
        MOV         #120,R4
L220:
        MOV         #98,R4
L221:
        RTS
        MOV         R4,R0
```

</td><td>

改善後アセンブリ展開コード

```
_f:
        MOV.L      L218+2,R0
        RTS
        MOV.B      @(R0,R4),R0
L218:
        .DATA.W     0
        .DATA.L    _chbuf
```

</td></tr>
</table>

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	32	12	13	5
SH-2	32	12	13	5
SH-2A	30	12	11	7
SH2A-FPU	30	12	11	7
SH-2E	32	12	13	5
SH2-DSP(SH7065)	32	12	14	5
SH-3	32	12	13	5
SH3-DSP	32	12	14	6
SH-4	32	12	10	4
SH-4A	32	12	10	4
SH4AL-DSP	32	12	10	4

【注】i=2 の場合

RENESAS

5.3.6 条件式

■ポイント

定数との比較は 0 で行うと効率の良いコードが生成されます。

■説明

0 との比較をする場合、定数値をロードする命令が生成されないので 0 以外との比較をする場合に比べ短いコードが生成されます。ループや if 文などの条件式は 0 との比較になるように設定してください。

■使用例

引数の値が 1 以上かどうかによりリターン値を変えます。

改善前ソースコード

```
int f (int x)
{
     if ( x >= 1 )
          return 1;
     else
          return 0;
}
```

改善前アセンブリ展開コード

```
_f:
         MOV         #1,R3
         CMP/GE       R3,R4
         MOVT        R0
         RTS
         NOP
```

改善後ソースコード

```
int f (int x)
{
     if ( x > 0 )
          return 1;
     else
          return 0;
}
```

改善後アセンブリ展開コード

```
_f:
         CMP/PL      R4
         MOVT        R0
         RTS
         NOP
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	8	6	5	4
SH-2	8	6	5	4
SH-2A	8	6	6	5
SH2A-FPU	8	6	6	5
SH-2E	8	6	5	4
SH2-DSP(SH7065)	8	6	5	4
SH-3	8	6	5	4
SH3-DSP	8	6	5	4
SH-4	8	6	5	4
SH-4A	8	6	4	3
SH4AL-DSP	8	6	4	3

5.3.7 ロードストア削除

■ポイント

メモリアクセス（ロード、ストア）命令を削減することで、実行サイクルを減少させます。

■説明

座標計算において、x,y,z 値を毎回、メモリへロードストアすることは、性能を低下させる大きな原因となります。できるだけ、座標値を構造体データ上で演算せずに FPU レジスタ変数上で演算させ、メモリロードストアを減少させるようなプログラムにすることで、実行スピードの向上を図ります。

■使用例

固定点 P と P0,P1,P2 から形成される面の各頂点距離（二乗値）を求め、距離を判定する。

【注】コンパイルオプションは –cpu=sh4△-fpu=single の例です。

改善前ソースコード

```
#define SCAL2(v)  ((v)->x*(v)->x ¥
             +(v)->y*(v)->y¥
             +(v)->z*(v)->z)
#define SubVect(a,b)
             ((a)->x-=(b)->x,¥
             (a)->y -= (b)->y,¥
             (a)->z -= (b)->z)

typedef  struct {
    float x,y,z;
} POINT3;
typedef  struct  {
   POINT3*   v;
} POLI;

int f(POINT3 *p, POLI *poli,
                        float rad)
{
    float  dst2;
    POINT3 dv;

    dv=poli->v[0];
    SubVect(&dv,p);
    dst2=SCAL2(&dv);
    if (dst2>rad) return 0;

    dv=poli->v[1];
    SubVect(&dv,p);
    dst2=SCAL2(&dv);
    if (dst2>rad) return 0;

    dv=poli->v[2];
    SubVect(&dv,p);
    dst2=SCAL2(&dv);
    if (dst2>rad) return 0;

    return 1;
}
```

改善後ソースコード

```
typedef  struct {
    float x,y,z;
} POINT3;
typedef  struct  {
   POINT3*   v;
} POLI;

float scal2(POINT3 *p1, POINT3 *q1)
{
    float a,b,c;
    float d,e,f;
    float *p=(float *)p1,*q=(float
*)q1;

    a=*p++; d=*q++;
    b=*p++; e=*q++; a-=d;
    c=*p++; f=*q++; b-=e;
                    c-=f;
    return a*a+b*b+c*c;
}


int f(POINT3 *p,POLI *poli, float rad)
{
    float   d;
    POINT3  *q;

    q=poli->v;
    d2=scal2(q++,p);
    if (d2>rad) return 0;

    d2=scal2(q++,p);
    if (d2>rad) return 0;

    d2=scal2(q++,p);
    if (d2>rad) return 0;

    return 1;
}
```

改善前アセンブリ展開コード

```
_f:
        ADD        #-16,R15
        MOV.L      @R5,R1
        MOV        #4,R0
        FMOV.S     FR4,FR5
        MOV.L      @R1,R6
        MOV.L      @(4,R1),R5
```

改善後アセンブリ展開コード

```
_scal2:
        FMOV.S     @R4,FR0
        FMOV.S     @R5,FR8
        ADD        #4,R4
        ADD        #4,R5
        FSUB       FR8,FR0
        FMOV.S     @R4,FR9
```

```
        MOV.L       @(8,R1),R7
        MOV.L       R6,@R15
        MOV.L       R5,@(4,R15)
        MOV.L       R7,@(8,R15)
        FMOV.S      @R4,FR4
        FMOV.S      @R15,FR8
        FMOV.S      @(R0,R4),FR7
        FSUB        FR4,FR8
        MOV.L       R1,@(12,R15)
        FMOV.S      FR8,@R15
        FMOV.S      @(R0,R15),FR8
        FSUB        FR7,FR8
        FMOV.S      FR8,@(R0,R15)
        MOV         #8,
        FMOV.S      @(R0,R4),FR6
        FMOV.S      @(R0,R15),FR8
        FMOV.S      @R15,FR0
        FSUB        FR6,FR8
        FMOV.S      FR8,@(R0,R15)
        MOV         #4,R0
        FMOV.S      @(R0,R15),FR9
        FMOV.S      @(R0,R15),FR8
        MOV         #8,R0
        FMUL        FR8,FR9
        FMOV.S      @(R0,R15),FR8
        FMAC        FR0,FR0,FR9
        FMOV.S      @(R0,R15),FR0
        FMAC        FR0,FR8,FR9
        FCMP/GT     FR5,FR9
        BT          L12
        MOV.L       @(12,R1),R6
        MOV         #4,R0
        MOV.L       @(16,R1),R4
        MOV.L       @(20,R1),R5
        MOV.L       R6,@R15
        MOV.L       R4,@(4,R15)
        MOV.L       R5,@(8,R15)
        FMOV.S      @R15,FR8
        FSUB        FR4,FR8
        FMOV.S      FR8,@R15
        FMOV.S      @(R0,R15),FR8
        FSUB        FR7,FR8
        FMOV.S      FR8,@(R0,R15)
        MOV         #8,R0
        FMOV.S      @(R0,R15),FR8
        FSUB        FR6,FR8
        FMOV.S      FR8,@(R0,R15)
        MOV         #4,R0
        FMOV.S      @(R0,R15),FR9
        FMOV.S      @(R0,R15),FR8
        MOV         #8,R0
        FMOV.S      @R15,FR0
        FMUL        FR8,FR9
        FMOV.S      @(R0,R15),FR8
        FMAC        FR0,FR0,FR9
        FMOV.S      @(R0,R15),FR0
        FMAC        FR0,FR8,FR9
        FCMP/GT     FR5,FR9
        BT          L12
        MOV.L       @(24,R1),R6
        MOV         #4,R0
        MOV.L       @(28,R1),R7
        MOV.L       @(32,R1),R4
        MOV.L       R6,@R15
        MOV.L       R7,@(4,R15)
        MOV.L       R4,@(8,R15)
        FMOV.S      @R15,FR8
        FSUB        FR4,FR8
        FMOV.S      FR8,@R15
        FMOV.S      @(R0,R15),FR8
        FSUB        FR7,FR8
        FMOV.S      FR8,@(R0,R15)
        MOV         #8,R0
        :
        FMOV.S      @R5,FR8
        MOV         #4,R0
        FMOV.S      @(R0,R4),FR6
        FSUB        FR8,FR9
        FMOV.S      @(R0,R5),FR8
        FSUB        FR8,FR6
        FMOV.S      FR9,FR8
        FMUL        FR9,FR8
        FMAC        FR0,FR0,FR8
        FMOV.S      FR6,FR0
        FMAC        FR0,FR6,FR8
        RTS
        FMOV.S      FR8,FR0
_f:
        MOV.L       R13,@-R15
        MOV.L       R14,@-R15
        STS.L       PR,@-R15
        FMOV.S      FR14,@-R15
        MOV.L       @R5,R14
        MOV         R4,R13
        FMOV.S      FR4,FR14
        MOV         R4,R5
        MOV         R14,R4
        BSR         _scal2
        ADD         #12,R14
        FCMP/GT     FR14,FR0
        BT          L18
        MOV         R13,R5
        BSR       _ _scal2
        MOV         R14,R4
        FCMP/GT     FR14,FR0
        BT/S        L18
        ADD         #12,R14
        MOV         R13,R5
        BSR         _scal2
        MOV         R14,R4
        FCMP/GT     FR14,FR0
        BF/S        L20
        MOV         #1,R0
L18:
        MOV         #0,R0
L20:
        FMOV.S      @R15+,FR14
        LDS.L       @R15+,PR
        MOV.L       @R15+,R14
        RTS
        MOV.L       @R15+,R13
```

```
        FMOV.S     @(R0,R15),FR8
        FSUB       FR6,FR8
        FMOV.S     FR8,@(R0,R15)
        MOV        #4,R0
        FMOV.S     @(R0,R15),FR9
        FMOV.S     @(R0,R15),FR8
        MOV        #8,R0
        FMOV.S     @R15,FR0
        FMUL       FR8,FR9
        FMOV.S     @(R0,R15),FR8
        FMAC       FR0,FR0,FR9
        FMOV.S     @(R0,R15),FR0
        FMAC       FR0,FR8,FR9
        FCMP/GT    FR5,FR9
        BF/S       L14
        MOV        #1,R0
L12:
        MOV        #0,R0
L14:
        RTS
        ADD        #16,R15
```

■改善前後のコードサイズと実行速度

CPU 種別	**コードサイズ** [byte]		**実行速度** [cycle]	
	改善前	**改善後**	**改善前**	**改善後**
SH2A-FPU	196	90	115	106
SH-2E	196	62	141	128
SH-4	196	62	123	87
SH-4A	196	62	97	93

■改善前後プログラム解説

両者のロードストアの回数を比較する。
x,y,z で 1 回とすると

改善前は

```
dv=poli->v[0];            LOAD 1回
SubVect(&dv,p);           LOAD 2回  STORE 1回
dst2=SCAL2(&dv);          LOAD 2回
if (dst2>rad) rerun 0;
```

これを 3 回繰り返すため、合計 18 回の LOAD/STORE である。
改善後は

```
a=*p++; d=*q++;
b=*p++; e=*q++; a-=d;
c=*p++; f=*q++; b-=e;
                c-=f;
return a*a+b*b+c*c;
```

p と q のロードで計 2 回×3 回＝合計 6 回の LOAD/STORE となります。

このようにメモリアクセスを 1/3 に減少することができます。SuperH マイコンの命令セットには、基本的にメモリとの演算命令がないことから、FPU レジスタ上の演算に比べ命令数が多くなります。

また、メモリへのストアはパイプラインの乱れを起こす原因にもなります。メモリアクセスを減少させることは、パイプラインの流れをスムーズにさせることにもつながります。

■補足

改善後のプログラムでは、固定点 P を 3 回ロードしています。
これを、1 回のロードですむように改善すれば、さらに効果があがります。
一般に固定点に対して、複数の面に対するループ処理を行うことを考えると、固定点は、構造体でなく一度 FPU レジスタ変数にロードしてから、演算を行うようなプログラムに変更してください。

5.4 分岐

分岐に関して考慮すべき事項を以下に示します。

- 同じ判定はまとめてください。
- switch 文､else if 文が長い場合､早く処理したいケースや頻繁に分岐するケースを先頭近くに置いてください。
- switch 文、else if 文が長い場合、段階を分けて判定することにより実行速度を向上できます。

■ポイント

case の数が 5～6 個までの switch 文は if 文にすると実行速度を向上できます。

■説明

case の数が少ない switch 文は if 文に置換してください。
switch 文は case 値のテーブルを引く前に変数の値の範囲をチェックするので、オーバヘッドがあります。
一方、if 文は何度も比較するので、場合分けが増えると効率が低下します。

■使用例

変数 a の値により返却値を変えます。

改善前ソースコード

```
int  x(int a)
{
   switch (a)
   {
     case 1:
          a = 2; break;
     case 10:
          a = 4; break;
     default:
          a = 0; break;
   }
   return (a);
}
```

改善後ソースコード

```
int x (int a)
{
   if (a==1)
        a = 2;
   else if (a==10)
        a = 4;
   else
        a = 0;
   return (a);
}
```

改善前アセンブリ展開コード

```
_x:
        MOV        R4,R0
        CMP/EQ     #1,R0
        BT         L16
        CMP/EQ     #10,R0
        BT         L17
        BRA        L18
        NOP
L16:
        BRA        L19
        MOV        #2,R2
L17:
        BRA        L19
        MOV        #4,R2
L18:
        MOV        #0,R2
L19:
        RTS
        MOV        R2,R0
```

改善後アセンブリ展開コード

```
_x:
        MOV        R4,R0
        CMP/EQ     #1,R0
        BF         L22
        BRA        L23
        MOV        #2,R4
L22:
        CMP/EQ     #10,R0
        BF/S       L23
        MOV        #0,R4
        MOV        #4,R4
L23:
        RTS
        MOV        R4,R0
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	28	22	11	9
SH-2	28	22	11	9
SH-2A	22	20	8	5
SH2A-FPU	22	20	8	5
SH-2E	28	22	11	9
SH2-DSP(SH7065)	28	22	12	10
SH-3	28	22	11	9
SH3-DSP	28	22	12	10
SH-4	28	22	8	7
SH-4A	28	22	7	7
SH4AL-DSP	28	22	7	7

【注】a=1 のとき

5.5 インライン展開

インライン展開に関して考慮すべき事項を**表** 5.6 に示します。

表 5.6 **インライン展開における注意事項**

項目	注意点	参照
関数のインライン展開	・頻繁に呼び出される関数はインライン展開を試みてください。ただし、関数を展開するとプログラムサイズが増大するので実行速度と ROM 容量との兼ね合いで選択してください。	5.5.1
アセンブラ埋め込みインライン展開	・アセンブラコードで記述されたプログラムを C 言語の関数と同じインタフェースで呼び出せます。	5.5.2

5.5.1 関数のインライン展開

■ポイント

頻繁に呼び出される関数をインライン展開すると実行速度を向上できます。

■説明

頻繁に呼び出される関数をインライン展開することにより、実行速度の向上が図れます。特にループ内で呼ばれる関数などを展開すると大きな効果を得られる場合もあります。しかし、インライン展開をした場合、プログラムサイズが増大する傾向にありますので、プログラムサイズを犠牲にしても実行速度を向上させたい場合に適用してください。

■使用例

配列 a と配列 b の要素を交換します。

改善前ソースコード

```
int x[10], y[10];
static void g(int *a, int *b, int i)
{
      int temp;

      temp = a[i];
      a[i] = b[i];
      b[i] = temp;
}

void f (void)
{
      int i;

      for (i=0;i<10;i++)
              g(x, y, i);
}
```

改善前アセンブリ展開コード

```
_$g:
        SHLL2      R6
        MOV        R6,R0
        MOV.L      @(R0,R4),R1
        MOV.L      @(R0,R5),R2
        MOV.L      R2,@(R0,R4)
        RTS
        MOV.L      R1,@(R0,R5)
_f:
        MOV.L      R11,@-R15
        MOV.L      R12,@-R15
        MOV.L      R13,@-R15
        MOV.L      R14,@-R15
        STS.L      PR,@-R15
        MOV        #0,R14
        MOV.L      L14+2,R12
        MOV.L      L14+6,R13
        MOV        #10,R11
L12:
        MOV        R14,R6
        MOV        R12,R4
        MOV        R13,R5
        BSR        _$g
        ADD        #1,R14
        CMP/GE     R11,R14
        BF         L12
        LDS.L      @R15+,PR
        MOV.L      @R15+,R14
        MOV.L      @R15+,R13
        MOV.L      @R15+,R12
        RTS
        MOV.L      @R15+,R11
L14:
        .RES.W     1
        .DATA.L    _x
        .DATA.L    _y
```

改善後ソースコード

```
int x[10], y[10];
#pragma inline (g)
static void g(int *a, int *b, int i)
{
      int temp;

      temp = a[i];
      a[i] = b[i];
      b[i] = temp;
}

void f (void)
{
      int i;

      for (i=0;i<10;i++)
              g(x, y, i);
}
```

改善後アセンブリ展開コード

```
_f:
        MOV        #10,R1
        MOV.L      L13+2,R4
        MOV.L      L13+6,R5
L11:
        MOV.L      @R5,R6
        MOV.L      @R4,R2
        DT         R1
        MOV.L      R2,@R5
        MOV.L      R6,@R4
        ADD        #4,R5
        BF/S       L11
        ADD        #4,R4
        RTS
        NOP
L13:
        .RES.W     1
        .DATA.L    _y
        .DATA.L    _x
```

■ 改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	54	36	210	137
SH-2	54	36	210	118
SH-2A	38	32	164	74
SH2A-FPU	50	32	187	74
SH-2E	54	36	210	118
SH2-DSP(SH7065)	52	36	305	138
SH-3	54	36	234	147
SH3-DSP	52	36	294	156
SH-4	54	36	203	97
SH-4A	54	36	155	85
SH4AL-DSP	52	36	185	85

5.5.2 アセンブラ埋め込みのインライン展開

■ポイント

Ｃプログラム中にアセンブラコードを記述し、実行速度を向上できます。

■説明

性能上、特に実行速度を向上したい場合、アセンブラで記述したいことがあります。そのような場合、必要な部分だけをアセンブラで記述し、その部分をＣ言語の関数と同じ要領で呼び出すことができます。ただし本機能は -code=asmcode でアセンブラを生成するときのみ有効です。

■使用例

配列 big の要素の上位バイトと下位バイトを入れ換えて、配列 little に格納します。

改善前ソースコード

```
#define A_MAX 10
typedef unsigned char Uchar;
short big[A_MAX],little[A_MAX];
short swap(short p1)
{
    short ret;

    *((Uchar *)(&ret)+1) =
     *((Uchar *)(&p1));
    *((Uchar *)(&ret)) =
     *((Uchar *)(&p1)+1);
   return ret;
}

void f (void)
{
    int  i;
    short *x, *y;

    x = little;
    y = big;
    for(i=0; i<A_MAX; i++,
                      x++, y++){
        *x = swap(*y);
    }
}
```

改善後ソースコード

```
#define A_MAX 10
#pragma inline_asm (swap)
typedef unsigned char Uchar;
short big[A_MAX],little[A_MAX];
short swap(short p1)
{
   SWAP.B R4,R0
}

void f (void)
{
   int  i;
   short *x, *y;

   x = little;
   y = big;
   for(i=0; i<A_MAX; i++, x++, y++){
           *x = swap(*y);
  }
}
```

改善前アセンブリ展開コード

```
_swap:
        ADD       #-8,R15
        MOV       R4,R0
        MOV.L     R4,@R15
        MOV.W     R0,@(2,R15)
        MOV.B     @(2,R15),R0
        MOV.B     R0,@(5,R15)
        MOV.B     @(3,R15),R0
        MOV.B     R0,@(4,R15)
        MOV.W     @(4,R15),R0
        RTS
        ADD       #8,R15
_f:
        MOV.L     R12,@-R15
        MOV.L     R13,@-R15
        MOV.L     R14,@-R15
        STS.L     PR,@-R15
        MOV.L     L14+2,R13
        MOV.L     L14+6,R14
        MOV       #10,R12
L12:
        BSR       _swap
        MOV.W     @R14+,R4
        DT        R12
```

改善後アセンブリ展開コード

```
_swap:
        SWAP.B    R4,R0
        .ALIGN    4
        RTS
        NOP
_f:
        MOV.L     R12,@-R15
        MOV.L     R13,@-R15
        MOV.L     R14,@-R15
        MOV.L     L15,R13
        MOV.L     L15+4,R14
        MOV       #10,R12
L12:
        BRA       L14
        MOV.W     @R14+,R4
L15:
        .DATA.L   _little
        .DATA.L   _big
L14:
        SWAP.B    R4,R0
        .ALIGN    4
        DT        R12
        MOV.W     R0,@R13
        BT/S      L17
```

```
        MOV.W       R0,@R13             ADD         #2,R13
        BF/S        L12                 MOV.L       L16+2,R3
        ADD         #2,R13              JMP         @R3
        LDS.L       @R15+,PR            NOP
        MOV.L       @R15+,R14      L17:
        MOV.L       @R15+,R13           MOV.L       @R15+,R14
        RTS                             MOV.L       @R15+,R13
        MOV.L       @R15+,R12           RTS
L14:                                    MOV.L       @R15+,R12
        .RES.W      1              L16:
        .DATA.L     _little             .RES.W      1
        .DATA.L     _big                .DATA.L     L12
```

■改善前後のコードサイズと実行速度

CPU 種別	**コードサイズ** [byte]		**実行速度** [cycle]	
	改善前	**改善後**	**改善前**	**改善後**
SH-1	46	68	285	172
SH-2	46	64	106	171
SH-2A	30	60	202	126
SH2A-FPU	42	76	215	148
SH-2E	46	64	286	171
SH2-DSP(SH7065)	46	64	318	193
SH-3	46	64	113	185
SH3-DSP	46	64	112	177
SH-4	46	64	62	108
SH-4A	46	64	182	117
SH4AL-DSP	46	64	182	117

5.6 グローバルベースレジスタ（GBR）

5.6.1 グローバルベースレジスタ（GBR）を使ったオフセット参照

■ポイント

外部変数を GBR を使ったオフセット参照にすることにより、性能を向上させることができます。

■説明

頻繁にアクセスされる外部変数は GBR をベースレジスタとしたオフセット参照にすることにより、コンパクトなオブジェクトが生成されます。また、実行命令数の削減にもつながるので実行速度が向上する場合もあります。

■使用例

構造体 y の内容を構造体 x に代入します。

【注】コンパイルオプションは –cpu=sh2 –gbr=user の例です。

改善前ソースコード

```
struct {
    char  c1;
    char  c2;
    short s1;
    short s2;
    long  l1;
    long  l2;
} x, y;

void f (void)
{
   x.c1 = y.c1;
   x.c2 = y.c2;
   x.s1 = y.s1;
   x.s2 = y.s2;
   x.l1 = y.l1;
   x.l2 = y.l2;
}
```

改善前アセンブリ展開コード

```
_f:
        MOV.L      L12,R5
        MOV.L      L12+4,R6
        MOV.B      @(1,R5),R0
        MOV.B      @R5,R1
        MOV.B      R0,@(1,R6)
        MOV.W      @(2,R5),R0
        MOV.L      @(8,R5),R4
        MOV.W      R0,@(2,R6)
        MOV.W      @(4,R5),R0
        MOV.L      @(12,R5),R7
        MOV.B      R1,@R6
        MOV.W      R0,@(4,R6)
        MOV.L      R4,@(8,R6)
        RTS
        MOV.L      R7,@(12,R6)

L12:
        .DATA.L    _y
        .DATA.L    _x
```

改善後ソースコード

```
#pragma gbr_base(x,y)
struct {
    char  c1;
    char  c2;
    short s1;
    short s2;
    long  l1;
    long  l2;
} x, y;

void f (void)
{
    x.c1 = y.c1;
    x.c2 = y.c2;
    x.s1 = y.s1;
    x.s2 = y.s2;
    x.l1 = y.l1;
    x.l2 = y.l2;
}
```

改善後アセンブリ展開コード

```
_f:
  MOV.B  @(_y2-(STARTOF $G0),GBR),R0
  MOV.B  R0,@(_x2-(STARTOF $G0),GBR)
  MOV.B  @(_y2-(STARTOF $G0)+1,GBR),R0
  MOV.B  R0,@(_x2-(STARTOF $G0)+1,GBR)
  MOV.W  @(_y2-(STARTOF $G0)+2,GBR),R0
  MOV.W  R0,@(_x2-(STARTOF $G0)+2,GBR)
  MOV.W  @(_y2-(STARTOF $G0)+4,GBR),R0
  MOV.W  R0,@(_x2-(STARTOF $G0)+4,GBR)
  MOV.L  @(_y2-(STARTOF $G0)+8,GBR),R0
  MOV.L  R0,@(_x2-(STARTOF $G0)+8,GBR)
  MOV.L  @(_y2-(STARTOF $G0)+12,GBR),R0
  RTS
  MOV.L  R0,@(_x2-(STARTOF $G0)+12,GBR)
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	40	26	22	25
SH-2	40	26	22	25
SH-2A	40	26	17	18
SH2A-FPU	40	26	17	18
SH-2E	40	26	22	25
SH2-DSP(SH7065)	40	26	22	25
SH-3	40	26	26	27
SH3-DSP	40	26	36	31
SH-4	40	26	18	21
SH-4A	40	26	15	13
SH4AL-DSP	40	26	15	13

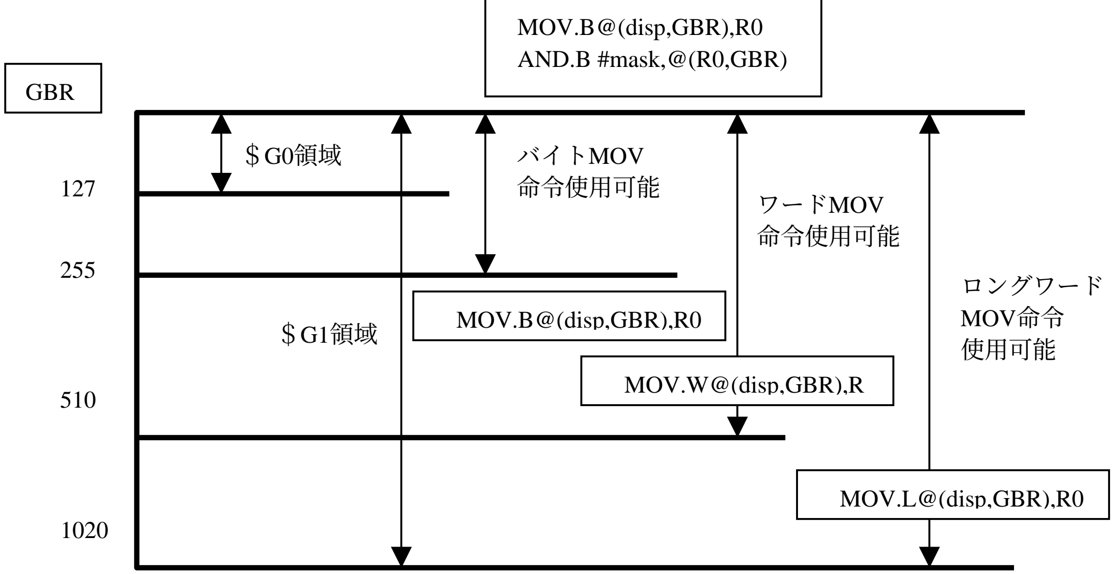
5.6.2 グローバルベースレジスタ（GBR）領域の使い分け

■ポイント

GBR0 と GBR1 の領域を使い分けることにより、性能を向上させることができます。

■説明

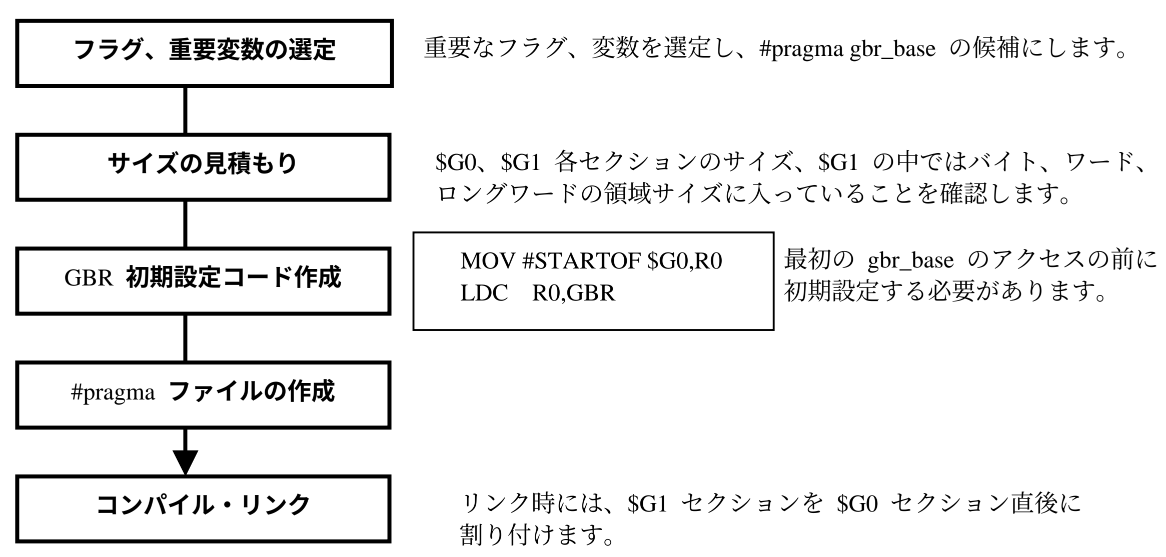
- #pragma gbr_base の領域



- GBR ベースアドレシングの特長および用途

セクション（領域）	特長	用途
$G0 (bgr_base)	バイトデータのビット処理、設定、参照の効率がよい	バイトのフラグデータ
$G1 (bgr_base1)	データの設定、参照の効率がよい	一般の変数

- GBR ベースの活用手順



RENESAS

■使用例

ビットフィールドへのアクセスを行います。

【注】コンパイルオプションは –cpu=sh2 –gbr=user の例です。

<table>
<tr><th><u>C ソース</u></th><th><u>＃pragma 指定なし</u></th><th><u>＃pragma 指定あり</u></th></tr>
<tr><td><pre>
#pragma gbr_base (bitf)

struct BitField {
    unsigned char a : 1 ;
    unsigned char b : 1 ;
    unsigned char c : 1 ;
    unsigned char d : 1 ;
    unsigned char e : 1 ;
    unsigned char f : 1 ;
    unsigned char g : 1 ;
    unsigned char h : 1 ;
} bitf ;

main()
{
    bitf.a = 1 ; // bit set
    bitf.b = 0 ; // bit clear

    if (bitf.c)
        bitf.d = 1 ;
    else
        bitf.e = 1 ;
}
</pre></td><td><pre>
.EXPORT   _bitf
    .EXPORT   _main
    .SECTION   P,CODE,ALIGN=4
_main: ; function: main
                       ; frame size=0
    MOV.L   L241,R4   ; _bitf
    MOV.B   @R4,R0
    OR   #128,R0
    MOV.B   R0,@R4
    MOV.B   @R4,R0
    AND   #191,R0
    MOV.B   R0,@R4
    MOV   R4,R0
    MOV.B   @R0,R0
    TST   #32,R0
    BT   L238
    MOV.B   @R4,R0
    BRA   L240
    OR   #16,R0
L238:
    MOV.B   @R4,R0
    OR   #8,R0
L240:
    RTS
    MOV.B   R0,@R4
L241:
    .DATA.L   _bitf
    .SECTION   B,DATA,ALIGN=4
_bitf: ; static: bitf
    .RES.B   1
    .END
</pre></td><td><pre>
.EXPORT   _bitf
    .EXPORT   _main
    .SECTION   ,CODE,ALIGN=4
_main: ;function:main
           ; frame size=0
   MOV#_bitf-(STARTOF$G0),R0
   OR.B   #128,@(R0,GBR)
   AND.B   #191,@(R0,GBR)
   TST.B   #32,@(R0,GBR)
    BT   L238
    BRA   L239
   OR.B   #16,@(R0,GBR)
   L238:
   OR.B   #8,@(R0,GBR)
   L239:
    RTS
    NOP
    .SECTION   G0,DATA,ALIGN=4
_bitf: ; static: bitf
    .DATAB.B   1,0
    .END
</pre></td></tr>
</table>

RENESAS

5.7 レジスタ退避／回復の制御

■ポイント

レジスタの退避／回復方法を工夫することにより実行速度が向上できます。（Ver.8）

■説明

末端の関数の出入口で行われるレジスタ変数用レジスタの退避／回復を削除することで実行速度および ROM 効率の向上が図れます。ただし、以下のいずれかの処理が必要となり逆効果になる場合があるので、適用する箇所はよく検討してください。

（1） レジスタの退避／回復を削除した関数の呼び出し元の関数でレジスタ変数用レジスタの退避／回復をする。

（2） 関数呼び出しを超えてレジスタ変数用レジスタを割り付けないオブジェクトにする。

■使用例

スタックの退避／回復を関数 table で一括して行わせます。

改善前ソースコード

```
#define LISTMAX 2
typedef
  int ARRAY[LISTMAX][LISTMAX][LISTMAX];
ARRAY ary1, ary2, ary3;
void init(int, ARRAY);
void copy(ARRAY, ARRAY);
void sum(ARRAY, ARRAY, ARRAY);
void table (void)
{
    init(74755, ary1);
    copy(ary1, ary2);
    sum(ary1, ary2, ary3);
}

void init (int seed, ARRAY p)
{
    int    i, j, k;

    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
            for ( k = 0; k < LISTMAX; k++ ){
                seed = ( seed * 1309 ) & 16383;
                p[i][j][k] = seed;
            }
}

void copy (ARRAY p, ARRAY q)
{
    int    i, j, k;

    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
            for ( k = 0; k < LISTMAX; k++ )
                q[k][i][j] = p[i][j][k];
}

void sum (ARRAY p, ARRAY q, ARRAY r)
{
    int    i, j, k;

    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
            for ( k = 0; k < LISTMAX; k++ )
                r[i][j][k] = p[i][j][k] +
                          q[i][j][j];
}
```

改善後ソースコード

```
#pragma regsave (table)
#pragma noregalloc (table)
#pragma noregsave (init, copy, sum)
#define LISTMAX 2
typedef
  int ARRAY[LISTMAX][LISTMAX][LISTMAX];
ARRAY  ary1, ary2, ary3;
void init(int, ARRAY);
void copy(ARRAY, ARRAY);
void sum(ARRAY, ARRAY, ARRAY);
void table (void)
{
    init(74755, ary1);
    copy(ary1, ary2);
    sum(ary1, ary2, ary3);
}

void init (int seed, ARRAY p)
{
    int    i, j, k;

    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
            for ( k = 0; k < LISTMAX; k++ ){
                seed = ( seed * 1309 ) & 16383;
                p[i][j][k] = seed;
            }
}

void copy (ARRAY p, ARRAY q)
{
    int    i, j, k;

    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
            for ( k = 0; k < LISTMAX; k++ )
                q[k][i][j] = p[i][j][k];
}

void sum (ARRAY p, ARRAY q, ARRAY r)
{
    int    i, j, k;

    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
             for ( k = 0; k < LISTMAX; k++ )
                r[i][j][k] = p[i][j][k] +
                                   q[i][j][j];
}
```

改善前アセンブリ展開コード

```
_table:
        MOV.L      R14,@-R15
        STS.L      PR,@-R15
        MOV.L      L270+6,R14
        MOV.L      L270+10,R4
        BSR        _init
        MOV        R14,R5
        MOV.L      L270+14,R5
        BSR        _copy
        MOV        R14,R4
        MOV        R14,R4
        LDS.L      @R15+,PR
        MOV.L      L270+18,R6
        MOV.L      L270+14,R5
        BRA        _sum
        MOV.L      @R15+,R14
_init:
        MOV        #2,R6
        MOV.L      R13,@-R15
        MOV        #0,R13
        MOV.L      R12,@-R15
        MOV        R5,R12
        MOV.L      R10,@-R15
        MOV.L      R9,@-R15
        MOV.L      R8,@-R15
        MOV        R5,R8
        STS.L      MACL,@-R15
        ADD        #32,R8
        MOV.W      L270,R9
        MOV.W      L270+2,R10
L261:
        MOV        R13,R1
        MOV        R12,R0
L262:
        MOV        R13,R7
        MOV        R0,R5
L263:
        MUL.L      R10,R4
        ADD        #1,R7
        STS        MACL,R3
        MOV        R3,R4
        AND        R9,R4
        CMP/GE     R6,R7
        MOV.L      R4,@R5
        BF/S       L263
        ADD        #4,R5
        ADD        #1,R1
        CMP/GE     R6,R1
        BF/S       L262
        ADD        #8,R0
        ADD        #16,R12
        CMP/HS     R8,R12
        BF         L261
        LDS.L      @R15+,MACL
        MOV.L      @R15+,R8
        MOV.L      @R15+,R9
        MOV.L      @R15+,R10
        MOV.L      @R15+,R12
        RTS
        MOV.L      @R15+,R13
_copy:
        MOV.L      R14,@-R15
        MOV.L      R13,@-R15
        MOV        #2,R13
        MOV.L      R11,@-R15
        MOV.L      R10,@-R15
        MOV.L      R9,@-R15
        MOV        #0,R9
        MOV.L      R8,@-R15
        MOV        R9,R14
        ADD        #-4,R15
```

改善後アセンブリ展開コード

```
_table:
        MOV.L      R14,@-R15
        MOV.L      R13,@-R15
        MOV.L      R12,@-R15
        MOV.L      R11,@-R15
        MOV.L      R10,@-R15
        MOV.L      R9,@-R15
        MOV.L      R8,@-R15
        FMOV.S     FR15,@-R15
        FMOV.S     FR14,@-R15
        FMOV.S     FR13,@-R15
        FMOV.S     FR12,@-R15
        STS.L      PR,@-R15
        MOV.L      L270+10,R4
        MOV.L      L270+6,R5
        STS.L      MACH,@-R15
        STS.L      MACL,@-R15
        BSR        _init
        NOP
        MOV.L      L270+6,R4
        MOV.L      L270+14,R5test3
        BSR        _copy
        NOP
        MOV.L      L270+6,R4
        MOV.L      L270+14,R5test3
        MOV.L      L270+18,R6
        BSR        _sum
        NOP
        LDS.L      @R15+,MACL
        LDS.L      @R15+,MACH
        LDS.L      @R15+,PR
        FMOV.S     @R15+,FR12
        FMOV.S     @R15+,FR13
        FMOV.S     @R15+,FR14
        FMOV.S     @R15+,FR15
        MOV.L      @R15+,R8
        MOV.L      @R15+,R9
        MOV.L      @R15+,R10
        MOV.L      @R15+,R11
        MOV.L      @R15+,R12
        MOV.L      @R15+,R13
        RTS
        MOV.L      @R15+,R14
_init:
        MOV.W      L270+2,R10
        MOV        R5,R8
        MOV.W      L270,R9
        ADD        #32,R8
        MOV        #2,R6
        MOV        R5,R12
        MOV        #0,R13
L261:
        MOV        R12,R0
        MOV        R13,R1
L262:
        MOV        R0,R5
        MOV        R13,R7
L263:
        MUL.L      R10,R4
        ADD        #1,R7
        CMP/GE     R6,R7
        STS        MACL,R3
        MOV        R3,R4
        AND        R9,R4
        MOV.L      R4,@R5
        BF/S       L263
        ADD        #4,R5
        ADD        #1,R1
        CMP/GE     R6,R1
        BF/S       L262
        ADD        #8,R0
```

```
        MOV         R5,R8
        ADD         #32,R8
L264:
        MOV         R9,R7
        MOV         R14,R10
        SHLL2       R10
        SHLL        R10
        MOV         R14,R3
        SHLL2       R3
        SHLL2       R3
        ADD         R4,R3
        MOV.L       R3,@R15
L265:
        MOV.L       @R15,R3
        MOV         R5,R6
        MOV         R7,R11
        SHLL2       R11
        SHLL        R11
        ADD         R3,R11
        MOV         R7,R1
        SHLL2       R1
L266:
        MOV.L       @R11+,R3
        MOV         R6,R0
        ADD         R10,R0
        ADD         #16,R6
        CMP/HS      R8,R6
        BF/S        L266
        MOV.L       R3,@(R0,R1)
        ADD         #1,R7
        CMP/GE      R13,R7
        BF          L265
        ADD         #1,R14
        CMP/GE      R13,R14
        BF          L264
        ADD         #4,R15
        MOV.L       @R15+,R8
        MOV.L       @R15+,R9
        MOV.L       @R15+,R10
        MOV.L       @R15+,R11
        MOV.L       @R15+,R13
        RTS
        MOV.L       @R15+,R14
L270:
        .DATA.W     H'3FFF
        .DATA.W     H'051D
        .DATA.W     0
        .DATA.L     _ary1
        .DATA.L     H'00012403
        .DATA.L     _ary2
        .DATA.L     _ary3
_sum:
        MOV.L       R14,@-R15
        MOV.L       R13,@-R15
        MOV.L       R12,@-R15
        MOV         #0,R12
        MOV.L       R11,@-R15
        MOV         #2,R11
        MOV.L       R10,@-R15
        MOV.L       R9,@-R15
        MOV.L       R8,@-R15
        ADD         #-4,R15
        MOV         R12,R8
L267:
        MOV         R12,R10
        MOV         R8,R13
        SHLL2       R13
        SHLL2       R13
L268:
        MOV         R12,R9
        MOV         R12,R7
        MOV         R10,R14
        SHLL2       R14
```

```
        ADD         #16,R12
        CMP/HS      R8,R12
        BF          L261
        RTS
        NOP
_copy:
        ADD         #-4,R15
        MOV         R5,R8
        MOV         #0,R9
        ADD         #32,R8
        MOV         R9,R14
        MOV         #2,R13
L264:
        MOV         R14,R3
        SHLL2       R3
        SHLL2       R3
        MOV         R14,R10
        ADD         R4,R3
        MOV         R9,R7
        SHLL2       R10
        MOV.L       R3,@R15
        SHLL        R10
L265:
        MOV         R7,R11
        MOV.L       @R15,R3
        SHLL2       R11
        MOV         R7,R1
        SHLL        R11
        MOV         R5,R6
        SHLL2       R1
        ADD         R3,R11
L266:
        MOV         R6,R0
        ADD         #16,R6
        MOV.L       @R11+,R3
        CMP/HS      R8,R6
        ADD         R10,R0
        BF/S        L266
        MOV.L       R3,@(R0,R1)
        ADD         #1,R7
        CMP/GE      R13,R7
        BF          L265
        ADD         #1,R14
        CMP/GE      R13,R14
        BF          L264
        RTS
        ADD         #4,R15
_sum:
        ADD         #-4,R15
        MOV         #0,R12
        MOV         R12,R8
        MOV         #2,R11
L267:
        MOV         R8,R13
        SHLL2       R13
        SHLL2       R13
        MOV         R12,R10
L268:
        MOV         R10,R14
        MOV         R10,R3
        SHLL2       R3
        MOV         R12,R9
        SHLL2       R14
        MOV.L       R3,@R15
        SHLL        R14
        MOV         R12,R7
L269:
        MOV         R13,R0
        ADD         R6,R0
        ADD         R14,R0
        MOV         R13,R2
        ADD         R7,R0
        MOV         R13,R3
```


RENESAS

```
        SHLL        R14                   ADD         R4,R2
        MOV         R10,R3                MOV.L       R0,@-R15
        SHLL2       R3                    ADD         R14,R2
        MOV.L       R3,@R15               MOV.L       @(4,R15),R0
L269:                                     ADD         R5,R3
        MOV         R13,R0                ADD         R7,R2
        ADD         R6,R0                 ADD         R14,R3
        ADD         R14,R0                MOV.L       @R2,R1
        ADD         R7,R0                 MOV.L       @(R0,R3),R3
        MOV         R13,R3                ADD         #1,R9
        MOV.L       R0,@-R15              MOV.L       @R15+,R2
        MOV         R13,R2                CMP/GE      R11,R9
        MOV.L       @(4,R15),R0           ADD         R1,R3
        ADD         #1,R9                 MOV.L       R3,@R2
        ADD         R5,R3                 BF/S        L269
        ADD         R14,R3                ADD         #4,R7
        MOV.L       @(R0,R3),R3           ADD         #1,R10
        ADD         R4,R2                 CMP/GE      R11,R10
        ADD         R14,R2                BF          L268
        ADD         R7,R2                 ADD         #1,R8
        MOV.L       @R2,R1                CMP/GE      R11,R8
        CMP/GE      R11,R9                BF          L267
        MOV.L       @R15+,R2              RTS
        ADD         R1,R3                 ADD         #4,R15
        MOV.L       R3,@R2              L270:
        BF/S        L269                  .DATA.W     H'3FFF
        ADD         #4,R7                 .DATA.W     H'051D
        ADD         #1,R10                .DATA.W     0
        CMP/GE      R11,R10               .DATA.L     _ary1
        BF          L268                  .DATA.L     H'00012403
        ADD         #1,R8                 .DATA.L     _ary2
        CMP/GE      R11,R8                .DATA.L     _ary3
        BF          L267
        ADD         #4,R15
        MOV.L       @R15+,R8
        MOV.L       @R15+,R9
        MOV.L       @R15+,R10
        MOV.L       @R15+,R11
        MOV.L       @R15+,R12
        MOV.L       @R15+,R13
        RTS
        MOV.L       @R15+,R14
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	292	288	684	669
SH-2	238	242	446	426
SH-2E	238	258	446	438
SH2-DSP(SH7065)	236	252	490	470
SH-3	238	242	476	458
SH3-DSP	236	252	489	487
SH-4	238	258	301	313

5.8 2バイトアドレスの指定

■ポイント

変数および関数のアドレスを2バイトで表現することによりROM効率の向上ができます。

■説明

変数または関数が2バイトで表現できるアドレスに配置されている場合、参照する側のコードを2バイトにすることによりコードサイズを縮小できます。

■使用例

変数xの値が1のとき、外部関数gを呼び出します。

改善前ソースコード

```
extern int  x;
extern void g(void);

void f (void)
{
  if (x == 1)
      g();
}
```

改善前アセンブリ展開コード

```
_f:
        MOV.L     L218+2,R3
        MOV.L     @R3,R0
        CMP/EQ    #1,R0
        BF        L219
        MOV.L     L218+6,R2
        JMP       @R2
        NOP
L219:
        RTS
        NOP
L218:
        .DATA.W    0
        .DATA.L    _x
        .DATA.L    _g
```

改善後ソースコード

```
#pragma abs16(x,g)
extern int  x;
extern void g(void);

void f (void)
{
  if (x == 1)
      g();
}
```

改善後アセンブリ展開コード

```
_f:
        MOV.W     L238+2,R3
        MOV.L     @R3,R0
        CMP/EQ    #1,R0
        BF        L239
        MOV.W     L238,R2
        JMP       @R2
        NOP
L239:
        RTS
        NOP
L238:
        .DATA.W    _g
        .DATA.W    _x
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-1	28	28	15	11
SH-2	28	28	15	11
SH-2A	24	24	13	11
SH2A-FPU	24	24	13	11
SH-2E	28	28	15	11
SH2-DSP(SH7065)	28	28	16	12
SH-3	28	28	15	11
SH3-DSP	28	28	17	12
SH-4	28	28	13	10
SH-4A	28	28	9	6
SH4AL-FPU	28	28	9	6

【注】 x=1,関数 g が void g( ){ }のとき

5.9 キャッシュの利用

キャッシュを有効に利用することによって性能向上が可能です。

5.9.1 プリフェッチ命令

■ポイント

配列変数をアクセスするとき、使用に先立ってプリフェッチ命令を実行すると、実行速度の向上が期待できます。（SH-2A、SH2A-FPU、SH-3、SH3-DSP、SH-4、SH-4A、SH4AL-DSP のみ有効）

■説明

ループで配列を順次アクセスする場合、配列のメンバ参照に先立ちプリフェッチを行うことで、実行速度が向上します。また、ループを展開することで、さらに効果的にプリフェッチが行えます。

なお、プリフェッチ命令は、連続して実行しても速度の向上は期待できませんので、前のプリフェッチ命令が完了するように十分に離して実行してください。

■使用例

配列 data の要素 a、b、c を使用して演算した結果を要素 d に格納します（SH-4）。

【注】コンパイルオプションは –cpu=sh4 –fpu=single の例です。

改善前ソースコード

```
typedef struct {
    float a, b, c, d;
} data_t;

data_t data[2048];

int f( void )
{
    data_t *p1, *p2;
    data_t *end = &data[2048];
    float a1, b1, c1, t11, t12;
    float a2, b2, c2, t21, t22;

    for( p1=data, p2=data+1; p1<end;
                         p1+=2, p2+=2 ){
        a1 = p1->a;
        b1 = p1->b;
        t11 = a1 * a1;
        t12 = b1 * b1;
        t11 += t12;
        c1 = 1/t11;
        p1->c = c1;
        a1 += b1;
        a1 += c1;
        p1->d = a1;

        a2 = p2->a;
        b2 = p2->b;
        t21 = a2 * a2;
        t22 = b2 * b2;
        t21 += t22;
        c2 = 1/t21;
        p2->c = c2;
        a2 += b2;
        a2 += c2;
        p2->d = a2;
    }
}
```

改善後ソースコード

```
#include <machine.h>

typedef struct {
    float a, b, c, d;
} data_t;

data_t data[2048];

int f( void )
{
    data_t *p1, *p2;
    data_t *end = &data[2048];
    float a1, b1, c1, t11, t12;
    float a2, b2, c2, t21, t22;
    data_t *next = data+4;

    for( p1=data, p2=data+1; p1<end;
                         p1+=2, p2+=2 ){
        prefetch(next);
        next += 2;

        a1 = p1->a;
        b1 = p1->b;
        t11 = a1 * a1;
        t12 = b1 * b1;
        t11 += t12;
        c1 = 1/t11;
        p1->c = c1;
        a1 += b1;
        a1 += c1;
        p1->d = a1;

        a2 = p2->a;
        b2 = p2->b;
        t21 = a2 * a2;
        t22 = b2 * b2;
        t21 += t22;
        c2 = 1/t21;
        p2->c = c2;
        a2 += b2;
        a2 += c2;
        p2->d = a2;
    }
}
```

改善前アセンブリ展開コード

```
_f:
        MOV.L       L252+6,R6
        MOV.L       L252+2,R7
```

改善後アセンブリ展開コード

```
_f:
        MOV.L       L253+6,R7
        MOV.L       L253+2,R0
```

```
        MOV         R7,R5
        MOV         R7,R4
        ADD         R7,R6
        CMP/HS      R6,R4
        ADD         #16,R5
        BT/S        L250
        FLDI1       FR5
L251:
        MOV         #4,R0
        FMOV.S      @R4,FR4
        FMOV.S      @(R0,R4),FR6
        MOV         #8,R0
        FMOV.S      FR4,FR7
        FMUL        FR4,FR7
        FMOV.S      FR6,FR8
        FMUL        FR6,FR8
        FADD        FR6,FR4
        FADD        FR8,FR7
        FMOV.S      FR7,FR3
        FMOV.S      FR5,FR7
        FDIV        FR3,FR7
        FADD        FR7,FR4
        FMOV.S      FR7,@(R0,R4)
        MOV         #12,R0
        FMOV.S      FR4,@(R0,R4)
        MOV         #4,R0
        FMOV.S      @(R0,R5),FR6
        MOV         #8,R0
        FMOV.S      @R5,FR4
        FMOV.S      FR6,FR8
        FMUL        FR6,FR8
        FMOV.S      FR4,FR7
        FMUL        FR4,FR7
        FADD        FR6,FR4
        FADD        FR8,FR7
        FMOV.S      FR7,FR3
        FMOV.S      FR5,FR7
        FDIV        FR3,FR7
        FADD        FR7,FR4
        FMOV.S      FR7,@(R0,R5)
        ADD         #32,R4
        MOV         #12,R0
        CMP/HS      R6,R4
        FMOV.S      FR4,@(R0,R5)
        BF/S        L251
        ADD         #32,R5
L250:
        RTS
        NOP
L252:
        .DATA.W     0
        .DATA.L     _data
        .DATA.L     H'00008000
```

```
        MOV         R0,R4
        MOV         R0,R5
        ADD         R0,R7
        MOV         R0,R6
        CMP/HS      R7,R4
        ADD         #16,R5
        ADD         #64,R6
        BT/S        L251
        FLDI1       FR5
L252:
        PREF        @R6
        MOV         #4,R0
        FMOV.S      @R4,FR4
        FMOV.S      @(R0,R4),FR6
        MOV         #8,R0
        FMOV.S      FR4,FR7
        FMUL        FR4,FR7
        FMOV.S      FR6,FR8
        FMUL        FR6,FR8
        FADD        FR6,FR4
        ADD         #32,R6
        FADD        FR8,FR7
        FMOV.S      FR7,FR3
        FMOV.S      FR5,FR7
        FDIV        FR3,FR7
        FADD        FR7,FR4
        FMOV.S      FR7,@(R0,R4)
        MOV         #12,R0
        FMOV.S      FR4,@(R0,R4)
        MOV         #4,R0
        FMOV.S      @(R0,R5),FR6
        MOV         #8,R0
        FMOV.S      @R5,FR4
        FMOV.S      FR6,FR8
        FMUL        FR6,FR8
        FMOV.S      FR4,FR7
        FMUL        FR4,FR7
        FADD        FR6,FR4
        FADD        FR8,FR7
        FMOV.S      FR7,FR3
        FMOV.S      FR5,FR7
        FDIV        FR3,FR7
        FMOV.S      FR7,@(R0,R5)
        FADD        FR7,FR4
        ADD         #32,R4
        MOV         #12,R0
        CMP/HS      R7,R4
        FMOV.S      FR4,@(R0,R5)
        BF/S        L252
        ADD         #32,R5
L251:
        RTS
        NOP
L253:
        .DATA.W     0
        .DATA.L     _data
        .DATA.L     H'00008000
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-3	18	24	609	31
SH3-DSP	18	16	609	21
SH-4	16	12	608	10

【注】 (1) SH-3、SH3-DSP、および SH-4 はプログラムを外部メモリへロード

(2) SH-3、SH3-DSP は外部メモリへのアクセスサイクル数を 16 にて測定

(3) SH-4 はメモリアクセスの待ちサイクル数を 15 にて測定

(4) キャッシュミスを考慮

5.9.2 タイリング

■ポイント

データアクセスに局所性を持たせてデータキャッシュミスを少なくすようなプログラミングを行います。
言い換えれば、キャッシュがヒットしている状態で計算できるものは、先にしてしまうテクニックです。

■説明

簡単な例として、2 つの配列、A、B に対する差分の総和をとる配列を作成する例を示します。
そこで、アクセスの順番を変えてプログラミングすることによって、データキャッシュミスを削減するようなプログラミングをします。

■使用例

構造体は配列のメンバ、a,b,c,d に対し、

di= Σj bj-aj

の計算を行う。

改善前ソースコード

```
typedef struct {
    float a,b,c,d;
} data_t;

f(data_t data[], int n)
{
    data_t *p,*q;
    data_t *p_end = &data[n];
    data_t *q_end = p_end;
    float a,d;

    for (p = data; p < p_end; p++){
        a = p->a;
        d = 0.0f;
        for (q = data; q < q_end; q++){
            d += q->b -a;
        }
        p->d=d;
    }
}
```

改善後ソースコード

```
#define STRIDE 512
typedef struct {
    float a,b,c,d;
} data_t;

f(data_t data[], int n)
{
    data_t *p,*q, *end=&data[n];
    data_t *pp, *qq;
    data_t *pp_end, *qq_end;
    float a,d;


    for (p = data; p < end; p = pp_end){
        pp_end = p + STRIDE;
        for (q = data; q < end; q = qq_end){
            qq_end = q + STRIDE;
            for (pp = p; pp < pp_end && pp
                                  <end; pp++){
                a = pp->a;
                d = pp->d;
                for (qq = q; qq < qq_end
                        && qq < end; qq++){
                    d += qq->b -a;
                }
                p->d = d;
            }
        }
    }
}
```

改善前アセンブリ展開コード

```
_f:
        MOV         R5,R1
        SHLL2       R1
        SHLL2       R1
        FLDI0       FR6
        ADD         R4,R1
        BRA         L244
        MOV         R4,R6
L245:
        MOV         R4,R5
        FMOV.S      @R6,FR5
        CMP/HS      R1,R5
        BT/S        L246
        FMOV.S      FR6,FR4
L247:
        STS         FPSCR,R3
        MOV.L       L248,R2
        MOV         #4,R0
```

改善後アセンブリ展開コード

```
_f:
        MOV.L       R14,@-R15
        MOV         R5,R7
        MOV.L       R13,@-R15
        SHLL2       R7
        MOV.L       R11,@-R15
        SHLL2       R7
        MOV.L       R10,@-R15
        ADD         R4,R7
        MOV.W       L259,R11
        BRA         L249
        MOV         R4,R13
L250:
        MOV         R13,R10
        ADD         R11,R10
        BRA         L251
        MOV         R4,R14
L252:
```

```
        FMOV.S     @(R0,R5),FR3
        ADD        #16,R5
        AND        R2,R3
        CMP/HS     R1,R5
        LDS        R3,FPSCR
        FSUB       FR5,FR3
        BF/S       L247
        FADD       FR3,FR4
L246:
        MOV        #12,R0
        FMOV.S     FR4,@(R0,R6)
        ADD        #16,R6
L244:
        CMP/HS     R1,R6
        BF         L245
        RTS
        NOP
L248:
        .DATA.L    H'FFE7FFFF
        .END
```

```
        MOV        R14,R1
        ADD        R11,R1
        BRA        L253
        MOV        R13,R6
L254:
        MOV        #12,R0
        FMOV.S     @R6,FR5
        FMOV.S     @(R0,R6),FR4
        BRA        L255
        MOV        R14,R5
L256:
        STS        FPSCR,R3
        MOV.L      L259+2,R2
        MOV        #4,R0
        FMOV.S     @(R0,R5),FR3
        ADD        #16,R5
        AND        R2,R3
        LDS        R3,FPSCR
        FSUB       FR5,FR3
        FADD       FR3,FR4
L255:
        CMP/HS     R1,R5
        BT         L257
        CMP/HS     R7,R5
        BF         L256
L257:
        MOV        #12,R0
        ADD        #16,R6
        FMOV.S     FR4,@(R0,R13)
L253:
        CMP/HS     R10,R6
        BT         L258
        CMP/HS     R7,R6
        BF         L254
L258:
        MOV        R1,R14
L251:
        CMP/HS     R7,R14
        BF         L252
        MOV        R10,R13
L249:
        CMP/HS     R7,R13
        BF         L250
        MOV.L      @R15+,R10
        MOV.L      @R15+,R11
        MOV.L      @R15+,R13
        RTS
        MOV.L      @R15+,R14
L259:
        .DATA.W    H'2000
        .DATA.L    H'FFE7FFFF
        .END
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [cycle]	
	改善前	改善後	改善前	改善後
SH-3	76	132	931×10^3	725×10^3
SH3-DSP	76	132	940×10^3	697×10^3
SH-4	52	104	315×10^3	43×10^3

【注】(1)n=4096、STRIDE=512 の場合
(2)キャッシュミスを考慮

■改善前後のプログラム解説

改善前と後では、改善後の方がループが 4 重になったため、処理が複雑になり、コードサイズも大きくなります。ただし、このような処理をすることで、キャッシュミスによるオーバヘッドを削減することが可能です。よって、処理するデータが小さな場合には効果はありませんが、データが大きい場合に効果があります。

改善前では、1 つのデータ data[0]->d を求めるために、data[0]~data[n-1]までを順次参照します。
次に、data[1]->d を求めるために、再度、data[0]~data[n-1]を参照しようとしますが、配列 data のサイズが、キャッシュのサイズに比べ大きいときには、すでに、data[0]の内容はキャッシュにはなくなっており、キャッシュミスを起こします。
大きな領域を順次参照していくため、同じデータの次の参照まで、キャッシュ中にデータが残っていないことになります。
改善後のプログラムでは、小さな区間に分割してデータをアクセスするため、そのデータアクセスの間のキャッシュミスは少なくなります。計算の順番を変えて、キャッシュがヒットしている間に、別の計算もしてしまう手法です。

5.10 マトリックス演算

■ポイント

行列演算の際、組み込み関数を使用すると、実行速度の向上が期待できます。
その際、乗数となる配列はあらかじめ、浮動小数点拡張レジスタに格納しておく必要があります。

■説明

4 行 4 列の配列の積は通常ならば、ループを用いて順次演算を行うため処理が複雑になり実行速度の向上は期待できませんが、SH-4 ではマトリックス演算を組み込み関数でサポートしているため、この関数を使用することにより実行速度の大幅な向上が期待できます。

■使用例

配列 data と配列 tbl の積を配列 ret に格納します。
【注】コンパイルオプションは –cpu=sh4 –fpu=single の例です。

改善前ソースコード

```
void mtrx4mul1 (float data[4][4],
  float tbl[4][4], float ret[4][4])
{
  int i,j,k;

  for(i=0;i<4;i++){
    for(j=0;j<4;j++){
      for(k=0;k<4;k++){
        ret[i][j]+=
          data[i][k]*tbl[k][j];
      }
    }
  }
}
```

改善前アセンブリ展開コード

```
_mtrx4mull:
          MOV.L      R14,@-R15
          MOV.L      R13,@-R15
          MOV.L      R11,@-R15
          MOV.L      R10,@-R15
          MOV.L      R9,@-R15
          MOV.L      R8,@-R15
          ADD        #-4,R15
          MOV        #0,R8
          MOV.L      R8,@R15
          MOV        #4,R14
L244:
          MOV.L      @R15,R11
          MOV        R8,R9
          SHLL2      R11
          SHLL2      R11
L245:
          MOV        R9,R1
          MOV        #0,R7
          SHLL2      R1
          MOV        R8,R10
          MOV        #0,R13
          ADD        R5,R7
L246:
          MOV        R11,R0
          ADD        R6,R0
          ADD        R1,R0
          MOV        R11,R3
          MOV.L      R0,@-R15
          ADD        R4,R3
          MOV.L      @R15+,R2
          MOV        R1,R0
          ADD        R13,R3
```

改善後ソースコード

```
#include <machine.h>
void _mtrx4mul (float data[4][4],
  float tbl[4][4],float ret[4][4])
{
   ld_ext(tbl);
   mtrx4mul(data,ret);
}
```

改善後アセンブリ展開コード

```
__mtrx4mul:
    ADD        #-12,R15
    MOV.L      R4,@(8,R15)
    MOV.L      R5,@(4,R15)
    MOV.L      R6,@R15
    MOV.L      @(8,R15),R2
    FRCHG
    FMOV.S     @R2+,FR0
    FMOV.S     @R2+,FR1
    FMOV.S     @R2+,FR2
    FMOV.S     @R2+,FR3
    FMOV.S     @R2+,FR4
    FMOV.S     @R2+,FR5
    FMOV.S     @R2+,FR6
    FMOV.S     @R2+,FR7
    FMOV.S     @R2+,FR8
    FMOV.S     @R2+,FR9
    FMOV.S     @R2+,FR10
    FMOV.S     @R2+,FR11
    FMOV.S     @R2+,FR12
    FMOV.S     @R2+,FR13
    FMOV.S     @R2+,FR14
    FMOV.S     @R2+,FR15
    FRCHG
    MOV.L      @(4,R15),R3
    MOV.L      @R15,R1
    FMOV.S     @R3+,FR0
    FMOV.S     @R3+,FR1
    FMOV.S     @R3+,FR2
    FMOV.S     @R3+,FR3
    FTRV       XMTRX,FV0
    ADD        #16,R1
    FMOV.S     FR3,@-R1
```


RENESAS

```
FMOV.S     @(R0,R7),FR0        FMOV.S     FR2,@-R1
FMOV.S     @R2,FR2             FMOV.S     FR1,@-R1
ADD        #1,R10              FMOV.S     FR0,@-R1
FMOV.S     @R3,FR3             FMOV.S     @R3+,FR0
CMP/GE     R14,R10             FMOV.S     @R3+,FR1
ADD        #16,R7              FMOV.S     @R3+,FR2
FMAC       FR0,FR3,FR2         FMOV.S     @R3+,FR3
FMOV.S     FR2,@R2             FTRV       XMTRX,FV0
BF/S       L246                ADD        #32,R1
ADD        #4,R13              FMOV.S     FR3,@-R1
ADD        #1,R9               FMOV.S     FR2,@-R1
CMP/GE     R14,R9              FMOV.S     FR1,@-R1
BF         L245                FMOV.S     FR0,@-R1
MOV.L      @R15,R3             FMOV.S     @R3+,FR0
ADD        #1,R3               FMOV.S     @R3+,FR1
CMP/GE     R14,R3              FMOV.S     @R3+,FR2
BF/S       L244                FMOV.S     @R3+,FR3
MOV.L      R3,@R15             FTRV       XMTRX,FV0
ADD        #4,R15              ADD        #32,R1
MOV.L      @R15+,R8            FMOV.S     FR3,@-R1
MOV.L      @R15+,R9            FMOV.S     FR2,@-R1
MOV.L      @R15+,R10           FMOV.S     FR1,@-R1
MOV.L      @R15+,R11           FMOV.S     FR0,@-R1
MOV.L      @R15+,R13           FMOV.S     @R3+,FR0
RTS                            FMOV.S     @R3+,FR1
MOV.L      @R15+,R14           FMOV.S     @R3+,FR2
                               FMOV.S     @R3+,FR3
                               FTRV       XMTRX,FV0
                               ADD        #32,R1
                               FMOV.S     FR3,@-R1
                               FMOV.S     FR2,@-R1
                               FMOV.S     FR1,@-R1
                               FMOV.S     FR0,@-R1
                               ADD        #12,R15
                               RTS
                               NOP
```

■改善前後のコードサイズと実行速度

CPU 種別	**コードサイズ** [byte]		**実行速度** [cycle]	
	改善前	**改善後**	**改善前**	**改善後**
SH-4	110	118	603	113

5.11 ソフトパイプ

■ポイント

演算の結果待ちをなくすようなプログラミングをすることで、パイプラインの流れをスムーズにします。

■説明

ソフトパイプは、データフロー（値の定義と使用）に伴う命令の発行待ちを解消するプログラミングです。たとえば、総和をとるようなプログラミングでは、ロード命令による定義の直後に ADD による加算がある場合、待ちが生じます。よって、早めにロード命令を発行しておけばこの待ちは解消できます。ループ中の処理であれば次回のデータのためのロードを、今回の繰り返しで行うという手法です。

顕著な例が、除算、平方根演算です。SH-4 には、FDIV,FSQRT の命令がありますがレイテンシ（命令発行から結果生成までのサイクル）が大きいため（SH-4 で 12 サイクル）結果を即使用するようなプログラムでは、次の命令実行までに待ちサイクルが出ます。

■使用例

平方根の総和をとるループの例

【注】コンパイルオプションは –cpu=sh4 –fpu=single の例です。

改善前ソースコード

```
#include <mathf.h>

float func1(float *p, int cnt){
   float ret=0.0f;
   do {
      ret+=sqrtf(*p++);
      x();

   } while(cnt--);
   return ret;
 }
```

改善前アセンブリ展開コード

```
_func1:
         MOV.L      R14,@-R15
         FMOV.S     FR15,@-R15
         STS.L      PR,@-R15
         ADD        #-8,R15
         MOV.L      L262,R14
         FLDI0      FR15
         MOV.L      R4,@(4,R15)
         MOV.L      R5,@R15
L260:
         MOV.L      @(4,R15),R3
         ADD        #4,R3
         MOV.L      R3,@(4,R15)
         ADD        #-4,R3
         FMOV.S     @R3,FR3
         FSQRT      FR3
         JSR        @R14
         FADD       FR3,FR15
         MOV.L      @R15,R3
         ADD        #-1,R3
         MOV.L      R3,@R15
         ADD        #1,R3
         TST        R3,R3
         BF         L260
         FMOV.S     FR15,FR0
         ADD        #8,R15
         LDS.L      @R15+,PR
         FMOV.S     @R15+,FR15
         RTS
         MOV.L      @R15+,R14
L262:
         .DATA.L    _x
```

改善後ソースコード

```
#include <mathf.h>

float func21(float *p, int cnt){
   float ret=0.0f;
   float sq=0.0f;
   do {
      ret+=sq;
      sq=sqrtf(*p++);
      x();
   } while (cnt--);
   ret+=sq;
   return ret;
 }
```

改善後アセンブリ展開コード

```
_func21:
         MOV.L      R14,@-R15
         FMOV.S     FR15,@-R15
         FMOV.S     FR14,@-R15
         STS.L      PR,@-R15
         ADD        #-8,R15
         FLDI0      FR4
         MOV.L      L263,R14
         FMOV.S     FR4,FR15
         MOV.L      R4,@(4,R15)
         MOV.L      R5,@R15
         FMOV.S     FR4,FR14
L261:
         MOV.L      @(4,R15),R3
         FADD       FR15,FR14
         ADD        #4,R3
         MOV.L      R3,@(4,R15)
         ADD        #-4,R3
         FMOV.S     @R3,FR15
         JSR        @R14
         FSQRT      FR15
         MOV.L      @R15,R3
         ADD        #-1,R3
         MOV.L      R3,@R15
         ADD        #1,R3
         TST        R3,R3
         BF         L261
         FMOV.S     FR14,FR0
         ADD        #8,R15
         LDS.L      @R15+,PR
         FMOV.S     @R15+,FR14
         FMOV.S     @R15+,FR15
```

```
                                   |        RTS
                                   |        MOV.L      @R15+,R14
                                   |L263:
                                   |        .DATA.L    _x
```

■改善前後プログラム解説

改善前では、FSQRT 直後に、FADD を実行するため、FSQRT 終了のサイクルまで FADD が待ちに入ります。
改善後では、FSQRT 実行後、次のループで FADD を発行するため、FADD の発行待ちをなくすことができます。

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]（１ループ）		実行速度 [cycle]（FSQRT による待ちサイクル）	
	改善前	改善後	改善前	改善後
SH-4	28	28	9	0

5.12 キャッシュメモリについて

SuperH シリーズには、キャッシュを搭載しているシリーズがあります。
キャッシュは、プログラム、データのメモリへのアクセスを減らしプログラム動作を高速化するための機構です。
キャッシュを使用することにより、プログラムは高速化しますが、キャッシュといってもさまざまなタイプのものがあり、その構造と機能を十分理解することで、さらに効果的なプログラミングが可能となります。
ここでは、SuperH シリーズに搭載されているいくつかのキャッシュの構造を説明し、キャッシュを効率的に活用するためのプログラミングについて説明します。

1．用語説明

■キャッシュヒット

CPU が外部メモリをアクセスしようとしたとき、その内容がキャッシュメモリ内にあるかどうかチェックします。キャッシュメモリ内データがある場合をキャッシュヒットといいます。

基本的に、キャッシュヒットした場合は、外部メモリをアクセスすることなく、高速なキャッシュメモリをアクセスすることになります。

■キャッシュミス

CPU が外部メモリをアクセスしようとしたとき、その内容がキャッシュメモリ内にあるかどうかチェックします。キャッシュメモリ内にない場合を、キャッシュミスといいます。

■キャッシュフィル

キャッシュミスした場合、CPU はそのメモリの内容をキャッシュに格納しようとします。これをキャッシュフィルといいます。

■キャッシュライン長

CPU がキャッシュフィルを行おうとするとき、アクセスしたメモリ内容だけをキャッシュに格納するのではなく、前後を含めたある連続した領域をまとめてキャッシュフィルしようとします。このときの領域のサイズをキャッシュのライン長といいます。ライン長は、CPU により固定の長さ（サイズ）に決まっています。キャッシュには、このライン長を基本にデータが格納されています。

■キャッシュサイズ、エントリ数（ライン数）

キャッシュに格納できるデータの容量をキャッシュサイズといいます。
エントリ数（ライン数）、キャッシュライン長とキャッシュサイズの関係は以下のようになります。

（キャッシュサイズ）＝（エントリ数）×（キャッシュライン長）

■ライトバックとライトスルー

キャッシュヒット状態で、そのメモリ内容を書き換えようとしたとき、書き換えの方式として次の 2 つの選択肢があります。

（1） キャッシュメモリの内容と、外部メモリの内容を同時に書き換える
（2） キャッシュメモリだけを書き換える

(1)の場合は、キャッシュメモリの内容と、外部メモリの内容が必ず一致しています。この方式をライトスルーといいます。

(2)の場合は、キャッシュメモリに最新のデータが残っているだけで、外部メモリは書き換えが行われないため、古いデータのままです。よって、この方式を取る場合、そのキャッシュエントリ内容が捨てられる前にエントリ内のデータを外部メモリに書き戻します。この方式をライトバックといいます。（普通はキャッシュのエントリに対し、一度でも書き込みがあったかどうかを示すフラグがあり、書き込みがあることを示した場合のみ、外部メモリに書き戻す仕組みとなっています。）

■キャッシュのコヒーレンシ

外部メモリの内容とキャッシュメモリの内容が一致していることです。

つまり、ライトバック方式でキャッシュを使用した場合には、キャッシュの内容と外部メモリの内容が一致していない可能性があり、CPU 以外のデバイスが外部メモリをアクセス使用した場合、データ内容が更新されていないために正しくソフトウェアが動作しません。他のデバイスが同じメモリをアクセスする（共有メモリを使用する）場合は、ライトスルー方式にするか、または、他のデバイスがアクセスする前に、その領域の該当キャッシュエントリをソフトウェアで書き戻す必要があります。

■ダイレクトマップ方式

キャッシュの方式の 1 つです。

基本的に、外部メモリのアドレスから、一義的に、キャッシュメモリのどのアドレスに格納されるか決まる方式です。外部メモリのオフセット値をとって、そのキャッシュ上のオフセットアドレスが同一の場所にデータが格納されます。

あるアドレスのデータが、キャッシュ上のどこに入っているか確認する際には、格納場所はアドレスから 1 個所に決まるので、この方式ではハードウェア側の負荷が軽くできます。ただし、頻繁に使用するメモリのアドレスのオフセットが一致していると、同じエントリのリプレースばかりがされてしまい、逆にほとんど使用されないようなエントリも存在することがあります。

よって、キャッシュが有効に使用されないこともあるためプログラムの配置に注意する必要があります。

■フルアソシアティブ方式

キャッシュの方式の 1 つです。

ダイレクトマップとは違い、エントリ上に、外部メモリ上のアドレスすべてとデータが格納されています。よって、最も長い時間アクセスされていないエントリから、リプレース（LRU 方式）がおこるため、最もキャッシュを有効に使用することができます。ただし、その反面、外部メモリのアドレスがどのキャッシュエントリに格納されているかを調べるためには、すべてのキャッシュエントリをチェックする必要がありハードウェアの機構が複雑になります。

■セットアソシアティブ方式

ダイレクトマップとフルアソシアティブの中間の方式で、ダイレクトマップのキャッシュが何枚かあるということです。（枚数を way 数といいます。）ダイレクトマップのように、外部メモリのアドレスのオフセット値から、すぐにキャッシュのどのエントリを使うか決まりますが、その何枚かあるキャッシュのうち、最も長い時間アクセスしていない way を使う方式です。

そのアドレスが、キャッシュのどこに格納されているか探す場合も、すべてのキャッシュエントリを探すわけではなく、way 数分だけを探すので、ハードウェア構成もそれほど複雑になりません。

ちなみに、SH7604,SH7708,SH7707,SH7709,SH7718 などは、4 way セットアソシアティブ方式を採用しています。

5.13 SuperH シリーズのキャッシュ

以下に、SuperH シリーズに使用されている各キャッシュの説明をします。

1. SH7032,SH7034,SH7020,SH7021 シリーズ(SH-1)
   本シリーズには、キャッシュは搭載されていません。本シリーズは内蔵 ROM/RAM タイプの CPU です。内蔵 ROM/RAM 上での実行であれば、キャッシュ以上の性能を出すことが可能な CPU です。

2. SH704x シリーズ(SH-2)
   本シリーズは、内蔵 ROM/RAM タイプの CPU で、SH7034 の上位 CPU という位置付けですが、内蔵 RAM の一部を命令キャッシュとして使用することができる機能を持っています。

   方式　　　　　　　：命令キャッシュ（ダイレクトマップ）
   キャッシュサイズ　：1KB （使用時は、内蔵 RAM2KB）
   キャッシュライン長：4 バイト（2 命令分）
   エントリ数　　　　：256

   キャッシュは命令キャッシュのみです。キャッシュフィルのオーバヘッドは少ないため 1KB 以内のループ処理では大きく威力を発揮します。キャッシュの有効な範囲は外部メモリで、内蔵の ROM/RAM に対しては働きません。（内蔵の ROM/RAM であれば、高速にアクセスできますから、使用する必要はありません。）
   ただし、内蔵 RAM の 4KB 中の 2KB を使用して、1KB のキャッシュとして使用するため、キャッシュ使用時は、内蔵 RAM2KB となります。
   データキャッシュはないため、キャッシュは使用せずに 4KB の RAM としてデータ用に使用し、使用頻度の高いプログラムを優先的に、内蔵 ROM を使用したほうがトータル性能は向上する場合もあります。

3. SH7604 シリーズ(SH-2)
   本シリーズは、ROM レス、キャッシュ搭載のプロセッサタイプ CPU です。

   方式　　　　　　　：4 ウェイセットアソシアティブ方式（命令、データ混在型）
   キャッシュサイズ　：4KB
   キャッシュライン長：16 バイト
   エントリ数　　　　：256（64×4）
   その他　　　　　　：ライトスルー方式

4. SH7707,SH7708,SH7709 シリーズ(SH-3)
   本シリーズは、ROM レス、キャッシュ搭載のプロセッサタイプ CPU です。

   方式　　　　　　　：4 ウェイセットアソシアティブ方式（命令、データ混在型）
   キャッシュサイズ　：8KB
   キャッシュライン長：16 バイト
   エントリ数　　　　：512（128×4）
   その他　　　　　　：ライトスルーまたはライトバック方式の選択可

5. SH7750 シリーズ(SH-4)
   方式　　　　　　　：ダイレクトマップ（命令、データ 非混在型）

■命令キャッシュ

   キャッシュサイズ　：8KB
   キャッシュライン長：32バイト
   エントリ数　　　　：256
   その他　　　　　　：4KB×2のインデックスモード可能

■データキャッシュ（オペランドキャッシュ）

   キャッシュサイズ　：16KB
   キャッシュライン長：32バイト

エントリ数　　　　　：512
その他　　　　　　　：8KBを内蔵RAMとして使用可能

■ストアキュー

32byte×2 の外部メモリへの高速転送用のストアキューが用意されています。

ストアキューは、外部メモリへの高速転送のためのバッファです。
これを使用することによって、外部メモリへの高速転送が可能になります。
データキャッシュは、キャッシュブロックができないために、キャッシュのリプレースが起こる可能性があります。
ストアキューは、キャッシュのリプレースなどによる性能低下をもたらさないで確実に高速転送するための機構です。

5.14　キャッシュ活用のテクニック

以下にキャッシュを有効に使用するためのテクニックを示します。

（1）性能が上がらない場合の対処

性能が上がらない場合、考えられる主な原因として**表5.7**に示すものがあります。
また、キャッシュを有効に活用するための手段や方法として、次のようなことが一般に言われているようです。

①　デバッガ、プロファイルツールなどの活用で、関数の依存関係、実行頻度の検証。
②　依存関係の近い場所に配置し、キャッシュミスを低減させる。
③　サイズ優先最適化により、実行頻度の高い部分の関数化。

表 5.7　性能が上がらない場合の主な原因

項目	考えられる原因	対処方法
キャッシュ上の配置アドレスのずれ	・キャッシュ上の配置アドレスがずれることによる影響や使用エントリがずれるなど、これらエントリがずれることで、キャッシュの競合関係が変わることが考えられます。	・アライメントの変更
プログラムサイズの増加	・境界調整用のダミー領域、プログラムサイズの増加などにより、プログラム自体が大きくなることで、キャッシュヒット率が低下が考えられます。	・サイズ優先最適化

以下で、具体的な対処方法を説明します。ただし、どんなときでも必ず効果があるとは限りませんので、注意が必要です。

（2）キャッシュ上の配置アドレスのずれの対処方法

プログラムのアライメントを変更することで、キャッシュの１ライン長に変更してしまう方法があります。SuperH RISC engine Ｃ/Ｃ++コンパイラの出力するプログラムのアライメントはデフォルトで4バイトです。このアライメントをSuperH RISC engine Ｃ/Ｃ++コンパイラのコンパイルオプション「align16」にてキャッシュの１ライン長の16バイトに変更することで、必ずキャッシュラインの先頭から配置されるようになります。
ただし、キャッシュの１ライン長が16バイトの時に有効となります。また、プログラムサイズは増加します。

①　最も効率のよい「align16」オプション指定方法

・コンパクトな（キャッシュのラインを１ラインから数ラインしか使用しない）関数群に対し、「align16」オプションを指定する。
・全体のプログラムの中で、汎用の共通ルーチン群の中に小さな関数群がある場合、その小さな関数群に対し、「align16」オプションを指定する。
(この小さな関数群が、同一モジュールに隣接して定義されていれば、キャッシュエントリの競合も少なくなります。)

（3）プログラムサイズの増加

境界調整用のダミー領域およびプログラムサイズの増加などにより、プログラム自体が大きくなり、キャッシュヒット率が低下します。

①　プログラムサイズの増加の対処方法

以下の方法で、プログラムサイズを抑えることができます。
・SuperH RISC engine Ｃ/Ｃ++コンパイラの最適化オプションの「サイズ優先指定」による最適化の実施。
・規模の小さな関数への改善
・特定汎用ルーチン化

（4）キャッシュエントリの使用方法

キャッシュの各エントリが均等に使用されているか。リプレース回数がエントリによって偏っていないか。これは特に、ダイレクトマップ形式のキャッシュでは重要です。
このチェックを行うためには、キャッシュの内容をトレースする仕組みが必要です。
しかし、実際にトレースできない場合が多くあります。
そこで、全体のパフォーマンスをあげる手順は、次のようになります。

① 実行頻度の高い関数をいくつかピックアップします。
② リンケージエディタのマップファイルから、その関数の開始アドレスと終了アドレスを調べます。
③ その関数が使用するキャッシュのエントリをチェックします。
④ 各関数の使用するエントリの統計をとります。
⑤ あるエントリにキャッシュの使用が偏っていないかチェックします。

この段階で、複数の関数が同一のキャッシュラインを使用しているなら、関数の配置アドレスを変えて偏りがなくなるようにします。
配置アドレスを変える手段は、コンパイル時にセクション名を変更する。または、リンクで入力する順番を変更することなどで対処できます。

③の使用エントリを調べる方法は、CPUとキャッシュ方式によって異なりますが、基本的にダイレクトマップ方式のキャッシュであれば、エントリ番号は、絶対アドレスのオフセット値によって決定します。（オフセットの範囲は、キャッシュのサイズに依存します。）

③④⑤の手順で、競合をチェックするのが困難であれば、まず以下を行ってください。
①でピックアップした関数のセクション名を変更してコンパイルする。
そうすることで、①の関数が連続アドレスに割り付きます。つまり、この関数間ではキャッシュ競合が起きないことになります。よって、セクション名を変える関数は、合計のサイズがキャッシュサイズより小さいことが条件です。

(5) プログラミング手法
キャッシュを有効活用するためのプログラミング手法があります。以下を参考にプログラミングしてください。

■タイリングプログラム
「5.9.2 **タイリング**」を参照

■プリフェッチ
「5.9.1 **プリフェッチ命令**」を参照

6. 効率の良いプログラミング技法 (追加)

6.1 オプションの指定方法

効率の良いプログラムを作成するためには、最適化オプションを指定することが効果的です。最適化オプションの選び方、また他のオプションとの組み合わせによって、さらに最適化の効果を高めることができます。

6.1.1 HEW 起動時のオプション (浮動小数点の設定)

HEW 起動時に設定するオプションは、エンディアン、浮動小数点の扱いなどの CPU の使用法全般にかかわる環境を設定します。

プログラムが浮動小数点を使用するとき、浮動小数点に関する設定が性能に大きく影響します。サイズ面、スピード面とも、単精度の浮動小数点形式 (32 bit) の方が倍精度の浮動小数点形式より効率がよくなります。応用分野の精度が単精度で十分のときは単精度の浮動小数点形式を用いるようにしてください。単精度は十進で約 7 桁, 倍精度は十進で約 17 桁の精度です。

【注】 ここで指定する浮動小数点の形式は、プロジェクト全体に影響します。したがって、ファイルごとに異なる指定をすることはできません。

(1) SH-1, SH-2, SH-2E, SH-2A, SH2-DSP, SH3, SH3-DSP, SH4AL-DSP **の場合**

図 6.1 の "double→float 変換" のチェックボックスをクリックします。これにより、プログラム内で使用しているすべての浮動小数点 (double 型と宣言しているものも含めて) を単精度として扱います。

図 6.1 **単精度の指定方法** (SH-1, SH-2, SH-2E, SH-2A, SH2-DSP, SH-3, SH3-DSP, SH4AL-DSP)

(2) SH2A-FPU, SH-4, SH-4A **の場合**

図 6.2 の FPU のメニューで "Single" を指定します。

図 6.2 **単精度の指定方法** (SH2A-FPU, SH-4, SH-4A)

ここで "Double" を指定するとすべての演算が倍精度になります。"Mix" を指定すると，プログラムの記述どおりに float を単精度, double を倍精度で演算します（これは SH-1, SH-2, SH-2E, SH-2A, SH2-DSP, SH3, SH3-DSP, SH4AL-DSP で "double→float 変換" をチェックしないのと同じ動作になりますが、FPU の単精度モードと倍精度モードを切り替えながら実行するので、double を指定するより効率が悪くなることがあります。

RENESAS

6.1.2 最適化オプションの指定方法 (スピードとサイズ)

最適化オプションには、以下の主要な選択肢があります (図 6.3)。

(a) サイズ&スピード (デフォルト)
(b) サイズ優先
(c) スピード優先

図 6.3 最適化オプション

"サイズ&スピード" (デフォルト) は、サイズもスピードも向上する最適化だけを実行します。サイズ優先は、"サイズ&スピード" に加えて、スピードを犠牲にしてもサイズを削減する最適化を実行します。"スピード優先"は "サイズ&スピード" に加えて、サイズを犠牲にしてもスピードを向上する最適化を実行します。

サイズ、またはスピードのどちらかを優先する場合は、それぞれ "サイズ優先"、"スピード優先"を指定してください。

多くのシステムにおいては、プログラム内でスピードが重要になる部分はごく一部です。このような場合は、スピードが必要なファイルを"スピード優先"で、それ以外のファイルを"サイズ優先"で最適化すると効果的です。これらの最適化オプションは、ファイル間のインタフェースを変更しないので、ファイルごとに設定できます。

ファイルごとにオプションを設定するには、左側のディレクトリで対象となるソースファイルを選択してからオプションを指定します (図 6.4)。

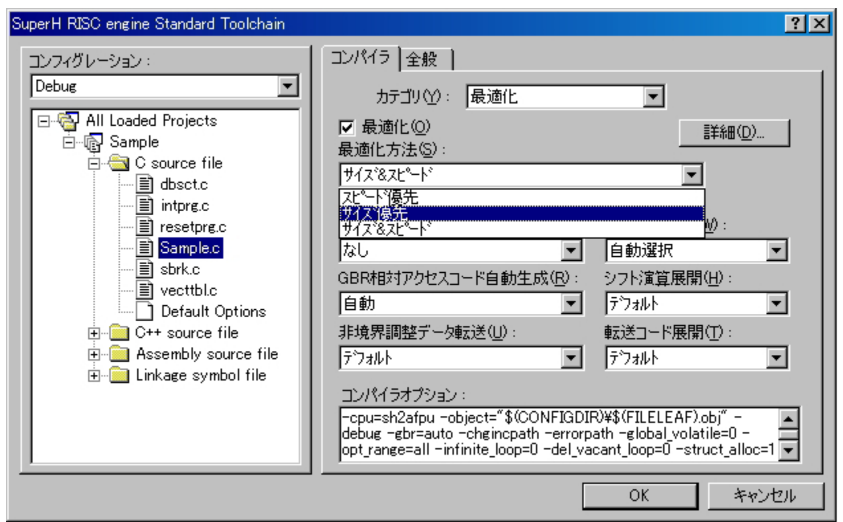


図 6.4 ファイルごとのオプションの指定方法

6.1.3 プログラムの互換性に注意が必要なオプション (関数インタフェース)

関数のインタフェースを変更するオプションで、プログラムの実行効率を向上できるオプションがあります。これらのオプションのデフォルトは、互換性のためコンパイラの旧バージョンとあわせていますが、プロジェクト全体にわたって変更すれば、互換性の問題なく効率を向上させることができます。これらのオプションは、コンパイラオプションのその他メニューにあります。

【注】 プロジェクトの中にアセンブラ記述プログラムが含まれている場合は、どちらの規約に従って作成しているかを確認する必要があります。

(1) "MAC **レジスタを保証する"チェックボックス**

MACH/MACL レジスタを使用する関数が関数の入口／出口でこれらのレジスタを退避／回復するオプションです。デフォルトでは退避／回復していますが、これらのレジスタはほとんどの場合ワークレジスタとしてしか用いないため、退避／回復しない (チェックボックスのチェックを外す) 方がサイズ／スピードとも向上します (**図** 6.5)。

図 6.5 MACH/MACL **レジスタ退避／回復オプション**

(2) "**返却値の拡張を行う"チェックボックス**

関数のリターン型が char, unsigned char, short, unsigned short 型のとき、デフォルトは、呼び出された関数では符号拡張(またはゼロ拡張) を実行せず、呼び出し側で符号拡張 (またはゼロ拡張) を実行することになっています (コンパイラの旧バージョンと互換の仕様です)。

関数が複数回呼び出される場合、リターン値は、呼び出し側で符号拡張するよりも、呼び出された側で符号拡張する(チェックボックスをチェックする) 方が、拡張のコードが 1 回で済み、サイズ面で有利になります。

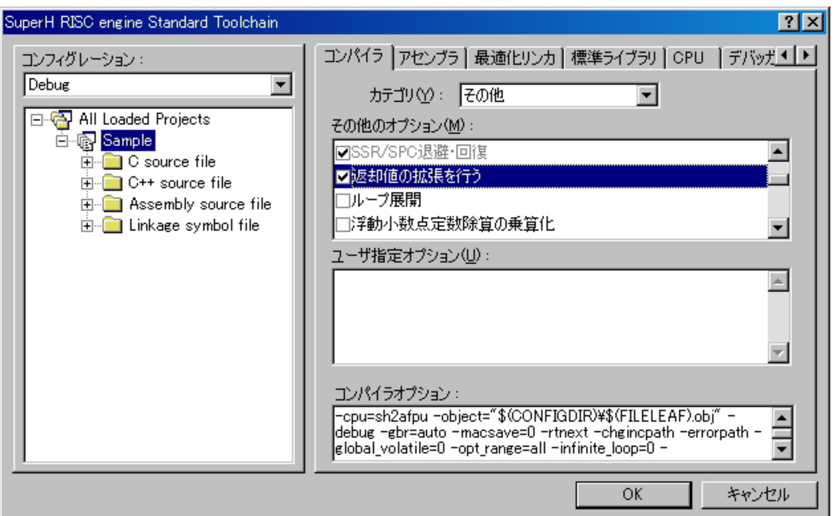


図 6.6 リターン値拡張オプション

【注】 SH **のレジスタサイズは** 4 **バイトなので、関数のリターン値だけでなく、関数パラメタ、局所変数などレジスタに置くデータは、**int, unsigned, long, unsigned long **の** 4 **バイト型で宣言すると、符号拡張処理をしなくてよいので効率の良いプログラムになります。(詳細は「**5.1.1 **局所変数(データサイズ)」を参照してください。)**

6.1.4 Volatile 宣言された変数の扱いに関するオプション (volatile 変数)

Volatile 宣言は、変数に対するアクセスの最適化を抑止するために指定します。プログラムで使用する外部変数を volatile と宣言するのは、主に以下の 2 つの目的があります。

(a) 周辺入出力レジスタへのアクセスを最適化することを防ぐ。
(b) 異なるタスクや割り込み処理によって共有する変数に対するアクセスを最適化することを防ぐ。

SH C/C++ コンパイラの V6.0 以前では、外部変数のアクセスに対する最適化はほとんど実施していませんでしたが、V7.0 以降では大幅に強化されました。このため、上記 (1), (2) に該当する変数を volatile 宣言していないプログラムを V6.0 以前のコンパイラで開発していた場合は、V7.0 以降のコンパイラバージョンに移行するとき、プログラムに volatile 宣言を追加することを推奨します。プログラムを変更しない場合は、コンパイラオプションの最適化メニュー内の "詳細" のボタンを押して詳細オプションを設定してください。

これらのオプションは最適化の抑止につながりますので、V6.0 以前のプログラムの動作を損なわない範囲で、最低限の最適化抑止を指定してください。

(1) 周辺入出力レジスタが volatile 宣言されていない場合

周辺入出力レジスタの場合、2 回連続して書き込んだり、2 回連続して読み出したりするアクセスを一度のアクセスに最適化すると、レジスタの仕様によっては動作が異なる場合があります。

このような最適化を抑止するために、最適化詳細オプションの "外部変数" タブでレベル１（図 6.7）を指定します。

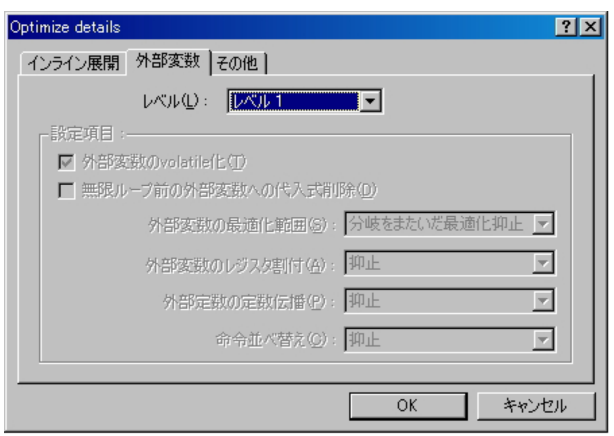


図 6.7 最適化詳細オプションレベル 1

ただし、レベル 1 はその他の最適化も抑止していますので、volatile 宣言の効果だけを残して他の最適化を最大限実施するためには、以下の手順で指定します (図 6.8)。

(a) レベルを "レベル 1" と指定する。
(b) その後レベルを "カスタム" に指定する (レベル 1 の設定が残ります)。
(c) "外部変数の最適化範囲" を "全範囲有効" にする。
(d) "外部変数のレジスタ割付" を "割り付け" にする。
(e) "外部定数の定数伝播" を "定数伝播" にする。
(f) "命令並べ替え" を "有効" にする。

例:

ソースコード	

extern int x;
 	
void f (void)
{
  x=1;
  x=2;
}	

アセンブラ展開コード
_f:
        MOV.L   L11,R6
        MOV   #2,R2
        RTS
        MOV.L   R2,@R6
	アセンブラ展開コード（オプション指定時）
	_f:
	        MOV.L L11,R6
	        MOV #1,R2
	        MOV.L R2,@R6
	        MOV #2,R2
	        RTS
	        MOV.L R2,@R6

この例では、オプションを指定しない場合入出力レジスタへのアクセスの 2 回が 1 回にまとめられるため、周辺入出力レジスタに対する効果が異なる場合があります。

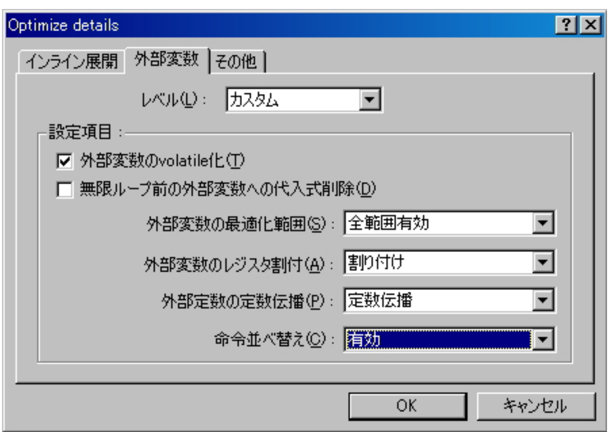


図 6.8 周辺入出力に対する volatile の指定

【注】 1. **HEW が生成する機種ごとのヘッダファイルでは、周辺入出力レジスタを volatile 宣言しています。Volatile 宣言をしていない外部変数が、タスクや割り込みで共有する変数だけである場合は、(2) で説明するオプションを指定してください。**

2. **1 つの式の中で外部変数を 2 回参照した場合は、その順序は保障されません。外部変数を volatile と宣言しても同じです。したがって、周辺入出力レジスタを複数回参照する場合は、異なる文で参照してください。**

(2) タスクや割り込みで共有する外部変数が volatile 宣言されていない場合

タスクや割り込みで共有する外部変数は、メモリ上の変数なので、連続したアクセスを統合してもプログラムの効果は変わりませんが、ループ内で参照する外部変数がレジスタに割り付けられてしまうと、他のタスクや割り込み処理でその変数が変更されても、ループの処理に影響が出ず、動作が変わってしまう場合があります。

このような最適化を抑止するために、最適化詳細オプションの "外部変数" タブでレベル 2 (**図** 6.9) を指定します。

RENESAS

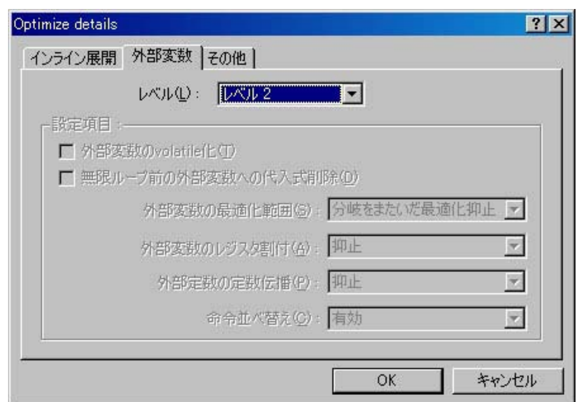


図 6.9 **最適化詳細オプション レベル** 2

ただし、レベル 2 はその他の最適化も抑止していますので、volatile 宣言の効果だけを残して他の最適化を最大限実施するためには、以下の手順で指定します (**図** 6.10)。

(a) レベルを "Level 2" と指定する。
(b) その後 レベルを "カスタム" に指定する (レベル2 の設定が残ります)。
(c) "外部変数の最適化範囲" を "ループ内最適化抑止" にする (ループに対する外部変数の最適化を抑止するという意味です。デフォルトの "分岐をまたいだ最適化抑止" のままだと、ループ以外の構文に対しても抑止されます)。
(d) "外部変数のレジスタ割付" を "割り付け" にする。
(e) "外部定数の定数伝播" を "定数伝播" にする。

例:

ソースコード

```
extern int  x; /* 割り込みで変更
                  される可能性が
                  ある */

void f (void)
{
  x=1;
  while (1){
    if (x!=1) break;
  }
}
```

アセンブラ展開コード

```
_f
L10:
         BRA        L10
         NOP
         RTS
         NOP
```

アセンブラ展開コード（オプション指定時）

```
_f
         MOV.L      L13,
         MOV        #1,R2
         MOV.L      R2,@R1
         MOV.L      @R1,R0
         CMP/EQ     #1,R0
         BT         L11
         RTS
         NOP
```

オプションを指定しない場合、ソースコードは、割り込みで外部変数 x が変更されることによりループを抜けることを想定していますが、x が volatile 宣言されていないので、最適化の結果無限ループになる可能性があります。

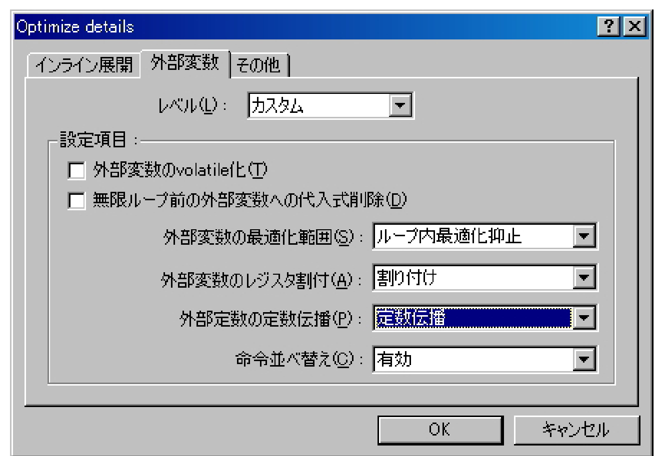


図 6.10　タスク、割り込みで共有する変数に対する volatile 指定

タスク、割り込みで共有する変数の参照がループの条件式である場合に限定されているときは、最適化詳細オプションは default (レベル 3) のままで、その他オプションの "ループ判定式の最適化抑止" を指定することによって、上記と同等の効果になり、しかも最適化抑止を低減することができます (**図** 6.11)。

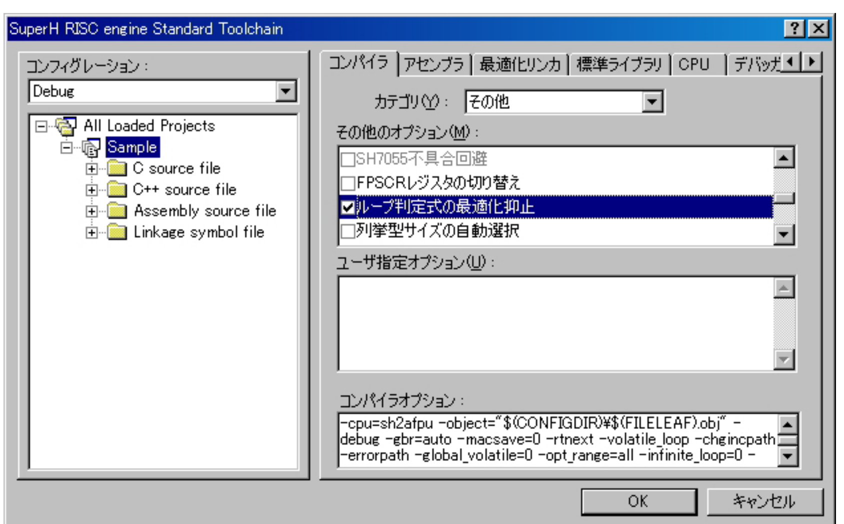


図 6.11　ループ条件の volatile 化

例:

<table>
<tr><td>

ソースコード

```
extern int  x; /* 割り込みで変更
                  される可能性が
                  ある */

void f (void)
{
  x=1;
  while (x){    /* 条件式で共有変数
                  をアクセスする */
  }
}
```

</td><td></td></tr>
<tr><td>

アセンブラ展開コード

```
_f:
L10:
        BRA        L10
        NOP
        RTS
        NOP
```

</td><td>

アセンブラ展開コード（オプション指定時）

```
_f:
        MOV.L      L13,R1
        MOV        #1,R2
        MOV.L      R2,@R1
L11:
        MOV.L      @R1,R2
        TST        R2,R2
        BF         L11
        RTS
        NOP
```

</td></tr>
</table>

この例ではタスクで共有される変数をループの条件式で判定していますので、ループ条件を volatile 扱いにすることによって、無限ループになることを抑止することができます。

【注】 本節の (2) で説明した方法は、タスクや割り込みが共有する変数が、ループの中で毎回参照されることを保証する方法です。これによって、ループで毎回変数を参照することになり、他のタスクや割り込みで変更された変数の値が正しく反映されることになります。
しかし、ループを含まない範囲で 2 回参照する変数に対しする参照は、この設定では最適化される場合があります。このような参照に対して毎回他のタスクや割り込みで変更された変数の値を正しく反映することが必要な場合は、「(1) 周辺入出力レジスタが volatile 宣言されていない場合」のオプションを指定してください。

例:

```
extern int x;
void f(void){
  int a;
  a=x;
  /* ループを含まない、長い処理 */
  a=x;
}
```

このようなプログラムの、x の 2 回の参照の間にタスクや割り込みで x が変更されたことを検知する必要がある場合は、「(1) **周辺入出力レジスタが volatile 宣言されていない場合**」のオプションを指定してください。

6.1.5 空ループ削除の抑止

タイミングをとる目的で、プログラム内に空のループを書いたとき、V7.0 以降では最適化によって削除されることがあります。これを抑止するためには、コンパイラオプションの最適化メニュー内の "詳細" のボタンを押して詳細オプションで、"その他" タブを選択し、"空ループ削除" のチェックが外れていることを確認してください (チェックが外れているのがデフォルトです) (**図** 6.12)。

図 6.12 空ループ削除の抑止

例:

ソースコード

```
void f (void)
{
  int x;
  for (x=0; x<100; x++){
    /* タイミングループ */
  }
}
```

アセンブラ展開コード

```
_f:
         RTS
         NOP
```

アセンブラ展開コード（オプション指定時）

```
_f:
         MOV        #100,R2
L11:
         DT         R2
         BF         L11
         RTS
         NOP
```

この例では、タイミングループ内の処理がないため、空ループ削除を抑止しないと、コードが削除される可能性があります。

【注】 **ループが削除されないようにするためには、ループ内で** volatile **宣言した変数をアクセスするか、ループ内で組み込み関数** nop() **を呼び出すなどの方法があります。このような場合は、本オプションをチェックすることにより、ループの最適化を強化することができます。**

RENESAS

6.1.6 Const 変数最適化の抑止

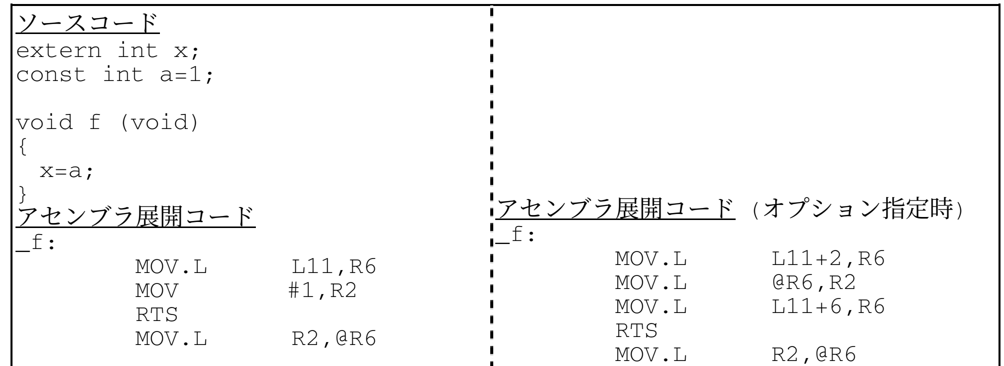
最適化処理により、const 宣言した変数を定数として最適化することがあります。これにより、プログラムの動作は変わりませんが、たとえば、デバッグ時に const 宣言した変数の値を変更しても、プログラムには影響がないことになります。

const 宣言された変数 a をその初期値に置き換える最適化を実施しているため、デバッグ時に a の値を変更してもプログラムの動作は変わりません。

このような最適化を抑止するためには、コンパイラオプションの最適化メニュー内の "詳細" のボタンを押して詳細オプションで、"外部変数" タブを選択し、以下の手順で指定してください (図 6.13)。

(a) レベルを "レベル3" (デフォルト) と指定する。
(b) その後レベルを "カスタム" に指定する (レベル3 の設定が残ります)。
(c) "外部変数の定数伝播" を "抑止" にする。

例:



ソースコード

```
extern int x;
const int a=1;

void f (void)
{
  x=a;
}
```

アセンブラ展開コード

```
_f:
        MOV.L      L11,R6
        MOV        #1,R2
        RTS
        MOV.L      R2,@R6
```

アセンブラ展開コード（オプション指定時）

```
_f:
        MOV.L      L11+2,R6
        MOV.L      @R6,R2
        MOV.L      L11+6,R6
        RTS
        MOV.L      R2,@R6
```

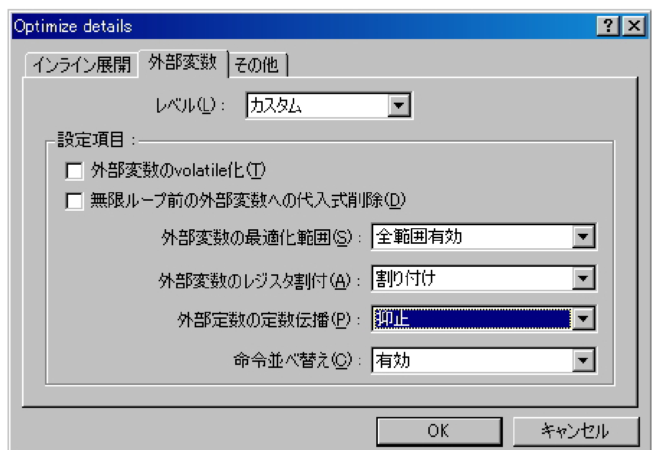


図 6.13 Const 変数最適化の抑止

6.1.7 浮動小数点の実行効率に効果があるオプション

(1) 浮動小数点定数による除算を乗算に置き換える最適化

浮動小数点定数による除算を、その定数の逆数による乗算に置き換えます。

この最適化は、結果の数値が異なる場合がある（ただし、誤差の範囲です）ので、C/C++ コンパイラの最適化のメニューではなく、その他のメニューに入っています。このメニューで "浮動小数点定数除算の乗算化" を選択します (図 6.14)。

例:

ソースコード

```
float x;

void f (float y)
{
  x=y/3.0;
}
```

アセンブラ展開コード

```
_f:
        MOVA      L11,R0
        MOV.L     L11+4,R2
        FMOV.S    @R0,FR8
        FDIV      FR8,FR4
        RTS
        FMOV.S    FR4,@R2
```

アセンブラ展開コード（オプション指定時）

```
_f:
        MOVA      L11,R0
        MOV.L     L11+4,R2
        FMOV.S    @R0,FR8
        FMUL      FR8,FR4
        RTS
        FMOV.S    FR4,@R2
```

3.0 で割る演算を、それより高速な乗算に置き換えます (結果は誤差の範囲ですが異なることがあります)。

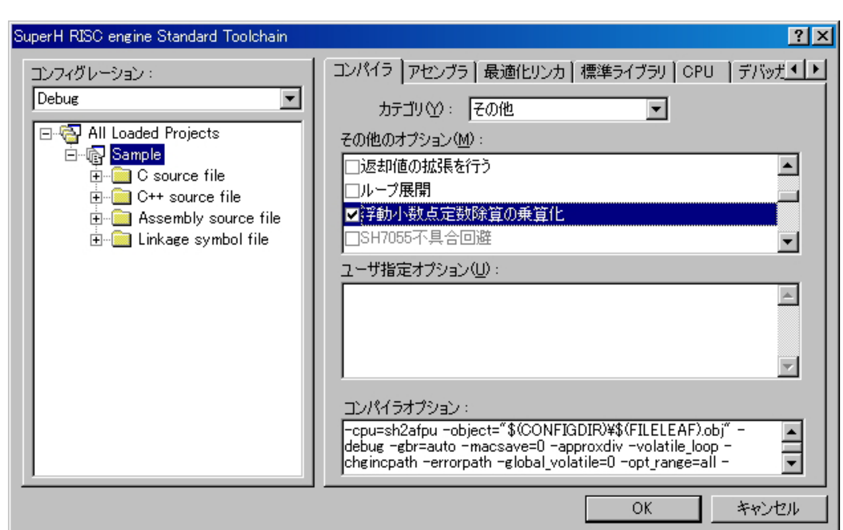


図 6.14 浮動小数点定数除算の最適化

(2) SH2A-FPU, SH-4, SH-4A で浮動小数点の設定で Mix を選択したときの注意事項

SH2A-FPU, SH-4, SH-4A の浮動小数点の設定で Mix を指定したとき、互換性のため、コンパイラの旧バージョンと同じ呼び出しインタフェースになります。このインタフェースは、関数からリターンしたときの浮動小数点の設定が不定になるため、そのたびに FPSCR を設定し直す必要がありました。

C/C++ コンパイラの Other オプションで、"FPSCR レジスタの切り替え" を指定することにより、FPSCR の切り替えは倍精度演算の前後だけになり、プログラムのサイズ、スピードがともに向上します (図 6.15)。

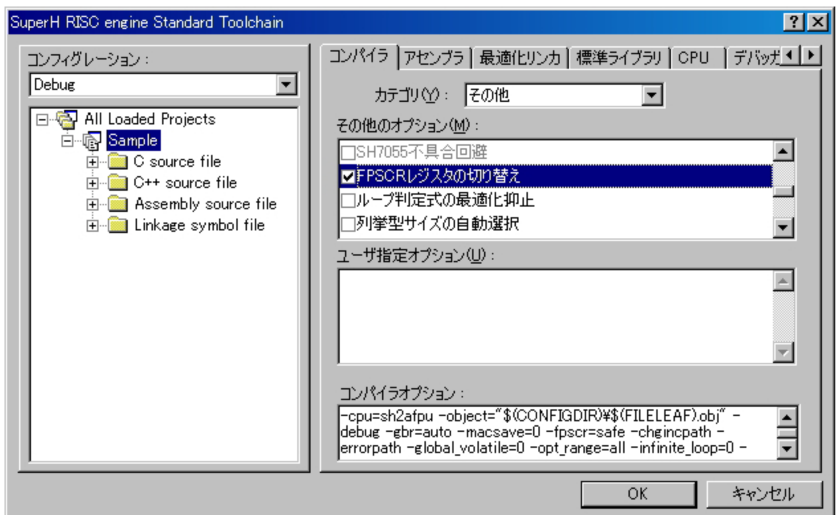


図 6.15　SH-4 で浮動小数点の設定で Mix を選択したときの推奨オプション

【注】　本オプションは関数のインタフェースを変更するため、全ファイルに対して同時に変更する必要があります。

6.2 定数除算最適化

■ポイント

定数による除算は、除算以外の演算に展開する最適化を実施しています。したがって、できるだけ定数の除算を使用してください。

■説明

定数による除算に対して、その逆数の近似値を乗算し、その結果を微調整する、という最適化を実施しています。これにより、除算のサブルーチンの呼び出しに対して大幅に実行速度を改善することができます。

■使用例

以下の改善例では、除数を定数にすることにより、除算ルーチンを呼び出さずに直接 3 による商を求める命令列が生成されます。他の定数による除算に対しても同様のコードが生成されます。

改善前ソースコード

```
int x;
int z=3;
void f (int y){
  x=y/z;
}
```

改善前アセンブリコード

```
_f:
          STS.L      PR,@-R15
          MOV.L      L11,R5
          MOV        R4,R1
          MOV.L      L11+4,R2
          JSR        @R2
          MOV.L      @R5,R0
          MOV.L      L11+8,R6
          LDS.L      @R15+,PR
          RTS
          MOV.L      R0,@R6
L11:
          .DATA.L    _z
          .DATA.L    __divls
          .DATA.L    _x
```

改善後ソースコード

```
int x;
void f (int y){
  x=y/3;
}
```

改善後アセンブリコード

```
_f:
          MOV.L      L11,R2
          DMULS.L    R4,R2
          STS        MACH,R6
          MOV        R6,R0
          ROTL       R0
          AND        #1,R0
          ADD        R6,R0
          MOV.L      L11+4,R6
          RTS
          MOV.L      R0,@R6
L11:
          .DATA.L    H'55555556
          .DATA.L    _x
```

【注】 この最適化は、スピードを大幅に改善しますが、展開したコードが大きくなる場合があるため、サイズ最適化のときは適用されません。

6.3 整数除算のサイズ

■ポイント

整数除算は int 型 (32 bit) で実行するよりも、より短い data 型 (char または short) で実行した方がスピードが向上します。

■説明

除算の実行時ルーチンは, 32 bit, 16 bit, 8 bit のサイズごとに用意されています。除算は、もし値の範囲が限定されているならば、サイズが小さい型で実行した方がスピードが向上します。

■使用例

以下の改善例は、除数、被除数、結果が 16 bit の範囲であるときに、除算のオペランドと結果を short 型と宣言することにより、32 bit 除算のルーチン (divls) ではなく、16 bit 除算のルーチン (divws) が呼ばれる例です。

改善前ソースコード

```
int x;
int y;
int z;
void f(){
  x=y/z;
}
```

改善前アセンブリコード

```
_f:
         STS.L      PR,@-R15
         MOV.L      L11+2,R6
         MOV.L      L11+6,R4
         MOV.L      L11+10,R2
         MOV.L      @R6,R0
         JSR        @R2
         MOV.L      @R4,R1
         MOV.L      L11+14,R6
         LDS.L      @R15+,PR
         RTS
         MOV.L      R0,@R6
L11:
         .RES.W     1
         .DATA.L    _z
         .DATA.L    _y
         .DATA.L    __divls
         .DATA.L    _x
```

改善後ソースコード

```
short x;
short y;
short z;
void f(){
  x=y/z;
}
```

改善後アセンブリコード

```
_f
         STS.L      PR,@-R15
         MOV.L      L11+2,R6
         MOV.L      L11+6,R4
         MOV.L      L11+10,R2
         MOV.W      @R6,R0
         JSR        @R2
         MOV.W      @R4,R1
         MOV.L      L11+14,R6
         LDS.L      @R15+,PR
         RTS
         MOV.W      R0,@R6
L11:
         .RES.W     1
         .DATA.L    _z
         .DATA.L    _y
         .DATA.L    __divws
         .DATA.L    _x
```

6.4 レジスタ宣言

■ポイント

SH C/C++ コンパイラ Ver. 7以降では、レジスタ宣言をしても、しなくても、変数の割り付けは変わりません。

■説明

コンパイラは､局所変数の使用頻度を (ループ内に出現する場合は優先度を上げるなど) 重み付けして､それに従ってレジスタ割り付けを実行します。変数に対するレジスタ宣言は無視します。

■使用例

以下の例は、レジスタ宣言のないプログラムとあるプログラムのコンパイル例です。どちらも同等のレジスタ割り付けをしたコードを生成します。

レジスタ宣言のないソースコード	レジスタ宣言したソースコード

<pre>int a[10];
int f(){
  int i;
  int s=0;
  for (i=0; i<10; i++)
    s+=a[i];
  return s;
}</pre>
	<pre>int a[10];
	int f(){
	  register int i;
	  register int s=0;
	  for (i=0; i<10; i++)
	    s+=a[i];
	  return s;
	}</pre>

アセンブリコード	アセンブリコード

<pre>_f:
        MOV   #0,R2
        MOV.L   L13,R5
        MOV   #5,R4
L11:
        MOV.L   @R5,R6
        DT   R4
        ADD   R6,R2
        MOV.L   @(4,R5),R6
        ADD   #8,R5
        BF/S   L11
        ADD   R6,R2
        RTS
        MOV   R2,R0
L13:
        .DATA.L   _a</pre>
	<pre>_f
	        MOV #0,R2
	        MOV.L L13,R5
	        MOV #5,R4
	L11:
	        MOV.L @R5,R6
	        DT R4
	        ADD R6,R2
	        MOV.L @(4,R5),R6
	        ADD #8,R5
	        BF/S L11
	        ADD R6,R2
	        RTS
	        MOV R2,R0
	L13:
	        .DATA.L _a</pre>

【注】 1. **変数のレジスタへの割り付けはコンパイラが自動的に実行しますが、関数内で使用する局所変数の数が多いと、有効なレジスタ割り付けが困難になります。適切に関数を分割し、ループ内で使用する局所変数の数が 8 個程度以内になるようにすることをお勧めします。**

2. Ver.9 **以降では、**enable_register **オプションを指定することにより、**register **記憶クラスを指定した変数を優先的にレジスタに割り付けます（デフォルトでは、**enable_register **は指定されていません）。**

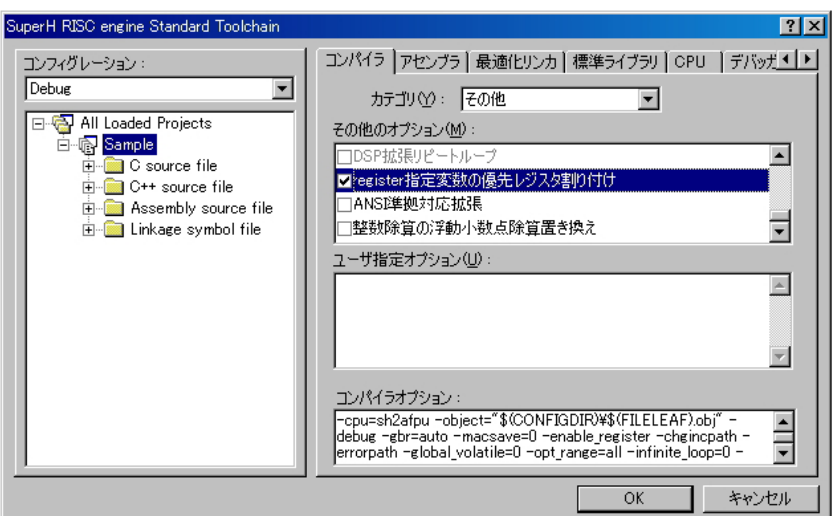


図 6.16 enable_register オプション指定

RENESAS

6.5 構造体宣言のメンバオフセット

■ポイント

構造体の中でよく使用するメンバは、先頭に宣言するようにすればサイズ、スピードとも向上します。

■説明

構造体メンバは、構造体アドレスにオフセットを加算してアクセスします。オフセットが小さいほうがサイズ、スピードとも有利なので、よく使用するメンバを先頭に宣言するようにしてください。

もっとも効果的なのは、char, unsigned char 型で先頭から 16 byte 未満, short, unsigned short 型で先頭から 32 byte 未満, int, unsigned, long, unsigned long 型で先頭から 64 byte 未満です。

■使用例

以下の例は、構造体のオフセットによってコードが変わる例を示します。

<u>改善前ソースコード</u>

```
struct S{
  int a[100];
  int x;
};

int f(struct S *p){
  return p->x;
}
```

<u>改善前アセンブリコード</u>

```
_f:
        MOV         #100,R0
        SHLL2       R0
        RTS
        MOV.L       @(R0,R4),R0
```

<u>改善後ソースコード</u>

```
struct S{
  int x;
  int a[100];
};

int f(struct S *p){
  return p->x;
}
```

<u>改善後アセンブリコード</u>

```
_f:
        RTS
        MOV.L       @R4,R0
```

6.6 ビットフィールドの割り付け

■ポイント

ビットフィールドで、同じ式で関連して参照されるものは、同じ構造体内に割り付けるようにしてください。

■説明

異なるビットフィールドのメンバを参照するためには、そのたびにビットフィールドを含むデータをロードしなければなりません。関連するビットフィールドを同じ構造体内にまとめて割り付けることによって、このロードを一度で済ますことができます。

■使用例

以下の例は、同じ構造体に関連するビットフィールドを割り付けることによってスピード、サイズともに改善する例です。

改善前ソースコード

```
struct bits{
  unsigned int b0: 1;
} f1, f2;

int f(void){
  if (f1.b0 && f2.b0) return 1;
  else return 0;
}
```

改善前アセンブリコード

```
_f:
        MOV.L      L15,R6
        MOV.B      @R6,R0
        TST        #128,R0
        BT         L12
        MOV.L      L15+4,R6
        MOV.B      @R6,R0
        TST        #128,R0
        BT         L12
        RTS
        MOV        #1,R0
L12:
        RTS
        MOV        #0,R0
L15:
        .DATA.L    _f1
        .DATA.L    _f2
```

改善後ソースコード

```
struct bits{
  unsigned int b0: 1;
  unsigned int b1: 1;
} f1;

int f(void){
  if (f1.b0 && f1.b1) return 1;
  else return 0;
}
```

改善後アセンブリコード

```
_f:
        MOV.L      L11,R6
        MOV        #-64,R3
        EXTU.B     R3,R3
        MOV.B      @R6,R0
        AND        #192,R0
        CMP/EQ     R3,R0
        RTS
        MOVT       R0
L11:
        .DATA.L    _f1
```


RENESAS

6.7 ソフトウェアパイプライン (浮動小数点のテーブルサーチ)

■ポイント

テーブルサーチのコードで、テーブルから参照したデータをすぐ比較するのではなく、ループの1つ前の回でロードしたデータと比較するようにすれば実行速度を改善できます。

■説明

テーブルサーチのように命令数の少ないループでは、パイプライン最適化をしても命令を並び替える余地がなく、十分な最適化ができないことがあります。たとえば浮動小数点のテーブルサーチでは、テーブルからのデータのロードの直後に比較すると、FCMP 演算がロードが完了するまで待たされます。これを避けるためには、ループの前の回で比較データを局所変数にロードし、次の回で比較するというようなプログラムの工夫が必要です。

■使用例

改善前は、ロードした浮動小数点データ (FR8) を直後の FCMP 命令で比較しています。ループの1つ前の回参照したデータを次の回で比較するように改善すると、ロードはループの分岐命令と平行して実行されます。

改善前ソースコード

```
float a[100];

int f(float b){
  int i=0;
  float *p=a;
  while (i<100){
    if (*p==b) return i;
    i++;
    p++;
  }
  return -1;
}
```

改善前アセンブリコード

```
_f:
        MOV         #0,R5
        MOV.L       L16,R2
        MOV         #100,R6
L11:
        FMOV.S      @R2,FR8
        FCMP/EQ     FR4,FR8
        BT          L12
        DT          R6
        ADD         #1,R5
        BF/S        L11
        ADD         #4,R2
        RTS
        MOV         #-1,R0
L12:
        RTS
        MOV         R5,R0
L16:
        .DATA.L     _a
```

改善後ソースコード

```
float a[100];

int f(float b){
  int i=0;
  float *p=a;
  float tmp=*p;
  while (i<100){
    if (tmp==b) return i;
    i++;
    p++;
    tmp=*p;
  }
  return -1;
}
```

改善後アセンブリコード

```
_f:
        MOV.L       L16+2,R2
        MOV         #0,R5
        MOV         #100,R6
        FMOV.S      @R2,FR8
L11:
        FCMP/EQ     FR4,FR8
        BT          L12
        ADD         #4,R2
        DT          R6
        FMOV.S      @R2,FR8
        BF/S        L11
        ADD         #1,R5
        RTS
        MOV         #-1,R0
L12:
        RTS
        MOV         R5,R0
L16:
        .RES.W      1
        .DATA.L     _a
```

6.8 データのアクセスサイズの保証

■ポイント

周辺レジスタをアクセスする場合、メモリアクセスする命令のサイズ (バイト、ワード、ロングワード) を保証する必要がある場合は、volatile 宣言してください。

■説明

大域変数、pointer をアクセスする命令のサイズを保証するためには、volatile 宣言してください。そうすれば、ロード・ストアする命令はデータ型のサイズどおりになります。ビットフィールドをアクセスする場合は、volatile 宣言すればビットフィールドを宣言するときのデータ型でアクセスします。volatile 宣言をしないと、ビットフィールドのアクセスが最適化され、宣言された型以外でのアクセスが発生することがあります。

■使用例

volatile 宣言しないとき、メンバ x のアクセスは最適化されてバイトアクセスになっています。volatile 宣言すると、宣言どおりの型 (ワード) でアクセスします。

volatile を指定しないソースコード

```
struct S{
  short x: 8;
  short y: 8;
} *p;

int f(){
  return p->x;
}
```

アセンブリコード

```
_f:
        MOV.L       L11+2,R2
        MOV.L       @R2,R6
        MOV.B       @R6,R2
        RTS
        EXTS.W      R2,R0
L11:
        .RES.W      1
        .DATA.L     _p
```

volatile を指定したソースコード

```
volatile struct S{
  short x: 8;
  short y: 8;
} *p;

int f(){
  return p->x;
}
```

アセンブリコード

```
_f:
        MOV.L       L11,R2
        MOV.L       @R2,R6
        MOV.W       @R6,R2
        SHLR8       R2
        RTS
        EXTS.B      R2,R0
L11:
        .DATA.L     _p
```

6.9 浮動小数点命令の活用

■ポイント

単精度浮動小数点命令 FABS (SH2-E, SH2A-FPU, SH-4, SH-4A), FSQRT (SH2A-FPU, SH-4, SH-4A) を活用するためには, インクルードファイル <mathf.h> をインクルードし、単精度の浮動小数点関数 fabsf, sqrtf を呼び出します。

■説明

単精度の浮動小数点命令 FABS (SH2-E, SH2A-FPU, SH-4, SH-4A), FSQRT (SH2A-FPU, SH-4, SH-4A) を活用するためには、以下のようにしてください。

(a) <math.h> をインクルードする

(b) fabsf 関数 (FABS), sqrtf 関数 (FSQRT) を呼び出す

■使用例

改善前の例では、<mathf.h> をインクルードしていないため、コンパイラは標準の関数であると認識せず、ライブラリから fabsf 関数を呼び出します。<mathf.h> をインクルードすれば、コンパイラが FABS 命令に対応する関数であることを認識でき、FABS 命令を直接生成します。

改善前ソースコード

```
float fabsf(float);

float f(float x, float y){
  return fabsf(x)+fabsf(y);
}
```

改善前アセンブリコード

```
_f:
        STS.L      PR,@-R15
        FMOV.S     FR14,@-R15
        FMOV.S     FR15,@-R15
        MOV.L      L12+2,R2
        JSR        @R2
        FMOV.S     FR5,FR15
        MOV.L      L12+2,R2
        FMOV.S     FR0,FR14
        JSR        @R2
        FMOV.S     FR15,FR4
        FADD       FR0,FR14
        FMOV.S     FR14,FR0
        FMOV.S     @R15+,FR15
        FMOV.S     @R15+,FR14
        LDS.L      @R15+,PR
        RTS
        NOP
L12:
        .RES.W     1
        .DATA.L    _fabsf
```

改善後ソースコード

```
#include <mathf.h>

float f(float x, float y){
  return fabsf(x)+fabsf(y);
}
```

改善後アセンブリコード

```
_f:
        FABS       FR4
        FABS       FR5
        FADD       FR5,FR4
        RTS
        FMOV.S     FR4,FR0
```

【注】 ヘッダ <mathf.h> は ANSI 標準の C ライブラリ機能ではありません。

7. HEW の活用

本章では HEW を使用時の、ビルドやシミュレーションについての活用方法を記述します。
HEW のバージョンによりサポートしている機能や方法が変わるのでご注意ください。
各項目の【備考】に対応バージョンを示します。

HEW 活用方法の一覧を示します。

No.	大項目	中項目	参照
1	ビルド編	自動生成ファイルの再生成と編集	7.1.1
2		メイクファイルの出力	7.1.2
3		メイクファイルの入力	7.1.3
4		カスタムプロジェクトタイプの作成	7.1.4
5		マルチ CPU 機能	7.1.5
6		ネットワーク機能	7.1.6
7		古い HEW から転用する場合	7.1.7
8		HIM システムから転用する場合	7.1.8
9		サポート CPU の追加	7.1.9
10	シミュレーション編	擬似割り込み	7.2.1
11		ブレークポイントの便利機能	7.2.2
12		カバレジ機能	7.2.3
13		ファイル入出力	7.2.4
14		デバッガターゲットの同期	7.2.5
15		タイマ使用方法	7.2.6
16		タイマ使用の具体例	7.2.7
17		デバッガターゲットの再登録	7.2.8
18	Call Walker 編	スタック情報ファイルの作成方法	7.3.1
19		Call Walker の起動	7.3.2
20		ファイルのオープンと Call Walker の画面	7.3.3
21		スタック情報の編集	7.3.4
22		アセンブラプログラムのスタック使用量	7.3.5
23		スタック情報のマージ(連結)	7.3.6
24		その他の機能	7.3.7

7.1　ビルド編

7.1.1　自動生成ファイルの再生成と編集

■説明

HEW はワークスペースの新規作成時にプロジェクトタイプで Application などを選択すると、I/O レジスタの定義や割り込み関数などのさまざまなファイルを自動的に生成することができます。

しかし、プロジェクトの新規作成時には必要と思わず、自動生成をしないことがあります。

また、編集や設定を忘れることがあります。

このようなときに、本機能を用いるとプロジェクト作成後に再度ファイルを自動的に生成、編集することができます。

ただし、本機能はワークスペースの新規作成時にプロジェクトタイプで Application を選択した場合に限り使用することができます。

■使用方法

HEW メニュー：プロジェクト>構成の編集

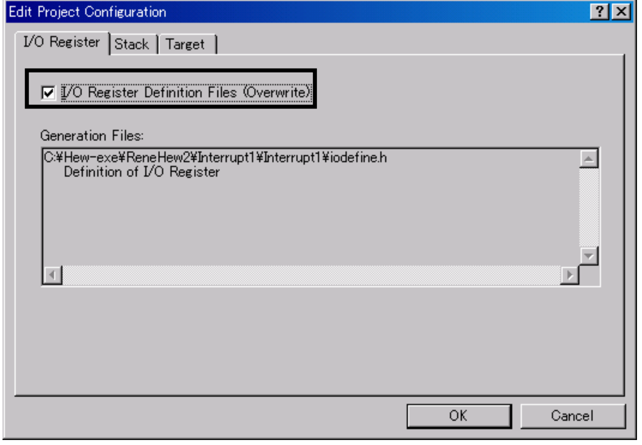
■再生成できるファイル

I/O ポートファイル定義ファイル　　　：iodefine.h

【生成方法】

Edit Project Configuration ダイアログの[I/O Register タブ-I/O Register Definition Files]をチェックすると、iodefine.h を再生成できます。

iodefine.h を誤って変更してしまったときは、iodefine.h が存在しても上書きすることができます。

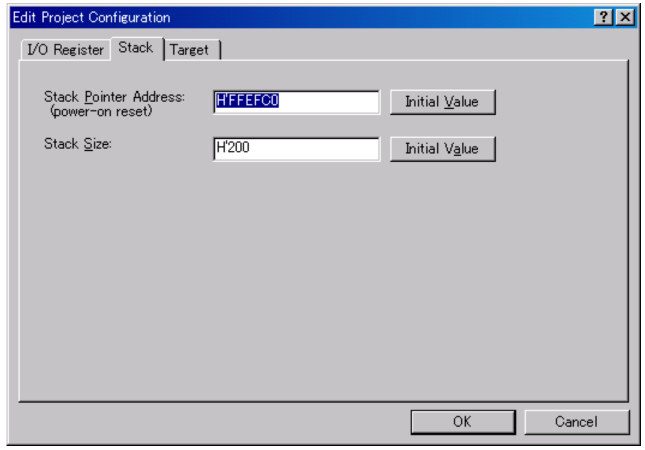


■再編集できるファイル

スタックサイズ設定ファイル：stacksct.h

【編集方法】

Edit Project Configuration ダイアログの Stack タブで、スタックポインタの初期アドレスとスタックのサイズを編集できます。



■備考

再生成、再編集は HEW2.0 以降でサポートしています。

7.1.2 メイクファイルの出力

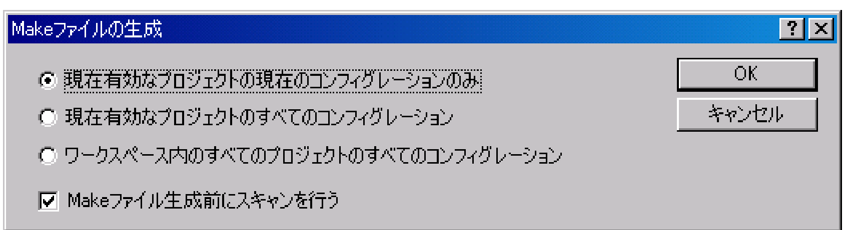
■説明

HEW では、現在のオプション指定状況を元にメイクファイルを生成することができます。

メイクファイルを使用すると、完全に HEW をインストールしていなくとも、現在のプロジェクトをビルドすることができます。HEW をインストールしていない相手にプロジェクトを送ったり、メイクファイルを含むビルド全体をバージョン管理したりする場合に便利です。

■メイクファイル出力方法

1. メイクファイルを生成するプロジェクトが現在のプロジェクトであることを確認してください。
2. プロジェクトをビルドするビルドコンフィグレーションが現在のコンフィグレーションであることを確認してください。
3. [ビルド>Makeファイルの生成]を選んでください。
4. 以下のダイアログが表示されるので、メイクファイルをどうのように生成するかを選択します。



■メイクファイル生成ディレクトリ

HEW は現在のワークスペースディレクトリ内に“ make ”サブディレクトリを作り、その中にメイクファイルを作成します。メイクファイルの名前は、現在のプロジェクトやコンフィグレーションに拡張子.mak を付けたものです。（例：debug.mak）。HEW により生成されたメイクファイルは、HEW をインストールしたディレクトリ（例：c:¥hew2）にある実行ファイル HMAKE.EXE で実行できます。ただし、ユーザが変更したメイクファイルは実行できません。

■メイクファイル実行方法

1. コマンドウインドウを開き、メイクファイルが生成された“ make ”ディレクトリに移動してください。
2. HMAKEを実行してください。コマンドラインはHMAKE.EXE　<メイクファイル名>です。

■備考

本機能は HEW1.1 以降でサポートしています。

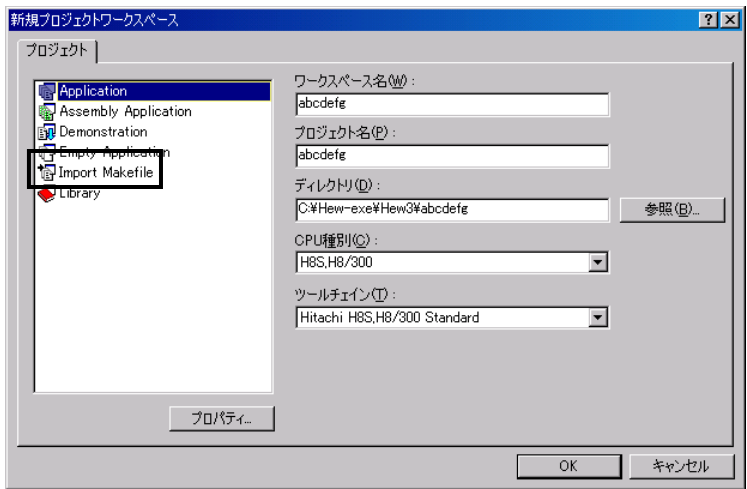
7.1.3 メイクファイルの入力

■説明

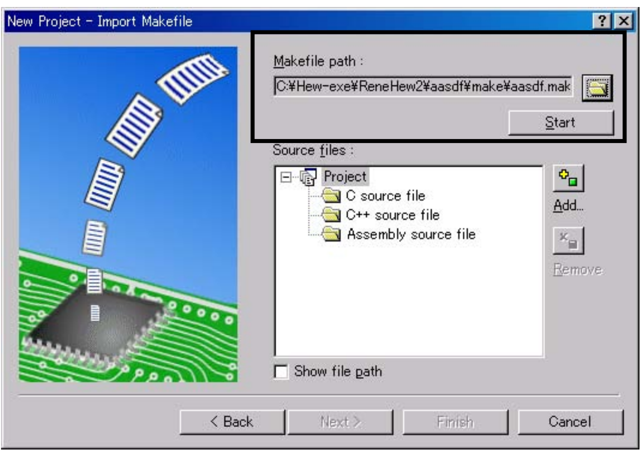
HEW では、HEW で作成したメイクファイルや UNIX 環境で使用したメイクファイルを入力することができます。
メイクファイルからプロジェクトのファイル構成を自動で取得することができます。
（オプション指定などは取得できません。）
これにより、コマンドラインから HEW への移行が容易になります。

■メイクファイルの入力方法

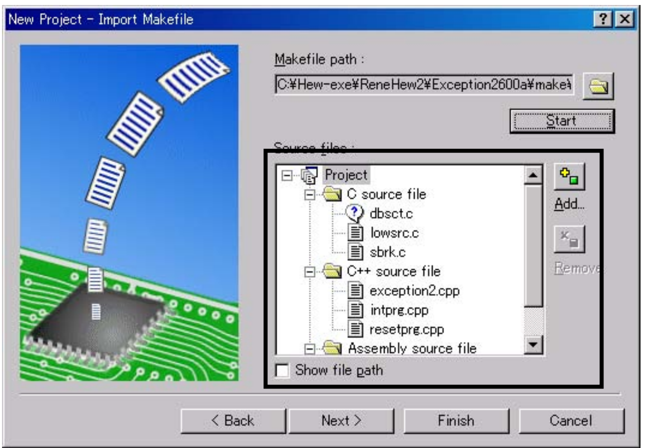
1. 新規ワークスペース作成時、新規プロジェクトワークスペースダイアログのプロジェクトタイプの中から Import Makefileを選択します。



2. New Project-Import MakefileダイアログのMakefile pathに入力するメイクファイルのパスを設定しStartボタンを押下します。



3. Source filesにメイクファイルのソースファイルの構成が表示されます。このとき、 が表示されたファイルはメイクファイルの内容を解析した結果、ファイルの実体がない状態を示しています。このファイルはプロジェクトに追加しません。（無視されます）



4. その後はウィザードに従い、CPUやオプション設定などを設定後、ワークスペースをオープンし、プログラム開発作業を開始できます。

■備考

本機能は HEW3.0 以降でサポートしています。

7.1.4 カスタムプロジェクトタイプの作成

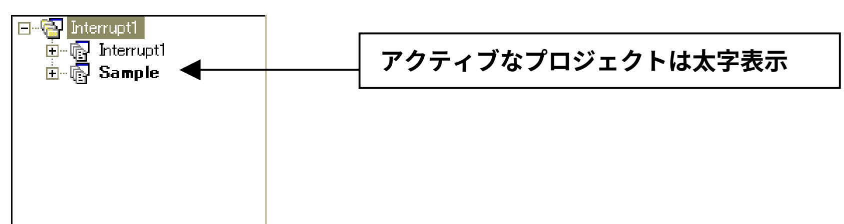
■説明

カスタムプロジェクト作成機能を使うと、あるユーザが作成したプロジェクトを雛型として、他のユーザが別なマシンでプログラム開発することができます。

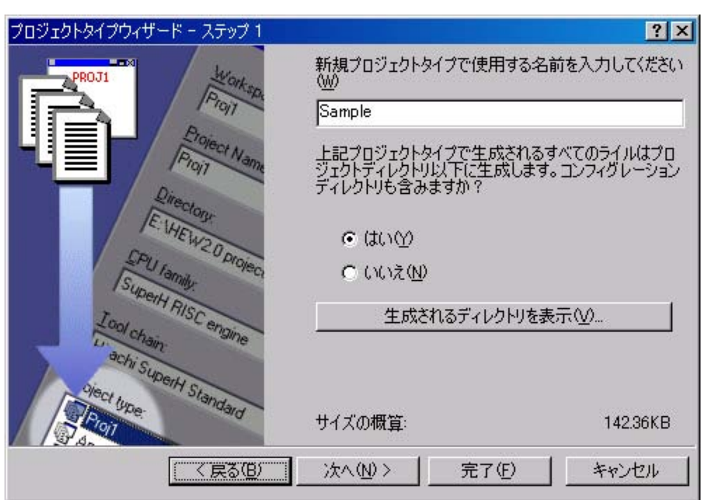
雛型として保存できる情報は、プロジェクトファイル構成やビルドオプションやデバッガ設定状況などのプロジェクト内容をすべて保存することが可能です。

■プロジェクトタイプ保存方法

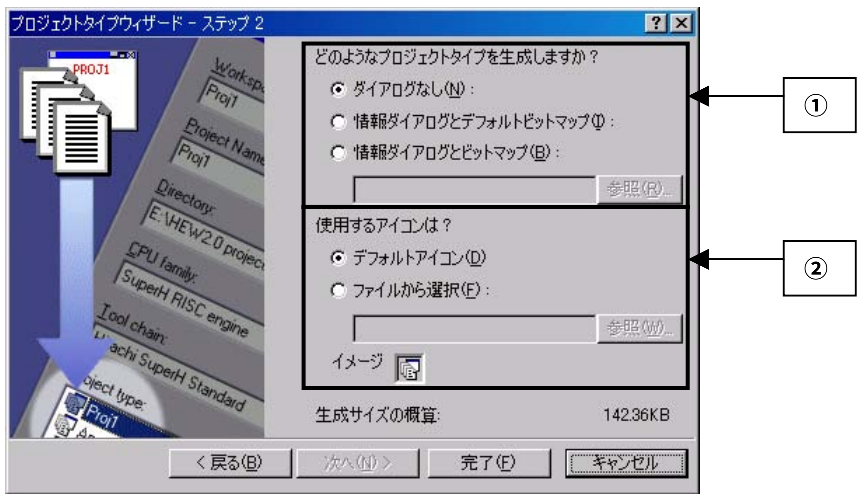
1. ワークスペースをオープンしている状態で、アクティブになっているプロジェクトが保存されるプロジェクトになるので、対象プロジェクトをアクティブにします。プロジェクトをアクティブにするには[プロジェクト->アクティブプロジェクトに設定]でプロジェクトを選択します。



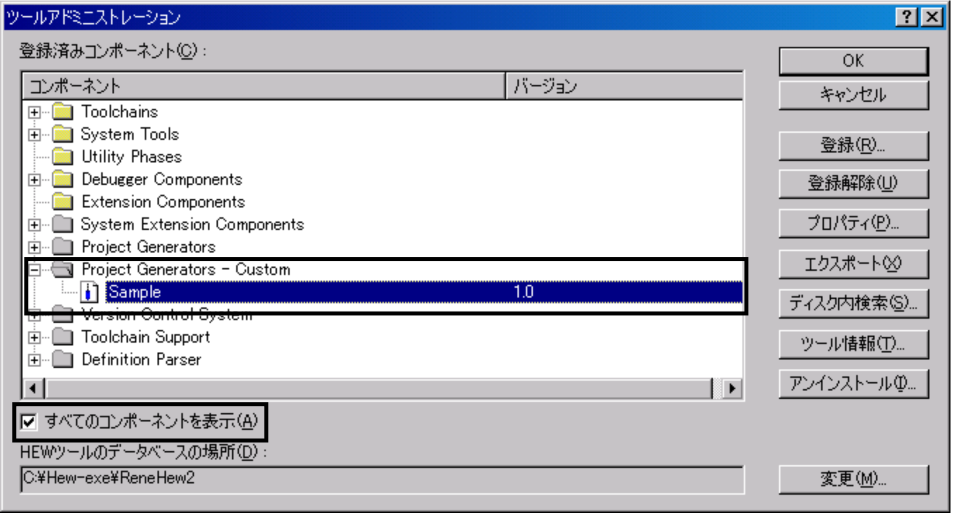
2. [プロジェクト->テンプレートの作成...]により以下のプロジェクトタイプウィザードを表示します。雛型として使用するプロジェクトタイプの名前を決めて、ビルド後の実行ファイルなどを含むコンフィグレーションディレクトリについても雛型とするか選択します。
ここで完了ボタンを押下し、プロジェクトタイプウィザードを終了しても構いません。



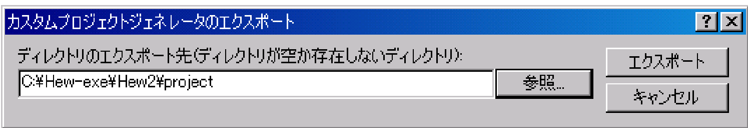
3. [プロジェクトタイプウィザード – ステップ1]で[次へ]ボタンを押下すると以下のウィザードが表示されます。以下の①ではプロジェクトタイプの雛型をオープンするときに、プロジェクトの情報やビットマップを表示するか選択します。
   また、②ではプロジェクトタイプのアイコンをユーザ指定のアイコンに変更することもできます。その後、完了ボタンを押下します。
   これらの指定は必須ではありません。



4. これでカスタムプロジェクトジェネレータというプロジェクトタイプの雛型が作成できました。この雛型を他のマシンで使用するときは、[ツール->アドミニストレーション]を選択し以下のダイアログを表示します。
   以下の[すべてのコンポーネントを表示]をチェックすると、[Project Generators - Custom]という項目が表示されます。
   表示されたら、作成したプロジェクトタイプをクリックしエクスポートボタンを押下します。



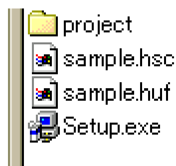
5. そうすると、以下のダイアログが表示されるのでカスタムプロジェクトジェネレータを出力するディレクトリを指定します。ディレクトリの中身は空である必要があります。
   これで、プロジェクトタイプの保存は終了です。



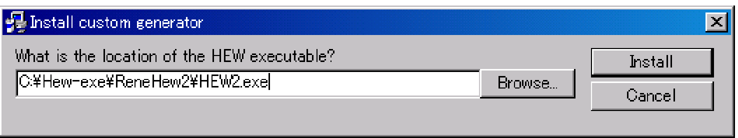
■カスタムプロジェクトジェネレータのインストール

上記の[プロジェクトタイプ保存方法]で作成したカスタムプロジェクトジェネレータを他のマシンにインストールする方法を説明します。

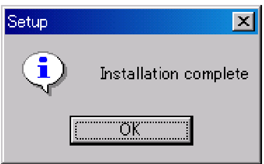
1. [プロジェクトタイプ保存方法]の5.で作成したディレクトリに、以下のようにインストール環境が作成されています。
   （インストール環境ディレクトリ）



2. 上記のインストール環境を他のマシンにコピーしインストールします。
   Setup.exeを実行すると、以下のダイアログが表示されるのでHEW2.exeのインストール場所を指定し、Installボタンを押下します。
   （ディレクトリ例：c:¥Hew2¥HEW2.exe）



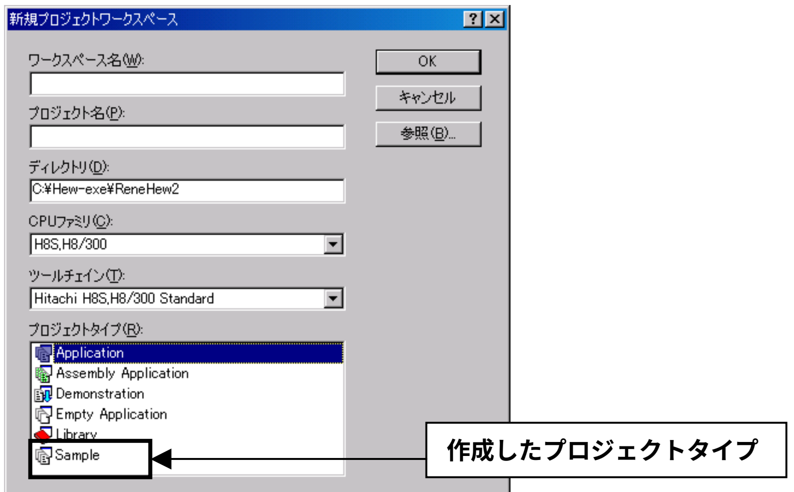
3. これで、環境の構築は終了です。



■カスタムプロジェクトジェネレータの使用例

インストールしたカスタムプロジェクトジェネレータの使用例を以下に示します。

1. HEWを起動し、[ようこそ！]ダイアログで[新規プロジェクトワークスペース]の作成を選択すると[プロジェクトタイプ]にインストールしたプロジェクトタイプが追加されているので、クリック後OKボタンを押下します。これで、新しいプロジェクトでも保存したプロジェクトの雛型を利用し、プログラム開発をすることができます。



■備考

本機能は HEW2.0 以降でサポートしています。

7.1.5 マルチ CPU 機能

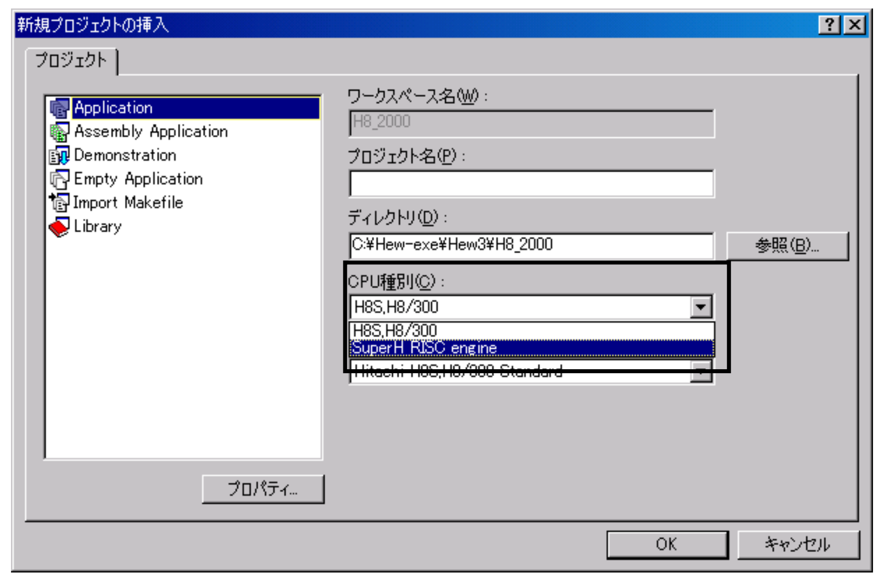
■説明

ワークスペースに新規プロジェクトを挿入するとき、別な CPU 種別の CPU を挿入することができます。
これにより SH と H8 のプロジェクトが 1 つのワークスペースで管理することができます。

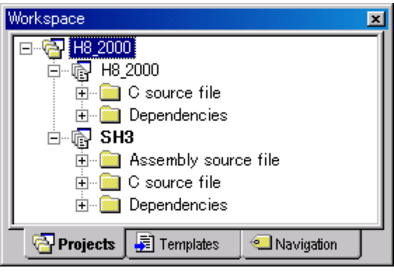
■別 CPU ファミリ挿入の例

1. H8(SH)のプロジェクトを開いているとき、[プロジェクト->プロジェクトの挿入]をクリックし、プロジェクトの挿入ダイアログで新規プロジェクトを選択後にOKボタンを押下します。

2. 以下の新規プロジェクトの挿入ダイアログが出現します。ここでプロジェクト名とCPU種別でSH(H8)を選択後にOKを押下すると、別なCPU種別をワークスペースに混載することができます。



3. このような手順で以下のように1つのワークスペースにSHとH8のプロジェクトを混載させることができます。



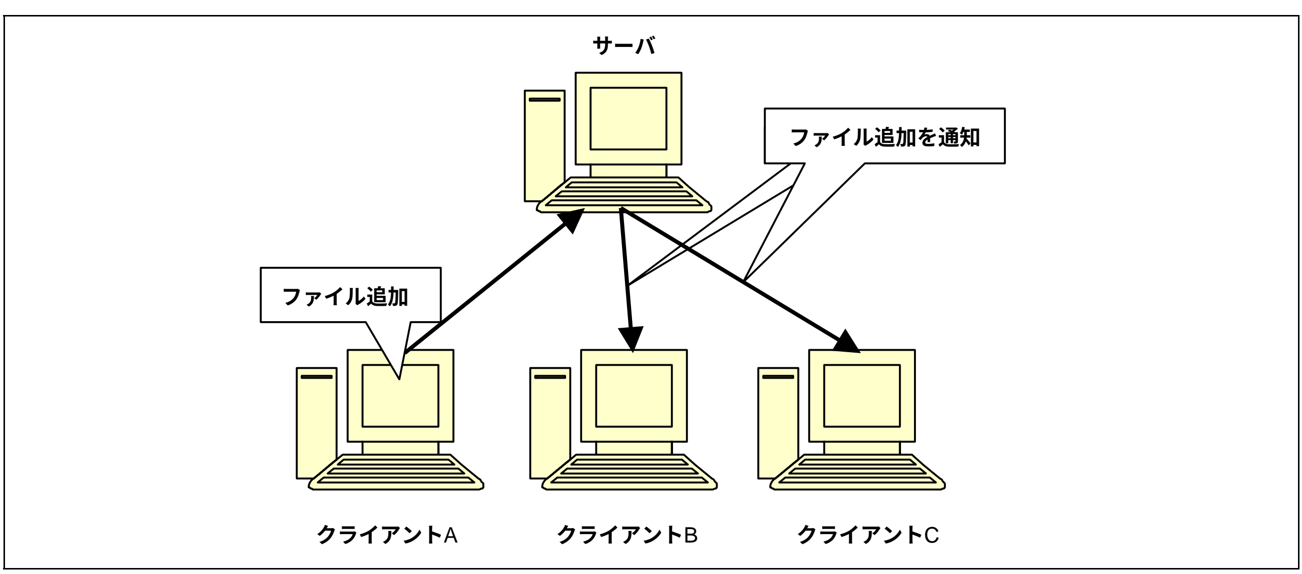
■備考

本機能は HEW3.0 以降でサポートしています。

7.1.6 ネットワーク機能

■説明

HEW は、ネットワークを介してワークスペースやプロジェクトを共有することができます。
よって、ユーザは共有したプロジェクトを同時に操作してお互いの変更を知ることができます。
このシステムは 1 つのコンピュータをサーバとして使います。
これにより、たとえば、クライアントがプロジェクトに新規ファイルを追加すると、サーバマシンに通知され、サーバがすべてのクライアントに追加を通知することができます。
さらに、ユーザにアクセス権を持たせることができ、プロジェクトやファイルに対する書き込みの権限を指定することができます。

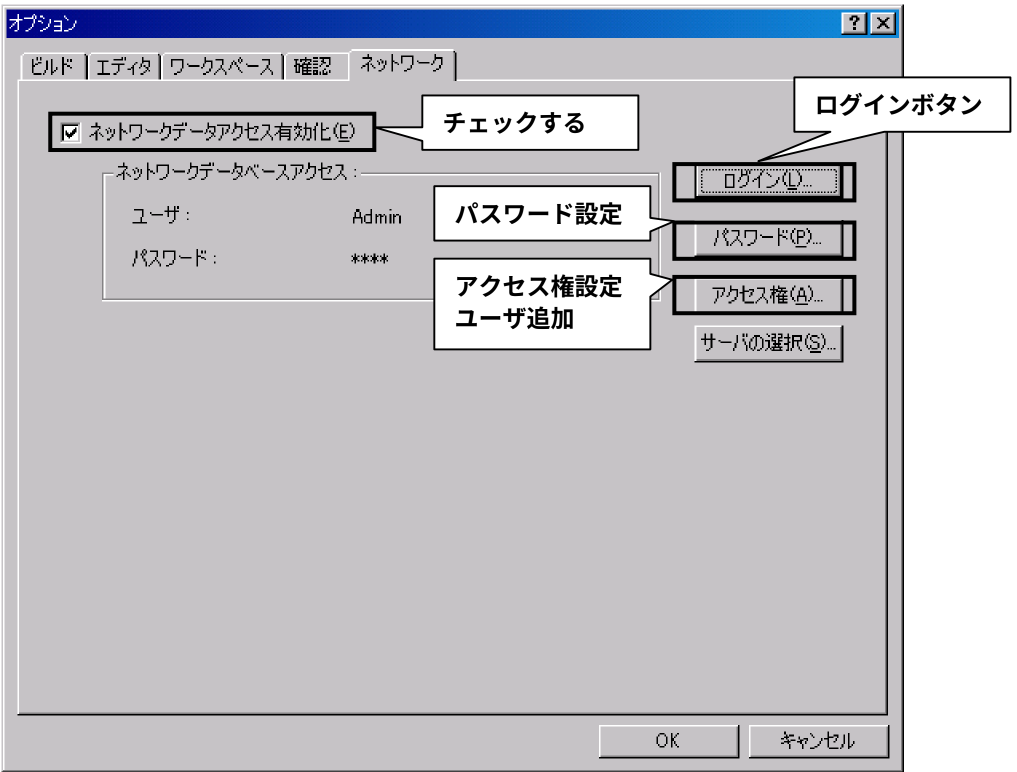


■ネットワークアクセスの設定

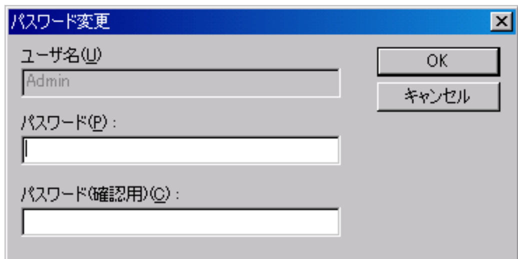
1. [ツール->オプション]を選んだ後、ネットワークタブを選択し、ネットワークデータアクセス有効化チェックボックスをチェックします。
2. アドミニストレータが追加されます。初期状態ではパスワードがないので、指定する必要があります。アドミニストレータは最高レベルのアクセス権を持ちます。
3. パスワード...ボタンをクリックし、アドミニストレータのパスワードを設定します。
4. OKボタンを押下します。これでネットワークアクセスが可能になります。

RENESAS

オプションダイアログ/ネットワークタブ



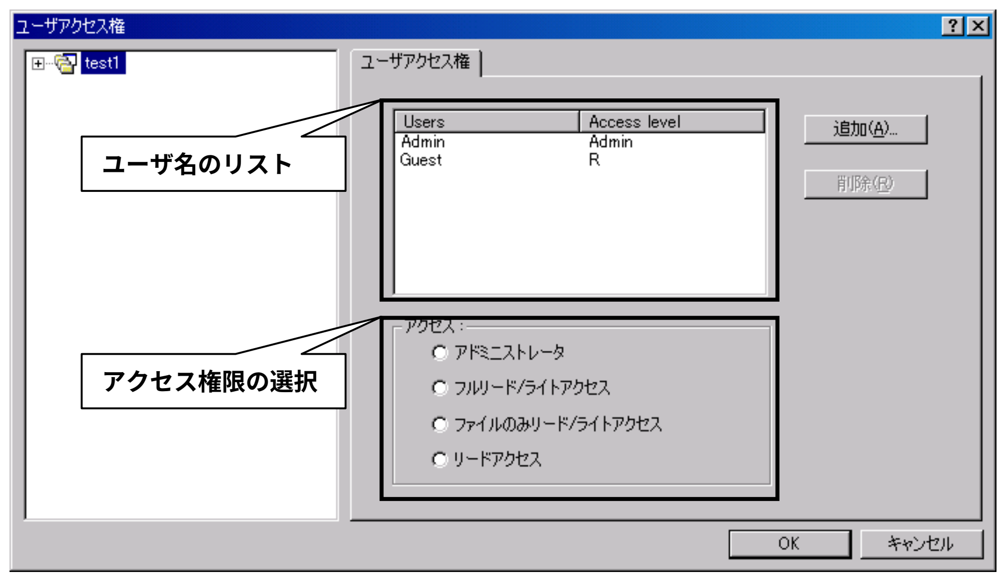
パスワード変更ダイアログ



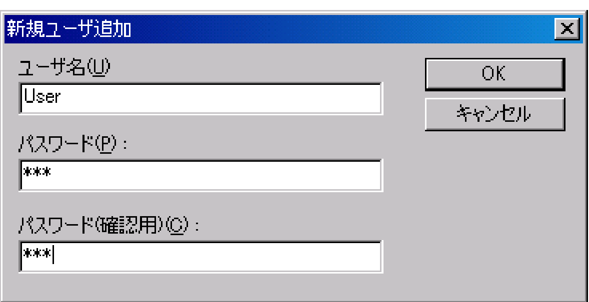
■新規ユーザの追加

デフォルトではアドミニストレータとゲストが追加されていますが、新たなユーザを登録することもできます。

1. 前ページのログイン...ボタンを押下し、アドミニストレータのアクセス権を持つユーザでログインします。
2. アクセス権...ボタンをクリックし、以下のユーザアクセス権ダイアログを表示します。



3. 追加...ボタンをクリックし、新規ユーザ追加ダイアログを表示します。
4. 新たなユーザ名とパスワードを登録します。（パスワード指定は必須）



5. するとユーザのリストに新たなユーザ名が追加されます。ここで、ユーザ名を選択し、このユーザの権限を決定します。
6. OKボタンを押下すると設定が反映されます。

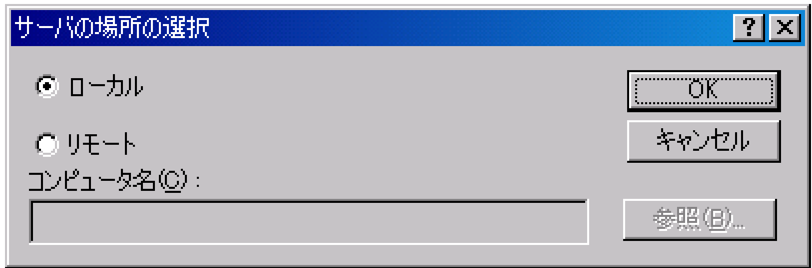
RENESAS

■サーバマシンの決定

どのマシンをサーバにするか決定します。自分のマシンをサーバにする場合は、何もする設定する必要はありません。

他のマシンをサーバにする場合は、オプションダイアログのサーバの選択...ボタンを押下し、以下のダイアログでリモートを選んだ後に、コンピュータ名を指定します。

OK ボタンを押下すると、設定が反映されます。



■備考

本機能は HEW3.0 以降でサポートしています。

本機能を使用すると HEW の性能が低下します。

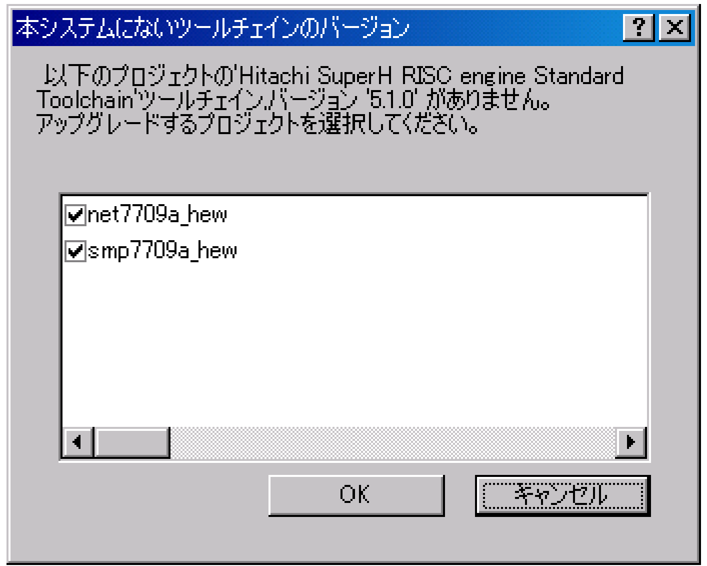
7.1.7　　古い HEW から転用する場合

統合化環境でのコンパイラバージョンの指定方法について説明します。バージョンの指定は統合化環境をアップグレードすることで可能になります。

旧バージョン（例 HEW1.1、SHC 5.1B）で作成したワークスペースを新バージョン（例 HEW3.0、SHC 8.0）でオープン時、以下のダイアログボックスを表示します。

(1)　アップグレード対象プロジェクトの確認

アップグレード対象のプロジェクト名を確認します。



High-performance Embedded Workshop

(2)　バージョンの指定

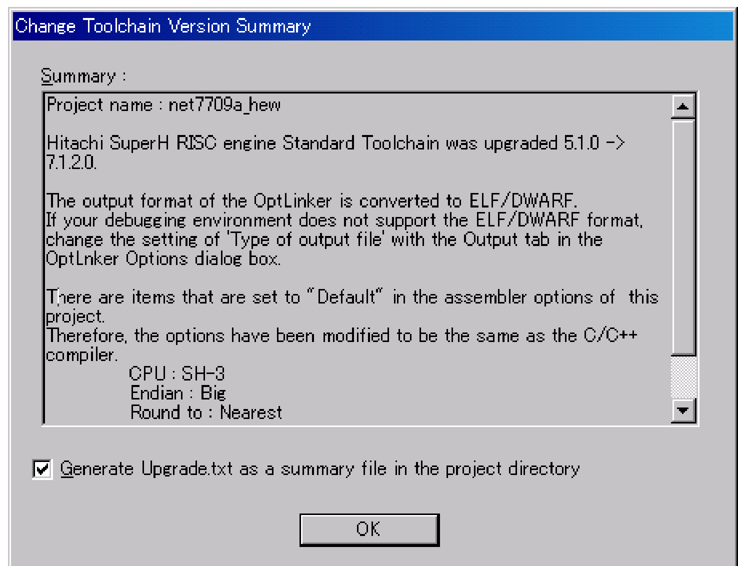
アップグレードできるバージョンを選択します。

ツールチェインのバージョン変更ダイアログボックス

RENESAS

(3) 確認メッセージ

C/C++コンパイラ Ver6.0 以降、出力するオブジェクトのファイル形式が ELF/DWARF のみをサポートしています。アップグレード時に、ELF/DWARF フォーマットに変換するため、現在使用中のデバック環境が ELF/DWARF フォーマットをサポートしていない場合は、アップグレード後に、デバック環境がサポートするフォーマット形式に変換してください。



確認メッセージダイアログ

(4) 標準ライブラリツール構築オプション

アップグレードをすると標準ライブラリ構築ツールの[標準ライブラリ]/[モード]/[モード]がライブラリファイル作成(常に)に変更されるので注意が必要です。

7.1.8 HIM システムから転用する場合

HEW システムに添付されている HimToHew ツールを使用し、HIM から HEW へ変換することが可能です。

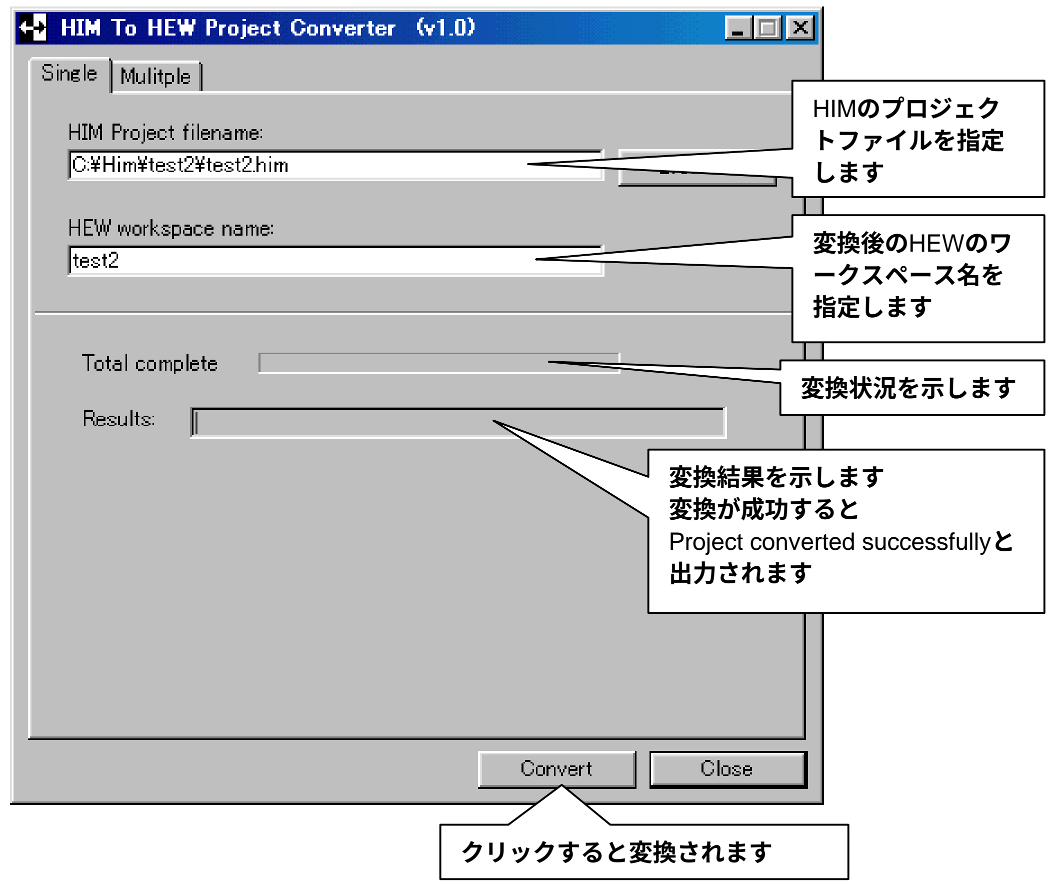
Windows®[スタートメニュー]の[プログラム(P)]の[Renesas High-performance Embedded Workshop]から[Him To Hew Project Converter]を選択します。

Single タブと Multiple タブがあります。

1 つの HIM プロジェクトから HEW ワークスペースおよび HEW プロジェクトを作成する場合は Single タブを選択します。

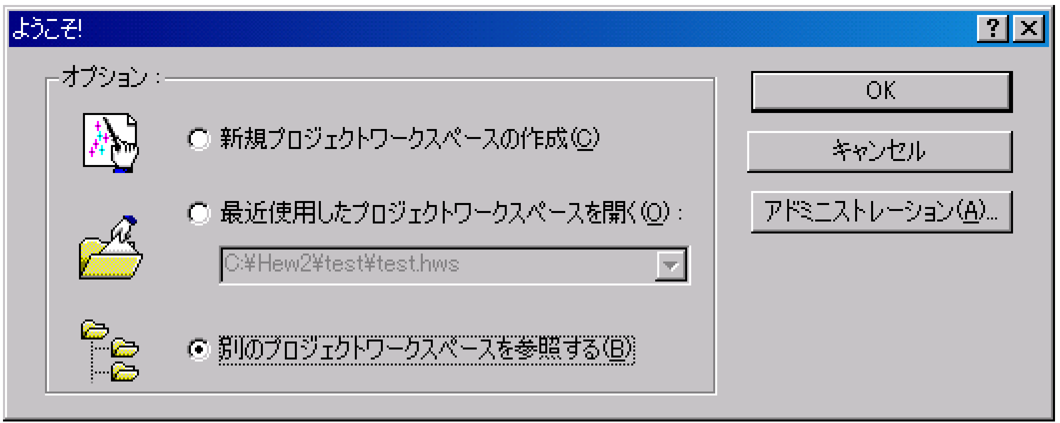
複数の HIM プロジェクトを HEW プロジェクトに変換し、HEW ワークスペースに一括登録する場合は、Multiple タブを選択します。

(1) Single タブ

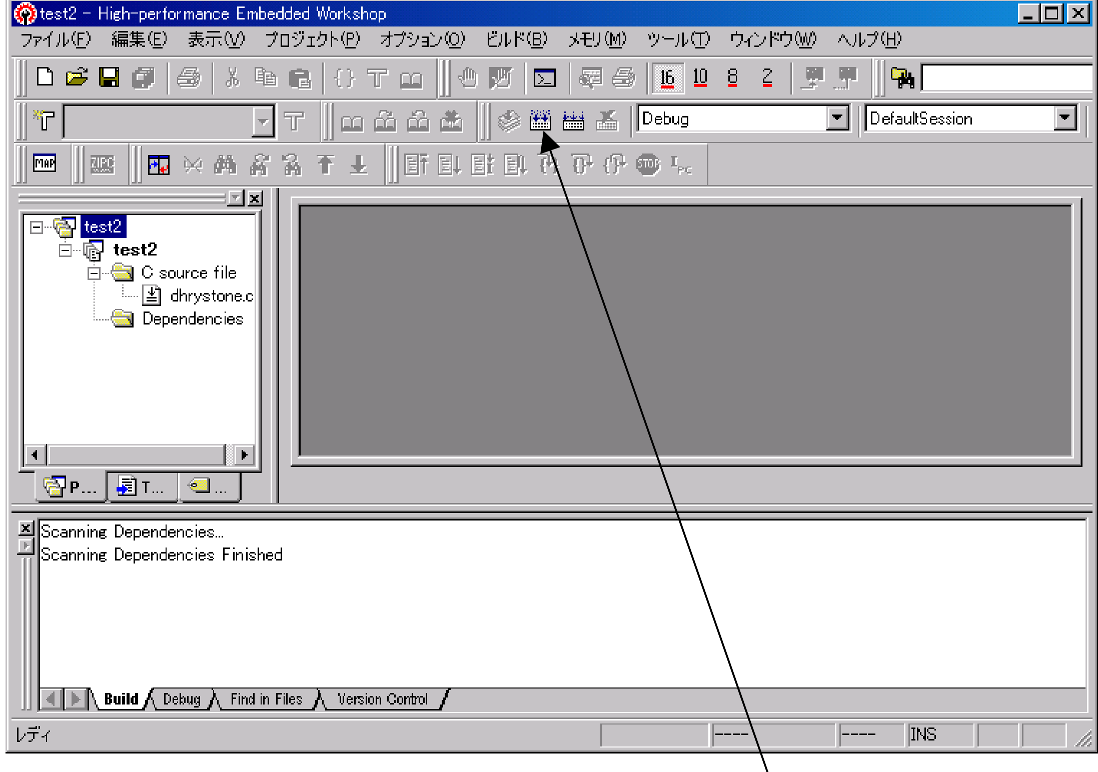


次に HEW を立ち上げます。

別のプロジェクトワークスペースを開くを選択し、[OK]ボタンをクリックし、変換した HEW プロジェクトを指定します。



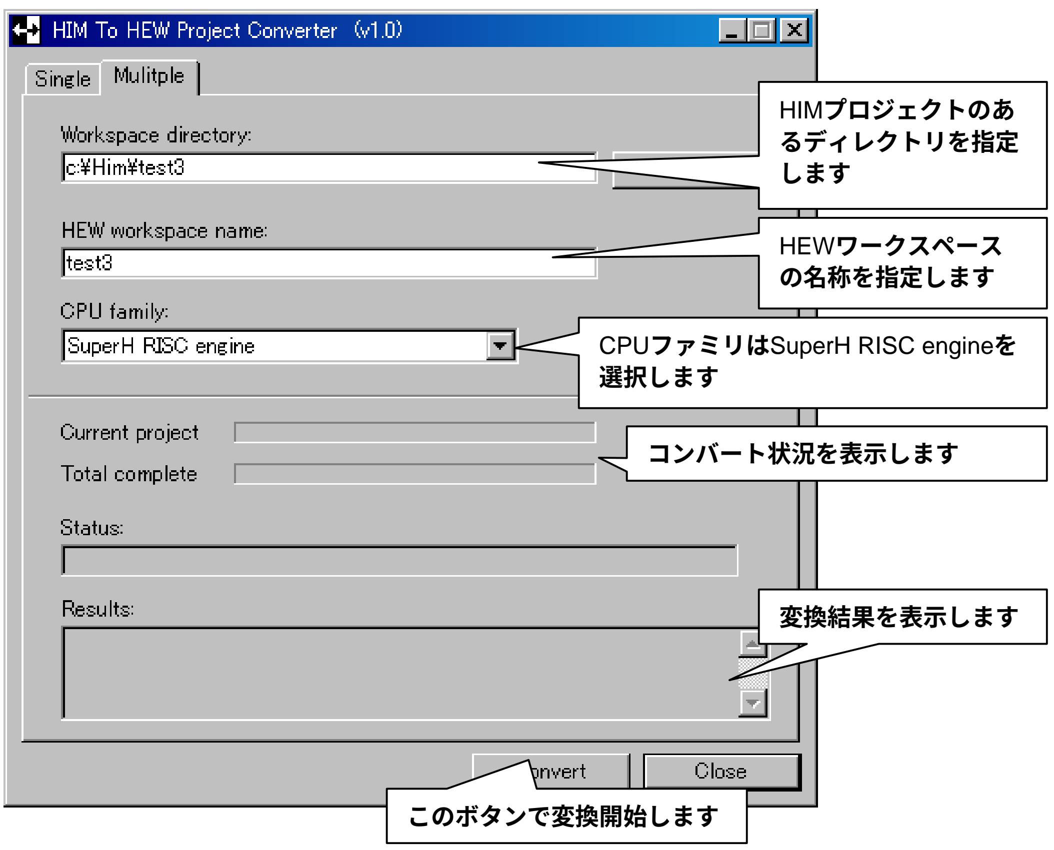
次のように HEW プロジェクトが開かれました。



[ビルド->ビルド]を指定するとビルドを実行します。コマンドメニューでは**ここ**をクリックします。

(2)　Multiple タブ

複数の HIM プロジェクトを HEW プロジェクトへコンバートします。



変換後は Single タブと同様に HEW を起動させ、変換した HEW ワークスペースをビルドします。

RENESAS

7.1.9 サポート CPU の追加

■説明

HEW は I/O 定義やベクタテーブルのファイルを自動生成できますが、HEW リリース後にリリースされた新 CPU などは対応していません。

このように対応する CPU がない場合は、DeviceUpdater というツールを入手することにより、HEW を新 CPU に対応させることができます。

また、自動生成ファイルをバグ対応版に更新することもできます。

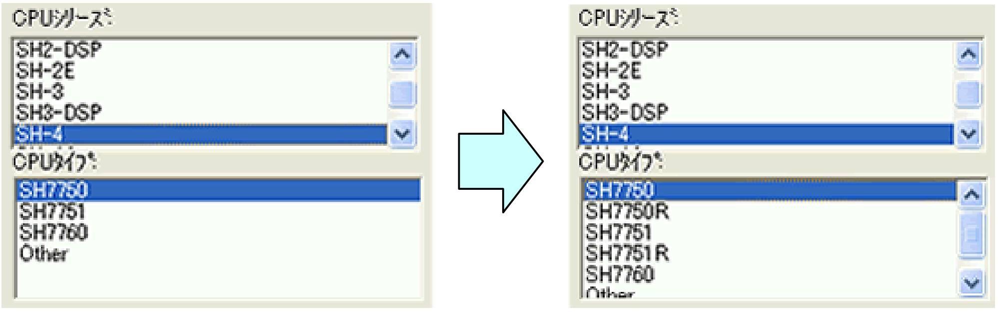
■DeviceUpdater 入手方法

ルネサスのウェブサイトからダウンロードできます。

あわせて注意事項等も参照してください。

■DeviceUpdater 実行結果

以下のように CPU タイプが追加されます。



■備考

HEW2.2 以降が必要です。

7.2 シミュレーション編

7.2.1 擬似割り込み

■説明

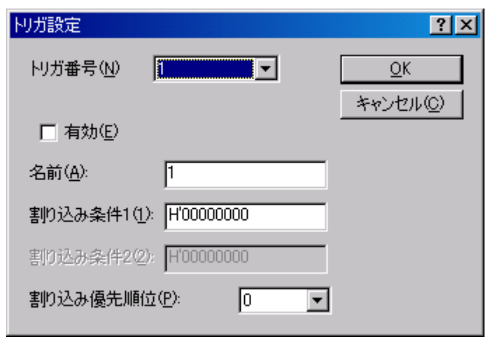
この擬似割り込みはボタンを、ある割り込み要因に見立てて、ボタンを押下することにより手動で擬似的な割り込みを発生させることができます。

ボタンには割り込み優先度と割り込み条件を指定します。

■使用方法

1. [表示->CPU->トリガ]を選択すると、以下のビューが表示されます。

2. このビュー上でマウスを右クリックし[設定…]を押下します。そうするとトリガ設定ダイアログが表示されます。
   ここで、有効チェックボックスをチェックすると、トリガ番号1の割り込みが有効になります。
   その他、割り込み名や割り込みの優先順位や割り込み条件（ベクタ番号）を登録します。
   これで、トリガ番号1のボタンがアクティブになります。



3. これで設定は終了です。プログラム実行中に上記で設定したボタンを押下すると、該当のベクタテーブルでプログラムが停止させることができます。

■備考

本機能は HEW2.1 以降でサポートしています。

RENESAS

7.2.2 ブレークポイントの便利機能

■説明

HEW のブレークポイント機能は、通常のブレークに加えてブレーク条件成立時に以下の便利な機能があります。
ファイル入力
ファイル出力
割り込み

■ブレークポイントビューの表示方法

HEW2.2 以前：表示->コード->ブレークポイント
HEW3.0 以降：表示->コード->イベントポイント
【注】HEW3.0 以降の場合は、ブレークポイントビューの Software Event タブをクリック

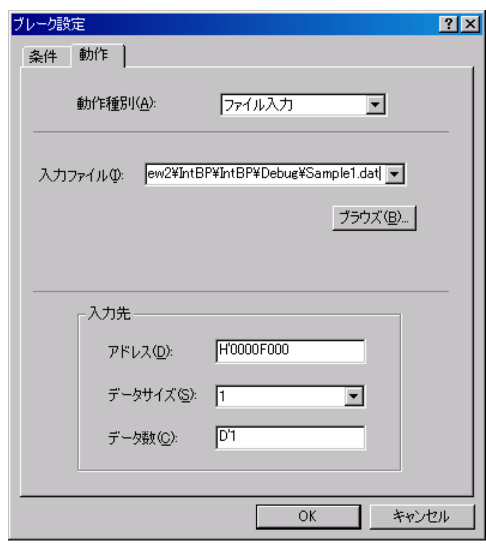
■ファイル入力設定例

ブレークポイントビュー上で右クリック後に[設定…]を押下し、以下のブレーク設定ダイアログを表示します。以下のように、PC ブレークポイントを使い PC が以下のアドレス時にブレーク条件が整う例とします。他のブレーク種別でも設定方法は同様です。

次に動作タブをクリックし、[動作種別]でファイル入力を選択し、入力ファイル名、入力先のアドレスなどを指定し OK ボタンを押下します。

(条件タブ)

(動作タブ)



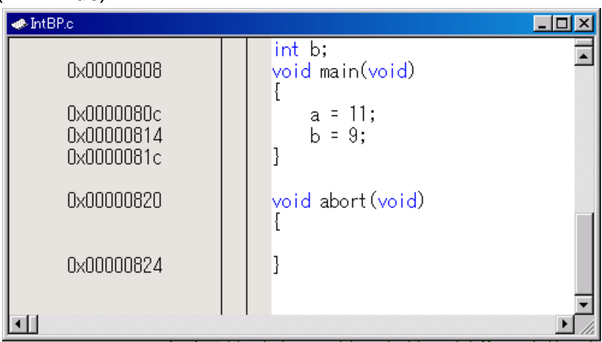
■ファイル入力動作例

次に実際の動作例を見てみましょう。

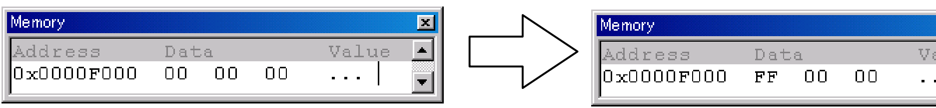
上記設定によりブレークポイントが[H'00000814]になっており、入力するファイルには[H'FF]が入っています。

Go コマンドなどでプログラム実行させます。

(ソース例)



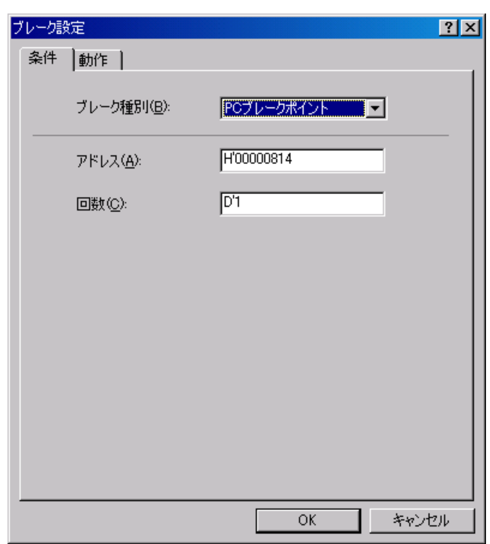
そうすると、PC の[H'00000814]を通過するときに条件が成立し、以下のように H'F000 番地のメモリ内容が変化することを確認できます。



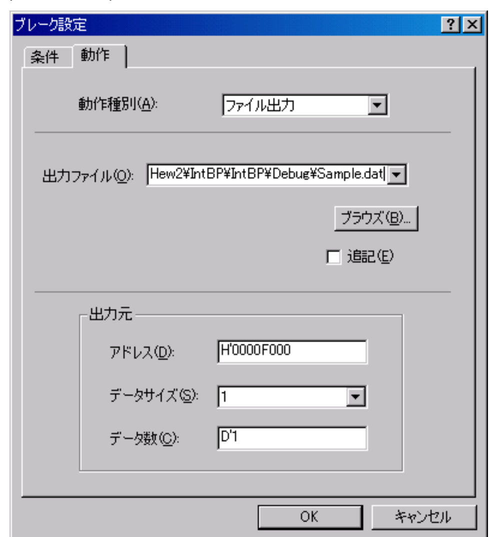
■ファイル出力設定例

ブレーク設定ダイアログの出力方法は、ファイル入力と同様です。ファイル出力の場合も、PC ブレークポイントを使い PC が以下のアドレス時にブレーク条件が整う例とします。次に動作タブをクリックし、[動作種別]でファイル出力を選択し、出力ファイル名、出力元のアドレスなどを設定し OK ボタンを押下します。

(条件タブ)



(動作タブ)



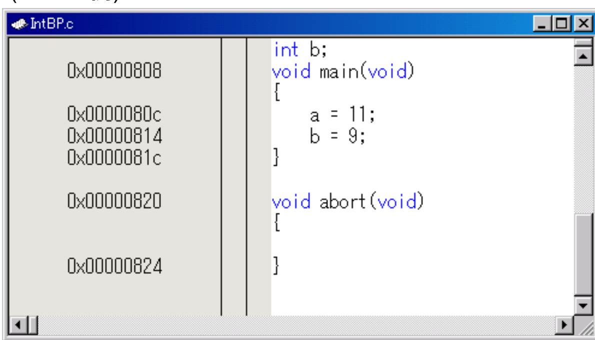
■ファイル出力動作例

次に実際の動作例を見てみましょう。

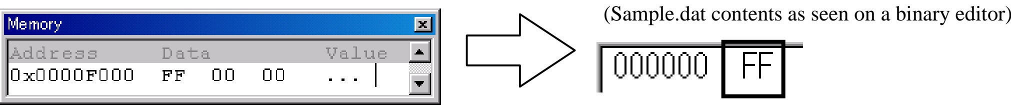
上記設定によりブレークポイントが[H'00000814]になっており、H'F000 番地の内容は[H'FF]です。

Go コマンドなどでプログラム実行させます。

(ソース例)



そうすると、PC の[H'00000814]を通過するときに条件が成立し、以下のように H'F000 番地のメモリ内容をファイルに出力することを確認できます。



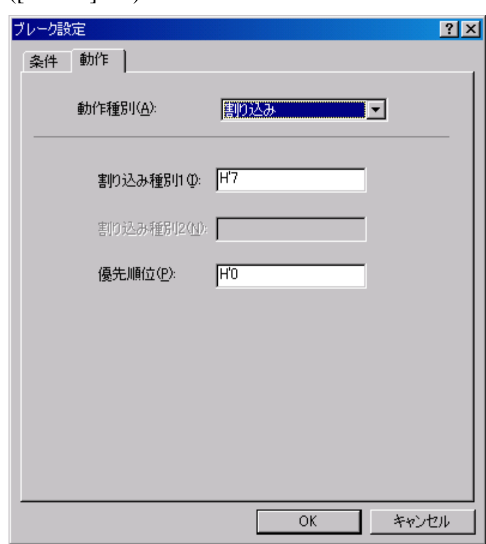
■割り込み設定例

ブレーク設定ダイアログの出力方法は、ファイル入力と同様です。以下のように、PC ブレークポイントを使い PC が以下のアドレス時にブレーク条件が整う例とします。他のブレーク種別でも設定方法は同様です。

次に動作タブをクリックし、[動作種別]で割り込みを選択し、割り込みの優先順位や割り込み種別（ベクタ番号）を[7]と指定し OK ボタンを押下します。

([Condition] tab)

([Action] tab)



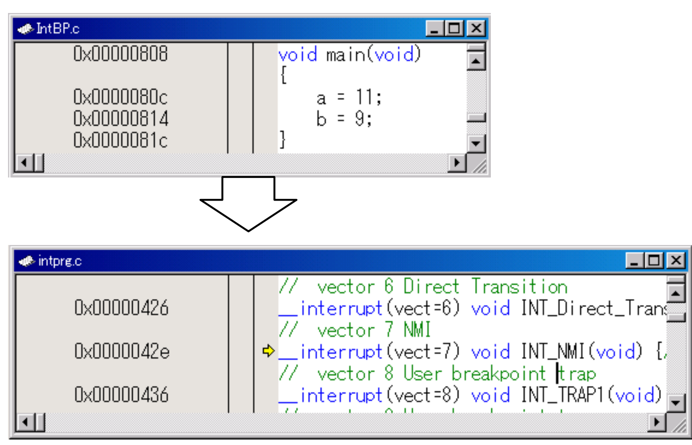
■割り込み動作例

次に実際の動作例を見てみましょう。

上記設定によりブレークポイントが[H'00000814]になっている状態で、Go コマンドなどでプログラム実行させせます。

PC が[H’00000814]になると、以下のようにベクタ番号 7 の NMI 割り込みが発生することを確認できます。

(Source code fragment)



RENESAS

7.2.3 カバレジ機能

■説明

HEW では、ユーザが指定したアドレス範囲について、命令実行中に命令カバレジ情報を収集できます。命令カバレジ情報を利用することで各命令の実行状態を観察できます。さらにプログラムのどの部分が未実行であるかを容易に特定できます。

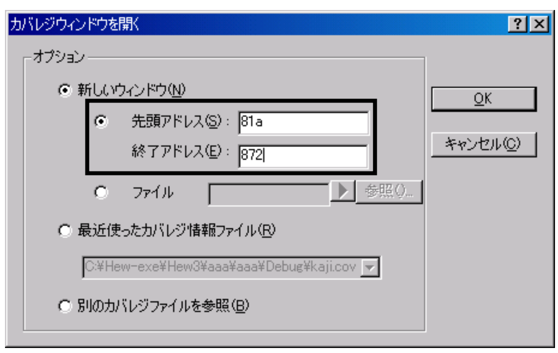
■[カバレジウインドウを開く]ダイアログの表示方法

表示->コード->カバレジ

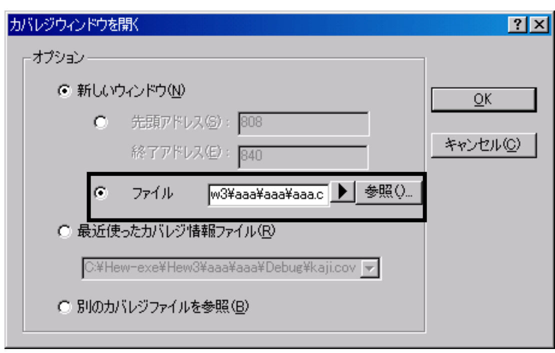
■新規のカバレジ情報収集方法

1. [カバレジウインドウを開く]ダイアログを表示し、 [新しいウインドウ]を選択し情報を収集したい箇所の先頭アドレスと終了アドレスを入力します。また、HEW3.0からは情報を収集したい箇所をCまたはC++ソースファイル名で指定することもできます。
   これらを指定した後にOKボタンを押下します。

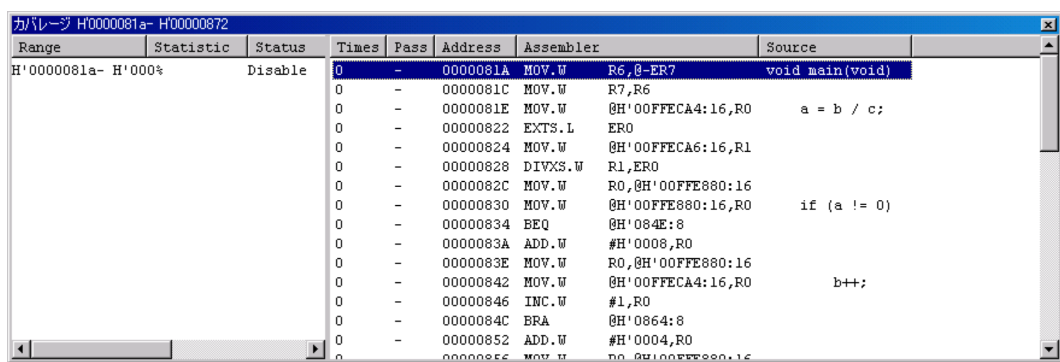
(アドレス指定)



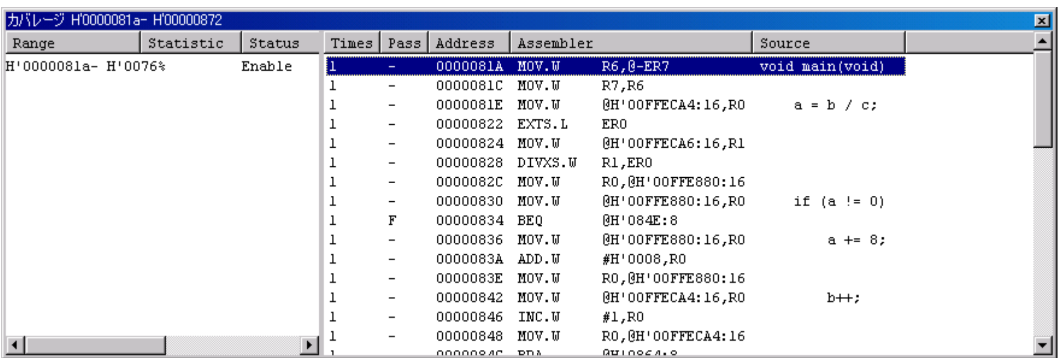
(ファイル名指定) *HEW3.0以降サポート



2. OKボタンを押下すると以下のカバレジビューが表示されます。
   ここの右側のビューで右クリックし有効を選択すると、カバレジが有効になります。

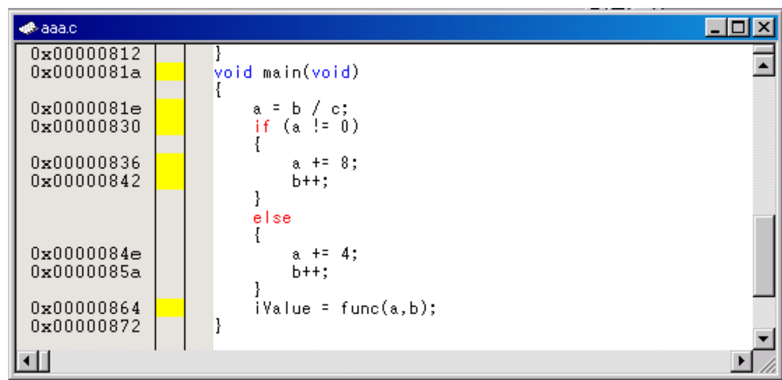


3. 次に実際にプログラムを実行してみましょう。そうすると、カバレジビュー右側のTimes項目が1に変化している箇所があります。これは該当アドレスの命令を実行したことを意味します。
   また、左側には該当アドレス範囲のC0カバレジ値が表示されます。



【注】 カバレジビューの左側は HEW3.0 以降の場合のみ存在します。

4. カバレジはカバレジビュー以外にも、以下のようにエディタの左側のカラムでソース行を通過したことを確認できます。



■カバレジ情報の保存方法

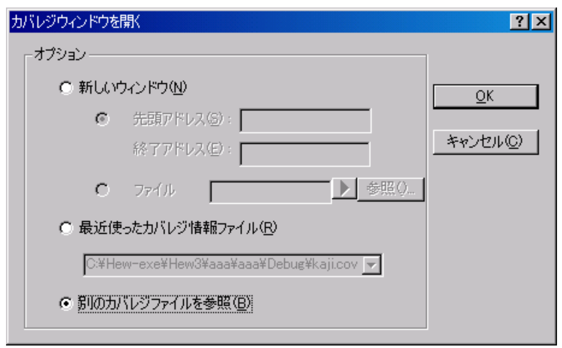
カバレジ情報を保存するには、カバレジビューの右側で右クリックし拡張子*.cov で保存します。

■既存カバレジ情報使った情報収集

カバレジ情報収集作業は一度の実行で全プログラムを網羅できることは、あまりありません。

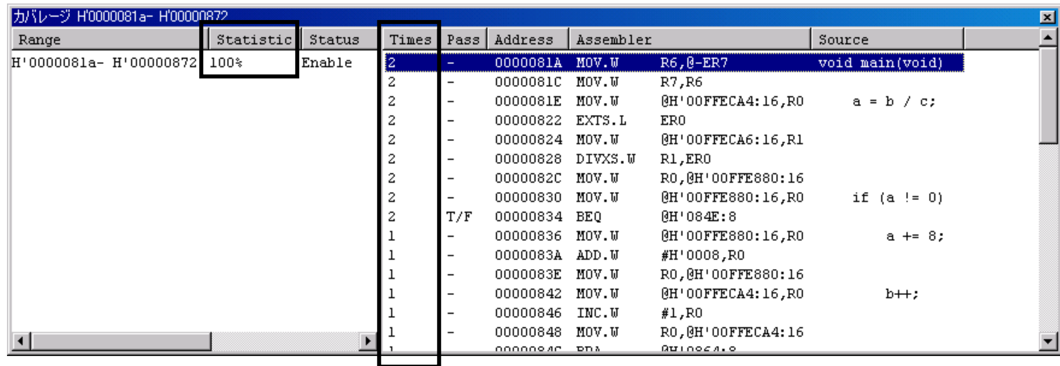
そのために繰り返しテストケースを変更しながら網羅率を上げることと考えられます。

そのようなときは、[カバレジ情報の保存方法]で保存したファイルを[カバレジウインドウを開く]ダイアログの[最近使ったカバレジ情報ファイル]、または[別のカバレジファイルを参照]を選択し、OK ボタンを押下します。



カバレジビューがオープンされるので、新しい条件で再度プログラムを実行します。

そうすると以下のように、カバレジビューとエディタに今回実行した結果の実行回数、C0 カバレジ値などが加えられます。



RENESAS

7.2.4　　ファイル入出力

■説明

HEW では I/O シミュレーション機能を使って、擬似的に標準入出力を行っていました。
しかし、以下に示すファイルを入れ替えると、実際のファイルで入出力処理を行うことができます。

■ファイル入手方法

ルネサスのウェブサイトから、シミュレータ・デバッガ用ファイル操作低水準インタフェースルーチン環境作成ファイルをダウンロードしてください。

■環境作成方法

（1）　HEWでプロジェクトを作成します。
プロジェクトタイプには、「Application」・「Demonstration」のいずれかを選択してください。
すると、作成したプロジェクト配下にいくつかのファイルが自動的に作成されます。
☆プロジェクトタイプに「Application」を選択した場合は、プロジェクト作成Step3において[Use I/O Library]にチェックをしてください。
また、[Number of I/O Stream]の数は、"実際に操作するファイル数＋３(標準入出力ファイル)以上に設定して下さい。
（2）　作成されたファイルのうち「lowsrc.c」・「lowlvl.src」をそれぞれ差し替えます。*1
（3）　"C:¥Hew2¥stdio"ディレクトリを作成します。*2
（4）　再ビルドを行うことでファイル入出力の可能なシミュレータ・デバッガ環境が完成します。

【注】 *1　-lowsrc.c-
これらのファイルは、SH・H8 **共用のファイルです。**
それぞれプロジェクト内にある「lowsrc.c**」と置き換えてください。**
-lowlvl.src-
このファイルは、各 CPU **によって異なります。**
それぞれプロジェクトを作成した CPU **に対応するフォルダに格納されている「**lowlvl.src**」と置き換えてください。**

*2　**本環境は、ファイル入出力処理に伴い、現在まで擬似的に行っていた標準入出力ファイルのオープン処理を実際に行います。**
よって、標準入出力用として実際にオープンする「stdin**」・「**stdout**」・「**stderr**」というファイルが初回実行時に自動生成されるようになっています。**
これらのファイルは、"C:¥Hew2¥stdio"**に生成されるように指定してあるため、上記(３)のようにディレクトリを作成する必要があります。本ディレクトリが存在しない場合、正しく動作しません。**
これらは、シミュレータ実行時にプロジェクト内にある「lowsrc.c**」の**_INIT_IOLIB()**でオープンされます。**
このオープン処理より、

```
stdin  = 0
stdout = 1
stderr = 2
```

のようにファイル番号が割り付けられます。

■使用例

以下の例のように printf などで、標準出力（stdout）に文字を出力します。

```
（プログラム例）

void main(void)
{
        printf("***** ID-1 OK *****¥n");
}
```

実行すると、あらかじめ作成しておいた“ c:¥Hew2¥stdio ”ディレクトリに stdout というファイルが作成され、その内容が以下になります。

```
（stdoutの内容）

***** ID-1 OK *****
```

■入出力先変更方法

入出力先を変更するには、lowsrc.c 内_INIT_IOLIB 関数の**ここ**を変更します。

```
void _INIT_IOLIB(void)
{
FILE *fp;

    for( fp = _iob; fp < _iob + _nfiles; fp++ ) /* ファイル型デー
     {
        fp->_bufptr = NULL;                     /* バッファへのポ
        fp->_bufcnt = 0;                        /* バッファカウン
        fp->_buflen = 0;                        /* バッファ長
        fp->_bufbase = NULL;                    /* バッファへのベ
        fp->_ioflag1 = 0;                       /* I/Oフラグ
        fp->_ioflag2 = 0;                       /* I/Oフラグ
        fp->_iofd = 0;                          /* ファイル番号
     }

    // 標準入出力用ファイルをオープン。
    // "stdin"・"stdout"・"stderr"は、ファイルが存在しなくても自動生
    // "stdin"は、実際は"r"でオープンしなくてはならないが、
    // 自動生成するために"w"でオープンした後I/Oフラグに"r"を設定して
   if(freopen( "C:¥¥Hew2¥¥stdio¥¥stdin", "w", stdin )==NULL)   /
        stdin->_ioflag1 = 0xff;                                 /
    stdin->_ioflag1  = _IOREAD;                                 /
    stdin->_ioflag1 |= _IOUNBUF;                                /
    if(freopen( "C:¥¥Hew2¥¥stdio¥¥stdout" "w", stdout )==NULL) /
        stdout->_ioflag1 = 0xff;                                /
    stdout->_ioflag1 |= _IOUNBUF;                               /
    if(freopen( "C:¥¥Hew2¥¥stdio¥¥stderr" "w", stderr )==NULL) /
        stderr->_ioflag1 = 0xff;                                /
    stderr->_ioflag1 |= _IOUNBUF;
}
```

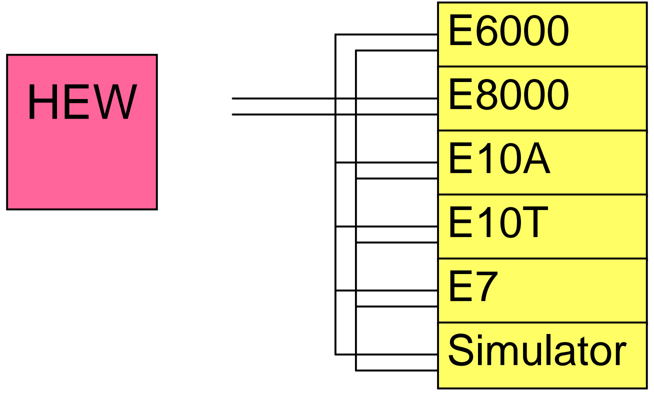
7.2.5 デバッガターゲットの同期

■説明

HEW では 1 つの HEW 上で複数ターゲットのデバッグが可能です。

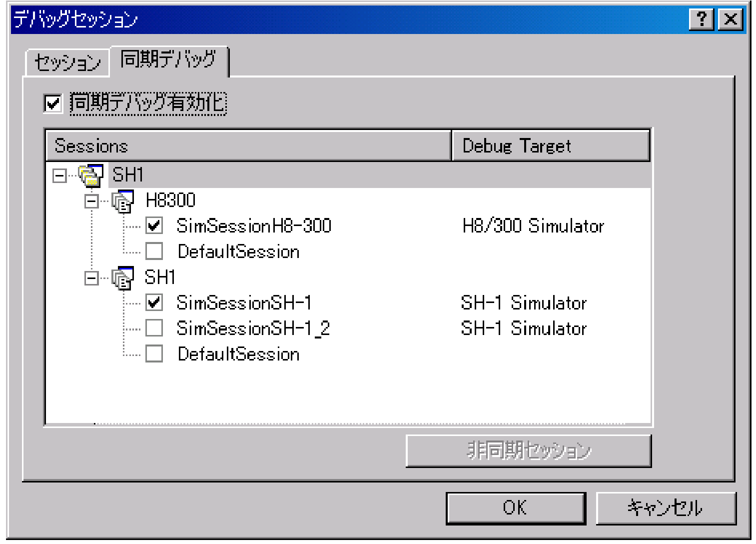
複数ターゲット間の同期を取り、デバッグすることが可能です。

デバッグ時に、1 つのセッションのイベント（ステップ､Go など）に同期して他のセッションで同じイベントを引き起こすことができます。



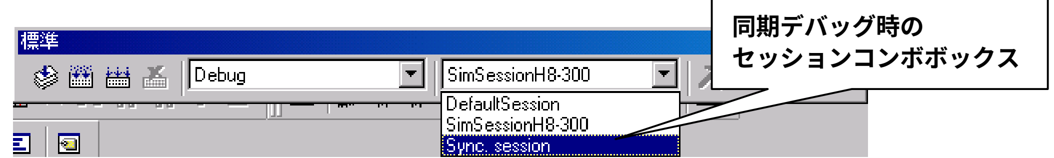
■デバッガターゲット同期の設定方法

1. [オプション->デバッグセッション]で以下のダイアログを表示し、同期デバッグタブをクリックします。
   ここで、同期させたいセッションをチェックし、その後に同期デバッグ有効化チェックボックスをチェックします。



RENESAS

2.　次に標準ツールバーのセッションコンボボックスから[Sync. session]を選びます。



3.　ツールバーにシンクロナイズセッションツールバーが登場します。これで設定終了です。

■操作可能コマンド

同期デバッグ有効時では以下の操作を、同期して使用することができます。

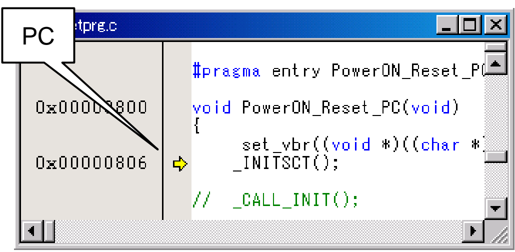
ユーザ操作	ターゲットデバッガセッション 1	ターゲットデバッガセッション 2
いずれかのセッションでの[実行]	"実行"	"実行"
いずれかのセッションでの[ステップ]	"ステップ"	"ステップ"
いずれかのセッションでの ESC 押下	"停止"	"停止"
-	ブレークポイントまたはユーザプログラム不正による"停止"	実行停止（ESC 押下による停止と同じ結果）
-	実行停止（ESC 押下による停止と同じ結果	ブレークポイントまたはユーザプログラム不正による"停止"
いずれかのセッションでの[CPU リセット]	"CPU リセット"	"CPU リセット"

■同期デバッグ例

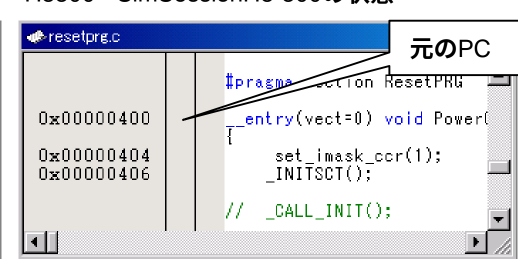
次にステップコマンドを実行した場合の例を示します。

1.　[SH1 - SimSessionSH-1]でステップ実行します。以下の状態になります。

SH – SimSessionSH-1の状態



H8300 - SimSessionH8-300の状態

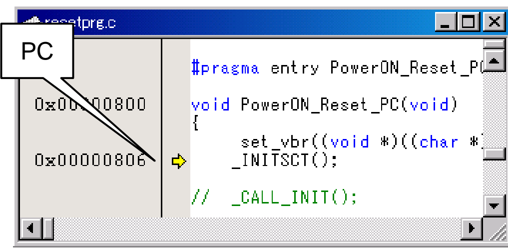


RENESAS

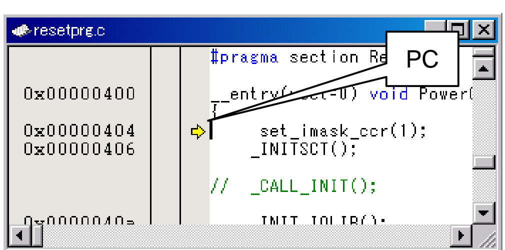
2. 次にシンクロナイズセッションツールバーでセッションを切り替えます。

3. 以下のように、[H8300 - SimSessionH8-300]セッションでもPCが次の行に移動していることを確認できます。

SH – SimSessionSH-1の状態



H8300 - SimSessionH8-300の状態



■備考

本機能は HEW3.0 以降でサポートしています。

RENESAS

7.2.6 タイマ使用方法

■説明

HEW ではタイマおよび割り込みの優先順位設定をサポートしています。
なお、各タイマのチャネル 0 のみをサポートしています。
また、オーバフロー／アンダフロー／コンペアマッチによる割り込みだけをサポートしており、インプットキャプチャなどの端子入出力を伴う機能はサポートしていません。

■各 CPU のサポートしているタイマ制御レジスタ

表中サポート状況の○はサポート、△は【説明】の機能に関するビットのみサポートしています。

デバッグ プラットフォーム名	タイマ名	サポートする制御レジスタ	サポート状況
SH-1	ITU0	TSTR	△
		TCR	△
		TIER	○
		TSR	○
		TCNT	○
		GRA	○
		GRB	○
SH-2/SH-2E/ SH2-DSP(SH7065)	CMT0	CMSTR	○
		CMCSR	○
		CMCNT	○
		CMCOR	△
SH-3/SH3-DSP/ SH3-DSP(Core)/SH-4/SH-4BSC/ SH-4(SH7750R)	TMU0	TCR	△
		TCNT	○
		TSTR	○
		TCOR	○
SH2-DSP(Core)	FRT0	TIER	△
		FTCSR	△
		FRC	○
		OCRA	○
		OCRB	○
		TCR	△
		TOCR	△

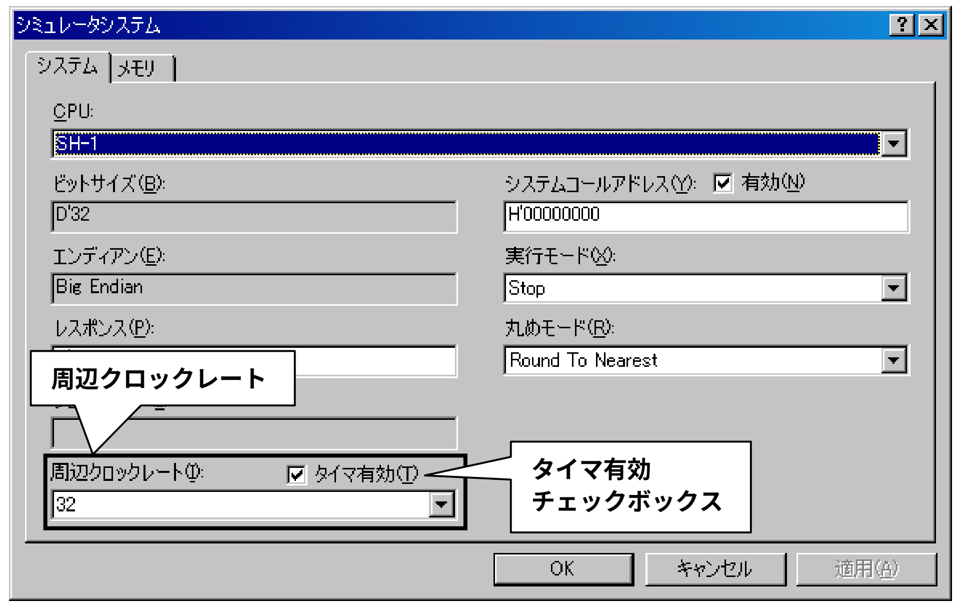
■各 CPU のサポートしている割り込み優先度レベル設定レジスタ

表中サポート状況の○はサポート、△は【説明】の機能に関するビットのみサポートしています。

デバッグ プラットフォーム名	サポートする制御レジスタ	サポート状況
SH-1	IPRC	△
SH-2	IPRG	△
SH-2E	IPRJ	△
SH2-DSP(SH7065)	IPRL	△
SH-3/SH3-DSP/ SH3-DSP(Core)/ SH-4/SH-4BSC/ SH-4(SH7750R)	IPRA	△
SH2-DSP(Core)	INTPRI0B	△

■タイマシミュレーション方法

[オプション->シミュレータ->システム]で以下のシミュレータシステムダイアログを表示し、タイマ有効チェックボックスをチェック、また外部クロックと周辺モジュールのクロック比を設定します。



さらには、以下のようにタイマ制御レジスタを使い、プログラムでタイマを有効にします。
また、周辺モジュールからタイマを動かすクロックを作るときの分周率は、タイマ制御レジスタで指定します。

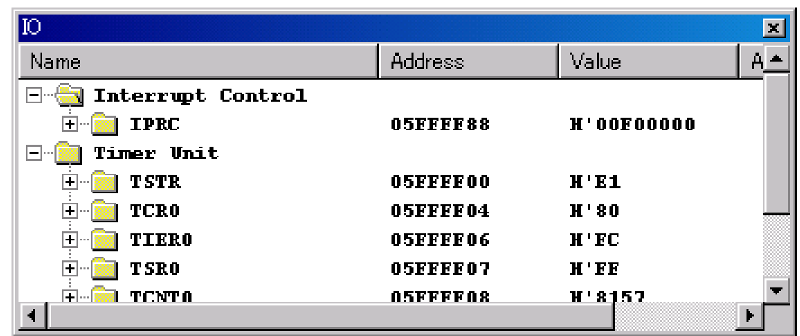
```
// ITU0 start
P_ITU.TSTR.BIT.STR0 = 1;
// ITU0 OverFlow interrupt enable
P_ITU0.TIER.BIT.OVIE = 1;
while(1);
```

ITU0タイマ有効

【注】タイマ制御レジスタを設定する前に、“シミュレータシステム”ダイアログ“メモリ”タブにおいて、該当レジスタへのアクセスが可能であることを確認してください。アクセス可能でない場合、制御レジスタへ値を設定することができず、タイマを使用することができません。

■タイマレジスタ確認方法

タイマレジスタ、割り込み優先度レベル設定レジスタの内容を見るには[表示->CPU->I/O]で以下の I/O ウインドウを表示し確認します。



■備考

本機能は HEW3.0 以降でサポートしています。

7.2.7　　タイマ使用の具体例

■説明

ここでは SH7034(SH-1)の ITU を例にとり、コンペアマッチ、周期ハンドラによる割り込みの使用方法を紹介します。

■HEW の設定

「7.2.6　**タイマ使用方法**」を参照し、タイマを有効にしてください。

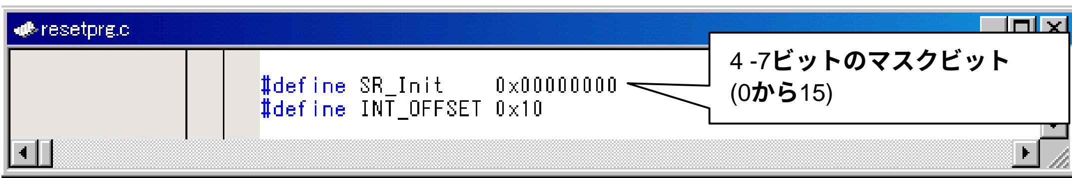
■コンペアマッチ割り込みのプログラム例

以下にコンペアマッチ割り込みを発生させる、プログラム例を示します。

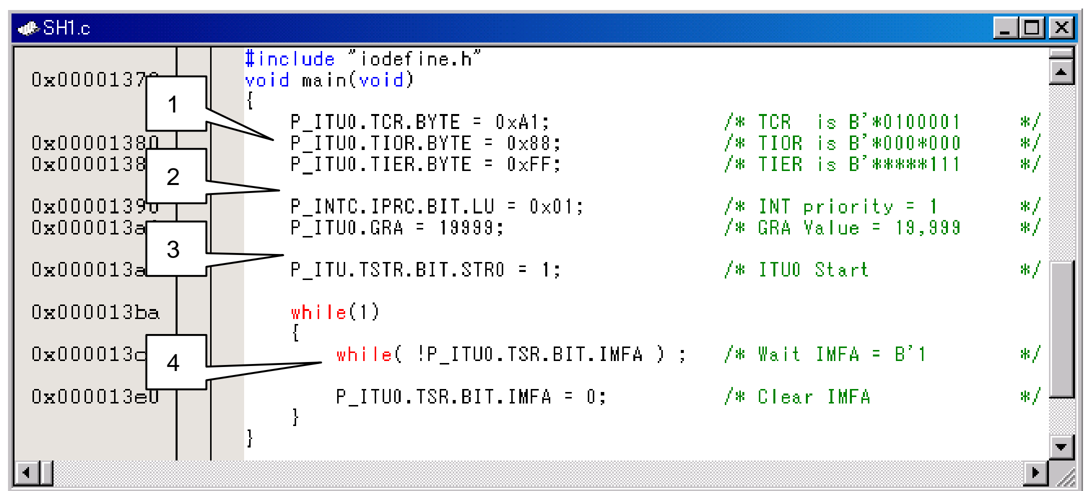
コンペアマッチ割り込みのためには、IPRC(インタラプトプライオリティレジスタ)の割り込み優先度が、SR レジスタ(ステータスレジスタ)の割り込みマスクビット以上になっている必要があります。

【SR レジスタの割り込みマスクビット設定】

以下のリセットルーチンが存在するファイルで SR レジスタの 4 -7 ビット目に 0 から 15 の値を設定します。



【割り込み発生プログラム説明】



1. TIER(タイマインタラプトイネーブルレジスタ)でIMFA(コンペアマッチフラグA)ビットが、1になったときの割り込みを許可します。
2. IMFAによる割り込み優先度を、設定します。
3. ITU0のタイマをスタートさせます。
4. IMFAビットが1になるのを、待ちます。(コンペアマッチ発生待ち)

■プログラム実行

「割り込み発生プログラム説明」の 4.の箇所で、TCNT0(タイマカウンタ 0)と GRA(ジェネラルレジスタ A)が一致する(コンペアマッチする)、状態を待ちます。

一致すると、コンペアマッチ割り込みが発生し、以下に示す割り込みルーチンが呼び出されます。

詳細は該当のハードウェアマニュアルを参照してください。

```
intprg.c
0x000004d0    // 78 DMAC3 DEI3
              void INT_DMAC3_DEI3(void){/* sleep(); */}
              // 79 DMAC3 Reserved
0x000004d4    void INT_DMAC3_Reserved(void){/* sleep(); */}
              // 80 ITU0 IMIA0
0x000004d8    void INT_ITU0_IMIA0(void)
              {
                  return;
              }
              // 81 ITU0 IMIB0
```

■周期ハンドラのプログラム例

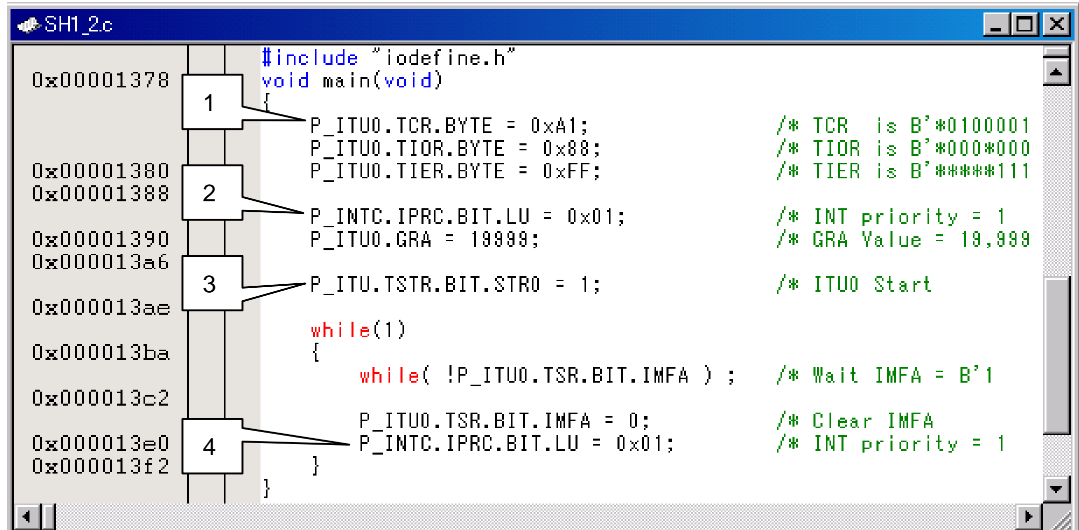
以下に周期ハンドラのプログラム例を示します。

コンペアマッチが起きたときに、タイマをクリアした後、割り込みハンドラへ分岐し処理を行います。

処理を行った後に、IPRC(インタラプトプライオリティレジスタ)の割り込み優先度を下げます。

その後、割り込み発生元に戻り、再び IMFA が入るように割り込み優先度を上げています。

SR レジスタの割り込みマスクビット設定は、コンペアマッチの例を参照してください。

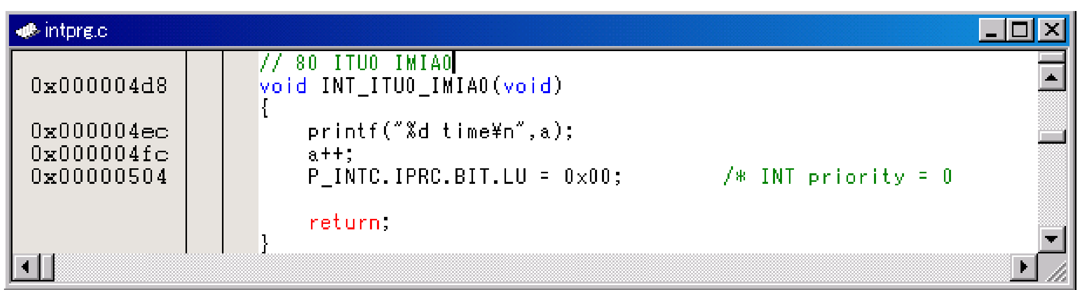


1. TCR(タイマコントロールレジスタ)でIMFA(コンペアマッチフラグA)ビットが、1になったときのタイマカウンタ(TCNT)をクリアするように設定します。
2. IMFAによる割り込み優先度を、設定します。
3. ITU0のタイマをスタートさせます。
4. コンペアマッチが発生した後、割り込み優先度を上げます。

RENESAS

■プログラム実行

コンペアマッチが発生するのを待ち、発生すると以下の割り込みルーチンに分岐します。
割り込みルーチンでは、処理を行い IMFA 割り込みの優先度を下げてリターンします。
そうすることによって、この割り込みを終了することができます。
そして、次の周期のコンペアマッチ割り込みを受け付けることができます。
詳細は該当のハードウェアマニュアルを参照してください。
なお、HEW の仕様では、割り込み発生時に割り込み関数の先頭で PC が停止します。
周期ハンドラのシミュレーションをする場合は、Go コマンドなどを使用し、その都度 PC を進める必要があります。



RENESAS

7.2.8　　デバッガターゲットの再登録

■説明

HEW はワークスペースの新規作成時にプロジェクトタイプで Application などを選択すると、デバッガの登録をすることができます。

しかし、プロジェクトの新規作成時には必要と思わず、登録をしないことがあります。

このようなときに、本機能を用いるとプロジェクト作成後に再度デバッガの登録をすることができます。

ただし、本機能はワークスペースの新規作成時にプロジェクトタイプで Application を選択した場合に限り使用することができます。

■使用方法

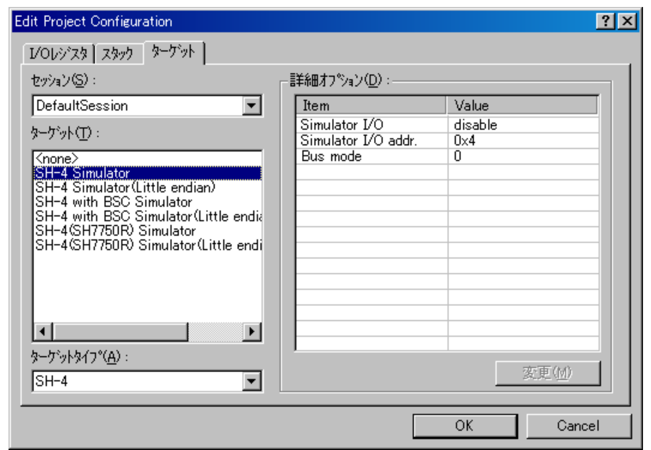
HEW メニュー：プロジェクト>構成の編集

■再設定できる機能

【設定方法】

Edit Project Configuration ダイアログのターゲットタブで、シミュレータなどのデバッガターゲットを設定できます。

該当のセッションにすでにデバッガが接続されている場合は、「This target has already existed. It does not support duplicated targets」と表示し、接続することはできません。



■備考

再登録は HEW2.1 以降でサポートしています。

7.3 Call Walker 編

Call Walker は、最適化リンケージエディタが出力したスタック情報ファイル(*.sni)またはシミュレータデバッガが出力したプロファイル情報ファイル(*.pro)を読み込んで、静的なスタック使用量を表示します。

スタック情報ファイルに出力できないアセンブリプログラムのスタック使用量は、編集機能を用いて情報を追加することが可能であり、システム全体のスタック使用量を求めることもできます。

編集したスタック使用量に関する情報は、呼び出し情報ファイル(*.cal)として保存・読み込みが可能です。

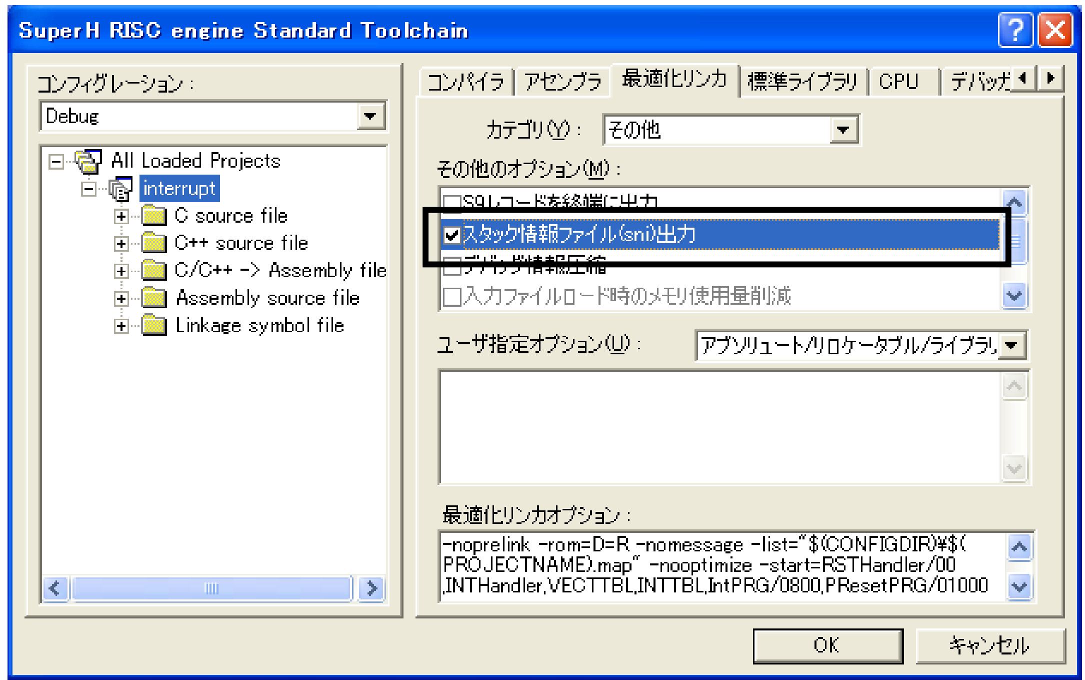
また、呼び出し情報ファイルをマージ(連結)することも可能です。

7.3.1 スタック情報ファイルの作成方法

以下の手順に従って、スタック情報ファイル、プロファイル情報ファイルのいずれかを作成します。

■スタック情報ファイル(*.sni)の作成方法

スタック情報ファイルは最適化リンカの以下のオプションにより作成することができます。



ダイアログメニュー：最適化リンカタブカテゴリ:[その他]のスタック情報ファイル(sni)出力
コマンドライン　　：*STACk*

RENESAS

■プロファイル情報ファイル(*.pro)の作成方法

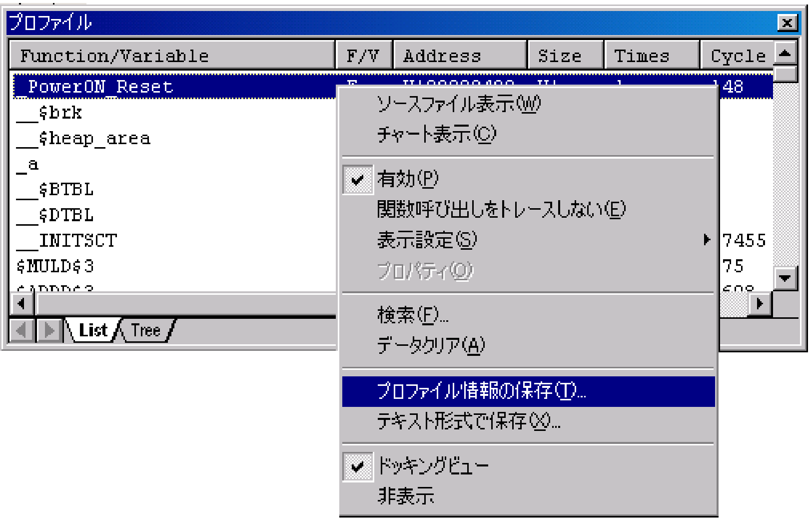
以下のプロファイル機能により、ユーザプログラムを実行させます。

実行終了後プロファイルウインドウで右クリックをし、プロファイル情報の保存によりプロファイル情報ファイル(*.pro)が作成されます。

プロファイル情報作成方法の詳細は、High-performance Embedded Workshop 3 **ユーザーズマニュアル シミュレータ・デバッガ編「4.15 プロファイル情報を見る」**を参照してください。

【プロファイルウインドウ】

表示->パフォーマンス->プロファイル



7.3.2　　Call Walker の起動

以下のいずれかの方法で起動します。

■スタートメニューから起動

プログラム->Renesas High-performance Embedded Workshop->Call Walker をクリック

■HEW から起動

ツール->Call Walker をクリック

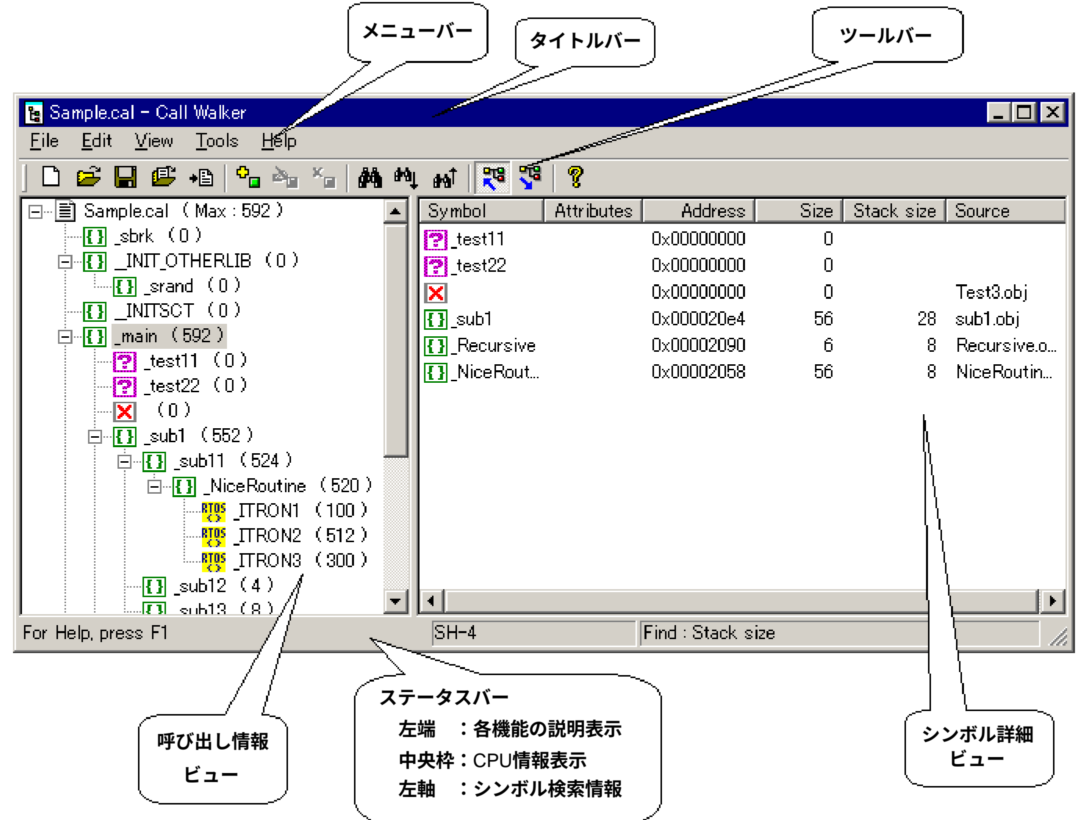
7.3.3　　ファイルのオープンと Call Walker の画面

Call Walker を起動後、スタック情報ファイル(*.sni)またはプロファイル情報ファイル(*.pro)を[File メニュー->Import Stack File...]でオープンします。

[File メニュー->Open...]は既存の編集ファイル(*.cal)をオープンする際に、使用します。

オープンすると以下の画面が出力されます。

【注】　標準ライブラリ以外のアセンブラ関数はスタック使用量が 0 と表示されます。
「7.3.4 スタック情報の編集」をご参照の上、スタック使用量を設定してください。



RENESAS

■呼び出し情報ビュー

シンボル間のリンク階層構造を表示します。
各シンボル名の右側に使用しているスタック使用量を表示しています。

（1）シンボルの区分表示
シンボル名の左側に、シンボルの種類をアイコンで表示しています。
次の種類があります。

編集中のファイル
アセンブラのシンボル
C/C++の関数

再帰呼び出し(リカーシブコール)関数または循環関数

(a)再帰呼び出し(リカーシブコール)
関数内で自関数を呼び出す場合に表示されます。

【例】

```
void func(int x)
{
    x++;
    if(x != OFF)
        func(x);

    if(x == MAX)
        return;
}
```

```
_func （0x00000006 ）
  _func(Recursive)
```

(b)循環関数
間接的に自関数を呼び出す場合に表示されます。

【例】

```
void func1(int a)
{
    func2(10);
}
void func2(int b)
{
    func1(9);
}
```

```
_func1 （0x00000008 ）
_func2 （0x00000004 ）
  _func1(Recursive)
```

: RTOS関数(ITRONなどのシンボル)

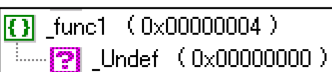
: 参照元シンボル不明関数

下記例の場合、関数func1()で関数Undef()を呼び出していますが、関数Undef()の実体がない場合、関数Undef()に本アイコンが表示されます。

実際には実体のない関数呼び出しは、リンク時にエラーになりますがリンクオプションchange_messageを使用することにより、エラーをウォーニングに変更できます。
ウォーニングにすると、ロードモジュールが作成できるので、スタック情報ファイルも作成されます。

【例】

```
void func1(void)
{
    Undef();
}
```

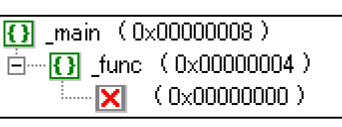


: アドレス参照未解決関数

下記のように、関数をテーブル呼び出ししている場合に表示されます。

【例】

```
static int (*key[3])()=
                    {nop, stop, play};
void func(int x)
{
    (*key[a])();
}
```



: 省略表示シンボル

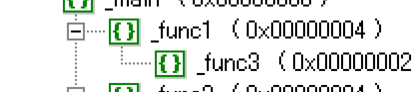
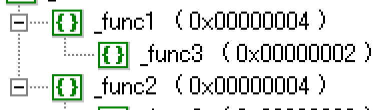
本ツールでは、リンク階層を全て表示するため、ユーザアプリケーションが大きい場合表示量が膨大になります。そこで、各シンボルの階層表示について最初の1つだけを表示し省略表示シンボルで他の同様な部分は表示を省略します。

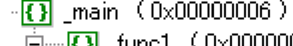
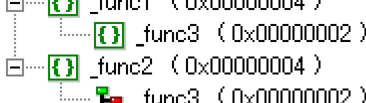
この表示は、[View->Show All Symbols / Show Simple Symbols]で切り替えることができます。

【例】

Show All Symbols　　　Show Simple Symbols

a

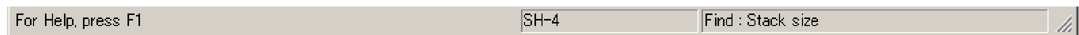
RENESAS

■シンボル詳細ビュー

Symbol	Attri...	Address	Size	Stack size	Source
_INT_TXI1_...	I	0x000004...	0x00000002	0x00000004	intprg.obj
_abort		0x000008...	0x00000002	0x00000004	CallWalker2....
_sbrk		0x000008...	0x0000002c	0x00000008	sbrk.obj
_sub		0x000008...	0x00000002	0x00000004	CallWalker2....
_nop		0x000008...	0x00000002	0x00000004	CallWalker2....
PowerON...		0x000004...	0x00000016	0x00000004	resetprg.obj
_play		0x000008...	0x00000002	0x00000004	CallWalker2....
_stop		0x000008...	0x00000002	0x00000004	CallWalker2....
_INT_TGI1...	I	0x000004...	0x00000002	0x00000004	intprg.obj
_INT_TGI0...	I	0x000004...	0x00000002	0x00000004	intprg.obj
_INT_TGI0	I	0x000004	0x00000002	0x00000004	intprg.obj

各シンボルごとのアドレス、属性、スタック使用量を表示します。
シンボルをクリック後、右クリックを押下すると各編集コマンドを実行できます。

■ステータスバー



現在オープンしているスタック情報ファイルを作成した際の、CPU 種別などを表示します。

■最大スタック使用量

CallWalker3.cal （ Max : 0x00000006 ）
main （0x00000006 ）

現在オープンしているスタック情報ファイルの、静的な最大スタック使用量を表示します。

■標準ライブラリバージョン選択

Standard Library Version : Standard_library_SH_V8

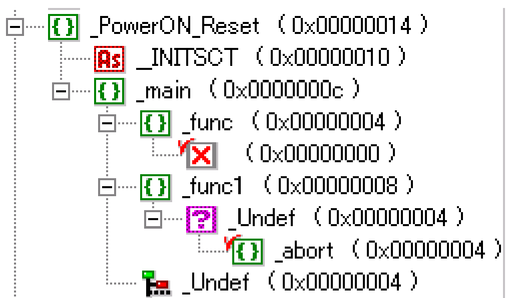
現在オープンしているスタック情報ファイルを作成した際の、標準ライブラリバージョンを選択します。
これにより、標準ライブラリ内アセンブラ関数のスタック使用量を表示しています。
HEW パッケージを 1 つしかインストールしていない場合は、選択する必要はありません。

7.3.4 スタック情報の編集

ファイルをオープン中、右側のシンボル詳細ビューでシンボル名を選択し、Edit メニューの Add...、Modify...、Delete... コマンドでシンボルの追加、変更、削除ができます。
シンボル詳細ビューで右クリックを押下しても、同様の操作が行えます。

本ツールは静的なスタック最大使用量を算出できますが、多重割り込みなどによる、動的な最大使用量を調査するためには、ユーザ側で情報ファイルを編集する必要があります。

また、左側の呼び出し情報ビューでシンボルをドラッグ＆ドロップすると、シンボルの位置を変更できます。
シンボルの移動や編集をすると、以下のように左側の呼び出し情報ビューの、該当シンボルに変更を示すチェックが付きます。

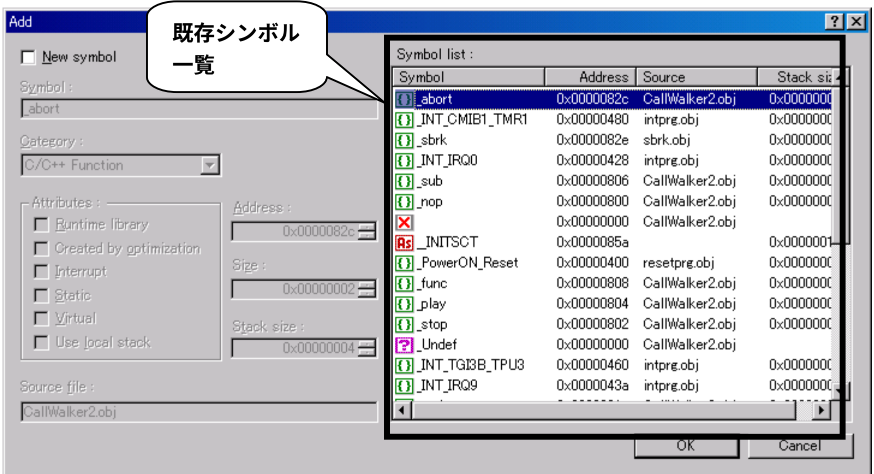


以降に、各コマンドの説明をします。

■Add...**コマンド**

（1） 既存シンボルの追加

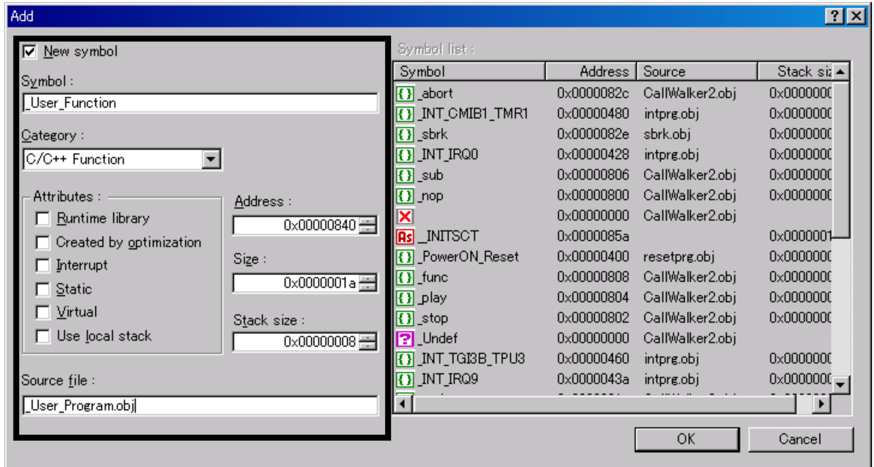
Add...コマンドをクリックすると、以下のダイアログが表示されます。右側の一覧は現在のファイル内のシンボルです。既存のシンボルを追加する場合は、このリストからシンボルを選択しOKボタンを押下します。



（2） 新規シンボルの追加

左側のチェックボックスをチェックすると、シンボルを新規作成できます。

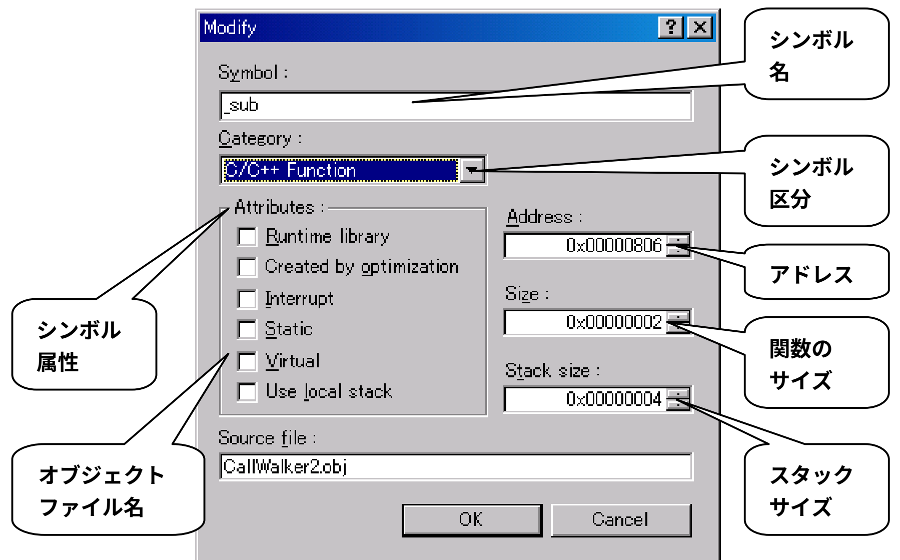
その際に、シンボル名、シンボル区分、属性、アドレス、スタック使用量などを決定できます。



RENESAS

■Modify...**コマンド**

情報を変更したいシンボルを選択し、Modify...コマンドをクリックすると、以下のダイアログが表示されます。ここで各種情報を変更することができます。



■Delete...**コマンド**

スタック使用量調査に不要なシンボルを、選択し(左側、右側どちらでも良い) Delete...コマンドをクリックするとシンボルを削除できます。

7.3.5 アセンブラプログラムのスタック使用量

C/C++プログラムと違い、アセンブラプログラムは、アセンブルをしてもスタック使用量を自動で算出することができません。
そのため、Call Walker 上でアセンブラプログラム関数のスタック使用量を編集する必要がありました。
しかし、.STACK 制御命令を使用すると、アセンブラプログラム関数内にスタック使用量を記述することができます。Call Walker は.STACK 制御の数値を画面に表示します。

■.STACK 制御命令説明

シンボルに対して、Call Walker で参照するスタック使用量を定義します。1 つのシンボルに対して定義できるスタック値は 1 度のみ有効です。2 度以上指定した場合は、その定義を無効とします。また、指定できるスタック値は、H'00000000～H'FFFFFFFE の範囲の 2 の倍数のみとし、それ以外を指定した場合はその定義を無効とします。

・定数値を指定する。
　かつ
・前方参照シンボル外部参照シンボル、相対アドレスを使わずに指定する。

■.STACK アセンブラ制御命令記述方法

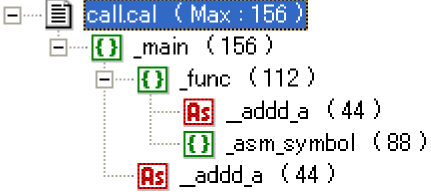
△.STACK△<シンボル>=<スタック値>

■アセンブラプログラム例

```
            .CPU        SH1
            .EXPORT     _asm_symbol
            .SECTION    P,CODE,ALIGN=4
_asm_symbol:
            .STACK      _asm_symbol=88      _asm_symbol関数のスタックサイズ
                  :
            RTS
            .END
```

■Call Walker の表示例

下記のように、Call Walker 上で_asm_symbol 関数のスタック使用量が、「88」と表示されるようになります。



■備考

（1） .STACK制御命令はCall Walkerにスタックサイズを表示させる機能です。プログラムの動作に影響を与えるものではありません。
（2） 本制御命令はSuperH RISC engine アセンブラVer.7からサポートしています。

7.3.6 スタック情報のマージ（連結）

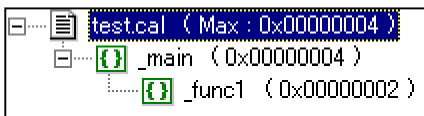
保存後、または編集中のスタック情報ファイルと、他のスタック情報ファイルをマージ(連結)することができます。これにより、編集したスタック情報が、再ビルド後のスタック情報に上書きされません。

■マージの例

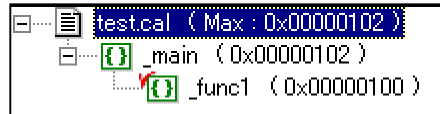
(1) test.cの内容

```
void main(void)
{
    func1();
}
```

(2) Call Walkerでスタック情報ファイルをオープン



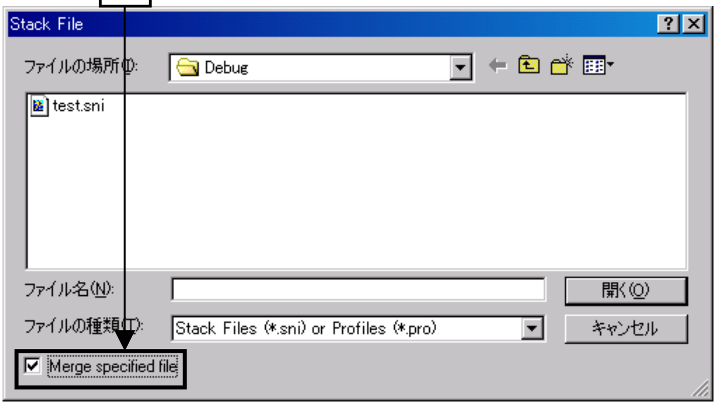
(3) 内容を変更(func1のスタック使用量を100に変更)



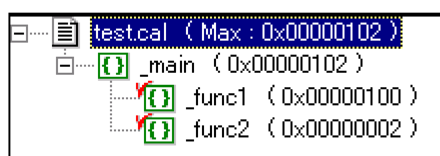
(4) test.cの内容を変更後にビルド(**func2の呼び出しを追加**)

```
void main(void)
{
    func1();
    func2();
}
```

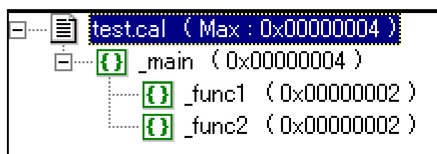
(5) Call Walkerでtest.calを開いたままtest.sniをオープンします。
このとき、ここをチェックし、開くボタンを押下します。



（6）　すると、(3)で変更したfunc1のスタック使用量を残しつつ、func2の情報が追加になります。
　　　これがスタック情報のマージ(連結)です。



もし、(5)でチェックをしない場合は、以下のように(3)で変更したfunc1のスタック使用量は変更前の値に戻ってしまいます。

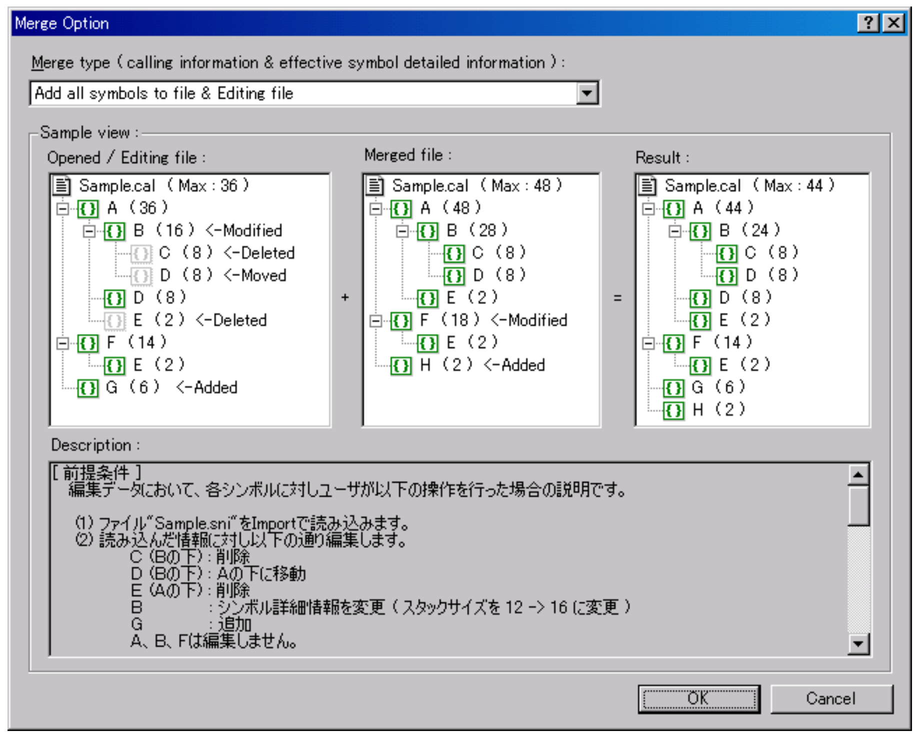


■マージの詳細オプション

マージをする際の方法を変更することもできます。5 通りの方法があります。
詳細なマージ方法は、本ダイアログの Description を参照してください。

【指定方法】

Toolsメニュー-> Merge Option...



■備考

マージ機能は Call Walker のバージョン 1.3 以上で使用できます。

7.3.7 その他の機能

■リアルタイム OS シンボル

以下の指定を行うと、画面左側の呼び出し情報ビューでリアルタイム OS のシンボルを、 と表示することができます。

【指定方法】

Toolsメニュー-> Realtime OS Option...

この拡張子 csv のファイルは各リアルタイム OS 製品に同梱されています。

■リスト出力

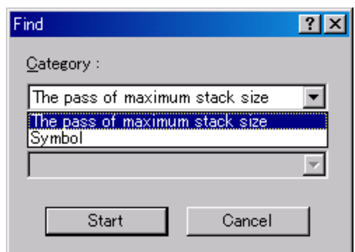
スタック情報をテキスト形式で、ファイルに出力することができます。

【出力方法】

Fileメニュー->Output List...

■検索機能

以下のダイアログで、2 通りの呼び出し情報ビューからの検索ができます。



(1) スタック使用量が最大のパスを検索
(2) シンボル名の検索

【指定方法】

Editメニュー->Find...
Editメニュー->Find Next...(次を検索)
Editメニュー->Find Previous...(前を検索)

■呼び出し情報ビューの表示形式設定

以下のコマンドで、スタック使用量の表示について 2 通りの方法を選択できます。

(1) Show Required Stack
下から上にスタック使用量が積み重なっていきます。

(2) Show Used Stack
上から下にスタック使用量が積み重なっていきます。

【指定方法】

Viewメニュー->Show Required Stack または Show Used Stack

8. 効率の良い C++プログラミング技法

本コンパイラは C++言語および C 言語をサポートしています。
本章では、オブジェクト指向言語 C++のオプションおよび各種 C++機能の使用方法などについて詳しく説明します。

また、C++を組み込みシステムで適用する場合、注意を払いながらプログラミングをしないと、予想以上に大きなオブジェクトサイズになる、またはスピードの低下を招いてしまいます。
そのため本章では、C 言語と比較し性能劣化を招く事例を交え紹介し、性能劣化に影響しない記述を紹介します。

効率的 C++プログラミング技法の一覧を示します。

No.	大項目	中項目	参照
1	初期処理／後処理	グローバルクラスオブジェクトの初期処理と後処理	8.1.1
2	C++機能紹介	C 言語オブジェクトの参照方法	8.2.1
3		new,delete の実装方法	8.2.2
4		スタティックメンバ変数	8.2.3
5	オプション活用法	組み込み向け C++言語	8.3.1
6		実行時型情報	8.3.2
7		例外処理機能	8.3.3
8		プレリンカの起動抑止	8.3.4
9	C++記述のメリット・デメリット	コンストラクタ (1)	8.4.1
10		コンストラクタ (2)	8.4.2
11		デフォルトパラメータ	8.4.3
12		インライン展開	8.4.4
13		クラスメンバ関数	8.4.5
14		operator 演算子	8.4.6
15		関数のオーバロード	8.4.7
16		リファレンス型	8.4.8
17		スタティック関数	8.4.9
18		スタティックメンバ変数	8.4.10
19		匿名 union	8.4.11
20		仮想関数	8.4.12

8.1 初期処理／後処理

8.1.1 グローバルクラスオブジェクトの初期処理と後処理

■ポイント

C++言語でグローバルクラスオブジェクトを使用する場合、初期処理関数(_CALL_INIT)と後処理関数(_CALL_END)を main 関数の前後で呼び出す必要があります。

■グローバルクラスオブジェクトとは？

下記のようにクラスオブジェクトの宣言を、関数の外で宣言しているものです。

（関数内クラスオブジェクト宣言）

```
void main(void)
{
    X XSample(10);
    X* P = &XSample;

    P->Sample2();
}
```

（グローバルクラスオブジェクト宣言）

```
X XSample(10);
void main(void)
{
    X* P = &XSample;

    P->Sample2();
}
```

関数の外で宣言

■なぜ初期処理／後処理が必要か？

上記のように、関数内でクラスオブジェクト宣言している場合、関数 main を実行しているときに、クラス X のコンストラクタを呼び出します。

それに対し、グローバルなクラスオブジェクト宣言は、関数を実行しても宣言が実行されることはありません。

そのため、main 関数の呼び出し前に_CALL_INIT を呼び出して、明示的にクラス X のコンストラクタを呼び出す必要があります。また同様に_CALL_END を main 関数後に呼び出しクラス X のデストラクタを呼び出すようにします。

■_CALL_INIT/_CALL_END の使用時と未使用時の動作

クラス X のメンバ変数 x の値を、参照した場合の値を下記に示します。

未使用の場合は以下のように、正しい値が得られず while 文内の式を実行しません。

（メンバ変数xの値）
_CALL_INIT使用時　→10

_CALL_INIT未使用時→0

```
class X{
    int x;
public:
     X(int n){x = n}; // constructor
     ~X(){}           // destructor
    void Sample2(void);
};
X XSample(10); // global class object
void X::Sample2(void)
{
    while(x == 10)
    {
    }
}
void main(void)
{
    X* P = &XSample;

    P->Sample2();
}
```

←参照場所

■_CALL_INIT/_CALL_END の呼び出し方法

main 関数の呼び出し前後で以下のように記述します。

```
void INIT(void)
{
    _INITSCT();
    _CALL_INIT();
    main();
    _CALL_END();
}
```

また、HEW をご使用の場合は、resetprg.c の_CALL_NIT/_CALL_END 呼び出し箇所のコメントを削除します。

（resetprg.c の PowerON_Reset 関数）

```
__entry(vect=0) void PowerON_Reset(void)
{
    set_imask_ccr(1);
    _INITSCT();

   _CALL_INIT();      // Remove the comment when you use global class object

// _INIT_IOLIB();     // Remove the comment when you use SIM I/O

// errno=0;           // Remove the comment when you use errno
// srand(1);          // Remove the comment when you use rand()
// _s1ptr=NULL;       // Remove the comment when you use strtok()

    HardwareSetup();  // Use Hardware Setup
    set_imask_ccr(0);

    main();

// _CLOSEALL();       // Remove the comment when you use SIM I/O

   _CALL_END();       // Remove the comment when you use global class object

    sleep();
}
```

8.2 C++機能紹介

8.2.1 C 言語オブジェクトの参照方法

■ポイント

「extern "C"」宣言を用いることにより、既存の C オブジェクトプログラムの財産を、直接 C++プログラムから利用することができます。

また、C++オブジェクトの財産を、C プログラムから利用することができます。

■使用例

1.「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++;
    }
}
```

2.「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++;
    }
}
```

■注意事項

1. エンコード方式、実行方式を変更したため、旧バージョン(Ver.5)のコンパイラが生成したC++のオブジェクトはリンクできません。
   必ずリコンパイルしてから使用してください。

2. 上記方法で呼び出した関数は、オーバロードすることはできません。

8.2.2　　new,delete の実装方法

■ポイント

new を使用する場合は、低水準関数を実装する必要があります。

■説明

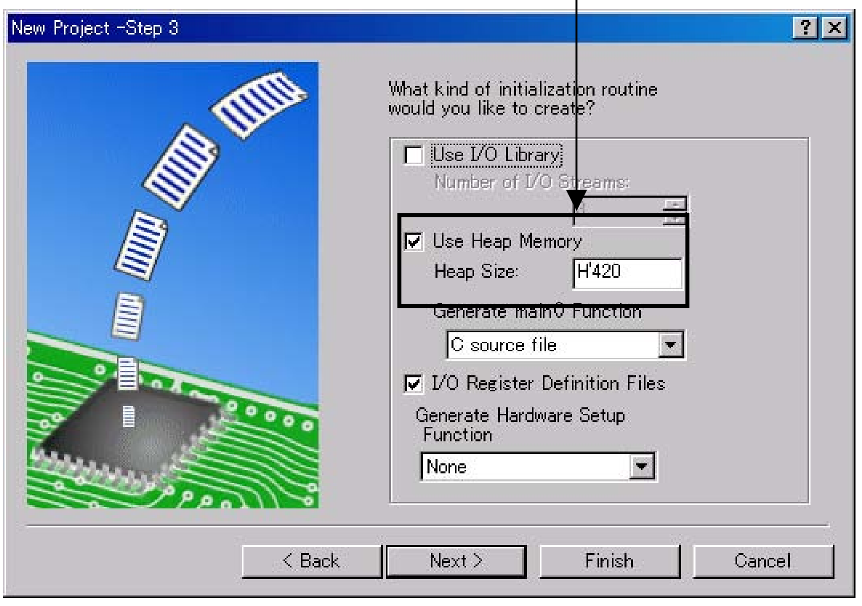
組み込みシステムにおいて new を使用する場合、実際のヒープメモリの動的な確保は malloc を使用することによって実現しています。

よって、malloc 使用時と同様に低水準インタフェースルーチン(sbrk)を実装し、割り付けるヒープメモリの容量を指定する必要があります。

■実装方法

HEW を使用する場合、ワークスペース作成時に「Use Heap Memory」がチェックされていることを確認してください。
チェックすることにより、次紙に示す sbrk.c と sbrk.h が自動的に生成されます。
確保するヒープメモリの容量は Heap Size で指定してください。
ワークスペース作成後に容量を変更する場合は sbrk.h で HEAPSIZE に定義する値を変更してください。

また、HEW を使用しない場合、次紙に示すファイルを作成しプロジェクトに実装してください。



RENESAS

```
（sbrk.c）

#include <stdio.h>
#include "sbrk.h"

//const size_t _sbrk_size=   /* Specifies the minimum unit of    */
                             /* the defined heap area            */

static  union  {
   long  dummy ;         /* Dummy for 4-byte boundary         */
   char heap[HEAPSIZE]; /* Declaration of the area managed   */
                        /*                       by sbrk */
}heap_area ;

static  char  *brk=(char *)&heap_area;/* End address of area assigned    */

/****************************************************************************/
/*      sbrk:Data write                                                     */
/*      Return value:Start address of the assigned area (Pass)              */
/*                   -1                (Failure)                            */
/****************************************************************************/
char  *sbrk(size_t size)     /* Assigned area size  */
{
   char  *p;

   if(brk+size>heap_area.heap+HEAPSIZE) /* Empty area size       */
     return (char *)-1 ;

   p=brk ;                   /* Area assignment        */
   brk += size ;             /* End address update  */
   return p ;
}
```

```
(sbrk.h)

/* size of area managed by sbrk */
#define HEAPSIZE 0x420
```

8.2.3 スタティックメンバ変数

■説明

C++では、クラスのメンバ変数を static 属性にすると、そのメンバ変数はクラス型の複数のオブジェクト間で共有することができます。

これにより、同じクラス型の複数オブジェクト間で、共通なフラグなどに利用することができるので便利です。

■使用例

main 関数内で、クラス A 型のオブジェクトを 5 つ作成します。

static なメンバ変数 num の初期値は 0 です。この値がオブジェクトの作成毎に、コンストラクタでインクリメントされます。

static なメンバ変数 num は各オブジェクト間で共有されるので、変数 num の値は 5 まで上昇します。

■FAQ

スタティックメンバ変数使用時のよくある質問を以下に示します。

【L2310 エラー発生】

static メンバ変数を使用したとき、リンク時に「** L2310 (E) Undefined external symbol "クラス名::static メンバ変数名" referenced in "ファイル名"」が出力される。

【解決策】

static メンバ変数の実体が定義されていないため、エラーが発生しています。

次紙に示すように、以下の定義を追加してください。

初期値がある場合：`int A::num = 0;`
初期値がない場合：`int A::a;`

【初期値代入不可能】

初期値を持っている static メンバ変数に初期値が代入されていない。

【解決策】

初期値を持つ static メンバ変数は、初期値付き変数として扱われるためデフォルトで D セクションに生成されます。よって、最適化リンケージエディタの ROM 化支援オプションの指定、およびイニシャルルーチンで_INITSCT 関数による ROM から RAM への D セクションコピー*が必要です。

【注】* HEW でイニシャルルーチンを自動生成している場合は、本対策は不要です。

（C＋＋プログラム）

```
class A
{
private:
    static int num;
public:
    A(void);
    ~A(void);
};

int A::num = 0;

void main(void)
{
    A a1;
    A a2;
    A a3;
    A a4;
    A a5;
}

A::A(void)
{
    ++num;
}

A::~A(void)
{
    --num;
}
```

スタティックメンバ変数の実体定義

クラスA型クラスオブジェクト作成

static**メンバ変数をインクリメント**

8.3 オプション活用法

8.3.1 組み込み向け C++言語

■説明

組み込みシステムでは、ROM/RAM サイズおよび実行速度が重要です。
組み込み向け C++言語(EC++)は C++言語のサブセットで、組み込みシステムに向かない機能を排除した言語仕様になっています。
よって、組み込みシステムに適したオブジェクトを生成できます。

■指定方法

ダイアログメニュー：C/C++タブ Category:[Other]の Check against EC++ language specification
コマンドライン　　：*eccp*

■未サポートキーワード

以下のキーワードを記述するとエラーメッセージを出力します。

catch、const_cast、dynamic_cast、explicit、mutable、namespace、reinterpret_cast、static_cast、template、throw、try、typeid、typename、using

■未サポート言語仕様

以下の言語仕様を記述するとウォーニングメッセージを出力します。

多重継承、仮想基底クラス

8.3.2 実行時型情報

■説明

C++では仮想関数を持つクラスオブジェクトの場合、実行時でなければ判明しない型が存在します。
このような状況を支援する機能として、実行時識別の機能をサポートしています。
C++でこの機能を使うには、type_info クラス,typeid 演算子,dynamic_cast 演算子を使います。
本コンパイラでは下記のオプションを指定することにより、実行時型情報が使えるようになります。
また、リンク時に以下のオプションで、プレリンカを起動する必要があります。

■指定方法

ダイアログメニュー： CPUタブEnable/disable runtime type information
コマンドライン　　： *rtti=on | off*

ダイアログメニュー： Link/LibraryタブCategory:[Input]のPrelinker controlをAutoまたはRun prelinker
コマンドライン　　： *noprelinkを指定しない（デフォルト）*

■type_info クラス,typeid 演算子の使用例

type_info クラスは、オブジェクトの実行時の型に関する識別操作のためのクラスです。
type_info クラスを使うと、プログラム実行時の型比較判定、クラスの型名取得などができるようになります。
type_info クラスを使うには、typeid 演算子で仮想関数を持つクラスオブジェクトを指定します。

```
#include <typeinfo.h>
#include <string>
class Base{
protected:
    string *pname1;
public:
    Base() {
        pname1 = new string;
        if (pname1)
            *pname1 = "Base";
    }
    virtual string Show() {return *pname1;}
    virtual ~Base() {
        if (pname1)
            delete pname1;
    }
};
class Derived : public Base{
    string *pname2;
public:
    Derived() {
        pname2 = new string;
        if (pname2)
            *pname2 = "Derived";
    }
    string Show() {return *pname2;}
    ~Derived() {
        if (pname2)
            delete pname2;
    }
};
void main(void)
{
    Base* pb = new Base;
    Derived* pd = new Derived;

    const type_info& t = typeid(pb);
    const type_info& t1 = typeid(pd);
    t.name();
    t1.name();
}
```

必ずインクルードする

基本クラス

仮想関数

仮想デストラクタ

派生クラス

仮想関数

仮想デストラクタ

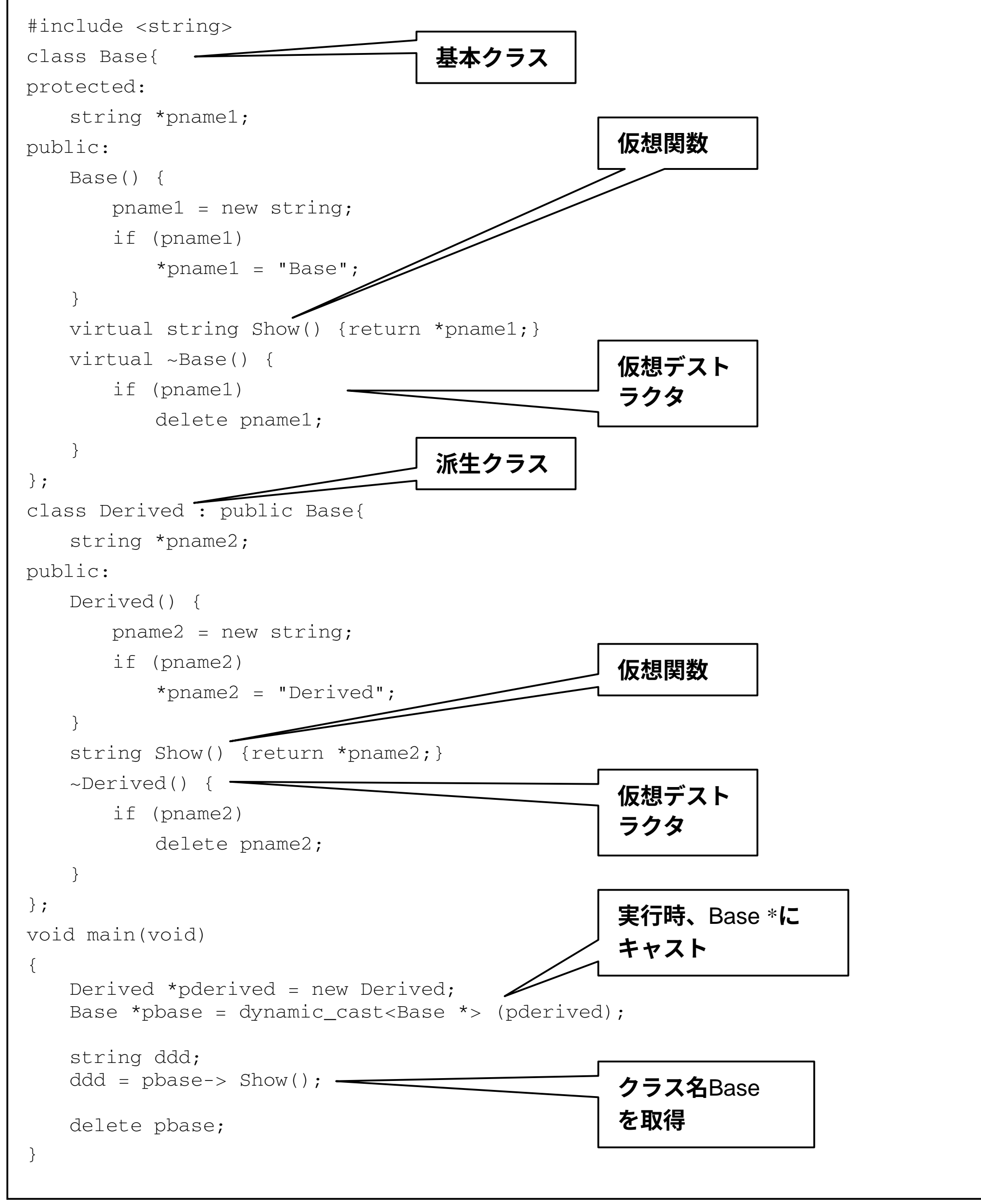
クラスオブジェクトを指定

型名[Base*]取得

型名[Derived *]取得

■dynamic_cast 演算子の使用例

dynamic_cast 演算子を使うと、たとえば仮想関数を含むクラスとその派生クラス間では、実行時に dynamic_cast 演算子を使って、派生クラス型のポインタや参照を、基本クラス型のポインタまたは参照にキャストすることができます。



```
#include <string>
class Base{
protected:
    string *pname1;
public:
    Base() {
        pname1 = new string;
        if (pname1)
            *pname1 = "Base";
    }
    virtual string Show() {return *pname1;}
    virtual ~Base() {
        if (pname1)
            delete pname1;
    }
};
class Derived : public Base{
    string *pname2;
public:
    Derived() {
        pname2 = new string;
        if (pname2)
            *pname2 = "Derived";
    }
    string Show() {return *pname2;}
    ~Derived() {
        if (pname2)
            delete pname2;
    }
};
void main(void)
{
    Derived *pderived = new Derived;
    Base *pbase = dynamic_cast<Base *> (pderived);

    string ddd;
    ddd = pbase-> Show();

    delete pbase;
}
```


RENESAS

8.3.3 例外処理機能

■説明

C++には、C にはない例外というエラー処理のメカニズムがあります。
例外とは、プログラム内部のエラー箇所とエラー対処コードを結び付けるための仕組みです。
例外のメカニズムを使ってエラー対処コードを一箇所にまとめることができます。
本コンパイラでは以下のオプションを指定することにより使用できます。

■指定方法

ダイアログメニュー：CPUタブUse try,throw and catch of C++
コマンドライン　　：*exception*

■使用例

ファイル"INPUT.DAT"のオープンに失敗したとき、例外処理を発生させて標準エラー出力に、エラーを表示します。

（例外処理発生C++プログラム例）

```
void main(void)
{
    try
    {
        if ((fopen("INPUT.DAT","r"))==NULL){
            char * cp = "cannot open input file¥n";
            throw cp;
        }
    }
    catch(char *pstrError)
    {
        fprintf(stderr,pstrError);
        abort();
    }
    return;
}
```

■注意事項

コード性能が低下する場合があります。

8.3.4 プレリンカの起動抑止

■説明

プレリンカを起動するとリンク速度が遅くなりますが、C++のテンプレート機能、実行時型変換を使用していないときは動作させる必要はありません。
リンカをコマンドラインでご使用の場合は、以下の noprelink オプションを指定してください。
Hew をご使用の場合は、Prelinker control リストボックスが Auto であれば、自動で noprelink オプションの出力を制御します。

■指定方法

ダイアログメニュー：Link/LibraryタブCategory:[Input]のPrelinker control
コマンドライン　　：*noprelink*

8.4 C++記述のメリット・デメリット

コンパイラは C++プログラムをコンパイルする際、内部的に C++プログラムを C プログラムに変換してオブジェクトを生成します。

本章では C++プログラムと変換後の C プログラムを比較し、各機能のコード効率への影響を記述します。

No.	機能	開発・保守	サイズ	処理速度	参照
1	コンストラクタ (1)	◎	△	△	8.4.1
2	コンストラクタ (2)	◎	△	△	8.4.2
3	デフォルトパラメータ	◎	○	○	8.4.3
4	インライン展開	○	△	○	8.4.4
5	クラスメンバ関数	◎	△	△	8.4.5
6	operator 演算子	◎	△	△	8.4.6
7	関数のオーバロード	◎	○	○	8.4.7
8	リファレンス型	◎	○	○	8.4.8
9	スタティック関数	◎	○	○	8.4.9
10	スタティックメンバ変数	◎	○	○	8.4.10
11	匿名 union	◎	○	○	8.4.11
12	仮想関数	◎	△	△	8.4.12

◎ : 性能低下なし ○: 使用上注意要 △: 性能低下

8.4.1 コンストラクタ (1)

開発・保守	◎	サイズ	△	処理速度	△

■ポイント

コンストラクタを使うとクラスオブジェクトを自動的に初期化できますが、以下のようにオブジェクトサイズや処理速度に影響するため、注意が必要です。

■使用例

クラスAのコンストラクタとデストラクタを作成しコンパイルします。クラス宣言箇所でコンストラクタ/デストラクタの呼び出しが入り、コンストラクタ／デストラクタ本体では判定が加わるためサイズ／処理速度に影響がでます。

```
（C＋＋プログラム）

class A
{
private:
    int a;
public:
    A(void);
    ~A(void);
    int getValue(void){ return a; }
};

void main(void)
{
    A a;
    b = a.getValue();
}

A::A(void)
{
    a = 1234;
}

A::~A(void)
{
}
```


RENESAS

（変換後のＣプログラム）

```
struct A {
    int a;
};

void *_nw__FUl(unsigned long);
void __dl__FPv(void *);
void main(void);
struct A *__ct__A(struct A *);
void __dt__A(struct A *const, int);

void main(void)
{
    struct A a;
    __ct__A(&a);
    _ b = ((a.a));
    __dt__A(&a, 2);
}
```

コンストラクタ呼び出し

デストラクタ呼び出し

```
struct A * __ct__A( struct A *this)
{
    if ( this != (struct A *)0
    ||  (  this = (struct A *)_nw__FUl(4))
            != (struct A *)0 )
    {
        (this->a) = 1234;
    }
    return this;
}
```

コンストラクタ本体

```
void __dt__A( struct A *const this,
  int flag)
{
    if (this != (struct A *)0){
        if (flag & 1) {
            dl__FPv((void *)this);
        }
    }
    return;
}
```

デストラクタ本体

RENESAS

8.4.2 コンストラクタ (2)

開発・保守	◎	サイズ	△	処理速度	△

■ポイント

クラスを配列で宣言する場合に、コンストラクタを使うとクラスオブジェクトを自動的に初期化できますが、以下のようにオブジェクトサイズや処理速度に影響するため注意が必要です。

■使用例

クラス A のコンストラクタとデストラクタを作成しコンパイルします。クラス宣言箇所でコンストラクタ／デストラクタの呼び出しが入りますが配列で宣言しているため、動的なメモリの割り当て／解放が必要になります。

動的なメモリの割り当て／解放のために、new,delete を使用します。

そのため、低水準関数を実装する必要があります。（実装方法は SuperH RISC engine C/C++**コンパイラ、アセンブラ、最適化リンケージエディタ　ユーザーズマニュアル**の「9.2.2　**実行環境の設定**」参照）

コンストラクタ／デストラクタ本体では判定と低水準関数の処理が加わるためサイズ／処理速度に影響が出ます。

```
（C++プログラム）
class A
{
private:
    int a;
public:
    A(void);
    ~A(void);
    int getValue(void){ return a; }
};

void main(void)
{
    A a[5];
    b = a[0].getValue();
}

A::A(void)
{
    a = 1234;
}

A::~A(void)
{
}
```


RENESAS

（変換後のＣプログラム）

```
struct A {
    int a;
};

void *__nw__FUl(unsigned long);
void __dl__FPv(void *);
void main(void);
void *__vec_new();
void __vec_delete();
struct A *__ct__A(struct A *);
void __dt__A(struct A *const, int);

void main(void)
{
    struct A a[5];
    __vec_new( (struct A *)a, 5, 4,
__ct__A);
    _ b = ((a.a));
    __vec_delete( &a, 5, 4, __dt__A, 0,
0);
}
```



```
struct A *__ct__A( struct A *this)
{
    if((this != (struct A *)0)
     || ( (this = (struct A
*)__nw__FUl(4)) != (struct A *)0) )
    {
        (this->a) = 1234;
    }
    return this;
}
```



```
void __dt__A( struct A *const this,
 int flag)
{
    if (this != (struct A *)0){
        if (flag & 1){
            __dl__FPv((void *)this);
        }
    }
    return;
}
```

8.4.3　デフォルトパラメータ

開発・保守	◎	サイズ	○	処理速度	○

■ポイント

C++では、デフォルトパラメータという、関数呼び出し時のデフォルト用法が使えます。

これは関数の宣言時に関数のパラメータにデフォルト値指定をすることにより使うことができます。

これにより多くの関数呼び出しではパラメータ指定の必要がなく、デフォルトのパラメータ値を使用することができ、開発効率が上がります。

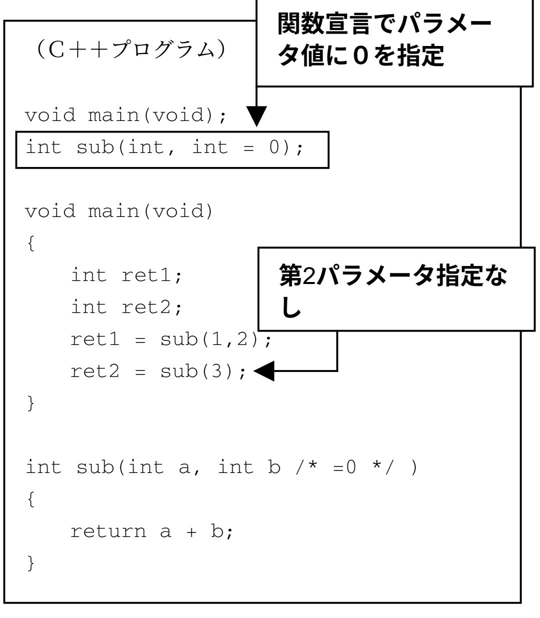
また、パラメータを指定するとパラメータの値を変更することができます。

■使用例

関数 sub の宣言にデフォルトパラメータ値として、0を指定した場合の関数 sub 呼び出し例を以下に示します。

以下のように、関数 sub 呼び出しの際に、デフォルトパラメータの値で良い場合はパラメータを記述する必要がありません。しかも、C に変換してもプログラムの効率は悪化しておりません。

このようにデフォルトパラメータは開発・保守効率が良いことに加え、C 言語と比較してもデメリットがありません。



（変換後のＣプログラム）

```
void main(void);
int sub(int, int);

void main(void)
{
    int ret1;
    int ret2;
    ret1 = sub(1, 2);
    ret2 = sub(3, 0);
}

int sub( int a,  int b)
{
    return a + b;
}
```

デフォルトパラメータ値に変換

8.4.4 インライン展開

開発・保守	○	サイズ	△	処理速度	○

■ポイント

関数の本体定義を記述する際、先頭に inline を指定すると、その関数をインライン展開するので関数呼び出しのオーバヘッドがなくなり処理速度を向上させることができます。

■使用例

関数 sub を inline 指定し、メイン関数内にインライン展開します。その後に関数 sub 本体を削除します。

ただし、他のファイルから関数 sub を参照することはできません。

インライン展開は処理速度が確実に向上しますが、小さい関数に限定しないと、サイズが大きくなるので注意が必要です。

（C＋＋プログラム）

```
int a;

inline int sub(int x, int y)
{
    return (x+y);
}

void main(void)
{
    a = sub(1,2);
}
```

（変換後のＣプログラム）

```
int a;
void main(void)
{
    a = 3;
    return;
}
```

関数subの内容を展開（1 + 2 = 3）

RENESAS

8.4.5 クラスメンバ関数

開発・保守	◎	サイズ	△	処理速度	△

■ポイント

クラスを定義すると、情報隠蔽が可能になり、開発・保守の効率が上がります。
しかし、サイズ／処理速度に影響が出るので注意が必要です。

■使用例

private クラスメンバ変数 a,b,c をアクセスするクラスメンバ関数 set,add を例とします。
クラスメンバ関数呼び出し時、C++プログラム上のパラメータ指定では、値のみもしくはパラメータなしです
しかし、変換後の C プログラムを見るとクラス A（struct A）のアドレスもパラメータとして渡されます。
また、クラスメンバ関数本体で private クラスメンバ変数 a,b,c をアクセスをしています。
しかし、this ポインタを使用してアクセスすることになります。
以上のことから、クラスメンバ関数を使用すると、サイズ／処理速度に影響が出るので注意が必要です。

（C＋＋プログラム）

```
class A
{
private:
    int a;
    int b;
    int c;
public:
    void set(int, int, int);
    int add();
};

int main(void)
{
    A a;
    int ret;

    a.set(1,2,3);
    ret = a.add();

    return ret;
}
void A::set(int x, int y, int z)
{
    a = x;
    b = y;
    c = z;
}
int A::add()
{
    return (a += b + c);
}
```

（変換後のＣプログラム）

```
struct A {
    int a;
    int b;
    int c;
};
void set__A_int_int(struct A *const, int, int, int);
int add__A(struct A *const);

int main(void)
{
    struct A a;
    int ret;

    set__A_int_int(&a, 1, 2, 3);
    ret = add__A(&a);

    return ret;
}
void set__A_int_int(struct A *const this, int x, int y, int z)
{
    this->a = x;
    this->b = y;
    this->c = z;
    return;
}
int add__A(struct A *const this)
{
    return (this->a += this->b + this->c);
}
```

8.4.6 operator 演算子

開発・保守	○	サイズ	△	処理速度	△

■ポイント

C++では operator というキーワードにより、演算子を多重定義することができます。
これによりマトリックス演算やベクトル計算などのユーザの演算処理を、簡潔に記述することができます。
しかし、operator を使う場合はサイズ／処理速度に影響が出るので注意が必要です。

■使用例

以下は単項演算子 " + " を operator キーワードで多重定義している例です。
これで下記 Vector クラスを宣言した場合、単項演算子 " + " がユーザの演算処理に変更できます。
しかし、変換後の C プログラムを見ると、this ポインタによる参照が行われているため、サイズ／処理速度に影響があります。

（C++プログラム）

```
class Vector
{
private:
    int x;
    int y;
    int z;
public:
    Vector & operator+ (Vector &);
};

void main(void)
{
    Vector a,b,c;

    a = b + c;
}

Vector & Vector::operator+ (Vector & vec)
{
    static Vector ret;

    ret.x = x + vec.x;
    ret.y = y + vec.y;     ← ユーザの演算処理（加算）
    ret.z = z + vec.z;

    return ret;
}
```


RENESAS

（変換後のCプログラム）

```
struct Vector {
    int x;
    int y;
    int z;
};

void main(void);
struct Vector *__plus__Vector_Vector(struct Vector *const, struct Vector *);

void main(void)
{
    struct Vector a;
    struct Vector b;
    struct Vector c;

    a = *__plus__Vector_Vector(&b, &c);
    return;
}

struct Vector *__plus__Vector_Vector( struct Vector *const this,  struct
Vector *vec)
{
    static struct Vector ret;

    ret.x = this->x + vec->x;
    ret.y = this->y + vec->y;
    ret.z = this->z + vec->z;

    return &ret;
}
```

thisポインタでの参照

RENESAS

8.4.7 関数のオーバロード

開発・保守	◎	サイズ	○	処理速度	○

■ポイント

C++では異なる関数に同一名を与える“ オーバロード ”が可能です。
具体的には同じような処理をする関数で、引数の型だけが違う場合などに有効な機能です。
共通性のない関数に同じ名前をつけると、不具合のもとになるので注意が必要です。
本機能を使用しても、サイズ／処理速度に影響を与えません。

■使用例

第 1､第 2 パラメータ同士を加算し、加算した値を戻り値とする例です。
関数名はすべて add で、それぞれパラメータと戻り値の型が違います。
変換後の C プログラムのとおり、add 関数呼び出し、add 関数本体でコードサイズの増加はありません。
よって、サイズ／処理速度に影響を与えません。

（C＋＋プログラム）

```
void main(void);
int add(int,int);
float add(float,float);
double add(double,double);

void main(void)
{
    int    ret_i = add(1, 2);
    float  ret_f = add(1.0f, 2.0f);
    double ret_d = add(1.0, 2.0);
}

int add(int x,int y)
{
    return x+y;
}

float add(float x,float y)
{
    return x+y;
}

double add(double x,double y)
{
    return x+y;
}
```

（変換後のＣプログラム）

```
void main(void);
int add__int_int(int, int);
float add__float_float(float, float);
double add__double_double(double, double);

void main(void)
{
    auto int ret_i;
    auto float ret_f;
    auto double ret_d;

    ret_i = add__int_int(1, 2);
    ret_f = add__float_float(1.0f, 2.0f);
    ret_d = add__double_double(1.0, 2.0);
}

int add__int_int( int x,  int y)
{
    return x + y;
}

float add__float_float( float x,  float y)
{
    return x + y;
}

double add__double_double( double x,  double y)
{
    return x + y;
}
```


RENESAS

8.4.8 リファレンス型

開発・保守	◎	サイズ	○	処理速度	○

■ポイント

パラメータをリファレンス型とすると、プログラムを簡潔に記述することができるので、開発・保守性が向上します。
また、リファレンス型を使用しても、サイズ／処理速度に影響を与えません。

■使用例

以下のようにポインタ渡しでなく、リファレンス型で渡すと、記述が簡潔になります。
また、リファレンス型はパラメータ a,b の値でなく、a,b のアドレスが渡されます。
リファレンス型を使用しても変換後のＣプログラムのとおり、サイズ／処理速度に影響を与えません。

（C＋＋プログラム）

```
void main(void);
void swap(int&, int&);

void main(void)
{
    int a=100;
    int b=256;

    swap(a,b);
}

void swap(int &x, int &y)
{
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

（変換後のＣプログラム）

```
void main(void);
void swap(int *, int *);

void main(void)
{
    int a=100;
    int b=256;

    swap(&a, &b);
}

void swap(int *x,  int *y)
{
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}
```

8.4.9 スタティック関数

開発・保守	◎	サイズ	○	処理速度	○

■ポイント

クラスの構成が派生クラスなどにより、複雑になってくると、private 属性の static なクラスメンバ変数のアクセスが大変になり、public 属性にすることになってしまいます。

このようなときに、private 属性のまま static なクラスメンバ変数をアクセスするには、インタフェース用のメンバ関数を作成し、その関数に static を指定します。

このように、static なクラスメンバ変数のみにアクセスするのが、スタティック関数です。

■使用例

次紙のようにスタティック関数を使って、スタティックメンバ変数をアクセスします。

クラスを使用するのでクラスはコード効率に影響を与えますが、スタティック関数自体はサイズ／処理速度に影響を与えません。

■備考

スタティックメンバ変数については、「8.2.3　**スタティックメンバ変数**」を参照してください。

（C＋＋プログラム）

```
class A
{
private:
    static int num;              ← スタティックメンバ変数
public:
    static int getNum(void);     ← スタティック関数
    A(void);
    ~A(void);
};

int A::num = 0;

void main(void)
{
    int num;

    num = A::getNum();

    A a1;
    num = a1.getNum();

    A a2;
    num = a2.getNum();
}

A::A(void)
{
    ++num;
}

A::~A(void)
{
    --num;
}

int A::getNum(void)
{
    return num;                  ← スタティックメンバ変数にアクセス
}
```


RENESAS

（変換後のＣプログラム）

```
struct A
{
    char __dummy;
};
void *__nw__FUl(unsigned long);
void __dl__FPv(void *);
int getNum__A(void);                    ← スタティック関数
struct A *__ct__A(struct A *);
void __dt__A(struct A *const, int);
int num__1A = 0;                        ← スタティックメンバ変数
void main(void)
{
    int num;
    struct A a1;
    struct A a2;

    num = getNum__A();

    __ct__A(&a1);
    num = getNum__A();

    __ct__A(&a2);
    num = getNum__A();

    __dt__A(&a2, 2);
    __dt__A(&a1, 2);
}
int getNum__A(void)
{
    return num__1A;                     ← スタティックメンバ変数にアクセス
}
struct A *__ct__A( struct A *this)
{
    if ( (this != (struct A *)0)
    || ( (this = (struct A *)__nw__FUl(1)) != (struct A *)0) ){
        ++num__1A;
    }
    return this;
}
void __dt__A( struct A *const this,  int flag)
{
    if (this != (struct A *)0){
        --num__1A;
        if(flag & 1){
            __dl__FPv((void *)this);
        }
    }
    return;
}
```

8.4.10 スタティックメンバ変数

開発・保守	◎	サイズ	○	処理速度	○

■ポイント

C++では、クラスのメンバ変数を static 属性にすると、そのメンバ変数はクラス型の複数のオブジェクト間で共有することができます。

これにより、同じクラス型の複数オブジェクト間で、共通なフラグなどに利用することができるので便利です。

■使用例

main 関数内で、クラス A 型のオブジェクトを 5 つ作成します。

static なメンバ変数 num の初期値は 0 です。この値がオブジェクトの作成ごとに、コンストラクタでインクリメントされます。

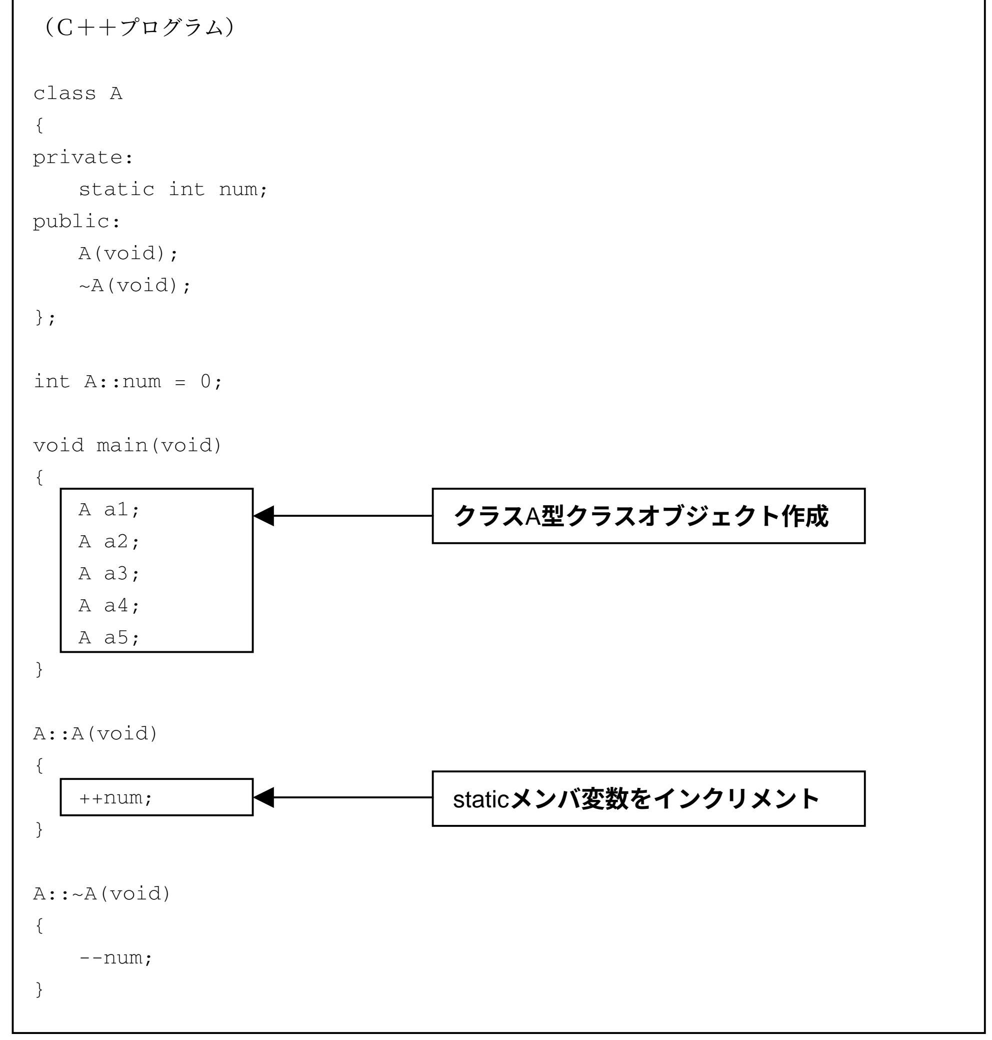
static なメンバ変数 num は各オブジェクト間で共有されるので、変数 num の値は 5 まで上昇します。

また、クラスを使用するのでクラス自体はコード効率に影響を与えます。

しかし、スタティックメンバ変数自体は、コンパイラの内部でメンバ変数 num は通常のグローバル変数のように扱われるので、サイズ／処理速度に影響を与えません。

■備考

スタティックメンバ変数の詳細は、「8.2.3 **スタティックメンバ変数**」を参照してください。



（C++プログラム）

```
class A
{
private:
    static int num;
public:
    A(void);
    ~A(void);
};

int A::num = 0;

void main(void)
{
    A a1;
    A a2;
    A a3;
    A a4;
    A a5;
}

A::A(void)
{
    ++num;
}

A::~A(void)
{
    --num;
}
```


RENESAS

（変換後のＣプログラム）

```
struct A
{
    char __dummy;
};
void *__nw__FUl(unsigned long);
void __dl__FPv(void *);
struct A *__ct__A(struct A *);
void __dt__A(struct A *const, int);
int num__1A = 0;
void main(void)
{
    struct A a1;
    struct A a2;
    struct A a3;
    struct A a4;
    struct A a5;
    __ct__A(&a1);
    __ct__A(&a2);
    __ct__A(&a3);
    __ct__A(&a4);
    __ct__A(&a5);
    __dt__A(&a5, 2);
    __dt__A(&a4, 2);
    __dt__A(&a3, 2);
    __dt__A(&a2, 2);
    __dt__A(&a1, 2);
}
struct A *__ct__A( struct A *this)
{
    if( (this != (struct A *)0)
    || ( (this = (struct A *)__nw__FUl(1)) != (struct A *)0) ){
        ++num__1A;
    }
    return this;
}
void __dt__A( struct A *const this,  int flag)
{
    if(this != (struct A *)0){
        --num__1A;
        if (flag & 1){
            __dl__FPv((void *)this);
        }
    }
    return;
}
```

8.4.11 匿名 union

開発・保守	◎	サイズ	○	処理速度	○

■ポイント

C++で匿名 union を使うと、C 言語とは異なりメンバ名を指定せずに、メンバをダイレクトにアクセスすることができます。

これにより開発効率が向上します。また、サイズや処理速度に影響しません。

■使用例

以下のように関数 main で、union のメンバ変数 s をアクセスする場合を例とします。

C++プログラムではメンバ変数 s をダイレクトにアクセスしていますが、変換後の C プログラムではコンパイラが自動でメンバ名を作成し、このメンバ名を用いてアクセスしています。

このように簡潔な記述で、オブジェクト効率に影響を与えず、メンバ変数にアクセスできます。

（C＋＋プログラム）

```
struct tag {
    union {
        unsigned char  c[4];
        unsigned short s[2];
        unsigned long  l;
    };
};

void main(void)
{
    tag t;
    t.s[1] = 1;
}
```

メンバ名なし

（変換後のCプログラム）

```
struct tag {
    union _uni {
        unsigned char  c[4];
        unsigned short s[2];
        unsigned long  l;
    } access;
};

void main(void)
{
    struct tag t;
    t.access.s[1] = 1;
}
```

メンバ名あり

RENESAS

8.4.12 仮想関数

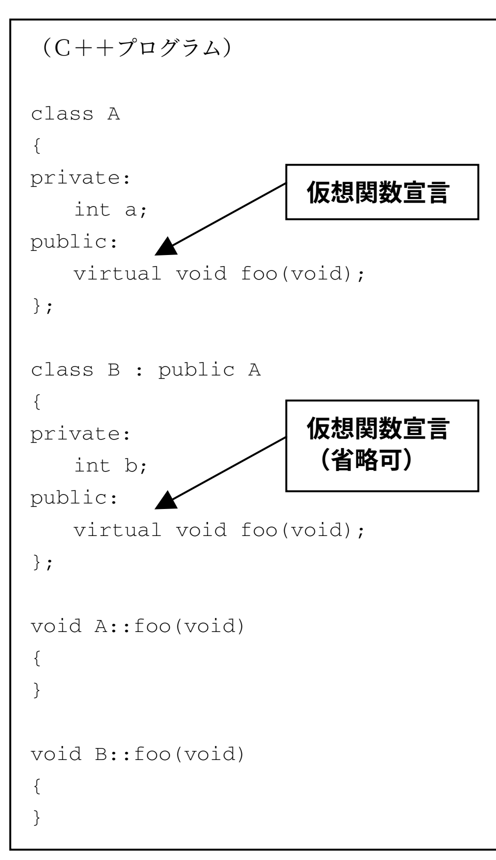
開発・保守	◎	サイズ	△	処理速度	△

■ポイント

仮想関数を使わない場合、以下のプログラムのような、基本クラス、派生クラスにそれぞれ存在する同名の関数があった場合、意図どおりに正しく関数呼び出しをすることができません。
仮想関数を宣言すると、上記の呼び出しが意図どおりに正しく行うことができます。
仮想関数を使うと、開発効率が向上しますが、サイズや処理速度に影響を与えるので注意が必要です。

■使用例

main3 関数の呼び出しでは、2 つのポインタにクラス B のアドレスを格納しています。
virtual 宣言をしているので、正しくクラス B の foo 関数を呼び出します。
virtual 宣言をしないと、一方はクラス A の foo 関数を呼び出します。
また、仮想関数を使うと次紙以降に示す、テーブルなどを作成するため、サイズや速度に影響を与えます。




```
void main1(void)
{
    A a;
    a.foo();
}

void main2(void)
{
    B b;
    b.foo();
}

void main3(void)
{
    B b;
    A * pa = &b;
    B * pb = &b;

    pa->foo();
    pb->foo();
}
```

仮想関数
呼び出し

■変換後のCプログラム（仮想関数のためのテーブルなど）

```
struct __T5585724;
struct __type_info;
struct __T5584740;
struct __T5579436;
struct A;
struct B;
extern void main1__Fv(void);
extern void main2__Fv(void);
extern void main3__Fv(void);
extern void foo__1AFv(struct A *const);
extern void foo__1BFv(struct B *const);
struct __T5585724
{
   struct __T5584740 *tinfo;
   long offset;
   unsigned char flags;
};
struct __type_info
{
   struct __T5579436 *__vptr;
};
struct __T5584740
{
   struct __type_info tinfo;
   const char *name;
   char *id;
   struct __T5585724 *bc;
};
struct __T5579436
{
   long d;                        // thisポインタオフセット
   long i;                        // 未使用
   void (*f)();                   // 仮想関数コール用
};
struct A {                        // クラスA宣言
   int a;
   struct __T5579436 *__vptr;       // 仮想関数テーブルへのポインタ
};
struct B {                        // クラスB宣言
   struct A __b_A;
   int b;
};
static struct __T5585724 __T5591360[1];
#pragma section $VTBL
extern const struct __T5579436 __vtbl__1A[2];
extern const struct __T5579436 __vtbl__1B[2];
extern const struct __T5579436 __vtbl__Q2_3std9type_info[];
#pragma section
extern struct __T5584740 __T_1A;
extern struct __T5584740 __T_1B;
```

```
static char __TID_1A;                   // 未使用
static char __TID_1B;                   // 未使用
static struct __T5585724 __T5591360[1] =   // 未使用
{
    {
        &__T_1A,
        0L,
        ((unsigned char)22U)
    }
};
#pragma section $VTBL
const struct __T5579436 __vtbl__1A[2] =    // クラスA用仮想関数テーブル
{
    {
        0L,                             // 未使用領域
        0L,                             // 未使用領域
        ((void (*)())&__T_1A)              // 未使用領域
    },
    {
        0L,                             // thisポインタオフセット
        0L,                             // 未使用領域
        ((void (*)())foo__1AFv)            // A::foo()へのポインタ
    }
};
const struct __T5579436 __vtbl__1B[2] =    // クラスB用仮想関数テーブル
{
    {
        0L,                             // 未使用領域
        0L,                             // 未使用領域
        ((void (*)())&__T_1B)              // 未使用領域
    },
    {
        0L,                             // thisポインタオフセット
        0L,                             // 未使用領域
        ((void (*)())foo__1BFv)            // B::foo()へのポインタ
    }
};
#pragma section
struct __T5584740 __T_1A =                 // クラスA用型情報(未使用)
{
    {
        (struct __T5579436 *)__vtbl__Q2_3std9type_info
    },
    (const char *)"A",
    &__TID_1A,
    (struct __T5585724 *)0
};
struct __T5584740 __T_1B =                 // クラスB用型情報(未使用)
{
    {
        (struct __T5579436 *)__vtbl__Q2_3std9type_info
    },
    (const char *)"B",
    &__TID_1B,
    __T5591360
};
```

■変換後のCプログラム（仮想関数の呼び出し）

```
void main1__Fv(void)
{
    struct A _a;
    _a.__vptr = __vtbl__1A;
    foo__1AFv( &_a );                          // A::foo()のコール
    return;
}
void main2__Fv(void)
{
    struct B _b;
    _b.__b_A.__vptr = __vtbl__1A;
    _b.__b_A.__vptr = __vtbl__1B;
    foo__1BFv( &_b );                          // B::foo()のコール
    return;
}
void main3__Fv(void)
{
    struct __T5579436 *_tmp;
    struct B _b;
    struct A *_pa;
    struct B *_pb;

    (*((struct A*)(&_b))).__vptr = __vtbl__1A;
    (*((struct A*)(&_b))).__vptr = __vtbl__1B;
    _pa = (struct A *)&_b;
    _pb = &_b;

    _tmp = _pa->__vptr + 1;
    (  (void (*)(struct A *const))  _tmp->f  )  (  (struct A *)_pa + _tmp->d);
    // B::foo()をコール(_paの指すオブジェクトがBのため)

    _tmp = _pb->__b_A.__vptr + 1;
    (  (void (*)(struct B *const))  _tmp->f  )  (  (struct B *)_pb + _tmp->d );
    // B::foo()をコール

    return;
}
```

9. 最適化リンケージエディタ

本章では、リンク時における便利なオプションの説明、またリンク時にモジュール間をまたいで横断的に最適化する最適化機能について説明いたします。

最適化リンケージエディタ編の一覧を示します。

No.	大項目	中項目		参照
1	入出力オプション	入力オプション		9.1.1
		出力オプション		9.1.2
2	リストオプション	シンボル情報表示		9.2.1
3		シンボル参照回数表示		9.2.2
4		クロスリファレンス情報表示		9.2.3
5	便利な機能	空きエリア出力指定		9.3.1
6		S タイプファイルの終端コード		9.3.2
7		デバッグ情報の圧縮		9.3.3
8		リンク時間の短縮		9.3.4
9		参照されない定義シンボルの通知		9.3.5
10		セクション内データの詰め込み配置		9.3.6
11	最適化機能	リンク時の最適化とは？		9.4.1
12		最適化のサブオプション説明		
13			定数/文字列の統合	9.4.2
14			未参照シンボルの削除	9.4.3
15			レジスタ退避・回復の最適化	9.4.4
16			共通コードの統合	9.4.5
17			分岐命令の最適化	9.4.6
18		最適化部分抑止		9.4.7
19		最適化結果の確認		9.4.8

9.1 入出力オプション

9.1.1 入力オプション

■説明

最適化リンケージエディタは、ユーザの使用状況に応じて以下の 4 種類のファイルを入力することができるので便利です。

■指定方法

ダイアログメニュー　：最適化リンカタブカテゴリ:[入力] オプション項目
コマンドライン　　　：*input <サブオプション>:<ファイル名>*
library <ファイル名>
binary <サブオプション>:<ファイル名>

■入力可能なファイル

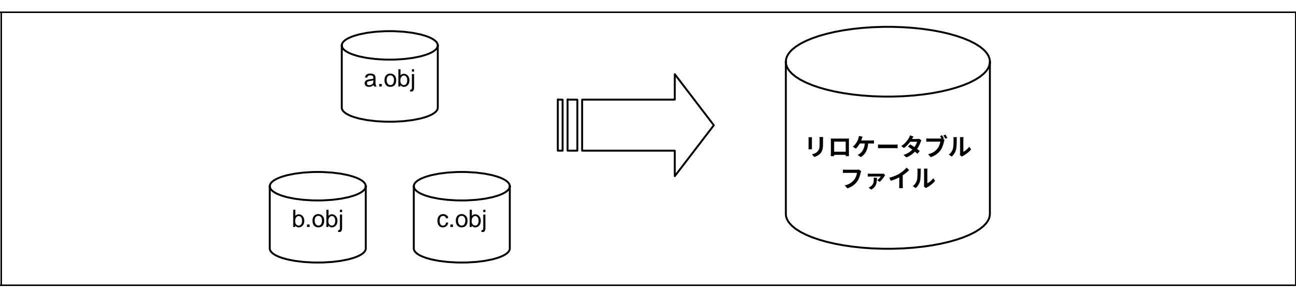
ファイルの種類	コマンドラインでの指定
オブジェクトファイル	input
リロケータブルファイル	input
ライブラリファイル	library
バイナリファイル	binary

(1) オブジェクトファイル

コンパイラ／アセンブラが出力する通常のファイルです。

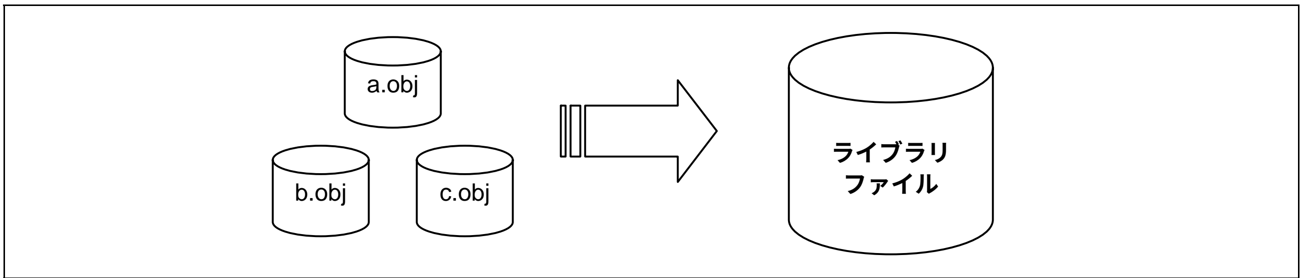
(2) リロケータブルファイル

再配置可能(アドレス解決されていない)ファイルです。
1 つ以上の複数個のオブジェクトの集まりで、最適化リンケージエディタの出力オプションで作成できます。
リロケータブルファイル内のシンボルは、他のファイルから参照されていなくてもリンクされます。
リロケータブルファイル使用時は、この点に注意しないと、不要なファイルがリンクされ ROM 容量が増えてしまいます。



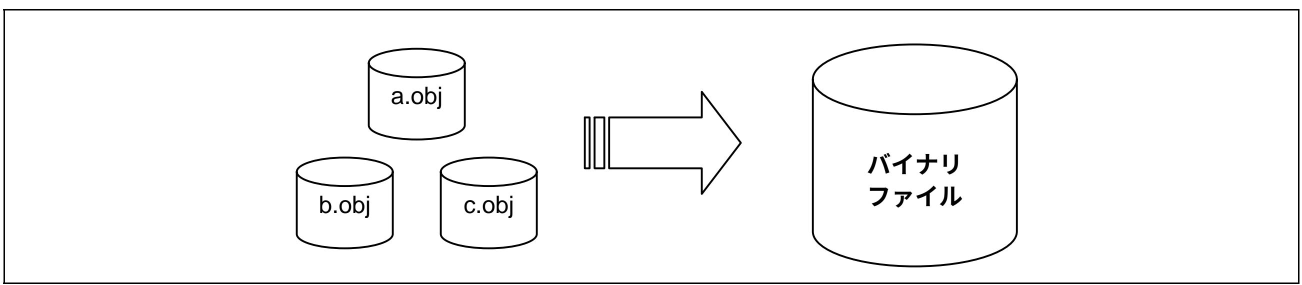
(3) ライブラリファイル

再配置可能(アドレス解決されていない)ファイルです。
複数のオブジェクトの集まりで、最適化リンケージエディタの出力オプションで作成できます。
ライブラリファイル内の参照されていないシンボルはリンクされません。



(4) バイナリファイル

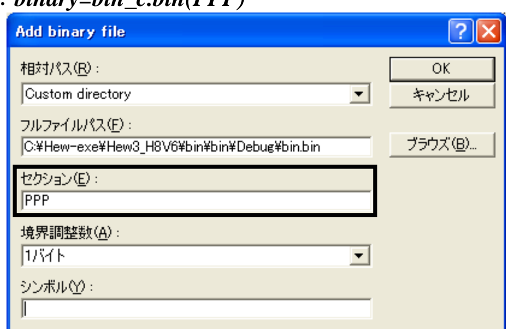
バイナリファイルを入力できます。
複数のオブジェクトの集まりで、最適化リンケージエディタの出力オプションで作成できます。
バイナリファイル入力時にはセクション名の指定が必須です。このセクション名を start オプションで配置します。
なお、デバッグ情報は付加されていないため、C/C++ソースレベルデバッグはできません。



【指定方法 1】

必ずセクション名を指定する必要があります。

ダイアログメニュー　：最適化リンカタブカテゴリ:[入力] オプション項目
　　　　　　　　　　　バイナリファイル 追加
コマンドライン　　　：***binary=bin_c.bin(PPP)***



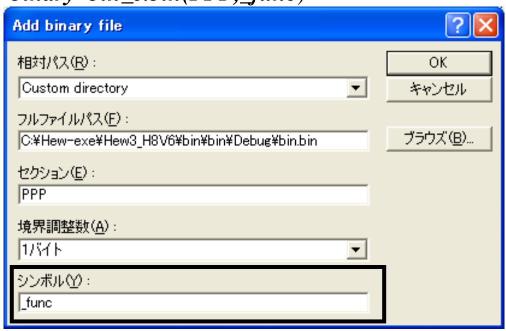
RENESAS

【指定方法 2】

バイナリファイルの先頭にシンボルを割り付けることができます。

この場合、セクション名と共にシンボル名を指定します。C/C++から参照する場合は、シンボルの先頭に‘ _ ’を付加します。

ダイアログメニュー　：最適化リンカタブカテゴリ:[入力] オプション項目
　　　　　　　　　　　バイナリファイル 追加
コマンドライン　　　：***binary=bin_c.bin(PPP,_func)***



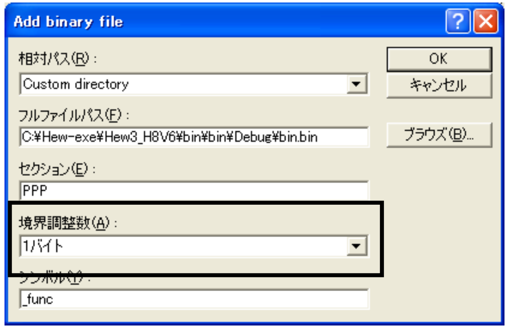
【指定方法３】

バイナリファイル入力時に境界調整数を指定することができます。

境界調整数の指定がないときは、旧バージョンとの互換性を保つために 1 として扱われます。

なお、境界調整数の指定は最適化リンケージエディタ V9.0 から可能です。

ダイアログメニュー：**最適化リンカタブカテゴリ:[入力] オプション項目 バイナリファイル 追加**
コマンドライン　　：***binary=bin_c.bin(PPP:<境界調整数>,_func)***
　　　　　　　　　<境界調整数>：1 | 2 | 4 | 8 | 16 | 32(デフォルトは 1)



9.1.2 出力オプション

■説明

ROM ライタによっては HEX ファイルしか対応していなかったり、S タイプしか対応していないことがあります。
最適化リンケージエディタは、ユーザの使用状況に応じて以下の 8 種類のファイルを出力することが出来ます。必要に応じて出力ファイルの種類を変更してください。

■指定方法

ダイアログメニュー ：最適化リンカタブカテゴリ:[出力] 出力形式
コマンドライン ：*form{ absolute | relocate | object | library=s | library=u | hexadecimal | stype | binary }*

■出力可能なファイル

No.	ファイルの種類	コマンドラインでの指定
1	アブソリュートファイル	form absolute
2	リロケータブルファイル	form relocate
3	オブジェクトファイル	form object
4	ユーザライブラリファイル	form library=s
5	システムライブラリファイル	form library=u
6	HEX ファイル	form hexadecimal
7	S タイプファイル	form stype
8	バイナリファイル	form binary

(1) **アブソリュートファイル**

最適化リンケージエディタでアドレスが解決されたファイルです。
デバッグ情報付きなので C/C++ソースレベルデバッグができます。
ROM に書き込む際は、S タイプ、HEX、バイナリのいずれかに変換する必要があります。

(2) **リロケータブルファイル**

再配置可能(アドレス解決されていない)ファイルです。
デバッグ情報付きなので C/C++ソースレベルデバッグができます。
作成後に、再びリンクを行いアブソリュートファイルにすることにより動作が可能になります。

(3) **オブジェクトファイル**

ライブラリファイルから extract オプションで、1 モジュール（オブジェクト）抽出するときに使用します。

コマンドライン指定の場合、本オプションで指定したライブラリファイルから、必要なオブジェクトファイルを取得することができます。
HEW をお使いの場合は、最適化リンカタブカテゴリ:[その他] ユーザ指定オプション: で以下のオプションを入力してください。

【取得オプション】

form=object
extract=<モジュール名>

(4) **ユーザライブラリ／システムライブラリ**

ライブラリファイルを出力します。

(5) HEX **ファイル**

HEX ファイルを出力します。
デバッグ情報が付いていないので、C/C++ソースレベルデバッグはできません。
HEX ファイルの詳細は「SuperH RISC engine C/C++**コンパイラ、アセンブラ、最適化リンケージエディタ ユーザーズマニュアル**」の「18.1.2 HEX **ファイル形式**」を参照してください。

(6) S **タイプファイル**

S タイプファイルを出力します。
デバッグ情報が付いていないので、C/C++ソースレベルデバッグはできません。
S タイプファイルの詳細は「SurperH RISC engine C/C++**コンパイラ、アセンブラ、最適化リンケージエディタ ユーザーズマニュアル**」の「18.1.1 S **タイプファイル形式**」を参照してください。

(7) **バイナリファイル**

バイナリファイルを出力します。
デバッグ情報が付いていないので、C/C++ソースレベルデバッグはできません。

9.2 リストオプション

9.2.1 シンボル情報表示

■説明

最適化リンケージエディタは、リンケージマップ情報のほかに、以下のサブオプションを指定するとシンボルアドレス、サイズ、および最適化による変更を受けたかの情報を得ることができます。

① シンボルアドレス-ADDR
② サイズ-SIZE
③ 最適化-OPT(ch-変更、cr-新規作成、mv-移動)

■指定方法

ダイアログメニュー ：最適化リンカタブカテゴリ:[リスト] リスト内容 シンボル情報
コマンドライン ：*list [=<ファイル名>]*
show symbol

```
<*.mapファイル>
*** Options ***
         :
*** Error information ***
         :
*** Mapping List ***
         :
*** Symbol List ***

SECTION=
FILE=                                   START        END      SIZE
  SYMBOL                               ①ADDR       ②SIZE    INFO    COUNTS   ③OPT

SECTION=P
FILE=C:¥Hew-exe¥Hew3_SHV9¥bin¥bin¥Debug¥bin.obj
                                        00000800   00000821        22
  _main
                                        00000800          6   func ,g       *     ch
  _abort
                                        00000806          4   func ,g       *     ch
  _com_opt1
                                        0000080a         18   func ,g       *     cr ch
*** Delete Symbols ***
         :
*** Variable Accessible with Abs8 ***
         :
*** Variable Accessible with Abs16 ***
         :
*** Function Call ***
         :
```

9.2.2 シンボル参照回数表示

■説明

最適化リンケージエディタは、リンケージマップ情報のほかに、以下のサブオプションを指定すると静的なシンボル参照回数を調査できます。

① 参照回数**-COUNTS**

■指定方法

ダイアログメニュー ：最適化リンカタブカテゴリ:[リスト] リスト内容 参照回数
コマンドライン ：*list [=<ファイル名>]*
show reference

```
<*.mapファイル>
*** Options ***
          :
*** Error information ***
          :
*** Mapping List ***
          :
*** Symbol List ***

SECTION=
FILE=                                   START         END      SIZE
  SYMBOL                              ADDR        SIZE    INFO            ①COUNTS   OPT

SECTION=P
FILE=C:¥Hew-exe¥Hew3_SHV9¥bin¥bin¥Debug¥bin.obj
                                       00000800   00000821          22
   _main
                                      00000800            6     func ,g          1   ch
   _abort
                                      00000806            4     func ,g          0   ch
   _com_opt1
                                      0000080a           18     func ,g          2   cr ch
*** Delete Symbols ***
          :
*** Variable Accessible with Abs8 ***
          :
*** Variable Accessible with Abs16 ***
          :
*** Function Call ***
```

9.2.3 クロスリファレンス情報表示

■説明

最適化リンケージエディタは、リンケージマップ情報の他に、以下のサブオプションを指定すると、グローバルシンボルがどこで参照されているか、調べることができるクロスリファレンス情報を出力します。

ローカルシンボルやスタティックシンボルについては出力しません。

■指定方法

ダイアログメニュー：**最適化リンカタブカテゴリ:[リスト] リスト内容 クロスリファレンス情報**

コマンドライン　：*list [=<ファイル名>]*

show xreference

<*.map ファイル>

```
*** Cross Reference List ***

No   Unit Name    Global.Symbol    Location   External Information
(1)     (2)            (3)            (4)             (5)
---- ----------- -------------- -------- ---------------------
0001 test1
       SECTION=P
                     _main
                                  00000100
       SECTION=B
                     _sl1
                                  00007000 0001(0000011a:P)
                     _sl2
                                  00007004 0001(0000010e:P)
                     _ret
                                  00007008 0001(00000128:P)
       SECTION=D
0002 test2
       SECTION=P
                     _func1
                                  0000015c 0001(00000124:P)
                     _func2
                                  00000164 0001(0000013c:P)
                     _func3
                                  00000170 0001(00000150:P)
```

■各項目説明

(1) オブジェクトファイル単位の識別番号です。[External Information]に、この識別番号が表示されます。
(2) オブジェクトファイル名。リンク時の入力順に表示されます。
(3) 外部シンボル名。セクションごとに昇順に表示されます。
(4) 外部シンボルの配置アドレスを表示します。出力ファイル形式にリロケータブル形式(form=relocate)を選択している場合はセクション先頭からの相対値となります。
(5) 外部シンボルを参照している場所のアドレスを表示します。
出力形式は以下のようになります。
<Unit番号>　(<アドレス or セクション内オフセット>:<セクション名>)

■備考

本オプションは最適化リンケージエディタ Ver.9 から対応しています。

9.3 便利な機能

9.3.1 空きエリア出力指定

■説明

最適化リンケージエディタでは空きエリアに任意のデータを書き込むことができます。

ROM 伝送、またはプログラムが暴走し、データのない空きエリアを実行したときの異常割り込み検出をしたいときなどに便利です。

また、出力データのサイズは 1、2、4 バイト単位で有効となります。奇数バイトデータを入力した場合は、上位桁に 0 拡張し偶数桁として扱われます。

出力データの最大サイズは 4 バイトで、4 バイトを超えるデータを指定すると、下位 4 バイトが有効となります。

本オプションは出力ファイルが S タイプファイル、バイナリファイル、HEX ファイルのとき、有効です。

■指定方法

ダイアログメニュー　：最適化リンカタブカテゴリ:[出力] オプション項目
　　空きエリア出力指定

コマンドライン　：*space [=<数値>]*

■指定例

（1） 最適化リンカタブカテゴリ:[出力] オプション項目 出力ファイルの分割(output)でファイル分割し、空きエリアをデータで埋めたい範囲を決める。
-output="C:¥bin¥Debug¥a.bin"=00-0FFFF

（2） 最適化リンカタブカテゴリ:[出力] オプション項目 空きエリア出力指定(space)で埋め込むデータを指定する。
-space=FF

次ページからの例における＜空きエリア出力指定あり[H'FF **を設定**]＞のようにデータが埋め込まれます。

■S タイプファイルでの例

以下のようにデータが存在する範囲の空きエリアに、0xFF のレコードが追加されています。

また、本オプションを指定しない場合、データが存在する範囲以外はレコードを出力していませんが、本オプションを指定すると、空きエリア出力ファイルの分割で決めた範囲で、元々データが存在しない場所にも 0xFF のレコードが追加されます。

＜空きエリア出力指定なし＞

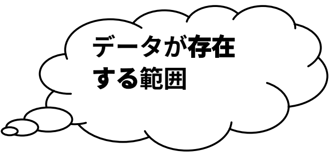
```
S00E000062696E20202020206D6F74C8
S107000000000400F4
S10700140000041AC6
S107001C0000041CBC
S10700200000041EB6
S1070024000004200B0
S10700280000422AA
S107002C0000424A4
S10700400000004268E
S107004400000042888
S10700480000042A82
S107004C0000042C7C
S10700500000042E76
S107005400000043070
```

＜途中省略＞

```
S11308901F9045EC7A00000008C67A01000008D2D7
S11308A0401801006D0401006D0501006D0640064D
S11308B06C4A68EA0B061FD445F61F9045E40120F4
S10908C06D766D725470A8
S10F08C6000008DA000008DE00FFE42A4D
S10B08D200FFE00000FFE42A2E
S10708DA00FFE00A2D
S10F08DE7900000A6BA00000200C54708C
S9030400F8
```

＜空きエリア出力指定あり[H'FF **を設定**]＞

```
S00E000062696E20202020206D6F74C8
S107000000000400F4
S1130004FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF8
S10700140000041AC6
S1070018FFFFFFFFE4
S107001C0000041CBC
S10700200000041EB6
S1070024000004200B0
S10700280000422AA
S107002C0000424A4
S1130030FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFCC
S10700400000004268E
S107004400000042888
S10700480000042A82
```



＜途中省略＞

```
S113FF8AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF73
S113FF9AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF63
S113FFAAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF53
S113FFBAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF43
S113FFCAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF33
S113FFDAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF23
S113FFEAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF13
S109FFFAFFFFFFFFFFFF03
S9030400F8
```

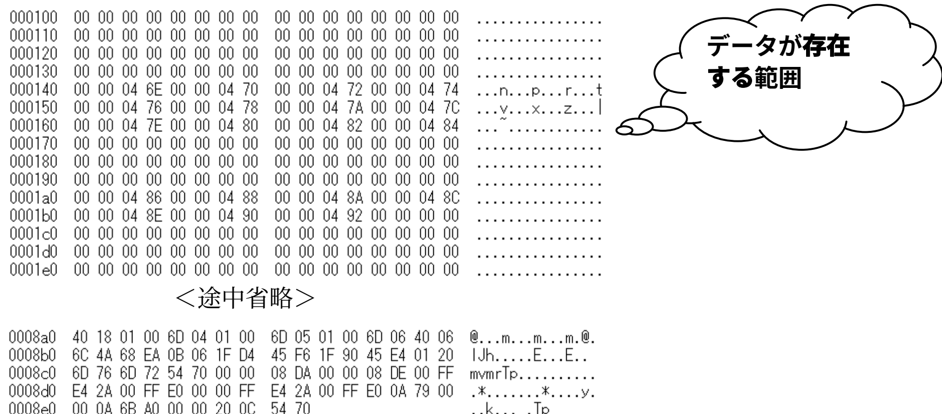
RENESAS

■バイナリファイルでの例

以下のようにデータが存在する範囲の空きエリアが 0x00 から 0xFF に変化します。

また、本オプションを指定しない場合、データが存在する範囲以外はデータを出力しませんが本オプションを指定すると、空きエリア出力ファイルの分割で決めた範囲で、元々データが存在しない場所にも 0xFF が格納されます。

＜空きエリア出力指定なし＞

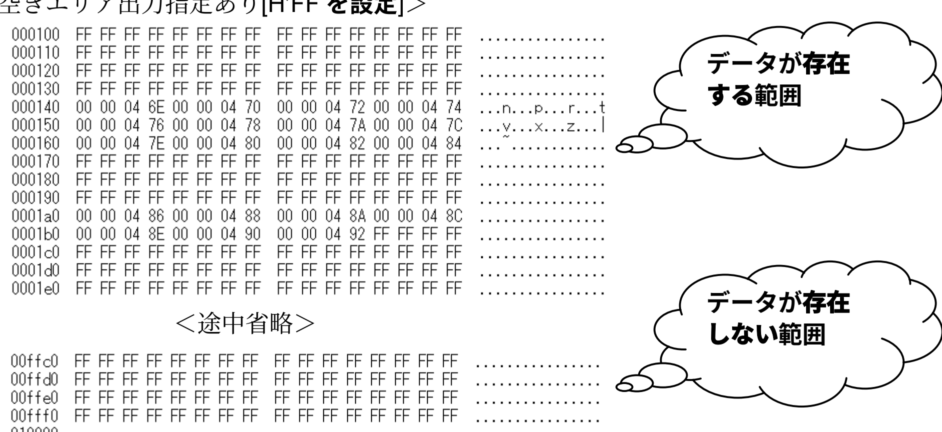


```
000100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000120  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000130  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000140  00 00 04 6E 00 00 04 70  00 00 04 72 00 00 04 74  ...n...p...r...t
000150  00 00 04 76 00 00 04 78  00 00 04 7A 00 00 04 7C  ...v...x...z...|
000160  00 00 04 7E 00 00 04 80  00 00 04 82 00 00 04 84  ...~............
000170  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0001a0  00 00 04 86 00 00 04 88  00 00 04 8A 00 00 04 8C  ................
0001b0  00 00 04 8E 00 00 04 90  00 00 04 92 00 00 00 00  ................
0001c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0001d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0001e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

＜途中省略＞

```
0008a0  40 18 01 00 6D 04 01 00  6D 05 01 00 6D 06 40 06  @...m...m...m.@.
0008b0  6C 4A 68 EA 0B 06 1F D4  45 F6 1F 90 45 E4 01 20  lJh.....E...E..
0008c0  6D 76 6D 72 54 70 00 00  08 DA 00 00 08 DE 00 FF  mvmrTp..........
0008d0  E4 2A 00 FF E0 00 00 FF  E4 2A 00 FF E0 0A 79 00  .*.......*....y.
0008e0  00 0A 6B A0 00 00 20 0C  54 70                    ..k... .Tp
```


＜空きエリア出力指定あり[H'FF **を設定**]＞



```
000100  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000110  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000120  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000130  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000140  00 00 04 6E 00 00 04 70  00 00 04 72 00 00 04 74  ...n...p...r...t
000150  00 00 04 76 00 00 04 78  00 00 04 7A 00 00 04 7C  ...v...x...z...|
000160  00 00 04 7E 00 00 04 80  00 00 04 82 00 00 04 84  ...~............
000170  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000180  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000190  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
0001a0  00 00 04 86 00 00 04 88  00 00 04 8A 00 00 04 8C  ................
0001b0  00 00 04 8E 00 00 04 90  00 00 04 92 FF FF FF FF  ................
0001c0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
0001d0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
0001e0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
```

＜途中省略＞

```
00ffc0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00ffd0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00ffe0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00fff0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
010000
```

■HEX ファイルでの例

以下のようにデータが存在する範囲の空きエリアに、0xFF のレコードが追加されています。

また、本オプションを指定しない場合、データが存在する範囲以外はレコードを出力していませんが、本オプションを指定すると、空きエリア出力ファイルの分割で決めた範囲で、元々データが存在しない場所にも 0xFF のレコードが追加されます。

＜空きエリア出力指定なし＞

```
:0400000000000400F8
:04001400000041ACA
:04001C000000041CC0
:040020000000041EBA
:040024000000004 20B4
:0400280000000422AE
:04002C0000000424A8
:0400400000000042692
:04004400000004288C
:0400480000000042A86
:04004C000000042C80
```

＜途中省略＞

```
:0C08C600000008DA000008DE00FFE42A51
:0808D20000FFE00000FFE42A32
:0408DA0000FFE00A31
:0C08DE007900000A6BA00000200C547090
:00000001FF
:0400000300000400F5
```

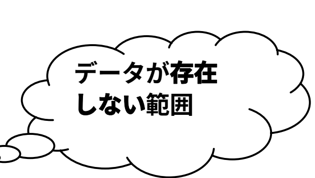
＜空きエリア出力指定あり[H'FF **を設定**]＞

```
:0400000000000400F8
:10000400FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC
:04001400000041ACA
:04001800FFFFFFFFE8
:04001C000000041CC0
:040020000000041EBA
:04002400000000420B4
:0400280000000422AE
:04002C0000000424A8
:10003000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD0
:0400400000000042692
```

データが存在
する範囲

＜途中省略＞

```
:FFFCF500FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFI
:FFFDF400FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFI
:FFFEF300FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFI
:0EFFF200FFFFFFFFFFFFFFFFFFFFFFFFFFFF0F
:00000001FF
:0400000300000400F5
```



■備考

本オプションは最適化リンケージエディタ Ver.8 から対応しています。

9.3.2 S タイプファイルの終端コード

■説明

ROM ライタの種類によっては、S タイプファイルの終端コードが S9 レコードでないと、ROM ライタへの入力時にタイムアウトエラーになってしまうことがあります。

これはエントリアドレスが 0x10000 を超えると終端コード S7 や S8 になってしまうためです。

本オプションを指定することにより、終端コードを常に S9 レコードにすることができます。

■指定方法

ダイアログメニュー ：最適化リンカタブカテゴリ:[**その他**] S9コートを終端に出力
コマンドライン ：*S9*

■備考

S タイプファイルの詳細は SuperH RISC engine C/C++**コンパイラ、アセンブラ、最適化リンケージエディタユーザーズマニュアル**の「18.1.1 S **タイプファイル形式**」を参照してください。

9.3.3 デバッグ情報の圧縮

■説明

デバッガへファイルをロードするときは、本オプションを指定するとロード時間が短くなります。

しかし、反対にリンク時間は長くなります。

■指定方法

ダイアログメニュー ：最適化リンカタブカテゴリ:[**その他**] デバッグ情報圧縮
コマンドライン ：*compress*
nocompress

■備考

本オプションは出力ファイルがアブソリュートファイル時のみ有効です。

9.3.4 リンク時間の短縮

■説明

本オプションを指定すると、リンク時に必要な情報のロードを細かく行うことにより、使用するメモリ量の削減を行います。

その結果、リンク時間を短縮できる場合があります。

大規模なプロジェクトをリンクした際に、メモリ使用量が稼動マシンの実装メモリ量を超えてしまい、動作が遅くなっているような場合にお試しください。

■指定方法

ダイアログメニュー　：最適化リンカタブカテゴリ:[**その他**]
入力ファイルロード時のメモリ使用量削減

コマンドライン　：*memory={high | low}*

■具体例

本オプション指定有無によるリンク時間比較を下記に示します。

下記のケースでは、34% リンク時間が短縮されています。

＜測定環境＞

・ファイル数は1000ファイル
・1ファイルのシンボル数が100
・関数シンボル数1000
・本オプション以外は同一なオプション指定

＜memory=high＞

111 秒

＜memory=low＞

73 秒

■備考

本オプションは最適化リンケージエディタ Ver.8 から対応しています。

9.3.5 参照されない定義シンボルの通知

■説明

大規模なプロジェクトの場合、定義がされていてもどこからも参照されない、外部定義シンボルがあっても発見が容易ではありません。

本オプションを指定すると、リンク時にメッセージ出力によって、参照されないシンボルを調査することができます。

なお、同時に message オプション（最適化リンカタブ カテゴリ:[オプション項目][出力ファイル/インフォメーション抑止] インフォメーションレベルメッセージ抑止）を指定しないと、本機能は動作しません。

■指定方法

ダイアログメニュー：最適化リンカタブカテゴリ: [出力] [オプション項目][メッセージ出力指定] 参照されない定義シンボルの通知

コマンドライン　　：*msg_unused*

■出力メッセージ

```
L0400 (I) Unused symbol "ファイル"-"シンボル"
```

"ファイル"内の"シンボル"は使用されていません。

■備考

(1) 本オプションは最適化リンケージエディタVer.9から対応しています。

(2) 以下の場合、参照関係の解析が正しく行うことができず、メッセージ出力により通知される情報が不正確になります。

・アセンブル時に goptimize オプション指定されておらず、同一ファイル内、かつ同一セクションへの分岐がある場合

・同一ファイル内の定数シンボルへの参照

・コンパイル時に最適化が有効で、直下の関数を呼び出す場合

・リンク時の最適化によって、定数の統合が生じる場合

9.3.6 セクション内データの詰め込み配置

■説明

本オプションを指定すると、オブジェクトファイル単位のセクションの境界調整により生じる空き領域を詰めてリンクを行います。

これにより、境界調整で生じる冗長な空き領域を詰めることができ、データサイズを削減することができます。

対象となるのは定数領域（C セクション）、初期化データ領域（D セクション）、未初期化データ領域（B セクション）です。

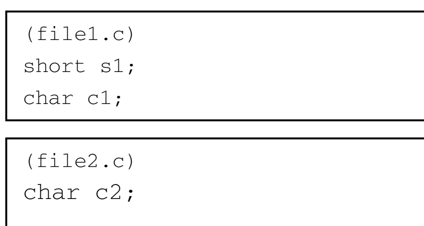
■指定方法

ダイアログメニュー：最適化リンカタブカテゴリ:[出力][オプション項目] セクション内データの詰め込み配置

コマンドライン　　：*data_stuff*

■指定例

下記プログラムを例に詰め込み配置を説明します。



```
(file1.c)
short s1;
char c1;
```

```
(file2.c)
char c2;
```

【data_stuff 指定なしのデータ配置】

data_stuff 指定がない場合、SH は CPU の仕様上、境界調整数が４のため、file1.c と file2.c の間に、境界調整用の空き領域１バイトを挿入し、境界調整を行います。

本プログラム例のように、後ろにリンクされる先頭のデータが 1 バイトであれば、この調整は必要ありません。

しかし、次のファイルの先頭データが 2 バイト以上の場合は、当該ファイル（file1.c）の末尾で境界調整をする必要があります。

その結果データ並びが、s1(2 バイト)＋c1(1 バイト)＋空き領域(1 バイト)＋c2(1 バイト)になり、データサイズの合計は５バイトになります。

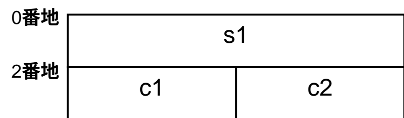
0番地	s1	
2番地	c1	空き領域
4番地	c2	

【data_stuff 指定ありのデータ配置】

data_stuff 指定がある場合は、本プログラム例のように後ろにリンクされる先頭のデータが、1 バイトであれば空き領域を作成せずにデータ領域を作成します。

その結果データ並びが、s1(2 バイト)＋c1(1 バイト)＋c2(1 バイト)になり、データサイズの合計は 4 バイトに削減されます。

なお、本プログラム例のような詰め込み配置を行いますが、データの順序を変えることは行いません。



■備考

（1）　本オプションは最適化リンケージエディタV.8.00.06から対応しています。

（2）　アセンブラ出力のオブジェクトファイルに対しては、本オプション機能は適用されません。

（3）　下記のいずれかの場合、本オプション指定は無効です。

- 最適化リンケージエディタの出力ファイル形式にライブラリファイル、オブジェクトファイルを指定した場合
- 最適化リンケージエディタの入力ファイル形式にアブソリュートファイルを指定した場合
- memory=low 指定時
- リンク時の最適化(optimzie)を指定

（4） 本オプションを指定して生成したリロケータブルファイルに対しては、リンク時の最適化が適用されません。

9.4 最適化機能

9.4.1 リンク時の最適化とは？

■説明

コンパイラがオブジェクトファイルを作成するとき、各モジュールに付加情報を出力し、この付加情報を基にコンパイル時には不可能なモジュール間に渡る最適化を行い、リンクします。

その結果、ROM サイズ／実行速度ともに向上します。

■指定方法

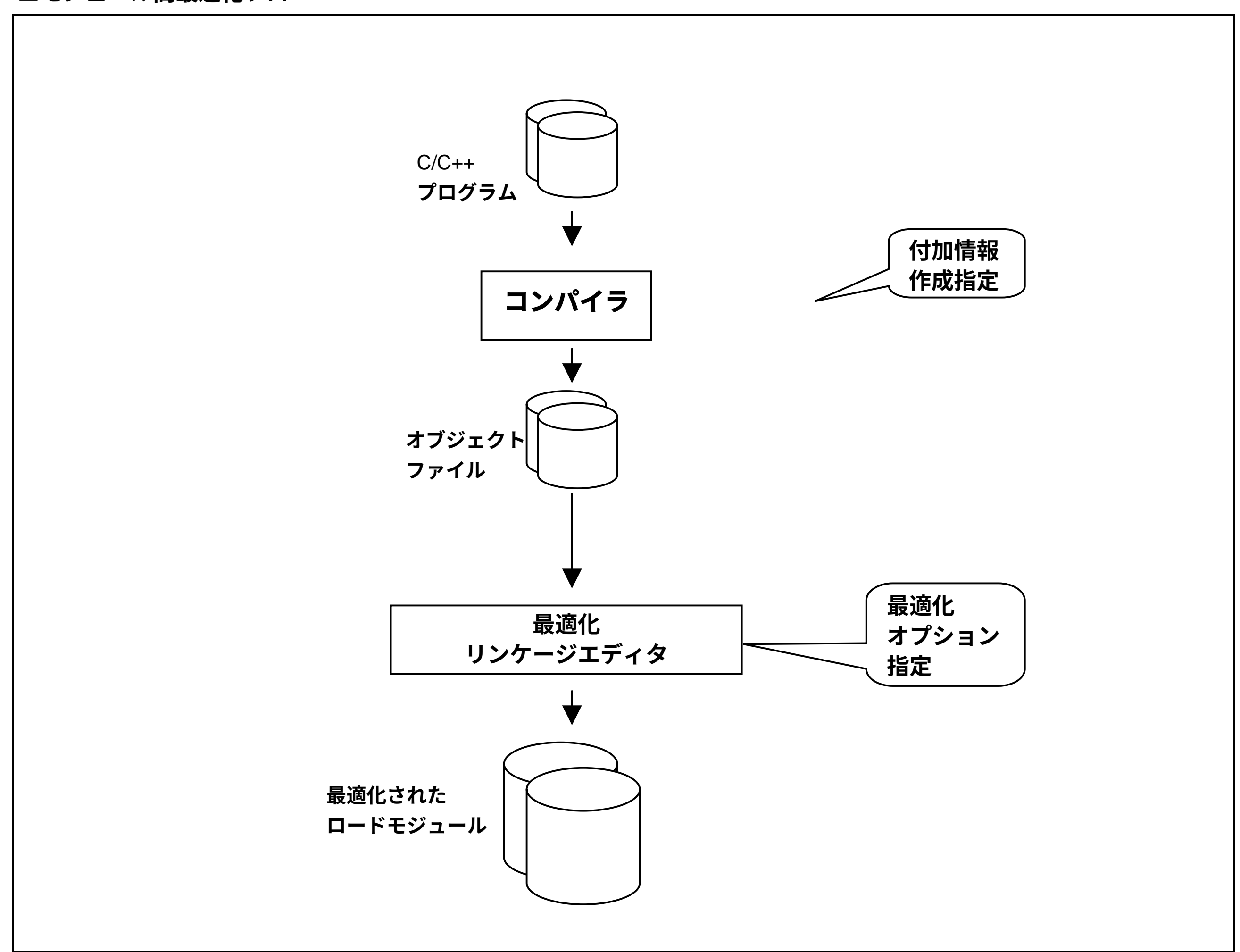
ダイアログメニュー	：	最適化リンカタブカテゴリ:[**最適化**]
コマンドライン	：	*optimize=<サブオプション>*
	：	<サブオプション>は「9.4.2　**定数／文字列の統合**」～「9.4.6　**分岐命令の最適化**」の各最適化項目参照

また、リンク時の最適化指定をしても、コンパイル時に以下の付加情報作成指定が必須です。指定をしないとリンク時の最適化は動作しません。

■付加情報作成指定

ダイアログメニュー	：	コンパイラタブカテゴリ:[**最適化**] モジュール間最適化
コマンドライン	：	*goptimize*

■モジュール間最適化フロー



RENESAS

9.4.2 定数／文字列の統合

サイズ効率	○	処理速度	-

■説明

const 属性を持つ定数／文字列に対し、同一値定数および同一文字列の統合をモジュール間にわたって実施します。
本サブオプションはコンストセクションを削除するので、サイズ効率は上がりますが、処理速度に変化はありません。

■指定方法

ダイアログメニュー　：最適化リンカタブカテゴリ:[**最適化**]: 設定: 定数／文字列の統合
コマンドライン　　　：*optimize=string_unify*

■同一値定数の最適化例

同一な定数値を持つ、const long の変数「cl1,cl2」を 1 つに統合します。
それにより ROM サイズが 4 バイト小さくなります。

```
(file1.c)
#include <machine.h>
const long cl1=100;
void main(void);
void func01(long);
long g_max;
void main(void)
{
        func01(cl1+1);
        func02(cl1+2);
        func03(cl1+3);
}
void func01(long c_litr)
{
        g_max = c_litr++;
}
```

```
(file2.c)
#include <machine.h>
const long cl2=100;
void main(void);
void func02(long);
void func03(long);
extern long g_max;

void func02(long c_litr)
{
        func03(cl2+c_litr);
        nop();
}
void func03(long c_litr)
{
        g_max = c_litr;
}
```

削除

RENESAS

9.4.3 未参照シンボルの削除

サイズ効率	○	処理速度	-

■説明

一度も参照のない変数／関数を削除します。この最適化を指定する場合は、必ずエントリ関数の指定をしてください。エントリ関数の指定がないと、本最適化は実行されません。

エントリ関数およびエントリ関数以前のアドレスの関数を最適化すると、リセット時にベクタテーブルからエントリ関数にジャンプしますが、最適化を行うとジャンプするアドレスが狂ってしまうためです。(指定方法は下記参照)

■指定方法

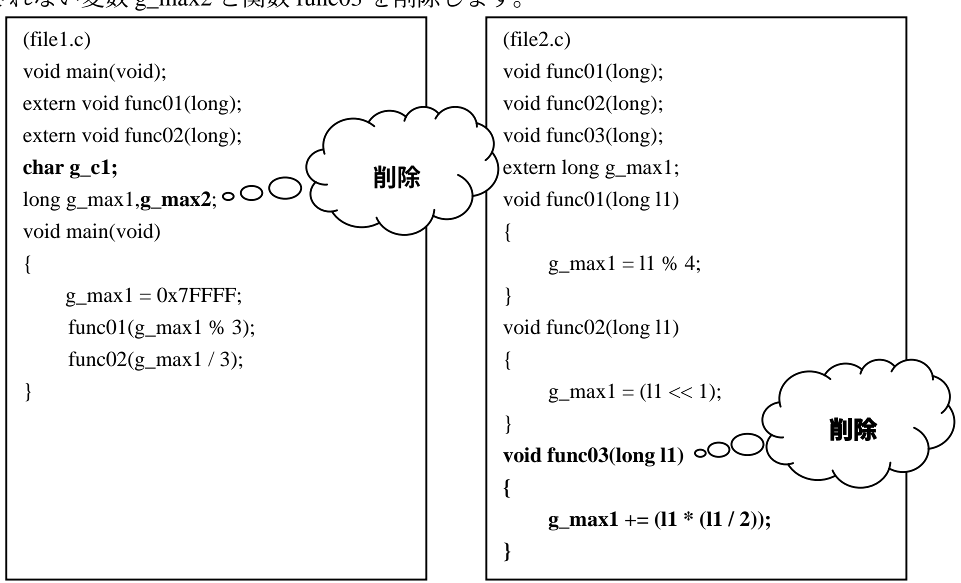
ダイアログメニュー　：最適化リンカタブカテゴリ:[**最適化**]: 設定: 未参照シンボルの削除
コマンドライン　　　：*optimize=symbol_delete*

■エントリ関数指定方法

ダイアログメニュー　：最適化リンカタブカテゴリ:[**入力**]: エントリポイント
コマンドライン　　　：*entry=<シンボル名> | <アドレス>*
*シンボル名は‘_’付きで記述します。（例：main->_main）

■未参照シンボルの削除例

一度も参照されない変数 g_max2 と関数 func03 を削除します。

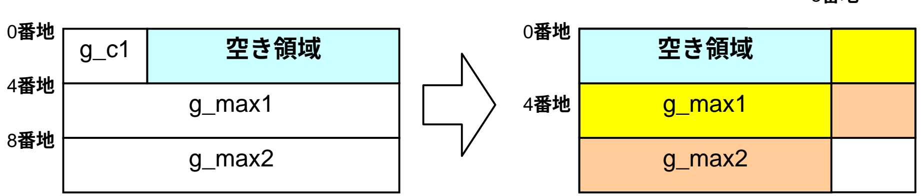


char 型の変数 g_c1 も一度も参照されませんが、削除されません。

これは以下のように、SH の境界調整数が 4 のため、削除してしまうと後続の変数のアドレスが 4 の倍数でなくなってしまうためです。

CPU の仕様上、シンボルのアドレスが奇数番地になると参照したときにアドレスエラーになってしまいます。

【もし、１バイトの変数を削除した場合】



もし、最適化をしてしまうと 3 番地から g_max1(4 バイト変数)をアクセスしてしまいます。

RENESAS

9.4.4 レジスタ退避・回復の最適化

サイズ効率	○	処理速度	○

■説明

関数の呼び出し関係を解析し、冗長なレジスタ退避･回復コードを削除します。また、呼び出し前後のレジスタの使用状況により、使用レジスタ番号を変更することもあります。

■指定方法

ダイアログメニュー　：最適化リンカタブカテゴリ:[**最適化**]: 設定: レジスタ退避・回復の最適化
コマンドライン　　　：optimize=register

■レジスタ退避・回復の最適化例

func1 関数から func2 関数、func3 関数を呼び出しています。

```
(file1.c)
void func1(int i1,int i2,int i3,int i4,int i5,long *l6)
{
    a = 0 * i1;
    b = 1 * i2;
    d = 4 * i4;
    h = 8 * i5;
    i = 9
    *l6 = b;
    func2(i,h,3000,200,100,l6);
    func3(i,h,3000,200,100,l6);
}
```

```
(file2.c)
void func2(int i1,int i2,int i3,int i4,int i5,long *l6)
{
    a = 0 * i1;
    b *= 1 * i2;
    d *= 4 * i4;
    f *= 6 / i2;
    g = 7 / i3;
    h = 8 * i5;
    i *= 9 / b ;
    *l6 = b * g;
}
```

```
(file3.c)
void func3(int i1,int i2,int i3,int i4,int i5,long *l6)
{
    a = 0 * i1;
    b *= 1 * i2;
    c = 2 * i3;
    d *= 4 * i4;
    e *= 5 * i1;
    f *= 6 / i2;
    g *= i2 / i3;
    h *= 8 * i5;
    i *= (9 / b) * ((*l6)++);
    *l6 *= b * g;
}
```

■レジスタ退避・回復の最適化のコード例

レジスタ退避・回復の最適化前後のコード変化は下記のようになります。
親関数で退避・回復レジスタを増やすことにより、子関数での退避・回復レジスタを減らしています。
本例の場合、ROM サイズが 532→524 バイト、実行速度が 718→711 サイクルになります。
(SH-1 の場合)

(最適化前)

R13-R14を退避・回復(2本)

```
_func1:
        MOV.L    R13,@-R15
        MOV.L    R14,@-R15
        :(途中省略)
        MOV.L    @R15+,R14
        RTS
        MOV.L    @R15+,R13
```

R10,R11, R12, R14を退避・回復(4本)

```
_func2:
        MOV.L    R10,@-R15
        MOV.L    R11,@-R15
        MOV.L    R12,@-R15
        MOV.L    R14,@-R15
        (途中省略)
        MOV.L    @R15+,R14
        MOV.L    @R15+,R12
        MOV.L    @R15+,R11
        RTS
        MOV.L    @R15+,R10
```

R8-R14を退避・回復(7本)

```
_func3:
        MOV.L    R8,@-R15
        MOV.L    R9,@-R15
        MOV.L    R10,@-R15
        MOV.L    R11,@-R15
        MOV.L    R12,@-R15
        MOV.L    R13,@-R15
        MOV.L    R14,@-R15
         (途中省略)
        MOV.L    @R15+,R14
        MOV.L    @R15+,R13
        MOV.L    @R15+,R12
        MOV.L    @R15+,R11
        MOV.L    @R15+,R10
        MOV.L    @R15+,R9
        RTS
        MOV.L    @R15+,R8
```

(最適化後)

ER2-ER4を退避・回復(7本)

```
_func1:
        MOV.L    R8,@-R15
        MOV.L    R9,@-R15
        MOV.L    R10,@-R15
        MOV.L    R11,@-R15
        MOV.L    R12,@-R15
        MOV.L    R13,@-R15
        MOV.L    R14,@-R15
         (途中省略)
        MOV.L    @R15+,R14
        MOV.L    @R15+,R13
        MOV.L    @R15+,R12
        MOV.L    @R15+,R11
        MOV.L    @R15+,R10
        MOV.L    @R15+,R9
        RTS
        MOV.L    @R15+,R8
```

退避・回復なし(0本)

```
_func2:
        STS.L    PR,@-R15
         (途中省略)
        LDS.L    @R15+,PR
        NOP
        RTS
        NOP
```

R13,R14退避・回復(2本)

```
_func3:
        MOV.L    R13,@-R15
        MOV.L    R14,@-R15
         (途中省略)
        MOV.L    @R15+,R14
        RTS
        MOV.L    @R15+,R13
```

9.4.5 共通コードの統合

サイズ効率	○	処理速度	-

■説明

モジュール間における、複数の同一命令列をサブルーチン化して、コードサイズを削減します。
よって、関数呼び出しオーバヘッドが増えて、処理速度が低下するので注意が必要です。
また、統合するときの最低サイズを決めることができます。
なお、処理速度が低下するのでコンパイル時に関数のインライン展開指定をすると、本最適化は実施されません。

■指定方法

ダイアログメニュー　：最適化リンカタブカテゴリ:[**最適化**]: 共通コードの統合
コマンドライン　　　：*optimize=same_code*

■統合サイズ指定方法

ダイアログメニュー　：最適化リンカタブカテゴリ:[**最適化**]: 統合サイズ
コマンドライン　　　：*samesize=<サイズ>*

■共通コードの統合 C ソース例

func00 関数と func01 関数に同様な式の並びがあります。

```
(file1.c)
void main(void);
int func00(int,int,int);
extern int func01(int,int,int);
int ret;
void main(void)
{
        ret =    func00(10,11,12);
        ret += func01(20,21,22);
}
int func00(int i1,int i2,int i3)
{
        i1++;
        i2++;
        i3++;
        i1 = i3 & i2;
        i2 = i1 & i3;
        i3 = i2 & i3;
        return i1+i2+i3;
}
```

```
(file2.c)
void func01(void);
int func01(int,int,int);
int func01(int i1,int i2,int i3)
{
        i1++;
        i2++;
        i3++;
        i1 = i3 & i2;
        i2 = i1 & i3;
        i3 = i2 & i3;
        return i1+i2+i3;
}
```

■共通コードの統合 コード例

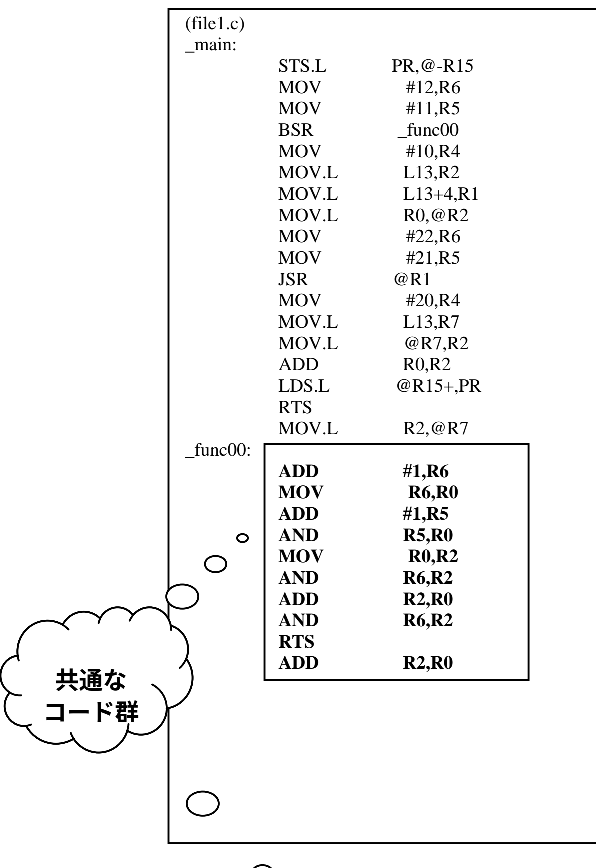
共通コードの統合前後のコード変化は下記のようになります。

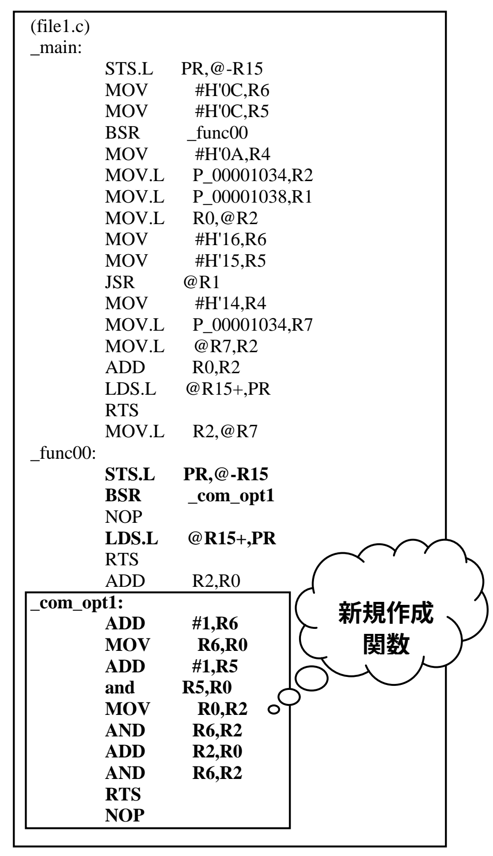
共通なコード群を新規作成関数(_com_opt1)に集約し、元の場所から新規作成関数を呼び出すようになります。

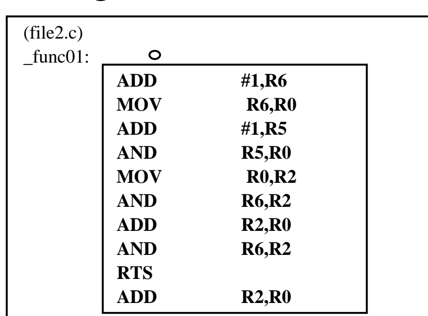
本例の場合、ROM サイズが 40→24 バイト、実行速度が 46→60 サイクルになります。

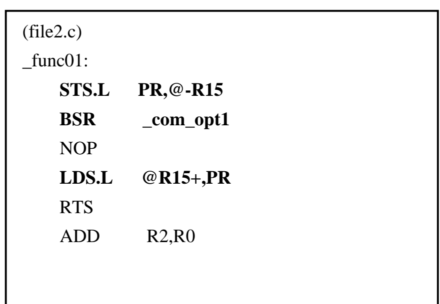
(SH-1 の場合)

(最適化前) **(最適化後)**









9.4.6 分岐命令の最適化

サイズ効率	○	処理速度	○

■説明

C/C++コンパイラは、別ファイルにある関数をアクセスする場合や、PC 相対の関数呼び出し命令(BSR 命令)でアクセスできる範囲*を超えている場合は、絶対アドレスで関数を呼び出す命令(JSR 命令)で関数を呼び出します。

最適化リンケージエディタは、リンク時に最適化を行うので、分岐先が別ファイルの関数でも、リンク後に分岐幅を再計算することができます。

そのとき、可能であれば、PC 相対で関数を呼び出す BSR 命令に変換します。

また、当初の分岐幅が PC 相対で関数をアクセスできる範囲を超えていても、他の最適化により分岐幅が縮まった場合、同様に BSR 命令に変換します。

なお、他の最適化項目を 1 つでも実行すると、本最適化は指定の有無にかかわらず必ず実行されます。

【注】 * PC **相対で関数をアクセスできる範囲：**－4096～4094 **バイト**

■指定方法

ダイアログメニュー　：最適化リンカタブカテゴリ:[**最適化**]: 分岐命令の最適化
コマンドライン　　　：*optimize=branch*

■分岐命令の最適化ソース例

main 関数から func01 関数を呼び出す例です。

```
(file1.c)
long func01(long,long);
void main(void);
long g_l1,g_l2;
void main(void)
{
      g_l1 = 100;
      g_l2 = 200;
      g_l1 = func01(g_l1,g_l2);
}
            :
      <途中省略>
            :
long func01(long l1,long l2)
{
      return l1 + l2;
}
```

■分岐命令の最適化 コード例

分岐命令の最適化前後のコード変化は下記のようになります。
func01 を BSR で呼び出すようになります。
本例の場合、ROM サイズが 46→42 バイト、実行速度が 22→21 サイクルになります。
(SH-1 の場合)

(最適化前)

```
_main:
        STS.L     PR,@-R15
        MOV.L     L13,R1
        MOV       #-56,R5
        MOV.L     L13+4,R2
        MOV       #100,R4
        EXTU.B    R5,R5
        MOV.L     R4,@R1
        MOV.L     L13+8,R3
        JSR       @R3
        MOV.L     R5,@R2
        MOV.L     L13,R7
        LDS.L     @R15+,PR
        RTS
        MOV.L     R0,@R7
L13:
        .DATA.L   _g_l1
        .DATA.L   _g_l2
        .DATA.L   _func01
```

```
_func01:
        ADD       R5,R4
        RTS
        MOV       R4,R0
```

(最適化後)

```
_main:
        STS.L     PR,@-R15
        MOV.L     L13,R1
        MOV       #--56,R5
        MOV.L     L13+4,R2
        MOV       #100,R4
        EXTU.B    R5,R5
        MOV.L     R4,@R1
        NOP
        BSR       _func01
        MOV.L     R5,@R2
        MOV.L     L13,R7
        LDS.L     @R15+,PR
        RTS
        MOV.L     R0,@R7
L13:
        .DATA.L   _g_l1
        .DATA.L   _g_l2
```

```
_func01:
        ADD       R5,R4
        RTS
        MOV       R4,R0
```


RENESAS

9.4.7 最適化部分抑止

■説明

最適化リンケージエディタによって、最適化されたくない関数／変数シンボルを下記により指定することができます。シンボル名による抑止と、アドレス範囲による抑止方法があります。

■未参照シンボル削除の抑止

■指定方法

ダイアログメニュー　：最適化リンカタブカテゴリ:[**最適化**] 最適化方法 最適化部分抑止
　未参照シンボル削除抑止シンボル

コマンドライン　：*symbol_forbid=<シンボル名>*

■共通コード統合の抑止

■指定方法

ダイアログメニュー　：最適化リンカタブカテゴリ:[**最適化**] 最適化方法 最適化部分抑止
　共通コード統合抑止シンボル

コマンドライン　：*samecode_forbid=<関数名>*

■最適化抑止アドレス範囲

■指定方法

ダイアログメニュー　：最適化リンカタブカテゴリ:[**最適化**] 最適化方法 最適化部分抑止
　最適化抑止アドレス範囲

コマンドライン　：*absolute_forbid=<アドレス> [+サイズ]*

9.4.8 最適化結果の確認

■説明

リンケージエディタで最適化をした結果を以下の方法により確認できます。

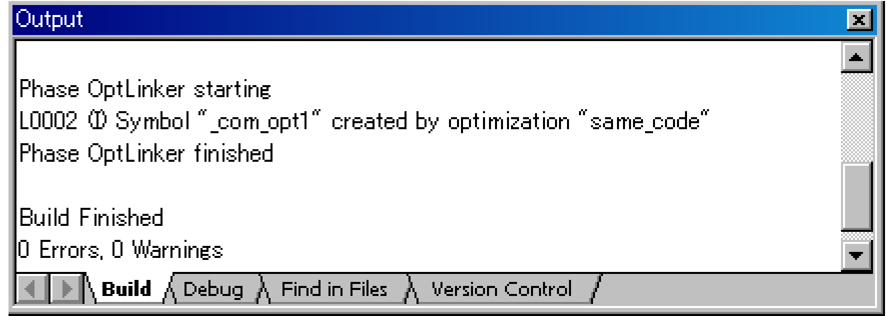
■メッセージでの確認

HEW の場合以下のチェックを外すことにより、最適化結果を出力できます。

ダイアログメニュー	：最適化リンカタブカテゴリ:[**出力**] インフォメーションレベルメッセージ抑止
コマンドライン	：*message[=<エラー番号>]* ：*nomessage*

■メッセージ出力例

共通コードの統合により新規関数が作成された例です。



■リストでの確認

以下を指定することにより、確認することができます。詳細は「9.2.1 **シンボル情報表示**」を参照してください。

ダイアログメニュー	：最適化リンカタブカテゴリ:[**リスト**] リスト内容 シンボル情報
コマンドライン	：*list [=<ファイル名>]* *show symbol*

10. MISRA C

10.1 MISRA C

10.1.1 MISRA C とは

MISRA C とは Motor Industry Software Reliability Association（MISRA）が 1998 年に発行したＣ言語の使用ガイドライン「Guideline for the use of the C language in vehicle based software」、もしくはそのガイドラインで規格化されたＣ言語記述のルールです。C 言語は優れた言語ですが、いくつかの問題を持っています。MISRA　C ガイドラインでは C 言語には 5 種類の問題があるとしています。プログラマによるエラー、言語に対する誤解、意図しないコンパイラの動作、実行時のエラー、コンパイラ自体のエラーです。MISRA C の目的はこれらの問題を回避し C 言語の安全な使用を促進することです。MISRA C には 127 のルールがあり全ルールに合致するようコードの開発を行います。ルールは必要項目と推奨項目の 2 種類に分けられています。すべてのルールを守るのは現実的に難しい場合もあるので、しかたのないルール違反に対してはそれを文書化し認める手順もあります。またルール以外にもソフトウェアメトリックスを計測しなければいけないなど諸問題への対応が求められます。

10.1.2 ルールの例

実際に MISRA C のルールをいくつか紹介します。**図 10.1** はルール 62「switch 文はすべて最後に default 節を置かなければならない」です。これはプログラマによるエラーに分類される問題です。switch 文で”default”ラベルを”defalt”とタイプミスしてもコンパイラはエラーにしません。プログラマ自身が気づかなければデフォルト時に期待する動作は永久に実行されません。ルール 62 を適用することでこの問題を回避できます。

例）
```
switch(x) {
    :
defalt:                 ← スペルミス
    err = 1;
    break;
}
```

図 10.1　ルール 62

図 10.2 はルール 46「式の値は規格が定めるどのような順序で評価されようとも同じでなければならない」です。これは言語に対する誤解に分類される問題です。++i が先に評価されると 2+2 となり i が先に評価されると 2+1 になります。同様に関数の引数の評価順序も未規定なので++ j が先に評価されると f(2,2)となり j が先に評価されると f(1,2)になります。ルール 46 を適用することでこの問題を回避できます。

例）
```
 i = 1;
x = ++i + i;        x = 2 + 2?   x = 2 + 1?

j = 1;
func(j, ++j);       func(1, 2)? func(2, 2)?
```

図 10.2　ルール 46

図 10.3 はルール 38「シフト演算子の右辺の項はゼロ以上、左辺の項のビット幅未満でなければならない」です。これは意図しないコンパイラの動作に分類される問題です。ANSI ではビットシフト演算子のシフト数が負の値の場合およびシフトされるオブジェクトのサイズ以上の場合演算結果は未定義としています。**図 10.3** では us をシフトする場合のシフト数は 0 以上 15 以下でなければ結果は未定義となりコンパイラによってその値は異なります。ルール 38 を適用することでこの問題を回避することができます。

図 10.3　**ルール** 38

図 10.4 はルール 51「符号なし整数定数式の評価は結果の型にはまるべきである」です。これは実行時のエラーに分類される問題です。符号なし整数の演算の結果が論理的に負になる場合は論理的な負の値を期待しているのか符号なしとして演算した結果でよいのかが不明確になり不具合の原因となる恐れがあります。また足し算の結果がオーバフローを起こして小さな値になることもあります。この問題はルール 51 を適用することで回避可能です。

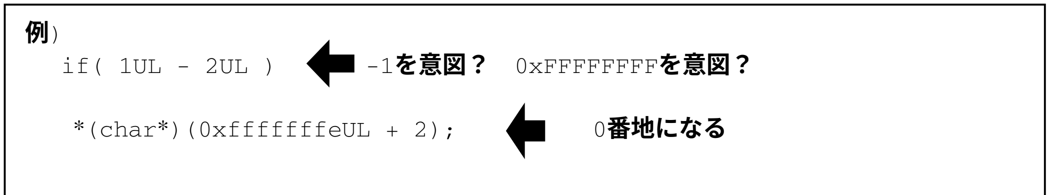


図 10.4　**ルール** 51

10.1.3　合致マトリクス

MISRA C では 127 あるルールのすべてについてソースコードを確認することになっています。さらにそれぞれのルールが守られているかを一目で確認できるようにするために合致マトリクスと呼ばれる表を作成することが要求されています（**表** 10.1）。すべてのルールを目視でチェックするのは大変なので静的チェックツールの使用が推奨されています。MISRA C ガイドラインでも「ルールを遵守するためにツールの使用が非常に重要、ツールの使用があたりまえになることが望ましい。」となっています。ツールではチェックできないルールもあるのでそれらについては目視でレビューを行う必要があります。

表 10.1　**合致マトリクス**

ルール番号	コンパイラ	ツール 1	ツール 2	レビュー（目視）
1	警告　347			
2		違反 38		
3			警告 97	
4				合格
…	…	…	…	…

10.1.4　ルール違反

ルール違反の中には安全であることがわかっているものもあり、その方が効果的なものもあります。そのようなルール違反は認められるべきですが安易にルール違反を認めるのも安全性を損ないます。そこで MISRA C ではルール違反を認める手順を定めています。ルール違反に対する正当な理由付けがあること、そのルール違反が安全であると証明できることが必要です。認める違反のそれぞれすべてについてその場所と正当な理由を文書化します。安易に違反を認められないように、専門家のアドバイスの下、それらの文書に組織で権威を持つ人の署名を入れます。一度認められたルール違反と同じパターンのルール違反を「認められたルール違反」と呼び上記の手続きなしで認められたものとして扱ってよいことになっています。しかしそれも定期的に見直す必要があるとしています。

10.1.5　MISRA C 準拠

MISRA C に準拠していることを主張するにはルールに合致したコードを開発することとルール以外の諸問題への対策が必要になります。コードがルールに合致していることを示すには合致マトリクス、認めるルール違反に関する文書、各ルール違反への署名が必要になります。諸問題への対策とはC言語や使用するツールを使いこなすための技術的訓練を行ったり、コーディングスタイルを規定したり、ツールの選定時に妥当性を確認したり、各種ソフトウェアメトリックスを計測するなどの対応を求められます。またこれらの取り組みが正式に標準化されている必要があります。標準化とは文書化と実践の両方が行われていることを指します。MISRA C の準拠はガイドラインに従って開発された個々の製品に対してしか言えず、組織に対して言うことはできません。

10.2　SQMlint

10.2.1　SQMlint とは

SQMlint はルネサス製 C コンパイラに、MISRA C ルールに違反していないかを検査する機能を付加する機能追加パッケージです。C ソースコードを静的に検査してルールに違反している個所をレポートします。SQMlint はルネサス製品開発環境の中でCコンパイラの一部として動作します。（**図** 10.5）コンパイル時にオプションを追加するだけで SQMlint が起動します。コンパイラが生成するコードに影響をあたえることはありません。SQMlint が対応しているルールは**表** 10.2 のようになっています。

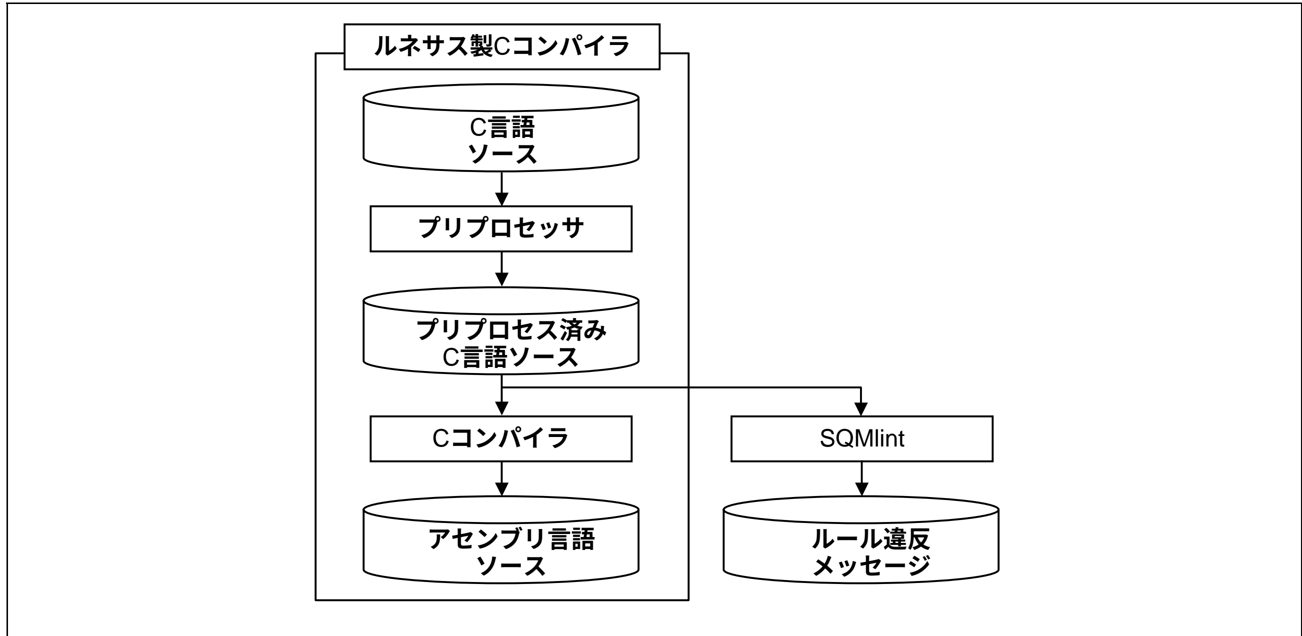


図 10.5　SQMlint **の位置付け**

表 10.2 SQMlint 対応ルール

ルール	可否	ルール	可否	ルール	可否	ルール	可否	ルール	可否	ルール	可否
1	○	26	×	51	○*	76	○	101	○	126	○
2	×	27	×	52	×	77	○	102	○	127	○
3	×	28	○	53	○	78	○	103	○		
4	×	29	○	54	○*	79	○	104	○		
5	○	30	×	55	○	80	○	105	○		
6	×	31	○	56	○	81	×	106	○*		
7	×	32	○	57	○	82	○	107	×		
8	○	33	○	58	○	83	○	108	○		
9	×	34	○	59	○	84	○	109	×		
10	×	35	○	60	○	85	○	110	○		
11	×	36	○	61	○	86	×	111	○		
12	○	37	○	62	○	87	×	112	○		
13	○	38	○	63	○	88	×	113	○		
14	○	39	○	64	○	89	×	114	×		
15	×	40	○	65	○	90	×	115	○		
16	×	41	×	66	×	91	×	116	×		
17	○*	42	○	67	×	92	×	117	×		
18	○	43	○	68	○	93	×	118	○		
19	○	44	○	69	○	94	×	119	○		
20	○	45	○	70	○*	95	×	120	×		
21	○*	46	○*	71	○	96	×	121	○		
22	○*	47	×	72	○*	97	×	122	○		
23	×	48	○	73	○	98	×	123	○		
24	○	49	○	74	○	99	○	124	○		
25	×	50	○	75	○	100	×	125	○*		

○：検査可　×：検査対象外　○∗：制限付で検査可

表 10.3 SQMlint 対応ルール数

ルール分類	検査可能なルールの数 （SQMlint 対応ルール数/全ルール数）
必要ルール	67/93
推奨ルール	19/34
合計	86/127

10.2.2　使用方法

HEW のコンパイルオプション設定画面でも簡単に SQMlint の起動を設定できます。**図** 10.6 は HEW のオプション指定ダイアログです。カテゴリから「MISRA C ルール検査」を選びます。

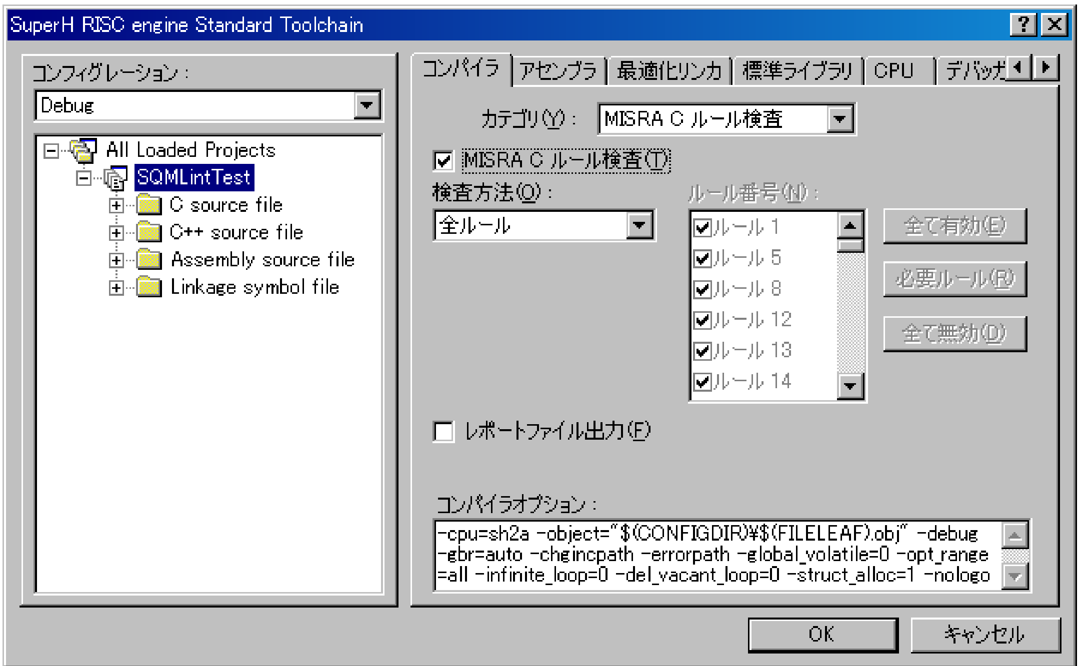


図 10.6　HEW のオプション選択画面

これを選択することでコンパイル時に SQMlint が起動されるようになります。本ダイアログにおける「検査方法」の意味は次のようになります。

- 全ルール　・・・すべてのルールを対象に検査します。
- 必要ルール・・・MISRA C ルール中で、“必須”と書かれているルールのみを検査します。
- カスタム　・・・ユーザが指定する MISRA C ルールを対象に検査します。ルール番号、および右側のボタンを用いて、対象となる MISRA C ルールを選択してください。

10.2.3　検査結果の確認方法

検査結果の出力は次の 3 つの形式があります。

(a)　標準エラー出力

HEW上でコンパイルエラーと同様にメッセージが出力されます。メッセージをダブルクリックするか右クリックをしてジャンプを選ぶとタグジャンプも可能です。コンパイルエラーと同じ操作で手軽にソースコードを修正できます。またメッセージを右クリックしてヘルプを選ぶとそのメッセージの説明が表示されます。

(b)　CSV形式のファイル

表計算ソフトで読み込み可能な形式のファイルです。そのためレビューなどに使用しやすくなっています。

(c)　SQMmerger

SQMmergerはSQMlintで生成したレポートファイル(CSV形式)とCソースファイルから、Cソース行と対応するレポートメッセージの混合表示ファイルを作成するツールです。SQMmergerの入力書式は次のようになります。

sqmmerger –src C ソースファイル名　–r レポートファイル名　–o 出力ファイル名

図10.7のようなソースファイルと検査結果の混合表示を行います。

```
1 : void func(void);
2 : void func(void)
3 : {
4 : LABEL:
 [MISRA(55) Complain] label ('LABEL') should not be used
5 :
6 : goto LABEL;
 [MISRA(56) Complain] the 'goto' statement shall not be used
7 : }
```

図 10.7　SQMmerger

10.2.4　開発手順

SQMlint を使用した開発手順を**図** 10.8 に示します。

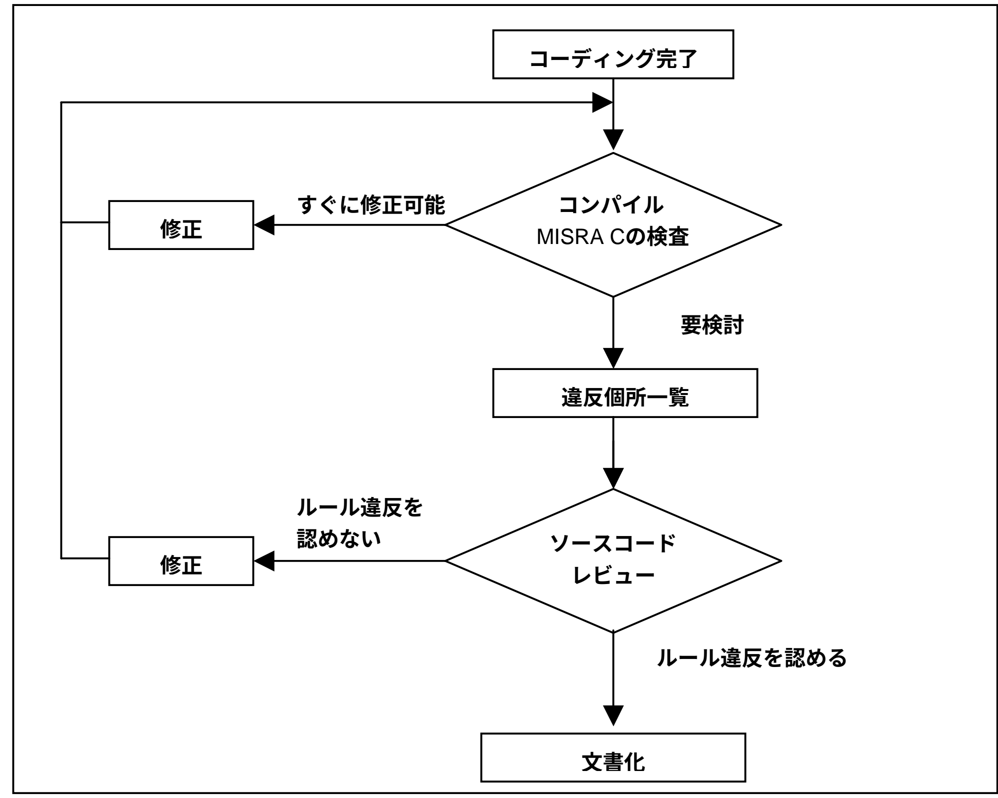


図 10.8　SQMlint **を用いた開発手順**

(1)　コンパイルエラーをすべて取ります。SQMlintは正しいCソースコードを前提としています。
(2)　SQMlintで検出したエラーを調べます。
(3)　簡単に修正できるものはすぐに修正します。
(4)　検討が必要なものはルール違反個所一覧を作成しレビューを行います。
(5)　レビューの結果ルール違反を認めないものは修正を行います。
(6)　レビューの結果ルール違反を認めるものはそれを文書化し記録として残します。

10.2.5　対応コンパイラ

以下のコンパイラが SQMlint に対応しています。

- SHC/C++コンパイラパッケージ V.9.00 Release00 以上（Windows 版）

10.2.6 SH C/C++コンパイラでチェック可能なルール

以下のルールは SQMlint では検査できませんが、SH C/C++のコンパイルメッセージによりルール違反を検出することができます。

表 10.4 コンパイラでチェック可能なルール

ルール番号	ルール内容	SHC/C++のコンパイルメッセージ
9	コメントは入れ子であってはならない。	C5009 (I) Nested comment is not allowed /* */コメントがネストしています。
26	オブジェクトや関数が２度以上宣言された場合、互換性のある宣言を持たなければならない。	C2136 (E) Type mismatch extern あるいは static 記憶クラスを持つ変数あるいは関数を 2 度以上宣言しており、その型が一致していません。
52	到達しないコードがあってはならない	C0003 (I) Unreachable statement 実行されることのない文があります。

11.　Q & A

本章では、ユーザから多く寄せられた質問についての回答を記載します。

11.1　C/C++コンパイラ、アセンブラ

11.1.1　const 宣言

■質問

const 宣言を行いましたが、定数領域（C）セクションに割り付けられません。

■回答

シンボルを const 宣言すると、次に示す意味になるので注意してください。

（1）　const char msg[]="sun";
　　　Cセクションへ割り付け　：文字列"sun"

（2）　const char *msg[]={"sun", "moon"};
　　　Cセクションへ割り付け　：文字列"sun"と"moon"
　　　Dセクションへ割り付け　：msg[0]とmsg[1]
　　　(*msg[0]と*msg[1]の先頭アドレス)

（3）　const char *const msg[]={"sun", "moon"};
　　　Cセクションへ割り付け　：文字列"sun"と"moon"、msg[0]とmsg[1]
　　　(*msg[0]と*msg[1]の先頭アドレス)

（4）　char *const msg[]={"sun", "moon"};
　　　Cセクションへ割り付け　：文字列"sun"と"moon"、msg[0]とmsg[1]
　　　(*msg[0]と*msg[1]の先頭アドレス)

11.1.2 1 ビットデータの正しい判定方法

■質問

ビットフィールドでサイズが 1 ビットのデータに対し、セットされているか、いないかを判定しようとしました。ところが、正しく判定できない場合がありました。

■回答

1 ビットデータを、符号付き（signed）で宣言した場合、その 1 ビットデータを符号ビットとして解釈します。
したがって、1 ビットデータで表現できる値は、"0"と"-1"になります。
"0"と"1"を表現するためには、必ず符号なし（unsigned）で宣言してください。

例）

判定が常に偽となる例

```
struct{
       char  p7:1;
       char  p6:1;
       char  p5:1;
       char  p4:1;
       char  p3:1;
       char  p2:1;
       char  p1:1;
       char  p0:1;
}s1;

if(s1.p0 == 1){
   s1.p1 = 0;
}
```

正しく判定される例

```
struct{
       unsigned char  p7:1;
       unsigned char  p6:1;
       unsigned char  p5:1;
       unsigned char  p4:1;
       unsigned char  p3:1;
       unsigned char  p2:1;
       unsigned char  p1:1;
       unsigned char  p0:1;
}s1;

if(s1.p0 == 1){
  s1.p1 = 0;
}
```

【注】 if 文の条件式を 0 との比較にした方が生成されるコード効率が良くなります。

11.1.3 インストール

■質問

コンパイラ、アセンブラ、リンカのコマンドを投入したが起動できない。

■回答

環境変数"PATH"の指定にコンパイラ、アセンブラ、リンカをインストールしたディレクトリが含まれているか確認してください。

DOS 窓から起動させるためには、以下の環境設定を行います。

（1） PATHの設定

各ツールのある場所をPATHに設定します。

例）

c:¥> PATH=%PATH%;C:¥Hew3¥Tools¥Renesas¥Sh¥9_0_0¥bin (RET)

すでに設定されているPATHに追加します。

（2） SHC_LIBの設定

SuperH RISC engine C/C++コンパイラ本体の格納場所を示します。

この設定を省略することはできません。

例）

c:¥> set SHC_LIB=C:¥Hew3¥Tools¥Renesas¥Sh¥9_0_0¥bin (RET)

（3） SHC_TMPの設定

SuperH RISC engine C/C++コンパイラが作業用のテンポラリファイルを作成するパスを指定します。この設定を省略することはできません。

例）

c:¥> set SHC_TMP=C:¥tmp

（4） SHC_INCの設定

SuperH RISC engine C/C++コンパイラの標準ヘッダファイルを特定のパスから取り込む場合に指定します。このパスはカンマ(‘,’)で区切ることにより複数指定することができます。この設定がない場合はSHC_LIBから標準ヘッダファイルを取り込みます。

例）

c:¥> set SHC_INC=C:¥Hew3¥Tools¥Renesas¥Sh¥9_0_0¥include

11.1.4 実行時ルーチンの仕様とスピード

■質問

コンパイラが提供する実行時ルーチンのスピードを教えてください。

■回答

内蔵の ROM、RAM を使用したときの実行時ルーチン速度／FPL 速度一覧を掲載します。なお、実行時ルーチンの命名規則については、「**付録 A. 実行時ルーチン命名規則**」を参照してください。ライブラリ構築時のオプションは以下のとおりです。

表 11.1 ライブラリ構築オプション

	cpu	Pic	endian	denormaliaztion	round	fpu	double=float
SH-1	sh1	-	big	-	-	-	**なし**
SH-2	sh2	1	big	-	-	-	**なし**
SH-2A	sh2a	1	big	-	-	-	**なし**
SH-3	sh3	1	big	-	-	-	**なし**
SH-4	sh4	0	big	off	zero	**なし**	-
SH-4A	sh4a	0	big	off	zero	**なし**	-

表 11.2 実行時ルーチン速度／FPL 速度一覧（1）

項番	種類		関数名	スタックサイズ	実行サイクル数					
					SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
1.1	整数演算	乗算	_muli	12	38	-	-	-	-	-
2.1		除算	_divbs	4	38	38	-	26	24	24
2.2			_divbu	0	28	28	-	19	18	18
2.3			_divws	4	49	50	-	34	31	31
2.4			_divwu	0	39	39	-	26	25	26
2.5			_divls	8	37 / 109	39 / 109	-	26 / 73	20 / 50	21 /61
2.6			_divlsp	12	-	84	-	-	-	-
2.7			_divlspnm	8	-	57	-	-	-	-
2.8			_divlu	4	31 / 82	33 / 84	-	22 /56	17 / 50	19 / 50
3.1		剰余	_modbs	8	57	60	-	40	33	33
3.2			_modbu	4	39	40	-	27	23	25
3.3			_modws	8	66	69	-	46	39	39
3.4			_modwu	4	49	50	-	34	29	31
3.5			_modls	12	45 / 95	47 / 97	-	31 / 65	23 / 57	23 / 56
3.6			_modlsp	12	-	84	-	-	-	-
3.7			_modlspnm	8	-	57	-	-	-	-
3.8			_modlu	8	34 / 72	36 / 71	-	24 / 48	18 /43	20 /46

【注】1. **単位は** Cycle**。測定値には誤差が含まれています。**

2. **入力値により処理が大きく異なるルーチンは 最大パターン、最小パターンのそれぞれを掲載。**
[**最小**/**最大**]

RENESAS

表 11.2 実行時ルーチン速度／FPL 速度一覧（2）

項番	種類		関数名	スタックサイズ	実行サイクル数					
					SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
4.1	浮動小数点演算数	加算	_adds	24	129	139	60	80	-	-
4.2			_addd_a	44	320	297	147	195	-	-
5.1		ポストインクリメント	_poas	44	135	145	64	84	-	-
5.2			_poad	84	327	303	150	199	-	-
6.1		減算	_subs	24	144	125	62	86	-	-
6.2			_subdr	44	383	308	149	213	-	-
7.1		ポストデクリメント	_poss	44	175	192	93	120	-	-
7.2			_posd	84	570	550	302	365	-	-
8.1		乗算	_muls	24	144	17	9	11	-	-
8.2			_muld_a	64	383	108	50	69	-	-
9.1		除算	_divs	20	175	17	16	11	-	-
9.2			_divdr	60	570	108	50	69	-	-
10.1		比較	_eqs	20	16	36	16	24	-	-
10.2			_eqd_a	32	90	108	50	70	-	-
10.3			_nes	20	16	36	16	24	-	-
10.4			_ned_a	32	90	108	50	70	-	-
10.5			_gts	20	33	36	16	24	-	-
10.6			_gtd_a	32	90	108	50	70	-	-
10.7			_lts	20	33	36	16	24	-	-
10.8			_ltd_a	32	90	108	50	70	-	-
10.9			_ges	20	33	36	16	24	-	-
10.10			_ged_a	32	90	108	50	70	-	-
10.11			_les	20	33	36	16	24	-	-
10.12			_led_a	32	90	108	50	70	-	-

【注】 単位は Cycle。測定値には誤差が含まれています。

表 11.2　実行時ルーチン速度／FPL 速度一覧（3）

項番	種類	関数名	スタックサイズ	実行サイクル数					
				SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
11.1	符号変換	_negs	0	7	7	4	5	-	-
11.2		_negd_a	12	30	39	18	26	-	-
12.1	変換	_stod_a	12	66	73	35	50	-	-
12.2		_dtos_a	20	122	128	61	82	-	-
12.3		_stoi	12	50	63	21	31	-	-
12.4		_dtoi_a	20	148	141	72	86	-	-
12.5		_stou	12	50	63	21	31	-	-
12.6		_dtou_a	20	148	141	72	86	-	-
12.7		_itos	12	88	91	45	59	-	-
12.8		_itod_a	12	189	179	96	110	-	-
12.9		_utos	8	81	82	46	52	-	-
12.10		_utod_a	8	99	96	51	61	-	-

【注】　単位は Cycle。測定値には誤差が含まれています。

表 11.2　実行時ルーチン速度／FPL 速度一覧（4）

項番	種類	関数名	スタックサイズ	実行サイクル数					
				SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
13.1	**領域移動**	_quick_evn_mvn	4	12+3*(n/4)					
13.2		_quick_mvn	8	17+3*(n/4) (n<=64)					
				24+1.625*(n/4) (n>=68)					
13.3		_quick_odd_mvn	4	12+3*(n/4)					
13.4		_slow_mvn	12	21+5*n+3*((n-1)/4)					
14.1	**文字列比較**	_quick_strcmp1	0	26+7*(n/4)+5*((n-1)%4)					
14.2		_slow_strcmp1	0	35+7*n					
15.1	**文字列コピー**	_quick_strcpy	16	30+6*(n/4)+4*((n-1)%4)					
15.2		_slow_strcpy	24	24+6*n+2*((n-1)/4)					
16.1	**左シフト**	_sftl	4	19 / 42	21 / 39	-	-	-	-
17.1	**右シフト**	_sftrl	0	19 / 42	21 / 39	-	-	-	-
17.2		_sftra	4	20 /43	22 / 47	-	-	-	-
17.3		_sta_sftr6	0	8	9	-	-	-	-
17.4		_sta_sftr7	0	10	11	-	-	-	-
17.5		_sta_sftr10	0	7	8	-	-	-	-
17.6		_sta_sftr11	0	8	9	-	-	-	-
17.7		_sta_sftr12	0	9	10	-	-	-	-
17.8		_sta_sftr13	0	10	11	-	-	-	-
17.9		_sta_sftr21	0	10	11	-	-	-	-
17.10		_sta_sftr27	0	10	11	-	-	-	-
17.11		_sta_sftr28	0	10	11	-	-	-	-
17.12		_sta_sftr29	0	10	11	-	-	-	-
18.1	Packed **構造体**	_pack1_st16	4	12	13	5	10	6	8
18.2		_pack1_st32	4	18	19	8	16	8	12
18.3		_pack1_st64	4	33	35	16	30	16	22
18.4		_pack1_ld16	4	17	18	10	13	11	14
18.5		_pack1_ld32	4	29	30	17	22	18	-
18.6		_pack1_ld64	8	67	73	38	52	39	53
18.7		_bfs64sp1	60	289 / 599	333 / 580	174 / 339	205 / 392	141 / 295	163 / 266
18.8		_bfs64up1	60	289 / 599	333 / 580	174 / 339	205 / 392	141 / 295	163 / 266
18.9		_bfx64sp1	36	239 / 591	276 / 563	144 / 334	194 / 385	130 / 289	147 / 256
18.10		_bfx64up1	40	227 / 588	264 / 550	144 / 332	186 / 377	124 / 282	149 / 266

【注】 1.　**単位は** Cycle**。測定値には誤差が含まれています。**

2.　**入力値により処理が大きく異なるルーチンは 最大パターン、最小パターンのそれぞれを掲載。**[**最小**/**最大**]

表 11.2 実行時ルーチン速度／FPL 速度一覧（5）

項番	種類	関数名	スタックサイズ	SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
19.1	longlong	add64	8	32	42	21	27	18	25
19.2		_sub64	8	32	42	21	27	18	25
19.3		_mul64	36	134	92	40	64	48	45
19.4		_div64s	64	148 / 601	165 / 351	87 / 183	108 / 245	72 / 195	64 / 161
19.5		_div64u	60	121 / 527	137 / 326	74 / 169	90 / 227	59 / 182	51 / 152
19.6		_mod64s	64	142 /550	158 / 342	80 / 179	105 / 241	65 / 190	61 / 155
19.7		_mod64u	60	117 / 569	132 / 312	70 / 165	87 / 223	55 / 178	48 / 147
19.8		_neg64	8	26	33	17	24	15	19
19.9		_not64	8	24	31	16	21	15	19
19.10		_and64	8	32	42	19	28	18	26
19.11		_or64	8	32	42	19	28	18	26
19.12		_xor64	8	32	42	19	28	18	26
19.13		_shlld64	20	86	96	35	45	27	35
19.14		_shlrd64	20	85	94	37	48	29	40
19.15		_shard64	24	93	105	38	49	29	39
19.16		_bfs64s	52	133 / 446	157 / 404	82 / 241	79 / 266	51 /205	59 / 160
19.17		_bfs64u	52	133 / 446	157 / 404	82 / 241	79 / 266	51 /205	59 / 160
19.18		_bfx64s	24	89 / 441	105 / 392	47 / 238	71 / 262	43 / 202	42 / 151
19.19		_bfx64u	24	77 / 428	93 / 379	49 / 238	63 / 254	37 / 195	38 / 148
19.20		_cmplt64	4	23	26	12	16	13	16
19.21		_cmplt64u	4	23	26	12	16	13	16
19.22		_cmpgt64	4	23	26	12	16	13	16
19.23		_cmpgt64u	4	23	26	12	16	13	16
19.24		_cmple64	4	23	26	12	16	13	16
19.25		_cmple64u	4	23	26	12	16	13	16
19.26		_cmpge64	4	23	26	12	16	13	16
19.27		_cmpge64u	4	23	26	12	16	13	16
19.28		_cmpeq64	4	23	27	12	17	13	17
19.29		_cmpne64	4	24	28	14	18	14	18
19.30		_convi64	8	21	26	11	20	11	13
19.31		_convu64	8	18	23	9	18	10	13
19.32		_convs64	20	146	147	81	97	-	-
19.33		_convs64u	20	146	147	81	97	-	-
19.34		_convf64	20	-	-		-	74	67
19.35		_convf64u	20	-	-		-	74	67
19.36		_convw64	20	175	161	86	102	-	-
19.37		_convw64u	20	175	161	86	102	-	-
19.38		_convd64	20	-	-	-	-	75	77
19.39		_convd64u	20	-	-	-	-	75	77
19.40		_conv64i	0	4	4	3	3	3	4
19.41		_conv64u	0	4	4	3	3	3	4
19.42		_conv64s	24	258	260	141	166	-	-
19.43		_conv64us	24	242	246	136	156	-	-
19.44		_conv64f	28	-	-	-	-	78	75
19.45		_conv64uf	28	-	-	-	-	71	65
19.46		_conv64w	20	164	168	88	111	-	-
19.47		_conv64uw	20	133	140	72	93	-	-
19.58		_conv64d	20	-	-	-	-	80	84
19.59		_conv64ud	20	-	-	-	-	67	70

【注】 1. 単位は Cycle。測定値には誤差が含まれています。

2. 入力値により処理が大きく異なるルーチンは 最大パターン、最小パターンのそれぞれを掲載。[最小/最大]

表 11.3 実行時ルーチン速度／FPR 速度一覧

項番	種類	関数名	スタックサイズ	実行サイクル数		
				SH2-DSP	SH3-DSP	SH4AL-DSP
1.1	DSP 実行時ルーチン	_padd24	8	50	33	32
1.2		_padd40	8	60	38	36
1.3		_pdiv16	24	830	514	442
1.4		_pdiv32	36	1164	742	625
1.5		_pdiv24	36	2279	1446	1246
1.6		_pdiv40	36	2750	1696	1439
1.7		_pmul32	16	51	35	32
1.8		_pmul24	24	143	94	87
1.9		_pmul40	44	188	135	105
1.10		_psub24	24	50	33	32
1.11		_psub40	8	60	38	36
1.12		_pconv16s	12	19 / 199	12 / 123	20 / 102
1.13		_pconv16w	16	57 / 212	37 / 126	39 / 115
1.14		_pconv32s	12	20 /340	12 / 196	19 / 140
1.15		_pconv32w	16	53 / 381	34 / 233	37 / 148
1.16		_pconv24s	12	18 / 280	11 / 171	19 / 116
1.17		_pconv24w	16	58 / 286	38 / 168	33 / 172
1.18		_pconv40s	16	29 / 568	18 / 339	24 / 220
1.19		_pconv40w	16	41 / 515	29 / 316	25 / 231
1.20		_pconvs16	16	71 / 1597	47 / 937	50 / 459
1.21		_pconvs32	16	70 / 1341	48 / 809	44 / 457
1.22		_pconvs24	16	104 / 1633	68 / 958	60 / 482
1.23		_pconvs40	16	106 / 1618	70 / 951	64 / 467
1.24		_pconvw16	16	86 / 12374	56 / 7223	49 / 3156
1.25		_pconvw32	20	106 / 3160	68 / 1848	59 / 853
1.26		_pconvw24	20	135 / 10354	86 / 6215	77 / 3172
1.27		_pconvw40	20	142 / 10338	91 / 6207	84 / 3160
1.28		_pcmplt40	4	30	19	17
1.29		_pcmple40	4	30	19	20
1.30		_pcmpgt40	4	30	19	20
1.31		_pcmpge40	4	30	19	20
1.32		_pcmpeq40	4	28	18	16
1.33		_pcmpne40	4	29	18	20
1.34		_pdiv16_sat	28	859	530	459
1.35		_pdiv32_sat	40	1262	790	625
1.36		_pmul32_sat	16	66	42	38

【注】1. 単位は Cycle。測定値には誤差が含まれています。
2. 入力値により処理が大きく異なるルーチンは 最大パターン、最小パターンのそれぞれを掲載。[最小/最大]

11.1.5 SH シリーズオブジェクト互換性

■質問

コンパイルオプション"-cpu=sh1"（または sh2、sh2e、sh3、および sh4）、 "-pic=1"などを使用したオブジェクトをリンクする場合、何か問題がありますか。

■回答

基本的には上位互換であるため、SH-1 のオブジェクトと SH-3 のオブジェクトをリンクし、SH-3 で実行することは可能です。これにより、以前の財産がそのまま使えることになります。

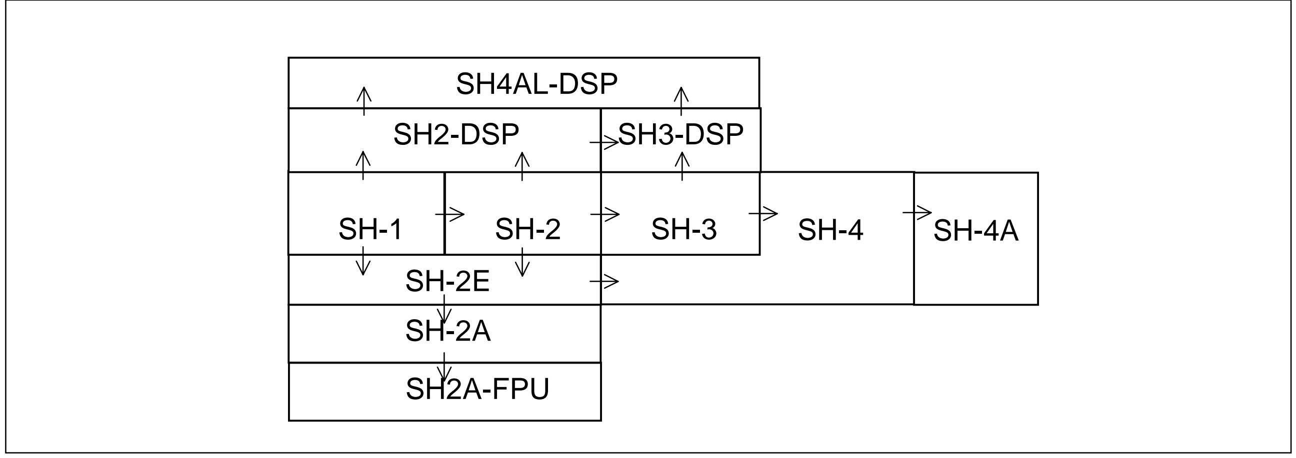


図 11.1 オブジェクト相互関係

【注】 (1) SH-1、SH-2、SH-2E、SH2-DSP、SH-2A、SH2A-FPU **は** Big Endian **です。**SH-3、SH3-DSP、SH4AL-DSP、SH-4、SH-4A **で使うときには** Big Endian **で使用してください。**

(2) "-pic=1"**オプションを付けてコンパイルされたオブジェクトと** "-pic=0"**オプションを付けてコンパイルされたオブジェクトはリンクすることができます。ただし、このときポジションインディペンデントにはなりません。**

(3) SH-3、SH3-DSP、SH4AL-DSP、SH-4、SH-4A **では割り込み時の動作が** SH-1、SH-2、SH-2E、SH2-DSP、SH-2A、SH2A-FPU **の場合と異なり、割り込みハンドラが必要となります。**
"-endian"**オプションについては** 「11.1.15 **データ割り付け** Endian」**も参照してください。**

RENESAS

11.1.6 稼働するホストマシンと OS について

■質問

稼働するホストマシンと OS は何ですか。

■回答

以下に SuperH RISC engine C/C++コンパイラ(Ver. 9.0)の稼働マシンと OS の一覧表を掲載します。

表 11.4 稼働マシン＆OS 一覧表

対応システム名称	OS	備考
HP9000/700 HITACHI9000 HITACHI9000V	HP-UX Ver.10.2	
IBM-PC/AT	Windows98/Me/2000/XP/NT 4.0	Pentium®で動作可能
SPARC	Solaris Ver.2.5 Solaris Ver.8	

11.1.7 C/ C++ソースレベルデバッグができない

■質問

コンパイラオプションに"-debug"を指定しているが、C ソースレベルのデバッグができない。

■回答 1

コンパイル時に加え、リンク時にもデバッグ情報の出力をオプションで指定する必要があります。

また、ソースプログラムが存在するディレクトリがコンパイルしたときと異なる場合、C ソースレベルデバッグはできません。この場合、元のディレクトリに戻すか再コンパイルしてください。

＜リンカ Ver.7 以降をご使用の場合＞

リンク時に出力範囲を指定して複数ファイルに分割して出力した場合、デバッグ情報は各々の出力ファイルには付加せず、デバッグ情報だけ別の１ファイルに出力します。この場合、デバッグ情報ファイルをデバッガに読み込まないと C ソースレベルデバッグはできません。

＜リンカ Ver.6 をご使用の場合＞

リンク時、オプション指定の組み合わせで数種類のオブジェクトフォーマットが出力可能ですが、デバッガによって使用不可能なものがあります。下表からご使用のデバッガにあったオブジェクトフォーマットを選択してください。

表 11.5 使用可能なデバッガとオプション／サブコマンドの関係

使用可能なデバッガ	オプション／サブコマンド	
	オブジェクトフォーマット	デバッグ情報出力
3rd party 製 ELF/DWARF サポートのデバッガ	ELF	DEBUG
日立統合化マネージャ（Ver.4）+E7000	SYSROFPLUS	SDEBUG
日立統合化マネージャ（Ver.3）+E7000	SYSROF	SDEBUG
日立デバッキング インタフェース(Ver.2)+E6000	SYSROF	DEBUG
日立デバッキング インタフェース(Ver.3)+E6000	ELF	SDEBUG

■回答2

-code=asmを指定したときはCソースレベルのデバッグができません。しかしながらインラインアセンブラを使用する場合には-code=asmの指定が必要になります。インラインアセンブラを使用したプロジェクトでCソースレベルのデバッグを行いたい場合は-code=asmをインラインアセンブラが使われているファイルだけに指定してください。

11.1.8 インライン展開時にウォーニングが出る

■質問

（1）インライン展開させようとしたところ、"Function "関数名" in #pragma inline is not expanded"のウォーニングが出ました。

（2）インライン展開させようとしたところ"Function not optimized" のウォーニングが出ました。

■回答

このウォーニングメッセージは実行には支障がありません。

（1） #pragma inline指定をした関数が、インライン展開される条件にあっているかどうか#pragma inlineで指定した関数名の関数と関数指定子inline(C++言語)を指定した関数は、その関数を呼び出したところにインライン展開されます。ただし、以下の場合はインライン展開しません。

- ・#pragma inline指定より前に関数の定義がある。
- ・可変パラメタを持つ関数である。
- ・関数内でパラメタのアドレスを参照している。
- ・展開対象関数のアドレスを介して呼び出しを行っている。
- ・条件／論理演算子の第２演算子以降。

例）

```
#pragma inline(A,B)
int A(int a)
{
     if(a>10) return 1;
     else return 0;
}
int B(int a)
{
     if(a<25) return 1;
     else return 0;
}
void main()
{
     int a;
     if( A(a)==1 && B(a)==1 )
     {
          .....
     }
}
```

A()はインライン展開されるが、B()はインライン展開されない。
(B(a)==1 の判定はしなくても済む場合もあるため)

（2） メモリの不足によるものです。SuperH RISC engine C/C++コンパイラはインライン展開すると関数サイズが大きくなり、最適化処理の途中でメモリが不足し、式単位以上の最適化処理が行えなくなることが考えられます。対策として以下のことを行ってみてください。

- ・大きな関数はインライン展開しない。
- ・多くの箇所で呼び出される関数はインライン展開しない。
- ・インライン展開させる関数の個数を減らす。
- ・メモリを増設する。

11.1.9 コンパイル時に Function not optimized が出る

■質問

オプションに"-optimize=1"をつけてコンパイルしたら"Function not optimized"のウォーニングが出ました。このプログラムは以前同じシステム環境で、同じコンパイルオプションをつけて問題なくコンパイルできたことがあります。これはどういうことですか。

■回答

このウォーニングメッセージは実行には支障がありません。

メッセージが表示された原因には、以下のことが考えられます。

（1）コンパイラの限界値を超えた場合

最適化処理の際にコンパイラが新たな内部変数を生成するため、コンパイラの限界値を超えてしまう場合があります。このような場合、関数を分割することで対処してください。

コンパイラの限界値についてはSuperH RISC engine　C/C++**コンパイラ、アセンブラ、最適化リンケージエディタ ユーザーズマニュアル「**16.1 **コンパイラの限界値」**を参照してください。

（2）メモリが足りない場合

SuperH RISC engine C/C++コンパイラは最適化処理の途中でメモリが不足すると、式単位以上の最適化を中止しこのウォーニングを出します。このとき、コンパイルは継続されますが、得られる結果の最適化レベルはoptimize=0の場合と同じです。このウォーニングを回避するには、Cソースプログラム中の大きな関数を分割するように書き換えてください。

それができない場合にはコンパイラの使えるメモリを増やすしかありません。

（3）インライン展開した場合

「11.1.8　**インライン展開時にウォーニングが出る**」を参照してください。

11.1.10 コンパイル時に compiler version mismatch が出る

■質問

コンパイル時に"compiler version mismatch"というフェータルメッセージが出ます。なぜですか。

■回答

環境変数"PATH"および"SHC_LIB"で指定しているディレクトリが違っていないか確認してください。

例）次のような環境変数の設定になっている場合、上記のメッセージが出力されます。

`PATH =`（SHC Ver.8.0 のパス）
`SHC_LIB =`（SHC Ver. 6.0 のＣコンパイラ本体のパス名称）

11.1.11 コンパイル時に memory overflow が出る

■質問

コンパイル時に"memory overflow"というフェータルメッセージが出ます。なぜですか。

■回答

memory overflow エラーには、以下のようなことが考えられます。

（1） メモリ不足。

（2） 環境変数"SHC_LIB"で指定したパス名のディレクトリにC/C++コンパイラ本体のファイルがすべてそろっていない場合。

例） 次のような設定になっている場合、上記のメッセージが出力されます。

環境変数が SHC_LIB = /SHC/BIN と設定されていて、/SHC/BIN以下と/SHC/MSG以下に別れて、ファイルが格納されている場合。

このときは/SHC/BIN以下にすべてのファイルがなければなりません。

（3） 環境変数の設定が正しくない場合。
PC版の場合は環境変数"SHC_LIB"には、ライブラリのあるディレクトリではなく、SHC.EXEのあるディレクトリを設定してください。コンパイラのインストール時に作成されるSETSHC.BAT では"SHC_LIB"には通常SHC.EXE のあるディレクトリ、"C:¥SHC¥BIN"が設定されます。

11.1.12 インクルード指定の優先順位

■質問

ファイルをインクルードするのに、いろいろなオプションがありわかりづらい。
その用途と優先順位を教えてください。

■回答

インクルードファイルの検索パスを指定するには、オプションまたは環境変数で行います。

「<」、「>」で囲まれたファイルは"-include"オプションで指定されたディレクトリから読み込み、複数ディレクトリを指定した場合は指定した順番に検索します。"-include"オプションで指定されたディレクトリでファイルが見つからない場合は、環境変数 SHC_INC に指定したディレクトリ、次にシステムディレクトリ(SHC_LIB)の順序で各ディレクトリを検索します。

「"」で囲まれたファイルはカレントディレクトリから検索を始めます。カレントディレクトリにない場合は上記の規則に従って検索します。

インクルードファイルの検索パスの優先順位を直感的に示すと

```
-inc > SHC_INC > SHC_LIB
```

となります。

また、上記とは別に指定されたファイルを強制的に読み込ませる"-preinclude"オプションがあります。このオプションが指定されると、コンパイルされるファイルの先頭にオプションで指定されたファイルを挿入してコンパイルを行います。

このオプションで#pragma やテストデータなど一時的に読み込ませたい内容を別ファイルとして読み込ませれば、ソースファイルに手を加えることなくリコンパイルすることができます。

11.1.13 コンパイルバッチファイル

■質問

コンパイルオプションで指定するのが多く、毎回同じものを指定するのが煩わしい。
よい方法はありませんか。

■回答

コンパイル時に"-subcommand"オプション（"-subcommand = <ファイル名>"）を使用します。

"-subcommand"オプションは、コマンドラインの中に複数回指定できます。サブコマンドファイル内には、コマンドラインの引数を空白、改行またはタブで区切って並べてください。サブコマンドファイルの内容がコマンドライン引数の subcommand 指定位置に展開されます。

なお、サブコマンドファイル内に"-subcommand"オプションを指定することはできません。

例）下記の例は、コマンドラインで

```
shc△-optimize=1△-listfile△-debug△-cpu=sh2△-pic=1△-size△-euc
△-endian=big△test.c
```

と入力するのと等価になります。

コマンドライン

```
shc△-sub=test.sub△test.c
```

test.sub の内容

```
-optimize=1
-listfile
-debug
-cpu=sh2
-pic=1
-size
-euc
-endian=big
```

11.1.14 プログラム内への日本語記述

■質問

ソースファイルをワークステーションとパソコンで開発しているが、ワークステーションとパソコンの漢字コードが違うため、ソースファイルの管理が煩わしい。何かよい方法はありませんか。

■回答

漢字コードをシフト JIS で記述しているとき、ワークステーション（EUC コード）でコンパイルする場合は、コンパイラのオプションで"-sj"を使用してください。また、逆に EUC コードで記述している場合は、パソコンでのコンパイル時に、コンパイルオプション"-euc"を指定しコンパイルしてください。EUC、シフト JIS が混在しているワークステーションネットワーク環境でも、コンパイルオプションで指定することにより、どちらの漢字コードでもコンパイルすることができます。

ターゲット（実機）上での漢字コードでコンパイルできます。

表 11.6 システム,漢字コード対応表

ホスト	デフォルト
SPARC	EUC
HP9000/700	シフト JIS
PC9800 シリーズ	シフト JIS
IBM-PC	シフト JIS

例） ワークステーション(SPARC)でソースを書き、パソコン(IBM-PC)上でコンパイルするとき"-euc"オプションを付けてコンパイルすれば文字列中での漢字コードの文字化けを心配する必要がありません。

11.1.15 データ割り付け Endian

■質問

SH のデータ割り付けは Big Endian ですか Little Endian ですか。

■回答

ルネサステクノロジ SuperH RISC engine ファミリは Big Endian です。

ただし、SH-3,SH3-DSP,SH-4,SH-4A,SH4AL-DSP では CPU の Big/Little 切り替え機能に対応して"-endian=Big(Little)"のオプションをサポートしています。

【注】 (1) "-endian"オプションは、"-cpu"オプションの任意のサブオプションと組み合わせが可能ですが、 Little Endianのオブジェクトプログラムは、SH-3、SH3-DSP,SH-4,SH-4A,SH4AL-DSP以外では実行できません。

(2) Big Endian のオブジェクトとLittle Endian のオブジェクトを混在して使うことはできません。

(3) Endianの違いにより、プログラムの実行結果に影響がでることがあります。

例）Endianの違いで影響のでるコーディング

```
f( ){
    int a=0x12345678;
    char *p;

    p=((char *)(&a));

    if(*p==0x12){ (1) }
    else{ (2) }
}
```

この場合、Big Endianならば(1)の処理が実行され、Little Endianならば*pは0x78ですから、(2)の処理が実行されます。

（データの割り付けについて詳しくはSuperH RISC engine C/C++**コンパイラ、アセンブラ、最適化リンケージエディタユーザーズマニュアル「**10.1.2 (4) Little Endian**のメモリ割り付け」**を参照してください。）

(4) “ -denormalize=on|off ” オプション指定により、非正規化数を扱うか０とするかの選択が可能です。(-cpu=sh4または-cpu=sh4a時のみ)
ただし、“ -denormalize=on ” のとき、FPUに非正規化数が入力されると例外発生するので、非正規化数を処理するための例外処理をソフトウェアで記述する必要があります。

11.1.16 #pragma inline_asm 使用時のアセンブル

■質問

#pragma inline_asm を使用したプログラムでアセンブル時に"ILLEGAL DATA AREA ADDRESS（エラー番号 452）"のエラーがでてしまいます。

■回答

（1）"-code=asmcode"オプションを付けてコンパイルしていますか。
（2）アセンブリコード中でデータテーブルを記述していませんか。

これには次のような原因が考えられます。

```
#pragma inline_asm(bar)
int bar()
{
    MOV.L   #160,R9
}
```

上記コーディング中の

```
MOV.L   #160,R9
```

の部分は、SuperHマイコンの命令では値"160"を直接レジスタへMOVする命令がありません。

通常データプールを作成し、ロードしなければなりません。アセンブラではこれを自動的に認識し、データプールを生成していますが、逆に生成された分、コンパイラの出力したアセンブラソース上のアラインメントがずれてしまい、エラーとなります。現在のコンパイラでは、このようにアセンブラが自動的にデータを生成してしまうケースを想定していないため、当面「inline_asm関数中のアセンブラソースには、アセンブラが自動的にデータプールを生成してしまうようなコーディング」はできません。ただし、上記例のケースでは以下のようにコーディングを変更することで回避可能です。

【回避例】

<変更前>

```
MOV.L       #160,R9
```

<変更後>

```
MOV                 #100,R9
ADD                 #60,R9
```

11.1.17 特権モード

■質問

組み込み関数 "set_cr", "get_cr" が正常に動作しません。

■回答

上記組み込み関数は、SH-3、SH-4 では特権モードでのみ使用可能な関数です。

SuperH RISC engine C/C++**コンパイラ、アセンブラ、最適化リンケージエディタユーザーズマニュアル「**10.3.3 **組み込み関数」**参照、特権モードについては**各デバイスのハードウェアマニュアル**参照。当該組み込み関数を呼び出す時点で特権モードになっているかどうかご確認ください。（特権モードは SR レジスタの MD ビットが ON）なお、非特権モードから特権モードに遷移するには TRAPA 命令を発行する必要があります。

11.1.18 オブジェクトの生成について

■質問

コンパイラから直接オブジェクトを生成した場合と、アセンブラを経由して生成した場合では、
（1）プログラムサイズが異なります。
（2）シンボルのTYPEがENTではなくDATになってしまいます。

■回答

直接オブジェクトを生成した場合とアセンブラを経由した場合では、オブジェクト生成方法の違いにより生成されるロードモジュールは一般的に違ってきます。異常ではありません。

アセンブラの出力するオブジェクト上は、ENT と DAT を区別していません。これも異常ではありません。

11.1.19 #pragma gbr_base 指定機能について

■質問

#pragma gbr_base 指定機能を使うとエミュレータへのロード時、または ROM 書き込み時にエラーになります。

■回答

$G0, $G1 セクションは初期化データ領域として取扱ってください。

通常の変数は
（1）「初期値指定なし」で未初期化データ領域（デフォルトセクション名"B"）
（2）「初期値指定あり」で初期化データ領域（デフォルトセクション名"D"）
（3）「const指定あり」で定数領域（デフォルトセクション名"C"）

にそれぞれ割り付けられます。しかし#pragma gbr_base（および gbr_base1）指定した変数はこの区別をせず、すべて$G0（または$G1）セクションに割り付けられるため、コンパイラは $G0,$G1 セクションを初期化データ領域として扱い、変数に初期値が指定されていなかった場合"0"が指定されたものと仮定してオブジェクトを生成しています。

11.1.20 漢字コードを含むプログラムのコンパイル

■質問

SPARC 上でコンパイル可能であることを確認したプログラムを PC 上でコンパイルしたところ、エラーになりました。

■回答

ソースプログラム中に漢字コードが含まれていませんか。SuperH RISC engine C/C++コンパイラは漢字コードとして EUC、シフト JIS をサポートしていますが、デフォルトのコードはホストマシンにより異なります。SPARC 上でのデフォルト漢字コードは EUC ですが、PC はシフト JIS です。EUC 漢字コードが使われているプログラムを PC 上でコンパイルする際は -euc を指定してください。ホストマシンごとのデフォルト漢字コードについては**「11.1.14 プログラム内への日本語記述」**を参照してください。

11.1.21 浮動小数点演算の速度

■質問

浮動小数点演算の処理速度を教えてください。

■回答

標準ライブラリを用いた初等関数の演算速度を**表** 11.8(SH-1,SH-2,SH-3)、**表** 11.9(SH-2E) 、**表** 11.10(SH-4)、**表** 11.11(SH-4A)、**表** 11.12(SH-2A,SH2A-FPU)に示します。なお、四則演算などの浮動小数点演算性能については**「11.1.4 実行時ルーチンの仕様とスピード」**を参照してください。

また、**表** 11.7 に標準ライブラリ構築時の条件を示します。

表 11.7　**標準ライブラリ構築時条件**

条件	**ライブラリ構築時のオプション**						
	Cpu	pic	endian	denormal	round	fpu	double=float
1	sh1	—	big	—	—	—	**なし**
2	sh2	0	big	—	—	—	**なし**
3	sh3	0	big	—	—	—	**なし**
4	sh2e	0	big	—	—	—	**なし**
5	sh4	0	big	off	zero	**なし**	—
6	sh4	0	big	off	zero	single	—
7	sh4	0	big	off	zero	double	—
8	sh4a	0	big	off	zero	**なし**	—
9	sh4a	0	big	off	zero	single	—
10	sh4a	0	big	off	zero	double	—
11	sh2a	0	big	—	—	—	**なし**
12	sh2afpu	0	big	off	zero	**なし**	—
13	sh2afpu	0	big	off	zero	single	—
14	sh2afpu	0	big	off	zero	double	—

表 11.8　浮動小数点ライブラリの演算速度(SH-1,SH-2,SH-3)

CPU		SH-1	SH-2	SH-3
ライブラリ構築条件		1	2	3
単精度	Sinf	2,438	2,497	1,632
	Cosf	2,384	2,434	1,599
	Tanf	3,120	3,196	2,091
	asinf	5,176	5,418	3,526
	acosf	5,355	5,622	3,659
	atanf	2,924	3,160	2,054
	logf	3,710	3,816	2,490
	sqrtf	3,252	1,018	661
	expf	4,327	4,432	2,873
	powf	4,649	4,824	3,139
倍精度	sin	5,297	4,964	3,282
	cos	5,289	4,918	3,279
	tan	7,460	7,087	4,673
	asin	13,898	13,788	9,004
	acos	14,158	14,084	9,196
	atan	5,583	5,687	3,712
	log	8,756	8,368	5,535
	sqrt	2,903	2,946	1,803
	exp	9,501	8,952	5,912
	pow	9,337	8,943	5,918

【注】　単位は Cycle。測定値には誤差が含まれています。

表 11.9　浮動小数点ライブラリの演算速度(SH-2E)

CPU		SH-2E
ライブラリ構築条件		4
単精度	sinf	307
	cosf	302
	tanf	343
	asinf	1,267
	acosf	1,289
	atanf	468
	logf	213
	sqrtf	648
	expf	299
	powf	472
倍精度	sin	3,005
	cos	3,002
	tan	4,339
	asin	8,544
	acos	8,717
	atan	3,434
	log	5,144
	sqrt	1,896
	exp	5,475
	pow	5,437

【注】　単位は Cycle。測定値には誤差が含まれています。

表 11.10 浮動小数点ライブラリの演算速度(SH-4)

CPU		SH-4		
ライブラリ構築条件		5	6	7
単精度	Sinf	63	59	139
	Cosf	62	59	135
	Tanf	80	78	186
	Asinf	75	71	264
	Acosf	72	72	269
	Atanf	104	72	155
	Logf	86	85	192
	Sqrtf	—*	—*	—*
	Expf	119	100	193
	Powf	387	366	213
倍精度	Sin	331	70	139
	Cos	310	66	135
	Tan	408	71	186
	Asin	523	71	206
	Acos	616	72	253
	Atan	393	58	145
	Log	403	85	192
	Sqrt	—*	—*	—*
	Exp	403	90	193
	Pow	1,032	366	213

【注】 単位は Cycle。測定値には誤差が含まれています。
* SH-4 では sqrt の命令があるため sqrt の関数は省略。

表 11.11 浮動小数点ライブラリの演算速度(SH-4A)

CPU		SH-4A		
ライブラリ構築条件		8	9	10
単精度	Sinf	100	95	195
	Cosf	113	96	188
	Tanf	139	134	277
	Asinf	117	113	336
	Acosf	124	123	344
	Atanf	148	122	205
	Logf	131	130	233
	Sqrtf	—*	—*	—*
	Expf	169	146	219
	Powf	408	388	194
倍精度	Sin	305	110	194
	Cos	288	107	187
	Tan	377	128	247
	Asin	466	113	267
	Acos	558	122	324
	Atan	331	97	191
	Log	344	130	133
	Sqrt	—*	—*	—*
	Exp	387	133	256
	Pow	877	388	219

【注】 単位は Cycle。測定値には誤差が含まれています。
* SH-4A では sqrt の命令があるため sqrt の関数は省略。

表 11.12　浮動小数点ライブラリの演算速度(SH-2A, SH2A-FPU)

CPU		SH-2A	SH2A-FPU		
ライブラリ構築条件		11	12	13	14
単精度	Sinf	1,001	68	65	139
	Cosf	954	68	64	135
	Tanf	1,806	83	82	188
	Asinf	1,545	79	75	273
	Acosf	1,699	74	73	277
	Atanf	1,602	98	73	156
	Logf	1,720	92	93	196
	Sqrtf	562	—*	—*	—*
	Expf	1,463	121	102	208
	Powf	2,140	407	386	246
倍精度	Sin	3,431	302	77	140
	Cos	3,387	288	72	135
	Tan	4,425	385	75	188
	Asin	5,550	463	75	208
	Acos	5,949	544	73	259
	Atan	3,641	348	59	145
	Log	4,557	401	93	196
	Sqrt	1,622	—*	—*	—*
	Exp	4,137	410	93	208
	Pow	4,086	903	386	246

【注】　単位は Cycle。測定値には誤差が含まれています。
　　* SH2A-FPU では sqrt の命令があるため sqrt の関数は省略。

11.1.22 PIC オプションの使用方法

■質問

ポジションインディペンデントコードを使用してプログラミングをしたいのですがどうしたら良いでしょうか。

＜詳細内容＞

(1) 複数のアプリケーションを動的に空いているRAM領域に転送して実行したい。
(2) 初期化処理はどうするか。
(3) 使用上の制限事項、注意点は何か。

■回答

プログラムをROM上から､RAM上の固定アドレスへ転送して実行するなら､-PICオプションは使用せずに､**「11.2.4 プログラムのRAMへの転送実行」**の手法でプログラミングしてください。

動的に空いているRAM領域へ転送したい場合は-PICオプションが有効ですが、本オプションはプログラムセクションにのみ有効で､データに対してはポジションインディペンデントになりません。よって､データ領域については､固定のアドレスにロードすることしかできないため注意が必要です。

このような制約があるため、プログラム全体（データを含む）をポジションインディペンデントにするためには、プログラムの記述方法を工夫する必要があります。

以下に、データセクションを含まない場合のプログラミング手順を説明します。

・データセクションを含まない場合のプログラミング手順

プログラム構成アプリケーション側

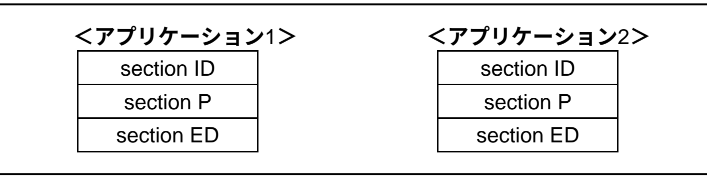


図 11.2 セクション

C言語プログラム

```
<main.c>
main()
{
     int i;
     for (i=0;i<10;i++){
        sub(i);
     }
}

<sub.c>
sub(int p)
{
    int i;
    for (i=0;i<p;i++){
         ;
    }
}
```

アセンブラプログラム

```
<pic.src>
        .import                 _main
        .section        ED,DATA,ALIGN=4   ; 終了のセクション ED を生成
        .section        ID,DATA,ALIGN=4   ;  ヘッダ用のデータセクション
        .data.l         (STARTOF ED)
        .data.l         _main
```

```
        .end

<lnk.sub>
        input main
        input sub
        input pic
        start ID,P,ED/0;0 番地から割り付ける。先頭に ID、最後が ED とする
        list  pic
        exit
```

このように各プログラムにヘッダ（ID セクション）を付けます。

ID セクションの内容は
オフセット 0 番地　　プログラムのサイズ
オフセット 4 番地　　エントリーポイント（main のアドレス）

このような形でプログラムを個々に作成し、これらをコントロールするプログラムが、ID を見ながら、ロードアドレスと、実行アドレスを計算します。
以下は、コントロールプログラムのイメージです。

```
<control.c>
void load_program(int ID){
      char *p;
      size=load_ID( ID);        /* プログラムの ID ヘッダデータをロードする */
                                /* 戻り値は プログラムサイズ               */
     p=malloc(size);
     if(p!=NULL){
       mload(p,ID);             /* ヒープ領域にプログラムデータを書き込む  */
      go((*(long**)p+1)+(long*)p);
                                /* プログラムの先頭アドレスに PC を         */
                                /*  セットして実行させる。                 */
    }
    else {
       error("Insufficient Memory");
   }
}
```

このプログラムは、プログラムイメージであり、使用している OS に依存して実装方法は異なってきます。上記の例は、動的にプログラムを動作させる場合のフローレベルのものと考えてください。

11.1.23 最適化によって、コードが大幅に削除されてしまう

■質問

コンパイル後のコードが大幅に削除されてしまいます。

■回答

以下のような最適化の可能性があります。

(1) 空ループの削除

プログラムにある一定時間の待ちを与えるために、空ループを記述しても最適化によりループ自体が削除されます。

例）

```
set_param();           /* パラメタセット                           */
for(i=0;i<10000;i++);  /* パラメタセット後、結果がセットされる     */
                       /* まで一定時間待ちを与えようとする空ループ */
                       /* コンパイラは無意味なループとして         */
                       /* ループ自体を削除する                     */
 read_data();          /* 結果の取得                               */
                       /* ループが削除されたことにより、待ち時間が */
                       /* なくなり結果を得られる前に参照しNGとなる */
```

(2) ローカル変数への代入削除

ローカル変数に値を代入しているにもかかわらず、その値を参照していなければ、代入のための演算処理自体が削除されます。

例）

```
int data1, data2, data3;
func()
{
int   res1,res2,res3;

res1=data1*data2;
res2=data2*data3;        /*  res2はこの後参照されないため式自体が削除   */
res3=data3*data1;
sub(res1,res1,res3);     /*  第2パラメタの記述ミス                      */
                         /*  res1->res2とすれば削除しない               */
}
```

ローカル変数は、関数の末までが有効な区間なので、関数内で値を代入して、参照しないようなことは普通ありません。よって、この例のようなコーディングのミスで引き起こされるようなケースが考えられます。

11.1.24 デバッグ時にローカル変数の値が見えない

■質問

ローカル変数の値が見えません。
デバッガでローカル変数を参照しましたが、値が参照できない、または値が異なっています。

■回答

以下のような最適化の可能性があります。

(1) コンパイル時の定数演算

コンパイル時にあらかじめ値が確定してしまうものは、実行時に演算しないでコンパイル時に演算してしまうため、変数自体がなくなってしまうことがあります。

例1）

```
int x;
func()
{
int a;
a=3;
x=x+a;    /* こういった場合は、aはコンパイル時  x=x+3; となる。      */
          /* このほかに、aが使用されないような場合は、aを変数と        */
          /* して扱う意味がないため、デバッグ情報としても削除される。  */
}
```

例2）

```
func(int a,int b)
{
int tmp;
int len;

tmp=a*a+b*b;
len=sq(tmp);    /* len=sq(a*a+b*b); とされ tmpが削除される。     */

:
}
```

このようなケースが考えられますが、実際のプログラム動作には影響はありません。

(2) 未参照変数の削除

例3）

```
int data1, data2, data3;
func()
{
int   res1,res2,res3;

res1=data1*data2;
res2=data2*data3;       /*  式が削除されres2自体も削除される    */
res3=data3*data1;
sub(res1,res1,res3);    /*  第2パラメタの記述ミス               */
                        /*  res1->res2とすれば削除しない        */
}
```

ローカル変数は、関数の末までが有効な区間なので、関数内で値を代入して参照しないようなことは普通ありません。よって、この例のようなコーディングのミスで引き起こされるようなケースが考えられます。

11.1.25 割り込み禁止／許可マクロ

■質問

割り込みの禁止／許可をマクロで実現したいのですが、どうしたらよいでしょうか。

■回答

組み込み関数を用いて以下の例のように実現できます。組み込み関数の詳細については SuperH RISC engine C/C++**コンパイラ、アセンブラ、最適化リンケージエディタユーザーズマニュアル**の「10.3.3 **組み込み関数**」を参照してください。

例)

```
#include <machine.h>
#define disable() { save_cr=get_cr(); set_imask(0x0f); }
#define enable()  { set_cr(save_cr); }

function()
{
    int save_cr;

    disable();
    sub();
    enable();
}
```

11.1.26 SH-3 以降での割り込み関数

■質問

SH-3 以降の SuperH マイコンでは割り込み関数の書き方が異なっているのでしょうか。

(1)多重割り込みさせたいが #pragma interrupt 指定した関数では

(a) SSR,SPC の退避命令が出ない。

(b) SR の RB,BL ビットのクリア命令が出ない。

(c) SSR,SPC の回復命令が出ない。

(2)#pragma interruptでTRAPナンバ指定を使いたいが、SRのBLビットが1のままなので、このままではTRAPA命令発行時に命令例外が発生してしまいます。

■回答

コンパイラは SSR, SPC の退避回復命令を出力しません。#pragma inline_asm 機能を用いて明示的に記述するか、あるいはプログラムをアセンブリ言語で記述してください。SR の設定は組み込み関数 set_cr, get_cr を用いて記述できます。

SH-3 以降は SH-1、SH-2、SH-2E と比べて割り込み時の動作が大きく異なります。SH-1、SH-2、SH-2E では割り込み時にはベクタテーブルを参照して、対応する割り込みルーチンへ分岐しましたが、SH-3 以降では固定的なアドレスへ分岐します。このため通常は割り込みでの分岐先に、多重割り込みの許可／禁止、割り込み要因の判定と要因ごとの処理の起動などを行う割り込みハンドラが必要となります。通常この割り込みハンドラはアセンブリ言語で書かれることになります。

「**2.2、2.3 サンプルプログラムの紹介**」参照。

11.1.27 SH4 の浮動小数点演算結果

■質問

SH4 の浮動小数点演算で演算結果が期待値とならない。

■回答

コンパイルオプションの 1 つに FPU オプションがあります。これは、FPU=single/double/指定なしの 3 通りがあります。この違いは、

FPU=single：全ての浮動小数点データを単精度とする
FPU=double：全ての浮動小数点データを倍精度とする
FPU 指定なし：浮動小数点データをソースコードの型どおりに扱う
と言うコード生成を行います。また、それに伴って、FPSCR レジスタの PR ビットの値の考慮が必要になります。

(1) CPUの初期状態ではこの値(PRビット)は「0(=単精度)」となっています。
(2) コンパイル時にFPUオプションの指定がない場合は、浮動小数点の演算時にその精度に応じて上記ビットを変更するコードが生成されますが、FPU=double/singleを指定した場合には、上記ビットにアクセスするコードが一切生成されません。
(3) そのため、FPU指定なし、FPU=single指定の場合には結果的に上記ビットを意識しなくても正しく動作を致しますが、FPU=double指定時にはユーザ側にて上記ビット(PRビット)に明示的に[1(=double)」を指定する必要があります。

11.1.28 最適化オプションについて

■質問

最適化オプション(speed, size)によって、何が変わるのでしょうか。

■回答

指定する最適化オプションによって、生成されるコードが変わってきます。（なお、最適化によってユーザプログラムのアルゴリズムを崩すようなことはしません。）最適化により、関数のインライン展開やループ展開が行われ、実行時のサイクル数が変わってきます。これにより当然のことながら動作のタイミングが変わってきます。まずは、タイミングについて十分検証頂きますよう、お願いいたします。また、上記以外の懸念事項として変数アクセスの最適化も考えられます。データの授受がメモリを介さずにレジスタ間で実現できてしまうような場合がこれに該当し、一言で言うと、「タイミング検証」の範疇になるのですが、「最適化して欲しくない」変数については、volatile 宣言の付加の必要性なども含めて、ご確認いただけるようお願い致します。

11.1.29 関数の引数が正しく渡されない

■質問

関数の引数が正しく渡されない。

■回答

関数の原型宣言をしているか確認してください。

関数の原型宣言をしていないと、引数(char,unsigned char,float)は自動型変換の対象になります。そのとき、呼び出される関数側は変換後の型で宣言する必要があります。

関数は原型宣言することをお勧めします。

なお、コンパイラの message オプションで、関数の原型宣言の有無をチェックできます。

11.1.30 不正動作になりやすいコーディングを調べたい

■質問

関数の原型宣言の抜けなど、不正動作になる可能性のあるコーディングを調べる機能はありませんか。

■回答

言語仕様上誤りではありませんが、プログラム記述において、不正動作になる可能性のあるコーディングがあります。それらのコーディングに対し、オプションで、インフォメーションメッセージを出力しチェックすることができます。

Ver.9 以上であれば MISRA-C チェックツールを使用できます。

例）

```
   shc △ -message △ test.c (RET)

（Cプログラム）

/*  /* COMMENT */           →5009 : 注釈の中に、文字列「/*」があります。
int ;                       →0002 : 宣言子のない宣言があります。
void func(int);
void main(void)
{
  long a;
  func(a+1);                →0006 : 関数の引数の式が、原型宣言で指定した引数の型に変換さ
                                    れます。
  sub();                    →0200 : 呼び出す関数の原型宣言がありません。
}
```

【指定方法】

ダイアログメニュー：コンパイラタブ カテゴリ:[ソース] インフォメーションメッセージのインフォメーションレベルメッセージの表示

コマンドライン　　：*message*

■備考

ダイアログメニューでは、個々のメッセージの左側のチェックを外すと、そのメッセージの出力を抑止することができます。コマンドラインでは、*nomessage* オプションのサブオプションでエラー番号を指定すると、そのメッセージの出力を抑止することができます。各エラー番号の詳細については、SuperH RISC eigine C/C++**コンパイラ、アセンブラ、最適化リンケージエディタユーザーズマニュアル**の**「第 12 章　コンパイラのエラーメッセージ」**を参照してください。

コンパイラはインフォメーションメッセージを出力した後エラー回復をし、オブジェクトプログラムを生成します。コンパイラの行ったエラー回復がプログラムの意図と一致しているかどうかを確認してください。

11.1.31 コメントの記述について

■質問

（1） コメントをネストして記述したい。
（2） C++用のコメントをＣプログラム中に記述したい。

■回答

（1） オプションで、コメントをネストして記述してもエラーとならない指定ができます。

【指定方法】

ダイアログメニュー：コンパイラタブ カテゴリ:[その他] その他のオプション:のコメントネスト(/* */)のネストを許す
コマンドライン　　：*comment*

表 11.13 コメントのネスト

C/C++ソース内記述	コメントネスト許可なし	コメントネスト許可
/* comment */	コメント文として認識	コメント文として認識
/* /* comment */ */	記述誤り	コメント文として認識
/* /* /* comment */	コメント文として認識	記述誤り

（2） C++用のコメント記述"//"を記述することが可能です。Cのコメント（/*　*/）との間には次のような関係があります。コメントとして認識する部分は下線をしてあります。

```
void func()
{
    abc=0;       //  /* comment  */
    
    def=1;       /* comment
    ghi=2;       // comment       */
}
```

←//以降をコメントとして認識

←/*　*/で囲まれた部分をコメントとして認識

11.1.32 アセンブラを埋め込んだ場合のビルドの仕方

■質問

#pragma inline_asm を使用して、アセンブラ埋め込みを行った場合に、コンパイル時にウォーニングメッセージが出力される。

■回答

アセンブラ埋め込みを行ったファイルは、アセンブリ言語で出力し、その後、アセンブルしなければなりません。
HEW 上でのビルドの方法としては、アセンブラ埋め込みのあるファイルをアセンブル出力指定します。そして、ビルドすると、アセンブル出力されたファイルは、自動的にアセンブルされます。
例として、test.c をアセンブラ埋め込みのあるファイルとすると、次のように指定します。

＜HEW2.0 以降＞

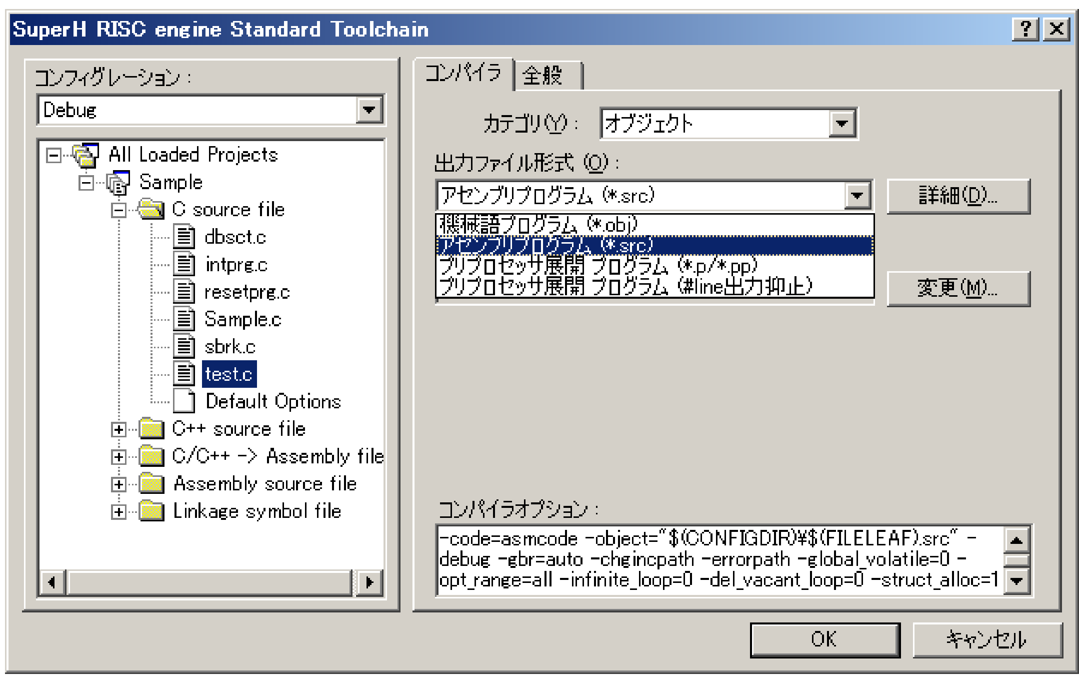


図 11.3 コンパイラダイアログボックス

コンパイラタブカテゴリ:[オブジェクト]の出力ファイル形式:からアセンブリプログラム (*.src) を選択します。
この指定で、通常にビルドすることができます。
ただしこの指定を行うと C ソースデバッグができなくなります。

11.1.33 C++ 言語仕様についての機能

■質問

C++言語で開発していく上で、用意されている機能はありますか？

■回答

SuperH RISC engine C/C++コンパイラでは C++言語開発用に以下の機能をサポートしています。

（1） EC++クラスライブラリサポート
EC++クラスライブラリをサポートすることにより､C++プログラムから組み込み向けC++クラスライブラリを標準的に利用することができます。
このライブラリには次の4種類があります。
・ストリーム入出力用クラスライブラリ
・メモリ操作用ライブラリ
・複素数計算用クラスライブラリ
・文字列操作用クラスライブラリ

それぞれの内容の詳細は SuperH RISC engine C/C++**コンパイラ、アセンブラ、最適化リンケージエディタ ユーザーズマニュアル**の「10.4.2 EC++**クラスライブラリ**」を参照してください。

（2） EC++言語仕様シンタックスチェック機能
コンパイラのオプション指定すると、EC++言語仕様に基づいてC++プログラムをシンタックスチェックします。

【指定方法】

ダイアログメニュー：コンパイラタブ カテゴリ:[その他] その他のオプション:EC++言語に基づいたチェック
コマンドライン　　：*ecpp*

（3） その他の機能
その他にC++プログラムを効率よく記述するために次のC++機能を理解して使ってください。
＜Better C機能＞
・関数インライン展開
・+,−,<<などの演算子カスタマイズ
・多重定義関数による名称の単純化
・コメント記述容易
＜オブジェクト指向機能＞
・クラス
・コンストラクタ
・仮想関数

C++プログラムでライブラリ関数を使用する場合の実行環境の設定についてはSuperH RISC engine C/C++**コンパイラ、アセンブラ、最適化リンケージエディタ ユーザーズマニュアル**の「9.2.2(4) C/C++**ライブラリ関数の初期設定**（_INTLIB）」を参照してください。

11.1.34 プリプロセッサ展開後のソースが見たい

■質問

マクロなどを展開したあとのプログラムを見てみたい。

■回答

コンパイラのオプション指定でプリプロセッサ展開後のソースが出力されます。

展開前のソースプログラムがＣ言語の場合は、拡張子が、<ファイル名>.p として出力されます。C++言語の場合は<ファイル名>.pp となります。

この場合は、オブジェクトが作成されませんので、最適化に関するオプションなどを指定しても、無効となります。

【指定方法】

ダイアログメニュー： コンパイラタブ カテゴリ:[オブジェクト] 出力ファイル形式:のプリプロセッサ展開プログラム(*.p/*.pp))

コマンドライン ： *preprocessor*

11.1.35 ICE でうまく行くが実チップ上では暴走する

■質問

ICE でデバッグするとうまく行くが、実チップ上で動かすと NG となる。

■回答

初期化データ領域（Ｄセクション）があると、ICE では代替メモリを使用するため、リード／ライトすることができますが、実チップ上のメモリは ROM であるため、リードしかできません。そのため、書き込みにいったときに暴走します。

初期化データ領域はパワーオンリセット時に ROM 領域から RAM 領域へコピーする必要があります。

HEW2.0 以降最適化リンケージエディタ、HEW1.2 モジュール間最適化ツールの ROM 化支援オプションを使用して、ROM と RAM に二重に領域を確保します。

ROM 領域から RAM 領域へのコピー方法については「**2.2.4 初期化部の作成**」を参照してください。

11.1.36 H8 マイコン用に開発した C プログラムの利用について

■質問

H8 マイコン用に開発したＣプログラムを SH マイコン用に利用したいが、注意点は？

■回答

プログラム上で注意しなければならない点は、次のとおりです。

（1） int型データは４バイトデータとなります。
H8S,H8/300シリーズではint型データは2バイトデータとして扱っていましたが、SHシリーズでは4バイトとなるため、値の範囲が問題ないか確認してください。

（2） 一部の拡張機能で使用できないものがあります。
SHシリーズC/C++コンパイラとH8S,H8/300シリーズC/C++コンパイラではそれぞれに#pragma文などで固有の処理を行えますが、使用できない拡張機能や、仕様が異なる場合があります。
また、組み込み関数はCPU固有ですのでご注意ください。

（3） アセンブラ埋め込み部分の注意
SHシリーズとH8S,H8/300シリーズではアーキテクチャが違うため、H8S,H8/300シリーズのアセンブリソースを組み込んだ部分は使えません。

なお、M32R マイコン用に開発した C プログラムを SH マイコンに利用する場合には、ソース移植用補助ツールである Translation Helper をご使用ください。

Translation Helper は M32R マイコンから SH マイコンへのプロジェクト変換、および C ソースプログラムの変換を自動的に行うツールです。

Translation Helper はルネサス開発環境 WEB より無償で配布しております。

11.1.37 最適化により無限ループになる

■質問

コンパイラをバージョンアップしたり、最適化を ON にすると、無限ループから抜けない状況になる場合があります。

■回答

コンパイラの最適化により、無限ループになるよくある例としては次ソースのように a への代入がメモリからのリードではなくレジスタからのリードになっているために、割り込みなどで*d の値を変えてもそれが反映されず無限ループになることがあります。本最適化はコンパイラの仕様です。volatile 型指定子を付記することで最適化を抑止することが可能です。

例）

Ｃソース

```
void f( int *d)
{
       int a;
       do
       {
              a=*d;
       }while(a!=0);
}
```

最適化ありアセンブラソース

```
_f:                             ; function: f
                               ; frame size=0
         .STACK      _f=0
         MOV.L       @R4,R2
L11:
         TST         R2,R2       ;メモリからリードされない。
         BF          L11
         RTS
         NOP
         .END
```

変更後Ｃソース

```
void f( volatile int *d)
{
       int a;
       do
       {
              a=*d;
       }while(a!=0);
}
```

変更後最適化ありアセンブラソース

```
_f:                          ; function: f
                             ; frame size=0
        .STACK     _f=0
L10:
        MOV.L      @R4,R2    ;メモリからリードされる。
        TST        R2,R2
        BF         L10
        RTS
        NOP
       .END
```


RENESAS

11.1.38 DSP ライブラリの注意事項

■質問

DSP ライブラリを使用していますが、実行中に不正終了してしまう、期待する結果が得られない、などで苦労しています。DSP ライブラリ使用時の注意事項を教えて下さい。

■回答

以下をご確認ください。

1.メモリの破壊が起きていないか
DSP ライブラリはヒープメモリを使用しています。ヒープメモリ破壊が起きている場合、正しい演算結果は得られません。
また、異常動作の原因ともなります。

2.DSP メモリ（X,Y メモリ）が正しく使用されているかどうか
DSP ライブラリ関数の中には入出力データを X/Y メモリに配置する必要があるものがあります。その場合はそれぞれの DSP ライブラリ関数の説明に従って入出力データを含むセクションを X/Y メモリに割り付けるようにしてください。（#pragma section を使用すると細かなセクション切り分けができます。詳しくは「3.7.2 **セクション切り替え**」を参照してください。また-dspc オプション使用時には X/Y メモリ修飾子を使用することで容易に X./Y メモリ用のセクション切り分けができます。）
またフィルタ関数使用時は作業領域を Y－RAM に割り付ける必要があります。-dspc オプションを指定しない場合は DY,BY セクションを、-dspc オプションを指定した場合は$YD,$YB セクションをリンク時に Y－RAM に割り付けるようにしてください。

3.DSP ライブラリ関数の使用方法
DSP ライブラリ関数によっては、使用前後に特別な処理が必要なものがあります。
使用前後を含めて、当該ライブラリ関数を正しく使用しているかどうかをご確認ください。
各ライブラリ使用方法の詳細は「3.13 DSP **ライブラリ**」を参照してください。

4.スケーリングによる誤差
DSP ライブラリ関数はスケーリング処理を行っており、その過程で誤差が生じる場合があります。
スケーリングに関する詳細は、「3.13 DSP **ライブラリ**」を参照してください。

11.1.39 DSP ライブラリの最大サンプリングデータ数

■質問

DSP ライブラリ関数における、最大サンプリングデータ数を教えてください。

■回答

DSP ライブラリ関数における最大サンプリングデータ数は、大別して DSP メモリ（X,Y メモリ）の容量、および in-place/non-in-place 関数であるかどうかの２つに大きく依存します。

in-place の関数は、

(最大サンプリング数) = (X or Y メモリサイズ) / (2 (=short 型のサイズ))

not-in-place の関数は、入出力の領域を別にする必要あるのでさらに半分です。

X-RAM、Y-RAM が 8K の場合、

・FftComplex
最大サンプリング数：2048
使用ヒープサイズ：17334

・FftReal
・入力データを X/Y メモリ以外に配置
最大サンプリング数：4096
使用ヒープサイズ：18358
・入力データを X/Y メモリに配置
最大サンプリング数：2048
使用ヒープサイズ：17334

・IfftComplex
最大サンプリング数：2048
使用ヒープサイズ：17334

・IfftReal
最大サンプリング数：2048
使用ヒープサイズ：19382 (17334 + 2048)
(IfftReal()内でも malloc で領域を確保するため)

・FftInComplex
最大サンプリング数：4096
使用ヒープサイズ：18358

・FftInReal
最大サンプリング数：4096
使用ヒープサイズ：18358

・IfftInComplex
最大サンプリング数：4096
使用ヒープサイズ：18358

・IfftInReal
最大サンプリング数：4096
使用ヒープサイズ：18358

となります。

11.1.40 ビットフィールドのリードライト命令

■質問

```
struct bit{
 unsigned short int b0 : 1;
 unsigned short int b1 : 1;
 unsigned short int b2 : 1;
 unsigned short int b3 : 1;
 unsigned short int b4 : 1;
 unsigned short int b5 : 1;
 unsigned short int b6 : 1;
 unsigned short int b7 : 1;
 unsigned short int b8 : 1;
 unsigned short int b9 : 1;
 unsigned short int b10 : 1;
 unsigned short int b11 : 1;
 unsigned short int b12 : 1;
 unsigned short int b13 : 1;
 unsigned short int b14 : 1;
 unsigned short int b15 : 1;
} ;
```

のような、ビットフィールドを定義し、16 ビット幅で特定レジスタのビットをアクセスしたいのですが、バイトアクセスやビット操作命令でのアクセスになってしまいます。16 ビットしかアクセスできないレジスタである場合には、バイトアクセスやビット操作命令を生成されると、レジスタ値が正常に読めない現象が発生します。

■回答

プログラム中で特に指定がないかぎりビットフィールドのメンバにはコンパイラにより最適な命令でアクセスされます。SH-2A,SH2A-FPU はビットアクセス命令を生成します。それ以外の CPU はバイトアクセス命令を生成します。その結果意図しない命令でのアクセスになることがあります。volatile をつけることでメンバ変数の型どおりの命令でアクセスすることができます。

アクセス方法や複数アクセスの抑止などに関して、コンパイラによる変更を抑止したい変数に対しては、明示的に volatile 指定してください。

volatile なし C ソース

```
struct bit reg;

void f()
{
      reg.b6=1;
}
```

volatile なしアセンブラソース(SH-2A,SH2A-FPU 以外)

```
 _f:                      ; function:f
                          ; frame size=0
         .STACK      _f=0
         MOV.L       L11+2,R6   ; _reg
         MOV.B       @R6,R0
         OR          #2,R0
         RTS
         MOV.B       R0,@R6
```

volatile なしアセンブラソース（SH-2A,SH2A-FPU）

```
  f:                              ;function:f
        .STACK              _f=0
         MOV.L               L11,R2    ; reg
         BSET.B              #1,@(0,R2)
         RTS/N
```

volatile ありＣソース

```
volatile struct bit reg;

void f()
{
        reg.b6=1;
}
```

volatile ありアセンブラソース(SH-2A,SH2A-FPU 以外)

```
_f:                             ; function: f
                                ; framesize=0
        .STACK     _f=0
        MOV.L      L11+2,R6   ; _reg
        MOV        #2,R5      ; H'00000002
        MOV.W      @R6,R2
        SHLL8      R5
        OR         R5,R2
        RTS
        MOV.W      R2,@R6
```

volatile ありアセンブラソース（SH-2A,SH2A-FPU）

```
f:                           ;function: f
                             ;framesize=0;
      .STACK   _f=0
      MOV.L    L11+2,R6;reg
      MOV.W    @R6,R2
      MOVI20   #512,R5  ;H'00000200
      OR       R5,R2
      RTS
      MOV.W    R2,@R6
```

またビットフィールドの型が long long の時は常に実行時ルーチンによるアクセスになります。

Ｃソース

```
struct bit{
 unsigned long long int b0 : 1;
 unsigned long long int b1 : 1;
 unsigned long long int b2 : 1;
 unsigned long long int b3 : 1;
 unsigned long long int b4 : 1;
 unsigned long long int b5 : 1;
 unsigned long long int b6 : 1;
 unsigned long long int b7 : 1;
};

struct bit reg;

void f()
{
       reg.b6=1;
}
```

アセンブラソース

```
_f:                             ; function: f
                                ; frame size=12
        .STACK     _f=12
        STS.L      PR,@-R15
        MOV        #1,R1      ; H'00000001
        MOV.L      R1,@-R15
        MOV        #0,R4      ; H'00000000
        MOV.L      R4,@-R15
        MOV.L      L11+4,R1   ; _reg
        MOV.L      L11+8,R5   ; __bfs64u_p
        MOV.W      L11,R0     ; H'0601
        JSR        @R5
        MOV        R15,R2
        ADD        #8,R15
        LDS.L      @R15+,PR
        RTS
        NOP
```


RENESAS

11.1.41 割り込み処理を記述したい

■質問

割り込み処理を記述したいのですが、どうしたら良いですか。

■回答

割り込み処理を記述するにはまず HEW のプロジェクト構築時にベクタテーブル定義をチェックしてプロジェクトを構築してください。割り込み処理関数のテンプレートが記述されたファイルが生成されますのでそれを編集してください。SH-1,SH-2 と SH-3,SH-4 では割り込みの処理方式が異なるので HEW が生成するファイルも異なります。

- SH-1,SH-2 の場合

割り込み処理を使用するには割り込み処理関数とベクタテーブルとステータスレジスタの割り込みマスクビットの初期化が必要になります。SH-1、SH-2 の場合の例として SH7020 のプロジェクトで割り込み要因 IRQ0 の処理を記述します。

1．割り込み処理関数

HEW が空の割り込み処理関数を用意しています。intprg.c に void INT_IRQ0(void)が定義されていますのでここに IRQ0 の処理を記述してください。また、割り込み処理関数は#pragma interrupt を指定する必要があります。これは vect.h で行われています。vect.h は変更する必要はありません。

```
//intprg.c
// 64 Interrupt IRQ0
void INT_IRQ0(void)
{
    /*ここに処理を記述。*/
}
```

```
// vect.h
// 64 Interrupt IRQ0
#pragma interrupt INT_IRQ0
extern void INT_IRQ0(void);
```

2．ベクタテーブル

これは HEW が生成したものをそのまま使えばよく、編集する必要はありません。ベクタテーブルは vecttbl.c の void *INT_Vectors[]です。SH-1,SH-2 では割り込みが発生するとベクタテーブルに登録された関数のいずれかに実行が遷移します。IRQ0 のベクタ番号はハードウェアマニュアルを参照すると 64 であることが分かります。割り込み要因 IRQ0 の割り込みが発生したとき 64 から 4 引いた INT_Vectors[60]の関数が実行されます。INT_Vectors[60]には INT_IRQ0 という関数が登録されているので割り込み要因 IRQ0 の割り込みが発生したときは INT_IRQ0 が実行されることになります。

```
void *INT_Vectors[] = {
// 4 Illegal code
    (void*) INT_Illegal_code,
...
// 64 Interrupt IRQ0
    (void*) INT_IRQ0,
...
};
```

3．ステータスレジスタの割り込みマスクビットの初期化

割り込み処理を利用するにはステータスレジスタの割り込みマスクビットを適切に初期化する必要があります。resetprg.c の SR_Init の値を 0x000000F0 から適切な値に設定してください。PowerON_Reset_PC 内の set_cr によってステータスレジスタの割り込みマスクビットが設定されます。

```
#define SR_Init    0x000000F0
```

- SH-3,SH-4 の場合

SH-3,SH-4 の場合も割り込み処理を使用するには割り込み処理関数とベクタテーブルとステータスレジスタの割り込みマスクビットの初期化が必要になります。SH-3,SH-4 の場合の例として SH7705 のプロジェクトで割り込み要因 IRQ0 の処理を記述します。

１．割り込み処理関数

HEW が空の割り込み関数を用意していますのでそれを消去してから新しく関数を定義してください。intprg.src に _INT_IRQ0 が定義されていますのでそれを消してください。また、vect.inc に.global INT_IRQ0 という記述があるのでこれも消してください。あとは普通に C 言語で void INT_IRQ0(void)を定義してください。#pragma interrupt を指定する必要はありません。

```
;intprg.src
...
;H'5E0 H-UDI
_INT_H_UDI
;H'600 IRQ0   ;ここを消す
_INT_IRQ0     ;ここを消す
;H'620 IRQ1
_INT_IRQ1
...
```

```
;vect.h
...
;H'5E0 H-UDI
.global        _INT_H_UDI
;H'600 IRQ0                    ;ここを消す
.global        _INT_IRQ0   ;ここを消す
;H'620 IRQ1
.global        _INT_IRQ1
...
```

２．ベクタテーブル

HEW が生成したものをそのまま使えばよく、編集する必要はありません。ベクタテーブルは vecttbl.src の_INT_Vectors です。SH-3,SH-4 では割り込みが発生すると vhandler.src の IRQHandler に遷移します。割り込み事象レジスタの値から割り込み処理ルーチンのアドレスを計算し、割り込み処理ルーチンへ遷移します。ハードウェアマニュアルを参照すると IRQ0 の例外コードは H’600 であることが分かります。H’600 から H’40 を引いて 8 で割ったもの（H’B8）が INT_Vectors からのオフセットになります。INT_Vectors の要素のサイズは４なので、H’B8 割る 4 で 46 番目の INT_Vectors の要素、INT_IRQ０が割り込みルーチンとして呼ばれることになります。IRQHandler の処理は次のようになります。

① 割り込み事象レジスタから例外コードを取得
② INT_Vectors のアドレスを取得
③ 割り込み処理ルーチンのアドレスを計算
④ 割り込みマスクを取得
⑤ 割り込みマスクを ssr に設定
⑥ 割り込み処理ルーチンのアドレスを spc に設定
⑦ rte で割り込み処理ルーチンにジャンプ

```
        .org    H'500
_IRQHandler:
        PUSH_EXP_BASE_REG
;
        mov.l   #INTEVT,r0              ; set event address             —①
        mov.l   @r0,r1                  ; set exception code
        mov.l   #_INT_Vectors,r0        ; set vector table address      —②
        add     #-(h'40),r1             ; exception code - h'40
        shlr2   r1
        shlr    r1
        mov.l   @(r0,r1),r3             ; set interrupt function addr   —③
;
        mov.l   #_INT_MASK,r0           ; interrupt mask table addr
        shlr2   r1
        mov.b   @(r0,r1),r1             ; interrupt mask
        extu.b  r1,r1                                                   —④
;
        stc     sr,r0                   ; save sr
        mov.l   #(RBBLclr&IMASKclr),r2  ; RB,BL,mask clear data
        and     r2,r0                   ; clear mask data
        or      r1,r0                   ; set interrupt mask
        ldc     r0,ssr                  ; set current status            —⑤
;
        ldc.l   r3,spc                                                  —⑥
        mov.l   #__int_term,r0          ; set interrupt terminate
        lds     r0,pr
;
        rte                                                             —⑦
        nop
;
        .pool
        .end
```

3．ステータスレジスタの割り込みマスクビットの初期化

SH-3,SH-4 でも SH-1,SH-2 と同様にステータスレジスタの割り込みマスクビットを適切に初期化する必要があります。resetprg.c の SR_Init を 0x000000F0 から適切な値に設定してください。PowerOn_Reset 内の set_cr によってステータスレジスタの割り込みマスクビットが設定されます。

```
#define SR_Init    0x000000F0
```

11.1.42 プログラムを長時間実行すると一般不当命令例外が発生することがある

■質問

機器を動作後、10分～2時間ほど経過した後に一般不当命令例外が発生し、RESETがかかります。どこで問題が発生しているか解析する方法はありませんか。

■回答

結果的に一般不当命令になっていますが、以下の理由でシステムが暴走して一般不当命令例外となることが考えられます。長時間動かして暴走する場合は、(2)の可能性が高いです。

（1）意図しない割り込みなどが入ってしまっている場合
（2）スタックオーバフローで有効な RAM データを壊す場合
（3）ボード環境がおかしい場合（データ衝突、メモリソフトエラーなど）

発生原因の調査方法として以下の機能を有効にして機器を動作させます。

・命令トレースを有効にする。
・一般不当命令例外時にジャンプする割り込み関数にブレークポイントを設定する。

機器を動作後、一般不当命令例外が発生すると割り込み関数に設定したブレークポイントでストップします。そのときの命令トレースの状態を解析し、原因を特定します。

また、スタックオーバフローが原因の場合の解析方法として以下の機能を動作させます。

・スタック領域の先頭アドレスの直前のアドレスに対して Read/Write ブレークアクセスを設定する。

機器を動作後、スタックをオーバしたアクセスが発生した場合上述のブレークアクセスでストップします。その時のアクセス命令がスタックアクセス命令の場合、原因がスタックオーバフローであることが考えられます。

11.1.43 整数演算結果が期待値と異なる

■質問

整数の乗算結果を long long 型の変数へ代入したいのですがまったく意図しない値になってしまいます。
60000*70000 を 60000*30000 に変更すると正しい値を得ることができます。
long long 型変数への代入なのに乗算結果が int の値を超えると正しくならないのはなぜでしょう？
＜例 1＞

```
long long l_max;
        :
l_max=60000*70000;
```

＜例 2＞

```
long long l_max;
        :
int test=70000;
        :
l_max=60000*test;
```

■回答

代入する変数が long long 型でも、演算する整数を定数で記述すると int 型（4 バイト）で扱われます。
そのため、乗算をした段階で 60000*70000 は 0xFA56EA00 ですが、long long 型に代入する際に符号拡張が起こり 0xFFFFFFFFFA56EA00 となってしまいます。
60000*30000 の場合は 0x6B49D200 となるので符号拡張が起こらないため、0x000000006B49D200 となり期待値が得られています。

意図した演算結果を得るには、定数値の後ろに‘ LL ’を記述し、定数を意図的に long long 型とコンパイラに認識させる必要があります。
<例 1>

```
long long l_max;
        :
l_max = 60000LL * 70000LL;    // 定数の値の後ろに LL を付加、片方の定数のみでも OK
```

<例 2>

```
long long l_max;
        :
int test = 70000;
        :
l_max = 60000LL * test;    // 定数の値の後ろに LL を付加
```

11.2 リンケージエディタ

11.2.1 リンク時に UNDEFINED SYMBOL が出る

■質問

リンク時に"undefined symbol"というメッセージが出ます。なぜですか。
また、どういった意味ですか。

■回答

ライブラリがリンクされているかどうか確認してください。 また、宣言した関数、もしくは使用している関数の実体がありますか。 詳しくは「3.15.2 (2) **リンク時の注意事項**」を参照してください。

11.2.2 リンク時に RELOCATION SIZE OVERFLOW が出る

■質問

リンク時に"RELOCATION SIZE OVERFLOW"のウォーニングメッセージが出てしまいます。
また、セクションの配置アドレスの指定漏れをチェックしたい場合にはどうすればよいですか。

■回答

#pragma abs16, #pragma gbr_base, #pragma gbr_base1 で領域の制限を超えて指定していないか確認してください。

セクションの配置アドレスは、START コマンドによるセクション名指定で行いますが、指定されていないセクションに関しては、指定された最後のセクションの後に配置されるという仕様になっています。
特に、セクション名が多くなった場合、この指定の消し忘れによるプログラミングミスが考えられます。
このコマンドは、START コマンドで指定のないセクションが存在すると警告を出力します。

(1) メッセージ例

以下にメッセージが出力されたリンケージエディタのオプション例を示します。

```
input sample.obj
input low/__main.obj
input low/__exit.obj
library lib/shclib.lib
library       low/shclow.lib
output sample.abs
form a
entry _$main
start C,B,D,P/0400           ($G0 と$G1 の指定が抜けている)
;start C,B,D,$G0,$G1,P/0400  (正常終了時のパラメータ指定 )

** L1120 (W) Section address is not assigned to "$G0"
** L1120 (W) Section address is not assigned to "$G1"

LINKAGE EDITOR COMPLETED?
```

$G0 と$G1 のセクション名が定義されていないため
ウォーニングメッセージが出力される

11.2.3 リンク時に SECTION ATTRIBUTE MISMATCH が出る

■質問

リンク時に"SECTION ATTRIBUTE MISMATCH"のウォーニングメッセージが出てしまいます。
どうすればよいですか。

■回答

このエラーには以下の原因が考えられます。

（1） 同一セクションで異なるアラインメントを指定している
同じセクション名で違うアラインメントをしていないか確認してください。

（2） "-cpu=sh4"オプションでコンパイルしたオブジェクトとそれ以外のcpuオプションでコンパイルしたオブジェクトをリンクしようとした場合
"-cpu=sh4"オプション*でコンパイルすると無条件に各セクションがaligndata8となってしまいます。そのため他のcpuオプションでコンパイルしたオブジェクトとアラインメントが異なることになります。この場合も同様にリンケージエディタのALIGN_SECTIONオプション／サブコマンドで回避できます。

（3） 「11.2.4 **プログラムの**RAM**への転送実行　■回答**2」に示される変更方法において、変更内容が以下のすべての条件を満たす場合。
なお、この場合はウォーニングを無視して頂いて問題ありません。
(a) C/C++コンパイラのsectionオプションなどでプログラムセクション(P)を別名に変更した場合
(b) (a)のセクションを転送元のセクションに指定した場合

【注】* "-cpu=sh4"**オプションでコンパイルすると無条件に各セクションが** aligndata8 **となっています**(Ver.5 **以前)。セクションが** 8 **バイト境界におかれるためセクション間のメモリが増えることがありますのでご注意ください。**

11.2.4 プログラムの RAM への転送実行

■質問

プログラムを実行速度の速い RAM に置きたいがどうすればよいですか。

＜動作環境＞

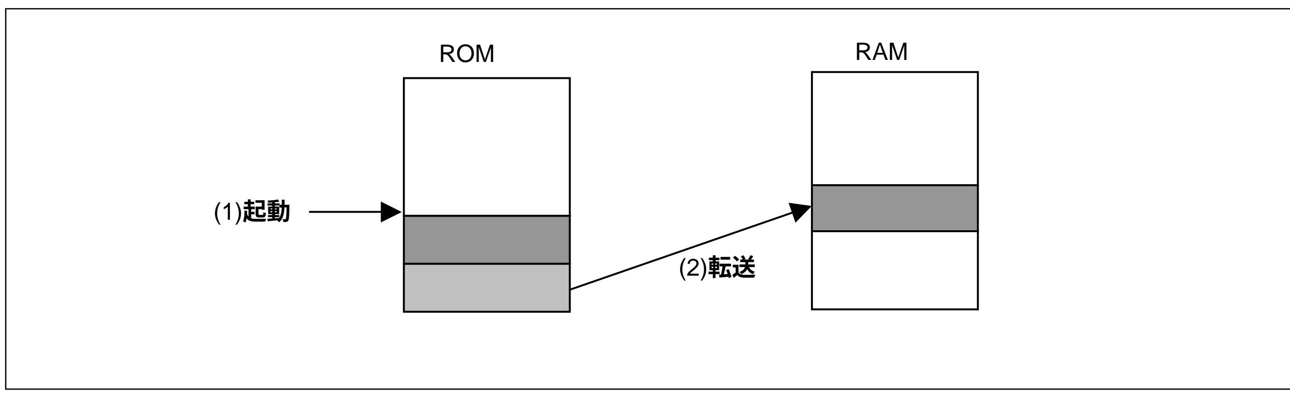


図 11.4 ROM から RAM へのプログラム転送

＜詳細内容＞

（1）ROMに常駐するプログラムを起動する。
（2）自プログラムコードの一部のセクションをRAMに転送する。

■回答１

RAM 上の固定番地に必ずプログラムコードをコピーする場合は、初期化データ同様、リンカの ROM 化支援機能を使用することにより、RAM 上でプログラムを実行することができます（リンク時にアドレス解決するため、実行時に RAM 上のアドレスを決定し、プログラムコードをコピーすることはできません）。

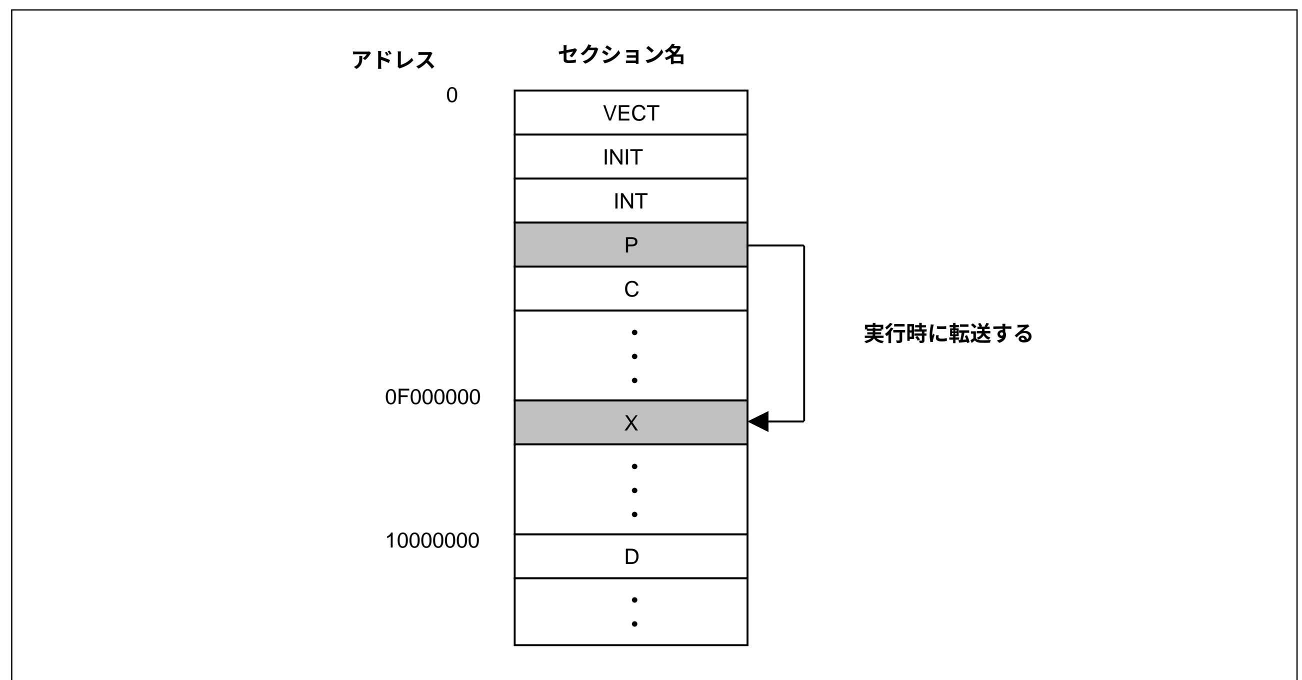


図 11.5 セクション構成の例

図 11.5 のようなセクション構成のプログラム例を以下に示します。

C 言語部分

```
/*****************************************************************/
/*                  file name "init.c"                           */
/*---------------------------------------------------------------*/
/*     コンパイルオプションでプログラムセクション名を"INIT"する           */
/*****************************************************************/
#include "sample.h"          /* 2章の sample.h を include してください */
extern int *_B_BGN,*_B_END;
extern int *_P_BGN;          /* P セクションの先頭アドレス            */
extern int *_X_BGN;          /* X セクションの先頭アドレス            */
extern int *_X_END;          /* X セクションの最終アドレス            */
extern void _INITSCT(void);
extern void _INIT();
extern void main();

void _INIT()
{
       _INITSCT();
       main();
       for ( ; ; )
           ;
}

void _INITSCT(void)
{
       int *p,*q;

       for ( p = _B_BGN; p < _B_END; p++ )
           *p = 0;

       /* P セクションから X セクションへのコピー */
       for ( p = _X_BGN, q = _P_BGN; p < _X_END; p++, q++ )
           *p = *q;
}


/***************************************************************/
/*                  file name "main.c"                         */
/*-------------------------------------------------------------*/
/*          プログラムセクション名はデフォルトの"P"とする            */
/***************************************************************/
int a = 1;
int b;
const int c = 100;
void main(void)
{
       /* このルーチンはコピー先(RAM)で実行される */
       for ( ; ; )
           ;
}


/*****************************************************************/
/*                    file name "int.c"                          */
/*****************************************************************/
#include "sample.h"         /* 2章の sample.h を include してください  */
#include "7032.h"           /* 2章の 7032.h を include してください    */
extern int a;               /* section D code                        */
extern int b;               /* section B code                        */
extern const int c;         /* section C code                        */
#pragma interrupt(IRQ0, inv_inst)


/*****************************************************************/
/*                  interrupt module IRQ0                        */
/*****************************************************************/
extern void IRQ0(void)
```

```
{
       a = PB.DR.WORD;
       PC.DR.BYTE = c;
}

/*****************************************************************/
/*                 interrupt module inv_inst                     */
/*****************************************************************/
extern void inv_inst(void)
{
       return;
}
```

アセンブリ言語部分

```
;***************************************************************
;*                 file name "sct.src"                         *
;***************************************************************
                .SECTION      P,CODE,ALIGN=4
                .SECTION      X,CODE,ALIGN=4
                .SECTION      B,DATA,ALIGN=4
                .SECTION      C,DATA,ALIGN=4

__P_BGN:    .DATA.L (STARTOF P)             ;P セクションの先頭アドレス
__X_BGN:    .DATA.L (STARTOF X)             ;P セクションの RAM 上での先頭アドレス
__X_END:    .DATA.L (STARTOF X)+(SIZEOF X)  ;P セクションの RAM 上での最終アドレス
__B_BGN:    .DATA.L (STARTOF B)             ;BBS セクションの先頭アドレス
__B_END:    .DATA.L (STARTOF B)+(SIZEOF B)  ;BBS セクションの最終アドレス

                .EXPORT __P_BGN
                .EXPORT __X_BGN
                .EXPORT __X_END
                .EXPORT __B_BGN
                .EXPORT __B_END
                .END

;***************************************************************
;*                 file name "vect.src"                        *
;***************************************************************
       .SECTION       VECT,DATA,ALIGN=4

       .IMPORT __INIT
       .IMPORT _inv_inst
       .IMPORT _IRQ0

       .DATA.L __INIT
       .DATA.L H'FFFFFFC
       .ORG    H'0080
       .DATA.L _inv_inst
       .ORG    H'0100
       .DATA.L _IRQ0
       .END
```


RENESAS

コマンドラインで次のようにします。

コマンド指定

```
shc△-debug△-section=P=INIT△init.c
shc△-debug△-section=P=INT△int.c
shc△-debug△main.c
asmsh△sct.src△-debug
asmsh△vect.src△-debug
optlnk△-nooptimize△-sub=rom.sub
```

リンカオプションファイル

```
;*****************************************************************
;*                 file name "rom.sub"                           *
;*****************************************************************
sdebug
input vect, sct, init, int, main
ROM P=X             ; P セクションが X に割り付いたようにアドレス解決する
start VECT/0,INIT,INT,P,C,D/10000000,X/0f000000
                      ; VECT,INIT,INT,P,C,D は ROM 上に、X は RAM 上に配置する
output sample.abs
list sample.map
exit
```

上記のようにプログラムするとセクション P のプログラムをセクション X にコピーし実行します。セクション INIT はコピーするルーチンであるためコピーされるルーチンとは別のセクションでなければなりません。これで main プログラム（セクション P）がコピー先で実行されます。

■回答２

HEW2.0 以降ではより簡単に最適化リンケージエディタの ROM 化支援機能を用いて、実行時に RAM 上の固定番地（リンク時に決定）にプログラムの 1 部のセクションをコピーし RAM 上でプログラムを実行することができます。

まず、起動時に RAM 上で実行したいプログラムのセクションを転送するために、セクションのアドレスを指定します。この処理は HEW が生成したファイル dbsct.c に追加します。いま PXX セクションにあるコードを XX セクションに転送するとします。以下のように追記します。

```
#pragma section $DSEC
static const struct {
        char *rom_s;      /* 初期化データセクションのROM 上の先頭アドレス */
        char *rom_e;      /* 初期化データセクションのROM 上の最終アドレス */
        char *ram_s;      /* 初期化データセクションのRAM 上の先頭アドレス */
}DTBL[]= {{__sectop("D"), __secend("D"), __sectop("R")},
         {__sectop("PXX"),__secend("PXX"),__sectop("XX")}};
#pragma section $BSEC
static const struct {
        char *b_s;        /* 未初期化データセクションの先頭アドレス */
        char *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")};
```

↑PXXセクションおよびXXセクションに対する設定

この処理をすると起動時に PXX セクションから XX セクションへのコピーが行われます。

その後、最適化リンケージエディタで転送先セクション XX の先頭アドレスを指定します。

[ビルド->SuperH RISC engine Standard Toolchain...->最適化リンカ]のカテゴリ・セクションを選択し編集ボタンをクリックするとセクション設定のダイアログボックスが表示されます。

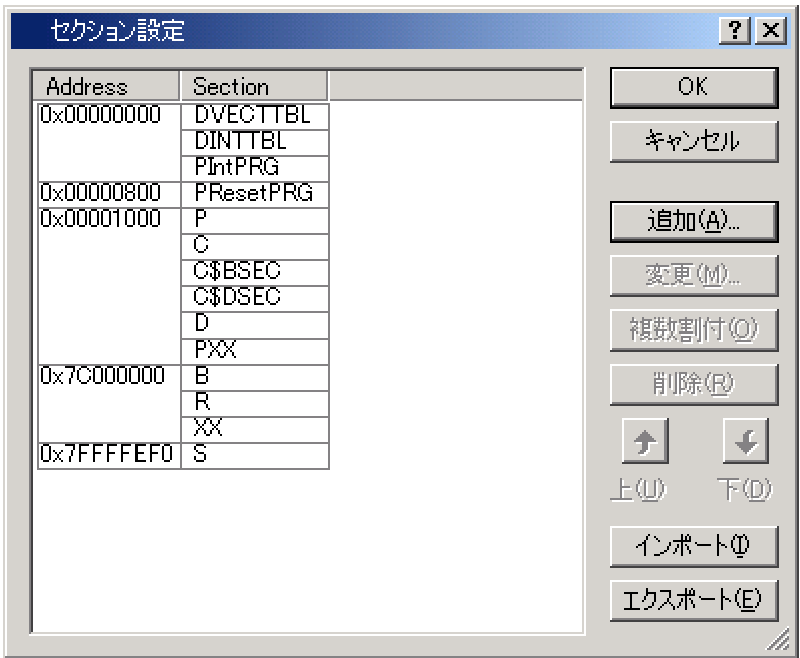


図 11.6　**セクション設定ダイアログボックス**

ここで PXX セクションと XX セクションを設定します。

また[ビルド->SuperH RISC engine Standard Toolchain->最適化リンカ]を選択し、カテゴリ・出力、オプション項目・ROM から RAM へマッピングするセクションを選び、PXX から XX へマップするよう設定します。

以上で RAM 上でプログラムを実行できるようになります。

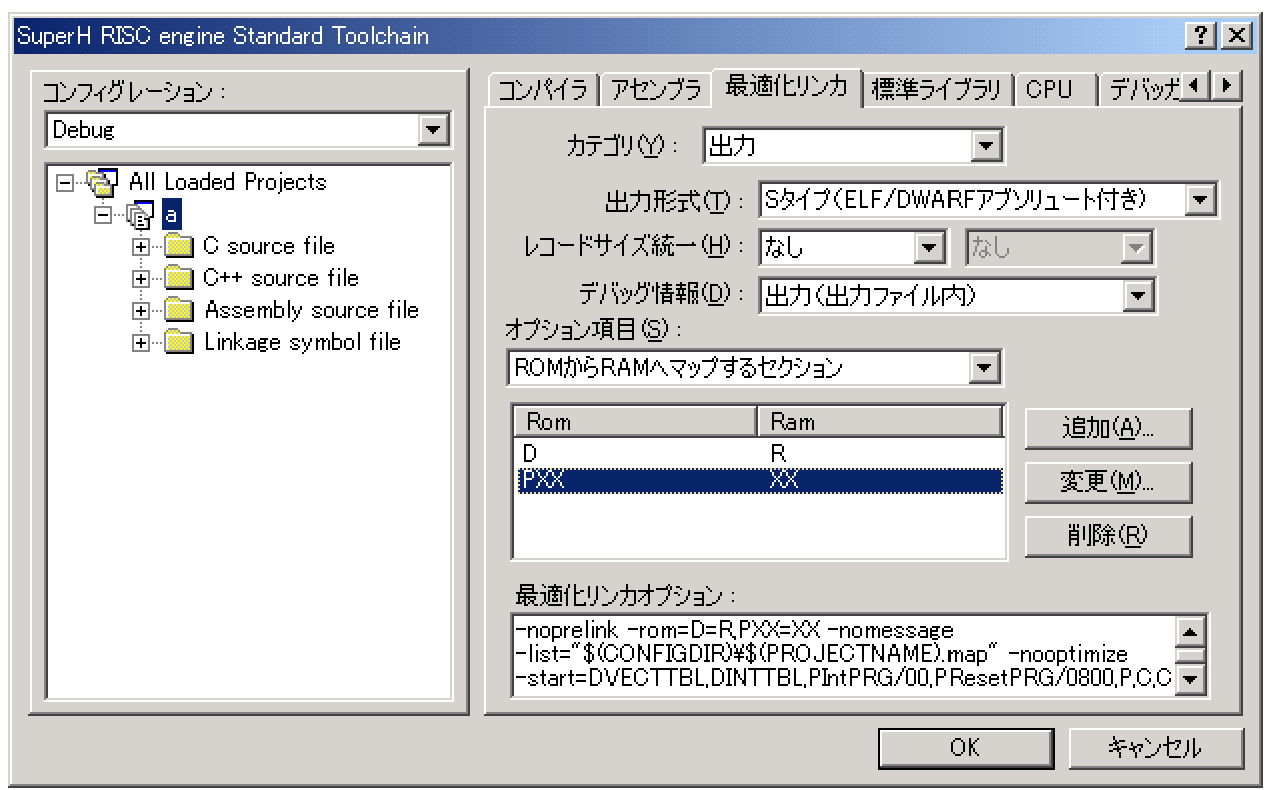


図 11.7　**最適化リンカダイアログボックス**

RENESAS

■備考および注意事項

以上の処理を行う際、HEW1.2 のモジュール間最適化ツールでウォーニングメッセージ（1300 SECTION ATTRIBUTE MISMATCH IN ROM OPTION/SUBCOMMAND(XX)）が出力される場合があります。

これは、__sectop、__secend 演算子でプログラムセクションを指定したために出力されます。本ウォーニングは無視できます。

HEW2.0 以降では改善され通常はメッセージは、出力されなくなりましたが、以下の場合は HEW1.2 と同様にウォーニングメッセージ(L1323 (W) Section attribute mismatch : "FXX")が出力される場合があります。この場合も問題ありません。

(1) C/C++コンパイラの section オプションなどでプログラムセクション(P)を別名に変更。

(2) (1)のセクションを転送元のセクションにする。

11.2.5　一部のアドレス領域のシンボルアドレスを FIX してリンクしたい

■質問

内蔵 ROM プログラムを FIX した後に、外部メモリプログラムを開発し、今後外部メモリプログラムだけをアップデートしていきたい。

■回答

内蔵 ROM プログラム FIX の際、リンクコマンド fsymbol を使用して、内蔵 ROM の外部定義ラベルの定義ファイルを出力してください。

定義ファイルは、アセンブラの EQU 文で作成されているため、外部メモリプログラムの作成時に、このファイルをアセンブルしたものを入力すれば ROM 上の固定のアドレスを参照するプログラムになります。

■使用例

図 11.8 は、製品 A の機能 A を機能 B に変更し、製品 B を開発する例です。本機能を用いて、共通 ROM 内シンボルのアドレスを解決することにより、共通 ROM が流用できます。

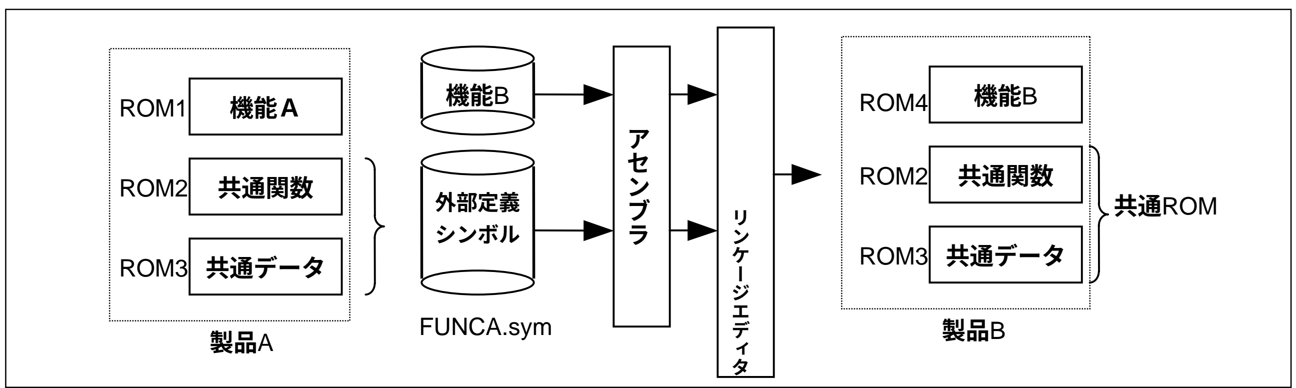


図 11.8　シンボルアドレス出力機能の使用例

【外部定義シンボルファイル出力の指定例】

```
optlnk△ROM1,ROM2,ROM3△-output=FUNCA△-fsymbol=sct2,sct3
```

sct2とsct3の外部定義シンボルをファイルに出力します。

【ファイル(FUNCA.sym)の出力例】

```
;H SERIES LINKAGE EDITOR GENERATED FILE   1997.10.10
;fsymbol = sct2, sct3

;SECTION NAME = sct1
.export sym1
sym1:  .equ    h'00FF0080
.export sym2
sym2:  .equ    h'00FF0100
;SECTION NAME = sct2
.export sym3
sym3:  .equ    h'00FF0180
 .end
```

【アセンブル、再リンクの指定例】

```
asmsh△ROM4
asmsh△FUNCA.sym
optlnk△ROM4,FUNCA
```

ROM2,ROM3のオブジェクトファイルをリンクすることなく、ROM4の外部参照シンボルを解決します。

【注】 本機能を使用する場合、共通関数から機能A内シンボルは参照できません。

11.2.6 オーバレイの実現

■質問

オーバレイを実現したい。

実行時に、あるプログラムを ROM から、RAM に転送実行したいのだが、同時に実行されない、2 つ以上のルーチンを、同一の RAM アドレスで実行したい。

■回答

ROM から、RAM への転送実行は、**「11.2.4 プログラムの RAM への転送実行」**を参照してください。

基本的なプログラムはこのとおりですが、以下のことがこのほかに必要になります。

・指定例

本機能を用いて、同時に存在しない複数のプログラム／データを外部 ROM から高速な内部 RAM に転送して実行する例を示します。

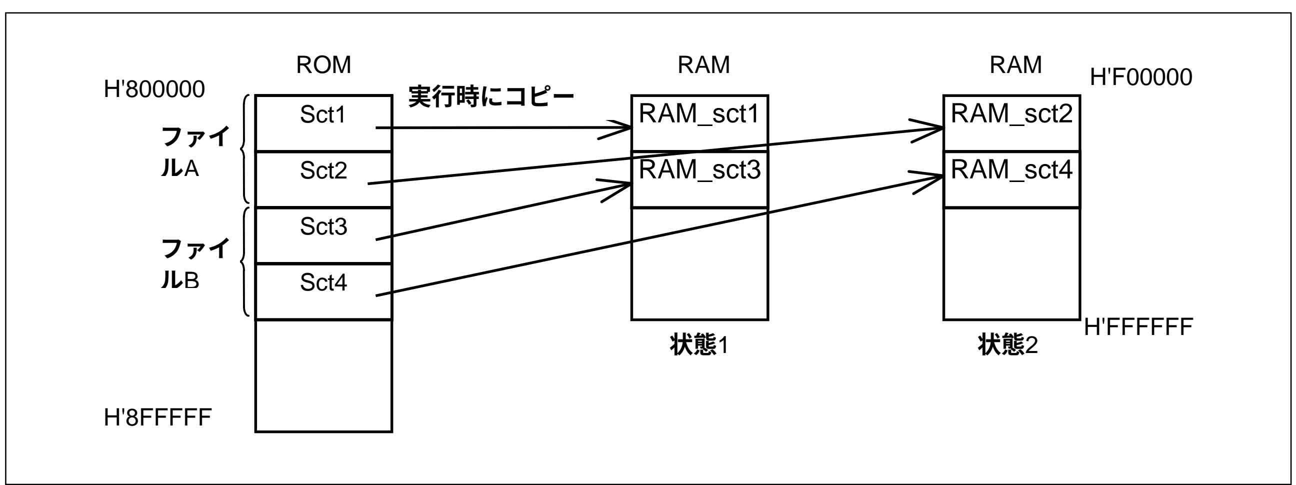


図 11.9 同一アドレスへの複数セクション割り付け

【コマンド指定例】

```
optlnk△-subcommand=test.sub
```

【test.sub の内容】

```
INPUT  A,B
ROM    Sct1=RAM_sct1
ROM    Sct3=RAM_sct3
ROM    Sct2=RAM_sct2
ROM    Sct4=RAM_sct4
START  Sct1,Sct2,Sct3,Sct4/800000
START  RAM_sct1,RAM_sct3:RAM_sct2,RAM_sct4/0F00000
```

【説明】

RAM_sct1 と RAM_sct2 を同一アドレスから割り付けます。RAM_sct3 は RAM_sct1 に、RAM_sct4 は RAM_sct2 におのおの連結して割り付けます。

11.2.7 未定義シンボルのエラー出力指定

■質問

リンク時に未定義シンボルがある場合にエラーメッセージを出力し、ロードモジュール出力を抑止したい。

■回答

リンク時に UDFCHECK オプションを指定してください。

これにより、未定義シンボルが含まれる際にはエラーメッセージ 221 を出力して、ロードモジュール出力を抑止します。

（UDFCHECK オプション／サブコマンド指定がない場合はウォーニングメッセージ 105 を表示し、ロードモジュールは生成されます。）

ただし、リンケージエディタ Ver7 以降では UDFCHECK オプションは廃止されており、常に有効になっています。

11.2.8 S タイプファイルの出力形式の統一

■質問

S タイプファイルの出力形式が S1,S2,S3 混在しているが、統一したい。

■回答

record オプションでロードアドレスに関係なく一定のデータレコード(S1,S2,S3)で出力することができます。

例）optlnk test.abs -form=stype -output=test.mot -record=s2　;すべてのデータレコードを S2 で出力します。

11.2.9 出力ファイルの分割

■質問

出力ファイルを ROM にあわせて複数ファイルに分割したい。

■回答

出力ファイル名の後に開始アドレスと終了アドレスを指定すると、指定範囲のオブジェクトをファイルに出力することができます。出力ファイル名は複数指定することが可能です。

例） optlnk test.abs -form=stype -output=test1.mot=0-FFFF test2.mot=10000-1FFFF　;test1.motに0x0～0xFFFFの範囲のオブジェクトを、test2.motに0x10000～0x1FFFFの範囲のオブジェクトをそれぞれ出力します。

11.2.10 Windows2000 での optlnksh.exe の実行

■質問

Windows2000 にて optlnksh.exe を実行すると、「2020 SYNTAX ERROR」が出力される。

■回答

環境変数 SHC_TMP に、スペースが入っていないかご確認いただけますようお願い申し上げます。

shc では SHC_TMP にスペースが入っていても正しく動作しますが、optlnksh ではエラー（2020 SYNTAX ERROR）になります。ちなみに、Windows2000 でデフォルトのテンポラリディレクトリは C:¥Documents and Settings¥foo¥Local Settings¥Temp のようになります（foo はユーザ名）。

11.2.11 最適化リンケージエディタが出力するファイル形式

■質問

プログラマで使用可能なロードモジュールファイル形式を教えてください。

■回答

最適化リンケージエディタが出力するロードモジュールを以下に示します。

・プログラマ用のロードモジュールを作成する場合、Hexdecimal形式またはSType形式のフォーマットで出力してください。この場合、デバッグ情報は出力されません。

・ C/C++コンパイラV7.1,V8.0に対応した最適化リンケージエディタは、デバッグ時、ELF/DWARF2フォーマット形式のロードモジュールを出力します。旧バージョンで作成したロードモジュールはSYSROFフォーマットまたはELF/DWARF1フォーマットのため、新バージョンで使用する際はELF/DWARFフォーマットコーンバータを利用しフォーマットを変更してください。

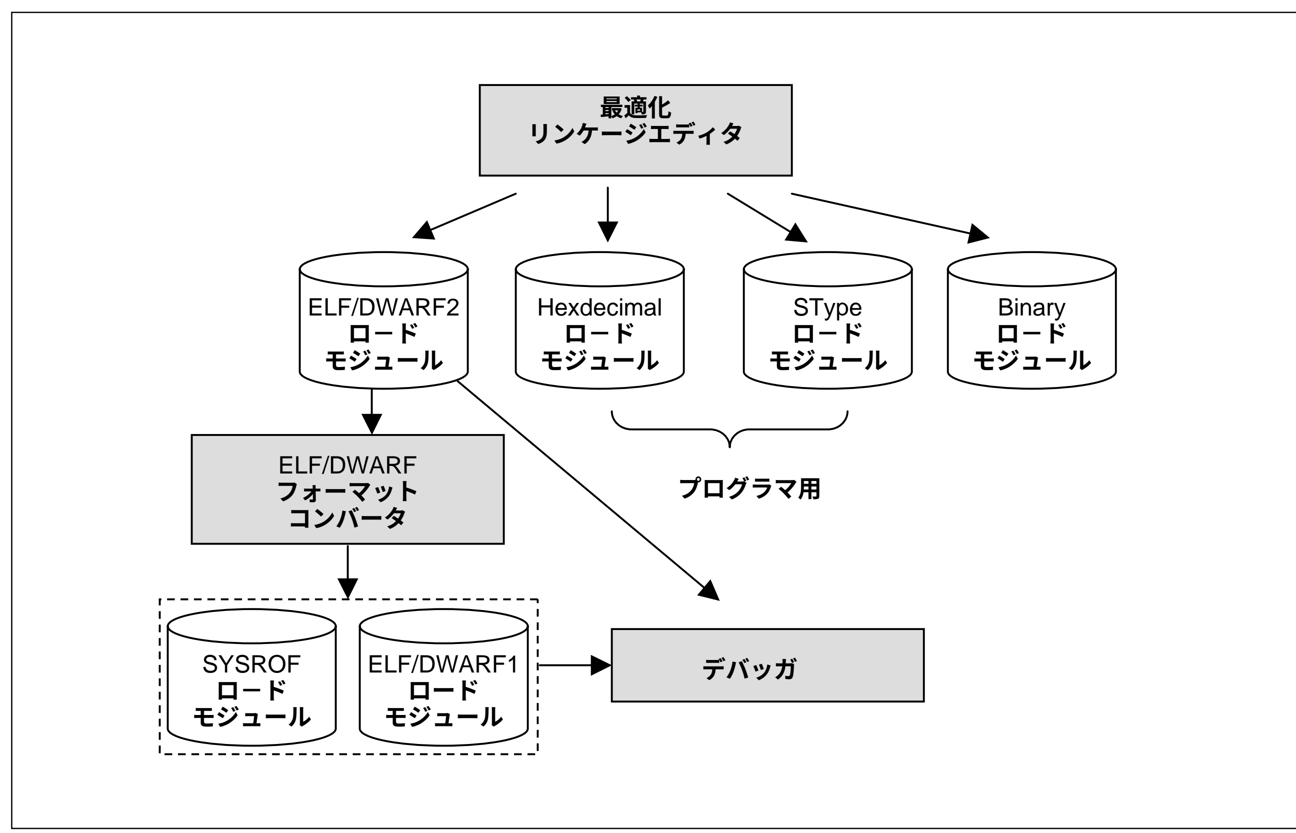


図 11.10 最適化リンケージエディタ出力ロードモジュール

RENESAS

11.2.12 プログラムサイズ（ROM,RAM）サイズの算出方法

■質問

ROM,RAM 容量を正確に測りたいのですが、方法が分かりません。

■回答

最適化リンケージエディタが出力するリストファイルで確認することができます。

■指定方法

ダイアログメニュー：**最適化リンカタブカテゴリ:[リスト] リンケージリスト出力**

コマンドライン　　：*-list=<ファイル名>*

■確認方法

本オプションを指定することにより、以下のリストファイル（*.map）を出力することができます。

この例の場合、コード属性のセクションは DVECTTBL,DINTTBL,PIntPRG,PResetPRG,P,C$BSEC,C$DSEC,D なので ROM サイズは 0x000006a8 となります。

RAM 容量は B,R,S なので 0x0000052c となります。

```
*** Mapping List ***

SECTION                         START      END         SIZE      ALIGN

DVECTTBL
                              00000000  0000000f        10           4
DINTTBL
                              00000010  000003ff       3f0           4
PIntPRG
                              00000400  00000557       158           4
PResetPRG
                              00000800  00000833        34           4
P
                              00001000  000010db        dc           4
C$BSEC
                              000010dc  000010e3         8           4
C$DSEC
                              000010e4  000010ef         c           4
D
                              000010f0  0000111b        2c           4
B
                              7c000000  7c0003ff       400           4
R
                              7c000400  7c00042b        2c           4
S
                              7c000500  7c0005ff       100           4
```

11.2.13 Section alignment mismatch が出力される

■質問

以下のようにバイナリファイルを入力し、バイナリファイルのセクション名をセクションアドレス演算子で参照すると L1322 の Warning が出力されてしまいます。どのように回避すればよいのでしょうか？

[オプション指定]

```
binary=project.bin(BIN_SECTION)
```

[C/C++プログラム]

```
void main(void)
{
    unsigned char *s_ptr;
    s_ptr = __sectop("BIN_SECTION");
    dummy(s_ptr);
}
```

■回答

セクションアドレス演算子（__sectop,__secend）を使用すると、以下のようにコンパイラが生成するコードに、当該のセクションに関するサイズが 0 で境界調整数が 4 のセクションが形成されます。

本ケースの場合、バイナリのセクションを入力していますが、バイナリセクション実体の境界調整数は 1 なので、同名セクションで境界調整数が混在し L1322 の Warning メッセージが出力されています。

しかし、本 Warning メッセージが出力されてもプログラムの動作に影響はありません。

本 Warning メッセージは、最適化リンカでバイナリファイルを入力する際、境界調整数を指定することで回避できます。

[__sectop 使用箇所のコード]

```
_main:                          ; function: main
                                ; frame size=0
        .STACK     _main=0
        MOV.L      L13+2,R4     ; STARTOF BIN_SECTION
        BRA        _dummy
        NOP
・・・
        .SECTION   BIN_SECTION,DATA,ALIGN=4        ;←サイズが0で境界調整数が4のセクション
        .END
```

■回避方法例

ダイアログメニュー：**最適化リンカタブカテゴリ:[入力] オプション項目:バイナリファイル**

コマンドライン　　：`binary=binary_data.bin(BIN_SECTION:4)`

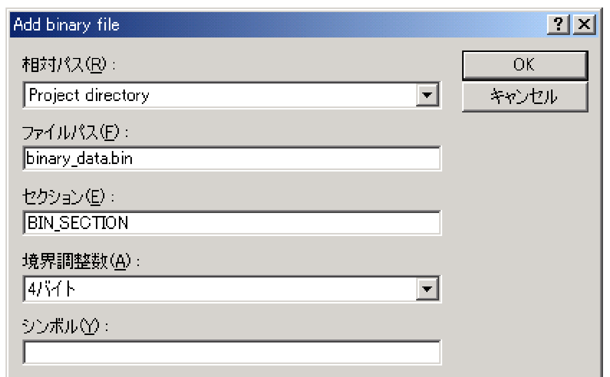


図 11.11 Add binary file ダイアログボックス

■備考

バイナリファイル入力時の境界調整数指定はリンケージエディタ Ver.9 から対応しています。

詳細は「9.1.1(4) **バイナリファイル**」を参照してください。

11.3　標準ライブラリ

11.3.1　リエントラントと標準ライブラリ

■質問

関数をリエントラントにする場合の注意点を教えてください。

■回答

大域変数を使用する関数はリエントラントではなくなります。

また、リエントラントな関数を作ったつもりでも、 以下の標準インクルードファイルを用いて標準ライブラリを使用すると、大域変数が使われているのでリエントラントではなくなります。

以下にリエントラントライブラリ一覧表を掲載します。 表中、△で示した関数は、_errno 変数を設定しますので 、プログラム中で_errno を参照していなければリエントラントに実行できます。また標準ライブラリをリエントラント化することもできます。「11.3.2 **標準ライブラリで、リエントラントライブラリを使用したい**」に標準ライブラリをリエントラント化する方法を載せています。併せてご参照ください。

表 11.14　**リエントラントライブラリ一覧（**1**）**

リエントラント欄　○:**リエントラント**　×:**ノンリエントラント**　△:_errno **を設定**

No.	標準インクルードファイル	関数名		リエントラント
1	stddef.h	1	offsetof	○
2	assert.h	2	assert	×
3	ctype.h	3	isalnum	○
		4	isalpha	○
		5	iscntrl	○
		6	isdigit	○
		7	isgraph	○
		8	islower	○
		9	isprint	○
		10	ispunct	○
		11	isspace	○
		12	isupper	○
		13	isxdigit	○
		14	tolower	○
		15	toupper	○

No.	標準インクルードファイル	関数名		リエントラント
4	math.h	16	acos	△
		17	asin	△
		18	atan	△
		19	atan2	△
		20	cos	△
		21	sin	△
		22	tan	△
		23	cosh	△
		24	sinh	△
		25	tanh	△
		26	exp	△
		27	frexp	△
		28	ldexp	△
		29	log	△
		30	log10	△

表 11.14 リエントラントライブラリ一覧（2）

No.	標準インクルードファイル	関数名		リエントラント
4	math.h	31	modf	△
		32	pow	△
		33	sqrt	△
		34	ceil	△
		35	fabs	△
		36	floor	△
		37	fmod	△
5	setjmp.h	38	setjmp	○
		39	longjmp	○
6	stdarg.h	40	va_start	○
		41	va_arg	○
		42	va_end	○
7	stdio.h	43	fclose	×
		44	fflush	×
		45	fopen	×
		46	freopen	×
		47	setbuf	×
		48	setvbuf	×
		49	fprintf	×
		50	fscanf	×
		51	printf	×
		52	scanf	×
		53	sprintf	△
		54	sscanf	△
		55	vfprintf	×
		56	vprintf	×
		57	vsprintf	△
		58	fgetc	×
		59	fgets	×
		60	fputc	×

No.	標準インクルードファイル	関数名		リエントラント
7	stdio.h	61	fputs	×
		62	getc	×
		63	getchar	×
		64	gets	×
		65	putc	×
		66	putchar	×
		67	puts	×
		68	ungetc	×
		69	fread	×
		70	fwrite	×
		71	fseek	×
		72	ftell	×
		73	rewind	×
		74	clearerr	×
		75	feof	×
		76	ferror	×
		77	perror	×
8	stdlib.h	78	atof	△
		79	atoi	△
		80	atol	△
		81	strtod	△
		82	strtol	△
		83	rand	×
		84	srand	×
		85	calloc	×
		86	free	×
		87	malloc	×
		88	realloc	×
		89	bsearch	○
		90	qsort	○

表 11.14 リエントラントライブラリ一覧（3）

No.	標準インクルードファイル	関数名		リエントラント	No.	標準インクルードファイル	関数名		リエントラント
8	stdlib.h	91	abs	○	9	string.h	103	memchr	○
		92	div	△			104	strchr	○
		93	labs	○			105	strcspn	○
		94	ldiv	△			106	strpbrk	○
9	string.h	95	memcpy	○			107	strrchr	○
		96	strcpy	○			108	strspn	○
		97	strncpy	○			109	strstr	○
		98	strcat	○			110	strtok	×
		99	strncat	○			111	memset	○
		100	memcmp	○			112	strerror	○
		101	strcmp	○			113	strlen	○
		102	strncmp	○			114	memmove	○

11.3.2 標準ライブラリで、リエントラントライブラリを使用したい

■質問

標準ライブラリで、リエントラントなライブラリを使用したい。

■回答

項番 11.3.1 にリエントラント関数一覧があります。なお、SHC V7.0 以降、標準ライブラリ構築ツールの設定で、リエントラントな関数を生成することができます。

コマンドラインでは lbgsh -reent オプションを指定してください。

また、HEW での設定方法は**図** 11.12 のとおりです。

図 11.12 Standard Library **ダイアログボックス**

11.3.3 標準ライブラリが存在しない（SHC V6 以降）

■質問

SHC V5 以前で付属されていた標準ライブラリがありません。（SHC V6 以降）

■回答

SHC V6 以降は、標準ライブラリに対してもユーザによりオプション指定が可能な仕様としました。これにより、標準ライブラリに対しても、オプションによるチューニングが可能です。SHC V6 以降では、製品に標準ライブラリを添付しておらず、標準ライブラリ構築ツールを使用して標準ライブラリを生成してください。

11.3.4 標準ライブラリ構築時のウォーニング

■質問

標準ライブラリ構築時、「L1200(W) Bucked up file "a.lib" into "b.lbk"」が出力される。

■回答

ライブラリ生成時、ライブラリのバックアップを取るという意味のメッセージで、特に問題にはなりません。

HEW/[オプション]/[SuperH RISC engine Standard Toolchain]の[標準ライブラリ]モード：で"標準ライブラリファイル作成（オプション変更時）"を選択していただければウォーニングは出力されなくなります。HEW は、「すべてをビルド」をすると、まずリンカは、標準ライブラリを自動生成します。初めて作成したプロジェクトの場合は、標準ライブラリを生成する必要があるので、HEW/[オプション]/[SuperH RISC engine Standard Toolchain.]の[標準ライブラリ]モード：で「標準ライブラリファイル作成」を選択する必要がありますが、一度「すべてをビルド」したファイルには、すでに標準ライブラリが生成されているため、新たに標準ライブラリを自動生成する必要がありません。今回のケースでは、「すべてをビルド」のたびに標準ライブラリを自動生成を生成しているため、既存のライブラリのバックアップをとるという操作をしてしまっています。

"標準ライブラリファイル作成（オプション変更時）"を選択いただければ本ウォーニングの回避ができます。またこれにより「すべてをビルド」時に標準ライブラリを自動生成する時間を省くことができます。

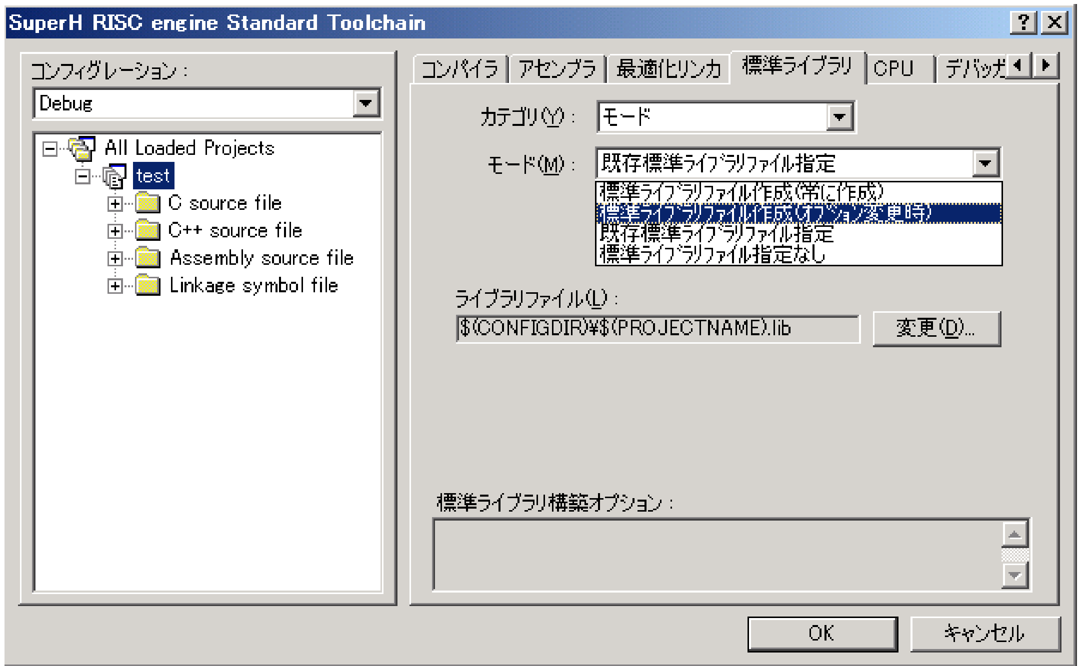


図 11.13 標準ライブラリダイアログボックス

RENESAS

11.3.5 ヒープ領域で使用するメモリのサイズ

■質問

ヒープ領域で使用するメモリサイズの算出方法を教えてください。

■回答

ヒープ領域で使用するメモリ領域のサイズは、C/C++プログラム内でメモリ管理ライブラリ関数(calloc,malloc,realloc,new 関数)によって割り付ける領域の合計です。ただし、メモリ管理ライブラリ関数は、1 回の呼び出しのたびに管理用の領域として 4 バイト使用します。実際に確保する領域サイズにこの管理領域のサイズを加えて計算してください。

また、コンパイラは、ヒープ領域を 1024 バイト単位で管理しています。ヒープ領域として確保する領域サイズ(HEAPSIZE)は次のように計算してください。

HEAPSIZE = 1024 × n (n≧1) (メモリ管理ライブラリによって割り付ける領域サイズ) + 管理領域サイズ≦HEAPSIZE 入出力ライブラリ関数は、内部処理の中でメモリ管理ライブラリ関数を使用しています。入出力の中で割り付ける領域のサイズは、516 バイト×(同時にオープンするファイルの数の最大値)になります。

【注】 メモリ管理ライブラリ関数の free、または delete 関数で解放した領域は、再びメモリ管理ライブラリ関数で領域を確保するときに再利用しますが、割り付けを繰り返すことによって空き領域の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。このような状況を避けるために、以下の注意に従ってヒープ領域を使用してください。

(ア) サイズの大きな領域は、なるべくプログラムの実行開始直後に確保してください。

(イ) 解放して再利用するデータ領域のサイズをなるべく一定にしてください。

RENESAS

11.3.6　ライブラリファイルを編集したい

■質問

既存のライブラリファイルを生かすために、ライブラリファイルを編集したいのですが、方法が分かりません。

■回答

最適化リンケージエディタのオプションにより編集が可能です。以下で各編集機能を説明します。
また、最適化リンケージエディタを GUI から動作させる H Series Librarian Interface も用意されています。

■H Series Librarian Interface 起動方法

HEW の[ツール-> H Series Librarian Interface]を選択し、H Series Librarian Interface を起動します。

(A) ライブラリ内モジュールのセクション名変更

ライブラリの特定モジュールについて、セクション名称を変更し特定のアドレスへセクションを配置することができます。

(1)　当該のライブラリを開き、特定のアドレスに割り付けたいモジュールを選択。
(2)　[Action->Rename Section...]で以下のダイアログを表示し、Afterボタンによりセクション名を変更する。

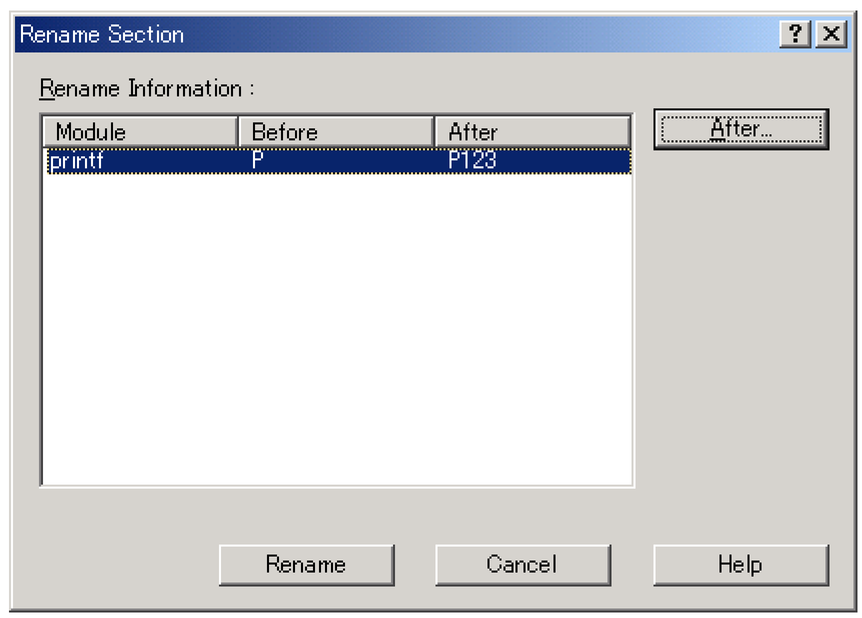


図 11.14 Rename Section ダイアログボックス

【コマンドラインの場合】
optlnk –form=lib –lib=<ライブラリファイル名> -rename=<ライブラリ内モジュール名>(P=P123)

(B) ライブラリ内モジュールの置換／ライブラリへのモジュール追加

ライブラリのモジュールを置換することができます。また、新たにモジュールを追加することも可能です。

(1)　当該のライブラリを開き、 [Action->Add/Replace...]を選択。
(2)　置換するべき同名のモジュールを開く。同名でないモジュールを開くとモジュールの追加になります。

【コマンドラインの場合】
optlnk –form=lib –lib=<ライブラリファイル名> -replace=<ライブラリ内モジュール名>

(C) ライブラリ内モジュールの削除

ライブラリのモジュールを削除することができます。

(1) 当該のライブラリを開き、削除したいモジュールを選択(複数選択可)。
(2) [Action->Delete...]でDeleteダイアログを表示し、Deleteボタンを押下します。

【コマンドラインの場合】
optlnk –form=lib –lib=<ライブラリファイル名> -delete=<ライブラリ内モジュール名>

(D) ライブラリ内モジュールの抽出

ライブラリのモジュールを抽出することができます。
(1) 当該のライブラリを開き、抽出したいモジュールを選択(複数選択可)。
(2) [Action->Extract...]で以下のダイアログを表示し、出力先を設定した後にOKボタンを押下します。
(3) 設定した出力先にモジュールが出力されます。(下記例ではC:¥)

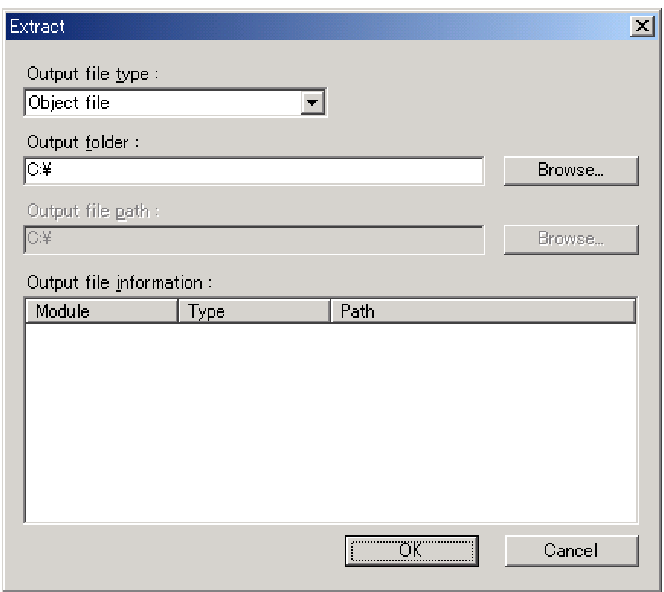


図 11.15 Extract ダイアログボックス

【コマンドラインの場合】
optlnk –lib=<ライブラリファイル名> -extract=<ライブラリ内モジュール名> -form=<出力ファイル形式>
【注】 本例の出力形式は object です。

11.4 HEW

11.4.1 ダイアログメニューが正しく表示されない

■質問

HIM および HEW で各ツールのオプションのダイアログボックスを開いたが、正しく表示されません。

■回答

Windows®95 の古いリリース（4.00.950a など）をご使用の場合、C/C++ Compiler, Assembler, IM OptLinker などのオプションを開くとアプリケーションエラーが発生し HEW が異常終了したり、オプションのダイアログボックスが正しく表示されないことがあります。これは、Windows ディレクトリの下の System ディレクトリにある COMCTL32.DLL のバージョンが古いために起こります。このような場合は Windows®95 をより新しいものにバージョンアップしてください。

11.4.2 オブジェクトファイルのリンク順序

■質問

HEW 上で、オブジェクトファイルのリンク順序を指定したい。

■回答

SuperH RISC engine Standard Toolchain の最適化リンカタブのカテゴリ:[ソース]からオプション項目:[リロケータブルファイル／オブジェクトファイル]を選択し、追加を押し、オブジェクトファイルを追加してください。ここで指定した順番でオブジェクトがリンクされます。

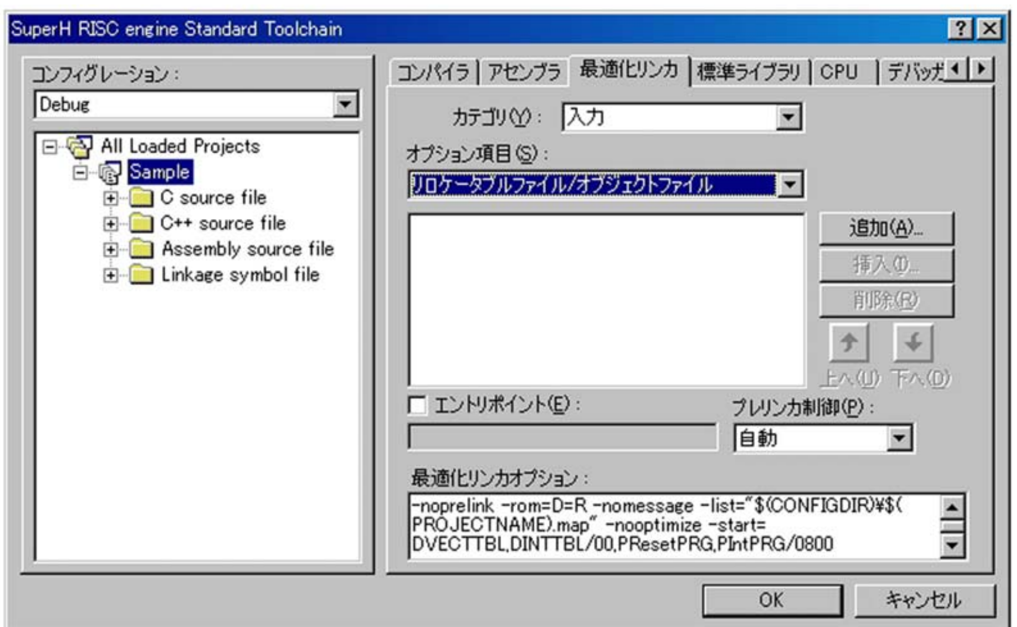


図 11.16 Link/Library ダイアログボックス

RENESAS

SHC V.8.00 Release02 以降では HEW により簡単にリンクの順序を指定できます。[ビルド->リンク順の指定]からリンク順序のカスタマイズダイアログを呼び出します。ここでリンク順序を指定してください。上にリストされているものほど先にリンクされます。

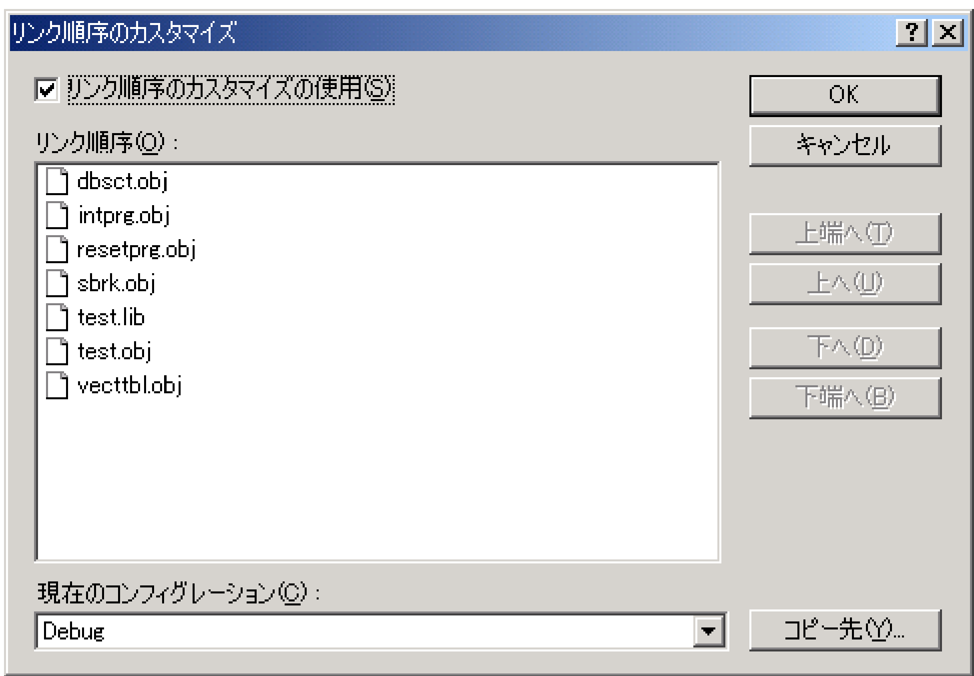


図 11.17 **リンク順序のカスタマイズダイアログボックス**

11.4.3 MAP 最適化の指定方法

■質問

MAP 最適化を指定するとウォーニングメッセージが表示されます。

■回答

SuperH RISC engine Standard Toolchain のコンパイラタブのカテゴリ:[最適化]から外部変数アクセス最適化をチェックすると、**図** 11.18 のウォーニングメッセージが表示されます。これは、最適化リンカタブのカテゴリ:[アウトプット]の外部シンボル割り付け情報ファイル出力を自動的に有効にするためです。

図 11.18 C/C++**ダイアログボックス**

図 11.19 WARNING **メッセージ**

11.4.4 プロジェクトファイルの除外

■質問

プロジェクトのファイルを一時的にビルドから除外したい。

■回答

ワークスペースウィンドウの“ Projects”タブのファイル上でマウスの右ボタンを押下し、[ビルドから除外 <file>]を選択してください。すると、そのファイルがビルドから除外されます。再びファイルをビルドに戻すには、ワークスペースウィンドウの“ Projects”タブの当該ファイル上でマウスの右ボタンを押下し、[ビルドから除外の解除 <file>]を選択してください。

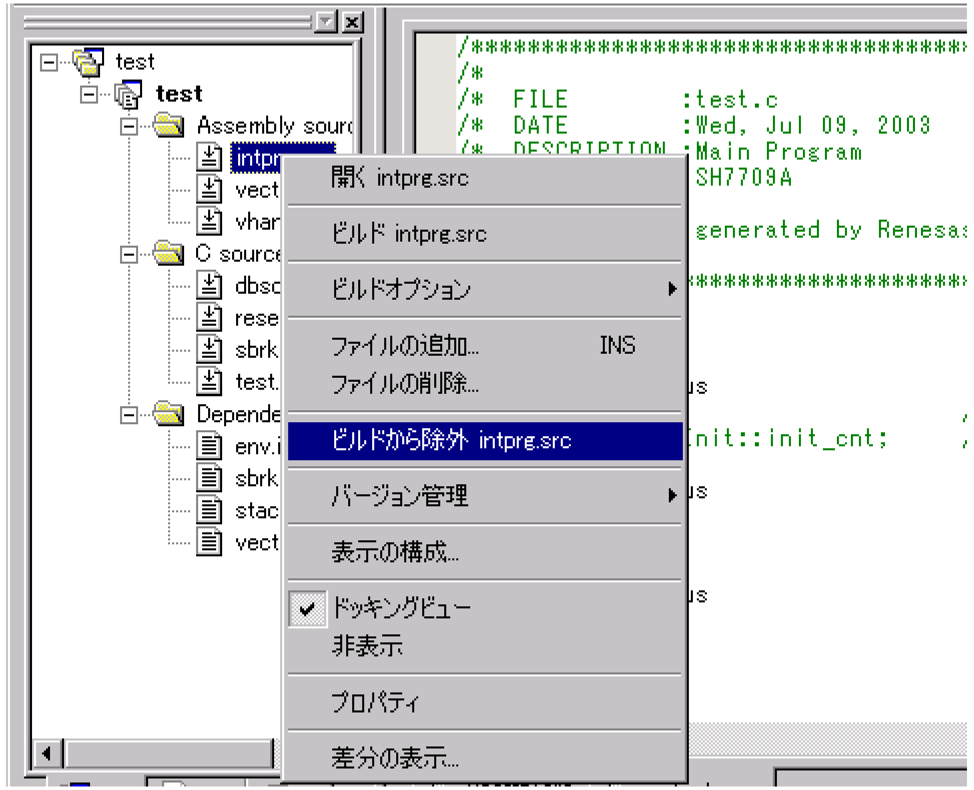


図 11.20 ビルドから除外 メニュー

11.4.5 プロジェクトファイルのデフォルトオプション指定

■質問

プロジェクトにファイルを追加するとき、ファイルに自動的にデフォルトオプションを指定したい。

■回答

SuperH RISC engine Standard Toolchain ダイアログボックスの左側にファイルのリストが表示されます（図 11.21）。ファイルリストで、デフォルトオプションを指定したいファイルグループのフォルダを開いてください。フォルダ内に“ Default Options” アイコンが表示されます。アイコンを選択して、オプションダイアログボックスの右側でオプションを指定して“ OK”をクリックしてください。このオプションは、プロジェクトにそのファイルグループのファイルを初めて追加するときに適用されます。

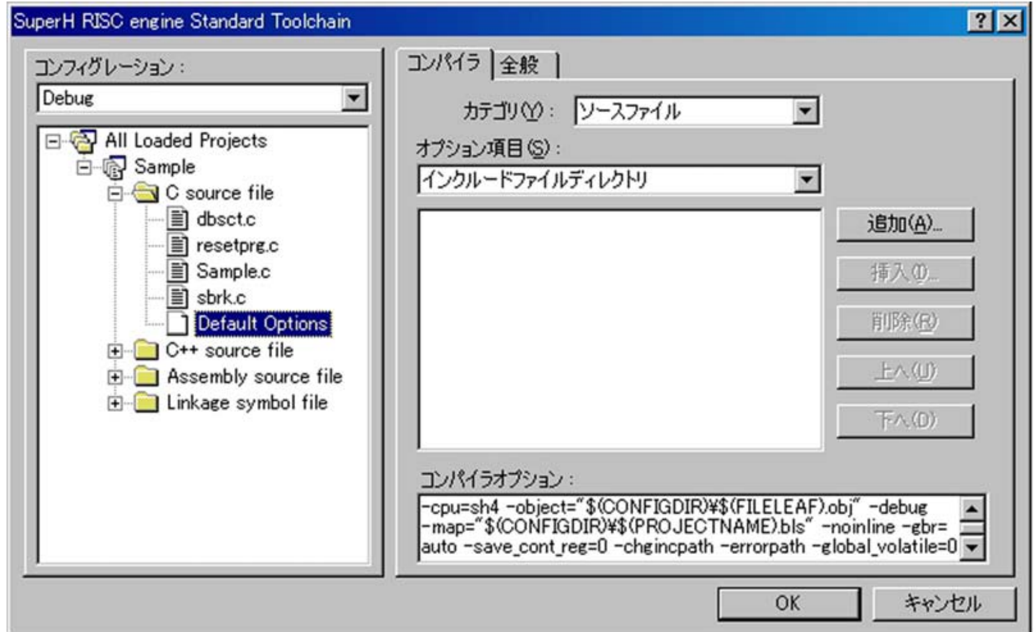


図 11.21 デフォルトオプション

11.4.6 メモリマップの変更方法

■質問

メモリマップを変更できない。

■回答

メモリウィンドウのメモリリソースがマッピングされていると、システムコンフィギュレーションウィンドウでメモリマップの変更はできません。メモリリソースのマッピングを解除してからメモリマップを変更してください。

11.4.7 HEW のネットワーク上での使用について

■質問

(1) HEWをネットワーク上にインストールできますか？
(2) プロジェクトおよびプログラムをネットワーク上においても問題ありませんか？

■回答

(1) HEW本体はネットワーク上にはインストールできません。
(2) 問題ありません。ただし、同一ファイルへ複数ユーザがアクセスしないように管理してださい。

11.4.8 HEW で作成するファイル、ディレクトリ名の制限

■質問

HEW システムを起動させようとしたところ”Error has occurred whilst saving file <ファイル名>”というエラーが出力されますが、どうしてですか？

■回答

HEW システムで作成するファイルやディレクトリは、制限があります。
次に示す項目は半角英数字、または、半角の下線のみを使用してください。

・インストールするディレクトリ名
・プロジェクトを作成するディレクトリ名
・プロジェクト名

11.4.9　HEW エディタ、HDI での日本語表示フォントがおかしい

■質問

(1)　HEWのエディタで日本語が表示されない。
(2)　HEWエディタで漢字が90度回転して表示される。
(3)　モジュール間最適化ツールでSYNTAX ERRORが出力される。

■回答

HEW エディタで日本語を記述する場合はフォントを日本語用フォントに変更してください。

＜HEW2.0 以降＞

ツール-> 表示形式...のフォントタブのフォントで変更します。

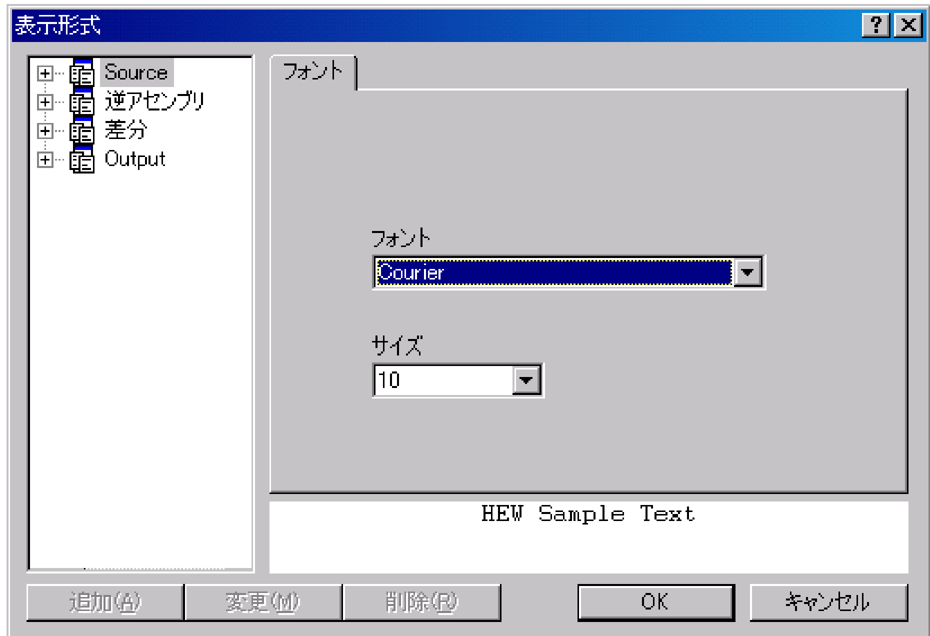


図 11.22　フォントダイアログボックス

RENESAS

HDI で日本語フォントが正しく表示されない場合は、次のように変更します。
[Setup->Customize->Font…]で、変更します。

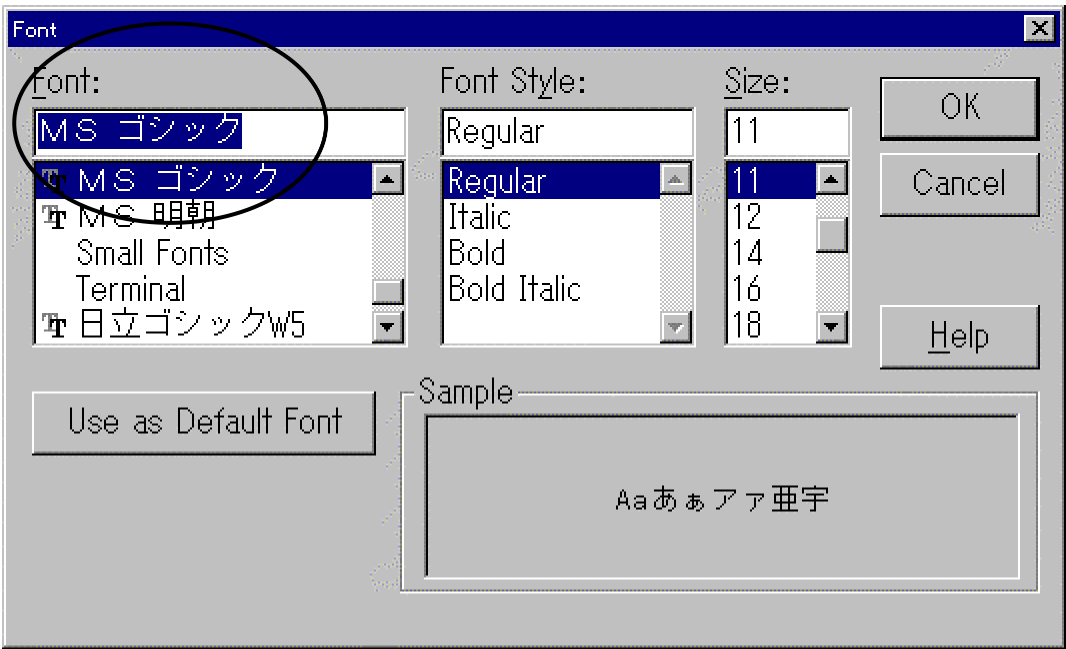


図 11.23 Font ダイアログボックス

11.4.10 HIM から HEW への変換方法

■質問

HIM(Hitachi Integration Manager)で作成したプロジェクトを HEW で運用したい。

■回答

HEW システムに添付されている HIM To HEW Project Converter というツールを使用し、HIM から HEW へ変換することが可能です。

このツールの詳細については High-performance Embedded Workshop **リリースノート**「**第 3 章 HIM から HEW のプロジェクト変換**」を参照してください。

RENESAS

11.4.11 HEW のプロジェクト構築時に当該デバイスがない

■質問

HEW でプロジェクトを構築するときに当該デバイスが選択肢にありません。

■回答

ルネサスの Web サイトからデバイスアップデータを入手してください。

デバイスアップデータは HEW が生成するプロジェクトファイルを更新するツールです。新しい CPU に対するプロジェクトのサポートも順次行っていきます。

デバイスアップデータでプロジェクトファイルを更新しても当該デバイスが選択できない場合には手動で HEW が生成したファイルを書き換える必要があります。例として SH-2 コアの SH7018 のプロジェクトを構築します。まず HEW の新規プロジェクト作成で CPU シリーズで SH-2、CPU タイプで Other を選択しプロジェクトを構築します。ルネサスの Web サイトから I/O レジスタ定義ファイルをダウンロードし SH7018.H をプロジェクトに追加してください。プロジェクト構築時に CPU タイプで Other を選択しなかった場合は SH7018.H を iodefine.h にリネームして HEW が生成した同名ファイルに上書きしてください。

また割り込み関数を使用するときは HEW が生成したファイルを変更する必要があります。「11.1.41　**割り込み処理を記述したい**」をご参照ください。

RENESAS

11.4.12 古いコンパイラ(ツールチェイン)を最新の HEW に登録したい

■質問

エミュレータを購入したところ、新しい HEW が付属してきました。デバッグとビルドを同じ HEW で行うため、古いコンパイラ(ツールチェイン)を新しい HEW に登録したいのですが、どうしたらよいでしょうか？

コンパイラパッケージの組み合わせによって対応が異なります。以下をご参照下さい。

[SHC V.4 以前]

＜ビルド＞

ツールチェインを最新の HEW に登録することはできません。そのため、HEW を用いたビルドはできません。

＜デバッグ＞

アブソリュートファイル(*.abs)を使用することはできません。S タイプファイルを使用してデバッグしてください。
このとき、C ソースレベルデバッグを行うことはできません。アセンブラレベルのデバッグとなります。

[SHC V.5.0]

＜ビルド＞

ツールチェインを最新の HEW に登録することはできません。そのため、HEW を用いたビルドはできません。

(補足)

SHC V.5.1 をお持ちであれば、「HIM to HEW Project Converter」を用いて HIM のプロジェクトファイルを HEW のプロジェクトファイルに変換することが可能です。変換後は SHC V.5.1 でのビルドが可能です。

＜デバッグ＞

アブソリュートファイル(*.abs)を使用することはできません。S タイプファイルを使用してデバッグしてください。
このとき、C ソースレベルデバッグを行うことはできません。アセンブラレベルのデバッグとなります。

[SHC V.5.1]

＜ビルド＞

ツールチェインを最新の HEW に登録することが可能です。そのため、HEW を用いてビルドすることが可能です。
ただし、最新の HEW ではプロジェクトを構築することができません。
プロジェクトを構築する場合は、古いコンパイラに付属している HEW V.1 を用いて構築する必要があります。
HEW V.1 で作成したプロジェクトファイルは最新の HEW で開くことが可能です。

＜デバッグ＞

アブソリュートファイル(*.abs)を使用することはできません。S タイプファイルを使用してデバッグしてください。
このとき、C ソースレベルデバッグを行うことはできません。アセンブラレベルのデバッグとなります。

[SHC V.6]

＜ビルド＞

ツールチェインを最新の HEW に登録することが可能です。そのため、HEW を用いてビルドすることが可能です。
ただし、最新の HEW ではプロジェクトを構築することができません。
プロジェクトを構築する場合は、古いコンパイラに付属している HEW V.1 を用いて構築する必要があります。
HEW V.1 で作成したプロジェクトファイルは最新の HEW で開くことが可能です。

＜デバッグ＞

アブソリュートファイル(*.abs)を最新の HEW に登録することで、ソースレベルデバッグが可能です。

[SHC V.7 以降]

最新の HEW にアップデート可能です。最新 HEW のすべての機能をご利用いただけます。

付録

付録 A.　実行時ルーチン命名規則

実行時ルーチンの関数名の命名規則を以下に示します。

(1)　整数演算、浮動小数点演算、符号変換、ビットフィールド関数の命名規則

［演算名］［サイズ］［符号］［r］［p］［nm］

［サイズ］　：b・・・1バイト
　　　　　　：w・・・2バイト
　　　　　　：l・・・4バイト
　　　　　　：s・・・4バイト[単精度浮動小数点]
　　　　　　：d・・・8バイト[倍精度浮動小数点]
［符号］　　：s・・・符号付き
　　　　　　：u・・・符号なし
［r］　　　：_subdr,_divdrのみ。それぞれ_subd,_divdとパラメータのスタックプッシュ順序が異なるときのみ
［p］　　　：ペリフェラル時のみ付与
［nm］　　：ノーマスク。ペリフェラルで割り込みノーマスク時のみ付与
例外　　　：_muli

【注】［符号］は整数演算のみ付与

(2)　変換関数の命名規則

_［サイズ］to［サイズ］

［サイズ］　：i・・・符号付き4バイト
　　　　　　：u・・・符号なし4バイト
　　　　　　：s・・・単精度浮動小数点
　　　　　　：d・・・倍精度浮動小数点

(3)　シフト関数の命名規則

［sta］sft［方向］［符号］［ビット数］

［sta_］　　：ビット数の付く場合のみ付与
［方向］　　：l・・・左シフト
　　　　　　：r・・・右シフト
［符号］*1　：l・・・論理シフト
　　　　　　：a・・・算術シフト
［ビット数］*2：0～31

【注】*1　［符号］は［方向］が r のときのみ付与
**　　　*2　［ビット数］は［sta_］があるときのみ付与**

(4)　その他の関数の命名規則

領域移動、文字列比較、文字列コピーは特例。

付録 B.　追加機能について

B.1　Ver.1.0 から Ver.2.0 への追加機能

SHC コンパイラ Ver.2.0 で追加された機能概要を表 B.1 に示します。

表 B.1　SHC コンパイラ Ver.2.0 追加機能概要

項番	機能	内容
1	SH7600 シリーズのサポート	SH7000 シリーズのほかに、SH7600 シリーズの命令を活用したオブジェクトを生成することもできます。
2	ポジションインディペンデントコード	SH7600 シリーズのオブジェクトでは、プログラムセクションを任意のアドレスに配置できるオブジェクトが生成できます。
3	文字列の出力領域指定	文字列データを定数セクション(ROM)に置くかデータセクション(RAM)に置くかをオプションで選択できます。
4	コメントのネスト	コメントをネストさせるかさせないかを指定するオプションをサポートします。
5	サイズ、速度の優先指定	オブジェクト生成時にサイズを優先するかスピードを優先するかをオプションで指定できます。
6	セクション名切り替えのサポート	プログラムの途中で#pragma 指令によってオブジェクトを出力するセクション名を切り替えることができます。
7	mac 組み込み関数	MAC 命令を用いて 2 つの配列の積和演算を行う組み込み関数をサポートします。
8	システムコール組み込み関数	ITRON 仕様 OS HI-SH7 のシステムコールを直接呼び出す組み込み関数をサポートします。
9	単精度初等関数ライブラリ	単精度の初等関数ライブラリをサポートします。
10	char 型のビットフィールド	char 型のビットフィールドをサポートします。

B.2 Ver.2.0 から Ver.3.0 への追加機能

SHC コンパイラ Ver.3.0 で追加された機能概要を表 B.2 に示します。

表 B.2 SHC コンパイラ Ver.3.0 追加機能概要

No.	機能	内容
1	最適化強化	最適化機能を大幅に強化します。 また、スピード重視、サイズ重視の最適化オプションを使い分けることができます。
2	SH-3 サポート	SH-3 用のオブジェクト生成オプションを実現すると共に、SH-3 の特長機能である Little Endian もサポートします。また、SH-3 のデータプリフェッチ命令を組み込み関数としてサポートします。
3	コンパイラ限界値の拡張	一度にコンパイルできるファイル数、インクルードファイルのネストレベルなどの限界値を拡張します。
4	文字列漢字コードのサポート	シフト JIS、EUC の漢字コードを、プログラム内に文字列データとして記述できます。
5	ファイルによるオプション指定	コマンドラインのオプション指定をファイルで行うことができます。
6	SH-2 除算器の活用	SH-2 の除算器を活用した除算コードを生成します。
7	インライン展開	C 記述、アセンブラ記述のユーザルーチンをインライン展開することを指定できます。
8	短いアドレス指定の活用	2 バイトサイズのアドレス、GBR 相対のデータなど、短いアドレシングが可能な変数を指定できます。
9	レジスタ退避・回復の制御	レジスタの退避・回復の抑止を指定し、関数のスピード、サイズを向上させることができます。

(1) 最適化強化

Ver.3.0 の最適化は、スピード重視(-SPEED オプション)、サイズ重視(-SIZE オプション)の両オプションを設け、それぞれの最適化機能を大幅に強化しています。

スピードに関しては、ループ最適化の強化、インライン展開の実施などにより、実行スピードが約 10%向上、1MIPS/1MHz の性能を達成しています。

サイズに関しては、サイズ重視の命令生成、重複処理の併合の大幅な強化などにより、オブジェクトサイズを約 20%削減しています。さらに Ver.3.0 で導入された拡張機能(8.短いアドレス指定の活用、9.レジスタ退避・回復の制御)の活用により、さらにオブジェクトサイズを削減することが可能です。

(2) SH-3 サポート

SH-1、SH-2 に加えて、SH-3 のオブジェクト生成を指定することができます(-CPU=SH3 オプション)。さらに、SH-3 用の機能として、以下をサポートします。

(a) **メモリ内のビット並び順の設定機能に対応して、-ENDIAN オプション(-ENDIAN=BIG、または LITTLE)をサポート。**

(b) **キャッシュのプリフェッチ命令(PREF)を生成する拡張組み込み関数 prefetch をサポート。**

(3) コンパイラ限界値の拡張

以下の点で、コンパイラ限界値をさらに拡張します。

表 B.3 コンパイラ限界値の拡張

No.	項目	Ver.2.0	Ver.3.0
1	一度にコンパイルできるソースプログラムの数	16 ファイル	制限なし
2	1 ファイルあたりのソース行数	32767 行	65535 行
3	コンパイル単位全体のソース行数	32767 行	制限なし
4	#include のネストレベル	8 レベル	30 レベル

(4) 文字列内漢字コードのサポート

シフト JIS、EUC の漢字コードを、プログラム内に文字列データとしても記述できます。

入力コードがシフト JIS の場合(-SJIS オプション)、出力コードもシフト JIS、入力コードが EUC の場合(-EUC オプション)、出力コードも EUC です。

ただし、現状の GUI は漢字コードのデータ表示には対応しておりません。

(5) ファイルによるオプションの指定

-SUBCOMMAND オプションでファイル名を指定することによって、オプションをファイル内から取り込むことができるようになります。これによって、複雑なオプションを毎回コマンドラインから指定する必要がなくなります。

(6) SH-2 除算器の活用

SH-2 の除算器を活用するために、以下のオプションをサポートします。

(a) -DIVISION=CPU除算器を使用しないオブジェクトを生成します。

(b) -DIVISION=PERIPHERAL除算器を使用するオブジェクトを生成します。
除算器を使用するときは割り込みを禁止します。

(c) -DIVISION=NOMASK除算器を使用するオブジェクトを生成します。
割り込み処理では除算器を使用しないことを想定します。

(7) インライン展開

(a) C 関数のインライン展開

-SPEED オプションを指定すると、コンパイラは、小さな関数を自動的にインライン展開します。さらに、-INLINE オプションによって、インライン展開する関数の大きさの条件を変更することができます。

インライン展開は、#pragma 指定によって明示的にすることもできます。

#pragma inline は、C 記述のユーザ関数をインライン展開することを指定します。

例(C 関数のインライン展開)：

```
#pragma inline (func)
int func(int a,int b)
{
return (a+b)/2 ;
}

main()
{
i=func(10,20); /*i=(10+20)/2 に展開されます*/
}
```

(b) 2 アセンブラ関数のインライン展開

#pragma inline_asm はアセンブラ記述のユーザ関数をインライン展開することを指定します。

ただし、#pragma inline_asm でインライン展開を行った場合、コンパイラの出力はアセンブラソースになります。この場合 C 言語レベルのデバッグはできなくなります。

例 (アセンブラ関数のインライン展開)：

```
#pragma inline_asm(rotl)
int rotl(int a)
{
 ROTL R4
 MOV R4,R0
}

main()
{
 i=rotl(i) ; /*変数iをレジスタR4に設定し、rotl関数のコードを展開します */
}
```

(8) 短いアドレス指定の活用

(a) 2 バイトアドレス変数の指定

#pragma abs16(<変数名>)によって、変数が 2 バイトで指定できるアドレス範囲(-32768～32767)に割り付けられていることを指定できます。この指定によって、変数を参照するオブジェクトサイズを削減することができます。

(b) GBR ベース変数の指定

#pragma gbr(<変数名>)によって、変数を GBR 相対アドレシングモードで参照することができることを指定できます。この指定によって変数を参照するオブジェクトサイズを削減するとともに、GBR 相対アドレシングモードに特有なメモリ上のビット操作命令を活用することができます。

(9) レジスタ退避・回復の制御

#pragma noregsave(<関数名>)によって、関数の入口、出口でのレジスタの退避・回復を抑止することを指定できます。これにより、レジスタの退避・回復のない高速でコンパクトな関数を作成できます。#pragma noregsave を指定した関数は、通常の関数から呼び出すことはできませんが、#pragma noregsave を指定した関数を呼び出すように明示的に指定した C 言語関数(#pragma regsave)からは呼び出すことができます。

頻繁に実行する関数を#pragma noregsave と指定することによって、プログラムサイズを削減し、実行速度を向上させることができます。

B.3 Ver.3.0 から Ver.4.1 への追加機能

SuperH RISC engine C/C++コンパイラ Ver.4.1 で追加した機能概要を説明します。

(1) 外部変数のレジスタ割り当て

#pragma global_register(<変数名>=<レジスタ番号>)によって、外部変数をレジスタに割り当てることが可能となりました。

(2) キャッシュを意識した最適化

命令フェッチやキャッシュを無駄なく利用するためにラベルを 16 バイト整合して割り付けるオプション、"-align16" をサポートしました。

(3) インライン展開機能の強化

インライン展開により、関数本体が使用されなくなった場合に、その削除を行う機能を追加しました。インライン展開後の関数本体が必要ない関数には static を指定してください。また、呼び出しやアドレス参照されることのない static 関数も同様に削除します。

例)

```
#pragma inline(func)   #pragma inline(func)
int a;   int a;
static int func(){   /* func() 関数本体削除 */
a++;
 }
main(){main(){
func();    a++; /* インライン展開 */
}   }
```

(4) 再帰的なインライン展開

関数を再帰的にインライン展開する機能を追加しました。再帰の深さは、"-nestinline"オプションにより指定可能です。

(5) ループ展開最適化オプション

"-loop", "-noloop"オプション指定によりループ処理を展開させる最適化を行うかどうかを"-speed", "-size"オプションとは独立に指定可能となりました。
(非最適化オプション指定時、本オプションは無効)

(6) 2 バイトアドレス変数指定のオプション化

従来 2 バイトアドレス変数は #pragma abs16 を用いて個々に指定する必要がありましたが、"-abs16"オプションにより一括指定する機能をサポートしました。"-abs16=run" では実行時ルーチンのみを、"-abs16=all"では実行時ルーチンを含む全変数および関数を 2 バイトアドレスとして指定できます。

(7) 関数 return 値の上位バイトの保証

従来 (unsigned) char, short 型の値を返す関数の return 値の上位バイトは非保証でしたが、オプション "-rtnext" を指定することにより保証（R0 の上位バイトを符号拡張または 0 拡張）する機能を追加しました。

(8) リスティングファイルの充実化

以前のバージョンに比べ、オブジェクトリスト、アセンブリソースに情報を充実し、見やすくしました。

- リストファイル中に C ソースとアセンブリプログラムを文単位に同時出力することにより、その対応が見やすくなりました。（"-show=source,object" オプション指定時）
- （これに伴い、"-show"オプションのデフォルトを source から nosource に変更しました。）
- 関数のスタック使用量算出のための情報として、その関数での使用実行時ルーチン名一覧を追加。
- 定数プールからのデータロード命令に、ロードデータをコメント表示。

例）

```
1:float x;
2:func(){
3:x/=1000;
4:}
```

リスティングファイル

```
func.c1float x;
func.c2func(){※(a)Cソースとアセンブリプログラムの同時出力
000000_func:          ; function: func
  ; frame size=4
                     ; used runtime library name:
                     ; __divs ※(b)実行時ルーチン名
000000 4F22         STS.L  PR,@-R15
func.c 3 x/=1000;
000002 D404         MOV.L  L216+2,R4; _x
000004 D004         MOV.L  L216+6,R0; H'447A0000 ※(c)ロードデータ
000006 D305         MOV.L  L216+10,R3 ; __divs
000008 430B         JSR  @R3
00000A 6142         MOV.L  @R4,R1
func.c 4 }
00000C 4F26       LDS.L  @R15+,PR
00000E 000B       RTS
000010 2402       MOV.L  R0,@R4
000012 L216:
000012 00000002   .RES.W  1
000014 <00000000> .DATA.L  _x
000018 447A0000   .DATA.L  H'447A0000
00001C <00000000> .DATA.L  __divs
000000_ x:; static:x
000000 00000004.RES.L  1
```

(9) エラーメッセージの強化

"-message"オプション指定によりインフォメーションメッセージを出力することでコーディングミスのチェックを強化しました。

例）

```
1:void func(){
2:int a;
3:a++;
4:sub(a);
5:}
```

インフォメーションメッセージ

```
line 3: 0011 (I) Used before set symbol : "a"（auto変数の未定義参照）
line 4: 0200 (I) No prototype function      （プロトタイプ宣言なし）
```

また、エラーとなっている識別子、字句、番号をメッセージ中に追加することによりエラー箇所を見つけやすくなりました。

例）

```
: 2118 (E) Prototype mismatch "識別子"
: 2119 (E) Not a parameter name "識別子"
: 2201 (E) Cannot covert parameter "番号"
: 2225 (E) Undeclared name "識別子"
: 2500 (E) Illegal token "字句"
```

(10) 日本語文字コードの自動変換

EUC、またはシフト JIS 日本語コードで記述された文字列をオブジェクトファイルに出力する際に、オプションにより指定された日本語コードへの自動変換が可能です。

(a) "-outcode=euc"オプションにより日本語コードをEUCコードへ変換。

(b) "-outcode=sjis"オプションにより日本語コードをシフトJISコードへ変換。

(11) 環境変数による CPU タイプ指定

CPU タイプをコマンドラインオプションで指定する代わりに、環境変数による指定を可能にしました。

環境変数指定
SHCPU=SH1（"-cpu=sh1"オプションと同意）
SHCPU=SH2（"-cpu=sh2"オプションと同意）
SHCPU=SH3（"-cpu=sh3"オプションと同意）

(12) double 型データの float 型データ化オプション

"-double=float"オプションにより、double型で宣言されたデータを float型に読み替える機能を追加しました。double型の精度の必要のないプログラムでは、ソース修正なしで実行速度の向上が可能です。

B.4 Ver.4.1 から Ver.5.0 への追加機能

SuperH RISC engine C/C++コンパイラ Ver.5.0 で追加した機能概要を説明します。

(1) 文字数の拡張

1 行論理行の文字数の制限を、4,096 文字から 32,768 文字までに拡張しました。

(2) コンパイルソース行制限の廃止

1 ファイルで 65,535 行を超えるソースはコンパイルできない制限を廃止しました。
ただし、65,535 行を超えた部分はデバッグできません。

(3) SH-4 命令対応

SH ファミリの CPU 展開に即応し、新たに SH-4 対応を追加しました。“ -cpu=sh4 ” オプション指定により、SH-4 のオブジェクトを生成できます。

(4) 正規化モードの追加

“ -denormalize=on|off ”オプション指定により、非正規化数を扱うか、0 とするか選択が可能になりました。-cpu=sh4 時のみ有効です。

ただし、“ -denormalize=on ” のとき、FPU に非正規化数が入力されると例外発生するので、非正規化数を処理するための例外処理をソフトウェアで記述する必要があります。

(5) 丸めモード追加

“ -round=nearest|zero ” オプション指定により、Round to zero で丸めるか、Round to nearest で丸めるかの選択が可能になりました。-cpu=sh4 時のみ有効です。

(6) 環境変数によるコンパイラオプション指定の SH-4 対応

CPU をコマンドラインオプションで指定する代わりに、環境変数 “ SHCPU ” による SH-4 指定を可能にしました。“ SHCPU=SH4 ” で指定できます。

(7) SH-2E 対応

“ -cpu=sh2e ” オプション指定により、SH-2E のオブジェクトを生成できます。

(8) 環境変数によるコンパイラオプション指定の SH-2E 対応

CPU をコマンドラインオプションで指定する代わりに、環境変数“ SHCPU ”による SH-2E 指定を可能にしました。“ SHCPU=SH2E ” で指定できます。

(9) 拡張子による C/C++の判別

コンパイラは shc、shcpp のコマンドの使い分けでもコンパイル時の文法を決定しますが、shc コマンド使用時でもファイルの拡張子やオプションにより C++コンパイルを行います。詳細を**表 B.4** に示します。

表 B.4 コンパイル条件判別表

コマンド	オプション	コンパイル対象ファイルの拡張子	コンパイル条件
shcpp	任意	任意	C++でコンパイル
shc	-lang=c	任意	C でコンパイル
	-lang=cpp		C++でコンパイル
	-lang オプションの指定なし	*.c	C でコンパイル
		*.cpp, *.cc, *.cp, *.CC	C++でコンパイル

B.5　Ver.5.0 から Ver.5.1 への追加機能

SuperH RISC engine C/C++コンパイラ Ver.5.1 で追加した機能概要を説明します。

(1)　SH3-DSP ライブラリのサポート

従来の SH-DSP 用に加えて SH3-DSP にも適用可能なライブラリをサポートしました。

(2)　Embedded C++言語のサポート

組み込みシステムに適した C++仕様である Embedded C++の言語仕様とライブラリをサポートしました。

・bool型サポート
・多重継承の警告
・Embedded C++クラスライブラリのサポート

(3)　モジュール間最適化機能のサポート

以下の最適化を実施してサイズ／スピードに優れたオブジェクトを生成します。
この最適化により約 10%のサイズ削減、7～8%の実行スピード向上を実現しました。

・無駄なレジスタ退避回復コードの削減
・参照しない変数／関数の削除
・共通コードをルーチン化
・関数呼び出しコードの最適化

(4)　コンパイルスピードの向上

最適化処理の改善によるコンパイラスピードの高速化を実現しました。
最大 2 倍、平均 130%のスピードアップを達成しました。

(5)　制限値の拡張

・コマンドライン長の制限を256から4096へ拡張しました。
・ファイル名長の制限を128から251へ拡張しました。
・文字列リテラル長の制限を512から32767へ拡張しました。

(6)　最適化強化

オブジェクト性能を向上させる各種最適化を強化しました。

(7)　C++コメントのサポート

C 言語でも「//」コメントの使用が可能になりました。

(8)　統合環境の変更（PC 版）

従来の PC の統合化環境 HIM(Hitachi Integration Manager)に代わる新たな統合化環境 HEW(High-performance Embedded Workshop)になりました。

HIM と比較し以下機能を追加致しました。

・プロジェクトジェネレータ
各CPUごとに周辺I/Oなどを定義したヘッダファイルを自動生成します。
・バージョン管理ツールとの連動
市販のバージョン管理ツールとの連動インタフェースをサポートしました。
・階層プロジェクトのサポート
プロジェクト内に複数のサブプロジェクトを定義し、階層的に管理することが可能となりました。
・ネットワーク対応
WindowsNTのCSS環境下での開発が可能になりました。

B.6 Ver.5.1 から Ver.6.0 への追加機能

SuperH RISC engine C/C++コンパイラ Ver.6.0 で追加した機能概要を説明します。

(1) 制限値の緩和

ソースプログラムやコマンドラインの制限を大幅に緩和しました。

- ・ファイル名長：251バイト→無制限
- ・シンボル長：251バイト→無制限
- ・シンボル数：32767個→無制限
- ・ソースプログラム行数：C/C++：32767行、ASM：65535行→無制限
- ・Cプログラム文字列長：512 文字→ 32766 文字
- ・アセンブリプログラム行長：255 文字→ 8192 文字
- ・サブコマンドファイル行長：ASM:300 バイト、optlnk:512 バイト→無制限
- ・最適化リンケージエディタrom オプションのパラメータ数:64 個→ 無制限

(2) ディレクトリ名、ファイル名のハイフン(-)

ディレクトリ名、ファイル名にハイフン(-)を指定できるようになりました。

(3) コピーライト表示抑止

logo/nologo オプション指定により、コピーライト表示有無を指定できるようになりました。

(4) エラーメッセージのプリフィックス

統合開発環境でのエラーヘルプ機能サポートに伴い、コンパイラ、最適化リンケージエディタのエラーメッセージの先頭にプリフィックスを付与しました。

(5) fpscr オプションの追加

cpu=sh4 オプションを指定しかつ fpu オプションを指定していないとき、関数呼び出し前後の FPSCR レジスタの精度モードを保証するかどうかを指定できるようになりました。

(6) #pragma 拡張子

#pragma 拡張子が( )なしで記述できるようになりました。

(7) 組み込み関数の追加

trace 関数を追加しました。

(8) 暗黙の宣言の追加

_ _HITACHI_ _、_ _HITACHI_VERSION_ _が暗黙に#define 宣言されます。

(9) static ラベル名

#pragma inline_asm でファイル内 static ラベルを参照できるように、ラベル名を_ _$(名前)に変更しました。ただし、リンケージリストでは_(名前)と表示されます。

(10) 言語仕様拡張・変更

- ・共用体初期化時のエラーを抑止します。

例：

```
union{
char c[4];
} uu={{'a','b','c'}};
```

- ・構造体の代入と宣言を同時にできるようになりました。

RENESAS

例：

```
struct{
int a, int b;
} s1
void test()
{
struct S s2=s1;
}
```

・bool 型データの境界調整数が4byteになりました。
・C++言語仕様として、例外処理やテンプレート機能もサポートしました。
・CプリプロセッサがANSI/ISO対応しました。

B.7 Ver.6.0 から Ver.7.0 への追加機能

SuperH RISC engine C/C++コンパイラ Ver.7.0 では、最適化およびコード生成のアルゴリズムが大幅に改良されました。そのため、オプションや生成コードが Ver.6.0 までとは大きく異なります。

SuperH RISC engine C/C++コンパイラ Ver.7.0 で追加した機能概要を説明します。

(1) 外部アクセス最適化機能（map オプションサポート）

リンク時の変数、関数の割り付けアドレスに基づいた外部変数アクセス、関数分岐命令の最適化を行います。一度目のリンク時に最適化リンケージエディタより出力(map オプション指定)された外部シンボル割り付け情報ファイルを用いてリコンパイルすることにより、最適化を実施します。

(2) GBR 相対アクセスコード自動生成（gbr オプションサポート）

gbr=auto を指定した場合、コンパイラが GBR の設定および GBR 相対アクセスコードを自動生成します。関数呼び出し前後で GBR の値を保証します。ただし、GBR 関連の組み込み関数は使用できません。

(3) speed/size 選択オプションの強化

speed/size 選択オプション(shift、blockcopy、division、approxdiv オプション)を追加し、より細かな size/speed 調整を可能にしました。

(4) 組み込み向け機能の強化

・組み込み関数の追加
倍精度乗算、SWAP 命令、LDTLB 命令、NOP 命令

・#pragma 拡張子の追加、変更
#pragma entry エントリ関数指定、SP の設定
#pragma stacksize スタックサイズの指定
#pragma interrupt sp=<変数>+<定数>、sp=&<変数>+<定数>のサポート

・セクション演算子のサポート
セクションのアドレス、サイズ参照をC 言語で記述できる機能をサポートしました。

・アドレスキャストのエラー緩和
外部変数初期化時のアドレス初期化に対するキャスト式のエラーを緩和しました。

(5) ライブラリの改善

・リエントラントライブラリサポート
ライブラリ構築ツールでreent オプションを指定した場合、リエントラントライブラリが生成されます。

・malloc 確保サイズ単位、入出力ファイル数の可変化
C/C++ライブラリ関数の初期設定において、_sbrk_sizeでmalloc 確保サイズを、_nfilesで入出力ファイル数を指定できるようになりました。これにより、RAM 容量を節約できます。
指定を省略した場合、malloc 確保サイズは520、入出力ファイル数は20 になります。

・簡易I/O のサポート
ライブラリ構築ツールでnofloat オプションを指定した場合、浮動小数点変換をサポートしない、サイズの小さいI/O ルーチンを生成します。

(6) 最適化オプションの追加（V7.0.06）

・追加オプション
V7.0.06で追加したオプション一覧を以下に示します。英大文字は短縮形指定時の文字を、下線は省略時解釈を示します。

RENESAS

表 B.5　追加オプション一覧

	項目	オプション名	内容
1	外部変数の volatile 化	GLOBAL_Volatile = {0 |	外部変数について volatile 宣言されたものとして扱わない(volatile 宣言されたものは除く)
		1 }	すべての外部変数について volatile 宣言されたものとして扱う
2	外部変数最適化範囲指定	OPT_Range = {All |	関数内の全範囲で外部変数を最適化対象とする
		NOLoop |	ループ制御変数やループ内の外部変数のループ外への移動を抑止
		NOBlock}	ループや、分岐をまたいだ外部変数に対する最適化をすべて抑止
3	空ループ削除	DEL_vacant_loop ={0 |	空ループ削除を抑止
		1}	空ループ削除を行う
4	ループ最大展開数の指定	MAX_unroll = <数値> <数値>:1-32	ループ展開時の最大展開数を指定 default：1 (speed,loop 指定時 2)
5	無限ループ前の式削除	INFinite_loop = {0 |	無限ループ前の外部変数定義削除を抑止
		1 }	無限ループ前の外部変数定義削除を行う
6	外部変数のレジスタ割り付け	GLOBAL_Alloc = {0 |	外部変数のレジスタ割り付けを抑止
		1 }	外部変数のレジスタ割り付けを行う
7	構造体/共用体メンバのレジスタ割り付け	STRUCT_Alloc = {0 |	構造体／共用体メンバのレジスタ割り付けを抑止
		1 }	構造体／共用体メンバのレジスタ割り付けを行う
8	const 定数伝播	CONST_Var_propagate = {0 |	const 宣言された外部変数の定数伝播を抑止
		1 }	const 宣言された外部変数の定数伝播を行う
9	定数ロードの命令展開	CONST_Load = {Inline |	定数ロードを命令展開
		Literal }	定数ロードをリテラルアクセス default：size 指定時 2-3 命令まで展開
10	命令並べ替え	SChedule = {0 |	命令並べ替えを行わない
		1 }	命令並べ替えを行う

外部変数の volatile 化

GLOBAL_Volatile

Optimize[Details][Global variables][Treat global variables as volatile qualified]

書　式　GLOBAL_Volatile = { 0 | 1 }

説　明　global_volatile=0 を指定した場合、volatile 宣言のない外部変数に対して最適化を行います。したがって、外部変数のアクセス回数、アクセス順序が C/C++プログラムで記述した場合と異なることがあります。
global_volatile=1 を指定した場合、すべての外部変数を volatile 宣言したものとして扱います。したがって、外部変数のアクセス回数、アクセス順序は C/C++で記述したとおりになります。
本オプション省略時解釈は、global_volatile=0 です。

備　考　global_volatile=1 を指定した場合、schedule=0 がデフォルトになります。

外部変数最適化範囲指定

OPT_Range

Optimize[Details][Global variables][Specify optimizing range :]

書　式　`OPT_Range = { All | NOLoop | NOBlock }`

説　明　opt_range=all を指定した場合、関数内の全範囲を対象に外部変数に対する最適化を行います。
opt_range=noloop を指定した場合、ループ内にある外部変数やループ判定式で使用されている外部変数を最適化の対象外にします。
opt_range=noblock を指定した場合、分岐をまたいだ外部変数の最適化(ループを含む)をすべて抑止します。
本オプション省略時解釈は、opt_range=all です。

例

(1) 分岐をまたいだ最適化例(opt_range=all/noloop 指定時に行う)

```
int A,B,C;
void f(int a) {
    A = 1;
    if (a) {
        B = 1;
    }
    C = A;
}
```

＜最適化後のソースイメージ＞

```
void f(int a) {
    A = 1;
    if (a) {
        B = 1;
    }
    C = 1;      /* A の参照を削除し、A=1 を伝播する */
}
```

(2) ループにおける最適化例(opt_range=all 指定時に行う)

```
int A,B,C[100];
void f() {
    int i;
    for (i=0;i<A;i++) {
        C[i] = B;
    }
}
```

＜最適化後のソースイメージ＞

```
void f() {
    int i;
    int temp_A, temp_B;
    temp_A = A; /* ループ判定式の A の参照をループ外に移動*/
    temp_B = B; /* ループ内の B の参照をループ外に移動     */
    for (i=0;i<temp_A;i++) {     /* A のループ内での参照を削除 */
        C[i] = temp_B;           /* B のループ内での参照を削除 */
    }
}
```

備　考　opt_range=noloop を指定した場合、常に max_unroll=1 がデフォルトになります。
opt_range=noblock を指定した場合、常に max_unroll=1、const_var_propagate=0、global_alloc=0 がデフォルトになります。

空ループ削除

DEL_vacant_loop

Optimize[Details][Miscellaneous][Delete vacant loop]

書 式　DEL_vacant_loop = { 0 | 1 }

説 明　del_vacant_loop=0 を指定した場合、ループ内処理がない場合でもループを削除しません。
del_vacant_loop=1 を指定した場合、ループ内処理がないループは削除します。
本オプション省略時解釈は、del_vacant_loop=0 です。

備 考　SHC/C++V7 台(V7.0.04 まで)のコンパイラとはデフォルトが異なりますのでご注意ください。
V7.0.04 まで ：空ループを削除する
V7.0.06 ：空ループを削除しない

ループ最大展開数の指定

MAX_unroll

Optimize[Details][Miscellaneous][Specify maximum unroll factor :]

書 式　MAX_unroll = <数値>

説 明　ループ展開時の最大展開数を指定します。<数値>には 1 から 32 までの整数を指定することができます。それ以外の値を指定した場合はエラーになります。
本オプション省略時解釈は、speed または loop オプションを指定した場合は max_unroll=2、それ以外の場合は max_unroll=1 です。

備 考　opt_range=noloop/noblock を指定した場合、常に max_unroll=1 がデフォルトになります。

無限ループ前の式削除

INFinite_loop

Optimize[Details][Global variables]
[Delete assignment to global variables before an infinite loop]

書 式　INFinite_loop = { 0 | 1 }

説 明　infinite_loop=0 を指定した場合、無限ループ直前の外部変数への代入式を削除しません。
infinite_loop=1 を指定した場合、無限ループ直前にあり無限ループ内で参照されない外部変数への代入式を削除します。
本オプション省略時解釈は、infinite_loop =0 です。

例

```
int A;
void f()
{
   A = 1;       /* 外部変数 A への代入式*/
   while(1) {}  /* A は参照されない    */
}

＜infinite_loop=1 指定時のイメージ＞
void f()
{
                /* 外部変数 A への代入式を削除 */
   while(1) {}
}
```

備 考　SHC/C++V7 台(V7.0.04 まで)のコンパイラとはデフォルトが異なりますのでご注意ください。
V7.0.04 まで ：無限ループ直前のループ内で参照されない外部変数への代入式を削除する
V7.0.06 ：無限ループ直前の外部変数への代入式を削除しない

外部変数のレジスタ割り付け

GLOBAL_Alloc

Optimize[Details][Global variables][Allocate registers to global variables :]

書 式　GLOBAL_Alloc = { 0 | 1 }

説 明　global_alloc=0 を指定した場合、外部変数のレジスタ割り付けを抑止します。
global_alloc=1 を指定した場合、外部変数のレジスタ割り付けを行います。
本オプション省略時解釈は、global_alloc=1 です。

備 考　opt_range=noblock を指定した場合、global_alloc=0 がデフォルトになります。
optimize=0 を指定した場合、SHC/C++V7 台(V7.0.04 まで)のコンパイラとはデフォルトが異なりますのでご注意ください。
V7.0.04 まで ：外部変数のレジスタ割り付けを行う
V7.0.06 ：外部変数のレジスタ割り付けを抑止する

構造体/共用体メンバのレジスタ割り付け

STRUCT_Alloc

Optimize[Details][Miscellaneous][Allocate registers to struct/union members]

書　式　STRUCT_Alloc = { 0 | 1 }

説　明　struct_alloc=0 を指定した場合、構造体/共用体メンバのレジスタ割り付けを抑止します。
struct_alloc=1 を指定した場合、構造体/共用体メンバのレジスタ割り付けを行います。
本オプション省略時解釈は、struct_alloc=1 です。

備　考　opt_range=noblock もしくは global_alloc=0 を指定しかつ struct_alloc=1 を指定した場合、ローカル構造体/共用体メンバのみレジスタ割り付けを行います。
optimize=0 を指定した場合、SHC/C++V7 台(V7.0.04 まで)のコンパイラとはデフォルトが異なりますのでご注意ください。
V7.0.04 まで ：ローカル構造体/共用体メンバのレジスタ割り付けを行う
V7.0.06 ：ローカル構造体/共用体メンバのレジスタ割り付けを抑止する

const 定数伝播

CONST_Var_propagate

Optimize[Details][Global variables][Propagate variables which are const qualified :]

書　式　CONST_Var_propagate = { 0 | 1 }

説　明　const_var_propagate=0 を指定した場合、const 宣言された外部変数の定数伝播を抑止します。
const_var_propagate=1 を指定した場合、const 宣言された外部変数についても定数伝播を行います。
本オプション省略時解釈は、const_var_propagate=1 です。

例

```
const int X = 1;
int A;
void f() {
    A = X;
}

＜const_var_propagate =1 指定時のソースイメージ＞
void f() {
    A = 1;      /* X=1 を伝播 */
}
```

備　考　opt_range=noblock を指定した場合、const_var_propagate=0 がデフォルトになります。
C++プログラムで const 宣言された変数については本オプションで制御することはできません(常に定数伝播されます)。

定数ロードの命令展開

CONST_Load

Optimize[Details][Miscellaneous][Load constant value as :]

書 式　CONST_Load = { Inline | Literal }

説 明　const_load=inline を指定した場合、すべての 2 バイト定数および一部の 4 バイト定数ロードを命令展開します。
const_load=literal を指定した場合、すべての 2 バイト以上の定数ロードをリテラルアクセスします。
本オプション省略時解釈は、speed オプションを指定した場合は const_load=inline、size または nospeed オプションを指定した場合は 2 バイト定数の場合は 2 命令、4 バイト定数の場合は 3 命令で展開できる場合は const_load=inline、それ以外は const_load=literal です。

例

```
int f() {
   return (257);
}
```

(1) const_load=inline または speed 指定時

```
MOV     #1,R0   ; R0 <- 1
SHLL8   R0      ; R0 <- 256 (1<<8)
RTS
ADD     #1,R0   ; R0 <- 257 (256+1)
```

(2) const_load=literal または size/nospeed 指定時

```
   MOV.W   L11,R0
   RTS
   NOP
L11:
   .DATA.W H'0101
```

命令並べ替え

SChedule

Optimize[Details][Global variables][Schedule instructions :]

書 式　SChedule = { 0 | 1 }

説 明　schedule=0 を指定した場合、命令並べ替えを行いません。したがって C/C++記述した順番で処理を行います。
schedule=1 を指定した場合、パイプライン処理、スーパスカラ(SH-4)を考慮した命令並べ替えを行います。
本オプション省略時解釈は、schedule=1 です。

備 考　global_volatile=1 を指定した場合、schedule=0 がデフォルトになります。

- optimize=0 **指定時のデフォルト**

optimize=0 を指定した場合、追加オプションのデフォルトは以下のようになります。

global_volatile=0
opt_range=noblock
del_vacant_loop=0
max_unroll=1
infinite_loop=0
global_alloc=0
struct_alloc=0
const_var_propagate=0
const_load=literal
schedule=0

以下のオプションについては、optimize=1 指定時とデフォルトの仕様が異なります。

	optimize=0	optimize=1
opt_range	noblock	all
global_alloc	0	1
struct_alloc	0	1
const_var_propagate	0	1
const_load	literal	speed/size/nospeed **による**
schedule	0	1

- V7（V7.0.04 **まで）との互換性**

以下オプションは、V7(V7.0.04 まで)とデフォルトの仕様が異なります。

(i) 空ループ削除(del_vacant_loop)
V7.0.04まで　：空ループを削除する
V7.0.06　：空ループを削除しない

(ii) 無限ループ前の式削除(infinite_loop)
V7.0.04まで　：無限ループ前の外部変数定義を削除する
V7.0.06　：無限ループ前の外部変数定義を削除しない

また、以下については V7(V7.0.04 まで)と optimize=0 指定時の仕様が異なります。

(i) 外部変数レジスタ割り付け(global_alloc)
V7.0.04まで　：外部変数のレジスタ割り付けを行う
V7.0.06　：外部変数のレジスタ割り付けを抑止

(ii) 構造体/共用体メンバレジスタ割り付け(struct_alloc)
V7.0.04まで　：構造体/共用体メンバのレジスタ割り付けを行う
V7.0.06　：構造体/共用体メンバのレジスタ割り付けを抑止

- **オプション体系**

外部変数に対する最適化を以下のレベルで設定しました。HEW で以下のレベルを設定することにより、当該オプションを一括制御することができます。

設定は、Optimize[Details][Level :]で行います。

(i) Level 1
外部変数の最適化をすべて抑止する。

global_volatile=1
opt_range=noblock
infinite_loop=0
global_alloc=0
const_var_propagate=0
schedule=0

(ii) Level 2

volatile指定のない外部変数を分岐(ループを含む)を超えない範囲で最適化する。

global_volatile=0
opt_range=noblock
infinite_loop=0
global_alloc=0
const_var_propagate=0
schedule=1

(iii)Level 3

volatile指定のない外部変数をすべて最適化対象とする。

global_volatile=0
opt_range=all
infinite_loop=0
global_alloc=1
const_var_propagate=1
schedule=1

(iv)Custom

外部変数に対する最適化をプログラムにあわせてユーザが指定する。

Level 1～3 を指定した場合、上記オプションについては変更することはできません。

（7）以降は最適化リンケージエディタで追加した機能です。

(7) ワイルドカードのサポート

入力ファイルや start オプションのセクション名でワイルドカードを指定できます。

(8) サーチパス

環境変数(HLNK_DIR)により、複数の入力ファイル、ライブラリファイルのサーチパスを指定できます。

(9) ロードモジュール分割出力

アブソリュートロードモジュールファイルを分割出力できます。

(10) エラーレベルの変更

インフォメーション、ウォーニング、エラーレベルのメッセージは、個別にエラーレベルや出力有無を変更できます。

(11) バイナリ、HEX サポート

バイナリファイルを入出力できるようになりました。
また、インテル HEX タイプの出力も選択できるようになりました。

(12) stack 使用量情報の出力

stack オプションにより、スタック解析ツール用情報ファイルを出力できます。

(13) デバッグ情報削除機能

strip オプションにより、ロードモジュールファイルやライブラリファイル内のデバッグ情報だけを削除できます。
SuperH RISC engine C/C++コンパイラ Ver.7.1 最適化リンケージエディタで追加した機能概要を説明します。

(14) 外部シンボル割付情報ファイルの出力（map オプションサポート）

map オプションを指定した場合、コンパイラが外部変数アクセス最適化で使用する外部シンボル割り付け情報ファイルを生成します。

B.8 Ver.7.0 から Ver.7.1 への追加機能

- SuperH RISC engine C/C++コンパイラ Ver.7.1 で追加した機能概要を説明します。

(1) 最適化の強化

(a) MOVT 直後の EXTU 削除

MOVT の直後の不要な EXTU を削除する。
（1または0以外が設定されることはないので EXTU は不要）

最適化前	最適化後
_f: MOV.L L12+2,R6 ; _a1 MOV.B @R6,R0 TST #128,R0 MOVT R0 EXTU.B R0,R0	_f: MOV.L L12+2,R6 ; _a1 MOV.B @R6,R0 TST #128,R0 MOVT R0
R0 には 1 または 0 以外は設定されないので EXTU は不要。	

(b) ゼロ拡張したレジスタの右シフト後の EXTU 削除

ゼロ拡張後のレジスタを右シフトした後にゼロ拡張しても値は変わらないため削除する。

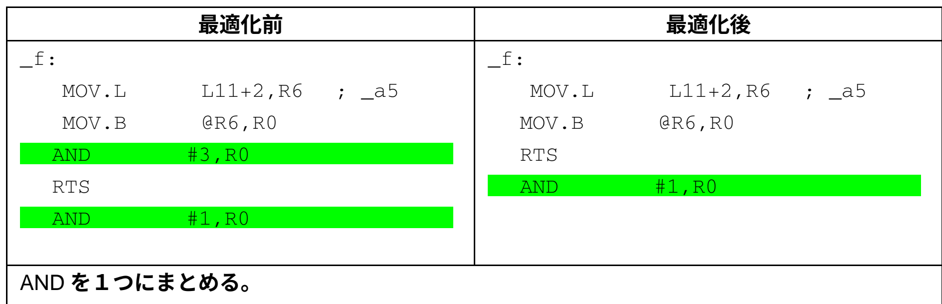
最適化前	最適化後

_f:
MOV.L L13+2,R2; _a2
MOV #1,R5
MOV.W @R2,R6
EXTU.W R6,R6
MOV R6,R2
SHLR2 R2
SHLR R2
EXTU.W R2,R2
CMP/GE R5,R2
:
	_f:
	MOV.L L13+2,R2; _a2
	MOV #1,R5
	MOV.W @R2,R6
	EXTU.W R6,R6
	MOV R6,R2
	SHLR2 R2
	SHLR R2
	CMP/GE R5,R2
	:

EXTU で上位 2 バイトはゼロクリアされているので再度 EXTU を行っても値は変わらない。	

(c) 連続した AND の統合

同じ変数への AND が連続した場合、１つの AND にまとめる。

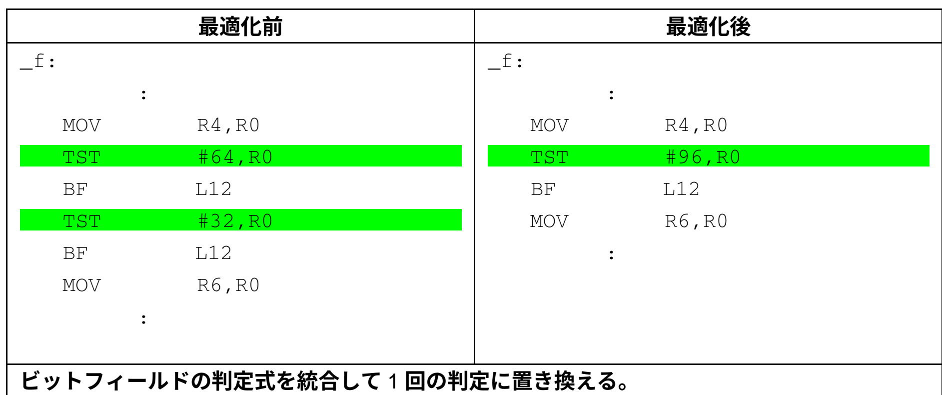


最適化前	最適化後
_f: MOV.L L11+2,R6 ; _a5 MOV.B @R6,R0 AND #3,R0 RTS AND #1,R0	_f: MOV.L L11+2,R6 ; _a5 MOV.B @R6,R0 RTS AND #1,R0
AND を１つにまとめる。	

RENESAS

(d) **ビットフィールド比較結合**

複数のビットフィールドの判定(TST #n, R0)を統合する。



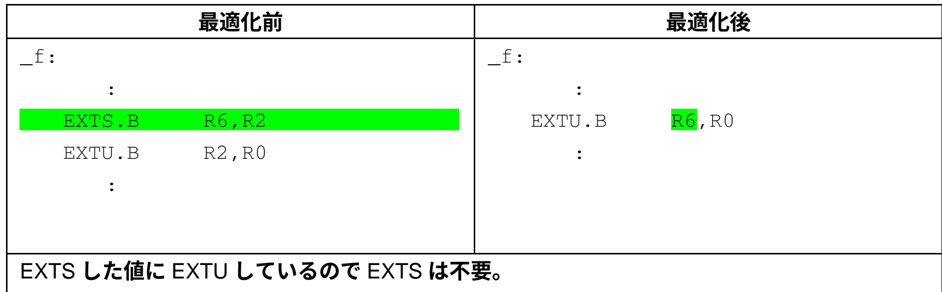
最適化前	最適化後

_f:
:
MOV R4,R0
TST #64,R0
BF L12
TST #32,R0
BF L12
MOV R6,R0
:
	_f:
	:
	MOV R4,R0
	TST #96,R0
	BF L12
	MOV R6,R0
	:

ビットフィールドの判定式を統合して 1 回の判定に置き換える。	

(e) **連続した EXTS＋EXTU の EXTS 削除**

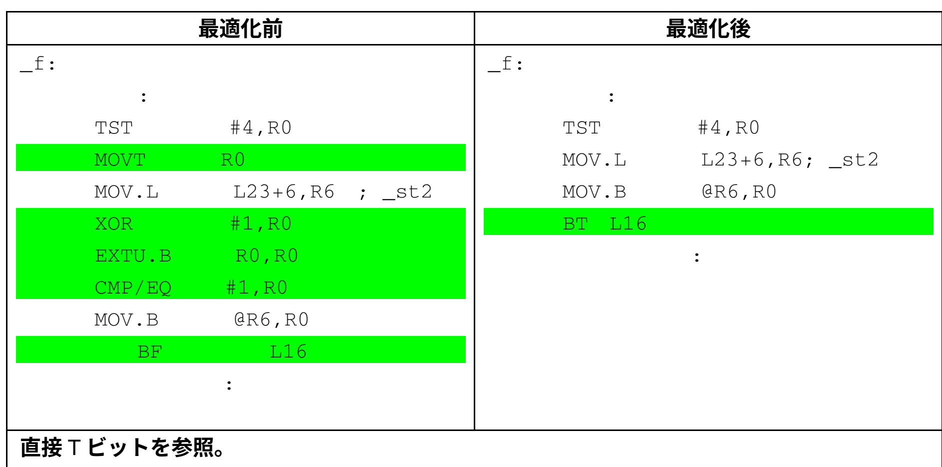
EXTS 後に同じサイズの EXTU を行った場合、EXTS は不要なため削除する。



最適化前	最適化後
_f: : EXTS.B R6,R2 EXTU.B R2,R0 :	_f: : EXTU.B R6,R0 :
EXTS した値に EXTU しているので EXTS は不要。	

(f) **MOVT(+XOR)+EXTU+CMP/EQ の削除**

TST 後の不要な MOVT(+XOR)+EXTU+CMP/EQ を削除し、T ビットを直接分岐命令で参照するように変換する。



最適化前	最適化後
_f: : TST #4,R0 MOVT R0 MOV.L L23+6,R6 ; _st2 XOR #1,R0 EXTU.B R0,R0 CMP/EQ #1,R0 MOV.B @R6,R0 BF L16 :	_f: : TST #4,R0 MOV.L L23+6,R6; _st2 MOV.B @R6,R0 BT L16 :
直接 T ビットを参照。	

(g) AND #imm,R0+CMP/EQ #imm,R0->TST #imm,R0

AND #imm,R0+CMP/EQ #imm,R0 を TST #imm,R0 に置き換える。

最適化前	最適化後
L17: MOV.B @R6,R0 AND #1,R0 CMP/EQ #1,R0 BF L19 MOV.B @R5,R0 AND #1,R0	L17: MOV.B @R6,R0 TST #1,R0 BT L19 MOV.B @R5,R0 AND #1,R0

(h) unsigned char と定数の比較(==)時の EXTU 削除

ロード直後の unsigned char と定数の比較時の不要な EXTU を削除する。

最適化前	最適化後
_f: MOV.L L11,R6 ; _b MOV.B @R6,R2 MOV #-128,R6; H'FFFFFF80 EXTU.B R6,R6 EXTU.B R2,R2 CMP/EQ R6,R2 MOVT R2 MOV.L L11+4,R6 ; _a RTS MOV.B R2,@R6	_f: MOV.L L11,R6 ; _b MOV.B @R6,R2 MOV #-128,R6; H'FFFFFF80 CMP/EQ R6,R2 MOVT R2 MOV.L L11+4,R6 ; _a RTS MOV.B R2,@R6
不要な拡張を削除。	

(i) ビットフィールド LOAD 後/STORE 前の拡張削除

ビットフィールドの LOAD 後/STORE 前の不要な拡張を削除する。

最適化前	最適化後
_f: MOV.L L11+2,R6;_st MOV.B @R6,R2 EXTU.B R2,R0 OR #128,R0 :	_f: MOV.L L11+2,R6;_st MOV.B @R6,R2 OR #128,R0 :
不要な拡張を削除。	

(j) switch-case の判定時のコピー削除

switch 文で case 判定ごとに行っている値のコピーを削除する。

最適化前	最適化後

_f:
                    :
MOV   R0,R2
MOV   R2,R0
CMP/EQ   #1,R0
BT   L24
CMP/EQ   #2,R0
BT   L26
MOV   R2,R0
CMP/EQ   #3,R0
BT   L28
MOV   R2,R0
CMP/EQ   #4,R0
BT   L30
MOV   R2,R0
        :
	_f:
	                    :
	MOV R0,R2
	MOV R2,R0
	CMP/EQ #1,R0
	BT L24
	CMP/EQ #2,R0
	BT L26
	CMP/EQ #3,R0
	BT L28
	CMP/EQ #4,R0
	BT L30
	        :

不要なコピー削除。	

(k) 連続した OR の統合

同じ変数への OR が連続した場合、１つの OR にまとめる。

最適化前	最適化後
_f: MOV.L L11+2,R6 ; _a5 MOV.B @R6,R0 OR #3,R0 RTS OR #1,R0	_f: MOV.L L11+2,R6 ; _a5 MOV.B @R6,R0 RTS OR #3,,R0
OR を１つにまとめる。	

(l) AND #imm,R0 または TST #imm,R0 直前の EXTS 削除

(i)AND #imm,R0
(ii)TST #imm,R0
直前の不要な拡張を削除する。

最適化前	最適化後

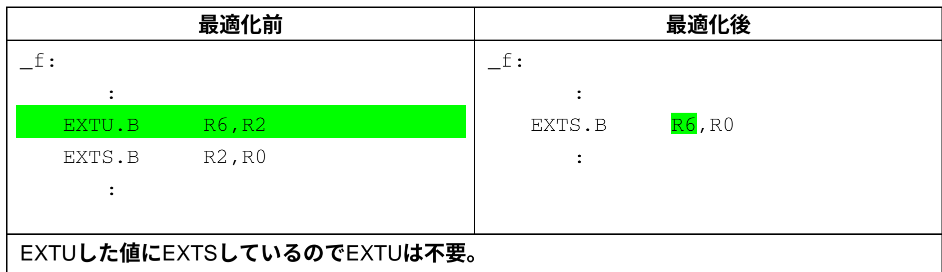
`_f:`
`:`
`EXTS.B R6,R0`
`AND #32,R0`
`:`
	`_f:`
	`:`
	`AND #32,R0`
	`:`

`_f:`
`:`
`EXTS.B R6,R0`
`TST #32,R0`
`:`
	`_f:`
	`:`
	`TST #32,R0`
	`:`

不要な拡張を削除。	

(m) 連続した EXTU+EXTS の EXTU 削除

EXTU 後に同じサイズの EXTS を行った場合、EXTU は不要なため削除する。



最適化前	最適化後

`_f:`
`:`
`EXTU.B R6,R2`
`EXTS.B R2,R0`
`:`
	`_f:`
	`:`
	`EXTS.B R6,R0`
	`:`

EXTUした値にEXTSしているのでEXTUは不要。	

RENESAS

(n) MOVT 後の XOR #imm,R0(OR,AND)直後の EXTU 削除

MOVT 後の
(i)XOR #imm,R0
(ii)OR #imm,R0
(iii)AND #imm,R0
直後の不要な EXTU を削除する。

最適化前	最適化後
: MOVT R0 XOR #1,R0 RTS EXTU.B R0,R0	: MOVT R0 RTS XOR #1,R0
MOVT R0 OR #2,R0 RTS EXTU.B R0,R0	: MOVT R0 RTS OR #2,R0
: MOVT R0 AND #1,R0 RTS EXTU.B R0,R0	: MOVT R0 RTS AND #1,R0
不要な拡張を削除。	

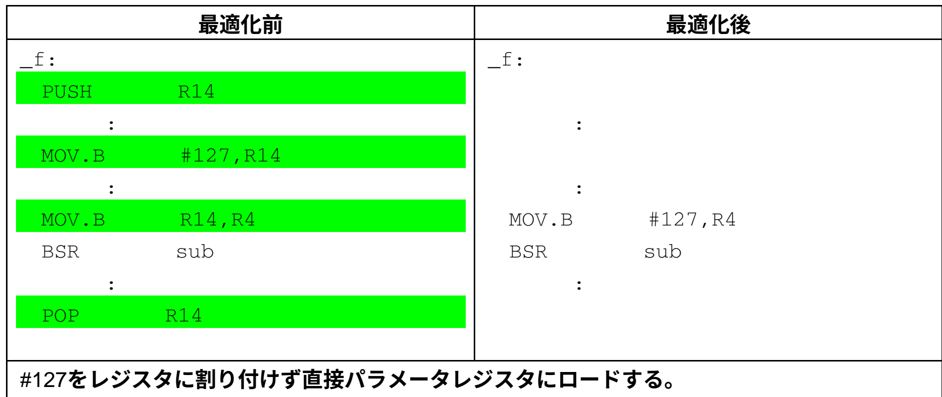
(o) 比較時の不要 EXTS 削除

符号拡張後のレジスタに対し、比較時に再度出力される冗長な EXTS を削除する。

最適化前	最適化後
_f: : EXTS.B R6,R6 CMP/GT R6,R2 BF L13 :	_f: : CMP/GT R6,R2 BF L13 :
R6 がすでに前で拡張済みの場合は EXTS は不要。	

(p) 定数値へのレジスタ割付（即値）抑止

関数引数の定数（-128～127）がレジスタに割り付くのを抑止する。



最適化前	最適化後
_f:	_f:
PUSH R14	
:	:
MOV.B #127,R14	
:	:
MOV.B R14,R4	MOV.B #127,R4
BSR sub	BSR sub
:	:
POP R14	
#127をレジスタに割り付けず直接パラメータレジスタにロードする。	

(q) DT 命令強化

レジスタに割りついた変数に対して DT 命令化を行う。

最適化前	最適化後
_f:	_f:
MOV.L L16+2,R6; _x	MOV.L L16+2,R6; _x
MOV.L @R6,R2	MOV.L @R6,R2
ADD #-1,R2	DT R2
TST R2,R2	BT/S L12
BT/S L12	:
:	
DT命令化を行う。	

(r) リテラル出力位置改善

リテラルデータ出力位置決定の際の命令サイズ計算の精度をあげ、リテラルデータ出力位置を後方に出力可能にする。

(s) 1byte&=1byte の冗長 EXTU 削除

1byte&=1byte 時の不要な EXTU を削除する。

最適化前	最適化後
_f:	_f:
:	:
MOV.B @(R0,R7),R6	MOV.B @(R0,R7),R6
MOV.B @R5,R2	MOV.B @R5,R2
EXTU.B R6,R6	AND R6,R2
AND R6,R2	MOV.B R2,@R5
MOV.B R2,@R5	MOV.B @R14,R2
MOV.B @R14,R2	:
:	
不要な拡張を削除する。	

(t)　2byte リテラルの展開

同じコードが 2 回展開されるのを改善する。

最適化前	最適化後
_f: MOV.L　L13+4,R4 ; _b SHLL8　R0 ADD　#-48,R0 MOV.W　@(R0,R4),R2 MOV　#8,R0 SHLL8　R0 ADD　#-46,R0 EXTU.W　R2,R6 MOV.W　@(R0,R4),R2 MOV　#8,R0 SHLL8　R0 ADD　#-44,R0 EXTU.W　R2,R5 MOV.W　@(R0,R4),R2	_f: MOV.L　L13+4,R4 ; _b SHLL8　R0 ADD　#-48,R0 MOV.W　@(R0,R4),R2 MOV　#8,R0 SHLL8　R0 ADD　#-46,R0 EXTU.W　R2,R6 MOV.W　@(R0,R4),R2 ADD　#2,R0 EXTU.W　R2,R5 MOV.W　@(R0,R4),R2
同じコードが2回展開されているのを改善。	

(u)　ループ条件判定の展開改善

size 優先時、ループ条件判定の際のループ判定の複写を行わない。

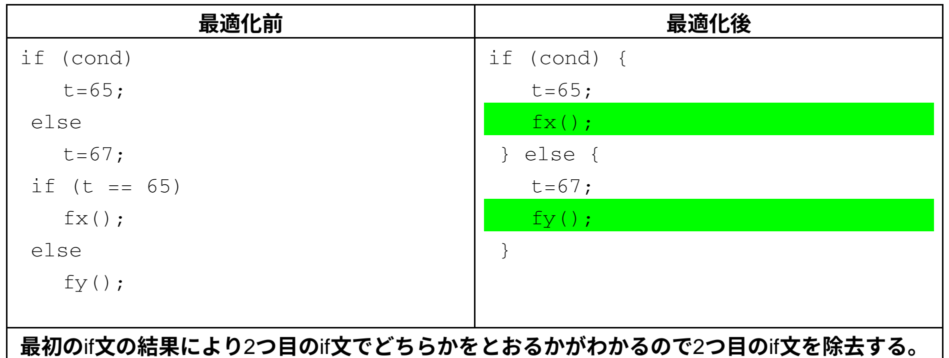
最適化前	最適化後(v7)	最適化後(v7.1)

while (cond) {
:
}
	if (cond) {
	do {
	:
	} while (cond);
	}
		goto L1;
		do {
		:
		L1:;
		} while (cond);

condの出現が2箇所から1箇所へ。		

(v)　冗長な if 文条件判定の除去

先の if 文の結果により後の if 文でどちらをとおるかがわかる場合は、後の if 文を除去する。



最適化前	最適化後

if (cond)
t=65;
else
t=67;
if (t == 65)
fx();
else
fy();
	if (cond) {
	t=65;
	fx();
	} else {
	t=67;
	fy();
	}

最初のif文の結果により2つ目のif文でどちらかをとおるかがわかるので2つ目のif文を除去する。	

RENESAS

(w) テンポラリ変数への直接演算

冗長な temp 変数への代入を抑止し、式の演算順序を変更する。

最適化前	最適化後
`k = i + prime;` `p = flags + k;`	`p = i + prime + flags;`
kはこの後使用されない -> 無駄なtempへの代入を行わない。	

(x) ポストインクリメントのアドレッシング

４バイト変数の LOAD について MOV.L @Rm+,Rn を活用する。

最適化前	最適化後

`:`
`L11:`
`MOV.L @R5,R2`
`ADD #4,R5`
`DT R6`
`ADD R2,R4`
`BF L11`
`:`
	`:`
	`L11:`
	`MOV.L @R5+,R2`
	`DT R6`
	`ADD R2,R4`
	`BF L11`
	`:`

MOV.L @Rm+,Rn一命令で実行。	

(y) ループ解消条件改善

ループ解消最適化を行う条件を緩和し、最適化がかかりやすくする。

最適化前	最適化後

`int a, b;`
`func() {`
`unsigned short sx;`
`for (sx=0; sx<1; sx++) {`
`a++;`
`b++;`
`f();`
`}`
`}`
	`int a, b;`
	`func() {`
	`a++;`
	`b++;`
	`f();`
	`}`

ループ解消を行う。	

(z) １ビット判定の最適化

１ビット幅のビットフィールドを複数個参照する条件式を１つにまとめ、ビット AND を用いて値の取り出しと比較を同時に行うコードを生成する。

最適化前	最適化後

struct S {
    char bit0:1;
    char bit1:1;
    char bit2:1;
    char bit3:1;
}ss1;
if ((ss1.bit0|ss1.bit1|ss1.bit2)!= 0){
    :
    :
}
	struct S {
	    char bit0:1;
	    char bit1:1;
	    char bit2:1;
	    char bit3:1;
	}ss1;
	if ((*(char *)&ss1 & 0xe0)!= 0){
	    :
	    :
	}

ANDを使用し、取り出しと比較を同時に行う。	

RENESAS

B.9 Ver.7.1 から Ver.8.0 への追加機能

- SuperH RISC engine C/C++コンパイラ Ver.8.0 で追加した機能概要を説明します。

(1) 新 CPU のサポート

SH-4A, SH4AL-DSP をサポートしました。

(2) 言語仕様拡張・変更

・DSP-C言語をサポートしました。
・long long、unsigned long long 型をサポートしました。

(3) 組み込み向け機能の強化

・DSP向け組み込み関数の追加
絶対値、MSB検出、算術シフト、丸め演算、ビットパターンコピー、モジュロアドレッシング設定、モジュロアドレッシング解除、CSビットの設定
・SH-4A、SH4AL-DSP向け組み込み関数の追加
正弦・余弦の計算、平方根の逆数、命令キャッシュブロックの無効化、命令キャッシュブロックのプリフェッチ、データ操作の同期
・#pragma 拡張子の追加、変更

#pragma ifunc	浮動小数点レジスタの退避／回復抑止
#pragma bit_order	ビットフィールド並び順指定
#pragma pack	構造体、共用体、クラスの境界調整数指定

(4) 列挙型サイズの自動選択(auto_enum オプションサポート)

列挙型が収まる最小型として列挙型を処理します。

(5) 構造体、共有体、クラスメンバの境界調整数を指定(pack オプションサポート)

構造体、共有体、クラスメンバの境界調整数を指定できます。

(6) ビットフィールド並び順指定(bit_order オプションサポート)

ビットフィールドのメンバの並び順を指定できます。

(7) エラーレベルの変更(change_message オプションサポート)

インフォメーション、ウォーニングレベルのメッセージは、個別にエラーレベルを変更できます。

(8) 制限値の緩和

switch 文の数を 2048 個に拡張しました。

(9) DSP ライブラリの固定小数点サポート

DSP ライブラリの固定小数点をサポートしました。

B.10　Ver.8.0 から Ver.9.0 への追加機能

- SuperH RISC engine　C/C++コンパイラ Ver.9.0 で追加した機能概要を説明します。

(1)　新 CPU のサポート

SH-2A、SH2A-FPU をサポートしました。
また、SH-2A、SH2A-FPU の TBR を活用する機能(tbr オプション、#pragma tbr)を追加しました。

(2)　言語仕様拡張・変更

・以下の項目をANSI 準拠しました。
・配列のインデックス
```
int iarray[10], i=3;
i[iarray] = 0; /* iarray[i] = 0;と同じになる */
```
・union のビットフィールド指定を可能にする
```
union u {
int a:3;
};
```
・定数演算
```
static int i=1||2/0; /* ゼロ除算をError からWarning に変更 */
```
・ライブラリ、マクロ追加
```
strtoul,FOPEN_MAX
```

・strict_ansi オプションを指定することにより、以下項目がANSI 準拠になります。
これにより、Ver.9 における結果が、Ver.8 までの結果と異なる場合があります。
・unsigned int とlong 型演算
・浮動小数点演算の結合則

・enable_register オプションを指定することにより、register 記憶クラスを指定した変数を優先的にレジスタに割り付けます。

(3)　組み込み向け機能強化

・SH-2A、SH2A-FPU 向け組み込み関数の追加
飽和演算、TBR 設定・参照

・C 言語で記述できない命令の組み込み関数サポート
T ビット参照・設定、連結レジスタの中央切り出し、キャリ付き加算、ボロー付き減算、符号反転、1 ビット除算、回転、シフト

(4)　制限値の緩和

以下の制限値を緩和しました。
・繰り返し文(while 文、do 文、for 文)、選択文(if 文、switch 文)の組み合わせによるネストの深さ：32 レベル → 4096 レベル
・1 関数内で指定可能なgoto ラベルの数：511 個 → 2147483646 個
・switch 文のネストの深さ：16 レベル → 2048 レベル
・1 つのswitch 文内で指定可能なcase ラベルの数：511 個 → 2147483646 個
・関数定義、関数呼び出しで指定可能引数：63 個 → 2147483646 個
・セクション名長：31 バイト → 8192 バイト
・1 ファイルあたりの#pragma section で指定できるセクション数：64 個 → 2045 個

(5)　メモリ空間配置指定の拡張

メモリ空間配置指定をより詳細に指定できます。
・abs16/abs20/abs28/abs32 オプション
・#pragma abs16/abs20/abs28/abs32

(6) 変数の絶対アドレス指定(#pragma address サポート)

外部変数のアドレスを絶対アドレス指定できます。

(7) 外部変数アクセス最適化機能拡張(smap オプションサポート)

コンパイル対象ファイル内で定義された外部変数について外部変数アクセス最適化を実施します。
map オプションのようなリコンパイルは必要ありません。

(8) 数学関数ライブラリ精度向上

数学関数ライブラリの演算精度が向上しました。
これにより、Ver.9 における結果が、Ver.8 までの結果と異なる場合があります。

付録 C. バージョンアップにおける注意事項

旧バージョン(「SuperH RISC engine C/C++コンパイラパッケージ」Ver.6.x 以前)からバージョンアップして使用する場合の注意事項を説明します。

C.1 プログラムの動作保証

コンパイラをバージョンアップしてプログラム開発する場合、プログラムの動作が変わることがあります。プログラムを作成する際は、以下の点に注意して、お客様のプログラムを十分にテストしてください。

(1) プログラムの実行時間やタイミングに依存するプログラム

言語仕様は、プログラムの実行時間については何も規定していません。したがってコンパイラのバージョンの違いによりプログラムの実行時間と I/O など周辺機器のタイミングのずれ、あるいは割り込み処理のような非同期処理の時間の差などにより、プログラムの動作が変わる場合があります。

(2) 1 つの式に 2 個以上の副作用が含まれているプログラム

1 つの式に 2 個以上の副作用が含まれている場合、コンパイラのバージョンにより、動作が変わる可能性があります。

例：

```
a[i++]=b[i++];          /* i のインクリメント順序は不定です。*/
f(i++,i++) ;            /* インクリメントの順序でパラメータの値が変わります。*/
                        /* i の値が 3 の時 f(3,4)または f(4,3)になります。*/
```

(3) 結果がオーバフローや不当演算に依存するプログラム

オーバフローが生じた場合や、不当演算を実施した場合、結果の値は保証しません。したがって、バージョンが変わると動作が変わる可能性があります。

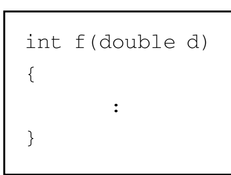
例：

```
int a, b;
x=(a*b)/10;   /* a と b の値の範囲によってはオーバフローする可能性があります*/
```

(4) 変数の初期化抜け、型の不一致

変数が初期化されていない場合や、パラメータやリターン値の型が呼び出し側と呼び出される側で対応していない場合、不正な値をアクセスすることになります。したがって、コンパイラのバージョンによって動作が変わる場合があります。

file1:



file2:

```
int g(void)
{
        f(1);
}
```

関数呼び出し側のパラメータは int型ですが、関数定義側のパラメータは、double型のため、値を正しく参照できません。

上記に記載された情報がすべての起こりうる状況を示したわけではありません。したがって、お客様の責任で本コンパイラを正しくご使用の上、お客様のプログラムを十分にテストしてください。

C.2 旧バージョンとの互換性

旧バージョンのコンパイラ(Ver.5.x 以前)、アセンブラ(Ver.4.x 以前)およびリンケージエディタ(Ver.6.x 以前)出力のオブジェクトファイル、ライブラリファイルとリンクする場合、または旧バージョンで使用していたデバッガをそのまま使用する場合に注意すべき点を説明します。

(1) オブジェクト形式

オブジェクトファイル形式は、従来の SYSROF から標準フォーマットの ELF 形式に変更しました。

また、デバッグ情報形式も、標準フォーマットの DWARF2 形式に変更しました。

旧バージョンのコンパイラ(Ver.5.x 以前)、アセンブラ(Ver.4.x 以前)出力オブジェクトファイル(SYSROF)を最適化リンケージエディタに入力する場合は、ファイルコンバータを使用して ELF 形式に変換してください。ただし、リンケージエディタ出力リロケータブルファイル(拡張子 rel)およびリロケータブルファイルを含むライブラリファイルは変換できません。

また、SYSROF 形式および ELF/DWARF1 形式ロードモジュールをサポートするデバッガを使用する場合は、ファイルコンバータを使用して ELF/DWARF2 形式ロードモジュールを、SYSROF または ELF/DWARF1 形式に変換してください。

(2) インクルードファイルの基点

ディレクトリ相対形式で指定されたインクルードファイル検索時、旧バージョンではコンパイラ起動ディレクトリを基点に検索していましたが、ソースファイルのあるディレクトリを基点に検索するように変更しました。

(3) C++プログラム

エンコード規則、実行方式を変更しましたので、旧バージョンのコンパイラで作成した C++オブジェクトファイルはリンクできません。必ずリコンパイルしてから使用してください。

また実行環境の設定で用いる、グローバルクラスオブジェクト初期処理／後処理のライブラリ関数名も変更になりました。SuperH RISC engine C/C++**コンパイラ、アセンブラ、最適化リンケージエディタユーザーズマニュアル**の「9.2.2 **実行環境の設定**」を参照し、修正してください。

(4) コモンセクションの廃止(アセンブリプログラム)

オブジェクトフォーマットの変更に伴い、コモンセクションのサポートを廃止しました。

(5) .END のエントリ指定(アセンブリプログラム)

.END でエントリ指定できるシンボルは外部定義シンボルだけになりました。

(6) モジュール間最適化

旧バージョンのコンパイラ(Ver.5.x 以前)、アセンブラ(Ver.4.x 以前)出力オブジェクトファイルは、モジュール間最適化の対象になりません。モジュール間最適化の対象にしたいファイルについては、必ずリコンパイル、リアセンブルしてください。

付録 D.　ASCII コード表

表 D.1　ASCII コード表

上位 4 ビット / 下位 4 ビット	0	1	2	3	4	5	6	7
0	NULL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	＼	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

ルネサスマイクロコンピュータ開発環境システム
アプリケーションノート
SuperH RISC engine C/C++コンパイラパッケージ

発行年月日　2007 年 2 月 2 日　Rev.7.00
発　　　行　株式会社ルネサス テクノロジ 営業統括部
〒100-0004 東京都千代田区大手町 2-6-2
編　　　集　株式会社ルネサスソリューションズ
グローバルストラテジックコミュニケーション本部
カスタマサポート部

© 2007. Renesas Technology Corp., All rights reserved. Printed in Japan.

株式会社ルネサス テクノロジ　営業統括部　〒100-0004 東京都千代田区大手町2-6-2 日本ビル

営業お問合せ窓口
株式会社ルネサス販売

RENESAS

http://www.renesas.com

本　　　　　社	〒100-0004	千代田区大手町2-6-2 (日本ビル)	(03) 5201-5350
京　浜　支　社	〒212-0058	川崎市幸区鹿島田890-12 (新川崎三井ビル)	(044) 549-1662
西　東　京　支　社	〒190-0023	立川市柴崎町2-2-23 (第二高島ビル2F)	(042) 524-8701
東　北　支　社	〒980-0013	仙台市青葉区花京院1-1-20 (花京院スクエア13F)	(022) 221-1351
い　わ　き　支　店	〒970-8026	いわき市平小太郎町4-9 (平小太郎ビル)	(0246) 22-3222
茨　城　支　店	〒312-0034	ひたちなか市堀口832-2 (日立システムプラザ勝田1F)	(029) 271-9411
新　潟　支　店	〒950-0087	新潟市東大通1-4-2 (新潟三井物産ビル3F)	(025) 241-4361
松　本　支　社	〒390-0815	松本市深志1-2-11 (昭和ビル7F)	(0263) 33-6622
中　部　支　社	〒460-0008	名古屋市中区栄4-2-29 (名古屋広小路プレイス)	(052) 249-3330
関　西　支　社	〒541-0044	大阪市中央区伏見町4-1-1 (明治安田生命大阪御堂筋ビル)	(06) 6233-9500
北　陸　支　社	〒920-0031	金沢市広岡3-1-1 (金沢パークビル8F)	(076) 233-5980
広　島　支　店	〒730-0036	広島市中区袋町5-25 (広島袋町ビルディング8F)	(082) 244-2570
鳥　取　支　店	〒680-0822	鳥取市今町2-251 (日本生命鳥取駅前ビル)	(0857) 21-1915
九　州　支　社	〒812-0011	福岡市博多区博多駅前2-17-1 (ヒロカネビル本館5F)	(092) 481-7695

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：コンタクトセンタ　E-Mail: csc@renesas.com

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

ルネサス エレクトロニクス株式会社
神奈川県川崎市中原区下沼部1753 〒211-8668

RJJ05B0557-0700