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

RENESAS

H8S、H8/300シリーズ 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.5.00　2005.08

安全設計に関するお願い

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

本資料ご利用に際しての留意事項

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

はじめに

本アプリケーションノートは、H8SX、H8S/2600、H8S/2000、H8/300H、H8/300、H8/300L シリーズマイクロコンピュータ上で動作する応用プログラムを H8S,H8/300 シリーズ C/C++コンパイラパッケージを用いて効果的に作成する方法を説明します。

なお、本アプリケーションノートで説明した内容の、詳細については、次の関連マニュアルに記載されておりますのであわせて参照してください。

High-performance Embedded Workshop 3 **ユーザーズマニュアル**
H8S,H8/300 **シリーズ** High-performance Embedded Workshop **チュートリアル**
H8S,H8/300 **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**
H8S,H8/300 **シリーズ シミュレータ・デバッガ ユーザーズマニュアル**
各製品ハードウェアマニュアル、プログラミングマニュアル

本アプリケーションノートは次のような構成になっております。
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。
第 3 章では、プログラム開発時に利用する拡張機能について説明いたします。
第 4 章では、HEW のオプションを説明します。
第 5 章では、最適化機能の説明および利用方法の説明をいたします。
第 6 章では、効果的なプログラミング技法を示します。
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、最適化リンケージエディタの説明および利用方法の説明いたします。
第 10 章では、MISRA C および SQMlint の活用法を説明いたします。
第 11 章では、ユーザから多くよ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、下記を記載しています。
A. 浮動小数点演算の性能一覧
B. 追加機能について
C. 制限値一覧
D. ASCII コード表

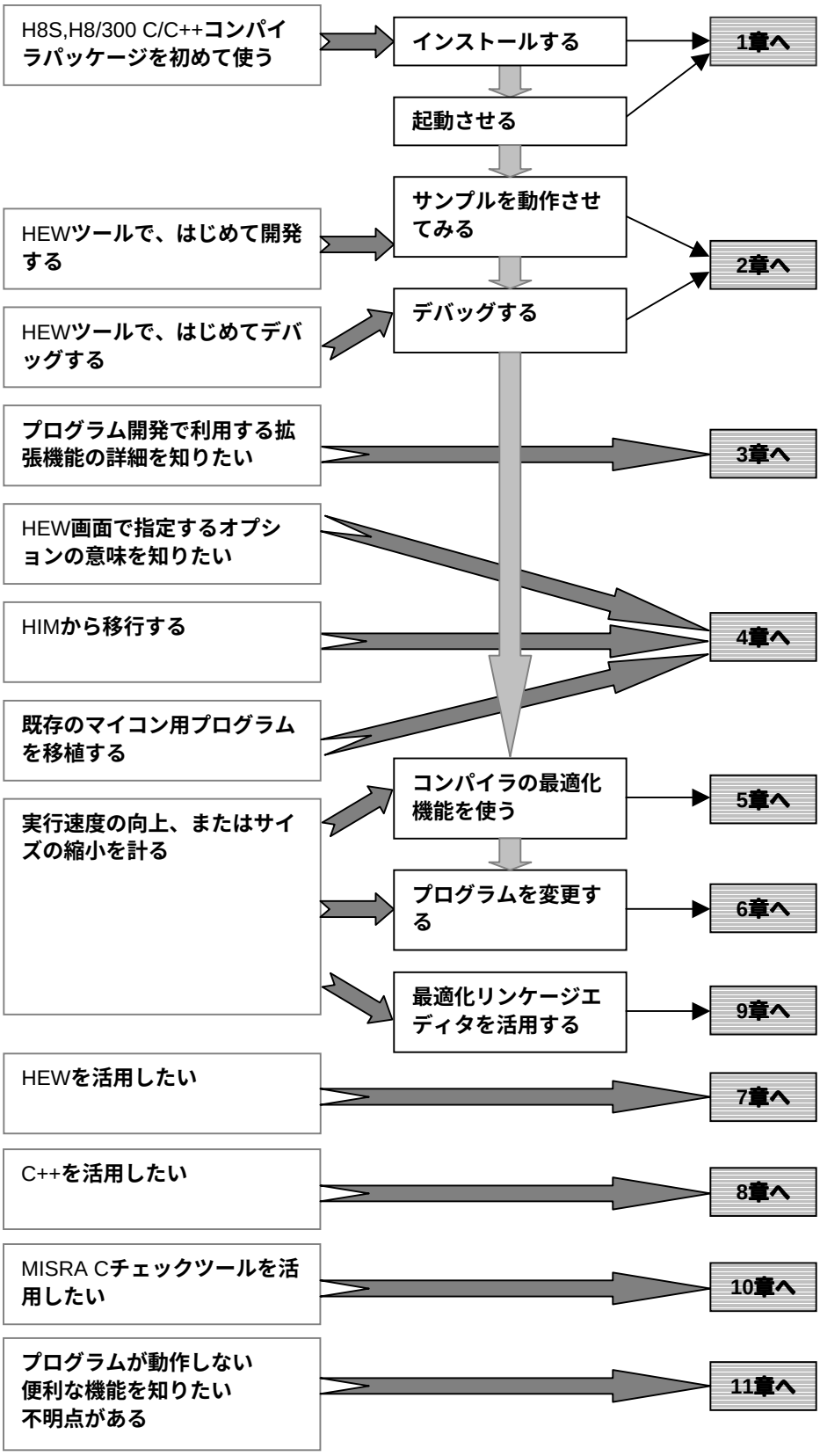
本アプリケーションノートは、HEW3.0 と H8 コンパイラバージョン 6.0 を中心に記載されておりますが、HEW1.2 と H8 コンパイラバージョン 3.0 など、操作方法が異なる場合は、別途区別して記載しております。

〈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。
[Menu->Menu Option]　：太字と->はメニューオプションを示します。

本アプリケーションノートの見方

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勧めします。



目次

1. 概説

1.1 概要

H8S,H8/300 C/C++コンパイラは、機器組み込み用シングルチップマイコンルネサステクノロジ H8S,H8/300 シリーズの機能・性能を活かしたプログラムを、C 言語または 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。

以下の CPU をサポートしています。
H8SX シリーズ（以下、H8SX と略す）
H8S/2600 シリーズ（以下、H8S/2600 と略す）
H8S/2000 シリーズ（以下、H8S/2000 と略す）
H8/300H シリーズ（以下、H8/300H と略す）
H8/300 シリーズ（以下、H8/300 と略す）
H8/300L シリーズ（以下、H8/300L と略す）
AE5 シリーズ（以下、AE5 と略す）

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。
また、本書ではコンパイラバージョン 6.0（HEW2.0 以降）を中心に説明しておりますが必要に応じて旧バージョン 3.0（HEW1.2）についても説明しております。

1.2 特長

H8S,H8/300 C/C++コンパイラパッケージの特長を以下に示します。

［Windows®版］
統合環境 HEW（High-performance Embedded Workshop）をサポートしており、Windows®画面上から一貫したプログラム開発が可能です。
HEW の主な機能は以下のとおりです。

- プロジェクトジェネレータ
  →各CPUごとの雛型ソフトウェアプロジェクトを自動生成できます。
- 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です。
- ネットワーク対応
  →WindowsNT®のCSS環境下での開発が可能です。

［UNIX 版］
統合化マネージャ IDM（Integrated Development Manager）をサポートしました。
これによって，エディタからデバッグまで連結した開発作業が可能となります。
以下に主な機能を示します。

- コンパイル／アセンブルエラー時にエディタを起動できます。
  （ソース内のエラー行にカーソルを配置）
- アセンブル／コンパイル，オブジェクトモジュールの結合、デバッガへのローディングまでを自動的に実行できます。
- グラフィックユーザインタフェースによるソースレベルデバッグが可能です。

1.3 インストール方法

1.3.1 PC 版

Windows®98、Windows®Me、Windows NT®4.0、Windows®2000 または Windows®XP 対応 H8S,H8/300 C/C++コンパイラパッケージの動作環境、および Windows®98、Windows®Me、Windows NT®4.0、Windows®2000 または Windows®XP 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：日本語Windows®98、Windows®Me、Windows NT®4.0、Windows®2000またはWindows®XPが動作するもの）
- OS：日本語 Windows®98、Windows®Me、Windows NT®4.0、Windows®2000 または Windows®XP
- メモリ容量：128MB 以上を推奨
- ハードディスク容量：統合開発環境：空き容量 100MB 以上（フルインストールに必要な容量）
- Acrobat® Reader：空き容量 10MB 以上
- 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

以下の手順で PC への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。

(a) H8S,H8/300 C/C++**コンパイラパッケージのインストール**

(i) コンパイラパッケージのCD-ROMをCD-ROMドライブに挿入します。（以下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のルートディレクトリにあるSetup.EXE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:¥Setup.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。

統合開発環境のインストールの注意事項：
統合開発環境は半角英数字と半角下線のみからならディレクトリパスにインストールしてください。特に、全角文字や空白のないディレクトリパスを使用してください。

(i) HEW(High-performance Embedded Workshop)をHIM(Hitachi Integration Manager)と同じディレクトリにインストールしないでください。
(ii) ネットワーク上で使用される場合でも、High-performance Embedded Workshopは、各PCのドライブにインストールしてください。ツールチェイン、ライブラリアンインタフェース、HDI、オンラインマニュアルはネットワークドライブにインストール可能です。他のPCでインストールしたツールチェイン、ライブラリアンインタフェースを自分のPCに登録する方法はHigh-performance Embedded Workshop V.4.00 **ユーザーズマニュアル「5章 ツール管理」**を参照してください。
(iii) HEWをインストールした直後に[High-performance Embedded Workshop]がWindows®スタートメニューの[プログラム]の中に表示されない場合はWindows®を再起動してください。
(iv) Windows®98でインストール中にインストーラが異常終了した場合、コンピュータを再起動してから再度インストールしてください。

(b) Acrobat® Reader **のインストール**

(i) コンパイラパッケージの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Ar505jpn.exe（日本語版）または[PDF_read¥English]ディレクトリにあるAr505eng.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¥Ar505jpn.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。

(c) **オンラインマニュアルおよび添付資料の参照**

- オンラインマニュアルをインストールした場合
  Windows®スタートメニューの[プログラム]の中にある[High-performance Embedded Workshop]メニューのPDFファイルOnline Manual[H8S,H8/300]-English(xx xx)（英語版）またはOnline Manual [H8S,H8/300]-Japanese(xx xx)（日本語版）をクリックします。(xx xx)は年月のキーワードを表示しています）
  （例 Online Manual [H8S,H8/300]-Japanese(01 10)）

- オンラインマニュアルをインストールしていない場合
  コンパイラパッケージのCD-ROMをCD-ROMドライブに挿入します。
  （ここでは、仮にDドライブとします）
  Windows®スタートメニューの[ファイルを指定して実行…]をクリックします。
  CD-ROMの[Manuals]ディレクトリにある、jH8_xxxx.PDF（日本語版）またはeH8_xxxx.PDF（英語版）(xxxxは年月のキーワードを表示しています)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:¥Manuals¥jH8_0110.PDF）、[OK]をクリックします。

1.3.2 UNIX 版

H8S,H8/300 C/C++コンパイラ パッケージを 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**

CD-ROM 1 枚で提供いたします。

(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。

(a) **コンパイラパッケージのインストール**

コンパイラパッケージのインストール手順を以下に示します。

(i) コンパイラパッケージ用パスの作成
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。
（以下説明ではインストールディレクトリを/usr/cross_softとします）
% mkdir△/usr/cross_soft (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/h8s_sparc△/cdrom/h8s_sparc (RET)
[HP-UXの場合]
% mount△/dev/dsk/c201d2s0△/cdrom (RET)

(iii) コンパイラパッケージのコピー
作成パスに移動して、提供CD-ROM から(i)で作成したパスにH8S,H8/300 C/C++コンパイラパッケージのソフトウェア一式を解凍します。

[Solarisの場合]
% cd△/usr/cross_soft (RET)
% tar△xvf△/cdrom/h8s_sparc/Program.tar (RET)
[HP-UXの場合]
% cd△/usr/cross_soft (RET)
% tar△xvf△/cdrom/"PROGRAM.TAR;1" (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**「H8S,H8/300 C/C++コンパイラ ユーザーズマニュアル」**をご覧ください。
Cシェルの場合の例を示します。

% setenv△CH38△/usr/cross_soft (RET)

システムインクルードファイルの格納場所を指定します。

% setenv△CH38TMP△/usr/tmp (RET)

コンパイラ、またはモジュール間最適化で作成する中間ファイル格納ディレクトリを指定します。（ここでは/usr/tmpとしています）
指定がない場合は、カレントディレクトリに中間ファイルを作成します。

% setenv△H38CPU△**:** (RET)**

CPU/動作モードを指定します。指定できるCPUは2000n、2000a、2600n、2600a、
300hn、300ha、300、300lです。
また、CPUが2000a、2600a、300haの場合はアドレス空間サイズを指定することが可能です。
（例 **% setenv H38CPU 2600a:24(RET)**）

% setenv△HLNK_TMP△/usr/tmp (RET)

リンケージエディタ、またはモジュール間最適化で作成する中間ファイル格納ディレクトリを指定します。（ここでは/usr/tmpとしています）
指定がない場合は、カレントディレクトリに中間ファイルを作成します。

% setenv△HLNK_LIBRARY1△/usr/cross_soft/****.lib (RET)**
% setenv△HLNK_LIBRARY2△/usr/cross_soft/****.lib (RET)**

リンク時にLIBRARYオプション、サブコマンドオプションを使用せずに暗黙のうちにライブラリを入力することができます。
詳細はH8S、H8/300**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。

(v) CD-ROMをアンマウントします。

[Solarisの場合]
% umount△/cdrom/h8s_sparc (RET)
[HP_UXの場合]
% umount△/cdrom (RET)

(b) 統合化マネージャ、および統合化マネージャ用定義ファイルのインストール

統合化マネージャをインストールします。
(i) CD-ROM上のtarfileよりインストーラを読み込みます。（CD-ROMドライバ装置名を/cdromとします）

% tar△xvf△/cdrom/idm.tar△idm_install (RET) [Solarisの場合]

(ii) インストーラを起動します。

% idm_install (RET)

その後は、画面の指示に従い、インストールしていきます。インストール方法の詳細はH8S,H8/300定義ファイルインストーラを参照してください。

(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 Windows®98、Windows®Me、Windows NT®4.0、Windows®2000、または Windows®XP 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。

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) 統合開発環境の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Ar40jpn.exe（日本語版）または[PDF_read¥English]ディレクトリにあるAr40eng.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¥Ar40jpn.exe）、[OK]をクリックします。
画面に表示されるインストールの指示に従います。

(d) オンラインマニュアルの参照

- オンラインマニュアルをインストールした場合
Windows®スタートメニューの[プログラム]の中にある[High-performance Embedded Workshop]メニューのPDFファイルOnline Manual[H8S,H8/300]-English(xx xx)（英語版）またはOnline Manual [H8S,H8/300]-Japanese(xx xx)（日本語版）をクリックします。（(xx xx)は年月のキーワードを表示しています）
（例 Online Manual [H8S,H8/300]-Japanese(01 10)）

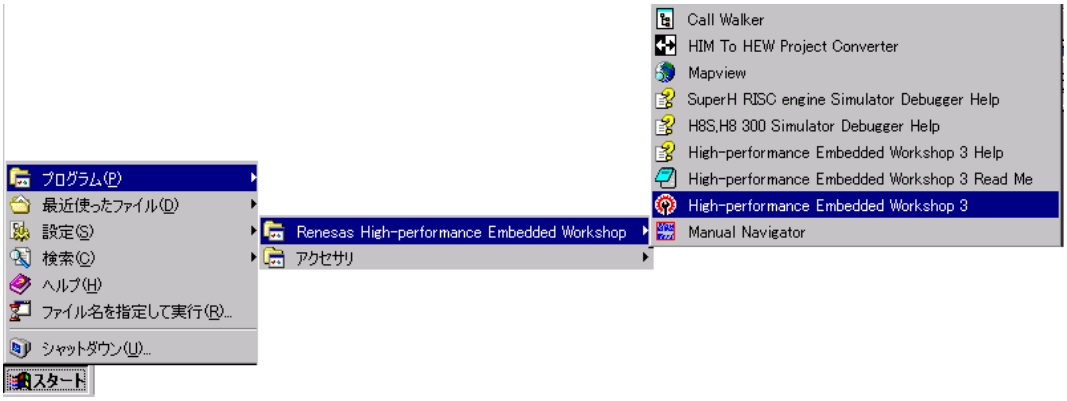
- オンラインマニュアルをインストールしていない場合

(i) 統合開発環境のCD-ROMをCD-ROMドライブに挿入します。
（ここでは、仮にDドライブとします）
(ii) Windows®スタートメニューの[ファイルを指定して実行…]をクリックします。
(iii) CD-ROMの[Manuals]ディレクトリにある、jH8_xxxx.PDF（日本語版）またはeH8_xxxx.PDF（英語版）(xxxxは年月のキーワードを表示しています)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:¥Manuals¥jH8_0110.PDF）、[OK]をクリックします。

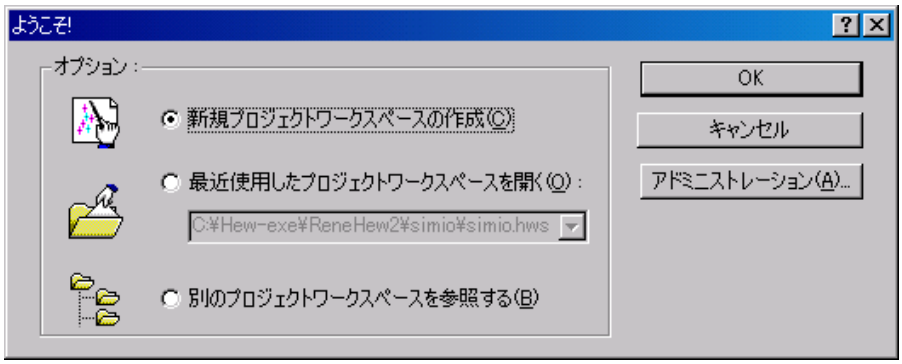
1.4 起動方法

1.4.1 HEW の起動方法

H8S,H8/300 C/C++コンパイラパッケージ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の下のツール群に名称がありますので、この中から実行プログラムを起動させてください。



すると、下記のようなようこそ！ダイアログボックスが表示されます。



上記の画面から、プロジェクトワークスペースを選択します。

新規プロジェクトワークスペースの作成	**新規にワークスペースを作成します。**
最近使用したプロジェクトワークスペースを開く	**最近使用した既存のワークスペースを開きます。**
別のプロジェクトワークスペースを参照する	**その他のワークスペースを開きます。**

[アドミニストレーション]を選択すると、使用するシステムツールの登録や削除を行います。

1.4.2 コマンド上での起動方法

本節では、H8S,H8/300 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、**H8S、H8/300 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「2. コンパイラ操作方法」**を参照してください。

以下、コンパイラの基本的な操作方法を説明します。

(1) コンパイラの起動

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。

ch38 (RET)

(2) プログラムのコンパイル

C ソースプログラム「test1.c」をコンパイルします。
ch38△test1.c (RET)

C++ソースプログラム「test2.cpp」をコンパイルします。
ch38△test2.cpp(RET)

複数のソースプログラムを一度にコンパイルします。
ch38△test1.c△test2.cpp(RET)

(3) オプション指定方法

オプション(goptimize、debug、show=object,allocation 等)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。
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。
ch38△-goptimize△-debug△-show=object,allocation△test1.c (RET)

オプションは短縮形を用いることができます。
ch38△-g△-deb△-sh=o,a△test1.c (RET)

複数のプログラムをコンパイルするときはオプション位置で有効範囲が異なります。

＜ソースプログラムにすべて有効なオプション指定例＞
最初のソースプログラムより前に指定したオプションはすべてのソースプログラムに有効です。
ch38△-g△-deb△-sh=o,a△test1.c△test2.cpp (RET)

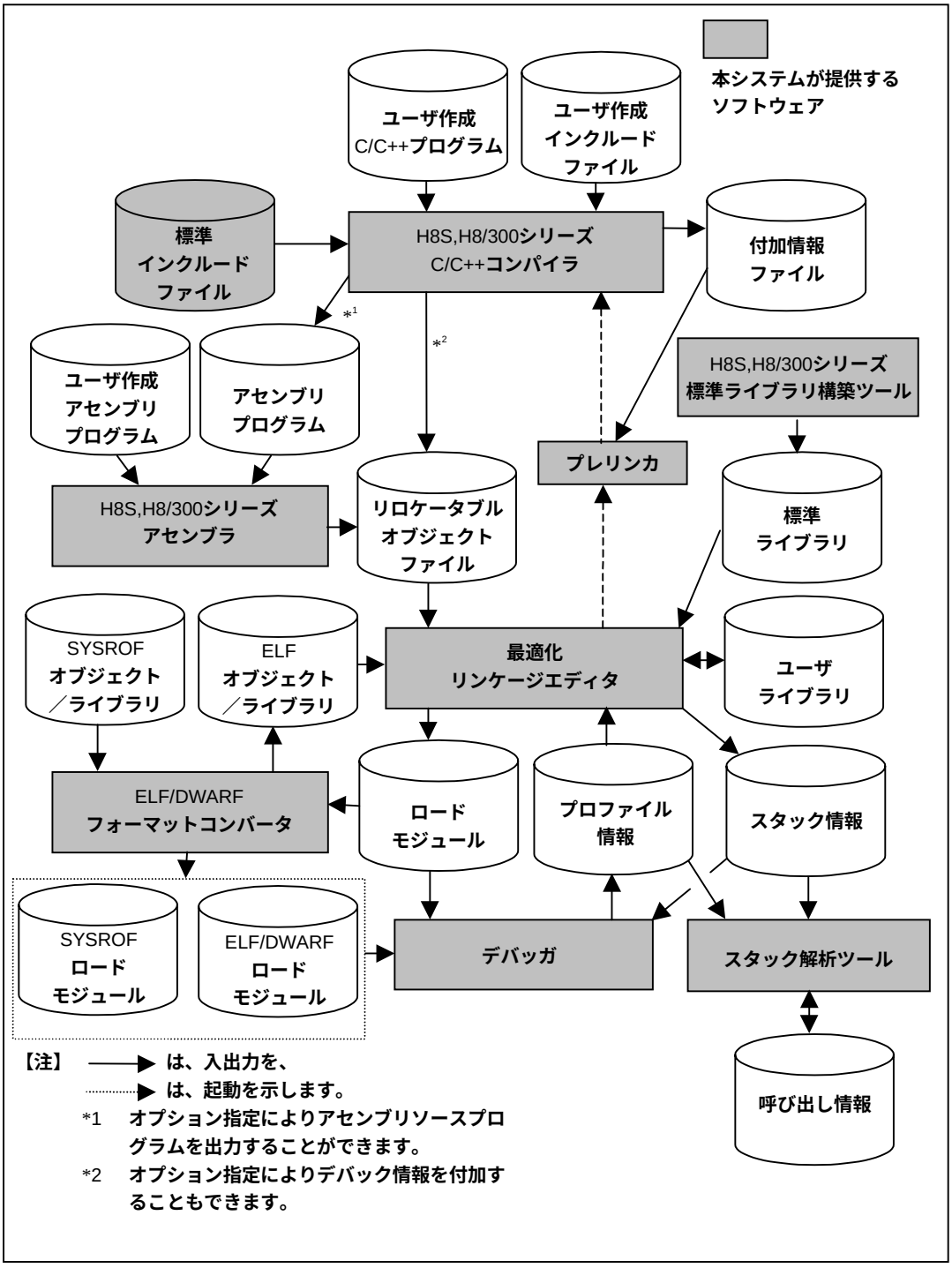
＜各ソースプログラムごとに有効なオプション指定例＞
ソースプログラム直後のオプション指定は、直前のソースプログラムだけに有効です。
ch38△test1.c△test2.cpp△-deb△-sh=o,a (RET)

【注意事項】

(1) **コンパイラは、C 言語、C++言語の区別を**-lang **オプションとファイルの拡張子、または** lang **オプションにより行います。ファイルの拡張子の詳細については** H8S、H8/300 **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「**8. **ファイル仕様」をご参照ください。**

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示します。



2. プログラムの作成とデバッグまでの手順

2.1 プロジェクト構築

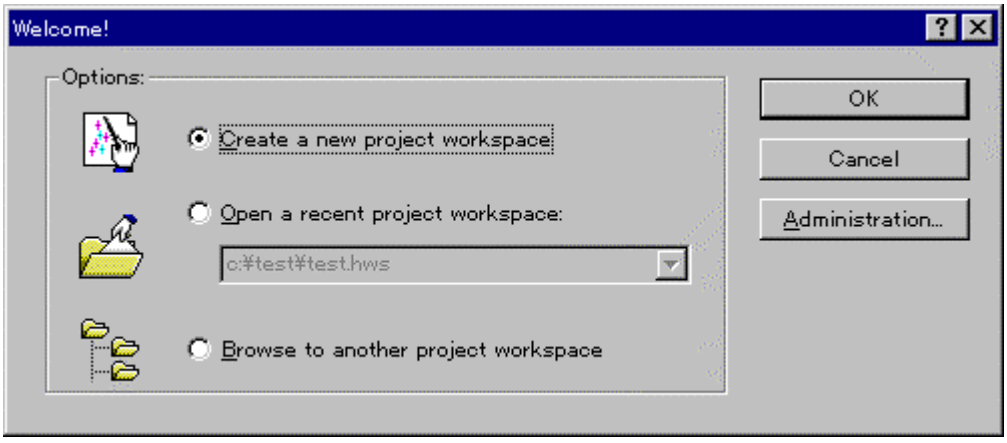
ロードモジュールの作成方法は、それぞれの作業環境、HEW のバージョンによって異なります。
次の中から選択し、手順にそって作成してください。

○HEWで新規にプロジェクトを作成する1（HEW1.2）	2.1.1へ
○HEWで新規にプロジェクトを作成する2（HEW2.0）	2.1.2へ
○HEWを使用せず、コマンドラインで作成する	2.1.3へ
○HIMからHEWへ移行する	7.1.8へ
○既存ファイルを流用してHEWでプロジェクトを作成する	4.3へ

なお、「2.3　HDI を使用したデバッグ方法」は HEW で新規に作成したプロジェクトを使用して説明いたします。

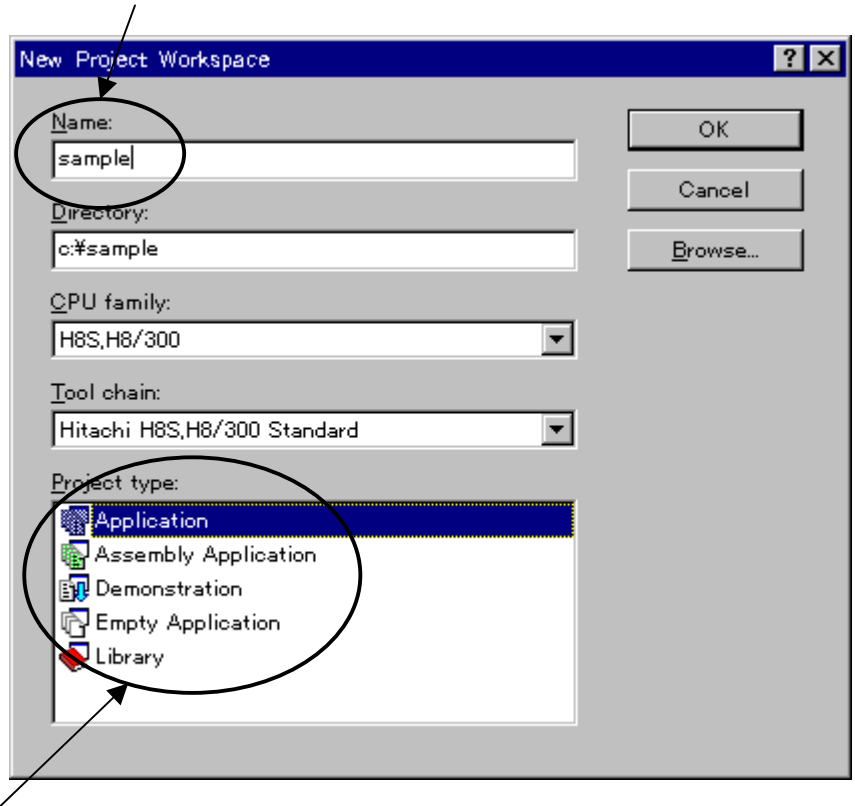
2.1.1 ワークスペースの新規作成 1（HEW1.2）

新規にプロジェクトワークスペースを作成する場合は Welcome！ダイアログボックスから Create a new project workspace を選択します。



(1) Project type の設定

次の画面がでたら、まず、**Name** 欄にプロジェクト名を記入します。



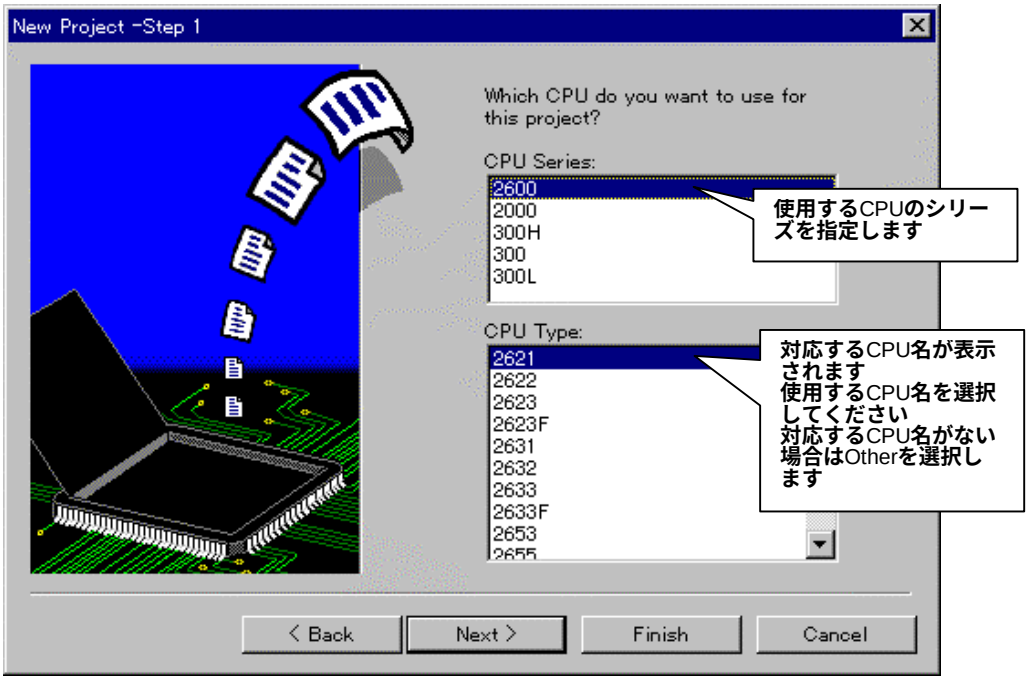
その後、**Project type:**欄を選択します。

Project type	内容
Application	C/C++**プログラムファイルを含む場合のアプリケーション作成プロジェクトタイプ**
Assembly Application	**アセンブリプログラムだけで作成する場合のアプリケーション作成プロジェクトタイプ**
Demonstration	**サンプルプロジェクトタイプ**
Empty Application	**空のプロジェクト作成**
Library	**ライブラリ作成時プロジェクトタイプ**

Project type を選択後、[OK]ボタンをクリックすると新規プロジェクトの初期設定のためのステップに進みます。
ここでは、Project type として Application を選択した場合を説明します。

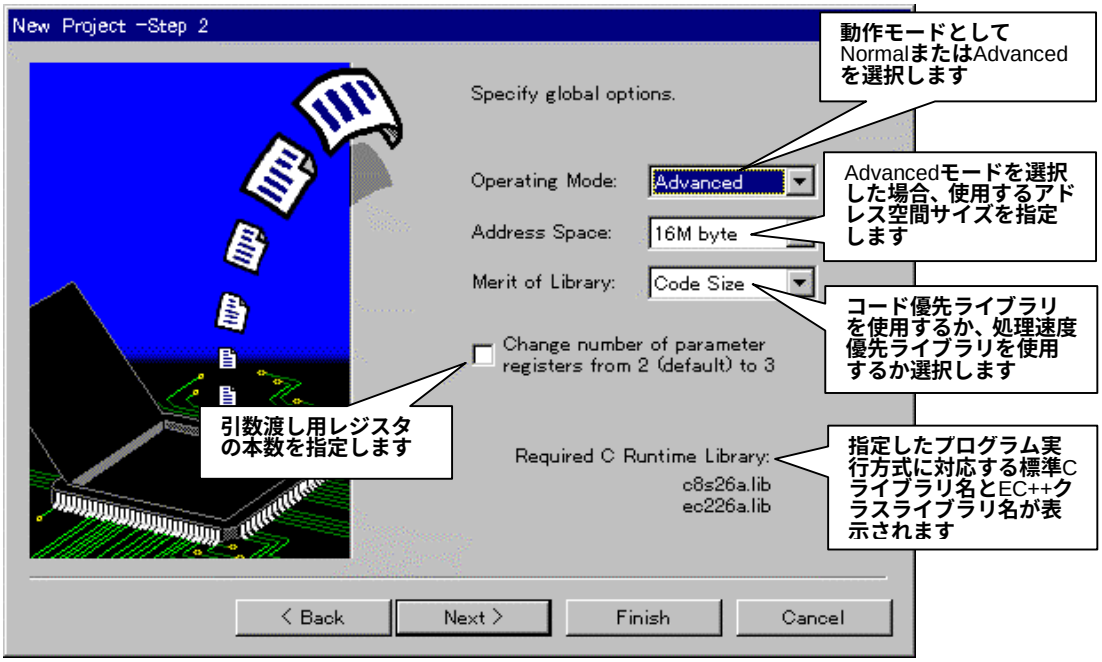
(2) **新規プロジェクト**-Step1

CPU を指定して NEXT> を押下します。



(3) **新規プロジェクト**-Step2

グローバルオプションを指定して NEXT> を押下します。



グローバルオプションは、すべてのプロジェクトファイルをとおして、一貫していなければなりません。
グローバルオプションは次のとおりです。

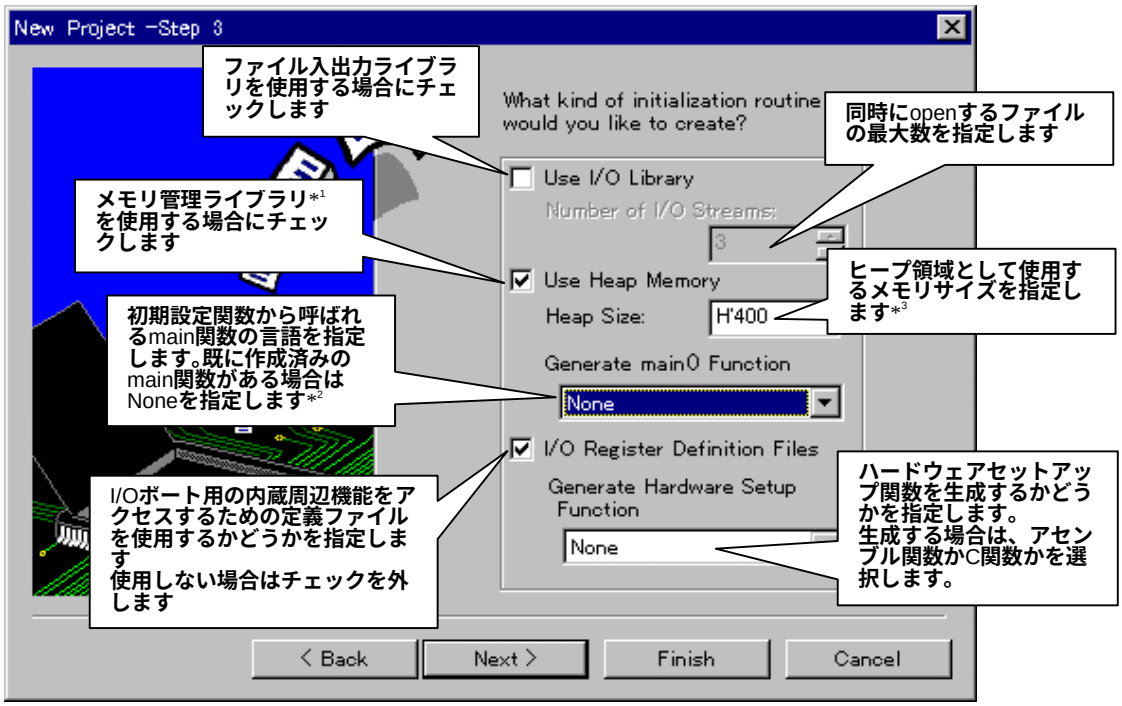
- CPU 種別
- 引数渡し用レジスタ数指定

ここで指定したグローバルオプションを、新規プロジェクトの初期設定終了後に変更する場合は、リンクする標準ライブラリの指定も変更する必要があります。

グローバルオプションと標準ライブラリの変更方法は、**「**11.2.1 Undefined external symbol **が出力される」**にある表を参照してください。

(4) **新規プロジェクト**-Step3

初期設定プログラムの内容を指定して NEXT> を押下します。



【注】 *1 **メモリライブラリ関数は** malloc、realloc、calloc、new **を指します。**

*2 **ここでは** main **関数を生成しないでください。**(9)**で「**2.3 HDI **を使用したデバッグ方法」の準備として、**main **関数を含むサンプルプログラムを追加します。**

*3 **ヒープ領域のサイズは次の計算式で求めます。**

> **（ヒープ領域サイズ）≧**
> **（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）**

管理領域サイズは次のようになります。

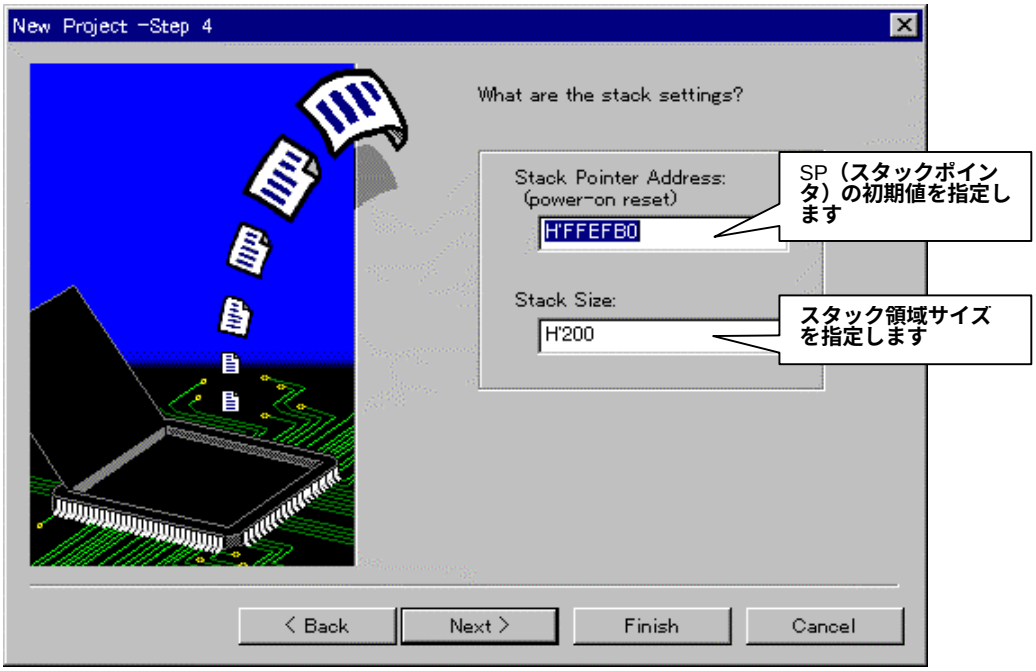
CPU **種別**	**管理領域サイズ**
H8S/2600 ADV,H8S/2000 ADV, H8/300H ADV	16 **バイト**
H8S/2600 NRM,H8S/2000 NRM,H8/300H NRM,H8/300	8 **バイト**

ADV:**アドバンストモード**、NRM:**ノーマルモード**

ここで指定したヒープ領域のサイズを新規プロジェクトの初期設定終了後に変更する方法は、**「**2.2.1(2) **ヒープ領域の確保」**を参照してください。

(5) **新規プロジェクト**-Step4

スタックを設定して NEXT> を押下します。



スタックサイズは次のようにして求めます。

各関数の呼び出し関係における呼び出しレベルの一番深いネストについて、スタック領域のサイズを計算します。その最大値がスタック領域サイズです。

たとえば、関数の呼び出しレベルが一番深い場合が以下の場合、全部のスタックサイズを加算します。

main**関数（スタックサイズ**10**バイト）→** func**関数（**20**バイト）→** sub**関数（**30**バイト）**

この場合は、スタックサイズが 60 バイトとなります。

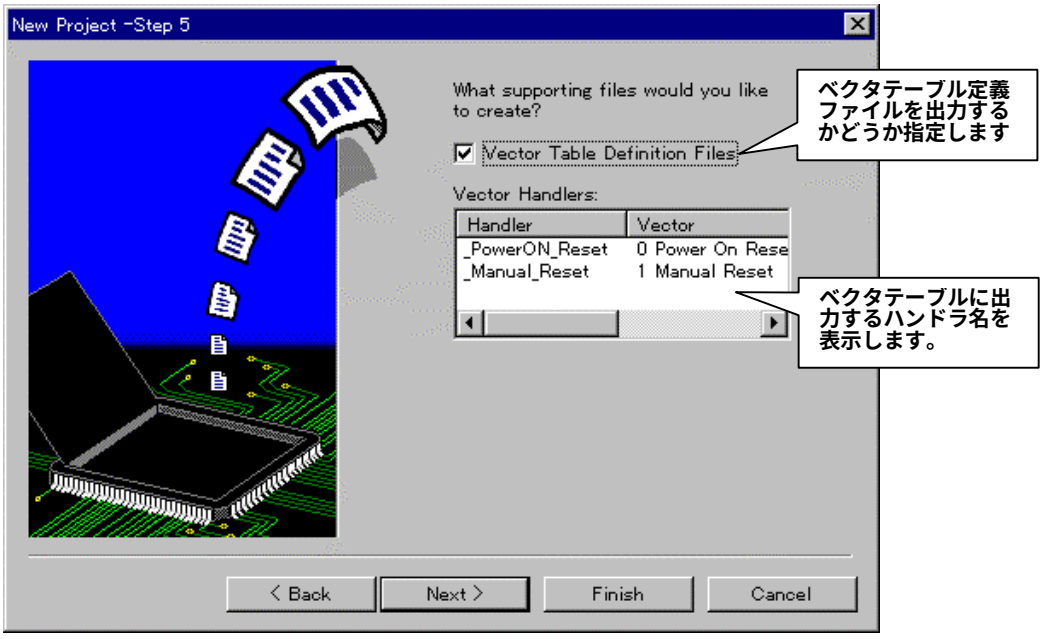
それぞれの関数のスタックサイズはオブジェクトリストファイル出力指定時にシンボル割り付け情報出力を指定すると出力されます。

また、実行時ルーチンが呼び出された場合は H8S,H8/300 **シリーズ** C/C++**コンパイラ 添付資料**の**「標準ライブラリのスタック使用量一覧」**を参照してください。

ここで指定したスタックサイズを新規プロジェクトの初期設定終了後に変更する方法は、**「**2.2.1(8) **スタックサイズの設定」**を参照してください。

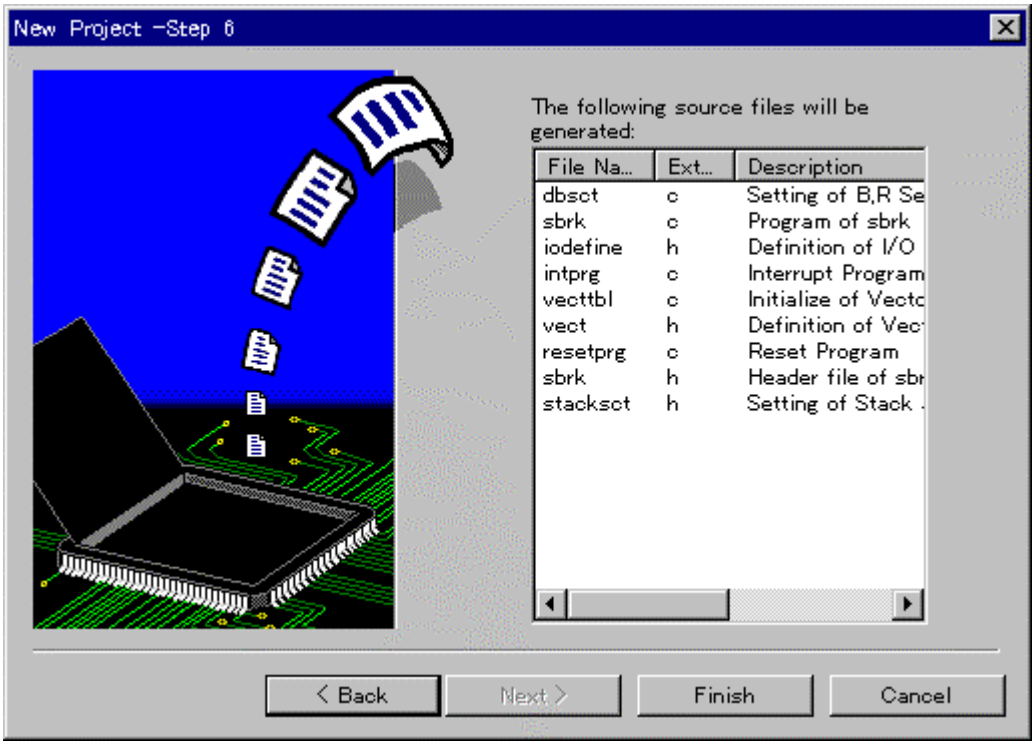
(6) **新規プロジェクト**-Step5

ベクタテーブルの設定内容を指定して NEXT> を押下します。



(7) **新規プロジェクト**-Step6

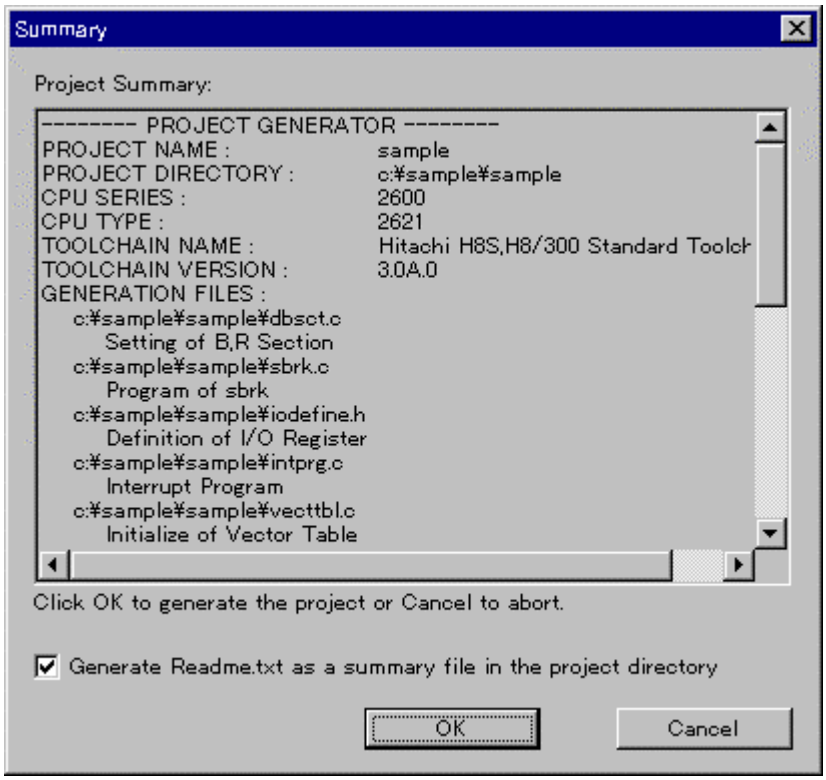
プロジェクトジェネレータで作成したファイルが表示されます。その後 Finish を押下します。



ここで作成したファイルの詳細については、**「2.2 サンプルプログラムの作成」**を参照してください。

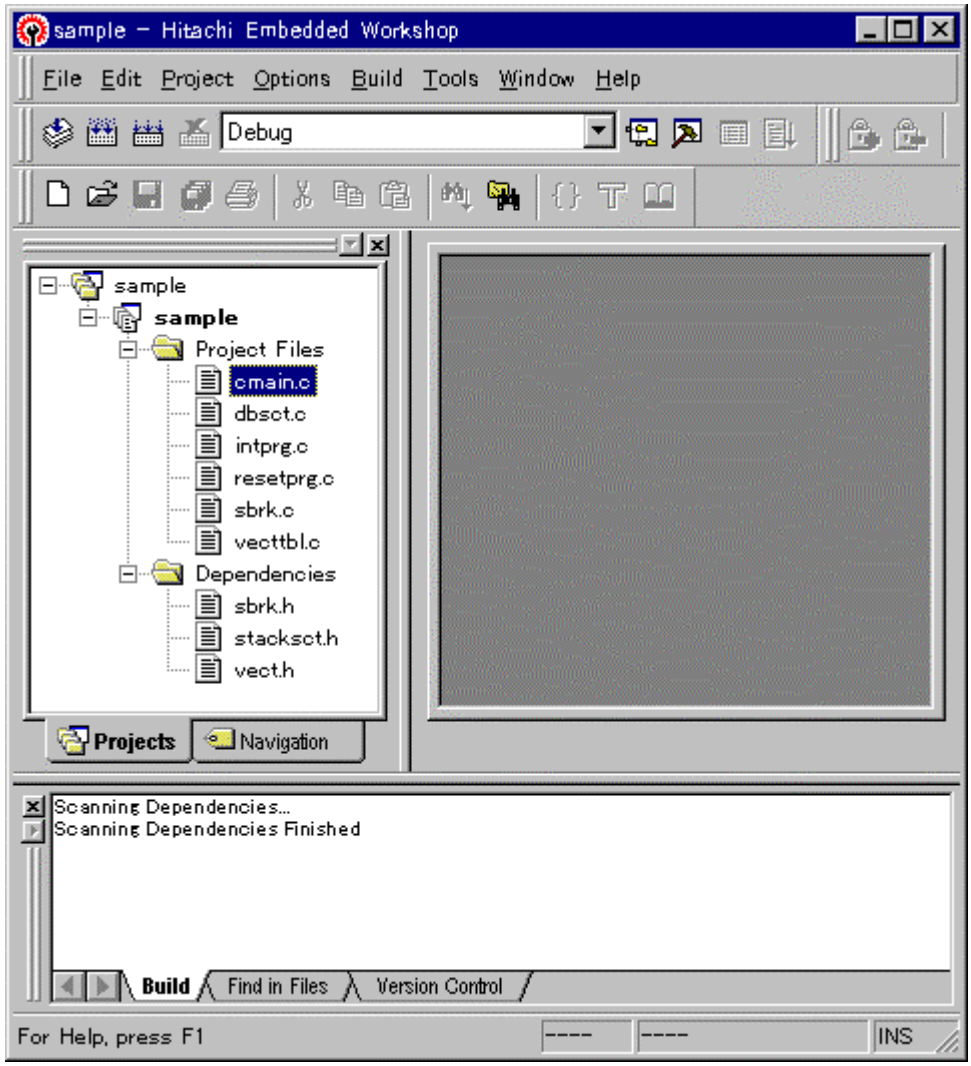
(8) **新規プロジェクト**-Step7

Finish を指定すると以下の画面が表示されます。



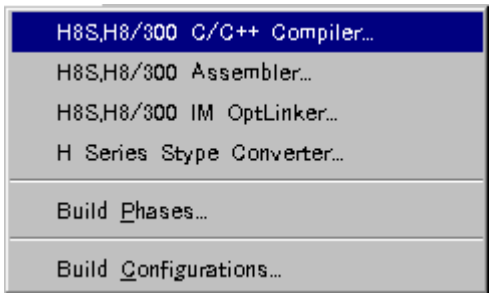
(9) main ファイルの追加

でき上がったプロジェクトに、main 処理を行うファイル cmain.c を追加します。
[Project->Add Files…]で、（HEW ディレクトリ）¥Tools¥HITACHI¥H8¥3_0a_0¥sample ¥cmain.c を指定します。



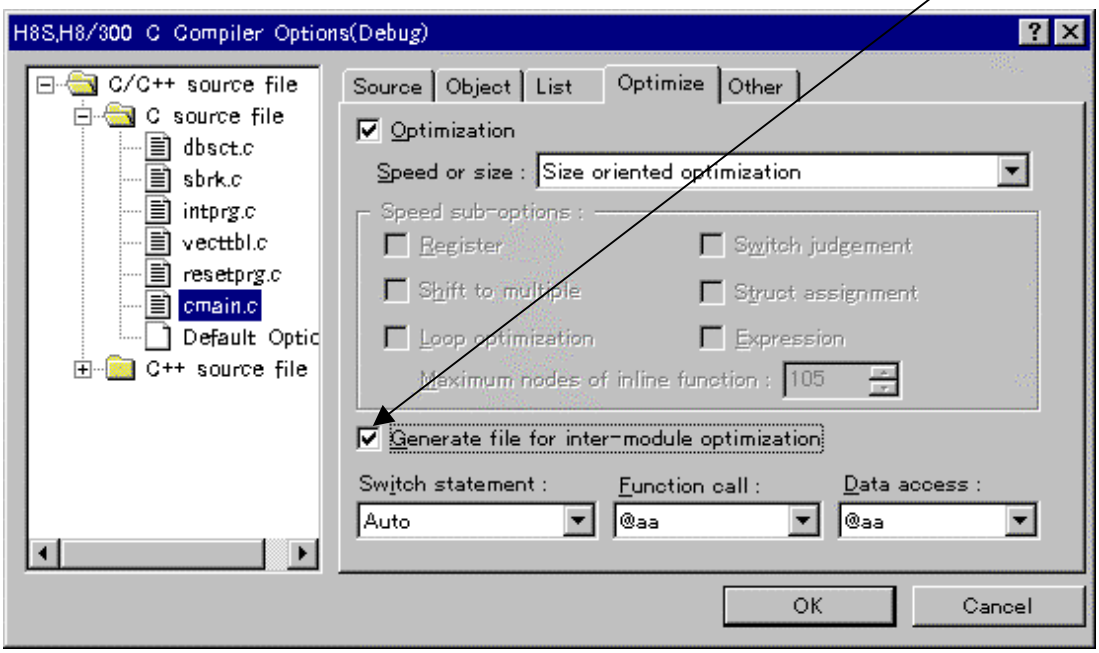
(10) オプションの設定

Options メニューから次のように H8S,H8/300 C/C++ Compiler…を選択します。

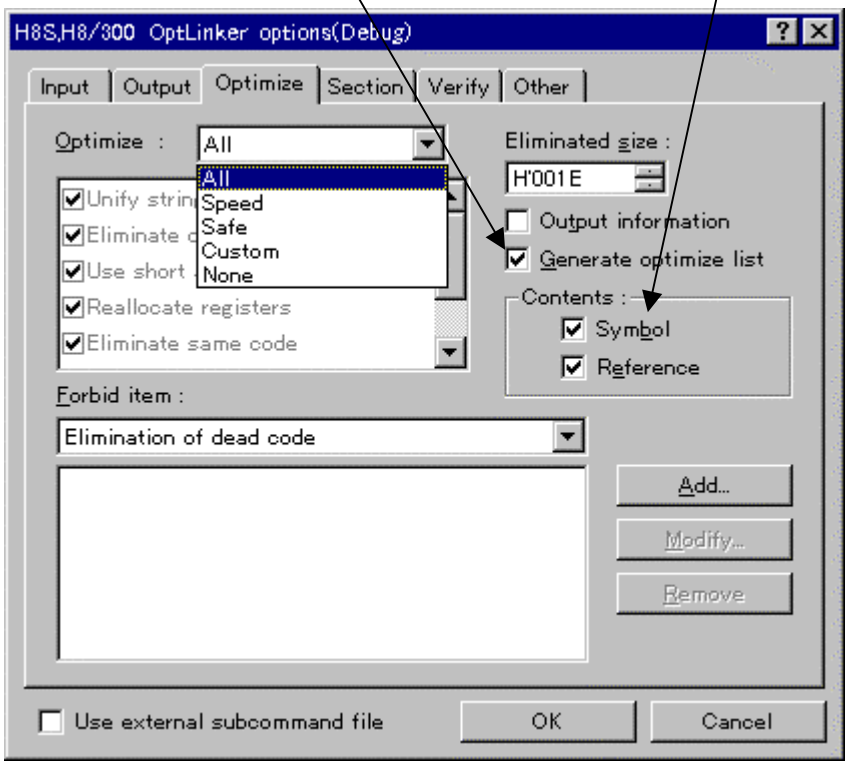


この画面で cmain.c のコンパイラオプションを指定します。

ここではモジュール間最適化ツール用付加情報ファイルの出力を指定します。cmain.c を指定して[ここ]をチェックします。

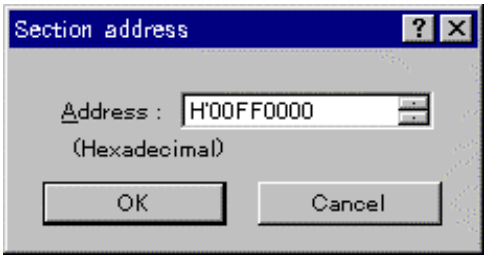


次に[Options->H8S,H8/300 IM Optlinker…]を選択して、モジュール間最適化ツールのオプションを指定します。
まず、モジュール間最適化機能としてすべての機能が有効になるように All を Optimize タブにて指定します。
また、このタブで最適化情報リストの出力をここで指定します。
さらに、このリストへのシンボル最適化情報の出力と、シンボル参照回数の出力をここで指定します。

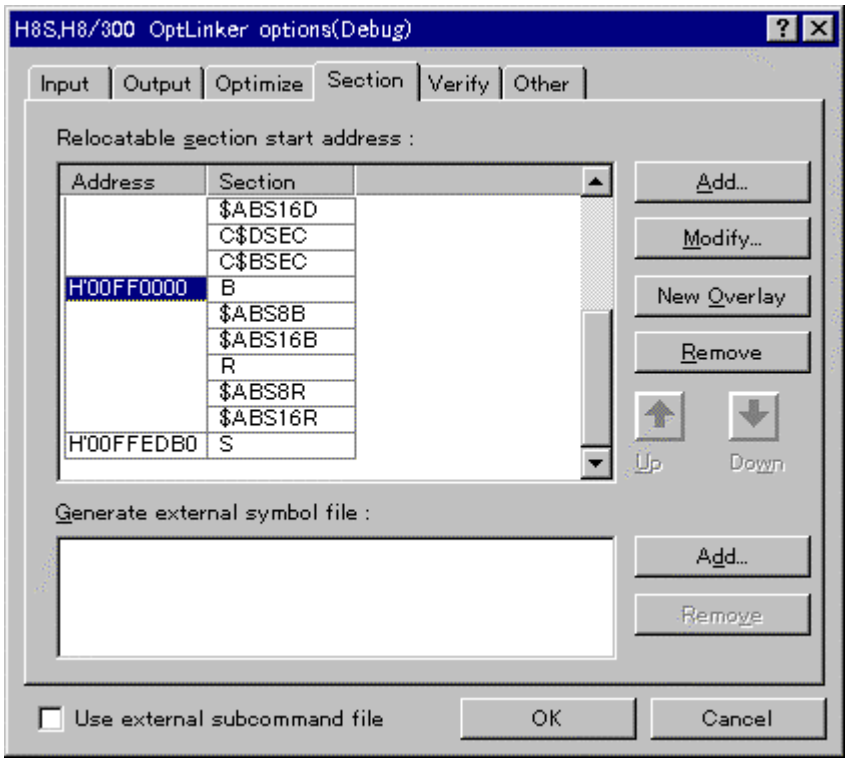


次にリンク時のファイルの割り付け方を section タブにて指定します。

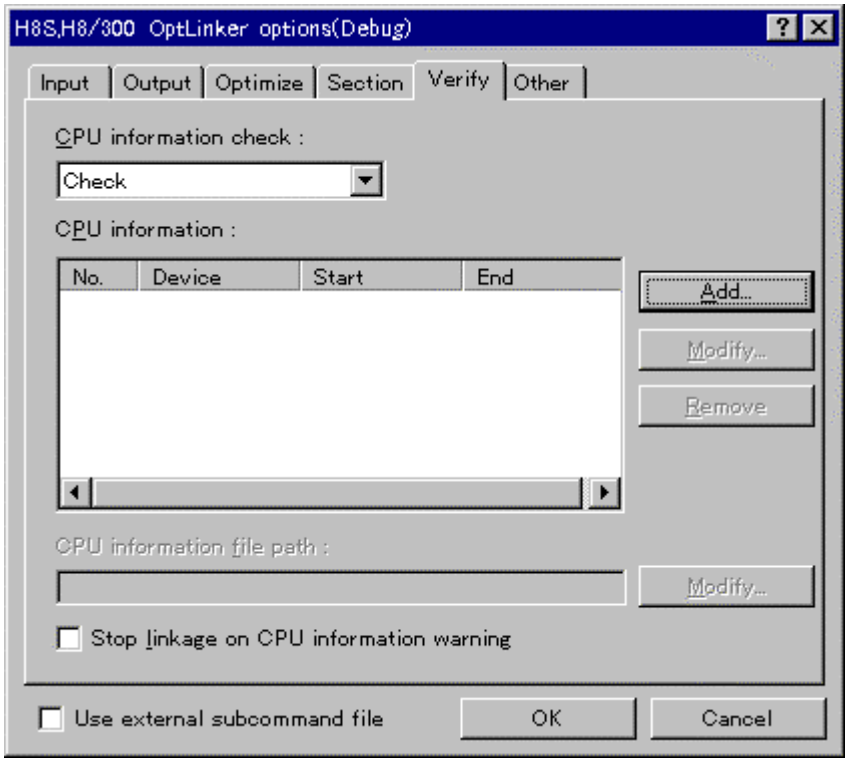
ここでは B セクションが割り付けられているセクションのアドレスを H'00FF0000 に変更します。Address 欄をクリックし、Modify ボタンを押して指定します。



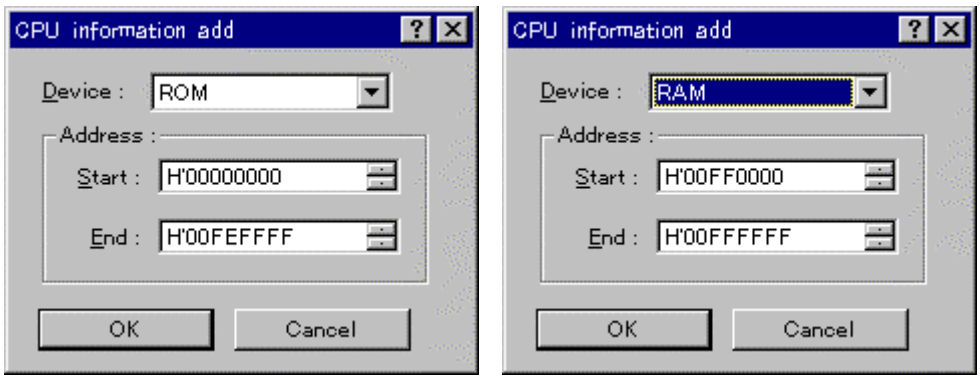
次のようになります。



次に CPU 割り付けのチェックとして、Verify タブにて CPU 情報を作成します。

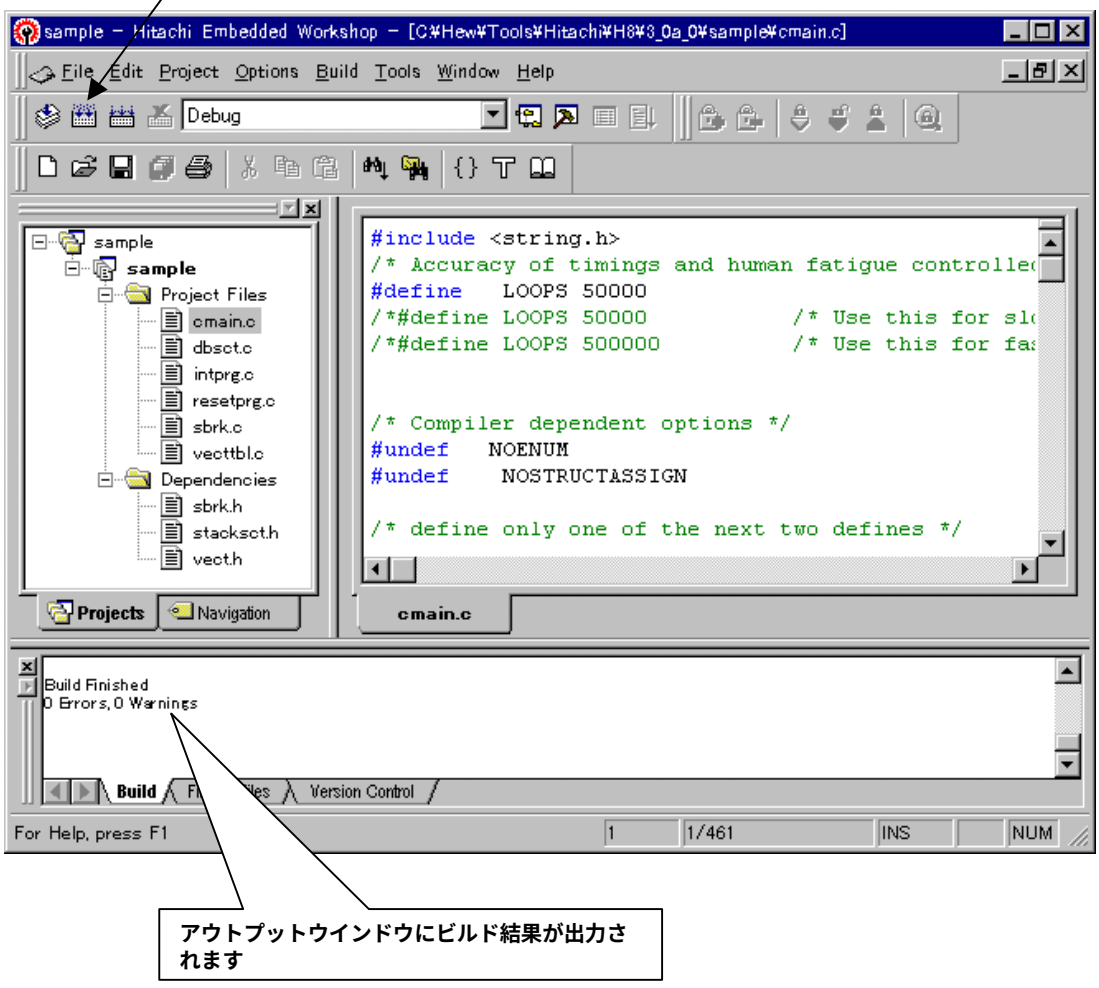


CPU information check:を Check にすると、CPU 情報のチェックが可能になります。
[Add…]ボタンを押すと次のダイアログボックスが出てくるので、ROM と RAM の領域をここで次のように指定します。



(11) ビルドの実行

ビルドを実行してロードモジュールを作成します。
コマンドボタンでもここをクリックするとビルドが可能です。

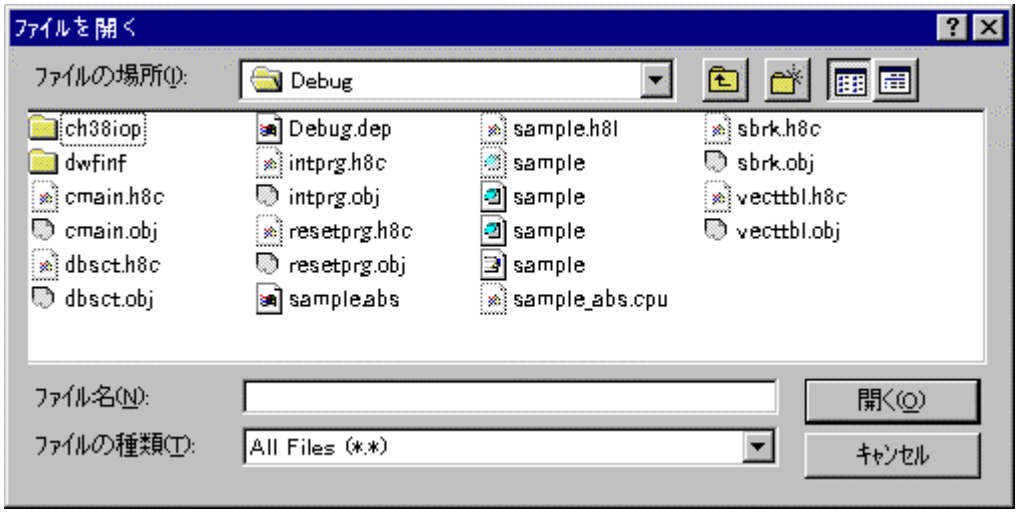


(12) **生成ファイルの確認**

ビルドが終わると次のように各ファイルが作成されます。

アブソリュートロードモジュールはプロジェクトディレクトリの下にプロジェクト名と同じ名前のディレクトリが作成されますが、その下の debug ディレクトリの中に sample.abs という形で作成されます。

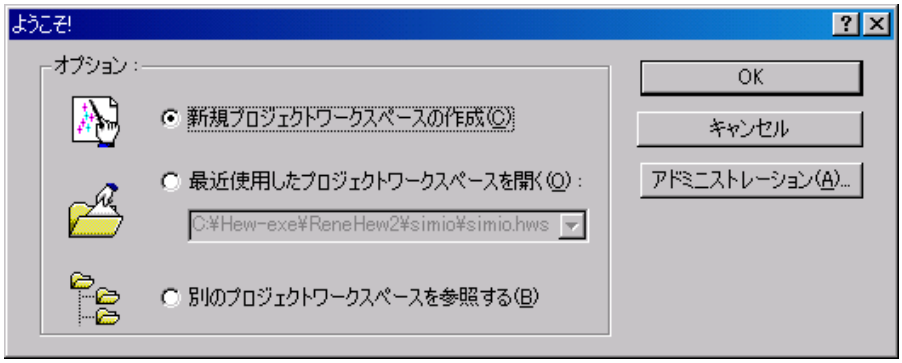
また、ビルドの際に生成された、マップファイル、最適化情報リストファイルも同じディレクトリに作成されるので、こちらを[File->Open]で開いて確認することができます。



マップファイルは sample.map、最適化情報リストファイルは sample.lop という名称で作成されます。

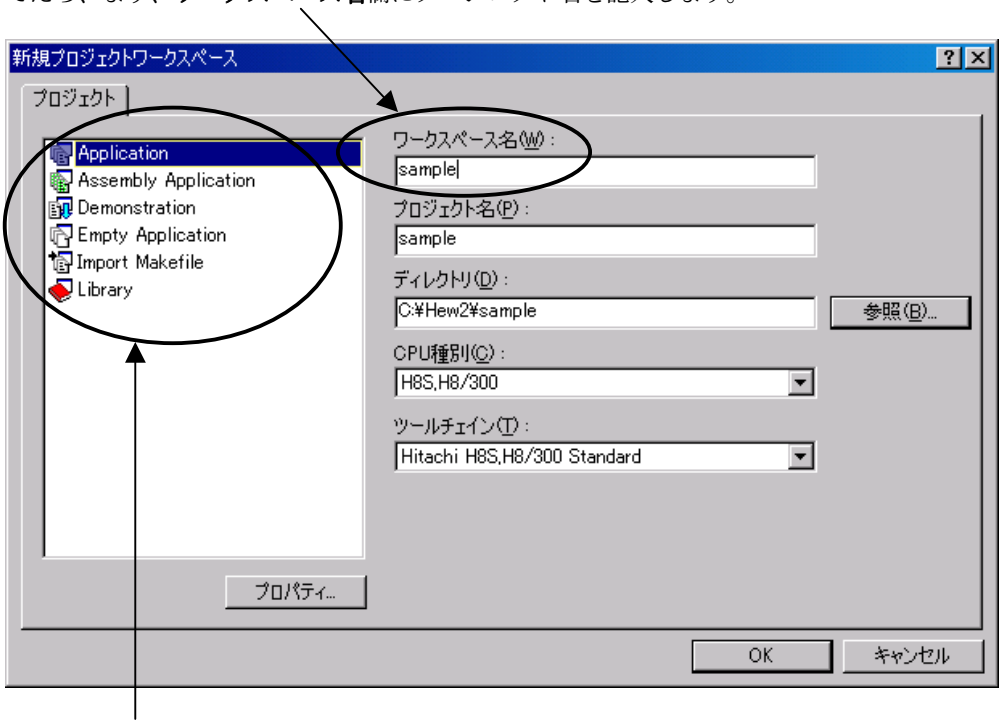
2.1.2 ワークスペースの新規作成 2（HEW2.0 以降）

新規にプロジェクトワークスペースを作成する場合はようこそ！ダイアログボックスから新規プロジェクトワークスペースを選択します。



(1) Project type の設定

次の画面がでたら、まず、**ワークスペース名**欄にプロジェクト名を記入します。



その後、**プロジェクトタイプ**を選択します。

プロジェクトタイプ	内容
Application	**C/C++プログラムファイルを含む場合のアプリケーション作成プロジェクトタイプ**
Assembly Application	**アセンブリプログラムだけで作成する場合のアプリケーション作成プロジェクトタイプ**
Demonstration	**サンプルプロジェクトタイプ**
Empty Application	**空のプロジェクト作成**
Library	**ライブラリ作成時プロジェクトタイプ**

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“ sample ”と入力すると、[プロジェクト名]も“ sample ”になり、[ディレクトリ]も“ c:¥hew2¥ sample ”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。

プロジェクトタイプを選択後、[OK]ボタンをクリックすると新規プロジェクトの初期設定のためのステップに進みます。

ここでは、プロジェクトタイプとして Application を選択した場合を説明します。

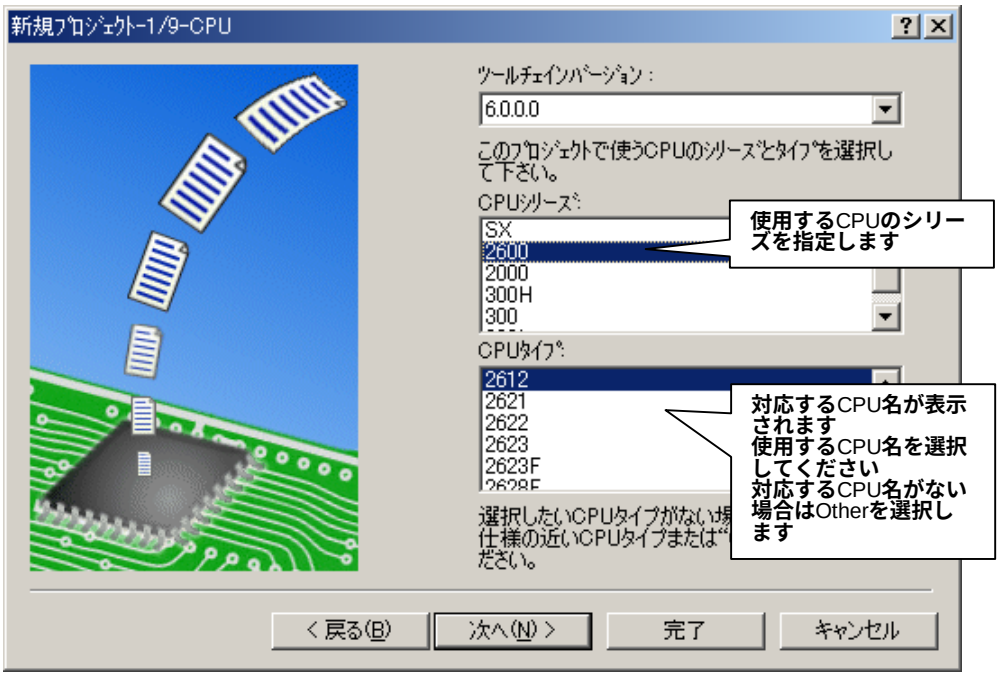
(2) **新規プロジェクト**-1/9

CPU を指定して 次へ> を押下します。

“ 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 >］をクリックすると、次の画面を表示します。
- ［< 戻る］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］、［次へ >］、［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。



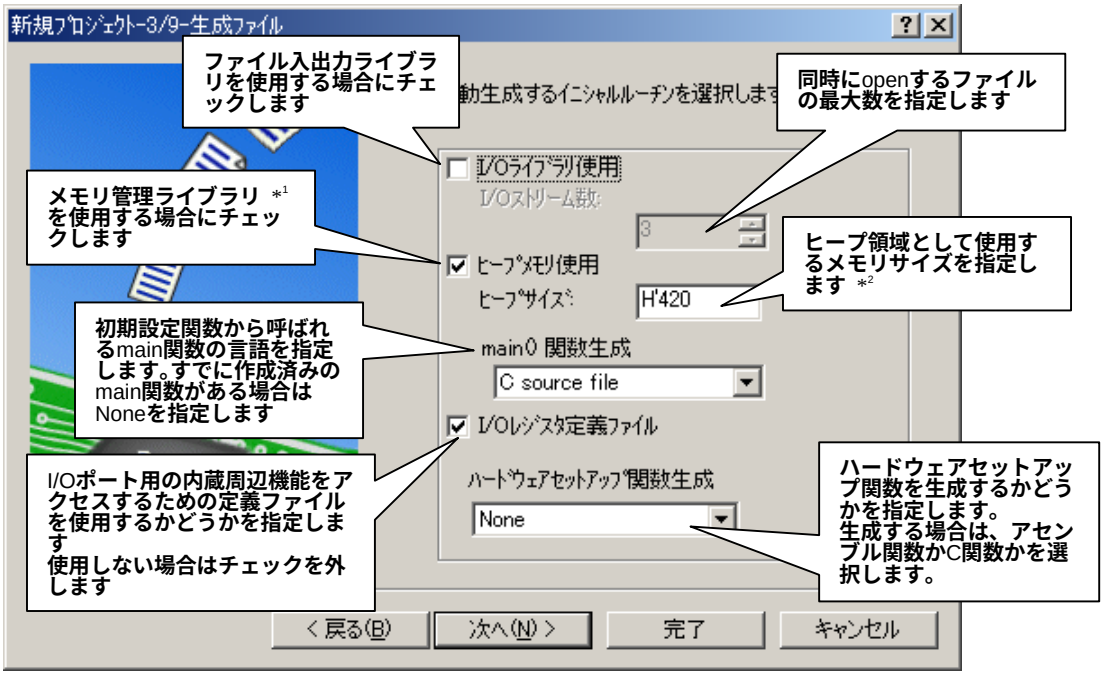
(3) 新規プロジェクト-2/9

グローバルオプションを指定して 次へ> を押下します。

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、1/9 画面で選択した CPU シリーズにあわ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-> H8S,H8/300 Standard Toolchain]の[CPU タブ]で変更できます。

(4) **新規プロジェクト**-3/9

初期設定プログラムの内容を指定して 次へ> を押下します。



【注】 *1 **メモリライブラリ関数は** malloc、realloc、calloc、new **を指します。**

*2 **ヒープ領域のサイズは次の計算式で求めます。**

（ヒープ領域サイズ）≧
（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）

管理領域サイズは次のようになります。

CPU **種別**	**管理領域サイズ**
H8S/2600 ADV,H8S/2000 ADV, H8/300H ADV	16 **バイト**
H8S/2600 NRM,H8S/2000 NRM,H8/300H NRM,H8/300	8 **バイト**

ADV:**アドバンストモード**、NRM:**ノーマルモード**

ここで指定したヒープ領域のサイズを新規プロジェクトの初期設定終了後に変更する方法は、**「**2.2.1(2) **ヒープ領域の確保」**を参照してください。

(5) **新規プロジェクト**-4/9

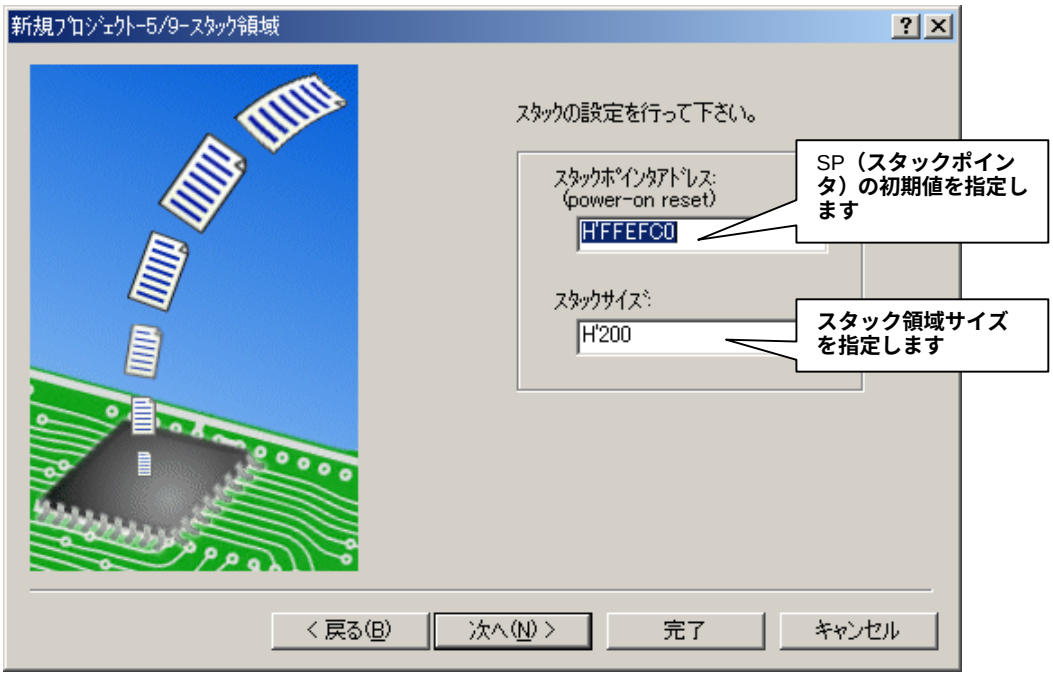
この画面で、C/C++コンパイラで使う標準ライブラリの構成を決定します。また、プロジェクト作成後に標準ライブラリの構成を変更する場合は、HEW の[オプション->H8S,H8/300 Standard Toolchain...]の[Standard Library タブ]で変更できます。

(6) **新規プロジェクト**-5/9

スタックを設定して 次へ> を押下します。

この画面で、スタック領域を設定します。スタック領域として設定する値の初期値は、1/9 画面の［CPU タイプ:］により異なります。

また、プロジェクト作成後にスタックサイズを変更する場合は、HEW の[プロジェクト->構成の編集...]で変更できます。



スタックサイズは次のようにして求めます。

各関数の呼び出し関係における呼び出しレベルの一番深いネストについて、スタック領域のサイズを計算します。その最大値がスタック領域サイズです。

たとえば、関数の呼び出しレベルが一番深い場合が以下の場合、全部のスタックサイズを加算します。

main**関数（スタックサイズ**10**バイト）→** func**関数（**20**バイト）→** sub**関数（**30**バイト）**

この場合は、スタックサイズが 60 バイトとなります。

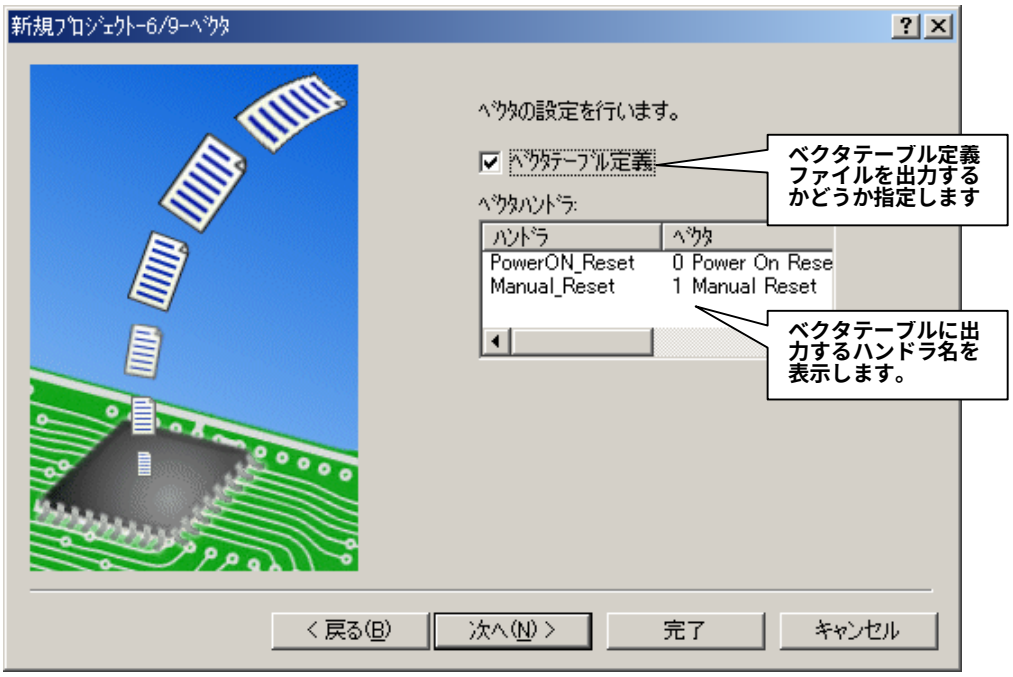
それぞれの関数のスタックサイズはオブジェクトリストファイル出力指定時にシンボル割り付け情報出力を指定すると出力されます。

C/C++プログラム、標準ライブラリの使用するスタック領域は、最適化リンケージエディタの stack オプションを指定してスタック情報ファイルを出力すると、スタック解析ツールを用いて最大使用量を算出することができます。スタック解析ツールの使用方法については、H8S、H8/300 **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「**6. **スタック解析ツール操作方法」**を参照してください。

(7) 新規プロジェクト-6/9

ベクタテーブルの設定内容を指定して 次へ> を押下します。

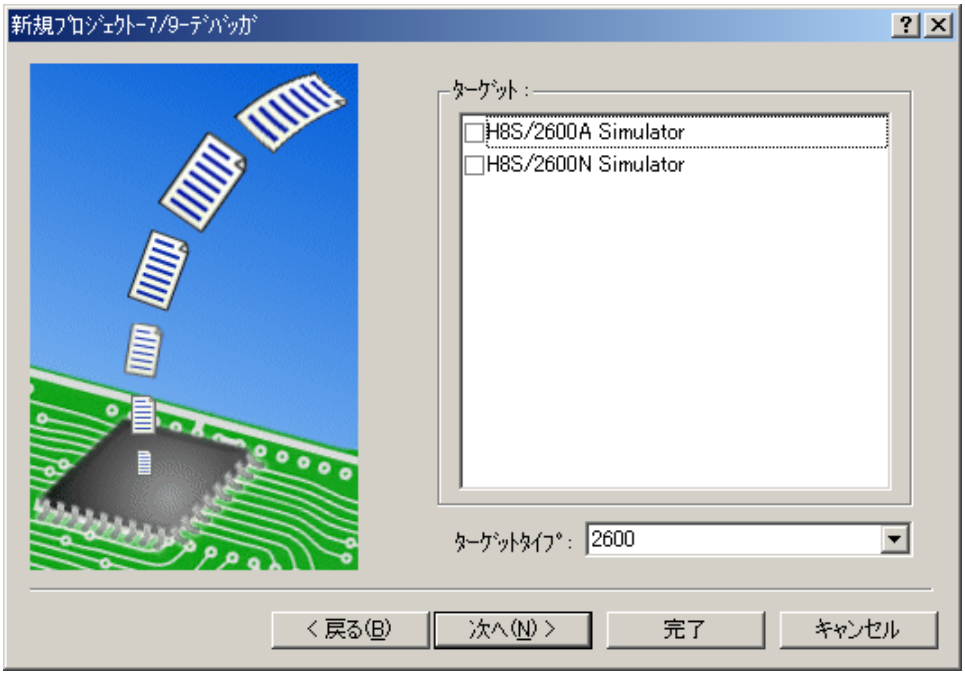
ハンドラプログラムを変更したい場合は、ハンドラプログラム名を選択してクリック後、入力してください。なお、ハンドラプログラムを変更すると、リセットプログラム（reserprg.c）は生成しません。



(8) 新規プロジェクト-7/9

デバッガターゲットを指定して 次へ> を押下します。

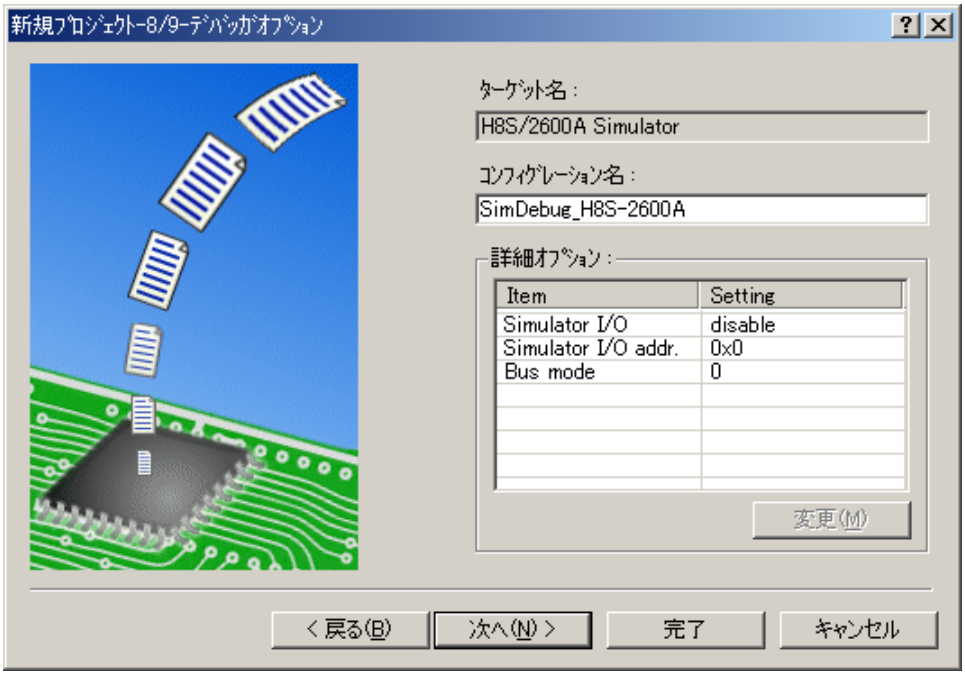
[ターゲット:]から使用するデバッガターゲットを選択（チェック）してください。デバッガターゲットは、未選択でも複数選択してもかまいません。



(9) **新規プロジェクト**-8/9

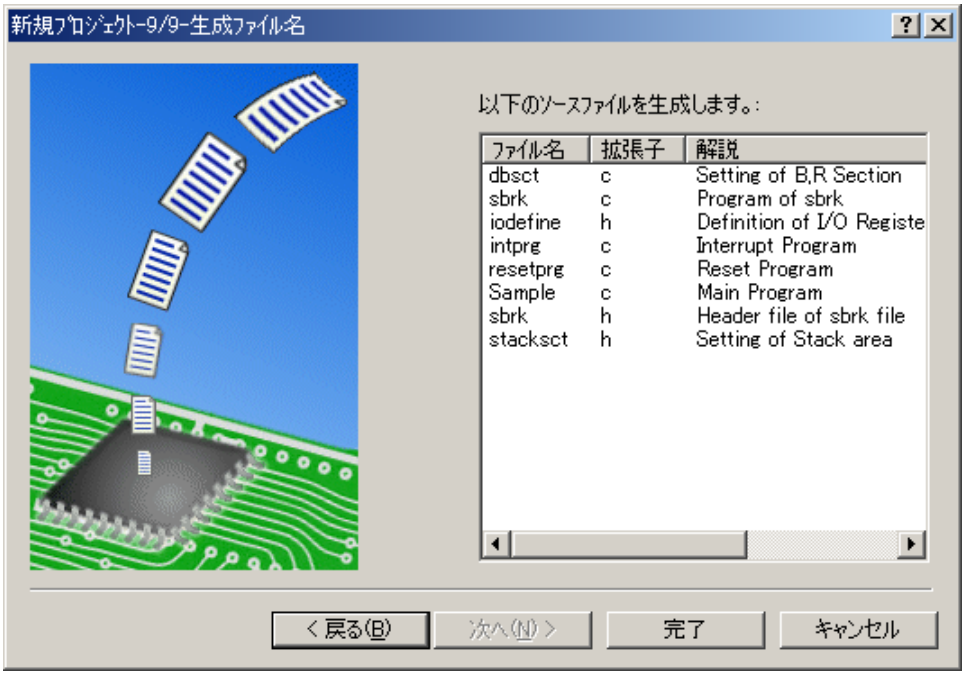
選択したデバッガターゲットのオプションを設定して 次へ> を押下します。

HEW はデフォルトで、“ Release ”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、[コンフィグレーション名:]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、[詳細オプション:]で表示します。変更する場合は、[Item]を選択して[変更]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[変更]はグレーのままです。



(10) **新規プロジェクト**-9/9

プロジェクトジェネレータで作成したファイルが表示されます。その後 完了 を押下します。

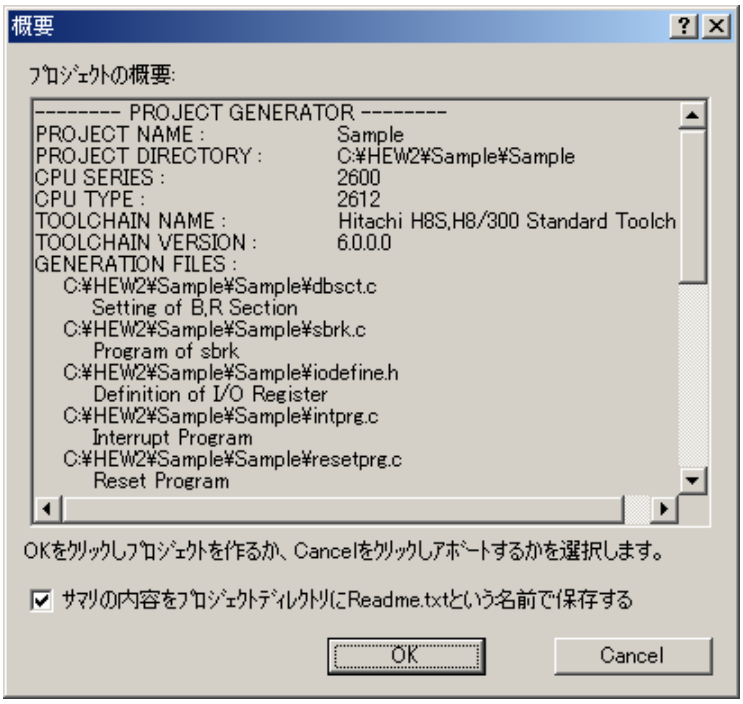


ここで作成したファイルの詳細については、**「2.2 サンプルプログラムの作成」**を参照してください。

(11) 新規プロジェクト-概要

9/9 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概要ダイアログボックスで表示しますので、確認後、[OK]をクリックしてください。

なお、[サマリの内容をプロジェクトディレクトリに Readme.txt という名前で保存する]をチェックすると、概要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。



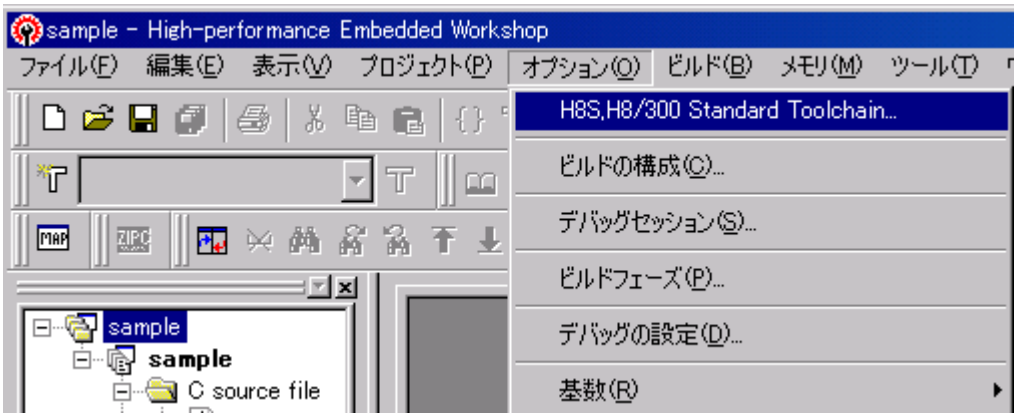
(12) その他

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。

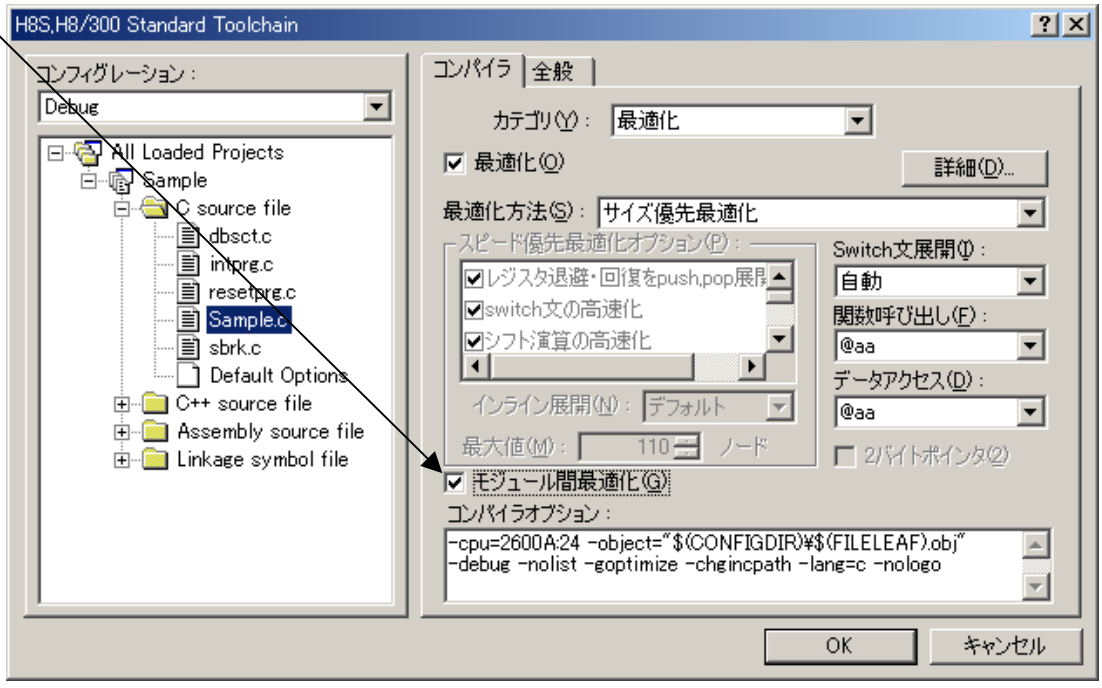
- lowlvl.src （標準入出力サンプルアセンブラリスト）
- lowsrc.c （低水準ライブラリソースファイル）
- lowsrc.h （低水準ライブラリヘッダファイル）

(13) オプションの設定

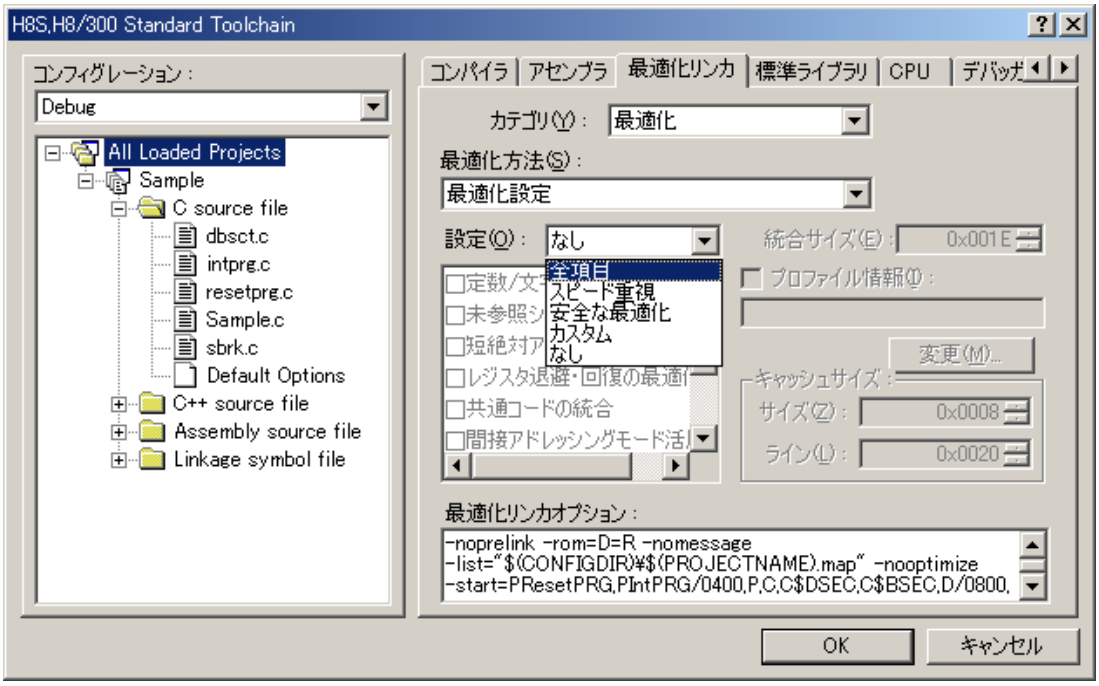
オプションメニューから次のように H8S,H8/300 Standard Toolchain....を選択します。



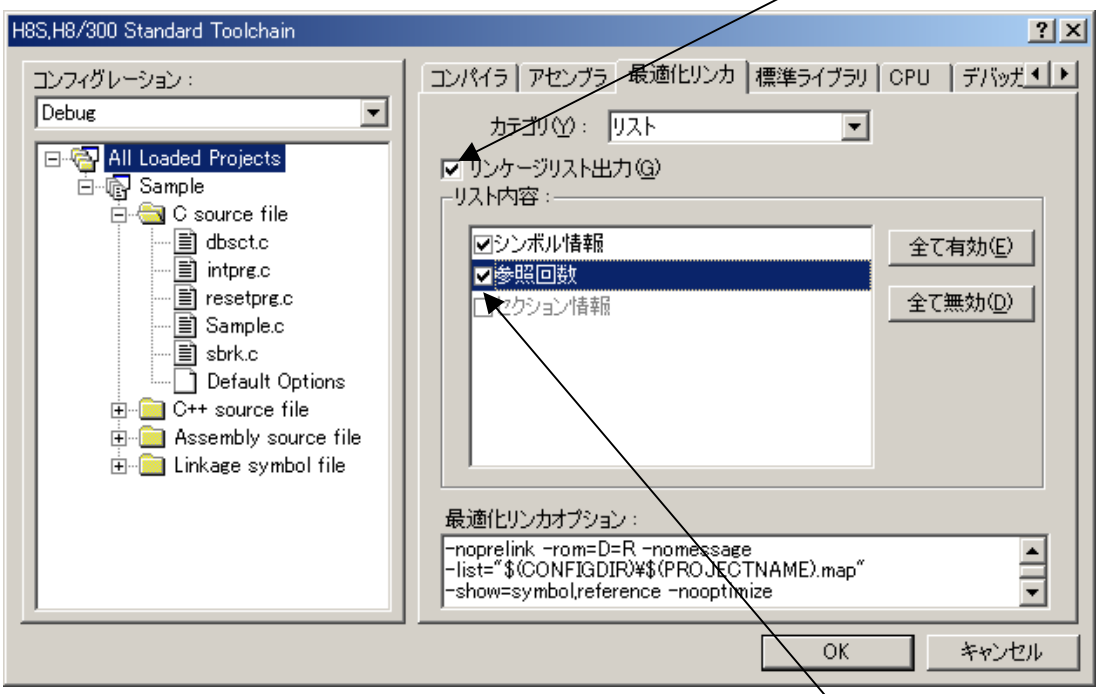
sample.c のコンパイラオプションを指定します。
HEW の[オプション-> H8S,H8/300 Standard Toolchain]の[コンパイラタブ] [カテゴリ/最適化]を選択します。
ここではモジュール間最適化用付加情報の出力を指定します。プロジェクトファイルリストの sample.c を指定して ここ をチェックします。



次にプロジェクトファイルリストの All Loaded Project を指定して[最適化リンカタブ] [カテゴリ/最適化] を選択して、モジュール間最適化のオプションを指定します。
まず、モジュール間最適化機能としてすべての機能が有効になるように全項目を設定にて指定します。



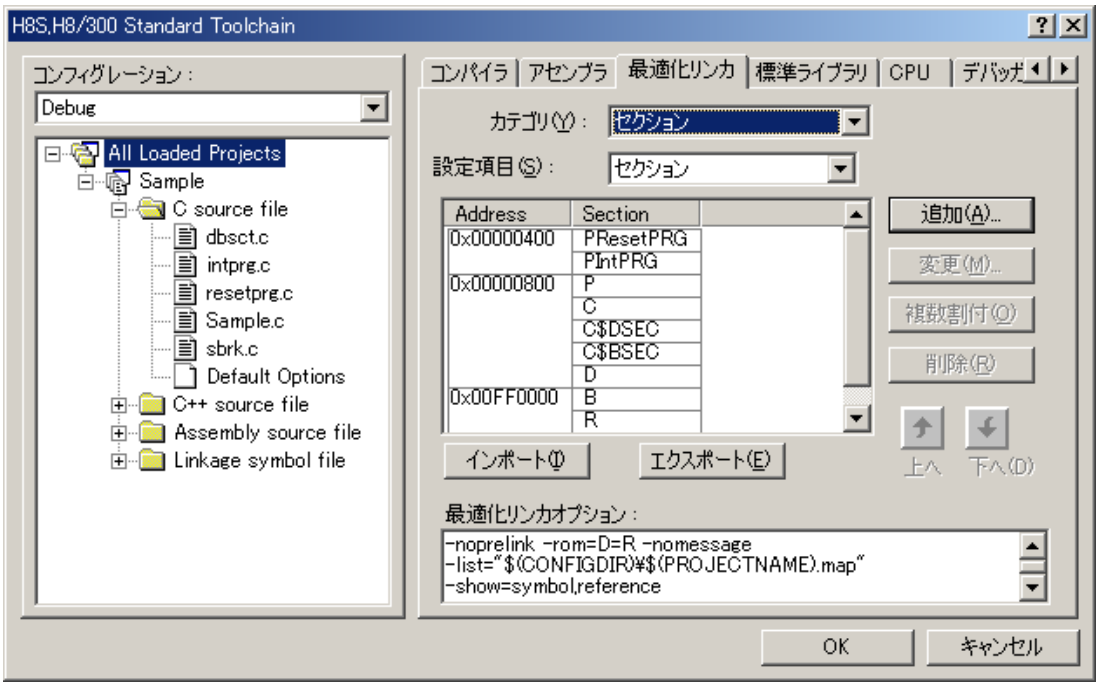
また、[最適化リンカタブ] [カテゴリ/リスト]で最適化情報リストを出力するのにここを指定します。



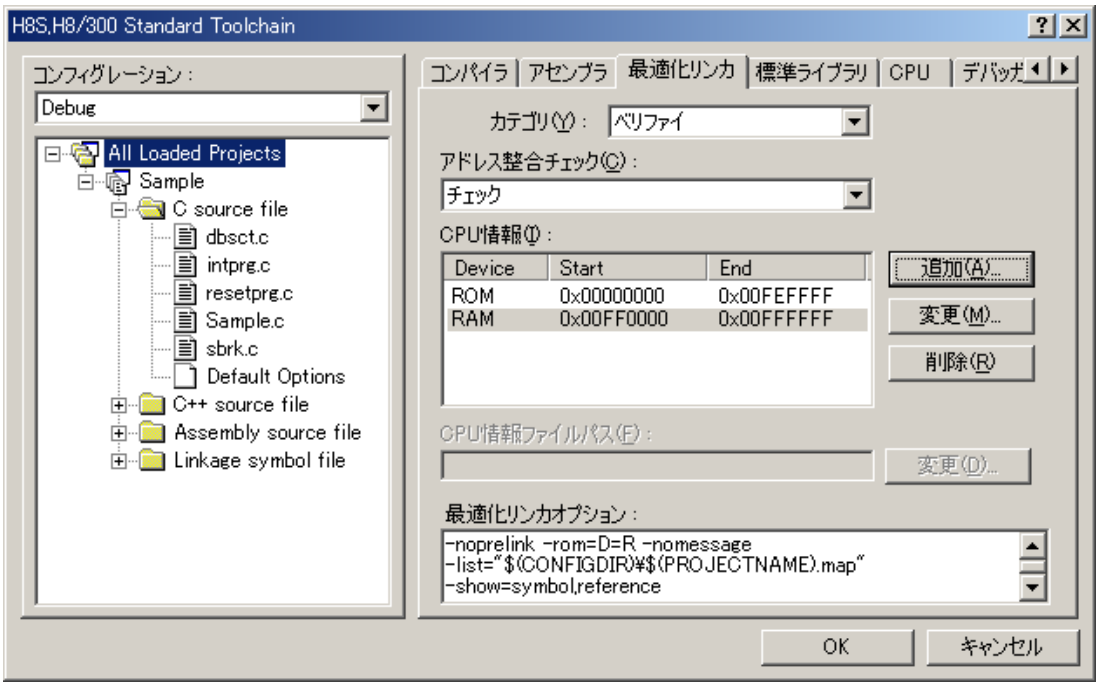
さらに、このリストへのシンボル最適化情報の出力と、シンボル参照回数の出力をここで指定します。

次にリンク時のファイルの割り付け方を[最適化リンカタブ] [カテゴリ/セクション]にて指定します。
ここではBセクションが割り付いているセクションのアドレスをH'00FF0000に変更します。Address欄をクリックし、変更ボタンを押して指定します。

次のようになります。



次に CPU 割り付けのチェックとして、[最適化リンカタブ] [カテゴリ/ベリファイ]にて CPU 情報を作成します。

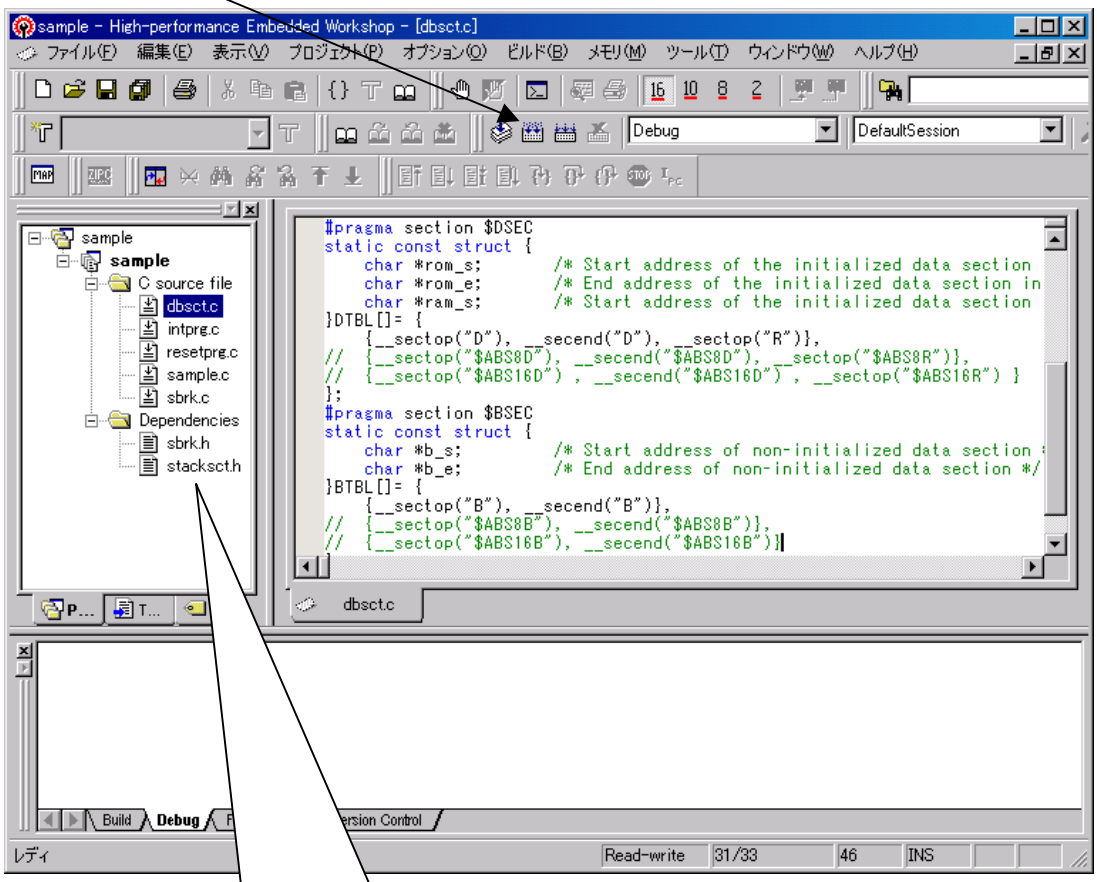


アドレス整合チェック:をチェックにすると、CPU 情報のチェックが可能になります。

[追加...]ボタンを押すと次のダイアログボックスが出てくるので、ROM と RAM の領域をここで次のように指定します。

(14) ビルドの実行

ビルドを実行してロードモジュールを作成します。
コマンドボタンでも[ここ]をクリックするとビルドが可能です。

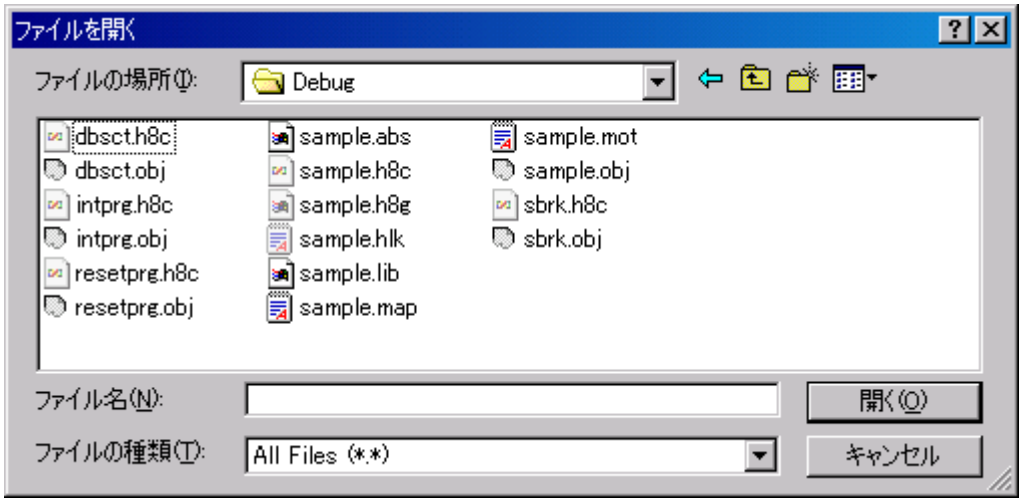


(15) **生成ファイルの確認**

ビルドが終わると次のように各ファイルが作成されます。

アブソリュートロードモジュールはプロジェクトディレクトリの下にプロジェクト名と同じ名前のディレクトリが作成されますが、その下の debug ディレクトリの中に sample.abs という形で作成されます。

また、ビルドの際に生成された、マップファイルも同じディレクトリに作成されるので、こちらを[ファイル->開く]で開いて確認することができます。



マップファイルは sample.map という名称で作成されます。

2.1.3 コマンドラインで起動する

コマンドラインでツールを起動する場合は次の手順で行います。

以下の例は CPU が H8S/2600 アドバンストモードを使用します。
HEW1.2 では、HEW ディレクトリ¥Tools¥HITACHI¥H8¥3_0a_0¥sample にサンプルプログラムが用意されています。

No.	HEW1.2 **ファイル**	**ファイルの内容**
1	init.c	**初期化ルーチン**
2	vectbl.c	**ベクタテーブル設定**
3	scttbl.c	**セクション初期化ルーチン**
4	cmain.c	**メイン関数ファイル**
5	2600a.cpu	CPU **情報ファイル**
6	c2600a.sub	**モジュール間最適化ツール用サブコマンドファイル**

HEW2.0 以降では、サンプルプログラムは用意されていませんので、自作にてご用意していただくか、サンプルプロジェクトを作成してそのとき生成される次のファイルをサンプルプログラムとして使用してください。

サンプルプロジェクトの作成方法は、**「**2.1.2 **ワークスペースの新規作成** 2**（**HEW2.0 **以降）」**に従い Project type の設定を Demonstration に選択してプロジェクトの作成を行ってください。

No.	HEW2.0 **以降のファイル**	**ファイルの内容**
1	resetprg.c	**初期化ルーチン**
2	intprg.c	**ベクタテーブル設定**
3	dbsct.c	**セクション初期化ルーチン**
4	main.c	**メイン関数ファイル**
5	2600a.sub(**自作**)	**サブコマンドファイル**

(1) 環境設定を行う

- PC 版
  set path=<HEWインストールディレクトリ>¥tools¥hitachi¥h8¥v3_0a_0¥bin;%path%
  set CH38=<HEWインストールディレクトリ>¥tools¥hitachi¥h8¥v3_0a_0¥include
  set hlnk_linrary1=<HEWインストールディレクトリ>¥tools¥hitachi¥h8¥v3_0a_0¥lib¥c8s26a.lib

- unix 版
  「1.3 **インストール方法」**を参照してください。

(2) コンパイルする

C プログラムファイルをコンパイルします。

ch38△–cpu=2600a△–debug△init.c△vectbl.c△scttbl.c (RET)
ch38△–cpu=2600a△–debug△-show=allocation,object△-goptimize△cmain.c (RET)

(3) CPU 情報ファイルを作成する (HEW1.2 の場合のみ指定可、HEW2.0 以降では指定できません)

unix 版では cia38 を起動し、使用する ROM,RAM のアドレス範囲を指定します。
cia38 の使用方法は H8S,H8/300 シリーズ C/C++コンパイラユーザーズマニュアル「付録 J. CPU 情報ファイルの作成」を参照してください。
また、PC 版では HEW を使用します。**「**2.1.1(10) **オプションの設定」**を参照してください。
ここでは sample ディレクトリ内の CPU 情報ファイル 2600a.cpu を使用します。

モジュール間最適化用サブコマンドファイルを作成する
モジュール間最適化で指定するサブコマンドファイルを作成します。
ここでは sample ディレクトリ内の c2600a.sub を使用します。（HEW1.2 の場合のみ使用できます。
HEW2.0 では、次の subcommand ファイルを参考に自作願います。）

<c2600a.sub> （subcommand ファイルを変更）

```
input init.obj
input vectbl.obj
input scttbl.obj
input cmain.obj

output test.abs

format absolute

debug

sysrof

rom(D,R)
rom($ABS8D,$ABS8R)
rom($ABS16D,$ABS16R)

start Cvect1,Cvect2(0)
start P,C,D,C$BSEC,C$DSEC,$ABS8D,$ABS816(0200)
start R,B(0ED00),S(0FE00)

exit
```

←入力ファイル
←出力ファイル
←デバッグ情報出力指定
←出力フォーマット指定
←ROM化支援機能
←セクション割り付け指定

このサブコマンドファイルを使用してモジュール間最適化を実行します。

optlnk38△–sub=test.sub (RET) (HEW1.2 コマンドライン)
optlnk△–sub=test.sub (RET) (HEW2.0 コマンドライン)

ロードモジュールファイル「sample.abs」を出力します。また、HEW1.2 では、リンケージリスト「sample.map」にメモリ割り付け情報を、最適化情報リスト「sample.lop」にシンボル最適化情報を出力し、HEW2.0 では、リンケージリスト「sample.map」にメモリ割り付け情報とシンボル最適化情報を出力します。

(4) オブジェクトファイルを変換する

ROM 化するためにオブジェクトロードモジュール（この場合は、SYSROF 形式です）を S タイプ形式に変換します。

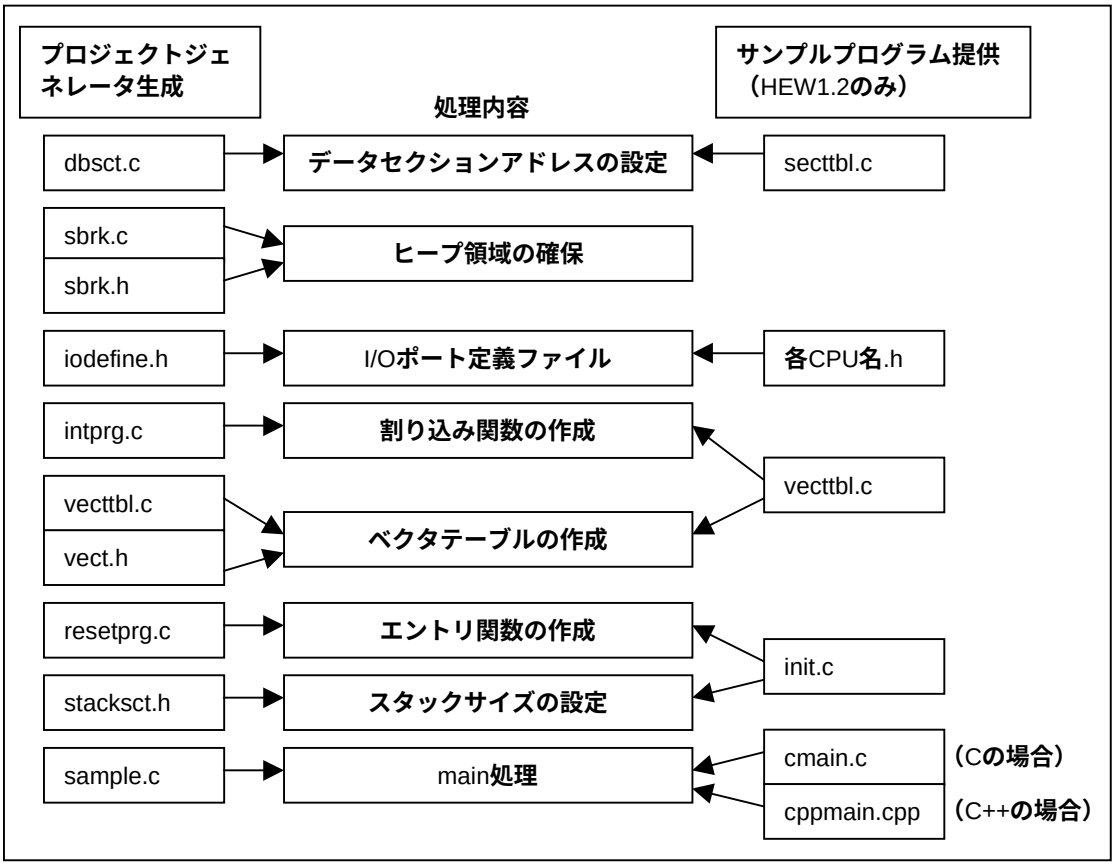
cnvs△test.abs (RET) (HEW1.2 コマンドライン)

HEW2.0 以降の場合、最適化リンカにコンバータ機能が統合されているため、S タイプ形式はコンバータを使用しなくても出力できます。
サブコマンドファイルに form=stype を記述して S タイプ形式を出力してください。

2.2 サンプルプログラムの作成

2.2.1 ROM 化プログラムとして必要な初期設定処理について

プロジェクトジェネレータにより生成されたプログラムを用いて説明します。
まず、プロジェクトジェネレータにより生成されるプログラムと製品に添付されているサンプルプログラムのファイル構成を示します。
（サンプルプログラムは、HEW1.2 では添付していますが、HEW2.0 以降では添付しておりません。
また HEW2.0 以降では vecttbl.c と vect.h は intprg.c の拡張機能の記述により生成されなくなりました。）



(1) データセクションアドレスの設定

（HEW **プロジェクトファイル名：**dbsct.c、**サンプルプログラム名：**scttbl.c）

```
/*********************************************************************/
/*                                                                   */
/*  FILE       :dbsct.c                                              */
/*  DATE       :Thu, Nov 04, 1999                                    */
/*  DESCRIPTION :Setting of B,R Section                              */
/*  CPU TYPE   :H8S/2621                                             */
/*                                                                   */
/*  This file is generated by Renesas Project Generator (Ver.3.0).   */
/*                                                                   */
/*********************************************************************/


#pragma section $DSEC
static const struct {
    char *rom_s;      /* Start address of the initialized data section in ROM */
    char *rom_e;      /* End address of the initialized data section in ROM   */
    char *ram_s;      /* Start address of the initialized data section in RAM */
}DTBL[]= {
    {__sectop("D"), __secend("D"), __sectop("R")},
    {__sectop("$ABS8D"), __secend("$ABS8D"), __sectop("$ABS8R")},
    {__sectop("$ABS16D") , __secend("$ABS16D") , __sectop("$ABS16R") }
    
};

#pragma section $BSEC
static const struct {
    char *b_s;        /* Start address of non-initialized data section */
    char *b_e;        /* End address of non-initialized data section */
}BTBL[]= {
    {__sectop("B"), __secend("B")},
    {__sectop("$ABS8B"), __secend("$ABS8B")},
    {__sectop("$ABS16B"), __secend("$ABS16B")}
    
};
```

初期化データ領域のセクションアドレスの設定

未初期化データ領域のセクションアドレスの設定

初期化データ領域セクションと、未初期化データ領域セクションの初期設定を行うルーチンで使用するために、それぞれのセクションのアドレスを設定します。

初期化データ領域セクション名を追加する場合、ここに前の行と同じようにセクション名を追加します。

未初期化データ領域セクション名を追加する場合、ここに前の行と同じようにセクション名を追加します。

ここで使用している_ _sectop、_ _secend はセクションアドレスを求める拡張機能です。

この機能については**「**3.3 **セクションアドレス演算子」**で説明いたします。

(2) ヒープ領域の確保

（HEW **プロジェクトファイル名：**sbrk.c、sbrk.h、**サンプルプログラム名：**sbrk.c、lowsrc.c、otherlb.c）

メモリ管理ライブラリで使用するヒープ領域を割り付ける関数を作成します。

```
/************************************************************************/
/*                                                                      */
/*  FILE       :sbrk.c                                                  */
/*  DATE       :Thu, Nov 04, 1999                                       */
/*  DESCRIPTION :Program of sbrk                                        */
/*  CPU TYPE   :H8S/2621                                                */
/*                                                                      */
/*  This file is generated by Renesas Project Generator (Ver.3.0).      */
/*                                                                      */
/************************************************************************/
#include <stdio.h>
#include "sbrk.h"

//const size_t _sbrk_size=    /* Specifies the minimum unit of  */
                              /* the defined heap area          */
extern char *_s1ptr;
extern void srand(unsigned int);

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(unsigned long 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 ;
}

/************************************************************************/
/* _INIT_OTHERLIB                                                       */
/*  Initialize C library Functions, if necessary.                       */
/*  Define OTHERLIB on Assembler Option.                                */
/************************************************************************/

void _INIT_OTHERLIB(void)
{
    srand(1);
    _s1ptr=NULL;
}
```

低水準インタフェースルーチンの作成方法として HEW1.2 の場合 H8S,H8/300 **シリーズ** C/C++**コンパイラユーザーズマニュアル「付録 E. 低水準インタフェースルーチンの作成例」**として、HEW2.0 以降では H8S、H8/300 **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「**9.2.2(7) **低水準インタフェースルーチン」**に記載しています。

（HEW プロジェクトファイル名；sbrk.h）

```
/************************************************************************/
/*                                                                      */
/*  FILE       :sbrk.h                                                  */
/*  DATE       :Thu, Nov 04, 1999                                       */
/*  DESCRIPTION :Header file of sbrk file                               */
/*  CPU TYPE   :H8S/2621                                                */
/*                                                                      */
/*  This file is generated by Renesas Project Generator (Ver.3.0).      */
/*                                                                      */
/************************************************************************/
/* size of area managed by sbrk */
#define HEAPSIZE 0x400
```

低水準ルーチン sbrk の定義時に使用するインクルードファイルです。ヒープ領域のサイズを示しています。プロジェクト指定後にヒープ領域のサイズを変更したい場合は、この値を変更します。

例）ヒープ領域のサイズを 514(0x202)バイトとしたい場合
#define HEAPSIZE 0x202

(3) I/O ポートファイル定義

（HEW プロジェクトファイル名：iodefine.h**、サンプルプログラム名：<各** CPU **名**>.h**）**

I/O ポートを変数名でアクセスできるようにするために、設定しています。

```
/************************************************************************/
/*                                                                      */
/*  FILE       :iodefine.h                                              */
/*  DATE       :Thu, Nov 04, 1999                                       */
/*  DESCRIPTION :Definition of I/O Register                             */
/*  CPU TYPE   :H8S/2621                                                */
/*                                                                      */
/*  This file is generated by Renesas Project Generator (Ver.3.0).      */
/*                                                                      */
/************************************************************************/


/************************************************************************/
/*     H8S/2623 Group Include File                    Ver 1.1     */
/************************************************************************/
struct st_hcan {                                        /* struct HACN  */
               union {                                  /* MCR          */
                     unsigned char BYTE;                /*  Byte Access */
                     struct {                           /*  Bit  Access */
                            unsigned char SLPME :1;     /*    SLPME     */
                            unsigned char       :1;     /*              */
                            unsigned char SLPM  :1;     /*    SLPM      */
                            unsigned char       :2;     /*              */
                            unsigned char MSM   :1;     /*    MSM       */
                            unsigned char HALT  :1;     /*    HALT      */
                            unsigned char RST   :1;     /*    RST       */
                      }             BIT;                /*              */
               }                MCR;                    /*              */
               union {                                  /* GSR          */
                     unsigned char BYTE;                /*  Byte Access */
                     struct                             /*  Bit  Access */
                            unsigned char wk  :4;       /*              */
                            unsigned char RSF :1;       /*    RSF       */
                            unsigned char MSEF:1;       /*    MSEF      */
                            unsigned char SRWF:1;       /*    SRWF      */
                            unsigned char BOF :1;       /*    BOF       */
                      }             BIT;                /*              */
               }                GSR;                    /*              */
                                    （ 中略 ）

#define HCAN   (*(volatile struct st_hcan  *)0xFFF800) /* HCAN  Address*/
#define SCRX   (*(volatile union  un_scrx  *)0xFFFDB4) /* SCRX  Address*/
#define SBYCR  (*(volatile union  un_sbycr *)0xFFFDE4) /* SBYCR Address*/
#define SYSCR  (*(volatile union  un_syscr *)0xFFFDE5) /* SYSCR Address*/
#define SCKCR  (*(volatile union  un_sckcr *)0xFFFDE6) /* SCKCR Address*/

                                    (以下続く)
```

プロジェクトジェネレータを使用しない場合は、サンプルとして製品にも添付されています。
CPU と同名のインクルードファイル、C ソースファイルを探し、確認してから使用してください。

(4) 割り込み関数の作成

（HEW プロジェクトファイル名：intprg.c**、サンプルプログラム名：**vecttbl.c**）**

割り込み呼び出しを行う関数を定義します。

＜HEW1.2＞

```
/************************************************************************/
/*                                                                      */
/*  FILE        :intprg.c                                               */
/*  DATE        :Thu, Nov 04, 1999                                      */
/*  DESCRIPTION :Interrupt Program                                      */
/*  CPU TYPE    :H8S/2621                                               */
/*                                                                      */
/*  This file is generated by Renesas Project Generator (Ver.3.0).      */
/*                                                                      */
/************************************************************************/


#include   <machine.h>
#include   "vect.h"
#pragma section IntPRG
// vector 2 Reserved

// vector 3 Reserved

// vector 4 Reserved

// vector 5 Treace
void   INT_Treace(void) {/* sleep(); */}
// vector 6 Reserved

// vector 7 NMI
void   INT_NMI(void) {/* sleep(); */}
// vector 8 User breakpoint trap
void   INT_TRAP1(void) {/* sleep(); */}
// vector 9 User breakpoint trap
void   INT_TRAP2(void) {/* sleep(); */}
// vector 10 User breakpoint trap
void   INT_TRAP3(void) {/* sleep(); */}
// vector 11 User breakpoint trap
void   INT_TRAP4(void) {/* sleep(); */}
// vector 12 Reserved
```

←セクション名指定

↓割り込み関数本体の定義

【注】 #pragma section IntPRG **と指定すると、**PIntPRG **というセクションに割り付けられます。モジュール間最適化ツールで、セクション名を変更する必要がある場合はご注意ください。**

＜HEW2.0 以降＞

```
/************************************************************************/
/*                                                                      */
/*  FILE       :intprg.c                                                */
/*  DATE       :Tue, Aug 20, 2002                                       */
/*  DESCRIPTION :Interrupt Program                                      */
/*  CPU TYPE   :H8S/2612                                                */
/*                                                                      */
/*  This file is generated by Renesas Project Generator (Ver.3.0).      */
/*                                                                      */
/************************************************************************/


#include   <machine.h>
#pragma section IntPRG
//  vector 2 Reserved

//  vector 3 Reserved

//  vector 4 Reserved

//  vector 5 Trace
__interrupt(vect=5) void INT_Trace(void) {/* sleep(); */}
//  vector 6 Reserved

//  vector 7 NMI
__interrupt(vect=7) void INT_NMI(void) {/* sleep(); */}
//  vector 8 User breakpoint trap
__interrupt(vect=8) void INT_TRAP0(void) {/* sleep(); */}
//  vector 9 User breakpoint trap
__interrupt(vect=9) void INT_TRAP1(void) {/* sleep(); */}
//  vector 10 User breakpoint trap
__interrupt(vect=10) void INT_TRAP2(void) {/* sleep(); */}
//  vector 11 User breakpoint trap
__interrupt(vect=11) void INT_TRAP3(void) {/* sleep(); */}
//  vector 12 Reserved

//  vector 13 Reserved
```

__interrupt(vect=5)**記述によりベクタテーブルを自動作成**

割り込み関数の詳細な指定方法に関しては**「**3.1　**割り込み関数の指定方法」**を参照してください。

(5) ベクタテーブルの作成

（HEW プロジェクトファイル名：vecttbl.c**、サンプルプログラム名：**vecttbl.c**）**

ベクタテーブルに関数のアドレスの設定を行います。（HEW1.2 のみ生成）

```
/************************************************************************/
/*                                                                      */
/*  FILE        :vecttbl.c                                              */
/*  DATE        :Thu, Nov 04, 1999                                      */
/*  DESCRIPTION :Initialize of Vector Table                             */
/*  CPU TYPE    :H8S/2621                                               */
/*                                                                      */
/*  This file is generated by Renesas Project Generator (Ver.3.0).      */
/*                                                                      */
/************************************************************************/



#include   "vect.h"

#pragma section VECTTBL
void *RESET_Vectors[] = {
//;<<VECTOR DATA START (POWER ON RESET)>>
//;0 Power On Reset
    PowerON_Reset,
//;<<VECTOR DATA END (POWER ON RESET)>>
//;<<VECTOR DATA START (MANUAL RESET)>>
//;1 Manual Reset
    Manual_Reset
//;<<VECTOR DATA END (MANUAL RESET)>>
};
#pragma section INTTBL
void *INT_Vectors[] = {
    // 2 Reserved
    (void *) Dummy,
    // 3 Reserved
    (void *) Dummy,
    // 4 Reserved
    (void *) Dummy,
    // 5 Treace
    (void *) INT_Treace,
    // 6 Reserved
    (void *) Dummy,
    // 7 NMI
    (void *) INT_NMI,
    // 8 User breakpoint trap
    (void *) INT_TRAP1,
    // 9 User breakpoint trap
    (void *) INT_TRAP2,
    // 10 User breakpoint trap
    (void *) INT_TRAP3,
    // 11 User breakpoint trap
    (void *) INT_TRAP4,
    // 12 Reserved
    (void *) Dummy,
    // 13 Reserved
    (void *) Dummy,
    // 14 Reserved
    (void *) Dummy,

                          （以降省略）
```

VECTTBL**セクションに**RESET_Vectors**という名称のベクタテーブルを作成**

INTTBL**セクションに**INT_Vectors**という名称のベクタテーブルを作成**

【注】 #pragma section **でセクション名を指定した場合、デフォルトのセクション名に付加されてしまうため、モジュール間最適化ツールでアドレスを割り付けるときに、セクション名を変更する必要があります。**

(6) vect.h

ベクタテーブル設定時に参照される割り込み関数のプロトタイプ宣言をしています。
（HEW1.2 のみ生成）

```
; /************************************************************************/
/*                                                                      */
/*  FILE        :vect.h                                                 */
/*  DATE        :Thu, Nov 04, 1999                                      */
/*  DESCRIPTION :Definition of Vector                                   */
/*  CPU TYPE    :H8S/2621                                               */
/*                                                                      */
/*  This file is generated by Renesas Project Generator (Ver.3.0).      */
/*                                                                      */
/************************************************************************/




//;<<VECTOR DATA START (POWER ON RESET)>>
//;0 Power On Reset
extern void PowerON_Reset(void);
//;<<VECTOR DATA END (POWER ON RESET)>>
//;<<VECTOR DATA START (MANUAL RESET)>>
//;1 Manual Reset
extern void Manual_Reset(void);
//;<<VECTOR DATA END (MANUAL RESET)>>
// 2 Reserved

// 3 Reserved

// 4 Reserved

// 5 Treace
#pragma interrupt INT_Treace
extern void INT_Treace(void);
// 6 Reserved

// 7 NMI
#pragma interrupt INT_NMI
extern void INT_NMI(void);
// 8 User breakpoint trap
#pragma interrupt INT_TRAP1
extern void INT_TRAP1(void);
// 9 User breakpoint trap
#pragma interrupt INT_TRAP2
extern void INT_TRAP2(void);
// 10 User breakpoint trap
#pragma interrupt INT_TRAP3
extern void INT_TRAP3(void);
// 11 User breakpoint trap
#pragma interrupt INT_TRAP4
extern void INT_TRAP4(void);
// 12 Reserved

// 13 Reserved

// 14 Reserved

                                （以降省略）
```

←この#pragma interrupt指定することにより割り込み関数とみなしリターン時にRTE命令を生成します。
割り込み関数の詳細については「3.1 割り込み関数の指定方法」を参照してください。

(7) **エントリ関数の作成**

（HEW **プロジェクトファイル名：**resetprg.c、**サンプルプログラム名：**init.c）

```
/*********************************************************************/
/*                                                                   */
/*  FILE        :resetprg.c                                          */
/*  DATE        :Thu, Nov 04, 1999                                   */
/*  DESCRIPTION :Reset Program                                       */
/*  CPU TYPE    :H8S/2621                                            */
/*                                                                   */
/*  This file is generated by Renesas Project Generator (Ver.3.0).   */
/*                                                                   */
/*********************************************************************/


#include   <machine.h>
#include   "stacksct.h"

#pragma entry PowerON_Reset

extern void main(void);

#ifdef __cplusplus
extern "C" {
#endif
extern void _INITSCT(void);
#ifdef __cplusplus
}
#endif

//#ifdef __cplusplus            // Remove the comment when you use SIM I/O
//extern "C" {
//#endif
//extern void _INIT_IOLIB(void);
//extern void _CLOSEALL(void);
//#ifdef __cplusplus
//}
//#endif

//extern void srand(unsigned int);  // Remove the comment when you use rand()
//extern char *_s1ptr;              // Remove the comment when you use strtok()

//#ifdef __cplusplus            // Remove the comment when you use Hardware Setup
//extern "C" {
//#endif
//extern void Hardware Setup(void);
//#ifdef __cplusplus
//}
//#endif
```

組み込み関数用インクルードファイルをインクルードします

PowerON_Reset**をエントリ関数として指定します**
コンパイラはエントリ関数に対してSP**の初期設定を行うコードを出力します**

（以降次ページ）

```
                                （前ページ続き）

#pragma section ResetPRG

void PowerON_Reset(void);
void PowerON_Reset(void)
{
    set_imask_ccr(1);       CCRの割り込みフラグをイネーブルにします

   _INITSCT();              セクションの初期化ルーチンを呼び出します

// _INIT_IOLIB();                  // Remove the comment when you use SIM I/O

// srand(1);                       // Remove the comment when you use rand()
// _s1ptr=NULL;                    // Remove the comment when you use strtok()

// HardwareSetup();                // Remove the comment when you use Hardware Setup

   main();                  main関数を呼び出します

// _CLOSEALL();                    // Remove the comment when you use SIM I/O

   sleep();                 低消費電力モードにします
}

void Manual_Reset(void);
void Manual_Reset(void)
{
}
```

(8) スタックサイズの設定

（HEW プロジェクトファイル名： stacksct.h）

```
/*********************************************************************/
/*                                                                   */
/*  FILE        :stacksct.h                                          */
/*  DATE        :Thu, Nov 04, 1999                                   */
/*  DESCRIPTION :Setting of Stack area                               */
/*  CPU TYPE    :H8S/2621                                            */
/*                                                                   */
/*  This file is generated by Renesas Project Generator (Ver.3.0).   */
/*                                                                   */
/*********************************************************************/
#pragma stacksize 0x200
```

スタックサイズを指定します。この指定により、512 バイトのスタックセクションが作成されます。スタックセクションの名称は S が固定です。

スタックセクションのサイズは関数の呼び出し関係のネストの一番深い位置の値です。

オブジェクトリストの割り付け情報で出力される Total Frame Size を参照し計算することになります。

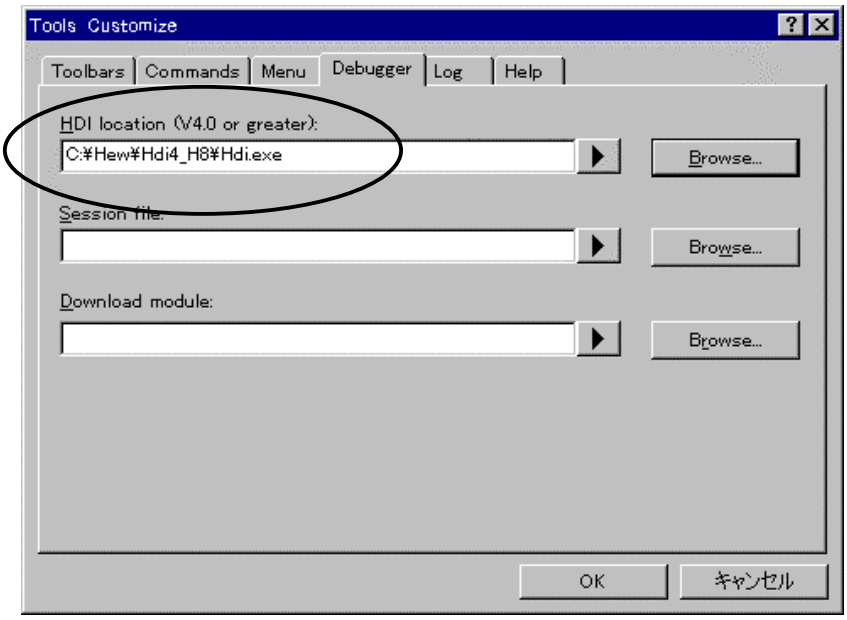
スタックサイズの指定を変更する場合はこの数値を変更します。

2.3 HDI を使用したデバッグ方法

新規に作成した HEW ワークスペースを使用して HDI を使って簡単にデバッグしてみましょう。（HEW1.2、HEW2.0 以降両方から HDI を操作できます）

2.3.1 HEW との連動 1

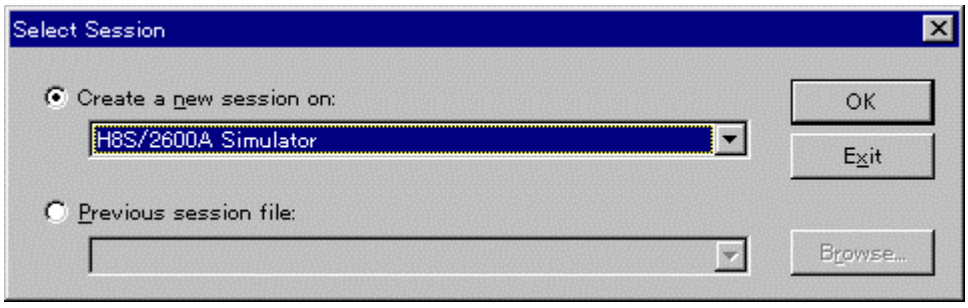
HEW メニューの Tools から Customize...を選択すると Tools Customize ダイアログの Debugger シートの HDI location 欄に HDI.exe を指定すると HEW のメニューの Launch Debugger ボタンから HDI を起動することができます。



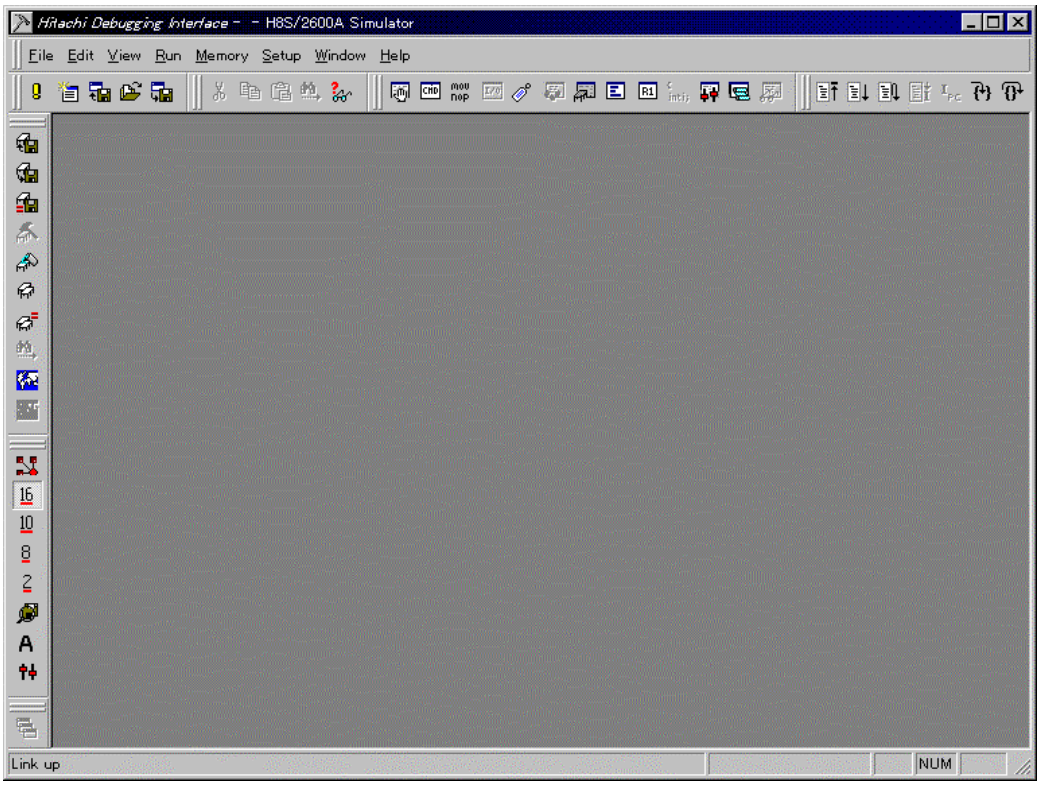

Launch Debugger

2.3.2 ターゲットの選択

次の画面が現れるので CPU 種別とデバッガの種類を選択します。
ここでは、H8S/2600 アドバンストモードを指定したので、H8S/2600A Simulator を選択します。
その後[OK]ボタンを押します。



するとスプラッシュウィンドウ表示後、HDI のウィンドウが開きます。



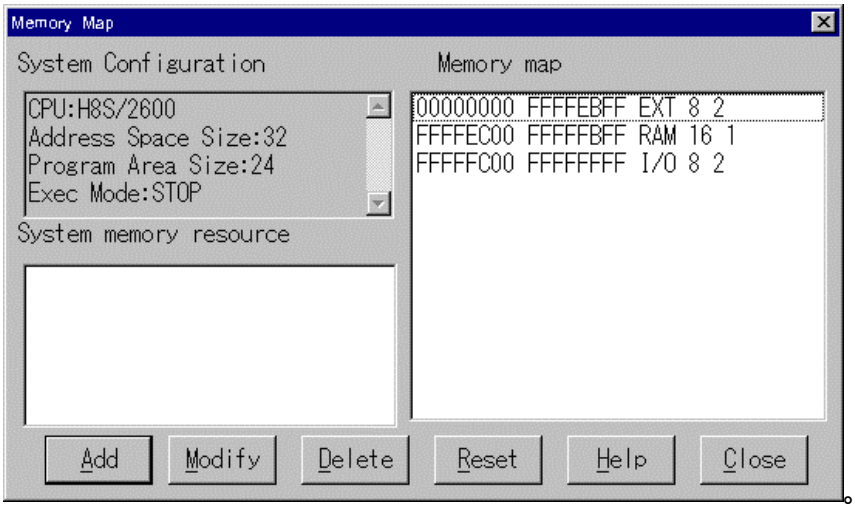
2.3.3 メモリリソースの確保

次に、このロードモジュールを動作させるためのメモリリソースの確保を行います。

[View]メニューから[Memory Mapping Window]を選択、または、ツールバーの Memory Mapping ボタンをクリックします。

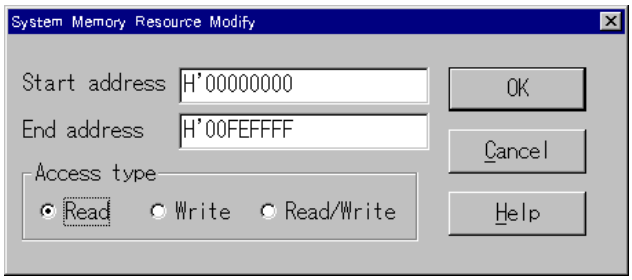

Memory Mapping

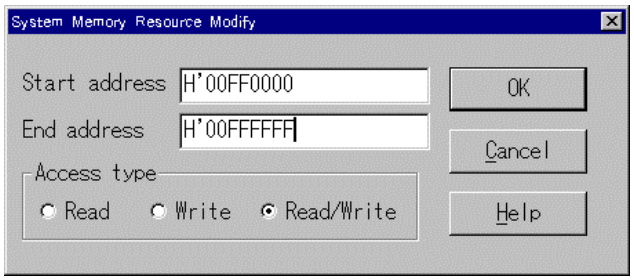
すると、Memory Map ダイアログボックスが表示されます。

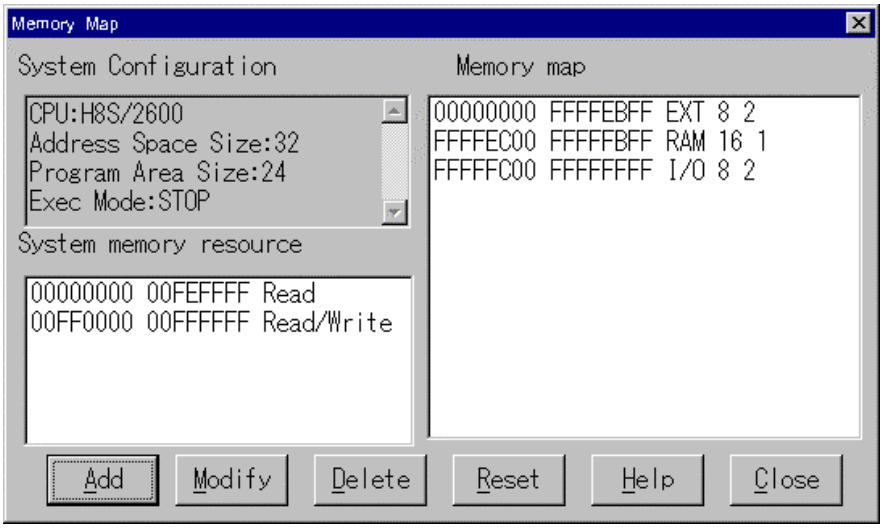


[Add]ボタンを押して System Memory Resource Modify 画面で、メモリリソースを確保します。

この場合、領域のすべてを指定してみます。ROM 領域として H'0 から H'00FEFFFF、RAM 領域として H'00FF0000 から H'00FFFFFF までを指定します。Access type は、ROM 領域に Read、RAM 領域に Read/Write を指定しておきます。
その後、[OK]ボタンを押します。







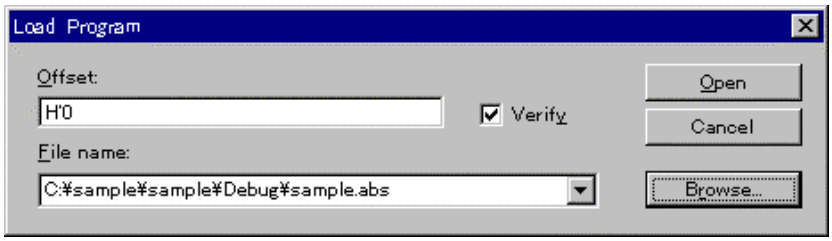
すると、このように指定されたこととなります。
[Close]ボタンを押して、このウィンドウを閉じます。

2.3.4 ロードモジュールのダウンロード

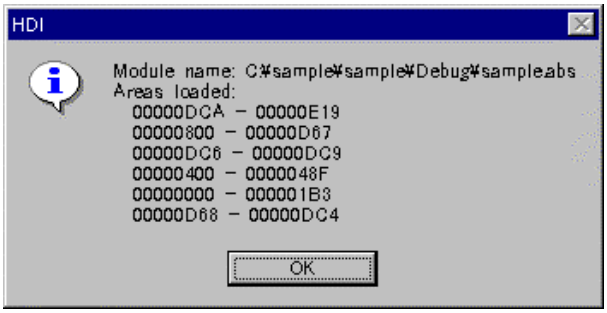
[File]メニューから[Load Program]を選び、デバッグ対象のアブソリュートロードモジュールを選択します。ボタンで指定する場合は、ツールバーの Load Program ボタンをクリックします。

Load Program

sample.abs ファイルを選択します。Open をクリックします。

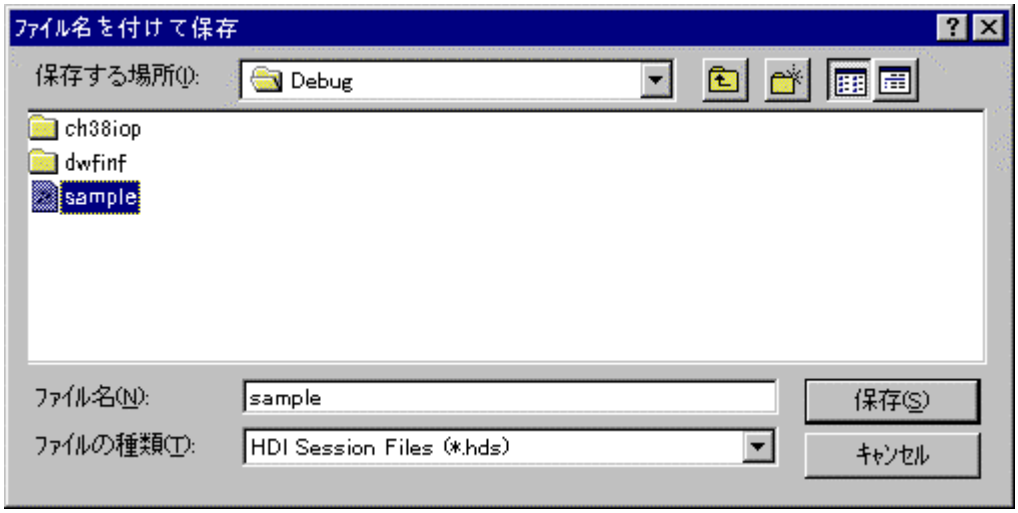


すると、次の画面が表示されます。
ファイルがロードされ、プログラムコードが書き込まれたメモリエリアに関する情報を表示します。

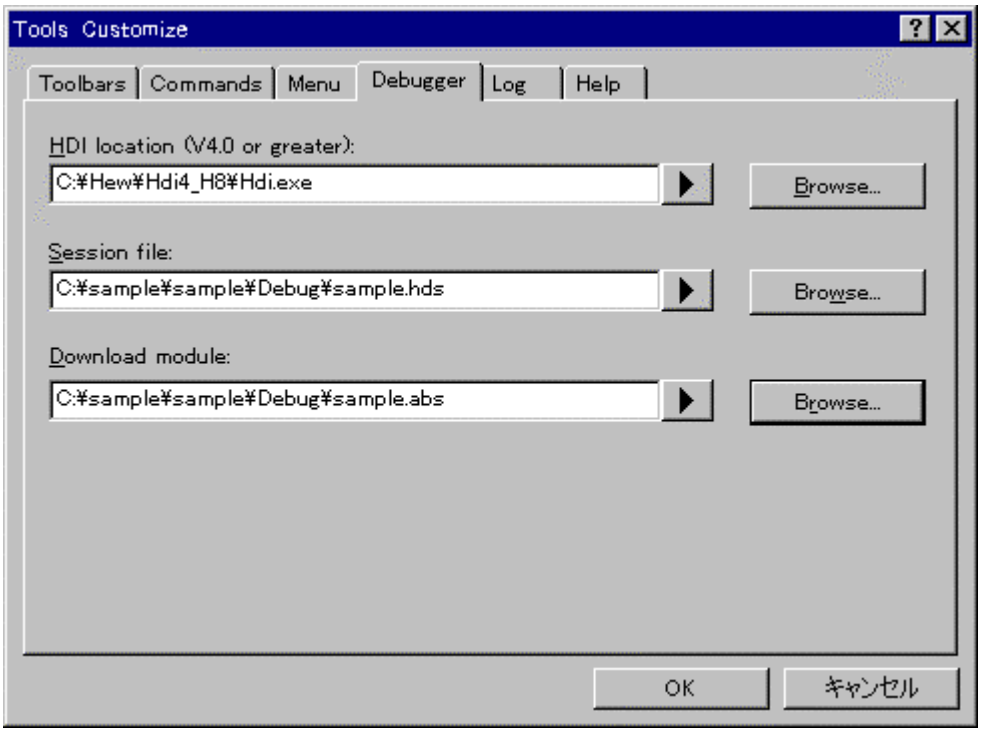


2.3.5 HEW との連動 2

File メニューから Save Session As…を選択します。



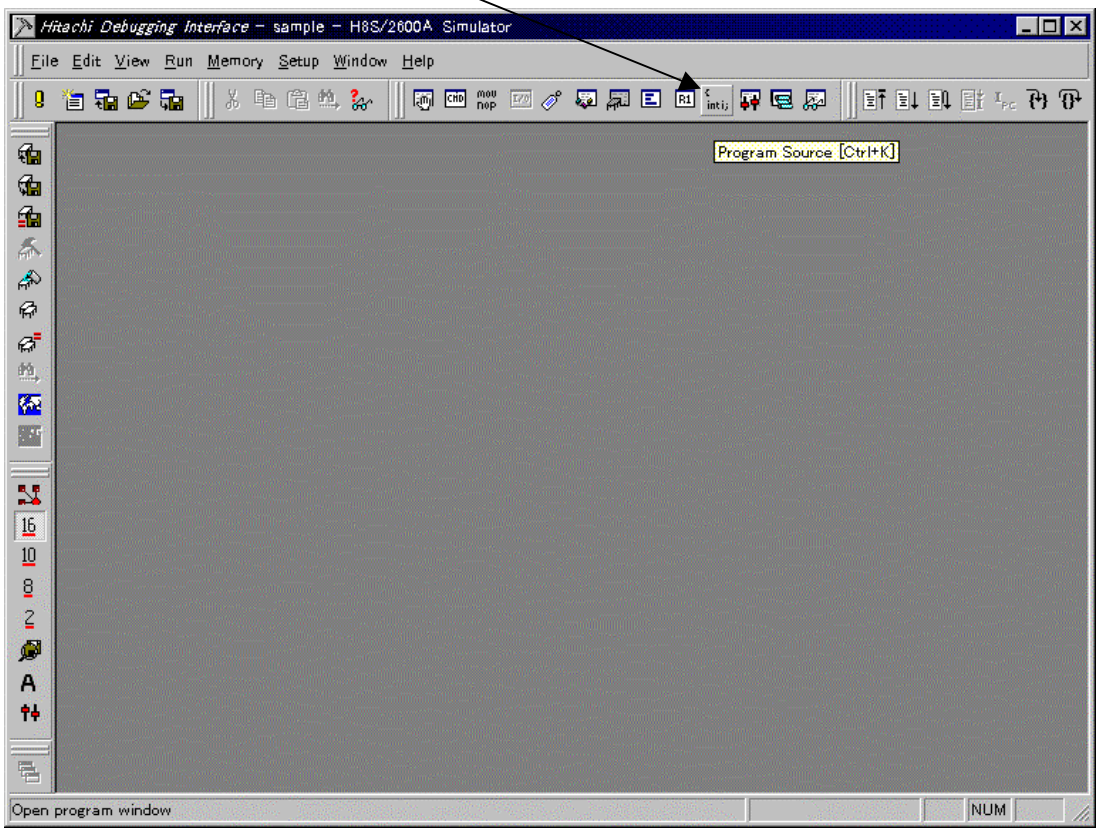
HEW メニューの Tools から Customize...を選択すると Tools Customize ダイアログの Debugger シートの Session file 欄にセッションファイル名、Download module 欄にロードモジュールを指定すると HEW のメニューの Launch Debugger ボタンから HDI を起動したときに、セッションがロードされます。

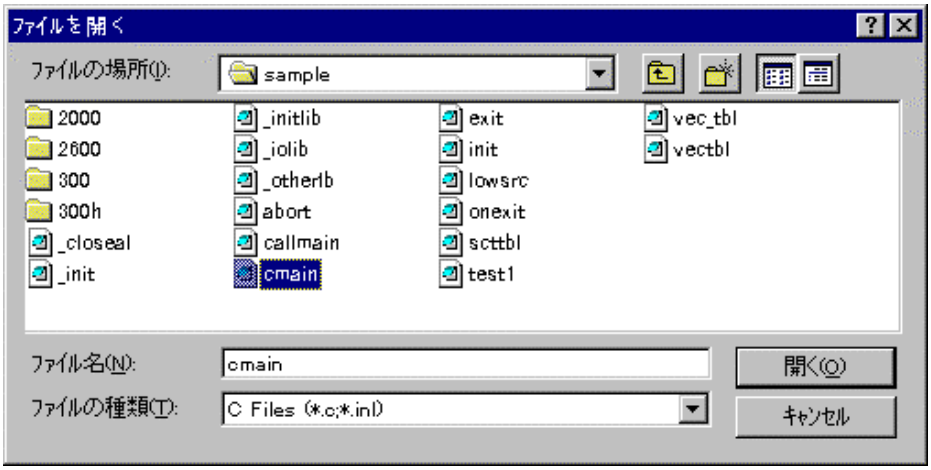


2.3.6 ソースプログラムの表示

Program Source ボタンをクリックします。

Program file

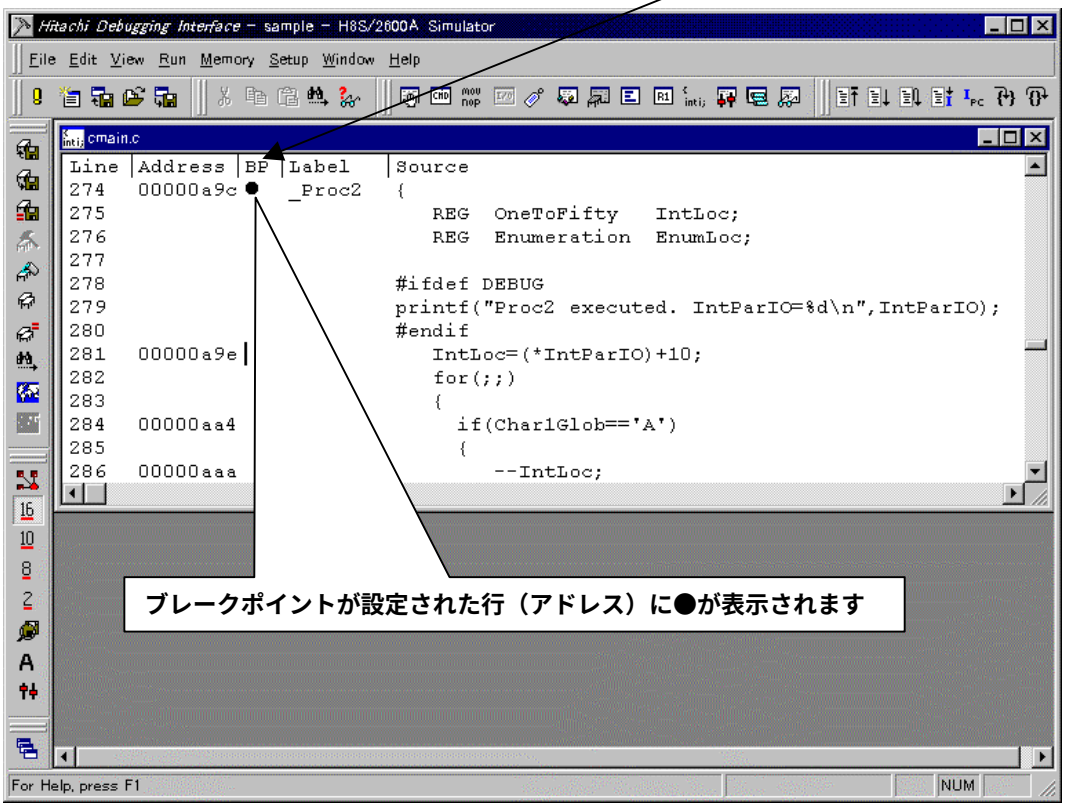




cmain.c を選択します。

2.3.7 ブレイクポイントの設定

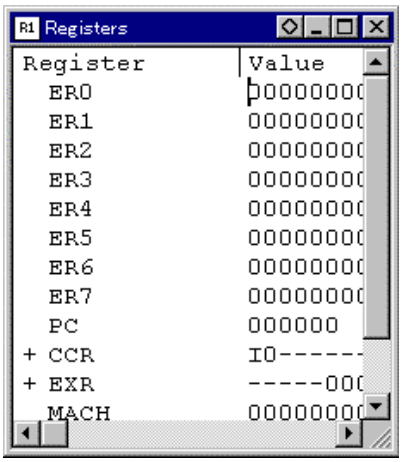
プログラムの BP カラムでブレークポイントを設定するソース行の位置をダブルクリックします。
たとえば、main 関数の開始時点にブレイクポイントを設定する場合は、ここをダブルクリックします。



2.3.8 レジスタ状況の表示

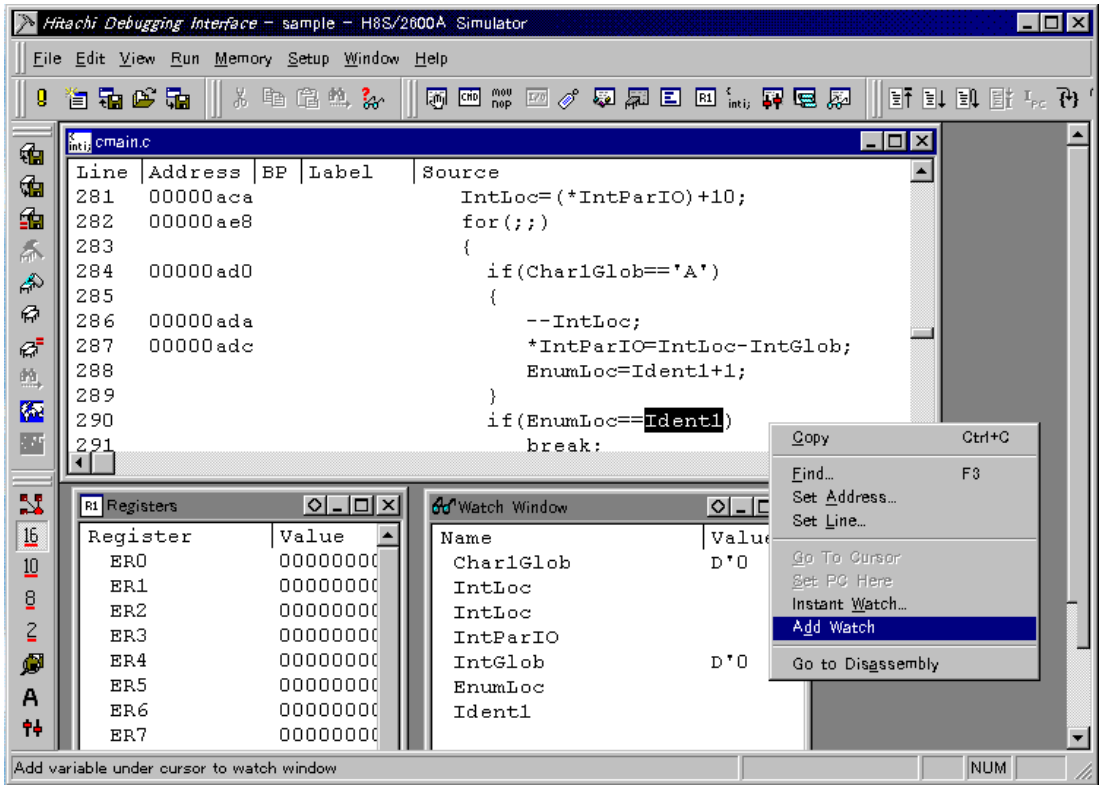
[View]メニューから[Registers]を選択または、ツールバーの CPU Registers ボタンをクリックします。
[View]メニューから Register Window を開くと、レジスタの状況を見ることができます。

CPU Register



2.3.9 外部変数の参照

注目する変数名を選択して、右ボタンクリックし、ポップアップメニューから Add Watch を選択します。Watch Window で変数の値を参照することができます。また、マウスカーソル変数上に置くと変数値を表示します。



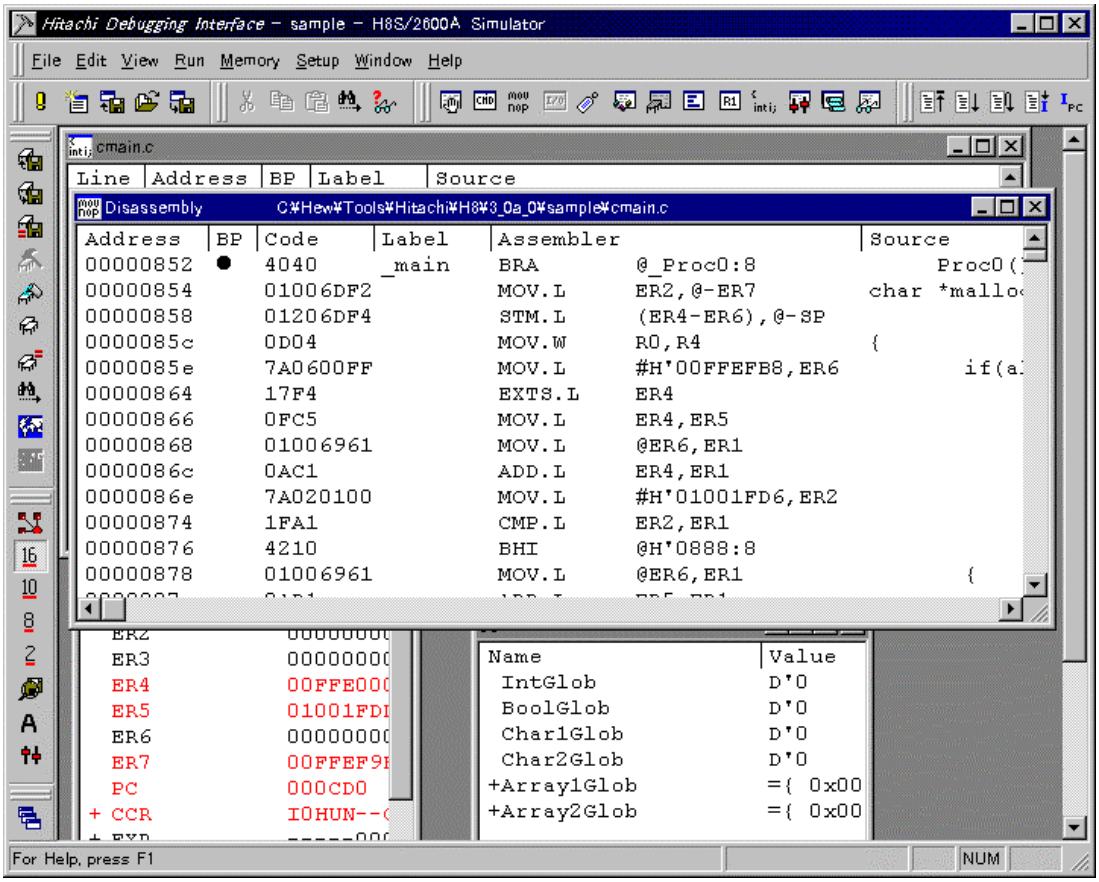
この準備ができたら、プログラムを実行させてみます。

2.3.10 ResetGo コマンド

[Run]メニューから RestGo を選択すると、PC がブレイクポイントに到達するまでプログラムを実行します。

ResetGo

C ソースプログラム上で、右クリックでポップアップメニューから Go to Disassembly を選択すると Disassembly ウィンドウが表示されます。Disassembly ウィンドウの最右カラムには Source カラムがあり、Disassemble に対応する C ソースプログラムが表示されます。



2.3.11 局所変数の参照

[View]メニューから[Locals]を選択すると、Locals ウィンドウが表示されます。本ウィンドウには、現在の PC のある位置から参照可能な局所変数とその値を表示します。

関数内部へは Step ボタンで入ります。次にステップ実行の方法を示します。

2.3.12 プログラムのステップ実行

[Run]メニューの Step In、Step Over、Step Out を使って、ステップ実行してみます。
Step In はサブルーチン呼び出しがある場合には、そのサブルーチン内に PC を移動します。
Step Over はサブルーチン呼び出し行を 1 行として PC を移動します。
Step Out は現在 PC のあるサブルーチンから呼び出されたサブルーチンに PC を移動します。


Step


Step Over

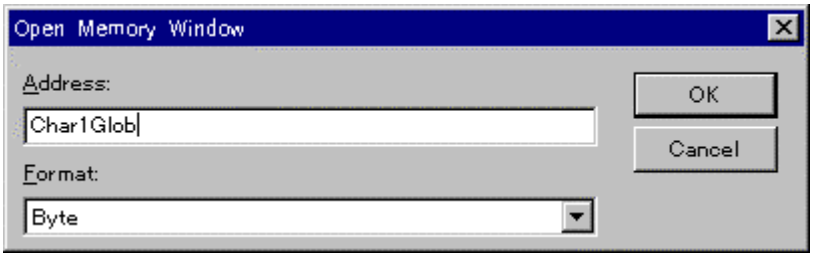

Step Out

[Run]メニューから[Run…]を選択すると、Run ダイアログボックスを開きます。本ダイアログでは、1 ステップの基準値を変更することができます。

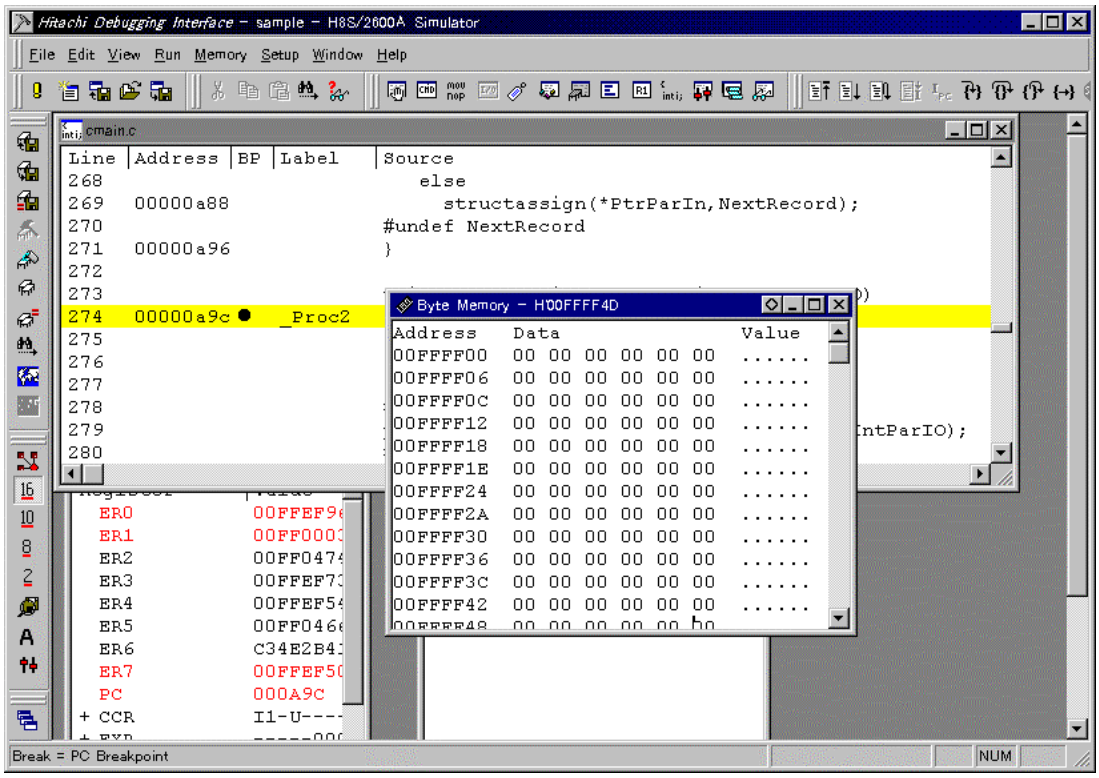
本設定では、1 ステップの基準を C ソースプログラムの 1 行としています。

2.3.13 メモリ内容の確認

[View->Memory…]で指定します。
Open Memory Window ダイアログボックスが表示されます。Address フィールドにシンボル名を入力します。

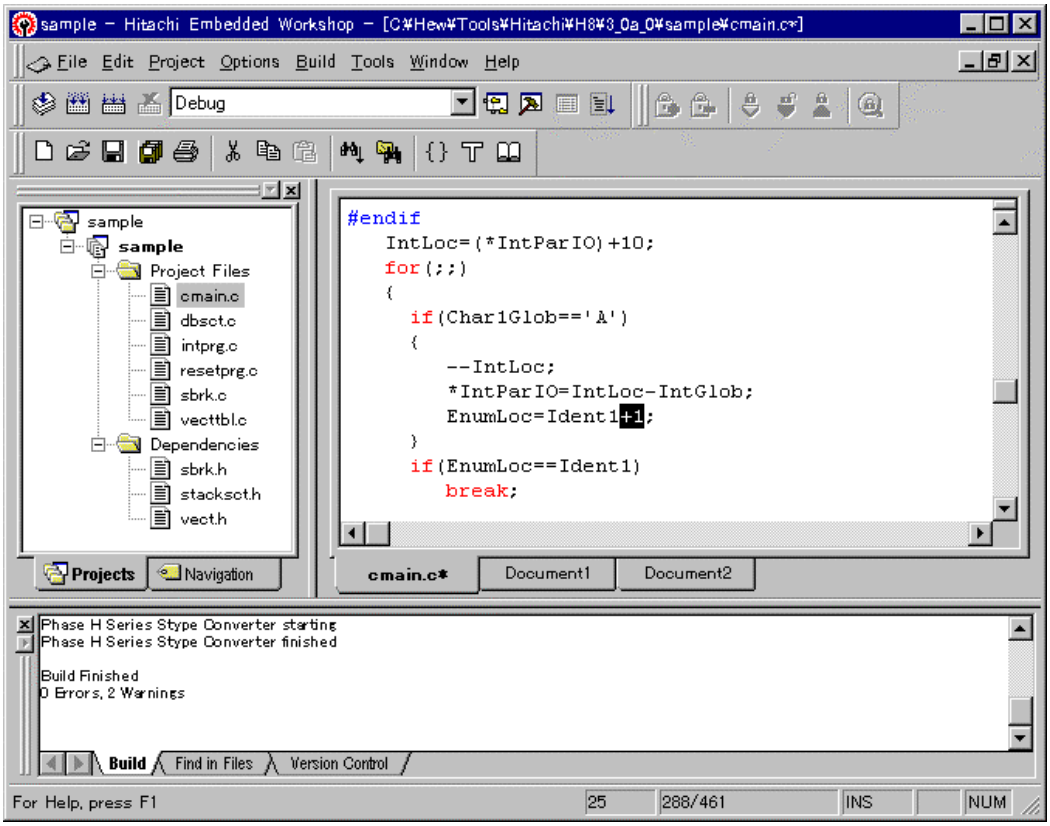


次のように Byte Memory 画面が表示されるので、内容を確認することができます。

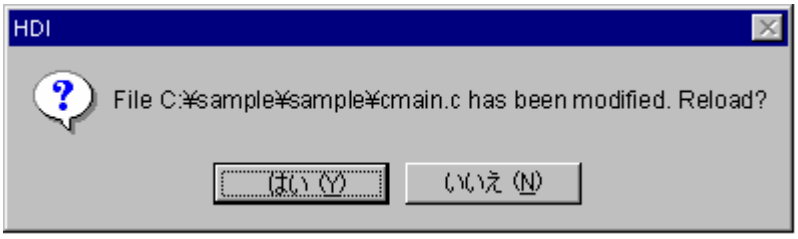


2.3.14 HEW との連動 3

HEW から HDI を起動した場合に、HDI のソースウィンドウ上でダブルクリックすると HEW の Editor に該当ファイルがオープンされます。

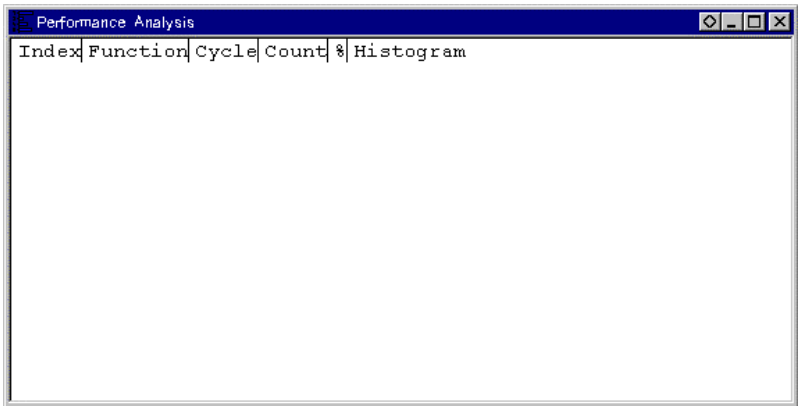


これを編集して save し、リコンパイルします。（サンプルプログラムは現在 Read Only のため書き換えはできません。）
HDI をアクティブにすると、メッセージダイアログを表示し再ロードするかどうかを通知します。

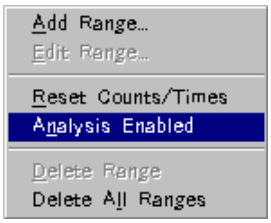


Yes を選択すると HDI は再ロードを行います。

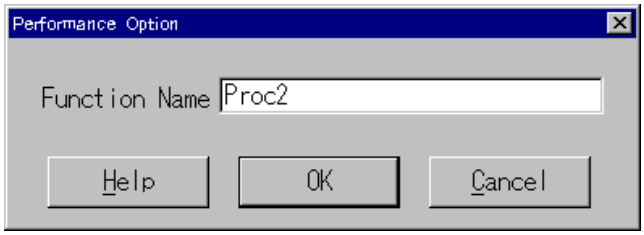
このようにしてデバッグをしていきます。
HDI は、性能測定機能があります。
プログラムの性能を測定する場合には、[View]メニューから Performance Analysis を選択すると、Performance Analysis ウィンドウが開きます。

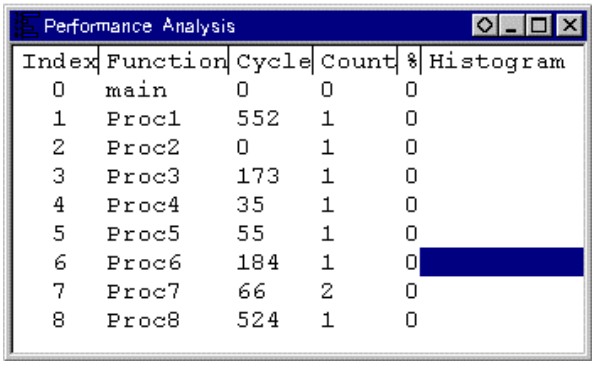


性能を測定する場合にはポップアップメニューで Enable をチェックします。



性能を測定するラベルをポップアップメニューの Add Range でラベルを指定します。





そしてプログラムを実行すると、各ラベルの性能測定結果を表示します。
HDI の機能の詳細については**「HDI ユーザーズマニュアル」**を参照してください。

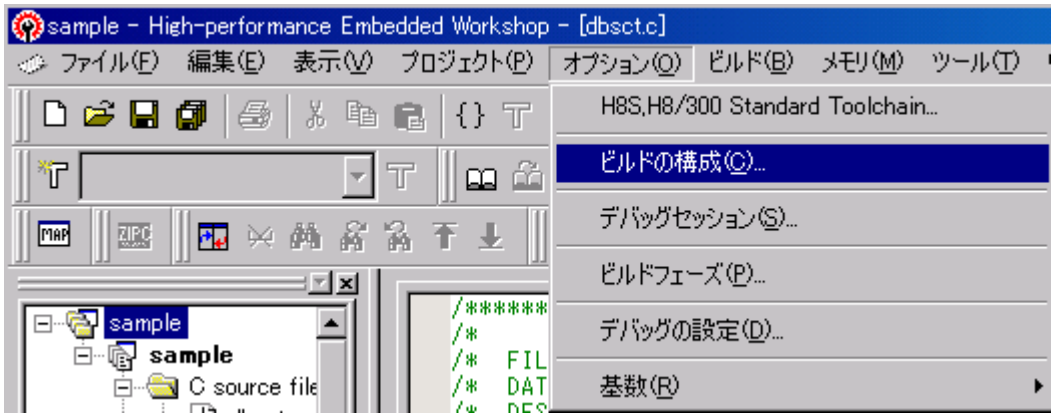
2.4 シミュレータデバッガを使用したデバッグ方法

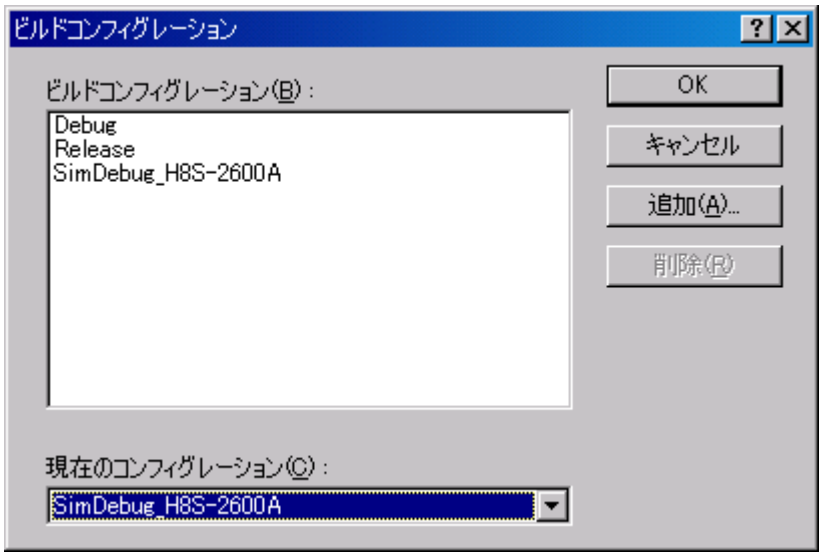
HEW2.0 より HEW 上でデバックすることができるようになりました。（HEW1.2 では使用できません）

ワークスペースの新規作成にて Project type の設定を Demonstration に選択して作成したサンプル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、Build configurations の画面で、使用する環境を選択してください。この場合は、[SimDebug_H8S-2600A]を選択します。
  コンフィグレーションを変更した場合、ビルドを実行してください。

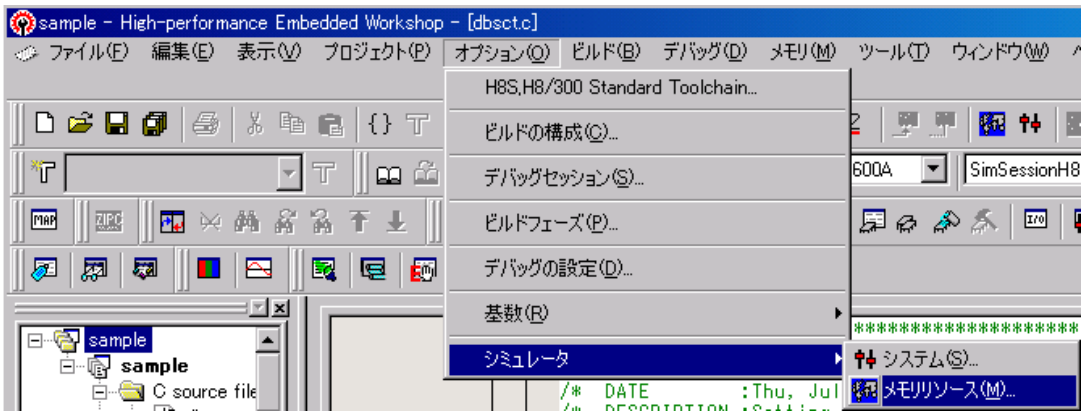


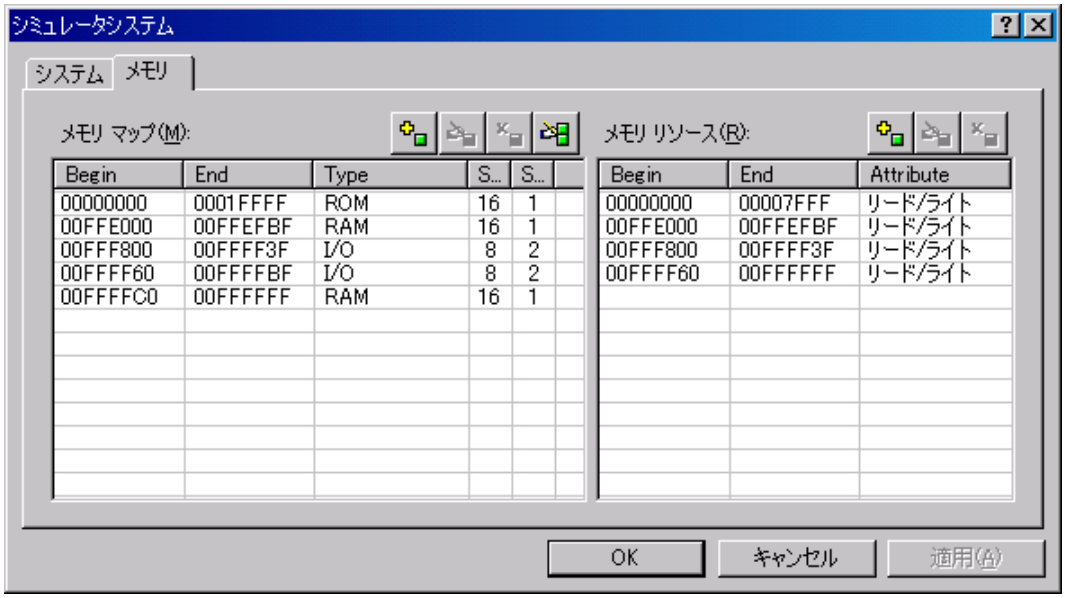


2.4.2 メモリリソースの確保

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。





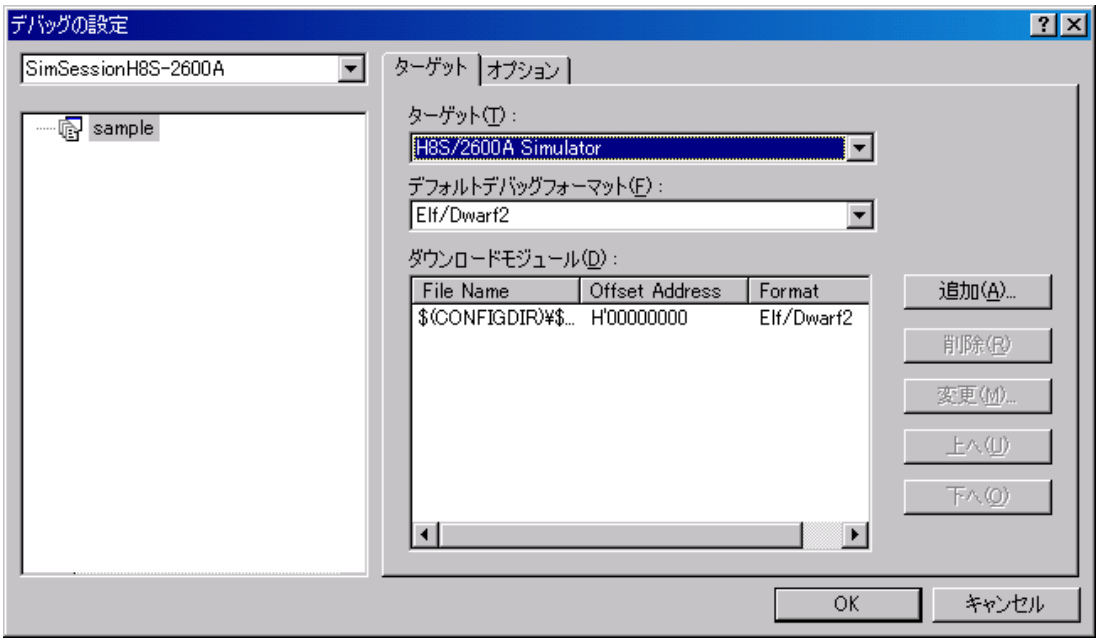
プログラム領域として H'00000000 から H'00007FFF を、スタック領域として H'00FFEC00 から H'00FFFFFF を読み出し／書き込み可能領域として確保しています。

- [Close]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、H8S,H8/300 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。相互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。



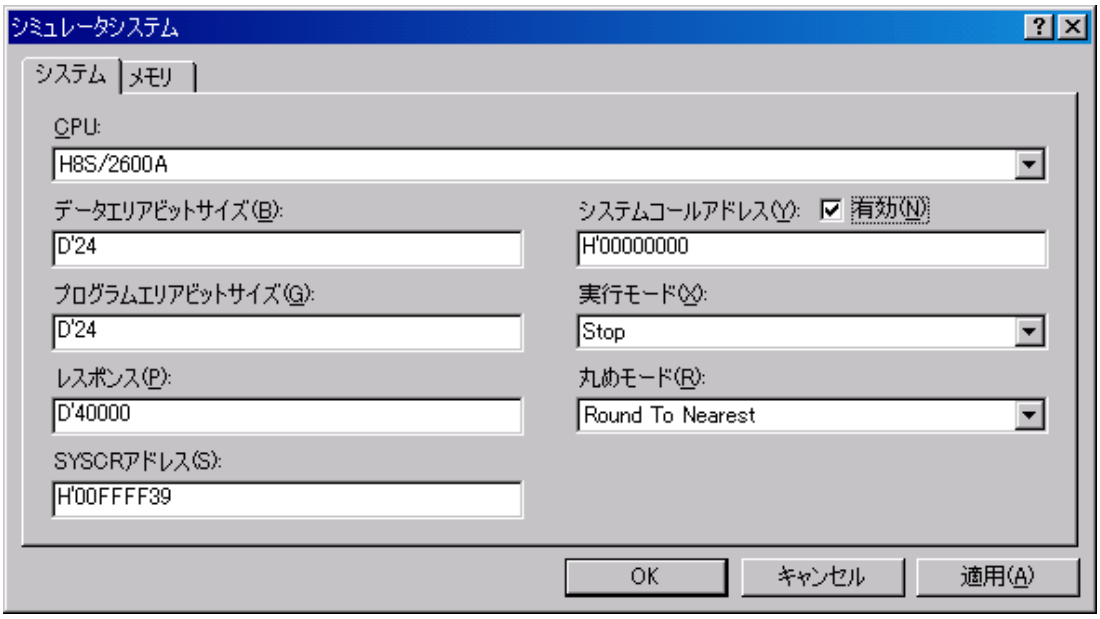
[ダウンロードモジュール]に設定しているファイルがダウンロードするファイルです。

- [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[ダウンロードモジュール -> 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。

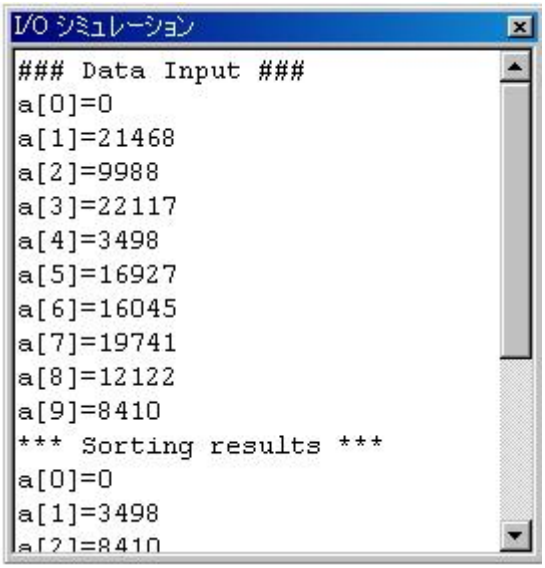
2.4.4 I/O シミュレーションの設定

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。

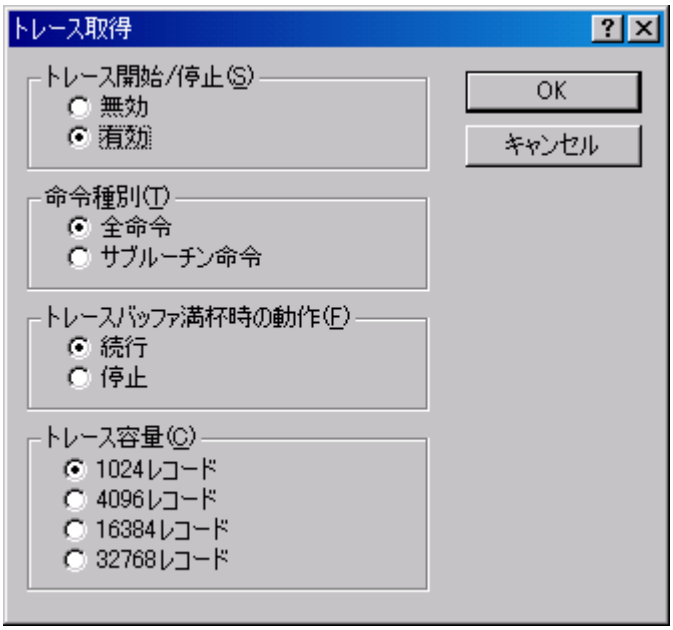


- [システムコールアドレス]の[有効]にチェックがあることを確認してください。
- [OK]ボタンをクリックして I/O シミュレーションを有効にしてください。
- [表示]メニューから[CPU->I/O シミュレーション]を選択して、I/O シミュレーション ウィンドウを開いてください。
  I/Oシミュレーションウィンドウを開かなければ、I/Oシミュレーションが有効になりません。



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。
  以下にトレース取得ダイアログボックスを表示します。

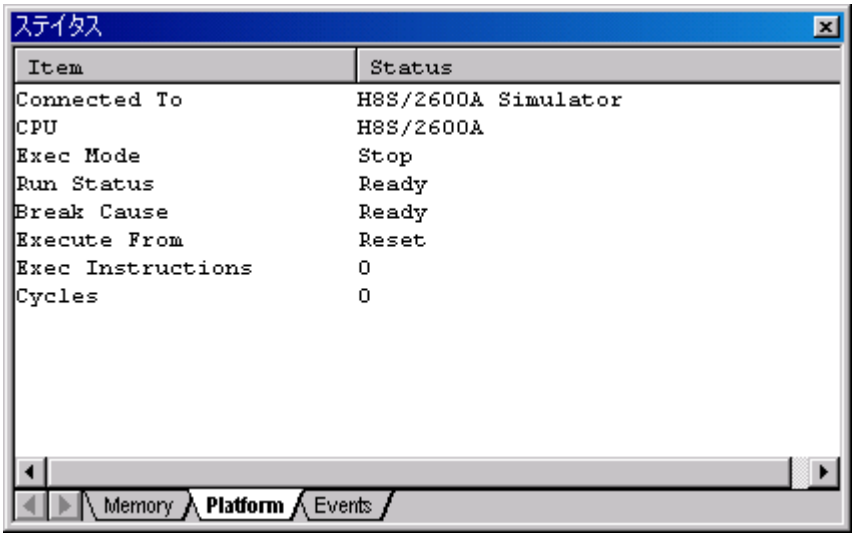


- トレース取得ダイアログボックスの[トレース開始/停止]を[有効]に設定し、[OK]ボタンをクリックしてトレース情報取得を有効にしてください。

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。

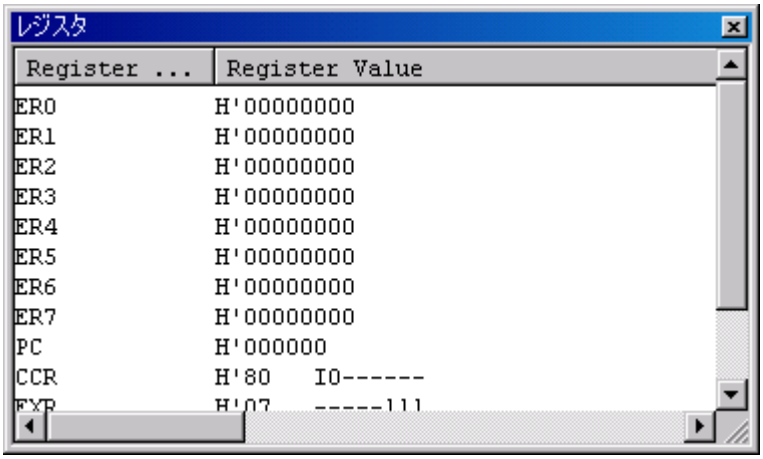
- [表示]メニューから[CPU->ステイタス]を選択して、ステイタスウィンドウを開いてください。さらに、ステイタスウィンドウのうち[Platform]シートを表示してください。



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。

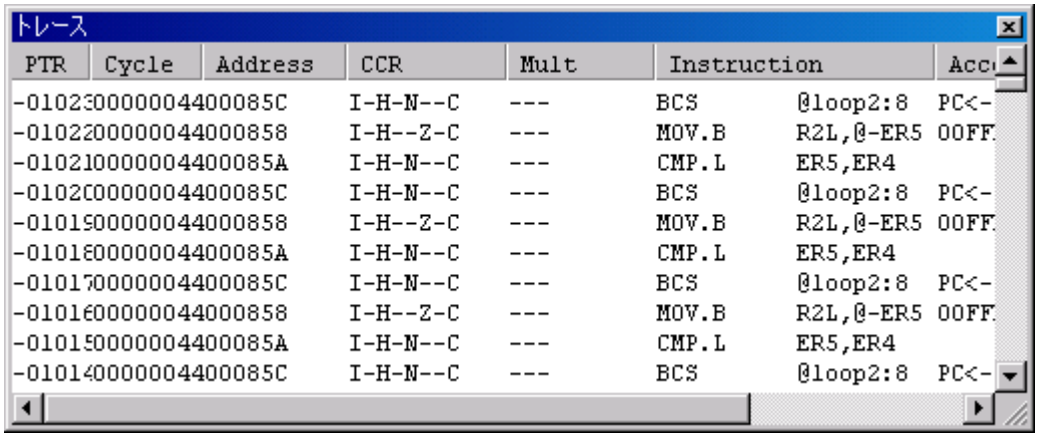


2.4.8 トレース

(1) トレースバッファ

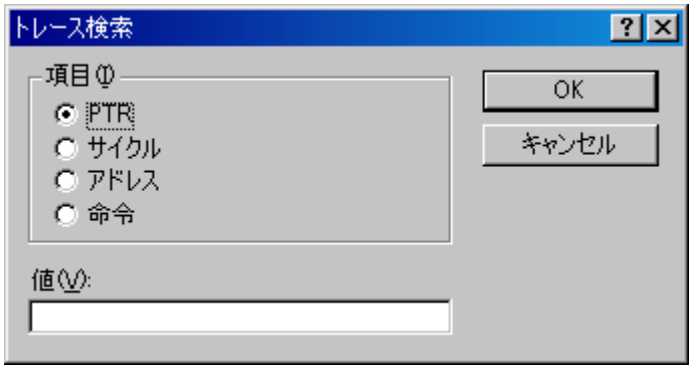
トレースバッファを使って、命令実行の履歴を知ることができます。

[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。

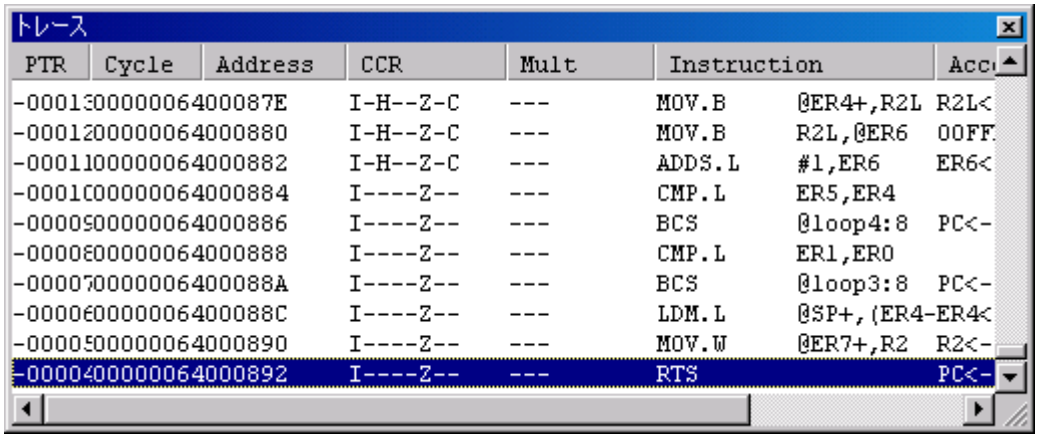


(2) トレースサーチの実行

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。

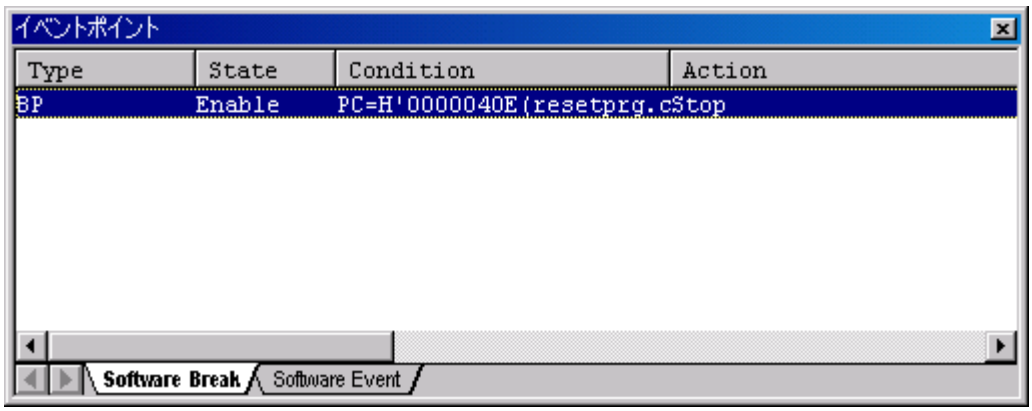


サーチ項目[項目]とサーチ内容[値]を設定して[OK]ボタンをクリックすると、トレースサーチを実行します。該当するトレース情報があった場合、該当する最初の行を強調表示します。同じサーチ内容[値]でトレースサーチを続ける場合は、トレースウィンドウ上で右クリックしてポップアップメニューを表示し、[次を検索]を選択してください。次に該当する行を強調表示します。



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。



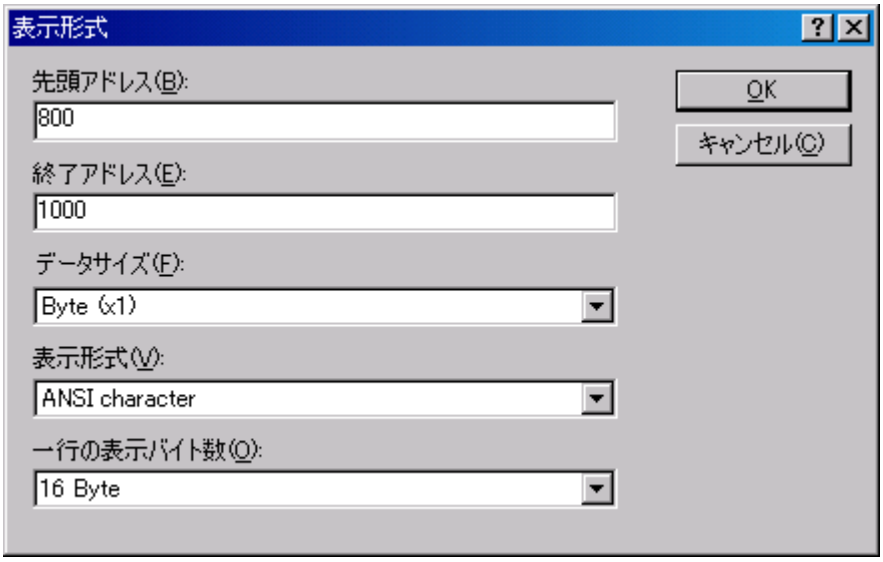
イベントポイントウィンドウによって、ブレークポイントの設定、新しいブレークポイントの定義、およびブレイクポイントの表示ができます。

イベントポイントウィンドウを閉じてください。

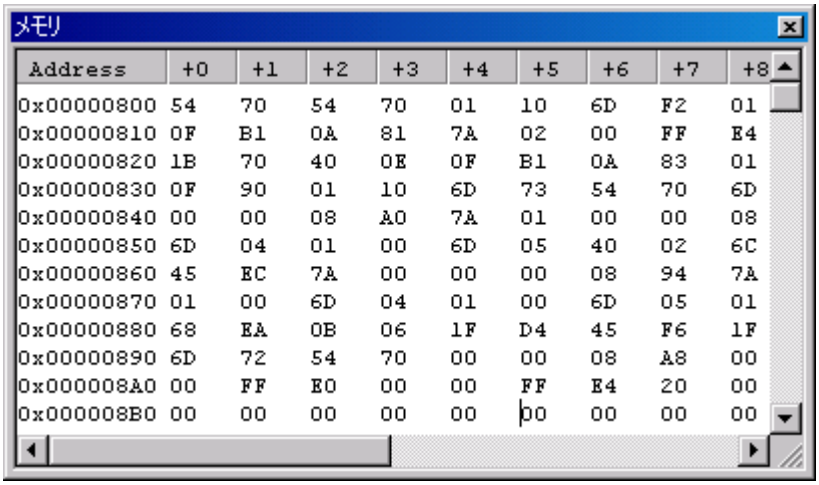
2.4.10 メモリ内容の確認

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。



- [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。



3. コンパイラ

本章では、C/C++プログラムにおけるプログラム開発時に使用すると便利な拡張機能について説明いたします。
以下の機能を使用すると、割り込み処理や、通常 C/C++プログラムでは実現できない処理を行うことができます。

3.1 割り込み関数の指定方法

■説明

#pragma interrupt <関数名> で割り込み関数を宣言します。宣言された割り込み関数は関数内部で使用するレジスタがすべて保証（退避／回復）され、RTE 命令で復帰します。これにより例外処理からの復帰が可能です。

【書式】

#pragma interrupt（<関数名>[(<割り込み仕様>)][,<関数名>[(<割り込み仕様>)]…]）

■使用例

割り込み関数 f1 を宣言します。処理終了後、RTE 命令で復帰します。

（C/C++プログラム）

```
extern unsigned char a;
#pragma interrupt (f1)
void f1(void)
{
   a=0;
}
```

←関数f1を割り込み関数とする

（コンパイル結果アセンブリ展開コード）

```
_f1:
        PUSH.W      R0
        SUB.B       R0L,R0L
        MOV.B       R0L,@_a:32
        POP.W       R0
        RTE
        .END
```

←割り込み関数はRTE命令でリターンする

■備考および注意事項

割り込み関数宣言には「スタック切り替え指定」、「トラップ命令リターン指定」、「割り込み終了関数指定」、「ベクタテーブル指定」の機能があります。

No.	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） &<変数>:変数（オブジェクトの型）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	割り込み終了関数指定	sy=	<関数名>| <定数>| $<関数名>	終了を割り込み関数へのジャンプ命令で指定 <関数名>：割り込み関数名 <定数>：絶対アドレス $<関数名>：下線を付加しない割り込み関数名

3.1.1 スタック切り替え指定

■説明

割り込み関数用スタック領域を別に指定することができます。

スタック切り替え指定（sp=）は、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。

■使用例

新しいスタックのアドレスを変数または定数で指定します。以下の例では、配列 STK[100]を割り込み関数 f で使用するスタックとして設定します。

（C/C++プログラム）

```
extern int STK[100];
extern unsigned char a;
#pragma interrut (f(sp=STK+100))

void f(void)
{
    a=0;
}
```

←割り込み関数指定し、スタックポインタを切り替える

（コンパイル結果アセンブリ展開コード）

```
_f:
        MOV.L       SP,@_STK+96:32
        MOV.L       #_STK+96:32,SP
        PUSH.W      R0
        SUB.B       R0L,R0L
        MOV.B       R0L,@_a:32
        POP.W       R0
        MOV.L       @SP,SP
        RTE
        .END
```

←スタックポインタを変更する

←RTE命令でリターンする

■備考および注意事項

(i) 本指定は、トラップ命令リターン指定または割り込み関数終了指定と同時に指定することができます。

(ii) スタック切り替え指定 “ sp= ” は必ず小文字で指定してください。

3.1.2 トラップ命令リターン指定

■説明

#pragma interrupt で宣言された関数は、通常 RTE 命令でリターンしますが、トラップ命令リターン指定（tn=）をした場合、TRAPA 命令で復帰します。

■使用例

割り込み関数終了時に"TRAPA #2"命令でトラップ例外処理を開始することを指定します。

（C/C++プログラム）

```
extern unsigned char a;
#pragma interrupt (f(tn=2))

void f(void)
{
    a=0;
}
```

←割り込み関数fのリターンをTRAPA命令で行う

（コンパイル結果アセンブリ展開コード）

```
_f:
        PUSH.W      R0
        SUB.B       R0L,R0L
        MOV.B       R0L,@_a:32
        POP.W       R0
        TRAPA       #2          ←TRAPA命令でリターンする
        .END
```

■備考および注意事項

(i) 本指定はスタック切り替え指定と同時に指定することはできますが、割り込み関数終了指定とは同時に指定することはできません。
(ii) トラップ命令リターン指定“ tn= ”は必ず小文字で記述してください。
(iii) CPU/動作モードが300のとき、本指定は使えません。

3.1.3 割り込み関数終了指定

■説明

#pragma interrupt で宣言した関数は、通常 RTE 命令で復帰しますが、割り込み関数終了指定（sy=）をした場合は JMP 命令で指定されたアドレスへジャンプします。

■使用例

関数 f2 のアドレスへ JMP 命令でジャンプします。
（C/C++プログラム）

```
extern int f2();
extern unsigned char a;
#pragma interrupt (f1(sy=$ f2))     ←割り込み関数f1のリターンを関数f2のアドレスへJMP命令でとぶ

void f1(void)
{
   a=0;
}
```

（コンパイル結果アセンブリ展開コード）

```
_f1:
        PUSH.W      R0
        SUB.B       R0L,R0L
        MOV.B       R0L,@_a:32
        POP.W       R0
        JMP         @f2:24      ←JMP命令でリターンする
        .END
```

■備考および注意事項

(i) 本指定は、スタック切り替え指定と同時に指定することはできますが、トラップ命令リターン指定とは同時に指定することはできません。
(ii) “ $<関数名> ”を指定した場合、アセンブリプログラムで参照するときに下線を付加していない名前（関数名）になります。
(iii) 割り込み関数終了指定“ sp= ”は必ず小文字で指定してください。

3.1.4 ベクタテーブル自動生成機能

■説明

#pragma interrupt のベクタ番号指定を用いて自動的に関数のベクタテーブルを生成できます。
【書式】
#pragma interrupt（<関数名>[(vect=<ベクタ番号>)]）

■使用例

ベクタ番号を指定してベクタテーブルを作成します。
（C/C++プログラム）
（CPU=2600a）

```
#pragma entry f1(vect=0)
void f1(){
}
#pragma interrupt (f2(vect=4))
void f2(void){
}
#pragma indirect (f3(vect=5))
unsigned char f3(void){
}
```

←エントリ関数f1をベクタ番号0に割り付ける

←割り込み関数f2をベクタ番号4に割り付ける

←メモリ間接アクセス関数f3をベクタ番号5に割り付ける

（メモリマップ内容）

```
$VECT0   00000000  00000003
$VECT4   00000010  00000013
$VECT5   00000014  00000017
```

■備考および注意事項

(i) ベクタテーブル指定“ vect= ”は必ず小文字で指定してください。
(ii) 割り付けるベクタ番号がその他のベクタテーブルと重複しないよう注意してください。
(iii) ベクタテーブル自動生成機能は、C/C++コンパイラのバージョンが4.0以降の場合にサポートしています。

3.2 組み込み関数

コンディションコードレジスタの設定など、C/C++言語仕様でサポートしていない CPU 命令を拡張組み込み関数としてサポートしています。

組み込み関数の使用の際は、必ずシステムインクルードファイル machine.h を宣言してください。

No.	項目	機能	参照
1	コンディションコードレジスタ(CCR)	割り込みマスクの設定	3.2.1
2		割り込みマスクの参照	
3		CCR の設定	
4		CCR の参照	
5		CCR の論理積	
6		CCR の論理和	
7		CCR の排他的論理和	
8	エクステンドレジスタ(EXR)	割り込みマスクの設定	3.2.2
9		割り込みマスクの参照	
10		EXR の設定	
11		EXR の参照	
12		EXR の論理積	
13		EXR の論理和	
14		EXR の排他的論理和	
15	ベクタベースレジスタ（VBR）*	VBR の設定	3.2.3
16	オーバフロー判定付き演算	1 バイト加算＋CC 反映	3.2.4
17		2 バイト加算＋CC 反映	
18		4 バイト加算＋CC 反映	
19		1 バイト減算＋CC 反映	
20		2 バイト減算＋CC 反映	
21		4 バイト減算＋CC 反映	
22		1 バイト左シフト＋CC 反映	
23		2 バイト左シフト＋CC 反映	
24		4 バイト左シフト＋CC 反映	
25		1 バイト符号反転＋CC 反映	
26		2 バイト符号反転＋CC 反映	
27		4 バイト符号反転＋CC 反映	
28	転送命令	MOVFPE 命令	3.2.5
29		MOVTPE 命令	
30	算術演算命令	10 進加算	3.2.6
31		10 進減算	
32		TAS 命令	
33		MAC 命令	
34		64bit 乗算*	
35	シフト命令	1 バイト左ローテート	3.2.7
36		2 バイト左ローテート	
37		4 バイト左ローテート	
38		1 バイト右ローテート	
39		2 バイト右ローテート	
40		4 バイト右ローテート	
41	システム制御命令	TRAPA 命令	3.2.8
42		SLEEP 命令	
43	ブロック転送命令	EEPMOV 命令	3.2.9
		割り込み要求対応 EEPMOV 命令	
44	H8SX 用ブロック転送命令	MOVMD 命令	3.2.10
		MOVSD 命令	
45	NOP 命令	NOP 命令	

【注】 * H8SX のみで使用可能です。

3.2.1 コンディションコードレジスタ（CCR）の設定・参照

■説明

コンディションコードレジスタの設定・参照について次の表の関数を用意しています。

No.	項目	書式	説明
1	割り込みマスクの設定	void set_imask_ccr(unsigned char mask)	CCR の割り込みマスクビットに mask 値（0 または 1）を設定
2	割り込みマスクの参照	unsigned char get_imask_ccr(void)	CCR の割り込みマスクビット(I)の値（0 または 1）を参照
3	CCR の設定	void set_ccr(unsigned char ccr)	CCR に ccr の値（8 ビット）を設定
4	CCR の参照	unsigned char get_ccr(void)	CCR の値を参照
5	CCR の論理積	void and_ccr(unsigned char ccr)	CCR の値と ccr の論理積を算出し、結果を CCR に設定
6	CCR の論理和	void or_ccr(unsigned char ccr)	CCR の値と ccr の論理和を算出し、結果を CCR に設定
7	CCR の排他的論理和	void xor_ccr(unsigned char ccr)	CCR の値と ccr の排他的論理和を算出し、結果を CCR に設定

■使用例

コンディションコードレジスタを操作し、その後、操作前の状態に戻します。

（C/C++プログラム）

←組み込み関数用インクルードファイル

```
#include <machine.h>
void main(void)
{
    unsigned char mask;

    if (mask=get_imask_ccr()){          /* 割り込みマスクビット(I)の値を保存 */
        set_imask_ccr(1);               /* 例外処理の実行の開始指定 */
        and_ccr((unsigned char)0xFC) ;  /* CCRと0xFCの論理積をCCRにセット */
    }
    set_imask_ccr(mask);                /* 割り込みマスクビット(I)の値を戻す */
}
```

（コンパイル結果アセンブリ展開コード）

```
  main:  STC.B      CCR,R0L
         AND.B      #-128:8,R0L
         ROTL.B     R0L
         BEQ        L48:8
         ORC.B      #-128:8,CCR
         ANDC.B     #-4:8,CCR
L48:     STC.B      CCR,R0H
         BLD.B      #0,R0L
         BST.B      #7,R0H
         LDC.B      R0H,CCR
         RTS
         .END
```

■備考および注意事項

コンディションコードレジスタは 8 ビットのレジスタで、CPU の内部状態を示しています。

<コンディションコードレジスタ>

I	UI	H	U	N	Z	V	C

(I) 割り込みマスタビット
(UI) ユーザビット／割り込みマスタビット
(H) ハーフキャリフラグ
(U) ユーザビット
(N) ネガティブフラグ
(Z) ゼロフラグ
(V) オーバフローフラグ
(C) キャリフラグ

3.2.2 エクステンドレジスタの設定・参照

■説明

エクステンドレジスタの設定・参照について次の関数を用意しています。

No.	項目	書式	説明
1	割り込みマスクの設定	void set_imask_exr(unsigned char mask)	EXR の割り込みマスクビット（I2～I0）に mask 値（0～7）を設定
2	割り込みマスクの参照	unsigned char get_imask_exr(void)	EXR の割り込みマスクビット（I2～I0）の値（0～7）を参照
3	EXR の設定	voir set_exr(unsigned char exr)	EXR に exr の値（8 ビット）を設定
4	EXR の参照	unsigned char get_exr(void)	EXR の値を参照
5	EXR の論理積	void and_exr(unsigned char exr)	EXR の値と exr の論理積を算出し、結果を EXR に設定
6	EXR の論理和	void or_exr(unsigned char exr)	EXR の値と exr の論理和を算出し、結果を EXR に設定
7	EXR の排他的論理和	void xor_exr(unsigned char exr)	EXR の値と exr の排他的論理和を算出し、結果を EXR に設定

■使用例

EXR の割り込みマスクビットの値はそのままで、EXR の状態を変更します。
（C/C++プログラム）

```
#include <machine.h>
extern unsigned char e;

void main()
{
   unsigned char mask;

   if (mask=get_imask_exr()){
       set_exr((unsigned char)0x05);
       xor_exr((unsigned char)0xff);
       e=get_exr();
   }
   set_imask_exr(mask);
}
```

←割り込みマスクビットの値を保存

←EXRの値を設定し、排他的論理積を取り、その値を外部変数eに設定

←割り込みマスクビットの値を元に戻す

（コンパイル結果アセンブリ展開コード）

```
_main:
        STC.B      EXR,R1L
        AND.B      #7:8,R1L
        BEQ        L49:8
        MOV.B      #5:8,R0L
        LDC.B      R0L,EXR
        XORC.B     #-1:8,EXR
        STC.B      EXR,R0L
        MOV.B      R0L,@_e:32
L49:
        AND.B      #7:8,R1L
        STC.B      EXR,R1H
        AND.B      #-8:8,R1H
        OR.B       R1L,R1H
        LDC.B      R1H,EXR
        RTS
        .END
```

■備考および注意事項

エクステンドレジスタの設定・参照に関する組み込み関数は、CPU/動作モードが 2600n、2600a、2000n、2000a のときのみ有効です。

<エクステンドレジスタ>

T	−	−	−	−	I2	I1	I0

(T)　　トレースビット

(I2～I0)　割り込みマスクビット

3.2.3　ベクタベースレジスタの設定

■説明

H8SX ファミリでは例外処理ベクタテーブルを任意のアドレスに配置する機能があります。H8/300,H8/300H,H8S ファミリでは例外処理ベクタテーブルは 0 番地からの配置に固定されていますが、H8SX ファミリではベクタベースレジスタ（VBR）に値を設定することによって例外処理ベクタテーブルの配置アドレスを変更することが可能です。

ベクタベースレジスタの設定について次の関数を用意しています。

No.	項目	書式	説明
1	VBR の設定	void set_vbr(void* vbr)	VBR に値（32 ビット）を設定

■使用例

ベクタベースレジスタ（VBR）の値を設定します。

（C/C++プログラム）

```
#include <machine.h>          ←組み込み関数用インクルードファイル
void main(void)
{
    set_imask_ccr(1);          /* 割り込みマスクビットを設定します。          */
    set_vbr((void*)0x20000);   /* ベクタベースレジスタに 0x20000 を設定します。 */
    set_imask_ccr(0);          /* 割り込みマスクビットをクリアします。        */
}
```

（コンパイル結果アセンブリ展開コード）

```
_main:
        ORC.B       #H'80:8,CCR
        SUB.L       ER0,ER0
        MOV.W       #2:3,E0
        LDC.L       ER0,VBR
        ANDC.B      #H'7F:8,CCR
        RTS
        .END
```

■備考および注意事項

（1） ベクタベースレジスタの設定に関する組み込み関数は、CPU/動作モードがH8SXN、H8SXM、H8SXA、H8SXXのときのみ有効です。

（2） CPU/動作モードがH8SXNの場合は、指定されたベクタベースレジスタ値の下位16ビットが有効です。

（3） ベクタテーブルのアドレス切り替えの詳細は**「3.8.3 ベクタテーブルアドレス切り替え」**をご参照ください。

（4） ベクタベースレジスタ（VBR）の切り替えは割り込み禁止状態で行う必要があります。割り込み禁止にしない状態でベクタベースレジスタ（VBR）の切り替え中に割り込みが発生した場合には正常な動作が行われません。

3.2.4 オーバフロー（V フラグ）判定付き演算

■説明

オーバフロー（V フラグ）判定付き演算の組み込み関数を示します。

（CC：コンディションコード）

No.	項目	書式	説明
1	1 バイト加算＋CC 反映	int ovfaddc(char dst,char src,char *rst)	1 バイト同士の dst＋src を行い、rst≠0 の場合のみ結果を rst の示すエリアへ格納
2	2 バイト加算＋CC 反映	int ovfaddw(int dst,int src,int *rst)	2 バイト同士の dst＋src を行い、rst≠0 の場合のみ結果を rst の示すエリアへ格納
3	4 バイト加算＋CC 反映	int ovfaddl(long dst,long src,long *rst)	4 バイト同士の dst＋src を行い、rst≠0 の場合のみ結果を rst の示すエリアへ格納
4	1 バイト減算＋CC 反映	int ovfsubc(char dst,char src,char *rst)	1 バイト同士の dst－src を行い、rst≠0 の場合のみ結果を rst の示すエリアへ格納
5	2 バイト減算＋CC 反映	int ovfsubw(int dst,int src,int *rst)	2 バイト同士の dst－src を行い、rst≠0 の場合のみ結果を rst の示すエリアへ格納
6	4 バイト減算＋CC 反映	int ovfsubl(long dst,long src,long *rst)	4 バイト同士の dst－src を行い、rst≠0 の場合のみ結果を rst の示すエリアへ格納
7	1 バイト左シフト＋CC 反映	int ovfshalc(char dst, char *rst)	1 バイト同士の dst を左方向へ算術的 1 ビットシフトし、rst≠0 の場合のみ結果を rst の示すエリアへ格納
8	2 バイト左シフト＋CC 反映	int ovfshalw(int dst, int *rst)	2 バイト同士の dst を左方向へ算術的 1 ビットシフトし、rst≠0 の場合のみ結果を rst の示すエリアへ格納
9	4 バイト左シフト＋CC 反映	int ovfshall(long dst, long *rst)	4 バイト同士の dst を左方向へ算術的 1 ビットシフトし、rst≠0 の場合のみ結果を rst の示すエリアへ格納
10	1 バイト符号反転＋CC 反映	int ovfnegc(char dst, char *rst)	1 バイト dst の 2 の補数を算出し、rst≠0 の場合のみ結果を rst の示すエリアへ格納
11	2 バイト符号反転＋CC 反映	int ovfnegw(int dst, int *rst)	2 バイト dst の 2 の補数を算出し、rst≠0 の場合のみ結果を rst の示すエリアへ格納
12	4 バイト符号反転＋CC 反映	int ovfnegl(long dst, long *rst)	4 バイト dst の 2 の補数を算出し、rst≠0 の場合のみ結果を rst の示すエリアへ格納

■使用例

加算結果がオーバフローかどうかの判定をし、それに従いそれぞれの処理を行います。
（C/C++プログラム）

```
#include <machine.h>
extern int dst, src;
void f()
{
   if (ovfaddw(dst,src,0))
      dst++;
   else
      dst--;
}
```

←dstとsrcの加算がオーバフローかどうかの判定

（コンパイル結果アセンブリ展開コード）

```
_f:
        PUSH.L     ER6
        MOV.L      #_dst:32,ER6
        MOV.W      @ER6,R0
        MOV.W      @_src:32,R1
        ADD.W      R1,R0
        BVC        L48:8
        MOV.W      @ER6,R0
        INC.W      #1,R0
        BRA        L50:8
L48:
        MOV.W      @ER6,R0
        DEC.W      #1,R0
L50:
        MOV.W      R0,@ER6
        POP.L      ER6
        RTS
        .END
```

■備考および注意事項

コンディションコード演算の関数は、if 文、do 文、while 文、for 文の条件を判定する式でのみ指定することが可能です。

3.2.5 転送命令

■説明

システム制御転送命令については次の関数を用意しています。

No.	項目	書式	説明
1	MOVFPE 命令	void movfpe(char *addr,char data) char _movfpe(char *addr) *1	E クロック同期データ転送命令 MOVFPE に展開
2	MOVTPE 命令	void movtpe(char data ,char *addr)	E クロック同期データ転送命令 MOVTPE に展開

【注】 *1 H8SX のみで使用可能です。

■使用例

(a) MOVFPE 命令

16 ビット絶対アドレスで指定されるメモリの内容を E クロックに同期したタイミングで取り込みます。
_movfpe も取り出した値をリターン値として返却するだけで movfpe と同様な関数です。
（C/C++プログラム）

```
#include <machine.h>
#define P1DR (*(unsigned char *)0x00FFFF60)
extern unsigned char data;
void f()
{
   movfpe((char*)&P1DR,data);
}
```

←MOVFPE命令を発行

（コンパイル結果アセンブリ展開コード）

```
_f:
        MOVFPE.B    @16777056:16,R0L
        MOV.B       R0L,@_data:32
        RTS
        .END
```

（C/C++プログラム）

```
#include <machine.h>
#define P1DR (*(unsigned char *)0x00FFFF60)
extern unsigned char data;
void f()
{
   data = _movfpe((char*)&P1DR);
}
```

←MOVFPE命令を発行

（コンパイル結果アセンブリ展開コード）

```
_f:
        MOVFPE.B    @16777056:16,R0L
        MOV.B       R0L,@_data:32
        RTS
        .END
```

(b) MOVTPE 命令

データを E クロックに同期したタイミングで 16 ビット絶対アドレスで指定されるエリアへ設定します。
（C/C++プログラム）

```
#include <machine.h>
extern unsigned char data;
#define P1DR (*(unsigned char*)0x00FFFF60)
void f()
{
   movtpe(data,(char*)&P1DR);
}
```

←MOVTPE命令を発行

（コンパイル結果アセンブリ展開コード）

```
_f:
        MOV.B       @_data:32,R0L
        MOVTPE.B    R0L,@16777056:16
        RTS
        .END
```

3.2.6 算術演算命令

■説明

算術演算命令について次の関数を用意しています。

No.	項目	書式	説明
1	10 進加算	void dadd(unsiged char size, char*ptr1, char*ptr2, char*rst)	ptr1 からはじまる size バイトのデータと、ptr2 から始まる size バイトのデータの 10 進加算を行い、結果を rst から始まる size バイトのエリアへ格納
2	10 進減算	void dsub(unsiged char size, char*ptr1, char*ptr2, char*rst)	ptr1 からはじまる size バイトのデータと、ptr2 から始まる size バイトのデータの 10 進減算を行い、結果を rst から始まる size バイトのエリアへ格納
3	TAS 命令	void tas(char*addr)	テスト・アンド・セット命令 TAS に展開
4	MAC 命令	long mac(long val, int*ptr1, int*ptr2, unsigned long count) long macl(long val, int*ptr1, int*ptr2, unsigned long count, unsigned long mask)	積和演算 MAC 命令に展開
5	64bit 乗算*1	long mulsu(long val1, long val2) unsigned long muluu(unsigned long val1, unsigned long val2)	MULS/U,MULU/U 命令に展開

【注】 *1 H8SX のみで使用可能です。

■使用例

(1) 10 進演算

ptr1 からはじまる 6 桁の 4 ビット BCD データ(3 バイト)と、ptr2 からはじまる 4 ビット BCD データの 10 進加算を行い、結果を rst からはじまる 3 バイトのエリアへ格納します。

（C/C++プログラム）

```
#include <machine.h>
char ptr1[3]={0,1,2};
char ptr2[3]={2,1,0};
char rst[3];
void f()
{
   dadd((char)3,ptr1,ptr2,rst);
}
```

←DAA命令出力

（コンパイル結果アセンブリ展開コード）

```
_f:     STM.L     (ER4-ER6),@-SP
        MOV.L     #_ptr1+2:32,ER0
        MOV.L     #_ptr2+2:32,ER1
        MOV.L     #_rst+3:32,ER5
        MOV.B     #3:8,R6L
        ANDC.B    #-34:8,CCR
L49:    MOV.B     @ER0,R4L
        MOV.B     @ER1,R4H
        ADDX.B    R4H,R4L
        DAA.B     R4L
        MOV.B     R4L,@-ER5
        DEC.L     #1,ER0
        DEC.L     #1,ER1
        DEC.B     R6L
        BNE       L49:8
        LDM.L     @SP+,(ER4-ER6)
        RTS
```

■備考および注意事項
関数 dadd、dsub の第一引数は 1～255 の定数です。

(2) TAS 命令

メモリの内容をテスト(0 と比較)した後、メモリの内容の最上位ビット(ビット 7)を“1”にセットします。

（C/C++プログラム）

```
extern unsigned char data;
#define ADR (*(volatile unsigned char *)0x00fff000)
#include <machine.h>
void main()
{
   tas((char*)&ADR);

   if (data=get_ccr())
      and_ccr(data);
   else
      or_ccr(data);
}
```

←メモリの内容を0と比較して結果をCCRに反映

←メモリの内容によりCCRに論理積または論理和を格納する

（コンパイル結果アセンブリ展開コード）

```
_main:  MOV.L      #16773120:32,ER0
        TAS        @ER0
        MOV.L      #_data:32,ER1
        STC.B      CCR,R0L
        MOV.B      R0L,@ER1
        BEQ        L47:8
        MOV.B      @ER1,R1L
        STC.B      CCR,R1H
        AND.B      R1L,R1H
        LDC.B      R1H,CCR
        RTS
L47:    MOV.B      @ER1,R1L
        STC.B      CCR,R1H
        OR.B       R1L,R1H
        LDC.B      R1H,CCR
        RTS
        .END
```

■備考および注意事項

関数 tas は、CPU/動作モードが、2600a、2600n、2000a、2000n の時のみ有効です。

(3) MAC 命令

H8S/2600 マイコンには、積和レジスタ（MAC）があります。これは、64 ビットのレジスタで積和演算結果を格納します。積和レジスタの構成を示します。

MAC 命令は、メモリとメモリ間の乗算を行い、結果を MAC レジスタに加算します。このレジスタを利用して、16 ビット×16 ビット＋32 ビット＝32 ビットの積和演算ができます。

以下の例では、次のような解釈を行います。

＜関数 mac＞

100 を MAC レジスタに初期値とし、次に ptr1、ptr2 で示される 2 バイトのデータを符号付きで乗算し、結果の 4 バイトデータを MAC レジスタに加算後、ptr1 と ptr2 を共に+2 します。これを 4 回繰り返します。最後に MAC レジスタの内容を返します。

＜関数 macl＞

ptr2 のデータをリングバッファとして使用するため、~4 との積和演算を行います。

ptr2&mask をアドレスとして使用するため、ptr2 は 8 の倍数アドレスに割り付ける必要があります。

（C/C++プログラム）

```
#include <machine.h>
int ptr1[10]={0,1,2,3,4,5,6,7,8,9};
int ptr2[10]={9,8,7,6,5,4,3,2,1,0};
int ptr3[2]={9,8};
long l1,l2;
void func()
{
    l1=mac(100,ptr1,ptr2,4);
                              /* l1=100+0*9+1*8+2*7+3*6 */
    l2=macl(100,ptr1,ptr3,4,~4);
                              /* l2=100+0*9+1*8+2*9+3*8 */
}
```

←積和演算

（コンパイル結果アセンブリ展開コード）

```
_func:   PUSH.L     ER2
         MOV.L      #100:32,ER0
         CLRMAC
         LDMAC.L    ER0,MACL
         MOV.L      #_ptr2:32,ER0
         MOV.L      #_ptr1:32,ER1
         MAC        @ER1+,@ER0+
         MAC        @ER1+,@ER0+
         MAC        @ER1+,@ER0+
         MAC        @ER1+,@ER0+
         STMAC.L    MACL,ER0
         MOV.L      ER0,@_l1:32
         MOV.L      #100:32,ER0
         CLRMAC
         LDMAC.L    ER0,MACL
         MOV.L      #_ptr3:32,ER0
         MOV.L      #_ptr1:32,ER1
         MOV.L      #-5:32,ER2
         MAC        @ER1+,@ER0+
         AND.L      ER2,ER0
         MAC        @ER1+,@ER0+
         AND.L      ER2,ER0
         MAC        @ER1+,@ER0+
         AND.L      ER2,ER0
         MAC        @ER1+,@ER0+
         AND.L      ER2,ER0
         STMAC.L    MACL,ER0
         MOV.L      ER0,@_l2:32
         POP.L      ER2
         RTS
```

備考および注意事項
関数 mac、macl **は** CPU/**動作モードが** 2600a、2600n、H8SX **のときのみ使用できます。**

(4) MULS/U,MULU/U 命令

32bit×32bit=64bit 乗算を行う MULS/U,MULU/U 命令へ展開します。
本組み込み関数の 32bit の各引数（val1,val2）同士を乗算し、64bit 乗算結果の上位 32bit を演算結果として返します。
（C/C++プログラム）

```
#include <machine.h>
long sval1, sval2, sans;
unsigned long uval1, uval2, uans;
void f(void)
{
    sans = mulsu(sval1, sval2);

    uans = muluu(uval1, uval2);
}
```

←符号付き32bit乗算結果の上位32bitを取得

←符号なし32bit乗算結果の上位32bitを取得

（コンパイル結果アセンブリ展開コード）

```
_f:
        PUSH.L      ER2
        MOV.L       @sval1:32,ER1
        MOV.L       @sval2:32,ER2
        MULS/U.L    ER2,ER1
        MOV.L       ER1,@sans:32
        MOV.L       @uval1:32,ER1
        MOV.L       @uval2:32,ER2
        MULU/U.L    ER2,ER1
        MOV.L       ER1,@uans:32
        RTS/L       ER2
```

■備考および注意事項

関数 mulsu、muluu は、CPU が H8SX で乗算器指定（H8SX*:{M|MD}）をしている時に使用可能です。

3.2.7 シフト命令

■説明

ローテート命令の組み込み関数は次のようになっています。

No.	項目	書式	説明
1	1 バイト左ローテート	char rotlc(int count,char data)	1 バイトの data を左方向に count ビット分ローテートし、結果を返す
2	2 バイト左ローテート	int rotlw(int count,int data)	2 バイトの data を左方向に count ビット分ローテートし、結果を返す
3	4 バイト左ローテート	long rotll(int count,long data)	4 バイトの data を左方向に count ビット分ローテートし、結果を返す
4	1 バイト右ローテート	char rotrc(int count,char data)	1 バイトの data を右方向に count ビット分ローテートし、結果を返す
5	2 バイト右ローテート	int rotrw(int count,int data)	2 バイトの data を右方向に count ビット分ローテートし、結果を返す
6	4 バイト右ローテート	long rotrl(int count,long data)	4 バイトの data を右方向に count ビット分ローテートし、結果を返す

■使用例

data の内容にビット群をローテート(回転)します。

（C/C++プログラム）

```
#include <machine.h>
extern unsigned char data;
char i;
void func()
{
   i=rotlc(2,data);
}
```

←2ビット左ローテート

（コンパイル結果アセンブリ展開コード）

```
_func:
        MOV.B       @_data:32,R0L
        ROTL.B      #2,R0L
        MOV.B       R0L,@_i:32
        RTS
        .SECTION    B,DATA,ALIGN=2
_i:
        .RES.B      1
```

3.2.8 システム制御命令

■説明

システム制御命令について以下の関数を用意しています。

No.	項目	書式	説明
1	TRAPA 命令	void trapa(unsigned int trap_no)	無条件トラップの TRAPA #trap_no に展開
2	SLEEP 命令	void sleep(void)	低消費電力状態命令 SLEEP に展開

■使用例

(1) TRAPA 命令

指定したベクタテーブル番号 0 に対応するベクタアドレスの内容によって示されるアドレスへ分岐します。

（C/C++プログラム）

```
#include <machine.h>
#define dummy (void*)0
extern void f1(void);
extern void f2(void);
extern void f3(void);
void (*const vect_table[])(void)={
   f1,dummy,f2,f3
};
void func()
{
   trapa(0);
}
```

←関数f1へのトラップ命令

注：この場合、ベクタテーブルを割り込みベクタアドレスに割り付ける必要があります。

（コンパイル結果アセンブリ展開コード）

```
_func:
        TRAPA      #0
        RTS
        .SECTION    C,DATA,ALIGN=2
_vect_table:
        .DATA.L     _f1
        .DATA.L     H'00000000
        .DATA.L     _f2,_f3
        .END
```

■備考および注意事項

(i) 関数trapaの引数は0～3の定数のみ指定できます。

(ii) CPU／動作モードが300以外の時のみ有効です。

(2) SLEEP 命令

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 の内部状態を保持指定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/C++プログラム）

```
#include <machine.h>
extern int a;
void func()
{
   while(a);
   sleep();
}
```

←SLEEP命令の発行

（コンパイル結果アセンブリ展開コード）

```
_func:
        MOV.W       @_a:32,R0
L49:
        BNE         L49:8
        SLEEP
        RTS
```

3.2.9 ブロック転送命令

■説明

システム制御のブロック転送命令について次の関数を用意しています。

No.	項目	書式	説明
1	EEPMOV 命令	void eepmov(void*dst, const void*src, unsigned char size) void eepmov(void*dst, const void*src, unsigned int size)	ブロック転送命令 EEPMOV に展開
		void eepmovb(void*dst, const void*src, unsigned char size) *1	常にブロック転送命令 EEPMOV.B に展開 size に変数も指定可能
		void eepmovw(void*dst, const void*src, unsigned int size) *1	常にブロック転送命令 EEPMOV.W に展開 size に変数も指定可能
		void eepmovi(void*dst, const void*src, unsigned int size) *1	ブロック転送命令 EEPMOV に展開 割り込み発生後も継続可能 size に変数も指定可能
2	EEPMOV 命令 (ECR 設定あり)	void eepromb(void*dst, const void*src, unsigned char size, volatile unsigned char*ecr, unsigned char ecrval) void eepromw(void*dst, const void*src, unsigned int size, volatile unsigned char*ecr, unsigned char ecrval)	ECR へ値を設定 ブロック転送命令 EEPMOV.B,EEPMOV/P.W に展開 size に変数も指定可能
	EEPMOV 命令 (EPR,ECR 設定あり)	void eepromb_epr(void*dst, const void*src, unsigned char size, volatile unsigned char*ecr, unsigned char ecrval, volatile unsigned char*epr, unsigned char eprval) void eepromw_epr(void*dst, const void*src, unsigned int size, volatile unsigned char*ecr, unsigned char ecrval, volatile unsigned char*epr, unsigned char eprval)	ECR,EPR へ値を設定 ブロック転送命令 EEPMOV.B,EEPMOV/P.W に展開 size に変数も指定可能

【注】 *1 H8SX のみで使用可能です。

■使用例

(1) eepmov, eepmovb, eepmovw

第 2 引数で示されるアドレスから、第 3 引数で示されるバイト数分、第 1 引数で示すアドレスへブロック転送します。

（C/C++プログラム）

```
#include <machine.h>
struct STR{
   char a[300];
}ST1;
struct STR ST2={0};
void f()
{
   eepmov((char*)&ST1,(char*)&ST2,255);
}
```

←EEPMOV命令を発行

（コンパイル結果アセンブリ展開コード）

```
_f:
        STM.L      (ER4-ER6),@-SP
        MOV.L      #_ST2:32,ER5
        MOV.B      #-1:8,R4L
        MOV.L      #_ST1:32,ER6
        EEPMOV.B
        LDM.L      @SP+,(ER4-ER6)
        RTS
```

■備考および注意事項

(i) CPU/動作モードが300のとき、ブロック転送されるサイズは最大255の制限があります。

(ii) CPU/動作モードが300以外のとき、ブロック転送されるサイズは最大65535の制限があります。ただし、256～65535のときは、EEPMOV.Wに展開されますのでNMI割り込みに注意する必要があります。
この割り込みに関する詳細は各製品ソフトウェアマニュアルを参照してください。

(2) eepmovi

第 2 引数で示されるアドレスから、第 3 引数で示されるバイト数分、第 1 引数で示すアドレスへブロック転送します。

本関数を使用すると EEPMOV 命令実行中に割り込みが発生した場合でも、割り込みから復帰後に継続可能な形で命令展開されます。

（C/C++プログラム）

```
#include <machine.h>
struct STR{
   char a[300];
}ST1;
struct STR ST2={0};
void f()
{
   eepmovi((char*)&ST1,(char*)&ST2,256);
}
```

←EEPMOV命令を発行

（コンパイル結果アセンブリ展開コード）

```
_f:
        STM.L      (ER4-ER6),@-SP
        MOV.L      #_ST1,ER6
        MOV.L      #_ST2,ER5
        MOV.W      #256:16,R4
L28:
        EEPMOV.W
        MOV.W      R4,R4
        BNE        L28:8
        RTS/L      (ER4-ER6)
```

←転送サイズの残りが0になるまで継続

■備考および注意事項

CPU が H8SX のみ使用できます。

(3) eepromb,eepromw

第 2 引数で示されるアドレスから、第 3 引数で示されるバイト数分、第 1 引数で示すアドレスへブロック転送します。

eepromb 関数は EEPMOV.B 命令、eepromw 関数は EEPMOV/P.W 命令によって、それぞれブロック転送命令展開を行います。

本組み込み関数は第 1,2,3 引数をレジスタに設定し、ecr の指すアドレスへ ecrval を設定してからブロック転送命令を実行します。

転送に成功した場合は 0 を、失敗した場合は転送されなかった分のデータサイズが戻ります。

eepromb 関数は size に 0 から 255 まで、eepromw 関数は size に 0 から 65535 まで指定できます。size に 0 を指定した場合転送命令を出力しません。

（C/C++プログラム）

```
#include <machine.h>
#define ecr_ptr ((volatile unsigned char *)(0x123456))
char a[10], b[10];
unsigned char x;
void f(void)
{
   x = eepromw(b, a, 10, ecr_ptr, 1);
}
```

←EEPMOV/P.W命令を発行

（コンパイル結果アセンブリ展開コード）

```
_f:
        STM.L      (ER4-ER6),@-SP
        MOV.L      #_b,ER6
        MOV.L      #_a,ER5
        MOV.W      #H'000A:16,R4
        MOV.B      #1:4,@H'00123456:32
        EEPMOV/P.W
        MOV.B      R4L,@_x:32
        RTS/L      (ER4-ER6)
```

■備考および注意事項

(i) 本組み込み関数はCPU種別がAE5、またはH8SXで-eepromオプションを指定した場合にのみ有効です。

(ii) ECR、EPR、およびその他関連事項についてはハードウェアマニュアルを参照してください。

(4) eepromb_epr,eepromw_epr

第 2 引数で示されるアドレスから、第 3 引数で示されるバイト数分、第 1 引数で示すアドレスへブロック転送します。

eepromb_epr 関数は EEPMOV.B 命令、eepromw_epr 関数は EEPMOV/P.W 命令によって、それぞれブロック転送命令展開を行います。本組み込み関数は第 1,2,3 引数をレジスタに設定し、epr の指すアドレスへ eprval を設定し、ecr の指すアドレスへ ecrval を設定してからブロック転送命令を実行します。

転送に成功した場合は 0 を、失敗した場合は転送されなかった分のデータサイズが戻ります。

eepromb_epr 関数は size に 0 から 255 まで、eepromw_epr 関数は size に 0 から 65535 まで指定できます。size に 0 を指定した場合転送命令を出力しません。

（C/C++プログラム）

```
#include <machine.h>
#define ecr_ptr ((volatile unsigned char *)(0x123456))
#define epr_ptr ((volatile unsigned char *)(0x123457))
char a[10], b[10];
unsigned char x;
void f(void)
{
   x = eepromw_epr(b, a, 10, ecr_ptr, 1, epr_ptr, 1);
}
```

←EEPMOV/P.W命令を発行

（コンパイル結果アセンブリ展開コード）

```
_f:
        STM.L      (ER4-ER6),@-SP
        MOV.L      #_b,ER6
        MOV.L      #_a,ER5
        MOV.W      #H'000A:16,R4
        MOV.B      #1:4,@H'00123457:32
        MOV.B      #1:4,@H'00123456:32
        EEPMOV/P.W
        MOV.B      R4L,@_x:32
        RTS/L      (ER4-ER6)
```

■備考および注意事項

(i) 本組み込み関数はCPU種別がAE5、またはH8SXで-eepromオプションを指定した場合にのみ有効です。

(ii) ECR、EPR、およびその他関連事項についてはハードウェアマニュアルを参照してください。

3.2.10 H8SX 用ブロック転送命令

■説明

H8SX 用のブロック転送命令について次の関数を用意しています。

No.	項目	書式	説明
1	MOVMD 命令	void movmdb(void*dst, const void*src, unsigned int count) void movmdw(int*dst, const int*src, unsigned int count) void movmdl(long*dst, const long*src, unsigned int count)	MOVMD に展開
2	MOVSD 命令	unsigned int movsd(char*dst, const char*src, unsigned int size)	MOVSD に展開

■使用例

(1) movmdb, movmdw, movmdl

MOVMD.B,MOVMD.W,MOVMD.L 命令は、それぞれ 1 バイト、2 バイト、4 バイトのメモリブロックを **count** で指定した回数分だけ **src** で示すアドレスから、**dst** で示すアドレスへ転送します。下記の例の場合、100 バイトの転送を movmdb は 1 バイトずつ 100 回、movmdw は 2 バイトずつ 50 回、movmdl は 4 バイトずつ 25 回で転送しています。

（C/C++プログラム）

```
#include <machine.h>
char s1[100], d1[100];
int s2[50], d2[50];
long s4[25], d4[25];
void f(void)
{
   movmdb(d1, s1, 100);
   movmdw(d2, s2, 50);
   movmdl(d4, s4, 25);
}
```

←MOVMD命令を発行

（コンパイル結果アセンブリ展開コード）

```
_f:
        STM.L      (ER4-ER6),@-SP
        MOV.L      #_d1,ER6
        MOV.L      #_s1,ER5
        MOV.W      #100:16,R4
        MOVMD.B
        MOV.L      #_d2,ER6
        MOV.L      #_s2,ER5
        MOV.W      #50:16,R4
        MOVMD.W
        MOV.L      #_d4,ER6
        MOV.L      #_s4,ER5
        MOV.W      #25:16,R4
        MOVMD.L
        RTS/L      (ER4-ER6)
```

■備考および注意事項

(i) CPUがH8SXのとき使用可能です。

(ii) **count**には0から65535まで指定することができます。0を指定した場合、転送回数は65536となります。

(2) movsd

ブロック転送命令 movsd によって、**src** で示すアドレスから **dst** で示すアドレスへブロック転送を行います。ただし、データとして 0(H'00)を転送した時点、または **size** で示されるバイト数分だけ転送した時点で終了します。リターン値は **size** から実際に転送したバイト数を引いた値です。

（C/C++プログラム）

```
#include <machine.h>
const char *s = "1234";
char d[100];
unsigned int remain;
void f(void)
{
   remain = movsd(d, s, 100);
}
```

←MOVSD命令を発行、最大100バイト転送可

（コンパイル結果アセンブリ展開コード）

```
_f:
        STM.L      (ER4-ER6),@-SP
        MOV.L      # d,ER6
        MOV.L      @ s:32,ER5
        MOV.W      #100:16,R4
        MOVSD.B    ($+4)
        MOV.W      R4,@ remain:32
        RTS/L      (ER4-ER6)
```

←転送後に最大転送サイズの残りを格納

■備考および注意事項

(i) CPUがH8SXのとき使用可能です。

(ii) **size**には0から65535まで指定することができます。0を指定した場合、最大転送バイト数が65536となります。

3.3 セクションアドレス演算子

■説明

コンパイラで用意している__sectop 演算子、__secend 演算子を利用すると、セクションのアドレスを指定することができます。

コンパイラの出力するオブジェクトでは、セクションの割り付け先が不明のため、セクションのアドレスを指定することはできませんが、__sectop 演算子、__secend 演算子を使用すると、モジュール間最適化ツールでリンクされた後の状態のセクションのアドレスを指定することができます。

この 2 つの演算子は次のように指定します。

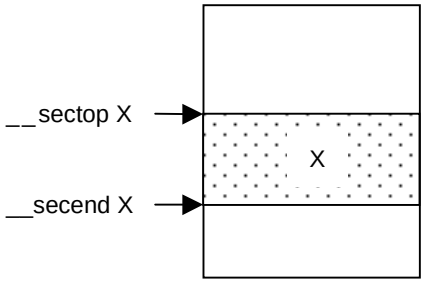
【書式】

__sectop(“<セクション名>”)
__secend(“<セクション名>”)

セクション名をXとすると、この演算子を使用すると次のように展開されます。

__sectop(“X”) → STARTOF X
__secend(“X”) → STARTOF X+SIZEOF X

セクション状態例



STARTOF、SIZEOF はアセンブラの演算子です。
STARTOF はリンク処理が済んだセクション集合の先頭アドレスを求めます。
SIZEOF はリンク処理が済んだセクション集合のサイズを求めます。

■使用例

セクション X の内容をセクション Y へコピーします。
（C/C++プログラム）

```
char *X_BGN;
char *X_END;
char *Y_BGN;
void func(void)
{
    char *p, *q;

    X_BGN=(char *)__sectop("X");
    X_END=(char *)__secend("X");
    Y_BGN=(char *)__sectop("Y");

    for (p=X_BGN,q=Y_BGN;p<X_END;p++,q++)
        *q = *p;
}
```

（コンパイル結果アセンブリ展開コード）

```
_func:
     STM.L      (ER4-ER5),@-SP
     MOV.L      #_X_END:32,ER4
     MOV.L      #STARTOF X:32,ER0
     MOV.L      ER0,@_X_BGN:32
     MOV.L      #STARTOF X+SIZEOF X:32,ER0
     MOV.L      ER0,@ER4
     MOV.L      #STARTOF Y:32,ER0
     MOV.L      ER0,@_Y_BGN:32
     MOV.L      @_X_BGN:32,ER1
     MOV.L      ER0,ER5
     BRA        L12:8
L11:
     MOV.B      @ER1,R0L
     MOV.B      R0L,@ER5
     INC.L      #1,ER1
     INC.L      #1,ER5
L12:
     MOV.L      @ER4,ER0
     CMP.L      ER0,ER1
     BCS        L11:8
     LDM.L      @SP+,(ER4-ER5)
     RTS
```

■備考

セクションアドレス演算子で指定したセクションが存在しない場合、サイズ 0 のセクションを作成します。このセクションの属性はデータで、境界調整数は 2 となります。

3.4 C++言語使用時の設定

C++言語を使用した場合は、C 言語での設定のほかに次のような設定が必要となります。

3.4.1 EC++クラスライブラリの設定

HEW1.2 では、C++言語を使用した場合は、標準ライブラリのほかに EC++クラスライブラリをリンクする必要があります。EC++クラスライブラリも、標準ライブラリ同様、CPU 種別、最適化の目的、引数渡し用レジスタ数の指定で次のように選択されます。標準ライブラリの指定、およびコンパイラのオプションと異なる EC++クラスライブラリをリンクすることはできません。

HEW2.0 以降では、標準ライブラリ作成ツールにて EC++クラスライブラリを作成してください。

設定方法は、Standard Library タブ Category:[Standard Liblary] EC++を選択してください。

CPU Series:	Operating Mode:	Merit of Library:	Change number of parameter …	EC++ClassLibrary
H8S/2600	Normal	Code Size	2	ec226n.lib
		Speed	2	ec226ns.lib
		Code Size	3	ec226n3.lib
		Speed	3	ec226ns3.lib
	Advanced	Code Size	2	ec226a.lib
		Speed	2	ec226as.lib
		Code Size	3	ec226a3.lib
		Speed	3	ec226as3.lib
H8S/2000	Normal	Code Size	2	ec226n.lib
		Speed	2	ec226ns.lib
		Code Size	3	ec226n3.lib
		Speed	3	ec226ns3.lib
	Advanced	Code Size	2	ec226a.lib
		Speed	2	ec226as.lib
		Code Size	3	ec226a3.lib
		Speed	3	ec226as3.lib
H8/300H	Normal	Code Size	2	ec2hn.lib
		Speed	2	ec2hns.lib
		Code Size	3	ec2hn3.lib
		Speed	3	ec2hns3.lib
	Advanced	Code Size	2	ec2ha.lib
		Speed	2	ec2has.lib
		Code Size	3	ec2ha3.lib
		Speed	3	ec2has3.lib
H8/300	-	Code Size	2	ec2reg.lib
		Speed	2	ec2regs.lib
		Code Size	3	ec2reg3.lib
		Speed	3	ec2regs3.lib
H8/300L	-	Code Size	2	ec2reg.lib
		Speed	2	ec2regs.lib
		Code Size	3	ec2reg3.lib
		Speed	3	ec2regs3.lib

3.4.2 初期設定方法の変更

C++言語では初期設定を次のように変更する必要があります。

ここでは、**「2.1.1 ワークスペースの新規作成 1（HEW1.2）」**で新規に作成したワークスペースの resetprg.c ファイルを使って変更方法を説明いたします。

```
#include   <machine.h>
#include   "stacksct.h"                                    ：追加箇所

#pragma entry PowerON_Reset
extern void main(void);
#ifdef __cplusplus
extern “C” {
#endif
extern void _INITSCT(void);
#ifdef USES_SIMIO
extern void _INIT_IOLIB(void);
extern void _CLOSEALL(void);
#endif
#ifdef OTHERLIB
extern void _INIT_OTHERLIB(void);
#endif
#ifdef HWSETUP
extern void HardwareSetup(void);
#endif
#ifdef __cplusplus
}
#endif

#pragma section ResetPRG
void PowerON_Reset(void);
void PowerON_Reset(void)
{
     set_imask_ccr(1);
    _INITSCT();
#ifdef USES_SIMIO
    _INIT_IOLIB();
#endif
#ifdef OTHERLIB
    _INIT_OTHERLIB();
#endif
#ifdef HWSETUP
    HardwareSetup();
#endif
    _call_init();
    main();
#ifdef USES_SIMIO
    _CLOSEALL();
#endif

    _call_end():
    sleep();
}
```

C++プログラム中に静的データが存在する場合に呼び出します

_call_init 関数は C++初期処理データ領域を初期化します。この領域にはグローバルクラスオブジェクトに対して呼び出されるコンストラクタのアドレスを格納します。

_call_end 関数は C++後処理データ領域を初期化します。この領域にはグローバルクラスオブジェクトに対して呼び出されるデストラクタのアドレスを格納します。

双方とも、標準ライブラリとして提供されています。

3.4.3 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、または#pragma pack1／#pragma pack2／#pragma unpack を使用し、変更することができます。

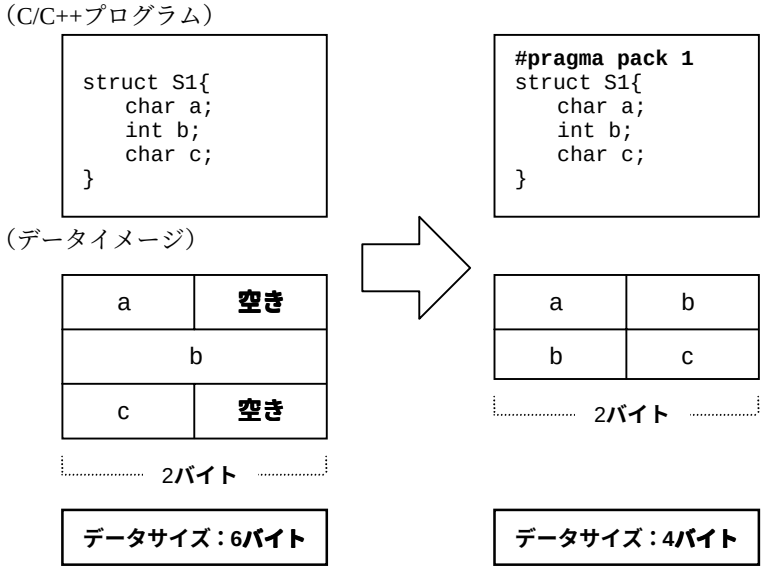
これらの指定による境界調整数は次のようになります。

指定	#pragma pack1	#pragma pack2	#pragma unpack **または指定なし**
[unsigned]char	1	1	1
[unsigned]short、[unsigned]int、[unsigned]long、**浮動小数点型、ポインタ型**	1	2	pack **オプション指定**
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	pack **オプション指定**

■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。



■注意事項

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。

ただし、CPU が H8SX の場合はワード、ロングワードのメンバが奇数番地にあっても、デバイスの仕様上ワードアクセスしてもアドレスエラーにならないため、ワード、ロングワード命令でアクセスできます。

よって、コードが増大することもありません。

また、H8SX 以外は下記の例のように、構造体メンバをポインタで参照すると正しくメンバアクセスができないので注意が必要です。

（C/C++プログラム）

```
struct S {
    char x;
    int y;
} s;
int *p=&s.y;
void test()
{
    s.y=1;

    *p =7;
}
```

←s.y は奇数アドレスの可能性があります

←正しくアクセスできます

←正しくアクセスできません

3.5 コンパイラ Ver4.0 新規拡張機能

コンパイラ Ver4.0 より新たに追加された拡張機能を説明します。

3.5.1 ベクタテーブル自動生成機能

■説明

#pragma interrupt、#pragma indirect、#pragma entry のベクタ番号指定を用いて自動的に関数のベクタテーブルを生成できます。

【書式】

#pragma interrupt (<関数名>[(vect=<ベクタ番号>)])
#pragma indirect (<関数名>[(vect=<ベクタ番号>)])
#pragma entry <関数名>[(vect=<ベクタ番号>)]

■使用例

ベクタ番号を指定してベクタテーブルを作成します。

（C/C++プログラム）

（CPU=2600a）

```
#pragma entry f1(vect=0)
void f1(){
}
#pragma interrupt (f2(vect=4))
void f2(void){
}
#pragma indirect (f3(vect=5))
unsigned char f3(void){
}
```

←エントリ関数f1をベクタ番号0に割り付ける

←割り込み関数f2をベクタ番号4に割り付ける

←メモリ間接アクセス関数f3をベクタ番号5に割り付ける

（メモリマップ内容）

```
$VECT0   00000000  00000003
$VECT4   00000010  00000013
$VECT5   00000014  00000017
```

■備考および注意事項

(i) ベクタテーブル指定 " vect= " は必ず小文字で指定してください。

(ii) 割り付けるベクタ番号がその他のベクタテーブルと重複しないよう注意してください。

3.5.2 引数用レジスタ数指定

■説明

関数ごとに引数用レジスタ数を指定することができます。

_ _ regparam2 で指定された関数は ER0,ER1(H8/300 時は R0,R1)、_ _ regparam3 で指定された関数は ER0,ER1,ER2 (H8/300 時は R0,R1,R2)を使用します。

【書式】

<型指定子> _ _ regparam2 <関数名>

<型指定子> _ _ regparam3 <関数名>

■使用例

レジスタ数指定により変数が stack に格納するか ER2 に割り付くか指定することができます。

（C/C++プログラム）

```
void _ _ regparam2 func1(long a, int b, int c, long d);
void _ _ regparam3 func2(long a, int b, int c, long d);
void main(void)
{
          ::
       func1(a,b,c,d);
         :
         :
         :
       func2(a,b,c,d);
         :
         :
         :
}
```

変数の割り付けパターン(CPU=2600a)

```
func1
        long a  :ER0
        int b   :E1
        int c   :R1
        long d  :stack
func2
        long a  :ER0
        int b   :E1
        int c   :R1
        long d  :ER2
```

■備考および注意事項

(i) 本機能は、キーワード指定のみサポートしています。

(ii) コンパイラCPUオプションregparam=3では、引数用レジスタはすべての関数に対してER0,ER1,ER2(H8/300 時は R0,R1,R2)を使用します。

3.5.3 偶数バイトアクセス指定機能

■説明

2 または 4 バイトのスカラ型の変数，定数に対して，バイトアクセスせず、必ず偶数バイトでアクセスします。

【書式】

_ _ evenaccess <型指定子> <変数名>

<型指定子> _ _ evenaccess <変数名>

■使用例

（C/C++プログラム）

```
#define A (*(volatile unsigned short __evenaccess
*)0xff01178)
void test(void)
{
    A &= ~0x2000 ;
}
```

（コンパイル結果アセンブリ展開コード）

__evenaccess 未指定時

```
_test:
     BCLR.B    #5,@15733112:32
     RTS
```

__evenaccess 指定時

```
_test:
     MOV.W     @15733112:32,R0
     BCLR.B    #5,R0H
     MOV.W     R0,@15733112:32
     RTS
```

ワード命令でアクセスをしている

■備考および注意事項

(i) H8/300では、共に2byte単位でアクセスします。
(ii) 本機能は、キーワード指定のみサポートしています。

3.6 コンパイラ Ver6.0 新規拡張機能

コンパイラ Ver6.0 より新たに追加された拡張機能を説明します。

3.6.1 ビットフィールド並び順指定

■説明

#pragma bit_order、bit_order オプション指定を使用すると、ビットフィールドの並び順を変更することができます。

マイコンによってはビットフィールドの並び規則が違うものがあるので、本機能を使用すると他のマイコンで動作していたプログラムの移植性が向上します。本オプションの省略時解釈は、BIt_order = Left です。

(1) 拡張機能での書式
#pragma bit_order (left | right)
(2) オプションでの指定
BIt_order = {Left | Right}

■使用例

ビットフィールドの割り付け例です。
#pragma bit_order left と#pragma bit_order right で（データ並びイメージ）のように、それぞれ左右に割り付きます。
left と right を省略して#pragma bit_order を記述すると、それより先の行は bit_order オプションの指定に従います。
（C/C++プログラム）

【左からメンバ割り付け】

```
#pragma bit_order left
struct {
    unsigned char a:2;
    unsigned char b:3;
}x;
void func(void)
{
    x.a = 3;
    x.b = 5;
}
```

【右からメンバ割り付け】

```
#pragma bit_order right
struct {
    unsigned char a:2;
    unsigned char b:3;
}x;
void func(void)
{
    x.a = 3;
    x.b = 5;
}
```

（データ並びイメージ）

3.7 コンパイラ Ver6.1 新規オプション、拡張機能

コンパイラ Ver6.1 より新たに追加されたオプション,拡張機能を説明します。

3.7.1 legacy=v4

■説明

本オプションを指定すると、C/C++コンパイラ Ver4.0 と同様な最適化処理でオブジェクトを出力します。
Ver4.0 と比較しオブジェクトが変化しないため、タイミングに依存する処理などに有効です。
本オプションを指定しない場合、Ver4.0 のオブジェクトと比較し、最適化が強化されたオブジェクトを出力します。

■オプション指定方法

コマンドライン　　：*legacy = v4*

■備考

本オプションは CPU 種別が 2600A,2600N,2000A,2000N の時に有効です。
また legacy=v4 オプションを指定した場合、以下のオプションが無効になります。
opt_range, del_vacant_loop, max_unroll, infinite_loop, global_alloc, struct_alloc, const_var_propagate, volatile_loop, scope, strict_ansi, file_inline, file_inline_path, enable_register

3.7.2 cpuexpand=v6

■説明

cpuexpand オプションは、変数の乗除算のコード展開を ANSI 規格から拡張解釈して生成します。

しかし、CPU 種別が 2600A,2600N,2000A,2000N の時、cpuexpand オプションをそのまま使用すると、C/C++コンパイラの V4.0 と V6.0 以降で、オブジェクトが変化してしまう場合があります。

もし、オブジェクトの変化により不都合が起きる場合、cpuexpand=v6 オプションを使用するとオブジェクトが変化しないので不都合が生じません。

■オプション指定方法

コマンドライン　　：*cpuexpand[= v6]*

■影響を受ける式

```
(a)   signed long =    signed int  << 定数
(b)   signed long =  unsigned int  << 定数
(c) unsigned long=    signed int  << 定数
(d) unsigned long=  unsigned int  << 定数
(e)   signed int  = (  signed int << 定数)  /   signed int
(f)   signed int  = (unsigned int << 定数)  /   signed int
(g)   signed int  = (unsigned int << 定数)  / unsigned int
(h) unsigned int = (  signed int << 定数)  /   signed int
(i) unsigned int = (unsigned int << 定数)  /   signed int
(j) unsigned int = (unsigned int << 定数)  / unsigned int
```

■コード例

【(unsigned signed long = unsigned signed int << 定数)の例】

-cpuexpand- legacy=v4

```
MOV.W      @_i1:32,R0
MOV.W      #1024,E0
MULXS.W    E0,ER0
MOV.L      ER0,@_l1:32
```

シフト結果をunsigned longに格納します

-cpuexpand=V6 -legacy=v4

```
MOV.B      @_i1+1:32,R0H
SUB.B      R0L,R0L
SHLL.W     #2,R0
EXTU.L     ER0
MOV.L      ER0,@_l1:32
```

シフト結果を0拡張してからunsigned longに格納します

■備考

本オプションは CPU 種別に 2600A,2600N,2000A,2000N を指定し、かつ **legacy=v4** を指定した場合に有効になります。

3.7.3 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ます。

■オプション指定方法

コマンドライン　　：*enable_register*

■使用例

```
int g_i1;
void func()
{
register long Reg_l1 = 999;
long l2 = 126;
long l3 = 248;

 switch(g_i1){
 case 2:
    Reg_l1++;
    break;
 case 3:
     l2 += 5;
    break;
 case 4:
     l2 += 7;
    break;
 case 9:
    l3 -= 11;
    break;
 case 10:
    l3 -= 19;
    break;
 }
   printf("%d,%d,%d¥n",Reg_l1,l2,l3); // ' Reg_l1'の値は ER1 経由で printf に渡されるので
                                      // ' Reg_l1'に ER1 が割り付けられると都合が良い。
}
```

■コード例

-enable_register指定なし

```
_func:
    STM.L      (ER4-ER6),@-SP
    SUB.W      #8:16,R7
    MOV.L      #H'000003E7,ER5
    SUB.L      ER6,ER6
    MOV.B      #H'7E:8,R6L
    SUB.L      ER4,ER4
    MOV.B      #H'F8:8,R4L
    MOV.W      @_g_i1:16,R0
    MOV.W      R0,R1
    MOV.B      R0H,R0H
    BNE        L26:8
    CMP.B      #2:8,R1L
    BEQ        L27:8
    CMP.B      #3:8,R1L
    BEQ        L28:8
    CMP.B      #4:8,R1L
    BEQ        L29:8
    CMP.B      #9:8,R1L
    BEQ        L30:8
    CMP.B      #H'0A:8,R1L
    BNE        L26:8
    MOV.B      #H'E5:8,R4L
    BRA        L26:8
L30:
    MOV.B      #H'ED:8,R4L
    BRA        L26:8
L29:
    MOV.B      #H'85:8,R6L
    BRA        L26:8
L28:
    MOV.B      #H'83:8,R6L
    BRA        L26:8
L27:
    MOV.B      #H'E8:8,R5L
L26:
    MOV.W      #LWORD L45:16,R0
    MOV.L      ER6,@SP
    MOV.L      ER4,@(4:16,SP)
    MOV.L      ER5,ER1
    JSR        @_printf:16
    ADD.W      #8:16,R7
    LDM.L      @SP+,(ER4-ER6)
    RTS
```

変数Reg_l1の優先度が上がったのでER1を割り付けている

-enable_register

```
_func:
    STM.L      (ER4-ER6),@-SP
    SUB.W      #8:16,R7
    MOV.L      #H'000003E7,ER1
    SUB.L      ER4,ER4
    MOV.B      #H'7E:8,R4L
    SUB.L      ER6,ER6
    MOV.B      #H'F8:8,R6L
    MOV.W      @_g_i1:16,R0
    MOV.W      R0,R5
    MOV.B      R0H,R0H
    BNE        L26:8
    CMP.B      #2:8,R5L
    BEQ        L27:8
    CMP.B      #3:8,R5L
    BEQ        L28:8
    CMP.B      #4:8,R5L
    BEQ        L29:8
    CMP.B      #9:8,R5L
    BEQ        L30:8
    CMP.B      #H'0A:8,R5L
    BNE        L26:8
    MOV.B      #H'E5:8,R6L
    BRA        L26:8
L30:
    MOV.B      #H'ED:8,R6L
    BRA        L26:8
L29:
    MOV.B      #H'85:8,R4L
    BRA        L26:8
L28
    MOV.B      #H'83:8,R4L
    BRA        L26:8
L27:
    MOV.B      #H'E8:8,R1L
L26:
    MOV.W      #LWORD L45:16,R0
    MOV.L      ER4,@SP
    MOV.L      ER6,@(4:16,SP)

    JSR        @_printf:16
    ADD.W      #8:16,R7
    LDM.L      @SP+,(ER4-ER6)
    RTS
```

■備考

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。
本オプションは、CPU 種別に H8SX および H8S を指定した場合に有効となります。

3.7.4 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

```
_main:
        MOV.L      #H'0A:8,@_io:16
        RTS
        .SECTION    $ADDRESS$B100,DATA,LOCATE=H'100
_io:
        .RES.L     1
        .END
```

■注意事項

本拡張機能は、CPU 種別に H8SX および H8S を指定した場合に有効となります。

（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 abs8/abs16
#pragma global_register
```

3.7.5 ファイル間インライン展開

■説明

C/C++コンパイラはファイル単位にコンパイルするため、-speed=inline や#pragma inline, inline キーワードでインライン展開対象となっている関数があっても、ファイル間をまたがってインライン展開対象関数を呼び出している場合はインライン展開されません。

ファイル間インライン展開オプションはファイル間をまたがって、インライン展開対象関数を呼び出している時も、インライン展開を可能にします。

また、インライン展開をしたい関数を含むファイルが別なディレクトリにある場合は、ファイル間インライン展開対象にしたいファイルが存在するディレクトリを指定することにより、インライン展開することが可能です。

■オプション指定方法

[ファイル間インライン展開]
ダイアログメニュー：**コンパイラ**タブ**カテゴリ:[最適化][インライン展開][インライン展開ファイル]**
コマンドライン　　：*FILe_inline=<ファイル名>[,...]*

[ファイル間インライン展開ディレクトリ指定]
ダイアログメニュー：**コンパイラ**タブ**カテゴリ:[ソース][オプション項目 :][ファイル間インライン展開ディレクトリ]**
コマンドライン　　：*file_inline_path=<パス名>[,...]*

* **対象ファイルの検索は、[ファイル間インライン展開ディレクトリ指定]で指定したディレクトリ、カレントディレクトリの順序で行います。**

■使用例

以下のように__inline キーワード指定された関数(func)を、別のファイルから呼び出している時、呼び出している関数側(test_1.c)のコンパイル時にファイル間インライン展開オプション指定します。

（C/C++プログラム）

```
ch38 –cpu=h8sxa –file_inline=test_2.c test_1.c
ch38 –cpu=h8sxa test_2.c
```

```
[test_1.c]
void main(void);
void func(void);
int si1,si2;
void main(void)
{
    func();
}
```

```
[test_2.c]
extern int si1,si2;
void func(void);
__inline void func(void)
{
    si1 = 10;
    si2 = 20;
}
```

■コード例

ファイル間インライン指定がある場合、test_2.c のコードが呼び出し側に展開されます。

指定なし

```
[test_1.c]
_main:
    JMP        @_func:24
```

```
[test_2.c]
_func:
    MOV.W      #H'A:4,@_si1:32
    MOV.W      #H'14:8,@_si2:32
    RTS
```

指定あり

```
[test_1.c]
_main:
    MOV.W      #H'A:4,@_si1:32
    MOV.W      #H'14:8,@_si2:32
    RTS
```

```
[test_2.c]
_func:
    MOV.W      #H'A:4,@_si1:32
    MOV.W      #H'14:8,@_si2:32
    RTS
```

■備考

本オプションは CPU 種別に H8SX および H8S（legacy=v4 オプション指定なし）を指定した場合に有効となります。

（1） 本オプションを指定した場合、<ファイル名>で指定されたファイルの関数のうち、#pragma inline、inlineキーワードが指定された関数についてのみインライン展開対象とします。同時に-speed=inlineオプションを指定した場合、インライン可能な関数をすべて展開します。

（2） 本オプションで指定された複数のファイルで同じ名前の大域変数が定義されていた場合、動作は保障しません。（任意に選んだ１つの関数定義を用いてインライン展開します）

（3） <ファイル名>で指定するファイル名の拡張子を省略することはできません。

（4） <ファイル名>にワイルドカード（*,?）を指定することはできません。

（5） #pragma asm～endasm、#pragma inline_asmおよび__asmの記述があるファイルは展開しません。

3.7.6 最適化範囲分割機能

■説明

最適化範囲分割機能オプションは、サイズの大きい関数について、最適化範囲を複数に分割してコンパイルします。

最適化範囲分割機能オプションを指定しない場合は、最適化範囲を分割せずにコンパイルします。最適化範囲が広がることによりコンパイル時間は長くなりますが、一般的にはオブジェクト性能が向上します。ただし、レジスタ数が不足するとオブジェクト性能が低下する場合があります。本オプションは、プログラムによって効果が変わるため、性能チューニング時に指定することをお勧めします。

■オプション指定方法

ダイアログメニュー：**なし**
コマンドライン　　：*SCOpe*
　　：*NOSCope*

■使用例

変数宣言が 1000 個あり、それぞれの変数への代入がある C プログラムでの ROM サイズ例を下記に示します。
noscope オプション指定時に 6 バイトサイズが減少しています。

なお、messege オプション(HEW の場合、**コンパイラ**タブ**カテゴリ:[ソース][オプション項目:][インフォメーションメッセージ][インフォメーションレベルメッセージの表示])**を指定し、かつ scope オプションにより最適化範囲が分割された時は、以下のメッセージを出力します。

（C/C++プログラム）
cpu=h8sxa
 scope 指定　 8010 バイト
 noscope 指定 8004 バイト

（出力メッセージ）
C0101 (I) Optimizing range divided in function "関数名"

■備考

本オプションは CPU 種別に H8SX および H8S（legacy=v4 オプション指定なし）を指定した場合に有効となります。

3.8 H8SX の特長

3.8.1 アドレス空間

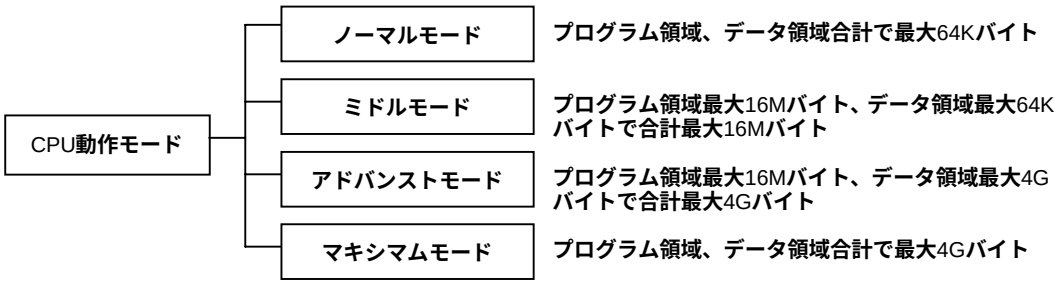
■説明

H8/300,H8/300H,H8S/2000,H8S/2600 は最大で 2 つの CPU モードでしたが、H8SX では以下の 4 つの CPU 動作モードがあります。

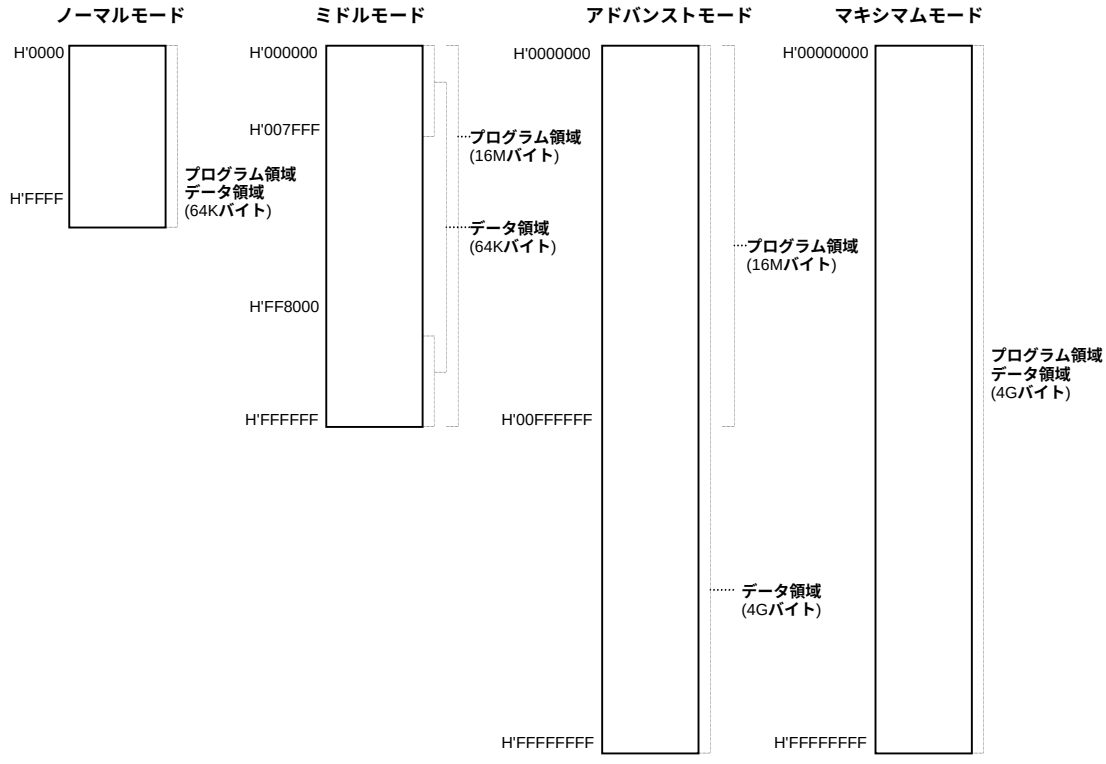
各モードは LSI のモード端子などによって選択されます。

コンパイル時は CPU 種別／動作モードオプションで選択します。

CPU モードはデバイスにより使用できるモードや領域に違いがあるので、該当のハードウェアマニュアルを参照し設定をしてください。



■各モードのアドレス空間



3.8.2 8bit 絶対アドレス空間の任意設定

■説明

8bit 絶対アドレス空間にデータを配置しアクセスすると、低サイズかつ高速なオブジェクトにすることができます。

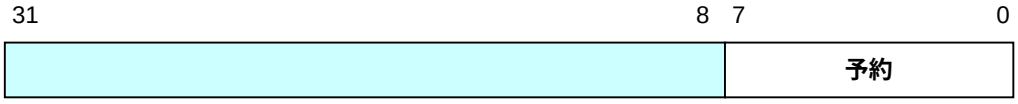
H8SX ではこの 8bit 絶対アドレス空間をユーザの指定する任意のアドレスに変更することができます。

従来の H8 ファミリでは、8bit 絶対アドレス空間は H'FFFF00～H'FFFFFF 番地固定で、内蔵 I/O 空間と重複していました。

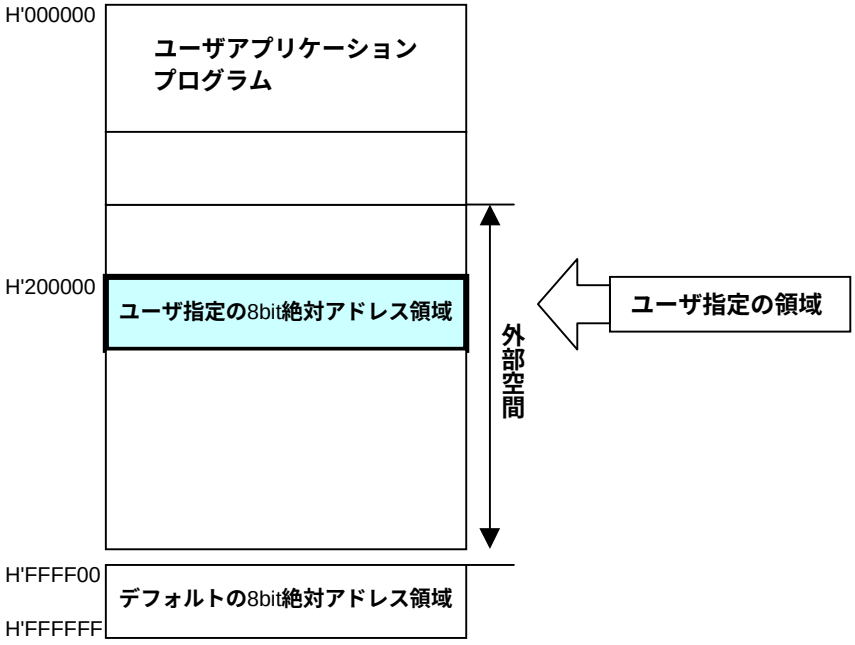
H8SX では SBR レジスタ（ショートアドレスベースレジスタ）で設定した任意アドレスからの 256 バイトのエリアを 8bit 絶対アドレス空間として設定できます。

■レジスタの構成

SBR レジスタ（ショートアドレスベースレジスタ）は 32 ビットのレジスタで上位 24 ビットが有効、下位 8 ビットは予約でリードすると 0 が読み出されます。

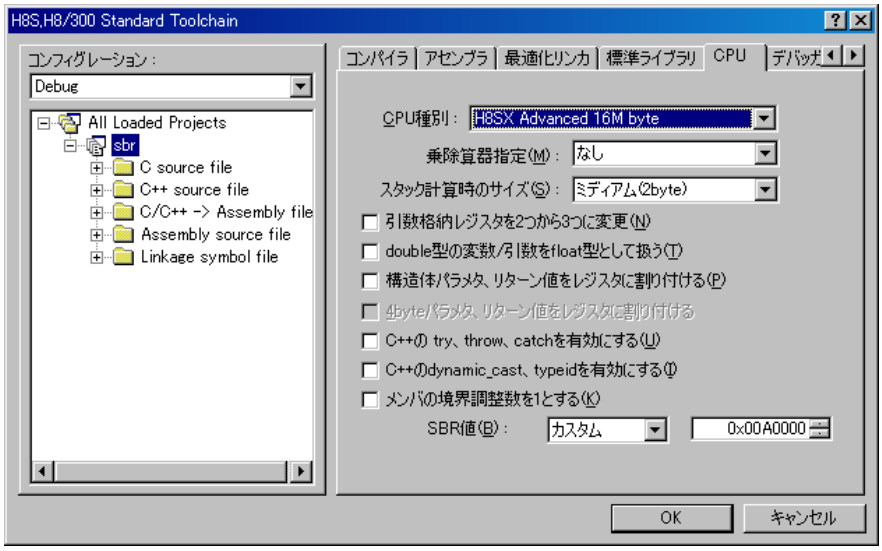


■8bit 絶対アドレス空間のイメージ



■オプション指定方法

ダイアログメニュー：**CPU タブ**の **SBR 値**



コマンドライン　　：*sbr = <アドレス>*

■サンプルプログラム

SBR レジスタ（ショートアドレスベースレジスタ）は C/C++言語から直接アクセスすることはできません。

したがって、アセンブラ命令で記述する必要があります。

しかし、以下のようにコンパイラの拡張機能__asm を用いると C/C++言語にアセンブラ命令を記述することができます。

古いバージョンのコンパイラでもアセンブラ命令を#pragma asm で記述することができましたが、コンパイル後にアセンブラソースコードにする必要がありました。

しかし、__asm はコンパイラが直接解析しオブジェクトに出力するので、大変便利な上にデバッグ時シンボルが参照できないなどの不都合もなくなります。

__asm の詳細は H8S、H8/300 **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「**10.2.1(3)　**その他の拡張機能」**の**「**__asm**」**をご参照ください。

（C/C++プログラム）

```
__abs8 unsigned char a,b;
void main(void)
{
__asm{
    mov.l #0xa0000,er0
    ldc.l er0,sbr
}
    a = 7;
    b = -1;

}
```

←__abs8で8bit絶対アドレスの指定

←0xa0000番地をSBRに設定

←0xa0000番地の8bit絶対アドレス領域にアクセス

■セクションの配置

__abs8 で 8bit 絶対アドレス空間を使用すると宣言しているので、$ABS8B というセクションが出力されます。最適化リンカで、$ABS8B セクションを上記プログラム例の場合、0xa0000 番地に配置します。

■C/C++プログラム例

HEW の場合、resetprg.c の下記太字部分のコメントを外し有効にします。
HEW を使用しない場合、イニシャルルーチンに同様の記述を追加してください。

```
__entry(vect=0) void PowerON_Reset(void)
{
// Remove the comment when you make the initial setting of SBR/VBR for H8SX
    __asm{
        mov.l  #0xa0000,er0
        ldc.l  er0,sbr
        mov.l  #0x00000000,er0
        ldc.l  er0,vbr
    }

     set_imask_ccr(1);
    _INITSCT();
       :
```

（8bit 絶対アドレス空間**未使用**）

```
unsigned char c1,c2;
void main(void)
{
    c1 = 7;
    c2 = -1;
}
```

（8bit 絶対アドレス空間**使用**）

```
__abs8 unsigned char c1,c2;
void main(void)
{
    c1 = 7;
    c2 = -1;
}
```

■アセンブリ展開コード例

H8SX アドバンストモード 16M の例です。

（8bit 絶対アドレス空間**未使用**）

```
_main:
    MOV.B      #7:4,@_c1:32
    MOV.B      #255:8,@_c2:32
    RTS
```

（8bit 絶対アドレス空間**使用**）

0xa0000 – 0xa00FFをアクセス

```
_main:
    MOV.B      #7:8,R0L
    MOV.B      R0L,@_c1:8
    MOV.B      #255:8,R0L
    MOV.B      R0L,@_c2:8
    RTS
```

オブジェクトサイズ表 [バイト]

CPU **種別**	H8SX		
	MAX	ADV	NML
改善前	16	16	12
改善後	10	10	10

実行処理速度表 [サイクル]

CPU **種別**	H8SX		
	MAX	ADV	NML
改善前	12	12	11
改善後	11	11	11

3.8.3 ベクタテーブルアドレス切り替え

■説明

H8SX ファミリでは例外処理ベクタテーブルを任意のアドレスに配置する機能があります。H8/300,H8/300H,H8S ファミリでは例外処理ベクタテーブルは 0 番地からの配置に固定されていますが、H8SX ファミリではベクタベースレジスタ（VBR）を設定することによって例外処理ベクタテーブルの配置アドレスを変更することが可能です。

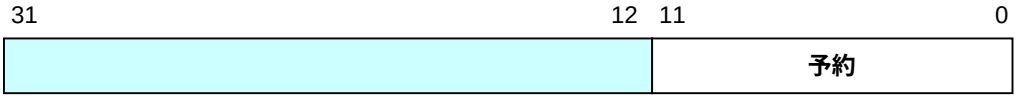
■ベクタベースレジスタのメリット

ベクタベースレジスタ（VBR）を使い任意のアドレスに例外処理ベクタを配置できるので、ROM レス品であっても、高速な内蔵 RAM にベクタテーブルを作成できます。

これにより割り込み処理の応答性を高速化できます。

■レジスタの構成

ベクタベースレジスタ（VBR）は 32 ビットのレジスタで上位 20 ビットが有効、下位 12 ビットは予約でリードすると 0 が読み出されます。



■コンパイラ Ver6.1 でのベクタベースレジスタへの設定

組み込み関数 set_vbr を使用することにより C/C++言語から設定することができます。詳細は**「3.2.3 ベクタベースレジスタの設定」**をご参照ください。

■コンパイラ Ver6.0 でのサンプルプログラム

コンパイラのバージョンが 6.0 の場合、ベクタベースレジスタ（VBR）は C/C++言語から直接アクセスすることはできません。

したがって、アセンブラ命令で記述する必要があります。

しかし、以下のようにコンパイラの拡張機能__asm を用いると C/C++言語にアセンブラ命令を記述することができます。

古いバージョンのコンパイラでもアセンブラ命令を#pragma asm で記述することができましたが、コンパイル後にアセンブラソースコードにする必要がありました。

しかし、__asm はコンパイラが直接解析しオブジェクトに出力するので、大変便利な上にデバッグ時シンボルが参照できないなどの不都合もなくなります。

__asm の詳細は **H8S、H8/300 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「**10.2.1(3) **その他の拡張機能」**の**「__asm」**をご参照ください。

ベクタベースレジスタ（VBR）の切り替えは割り込み禁止状態で行う必要があります。割り込み禁止にしない状態でベクタベースレジスタ（VBR）の切り替え中に割り込みが発生した場合には正常な動作が行われません。

（C/C++プログラム）

```
void main(void)
{
__asm{
    orc.b #0x80,ccr
    mov.l #0xffa000,er0
    ldc.l er0,vbr
    andc.b #0x7F,ccr
    RTS
}
}
```

←割り込みマスクビットをセット

←割り込みマスクビットをクリア

4. HEW

本章では HEW1.2、2.0 以降使用時の、C/C++コンパイラ、アセンブラ、モジュール間最適化ツール、オブジェクトコンバータ、およびライブラリアンのオプション画面とコマンドオプションとの対応を説明いたします。

それぞれのオプションについての詳細は各ツールのユーザーズマニュアルを参照してください。（モジュール間最適化ツールのオプション説明は、H8S、H8/300 **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。）

HEW1.2 各ツールのオプション画面は次のように選択します。

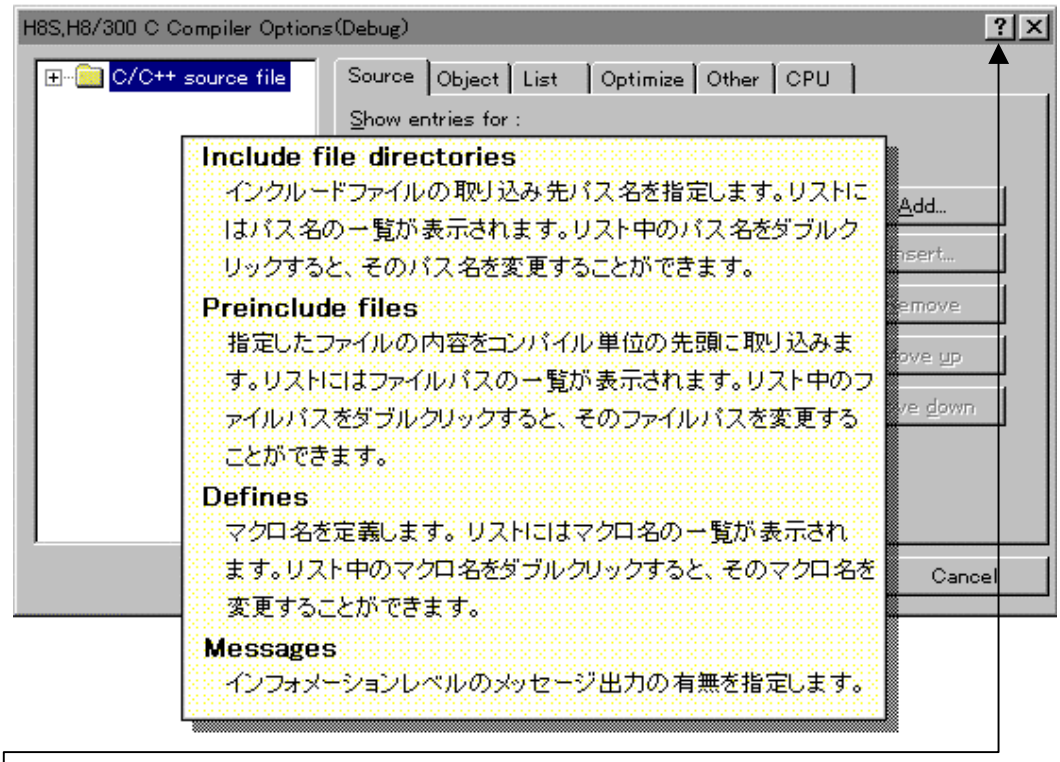
ツール名	選択方法
C/C++**コンパイラ**	**[Options->H8S,H8/300 C/C++ Compiler…]**
クロスアセンブラ	**[Options->H8S,H8/300 Assembler…]**
モジュール間最適化ツール	**[Options->H8S,H8/300 IM Optlinker…]**
オブジェクトコンバータ	**[Options->H Series Stype Converter…]**
ライブラリアン	**[Options->H Series Librarian…]**

【注】 Options **メニューに該当ツールがない場合は、[Options->Build Phases…]で追加します。**

HEW2.0 以降 ではオプションメニューから H8S,H8/300 Standard Toolchain...を選択します。
HEW4.0 以降 ではビルドメニューから H8S,H8/300 Standard Toolchain...を選択します。

ツール名	選択方法
C/C++**コンパイラ**	**[オプション->H8S,H8/300 Standard Toolchain…->コンパイラタブ]**
クロスアセンブラ	**[オプション->H8S,H8/300 Standard Toolchain…->アセンブラタブ]**
最適化リンケージエディタ	**[オプション->H8S,H8/300 Standard Toolchain…->最適化リンカタブ]**
標準ライブラリ構築ツール	**[オプション->H8S,H8/300 Standard Toolchain…->標準ライブラリタブ]**
CPU **オプション**	**[オプション->H8S,H8/300 Standard Toolchain…->CPU タブ]**

また、それぞれのオプション画面においてヘルプを参照することができます。



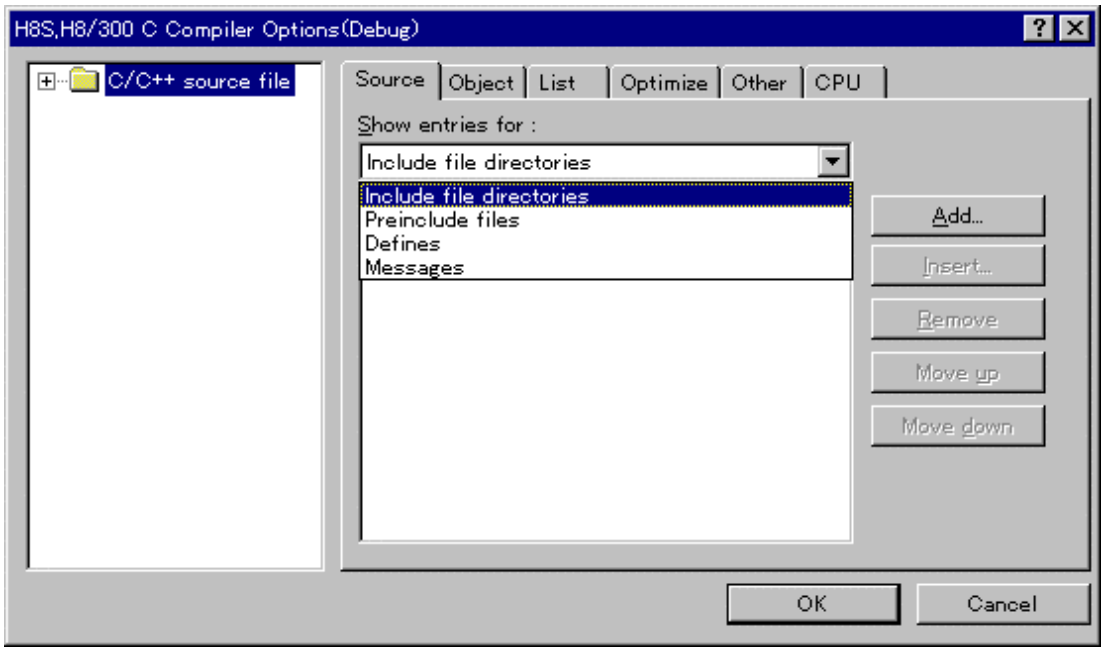
右上の[?]をクリックし、その後参照したい箇所をクリックすると、上記のように説明文が出てきます。
簡単に参照できるヘルプなので併せてお使いください。

4.1 HEW1.2 のオプション指定方法

HEW2.0 以降のオプション指定方法については、「4.2 HEW2.0 以降のオプション指定方法」を参照ください。

4.1.1 C/C++コンパイラのオプション

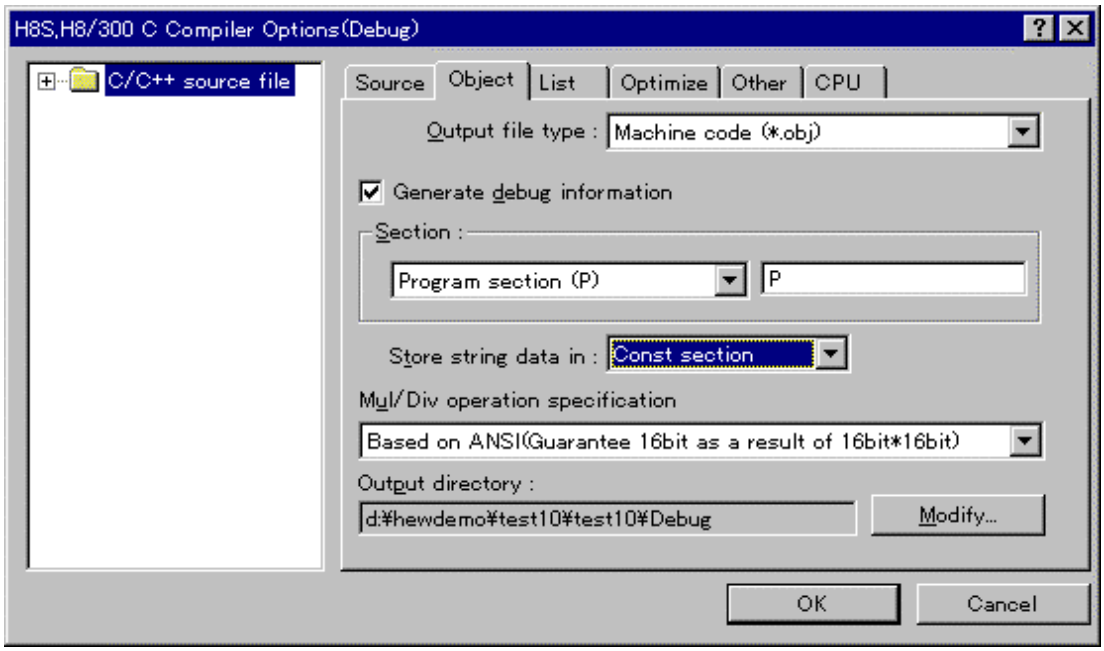
(1) Source タブ



Show entries for:

ダイアログメニュー	コマンドオプション	機能
Include file directories	*include*	インクルードファイルの取り込み先パス名指定
Preinclude files	*preinclude*	指定したファイルをコンパイル単位の先頭にインクルードファイル指定
Defines	*define*	マクロ名の定義
Messages	*message*	インフォメーションメッセージの出力

(2) Object タブ



Output file type:

ダイアログメニュー	コマンドオプション	機能
Machine code (*.obj)	*code=machinecode*	**機械語プログラムを出力**
Assembly source code (*.src)	*code=asmcode*	**アセンブリプログラムを出力**
Preprocessed source file (*.p/*.pp)	*preprocessor*	**プリプロセッサ展開後ソースプログラム出力**

Generate debug information

チェックボックス	コマンドオプション	機能
☑	*debug*	**デバッグ情報出力**
☐	*nodebug*	**デバッグ情報出力抑止**

Section:

ダイアログメニュー	コマンドオプション	機能
-	*section*	**デフォルトのセクション名を変更**

Store string data in:

ダイアログメニュー	コマンドオプション	機能
Const section	*string=const*	**文字列を定数領域へ出力**
Data section	*string=data*	**文字列を初期化データ領域へ出力**

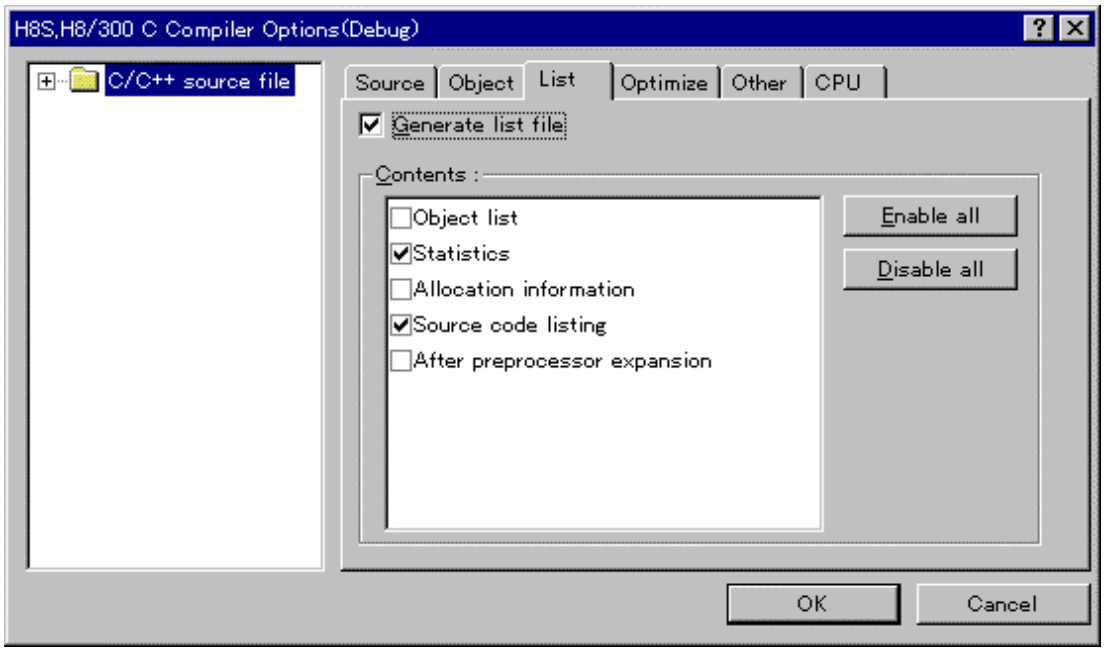
Mul/Div operation specification

ダイアログメニュー	コマンドオプション	機能
Based on ANSI(Guarantee 16bit as a result of 16bit*16bit)	*nocpuexpand*	**乗除算を ANSI C 言語仕様準拠でコード展開**
Non ANSI(Guarantee 32bit as a result of 16bit*16bit)	*cpuexpand*	**乗除算を CPU 命令仕様にあわせてコードを展開**

Output directory

ダイアログメニュー	コマンドオプション	機能
-	*object*	オブジェクトファイルの出力先ディレクトリを指定

(3) List タブ



Generate list file

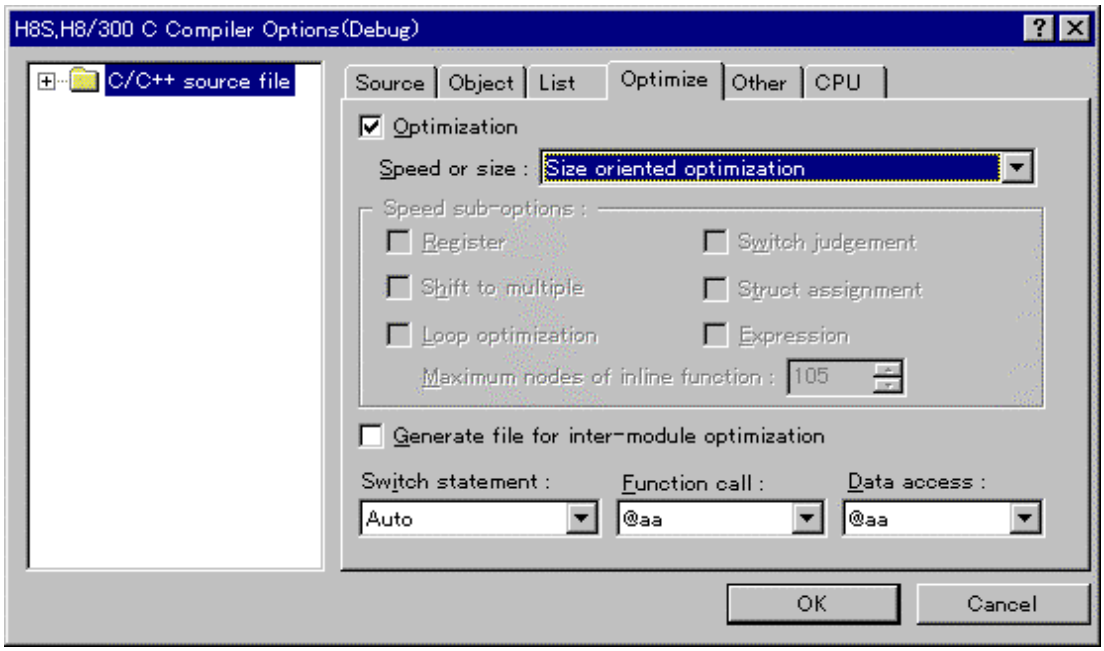
チェックボックス	コマンドオプション	機能
☑	*list*	オブジェクトリストファイル出力
☐	*nolist*	オブジェクトリストファイル出力抑止

Contents:オブジェクトリストファイルへの出力内容を選択します。

ダイアログメニュー	コマンドオプション	機能
Object list	*show=object*	オブジェクトリストの出力
Statictics	*show=statictics*	統計情報の出力
Allocation information	*show=allocation*	シンボル割り付けリストの出力
Source code listing	*show=source*	マクロ展開後リストの出力
After preprocessor expansion	*show=expansion*	ソースリストの出力

[Enable all]ボタンを押すとすべてが出力対象となり、[Disable all]ボタンを押すとすべてが無効となります。このとき出力されたオブジェクトリストファイルには何も出力されません。

(4) Optimize **タブ**



Optimization

チェックボックス	コマンドオプション	機能
☑	*optimize=1*	**最適化指定**
☐	*optimize=0*	**最適化指定なし**

Speed or size:最適化の方式を選択します。

ダイアログメニュー		コマンドオプション	機能
Size oriented optimization		-	**サイズ優先の最適化**
Speed oriented optimization		*speed*	**スピード優先の最適化**
Speed sub-options	**Register**	*speed=register*	**レジスタ退避／回復** push/pop **展開**
	Swtich judgement	*speed=switch*	switch **文の高速化**
	Shift to multiple	*speed=shift*	**シフト演算の高速化**
	Struct assignment	*speed=struct*	**構造体代入式の高速化**
	Loop optimization	*speed=loop*	**ループ文をループ展開**
	Expression	*speed=expression*	**四則演算、比較、代入式の高速化**
	Maximum nodes of inline function	*speed=inline [=<**数値**>]*	**自動インライン展開**

Generate file for inter-module optimization

チェックボックス	コマンドオプション	機能
☑	*goptimize*	**モジュール間最適化ツール用付加情報ファイル出力**
☐	-	**モジュール間最適化ツール用付加情報ファイル出力なし**

Switch statement:switch 文の展開方式を選択します。

ダイアログメニュー	コマンドオプション	機能
Auto	*case=auto*	speed **オプション指定有無で判定**
If then	*case=ifthen*	if_then **方式で展開**
Table	*case=table*	**テーブルジャンプ方式で展開**

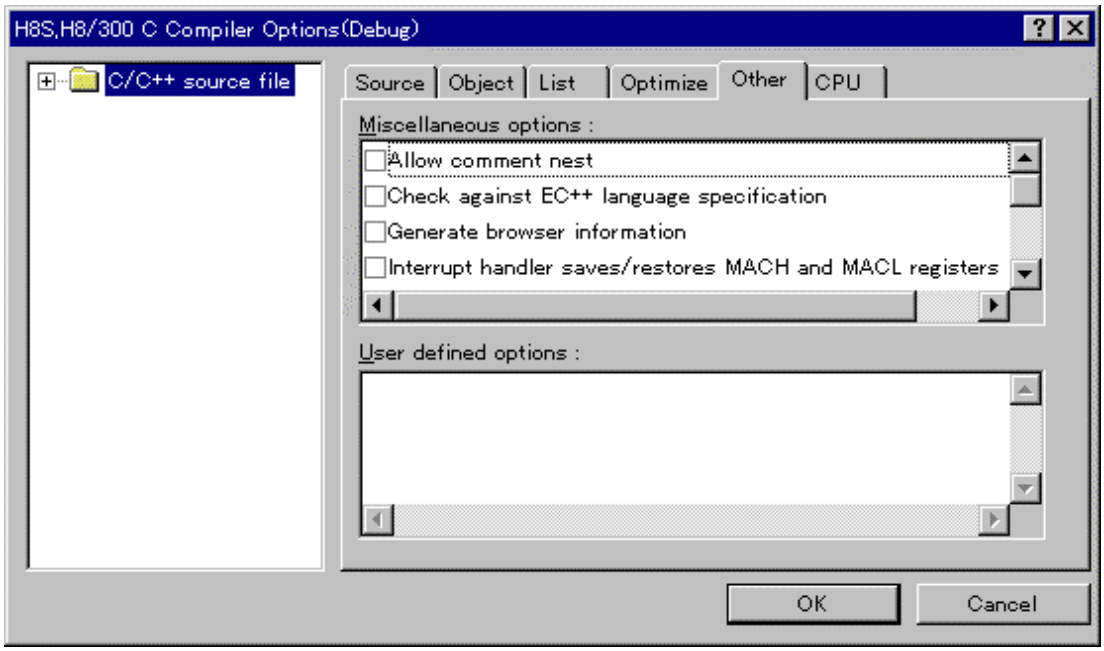
Function call:では関数の呼び出し形式を選択します。

ダイアログメニュー	コマンドオプション	機能
@aa	-	**通常の関数呼び出し**
@@aa:8	*indirect*	**メモリ間接形式の関数呼び出し**

Data access:データのアクセス方法を選択します。

ダイアログメニュー	コマンドオプション	機能
@aa	-	**通常のデータアクセス**
@aa:8	*abs8*	8 **ビット絶対アドレスアクセス**
@aa:16	*abs16*	16 **ビット絶対アドレスアクセス**

(5) Other **タブ**

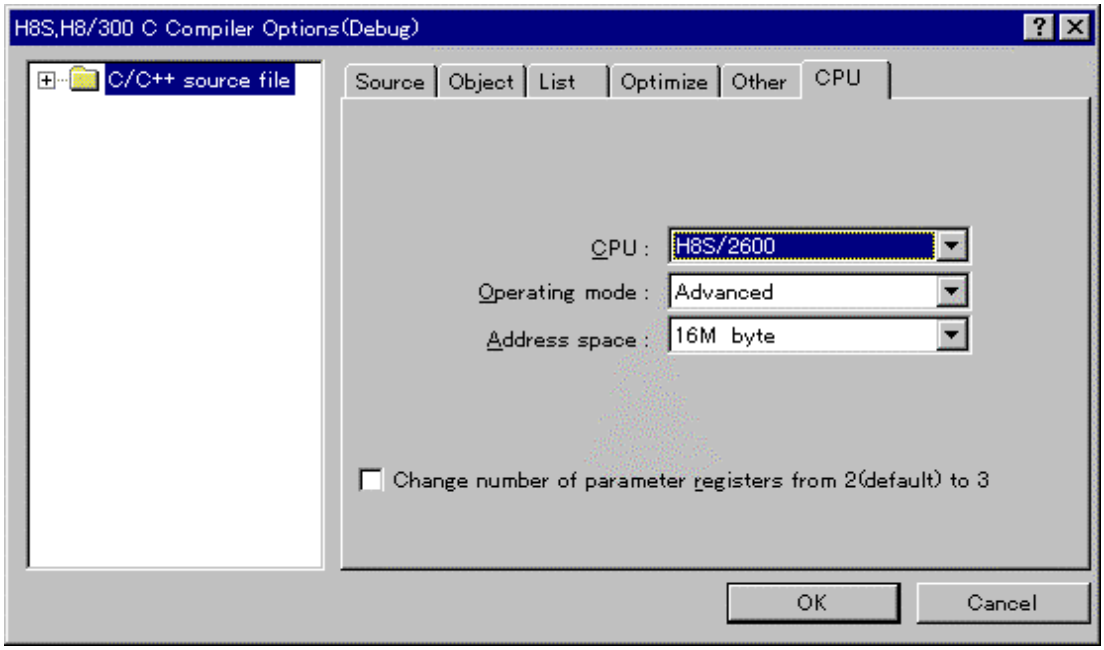


Miscellaneous options:

ダイアログメニュー	コマンドオプション	機能
Allow comment nest	*comment*	**コメントのネストを許可**
Check against EC++ language specification	*ecpp*	EC++**言語仕様に基づいてシンタックスチェック**
Generate browser information	*browser*	**ブラウザ情報の出力**
Interrupt handler saves/restores MACH and MACL registers if used	*macsave*	MAC **レジスタ保証**
Pack struct ,union and class	*pack=1 | 2*	**境界調整数の指定**
Avoid optimizing external symbols treating them as volatile	*volatile*	**外部変数の最適化** **外部変数の最適化抑止**
Treat enum as char if it is in the range of char	*byteenum*	**列挙型のデータを** char **扱い**
Increase a register for register variable	*regexpansion | noregexpansion*	**変数割り付けレジスタ数** 2 **変数割り付けレジスタ数** 3
Put common subexpression on a register temporarily	*cmncode*	**共通式削除の最適化強化**
Use EEPMOV in block copy	*eepmov*	**構造体の代入を** eepmov **命令で展開**

User defined options:ではコマンドオプションを指定することができます。

(6) CPU タブ



CPU では、CPU 種別を指定します。

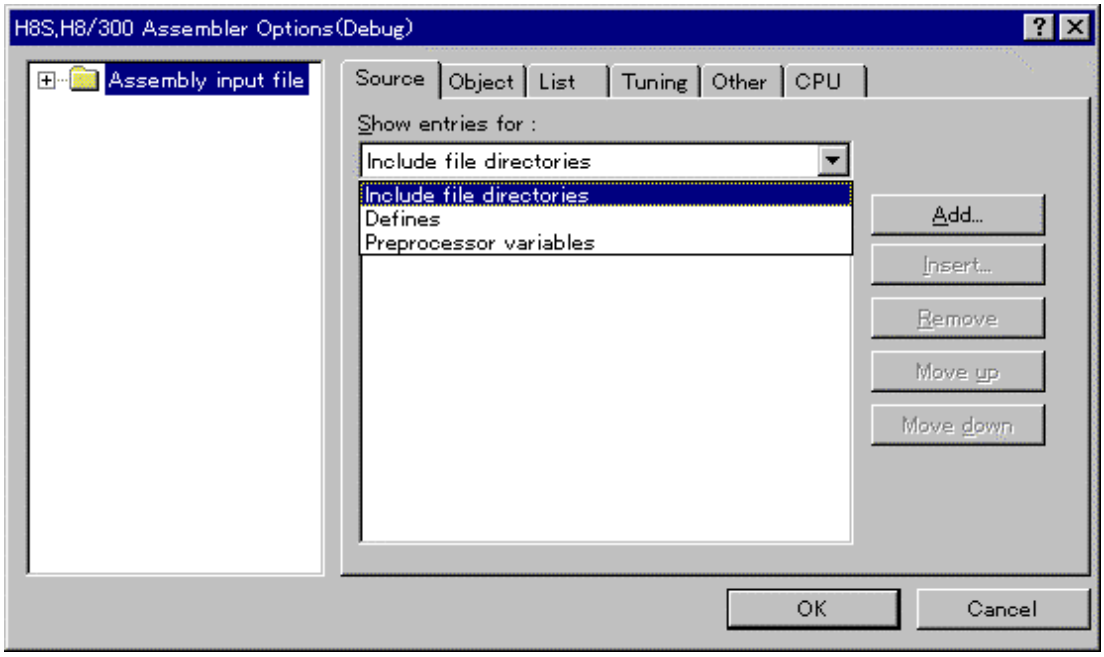
CPU	Operating mode	Address space:	**内容**
Environmento variable	-	-	**環境変数** H38CPU **依存**
H8S/2600	**Normal**		*cpu=2600N*
	Advanced	**1M byte**	*cpu=2600A:20*
		16M byte	*cpu=2600A:24*
		256M byte	*cpu=2600A:28*
		4G byte	*cpu=2600A:32*
H8S/2000	**Normal**		*cpu=2000N*
	Advanced	**1M byte**	*cpu=2000A:20*
		16M byte	*cpu=2000A:24*
		256M byte	*cpu=2000A:28*
		4G byte	*cpu=2000A:32*
H8/300H	**Normal**		*cpu=300HN*
	Advanced	**1M byte**	*cpu=300HA:20*
		16M byte	*cpu=300HA:24*
H8/300	-	-	*cpu=300*
H8/300L	-	-	*cpu=300*

Change number of parameter registers from 2(default) to 3

チェックボックス	**コマンドライン**	**機能**
☑	*regparam=3*	**引数渡し用レジスタ数** 3 **つ**
☐	*regparam=2*	**引数渡し用レジスタ数** 2 **つ**

4.1.2 アセンブラのオプション

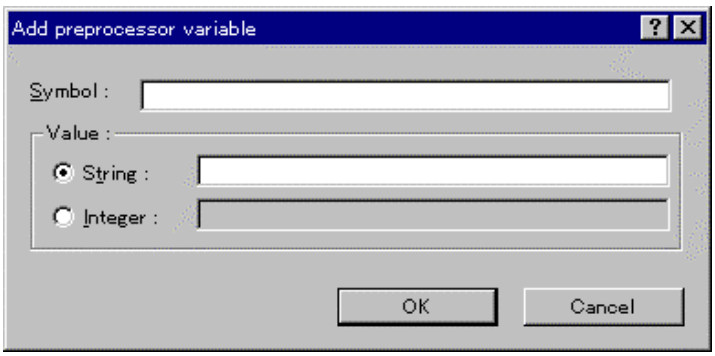
(1) Source タブ



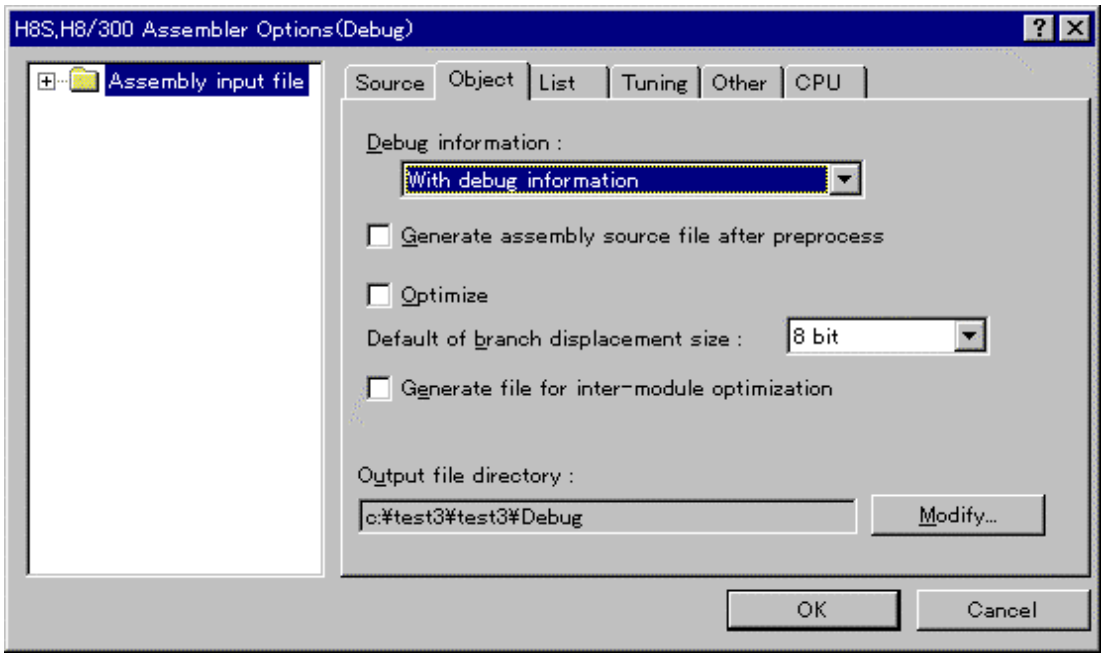
Show entries for:

ダアログメニュー	コマンドオプション	機能
Include file directories	*include*	**インクルードファイルのディレクトリ指定**
Defines	*define*	**文字列の置き換えの定義**
Preprocessor variable*	*assigna*	**整数型プリプロセッサ変数の定義**
	assignc	**文字型プリプロセッサ変数の定義**

【注】 * **次のダイアログボックスで指定します。**



(2) Object タブ



Debug information:

ダイアログメニュー	コマンドオプション	機能
Default	-	.DEBUG 制御命令のみ有効
With debug information	*debug*	デバッグ情報出力指定
Without debug information	*nodebug*	デバッグ情報出力抑止

Generate assembly source file after preprocess

チェックボックス	コマンドオプション	機能
☑	*expand*	プリプロセッサの展開結果の出力
☐	-	プリプロセッサの展開結果の出力なし

Optimize

チェックボックス	コマンドオプション	機能
☑	*optimize*	最適化の指定
☐	*nooptimize*	最適化の指定なし

Default of branch displacement size:

ダイアログメニュー	コマンドオプション	機能
8bit	*br_relative=8*	分岐命令のディスプレースメントが前方参照の場合のディスプレースメントサイズ 8 ビット
16bit	*br_relative=16*	分岐命令のディスプレースメントが前方参照の場合のディスプレースメントサイズ 16 ビット

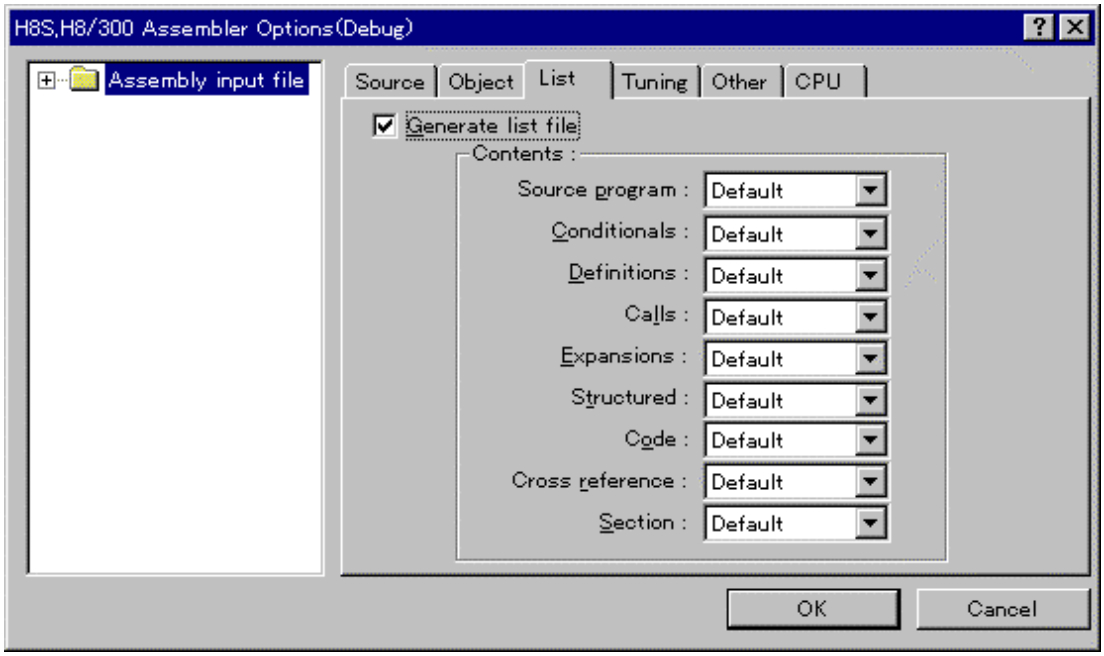
Generate file for inter-module optimization

チェックボックス	コマンドオプション	機能
☑	*goptimize*	モジュール間最適化情報の出力
☐		モジュール間最適化情報の出力なし

Output directory

ダイアログメニュー	コマンドオプション	機能
-	*object[=<ファイル名>]*	オブジェクトの出力ディレクトリ指定

(3) List タブ



Generate list file

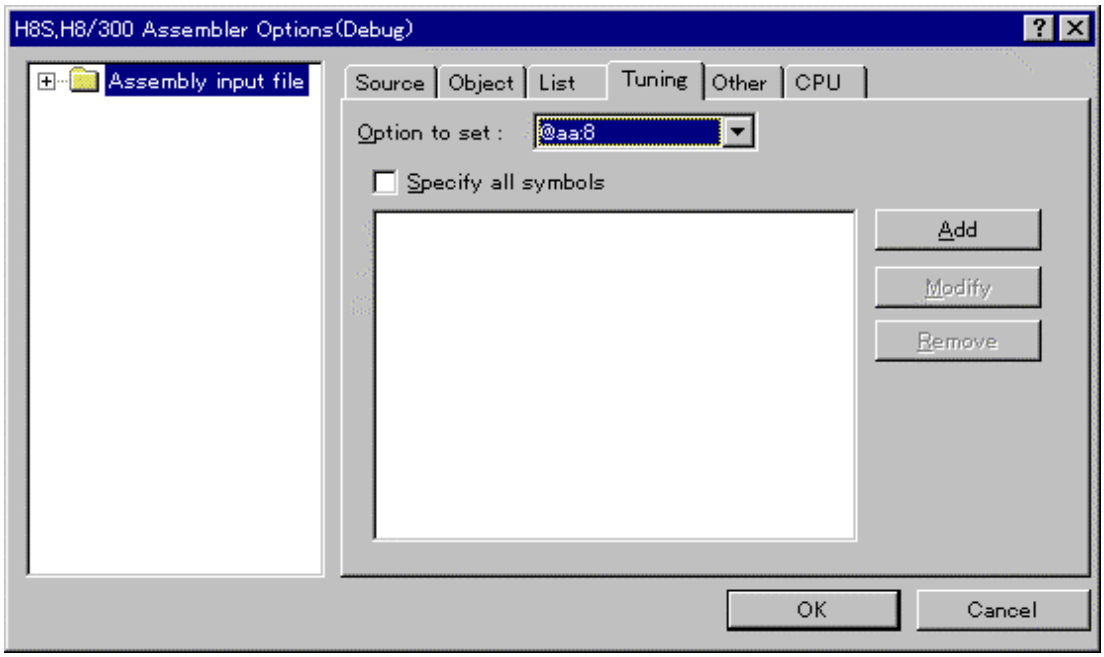
チェックボックス	コマンドオプション	機能
☑	*list*	アセンブルリストの出力
☐	*nolist*	アセンブルリストの出力なし

Contents: リストファイルに出力する内容を指定します。

ダイアログメニュー	コマンドオプション	機能
Source program	*source*	アセンブリリストのソースプログラムリスト
Conditionals	*show=conditionals*	.AIF、.AIFDEF の不成立部分
Definitions	*show=definitions*	マクロ定義部分、.AREPEAT,.AWHILE 定義部分、.INCLUDE,.ASSIGNA,.ASSIGNC 制御文
Calls	*show=calls*	マクロコール文、.AIF,.AIFDEF,.AENDI 制御文
Expansions	*show=expansions*	マクロ展開部分、.AREPEAT,.AWHILE 展開部分
Structured	*show=structured*	構造化アセンブリ展開部分
Code	*show=code*	制御命令のオブジェクトコード表示がソースステートメントの行数を超える部分
Cross reference	*cross_refernce*	クロスリファレンスリスト
Section	*section*	セクションインフォメーションリスト

【注】 各出力内容で Default を選択すると、ソースリスト中の制御命令の指定を有効にします。

(4) Tuning タブ



Option to set:

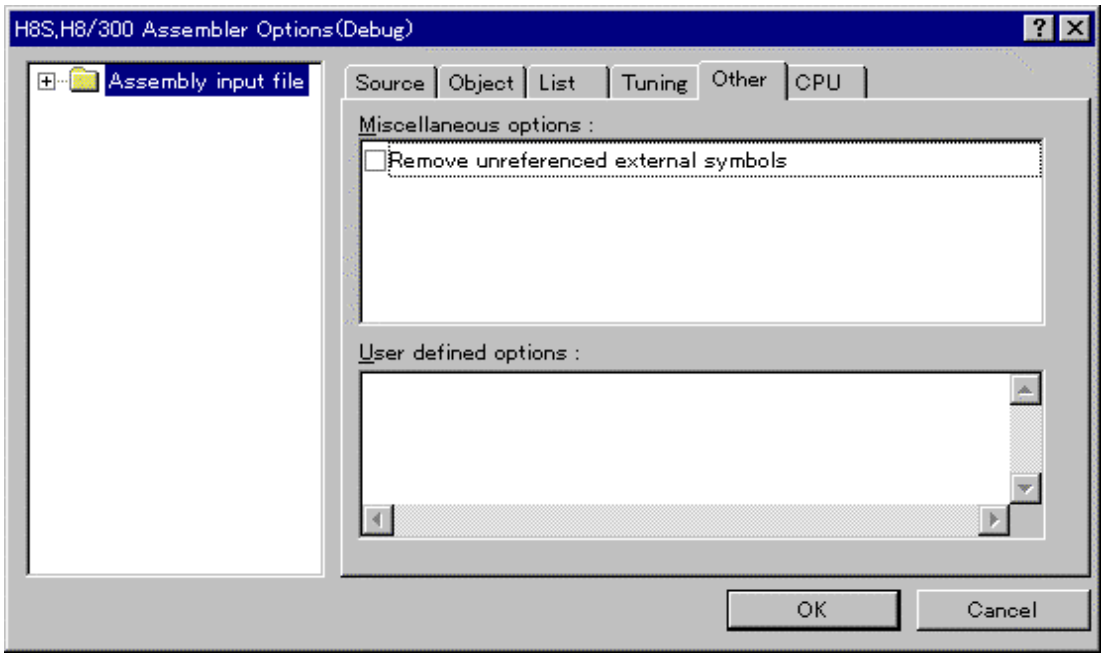
ダイアログメニュー	コマンドオプション	機能
@aa:8	*abs8*	8 ビット絶対アドレス形式シンボルの指定
@aa:16	*abs16*	16 ビット絶対アドレス形式シンボルの指定

【注】 対象とする外部参照／外部定義シンボルを選択します。

Specify all symbols

チェックボックス	機能
☑	すべての外部参照／外部定義シンボルに選択したサイズを割り当てる
☐	個別に割り付ける、または割り付けない

(5) Other タブ



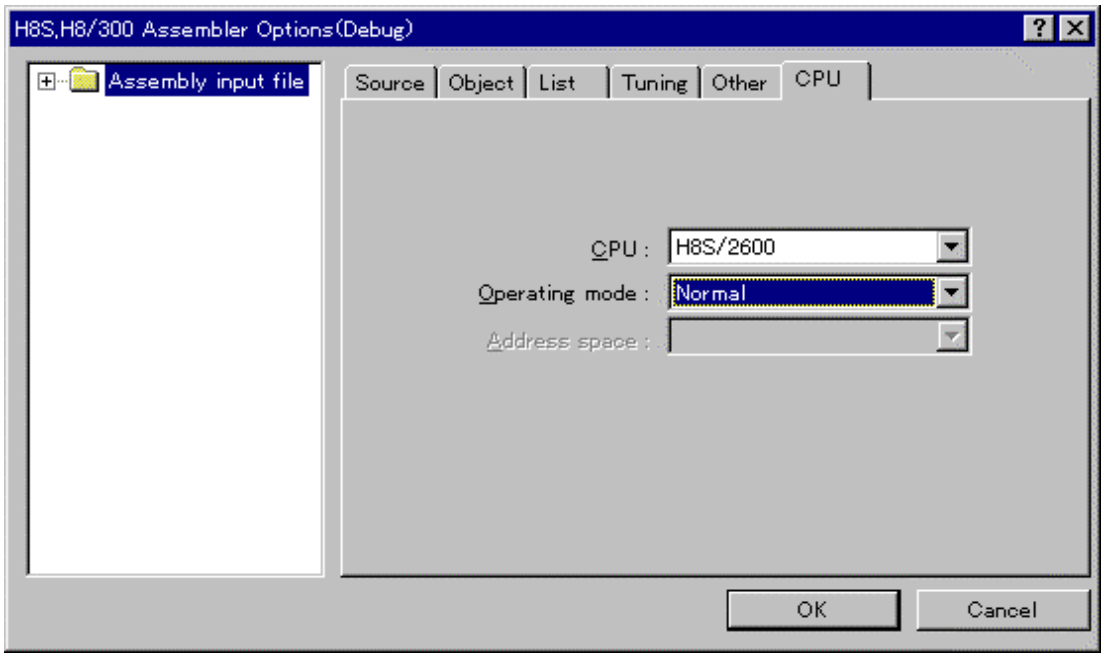
Miscellaneous options:

ダイアログメニュー	チェックボックス	コマンドオプション	機能
Remove unreferenced external symbols	☑	*exclude*	**未参照外部参照シンボル情報の出力抑止**
	☐	*noexclude*	**未参照外部参照シンボル情報の出力**

User defined options:

コマンドオプションを記述することができます。

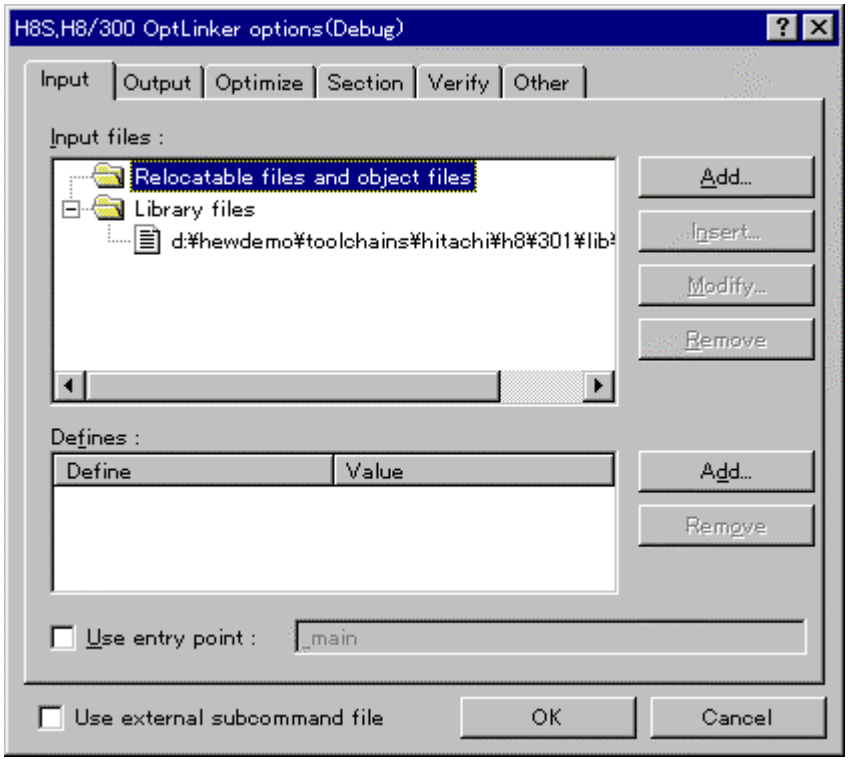
(6) CPU タブ



ダイアログメニュー			コマンドオプション	機能
CPU:	**Operating mode:**	**Address space:**		
default	-	-	-	.CPU **制御命令指定が有効**
H8S/2600	**Normal**	-	*cpu=2600n*	H8S/2600 **ノーマルモード**
	Advanced	**1M byte**	*cpu=2600a:20*	H8S/2600 **アドバンストモード**
		16M byte	*cpu=2600a[:24]*	H8S/2600 **アドバンストモード**
		256M byte	*cpu=2600a:28*	H8S/2600 **アドバンストモード**
		4G byte	*cpu=2600a:32*	H8S/2600 **アドバンストモード**
H8S/2000	**Normal**	-	*cpu=2000n*	H8S/2000 **ノーマルモード**
	Advanced	**1M byte**	*cpu=2000a:20*	H8S/2000 **アドバンストモード**
		16M byte	*cpu=2000a[:24]*	H8S/2000 **アドバンストモード**
		256M byte	*cpu=2000a:28*	H8S/2000 **アドバンストモード**
		4G byte	*cpu=2000a:32*	H8S/2000 **アドバンストモード**
H8/300H	**Normal**	-	*cpu=300hn*	H8/300H **ノーマルモード**
	Advanced	**1M byte**	*cpu=300ha:20*	H8/300H **アドバンストモード**
		16M byte	*cpu=300ha[:24]*	H8/300H **アドバンストモード**
H8/300	-	-	*cpu=300*	H8/300
H8/300L	-	-	*cpu=300l*	H8/300L

4.1.3 モジュール間最適化ツールのオプション

(1) Input タブ



Input files: リンクするロードモジュールとライブラリを指定します。

ダイアログメニュー	サブコマンド	機能
Relocatable files and object files	*input*	入力ファイルの指定*
Library files	*library*	ライブラリファイルの指定

【注】 * プロジェクトファイル以外の.obj を入力したい場合、またはプロジェクトファイルの入力順序を変更したい場合に指定します。

Defines:

ダイアログメニュー	サブコマンド	機能
-	*define*	外部参照シンボルの強制定義

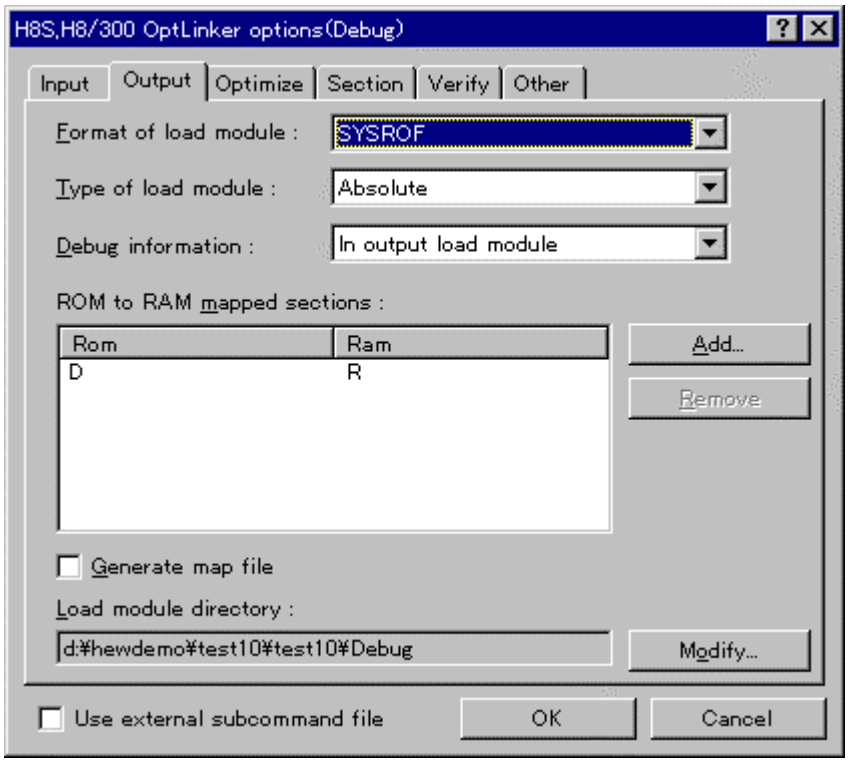
Use entry point:

ダイアログメニュー	サブコマンド	機能
-	*entry*	実行開始アドレスの指定

Use external subcommand file

ダイアログメニュー	サブコマンド	機能
-	*subcommand*	既存のサブコマンドファイルを指定

(2) Output タブ



Format of load module:ロードモジュールの出力形式を選択します。

ダイアログメニュー	サブコマンド	機能
ELF	*elf*	ELF フォーマット出力
SYSROF	*sysrof*	sysrof フォーマット出力
SYSROFPLUS	*sysrofplus*	sysrof フォーマット出力で、dwarf デバッグ情報出力

Type of load module:出力ロードモジュールファイル形式を指定します。

ダイアログメニュー	サブコマンド	機能
Absolute	*form△abs*	アブソリュート形式で出力
Relocatable	*form△rel*	リロケータブル形式で出力

Debug information:デバッグ情報の出力を指定します。

ダイアログメニュー	サブコマンド	機能
None	*nodebug*	デバッグ情報出力なし
In output load module	*debug*	デバッグ情報をロードモジュールに出力
In separate debug file (*.dbg)	*sdebug*	デバッグ情報をファイル出力

ROM to RAM mapped sections:

ダイアログメニュー	サブコマンド	機能
-	*rom*	初期化データ領域を ROM、RAM 上に二重に確保

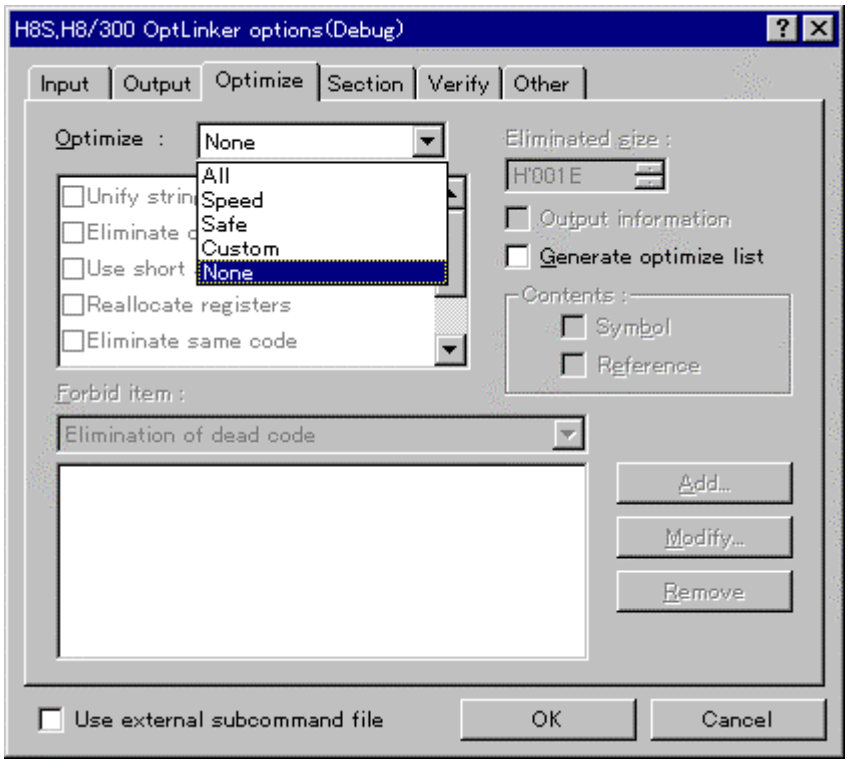
Generate map file

チェックボックス	サブコマンド	機能
☑	*print[△ファイル名]*	リンケージリストファイルの出力
☐	-	リンケージリストファイルの出力なし

Load module directory:

ダイアログメニュー	サブコマンド	機能
-	*output*	ロードモジュールの出力先ディレクトリを選択

(3) Optimize **タブ**



Optimize:最適化内容を指定します。

ダイアログメニュー		サブコマンド	機能
All		*Optimize*	すべての最適化項目が有効
Speed		*Optimize△speed*	スピード重視の最適化
Safe		*Optimize△safe*	安全な最適化
Custom		*Optimize△*	最適化項目の選択可能
	Unify strings	*String_unify*	定数／文字列の統合
	Elimibate dead code	*Symbol_delete*	未参照シンボルの削除
	Use short addressing	*Variable_access*	短絶対アドレスモードの活用
	Reallocate registers	*Register*	レジスタの再割り付け
	Eliminate same code	*Same_code*	共通コードの統合
	Eliminated size:	*Samesize*	共通コード統合対象サイズの指定
	Use indirect call/jump	*Function_call*	間接アドレスモードの活用
	Optimize branches	*Branch*	分岐命令の最適化
None		*Nooptimize*	最適化の抑止指定

Output information

チェックボックス	サブコマンド	機能
☑	*information*	最適化された関数名を表示
☐		最適化された関数名を表示なし

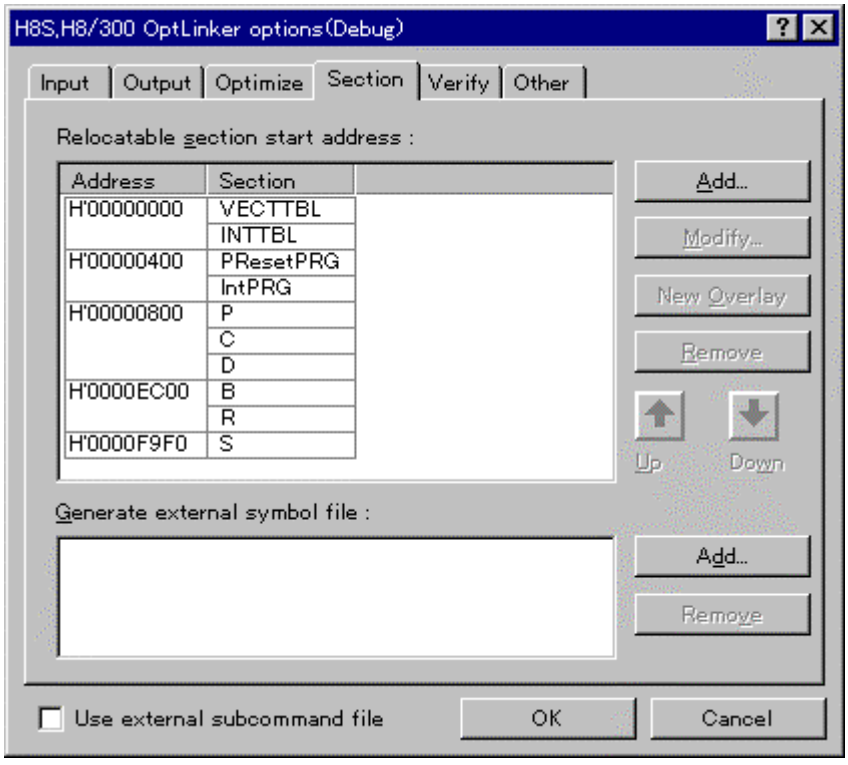
Generate optimize list

ダイアログメニュー		サブコマンド	機能
-		*Mlist[△ファイル名]*	最適化情報リストの出力
Contents:	**Symbol**	*show△symbol*	シンボル最適化情報出力
	Reference	*show△reference*	シンボル参照回数出力

Forbid item:

ダイアログメニュー	サブコマンド	機能
Elimination of dead code	*Symbol_forbid*	未参照シンボル削除の最適化を抑止する変数／関数名を指定
Elimination of same code	*Samecode_forbid*	共通コード統合の最適化を抑止する関数名を指定
Use of short addressing to	*Variable_forbid*	短絶対アドレッシングモード活用の最適化を抑止する変数名を指定
Use of indirect call/jump to	*Function_forbid*	間接アドレッシングモード活用の最適化を抑止する関数名を指定
Memory allocation in	*Absolute_forbid*	アドレス割り付けの対象外となるアドレス領域を指定

(4) Section **タブ**



Relocatable section start address:

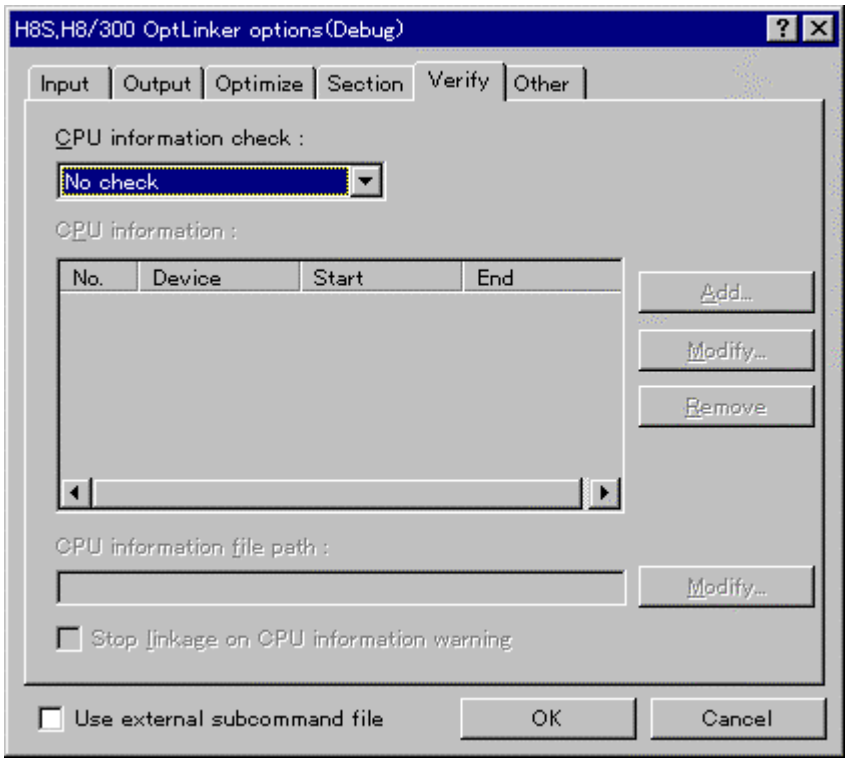
ダイアログメニュー	サブコマンド	機能
-	*start*	各セクションの先頭アドレスや結合順序を指定

Generate external symbol file:

ダイアログメニュー	サブコマンド	機能
-	*fsymbol*	リンケージ機能で解決した外部定義シンボルをアセンブラ制御命令形式でファイルに出力

【注】 出力するファイル名は<プロジェクト名>.fsy となります。

(5) Verify タブ



CPU information check:

ダイアログメニュー	サブコマンド	機能
No check		CPU 割り付けチェックなし
Check		CPU 情報ファイルによるメモリ割り付けチェック
Use CPU information file	*CPU*	既存 CPU 情報ファイルによるメモリ割り付けチェック

CPU information

ダイアログメニュー	サブコマンド	機能
-	-	CPU 情報ファイルを作成／変更 メモリ種別を指定し、それぞれのメモリのアドレスを指定

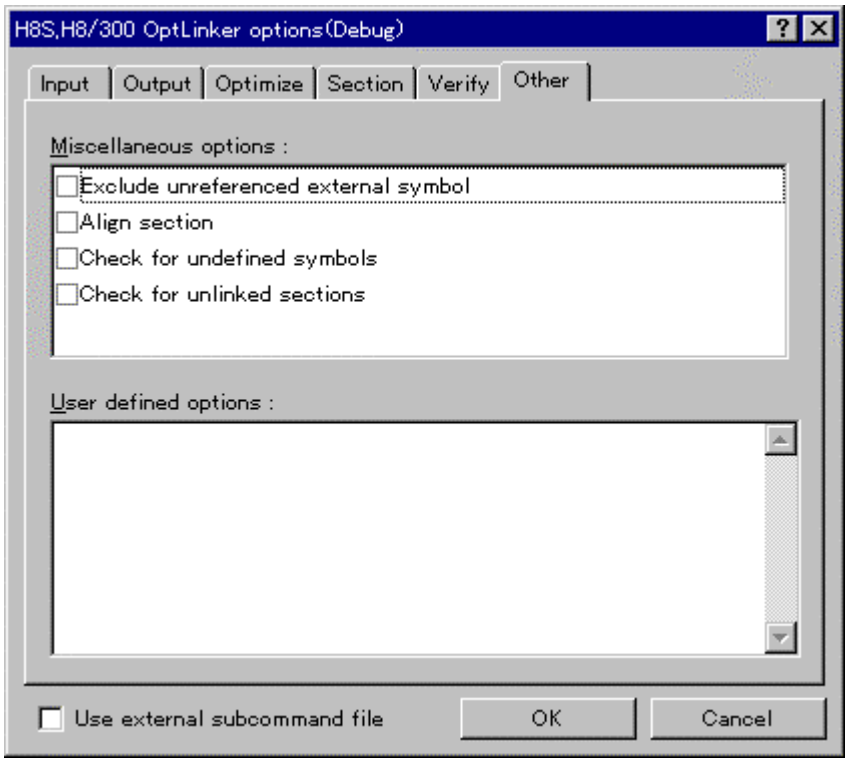
CPU information file path

ダイアログメニュー	サブコマンド	機能
-	-	既存の CPU 情報ファイルを指定

Stop linkage on CPU information warning

チェックボックス	サブコマンド	機能
☑	*CPUCheck*	CPU 情報ファイルによるメモリ割り付けチェック時にエラー出力
☐	-	メモリ割り付けチェック時にエラー出力なし

(6) Other **タブ**

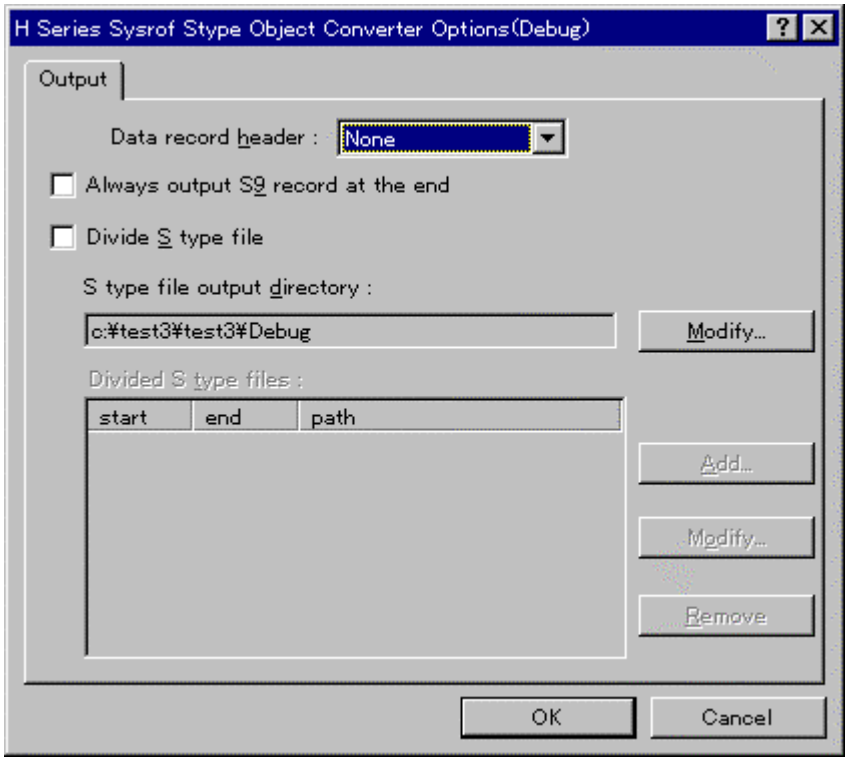


Miscellaneous options:その他の機能を指定します。

ダイアログメニュー	サブコマンド	機能
Exclude unrefernced external symbols	*Exclude*	**未参照ライブラリの結合抑止**
Align section	*Align_section*	**境界調整が異なるセクションのチェック**
Check for undefined symbols	*Udfcheck*	**未定義シンボル存在時のエラー出力**
Check for unlinked sections	*Check_section*	**アドレス未指定セクションのチェック**

4.1.4 S タイプコンバータのオプション

(1) Output タブ



Data record header:

ダイアログメニュー	コマンドオプション	機能
None	-	-
S1	*record=s1*	S1 データレコードで出力
S2	*record=s2*	S2 データレコードで出力
S3	*record=s3*	S3 データレコードで出力

Always output S9 record at the end

チェックボックス	コマンドオプション	機能
☑	*s9*	エントリアドレスが H'10000 を超える場合でも S9 レコードを終端に出力
☐	-	通常に出力

Divide S type file

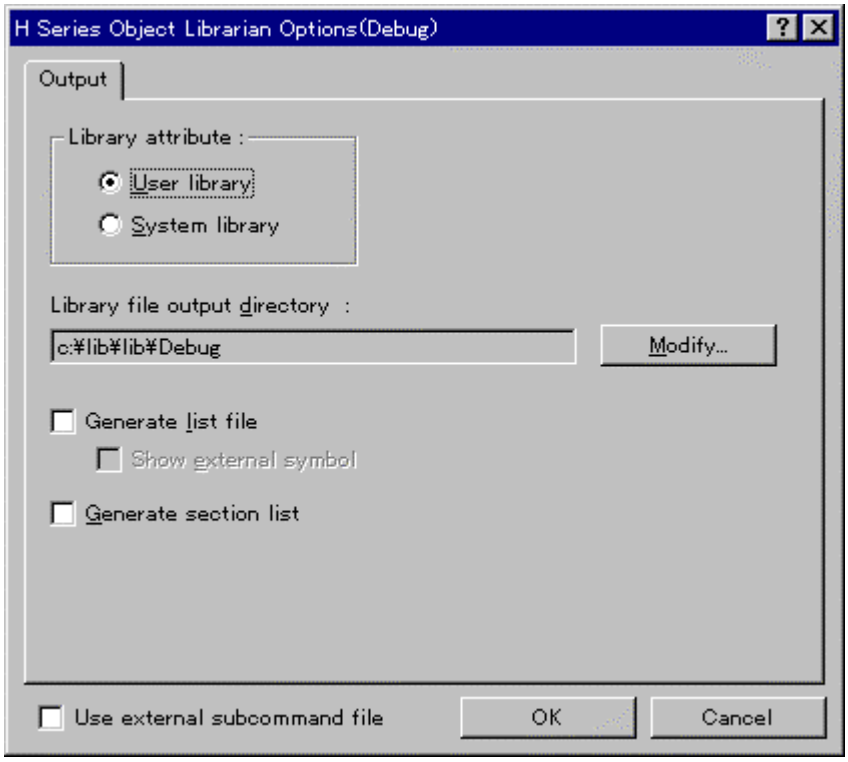
チェックボックス	コマンドオプション	機能
☑	-	S タイプを任意のアドレス範囲に分割して出力
☐	-	S タイプを分割しないで出力

S type file output directory

ダイアログメニュー	コマンドオプション	機能
-	-	S タイプファイル出力ディレクトリ指定

4.1.5 ライブラリアンのオプション

(1) Output タブ



Library attribute:出力するライブラリの属性を指定します。

ダイアログメニュー	オプション／サブコマンド	機能
User library	*output*	ライブラリ属性をユーザライブラリ指定
System library	*output*	ライブラリ属性をシステムライブラリ指定

Library file output directory:

ダイアログメニュー	オプション／サブコマンド	機能
-	*output*	ライブラリを出力するディレクトリを指定

Generate list file ライブラリリストファイルを出力するかどうかを指定します。

チェックボックス	オプション／サブコマンド	機能
☑	*list*	ライブラリファイルの内容表示
☐	-	ライブラリファイルの内容表示なし

Show external symbol モジュール内で定義されている定義シンボル名の出力を指定します。

チェックボックス	オプション／サブコマンド	機能
☑	*list*	モジュール内で定義されている外部定義シンボル名を表示
☐	-	表示なし

Generate section list セクション名リストファイルの出力を指定します。

チェックボックス	オプション／サブコマンド	機能
☑	*slist*	セクション内容表示
☐	-	セクション内容表示なし

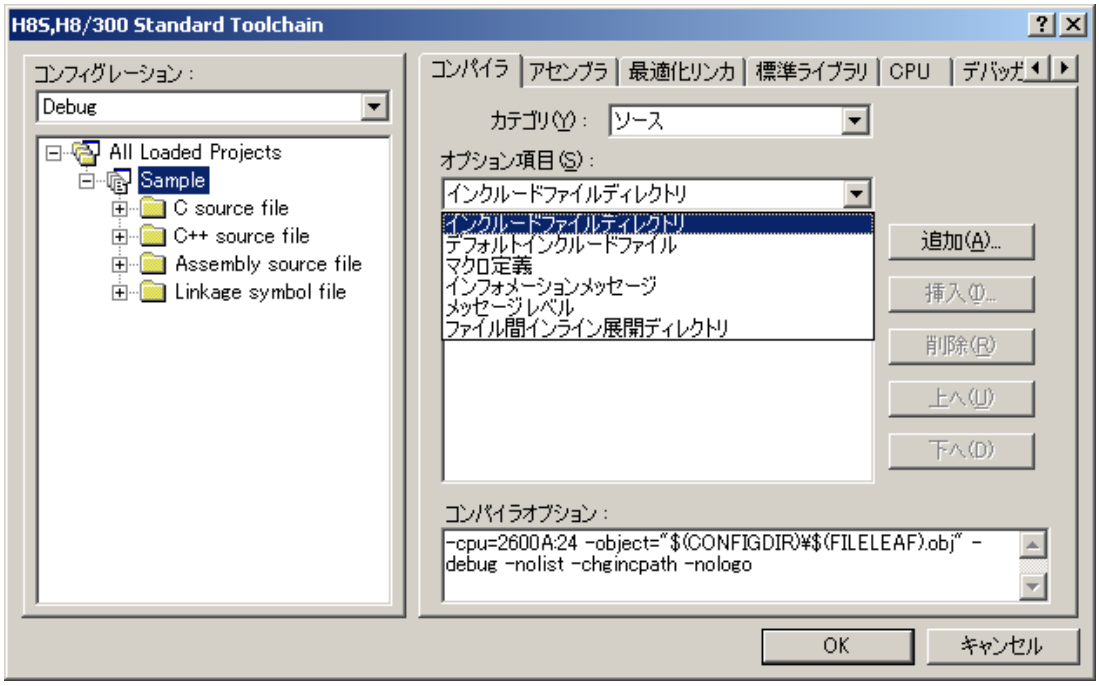
4.2 HEW2.0 以降のオプション指定方法

HEW1.2 のオプション指定方法については、「4.1 HEW1.2 のオプション指定方法」を参照ください。

4.2.1 C/C++コンパイラのオプション

H8S,H8/300 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**]



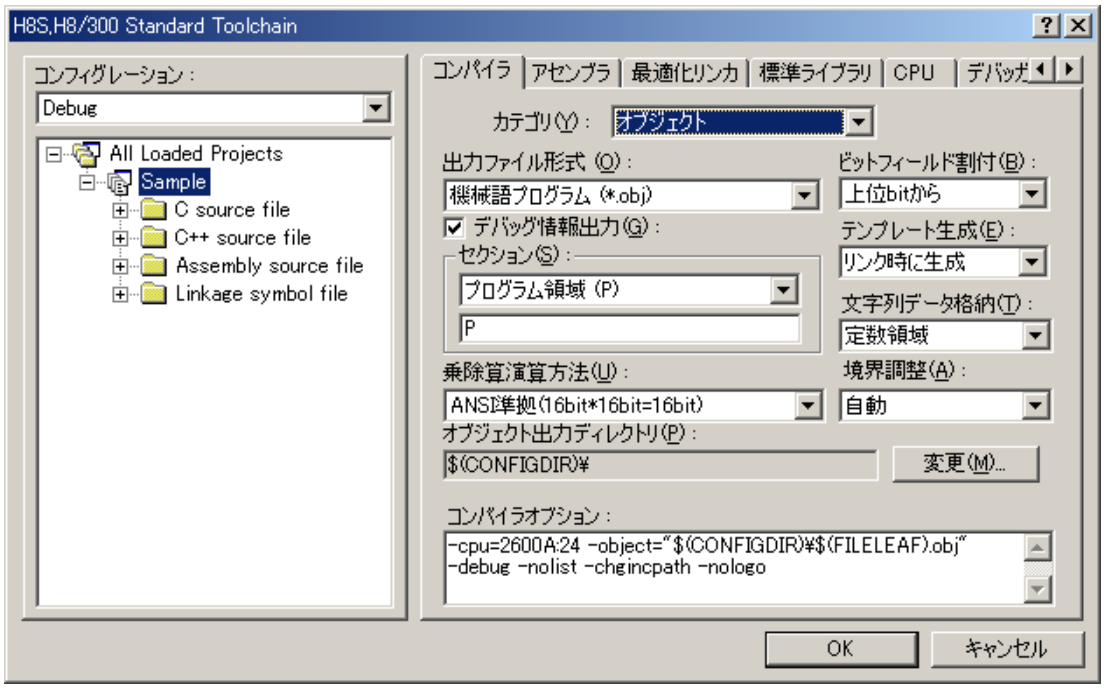
オプション項目:

ダイアログメニュー	コマンドオプション	機能
インクルードファイルディレクトリ	*include*	**インクルードファイルの取り込み先パス名指定**
デフォルトインクルードファイル	*preinclude*	**指定したファイルをコンパイル単位の先頭にインクルードファイル指定**
マクロ定義	*define*	**マクロ名の定義**
インフォメーションメッセージ	*message*	**インフォメーションメッセージの出力**
メッセージレベル	*change_message*	**インフォメーション、ウォーニングのエラーレベル変更**
ファイル間インライン展開ディレクトリ	*file_inline_path*	**ファイル間インライン展開ファイル取り込み先パス名を指定**

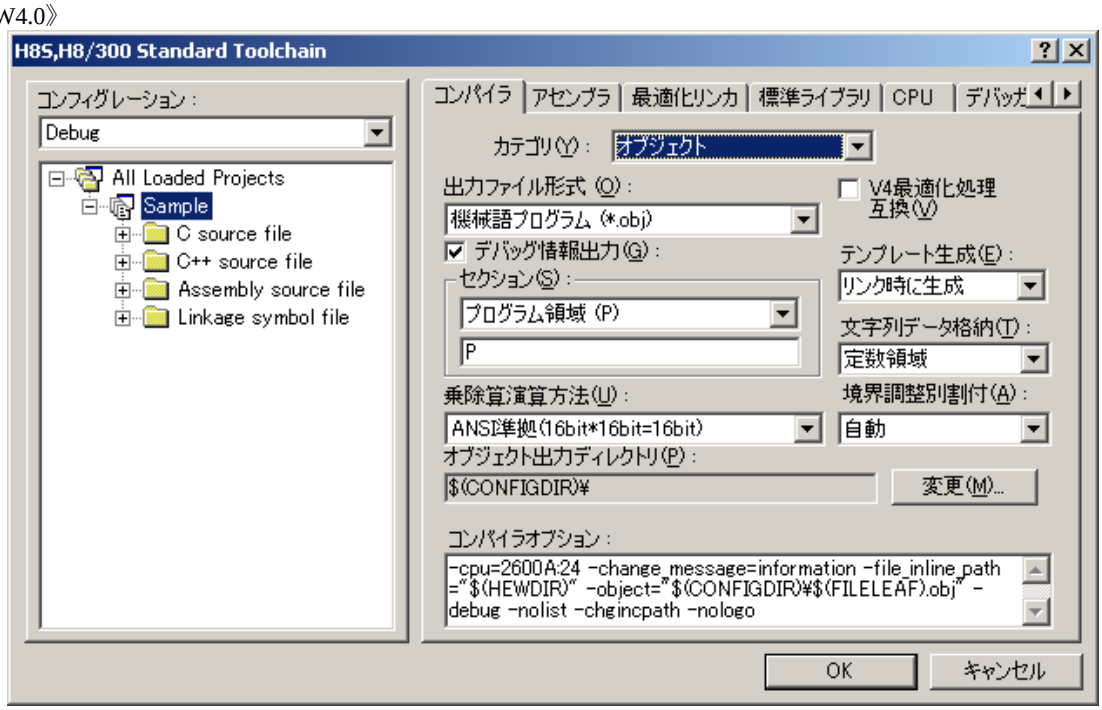
(2) **カテゴリ**:[**オブジェクト**]

オブジェクトカテゴリは HEW4.0 と HEW3.0 より前のバージョンで違いがあるので併せて紹介します。

《HEW2.0 から HEW3.0》



《HEW4.0》



出力ファイル形式:

ダイアログメニュー	コマンドオプション	機能
機械語プログラム (*.obj)	*code=machinecode*	**機械語プログラムを出力**
アセンブリプログラム (*.src)	*code=asmcode*	**アセンブリプログラムを出力**
プリプロセッサ展開プログラム (*.p/*.pp)	*preprocessor*	**プリプロセッサ展開後ソースプログラム出力**

デバッグ情報出力

チェックボックス	コマンドオプション	機能
☑	*debug*	デバッグ情報出力
☐	*nodebug*	デバッグ情報出力抑止

セクション:

ダイアログメニュー	コマンドオプション	機能
-	*section*	デフォルトのセクション名を変更

文字列データ格納:

ダイアログメニュー	コマンドオプション	機能
定数領域	*string=const*	文字列を定数領域へ出力
初期化データ領域	*string=data*	文字列を初期化データ領域へ出力

乗除算演算方法

ダイアログメニュー	コマンドオプション	機能
ANSI 準拠(16bit*16bit=16bit)	*nocpuexpand*	乗除算を ANSI C 言語仕様準拠でコード展開
ANSI 非準拠(16bit*16bit=32bit)	*cpuexpand*	乗除算を CPU 命令仕様にあわせてコードを展開

オブジェクト出力ディレクトリ

ダイアログメニュー	コマンドオプション	機能
-	*object*	オブジェクトファイルの出力先ディレクトリを指定

テンプレート生成

ダイアログメニュー	コマンドオプション	機能
なし	*template=none*	インスタンスを生成しない
内部リンケージ	*template=static*	参照されたものだけ内部リンケージとして生成
外部リンケージ	*template=used*	参照されたものだけ外部リンケージとして生成
全て生成	*template=all*	すべてのテンプレートのインスタンスを生成
リンク時に生成	*template=auto*	リンク時に必要なインスタンスの生成

ビットフィールド割り付け（HEW4.0 では CPU タブで設定）

ダイアログメニュー	コマンドオプション	機能
上位 bit から	*bit_order=left*	メンバを上位 bit から格納
下位 bit から	*bit_order=right*	メンバを下位 bit から格納

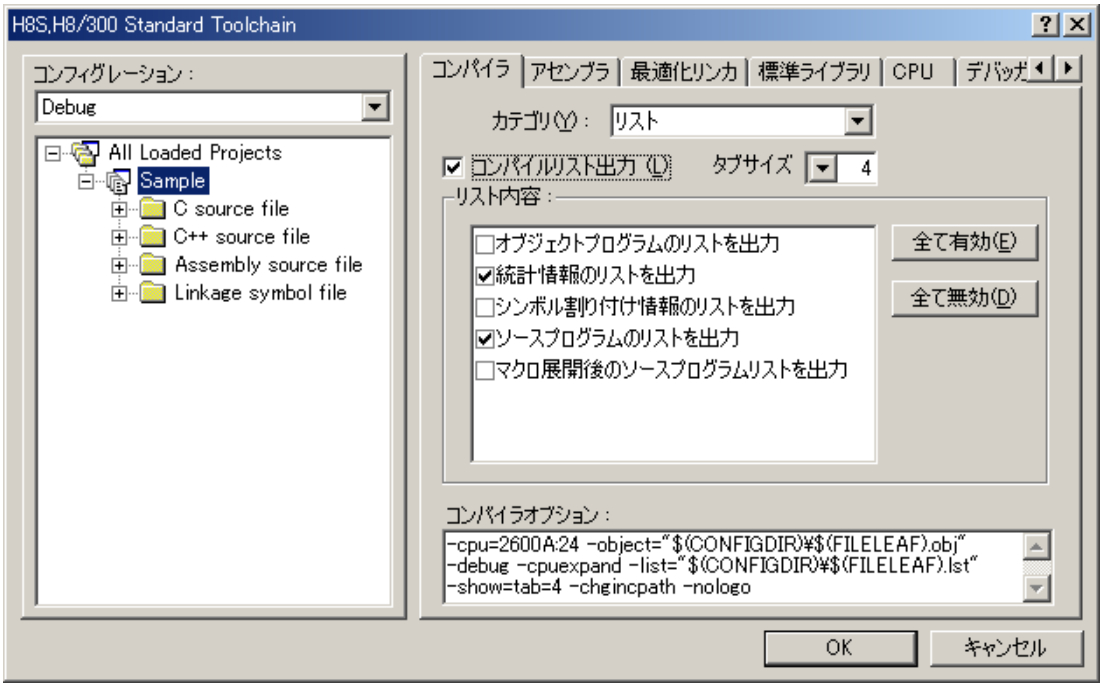
境界調整:

ダイアログメニュー	コマンドオプション	機能
なし	*noalign*	変数を宣言順に配置
自動	*align*	変数を境界調整数による空き領域が最小になるように配置
4byte	*align=4*	align の機能に加え、データの境界調整数が 4,2,1 のセクション分割し、それぞれの倍数のアドレスに配置します。それにより実行速度が向上します。

V4 最適化処理互換（HEW4.0 からサポート）

チェックボックス	コマンドオプション	機能
☑	*legacy=v4*	H8S の Ver4.0 と同様の最適化処理でオブジェクト出力
☐	-	H8S の最適化を V6.1 の最適化処理でオブジェクト出力

(3) **カテゴリ**:[**リスト**]



コンパイルリスト出力

チェックボックス	コマンドオプション	機能
☑	*list*	**オブジェクトリストファイル出力**
☐	*nolist*	**オブジェクトリストファイル出力抑止**

リスト内容:オブジェクトリストファイルへの出力内容を選択します。

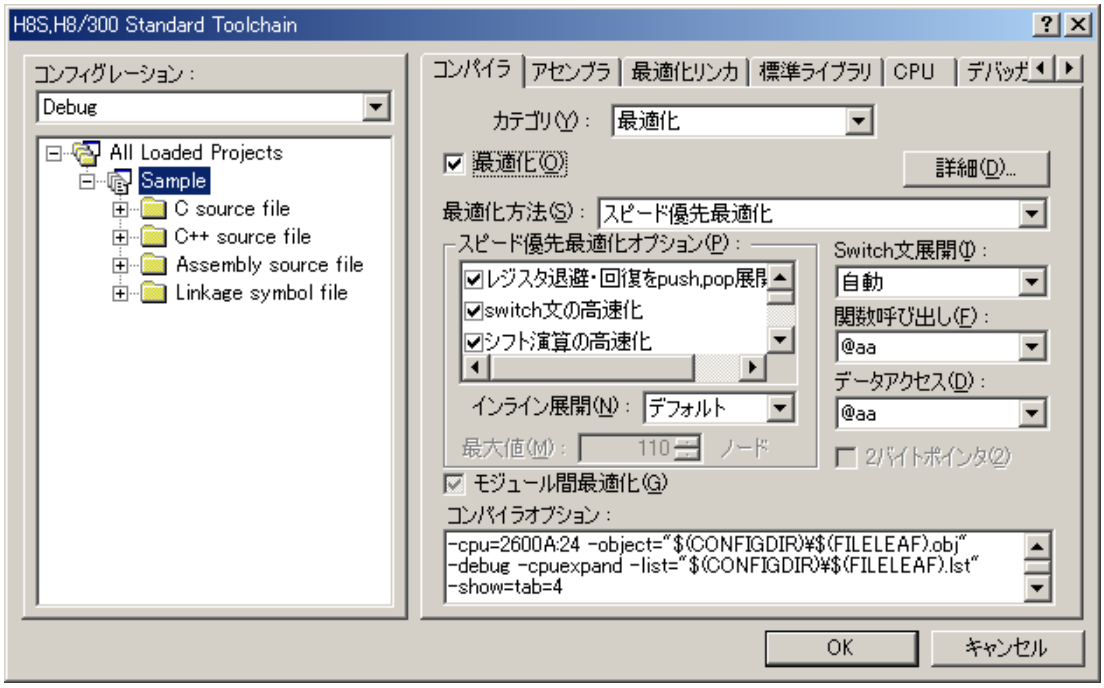
ダイアログメニュー	コマンドオプション	機能
オブジェクトプログラムのリストを出力	*show=object*	**オブジェクトリストの出力**
統計情報のリストを出力	*show=statistics*	**統計情報の出力**
シンボル割り付け情報のリストを出力	*show=allocation*	**シンボル割り付けリストの出力**
ソースプログラムのリスを出力	*show=source*	**ソースリストの出力**
マクロ展開後のソースプログラムリストの出力	*show=expansion*	**マクロ展開した後のソースプログラムリストの出力**

[Enable all]ボタンを押すとすべてが出力対象となり、[Disable all]ボタンを押すとすべてが無効となります。このとき出力されたオブジェクトリストファイルには何も出力されません。

タブサイズ

ダイアログメニュー	コマンドオプション	機能
4	*show=tab=4*	**リスト表示時のタブサイズ** 4 **を指定**
8	*show=tab=8*	**リスト表示時のタブサイズ** 8 **を指定**

(4) **カテゴリ**:[**最適化**]



最適化

チェックボックス	コマンドオプション	機能
☑	*optimize=1*	**最適化指定**
☐	*optimize=0*	**最適化指定なし**

最適化方法:最適化の方式を選択します。

ダイアログメニュー		コマンドオプション	機能
サイズ優先最適化		-	**サイズ優先の最適化**
スピード優先最適化		*speed*	**スピード優先の最適化**
スピード優先最適化サブオプション	**レジスタ退避・回復を push,pop 展開**	*speed=register*	**レジスタ退避/回復** push/pop **展開**
	swith 文の高速化	*speed=switch*	switch **文の高速化**
	シフト演算の高速化	*speed=shift*	**シフト演算の高速化**
	構造体代入式の高速化	*speed=struct*	**構造体代入式の高速化**
	四則演算、比較、代入式の高速化	*speed=expression*	**四則演算、比較、代入式の高速化**
	ループ文での帰納変数の削除	*speed=loop1*	**帰納変数の削除**
	ループ文での帰納変数削除、ループ展開	*speed=loop2*	**帰納変数の削除とループ展開**
	インライン展開	*speed=inline [=<数値>]*	**自動インライン展開の有無**

モジュール間最適化

チェックボックス	コマンドオプション	機能
☑	*goptimize*	**モジュール間最適化用付加情報出力**
☐	-	**モジュール間最適化用付加情報出力なし**

Switch 文展開: switch 文の展開方式を選択します。

ダイアログメニュー	コマンドオプション	機能
自動	*case=auto*	speed **オプション指定有無で判定**
If then 方式	*case=ifthen*	if_then **方式で展開**
テーブルジャンプ方式	*case=table*	**テーブルジャンプ方式で展開**

関数呼び出し:では関数の呼び出し形式を選択します。

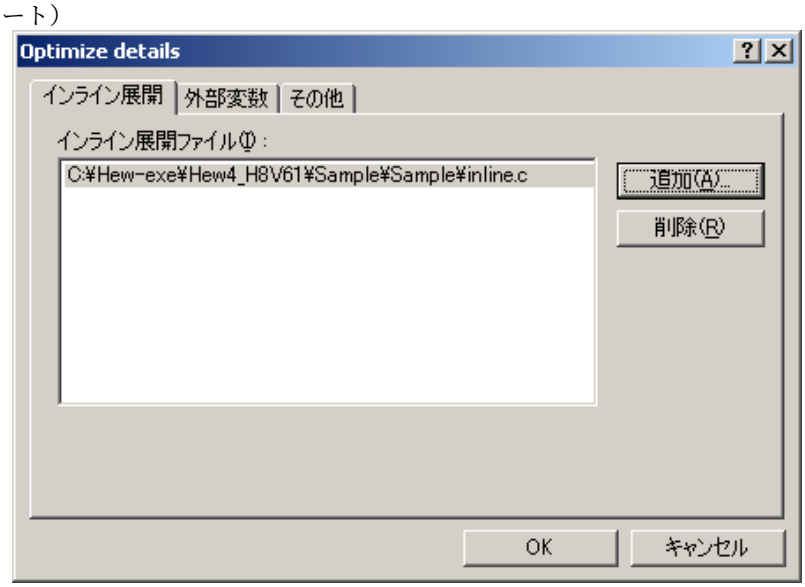
ダイアログメニュー	コマンドオプション	機能
@aa	-	**通常の関数呼び出し**
@@aa:8	*indirect=normal*	**メモリ間接形式の関数呼び出し**
@@vec:7	*indirect=extended*	**拡張メモリ間接形式の関数呼び出し**

データアクセス:データのアクセス方法を選択します。

ダイアログメニュー	コマンドオプション	機能
@aa	-	**通常のデータアクセス**
@aa:8	*abs8*	8 **ビット絶対アドレスアクセス**
@aa:16	*abs16*	16 **ビット絶対アドレスアクセス**

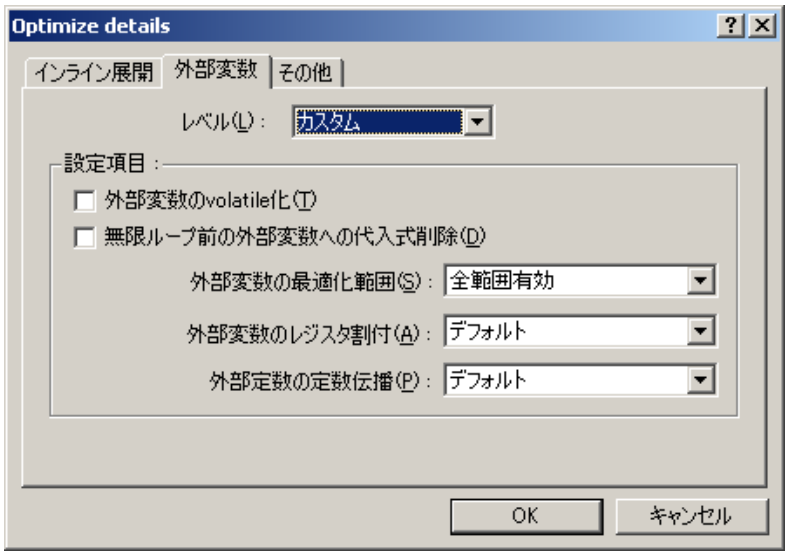
(a) **[詳細]ボタン**:**[インライン展開]**

（HEW4.0 からサポート）



ダイアログメニュー	コマンドオプション	機能
インライン展開ファイル	*file_inline*	**ファイル間インライン展開を実施**

(b) **[詳細]ボタン:[外部変数]**



レベル:外部変数に対する最適化のレベルを設定します。

ダイアログメニュー	コマンドオプション	機能
レベル 1	*volatile* *infinite_loop=0* *opt_range=noblock* *global_alloc=0* *const_var_propagate=0*	**外部変数最適化のすべてを抑止** *[外部変数の最適化]=[チェックあり]* *[無限ループ前の式削除]=[チェックなし]* *[外部変数の最適化範囲]=[最適化抑止]* *[外部変数のレジスタ割り付け]=[抑止]* *[const 定数伝播]=[抑止]*
レベル 2	*novolatile* *infinite_loop=0* *opt_range=noblock* *global_alloc=0* *const_var_propagate=0*	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** *[外部変数の最適化]=[チェックなし]* *[無限ループ前の式削除]=[チェックなし]* *[外部変数の最適化範囲]=[最適化抑止]* *[外部変数のレジスタ割り付け]=[抑止]* *[const 定数伝播]=[抑止]*
レベル 3	*novolatile* *infinite_loop=0* *opt_range=all* *global_alloc=1* *const_var_propagate=1*	volatile **指定のない外部変数をすべて最適化対象** *[外部変数の最適化]=[チェックなし]* *[無限ループ前の式削除]=[チェックなし]* *[外部変数の最適化範囲]=[最適化]* *[外部変数のレジスタ割り付け]=[割り付け]* *[const 定数伝播]=[定数伝播]*
カスタム	-	**外部変数に対する最適化をユーザが設定**

外部変数の最適化

チェックボックス	コマンドオプション	機能
☑	*volatile*	**外部変数の最適化をすべて抑止**
☐	*novolatile*	volatile **修飾子のない外部変数を最適化する**

無限ループ前の式削除

チェックボックス	コマンドオプション	機能
☑	*infinite_loop=1*	**無限ループ直前の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する**
☐	*infinite_loop=0*	**無限ループ直前での外部変数への代入を削除しない**

外部変数の最適化範囲

ダイアログメニュー	コマンドオプション	機能
最適化	*opt_range=all*	関数内の全範囲を対象に外部変数の最適化を行う
移動禁止	*opt_range=noloop*	ループ内にある外部変数やループ内判定式で使用されている外部変数を最適化の対象外にする
最適化抑止	*opt_range=noblock*	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

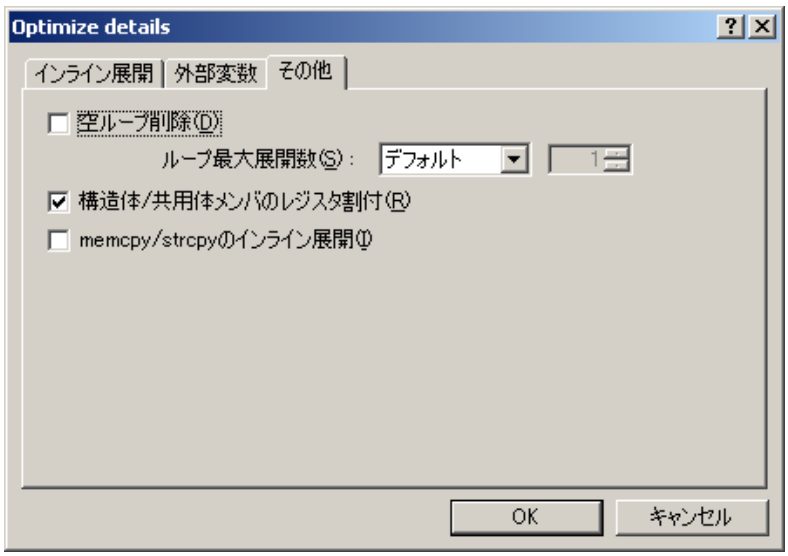
外部変数のレジスタ割り付け

ダイアログメニュー	コマンドオプション	機能
抑止	*global_alloc=0*	外部変数のレジスタ割り付けを抑止
割り付け	*global_alloc=1*	外部変数のレジスタ割り付けを行う
デフォルト	*global_alloc=1*	外部変数のレジスタ割り付けを行う

外部定数の定数伝播

ダイアログメニュー	コマンドオプション	機能
抑止	*const_var_propagate=0*	const 宣言された外部変数の定数伝播を抑止
定数伝播	*const_var_propagate=1*	const 宣言された外部変数の定数伝播を行う
デフォルト	*const_var_propagate=1*	const 宣言された外部変数の定数伝播を行う

(c) [詳細]ボタン:[その他]



空ループ削除

チェックボックス	コマンドオプション	機能
☑	*del_vacant_loop=1*	ループ内に処理がない場合、ループを削除
☐	*del_vacant_loop=0*	ループ内に処理がない場合、ループを削除しない

ループ展開最大数

ダイアログメニュー	コマンドオプション	機能
デフォルト	*max_unroll=2 or 1*	ループ展開数の最大数が 2 または 1
カスタム	*max_unroll=<**数値**>*	ループ展開数の最大数が 1 から 32 のユーザ任意の数値

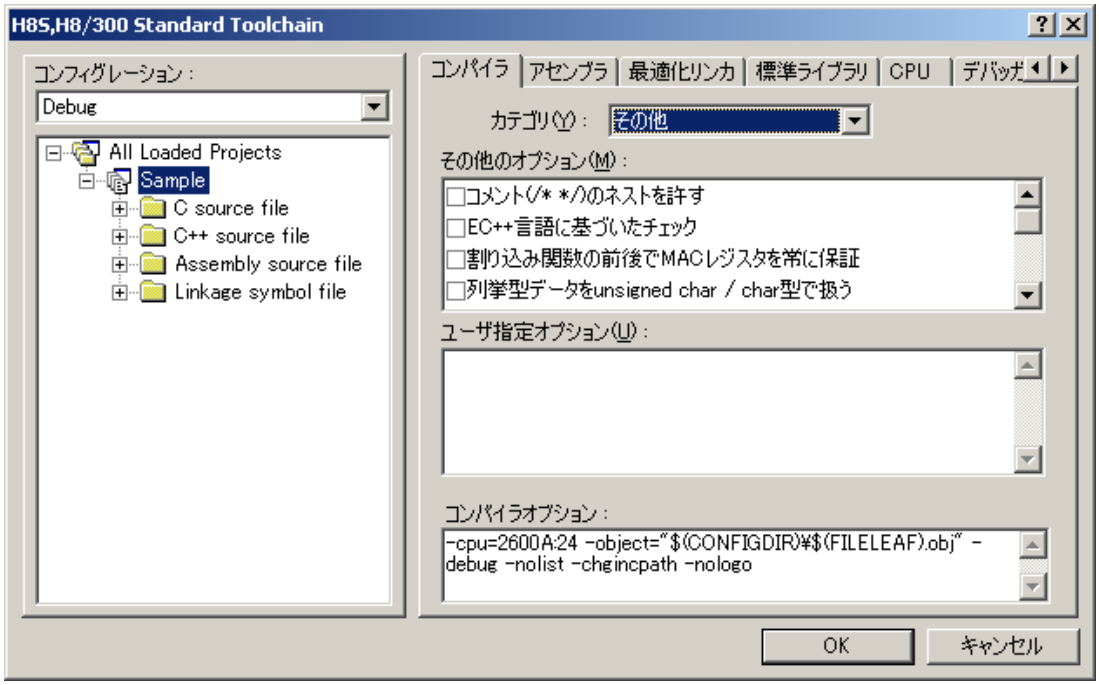
構造体／共用体メンバのレジスタ割り付け

チェックボックス	コマンドオプション	機能
☑	*struct_alloc=1*	構造体／共用体メンバのレジスタ割り付けを行う
☐	*struct_alloc=0*	構造体／共用体メンバのレジスタ割り付けを抑止

memcpy/strcpy のインライン展開

チェックボックス	コマンドオプション	機能
☑	*library=intrinsic*	memcpy/strcpy **をインライン展開**
☐	*library=function*	memcpy/strcpy **を関数呼び出し**

(5) **カテゴリ**:[**その他**]



その他のオプション:

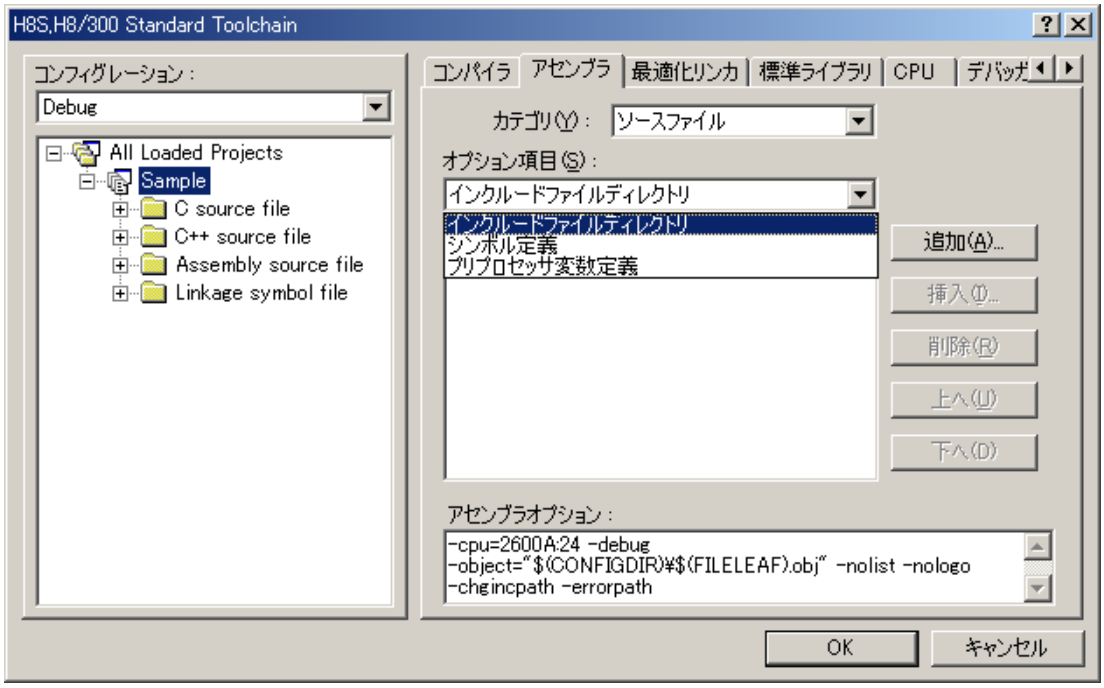
ダイアログメニュー	コマンドオプション	機能
コメント(/* */)のネストを許す	*comment*	**コメントのネストを許可**
EC++言語に基づいたチェック	*ecpp*	EC++**言語仕様に基づいてシンタックスチェック**
割り込み関数の前後で MAC レジスタを常に保証	*macsave*	MAC **レジスタ保証**
ループ判定式の最適化抑止	*volatile_loop*	**ループ判定式の最適化を抑止する**
列挙型データを unsigned char/char 型で扱う	*byteenum*	**列挙型のデータを** unsigned char/char **扱い**
変数割り付けレジスタ数を拡張	*regexpansion* *noregexpansion*	**変数割り付けレジスタ数** 2 **変数割り付けレジスタ数** 3
共通式削除の最適化を強化	*cmncode*	**共通式削除の最適化強化**
構造体の代入式を eepmov 命令で展開	*eepmov*	**構造体の代入を** eepmov **命令で展開**
ループ判定式の最適化抑止	*volatile_loop*	**ループ判定式の最適化を抑止**
プリプロセッサ展開時に#line 出力抑止	*noline*	**プリプロセッサ展開時に**#line **の出力を抑止**
register 指定変数の優先レジスタ割り付け	*enable_register*	register **記憶クラスを指定した変数を優先的にレジスタ割り付け**
ANSI 準拠対応拡張	*strict_ansi*	**以下を** ANSI **準拠で行う** **・浮動小数点演算の結合則**

ユーザ指定オプション:ではコマンドオプションを指定することができます。

4.2.2 アセンブラのオプション

H8S,H8/300 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。

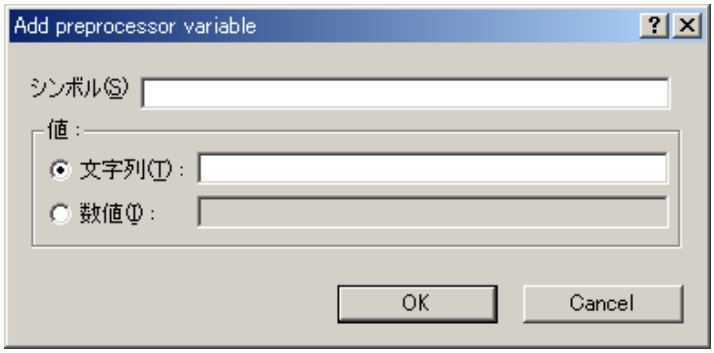
(1) **カテゴリ**:[**ソースファイル**]



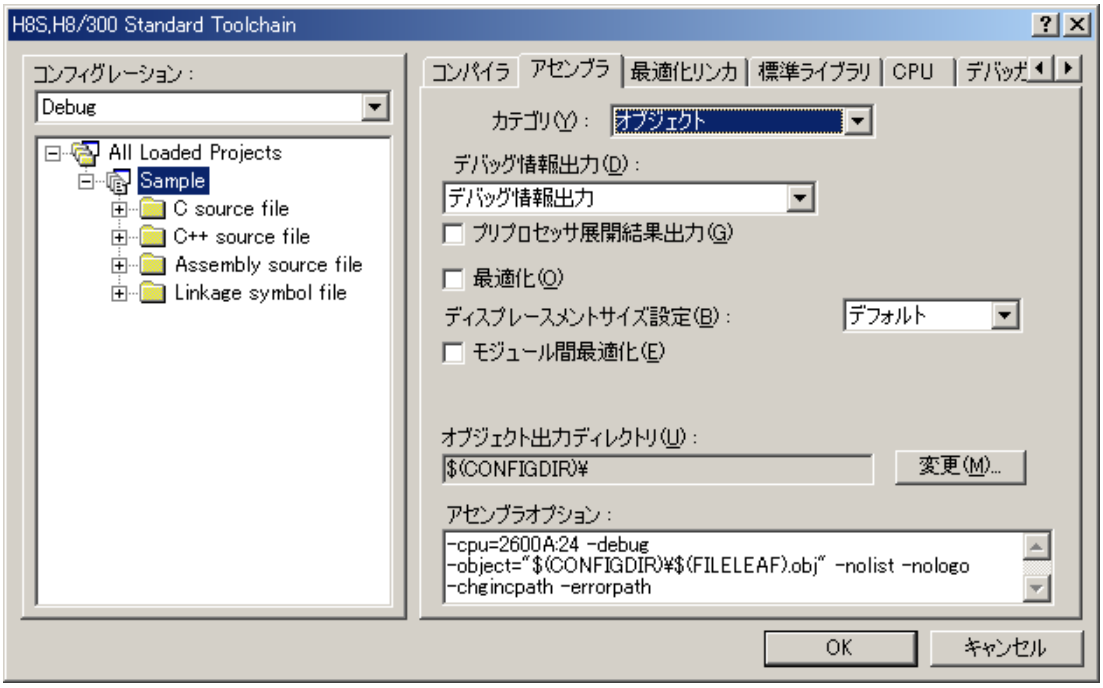
オプション項目:

ダイアログメニュー	コマンドオプション	機能
インクルードファイルディレクトリ	*include*	**インクルードファイルのディレクトリ指定**
シンボル定義	*define*	**文字列の置き換えの定義**
プリプロセッサ変数*定義	*assigna*	**整数型プリプロセッサ変数の定義**
	assignc	**文字型プリプロセッサ変数の定義**

【注】 * **次のダイアログボックスで指定します。**



(2) **カテゴリ**:[**オブジェクト**]



デバッグ情報出力:

ダイアログメニュー	コマンドオプション	機能
デフォルト	-	.DEBUG 制御命令のみ有効
デバッグ情報出力	*debug*	**デバッグ情報出力指定**
デバッグ情報出力なし	*nodebug*	**デバッグ情報出力抑止**

プリプロセッサ展開結果出力

チェックボックス	コマンドオプション	機能
☑	*expand*	**プリプロセッサの展開結果の出力**
☐	-	**プリプロセッサの展開結果の出力なし**

最適化

チェックボックス	コマンドオプション	機能
☑	*optimize*	**最適化の指定**
☐	*nooptimize*	**最適化の指定なし**

ディスプレースメントサイズ設定:

ダイアログメニュー	コマンドオプション	機能
デフォルト		**ソースファイルの制御命令に従います**
8bit	*br_relative=8*	**分岐命令のディスプレースメントが前方参照の場合のディスプレースメントサイズ** 8 **ビット**
16bit	*br_relative=16*	**分岐命令のディスプレースメントが前方参照の場合のディスプレースメントサイズ** 16 **ビット**

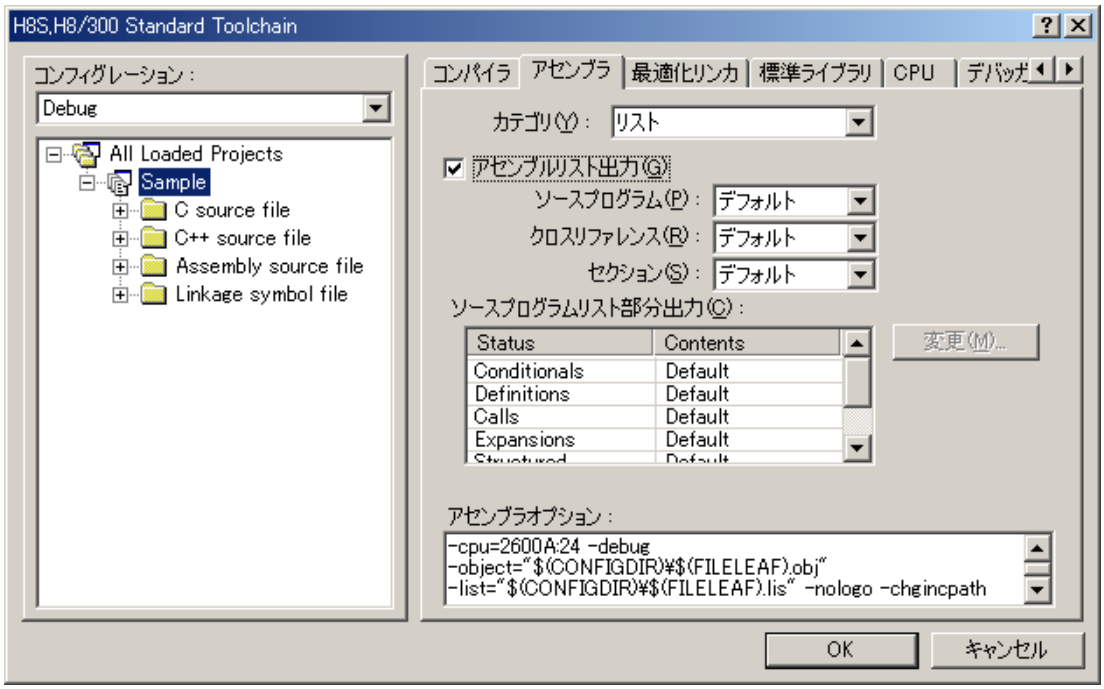
モジュール間最適化

チェックボックス	コマンドオプション	機能
☑	*goptimize*	**モジュール間最適化情報の出力**
☐		**モジュール間最適化情報の出力なし**

オブジェクト出力ディレクトリ

ダイアログメニュー	コマンドオプション	機能
-	*object*	**オブジェクトの出力ディレクトリ指定**

(3) **カテゴリ**:[**リスト**]



アセンブルリスト出力

チェックボックス	コマンドオプション	機能
☑	*list*	**アセンブルリストの出力**
☐	*nolist*	**アセンブルリストの出力なし**

ソースプログラム;

ダイアログメニュー	コマンドオプション	機能
出力	*source*	**ソースプログラムリストの出力**
出力なし	*nosource*	**ソースプログラムリストの出力なし**

クロスリファレンス:

ダイアログメニュー	コマンドオプション	機能
出力	*cross_refernce*	**クロスリファレンスリストの出力**
出力なし	*nocross_refernce*	**クロスリファレンスリストの出力なし**

セクション:

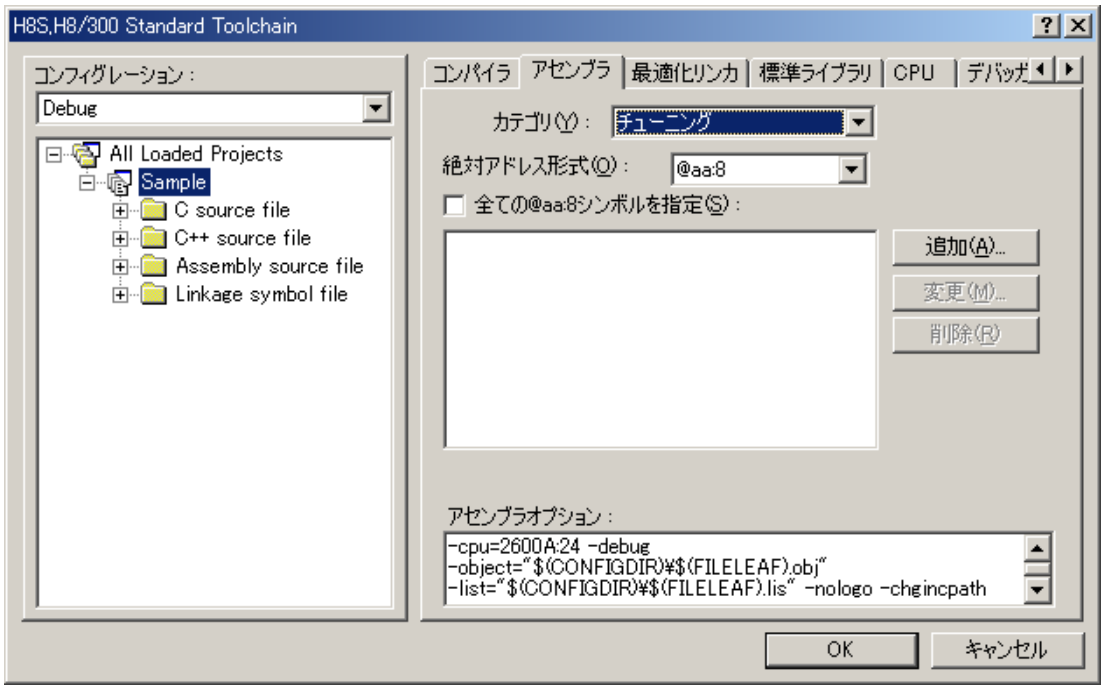
ダイアログメニュー	コマンドオプション	機能
出力	*cross_refernce*	**セクション情報リストの出力**
出力なし	*nocross_refernce*	**セクション情報リストの出力なし**

ソースプログラムリスト部分出力:リストファイルに出力する内容を指定します。

ダイアログメニュー	コマンドオプション	機能
Conditionals（条件つき不成立）	*show=conditionals*	.AIF、.AIFDEF **の不成立部分**
Definitions（定義）	*show=definitions*	**マクロ定義部分、**.AREPEAT,.AWHILE **定義部分、**.INCLUDE, .ASSIGNA,.ASSIGNC **制御文**
Calls（コール）	*show=calls*	**マクロコール文、**.AIF,.AIFDEF,.AENDI **制御文**
Expansions（展開）	*show=expansions*	**マクロ展開部分、**.AREPEAT,.AWHILE **展開部分**
Structured（構造化展開）	*show=structured*	**構造化アセンブリ展開部分**
Code（コード）	*show=code*	**制御命令のオブジェクトコード表示がソースステートメントの行数を超える部分**

【注】 各出力内容で<u>デフォルト</u>（Default）を選択すると、ソースリスト中の制御命令の指定を有効にします。

(4) **カテゴリ**:[**チューニング**]



絶対アドレス形式:

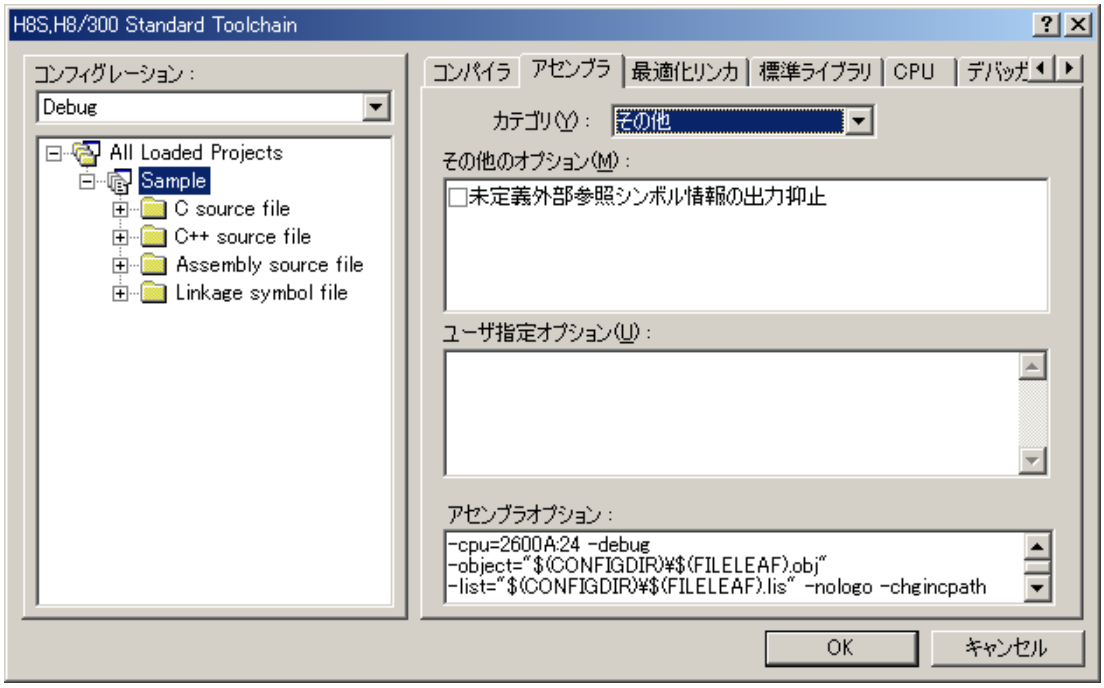
ダイアログメニュー	コマンドオプション	機能
@aa:8	*abs8*	8 **ビット絶対アドレス形式シンボルの指定**
@aa:16	*abs16*	16 **ビット絶対アドレス形式シンボルの指定**

【注】 対象とする外部参照／外部定義シンボルを選択します。

すべての@aa8 シンボルを指定

チェックボックス	機能
☑	**すべての外部参照／外部定義シンボルに選択したサイズを割り当てる**
☐	**個別に割り付ける、または割り付けない**

(5) **カテゴリ**:[**その他**]



その他のオプション:

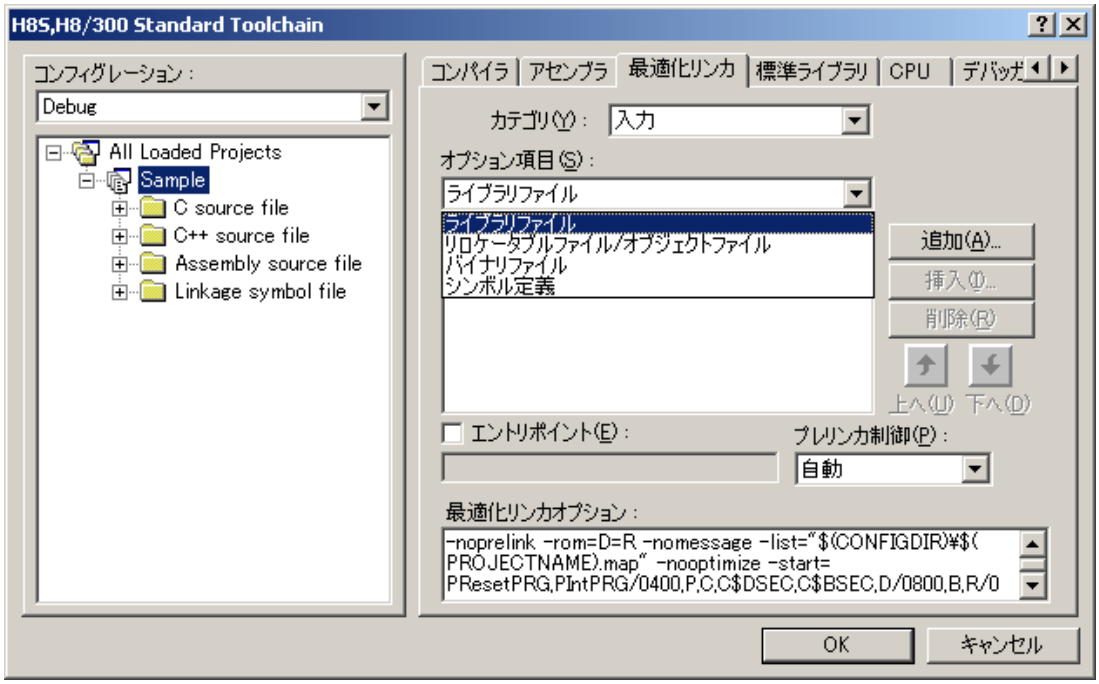
ダイアログメニュー	チェックボックス	コマンドオプション	機能
未定義外部参照シンボル情報の出力抑止	☑	*exclude*	**未参照外部参照シンボル情報の出力抑止**
	☐	*noexclude*	**未参照外部参照シンボル情報の出力**

ユーザ指定オプション:
コマンドオプションを記述することができます。

4.2.3 最適化リンケージエディタのオプション

H8S,H8/300 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。

(1) **カテゴリ**:[**入力**]



オプション項目:

ダイアログメニュー	コマンドオプション	機能
ライブラリファイル	*library*	**入力ライブラリ名指定**
リロケータブルファイル／オブジェクトファイル	*input*	**入力ファイル指定**
バイナリファイル	*binary*	**入力バイナリファイル指定**
シンボル定義	*define*	**未定義シンボルの強制定義**

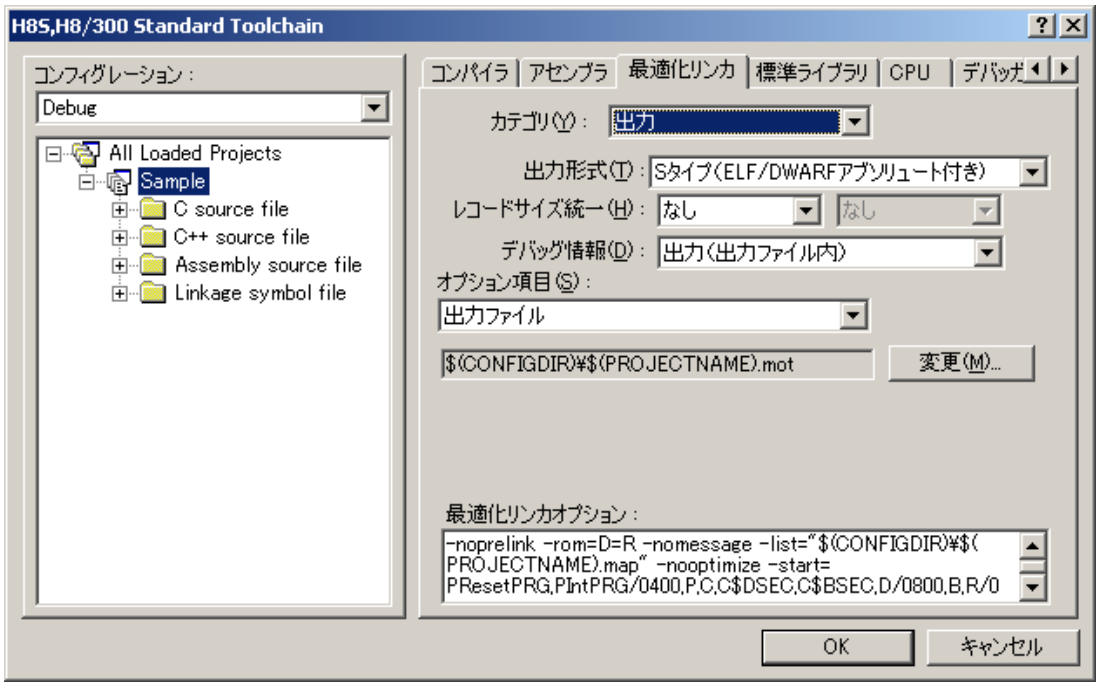
エントリポイント指定:

チェックボックス	コマンドオプション	機能
☑	*entry*	**エントリシンボル、エントリアドレスの指定**
☐	-	**エントリシンボル、エントリアドレスの指定なし**

プレリンカ制御:

ダイアログメニュー	コマンドオプション	機能
自動	-	**インスタンス情報ファイルがないとき、プレリンカを起動しません**
不使用	*noprelink*	**プレリンカを起動しません**
使用	-	**プレリンカを起動する**

(2) **カテゴリ**:[**出力**]



出力形式:

ダイアログメニュー	コマンドオプション	機能
アブソリュート (ELF/DWARF)	*form=absolute*	ERF/DWARF **形式のアブソリュートロードモジュールを出力**
アブソリュート (SYSROF)	*form=absolute helfcnv.exe*	SYSROF **形式のアブソリュートロードモジュールを出力**
リロケータブル	*form=relocate*	**リロケータブルロードモジュールを出力**
システムライブラリ	*form=library=s*	**システムライブラリを出力**
ユーザライブラリ	*form=library=u*	**ユーザライブラリを出力**
HEX(ELF/DWARF アブソリュート付き)	*form=hexadecimal*	HEX **ファイルを出力**
S タイプ(ELF/DWARF アブソリュート付き)	*form=stype*	S-type **ファイルを出力**
バイナリ(ELF/DWARF アブソリュート付き)	*form=binary*	**バイナリファイルを出力**

レコードサイズ統一:

ダイアログメニュー	コマンドオプション	機能
なし	-	**ロードアドレスどおりに出力**
H16	*record=h16*	HEX **レコードの出力**
H20	*record=h20*	**拡張** HEX **レコードの出力**
H32	*record=h32*	32bitHEX **レコードの出力**
S1	*record=s1*	S1 **レコードの出力**
S2	*record=s2*	S2 **レコードの出力**
S3	*record=s3*	S3 **レコードの出力**

デバッグ情報:

ダイアログメニュー	コマンドオプション	機能
なし	*nodebug*	デバッグ情報出力なし
出力(出力ファイル内)	*debug*	デバッグ情報をロードモジュールに出力
デバッグ情報ファイル (*.dbg)出力	*sdebug*	デバッグ情報をファイル出力

オプション項目:

ダイアログメニュー	コマンドオプション	機能
出力ファイル	-	出力ファイルのパス指定
ROM か RAM へマップするセクション	*rom*	RAM に領域を確保し、シンボルを RAM 上のアドレスでリロケーション解決
出力ファイルの分割	-	出力範囲の指定有無
空きエリア出力指定	-	空きエリアへの出力値の指定
メッセージ出力指定	-	インフォメーションレベルメッセージ出力抑止
データ詰め込み	-	コンパイル単位間の空き領域を詰めてデータを配置

インフォメーションレベルメッセージ抑止:

チェックボックス	コマンドオプション	機能
☑	*nomessage*	インフォメーションレベルメッセージ出力抑止
☐	*message*	インフォメーションレベルメッセージ出力

参照されない定義シンボルの通知:

チェックボックス	コマンドオプション	機能
☑	*msg_unused*	1回も参照されない定義シンボルをメッセージ出力により通知
☐	-	1回も参照されない定義シンボルをメッセージ出力により通知しない

出力ファイルの分割:

チェックボックス	コマンドオプション	機能
☑	*output*	出力ファイル名指定、出力範囲の指定あり
☐	-	出力ファイル名指定、出力範囲の指定なし

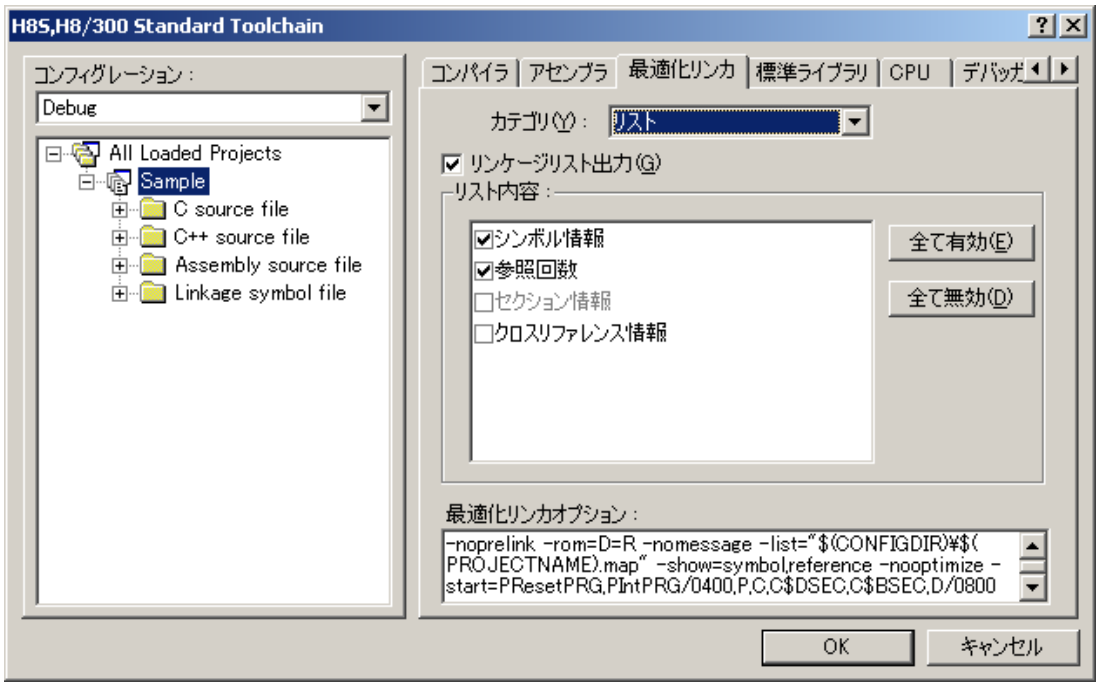
空きエリア出力:

チェックボックス	コマンドオプション	機能
☑	*space=<**数値**>*	空きエリアへの出力値を指定
☐	-	空きエリアへの出力値を指定なし

セクション内データ詰め込み配置:

チェックボックス	コマンドオプション	機能
☑	*data_stuff*	コンパイル単位間の空き領域を詰めてデータを配置
☐	-	コンパイル単位間の空き領域を詰めてデータを配置しない

(3) **カテゴリ**:[**リスト**]



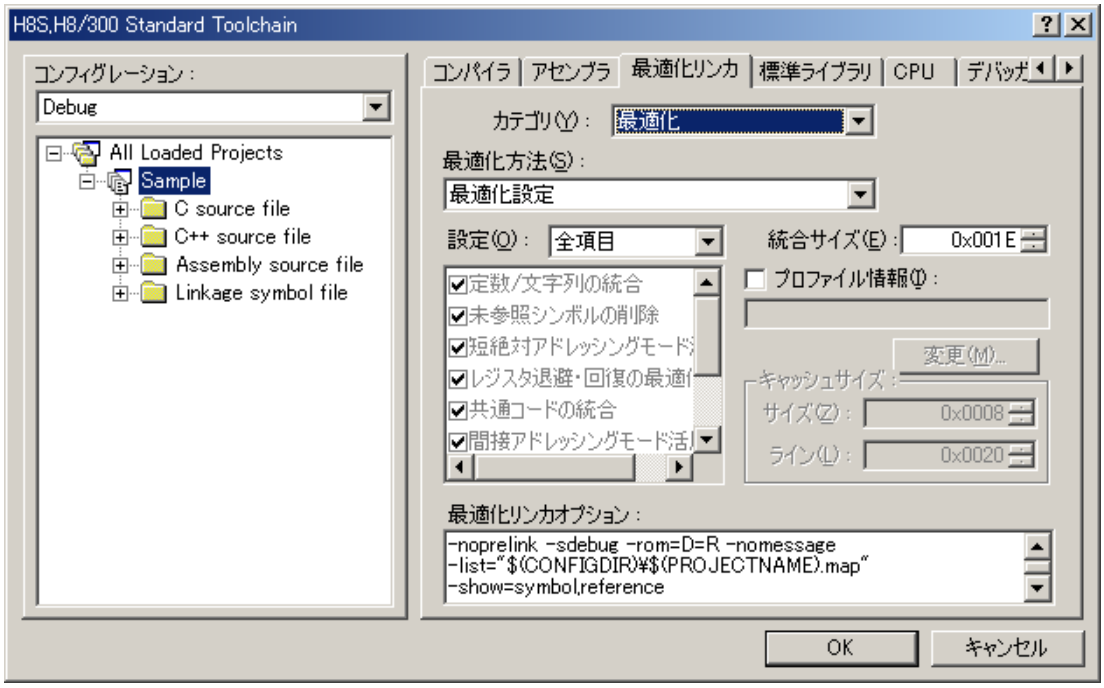
リンケージリスト出力:

チェックボックス	コマンドオプション	機能
☑	*list*	**リストファイルの出力**
☐	-	**リストファイルの出力なし**

リスト内容:

ダイアログメニュー	コマンドオプション	機能
シンボル情報	*show=symbol*	**シンボル名一覧の出力**
参照回数	*show=reference*	**シンボルの参照回数の出力**
セクション情報	*sher=section*	**セクション一覧の出力**
クロスリファレンス情報	*show=xreference*	**クロスリファレンス情報の出力**

(4) **カテゴリ**:[**最適化**]



最適化方法:

ダイアログメニュー		コマンドオプション	機能
最適化設定		-	**最適化指定**
最適化部分抑止		-	**特定のシンボル、アドレス範囲の最適化を抑止**
	未参照シンボル削除抑止シンボル	*Symbol_forbid*	**未参照シンボル削除の最適化を抑止する変数／関数名を指定**
	共通コード統合抑止シンボル	*Samecode_forbid*	**共通コード統合の最適化を抑止する関数名を指定**
	短絶対アドレッシングモード活用抑止シンボル	*Variable_forbid*	**短絶対アドレッシングモード活用の最適化を抑止する変数名を指定**
	間接アドレッシングモード活用抑止シンボル	*Function_forbid*	**間接アドレッシングモード活用の最適化を抑止する関数名を指定**
	最適化抑止アドレス範囲	*Absolute_forbid*	**アドレス割り付けの対象外となるアドレス領域を指定**

設定:

ダイアログメニュー		コマンドオプション	機能
全項目		*Optimize*	**すべての最適化項目が有効**
スピード重視		*Optimize△speed*	**スピード重視の最適化**
安全な最適化		*Optimize△safe*	**安全な最適化**
カスタム		*Optimize△*	**最適化項目の選択可能**
	定数／文字列の統合	*String_unify*	**定数／文字列の統合**
	未参照シンボルの削除	*Symbol_delete*	**未参照シンボルの削除**
	短絶対アドレッシングモード活用	*Variable_access*	**短絶対アドレスモードの活用**
	レジスタ退避・回復の最適化	*Register*	**レジスタの再割り付け**
	共通コードの統合	*Same_code*	**共通コードの統合**
	間接アドレッシングモード活用	*Function_call*	**間接アドレスモードの活用**
	分岐命令の最適化	*Branch*	**分岐命令の最適化**
	アドレッシングモードの短縮	*Samesize*	**共通コード統合対象サイズの指定**
なし		*Nooptimize*	**最適化の抑止指定**

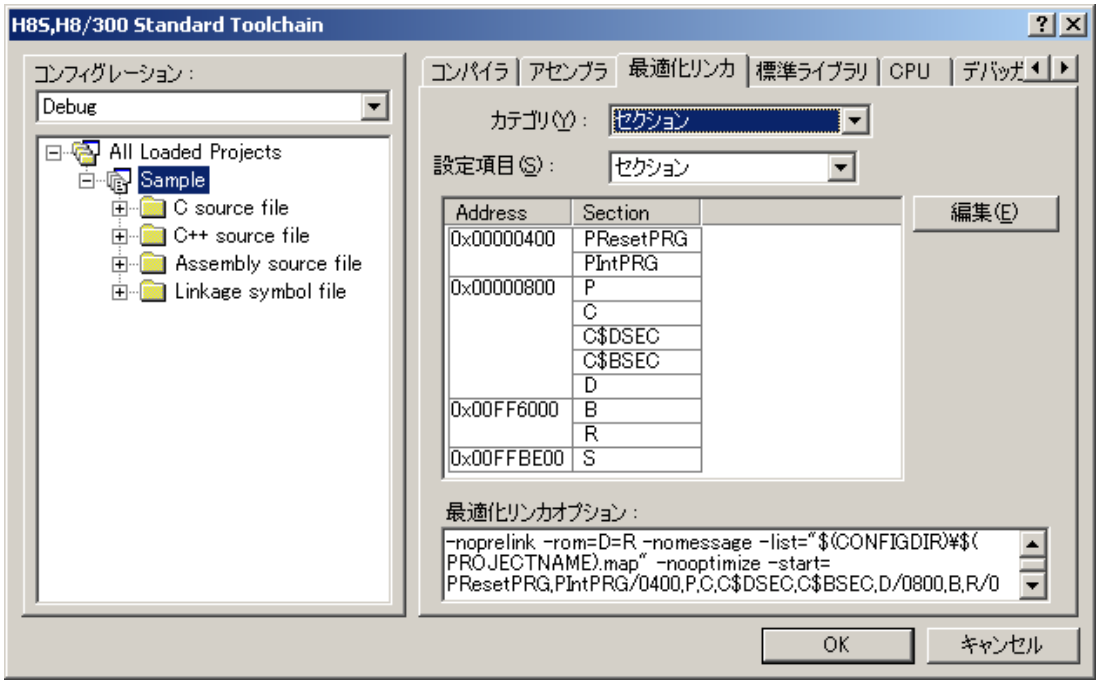
プロファイル情報

チェックボックス	コマンドオプション	機能
☑	*profile*	**プロファイル情報ファイルの指定**
☐	-	**プロファイル情報ファイルの指定なし**

キャッシュサイズ:

ダイアログメニュー	コマンドオプション	機能
サイズ	*cachesize =sized*	**キャッシュサイズの指定**
ライン	*cachesize =align*	**キャッシュラインサイズの指定**

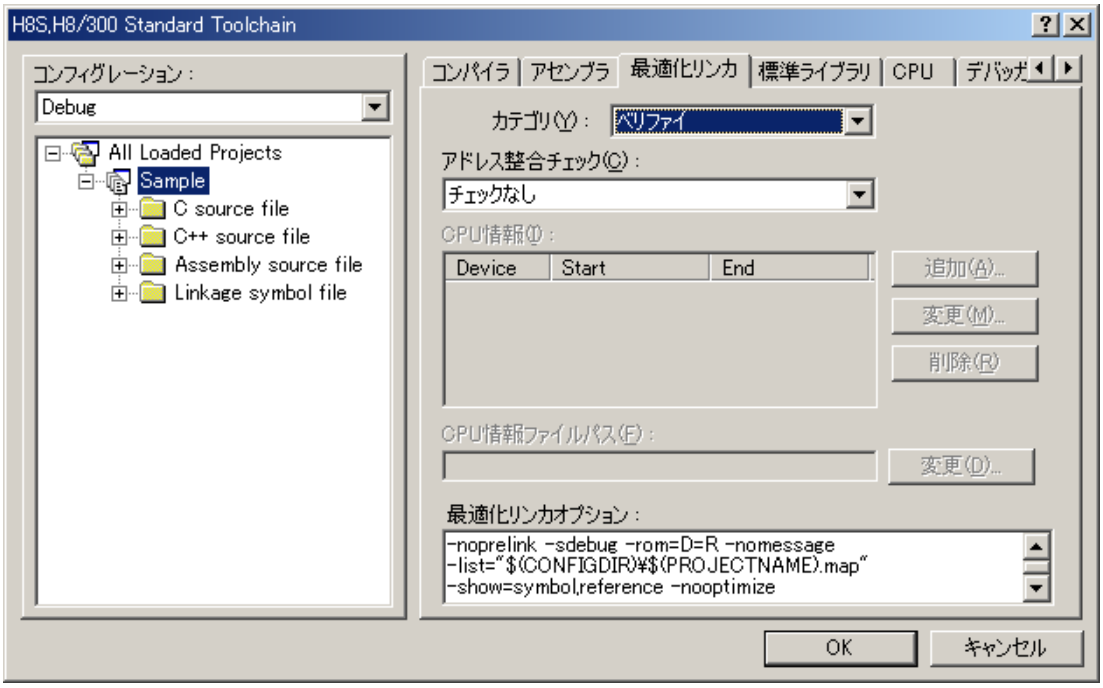
(5) **カテゴリ**:[**セクション**]



設定項目:

ダイアログメニュー	コマンドオプション	機能
セクション	*start-*	**各セクションの先頭アドレスや結合順序を指定**
シンボルアドレスファイル	*fsymbol*	**リンケージ機能で解決した外部定義シンボルをアセンブラ制御命令形式でファイルに出力**

(6) **カテゴリ**:[**ベリファイ**]



アドレス整合チェック:

ダイアログメニュー	コマンドオプション	機能
チェックなし	-	CPU 割り付けチェックなし
チェック	*CPU*	CPU 情報ファイルによるメモリ割り付けチェック
CPU 情報ファイルを指定してチェック	*CPU*	既存 CPU 情報ファイルによるメモリ割り付けチェック

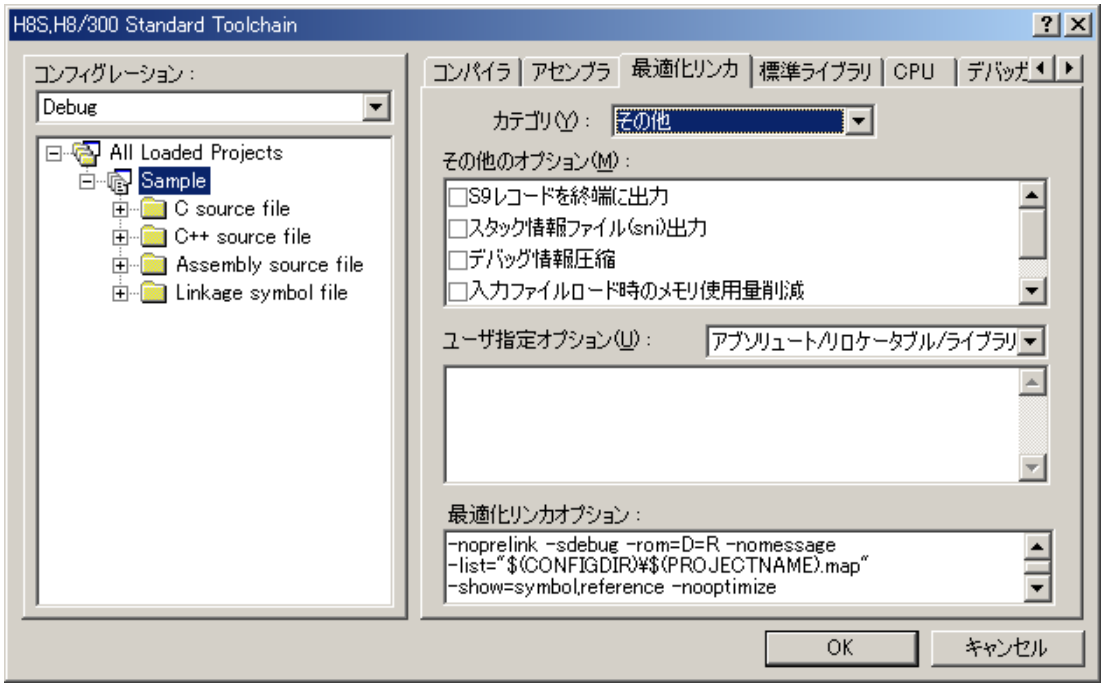
CPU 情報

ダイアログメニュー	サブコマンド	機能
-	*{ROm | RAm}=<アドレス範囲>*	CPU 情報ファイルを作成／変更 メモリ種別を指定し、それぞれのメモリのアドレスを指定

CPU 情報ファイルパス

ダイアログメニュー	サブコマンド	機能
-	<ファイル名>	既存の CPU 情報ファイルを指定

(7) **カテゴリ**:[**その他**]

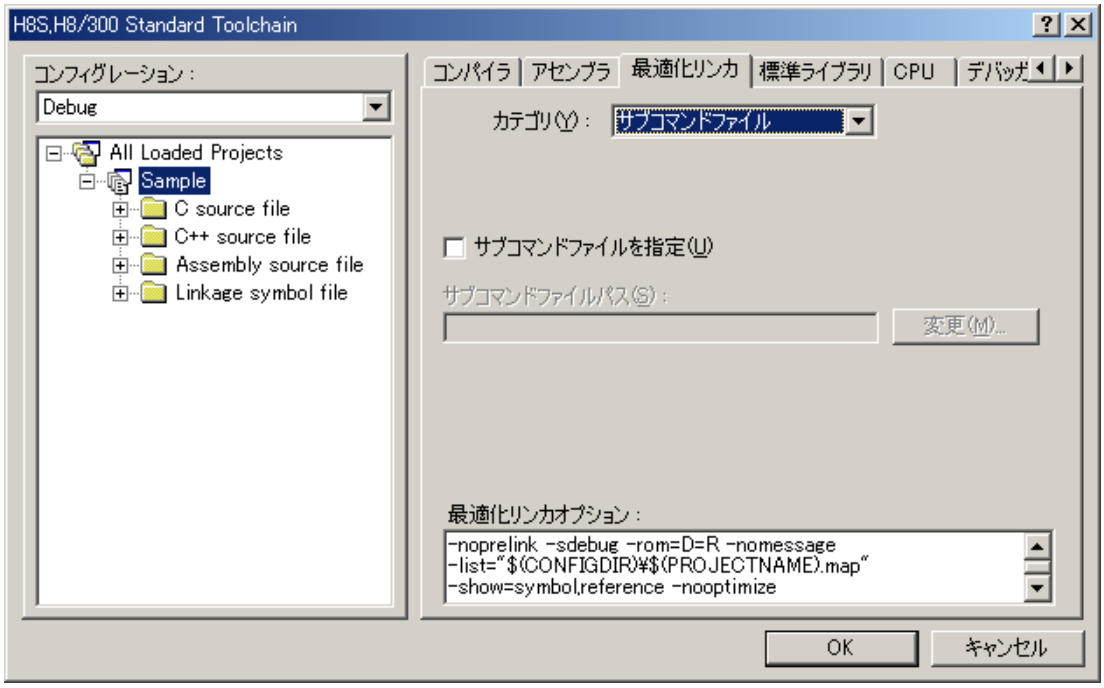


その他のオプション:その他の機能を指定します。

ダイアログメニュー	コマンドオプション	機能
S9 レコードを終端に出力	*S9*	S9 レコードを常に出力
スタック情報ファイル(sni)出力	*stack*	**スタック使用量情報ファイルの出力**
デバッグ情報圧縮	*compress*	**デバック情報の圧縮あり**
	nocompress	**デバック情報の圧縮なし**
入力ファイルロード時のメモリ使用量削減	*Memory=high*	**従来どおりのメモリ使用量**
	Memory=low	**メモリ使用量を削減**

ユーザ指定オプション:ではコマンドオプションを指定することができます。

(8) **カテゴリ**:[**サブコマンドファイル**]



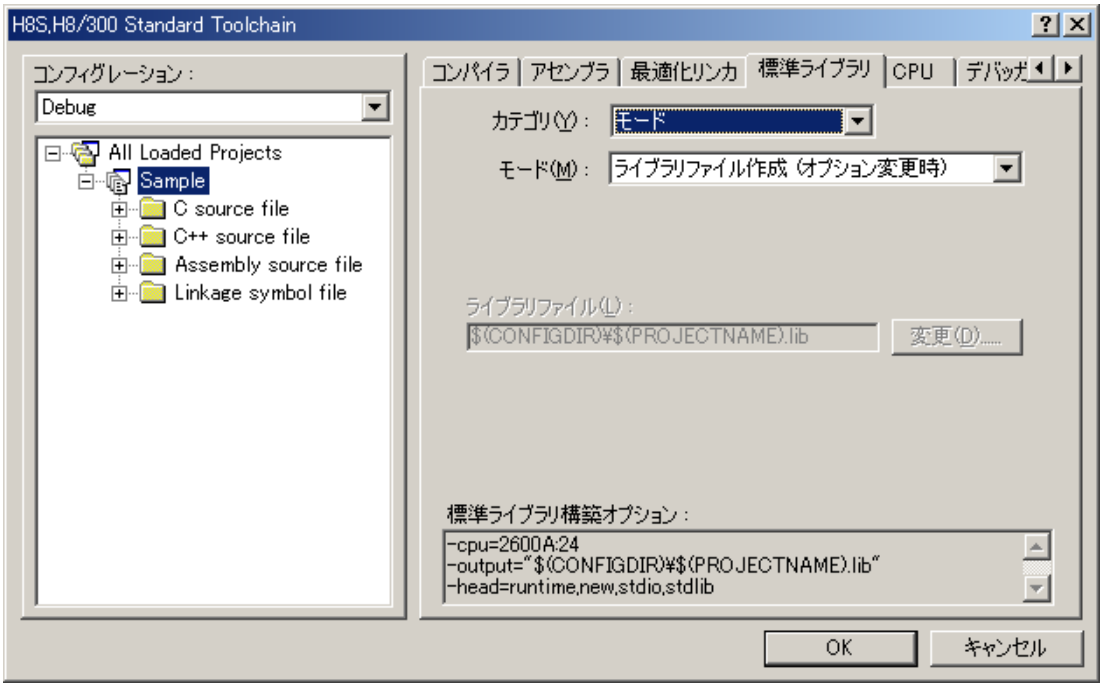
サブコマンドファイルを指定

チェックボックス	コマンドオプション	機能
レ	*SUbcommand*	**オプションをサブコマンドファイルで指定**
	-	**サブコマンドファイル指定なし**

4.2.4 標準ライブラリ構築ツールのオプション

H8S,H8/300 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。

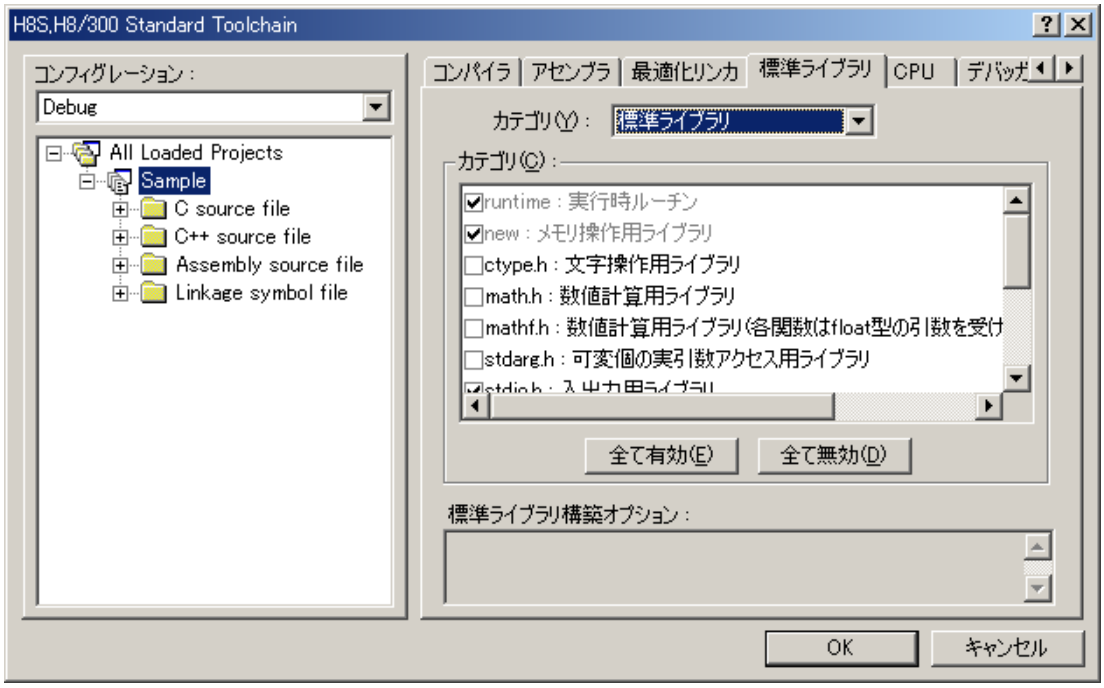
(1) **カテゴリ**:[**モード**]



モード:

ダイアログメニュー	コマンドオプション	機能
ライブラリファイル作成 (常に)	-	**最新の標準ライブラリ作成**
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**
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**
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**

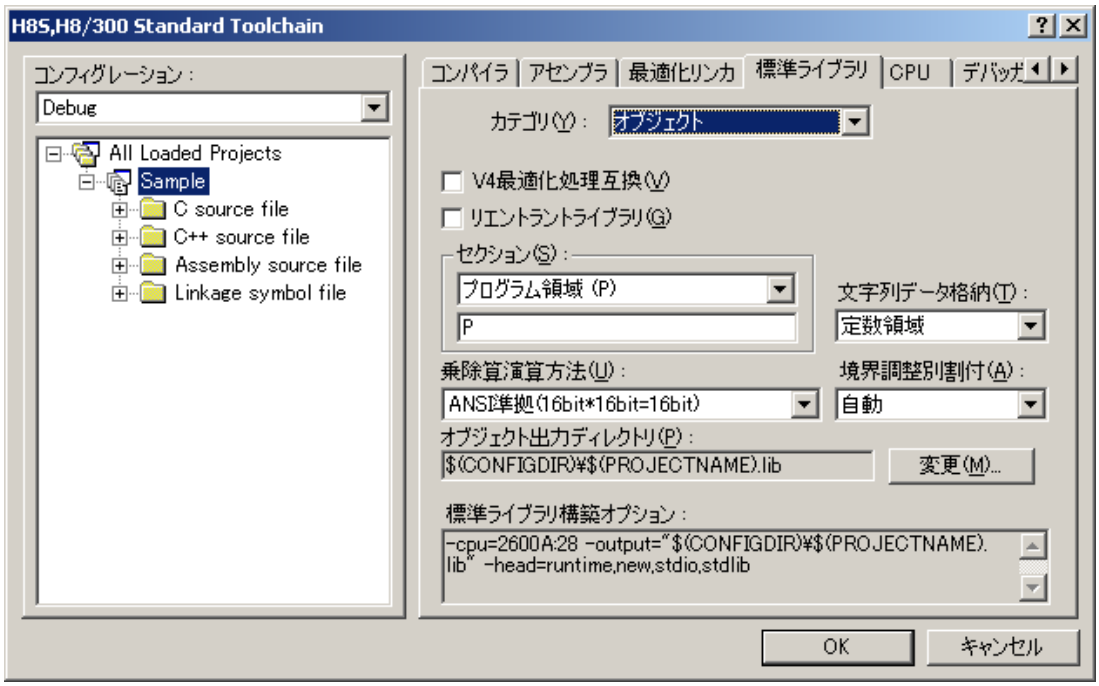
(2) **カテゴリ**:[**標準ライブラリ**]



カテゴリ:

ダイアログメニュー	コマンドオプション	機能
runtime	*Head=RUNTIME*	**実行時ルーチンを指定**
new	*Head=NEW*	new **で宣言されている** EC++**を指定**
ctype.h	*Head=CTYPE*	ctype.h **を指定**
math.h	*Head=MATH*	math.h **を指定**
mathf.h	*Head=MATHF*	mathf.h **を指定**
stdarg.h	*Head=STDARG*	stdarg.h **を指定**
stdio.h	*Head=STDIO*	stdio.h **を指定**
stdlib.h	*Head=STDLIB*	stdlib.h **を指定**
string.h	*Head=STRING*	string.h **を指定**
ios(EC++)	*Head=IOS*	ios(EC++)**を指定**
complex(EC++)	*Head=COMPREX*	complex(EC++)**を指定**
string(EC++)	*Head=CPPSTRING*	string(EC++)**を指定**

(3) **カテゴリ**:[**オブジェクト**]



V4 最適化処理互換（HEW4.0 からサポート）

チェックボックス	コマンドオプション	機能
☑	*legacy=v4*	H8S の Ver4.0 と同様の最適化処理でオブジェクト出力
☐	-	H8S の最適化を V6.1 の最適化処理でオブジェクト出力

リエントラントライブラリ

チェックボックス	コマンドオプション	機能
☑	*reent*	リエントラントライブラリを生成
☐	-	リエントラントライブラリを生成しない

セクション:

ダイアログメニュー	コマンドオプション	機能
-	*section*	デフォルトのセクション名を変更

文字列データ格納:

ダイアログメニュー	コマンドオプション	機能
定数領域	*string=const*	文字列を定数領域へ出力
初期化データ領域	*string=data*	文字列を初期化データ領域へ出力

乗除算演算方法

ダイアログメニュー	コマンドオプション	機能
ANSI 準拠(16bit*16bit=16bit)	*nocpuexpand*	乗除算を ANSI C 言語仕様準拠でコード展開
ANSI 非準拠(16bit*16bit=32bit)	*cpuexpand*	乗除算を CPU 命令仕様にあわせてコードを展開

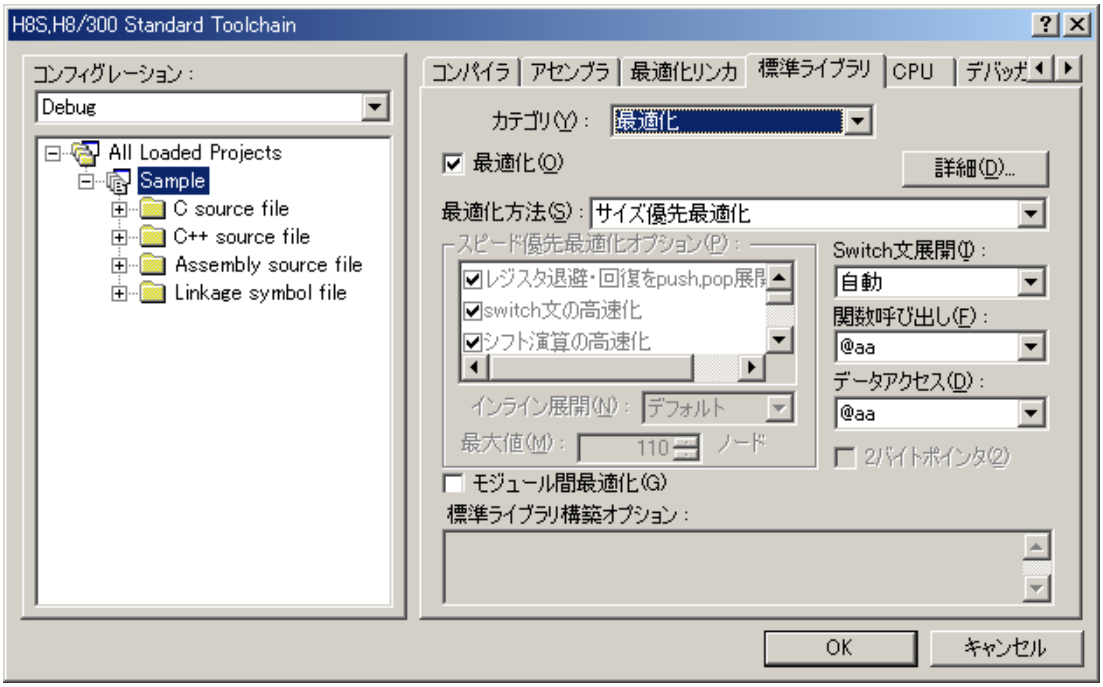
オブジェクト出力ディレクトリ

ダイアログメニュー	コマンドオプション	機能
-	*output*	ライブラリファイルの出力先ディレクトリを指定

境界調整:

ダイアログメニュー	コマンドオプション	機能
なし	*noalign*	**変数を宣言順に配置**
自動	*align*	**変数を境界調整数による空き領域が最小になるように配置**
4byte	*align=4*	align **の機能に加え、データの境界調整数が** 4,2,1 **のセクション分割し、それぞれの倍数のアドレスに配置します。それにより実行速度が向上します。**

(4) **カテゴリ**:[**最適化**]



最適化

チェックボックス	コマンドオプション	機能
☑	*optimize=1*	**最適化指定**
☐	*optimize=0*	**最適化指定なし**

最適化方法:最適化の方式を選択します。

ダイアログメニュー		コマンドオプション	機能
サイズ優先最適化		-	**サイズ優先の最適化**
スピード優先最適化		*speed*	**スピード優先の最適化**
スピード優先最適化サブオプション	**レジスタ退避・回復を push,pop 展開**	*speed=register*	**レジスタ退避／回復** push/pop **展開**
	swith 文の高速化	*speed=switch*	switch **文の高速化**
	シフト演算の高速化	*speed=shift*	**シフト演算の高速化**
	構造体代入式の高速化	*speed=struct*	**構造体代入式の高速化**
	四則演算、比較、代入式の高速化	*speed=expression*	**四則演算、比較、代入式の高速化**
	ループ文での帰納変数の削除	*speed=loop1*	**帰納変数の削除**
	ループ文での帰納変数削除、ループ展開	*speed=loop2*	**帰納変数の削除とループ展開**
	インライン展開	*speed=inline [=<**数値**>]*	**自動インライン展開の有無**

モジュール間最適化

チェックボックス	コマンドオプション	機能
☑	*goptimize*	**モジュール間最適化用付加情報出力**
☐	-	**モジュール間最適化用付加情報出力なし**

Switch 文展開: switch 文の展開方式を選択します。

ダイアログメニュー	コマンドオプション	機能
自動	*case=auto*	speed **オプション指定有無で判定**
If then 方式	*case=ifthen*	if_then **方式で展開**
テーブルジャンプ方式	*case=table*	**テーブルジャンプ方式で展開**

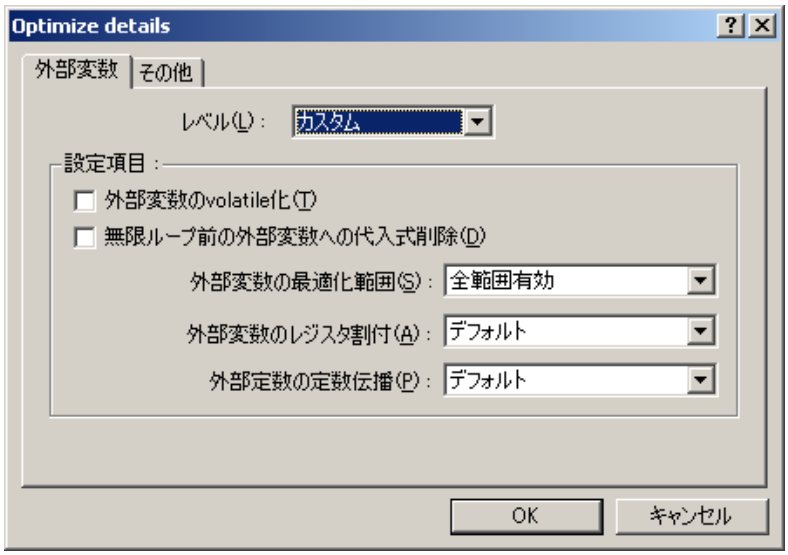
関数呼び出し:では関数の呼び出し形式を選択します。

ダイアログメニュー	コマンドオプション	機能
@aa	-	**通常の関数呼び出し**
@@aa:8	*Indirect=normal*	**メモリ間接形式の関数呼び出し**
@@vec:7	*Indirect=extended*	**拡張メモリ間接形式の関数呼び出し**

データアクセス:データのアクセス方法を選択します。

ダイアログメニュー	コマンドオプション	機能
@aa	-	**通常のデータアクセス**
@aa:8	*abs8*	8 **ビット絶対アドレスアクセス**
@aa:16	*abs16*	16 **ビット絶対アドレスアクセス**

(a) **[詳細]ボタン**:**[外部変数]**



レベル:外部変数に対する最適化のレベルを設定します。

ダイアログメニュー	コマンドオプション	機能
レベル 1		**外部変数最適化のすべてを抑止**
	volatile	***[外部変数の最適化]=[チェックあり]***
	infinite_loop=0	***[無限ループ前の式削除]=[チェックなし]***
	opt_range=noblock	***[外部変数の最適化範囲]=[最適化抑止]***
	global_alloc=0	***[外部変数のレジスタ割り付け]=[抑止]***
	const_var_propagate=0	***[外部変数の定数伝播]=[抑止]***
レベル 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**
	novolatile	***[外部変数の最適化]=[チェックなし]***
	infinite_loop=0	***[無限ループ前の式削除]=[チェックなし]***
	opt_range=noblock	***[外部変数の最適化範囲]=[最適化抑止]***
	global_alloc=0	***[外部変数のレジスタ割り付け]=[抑止]***
	const_var_propagate=0	***[外部変数の定数伝播]=[抑止]***
レベル 3		volatile **指定のない外部変数をすべて最適化対象**
	novolatile	***[外部変数の最適化]=[チェックなし]***
	infinite_loop=0	***[無限ループ前の式削除]=[チェックなし]***
	opt_range=all	***[外部変数の最適化範囲]=[最適化]***
	global_alloc=1	***[外部変数のレジスタ割り付け]=[割り付け]***
	const_var_propagate=1	***[外部変数の定数伝播]=[定数伝播]***
カスタム	-	**外部変数に対する最適化をユーザが設定**

外部変数の最適化

チェックボックス	コマンドオプション	機能
☑	*volatile*	**外部変数の最適化をすべて抑止**
☐	*novolatile*	volatile **修飾子のない外部変数を最適化する**

無限ループ前の式削除

チェックボックス	コマンドオプション	機能
☑	*infinite_loop=1*	**無限ループ直前の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する**
☐	*infinite_loop=0*	**無限ループ直前での外部変数への代入を削除しない**

外部変数の最適化範囲

ダイアログメニュー	コマンドオプション	機能
最適化	*opt_range=all*	**関数内の全範囲を対象に外部変数の最適化を行う**
移動禁止	*opt_range=noloop*	**ループ内にある外部変数やループ内判定式で使用されている外部変数を最適化の対象外にする**
最適化抑止	*opt_range=noblock*	**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**

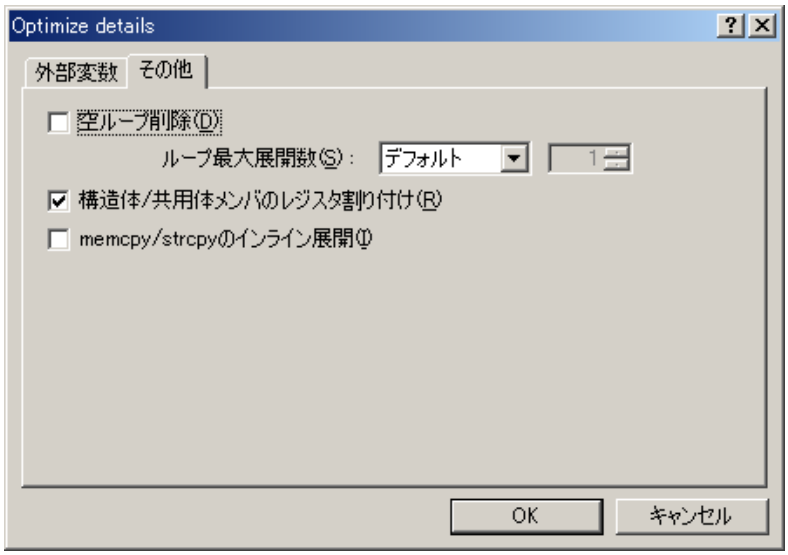
外部変数のレジスタ割り付け

ダイアログメニュー	コマンドオプション	機能
抑止	*global_alloc=0*	**外部変数のレジスタ割り付けを抑止**
割り付け	*global_alloc=1*	**外部変数のレジスタ割り付けを行う**
デフォルト	*global_alloc=1*	**外部変数のレジスタ割り付けを行う**

外部定数の定数伝播

ダイアログメニュー	コマンドオプション	機能
抑止	*const_var_propagate=0*	const **宣言された外部変数の定数伝播を抑止**
定数伝播	*const_var_propagate=1*	const **宣言された外部変数の定数伝播を行う**
デフォルト	*const_var_propagate=1*	const **宣言された外部変数の定数伝播を行う**

(b) **[詳細]ボタン:[その他]**



空ループ削除

チェックボックス	コマンドオプション	機能
☑	*del_vacant_loop=1*	**ループ内に処理がない場合、ループを削除**
☐	*del_vacant_loop=0*	**ループ内に処理がない場合、ループを削除しない**

ループ展開最大数

ダイアログメニュー	コマンドオプション	機能
デフォルト	*max_unroll=2 or 1*	**ループ展開数の最大数が** 2 **または** 1
カスタム	*max_unroll=<**数値**>*	**ループ展開数の最大数が** 1 **から** 32 **のユーザ任意の数値**

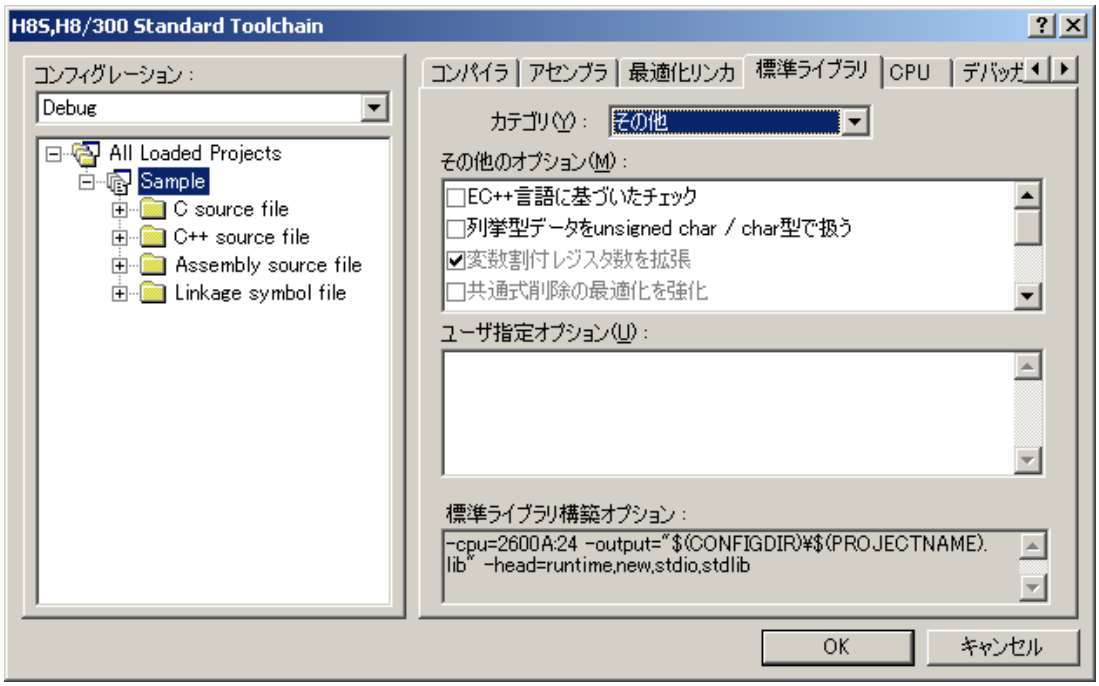
構造体／共用体メンバのレジスタ割り付け

チェックボックス	コマンドオプション	機能
☑	*struct_alloc=1*	**構造体／共用体メンバのレジスタ割り付けを行う**
☐	*struct_alloc=0*	**構造体／共用体メンバのレジスタ割り付けを抑止**

memcpy/strcpy のインライン展開

チェックボックス	コマンドオプション	機能
☑	*library=intrinsic*	memcpy/strcpy **をインライン展開**
☐	*library=function*	memcpy/strcpy **を関数呼び出し**

(5) **カテゴリ**:[**その他**]



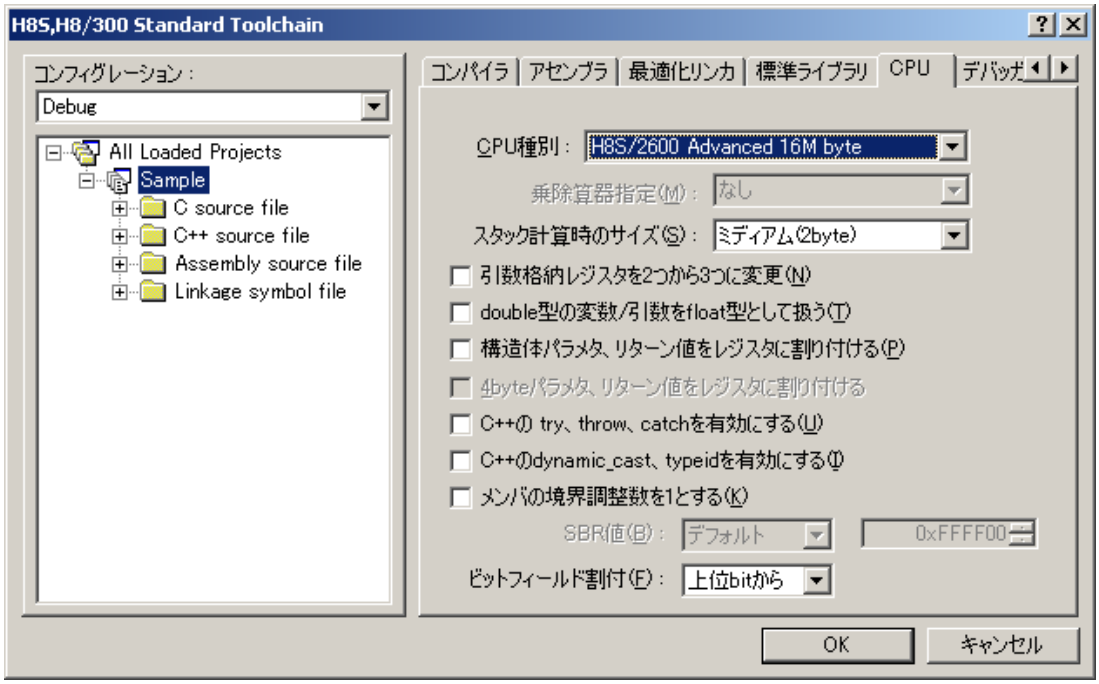
その他のオプション:

ダイアログメニュー	コマンドオプション	機能
EC++言語に基づいたチェック	*ecpp*	EC++言語仕様に基づいてシンタックスチェック
ループ判定式の最適化抑止	*volatile_loop*	**ループ判定式の最適化を抑止する**
列挙型データを unsigned char/char 型で扱う	*Byteenum*	**列挙型のデータを** char **扱い**
変数割り付けレジスタ数を拡張	*regexpansion* *noregexpansion*	**変数割り付けレジスタ数** 2 **変数割り付けレジスタ数** 3
共通式削除の最適化を強化	*cmncode*	**共通式削除の最適化強化**
構造体の代入式を eepmov 命令で展開	*eepmov*	**構造体の代入を** eepmov **命令で展開**
ループ判定式の最適化抑止	*volatile_loop*	**ループ判定式の最適化を抑止**
register 指定変数の優先レジスタ割り付け	*enable_register*	register **記憶クラスを指定した変数を優先的にレジスタ割り付け**
ANSI 準拠対応拡張	*strict_ansi*	**以下を** ANSI **準拠で行う** **・浮動小数点演算の結合則**

ユーザ指定オプション:ではコマンドオプションを指定することができます。

4.2.5 CPU オプション

H8S,H8/300 Standard Toolchain ダイアログボックスから CPU タブを選択します。



CPU 種別では、CPU 種別を指定します。

CPU	内容
Environmento variable	**環境変数** H38CPU **依存**
H8SX Maximum 4G byte	*cpu=h8sxx:32*
H8SX Maximum 256M byte	*cpu=h8sxx:28*
H8SX Advanced 4G byte	*cpu=h8sxa:32*
H8SX Advanced 256M byte	*cpu=h8sxa:28*
H8SX Advanced 16M byte	*cpu=h8sxa:24*
H8SX Advanced 1M byte	*cpu=h8sxa:20*
H8SX Middle 16M byte	*cpu=h8sxm:24*
H8SX Middle 1M byte	*cpu=h8sxm:20*
H8SX Normal	*cpu=h8sxn*
H8S/2600 Advanced 4G byte	*cpu=2600A:32*
H8S/2600 Advanced 256M byte	*cpu=2600A:28*
H8S/2600 Advanced 16M byte	*cpu=2600A:24*
H8S/2600 Advanced 1M byte	*cpu=2600A:20*
H8S/2600 Normal	*cpu=2600N*
H8S/2000 Advanced 4G byte	*cpu=2000A:32*
H8S/2000 Advanced 256M byte	*cpu=2000A:28*
H8S/2000 Advanced 16M byte	*cpu=2000A:24*
H8S/2000 Advanced 1M byte	*cpu=2000A:20*
H8S/2000 Normal	*cpu=2000N*
H8/300H Advanced 16M byte	*cpu=300HA:24*
H8/300H Advanced 1M byte	*cpu=300HA:20*
H8/300H Normal	*cpu=300HN*
H8/300	*cpu=300*
H8/300L	*cpu=300l*

乗除算器指定:

ダイアログメニュー	コマンドオプション	機能
なし	*cpu=[...][][]*	**乗除算器使用なし**
乗除算器あり	*cpu=[...][][MD]*	**乗除算器使用あり**
乗算器あり	*cpu=[...][][M]*	**乗算器使用あり**
除算器あり	*cpu=[...][][D]*	**除算器使用あり**

スタック計算時のサイズ:

ダイアログメニュー	コマンドオプション	機能
スモール(1byte)	*STAck=Small*	**スタックアドレス計算を** 1byte **で行う**
ミディアム(2byte)	*STAck=Medium*	**スタックアドレス計算を** 2byte **で行う**
ラージ(4byte)	*STAck=Large*	**スタックアドレス計算を** 4byte **で行う**

引数格納レジスタを 2 つから 3 つに変更

チェックボックス	コマンドオプション	機能
☑	*regparam=3*	**引数渡し用レジスタ数** 3 **つ**
☐	*regparam=2*	**引数渡し用レジスタ数** 2 **つ**

Double 型の変数/引数を float 型として扱う

チェックボックス	コマンドオプション	機能
☑	*DOuble=Float*	double **型の変数／数値を** float **型として扱う**
☐	-	-

構造体パラメータ、リターン値をレジスタに割り付ける

チェックボックス	コマンドオプション	機能
☑	*STRUctreg*	**構造体のパラメータをレジスタに割り付ける**
☐	*NOSTRUctreg*	**構造体のパラメータをレジスタに割り付けない**

4byte パラメータ、リターン値をレジスタに割り付ける

チェックボックス	コマンドオプション	機能
☑	*LONgreg*	4byte **のパラメータやリターン値をレジスタに割り付ける**
☐	*NOLONgreg*	4byte **のパラメータやリターン値をレジスタに割り付けない**

C++の try、throw、catch を有効にする

チェックボックス	コマンドオプション	機能
☑	*EXception*	**例外処理機能有効**
☐	*NOEXception-*	**例外処理機能無効**

C++の dynamic_cast、typeid を有効にする

チェックボックス	コマンドオプション	機能
☑	*RTti=ON*	dynamic_cast 、typeid **を有効**
☐	*RTti=OFF*	dynamic_cast 、typeid **を無効**

データの境界調整数を 1 とする

チェックボックス	コマンドオプション	機能
☑	*PAck=1*	**構造体、共用体、クラスメンバの境界調整を** 1 **にする**
☐	*PAck=2*	**データの境界調整に従う**

8bit 絶対領域指定

チェックボックス	コマンドオプション	機能
デフォルト	-	**デフォルトの** 8bit **絶対領域の開始アドレスを指定**
カスタム	*sbr=<**アドレス**>*	**ユーザ任意の** 8bit **絶対領域の開始アドレスを指定**

ビットフィールド割り付け

ダイアログメニュー	コマンドオプション	機能
上位 bit から	*bit_order=left*	**メンバを上位** bit **から格納**
下位 bit から	*bit_order=right*	**メンバを下位** bit **から格納**

4.3 既存ファイルを HEW でビルドする方法

HIM プロジェクト以外ですでに用意してある、一連のロードモジュール作成手順を HEW のプロジェクトファイルとして登録する手順を説明します。

HEW1.2 では、HEW ディレクトリ¥Tools¥HITACHI¥H8¥3_0a_0¥sample にサンプルプログラムが用意されています。

No.	HEW1.2 ファイル	ファイルの内容
1	init.c	初期化ルーチン
2	vectbl.c	ベクタテーブル設定
3	scttbl.c	セクション初期化ルーチン
4	cmain.c	メイン関数ファイル
5	c2600a.sub	モジュール間最適化ツール用サブコマンドファイル

HEW2.0 以降では、サンプルプログラムは用意されていませんので、自作にてご用意していただくか、

サンプルプロジェクトを作成してそのとき生成される次のファイルをサンプルプログラムとして使用してください。

サンプルプロジェクトの作成方法は、**「**2.1.2 **ワークスペースの新規作成** 2（HEW2.0 **以降）」**に従い Project type の設定を Demonstration に選択してプロジェクトの作成を行ってください。

No.	HEW2.0 以降のファイル	ファイルの内容
1	resetprg.c	初期化ルーチン
2	intprg.c	ベクタテーブル設定
3	dbsct.c	セクション初期化ルーチン
4	main.c	メイン関数ファイル
5	2600a.sub(**自作**)	サブコマンドファイル

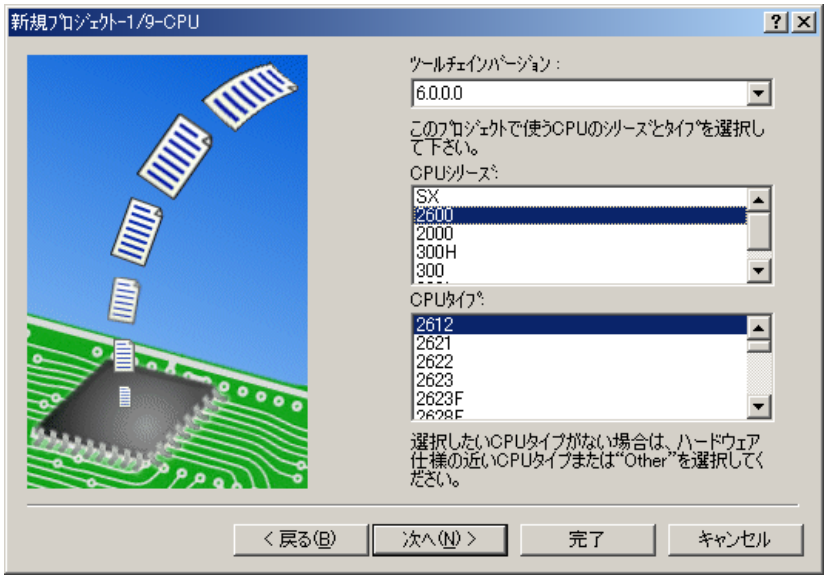
(1) 新規プロジェクトを作成する

「2.1.1 **ワークスペースの新規作成** 2（HEW2.0 **以降）」**に従い、新規プロジェクトを作成します。

プロジェクトタイプは Empty Application を選択します。

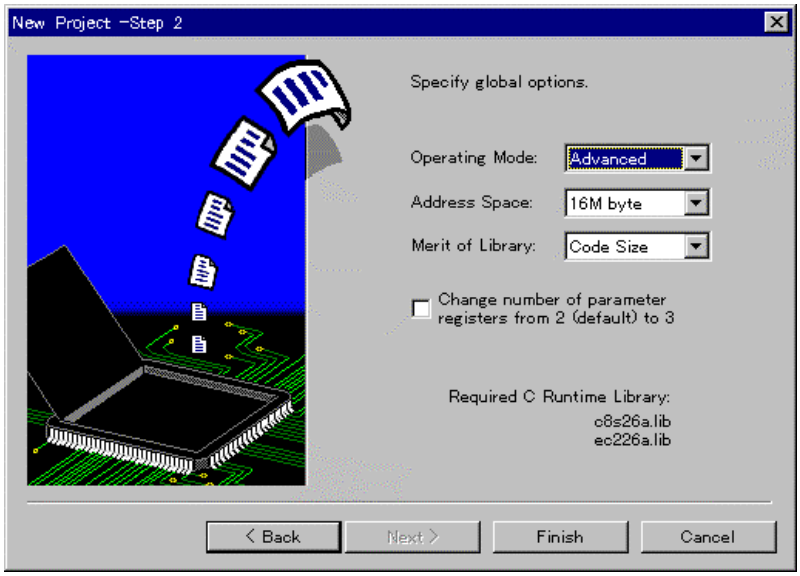
(2) CPU を選択する

Step1 の画面で CPU 種別を選択します。

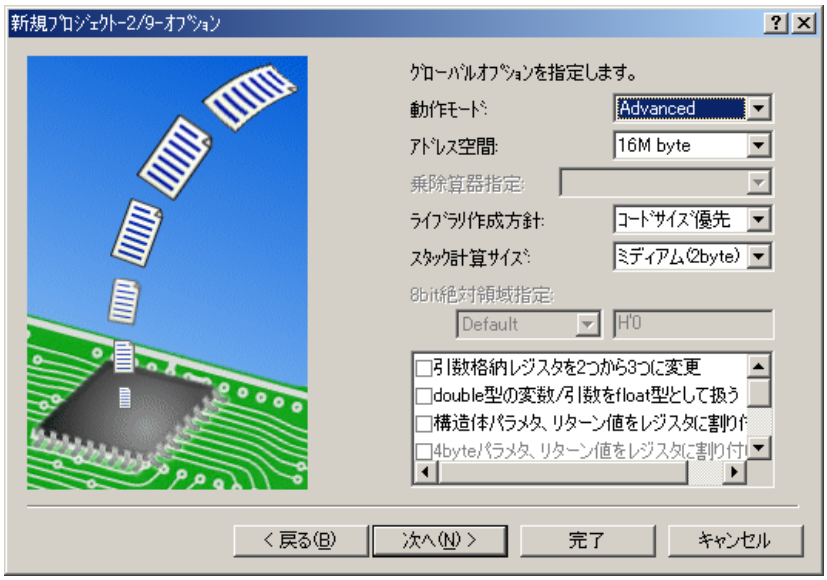


(3) グローバルオプションを選択する

Step2 の画面でグローバルオプションを選択します。
＜HEW1.2＞



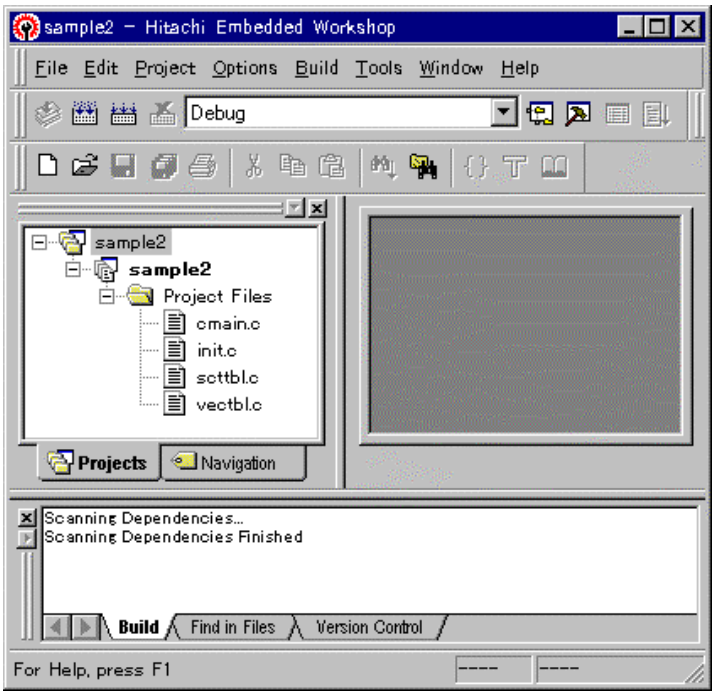
＜HEW2.0 以降＞



初期設定後にグローバルオプションを変更したい場合は、**「11.2.1 Undefined external symbol が出力される」**を参照してください。

(4) プロジェクトにファイルを追加する

[Project->Add Files…]でプロジェクトに追加するソースファイルを指定します。
HEW1.2 では init.c、vecttbl.c、scttbl.c、cmain.c を HEW2.0 以降では resetprg.c、intprg.c、dbsct.c、main.c を組み込みます。

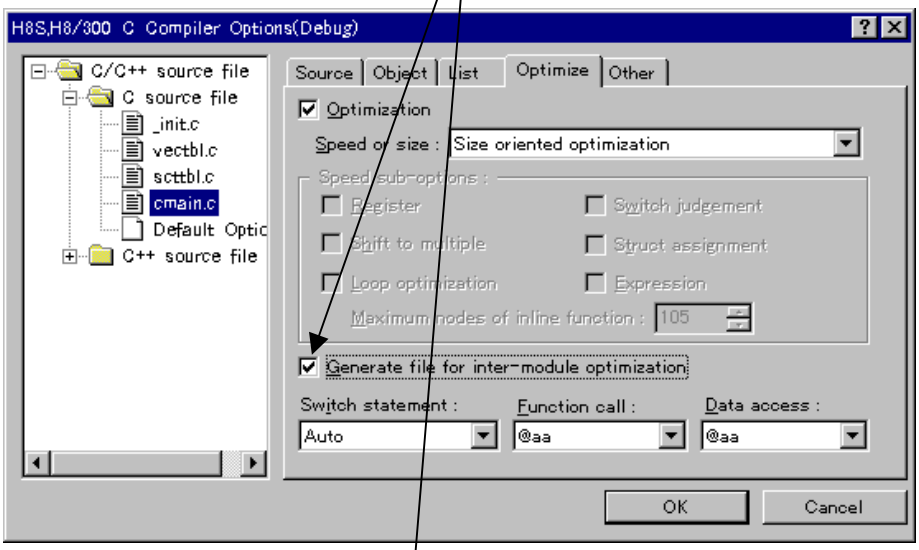


(5) コンパイルオプションを指定する

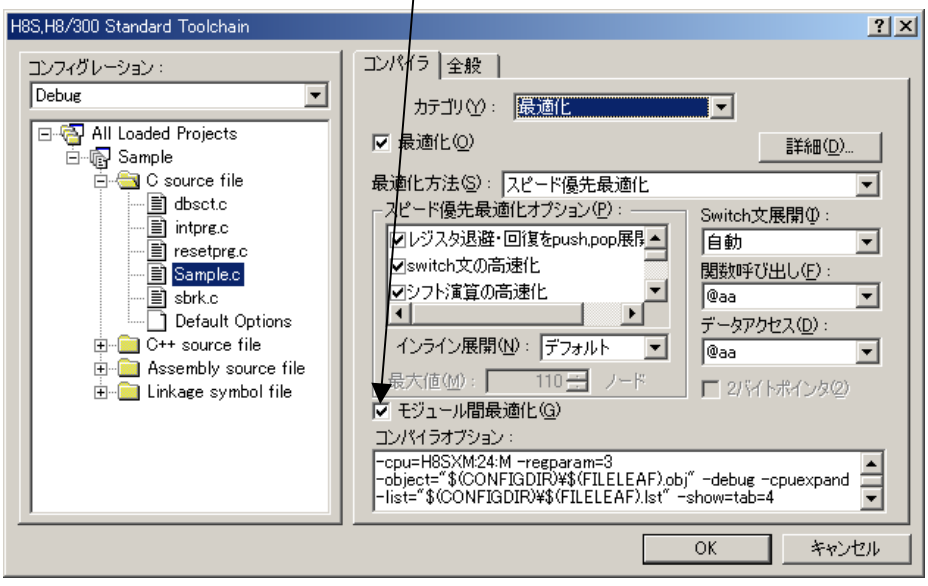
HEW1.2 では、[Options->H8S,H8/300 C/C++ Compiler…]、HEW2.0 以降では、[コンパイラタブ] [カテゴリ/最適化]でコンパイルオプションを指定します。

ここではモジュール間最適化用付加情報ツールの出力を[ここ]で指定します。

＜HEW1.2＞



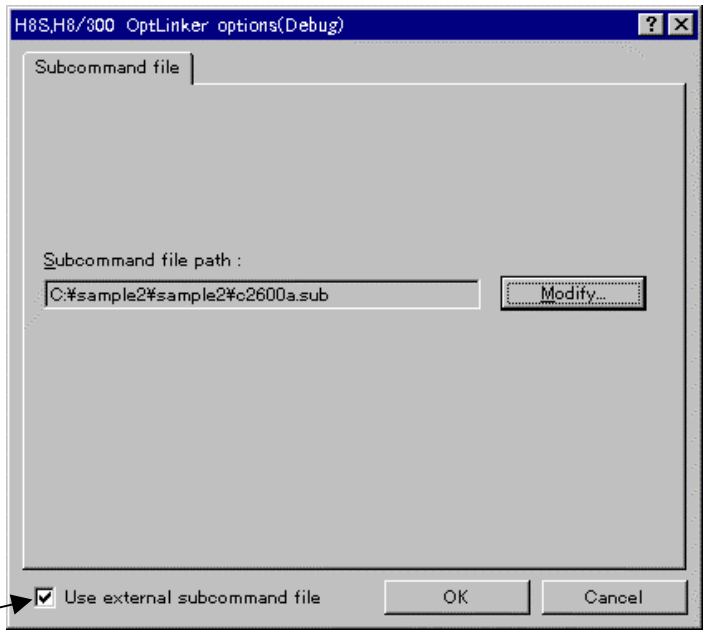
＜HEW2.0 以降＞



(6) モジュール間最適化ツール用サブコマンドファイルを指定する

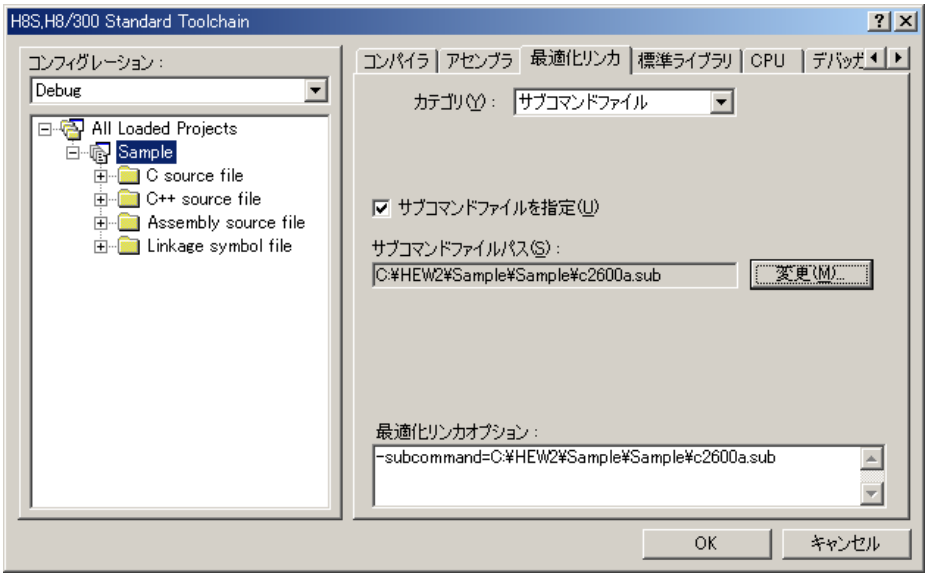
HEW1.2 では、[Options->H8S,H8/300 IM OptLinker…]を HEW2.0 以降では、[Link/Library タブ] [Category/Subcommand file]でモジュール間最適化ツール用サブコマンドファイルを HEW オプションダイアログで指定します。

＜HEW1.2＞



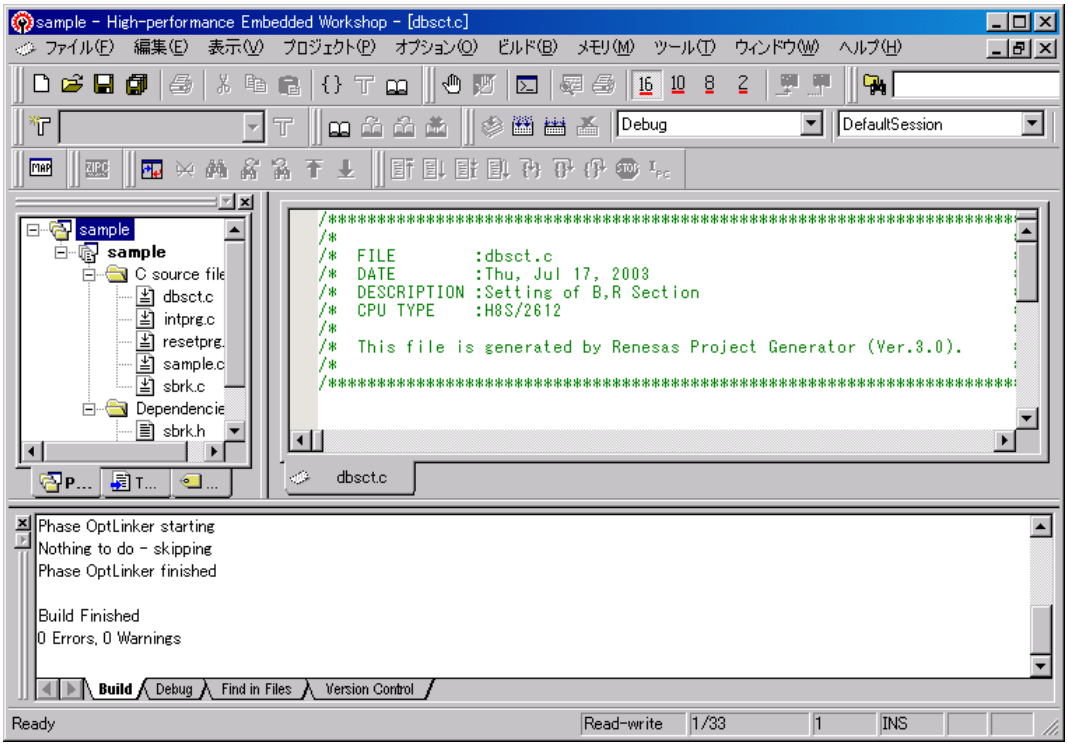
ここをチェックすると Subcommand file タブが出現します。

＜HEW2.0 以降＞



(7) ビルドの実行

ビルドを実行するとロードモジュールが作成されます。



【注】 製品に添付されているサンプルのモジュール間最適化用サブコマンドファイルを使用すると、エラーとなります。これは、標準ライブラリが指定されていないためと、CPU 情報チェックファイルが当該ディレクトリに存在しないためです。
サブコマンドファイルと、CPU 情報チェックファイルをコピーしてきて、標準ライブラリの指定をしてください。

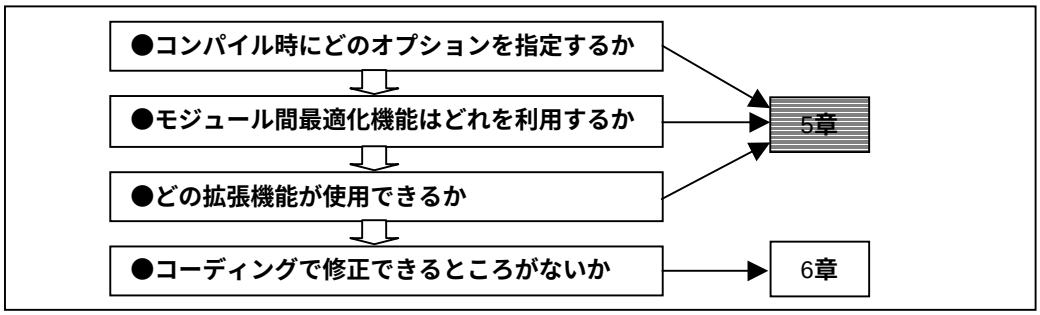
5. 最適化機能の活用

動作するロードモジュールができたら、今度はそのロードモジュールをより効率の良い、目的にあったロードモジュールに作り替えるために、オブジェクトプログラムの性能を上げる工夫をします。

オブジェクトプログラムの性能を上げるためには、次の 4 つの方法があります。

(i) オプションを利用して最適化を行う
(ii) モジュール間最適化ツールを利用して最適化を行う
(iii) 拡張機能を利用して最適化を行う
(iv) コーディングを変更して効率よいプログラミングを行う

この手順は以下のように検討します。



本章では、ロードモジュール作成時にどのオプションを指定するか、どの拡張機能が使用できるか、また、モジュール間最適化ツールのどのオプションを使用するかという説明をします。

コンパイラの最適化機能としては以下のものを用意しています。

No.	最適化機能	指定形態	サイズ	スピード
1	1 バイト enum 型の活用	オプション	○	○
2	乗除算仕様の拡張解釈	オプション	○	○
3	引数渡し用レジスタ数の指定	オプション	△	△
4	変数割り付けレジスタ数の拡張	オプション	△	△
5	外部変数の最適化	オプション	-	-
6	ブロック転送命令	オプション	×	○
7	SPEED オプション	オプション	×	○
8	グローバル変数のレジスタ割り付け	拡張機能	△	△
9	関数の入口／出口でレジスタの退避／回復コード出力を制御	拡張機能	○	○
10	関数のインライン展開を指定	拡張機能	×	○
11	アセンブリ記述関数をインライン展開	拡張機能	×	○
12	8 ビット絶対アドレス領域の活用	オプション／拡張機能	○	○
13	16 ビット絶対アドレス領域の活用	オプション／拡張機能	○	○
14	メモリ間接領域への割り付け	オプション／拡張機能	○	×

【記号説明】

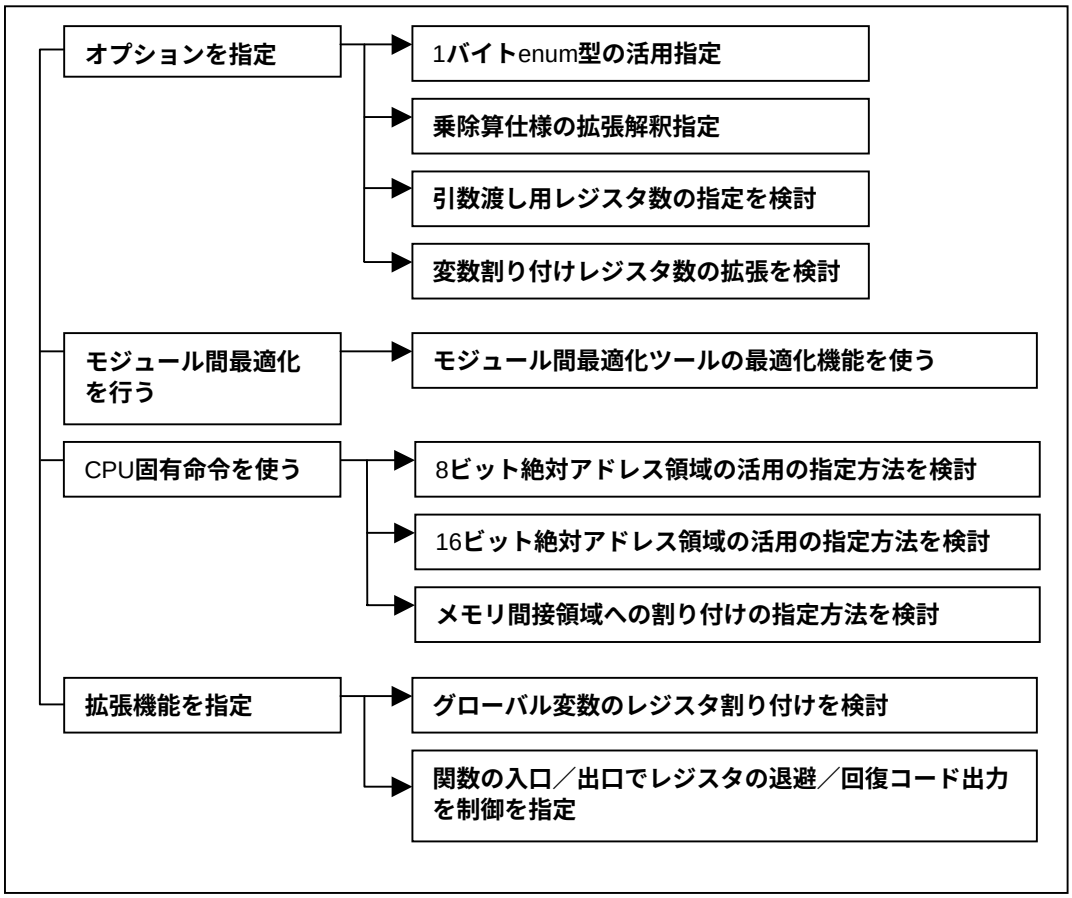
○：効果的、△：プログラムによっては効果的、×：効率が低下

モジュール間最適化ツールの最適化項目としては次のものを用意しています。

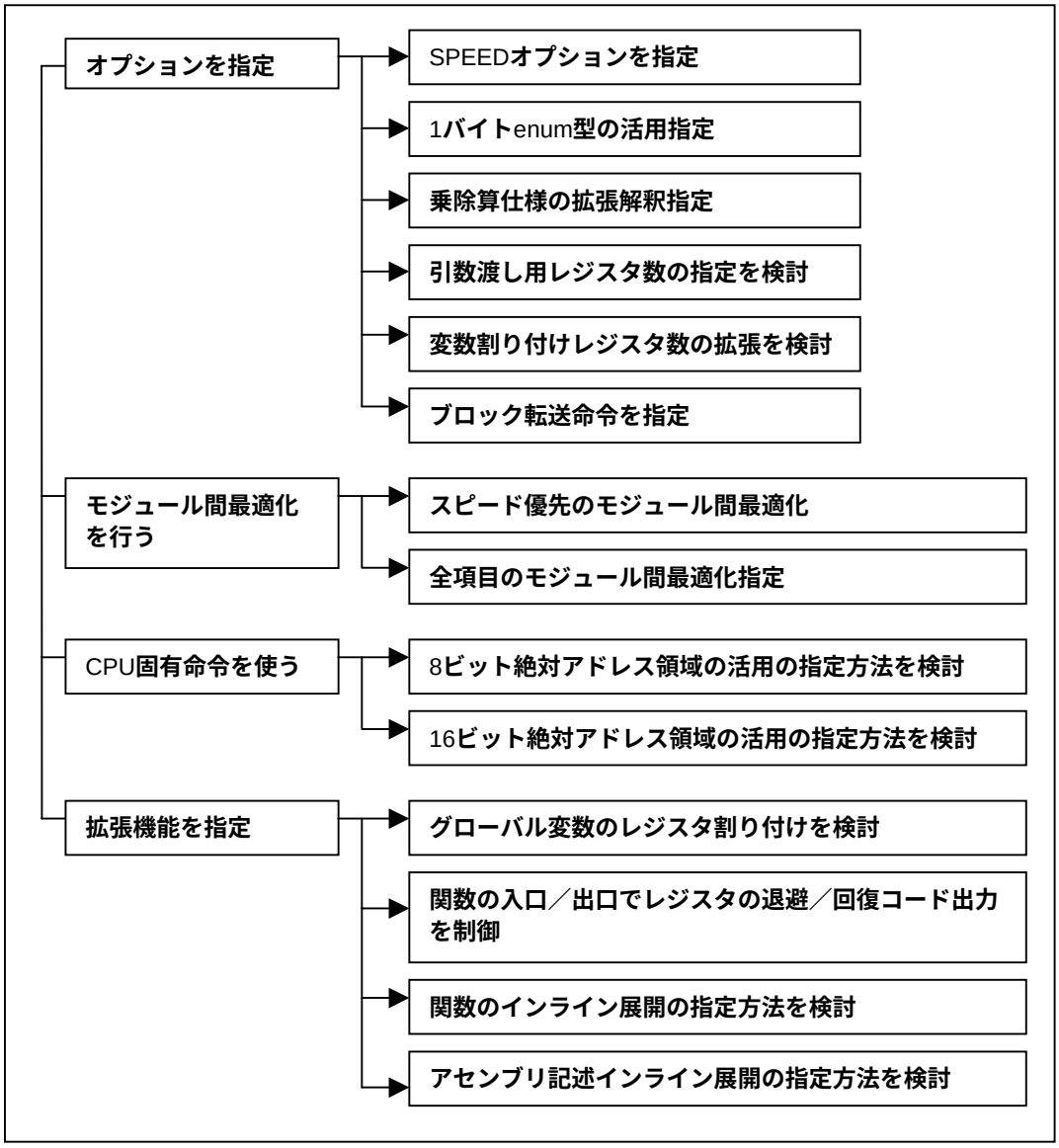
No.	内容	指定形態
1	定数／文字列の統合	オプション
2	未参照変数／関数の削除	オプション
3	変数アクセスの最適化	オプション
4	関数アクセスの最適化	オプション
5	レジスタの再割り付け	オプション
6	共通コードの統合	オプション
7	分岐命令の最適化	オプション

これらの最適化機能を利用する方法として、サイズ効率の良い最適化優先の場合と、スピード効率の良い最適化優先の場合とそれぞれに分けて検討をしています。それぞれの最適化機能の特長をみながら、次のフローで検討しています。

＜サイズ効率優先の場合の最適化機能指定検討方法＞



＜スピード効率優先の場合の最適化機能指定検討方法＞



5.1 サイズ効率のよいオブジェクトの出力方法

サンプルプログラムとして、一般的なベンチマークプログラム Dhrystone Ver.2.1 を使用して説明いたします。
ここで示したサイズ、スピードのデータは H8S/2600 アドバンストモードでコンパイルした結果です。

5.1.1 デフォルトでコンパイルする

まず、最適化オプションを何もつけない状態でコンパイルします。
オブジェクトサイズ／実行サイクル数は次のようになります。

最適化機能	サイズ（ROM）	実行サイクル数
なし(default)	3048	1580

コンパイラは、最適化オプションを何もつけない状態でも基本的な最適化を実行します。これはいくつかの最適化オプションが省略時にも有効になるからです。

5.1.2 最適化指定なしの場合

最適化指定を行わない場合の結果は次のようなオブジェクトサイズ／実行サイクル数になります。

最適化機能	サイズ（ROM）	実行サイクル数
なし(default)	3048	1580
最適化なし	3582	1713

【指定方法】
ダイアログメニュー：コンパイラタブカテゴリ:[最適化] 最適化
コマンドライン　：*optimize=0*

5.1.3 最適化オプションのチューニング

(1) 1 バイト enum 型を指定する

この指定は enum 型データのあるプログラムでなければ効果はありませんが、常に指定しておいた方がよいでしょう。

このときのオブジェクトサイズ／実行サイクルは次のようになります。

最適化機能	サイズ（ROM）	実行サイクル数
なし(default)	3048	1580
1 **バイト** enum **型指定**	3050	1580

【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] 列挙型データを char 型で扱う
コマンドライン　：*byteenum*

この指定の詳細については**「**5.4.1　1 **バイト** enum **型の活用」**を参照してください。

(2) 引数渡し用レジスタ数の指定

引数渡し用のレジスタ数を 2 つから 3 つへ増やします。この指定をすると次のようになります。

最適化機能	サイズ（ROM）	実行サイクル数
引数渡し用レジスタ 2 **つ**	3048	1580
引数渡し用レジスタ 3 **つ**	3034	1528

【指定方法】
ダイアログメニュー：　CPU タブ引数格納レジスタを 2 つから 3 つに変更
コマンドライン　：*regparam=3*

プログラム中の関数の引数の数と関係があり、レジスタに割り付く引数と、レジスタの数、レジスタの型を調べて選択します。

しかしながら、すべての引数を調べるわけにはいかない場合、オプションを指定してみて、サイズの小さい方を選択してください。

1 バイト enum 型の指定とあわせた結果は次のようになります。

最適化機能	サイズ（ROM）	実行サイクル数
デフォルト	3048	1580
1 **バイト** enum **型指定＋引数渡し用レジスタ** 3 **つ**	3034	1527

この指定の詳細については**「**5.4.3　**引数渡し用レジスタ数の指定」**を参照してください。

(3)　変数割り付けレジスタ数の拡張

デフォルトでは、コンパイラは[E]R3～[E]R6 までのレジスタをレジスタ変数用として使用します。

この指定を外した場合は、[E]R4～[E]R6 をレジスタ変数用として使用します。

この指定を比較した結果は次のようになります。

最適化機能	サイズ（ROM）	実行サイクル数
レジスタ変数[E]R3～[E]R6	3048	1580
レジスタ変数[E]R4～[E]R6	3048	1580

【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] 変数割り付けレジスタ数を拡張

コマンドライン　　：*regexpansion*

このプログラムでは結果に違いがありませんでしたが、本コンパイラでは 1 つの式文を複雑にしない限り、レジスタ変数用のレジスタが多い方がオブジェクト性能がよくなる傾向にあります。

H8S で V6.01 の場合、本オプションはサポートしていないため、性能に変化がありません。

この指定の詳細については**「**5.4.4　**変数割り付けレジスタ数の拡張」**を参照してください。

(4)　外部変数の volatile 化

外部変数の volatile 化を指定する場合と、外部変数の volatile 化を抑止する場合とで比較します。

最適化機能	サイズ（ROM）	実行サイクル数
外部変数の volatile **化抑止（**novolatile**）**	3048	1580
外部変数の volatile **化（**volatile**）**	3076	1592

【指定方法】

ダイアログメニュー：コンパイラタブカテゴリ:[最適化] 詳細ボタン...外部変数タブ**外部変数の volatile 化**

コマンドライン　　：*volatile*

しかし、外部変数は最適化してはいけない場合があります。

（例 1）

```
int a;
void f()
{
    a=1;
    a=2;
}
```

←最初の代入は削除される

（例 2）

```
volatile int a;
void f()
{
    a=1;
    a=2;
}
```

例 1 では、変数 a に対する代入が 2 回連続しているので、最適化により最初の代入に対するコードは削除されます。もし、2 つの代入の間で割り込みが発生して a の値を参照すると、結果は不正になります。

volatile オプション（外部変数の volatile 化）を指定すると、最適化を抑止して最初の代入に対するコードも生成しますので、このような問題は回避できます。しかしすべての外部変数に対する最適化を抑止してしまい、オブジェクト性能が著しく低下します。

必要な外部変数の最適化だけを抑止するために、例 2 のように割り込み関数で使用する変数や I/O レジスタなど、最適化してはいけない変数に対してソースプログラムで volatile 宣言をし、このオプションを外してコンパイルしましょう。
この指定の詳細については**「**5.4.5 **外部変数の** volatile **化」**を参照してください。

(5) 乗除算仕様の拡張解釈

乗除算のコード展開を ANSI 規格から拡張解釈すると次のようになります。

最適化機能	サイズ（ROM）	実行サイクル数
乗除算仕様の ANSI **準拠**	3048	1580
乗除算仕様の拡張解釈	3048	1580

【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] 乗除算演算方法
コマンドライン　　：*cpuexpand*

このプログラムでは結果に違いがありませんでした。
しかし、乗除算コードの拡張解釈をした場合、言語仕様で規定された値の保証範囲と仕様が異なるため、演算結果が異なる場合があります。十分確認してから指定するようにしてください。
この指定の詳細については**「**5.4.2 **乗除算仕様の拡張解釈」**を参照してください。

5.1.4 モジュール間最適化機能を使用する

次にモジュール間最適化ツールを使用して最適化を行い、さらにサイズ効率の良いオブジェクトを目指します。
モジュール間最適化ツールで最適化を指定する場合、あらかじめコンパイラやクロスアセンブラでモジュール間最適化用付加情報ファイルの出力を指定しておきます。

【指定方法】
C/C++コンパイラ
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*

クロスアセンブラ
ダイアログメニュー：アセンブラタブカテゴリ:[オブジェクト] モジュール間最適化
コマンドライン　　：*goptimize*

また、HEW1.2 ではモジュール間最適化時にリンクする標準ライブラリについてもモジュール間最適化用付加情報ファイルが用意されています。Windows 版では圧縮した形で提供するので、解凍してから使用します。
使用するライブラリ名と同じ圧縮ファイル（*.exe）をダブルクリックすると、自己解凍し、付加情報ファイルを含むディレクトリが生成されます。
このライブラリのモジュール間最適化については、H8S,H8/300 **シリーズ** C/C++**コンパイラ補足資料**に記載されているので参照してください。
HEW2.0 以降では標準ライブラリ構築ツールのモジュール間最適化機能にてライブラリを作成してください。**標準ライブラリタブカテゴリ:[最適化] モジュール間最適化**をチェックするとモジュール間最適化用付加情報が出力されます。

(1) デフォルトの最適化

モジュール間最適化ツールは次の最適化機能を持っています。

No.	内　容	ダイアログメニュー	サブコマンドオプション
1	**定数／文字列の統合**	**Unify strings　または 定数／文字列の統合**	*String_Unify*
2	**未参照変数／関数の削除**	**Eliminate dead code　または 未参照シンボルの削除**	*Symbol_delete*
3	**変数アクセスの最適化**	**Use short addressing　または 短絶対アドレッシングモード活用**	*Variable_access*
4	**関数アクセスの最適化**	**Use indirect call/jump　または 間接アドレッシングモード活用**	*Funcation_call*
5	**レジスタの再割り付け**	**Reallocate registers　または レジスタ退避・回復の最適化**	*Register*
6	**共通コードの統合**	**Eliminate same code　または 共通コードの統合**	*Same_code*
7	**分岐命令の最適化**	**Optimize branches　または 分岐命令の最適化**	*Branch*

まずコンパイラで最も効率のよい最適化を行った場合の指定は次のとおりです。

最適化機能	サイズ（ROM）	実行サイクル数
有効だったコンパイラ最適化オプション 1 **バイト** enum **型指定** + **引数渡し用レジスタ** 3 **つ**	3034	1527

HEW を使用した場合モジュール間最適化ツールで最適化のデフォルトは最適化を行いません。つまり、リンクを行ったのみの状態です。そのため、デフォルト状態だとコンパイラの最適化を行った結果と同じ結果になります。

(2) 各モジュール間最適化項目を指定する

(a) モジュール間最適化ツールの各最適化項目を順に指定していきます。

最適化機能	モジュール間最適化機能	サイズ（ROM）	実行サイクル数
コンパイラ最適化オプション指定	-	3034	1527
	定数／文字列の統合	3034	1527
	未参照変数／関数の削除	3034	1527
	変数アクセスの最適化	2970	1513
	関数アクセスの最適化	3024	1538
	レジスタの再割り付け	3018	1535
	共通コードの統合	3034	1527
	分岐命令の最適化	3034	1527

(b) すべてのモジュール間最適化機能を有効にする

すべての最適化機能を有効にします。

最適化機能	モジュール間最適化機能	サイズ（ROM）	実行サイクル数
コンパイラ最適化オプション指定	-	3034	1527
	全項目の最適化	2946	1517

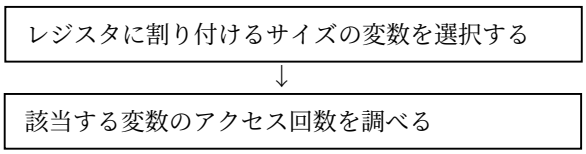
すべての最適化を行うと、抑止しなければならない部分でも、最適化がかかってしまうことがあります。list オプション(HEW1.2 では、mlist オプション)を指定すると、最適化によって削除、移動されたシンボルを出力します。最適化を抑止したいシンボルは、symbol_forbid_xxxx で指定します。対象シンボルをよく確認の上で指定をしてください。

5.1.5 拡張機能を選択する

(1) グローバル変数にレジスタを割り付ける

#pragma global_register を使用して外部変数を決められたレジスタに割り付けておくと、アクセスするときのサイズが小さくなります。

レジスタに割り付ける対象の変数は次のように選択します。



これらはリンケージエディタ（HEW1.2 ではモジュール間最適化ツール）で最適化情報リストの出力を指定しておくと、調査することができます。

【指定方法】

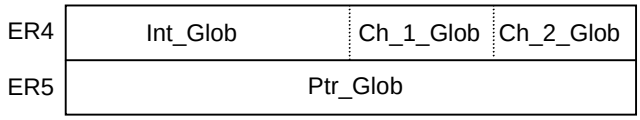
ダイアログメニュー：最適化リンカタブカテゴリ:[リスト] リンケージリスト出力
最適化リンカタブカテゴリ:[リスト] 参照回数

サブコマンド　　：*list*
show=reference

その結果、以下のようなファイルが作成されます。

```
*** Variable Accessible with Abs8 ***
SYMBOL                          SIZE    COUNTS  OPTIMIZE
_Ch_1_Glob
                                  1        4
_Ch_2_Glob
                                  1        2
*** Variable Accessible with Abs16 ***
SYMBOL                          SIZE    COUNTS  OPTIMIZE
_Ptr_Glob
                                  4        4
_Next_Ptr_Glob
                                  4        2
_Int_Glob
                                  2        6
_Bool_Glob
                                  2        2
_Arr_2_Glob
                               1388        1
_flmod
                                  3        2
_brk
                                  4        2
```

グローバルレジスタ用のレジスタは ER4 と ER5 なので、全部で 8 バイト分のデータを割り付けることができます。
この結果からアクセス回数の多い、Int_Glob と Ptr_Glob をレジスタに割り付けるとします。このレジスタに全部割り付けるとすると、さらに 2 バイト分割り付けることができます。
空いている 2 バイトに Ch_1_Glob と Ch_2_Glob を割り付けます。



以下のように指定します。

```
#pragma global_register(Int_Glob=E4,Ch_1_Glob=R4H,Ch_2_Glob=R4L,Ptr_Glob=ER5)
```

この状態をデフォルトと比較すると

最適化機能	サイズ（ROM）	実行サイクル数
なし（default）	3048	1580
#pragma global_register	2940	1512

オブジェクトサイズが小さくなり、実行速度も速くなりますが、外部変数をレジスタに割り付けるとワークレジスタの数が不足し、メモリに割り付いて逆にオブジェクト性能が低下してしまう場合があります。
プログラムによって異なるため、検討しながら使用してください。
この指定に、デフォルト以外の、サイズ効率の良かったコンパイラオプション（1 バイト enum 型指定、引数渡し用レジスタ 3 つ指定）をあわせた結果を示します。

最適化機能	サイズ（ROM）	実行サイクル数
デフォルト	3048	1580
#pragma global_register	3010	1487

この指定の詳細については**「5.4.8 グローバル変数にレジスタを割り付け」**を参照してください。
ただし、モジュール間最適化で、ライブラリをモジュール間最適化の対象にした場合は、グローバルレジスタは指定できません。そのため、ここでは指定を行わないこととします。

(2) 関数の入口／出口でレジスタの退避／回復コード出力を制御する

#pragma regsave は全レジスタの退避／回復を行う関数を宣言します。また、関数呼び出しを超えて保証するレジスタ（[E]R2～[E]R6）を割り付けないオブジェクトを生成します。
#pragma noregsave は全レジスタの退避／回復を行わない関数を宣言します。また、他の関数から呼び出されることなく、最初に起動する関数として使用し、さらに#pragma regsave を指定した関数から呼び出される関数として使用します。

これらの機能を使うためには関数の呼び出し関連図を作成します。
HEW2.0 以降では、関連図の作成に、スタック使用量情報ファイルを出力しシミュレータデバッガに情報ファイルを読み込ませることにより関数呼び出し関係をみることができます。
スタック使用量情報ファイルの出力方法は、[Options->H8S,H8/300 Standard Toolchain…->Link/Library タブ] Category:[Other] Stack information output（オプション->H8S,H8/300 Standard Toolchain…->最適化リンカタブ] Category:[その他] スタック情報ファイル(sni)出力）をチェックしてください。
Dhrystone Ver.2.1 の場合は以下のようになります。

```
main:
    malloc:
    strcpy:
    Proc_1:
       Proc_3:
          Proc_7:
       Proc_6:
          Func_3:
       Proc_7:
    Func_2:
       Func_1:
       strcmp:
    Func_1:
    Proc_8:
    Proc_7:
    Proc_6:
       Func_3:
    Proc_5:
    Proc_4:
    Func_2:
```

#pragma noregsave で宣言する関数：

main()は一番最初の処理を行う関数ですから、それ以前のレジスタを退避／回復する必要はありません。したがって#pragma noregsave で宣言します。

main()の中の処理が関数呼び出しだけの場合、main()から呼び出ししている関数はすべて#pragma noregsave で宣言することができます。

<inc.h>

```
#pragma noregsave (main)
```

<dhrystone21.c>

```
#include "inc.h"
          :
```

この結果は次のようになります。

最適化機能	サイズ（ROM）	実行サイクル数
なし（default）	3048	1580
#pragma noregsave **指定**	3030	1580

#pragma regsave/noregsave で宣言する関数群：

main 関数以外で関数呼び出しだけを行う関数がないか調べて見てください。

たとえば割り込み関数 intr1 が関数呼び出しだけを行う関数で、次のような呼び出し関係を持つとします。

```
intr1:
   proc1:
      func1:
   proc2:
   proc3:
```

proc1、proc2、proc3は他の関数から呼び出されていないと仮定します。

intr1()を#pragma regsave で宣言します。

proc1()、proc2()、proc3()を#pragma noregsave で宣言します。

これで、3 関数分のレジスタ退避／回復を 1 関数のレジスタ退避／回復に置き換えることができます。

先ほどの、Dhrystone Ver.2.1 の場合で、今までに、効果的だったオプション（1 バイト enum 型指定、引数渡し用レジスタ 3 つ）とモジュール間最適化機能、および#pragma noregsave をあわせて指定した場合の結果を示します。

最適化機能	サイズ（ROM）	実行サイクル数
なし（default）	3048	1580
#pragma noregsave **指定** **コンパイラオプション** **モジュール間最適化**	2944	1517

この指定の詳細については**「**5.4.9　**関数の入口／出口でレジスタの退避／回復コード出力を制御」**を参照してください。

5.1.6　CPU 特有の命令を活用する

(1)　8 ビット短絶対領域へ割り付ける

char/unsigned char 型データを 8 ビット絶対アドレスでアクセスします。

最適化機能	サイズ（ROM）	実行サイクル数
なし（default）	3048	1580
8 **ビット絶対アドレス指定**	3014	1566

【注】 オプションで 8 **ビット短絶対アドレス領域割り付け指定をした場合、**8 **ビット絶対アドレスエリアを超えてしまったため、エリア不足でシミュレータが正常に動作せず、実行サイクル数が測定できない場合もあります。**

【指定方法】
ダイアログメニュー：コンパイラタブカテゴリ:[最適化] データアクセスの@aa:8
コマンドライン　　：*abs8*

8 ビット短絶対アドレス領域は H'FFFF00～H'FFFFFF の範囲です。その範囲を超える場合は$ABS8 セクションにあるセクションをすべて、8 ビット短絶対アドレス領域へ割り付けることができません。
そのため、ファイル全体に abs8 オプションを指定するのではなく、どの変数を 8 ビット短絶対アドレスへ割り付けるかを検討します。
検討のポイントは、領域に収まるサイズでアクセス回数が多いことです。
コンパイラ出力のオブジェクトリストから、変数のサイズはわかりますが、アクセス回数はモジュール間最適化ツールの最適化情報リストからわかります。
モジュール間最適化ツールの最適化オプションで短絶対アドレス領域の活用を指定し、最適化情報リストを見てみます。

```
*** Variable Accessible with Abs8 ***
SYMBOL                          SIZE    COUNTS  OPTIMIZE
_Ch_1_Glob
                                 1        4
_Ch_2_Glob
                                 1        2
```

このファイルから変数の参照回数がわかります。
この結果を参考にして、#pragma abs8 で指定します。

```
#pragma abs8 (Ch_1_Glob,Ch_2_Glob)
```

この結果は次のようになります。

最適化機能	サイズ（ROM）	実行サイクル数
なし(default)	3048	1580
#pragma abs8 **指定**	3014	1566

この場合以外に、8 ビット絶対アドレス領域に割り付け可能な変数が多数ある場合は、アクセス回数を調べ、多い変数を割り付けます。

そこで、すでに、サイズが縮小されている指定を加えます。

最適化機能	サイズ（ROM）	実行サイクル数
なし(default)	3048	1580
#pragma abs8(Ch_1_Glob,Ch_2_Glob) ＋#pragma noregsave **指定** **＋コンパイラオプション** **＋モジュール間最適化機能**	2944	1517

わずかですが、上の指定の方が効率が良いようです。
この abs8 指定の詳細は**「**5.4.11　8 **ビット絶対アドレス領域の活用」**を参照してください。

(2)　16 ビット短絶対領域へ割り付ける

16 ビット絶対アドレスでアクセスするコードを生成します。

最適化機能	サイズ（ROM）	実行サイクル数
なし(default)	3048	1580
abs16 **オプション**	2988	1558

【指定方法】
ダイアログメニュー：コンパイラタブカテゴリ:[最適化] データアクセスの@aa:16
コマンドライン　　：*abs16*

16 ビット絶対アドレス領域は H'000000～H'007FFF、H'FF8000～H'FFFFFF です。
まず、オプションで abs16 を指定します。これによって ABS16 セクションに割り付け可能な変数が分かります。範囲内におさまるようであればそのままオプションで指定しますが、この領域はいろいろな領域と重なっているため、範囲外になってしまう場合は本文中で#pragma abs16 指定します。
シンボルのアクセス回数を調べ、回数の多い変数を ABS16 セクションへ割り付けます。
そのためモジュール間最適化ツールの最適化情報リストを見ます。
モジュール間最適化ツールの最適化オプションで短絶対アドレッシングモード活用を指定します。
すると、アクセス回数が出てきます。

```
*** Variable Accessible with Abs16 ***

SYMBOL                          SIZE    COUNTS  OPTIMIZE
_Ptr_Glob
                                  4        4
_Next_Ptr_Glob
                                  4        2
_Int_Glob
                                  2        6
_Bool_Glob
                                  2        2
_Arr_2_Glob
                               1388        1
_flmod
                                  3        2
_brk
                                  4        2
```

この結果を参考に、16 ビット絶対アドレス領域に割り付ける変数を#pragma abs16 で指定します。

```
#pragma abs16 (Int_Glob,Bool_Glob,Arr_2_Glob,Ptr_Glb,Next_Ptr_Glob)
```

最適化機能	サイズ（ROM）	実行サイクル数
なし（default**）**	3048	1580
abs16 **オプション**	2988	1558
#pragma abs16 **指定**	3012	1575

先ほどの#pragma abs8 指定も加えると

最適化機能	サイズ（ROM）	実行サイクル数
なし（default**）**	3048	1580
abs16 **オプション**	2988	1558
#pragma abs16 **指定**	3012	1575
#pragma abs16 + #pragma abs8 **指定**	2980	1561

という結果になります。

次に、Int_Glob、Ptr_Glob を 16 ビット絶対領域に割り付けるか、グローバルレジスタに割り付けるか検討します。既に調査済みの、効率が良い指定とあわせます。

最適化機能	サイズ（ROM）	実行サイクル数
なし（default**）**	3048	1580
#pragma abs16 (Bool_Glob, Arr_2_Glob, Next_Ptr_Glob Ptr_Glob,Int_Glob) ＋#pragma abs8(Ch_1_Glob,Ch_2_Glob) ＋#pragma noregsave **指定** **＋コンパイラオプション**	2920	1484
#pragma global_register(Int_Glob=E4,Ptr_Glob=ER5) ＋#pragma abs16 (Bool_Glob, Arr_2_Glob, Next_Ptr_Glob) ＋#pragma abs8(Ch_1_Glob,Ch_2_Glob) ＋#pragma noregsave **指定** **＋コンパイラオプション**	2958	1486

この結果、Int_Glob、Ptr_Glob はグローバルレジスタに割り付けるよりも、16 ビット絶対アドレス領域に割り付ける方のサイズ効率が良いことが分かりました。
abs16 指定の詳細については、**「**5.4.12　16 **ビット絶対アドレス領域の活用」**を参照してください。

またモジュール間最適化ツールで CPU の空き状況によっては 8 ビット絶対アドレス領域、16 ビット絶対アドレス領域へ割り付けられる場合があります。

(3)　メモリ間接領域への割り付け

この指定をすると、関数呼び出しがメモリ間接形式になります。
出力されたオブジェクトを参照したいため、リストの出力も指定しておきます。
デフォルト（サイズ優先）でこのメモリ間接領域割り付けオプションを指定した場合で見てみます。

最適化機能	サイズ（ROM）	実行サイクル数
デフォルト	3048	1580
メモリ間接領域割り付け指定	2994	1599

【指定方法】
ダイアログメニュー：コンパイラタブカテゴリ:[最適化] 関数アクセスの@@aa:8
コマンドライン　　：*indirect*

この領域へは実行時ルーチンを割り付けることも可能です。
#include <indirect.h>を指定すると、実行時ルーチンの呼び出しがメモリ間接呼び出しとなります。
コンパイラのオブジェクトリストの出力指定で、スタックフレーム情報出力を指定すると、各関数内で呼び出された、実行時ルーチン名が出てきます。

```
Function (File hv21_dhry_, Line   309): Proc_1

  Optimize Option Specified : No Allocation Information Available

Parameter Area Size     : 0x00000000 Byte(s)
Linkage Area Size       : 0x00000004 Byte(s)
Local Variable Size     : 0x00000000 Byte(s)
Temporary Size          : 0x00000000 Byte(s)
Register Save Area Size : 0x0000000c Byte(s)
Total Frame Size        : 0x00000010 Byte(s)

Used Runtime Library Name
$MVN$3
```

その結果、$MVN$3 の呼び出しがメモリ間接となりました。

個別に指定するには#pragma indirect $MVN$3 と指定します。

メモリ間接領域は 0x00000000～0x000000ff ですので、すべてを割り付けることが可能です。
しかし、この領域は例外処理ベクタ領域と重なっているため、割り付けに注意が必要となります。
セクションを分割して、重ならないように割り付ける必要があります。
この場合は、すべてサイズに収まりますが、メモリ間接領域よりも大きなサイズの$INDIRECT セクションになった場合など、各関数でアクセス回数の多いものを改めて、#pragma indirect にて個別に指定します。さらに#pragma indirect section でセクションを分割し、割り付ける必要があります。
この場合のオプションでの指定は以下の指定と同じになります。

```
#pragma
indirect(main,malloc,Proc1,Proc2,Proc3,Proc4,Proc5,Proc6,Proc7,Proc8,Func1,Func2,Func3)
#pragma indirect $MVN$3
```

すでに調査済みの、効率が良い指定とあわせると、次のような結果になります。

最適化機能	サイズ（ROM）	実行サイクル数
なし（default）	3048	1580
#pragma abs8(Ch_1_Glob,Ch_2_Glob) ＋#pragma abs16 (Bool_Glob, Arr_2_Glob, Ptr_Glob, Next_Ptr_Glob) ＋#pragma noregsave **指定** **＋コンパイラオプション** **＋モジュール間最適化機能** ＋#pragma indirect **指定**	2902	1496

という結果になります。この詳細については**「**5.4.13　**メモリ間接形式の活用」**を参照してください。

しかし、ここでメモリ間接形式の関数呼び出しの指定をしなくても、モジュール間最適化機能で CPU の空き状況によってはメモリ間接形式の関数呼び出しを行う場合があります。

5.2 実行速度優先の最適化

5.2.1 SPEED オプションを指定する

実行速度優先の最適化を行いたいときは、SPEED オプションを指定します。

この結果次のようになります。

最適化機能	サイズ（ROM）	実行サイクル数
デフォルト	3048	1580
SPEED **オプション**	3420	1325

【指定方法】
ダイアログメニュー：コンパイラタブカテゴリ:[最適化] 最適化方法のスピード優先最適化
コマンドライン　　：*speed*

Dhrystone Ver.2.1 では 372 バイト大きくなりますが、実行サイクル数が 255 サイクル速くなりました。

(1) サブオプションを選択する

SPEED オプションを指定することによって、サイズが大きくなるようなスピード最適化も実行します。
そのため、チューニングなどで細かく指定する必要があるかもしれません。それには、各サブオプションの効果を知っておくことが一番確実な方法です。目標の ROM サイズになるようにそれぞれの効果を組み合わせて、指定するサブオプションを決めます。下表の Dhrystone Ver.2.1 のデータを参照するとよいでしょう。

最適化機能	サイズ（ROM）	実行サイクル数
すべて指定	3420	1325
Register	3048	1580
Shift to multiple	3048	1580
Struct assignment	3074	1527
Switch judgement	3048	1580
Maximum nodes of inline function(105)	3314	1437
Loop optimization	3048	1580
Expression	3080	1526

【注】 H8/300、H8/300H **の場合、Register が指定されていないとき、コンパイラはレジスタの退避／回復を関数（実行時ルーチンライブラリ）呼び出しで行います。Register 指定時は関数呼び出しの代わりに** PUSH/POP **命令を生成します。**
H8S/2000、H8S/2600 **シリーズでは、レジスタの退避／回復は常に** STM/LDM **命令を用いて行います。（使用レジスタにより** PUSH/POP **命令の場合あり）そのため、Register を指定しても効果はありません。**
H8/300H **アドバンストモードで Register を指定したときの効果は次のとおりです。**

最適化機能	サイズ（ROM）	実行サイクル数
すべて指定	3422	1598
Register	3262	1721

Maximum nodes if inline function では自動的にインライン展開される関数のノード数を指定します。
このノード数とはコンパイラ内部で使用する処理単位ですので、正確に調査することはできませんが、一般的にサイズの大きな関数はノード数が多くなります。デフォルトではノード数 105 を指定してあります。
インライン展開を行いたくない(ノード数 0)場合は、チェックをはずしてください。
また、ノードを最高にした場合と結果を比較します。(ノード数は、1～65535 まで選択できます）

最適化機能	サイズ（ROM）	実行サイクル数
Maximum nodes of inline function(1)	3052	1549
Maximum nodes of inline function(105)	3314	1437
Maximum nodes of inline function(65535)	3314	1437

なお、すべての関数をインライン展開すると、サイズ面のみならず、スピード面でも効率が低下することがあります。これは、関数のサイズが大きくなり、最適化処理ができなくなるためです。

自動インライン展開は、なるべくノード数を多くしないようにして、特にインライン展開したい関数に関しては、#pragma inline で指定した方が効率良いでしょう。

SPEED オプションの詳細については**「**5.4.7　SPEED **オプション」**を参照してください。

5.2.2　最適化オプションのチューニング

(1)　ブロック転送命令（eepmov）の利用

構造体の代入などでブロック転送命令（EEPMOV）を使用します。

オブジェクトサイズ／実行サイクル数は次のようになります。

最適化機能	サイズ（ROM）	実行サイクル数
SPEED **オプション**	3420	1325
SPEED **オプション＋ブロック転送命令の利用**	3366	1285

【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構造体の代入式を eepmov 命令で展開

コマンドライン　　：*eepmov*

EEPMOV 命令には CPU 仕様上の次の制限があります。

EEPMOV.B　→NMI 以外の割り込みを検出しません。

EEPMOV.W　→NMI 以外の割り込みを検出しません。

命令実行中に NMI 割り込みが発生すると、転送結果が保証されません。

コンパイルリストを開いてみましょう。オブジェクトリストの中で、EEPMOV 命令をサーチします。EEPMOV 命令を使用している箇所が、上記使用条件で問題ないことを確認してください。

ファイル全体ではなく、一部の構造体データ転送に EEPMOV 命令を使用したいとき、組み込み関数 eepmov();を使用してください。

この指定に関する詳細は**「**5.4.6　**ブロック転送命令」**を参照してください。

(2)　その他の最適化オプションのチューニング

さらに、サイズの面でも有効だった、各オプションの指定をしてみます。

1 バイト enum 型を指定します。

最適化機能	サイズ（ROM）	実行サイクル数
SPEED **オプション**	3420	1325
SPEED **オプション＋ブロック転送命令指定**	3366	1285
SPEED **オプション＋ブロック転送命令指定** **＋**1 **バイト** enum **型指定**	3392	1296

少し、実行速度が落ちました。

次に引数渡し用レジスタ数 3 を指定します。

最適化機能	サイズ（ROM）	実行サイクル数
SPEED **オプション**	3420	1325
SPEED **オプション＋ブロック転送命令指定**	3366	1285
SPEED **オプション＋ブロック転送命令指定** **＋引数渡し用レジスタ** 3 **つ**	3348	1249

実行速度が向上しました。

さらに変数割り付けレジスタ数を指定します。

最適化機能	サイズ（ROM）	実行サイクル数
SPEED **オプション**	3420	1325
SPEED **オプション＋ブロック転送命令指定**	3366	1285
SPEED **オプション＋ブロック転送命令指定** **＋変数割り付けレジスタ数拡張なし**	3366	1285

これらの結果から、オプションではブロック転送命令指定と、引数渡し用レジスタ 3 指定を選びます。この指定の詳細については**「**5.4.3 **引数渡し用レジスタ数の指定」**を参照してください。

5.2.3 モジュール間最適化機能を使用する

次にモジュール間最適化ツールを使用して最適化を行い、さらに、実行速度効率の良いオブジェクトを目指します。

モジュール間最適化ツールで最適化を指定する場合、あらかじめコンパイラやクロスアセンブラでモジュール間最適化用付加情報ファイルの出力を指定しておきます。

【指定方法】
C/C++コンパイラ
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*

クロスアセンブラ
ダイアログメニュー：アセンブラタブカテゴリ:[オブジェクト] モジュール関最適化
コマンドライン　：*goptimize*

また、HEW1.2 ではモジュール間最適化時にリンクする標準ライブラリについてもモジュール間最適化用付加情報ファイルが用意されています。Windows 版では圧縮した形で提供するので、解凍してから使用します。

使用するライブラリ名と同じ圧縮ファイル（*.exe）をダブルクリックすると、自己解凍し、付加情報ファイルを含むディレクトリが生成されます。

この、ライブラリのモジュール間最適化については、H8S,H8/300 **シリーズ** C/C++**コンパイラ補足資料**に記載されているので参照してください。

HEW2.0 以降では標準ライブラリ構築ツールのモジュール間最適化機能にてライブラリを作成してください。**標準ライブラリタブカテゴリ:[最適化] モジュール間最適化**をチェックするとモジュール間最適化用付加情報が出力されます。

(1) デフォルトの最適化

モジュール間最適化ツールは次の最適化機能を持っています。

No.	内容	ダイアログメニュー	サブコマンドオプション
1	**定数／文字列の統合**	**Unify strings　または** **定数/文字列の統合**	*String_Unify*
2	**未参照変数／関数の削除**	**Eliminate dead code　または** **未参照シンボルの削除**	*Symbol_delete*
3	**変数アクセスの最適化**	**Use short addressing　または** **短絶対アドレッシングモード活用**	*Variable_access*
4	**関数アクセスの最適化**	**Use indirect call/jump　また** **間接アドレッシングモード活用**	*Funcation_call*
5	**レジスタの再割り付け**	**Reallocate registers　または** **レジスタ退避・回復の最適化**	*Register*
6	**共通コードの統合**	**Eliminate same code　または** **共通コードの統合**	*Same_code*
7	**分岐命令の最適化**	**Optimize branches　または** **分岐命令の最適化**	*Branch*

まずコンパイラで最も効率の良い最適化を行った場合の指定は次のとおりです。

最適化機能	サイズ（ROM）	実行サイクル数
有効なコンパイラ最適化オプション （SPEED オプション＋ブロック転送命令指定 ＋引数渡し用レジスタ 3 つ）	3348	1249

モジュール間最適化ツールで最適化のデフォルトは最適化を行いません。つまり、リンクを行ったのみの状態です。そのため、デフォルト状態だとコンパイラの最適化を行った結果と同じ結果になります。

(2) 各モジュール間最適化項目を指定する

モジュール間最適化ツールの各最適化項目を順に指定していきます。

最適化機能	モジュール間最適化機能	サイズ（ROM）	実行サイクル数
有効な コンパイラ最適化オプション	-	3348	1249
	定数／文字列の統合	3348	1249
	未参照変数／関数の削除	2984	1249
	変数アクセスの最適化	3258	1232
	関数アクセスの最適化	3332	1250
	レジスタの再割り付け	3332	1249
	共通コードの統合	3348	1249
	分岐命令の最適化	3348	1249

(3) スピード優先のモジュール間最適化機能を有効にする

定数／文字列の統合、未参照変数／関数の削除、変数アクセスの最適化、レジスタの再割り付け、分岐命令の最適化を行います。

最適化機能	モジュール間最適化機能	サイズ（ROM）	実行サイクル数
有効な コンパイラ最適化オプション	-	3348	1249
	スピード優先の最適化	2906	1232

(4) すべてのモジュール間最適化機能を有効にする

すべての最適化機能を有効にします。

最適化機能	モジュール間最適化機能	サイズ（ROM）	実行サイクル数
有効な コンパイラ最適化オプション	-	3348	1249
	全項目の最適化	2902	1232

すべての最適化を行うと、抑止しなければならない部分でも、最適化がかかってしまうことがあります。よくリストを見て確認の上で指定をしてください。

5.2.4 拡張機能を選択する

(1) グローバル変数にレジスタを割り付ける

サイズ最適化の際に指定した変数をグローバルレジスタ指定します。

（A）
```
#pragma global_register(Int_Glob=E4,Ch_1_Glob=R4H,Ch_2_Glob=R4L,Ptr_Glob=ER5)
```

また、ワークレジスタを増やすため、次のような場合も行います。

（B）
```
#pragma global_register(Int_Glob=E4,Ch_1_Glob=R4H,Ch_2_Glob=R4L)
```

（C）
```
#pragma global_register(Ptr_Glob=ER5)
```

最適化機能	サイズ（ROM）	実行サイクル数
SPEED **オプション**	3420	1325
SPEED **オプション＋ブロック転送命令指定** **＋引数渡し用レジスタ** 3 **つ**	3348	1249
SPEED **オプション＋ブロック転送命令指定** **＋引数渡し用レジスタ** 3 **つ＋**#pragma global_register **（A）指定**	3318	1246
SPEED **オプション＋ブロック転送命令指定** **＋引数渡し用レジスタ** 3 **つ＋**#pragma global_register **（B）指定**	3370	1262
SPEED **オプション＋ブロック転送命令指定** **＋引数渡し用レジスタ** 3 **つ＋**#pragma global_register **（C）指定**	3296	1246

この指定についての詳細は**「**5.4.8 **グローバル変数にレジスタを割り付け」**を参照してください。

(2) 関数の入口／出口でレジスタの退避／回復コード出力を制御する

サイズ優先のときに調査した次の指定を行います。

```
#pragma noregsave (main)
```

結果は次のようになります。

最適化機能	サイズ（ROM）	実行サイクル数
コンパイラオプション **モジュール間最適化機能**	2906	1232
コンパイラオプション **モジュール間最適化機能** **＋**#pragma noregsave **指定**	2906	1232

実行速度が速くなりました。この指定についての詳細は**「**5.4.9 **関数の入口／出口でレジスタの退避／回復コード出力を制御」**を参照してください。

5.2.5 インライン展開機能を使う

(1) 関数のインライン展開を指定

#pragma inline は関数呼び出しの代わりにインライン展開を行う関数を宣言します。インライン展開の結果は、オブジェクトサイズは大きくなりますが、実行速度が速くなります。

しかし、SPEED オプションの自動インライン展開のところでも説明しましたが、すべての関数をインライン展開指定すると、オブジェクトサイズのみならず、実行速度も遅くなります。

#pragma inline で宣言する関数は、ループの特にネストが深いところで呼び出されている関数を宣言すると、実行速度が効果的に向上できます。

関数のネスト関係を調べたところ Dhrystone Ver.2.1 では、以下のようになります。

```
main:
    malloc:
    strcpy:
    Proc_1:
       Proc_3:
          Proc_7:
       Proc_6:
          Func_3:
       Proc_7:
    Func_2:
       Func_1:
       strcmp:
    Func_1:
    Proc_8:
    Proc_7:
    Proc_6:
       Func_3:
    Proc_5:
    Proc_4:
    Func_2:
```

この結果、ネストの深いところから指定していきます。自動インライン展開（オプションで指定したインライン展開）のノード数 105 の場合と比較します。

最適化機能	サイズ（ROM）	実行サイクル数
なし(default)	3052	1549
自動インライン展開	3306	1445
インライン展開指定（Proc7,Func3）	3048	1589
インライン展開指定（Proc7,Func3,Func1,strcmp,Proc3,Proc6）	3048	1589
インライン展開指定（Proc7,Func3,Func1,strcmp,Proc3,Proc6, malloc,strcpy,Proc5,Proc4,Proc1）	3048	1589
すべての関数をインライン展開指定	3322	1445

【注】 Proc8 **と** Func2 **はインライン展開されません。**

この結果から、自動インライン展開がスピードが速いオブジェクトとなります。

このように、関数の呼び出し関係と実行サイクル数とオブジェクトサイズとの兼ね合いで、細かく指定すると、理想的なオブジェクトが作成されます。

#pragma inline 宣言は、同一ファイル内に関数本体と関数呼び出しが同時に存在しないと無効になります。また、インライン展開の対象となる関数を static 指定しておくと、実態のコードが出力されず、呼び出された関数側でのみコード展開されるため、サイズが小さくできます。できるだけ、指定した方がよいでしょう。

この指定の詳細については**「**5.4.10 **関数のインライン展開を指定」**を参照してください。

(2) アセンブリ記述関数をインライン展開

C/C++でプログラムを組んでいく中で、この部分は特に性能を上げたいという場合に、アセンブリ言語で記述することがあります。その場合は、関数内にアセンブリ言語を記述し、その関数を#pragma inline_asm 指定すると、呼び出された先でインライン展開をすることができます。

この指定の詳細については H8S、H8/300 **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「**10.2.1　#pragma **拡張子、キーワード」**の#pragma inline_asm を参照してください。

5.2.6 CPU 特有の命令を活用する

(1) 8 ビット短絶対領域へ割り付ける

サイズ優先の最適化の場合に選択した変数を 8 ビット絶対アドレス指定します。

```
#pragma abs8 (Ch_1_Glob,Ch_2_Glob)
```

この結果、次のようになります。

最適化機能	サイズ（ROM）	実行サイクル数
コンパイラオプション **モジュール間最適化機能** **＋**#pragma noregsave **指定**	2906	1232
コンパイラオプション **モジュール間最適化機能** **＋**#pragma noregsave **指定** **＋**#pragma abs8 **指定**	2906	1232

サイズ優先時（5.1.6）と異なり、測定した Dhrystone Ver.2.1 のプログラムでは結果に違いがありませんでしたが指定しておいた方が良いでしょう。

この指定の詳細については**「**5.4.11　8 **ビット絶対アドレス領域の活用」**を参照してください。

(2) 16 ビット短絶対領域へ割り付ける

サイズ優先の最適化の際に選択した変数を 16 ビット絶対アドレスへ指定します。

```
#pragma abs16 (Int_Glob,Bool_Glob,Arr_2_Glob,Ptr_Glob,Ptr_Glb_Next)
```

この結果は次のようになります。

最適化機能	サイズ（ROM）	実行サイクル数
コンパイラオプション **モジュール間最適化機能** **＋**#pragma noregsave **指定**	2906	1232
コンパイラオプション **モジュール間最適化機能** **＋**#pragma noregsave **指定＋**#pragma abs8 **指定** **＋**#pragma abs16 **指定**	2894	1231

スピードもサイズも良くなりました。

この指定の詳細については**「**5.4.12　16 **ビット絶対アドレス領域の活用」**を参照してください。

(3) メモリ間接領域割り付け

サイズ優先の最適化の際に選択した関数をメモリ間接領域割り付け指定します。

```
#pragma
indirect(Proc0,main,malloc,Proc1,Proc2,Proc3,Proc4,Proc5,Proc6,Proc7,Proc8,Func1,Func2,Func3)
```

この結果次のようになります。

最適化機能	サイズ（ROM）	実行サイクル数
コンパイラオプション **モジュール間最適化機能** **＋**#pragma noregsave **指定＋**#pragma abs8 **指定** **＋**#pragma abs16 **指定**	2894	1231
コンパイラオプション **モジュール間最適化機能** **＋**#pragma noregsave **指定＋**#pragma abs8 **指定** **＋**#pragma abs16 **指定＋**#pragma indirect **指定**	2892	1232

スピードが落ちたのでこの指定は行わないこととします。

5.3 サイズとスピードの兼ね合い

前節までで述べたように、コンパイラの最適化にはオブジェクトサイズを小さくするものと、実行速度を速くするものがあります。それぞれの最適化方法は述べてきたとおりですが、それぞれの兼ね合いとして実行速度を要求される関数と、サイズを要求される関数を別ファイルにして、ファイルごとにサイズ最適化とスピード最適化を選択できるようにするのが理想的です。

完全にファイルを分割することはできなくても、プログラム全体でどこが一番スピードを要求されるところかを把握しておくことは重要です。最もスピードが要求されるファイル（あるいは関数）については（オプション＋拡張機能＋コーディング＋モジュール間最適化）により高速化を行い、それ以外はサイズ最適化を選択すれば効果的にオブジェクト性能を向上することができます。

これまでの調査結果を表にまとめると、以下のようになります。

サイズ効率が最も良い、オプション、拡張機能の指定を示します。

指定	内容	サイズ		実行速度	
		Byte	%	Cycle	%
デフォルト	**－**	3048	100	1580	100
コンパイラオプション	1 **バイト** enum **型指定** **＋引数渡し用レジスタ** 3 **つ**	3034	99	1527	97
コンパイラオプション **＋モジュール間最適化機能**	1 **バイト** enum **型指定** **＋引数渡し用レジスタ** 3 **つ** **＋すべてのモジュール間最適化機能**	2946	97	1517	96
コンパイラオプション **＋モジュール間最適化機能** **＋拡張機能指定**	1 **バイト** enum **型指定** **＋引数渡し用レジスタ** 3 **つ** **＋すべてのモジュール間最適化機能** **＋**#pragma abs8 **指定** **＋**#pragma abs16 **指定** **＋**#pragma noregsave **指定** **＋**#pragam indirect **指定**	2902	95	1496	95

次にスピード効率が最も良い、オプション、拡張機能の指定を示します。

指定	内容	サイズ		実行速度	
		Byte	%	Cycle	%
デフォルト	−	3048	100	1580	100
コンパイラオプション	SPEED **オプション** **＋ブロック転送命令指定** **＋引数渡し用レジスタ** 3 **つ**	3348	110	1249	79
コンパイラオプション **＋モジュール間最適化機能**	SPEED **オプション** **＋ブロック転送命令指定** **＋引数渡し用レジスタ** 3 **つ** **＋スピード優先のモジュール間最適化機能**	2906	95	1232	78
コンパイラオプション **＋モジュール間最適化機能** **＋拡張機能指定**	SPEED **オプション** **＋ブロック転送命令指定** **＋引数渡し用レジスタ** 3 **つ** **＋スピード優先のモジュール間最適化機能** **＋**#pragma noregsave **指定** **＋**#pragma abs8 **指定** **＋**#pragma abs16 **指定**	2894	95	1231	78

以上のように Dhrystone Ver.2.1 はオプション指定なしのときと比較すると、オプション、拡張機能を活用することにより、サイズで最大 5%、実行サイクルで最大 22%性能を向上することができました。

オプションや拡張機能は、コーディングの修正に比べ、簡単に、かつ高い効果を得ることができます。積極的に活用して、性能向上に役立ててください。

5.4 最適化機能の詳細

最適化機能としては以下のものを用意しています。
1 から 23 まではコンパイラの最適化で、24 から 30 はモジュール間最適化ツールで行う最適化です。
なお、性能測定は以下の条件で測定しています。

【測定クロスツール】
H8S,H8/300 C/C++ Library Generator (V. 2.01.00.001)
H8S,H8/300 C/C++ Compiler (V. 6.01.00.009)
H8S,H8/300 Assembler (Ver. 6.01.01.000)
Optimizing Linkage Editor (V. 9.00.02.000)

【オプション指定】
各項目の詳細にオプション指定指示がない限り、デフォルトオプションを設定。

【測定条件】

条件	H8/300,H8/300H	H8S/2600,H8S/2000	H8SX
バス幅	16	16	32
メモリへのアクセスステート	2	1	1
フェッチサイズ	-	-	32

No.	最適化機能	サイズ効率向上	実行速度向上	参考
1	1 バイト enum 型の活用	○	○	5.4.1
2	乗除算仕様の拡張解釈	○	○	5.4.2
3	引数渡し用レジスタ数の指定	△	△	5.4.3
4	変数割り付けレジスタ数の拡張	△	△	5.4.4
5	外部変数の最適化	-	-	5.4.5
6	ブロック転送命令	×	○	5.4.6
7	SPEED オプション			5.4.7
8	レジスタ退避/回復コードのスピード優先コード展開	×	○	5.4.7(1)
9	シフト式のスピード優先コード展開	×	○	5.4.7(2)
10	構造体、double 型データの代入コード展開	×	○	5.4.7(3)
11	switch 文のスピード優先コード展開	×	○	5.4.7(4)
12	サイズの小さい関数のインライン展開	×	○	5.4.7(5)
13	ループ式のスピード優先コード展開	△	○	5.4.7(6)
14	実行時ルーチン呼び出し抑止	×	○	5.4.7(7)
15	グローバル変数にレジスタを割り付け	△	△	5.4.8
16	関数の入口／出口でレジスタの退避／回復コード出力を制御	○	○	5.4.9
17	関数のインライン展開を指定	×	○	5.4.10
18	8 ビット絶対アドレス領域の活用	○	○	5.4.11
19	16 ビット絶対アドレス領域の活用	○	○	5.4.12
20	メモリ間接領域への割り付け	○	×	5.4.13
21	拡張メモリ間接形式の活用	○	×	5.4.14
22	ポインタサイズ 2byte 指定	○	○	5.4.15
23	境界調整数、バウンダリ調整指定	○	○	5.4.16
24	定数／文字列の統合	−	−	5.4.17(1)
25	未参照変数／関数の削除	−	−	5.4.17(2)
26	変数アクセスの最適化	−	−	5.4.17(3)
27	関数アクセスの最適化	−	−	5.4.17(4)
28	レジスタ退避／回復コードの最適化	−	−	5.4.17(5)
29	共通コードの統合	−	−	5.4.17(6)
30	分岐命令の最適化	−	−	5.4.17(7)

【記号説明】

○：向上が見られる項目

△：プログラムによっては向上が見られる項目

×：効率が低下する項目

5.4.1 1 バイト enum 型の活用

サイズ効率	○	実行速度	○

■説明

enum 型メンバの値が –128～127 の場合、1 バイト型として演算をオプションで指定することができます。

enum 型は、言語仕様では 2 バイト領域を確保しますが、enum オプションを指定すると enum 型メンバの値が 1 バイトデータとして確保され、演算が行われます。

言語仕様に準拠していないため、コンパイラオプションでは default 状態で指定なしとしていますが、常に指定しておくことをお勧めします。

■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] 列挙型データを char 型で扱う**

コマンドライン　　：*byteenum*

■使用例

enum 型データ E1 を 1 とします。
（C/C++プログラム）

```
enum EN1 {a=0,b,c,d,e}E1;
void func(void)
{
   E1=1;
}
```

←enumのメンバの値がバイトの範囲です

（アセンブラ展開コード）

指定なし

```
_func:
     MOV.W      #1,R0
     MOV.W      R0,@_E1:32
     RTS
     .SECTION   B,DATA,ALIGN=2
_E1:
     .RES.W     1
```

2バイトデータ

指定あり

```
_func:
     MOV.B      #1,R0L
     MOV.B      R0L,@_E1:32
     RTS
     .SECTION   B,DATA,ALIGN=2
_E1:
     .RES.B     1
```

1バイトデータ

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	12	10	12	10	10
指定あり	10	8	10	8	8

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	8	8	6
指定あり	8	8	6

実行処理速度表 [サイクル]

CPU 種別	H8S/2600, H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	11	9	22	18	18
指定あり	10	8	20	16	16

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	8	8	7
指定あり	8	8	7

5.4.2 乗除算仕様の拡張解釈

サイズ効率	○	実行速度	○

■説明

乗除算のコード展開を ANSI 規格から拡張解釈して出力します。
このオプションを指定した場合、以下の様な解釈となるため言語仕様で規定された値の保証範囲と仕様が異なるので、演算結果が指定しない場合と、異なることがあります。

対象演算	us1*us2 の演算サイズ（H8S/2600 の例）	
	拡張解釈指定時	ANSI 規格解釈指定時
unsigned short us1,us2; unsigned long ul; ul＝us1*us2;	us1*us2 は unsigned long で演算します 出力例：MOV.W @_us1.Rd MOV.W @_us2.Rs MULXU.W Rs,ERd MOV.L ERd,@_ul us1*us2 の結果 4 バイトを u1 に代入します	us1*us2 は unsigned short で演算します 出力例：MOV.W @_us1.Rd MOV.W @_us2.Rs MULXU.W Rs,ERd EXTU.L ERd MOV.L ERd,@_ul us1*us2 の結果の下位 2 バイトを 0 拡張して u1 に代入します
Unsigned short us1,us2,us3 Unsigned short us; us＝ us1*us2/us3;	us1*us2 は unsigned long で演算します 出力例：MOV.W @_us1.Rd MOV.W @_us2.Rs MULXU.W Rs,ERd MOV.L @_us3.Rs DIVXU.W Rs,ERd MOV.L Rd,@_us us1*us2 の結果 4 バイトを演算命令の被除数にします	us1*us2 は unsigned short で演算します 出力例：MOV.W @_us1.Rd MOV.W @_us2.Rs MULXU.W Rs,ERd EXTU.L ERd MOV.L @_us3.Rs DIVXU.W Rs,ERd MOV.L Rd,@_us us1*us2 の結果の下位 2 バイトを 0 拡張した値を除算命令の被除数にします

■指定方法

ダイアログメニュー：**コンパイラタブカテゴリ:[オブジェクト] 乗除算演算方法**
ANSI 非準拠(16bit*16bit=32bit)
コマンドライン　：*cpuexpand*

■使用例

2 バイトデータ同士の乗算の結果を 4 バイト型データに格納します。
（C/C++プログラム）

```
unsigned long ll;
unsigned short a,b;
void func()
{
   ll=a*b;
}
```

（アセンブリ展開コード）

指定なし

```
_func:
    MOV.W      @_a:32,R0
    MOV.W      @_b:32,E0
    MULXU.W    E0,ER0
    EXTU.L     ER0
    MOV.L      ER0,@_ll:32
    RTS
```

指定あり

```
_func:
    MOV.W      @_a:32,R0
    MOV.W      @_b:32,E0
    MULXU.W    E0,ER0

    MOV.L      ER0,@_ll:32
    RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	26	20	26	20	24
指定あり	24	18	24	18	22

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	26	26	20
指定あり	24	24	18

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	24	20	62	54	136
指定あり	23	19	60	52	184

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	14	14	13
指定あり	14	14	12

5.4.3 引数渡し用レジスタ数の指定

サイズ効率	△	実行速度	△

■説明

引数を割り付けるレジスタの数を選択することができます。引数をレジスタに割り付けると、スタックに割り付ける場合よりアクセスサイズが小さくなります。しかし、引数用のレジスタが増えるとその分ワークレジスタが減ることになり、複雑な計算などの場合にデータがレジスタに割り付けられず、かえってオブジェクト効率が悪くなってしまうことがあります。

regparam=3 オプションあり、なしの両方を試してみて、オブジェクト効率の良い結果が出た方を指定してください。

■指定方法

ダイアログメニュー：**CPU タブの引数格納レジスタを 2 つから 3 つに変更**
コマンドライン　　：*regparam=3*

■使用例

引数レジスタ数を 3 にした方が効率がよくなる場合の使用例を示します。

（C/C++プログラム）

```
extern short ee;
void func(short a,short b,short c,short d,long e)
{
   ee=a*b*c*d/e;
}
```

（コンパイル結果アセンブリ展開コード）

指定なし

```
_func:
      PUSH.L      ER2
      SUBS.L      #2,SP
      MOV.W       R0,R2
      MULXU.W     E0,ER2
      MULXU.W     R1,ER2
      MOV.W       R2,R1
      MULXU.W     E1,ER1
      EXTS.L      ER1
      MOV.W       R0,@SP
      MOV.L       ER1,ER0
      MOV.L       @(10:16,SP),ER1
      JSR         @$DIVL$3:24
      MOV.W       R0,@_ee:32
      POP.L       ER2
      RTS
```

指定あり

```
_func:
      PUSH.L      ER3
      SUBS.L      #2,SP
      MOV.W       R0,R3
      MULXU.W     E0,ER3
      MULXU.W     R1,ER3
      MOV.W       R3,R1
      MULXU.W     E1,ER1
      EXTS.L      ER1
      MOV.W       R0,@SP
      MOV.L       ER1,ER0
      MOV.L       ER2,ER1
      JSR         @$DIVL$3:24
      MOV.W       R0,@_ee:32
      ADDS.L      #2,SP
      POP.L       ER3
      RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	42	40	46	44	72
指定あり	38	36	42	40	70

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	36	36	36
指定あり	34	34	32

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	144	138	294	282	686
指定あり	140	134	284	272	682

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	44	43	44
指定あり	39	39	41

■備考および注意事項

この指定は、ファイルごとに指定を変えたり、リンクするライブラリの指定と異なった指定はできません。この指定を変更した場合は、すべてのファイルのオプションの指定と、リンクするライブラリも変更してください。

さらに、アセンブリプログラムとリンクしている際には、関数呼び出しのインタフェースを変更する必要があります。

C/C++プログラムとアセンブリプログラムとのリンケージについては H8S、H8/300 **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「**9.3 C/C++**プログラムとアセンブリプログラムとの結合」**を参照してください。

5.4.4 変数割り付けレジスタ数の拡張

サイズ効率	△	実行速度	△

■説明

オプションで変数を割り付けるレジスタの数を変更することができます。（4 個または 3 個）

一般のプログラムでは変数割り付け用レジスタ数が 4 個の方が性能がよくなります。複雑な式を記述してテンポラリ用のレジスタが不足したときに、変数割り付け用レジスタ数が 3 個の方がオブジェクト性能がよくなる場合があります。

通常は変数割り付け用レジスタ数が 4 個を指定し、ROM 詰めの際などに、どちらのオプションを指定した方が効率が良いか試してみてください。

■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] 変数割り付けレジスタ数を拡張**
コマンドライン　　：*regexpansion*

■使用例

変数割り付け用レジスタ数が 3 個の方が効率が良いプログラムを示します。
（C/C++プログラム）

```
long func(short a,long b,short c,char d,long e)
{
    long x,y,z;
    x=a+b;
    y=b*c;
    z=a/e;
    return (a*x*(z+y)*b*d+e*z-e/x*c/(x*y*a*z));
}
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
変数レジスタ数 4	202	202	190	190	416
変数レジスタ数 3	202	202	190	190	416

CPU 種別	H8SX		
	MAX	ADV	NML
変数レジスタ数 4	150	150	150
変数レジスタ数 3	150	150	150

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
変数レジスタ数 4	783	752	2158	2082	4836
変数レジスタ数 3	783	752	2158	2082	4836

CPU 種別	H8SX		
	MAX	ADV	NML
変数レジスタ数 4	174	187	169
変数レジスタ数 3	174	187	169

5.4.5 外部変数の volatile 化

サイズ効率	—	実行速度	—

■説明

```
    :
a=0;  //①
a=1;  //②
    :
```

上記式に対してコンパイラは①の代入式を削除する最適化を行います。I/O ポートや割り込み処理で使用する変数のように、①の代入式を削除してはいけない場合は、最適化を抑止するために変数を volatile 宣言するのが一般的です。

volatile オプションを指定すると、指定したファイル中のすべての外部変数が volatile 宣言されたものとみなされるため、volatile 宣言のない外部変数に対しても、上記のような最適化を抑止することができます。

ただし、オブジェクト性能も低下してしまいます。割り込み関数を記述したファイルや I/O レジスタへのアクセスのみを行うファイルなど、最適化してはいけないプログラムが記述されたファイルに対して使用するようにしてください。通常、最適化してはいけない変数に対しては、ソースプログラムで volatile 宣言をし、外部変数の volatile は抑止してコンパイルします。

■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] ループ判定式の最適化抑止**
コマンドライン　　：*volatile*

■使用例

外部変数 a に順次 0、1、2 を代入します。
（C/C++プログラム）

```
unsigned int a;
void func()
{
    a=0;
    a=1;
    a=2;
}
```

（アセンブリ展開コード）

指定なし

```
_func:
                    a=2のコードのみ
     MOV.W      #2,R0
     MOV.W      R0,@_a:32
     RTS
```

指定あり

```
_func:
     SUB.W      R0,R0          a=0
     MOV.W      R0,@_a:32
     MOV.B      #1,R0L         a=1
     MOV.W      R0,@_a:32
     MOV.B      #2,R0L         a=2
     MOV.W      R0,@_a:32
     RTS
```

■備考および注意事項

volatile オプションを指定すると、ファイル中のすべての外部変数が volatile 変数となります。個々の外部変数を volatile 指定したい場合は、次のようにソース上で volatile 指定を行い、volatile オプションを使用しないようにしてください。

```
volatile unsigned int a;
void func()
{
    a=0;
    a=1;
    a=2;
}
```

上記例のvolatileオプションありのコードと同じコードが出力されます

5.4.6 ブロック転送命令

サイズ効率	×	実行速度	○

■説明

構造体の代入の場合、実行時ルーチンを呼び出して処理しますが、オプションで、構造体の代入式においてブロック転送命令が出力されるので実行処理速度を向上することができます。

しかし、EEPMOV.W 命令は命令実行中に NMI 割り込みが発生すると、転送結果が保証されません。

この条件で問題がないことを確認した上で、このオプションを指定してください。

一部の構造体データ転送に対して、EEPMOV 命令を出力させたい場合は、組み込み関数の eepmov()を指定してください。

■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] 構造体の代入式を eepmov 命令で展開**

コマンドライン　　：*eepmov*

■使用例

構造体 s2 を s1 へ代入します。

（C/C++プログラム）

```
struct S{
   char  cc;
   short ss;
   long  ll;
   long  ll2;
}s1,s2;
void main()
{
   s1=s2;
}
```

（コンパイル結果アセンブリ展開コード）

指定なし

```
_main:
     PUSH.L     ER2
     MOV.L      #_s2,ER0
     MOV.L      #_s1,ER1
     SUB.L      ER2,ER2
     MOV.B      #12,R2L
     JSR        @$MVN$3:24
```

実行時ルーチン呼び出しで処理

```
     POP.L      ER2
     RTS
```

指定あり

```
_main:
     STM.L      (ER4-ER6),@-SP
     MOV.L      #_s2,ER5
     MOV.B      #12,R4L
     MOV.L      #_s1,ER6
     EEPMOV.B
```

EEPMOV命令に展開

```
     LDM.L      @SP+,(ER4-ER6)
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	30	22	30	22	22
指定あり	28	24	26	22	22

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	28	26	22
指定あり	22	22	18

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	117	102	270	224	256
指定あり	58	55	226	210	168

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	66	66	57
指定あり	24	24	24

5.4.7 SPEED オプション

■説明

通常、本コンパイラではコードサイズを優先したオブジェクトを出力します。しかし、
このオプションを指定すると、実行処理速度を優先したオブジェクトを出力します。
この指定により、サイズよりもスピードが優先されたコードが出力されるのは次の項目です。

内容	参考
レジスタ退避/回復コードのスピード優先コード展開	5.4.7(1)
シフト式のスピード優先コード展開	5.4.7(2)
構造体、double 型データの代入コード展開	5.4.7(3)
関数のインライン展開	5.4.7(4)
ループ式のスピード優先コード展開	5.4.7(5)
switch **文のスピード優先コード展開**	5.4.7(6)
実行時ルーチン呼び出しの抑止	5.4.7(7)

これらの項目は、個別に指定が可能です。

■指定方法

ダイアログメニュー：**コンパイラタブカテゴリ:[最適化]** の**スピード優先最適化**
コマンドライン　　：*speed*

(1) レジスタ退避／回復コードのスピード優先コード展開

サイズ効率	×	実行速度	○

■説明

通常、関数の入口／出口では、関数内で使用するレジスタを退避／回復します。このとき、H8/300H、H8/300 シリーズでは退避／回復するレジスタの数が 3 個以上のときに、実行時ルーチンを呼び出してレジスタの退避／回復を行います。

実行時ルーチンを使用するとオブジェクトサイズが小さくなりますが、関数呼び出し処理と、不要なレジスタまで退避／回復する場合があるため、実行速度が遅くなります。実行時ルーチンを呼び出さずに必要なレジスタのみ退避/回復するような指定をすると、オブジェクトサイズは増大しますが、実行処理速度が向上します。

■指定方法

ダイアログメニュー：**コンパイラタブカテゴリ:[最適化]** の**スピード優先最適化オプション:レジスタ退避・回復を push,pop 展開**

コマンドライン　　：*speed=register*

■使用例

関数 sub を定義します。なお、CPU/動作モードは 300HA を指定しています。

（C/C++プログラム）

```
long a,b;
long sub(char c1,short s2,short s3)
{
   s3=a+b;
   return (c1+s2+s3);
}
```

レジスタの退避／回復で呼び出される実行時ルーチンは最適化の有無、引数渡し用レジスタ数により、異なります。

（アセンブリ展開コード）

指定なし

```
_sub:
     JSR       @$sp_regsv$3:24

     MOV.B     R0L,R5L
     MOV.W     @_a+2:24,R1
     MOV.W     @_b+2:24,R2
     ADD.W     R2,R1
     EXTS.W    R5
     ADD.W     E0,R5
     ADD.W     R1,R5
     EXTS.L    ER5
     MOV.L     ER5,ER0
     JMP       @$spregld2$3:24
```

指定あり

```
_sub:
     PUSH.L    ER5
     PUSH.W    R2
     MOV.B     R0L,R5L
     MOV.W     @_a+2:24,R1
     MOV.W     @_b+2:24,R2
     ADD.W     R2,R1
     EXTS.W    R5
     ADD.W     E0,R5
     ADD.W     R1,R5
     EXTS.L    ER5
     MOV.L     ER5,ER0
     POP.W     R2
     POP.L     ER5
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	24	20	28	24	38
指定あり	24	20	28	24	44

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	24	24	20
指定あり	24	24	20

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	18	15	48	42	134
指定あり	18	15	48	42	94

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	17	15	14
指定あり	17	15	14

(2) シフト式のスピード優先コード展開

サイズ効率	×	実行速度	○

■説明

シフト演算のオブジェクトコードをサイズよりスピード優先で生成します。

■指定方法

ダイアログメニュー：**コンパイラタブカテゴリ:[最適化]** の**スピード優先最適化オプション:シフト演算の高速化**
コマンドライン　　：*speed=shift*

■使用例

変数 a を複数回シフトします。
（C/C++プログラム）

```
unsigned char a=0x80;
int dat;
void main(void)
{
    a>>=dat;
}
```

（コンパイル結果アセンブリ展開コード）

指定なし

```
_main:
      MOV.L       #_a,ER0
      MOV.W       @_dat:32,R1
      JSR         @$DSRUC$3:24
                  ↑実行時ルーチン呼び出しで処理
      RTS
```

指定あり

```
_main:
            MOV.B       @_a:32,R0L
            MOV.B       @_dat+1:32,R0H
L5:
            DEC.B       R0H
            BMI         L8:8
            SHLR.B      R0L
            BRA         L7:8
L6:
            MOV.B       R0L,@_a:32
            RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	25	19	19	15	15
指定あり	49	43	29	23	23

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	25	25	19
指定あり	25	25	19

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	39	33	78	64	64
指定あり	29	25	40	32	32

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	14	14	12
指定あり	14	14	12

(3) 構造体、double 型データの代入コード展開

サイズ効率	△	実行速度	○

■説明

構造体や double 型データの代入を行うと、通常（構造体のサイズが小さいときを除く）実行時ルーチンを呼び出すコードが出力されます。そのため実行時ルーチンを呼び出さずに処理するようにすると、実行速度が向上します。

■指定方法

ダイアログメニュー：**コンパイラタブカテゴリ:[最適化]** の**スピード優先最適化オプション:構造体代入式の高速化**
コマンドライン　　：*speed=struct*

■使用例

構造体 s2 を s1 へ代入します。

（C/C++プログラム）

```
struct S{
    unsigned char cc;
    short ss;
    long ll;
}s1,s2;
void main(void)
{
    s1=s2;
}
```

（コンパイル結果アセンブリ展開コード）

指定なし

```
_main:
     PUSH.L     ER2
     MOV.L      #_s2,ER0
     MOV.L      #_s1,ER1
     SUB.L      ER2,ER2
     MOV.B      #8,R2L
     JSR        @$MVN$3:24

     POP.L      ER2
     RTS
```

↑実行時ルーチン処理

指定あり

```
_main:
     PUSH.L     ER2
     MOV.L      #_s2,ER0
     MOV.L      #_s1,ER1
     MOV.L      @ER0+,ER2
     MOV.L      ER2,@ER1
     MOV.L      @ER0,ER2
     MOV.L      ER2,@(4:16,ER1)
     POP.L      ER2
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	18	14	30	22	22
指定あり	40	32	40	36	34

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	22	22	18
指定あり	22	22	18

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	49	44	244	198	220
指定あり	39	32	78	72	124

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	18	14	14
指定あり	18	14	14

(4) 関数のインライン展開

サイズ効率	×	実行速度	○

■説明

オプションでインライン展開を指定すると、サイズの小さい関数がインライン展開され、実行速度が向上します。ただし、下記条件に 1 つでも該当するとインライン展開されません。

・#pragma inline 指定より前に関数の定義がある。
・可変引数を持つ。
・引数のアドレスを参照している。
・実引数と仮引数の型が不一致である。
・インライン展開の制限サイズを超えている。

インライン展開の制限サイズとは、指定した関数のノード数のことです。

このノード数とは、コンパイラ内部処理で使用する処理単位ですが、1～65535 の範囲で選択することができます。一般に小さな数を指定した場合小さな関数をインライン展開し、大きな値を指定した場合大きなサイズの関数もインライン展開の対象とします。

デフォルトは 105 です。

#pragma inline 指定した関数は、インライン展開の制限サイズにかかわらず、インライン展開を行います。

■指定方法

ダイアログメニュー：**コンパイラタブカテゴリ:[最適化]** の**スピード優先最適化オプション:インライン展開**
コマンドライン　　：*speed=inline[=(node)]*

■使用例

関数 func を呼び出します。

（C/C++プログラム）

```
extern long a,b;
void func(void);
void sub(void)
{
   func();
   a+=2;
}
void func(void)
{
   a++;
}
```

（コンパイル結果アセンブリ展開コード）

指定なし

```
_sub:
     BSR       _func:8      ←関数呼び出し

     MOV.L     #_a,ER0
     MOV.L     @ER0,ER1
     INC.L     #2,ER1
     MOV.L     ER1,@ER0
     RTS
_func:
     MOV.L     #_a,ER0
     MOV.L     @ER0,ER1
     INC.L     #1,ER1
     MOV.L     ER1,@ER0
     RTS
```

指定あり

```
_sub:
     MOV.L     @_a:32,ER0
     INC.L     #1,ER0
     INC.L     #2,ER0
     MOV.L     ER0,@_a:32
     RTS
_func:
     MOV.L     #_a,ER0
     MOV.L     @ER0,ER1
     INC.L     #1,ER1
     MOV.L     ER1,@ER0
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	44	36	40	36	38
指定あり	58	46	52	46	42

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	30	28	24
指定あり	34	34	28

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	41	34	78	68	182
指定あり	31	26	58	52	166

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	23	23	21
指定あり	15	15	14

■備考

呼び出し側と同じファイル中に呼び出される側の関数がある場合で、他のファイルで呼び出されていないときは、関数の宣言に static を指定するとその関数の外部定義を生成しないため、サイズが小さくなります。

(5) ループ式のスピード優先コード展開

サイズ効率	△	**実行速度**	○

■説明

ファイル中の下記条件をすべて満たすループが展開されたコードで出力されます。

- ループの初期値が定数
- ループの終了判定が定数
- ループの反復回数が 3 の倍数または偶数
- ループの中に goto ラベルがない
- ループ内には式のみで、その数が 10 以下
- 最適化が指定されている

しかし、ループを展開するとプログラムサイズが増大します。特定のループを高速にしたい場合は、プログラム上でループ展開したコーディングをしてください。

■指定方法

ダイアログメニュー：**コンパイラタブカテゴリ:[最適化]** の**スピード優先最適化オプション:ループ文での帰納変数削除**
コマンドライン　　：*speed=loop*

■使用例

配列 a の内容をゼロクリアします。

（C/C++プログラム）

```
int a[10];

void f(void)
{
   int i;

   for (i=0;i<10;i++)
      a[i]=0;
}
```

（コンパイル結果アセンブリ展開コード）

指定なし

```
_f:
     PUSH.L     ER6
     SUB.W      R6,R6
     SUB.W      R1,R1
L6:
     EXTS.L     ER6
     MOV.L      ER6,ER0
     SHLL.L     ER0
     MOV.W      R1,@(_a:32,ER0)
     INC.W      #1,R6
     CMP.W      #10:16,R6
     BLT        L6:8
     POP.L      ER6
     RTS
```

指定あり

```
_f:
     MOV.L      #_a,ER1
     SUB.L      ER0,ER0

L6:
     MOV.W      R0,@ER1
     INC.W      #1,E0
     INC.L      #2,ER1
     MOV.W      R0,@ER1
     INC.W      #1,E0
     INC.L      #2,ER1
     CMP.W      #10,E0
     BLT        L6:8
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	34	26	36	22	28
指定あり	38	28	40	30	40

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	20	20	18
指定あり	36	36	32

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	132	103	294	212	244
指定あり	88	72	162	138	244

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	95	87	87
指定あり	75	71	71

■備考

この指定をした方が時として、サイズの上で優れたコードになることもあります。
オプションのチューニングのときなどに、オプションを指定したり、外したりして試してみてください。

(6) switch 文のスピード優先コード展開

サイズ効率	△	実行速度	○

■説明

switch 文の展開方式を実行サイクル数の少ない方式で出力します。

switch 文の展開方式にはテーブル方式と if-then 方式とがあります。

通常では、サイズ優先でどちらの方法が適しているか、コンパイラが判定します。

If･then 方式は switch 文の評価式の値と case のラベルの値を比較し、一致すれば case ラベルの文へ飛ぶ処理を case ラベルの回数繰り返す展開方式です。この展開方式は 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のサイズが増大します。

一方、テーブル方式は case ラベルの飛び先をジャンプテーブルに確保し、1 回のジャンプテーブルの参照で switch 文の評価式と一致する case ラベルの文へ飛び超す展開方式です。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が増えますが、実行速度は常に一定です。

SPEED オプションを指定すると、これらを考慮して、処理速度が高速な方式を選択します。

■指定方法

ダイアログメニュー：**コンパイラタブカテゴリ:[最適化]** の**スピード優先最適化オプション:switch 文の高速化**

コマンドライン　　：*speed=switch*

■使用例

変数 a の値を入れ替えます。

（C/C++プログラム）

```
extern unsigned a;
void sub(void)
{
    switch(a){
    case 0: a=1;break;
    case 2: a=2;break;
    case 4: a=3;break;
    case 6: a=4;break;
    case 8: a=5;break;
    case 10: a=6;break;
    case 12: a=7;break;
    case 14: a=8;break;
    case 16: a=9;break;
    case 18: a=10;break;
    case 20: a=11;break;
    }
}
```

（コンパイル結果アセンブリ展開コード）

指定なし

```
_sub:MOV.L      #_a:32,ER1
     MOV.W      @ER1,R0
     MOV.B      R0H,R0H
     BNE        L16:8
     CMP.B      #0:8,R0L
     BEQ        L5:8
     CMP.B      #2:8,R0L
     BEQ        L6:8
     CMP.B      #4:8,R0L
     BEQ        L7:8
     CMP.B      #6:8,R0L
     BEQ        L8:8
     CMP.B      #8:8,R0L
     BEQ        L9:8
     CMP.B      #10:8,R0L
     BEQ        L10:8
     CMP.B      #12:8,R0L
     BEQ        L11:8
     CMP.B      #14:8,R0L
     BEQ        L12:8
     CMP.B      #16:8,R0L
     BEQ        L13:8
     CMP.B      #18:8,R0L
     BEQ        L14:8
     CMP.B      #20:8,R0L
     BEQ        L15:8
     RTS
L5:  MOV.W      #1:16,R0
     BRA        L26:8
L6:  MOV.W      #2:16,R0
     BRA        L26:8
L7:  MOV.W      #3:16,R0
     BRA        L26:8
L8:  MOV.W      #4:16,R0
     BRA        L26:8
L9:  MOV.W      #5:16,R0
     BRA        L26:8
L10: MOV.W      #6:16,R0
     BRA        L26:8
L11: MOV.W      #7:16,R0
     BRA        L26:8
L12: MOV.W      #8:16,R0
     BRA        L26:8
L13: MOV.W      #9:16,R0
     BRA        L26:8
L14: MOV.W      #10:16,R0
     BRA        L26:8
L15: MOV.W      #11:16,R0
L26: MOV.W      R0,@ER1
L16: RTS
```

指定あり

```
_sub:MOV.L      #_a,ER1
     MOV.W      @ER1,R0
     CMP.W      #20,R0
     BHI        L18:8
     EXTU.L     ER0
     MOV.B      @(L19:32,ER0),R0L
     EXTU.W     R0
     EXTU.L     ER0
     ADD.L      #L7,ER0
     JMP        @ER0
L5:  MOV.W      #1,R0
     BRA        L27:8
L6:  MOV.W      #2,R0
     BRA        L27:8
L7:  MOV.W      #3,R0
     BRA        L27:8
L8:  MOV.W      #4,R0
     BRA        L27:8
L9:  MOV.W      #5,R0
     BRA        L27:8
L10: MOV.W      #6,R0
     BRA        L27:8
L11: MOV.W      #7,R0
     BRA        L27:8
L12: MOV.W      #8,R0
     BRA        L27:8
L13: MOV.W      #9,R0
     BRA        L27:8
L14: MOV.W      #10,R0
     BRA        L27:8
L15: MOV.W      #11,R0
L27: MOV.W      R0,@ER1
L16: RTS
     .SECTION   C,DATA,ALIGN=2
L17: .DATA.B    L5-L5
     .DATA.B    L16-L5
     .DATA.B    L6-L5
     .DATA.B    L16-L5
     .DATA.B    L7-L5
     .DATA.B    L16-L5
     .DATA.B    L8-L5
     .DATA.B    L16-L5
     .DATA.B    L9-L5
     .DATA.B    L16-L5
     .DATA.B    L10-L5
     .DATA.B    L16-L5
     .DATA.B    L11-L5
     .DATA.B    L16-L5
     .DATA.B    L12-L5
     .DATA.B    L16-L5
     .DATA.B    L13-L5
     .DATA.B    L16-L5
     .DATA.B    L14-L5
     .DATA.B    L16-L5
     .DATA.B    L15-L5
     .DATAB.B   1,0
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	136	120	126	114	120
指定あり	136	120	126	114	120

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	118	118	108
指定あり	118	118	108

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	44	33	66	52	66
指定あり	44	33	66	52	66

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	23	24	18
指定あり	23	24	18

■備考

この例の場合は、サイズ、スピード両面で共に優れたコードとなります。

これは、テーブル方式に展開されたためで、a の値によってはこのオプションを指定しない方が、優れたコードとなる場合があります。

(7) 実行時ルーチン呼び出し抑止

サイズ効率	×	実行速度	○

■説明

四則演算、比較、代入式を実行時ルーチンを使用しないコードで展開するオプションがあります。（一部の式で対象外になるものがあります。）

■指定方法

ダイアログメニュー：**コンパイラタブカテゴリ:[最適化]** の**スピード優先最適化オプション:四則演算、比較、代入式の高速化**

コマンドライン　　：*speed=expression*

■使用例

乗算を行います。

（C/C++プログラム）

```
long a,b;
char c;
void main()
{
   a=b*c;
}
```

（コンパイル結果アセンブリ展開コード）

指定なし

```
_main:

     MOV.B      @_c:32,R0L
     EXTS.W     R0
     EXTS.L     ER0
     MOV.L      @_b:32,ER1
     JSR        @$MULL$3:24









     MOV.L      ER0,@_a:32

     RTS
```

指定あり

```
_main:
     STM.L      (ER2-ER3),@-SP
     MOV.B      @_c:32,R0L
     EXTS.W     R0
     EXTS.L     ER0
     MOV.L      @_b:32,ER1
     MOV.W      E0,R2
     MULXU.W    R1,ER2
     MOV.W      E1,R3
     MULXU.W    R0,ER3
     MULXU.W    R1,ER0
     ADD.W      R2,E0
     ADD.W      R3,E0
     MOV.L      ER0,@_a:32
     LDM.L      @SP+,(ER2-ER3)
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	32	26	32	26	38
指定あり	52	46	48	42	46

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	30	30	24
指定あり	30	30	24

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	63	57	180	168	410
指定あり	54	50	266	248	366

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	18	18	17
指定あり	18	18	17

5.4.8 グローバル変数にレジスタを割り付け

サイズ効率	△	実行速度	△

■説明

使用頻度の高い外部変数をレジスタに割り付けることにより、アクセスコードが短縮されます。
ただし、I/O などの、最適化をしない外部変数は割り付けられません。
外部変数が割り付くことができるレジスタは次のようになっています。

ER4	E4	R4H	R4L
ER5	E5	R5H	R5L

CPU が 300 の場合は R4 と R5 です。
【書式】
#pragma global_register (<変数名>=<レジスタ名>[,<変数名>=<レジスタ名>...])

■使用例

1 バイトデータと 2 バイトデータをレジスタに割り付けます。
（C/C++プログラム）

```
#pragma global_register (a=R4,b=R5L)
int a; char b;
void func();
void main()
{
    a=10;
    b=20;
    func();
}
void func()
{
    a++;
    b-=2;
}
```

（アセンブリ展開コード）

指定なし

```
_main:
    MOV.W      #10,R0
    MOV.W      R0,@_a:32
    MOV.B      #20,R0L
    MOV.B      R0L,@_b:32
_func:
    MOV.L      #_a,ER0
    MOV.W      @ER0,R1
    INC.W      #1,R1
    MOV.W      R1,@ER0
    MOV.L      #_b,ER0
    MOV.B      @ER0,R1L
    ADD.B      #-2,R1L
    MOV.B      R1L,@ER0
    RTS
    .SECTION    B,DATA,ALIGN=2
_a:.RES.W     1
_b:.RES.B     1
```

指定あり

```
_main:
    MOV.W      #10,R4
    MOV.B      #20,R5L
_func:
    INC.W      #1,R4
    ADD.B      #-2,R5L
    RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	52	40	48	40	40
指定あり	20	20	16	16	16

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	38	36	28
指定あり	18	16	16

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	37	30	70	60	60
指定あり	15	14	26	24	24

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	21	21	18
指定あり	12	12	12

■使用上の注意事項

(a) #pragma global_register宣言後の変数定義・変数宣言が対象になります。
(b) 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。double型の変数は指定できません。
(c) 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。
(d) 指定された変数の（ファイル内にレジスタ指定のない）リンク先からの参照は保証されません。
(e) 割り込み関数内での設定・参照は保証されません。
(f) 変数、レジスタの重複指定、#pragma abs8、#pragma abs16との二重指定はできません。

また、この指定を行ったときは、ライブラリをモジュール間最適化の対象にすることができません。すべてのライブラリ関数をモジュール間最適化の対象外にしてください。
モジュール間最適化の対象の外し方は次のようになります。

【PC 版】
<HEW1.2>
ライブラリのモジュール間最適化を行う際に解凍してできたライブラリと同名のディレクトリを削除する。
<HEW2.0 以降>
標準ライブラリ構築ツールの**カテゴリ:[その他] ユーザ指定オプション**にて preinclude オプションで#pragma global_register 宣言を含むヘッダファイルをインクルード指定する。

【UNIX 版】
ライブラリと同名のディレクトリの名称を変更する。

5.4.9 関数の入口／出口でレジスタの退避／回復コード出力を制御

サイズ効率	○	実行速度	○

■説明

コンパイラはすべての関数に対し、使用するレジスタを入口で退避し、出口で回復します。

main 関数や関数呼び出しだけの関数は、レジスタ退避／回復処理を制御することで、レジスタ退避／回復コードを縮小することができます。

#pragma regsave を指定するとすべてのレジスタを退避／回復します。また、関数呼び出しをまたいで関数呼び出し前後で値を保証するレジスタを割り付けません。

#pragma noregsave を指定すると関数内でのレジスタの使用／未使用にかかわらずレジスタの退避／回復は抑止されます。

【書式】

#pragma regsave (<関数名>[,…])
#pragma noregsave (<関数名>[,…])

■使用例

関数 regf から関数 noregf を呼び出します。

（C/C++プログラム）

指定なし

```
void regf();
void noregf(int);
void func();

extern int X,Y,Z,XX;
void regf(void)
{
    int A=X;
    Y=A;
    noregf(X);
    Z=A;
}
void noregf(int P)
{
    int B=P;
    Y=B;
    func(X);
    Z=B;
}
```

指定あり

```
#pragma regsave (regf)
#pragma noregsave (noregf)
void regf();
void noregf(int);
void func();

extern int X,Y,Z,XX;
void regf(void)
{
    int A=X;
    Y=A;
    noregf(X);
    Z=A;
}
void noregf(int P)
{
    int B=P;
    Y=B;
    func(X);
    Z=B;
}
```

（アセンブリ展開コード）

指定なし

```
_regf:
     PUSH.W     R6
     MOV.W      @_X:32,R6
     MOV.W      R6,@_Y:32
     MOV.W      R6,R0
     BSR        _noregf:8
     MOV.W      R6,@_Z:32
     POP.W      R6
     RTS
_noregf:
     PUSH.W     R6
     MOV.W      R0,R6
     MOV.W      R6,@_Y:32
     MOV.W      @_X:32,R0
     JSR        @_func:24
     MOV.W      R6,@_Z:32
     POP.W      R6
     RTS
```

指定あり

```
_regf:
     STM.L      (ER2-ER3),@-SP
     STM.L      (ER4-ER6),@-SP
     MOV.W      @_X:32,R6
     MOV.W      R6,@_Y:32
     MOV.W      R6,R0
     PUSH.W     R6
     BSR        _noregf:8
     POP.W      R6
     MOV.W      R6,@_Z:32
     LDM.L      @SP+,(ER4-ER6)
     LDM.L      @SP+,(ER2-ER3)
     RTS
_noregf:
     MOV.W      R0,R6
     MOV.W      R6,@_Y:32
     MOV.W      @_X:32,R0
     JSR        @_func:24
     MOV.W      R6,@_Z:32
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	66	52	66	52	52
指定あり	80	66	68	54	54

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	68	66	52
指定あり	78	76	62

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	66	54	132	108	108
指定あり	91	79	266	232	190

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	46	43	38
指定あり	60	58	60

5.4.10 関数のインライン展開を指定

サイズ効率	×	実行速度	○

■説明

処理速度を向上させるために、関数を呼び出す代わりに、呼び出す側の関数内で展開するというインライン展開指定があります。

この指定には 2 つの方法があります。

(1) **拡張機能での指定**

【書式】

#pragma inline (<関数名>[,…])

(2) **オプションでの指定**

ダイアログメニュー：**コンパイラタブカテゴリ:[最適化] スピード優先最適化オプション:最大値**

コマンドライン　　：*speed=inline[=(node)]*

通常、関数を呼び出すと、JSR、BSR 命令が出力されますが、インライン展開を指定すると、呼び出した場所で直接コードが展開されます。そのため、関数呼び出し時の JSR、BSR 命令と関数から戻るときの RTS 命令が不要となり、実行速度が向上します。

■使用例

関数 func をインライン展開します。

（C/C++プログラム）

#pragma 文での指定の場合

```
#pragma inline func
int a,b;
void func()
{
   a+=b;
}
void main()
{
   a=0;
   func();
}
```

（アセンブリ展開コード）

指定なし

```
_func:
     MOV.W     @_b:32,R0
     MOV.L     #_a,ER1
     MOV.W     @ER1,E0
     ADD.W     R0,E0
     MOV.W     E0,@ER1
     RTS
_main:
     SUB.W     R0,R0
     MOV.W     R0,@_a:32
     BRA       _func:8
```

指定あり

```
_func:
     MOV.W     @_b:32,R0
     MOV.L     #_a,ER1
     MOV.W     @ER1,E0
     ADD.W     R0,E0
     MOV.W     E0,@ER1
     RTS
_main:
     SUB.W     E0,E0
     MOV.W     @_b:32,R0
     ADD.W     R0,E0
     MOV.W     E0,@_a:32
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	32	24	30	24	24
指定あり	36	26	38	30	30

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	22	22	16
指定あり	28	28	20

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	25	20	48	40	40
指定あり	13	10	30	24	24

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	16	16	15
指定あり	11	11	10

■備考および注意事項

(1) #pragma inlineは、関数本体の定義前に指定してください。
最適化を指定しない場合は、本指定が無効になります。#pragma指定は有効です。
以下の関数はインライン展開しません。

- 可変引数を持つ。
- 引数のアドレスを参照している。
- 実引数と仮引数の型が不一致である。
- インライン展開関数を呼び出している。
- インライン展開の制限サイズを超えている。

(2) インライン展開対象となる関数をstatic指定すると、呼ばれる側でのみ展開されるため、サイズ効率が良くなります。この場合、インライン展開関数は同一ファイル内でのみ使用されることとなります。

5.4.11 8 ビット絶対アドレス領域の活用

サイズ効率	○	**実行速度**	○

■説明

H8S、H8/300 シリーズには 8 ビット絶対アドレス領域があります。この領域に頻繁にアクセスするバイトデータを割り付けることにより、8 ビット絶対アドレス形式でアクセスすることができ、通常の絶対アドレス形式でアクセスするときよりも、ROM 効率、RAM 効率および実行処理速度を向上させることができます。

この指定には 2 つの方法があります。

(1) 拡張機能での指定

【書式】

#pragma abs8 (<変数名または構造体名、配列名>[,…])

(2) オプションでの指定

ダイアログメニュー：**コンパイラタブカテゴリ:[最適化] データアクセス:@aa:8**
コマンドライン　　：*abs8*

#pragma abs8 では、8 ビット絶対アドレス形式でアクセスしたい変数を指定することができます。
また、オプションで指定した場合、ファイル内の 1 バイトデータすべてが 8 ビット絶対アドレス形式アクセスの対象となります。
各 CPU／動作モードごとの 8 ビット絶対アドレス領域の範囲を記述します。

CPU **種別**	**アドレス空間サイズ**	8 **ビット絶対アドレス領域**
H8SX **マキシマムモード**	32	H'FFFFFF00〜H'FFFFFFFF
H8SX **アドバンストモード**	28	H'FFFFF00〜H'FFFFFFF
H8SX **ミドルモード**	24	H'FFFF00〜H'FFFFFF
H8S/2600 **アドバンストモード** H8S/2000 **アドバンストモード** H8/300H **アドバンストモード**	20	H'FFF00〜H'FFFFF
H8SX **ノーマルモード** H8S/2600 **ノーマルモード** H8S/2000 **ノーマルモード** H8/300H **ノーマルモード** H8/300	16	H'FF00〜H'FFFF

■使用例

8 ビット絶対アドレス領域に割り付けた変数 a、b、c をアクセスします。
（C/C++プログラム）
#pragma 文での指定の場合

```
#pragma abs8 (a,b,c)

const char a=1;
     char b=1;
     char c;
void func(void)
{
   c=b=a;
}
```

（コンパイル結果アセンブリ展開コード）

指定なし

```
_func:
   MOV.B      #1,R0L
   MOV.B      R0L,@_b:32
   MOV.B      R0L,@_c:32
   RTS
   .SECTION   C,DATA,ALIGN=2
_a: .DATA.B    H'01
   .SECTION   D,DATA,ALIGN=2
_b: .DATA.B    H'01
   .SECTION   B,DATA,ALIGN=2
_c: .RES.B     1
```

指定あり

```
_func:
   MOV.B      #1,R0L
   MOV.B      R0L,@_b:8
   MOV.B      R0L,@_c:8
   RTS
   .SECTION   $ABS8C,DATA,ALIGN=2
_a: .DATA.B    H'01
   .SECTION   $ABS8D,DATA,ALIGN=2
_b: .DATA.B    H'01
   .SECTION   $ABS8B,DATA,ALIGN=2
_c: .RES.B     1
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	18	14	18	14	14
指定あり	10	10	10	10	10

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	16	16	12
指定あり	10	10	10

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	14	11	28	22	22
指定あり	10	9	20	18	18

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	10	10	10
指定あり	9	9	9

■備考および注意事項

すでに宣言済みのデータを#pragma abs8 指定することはできません。
本指定は、1 バイトの外部変数にのみ有効です。
リンク時に$ABS8 ではじまる各セクションを 8 ビット絶対アドレス領域に割り付けます。

5.4.12 16 ビット絶対アドレス領域の活用

サイズ効率	○	実行速度	○

■説明

H8S、H8/300 シリーズには 16 ビット絶対アドレス領域があります。この領域に頻繁にアクセスするデータを割り付けることにより、16 ビット絶対アドレス形式でアクセスすることができ、通常の絶対アドレス形式でアクセスするときよりも、ROM 効率、RAM 効率および実行処理速度を向上させることができます。
この指定には 2 つの方法があります。

(1) **拡張機能での指定**

【書式】
#pragma abs16 (<変数名または構造体名、配列名>[,…])

(2) **オプションでの指定**

ダイアログメニュー：**コンパイラタブカテゴリ:[最適化] データアクセス:@aa:16**
コマンドライン　　：*abs16*

#pragma abs16 では、16 ビット絶対アドレス形式でアクセスしたい変数を指定することができます。オプションで指定した場合、ファイル内のデータすべてを 16 ビット絶対アドレス指定の対象となります。
各 CPU／動作モードごとの 16 ビット絶対アドレス領域を示します。

CPU 種別	アドレス空間サイズ	16 ビット絶対アドレス領域
H8SX マキシマムモード	32	0〜H'7FFF, H'FFFF0000〜H'FFFFFFFF
H8SX アドバンストモード	28	0〜H'7FFF, H'FFF0000〜H'FFFFFFF
H8SX ミドルモード	24	0〜H'7FFF, H'FF0000〜H'FFFFFF
H8S/2600 アドバンストモード H8S/2000 アドバンストモード H8/300H アドバンストモード	20	0〜H'7FFF, H'F0000〜H'FFFFF

■使用例

16 ビット絶対アドレス領域に割り付けた変数 a、b、c をアクセスします。

（C/C++プログラム）

#pragma 文で指定した場合

```
#pragma abs16 (a,b,c)
const int a=1;
      int b=1;
      int c;

void func(void)
{
   c=b=a;
}
```

（コンパイル結果アセンブリ展開コード）

指定なし

```
_main:
    MOV.W    #1,R0
    MOV.W    R0,@_b:32
    MOV.W    R0,@_c:32
    RTS
    .SECTION  C,DATA,ALIGN=2
_a:
    .DATA.W   H'0001
    .SECTION  D,DATA,ALIGN=2
_b:
    .DATA.W   H'0001
    .SECTION  B,DATA,ALIGN=2
_c:
    .RES.W    1
```

指定あり

```
_main:
    MOV.W    #1,R0
    MOV.W    R0,@_b:16
    MOV.W    R0,@_c:16
    RTS
    .SECTION  $ABS16C,DATA,ALIGN=2
_a:
    .DATA.W   H'0001
    .SECTION  $ABS16D,DATA,ALIGN=2
_b:
    .DATA.W   H'0001
    .SECTION  $ABS16B,DATA,ALIGN=2
_c:
    .RES.W    1
```

オブジェクトサイズ表 [**バイト**]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	22	18	22	18	18
指定あり	18	18	18	18	18

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	18	18	14
指定あり	14	14	14

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	15	12	30	24	24
指定あり	13	12	26	24	24

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	10	10	9
指定あり	9	9	9

■備考および注意事項

本指定は CPU／動作モードが H8SXX、H8SXA、H8SXM、2600a、2000a、300ha のときのみ有効です。
すでに宣言済みのデータを#pragma abs16 指定することはできません。
本指定は、外部変数にのみ有効です。
データの出力されるセクション名は#pragma 文にて変更することができます。
リンク時に$ABS16 ではじまる各セクションを 16 ビット絶対アドレス領域に割り付けます。

5.4.13 メモリ間接形式の活用

サイズ効率	○	実行速度	×

■説明

頻繁に使用する関数をメモリ間接形式でアクセスできれば、ROM 効率が向上します。リンク時に関数のアドレスをメモリ間接領域へ格納すると、その関数を呼び出す際にメモリ間接形式で呼び出されます。これにより、実行処理速度は下がりますが、関数を短い命令で呼び出すことができ、プログラムサイズが縮小されます。
この指定には 2 つの方法があります。

(1) 拡張機能での指定

【書式】

```
#pragma indirect (<関数名>[(vect=<ベクタ番号>)] [,…])
__indirect[(vect=<ベクタ番号>)] <型指定子> <関数名>
<型指定子> __indirect[(vect=<ベクタ番号>)] <関数名>
```

(2) オプションでの指定

ダイアログメニュー：**コンパイラタブカテゴリ:[最適化]**の**関数呼び出し:@@aa:8**
コマンドライン　　：*indirect=Normal*

メモリ間接アドレス領域は 00～FF の範囲です。
また、インクルードファイル indirect.h をインクルード指定すると、使用するすべての実行時ルーチンの呼び出しをメモリ間接形式で呼び出します。
さらに、個別に実行時ルーチンをメモリ間接形式で呼び出すこともできます。

■使用例

関数 func をメモリ間接形式で呼び出します。

（C/C++プログラム）

#pragma 文での指定の場合

```
#pragma indirect func
extern void func(int, int);
extern int a,b,c;
int d;
void main(void)
{
    b=0;
    func(a,b);
    func(b,c);
    func(c,a);
    d=c;
}
```

←**#pragma indirect指定**

（コンパイル結果アセンブリ展開コード）

指定なし

```
_main:
    PUSH.L      ER6
    SUB.W       R0,R0
    MOV.W       R0,@_b:32
    MOV.W       R0,E0
    MOV.W       @_a:32,R0
    JSR         @_func:24
    MOV.L       #_c,ER6
    MOV.W       @ER6,E0
    MOV.W       @_b:32,R0
    JSR         @_func:24
    MOV.W       @_a:32,E0
    MOV.W       @ER6,R0
    JSR         @_func:24
    MOV.W       @ER6,R6
    MOV.W       R6,@_d:32
    POP.L       ER6
    RTS
```

指定あり

```
_main:
    PUSH.L      ER6
    SUB.W       R0,R0
    MOV.W       R0,@_b:32
    MOV.W       R0,E0
    MOV.W       @_a:32,R0
    JSR         @@$func:8
    MOV.L       #_c,ER6
    MOV.W       @ER6,E0
    MOV.W       @_b:32,R0
    JSR         @@$func:8
    MOV.W       @_a:32,E0
    MOV.W       @ER6,R0
    JSR         @@$func:8
    MOV.W       @ER6,R6
    MOV.W       R6,@_d:32
    POP.L       ER6
    RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	58	52	70	54	54
指定あり	66	48	68	50	50

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	70	64	50
指定あり	62	62	46

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	75	58	152	118	118
指定あり	78	58	158	118	118

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	49	46	43
指定あり	50	50	44

■備考および注意事項

リンク時に$INDIRECT セクションをメモリ間接形式でアクセスできるメモリ(00～FF)に割り付ける必要があります。

メモリ間接領域は"$INDIRECT"セクションに出力されますが、セクション名を変更したい場合は、#pragma indirect section で指定できます。

5.4.14 拡張メモリ間接形式の活用

サイズ効率	○	実行速度	×

■説明

頻繁に使用する関数をメモリ間接形式でアクセスできれば、ROM 効率が向上しますが、H8SX ではメモリ間接に加え、拡張メモリ間接というアドレッシングモードがあります。

こちらもメモリ間接と同様に ROM 効率が向上します。

この指定には 2 つの方法があります。

(1) 拡張機能での指定

【書式】

<型指定子> __indirect_ex[(vect=<ベクタ番号>)] <関数名>

__indirect_ex[(vect=<ベクタ番号>)] <型指定子> <関数名>

(2) オプションでの指定

ダイアログメニュー：**コンパイラタブカテゴリ:[最適化]**の**関数呼び出し:@@vec:7**

コマンドライン　　：*indirect=Extended*

【拡張メモリ間接アドレス領域】

H8SX ノーマルモード：0x0100-0x01FF 番地

H8SX その他のモード：0x0200-0x03FF 番地

■使用例

関数 func を拡張メモリ間接形式で呼び出します。

キーワードでベクタ番号を指定しない場合は、アドレステーブルのセクション($EXINDIRECT)に出力されます。

ベクタ番号が指定されている場合は、ベクタテーブルのセクション($VECT***)が出力され、リンク時に最適化リンカが自動でセクションを該当アドレスに割り付けます。

（C/C++プログラム）

キーワードで指定の場合

```
__indirect_ex void func(int, int);
extern int a,b,c;
int d;
void main(void)
{
    b=0;
    func(a,b);
    func(b,c);
    func(c,a);
    d=c;
}
```

←__indirect_ex指定

（コンパイル結果アセンブリ展開コード）

指定なし

```
_main:
    STM.L      (ER2-ER3),@-SP
    SUB.W      E0,E0
    MOV.W      E0,@_b:32
    MOV.L      #_func,ER2
    MOV.W      @_a:32,R0
    JSR        @ER2
    MOV.W      @_b:32,R0
    MOV.L      #_c,ER3
    MOV.W      @ER3,E0
    JSR        @ER2
    MOV.W      @_a:32,E0
    MOV.W      @ER3,R0
    JSR        @ER2
    MOV.W      @ER3,@_d:32
    RTS/L      (ER2-ER3)
```

指定あり

```
_main:
    PUSH.L     ER2
    SUB.W      E0,E0
    MOV.W      E0,@_b:32
    MOV.W      @_a:32,R0
    JSR        @@$$func:7
    MOV.W      @_b:32,R0
    MOV.L      #_c,ER2
    MOV.W      @ER2,E0
    JSR        @@$$func:7
    MOV.W      @_a:32,E0
    MOV.W      @ER2,R0
    JSR        @@$$func:7
    MOV.W      @ER2,@_d:32
    RTS/L      ER2
```

オブジェクトサイズ表 [**バイト**]

CPU **種別**	H8SX		
	MAX	ADV	NML
指定なし	82	76	58
指定あり	74	74	54

実行処理速度表 [**サイクル**]

CPU **種別**	H8SX		
	MAX	ADV	NML
指定なし	61	58	55
指定あり	62	62	56

■備考および注意事項

ベクタ番号を指定していない場合、リンク時に$EXINDIRECT セクションを拡張メモリ間接形式でアクセスできるメモリに割り付ける必要があります。

拡張メモリ間接領域は$EXINDIRECT セクションに出力されますが、セクション名を変更したい場合は、#pragma indirect section で変更できます。

5.4.15 ポインタサイズ 2byte 指定

サイズ効率	○	**実行速度**	○

■説明

頻繁に使用する変数を 16 ビット絶対アドレスに配置するとサイズ効率、実行速度共に向上します。

16 ビット絶対アドレスを利用する ABS16 オプション、拡張機能#pragma abs16 もありますが、ABS16 がデータを 16 ビット絶対アドレスに割り付けるのに対し、本機能はデータをアクセスするポインタを 2 バイトにします。

この指定には 2 つの方法があります。

(1) 拡張機能での指定

【書式】

<型指定>__ptr16 * <変数名>

(2) オプションでの指定

ダイアログメニュー：**コンパイラタブカテゴリ:[最適化]**の **2 バイトポインタ**
コマンドライン　　：*ptr16*

通常、データを指すポインタは 4 バイトですが、本オプションまたは拡張機能の使用と共に、変数を 16 ビット絶対アドレスに割り付けるとデータを指すポインタのサイズが 2 バイトになります。

■使用例

変数 b を 2 バイトポインタで参照します。
（C/C++プログラム）
キーワードで指定の場合

```
__abs16 int a;          ←__abs16指定
int __ptr16 *b;         ←__ptr16指定
int c;
void func(void);
void func(void)
{
    b = (int __ptr16 *)&a;
    *b = 10;
    c = *b;
}
```

（コンパイル結果アセンブリ展開コード）
指定なし

```
_func
        MOV.L       #_a:32,@_b:16
        MOV.L       @_b:16,ER0
        MOV.W       #10:8,@ER0
        MOV.L       @_b:16,ER0
        MOV.W       @ER0,@_c:16
        RTS
```

指定あり

```
_func:
        MOV.L       #_a,ER1
        MOV.W       R1,@_b:16
        MOV.W       R1,R0
        EXTS.L      ER0
        MOV.W       #10:8,@ER0
        MOV.W       @_b:16,R0
        EXTS.L      ER0
        MOV.W       @ER0,@_c:16
        RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	36	36	24
指定あり	34	34	24

実行処理速度表 [サイクル]

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	23	19	16
指定あり	22	18	16

■備考および注意事項

本機能は H8SX アドバンストモードおよび H8SX マキシマムモードのみ有効です。
本キーワードを指定する場合は、間接演算子"*"の前に指定してください。

5.4.16 境界調整数、バウンダリ調整指定

サイズ効率	○	実行速度	○

■説明

align オプションを指定すると変数を配置する際、空き領域が小さくなるよう変数の再配置を行います。
さらに、align=4 を指定するとデータをそれぞれ境界調整数が 4、2、1 のセクションに分割します。
（align=4 オプション指定は CPU が H8SX のみ有効です。）
これによりサイズ効率と実行速度(H8SX のみ)が向上します。*

■指定方法

ダイアログメニュー：**コンパイラタブカテゴリ:[オブジェクト]**の**境界調整**
コマンドライン　　：*ALign [=4]（デフォルトは ALign）*
　　NOALign

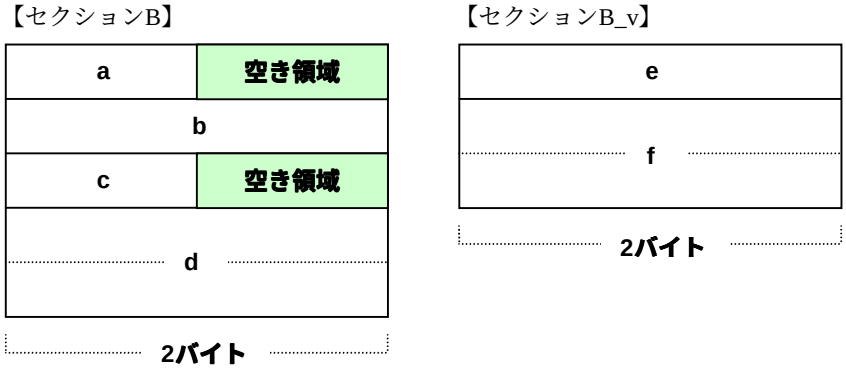
■使用例

下記プログラムをコンパイルしたときの、各オプション指定によるデータ並びの違いを説明します。
（C/C++プログラム）

```
char a;
short b;
char c;
long d;
#pragma section _v
short e;
long f;
#pragma section

void func(void)
{
    a = 127;
    b = 0x7fff;
    c = 30;
    d = 0x7fffffff;
    e = 0x1000;
    f = 0x1ffff;
}
```

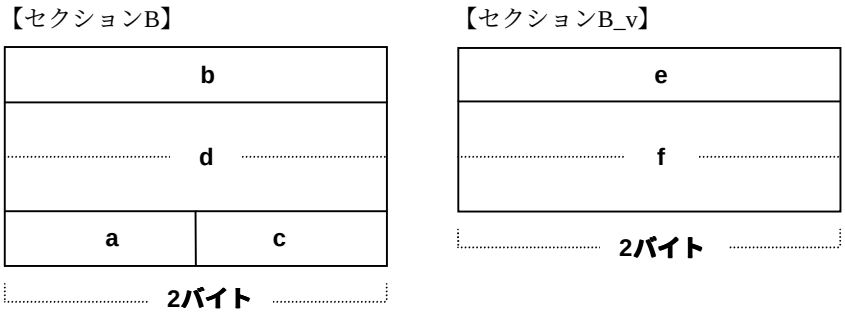
(1) noalign **指定の場合**

セクション B とセクション B_v に変数を宣言順に配置します。
以下のように、境界調整数が 2 のデータは必ず偶数アドレスに割り当てられるので、奇数サイズのデータが前にあると空き領域が発生する可能性があります。



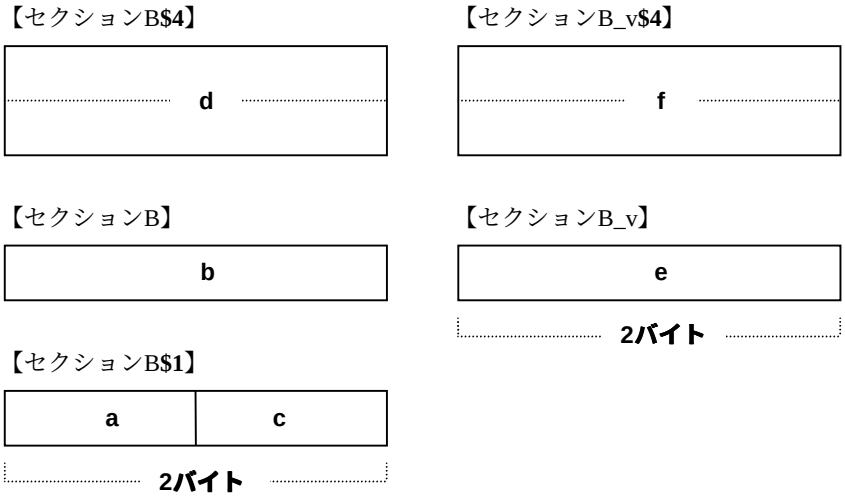
(2) align 指定の場合

セクション B とセクション B_v に空き領域が最小になるように境界調整数が 2 のデータ（1 バイト以外の short,long,float 型など）を配置し、その後に境界調整数が 1 のデータを配置します。
以下のように空き領域が発生しません。



(3) align=4 指定の場合

各セクションごとにデータを 3 つに分類します。
分類方法はサイズが 4 の倍数のデータ、奇数のデータ、それ以外のデータ（4 の倍数以外の偶数データ）に分類します。
分類する際に、セクション名の末尾に **4 の倍数のデータは"$4"、奇数のデータ"$1"、それ以外のデータ（4 の倍数以外の偶数データ）は元のセクション名**にします。
これにより、4 バイト境界に割りついた 4 バイト変数のアクセス速度が向上します。*



【align=4 指定時セクションのアドレス配置】
最適化リンカの start オプションでそれぞれのセクションを任意のアドレスへ明示的に配置する必要があります。
HEW ではダイアログメニュー：**最適化リンカタブカテゴリ:[セクション]**で配置します。
（例：$4 が付くセクションは 4 の倍数アドレスに配置する。$1 は空きが出ないように配置する。）

オブジェクトサイズ表（RAM サイズ）[バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	16	16	16	16	24
指定あり	14	14	14	14	22

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	16	16	16
指定あり	14	14	14

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
指定なし	45	38	90	76	156
指定あり	45	38	90	76	156

CPU 種別	H8SX		
	MAX	ADV	NML
指定なし	28	28	24
指定あり	27	26	23

■備考および注意事項

align=4 は CPU が H8SX のみ有効です。

【注】* 通常 4 バイトのデータはワード命令 2 回でアクセスしますが、align=4 **指定で 4 の倍数アドレスに 4 バイトデータが配置されていて、バス幅が 32 ビットの場合、1 回の命令アクセスでデータをアクセスすることができます。**
バス幅が 16 ビットの場合は、ワード命令 2 回でアクセスするので高速化されません。

5.4.17 モジュール間最適化項目の説明

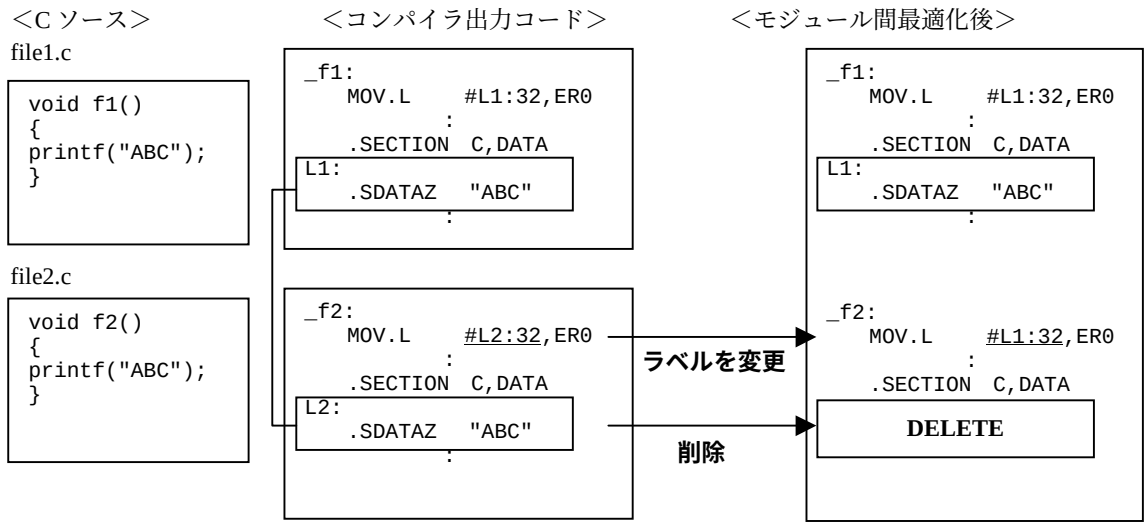
モジュール間最適化ツールは次の最適化機能を持っています。

内容	ダイアログメニュー	サブコマンド	参考
定数／文字列の統合	**Unify strings　または 定数/文字列の統合**	*String_Unify*	5.4.17(1)
未参照変数／関数の削除	**Eliminate dead code　または 未参照シンボルの削除**	*Symbol_delete*	5.4.17(2)
変数アクセスの最適化	**Use short addressing　または 短絶対アドレッシングモード活用**	*Variable_access*	5.4.17(3)
関数アクセスの最適化	**Use indirect call/jump　または 間接アドレッシングモード活用**	*Funcation_call*	5.4.17(4)
レジスタ退避／回復コードの最適化	**Reallocate registers　または レジスタ退避・回復の最適化**	*Register*	5.4.17(5)
共通コードの統合	**Eliminate same code　または 共通コードの統合**	*Same_code*	5.4.17(6)
分岐命令の最適化	**Optimize branches　または 分岐命令の最適化**	*Branch*	5.4.17(7)

以降、それぞれの最適化の詳細を説明します。

(1) 定数／文字列の統合

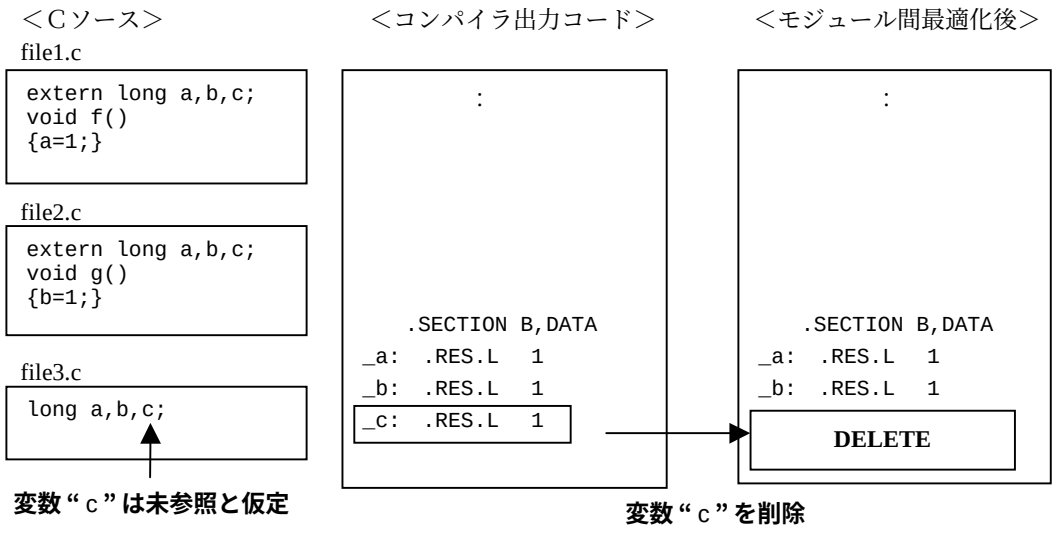
const 属性を持つ定数／文字列に対し、同一値定数および同一文字列の統合をモジュール間にわたって実施します。次に例を示します。



(2) 未参照変数／関数の削除

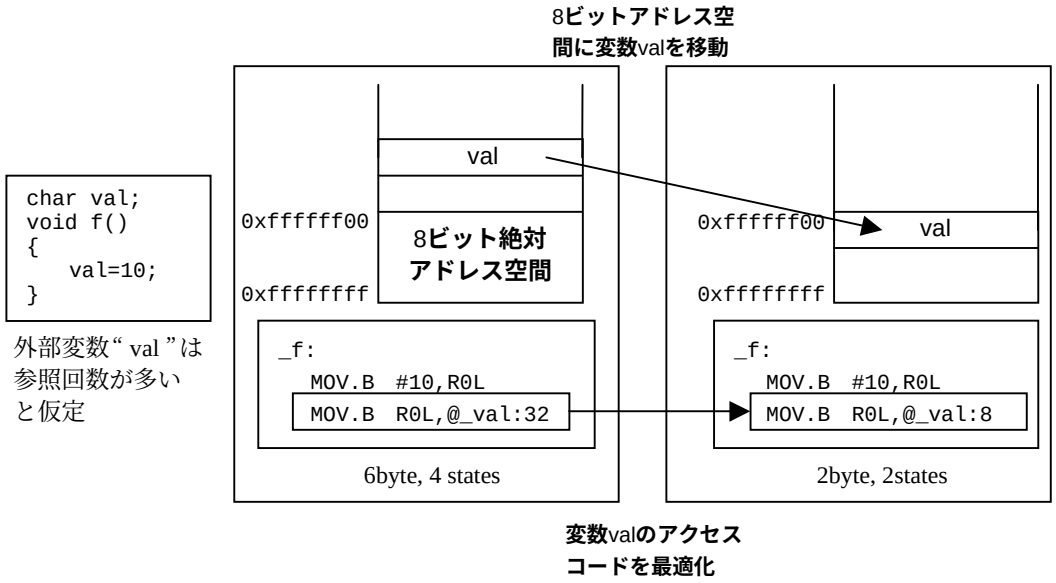
一度も参照のない変数／関数を削除します。この最適化を指定する場合は、必ずエントリ関数の指定をしてください。エントリ関数の指定がないと、本最適化は実行されません。

以下に仕組みを書きます。



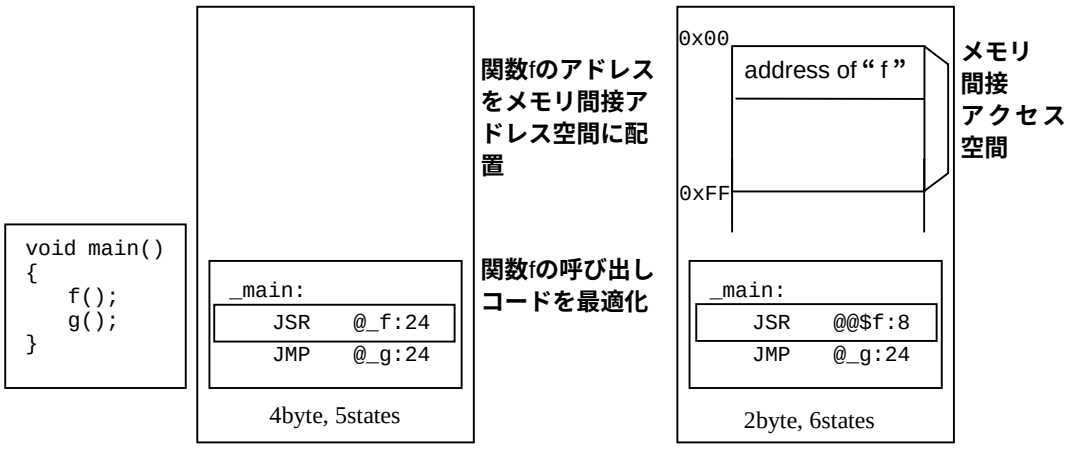
(3) 変数アクセスの最適化

8 ビットおよび 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空きがあれば、アクセス回数の多い変数を割り当て、当該変数のアクセスコード最適化を割り当て、当該変数のアクセスコード最適化を行います。



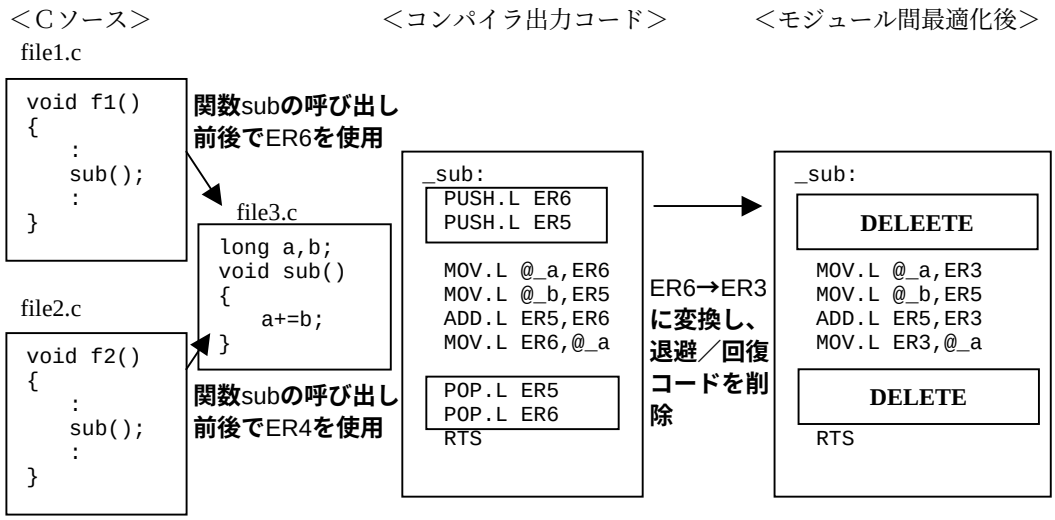
(4) 関数アクセスの最適化

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当てる最適化を行います。



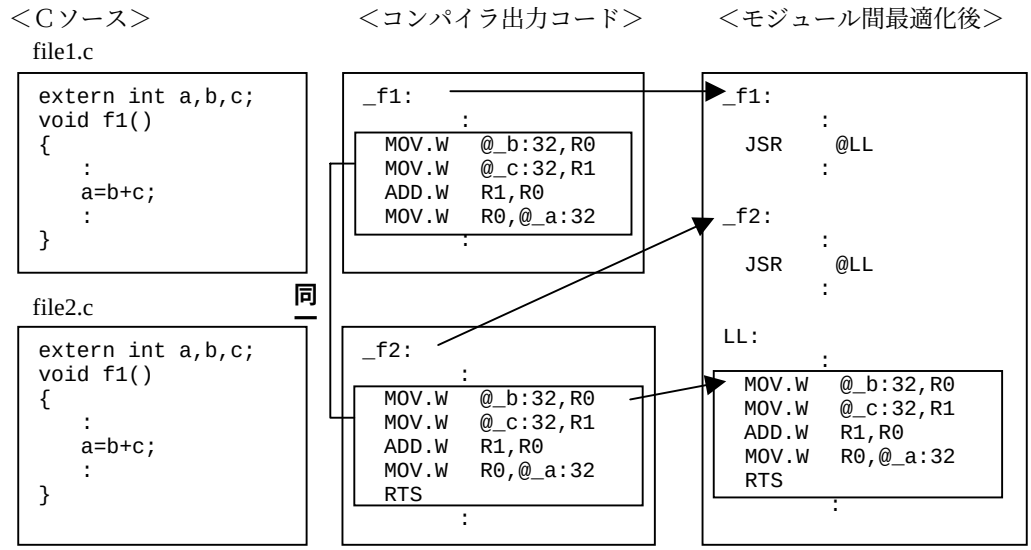
(5) レジスタの再割り付け

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。



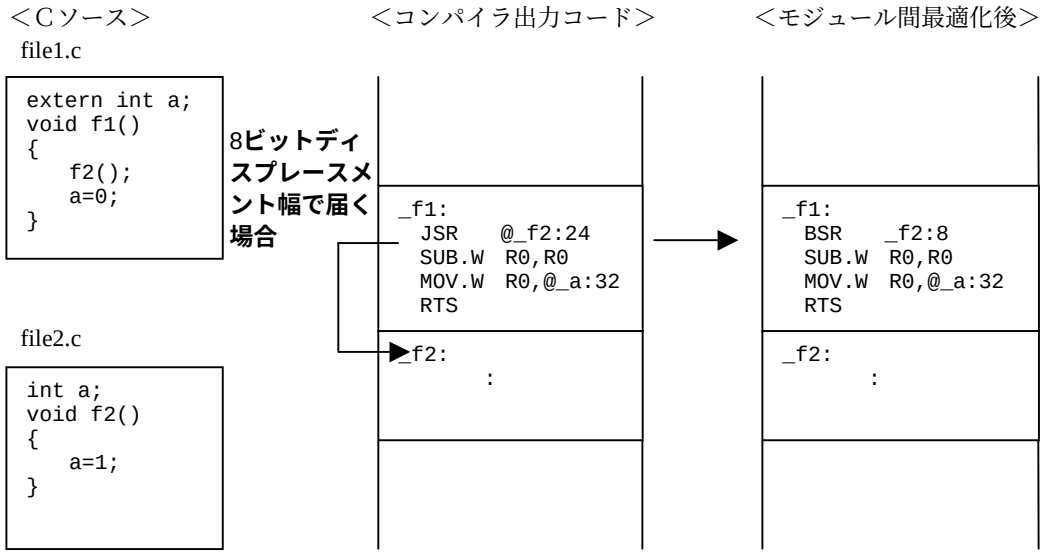
(6) 共通コードの統合

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。



(7) **分岐命令の最適化**

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。また、他の最適化項目をひとつでも実行すると、本最適化は指定の有無にかかわらず、必ず実行します。



5.4.18 モジュール間最適化を抑止する

モジュール間最適化ツールには、指定した最適化を抑止するための機能を用意してあります。

プログラムによっては、該当する最適化を抑止しなければならない場合がありますが、この機能を利用すると、細かい指定ができるため、最小限の最適化抑止をすることができます。

モジュール間最適化ツールが持つ、最適化抑止機能は以下のようになっています。

最適化抑止項目	指定単位	ダイアログメニュー	サブコマンド
未参照シンボル削除抑止	シンボル名	Elimination of dead code または 未参照シンボル削除抑止シンボル	symbol_forbid
共通コード結合抑止	関数名	Elimination of same code または 共通コード統合抑止シンボル	samecode_forbid
短絶対アドレス領域割り付け抑止	変数名	Use of short addressing to または 短絶対アドレッシングモード活用抑止シンボル	variable_forbid
間接アドレス呼び出し抑止	関数名	Use of indirect call/jump to または 間接アドレッシングモード活用抑止シンボル	function_forbid
レジスタ再割り付け抑止	アドレス [+サイズ]	Memory allocation または 最適化抑止アドレス範囲	absolute_forbid

6. 効率の良いプログラミング技法

H8S、H8/300 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いっそう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、ユーザに試して頂きたい方法について記述します。

(i) 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。

(ii) 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。

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 CPU の指定が H8S/2600 シリーズアドバンストモードの場合として記述しています。CPU の指定が異なる場合のみ、その旨を記しています。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。

なお、性能測定は以下の条件で測定しています。

【測定クロスツール】
H8S,H8/300 C/C++ Library Generator (V. 2.01.00.001)
H8S,H8/300 C/C++ Compiler (V. 6.01.00.009)
H8S,H8/300 Assembler (V. 6.01.01.000)
Optimizing Linkage Editor (V. 9.00.02.000)

【オプション指定】
各項目の詳細にオプション指定指示がない限り、デフォルトオプションを設定。

【測定条件】

条件	H8/300,H8/300H	H8S/2600,H8S/2000	H8SX
バス幅	16	16	32
メモリへのアクセスステート	2	1	1
フェッチサイズ	-	-	32

効率的プログラミング技法の一覧を示します。

No.	種類	項目	サイズ効率	処理速度	参照
1	型宣言	1 バイトデータ（char/unsigned char）型の活用	○	○	6.1.1
2		符号なし（unsigned）変数の活用	○	○	6.1.2
3		冗長な型変換の抑止	○	○	6.1.3
4		const 修飾子の活用	○	○	6.1.4
5		変数のサイズをそろえる	○	○	6.1.5
6		ファイル内関数を static 指定する	○	－	6.1.6
7	演算	共通式の統合	○	○	6.2.1
8		条件判定の改善	○	○	6.2.2
9		代入値による条件判定	○	△	6.2.3
10		数学的手法の活用	○	○	6.2.4
11		公式の活用	○	○	6.2.5
12		局所変数の活用	○	○	6.2.6
13		float 型定数には f をつける	○	○	6.2.7
14		シフト演算の定数指定	○	○	6.2.8
15		シフト演算の活用	○	○	6.2.9
16		連続した加算命令の統合	○	○	6.2.10
17	ループ処理	ループカウンタの選択	○	○	6.3.1
18		繰り返し制御文の選択	○	○	6.3.2
19		ループ内不変式のループ外移動	○	○	6.3.3
20		ループ条件のマージ	○	○	6.3.4
21	ポインタ	ポインタ変数の活用	○	○	6.4.1
22	データ構造	データの整合性	○	－	6.5.1
23		データの初期化方法	○	○	6.5.2
24		配列要素の初期化の統合	○	○	6.5.3
25		引数の構造体アドレス渡し	○	○	6.5.4
26		構造体のレジスタ割り付け	○	○	6.5.5
27	関数	関数定義位置の改善	－	○	6.6.1
28		マクロ呼び出し	○	○	6.6.2
29		原型宣言	－	－	6.6.3
30		テイルリカージョン最適化	○	○	6.6.4
31		引数の渡し方法の工夫	○	○	6.6.5
32	分岐	switch 文のテーブル方式への書き換え	○	○	6.7.1
33		case 文の飛び先が同じ場合の記述	○	○	6.7.2
34		直下の関数への分岐	○	○	6.7.3

【記号説明】

○：効率向上

△：変化なし

×：効率低下

－：該当せず

6.1 型宣言

6.1.1 1 バイトデータ（char/unsigned char）型の活用

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

1 バイトサイズで表現が可能なデータは char 型／unsigned char 型で宣言すると ROM 効率および実行処理速度が向上します。

■説明

H8S,H8/300 シリーズ CPU は、バイトサイズのデータを効率よく演算できる命令体系になっています。
このため、あらかじめ char 型／unsigned char 型で宣言することによって、ROM 効率および実行処理速度が向上します。

■使用例

変数 a と 0x80 の論理積を求め、結果を変数 a に格納します。

（改善前のCプログラム）

```
int a;
void func(void)
{
   a&=0x80;
}
```

（改善後のCプログラム）

```
char a;
void func(void)
{
   a&=0x80;
}
```

（改善前のアセンブラ展開コード）

```
_func:
     MOV.L      #_a,ER0
     MOV.W      @ER0,R1
     AND.W      #128,R1
     MOV.W      R1,@ER0
     RTS
_a:
    .RES.W     1
```

（改善後のアセンブラ展開コード）

```
_func:
     MOV.L      #_a,ER0
     MOV.B      @ER0,R1L
     AND.B      #-128,R1L
     MOV.B      R1L,@ER0
     RTS
_a:
     .RES.B     1
```

オブジェクトサイズ表 [**バイト**]

CPU **種別**	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	18	14	16	14	14
改善後	16	12	14	12	12

CPU **種別**	H8SX		
	MAX	ADV	NML
改善前	12	12	10
改善後	10	10	8

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	15	12	28	24	24
改善後	14	11	26	22	22

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	11	11	10
改善後	9	9	9

6.1.2 符号なし（unsigned）変数の活用

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

変数の値が常に正のときは、unsigned で宣言すると、オブジェクト効率および実行処理速度が向上します。

■説明

データをより大きなデータ型へ拡張するときには、データが符号付きであれば符号拡張を、符号なしであれば、ゼロ拡張を行います。H8/300 シリーズ CPU は、データ拡張命令が用意されていませんので、符号付きのデータについては符号判定オブジェクトが必要になります。したがって、変数の値が常に正のときは符号なしで修飾すると、ROM 効率および実行処理速度が向上します。

ただし、H8S および H8/300H シリーズ CPU では、データ拡張命令が用意されているため、変数の値が正のときは、unsigned 宣言をしてもしなくても変わりありません。

■使用例

変数 a を int 型に拡張して、変数 b に代入します。
cpu が 300 でコンパイルした結果を記述します。

（改善前のCプログラム）

```
char a;
int  b;
void func(void)
{
   b=a;
}
```

（改善後のCプログラム）

```
unsigned char a;
int  b;
void func(void)
{
   b=a;
}
```

（改善前のアセンブラ展開コード）

```
_func:
     MOV.B      @_a:16,R0L
     BLD.B      #7,R0L
     SUBX.B     R0H,R0H
     MOV.W      R0,@_b:16
     RTS
```

（改善後のアセンブラ展開コード）

```
_func:
     MOV.B      @_a:16,R0L
     SUB.B      R0H,R0H
     MOV.W      R0,@_b:16
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	16	12	16	12	14
改善後	16	12	16	12	12

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	16	16	12
改善後	16	16	12

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	14	11	28	22	24
改善後	14	11	28	22	22

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	11	11	10
改善後	11	11	10

6.1.3 冗長な型変換の抑止

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

演算は、同一のデータサイズ間で行うと、ROM 効率、および実行処理速度が向上します。

■説明

異なるデータサイズ間で演算を行うと、命令的に無駄な符号拡張やゼロ拡張命令が出力され、小さなデータ型から大きなデータ型へ型変換されます。したがって、あらかじめデータサイズを統一することによって、ROM 効率および実行処理速度が向上します。

■使用例

変数 a と b を加算して、結果を変数 c に代入します。

（改善前のCプログラム）

```
unsigned char a;
        int  b,c;
void func(void)
{
    c=a+b;
}
```

（改善後のCプログラム）

```
int a,b,c;

void func(void)
{
    c=a+b;
}
```

（改善前のアセンブラ展開コード）

```
_func:
     MOV.B      @_a:32,R0L
     EXTU.W     R0
     MOV.W      @_b:32,E0
     ADD.W      E0,R0
     MOV.W      R0,@_c:32
     RTS
_a:
     .RES.B     1
_b:
     .RES.W     1
_c:
     .RES.W     1
```

（改善後のアセンブラ展開コード）

```
_func:
     MOV.W      @_a:32,R0
     MOV.W      @_b:32,E0
     ADD.W      E0,R0
     MOV.W      R0,@_c:32
     RTS

_a:
     .RES.W     1
_b:
     .RES.W     1
_c:
     .RES.W     1
```

オブジェクトサイズ表 [サイズ]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	24	18	24	18	18
改善後	22	16	22	16	16

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	24	24	18
改善後	22	22	16

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	19	15	38	30	30
改善後	18	14	36	28	28

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	13	13	12
改善後	13	13	12

6.1.4 const 修飾子の活用

サイズ効率	○	処理速度	○	スタックサイズ	-

■ポイント

値が不変な初期化データは、const 修飾を行うと、RAM 領域を節約できます。

■説明

初期値を持ったデータは、値の変更が可能なので、リンク時には ROM 上に配置し、プログラム実行開始時に RAM 上に複写します。このため、初期化データに関しては、ROM 上と RAM 上の二重に領域を確保します。値が不変なデータは、const 修飾をすることで RAM 上に変数領域を割り付けずに ROM 上のみに領域を確保することが可能です。

■使用例

初期化データを 5 バイト確保します。

（改善前のCプログラム）

```
unsigned char a[5]=
   {1, 2, 3, 4, 5};
```

（改善後のCプログラム）

```
const unsigned char a[5]=
         {1, 2, 3, 4, 5};
```

（改善前のアセンブラ展開コード）

```
  .SECTION  D,DATA,ALIGN=2
_a:
  .DATA.B   H'01,H'02,H'03,H'04,H'05
```

（改善後のアセンブラ展開コード）

```
  .SECTION  C,DATA,ALIGN=2
_a:
  .DATA.B   H'01,H'02,H'03,H'04,H'05
```

■備考および注意事項

改善前のプログラムでは、オブジェクトサイズ表に記されている容量のほかに、同量の RAM を確保する必要があります。

データが文字列の場合は、出力先をオプションで指定することができます。

【指定方法】

ダイアログメニュー：**コンパイラタブカテゴリ:[オブジェクト]**の**文字列データ格納:定数領域 ｜ 初期化データ領域**

コマンドオプション：*string=const* | *data*

デフォルトでは Const section へ出力されます。

6.1.5 変数のサイズをそろえる

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

ループ文などで比較を行う場合、変数のサイズをそろえると、拡張コードの分、コードが短縮されます。

■説明

コンパイラでは、データを比較する際に、データの大きさをそろえてから比較します。そのため、データの大きさをそろえておくと、データサイズをそろえ直すためのプログラムサイズの増加や実行時間の増大を抑制できます。

■使用例

ループして、関数 func1 を呼び出します。

（改善前のCプログラム）

```
extern char tb[5];
void sub(void)
{
   int i;
   for (i=0; i<2L; i++)
      func1(tb[i]);
}
```

（改善後のCプログラム）

```
extern char tb[5];
void sub(void)
{
   unsigned int i;
   for (i=0; i<2L; i++)
      func1(tb[i]);
}
```

（改善前のアセンブラ展開コード）

```
_sub:
      PUSH.L      ER6
      SUB.W       R6,R6
L6:
      EXTS.L      ER6
      MOV.B       @(_tb:32,ER6),R0L
      JSR         @_func1:24
      INC.W       #1,R6
      EXTS.L      ER6
      CMP.L       #2,ER6
      BLT         L6:8
      POP.L       ER6
      RTS
```

（改善後のアセンブラ展開コード）

```
_sub:
      PUSH.L      ER6
      SUB.W       R6,R6
L6:
      EXTU.L      ER6
      MOV.B       @(_tb:32,ER6),R0L
      JSR         @_func1:24
      INC.W       #1,R6

      CMP.W       #2,R6
      BLO         L6:8
      POP.L       ER6
      RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	34	30	40	34	50
改善後	34	28	36	26	28

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	32	30	26
改善後	32	30	24

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	59	50	124	102	378
改善後	59	49	116	86	90

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	43	43	41
改善後	43	43	37

6.1.6 ファイル内関数を static 指定する

サイズ効率	○	処理速度	○	スタックサイズ	○

■ポイント

ファイル内でのみ使用する関数は static 指定をします。

■説明

関数を static 指定すると、その関数が外部関数からの呼び出しのない場合は、実体を削除されます。また、インライン展開指定をした場合も、実体のコードが削除されるので、サイズ効率が良くなります。

■使用例

関数をインライン展開指定します。
関数 main から関数 func を呼び出します。

（改善前のCプログラム）

```
#pragma inline func
int a,b;
void func()
{
    a+=10;
}
void main()
{
    a=1;
    func();
    b=a;
}
```

（改善後のCプログラム）

```
#pragma inline func
int a,b;
static void func()
{
    a+=10;
}
void main()
{
    a=1;
    func();
    b=a;
}
```

（改善前のアセンブラ展開コード）

```
_func:
    MOV.L     #_a,ER0
    MOV.W     @ER0,R1
    ADD.W     #10,R1
    MOV.W     R1,@ER0
    RTS
_main:
    MOV.W     #11,R0
    MOV.W     R0,@_a:32
    MOV.W     R0,@_b:32
    RTS
```

（改善後のアセンブラ展開コード）

```
_main:
    MOV.W     #11,R0
    MOV.W     R0,@_a:32
    MOV.W     R0,@_b:32
    RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	36	28	34	28	28
改善後	18	14	18	14	14

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	26	26	20
改善後	14	14	10

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	15	12	30	24	24
改善後	15	12	30	24	24

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	10	10	8
改善後	10	10	8

6.2 演算

6.2.1 共通式の統合

サイズ効率	○	処理速度	○	スタックサイズ	○

■ポイント

複数の演算式において、共通な部分式を統合すると、ROM 効率および実行処理速度が向上します。

■説明

複数の式において、共通な部分式が 2 回以上現れるとき、最初の式における演算結果を 2 回目以降の式で利用します。このため、演算回数を削減できます。したがって、ROM 効率および実行処理速度が向上します。

なお、局所変数で共通部分式が 3 回以上現れるときは、コンパイラが最適化します。

■使用例

変数 x、y、z を加算し、結果を変数 a に格納します。また、変数 x、y、w を加算し、結果を変数 b に格納します。

（改善前のCプログラム）

```
unsigned char a,b,w,x,y,z;
void func(void)
{
    a=x+y+z;
    b=x+y+w;
}
```

（改善後のCプログラム）

```
unsigned char a,b,w,x,y,z;
void func(void)
{
    unsigned char tmp;
    tmp=x+y;
    a=tmp+z;
    b=tmp+w;
}
```

（改善前のアセンブラ展開コード）

```
_func:
     MOV.B      @_x:32,R1L
     MOV.B      @_y:32,R0H
     ADD.B      R1L,R0H
     MOV.B      R0H,R1H
     MOV.B      @_z:32,R0L
     ADD.B      R0L,R0H
     MOV.B      R0H,@_a:32
     MOV.B      @_w:32,R0L
     ADD.B      R0L,R1H
     MOV.B      R1H,@_b:32
     RTS
```

（改善後のアセンブラ展開コード）

```
_func:
     MOV.B      @_x:32,R0H
     MOV.B      @_y:32,R0L
     ADD.B      R0L,R0H
     MOV.B      @_z:32,R0L
     ADD.B      R0H,R0L
     MOV.B      R0L,@_a:32
     MOV.B      @_w:32,R0L
     ADD.B      R0L,R0H
     MOV.B      R0H,@_b:32
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	44	32	46	34	34
改善後	46	34	44	32	32

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	44	44	32
改善後	46	46	34

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	32	25	66	52	52
改善後	33	26	64	50	50

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	21	21	19
改善後	22	22	19

■備考および注意事項

本コンパイラは共通式統合の最適化を局所変数に対しては行いますが、外部変数に対しては行いません。

6.2.2 条件判定の改善

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

同じ条件式の評価は一括で行うと、ROM 効率が向上します。

■説明

同じ条件式を一括で評価します。このため、条件判定や条件式の評価回数を減らすことができます。したがって、ROM 効率および実行処理速度が向上します。

■使用例

変数 a、b の論理積を求めて、呼び出し側に結果を返します。

（改善前のCプログラム）

```
unsigned char a=1,b=1;
unsigned char func(void)
{
   if (!a)   return(0);
   if (a&&!b) return(0);
   return(1);
}
```

（改善後のCプログラム）

```
unsigned char a=1,b=1;
unsigned char func(void)
{
   if (a&&b) return(1);
   else      return(0);
}
```

（改善前のアセンブラ展開コード）

```
_func:
     MOV.B      @_a:32,R0H
     BNE        L6:8
     SUB.B      R0L,R0L
     RTS
L6:  MOV.B      R0H,R0H
     BEQ        L7:8
     MOV.B      @_b:32,R0L
     BEQ        L8:8
L7:  MOV.B      #1,R0L
L8:  RTS
```

（改善後のアセンブラ展開コード）

```
_func:
     MOV.B      @_a:32,R0L
     BEQ        L5:8
     MOV.B      @_b:32,R0L
     BEQ        L5:8
     MOV.B      #1,R0L
     RTS
L5:  SUB.B      R0L,R0L
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	26	22	30	24	24
改善後	26	22	26	20	20

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	26	26	22
改善後	26	26	22

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	18	15	42	36	36
改善後	18	15	36	30	30

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	15	15	14
改善後	15	15	14

■備考および注意事項

実行処理速度は a=1、b=1 の場合を仮定して、計測しています。

6.2.3 代入値による条件判定

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

代入値を判定文の条件式に用いる場合は、代入文をそのまま条件判定文とすると、ROM 効率が向上します。

■説明

条件式の判定と代入を同時に行うことにより、コードが大幅に削除されます。

■使用例

文字列 s を複写します。

（改善前のCプログラム）

```
char *s,*d;
void func(void)
{
    while(*s){
        *d++ = *s++;
    }
    *d++ = *s++;
}
```

（改善後のCプログラム）

```
char *s,*d;
void func(void)
{
    while(*d++ = *s++);
}
```

（改善前のアセンブラ展開コード）

```
_func:
     STM.L      (ER4-ER5),@-SP
     MOV.L      #_s,ER5
     MOV.L      #_d,ER4
     BRA        L7:8
L6:   MOV.B      @ER0+,R1L
     MOV.L      ER0,@ER5
     MOV.L      @ER4,ER0
     MOV.B      R1L,@ER0
     MOV.L      @ER4,ER0
     INC.L      #1,ER0
     MOV.L      ER0,@ER4
L7:   MOV.L      @ER5,ER0
     MOV.B      @ER0,R1L
     BNE        L6:8
     MOV.B      @ER0+,R1L
     MOV.L      ER0,@ER5
     MOV.L      @ER4,ER0
     MOV.B      R1L,@ER0
     MOV.L      @ER4,ER0
     INC.L      #1,ER0
     MOV.L      ER0,@ER4
     LDM.L      @SP+,(ER4-ER5)
     RTS
```

（改善後のアセンブラ展開コード）

```
_func:
     STM.L      (ER4-ER5),@-SP
     MOV.L      #_s,ER5
     MOV.L      #_d,ER4
L5:   MOV.L      @ER5,ER0
     MOV.B      @ER0+,R1L
     MOV.L      ER0,@ER5
     MOV.L      @ER4,ER0
     INC.L      #1,ER0
     MOV.L      ER0,@ER4
     MOV.B      R1L,@-ER0
     BNE        L5:8
     LDM.L      @SP+,(ER4-ER5)
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	80	62	74	54	54
改善後	52	32	44	36	34

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	70	70	56
改善後	52	52	32

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	59	48	232	84	84
改善後	45	26	218	74	74

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	37	43	31
改善後	26	27	20

6.2.4 数学的手法の活用

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

数学的手法を用いると、ROM 効率および実行処理速度が向上します。

■説明

演算式に共通項がある場合には、共通項を括りだします。このため、演算回数を削減することができます。したがって、ROM 効率および実行処理速度が向上します。

■使用例

3 次方程式を解きます。

（改善前のCプログラム）

```
unsigned char a,b,c,d,x,y;
void func(void)
{
   y=a*x*x*x+b*x*x+c*x+d;
}
```

（改善後のCプログラム）

```
unsigned char a,b,c,d,x,y;
void func(void)
{
   y=x*(x*(a*x+b)+c)+d;
}
```

（改善前のアセンブラ展開コード）

```
_func:
     PUSH.W     R6
     MOV.B      @_x:32,R6L
     MOV.B      R6L,R0L
     MULXU.B    R6L,R0
     MOV.B      R0L,R6H
     MULXU.B    R6L,R0
     MOV.B      @_a:32,R0H
     MULXU.B    R0H,R0
     MOV.B      @_b:32,R1L
     MULXU.B    R6H,R1
     ADD.B      R1L,R0L
     MOV.B      @_c:32,R1L
     MULXU.B    R6L,R1
     ADD.B      R1L,R0L
     MOV.B      @_d:32,R0H
     ADD.B      R0H,R0L
     MOV.B      R0L,@_y:32
     POP.W      R6
     RTS
```

（改善後のアセンブラ展開コード）

```
_func:
     MOV.B      @_x:32,R1L
     MOV.B      @_a:32,R0L
     MULXU.B    R1L,R0
     MOV.B      @_b:32,R0H
     ADD.B      R0H,R0L
     MULXU.B    R1L,R0
     MOV.B      @_c:32,R0H
     ADD.B      R0H,R0L
     MULXU.B    R1L,R0
     MOV.B      @_d:32,R0H
     ADD.B      R0H,R0L
     MOV.B      R0L,@_y:32
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	64	52	62	50	50
改善後	56	44	50	38	38

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	64	64	52
改善後	58	58	46

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	56	49	150	136	136
改善後	48	41	106	92	92

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	33	29	26
改善後	27	27	24

6.2.5 公式の活用

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

演算式に数学の公式があてはまるときには、公式を利用すると ROM 効率および実行処理速度が向上します。

■説明

アルゴリズムを意識したコーディングとして、数学の公式を利用して演算回数を削減します。したがって、ROM 効率および実行処理速度が向上します。

■使用例

1 から 100 までの総和を求めます。

（改善前のCプログラム）

```
unsigned int s;
unsigned int n=100;
void func(void)
{
   unsigned int i;
   for (s=0,i=1;i<=n;i++)
      s+=i;
}
```

（改善後のCプログラム）

```
unsigned int s;
unsigned int n=100;
void func(void)
{
   s=n*(n+1)>>1;
}
```

（改善前のアセンブラ展開コード）

```
_func:
     MOV.L      #_s:32,ER1
     SUB.W      R0,R0
     MOV.W      R0,@ER1
     MOV.W      #1:16,E0
     BRA        L7:8
L6:  MOV.W      @ER1,R0
     INC.W      #1,R0
     MOV.W      R0,@ER1
     INC.W      #1,E0
L7:  MOV.W      @_n:32,R0
     CMP.W      R0,E0
     BLS        L6:8
     RTS
```

（改善後のアセンブラ展開コード）

```
_func:
     MOV.W      @_n:32,R1
     MOV.W      R1,R0
     INC.W      #1,R0
     MULXU.W    R1,ER0
     SHLR.W     R0
     MOV.W      R0,@_s:32
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	38	32	38	34	38
改善後	24	20	24	20	30

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	36	36	30
改善後	24	24	20

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	1322	1118	2644	2438	2450
改善後	20	17	54	48	144

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	916	916	815
改善後	14	14	13

6.2.6 局所変数の活用

サイズ効率	○	処理速度	○	スタックサイズ	○

■ポイント

一時変数やループカウンタなど、局所変数として使用できるものは、局所変数として宣言すると ROM 効率、実行速度が向上します。

また、複数の演算式に共通している外部変数は、局所変数に代入してから演算を行うと、同様に向上します。

■説明

局所変数は、ほとんどの場合、レジスタに割り付くため、外部変数のようにメモリとレジスタ間のデータ転送のないオブジェクトを生成できます。

また、関数の割り込みなどで値の変わらない変数なら、局所変数に代入して演算を行えば、上記理由により、ROM 効率、実行処理速度が向上します。

■使用例

変数 a と変数 b、c、d をそれぞれ加算して、結果を b、c、d に格納します。

（改善前のCプログラム）

```
unsigned char a,b,c,d;
void func(void)
{
    b+=a;
    c+=a;
    d+=a;
}
```

（改善後のCプログラム）

```
unsigned char a,b,c,d;
void func(void)
{
    unsigned char wk;
    wk=a;
    b+=wk;
    c+=wk;
    d+=wk;
}
```

（改善前のアセンブラ展開コード）

```
_func:
     STM.L      (ER2-ER3),@-SP
     MOV.L      #_a:32,ER3
     MOV.B      @ER3,R0L
     MOV.L      #_b:32,ER1
     MOV.B      @ER1,R2L
     ADD.B      R0L,R2L
     MOV.B      R2L,@ER1
     MOV.B      @ER3,R0L
     MOV.L      #_c:32,ER1
     MOV.B      @ER1,R2L
     ADD.B      R0L,R2L
     MOV.B      R2L,@ER1
     MOV.B      @ER3,R3L
     MOV.L      #_d:32,ER0
     MOV.B      @ER0,R1L
     ADD.B      R3L,R1L
     MOV.B      R1L,@ER0
     LDM.L      @SP+,(ER2-ER3)
     RTS
```

（改善後のアセンブラ展開コード）

```
_func:
     MOV.B      @_a:32,R1H
     MOV.L      #_b:32,ER0
     MOV.B      @ER0,R1L
     ADD.B      R1H,R1L
     MOV.B      R1L,@ER0
     MOV.L      #_c:32,ER0
     MOV.B      @ER0,R1L
     ADD.B      R1H,R1L
     MOV.B      R1L,@ER0
     MOV.L      #_d:32,ER0
     MOV.B      @ER0,R1L
     ADD.B      R1H,R1L
     MOV.B      R1L,@ER0
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	50	36	60	50	50
改善後	50	36	48	40	40

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	32	32	24
改善後	32	32	24

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	36	28	156	64	64
改善後	36	28	56	48	48

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	20	20	18
改善後	20	20	18

■備考および注意事項

本技法は、コンパイラ Ver3.0 以前にて有効な技法です。

コンパイラ Ver4.0 より変数がレジスタに割り付くように改善されましたので局所変数を使用してもアセンブラ展開コードは、同じになります。

本項のアセンブラ展開コード、オブジェクトサイズ、実行処理速度は、コンパイラ Ver3.0 にてコンパイルした結果を記述します。(H8SX 以外)

外部変数を代入した局所変数がレジスタに割り付かない場合がありますので、オブジェクトリストを確認してください。

6.2.7 float 型定数には f をつける

サイズ効率	○	処理速度	○	スタックサイズ	○

■ポイント

float 型の範囲（7.0064923216240862e-46f～3.4028235677973364e+38f）にある定数の浮動小数点演算の場合、数字のあとに“ f ”をつけると、定数が float 型とみなされ、double 型への不要な型変換が行われません。

■説明

浮動小数点型の定数は、double 型となります。そのため、その定数をそのまま使用すると、double 型で演算するため、大変面倒な手続きが必要となります。しかし、値の範囲が、（7.0064923216240862e-46f～3.4028235677973364e+38f）の対数のとき、その定数の最後に“ f ”を付加すると、float 型とみなされるので、命令数が大幅に減少します。これにより、ROM 効率、RAM 効率、実行速度のすべてを向上することができます。

■使用例

変数 a に変数 b と定数の和を代入します。

（改善前のCプログラム）

```
float a,b;
void func(void)
{
   a=b+1.0;
}
```

（改善後のCプログラム）

```
float a,b;
void func(void)
{
   a=b+1.0f;
}
```

（改善前のアセンブラ展開コード）

```
_func:
     PUSH.L     ER2
     SUB.W      #16,R7
     MOV.L      @_b:32,ER1
     MOV.L      SP,ER0
     ADD.W      #8,R0
     JSR        @$FTOD$3:24
     MOV.L      ER0,ER1
     MOV.L      #L5,ER2
     MOV.L      SP,ER0
     JSR        @$ADDD$3:24
     JSR        @$DTOF$3:24
     MOV.L      ER0,@_a:32
     ADD.W      #16,R7
     POP.L      ER2
     RTS
L5:  .DATA.L   H'3FF00000,H'00000000
_a:  .RES.L     1
_b:  .RES.L     1
```

（改善後のアセンブラ展開コード）

```
_func:
     MOV.L      @_b:32,ER0
     MOV.L      #1065353216,ER1
     JSR        @$ADDF$3:24
     MOV.L      ER0,@_a:32
     RTS
_a:   .RES.L    1
_b:   .RES.L    1
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	76	66	70	60	62
改善後	28	24	28	24	26

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	78	72	66
改善後	30	28	24

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	351	337	798	770	1076
改善後	124	119	260	250	352

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	259	260	261
改善後	96	97	103

6.2.8 シフト演算の定数指定

サイズ効率	○	処理速度	○	スタックサイズ	○

■ポイント

シフト演算のときに、シフト数が変数の場合、実行時ルーチンを呼び出して処理します。シフト数が定数ならば、実行時ルーチンを呼び出さずに処理するため、処理速度が向上します。

■説明

定数が解決されていると、その場で処理することが可能です。

■使用例

変数 data を 8 ビットシフトします。

（改善前のCプログラム）

```
int data;
int sht=8;
void func(void)
{
    data=data<<sht;
}
```

（改善後のCプログラム）

```
#define SHT 8
int data;
void func(void)
{
    data=data<<SHT;
}
```

（改善前のアセンブラ展開コード）

```
_func:
        MOV.L       #_data,ER0
        MOV.W       @_sht:32,R1
        JSR         @$DSLI$3:24
        RTS
_sht:
        .DATA.W     H'0008
_data:
        .RES.W      1
```

（改善後のアセンブラ展開コード）

```
_func:
      MOV.B       @_data+1:32,R0H
      SUB.B       R0L,R0L
      MOV.W       R0,@_data:32
      RTS
_data:
      .RES.W      1
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	26	20	20	16	16
改善後	16	12	16	12	12

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	26	26	20
改善後	16	16	12

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	44	38	166	136	140
改善後	14	11	28	22	22

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	14	14	12
改善後	11	11	10

6.2.9 シフト演算の活用

サイズ効率	○	処理速度	○	スタックサイズ	○

■ポイント

乗算や加算を行いたいときでも、可能であれば、シフト演算を使います。

■説明

複合代入演算子（+=、-=、&=、|=、…など）や、シフト演算子は CPU の特性を引き出すために用意されているものなので、利用するとコードが少なくなるため、サイズ、スピード両面で効率向上します。特に、変数に対して定数値を乗算する場合は<<（左シフト演算子）で行います。

■使用例

変数 a に data の値 3 つ分をいれます。

（改善前のCプログラム）

```
int data,a;
void main()
{
    a=data+data+data;
}
```

（改善後のCプログラム）

```
int data,a;
void main()
{
    a=(data<<1)+data;
}
```

（改善前のアセンブラ展開コード）

```
_main:
    PUSH.L    ER6
    MOV.L     #_data:32,ER6
    MOV.W     @ER6,R0
    MOV.W     R0,R1
    ADD.W     R1,R0
    ADD.W     R1,R0
    MOV.W     R0,@_a:32
    POP.L     ER6
    RTS
```

（改善後のアセンブラ展開コード）

```
_main:
    MOV.W     @_data:32,R0
    SHLL.W    R0
    MOV.W     @_data:32,R1
    ADD.W     R1,R0
    MOV.W     R0,@_a:32
    RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	30	22	30	22	22
改善後	24	18	24	18	18

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	20	20	16
改善後	20	20	16

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	17	13	34	26	26
改善後	14	11	28	22	22

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	12	12	11
改善後	13	13	12

■備考および注意事項

本技法は、コンパイラ Ver3.0 以前にて使用できる技法です。

コンパイラ Ver4.0 より乗算や加算でもシフト演算を行うように改善しましたのでアセンブラ展開コードは、同じになります。

本項のアセンブラ展開コード、オブジェクトサイズ、実行処理速度は、コンパイラ Ver3.0 にてコンパイルした結果を記述します。(H8SX 以外)

6.2.10 連続した加算命令の統合

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

加算を行うときに、連続して記述すれば、統合されるためコードが少なくなります。

■説明

加算コードは連続していると統合するという最適化が起こります。この最適化を有効にするために、連続して記述します。

■使用例

変数 a の値を加算します。

（改善前のCプログラム）

```
int a,b;
void main()
{
    a+=10;
    b=10;
    a+=20;
}
```

（改善後のCプログラム）

```
int a,b;
void main()
{
    b=10;
    a+=10;
    a+=20;
}
```

（改善前のアセンブラ展開コード）

```
_main:
     MOV.W      @_a:32,E0
     ADD.W      #10,E0
     MOV.W      #10,R0
     MOV.W      R0,@_b:32
     ADD.W      #20,E0
     MOV.W      E0,@_a:32
     RTS
```

（改善後のアセンブラ展開コード）

```
_main:
     MOV.W      #10,R0
     MOV.W      R0,@_b:32
     MOV.W      @_a:32,E0
     ADD.W      R0,E0
     ADD.W      #20,E0
     MOV.W      E0,@_a:32
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	28	22	32	26	26
改善後	28	22	30	24	24

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	18	18	14
改善後	18	18	14

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	21	17	46	38	38
改善後	21	17	44	36	36

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	13	13	12
改善後	13	13	12

6.3 ループ処理

6.3.1 ループカウンタの選択

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

デクリメントカウンタを使用して、終了条件をゼロと比較すると、ROM 効率および実行処理速度が向上します。

■説明

H8S、H8/300 シリーズは、データ転送命令（MOV 命令）実行時にコンディションコードレジスタの N および Z フラグが変化します。このため、データ転送命令直後の比較命令を省略できますので、ROM 効率および実行処理速度が向上します。

■使用例

配列 a のすべての要素を配列 b に複写します。

（改善前のCプログラム）

```
unsigned char a[10],b[10];
        int  i;
void func(void)
{
   for(i=0; i<10; i++)
      b[i]=a[i];
}
```

（改善後のCプログラム）

```
unsigned char a[10],b[10];
        int  i;
void func(void)
{
   for(i=9; i>=0; i--)
      b[i]=a[i];
}
```

（改善前のアセンブラ展開コード）

```
_func:
     STM.L      (ER4-ER5),@-SP
     MOV.L      #_i,ER5
     MOV.W      #1,R0
     MOV.W      R0,@ER5
     BRA        L8:8
L6:  MOV.W      R0,R1
     EXTS.L     ER1
     MOV.L      ER1,ER4
     MOV.B      @(_a:32,ER4),R0L
     MOV.B      R0L,@(_b:32,ER4)
     INC.W      #1,R1
     MOV.W      R1,@ER5
L8:  MOV.W      @ER5,R0
     CMP.W      #10,R0
     BLT        L6:8
     LDM.L      @SP+,(ER4-ER5)
     RTS
```

（改善後のアセンブラ展開コード）

```
_func:
     STM.L      (ER4-ER5),@-SP
     MOV.L      #_i,ER5
     MOV.W      #9,R0
     MOV.W      R0,@ER5
     BRA        L8:8
L6:  MOV.W      R0,R1
     EXTS.L     ER1
     MOV.L      ER1,ER4
     MOV.B      @(_a:32,ER4),R0L
     MOV.B      R0L,@(_b:32,ER4)
     DEC.W      #1,R1
     MOV.W      R1,@ER5
L8:  MOV.W      @ER5,R0
     BGE        L6:8
     LDM.L      @SP+,(ER4-ER5)
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	46	30	54	38	38
改善後	48	34	52	36	36

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	30	30	24
改善後	36	36	32

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	163	121	624	366	386
改善後	164	132	582	324	324

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	125	107	98
改善後	106	118	108

6.3.2 繰り返し制御文の選択

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

必ず 1 回は実行されるループは、do-while 文を選択すると、ROM 効率および実行処理速度が向上します。

■説明

ループ文が必ず 1 回は実行されるときには do-while 文を使用すると、ループの判定回数を 1 回省略することができます。したがって、ROM 効率および実行処理速度が向上します。

■使用例

配列 p2 の内容を p1 へコピーします。

（改善前のCプログラム）

```
unsigned char a[10],len=10;
unsigned char p1[10],p2[10];
void func(void)
{
   char i;
   for (i=len; i>0; i--)
      p1[i-1]=p2[i-1];
}
```

（改善後のCプログラム）

```
unsigned char a[10],len=10;
unsigned char p1[10],p2[10];
void func(void)
{
   char i=len;
   do{
      p1[i-1]=p2[i-1];
   } while(--i);
}
```

（改善前のアセンブラ展開コード）

```
_func:
       PUSH.L ER5
       MOV.B  @_len:32,R1L
       BRA    L9:8
L8:    EXTS.W R1
       EXTS.L ER1
       MOV.L  ER1,ER5
       MOV.B  @(_p2-1:32,ER5),R0L
       MOV.B  R0L,@(_p1-1:32,ER5)
       DEC.B  R1L
L9:    MOV.B  R1L,R1L
       BGT    L8:8
       POP.L  ER5
       RTS
```

（改善後のアセンブラ展開コード）

```
_func:
       PUSH.L ER5
       MOV.B  @_len:32,R1L
L9:
       EXTS.W R1
       EXTS.L ER1
       MOV.L  ER1,ER5
       MOV.B  @(_p2-1:32,ER5),R0L
       MOV.B  R0L,@(_p1-1:32,ER5)
       DEC.B  R1L
       BNE    L9:8
       POP.L  ER5
       RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	47	29	47	39	33
改善後	43	25	43	35	29

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	35	35	29
改善後	31	31	25

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	210	142	388	324	296
改善後	195	127	358	294	266

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	134	134	133
改善後	117	117	116

6.3.3 ループ内不変式のループ外移動

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

ループ内での不変式をループ外に定義すると、実行処理速度が向上します。

■説明

ループ内不変式をループ外に定義すると、不変式の評価をループ開始時のみ行います。このため、ループ本体の実行命令数が減ります。したがって、実行処理速度が向上します。

■使用例

変数 b と c の加算値で、配列 a を初期化します。

（改善前のCプログラム）

```
unsigned char a[10],b,c;
        int  i;
void func(void)
{
    for (i=9; i>=0; i--)
        a[i]=b+c;
}
```

（改善後のCプログラム）

```
unsigned char a[10],b,c;
        int  i;
void func(void)
{
    unsigned char tmp;
    tmp=b+c;
    for (i=9; i>=0; i--)
        a[i]=tmp;
}
```

（改善前のアセンブラ展開コード）

```
_func:
     PUSH.L     ER5
     MOV.L      #_i,ER5
     MOV.W      #9,R0
     MOV.W      R0,@ER5
     BRA        L9:8
L7:  MOV.W      R0,R1
     MOV.B      @_b:32,R0L
     MOV.B      @_c:32,R0H
     ADD.B      R0H,R0L
     EXTS.L     ER1
     MOV.B      R0L,@(_a:32,ER1)
     DEC.W      #1,R1
     MOV.W      R1,@ER5
L9:  MOV.W      @ER5,R0
     BGT        L7:8
     POP.L      ER5
     RTS
```

（改善後のアセンブラ展開コード）

```
_func:
     PUSH.W     R4
     MOV.L      #_i,ER1
     MOV.B      @_b:32,R4L
     MOV.B      @_c:32,R0L
     ADD.B      R0L,R4L
     MOV.W      #9,R0
     BRA        L10:8
L8:  MOV.W      @ER1,R0
     EXTS.L     ER0
     MOV.B      R4L,@(_a:32,ER0)
     DEC.W      #1,R0
L10: MOV.W      R0,@ER1
     BGT        L8:8
     POP.W      R4
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	50	40	58	42	42
改善後	50	40	50	38	38

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	46	46	38
改善後	46	46	38

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	119	109	516	404	400
改善後	119	109	322	254	254

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	101	95	83
改善後	101	95	83

6.3.4 ループ条件のマージ

サイズ効率	○	処理速度	○	スタックサイズ	○

■ポイント

ループの条件が同一または、類似している場合、マージすると ROM 効率、実行処理速度とも向上させることができます。

■説明

ループ判定文のオブジェクトが削減され、実行速度が大幅に向上します。

■使用例

配列 a を 0 で、配列 b を 1 で初期化します。

（改善前のCプログラム）

```
int a[10],b[10];
void f(void)
{
   int i,j;
   for (i=0; i<10; i++)
      a[i]=0;
   for (j=0; j<10; j++)
      b[j]=1;
}
```

（改善後のCプログラム）

```
int a[10],b[10];
void f(void)
{
   int i;
   for (i=0; i<10; i++){
      a[i]=0;
      b[i]=1;
   }
}
```

（改善前のアセンブラ展開コード）

```
_f:
      PUSH.L     ER6
      SUB.W      R6,R6
      SUB.W      R1,R1
L9:   EXTS.L     ER6
      MOV.L      ER6,ER0
      SHLL.L     ER0
      MOV.W      R1,@(_a:32,ER0)
      INC.W      #1,R6
      CMP.W      #10,R6
      BLT        L9:8
      SUB.W      R6,R6
      MOV.W      #1,R1
L10:  EXTS.L     ER6
      MOV.L      ER6,ER0
      SHLL.L     ER0
      MOV.W      R1,@(_b:32,ER0)
      INC.W      #1,R6
      CMP.W      #10,R6
      BLT        L10:8
      POP.L      ER6
      RTS
```

（改善後のアセンブラ展開コード）

```
_f:
     PUSH.L     ER6
     SUB.W      R6,R6
     SUB.W      R1,R1
L7:  EXTS.L     ER6
     MOV.L      ER6,ER0
     SHLL.L     ER0
     MOV.W      R1,@(_a:32,ER0)
     INC.W      #1,R6
     CMP.W      #10,R6
     BLT        L7:8
     EXTS.L     ER6
     SHLL.L     ER6
     MOV.W      #1,R0
     MOV.W      R0,@(_b:32,ER6)
     POP.L      ER6
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	58	48	62	44	50
改善後	46	32	46	32	34

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	38	38	34
改善後	28	28	24

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	245	197	558	418	468
改善後	188	134	432	350	342

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	175	177	168
改善後	125	117	117

6.4 ポインタ

6.4.1 ポインタ変数の活用

サイズ効率	○	処理速度	○	スタックサイズ	○

■ポイント

同一の変数（外部変数）を何度も参照する場合や、配列要素のアクセスにポインタ変数を用いると、ROM 効率および実行処理速度が向上します。

■説明

ポインタ変数を用いることにより、効率の良いアドレッシングモード（@Rn、@Rn+、@-Rn）でコードを生成することができます。

■使用例

配列 data2 の要素を配列 data1 に複写します。

（改善前のCプログラム）

```
void func(int data1[],int data2[])
{
   int i;

   for (i=0; i<10; i++)
      data1[i]=data2[i];
}
```

（改善後のCプログラム）

```
void func(int *data1,int *data2)
{
   int i;

   for (i=0; i<10; i++){
      *data1=*data2;
      data1++; data2++;
   }
}
```

（改善前のアセンブラ展開コード）

```
_func:
        PUSH.L      ER3
        STM.L       (ER4-ER6),@-SP
        MOV.L       ER0,ER4
        MOV.L       ER1,ER3
        SUB.W       R6,R6
L6:
        EXTS.L      ER6
        MOV.L       ER6,ER5
        SHLL.L      ER5
        MOV.L       ER4,ER0
        ADD.L       ER5,ER0
        MOV.L       ER3,ER1
        ADD.L       ER5,ER1
        MOV.W       @ER1,R1
        MOV.W       R1,@ER0
        INC.W       #1,R6
        CMP.W       #10:16,R6
        BLT         L6:8
        LDM.L       @SP+,(ER4-ER6)
        POP.L       ER3
        RTS
```

（改善後のアセンブラ展開コード）

```
__func:
        PUSH.L      ER5
        MOV.L       ER0,ER5
        MOV.W       #10:16,E0
L7:
        MOV.W       @ER1,R0
        MOV.W       R0,@ER5
        INC.L       #2,ER5
        INC.L       #2,ER1
        DEC.W       #1,E0
        BNE         L7:8
        POP.L       ER5
        RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	46	38	40	38	40
改善後	24	22	28	26	30

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	42	42	34
改善後	18	18	18

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	171	152	482	336	428
改善後	107	102	216	208	238

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	156	164	155
改善後	79	81	80

6.5 データ構造

6.5.1 データの整合性

サイズ効率	○	処理速度	○	スタックサイズ	−

■ポイント

データは宣言順に割り付けられます。したがって、宣言する順序を工夫して未使用領域を作成しないようにすると、ROM、RAM 領域を有効に活用できます。

■説明

偶数領域を保持するために、2 バイト以上の変数が奇数番地からの配置になる場合、強制的に 1 バイトの未使用領域が設けられます。大きさが同じ型の変数をまとめて宣言すると、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。

このデータの場合は、外部変数のみでなく、局所変数、構造体や共用体のメンバ、または、関数の引数においても必要です。

■使用例

全部で 8 バイトのデータを配置します。

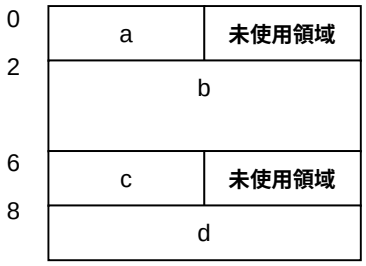
（改善前のCプログラム）

```
char  a;
long  b;
char  c;
short d;
```

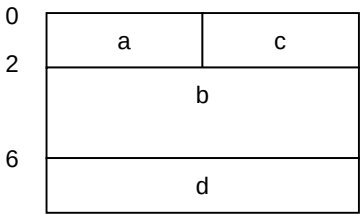
（改善後のCプログラム）

```
char  a;
char  c;
long  b;
short d;
```

（改善前データ配置）



（改善後データ配置）



オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	10	10	10	10	10
改善後	8	8	8	8	8

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	8	8	8
改善後	8	8	8

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	–	–	–	–	–
改善後	–	–	–	–	–

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	–	–	–
改善後	–	–	–

■使用例

上記の H8SX 以外は Ver3.0 の結果です。Ver4.0 以降は align オプションがデフォルトになったので、自動で空き領域が発生しないように調整します。

そのため、改善による結果の差異は生じません。

6.5.2 データの初期化方法

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

初期化が必要な変数は、宣言時に初期値を設定すると、プログラムサイズが削減されます。

■説明

宣言時に初期値を設定されたデータは、初期化データ領域（D セクション）に割り付けられると、実行時に RAM 上にコピーされます。初期値の代入は、プログラム実行開始時に一度にまとめて行います。

一方、宣言時に初期値を設定しないと、未初期化データ領域（B セクション）に割り付けられます。これにより、初期化データ領域に割り付けられた場合に比べ、メモリが半分で済みます。

しかし、初期値をプログラム上で代入文により設定するためのプログラム領域（P セクション）が増えてしまいます。

初期値の必要な変数が複数あるときには、宣言時に初期値を与えた方が効率が良くなります。

■使用例

変数 a を初期化します。

（改善前のCプログラム）

```
int a;
void main(void)
{
   a=1;
}
```

（改善後のCプログラム）

```
int a=1;
void main(void)
{
}
```

（改善前のアセンブラ展開コード）

```
_main:
        MOV.W      #1:16,R0
        MOV.W      R0,@_a:32
        RTS
        .SECTION    B,DATA,ALIGN=2
_a:
        .RES.W     1
```

（改善後のアセンブラ展開コード）

```
_main:
        RTS
        .SECTION    D,DATA,ALIGN=2
_a:
        .DATA.W     H'0001
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	12	10	12	10	10
改善後	4	4	4	4	4

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	8	8	6
改善後	4	4	4

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	11	9	22	18	18
改善後	5	4	10	8	8

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	8	8	7
改善後	6	6	6

6.5.3 配列要素の初期化の統合

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

いくつかの配列要素への初期化が存在する場合には、構造体などにまとめて 1 回で初期化を行うと ROM 効率が向上します。

■説明

データをまとめて初期化すると、転送命令が 1 回に短縮できます。

■使用例

配列 a、b、c をそれぞれの値に初期化します。

（改善前のCプログラム）

```
void f(void)
{
   unsigned char a[]={0,1,2,3};
   unsigned char b[]="abcdefg";
   unsigned char c[]="ABCDEFG";
}
```

（改善後のCプログラム）

```
void f(void)
{
   struct x{
      unsigned char a[4];
      unsigned char b[8];
      unsigned char c[7];
   } A
   ={0,1,2,3,"abcdefg","ABCDEFG"};
}
```

（改善前のアセンブラ展開コード）

```
_f:
     PUSH.L     ER2
     SUB.W      #20,R7
     MOV.L      #L4,ER0
     MOV.L      SP,ER1
     ADD.W      #16,R1
     SUB.L      ER2,ER2
     MOV.B      #4,R2L
     JSR        @$MVN$3:24
     MOV.L      #L6,ER0
     MOV.L      SP,ER1
     ADD.W      #8,R1
     SUB.L      ER2,ER2
     MOV.B      #8,R2L
     JSR        @$MVN$3:24
     MOV.L      #L8,ER0
     MOV.L      SP,ER1
     SUB.L      ER2,ER2
     MOV.B      #8,R2L
     JSR        @$MVN$3:24
     ADD.W      #20,R7
     POP.L      ER2
     RTS
L4:    .DATA.B    H'00,H'01,H'02,H'03
L6:    .SDATAZ    "abcdefg"
L8:    .SDATAZ    "ABCDEFG"
```

（改善後のアセンブラ展開コード）

```
_f:
         PUSH.L     ER2
         SUB.W      #20,R7
         MOV.L      #L4,ER0
         MOV.L      SP,ER1
         SUB.L      ER2,ER2
         MOV.B      #19,R2L
         JSR        @$MVN$3:24
         ADD.W      #20,R7
         POP.L      ER2
         RTS
L4:    .DATA.B    H'00,H'01,H'02,H'03
       .SDATAZ     "abcdefg"
       .SDATA      "ABCDEFG"
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	120	106	122	106	110
改善後	81	69	81	75	79

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	104	104	96
改善後	79	77	69

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	294	256	690	572	632
改善後	162	145	488	324	376

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	51	48	41
改善後	90	89	79

■備考

CPU が H8SX の場合、1 回の転送命令に実行時ルーチンを使わず、命令で実現できるため、改善前の処理速度の方が速くなります。

6.5.4 引数の構造体アドレス渡し

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

レジスタに割り付かない引数は、構造体のアドレスで渡すとプログラムサイズが小さくなります。

■説明

引数はなるべくレジスタに割り付けられるよう、引数の数とサイズを工夫してください。引数のレジスタ渡しについてはユーザーズマニュアルを参照してください。

引数のサイズが大きい場合や個数が多い時には構造体にまとめて目的の関数に渡すとプログラムサイズが減少します。引数を構造体のメンバとして宣言し、目的の関数に先頭アドレスを引数として渡すと、渡された関数の側で、そのアドレスを基にして各メンバをアクセスすることができます。

■使用例

long 型データ a、b、c、d を関数 func へ渡します。

（改善前のCプログラム）

```
void sub(long,long,long,long);
long a,b,c,d;

void func(void)
{
   sub(a,b,c,d);
}
```

（改善後のCプログラム）

```
void sub(struct ctag *);
struct ctag{
   long a;
   long b;
   long c;
   long d;
}x;

void func(void)
{
   sub(&x);
}
```

（改善前のアセンブラ展開コード）

```
_func:
        MOV.L      @_d:32,ER0
        PUSH.L     ER0
        MOV.L      @_c:32,ER0
        PUSH.L     ER0
        MOV.L      @_b:32,ER1
        MOV.L      @_a:32,ER0
        JSR        @_sub:24
        ADDS.L     #4,SP
        ADDS.L     #4,SP
        RTS
_a:     .RES.L     1
_b:     .RES.L     1
_c:     .RES.L     1
_d:     .RES.L     1
```

（改善後のアセンブラ展開コード）

```
_func:
        MOV.L      #_x:32,ER0
        JMP        @_sub:24
_x:
        .RES.W     8
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	58	50	52	44	62
改善後	14	12	10	10	10

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	50	48	40
改善後	16	14	12

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	52	45	102	88	126
改善後	18	14	22	18	18

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	30	28	28
改善後	14	14	14

6.5.5 構造体のレジスタ割り付け

サイズ効率	○	処理速度	○	スタックサイズ	○

■ポイント

ローカル変数に構造体を使うときは、そのままレジスタに割り付けられるようにメンバを宣言しましょう。

■説明

構造体でも、レジスタに割り付けられるのでメンバの割り付けを工夫すればコードが短縮されるため、サイズ効率、処理速度ともに向上します。

■使用例

構造体データを関数 func へ渡します。

（改善前のCプログラム）

```
struct ST {
   char  a;
   short b;
   char  c;
}pst;
void main()
{
   struct ST s;
   s.a=pst.a+10;
   s.b=s.a+s.c;
   func(s);
}
```

（改善後のCプログラム）

```
struct ST {
   short b;
   char  a;
   char  c;
}pst;
void main()
{
   struct ST s;
   s.a=pst.a+10;
   s.b=s.a+s.c;
   func(s);
}
```

（改善前のアセンブラ展開コード）

```
_main:
     STM.L      (ER2-ER3),@-SP
     SUBS.L     #4,SP
     SUBS.L     #2,SP
     MOV.L      SP,ER3
     MOV.B      #10,R0L
     MOV.B      R0L,@_pst:32
     MOV.B      R0L,@ER3
     EXTS.W     R0
     MOV.B      @(4:16,ER3),R1L
     EXTS.W     R1
     ADD.W      R1,R0
     MOV.W      R0,@(2:16,ER3)
     MOV.L      ER3,ER0
     SUBS.L     #4,SP
     SUBS.L     #2,SP
     MOV.L      SP,ER1
     SUB.L      ER2,ER2
     MOV.B      #6,R2L
     JSR        @$MVN$3:24
     JSR        @_func:24
     ADDS.L     #4,SP
     ADDS.L     #4,SP
     ADDS.L     #4,SP
     LDM.L      @SP+,(ER2-ER3)
     RTS
```

（改善後のアセンブラ展開コード）

```
_main:
     PUSH.L     ER6
     MOV.B      #10,R0L
     MOV.B      R0L,@_pst+2:32
     MOV.B      R0L,R6H
     EXTS.W     R0
     MOV.B      R6L,R1L
     EXTS.W     R1
     ADD.W      R1,R0
     MOV.W      R0,E6
     PUSH.L     ER6
     JSR        @_func:24
     ADDS.L     #4,SP
     POP.L      ER6
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	66	60	64	64	80
改善後	44	46	42	40	72

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	62	60	64
改善後	42	40	38

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	126	110	416	252	286
改善後	42	40	84	76	260

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	40	40	36
改善後	31	29	32

6.6 関数

6.6.1 関数定義位置の改善

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

モジュール内で頻繁に呼び出される関数は、同一ファイル内に定義すると ROM 効率および実行処理速度が向上する可能性があります。

■説明

H8S、H8/300 シリーズは、分岐先が−128～127 バイトの範囲内であるときは、PC 相対アドレッシングモード（BSR）を使用します。このモードは、外部参照関数が宣言したときの絶対アドレッシングモード（JSR）よりも ROM 効率および実行処理速度が向上します。

■使用例

関数 func および func1 から関数 func2 を呼び出します。

（改善前のCプログラム）

```
extern int func2(void);
int ret;
void func(void)
{
  int i;
  i=func2();
  ret = i;
}
void func1(void)
{
  int i;
  i=func2();
  ret = i;
}
```

（改善後のCプログラム）

```
int ret;
int func2(void)
{
  return 0;
}
void func(void)
{
  int i;
  i=func2();
  ret = i;
}
void func1(void)
{
  int i;
  i=func2();
}
```

（改善前のアセンブラ展開コード）

```
_func:
        JSR         @_func2:24
        MOV.W       R0,@_ret:32
        RTS
_func1:
        JSR         @_func2:24
        MOV.W       R0,@_ret:32
        RTS
```

（改善後のアセンブラ展開コード）

```
_func2:
        SUB.W       R0,R0
        RTS
_func:
        BSR         _func2:8
        MOV.W       R0,@_ret:32
        RTS
_func1:
        BSR         _func2:8
        MOV.W       R0,@_ret:32
        RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	28	24	28	24	24
改善後	24	20	24	20	20

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	32	28	24
改善後	24	24	20

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	20	16	40	32	32
改善後	19	15	38	30	30

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	16	16	15
改善後	16	16	16

6.6.2 マクロ呼び出し

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

頻繁に呼び出しがある関数をマクロ定義すれば、サイズ効率、処理速度が向上します。

■説明

同一の処理をマクロにすると、呼び出しの箇所にインライン展開され、コードが生成されないので、効率が向上します。

■使用例

関数 abs を呼び出します。

（改善前のCプログラム）

```
extern int a,b,c;
int abs(x)
int x;
{   return x>=0?x:-x;  }
void f(void)
{
   a=abs(b);
   b=abs(c);
}
```

（改善後のCプログラム）

```
#define abs(x) ((x)>=0?(x):-(x))
extern int a,b,c;
void f(void)
{
   a=abs(b);
   b=abs(c);
}
```

（改善前のアセンブラ展開コード）

```
_abs:
        PUSH.W     R6
        MOV.W      R0,R6
        BLT        L9:8
        MOV.W      R6,R1
        BRA        L10:8
L9:     MOV.W      R6,R1
        NEG.W      R1
L10:    MOV.W      R1,R0
        POP.W      R6
        RTS
_f:     MOV.W      @_b:32,R0
        BSR        _abs:8
        MOV.W      R0,@_a:32
        MOV.W      @_c:32,R0
        BSR        _abs:8
        MOV.W      R0,@_b:32
        RTS
```

（改善後のアセンブラ展開コード）

```
_f:
     PUSH.W     R6
     MOV.W      @_b:32,R6
     BLT        L7:8
     MOV.W      R6,R0
     BRA        L8:8
L7:  MOV.W      R6,R0
     NEG.W      R0
L8:  MOV.W      R0,@_a:32
     MOV.W      @_c:32,R6
     BLT        L9:8
     MOV.W      R6,R0
     BRA        L10:8
L9:  MOV.W      R6,R0
     NEG.W      R0
L10:  MOV.W     R0,@_b:32
     POP.W      R6
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	38	30	46	38	42
改善後	32	26	50	42	46

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	38	38	30
改善後	34	34	26

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	45	36	106	88	112
改善後	24	20	74	64	64

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	36	36	34
改善後	20	20	17

6.6.3 原型宣言

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

char 型や unsigned char 型の引数を持つ関数は、原型宣言をしてから呼び出すと不要な型変換コードが出力されません。

■説明

char 型や unsigned char 型の引数を持つ関数は原型宣言をしないで呼び出すと、int 型に変換されるので無駄な符号拡張命令やゼロ拡張命令が出力されます。

また、引数が正しく渡らないこともあります。

■使用例

char 型と unsigned char 型の引数を持つ関数 sub1 を呼び出します。

（改善前のCプログラム）

```
char a;
unsigned char b;
void func(void)
{
   sub1(a,b);
}
```

（改善後のCプログラム）

```
void sub1(char, unsigned char);
char a;
unsigned char b;
void func(void)
{
   sub1(a,b);
}
```

（改善前のアセンブラ展開コード）

```
_func:
        MOV.B      @_b:32,R0L
        EXTU.W     R0
        MOV.B      @_a:32,R1L
        EXTS.W     R1
        MOV.W      R0,E0
        MOV.W      R1,R0
        JMP        @_sub1:24
        RTS
```

（改善後のアセンブラ展開コード）

```
_func:
        MOV.B      @_b:32,R0H
        MOV.B      @_a:32,R0L
        JMP        @_sub1:24
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	22	18	24	20	18
改善後	18	14	16	20	12

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	24	22	18
改善後	20	18	14

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	19	16	40	34	32
改善後	23	18	32	26	26

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	15	15	14
改善後	19	17	17

6.6.4 テイルリカージョン最適化

サイズ効率	○	処理速度	○	スタックサイズ	○

■ポイント

関数コールで呼び出す側の関数の末尾に関数の呼び出しを移動できないか検討してください。ROM 効率および実行処理速度が向上します。

■説明

テイルリカージョン最適化は、下記の条件をすべて満たすとき作用します。

- 呼び出す関数が引数をスタックにつまないとき、かつリターン値アドレスをスタックにつまないとき
- 関数呼び出しの直後が RTS 命令のとき

■使用例

関数 sub を呼び出します。また、外部変数の値を更新します。

（改善前のCプログラム）

```
void g(void);
int a;
void main(void)
{
   if (a==0)    a++;
   else{
      g();
      a+=2;
   }
}
```

（改善後のCプログラム）

```
void g(void);
int a;
void main(void)
{
   if (a==0)    a++;
   else{
      a+=2;
      g();
   }
}
```

（改善前のアセンブラ展開コード）

```
_main:
     PUSH.L     ER6
     MOV.L      #_a,ER6
     MOV.W      @ER6,R0
     BNE        L6:8
     INC.W      #1,R0
     BRA        L8:8
L6:  JSR        @_g:24
     MOV.W      @ER6,R0
     INC.W      #2,R0
L8:  MOV.W      R0,@ER6
     POP.L      ER6
     RTS
```

（改善後のアセンブラ展開コード）

```
_main:
     MOV.L      #_a64,ER1
     MOV.W      #1,R0
     INC.W      #2,R0
     MOV.W      R0,@ER1
     JMP        @_g:24
     RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	38	32	38	32	32
改善後	30	32	30	28	28

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	36	34	30
改善後	30	28	34

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	37	29	74	58	58
改善後	20	27	40	36	36

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	22	25	20
改善後	15	15	23

■備考および注意事項

上記使用例のプログラムにおいて、関数 g 内で変数 a を参照しないことを仮定しています。

6.6.5 引数の渡し方法の工夫

サイズ効率	○	処理速度	○	スタックサイズ	○

■ポイント

引数の並び順をギャップのないようにすると、コードが少なくなります。

■説明

レジスタ渡しの対象の型である引数は、宣言順にレジスタ ER0、ER1（CPU が 300 の場合は R0、R1）に割り付けます。そのため、なるべくギャップがなくなるように宣言順を移動すると、コードが短縮される場合があります。

■使用例

関数 func を呼び出します。

（改善前のCプログラム）

```
long rtn;
void func(char,short,char);
void main()
{
    short a;
    char  b,c;

    func(b,a,c);
}
void func(char x,short y,char z)
{
    rtn=x*y+z;
}
```

（改善後のCプログラム）

```
long rtn;
void func(char,char,short);
void main()
{
    short a;
    char  b,c;

    func(b,c,a);
}
void func(char x,char y,short z)
{
    rtn=x*y+z;
}
```

（改善前のアセンブラ展開コード）

```
_main:
    SUBS.L    #4,SP
    SUBS.L    #2,SP
    MOV.B     @(5:16,SP),R0H
    MOV.W     @(2:16,SP),E0
    MOV.B     @SP,R0L
    BSR       _func:8
    ADDS.L    #2,SP
    ADDS.L    #4,SP
    RTS
_func:
    PUSH.L      ER6
    MOV.B       R0H,R6H
    EXTS.W      R0
    MOV.W       R0,R1
    MULXU.W     E0,ER1
    MOV.B       R6H,R6L
    EXTS.W      R6
    ADD.W       R6,R1
    EXTS.L      ER1
    MOV.L       ER1,@_rtn:32
    POP.L       ER6
    RTS
```

（改善後のアセンブラ展開コード）

```
_main:
    SUBS.L  #4,SP
    MOV.W   @(2:16,SP),E0
    MOV.B   @(1:16,SP),R0H
    MOV.B   @SP,R0L
    BSR     _func:8
    ADDS.L  #4,SP
    RTS
_func:
    MOV.B   R0L,R1L
    MULXS.B R0H,R1
    ADD.W   E0,R1
    EXTS.L  ER1
    MOV.L   ER1,@_rtn:32
    RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	26	24	56	54	60
改善後	22	20	38	36	48

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	26	26	24
改善後	22	22	20

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	29	25	120	112	228
改善後	26	22	82	74	174

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	25	21	21
改善後	21	19	19

■備考および注意事項

引数の割り付け方については H8S、H8/300 **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「**9.3.3 **引数割り付けの具体例」**に記載されておりますので、参照してください。

また、引数渡し用レジスタ数をオプションで変更した場合は引数割り付け用のレジスタが変わりますので、留意してください。

6.7 分岐

6.7.1 switch 文のテーブル方式への書き換え

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

switch 文の各 case 文の処理がほぼ同じ場合は、テーブルを使用して記述するとオブジェクトサイズが縮小されます。

■説明

switch 文をテーブルを使用して書き換えると、データサイズは増えますが、プログラムサイズが大幅に削減されます。しかし、case 値の範囲が広い場合にテーブルに書き替えるとテーブルサイズが増大するため、全体のサイズが増えてしまいますので、注意が必要です。

■使用例

関数 a の値によって関数へ分岐します。

（改善前のCプログラム）

```
extern void f1(void);
extern void f2(void);
extern void f3(void);
extern void f4(void);
extern void f5(void);
int a=4;
void sub(void)
{
    switch(a){
    case 0:f1();break;
    case 1:f2();break;
    case 2:f3();break;
    case 3:f4();break;
    case 4:f5();break;
    }
}
```

（改善後のCプログラム）

```
extern void f1(void);
extern void f2(void);
extern void f3(void);
extern void f4(void);
extern void f5(void);
int a=4;
void sub(void)
{
    static int (*key[5])()=
        {f1,f2,f3,f4,f5};

    (*key[a])();
}
```

（改善前のアセンブラ展開コード）

```
_sub:   MOV.W     @_a:32,R0
        MOV.B     R0H,R0H
        BNE       L15:8
        CMP.B     #0:8,R0L
        BEQ       L10:8
        CMP.B     #1:8,R0L
        BEQ       L11:8
        CMP.B     #2:8,R0L
        BEQ       L12:8
        CMP.B     #3:8,R0L
        BEQ       L13:8
        CMP.B     #4:8,R0L
        BEQ       L14:8
        RTS
L10:    JMP       @_f1:24
L11:   JMP       @_f2:24
L12:   JMP       @_f3:24
L13:   JMP       @_f4:24
L14:   JSR       @_f5:24
L15:    RTS
```

（改善後のアセンブラ展開コード）

```
_sub: MOV.W     @_a:32,R0
      EXTS.L    ER0
      SHLL.L    #2,ER0
      MOV.L     @(L9:32,ER0),ER0
      JSR       @ER0
      RTS
L9:   .DATA.L   _f1,_f2,_f3,_f4,_f5
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	78	64	66	62	76
改善後	56	36	56	34	34

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	94	78	58
改善後	50	50	36

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	31	28	74	54	68
改善後	27	18	42	26	26

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	24	32	24
改善後	19	20	18

6.7.2 case 文の飛び先が同じ場合の記述

サイズ効率	○	処理速度	○	スタックサイズ	△

■ポイント

case 文で式が同じ場合は、まとめると分岐命令が少なくなるため、オブジェクトサイズも少なくなります。

■説明

switch 文の展開方法で if-then 方式の場合、分岐命令が少ない方とコードが短縮されるので効率も向上します。

■使用例

c の値に応じて ll に値を代入します。

（改善前のCプログラム）

```
long ll;
void func(void)
{
   char c;
   switch(c){
   case 0: ll=0; break;
   case 1: ll=0; break;
   case 2: ll=1; break;
   case 3: ll=1; break;
   case 4: ll=2; break;
   }
}
```

（改善後のCプログラム）

```
long ll;
void func(void)
{
   char c;
   switch(c){
   case 0:
   case 1: ll=0; break;
   case 2:
   case 3: ll=1; break;
   case 4: ll=2; break;
   }
}
```

（改善前のアセンブラ展開コード）

```
_func:
        SUBS.L     #2,SP
        MOV.L      #_ll:32,ER1
        MOV.B      @(1:16,SP),R0L
        BEQ        L6:8
        CMP.B      #1:8,R0L
        BEQ        L7:8
        CMP.B      #2:8,R0L
        BEQ        L8:8
        CMP.B      #3:8,R0L
        BEQ        L9:8
        CMP.B      #4:8,R0L
        BEQ        L10:8
        BRA        L11:8
L6:
L7:    SUB.L      ER0,ER0
        BRA        L15:8
L8:     SUB.L      ER0,ER0
        MOV.B      #1:8,R0L
        BRA        L15:8
L9:     SUB.L      ER0,ER0
        MOV.B      #1:8,R0L
        BRA        L15:8
L10:   SUB.L      ER0,ER0
        MOV.B      #2:8,R0L
L15:   MOV.L      ER0,@ER1
L11:    ADDS.L     #2,SP
        RTS
_ll:   .RES.L     1
```

（改善後のアセンブラ展開コード）

```
_func:
        SUBS.L     #2,SP
        MOV.L      #_ll:32,ER1
        MOV.B      @(1:16,SP),R0L
        BEQ        L6:8
        CMP.B      #1:8,R0L
        BEQ        L7:8
        CMP.B      #2:8,R0L
        BEQ        L8:8
        CMP.B      #3:8,R0L
        BEQ        L9:8
        CMP.B      #4:8,R0L
        BEQ        L10:8
        BRA        L11:8
L6:
L7:     SUB.L      ER0,ER0
        BRA        L13:8
L8:
L9:     SUB.L      ER0,ER0
        MOV.B      #1:8,R0L
        BRA        L13:8
L10:    SUB.L      ER0,ER0
        MOV.B      #2:8,R0L
L13:   MOV.L      ER0,@ER1
L11:    ADDS.L     #2,SP
        RTS
_ll:    .RES.L     1
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	49	45	63	59	69
改善後	45	43	57	53	61

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	55	55	47
改善後	51	51	45

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	33	30	82	72	68
改善後	20	18	82	72	68

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	24	24	23
改善後	13	14	12

■備考

c=4 の場合で測定しています。

H8SX を除く CPU では、コンパイラ Ver4.0 より case 文の飛び先をまとめるように改善しましたので、アセンブラ展開コードは、同じになります。

本項のアセンブラ展開コード、オブジェクトサイズ、実行処理速度は、コンパイラ Ver3.0 にてコンパイルした結果を記述します。(H8SX 以外)

また、一般的に switch 分の case 値のうち、よく実行されるものを判定の早い方に持ってくると、実行結果が良くなります。実行時に試してみてください。

6.7.3 直下の関数への分岐

サイズ効率	○	処理速度	○	スタックサイズ	○

■ポイント

関数の末尾に関数呼び出しがある場合は、呼び出される関数は直下に置く。

■説明

テイルリカージョン最適化がかかった場合、呼び出される関数が直下にある場合は、最適化がかかり、関数呼び出しのコードが削除されます。

このコードの分、サイズが縮小され、処理速度も速くなります。

■使用例

関数 main から関数 func を呼び出します。

（改善前のCプログラム）

```
int a;
void func();
void func()
{
    a++;
}
void main()
{
    a=0;
    func();
}
```

（改善後のCプログラム）

```
int a;
void func();
void main()
{
    a=0;
    func();
}
void func()
{
    a++;
}
```

（改善前のアセンブラ展開コード）

```
_func:
       MOV.L   #_a,ER0
       MOV.W   @ER0,R1
       INC.W   #1,R1
       MOV.W   R1,@ER0
       RTS
_main:
       SUB.W   R0,R0
       MOV.W   R0,@_a:32
       BRA     _func:8
```

（改善後のアセンブラ展開コード）

```
_main:
       SUB.W   R0,R0
       MOV.W   R0,@_a:32
_func:
       MOV.L   #_a:32,ER0
       MOV.W   @ER0,R1
       INC.W   #1,R1
       MOV.W   R1,@ER0
       RTS
```

オブジェクトサイズ表 [バイト]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	26	20	24	20	20
改善後	24	18	22	18	18

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	18	18	14
改善後	16	16	12

実行処理速度表 [サイクル]

CPU 種別	H8S/2600,H8S/2000		H8/300H		H8/300
	ADV	NML	ADV	NML	
改善前	21	17	40	34	34
改善後	19	15	36	30	30

CPU 種別	H8SX		
	MAX	ADV	NML
改善前	14	14	13
改善後	12	12	11

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注意してください。
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		sni,pro ファイルのオープン	7.3.3
21		スタック情報の変更	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	7.3.6
24		その他	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**

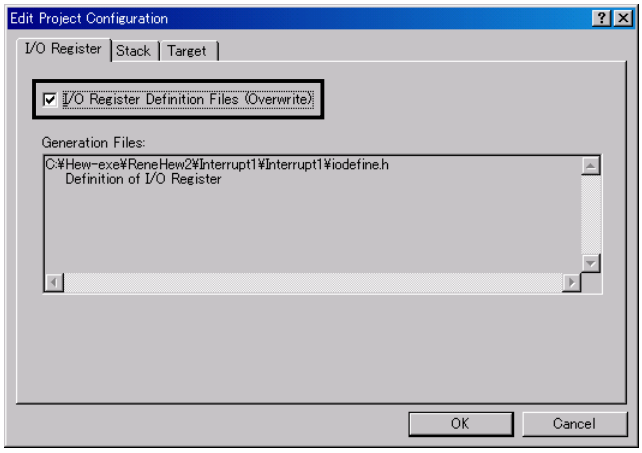
■再生成できるファイル

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時は、iodefine.h が存在しても上書きすることができます。

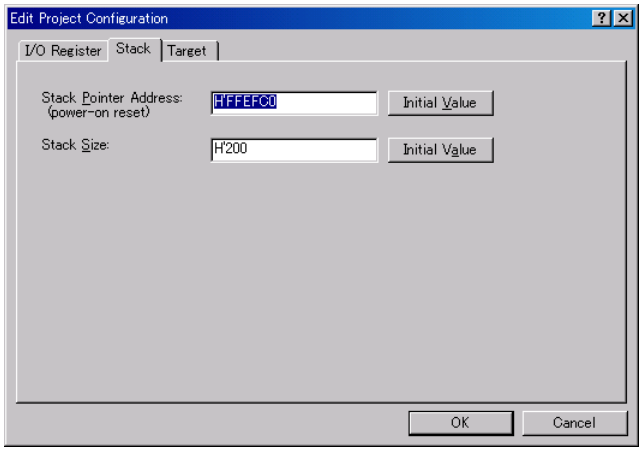


■再編集できるファイル

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

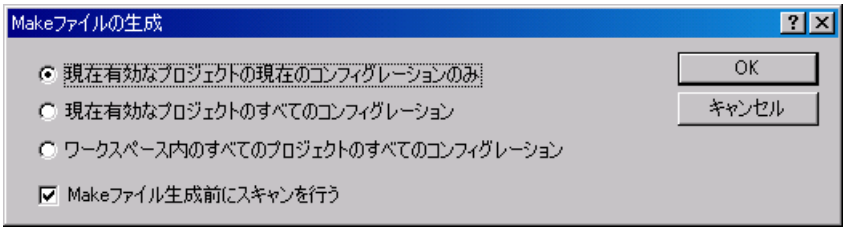
■説明

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。

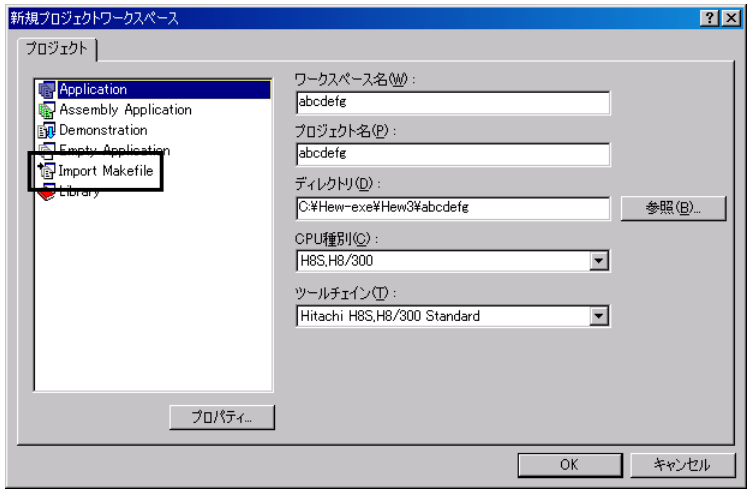
7.1.3 メイクファイルの入力

■説明

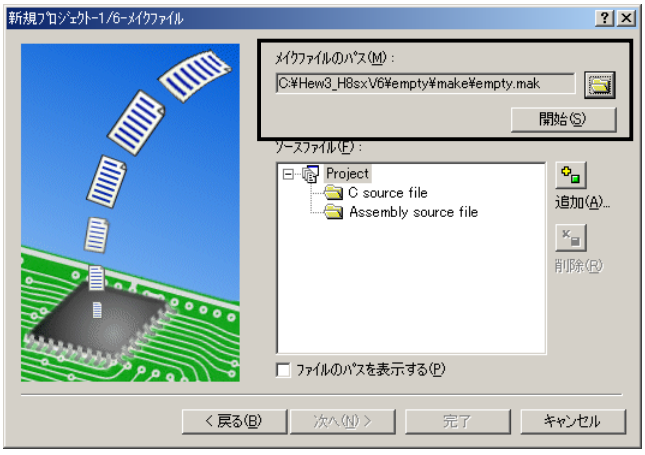
HEW では、HEW で作成したメイクファイルや UNIX 環境で使用したメイクファイルを入力することができます。
メイクファイルからプロジェクトのファイル構成を自動で取得することができます。
（オプション指定などは取得できません。）
これにより、コマンドラインから HEW への移行が容易になります。

■メイクファイルの入力方法

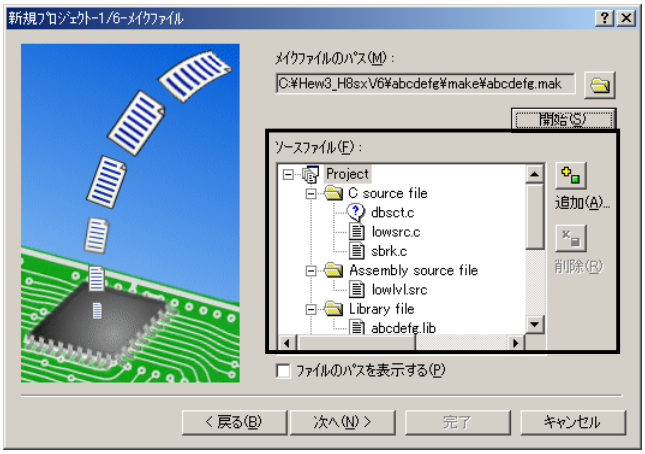
1. 新規ワークスペース作成時、新規プロジェクトワークスペースダイアログのプロジェクトタイプの中から Import Makefile を選択します。



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

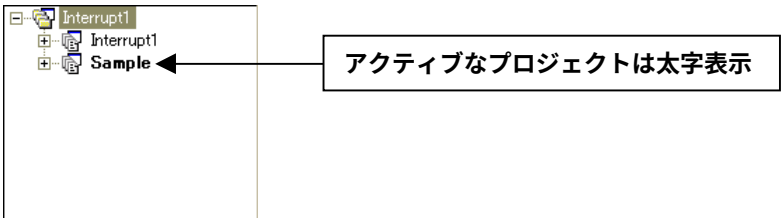
■説明

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、他のユーザが別のマシンでプログラム開発することができます。

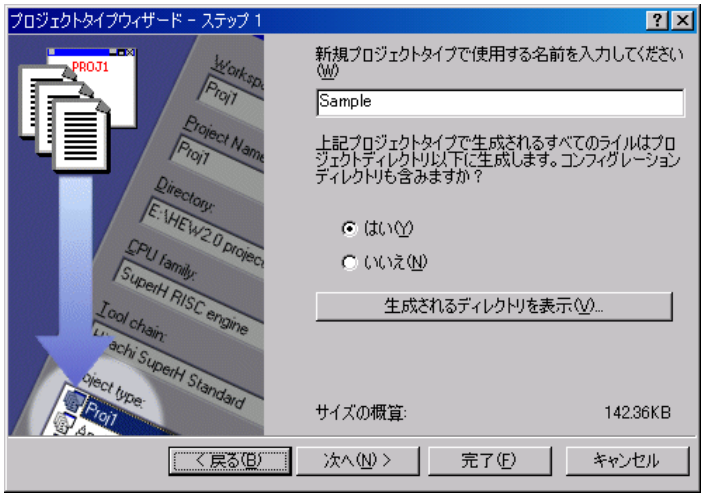
雛型として保存できる情報は、プロジェクトファイル構成やビルドオプションやデバッガ設定状況などのプロジェクト内容をすべて保存することが可能です。

■プロジェクトタイプ保存方法

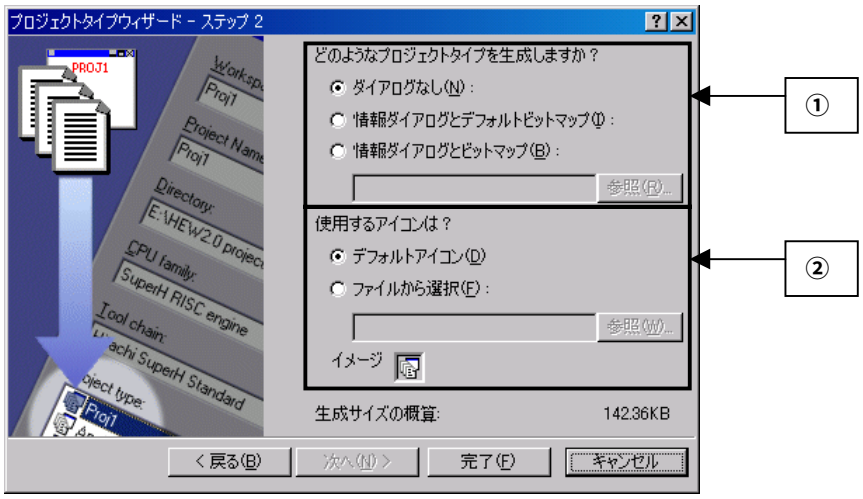
1. ワークスペースをオープンしている状態で、アクティブになっているプロジェクトが保存されるプロジェクトになるので、対象プロジェクトをアクティブにします。プロジェクトをアクティブにするには[プロジェクト->アクティブプロジェクトに設定]でプロジェクトを選択します。



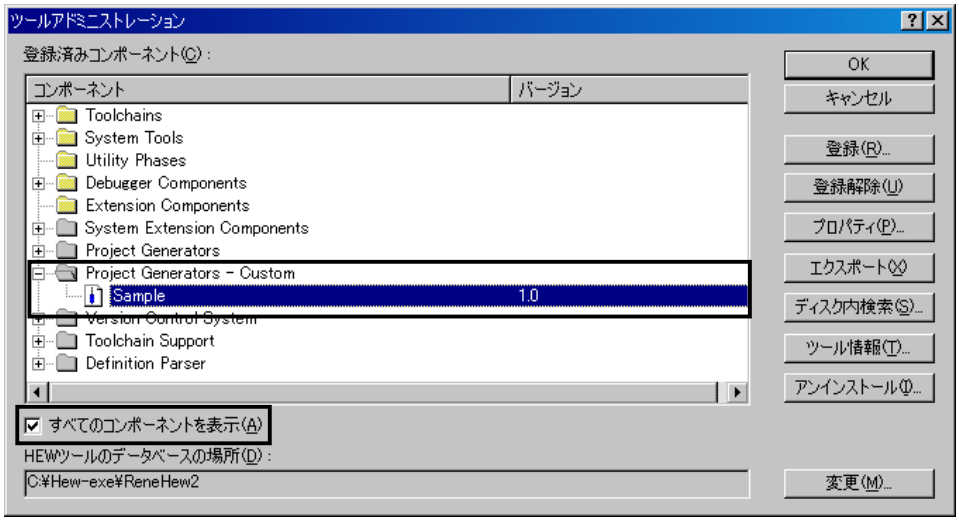
2. [プロジェクト->テンプレートの作成...]により以下のプロジェクトタイプウィザードを表示します。雛型として使用するプロジェクトタイプの名前を決めて、ビルド後の実行ファイルなどを含むコンフィグレーションディレクトリについても雛型とするか選択します。
   ここで完了ボタンを押下し、プロジェクトタイプウィザードを終了しても構いません。



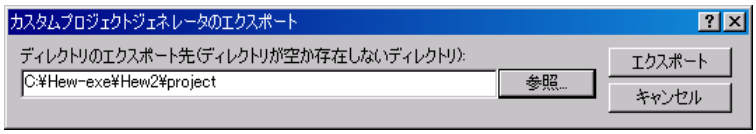
3. [プロジェクトタイプウィザード – ステップ 1]で[次へ]ボタンを押下すると以下のウィザードが表示されます。以下の①ではプロジェクトタイプの雛型をオープンするときに、プロジェクトの情報やビットマップを表示するか選択します。
   また、②ではプロジェクトタイプのアイコンをユーザ指定のアイコンに変更することもできます。その後、完了ボタンを押下します。
   これらの指定は必須ではありません。



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。



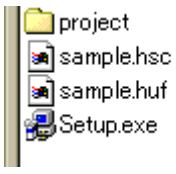
5. すると、以下のダイアログが表示されるのでカスタムプロジェクトジェネレータを出力するディレクトリを指定します。ディレクトリの中身は空である必要があります。
これで、プロジェクトタイプの保存は終了です。



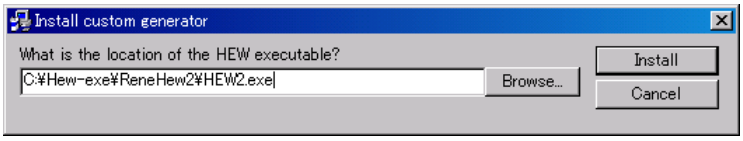
■カスタムプロジェクトジェネレータのインストール

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。

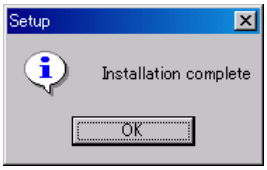
1. **「プロジェクトタイプ保存方法」**の 5.で作成したディレクトリに、以下のようにインストール環境が作成されます。
（インストール環境ディレクトリ）



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）



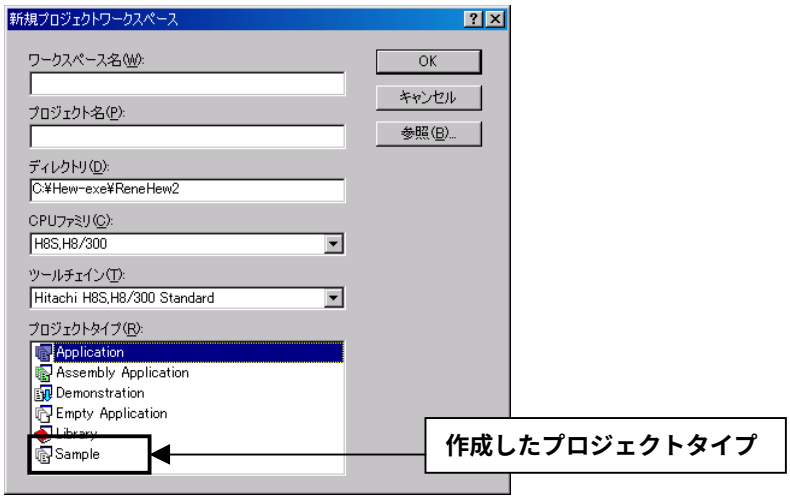
3. これで、環境の構築は終了です。



■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

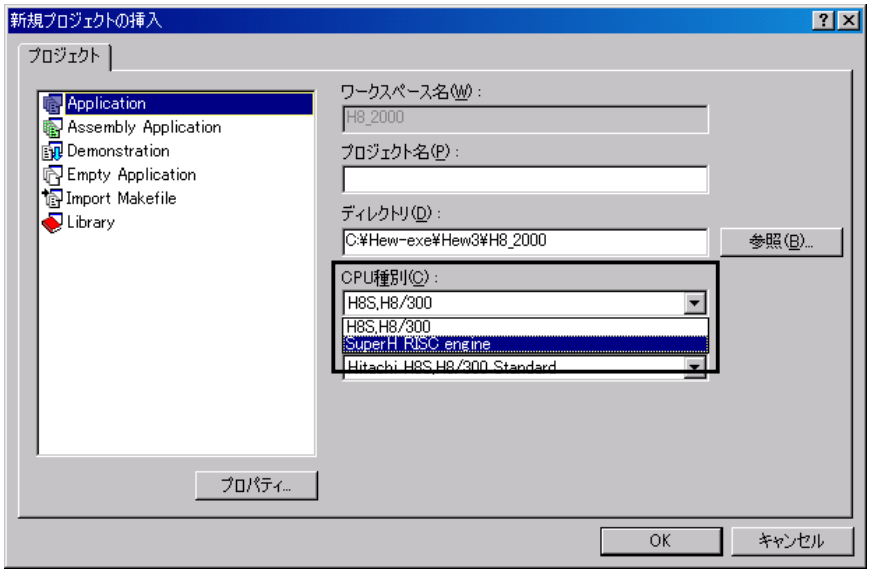
■説明

ワークスペースに新規プロジェクトを挿入する時、別の種類の 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。

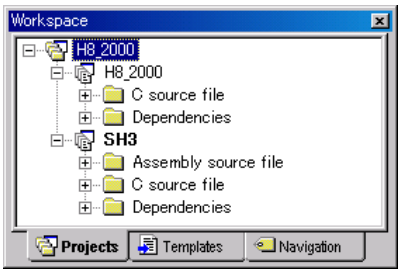
■CPU 別ファミリ挿入の例

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、別の種類の CPU 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

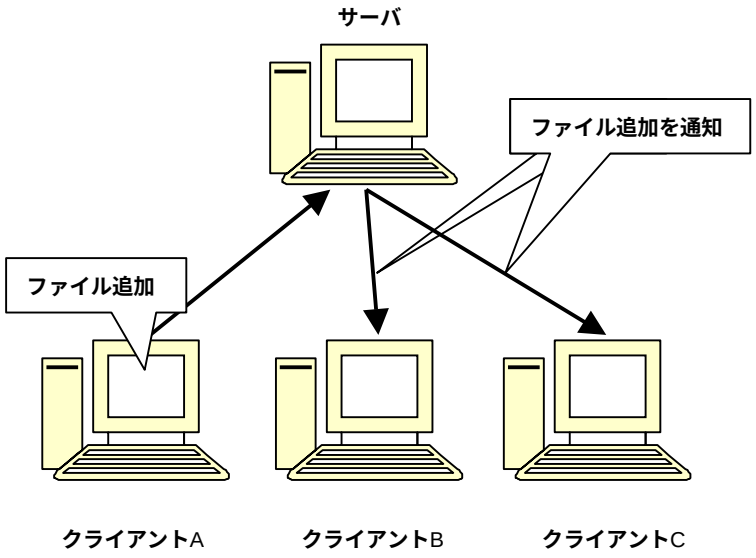
本機能は HEW3.0 以降でサポートしています。

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。これにより、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。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。

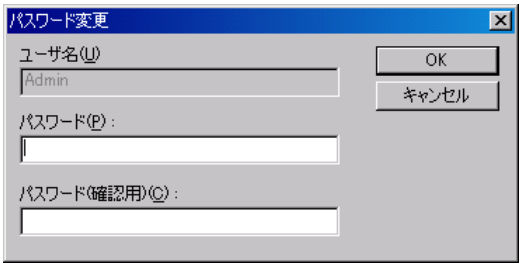


■ネットワークアクセスの設定

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。

オプションダイアログ／ネットワークタブ

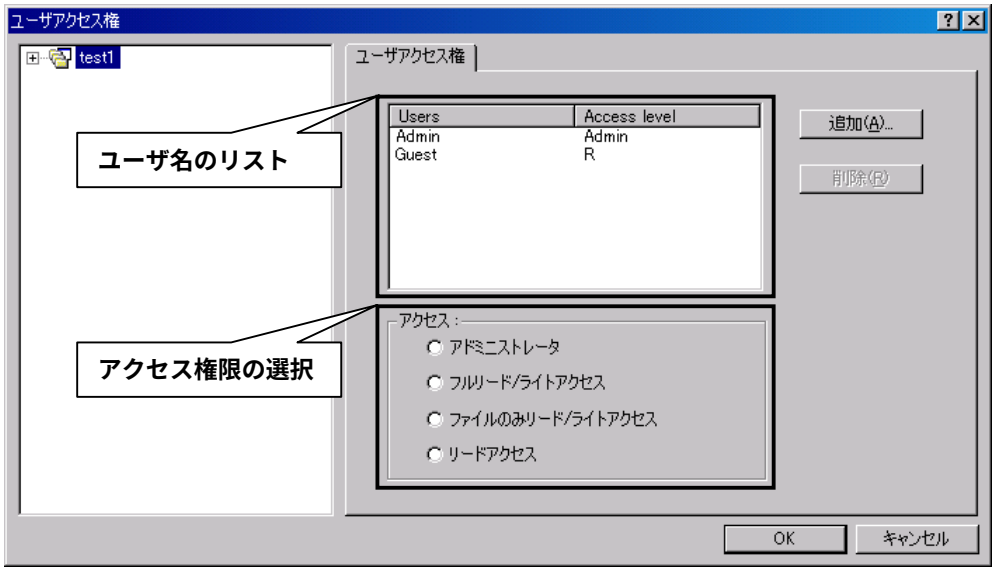
パスワード変更ダイアログ



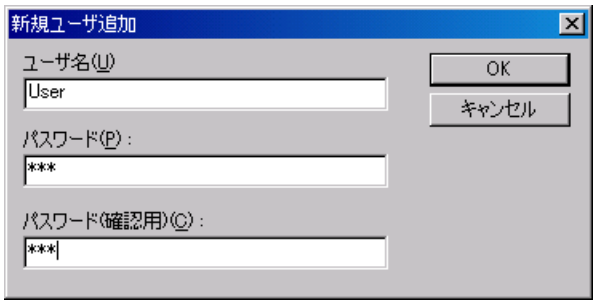
■新規ユーザの追加

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。

■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:H8C 3.0C など）で作成したワークスペースを新バージョン（例 HEW3.0:H8C 6.0）でオープン時、以下のダイアログボックスを表示します。

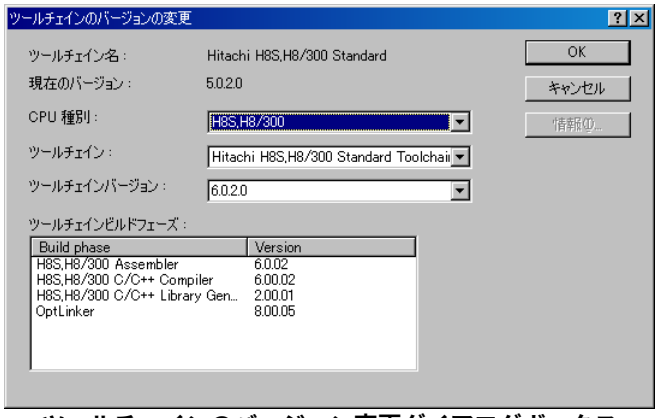
(1) アップグレード対象プロジェクトの確認

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

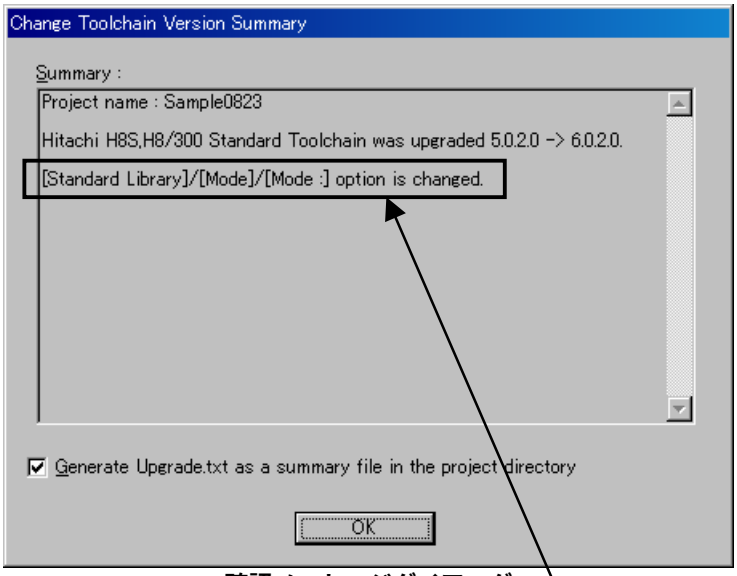
アップグレードできるバージョンを選択します。



ツールチェインのバージョン変更ダイアログボックス

(3) 確認メッセージ

C/C++コンパイラ Ver4.0 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。

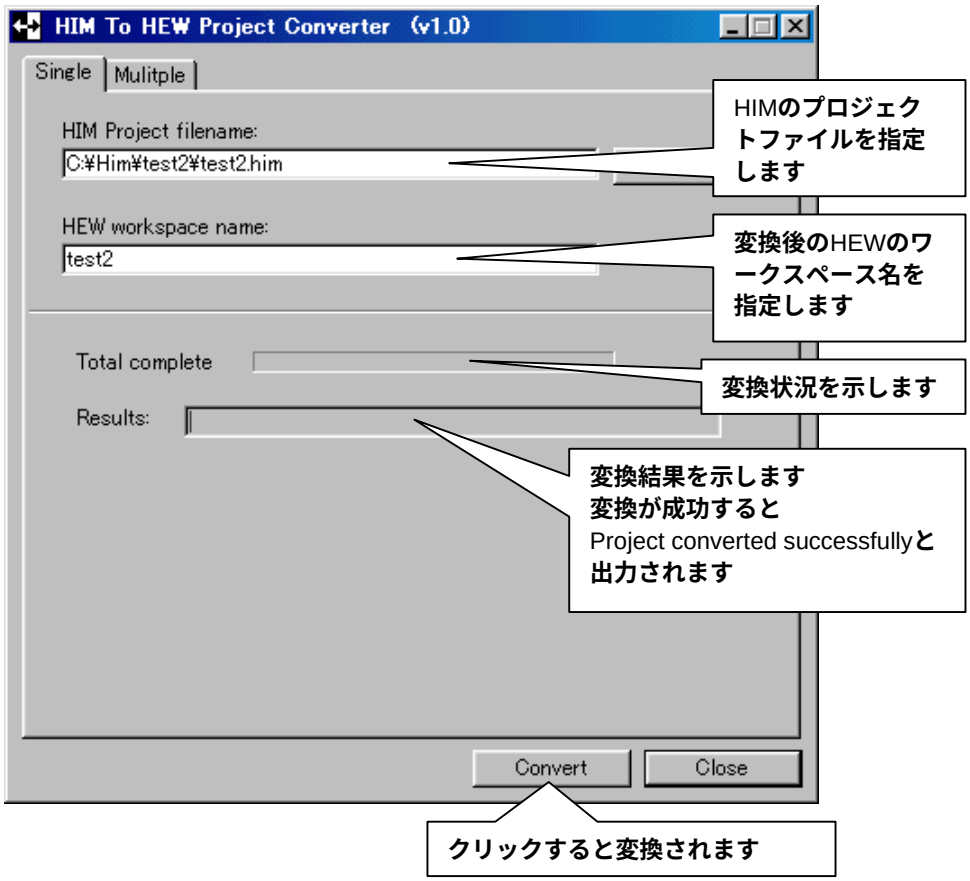
Windows®[スタートメニュー]の[プログラム(P)]の[Renesas High-performance Embedded Workshop]から[Him To Hew Project Converter]を選択します。

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。

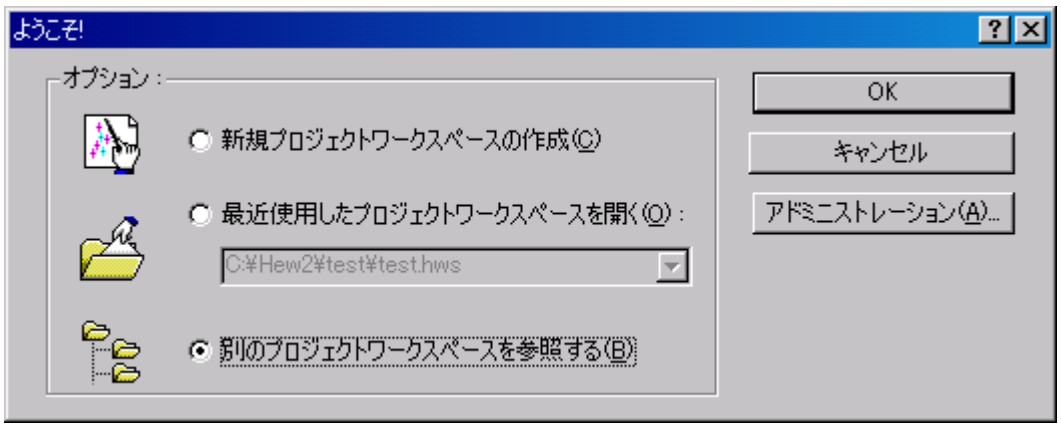
複数の HIM プロジェクトを HEW プロジェクトに変換し、HEW ワークスペースに一括登録する場合は、Multiple タブを選択します。

(1) Single タブ

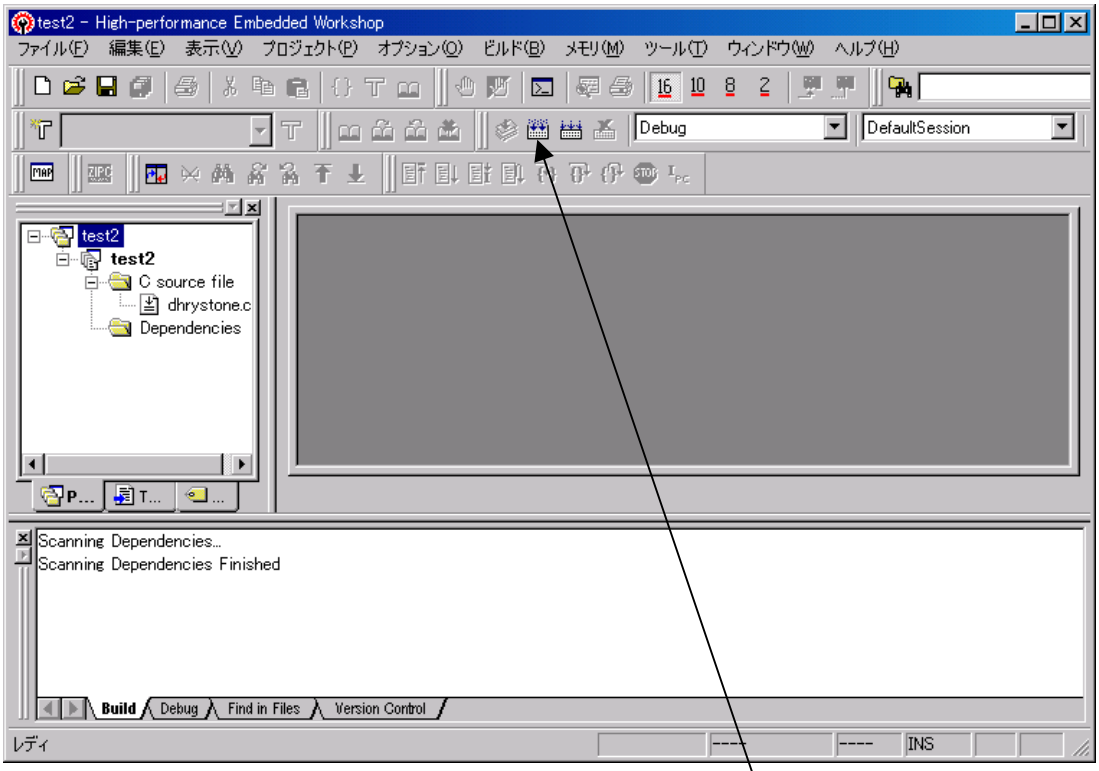


次に HEW を立ち上げます。

別のプロジェクトワークスペースを参照するを選択し、[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。



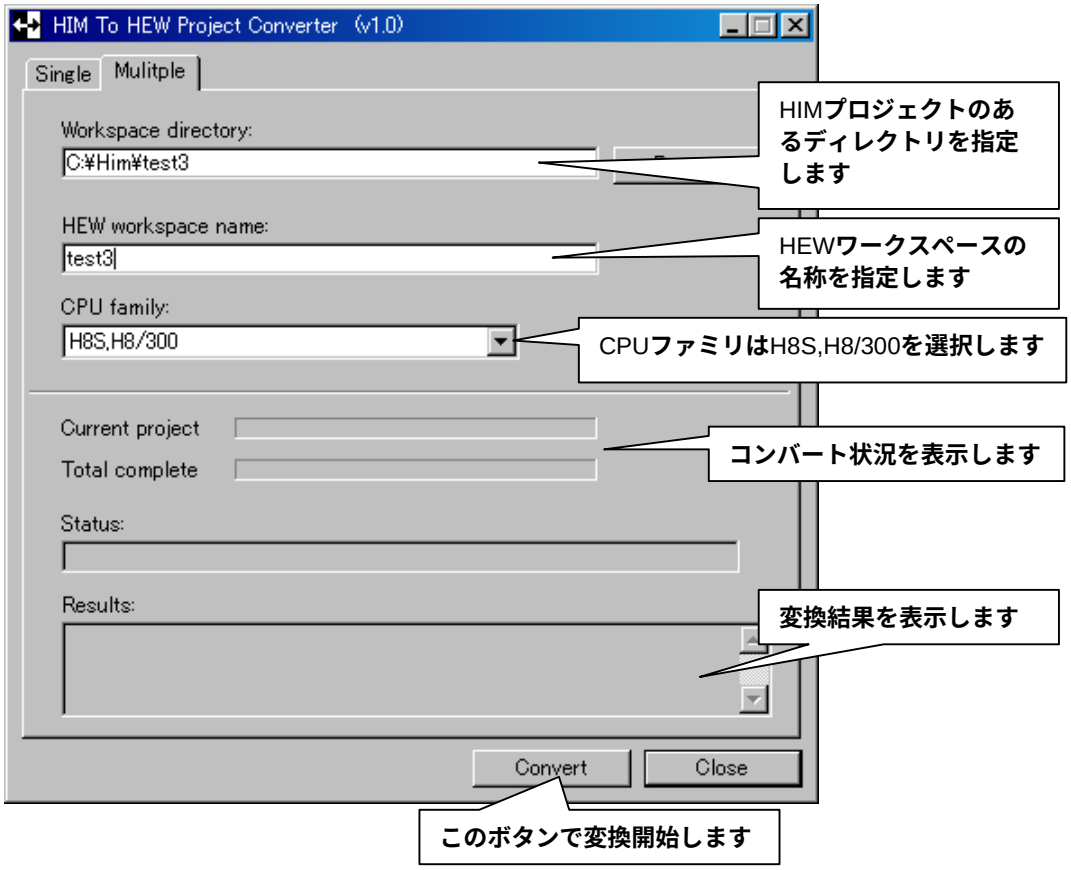
次のように HEW プロジェクトが開かれました。



[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。

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。

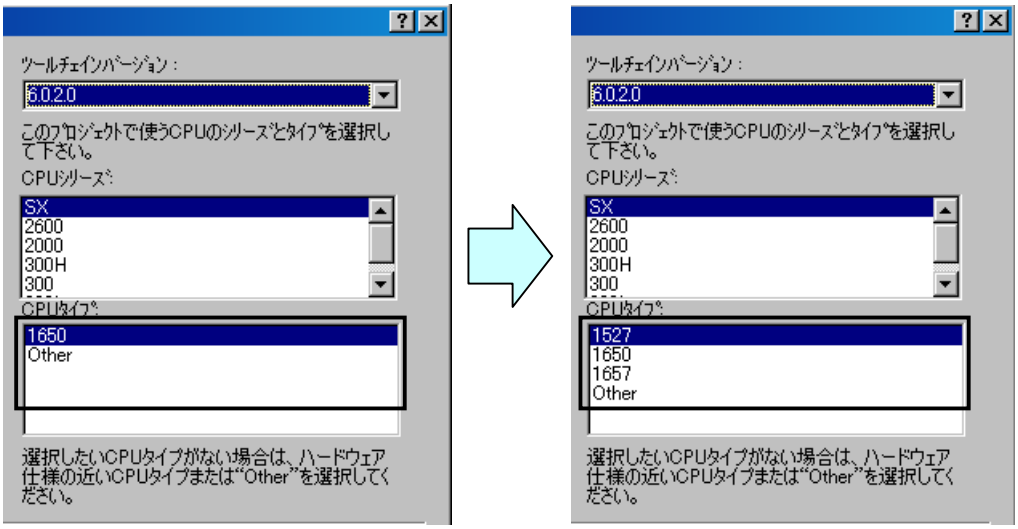
また、自動生成ファイルをバグ対応版に更新することもできます。

■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。
あわせて注意事項など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

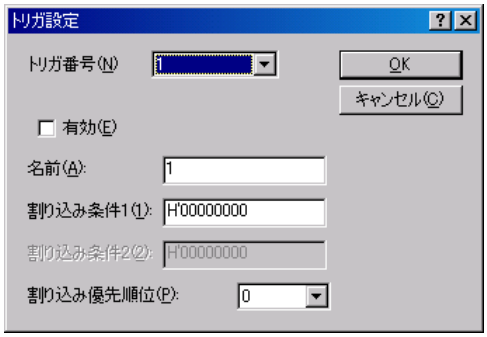
この擬似割り込みはボタンを、ある割り込み要因に見立てて、ボタンを押下することにより手動で擬似的な割り込みを発生させることができます。

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。

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

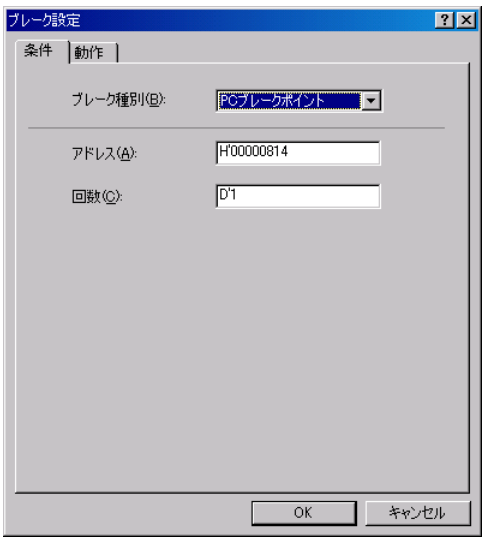
HEW2.2 **以前**：表示->コード->ブレークポイント
HEW3.0 **以降**：表示->コード->イベントポイント
【注】 HEW3.0 **以降の場合は、ブレークポイントビューの** Software Event **タブをクリック**

■ファイル入力設定例

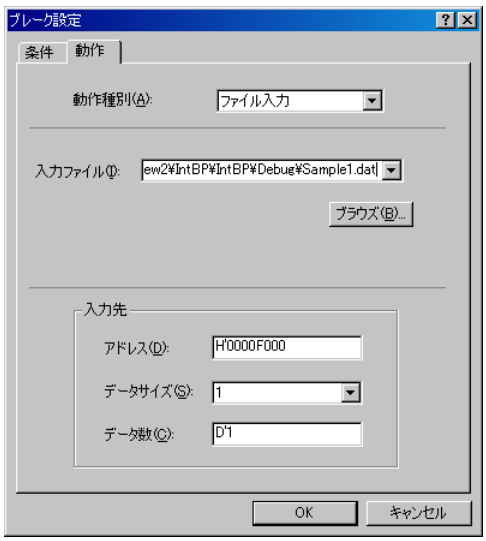
ブレークポイントビュー上で右クリック後に[設定…]を押下し、以下にブレーク設定ダイアログを表示します。PC ブレークポイントを使い、PC が以下のアドレス時にブレーク条件が整う例です。他のブレーク種別でも設定方法は同様です。

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）



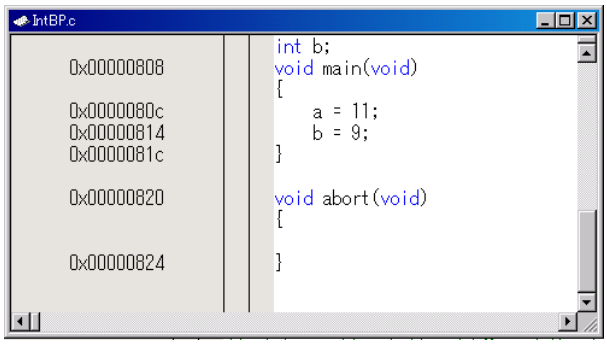
（動作タブ）



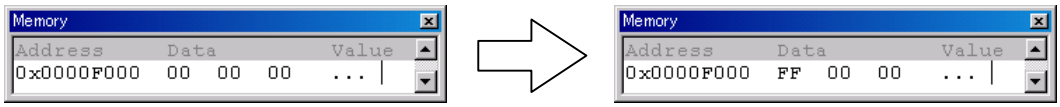
■ファイル入力動作例

次に実際の動作例を示します。
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を実行させます。

（ソース例）



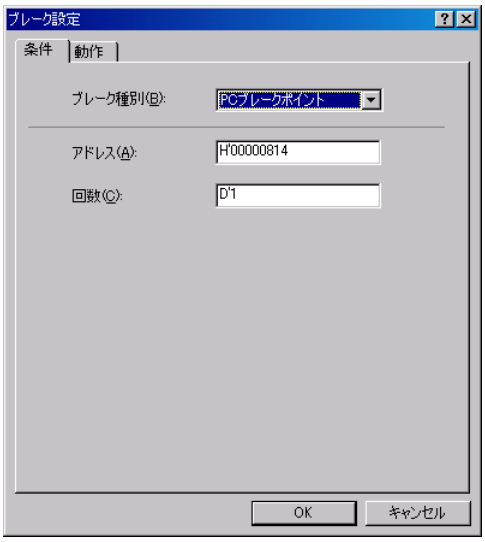
すると、PC の[H'00000814]を通過するときに条件が成立し、以下のように H'F000 番地のメモリ内容が変化することを確認できます。



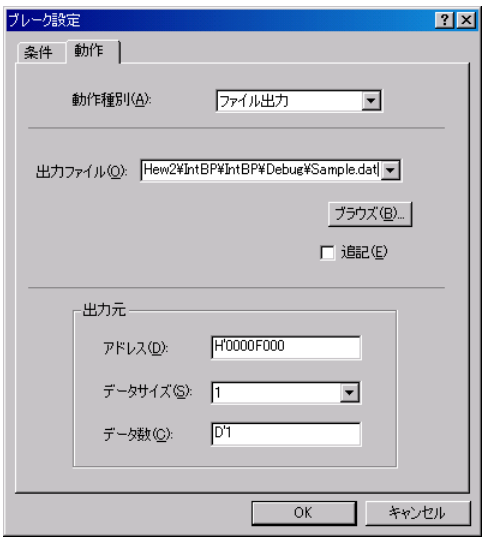
■ファイル出力設定例

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います。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）



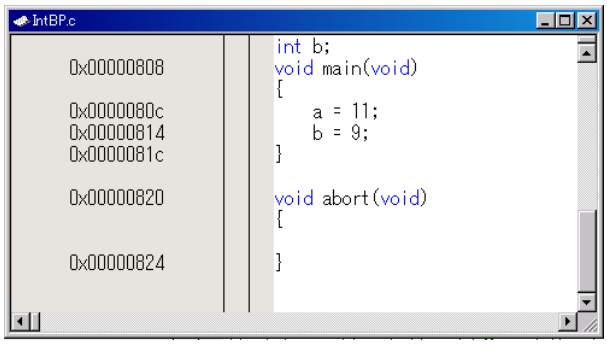
（動作タブ）



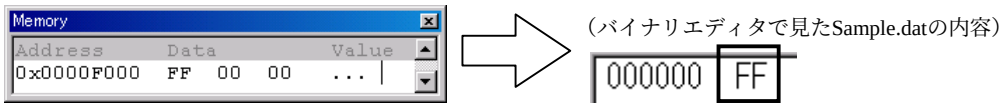
■ファイル出力動作例

次に実際の動作例を示します。
上記の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を実行させます。

（ソース例）



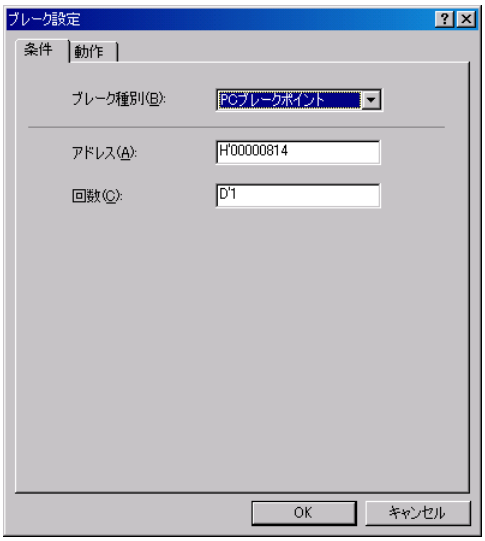
すると、PC の[H'00000814]を通過するときに条件が成立し、以下のように H'F000 番地のメモリ内容をファイルに出力することが確認できます。



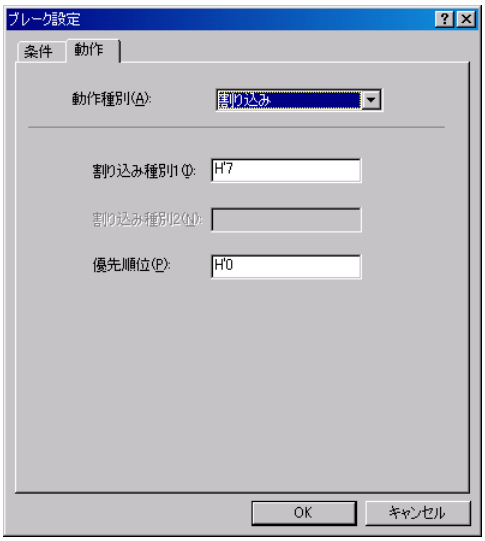
■割り込み設定例

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。

（条件タブ）



（動作タブ）



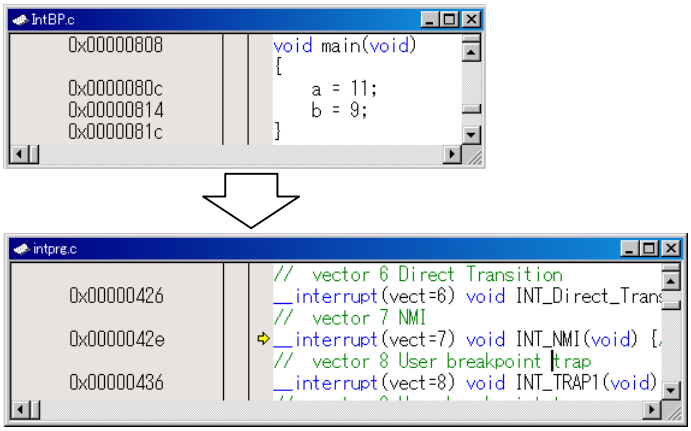
■割り込み動作例

次に実際の動作例を示します。

上記の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を実行させます。

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。

（ソース例）



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。

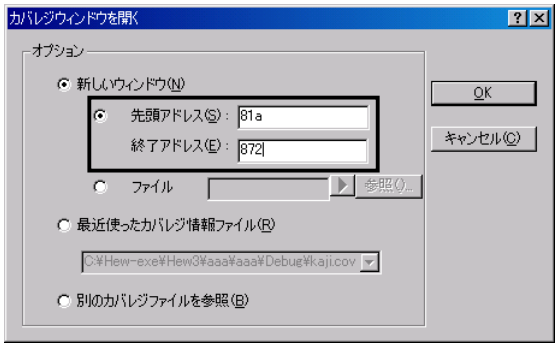
■[カバレジウインドウを開く]ダイアログの表示方法

表示->コード->カバレジ

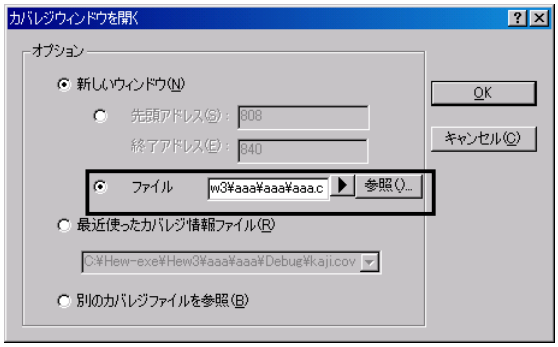
■新規のカバレジ情報収集方法

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。

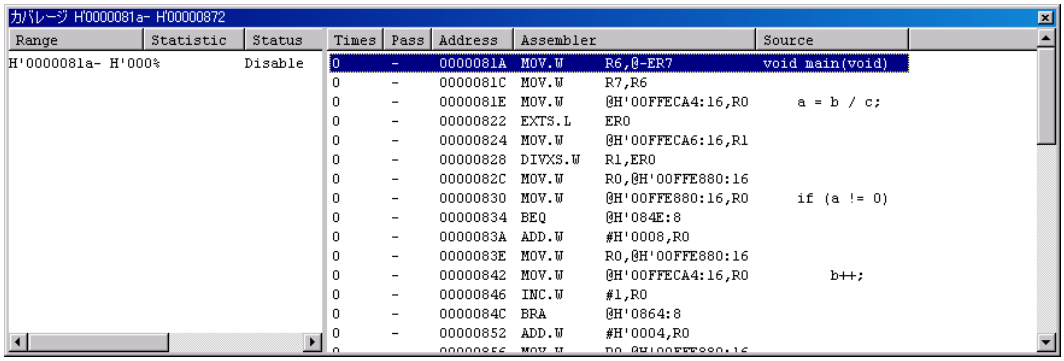
（アドレス指定）



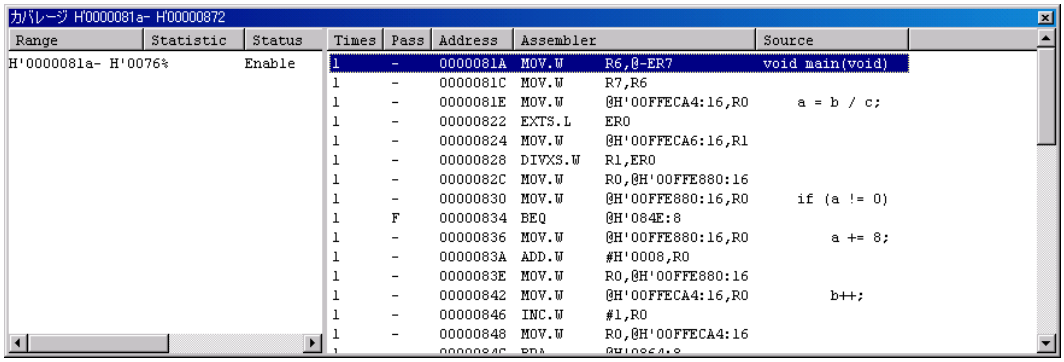
（ファイル名指定）*HEW3.0以降サポート



2. OK ボタンを押下すると以下のカバレー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。

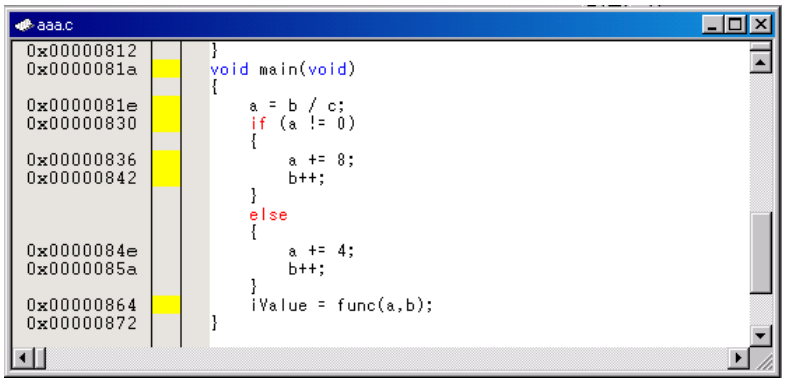


3. 次に実際にプログラムを実行してみましょう。そうすると、カバレージビュー右側の Times 項目が 1 に変化している箇所があります。これは該当アドレスの命令を実行したことを意味します。
   また、左側には該当アドレス範囲の C0 カバレジ値が表示されます。



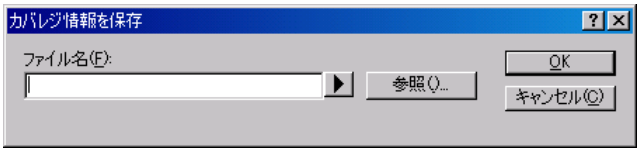
【注】カバレジビューの左側は HEW3.0 以降の場合のみ存在します。

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。



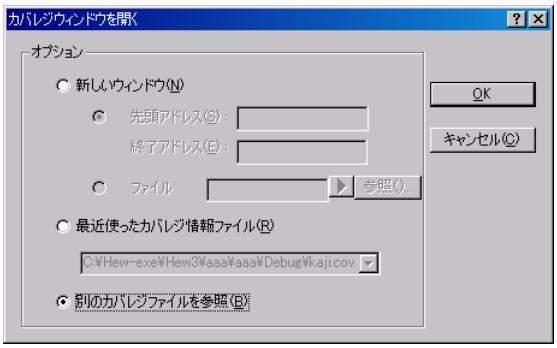
■カバレジ情報の保存方法

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。

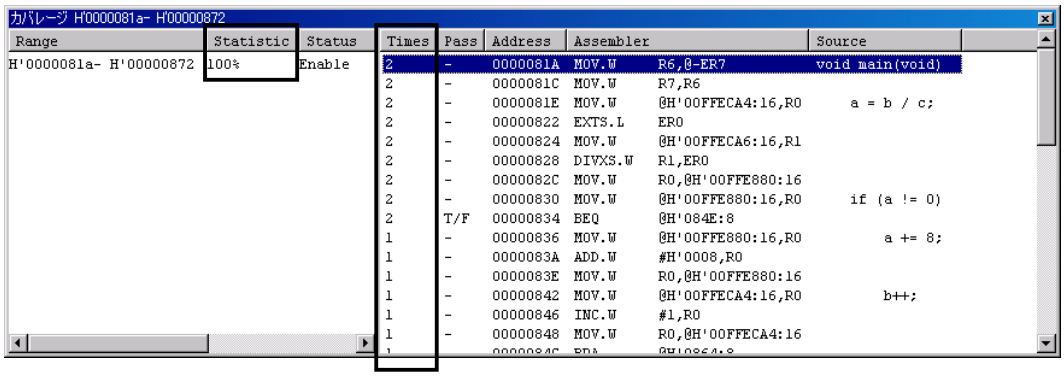


■既存カバレジ情報使った情報収集

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することはできません。
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が考えられます。
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。



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います。
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)以上に設定してください。

（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

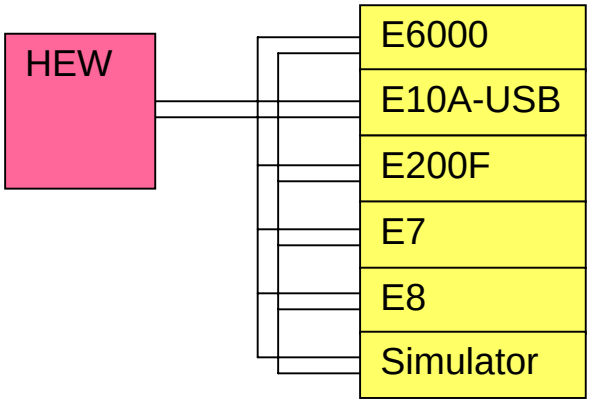
入出力先を変更するには、lowsrc.c 内_INIT_IOLIB 関数の ここ を変更します。

```
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

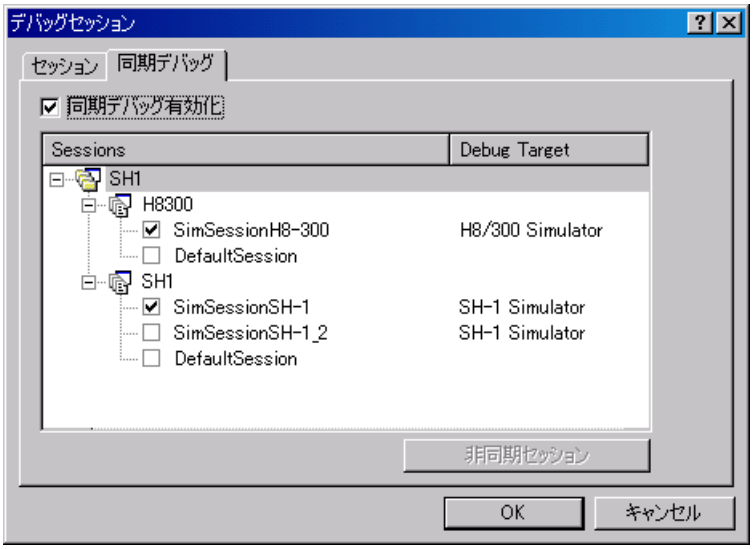
■説明

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。

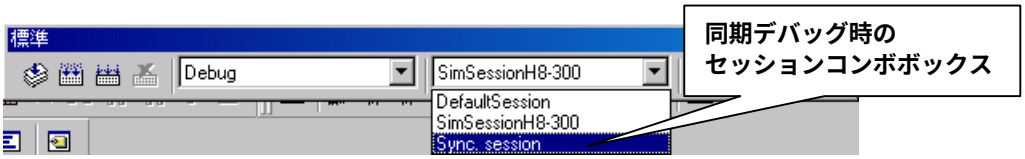


■デバッガターゲット同期の設定方法

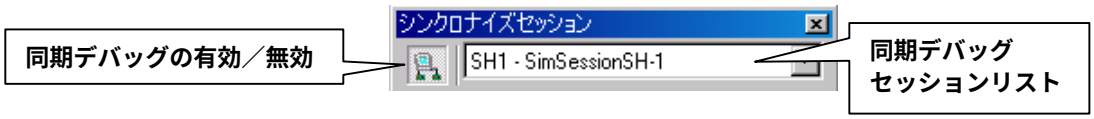
1. [オプション->デバッグセッション]で以下のダイアログを表示し、同期デバッグタブをクリックします。
   ここで、同期させたいセッションをチェックし、その後に同期デバッグ有効化チェックボックスをチェックします。



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。

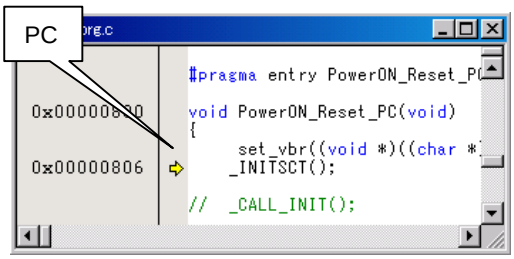
ユーザ操作	ターゲットデバッガセッション 1	ターゲットデバッガセッション 2
いずれかのセッションでの[実行]	“ 実行 ”	“ 実行 ”
いずれかのセッションでの[ステップ]	“ ステップ ”	“ ステップ ”
いずれかのセッションでの ESC 押下	“ 停止 ”	“ 停止 ”
-	ブレークポイントまたはユーザプログラム不正による “ 停止 ”	実行停止（ESC 押下による停止と同じ結果）
-	実行停止（ESC 押下による停止と同じ結果	ブレークポイントまたはユーザプログラム不正による “ 停止 ”
いずれかのセッションでの[CPU リセット]	“ CPU リセット ”	“ CPU リセット ”

■同期デバッグ例

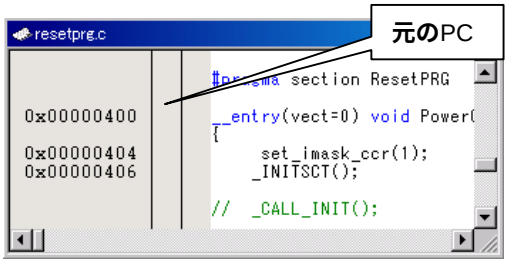
次にステップコマンドを実行した場合の例を示します。

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



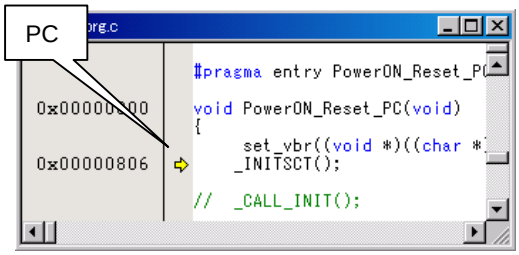
H8300 - SimSessionH8-300の状態



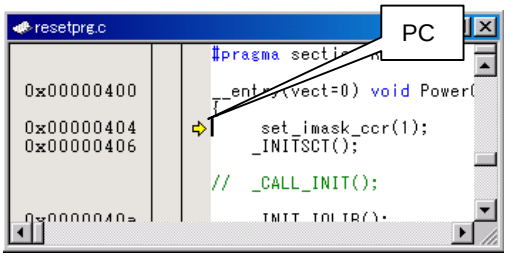
2. 次にシンクロナイズセッションツールバーでセッションを切り替えます。

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。

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／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。

■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
H8SX	TPU0	TSTR	△
		TCR	△
		TIER	○
		TSR	○
		TCNT	○
		TGRA	○
		TGRB	○
		TGRC	○
		TGRD	○

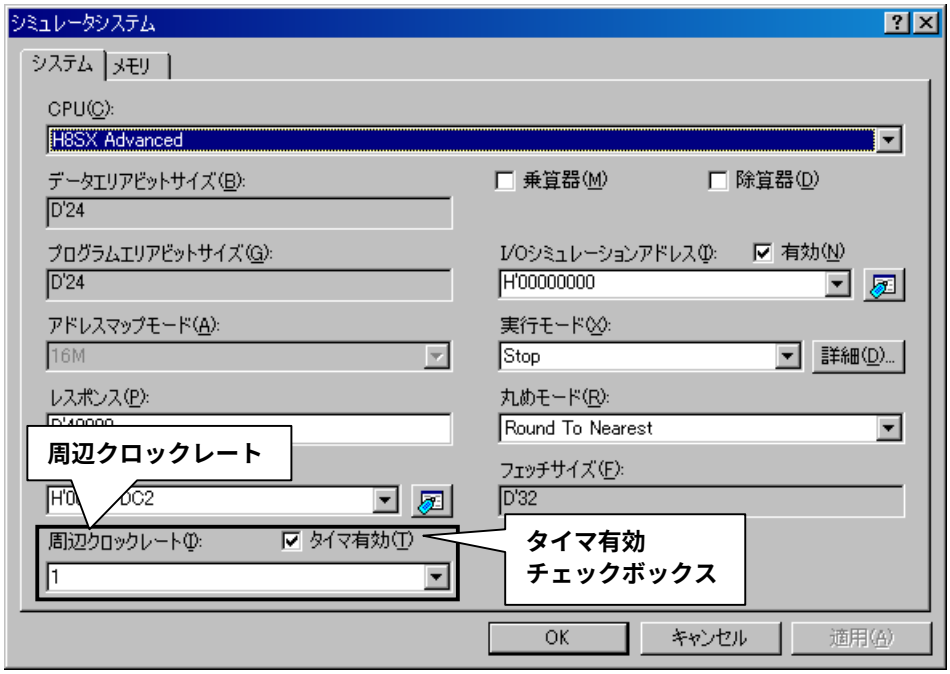
■サポートしている割り込み優先度レベル設定レジスタ

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
H8SX	IPRF	△

■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。

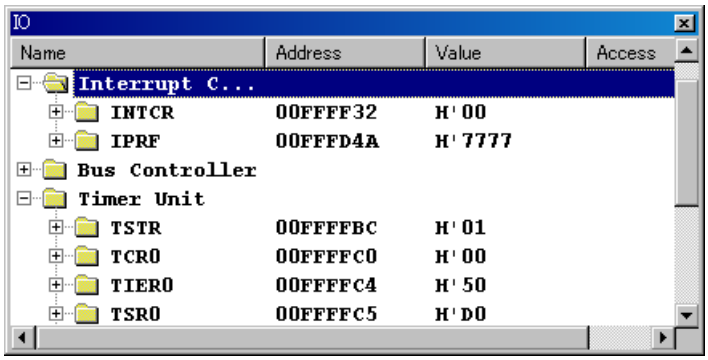
```
// TPU0 start
TPU.TSTR.BIT.CST0 = 1;
// TPU0 Overflow interrupt enable
TPU0.TSR.BIT.TCFV = 1;
TPU0.TIER.BIT.TCIEV = 1;
while(1);
```

TPU0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 HEW3.1 以降でサポートしています。
また、CPU が H8SX のときのみサポートしています。

7.2.7 タイマ使用の具体例

■説明

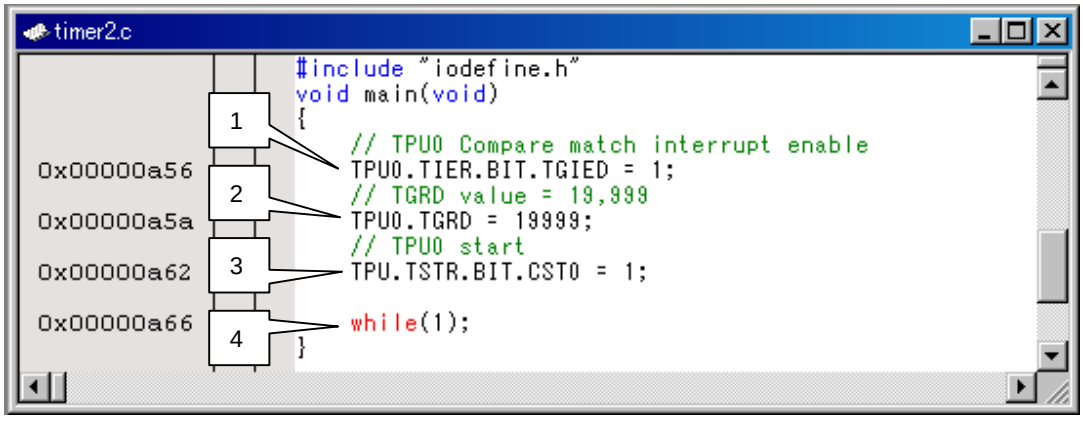
ここでは H8SX/1650(H8SX)の TPU を例にとり、コンペアマッチ、周期ハンドラによる割り込みの使用方法を紹介します。

■HEW の設定

「7.2.6 **タイマ使用方法■タイマシミュレーション方法」**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。



【割り込み発生プログラム説明】

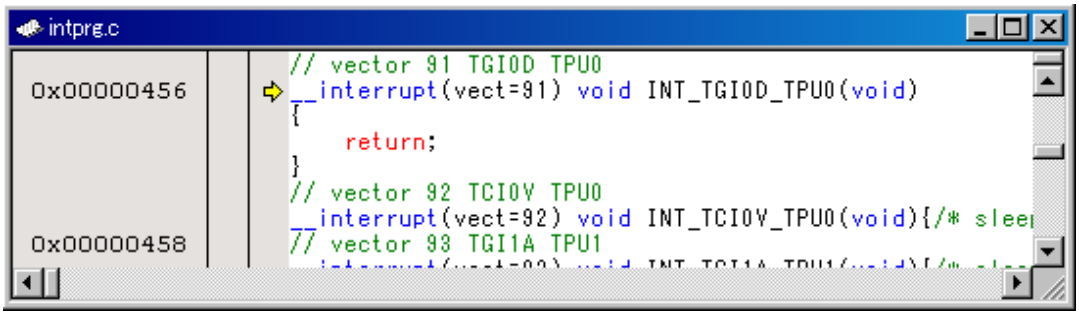
1. TIER(タイマインタラプトイネーブルレジスタ)で TGIED（TGR インタラプトイネーブル D）ビットが、1 になったときの割り込みを許可します。
2. TGRD の値を設定します。
3. TPU0 のタイマをスタートさせます。
4. TCNT0 と TGRD の値が一致するのを、待ちます。（コンペアマッチ発生待ち）

■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 TGRD（タイマジェネラルレジスタ D）が一致する（コンペアマッチ）状態を待ちます。

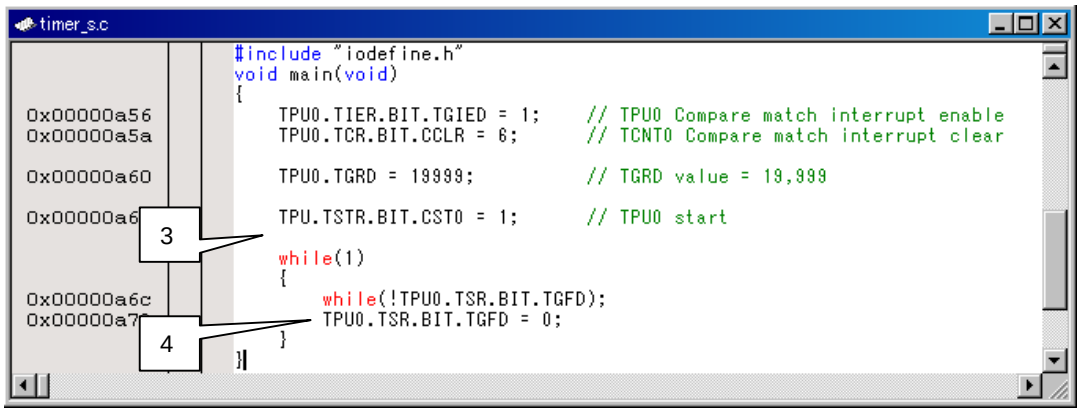
一致すると、コンペアマッチ割り込みが発生し、以下に示す割り込みルーチンが呼び出されます。

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。



■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、IPRF（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。



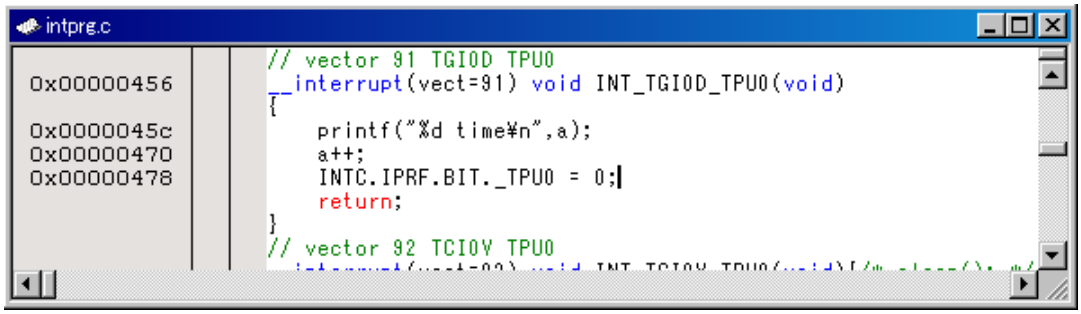
1. TIER（タイマインタラプトイネーブルレジスタ）で TGIED（TGR インタラプトイネーブル D）ビットが、1 になったときの割り込みを許可します。
2. TGRD の値を設定します。
3. TPU0 のタイマをスタートさせます。
4. コンペアマッチのフラグをクリア。

■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。



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

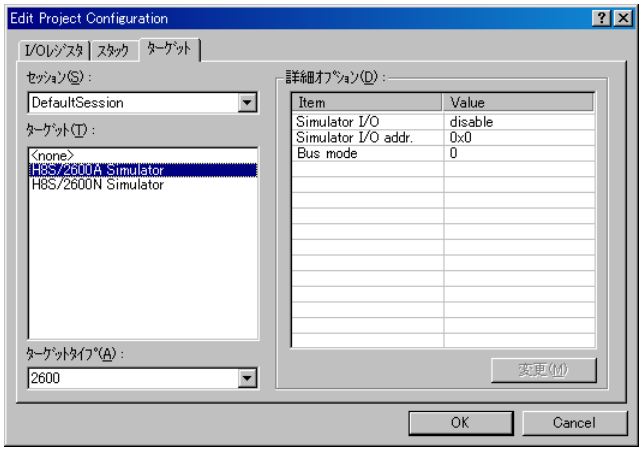
HEW メニュー：**プロジェクト>構成の編集**

■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。

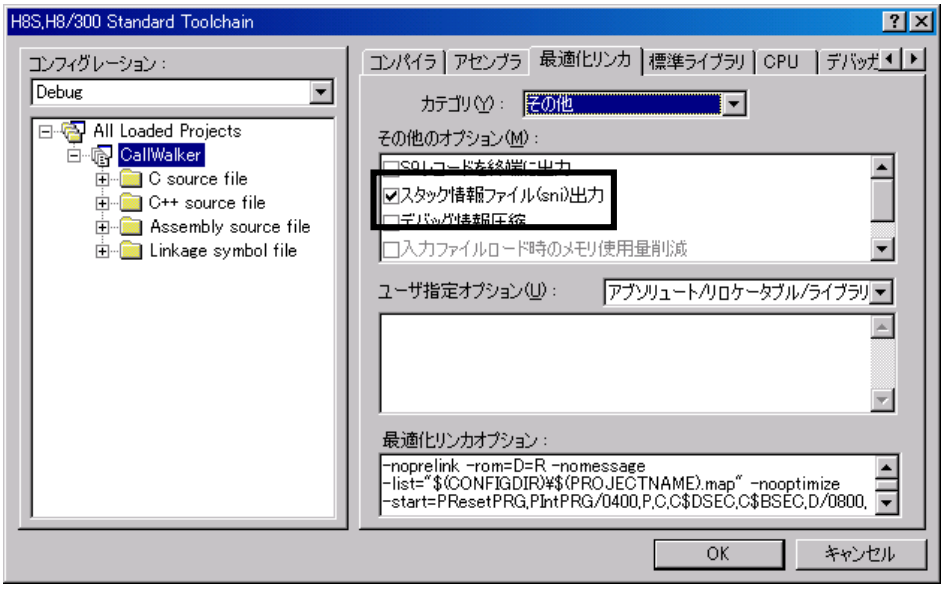
編集したスタック使用量に関する情報は、呼び出し情報ファイル（*.cal）として保存・読み込みが可能です。また、呼び出し情報ファイルをマージ（連結）することも可能です。

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*

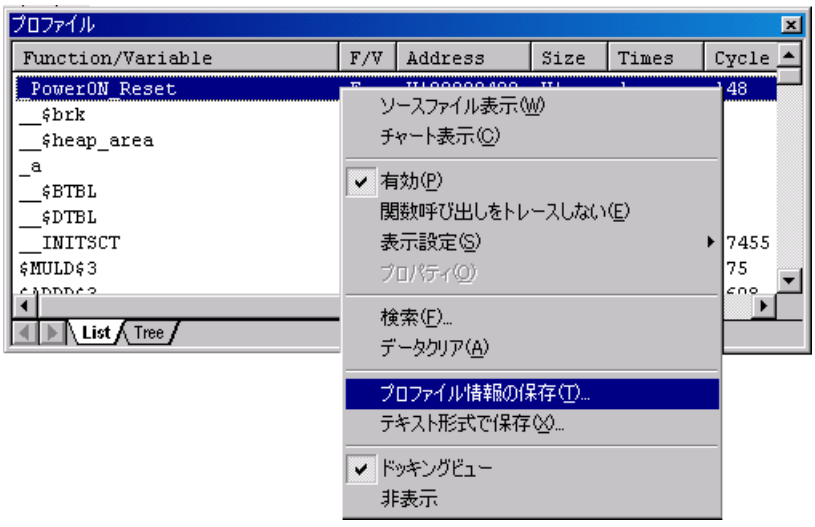
■プロファイル情報ファイル(*.pro)の作成方法

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、H8S,H8/300 **シリーズ** High-performance Embedded Workshop 3 **ユーザーズマニュアル「**4.13　**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

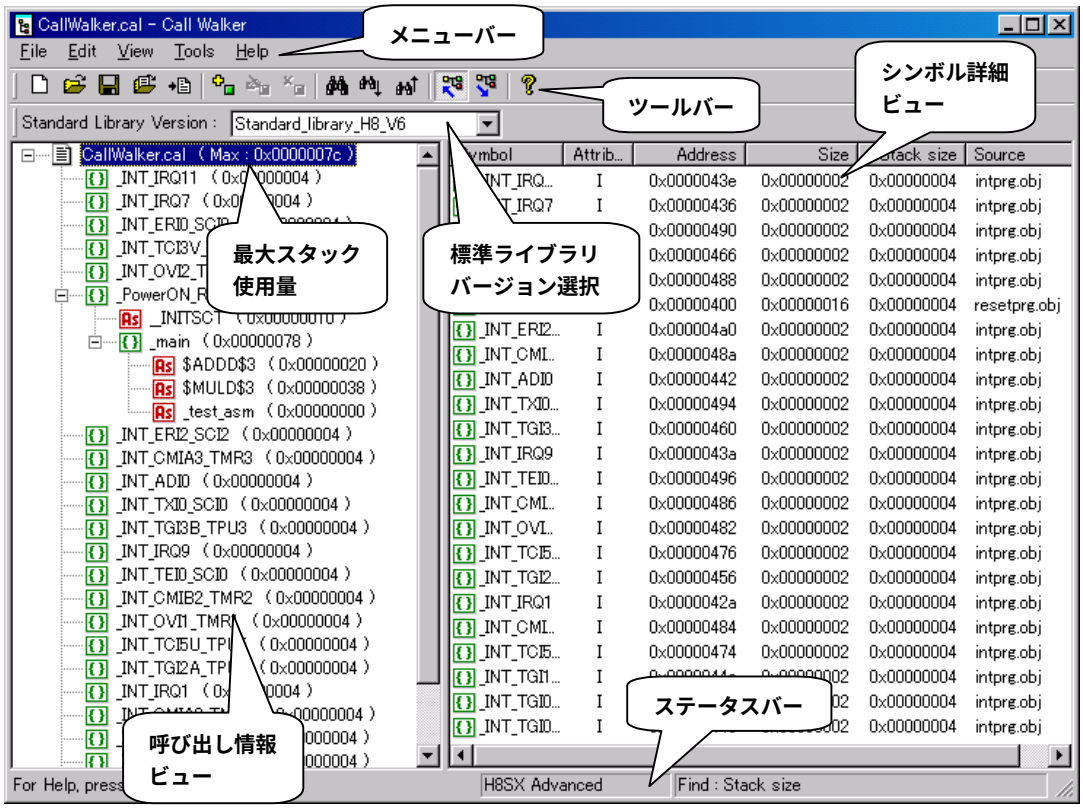
7.3.3 ファイルのオープンと Call Walker の画面

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。



■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**

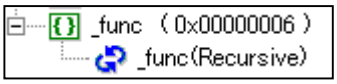

: **C/C++の関数**


: **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】

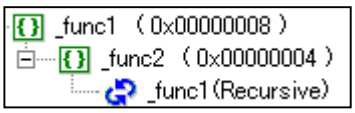
```
void func(int x)
{
    x++;
    if(x != OFF)
        func(x);

    if(x == MAX)
        return;
}
```



(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);
}
```



RTOS関数（ITRONなどのシンボル）

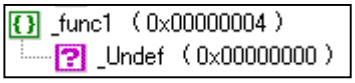
参照元シンボル不明関数

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。
（change_messageの詳細はH8S、H8/300**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**「4.2.7 **その他のオプション**」の「CHange_message」を参照してください。）

【例】

```
void func1(void)
{
    Undef();
}
```

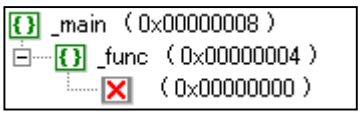


アドレス参照未解決関数

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])();
}
```



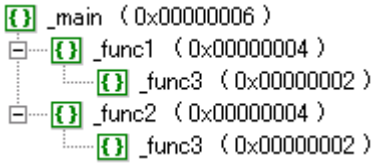
省略表示シンボル

本ツールでは、リンク階層をすべて表示するため、ユーザアプリケーションが大きい場合は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。

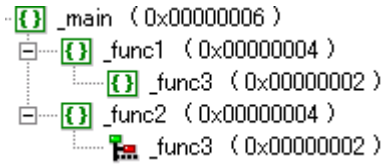
この表示は、[View->Show All Symbols / Show Simple Symbols]で切り替えることができます。

【例】

Show All Symbols



Show Simple Symbols



■シンボル詳細ビュー

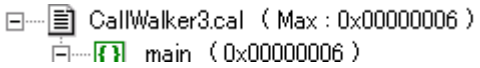
Symbol	Attri...	Address	Size	Stack size	Source
_INT_TXI1_...	I	0x000004...	0x00000002	0x00000004	intprg.obj
_abort		0x000008...	0x00000002	0x00000004	CallWalker2....
_sbrk		0x000008...	0x0000002c	0x00000008	sbrk.obj
_sub		0x000008...	0x00000002	0x00000004	CallWalker2....
_nop		0x000008...	0x00000002	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



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

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

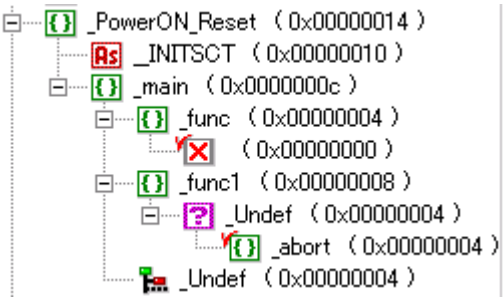
ファイルをオープン中、右側のシンボル詳細ビューでシンボル名を選択し、Edit メニューの Add...、Modify...、Delete... コマンドでシンボルの追加、変更、削除ができます。

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。

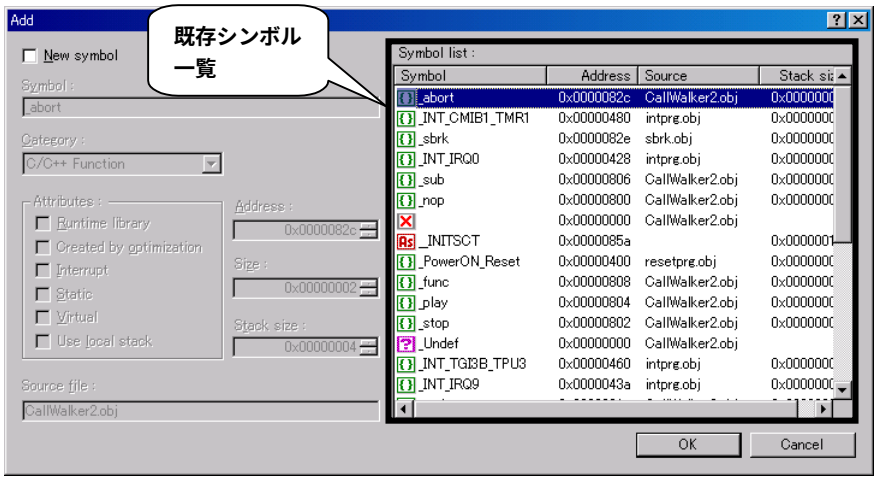


以降に、各コマンドの説明をします。

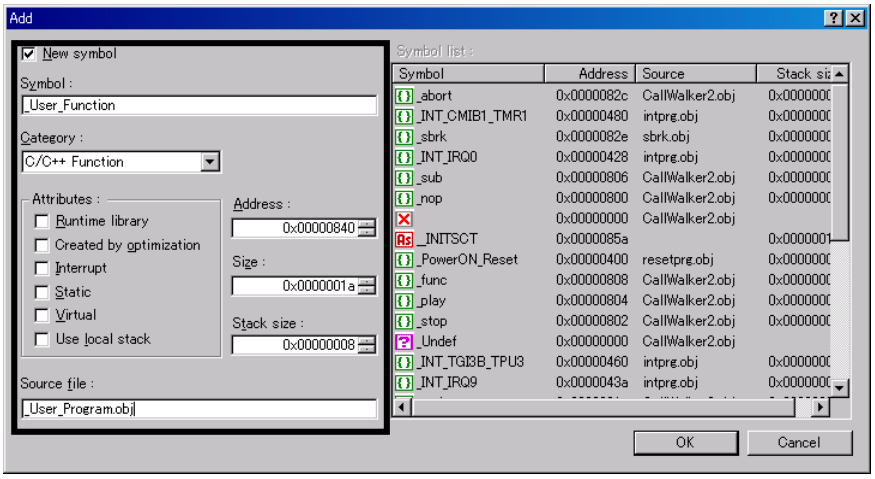
■Add...コマンド

（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。

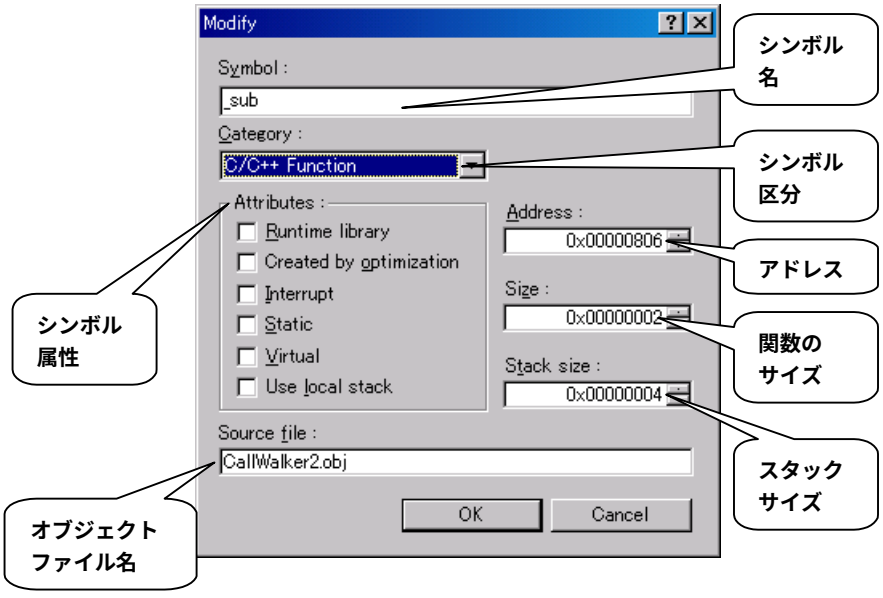


（2） 新規シンボルの追加
左側のチェックボックスをチェックすると、シンボルを新規作成できます。
その際に、シンボル名、シンボル区分、属性、アドレス、スタック使用量などを決定できます。



■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 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。

スタック値は次のように指定します。

- 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

Δ.STACKΔ<シンボル>=<スタック値>

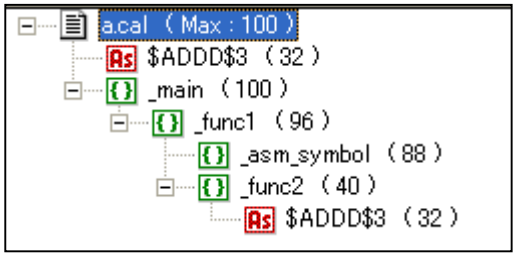
■アセンブラプログラム例

```
                .CPU        H8SXA:24
                .EXPORT      _asm_symbol
                .SECTION     P,CODE,ALIGN=2
_asm_symbol:
                .STACK       _asm_symbol=88
                :
                RTS
                .END
```

_asm_symbol関数のスタックサイズを88に設定

■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H8S,H8/300シリーズアセンブラVer6.01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）

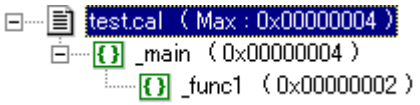
保存後、または編集中のスタック情報ファイルと、他のスタック情報ファイルをマージ（連結）することができます。これにより、編集したスタック情報は、再ビルド後のスタック情報に上書きされません。

■マージの例

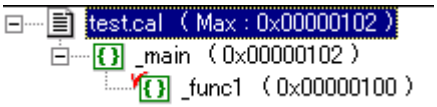
（1） test.cの内容

```
void main(void)
{
    func1();
}
```

（2） Call Walkerでスタック情報ファイルをオープン



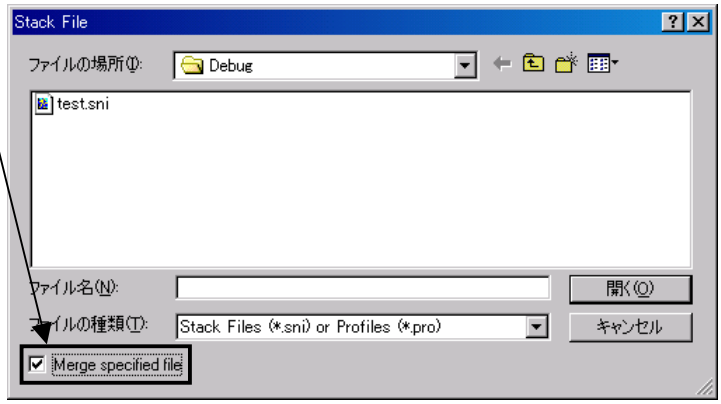
（3） 内容を変更（func1のスタック使用量を100に変更）



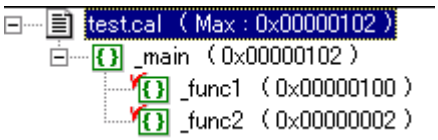
（4） test.cの内容を変更後にビルド（**func2の呼び出しを追加**）

```
void main(void)
{
    func1();
    func2();
}
```

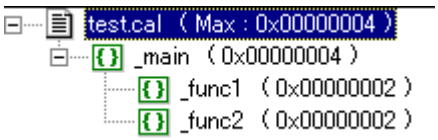
（5） Call Walkerでtest.calを開いたままtest.sniをオープンします。
このとき、ここ をチェックし、[開く]ボタンを押下します。



（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。

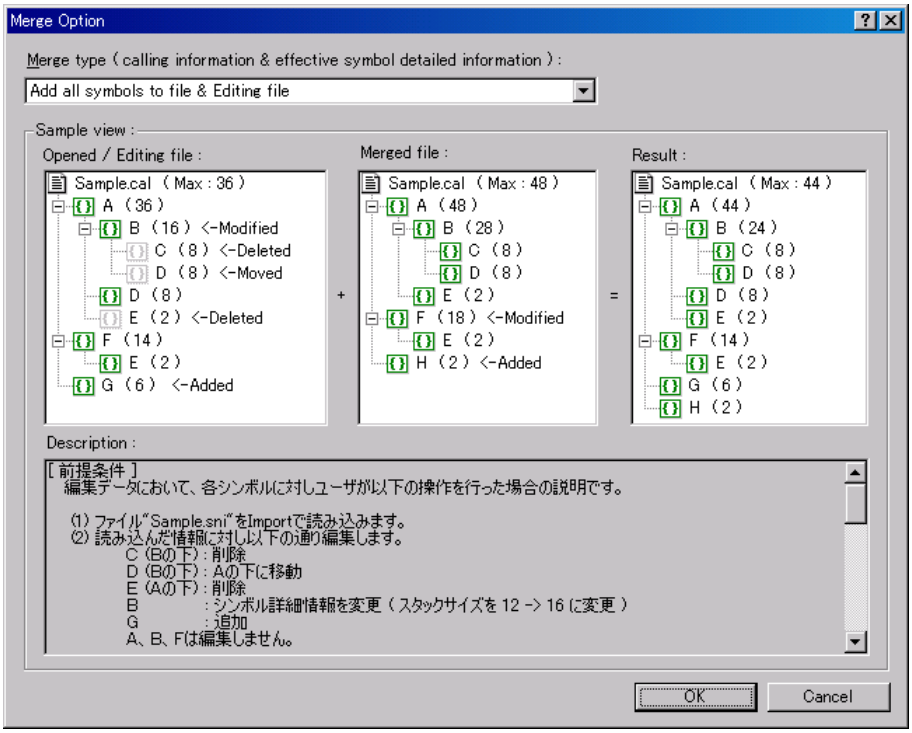


もし、（5）でチェックをしない場合は、以下のように（3）で変更した func1 のスタック使用量は変更前の値に戻ります。



■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、RTOS と表示することができます。

【指定方法】

Tools メニュー-> Realtime OS Option...

■リスト出力

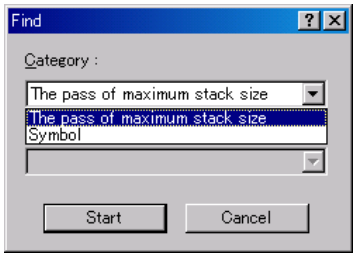
スタック情報をテキスト形式で、ファイルに出力することができます。

【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
4	**オプション**	**組み込み向け** C++**言語**	8.3.1
5	**活用法**	**実行時型情報**	8.3.2
6		**例外処理機能**	8.3.3
7		**プレリンカの起動抑止**	8.3.4
8	C++**記述のメリット・**	**コンストラクタ①**	8.4.1
9	**デメリット**	**コンストラクタ②**	8.4.2
10		**デフォルトパラメータ**	8.4.3
11		**インライン展開**	8.4.4
12		**クラスメンバ関数**	8.4.5
13		operator **演算子**	8.4.6
14		**関数のオーバロード**	8.4.7
15		**リファレンス型**	8.4.8
16		**スタティック関数**	8.4.9
17		**スタティックメンバ変数**	8.4.10
18		**匿名** union	8.4.11
19		**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 を実行している時に、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_INIT/_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

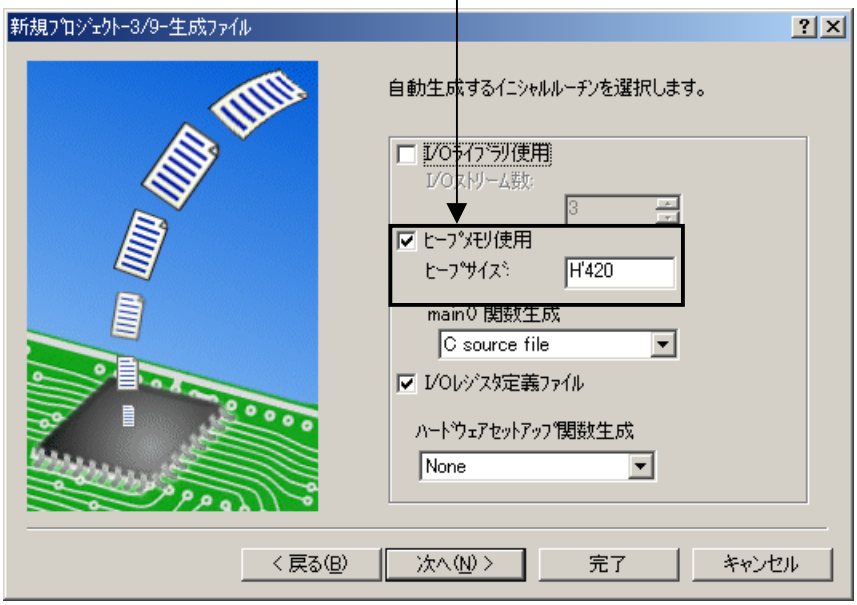
組み込みシステムにおいて new を使用する場合、実際のヒープメモリの動的な確保は malloc を使用することによって実現しています。

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「ヒープメモリ使用」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。



（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。

■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：コンパイラタブカテゴリ:[その他] EC++言語に基づいたチェック
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。

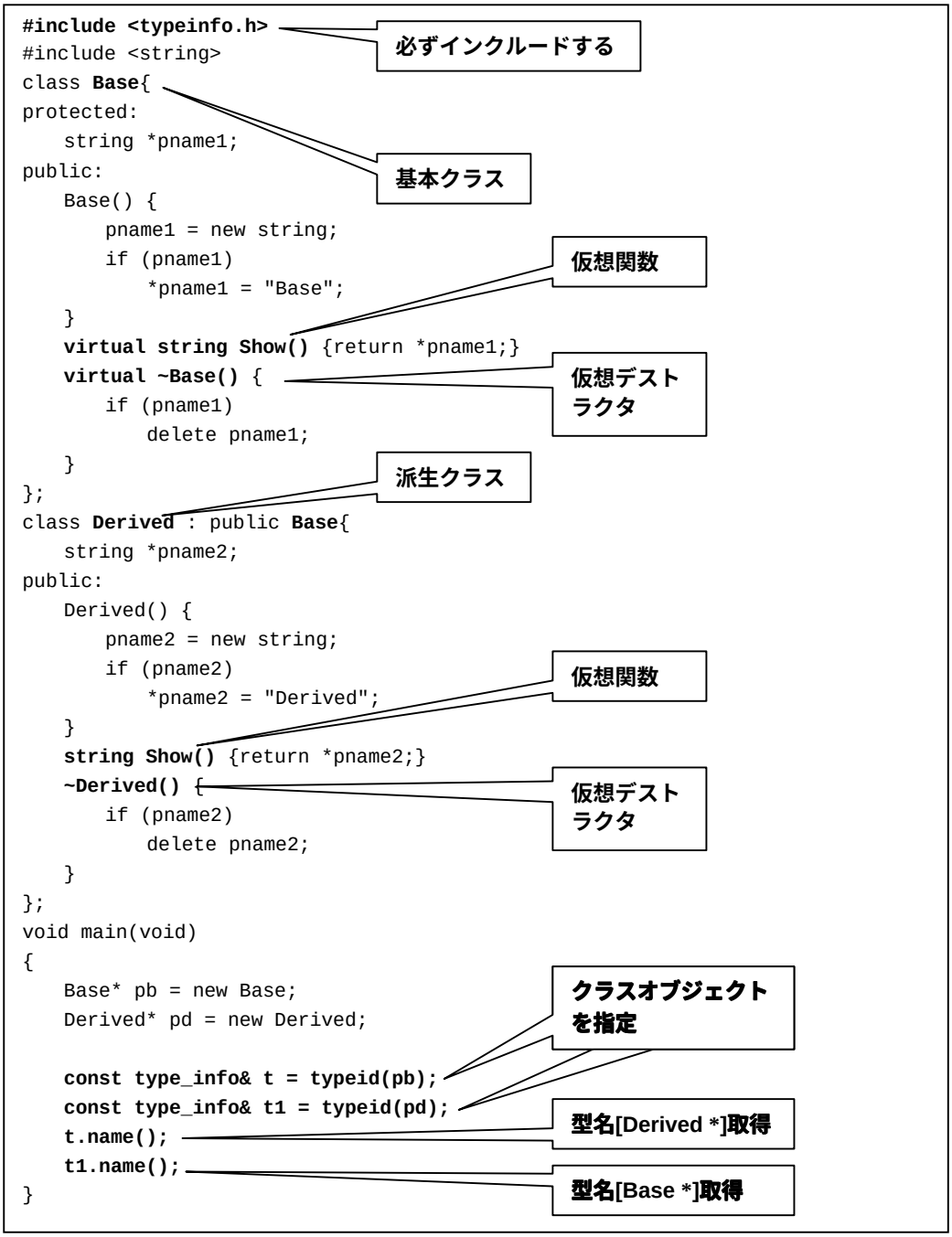
■指定方法

ダイアログメニュー：CPU タブ C++の dynamic_cast、typeid を有効にする
コマンドライン　　：*rtti=on | off*

ダイアログメニュー：最適化リンカタブカテゴリ:[インプット] プレリンカ制御を自動または使用
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

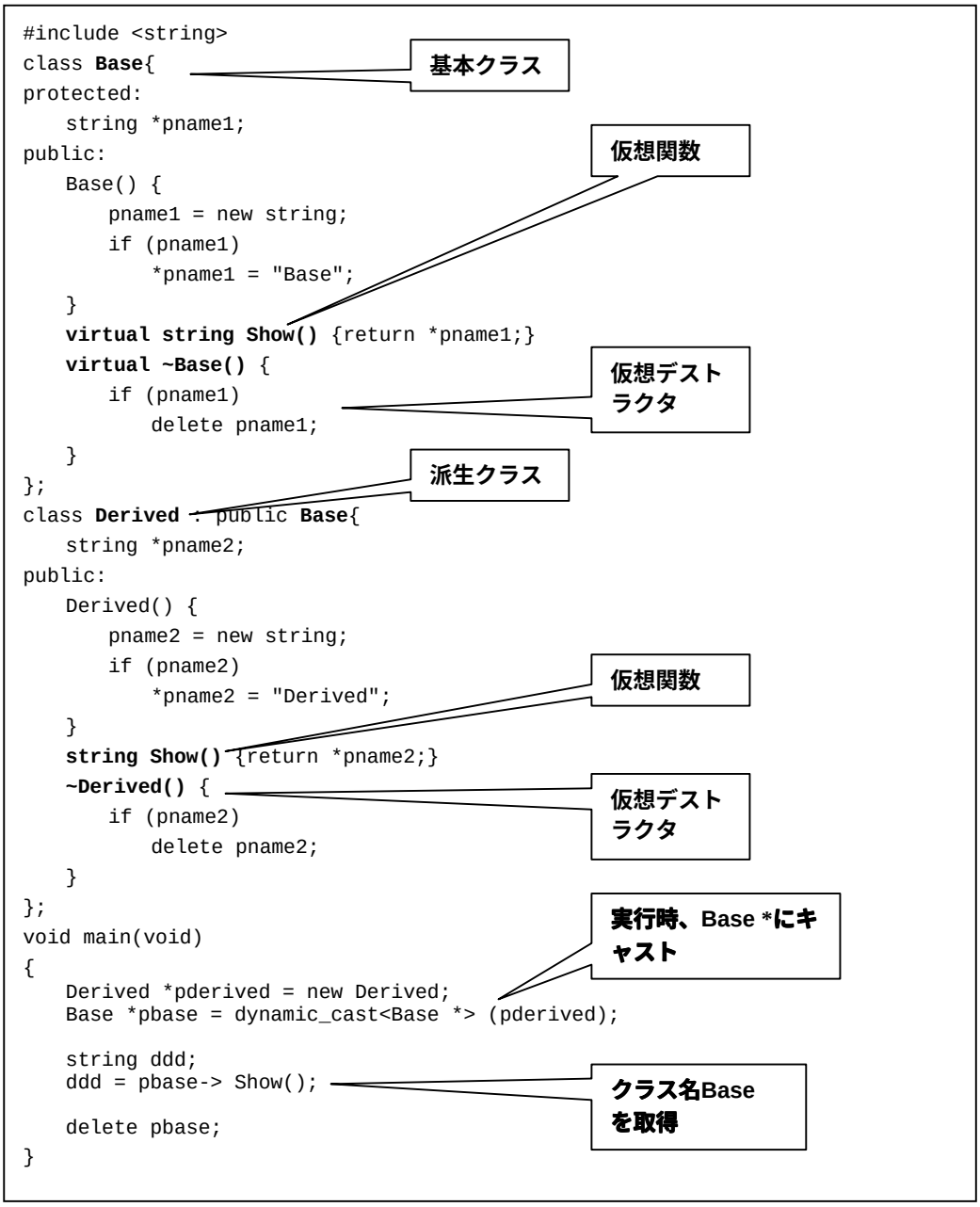
type_info クラスは、オブジェクトの実行時の型に関する識別操作のためのクラスです。
type_info クラスを使うと、プログラム実行時の型比較判定、クラスの型名取得などができるようになります。
type_info クラスを使うには、typeid 演算子で仮想関数を持つクラスオブジェクトを指定します。




```
#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();
}
```

■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;
}
```

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 C++の try、throw、catch を有効にする
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時、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：最適化リンカタブカテゴリ:[インプット] プレリンカ制御
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することができますが、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;
    b = a.getValue();
}

A::A(void)
{
    a = 1234;
}

A::~A(void)
{
}
```

（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

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することができますが、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。

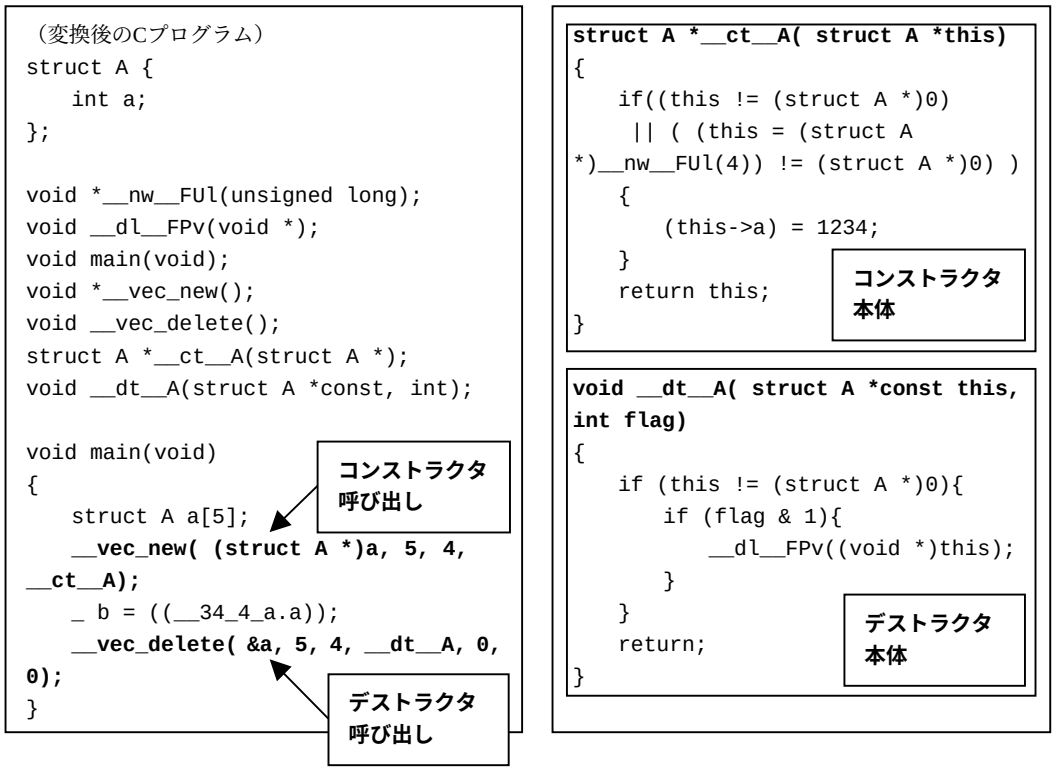
動的なメモリの割り当て／解放のために、new,delete を使用します。

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 H8S、H8/300 **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**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)
{
}
```



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の必要がありません。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。

（C++プログラム）

```
void main(void);
int sub(int, int = 0);

void main(void)
{
    int ret1;
    int ret2;
    ret1 = sub(1,2);
    ret2 = sub(3);
}

int sub(int a, int b /* =0 */ )
{
    return a + b;
}
```

関数宣言でパラメータ値に0を指定

第2パラメータ指定なし

（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）

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 プログラムでは、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;
    ret.z = z + vec.z;

    return ret;
}
```

ユーザの演算処理（加算）

（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;
}
```

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;
}
```

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。

このような時に、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;
public:
    static int getNum(void);
    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

スタティック関数

スタティックメンバ変数にアクセス

（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);
struct A *__ct__A(struct A *);
void __dt__A(struct A *const, int);
int num__1A = 0;
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;
}
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;
}
```

スタティック関数

スタティックメンバ変数

スタティックメンバ変数にアクセス

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 のメンバ変数 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 のメンバ変数 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;
}
```

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

（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;
}
```

コンパイラ内部では通常のグローバル変数扱い

クラスA型クラスオブジェクト作成

コンストラクタをコール

デストラクタをコール

staticメンバ変数をインクリメント

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

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。

（C++プログラム）

```
class A
{
private:
   int a;
public:
   virtual void foo(void);
};

class B : public A
{
private:
   int b;
public:
   virtual void foo(void);
};

void A::foo(void)
{
}

void B::foo(void)
{
}
```

仮想関数宣言

仮想関数宣言
*省略可

```
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*)&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 _a;
   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->b);
   // 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			**短絶対アドレッシングモード活用**	9.4.4
16			**レジスタ退避・回復の最適化**	9.4.5
17			**共通コードの統合**	9.4.6
18			**間接アドレッシングモード活用**	9.4.7
19			**分岐命令の最適化**	9.4.8
20			**アドレッシングモードの短縮**	9.4.9
21		**最適化部分抑止**		9.4.10
22		**最適化結果の確認**		9.4.11

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

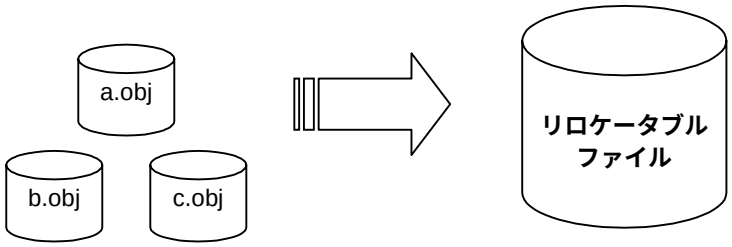
ファイルの種類	コマンドラインでの指定
オブジェクトファイル	input
リロケータブルファイル	input
ライブラリファイル	library
バイナリファイル	binary

(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。

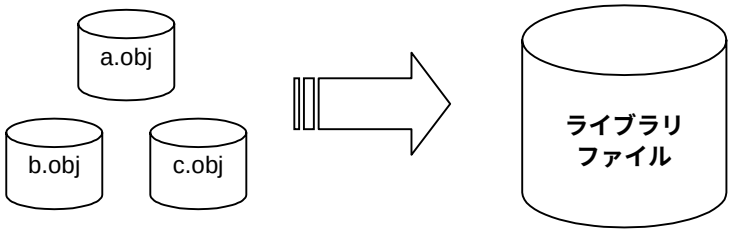
(2) リロケータブルファイル

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。



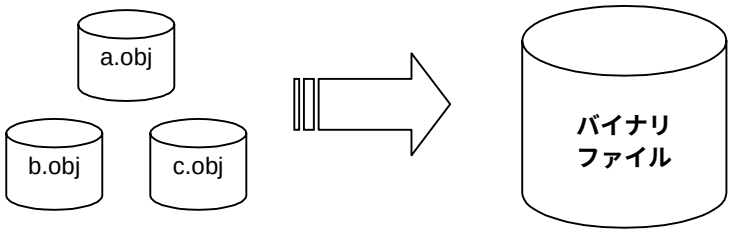
(3) ライブラリファイル

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。



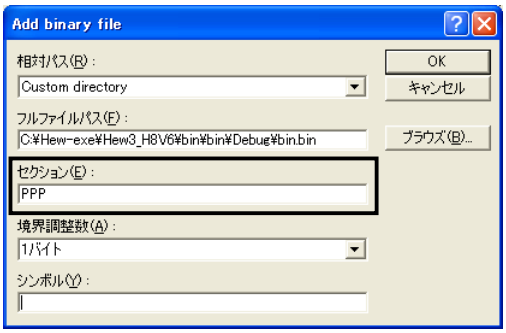
(4) バイナリファイル

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。



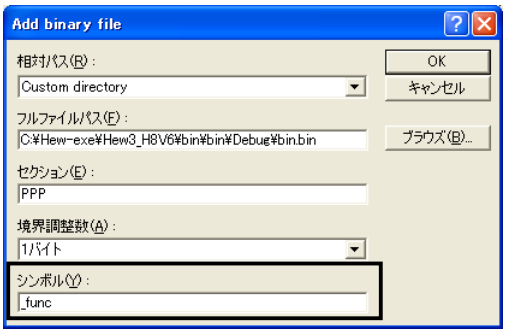
【指定方法 1】
必ずセクション名を指定する必要があります。

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)***



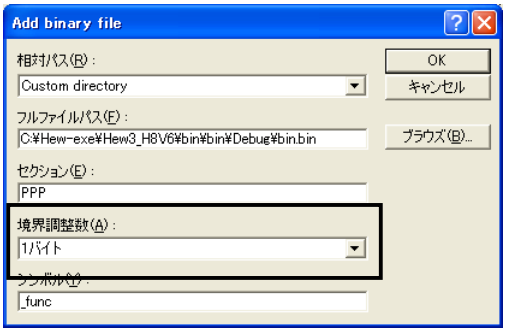
【指定方法 2】
バイナリファイルの先頭にシンボルを割り付けることができます。
この場合、セクション名と共にシンボル名を指定します。C/C++から参照する場合は、シンボルの先頭に‘ _ ’を付加します。

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時は、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できます。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 H8S、H8/300 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 H8S、H8/300 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_H8V6¥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_H8V6¥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ほかに、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時などに便利です。

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 ファイルの時、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

次頁からの例における＜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＞

```
S00E000062696E2020202020206D6F74C8
S1070000000040 0F4
S1070014000041AC6
S107001C000041CBC
S1070020000041EB6
S1070024000042080
S1070028000042 2AA
S107002C000042 4A4
S1070040000042 68E
S1070044000042888
S1070048000042A82
S107004C000042C7C
S1070050000042E76
S1070054000043070
```

＜途中省略＞

```
S11308901F9045EC7A00000008C87A01000008D2D7
S11308A040180100 6D0401006D0501006D0640064D
S11308B06C4A68EA0B061FD445F81F9045E40120F4
S10908C06D766D725470A8
S10F08C8000008DA000008DE00FFE42A4D
S10B08D200FFE0000FFE42A2E
S10708DA00FFE00A2D
S10F08DE7900000A6BA00000200C54708C
S9030400F8
```

＜空きエリア出力指定あり[H'FF を設定]＞

```
S00E000062696E2020202020206D6F74C8
S1070000000040 0F4
S1130004FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF8
S1070014000041AC6
S1070018FFFFFFFE4
S107001C000041CBC
S1070020000041EB6
S1070024000042080
S1070028000042 2AA
S107002C000042 4A4
S1130030FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFCC
S1070040000042 68E
S1070044000042888
S1070048000042A82
```

＜途中省略＞

```
S113FF8AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF73
S113FF9AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF63
S113FFAAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF53
S113FFBAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF43
S113FFCAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF33
S113FFDAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF23
S113FFEAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF13
S109FFFAFFFFFFFFFFFF03
S9030400F8
```

■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
```

データが**存在する**範囲

＜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
```

データが**存在する**範囲

データが**存在しない**範囲

■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
:040014000000041ACA
:04001C000000041CC0
:040020000000041EBA
:0400240000000420B4
:0400280000000422AE
:04002C0000000424A8
:040040000000042692
:04004400000004288C
:040048000000042A86
: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
:040014000000041ACA
:04001800FFFFFFFFE8
:04001C000000041CC0
:040020000000041EBA
:0400240000000420B4
:0400280000000422AE
:04002C0000000424A8
:10003000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD0
:040040000000042692
```

＜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

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。

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。

■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 H8S、H8/300 **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**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時は、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;
```

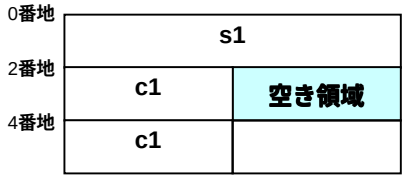
【data_stuff 指定なしのデータ配置】

data_stuff 指定がない場合、H8 は CPU の仕様上、境界調整数が 2 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。

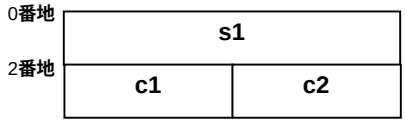


【data_stuff 指定ありのデータ配置】

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／アセンブ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

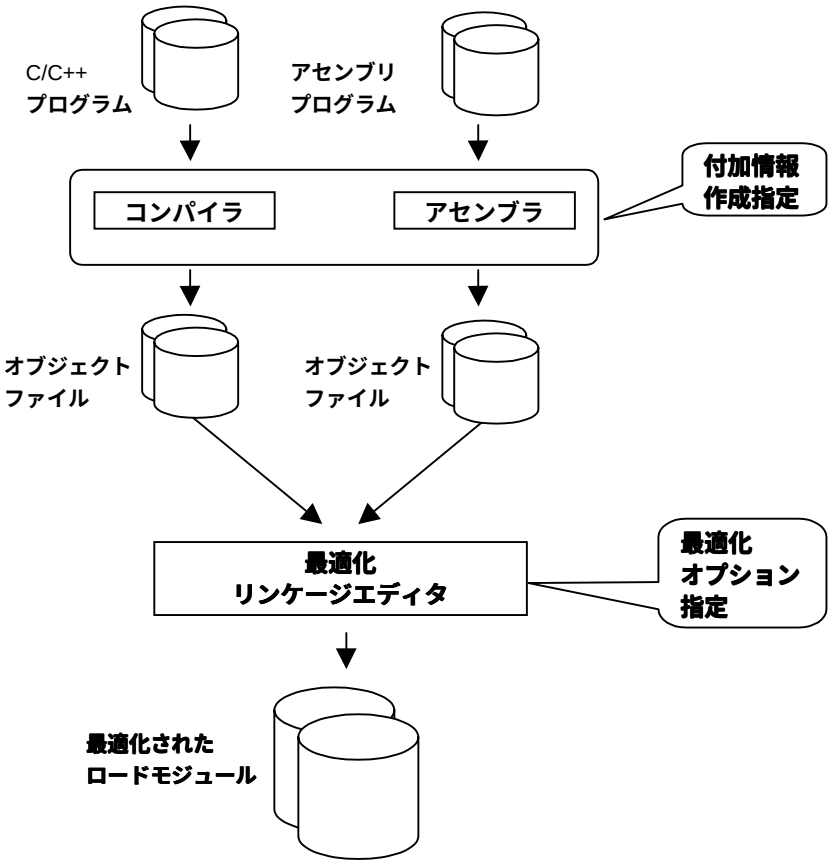
ダイアログメニュー：**最適化リンカタブカテゴリ:[最適化]**
コマンドライン　　：*optimize*=<サブオプション>
　　：<サブオプション>は「9.4.2」～「9.4.9」の各最適化項目参照

また、リンク時の最適化指定をしても、コンパイル／アセンブ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**
ダイアログメニュー：**アセンブラタブカテゴリ:[オブジェクト] 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



9.4.2 定数／文字列の統合

サイズ効率	○	処理速度	-

■説明

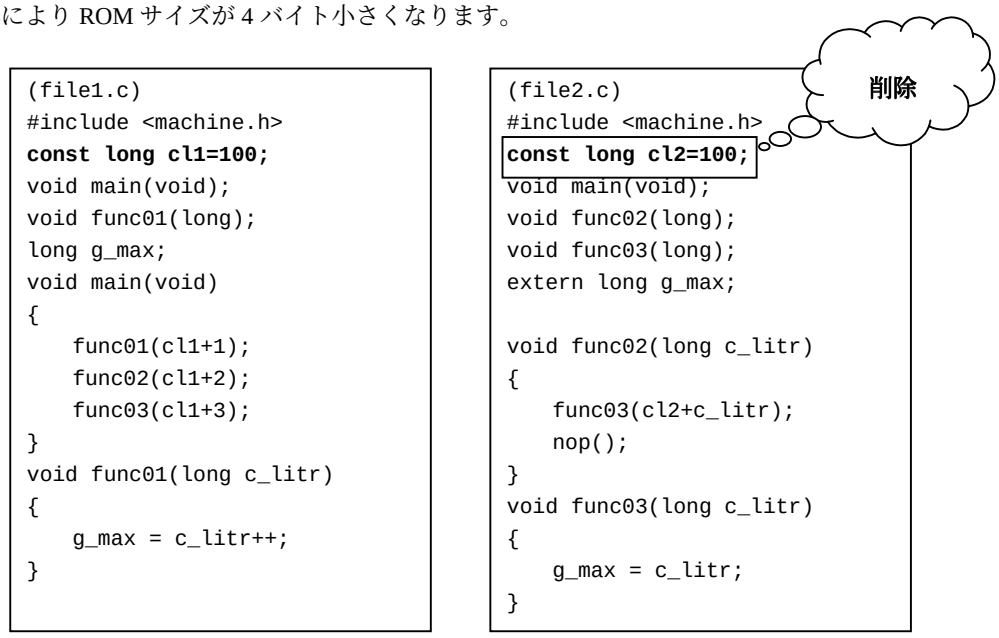
const 属性を持つ定数／文字列に対し、同一値定数および同一文字列の統合をモジュール間にわたって実施します。
本サブオプションはコンストセクションを削除するので、サイズ効率は上がりますが、処理速度に変化はありません。

■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。



■備考

本オプションは C/C++コンパイラからの出力オブジェクトにのみ有効です。アセンブラから出力されたオブジェクトは最適化しません。

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*

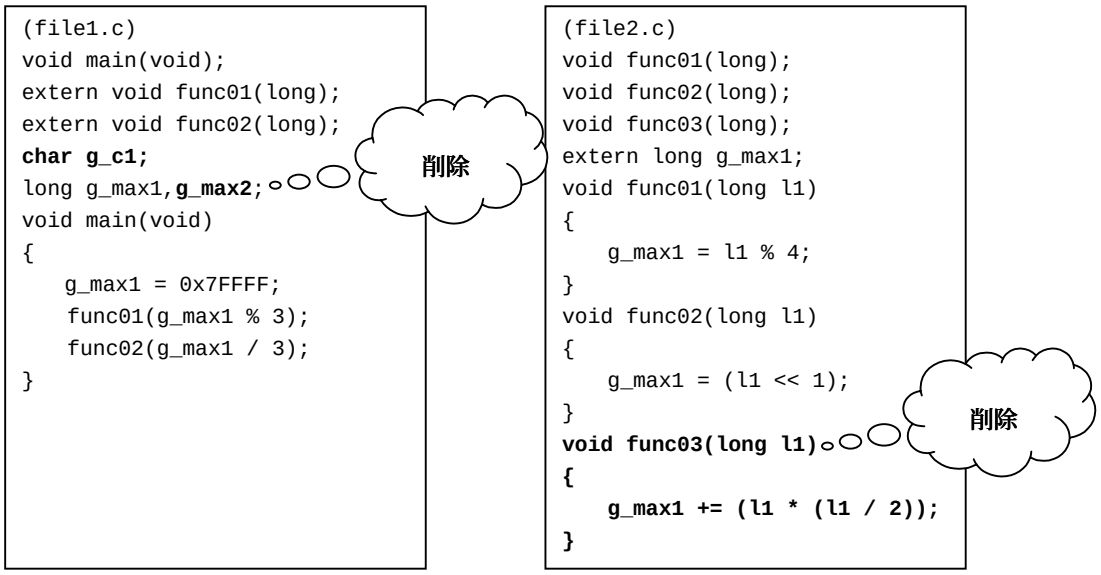
■エントリ関数指定方法

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>*

【注】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。

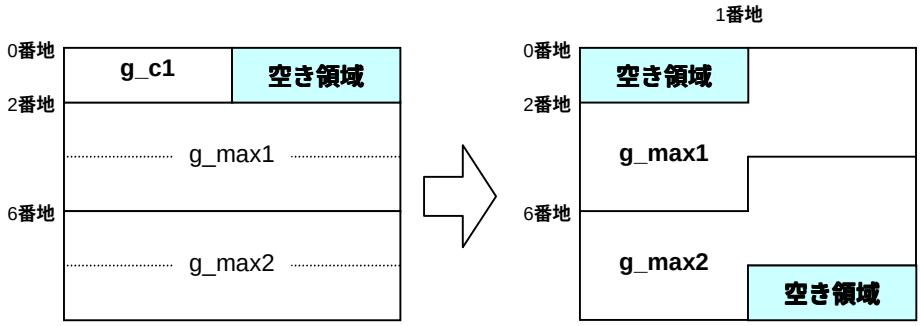


char 型の変数 g_c1 も一度も参照されませんが、削除されません。

これは以下のように、H8 の境界調整数が 2 のため、削除してしまうと後続の変数のアドレスが 2 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。（H8SX を除く。）

【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 1 番地から g_max1（4 バイト変数）をアクセスしてしまいます。

■備考

本オプションは C/C++コンパイラからの出力オブジェクトにのみ有効です。アセンブラから出力されたオブジェクトは最適化しません。

9.4.4 短絶対アドレッシングモード活用

サイズ効率	○	処理速度	○

■説明

8 ビット／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空きがあれば、アクセス回数の多い変数を割り当て、当該変数のアクセスコード最適化を行います。

最適化リンケージエディタは、これらの変数を、自動的に作成するセクションに自動的に割り付けます。

コンパイラにも同様の機能がありますが、最適化リンケージエディタの場合は、自動でセクションの割り付けまで行います。

ただし、CPU 種別によって短絶対アドレス領域が違うため、cpu オプションによって ROM や RAM のアドレスを指定する必要があります。(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: 短絶対アドレッシングモード活用**
コマンドライン　：*optimize=variable_access*

■cpu オプション指定方法

ダイアログメニュー：**最適化リンカタブカテゴリ:[ベリファイ]:**
コマンドライン　：*cpu=<メモリ種別>=<アドレス範囲>　または*
：*cpu= <cpu 情報ファイル名>*
：*<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>*

■短絶対アドレッシングモード活用例

char 型の変数 g_c1 が ABS8B_OPT1 セクション、short 型の変数 g_s1,g_s2 が ABS16B_OPT1 セクションに割り付けられます。

したがって、これらの変数へのアクセスコードの、サイズ効率／処理速度ともに向上します。

```
#include <machine.h>
void init(void);
void main(void);
short func01(short);
char g_c1;
short g_s1,g_s2;

void init(void)
{
    main();
    sleep();
}
```

```
void main(void)
{
    g_s1 = 7;
    g_s2 = 8;
    g_c1 = 10;
    g_s1 =
func01(g_s1+g_s2+g_c1);
    nop();
}
short func01(short p_s1)
{
    short wk = ++p_s1;
    return wk;
}
```

■最適化されたアクセスコード

本例の場合、ROM サイズが 40→30 バイト、実行速度が 41→36 サイクルになります。

（H8S/2600 アドバンストモードの場合）

（オプション指定なし）

```
_main:
    MOV.W      #7,R0
    MOV.W      R0,@_g_s1:32
    MOV.B      #8,R0L
    MOV.W      R0,@_g_s2:32
    MOV.B      #10,R0L
    MOV.B      R0L,@_g_c1:32
    MOV.B      #25,R0L
    BSR        _func01:8
    MOV.W      R0,@_g_s1:32
    NOP
    RTS
```

（オプション指定あり）

```
_main:
    MOV.W      #7,R0
    MOV.W      R0,@_g_s1:16
    MOV.B      #8,R0L
    MOV.W      R0,@_g_s2:16
    MOV.B      #10,R0L
    MOV.B      R0L,@_g_c1:8
    MOV.B      #25,R0L
    BSR        _func01:8
    MOV.W      R0,@_g_s1:16
    NOP
    RTS
```

■備考

（1） 短絶対アドレス領域の詳細は**「**5.4.11 8**ビット絶対アドレス領域の活用」**、**「**5.4.12 16**ビット絶対アドレス領域の活用」**を参照してください。

（2） 本オプションはC/C++コンパイラおよびアセンブラから出力されたオブジェクトに有効です。

9.4.5 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

main 関数から func01 関数を呼び出し、func01 関数から func02 関数を呼び出しています。

```
(file1.c)
void main();
extern void func01(long *,long *,long *,long *);
long g_l1,g_l2,g_l3,g_l4;
void main()
{
    g_l1 = 1;
    g_l2 = 2;
    g_l3 = 3;
    g_l4 = 4;
    func01(&g_l1,&g_l2,&g_l3,&g_l4);
}
```

```
(file2.c)
extern long g_l1,g_l2,g_l3,g_l4;
extern void func02(long *,long *,long *,long *);
void func01(long *l_p1,long *l_p2,long *l_p3,long *l_p4)
{
   g_l2 = 2;
   g_l2 += *l_p1;
   func02(&g_l1,&g_l2,&g_l3,&g_l4);
}
```

```
(file3.c)
extern long g_l1,g_l2,g_l3,g_l4;
void func02(long *l_p1,long *l_p2,long *l_p3,long *l_p4)
{
   g_l1++;
   *l_p1 = g_l1;
}
```

■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 202→198 バイト、実行速度が 172→166 サイクルになります。

（H8S/2600 アドバンストモードの場合）

（最適化前）
ER2-ER3を退避・回復（**2本**）

```
_main:
    STM.L     (ER2-ER3),@-SP
    SUB.L     ER0,ER0
    MOV.B     #1,R0L
    MOV.L     ER0,@_g_l1:32
    SUB.L     ER1,ER1
        : (途中省略)
    MOV.L     #_g_l3,ER2
    PUSH.L    ER2
    MOV.L     #_g_l2,ER1
    MOV.L     #_g_l1,ER0
    JSR       @_func01:24
    ADDS.L    #4,SP
    ADDS.L    #4,SP
    LDM.L     @SP+,(ER2-ER3)
    RTS
```

（最適化後）
ER2-ER4を退避・回復（**3本**）

```
_main:
    STM     (ER2-ER3),@-SP
    PUSH.L  ER4
    SUB.L   ER0,ER0
    MOV.B   #1:8,R0L
    MOV.L   ER0,@_g_l1:32
    SUB.L   ER1,ER1
        : (途中省略)
    MOV.L   #h'00f00004:32,ER1
    MOV.L   #h'00f00000:32,ER0
    BSR     _func01:8
    ADDS    #4,SP
    ADDS    #4,SP
    POP.L   ER4
    LDM     @SP+,(ER2-ER3)
    RTS
```

ER2-ER3を退避・回復（**2本**）

```
_func01:
    STM.L     (ER2-ER3),@-SP
    MOV.L     #_g_l2,ER1
        : (途中省略)
    MOV.L     #_g_l1,ER0
    JSR       @_func02:24
    ADDS.L    #4,SP
    ADDS.L    #4,SP
    LDM.L     @SP+,(ER2-ER3)
    RTS
```

ER2を退避・回復（**1本**）

```
_func01:
    PUSH.L  ER2
    MOV.L   #_g_l2,ER1
        : (途中省略)
    MOV.L   #_g_l1,ER0
    BSR     _func02:8
    ADDS    #4,SP
    ADDS    #4,SP
    POP.L   ER2
    RTS
```

ER2を退避・回復（**1本**）

```
_func02:
    PUSH.L      ER2
    MOV.L       #_g_l1,ER1
    MOV.L       @ER1,ER2
    INC.L       #1,ER2
    MOV.L       ER2,@ER1
    MOV.L       ER2,@ER0
    POP.L       ER2
    RTS
```

退避・回復なし（**0本**）

```
_func02:
    MOV.L   #_g_l1,ER1
    MOV.L   @ER1,ER2
    INC.L   #1,ER2
    MOV.L   ER2,@ER1
    MOV.L   ER2,@ER0
    RTS
```

■備考

本オプションは C/C++コンパイラからの出力オブジェクトにのみ有効です。アセンブラから出力されたオブジェクトは最適化しません。

9.4.6 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ーヘッドが増えて、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 関数に同様の式の並びがあります。

```
(file1.c)
void main(void);
void func00(void);
long g_l1,g_l2,g_l3,g_l4,g_l5;
void main(void)
{
    func00();
    func01();
}
void func00(void)
{
    g_l1 = 1;
    g_l2 = 3;
    g_l3 = 5;
    g_l4 = 7;
    g_l5 = 9;
}
```

```
(file2.c)
void func01(void);
extern long g_l1,g_l2,g_l3,g_l4,g_l5;
void func01(void)
{
    g_l1 = 1;
    g_l2 = 3;
    g_l3 = 5;
    g_l4 = 7;
    g_l5 = 9;
}
```

■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 114→66 バイト、実行速度が 91→108 サイクルになります。

（H8S/2600 アドバンストモードの場合）

（最適化前）

```
(file1.c)
_main:
    BSR   _func00:8
    JMP   @_func01:24
_func00:
    SUB.L   ER0,ER0
    MOV.B   #1,R0L
    MOV.L   ER0,@_g_l1:32
    MOV.B   #3,R0L
    MOV.L   ER0,@_g_l2:32
    MOV.B   #5,R0L
    MOV.L   ER0,@_g_l3:32
    MOV.B   #7,R0L
    MOV.L   ER0,@_g_l4:32
    MOV.B   #9,R0L
    MOV.L   ER0,@_g_l5:32
    RTS
```

共通なコード群

```
(file2.c)
_func01:
    SUB.L   ER0,ER0
    MOV.B   #1,R0L
    MOV.L   ER0,@_g_l1:32
    MOV.B   #3,R0L
    MOV.L   ER0,@_g_l2:32
    MOV.B   #5,R0L
    MOV.L   ER0,@_g_l3:32
    MOV.B   #7,R0L
    MOV.L   ER0,@_g_l4:32
    MOV.B   #9,R0L
    MOV.L   ER0,@_g_l5:32
    RTS
```

（最適化後）

```
(file1.c)
_main:
    BSR     _func00:8
    BRA     _func01:8
_func00:
    BSR     _com_opt1:8
    RTS
_com_opt1:
    SUB.L   ER0,ER0
    MOV.B   #1:8,R0L
    MOV.L   ER0,@_g_l1:32
    MOV.B   #3:8,R0L
    MOV.L   ER0,@_g_l2:32
    MOV.B   #5:8,R0L
    MOV.L   ER0,@_g_l3:32
    MOV.B   #7:8,R0L
    MOV.L   ER0,@_g_l4:32
    MOV.B   #9:8,R0L
    MOV.L   ER0,@_g_l5:32
    RTS
```

新規作成関数

```
(file2.c)
_func01:
    BSR     _com_opt1:8
    RTS
```

■備考

本オプションは C/C++コンパイラからの出力オブジェクトにのみ有効です。アセンブラから出力されたオブジェクトは最適化しません。

9.4.7 間接アドレッシングモード活用

サイズ効率	○	処理速度	-

■説明

メモリ間接アクセス空間に空きがあれば、アクセス回数の多い関数のアドレスを INDIRECT_OPT セクションに配置し、自動的にメモリ間接アクセス空間に割り付けます。

メモリ間接形式でアクセスすることにより、サイズ効率が上がります。

ただし、この領域はベクタテーブルで使用する場所なので、注意が必要です。

また、cpu オプションによって ROM のアドレスを指定する必要があります。

(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]: 間接アドレッシングモード活用**

コマンドライン　　：*optimize=function_call*

■cpu オプション指定方法

ダイアログメニュー：**最適化リンカタブカテゴリ:[ベリファイ]:**

コマンドライン　　：*cpu=<メモリ種別>=<アドレス範囲>　または*

：*cpu= <cpu 情報ファイル名>*

：*<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>*

■間接アドレッシングモード活用 ソース例

main 関数から func01, func02,func03 関数を呼び出していて、func01 の呼び出し回数が多い状態の例です。

```
(file1.c)
extern long func01(void);
extern long func02(void);
extern long func03(void);
void main(void);
long g_l1,g_l2,g_l3,g_l4,g_l5;
void main(void)
{
    g_l1 = 100;
    g_l1 = func01();
    g_l2 = 1000;
    g_l2 = func02();
    g_l3 = func03();
    g_l1 = func01();
    g_l1 = func01();
}
```

```
(file2.c)
long func01(void);
long func02(void);
long func03(void);
extern long g_l1,g_l2,g_l3,g_l4,g_l5;
long func01(void)
{
    return g_l1 *= 100;
}
long func02(void)
{
    return g_l2 /= 100;
}
long func03(void)
{
    return g_l2 %= 4;
}
```

■間接アドレッシングモード活用 コード例

間接アドレッシングモード活用前後のコード変化は下記のようになります。
呼び出し回数の多い func01 関数をメモリ間接形式で呼び出しています。
本例の場合、ROM サイズが 288→274 バイト、実行速度が 491→485 サイクルになります。

（H8S/2600 アドバンストモードの場合）

（最適化前）

```
_main
    PUSH.L    ER6
    MOV.L     #_g_l1,ER6
    SUB.L     ER0,ER0
    MOV.B     #100,R0L
    MOV.L     ER0,@ER6
    JSR       @_func01:24
    MOV.L     ER0,@ER6
    MOV.L     #1000,ER0
    MOV.L     ER0,@_g_l2:32
    JSR       @_func02:24
    MOV.L     ER0,@_g_l2:32
    JSR       @_func03:24
    MOV.L     ER0,@_g_l3:32
    JSR       @_func01:24
    MOV.L     ER0,@ER6
    JSR       @_func01:24
    MOV.L     ER0,@ER6
    POP.L     ER6
    RTS
```

（最適化後）

```
_main:
    PUSH.L  ER6
    MOV.L   #_g_l1,ER6
    SUB.L   ER0,ER0
    MOV.B   #100,R0L
    MOV.L   ER0,@ER6
    JSR     @@_$ind_opt1:8
    MOV.L   ER0,@ER6
    MOV.L   #1000,ER0
    MOV.L   ER0,@_g_l2:32
    BSR     _func02:8
    MOV.L   ER0,@_g_l2:32
    BSR     _func03:8
    MOV.L   ER0,@_g_l3:32
    JSR     @@_$ind_opt1:8
    MOV.L   ER0,@ER6
    JSR     @@_$ind_opt1:8
    MOV.L   ER0,@ER6
    POP.L   ER6
    RTS
```

メモリ間接呼び出し

```
_func01:
    MOV.L     @_g_l1:32,ER0
    SUB.L     ER1,ER1
    MOV.B     #100,R1L
    JSR       @$MULL$3:24
    MOV.L     ER0,@_g_l1:32
    RTS
_func02:
    MOV.L     @_g_l2:32,ER0
    SUB.L     ER1,ER1
    MOV.B     #100,R1L
    JSR       @$DIVL$3:24
    MOV.L     ER0,@_g_l2:32
    RTS
_func03:
    MOV.L     @_g_l2:32,ER0
    SUB.L     ER1,ER1
    MOV.B     #4,R1L
    JSR       @$DIVL$3:24
    MOV.L     ER1,@_g_l2:32
    MOV.L     ER1,ER0
    RTS
```

```
_func01:
    MOV.L   @_g_l1:32,ER0
    SUB.L   ER1,ER1
    MOV.B   #100:8,R1L
    BSR     $MULL$3:8
    MOV.L   ER0,@_g_l1:32
    RTS
_func02:
    MOV.L   @_g_l2:32,ER0
    SUB.L   ER1,ER1
    MOV.B   #100,R1L
    BSR     $DIVL$3:8
    MOV.L   ER0,@_g_l2:32
    RTS
_func03:
    MOV.L   @_g_l2:32,ER0
    SUB.L   ER1,ER1
    MOV.B   #4,R1L
    BSR     $DIVL$3:8
    MOV.L   ER1,@_g_l2:32
    MOV.L   ER1,ER0
    RTS
```

■備考

（1） メモリ間接アクセス空間の詳細は「5.4.13 **メモリ間接形式の活用**」、「5.4.14 **拡張メモリ間接形式の活用**」を参照してください。

（2） 本オプションはC/C++コンパイラおよびアセンブラから出力されたオブジェクトに有効です。

9.4.8 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。

その時、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：**−126～128 **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 関数から別ファイルの func01 関数を呼び出す例です。

```
(file1.c)
#include <machine.h>
extern 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);
}
```

```
(file2.c)
long func01(long,long);
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。
別ファイルの関数 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 52→50 バイト、実行速度が 46→45 サイクルになります。

（H8S/2600 アドバンストモードの場合）

（最適化前）

```
_main:
    PUSH.L  ER6
    MOV.L   #_g_l1,ER6
    SUB.L   ER0,ER0
    MOV.B   #100,R0L
    MOV.L  ER0,@ER6
    MOV.B   #-56,R0L
    MOV.L   ER0,@_g_l2:32
    MOV.L   ER0,ER1
    MOV.L   @ER6,ER0
    JSR     @_func01:24
    MOV.L   ER0,@ER6
    POP.L    ER6
    RTS
```

```
_func01
    ADD.L   ER1,ER0
    RTS
```

（最適化後）

```
_main:
    PUSH.L  ER6
    MOV.L   #_g_l1,ER6
    SUB.L   ER0,ER0
    MOV.B   #100:8,R0L
    MOV.L   ER0,@ER6
    MOV.B   #56,R0L
    MOV.L   ER0,@_g_l2:32
    MOV.L   ER0,ER1
    MOV.L   @ER6,ER0
    BSR     _func01:8
    MOV.L   ER0,@ER6
    POP.l   ER6
    RTS
```

```
_func01:
    ADD.L   ER1,ER0
    RTS
```

■備考

本オプションは C/C++コンパイラおよびアセンブラから出力されたオブジェクトに有効です。

9.4.9 アドレッシングモードの短縮

サイズ効率	○	処理速度	-

■説明

ディスプレースメント／イミディエイトのコードサイズが短縮可能な場合、コードサイズがより小さくなる命令に変換します。

コンパイル時には、ファイル単位に処理を行うため、変数を参照している命令から変数定義までの距離が分かりません。

しかし、リンク時には命令および変数のアドレスが決定するのでこれらの距離がわかるようになり、本最適化を実施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:[最適化]: アドレッシングモードの短縮**
コマンドライン　　：*optimize=short_format*

■アドレッシングモードの短縮 ソース例

配列への代入と変数のアドレスを変数に格納する例です。

```
(file1.c)
short str1[4];
short str2[4];
void main(void);
void func01(short);
void func02(void);
char g_c1;
unsigned long g_l1;
void main(void)
{
   int i;
   for (i = 0;i < 4;i++)
   {
      str1[i] = i + 1;
      str2[i] = i * 2;
   }
   func01(i - 1);
   func02();
}
void func01(short s1)
{
   str1[s1] = s1;
   str2[s1] = s1+4;
}
void func02(void)
{
   g_l1 = (unsigned long)&g_c1;
}
```

■アドレッシングモードの短縮 コード例

アドレッシングモードの短縮前後のコード変化は下記のようになります。
それぞれ 32 ビットアクセスを 16／8 ビットアクセスに変更しています。
本例の場合、ROM サイズが 80→68 バイト、実行速度が 96→96 サイクルになります。

（H8SX アドバンストモードの場合）

（最適化前）

```
_main:
    SUB.W   R1,R1
L36:
    MOV.W   R1,R0
    INC.W    #1,R0
    MOV.W   R0,@(_str1:32,R1.W)
    MOV.W   R1,R0
    SHLL.W  R0
    MOV.W   R0,@(_str2:32,R1.W)
    INC.W   #1,R1
    CMP.    #4:3,R1
    BLT     L36:8
    DEC.W   #1,R1
    MOV.W   R1,R0
    BSR     _func01:8
    BSR     _func02:8
    RTS
_func01:
    MOV.W    R0,@(_str1:32,R0.W)
    MOV.W    R0,E0
    ADD.W    #4:3,E0
    MOV.W    E0,@(_str2:32,R0.W)
    RTS
_func02:
    MOV.L  #_g_c1:32,@_g_l1:32
    RTS
```

（最適化後）

```
_main:
    SUB.W   R1,R1
L36:
    MOV.W   R1,R0
    INC.W   #1,R0
    MOV.W   R0,@(h'0044:16,R1.W)
    MOV.W   R1,R0
    SHLL.W  R0
    MOV.W   R0,@(h'004c:16,R1.W)
    INC.W   #1,R1
    CMP.W   #4:3,R1
    BLT     L36
    DEC.W   #1,R1
    MOV.W   R1,R0
    BSR     _func01:8
    BSR     _func02:8
    RTS
_func01:
    MOV.W   R0,@(_str1:16,R0.W)
    MOV.W   R0,E0
    ADD.W   #4:3,E0
    MOV.W   E0,@(_str2:16,R0.W)
    RTS
_func02:
    MOV.L   #_g_c1:8,@_g_l1:32
    RTS
```

■備考

（1） 本オプションはCPUがH8SXN,H8SXM,H8SXA,H8SXXのときのみ有効です。
（2） 本オプションはC/C++コンパイラおよびアセンブラから出力されたオブジェクトに有効です。

9.4.10 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>*

■短絶対アドレッシングモード活用抑止

■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 短絶対アドレッシングモード活用抑止シンボル**
コマンドライン　：*variable_forbid=<シンボル名>*

■間接アドレッシングモード活用抑止

■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 間接アドレッシングモード活用抑止シンボル**
コマンドライン　：*function_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]*

9.4.11 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。

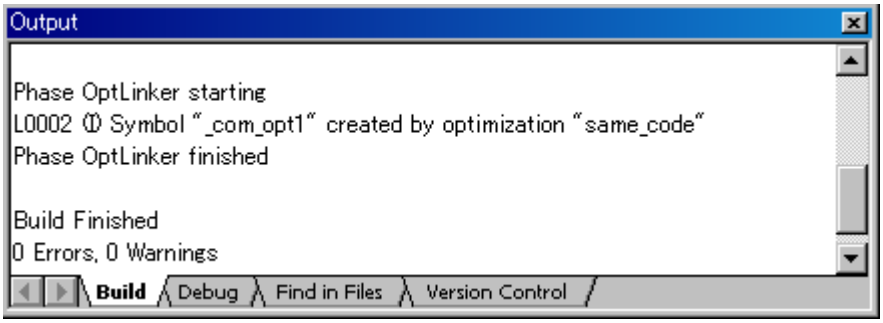
■メッセージでの確認

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。

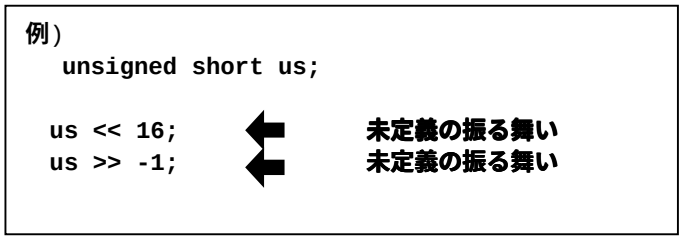


図 10.3 **ルール** 38

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。

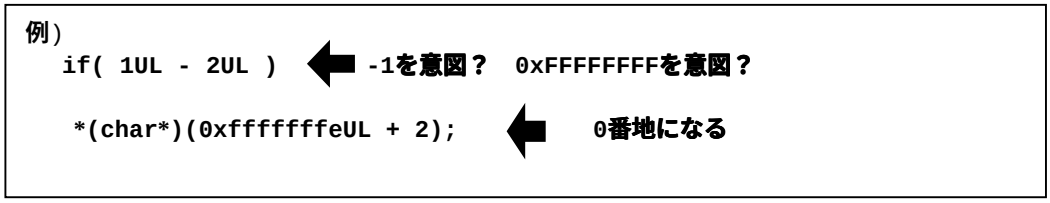


図 10.4 **ルール** 51

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、そのほう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。認める違反ごとに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。

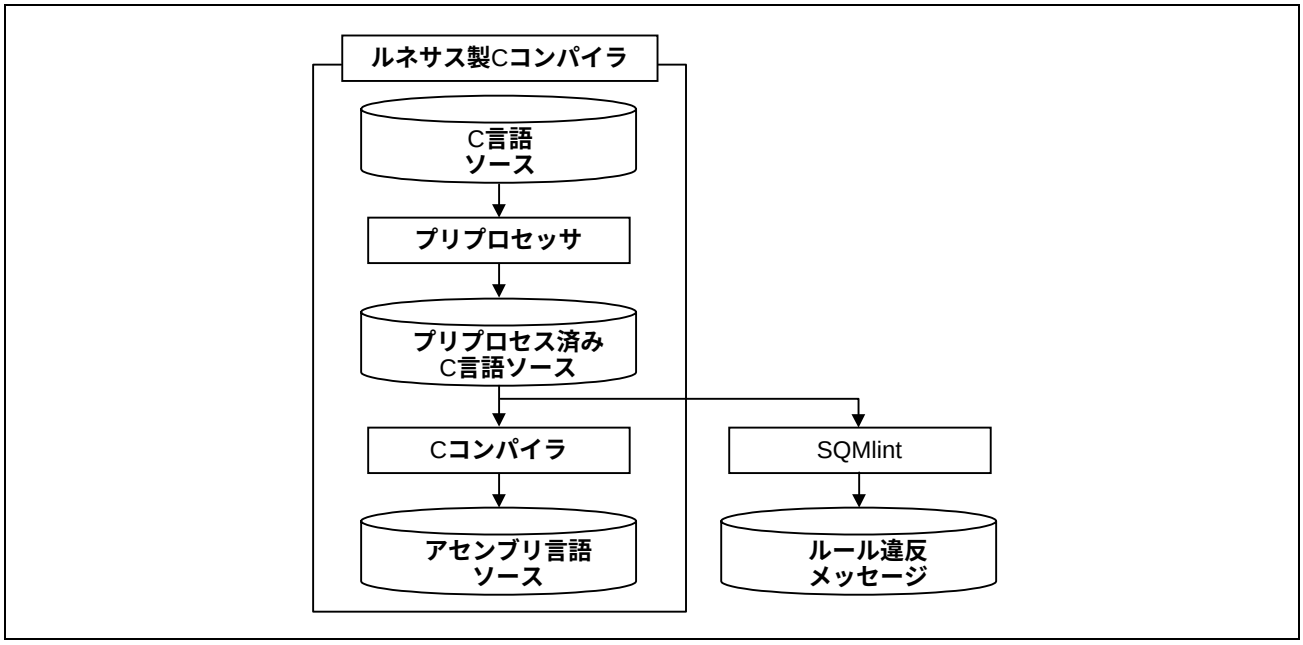


図 10.5 SQMlint **の位置付け**

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。

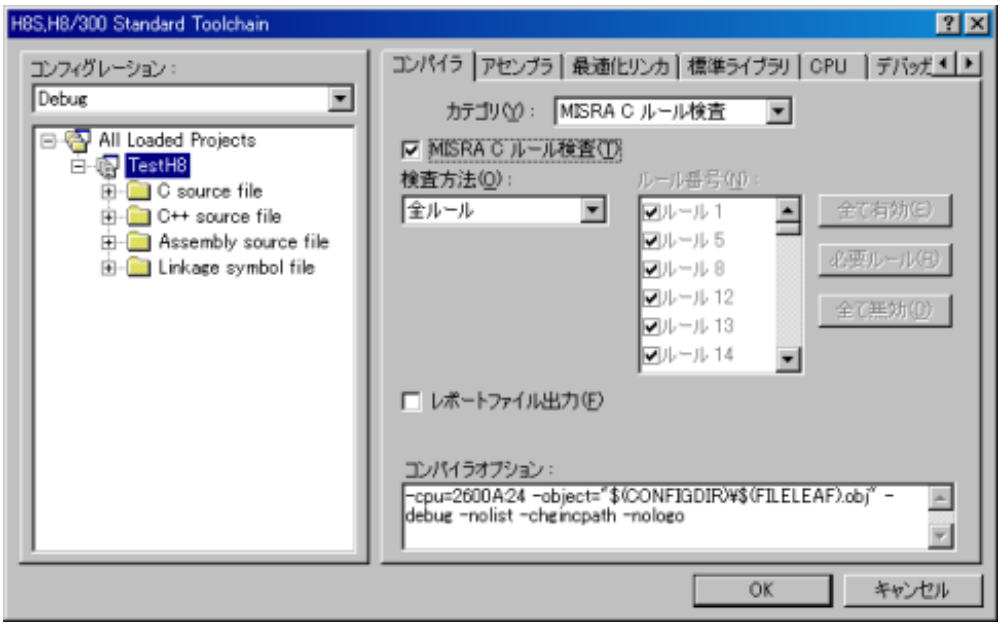


図 10.6 HEW **のオプション選択画面**

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。

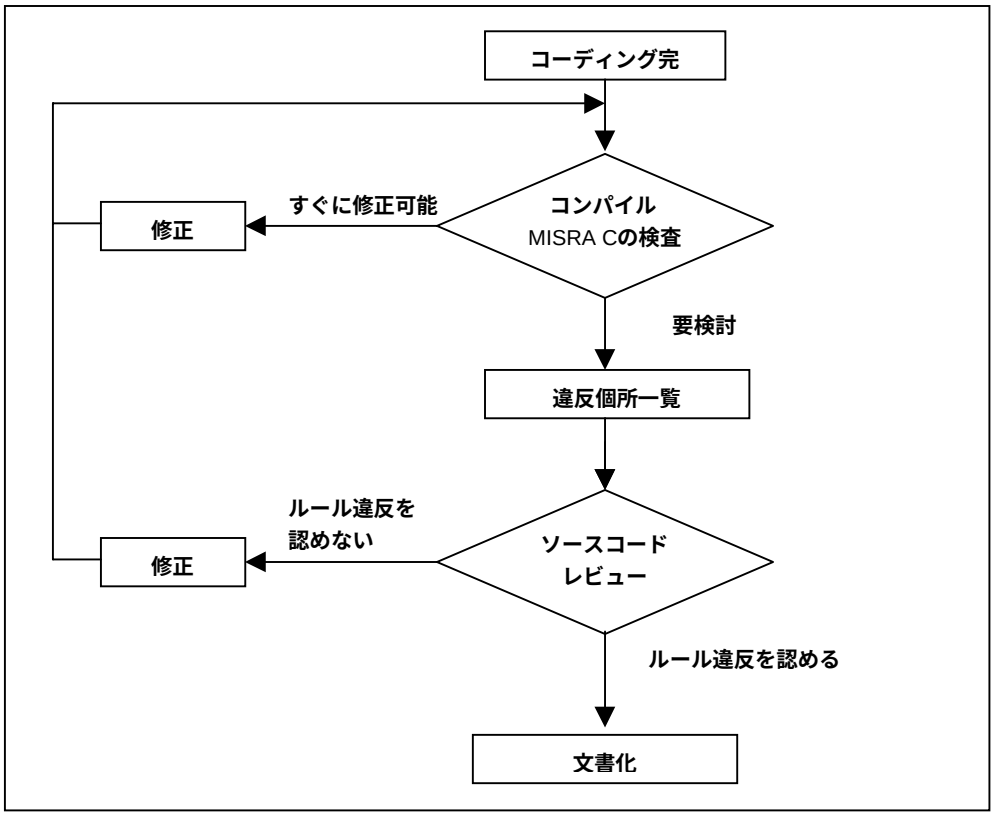


図 10.8　SQMlint **を用いた開発手順**

（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。

- H8C/C++コンパイラパッケージ V.6.01 Release00 以上（Windows 版）

11. Q&A 集

本章では、過去に問い合わせが多かった項目を、Q&A 集としてまとめました。

No.	ツール名	内容	参照
1	C/C++	文字列の割り付け先の変更について	11.1.1
2	コンパイラ	1 ビットのデータが正しく判定されない	11.1.2
3		DOS 画面からの起動方法	11.1.3
4		実行時ルーチンの仕様とスピード	11.1.4
5		H8 ファミリオブジェクト互換性	11.1.5
6		稼動するホストマシンと OS について	11.1.6
7		C/C++ソースレベルのデバッグができない	11.1.7
8		インライン展開時にウォーニングメッセージが出る	11.1.8
9		Function Not Optimize が出る	11.1.9
10		インクルードファイルの指定について	11.1.11
11		プログラムの日本語記述について	11.1.11
12		クロスアセンブラで Illegal value in operand が出力される	11.1.12
13		最適化によってコードが大幅に削除されてしまう	11.1.13
14		デバッグ時にローカル変数の値が見えない	11.1.14
15		最適化オプションについて	11.1.15
16		関数の引数が正しく渡されない	11.1.16
17		ライト専用レジスタのビット操作が正しく行われない	11.1.17
18		アセンブリプログラムとのリンケージで注意すべき点	11.1.18
19		不正動作になりやすいコーディングを調べたい	11.1.19
20		コメントの記述について	11.1.20
21		各ファイルごとにオプションを指定する方法	11.1.21
22		アセンブラを埋め込んだ場合のビルドの仕方	11.1.22
23		リンク時にシンタックスエラーが出力される	11.1.23
24		C++言語仕様についての機能	11.1.24
25		プリプロセッサ展開後のソースが見たい	11.1.25
26		MACH、MACL レジスタの退避／回復コードを出力したい	11.1.26
27		ICE でうまく行くが実チップ上では暴走する	11.1.27
28		SH マイコン用に開発した C プログラムの利用について	11.1.28
29		グローバルオプションの変更方法	11.1.29
30		最適化により無限ループになる	11.1.30
31		ビットフィールドのリードライト命令	11.1.31
32		プログラムを長時間実行すると一般不当命令例外発生することがある	11.1.32
33		整数演算結果が期待値と異なる	11.1.33

No.	ツール名	内容	参照
34	最適化リンケージエディタ	Undefined External Symbol が出力される	11.2.1
35		Relocation Size Overflow が出力される	11.2.2
36		RAM 上でプログラムを実行したい	11.2.3
37		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	11.2.4
38		オーバレイの実現	11.2.5
39		未定義シンボルのエラー出力指定	11.2.6
40		S タイプファイルの出力形式の統一	11.2.7
41		出力ファイルの分割	11.2.8
42		最適化リンケージエディタが出力するファイル形式	11.2.9
43		プログラムサイズ(ROM,RAM)の算出方法	11.2.11
44		Section alignment mismatch が出力される	11.2.11
45	標準ライブラリ構築ツール	リエントラントと標準ライブラリ	11.3.1
46		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	11.3.2
47		標準ライブラリが存在しない（H8C V4 以降）	11.3.3
48		標準ライブラリ構築時のウォーニング	11.3.4
49		ヒープ領域で使用するメモリのサイズ	11.3.5
50		入出力用ライブラリの ROM サイズを減らす方法	11.3.6
51		ライブラリファイルを編集したい	11.3.7
52	HEW	ダイアログメニューが正しく表示されない	11.4.1
53		オブジェクトファイルのリンク順序	11.4.2
54		プロジェクトファイルの除外	11.4.3
55		プロジェクトファイルのデフォルトオプション指定	11.4.4
56		メモリマップの変更方法	11.4.5
57		HEW のネットワーク上での使用について	11.4.6
58		HEW で作成するファイル、ディレクトリ名の制限	11.4.7
59		HEW エディタ、HDI での日本語表示フォントがおかしい	11.4.8
60		HIM から HEW への変換方法	11.4.9
61		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	11.4.11

11.1 C/C++コンパイラ

11.1.1 文字列の割り付け先の変更について

■質問

文字列やデータの割り付けるセクションの属性を変更したい。

■回答

文字列は通常定数領域に割り付けられますが、次の処理をすると初期化領域へ割り付けることができます。

(1) オプションで変更する
オプションで文字列をDセクションに割り付ける指定を行うことができます。

【指定方法】
ダイアログメニュー：**コンパイラタブカテゴリ:[オブジェクト]** の**文字列データ格納:初期化データ領域**
コマンドライン　　：*string=data*

(2) 文字列の格納領域を限定する
次のようにします。

```
char *str1=”ABC”;
char str2[4]=”ABC”;
```

文字列ABC**は**C**セクションに割り付けられる**

文字列ABC**は**D**セクションに割り付けられる**

この結果次のようになります。

```
     .SECTION    D,DATA,ALIGN=2
_str1:
     .DATA.L     L2
_str2:
     .SDATAZ     "ABC"
     .SECTION    C,DATA,ALIGN=2
L2:
     .SDATAZ     "ABC"
```

（3） 定数領域に割り付けたデータをvolatile指定すると初期化領域に割り付けられます。
次のようなデータの場合

```
const int a=1;
```

volatileオプションを指定します。

（指定なし）

```
    .SECTION    C,DATA,ALIGN=2
_a:
    .DATA.W     H'0001
```

（指定あり）

```
    .SECTION    D,DATA,ALIGN=2
_a:
    .DATA.W     H'0001
```

【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] ループ判定式の最適化抑止**
コマンドメニュー　：*volatile*

11.1.2 1 ビットのデータが正しく判定されない

■質問

1 ビットのデータを“ 1 ”と比較すると、正しく分岐しない場合があるようですが、どうしてですか。

■回答

データを符号付き（int,short,char）で宣言していないか確認してください。
ビットフィールドでサイズが 1 ビットのデータを、符号付き（signed）で宣言した場合、1 ビットデータそのものを符号として解釈します。
したがって、表現できる値は、“ 0 ”と“ −1 ”だけになります。
“ 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。

（判断が常に偽となる例）

```
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;
}
```

（正しく判断される例）

```
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;
}
```

11.1.3 DOS 画面からの起動方法

■質問

PC 版で DOS 画面からコマンドで H8S,H8/300C/C++コンパイラシステムを起動させる方法を知りたい。

■回答

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にPATHを設定します。
例） 各ツールがC:¥Hew2¥Tools¥Hitachi¥H8¥5_0_1¥bin**とした場合。**
c:¥> PATH=%PATH%;C:¥Hew2¥Tools¥Hitachi¥H8¥5_0_1¥bin (RET)
既に設定されているPATHに追加します。

（2） CH38の設定
コンパイラで使用するシステムインクルードファイルの位置を示します。
例） システムインクルードファイルがC:¥Hew2¥Tools¥Hitachi¥H8¥5_0_1¥include**に存在するとした場合。**
c:¥> set CH38=C:¥Hew2E¥Tools¥Hitachi¥H8¥5_0_1¥include (RET)

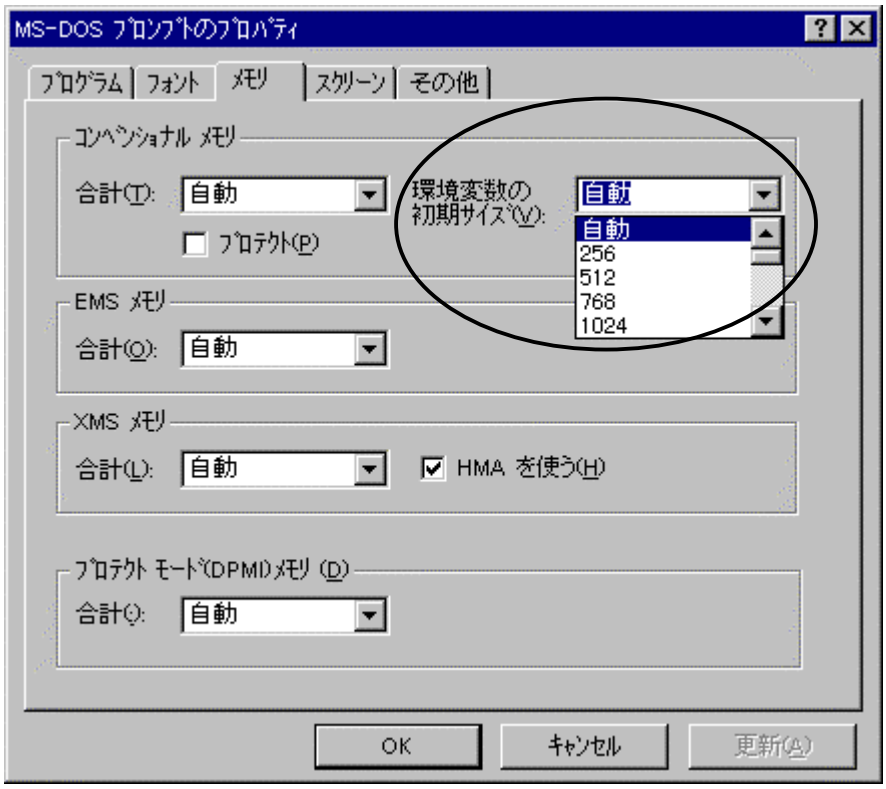
（3） CH38TMPの設定
コンパイラが生成する中間ファイル作成ディレクトリを設定します。
例）中間ファイル作成ディレクトリをC:¥tempとした場合。
c:¥> set CH38TMP=C:¥temp
この指定を行わなかった場合、中間ファイルはカレントディレクトリに作成します。通常は設定する必要はありませんが、カレントディレクトリのディスク容量が不足してしまう場合などに利用します。

（4） H38CPUの設定
CPU/動作モードを指定しておきます。
例） CPU/動作モードを2600a:24と指定したい場合。
c:¥> set H38CPU=2600a:24
この指定はコンパイラのオプションで指定することもできます。また、オプションの指定と異なった場合は、オプション指定を優先させます。

■備考

これらの環境設定をして、コンパイラを起動させたときに「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。

DOS プロンプトのプロパティを開きます。



[コンベンショナル メモリ]の環境変数の初期サイズを増やしてください。1024 以上を推奨します。

この設定後、DOS プロンプトを開き直します。

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一覧を掲載します。

ライブラリ構築時のオプションはデフォルト指定です。

実行時ルーチン速度一覧（1）

項番	種類	関数名	300	300HN	300HA	2000N	2000A	H8sxn	H8sxa	H8sxx
1	加算	$ADDD$3	1002	746	480	206	208	175	175	175
2		$ADDF$3	426	216	174	102	104	87	87	87
3		$ADDL$3	76	-	-	-	-	-	-	-
4	減算	$SUBD$3	1212	618	626	268	272	240	226	226
5		$SUBF$3	448	224	228	106	108	91	91	91
6		$SUBL$3	76	-	-	-	-	-	-	-
7	乗算	$MULD$3	1886	984	992	606	610	539	539	539
8		$MULF$3	702	388	392	220	222	192	192	192
9		$MULI$3	102	-	-	-	-	-	-	-
10		$MULL$3	304	130	134	95	88	-	-	-
11		$MULXSB$3	60	-	-	-	-	-	-	-
12		$MULXSW$3	168	-	-	-	-	-	-	-
13		$MULXUW$3	148	-	-	-	-	-	-	-
14		$CMLI$3	142	-	-	-	-	-	-	-
15	除算	$DIVC$3	82	-	-	-	-	-	-	-
16		$DIVD$3	7304	2544	356	1236	1238	1248	1248	1248
17		$DIVF$3	1688	1176	1180	551	553	649	649	649
18		$DIVI$3	262	-	-	-	-	-	-	-
19		$DIVL$3	1068	154	162	95	99	91	91	91
20		$DIVUI$3	208	-	-	-	-	-	-	-
21		$DIVUL$3	1038	100	108	68	70	91	91	91
22		$DIVUX$3	936	-	-	-	-	-	-	-
23		$DIVXSB$3	80	-	-	-	-	-	-	-
24		$DIVXSW$3	188	-	-	-	-	-	-	-
25		$DIVXUW$3	158	-	-	-	-	-	-	-
26		$CDVC$3	132	-	-	-	-	-	-	-
27		$CDVI$3	310	-	-	-	-	-	-	-
28		$CDVUI$3	258	-	-	-	-	-	-	-
29	剰余	$MODL$3	254	-	-	-	-	-	-	-
30		$MODUL$3	224	-	-	-	-	-	-	-
31		$CMDC$3	132	-	-	-	-	-	-	-
32		$CMDI$3	310	-	-	-	-	-	-	-
33		$CMDUI$3	256	-	-	-	-	-	-	-
34	ポスト	$POID$3	1164	624	542	278	283	-	-	-
35	インクリ	$POIF$3	476	-	-	-	-	-	-	-
36	メント	$POIL$3	102	-	-	-	-	-	-	-
37	ポスト	$PODD$3	1114	604	618	268	273	-	-	-
38	デクリ	$PODF$3	490	-	-	-	-	-	-	-
39	メント	$PODL$3	98	-	-	-	-	-	-	-
40	プリ	$PRID$3	1112	572	498	254	267	229	228	228
41	インクリ	$PRIF$3	448	314	292	123	127	101	99	99
42	メント	$PRIL$3	56	-	-	-	-	-	-	-
43	プリ	$PRDD$3	1066	556	578	246	259	216	212	212
44	デクリ	$PRDF$3	466	326	342	131	135	108	106	106
45	メント	$PRDL$3	56	-	-	-	-	-	-	-
46	論理演算	$ANDL$3	78	-	-	-	-	-	-	-
47		$NEGD$3	74	76	80	38	40	20	20	20
48		$NEGF$3	50	-	-	-	-	-	-	-

実行時ルーチン速度一覧（2）

項番	種類	関数名	300	300HN	300HA	2000N	2000A	H8sxn	H8sxa	H8sxx
49	論理演算	$NEGL$3	76	-	-	-	-	-	-	-
50		$ORL$3	78	-	-	-	-	-	-	-
51		$XORL$3	78	-	-	-	-	-	-	-
52	ブロック転送	$MV4$3	48	-	-	-	-	-	-	-
53		$MV8$3	72	72	76	36	38	17	17	17
54		$MVN$3	170	296	328	138	146	64	71	71
55		$mv3mm $	-	-	-	30	32	-	-	-
56		$mv3mr$	-	-	-	28	30	-	-	-
57		$mv3rm$	-	-	-	17	19	-	-	-
58		$mv4mm$	-	-	-	36	38	-	-	-
59		$mv4mr$	-	-	-	31	33	-	-	-
60		$mv4rm$	-	-	-	20	22	-	-	-
61	ビット	$BFINC$3	102	96	100	47	49	-	-	-
62	フィールド	$BFINCR$3	94	88	92	43	45	-	-	-
63	の設定	$BFINI$3	256	180	184	71	73	35	35	35
64		$BFINIR$3	248	156	160	67	69	31	31	31
65		$BFINL$3	820	346	350	135	137	45	45	45
66		$BFINLR$3	-	330	334	127	129	39	39	39
67	ビット	$BFSC$3	78	78	82	38	40	-	-	-
68	フィールド	$BFSI$3	196	168	172	67	69	34	34	34
69	の参照	$BFSL$3	578	270	270	122	124	37	37	37
70		$BFUC$3	68	68	72	33	35	-	-	-
71		$BFUI$3	168	144	148	55	57	-	-	-
72		$BFUL$3	546	236	240	105	107	-	-	-
73	比較	$CMPD$3	230	226	218	101	97	66	62	62
74		$CMPF$3	178	90	94	45	47	36	36	36
75		$CMPL$3	94	-	-	-	-	-	-	-
76		$EQD$3	254	250	246	113	111	87	73	73
77		$EQF$3	202	114	122	57	61	49	47	47
78		$GED$3	264	250	256	118	116	91	77	77
79		$GEF$3	202	114	122	57	61	49	47	47
80		$GTD$3	262	250	254	117	115	90	76	76
81		$GTF$3	202	114	122	57	61	49	47	47
82		$LED$3	264	250	266	123	121	93	79	79
83		$LEF$3	212	114	122	57	61	49	47	47
84		$LTD$3	264	250	266	123	121	93	79	79
85		$LTF$3	212	115	122	57	61	49	47	47
86		$NED$3	250	252	248	114	112	78	75	75
87		$NEF$3	204	116	124	58	62	47	47	47
88	変換	$CTOL$3	60	-	-	-	-	-	-	-
89		$DTOF$3	316	238	242	110	112	87	87	87
90		$DTOI$3	508	-	-	-	-	-	-	-
91		$DTOL$3	464	290	294	100	102	105	105	105
92		$FTOD$3	178	144	148	62	64	56	56	56
93		$FTOI$3	608	-	-	-	-	-	-	-
94		$FTOL$3	564	338	342	150	152	188	188	188
95		$ITOD$3	176	152	156	74	76	82	84	84
96		$ITOF$3	164	124	128	62	64	80	80	80
97		$ITOL$3	44	-	-	-	-	-	-	-
98		$LTOD$3	366	244	256	126	128	150	150	150

実行時ルーチン速度一覧（3）

項番	種類	関数名	300	300HN	300HA	2000N	2000A	H8sxn	H8sxa	H8sxx
99	**変換**	$LTOF$3	334	224	236	116	118	151	151	151
100		$ULTOD$3	180	84	124	54	56	55	51	51
101		$ULTOF$3	150	22	104	50	52	47	47	47
102		$UTOD$3	114	62	94	43	45	38	36	36
103		$UTOF$3	80	22	52	21	23	25	25	25
104	**左シフト**	$DSLC$3	70	70	84	31	37	-	-	-
105		$DSLI$3	82	78	92	35	41	-	-	-
106		$DSLL$3	-	98	112	45	51	-	-	-
107		$SLC$3	-	-	-	23	25	-	-	-
108		$SLI$3	62	-	-	26	28	-	-	-
109		$SLL$3	118	-	-	29	31	-	-	-
110	**右シフト**	$DSRC$3	70	70	84	31	37	-	-	-
111		$DSRI$3	88	78	92	35	41	-	-	-
112		$DSRL$3	-	98	112	45	51	-	-	-
113		$DSRUC$3	70	70	84	31	37	-	-	-
114		$DSRUI$3	88	78	92	35	39	-	-	-
115		$DSRUL$3	-	98	112	45	51	-	-	-
116		$SRC$3	-	-	-	18	25	23	18	19
117		$SRI$3	68	-	-	28	28	17	17	17
118		$SRL$3	110	-	-	29	31	18	18	18
119		$SRUC$3	-	-	-	23	25	-	-	-
120		$SRUI$3	68	-	-	26	28	-	-	-
121		$SRUL$3	110	-	-	29	31	-	-	-
122	**レジスタ**	fp_regld3	52	70	80	-	-	-	-	-
123	**退避／回復**	$fp_rgld3$3	46	60	70	-	-	-	-	-
124		fp_regsv3	52	70	80	-	-	-	-	-
125		$fp_rgsv3$3	46	60	70	-	-	-	-	-
126		sp_regld3	58	80	90	-	-	-	-	-
127		$sp_rgld3$3	52	70	90	-	-	-	-	-
128		sp_regsv3	58	80	90	-	-	-	-	-
129		$sp_rgsv3$3	52	70	90	-	-	-	-	-
130		$spregld2$3	50	66	70	-	-	-	-	-
131		$sprgld23$3	40	56	60	-	-	-	-	-
132	**その他**	$SWI$3	124	-	-	-	-	-	-	-

■備考

測定条件は、実行時ルーチンを呼び出して戻ってくるまでです。

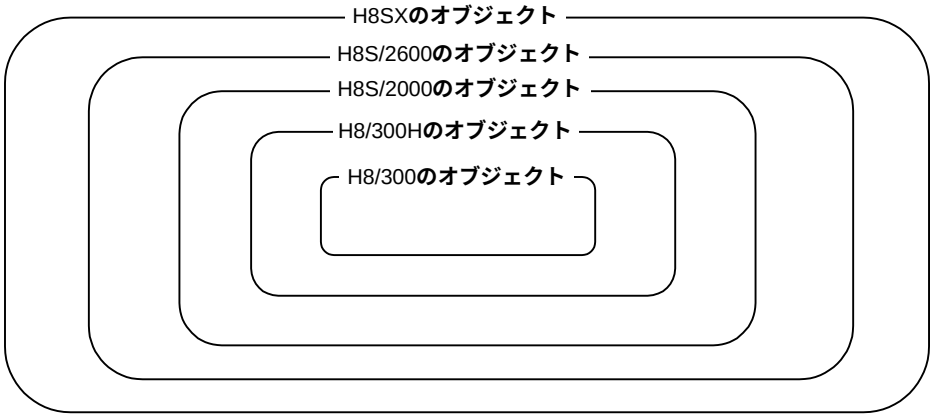
11.1.5 H8 ファミリオブジェクト互換性

■質問

コンパイルオプション"-cpu=300"（または 300h、2000、2600、および h8sx）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、H8/300 のオブジェクトと H8S/2000 のオブジェクトをリンクし H8S/2000 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。



オブジェクト相互関係

11.1.6 稼動するホストマシンと OS について

■質問

起動するホストマシンと OS のバージョンを知りたい。

■回答

動作環境は以下のようになります。

H8S,H8/300 C/C++コンパイラパッケージ

ホストマシン	OS	ディスク容量
IBM-PC/AT	Windows98/Me/2000/XP/NT 4.0	約 120MB
HP9000	HP-UX 10.2	約 30MB
Sun SPARC	日本語 Solaris2.5 以上	約 30MB

このほかに、オンラインマニュアルも添付されています。
オンラインマニュアルの動作環境は次のようになっています。

- Pentium®プロセッサを搭載したパーソナルコンピュータ
- Microsoft Windows®98、Windows®ME 日本語版、Microsoft WindowsNT®4.0 日本語版、Microsoft Windows®2000、Windows®XP 日本語版
- 2 倍速以上の CD-ROM ドライブ
- ディスク容量：約 15MB

オンラインマニュアルは Windows®98、Windows®ME、Windows NT®4.0、Windows®2000、または Windows®XP 上で参照できます。

11.1.7 C/C++ソースレベルのデバッグができない

■質問

コンパイル時にデバッグ情報出力を指定していますが、C ソースレベルのデバッグができません。

■回答

次の点を確認してください。

(1) コンパイル時、モジュール間最適化時のそれぞれにデバッグ情報出力指定されていますか？
デバッガによって、対応する出力オブジェクトのフォーマットと、デバッグ情報の出力の仕方が異なります。以下に、使用可能なデバッガ例と出力オブジェクト／デバッグ情報の関係を示します。

使用可能なデバッガ例	オブジェクトフォーマット	デバッグ情報出力	デバッグ情報出力方法
3rd party 製 ELF/DWARF2 サポートデバッガ	ELF/DWARF2	debug	ロードモジュール内
3rd party 製 ELF/DWARF サポートデバッガ	ELF	debug	ロードモジュール内
日立統合化マネージャ（Ver4 以降）+ E7000	SYSROFPLUS	sdebug	デバッグ情報ファイル
日立統合化マネージャ（Ver3 以降）+ E7000	SYSROF	debug	デバッグ情報ファイル
日立デバッギング インタフェース（Ver.2 以降）+ E6000	SYSROF	debug	ロードモジュール内
日立デバッギング インタフェース（Ver.3 以降）+ E6000	ELF	sdebug	デバッグ情報ファイル

【注意事項】
C++言語を使用してプログラムを組んだ場合は、オブジェクトフォーマットを ELF で出力しなければなりません。

(2) コンパイル時のソースプログラムがあるディレクトリを変更指定していませんか？
デバッグ情報はソースプログラムのディレクトリ位置を含めて格納しています。したがって、コンパイル時のソースプログラムがあるディレクトリを変更した場合、ソースプログラムを変更できなくなります。なお、デバッガによっては、ソースプログラムのディレクトリを指定する機能をサポートしているものがあります。
また、ディレクトリを移動した場合など、dwfinfディレクトリも含めて移動してください。
dwfinfディレクトリにはモジュール間最適化用付加情報ファイルが格納されますので、デバッグの際に必要になることがあります。

(3) Cソースファイルをアセンブリ出力したファイルをデバッグ中ですか？
この場合は、コンパイル時と、アセンブル時にそれぞれデバッグ情報の出力を指定してください。
これにより、Cソースレベルのステップ実行、外部変数を参照できるようになります。

■回答 2

-code=asm を指定したときは C/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/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。

■備考

C ソースレベルのデバッグの詳細については「H8S,H8/300 シリーズシミュレータ・デバッガユーザーズマニュアル」を参照してください。

11.1.8 インライン展開時にウォーニングメッセージが出る

■質問

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"のウォーニングメッセージが出ました。

■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。
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
- 可変パラメータを持つ関数である
- 関数内でパラメー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

```
#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

#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。

11.1.9 Function not optimize が出る

■質問

"Function not optimized"のウォーニングメッセージが出力されます。しかし、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付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。
メッセージが出力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あら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。

(2) メモリが足りない場合
H8S,H8/300 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、このウォーニングメッセージを出力します。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最適化オプションを指定しない場合と同じです。このウォーニングメッセージを回避するには、C/C++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。またはコンパイラの使えるメモリを増やしてください。

11.1.10 インクルードファイルの指定について

■質問

（1） 他のディレクトリにある、インクルードファイルを指定したい。
（2） 既存のファイルにインクルード指定をしたい。

■回答

これらはコンパイラの機能により回避することができます。
以下に、それぞれの説明をいたします。
（1） 指定したインクルードファイルのディレクトリを指定すると、そのディレクトリにある、インクルードファイルを指定することができるコンパイラオプションを用意しています。

【指定方法】
ダイアログメニュー：**コンパイラタブカテゴリ:[ソースファイル]オプション項目:**の**インクルードファイルディレクトリ**
コマンドライン　　：*include*

（2） ファイルを指定すると、ソースファイルでインクルード指定されていないファイルでも、インクルード指定することができます。指定したファイルの内容をコンパイル単位の先頭に取り込みます。

【指定方法】
ダイアログメニュー：**コンパイラタブカテゴリ:[ソースファイル] オプション項目:デフォルトインクルードファイル**
コマンドライン　　：*preinclude*

11.1.11 プログラムの日本語記述について

■質問

プログラム中の文字列や、コメントへの日本語の記述が可能ですか？

■回答

可能です。ただし、基本的に、日本語環境はホストマシンの日本語環境に依存します。各ホストマシンの日本語環境は以下のとおりです。

ホストマシン	日本語コード
PC	**シフト** JIS **コード**
HP9000	**シフト** JIS **コード**
SPARC	EUC **コード**

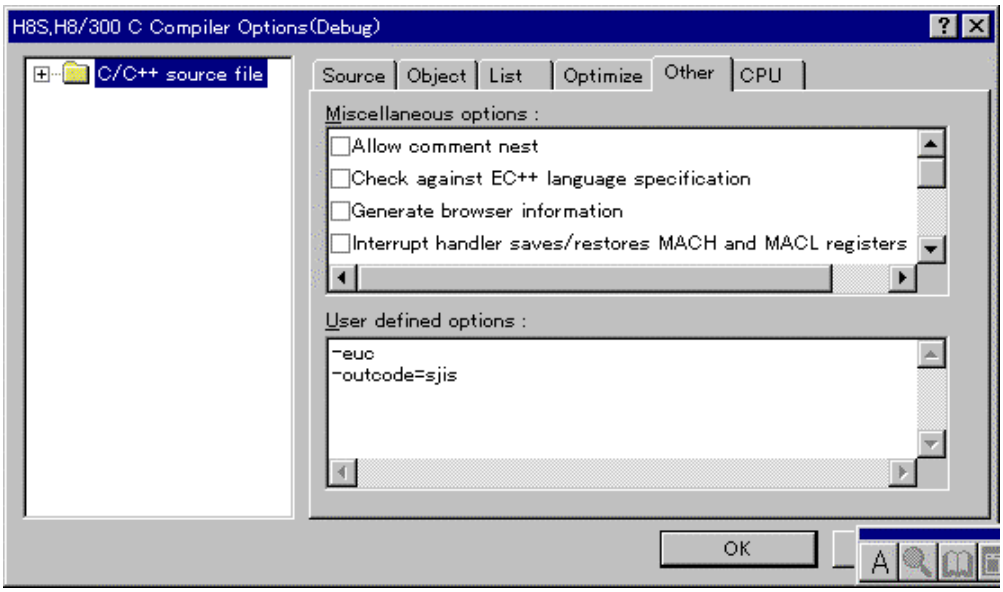
しかし、たとえば、SPARC で作成したファイルをＰＣでコンパイルしたい場合などは、日本語コードの認識を変更する必要があります。
これはオプションで指定します。
コマンドオプションは以下のとおりです。

コマンドオプション	内容
sjis	**シフト** JIS **コードを選択**
euc	EUC **コードを選択**
latin1	Latin1 **コードを選択**

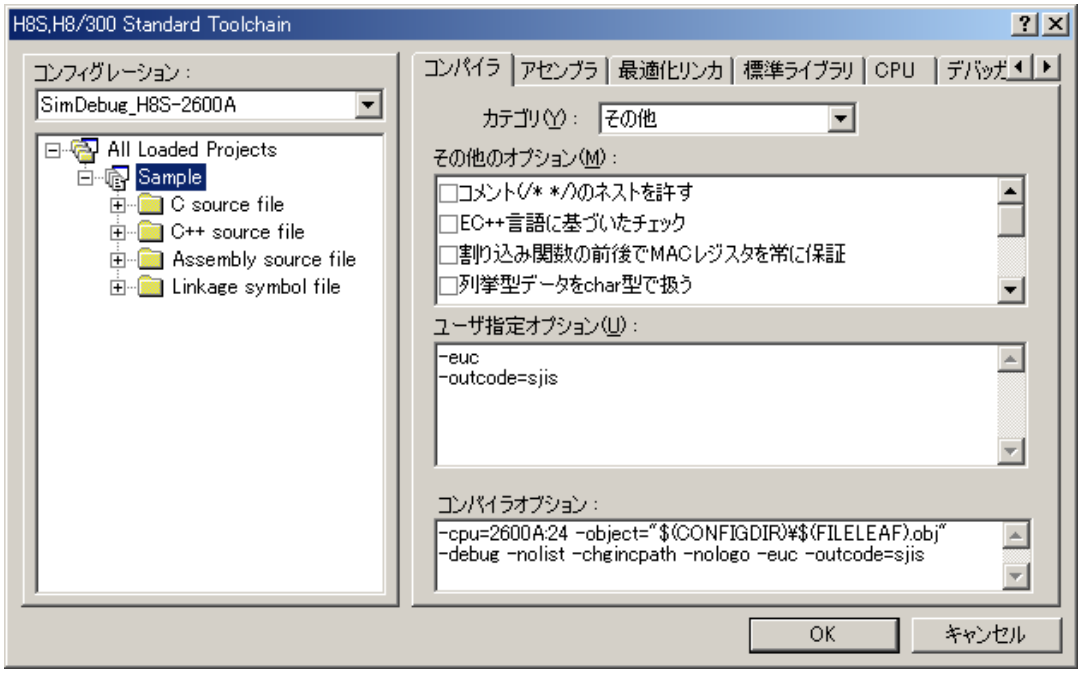
また、*outcode* オプションでオブジェクトプログラムに出力する漢字コードを指定することができます。*outcode=sjis* オプションは漢字コードをシフト JIS コードで出力します。*outcode=euc* オプションは漢字コードを EUC コードで出力します。

HEW で日本語環境をオブジェクトプログラム出力コードを指定する場合は Other タブの User defined options;[ユーザ指定オプション:]に記述します。オプションの指定方法はコマンドラインでの指定と同じです。

<HEW1.2>



<HEW2.0 以降>



11.1.12 クロスアセンブラで Illegal value in operand が出力される

■質問

コンパイラで、アセンブリソース出力したファイルをクロスアセンブラでアセンブルすると、
“ Illegal value in operand ”のエラーメッセージが出力されます。

■回答

#pragma asm～#pragma endasm、または#pragma inline_asm でアセンブラ埋め込み処理を行っていないか確認してください。

これは、アセンブラ埋め込み箇所を含んだ分岐幅を 16 ビットディスプレースメントで出力しているため、実際の分岐幅がその範囲を超えると出力されます。分岐幅が届くように JMP 命令を使用して、コンパイラ出力のアセンブリプログラムを修正してください。

■使用例

次のように変更します。

＜アセンブリ展開結果＞

```
_sub:
    MOV.L       #_a:32,ER0
    MOV.L       @ER0,ER1
    INC.L       #1,ER1
    MOV.L       ER1,@ER0
    RTS
_main:
    MOV.L       #_a:32,ER1
    MOV.L       @ER1,ER0
    CMP.L       #2:32,ER0
    BNE         L8
    BRA         _sub
```

ASM**埋め込み部分**

```
L8:
    MOV.L       @ER1,ER0
    INC.L       #1,ER0
    MOV.L       ER0,@ER1
    RTS
```

```
BEQ         $+6:8
JMP         _L8
BSR         @_sub
```

11.1.13 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) 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。

```
  void func(void)
_func:
          PUSH.L     ER6
   {
       int res1,res2,res3;
       res1=data1*data2;
          MOV.W      @_data1:32,R0
          MOV.W      @_data2:32,E0
          MULXU.W    E0,ER0
          MOV.W      R0,R6
       res2=data2*data3;
       res3=data3*data1;
          MOV.W      @_data1:32,R1
          MOV.W      @_data3:32,E1
          MULXU.W    E1,ER1
       sub(res1,res1,res3);
          MOV.W      R0,E0
          JSR        @_sub:24
   }
          POP.L      ER6
          RTS
```

←res2はこの後参照されないため式自体が削除される

←第2パラメータの記述ミス
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。

(2) 外部変数への代入が最適化された

つぎのような外部変数への代入式などは、最適化され、最後の演算式のみが反映されます。

```
   int glb;
   void main()
_main:
   {
       glb=0;
       glb=1;
          MOV.W   #1:16,R0
          MOV.W   R0,@_glb:32
   }
          RTS
```

←glb=0のコードは生成されない

その場合は、外部変数の型宣言の際にvolatile指定をしておくとglb=0のコードも生成されます。コンパイラの*volatile*オプションを指定すると、ファイル全体の外部変数をvolatile指定することができます。

11.1.14 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。

```
int x;
void func(void)
{
    int a;
    a=3;
    x=x+a;
}
```

こういった場合は、aはコンパイル時にx=x+3;となり、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

```
void 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

```
int data1,data2,data3;
void func(void)
{
    int res1,res2,res3;

    res1=data1*data2;
    res2=data2*data3;
    res3=data3*data1;
    sub(res1,res1,res3);
}
```

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。

11.1.15 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いたします。

11.1.16 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ません。

■回答

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を指定していないと、動作不良になります。

（動作を保証しない指定例）

```
void f(x)
char x;
{
   x+=10;
}
void main(void)
{
   char x;
   f(x);
}
```

（正しい指定例）

```
void f(char x)
{
   x+=10;
}
void main(void)
{
   char x;
   f(x);
}
```

コンパイル時に**コンパイラタブカテゴリ:[ソースファイル] 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**（*message* オプション）でチェックする方法があります。この指定では、各インフォメーションメッセージの出力を選択できます。(I)0200 No prototype function をチェックすると、関数の原型宣言の有無をチェックできます。

■備考

上記の「動作を保証しない指定例」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 int 型に変換します。引数の原型宣言によって型宣言がされていない場合、以下のように型変換されます。

- char 型、unsigned char 型は、int 型に変換されます
- float 型の引数は、double 型に変換されます
- 上記以外の型は、型変換されません

11.1.17 ライト専用レジスタのビット操作が正しく行われない

■質問

ライト専用レジスタをビット操作すると目的の結果が得られません。

■回答

本コンパイラは、BSET、BCLR、BNOT、BST、BIST の各ビット操作命令を生成します。これらの命令は、バイト単位でデータを読み込み、ビット操作後に再びバイト単位でデータを書き込みます。一方、CPU は、ライト専用レジスタを読み込むと、レジスタの内容に関係なく不定値のデータを取り込みます。このため、ライト専用レジスタのビット操作命令では、操作するビット以外のビットの内容が変化してしまう場合があります。

■対策

ライト専用レジスタを直接ビット操作しないでください。
1 バイトデータにいったん代入してから演算します。以下に操作例を示します。

（インクルードファイル（300x.h））

```
struct  S_p4ddr {
  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;
};
union SS {
  unsigned char Schar;
  struct S_p4ddr Sstr;
};
#define P4DDR (*(union SS
*)0xffffc5)
#define P0 0x1
```

（Cプログラム）

```
#include "300x.h"
unsigned char DDR;
//書き込み専用レジスタのバックアップ用
//データを用意します
void sub(void)
{

  DDR &=~P0;
  P4DDR.Schar=DDR;
}
```

■備考

ライト専用レジスタには、I/O ポートや周辺機能用レジスタなど、さまざまな種類があります。したがって、プログラミング時には、各製品のハードウェアマニュアルを参照して、ライト専用レジスタを確認するようにお願いいたします。

11.1.18 アセンブリプログラムとのリンケージで注意すべき点

■質問

（1） アセンブリプログラムのサブルーチンをCプログラムから呼び出す場合に、アセンブリプログラム側で注意しなければならないことは何ですか。

（2） Cプログラムのサブルーチンをアセンブリプログラムから呼び出す場合に、アセンブリプログラム側で注意しなければならないことは何ですか。

■回答

（1） アセンブリプログラムのサブルーチンをCプログラムから呼び出す場合、アセンブリプログラム内で下記レジスタを使用するときは、関数の入口／出口でレジスタの退避／回復を行ってください。

CPU シリーズ	引数渡し用レジスタ数 2	引数渡し用レジスタ数 3
H8SX,H8S/2600, H8S/2000 H8/300H	最適化指定時　：ER2～ER6	最適化指定時　：ER3～ER6
	最適化なし指定時：ER2～ER5	最適化なし指定時：ER3～ER5
H8/300	最適化指定時　：R2～R6	最適化指定時　：R3～R6
	最適化なし指定時：R2～R5	最適化なし指定時：R3～R5

（2） Cプログラムのサブルーチンをアセンブリプログラムから呼び出す場合、Cプログラムでは下記レジスタの値を呼び出し前後で保証しません。アセンブリプログラム内で使用している場合は、Cプログラム呼び出し前に退避してください。

CPU シリーズ	引数渡し用レジスタ数 2	引数渡し用レジスタ数 3
H8SX,H8S/2600, H8S/2000 H8/300H	ER0,ER1	ER0,ER1,ER2
H8/300	R0,R1	R0,R1,R2

■備考

アセンブリプログラムとのリンケージの詳細については H8S、H8/300 **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「**9.3 C/C++**プログラムとアセンブリプログラムとの結合」**に説明があります。

11.1.19 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.6.1 以上であれば MISRA-C チェックツールを使用できます。

例）

```
   ch38 △ -message △ test.c (RET)
```

（Cプログラム）

```
/*  /* COMMENT */          →0001 : 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。
int tmp;
void func(int);
void main(void)
{
  long a;
  tmp=a;                   →0011 : 値の設定されていない局所変数を参照しています。
  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:[ソースファイル] 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。なお、このオプションが有効な範囲は、エラー番号で 0001～0307 の範囲です。各エラー番号の詳細については、H8S、H8/300 **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**の**「**12. **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。

11.1.20 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。ただし次のように解釈しますので注意して使用してください。コンパイラVer.4でのコメントのネストレベルは無制限ですがコンパイラVer.3でのコメントのネストレベルは255です。

■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*

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**

11.1.21 各ファイルごとにオプションを指定する方法

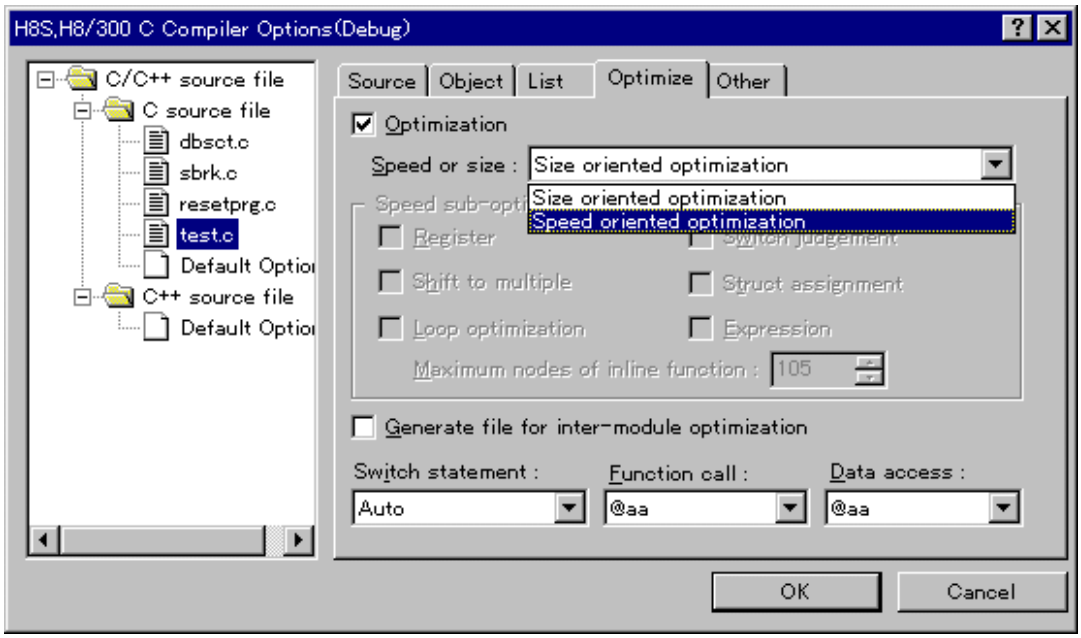
■質問

HEW システムでプロジェクトの各ファイルごとにオプションを変更して指定したい。

■回答

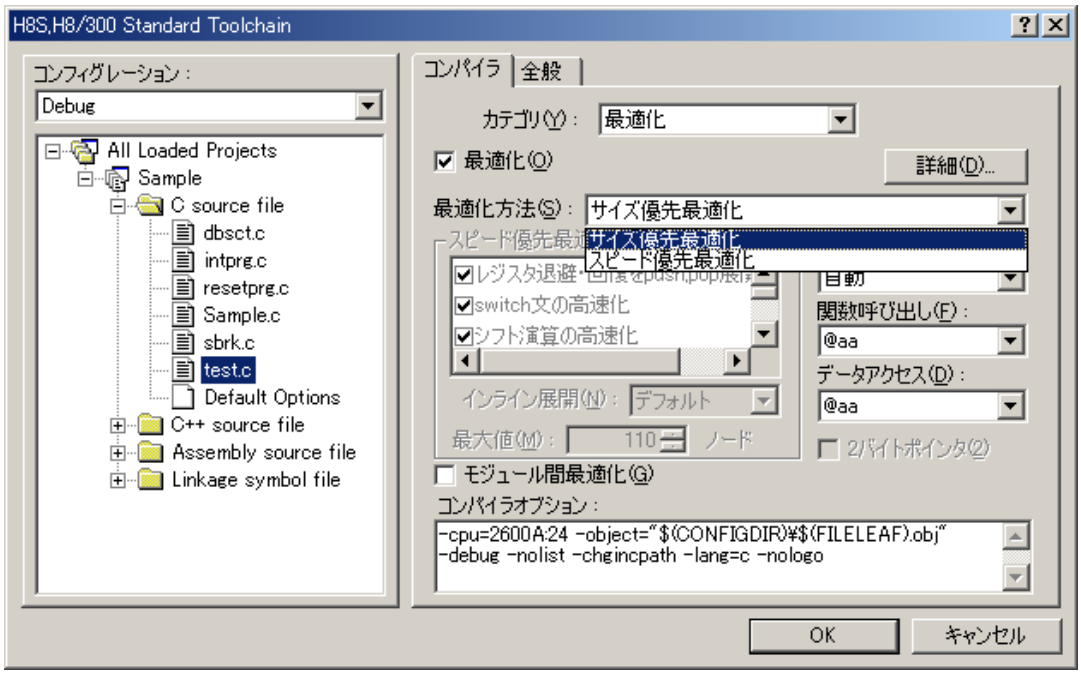
HEW システムでは、コンパイラ、アセンブラでファイルごとにオプションを変更して指定することができます。
コンパイラのオプション指定の場合、オプションの画面で左側の C/C++ source file の階層を展開します。そこで、個別のファイルをクリックし、オプション設定を行います。
フォルダごと指定した場合は、指定したディレクトリ内のファイルすべてに対してオプションが有効になります。
例としてプロジェクト中のファイルで test.c ファイルのみをスピード優先のオプションを指定したい場合は、次のようになります。

＜HEW1.2＞



左の画面から test.c を選択し、**Optimize** タブの **Speed or size:**から **Speed oriented optimization** を選択します。

＜HEW2.0 以降＞



左の画面から test.c を選択し、**コンパイラタブカテゴリ:[最適化]**の**スピード優先最適化**から**サイズ優先最適化**を選択します。

11.1.22 アセンブラを埋め込んだ場合のビルドの仕方

■質問

#pragma asm～#pragma endasm または、#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。

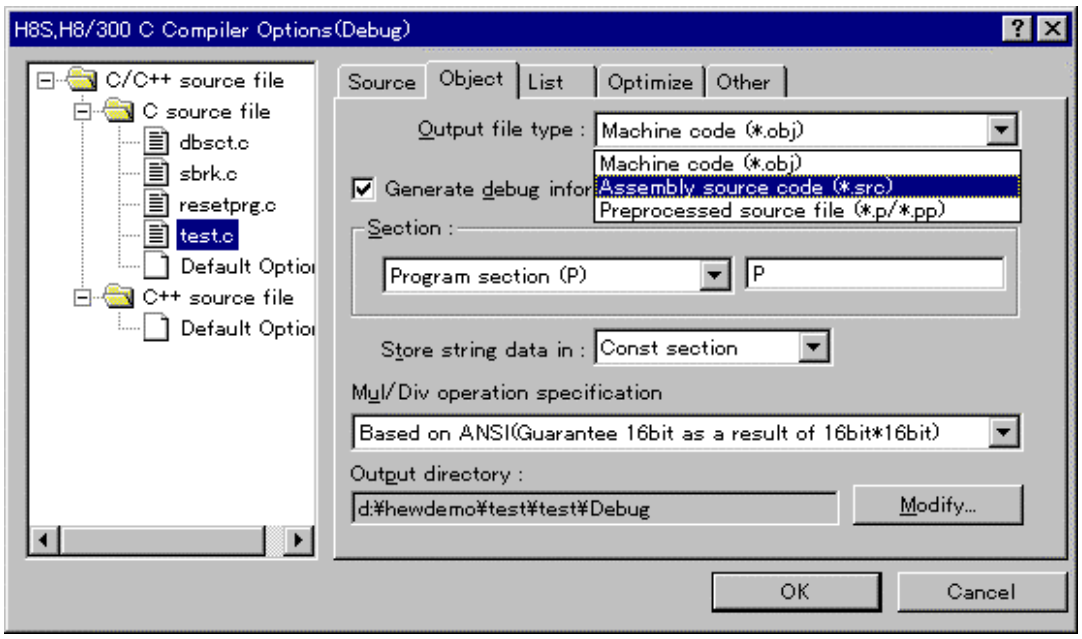
■回答

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、**「11.1.21 各ファイルごとにオプションを指定する方法」**の要領で、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。

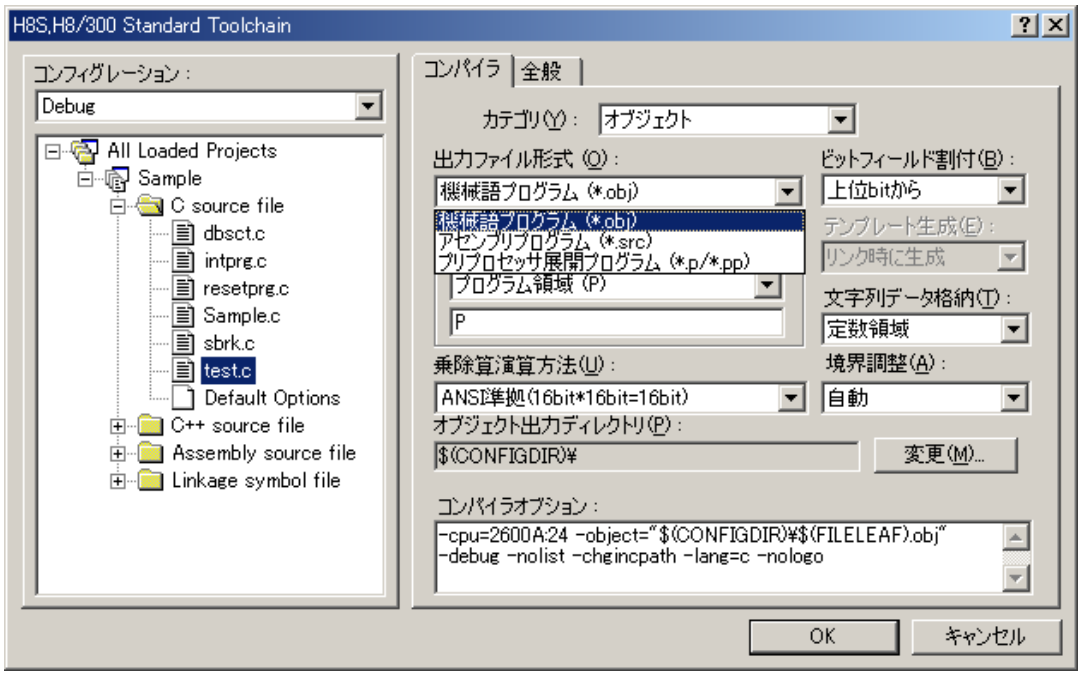
例として、test.c をアセンブラ埋め込みのあるファイルとすると、次のように指定します。

＜HEW1.2＞



Object タブの **Output file type:**から **Assembly source code (*.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/C++ソースレベルデバッグができなくなります。

＜HEW2.0 以降＞



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。

11.1.23 リンク時にシンタックスエラーが出力される

■質問

HEW1.2 のモジュール間最適化ツールでエラー202 SYMTAX ERROR が出力されます。

■回答

ファイル名、プロジェクト名に漢字、マイナス記号、空白文字などを含んでいませんか？
コンパイラ、アセンブラ、モジュール間最適化ツール、ライブラリアン、S タイプコンバータはファイル名に、漢字、マイナス記号、空白文字を指定できません。たとえば、プロジェクト名に漢字を含み、かつモジュール間最適化ツールのオプションで出力先を選択の際に、プロジェクト名に漢字が含まれるため、シンタックスエラーが出力されてしまいます。

■備考

HEW2.0 以降では、ファイル名、プロジェクト名に漢字、マイナス記号、空白文字などを含んでいてもエラーは出力されず正常にビルドできますが、漢字、マイナス記号、空白文字は、なるべく使用しないほうがよいでしょう。

11.1.24 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

H8S,H8/300 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 H8S、H8/300 **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**の「10.3.2 C++**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
・＋,－,<<等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 H8S、H8/300 **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**の「9.2.2(5) C/C++**ライブラリ関数の初期設定（**__INITLIB**）」**を参照してください。

11.1.25 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*

11.1.26 MACH、MACL レジスタの退避／回復コードを出力したい

■質問

MAC レジスタの退避／回復コードを出力する方法が知りたい。

■回答

コンパイラのオプションで MAC レジスタの退避／回復コードを出力することができます。
この指定をすると、割り込み関数内で MAC レジスタを使用する場合（組み込み関数の mac、macl を使った場合）、または割り込み関数内で関数呼び出しがある場合、MAC レジスタを常に保証します。
このオプションが指定されない場合でも、割り込み関数内で MAC レジスタを使用する場合は、MAC レジスタの退避／回復コードを出力します。

■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:割り込み関数の前後で MAC レジスタを常に保証**
コマンドライン　：*macsave*

■使用例

割り込み関数から関数 sub を呼び出します。
（C/C++プログラム）

```
extern void sub(void);
#pragma interrupt func
void func(void)
{
    sub();
}
```

（コンパイル結果アセンブリ展開コード）

オプションなし

```
_func:
     STM.L   (ER0-ER1),@-SP




     JSR     @_sub:24




     LDM.L   @SP+,(ER0-ER1)
     RTE
```

オプションあり

```
_func:
     STM.L   (ER0-ER1),@-SP
     STMAC.L MACL,ER1
     PUSH.L  ER1
     STMAC.L MACH,ER1
     PUSH.L  ER1
     JSR     @_sub:24
     POP.L   ER1
     LDMAC.L ER1,MACH
     POP.L   ER1
     LDMAC.L ER1,MACL
     LDM.L   @SP+,(ER0-ER1)
     RTE
```

11.1.27 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**「3.3 セクションアドレス演算子」**を参照してください。

11.1.28 SH 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

SH マイコン用に開発したＣプログラムを H8S,H8/300 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2バイトデータとなります。
SHではint型データは4バイトデータとして扱っていましたが、H8S,H8/300シリーズでは2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、SH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。

■備考

なお､M16C マイコン用に開発した C プログラムを H8 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 をご使用下さい。

Translation Helper は M16C マイコンから H8 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。

11.1.29 グローバルオプションの変更方法

■質問

引数渡し用のレジスタ数を変更したら、モジュール間最適化ツールでエラーとなる。

■回答

引数渡し用レジスタ数指定のコンパイラオプションはグローバルオプションです。
グローバルオプションはプロジェクトを通じて一貫したオプションでなければなりません。
そのため、コンパイラのオプションのみを変更するとエラーとなる場合があります。

グローバルオプションには引数渡し用レジスタ数指定と、CPU 種別の指定があります。
グローバルオプションは次のように変更してください。

例）引数渡し用レジスタ数を変更する場合
(1) コンパイラオプションを変更する

【指定方法】
ダイアログメニュー：**CPU タブ 引数格納レジスタを 2 つから 3 つに変更**
コマンドライン　　：*regparam=3*

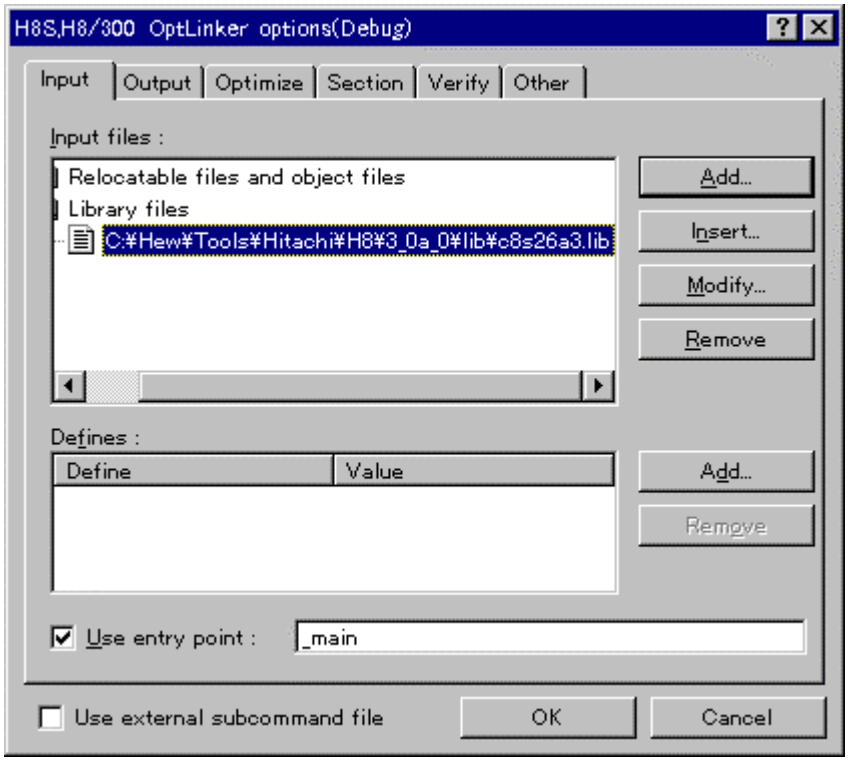
該当するすべてのC/C++ファイルのオプションを変更します。

(2) アセンブラファイルを変更する
C/C++プログラムとアセンブラファイルのリンケージにおいて、使用されるレジスタが変ります。
そのため、アセンブラファイルを変更する必要があります。
引数渡し用レジスタ数を3つにすると次のように変更となります。

CPUがH8/300の場合：R0，R1 → R0，R1，R2
上記以外のCPU　　：ER0，ER1 → ER0，ER1，ER2

これらのインタフェースの詳細については H8S、H8/300 **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**の「9.3.2(3) **レジスタに関する規則」**を参照してください。
また、これらのインタフェースは C/C++プログラム中に埋め込んだアセンブラ部分についても同様です。

（3） リンクする標準ライブラリ/EC++クラスライブラリを変更する
モジュール間最適化ツールでリンクするライブラリを変更してください。
HEW1.2では次のように変更します。
すでにリンクされているライブラリがc8s26a.libの場合は、c8s26a3.libに変更します。



グローバルオプションと関連するライブラリについてはH8S,H8/300**シリーズ** C/C++**コンパイラ ユーザーズマニュアル**の**表**1.1 **標準ライブラリとコンパイルオプションの関係**を参照してください。

HEW2.0以降でのグローバルオプションは、コンパイラ、アセンブラ、ライブラリ生成ツールに共通に設定されますので最適化リンケージエディタでは特に設定の必要はありません。
（**Lik/Library**タブ**Category:[Input] Library files**にて旧バージョンのライブラリを設定している場合は、上記のような変更が必要です）

11.1.30 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

```
int i1;
void f( int *d)
{
    int a;
```

```
        do
        {
            a=*d;
        }while(a!=0);
        i1 = a;
    }
```

最適化ありアセンブラソース

```
    _f:                                         ; function: f
                .STACK      _f=4
                MOV.W       @ER0,R1
    L25:
                MOV.W       R1,R1           ; メモリからリードされない
                BNE         L25:8
                MOV.W       R1,@_i1:32
                RTS
```

変更後 C ソース

```
    int i1;
    void f( volatile int *d)
    {
        int a;
        do
        {
            a=*d;
        }while(a!=0);
        i1 = a;
    }
```

変更後最適化ありアセンブラソース

```
    _f:                                         ; function: f
                .STACK      _f=4
                MOV.L       ER0,ER1
    L25:
                MOV.W       @ER1,R0             ; メモリからリードされる
                BNE         L25:8
                MOV.W       R0,@_i1:32
                RTS
```

11.1.31 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限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。このアクセスのためにコンパイラ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。__evenaccess キーワード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等に関して、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 __evenaccess キーワードを指定してください。

__evenaccess なし C ソース

```
struct bit reg;

void main()
{
    reg.b6=1;
}
```

__evenaccess なしアセンブラソース

```
  _main:                                    ; function: main
            .STACK      _main=4
            BSET.B      #1,@_reg:32
            RTS
```

__evenaccess あり C ソース

```
__evenaccess struct bit reg;

void main()
{
    reg.b6=1;
}
```

__evenaccess ありアセンブラソース

```
  _main:                                    ; function: main
            .STACK      _main=4
            MOV.W       @_reg:32,R0
            BSET.B      #1,R0H
            MOV.W       R0,@_reg:32
            RTS
```

■備考

__evenaccess キーワードの詳細は**「3.5.3　偶数バイトアクセス指定機能」**を参照してください。

11.1.32 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事が考えられます。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事が考えられます。

11.1.33 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 型のグローバル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。
[20 * 2000]を[15 * 2000]に変更すると正しい値を得ることができます。
long 型変数への代入なのに、乗算結果が short の範囲を超えると正しくならないのはなぜでしょう？
<例>

```
long l_max;
        :
    l_max = 20 * 2000;
```

■回答

代入する変数が long 型でも、演算する整数を定数で記述すると int 型（2 バイト）で扱われます。そのため、乗算をした段階で
[20 * 2000]は 0x9C40 ですが、long 型に代入する際に符号拡張が起こり 0xFFFF9C40 となってしまいます。
[15 * 2000]の場合は 0x7530 となるので符号拡張が起こらないため、0x00007530 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 ‘ L ’ を記述し、定数を意図的に long 型とコンパイラに認識させる必要があります。
<例>

```
long l_max;
        :
    l_max = 20L * 2000L;      // 定数の値の後ろに L を付加、片方の定数のみでも OK
```

11.2 最適化リンケージエディタ

11.2.1 Undefined external symbol が出力される

■質問

モジュール間最適化ツールで“ Undefined external symbol(XXX) ”というメッセージが出力されます。しかし、当該シンボル XXX はプログラム中では使用していません。

■回答

（1） コンパイラで提供している標準ライブラリ、またはEC++クラスライブラリをリンクしていますか？
標準ライブラリには、Cライブラリ関数と、実行時ルーチン（Cプログラム実行時に必要な演算ルーチン群）が含まれています。
ユーザプログラムでCライブラリ関数を使用しなくても、コンパイラが生成するオブジェクトプログラムで標準ライブラリ中の関数を必要とすることがありますので、モジュール間最適化ツールの*library*オプションで標準ライブラリを指定してください。

なお、HEW1.2の場合指定する標準ライブラリ・EC++クラスライブラリは、CPU種別、最適化方法、引数渡しレジスタ数により、選択しなければなりません。
次の表のようになっています。

CPU／動作モード	引数渡し	標準 C ライブラリ		EC++クラスライブラリ	
	レジスタ数	サイズ優先	スピード優先	サイズ優先	スピード優先
H8/300	2	c38reg.lib	c38regs.lib	ec2reg.lib	ec2regs.lib
	3	c38reg3.lib	c38regs3.lib	ec2reg3.lib	ec2regs3.lib
H8/300H NRM	2	c38hn.lib	c38hns.lib	ec2hn.lib	ec2hns.lib
	3	c38hn3.lib	c38hns3.lib	ec2hn3.lib	ec2hns3.lib
H8/300H ADV	2	c38ha.lib	c38has.lib	ec2ha.lib	ec2has.lib
	3	c38ha3.lib	c38has3.lib	ec2ha3.lib	ec2has3.lib
H8S NRM	2	c8s26n.lib	c8s26ns.lib	ec226n.lib	ec226ns.lib
	3	c8s26n3.lib	c8s26ns3.lib	ec226n3.lib	ec226ns3.lib
H8S ADV	2	c8s26a.lib	c8s26as.lib	ec226a.lib	ec226as.lib
	3	c8s26a3.lib	c8s26as3.lib	ec226a3.lib	ec226as3.lib

【記号説明】

NRM:**ノーマルモード**、ADV:**アドバンストモード**

サイズ優先、スピード優先はコンパイラのオプションにかかわらず指定することができます。しかし、CPU 種別、引数渡しレジスタ数については、コンパイラの指定と同じにしなければなりません。

HEW2.0 以降の場合 **Standard Library タブ Category:[Mode]**の **Build a library file(Option Changed)**を選択して標準ライブラリを作成しているか確認してください。

（標準ライブラリタブカテゴリ:[モード] ライブラリファイル作成(オプション変更時)）入出力、メモリ管理ライブラリを指定していることが考えられます。C ライブラリ関数 stdio.h、stdlib.h で宣言された関数を指定する場合は、低水準インタフェースルーチンが必要です。H8S、H8/300 **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「**9.2.2 **実行環境の設定」**を参照して、低水準インタフェースルーチンを作成してください。

また、サンプルプログラムの中に低水準インタフェースルーチンの例がありますので、参照して作成してください。

低水準インタフェースルーチンとしては、次のものがあります。

名称	機能
open	ファイルのオープン
close	ファイルのクローズ
read	ファイルからの読み込み
write	ファイルへの書き出し
lseek	ファイルの読み込み／書き出しの位置の設定
sbrk	メモリ領域の確保

11.2.2 Relocation size overflow が出力される

■質問

HEW2.0 以降の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でリンク時に“ Relocation size overflow ”というメッセージが出力されます。回避策はありますか？

■回答

まずはリンケージマップを確認してください。

- $ABS8 セクション、$ABS16 セクションが当該 CPU の 8 ビット絶対アドレス、16 ビット絶対アドレスでアクセス可能な範囲におさまっているか
- $INDIRECT セクションが当該 CPU の 0～FF におさまっているか

オプションまたは、#pragma 演算子によって 8 ビット絶対アドレス、16 ビット絶対アドレス、メモリ間接アドレスに割り付けることを指定したデータが正しいアドレスに割り付けられていない場合、このウォーニングメッセージが出力されます。

これらの範囲は各製品のプログラミングマニュアルに載っています。範囲を確認して、範囲外に割り付けられている場合は調整してください。

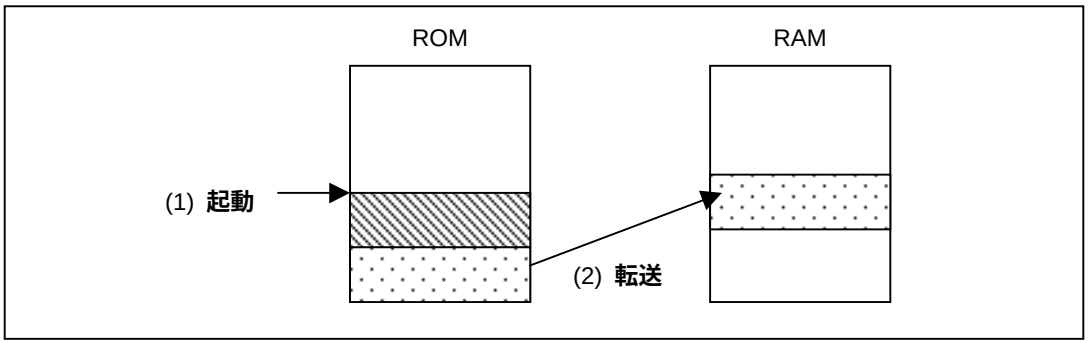
次にアセンブラ埋め込み箇所のアセンブラ記述が正しいか確認してください。

C/C++プログラムにアセンブラ埋め込みを行った場合、分岐幅などがあわないためにこのメッセージが出力される場合があります。

11.2.3 RAM 上でプログラムを実行したい

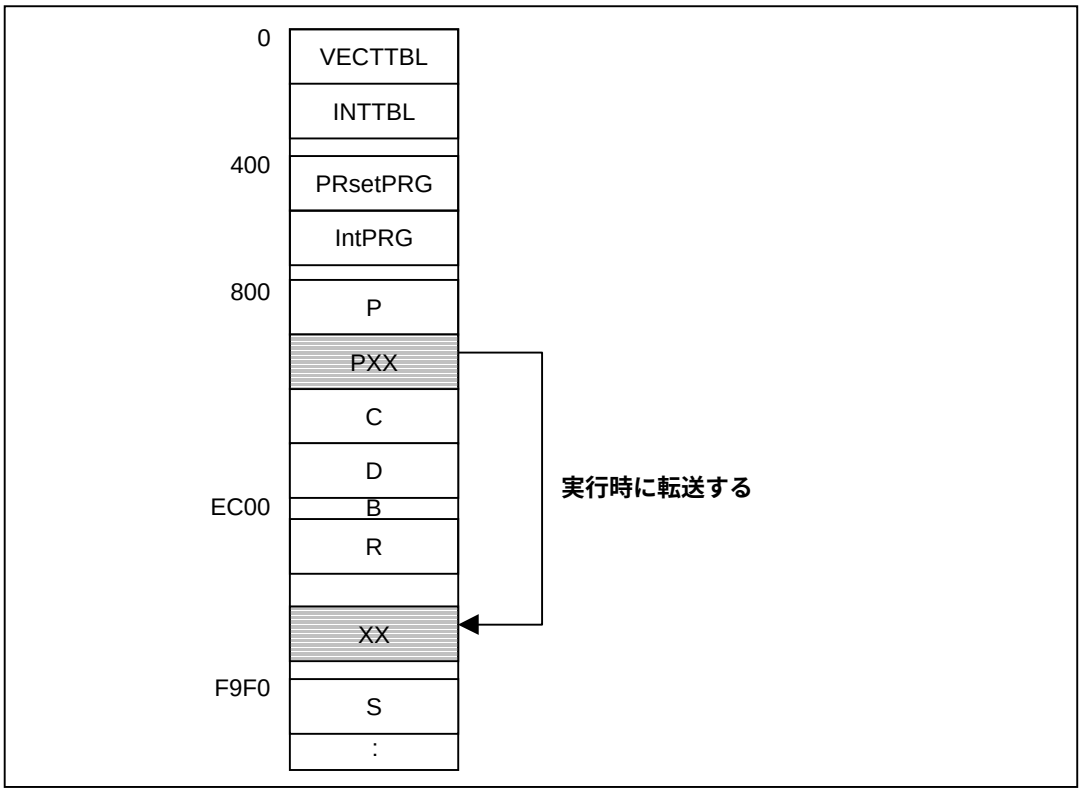
■質問

プログラムを実行速度の速い RAM におきたいがどうすればよいのですか？



■回答

HEW2.0 以降では最適化リンケージエディタ、HEW1.2 ではモジュール間最適化ツールの 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）にプログラムの一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。



2 章で作成したプロジェクトに次のように組み込みます。

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。この処理は既存のファイルに追加します。

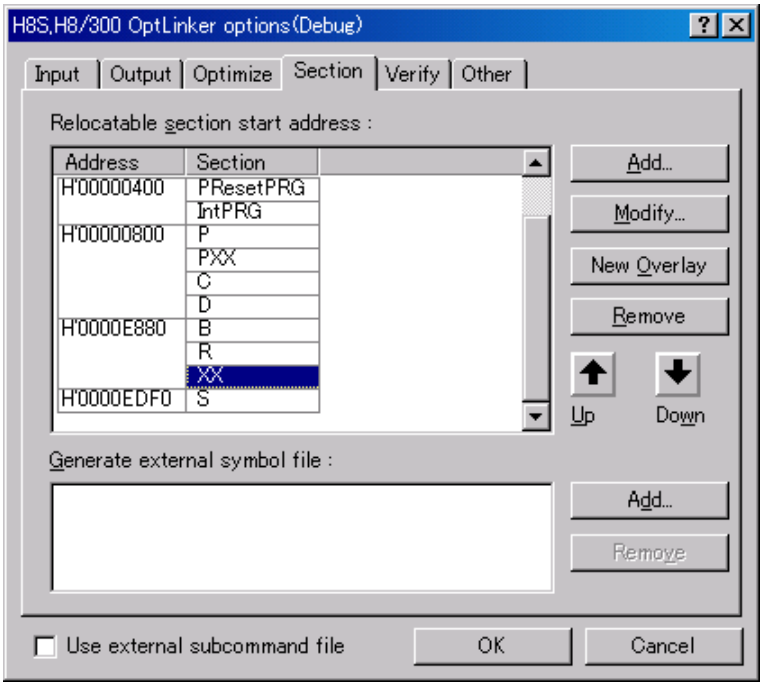
```
#pragma section $DSEC
static const struct {
    char *rom_s;      /* Start address of the initialized data section in ROM */
    char *rom_e;      /* End address of the initialized data section in ROM   */
    char *ram_s;      /* Start address of the initialized data section in RAM */
}DTBL[]= {
    {__sectop("D"), __secend("D"), __sectop("R")},
//  {__sectop("$ABS8D"), __secend("$ABS8D"), __sectop("$ABS8R")},
//  {__sectop("$ABS16D") , __secend("$ABS16D") , __sectop("$ABS16R") },
    {__sectop("PXX"),__secend("PXX"),__sectop("XX")}
};

#pragma section $BSEC
static const struct {
    char *b_s;        /* Start address of non-initialized data section */
    char *b_e;        /* End address of non-initialized data section */
}BTBL[]= {
    {__sectop("B"), __secend("B")},
//  {__sectop("$ABS8B"), __secend("$ABS8B")},
//  {__sectop("$ABS16B"), __secend("$ABS16B")}
};
```

↑PXXセクションおよびXXセクションのセクションアドレスを設定

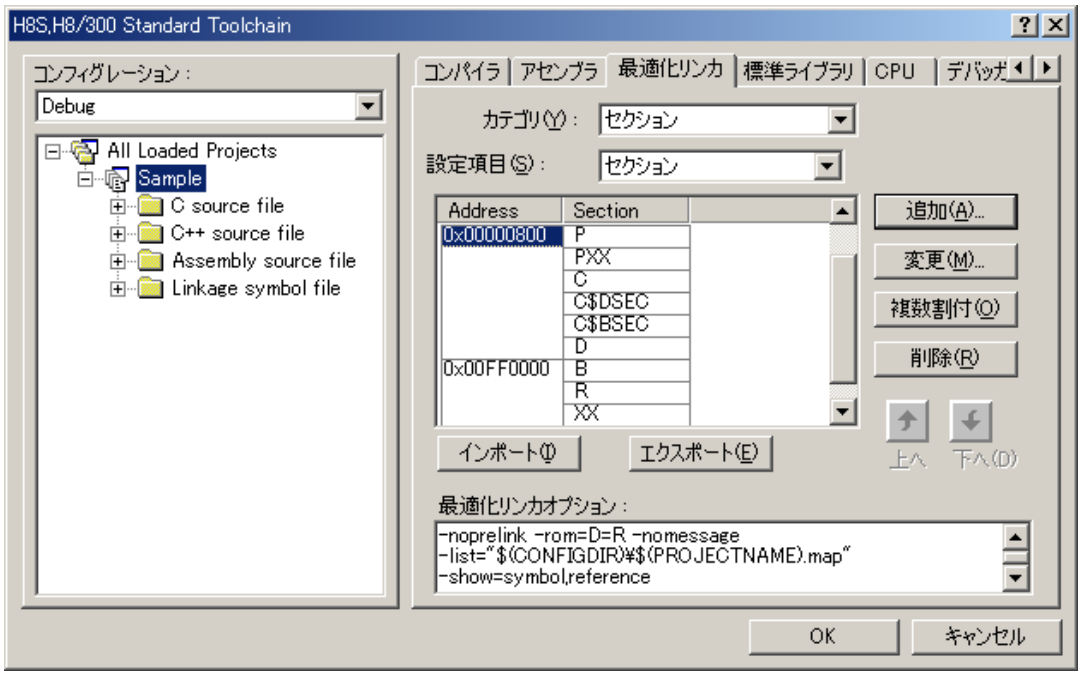
この処理をすると、起動時に PXX セクションから XX セクションへのコピーを行います。

その後、最適化リンケージエディタ、モジュール間最適化ツールで転送先セクション XX の先頭アドレスを指定します。

＜HEW1.2＞



＜HEW2.0 以降＞

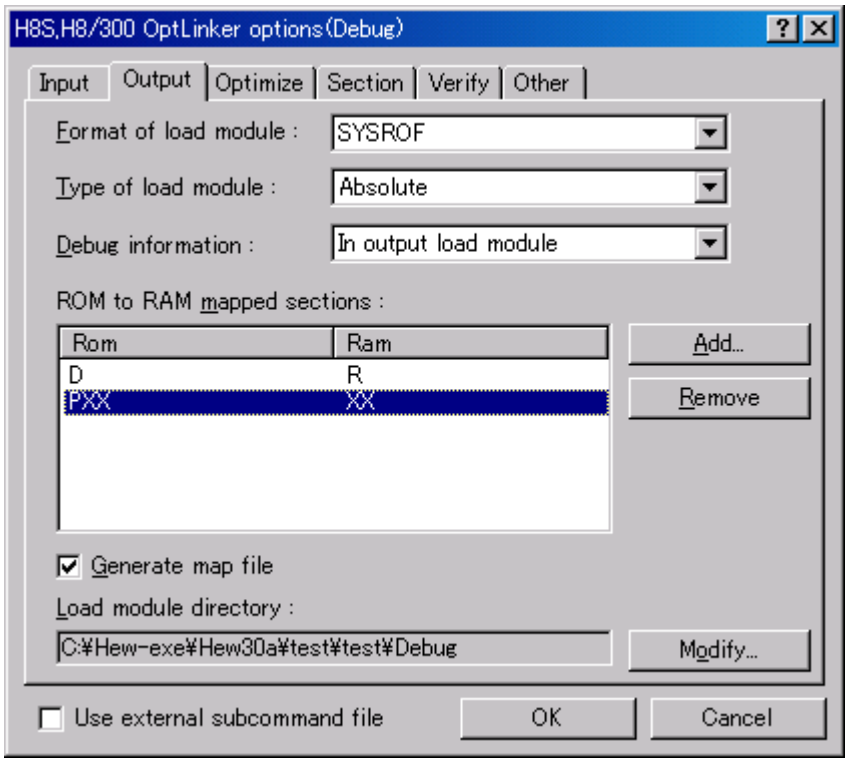


その後に、ROM 化支援オプションを使用して、転送元の PXX セクションと同じ領域を RAM 上に割り付けられるように指定します。ROM 指定すると PXX セクションと同じサイズになります。

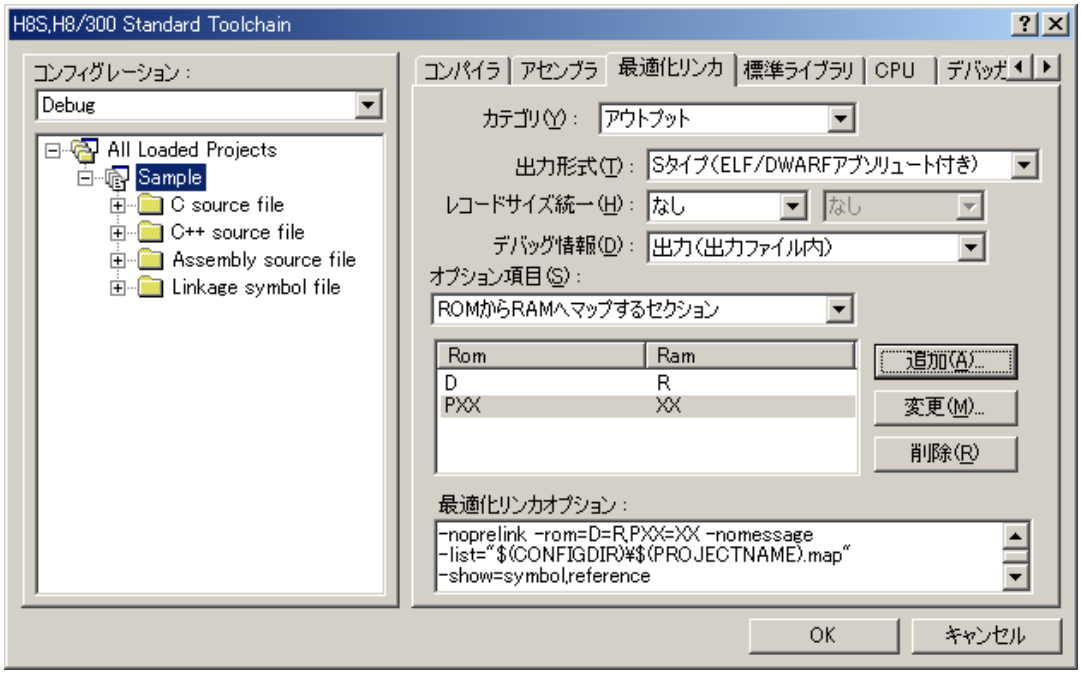
この最適化リンケージエディタ、モジュール間最適化ツールの動作をサブコマンドに記入すると次のようになります。

```
        :
start VECTTBL,INTTBL(0),PRsetPRG,IntPRG(0400),P,PXX,C,D(0800),B,R,XX(0EC00),S…
        :
```

＜HEW1.2＞



＜HEW2.0 以降＞



サブコマンドでは rom を用いて次のように指定します。

```
        :
rom(D,R)
rom(PXX,XX)
        :
```

これらを記述したサブコマンドファイルを使用して最適化リンケージエディタ、モジュール間最適化ツールを起動させます。

＜HEW1.2＞

% optlnk38 –sub=test.sub(RET) （モジュール間最適化ツール）

＜HEW2.0 以降＞

% optlnk –sub=test.sub(RET) （最適化リンケージエディタ）

■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。この場合は、問題ありません。

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オプション等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。

11.2.4 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。

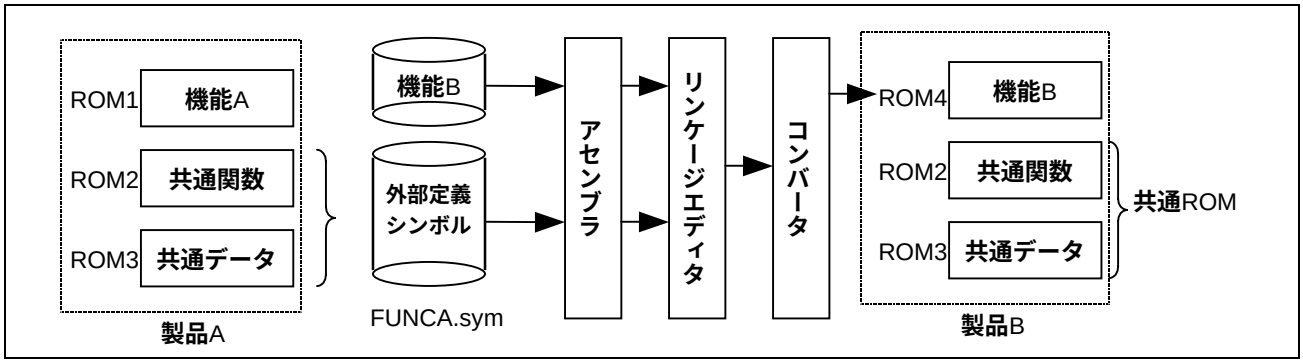
■回答

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

以下は、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。



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】

```
asm38△ROM4
asm38△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。

11.2.5 オーバレイの実現

■質問

同時に存在しないセクションを同一アドレスに配置したい。

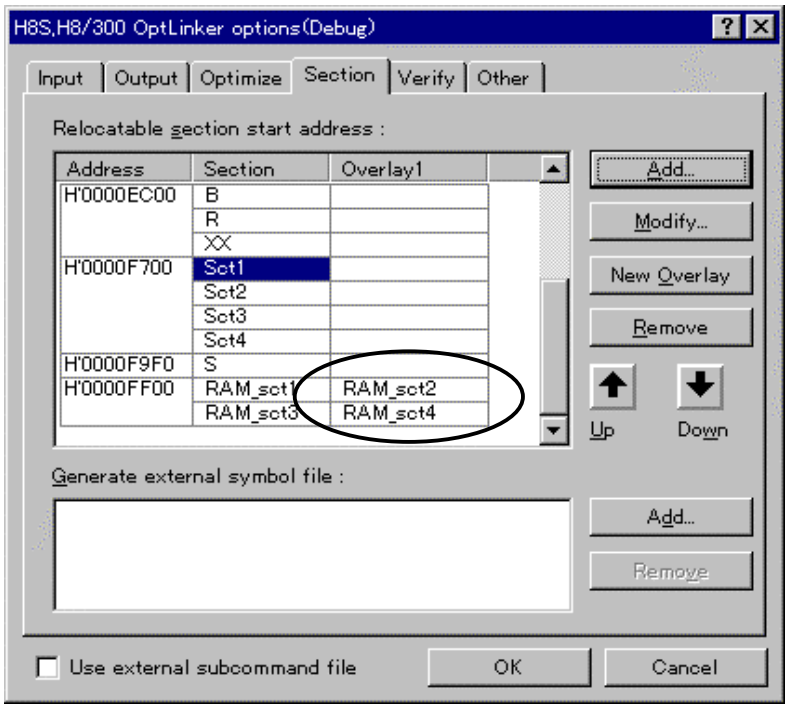
■回答

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オプションで、指定することができます。

■指定方法

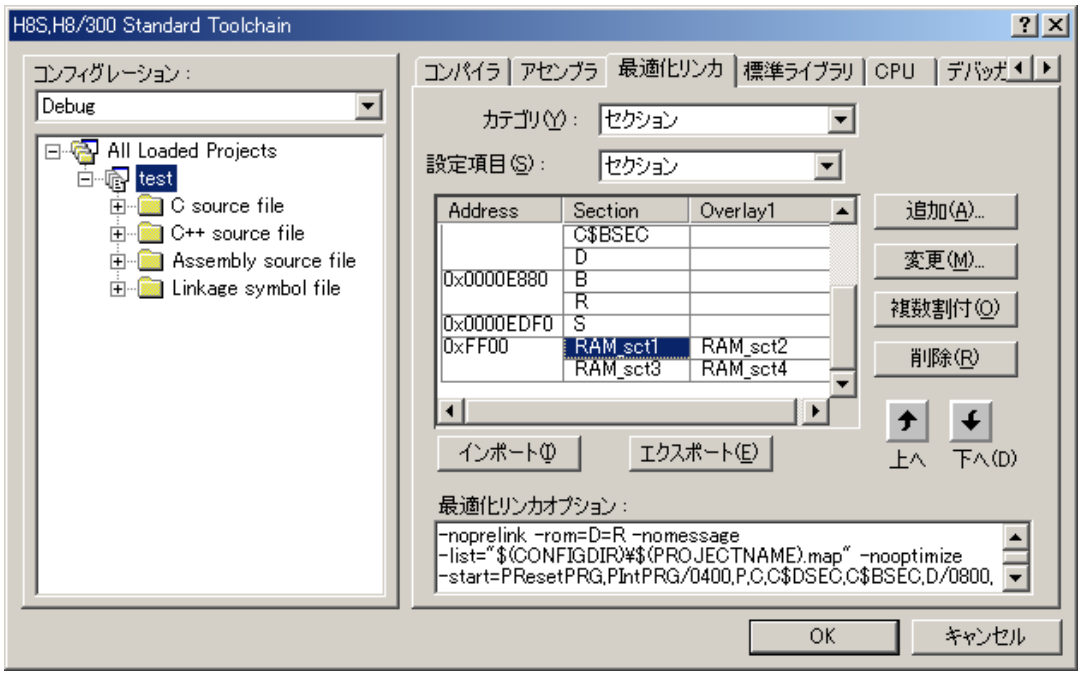
＜HEW1.2＞

Section タブ Relocatable section start address:対象セクションにカーソルを送って New Overlay を指定します。
画面では次のようになります。



＜HEW2.0 以降＞

最適化リンカタブカテゴリ:[セクション] (Lik/Library タブ Category:[Section]) 対象セクションにカーソルを送って複数割り付け(New Overlay)を指定します。



サブコマンドの場合：start オプションで指定します。

```
        :
start RAM_Sct1,RAM_Sct3:RAM_Sct2,RAM_Sct4(0FF00)
        :
```

■備考および注意事項

オーバレイを指定したいセクションがプログラムセクションの場合には転送が必要です。
「11.2.3　RAM **上でプログラムを実行したい」**を参照してください。

11.2.6　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

HEW1.2 のモジュール間最適化ツールでは、オプションで未定義シンボルをチェックすることができます。
これにより、未定義シンボルが含まれる場合は、エラーメッセージが出力され、ロードモジュール出力が抑止されます。
この指定がない場合は、ウォーニングメッセージが出力されますが、ロードモジュールは生成されます。

■指定方法

ダイアログメニュー：**Link/Library+タブ Category:[Other] Miscellaneous options:**の **Check for undefined symbols**
サブコマンド　　：*udfcheck*

■備考

HEW2.0 以降の最適化リンケージエディタでは、未定義シンボルのチェックを常に行っており未定義シンボルが含まれる場合は、エラーメッセージを出力し、ロードモジュール出力が抑止されます。

11.2.7　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。

11.2.8　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。

11.2.9 最適化リンケージエディタが出力するファイル形式

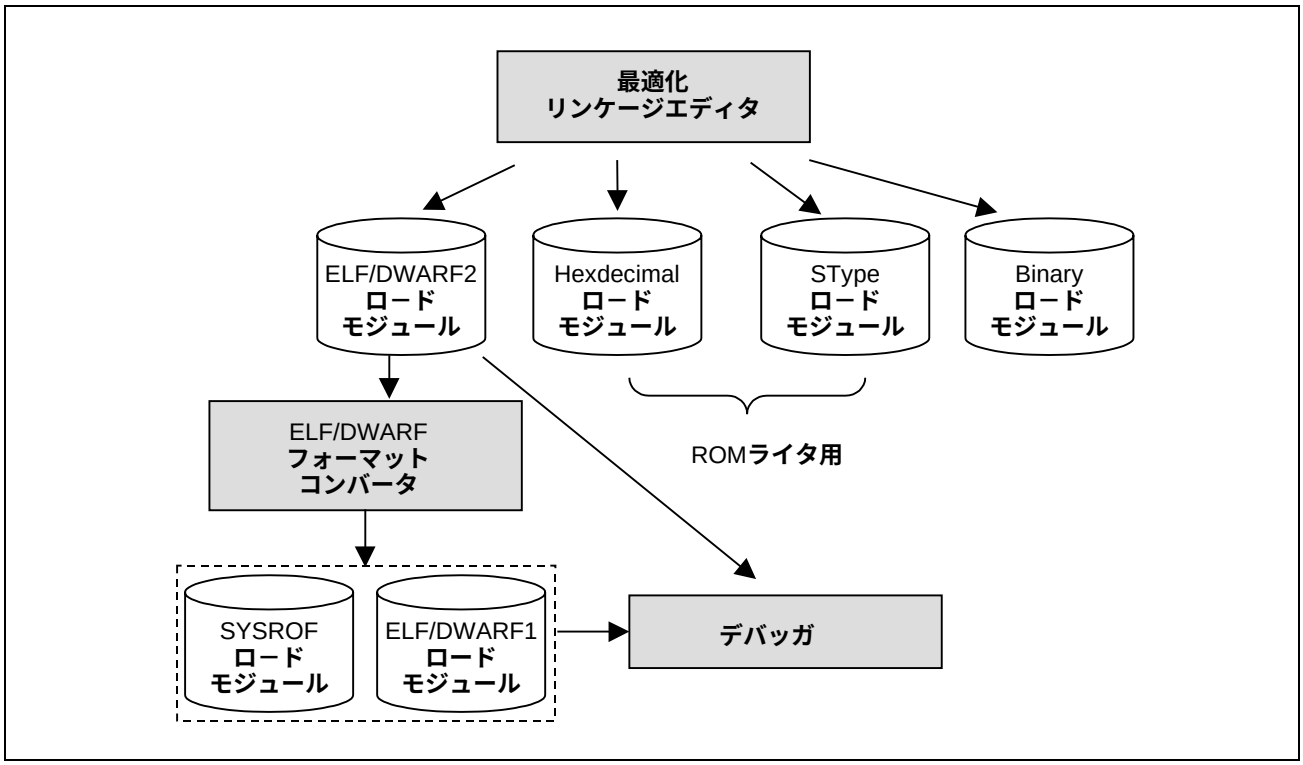
■質問

ROM ライタ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。

- ROM ライタ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 V4.0 以降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 フォーマットコンバータを利用しフォーマットを変更してください。



最適化リンケージエディタ出力ロードモジュール

11.2.10 プログラムサイズ（ROM,RAM）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 PResetPRG,PIntPRG,P,C$DSEC,C$BSEC,D なので ROM サイズは 0x00000146 となります。
RAM 容量は B,R,S なので 0x00000628 となります。

〈リストファイル例〉

```
*** Mapping List ***

SECTION                    START     END        SIZE   ALIGN
PResetPRG
                          00000400  00000415       16   2
PIntPRG
                          00000416  0000048f       7a   2
P
                          00000800  0000089d       9e   2
C$DSEC
                          0000089e  000008a9        c   2
C$BSEC
                          000008aa  000008b1        8   2
D
                          000008b2  000008b5        4   2
B
                          00ffe000  00ffe423      424   2
R
                          00ffe424  00ffe427        4   2
S
                          00ffedc0  00ffefbf      200   2
```

11.2.11 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良いのでしょうか？

[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");

  dumy(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、該当のセクションに関するサイズが 0 で境界調整数が 2 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。

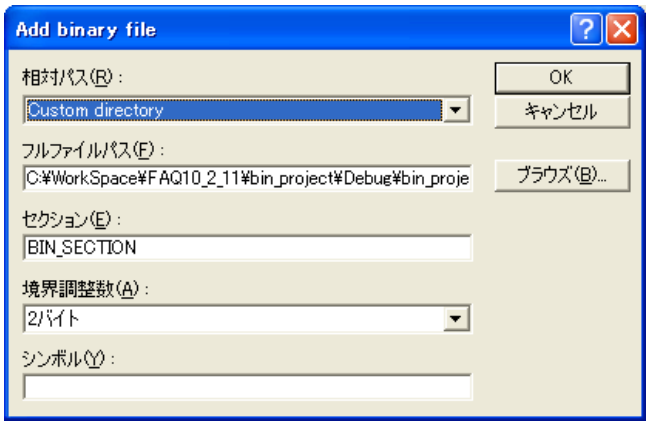
本 Warning メッセージは、最適化リンカでバイナリファイルを入力する際、境界調整数を指定することで回避できます。

[__sectop 使用箇所のコード]

```
_main:                                  ; function: main
        .STACK      _main=4
        MOV.L       #STARTOF BIN_SECTION,ER0
        BRA         _dumy:8
        .SECTION    BIN_SECTION,DATA,ALIGN=2        ←サイズが 0 で境界調整数が 2 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=project.bin(BIN_SECTION:2)`

■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 標準ライブラリ構築ツール

11.3.1 リエントラントと標準ライブラリ

■質問

標準ライブラリを使用している場合、リエントラントなオブジェクトプログラムを作成することはできますか。

■回答

外部変数を設定／参照しているライブラリ関数を使用すると、リエントラントではなくなります。表にリエントラントライブラリ一覧を表示します。△は_errno を設定する関数です。これらの関数はプログラム中で_errno を参照しなければリエントラントに実行できます。

リエントラントライブラリー覧

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	No.	関数名	リエントラント	備考
1	stddef.h	1	offsetof	○	マクロ
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	○	
4	math.h	16	acos	△	浮動小数点
		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	△	〃
		31	modf	△	〃
		32	pow	△	〃
		33	sqrt	△	〃
		34	ceil	△	〃
		35	fabs	△	〃
		36	floor	△	〃
		37	fmod	△	〃

No.	標準インクルードファイル	No.	関数名	リエントラント	備考
5	mathf.h	38	acosf	△	浮動小数点
		39	asinf	△	〃
		40	atanf	△	〃
		41	atan2f	△	〃
		42	cosf	△	〃
		43	sinf	△	〃
		44	tanf	△	〃
		45	coshf	△	〃
		46	sinhf	△	〃
		47	tanhf	△	〃
		48	expf	△	〃
		49	frexpf	△	〃
		50	ldexpf	△	〃
		51	logf	△	〃
		52	log10f	△	〃
		53	modff	△	〃
		54	powf	△	〃
		55	sqrtf	△	〃
		56	ceilf	△	〃
		57	fabsf	△	〃
		58	floorf	△	〃
		59	fmodf	△	〃
6	setjmp.h	60	setjmp	○	
		61	longjmp	○	
7	stdarg.h	62	va_start	○	マクロ
		63	va_arg	○	〃
		64	va_end	○	〃
8	stdio.h	65	fclose	×	
		66	fflush	×	
		67	fopen	×	
		68	freopen	×	
		69	setbuf	×	
		70	setvbuf	×	
		71	fprintf	×	
		72	fscanf	×	
		73	printf	×	
		74	scanf	×	
		75	sprintf	△	
		76	sscanf	△	
		77	vfprintf	×	
		78	vprintf	×	
		79	vsprintf	△	
		80	fgetc	×	
		81	fgets	×	
		82	fputc	×	
		83	fputs	×	
		84	getc	×	
		85	getchar	×	
		86	gets	×	
		87	putc	×	
		88	putchar	×	

No.	標準インクルードファイル	No.	関数名	リエントラント	備考
8	stdio.h	89	puts	×	
		90	ungetc	×	
		91	fread	×	
		92	fwrite	×	
		93	fseek	×	
		94	ftell	×	
		95	rewind	×	
		96	clearerr	×	
		97	feof	×	
		98	ferror	×	
		99	perror	×	
9	stdlib.h	100	atof	△	非 ANSI
		101	atoi	△	〃
		102	atol	△	〃
		103	strtod	△	
		104	strtol	△	
		105	rand	×	浮動小数点
		106	srand	×	
		107	calloc	×	
		108	free	×	
		109	malloc	×	
		110	realloc	×	
		111	bsearch	○	
		112	qsort	○	再帰的関数
		113	abs	○	
		114	div	△	
		115	labs	○	
		116	ldiv	△	
10	string.h	117	memcpy	○	
		118	strcpy	○	
		119	strncpy	○	
		120	strcat	○	
		121	strncat	○	
		122	memcmp	○	
		123	strcmp	○	
		124	strncmp	○	
		125	memchr	○	
		126	strchr	○	
		127	strcspn	○	
		128	strpbrx	○	
		129	strrchr	○	
		130	strspn	○	
		131	strstr	○	
		132	strtok	×	
		133	memset	○	
		134	strerror	○	
		135	strlen	○	
		136	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

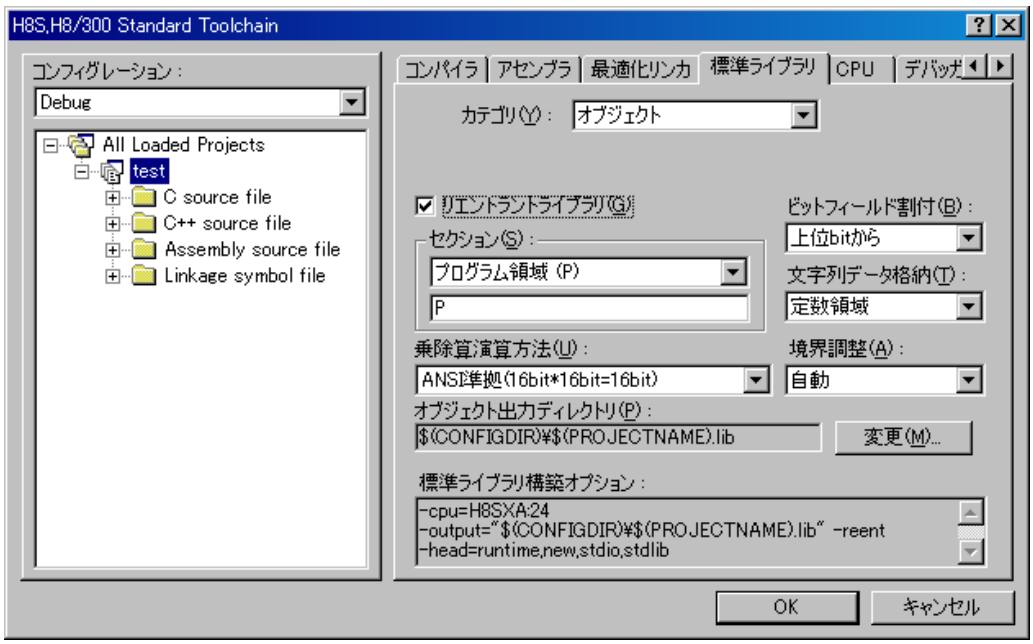
標準ライブラリで、リエントラントなライブラリを使用したい。

■回答

11.3.1 にリエントラントライブラリ一覧があります。なお、H8C V6.0 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 lbg38　-reent オプションを指定してください。

また、HEW での設定方法は以下のとおりです。



11.3.3 標準ライブラリが存在しない（H8C V4 以降）

■質問

H8C V3 で付属されていた標準ライブラリがありません。（H8C V4 以降）

■回答

H8C V4 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。H8C V4 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。

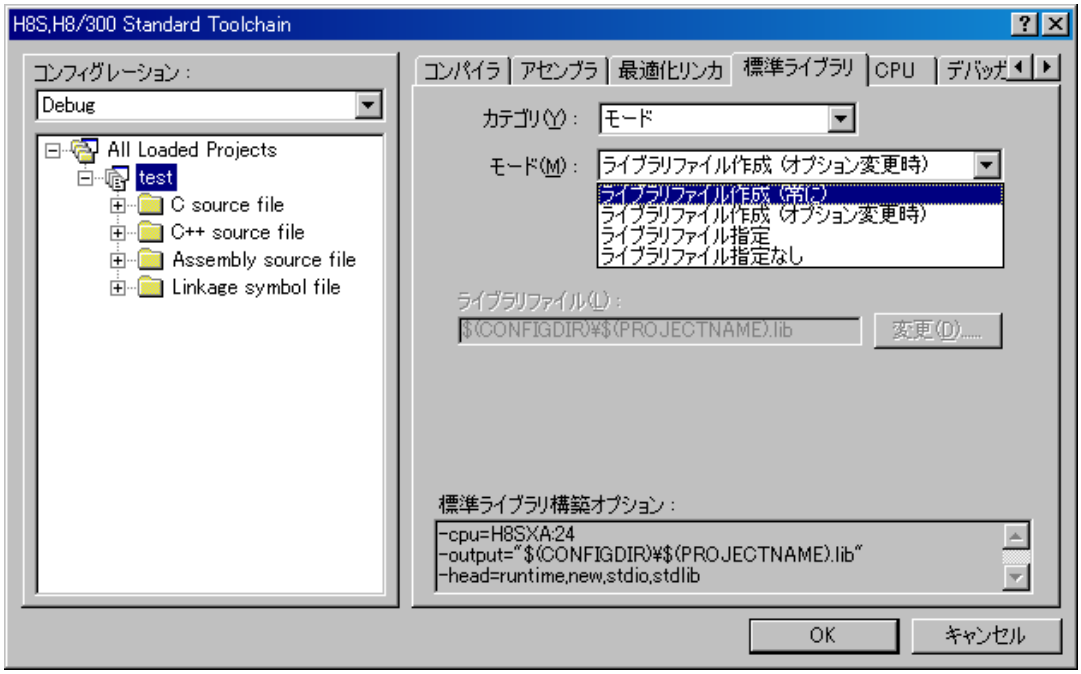
■回答

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]／[H8S,H8 Standard Toolchain...]の[標準ライブラリ]モード：で“ ライブラリファイル指定 ”を選択すればワ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]/[H8S,H8 Standard Toolchain...]の[標準ライブラリ]モード：で「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が自動生成されるため、既存のライブラリのバックアップを取るという操作をしてしまっています。

“ ライブラリファイル指定 ”（既存のライブラリを使う）を選択すればワ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。

標準ライブラリ ダイアログボックス



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。

11.3.6 入出力用ライブラリの ROM サイズを減らす方法

■質問

標準インクルードファイルの入出力用ライブラリの ROM サイズを減らしたい。

■回答

no_float.h インクルードファイルを指定すると、浮動小数点変換をしない簡易入出力関数を使用することができます。
対象となる関数は、以下のようになっています。

fprintf、fscanf、printf、scanf、sprintf、sscanf、vfprintf、vprintf、vsprintf

使用方法は、標準入出力用ファイルの stdio.h の前に、no_float.h をインクルード指定します。

例）

```
#include <no_float.h>        ←マクロ宣言する

#include <stdio.h>
void main(void)
{
    printf("HELLO¥n");
}
```

または、すでにある標準入出力ライブラリを使用したファイルの場合は、*preinclude* オプションで指定することができます。

簡易入出力関数にすると、ファイル入出力を行う場合、ROM サイズを削減できます。
しかし、この指定をした上で、浮動小数点数（%f、%e、%E、%g、%E）を指定した場合、実行時の動作は保証しません。

11.3.7 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、方法がわかりません。

■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。以下に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。

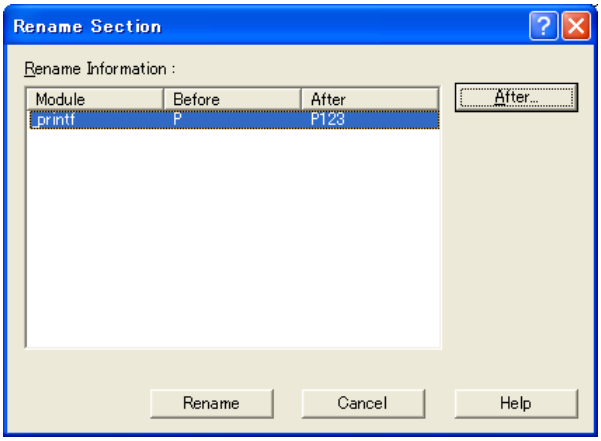
■H Series Librarian Interface 起動方法

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） 該当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。



【コマンドラインの場合】
optlnk –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） 該当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 –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） 該当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 –lib=<ライブラリファイル名> -delete=<ライブラリ内モジュール名>

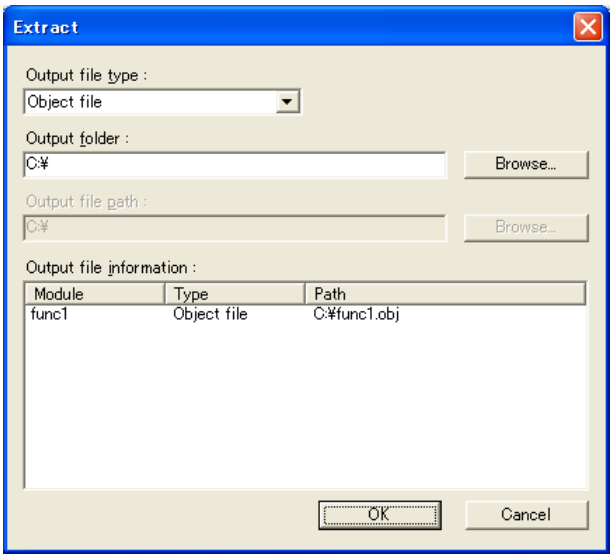
(D) ライブラリ内モジュールの抽出

ライブラリのモジュールを抽出することができます。

(1) 該当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:¥)



【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

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。

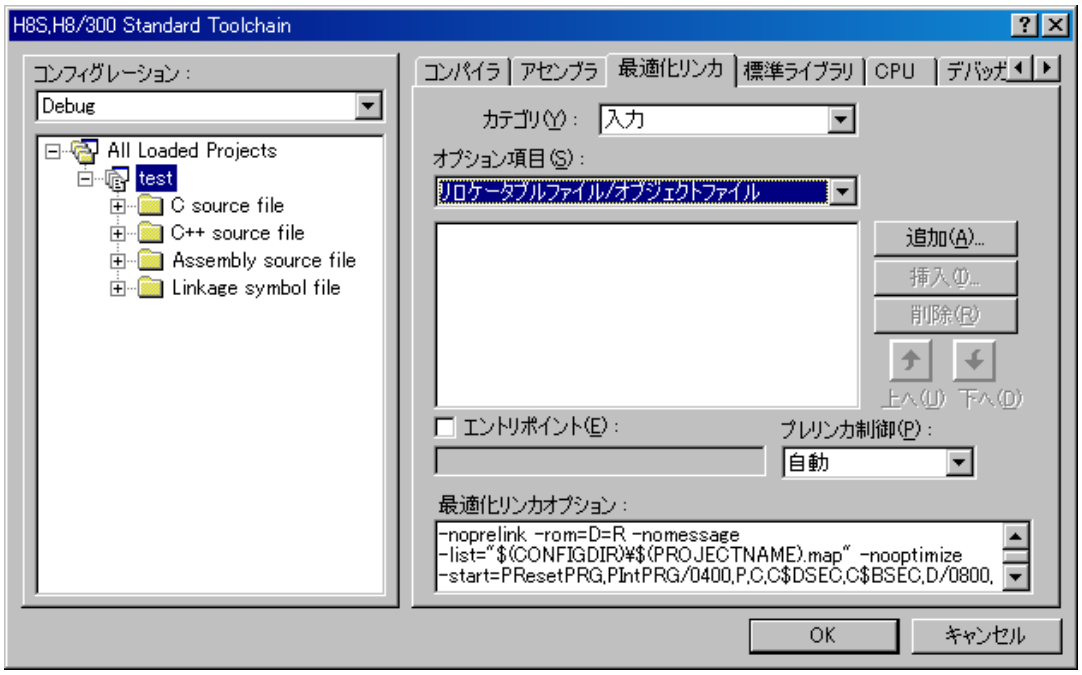
11.4.2 オブジェクトファイルのリンク順序

■質問

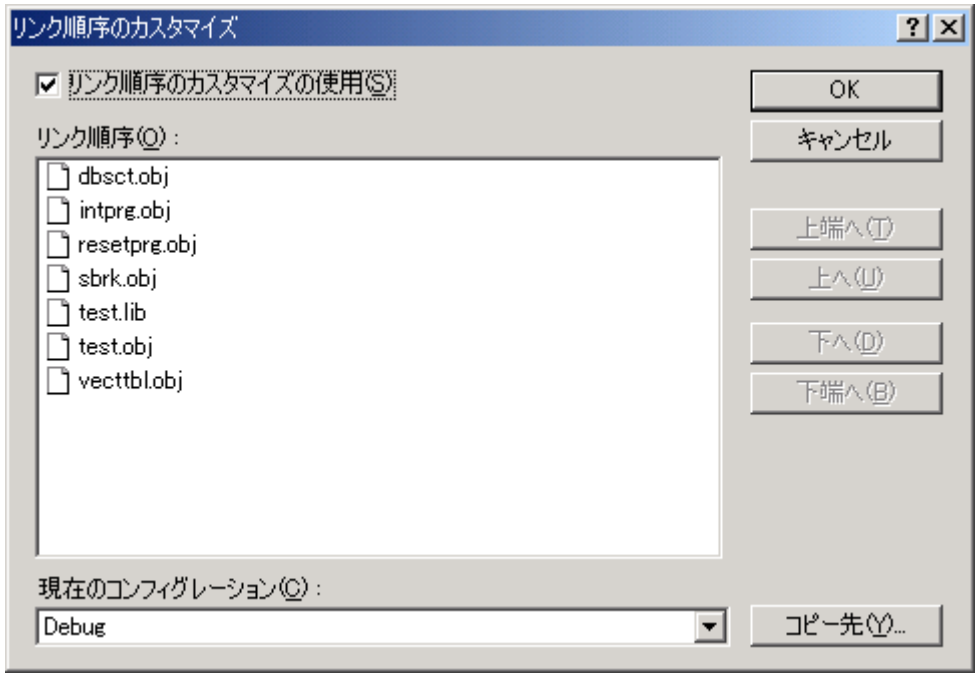
HEW 上で、オブジェクトファイルのリンク順序を指定したい。

■回答

H8S,H8 Standard Toolchain...の最適化リンカタブのカテゴリ:[入力]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、Add 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。



H8C V.6.00Release02 以降では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。


リンク順序のカスタマイズ
リンク順序のカスタマイズの使用(S)
リンク順序(O):
dbsct.obj
intprg.obj
resetprg.obj
sbrk.obj
test.lib
test.obj
vecttbl.obj
現在のコンフィグレーション(C):
Debug
OK
キャンセル
上端へ(T)
上へ(U)
下へ(D)
下端へ(B)
コピー先(Y)...

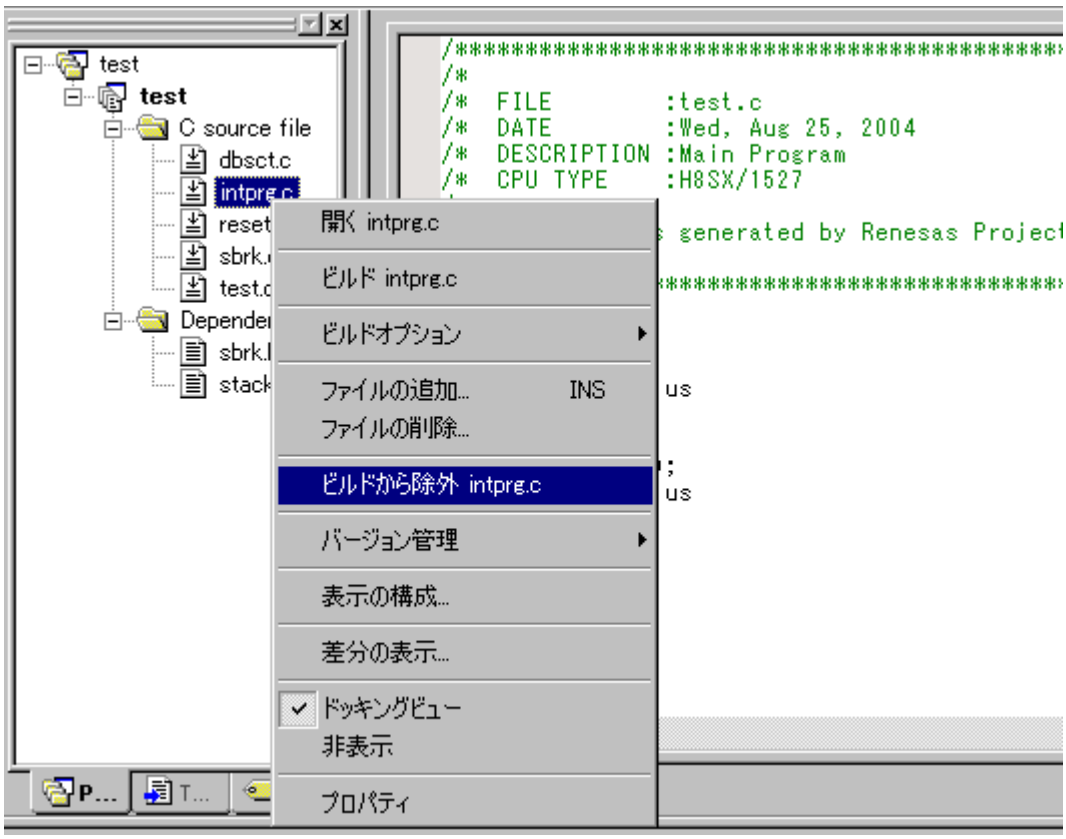
11.4.3 プロジェクトファイルの除外

■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。



ビルドから除外メニュー

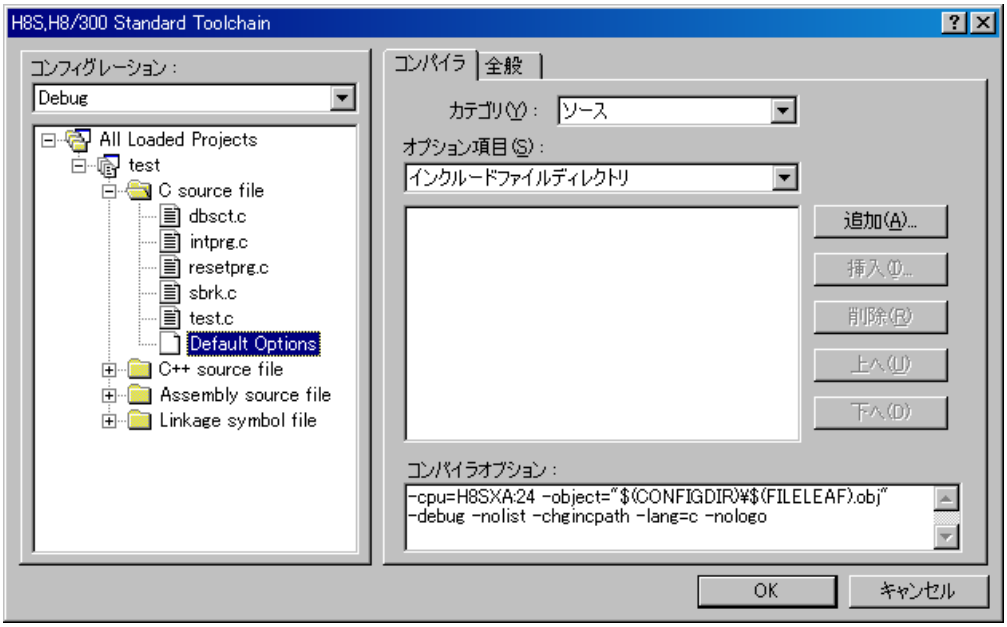
11.4.4 プロジェクトファイルのデフォルトオプション指定

■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

H8S,H8 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（下図）。 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。



デフォルトオプション

11.4.5 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。

11.4.6 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ください。

11.4.7 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

11.4.8 HEW エディタ、HDI での日本語表示フォントがおかしい

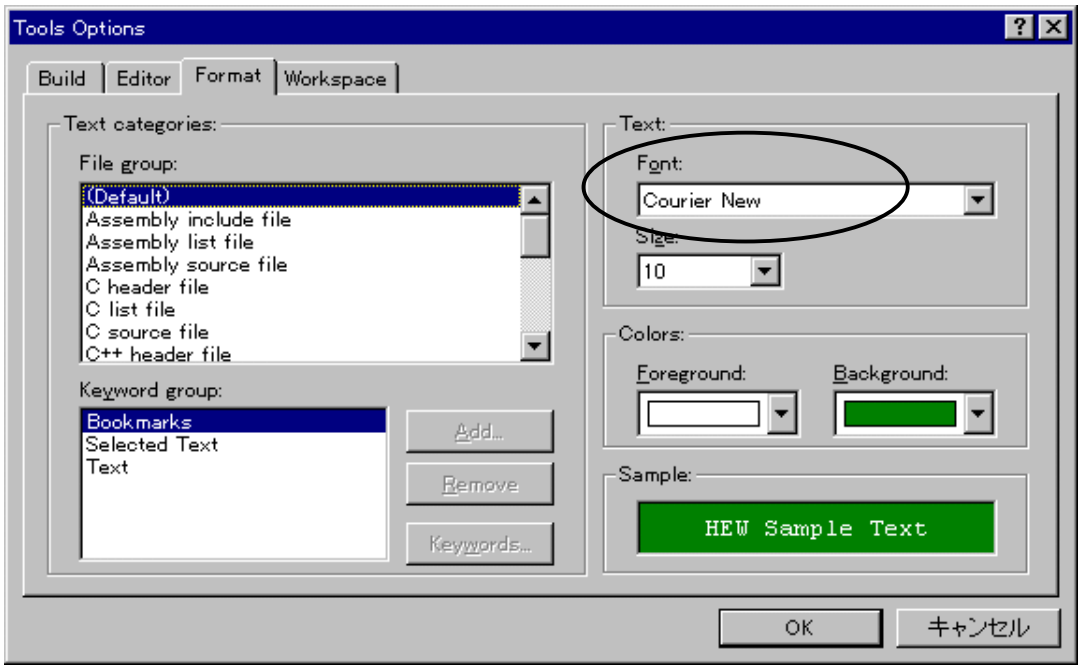
■質問

（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。

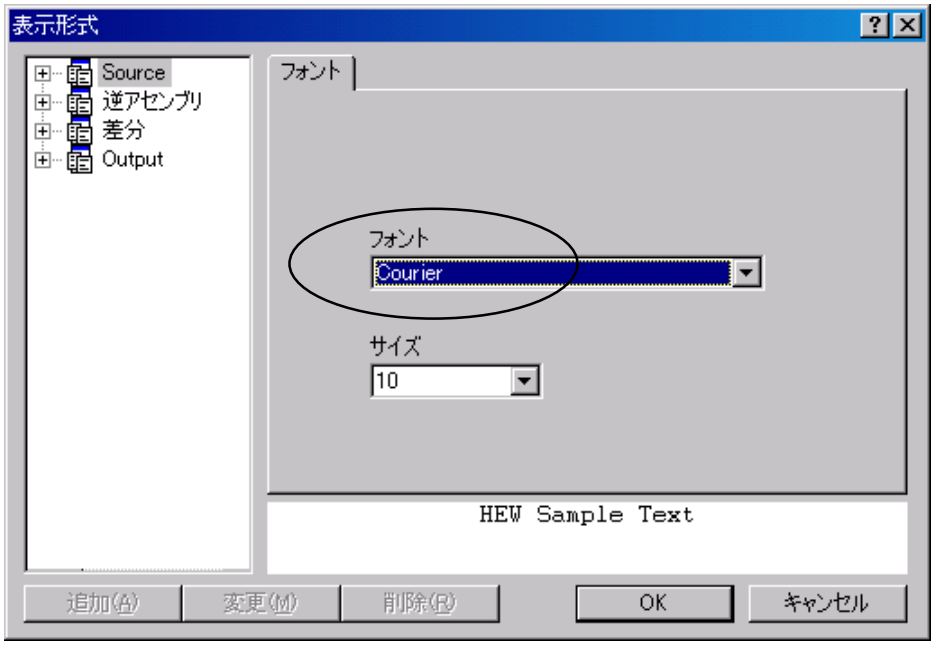
■回答

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。

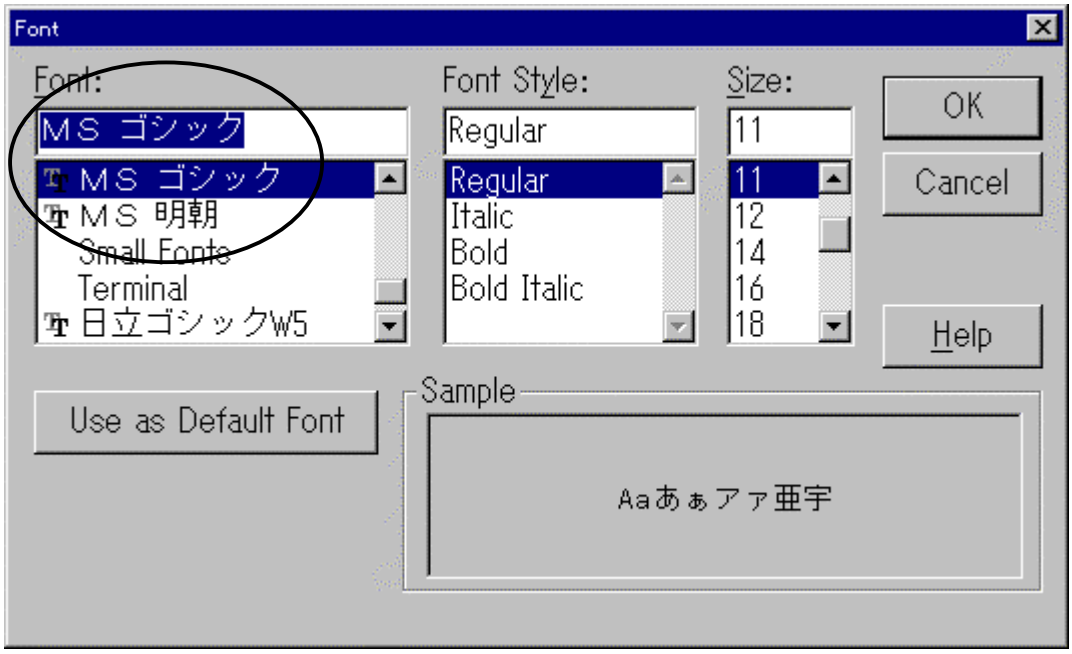
＜HEW1.2＞
Tools->Options の **Format** タブの **Text** 欄の **Font** で変更します。



＜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。



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。



11.4.9 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。

11.4.10 HEW のプロジェクト構築時に該当デバイスがない

■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枝にありません。

■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 H8SX コアの H8S/1527 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 H8SX、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 1520g.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 1520g.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。

11.4.11 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？

■回答

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。以下をご参照ください。

[H8C V.3.0]
＜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）
H8C V.3.0A 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 H8C V.3.0A 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。

[H8C V.3.0A]
＜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。

[H8C V.4]
＜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。

[H8C V.5 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

A. 浮動小数点演算の性能一覧

A.1 単精度の浮動小数点演算性能

A.1.1 単精度の浮動小数点演算性能（H8/300,H8/300H,H8S/2600）

No.	関数名	パラメータ 1	パラメータ 2	H8/300	H8/300H		H8S/2000,H8S/2600	
					NRM	ADV	NRM	ADV
1	acos	0.4	-	30,850	23,316	25,636	8,495	8,852
		1.57075	-	3,830	3,022	3,484	930	999
		0.6	-	30,864	23,226	25,546	8,450	8,806
		-0.4	-	30,918	23,302	25,622	8,496	8,853
2	asin	0.4	-	29,840	22,390	24,576	8,158	8,494
		1.57075	-	2,904	2,194	2,522	642	690
		0.6	-	29,850	22,310	24,496	8,118	8,453
		-0.4	-	29,926	22,402	24,588	8,172	8,508
3	atan	0.11	-	13,166	9,948	11,010	3,581	3,767
		0.27	-	18,122	14,302	15,718	5,269	5,502
		0.547	-	17,964	14,128	15,544	5,179	5,412
		0.777	-	18,890	14,820	16,270	5,436	5,672
		0.975	-	17,924	14,210	15,582	5,277	5,502
		54.45	-	21,834	17,744	19,390	6,659	6,922
		154.233	-	21,952	17,840	19,486	6,707	6,970
		-54.45	-	21,920	17,754	19,400	6,672	6,935
		-0.975	-	18,010	14,220	15,592	5,290	5,515
		-0.777	-	18,976	14,830	16,280	5,449	5,685
4	atan2	0.3	0.7	20,898	16,758	18,494	6,182	6,441
		0.2	0.1	24,736	20,204	22,214	7,523	7,820
		0.1	0.9	16,156	12,648	14,030	4,619	4,831
5	cos	0.523333333	-	11,124	8,148	8,780	3,114	3,262
		1.046666667	-	13,090	9,610	10,506	3,588	3,757
		1.9625	-	12,420	9,024	9,842	3,404	3,562
		2.7475	-	12,074	8,932	9,642	3,398	3,557
		3.5325	-	11,332	8,284	8,916	3,183	3,331
		4.3175	-	13,184	9,748	10,644	3,656	3,825
		5.1025	-	12,462	9,114	9,932	3,448	3,606
		5.8875	-	12,050	8,960	9,670	3,411	3,570
		-0.52333333	-	11,210	8,158	8,790	3,127	3,275
		-1.04666667	-	13,176	9,620	10,516	3,601	3,770
		-1.9625	-	12,506	9,034	9,852	3,417	3,575
		-2.7475	-	12,160	8,942	9,652	3,411	3,570
		-3.5325	-	11,418	8,294	8,926	3,196	3,344
		-4.3175	-	13,270	9,758	10,654	3,669	3,838
		-5.1025	-	12,548	9,124	9,942	3,461	3,619
		-5.8875	-	12,136	8,970	9,680	3,424	3,583

No.	関数名	パラメータ 1	パラメータ 2	H8/300	H8/300H		H8S/2000,H8S/2600	
					NRM	ADV	NRM	ADV
6	sin	0.523333333	-	12,170	8,838	9,656	3,314	3,472
		1.046666667	-	11,942	8,872	9,582	3,373	3,532
		1.9625	-	11,196	8,202	8,834	3,147	3,295
		2.7475	-	13,240	9,764	10,660	3,671	3,840
		3.5325	-	12,482	9,060	9,878	3,428	3,586
		4.3175	-	12,120	8,954	9,664	3,415	3,574
		5.1025	-	11,382	8,312	8,944	3,203	3,351
		5.8875	-	13,204	9,776	10,672	3,674	3,843
		-0.52333333	-	12,348	8,876	9,694	3,342	3,500
		-1.04666667	-	12,120	8,910	9,620	3,401	3,560
		-1.9625	-	11,374	8,240	8,872	3,175	3,323
		-2.7475	-	13,418	9,802	10,698	3,699	3,868
		-3.5325	-	12,660	9,098	9,916	3,456	3,614
		-4.3175	-	12,298	8,992	9,702	3,443	3,602
		-5.1025	-	11,560	8,350	8,982	3,231	3,379
		-5.8875	-	13,382	9,814	10,710	3,702	3,871
7	tan	0.3925	-	16,682	12,494	13,374	4,768	4,997
		1.1775	-	17,522	13,240	14,198	5,055	5,276
		1.9625	-	16,908	12,634	13,514	4,863	5,074
		2.7475	-	17,696	13,344	14,302	5,111	5,332
8	cosh	0.33	-	44,886	33,624	35,796	13,237	13,735
		0.78	-	46,018	34,462	36,646	13,354	13,864
		-0.33	-	44,904	33,636	35,808	13,243	13,741
		-0.78	-	46,036	34,474	36,658	13,360	13,870
9	sinh	0.33	-	12,538	9,004	9,660	3,375	3,520
		0.98	-	47,040	35,310	37,568	13,689	14,209
		-0.33	-	12,538	9,004	9,660	3,375	3,520
		-0.98	-	47,058	35,322	37,580	13,695	14,215
10	tanh	0.0033+00	-	9,772	7,102	7,710	2,553	2,672
11	exp	0.33	-	21,860	16,184	17,180	6,471	6,713
		0.98	-	22,588	16,740	17,742	6,598	6,846
		-0.33	-	21,980	16,212	17,208	6,485	6,727
		-0.98	-	22,684	16,732	17,734	6,594	6,842
12	frexp	0.3	-	186	102	118	54	60
		400	-	186	102	118	54	60
13	ldexp	0.3	30	1,382	964	1,068	316	337
		0.1	100	1,382	964	1,068	316	337
14	log	1.2	-	18,766	14,272	15,410	5,353	5,575
		2.5	-	18,882	14,476	15,614	5,455	5,677
		0.999	-	19,376	14,996	16,134	5,715	5,937
		0.3	-	19,016	14,604	15,742	5,519	5,741
15	log10	1.2	-	20,138	15,260	16,532	5,686	5,929
		2.5	-	20,254	15,464	16,736	5,788	6,031
		0.999	-	20,764	16,008	17,280	6,060	6,303
		0.3	-	20,372	15,572	16,844	5,482	6,085
16	modf	256.3	-	3,518	2,890	3,388	914	975
		0.032	-	3,342	2,760	3,252	850	908
		10000.2345	-	3,608	2,962	3,460	950	1,011

No.	関数名	パラメータ 1	パラメータ 2	H8/300	H8/300H		H8S/2000,H8S/2600	
					NRM	ADV	NRM	ADV
17	pow	2.3	4.2	43,236	33,010	35,340	12,577	13,074
		45.2	-5	43,642	33,412	35,742	12,789	13,286
		-4.56	-3	47,134	36,326	39,066	13,678	14,231
		-85.55	476	45,988	35,406	38,064	13,360	13,904
18	sqrt	2	-	4,918	1,878	1,980	829	852
		3	-	4,966	1,910	2,012	845	868
		0.1	-	4,906	1,890	1,992	835	858
19	ceil	0.3	-	2,998	2,452	2,790	749	801
		-0.6	-	1,806	1,314	1,502	393	426
20	fabs	5	-	126	38	40	24	27
		-5	-	126	38	40	24	27
21	floor	0.3	-	1,806	1,314	1,502	393	426
		-0.6	-	2,998	2,446	2,784	746	798
22	fmod	11.1	3.2	1,964	1,498	1,654	533	564
		500.55	0.4	2,436	1,858	2,014	713	744
		1.05E+06	9.54E-07	4,178	3,186	3,342	1,377	1,408

A.1.2 単精度の浮動小数点演算性能（H8SX）

No.	関数名	パラメータ 1	パラメータ 2	H8SX		
				NRM	ADV	MAX
1	acos	0.4	-	6,108	6,403	6,397
		1.57075	-	495	438	438
		0.6	-	6,079	6,373	6,368
		-0.4	-	6,100	6,396	6,390
2	asin	0.4	-	5,880	6,219	6,220
		1.57075	-	340	309	309
		0.6	-	5,885	6,195	6,196
		-0.4	-	5,884	6,225	6,226
3	atan	0.11	-	2,167	2,550	2,549
		0.27	-	3,542	4,012	4,011
		0.547	-	3,449	3,919	3,918
		0.777	-	3,643	4,122	4,121
		0.975	-	3,595	4,055	4,054
		54.45	-	4,769	5,308	5,307
		154.233	-	4,828	5,368	5,367
		-54.45	-	4,773	5,314	5,313
		-0.975	-	3,599	4,061	4,060
		-0.777	-	3,647	4,128	4,127
4	atan2	0.3	0.7	4,370	4,811	4,806
		0.2	0.1	5,570	6,088	6,085
		0.1	0.9	3,146	3,497	3,493
5	cos	0.523333333	-	2,039	2,347	2,349
		1.046666667	-	2,229	2,658	2,660
		1.9625	-	2,208	2,543	2,547
		2.7475	-	2,222	2,552	2,556
		3.5325	-	2,201	2,408	2,411
		4.3175	-	2,371	2,732	2,737
		5.1025	-	2,256	2,593	2,597
		5.8875	-	2,240	2,572	2,576
		-0.52333333	-	2,045	2,351	2,353
		-1.04666667	-	2,305	2,662	2,664
		-1.9625	-	2,214	2,547	2,551
		-2.7475	-	2,228	2,556	2,560
		-3.5325	-	2,108	2,412	2,415
		-4.3175	-	2,377	2,736	2,741
		-5.1025	-	2,262	2,597	2,601
		-5.8875	-	2,246	2,576	2,580
6	sin	0.523333333	-	2,385	2,459	2,460
		1.046666667	-	2,464	2,537	2,539
		1.9625	-	2,308	2,377	2,380
		2.7475	-	2,650	2,728	2,733
		3.5325	-	2,497	2,571	2,575
		4.3175	-	2,501	2,574	2,578
		5.1025	-	2,361	2,430	2,433
		5.8875	-	2,663	2,741	2,746
		-0.52333333	-	2,397	2,468	2,469

No.	関数名	パラメータ 1	パラメータ 2	H8SX		
				NRM	ADV	MAX
6	sin	-1.04666667	-	2,476	2,546	2,548
		-1.9625	-	2,320	2,386	2,389
		-2.7475	-	2,662	2,737	2,742
		-3.5325	-	2,509	2,580	2,584
		-4.3175	-	2,513	2,583	2,587
		-5.1025	-	2,373	2,439	2,442
		-5.8875	-	2,675	2,750	2,755
7	tan	0.3925	-	3,366	3,775	3,771
		1.1775	-	3,566	4,000	3,996
		1.9625	-	3,448	3,854	3,850
		2.7475	-	3,609	4,040	4,036
8	cosh	0.33	-	10,276	9,214	9,214
		0.78	-	10,294	9,237	9,237
		-0.33	-	10,272	9,219	9,219
		-0.78	-	10,299	9,242	9,242
9	sinh	0.33	-	2,413	2,110	2,110
		0.98	-	10,623	9,548	9,548
		-0.33	-	2,413	2,110	2,110
		-0.98	-	10,628	9,553	9,553
10	tanh	0.0033+00	-	1,604	1,553	1,552
11	exp	0.33	-	5,110	4,564	4,556
		0.98	-	5,215	4,667	4,663
		-0.33	-	5,116	4,570	4,562
		-0.98	-	5,221	4,673	4,669
12	frexp	0.3	-	41	42	42
		400	-	41	42	42
13	ldexp	0.3	30	220	196	196
		0.1	100	220	196	196
14	log	1.2	-	3,706	3,573	3,573
		2.5	-	3,770	3,636	3,636
		0.999	-	4,058	3,924	3,924
		0.3	-	3,858	3,724	3,724
15	log10	1.2	-	3,884	3,754	3,755
		2.5	-	3,948	3,818	3,818
		0.999	-	4,245	4,115	4,115
		0.3	-	4,029	3,889	3,899
16	modf	256.3	-	535	514	514
		0.032	-	469	450	450
		10000.2345	-	571	550	550
17	pow	2.3	4.2	9,338	8,626	8,634
		45.2	-5	9,492	8,780	8,795
		-4.56	-3	10,016	9,242	9,254
		-85.55	476	9,783	9,033	9,053
18	sqrt	2	-	885	859	859
		3	-	893	867	867
		0.1	-	889	863	863
19	ceil	0.3	-	446	390	390
		-0.6	-	246	215	215

No.	関数名	パラメータ 1	パラメータ 2	H8SX		
				NRM	ADV	MAX
20	fabs	5	-	21	21	21
		-5	-	21	21	21
21	floor	0.3	-	246	215	215
		-0.6	-	445	393	393
22	fmod	11.1	3.2	367	413	415
		500.55	0.4	581	627	629
		1.05E+06	9.54E-07	1,388	1,434	1,436

A.2 倍精度の浮動小数点演算性能

A.2.1 倍精度の浮動小数点演算性能（H8/300,H8/300H,H8S/2600）

No.	関数名	パラメータ 1	パラメータ 2	H8/300	H8/300H		H8S/2000,H8S/2600	
					NRM	ADV	NRM	ADV
1	acos	0.4	-	88,070	44,762	47,294	20,277	21,016
		1.57075	-	4,646	3,786	4,284	1,814	1,994
		0.6	-	88,396	44,974	47,506	20,383	21,121
		-0.4	-	88,114	44,730	47,262	20,269	21,008
2	asin	0.4	-	86,796	43,666	46,062	19,681	20,369
		1.57075	-	3,542	2,834	3,196	1,290	1,419
		0.6	-	87,104	43,862	46,258	19,779	20,466
		-0.4	-	86,882	43,678	46,074	19,695	20,383
3	atan	0.11	-	29,172	18,570	19,780	7,784	8,156
		0.27	-	41,948	25,560	27,142	11,126	11,590
		0.547	-	41,590	25,512	27,094	11,099	11,563
		0.777	-	43,906	26,862	28,484	11,640	12,114
		0.975	-	41,862	25,714	27,250	11,218	11,669
		54.45	-	53,282	30,720	32,546	13,589	14,113
		154.233	-	53,626	31,070	32,896	13,764	14,288
		-54.45	-	53,368	30,730	32,556	13,602	14,126
		-0.975	-	41,948	25,724	27,260	11,231	11,682
		-0.777	-	43,992	26,872	28,494	11,653	12,127
4	atan2	0.3	0.7	51,604	29,210	31,122	12,919	13,457
		0.2	0.1	62,958	34,532	36,734	15,451	16,062
		0.1	0.9	39,414	22,708	24,248	9,824	10,270
5	cos	0.523333333	-	24,152	15,346	16,078	6,412	6,681
		1.046666667	-	27,734	17,718	18,730	7,411	7,723
		1.9625	-	26,848	17,014	17,944	7,091	7,382
		2.7475	-	25,478	16,430	17,244	6,922	7,212
		3.5325	-	24,488	15,598	16,330	6,538	6,807
		4.3175	-	27,984	17,876	18,888	7,489	7,801
		5.1025	-	26,982	17,064	17,994	7,115	7,406
		5.8875	-	25,488	16,446	17,260	6,930	7,220
		-0.52333333	-	24,238	15,350	16,082	6,422	6,691
		-1.04666667	-	27,796	17,728	18,740	7,424	7,736
		-1.9625	-	26,934	17,024	17,954	7,104	7,395
		-2.7475	-	25,564	16,440	17,254	6,935	7,225
		-3.5325	-	24,574	15,608	16,340	6,551	6,820
		-4.3175	-	28,070	17,886	18,898	7,502	7,814
		-5.1025	-	27,068	17,074	18,004	7,128	7,419
		-5.8875	-	25,574	16,456	17,270	6,943	7,233
6	sin	0.523333333	-	26,522	16,820	17,750	7,000	7,289
		1.046666667	-	25,278	16,214	17,028	6,821	7,109
		1.9625	-	24,278	15,556	16,288	6,524	6,791
		2.7475	-	27,926	17,840	18,852	7,480	7,790
		3.5325	-	26,960	17,002	17,932	7,093	7,382
		4.3175	-	25,590	16,472	17,286	6,951	7,239
		5.1025	-	24,636	15,712	16,444	6,603	6,870
		5.8875	-	27,988	17,908	18,920	7,512	7,822
		-0.52333333	-	26,700	16,858	17,788	7,028	7,317

No.	関数名	パラメータ 1	パラメータ 2	H8/300	H8/300H		H8S/2000,H8S/2600	
					NRM	ADV	NRM	ADV
6	sin	-1.04666667	-	25,480	16,300	17,114	6,873	7,161
		-1.9625	-	24,456	15,594	16,326	6,552	6,819
		-2.7475	-	28,104	17,878	18,890	7,508	7,818
		-3.5325	-	27,138	17,040	17,970	7,121	7,410
		-4.3175	-	25,768	16,510	17,324	6,979	7,267
		-5.1025	-	24,814	15,750	16,482	6,631	6,898
		-5.8875	-	28,166	17,946	18,958	7,540	7,850
7	tan	0.3925	-	38,230	21,734	22,712	9,149	9,483
		1.1775	-	39,408	22,136	23,196	9,323	9,677
		1.9625	-	38,490	21,456	22,434	9,017	9,357
		2.7475	-	39,672	22,672	23,732	9,595	9,955
8	cosh	0.33	-	99,902	56,136	58,518	23,476	24,258
		0.78	-	101,046	57,590	59,980	24,901	24,693
		-0.33	-	99,920	56,140	58,522	23,478	24,260
		-0.78	-	101,064	57,594	59,984	23,903	24,695
9	sinh	0.33	-	28,064	17,778	18,546	7,269	7,535
		0.98	-	102,482	57,370	59,838	23,765	24,586
		-0.33	-	28,064	17,778	18,546	7,269	7,535
		-0.98	-	102,500	57,374	59,842	23,765	24,588
10	tanh	0.0033+00	-	109,818	63,362	66,024	26,975	27,838
11	exp	0.33	-	49,318	27,448	28,558	11,505	11,886
		0.98	-	50,186	27,746	28,860	11,503	11,889
		-0.33	-	49,428	27,556	28,666	11,559	11,940
		-0.98	-	50,288	27,782	28,896	11,521	11,907
12	frexp	0.3	-	290	246	274	134	147
		400	-	290	246	274	134	147
13	ldexp	0.3	30	1,792	1,436	1,576	659	721
		0.1	100	1,792	1,436	1,576	659	721
14	log	1.2	-	43,214	25,574	26,854	10,931	11,341
		2.5	-	43,360	26,242	27,522	11,265	11,675
		0.999	-	44,000	25,250	26,530	10,769	11,179
		0.3	-	43,580	25,936	27,216	11,112	11,522
15	log10	1.2	-	45,900	27,160	28,580	11,654	12,117
		2.5	-	46,022	27,808	29,228	11,978	12,441
		0.999	-	46,718	26,858	28,278	11,503	11,966
		0.3	-	46,266	27,516	28,936	11,832	12,295
16	modf	256.3	-	4,458	4,044	4,484	1,795	1,945
		0.032	-	4,148	3,712	4,148	1,632	1,780
		10000.2345	-	4,434	3,898	4,338	1,722	1,872
17	pow	2.3	4.2	96,904	56,372	58,948	23,829	24,677
		45.2	-5	97,438	55,556	58,132	23,432	24,280
		-4.56	-3	101,770	59,090	62,090	24,943	25,891
		-85.55	476	100,174	59,292	62,206	25,111	26,039
18	sqrt	2	-	30,274	9,906	10,040	4,940	5,000
		3	-	30,374	9,922	10,056	4,948	5,008
		0.1	-	29,250	9,780	9,914	4,877	4,937
19	ceil	0.3	-	3,720	3,196	3,572	1,451	1,578
		-0.6	-	2,238	1,816	2,034	827	915

No.	関数名	パラメータ 1	パラメータ 2	H8/300	H8/300H		H8S/2000,H8S/2600	
					NRM	ADV	NRM	ADV
20	fabs	5	-	214	166	188	102	112
		-5	-	214	166	188	102	112
21	floor	0.3	-	2,238	1,816	2,034	827	915
		-0.6	-	3,720	3,190	3,566	1,448	1,575
22	fmod	11.1	3.2	2,716	2,070	2,258	1,047	1,127
		500.55	0.4	3,724	2,524	2,712	1,274	1,354
		1.05E+06	9.54E-07	7,624	3,904	4,092	1,964	2,044

A.2.2 倍精度の浮動小数点演算性能（H8SX）

No.	関数名	パラメータ 1	パラメータ 2	H8SX		
				NRM	ADV	MAX
1	acos	0.4	-	16,203	16,305	16,306
		1.57075	-	1,097	1,136	1,135
		0.6	-	16,220	16,323	16,324
		-0.4	-	16,185	16,289	16,291
2	asin	0.4	-	15,881	15,903	15,904
		1.57075	-	738	805	804
		0.6	-	14,895	15,916	15,918
		-0.4	-	14,888	15,910	15,911
3	atan	0.11	-	5,081	5,873	5,872
		0.27	-	8,163	9,066	9,065
		0.547	-	8,089	8,992	8,991
		0.777	-	8,512	9,425	9,424
		0.975	-	8,271	9,159	9,158
		54.45	-	10,528	11,515	11,514
		154.233	-	10,693	11,680	11,679
		-54.45	-	10,534	11,522	11,520
		-0.975	-	8,277	9,166	9,164
		-0.777	-	8,518	9,432	9,430
4	atan2	0.3	0.7	9,791	10,739	10,740
		0.2	0.1	12,161	13,208	13,209
		0.1	0.9	6,978	7,815	7,816
5	cos	0.523333333	-	4,653	4,928	4,927
		1.046666667	-	5,340	5,691	5,691
		1.9625	-	5,147	5,443	5,443
		2.7475	-	5,028	5,355	5,355
		3.5325	-	4,788	5,060	5,060
		4.3175	-	5,418	5,771	5,771
		5.1025	-	5,181	5,479	5,479
		5.8875	-	5,039	5,369	5,368
		-0.52333333	-	4,656	4,931	4,931
		-1.04666667	-	5,341	5,692	5,693
		-1.9625	-	5,151	5,447	5,448
		-2.7475	-	5,032	5,359	5,360
		-3.5325	-	4,792	5,064	5,065
		-4.3175	-	5,422	5,775	5,776
		-5.1025	-	5,185	5,483	5,484
		-5.8875	-	5,043	5,373	5,373
6	sin	0.523333333	-	4,665	5,363	5,362
		1.046666667	-	4,601	5,260	5,260
		1.9625	-	4,405	5,040	5,040
		2.7475	-	5,033	5,754	5,754
		3.5325	-	4,764	5,461	5,461
		4.3175	-	4,725	5,384	5,384
		5.1025	-	4,473	5,108	5,108
		5.8875	-	5,061	5,783	5,782
		-0.52333333	-	4,674	5,372	5,372

No.	関数名	パラメータ 1	パラメータ 2	H8SX		
				NRM	ADV	MAX
6	sin	-1.04666667	-	4,622	5,281	5,282
		-1.9625	-	4,414	5,049	5,050
		-2.7475	-	5,042	5,763	5,764
		-3.5325	-	4,773	5,470	5,471
		-4.3175	-	4,734	5,393	5,394
		-5.1025	-	4,482	5,117	5,118
		-5.8875	-	5,072	5,792	5,792
7	tan	0.3925	-	7,096	7,418	7,418
		1.1775	-	7,284	7,631	7,631
		1.9625	-	7,050	7,317	7,371
		2.7475	-	7,451	7,797	7,797
8	cosh	0.33	-	16,425	16,725	16,727
		0.78	-	16,918	17,216	17,218
		-0.33	-	16,427	16,727	16,729
		-0.78	-	16,920	17,218	17,220
9	sinh	0.33	-	4,793	4,873	4,873
		0.98	-	16,705	17,003	17,006
		-0.33	-	4,793	4,873	4,873
		-0.98	-	16,707	17,005	17,008
10	tanh	0.0033+00	-	21,563	20,209	20,210
11	exp	0.33	-	8,073	8,249	8,248
		0.98	-	8,113	8,289	8,288
		-0.33	-	8,113	8,289	8,288
		-0.98	-	8,129	8,305	8,304
12	frexp	0.3	-	80	75	75
		400	-	80	75	75
13	ldexp	0.3	30	378	413	413
		0.1	100	378	413	413
14	log	1.2	-	8,345	7,889	7,889
		2.5	-	8,640	8,181	8,181
		0.999	-	8,258	7,799	7,799
		0.3	-	8,538	8,079	8,079
15	log10	1.2	-	8,114	8,313	8,316
		2.5	-	8,400	8,599	8,601
		0.999	-	8,035	8,234	8,236
		0.3	-	8,304	8,503	8,505
16	modf	256.3	-	1,226	1,194	1,194
		0.032	-	1,065	1,035	1,035
		10000.2345	-	1,150	1,118	1,118
17	pow	2.3	4.2	17,485	17,294	17,295
		45.2	-5	17,060	16,868	16,870
		-4.56	-3	17,965	17,820	17,820
		-85.55	476	18,237	18,076	18,078
18	sqrt	2	-	3,882	3,912	3,912
		3	-	3,888	3,918	3,918
		0.1	-	3,837	3,867	3,867
19	ceil	0.3	-	892	908	908
		-0.6	-	482	509	509

No.	関数名	パラメータ 1	パラメータ 2	H8SX		
				NRM	ADV	MAX
20	fabs	5	-	51	55	55
		-5	-	51	55	55
21	floor	0.3	-	482	498	498
		-0.6	-	893	894	894
22	fmod	11.1	3.2	688	750	749
		500.55	0.4	921	983	982
		1.05E+06	9.54E-07	1,712	1,774	1,773

B. 追加機能について

B.1 Ver.2.0 から Ver.3.0 への追加機能

B.1.1 Addition of Embedded Extended Functions

H8S,H8/300C/C++コンパイラ Ver.3.0 で追加した機能概要を説明します。

(1) entry 関数機能

#pragma entry によりエントリ関数を指定できるようになりました。エントリ関数は、パワーオンリセット時に最初に実行される関数です。

entry 関数機能を使用することにより、SP（スタックポインタ）の初期設定もアセンブラ埋め込み機能を使用せずに C/C++プログラムで記述できます。

(2) セクションアドレス演算子

セクションの先頭アドレス、最終アドレスを参照する演算子（_ _sectop、_ _secend）を追加しました。

それに伴い、セクション切り替え機能を使用している場合でもセクションの初期設定を行うライブラリ関数（_INITSCT）を使用できるようになりました。

(3) packed 構造体

pack オプションおよび#pragma pack により、構造体メンバの境界調整を指定できるようになりました。

B.1.2 Additional and Improved Functions

(1) C++言語機能

C++プログラムをコンパイルできるようになりました。コンパイラは、C プログラムと C++プログラムを lang オプションまたはファイルの拡張子で区別します。

(2) ライブラリ

数学関数（double 型、float 型）ライブラリを標準サポートしました。

また、組み込み向けクラスライブラリ（ios、istream、ostream、iostream、string、complex、new）をサポートしました。

(3) レジスタパラメタ可変化

regparam オプションにより、パラメタ渡し用レジスタの本数を選択できるようになりました。

(4) speed オプションの強化

speed オプションのサブオプションとして、speed=expression を追加しました。

speed=expression 指定時は、ほとんどの演算を実行時ルーチン呼び出しではなく、インラインに展開します。

(5) long 型ビットフィールドのサポート

long 型データのビットフィールドを追加サポートしました。

(6) 制限値拡張

以下の制限値を拡張しました。

- シンボル長（31 文字→250 文字）
- 複文のネスト（32 レベル→256 レベル）
- 繰り返し文（while 文、do 文、for 文）のネスト（32 レベル→256 レベル）
- 選択文（if 文、switch 文）の組み合わせによる文のネスト（32 レベル→256 レベル）
- switch 文のネスト（16 レベル→128 レベル）
- for 文のネスト（16 レベル→128 レベル）
- 1 行の文字数（4096 文字→8192 文字）
- malloc 確保サイズ（アドバンストモード時：INT_MAX→size_t サイズ）

(7) 最適化リストの出力

モジュール間最適化ツール実行時、シンボルの参照回数や最適化情報リスト出力機能を追加しました。

(8) コマンドラインのリスト出力機能

コマンドライン指定文字列をリスト出力します。

(9) message オプション強化

インフォメーションメッセージレベルの任意の番号を抑止できるようになりました。

(10) **文字コード変換**

Latin1 オプションにより、Latin1 コードをソース上に記述できるようになりました。

B.1.3 Modification of Language Specifications

(1) ***((int*)p)++のウォーニング化**

V2.0 台では、エラーを出力していましたが、V3.0 以降ではウォーニングレベルに変更になりました。

(2) **プロトタイプチェック**

引数型指定なしのプロトタイプ宣言と引数型指定ありのプロトタイプ宣言を同時指定をした場合、V2.0 台ではエラーを出力していましたが、V3.0 以降では正常にコンパイルできるようになりました。

■例

```
void f();
void
f(int);
```
→エラー2118を出力

V2.0

```
void f();
void
f(int);
```
←正常コンパイル

V3.0

(3) **構造体先頭の無名ビットフィールド**

構造体の先頭に無名ビットフィールドを記述できるようになりました。

■例

```
struct S{
  int  :1;
  int  a:1;
};
```
→エラー2141を出力

V2.0

```
struct S{
  int  :1;
  int  a:1;
};
```
→正常コンパイル

V3.0

(4) **構造体初期値時のエラー抑止**

構造体の代入と宣言を同時にできるようになりました。

■例

```
struct S
  int a,b;
}s1;
void test()
{
  struct S s2 = s1;
}
```
←エラー2130を出力

V2.0

```
struct S
  int a,b;
}s1;
void test()
{
  struct S s2 = s1;
}
```
←正常コンパイル

V3.0

(5) static **関数の未定義エラー出力条件変更**

宣言のみで定義のない static 関数に対して、未参照時のエラー出力を抑止します。

■例

```
static void
func();
void test()
{
}
```
←定義がないので無条件にエラー2143出力

V2.0

```
static void
func();
void test()
{
}
```
←参照もないので、エラー出力なし

V3.0

(6) **//コメントのサポート**

C プログラムで//コメントを記述できるようになりました。このため、旧バージョンと解釈が異なる場合があります。

■例

```
int b = a //* コメント */4;
  -a;
  ↓
int b = a /4; -a;  と解釈
V2.0
```

```
int b = a //* コメント */4;
  -a;
  ↓
int b = a -a;  と解釈
V3.0
```

B.2 Ver.3.0 から Ver.4.0 への追加機能

B.2.1 Common Additions and Improvements

H8S,H8/300 C/C++コンパイラ Ver.4.0 で追加した機能概要を説明します。

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：65,535 個→ 無制限
- ソースプログラム行数：C/C++:32,767 行、ASM:65,535 行→ 無制限
- C プログラム行長：8,192 文字→ 16,384 文字
- C プログラム文字列長：512 文字→ 16,384 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

(1) **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。

(2) **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。

(3) **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。

B.2.2 Added and Improved Compiler Functions

(1) **キーワードサポート**

キーワード（_ _interrupt, _ _indirect, _ _entry, _ _abs8, _ _abs16, _ _regsave,_ _noregsave, _ _inline,_ _global_register）を用いて、関数または変数の宣言および定義に対して属性を指定できるようになりました。

(2) **ベクタテーブル生成機能**

#pragma interrupt, indirect, entry および_ _interrupt, _ _indirect, _ _entry の vect 指定を用いて、自動的に関数のベクタテーブルを生成できます。

(3) _ _evenaccess **サポート**

_ _evenaccess で指定した定数、変数の偶数バイトアクセスを保証します。

(4) **レジスタパラメータ指定拡張**

_ _regparam2, _ _regparam3 を用いて、関数毎にレジスタパラメータ数を指定できます。

(5) **関数単位オプションの指定**

#pragma option を用いて、関数単位でオプション指定ができます。

(6) **データの** near **配置サポート**

_ _near8, _ _near16 を用いて、配列、構造体のアドレス計算コードを最適化することができます。
ただし、ポインタサイズは変わりません。

(7) **スタックの** near **配置サポート**

stack オプションを用いて、スタック領域のスタックアドレス計算コードを最適化することができます。

(8) **組み込み関数の追加**

次の組み込み関数を追加しました。

- 符号なしオーバフロー演算

(9) double=float **サポート**

double=float オプションにより、double 型宣言データや浮動小数点定数を float 型として扱います。

(10) noregsave **関数のサポート強化**

#pragma noregsave, _ _noregsave 宣言関数を呼び出す場合、呼び出し関数側でレジスタを保証するよう変更しました。

(11) **環境変数の複数指定**

インクルードディレクトリ用環境変数（CH38）で、複数のディレクトリ指定ができます。

(12) **構造体パラメータ／リターン値のレジスタ渡し**

structreg オプションを用いて、サイズの小さい構造体パラメータ／リターン値をレジスタで渡すことができます。

(13) 4byte **パラメータ／リターン値のレジスタ渡し（**cpu=300**）**

longreg オプションを用いて、4byte パラメータ／リターン値をレジスタで渡すことができます。

(14) **非** volatile **変数のループ外移動条件**

ループ判定式にある非 volatile の外部変数は、ループ内で副作用（関数呼び出し、代入等）がなくても、常にループ外移動最適化を抑止します。

(15) speed=loop=1 | 2 **のサポート**

speed=loop=1|2 オプションにより、ループ展開最適化の実行を制御できます。

(16) **アラインによるデータ割り付け変更**

align オプションにより、データを境界調整ごとに再配置し、境界調整による空きを最小限にできるようになりました。

(17) **暗黙の宣言の追加**

_ _HITACHI_ _、_ _HITACHI_VERSION_ _などが暗黙に#define 宣言されます。

(18) static **ラベル名**

#pragma asm～#pragma endasm および#pragma inline_asm 関数内でファイルスコープの 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。

(19) **言語仕様拡張・変更**

- union 初期化時のエラーを抑止します。

  ■例

```
union{
  char c[4];
}uu={ {‘a’,’b’,’c’}
};
```

- ビットフィールドに enum の記述ができるようになりました。

  ■例

```
struct{
  enum
E1{a,b,c}m1:2;
  enum E1     m2:2
}
```

- 列挙子の最後の","に対して、エラー出力を抑止します。

  ■例

```
enum E1{a,b,c,}m1;
```

- 共用体の代入と宣言を同時にできるようになりました。
  ■例
```
union U{
  int a,b;
}u1;
void test(){
  union U u2 = u1;
}
```

- C コンパイル時のアドレスに対するキャストのエラーチェックを緩和しました。
  アドレスに対するキャストを記述する場合は、必ずCコンパイル（lang=cオプション）を指定してください。
  ■例
```
int x;
short addr1=(short)&x;
```

- C プログラムの関数・変数宣言と#pragma 宣言の出現順の制約を緩和しました。
  ■例
```
void f(void);
#pragma interrupt f
void f(void){}  // 関数宣言後の#pragma宣言は有効になります。（Ver3台ではエラー）
```

- C++プログラムの関数・変数宣言と#pragma 宣言の出現順の規約を変更しました。
  ■例
```
void f(void){}
#pragma interrupt f
void f(void); // 関数定義後の#pragma宣言はエラーになります。
```

- 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。

B.3 Ver.4.0 から Ver.6.0 への追加機能

（※ Ver.5 は存在せず、欠番となります）

B.3.1 追加機能

(1) **新 CPU のサポート**

CPU 種別が H8SX のオブジェクトファイルの生成をサポートしました。

(2) 2byte **サイズポインタのサポート（**H8SX **のみ）**

_ _ptr16 キーワード指定か ptr16 オプション指定により 2byte サイズポインタが使用できます。
H8SX のアドバンストモードとマキシマムモードで有効です。

(3) **ビットフィールド並び順指定**

#pragma bit_order 指定か bit_order オプション指定により、メモリに対するビットフィールドメンバの詰め込み方を指定できます。

(4) **拡張メモリ間接方式の関数呼び出し（**H8SX **のみ）**

_ _indirect_ex キーワード指定か indirect=extended オプション指定により、拡張メモリ間接呼び出しとなる関数を宣言できます。また、#pragma indirect section はメモリ間接（@@aa:8）呼び出し用関数アドレス領域の $INDIRECT セクションだけでなく、拡張メモリ間接（@@aa:7）呼び出し用関数アドレス領域の $EXINDIRECT セクションのセクション名を切り替えることができます。

(5) **アセンブル機能（**H8SX **のみ）**

_ _asm キーワード指定により、C/C++ソースプログラムの中にアセンブリ言語を記述することが可能です。

(6) #line **出力抑止指定**

noline オプション指定により、プリプロセッサ展開時の#line の出力を抑止することができます。

(7) memcpy **関数、**strcpy **関数のインライン展開指定（**H8SX **のみ）**

library オプション指定により、memcpy と strcpy の 2 つのライブラリ関数をインライン展開することができます。

(8) **エラーレベルの変更**

change_message オプション指定により、インフォメーションレベルと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の変更が可能です。

(9) 8bit **絶対領域のアドレス指定（**H8SX **のみ）**

sbr オプション指定により、8bit 絶対アドレス領域の配置アドレスを指定することが可能です。

(10) **最適化機能の強化（**H8SX **のみ）**

以下のオプションが追加されたことにより、最適化の内容をさらに詳細に指定することが可能です。
opt_range、del_vacant_loop、max_unroll、infinite_loop、global_alloc、struct_alloc、const_var_propagate、volatile_loop

(11) **組み込み関数の追加**

以下の組み込み関数が追加されました。

- H8SX の 64bit 乗算（mulsu, muluu）
- H8SX のブロック転送命令（movmdb, movmdw, movmdl, movsd）
- ブロック転送命令組み込み関数強化（eepmovb, eepmovw, eepmovi）
- MOVFPE 命令組み込み関数見直し（_movfpe）

(12) **ワイルドカードのサポート**

入力ファイルをワイルドカードで指定することが可能です。

(13) **コンパイラ限界値の変更**

1 つのファイルに記述できる switch 文を 256 から 2048 にしました。

(14) **インフォメーションメッセージ表示の仕様変更**

Ver.4 では message や nomessage オプションをコマンドラインに複数指定すると最後に指定したオプションだけが有効になりました。本バージョンではコマンドライン内の各 nomessage オプションが指定する番号の和集合の番号のメッセージが抑止されます。

(15) **列挙型データの型**

byteenum オプション指定時に enum の値の範囲が 0～255 の場合に当該 enum 宣言した列挙型データを unsigned char 型として扱う仕様を追加しました。

(16) **インライン展開**

CPU 種別が H8SX の場合、speed=inline=<数値>オプションの<数値>の意味が、その他の CPU 種別と異なります。H8SX の場合、<数値>はプログラムサイズの増加率を表し、その他の CPU の場合、インライン展開できる関数のノード最大数を表します。

(17) 1 **バイトアラインデータセクション、**4 **バイトアラインデータセクション（**H8SX **のみ）**

align=4 オプションを指定するとサイズが奇数のデータを 1 バイトアラインのセクションへ、サイズが 4 の倍数のデータを 4 バイトアラインのセクションへ出力することができます。

(18) **セクション名**

section オプションで P、C、B、D セクションをセクション名 S に変更するとウォーニングエラーを発生します。S はスタック用に予約されたセクション名です。

(19) **暗黙の宣言の追加**

_ _H8SXN_ _、_ _H8SXM_ _、_ _H8SXA_ _、_ _H8SXX_ _、_ _HAS _MULTIPLIER_ _、_ _HAS _DIVIDER_ _、_ _INTRINSIC_LIB_ _、_ _DATA_ADDRESS_SIZE_ _、_ _H8_ _ 、_ _RENESAS_VERSION_ _、_ _RENESAS_ _が新たに暗黙に#define 宣言されます。

(20) **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。

(21) **リトルエンディアン空間サポート（**H8SX **のみ）**

H8SX は製品によりリトルエンディアン空間をサポートしています。リトルエンディアン空間の 2, 4 バイトのデータはデータサイズで書き込み、読み込みます。このために_ _evenaccess キーワードの機能を拡張しました。

B.3.2 Ver.6.0 の最適化機能に関する注意事項

この最適化に関する注意事項は、Ver.6.0 で H8SX のオブジェクトを生成するケースが対象です。それ以外のケースにおける最適化処理は Ver.4.0 以前と同様となります。

Ver.6.0 における最適化処理は最新のコンパイラ最適化技術を適用し、従来（Ver.4.0）では実現できなかった、ポインタや外部変数の別名解析や、制御フローを含めたデータ生存区間解析を実現しています。これにより、言語仕様で許されている範囲で Ver.4.0 よりも広範囲の最適化をしています。

一方、従来最適化対象にならなかったために動作していたプログラムが、最適化対象になったために動作しなくなるという場合もあります。

従来最適化されなかったもので、Ver.6.0 で最適化対象になる例を以下に示します。

(a) volatile 宣言のない外部変数、ポインタ変数のアクセスの扱い

volatile 宣言はプログラム上の逐次処理以外でデータの値が変更されるため、参照されるたびに必ず実際にデータが割り付けられた領域をアクセスすることを保証します。たとえば、割り込み処理やハードウェア処理によりデータの値が変更される場合などが挙げられます。

コンパイラ処理では、volatile 宣言のない変数については、プログラム内の逐次処理および関数呼び出しによる変更以外は変更されることはないと解釈します。

V4 までは、以下のような例で volatile 宣言がない外部変数に対する最適化を行っていました。

例：

```
int a;
f() {
  int *ptr=&a;
  *ptr=1; // <------ この代入式のみを削除
  *ptr=2;
}
```

V6 ではさらに以下のケースの最適化を行います。

これらの最適化を抑止したい場合は当該変数を volatile 宣言してください。

例 1：

```
int a;
f() {
  int *ptr=&a;
  *ptr &= ~( (0x0080) ); // <-- (1)
  while( !( *ptr & (0x0080) ) ) // <-- (2)
  {
    :
  }
}
```

この例では、最適化の結果(2)の while 文が無限ループになります。

- ポインタの別名解析により、（1）の*ptr と（2）の*ptr を同一値として扱います。
- （1）の式を（2）に伝播します。その結果、（2）式は以下のように変換されます。

```
 while( !( (*ptr & ~( (0x0080) ))  & (0x0080) ) ) // <-- (2)
-> while(!(*ptr & 0))
-> while(!(0))
-> while(1)
```

したがって、当該式は常に真となり、判定文が削除され、上記 while 文は無限ループになります。

例 2：

```
int a,b;
f() {
  a=1; // <-- (1)
  if(a) // <-- (2)
  {
    b=1; // <-- (3)
  }
}
```

この例では、最適化の結果（2）の if 文判定式が削除され、常に（3）が実行されます。

- 外部変数の別名解析により、（1）と（2）の a を同一値として扱います。

- （1）の定数値を（2）に伝播します。その結果、（2）式は以下のように変換されます。

```
-> if(1)
```

したがって、当該式は常に真となり、判定文が削除され、上記（3）式が常に実行されます。

例 3：

```
int a,b,c;
f() {
     a=1; // <-- (1)
     if(c) // <-- (2)
  {
      b=1; // <-- (3)
  }
  a=2; // <-- (4)
}
```

この例では、最適化の結果（1）の式が削除されます。

- if 文の制御式を含めた制御フローを求めます。
- 制御フローと外部変数の別名解析により、a に（1）で設定した値は使用されていないことがわかります。したがって、上記（1）式は参照されない冗長式となり、削除されます。

例 4：

```
int a;
int b[10];
f() {
   int i; // <-- (1)
   for(i=0; i<10; i++) // <-- (2)
   {
      b[i]=a; // <-- (3)
   }
}
```

この例では、最適化の結果（3）の a はループの前に一度だけ参照され、ループ内では常に一定値として扱われます。

- for ループ制御式を含めた制御フローを求めます。
- 制御フローと外部変数の別名解析により、（3）の a がループ内で一定値として扱います。
- （3）の a の参照式を（2）の for ループ外に移動します。

```
   temp=a;
   for(i=0; i<10; i++) // <-- (2)
   {
      b[i]=temp; // <-- (3)
   }
```

したがって、（3）式の変数 a はループ中は一定値となります。

例 5：

```
int a;
f() {
   a = 0; // <-- (1)
   while (1); // <-- (2)
}
```

この例では、最適化の結果（1）の文は不要とみなされ削除されます。

- （2）は無限ループなので本関数は出口がないと判断します。
- 無限ループ内で a の参照はないので、（1）の設定は不要コードとみなされ削除します。

(b) volatile_loop オプション

volatile_loop オプションは、ループ制御変数が非 volatile 外部変数でかつ判定式が単純な場合に、ループ制御変数を volatile として扱い、無限ループ化を抑止します。しかし、ループ制御変数がループ内不変でない場合は volatile 化の対象外になります。

Ver.6.0 では、このような場合は当該変数を volatile 宣言してください。

以下に例を示します。

例：

```
struct {
  unsigned char a:1;
} ST;
int a;
extern void f();
void func() {
  while (ST.a) { // <-- (1)
    if (a) { // <-- (2)
      f(); // <-- (3)
    }
  }
}
```

この例の場合、ST.a は f()で書き換えられる可能性があり、ループ内で不変とはみなされません。よって、volatile_loop オプションを指定しても ST は volatile 化されません。

- （2）の条件が成立した場合は、（3）が実行され f()で ST.a の値が書き換えられる可能性があるため、呼び出し後 ST.a を再ロードします。
- （2）の条件が不成立の場合は、ST.a が書き換えられないので前回（1）の判定で使用した ST.a の値をそのまま使用します。

B.3.3 Ver.4.0 オブジェクトと Ver.6.0 オブジェクトの互換性について

Ver.4.0 オブジェクトと Ver.6.0 オブジェクトをリンクするには、以下の条件を満たす必要があります。

(1) C ソースプログラム

以下の関数インタフェースに影響するオプションが同一であること

- reparam
- longreg/nolongreg
- double=float
- structreg/nostructreg
- stack
- byteenum
- pack/unpack

(2) アセンブリプログラム

H8S、H8/300 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の「9.3.2 関数呼び出しのインタフェース」の関数呼び出し規約に準拠していること。

【注】 1. マニュアルに記述のない内容については、バージョンアップによる互換性は保証していません。
レジスタの退避・回復順序など、コンパイラの出力コードに依存するアセンブリコードを記述している場合は、Ver.4.0 オブジェクトと Ver.6.0 オブジェクトをリンクできません。

2. OS やミドルウェアなどとのリンクについては、購入元にお問い合わせください。

B.4 Ver.6.0 から Ver.6.1 への追加機能

B.4.1 追加機能

(1) AE5 のサポート

AE5 の CPU をサポートしました。

(2) ANSI 準拠対応拡張

strict_ansi オプションによって、浮動小数点演算の結合則を ANSI に準拠して出力できます。

(3) Ver.4.0 コード互換オブジェクト出力

H8S の CPU において、旧バージョン(Ver.4.0)と互換性のあるオブジェクトを出力するための legacy=v4 オプションをサポートしました。

(4) legacy=v4 指定時の cpuexpand=v6 仕様拡張

cpuexpand=v6 は、legacy=v4 オプションを指定したときに、Ver.6.00 の cpuexpand の出力仕様に沿ったオブジェクトを出力できます。

(5) register 記憶クラス変数優先割り付け

enable_register オプションは、register 記憶クラスを指定した変数を優先的にレジスタ割り付けます。

(6) 最適化範囲分割機能

scope/noscope オプションは、関数内で最適化範囲を分割するか否かを選択できます。

(7) ファイル間インライン展開

ファイルをまたがった関数インライン展開を行う file_inline オプションと、対象となるファイルを取り込む先のパス名を指定する file_inline_path オプションをサポートしました。

(8) 組み込み関数の追加

VBR の設定設定を行う、set_vbr 組み込み関数をサポートしました。

(9) #pragma address

変数に指定した絶対アドレスを割り付けることができる、#pragma address 拡張命令を追加しました。

(10) .stack 制御命令出力

code=asmcode を指定した場合、アセンブリソース内に.stack 制御命令を出力するようになりました。

(11) 環境変数の追加

SBR レジスタの初期値を設定可能な CH38SBR の環境変数をサポートしました。

(12) 暗黙の宣言の追加

以下の暗黙の宣言が追加になりました。

_ _AE5_ _

_ _ABS16_ _

B.4.2 Ver.6.01 の最適化機能に関する注意事項

この最適化に関する注意事項は、Ver.6.01 で H8SX,H8S(legacy=v4 オプション指定なし)のオブジェクトを生成するケースが対象です。それ以外のケースにおける最適化処理は Ver.4.0 以前と同様となります。

Ver.6.01 における H8SX,H8S の最適化処理は、最新のコンパイラ最適化技術を適用し、従来(Ver.4.0)では実現できなかった、ポインタや外部変数の別名解析や、制御フローを含めたデータ生存区間解析を実現しています。これにより、言語仕様で許されている範囲で V4 よりも広範囲の最適化をしています。

そのため、特に H8S を Ver.6.0 で開発していた場合に、Ver.6.01 へリビジョンアップを行うと、上記の最適化技術により、生成されるコードが大幅に異なるようになります。

一方、従来最適化対象にならなかったために動作していたプログラムが、最適化対象になったために動作しなくなるという場合もあります。

従来最適化されなかったもので、Ver.6.01 で最適化対象になる例を以下に示します。

(1) volatile 宣言のない外部変数、ポインタ変数のアクセスの扱い

volatile 宣言はプログラム上の逐次処理以外でデータの値が変更されるため、参照されるたびに必ず実際にデータが割り付けられた領域をアクセスすることを保証します。たとえば、割り込み処理やハードウェア処理によりデータの値が変更される場合などが挙げられます。

コンパイラ処理では、volatile 宣言のない変数については、プログラム内の逐次処理および関数呼び出しによる変更以外は変更されることはないと解釈します。

Ver.4.0 までは、以下のような例で volatile 宣言がない外部変数に対する最適化を行っていました。

例：

```
int a;
f() {
   int *ptr=&a;
   *ptr=1; // <------ この代入式のみを削除
   *ptr=2;
}
```

Ver.6.01 では更に以下のケースの最適化を行います。

これらの最適化を抑止したい場合は当該変数を volatile 宣言してください。

例 1：

```
int a;
f() {
   int *ptr=&a;
   *ptr &= ~( (0x0080) ); // <-- (1)
   while( !( *ptr & (0x0080) ) ) // <-- (2)
   {
      :
   }
}
```

この例では、最適化の結果（2）の while 文が無限ループになります。

- ポインタの別名解析により、（1）の*ptr と（2）の*ptr を同一値として扱います。
- （1）の式を（2）に伝播します。その結果、（2）式は以下のように変換されます。

```
   while( !( (*ptr & ~( (0x0080) ))  & (0x0080) ) ) // <-- (2)
-> while(!(*ptr & 0))
-> while(!(0))
-> while(1)
```

したがって、当該式は常に真となり、判定文が削除され、上記 while 文は無限ループになります。

例 2：

```
int a,b;
f() {
   a=1; // <-- (1)
   if(a) // <-- (2)
   {
      b=1; // <-- (3)
   }
}
```

この例では、最適化の結果（2）の if 文判定式が削除され、常に（3）が実行されます。
- 外部変数の別名解析により、（1）と（2）の a を同一値として扱います。
- （1）の定数値を（2）に伝播します。その結果、（2）式は以下のように変換されます。

```
-> if(1)
```

したがって、当該式は常に真となり、判定文が削除され、上記（3）式が常に実行されます。

例 3：

```
int a,b,c;
f() {
   a=1; // <-- (1)
   if(c) // <-- (2)
   {
      b=1; // <-- (3)
   }
   a=2; // <-- (4)
}
```

この例では、最適化の結果(1)の式が削除されます。
- if 文の制御式を含めた制御フローを求めます。
- 制御フローと外部変数の別名解析により、a に（1）で設定した値は使用されていないことがわかります。したがって、上記（1）式は参照されない冗長式となり、削除されます。

例 4：

```
int a;
int b[10];
f() {
   int i; // <-- (1)
   for(i=0; i<10; i++) // <-- (2)
   {
      b[i]=a; // <-- (3)
   }
}
```

この例では、最適化の結果（3）の a はループの前に一度だけ参照され、ループ内では常に一定値として扱われます。
- for ループ制御式を含めた制御フローを求めます。
- 制御フローと外部変数の別名解析により、（3）の a がループ内で一定値として扱います。
- （3）の a の参照式を（2）の for ループ外に移動します。

```
temp=a;
for(i=0; i<10; i++) // <-- (2)
{
   b[i]=temp; // <-- (3)
}
```

したがって、（3）式の変数 a はループ中は一定値となります。

例 5：

```
int a;
f() {
   a = 0; // <-- (1)
   while (1); // <-- (2)
}
```

この例では、最適化の結果(1)の文は不要とみなされ削除されます。
- （2）は無限ループなので本関数は出口がないと判断します。
- 無限ループ内で a の参照はないので、（1）の設定は不要コードとみなされ削除します。

(2) volatile_loop **オプション**

volatile_loop オプションは、ループ制御変数が非 volatile 外部変数でかつ判定式が単純な場合に、ループ制御変数を volatile として扱い、無限ループ化を抑止します。しかし、ループ制御変数がループ内不変でない場合は volatile 化の対象外になります。

Ver.6.01 では、このような場合は当該変数を volatile 宣言してください。

以下に例を示します。

例：

```
struct {
   unsigned char a:1;
} ST;
int a;
extern void f();
void func() {
   while (ST.a) { // <-- (1)
      if (a) { // <-- (2)
      f(); // <-- (3)
      }
   }
}
```

この例の場合、ST.a は f()で書き換わる可能性があり、ループ内で不変とはみなされません。よって、volatile_loop オプションを指定しても ST は volatile 化されません。

- （2）の条件が成立した場合は、（3）が実行され f()で ST.a の値が書き換わる可能性があるため、呼び出し後 ST.a を再ロードします。
- （2）の条件が不成立の場合は、ST.a が書き換わらないので前回（1）の判定で使用した ST.a の値をそのまま使用します。

B.4.3 Ver.4.0 オブジェクトと Ver.6.01 オブジェクトの互換性について

Ver.4.0 オブジェクトと Ver.6.01 オブジェクトをリンクするには、以下の条件を満たす必要があります。

(1) **C ソースプログラム**

以下の関数インタフェースに影響するオプションが同一であること

- regparam
- longreg/nolongreg
- double=float
- structreg/nostructreg
- stack
- byteenum
- pack/unpack

(2) **アセンブリプログラム**

H8S、H8/300 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「9.3.2 関数呼び出しのインタフェース」の関数呼び出し規約に準拠していること

【注 1】

マニュアルに記述のない内容については、バージョンアップによる互換性は保証していません。
レジスタの退避／回復順序等、コンパイラの出力コードに依存するアセンブリコードを記述している場合は、
Ver.4.0 オブジェクトと Ver.6.01 オブジェクトをリンクできません。

【注 2】

OS やミドルウェアなどとのリンクについては、購入元にお問い合わせください。

C. 制限値一覧

H8S,H8/300 C/C++コンパイラ Ver6.01 の制限値を示します。

No	分類	項目	限界値
1	コンパイラの起動	一度にコンパイルできるソースプログラムの数	制限なし*1
2		define オプションで指定できるマクロ名の総数	制限なし
3		ファイル名の長さ	制限なし (OS に依存)
4	ソースプログラムの行数	1 行の長さ	32768 文字 (H8SX/H8S) 16384 文字 (300H/300)
5		1 ファイルあたりのソースプログラムの行数	制限なし
6		コンパイル可能なソースプログラムの行数	制限なし
7	プリプロセッサ	#include 文によるファイルのネストの深さ	制限なし
8		#define 文によるマクロ名の総数	制限なし
9		マクロ定義、マクロ呼び出しで指定できる引数の数	制限なし
10		マクロ名の再置き換えの数	制限なし
11		#if、#ifdef、#indef、#else、#elif 文のネストの深さ	制限なし
12		#if、#elif 文で指定できる演算子と被演算子の合計数	制限なし
13	宣言	関数定義の数	制限なし
14		外部結合となる識別子（外部名）の数	制限なし
15		一つの関数内で有効な識別子（内部名）の数	制限なし
16		基本型を修飾するポインタ型、配列型、関数型の合計数	16 個
17		配列の次元数	6 次元
18		配列、構造体のサイズ*2	
		H8SX ノーマルモード、 H8S/2600 ノーマルモード、 H8S/2000 ノーマルモード、 H8/300H ノーマルモード、 H8/300	65535 バイト
		H8SX ミドルモード、 H8SX アドバンストモード（ptr16 有）、 H8SX マキシマムモード（ptr16 有）	32767 バイト
		H8/300H アドバンストモード	16777215 バイト
		H8SX アドバンストモード（ptr16 無）、 H8SX マキシマムモード（ptr16 無） H8S/2600 アドバンストモード、 H8S/2000 アドバンストモード	2147483647 バイト 4294967295 バイト （legacy=v4 オプションを指定した場合）
19	文	複文のネストの深さ	制限なし
20		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文のネストの深さ	4096 レベル （H8SX/H8S） 256 レベル （300H/300）
21		一つの関数内で指定できる goto ラベルの数	2147483646 （H8SX/H8S） 511 個（300H/300）
22		switch 文の数	2048 個
23		switch 文のネストの深さ	2048 レベル （H8SX/H8S） 128 レベル （300H/300）

No	分類	項目	限界値
24	文	case ラベルの数	2147483646 個（H8SX/H8S） 511 個（300H/300）
25		for 文のネストの深さ	2048 レベル（H8SX/H8S） 128 レベル（300H/300）
26	式	文字列の長さ	32766 文字
27		関数定義、関数呼び出しで指定できる引数の数	2147483646 個（H8SX/H8S） 63 個（300H/300）*3
28		一つの式で指定できる演算子と被演算子の合計数	約 500 個
29	標準インクルード	open 関数で一度にオープンできるファイルの数	可変*4

【注】 *1 PC 版はコマンドラインの制約により 127 文字までの入力となります。

*2 アドバンストモードの場合、アドレス空間のビット幅を指定すると、アドレス空間のビット幅に対応するアドレス空間サイズが優先します。

*3 非静的関数メンバのときは、62 個になります。

*4 open 関数で一度にオープンできるファイルの数を指定できます。

D. ASCII コード表

ASCII コード表は以下のようになります。

上位 4 バイト / 下位 4 バイト	0	1	2	3	4	5	6	7
0	NUL	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
H8S、H8/300シリーズ　C/C++コンパイラパッケージ

発行年月日　1997 年 5 月　第 1 版
2005 年 8 月 29 日　Rev.5.00
発　　行　株式会社ルネサス テクノロジ 営業企画統括部
〒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

株式会社ルネサス テクノロジ 営業企画統括部　〒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

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

H8S、H8/300 シリーズ　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

RJJ05B0558-0500