お客様各位

カタログ等資料中の旧社名の扱いについて

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

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

2010年4月1日
ルネサスエレクトロニクス株式会社

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

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

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
3. 当社製品を改造、改変、複製等しないでください。
4. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器の設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
5. 輸出に際しては、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。本資料に記載されている当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途の目的で使用しないでください。また、当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器に使用することができません。
6. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「特定水準」に分類しております。また、各品質水準は、以下に示す用途に製品が使われることを意図しておりますので、当社製品の品質水準をご確認ください。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されていない用途に当社製品を使用したことによりお客様または第三者に生じた損害等に関し、当社は、一切その責任を負いません。なお、当社製品のデータ・シート、データ・ブック等の資料で特に品質水準の表示がない場合は、標準水準製品であることを表します。

   標準水準： コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通用信号機器、防災・防犯装置、各種安全装置、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   特定水準： 航空機器、航空宇宙機器、海底中継機器、原子力制御システム、生命維持のための医療機器（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害などを生じさせないようお客様の責任において冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、機器またはシステムとしての出荷保証をお願いいたします。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、詳細につきましては製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

注 1. 本資料において使用されている「当社」とは、ルネサスエレクトロニクス株式会社およびルネサスエレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

SuperH RISC engine C/C++ コンパイラパッケージ

アプリケーションノート：＜リファレンス＞ 追加機能一覧

本ドキュメントでは、SuperH RISC engine C/C++ コンパイラの各バージョンの追加機能とバージョンアップ時の注意点を紹介します。

A.1 Ver.1.0 から Ver.2.0 への追加機能

SHC コンパイラ Ver.2.0 で追加された機能概要を**表** A.1 に示します。

表 A .1 SHC **コンパイラ** Ver.2.0 **追加機能概要**

項番	機能	内容
1	SH7600 **シリーズのサポート**	SH7000 **シリーズのほかに、**SH7600 **シリーズの命令を活用したオブジェクトを生成することもできます。**
2	**ポジションインディペンデントコード**	SH7600 **シリーズのオブジェクトでは、プログラムセクションを任意のアドレスに配置できるオブジェクトが生成できます。**
3	**文字列の出力領域指定**	**文字列データを定数セクション**(ROM)**に置くかデータセクション**(RAM)**に置くかをオプションで選択できます。**
4	**コメントのネスト**	**コメントをネストさせるかさせないかを指定するオプションをサポートします。**
5	**サイズ、速度の優先指定**	**オブジェクト生成時にサイズを優先するかスピードを優先するかをオプションで指定できます。**
6	**セクション名切り替えのサポート**	**プログラムの途中で**#pragma **指令によってオブジェクトを出力するセクション名を切り替えることができます。**
7	mac **組み込み関数**	MAC **命令を用いて** 2 **つの配列の積和演算を行う組み込み関数をサポートします。**
8	**システムコール組み込み関数**	ITRON **仕様** OS HI-SH7 **のシステムコールを直接呼び出す組み込み関数をサポートします。**
9	**単精度初等関数ライブラリ**	**単精度の初等関数ライブラリをサポートします。**
10	char **型のビットフィールド**	char **型のビットフィールドをサポートします。**

A.2 Ver.2.0 から Ver.3.0 への追加機能

SHC コンパイラ Ver.3.0 で追加された機能概要を**表** A.2 に示します。

表 A.2 SHC **コンパイラ** Ver.3.0 **追加機能概要**

No.	機能	内容
1	**最適化強化**	**最適化機能を大幅に強化します。 また、スピード重視、サイズ重視の最適化オプションを使い分けることができます。**
2	SH-3 **サポート**	SH-3 **用のオブジェクト生成オプションを実現すると共に、**SH-3 **の特長機能である** Little Endian **もサポートします。また、**SH-3 **のデータプリフェッチ命令を組み込み関数としてサポートします。**
3	**コンパイラ限界値の拡張**	**一度にコンパイルできるファイル数、インクルードファイルのネストレベルなどの限界値を拡張します。**
4	**文字列漢字コードのサポート**	**シフト** JIS、EUC **の漢字コードを、プログラム内に文字列データとして記述できます。**
5	**ファイルによるオプション指定**	**コマンドラインのオプション指定をファイルで行うことができます。**
6	SH-2 **除算器の活用**	SH-2 **の除算器を活用した除算コードを生成します。**
7	**インライン展開**	C **記述、アセンブラ記述のユーザルーチンをインライン展開することを指定できます。**
8	**短いアドレス指定の活用**	2 **バイトサイズのアドレス、**GBR **相対のデータなど、短いアドレシングが可能な変数を指定できます。**
9	**レジスタ退避・回復の制御**	**レジスタの退避・回復の抑止を指定し、関数のスピード、サイズを向上させることができます。**

(1) 最適化強化

Ver.3.0 の最適化は、スピード重視(-SPEED オプション)、サイズ重視(-SIZE オプション)の両オプションを設け、それぞれの最適化機能を大幅に強化しています。

スピードに関しては、ループ最適化の強化、インライン展開の実施などにより、実行スピードが約 10%向上、1MIPS/1MHz の性能を達成しています。

サイズに関しては、サイズ重視の命令生成、重複処理の併合の大幅な強化などにより、オブジェクトサイズを約 20%削減しています。さらに Ver.3.0 で導入された拡張機能(8.短いアドレス指定の活用、9.レジスタ退避・回復の制御)の活用により、さらにオブジェクトサイズを削減することが可能です。

(2) SH-3 サポート

SH-1、SH-2 に加えて、SH-3 のオブジェクト生成を指定することができます(-CPU=SH3 オプション)。さらに、SH-3 用の機能として、以下をサポートします。

(a) メモリ内のビット並び順の設定機能に対応して、-ENDIAN オプション(-ENDIAN=BIG、または LITTLE)をサポート。
(b) キャッシュのプリフェッチ命令(PREF)を生成する拡張組み込み関数 prefetch をサポート。

(3) コンパイラ限界値の拡張

以下の点で、コンパイラ限界値をさらに拡張します。

表 A.3 **コンパイラ限界値の拡張**

No.	項目	Ver.2.0	Ver.3.0
1	**一度にコンパイルできるソースプログラムの数**	16 **ファイル**	**制限なし**
2	1 **ファイルあたりのソース行数**	32767 **行**	65535 **行**
3	**コンパイル単位全体のソース行数**	32767 **行**	**制限なし**
4	#include **のネストレベル**	8 **レベル**	30 **レベル**

(4) 文字列内漢字コードのサポート

シフト JIS、EUC の漢字コードを、プログラム内に文字列データとしても記述できます。
入力コードがシフト JIS の場合(-SJIS オプション)、出力コードもシフト JIS、入力コードが EUC の場合(-EUC オプション)、出力コードも EUC です。
ただし、現状の GUI は漢字コードのデータ表示には対応しておりません。

(5) ファイルによるオプションの指定

-SUBCOMMAND オプションでファイル名を指定することによって、オプションをファイル内から取り込むことができるようになります。これによって、複雑なオプションを毎回コマンドラインから指定する必要がなくなります。

(6) SH-2 除算器の活用

SH-2 の除算器を活用するために、以下のオプションをサポートします。

(a) -DIVISION=CPU除算器を使用しないオブジェクトを生成します。
(b) -DIVISION=PERIPHERAL除算器を使用するオブジェクトを生成します。
除算器を使用するときは割り込みを禁止します。
(c) -DIVISION=NOMASK除算器を使用するオブジェクトを生成します。
割り込み処理では除算器を使用しないことを想定します。

(7) インライン展開

(a) C関数のインライン展開

-SPEED オプションを指定すると、コンパイラは、小さな関数を自動的にインライン展開します。さらに、-INLINE オプションによって、インライン展開する関数の大きさの条件を変更することができます。
インライン展開は、#pragma 指定によって明示的にすることもできます。
#pragma inline は、C 記述のユーザ関数をインライン展開することを指定します。

例 (C 関数のインライン展開)：

```
#pragma inline (func)
int func(int a,int b)
{
    return (a+b)/2 ;
}

main()
{
    i=func(10,20); /*i=(10+20)/2 に展開されます*/
}
```

(b) 2アセンブラ関数のインライン展開

#pragma inline_asm はアセンブラ記述のユーザ関数をインライン展開することを指定します。
ただし、#pragma inline_asm でインライン展開を行った場合、コンパイラの出力はアセンブラソースになります。
この場合 C 言語レベルのデバッグはできなくなります。

例 (アセンブラ関数のインライン展開)：

```
#pragma inline_asm(rotl)
int rotl(int a)
{
   ROTL   R4
   MOV    R4,R0
}

main()
{
   i=rotl(i) ; /*変数 i をレジスタ R4 に設定し、rotl 関数のコードを展開します */
}
```

(8) 短いアドレス指定の活用

(a) 2バイトアドレス変数の指定

#pragma abs16(<変数名>)によって、変数が 2 バイトで指定できるアドレス範囲(-32768～32767)に割り付けられていることを指定できます。この指定によって、変数を参照するオブジェクトサイズを削減することができます。

(b) GBRベース変数の指定

#pragma gbr(<変数名>)によって、変数を GBR 相対アドレシングモードで参照することができることを指定できます。この指定によって変数を参照するオブジェクトサイズを削減するとともに、GBR 相対アドレシングモードに特有なメモリ上のビット操作命令を活用することができます。

(9) レジスタ退避・回復の制御

#pragma noregsave(<関数名>)によって、関数の入口、出口でのレジスタの退避・回復を抑止することを指定できます。これにより、レジスタの退避・回復のない高速でコンパクトな関数を作成できます。#pragma noregsave を指定した関数は、通常の関数から呼び出すことはできませんが、#pragma noregsave を指定した関数を呼び出すように明示的に指定した C 言語関数(#pragma regsave)からは呼び出すことができます。

頻繁に実行する関数を#pragma noregsave と指定することによって、プログラムサイズを削減し、実行速度を向上させることができます。

A.3 Ver.3.0 から Ver.4.1 への追加機能

SuperH RISC engine C/C++コンパイラ Ver.4.1 で追加した機能概要を説明します。

(1) 外部変数のレジスタ割り当て

#pragma global_register(<変数名>=<レジスタ番号>)によって、外部変数をレジスタに割り当てることが可能となりました。

(2) キャッシュを意識した最適化

命令フェッチやキャッシュを無駄なく利用するためにラベルを 16 バイト整合して割り付けるオプション、"-align16"をサポートしました。

(3) インライン展開機能の強化

インライン展開により、関数本体が使用されなくなった場合に、その削除を行う機能を追加しました。インライン展開後の関数本体が必要ない関数には static を指定してください。また、呼び出しやアドレス参照されることのない static 関数も同様に削除します。

例)

```
#pragma inline(func)
int a;
static int func(){  /* func() 関数本体削除 */
    a++;
}

main(){
    func();          /*  a++;  インライン展開 */
}
```

(4) 再帰的なインライン展開

関数を再帰的にインライン展開する機能を追加しました。再帰の深さは、"-nestinline"オプションにより指定可能です。

(5) ループ展開最適化オプション

"-loop", "-noloop"オプション指定によりループ処理を展開させる最適化を行うかどうかを"-speed", "-size"オプションとは独立に指定可能となりました。
（非最適化オプション指定時、本オプションは無効）

(6) 2 バイトアドレス変数指定のオプション化

従来 2 バイトアドレス変数は #pragma abs16 を用いて個々に指定する必要がありましたが、"-abs16"オプションにより一括指定する機能をサポートしました。"-abs16=run" では実行時ルーチンのみを、"-abs16=all"では実行時ルーチンを含む全変数および関数を 2 バイトアドレスとして指定できます。

(7) 関数 return 値の上位バイトの保証

従来 (unsigned) char, short 型の値を返す関数の return 値の上位バイトは非保証でしたが、オプション "-rtnext" を指定することにより保証（R0 の上位バイトを符号拡張または 0 拡張）する機能を追加しました。

(8) リスティングファイルの充実化

以前のバージョンに比べ、オブジェクトリスト、アセンブリソースに情報を充実し、見やすくしました。

- リストファイル中に C ソースとアセンブリプログラムを文単位に同時出力することにより、その対応が見やすくなりました。（"-show=source,object" オプション指定時）
- （これに伴い、"-show"オプションのデフォルトを source から nosource に変更しました。）
- 関数のスタック使用量算出のための情報として、その関数での使用実行時ルーチン名一覧を追加。
- 定数プールからのデータロード命令に、ロードデータをコメント表示。

例)

1: float x;
2: func(){
3: x/=1000;
4: }

リスティングファイル

```
func.c 1      float x;
func.c 2      func(){       ※(a)Cソースとアセンブリプログラムの同時出力
000000        func:         ; function: func
    ; frame size=4
                            ; used runtime library name:
                            ; divs ※(b)実行時ルーチン名
000000  4F22          STS.L   PR,@-R15
func.c  3 x/=1000;
000002  D404          MOV.L   L216+2,R4   ; x
000004  D004          MOV.L   L216+6,R0   ; H'447A0000 ※(c)ロードデータ
000006  D305          MOV.L   L216+10,R3  ; divs
000008  430B          JSR     @R3
00000A  6142          MOV.L   @R4,R1
func.c  4             }
00000C  4F26          LDS.L   @R15+,PR
00000E  000B          RTS
000010  2402          MOV.L   R0,@R4
000012  L216:
000012  00000002      .RES.W  1
000014  <00000000>    .DATA.L x
000018  447A0000      .DATA.L H'447A0000
00001C  <00000000>    .DATA.L divs
000000 _x:                              ; static: x
000000 00000004       .RES.L  1
```

(9) **エラーメッセージの強化**

"-message"オプション指定によりインフォメーションメッセージを出力することでコーディングミスのチェックを強化しました。

例)

```
1:    void func(){
2:          int a;
3:          a++;
4:          sub(a);
5:    }
```

インフォメーションメッセージ

```
line 3: 0011 (I) Used before set symbol : "a"  (auto変数の未定義参照)
line 4: 0200 (I) No prototype function         (プロトタイプ宣言なし)
```

また、エラーとなっている識別子、字句、番号をメッセージ中に追加することによりエラー箇所を見つけやすくなりました。

例)

```
: 2118 (E) Prototype mismatch "識別子"
: 2119 (E) Not a parameter name "識別子"
: 2201 (E) Cannot covert parameter "番号"
: 2225 (E) Undeclared name "識別子"
: 2500 (E) Illegal token "字句"
```

(10) **日本語文字コードの自動変換**

EUC、またはシフト JIS 日本語コードで記述された文字列をオブジェクトファイルに出力する際に、オプションにより指定された日本語コードへの自動変換が可能です。

(a) "-outcode=euc"オプションにより日本語コードをEUCコードへ変換。
(b) "-outcode=sjis"オプションにより日本語コードをシフトJISコードへ変換。

(11) 環境変数による CPU タイプ指定

CPU タイプをコマンドラインオプションで指定する代わりに、環境変数による指定を可能にしました。

環境変数指定

SHCPU=SH1 （"-cpu=sh1"オプションと同意）

SHCPU=SH2 （"-cpu=sh2"オプションと同意）

SHCPU=SH3 （"-cpu=sh3"オプションと同意）

(12) double 型データの float 型データ化オプション

"-double=float"オプションにより、double型で宣言されたデータを float型に読み替える機能を追加しました。
double型の精度の必要のないプログラムでは、ソース修正なしで実行速度の向上が可能です。

A.4 Ver.4.1 から Ver.5.0 への追加機能

SuperH RISC engine C/C++コンパイラ Ver.5.0 で追加した機能概要を説明します。

(1) 文字数の拡張

1 行論理行の文字数の制限を、4,096 文字から 32,768 文字までに拡張しました。

(2) コンパイルソース行制限の廃止

1 ファイルで 65,535 行を超えるソースはコンパイルできない制限を廃止しました。
ただし、65,535 行を超えた部分はデバッグできません。

(3) SH-4 命令対応

SH ファミリの CPU 展開に即応し、新たに SH-4 対応を追加しました。“ -cpu=sh4 ” オプション指定により、SH-4 のオブジェクトを生成できます。

(4) 正規化モードの追加

“ -denormalize=on|off ”オプション指定により、非正規化数を扱うか、0 とするか選択が可能になりました。-cpu=sh4 時のみ有効です。
ただし、“ -denormalize=on ” のとき、FPU に非正規化数が入力されると例外発生するので、非正規化数を処理するための例外処理をソフトウェアで記述する必要があります。

(5) 丸めモード追加

“ -round=nearest|zero ” オプション指定により、Round to zero で丸めるか、Round to nearest で丸めるかの選択が可能になりました。-cpu=sh4 時のみ有効です。

(6) 環境変数によるコンパイラオプション指定の SH-4 対応

CPU をコマンドラインオプションで指定する代わりに、環境変数 “ SHCPU ” による SH-4 指定を可能にしました。“ SHCPU=SH4 ” で指定できます。

(7) SH-2E 対応

“ -cpu=sh2e ” オプション指定により、SH-2E のオブジェクトを生成できます。

(8) 環境変数によるコンパイラオプション指定の SH-2E 対応

CPU をコマンドラインオプションで指定する代わりに、環境変数“ SHCPU ”による SH-2E 指定を可能にしました。“ SHCPU=SH2E ” で指定できます。

(9) 拡張子による C/C++の判別

コンパイラは shc、shcpp のコマンドの使い分けでもコンパイル時の文法を決定しますが、shc コマンド使用時でもファイルの拡張子やオプションにより C++コンパイルを行います。詳細を**表** B.4 に示します。

表 A.4 **コンパイル条件判別表**

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

A.5 Ver.5.0 から Ver.5.1 への追加機能

SuperH RISC engine C/C++コンパイラ Ver.5.1 で追加した機能概要を説明します。

(1) SH3-DSP ライブラリのサポート

従来の SH-DSP 用に加えて SH3-DSP にも適用可能なライブラリをサポートしました。

(2) Embedded C++言語のサポート

組み込みシステムに適した C++仕様である Embedded C++の言語仕様とライブラリをサポートしました。

- ・bool型サポート
- ・多重継承の警告
- ・Embedded C++クラスライブラリのサポート

(3) モジュール間最適化機能のサポート

以下の最適化を実施してサイズ／スピードに優れたオブジェクトを生成します。
この最適化により約 10%のサイズ削減、7～8%の実行スピード向上を実現しました。

- ・無駄なレジスタ退避回復コードの削減
- ・参照しない変数／関数の削除
- ・共通コードをルーチン化
- ・関数呼び出しコードの最適化

(4) コンパイルスピードの向上

最適化処理の改善によるコンパイラスピードの高速化を実現しました。
最大 2 倍、平均 130%のスピードアップを達成しました。

(5) 制限値の拡張

- ・コマンドライン長の制限を256から4096へ拡張しました。
- ・ファイル名長の制限を128から251へ拡張しました。
- ・文字列リテラル長の制限を512から32767へ拡張しました。

(6) 最適化強化

オブジェクト性能を向上させる各種最適化を強化しました。

(7) C++コメントのサポート

C 言語でも「//」コメントの使用が可能になりました。

(8) 統合環境の変更（PC 版）

従来の PC の統合化環境 HIM(Hitachi Integration Manager)に代わる新たな統合化環境 HEW(High-performance Embedded Workshop)になりました。
HIM と比較し以下機能を追加致しました。

- ・プロジェクトジェネレータ
  各CPUごとに周辺I/Oなどを定義したヘッダファイルを自動生成します。
- ・バージョン管理ツールとの連動
  市販のバージョン管理ツールとの連動インタフェースをサポートしました。
- ・階層プロジェクトのサポート
  プロジェクト内に複数のサブプロジェクトを定義し、階層的に管理することが可能となりました。
- ・ネットワーク対応
  WindowsNTのCSS環境下での開発が可能になりました。

A.6 Ver.5.1 から Ver.6.0 への追加機能

SuperH RISC engine C/C++コンパイラ Ver.6.0 で追加した機能概要を説明します。

(1) 制限値の緩和

ソースプログラムやコマンドラインの制限を大幅に緩和しました。

・ファイル名長：251バイト→無制限
・シンボル長：251バイト→無制限
・シンボル数：32767個→無制限
・ソースプログラム行数：C/C++：32767行、ASM：65535行→無制限
・Cプログラム文字列長：512 文字→ 32766 文字
・アセンブリプログラム行長：255 文字→ 8192 文字
・サブコマンドファイル行長：ASM:300 バイト、optlnk:512 バイト→無制限
・最適化リンケージエディタrom オプションのパラメータ数:64 個→ 無制限

(2) ディレクトリ名、ファイル名のハイフン(-)

ディレクトリ名、ファイル名にハイフン(-)を指定できるようになりました。

(3) コピーライト表示抑止

logo/nologo オプション指定により、コピーライト表示有無を指定できるようになりました。

(4) エラーメッセージのプリフィックス

統合開発環境でのエラーヘルプ機能サポートに伴い、コンパイラ、最適化リンケージエディタのエラーメッセージの先頭にプリフィックスを付与しました。

(5) fpscr オプションの追加

cpu=sh4 オプションを指定しかつ fpu オプションを指定していないとき、関数呼び出し前後の FPSCR レジスタの精度モードを保証するかどうかを指定できるようになりました。

(6) #pragma 拡張子

#pragma 拡張子が( )なしで記述できるようになりました。

(7) 組み込み関数の追加

trace 関数を追加しました。

(8) 暗黙の宣言の追加

_ _HITACHI_ _、_ _HITACHI_VERSION_ _が暗黙に#define 宣言されます。

(9) static ラベル名

#pragma inline_asm でファイル内 static ラベルを参照できるように、ラベル名を_ _$(名前)に変更しました。ただし、リンケージリストでは_(名前)と表示されます。

(10) 言語仕様拡張・変更
・共用体初期化時のエラーを抑止します。

例：

```
union{
    char c[4];
} uu={{'a','b','c'}};
```

・構造体の代入と宣言を同時にできるようになりました。

例：

```
struct{
    int a, int b;
} s1;
void test()
{
struct S s2=s1;
}
```

・bool 型データの境界調整数が 4byte になりました。
・C++言語仕様として、例外処理やテンプレート機能もサポートしました。
・C プリプロセッサが ANSI/ISO 対応しました。

A.7 Ver.6.0 から Ver.7.0 への追加機能

SuperH RISC engine C/C++コンパイラ Ver.7.0では、最適化およびコード生成のアルゴリズムが大幅に改良されました。そのため、オプションや生成コードが Ver.6.0 までとは大きく異なります。
SuperH RISC engine C/C++コンパイラ Ver.7.0 で追加した機能概要を説明します。

(1) 外部アクセス最適化機能 (map オプションサポート)

リンク時の変数、関数の割り付けアドレスに基づいた外部変数アクセス、関数分岐命令の最適化を行います。一度目のリンク時に最適化リンケージエディタより出力(map オプション指定)された外部シンボル割り付け情報ファイルを用いてリコンパイルすることにより、最適化を実施します。

(2) GBR 相対アクセスコード自動生成（gbr オプションサポート）

gbr=auto を指定した場合、コンパイラが GBR の設定および GBR 相対アクセスコードを自動生成します。関数呼び出し前後で GBR の値を保証します。ただし、GBR 関連の組み込み関数は使用できません。

(3) speed/size 選択オプションの強化

speed/size 選択オプション(shift、blockcopy、division、approxdiv オプション)を追加し、より細かな size/speed 調整を可能にしました。

(4) 組み込み向け機能の強化

・組み込み関数の追加
倍精度乗算、SWAP 命令、LDTLB 命令、NOP 命令
・#pragma 拡張子の追加、変更
#pragma entry エントリ関数指定、SP の設定
#pragma stacksize スタックサイズの指定
#pragma interrupt sp=<変数>+<定数>、sp=&<変数>+<定数>のサポート
・セクション演算子のサポート
セクションのアドレス、サイズ参照をC 言語で記述できる機能をサポートしました。
・アドレスキャストのエラー緩和
外部変数初期化時のアドレス初期化に対するキャスト式のエラーを緩和しました。

(5) ライブラリの改善

・リエントラントライブラリサポート
ライブラリ構築ツールでreent オプションを指定した場合、リエントラントライブラリが生成されます。
・malloc 確保サイズ単位、入出力ファイル数の可変化
C/C++ライブラリ関数の初期設定において、_sbrk_sizeでmalloc 確保サイズを、_nfilesで入出力ファイル数を指定できるようになりました。これにより、RAM 容量を節約できます。
指定を省略した場合、malloc 確保サイズは520、入出力ファイル数は20 になります。
・簡易I/O のサポート
ライブラリ構築ツールでnofloat オプションを指定した場合、浮動小数点変換をサポートしない、サイズの小さいI/O ルーチンを生成します。

(6) 最適化オプションの追加（V7.0.06）

・追加オプション
V7.0.06で追加したオプション一覧を以下に示します。英大文字は短縮形指定時の文字を、下線は省略時解釈を示します。

表 B.5　追加オプション一覧

	項目	オプション名	内容
1	外部変数の volatile 化	GLOBAL_Volatile = { 0 | 1 }	外部変数について volatile 宣言されたものとして扱わない(volatile 宣言されたものは除く) すべての外部変数について volatile 宣言されたものとして扱う
2	外部変数最適化範囲指定	OPT_Range = { All | NOLoop | NOBlock }	関数内の全範囲で外部変数を最適化対象とする ループ制御変数やループ内の外部変数のループ外への移動を抑止 ループや、分岐をまたいだ外部変数に対する最適化をすべて抑止
3	空ループ削除	DEL_vacant_loop ={ 0 | 1}	空ループ削除を抑止 空ループ削除を行う
4	ループ最大展開数の指定	MAX_unroll = <数値> <数値>:1-32	ループ展開時の最大展開数を指定 default：1 (speed,loop 指定時 2)
5	無限ループ前の式削除	INFinite_loop = { 0 | 1 }	無限ループ前の外部変数定義削除を抑止 無限ループ前の外部変数定義削除を行う
6	外部変数のレジスタ割り付け	GLOBAL_Alloc = { 0 | 1 }	外部変数のレジスタ割り付けを抑止 外部変数のレジスタ割り付けを行う
7	構造体/共用体メンバのレジスタ割り付け	STRUCT_Alloc = { 0 | 1 }	構造体／共用体メンバのレジスタ割り付けを抑止 構造体／共用体メンバのレジスタ割り付けを行う
8	const 定数伝播	CONST_Var_propagate = { 0 | 1 }	const 宣言された外部変数の定数伝播を抑止 const 宣言された外部変数の定数伝播を行う
9	定数ロードの命令展開	CONST_Load = { Inline | Literal }	定数ロードを命令展開 定数ロードをリテラルアクセス default：size 指定時 2-3 命令まで展開
10	命令並べ替え	SChedule = { 0 | 1 }	命令並べ替えを行わない 命令並べ替えを行う

GLOBAL_Volatile

Optimize[Details][Global variables][Treat global variables as volatile qualified]

書　式　GLOBAL_Volatile = { 0 | 1 }

説　明　global_volatile=0 を指定した場合、volatile 宣言のない外部変数に対して最適化を行います。したがって、外部変数のアクセス回数、アクセス順序が C/C++プログラムで記述した場合と異なることがあります。
global_volatile=1 を指定した場合、すべての外部変数を volatile 宣言したものとして扱います。したがって、外部変数のアクセス回数、アクセス順序は C/C++で記述したとおりになります。
本オプション省略時解釈は、global_volatile=0 です。

備　考　global_volatile=1 を指定した場合、schedule=0 がデフォルトになります。

OPT_Range

Optimize[Details][Global variables][Specify optimizing range :]

書　式　　OPT_Range = { All | NOLoop | NOBlock }

説　明　　opt_range=all を指定した場合、関数内の全範囲を対象に外部変数に対する最適化を行います。
opt_range=noloop を指定した場合、ループ内にある外部変数やループ判定式で使用されている外部変数を最適化の対象外にします。
opt_range=noblock を指定した場合、分岐をまたいだ外部変数の最適化（ループを含む）をすべて抑止します。
本オプション省略時解釈は、opt_range=all です。

例　　(1) 分岐をまたいだ最適化例(opt_range=all/noloop 指定時に行う)

```
int A,B,C;
void f(int a) {
    A = 1;
    if (a) {
        B = 1;
    }
    C = A;
}
```

＜最適化後のソースイメージ＞

```
void f(int a) {
    A = 1;
    if (a) {
        B = 1;
    }
    C = 1;      /* A の参照を削除し、A=1 を伝播する */
}
```

(2) ループにおける最適化例(opt_range=all 指定時に行う)

```
int A,B,C[100];
void f() {
    int i;
    for (i=0;i<A;i++) {
        C[i] = B;
    }
}
```

＜最適化後のソースイメージ＞

```
void f() {
    int i;
    int temp_A, temp_B;
    temp_A = A; /* ループ判定式の A の参照をループ外に移動          */
    temp_B = B; /* ループ内の B の参照をループ外に移動              */
    for (i=0;i<temp_A;i++) {      /* A のループ内での参照を削除 */
        C[i] = temp_B;            /* B のループ内での参照を削除 */
    }
}
```

備　考　　opt_range=noloop を指定した場合、常に max_unroll=1 がデフォルトになります。
opt_range=noblock を指定した場合、常に max_unroll=1、const_var_propagate=0、global_alloc=0 がデフォルトになります。

空ループ削除

DEL_vacant_loop

Optimize[Details][Miscellaneous][Delete vacant loop]

書　式　DEL_vacant_loop = { 0 | 1 }

説　明　del_vacant_loop=0 を指定した場合、ループ内処理がない場合でもループを削除しません。
del_vacant_loop=1 を指定した場合、ループ内処理がないループは削除します。
本オプション省略時解釈は、del_vacant_loop=0 です。

備　考　SHC/C++V7 台(V7.0.04 まで)のコンパイラとはデフォルトが異なりますのでご注意ください。
V7.0.04 まで ：空ループを削除する
V7.0.06 ：空ループを削除しない

ループ最大展開数の指定

MAX_unroll

Optimize[Details][Miscellaneous][Specify maximum unroll factor :]

書　式　MAX_unroll = <数値>

説　明　ループ展開時の最大展開数を指定します。<数値>には 1 から 32 までの整数を指定することができます。それ以外の値を指定した場合はエラーになります。
本オプション省略時解釈は、speed または loop オプションを指定した場合は max_unroll=2、それ以外の場合は max_unroll=1 です。

備　考　opt_range=noloop/noblock を指定した場合、常に max_unroll=1 がデフォルトになります。

INFinite_loop

Optimize[Details][Global variables]
[Delete assignment to global variables before an infinite loop]

書　式　INFinite_loop = { 0 | 1 }

説　明　infinite_loop=0 を指定した場合、無限ループ直前の外部変数への代入式を削除しません。
infinite_loop=1 を指定した場合、無限ループ直前にあり無限ループ内で参照されない外部変数への代入式を削除します。
本オプション省略時解釈は、infinite_loop =0 です。

例

```
int A;
void f()
{
   A = 1;       /* 外部変数 A への代入式       */
   while(1) {}  /* A は参照されない    */
}
```

＜infinite_loop=1 指定時のイメージ＞

```
void f()
{
                /* 外部変数 A への代入式を削除 */
   while(1) {}
}
```

備　考　SHC/C++V7 台(V7.0.04 まで)のコンパイラとはデフォルトが異なりますのでご注意ください。
V7.0.04 まで ：無限ループ直前のループ内で参照されない外部変数への代入式を削除する
V7.0.06 ：無限ループ直前の外部変数への代入式を削除しない

GLOBAL_Alloc

Optimize[Details][Global variables][Allocate registers to global variables :]

書　式　GLOBAL_Alloc = { 0 | 1 }

説　明　global_alloc=0 を指定した場合、外部変数のレジスタ割り付けを抑止します。
global_alloc=1 を指定した場合、外部変数のレジスタ割り付けを行います。
本オプション省略時解釈は、global_alloc=1 です。

備　考　opt_range=noblock を指定した場合、global_alloc=0 がデフォルトになります。
optimize=0 を指定した場合、SHC/C++V7 台(V7.0.04 まで)のコンパイラとはデフォルトが異なりますのでご注意ください。
V7.0.04 まで ：外部変数のレジスタ割り付けを行う
V7.0.06 ：外部変数のレジスタ割り付けを抑止する

STRUCT_Alloc

Optimize[Details][Miscellaneous][Allocate registers to struct/union members]

書　式　STRUCT_Alloc = { 0 | 1 }

説　明　struct_alloc=0 を指定した場合、構造体/共用体メンバのレジスタ割り付けを抑止します。
struct_alloc=1 を指定した場合、構造体/共用体メンバのレジスタ割り付けを行います。
本オプション省略時解釈は、struct_alloc=1 です。

備　考　opt_range=noblock もしくは global_alloc=0 を指定しかつ struct_alloc=1 を指定した場合、ローカル構造体/共用体メンバのみレジスタ割り付けを行います。
optimize=0 を指定した場合、SHC/C++V7 台(V7.0.04 まで)のコンパイラとはデフォルトが異なりますのでご注意ください。
V7.0.04 まで ：ローカル構造体/共用体メンバのレジスタ割り付けを行う
V7.0.06 ：ローカル構造体/共用体メンバのレジスタ割り付けを抑止する

CONST_Var_propagate

Optimize[Details][Global variables][Propagate variables which are const qualified :]

書　式　CONST_Var_propagate = { 0 | 1 }

説　明　const_var_propagate=0 を指定した場合、const 宣言された外部変数の定数伝播を抑止します。
const_var_propagate=1 を指定した場合、const 宣言された外部変数についても定数伝播を行います。
本オプション省略時解釈は、const_var_propagate=1 です。

例

```
const int X = 1;
int A;
void f() {
   A = X;
}
```

＜const_var_propagate =1 指定時のソースイメージ＞

```
void f() {
   A = 1;      /* X=1 を伝播 */
}
```

備　考　opt_range=noblock を指定した場合、const_var_propagate=0 がデフォルトになります。
C++プログラムで const 宣言された変数については本オプションで制御することはできません(常に定数伝播されます)。

CONST_Load

Optimize[Details][Miscellaneous][Load constant value as :]

書　式　CONST_Load = { Inline | Literal }

説　明　const_load=inline を指定した場合、すべての 2 バイト定数および一部の 4 バイト定数ロードを命令展開します。
const_load=literal を指定した場合、すべての 2 バイト以上の定数ロードをリテラルアクセスします。
本オプション省略時解釈は、speed オプションを指定した場合は const_load=inline、size または nospeed オプションを指定した場合は 2 バイト定数の場合は 2 命令、4 バイト定数の場合は 3 命令で展開できる場合は const_load=inline、それ以外は const_load=literal です。

例

```
int f() {
    return (257);
}
```

(1) const_load=inline または speed 指定時

```
MOV     #1,R0   ; R0 <- 1
SHLL8   R0      ; R0 <- 256 (1<<8)
RTS
ADD     #1,R0   ; R0 <- 257 (256+1)
```

(2) const_load=literal または size/nospeed 指定時

```
MOV.W   L11,R0
RTS
NOP
L11:
    .DATA.W H'0101
```

命令並べ替え

SChedule

Optimize[Details][Global variables][Schedule instructions :]

書　式　SChedule = { 0 | 1 }

説　明　schedule=0 を指定した場合、命令並べ替えを行いません。したがって C/C++記述した順番で処理を行います。
schedule=1 を指定した場合、パイプライン処理、スーパスカラ(SH-4)を考慮した命令並べ替えを行います。
本オプション省略時解釈は、schedule=1 です。

備　考　global_volatile=1 を指定した場合、schedule=0 がデフォルトになります。

optimize=0 指定時のデフォルト

optimize=0 を指定した場合、追加オプションのデフォルトは以下のようになります。

global_volatile=0

opt_range=noblock

del_vacant_loop=0

max_unroll=1

infinite_loop=0

global_alloc=0

struct_alloc=0

const_var_propagate=0

const_load=literal

schedule=0

以下のオプションについては、optimize=1 指定時とデフォルトの仕様が異なります。

	optimize=0	optimize=1
opt_range	noblock	all
global_alloc	0	1
struct_alloc	0	1
const_var_propagate	0	1
const_load	literal	speed/size/nospeed による
schedule	0	1

V7（V7.0.04 まで）との互換性

以下オプションは、V7(V7.0.04 まで)とデフォルトの仕様が異なります。

(i) 空ループ削除(del_vacant_loop)
V7.0.04まで　：空ループを削除する
V7.0.06　：空ループを削除しない

(ii) 無限ループ前の式削除(infinite_loop)
V7.0.04まで　：無限ループ前の外部変数定義を削除する
V7.0.06　：無限ループ前の外部変数定義を削除しない

また、以下については V7(V7.0.04 まで)と optimize=0 指定時の仕様が異なります。

(i) 外部変数レジスタ割り付け(global_alloc)
V7.0.04まで　：外部変数のレジスタ割り付けを行う
V7.0.06　：外部変数のレジスタ割り付けを抑止

(ii) 構造体/共用体メンバレジスタ割り付け(struct_alloc)
V7.0.04まで　：構造体/共用体メンバのレジスタ割り付けを行う
V7.0.06　：構造体/共用体メンバのレジスタ割り付けを抑止

オプション体系

外部変数に対する最適化を以下のレベルで設定しました。HEW で以下のレベルを設定することにより、当該オプションを一括制御することができます。
設定は、Optimize[Details][Level :]で行います。

(i) Level 1
外部変数の最適化をすべて抑止する。
global_volatile=1
opt_range=noblock
infinite_loop=0
global_alloc=0
const_var_propagate=0
schedule=0

(ii) Level 2
volatile指定のない外部変数を分岐(ループを含む)を超えない範囲で最適化する。
global_volatile=0
opt_range=noblock
infinite_loop=0
global_alloc=0
const_var_propagate=0
schedule=1

(iii)Level 3
volatile指定のない外部変数をすべて最適化対象とする。
global_volatile=0
opt_range=all
infinite_loop=0
global_alloc=1
const_var_propagate=1
schedule=1

(iv)Custom
外部変数に対する最適化をプログラムにあわせてユーザが指定する。

Level 1～3 を指定した場合、上記オプションについては変更することはできません。

（7）以降は最適化リンケージエディタで追加した機能です。

(7) ワイルドカードのサポート

入力ファイルや start オプションのセクション名でワイルドカードを指定できます。

(8) サーチパス

環境変数(HLNK_DIR)により、複数の入力ファイル、ライブラリファイルのサーチパスを指定できます。

(9) ロードモジュール分割出力

アブソリュートロードモジュールファイルを分割出力できます。

(10) エラーレベルの変更

インフォメーション、ウォーニング、エラーレベルのメッセージは、個別にエラーレベルや出力有無を変更できます。

(11) バイナリ、HEX サポート

バイナリファイルを入出力できるようになりました。
また、インテル HEX タイプの出力も選択できるようになりました。

(12) stack 使用量情報の出力

stack オプションにより、スタック解析ツール用情報ファイルを出力できます。

(13) デバッグ情報削除機能

strip オプションにより、ロードモジュールファイルやライブラリファイル内のデバッグ情報だけを削除できます。
SuperH RISC engine C/C++コンパイラ Ver.7.1 最適化リンケージエディタで追加した機能概要を説明します。

(14) 外部シンボル割付情報ファイルの出力（map オプションサポート）

map オプションを指定した場合、コンパイラが外部変数アクセス最適化で使用する外部シンボル割り付け情報ファイルを生成します。

A.8 Ver.7.0 から Ver.7.1 への追加機能

SuperH RISC engine C/C++コンパイラ Ver.7.1 で追加した機能概要を説明します。

最適化の強化

(a) MOVT 直後の EXTU 削除

MOVT の直後の不要な EXTU を削除する。
（1または0以外が設定されることはないので EXTU は不要）

最適化前	最適化後
_f: MOV.L L12+2,R6 ; _a1 MOV.B @R6,R0 TST #128,R0 MOVT R0 EXTU.B R0,R0	_f: MOV.L L12+2,R6 ; _a1 MOV.B @R6,R0 TST #128,R0 MOVT R0
R0 には 1 または 0 以外は設定されないので EXTU は不要。	

(b) ゼロ拡張したレジスタの右シフト後の EXTU 削除

ゼロ拡張後のレジスタを右シフトした後にゼロ拡張しても値は変わらないため削除する。

最適化前	最適化後

_f:
MOV.L L13+2,R2; _a2
MOV #1,R5
MOV.W @R2,R6
EXTU.W R6,R6
MOV R6,R2
SHLR2 R2
SHLR R2
EXTU.W R2,R2
CMP/GE R5,R2
:
	_f:
	MOV.L L13+2,R2; _a2
	MOV #1,R5
	MOV.W @R2,R6
	EXTU.W R6,R6
	MOV R6,R2
	SHLR2 R2
	SHLR R2
	CMP/GE R5,R2
	:

EXTU で上位 2 バイトはゼロクリアされているので再度 EXTU を行っても値は変わらない。	

(c) 連続した AND の統合

同じ変数への AND が連続した場合、1つの AND にまとめる。

最適化前	最適化後
_f: MOV.L L11+2,R6 ; _a5 MOV.B @R6,R0 AND #3,R0 RTS AND #1,R0	_f: MOV.L L11+2,R6 ; _a5 MOV.B @R6,R0 RTS AND #1,R0
AND を1つにまとめる。	

(d) ビットフィールド比較結合

複数のビットフィールドの判定(TST #n, R0)を統合する。

最適化前	最適化後

_f:
:
MOV R4,R0
TST #64,R0
BF L12
TST #32,R0
BF L12
MOV R6,R0
:
	_f:
	:
	MOV R4,R0
	TST #96,R0
	BF L12
	MOV R6,R0
	:

ビットフィールドの判定式を統合して 1 回の判定に置き換える。	

(e) 連続した EXTS＋EXTU の EXTS 削除

EXTS 後に同じサイズの EXTU を行った場合、EXTS は不要なため削除する。

最適化前	最適化後
_f: : EXTS.B R6,R2 EXTU.B R2,R0 :	_f: : EXTU.B R6,R0 :
EXTS **した値に** EXTU **しているので** EXTS **は不要。**	

(f) MOVT(+XOR)+EXTU+CMP/EQ の削除

TST 後の不要な MOVT(+XOR)+EXTU+CMP/EQ を削除し、T ビットを直接分岐命令で参照するように変換する。

最適化前	最適化後
_f: : TST #4,R0 MOVT R0 MOV.L L23+6,R6 ; _st2 XOR #1,R0 EXTU.B R0,R0 CMP/EQ #1,R0 MOV.B @R6,R0 BF L16 :	_f: : TST #4,R0 MOV.L L23+6,R6; _st2 MOV.B @R6,R0 BT L16 :
直接 T ビットを参照。	

(g) AND #imm,R0+CMP/EQ #imm,R0->TST #imm,R0

AND #imm,R0+CMP/EQ #imm,R0 を TST #imm,R0 に置き換える。

最適化前	最適化後
L17: MOV.B @R6,R0 AND #1,R0 CMP/EQ #1,R0 BF L19 MOV.B @R5,R0 AND #1,R0	L17: MOV.B @R6,R0 TST #1,R0 BT L19 MOV.B @R5,R0 AND #1,R0

(h) unsigned char と定数の比較(==)時の EXTU 削除

ロード直後の unsigned char と定数の比較時の不要な EXTU を削除する。

最適化前	最適化後
_f: MOV.L L11,R6 ; _b MOV.B @R6,R2 MOV #-128,R6; H'FFFFFF80 EXTU.B R6,R6 EXTU.B R2,R2 CMP/EQ R6,R2 MOVT R2 MOV.L L11+4,R6 ; _a RTS MOV.B R2,@R6	_f: MOV.L L11,R6 ; _b MOV.B @R6,R2 MOV #-128,R6; H'FFFFFF80 CMP/EQ R6,R2 MOVT R2 MOV.L L11+4,R6 ; _a RTS MOV.B R2,@R6
不要な拡張を削除。	

(i) ビットフィールド LOAD 後/STORE 前の拡張削除

ビットフィールドの LOAD 後/STORE 前の不要な拡張を削除する。

最適化前	最適化後
_f: MOV.L L11+2,R6;_st MOV.B @R6,R2 EXTU.B R2,R0 OR #128,R0 :	_f: MOV.L L11+2,R6;_st MOV.B @R6,R2 OR #128,R0 :
不要な拡張を削除。	

(j) switch-case の判定時のコピー削除

switch 文で case 判定ごとに行っている値のコピーを削除する。

最適化前	最適化後

_f:
:
MOV R0,R2
MOV R2,R0
CMP/EQ #1,R0
BT L24
CMP/EQ #2,R0
BT L26
MOV R2,R0
CMP/EQ #3,R0
BT L28
MOV R2,R0
CMP/EQ #4,R0
BT L30
MOV R2,R0
:
	_f:
	:
	MOV R0,R2
	MOV R2,R0
	CMP/EQ #1,R0
	BT L24
	CMP/EQ #2,R0
	BT L26
	CMP/EQ #3,R0
	BT L28
	CMP/EQ #4,R0
	BT L30
	:

不要なコピー削除。	

(k) 連続した OR の統合

同じ変数への OR が連続した場合、１つの OR にまとめる。

最適化前	最適化後
_f: MOV.L L11+2,R6 ; _a5 MOV.B @R6,R0 OR #3,R0 RTS OR #1,R0	_f: MOV.L L11+2,R6 ; _a5 MOV.B @R6,R0 RTS OR #3,,R0
OR **を１つにまとめる。**	

(l) AND #imm,R0 または TST #imm,R0 直前の EXTS 削除

(i)AND #imm,R0
(ii)TST #imm,R0
直前の不要な拡張を削除する。

最適化前	最適化後
_f: : EXTS.B R6,R0 AND #32,R0 :	_f: : AND #32,R0 :
_f: : EXTS.B R6,R0 TST #32,R0 :	_f: : TST #32,R0 :
不要な拡張を削除。	

(m) 連続した EXTU+EXTS の EXTU 削除

EXTU 後に同じサイズの EXTS を行った場合、EXTU は不要なため削除する。

最適化前	最適化後
_f: : EXTU.B R6,R2 EXTS.B R2,R0 :	_f: : EXTS.B R6,R0 :
EXTU **した値に** EXTS **しているので** EXTU **は不要。**	

(n) MOVT 後の XOR #imm,R0(OR,AND)直後の EXTU 削除

MOVT 後の
(i) XOR #imm,R0
(ii) OR #imm,R0
(iii) AND #imm,R0
直後の不要な EXTU を削除する。

最適化前	最適化後
: MOVT R0 XOR #1,R0 RTS EXTU.B R0,R0	: MOVT R0 RTS XOR #1,R0
MOVT R0 OR #2,R0 RTS EXTU.B R0,R0	: MOVT R0 RTS OR #2,R0
: MOVT R0 AND #1,R0 RTS EXTU.B R0,R0	: MOVT R0 RTS AND #1,R0
不要な拡張を削除。	

(o) 比較時の不要 EXTS 削除

符号拡張後のレジスタに対し、比較時に再度出力される冗長な EXTS を削除する。

最適化前	最適化後
_f: : EXTS.B R6,R6 CMP/GT R6,R2 BF L13 :	_f: : CMP/GT R6,R2 BF L13 :
R6 がすでに前で拡張済みの場合は EXTS は不要。	

(p) 定数値へのレジスタ割付（即値）抑止

関数引数の定数（-128～127）がレジスタに割り付くのを抑止する。

最適化前	最適化後
_f: PUSH R14 : MOV.B #127,R14 : MOV.B R14,R4 BSR sub : POP R14	_f: : : MOV.B #127,R4 BSR sub :
#127 をレジスタに割り付けず直接パラメータレジスタにロードする。	

(q) DT 命令強化

レジスタに割りついた変数に対して DT 命令化を行う。

最適化前	最適化後
_f: MOV.L L16+2,R6; _x MOV.L @R6,R2 ADD #-1,R2 TST R2,R2 BT/S L12 :	_f: MOV.L L16+2,R6; _x MOV.L @R6,R2 DT R2 BT/S L12 :
DT 命令化を行う。	

(r) リテラル出力位置改善

リテラルデータ出力位置決定の際の命令サイズ計算の精度をあげ、リテラルデータ出力位置を後方に出力可能にする。

(s) 1byte&=1byte の冗長 EXTU 削除

1byte&=1byte 時の不要な EXTU を削除する。

最適化前	最適化後
_f: : MOV.B @(R0,R7),R6 MOV.B @R5,R2	_f: : MOV.B @(R0,R7),R6 MOV.B @R5,R2

EXTU.B R6,R6 AND R6,R2 MOV.B R2,@R5 MOV.B @R14,R2 :	AND R6,R2 MOV.B R2,@R5 MOV.B @R14,R2 :
不要な拡張を削除する。	

(t) 2byte リテラルの展開

同じコードが 2 回展開されるのを改善する。

最適化前	最適化後

_f:
MOV.L L13+4,R4 ; _b
SHLL8 R0
ADD #-48,R0
MOV.W @(R0,R4),R2
MOV #8,R0
SHLL8 R0
ADD #-46,R0
EXTU.W R2,R6
MOV.W @(R0,R4),R2
MOV #8,R0
SHLL8 R0
ADD #-44,R0
EXTU.W R2,R5
MOV.W @(R0,R4),R2
	_f:
	MOV.L L13+4,R4 ; _b
	SHLL8 R0
	ADD #-48,R0
	MOV.W @(R0,R4),R2
	MOV #8,R0
	SHLL8 R0
	ADD #-46,R0
	EXTU.W R2,R6
	MOV.W @(R0,R4),R2
	ADD #2,R0
	EXTU.W R2,R5
	MOV.W @(R0,R4),R2

同じコードが 2 回展開されているのを改善。	

(u) ループ条件判定の展開改善

size 優先時、ループ条件判定の際のループ判定の複写を行わない。

最適化前	最適化後(v7)	最適化後(v7.1)

while (cond) {
:
}
	if (cond) {
	do {
	:
	} while (cond);
	}
		goto L1;
		do {
		:
		L1:;
		} while (cond);

cond **の出現が** 2 **箇所から** 1 **箇所へ。**		

(v) 冗長な if 文条件判定の除去

先の if 文の結果により後の if 文でどちらを通るか判る場合は、後の if 文を除去する。

最適化前	最適化後

if (cond)
t=65;
else
t=67;
if (t == 65)
fx();
else
fy();
	if (cond) {
	t=65;
	fx();
	} else {
	t=67;
	fy();
	}

最初の if 文の結果により 2 つ目の if 文でどちらかをとおるかがわかるので 2 つ目の if 文を除去する。	

(w) テンポラリ変数への直接演算

冗長な temp 変数への代入を抑止し、式の演算順序を変更する。

最適化前	最適化後
k = i + prime; p = flags + k;	p = i + prime + flags;
k **はこの後使用されない -> 無駄な** temp **への代入を行わない。**	

(x) ポストインクリメントのアドレッシング

４バイト変数の LOAD について MOV.L @Rm+,Rn を活用する。

最適化前	最適化後

:
L11:
MOV.L @R5,R2
ADD #4,R5
DT R6
ADD R2,R4
BF L11
:
	:
	L11:
	MOV.L @R5+,R2
	DT R6
	ADD R2,R4
	BF L11
	:

MOV.L @Rm+,Rn **一命令で実行。**	

(y) ループ解消条件改善

ループ解消最適化を行う条件を緩和し、最適化がかかりやすくする。

最適化前	最適化後

int a, b;
func() {
unsigned short sx;
 	
for (sx=0; sx<1; sx++) {
a++;
b++;
f();
}
}
	int a, b;
	func() {
	
	a++;
	b++;
	f();
	
	}

ループ解消を行う。	

(z) １ビット判定の最適化

１ビット幅のビットフィールドを複数個参照する条件式を１つにまとめ、ビット AND を用いて値の取り出しと比較を同時に行うコードを生成する。

最適化前	最適化後

struct S {
char bit0:1;
char bit1:1;
char bit2:1;
char bit3:1;
}ss1;
if ((ss1.bit0|ss1.bit1|ss1.bit2)!= 0){
:
:
}
	struct S {
	char bit0:1;
	char bit1:1;
	char bit2:1;
	char bit3:1;
	}ss1;
	if ((*(char *)&ss1 & 0xe0)!= 0){
	:
	:
	}

AND を使用し、取り出しと比較を同時に行う。	

A.9 Ver.7.1 から Ver.8.0 への追加機能

SuperH RISC engine C/C++コンパイラ Ver.8.0 で追加した機能概要を説明します。

(1) 新 CPU のサポート

SH-4A, SH4AL-DSP をサポートしました。

(2) 言語仕様拡張・変更

・DSP-C言語をサポートしました。
・long long、unsigned long long 型をサポートしました。

(3) 組み込み向け機能の強化

・DSP向け組み込み関数の追加
絶対値、MSB検出、算術シフト、丸め演算、ビットパターンコピー、モジュロアドレッシング設定、モジュロアドレッシング解除、CSビットの設定
・SH-4A、SH4AL-DSP向け組み込み関数の追加
正弦・余弦の計算、平方根の逆数、命令キャッシュブロックの無効化、命令キャッシュブロックのプリフェッチ、データ操作の同期
・#pragma 拡張子の追加、変更

#pragma ifunc	浮動小数点レジスタの退避／回復抑止
#pragma bit_order	ビットフィールド並び順指定
#pragma pack	構造体、共用体、クラスの境界調整数指定

(4) 列挙型サイズの自動選択(auto_enum オプションサポート)

列挙型が収まる最小型として列挙型を処理します。

(5) 構造体、共有体、クラスメンバの境界調整数を指定(pack オプションサポート)

構造体、共有体、クラスメンバの境界調整数を指定できます。

(6) ビットフィールド並び順指定(bit_order オプションサポート)

ビットフィールドのメンバの並び順を指定できます 。

(7) エラーレベルの変更(change_message オプションサポート)

インフォメーション、ウォーニングレベルのメッセージは、個別にエラーレベルを変更できます。

(8) 制限値の緩和

switch 文の数を 2048 個に拡張しました。

(9) DSP ライブラリの固定小数点サポート

DSP ライブラリの固定小数点をサポートしました。

(10) ファイル間インライン展開機能(File_inline オプションサポート)

ファイル間に跨った関数インライン展開を指定できるようになりました。

(11) セクション内データの詰め込み配置(Data_dtuff オプションサポート)

コンパ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する機能をサポートしました。本機能を指定することで、データセクション全体のサイズ低減が期待できます。

A.10 Ver.8.0 から Ver.9.0 への追加機能

SuperH RISC engine C/C++コンパイラ Ver.9.0 で追加した機能概要を説明します。

(1) 新 CPU のサポート

SH-2A、SH2A-FPU をサポートしました。
また、SH-2A、SH2A-FPU の TBR を活用する機能(tbr オプション、#pragma tbr)を追加しました。

(2) 言語仕様拡張・変更

・以下の項目をANSI 準拠しました。
・配列のインデックス

```
int iarray[10], i=3;
i[iarray] = 0; /* iarray[i] = 0;と同じになる */
```

・union のビットフィールド指定を可能にする

```
union u {
    int a:3;
};
```

・定数演算

```
static int i=1||2/0; /* ゼロ除算をError からWarning に変更 */
```

・ライブラリ、マクロ追加

```
strtoul,FOPEN_MAX
```

・strict_ansi オプションを指定することにより、以下項目がANSI 準拠になります。
これにより、Ver.9 における結果が、Ver.8 までの結果と異なる場合があります。
・ unsigned int とlong 型演算
unsigned int + long の演算
[strict_ansi なし]
unsigned int → long に型変換
long → 型変換なし
[strict_ansi あり]
unsigned int → unsigned long に型変換
long → unsigned long に型変換

・ 浮動小数点演算の結合則
1.0 + 2.0 + 3.0 の演算
[strict_ansi なし]
・3.0+2.0を先に演算し、最後に1.0を足す
・1.0+2.0を先に演算し、最後に3.0を足す
最適化によって、二通りの結合則がある。
→ 最適化により演算誤差が変わる
[strict_ansi あり]
・3.0+2.0を先に演算し、最後に1.0を足す
最適化によって、結合則が変わらない
→ 演算誤差が最適化によって変わることが無い

・enable_register オプションを指定することにより、register 記憶クラスを指定した変数を優先的にレジスタに割り付けます。

(3) 組み込み向け機能強化

・SH-2A、SH2A-FPU 向け組み込み関数の追加
　飽和演算、TBR 設定・参照

・C 言語で記述できない命令の組み込み関数サポート
　T ビット参照・設定、連結レジスタの中央切り出し、キャリ付き加算、ボロー付き減算、符号反転、
　1 ビット除算、回転、シフト

(4) 制限値の緩和

以下の制限値を緩和しました。

・繰り返し文(while 文、do 文、for 文)、
　選択文(if 文、switch 文)の組み合わせによるネストの深さ：32 レベル → 4096 レベル
・1 関数内で指定可能なgoto ラベルの数：511 個 → 2147483646 個
・switch 文のネストの深さ：16 レベル → 2048 レベル
・1 つのswitch 文内で指定可能なcase ラベルの数：511 個 → 2147483646 個
・関数定義、関数呼び出しで指定可能引数：63 個 → 2147483646 個
・セクション名長：31 バイト → 8192 バイト
・1 ファイルあたりの#pragma section で指定できるセクション数：64 個 → 2045 個

(5) メモリ空間配置指定の拡張

メモリ空間配置指定をより詳細に指定できます。

・abs16/abs20/abs28/abs32 オプション
・#pragma abs16/abs20/abs28/abs32

(6) 変数の絶対アドレス指定(#pragma address サポート)

外部変数のアドレスを絶対アドレス指定できます。

(7) 外部変数アクセス最適化機能拡張(smap オプションサポート)

コンパイル対象ファイル内で定義された外部変数について外部変数アクセス最適化を実施します。
map オプションのようなリコンパイルは必要ありません。

(8) 数学関数ライブラリ精度向上

数学関数ライブラリの演算精度が向上しました。
これにより、Ver.9 における結果が、Ver.8 までの結果と異なる場合があります。

[対象ライブラリ関数]

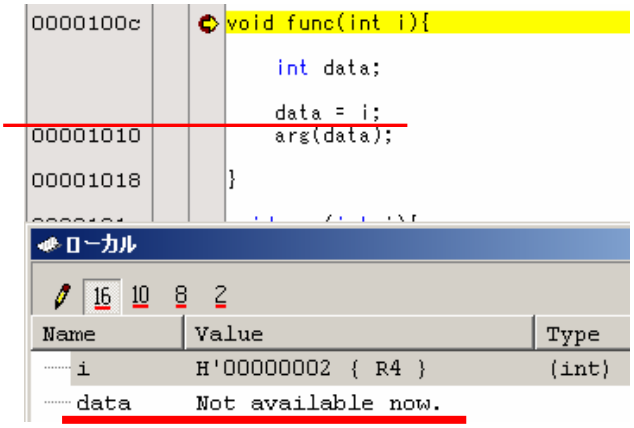
acos	acosf	asin	asinf	atan	atanf	cos	cosf	cosh	coshf	exp	expf
log	log10	log10f	logf	sin	sinf	sinh	sinhf	tan	tanf	tanh	tanhf

A.11　Ver.9.0 から Ver.9.1 への追加機能

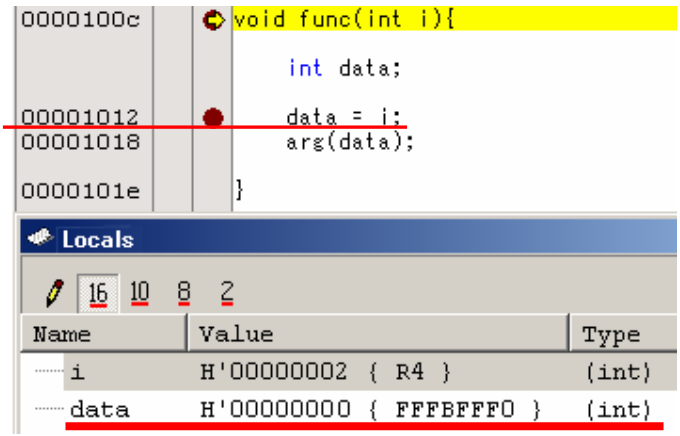
(1)　デバッグ情報出力モードの追加（optimize=debug_only）

optimize=debug_only オプションを指定することにより、デバッグ時にローカル変数の情報を常に参照できるようにしました。また、文単位の削除に関する最適化も完全に抑止し、C ソースコードの各文に break point を設定できるようにしました。　本オプションを用いてオブジェクトを生成した場合、optimize=0 (最適化なし) よりも、オブジェクト性能が低下する可能性がありますので、デバッグ時に一時的に使用することを推奨します。

[optimize=0]



[optimize=debug_only]



(2)　新割り込み仕様追加 (SH-3, SH3-DSP, SH-4, SH-4A, SH4AL-DSP)

C 言語記述で実行効率の良い割り込み関数を記述できるよう、以下の割り込み仕様および組み込み関数を追加しました。

＜割り込み仕様＞

#pragma interrupt sr_rts　-　レジスタバンク切り替え、RTS 命令リターン指定
RTS 命令で終了する。　関数内で使用したレジスタのみ退避コード生成し、
関数の最後に SR レジスタの RB ビット、BL ビットをセットする。

#pragma interrupt bank　-　割り込みハンドリング関数指定
sr_jsr() 組み込み関数が存在する場合、SSR/SPC レジスタの退避コードを生成し、
その関数内で使用したレジスタは退避コードを生成する。

#pragma interrupt rts　-　RTS 命令リターン指定
RTS 命令で終了する。SSR/SPC、R0～R7 レジスタの退避コードの出力を抑止。
その関数内で使用したレジスタは退避コードを生成する。

＜組み込み関数＞

sr_jsr(void * func, int imask)　-　多重割り込み制御用の組み込み関数
SR レジスタの RB ビット､BL ビットをクリアし､割り込みマスクを imask に設定した後に func 関数を呼び出す。

＜使用例１＞　多重割り込みを許可する場合

C ソースコード	生成コード

#include <machine.h>
 	
// ハンドリング関数宣言
#pragma interrupt (func(bank))
void func(void);
 	
// 割り込み関数宣言
#pragma interrupt (sub (sr_rts))
void sub(void);
 	
// 関数定義
void func(void){
　　:
/* RB=0,RL=0
　割り込みレベルを 8 にして
　sub() を呼び出す */
 	
　sr_jsr(sub,8);
　　:
}
 	
void sub(void){
　　:
}
	_func
	　　← 割り込み発生時、RB=1,BL=1
	　MOV.L　R14,@-R15
	　STS.L　PR,@-R15
	　STC　　SSR,@-R15
	　STC　　SPC,@-R15　} 関数内で使用した R0～R7 以外のレジスタと SPC/SSR の退避
	　　:
	　STC　　SR,R6
	　MOV.L　L12+6,R1　; H'CFFFFF0F
	　MOV　　#-128,R4　; H'FFFFFF80
	　EXTU.B　R4,R4
	　MOV.L　L12+10,R14 ; _sub　} RB=0、BL=0 IMASK=8 に変更
	　AND　　R1,R6
	　OR　　R4,R6
	　LDC　　R6,SR
	　JSR　　@R14　← sub()関数呼び出し RB=1,BL=1 に変更されて戻ってくる
	　NOP
	　　:
	　LDC　　@R15+,SPC
	　LDC　　@R15+,SSR
	　LDS.L　@R15+,PR　} 関数内で使用した R0～R7 以外のレジスタと SPC/SSR の回復
	　MOV.L　@R15+,R14
	　RTE
	　NOP
	
	_sub:　← func()から呼び出される時、RB=0,BL=0 に設定されている
	　MOV.L　R0,@-R15
	　MOV.L　R1,@-R15　} 関数内で使用したレジスタのみ退避
	　　:
	　STC　　SR,R0
	　MOV.L　L12+2,R1 ; H'30000000
	　OR　　R1,R0
	　MOV.L　@R15+,R1
	　LDC　　R0,SR　} RB=1,BL=1 に変更し、関数内で使用したレジスタを回復
	　RTS
	　LDC.L　@R15+,R0_BANK　← RB=1 に変更後のため

＜使用例 2＞　多重割り込みを許可しない場合

C ソースコード	生成コード

// ハンドリング関数宣言
#pragma interrupt (func(bank))
void func(void);
 	
// 割り込み関数宣言
#pragma interrupt (sub (rts))
void sub(void);
 	
// 関数定義
void func(void){
　　:
　sub();
　　:
}
 	
void sub(void){
　　:
}
	_func
	　　← 割り込み発生時、RB=1、BL=1
	　STS.L　PR,@-R15　} 関数内で使用した R0～R7 以外のレジスタの退避 sr_jsr()組み込み関数が無い為、SPC/SSR は退避せず。
	　　:
	　MOV.L　L12,R14 ; _sub
	　JSR　　@R14
	　NOP
	　　:
	　LDS.L　@R15+,PR
	　RTE　} 関数内で使用した R0～R7 以外のレジスタの回復
	　NOP
	
	_sub:
	
	　MOV.L　R14,@-R15
	　MOV.L　R13,@-R15　} 関数内で使用した R0～R7 以外のレジスタの退避
	　　:
	　MOV.L　@R15+,R13
	　RTS
	　MOV.L　@R15+,R14　} 関数内で使用した R0～R7 以外のレジスタの回復

(3) 浮動小数点数-整数変換時の範囲チェック省略機能の追加 - simple_float_conv オプション (SH-2E, SH2A-FPU, SH-4, SH-4A)

simple_float_conv オプションを指定することにより、符号なし整数型と浮動小数点型の間の型変換に対して、変換対象の値の範囲チェックを省略したコードを生成できるようになりました。

本オプションは、型変換前の値が 0～2147483647 の整数値、もしくは 0.0～2147483647.0 の浮動小数点値の時に使用できます。型変換前の値が範囲外の場合は変換結果を保証できませんのでご注意下さい。

<コード生成例>

C ソースコード

```
unsigned long func(float f)
{
      return((unsigned int)f);
}
```

オプション未使用時

```
  MOV       #79,R2     ; 0x0000004F
  SHLL8     R2
  SHLL16    R2         ; 0x4F000000
  LDS       R2,FPUL
  FSTS      FPUL,FR8
  FCMP/GT   FR4,FR8
  BT        L12
  FADD      FR8,FR8    ; f ≧ 0x4F000000 の場合、
  FSUB      FR8,FR4    ; 変換前の値を(f - 0x4F800000)とする

L12:
  FTRC FR4,FPUL        ; floatからsigned longへの変換
  STS FPUL,R0
```

オプション使用時

```
  FTRC      FR4,FPUL   ; float型 から signed long 型への変換
  STS       FPUL,R0
```

(4) 既存機能の仕様追加・変更

(a) division=cpu=inline オプションの拡張 (SH-2A, SH2A-FPU)

cpu=sh2a|sh2afpu の場合も、division=cpu=inline を使用できるようになりました。
SH-2A,SH2A-FPU に対して division=cpu=inline オプションを指定した場合、定数除算は乗算に変換してインライン展開し、変数除算は DIVS 命令または DIVU 命令を使用します。このことにより、定数除算の演算速度向上が見込めます。ただし、オブジェクトサイズは増加する恐れがありますのでご注意下さい。speed、nospeed オプション(最適化：スピード優先、サイズ&スピード) を指定している場合、division=cpu=inline オプションがデフォルトオプションとなります。

<コード生成例>

C ソースコード

```
unsigned long A;

int func(void){

     A = A/10;

}
```

cpu=sh2a , division=cpu=inline

```
_func:
    MOV.L     L11,R5     ; _A
    MOV.L     L11+4,R1   ; H'CCCCCCCD
    MOV.L     @R5,R6     ; A
    DMULU.L   R6,R1
    STS       MACH,R2
    SHLR2     R2
    SHLR      R2
    MOV.L     R2,@R5     ; A
    RTS
    MOV.L     R2,@R5     ; A
```

サイクル数 ： 12サイクル
サイズ ： 0x1C

cpu=sh2a , division=cpu=runtime

```
_func:
    MOV.L     L11,R5     ; _A
    MOV.L     @R5,R6     ; A
    MOV       #10,R0     ; H'0000000A
    DIVU      R0,R6
    RTS
    MOV.L     R6,@R5     ; A
```

サイクル数 ： 38サイクル
サイズ ： 0x10

(b) キャッシュブロック操作組み込み関数の対象拡張 (SH-4)

キャッシュブロック操作の為の組み込み関数 ocbi(), ocbp(), ocbwb() を、SH-4 でも使用できるようにしました。

(c) #pragma inline の仕様変更

#pragma inline 指定を行った関数に対する inline オプションの仕様を変更しました。

＜従来仕様＞

inline オプションが指定されている場合、#pragma inline を指定した関数は inline オプションの設定値に従う。

(補足)

speed オプション(スピード優先最適化)が指定されている時は inline=20 がデフォルトで指定されます。

＜新仕様＞

#pragma inline を指定した関数は inline オプションの設定値に関係なくインライン展開を行う。

(d) コンパイルリストへのサブコマンド内オプションの表示

コンパイル時にサブコマンドファイルが指定されている場合、そのサブコマンドファイル内で指定されているオプションをコンパイルリストに出力するように変更しました。これにより、ルネサス統合開発環境(High-performance Embedded Workshop)を用いてコンパイルリストを出力した場合、コンパイルリスト内にコンパイルオプションが出力されます。

(5) 数学関数ライブラリ性能改善 (SH-1, SH-2, SH2-DSP, SH-2A, SH-3, SH3-DSP, SH4AL-DSP)

浮動小数点数用の数学関数 sinf, cosf, tanf, expf, logf, sqrtf, atanf について、オブジェクトサイズが削減され、演算速度及び演算精度が向上しました。

尚、**演算精度が向上した事により、これら数学関数の演算結果が Ver.9.00 と異なる場合がありますのでご注意下さい。**

ベンチマーク測定比較　V.9.00 / V.9.01　(サイクル数)

ライブラリ関数	V.9.00 – SH-2	V.9.01– SH-2	V.9.00 – SH-2A	V.9.01– SH-2A
sinf	2497	477	1001	169
cosf	2434	465	954	162
tanf	3196	705	1806	274
atanf	3160	515	1602	218
logf	3816	491	1720	232
sqrtf	1018	236	562	109
expf	4432	469	1463	192

(6) デバッグ情報の改善

(a) 引数の参照

関数入り口地点で引数が”Not available now” と表示されるケースを改善しました。

(b) #pragma address 指定変数

#pragma address を使用した変数をシンボルとして参照できるよう改善しました。

(c) 不要な型情報の削除

C/C++ソースファイル内で参照しない型のデバッグ情報を削除し、オブジェクトファイルのサイズを削減しました。

(7) メッセージの改善

(a) 未初期化変数に対するインフォメーションメッセージ

C ソースコードで未初期化変数を使用している時、「C0011 (I) Used before set symbol : “変数名” in “関数名”」を出力するように改善しました。C++ソースコードでは、「C5549 (I) Variable "変数名" is used before its value is set」を出力します。

(b) return 文のないパスに対するインフォメーションメッセージ

関数出口が複数あり、片方のパスだけ return 文がない場合にも、「C0017 (I) Missing return statement」を出力するように改善しました。

(8) 最適化リンケージエディタの主な追加・改善内容

(a) セクション単位の最適化抑止機能 (最適化リンカ V.9.01 、パッケージ V.9.00R04(a) より対応)

section_forbid オプションを使用することにより、モジュール間最適化をセクション単位で抑止することができるようになりました。

(b) オーバーレイ機能の強化 (最適化リンカ V.9.01 、パッケージ V.9.00R04(a) より対応)

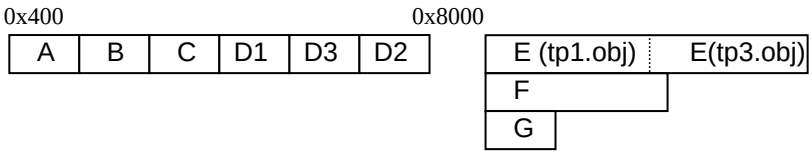
start オプションに括弧”( )” を用いた記述を導入しました。旧バージョンよりも複雑なオーバーレイ配置を記述することができるようになりました。

［例］

下記順番で.obj ファイルを入力する場合 (括弧内は各.obj が持つセクション)

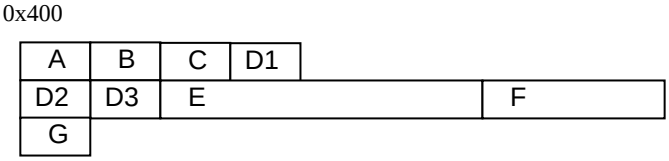
tp1.obj(A,D1,E) -> tp2.obj(B,D3,F) -> tp3.obj(C,D2,E,G)

(1) -start=A,B,C,D*/400,E:F:G/8000



- ":"で区切った E,F,G セクションは、同一アドレスに割りつきます。
- ワイルドカードで記述したセクション(ここでは D で始まる名前のセクション)は、入力した順番で割りつきます。
- 同名セクション内(ここでは E セクション)は、入力したオブジェクトから順番に割りつきます。

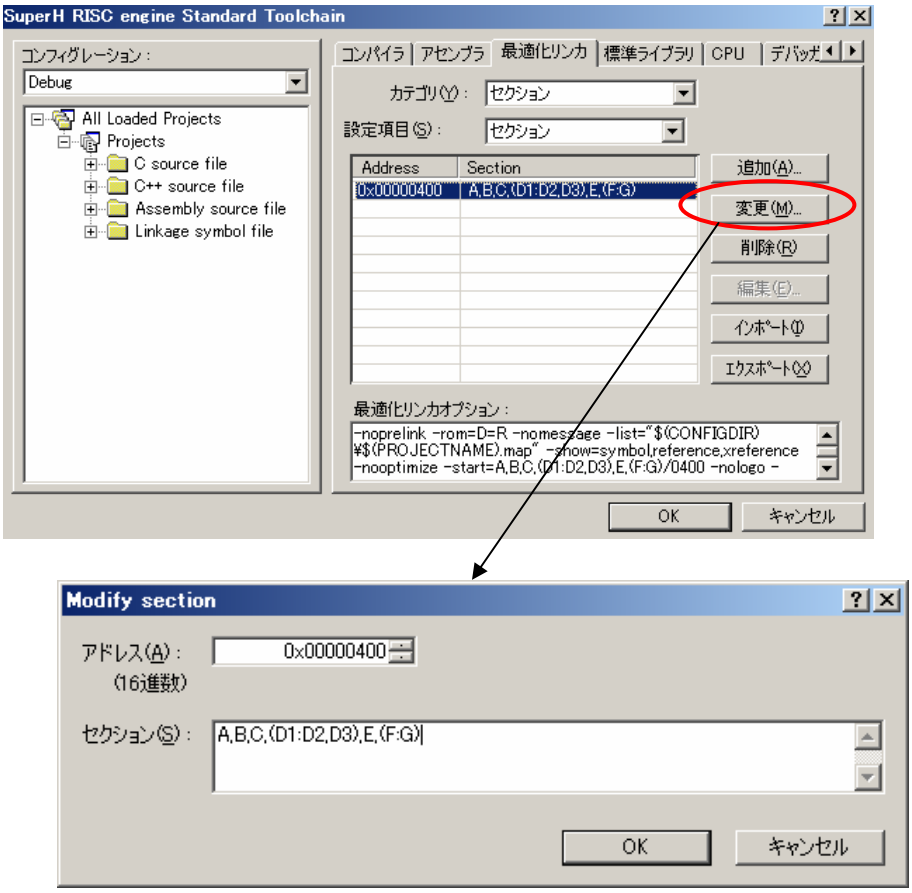
(2) -start=A,B,C,D1:D2,D3,E,F:G/400



- ":"で区切った直後のセクション(この場合 A,D2,G)を先頭として、それぞれ先頭が同一アドレスに割りつきます。

(3) -start=A,B,C,(D1:D2,D3),E,(F:G)/400

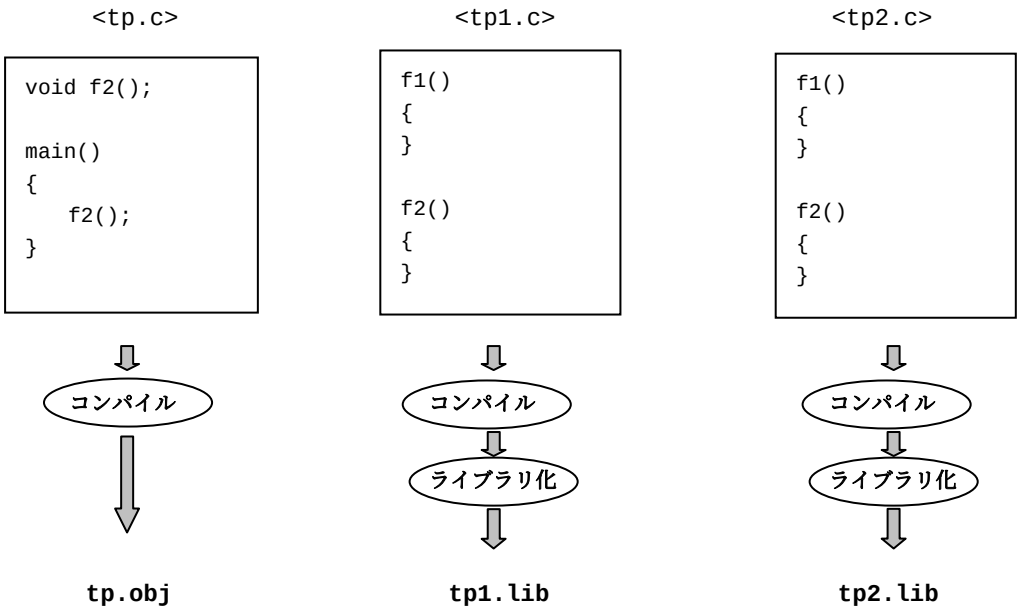
- 新しいオーバーレイの指定方法です。
- "()"で同一アドレス配置を括った場合、"()"の直前のセクション(この場合 C,E)の直後を先頭として "()"内の同一アドレス配置が行われます。
- "()"の直後のセクション(この場合 E)は、"()"内の最後尾のセクションの直後に続けて配置されます。
- ルネサス統合開発環境を使用している場合、新しいオーバーレイの指定方法は「編集」からでは行えません。「変更」より行ってください。



(c) ライブラリ内の同名シンボル通知 (最適化リンカ V.9.01 、パッケージ V.9.00R04(a) より対応)

リンク時に使用するライブラリファイル内に、複数個の同名シンボル(同じ名前の変数や関数)があることを ウォーニングメッセージ「L1320 (W) Duplicate Symbol "シンボル" in "ライブラリ(モジュール)"」で通知するようにしました。このことにより、今まで L1320 メッセージが出力されなかった環境で本メッセージが出力される可能性があります。ご注意下さい。 また、本メッセージは同名シンボルを含む全てのモジュールに対して出力される為、大量に通知されることがあります。このことにより他のメッセージの確認に支障が出る場合は、 nomessage=1320 オプションを指定して、L1320 メッセージを抑止してください。

【V.9.01 以降でのメッセージ出力例】



<リンクオプション>

```
optlnk tp.obj -lib=tp1.lib,tp2.lib -start=P/0
```

<上記リンクに対応するメッセージ出力>

```
** L1320 (W) Duplicate symbol "_f1" in "tp1.lib(tp1)"
** L1320 (W) Duplicate symbol "_f1" in "tp2.lib(tp2)"
** L1320 (W) Duplicate symbol "_f2" in "tp1.lib(tp1)"
** L1320 (W) Duplicate symbol "_f2" in "tp2.lib(tp2)"
```

(d) 物理空間配置時の重複検出機能 (最適化リンカ V.9.02 、パッケージ V.9.01R00 より対応)

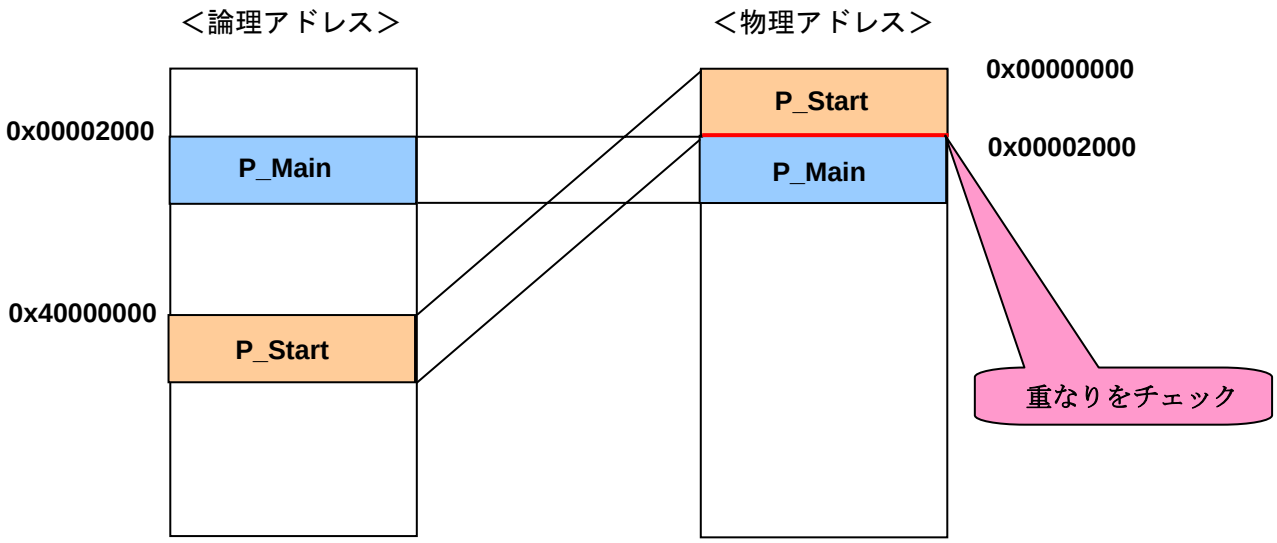
SH3 や SH4 などで、論理アドレス上では重ならないが実メモリ上に配置する際に重なってしまうオブジェクトを検出すためのオプション(ps_check) を追加しました。本オプションによって重複を検出した場合、エラーとしてリンク処理を終了します。

[例]

SH4 は、MMU が無効状態の場合、4G バイトのアドレス空間は、512M バイト(29bit)の外部メモリ空間へマッピングします(4G バイトアドレスの上位 3bit を無視してマッピングします)。たとえば、ユーザモードで使用可能な U0 領域(00000000～0x7fffffff)に対して、外部メモリ(512M)にマッピングする場合のオブジェクトの重なりは、下記記述で検出可能です。

-PS_check=00000000-1fffffff,20000000-3fffffff,40000000-5fffffff,60000000-7fffffff

本オプション記述により、00000000,20000000,40000000,60000000 番地はすべて、実メモリ上では同じ場所に配置されることを表します。



(e) データレコードのバイト数指定機能 (最適化リンカ V.9.02 、パッケージ V.9.01R00 より対応)

byte_count オプションによって、インテル HEX 形式ファイルのデータレコードの最大バイト数を変更できるようにしました。

(f) 空きエリアへの乱数充填機能 (最適化リンカ V.9.02 、パッケージ V.9.01R00 より対応)

空きエリア出力指定オプション(space) に乱数を埋め込む機能を追加しました。

(g) メモリ使用量削減オプション拡張 (最適化リンカ V.9.02 、パッケージ V.9.01R00 より対応)

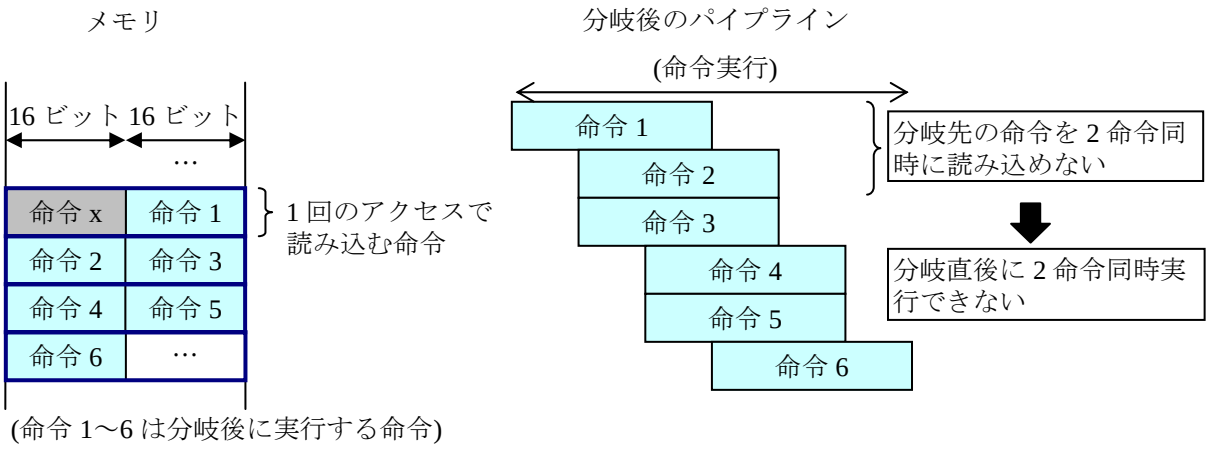
メモリ使用量削減機能オプション(memory=low)をライブラリファイル作成時にも使用できるようにしました。

A.12　Ver.9.1 から Ver.9.2 への追加機能

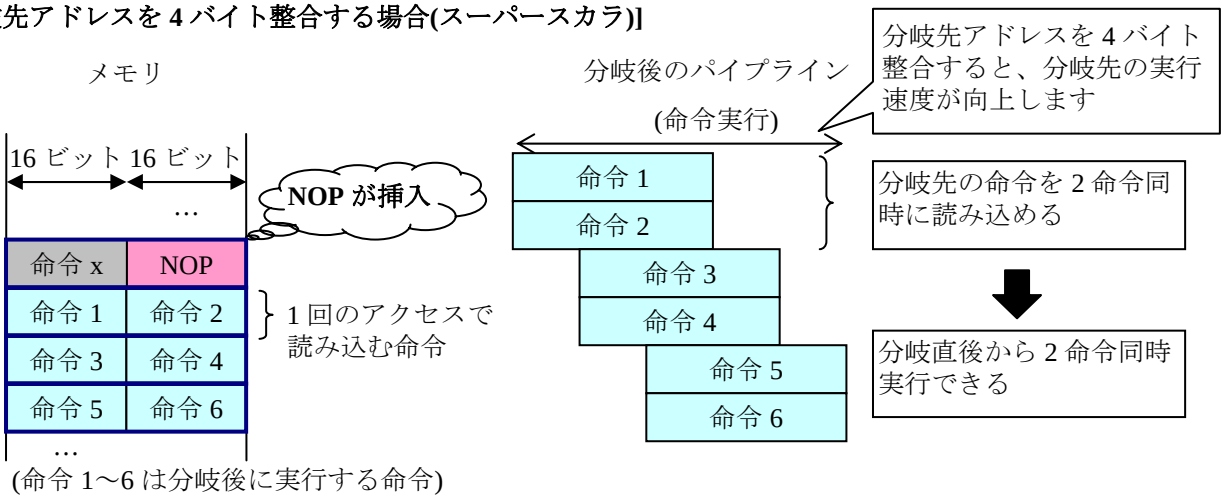
(1)　分岐先アドレスの 4 バイト整合 (#pragma align4 , -align4)

分岐先のアドレスを 4 バイト整合する機能を追加しました。SH マイコンは命令を読み込むときに 32 ビット幅で読み出します。SH マイコンの基本的な命令は 16 ビット長で構成されていますので、16 ビット長の命令が連続している場合は 2 命令を同時に読み込めます。しかし、分岐命令が存在する場合は、分岐先のアドレスが 4 バイトに整合されていないと、分岐先の命令を 2 命令同時に読み込むことが出来ません。このことにより、分岐先処理の効率が低下することがあります。本機能を使用すると、分岐先のアドレスが 4 バイトに整合され、分岐先の命令を 2 命令同時に取り込むことが可能となります。そのため、分岐後のメモリアクセス回数が削減され、実行速度の向上が期待できます。また、スーパースカラアーキテクチャを持つ CPU(SH-2A/SH2A-FPU/SH-4/SH-4A/SH4AL-DSP)であれば、分岐先の命令並列実行性の向上が期待できます。本機能は呼び出し回数の多い関数の条件文や繰り返し回数の多いループなど、実行回数が多い分岐に使用すると特に効果的です。

[分岐先アドレスを 4 バイト整合していない場合(スーパースカラ)]



[分岐先アドレスを 4 バイト整合する場合(スーパースカラ)]



本機能を使用すると、分岐先のアドレスを 4 バイト整合する為の NOP 命令が挿入されることがあります。このことにより、オブジェクトサイズが増加し処理効率が逆に低下する場合もあります。そのため、4 バイト整合の適用対象を 3 つの条件から選べるようになっています。どの適用対象のときが最も効率的であるかは、プログラムの内容に依存しますので、ユーザシステム上でそれぞれをお試しください。

align4 オプションおよび#pragma の書式は以下の通りです。

- オプション　　ALIGN4={ALL|LOOP|INMOSTLOOP}
- #pragma　　#pragma align4 [(]<関数名>=<指定種別>[,...][)]

指定種別		4バイト整合の対象	サイズ増加
オプション種別	**#pragma種別**		
ALL	all	**すべての分岐先アドレスを4バイト整合します。**	大
LOOP	loop	**すべてのループの先頭アドレスを4バイト整合します。**	
INMOSTLOOP	inmostloop	**最内側ループの先頭アドレスを4バイト整合します。**	小

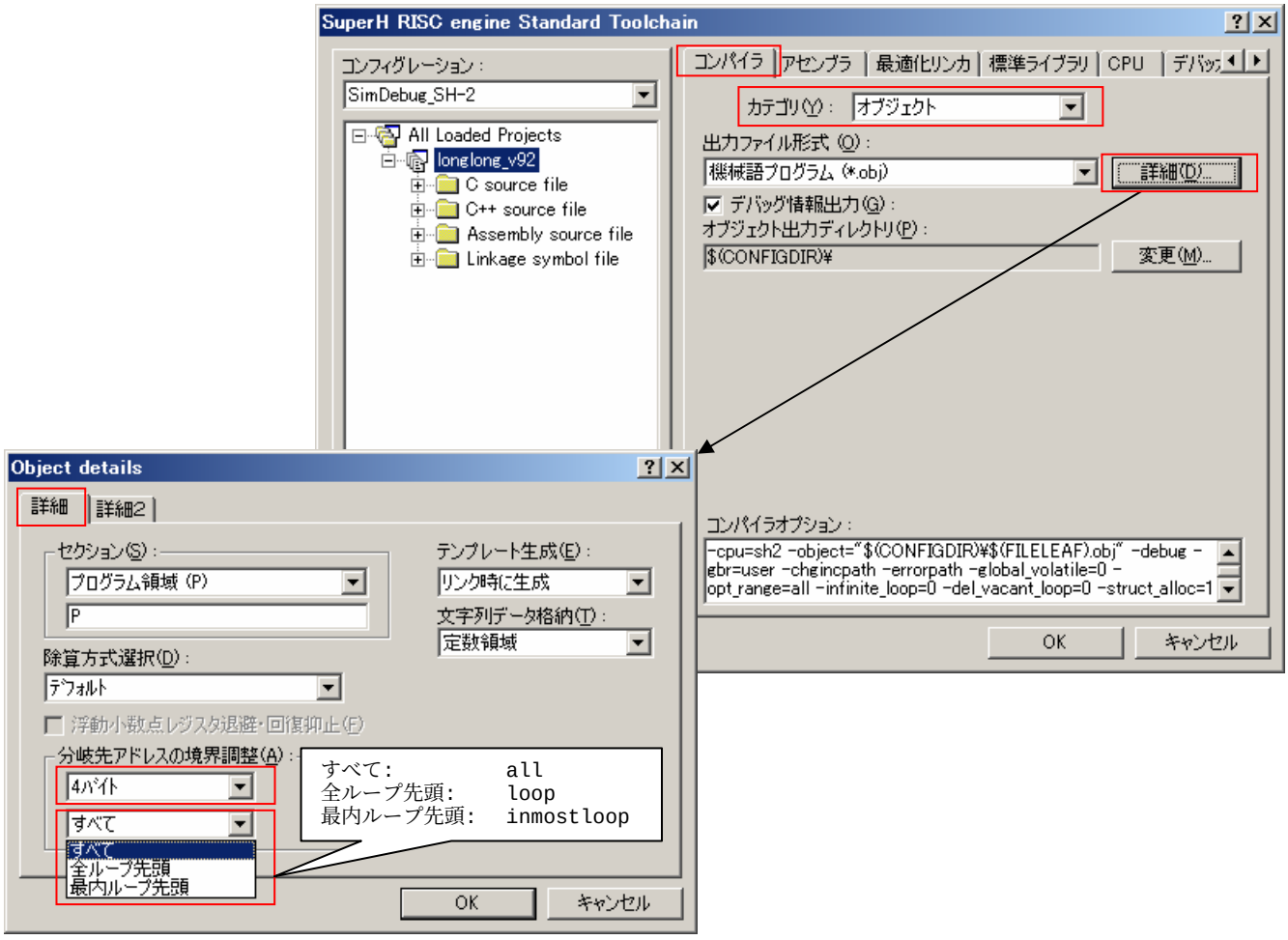
HEW で align4 オプションを指定するには、[SuperH RISC engine Standard Toolchain]ダイアログボックスの[コンパイラ]タブで以下の設定をしてください。

[カテゴリ]: “オブジェクト”

[詳細]:ボタンをクリック

[Object details]ダイアログボックスが表示されます。[Object details]ダイアログボックスの[詳細]タブで以下の設定をしてください。

[分岐先アドレスの境界調整]: ”4 バイト”にし、指定種別を選択



<例>

SH-2A で align4=all を指定した場合の例を以下に示します。

サンプルソース

```
int a[64],b[64];
int c;
void main() {
   int i;

   for(i=0;i<64;i++) {
      a[i] = b[i];
      c++;
   }
}
```

align4を指定していないときのアセンブリ展開コード

```
_main:
        MOV.L      L13,R7     ; _c
        MOV.L      @R7,R6     ; c
        MOV        #64,R1     ; H'00000040
        MOV.L      L13+4,R4   ; _a
        MOV.L      L13+8,R5   ; _b
L11:
        MOV.L      @R5+,R0    ; b[]
        DT         R1
        ADD        #1,R6
        BF/S       L11
        MOV.L      R0,@R4+    ; a[]
        RTS
        MOV.L      R6,@R7     ; c
L13:
        .DATA.L    _c
        .DATA.L    _a
        .DATA.L    _b
```

align4=allを指定したときのアセンブリ展開コード

```
_main:
        MOV.L      L13+2,R7   ; _c
        MOV.L      @R7,R6     ; c
        MOV        #64,R1     ; H'00000040
        MOV.L      L13+6,R4   ; _a
        MOV.L      L13+10,R5  ; _b
        NOP
L11:
        MOV.L      @R5+,R0    ; b[]
        DT         R1
        ADD        #1,R6
        BF/S       L11
        MOV.L      R0,@R4+    ; a[]
        RTS
        MOV.L      R6,@R7     ; c
L13:
        .RES.W     1
        .DATA.L    _c
        .DATA.L    _a
        .DATA.L    _b
```

サンプルソース実行時のコードサイズおよび実行速度は以下の通りです。

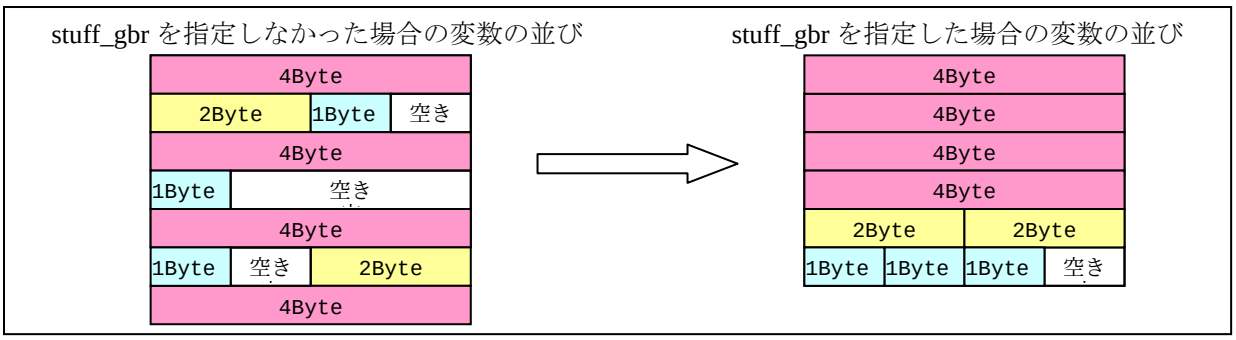
CPU **種別**	**コードサイズ** [Byte]		**実行速度**[Cycle]	
	align4 **指定なし**	align4=all **指定あり**	align4 **指定なし**	align4=all **指定あり**
SH-2A	36	38	393	330

（補足）

- 分岐先アドレスを 4 バイト整合した関数は、リンク時最適化の対象外となります。
- オプション align4/align16/align32/と#pragma align4 を同時に指定した場合、#pragma align4 が優先されます。
- オプション align4 とオプション align16/align32 を同時に指定した場合は次のエラーとなります。
  C3305 (F) Invalid command parameter "オプション名"
- #pragma align4 の記述に誤りがある場合、下記のエラーメッセージが表示されます。
  - 同一関数に対する#pragma align4の<指定種別>の異なる二重定義をしている場合
    C2806 (E) Multiple #pragma for one function
  - #pragma align4宣言より前に、その#pragma align4 で指定した関数を定義している場合
    C2857 (E) Function "関数名" in #pragma already declared
  - 関数以外のシンボルを #pragma align4 で指定した場合
    C2858 (E) Illegal #pragma <拡張子> function type
  - #pragma align4 に構文エラーがある場合
    C2859 (E) Illegal #pragma <拡張子> declaration

(2) GBR 領域($G0, $G1 セクション)のデータサイズ順変数配置指定 (stuff_gbr)

#pragma gbr_base/#pragma gbr_base1 を指定した変数をサイズ別に特定のセクションに配置させる、stuff_gbr オプションを追加しました。これにより境界調整用の空き領域が削減され、メモリを節約することができます。



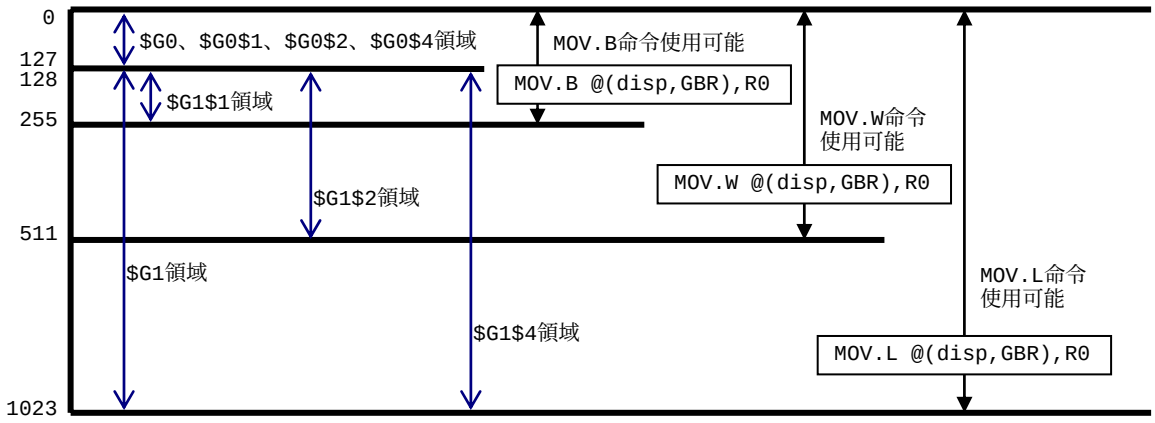
#pragma gbr_base/#pragma gbr_base1 を指定した変数は、変数のサイズにしたがって以下に示したセクションに配置されます。ただし、stuff_gbr オプションを指定していても、GBR 相対命令が出力される場合は、空の$G0 セクションが生成されます。

	stuff_gbr **指定あり**			stuff_gbr **指定なし**
	変数のサイズ(バイト)			
	4n	4n+2	2n+1	
#pragma gbr_base **指定変数**	$G0$4	$G0$2	$G0$1	$G0
#pragma gbr_base1 **指定変数**	$G1$4	$G1$2	$G1$1	$G1

stuff_gbr オプションは gbr=user オプションを指定した場合のみ有効になります。gbr=user オプションを指定せずに stuff_gbr オプションを指定した場合は、下記のウォーニングが表示され、stuff_gbr オプションを無視します。

C1301 (W) "stuff_gbr" option ignored

#pragma gbr_base / #pragma gbr_base1 を指定した変数は、MOV 命令を使用して GBR レジスタ＋相対値でアクセスされます。#pragma gbr_base を指定した変数は GBR レジスタ の指すアドレスからオフセット 0～127 バイトに収まるように配置する必要があります。#pragma gbr_base1 を指定した変数は型により配置範囲が変わります。MOV.B 命令を使用してアクセスする char/unsigned char 型の変数は、GBR レジスタ の指すアドレスからオフセット 128 ～255 バイトに収まるように配置する必要があります。同様に MOV.W 命令を使用してアクセスする short/unsigned short 型の変数は、GBR レジスタ の指すアドレスからオフセット 128～511 バイトに収まるように配置する必要があります。MOV.L 命令を使用してアクセスする int/unsigned int/long/unsigned long/float/double 型の変数は、GBR レジスタ の指すアドレスからオフセット 128～1023 バイトに収まるように配置する必要があります。



したがって、下記表に示す範囲に収めて、各セクションを割り付けてください。

セクション名	セクション割り付け範囲
$G0	GBR レジスタに設定するアドレスに割り付けます。・
$G0$1、$G0$2、$G0$4	($G0 の先頭アドレス+127)バイト以内に収まるように割り付けます。
$G1	($G0 の先頭アドレス+128)に割り付けます。
$G1$1	$G1 セクション以降に、($G0 の先頭アドレス+255)バイト以内に収まるように割り付けます。
$G1$2	$G1 セクション以降に、($G0 の先頭アドレス+511)バイト以内に収まるように割り付けます。
$G1$4	$G1 セクション以降に、($G0 の先頭アドレス+1023)バイト以内に収まるように割り付けます。

この範囲内に収めてセクションを割り付けない場合は、リンク時に下記のエラーになります。

L2330 (E)Relocation size overflow

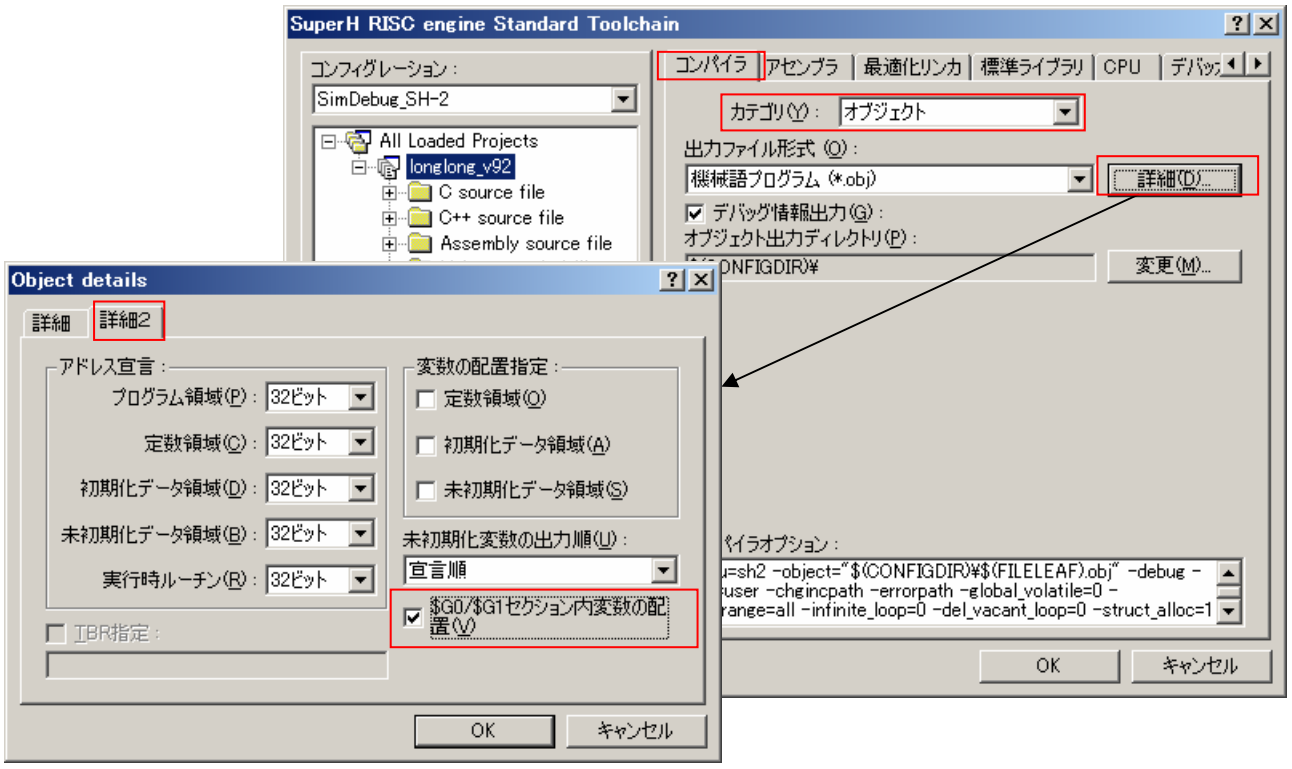
HEW で stuff_gbr オプションを指定するには、[SuperH RISC engine Standard Toolchain]ダイアログボックスの[コンパイラ]タブで以下の設定をしてください。

[カテゴリ]: “オブジェクト”

[詳細]:ボタンをクリック

[Object details]ダイアログボックスが表示されます。[Object details]ダイアログボックスの[詳細 2]タブで以下の設定をしてください。

[$G0/$G1 セクション内変数の配置]: チェックボックスをオン



<例>

サンプルソース

```
#pragma gbr_base(a,b,c,d,e,f,g,h,i)

long  a;
short b;
char  c;
long  d;
char  e;
long  f;
char  g;
short h;
long  i;
```

stuff_gbrを指定していないときのアセンブリ展開コード

```
          .SECTION    $G0,DATA,ALIGN=4
_a:                               ; static: a
          .DATAB.L    1,0
_b:                               ; static: b
          .DATAB.W    1,0
_c:                               ; static: c
          .DATAB.B    1,0
          .RES.B      1
_d:                               ; static: d
          .DATAB.L    1,0
_e:                               ; static: e
          .DATAB.B    1,0
          .RES.B      1
          .RES.W      1
_f:                               ; static: f
          .DATAB.L    1,0
_g:                               ; static: g
          .DATAB.B    1,0
          .RES.B      1
_h:                               ; static: h
          .DATAB.W    1,0
_i:                               ; static: i
          .DATAB.L    1,0
```

stuff_gbrを指定したときのアセンブリ展開コード

```
          .SECTION    $G0$4,DATA,ALIGN=4
_a:                               ; static: a
          .DATAB.L    1,0
_d:                               ; static: d
          .DATAB.L    1,0
_f:                               ; static: f
          .DATAB.L    1,0
_i:                               ; static: i
          .DATAB.L    1,0
          .SECTION    $G0$2,DATA,ALIGN=2
_b:                               ; static: b
          .DATAB.W    1,0
_h:                               ; static: h
          .DATAB.W    1,0
          .SECTION    $G0$1,DATA,ALIGN=1
_c:                               ; static: c
          .DATAB.B    1,0
_e:                               ; static: e
          .DATAB.B    1,0
_g:                               ; static: g
          .DATAB.B    1,0
```

境界調整用領域：5 バイト

<注意>

共用体、構造体もしくは配列を使用する場合、セクションを割り付けるアドレスに注意してください。共用体、構造体もしくは配列を #pragma gbr_base1 に指定している場合は、表で示した範囲内に収めてセクションを割り付けても、リンクエラーが発生する場合があります。例えば、下記サンプルソースのように char x[4]に#pragma gbr_base1 を指定した場合、サイズが 4 バイトの配列 x は $G1$4 セクションに配置されますが、配列 x を参照する処理では MOV.B 命令が使用されます。したがって、配列 x が配置される$G1$4 セクションを、オフセット 128～1023 バイト以内ではなく、オフセット 128～255 バイト以内に収まるように割り付けなければリンクエラーが発生します。

サンプルソース

```
#pragma gbr_base1(x)

char x[4];

void func(void)
{
   x[0] = 1;
}
```

アセンブリ展開コード

```
          .SECTION    P,CODE,ALIGN=4
_func:                            ; function: func
          MOV         #1,R0       ; H'00000001
          RTS
          MOV.B       R0,@(_x-(STARTOF $G0),GBR); x[]
          .SECTION    $G1$4,DATA,ALIGN=4
_x:                               ; static: x
          .DATAB.B    4,0
          .SECTION    $G0,DATA,ALIGN=4
```

(3) C++言語のインライン展開の抑止指定 (cpp_noinline)

cpp_noinline オプションを指定することで、C++言語の固有のインライン展開を抑止できるようになりました。C++言語の固有のインライン展開には、次の 2 通りがあります。

- 関数指定子inlineを指定した関数のインライン展開

サンプルソース

```
int x;
inline int func(int a){
    return a * 3;
}
void g(void){
     x = func(x);
}
```

インライン展開イメージ

```
void g(void) {
     x = x * 3;
}
```

- クラス内で定義したメンバ関数のインライン展開

サンプルソース

```
class A {
   int cx, cy;
public:
   int func() {
       return cx + cy;
   }
};
int g(class A a) {
    return = a.func();
}
```

インライン展開イメージ

```
int g(class A a) {
     return (a.cx + a.cy);
}
```

cpp_noinline オプション を指定しても C 言語でも有効な次のインライン展開は抑止されません。

- inlineオプションの指定により行われる関数の自動インライン展開
- #pragma inlineの指定により行われる関数のインライン展開

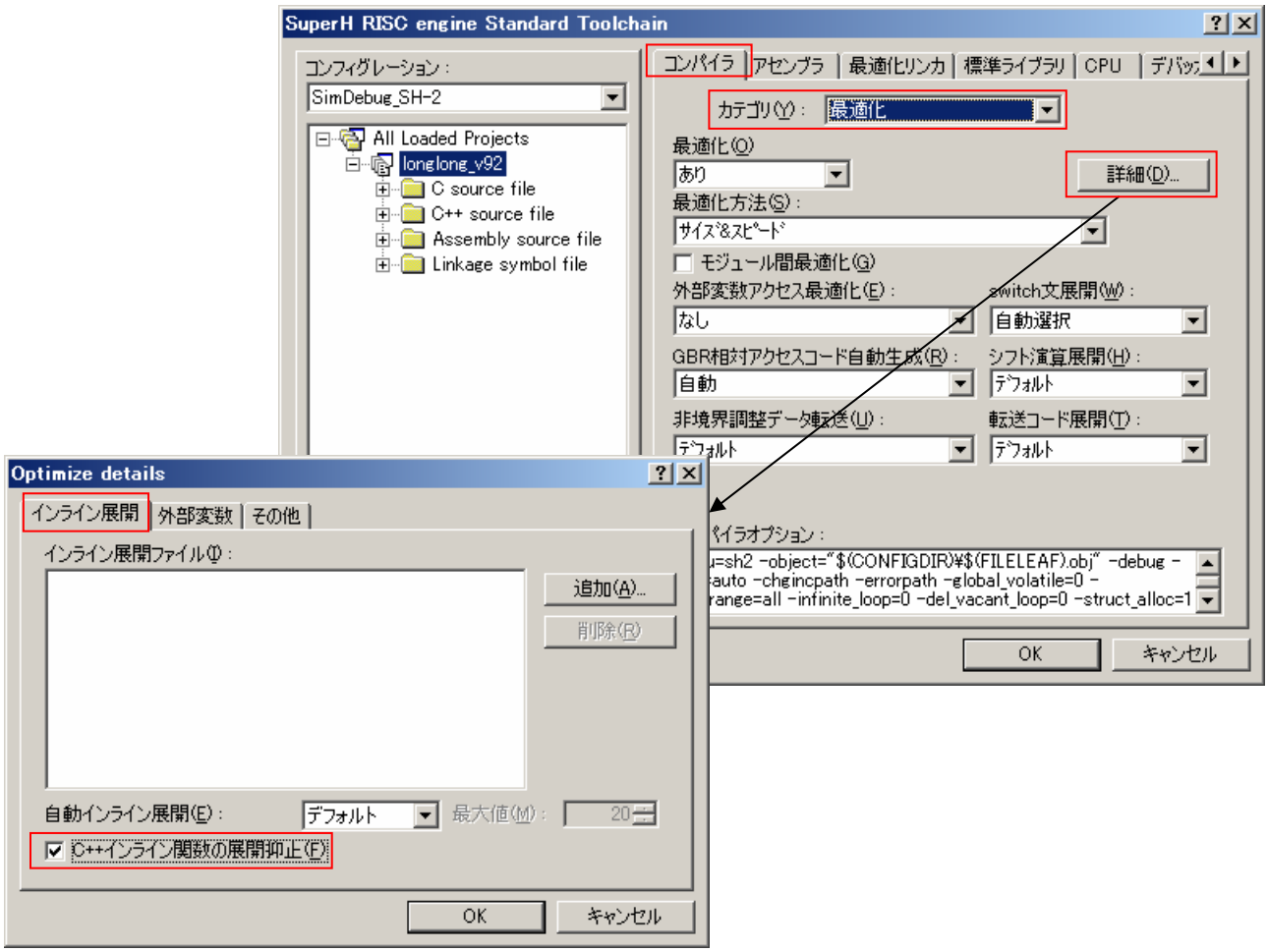
HEW で cpp_noinline オプションを指定するには、[SuperH RISC engine Standard Toolchain]ダイアログボックスの[コンパイラ]タブで以下の設定をしてください。

[カテゴリ]: “最適化”

[詳細]:ボタンをクリック

[Optimize details]ダイアログボックスが表示されます。[Optimize details]ダイアログボックスの[インライン展開]タブで以下の設定をしてください。

[C++インライン展開の展開抑止]: チェックボックスをオン



(4) 既存機能の仕様追加・変更

(a) #pragma inline 指定でのインライン展開条件の変更

#pragma inline を指定した場合のインライン展開条件を変更しました。インライン展開先の関数サイズの増加率が一定値を超える場合は、インライン展開を抑止していましたが、関数サイズの増加率によるインライン展開の抑止をしないように変更しました。なお、関数サイズの増加率とは別に、インライン展開処理によるコンパイル時間の増大やメモリ使用量の増大を考慮した制限は別途存在します。そのため、#pragma inline 指定のある関数であっても、必ずインライン展開されることが保証されているわけではありません。

なお、scope オプション(デフォルトで有効)を用いて最適化範囲が分割される場合は、インライン展開を抑止することがあります。その場合は、noscope オプションを指定することで、インライン展開されるようになります。

(b) 組み込み関数の原型宣言

組み込み関数の原型宣言を C 言語でも行うようにしました。

(5) メッセージの改善

(a) 除数をゼロとする除算に対するメッセージ出力変更

関数内の式で除数をゼロとする除算がある場合に、以下のウォーニングを出力するようになりました。

C1501 (W) Division by zero

サンプルソース

```
int a,b;

void main(){
    a = b/0;  /* C1501 (W) Division by zero */
}
```

エラーメッセージ

```
test.c(4) : C1501 (W) Division by zero
```

(補足)
C++言語を使用している場合は、以下のウォーニングが出力されます。

C5039 (W) Division by zero

(6) 最適化の強化

(a) long long 型演算の命令展開

実行時ルーチンの呼び出しによって処理されていた(unsigned) long long の演算を命令展開で処理するように最適化を強化しました。ただし、次の演算は条件を満たす場合のみ命令展開になります。条件を満たさない場合は、実行時ルーチンの呼び出しになります。

- 除算/剰余算演算
  次の全ての条件を満たす場合
  - 除数の下位 32bit が"0"の定数値
  - CPU が SH-2A/SH2A-FPU の場合もしくは除数が 0 以外の場合
  - 除数の上位 32bit が”-1”以外の場合(signed long long で 0xFFFFFFFF00000000 以外の場合)
  - 下記表で命令展開と記載されているオプションを指定した場合

CPU	除算方式選択	実行速度/サイズ最適化	命令展開/実行時ルーチン
SH-1	-	-	**実行時ルーチン**
SH-2A/SH2A-FPU	-	-	**命令展開**
その他	division=cpu=inline	-	**命令展開**
	division=cpu	speed	**命令展開**
		nospeed	**命令展開**
		size	**実行時ルーチン**
	division=cpu=runtime	-	**実行時ルーチン**

・ 比較演算
等価式(==、!=) もしくは 0 との不等式 (<、>、<=、>=)の場合

・ シフト演算
シフト数が 1、8、16、32 の定数値の場合

・ キャスト演算
float/double 型以外からのキャストもしくは float/double 型以外へのキャストの場合

<例>

SH-2 では long long 型の加算演算は次のように命令展開されます。

サンプルソース

```
long long x, y;
long long  func(void) {
   return x + y;
}
```

V.9.1で生成したアセンブリ展開コード

```
_func:
        STS.L      PR,@-R15
        MOV.L      L11+2,R4   ; _y
        MOV.L      @(4,R4),R1 ; (part of)y
        MOV.L      @R4,R2     ; (part of)y
        MOV.L      R1,@-R15
        MOV.L      R2,@-R15
        MOV.L      L11+6,R6   ; _x
        MOV.L      @(4,R6),R4 ; (part of)x
        MOV.L      @R6,R5     ; (part of)x
        MOV.L      R4,@-R15
        MOV.L      R5,@-R15
        MOV.L      @(20,R15),R7
        MOV.L      L11+10,R2  ; __add64
        JSR        @R2
        MOV.L      R7,@-R15
        ADD        #20,R15
        LDS.L      @R15+,PR
        RTS
        NOP
L11:
        .RES.W     1
        .DATA.L    _y
        .DATA.L    _x
        .DATA.L    __add64
```

V.9.2で生成したアセンブリ展開コード

```
_func:
        MOV.L      L11+2,R1   ; _x
        MOV.L      L11+6,R5   ; _y
        MOV.L      @(4,R1),R4 ; (part of)x
        MOV.L      @(4,R5),R7 ; (part of)y
        MOV.L      @R1,R6     ; (part of)x
        MOV.L      @R5,R1     ; (part of)y
        CLRT
        ADDC       R7,R4
        MOV.L      @R15,R2
        ADDC       R1,R6
        MOV.L      R6,@R2
        RTS
        MOV.L      R4,@(4,R2)
L11:
        .RES.W     1
        .DATA.L    _x
        .DATA.L    _y
```

サンプルソースのコードサイズおよび x=1 、y=1 指定時の実行速度は以下の通りです。

CPU **種別**	**コードサイズ** [Byte]		**実行速度**[Cycle]	
	V9.1	V9.2	V9.1	V9.2
SH-2	50	34	77	27

(b) CLIP 命令活用(SH-2A/SH2A-FPU)

SH-2A/SH2A-FPU には、飽和値比較命令(CLIPS.B、CLIPS.W、CLIPU.B、CLIPU.W)が用意されていますが、これらの命令を生成するには組み込み関数(clipsb()、clipsw()、clipub()、clipuw())を使用する必要がありました。組み込み関数を使用しなくても飽和判定処理で飽和値比較命令を生成するように最適化を強化しました。
<例>

・ 次の場合にCLIPS.B 命令を生成します。

サンプルソース(1)

```
void func(long a) {
      …
   if (a > 127) {
      a = 127;
```

アセンブリ展開コード

```
_func:
              …
        CLIPS.B    R4    ; a
              …
```

```
    }
    else if (a <  -128) {
        a = -128;
    }
        …
}
```

サンプルソース(2)

```
void func(long a) {
        …
    if (a > 127) {
        a = 127;
    }
    if (a < -128) {
        a = -128;
    }
        …
}
```

サンプルソース(3)

```
void func(long a) {
        …
    a = (a < 127) ? a : 127;
    a = (a > -128) ? a : -128;
        …
}
```

サンプルソース(4)

```
void func(long a) {
        …
    a = (a < 127) ? ((a > -128) ? a : -128) : 127;
        …
}
```

・ 次の場合にCLIPU.B 命令を生成します。

サンプルソース(1)

```
void func(unsigned long a) {
        …
    if (a > 255) {
        a = 255;
    }
        …
}
```

サンプルソース(2)

```
void func(unsigned long a) {
        …
    a = (a < 255) ? a : 255;
        …
}
```

アセンブリ展開コード

```
_func:
                …
        CLIPU.B     R4      ; a
                …
```

・ 次の場合にCLIPS.W 命令を生成します。

サンプルソース(1)

```
void func(long a) {
       …
    if (a > 32767) {
        a = 32767;
    }
    else if (a <  -32768) {
        a = -32768;
    }
       …
}
```

サンプルソース(2)

```
void func(long a) {
       …
    if (a > 32767) {
        a = 32767;
    }
    if (a < -32768) {
        a = -32768;
    }
       …
}
```

サンプルソース(3)

```
void func(long a) {
       …
    a = (a < 32767) ? a : 32767;
    a = (a > -32768) ? a : -32768;
       …
}
```

サンプルソース(4)

```
void func(long a) {
       …
    a = (a < 32767) ? ((a > -32768) ? a : -32768) : 32767;
       …
}
```

アセンブリ展開コード

```
_func:
                …
        CLIPS.W     R4      ; a
                …
```

・ 次の場合にCLIPU.W 命令を生成します。

サンプルソース(1)

```
void func(unsigned long a) {
       …
    if (a > 65535) {
        a = 65535;
    }
       …
}
```

サンプルソース(2)

```
void func(unsigned long a) {
       …
    a = (a < 65535) ? a : 65535;
       …
```

アセンブリ展開コード

```
_func:
                …
        CLIPU.W     R4      ; a
                …
```

それぞれの飽和比較命令を生成する、飽和判定対象の変数の型を以下に示します。

飽和比較命令	対象変数の型の条件
CLIPS.B	long、int、short
CLIPU.B	unsigned long、unsigned int、unsigned short
CLIPS.W	long、int
CLIPU.W	unsigned long、unsigned int

(補足)
- 上限値判定と下限値判定の順序を入れ替えても CLIPS.B、CLIPS.W 命令を生成します。
- 上限判定や下限判定にイコール付きの比較演算子(">="など)を使用しても飽和比較命令を生成します。
- 上限値の代入処理および下限値の代入処理がある場合には飽和比較命令を生成しますが、上限値や下限値を直接 return 文で返す場合などは、飽和比較命令を生成しません。
- 飽和判定とは関係のない変数への代入など、飽和判定対象の変数への上限値および下限値の代入以外の処理がある場合は、飽和比較命令を生成しません。

(7) 最適化リンケージエディタの主な追加・改善内容

(a) ROM/RAM 領域の合計セクションサイズの表示 (total_size, show=total_size) (最適化リンカ V.9.03 、パッケージ V.9.01 Release 01 より対応)

total_size オプションを指定することで、セクションの合計サイズ情報を 3 種類に分類して標準出力に出力します。show=total_size オプションを使用することで、リンケージリストファイルにも合計サイズの情報を出力できます。

RAMDATA SECTION: RAM 配置セクションの合計サイズ(バイト)
ROMDATA SECTION: プログラムセクション以外の ROM 配置セクションの合計サイズ(バイト)
PROGRAM SECTION: プログラムセクションの合計サイズ(バイト)

ROM 化支援機能(rom オプション)対象に指定されたセクションは、転送元(ROM)と転送先(RAM)の両方で領域を使用するため、RAMDATA SECTION と ROMDATA SECTION の両方に加算されることになります。
コンパイラの各デフォルトセクションに対する分類は以下の通りです。

デフォルトセクション名	種別	合計されるセクション種別
P	**プログラム領域**	PROGRAM SECTION
C	**定数領域**	ROMDATA SECTION
D	**初期化データ領域**	ROMDATA SECTION
R (HEW **使用時**)	**初期化データ領域(転送先)** (ROM **化支援機能使用時**)	RAMDATA SECTION
B	**未初期化データ領域**	RAMDATA SECTION
S	**スタック領域**	RAMDATA SECTION

出力例は以下の通りです。

- total_sizeオプション(標準出力に出力)

```
RAMDATA SECTION:  00000808 Byte(s)
ROMDATA SECTION:  0000041c Byte(s)
PROGRAM SECTION:  00000358 Byte(s)
```

- show=total_sizeオプション(リンケージリストファイルに出力)

```
*** Total Section Size ***

RAMDATA SECTION:  00000808 Byte(s)
ROMDATA SECTION:  0000041c Byte(s)
PROGRAM SECTION:  00000358 Byte(s)
```

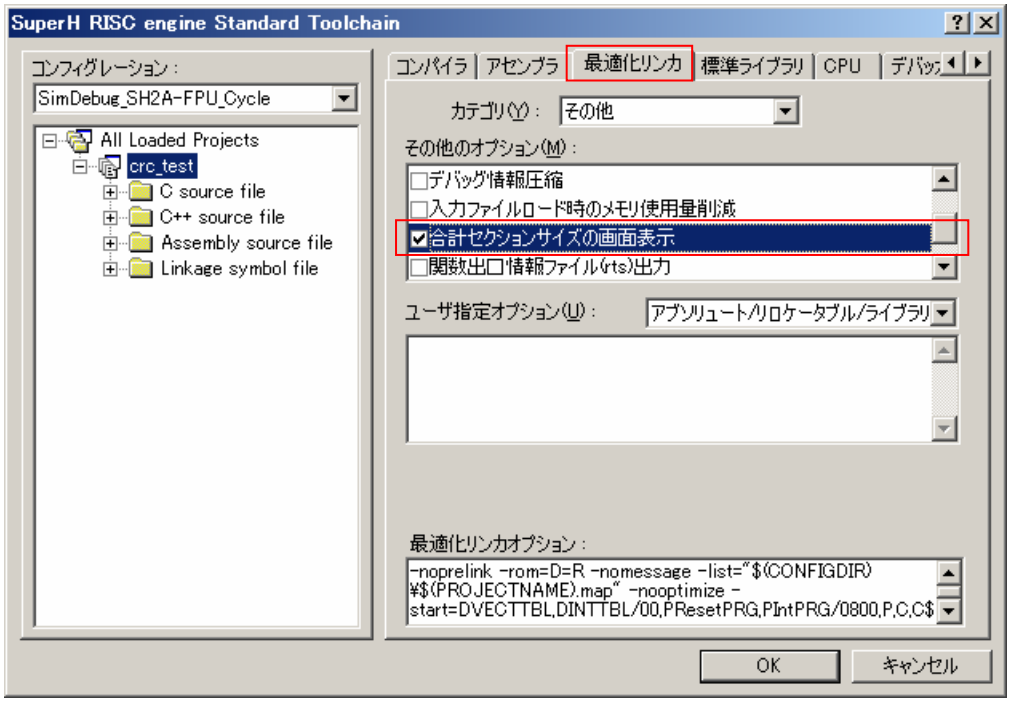
HEW で使用する場合は次のように設定してください。

- total_sizeオプション(標準出力)

  [SuperH RISC engine Standard Toolchain]ダイアログボックスの[最適化リンカ]タブで以下の項目を設定してください。

  [カテゴリ]: “その他”

  [その他のオプション]: “合計セクションサイズの画面表示”チェックボックスをオン

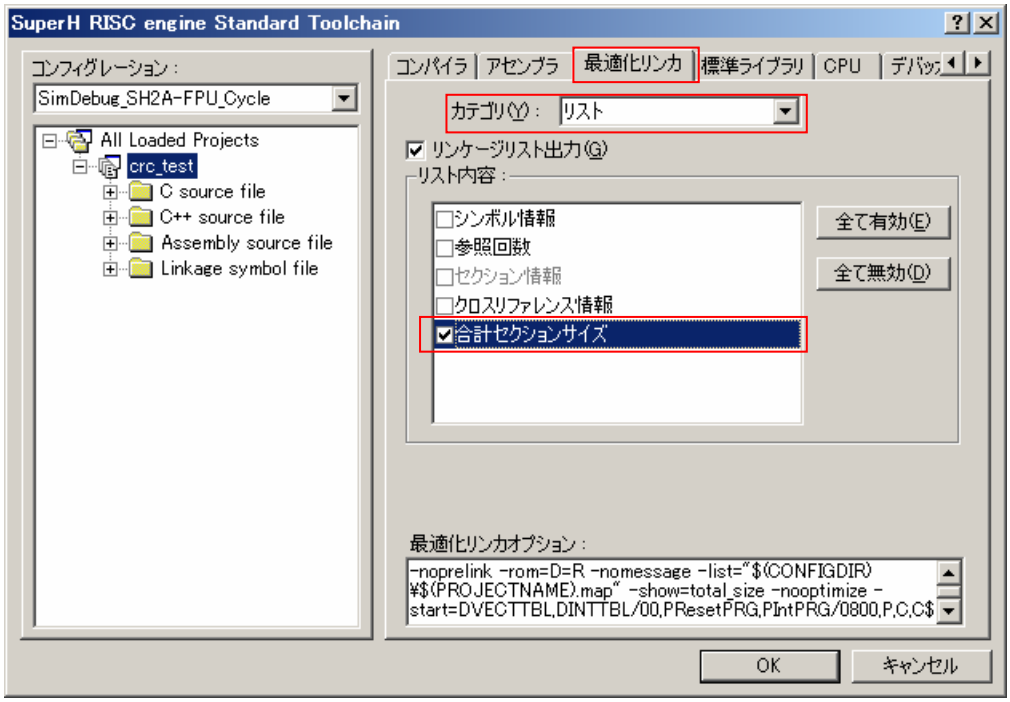


- show=total_sizeオプション(リストファイルに出力)

  [SuperH RISC engine Standard Toolchain]ダイアログボックスの[最適化リンカ]タブで以下の設定をしてください。

  [カテゴリ]: “リスト”

  [リスト内容]: “合計セクションサイズ”チェックボックスをオン



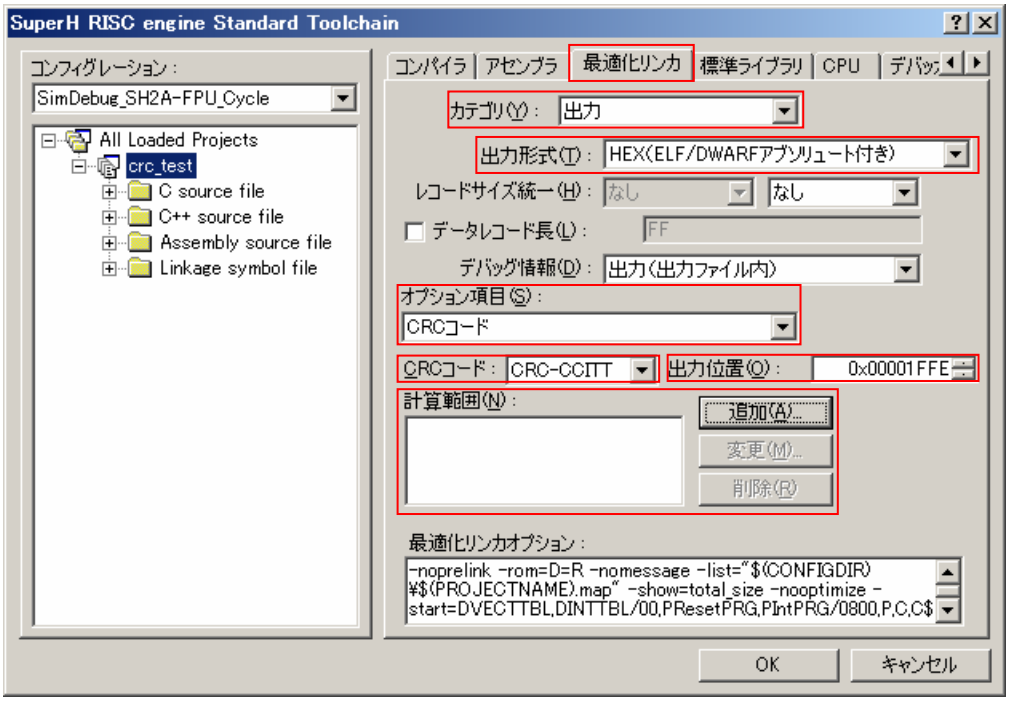
(b) CRC 演算結果の埋め込み (crc)

オプションで指定された領域に対する CRC (Cyclic Redundancy Check)を求め、その結果を特定のアドレスへ埋め込むことが可能となりました。CRC 演算結果を埋め込むことにより、組み込みシステム上のデータが生成時のデータと一致するかを確認することが出来ます。crc オプションの書式は以下の通りです。

-CRc = <サブオプション>
<サブオプション>: <出力位置>=<計算範囲>[/<多項式>]
<出力位置>： <アドレス>
<計算範囲>： <先頭アドレス>-<終了アドレス>[,...]
<多項式> ： { CCITT | 16 }

計算範囲で指定された内容を下位アドレスから上位アドレスの順で CRC 演算を行い、計算結果を出力位置のアドレスに出力します。多項式は CRC-CCITT または CRC-16 を選択できます(デフォルトは CRC-CCITT)。
多項式
CRC-CCITT
X^16+X^12+X^5+1
ビット表現(10001000000100001)
CRC-16
X^16+X^15+X^2+1
ビット表現(11000000000000101)

HEW で使用する場合は次のように設定してください。
[SuperH RISC engine Standard Toolchain]ダイアログボックスの[最適化リンカ]タブで、以下の設定をしてください。
[カテゴリ]: “出力”
[出力形式]: “HEX(ELF/DWARF アブソリュート付き)” もしくは “S タイプ(ELF/DWARF アブソリュート付き)”
[オプション項目]: “CRC コード”
[CRC コード]: “CRC-CCITT”もしくは “CRC-16”
[出力位置]: CRC 演算結果を出力するアドレスを指定
[計算範囲]: [追加]、[変更]、[削除]ボタンより CRC 演算範囲を指定



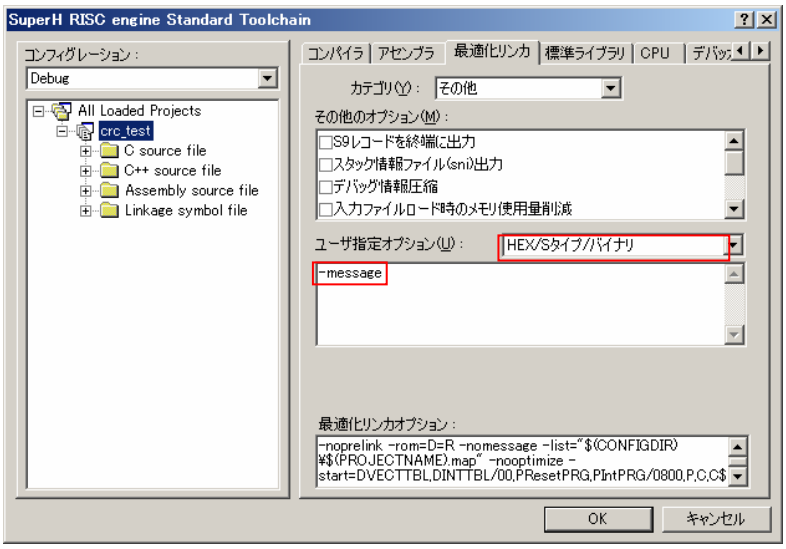
CRC 演算結果埋め込みに関する情報を出力するには、[SuperH RISC engine Standard Toolchain]ダイアログボックスの[最適化リンカ]タブで、以下の設定をしてください。

＜標準出力にメッセージ出力する場合＞

[カテゴリ]: “その他”

[ユーザー指定オプション]: “HEX/S タイプ/バイナリ”

-message　を追記



crc オプション指定時、標準出力に以下のメッセージが出力されます。

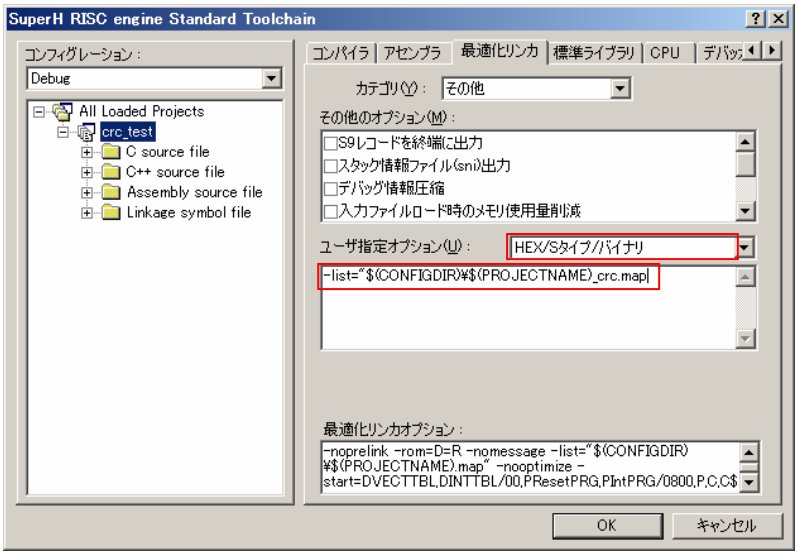
L0500 (I) Generated CRC code at "アドレス"

＜リンケージリストに出力する場合＞

[カテゴリ]: “その他”

[ユーザー指定オプション]: “HEX/S タイプ/バイナリ”

-list=″$(CONFIGDIR)¥$(PROJECTNAME)_crc.map″　を追記



crc オプション指定時、リンケージリスト(プロジェクト名_crc.map)に CRC 演算結果と出力位置アドレスが以下のように出力されます。

CODE：CRC 演算結果

ADDRESS:出力位置アドレス

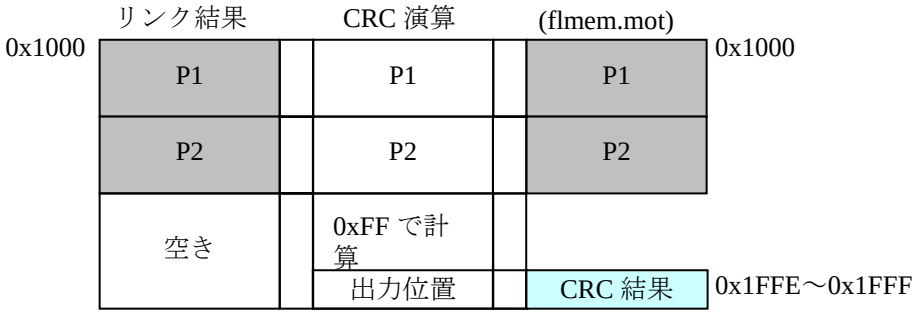
```
*** CRC code ***

CODE       : 5db6
ADDRESS : 00002ffe
```

<例>

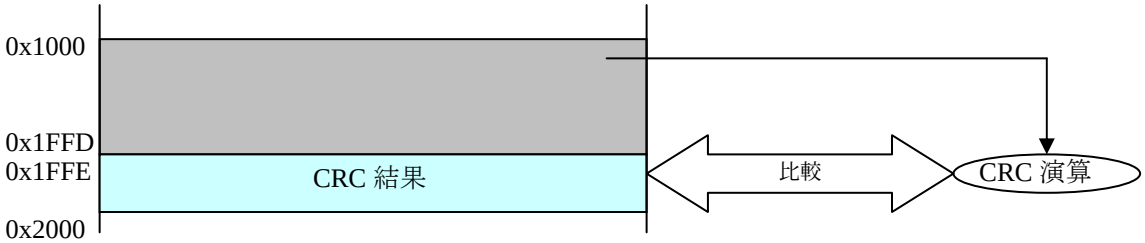
オプションを以下のように指定すると、

```
optlnk *.obj -form=stype -start=P1,P2/1000
             -crc=1FFE=1000-1FFD
             -output=flmem.mot
```

次の図のように、作成される flmem.mot に CRC 結果を埋め込みます。



上記のオプションで生成されたイメージを実機の ROM に書き込んだ場合、実機の ROM イメージから 0x1000～0x1FFD の CRC 演算結果を求めて、その演算結果と ROM イメージに埋め込まれた CRC 演算結果が同じ値であれば、0x1000～0x1FFD の範囲が生成時の ROM イメージファイルと実機の ROM イメージで差分がないことを確認できます。



CRC 演算を行うサンプルプログラムを以下に示します。このサンプルプログラムより求めた ROM イメージの CRC 演算値と ROM に書き込まれた CRC 値を比較することで、ROM に書き込まれたデータの整合性を確認できます。

・ 多項式CRC-CCITTの場合

サンプルソース

```
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
uint16_t CRC_CCITT(uint8_t *pData, uint32_t iSize)
{
    uint32_t ui32_i;
    uint8_t *pui8_Data;
    uint16_t ui16_CRC = 0xFFFFu;
    pui8_Data = (uint8_t *)pData;
    for(ui32_i = 0; ui32_i < iSize; ui32_i++)
    {
        ui16_CRC = (uint16_t)((ui16_CRC >> 8u) |
            ((uint16_t)((uint32_t)ui16_CRC << 8u)));
        ui16_CRC ^= pui8_Data[ui32_i];
        ui16_CRC ^= (uint16_t)((ui16_CRC & 0xFFu) >> 4u);
        ui16_CRC ^= (uint16_t) ((ui16_CRC << 8u) << 4u);
        ui16_CRC ^= (uint16_t)(((ui16_CRC & 0xFFu) << 4u) << 1u);
    }
    ui16_CRC = (uint16_t)( 0x0000FFFFul &
        ((uint32_t)~(uint32_t)ui16_CRC) );
    return ui16_CRC;
}
```

・ 多項式CRC-16の場合

サンプルソース

```
#define POLYNOMIAL 0xa001
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;

uint16_t CRC16(uint8_t *pData, uint32_t iSize)
{
    uint16_t crcdData = (uint16_t)0;
    uint32_t data = 0;
    uint32_t i,cycLoop;
    for(i=0;i<iSize;i++){
        data = (uint32_t)pData[i];
        crcdData = crcdData ^ data;
        for (cycLoop = 0; cycLoop < 8; cycLoop++) {
            if (crcdData & 1) {
                crcdData = (crcdData >> 1) ^ POLYNOMIAL;
            } else {
                crcdData = crcdData >> 1;
            }
        }
    }
    return crcdData;
}
```

(補足)

crc オプションを使用する場合以下のことに注意してください。

- CRC演算の計算順は計算範囲の指定順ではありません。下位アドレスから上位アドレスの順に計算されます。
- 複数のアブソリュートファイル入力時は、下記ウォーニングを出力しcrcオプションを無効にします。

  L1000 (W) Option "crc" ignored
- CRC出力位置は、計算範囲に含むことは出来ません、計算範囲に含めると以下のエラーになります。

  L2022 (E) Address ranges overlap in option "crc" : "<先頭アドレス>–<終了アドレス>"
- 出力形式がform={hexadecimal | stype}の場合に有効です。それ以外の出力形式を指定すると以下のエラーになります。

  L2004 (E) Option "crc" cannot be combined with option "form=<出力形式>”
- outputオプションで出力範囲が指定されている場合は、CRC出力位置がその出力範囲に含まれている必要があります。出力範囲に含まれていなければ以下のエラーとなります。

  L1181 (W) Fail to write "CRC Code"
- 指定された計算範囲の空き領域は0xFFと仮定してCRC演算を行います。spaceオプションで指定されている場合は、spaceオプションの指定値で演算を行います。なお、spaceオプションを指定する場合は、1バイトのデータを指定してください。randomもしくは2バイト以上の値を指定すると、以下のエラーになります。

  L2004 (E) Option "crc" cannot be combined with option "space=<指定値> "

(制限事項)

オーバーレイ使用時は、crc オプションを使用しないでください。

B. バージョンアップにおける注意事項

旧バージョン(「SuperH RISC engine C/C++コンパイラパッケージ」Ver.6.x 以前)からバージョンアップして使用する場合の注意事項を説明します。

B.1 プログラムの動作保証

コンパイラをバージョンアップしてプログラム開発する場合、プログラムの動作が変わることがあります。プログラムを作成する際は、以下の点に注意して、お客様のプログラムを十分にテストしてください。

(1) プログラムの実行時間やタイミングに依存するプログラム

言語仕様は、プログラムの実行時間については何も規定していません。したがってコンパイラのバージョンの違いによりプログラムの実行時間と I/O など周辺機器のタイミングのずれ、あるいは割り込み処理のような非同期処理の時間の差などにより、プログラムの動作が変わる場合があります。

(2) 1 つの式に 2 個以上の副作用が含まれているプログラム

1 つの式に 2 個以上の副作用が含まれている場合、コンパイラのバージョンにより、動作が変わる可能性があります。

例：

```
a[i++]=b[i++];              /* i のインクリメント順序は不定です。*/
f(i++,i++) ;                /* インクリメントの順序でパラメータの値が変わります。*/
                            /* i の値が 3 の時 f(3,4) または f(4,3) になります。*/
```

(3) 結果がオーバフローや不当演算に依存するプログラム

オーバフローが生じた場合や、不当演算を実施した場合、結果の値は保証しません。したがって、バージョンが変わると動作が変わる可能性があります。

例：

```
int a, b;
x=(a*b)/10;   /* a と b の値の範囲によってはオーバフローする可能性があります*/
```

(4) 変数の初期化抜け、型の不一致

変数が初期化されていない場合や、パラメータやリターン値の型が呼び出し側と呼び出される側で対応していない場合、不正な値をアクセスすることになります。したがって、コンパイラのバージョンによって動作が変わる場合があります。

file1:

```
int f(double d)
{
        :
}
```

file2:

```
int g(void)
{
        f(1);
}
```

関数呼び出し側のパラメータは int 型ですが、関数定義側のパラメータは、double 型のため、

上記に記載された情報がすべての起こりうる状況を示したわけではありません。したがって、お客様の責任で本コンパイラを正しくご使用の上、お客様のプログラムを十分にテストしてください。

B.2 旧バージョンとの互換性

旧バージョンのコンパイラ(Ver.5.x 以前)、アセンブラ(Ver.4.x 以前)およびリンケージエディタ(Ver.6.x 以前)出力のオブジェクトファイル、ライブラリファイルとリンクする場合、または旧バージョンで使用していたデバッガをそのまま使用する場合に注意すべき点を説明します。

(1) オブジェクト形式

オブジェクトファイル形式は、従来の SYSROF から標準フォーマットの ELF 形式に変更しました。
また、デバッグ情報形式も、標準フォーマットの DWARF2 形式に変更しました。
旧バージョンのコンパイラ(Ver.5.x 以前)、アセンブラ(Ver.4.x 以前)出力オブジェクトファイル(SYSROF)を最適化リンケージエディタに入力する場合は、ファイルコンバータを使用して ELF 形式に変換してください。ただし、リンケージエディタ出力リロケータブルファイル(拡張子 rel)およびリロケータブルファイルを含むライブラリファイルは変換できません。
また、SYSROF 形式および ELF/DWARF1 形式ロードモジュールをサポートするデバッガを使用する場合は、ファイルコンバータを使用して ELF/DWARF2 形式ロードモジュールを、SYSROF または ELF/DWARF1 形式に変換してください。

(2) インクルードファイルの基点

ディレクトリ相対形式で指定されたインクルードファイル検索時、旧バージョンではコンパイラ起動ディレクトリを基点に検索していましたが、ソースファイルのあるディレクトリを基点に検索するように変更しました。

(3) C++プログラム

エンコード規則、実行方式を変更しましたので、旧バージョンのコンパイラで作成した C++オブジェクトファイルはリンクできません。必ずリコンパイルしてから使用してください。
また実行環境の設定で用いる、グローバルクラスオブジェクト初期処理／後処理のライブラリ関数名も変更になりました。SuperH RISC engine C/C++**コンパイラ、アセンブラ、最適化リンケージエディタユーザーズマニュアル**の「9.2.2 **実行環境の設定**」を参照し、修正してください。

(4) コモンセクションの廃止(アセンブリプログラム)

オブジェクトフォーマットの変更に伴い、コモンセクションのサポートを廃止しました。

(5) .END のエントリ指定(アセンブリプログラム)

.END でエントリ指定できるシンボルは外部定義シンボルだけになりました。

(6) モジュール間最適化

旧バージョンのコンパイラ(Ver.5.x 以前)、アセンブラ(Ver.4.x 以前)出力オブジェクトファイルは、モジュール間最適化の対象になりません。モジュール間最適化の対象にしたいファイルについては、必ずリコンパイル、リアセンブルしてください。

ホームページとサポート窓口<website and support,ws>

ルネサステクノロジホームページ
http://japan.renesas.com/

お問合せ先
http://japan.renesas.com/inquiry
csc@renesas.com

改訂記録<revision history,rh>

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2007.6.1	—	初版発行
2.00	2008.4.1	45	「A.12　Ver.9.1 から Ver.9.2 への追加機能 」　追記

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。