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：＜コンパイラ活用ガイド＞オプション編

本ドキュメントでは、SuperH RISC engine C/C++ コンパイラ V.9 におけるオプションについての説明を行います。

目次

1. 最適化オプション 2
1.1 基本オプション (スピード優先 / サイズ優先 / サイズ&スピード) 2
1.1.1 自動インライン展開 3
1.1.2 ループ展開最適化 7
1.1.3 シフトコード展開 8
1.1.4 転送コード展開 9
1.1.5 除算方式選択<SH-1 以外のマイコン> 11
1.1.6 非アライメントデータ転送 13
1.1.7 定数ロードの命令展開 15

1.2 性能向上が期待できる詳細オプション 17
1.2.1 アドレス領域の宣言 17
1.2.2 変数の配置指定 18
1.2.3 外部変数アクセス最適化 20
1.2.4 GBR 相対論理演算生成 23
1.2.5 最適化範囲分割 25
1.2.6 MAC レジスタ保証 26
1.2.7 リターン値の拡張 28
1.2.8 列挙型サイズ 30
1.2.9 switch 文展開方式 31

2. 便利なオプション 32
2.1 デバッグ情報出力モード 32
2.2 プリプロセッサ展開 34
2.3 外部変数の volatile 化 35
2.4 空ループ削除 37
2.5 無限ループ前の式削除 38
2.6 ビットフィールド並び順指定 39
2.7 構造体、共用体、クラスのアライメント数指定 40

ホームページとサポート窓口<website and support,ws> 41

1. 最適化オプション

コンパイラの最適化オプションには、スピード優先、サイズ優先、サイズ&スピード の 3 つの基本オプションと各最適化項目を詳細設定するための詳細オプションが存在します。1.1では基本オプションと基本オプションに連動する詳細オプションについて、1.2では特に性能向上が期待できる詳細オプションについて説明します。

なお、本ドキュメントのアセンブリ言語展開コードは、“code=asmcode” および “cpu=sh2” を指定して取得しています。“cpu” オプションが異なる場合はアセンブリ言語展開コードが異なる場合があります。また、アセンブリ言語展開コードは今後のコンパイラ改善などにより変わる可能性があります。参考データとしてご活用下さい。

1.1 基本オプション (スピード優先 / サイズ優先 / サイズ&スピード)

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最適化と実行速度を速くする最適化があります。実行速度を優先させたい場合は “speed” オプションを、サイズを優先させたい場合は “size” オプションを、実行速度とサイズのバランスを取りたい場合は “nospeed” オプション(デフォルト)を指定します。各オプションが指定されると、次の最適化が実施されます。

・“speed” が指定された場合
　　サイズ・実行速度ともに効果的な最適化 ＋ サイズ増加を伴うが実行速度に効果がある最適化
・“size” が指定された場合
　　サイズ・実行速度ともに効果的な最適化 ＋ 実行速度低下を伴うがサイズ削減に効果がある最適化
・“nospeed” が指定された場合
　　サイズ・実行速度ともに効果的な最適化

実行速度が要求される関数とサイズが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。

【書式】

SPeed
SIze
<u>NOSPeed</u>

[High-Performance Embedded Workshop(以後、ルネサス統合開発環境と呼びます)でのオプション設定方法]

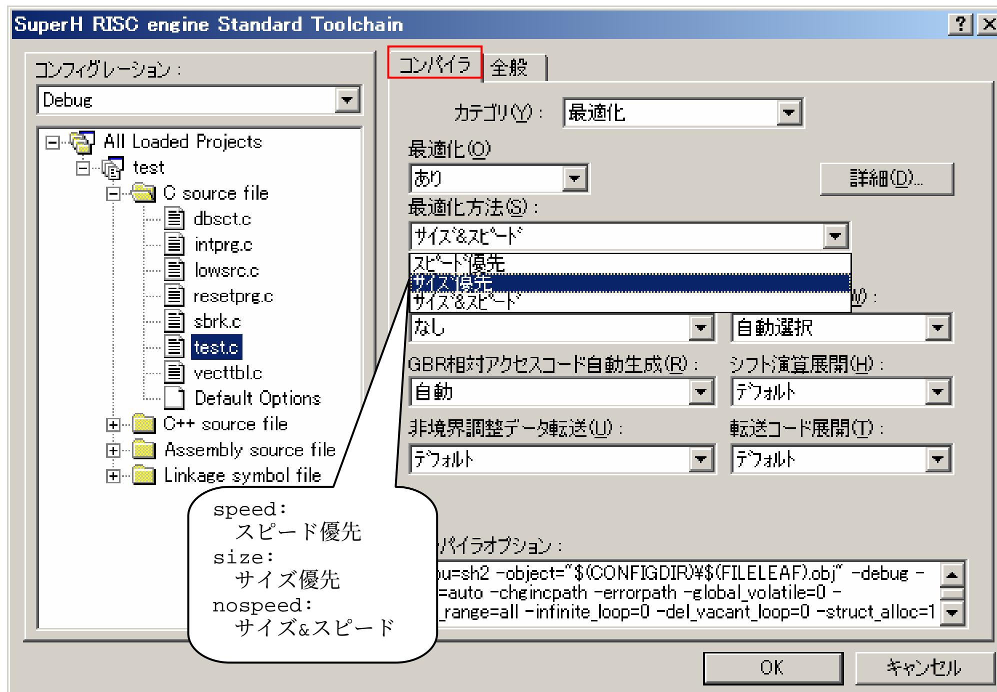

図 1-1

【補足】

実機での実行速度はコンパイラの生成コード以外にメモリアーキテクチャやキャッシュヒット率、割り込みなどの要因によって変化します。そのため、“speed” を指定した時が必ずしも最も速度が出るコードになるとは限りません。 本ドキュメントで紹介するさまざまなオプションは実機で効果を確認してください。

コンパイラの最適化項目の詳細を設定する詳細オプションの中には、基本オプションの設定によってデフォルト値が変更されるものがあります。デフォルト値が変更されるオプションを表 1-1 に示します。

表 1-1 デフォルト一覧

No	機能	nospeed(デフォルト)	size	speed	参照
1	自動インライン展開	noinline	noinline	Inline	1.1.1
2	ループ展開最適化	noloop	noloop	loop	1.1.2
3	シフトコード展開	命令展開	実行時ルーチン	命令展開	1.1.3
4	転送コード展開	命令展開	実行時ルーチン	命令展開	1.1.4
5	除算方式選択<SH-1 以外のマイコン>	命令展開	実行時ルーチン	命令展開	1.1.5
6	非アライメントデータ転送	命令展開	実行時ルーチン	命令展開	1.1.6
7	定数ロードの命令展開	リテラルデータ参照	リテラルデータ参照	命令展開	1.1.7

各オプションの詳細を以下に示します。

1.1.1 自動インライン展開

関数の自動インライン展開を行うかどうかを指定します。

“inline” オプションを指定すると自動インライン展開を行います。“inline=<数値>” で、プログラムサイズが何%増加するまでインライン展開を行うかを指定できます。例えば “inline=50” を指定した場合では、プログラムサイズが 50%増加するまで(1.5 倍になるまで)インライン展開を行います。数値指定をしない場合は、“inline=20” とみなします。

自動インライン展開は、呼び出し先関数のサイズが小さい関数が優先されます。

なお、#pragma inline が指定された関数は、自動インライン展開の指定に関わらずインライン展開が実施されます。ただし、自動インライン展開のサイズに関する上限判断は、#pragma inline によるインライン展開のサイズ増加分も含んで判断されます。

“noinline” オプションを指定した場合、自動インライン展開を行いません。

なお、次の条件を満たす関数は自動インライン展開されません。

1. 可変パラメータを持つ関数である。
2. 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。

インライン展開についての詳細は、
「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:
＜コンパイラ活用ガイド＞拡張機能編 1.2 関数のインライン展開指定」を参照してください。

【書式】

INLine [= <数値>]　　: **speed 指定時のデフォルト、デフォルトの数値は 20**
NOINLine　　: **size , nospeed 指定時のデフォルト**

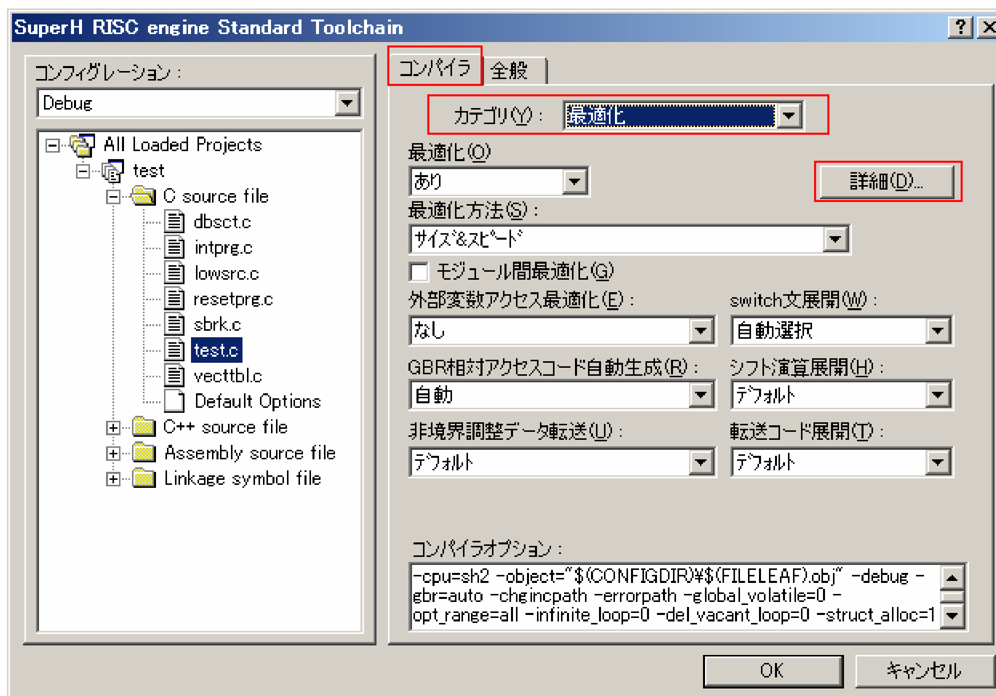

図 1-2

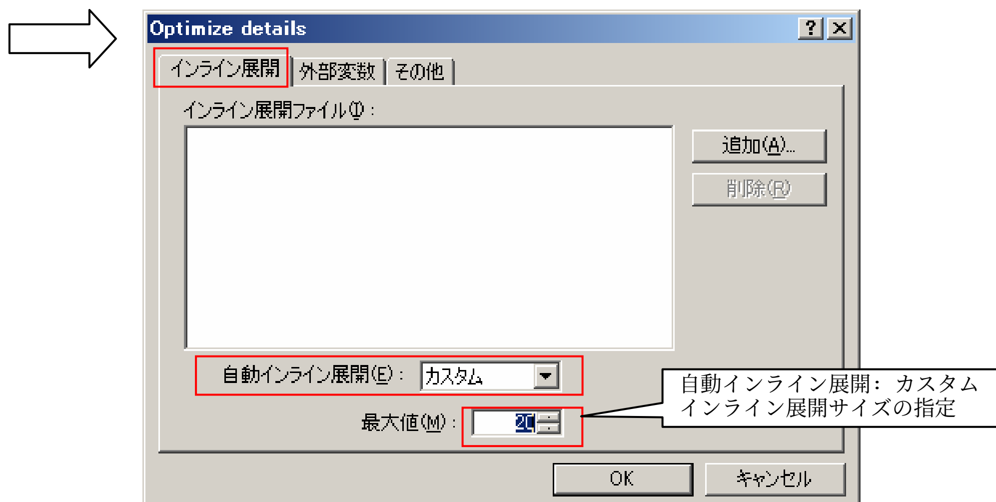

図 1-3

インライン展開はコンパイル時に展開される関数の定義が参照できる必要があります。そのため、通常のインライン展開では、同一ファイル内にある関数しか対象に出来ません。別ファイルにある関数を展開したい場合には、ファイル間インライン展開(file_inline=<ファイル名>[,…]) オプションを指定します。また、ファイル間インラインで指定された複数のファイルで同じ名前の extern 関数が定義されていた場合、動作は保証しません(任意に選ばれた 1 つの関数定義が用いられます)。

なお、ソースファイルと同一のファイルをインライン展開ファイルに指定した場合は、コンパイラが下記ウォーニングを出力し、該当ファイルをインライン展開ファイルの対象外にします。

C1315 (W) File_inline "ファイル名" ignored by same file as source file

[ルネサス統合開発環境でのオプション設定方法]

"コンパイラ"タブを選択 → [カテゴリ]に"最適化"を選択 → [詳細]を選択(図 1-2)すると表示されるダイアログで次のように設定します。

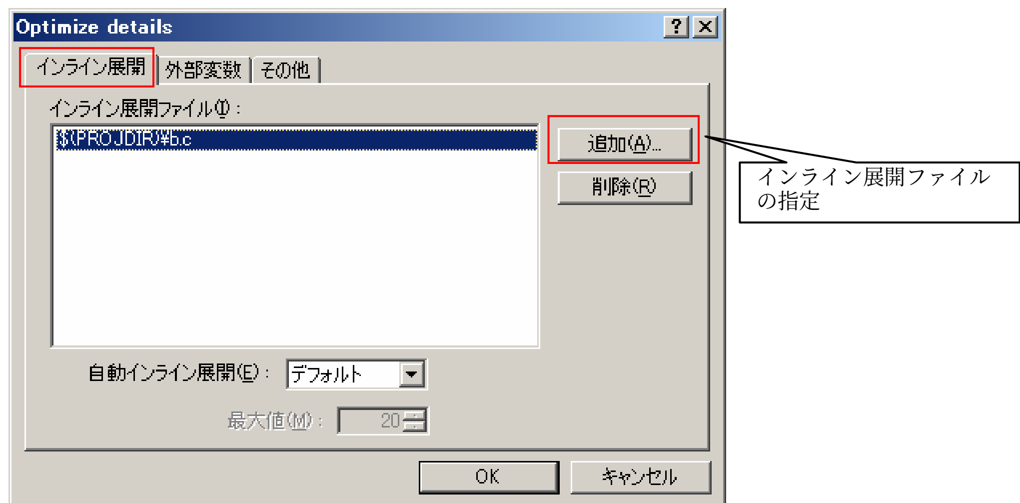

図 1-4

【例】

```
ソースコード
<a.c>
void func(void)
{
    g();
}
<b.c>
#pragma inline (g)
void g(void)
{
    h();
}

file_inline=b.cを指定したときのa.cの展開イメージ
void func(void)
{
    h();
}
```

カレントフォルダ以外のファイル間インライン展開ファイルを指定する場合は、ファイル間インライン展開フォルダ指定(file_inline_path=<パス名>[,…])をすると、ファイル間インライン展開ファイルのパス名を省略することができます。

ファイル間インライン展開対象ファイルの検索は、“file_inline_path” オプション指定フォルダ、カレントフォルダの順序で行います。

[ルネサス統合開発環境でのオプション設定方法]

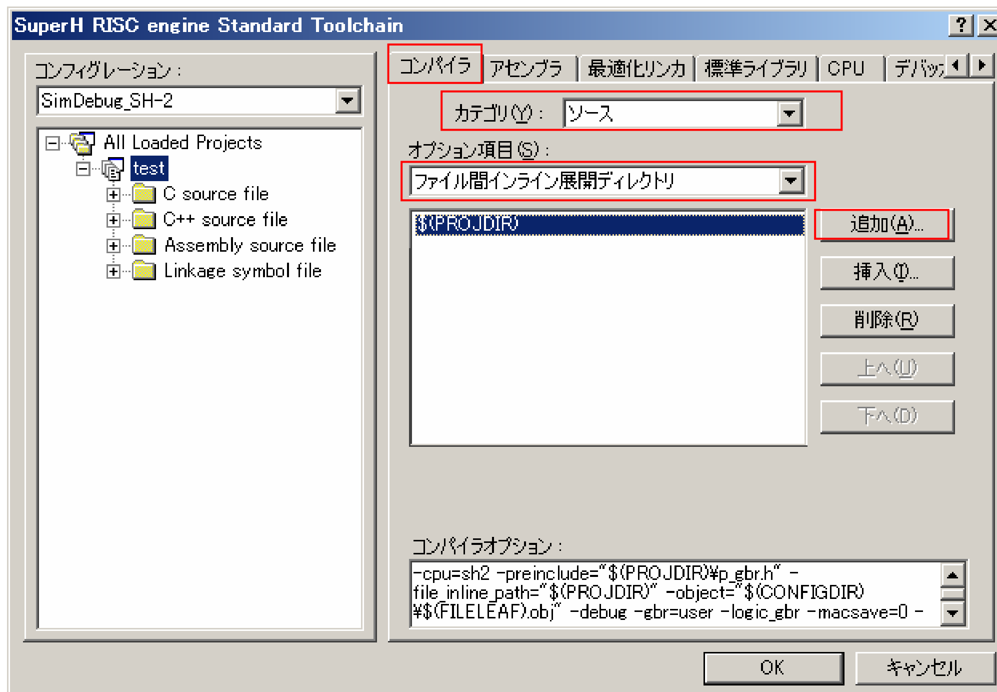

図 1-5

1.1.2 ループ展開最適化

ループ展開を行うかどうかを指定します。

"loop" オプションを指定するとループ展開が行われます。ループ展開については、

「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:

＜コンパイラ活用ガイド＞効率の良いプログラミング手法 4.1 ループ回数の削減」を参照してください。

ループ展開時の最大展開数を "max_unroll=<数値(1～32)>" で指定することができます。ループ展開最適化が有効な場合のデフォルトは 2 です。ループ展開最適化が無効の場合、"max_unroll" の指定は無視されます。

【書式】

LOop　　: speed 指定時のデフォルト、デフォルトの最大展開数 2
NOLOop　: size , nospeed 指定時のデフォルト

[ルネサス統合開発環境でのオプション設定方法]

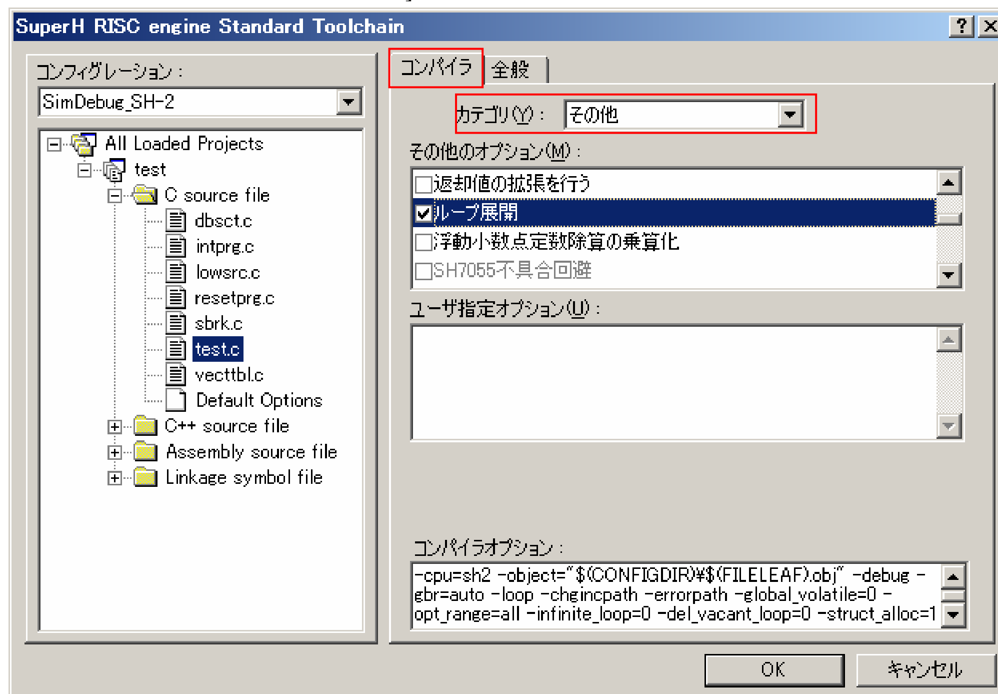

図 1-6

ループ展開時の最大展開数を指定する場合、"コンパイラ"タブを選択 → [カテゴリ]に"最適化"を選択 → [詳細]を選択(図 1-2)すると表示されるダイアログで次のように設定します。

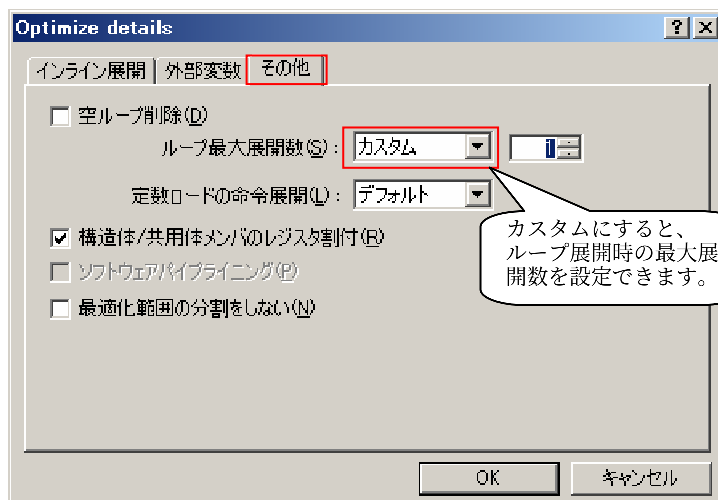

図 1-7

1.1.3 シフトコード展開

シフト演算を命令展開するか、実行時ルーチン呼び出しにするか選択します。

命令展開(inline)を指定した場合は常に命令展開をします。実行時ルーチン(runtime)を指定した場合、展開する命令が多ければ、実行時ルーチン呼び出しになります(展開コード数が５命令以下の場合、命令展開します)。

【書式】

SHIft = Inline : **speed , nospeed** 指定時のデフォルト

Runtime : **size** 指定時のデフォルト

[ルネサス統合開発環境でのオプション設定方法]

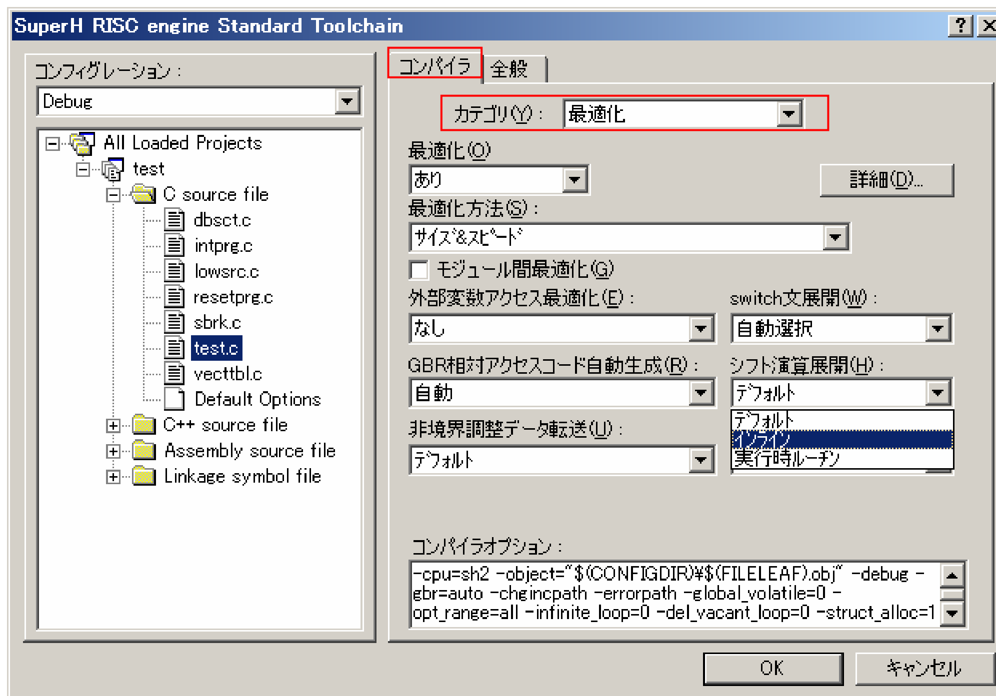

図 1-8

【例】

ソースコード
```
int var;

void f(void)
{
    var >>= 11;
}
```

アセンブリ展開コード(shift=inline指定)
```
_f:
        MOV.L       L11+2,R5    ; _var
        MOV.L       @R5,R2      ; var
        SHLR8       R2
        SWAP.W      R2,R2
        EXTS.B      R2,R6
        XTRCT       R6,R2
        SHAR        R2
        SHAR        R2
        SHAR        R2
        RTS
        MOV.L       R2,@R5      ; var
L11:
        .RES.W      1
        .DATA.L     _var
```

アセンブリ展開コード(shift=runtime指定)
```
_f:
        STS.L       PR,@-R15
        MOV.L       L11,R5      ; _var
        MOV.L       L11+4,R2    ; __sta_sftrall
        JSR         @R2
        MOV.L       @R5,R0      ; var
        LDS.L       @R15+,PR
        RTS
        MOV.L       R0,@R5      ; var
L11:
        .DATA.L     _var
        .DATA.L     __sta_sftrall
```

1.1.4 転送コード展開

構造体/配列/クラスの転送コードを命令展開するか、実行時ルーチン呼び出しにするか選択します。

命令展開(inline)を指定した場合は常に命令展開をします。実行時ルーチン(runtime)を指定した場合、展開する命令が多ければ、実行時ルーチン呼び出しになります。
(2 組のロード・ストア(４命令)以下でコピー可能な場合、命令展開します)

【書式】

BLOckcopy = Inline **: speed , nospeed** 指定時のデフォルト
Runtime **: size** 指定時のデフォルト

[ルネサス統合開発環境でのオプション設定方法]

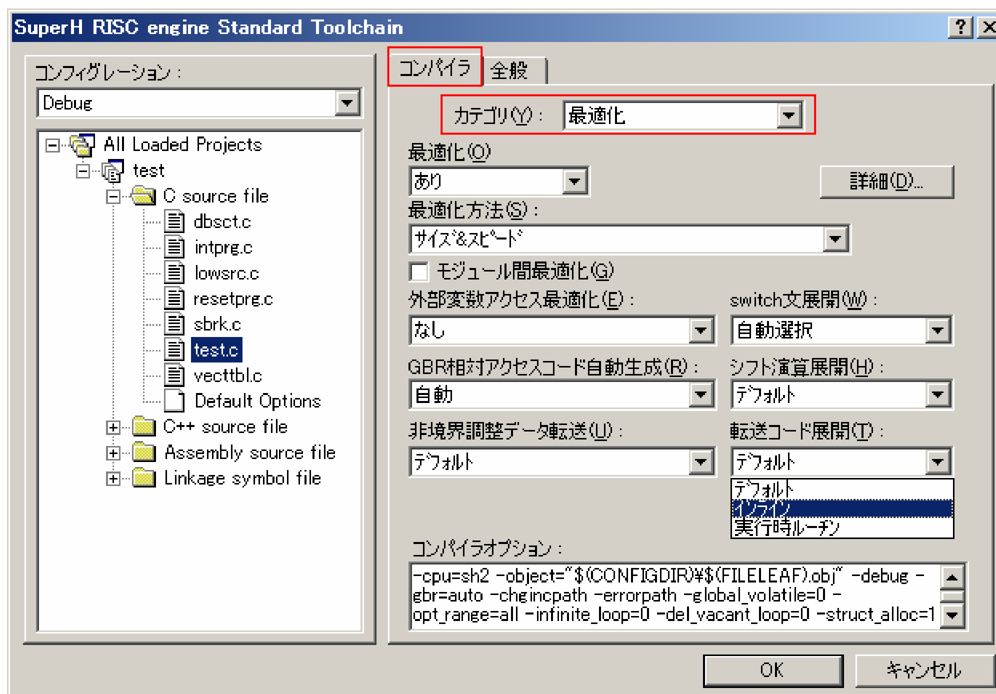

図 1-9

【例】

ソースコード
```
struct _ST_ {
    char a[5];
} x;

extern void g(struct _ST_);

void f(void)
{
    g(x);
}
```

アセンブリ展開コード(blockcopy=inline指定)
```
_f:
          STS.L       PR,@-R15
          ADD         #-8,R15
          MOV.L       L11+2,R6    ; _x
          MOV.L       L11+6,R4    ; _g
          MOV.B       @(1,R6),R0 ; (part of)x
          MOV.B       @R6,R1      ; (part of)x
          MOV.B       R0,@(1,R15)
          MOV.B       @(2,R6),R0 ; (part of)x
          MOV.B       R1,@R15
          MOV.B       R0,@(2,R15)
          MOV.B       @(3,R6),R0 ; (part of)x
          MOV.B       R0,@(3,R15)
          MOV.B       @(4,R6),R0 ; (part of)x
          JSR         @R4
          MOV.B       R0,@(4,R15)
          ADD         #8,R15
          LDS.L       @R15+,PR
          RTS
          NOP
L11:
          .RES.W      1
          .DATA.L     _x
          .DATA.L     _g
```

アセンブリ展開コード(blockcopy=runtime指定)
```
_f:
          STS.L       PR,@-R15
          ADD         #-8,R15
          MOV.L       L11,R2      ; _x
          MOV.L       L11+4,R5    ; __slow_mvn
          MOV         R15,R1
          JSR         @R5
          MOV         #5,R0       ; H'00000005
          MOV.L       L11+8,R1    ; _g
          JSR         @R1
          NOP
          ADD         #8,R15
          LDS.L       @R15+,PR
          RTS
          NOP
L11:
          .DATA.L     _x
          .DATA.L     __slow_mvn
          .DATA.L     _g
```

1.1.5 除算方式選択<SH-1 以外のマイコン>

プログラム中の整数型除算､剰余算の方式を選択します。本オプションは､マイコンが SH-1 の場合は無効になります。

“division=cpu=inline” を指定した場合、定数除算は乗算に変換してインライン展開します。変数除算は、マイコンが SH-2A または SH2A-FPU の場合は除算命令を、それ以外のマイコンの場合は実行時ルーチンをそれぞれ選択します。

“division=cpu=runtime” を指定した場合、2 のべき乗の定数除算は命令展開します。それ以外の除算について、マイコンが SH-2A または SH2A-FPU の場合は除算命令を、それ以外のマイコンの場合は実行時ルーチンをそれぞれ選択します。

【書式】

`DIvision = Cpu =`	**Inline**	**: speed , nospeed** 指定時のデフォルト
	Runtime	**: size** 指定時のデフォルト

[ルネサス統合開発環境でのオプション設定方法]

図 **1-10**

図 **1-11**

【例】

ソースコード

```
int x;
void f(int y)
{
    x = y/3;
}
```

アセンブリ展開コード(division=cpu=inline指定)

```
_f:
        STS.L       MACL,@-R15
        STS.L       MACH,@-R15
        MOV.L       L11,R1      ; H'55555556
        MOV.L       L11+4,R5    ; _x
        DMULS.L     R4,R1
        STS         MACH,R6
        MOV         R6,R0
        ROTL        R0
        AND         #1,R0
        ADD         R0,R6
        MOV.L       R6,@R5      ; x
        LDS.L       @R15+,MACH
        RTS
        LDS.L       @R15+,MACL
L11:
        .DATA.L     H'55555556
        .DATA.L     _x
```

アセンブリ展開コード(division=cpu=runtime指定)

```
_f:
        STS.L       PR,@-R15
        MOV.L       L11+2,R2    ; __divls
        MOV         R4,R1
        JSR         @R2
        MOV         #3,R0       ; H'00000003
        MOV.L       L11+6,R5    ; _x
        LDS.L       @R15+,PR
        RTS
        MOV.L       R0,@R5      ; x
L11:
        .RES.W      1
        .DATA.L     __divls
        .DATA.L     _x
```

1.1.6 非アライメントデータ転送

アライメント数が 1 の構造体・共用体・クラスのデータ転送を命令展開するか、実行時ルーチン呼び出しにするか選択します。命令展開(inline)を指定した場合は常に命令展開をします。実行時ルーチン(runtime)を指定した場合、展開する命令が多ければ、実行時ルーチン呼び出しになります。

【書式】

Unaligned = Inline	**: speed , nospeed** 指定時のデフォルト
Runtime	**: size** 指定時のデフォルト

[ルネサス統合開発環境でのオプション設定方法]

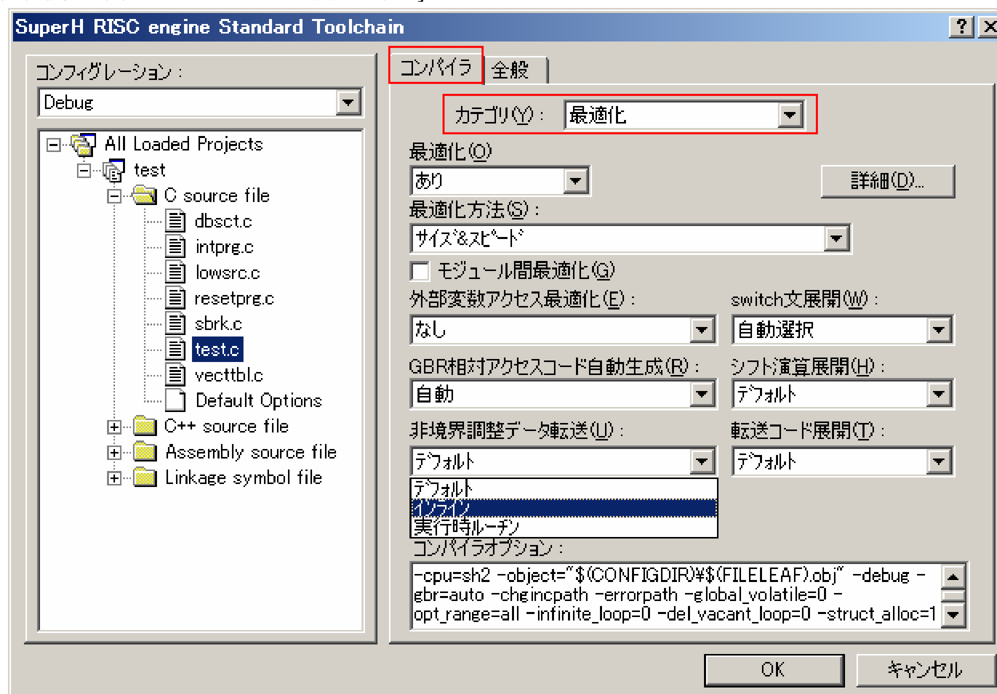

図 1-12

【例】

ソースコード

```
#pragma pack 1
struct {
   char  a;
   short b;
   int   c;
} x,y;
#pragma unpack

void func(void)
{
    x.c = y.c;
}
```

アセンブリ展開コード(unaligned=inline指定)

```
_func:
          MOV.L       L11+2,R3    ; H'00000003+_y
          MOV.L       L11+6,R7    ; H'00000003+_x
          MOV.B       @R3,R4      ; y.c
          MOV.B       @(1,R3),R0  ; y.c
          MOV.B       R4,@R7      ; x.c
          MOV.B       R0,@(1,R7)  ; x.c
          MOV.B       @(2,R3),R0  ; y.c
          MOV.B       R0,@(2,R7)  ; x.c
          MOV.B       @(3,R3),R0  ; y.c
          RTS
          MOV.B       R0,@(3,R7)  ; x.c
L11:
          .RES.W      1
          .DATA.L     H'00000003+_y
          .DATA.L     H'00000003+_x
```

アセンブリ展開コード(unaligned=runtime指定)

```
_func:
          STS.L       PR,@-R15
          MOV.L       L11+2,R2    ; H'00000003+_y
          MOV.L       L11+6,R1    ; H'00000003+_x
          MOV.L       L11+10,R7   ; __slow_mvn
          JSR         @R7
          MOV         #4,R0       ; H'00000004
          LDS.L       @R15+,PR
          RTS
          NOP
L11:
          .RES.W      1
          .DATA.L     H'00000003+_y
          .DATA.L     H'00000003+_x
          .DATA.L     __slow_mvn
```

1.1.7 定数ロードの命令展開

定数ロードを命令展開(inline)するか、リテラルロード(literal)にするかを選択します。
SH マイコンは命令内に 8 ビット(SH-2A では 20 ビット)の定数を持つことができます。2 バイト、4 バイトの定数を扱う時は下記のいずれかの方式が用いられます。

リテラルロード ： メモリ上に用意した定数データ(リテラルデータ)を参照する
命令展開 ： 命令内の 8 ビット定数から演算によって 2 バイト、4 バイト定数を生成する

リテラルロードはプログラムサイズが小さくなる傾向にあります。命令展開はメモリアクセスが少なくなる傾向にあります。リテラルロードを指定した場合、2 バイト以上の定数はリテラルロードで求めます。
命令展開を指定した場合、すべての 1 バイト・2 バイト定数および一部の 4 バイト定数を命令展開で求めます。“size” および “nospeed” を指定した場合は、当該定数が次の条件を満たす時は命令展開、満たさない時はリテラルロードとなります。

2 バイトの定数 ： 2 命令以内の演算で求まる
4 バイトの定数 ： 3 命令以内の演算で求まる

【書式】

CONST_Load = Inline **: speed 指定時のデフォルト**
Literal **: size , nospeed 指定時のデフォルト**
(ただし、size、nospeed 指定時、条件によっては命令展開を行う)

[ルネサス統合開発環境でのオプション設定方法]
“コンパイラ”タブを選択 → [カテゴリ]に“最適化”を選択 → [詳細]を選択(図 1-2)すると表示されるダイアログで次のように設定します。

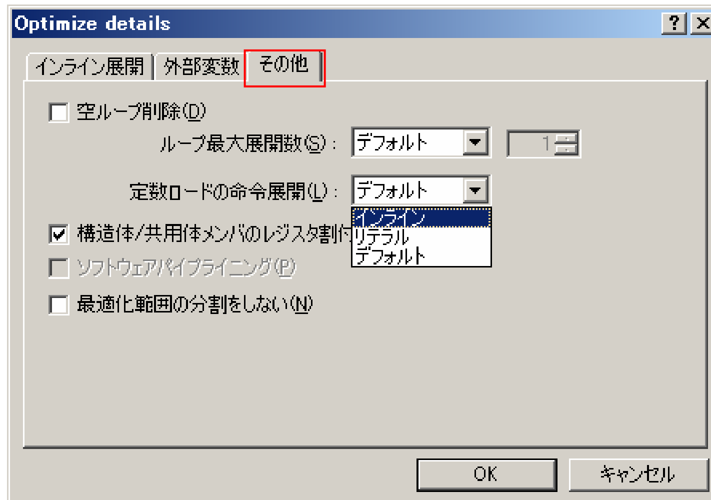

図 1-13

【例】

ソースコード
```
int a;

void func(void)
{
    a = 0x4567;
}
```

アセンブリ展開コード(const_load=inline指定)
```
_func:
          MOV         #69,R2      ; H'00000045
          SHLL8       R2
          MOV.L       L11,R6      ; _a
          ADD         #103,R2
          RTS
          MOV.L       R2,@R6      ; a
L11:
          .DATA.L     _a
```

アセンブリ展開コード(const_load=literal指定)
```
_func:
          MOV.L       L11+4,R6    ; _a
          MOV.W       L11,R2      ; H'4567
          RTS
          MOV.L       R2,@R6      ; a
L11:
          .DATA.W     H'4567
          .RES.W      1
          .DATA.L     _a
```

【補足】

本オプションはターゲットシステムのメモリアーキテクチャにより、最適な設定が異なります。メモリアクセスが速いシステムでは、リテラルアクセスが実行速度において有利となる傾向にあります。メモリアクセスが遅いシステムでは、命令展開が実行速度において有利となる傾向にあります。

1.2　性能向上が期待できる詳細オプション

本章では、各最適化項目を詳細設定するための詳細オプションの中で、特に性能向上が期待できるオプションについて説明します。

表 1-2 性能向上に特に期待できるオプション

No	機能	オプション	サイズ	実行速度	参照
1	アドレス領域の宣言	abs16/abs20/abs28/abs32	◎	○	1.2.1
2	変数の配置指定	stuff/nostuff	◎	-	1.2.2
3	外部変数アクセス最適化	map/smap	◎	◎	1.2.3
4	GBR 相対論理演算生成	logic_gbr	○	○	1.2.4
5	最適化範囲分割	scope/noscope	△	△	1.2.5
6	MACレジスタ保証	macsave	○	○	1.2.6
7	リターン値の拡張	rtnext/nortnext	△	△	1.2.7
8	列挙型サイズ	auto_enum	○	×	1.2.8
9	switch 文展開方式	case	△	△	1.2.9

◎　**特に有効である**
○　**有効である**
△　**良くなる場合と悪くなる場合がある**
×　**劣化する**
-　**影響なし**

1.2.1　アドレス領域の宣言

アドレス領域の指定(abs16/20/28/32)は、変数や関数のアドレスが 16/20/28/32 ビットであることをコンパイラに指示するオプションです。デフォルトでは 32 ビットです。

アドレス領域の指定は #pragma abs16/abs20/abs28/abs32 でも指定できます。オプションと #pragma が同時に指定された場合は、#pragma が優先されます。

機能の詳細は、
「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：
　＜コンパイラ活用ガイド＞拡張機能編 1.1 アドレス領域の指定」を参照してください。

【書式】

ABs16 = { Program | Const | Data | Bss | Run | All }[,…]
ABS20 = { Program | Const | Data | Bss | Run | All }[,…]
ABS28 = { Program | Const | Data | Bss | Run | All }[,…]
ABS32 = { Program | Const | Data | Bss | Run | All }[,…]

[ルネサス統合開発環境でのオプション設定方法]

“コンパイラ”タブを選択 → [カテゴリ]に“オブジェクト”を選択 → [詳細]を選択(図 1-10)すると表示されるダイアログで次のように設定します。

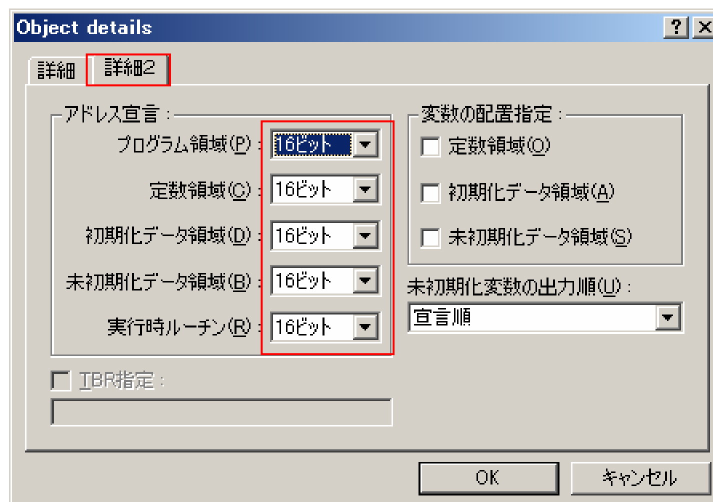

図 1-14

1.2.2 変数の配置指定

"stuff"オプションを使用することで、変数をアライメント数別のセクションに配置することができます。これにより境界調整用の空き領域(パディング)がなくなりメモリを節約することができます。

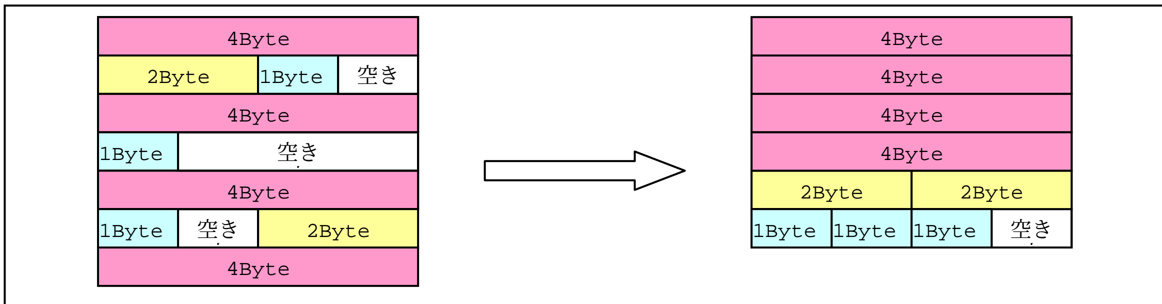

図 1-15

"stuff" オプションではセクション種別を指定することができます。指定したセクション種別に属する変数をデータサイズに応じて、アライメント数 4 のセクション、2 のセクション、1 のセクションに配置します。セクション種別を省略した場合はすべての種別が対象になります。

各セクション内のデータは定義順に出力されます("bss_order=declaration" 指定は無効)。

"nostuff" を指定した場合は全ての変数をアライメント数4のセクションに配置します。各セクション内のデータはC,Dセクションは定義順、B セクションは"bss_order"に従います。本オプションのデフォルトは "nostuff" です。

表 1-3 変数のサイズとセクション名の関係

	セクション種別	**デフォルトセクション**	**変数のサイズ**		
			4n	4n+2	2n+1
定数領域	const	C	C$4	C$2	C$1
初期化データ領域	data	D	D$4	D$2	D$1
未初期化データ領域	bss	B	B$4	B$2	B$1

セクション名を変更している場合は、変更したセクション名に、$4、$2、$1 が付きます。

【書式】

STUff [= <セクション種別>[,...]]
<u>NOSTuff</u>
<セクション種別>：**{ Bss | Data | Const }**

[ルネサス統合開発環境でのオプション設定方法]

"コンパイラ"タブを選択 → [カテゴリ]に"オブジェクト"を選択 → [詳細]を選択(図 1-10)すると表示されるダイアログで次のように設定します。

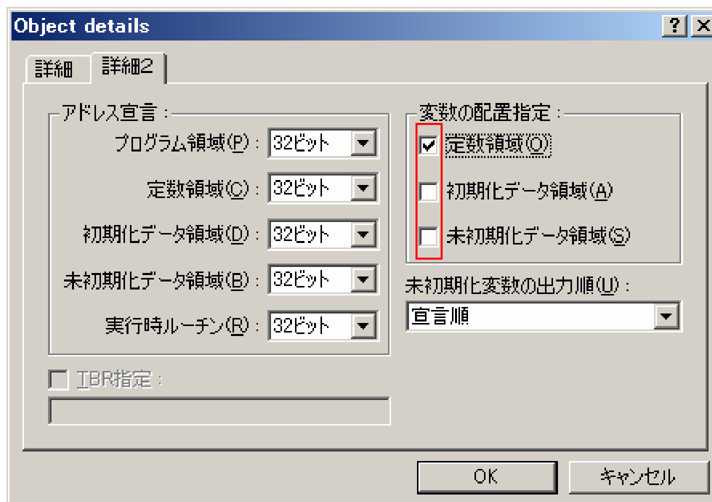

図 1-16

【例】

ソースコード

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

アセンブリ展開コード(nostuff指定)

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

<table>
<tr><td colspan="4">a</td></tr>
<tr><td colspan="2">b</td><td>c</td><td>空き</td></tr>
<tr><td colspan="4">d</td></tr>
<tr><td>e</td><td colspan="3">空き</td></tr>
<tr><td colspan="4">f</td></tr>
<tr><td>g</td><td>空き</td><td colspan="2">h</td></tr>
<tr><td colspan="4">i</td></tr>
</table>

アセンブリ展開コード(stuff指定)

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

<table>
<tr><td colspan="4">a</td></tr>
<tr><td colspan="4">d</td></tr>
<tr><td colspan="4">f</td></tr>
<tr><td colspan="4">i</td></tr>
<tr><td colspan="2">b</td><td colspan="2">h</td></tr>
<tr><td>c</td><td>e</td><td>g</td><td>空き</td></tr>
</table>

1.2.3 外部変数アクセス最適化

“map” オプションおよび “smap” オプションを用いると、外部変数へのアクセスがベースとなる外部変数からの相対アクセスとなります。これにより、外部変数アドレスのロード処理が不要となり、実行速度が向上します。また、アドレス値のリテラルを省略することができるため、プログラムサイズが削減されます。“gbr=auto” が指定されている場合は、通常の MOV 命令よりも相対値が大きい GBR 相対命令で外部変数をアクセスすることもあります。外部変数アクセス最適化は速度・サイズのどちらにも効果が期待できる最適化です。

“map” 最適化では、最適化リンケージエディタが生成する外部シンボル割り付け情報を利用して最適化を実施します。そのため、２回コンパイルが必要となります。

“smap” 最適化では、コンパイル対象のファイル内で定義された外部変数に対してのみ、外部変数最適化を実施します。リンケージエディタが生成する外部シンボル割り付け情報を使用しないため、1 回のコンパイルで最適化が実施できます。

“map” 最適化の方が “smap” 最適化よりも最適化の効果が高いですが、2 回コンパイルが必要であり、アドレス解決された実行モジュール(abs,mot 等)を作成する時にしか実施できません。“smap” 最適化は最適化の対象がファイル内に定義された外部変数に限定されますが、1 回のコンパイルで実施でき、ライブラリファイルなどのアドレス解決されていないファイルを生成する場合にも最適化を実施できます。

表 1-4 map と smap の特徴

オプション	2 回コンパイル	ビルド時間	最適化リンケージエディタ生成 外部シンボル割り付け情報ファイル	最適化効果	アドレス解決
map	必要	長い	必要	smap より高い	必要
smap	不要	短い	不要	map より低い	不要

【書式】

- **モジュール間指定**

MAP = <ファイル名>

“map” オプションを指定しないで 1 回コンパイルし、リンク時に “map=<ファイル名>” を指定し外部シンボル割り付け情報ファイルを作成してください。2 回目のコンパイルで、外部シンボル割り付け情報ファイルを指定(map=<ファイル名>)してコンパイルしてください。

外部変数もしくは静的変数の定義順を変更した場合は、外部シンボル割り付け情報ファイルを生成し直す必要があります。

次の条件で 2 回目のコンパイルを行った場合の動作は保証しません。

– オプション指定で「1 回目のコンパイルで指定したオプション+ “map”オプション」以外を指定した場合
– 1 回目のコンパイルで指定したソースファイルと差異のあるソースファイルを指定した場合

- **モジュール内指定**

SMap

[ルネサス統合開発環境でのオプション設定方法]

外部変数アクセス最適化の設定を[モジュール間]→[モジュール間]以外／[モジュール間]以外→[モジュール間]に設定しなおすと、ウォーニングが表示されます。これは、リンケージエディタの外部シンボル割り付け情報ファイル出力を自動的に有効／無効にするためです。

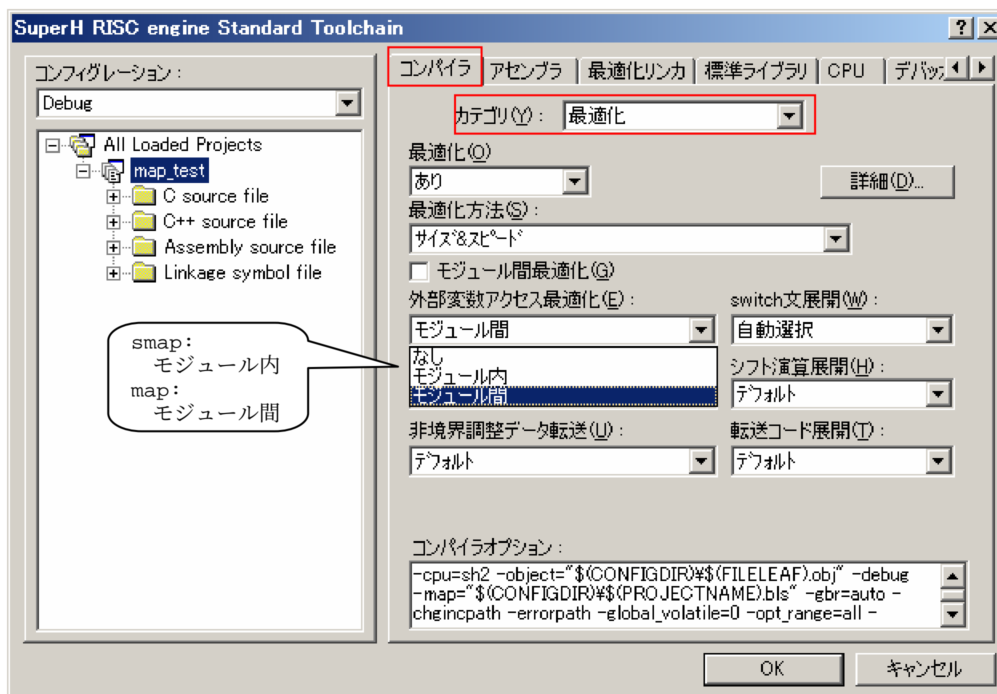

図 1-17

【例 1】

変数割り付け順序を意識して、連続して割り付けられる変数を同一レジスタからの相対でアクセスします。

ソースコード
```
int a,b;
void f(void)
{
    a=0;
    b=0;
}
```

アセンブリ展開コード(map/smap未指定)
```
_f:
    MOV.L       L11,R1      ; _a
    MOV.L       L11+4,R4    ; _b
    MOV         #0,R2       ; H'00000000
    MOV.L       R2,@R1      ; a
    RTS
    MOV.L       R2,@R4      ; b
L11:
    .DATA.L     _a
    .DATA.L     _b
```

アセンブリ展開コード(map/smap指定)
```
_f:
    MOV.L       L11+2,R6    ; _a
    MOV         #0,R2       ; H'00000000
    MOV.L       R2,@R6      ; a
    RTS
    MOV.L       R2,@(4,R6)  ; b
L11:
    .RES.W      1
    .DATA.L     _a
```

【例 2】

“gbr=auto” オプション(デフォルト)指定時、外部変数アクセスのベースとして、GBR を使用します。

ソースコード
```
int a[100];
void f(void)
{
    a[0]=0;
    a[50]=0;
    a[51]=0;
    a[52]=0;
}
```

アセンブリ展開コード(map/smap未指定)
```
_f:
        MOV.L       L11+2,R5    ; _a
        MOV         #-56,R0     ; H'FFFFFFC8
        MOV         #0,R4       ; H'00000000
        EXTU.B      R0,R0
        MOV.L       R4,@R5      ; a[]
        MOV.L       R4,@(R0,R5); a[]
        ADD         #4,R0
        MOV.L       R4,@(R0,R5); a[]
        ADD         #4,R0
        RTS
        MOV.L       R4,@(R0,R5); a[]
L11:
        .RES.W      1
        .DATA.L     _a
```

アセンブリ展開コード(map/smap指定)
```
_f:
        STC         GBR,@-R15
        MOV.L       L11,R0      ; _a
        LDC         R0,GBR
        MOV         #0,R0       ; H'00000000
        MOV.L       R0,@(0,GBR); a[]
        MOV.L       R0,@(200,GBR); a[]
        MOV.L       R0,@(204,GBR); a[]
        MOV.L       R0,@(208,GBR); a[]
        RTS
        LDC         @R15+,GBR
L11:
        .DATA.L     _a
```

1.2.4 GBR 相対論理演算生成

#pragma gbr_base/gbr_base1 指定の無い外部変数に対して、GBR 相対の論理演算コード化が可能な場合、GBR 相対アクセスします。

GBR 相対の論理演算コード化が可能なケースは以下の通りです。

- char/unsigned char 型の外部変数のビット演算(AND、OR、XOR)
- 外部変数のビットフィールドの参照

【書式】

LOGIc_gbr

本オプションは "gbr=user" 指定時のみ有効です。本オプションを使用する場合は、リンケージエディタで"$G0"セクションを配置し、あらかじめ GBR レジスタに "$G0" セクションの先頭アドレスを設定しておくことが必要です。組み込み関数 set_gbr() を使用し設定してください。

【例 1】

ソースコード

```
char a;

void func(void)
{
    a &= 0x0f;
}
```

アセンブリ展開コード(logic_gbr未指定)

```
_func:
          MOV.L       L11+2,R6     ; _a
          MOV.B       @R6,R0       ; a
          AND         #15,R0
          RTS
          MOV.B       R0,@R6       ; a
L11:
          .RES.W      1
          .DATA.L     _a
```

アセンブリ展開コード(logic_gbr指定)

```
_func:
          MOV.L       L11+2,R0     ; _a-(STARTOF $G0)
          RTS
          AND.B       #15,@(R0,GBR); a
L11:
          .RES.W      1
          .DATA.L     _a-(STARTOF $G0)
```

【例 2】

ソースコード

```
struct {
    unsigned char a:1;
    unsigned char b:1;
} x;

void func(void)
{
    if (x.a) {
        x.b = 1;
    }
}
```

アセンブリ展開コード(logic_gbr未指定)

```
_func:
          MOV.L       L13,R5      ; _x
          MOV.B       @R5,R0      ; (part of)x
          TST         #128,R0
          BT          L12
          OR          #64,R0
          MOV.B       R0,@R5      ; (part of)x
L12:
          RTS
          NOP
L13:
          .DATA.L     _x
```

アセンブリ展開コード(logic_gbr指定)

```
_func:
          MOV.L       L13,R0      ; _x-(STARTOF $G0)
          TST.B       #128,@(R0,GBR); (part of)x
          BT          L12
          OR.B        #64,@(R0,GBR); (part of)x
L12:
          RTS
          NOP
L13:
          .DATA.L     _x-(STARTOF $G0)
```

1.2.5 最適化範囲分割

関数の最適化範囲を複数に分割してコンパイルするか、分割せずにコンパイルするかを指定します。
“scope” を指定した場合、大きな関数は最適化範囲を分割してコンパイルされる可能性があります。
“noscope” を指定して最適化範囲を分割しなければ、関数全体にわたって最適化を実施できるため、一般的にはサイズ、実行速度共に性能が向上します。ただし、レジスタが不足すると逆に性能が低下する場合もあります。
本オプションは、プログラムによってコード効率が良くなる場合と悪くなる場合があるため、性能チューニング時にどちらも試してみてください。
なお、最適化範囲を複数に分割せずにコンパイルするとコンパイル時間が長くなります。

【書式】

```
SCOpe
NOSCope
```

最適化範囲が分割されたかは、インフォメーションレベルメッセージで確認できます。インフォメーションレベルメッセージは “message” オプションを指定すると有効となります。
最適化範囲が分割されている場合、下記のメッセージが表示されます。
C0101 (I) Optimizing range divided in function "関数名"

[ルネサス統合開発環境でのオプション設定方法]
“コンパイラ”タブを選択 → [カテゴリ]に“最適化”を選択 → [詳細]を選択(図 1-2)すると表示されるダイアログで次のように設定します。

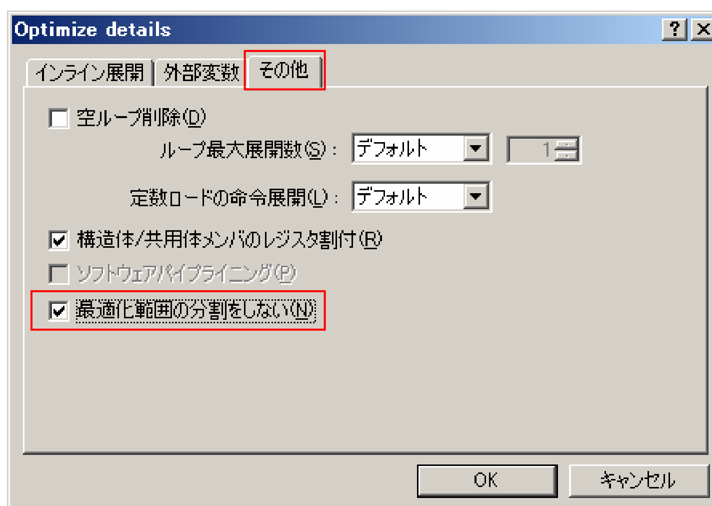

図 1-18

1.2.6 MAC レジスタ保証

MACH・MACL レジスタを関数の出入口で保証するレジスタとして扱うかどうかを指定することができます。

関数の出入口で保証する(macsave=1)場合は、関数の出入口でこれらのレジスタの退避／回復を行います。関数の出入口で保証しない(macsave=0)場合は、関数の呼び出し側でこれらのレジスタの退避／回復を行います。

現在のコンパイラの最適化処理では、“macsave=0” の方がサイズ・実行速度ともに良くなる傾向にあります。しかし、旧バージョンとの互換性のため、デフォルト値は “macsave=1” となっています。

“macsave=0” に変更することで性能が向上する場合があるため、試してみてください。

【書式】

```
Macsave = { 0 | 1 }
```

(注意)

MAC レジスタ保証ありでコンパイルした関数から MAC レジスタ保証なしでコンパイルした関数を呼び出すことはできません。MAC レジスタ保証なしでコンパイルした関数から MAC レジスタ保証ありでコンパイルした関数を呼び出すことは可能です。

[ルネサス統合開発環境でのオプション設定方法]

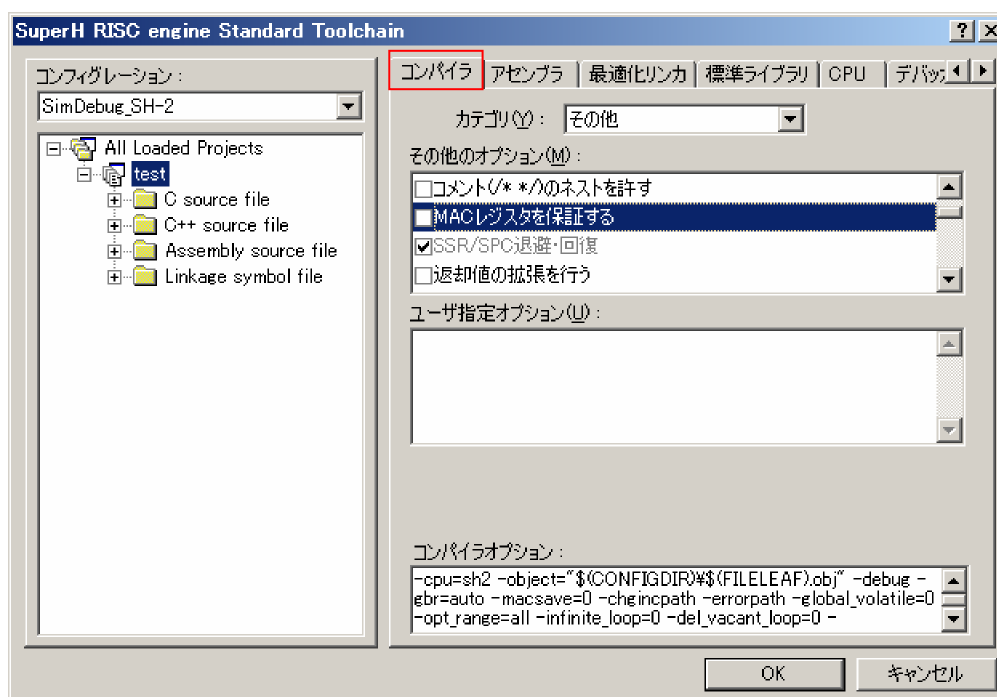

図 1-19

【例】

ソースコード

```
int sum;
int func(short a, short b)
{
   sum += a * b;

   return sum;
}
```

アセンブリ展開コード(macsave=1指定)

```
_func:
          STS.L       MACL,@-R15
          MULS.W      R4,R5
          MOV.L       L11+2,R1    ; _sum
          MOV.L       @R1,R0      ; sum
          STS         MACL,R2
          ADD         R2,R0
          MOV.L       R0,@R1      ; sum
          RTS
          LDS.L       @R15+,MACL
L11:
          .RES.W      1
          .DATA.L     _sum
```

アセンブリ展開コード(macsave=0指定)

```
_func:
          MULS.W      R4,R5
          MOV.L       L11+2,R1    ; _sum
          MOV.L       @R1,R0      ; sum
          STS         MACL,R2
          ADD         R2,R0
          RTS
          MOV.L       R0,@R1      ; sum
L11:
          .RES.W      1
          .DATA.L     _sum
```

1.2.7 リターン値の拡張

関数のリターン型が char、unsigned char、short、unsigned short 型のとき、呼び出された関数で符号拡張/ゼロ拡張を実行する(rtnext)か、呼び出し側で符号拡張/ゼロ拡張を実行する(nortnext)かを指定することができます。

デフォルトでは、呼び出し側で符号拡張/ゼロ拡張をします。

関数が複数回呼び出される場合、リターン値は呼び出し側で拡張するよりも、呼び出された側で拡張する方が、拡張のコードが 1 回で済みコード効率が良くなる傾向にあります。呼び出し側で拡張する場合は、無駄な拡張を削除する最適化が実施され易くなる傾向にあります。本オプションの最適な設定はプログラムの構成により異なるため、どちらも試してみてください。

本オプションは、プロジェクト全体で統一する必要があります。

【書式】

RTnext
<u>NORTnext</u>

[ルネサス統合開発環境でのオプション設定方法]

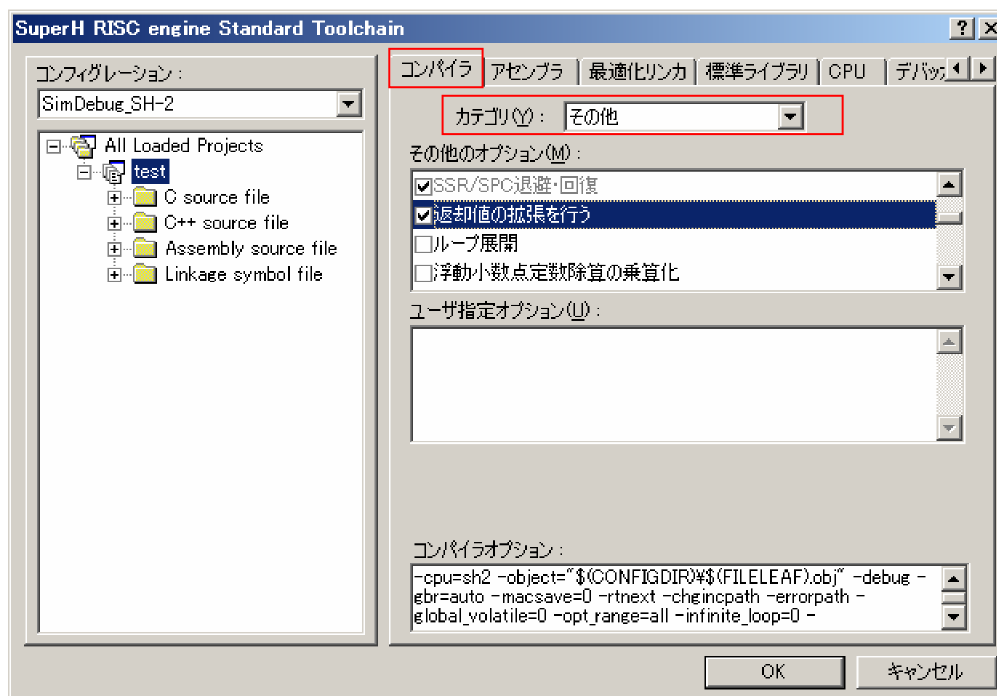

図 1-20

【例】

ソースコード

```
short x,y;
int   i,j,k;

short f(short a,  short b)
{
    return a * b;
}

void g(void)
{
    i = f(x,y);
    j = f(x,y);
    k = f(x,y);
}
```

アセンブリ展開コード(nortnext指定)

```
_f:
          STS.L       MACL,@-R15
          MULS.W      R4,R5
          STS         MACL,R0
          RTS
          LDS.L       @R15+,MACL
_g:
          MOV.L       R13,@-R15
          MOV.L       R14,@-R15
          STS.L       PR,@-R15
          MOV.L       L12,R13      ; _y
          MOV.L       L12+4,R14    ; _x
          MOV.W       @R13,R5      ; y
          BSR         _f
          MOV.W       @R14,R4      ; x
          MOV.L       L12+8,R2     ; _i
          EXTS.W      R0,R1
          MOV.W       @R13,R5      ; y
          MOV.W       @R14,R4      ; x
          BSR         _f
          MOV.L       R1,@R2       ; i
          MOV.L       L12+12,R1    ; _j
          EXTS.W      R0,R7
          MOV.W       @R13,R5      ; y
          MOV.W       @R14,R4      ; x
          BSR         _f
          MOV.L       R7,@R1       ; j
          MOV.L       L12+16,R6    ; _k
          EXTS.W      R0,R2
          MOV.L       R2,@R6       ; k
          LDS.L       @R15+,PR
          MOV.L       @R15+,R14
          RTS
          MOV.L       @R15+,R13
L12:
          .DATA.L     _y
          .DATA.L     _x
          .DATA.L     _i
          .DATA.L     _j
          .DATA.L     _k
```

アセンブリ展開コード(rtnext指定)

```
_f:
          STS.L       MACL,@-R15
          MULS.W      R4,R5
          STS         MACL,R2
          EXTS.W      R2,R0
          RTS
          LDS.L       @R15+,MACL
_g:
          MOV.L       R13,@-R15
          MOV.L       R14,@-R15
          STS.L       PR,@-R15
          MOV.L       L12,R13      ; _y
          MOV.L       L12+4,R14    ; _x
          MOV.W       @R13,R5      ; y
          BSR         _f
          MOV.W       @R14,R4      ; x
          MOV.L       L12+8,R1     ; _i
          MOV.W       @R13,R5      ; y
          MOV.W       @R14,R4      ; x
          BSR         _f
          MOV.L       R0,@R1       ; i
          MOV.L       L12+12,R2    ; _j
          MOV.W       @R13,R5      ; y
          MOV.W       @R14,R4      ; x
          BSR         _f
          MOV.L       R0,@R2       ; j
          MOV.L       L12+16,R7    ; _k
          MOV.L       R0,@R7       ; k
          LDS.L       @R15+,PR
          MOV.L       @R15+,R14
          RTS
          MOV.L       @R15+,R13
L12:
          .DATA.L     _y
          .DATA.L     _x
          .DATA.L     _i
          .DATA.L     _j
          .DATA.L     _k
```

1.2.8 列挙型サイズ

enum 宣言した列挙型のデータを、列挙値が収まる最小型で扱います。
"auto_enum" オプション未指定時は、列挙型サイズをint 型として処理します。"auto_enum" オプションを指定した場合は、列挙子の取り得る値により扱う型が変わります。表 1-5に、列挙子の取り得る値と型について示します。

表 1-5 列挙型の取り得る値と型の関係

列挙子		型
最小値	最大値	
-128	127	signed char
0	255	unsigned char
-32768	32767	signed short
0	65535	unsigned short
上記以外		int

データサイズを小さくできるため、サイズは良くなります。enum 型の変数、構造体メンバが多くある場合に使用すると特にサイズに効果的です。ただし、"auto_enum" 指定なしと比べて拡張処理が増えることがあります。そのため、速度は不利になる可能性がありますので、注意してください。

【書式】

```
AUto_enum
```

[ルネサス統合開発環境でのオプション設定方法]

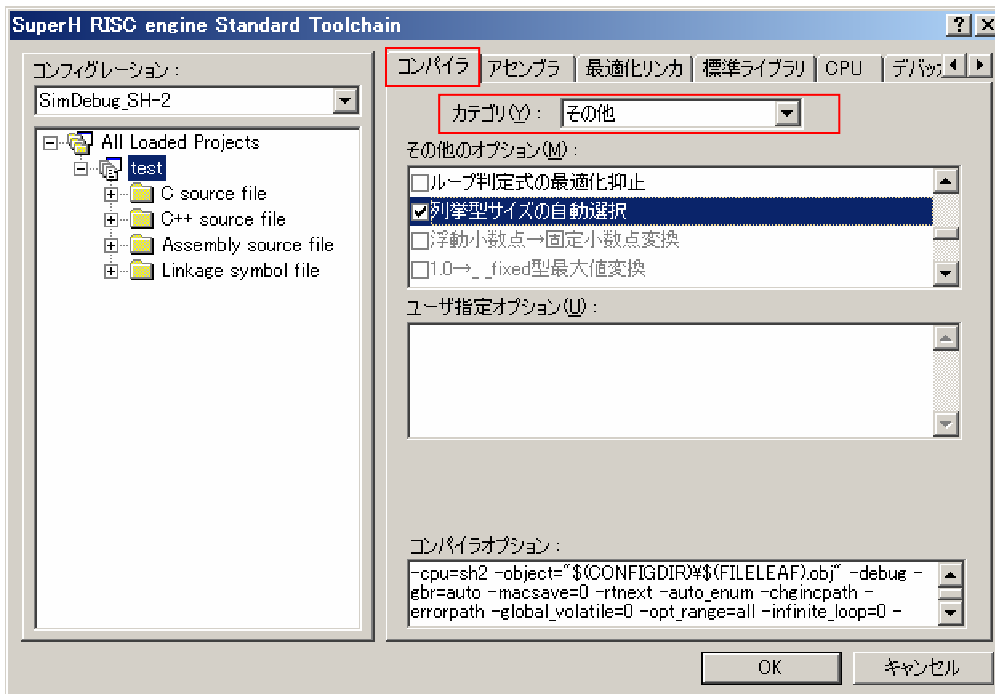

図 1-21

【例】

ソースコード

```
enum En {A_000 =0,A_001,A_002,A_003,A_END=255};
enum En x[3] = {A_000, A_001, A_END};
```

アセンブリ展開コード(auto_enum未指定)

```
_x:                                 ; static: x
         .DATA.L     H'00000000,H'00000001,H'000000FF
```

アセンブリ展開コード(auto_enum指定)

```
_x:                                 ; static: x
         .DATA.B     H'00,H'01,H'FF
```

1.2.9 switch 文展開方式

switch 文の判定処理を、case 値との比較を行う“if-then 方式”とするか、各 case 値の相対値から作成したデータテーブルを参照する“Table 方式”とするかを選択することができます。case の数が少ない場合や case 値の最大と最小の差が大きい場合は“case” オプションに関わらず、“if-then 方式”となる場合があります。

“case” オプションを指定しない場合は、いずれかの展開方式をコンパイラが自動的に選択します。

1. case の数が少ない場合や case 値の最大と最小の差が大きい場合は“if-then 方式”となります。
2. 1 以外で、“case” オプションが指定されている場合は、“case”オプションの指定に従います。
3. 1、2 以外で、“speed” オプションが指定されている場合、10 個程度以上の case ラベルがある時は“Table 方式”となります。

プログラムの実行時に、特定の case 値となる事が多い場合には、該当ケースを先頭に記述し“if-then 方式” を指定すると実行速度に有利となる傾向にあります。特定の case 値に飛ばない場合は、“Table 方式”を指定すると実行速度に有利となる傾向にあります。

詳細は、

「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:
＜コンパイラ活用ガイド＞効率の良いプログラミング手法 編 5 分岐」を参照してください。

【書式】

CAse = { Ifthen | Table }

[ルネサス統合開発環境でのオプション設定方法]

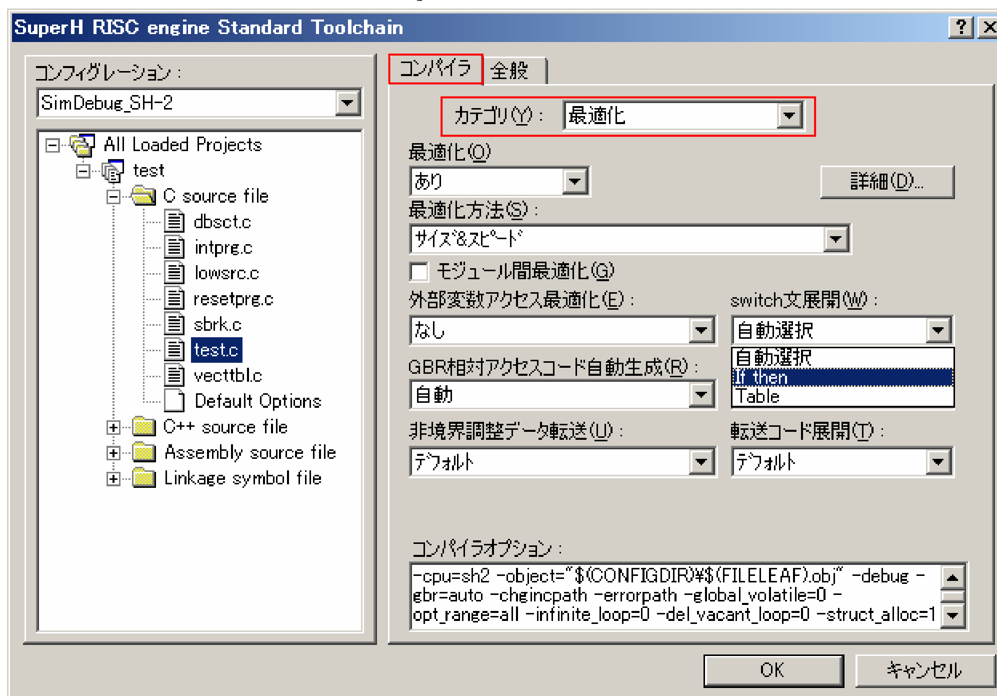

図 1-22

2. 便利なオプション

本章は、性能向上以外で便利なオプションについて説明します。

表 2-1 便利なオプション一覧

No	機能	オプション	参照
1	デバッグ情報出力モード	optimize	2.1
2	プリプロセッサ展開	preprocessor/noline	2.2
3	外部変数のvolatile化	global_volatile	2.3
4	空ループ削除	del_vacant_loop	2.4
5	無限ループ前の式削除	infinite_loop	2.5
6	ビットフィールド並び順指定	bit_order	2.6
7	構造体、共用体、クラスのアラインメント数指定	pack	2.7

2.1 デバッグ情報出力モード

“optimize=debug_only” オプションを指定すると、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なります。また、文単位の削除に関する最適化も完全に抑止されるため、C ソースコードの各文に break point を設定できるようになります。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。デバッグ時に一時的に使用することを推奨します。

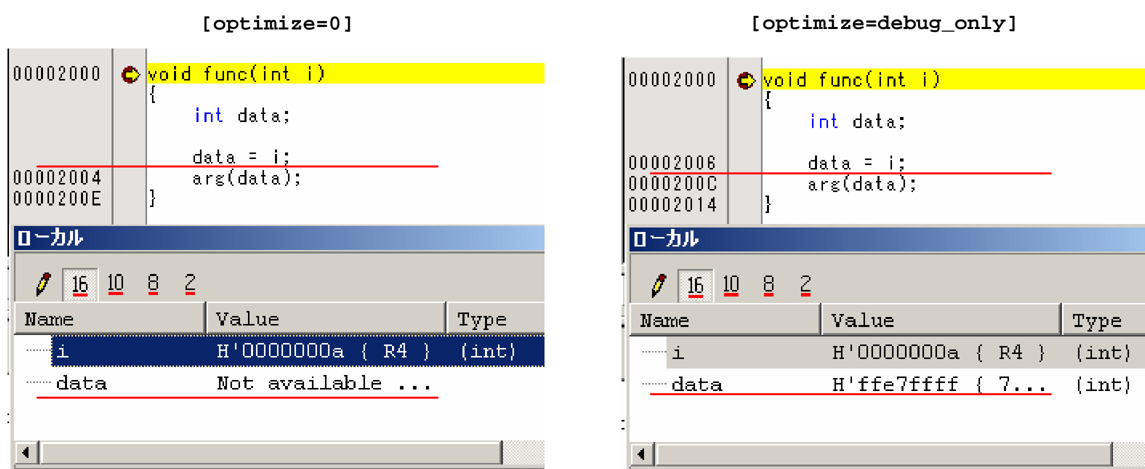

図 2-1

【書式】

OPtimize = { 0 | <u>1</u> | Debug_only }

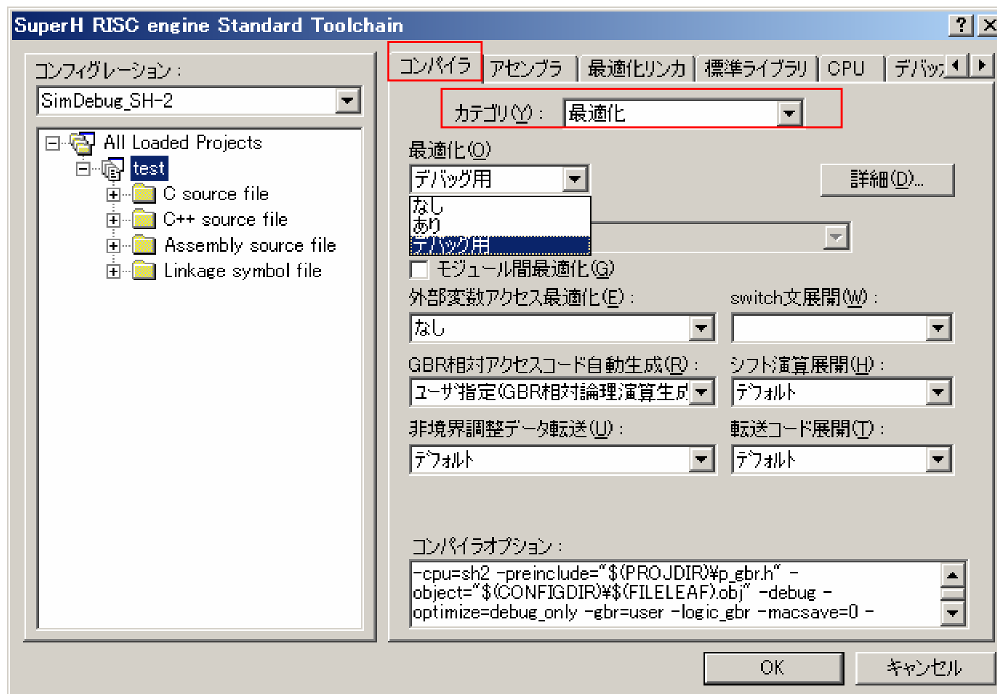

図 2-2

2.2 プリプロセッサ展開

プリプロセッサ展開後のソースプログラムを出力します。プリプロセッサ展開後のコードとは、#include や #define が置換された後のコードのことです。本ファイルはヘッダファイルの情報などが展開済みであるため、単独でコンパイル可能なファイルとなります。

<ファイル名>を指定しない場合は、ソースファイル名と同じファイル名で拡張子が「p」(入力ソースファイルが C プログラムの場合)、または「pp」(入力ソースプログラムが C++プログラムの場合)のファイルが作成されます。

本オプションを指定した場合は、オブジェクトプログラムを出力しません。

“noline” を指定した場合、プリプロセッサ展開時に #line の出力を抑止します。

【書式】

```
PREProcessor [= <ファイル名>]
NOLINe
```

[ルネサス統合開発環境でのオプション設定方法]

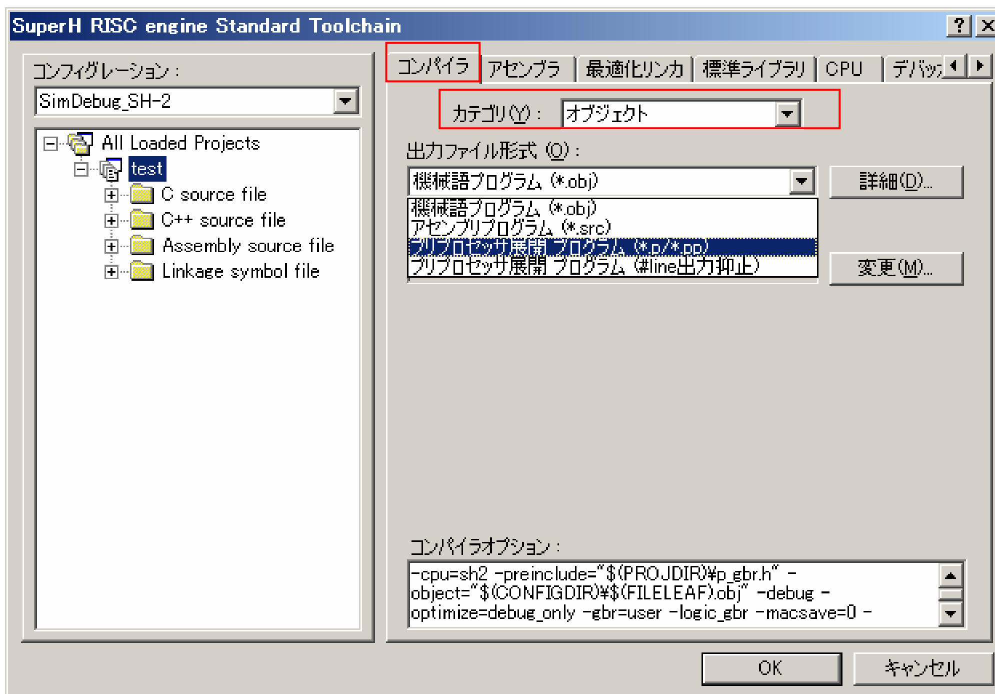

図 2-3

【例】

```
ソースコード
#define NUM 1
#define MESSAGE(num, name) {num, __DATE__, #name}

struct {
    int   num   ;
    char* date  ;
    char* string;
} data[] = {
        MESSAGE(NUM, aaaa),
};
プリプロセッサ展開
#line 1 "test.c"

struct {
    int   num   ;
    char* date  ;
    char* string;
} data[] = {
        {1, "Jun 13 2007", "aaaa"},
};
```

2.3 外部変数の volatile 化

コンパイラの最適化処理では、静的に C ソースコードを解析して意味が変わらない範囲で、変数のアクセス順序や回数などを最適化する可能性があります。しかし、I/O レジスタへのアクセス、割り込み処理で使用する変数などは、このような最適化が行われると意図した動作にならない場合があります。この場合、変数に対して volatile 宣言を行う必要があります。volatile 宣言を行うと、アクセス幅、アクセス順序、アクセス回数が C ソースコードの記述通りに実施されます。

volatile 宣言の指定は、必要となる変数を見定めて指定するのが望ましいですが、過去の資産を流用している時など、個々の変数をチェックするのが難しい場合があります。このような場合は、"global_volatile=1" を試してみてください。"global_volatile=1" を指定すると、すべての外部変数を volatile 宣言したものとして扱うことができます。

【書式】

```
GLOBAL_Volatile = { 0 | 1 }
```

[ルネサス統合開発環境でのオプション設定方法]

"コンパイラ"タブを選択 → [カテゴリ]に"最適化"を選択 → [詳細]を選択(図 1-2)すると表示されるダイアログで次のように設定します。

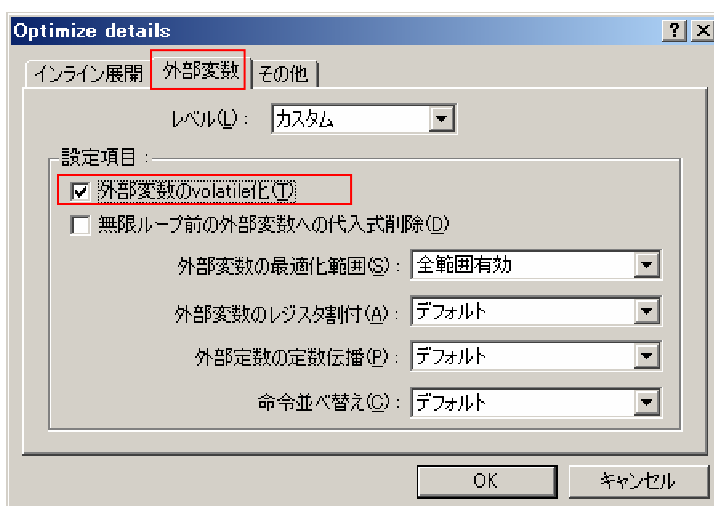

図 2-4

【例】

ソースコード
```
int   var;
void func(void)
{
    var = 1;
    var = 0;
}
```

最適化イメージ (global_volatile=0指定)
```
int   var;
void func(void)
{
    var = 0;
}
```

アセンブリ展開コード(global_volatile=0指定)
```
_func:
          MOV.L       L11,R6      ; _var
          MOV         #0,R2       ; H'00000000
          RTS
          MOV.L       R2,@R6      ; var
L11:
          .DATA.L     _var
```

最適化イメージ (global_volatile=1指定)
```
int   var;
void func(void)
{
    var = 1;
    var = 0;
}
```

アセンブリ展開コード(global_volatile=1指定)
```
_func:
          MOV.L       L11,R6      ; _var
          MOV         #1,R1       ; H'00000001
          MOV         #0,R4       ; H'00000000
          MOV.L       R1,@R6      ; var
          RTS
          MOV.L       R4,@R6      ; var
L11:
          .DATA.L     _var
```

2.4 空ループ削除

ループ内に処理がない空ループを削除するかを選択できます。

“del_vacant_loop=0” を指定した場合、ループ内に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、ループ内に処理がないループは削除します。

本オプションのデフォルトは、“del_vacant_loop=0” です。

“del_vacant_loop=1” を指定すると、タイミングをとるような目的で入れた必要な空ループも削除されますので、注意してください。

【書式】

```
DEL_vacant_loop = { 0 | 1 }
```

[ルネサス統合開発環境でのオプション設定方法]

“コンパイラ”タブを選択 → [カテゴリ]に“最適化”を選択 → [詳細]を選択(図 1-2)すると表示されるダイアログで次のように設定します。

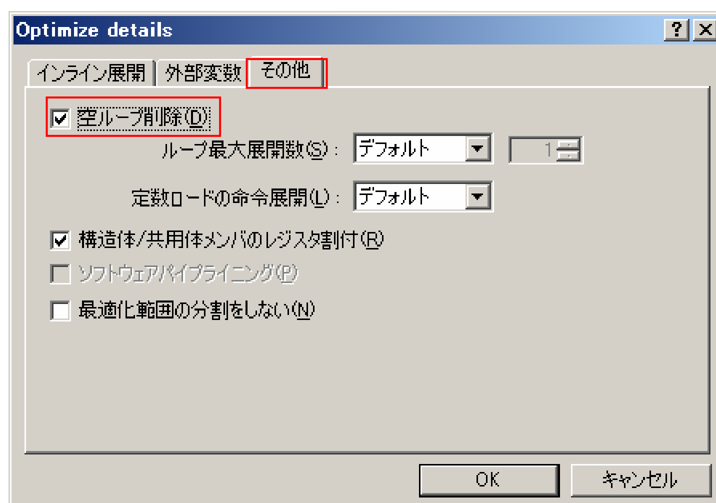

図 2-5

2.5 無限ループ前の式削除

無限ループ前の式で非 volatile の外部変数への代入があり、その外部変数がループ内で参照されないのであれば削除することができます。

"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。

本オプションデフォルトは、"infinite_loop =0" です。

【書式】

INFinite_loop = { 0 | 1 }

[ルネサス統合開発環境でのオプション設定方法]

"コンパイラ"タブを選択 → [カテゴリ]に"最適化"を選択 → [詳細]を選択(図 1-2)すると表示されるダイアログで次のように設定します。

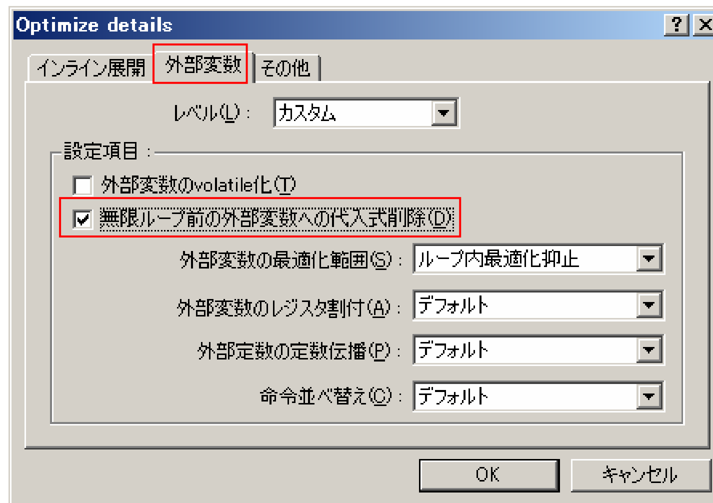

図 2-6

【例】

ソースコード

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

最適化イメージ (infinite_loop=0指定)

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

最適化イメージ (infinite_loop=1指定)

```
int a;
void f(void)
{
    while(1) {
    }
}
```

アセンブリ展開コード(infinite_loop=0指定)

```
_f:
          MOV.L     L13+2,R6    ; _a
          MOV       #1,R2       ; H'00000001
          MOV.L     R2,@R6      ; a
L11:
          BRA       L11
          NOP
L13:
          .RES.W    1
          .DATA.L   _a
```

アセンブリ展開コード(infinite_loop=1指定)

```
_f:
L10:
          BRA       L10
          NOP
```

2.6 ビットフィールド並び順指定

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ります。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。
"bit_order=left" を指定した場合は上位ビットからメンバを割り付けます。
"bit_order=right" を指定した場合は下位ビットからメンバを割り付けます。

#pragma bit_order の指定でも、ビットフィールドの並び順を指定することができます。オプション、#pragma 同時に指定された場合は、#pragma の指定を優先します。
機能の詳細は、
「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:
　＜コンパイラ活用ガイド＞拡張機能編 2.2 ビットフィールドの並び順指定」を参照してください。

【書式】

BIt_order = { Left | Right }

[ルネサス統合開発環境でのオプション設定方法]

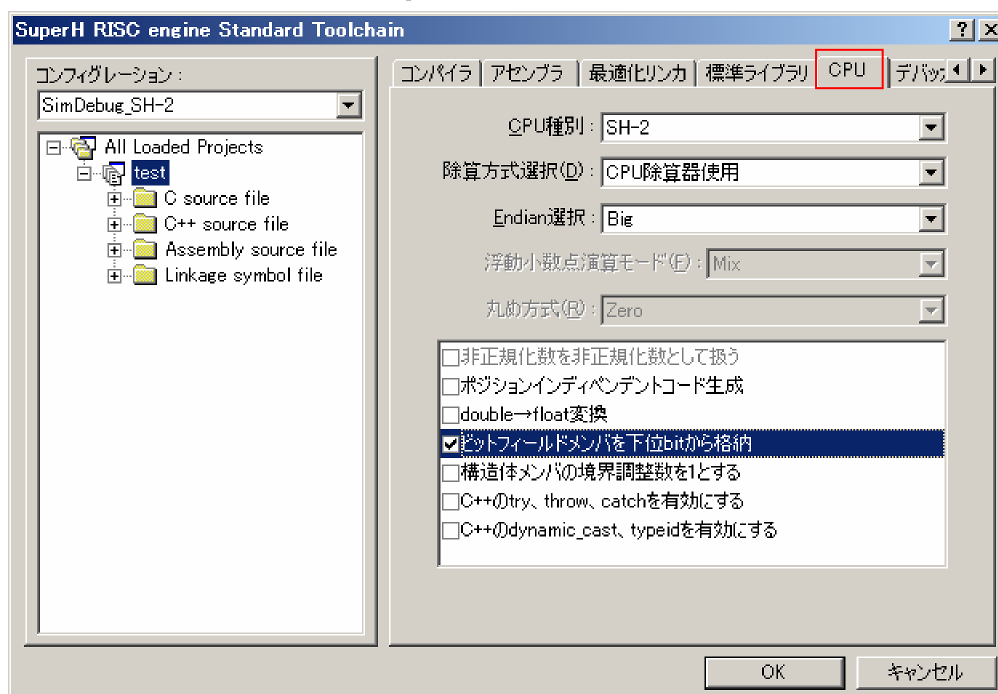

図 2-7

2.7 構造体、共用体、クラスのアライメント数指定

通信のプログラムで使用する構造体など、構造体に空き領域ビットを作りたくない場合があります(共用体・クラスも同様です)。このような場合は、オプションで “pack=1” を指定することで、構造体メンバのアライメント数を 1 とすることができます。アライメント数が 1 となった構造体は空き領域が作られなくなります。

#pragma pack の指定でも、構造体のアライメント数を指定することができます。オプションと、#pragma が同時に指定された場合は、#pragma の指定を優先します。
機能の詳細は、
「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：
<コンパイラ活用ガイド>拡張機能編 2.3 構造体、共用体、クラスのアライメント数指定」を参照してください。

【書式】

```
PACK = { 1 | 4 }
```

[ルネサス統合開発環境でのオプション設定方法]

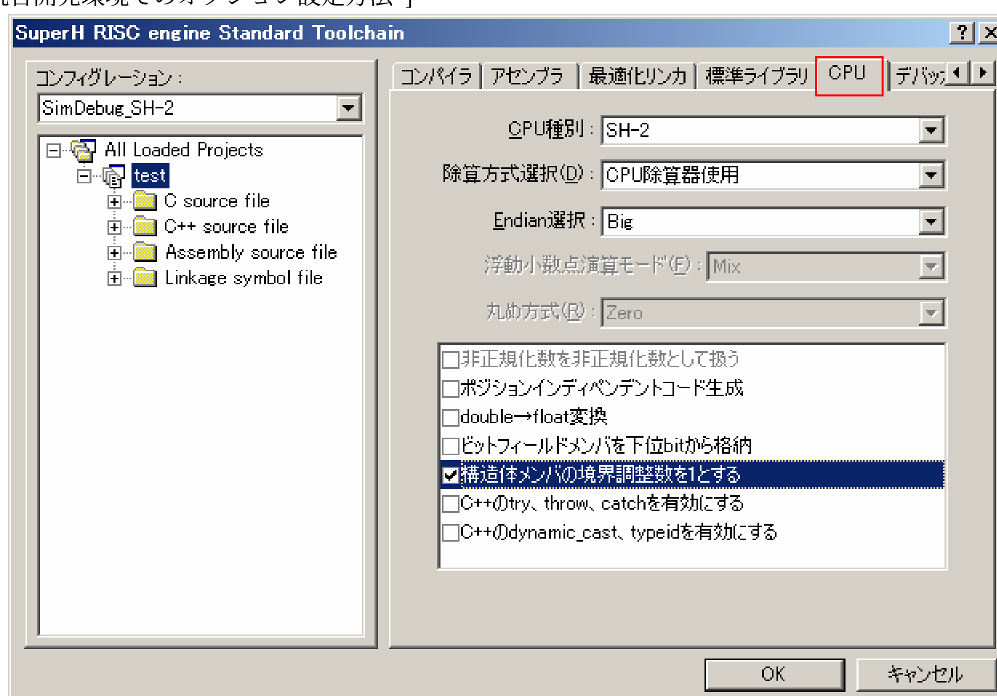

図 2-8

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	2007.9.1	—	初版発行

安全設計に関するお願い

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

本資料ご利用に際しての留意事項

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

© 2007. Renesas Technology Corp., All rights reserved.