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。

RX ファミリ用 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、RX ファミリ用 C/C++ コンパイラ V.1 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

コンパイラの最適化オプションには、スピード優先、サイズ優先の 2 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、“output=src” および “cpu=rx600” 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)

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 (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

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　　：デフォルト

[　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]

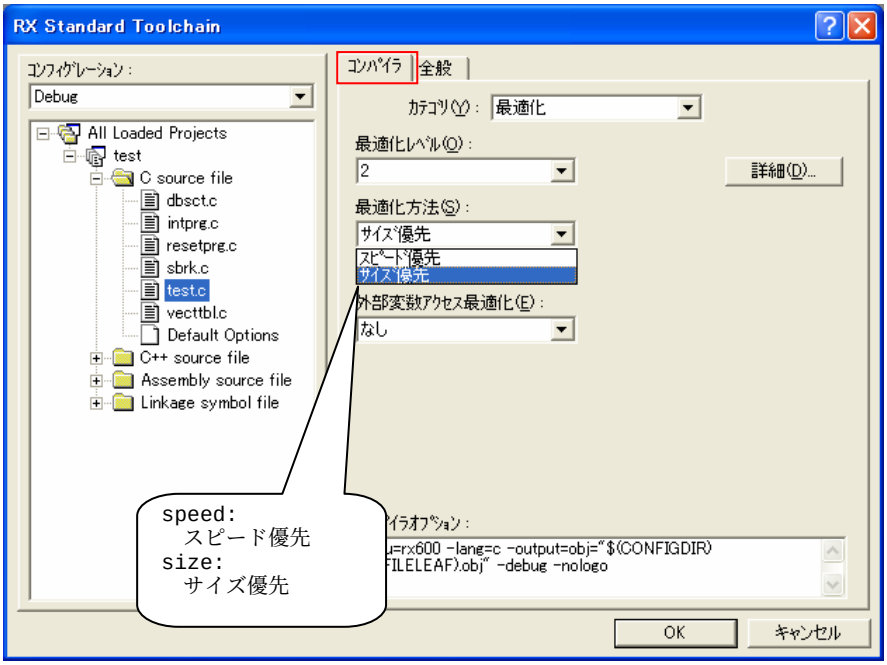


図 1-1

【補足 1】

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。

【補足 2】

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	size			speed			参照
	optimize	max	2	0 または 1	max	2	0 または 1	
1	**自動インライン展開**	inline=0	noinline	noinline	inline=500	inline=100	noinline	0
2	**ループ展開最適化**	loop=1	loop=1	loop=1	loop=32	loop=2	loop=1	1.1.2

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。”inline=0”を指定した場合では、インライン展開を行います。

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、

「RX ファミリ用 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 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は 100**
noinline **: 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]

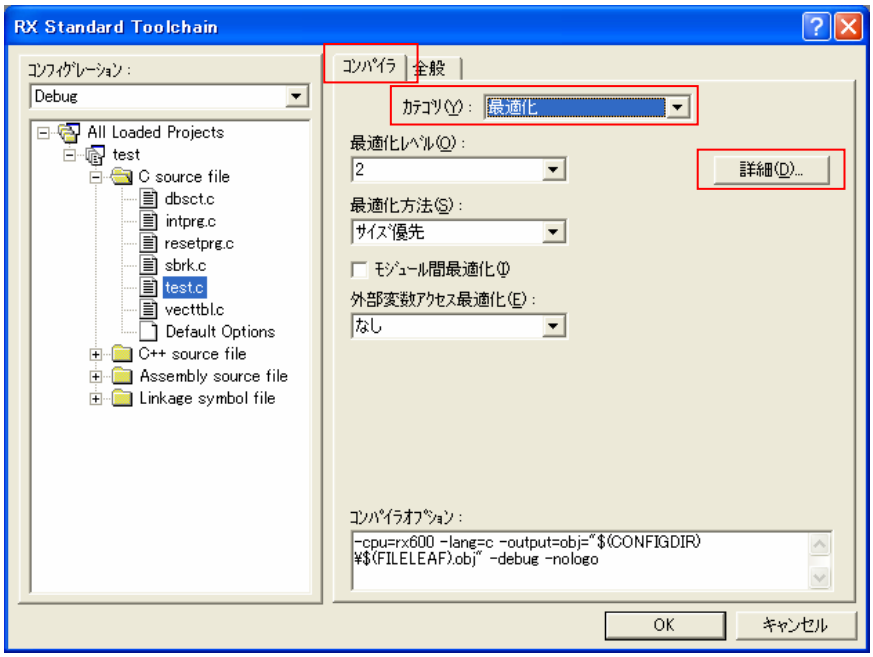


図 1-2

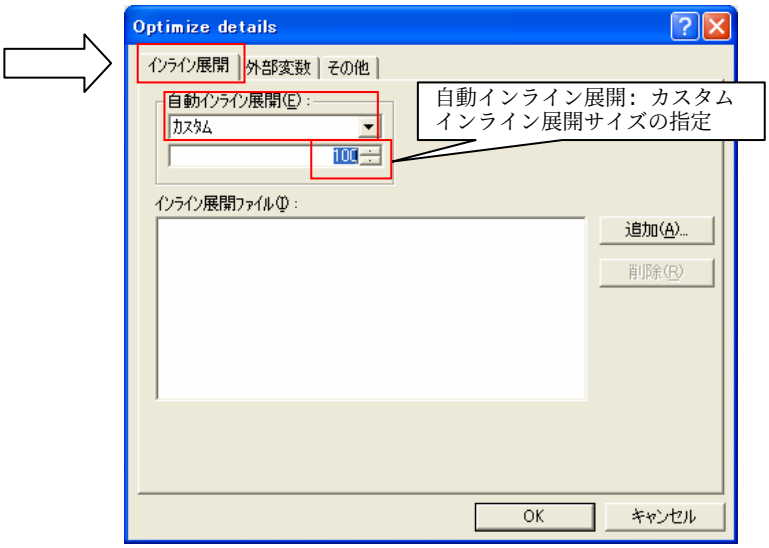


図 1-3

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、コンパイル対象のファイルから可視な関数定義がある関数しか対象に出来ません。別ファイルに定義が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。

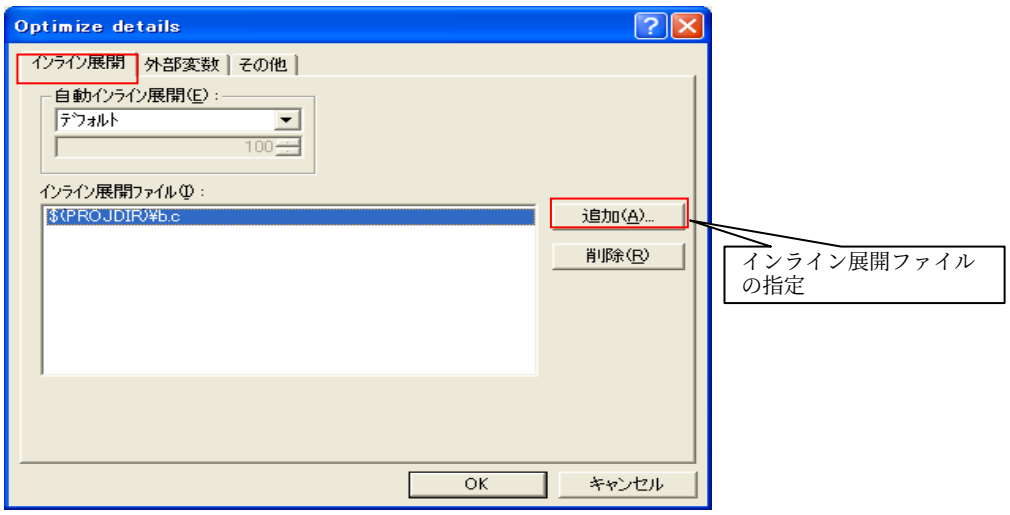


図 1-4

【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]

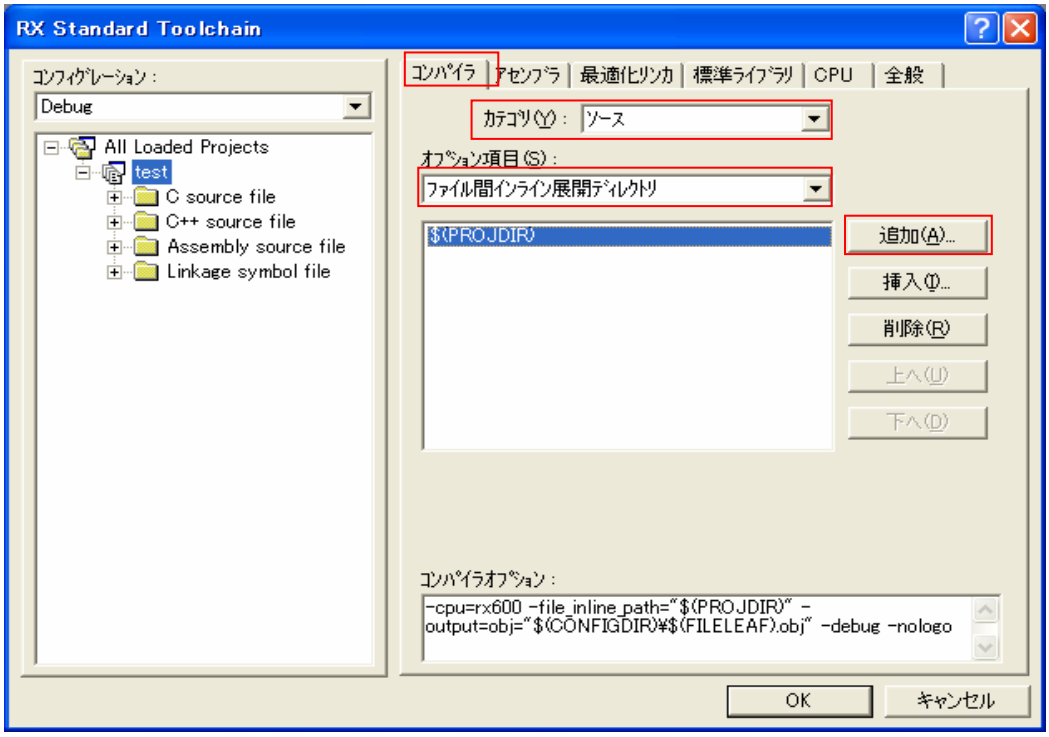


図 1-5

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。
ループ展開時の最大展開数を “loop=<数値(1～32)>” で指定することができます。

【書式】

loop[=<数値>] **: speed 指定時のデフォルトは 2**
: size 指定時のデフォルトは 1

[ルネサス統合開発環境でのオプション設定方法]
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。

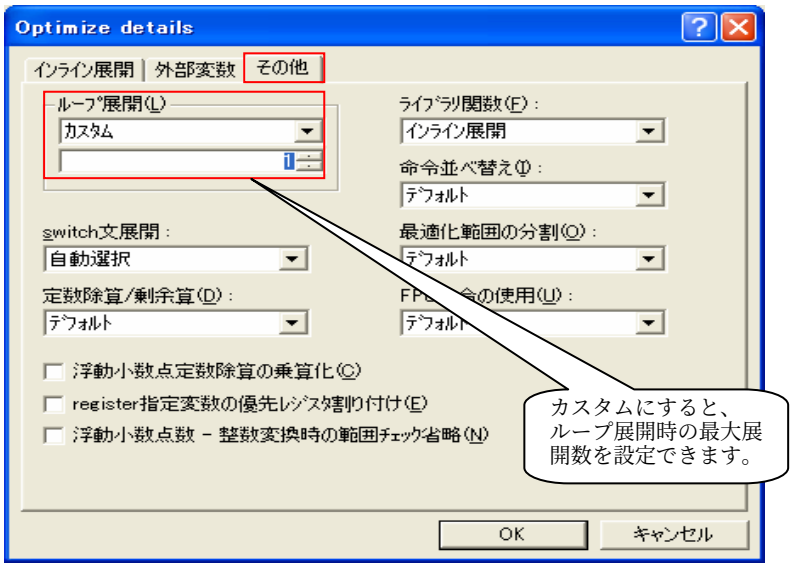


図 1-6

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	外部変数アクセス最適化	map/smap	◎	◎	1.2.1
2	最適化範囲分割	scope/noscope	△	△	1.2.2
3	列挙型サイズ	auto_enum	○	×	1.2.3
4	switch 文展開方式	case	△	△	1.2.4

◎ 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

1.2.1 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、アドレス値のロード処理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。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。（High-performance Embedded Workshop 環境では、自動実施）

“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、ビルド時間が長くなります。“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 1-3 map と smap の特徴

オプション	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	長い	必要	smap より高い	必要
smap	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>]

  output オプションにより使い方が異なります。

  [output=abs または mot の場合]
  “map” オプションのみ指定してください。

  [output=obj または src の場合]
  “map” オプションを指定しないで1回コンパイルし、リンク時に最適化リンカのオプション “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。

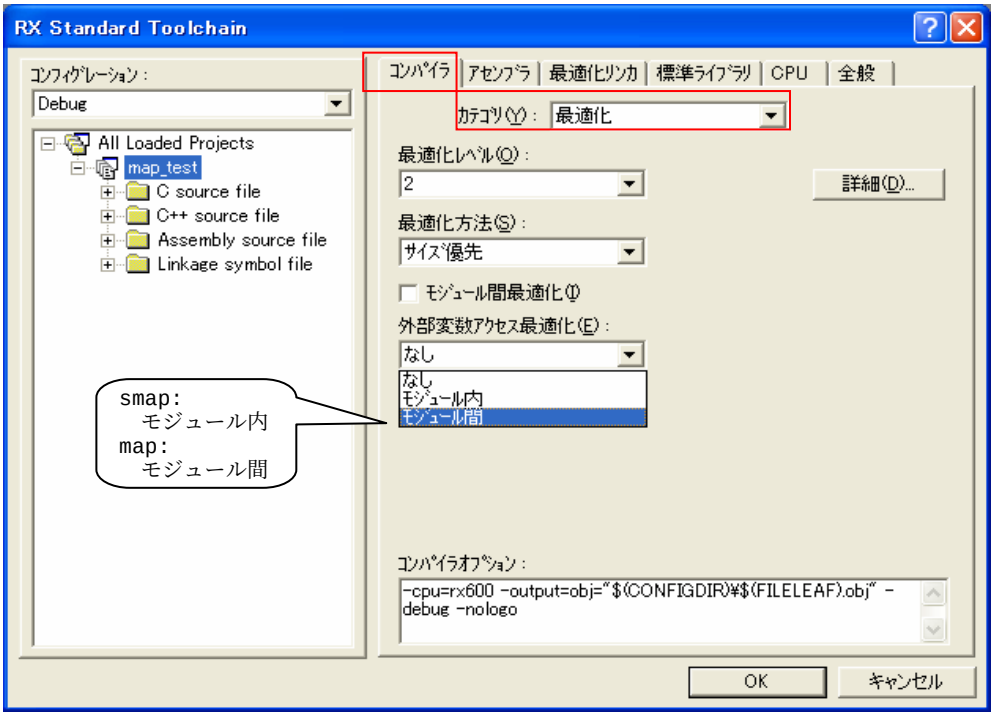


図 1-7

【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):コードサイズ19バイト

```
_f:                                     ; function: f
                .STACK      _f=4
L10:
                MOV.L       #00000000H,R5
                MOV.L       #_a,R4
                MOV.L       R5,[R4]
                MOV.L       #_b,R4
                MOV.L       R5,[R4]
                RTS
                .SECTION    B,DATA,ALIGN=4
                .glb        _a
_a:                                     ; static: a
                .blkl       1
                .glb        _b
_b:                                     ; static: b
                .blkl       1
```

アセンブリ展開コード(map/smap指定)コードサイズ18バイト

```
_f:                                     ; function: f
                .STACK      _f=4
L10:
                MOV.L       #00000000H,R5
                MOV.L       #_a,R4
                MOV.L       R5,[R4]
                MOV.L       R5,04H[R4]
                RTS
                .SECTION    B,DATA,ALIGN=4
                .glb        _a
_a:                                     ; static: a
                .blkl       1
                .glb        _b
_b:                                     ; static: b
                .blkl       1
```

1.2.2 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 : optimize=0|1|2 指定時のデフォルト
noscope : optimize=max 指定時のデフォルト

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。

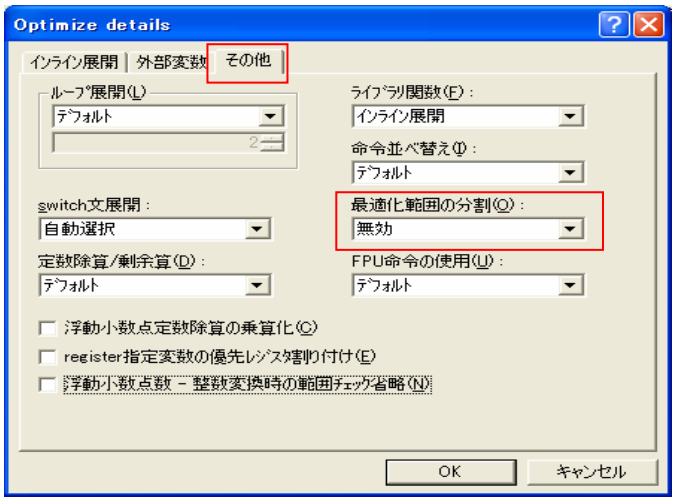


図 1-8

1.2.3 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signed 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 1-4に、列挙子の取り得る値と型について示します。

表 1-4 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		signed 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。

【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]

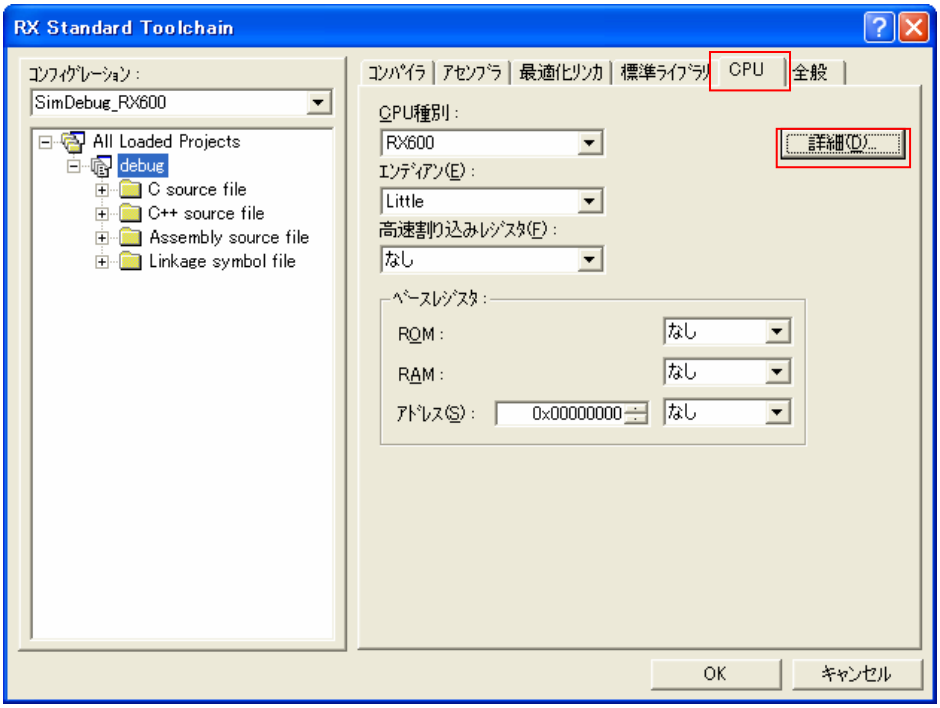


図 1-9

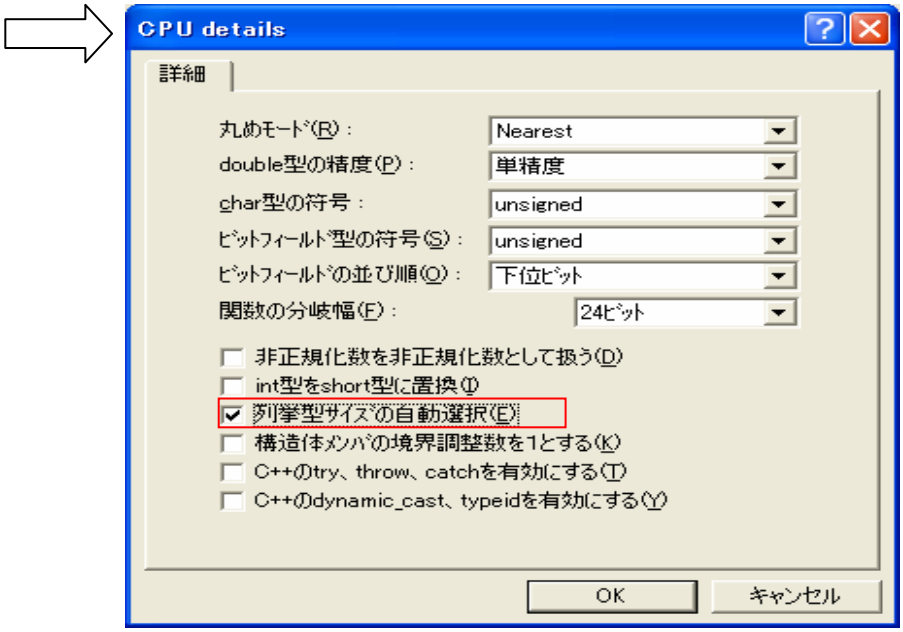


図 1-10

【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 .lword 00000000H, 00000001H, 000000FFH	 アセンブリ展開コード(auto enum指定) _x: ; static: x .byte 00H, 01H,0FFH

1.2.4 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” オプションで Table オプションを選択していても、“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。

【書式】

case = { ifthen | table | auto}

[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。

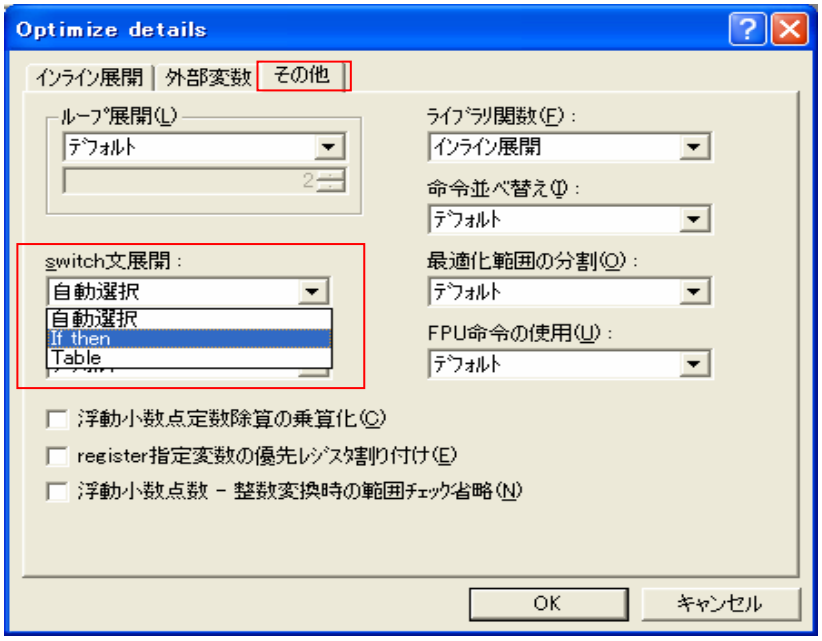


図 1-11

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。

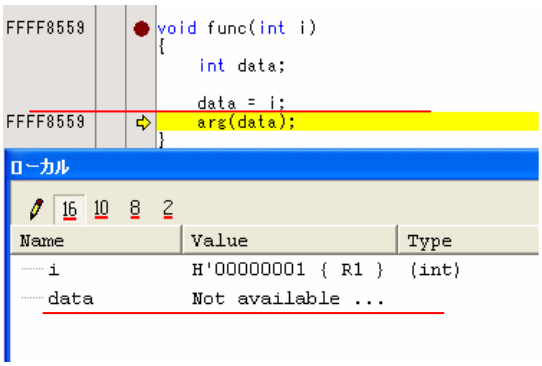
表 2-1 便利なオプション一覧

No	機能	オプション	参照
1	**高精度なデバッグ情報の生成**	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**	volatile	2.3
4	**ビットフィールド並び順指定**	bit_order	2.4
5	**構造体、共用体、クラスのアライメント数指定**	pack	2.5

2.1 高精度なデバッグ情報の生成

“optimize=0” 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 、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。

[optimize=2]



[optimize=0]

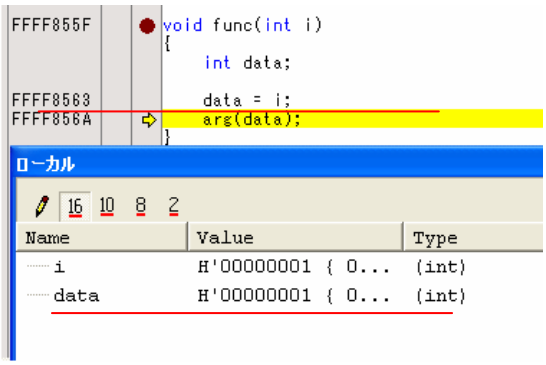


図 2-1

【書式】

optimize = { 0 | 1 | 2 | max }

[ルネサス統合開発環境でのオプション設定方法]

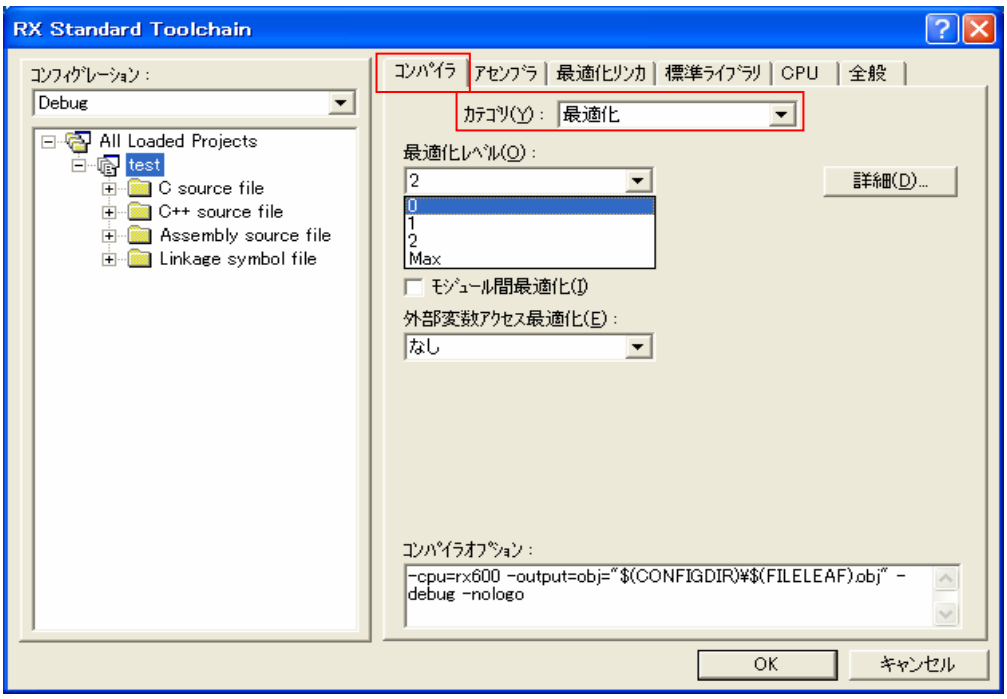


図 2-2

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 の内容を展開し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。

“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】

```
output=prep [= <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]

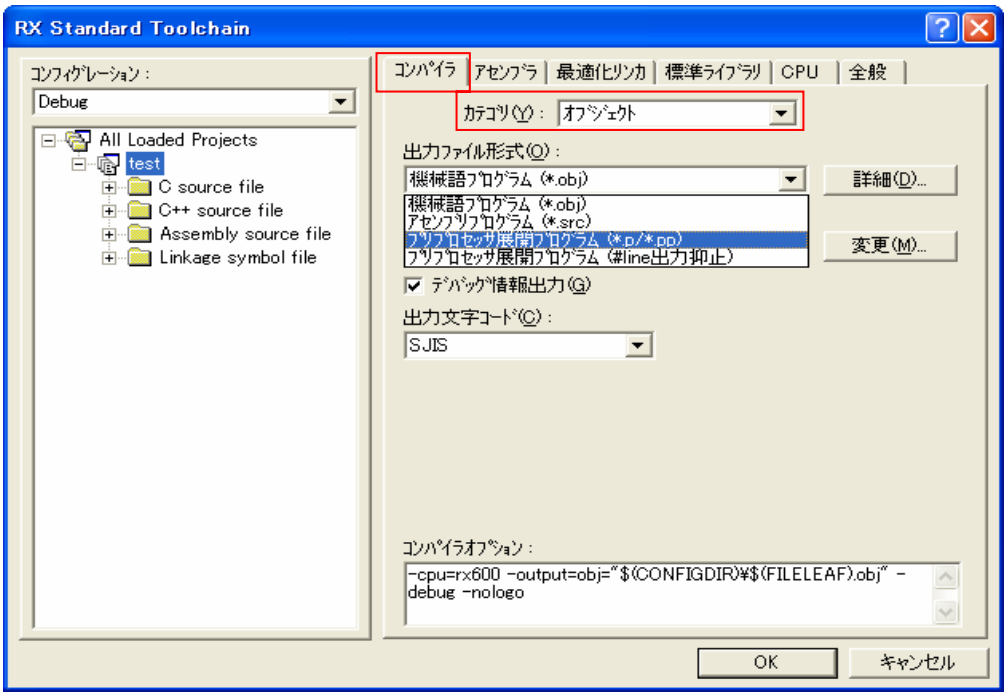


図 2-3

【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 "C:\\Workspace_Evaluation_RX\\test1\\test1\\a.c"



struct {
    int   num   ;
    char* date  ;
    char* string;
} data[] = {
        {1, "Jul 22 2009", "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ことがあります。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 修飾を行う必要があります。volatile 修飾を行うと、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 修飾の指定は、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、“volatile” を試してみてください。 “volatile” を指定すると、すべての外部変数を volatile 修飾したものとして扱うことができます。

【書式】

```
volatile
novolatile
```

[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。

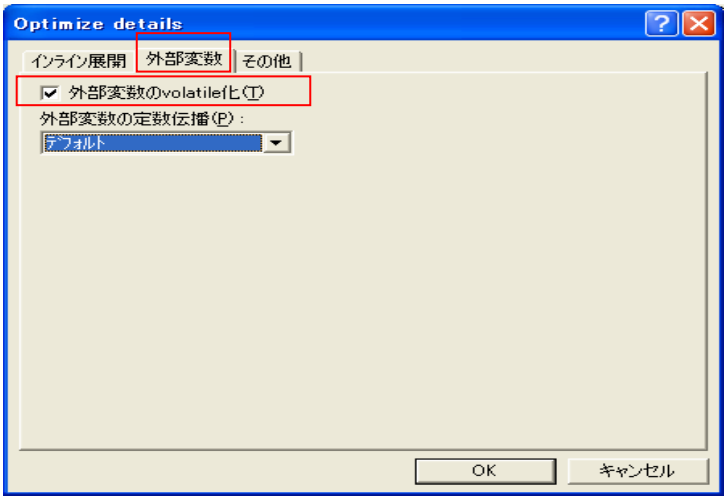


図 2-4

【例】

ソースコード

```
int   var;
void func(void)
{
    var = 1;
    var = 0;
}
```

最適化イメージ（novolatile指定）

```
int   var;
void func(void)
{
    var = 0;
}
```

アセンブリ展開コード(novolatile指定)

```
_func:                              ; function: func
            .STACK      _func=4
L10:
            MOV.L       #_var,R4
            MOV.L       #00000000H,[R4]
            RTS
            .SECTION    B,DATA,ALIGN=4
            .glb        _var
_var:                               ; static: var
            .blkl       1
```

最適化イメージ（volatile指定）

```
int   var;
void func(void)
{
    var = 1;
    var = 0;
}
```

アセンブリ展開コード(volatile指定)

```
_func:                              ; function: func
            .STACK      _func=4
L10:
            MOV.L       #_var,R4
            MOV.L       #00000001H,[R4]
            MOV.L       #00000000H,[R4]
            RTS
            .SECTION    B,DATA,ALIGN=4
            .glb        _var
_var:                               ; static: var
            .blkl       1
```

2.4 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、
「RX ファミリ用 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]

“CPU”タブを選択 → [詳細]を選択(図 1-9)すると表示されるダイアログで次のように設定します。

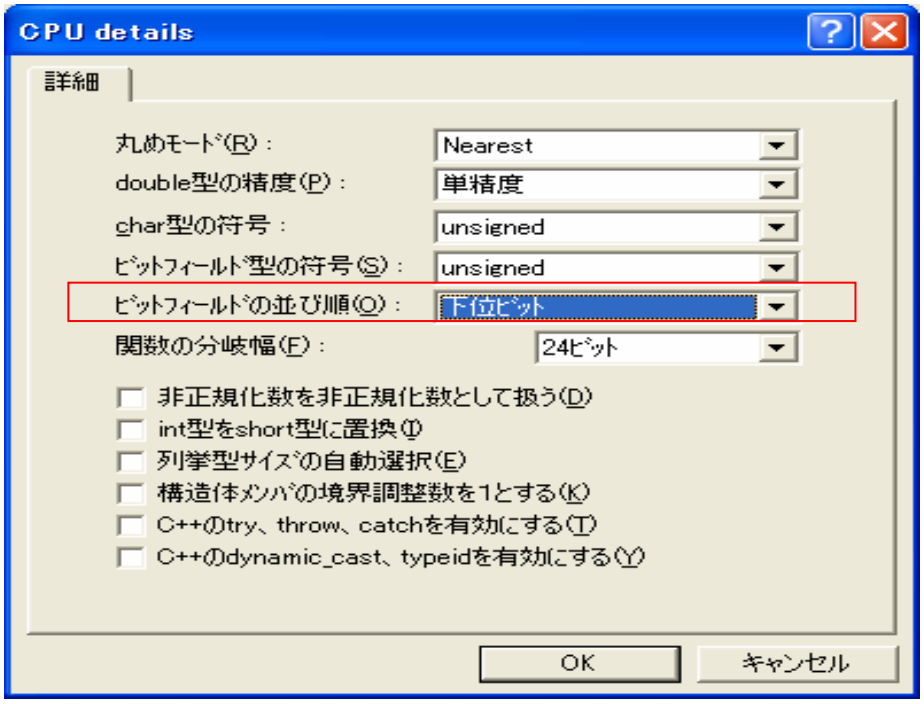


図 2-5

2.5 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” 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 1 にできます。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、

「RX ファミリ 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
unpack

[ルネサス統合開発環境でのオプション設定方法]

“CPU”タブを選択 → [詳細]を選択(図 1-9)すると表示されるダイアログで次のように設定します。

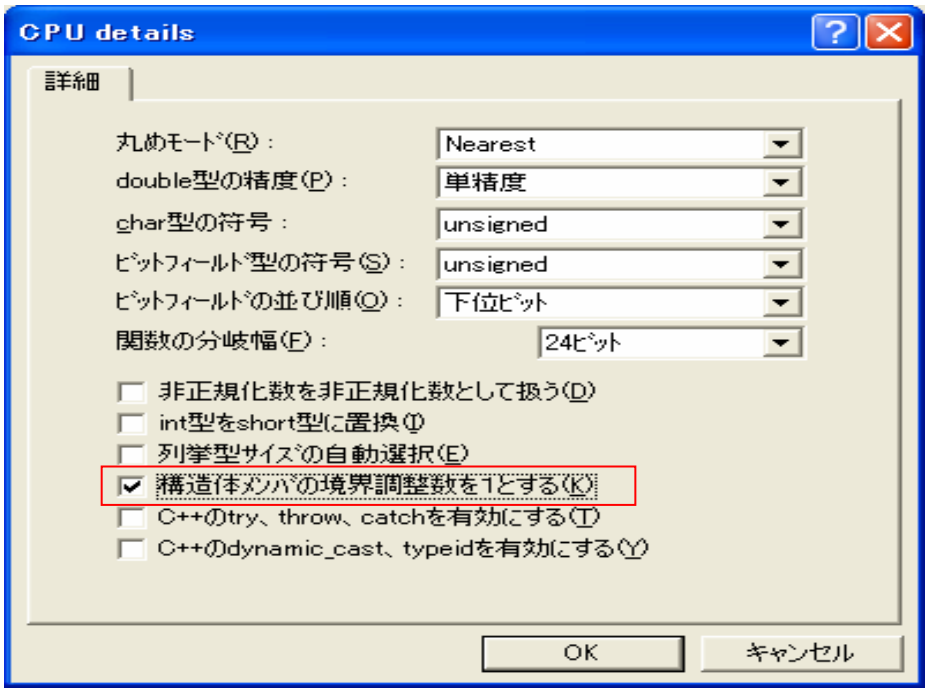


図 2-6

ホームページとサポート窓口<website and support>

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	2009.10.1	—	初版発行