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＞拡張機能編

本ドキュメントでは、SuperH RISC engine C/C++コンパイラ V.9 において、サイズもしくは実行速度の性能向上が期待できる拡張機能(#pragma) と便利な#pragma について説明します。

目次

1. サイズ効率・実行速度の性能向上が期待できる拡張機能

本章では、サイズもしくは実行速度の性能向上が期待できる拡張機能(#pragma)について説明します。
サイズもしくは実行速度の性能向上が期待できる拡張機能を表 1-1に示します。

表 1-1 性能向上が期待できる拡張機能

No	#pragma	機能	サイズ	実行速度	参照
1	#pragma abs16 #pragma abs20 #pragma abs28 #pragma abs32	**アドレス領域の指定**	◎	△	1.1
2	#pragma inline	**関数のインライン展開指定**	×	◎	1.2
3	#pragma inline_asm	**アセンブリ記述関数のインライン展開**	×	○	1.3
4	#pragma regsave #pragma noregsave #pragma noregalloc	**レジスタの退避／回復コード出力の制御**	○	○	1.4
5	#pragma global_register	**大域変数のレジスタ割り付け**	△	△	1.5
6	#pragma gbr_base #pragma gbr_base1	**GBRベース変数の指定**	◎	○	1.6

◎ **特に有効である**
○ **有効である**
△ **有効であるが注意して使用する必要がある**
× **劣化する**

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。
ただし、“cpu” オプションにより、アセンブリ言語展開コードが SH-1、SH-2、SH-2E、SH-3、および SH-4 で異なる場合があります。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。

1.1 アドレス領域の指定

アドレス領域の指定(#pragma 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宣言です。デフォルトでは 32 ビットです。
例えば、"#pragma abs16"の指定が行われた変数や関数は、アドレスが 16 ビットで表現できるアドレスに配置されることを意味します。これにより、アドレス値のデータサイズがデフォルトの 4 バイトから 2 バイトとなり、プログラムサイズが削減されます。参照箇所の多い変数や関数に対してアドレス領域の指定を行うと、効果的にサイズ削減が行えます。 #pragma abs16/20/28/32 が指定された変数や関数は、全てのソースコードで同一の #pragma abs16/20/28/32 が指定されている必要があります。 共通でインクルードするヘッダファイルに#pragma abs16/20/28/32 を記述することをお勧めします。

"#pragma abs20"及び"#pragma abs28" は SH-2A、SH2A-FPU でのみ使用することが出来るアドレス指定です。詳しくは、「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＞SH-2A, SH2A-FPU 編 2.1 20 ビット長即値ロード」 を参照してください。

アドレス領域の指定はオプションでも指定できます。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。オプションで、16 ビットを指定された場合、#pragma abs32 を指定した識別子はデフォルトの 32 ビットにすることができます。

【書式】

#pragma abs16 （<識別子> [,<識別子>・・・] ）
#pragma abs20 （<識別子> [,<識別子>・・・] ）
#pragma abs28 （<識別子> [,<識別子>・・・] ）
#pragma abs32 （<識別子> [,<識別子>・・・] ）
識別子：変数名 | 関数名

【例】

abs16 指定の時、対象となる変数・関数のアドレス格納領域は、.DATA.L (4 バイト)から .DATA.W(2 バイト) となります。

#pragma abs16 指定のないソースコード

```
extern int x(void);
int y;
long z;
void f(void)
{
    z = x() + y;
}
```

アセンブリ展開コード

```
_f:
            STS.L       PR,@-R15
            MOV.L       L11+2,R2    ; _x
            JSR         @R2
            NOP
            MOV.L       L11+6,R5    ; _y
            MOV.L       L11+10,R4   ; _z
            MOV.L       @R5,R1      ; y
            ADD         R1,R0
            LDS.L       @R15+,PR
            RTS
            MOV.L       R0,@R4      ; z
L11:
            .RES.W      1
            .DATA.L     _x
            .DATA.L     _y
            .DATA.L     _z
```

#pragma abs16 指定ソースコード

```
#pragma abs16 (x,y,z)
extern int x(void);
int y;
long z;
void f(void)
{
    z  = x() + y;
}
```

アセンブリ展開コード

```
_f:
            STS.L       PR,@-R15
            MOV.W       L11,R2      ; _x
            JSR         @R2
            NOP
            MOV.W       L11+2,R5    ; _y
            MOV.W       L11+4,R4    ; _z
            MOV.L       @R5,R1      ; y
            ADD         R1,R0
            LDS.L       @R15+,PR
            RTS
            MOV.L       R0,@R4      ; z
L11:
            .DATA.W     _x
            .DATA.W     _y
            .DATA.W     _z
```

【注意事項】

- abs16/20/28/32 指定をした変数および関数はセクション切り替え機能で別セクションにし、リンク時に指定ビットで表現できるアドレスになるようにセクションを配置してください。指定ビットアドレスで表現されるアドレスに配置されていないとリンク時にエラーとなります。
- 指定ごとの配置可能なアドレス範囲を下表に示します。

表 1-2 アドレス範囲

#pragma	アドレス配置	
	上位	下位
#pragma abs16	0x00000000	0x00007FFF
	0xFFFF8000	0xFFFFFFFF
#pragma abs20	0x00000000	0x0007FFFF
	0xFFF80000	0xFFFFFFFF
#pragma abs28	0x00000000	0x07FFFF7F *
	0xF8000000	0xFFFFFFFF
#pragma abs32	0x00000000	0xFFFFFFFF

[**注釈**] * 0x07FFFF7F **となることに注意**

- abs16 を指定する場合は、下記図で示した領域にセクションを配置してください。

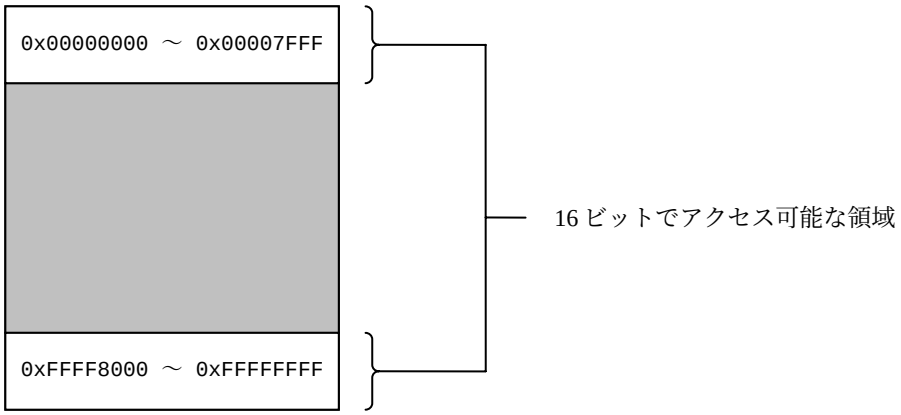


図 1-1

- コンパイル時にポジションインディペンデントコードの生成を指定 (オプション “pic=1” を指定) していると、相対アクセスによるアドレス参照が行われるため、関数アドレスは指定ビットで生成されません。

ソースコード

```
#pragma abs16 (x,y,z)
extern int x(void);
int y;
long z;
void f(void)
{
    z = x() + y;
}
```

アセンブリ展開コード(pic=0指定)

```
_f:
        STS.L       PR,@-R15
        MOV.W       L11,R2      ; _x
        JSR         @R2
        NOP
        MOV.W       L11+2,R5    ; _y
        MOV.W       L11+4,R4    ; _z
        MOV.L       @R5,R1      ; y
        ADD         R1,R0
        LDS.L       @R15+,PR
        RTS
        MOV.L       R0,@R4      ; z
L11:
        .DATA.W     _x
        .DATA.W     _y
        .DATA.W     _z
```

アセンブリ展開コード(pic=1指定)

```
_f:
        STS.L       PR,@-R15
        MOV.L       L12+6,R3    ; H'FFFFFFFC+_x-L11
L11
        BSRF        R3
        NOP
        MOV.W       L12,R5      ; _y
        MOV.W       L12+2,R2    ; _z
        MOV.L       @R5,R1      ; y
        ADD         R1,R0
        LDS.L       @R15+,PR
        RTS
        MOV.L       R0,@R2      ; z
L12:
        .DATA.W     _y
        .DATA.W     _z
        .RES.W      1
        .DATA.L     H'FFFFFFFC+_x-L11
```

1.2 関数のインライン展開指定

インライン展開とは、関数呼び出しの位置に呼び出し先のプログラムを展開する最適化処理で、関数呼び出しに伴うオーバーヘッドの削減により、速度向上が期待できます。特にループ内で呼ばれる関数を展開すると、呼び出し回数が多いために大きな効果が期待できます。なお、コンパイラはインライン展開された後のコードに対して最適化を実施します。そのため、大きな関数をインライン展開するとプログラムサイズが大きくなり、コンパイラの最適化効率が低下する恐れがあります。インライン展開は呼び出し頻度が高い小さな関数に指定すると効果的です。

【書式】

#pragma inline [(]<関数名>[,…][)]

#pragma inline は、関数本体の定義の前に指定してください。

#pragma inline で指定した関数に対しても外部定義を生成します。外部定義が不要な場合は、関数の宣言に static を指定してください。static を指定した関数がインラインｘ展開された場合は、当該関数の実体が生成されないため、サイズの削減が期待できます。

inline オプションを指定すると、指定されたサイズ分自動インライン展開を行います。自動インライン展開のサイズの限度を超える場合でも、#pragma inline 指定した関数はインライン展開を行います。

なお､#pragma inline の指定が行われていても､以下の条件を満たす場合はインライン展開されませんので、注意してください。

- #pragma inline 指定より前に関数の定義がある。
- 可変パラメータを持つ関数である。
- 関数内でパラメータのアドレスを参照している。
- 関数のアドレスを介して呼び出しを行っている。

```
#pragma inline(func)
static int func (int a, int b)
{
    return (a+b)/2;
}
int x;
void main(void)
{
    int (*func_p)(int,int);

    func_p = func;

    x=func_p(10,20);
}
```

関数アドレスを関数型ポインタに代入し、その関数型ポインタを使用して関数コールをすると、インライン展開指定関数でも､インライン展開されない。

【例】インライン展開のイメージ

ソースコード

```
#pragma inline(func)
static int func (int a, int b)
{
    return (a+b)/2;
}
int x;
void main(void)
{
    x=func(10,20);
}
```

展開イメージ

```
int x;
void main(void)
{
    int func_result;
    {
        int a_1=10, b_1=20;
        func_result=(a_1+b_1)/2;
    }
    x=func_result;
}
```

1.3 アセンブリ記述関数のインライン展開

C 言語でサポートしていない CPU 命令を使用したい場合や、C 言語で記述するよりもアセンブラで記述し、性能を向上させたい場合、アセンブラ埋め込みインライン展開機能を用いると便利です。C ソースファイルの関数に対して"#pragma inline_asm"でアセンブラ記述関数であることを宣言すると、関数内にアセンブリ記述を行うことができます(これをインラインアセンブリ関数と呼びます)。また、#pragma inline_asm での size=数値 の指定により、インラインアセンブリ関数のサイズを指定できます。サイズ指定を実施することにより、より効率が良い最適化が実施される事があります。

なお、#pragma inline_asm で指定するサイズは実際のオブジェクトサイズ以上の値を指定してください。オブジェクトサイズより小さい値を指定した場合、動作は保証されません。

【書式】

#pragma inline_asm [(]<関数名>[(size=数値)] [,…][)]

インラインアセンブリ関数を記述する際には、以下の点に注意してください。

- ラベルはローカルラベルを使用する。
  (ローカルラベル:‘？’で始まり 16 文字以内)
- リテラルプールを自動生成する命令は書かない(詳細は、【例 2】参照)。
- 定義の最後に RTS(リターン)命令は記述しない。
- 保証レジスタの退避／回復をする。
  (#pragma global_register で指定したレジスタを使用する場合も退避／回復が必要です。また、関数呼び出しがある場合、プロシージャレジスタ(PR)は書き換わるため、退避／回復が必要となります。)

```
extern void sub(void);

#pragma inline_asm (func(size=0x14))
static void func(void)
{
    .IMPORT _sub
    STS.L PR,@-R15
    MOV.L ?LOCAL1,R0
    JSR @R0
    NOP
    LDS.L @R15+,PR
    BRA   ?LOCAL2
    NOP
    .ALIGN 4
?LOCAL1:
    .DATA.L _sub
?LOCAL2:
}

void g(void)
{
    func(void);
}
```

関数呼び出しをする場合、PRの退避／回復が必要

インラインアセンブリ関数がある C ソースファイルは、出力フォーマット形式をアセンブリファイル(code=asmcode)とする必要があります。

[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]

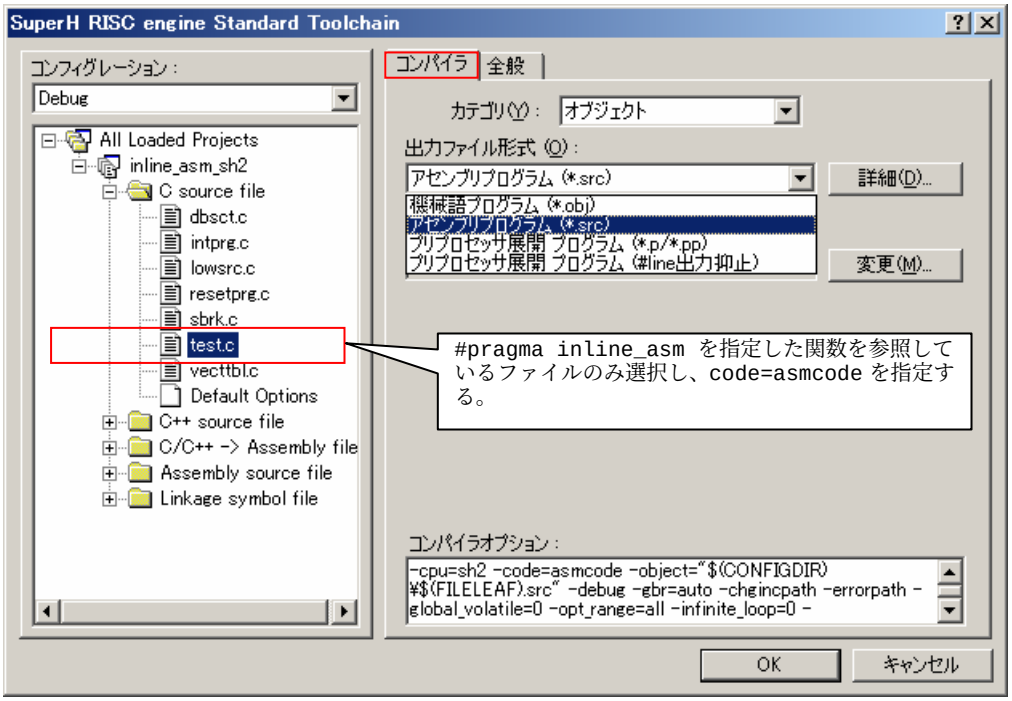


図 1-2

もし、#pragma inline_asm で指定した関数でコンパイルエラーになった場合、コンパイラのデバッグ情報を出力していると、C ソースの行情報が出力されるため、ルネサス統合開発環境には C ソースのライン情報が表示されます。したがって、表示されるライン情報からは、エラーの原因となっているアセンブリプログラムの行にはジャンプできません。コンパイラのデバッグ情報の出力を止めると、ルネサス統合開発環境に表示されるライン情報は、アセンブリプログラムの行が出力されます。#pragma inline_asm で指定した関数をデバッグする場合は、コンパイラのデバッグ情報の指定を抑止することをお勧めします。

なお、関数間のインタフェースはC/C++コンパイラの生成規則に従ってください(表 1-3、表 1-4)。

表 1-3 C 言語プログラムでの引数割り付けの一般規則

割付規則		
レジスタで渡される引数		スタックで渡される引数
引数格納用レジスタ	対象の型	
R4〜R7	char、unsigned char、bool、 short、unsigned short、 int、unsigned int、 long、unsigned long、 float(CPU が SH-2E、SH2A-FPU、SH-4、SH-4A 以外の場合)、ポインタ、データメンバへのポインタ、リファレンス	• 引数の型がレジスタ渡しの対象の型以外のもの • 他の引数がすでに R4〜R7 に割り付いている場合 • 他の引数がすでに FR4(DR4)〜FR11(DR10)に割り付いている場合 • long long,unsigned long long 型の引数 • __fixed 型、long __fixed 型、__accum 型、long __accum 型の引数
FR4〜FR11 *1	SH-2E のとき • 引数が float 型 • 引数が double 型かつ double=float オプション指定 SH2A-FPU、SH-4、SH-4A のとき • 引数が float 型かつ fpu=double オプション指定なし • 引数が double 型かつ fpu=single オプション指定	
DR4〜DR10 *2	SH2A-FPU、SH-4、SH-4A のとき • 引数が double 型かつ fpu=single オプション指定なし • 引数が float 型かつ fpu=double オプション指定	

[注釈] *1 SH-2E、SH2A-FPU、SH-4、SH-4A の単精度浮動小数点用のレジスタです。
*2 SH2A-FPU、SH-4、SH-4A の倍精度浮動小数点用レジスタです。

表 1-4 C 言語プログラムでのリターン値の型と設定場所

リターン値の型	設定場所
(singed) char、unsigned char、 (singed) short、unsigned short、 (singed) int、unsigned int、 long、unsigned long、ポインタ、bool、 リファレンス、データメンバへのポインタ	R0 : 32 ビット (signed) char,unsigned char の上位 3 バイト、(signed) short、unsigned short の上位 2 バイトの内容は保証しません。(ただし、rtnext オプション指定時は(signed) char,(singed) short 型は符号拡張、unsigned char,unsigned short 型はゼロ拡張を行います。)
float	FR0 : 32 ビット (1)SH-2E のとき • リターン値が float 型 • リターン値が double 型かつ double=float オプション指定 (2)SH2A-FPU,SH-4,SH-4A のとき • リターン値が float 型かつ fpu=double オプション指定なし • リターン値が浮動小数点型かつ fpu=single オプション指定
double、long double	DR0:64 ビット SH2A-FPU、SH-4、SH-4A のとき • リターン値が double 型かつ fpu=single オプション指定なし • リターン値が浮動小数点型かつ fpu=double オプション指定
構造体、共用体、クラス型、 関数メンバへのポインタ	リターン値設定領域(メモリ)
(signed)long long、unsigned long long	リターン値設定領域(メモリ)
__fixed、long __fixed、 __accum、long __accum	リターン値設定領域(メモリ)

【例 1】インラインアセンブリ関数内のアセンブリ記述例

ソースコード

```
/* Inline function definition */
/* FILE: inlasm.h */
#pragma inline_asm(rev4b(size=0x04))
static unsigned long rev4b(unsigned long p)
/* 関数はstatic で宣言*/
{
  ; 定義中のコメントは アセンブラの;（セミコロン）を使用する
    SWAP.W R4,R0
    SWAP.B R0,R0
  ; 最後にRTS 命令は記述しない
}
#pragma inline_asm(ovf)
static unsigned long ovf(void)
{
?LABEL001 ; インラインアセンブリ関数内ではローカルラベルを使用する
  ; ローカルラベル： ‘？’で始まり16 文字以内
    MOV R4,R0
      :
    CMP/EQ #1,R0
    BT ?LBABEL001
}
```

【例 2】リテラルプール自動生成の注意点

誤ったソースコード

```
#pragma inline_asm(f)
/* 正しくないインラインアセンブラの記述 */
static unsigned long f(void)
{
    MOV.L #H'f0000000,R0

  ; このような記述はアセンブラがリテラルプールを
  ; 自動生成するため、コンパイラ生成コードの
  ; アライメントが不正となる可能性があります
}
```

正しいソースコード1

```
#pragma inline_asm(f(size=0x0c))
/* 正しいインラインアセンブラの記述 */
static unsigned long f(void)
{
    MOV.L ?LOCAL1,R0 ;ローカルラベルからデータ参照
    BRA   ?LOCAL2    ;データ定義をジャンプ
    NOP
    .ALIGN 4
?LOCAL1:
    .DATA.L H'F0000000
?LOCAL2:
}
```

正しいソースコード2

```
#pragma inline_asm(f(size=0x06))
/* 正しいインラインアセンブラの定義 */
static unsigned long f(void)
{
     MOV         #-16,R0    ; H'FFFFFFF0
     SHLL8       R0
     SHLL16      R0
  ;ラベルからのデータ参照をしないようにする
}
```

【例 3】size 指定により、最適化が実施される例

サイズを指定することにより分岐判定処理がより最適化されます。

ソースコード(size指定なし)

```
#include <machine.h>
extern int a;

#pragma inline_asm (func)
static int func(void)
{
   NOP
}

void g(void)
{
    if (a) {
        func();
    }
    if (a) {
        nop();
    }
}
```

アセンブリ展開コード

```
_g:
            MOV.L       L16+2,R6    ; _a
            MOV.L       @R6,R2      ; a
            TST         R2,R2
            BF          L20
            MOV.L       L16+6,R3    ; L13
            JMP         @R3
            NOP
L20:
            BRA         L15
            NOP
L16:
            .RES.W      1
            .DATA.L     _a
            .DATA.L     L13
L15:
   NOP
            .ALIGN      4
            MOV.L       L18+2,R6    ; _a
            BRA         L17
            MOV.L       @R6,R2      ; a
L18:
            .RES.W      1
            .DATA.L     _a
L17:
            TST         R2,R2
            BT          L13
            NOP
L13:
            RTS
            NOP
```

ソースコード (size=0x20 指定)

```
#include <machine.h>
extern int a;

#pragma inline_asm (func(size=0x20))
static int func(void)
{
   NOP
}

void g(void)
{
    if (a) {
        func();
    }
    if (a) {
        nop();
    }
}
```

アセンブリ展開コード

```
_g:
            MOV.L       L15+2,R6    ; _a
            MOV.L       @R6,R2      ; a
            TST         R2,R2
            BT          L13
   NOP
            .ALIGN      4
            MOV.L       L15+2,R6    ; _a
            MOV.L       @R6,R2      ; a
            TST         R2,R2
            BT          L13
            NOP
L13:
            RTS
            NOP
L15:
            .RES.W      1
            .DATA.L     _a
```

1.4 レジスタの退避／回復コード出力の制御

関数の出入口で行われるレジスタの退避／回復を削除することで実行速度およびROM 効率の向上が図れます。#pragma noregsave、#pragma noregalloc、#pragma regsave を使用することで、表 1-5に示す保証するレジスタの退避／回復をきめ細かく制御することができます。

表 1-5 #pragma noregsave/noregalloc/regsave で制御される保証レジスタ

レジスタ	補足
R8～R14	-
FR12～FR15	SH-2E、SH2A-FPU、SH-4、SH-4A の単精度浮動小数点用レジスタ
DR12、DR14	SH2A-FPU、SH-4、SH-4A の倍精度浮動小数点用レジスタ

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。

(1) #pragma noregsave は、保証されるレジスタの退避／回復を関数の出入口で行わないことを指定します。

(2) #pragma noregalloc は、保証されるレジスタの退避／回復を関数の出入口で行わず、関数呼び出しを越えて、保証されるレジスタを使用しないオブジェクトを生成します。

(3) #pragma regsave は、保証されるレジスタ全ての退避／回復を関数の出入口で行い、関数呼び出しを超えて、保証されるレジスタを使用しないオブジェクトを生成します。

(4) #pragma regsave と#pragma noregalloc は同一関数に対して重複指定できます。重複指定した場合、保証されるレジスタ全ての退避／回復を関数の出入口で行い、関数呼び出しを越えて、保証されるレジスタを使用しないオブジェクトを生成します。

表 1-6 #pragma regsave/noregsave/noregalloc の動作

#pragma	レジスタ退避／回復	レジスタの使用
#pragma noregsave	保証されるレジスタの退避／回復を行わない	保証されるレジスタを使用する
#pragma noregalloc	保証されるレジスタの退避／回復を行わない	関数呼び出しを越えて 保証されるレジスタ を使用しない
#pragma regsave	保証されるレジスタを退避／回復	関数呼び出しを越えて 保証されるレジスタを使用しない 関数呼び出しを越えない範囲での 保証されるレジスタの使用頻度は低い
#pragma regsave + #pragma noregalloc	保証されるレジスタを退避／回復	関数呼び出しを越えて 保証されるレジスタを使用しない 関数呼び出しを越えない範囲での 保証されるレジスタの使用頻度は高い

#pragma noregsave を指定した関数は、通常の関数から呼び出されると動作が不正となる可能性があります。以下の#pragma 指定を行った関数から呼び出すようにしてください。

- #pragma regsave を指定した関数
- #pragma regsave を指定した関数から呼び出される#pragma noregalloc を指定した関数

#pragma noregsave/ noregalloc/ regsave 指定された関数を呼び出す場合は、#pragma がプロジェクト全体で統一されている必要があります。このため、共通でインクルードされるヘッダファイルに#pragam 指定することをお勧めします。

【例 1】

呼び出し元に戻らない関数の場合、レジスタの退避／回復を行う必要がないため、"#pragma noregsave"を指定することにより、オブジェクトサイズおよび実行速度を向上できます

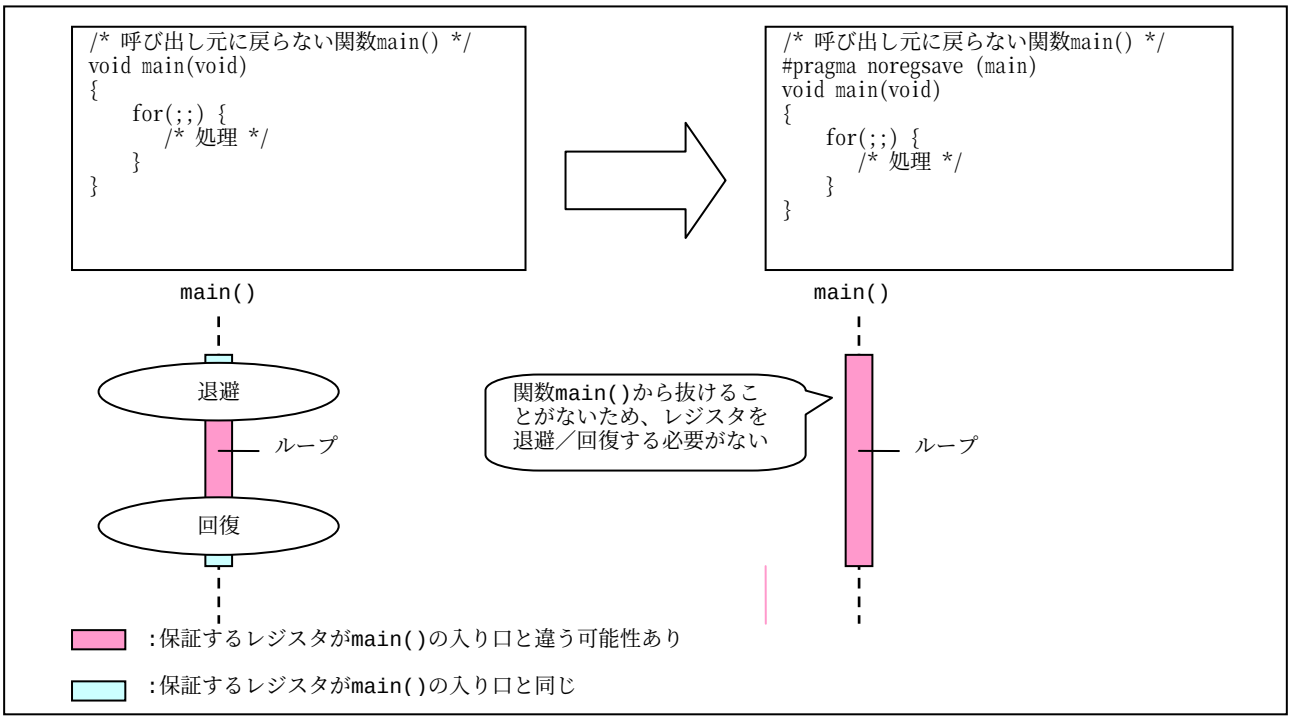


図 1-3

【例 2】

関数 b1()、b2()で保証するレジスタが使用される場合、関数 b1()、b2()の出入口で保証するレジスタの退避／回復が行われます。例えば、関数 a()から b1()、b2()を頻繁に呼び出すような場合、保証するレジスタの退避／回復も頻繁となり効率が悪くなります。

もし、関数 a()で保証するレジスタを使用していなければ、関数 b1()、b2()において保証するレジスタを退避／回復する必要がなくなります。関数 a()に regsave を指定すると、関数 a()の入り口と出口で保証するレジスタの退避／回復を行い、さらに関数呼び出しを越えて保証するレジスタを使用しなくなります。 関数 b1()、b2()に noregsave を指定すると保証するレジスタの退避／回復を抑止できます。

これにより、保証するレジスタの退避／回復は関数 a()にて行われ、関数 b1()、b2() では保証するレジスタを退避／回復することなく使用することができます。

このようにレジスタの退避／回復の位置を変更することで、レジスタの退避／回復の頻度を減らすことができます。

regsave と共に noregalloc を指定すると、関数呼び出しを越えない範囲における保証するレジスタの使用頻度が高くなり、最適化に有利となります。regsave を指定するときは共に noregalloc を指定することをお勧めします。

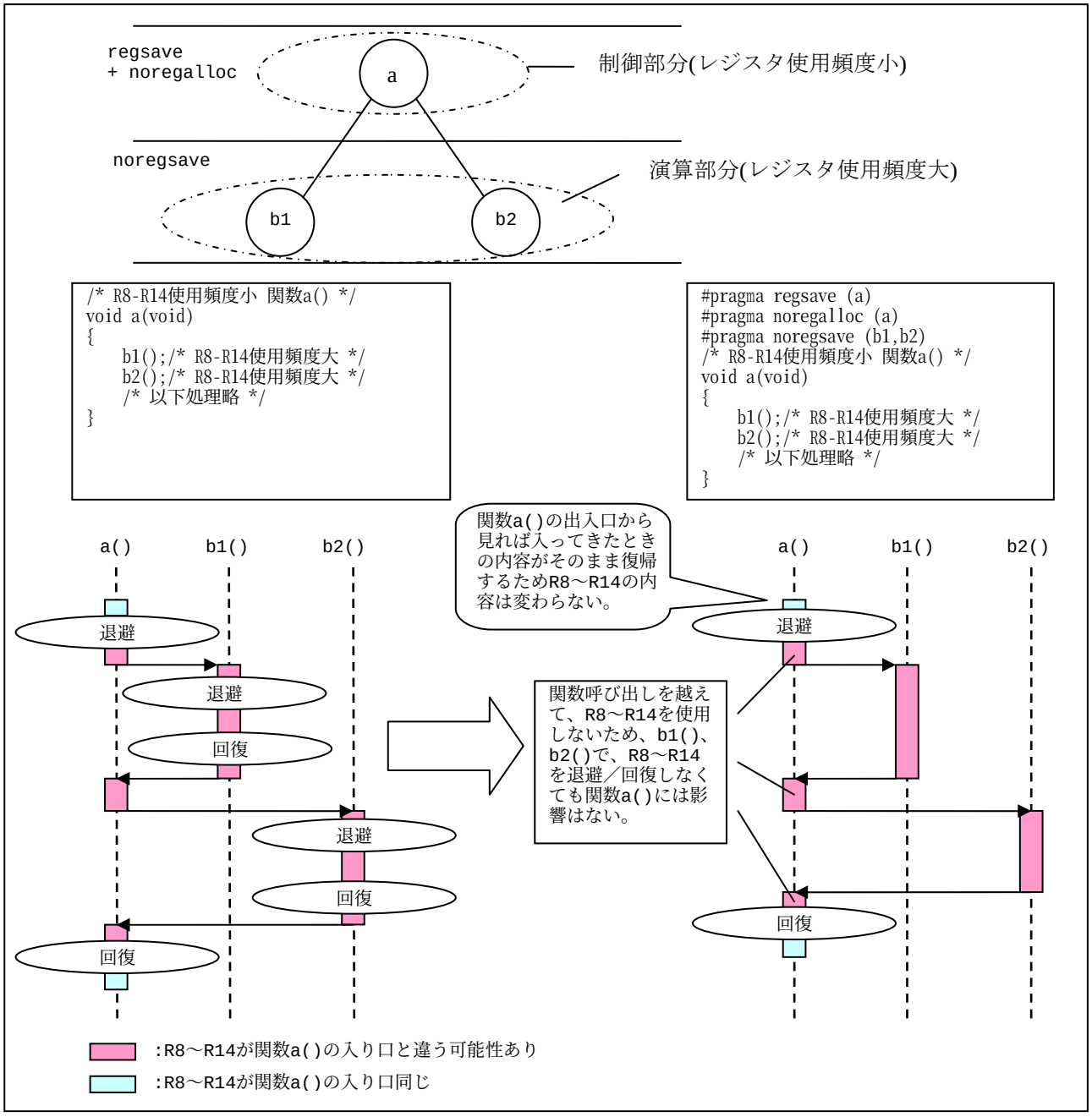


図 1-4

【例 3】

関数 b1()、b2()で保証するレジスタ が使用される場合、関数 b1()、b2()の出入口で、保証するレジスタ の退避／回復が行われます。例えば、関数 a() および 関数 a()から呼び出される関数 c2()からも b1()、b2()を頻繁に呼び出すような場合、保証するレジスタの退避／回復も頻繁となり効率が悪くなります。

もし、関数 a() および 関数 c2()で関数呼び出しを越えて保証するレジスタを使用していなければ、関数 b1()、b2()において保証するレジスタを退避／回復する必要がなくなります。関数 a()に regsave ＋ noregalloc を指定すると、関数 a()の入り口と出口で保証するレジスタの退避／回復を行い、さらに関数呼び出しを越えて保証するレジスタを使用しなくなります。関数 c2()に noregalloc を指定すると関数呼び出しを越えて保証するレジスタを使用しなくなります。関数 b1()、b2()に noregsave を指定すると保証するレジスタの退避／回復を抑止できます。

これにより、保証するレジスタの退避／回復は関数 a()にて行われ、関数 b1()、b2() では保証するレジスタを退避／回復することなく使用することができます。このようにレジスタの退避／回復の位置を変更することで、レジスタの退避／回復の頻度を減らすことができます。

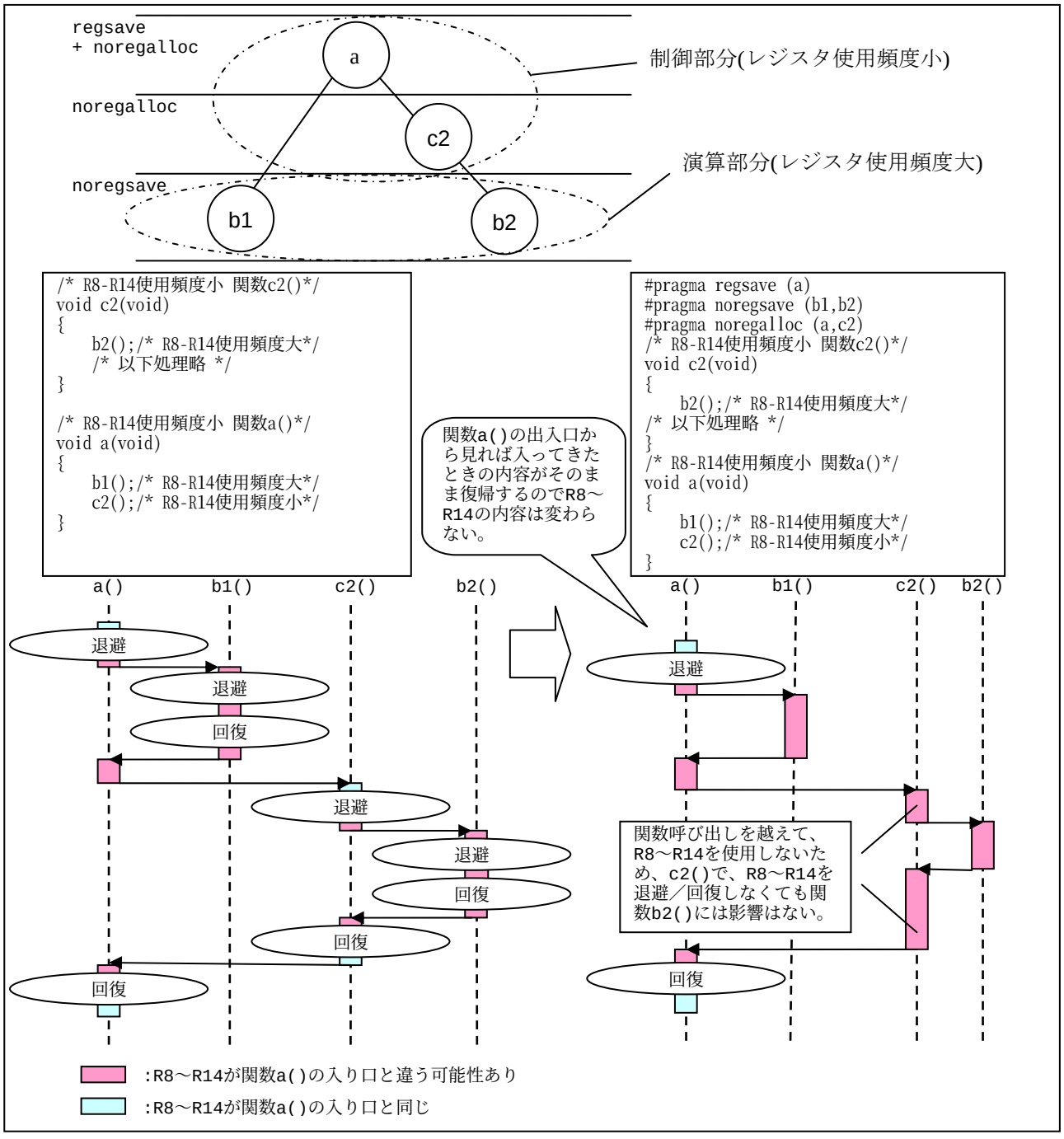


図 1-5

1.5 大域変数のレジスタ割り付け

<変数名>で指定されたグローバル変数に、〈レジスタ名〉で指定したレジスタを割り付けることができます。レジスタを割り付けることで、ロード、ストアの命令を減らすことができます。

#pragma global_register 指定は全ファイルに指定する必要があります。#pragma global_register の宣言を含むヘッダファイルを “preinclude” オプションで指定することで、全ファイルに対して指定することができます(ルネサス統合開発環境での設定方法は図 1-6参照)。

また、標準ライブラリについても #pragma global_register の指定が必要です。#pragma global_register の宣言を含むヘッダファイルを “preinclude” オプションで指定してください(ルネサス統合開発環境での設定方法は図 1-7参照)。

【書式】

#pragma global_register [(]<変数名>=<レジスタ名>[,…][)]

- グローバル変数で、整数型、浮動小数点数型、もしくはポインタ型の変数に使用できます。SH2A-FPU、SH-4、SH-4A 以外の CPU では、"double=float"オプションを指定した場合のみ、double 型の変数を指定できます。
- 指定可能なレジスタは、R8～R14、FR12～FR15（SH-2E、SH2A-FPU、SH-4、SH-4A の場合）、DR12、DR14（SH2A-FPU、SH-4、SH-4A の場合）です。

  o FR12～FR15 に指定可能な変数の型
  (i) SH-2E の場合
  - float 型変数
  - double 型変数(double=float オプション指定)

  (ii) SH2A-FPU、SH-4、SH-4A の場合
  - float 型変数(fpu=double オプション指定なし)
  - double 型変数(fpu=single オプション指定)

  o DR12、DR14 に指定可能な変数の型
  (i) SH2A-FPU、SH-4、SH-4A の場合
  - float 型変数(fpu=double オプション指定)
  - double 型変数(fpu=single オプション指定なし)

- 初期値の設定はできません。また、アドレスの参照もできません。
- #pragma global_register の指定が全ファイル/ライブラリで統一されていない場合の動作は保証されません。
- 静的データメンバの指定は可能ですが、非静的データメンバの指定は不可能です。

[ルネサス統合開発環境でのオプション設定方法]

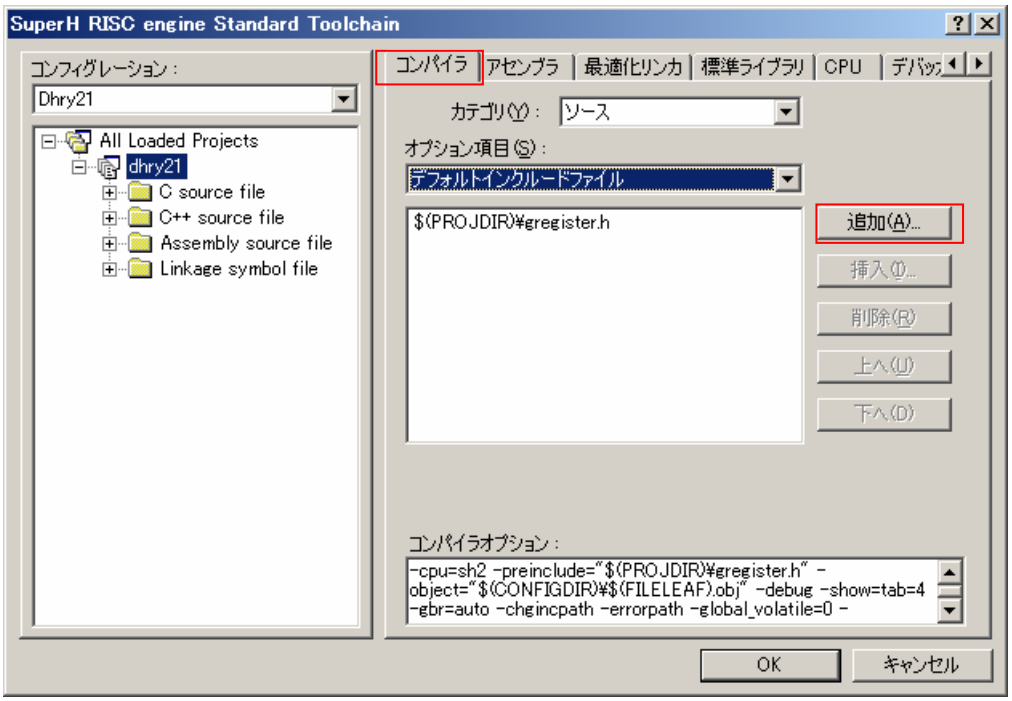


図 1-6

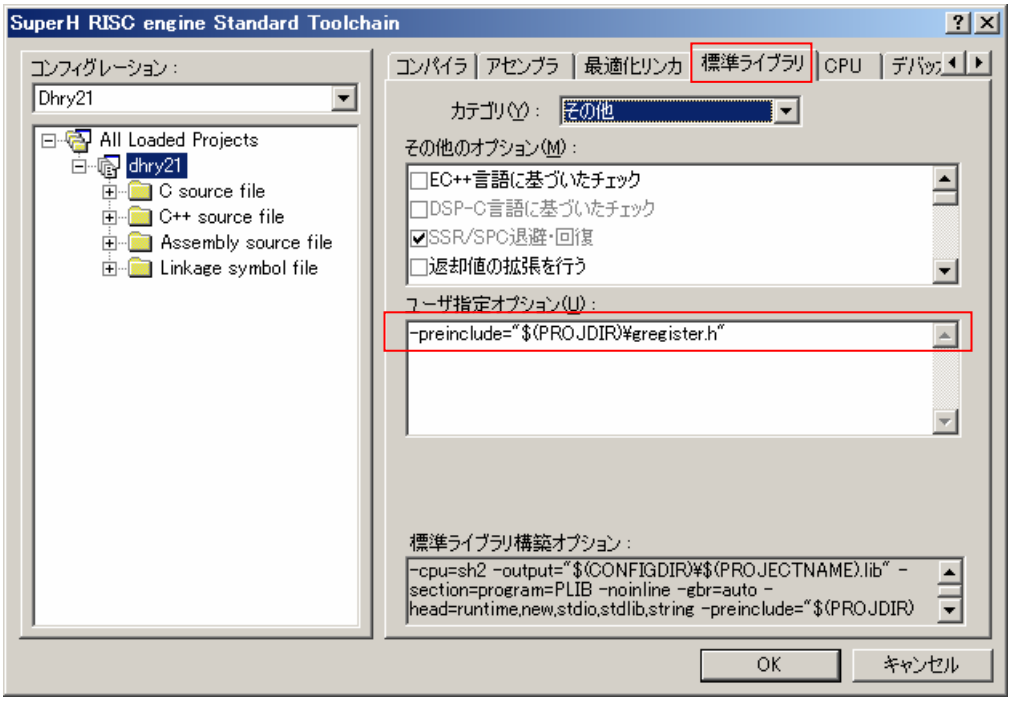


図 1-7

【例】

ソースコード

```
#pragma global_register(x=R13,y=R14)

int x;
char *y;

void func1(void)
{
    x++;
}

void func2(void)
{
    *y=0;
}

void func(int a)
{
    x = a;
    func1();
    func2();
}
```

アセンブリ展開コード

```
_func1:
          RTS
          ADD         #1,R13
_func2:
          MOV         #0,R2       ; H'00000000
          RTS
          MOV.B       R2,@R14     ; *(y)
_func:
          STS.L       PR,@-R15
          BSR         _func1
          MOV         R4,R13
          BRA         _func2
          LDS.L       @R15+,PR
```

1.6 GBR ベース変数の指定

GBR ベースを指定した変数は、GBR 相対アクセス可能となります。相対値でアクセス可能であるため、変数アドレスロードが不要となり、コンパクトなオブジェクトが生成されます。

ただし、指定可能な変数の数(サイズ)に限りがあります。使用頻度の高い変数を選別し、GBR ベース変数の指定をしてください。

【書式】

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

"#pragma gbr_base"は、GBR の指すアドレスからオフセット 0～127 バイトの範囲内に変数が割り付くことを指定します。ここで指定した変数は、"$G0"セクションに割り付けられます。

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

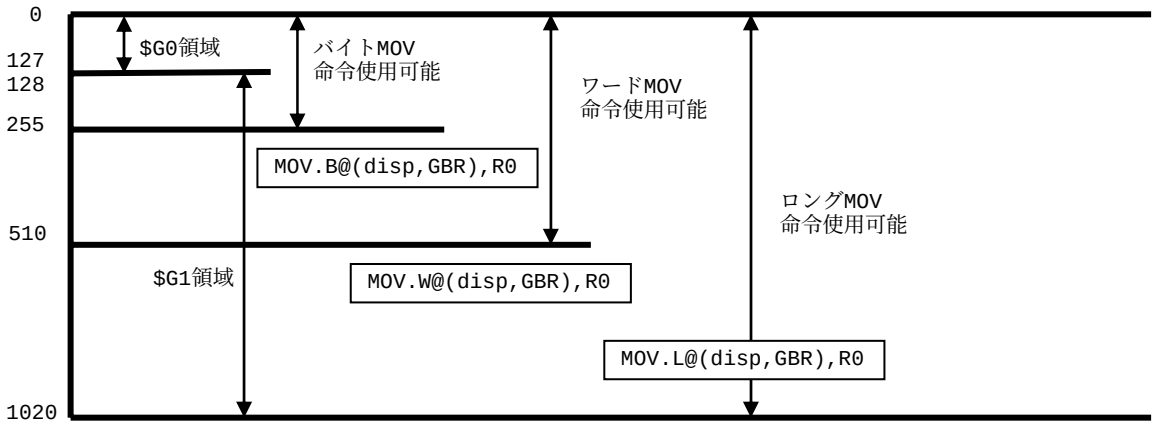


図 1-8

特に頻繁にアクセスされるデータ、ビット演算が行われるデータは、なるべく"$G0"セクションに割り付けてください。"$G1"セクションより"$G0"セクションに割り付けられたデータにアクセスするほうが実行速度、サイズともに効率の良いオブジェクトが生成されます(【例 2】参照)。

"#pragma gbr_base"および"#pragma gbr_base1"宣言された変数は、変数宣言された順番に各セクションに割り付けられます。異なるサイズの変数が交互に宣言されるとデータサイズが増えるため、注意が必要です。同じサイズの変数はなるべくまとめて宣言してください。

GBR ベース変数の指定で、上記に記載したオフセット内にデータが収まっていない場合は、リンク時に以下のエラーが発生します。

```
L2330 (E)Relocation size overflow
```

本エラーが出た時は制限を超えるデータに対する、#pragma gbr_base/ gbr_base1 宣言を削除してください。

#pragma gbr_base を使用する場合は "$G0"セクションをリンケージエディタで設定してください。#pragma gbr_base1 を使用する場合は"$G0"/"$G1"セクションをリンケージエディタで設定してください。"$G1"セクションは("$G0"セクションの先頭+0x80)の番地に設定してください。

なお、プロジェクト全体で #pragma gbr_base/ gbr_base1 指定を統一することをお勧めします。#pragma の指定をプロジェクト全体で統一するには、“preinclude” オプションを使用すると便利です。

#pragma gbr_base/gbr_base1 で指定した変数は、初期値指定あり、初期値指定なしに関わらず"$G0"/"$G1"セクションに割り付けられます。コンパイラは"$G0"/"$G1"セクションを初期化データとして扱いオブジェクトを生成します。

初期化データは初期値を持つデータ(変数)です。初期値は ROM 領域に持つ必要がありますが、データはプログラム実行中に書き換えられる可能性があるため、RAM 領域に配置する必要があります。

したがって、ROM 領域に初期値を配置しておき、その ROM 領域の初期値データを RAM 領域にコピーする処理が必要になります。また、ROM 領域に初期値を配置し、RAM 領域のアドレスでデータをアクセスするためには、リンケージエディタにて ROM 化支援オプションを指定する必要があります。"$G0"/"$G1"セクションについても、ROM 領域の初期値データを RAM 領域にコピーする処理を追加し、さらに、リンケージエディタにて ROM 化支援オプションを指定してください。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:

＜導入ガイド＞スタートアップルーチンガイド SH-1, SH-2, SH-2A 編 4.メモリ初期化」を参照してください。

"$G1"セクション用の RAM 領域は("$G0"セクション用の RAM 領域+0x80)番地に配置してください。

GBR ベース変数を使用する場合は、あらかじめ、"$G0"セクションをコピーする RAM 領域の先頭アドレスを GBR レジスタに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。

下記は､ROM 領域に配置した"$G0"セクションを RAM 領域に配置した"$RG0"セクションにコピーした場合の例です。

初期化プログラム

```
#include <machine.h>

/* main 関数の前に実行される関数 */
void PowerON_Reset(void)
{
        :
    set_gbr(__sectop("$RG0")); /* GBR レジスタに$RG0 セクションの先頭を設定 */
        :
}
```

GBR ベースを活用するための手順は以下の通りです。

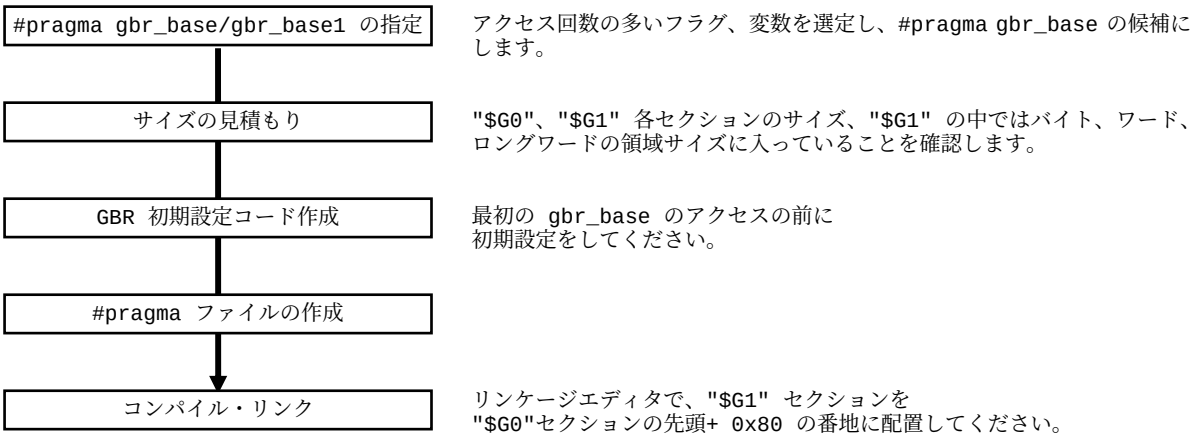


図 1-9

#pragma gbr_base/gbr_base1 を指定する場合は、“gbr=user” を指定してください (図 1-10)。“gbr=user” を指定しない場合は、#pragma gbr_base/gbr_base1 に対してウォーニングが出力され、#pragma gbr_base/gbr_base1 の指定が無視されます。

なお、“GBR 相対アクセスコード自動生成” の選択で、“ユーザ指定(GBR 相対論理演算生成)” を指定すると、“gbr=user” および “logic_gbr” が指定されます。これにより、#pragma gbr_base で指定された GBR ベース変数以外の演算にも、GBR 相対の論理演算命令が使用されることがあります。

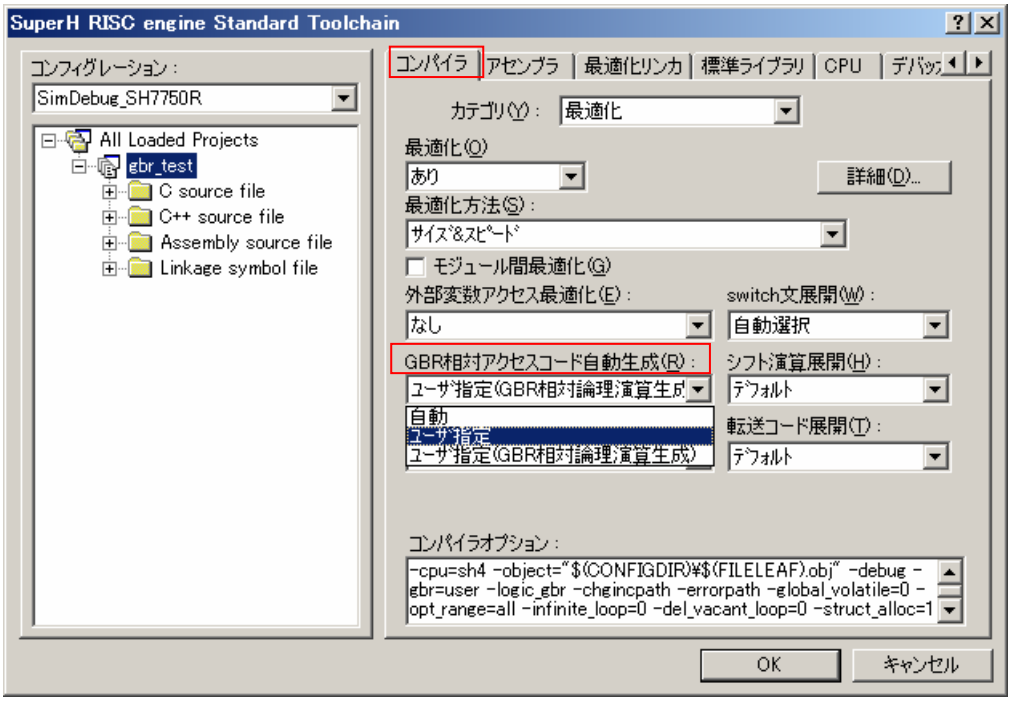


図 1-10

【例 1】

コンパイルオプション "gbr=user" を指定した場合に生成されるコードの例です。

#pragma gbr_base未指定ソースコード

```
struct BitField {
    unsigned char a:1;
    unsigned char b:1;
    unsigned char c:1;
    unsigned char d:1;
    unsigned char e:1;
    unsigned char f:1;
    unsigned char g:1;
    unsigned char h:1;
} bitf;

struct {
    char  c;
    short s;
    long  l;
} x, y;

void f (void)
{
    bitf.a = 1;
    bitf.b = 0;

    if (bitf.c) {
        bitf.d = 1;
    } else {
        bitf.e = 1;
    }

    x.c = y.c;
    x.s = y.s;
```

#pragma gbr_base指定ソースコード

```
#pragma gbr_base  (bitf)
#pragma gbr_base1 (x,y)

struct BitField {
    unsigned char a:1;
    unsigned char b:1;
    unsigned char c:1;
    unsigned char d:1;
    unsigned char e:1;
    unsigned char f:1;
    unsigned char g:1;
    unsigned char h:1;
} bitf;

struct {
    char  c;
    short s;
    long  l;
} x, y;

void f (void)
{
    bitf.a = 1;
    bitf.b = 0;

    if (bitf.c) {
        bitf.d = 1;
    } else {
        bitf.e = 1;
    }

    x.c = y.c;
    x.s = y.s;
```

```
    x.l = y.l;
}
```

アセンブリ展開コード

```
_f:
    MOV.L       L14,R5      ; _bitf
    MOV.B       @R5,R0      ; (part of)bitf
    AND         #191,R0
    OR          #128,R0
    TST         #32,R0
    BT/S        L12
    MOV.B       R0,@R5      ; (part of)bitf
    BRA         L13
    OR          #16,R0
L12:
    OR          #8,R0
L13:
    MOV.L       L14+4,R7    ; _y
    MOV.B       R0,@R5      ; (part of)bitf
    MOV.L       L14+8,R6    ; _x
    MOV.B       @R7,R1      ; y.c
    MOV.W       @(2,R7),R0  ; y.s
    MOV.L       @(4,R7),R4  ; y.l
    MOV.B       R1,@R6      ; x.c
    MOV.W       R0,@(2,R6)  ; x.s
    RTS
    MOV.L       R4,@(4,R6)  ; x.l
L14:
    .DATA.L     _bitf
    .DATA.L     _y
    .DATA.L     _x
    .SECTION    B,DATA,ALIGN=4
_bitf:                      ; static: bitf
    .RES.B      1
    .RES.B      1
    .RES.W      1
_x:                         ; static: x
    .RES.L      2
_y:                         ; static: y
    .RES.L      2
```

```
    x.l = y.l;
}
```

アセンブリ展開コード

```
_f:
    MOV         #_bitf-(STARTOF $G0),R0
    AND.B       #191,@(R0,GBR); (part of)bitf
    OR.B        #128,@(R0,GBR); (part of)bitf
    MOV.B       @(_bitf-(STARTOF $G0),GBR),R0
                                ; (part of)bitf
    TST         #32,R0
    BT          L12
    BRA         L13
    OR          #16,R0
L12:
    OR          #8,R0
L13:
    MOV.B       R0,@(_bitf-(STARTOF $G0),GBR)
                                ; (part of)bitf
    MOV.B       @(_y-(STARTOF $G0),GBR),R0; y.c
    MOV.B       R0,@(_x-(STARTOF $G0),GBR); x.c
    MOV.W       @(_y-(STARTOF $G0)+2,GBR),R0; y.s
    MOV.W       R0,@(_x-(STARTOF $G0)+2,GBR); x.s
    MOV.L       @(_y-(STARTOF $G0)+4,GBR),R0; y.l
    RTS
    MOV.L       R0,@(_x-(STARTOF $G0)+4,GBR); x.l
    .SECTION    $G0,DATA,ALIGN=4
_bitf:          ; static: bitf
    .DATAB.B    1,0
    .SECTION    $G1,DATA,ALIGN=4
_x:             ; static: x
    .DATAB.L    2,0
_y:             ; static: y
    .DATAB.L    2,0
```

【例 2】gbr_base と gbr_base1 の比較

コンパイルオプション“gbr=user” を指定した場合の例です。#pragma gbr_base を指定し"$G0"セクションに配置すると、リテラルアクセスがなくなります。

#pragma gbr_base1指定ソースコード

```
#pragma gbr_base1 (bitf)

struct BitField {
    unsigned char a:1;
    unsigned char b:1;
    unsigned char c:1;
    unsigned char d:1;
    unsigned char e:1;
    unsigned char f:1;
    unsigned char g:1;
    unsigned char h:1;
} bitf;

void f (void)
{
    bitf.a = 1;
    bitf.b = 0;
}
```

アセンブリ展開コード

```
_f:
    MOV.W       L11,R0 ; _bitf-(STARTOF $G0)
    AND.B       #191,@(R0,GBR);(part of)bitf
    RTS
    OR.B        #128,@(R0,GBR);(part of)bitf
L11:
    .DATA.W     _bitf-(STARTOF $G0)
    .SECTION    $G1,DATA,ALIGN=4
_bitf:                      ; static: bitf
    .DATAB.B    1,0
```

#pragma gbr_base指定ソースコード

```
#pragma gbr_base  (bitf)

struct BitField {
    unsigned char a:1;
    unsigned char b:1;
    unsigned char c:1;
    unsigned char d:1;
    unsigned char e:1;
    unsigned char f:1;
    unsigned char g:1;
    unsigned char h:1;
} bitf;

void f (void)
{
    bitf.a = 1;
    bitf.b = 0;
}
```

アセンブリ展開コード

```
_f:
    MOV         #_bitf-(STARTOF $G0),R0
    AND.B       #191,@(R0,GBR); (part of)bitf
    RTS
    OR.B        #128,@(R0,GBR); (part of)bitf
    .SECTION    $G0,DATA,ALIGN=4
_bitf:                      ; static: bitf
    .DATAB.B    1,0
```

2. その他の便利な拡張機能

本章では、性能向上以外に便利な拡張機能について説明します。便利な拡張機能を表 2-1に示します。

表 2-1 便利な拡張機能

No	#pragma	機能	参照
1	#pragma section	**セクションの切り替え指定**	2.1
2	#pragma bit_order	**ビットフィールドの並び順指定**	2.2
3	#pragma pack #pragma unpack	**構造体、共用体、クラスのアライメント数指定**	2.3

2.1 セクションの切り替え指定

コンパイラの出力するセクション名を切り替えることができます。

例えば、あるモジュールを外付け RAM に、別のモジュールを内蔵 RAM に割り付けたい場合など、別々のアドレスに割り付けたい場合があります。分割したいセクションそれぞれに名称をつけます。そして、それぞれのセクションに対して配置したいアドレスをリンケージエディタで指定します。

#pragma section で、セクション名を指定しない場合、それ以降はデフォルトのセクション名が適用されます。

【書式】

#pragma section [{<名前>|<数値>}]

デフォルト、切り替え後のセクション名を表 2-2に示します。

表 2-2 セクション切り替え機能とセクション名

	対象領域	指定方法	デフォルト名	切り替え後
1	**プログラム領域**	#pragma section <XX>	P	P<XX>
2	**定数領域**		C	C<XX>
3	**初期化データ領域**		D	D<XX>
4	**未初期化データ領域**		B	B<XX>

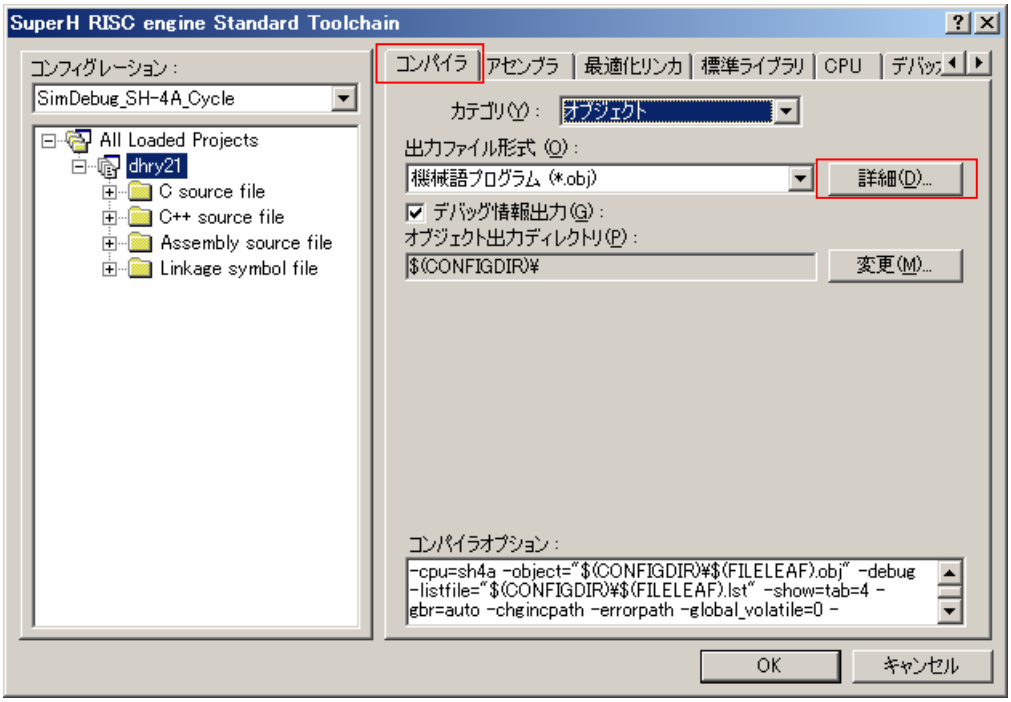
“section” オプションで、デフォルトのセクション名を変更できます。

[“secsion”オプションの書式]

SEction = <sub>[,…]

<sub>: { Program = <セクション名> |
Const = <セクション名> |
Data = <セクション名> |
Bss = <セクション名> }

[ルネサス統合開発環境でのオプション設定方法]



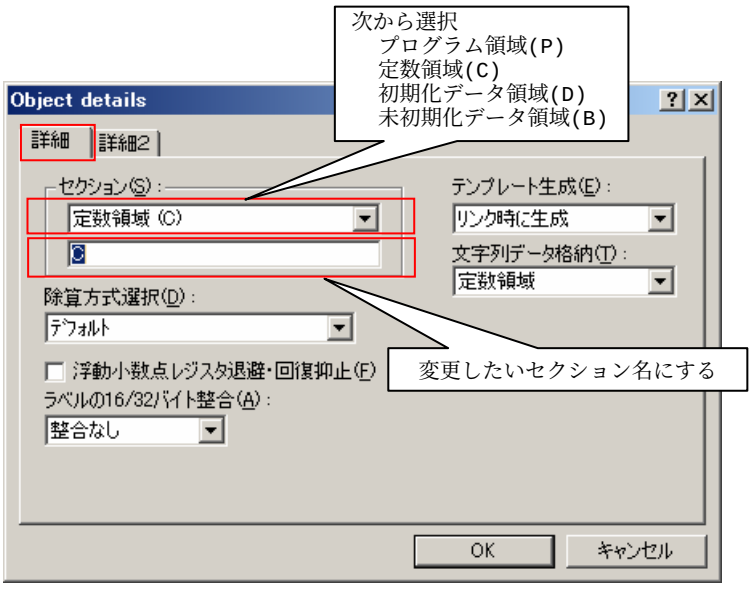


図 2-1

【例 1】

ソースコード

```
#pragma section abc

int a;          /* a はセクションBabc に割り付きます */
const int c=1; /* c はセクションCabc に割り付きます */
void f(void)    /* f はセクションPabc に割り付きます */
{
    a=c;
}

#pragma section

int b;         /* b はセクションB に割り付きます */
void g(void)  /* g はセクションP に割り付きます */
{
    b=c;
}
```

【例 2】

section=program=PX,const=CX,bss=BX と指定した場合

ソースコード

```
#pragma section abc

int a;            /* a はセクションBXabc に割り付きます */
const int c=1;  /* c はセクションCXabc に割り付きます */
void f(void)     /* f はセクションPXabc に割り付きます */
{
    a=c;
}

#pragma section

int b;          /* b はセクションBX に割り付きます */
void g(void)  /* g はセクションPX に割り付きます */
{
    b=c;
}
```

2.2 ビットフィールドの並び順指定

#pragma bit_order 指定を使用すると、ビットフィールドの並び順を変更することができます。マイコンによってはビットフィールドの並び規則が違うものがあります。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。

ファイルごとのビットフィールドの並び順指定はオプションでも指定できます。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。

【書式】

#pragma bit_order [{left|right}]

left を指定した場合は上位ビット側から、right を指定した場合は下位ビット側から、それぞれメンバが割り付けられます。デフォルトの設定は上位ビット側から割り付けとなります。#pragma bit_order の指定で、left|right を省略すると、その行以降はオプションに従います。

【例 1】

#pragma bit_order left と#pragma bit_order right で（データ並びイメージ）のように、それぞれ左右に割り付きます。

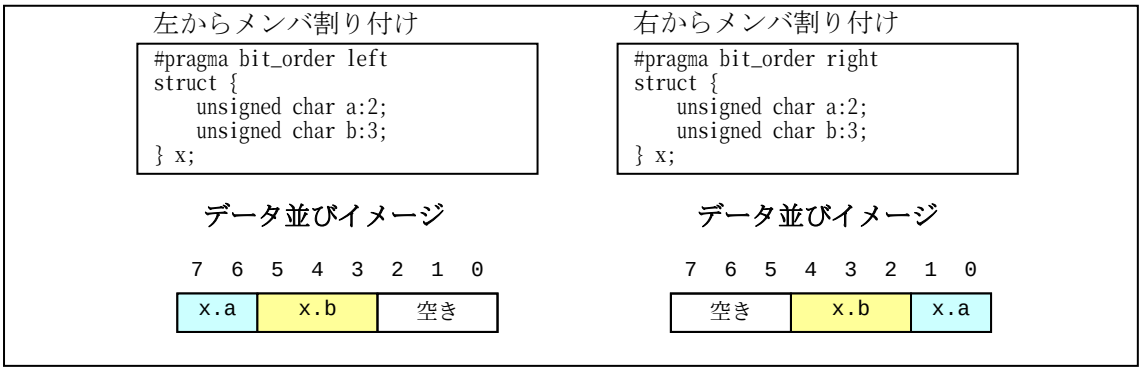


図 2-2

【例 2】

同じサイズの型指定子が連続している場合は、可能な限り同じ領域に詰め込みます。

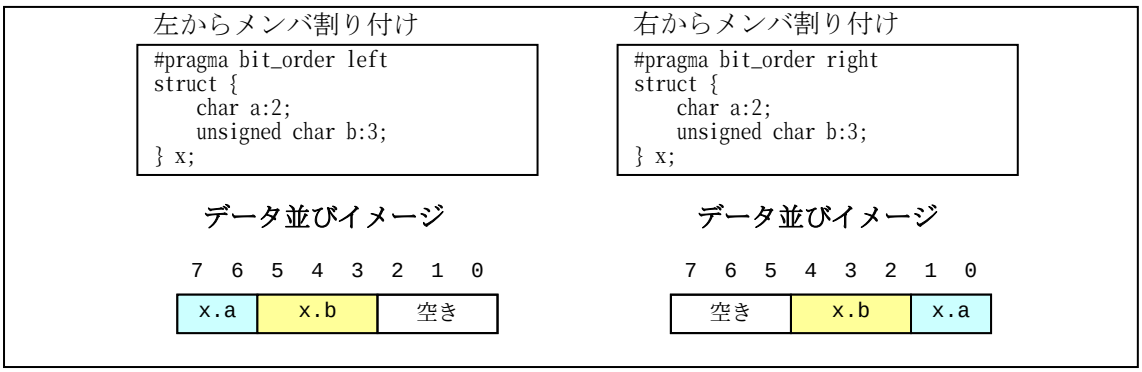


図 2-3

【例 3】

異なるサイズの型指定子が連続している場合は、次の領域に割り付けます。

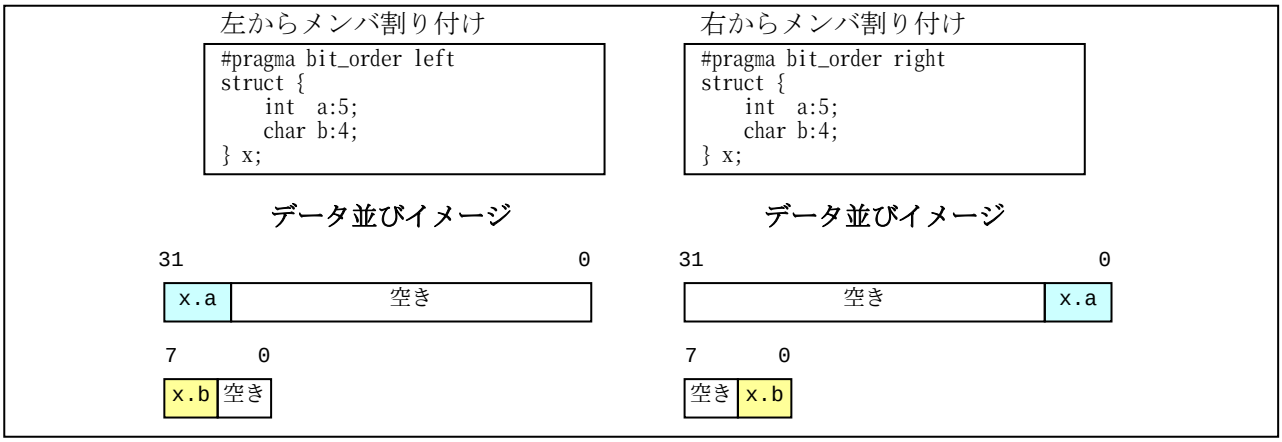


図 2-4

【例 4】

同じサイズの型指定子が連続していても、詰め込み先の領域の残りビットが、次のビットフィールドのサイズより小さい場合は、残りの領域は未使用領域となり、次の領域に割り付けます。

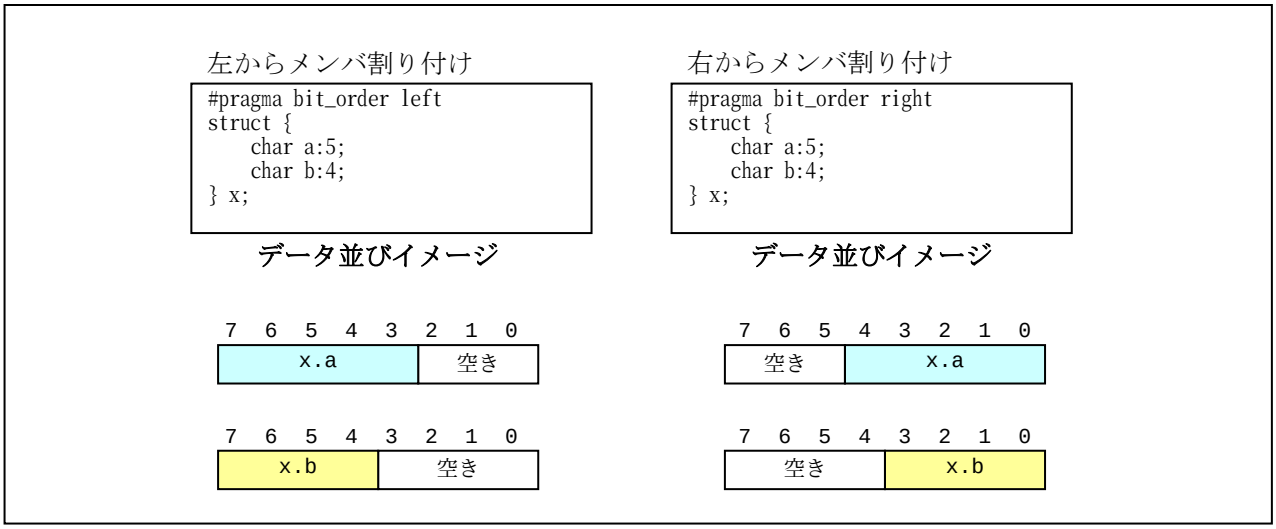


図 2-5

【例 5】

ビット幅 0 のビットフィールドのメンバを指定すると、次のメンバからは、強制的に次の領域に割り付けます。

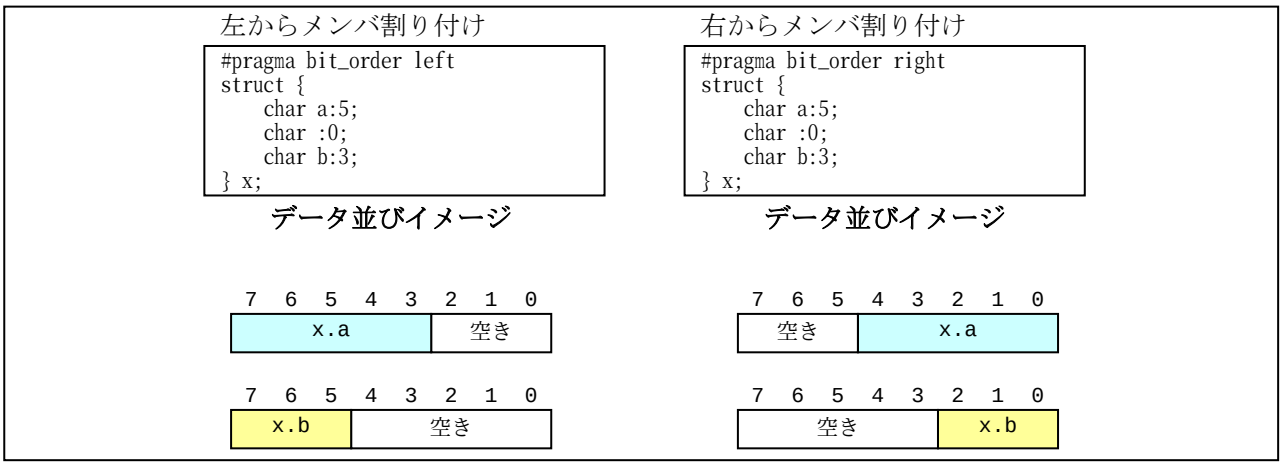


図 2-6

【例 6】

エンディアンのデフォルトはビックエンディアンです。一部の CPU はエンディアンをオプションで変更できます。リトルエンディアンを指定した場合(endian=little)、各領域の並び順はビックエンディアンでのバイトの並びと逆順になります。

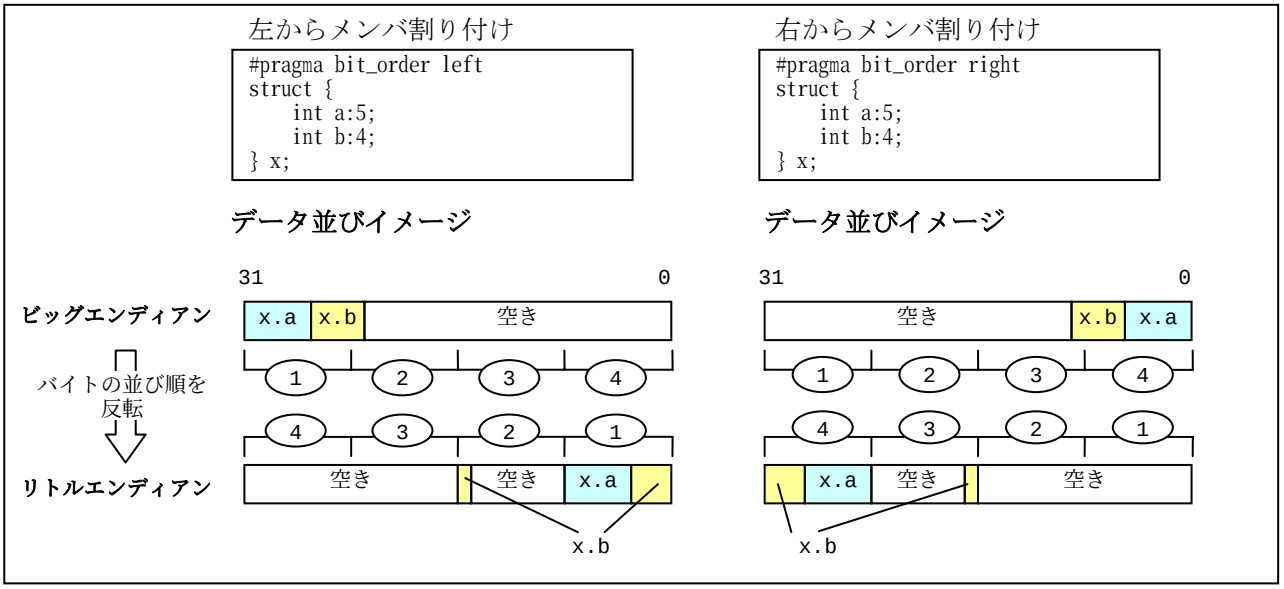


図 2-7

【注意事項】

ルネサス統合開発環境で作成された iodefine.h にある IO レジスタ用構造体については、#pragma もしくは、オプションでビットフィールドの並び順を下位ビットからメンバを割り付けるように指定すると、IO レジスタ用構造体のメンバが示すアドレスが正しいアドレスを指しません。

iodefine.h の先頭に、#pragma bit_order left を、iodefine.h の最後に、#pragma bit_order を記述してください。

```
/* iodefine.h */
#pragma bit_order left

    /* 略 */

#pragma bit_order
```

2.3 構造体、共用体、クラスのアライメント数指定

SH マイコンでは、4 バイトデータに 1 命令でアクセスするには、そのデータが 4 の倍数アドレスに配置されている必要があります。同様に、2 バイトデータに 1 命令でアクセスするには、2 の倍数アドレスに配置されている必要があります。このデータサイズに応じた配置アドレスのことをアライメント数と呼びます。

4 の倍数に配置される場合は”アライメント数 4“ となります。各型のアライメント数については「コンパイラユーザーズマニュアル 10.1.2 データの内部表現」をご参照ください。

構造体において(共用体・クラスも同様です)、1 バイト、2 バイト、4 バイトのメンバが混在している場合、各メンバはそれぞれのアライメント数に従うためにメンバとメンバの間に空き領域が発生する場合があります。

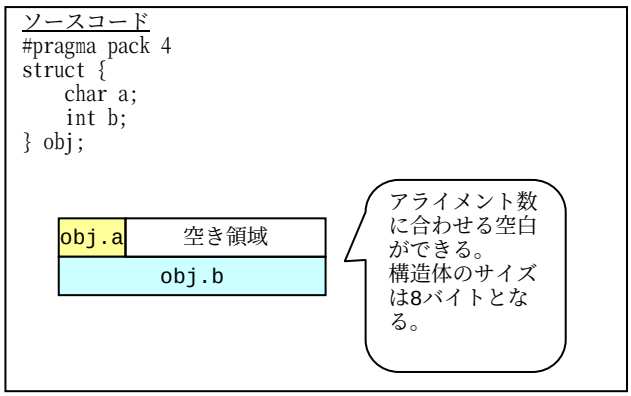


図 2-8

通信のプログラムで使用される構造体など、構造体に空き領域を作りたくない場合があります。この様な場合は、#pragma 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。ただし、アライメント数を 1 にした構造体は各メンバへのアクセスが全てバイトアクセスとなり、プログラムサイズが増加することがあります。また、アライメント数を 1 とした構造体メンバへはポインタを用いてアクセスすることができません。ポインタでのアクセスが行われた場合、ウォーニングが出力されます。

“pack” オプションを使用することで、ファイルごとに構造体のアライメント数を指定することができます。“pack” オプションと #pragma が同時に指定された場合は、#pragma の指定が優先されます。

【書式】

#pragma pack {1|4}
#pragma unpack

これらの指定によるアライメント数は次のようになります。

表 2-3 構造体、共用体、クラスメンバのアライメント数

指定	#pragma pack 1	#pragma pack 4	#pragma unpack **または指定なし**
[unsigned]char	1	1	1
[unsigned]short、__fixed	1	2	pack **オプションに従う**
[unsigned]int, [unsigned]long, [unsigned]long long, long __fixed, __accum, long, __accum, **浮動小数点数型**, **ポインタ型**	1	4	pack **オプションに従う**
アライメント数が 1 **の構造体、共用体、クラス**	1	1	1
アライメント数が 2 **の構造体、共用体、クラス**	1	2	pack **オプションに従う**
アライメント数が 4 **の構造体、共用体、クラス**	1	4	pack **オプションに従う**

【例】

構造体の割り付け方はそれぞれ以下のようになります。

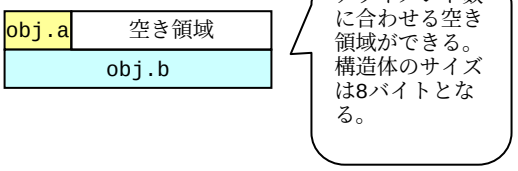
#pragma pack 4 指定ソースコード

```
#pragma pack 4
struct {
    char a;
    int b;
} obj;

int func(void)
{
    return obj.b;
}
```

アセンブラ展開コード

```
_func:
          MOV.L       L11+2,R6    ; _obj
          RTS
          MOV.L       @(4,R6),R0 ; obj.b
L11:
          .RES.W      1
          .DATA.L     _obj
          .SECTION    B,DATA,ALIGN=4
_obj:                             ; static: obj
          .RES.L      2
```



#pragma pack 1 指定ソースコード

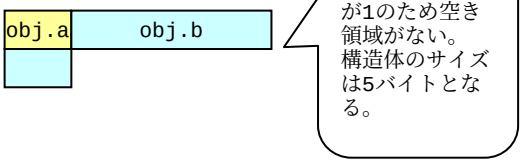
```
#pragma pack 1
struct {
    char a;
    int b;
} obj;

int func(void)
{
    return obj.b;
}
```

アセンブラ展開コード

```
_func:
          MOV.L       L11+2,R3    ; H'00000001+_obj
          MOV.B       @R3+,R2     ; (part of)obj.b
          MOV.B       @R3+,R7     ; (part of)obj.b
          SHLL8       R2
          MOV.B       @R3+,R5     ; (part of)obj.b
          EXTU.B      R7,R7
          OR          R2,R7
          SHLL8       R7
          EXTU.B      R5,R4
          MOV.B       @R3,R3      ; (part of)obj.b
          OR          R7,R4
          SHLL8       R4
          EXTU.B      R3,R0
          RTS
          OR          R4,R0
L11:
          .RES.W      1
          .DATA.L     H'00000001+_obj
          .SECTION    B,DATA,ALIGN=4
_obj:                             ; static: obj
          .RES.B      5
```

メンバへのアクセスがバイトアクセスとなる。



【注意事項】

ルネサス統合開発環境で作成された iodefine.h を使用する場合、#pragma もしくはオプションでアライメント数を 1 にすると、I/O レジスタ用構造体のメンバが示すアドレスが正しいアドレスを指しません。iodefine.h の先頭に、#pragma pack 4 を、iodefine.h の最後に、#pragma unpack を記述してください。

```
/* iodefine.h */
#pragma pack 4

    /* 略 */

#pragma unpack
```

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。