RX ファミリ用 C/C++ コンパイラパッケージ

RJJ06J0094-0100
Rev.1.00
2010.04.20

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

本ドキュメントでは、RX ファミリ用 C/C++ コンパイラ V.1 における効率の良いプログラミング技法を紹介します。

目次

1. はじめに

RX ファミリ用 C/C++ コンパイラは最適化を行っていますが、プログラミングの工夫により一層の性能向上が可能です。

本ドキュメントでは、効果的なプログラム作成のために、ユーザに試みて頂きたい手法を紹介します。

プログラムの評価基準には、実行速度が速いこととサイズが小さいことの 2 種類があります。効果的なプログラムを作成するための原則を以下に示します。

(1) 実行速度向上の原則

実行頻度の高い文、複雑な文で実行速度は決まるので、これらの処理を把握して、重点的に改良してください。

(2) サイズ縮小の原則

プログラムサイズ縮小のためには、類似処理の共通化、複雑な関数の見直しを行ってください。

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、割り込みなどの要因によって変化します。本ドキュメントで紹介するさまざまな手法は実機で実際に実行し、効果を確認してから適用してください。

本ドキュメントのアセンブリ言語展開コードは RX ファミリ用 C/C++ コンパイラを用いて

ccrx△〈C 言語ファイル〉△-output=src△-cpu=rx600

のコマンドラインで取得しています。なお、アセンブリ言語展開コードは前後に存在するプログラムや、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。

本ドキュメントに記載の実行速度は、コンパイラパッケージに付属のシミュレータデバッガを用いて測定しています。また、外部メモリへのアクセスサイクル数は 1 として測定しています。測定結果は参考値としてご参照ください。

効率の良いプログラミング技法の一覧を表 1-1に示します。

表 1-1 効率の良いプログラミング技法一覧

項番	項目	ROM 効率	RAM 効率	実行速度	参照
1	データの構造	○	-	○	2.1
2	変数とconst 型	-	○	-	2.2
3	局所変数と大域変数	○	-	○	2.3
4	構造体宣言のメンバオフセット	○	-	-	2.4
5	ビットフィールドの割り付け	○	-	-	2.5
6	ループ制御変数	×	-	○	2.6
7	ベースレジスタ指定時の外部変数アクセス最適化	○	-	○	2.7
8	外部変数アクセス最適化時のリンカのセクションアドレス指定順	○	-	○	2.8
9	関数のモジュール化	○	-	-	3.1
10	関数のインタフェース	-	○	○	3.2
11	ループ回数の削減	×	-	○	4.1
12	テーブルの活用	×	-	○	4.2
13	分岐	-	-	○	5
14	割り込み	○	-	○	6
15	インライン展開	-	-	○	7

[注意]
表中の○、×は以下の意味を示します。
○…性能向上に効果あり
×…性能低下の可能性あり

2. データ指定

データに関して考慮すべき点を表 2-1に示します。

表 2-1 データ指定における注意事項

項目	注意点	参照
データ型指定子、型、修飾子	• **データサイズを縮小しようとすると、プログラムサイズが増大する場合があります。データは用途を考えて型宣言してください。** • **符号あり／なしによりプログラムサイズが変わることがあるので、選択時に注意してください。** • **プログラム内で値が不変な初期化データの場合、**const **演算子を付けておくと使用メモリ量の節約になります。**	2.2
データの整合	• **データ領域に無駄なエリアを生じないように割り付けてください。**	
構造体の定義／参照	• **頻繁に参照／変更するデータは構造体にして、ポインタ変数を用いることによりプログラムサイズを縮小できる場合があります。** • **ビットフィールドを使用すると、データサイズを縮小できます。**	2.1
内蔵 ROM/RAM **の活用**	• **外部メモリに比べ内蔵メモリへのアクセスは速いので、共通変数は内蔵メモリへ格納するようにしてください。**	-

2.1 データの構造

■ポイント

関連するデータを構造体で宣言すると、実行速度を向上できる場合があります。

■説明

関連するデータを同一関数の中で何度も参照している場合、構造体を用いると相対アクセスを利用したコードが生成され易くなり、効率向上が期待できます。また、引数として渡す場合も効率が向上します。相対アクセスにはアクセス範囲に制限があるため、頻繁にアクセスするデータは構造体の先頭に集めると効果的です。

データを構造化すると、データの表現を変更するようなチューニングが容易になります。

■使用例

変数 a, b, c に数値を代入します。

改善前ソースコード

```
int a, b, c;

void func()
{
        a = 1;
        b = 2;
        c = 3;
}
```

改善後ソースコード

```
struct s{
        int a;
        int b;
        int c;
} s1;

void func()
{
        register struct s *p=&s1;
        p->a = 1;
        p->b = 2;
```

```
                                        p->c = 3;
                                    }
```

改善前アセンブリ展開コード

```
_ func:
        MOV.L     #_a,R4
        MOV.L     #00000001H,[R4]
        MOV.L     #_b,R4
        MOV.L     #00000002H,[R4]
        MOV.L     #_c,R4
        MOV.L     #00000003H,[R4]
        RTS
```

改善後アセンブリ展開コード

```
_func:
        MOV.L     #_s1,R5
        MOV.L     #00000001H,[R5]
        MOV.L     #00000002H,04H[R5]
        MOV.L     #00000003H,08H[R5]
        RTS
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
RX610	28	15	9	7

2.2　変数と const 型

■ポイント

値を変更しない変数は、const 型で宣言してください。

■説明

初期値のある変数は、通常、起動時に ROM エリアから RAM エリアに転送して、RAM エリアを使って処理を行います。このため、プログラム内で値が不変な初期化データの場合、確保した RAM エリアが無駄になります。初期化データに const 演算子を付けておくと、起動時の RAM エリアへの転送が抑止され、使用メモリ量の節約になります。

また、初期値は変更しない、というルールでプログラムを作成すると、ROM 化が容易になります。

■使用例

5 個の初期化データを設定します。

改善前ソースコード

```
char a[] =
    {1, 2, 3, 4, 5};
```

初期値をROMからRAMへ転送して処理を行います

改善後ソースコード

```
const char a[] =
    {1, 2, 3, 4, 5};
```

ROM 上の初期値を使用して処理を行います

2.3　局所変数と大域変数

■ポイント

一時変数、ループのカウンタなど、局所的に用いる変数は、関数の中で局所変数として宣言すると実行速度を向上できます。

■説明

局所変数として使用できるものは、大域変数として宣言しないで必ず局所変数として宣言してください。大域変数は、関数呼び出しやポインタ操作によって値が変化してしまう可能性があるため、最適化の効率が悪くなります。

局所変数を使用すると次の利点があります。

a. アクセスコストが安い。
b. レジスタに割り付けられる可能性がある。
c. 最適化の効率が良い

■使用例

一時変数に大域変数を使った場合(改善前) と 局所変数を使った場合(改善後)。

改善前ソースコード

```
int tmp;

void func(int* a, int* b)
{
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

改善前アセンブリ展開コード

```
__func:
          MOV.L      #_tmp,R4
          MOV.L      [R1],[R4]
          MOV.L      [R2],[R1]
          MOV.L      [R4],[R2]
          RTS
```

改善後ソースコード

```
void func(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

改善後アセンブリ展開コード

```
__func:
          MOV.L      [R1],R5
          MOV.L      [R2],[R1]
          MOV.L      R5,[R2]
          RTS
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
RX610	13	7	13	9

2.4　構造体宣言のメンバオフセット

■ポイント

構造体の中でよく使用するメンバは、先頭に宣言するようにすればサイズが向上します。

■説明

構造体メンバは、構造体アドレスにオフセットを加算してアクセスします。オフセットを小さくするとサイズが有利になるので、よく使用するメンバを先頭に宣言するようにしてください。

もっとも効果的なのは、signed char, unsigned char 型で先頭から 32 byte 未満, short, unsigned short 型で先頭から 64 byte 未満, int, unsigned, long, unsigned long 型で先頭から 128 byte 未満です。

■使用例

以下の例は、構造体のオフセットによってコードが変わる例を示します。

改善前ソースコード

```
struct str {
        long L1[8];
        char C1;
};

struct str STR1;
char x;

void func()
{
        x = STR1.C1;
}
```

改善前アセンブリ展開コード

```
_func:
            MOV.L       #_STR1,R4
            MOVU.B      20H[R4],R5
            MOV.L       #_x,R4
            MOV.B       R5,[R4]
            RTS
```

改善後ソースコード

```
struct str {
        char C1;
        long L1[8];
};

struct str STR1;
char x;

void func()
{
        x = STR1.C1;
}
```

改善後アセンブリ展開コード

```
_func:
            MOV.L       #_STR1,R4
            MOVU.B      [R4],R5
            MOV.L       #_x,R4
            MOV.B       R5,[R4]
            RTS
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [bte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
RX610	18	17	8	8

■注意事項

構造体を定義する際には、境界調整数を意識してメンバの宣言をおこなってください。

構造体の境界調整数は構造体内の最も大きな境界調整値に合わせられ、構造体のサイズは境界調整数の倍数となります。その為、構造体の末尾が構造体自身の境界調整数と合わない場合に、次の境界調整を保証するために生成される、未使用領域もサイズに含めてしまいます。

改善前ソースコード

```
/* 最大メンバがint型の為、協会調整数ば4 */
struct str {
       char C1; /* 1byte + 境界調整分 3byte */
       long L1; /* 4byte */
       char C2; /* 1byte */
       char C3; /* 1byte */
       char C4; /* 1byte + 境界調整分 1byte*/
}STR1;
```

改善前strサイズ

```
          .SECTION    B,DATA,ALIGN=4
          .glb        _STR1
_STR1:                          ; static: STR1
          .blkl       3
```

改善後ソースコード

```
/* 最大メンバがint型の為、協会調整数ば4 */
struct str {
       char C1; /* 1byte */
       char C2; /* 1byte */
       char C3; /* 1byte */
       char C4; /* 1byte */
       long L1; /* 4byte */
}STR1;
```

改善後strサイズ

```
          .SECTION    B,DATA,ALIGN=4
          .glb        _STR1
_STR1:                          ; static: STR1
          .blkl       2
```

2.5 ビットフィールドの割り付け

■ポイント

ビットフィールドで、連続して設定されるものは、同じ構造体内に割り付けるようにしてください。

■説明

異なるビットフィールドのメンバを設定するためには、そのたびにビットフィールドを含むデータにアクセスしなければなりません。関連するビットフィールドを同じ構造体内にまとめて割り付けることによって、このアクセスを一度で済ますことができます。

■使用例

同じ構造体に関連するビットフィールドを割り付けることによってサイズが改善する例を示します。

改善前ソースコード

```
struct str
{
        int flag1:1;
}b1,b2,b3;



void func()
{
        b1.flag1 = 1;
        b2.flag1 = 1;
        b3.flag1 = 1;
}
```

改善前アセンブリ展開コード

```
_func:
            MOV.L       #_b1,R5
            BSET        #00H,[R5]
            MOV.L       #_b2,R5
            BSET        #00H,[R5]
            MOV.L       #_b3,R5
            BSET        #00H,[R5]
            RTS
```

改善後ソースコード

```
struct str
{
        int flag1:1;
        int flag2:1;
        int flag3:1;
}a1;

void func()
{
        a1.flag1 = 1;
        a1.flag2 = 1;
        a1.flag3 = 1;
}
```

改善後アセンブリ展開コード

```
_func:
            MOV.L       #_a1,R4
            MOVU.B      [R4],R5
            OR          #07H,R5
            MOV.B       R5,[R4]
            RTS
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
RX610	25	13	14	9

2.6 ループ制御変数

■ポイント

ループ制御変数を 4 バイト整数型(signed long/unsigned long)に変更すると、ループ展開最適化が適用され易くなり、実行速度の向上が期待できます。

■説明

ループ終了条件の判定で、サイズの違いによりループ制御変数がその比較対象のデータを表現できない可能性のある場合はループ展開最適化がかかりません。たとえばループ制御変数が signed char で比較対象のデータが signed long の場合はループ展開最適化がかかりません。そのため、signed char、signed short、に比べ、signed long の方がループ展開最適化が適用され易いです。ループ展開最適化を活用したい場合はループ制御変数を 4 バイト整数型としてください。

■使用例

改善前ソースコード

```
signed long array_size=16;
signed char array[16];

void func()
{
        signed char i;
        for(i=0;i<array_size;i++)
        {
                array[i]=0;
        }
}
```

改善後ソースコード

```
signed long array_size=16;
signed char array[16];

void func()
{
        signed long i;
        for(i=0;i<array_size;i++)
        {
                array[i]=0;
        }
}
```

改善前アセンブリ展開コード

<loop=2 指定時>

```
_func:
            MOV.L       #_array_size,R4
            MOV.L       [R4],R2
            MOV.L       #00000000H,R5
            BRA         L11
L12:
            MOV.L       #_array,R4
            MOV.L       #00000000H,R3
            MOV.B       R3,[R5,R4]
            ADD         #01H,R5
L11:
            MOV.B       R5,R5
```

改善後アセンブリ展開コード

<loop=2 指定時>

```
_func:
            MOV.L       #_array_size,R5
            MOV.L       [R5],R2
            MOV.L       #00000000H,R4
            ADD         #0FFFFFFFFH,R2,R3
            CMP         R3,R2
            BLE         L12
L11:
            MOV.L       #_array,R1
            MOV.L       R1,R5
            BRA         L13
L14:
```

```
        CMP     R2,R5
        BLT     L12
L13:
        RTS
```

```
        MOV.W   #0000H,[R5]
        ADD     #02H,R5
        ADD     #02H,R4
L13:
        CMP     R3,R4
        BLT     L14
L15:
        CMP     R2,R4
        BGE     L17
L16:
        MOV.L   #00000000H,R5
        MOV.B   R5,[R4,R1]
        RTS
L12:
        MOV.L   #_array,R5
        MOV.L   #00000000H,R3
L19:
        CMP     R2,R4
        BGE     L17
L20:
        MOV.B   R3,[R5+]
        ADD     #01H,R4
        BRA     L19
L17:
        RTS
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度 [Cycle]	
	改善前	改善後	改善前	改善後
RX610	32	67	171	82

2.7　ベースレジスタ指定時の外部変数アクセス最適化

■ポイント

プロジェクト全体で、特定のレジスタを ROM/RAM セクションにアクセスする時のベースレジスタとして使用する場合、外部変数アクセス最適化と組み合わせることでコードサイズを小さくすることが可能です。

■説明

RAM セクションのベースレジスタに R13 を指定した場合、RAM セクションへのアクセスが、R13 レジスタ相対となります。更にモジュール間の外部変数サクセス最適化を有効にした場合、R13 レジスタ相対値が最適化されて、8bit 範囲以下の値であれば、命令サイズが小さくなる場合があります。ベースレジスタの指定は HEW のメニュー ビルド->RX Standard ToolChain->CPU->ベースレジスタで設定可能です。（図 1）

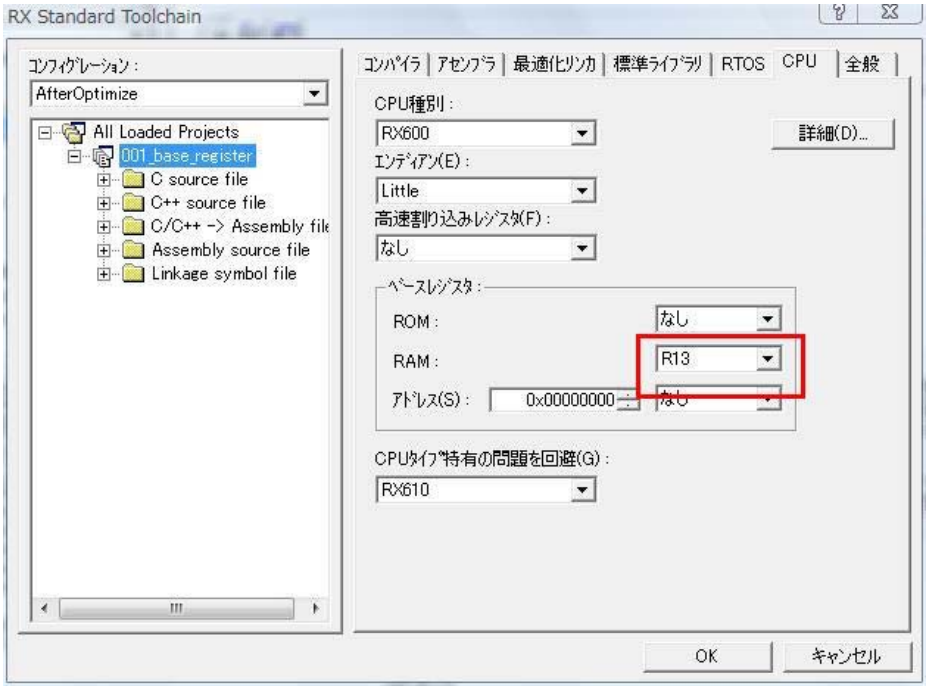

図 1 ベースレジスタ設定

■使用例

改善前ソースコード

```
int a;
int b;
int c;
int d;

void func()
{
        a=0;
        b=1;
        c=2;
        d=3;
}
```

改善前アセンブリ展開コード

```
_func:
            MOV.L       #_a,R4
            MOV.L       #00000000H,[R4]
            MOV.L       #_b,R4
            MOV.L       #00000001H,[R4]
            MOV.L       #_c,R4
            MOV.L       #00000002H,[R4]
```

改善後ソースコード

```
int a;
int b;
int c;
int d;

void func()
{
        a=0;
        b=1;
        c=2;
        d=3;
}
```

改善後アセンブリ展開コード

```
_func:
            MOV.L       #00000000H,_a-__RAM_TOP:16[R13]
            MOV.L       #00000001H,_b-__RAM_TOP:16[R13]
            MOV.L       #00000002H,_c-__RAM_TOP:16[R13]
            MOV.L       #00000003H,_d-__RAM_TOP:16[R13]
            RTS
```

```
        MOV.L     #_d,R4
        MOV.L     #00000003H,[R4]
        RTS
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
RX610	14	10	11	7

2.8　外部変数アクセス最適化時のリンカのセクションアドレス指定順

■ポイント

外部変数アクセス最適化を有効とする場合、リンカでのセクション割り付け順を変更することでさらにコードサイズを小さくできる場合があります。

■説明

レジスタ相対形式でメモリにアクセスする命令では、ディスプレースメント値が小さいほうが、命令サイズが小さくなります。以下の指標を参考にリンカでのセクション割り付け順を変更するとコードサイズを改善できる場合があります。

・関数内でのアクセス回数の多い外部変数のセクションを前にする。
・型サイズの小さい外部変数のセクションを前にする。

但し、外部変数アクセス最適化は、コンパイルが 2 度実行されるのでビルド時間は長くなります。

■使用例

改善前ソースコード

```
/* D_1 セクション */
char  d11=0, d12=0, d13=0, d14=0;
/* D_2 セクション */
short d21=0, d22=0, d23=0, d24=0, dmy2[12]={0};
/* D   セクション */
int   d41=0, d42=0, d43=0, d44=0, dmy4[60]={0};

void func(int a){
       d11 = a;
       d12 = a;
       d13 = a;
       d14 = a;
       d21 = a;
       d22 = a;
```

改善後ソースコード

```
/* D_1 セクション */
char  d11=0, d12=0, d13=0, d14=0;
/* D_2 セクション */
short d21=0, d22=0, d23=0, d24=0, dmy2[12]={0};
/* D   セクション */
int   d41=0, d42=0, d43=0, d44=0, dmy4[60]={0};

void func(int a){
       d11 = a;
       d12 = a;
       d13 = a;
       d14 = a;
       d21 = a;
       d22 = a;
```

```
        d23 = a;
        d24 = a;
        d41 = a;
        d42 = a;
        d43 = a;
        d44 = a;
}
```

改善前アセンブリ展開コード

<セクションの割り付け順をD,D_2,D_1　またはD*とした場合>

```
_func:
        MOV.L   #_d41,R4
        MOV.B   R1,0120H[R4]
        MOV.B   R1,0121H[R4]
        MOV.B   R1,0122H[R4]
        MOV.B   R1,0123H[R4]
        MOV.W   R1,0100H[R4]
        MOV.W   R1,0102H[R4]
        MOV.W   R1,0104H[R4]
        MOV.W   R1,0106H[R4]
        MOV.L   R1,[R4]
        MOV.L   R1,04H[R4]
        MOV.L   R1,08H[R4]
        MOV.L   R1,0CH[R4]
        RTS
```

```
        d23 = a;
        d24 = a;
        d41 = a;
        d42 = a;
        d43 = a;
        d44 = a;
}
```

改善後アセンブリ展開コード

<セクションの割り付け順をD_1,D_2,Dとした場合>

```
_func:
        MOV.L   #_d11,R4
        MOV.B   R1,[R4]
        MOV.B   R1,01H[R4]
        MOV.B   R1,02H[R4]
        MOV.B   R1,03H[R4]
        MOV.W   R1,04H[R4]
        MOV.W   R1,06H[R4]
        MOV.W   R1,08H[R4]
        MOV.W   R1,0AH[R4]
        MOV.L   R1,24H[R4]
        MOV.L   R1,28H[R4]
        MOV.L   R1,2CH[R4]
        MOV.L   R1,30H[R4]
        RTS
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
RX610	43	31	20	18

3. 関数呼び出し

関数呼び出しに関して考慮すべき事項を表 3-1に示します。

表 3-1 関数呼び出しにおける注意事項

項目	注意点	参照
関数位置	• 関連の深い関数は１ファイルにまとめてください。	3.1
インタフェース	• 引数がすべてレジスタに割り付くように（４個まで）引数の数を厳選してください。 • 引数が多い場合、構造体にしてポインタで渡してください。	3.2
マクロへの置換	• 関数呼び出しが多数ある場合、マクロにすれば実行速度を向上できます。ただし、マクロにするとプログラムサイズが増大するので、状況により選択してください。	-

3.1 関数のモジュール化

■ポイント

関連の深い関数は 1 ファイルにまとめることによりサイズを向上できます。

■説明

異なるファイルにある関数を呼び出す場合、4 バイトの BSR 命令に展開されますが、同一ファイル内の関数呼び出しでは、呼び出し範囲が近いと 3 バイトの BSR 命令に展開され、コンパクトなオブジェクトが生成されます。

また、モジュール化によって、チューンアップ時の修正が容易になります。

■使用例

関数 f から関数 g を呼び出します。

改善前ソースコード

```
extern void sub(void);



int func()
{
        sub();
        return(0);
}
```

改善前アセンブリ展開コード

```
  _func:
            BSR         _sub   ;length  A
            MOV.L       #00000000H,R1
            RTS
```

改善後ソースコード

```
void sub(void)
{
}
int func()
{
        sub();
        return (0);
}
```

改善後アセンブリ展開コード

```
  _func:
            BSR         _sub   ;length  W
            MOV.L       #00000000H,R1
            RTS
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
RX610	7	6	9	9

3.2　関数のインタフェース

■ポイント

関数の引数を工夫することにより RAM 容量を削減でき、実行速度も向上できます。
コンパイラマニュアル 8.2 関数呼び出しのインタフェースを参照してください。

■説明

引数がすべてレジスタに乗るように（4 個まで）引数の数を厳選してください。引数が多い場合は、構造体にしてポインタで渡してください。もし、構造体のポインタではなく、構造体そのものを受け渡すとレジスタに乗らない場合があります。引数がレジスタに乗れば、呼び出し、関数の出入り口の処理が簡単になります。また、スタック領域も節約できます。
なお、レジスタは R1～R4 が引数用です。

■使用例

関数 f の引数が引数用レジスタ個数よりも 4 個多くあります。

改善前ソースコード

```
void call_func ()
{
        func(1,2,3,4,5,6,7,8);
}
```

改善後ソースコード

```
struct str{
        char a;
        char b;
        char c;
        char d;
        char e;
        char f;
        char g;
        char h;
};

struct str arg = {1,2,3,4,5,6,7,8};

void call_func ()
{
        func(&arg);
}
```

改善前アセンブリ展開コード

```
_call_func:
            SUB         #04H,R0
            MOV.L       #08070605H,[R0]
            MOV.L       #00000004H,R4
            MOV.L       #00000003H,R3
            MOV.L       #00000002H,R2
            MOV.L       #00000001H,R1
            BSR         _func
            ADD         #04H,R0
            RTS
```

改善後アセンブリ展開コード

```
_ call_func:
            MOV.L       #_arg,R1
            BRA         _func
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
RX610	16	8	16	4

4. 演算方法

演算方式に関して考慮すべき事項を表 4-1に示します。

表 4-1 演算方式における注意事項

項目	注意点	参照
ループ回数の削減	■ ループ条件が同一または類似しているループ文のマージを検討してください。 • ループの展開を試みてください。	1.1
高速なアルゴリズムの利用	■ 配列におけるクイックソートのような計算時間が少なくてすむアルゴリズムを検討してください。	-
テーブルの活用	■ switch 文の各 case の処理がほぼ同じ場合は、テーブルを使用できないか検討してください。 ■ あらかじめ演算した結果をテーブルに代入しておき、演算結果が必要になった際、テーブルの値を参照することで実行速度を向上させる手法があります。ただし、この手法は、ROM 容量の増大になるので、必要実行速度と余裕 ROM 要領との兼ね合いで選択してください。	1.1
条件式	定数との比較は 0 で行うと効率の良いコードが生成されます。	-

4.1　ループ回数の削減

■ポイント

ループを展開すると、実行速度は大幅に向上できます。

■説明

ループの展開は特に内側のループが有効です。ループの展開によりプログラムサイズは増大するので、プログラムサイズを犠牲にしても実行速度を向上させたい場合に適用してください。

■使用例

配列 a[ ]を初期化します。

改善前ソースコード

```
extern int a[100];
void func()
{
        int i;
        for ( i = 0; i < 100; i++) {
                a[i] = 0;
        }
}
```

改善前アセンブリ展開コード

```
_func:
            MOV.L       #00000064H,R4
            MOV.L       #_a,R5
            MOV.L       #00000000H,R3
L11:
            MOV.L       R3,[R5+]
            SUB         #01H,R4
            BNE         L11
L12:
            RTS
```

改善後ソースコード

```
extern int a[100];
void func()
{
        int i;
        for ( i = 0; i < 100; i+=2)
        {
                a[i] = 0;
                a[i+1] = 0;
        }
}
```

改善前アセンブリ展開コード

```
_func:
            MOV.L       #00000032H,R4
            MOV.L       #_a,R5
L11:
            MOV.L       #00000000H,[R5]
            MOV.L       #00000000H,04H[R5]
            ADD         #08H,R5
            SUB         #01H,R4
            BNE         L11
L12:
            RTS
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
RX610	19	22	504	353

■補足

loop オプションを指定すると、ループ展開最適化が行われます。使用例の改善前ソースコードに loop オプションを指定しコンパイルすると、改善後ソースコードのアセンブリ展開コードと同じアセンブリ展開コードが出力されます。

改善前ソースコード

```
extern int a[100];
void func()
{
        int i;
        for ( i = 0; i < 100; i++) {
                a[i] = 0;
        }
}
```

改善前アセンブリ展開コード

```
<loop=2 指定時>

_func:
            MOV.L       #00000032H,R4
            MOV.L       #_a,R5
L11:
            MOV.L       #00000000H,[R5]
            MOV.L       #00000000H,04H[R5]
            ADD         #08H,R5
            SUB         #01H,R4
            BNE         L11
L12:
            RTS
```

改善後ソースコード

```
extern int a[100];
void func()
{
        int i;
        for ( i = 0; i < 100; i+=2)
        {
                a[i] = 0;
                a[i+1] = 0;
        }
}
```

改善後アセンブリ展開コード

```
_func:
            MOV.L       #00000032H,R4
            MOV.L       #_a,R5
L11:
            MOV.L       #00000000H,[R5]
            MOV.L       #00000000H,04H[R5]
            ADD         #08H,R5
            SUB         #01H,R4
            BNE         L11
L12:
            RTS
```

4.2　テーブルの活用

■ポイント

switch 文による分岐の代わりにテーブルを用いることで実行速度を向上できます。

■説明

switch 文の各 case の処理がほぼ同じ場合は、テーブルを使用できないか検討してください。

■使用例

変数 i の値により変数 ch に代入する文字定数を変えます。

改善前ソースコード

```
char func(int i)
{
        char ch;
        switch (i) {
        case 0:
                ch = 'a'; break;
        case 1:
                ch = 'x'; break;
        case 2:
                ch = 'b'; break;
        }
        return (ch);
}
```

改善前アセンブリ展開コード

```
_func:
            CMP         #00H,R1
            BEQ         L17
L16:
            CMP         #01H,R1
            BEQ         L19
L18:
            CMP         #02H,R1
            BEQ         L20
            BRA         L21
L12:
L17:
            MOV.L       #00000061H,R1
            BRA         L21
L13:
L19:
```

改善後ソースコード

```
char chbuf[] = { 'a', 'x', 'b' };

char func(int i)
{
        return (chbuf[i]);
}
```

改善後アセンブリ展開コード

```
_func:
            MOV.L       #_chbuf,R4
            MOVU.B      [R1,R4],R1
            RTS
```

```
            MOV.L       #00000078H,R1
            BRA         L21
L14:
L20:
            MOV.L       #00000062H,R1
L11:
L21:
            MOVU.B      R1,R1
            RTS
```

■改善前後のコードサイズと実行速度

CPU **種別**	**コードサイズ** [byte]		**実行速度**[Cycle]	
	改善前	**改善後**	**改善前**	**改善後**
RX610	28	10	13	6

[**注**] i=2 **の場合**

5. 分岐

■ポイント

分岐するケースの位置を変更することで実行速度が向上します。

■説明

else if 文のように上から順に比較をする場合、場合分けが増えると末端のケースの実行速度は低下します。頻繁に分岐するケースは先頭近くに配置してください。

■使用例

引数の値によりリターン値を変えます。

改善前ソースコード

```
int func(int a)
{
    if (a==1)
        a = 2;
    else if (a==2)
        a = 4;
    else if (a==3)
        a = 8;
    else
        a = 0;
    return (a);
}
```

改善前アセンブリ展開コード

```
_func:
            CMP         #01H,R1
            BEQ         L11
L12:
            CMP         #02H,R1
            BNE         L14
L13:
            MOV.L       #00000004H,R1
            RTS
L14:
            CMP         #03H,R1
            BNE         L17
L16:
            MOV.L       #00000008H,R1
            RTS
L17:
```

改善後ソースコード

```
int func(int a)
{
    if (a==3)
        a = 8;
    else if (a==2)
        a = 4;
    else if (a==1)
        a = 2;
    else
        a = 0;
    return (a);
}
```

改善後アセンブリ展開コード

```
_func:
            CMP         #03H,R1
            BEQ         L11
L12:
            CMP         #02H,R1
            BNE         L14
L13:
            MOV.L       #00000004H,R1
            RTS
L14:
            CMP         #01H,R1
            BNE         L17
L16:
            MOV.L       #00000002H,R1
            RTS
L17:
```

```
        MOV.L   #00000000H,R1
        RTS
L11:
        MOV.L   #00000002H,R1
        RTS
```

```
        MOV.L   #00000000H,R1
        RTS
L11:
        MOV.L   #00000008H,R1
        RTS
```

■改善前後のコードサイズと実行速度

CPU **種別**	**コードサイズ** [byte]		**実行速度**[Cycle]	
	改善前	**改善後**	**改善前**	**改善後**
RX610	22	22	11	7

[**注**]:a=3 **の場合**

6. 割り込み

■ポイント

高速割り込み機能を利用することで、割り込み応答時間を短縮できます。

■説明

割り込み処理の前後では多くのレジスタの退避・回復が発生し、期待する割り込み応答時間を得られない場合があります。その場合は、高速割り込み指定(fint)、ならびに fint_register オプションを利用することで、レジスタの退避・回復を抑えることができ、割り込み応答時間の短縮を図ることができます。

ただし、fint_register オプションを利用すると、他の関数での使用可能なレジスタが減るため、プログラム全体での効率が低下する場合があるのでご注意ください。

■使用例

改善前ソースコード

```
#pragma interrupt int_func

volatile int count;

void int_func()
{
       count++;
}
```

改善前アセンブリ展開コード

```
_int_func:
          PUSHM       R4-R5
          MOV.L       #_count,R4
          MOV.L       [R4],R5
          ADD         #01H,R5
          MOV.L       R5,[R4]
          POPM        R4-R5
          RTE
```

改善後ソースコード

```
#pragma interrupt int_func(fint)

volatile int count;

void int_func()
{
       count++;
}
```

改善後アセンブリ展開コード

<fint_register=2 オプション指定時>

```
_int_func:
          MOV.L       #_count,R12
          MOV.L       [R12],R13
          ADD         #01H,R13
          MOV.L       R13,[R12]
          RTFI
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
RX610	18	14	23	14

7. インライン展開

■ポイント

頻繁に呼び出される関数をインライン展開すると実行速度を向上できます。

■説明

頻繁に呼び出される関数をインライン展開することにより、実行速度の向上が図れます。特にループ内で呼ばれる関数などを展開すると大きな効果を得られる場合もあります。しかし、インライン展開をした場合、プログラムサイズが増大する傾向にありますので、プログラムサイズを犠牲にしても実行速度を向上させたい場合に適用してください。

■使用例

配列 a と配列 b の要素を交換します。

改善前ソースコード

```
int x[10], y[10];
static void sub(int *a, int *b, int i)
{
        int temp;
        temp = a[i];
        a[i] = b[i];
        b[i] = temp;
}
void func()
{
        int i;
        for (i=0;i<10;i++)
          sub(x, y, i);
}
```

改善前アセンブリ展開コード

```
__$sub:
           SHLL        #02H,R3
           ADD         R3,R1
           MOV.L       [R1],R5
           ADD         R3,R2
           MOV.L       [R2],[R1]
           MOV.L       R5,[R2]
           RTS
```

改善後ソースコード

```
int x[10], y[10];
#pragma inline (sub)
static void sub(int *a, int *b, int i)
{
        int temp;
        temp = a[i];
        a[i] = b[i];
        b[i] = temp;
}
void  func()
{
        int i;
        for (i=0;i<10;i++)
          sub(x, y, i);
}
```

改善後アセンブリ展開コード

```
; インライン展開により
; _subのコードが削減されている
```

```
_func:
        PUSHM   R6-R8
        MOV.L   #00000000H,R6
        MOV.L   #_x,R7
        MOV.L   #_y,R8
L12:
        MOV.L   R6,R3
        MOV.L   R7,R1
        MOV.L   R8,R2
        ADD     #01H,R6
        BSR     __$sub
        CMP     #0AH,R6
        BLT     L12
L13:
        RTSD    #0CH,R6-R8
```

```
_func:
        MOV.L   #0000000AH,R1
        MOV.L   #_y,R2
        MOV.L   #_x,R3
L11:
        MOV.L   [R3],R4
        MOV.L   [R2],R5
        MOV.L   R4,[R2+]
        MOV.L   R5,[R3+]
        SUB     #01H,R1
        BNE     L11
L12:
        RTS
```

■改善前後のコードサイズと実行速度

CPU 種別	コードサイズ [byte]		実行速度[Cycle]	
	改善前	改善後	改善前	改善後
RX610	47	29	119	84

ホームページとサポート窓口

- ルネサス エレクトロニクスホームページ
  http://japan.renesas.com/
- お問合せ先
  http://japan.renesas.com/inquiry

改訂記録

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2010.04.20	—	初版発行
-	-	-	-
-	-	-	-
-	-	-	-
-	-	-	-
-	-	-	-
-	-	-	-
-	-	-	-

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。

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口 http://www.renesas.com

※営業お問合せ窓口の住所・電話番号は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス販売株式会社　〒100-0004　千代田区大手町2-6-2（日本ビル） (03)5201-5307

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：http://japan.renesas.com/inquiry

Colophon 1.0