RL78 ファミリ用 C コンパイラパッケージ(CC-RL)

アプリケーションガイド　プログラミング・テクニック編

R20AN0529JJ0100
Rev.1.0
2018.11.26

要旨

コードサイズ・実行速度・ROM サイズに効果的なプログラミング方法を説明します。

動作確認リビジョン

RL78 ファミリ　CC-RL V1.07.00

目次

1.　はじめに

コードサイズ・実行速度・ROM サイズに効果的なプログラミング方法として、以下の 3 つの項目に分けてそれぞれ説明します。

- オプション
- 拡張言語
- コーディングテクニック

本アプリケーションノート記載の測定結果、アセンブリ言語展開コードは CC-RL V1.07 を用いて取得しています。-cpu オプションの指定値は –cpu=S3 です。

なお、これらのプログラミング方法の効果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。

2.　オプション

CC-RL のオプション指定によるコードサイズ・ROM サイズ・実行速度への影響を示します。なお、効果の程度はソースプログラムの内容によって異なります。

2.1　　コンパイル・オプション

○…改善する　×…劣化する　△…改善することもあれば劣化することもある　―…変化しない

( )…デフォルト設定

オプション	コードサイズ	ROM サイズ	クロック数	備考
-memory_model	△	△	△	
-far_rom	×	×	×	
-Onothing	×	―	×	
-Osize	○	―	△	コードサイズを優先して最適化を実施します。反面、実行速度は劣化する場合があります。
-Ospeed	△	―	○	実行速度を優先して最適化を実施します。反面、コードサイズは劣化する場合があります。
-Ounroll	×	―	○	効果はパラメータに依存します。
-Odelete_static_func	(○)	―	―	
-Oinline_level	×	―	○	効果はパラメータに依存します。
-Oinline_size	×	―	○	効果はパラメータに依存します。
-Opipeline【V1.03 以降】	―	―	(○)	
-Otail_call	(○)	―	(○)	
-Omerge_files	○	―	○	
-Ointermodule	○	―	○	
-Owhole_program	○	―	○	
-Oalias	○	―	○	
-Osame_code【V1.02 以降】	○	―	×	
-goptimize	○	○	―	本オプションを指定したファイルは、リンク時のモジュール間最適化の対象になります。 リンク時の最適化については 2.3 リンク・オプションを参照ください。
-dbl_size=8	×	×	×	
-signed_char	×	―	×	
-signed_bitfield	×	―	×	
-switch	△	△	△	
-merge_string	―	○	―	
-pack	×	○	×	
-stack_protector/ -stack_protector_all 【Professional 版のみ】 【V1.02 以降】	×	―	×	関数の入口・出口にスタック破壊検出コードを生成します。検出コードの分、コードサイズと実行速度は劣化します。
-control_flow_integrity 【Professional 版のみ】 【V1.06 以降】	×	×	×	呼び出し先関数のチェックコードを生成します。チェック用にコードとデータを生成するため、コー

				ドサイズと実行速度、ROM サイズは劣化します。
-unaligned_pointer_for_ca78k0r 【V1.06 以降】	×	―	×	

2.1.1　-memory_model

コンパイル時のメモリ・モデルの種類を指定します。

-memory_model=small の場合：変数、関数共にデフォルトを near とします。

-memory_model=medium の場合：変数のデフォルトを near、関数のデフォルトを far とします。

far 領域の方が大きい領域を扱えますが、関数呼び出しのサイズが大きくなります。尚、オプション省略時は-cpu オプションの設定により以下の通りとなります。

・-cpu=S1 の場合：-memory_model=small

・-cpu=S2/S3 の場合：-memory_model=medium

C ソース

```
int val;
#pragma noinline func1
void func1()
{
  ++val;
}

#pragma noinline func2
void (*func2(void)) ()
{
  return func1;
}

void main(void)
{
  func2()();
}
```

	-memory_model=small	-memory_model=medium
コードサイズ[バイト]	18	20
クロック数[クロック]	30	32

2.1.2　-far_rom

ROM データのデフォルトの near/far 属性を far にします。 far 領域の方が大きい領域を扱えますが、データアクセスのコードサイズが増加します。尚、オプション省略時のデフォルトの near/far 属性は、-memory_model オプションに従います。

C ソース

```
char* ptr;
const char* c_ptr;
void func()
{
  *ptr = *c_ptr;
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	15	9
クロック数[クロック]	19	13

注意：本オプションを指定した場合、ポインタの指す先が const データであるか否かによってポインタのサイズが異なり、C90 および C99 規約に違反することになります。

```
// -far_rom を使用
char* ptr;            // ポインタ・サイズは 2 バイト。
                      // __near の char を指すポインタ。

const char* c_ptr;    // ポインタ・サイズは 4 バイト。
                      // __far の const char を指すポインタ。
```

2.1.3　-O<level>

最適化のレベルを<default/size/speed/nothing>から指定します。オプション省略時は-Odefault となります。

default：デフォルト。オブジェクト・サイズと実行速度の両方に効果のある最適化を行います。

size：オブジェクト・サイズ優先の最適化。ROM/RAM 容量の削減を重視して、一般的なプログラムに対して有効な最大限の最適化を行います。

speed：実行速度優先の最適化。実行速度の向上を重視して、一般的なプログラムに対して有効な最大限の最適化を行います。

nothing：デバッグ優先の最適化。デバッグのしやすさを重視し、デフォルトで実行する最適化を含むすべての最適化を抑止します。

最適化レベルの選択により、以下の最適化項目のデフォルト値が変化します。

最適化レベル指定による最適化は、下記の最適化項目と 1 対 1 に対応しているわけではありません。例えば、-Odefault を指定して、各最適化項目を-Osize 指定時と同じ値に合わせたとしても、-Osize と同等の最適化を実施するわけではありません。

最適化項目（item）	最適化レベル（level）			
	-Odefault	-Osize	-Ospeed	-Onothing
-Ounroll	1	1	2	1
-Odelete_static_func	on	on	on	off
-Oinline_level	3	3	2	-
-Oinline_size	0	0	100	-
-Otail_call	on	on	on	off
-Opipeline	on	on	on	off
-Osame_code	off	on	off	off

C ソース

```
long a;

void main(void)
{
  unsigned long i = 0;
  unsigned long j = 0;
  for (i = 0; i < 5; ++i) {
    for (j = 0; j < 5; ++j) {
      a += (i + j);
      a *= (i + j);
    }
  }
  for (i = 0; i < 5; ++i) {
    for (j = 0; j < 5; ++j) {
      a += (i + j);
      a *= (i + j);
    }
  }
}
```

	-Odefault	-Osize	-Ospeed	-Onothing
コードサイズ[バイト]	309	279	688	345
クロック数[クロック]	4261	5701	2944	4061

2.1.4　-Ounroll

ループ展開を制御するオプションです。

パラメータに指定された倍数を最大としたループ展開を行います。パラメータが 0 の場合と、パラメータが 1 の場合の動作は同じです。尚、-Onothing を指定した場合、本オプションは無視されます。

ループ展開を行うと、コードサイズは増大しますが、実行速度は向上します。

C ソース

```
int val;

void main(void)
{
  unsigned int i, j, k, l;
  for (i = 1; i < 7; ++i) {
    for (j = 1; j < 6; ++j) {
      for (k = 1; k < 5; ++k) {
        for (l = 1; l < 4; ++l) {
          val += (i + j + k);
          val *= (i + j + k);
        }
      }
    }
  val += (i * 10);
  }
}
```

	-Ounroll=1	-Ounroll=2	-Ounroll=4294967295
コードサイズ[バイト]	152	214	309
クロック数[クロック]	11154	8670	7968

※-Ounroll=4294967295 はパラメータの最大値です

2.1.5　-Odelete_static_func

未使用の static 関数を削除するか否かを制御するオプションです。

-Odelete_static_func=on と指定した場合、最適化が有効となり、-Odelete_static_func=off と指定した場合は無効となります。

未使用の static 関数が削除された場合、コードサイズが減少します。

C ソース

```
int val;

static int func()
{
  return 100;
}

void main(void)
{
  val += func();
}
```

	-Odelete_static_func=on	-Odelete_static_func=off
コードサイズ[バイト]	10	14

2.1.6　-Oinline_level

関数のインライン展開を制御するオプションです。

下記の通り、パラメータの値により展開のレベルが変化します。

-Oinline_level=0 の場合：#pragma inline 指定した関数を含めて、すべてのインライン展開を抑止します。

-Oinline_level=1 の場合：#pragma inline 指定した関数のみ展開します。

-Oinline_level=2 の場合：自動的に展開対象の関数を判別して展開します。

-Oinline_level=3 の場合：コードサイズがなるべく増加しない範囲で、自動的に展開対象の関数を判別して展開します。

但し、1～3 を指定した場合でも、関数の内容やコンパイル状況により、#pragma inline 指定した関数が展開されない場合があります。尚、-Onothing を指定した場合、本オプションは無視されます。

関数のインライン展開を行うと、一般的にはコードサイズは増大しますが、実行速度は向上します。

C ソース

```
int val, x[1000], y[1000];

static void func1(void)
{
  ++val;
}

#pragma inline func2
void func2(int a)
{
    x[a] = x[a] + a;
}

void func3(int a)
{
  if (a) {
    y[a] = a;
  }
}

void main(void)
{
  int i;
  func2(val);
  func3(val);
  for (i = 0; i < 10; ++i) {
    func1();
  }
  func2(val);
  func3(val);
}
```

	-Oinline_level=0	-Oinline_level=1	-Oinline_level=2	-Oinline_level=3
コードサイズ[バイト]	72	88	100	80
クロック数[クロック]	247	229	150	155

※-Oinline_level=2 の場合、-Oinline_size=200 を指定（コードサイズ 200%増加までインライン展開を行う)

#pragma inline 指定の機能は、__inline 宣言と同一です。

#pragma noinline 指定をすると、指定した関数のインライン展開を抑止します。

```
#pragma inline in_func1
void in_func1(void)            // インライン展開を行う
{
}

__inline void in_func2(void)   // インライン展開を行う
{
}

#pragma noinline no_func       // インライン展開を行わない
void no_func(void)
{
}
```

2.1.7　-Oinline_size

コードサイズが何%増加するまで関数のインライン展開を行うか制御するオプションです。

パラメータに 100 を指定した場合、コードサイズが 100%増加するまでインライン展開します。

本オプションは-Oinline_level=2 を指定した場合に有効です。

関数のインライン展開を行うと、一般的にはコードサイズは増大しますが、実行速度は向上します。

C ソース

```
int x[10];

void func1(int a)
{
  x[a] = x[a] * x[a];
}

void func2(int a)
{
  x[a] = x[a] * x[a] * x[a];
}

void func3(int a)
{
  x[a] = x[a] * x[a] * x[a] * x[a] * x[a] * x[a] * x[a] * x[a];
}

void main(void)
{
  int i;
  for (i = 0; i < 10; ++i) {
    func1(i);
    func2(i);
    func3(i);
  }
}
```

	-Oinline_size=0	-Oinline_size=100	-Oinline_size=200	-Oinline_level=65535
コードサイズ[バイト]	107	126	135	165
クロック数[クロック]	988	922	742	612

※-Oinline_level=65535 はパラメータの最大値です

2.1.8　　-Opipeline【V1.03 以降】

パイプライン最適化の有効/無効を切り替えるオプションです

-Opipeline=on と指定した場合は最適化が有効となり、-Opipeline=off と指定した場合は無効となります。尚、-Onothing を指定した場合、本オプションは無視されます。

パイプライン最適化が有効である場合、コンパイラが命令を効率的に実行するための並び替えを行い、実行速度が向上します。

C ソース

```
#define N 2
int a[N*N], b[N*N], c[N*N];

void main(void) {
  int i, j, k;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      c[i*N+j] = 0;
      for (k = 0; k < N; k++) {
        int tmp = a[i*N+k] * b[k*N+j];
        c[i*N+j] += ((tmp >> 2) & (~(0xffffffff << 4))) *
                    ((tmp >> 5) & (~(0xffffffff << 7)));
      }
    }
  }
}
```

	-Opipeline=on	-Opipeline=off
コードサイズ[バイト]	197	197
クロック数[クロック]	279	283

2.1.9　-Otail_call

関数末尾の関数呼び出しを br 命令に置き換える最適化を制御するオプションです。

-Otail_call=on と指定した場合は最適化が有効となり、-Otail_call=off と指定した場合は無効となります。

関数の末尾が関数呼び出しであり、かつ一定の条件を満たす場合に、その呼び出しに対して call 命令ではなく br 命令を生成して ret 命令を削除し、コードサイズが減少し実行速度が向上します。ただし、一部のデバッグ機能を使用することができなくなります。

C ソース

```
int a, b;

void func(void)
{
  a += 1;
}

void main(void)
{
  a += b;
  func();
}
```

	-Otail_call=on	-Otail_call=off
コードサイズ[バイト]	15	17
クロック数[クロック]	14	20

2.1.10　-Omerge_files

複数ファイルをマージしてコンパイルする機能を有効にするオプションです。

本オプションを指定した場合、複数の C ソース・ファイルをマージしてコンパイルし、1 つのファイルを出力します。本オプションを指定しない場合、C ソース・ファイルをマージせず、ファイル単位でコンパイルします。

C ソース[tp1.c]

```
extern long func(long x, long y, long z);
long result;

void main(void)
{
  result = func(3, 4, 5);
}
```

C ソース[tp2.c]

```
#pragma inline (func)
long func(long x, long y, long z)
{
  return (x - y + z);
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	196	217
クロック数[クロック]	10	55

※-Ointermodule 指定時。サイズはスタートアップを含みます。

2.1.11　-Ointermodule

大域最適化を有効にするオプションです。

主に手続き間別名解析を利用した最適化や、パラメータ/戻り値の定数伝播を行います。

C ソース

```
static __near int func(int x, int y, int z) {
  return z - x - y;
}

int func2(void) {
  return func(3, 4, 8);
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	14	17
クロック数[クロック]	25	28

2.1.12　-Owhole_program

入力ファイルがプログラム全体であることを仮定した最適化を有効にするオプションです。

以下の条件を満たすことを前提にコンパイルを行い、条件を満たさなかった場合の動作は保証しません。

- コンパイル対象ファイル内で定義した extern 変数の値およびアドレスが、コンパイル対象ファイル以外で変更および参照されない。
- コンパイル対象ファイル内からコンパイル対象ファイル以外で定義した関数を呼び出している場合、その呼び出された関数からコンパイル対象ファイル内の関数が呼ばれることはない。

本オプションを指定した場合、 -Ointermodule オプションを指定したものとみなしてコンパイルを行います。また入力 C ソース・ファイルが複数の場合、-Omerge_files オプションを指定したものとみなしてコンパイルを行います。

C ソース[tp1.c]

```
extern const int c;
extern int func(void);
int result;

void main(void)
{
  result = c;
  result += func();
}
```

C ソース[tp2.c]

```
#pragma inline (func)
const int c = 1;
int x = 10;
int *p;

int func(void)
{
  int i;
  for (i = 0; i < x; ++i) {
    (*p) += c;
  }
  return (*p);
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	205	192
クロック数[クロック]	194	206

※サイズはスタートアップを含みます

2.1.13　-Oalias

ポインタ指示先の型を考慮した最適化を有効化するオプションです。

最適化を行うと、コードサイズや実行速度といったコード効率が改善します。但し、ISO/IEC 9899 に則った C ソースでない場合、期待した値と異なる実行結果になる可能性があります。

パラメータには ansi と noansi を指定可能です。ansi を指定した場合、ISO/IEC 9899 に基づき、ポインタ指示先の型を考慮した最適化を行います。noansi を指定した場合、ISO/IEC 9899 に基づくポインタ指示先の型を考慮しません。

一般に ansi を指定した場合は noansi を指定した場合よりもオブジェクト性能が向上しますが、異なる実行結果になる場合があります。

C ソース

```
long a, b;
short* ps;

void main(void)
{
  a = 1;
  *ps = 2;
  b = a + *ps;
}
```

	-Oalias=ansi	-Oalias=noansi
コードサイズ[バイト]	26	40
クロック数[クロック]	19	28

2.1.14　-Osame_code【V1.02 以降】

コンパイル単位の同一セクション内に存在する複数の同一命令列を統合し、関数化する最適化を制御するオプションです。

-Osame_code=on と指定した場合は最適化が有効となり、-Osame_code=off と指定した場合は無効となります。尚、-Onothing を指定した場合、本オプションは無視されます。

本最適化により関数呼び出しが増えて実行速度が低下しますが、コードサイズは減少します。

C ソース

```
volatile int value = 0;
int v1;
int v2;
int v3;
int v4;
int v5;

void func(void)
{
  value += v1;
  value += v2;
  value += v3;
  value += v4;
  value += v5;
}

void main(void)
{
  value += v1;
  value += v2;
  value += v3;
  value += v4;
  value += v5;
  func();
}
```

	-Osame_code=on	-Osame_code=off
コードサイズ[バイト]	55	93
クロック数[クロック]	57	39

2.1.15　-dbl_size

double 型と long double 型の解釈を変更するオプションです。

パラメータには 4 もしくは 8 (-cpu=S3 指定時のみ)を指定可能です。

パラメータに 4 を指定した場合、double 型および long double 型を共に float 型とみなします。パラメータに 8 を指定した場合、double 型および long double 型を float 型として扱いません。

なお本オプションを指定しなかった場合、double 型および long double 型を共に float 型とみなします（つまりパラメータに 4 を指定した場合と同じ動作となります）。

C ソース

```
double a, b;
const double c = 11.0;

void main(void)
{
  a = a / b;
  b = b / c;
}
```

	-dbl_size=4	-dbl_size=8
コードサイズ[バイト]	66	121
クロック数[クロック]	132	257

2.1.16　-signed_char

signed も unsigned も付かない char 型を符号付きとして扱います。

本オプション未指定時は signed も unsigned も付かない char 型を符号なしとして扱います。

C ソース

```
int func(char c)
{
  if (c > 10) {
    c++;
  }
  return c;
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	14	8
クロック数[クロック]	10	4

2.1.17　-signed_bitfield

signed も unsigned も付かない型のビットフィールドを符号付きとして扱います。

本オプション未指定時は signed も unsigned も付かない型のビットフィールドを符号なしとして扱います。

C ソース

```
typedef struct T {
  int b1:3;
  int b2:3;
  int b3:16;
}STB;
STB stb;
int i;
unsigned int u;

void func1(STB c)
{
  i = c.b1;
  u = c.b2;
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	22	19
クロック数[クロック]	11	9

2.1.18　-switch

switch 文のコード出力方式を指定します。

-switch=ifelse を指定した場合、if_then 方式で出力します。case ラベルの数が少ないときに有効です。

-switch=binary を指定した場合、バイナリ・サーチ形式で出力します。case ラベルが多いときに本項目を選択すると、どの case ラベルも同じくらいの速さで分岐することができます。case ラベルが多い場合でデータサイズを小さくしたい場合に有効です。

-switch=abs_table を指定した場合、switch 文の case 分岐テーブルを用いる方式で出力します。テーブルには各 case ラベル位置の絶対アドレスを登録します。case ラベルが多いほど ROM サイズが増大しますが、分岐の速度は一定です。データサイズが大きくてもよい場合に有効です。

-switch=rel_table を指定した場合、switch 文の case 分岐テーブルを用いる方式で出力します。テーブルには分岐命令から各 case ラベル位置までの相対距離を登録します。-switch=abs_table よりも ROM サイズは小さくなりますが、相対距離が 64K バイトを超える場合にはリンク・エラーとなります。case ラベルが多い場合でテーブル分岐命令からのラベル位置までの相対距離が 64K バイト以内の場合に有効です。

本オプション未指定時は、コンパイラが switch 文ごとに最適な出力形式を自動的に選択します。

C ソース

```
long val = 0;
void func(int val1)
{
  switch (val1) {
  case 21:
    val += 10;
    break;
  case 22:
    val *= 10;
    break;
  case 23:
    val /= 4;
    break;
  case 24:
    val -= 12;
    break;
  default:
    val = -1;
    break;
  }
}

void main()
{
  int i = 20;
  while (i < 25) {
    func(i);
    ++i;
  }
}
```

	-switch			
	ifelse 指定	binary 指定	abs_table 指定	rel_table 指定
コードサイズ [バイト]	120	130	121	126
ROM サイズ [バイト]	0	0	12	8
クロック数 [クロック]	189	205	225	217

※上記 C ソース例の場合 -switch 未指定では if_then 方式が選択されます

2.1.19　-merge_string

ソース・ファイル内で同じ文字列定数が複数存在する場合、これらをまとめて 1 つの領域に割り付けることで ROM サイズを削減できます。

C ソース

```
#include <string.h>
long val = 0;
char *a = "abcde";
char *b = "abcde";
void func(void)
{
  if (strcmp(a, b) == 0) {
    val = 1;
  }
}
```

	オプション指定	オプション未指定
ROM サイズ[バイト]	6	12

2.1.20　-pack

構造体のパッキング(構造体メンバのアライメントを 1 にする)をします。

本オプションを指定した場合、構造体のメンバをその型でアライメントせず、1 バイトでのアライメントに詰めてコード生成を行います。このため、ROM サイズは小さくなりますが、コードサイズと実行速度は劣化します。

本オプション指定によって整列条件が 2 バイトから 1 バイトに変更となった構造体、共用体、または、それらのメンバのアドレスを、標準ライブラリ関数の実引数として渡した場合、動作は保証しません。

本オプション指定によって整列条件が 2 バイトから 1 バイトに変更となった構造体、または共用体メンバのアドレスを、整列条件が 2 バイトである型のポインタに渡して間接参照をした場合、動作は保証しません。

C ソース

```
struct{
  signed char  a;
  signed long  b;
  struct{
    signed char c;
    signed long d;
  }f;
}data, *stp;

void func()
{
  data.a = 1;
  data.b = 2;
  data.f.c = 5;
  data.f.d = 6;
  if (stp->b != stp->f.d) {
    data.b++;
  }
}
```

	オプション指定	オプション未指定
ROM サイズ[バイト]	12	14
コードサイズ[バイト]	97	65
クロック数[クロック]	47	37

本機能は#pragma でも指定できます。オプションと#pragma の両方が指定された場合は、#pragma の指定を優先します。

```
struct s1 {
  char a;
  long b;       // -pack 指定時はアライメント 1、未指定時はアライメント 2
} data1;

#pragma pack
struct s2 {
  char a;
  long b;       // オプション指定に関わらず常にアライメント 1
} data2;

#pragma unpack
struct s3 {
  char a;
  long b;       // オプション指定に関わらず常にアライメント 2
} data3;
```

2.1.21　-stack_protector/-stack_protector_all【Professional 版のみ】【V1.02 以降】

関数の入口・出口にスタック破壊検出コードを生成します。

検出コードの分、コードサイズと実行速度が劣化します。

C ソース

```
#include <stdio.h>
#include <stdlib.h>

void f1()                    // スタックが破壊されるプログラムの例
{
  volatile char str[10];
  int i;
  for (i = 0; i <= 9; i++) {
    str[i] = i;              // i=10 の場合にスタックが破壊される
  }
}

void __stack_chk_fail(void)
{
  printf("stack is broken!");
}

void main()
{
  f1();
}
```

	-stack_protector 指定	-stack_protector_all 指定	オプション未指定
コードサイズ[バイト]	50	62	38
クロック数[クロック]	1075	1080	1073

本機能は#pragma でも指定できます。オプションと#pragma の両方が指定された場合は、#pragma の指定を優先します。

例) -stack_protector/-stack_protector_all 指定時

```
struct DATA
{
  int a, b, c, d;
};

struct DATA func1(void)    // スタック破壊検出コードを生成する
{
  struct DATA data = {0, 1, 2, 3};
  return data;
}

#pragma no_stack_protector (func2)
struct DATA func2(void)    // スタック破壊検出コードを生成しない
{
  struct DATA data = {0, 1, 2, 3};
  return data;
}
```

例) -stack_protector/-stack_protector_all 未指定時

```
struct DATA
{
  int a, b, c, d;
};

struct DATA func1(void)    // スタック破壊検出コードを生成しない
{
  struct DATA data = {0, 1, 2, 3};
  return data;
}

#pragma stack_protector (func2)
struct DATA func2(void)    // スタック破壊検出コードを生成する
{
  struct DATA data = {0, 1, 2, 3};
  return data;
}
```

2.1.22　-control_flow_integrity【Professional 版のみ】【V1.06 以降】

間接関数呼び出しに対して、呼び出し先関数のチェックを行います。

チェックのためのコードとデータが生成されるため、コードサイズ、ROM サイズ、実行速度が劣化します。

C ソース

```
#include <stdlib.h>
int glb;
void __control_flow_chk_fail(void)
{
  abort();
}
void func1(void) // 関数リストに追加される
{
  ++glb;
}
void func2(void) // 関数リストに追加されない
{
  --glb;
}
void (*pf)(void) = func1;
void main(void)
{
  pf(); // func1 関数の間接呼び出し
  func2();
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	72	60
ROM サイズ[バイト]	59	0
クロック数[クロック]	91	26

※コードサイズおよび ROM サイズはスタートアップを含みます。

2.1.23　-unaligned_pointer_for_ca78k0r【V1.06 以降】

ポインタ間接参照を 1 バイト単位でアクセスします。本オプションは、CA78K0R からのコンパイラ移行支援が目的です。

本オプションを指定した場合、オブジェクト・サイズが大きくなり、実行速度が低下します。

C ソース

```
#include <stdlib.h>

char c;
int *glbp = &c;

int func1(void)
{
  return *glbp;
}

void main()
{
  c = func1();
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	8	5
クロック数[クロック]	5	3

2.2　アセンブル・オプション

○…改善する　×…劣化する　△…改善することもあれば劣化することもある　―…変化しない

オプション	コード サイズ	ROM サイズ	クロック 数	備考
-goptimize	○	○	―	本オプションを指定したファイルは、リンク時のモジュール間最適化の対象になります。 リンク時の最適化については 2.3 リンク・オプションを参照ください。

2.3　リンク・オプション

リンカの最適化オプション指定によるコードサイズ・ROM サイズ・実行速度の影響を示します。コンパイル、アセンブル時に-goptimize を指定したファイルに対して最適化を行います。

-section_forbid で指定したセクションの最適化を抑止することができます。

また、-absolute_forbid で指定したアドレス＋サイズの範囲の最適化を抑止することができます。

○…改善する　×…劣化する　△…改善することもあれば劣化することもある　―…変化しない

オプション	コードサイズ	ROM サイズ	クロック数	備考
-optimize=symbol_delete	○	○	―	
-optimize=branch	○	―	―	

※コマンドラインから起動した場合、デフォルトでは全ての最適化を実行します。

2.3.1　-optimize=symbol_delete

1 度も参照のない変数/関数を削除します。リンカの-entry で entry シンボルを指定してください。

-symbol_forbid で指定した変数/関数の削除を抑止することができます。

C ソース

```
int value1 = 0;
int value2 = 0;

void func1(void)
{
  value1++;
}

void func2(void)
{
  value2++;
}

void main(void)
{
  func1();
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	156	162
ROM サイズ[バイト]	144	146

※コードサイズおよび ROM サイズはスタートアップを含みます

2.3.2　-optimize=branch

プログラムの配置情報に基づいて、分岐命令サイズを最適化します。

C ソース

```
extern void sub(void);

void main(void)
{
  sub();
  sub();
  sub();
  sub();
  sub();
}
```

	オプション指定	オプション未指定
コードサイズ[バイト]	168	176

※コードサイズはスタートアップを含みます

3.　拡張言語

拡張言語によるコードサイズ・ROM サイズ・実行速度への影響を示します。

3.1　　予約語

○…改善する　×…劣化する　△…状況に依存する　―…変化しない

予約語	コードサイズ	ROM サイズ	クロック数	備考
__saddr	○	―	○	
__callt	○	×	×	
__near __far	△	△	△	配置領域によって変化します。呼び出し頻度が高い関数、変数は near 領域に配置すると効果的です。

3.1.1　__saddr

__saddr 宣言された外部変数を saddr 領域に割り当てます。

初期値あり変数はセクション.sdata に配置します。

初期値なし変数はセクション.sbss に配置します。

使用頻度の高い外部変数および関数内 static 変数は、saddr 領域に割り当てることでオブジェクト性能が向上します。特に、1 ビットのビットフィールドを saddr 領域に割り当てると効果が大きくなる傾向にあります。

変更前	変更後

C ソース
 	
`typedef struct {`
`  unsigned char b0:1;`
`  unsigned char b1:1;`
`  unsigned char b2:1;`
`  unsigned char b3:1;`
`  unsigned char b4:1;`
`  unsigned char b5:1;`
`  unsigned char b6:1;`
`  unsigned char b7:1;`
`} BITF;`
`BITF data0, data1;`
 	
`void func(void)`
`{`
`  data0.b4 = data1.b1;`
`}`
	C ソース
	
	`typedef struct {`
	`  unsigned char b0:1;`
	`  unsigned char b1:1;`
	`  unsigned char b2:1;`
	`  unsigned char b3:1;`
	`  unsigned char b4:1;`
	`  unsigned char b5:1;`
	`  unsigned char b6:1;`
	`  unsigned char b7:1;`
	`} BITF;`
	`__saddr BITF data0, data1;`
	
	`void func(void)`
	`{`
	`  data0.b4 = data1.b1;`
	`}`

アセンブリ展開コード
 	
`_func:`
`  .STACK _func = 4`
`  movw hl, #LOWW(_data1)`
`  mov1 CY, [hl].1`
`  movw hl, #LOWW(_data0)`
`  mov1 [hl].4, CY`
`  ret`
	アセンブリ展開コード
	
	`_func:`
	`  .STACK _func = 4`
	`  mov1 CY, _data1.1`
	`  mov1 _data0.4, CY`
	`  ret`

コードサイズ:11 バイト クロック数:13 クロック	コードサイズ:7 バイト クロック数:9 クロック

本機能は#pragma saddr でも指定できます。__near/__far の宣言よりも#pragma の指定を優先します。

```
#pragma saddr value
int __far value;        // saddr 領域に配置
```

3.1.2　__callt

__callt 宣言された関数(callt 関数)は、callt 命令で呼び出します。callt 関数は__near 指定となり、アドレス参照は必ず near ポインタを返します。

callt テーブル領域[80H -BFH] にコールする関数のアドレスを格納し、通常のコール命令(call 命令)よりも小さいサイズで関数をコールすることが可能です。

関数アドレスのテーブルを生成するため、ROM サイズは増加します。また、callt 命令は call 命令よりも実行クロック数が多いため、実行速度は劣化します。

変更前	変更後

C ソース
 	
`#pragma noinline sub`
`void sub(void)`
`{`
`}`
 	
`void func(void)`
`{`
`  sub();`
`  sub();`
`  sub();`
`  sub();`
`  sub();`
`}`
	C ソース
	
	`#pragma noinline sub`
	`__callt void sub(void)`
	`{`
	`}`
	
	`void func(void)`
	`{`
	`  sub();`
	`  sub();`
	`  sub();`
	`  sub();`
	`  sub();`
	`}`

アセンブリ展開コード
 	
`_func:`
`  .STACK _func = 4`
`  call $!_sub`
`  call $!_sub`
`  call $!_sub`
`  call $!_sub`
`  br $_sub`
	アセンブリ展開コード
	
	`_func:`
	`  .STACK _func = 4`
	`  callt [@_sub]`
	`  callt [@_sub]`
	`  callt [@_sub]`
	`  callt [@_sub]`
	`  br !_sub`
	
	`  .SECTION .callt0,CALLT0  (※)`
	`@_sub:`
	`  .DB2 _sub`

コードサイズ:14 バイト クロック数:45 クロック	コードサイズ:11 バイト クロック数:53 クロック ※関数アドレスのテーブル生成のため、 ROM サイズは増加(+2 バイト)

本機能は#pragma callt でも指定できます。

```
#pragma callt sub
void sub(void)          // callt 関数
{
}
```

3.1.3　__near/__far

関数、変数の宣言時に__near/__far 型修飾子を追加することにより、関数、変数の配置場所を明示的に指定することができます。領域のサイズは far 領域の方が大きいですが、関数呼出しのコードサイズ、データアクセスのコードサイズも大きくなります。呼び出し頻度が高い関数、変数は near 領域に配置するとコードサイズを削減できます。

far 領域配置	near 領域配置

C ソース
 	
`#pragma noinline sub`
`void __far sub(void)`
`{`
`}`
 	
`int __far value;`
`void func(void)`
`{`
`  sub();`
`  value += 10;`
`}`
	C ソース
	
	`#pragma noinline sub`
	`void __near sub(void)`
	`{`
	`}`
	
	`int __near value;`
	`void func(void)`
	`{`
	`  sub();`
	`  value += 10;`
	`}`

アセンブリ展開コード
 	
`_func:`
`  .STACK _func = 4`
`  call $!_sub`
`  mov es, #LOW(HIGHW(_value))`
`  movw ax, #0x000A`
`  addw ax, es:!LOWW(_value)`
`  movw es:!LOWW(_value), ax`
`  ret`
	アセンブリ展開コード
	
	`_func:`
	`  .STACK _func = 4`
	`  call !_sub`
	`  movw ax, #0x000A`
	`  addw ax, !LOWW(_value)`
	`  movw !LOWW(_value), ax`
	`  ret`

コードサイズ:17 バイト クロック数:21 クロック	コードサイズ:13 バイト クロック数:18 クロック

関数の配置場所は#pragma near/#pragma far でも指定できます。(V1.05 以降)

__near/__far/__callt の宣言よりも#pragma の指定を優先します。

```
#pragma near func1
void __far func1(void)    // near 領域に配置
{
}
```

__near/__far 型修飾子を付けない宣言はメモリ・モデルにより決定するデフォルトの near/far 属性に従います。near/far 属性の決定方法を以下に示します。

	項目	near/far 属性の決定方法
(a)	-cpu	デフォルトのメモリ・モデルを決める。
(b)	-memory_model	(a)で決めたメモリ・モデルを上書きする。
(c)	-far_rom	ROM データのみを far 属性で上書きする。
(d)	__near/__far	(a)～(c)の影響を受けない。修飾子記述が有効になる。
(e)	#pragma near #pragma far	(a)～(d)の影響を受けない。#pragma 記述が有効になる。

3.2　#pragma 指令

○…改善する　×…劣化する　△…状況に依存する　―…変化しない

#pragma 指令	コードサイズ	ROM サイズ	クロック数	備考
#pragma interrupt #pragma interrupt_brk	△	△	△	割り込み仕様を変更することで、割り込み関数の性能が改善する場合があります。

3.2.1　#pragma interrupt/interrupt_brk

割り込み関数となる関数を宣言します。割り込み仕様を変更することで、割り込み関数の速度・サイズは変化します。

割り込み仕様	形式	内容
レジスタ・バンク指定	bank	レジスタ・バンクを変更し、汎用レジスタをスタックに退避しません。ただし、ES と CS はスタックに退避します。
多重割り込み許可指定	enable	関数入口に EI を生成し、多重割り込みを可能とします。

C ソース

```
#pragma interrupt_brk func
void func(void)
{
  sub(1, 2, 3);
}
```

この C ソースに対して割り込み仕様を指定しない場合と、割り込み仕様を指定した場合を比較すると次の表の通りになります。

	未指定	bank=RB0	enable
コードサイズ[バイト]	32	26	35
クロック数[クロック]	35	28	39

4.　変数／関数情報ファイルの利用

変数／関数情報ファイルは、C ソース・ファイル中で定義された変数、および関数に対して、saddr 変数や callt 関数の宣言を記述したテキスト形式のファイルです。

参照頻度の高い変数を saddr 変数、参照頻度の高い関数を callt 関数にすることでコードサイズを削減します。

[使用方法]

最初に、リンカの-vfinfo オプションを指定して変数／関数情報ファイルを出力してください。

変数／関数情報ファイルを次のいずれかの方法でコンパイル時にインクルードしてください。

・コンパイラの-preinclude オプションで指定する

・各 C ソース・ファイルに#include でインクルードする

C ソース

```
int value1;
int value2;

void func1(void)
{
  value1 += 100;
}

void func2(void)
{
  value1 += 100;
}

void sub(void)
{
  func1();
  func1();
  func1();
  func1();
  func1();
}

void main(void) {
  sub();
}
```

変数／関数情報ファイル出力

```
/* RENESAS OPTIMIZING LINKER GENERATED FILE xxxx.xx.xx */
/*** variable information ***/
#pragma saddr value1 /* count:4,size:2,near,VFINFO.obj */
/* #pragma saddr value2 */ /* count:0,size:2,near,unref,VFINFO.obj */ ※

/*** function information ***/
#pragma callt func1 /* count:5,far,VFINFO.obj */
#pragma callt main /* count:1,far,VFINFO.obj */
#pragma callt sub /* count:1,far,VFINFO.obj */
/* #pragma callt func2 */ /* count:0,far,unref,VFINFO.obj */          ※
```

※変数 value2 と関数 func2 は参照が無いのでコメントアウトで出力(saddr 変数、callt 関数にならない)

callt 関数呼び出しは、コードサイズが小さくなりますが実行速度は遅くなります。実行速度も優先する場合は near 関数に置き換えるのがよいです。-vfinfo(near)と指定すると、near 関数として出力することができます。

	変数／関数情報ファイル使用		変数／関数情報ファイル未使用
	-vfinfo(near)	-vfinfo	
コードサイズ[バイト]	187	182	191
ROM サイズ[バイト]	142	148	142
クロック数[クロック]	63	73	63

※サイズはスタートアップを含みます。

5.　コーディングテクニック

本章では、ユーザプログラムのコーディング方法によりコードサイズ・ROM サイズ・実行速度を改善する方法を説明します。

○…改善する　×…劣化する　△…状況に依存する　―…変化しない

項目	コードサイズ	ROMサイズ	クロック数	備考
変数と const 型	―	○	―	
局所変数と大域変数	○	○	○	
ビットフィールドの割り付け	○	―	○	
関数のインタフェース	○	―	○	
ループ回数の削減	×	―	○	
テーブルの活用	○	×	○	
分岐	―	―	○	
インライン展開	△	―	○	
条件分岐先の同一文は分岐前に移動する	○	―	○	
複雑な if 文を論理的に等しいものに置き換える	○	―	○	
変数のサイズ	○	△	○	
switch 文の共通な case の処理をまとめる	○	―	○	
for ループを do while ループに置き換える	○	―	○	
2 のべき乗の除算はシフト演算に置き換える	○	―	○	
2 ビット以上のビットフィールドは char 型に変更する	○	×	○	
構造体のアライメント	―	○	―	

5.1　変数と const 型

値を変更しない変数は、const 修飾で宣言してください。

大域変数を宣言と同時に初期化するプログラムを書くと、初期値は ROM に、大域変数は RAM にそれぞれ配置されます。大域変数の初期化は、プログラムの開始時に ROM から RAM に初期値を転送することにより実現されています。初期値のある変数を const 修飾しておくと、変数を書き換えるおそれがなくなるため、コンパイラは RAM を確保しません。その結果、RAM を節約することができ、ROM から RAM への転送処理も省略することができます。

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。

変更前	変更後

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

ROM サイズ:5 バイト RAM サイズ:5 バイト	ROM サイズ:5 バイト RAM サイズ:0 バイト

5.2　局所変数と大域変数

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が向上します。

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、最適化が掛りにくくなります。

変更前	変更後

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

アセンブリ展開コード
 	
`_func:`
`  .STACK _func = 6`
`  movw de, ax`
`  push bc`
`  pop hl`
`  movw ax, [de]`
`  movw !LOWW(_tmp), ax`
`  movw ax, [hl]`
`  movw [de], ax`
`  movw ax, !LOWW(_tmp)`
`  movw [hl], ax`
`  ret`
	アセンブリ展開コード
	
	`_func:`
	`  .STACK _func = 6`
	`  movw de, ax`
	`  push bc`
	`  pop hl`
	`  movw ax, [de]`
	`  movw bc, ax`
	`  movw ax, [hl]`
	`  movw [de], ax`
	`  movw ax, bc`
	`  movw [hl], ax`
	`  ret`

コードサイズ:14 バイト クロック数:15 クロック	コードサイズ:10 バイト クロック数:15 クロック

5.3　ビットフィールドの割り付け

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

異なる構造体内にあるビットフィールドのメンバを設定するためには、構造体ごとに分けて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せることができます。

以下は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です。

変更前	変更後

C ソース
 	
`struct str`
`{`
`  int flag1:1;`
`} b1, b2, b3;`
 	
`void func(void)`
`{`
`  b1.flag1 = 1;`
`  b2.flag1 = 1;`
`  b3.flag1 = 1;`
`}`
	C ソース
	
	`struct str`
	`{`
	`  int flag1:1;`
	`  int flag2:1;`
	`  int flag3:1;`
	`} a1;`
	
	`void func(void)`
	`{`
	`  a1.flag1 = 1;`
	`  a1.flag2 = 1;`
	`  a1.flag3 = 1;`
	`}`

アセンブリ展開コード
 	
`_func:`
`  .STACK _func = 4`
`  set1 !LOWW(_b1).0`
`  set1 !LOWW(_b2).0`
`  set1 !LOWW(_b3).0`
`  ret`
	アセンブリ展開コード
	
	`_func:`
	`  .STACK _func = 4`
	`  mov a, #0x07`
	`  or a, !LOWW(_a1)`
	`  mov !LOWW(_a1), a`
	`  ret`

コードサイズ:13 バイト クロック数:15 クロック	コードサイズ:9 バイト クロック数:12 クロック

5.4　関数のインタフェース

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。

引数がすべてレジスタに割り付くように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。

関数のインタフェース仕様は、コンパイラユーザーズマニュアルに記載されています。

変更前	変更後

C ソース
 	
`struct str`
`{`
`  char a;`
`  char b;`
`  char c;`
`  char d;`
`  char e;`
`  char f;`
`  char g;`
`  char h;`
`} arg;`
 	
`void func(char a, char b, char c,`
`  char d, char e, char f,`
`  char g, char h){}`
 	
`void call_func(void)`
`{`
`  func(arg.a, arg.b, arg.c, arg.d,`
`  arg.e, arg.f, arg.g, arg.h);`
`}`
	C ソース
	
	`struct str`
	`{`
	`  char a;`
	`  char b;`
	`  char c;`
	`  char d;`
	`  char e;`
	`  char f;`
	`  char g;`
	`  char h;`
	`} arg;`
	
	`void func(struct str* str_arg){}`
	
	`void call_func(void)`
	`{`
	`  func(&arg);`
	`}`

アセンブリ展開コード
 	
`_call_func:`
`  .STACK _call_func = 8`
`  mov a, !LOWW(_arg+0x00007)`
`  shrw ax, 8+0x00000`
`  push ax`
`  mov a, !LOWW(_arg+0x00006)`
`  shrw ax, 8+0x00000`
`  push ax`
`  mov a, !LOWW(_arg+0x00005)`
`  mov d, a`
`  mov a, !LOWW(_arg+0x00004)`
`  mov e, a`
`  mov b, !LOWW(_arg+0x00003)`
`  mov c, !LOWW(_arg+0x00002)`
`  mov x, !LOWW(_arg+0x00001)`
`  mov a, !LOWW(_arg)`
`  call $!_func`
`  addw sp, #0x04`
`  ret`
	アセンブリ展開コード
	
	`_call_func:`
	`  .STACK _call_func = 4`
	`  movw ax, #LOWW(_arg)`
	`  br $_func`

コードサイズ:38 バイト クロック数:31 クロック	コードサイズ:5 バイト クロック数:10 クロック

5.5　ループ回数の削減

ループを展開すると、実行速度は大幅に向上します。

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。

<table>
<tr><th>変更前</th><th>変更後</th></tr>
<tr><td>C ソース

<pre>
int a[100];

void func(void)
{
  int i;
  for (i = 0; i < 100; i++) {
    a[i] = 0;
  }
}
</pre></td><td>C ソース

<pre>
int a[100];

void func(void)
{
  int i;
  for (i = 0; i < 100; i += 2) {
    a[i] = 0;
    a[i+1] = 0;
  }
}
</pre></td></tr>
<tr><td>アセンブリ展開コード

<pre>
_func:
    .STACK _func = 4
    movw de, #LOWW(_a)
    movw bc, #0x0064
.BB@LABEL@1_1: ; bb
    clrw ax
    movw [de], ax
    movw ax, bc
    addw ax, #0xFFFF
    movw bc, ax
    incw de
    incw de
    bnz $.BB@LABEL@1_1
.BB@LABEL@1_2: ; return
    ret
</pre></td><td>アセンブリ展開コード

<pre>
_func:
    .STACK _func = 6
    push hl
    movw ax, #LOWW(_a)
    movw [sp+0x00], ax
    movw hl, #0x0032
.BB@LABEL@1_1: ; bb
    pop de
    push de
    clrw bc
    movw ax, bc
    movw [de], ax
    movw ax, de
    incw ax
    incw ax
    movw de, ax
    movw ax, bc
    movw [de], ax
    movw ax, [sp+0x00]
    addw ax, #0x0004
    movw [sp+0x00], ax
    movw ax, hl
    addw ax, #0xFFFF
    movw hl, ax
    bnz $.BB@LABEL@1_1
.BB@LABEL@1_2: ; return
    pop hl
    ret
</pre></td></tr>
<tr><td>コードサイズ:18 バイト
クロック数:1106 クロック</td><td>コードサイズ:36 バイト
クロック数:1059 クロック</td></tr>
</table>

5.6　テーブルの活用

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。

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。

以下の例では、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。

<table>
<tr><th>変更前</th><th>変更後</th></tr>
<tr><td>C ソース

<pre>
char func(int i)
{
  char ch;
  switch (i) {
  case 0:
    ch = 'a'; break;
  case 1:
    ch = 'x'; break;
  case 2:
    ch = 'b'; break;
  default:
    ch = 0; break;
  }
  return (ch);
}
</pre></td><td>C ソース

<pre>
const char chbuf[] =
  {'a', 'x', 'b'};

char func(int i)
{
  if ((unsigned int)i < 3) {
    return (chbuf[i]);
  }
  return (0);
}
</pre></td></tr>
<tr><td>アセンブリ展開コード

<pre>
_func:
    .STACK _func = 4
    cmpw ax, #0x0000
    bz $.BB@LABEL@1_4
.BB@LABEL@1_1: ; entry
    addw ax, #0xFFFF
    bz $.BB@LABEL@1_5
.BB@LABEL@1_2: ; entry
    cmpw ax, #0x0001
    bz $.BB@LABEL@1_6
.BB@LABEL@1_3: ; switch_clause_bb5
    clrb a
    ret
.BB@LABEL@1_4: ; switch_break_bb
    mov a, #0x61
    ret
.BB@LABEL@1_5: ; switch_clause_bb3
    mov a, #0x78
    ret
.BB@LABEL@1_6: ; switch_clause_bb4
    mov a, #0x62
    ret
</pre></td><td>アセンブリ展開コード

<pre>
_func:
    .STACK _func = 4
    cmpw ax, #0x0003
    bnc $.BB@LABEL@1_2
.BB@LABEL@1_1: ; if_then_bb
    movw bc, ax
    mov a, SMRLW(_chbuf)[bc]
    ret
.BB@LABEL@1_2: ; bb10
    clrb a
    ret
</pre></td></tr>
<tr><td>コードサイズ:59 バイト
クロック数:82 クロック</td><td>コードサイズ:44 バイト
クロック数:78 クロック</td></tr>
</table>

5.7　分岐

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。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。

<table>
<tr><th>変更前</th><th>変更後</th></tr>
<tr><td>C ソース


```
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);
}
```

</td><td>C ソース


```
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);
}
```

</td></tr>
<tr><td>アセンブリ展開コード


```
_func:
    .STACK _func = 4
    cmpw ax, #0x0001
    bnz $.BB@LABEL@1_2
.BB@LABEL@1_1: ;
entry.if_break_bb17_crit_edge
    onew ax
    incw ax
    br $.BB@LABEL@1_7
.BB@LABEL@1_2: ; if_else_bb
    cmpw ax, #0x0002
    bnz $.BB@LABEL@1_4
.BB@LABEL@1_3: ;
if_else_bb.if_break_bb17_crit_edge
    movw ax, #0x0004
    br $.BB@LABEL@1_7
.BB@LABEL@1_4: ; if_else_bb9
    cmpw ax, #0x0003
    oneb a
    skz
.BB@LABEL@1_5: ; if_else_bb9
    clrb a
.BB@LABEL@1_6: ; if_else_bb9
    mov x, #0x08
    mulu x
.BB@LABEL@1_7: ; if_break_bb17
    ret
```

</td><td>アセンブリ展開コード


```
_func:
    .STACK _func = 4
    cmpw ax, #0x0003
    bnz $.BB@LABEL@1_2
.BB@LABEL@1_1: ;
entry.if_break_bb17_crit_edge
    movw ax, #0x0008
    br $.BB@LABEL@1_7
.BB@LABEL@1_2: ; if_else_bb
    cmpw ax, #0x0002
    bnz $.BB@LABEL@1_4
.BB@LABEL@1_3: ;
if_else_bb.if_break_bb17_crit_edge
    movw ax, #0x0004
    br $.BB@LABEL@1_7
.BB@LABEL@1_4: ; if_else_bb9
    cmpw ax, #0x0001
    oneb a
    skz
.BB@LABEL@1_5: ; if_else_bb9
    clrb a
.BB@LABEL@1_6: ; if_else_bb9
    mov x, #0x02
    mulu x
.BB@LABEL@1_7: ; if_break_bb17
    ret
```

</td></tr>
<tr><td>クロック数:26 クロック
(a=3 の場合)</td><td>クロック数:17 クロック
(a=3 の場合)</td></tr>
</table>

5.8　インライン展開

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。#pragma inline でインライン展開する関数を指定することができます。一方、インライン展開をした場合、一般的にはプログラムサイズが増大する傾向にあります。

インライン展開する関数が他のソース・ファイルから参照されていない場合、static 関数にしておくと関数本体のコードが削除されコードサイズが小さくなる可能性があります。

変更前	変更後

C ソース
 	
`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);`
`  }`
`}`
	C ソース
	
	`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);`
	`  }`
	`}`

アセンブリ展開コード	アセンブリ展開コード

_sub@1:
.STACK _sub@1 = 8
push ax
push bc
pop hl
movw ax, de
addw ax, ax
movw bc, ax
movw ax, hl
addw ax, bc
movw de, ax
movw ax, [sp+0x00]
addw ax, bc
movw hl, ax
movw ax, [hl]
movw bc, ax
movw ax, [de]
movw [hl], ax
movw ax, bc
movw [de], ax
pop hl
ret
_func:
.STACK _func = 6
push hl
clrw ax
movw [sp+0x00], ax
.BB@LABEL@2_1: ; bb
movw de, ax
movw bc, #LOWW(_y)
movw ax, #LOWW(_x)
call $!_sub@1
movw ax, [sp+0x00]
incw ax
movw [sp+0x00], ax
cmpw ax, #0x000A
bnz $.BB@LABEL@2_1
.BB@LABEL@2_2: ; return
pop hl
ret
	_func:
	.STACK _func = 6
	push hl
	movw de, #LOWW(_y)
	movw bc, #0x000A
	movw hl, #LOWW(_x)
	.BB@LABEL@1_1: ; bb
	movw ax, [hl]
	movw [sp+0x00], ax
	movw ax, [de]
	movw [hl], ax
	movw ax, [sp+0x00]
	movw [de], ax
	movw ax, bc
	addw ax, #0xFFFF
	movw bc, ax
	incw de
	incw de
	incw hl
	incw hl
	bnz $.BB@LABEL@1_1
	.BB@LABEL@1_2: ; return
	pop hl
	ret

コードサイズ:26 バイト クロック数:411 クロック (-Oinline_level=1 指定時)	コードサイズ:31 バイト クロック数:183 クロック (-Oinline_level=1 指定時)

#pragma inline はインライン展開されることを保証するものではありません。-Oinline_level の指定と、関数の内容やコンパイル状況によりインライン展開されない場合があります。

5.9　条件分岐先の同一文は分岐前に移動する

条件分岐の各々の分岐先に同一の文がある場合、条件分岐の前に移動して 1 箇所にまとめてください。

変更前

C ソース

```
int s;

int func(int a, int b, int c)
{
  return (a + b + c);
}

int call_func(int x)
{
  if (x >= 0) {
    if (x > func(0, 1, 2)) {
       s++;
    }
  }
  else {
    if (x < -func(0, 1, 2)) {
       s--;
    }
  }
  return 0;
}
```

変更後

C ソース

```
int s;

int func(int a, int b, int c)
{
  return (a + b + c);
}

int call_func(int x)
{
  int tmp = func(0, 1, 2);
  if (x >= 0) {
    if (x > tmp) {
      s++;
    }
  }
  else {
    if (x < -tmp) {
      s--;
    }
  }
  return 0;
}
```

アセンブリ展開コード

```
_call_func:
   .STACK _call_func = 6
   push ax
   bt a.7, $.BB@LABEL@2_5
.BB@LABEL@2_1: ; if_then_bb
   movw de, #0x0002
   onew bc
   clrw ax
   call $!_func
   movw bc, ax
   movw ax, [sp+0x00]
   xor a, #0x80
   movw [sp+0x00], ax
   xchw ax, bc
   xor a, #0x80
   cmpw ax, bc
   bnc $.BB@LABEL@2_3
.BB@LABEL@2_2: ; if_then_bb9
   incw !LOWW(_s)
   br $.BB@LABEL@2_5
.BB@LABEL@2_3: ; if_else_bb
   movw de, #0x0002
   onew bc
   clrw ax
   call $!_func
   movw bc, ax
   clrw ax
   subw ax, bc
   xor a, #0x80
   movw bc, ax
   movw ax, [sp+0x00]
   cmpw ax, bc
   sknc
.BB@LABEL@2_4: ; if_then_bb18
   decw !LOWW(_s)
.BB@LABEL@2_5: ; if_break_bb22
   clrw ax
   pop hl
   ret
```

コードサイズ:55 バイト
クロック数:65 クロック

アセンブリ展開コード

```
_call_func:
   .STACK _call_func = 6
   push ax
   movw de, #0x0002
   onew bc
   clrw ax
   call $!_func
   movw bc, ax
   movw ax, [sp+0x00]
   bt a.7, $.BB@LABEL@2_5
.BB@LABEL@2_1: ; if_then_bb
   xor a, #0x80
   movw de, ax
   movw ax, bc
   xor a, #0x80
   cmpw ax, de
   bnc $.BB@LABEL@2_3
.BB@LABEL@2_2: ; if_then_bb12
   incw !LOWW(_s)
   br $.BB@LABEL@2_5
.BB@LABEL@2_3: ; if_else_bb
   clrw ax
   subw ax, bc
   xor a, #0x80
   movw bc, ax
   movw ax, de
   cmpw ax, bc
   sknc
.BB@LABEL@2_4: ; if_then_bb21
   decw !LOWW(_s)
.BB@LABEL@2_5: ; if_break_bb25
   clrw ax
   pop hl
   ret
```

コードサイズ:44 バイト
クロック数:50 クロック

5.10　複雑な if 文を論理的に等しいものに置き換える

if 文の条件式が複雑である場合、等しい意味の簡単な式に置き換えてください。

<table>
<tr><th>変更前</th><th>変更後</th></tr>
<tr><td>C ソース

<pre>
int x;

int func(int s, int t)
{
  s &= 1;
  t &= 1;
  if (!s) {
    if (t) {
       x = 1;
    }
  }
  else {
    if (!t) {
       x = 1;
    }
  }
  return 0;
}
</pre></td><td>C ソース

<pre>
int x;

int func(int s, int t)
{
  s &= 1;
  t &= 1;
  if (! (s ^ t)) {
    x = 1;
  }
  return 0;
}
</pre></td></tr>
<tr><td>アセンブリ展開コード

<pre>
_func:
    .STACK _func = 4
    movw de, ax
    mov a, c
    and a, #0x01
    mov x, a
    mov a, e
    bt a.0, $.BB@LABEL@1_2
.BB@LABEL@1_1: ; bb10.thread
    cmp0 x
    bnz $.BB@LABEL@1_3
    br $.BB@LABEL@1_4
.BB@LABEL@1_2: ; if_else_bb
    cmp0 x
    bnz $.BB@LABEL@1_4
.BB@LABEL@1_3: ; if_then_bb28
    onew ax
    movw !LOWW(_x), ax
.BB@LABEL@1_4: ; if_break_bb30
    clrw ax
    ret
</pre></td><td>アセンブリ展開コード

<pre>
_func:
    .STACK _func = 4
    xor a, b
    xch a, x
    xor a, c
    xch a, x
    mov a, x
    bt a.0, $.BB@LABEL@1_2
.BB@LABEL@1_1: ; if_then_bb
    onew ax
    movw !LOWW(_x), ax
.BB@LABEL@1_2: ; if_break_bb
    clrw ax
    ret
</pre></td></tr>
<tr><td>コードサイズ:23 バイト
クロック数:27 クロック</td><td>コードサイズ:16 バイト
クロック数:22 クロック</td></tr>
</table>

5.11　変数の型

変数は可能な限り小さいサイズの型を使用してください。RL78 ファミリが小さいサイズの型を得意としたデバイスであるためです。

注意：型を変更することで、変数や演算結果が取り得る値の範囲が変わります。型を変更する場合はプログラムの動作に影響が無いように注意してください。

<table>
<tr><th>変更前</th><th>変更後</th></tr>
<tr><td>C ソース

<pre>
int func(int a, int b, int c)
{
  int t = a + b;
  return (t >> c);
}
</pre></td><td>C ソース

<pre>
char func(char a, char b, char c)
{
  char t = a + b;
  return (t >> c);
}
</pre></td></tr>
<tr><td>アセンブリ展開コード

<pre>
_func:
    .STACK _func = 6
    push hl
    movw hl, ax
    movw ax, de
    clrb a
    mov a, x
    mov [sp+0x00], a
    movw ax, hl
    addw ax, bc
    movw bc, ax
    mov a, [sp+0x00]
    xchw ax, bc
    cmp0 b
    bz $.BB@LABEL@1_2
.BB@LABEL@1_1: ; entry
    sarw ax, 0x01
    dec b
    bnz $.BB@LABEL@1_1
.BB@LABEL@1_2: ; entry
    pop hl
    ret
</pre></td><td>アセンブリ展開コード

<pre>
_func:
    .STACK _func = 4
    add x, a
    mov a, x
    shrw ax, 8+0x00000
    cmp0 c
    bz $.BB@LABEL@1_2
.BB@LABEL@1_1: ; entry
    shrw ax, 0x01
    dec c
    bnz $.BB@LABEL@1_1
.BB@LABEL@1_2: ; entry
    mov a, x
    ret
</pre></td></tr>
<tr><td>コードサイズ:23 バイト
クロック数:32 クロック</td><td>コードサイズ:15 バイト
クロック数:25 クロック</td></tr>
</table>

尚、配列の要素に変数を記述する場合、アドレス計算時に int 拡張することがあるため、16bit の型を用いた方が出力コードは改善する場合があります。

変更前	変更後

C ソース
 	
`char array[100];`
`char index = 10;`
 	
`char func(void)`
`{`
`  return array[index-2];`
`}`
	C ソース
	
	`char array[100];`
	`int index = 10;`
	
	`char func(void)`
	`{`
	`  return array[index-2];`
	`}`

アセンブリ展開コード
 	
`_func:`
`  .STACK _func = 4`
`  mov a, !LOWW(_index)`
`  shrw ax, 8+0x00000`
`  addw ax, #LOWW(_array+0x0FFFE)`
`  movw de, ax`
`  mov a, [de]`
`  ret`
	アセンブリ展開コード
	
	`_func:`
	`  .STACK _func = 4`
	`  movw ax, #LOWW(_array+0x0FFFE)`
	`  addw ax, !LOWW(_index)`
	`  movw de, ax`
	`  mov a, [de]`
	`  ret`

コードサイズ:11 バイト クロック数:13 クロック	コードサイズ:9 バイト クロック数:12 クロック

for 文のループカウンタなど比較演算を伴う場合、変数の取り得る値が 0 以上であることが分かっているならば、unsigned 型を用いてください。RL78 ファミリの比較演算命令は符号無しで比較するためです。

変更前	変更後

C ソース
 	
`int value;`
`void func(int num)`
`{`
`  int i;`
`  for (i = 0; i < num; i++) {`
`  value += 10;`
`  }`
`}`
	C ソース
	
	`int value;`
	`void func(unsigned int num)`
	`{`
	`  unsigned int i;`
	`  for (i = 0; i < num; i++) {`
	`  value += 10;`
	`  }`
	`}`

アセンブリ展開コード
 	
`_func:`
`  .STACK _func = 4`
`  movw de, ax`
`  clrw bc`
`.BB@LABEL@1_1: ; bb6`
`  movw ax, de`
`  xor a, #0x80`
`  movw hl, ax`
`  movw ax, bc`
`  xor a, #0x80`
`  cmpw ax, hl`
`  bnc $.BB@LABEL@1_3`
`.BB@LABEL@1_2: ; bb`
`  movw ax, #0x000A`
`  addw ax, !LOWW(_value)`
`  movw !LOWW(_value), ax`
`  incw bc`
`  br $.BB@LABEL@1_1`
`.BB@LABEL@1_3: ; return`
`  ret`
	アセンブリ展開コード
	
	`_func:`
	`  .STACK _func = 4`
	`  movw bc, ax`
	`  clrw ax`
	`.BB@LABEL@1_1: ; entry`
	`  movw de, ax`
	`.BB@LABEL@1_2: ; bb6`
	`  cmpw ax, bc`
	`  bz $.BB@LABEL@1_4`
	`.BB@LABEL@1_3: ; bb`
	`  movw ax, #0x000A`
	`  addw ax, !LOWW(_value)`
	`  movw !LOWW(_value), ax`
	`  movw ax, de`
	`  incw ax`
	`  br $.BB@LABEL@1_1`
	`.BB@LABEL@1_4: ; return`
	`  ret`

コードサイズ:25 バイト クロック数:168 クロック	コードサイズ:20 バイト クロック数:134 クロック

5.12　switch 文の共通な case の処理をまとめる

複数の case ラベルの分岐先の処理が同一である場合、case ラベルを移動して処理を 1 箇所にまとめてください。

変更前	変更後

C ソース
 	
`void func(void)`
`{`
`  switch(x) {`
`  case 0:`
`  dummy1();`
`  break;`
`  case 1:`
`  dummy1();`
`  break;`
`  case 2:`
`  dummy1();`
`  break;`
`  case 3:`
`  dummy2();`
`  break;`
`  case 4:`
`  dummy2();`
`  break;`
`  default:`
`  break;`
`  }`
`}`
	C ソース
	
	`void func(void)`
	`{`
	`  switch(x) {`
	`  case 0:`
	`  case 1:`
	`  case 2:`
	`  dummy1();`
	`  break;`
	`  case 3:`
	`  case 4:`
	`  dummy2();`
	`  break;`
	`  default:`
	`  break;`
	`  }`
	`}`

アセンブリ展開コード	アセンブリ展開コード

_func:
　.STACK _func = 4
　movw ax, !LOWW(_x)
　cmpw ax, #0x0000
　bz $.BB@LABEL@3_6
.BB@LABEL@3_1: ; entry
　addw ax, #0xFFFF
　bz $.BB@LABEL@3_6
.BB@LABEL@3_2: ; entry
　addw ax, #0xFFFF
　bz $.BB@LABEL@3_6
.BB@LABEL@3_3: ; entry
　addw ax, #0xFFFF
　bz $.BB@LABEL@3_7
.BB@LABEL@3_4: ; entry
　cmpw ax, #0x0001
　bz $.BB@LABEL@3_7
.BB@LABEL@3_5: ; return
　ret
.BB@LABEL@3_6: ; switch_clause_bb2
　br $_dummy1
.BB@LABEL@3_7: ; switch_clause_bb4
　br $_dummy2
	_func:
	　.STACK _func = 4
	　movw ax, !LOWW(_x)
	　cmpw ax, #0x0003
	　bc $.BB@LABEL@3_3
	.BB@LABEL@3_1: ; entry
	　addw ax, #0xFFFD
	　cmpw ax, #0x0002
	　bc $.BB@LABEL@3_4
	.BB@LABEL@3_2: ; return
	　ret
	.BB@LABEL@3_3: ; switch_clause_bb
	　br $_dummy1
	.BB@LABEL@3_4: ; switch_clause_bb1
	　br $_dummy2

コードサイズ:33 バイト クロック数:23 クロック	コードサイズ:21 バイト クロック数:17 クロック

5.13　for ループを do while ループに置き換える

1 回以上ループを実行することが分かっている for 文の場合、do while 文に置き換えることでコードサイズが小さくなる可能性があります。また、条件式を等価比較に置き換えることでもコードサイズが小さくなる可能性があります。

変更前	変更後

C ソース
 	
`int array[10][10];`
 	
`void func(int nsize, int msize)`
`{`
`  int i;`
`  int *p;`
`  int s;`
`  p = &array[0][0];`
`  s = nsize * msize;`
`  for (i = 0; i < s; i++) {`
`  *p++ = 0;`
`  }`
`}`
	C ソース
	
	`int array[10][10];`
	
	`void func(int nsize, int msize)`
	`{`
	`  int i;`
	`  int *p;`
	`  int s;`
	`  p = &array[0][0];`
	`  s = nsize * msize;`
	`  i = 0;`
	`  do {`
	`  *p++ = 0;`
	`  i++;`
	`  } while (i != s);`
	`}`

アセンブリ展開コード	アセンブリ展開コード

_func:
　.STACK _func = 8
　push hl
　push bc
　pop de
　movw bc, ax
　movw ax, de
　mulh
　movw [sp+0x00], ax
　clrw bc
　movw de, #LOWW(_array)
.BB@LABEL@1_1: ; bb13
　xor a, #0x80
　movw hl, ax
　movw ax, bc
　xor a, #0x80
　cmpw ax, hl
　bnc $.BB@LABEL@1_3
.BB@LABEL@1_2: ; bb
　clrw ax
　movw [de], ax
　incw de
　incw de
　incw bc
　movw ax, [sp+0x00]
　br $.BB@LABEL@1_1
.BB@LABEL@1_3: ; return
　pop hl
　ret
	_func:
	　.STACK _func = 6
	　push bc
	　pop de
	　movw bc, ax
	　movw ax, de
	　mulh
	　movw bc, ax
	　movw de, #LOWW(_array)
	.BB@LABEL@1_1: ; bb
	　clrw ax
	　movw [de], ax
	　movw ax, bc
	　addw ax, #0xFFFF
	　movw bc, ax
	　incw de
	　incw de
	　bnz $.BB@LABEL@1_1
	.BB@LABEL@1_2: ; return
	　ret

コードサイズ:34 バイト クロック数:1626 クロック	コードサイズ:23 バイト クロック数:1112 クロック

5.14　2 のべき乗の除算はシフト演算に置き換える

除算の除数が 2 のべき乗である場合は、除算をシフト演算に置き換えてください。

変更前	変更後

C ソース
 	
`int s;`
`void func(void)`
`{`
`  s = s / 2;`
`}`
	C ソース
	
	`int s;`
	`void func(void)`
	`{`
	`  s = s >> 1;`
	`}`

アセンブリ展開コード
 	
`_func:`
`  .STACK _func = 4`
`  movw ax, !LOWW(_s)`
`  onew bc`
`  incw bc`
`  call !!__COM_sidiv`
`  movw !LOWW(_s), ax`
`  ret`
	アセンブリ展開コード
	
	`_func:`
	`  .STACK _func = 4`
	`  movw ax, !LOWW(_s)`
	`  sarw ax, 0x01`
	`  movw !LOWW(_s), ax`
	`  ret`

コードサイズ:13 バイト クロック数:75 クロック	コードサイズ:9 バイト クロック数:9 クロック

5.15　2 ビット以上のビットフィールドは char 型に変更する

ビットフィールドのメンバのサイズが 2 ビット以上の場合は、ビットフィールドは使用せずに char 型に変更してください。ただし、ROM サイズは増加します。

変更前	変更後

C ソース
 	
`struct {`
`  unsigned char b0:1;`
`  unsigned char b1:2;`
`} dw;`
 	
`unsigned char dummy;`
 	
`int func(void)`
`{`
`  if (dw.b1) {`
`  dummy++;`
`  }`
`  return (0);`
`}`
	C ソース
	
	`struct {`
	`  unsigned char b0:1;`
	`  unsigned char b1;`
	`} dw;`
	
	`unsigned char dummy;`
	
	`int func(void)`
	`{`
	`  if (dw.b1) {`
	`  dummy++;`
	`  }`
	`  return (0);`
	`}`

アセンブリ展開コード
 	
`_func:`
`  .STACK _func = 4`
`  mov a, #0x06`
`  and a, !LOWW(_dw)`
`  bnz $.BB@LABEL@1_2`
`.BB@LABEL@1_1: ; if_break_bb`
`  clrw ax`
`  ret`
`.BB@LABEL@1_2: ; if_then_bb`
`  inc !LOWW(_dummy)`
`  br $.BB@LABEL@1_1`
	アセンブリ展開コード
	
	`_func:`
	`  .STACK _func = 4`
	`  cmp0 !LOWW(_dw+0x00001)`
	`  bnz $.BB@LABEL@1_2`
	`.BB@LABEL@1_1: ; if_break_bb`
	`  clrw ax`
	`  ret`
	`.BB@LABEL@1_2: ; if_then_bb`
	`  inc !LOWW(_dummy)`
	`  br $.BB@LABEL@1_1`

コードサイズ:13 バイト クロック数:11 クロック ROM サイズ:1 バイト	コードサイズ:12 バイト クロック数:10 クロック ROM サイズ:2 バイト

5.16　構造体のアライメント

構造体を定義する際には、アライメントを意識してメンバを宣言してください。アライメントとは、変数を効率よくアクセスするために、変数の配置アドレスに制約を設けることです。

例えば、long 型のアライメント数は 2 バイトであり、long 型の変数は 2 の倍数アドレスに配置する必要があります。構造体変数では、メンバと構造体変数の両方にアライメントが設定されます。メンバのアライメント数は、メンバでない変数のアライメント数と同じです。構造体変数のアライメント数は、メンバのアライメント数の最大値です。構造体変数の中のメンバを隙間なく配置するとアライメントに違反してしまう場合、隙間を作って配置アドレスを調整します。この隙間のことをパディングと言います。また、構造体変数のサイズがアライメント数の倍数でない場合もパディングが生じます。パディングが増えるとメモリの利用効率が下がってしまいます。

変更前	変更後

C ソース
 	
`// 最大メンバが long 型の為、`
`// アライメント数は 2byte`
`struct str {`
`  char c1;  // 1byte`
`// パディング分  1byte`
`  long l1;  // 4byte`
`  char c2;  // 1byte`
`  char c3;  // 1byte`
`  char c4;  // 1byte`
`// パディング分  1byte`
`} str1;`
	C ソース
	
	`// 最大メンバが long 型の為、`
	`// アライメント数は 2byte`
	`struct str {`
	`  char c1;  // 1byte`
	`  char c2;  // 1byte`
	`  char c3;  // 1byte`
	`  char c4;  // 1byte`
	`  long l1;  // 4byte`
	`} str1;`

アセンブリ展開コード
 	
`_str1:`
`  .DS (10)`
	アセンブリ展開コード
	
	`_str1:`
	`  .DS (8)`

RAM サイズ:10 バイト	RAM サイズ:8 バイト

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/contact/

改訂記録

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2018.11.26		新規発行

ご注意書き

1. 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・システムの設計において、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て生じた損害（お客様または第三者いずれに生じた損害も含みます。以下同じです。）に関し、当社は、一切その責任を負いません。
2. 当社製品、本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害またはこれらに関する紛争について、当社は、何らの保証を行うものではなく、また責任を負うものではありません。
3. 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。
4. 当社製品を、全部または一部を問わず、改造、改変、複製、リバースエンジニアリング、その他、不適切に使用しないでください。かかる改造、改変、複製、リバースエンジニアリング等により生じた損害に関し、当社は、一切その責任を負いません。
5. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、各品質水準は、以下に示す用途に製品が使用されることを意図しております。

   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、産業用ロボット等

   高品質水準：　輸送機器（自動車、電車、船舶等）、交通制御（信号）、大規模通信機器、
   金融端末基幹システム、各種安全制御装置等

   当社製品は、データシート等により高信頼性、Harsh environment向け製品と定義しているものを除き、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（宇宙機器と、海底中継器、原子力制御システム、航空機制御システム、プラント基幹システム、軍事機器等）に使用されることを意図しておらず、これらの用途に使用することは想定していません。たとえ、当社が想定していない用途に当社製品を使用したことにより損害が生じても、当社は一切その責任を負いません。
6. 当社製品をご使用の際は、最新の製品情報（データシート、ユーザーズマニュアル、アプリケーションノート、信頼性ハンドブックに記載の「半導体デバイスの使用上の一般的な注意事項」等）をご確認の上、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他指定条件の範囲内でご使用ください。指定条件の範囲を超えて当社製品をご使用された場合の故障、誤動作の不具合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。
7. 当社は、当社製品の品質および信頼性の向上に努めていますが、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、データシート等において高信頼性、Harsh environment向け製品と定義しているものを除き、耐放射線設計を行っておりません。仮に当社製品の故障または誤動作が生じた場合であっても、人身事故、火災事故その他社会的損害等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、お客様の機器・システムとしての出荷保証を行ってください。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、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
8. 当社製品の環境適合性等の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。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。
9. 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・システムに使用することはできません。当社製品および技術を輸出、販売または移転等する場合は、「外国為替及び外国貿易法」その他日本国および適用される外国の輸出管理関連法規を遵守し、それらの定めるところに従い必要な手続きを行ってください。
10. お客様が当社製品を第三者に転売等される場合には、事前に当該第三者に対して、本ご注意書き記載の諸条件を通知する責任を負うものといたします。
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禁じます。
12. 本資料に記載されている内容または当社製品についてご不明な点がございましたら、当社の営業担当者までお問合せください。

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直接的、間接的に支配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。

(Rev.4.0-1　2017.11)

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 エレクトロニクス株式会社　〒135-0061　東京都江東区豊洲3-2-24（豊洲フォレシア）

■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：https://www.renesas.com/contact/