RL78 ファミリ用 C コンパイラ CC-RL

プログラミングテクニック

R01AN3184JJ0110
Rev.1.10
2017.04.10

要旨

本アプリケーションノートでは、C コンパイラ CC-RL 使用時のコードサイズの削減、実行速度の高速化およびバグ回避のプログラミングテクニックについて説明します。

開発統合環境の対象バージョンは以下の通りです。
・CS+ V4.01.00
・e^2 studio V4.0.0.26
・RL78 ファミリ用 C コンパイラ CC-RL V1.03.00

対象デバイス

RL78 ファミリ

目次

1.　コードサイズの削減

1.1　変数のサイズ

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。

変更前	変更後

void main(void)
{
　signed int i;
 	
　for (i=0; i < 10; i++)
　{
　　NOP();
　}
}
	void main(void)
	{
	　signed char i;
	
	　for (i=0; i < 10; i++)
	　{
	　　NOP();
	　}
	}

図 1.1　C ソースコード

変更前		変更後	
movw ax, #0x000A	3	mov a, #0x0A	2
.BB@LABEL@1_1:		.BB@LABEL@1_1:	
nop	1	nop	1
addw ax, #0xFFFF	3	dec a	1
bnz $.BB@LABEL@1_1	2	bnz $.BB@LABEL@1_1	2
.BB@LABEL@1_2:		.BB@LABEL@1_2:	
ret	1	ret	1
10 バイト		7 バイト	

図 1.2　出力アセンブラ

1.2　符号なし変数

負数を扱わないデータは、全て unsigned を付けてください。
RL78 ファミリが unsigned を得意としたデバイスであるためです。

変更前	変更後

signed int data0;
signed int data1;
 	
void main(void)
{
　if (data0 > 10)
　{
　　data1++;
　}
}
	unsigned int data0;
	unsigned int data1;
	
	void main(void)
	{
	　if (data0 > 10)
	　{
	　　data1++;
	　}
	}

図 1.3　C ソースコード

変更前		変更後	
movw ax, !LOWW(_data0)	3	movw ax, !LOWW(_data0)	3
xor a, #0x80	2		
cmpw ax, #0x800B	3	cmpw ax, #0x000B	3
skc	2	skc	2
.BB@LABEL@1_1:		.BB@LABEL@1_1:	
incw !LOWW(_data1)	3	incw !LOWW(_data1)	3
13 バイト		11 バイト	

図 1.4　出力アセンブラ

1.3　saddr 領域

使用頻度の高いグローバル変数、関数内 static 変数は、__saddr 修飾子、#pragma saddr 宣言等を使用してください。

saddr 領域に割り当てると、コード効率の良いコードとなります。

特に 1 ビットのビットフィールドは__saddr 修飾子､#pragma saddr 宣言の効果が大きくなる傾向にあります。

変数／関数情報ファイルでも saddr 領域への変数の指定が可能です。

変更前	変更後

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 main(void)
{
　data0.b4 = data1.b1;
}
	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 main(void)
	{
	　data0.b4 = data1.b1;
	}

図 1.5　C ソースコード

変更前		変更後	

movw hl, #LOWW(_data1)
mov1 CY, [hl].1
movw hl, #LOWW(_data0)
mov1 [hl].4, CY
	3
	2
	3
	2
		mov1 CY, _data1.1
		
		mov1 _data0.4, CY
			3
			
			3

10 バイト		6 バイト	

図 1.6　出力アセンブラ

1.4　callt 関数

関数呼び出し頻度の高い関数は、__callt 修飾子、#pragma callt 宣言等を使用してください。
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て直接関数をコールするよりも、短いコードで関数をコールすることができます。

変更前	変更後

void func_sub(void)
{
　…
}
 	
void func(void)
{
　func_sub();
　　:
　func_sub();
}
	__callt void func_sub(void)
	{
	　…
	}
	
	void func()
	{
	　func_sub();
	　　:
	　func_sub();
	}

図 1.7　C ソースコード

変更前		変更後	
		.SECTION .callt0,CALLT0 @_func_sub: 　.DB2 _func_sub	 2
.SECTION .textf,TEXTF _func: 　call $!_func_sub	 4	.SECTION .textf,TEXTF _func: 　callt [@_func_sub]	 2
call $!_func_sub	4	callt [@_func_sub]	2
8 バイト		6 バイト	

図 1.8　出力アセンブラ

注意
・呼び出し用の関数のアドレステーブルを生成します。(.callt0)
・1 回しか呼び出されない関数の場合には、コードサイズ削減の効果はありません。
・CALLT 命令は CALL 命令よりも実行クロック数は多くなります。
・変数／関数情報ファイルでも CALLT 命令で呼び出す関数の宣言の指定が可能です。

1.5　構造体メンバのアライメント

RL78 ファミリは奇数番地からワード・データのリード／ライトができないため、デフォルトオプションでは 2 バイト以上のメンバが偶数番地に配置されるようにアライン・データを挿入します。

したがって、構造体のメンバはアライメントを考慮し、無駄な隙間を作らないように配置してください。

変更前	変更後
struct { 　signed char a; 　signed int b; 　signed char c; 　struct 　{ 　　signed int d; 　　signed int e; 　} f; } data;	struct { 　signed char a; 　signed char c; 　signed int b; 　struct 　{ 　　signed int d; 　　signed int e; 　} f; } data;

図 1.9　C ソースコード

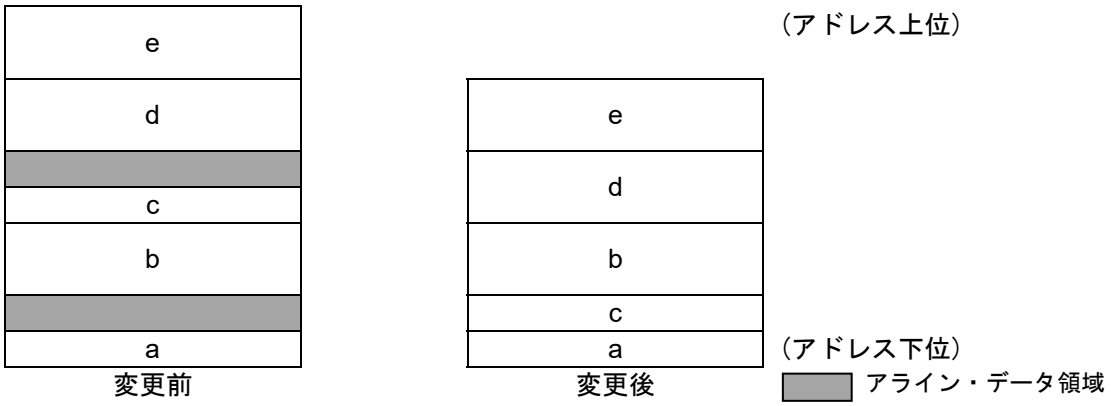

図 1.10　メモリ配置

1.6　ビットフィールドと 1 バイト変数

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、使用する RAM 容量は増加します。

変更前	変更後

struct
{
　unsigned char b0:1;
　unsigned char b1:2;
} data;
 	
unsigned char dummy;
 	
if (data.b1)
{
　dummy++;
}
	unsigned char data;
	unsigned char dummy;
	
	if (data)
	{
	　dummy++;
	}

図 1.11　C ソースコード

変更前		変更後	
mov a, #0x06	2		
and a, !LOWW(_data)	3	comp0, !LOWW(_data)	3
sknz	2	sknz	2
ret	1	ret	1
inc, !LOWW(_dummy)	3	inc, !LOWW(_dummy)	3
ret	1	ret	1
12 バイト		10 バイト	

図 1.12　出力アセンブラ

1.7　型変換

short 型、char 型の変数は演算時に int 型に拡張され、unsigned short 型、unsigned char 型の変数は演算時に unsigned int 型に拡張されます。そのため、これらの変数を使用したプログラムでは、型変換を行う命令が多く生成されます。プログラミング時に型変換をしておくと、型変換を行う命令が生成されないため、コードサイズが削減されます。

変更前	変更後

void main(void)
{
　unsigned char i;
 	
　for (i = 0; i <4; i++)
　{
　　array[2 + i] = *(p + i);
　}
}
	void main(void)
	{
	　int i;
	
	　for (i = 0; i <4; i++)
	　{
	　　array[2 + i] = *(p + i);
	　}
	}

図 1.13　C ソースコード

備考 array[], *p はグローバル変数。

変更前		変更後	
		movw de, #LOWW(_array+0x00004)	3
clrb a	1	clrw ax	1
.BB@LABEL@1_1:	2	.BB@LABEL@1_1:	
mov x, #0x02	2	movw bc, ax	2
mov b, a	1	addw ax, !LOWW(_p)	3
mulu x	2	movw hl, ax	1
movw de, ax	2	movw ax, [hl]	1
addw ax, #LOWW(_array+0x00004)	2	movw [de], ax	1
movw hl, ax	2	incw bc	1
movw ax, de	3	incw bc	1
addw ax, !LOWW(_p)	2	movw ax, bc	2
movw de, ax	1	cmpw ax, #0x0008	3
movw ax, [de]	1	incw de	1
movw [hl], ax	1	incw de	1
inc b	2	bnz $.BB@LABEL@1_1	2
mov a, b	3		
cmp a, #0x04	2		
bnz $.BB@LABEL@1_1			
29 バイト		23 バイト	

図 1.14　出力アセンブラ

1.8　帰納変数の削除

ループの制御を行う変数を帰納変数（誘導変数）といいます。ループの制御を他の変数を用いて行うと、帰納変数が削除されるため、コードサイズが削減されます。

変更前	変更後

int main(void)
{
　int i;
 	
　for (i = 0; *(table + i) != 0; ++i)
　{
　　if (x== (*(table + i) & 0xFF))
　　{
　　　return(*(table +i) & 0xFF00);
　　}
　}
}
	int main(void)
	{
	　const unsigned short *p;
	
	　for (p = table; *p != 0; ++p)
	　{
	　　if (x == (*p & 0xFF))
	　　{
	　　　return(*p & 0xFF00);
	　　}
	　}
	}

図 1.15　C ソースコード

備考 x, *table はグローバル変数。

変更前		変更後	
subw sp, #0x06	2		
movw hl, sp	3		
clrw ax	1		
movw [hl], ax	1		
movw [sp+0x02], ax	2	movw de, !LOWW(_table)	3
.BB@LABEL@1_1:		.BB@LABEL@1_1:	
movw bc, !LOWW(_table)	3	movw ax, [de]	1
movw ax, bc	1	movw bc, ax	1
movw [sp+0x04], ax	2		
movw ax, [hl]	1		
addw ax, bc	1		
movw de, ax	3		
movw ax, [de]	1		
cmpw ax, #0x0000	3	cmpw ax, #0x0000	3
bz $.BB@LABEL@1_5	2	bz $.BB@LABEL@1_5	2
.BB@LABEL@1_2:		.BB@LABEL@1_2:	
clrb a	1	clrb a	1
cmpw ax, !LOWW(_x)	3	cmpw ax, !LOWW(_x)	3
bz $.BB@LABEL@1_4	2	bz $.BB@LABEL@1_4	2
.BB@LABEL@1_3:		.BB@LABEL@1_3:	
movw ax, [hl]	1		1
incw ax	1	incw de	1
incw ax	1	incw de	
movw [hl], ax	1		
incw [hl+0x02]	3		
br $.BB@LABEL@1_1	2	br $.BB@LABEL@1_1	2
.BB@LABEL@1_4:		.BB@LABEL@1_4:	
movw ax, [sp+0x02]	2		
movw bc, ax	1		
shlw bc, 0x01	2		
movw ax, [sp+0x04]	2		
addw ax, bc	1		
movw de, ax	3		
movw ax, [de]	1	movw ax, bc	1
clrb x	1	clrb x	1
addw sp, #0x06	2		
ret	1	ret	1
.BB@LABEL@1_5:		.BB@LABEL@1_5:	
clrw ax	1	clrw ax	1
addw sp, #0x06	2		
60 バイト		24 バイト	

図 1.16　出力アセンブラ

1.9　ループの統合

同じ関数内にある異なるループ文を１つにまとめてループ文の数を減らすことをループの統合と言います。
ループの統合により、コードサイズの削減が可能です。また、ループの繰り返しによるオーバーヘッドを取り除くことで、実行速度の高速化も行えます。

変更前	変更後

void main(void)
{
　uint8_t i = 0;
　uint8_t total = 0;
　uint8_t test[10] = {0};
 	
　for (i = 0; i < 10; i++)
　{
　　test[i] = CSS;
　}
　for (i = 0; i < 10; i++)
　{
　　total += test[i];
　}
}
	void main(void)
	{
	　uint8_t i = 0;
	　uint8_t total = 0;
	　uint8_t test[10] = {0};
	
	　for (i = 0; i < 10; i++)
	　{
	　　test[i] = CSS;
	　　total += test[i];
	　}
	}

図 1.17　C ソースコード

変更前		変更後	
subw sp, #0x0C	2	subw sp, #0x0C	2
movw de, #0x000A	3	movw de, #0x000A	3
clrw bc	1	clrw bc	1
movw ax, sp	2	movw ax, sp	2
incw ax	1	incw ax	1
incw ax	1	incw ax	1
movw [sp+0x00], ax	2	movw [sp+0x00], ax	2
call !!_memset	4	call !!_memset	4
mov [sp+0x02], #0x00	3	mov [sp+0x02], #0x00	3
clrb b	1	clrb b	1
.BB@LABEL@1_1:		.BB@LABEL@1_1:	
mov a, 0xFFFA4	3	mov a, 0xFFFA4	3
shr a, 0x06	2	shr a, 0x06	2
and a, #0x01	2	and a, #0x01	2
mov c, a	1	mov c, a	1
pop hl	1	pop hl	1
push hl	1	push hl	1
mov a, c	1	mov a, c	1
mov [hl+b], a	2	mov [hl+b], a	2
inc b	1	inc b	1
mov a, b	1	mov a, b	1
cmp a, #0x0A	2	cmp a, #0x0A	2
bnz $.BB@LABEL@1_1	2	bnz $.BB@LABEL@1_1	2
.BB@LABEL@1_2:			
mov a, #0x0A	2		
.BB@LABEL@1_3:			
dec a	1		
bnz $.BB@LABEL@1_3	2		
.BB@LABEL@1_4:		.BB@LABEL@1_2:	
addw sp, #0x0C	2	addw sp, #0x0C	2
ret	1	ret	1
47 バイト		42 バイト	

図 1.18　出力アセンブラ

1.10　メモリモデル

RL78 ファミリには、アドレス長を 16 ビットとしてコード生成するスモールモデルとアドレス長を 20 ビットとしてコード生成するミディアムモデルがあります。

モデル	サイズ	関数	変数
スモールモデル	プログラム 64K バイト以下（データ 64K バイト以下）	near	near
ミディアムモデル	プログラム 64K バイト超　（データ 64K バイト以下）	far	near

図 1.19　メモリモデルの種類

プログラムが 64K バイトを超える場合は、ミディアムモデルを選択してください。このとき、関数呼び出し頻度の高い関数に__near 修飾子を付加するとコードサイズを削減できます。

ただし、__near 修飾子、__far 修飾子を付加した場合は、それらを取り扱うポインタ変数の型を合わせる必要があります。

2. 実行速度の高速化

2.1 配列への連続アクセス

ループ内で配列に連続アクセスする場合は、ポインタ変数を使用してください。ポインタ変数を使用しない場合、配列の添え字から実アドレスを求める処理が毎回出力され実行速度が遅くなる可能性があります。

注 本章のプログラムの実行時間計測は全て開発統合環境 CS+の RL78 シミュレータを用いて行っています。

変更前	変更後

int i;
 	
sum = 0;
 	
for (i = 0; i < 10; i++)
{
　　sum += array[i];
}
	int i;
	int *p;
	
	sum = 0;
	p = &array[0];
	
	for (i = 0; i < 10; i++)
	{
	　　sum += *p++;
	}

図 2.1　C ソースコード例

備考 sum, array[]はグローバル変数。

変更前		変更後	
clrb a	1	mov d, #0x00	2
mov c, a	1	movw hl, #LOWW(_array)	3
mov b, a	1	movw bc, #0x000A	3
.BB@LABEL@1_1:		.BB@LABEL@1_1:	
mov a, c	1	mov a, d	1
shrw ax, 8+0x00000	2		
addw ax, #LOWW(_array)	3		
movw hl, ax	3		
mov a, b	1		
add a, [hl]	1	add a, [hl]	1
mov b, a	1	mov d, a	1
mov !LOWW(_sum), a	3	mov !LOWW(_sum), a	3
mov a, c	1	movw ax, bc	1
inc a	1	addw ax, #0xFFFF	3
mov c, a	1	movw bc, ax	3
cmp a, #0x0A	2	incw hl	1
bnz $.BB@LABEL@1_1	2	bnz $.BB@LABEL@1_1	2
.BB@LABEL@1_2:		.BB@LABEL@1_2:	
ret	1	ret	1
26 バイト		25 バイト	
実行時間：636 サイクル / 19.875μsec. (32MHz 時)		実行時間：476 サイクル / 14.875μsec. (32MHz 時)	

図 2.2　出力アセンブラ

2.2　グローバル変数へのアクセス

ループ内では可能な限りグローバル変数を使用しないようにしてください。
アドレス計算やメモリ・アクセス(ロード／ストア命令)が毎回出力される可能性があるため、ローカル変数に置き換えてください。

変更前	変更後

int i;
int *p;
 	
sum = 0;
p = &array[0];
 	
for (i = 0; i < 10; i++)
{
　　sum += *p++;
}
	int i;
	int *p;
	int tmp;
	
	tmp = 0;
	p = &array[0];
	
	for (i = 0; i < 10; i++)
	{
	　　tmp += *p++;
	}
	
	sum = tmp;

図 2.3　C ソースコード例

備考　sum, array[]はグローバル変数。

変更前		変更後	
mov d, #0x00	2	mov d, #0x00	2
movw hl, #LOWW(_array)	3	movw hl, #LOWW(_array)	3
movw bc, #0x000A	3	movw bc, #0x000A	3
.BB@LABEL@1_1:		.BB@LABEL@1_1:	
mov a, d	1	mov a, d	1
add a, [hl]	1	add a, [hl]	1
mov d, a	1	mov d, a	1
mov !LOWW(_sum), a	3		
movw ax, bc	1	movw ax, bc	1
addw ax, #0xFFFF	3	addw ax, #0xFFFF	3
movw bc, ax	3	movw bc, ax	3
incw hl	1	incw hl	1
bnz $.BB@LABEL@1_1	2	bnz $.BB@LABEL@1_1	2
.BB@LABEL@1_2:		.BB@LABEL@1_2:	
		mov a, d	1
		mov !LOWW(_sum), a	3
ret	1	ret	1
25 バイト		26 バイト	
実行時間：476 サイクル / 14.875μsec. (32MHz 時)		実行時間：444 サイクル / 13.875μsec. (32MHz 時)	

図 2.4　出力アセンブラ

2.3　ループ内の if 文

ループ内の処理にできるだけ if 文を使用しないで記述してください。
ループ毎に if 文の処理が出力され、実行速度が遅くなる可能性があります。

変更前	変更後

int i = 0;
int j = 0;
 	
for (i = 0; i < N; i++)
{
　for (j = 0; j < N; j++)
　{
　　if (i != j)
　　{
　　　A[i][j] = B[i][j];
　　}
　　else
　　{
　　　A[i][j] = 1;
　　}
　}
}
	int i = 0;
	int j = 0;
	
	for (i = 0; i < N; i++)
	{
	　for (j = 0; j < N; j++)
	　{
	　　A[i][j] = B[i][j];
	　}
	}
	
	for (i=0; i < N; i++)
	{
	　A[i][i] = 1U;
	}

図 2.5　C ソースコード例

備考 A[][],B[][]はグローバル変数で、N は 10 であるとします。

変更前		変更後	
push hl	1	mov d, #0x00	2
clrb a	1	.BB@LABEL@1_1:	
br $.BB@LABEL@1_9	2	mov a, !LOWW(_N)	3
.BB@LABEL@1_1: ; bb44		mov h, a	1
cmp a, !LOWW(_N)	3	cmp d, a	2
bnc $.BB@LABEL@1_10	2	bc $.BB@LABEL@1_6	2
.BB@LABEL@1_2:		.BB@LABEL@1_2:	
clrb a	1	clrb a	1
.BB@LABEL@1_3:		.BB@LABEL@1_3:	
mov e, a	1	mov l, a	1
.BB@LABEL@1_4:		.BB@LABEL@1_4:	
cmp a, !LOWW(_N)	3	cmp a, h	2
bnc $.BB@LABEL@1_8	2	bnc $.BB@LABEL@1_11	2
.BB@LABEL@1_5:		.BB@LABEL@1_5	
mov a, d	1	shrw ax, 8+0x00000	2
mov x, #0x0A	2	movw de, ax	3
mulu x	1	push de	1
movw bc, ax	3	pop bc	1
mov a, e	1	shlw bc, 0x03	2
shrw ax, 8+0x00000	2	movw ax, de	2
addw ax, bc	1	addw ax, ax	1
movw [sp+0x00], ax	2	addw ax, bc	1
mov a, e	1	addw ax, #LOWW(_A)	3
cmp d, a	2	addw ax, de	1
oneb b	1	movw de, ax	2
bz $.BB@LABEL@1_7	2	mov [de+0x00], #0x01	3
.BB@LABEL@1_6:		mov a, l	1
pop bc	1	inc a	1
push bc	1	br $.BB@LABEL@1_3	2
mov a, LOWW(_B)[bc]	3	.BB@LABEL@1_6:	
mov b, a	1	clrb a	1
.BB@LABEL@1_7:		.BB@LABEL@1_7:	
movw ax, [sp+0x00]	2	mov e, a	1
xchw ax, bc	1	.BB@LABEL@1_8:	
mov LOWW(_A)[bc], a	3	cmp a, !LOWW(_N)	3
mov a, e	1	bnc $.BB@LABEL@1_10	2
inc a	1	.BB@LABEL@1_9:	
br $.BB@LABEL@1_3	2	mov a, d	1
.BB@LABEL@1_8:		mov x, #0x0A	2
mov a, d	1	mulu x	1
inc a	1	movw bc, ax	3
.BB@LABEL@1_9		mov a, e	1
mov d, a	1	shrw ax, 8+0x00000	2
br $.BB@LABEL@1_1	2	addw ax, bc	1
.BB@LABEL@1_10:		movw bc, ax	3
pop hl	1	mov a, LOWW(_B)[bc]	3
ret	1	mov LOWW(_A)[bc], a	3
		mov a, e	1
		inc a	1
		br $.BB@LABEL@1_7	2
		.BB@LABEL@1_10:	
		inc d	1
		br $.BB@LABEL@1_1	2
		.BB@LABEL@1_11:	
		ret	1
56 バイト		77 バイト	
実行時間：12600 サイクル /393.75μsec. (32MHz 時)		実行時間：9596 サイクル / 299.875μsec. (32MHz 時)	

図 2.6　出力アセンブラ

2.4　ループの終了条件

ループの終了条件に 0 との比較式を使用すると、ループ 1 回ごとの終了条件の演算が速くなる可能性があります。また、使用するレジスタ数が減る可能性もあります。

変更前	変更後

int i;
int Height;
int Width;
int *p;
int s;
 	
p = &array[0][0];
s = Height * Width;
for (i = 0; i < s; i++)
{
　　*p++ = 0;
}
	int i;
	int Height
	int Width
	int *p;
	
	p = &array[0][0];
	for (i = Height * Width; i > 0; i--)
	{
	　　*p++ = 0;
	}

図 2.7　C ソースコード例

備考 array[][]はグローバル変数

変更前		変更後	
movw de, #LOWW(_array)	3	movw de, #LOWW(_array)	3
clrw ax	1		
.BB@LABEL@1_1:			
cmpw ax, #0x0032	3	movw ax, #0x0032	3
bz $.BB@LABEL@1_3	2		
.BB@LABEL@1_2:		.BB@LABEL@1_1:	
mov [de+0x00], #0x00	3	mov [de+0x00], #0x00	3
incw ax	1	addw ax, #0xFFFF	3
incw de	1	incw de	1
br $.BB@LABEL@1_1	2	bnz $.BB@LABEL@1_1	2
.BB@LABEL@1_3:		.BB@LABEL@1_2:	
ret	1	ret	1
17 バイト		16 バイト	
実行時間：1812 サイクル / 56.625μsec. (32MHz 時)		実行時間：1392 サイクル / 43.5μsec. (32MHz 時)	

図 2.8　出力アセンブラ

2.5　ポインタ変数の最適化

ポインタ変数の最適化を行うことで演算処理が速くなります。

変更前	変更後

int i;
int *p;
 	
p = array;
for (i = N >> 2; i > 0; i--)
{
　*p++ = 0;
　*p++ = 0;
　*p++ = 0;
　*p++ = 0;
}
for (i = N & 3; i > 0; i--)
{
　*p++ =0;
}
	int i;
	int *p;
	
	p = array;
	for (i = N >> 2; i > 0; i--)
	{
	　*(p+0) = 0;
	　*(p+1) = 0;
	　*(p+2) = 0;
	　*(p+3) = 0;
	}
	for (i = N & 3; i > 0; i--)
	{
	　*p++ =0;
	}

図 2.9　C ソースコード例

備考 array[]はグローバル変数で、N は 10 であるとします。

変更前		変更後	
subw sp, #0x06	2	mov a, !LOWW(_N)	3
mov a, !LOWW(_N)	3	mov b, a	1
mov [sp+0x02], a	2	shr a, 0x02	2
shr a, 0x02	2	mov x, a	1
shrw ax, 8+0x00000	2	clrb a	1
movw hl, ax	3	.BB@LABEL@1_1:	
clrw bc	1	cmp a, x	2
movw ax, #LOWW(_array)	4	bz $.BB@LABEL@1_3	2
movw [sp+0x00], ax	2	.BB@LABEL@1_2:	
.BB@LABEL@1_1:		clrb !LOWW(_array)	3
movw ax, bc	1	clrb !LOWW(_array+0x00001)	3
shlw ax, 0x02	2	clrb !LOWW(_array+0x00002)	3
movw [sp+0x04], ax	2	clrb !LOWW(_array+0x00003)	3
movw ax, bc	2	inc a	1
cmpw ax, hl	1	br $.BB@LABEL@1_1	2
bz $.BB@LABEL@1_3	2	.BB@LABEL@1_3:	
.BB@LABEL@1_2:		mov a, b	1
pop de	1	and a, #0x03	2
push de	1	shrw ax, 8+0x00000	2
mov [de+0x00], #0x00	3	movw bc, ax	1
incw de	1	movw de, #LOWW(_array)	3
movw ax, [sp+0x00]	3	clrw ax	1
mov [de+0x00], #0x00	1	.BB@LABEL@1_4:	
incw de	3	cmpw ax, bc	1
addw ax, #0x0003	2	bz $.BB@LABEL@1_6	2
mov [de+0x00], #0x00	3	.BB@LABEL@1_5:	
movw de, ax	1	mov [de+0x00], #0x00	3
movw ax, [sp+0x00]	3	incw ax	1
mov [de+0x00], #0x00	2	incw de	1
addw ax, #0x0004	3	br $.BB@LABEL@1_4	2
movw [sp+0x00], ax	2	.BB@LABEL@1_6:	
incw bc	1	ret	1
br $.BB@LABEL@1_1	2		
.BB@LABEL@1_3:			
mov a, [sp+0x02]	2		
and a, #0x03	2		
shrw ax, 8+0x00000	2		
movw hl, ax	1		
clrw ax	1		
movw de, ax	1		
.BB@LABEL@1_4:			
movw ax, de	1		
cmpw ax, hl	1		
bz $.BB@LABEL@1_6	2		
.BB@LABEL@1_5:			
movw ax, [sp+0x04]	2		
addw ax, de	1		
movw bc, ax	1		
incw de	4		
mov LOWW(_array)[bc], #0x00	1		
br $.BB@LABEL@1_4	2		
.BB@LABEL@1_6:			
addw sp, #0x06	2		
ret	1		
90 バイト		48 バイト	
実行時間：400 サイクル / 12.5μsec. (32MHz 時)		実行時間：228 サイクル / 7.125μsec. (32MHz 時)	

図 2.10　出力アセンブラ

開発統合環境の最適化レベルによる高速化

開発統合環境の最適化レベルによって、実行速度の高速化を行う事が可能です。

2.5.1　統合開発環境 e² studio での設定

① 開発統合環境 e² studio のプロジェクト・エクスプローラーからプロジェクトを選び右クリックでメニューを表示し、[プロパティ]をクリックしてください。

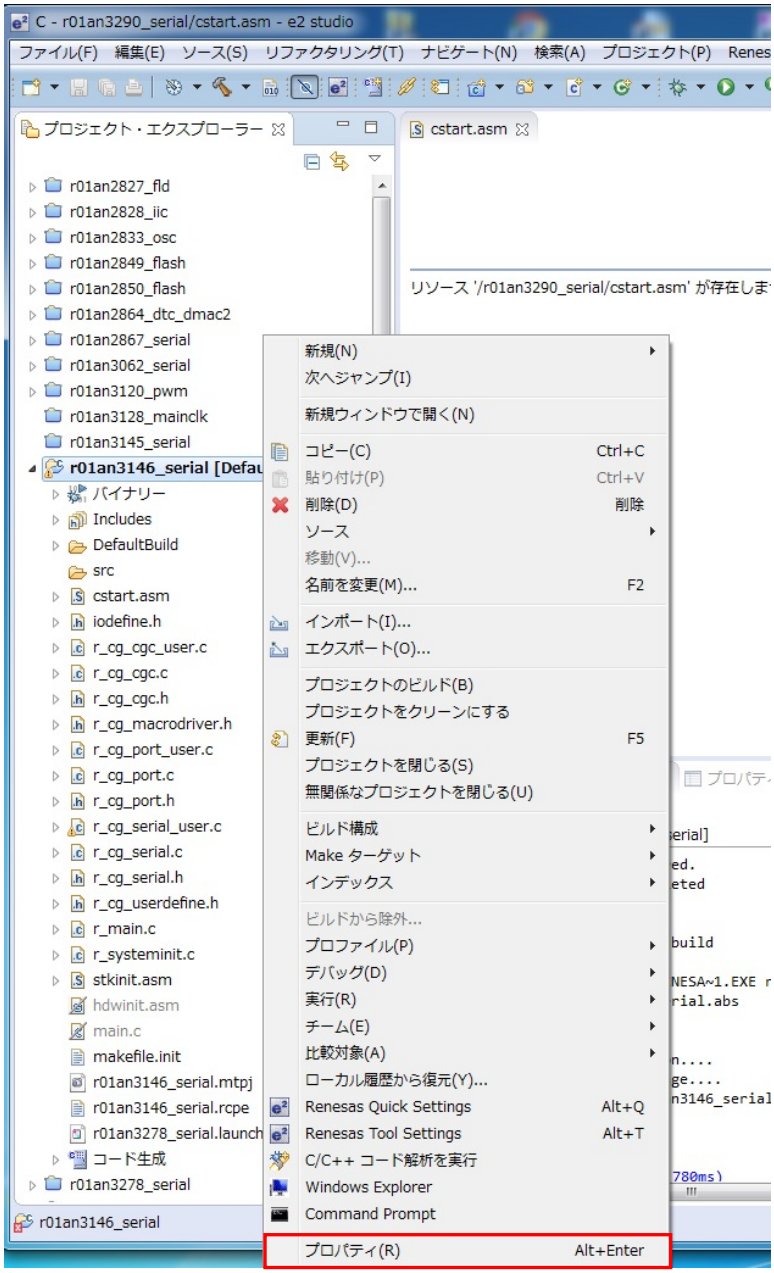

図 2.11　プロジェクト・エクスプローラー

② [C/C++ ビルド]の設定をクリックして設定画面を開き、[Compiler] → [最適化]で、最適化レベルの項目を、[実行速度優先]に変更してください。

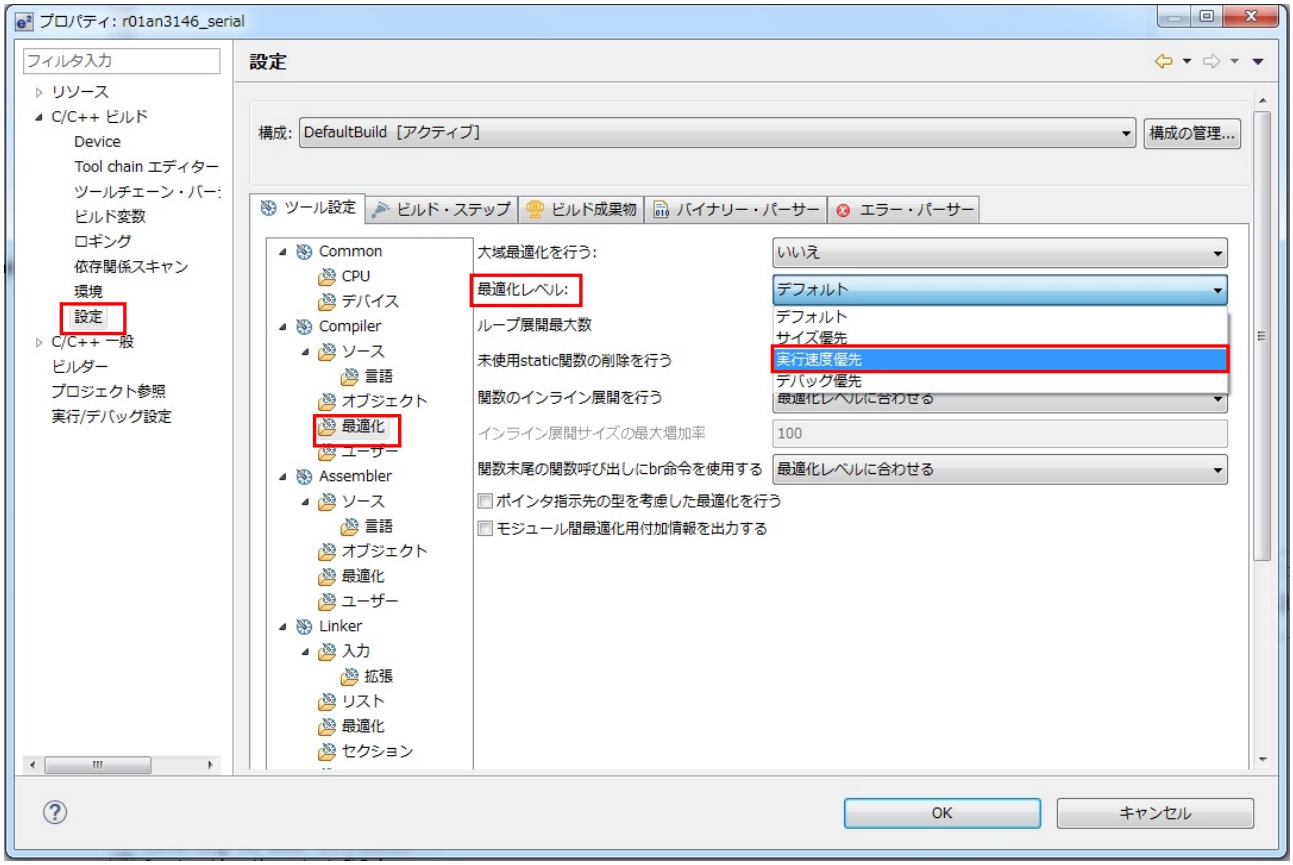

図 2.12　プロパティ画面

③ 実行例

ここでは、実際に同じソースコード(図 2.13　C ソースコード例)の開発統合環境 e² studio の最適化レベルを変更してビルドを行った場合のそれぞれの出力アセンブラを参考例として記載します。

使用したソースコード

```
int i;
int Height;
int Width;
int *p;
int s;

p = &array[0][0];
s = Height * Width;
for (i = 0; i < s; i++)
{
      *p++ = 0;
}
```

図 2.13　C ソースコード例

備考 array[][]はグローバル変数

サイズ優先		実行速度優先		デバッグ優先	
movw de, #0xf900	3	push hl	1	subw sp, #6	2
clrw ax	1	movw ax, #0xf900	3	mov [sp], #0	3
cmpw ax, #50	3	movw [sp], ax	2	mov [sp+2], #10	3
bz $0x220 <main+16>	2	movw bc, #25	4	mov [sp+3], #5	3
mov [de], #0	3	pop de	1	mov [sp+1], #0	3
incw ax	1	push de	1	movw ax, #0xf900	3
incw de	1	mov [de], #0	3	movw [sp+4], ax	2
br $0x214 <main+4>	2	incw de	1	mov a, [sp+3]	2
ret	1	mov [de], #0	3	mov x, a	2
28.156		movw ax, [sp]	2	mov a, [sp+2]	2
		addw ax, #2	3	mulu x	1
		movw [sp], ax	2	mov a, x	2
		movw ax, bc	1	mov [sp+1], a	2
		addw ax, #0xffff	3	mov [sp], #0	3
		movw bc, ax	1	br $0x26e <main+49>	2
		bnz $0x219 <main+9>	2	movw ax, [sp+4]	2
		pop hl	1	movw de, ax	1
		ret	1	mov [de], #0	3
				movw ax, [sp+4]	2
				incw ax	1
				movw [sp+4], ax	2
				mov a, [sp]	2
				inc a	1
				mov [sp], a	2
				mov a, [sp+1]	2
				shrw ax, 8	2
				movw bc, ax	1
				mov a, [sp]	2
				shrw ax, 8	2
				cmpw ax, bc	1
				bc $0x25e <main+33>	2
				addw sp, #6	2
				ret	1
17 バイト		35 バイト		66 バイト	
実行時間：1802 サイクル / 56.3125µsec. (32MHz 時)		実行時間：1694 サイクル / 52.9375µsec. (32MHz 時)		実行時間：2942 サイクル / 91.9375µsec. (32MHz 時)	

図 2.14　出力アセンブラ

2.5.2　開発統合環境 CS+での設定

① 開発統合環境 CS+のプロジェクト・ツリーから CC-RL(ビルド・ツール)を選び右クリックでメニューを表示し、[プロパティ]をクリックしてください。

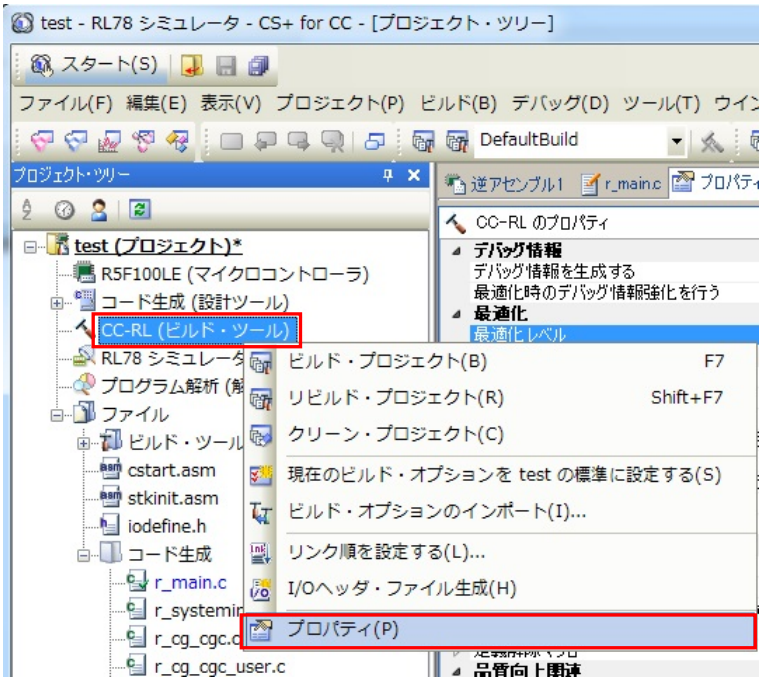

図 2.15　プロジェクト・ツリー

② CC-RL のプロパティから[コンパイル・オプション]タブを選択し、最適化の項目の最適レベルを[実行速度優先(-Ospeed)]に変更してください。

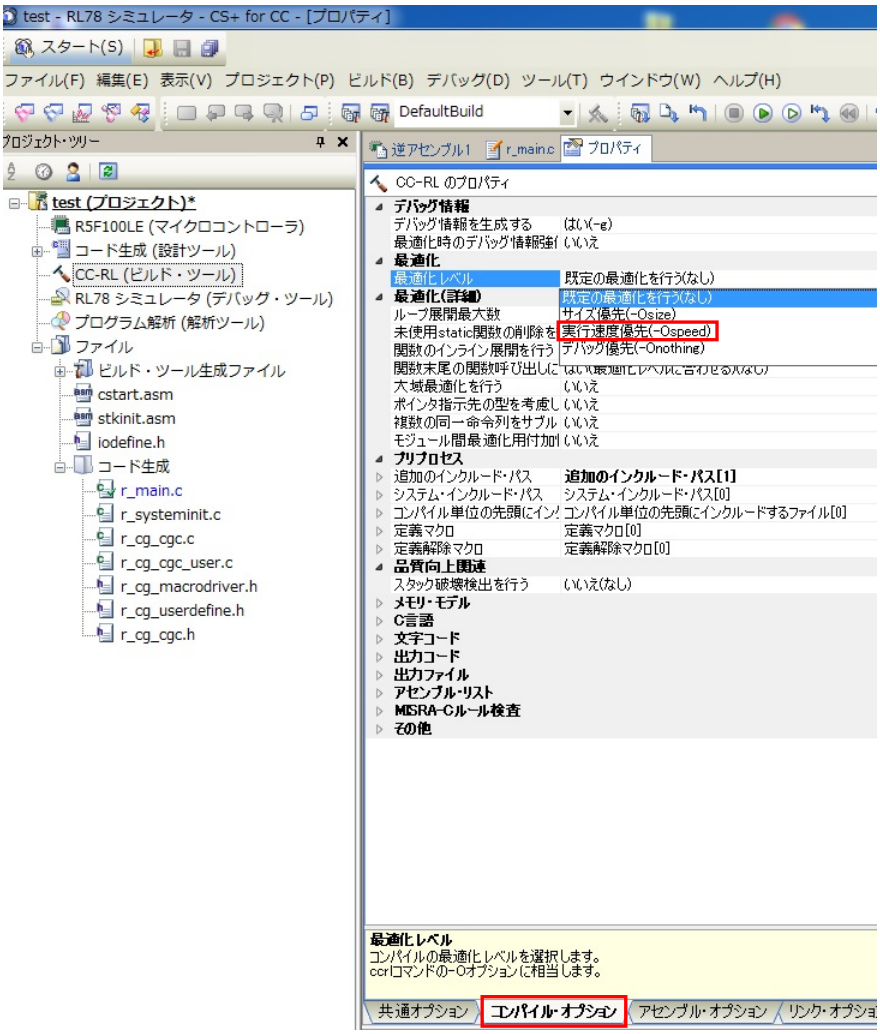

図 2.16　コンパイル・オプション

③ ②の設定で最適化レベルを[既定の最適化を行う(なし)]以外にする事で、個別にファイルの最適化レベルを変更する事も可能です。

④ プロジェクト・ツリーから最適化レベルを変更したいファイルを選び右クリックでメニューを表示し。[プロパティ]をクリックしてください。

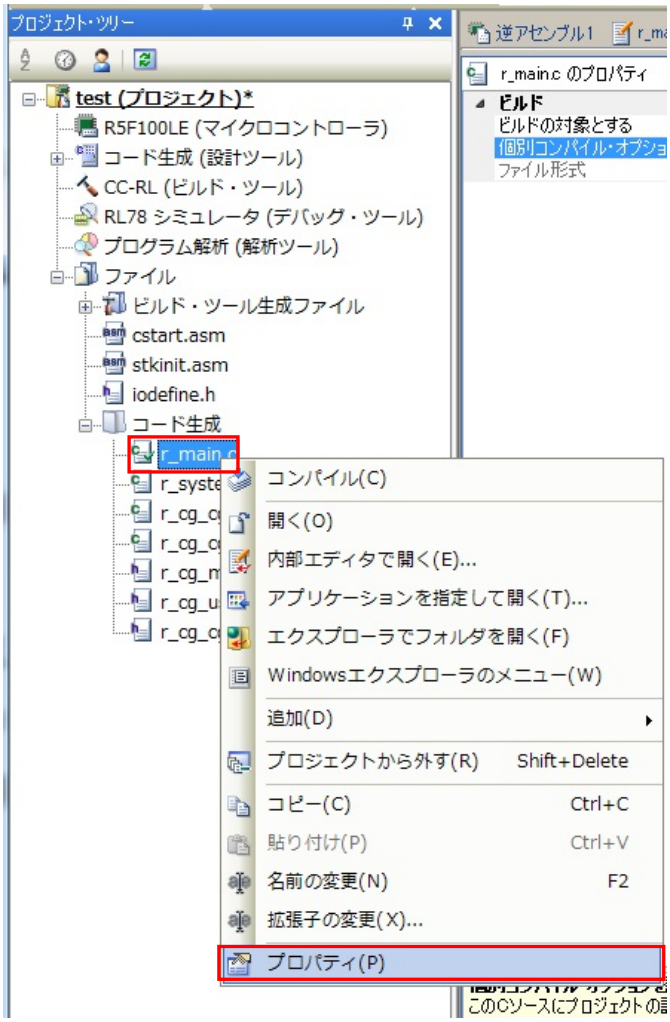

図 2.17　プロジェクト・ツリー

⑤ [ビルド設定]タブ、ビルド項目の[個別コンパイル・オプションを設定する]を[はい]にしてください。

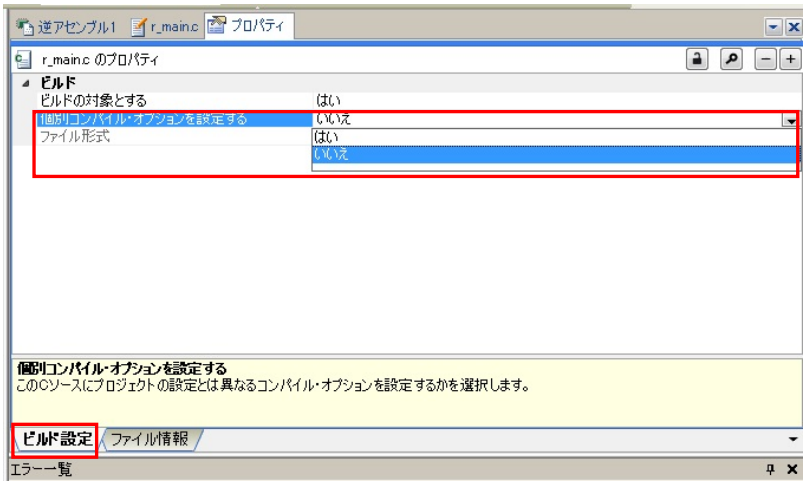

図 2.18　ファイルのプロパティ

⑥　追加された[個別コンパイル・オプション]タブを選び、最適化の項目の最適化レベルを[実行速度優先(-Ospeed)]に変更してください。

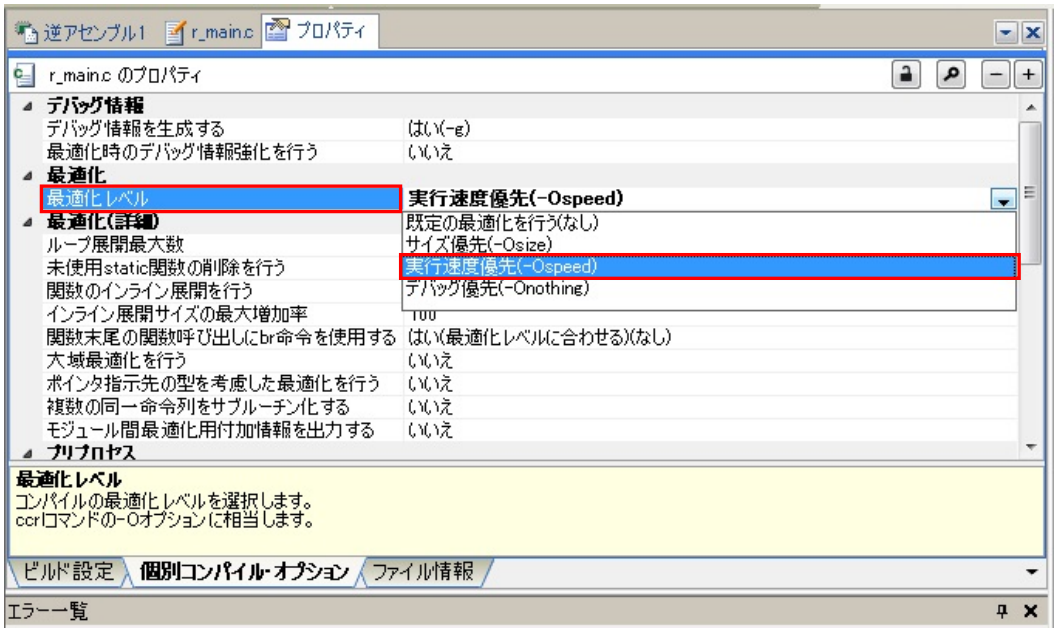

図 2.19　個別コンパイル・オプション

⑦　実行例

ここでは、実際に同じソースコード(図 2.20)の開発統合環境 CS+のコンパイル・オプションを変更してビルドを行った場合のそれぞれの出力アセンブラを参考例として記載します。

使用したソースコード

```
int i;
int Height;
int Width;
int *p;
int s;

p = &array[0][0];
s = Height * Width;
for (i = 0; i < s; i++)
{
      *p++ = 0;
}
```

図 2.20　C ソースコード例

備考 array[][]はグローバル変数

サイズ優先		実行速度優先		デバッグ優先	
movw de, #LOWW(_array)	3	push hl	1	subw sp, #0x06	2
clrw ax	1	movw ax, #LOWW(_array)	3	mov [sp+0x05], #0x05	3
.BB@LABEL@1_1: ; bb16		movw [sp+0x00], ax	2	.BB@LABEL@1_1:	
cmpw ax, #0x0032	3	movw bc, #0x0032	4	mov [sp+0x04], #0x0A	3
bz $.BB@LABEL@1_3	2	.BB@LABEL@1_1:		.BB@LABEL@1_2:	
.BB@LABEL@1_2:		pop de	1	movw ax, #LOWW(_array)	3
mov [de+0x00], #0x00	3	push de	1	movw [sp+0x02], ax	2
incw ax	1	mov [de+0x00], #0x00	3	.BB@LABEL@1_3:	
incw de	1	incw de	1	mov a, [sp+0x04]	2
br $.BB@LABEL@1_1	2	mov [de+0x00], #0x00	3	mov x, a	1
.BB@LABEL@1_3:		movw ax, [sp+0x00]	2	mov a, [sp+0x05]	2
ret	1	addw ax, #0x0002	3	mulu x	1
		movw [sp+0x00], ax	2	mov a, x	1
		movw ax, bc	1	mov [sp+0x01], a	2
		addw ax, #0xFFFF	3	.BB@LABEL@1_4:	
		movw bc, ax	1	mov [sp+0x00], #0x00	3
		bnz $.BB@LABEL@1_1	2	br $.BB@LABEL@1_6	2
		.BB@LABEL@1_2:		.BB@LABEL@1_5:	
		pop hl	1	movw ax, [sp+0x02]	2
		ret	1	movw de, ax	1
				mov [de+0x00], #0x00	3
				movw ax, [sp+0x02]	2
				incw ax	1
				movw [sp+0x02], ax	2
				mov a, [sp+0x00]	2
				inc a	1
				mov [sp+0x00], a	2
				.BB@LABEL@1_6:	
				mov a, [sp+0x01]	2
				shrw ax, 8+0x00000	2
				movw bc, ax	1
				mov a, [sp+0x00]	2
				shrw ax, 8+0x00000	2
				cmpw ax, bc	1
				bc $.BB@LABEL@1_5	2
				.BB@LABEL@1_7:	
				addw sp, #0x06	2
				ret	2
17 バイト		35 バイト		59 バイト	
実行時間：1812 サイクル / 56.625μsec. (32MHz 時)		実行時間：1500 サイクル / 46.875μsec. (32MHz 時)		実行時間：4088 サイクル / 127.75μsec. (32MHz 時)	

図 2.21　出力アセンブラ

3. バグ回避のプログラミングテクニック

3.1 条件式の数値を演算子の左側に記述する

条件式において、図 3.1のように変数を演算子の左側に記述することは推奨出来ません。

```
#define VAL_OK 1

if (ret == VAL_OK)
{
      sub();
}
```

図 3.1　良くない記述例(1)

なぜなら、記述ミスを見逃す可能性があるからです。図 3.2のように等価演算子(==)を代入演算子(=)にしても、コンパイル実行時にコンパイルエラーとならず(ワーニングは出ます)、実行ファイルが出来てしまいます。

```
#define VAL_OK 1

if (ret = VAL_OK)
{
      sub();
}
```

図 3.2　良くない記述例(2)

上記のようなケースを回避する為、条件式の数値を演算子の左側に記述することを推奨します。

```
#define VAL_OK 1

if (VAL_OK == ret)
{
      sub();
}
```

図 3.3　良い記述例

図 3.3のように記述すれば、等価演算子(==)が代入演算子(=)に変わっても、コンパイル実行時にコンパイルエラーとなるためプログラミングミスに気付くことが出来ます。

3.2　マジックナンバー

意味のある定数はマクロとして定義して使用し、マジックナンバー(直値)を使用しないことを推奨します。マクロ化することにより、定数の意味を明確に示すことができます。特に、複数箇所で使用している定数を変更する場合、1 つのマクロを変更するだけでよいため、ミスを未然に防ぐことができます。

変更前	変更後

if (8 == cnt)
{
　cnt++;
}
	#define CNTMAX 8
	if (CNTMAX == cnt)
	{
	　　cnt++:
	}

図 3.4　C ソースコード例

3.3　情報損失の恐れのある演算

型の異なる変数同士の演算は注意が必要です。変数値が変わる（情報喪失する）可能性があります。意図的に異なる型へ代入する場合は、その意図を明示するために型変換を記述してください。

演算では、演算結果がその型で表現できる値の範囲を超えた場合、意図しない値になる可能性があります。演算結果がその型で表現できる値の範囲であることを事前に確認してから演算することを推奨します。または、より大きな値を扱える型に変換してから演算を行ってください。

変更前	変更後

/* 代入の例 */
short s;
long l;
 	
void main(void)
{
　　s = l;
　　s = s + 1;
}
 	
/* 演算の例 */
unsigned int n;
unsigned int m;
 	
n = 0x8000;
m = 0x8000;
 	
if (0xffff < (n + m))
{
　　…
}
	/* 代入の例 */
	short s;
	long l;
	
	void main(void)
	{
	　　s = (short)l;
	　　s = (short)(s + 1);
	}
	
	/* 演算の例 */
	unsigned int n
	unsigned int m;
	
	n = 0x8000;
	m = 0x8000;
	
	if (0xffff < ((long)n + m))
	{
	　　…
	}

図 3.5　C ソースコード例

3.4　const, volatile を取り除く型変換

const や volatile 修飾された領域は、参照しかされない領域であり、最適化をしてはならない領域なので、その領域に対するアクセスに注意しなければなりません。これらの領域を指すポインタ変数に対し、const や volatile を取り除くキャストを行ってしまうと、コンパイラはプログラムの誤った記述に対し、チェックを行えなくなったり、意図しない最適化を行ってしまったりする可能性があります。

変更前	変更後

void sub(char *);
const char *p;
void main(void)
{
　　sub((char*)p);
　　…
}
	void sub(char *);
	const char *p;
	void main(void)
	{
	　　sub(p);
	　　…
	}

図 3.6　C ソースコード例

3.5　再帰呼び出しの禁止

関数は直接的か間接的かにかかわらず、その関数自身を呼び出してはいけません。（再帰呼び出しの禁止）

再帰呼び出しは実行時の利用スタックサイズが予測できないためスタックオーバーフローを引き起こす恐れがあります。

```
unsigned int calc(unsigned int n)
{
    if (1 >= n)
    {
        return (1);
    }
    else
    {
        return (n * calc(n-1));
    }
}
```

図 3.7　良くない記述例

3.6　アクセス範囲や関連データの局所化

① 同一ファイル内の複数の関数からアクセスされる変数は、static 変数宣言してください。

グローバル関数の数が少ないほど、プログラム全体を理解する際の可読性が向上します。グローバル関数を必要以上に増やさないように、static 指定子を付けてください。

変更前	変更後

int n;
void func1(void)
{
　…
　n = 0;
　…
}
 	
void func2(void)
{
　if (0 == n)
　{
　　n++;
　}
　…
}
	static int n;
	void func1(void)
	{
	　…
	　n = 0;
	　…
	}
	
	void func2(void)
	{
	　if (0 == n)
	　{
	　　n++;
	　}
	　…
	}

図 3.8　C ソースコード例

備考 n は他のファイルからアクセスされない変数

② 同一ファイル内で定義した関数でしか参照されない関数は、static 関数にしてください。

グローバル関数の数が少ないほど、プログラム全体を理解する際の可読性が向上します。グローバル関数を必要以上に増やさないように、static 指定子を付けてください。

変更前	変更後

void sub(void)
{
　…
　…
}
 	
void main(void)
{
　…
　sub();
　…
}
	static void sub(void)
	{
	　…
	　…
	}
	
	void main(void)
	{
	　…
	　sub();
	　…
	}

図 3.9　C ソースコード例

備考 sub は他のファイルから呼ばれない関数

③ 関連する定数を定義するときは、#define より enum を使用してください。

関連する定数ごとに enum 型で定義しておくと、定義されていない使い方をすると、コンパイラ等でチェックすることができます。
#define で定義されたマクロ名は、マクロ展開され、コンパイラが処理する名前となりませんが、enum 宣言で定義された enum 定数は、コンパイラが処理する名前になります。コンパイラが処理する名前は、デバッグ時に参照できデバッグが容易になります。

変更前	変更後

#define JANUALY 0
#define FEBRUALY 1
#define SUNDAY 0
#define MONDAY 1
int month;
int day;
 	
…
if (JANUALY == month)
{
　…
　if (MONDAY == day)
　{
　　…
　　…
　}
}
 	
if (SUNDAY == month)　←エラーにならない
{
　…
　…
}
	typedef enum
	{
	　JANUALY, FEBRUALY, …
	} month;
	typedef enum
	{
	　SUNDAY, MONDAY, …
	} day;
	…
	if (JANUALY == month)
	{
	　…
	　if (MONDAY == day)
	　{
	　　…
	　　…
	　}
	}
	
	if (SUNDAY == month)　←エラーになる
	{
	　…
	　…
	}

図 3.10　C ソースコード例

3.7　分岐条件の例外処理

① if-else if 文は最後に else 節を置いてください。特に、else の条件が通常発生しない場合は、else 節に例外処理もしくはプロジェクトで予め規定したコメントを入れてください。

if-else 文に else 節がないと、else 節を書き忘れているのか、else 節が発生しないのがわかりません。else 条件が発生しないことが予め分かっている場合でも、else 節を書くことによって想定外の条件が発生した場合のプログラム動作を予測することができます。

変更前	変更後

if (0 == var)
{
　…
}
else if (0 < var)
{
　…
}
	if (0 == var)
	{
	　…
	}
	else if (0 < var)
	{
	　…
	}
	else
	{
	　/* 例外処理の記述 もしくは コメント*/
	}

図 3.11　C ソースコード例

② switch 文は、最後に default 節を置いてください。特に、default 条件が通常発生しない場合は、default 節に例外処理もしくはプロジェクトで予め規定したコメントを入れてください。

switch 文に default 節がないと、default 節を書き忘れているのか、default 節が発生しないのかがわかりません。default 条件が発生しないことが予め分かっている場合でも、default 節を書くことによって想定外の条件が発生した場合のプログラム動作を予測することができます。

変更前	変更後

switch (var)
{
　case 0:
　　…
　　break;
 	
　case 1:
　　…
　　break;
}
	switch (var)
	{
	　case 0:
	　　…
	　　break;
	
	　case 1:
	　　…
	　　break;
	
	　default:
	　　/* 例外処理の記述 もしくは コメント */
	　　break;
	}

図 3.12　C ソースコード例

③ ループカウンタの比較に等価演算子(==)、不等価演算子(!=)は使用しないでください。

ループカウンタの変化量が 1 ではない場合に無限ループになる可能性があります。

変更前	変更後

void main(void)
{
　int i = 0;
　for (i = 0; i != 11; i += 2)
　{
　　…
　　…
　}
}
	void main(void)
	{
	　int i = 0;
	　for (i = 0; i < 11; i += 2)
	　{
	　　…
	　　…
	　}
	}

図 3.13　C ソースコード例

3.8　特殊な記述への配慮

① 意図的に何もしない文の記述する場合は、コメントまたは空マクロ等を利用して意図を明示してください。

変更前	変更後

for (;;)
{
}
 	
i = CNT;
while (0 < (--i));
	/* コメント使用例 */
	for (;;)
	{
	　　/* 割り込み待ち時間 */
	}
	
	/* 空マクロ使用例 */
	#define NO_OPERATION
	i = CNT;
	while (0 < (--i))
	{
	　　NO_OPERATION;
	}

図 3.14　C ソースコード例

② 無限ループの書き方を規定してください。

無限ループの書き方を規定し、書き方に統一してください。

例）
- 無限ループを for (;;)で統一する。
- 無限ループを while (1)で統一する。
- 無限ループを do ～ while (1)で統一する。
- マクロ化した無限ループを使用する。

同一プロジェクト内に、異なる書き方の無限ループが混在すると保守性が悪くなる恐れがあります。

3.9　使用しない記述の削除

① 使用しない関数、変数、引数、typedef、ラベル、マクロなどは定義しないでください。

使用しない関数（変数／引数／ラベルなど）の定義は、記述ミスであるか判別が困難なため保守性を損ないます。

変更前	変更後

void main(int n)
{
　/* main 関数内で n 未使用 */
　…
}
	void main(void)
	{
	　…
	}

図 3.15　C ソースコード例

② コードをコメントアウトすることを避けてください。

無効なコードを残すことは、コードの可読性を損なうため、極力避けてください。
ただし、デバッグ等でコードの無効化が必要な場合は、コメントアウトではなく、予め規定したルール（#if 0 で囲む等）に従って記述してください。

変更前	変更後
… // i++ …	… #if 0　/* デバッグのため一時無効 */ 　i++ #endif …

図 3.16　C ソースコード例

4. サンプルコード

サンプルコードは、ルネサス エレクトロニクスホームページから入手してください。

5. 参考ドキュメント

RL78 ファミリ ユーザーズマニュアル ソフトウェア編（R01US0015J）

RL78 コンパイラ CC-RL ユーザーズマニュアル（R20UT3123J）

RL78 ファミリ用 C コンパイラ CC-RL コーディングテクニック（R02UT3569J）

（最新版をルネサス エレクトロニクスホームページから入手してください。）

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	RL78 ファミリ用 C コンパイラ CC-RL プログラミングテクニック

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2016.09.20	—	初版発行
1.10	2017.04.10	14 ～ 28	実行時間を修正

製品ご使用上の注意事項

ここでは、マイコン製品全体に適用する「使用上の注意事項」について説明します。個別の使用上の注意事項については、本ドキュメントおよびテクニカルアップデートを参照してください。

1. 未使用端子の処理
   【注意】未使用端子は、本文の「未使用端子の処理」に従って処理してください。
   CMOS 製品の入力端子のインピーダンスは、一般に、ハイインピーダンスとなっています。未使用端子を開放状態で動作させると、誘導現象により、LSI 周辺のノイズが印加され、LSI 内部で貫通電流が流れたり、入力信号と認識されて誤動作を起こす恐れがあります。未使用端子は、本文「未使用端子の処理」で説明する指示に従い処理してください。
2. 電源投入時の処置
   【注意】電源投入時は，製品の状態は不定です。
   電源投入時には、LSI の内部回路の状態は不確定であり、レジスタの設定や各端子の状態は不定です。
   外部リセット端子でリセットする製品の場合、電源投入からリセットが有効になるまでの期間、端子の状態は保証できません。
   同様に、内蔵パワーオンリセット機能を使用してリセットする製品の場合、電源投入からリセットのかかる一定電圧に達するまでの期間、端子の状態は保証できません。
3. リザーブアドレス（予約領域）のアクセス禁止
   【注意】リザーブアドレス（予約領域）のアクセスを禁止します。
   アドレス領域には、将来の機能拡張用に割り付けられているリザーブアドレス（予約領域）があります。これらのアドレスをアクセスしたときの動作については、保証できませんので、アクセスしないようにしてください。
4. クロックについて
   【注意】リセット時は、クロックが安定した後、リセットを解除してください。
   プログラム実行中のクロック切り替え時は、切り替え先クロックが安定した後に切り替えてください。
   リセット時、外部発振子（または外部発振回路）を用いたクロックで動作を開始するシステムでは、クロックが十分安定した後、リセットを解除してください。また、プログラムの途中で外部発振子（または外部発振回路）を用いたクロックに切り替える場合は、切り替え先のクロックが十分安定してから切り替えてください。
5. 製品間の相違について
   【注意】型名の異なる製品に変更する場合は、製品型名ごとにシステム評価試験を実施してください。
   同じグループのマイコンでも型名が違うと、内部 ROM、レイアウトパターンの相違などにより、電気的特性の範囲で、特性値、動作マージン、ノイズ耐量、ノイズ輻射量などが異なる場合があります。型名が違う製品に変更する場合は、個々の製品ごとにシステム評価試験を実施してください。

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、その他の不適切に使用しないでください。かかる改造、改変、複製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、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、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できません。たとえ、意図し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。また、当社製品および技術を、(1)核兵器、化学兵器、生物兵器等の大量破壊兵器およびこれらを運搬することができるミサイル（無人航空機を含みます。）の開発、設計、製造、使用もしくは貯蔵等の目的、(2)通常兵器の開発、設計、製造または使用の目的、または(3)その他の国際的な平和および安全の維持の妨げとなる目的で、自ら使用せず、かつ、第三者に使用、販売、譲渡、輸出、賃貸もしくは使用許諾しないでください。
   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. お客様の転売、貸与等により、本書（本ご注意書きを含みます。）記載の諸条件に抵触して当社製品が使用され、その使用から損害が生じた場合、当社は一切その責任を負わず、お客様にかかる使用に基づく当社への請求につき当社を免責いただきます。
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. 本資料に記載された情報または当社製品に関し、ご不明点がある場合には、当社営業にお問い合わせください。

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。

(Rev.3.0-1　2016.11)

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/