

RX Family C/C++ Compiler Package (CC-RX)

R01AN4195EJ0100

Rev.1.0

Programming Techniques

May. 17, 2018

Introduction

This application note describes methods of programming for efficiency in terms of code size, speed of execution, and ROM data size.

Compiler Revision for which Correct Operation has been Confirmed

CC-RX V2.08.00 for the RX family

Contents

Introduction.....	1
1. Overview	4
2. Options.....	5
2.1 Compiler Options	5
2.1.1 -instalign4/-instalign8	8
2.1.2 -nouse_div_inst.....	10
2.1.3 -stack_protector/-stack_protector_all	11
2.1.4 -avoid_cross_boundary_prefetch	13
2.1.5 -optimize	14
2.1.6 -speed/-size.....	15
2.1.7 -loop	16
2.1.8 -inline.....	17
2.1.9 -case	19
2.1.10 -volatile.....	20
2.1.11 -const_copy	21
2.1.12 -const_div/-noconst_div	22
2.1.13 -library	23
2.1.14 -scope/-noscope.....	24
2.1.15 -schedule/-noschedule	26
2.1.16 -map/-smap	27
2.1.17 -approxdiv	28
2.1.18 -simple_float_conv	29
2.1.19 -nofpu	30
2.1.20 -alias	31
2.1.21 -ip_optimize	32
2.1.22 -merge_files	33
2.1.23 -whole_program	34

2.1.24	-dbl_size	35
2.1.25	-int_to_short	36
2.1.26	-auto_enum	37
2.1.27	-pack	38
2.1.28	-fint_register	40
2.1.29	-branch	43
2.1.30	-base	44
2.1.31	-nouse_pid_register	45
2.1.32	-save_acc	46
2.1.33	-control_flow_integrity	47
2.2	Assembler Option	48
2.3	Linkage Options	49
2.3.1	-optimize=symbol_delete	50
2.3.2	-optimize=same_code	51
2.3.3	-optimize=short_format	52
2.3.4	-optimize=branch	53
3.	Language Extensions	54
3.1	#pragma Directives	54
3.1.1	#pragma interrupt	55
4.	Coding Techniques	56
4.1	Using Structures	57
4.2	Variables and the const Qualifier	58
4.3	Local Variables and Global Variables	59
4.4	Offsets for Structure Members	60
4.5	Allocating Bit Fields	62
4.6	Loop Control Variable	63
4.7	Function Interfaces	64
4.8	Reducing the Number of Loops	65
4.9	Using Tables	66
4.10	Branches	67
4.11	Inline Expansion	68
4.12	Moving Identical Expressions in More than One Conditional Branch Destination before the Conditional Branch	70
4.13	Replacing a Sequence of Complicated if Statement with a Simple Statement Having the Same Logical Meaning	72
4.14	Converting short- or char-Type Variables into the int Type	73
4.15	Unifying Common case Processing in switch Statements	74
4.16	Replacing for Loops with do-while Loops	76
4.17	Replacing Division by Powers of Two with Shift Operations	77
4.18	Changing Bit Fields with Two or More Bits to the char Type	78

4.19 Assigning Small Absolute Values when Referring to Constants.....	79
Website and Support.....	80

1. Overview

The methods of programming which lead to efficiency in terms of code size, speed of execution, and ROM data size are classified under the following three headings.

- Options
- Language extensions
- Coding techniques

The results of measurement and assembly code given in this application note were obtained by using V2.08 of the CC-RX compiler. The value assumed for the **-isa/-cpu** option was **-isa=rxv2**. The default values for optimization options are as follows.

Emphasize efficiency in the generation of code	-size
Optimization level	-optimize=2
Unrolling loops	-loop=2
Inline expansion	-noinline
Converting division by constants into multiplication	-noconst_div
Scheduling instructions	-schedule
Propagation of const-qualified variables as constants	-const_copy
Division of optimizing ranges	-scope
Optimization of access to external variables	-nomap
Optimization in consideration of the types of data indicated by pointers	-alias=noansi

Note that the degrees of the effects depend on the details of the source code and may also change due to upgrading of the CC-RX compiler.

2. Options

This chapter describes the effects on code size, ROM data size, and speed of execution when options for CC-RX are specified.

The degrees of the effects depend on the details of the source code.

2.1 Compiler Options

√: Improved, x: Worsened, Δ: Depends on the situation, —: No effect, (): Default

Option	Code Size	ROM Size	Required Number of Cycles	Remarks
-instalign4	x	—	√	Instructions at branch destinations are aligned with 4-byte boundaries for CPUs with 32-bit unit instruction queues (mainly intended for RX200-series MCUs).
-instalign8	x	—	√	Instructions at branch destinations are aligned with 8-byte boundaries for CPUs with 64-bit unit instruction queues (mainly intended for RX600-series MCUs).
-nouse_div_inst	x	x	x	Generation of division operations is suppressed to shorten response times in the execution of interrupt functions. Specifying this option may lower efficiency in terms of code size and speed of execution.
-stack_protector -stack_protector_all	x	—	x	This option generates code to detect stack smashing at the entry and exit points of functions. The code for the detection of stack smashing may lower efficiency in terms of code size and speed of execution.
-avoid_cross_boundary_prefetch	x	—	x	This option is used to prevent the reading of data across 4-byte boundaries in prefetching for string manipulation instructions. Specifying this option may lower efficiency in terms of code size and speed of execution.
-optimize (specified with -size)	√	Δ	Δ	When -size is specified, optimization will emphasize code size. However, specifying this option may lower efficiency in terms of the speed of execution.
-optimize (specified with -speed)	Δ	Δ	√	When -speed is specified, optimization will emphasize execution performance. However, specifying this option may lower efficiency in terms of code size.

-goptimize	√	√	√	At the time of linkage, inter-module optimization is applied to files compiled with this option specified. For optimization at the time of linkage, refer to section 2.3, Linkage Options.
-speed	x	—	√	
-size	(√)	—	(x)	
-loop	x	—	√	The effect of specifying this option depends on its parameter.
-inline	x	—	√	The effect of specifying this option depends on its parameter.
-case=ifthen	x	√	Δ	The more case labels the statements have, the larger the code size will become.
-case=table	√	x	Δ	The more case labels the statements have, the larger the ROM data size will become. However, the speed of execution remains the same regardless of the number of labels.
-volatile	x	—	x	
-const_copy	(√)	—	(√)	Propagation as constants proceeds even for const -qualified external variables. Specifying this option will improve efficiency in terms of code size and speed of execution.
-noconst_copy	x	—	x	Specifying this option disables -const_copy .
-const_div	x	—	√	
-noconst_div	(√)	—	(√)	
-library=intrinsic	√	—	√	
-noscope	√	—	√	Specifying this option may lower the performance of object code for large functions.
-schedule	—	—	(√)	Instructions are scheduled to facilitate pipeline processing. Specifying this option will improve the speed of execution.
-noschedule	—	—	x	Specifying this option disables -schedule .
-map	√	—	√	
-smap	√	—	√	
-approxdiv	—	—	√	The precision and order of operations may be changed.
-simple_float_conv	√	—	√	Specifying this option will improve the performance when -isa=rxv1 .
-nofpu	x	—	x	
-alias=ansi	√	—	√	
-ip_optimize	√	Δ	√	
-merge_files	√	Δ	√	
-whole_program	√	Δ	√	

-dbl_size=8	x	x	x	Specifying this option leads to the double type being handled as 8 bytes by the compiler.
-int_to_short	Δ	√	Δ	The int type is replaced with the short type before compilation.
-auto_enum	Δ	√	Δ	
-pack	x	√	x	
-fint_register	Δ	—	Δ	This option specifies general registers as being only for use in fast interrupt functions. Specifying this option will improve the performance of such interrupt functions but may lower the performance of normal functions.
-branch=16	√	—	—	
-branch=32	x	—	x	
-base	Δ	—	Δ	Specifying this option is effective when the entire program includes frequent access to external variables.
-nouse_pid_register	x	—	x	Since the generated code does not use the PID register, specifying this option may lower efficiency in terms of code size and speed of execution.
-save_acc	Δ	—	Δ	This option generates code for the saving and restoring of the accumulators in the case of interrupt functions. Although specifying this option may lower the performance of interrupt functions, it allows the generation of instructions that use the accumulator.
-control_flow_integrity	x	x	x	This option leads to the checking of indirect function calls. The code for checking may lower efficiency in terms of code size, speed of execution, and ROM data size.

2.1.1 -instalign4/-instalign8

These options are used to align instructions at branch destinations with 4- or 8-byte boundaries. While these options facilitate efficient use of the CPU’s instruction queues and accelerate program execution, they may also increase the code size.

Specifying **-instalign4** aligns the addresses of instructions to suit the specifications of CPUs with 4-byte unit instruction queues. This option is mainly intended for RX200-series MCUs.

Specifying **-instalign8** aligns the addresses of instructions to suit the specifications of CPUs with 8-byte unit instruction queues. This option is mainly intended for RX600-series MCUs.

The specifications of instruction queues are covered in the user’s manuals (hardware manuals) for the individual MCUs.

C source code

```

long a;

int func1(int num)
{
    return (num + 1);
}

void func2(void)
{
    a += 1;
    a += a;
}

void main(void)
{
    unsigned int i;
    for (i = 0; i < 10; ++i) {
        if (func1(i) < 10) {
            func2();
        }
    }
    a += 1;
}

```

-cpu=rx200 (CPUs with 32-bit instruction queues)

	With -instalign4	Without -instalign4
Code size (bytes)	57	55
Number of cycles (cycles)	278	297

-cpu=rx600 (CPUs with 64-bit instruction queues)

	With -instalign8	Without -instalign8
Code size (bytes)	61	55
Number of cycles (cycles)	259	278

The same effects can also be obtained by **#pragma** directives. When **-instalign4** or **-instalign8** and a **#pragma** directive are specified at the same time, the **#pragma** directive will take priority.

Example: With **-instalign4**

```
void func1(void)          /* Aligned with a 4-byte boundary (by -instalign4) */
{
}

#pragma instalign8 func2
void func2(void)          /* Aligned with an 8-byte boundary */
{
}

#pragma noinstalign func3
void func3(void)          /* Not aligned */
{
}
```

Example: With **-instalign8**

```
void func1(void)          /* Aligned with an 8-byte boundary (by -instalign8) */
{
}

#pragma instalign4 func2
void func2(void)          /* Aligned with a 4-byte boundary */
{
}

#pragma noinstalign func3
void func3(void)          /* Not aligned */
{
}
```

2.1.2 -nouse_div_inst

This option leads to the generation of code in which DIV, DIVU, or FDIV instructions are never used for division and modular division operations in the program.

This option calls the equivalent runtime functions instead of DIV, DIVU, or FDIV instructions. This may shorten response times in the execution of interrupt functions by 1 to 20 cycles but lower efficiency in terms of code size and speed of execution.

C source code

```
long a, b;
unsigned long c, d;
float e, f;
const float cf= 11.0;

void main(void)
{
  a = a / b;
  c = c / d;

  e = e / f;
  f = e / cf;
}
```

	With -nouse_div_inst	Without -nouse_div_inst
Code size (bytes)	85	69
Number of cycles (cycles)	701	59

Note: Using this option may also increase the required ROM size.

2.1.3 -stack_protector/-stack_protector_all

This option generates code to detect stack smashing at the entry and exit points of functions.

The code to detect stack smashing may lower efficiency in terms of code size and speed of execution.

C source code

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

void func(void)
{
    volatile char str[10];
    int i;
    for (i = 0; i <= 9; i++) {
        str[i] = i;
    }
}

void __stack_chk_fail(void)
{
    /* stack is broken! */
    __brk();
}

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

	With -stack_protector	With -stack_protector_all	Without -stack_protector/ -stack_protector_all
Code size (bytes)	44	65	24
Number of cycles (cycles)	75	82	68

The same effects can also be obtained by **#pragma** directives. When **-stack_protector** or **-stack_protector_all** and a **#pragma** directive are specified at the same time, the **#pragma** directive will take priority.

Example: With `-stack_protector/-stack_protector_all`

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

struct DATA func1(void)    /* Generates code to detect stack smashing */
{
    struct DATA data = {0, 1, 2, 3};
    return data;
}

#pragma no_stack_protector (func2)
struct DATA func2(void)    /* Prevents the generation of code to detect
stack smashing */
{
    struct DATA data = {0, 1, 2, 3};
    return data;
}
```

Example: Without `-stack_protector/-stack_protector_all`

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

struct DATA func1(void)    /* Prevents the generation of code to detect
stack smashing */
{
    struct DATA data = {0, 1, 2, 3};
    return data;
}

#pragma stack_protector (func2)
struct DATA func2(void)    /* Generates code to detect stack smashing */
{
    struct DATA data = {0, 1, 2, 3};
    return data;
}
```

2.1.4 -avoid_cross_boundary_prefetch

This option is used to prevent the reading of data across 4-byte boundaries in prefetching for string manipulation instructions.

Specifying this option may lower efficiency in terms of code size and speed of execution when source code includes calls of library functions for string handling, i.e. memchr(), strlen(), strcpy(), strncpy(), strcmp(), strncmp(), strcat(), or strncat(), and **-library=intrinsic** has been specified for compilation.

C source code

```
#include <string.h>

unsigned long len;

void main(void)
{
    char str[] = "abcdefghijklmnopqrstuvwxyz";
    len = strlen(str);
}
```

	With -avoid_cross_boundary_prefetch	Without -avoid_cross_boundary_prefetch
Code size (bytes)	50	43
Number of cycles (cycles)	79	73

2.1.5 -optimize

This option specifies the optimization level. The default is **-optimize=2**.

The default values of the options listed below vary with the combination of whether the size or speed option has been selected, the optimization level, and whether the input source code is C or C++.

Item	Option
Unrolling loops	-loop
Inline expansion	-inline / -noinline
Converting division by constants into multiplication	-const_div / -noconst_div
Scheduling instructions	-schedule / -noschedule
Propagation of const-qualified variables as constants	-const_copy / -noconst_copy
Division of optimizing ranges	-scope / -noscope
Optimization of access to external variables	-map / -smap / -nomap
Optimization in consideration of the types of data indicated by pointers	-alias=ansi / -alias=noansi

C source code

```
int i = 0;
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 main(void)
{
    sub(x, y, i);
}
```

	-optimize=0	-optimize=1	-optimize=2	-optimize=max
Code size (bytes)	63	37	35	24
Number of cycles (cycles)	36	20	14	10

Note: Using this option may also increase the required ROM size.

2.1.6 -speed/-size

This option is used to select whether speed or size should be emphasized in optimization.

When **-speed** is specified, emphasis in optimization will be on execution performance.

When **-size** is specified, emphasis in optimization will be on code size (default).

C source code

```
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);
        }
    }
}
```

	-speed	-size
Code size (bytes)	70	45
Number of cycles (cycles)	226	310

2.1.7 -loop

This option specifies whether to optimize speed by unrolling loops.

Unrolling loop statements accelerates execution while increasing the code size.

C source code

```

long val;

void main(void)
{
    unsigned long 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);
    }
}

```

	-loop=1	-loop=2	-loop=8
Code size (bytes)	154	204	307
Number of cycles (cycles)	4391	3424	3069

Note: **-loop=8** is the default value when **-optimize=max/-speed**.

2.1.8 -inline

This option specifies whether functions are to be automatically inline expanded.

When **-inline** is specified, the compiler automatically performs inline expansion.

When **-noinline** is specified, the compiler does not automatically perform inline expansion.

You can also use a parameter with the **-inline** option to specify the allowed increase in the function's size due to the use of inline expansion. For example, when **inline=100**, functions will be inline-expanded if their size is increased by up to 100% (size is doubled). When **inline=0**, functions will only be inline-expanded if the size remains the same or decreases.

C source code

```
long val;
long x[1000];

static void func1(void)
{
    ++val;
}
void func2(int a)
{
    if (a) {
        x[a] = 0;
    }
}

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

	-inline=200	-inline=0	-noinline
Code size (bytes)	93	58	70
Number of cycles (cycles)	150	364	465

You can also use **#pragma** directives to enable or disable inline expansion for particular functions. When **-inline** or **-noinline** and a **#pragma** directive are specified at the same time, the **#pragma** directive will take priority.

Example: With **-inline**

```
void func1(void)          /* Inline expansion is enabled (by -inline). */
{
}

#pragma noinline(func2)
void func2(void)         /* Inline expansion is disabled. */
{
}
```

Example: With **-noinline**

```
void func1(void)          /* Inline expansion is disabled (by -noinline).
*/
{
}

#pragma inline(func2)
void func2(void)         /* Inline expansion is enabled. */
{
}
```

2.1.9 -case

This option specifies how switch statements will be expanded.

When **-case=ifthen** is specified, switch statements will be expanded by using the if_then method. The more case labels the statements have, the larger the code size will become. The speed of execution will also depend on the number of case labels.

When **-case=table** is specified, switch statements will be expanded by using the table method. The more case labels the statements have, the larger the ROM data size will become. However, the speed of execution remains the same regardless of the number of labels.

When **-case=auto** (default) is specified, the compiler automatically selects between the if_then and table methods.

C source code

```
long val = 10;

void main(void)
{
    switch (val) {
        case 1:
            val += 10;
            break;
        case 2:
            val *= 10;
            break;
        case 3:
            val /= 10;
            break;
        default:
            val -= 10;
            break;
    }
}
```

	-case=ifthen	-case=table
Code size (bytes)	31	39
ROM size (bytes)	4	7
Number of cycles (cycles)	15	13

Note: For the above example of C source code, the compiler selects if_then when **-case=auto**.

2.1.10 -volatile

This option is used to select whether all external variables should be handled as if they were volatile-qualified.

When **-volatile** is specified, all external variables are handled as if they were volatile-qualified. Accordingly, the number of times and order of access to external variables are exactly the same as is written in the C/C++ source file. However, this prevents the optimization of external variables and thus may lower efficiency in terms of code size and speed of execution.

C source code

```
long val = 0;

void main(void)
{
    val += 1;
    val -= 2;
    val *= 3;
    val /= 4;
}
```

	With -volatile	Without -volatile
Code size (bytes)	33	19
Number of cycles (cycles)	22	12

2.1.11 -const_copy

This option is used to enable or disable propagation by the compiler of const-qualified external variables as constants. Enabling constant propagation accelerates program execution.

When **-const_copy** (default) is specified, the compiler propagates const-qualified external variables as constants.

When **-noconst_copy** is specified, const-qualified external variables are not propagated as constants.

C source code

```

const long val = 0;
long result;

void main(void)
{
    result = val + 10;
}
    
```

	-const_copy	-noconst_copy
Code size (bytes)	10	19
Number of cycles (cycles)	5	8

2.1.12 -const_div/-noconst_div

These options are used to enable or disable converting calculations for division and modulo operations (obtaining the remainders of division) of integer constants into sequences of multiplication and bitwise operation (shift or bitwise AND operation) instructions. Enabling this conversion accelerates the speed of execution, while increasing the code size.

When **-const_div** is specified, calculations for division and modulo operations of integer constants in the source file are converted into sequences of multiplication and bitwise operation (shift or bitwise AND operation) instructions. Using this option in conjunction with the **-size** option increases the speed of execution compared to cases where **-noconst_div** is specified.

When **-noconst_div** is specified, the corresponding division and modulo instructions are used for calculating the results of division and modulo operations of integer constants in the source file (except in the case of unsigned integers that are powers of two). Using this option in conjunction with the **-speed** option reduces the code size compared to cases where **-const_div** is specified.

C source code

```
long a = 0x7FFFFFFF;

void main(void)
{
    a = a / 1000;
}
```

	-const_div (-size)	-const_div (-speed)	-noconst_div (-size/-speed)
Code size (bytes)	26	27	16
Number of cycles (cycles)	13	12	23

2.1.13 -library

This option is used to specify the extent to which library functions will be expanded.

When **-library=function** is specified, all library functions will be called. This may lower efficiency in terms of code size and speed of execution.

When **-library=intrinsic** (default) is specified, only `abs()`, `fabsf()`, and library functions which can use string manipulation instructions will be expanded. When **-library=intrinsic** and **-isa=rxv2** are selected at the same time, calls of the `sqrtf()` function or of the `sqrt()` function when **-dbl_size=4** are expanded as FSQRT instructions. Note, however, that no value is set for **errno** in such cases.

C source code

```
#include <stdlib.h>
int a;

void main(void)
{
    a = abs(a);
}
```

	-library=function	-library=intrinsic
Code size (bytes)	19	13
Number of cycles (cycles)	16	8

2.1.14 -scope/-noscope

This option is used to select whether to divide the target range for optimization before compilation.

When **-noscope** is specified, the target range for optimization is not divided before compilation. A larger target range generally improves the performance of the object code, although compilation will take longer.

In the compilation of large functions, the handling of numerous general-purpose registers in the object code may lower its performance. Using **-scope** in such cases allows large functions to be treated as multiple separate ranges in optimization. This will improve object performance compared to cases where **-noscope** is specified.

C source code

```
long array[40];
long val = 10;
void main(void)
{
    int i;
    for (i = 0; i < 40; ++i) {
        array[i] = val * i;
    }
    for (i = 0; i < 40; ++i) {
        if (array[i] > i) {
            array[i] += val + i;
        }
        else if (array[i] > (i * 2)) {
            array[i] += val + (i * 2);
        }
        else if (array[i] > (i * 3)) {
            array[i] += val + (i * 3);
        }
        else if (array[i] > (i * 4)) {
            array[i] += val + (i * 4);
        }
        else if (array[i] > (i * 5)) {
            array[i] += val + (i * 5);
        }
        else if (array[i] > (i * 6)) {
            array[i] += val + (i * 6);
        }
        else if (array[i] > (i * 7)) {
            array[i] += val + (i * 7);
        }
        else if (array[i] > (i * 8)) {
            array[i] += val + (i * 8);
        }
        else if (array[i] > (i * 9)) {
            array[i] += val + (i * 9);
        }
        else {
            array[i] += val + (i * 10);
        }
    }
}
```


	-scope	-noscope
Code size (bytes)	318	312
Number of cycles (cycles)	1089	1046

Note: **-loop=2** is assumed for the above results.

2.1.15 -schedule/-noschedule

This option is used to select whether to schedule instructions to facilitate pipeline processing. Scheduling instructions improves the speed of execution.

When **-schedule** is specified, instructions are scheduled to facilitate pipeline processing. **-schedule** is assumed when **-optimize=2** or **-optimize=max** is specified.

When **-noschedule** is specified, instructions are not scheduled so they are handled in the order in which they are written in the C/C++ source file. **-noschedule** is assumed when **-optimize=1** or **-optimize=0** is specified.

C source code

```
long a, b;
unsigned long c, d;
float e, f;

void main(void)
{
    a = a + b;
    c = c + d;
    e = e + f;
}
```

	-schedule	-noschedule
Code size (bytes)	58	58
Number of cycles (cycles)	24	25

2.1.16 -map/-smap

This option is used to select whether to optimize access to external variables. Optimizing access to external variables will improve efficiency in terms of code size and speed of execution.

When **-map** is specified, CC-RX optimizes access to external variables, generating code that uses addresses relative to a base address (selected according to the external-symbol allocation information file created by the optimizing linkage editor) for access to external or static variables. How the **-map** option is used differs with the specification of the **-output** option.

(1) When **-output=obj** or **-output=src**

Compile the source file without specifying **-map**. Then specify **-map=<filename>** at the time of linkage to create an external-symbol allocation information file (.bls).

```
ccrx -isa=rxv2 -output=obj tp1.c tp2.c
rlink tp1.obj tp2.obj -map=out.bls
```

Specify **-map=<name of the external-symbol allocation information file>** in **ccrx** and compile and link the source files again. CC-RX generates code in which access to external variables has been optimized.

```
ccrx -isa=rxv2 -output=obj -map=out.bls tp1.c tp2.c
rlink tp1.obj tp2.obj -output=out.abs
```

(2) When **-output=abs**, **-output=hex**, or **-output=sty**

Specify **-map** without any parameters. CC-RX automatically proceeds with compilation and linkage twice and generates code in which access to external variables has been optimized.

```
ccrx -isa=rxv2 -output=abs -map tp1.c tp2.c
```

When **-smap** is specified, CC-RX sets a base address for external or static variables defined in the file to be compiled and generates code that uses addresses relative to that base address for access to those variables.

C source code [tp1.c]

```
long a, b, c;
extern long d, e, f;
void main(void)
{
    a = d;
    b = e;
    c = f;
}
```

C source code [tp2.c]

```
long d = 10;
long e = 10;
long f = 10;
```

	-map	-smap	Without -map/-smap
Code size (bytes)	21	33	43
Number of cycles (cycles)	13	14	16

2.1.17 -approxdiv

This option is used to convert the division of floating-point constants into the multiplication of the reciprocals of the constants. Specifying this option improves performance in the division of floating-point constants. It may, however, change the precision and order of operations, so take care on these points.

C source code

```
float a;  
  
void main(void)  
{  
    a /= 1.1;  
}
```

	With -approxdiv	Without -approxdiv
Code size (bytes)	18	18
Number of cycles (cycles)	9	23

2.1.18 -simple_float_conv

This option omits part of the type-conversion processing for the floating-point types. This option can be expected to be effective when the instruction set architecture of the code to be generated is RXv1.

This option changes the code generated to handle type conversion of floating-point numbers in the following cases.

- (a) Type conversion from 32-bit floating-point type to unsigned integer type
- (b) Type conversion from unsigned integer type to 32-bit floating-point type
- (c) Type conversion from integer type to 64-bit floating-point type via 32-bit floating-point type (except when **-optimize=0**)

Specifying this option will improve efficiency in terms of code size and speed of execution. However, the results of conversion may differ from those for conversion in accord with the C/C++ language specifications, so take care on this point.

C source code

```

unsigned long isrc = 2;
float fsrc = 2.0;
unsigned long idst;
float fdst;

void main(void)
{
    idst = (unsigned long)fsrc;
    fdst = (float)isrc;
}

```

	With -simple_float_conv	Without -simple_float_conv
Code size (bytes)	36	73
Number of cycles (cycles)	17	23

Note: RXv1 (**-isa=rxv1**) is assumed in this case.

2.1.19 -nofpu

This option is used to select whether to generate code in which FPU instructions are used.

When **-nofpu** is specified, the generated code does not use FPU instructions.

When **-fpu** is specified, the generated code uses FPU instructions.

The default for this option is **-fpu** except when **-cpu=rx200** is specified, in which it is **-nofpu**. It is not possible to specify **-cpu=rx200** and **-fpu** at the same time.

C source code

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

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

	-fpu	-nofpu
Code size (bytes)	31	45
Number of cycles (cycles)	40	97

2.1.20 -alias

This option selects whether to perform optimization in consideration of the types of data indicated by pointers. Specifying this option improves code efficiency in terms of code size and speed of execution. However, the results of conversion may differ from the expected values if the C source code does not comply with the ISO/IEC 9899 standard.

When **-alias=ansi** is specified, optimization in consideration of the types of data indicated by pointers proceeds in accord with ISO/IEC 9899. The performance of object code is generally better when **-alias=ansi** is specified than when **-alias=noansi** is specified, but the results of execution may differ according to whether **-alias=ansi** or **-alias=noansi** is specified.

When **-alias=noansi** is specified, ISO/IEC 9899-based optimization in consideration of the types of data indicated by pointers is not performed.

C source code

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

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

	-alias=ansi	-alias=noansi
Code size (bytes)	30	36
Number of cycles (cycles)	10	16

2.1.21 -ip_optimize

This option is used to select whether to apply global optimization such as optimization in which inter-procedural alias analysis and the propagation of constant parameters and return values are utilized.

C source code

```
long result;  
  
static long func(long x, long y, long z)  
{  
    return (x - y + z);  
}  
void main(void)  
{  
    result = func(3, 4, 5);  
}
```

	With -ip_optimize	Without -ip_optimize
Code size (bytes)	21	23
Number of cycles (cycles)	17	18

2.1.22 -merge_files

This option allows the compiler to compile multiple C source files and output the results to a single object file.

Specifying both **-merge_files** and **-ip_optimize** can obtain a synergistic effect.

C source code [tp1.c]

```

long result;

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

C source code [tp2.c]

```

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

	With -merge_files	Without -merge_files
Code size (bytes)	122	131
Number of cycles (cycles)	5	19

- Notes:
1. In some cases, ROM size may also be improved.
 2. The code size here also includes the size of the startup routine.

2.1.23 -whole_program

This option is used to apply global optimization by merging all source files to be compiled on the assumption that the entire program is to be compiled.

When this option is specified, compilation proceeds on the assumption that the conditions listed below are satisfied. Correct operation is not guaranteed otherwise.

Condition 1: Files outside the scope of compilation at this time will neither modify nor refer to the values and addresses of **extern** variables defined in the target source files.

Condition 2: Files outside the scope of compilation at this time will not call functions within the target source files, although calls of functions in files outside the scope of compilation by target source files are allowed.

C source code [tp1.c]

```
extern const int c;
int result;

int func(void);

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

C source code [tp2.c]

```
const int c = 1;
int x = 10;
int *p;

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

	With -whole_program	Without -whole_program
Code size (bytes)	170	171
Number of cycles (cycles)	93	163

Notes: 1. In some cases, ROM size may also be improved.
2. The code size here also includes the size of the startup routine.

2.1.24 -dbl_size

This option specifies whether or not to change variables of the **double** and **long double** types to the **float** type.

When **-dbl_size=4** is specified, this option changes the given types to the **float** type (default).

When **-dbl_size=8** is specified, this option does not change the types.

C source code

```
double a, b;  
const double c = 11.0;  
  
void main(void)  
{  
    a = a / b;  
    b = b / c;  
}
```

	-dbl_size=4	-dbl_size=8
Code size (bytes)	31	63
ROM size (bytes)	4	8
Number of cycles (cycles)	40	107

2.1.25 -int_to_short

Before compilation, variables in the source file are changed to the **short** type if written as the **int** type and to the **unsigned short** type if written as the **unsigned int** type.

In the program with no portability and written on the assumption that the sizes of **int** and **unsigned int** are 32 bits, this may change the results of execution.

C source code

```

int x;
long y;
const int imm = 1;

void main(void)
{
    x += imm;
    y += x;
}
    
```

	With -int_to_short	Without -int_to_short
Code size (bytes)	26	24
ROM size (bytes)	2	4
Number of cycles (cycles)	13	12

2.1.26 -auto_enum

This option selects the processing of enumerated values, i.e. lists qualified by **enum**, as the minimum set of required values, i.e. only those which are actually used in the code.

Although this reduces the ROM data size, expanding the values in **enum** to the **long** type, etc., may also affect the code size and speed of execution.

C source code

```
enum number {
    one = 1,
    two = 2,
    three = 3
};

int x;
enum number num;
const enum number DATA = one;

void main(void)
{
    num += num;
    x += num;
}
```

	With -auto_enum	Without -auto_enum
Code size (bytes)	26	24
ROM size (bytes)	1	4
Number of cycles (cycles)	13	12

2.1.27 -pack

This option specifies the unit for the boundary alignment of structure and class members.

When **-pack** is specified, the unit of boundary alignment for structure and class members is 1, which reduces the ROM data size. However, in cases where alignment is required, it will also lead to deterioration in terms of the code size and speed of execution.

When **-unpack** (default) is specified, the boundary alignment value for structure and class members equals the maximum boundary alignment value for the members.

C source code

```

struct DATA
{
    char c;
    long l;
};

struct DATA data = {1, 2};
long result;

long func(void)
{
    return (data.l);
}

void main(void)
{
    result = func();
}

```

	-pack	-unpack
Code size (bytes)	23	21
ROM size (bytes)	5	8
Number of cycles (cycles)	17	16

The same effects can also be obtained by **#pragma** directives. When **-pack** and a **#pragma** directive are specified at the same time, the **#pragma** directive will take priority.

Example: With **-pack**

```
struct DATA1 /* The -pack option applies. */
{
    char a; /* Byte offset = 0 */
    long b; /* Byte offset = 1 */
    short c; /* Byte offset = 5 */
} data1; /* The total size is 7 bytes. */

#pragma unpack
struct DATA2 /* The -pack option is not applicable from here. */
{
    char a; /* Byte offset = 0 */
    long b; /* Byte offset = 4 */
    short c; /* Byte offset = 8 */
} data2; /* The total size is 12 bytes. */

#pragma packoption
struct DATA3 /* The -pack option applies. */
{
    char a; /* Byte offset = 0 */
    long b; /* Byte offset = 1 */
    short c; /* Byte offset = 5 */
} data3; /* The total size is 7 bytes. */
```

Example: With **-unpack**

```
struct DATA1 /* The -pack option applies. */
{
    char a; /* Byte offset = 0 */
    long b; /* Byte offset = 4 */
    short c; /* Byte offset = 8 */
} data1; /* The total size is 12 bytes. */

#pragma pack
struct DATA2 /* Operation from here is as if the -pack option were
specified. */
{
    char a; /* Byte offset = 0 */
    long b; /* Byte offset = 1 */
    short c; /* Byte offset = 5 */
} data2; /* The total size is 7 bytes. */

#pragma packoption
struct DATA3 /* The -pack option applies. */
{
    char a; /* Byte offset = 0 */
    long b; /* Byte offset = 4 */
    short c; /* Byte offset = 8 */
} data3; /* The total size is 12 bytes. */
```

2.1.28 -fint_register

This option specifies general registers as being only for use in fast interrupt functions (functions that have the fast interrupt setting (**fint**) in their interrupt specification as stated with **#pragma interrupt**). For details on the extended language specifications of **#pragma interrupt**, refer to section 3.1.1. The specified registers cannot be used in functions other than fast interrupt functions. The registers specified by this option can thus be used in fast interrupt (**fint**) functions without having to be saved and restored. Since this reduces the number of registers available to other functions, the performance of interrupt functions is improved but the performance of the program as a whole is in general lowered, although this depends on the number of registers specified.

Option	Registers for Use Only with Fast Interrupts
fint_register=0 (default)	None
fint_register=1	R13
fint_register=2	R12, R13
fint_register=3	R11, R12, R13
fint_register=4	R10, R11, R12, R13

C source code [normal function]

```

long val[40];
long tmp = 10;

void main(void)
{
    int i;
    for (i = 0; i < 40; ++i) {
        if (tmp > i) {
            val[i] = tmp + i;
        }
        else if (tmp > (i * 2)) {
            val[i] = tmp + (i * 2);
        }
        else if (tmp > (i * 3)) {
            val[i] = tmp + (i * 3);
        }
        else if (tmp > (i * 4)) {
            val[i] = tmp + (i * 4);
        }
        else if (tmp > (i * 5)) {
            val[i] = tmp + (i * 5);
        }
        else if (tmp > (i * 6)) {
            val[i] = tmp + (i * 6);
        }
        else if (tmp > (i * 7)) {
            val[i] = tmp + (i * 7);
        }
        else if (tmp > (i * 8)) {
            val[i] = tmp + (i * 8);
        }
        else if (tmp > (i * 9)) {
            val[i] = tmp + (i * 9);
        }
        else {
            val[i] = tmp + (i * 10);
        }
    }
}

```

	fint_register =0	fint_register =1	fint_register =2	fint_register =3	fint_register =4
Code size (bytes)	153	153	153	161	170
Number of cycles (cycles)	1654	1654	1654	1734	1873

C source code [interrupt function]

```
volatile int count;

#pragma interrupt int_func(vect=10, fint)
void int_func(void)
{
    count++;
}
```

	fint_register =0	fint_register =1	fint_register =2	fint_register =3	fint_register =4
Code size (bytes)	18	18	14	14	14
Number of cycles (cycles)	12	10	8	8	8

2.1.29 -branch

This option specifies the width of addresses for branches to functions defined in other sections or files.

When **-branch=16** is specified, the program is compiled with branch widths within 16 bits.

When **-branch=24** is specified, the program is compiled with branch widths within 24 bits (default).

When **-branch=32** is specified, no branch width is specified.

C source code

```
void sub(void);

void main(void)
{
    sub();
}

#pragma section ResetPRG
void sub(void)
{
}
```

	-branch=16	-branch=24	-branch=32
Code size (bytes)	3	4	8
Number of cycles (cycles)	6	6	7

2.1.30 -base

This option specifies a fixed general register for use with base addresses throughout the program.

In the case of an entire program that includes frequent access to external variables, specifying this option will improve efficiency in terms of code size and speed of execution.

C source code

```

long val[40];
long tmp = 10;

void main(void)
{
    int i;
    for (i = 0; i < 40; ++i) {
        if (tmp > i) {
            val[i] = tmp + i;
        }
        else if (tmp > (i * 2)) {
            val[i] = tmp + (i * 2);
        }
        else if (tmp > (i * 3)) {
            val[i] = tmp + (i * 3);
        }
        else if (tmp > (i * 4)) {
            val[i] = tmp + (i * 4);
        }
        else if (tmp > (i * 5)) {
            val[i] = tmp + (i * 5);
        }
        else if (tmp > (i * 6)) {
            val[i] = tmp + (i * 6);
        }
        else if (tmp > (i * 7)) {
            val[i] = tmp + (i * 7);
        }
        else if (tmp > (i * 8)) {
            val[i] = tmp + (i * 8);
        }
        else if (tmp > (i * 9)) {
            val[i] = tmp + (i * 9);
        }
        else {
            val[i] = tmp + (i * 10);
        }
    }
}

```

	With -base (-base=ram=R8)	Without -base
Code size (bytes)	165	166
ROM size (bytes)	4	4
Number of cycles (cycles)	1533	1540

2.1.31 -nouse_pid_register

When this option is specified, the generated code does not use the PID register.

Specifying this option may lower efficiency in terms of code size and speed of execution.

C source code

```

long val[40];
long tmp = 10;

void main(void)
{
    int i;
    for (i = 0; i < 40; ++i) {
        if (tmp > i) {
            val[i] = tmp + i;
        }
        else if (tmp > (i * 2)) {
            val[i] = tmp + (i * 2);
        }
        else if (tmp > (i * 3)) {
            val[i] = tmp + (i * 3);
        }
        else if (tmp > (i * 4)) {
            val[i] = tmp + (i * 4);
        }
        else if (tmp > (i * 5)) {
            val[i] = tmp + (i * 5);
        }
        else if (tmp > (i * 6)) {
            val[i] = tmp + (i * 6);
        }
        else if (tmp > (i * 7)) {
            val[i] = tmp + (i * 7);
        }
        else if (tmp > (i * 8)) {
            val[i] = tmp + (i * 8);
        }
        else if (tmp > (i * 9)) {
            val[i] = tmp + (i * 9);
        }
        else {
            val[i] = tmp + (i * 10);
        }
    }
}

```

	With -nouse_pid_register	Without -nouse_pid_register
Code size (bytes)	167	158
Number of cycles (cycles)	1823	1674

Note: Measurement was with **-fint_register=3**.

2.1.32 -save_acc

This option generates code for the saving and restoring of the accumulators (ACC, ACC0, or ACC1) in the case of interrupt functions.

When this option is specified, the values of the accumulators are retained even if interrupts occur. This permits the generation of instructions that use an accumulator, such as MACLO, from C/C++ statements.

C source code [normal function]

```
short src1[3] = {10, 11, 12};
short src2[3] = {20, 21, 22};
int result;

int func(const short* src1, const short* src2)
{
    return (src1[0] * src2[0]) + (src1[1] * src2[1]) + (src1[2] * src2[2]);
}

void main(void)
{
    result = func(src1, src2);
}
```

	With -save_acc	Without -save_acc
Code size (bytes)	49	49
Number of cycles (cycles)	25	27

Note: The **-speed** option is specified for the above results.

C source code [interrupt function]

```
#include <machine.h>

long src1 = 10;
long src2 = 20;
long result;

#pragma interrupt func(vect=10)
void func(void)
{
    result = src1 * src2;
}

void main(void)
{
    int_exception(10);
    nop();
}
```

	With -save_acc	Without -save_acc
Code size (bytes)	53	31
Number of cycles (cycles)	32	18

Note: The code size and number of cycles are for the interrupt function alone.

2.1.33 -control_flow_integrity

This option is used to check the calling functions in the case of indirect function calls.

Since this involves the addition of code and data for use in checking, specifying this option may lower efficiency in terms of code size, ROM data size, and speed of execution.

C source code

```

void __control_flow_chk_fail(void) {}

void func1(char a) {}
void func2(long b) {}
void func3(void){}
void (*p1)(char a) = func1;
void (*p2)(long b) = func2;

void func4(void)
{
    func3();
}

void main(void)
{
    p1(1);
    p2(1);
    func4();
}

```

	With -control_flow_integrity	Without -control_flow_integrity
Code size (bytes)	192	147
ROM size (bytes)	188	65
Number of cycles (cycles)	81	36

Note: The code size and ROM size include the size of the startup routine.

2.2 Assembler Option

√: Improved, x: Worsened, Δ: Depends on the situation, —: No effect

Option	Code Size	ROM Size	Required Number of Cycles	Remarks
-goptimize	√	√	√	At the time of linkage, inter-module optimization is applied to files for which this option was specified. For optimization at the time of linkage, refer to section 2.3, Linkage Options.

2.3 Linkage Options

This section describes the effects on code size, ROM data size, and speed of execution when optimizing linkage options are specified. Optimization is applied to files for which **-goptimize** was specified at the time of compilation or assembly.

Optimization is not applied to sections for which **-section_forbid** is specified.

Optimization is also not applied to ranges from the address plus the size for which **-absolute_forbid** is specified.

√: Improved, x: Worsened, Δ: Depends on the situation, —: No effect

Option	Code Size	ROM Size	Required Number of Cycles	Remarks
-optimize=symbol_delete	√	√	—	
-optimize=same_code	√	—	x	
-optimize=short_format	√	—	—	
-optimize=branch	√	—	—	

Note: When the linkage editor is started from the command line, all optimization options apply by default.

2.3.1 -optimize=symbol_delete

Variables or functions to which nothing refers are deleted. Be sure to specify **#pragma entry** at the time of compilation or the **entry** symbol with **-entry** in the linkage editor.

With this option specified, the deletion of variables and functions with **-symbol_forbid** specified is not allowed.

C source code

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

void func1(void)
{
    value1++;
}

void func2(void)
{
    value2++;
}

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

	With -optimize=symbol_delete	Without -optimize=symbol_delete
Code size (bytes)	59	135
ROM size (bytes)	56	64

Note: The code size and ROM size include the size of the startup routine.

2.3.2 -optimize=same_code

Subroutines are created from identical sequences of instructions.

Specifying this option may improve code size but lower the speed of execution.

-samesize specifies the minimum code size to which this form of optimization is to be applied (the default is **-samesize=1E**).

C source code

```
volatile int value = 0;

int v1;
int v2;
int v3;
int v4;
int v5;

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

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

    sub();
}
```

	With -optimize=same_code	Without -optimize=same_code
Code size (bytes)	116	253
Number of cycles (cycles)	91	74

Note: The code size includes the size of the startup routine.

2.3.3 -optimize=short_format

Instructions having a displacement or immediate value are replaced with smaller instructions, reducing the code size by the amounts taken up by displacements and immediate values.

C source code

```
volatile int value = 0;

int v1;
int v2;
int v3;
int v4;
int v5;

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

	With -optimize=short_format	Without -optimize=short_format
Code size (bytes)	157	179

Note: The code size includes the size of the startup routine.

2.3.4 -optimize=branch

The sizes of branch instructions are optimized with the use of program allocation information.

C source code

```
extern void sub(void);

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

	With -optimize=branch	Without -optimize=branch
Code size (bytes)	116	135

Note: The code size includes the size of the startup routine.

3. Language Extensions

This chapter describes the effects on code size, ROM data size, and speed of execution of **#pragma** directives among the language extensions.

3.1 #pragma Directives

√: Improved, x: Worsened, Δ: Depends on the situation, —: No effect

Directive	Code Size	ROM Size	Required Number of Cycles	Remarks
#pragma interrupt	Δ	—	Δ	Changing interrupt specifications used with this directive (fint , etc.) can improve the performance of interrupt functions.

3.1.1 #pragma interrupt

This directive is used to declare that a function is an interrupt function. Changing interrupt specifications can improve the performance in terms of the speed of execution and code size of interrupt functions.

Interrupt Specifications	Format	Specifications
Fast interrupt	fint	Specifies the function as being for use as the handler for a fast interrupt.
Limitation on registers in interrupt function	save	Limits the number of registers used in the interrupt function to R1 to R5, R14, and R15. The code for saving and restoring the values of these registers is not generated, so they cannot be used in interrupt functions with this parameter specified. It may thus lower code efficiency in terms of speed.
Nested interrupt enable	enable	Sets the I flag in the PSW to 1 at the beginning of the interrupt function to enable the nesting of interrupts.
Accumulator saving	acc	Generates instructions for saving and restoring the accumulators.

C source code

```

long count;
long l1, l2;

#pragma interrupt int_func
void int_func(void)
{
    count = l1 * l2;
}

```

The following table shows the result of comparison when no particular interrupt specifications are made and when **fint**, **save**, **enable**, or **acc** is specified for the C source code above.

	None	fint	save	enable	acc
Code size (bytes)	31	27	31	33	53
Number of cycles (cycles)	18	11	18	19	32

Specifying the **-fint_register** compiler option for fast interrupts (**fint**) will increase the speed of functions for fast interrupts. For **-fint_register**, refer to section 2.1.28.

4. Coding Techniques

This chapter describes the effects on code size, ROM data size, and speed of execution through particular methods for the coding of user programs.

√: Improved, x: Worsened, Δ: Depends on the situation, —: No effect

Item	Code Size	ROM Size	Required Number of Cycles	Remarks
Using Structures	√	—	√	
Variables and the const Qualifier	—	√	—	
Local Variables and Global Variables	√	√	√	
Allocating Bit Fields	√	√	√	
Loop Control Variable	x	—	√	
Function Interfaces	√	—	√	
Reducing the Number of Loops	x	—	√	
Using Tables	√	x	√	
Branches	—	—	√	
Inline Expansion	Δ	—	√	
Using if-else Statements Instead of switch Statements	Δ	—	x	
Using Temporary Variables to Reduce Multiple Lines of Code for Access to an External Variable into a Single Line for This Purpose	√	—	√	
Moving Identical Expressions in More than One Conditional Branch Destination before the Conditional Branch	√	—	√	
Replacing a Sequence of Complicated if Statement with a Simple Statement Having the Same Logical Meaning	√	—	√	
Converting short- or char-Type Variables into the int Type	√	—	√	
Unifying Common case Processing in switch Statements	√	—	√	
Replacing for Loops with do-while Loops	√	—	√	
Replacing Division by Powers of Two with Shift Operations	√	—	√	
Changing Bit Fields with Two or More Bits to the char Type	√	x	√	
Assigning Small Absolute Values when Referring to Constants	√	—	—	

4.1 Using Structures

Declaring related data in structures may improve the speed of execution.

In cases of repeated reference to related data in the same function, using a structure makes it easy for the compiler to generate code using relative access and this can be expected to improve efficiency in terms of code size and speed of execution. Passing the data as an argument also improves code efficiency. Since relative access places a limit on the range of access, it is effective when the data which are frequently accessed are placed at the top of the structure.

Declaring data in structures facilitates tuning through the adjustment of data expressions.

Without a Structure	With a Structure
<p><u>C source code</u></p> <pre>int a, b, c; void func(void) { a = 1; b = 2; c = 3; }</pre>	<p><u>C source code</u></p> <pre>struct s { int a; int b; int c; } s1; void func(void) { struct s *p = &s1; p->a = 1; p->b = 2; p->c = 3; }</pre>
<p><u>Assembly-language expanded code</u></p> <pre>_func: .STACK_func=4 MOV.L #_a, R14 MOV.L #00000001H, [R14] MOV.L #_b, R14 MOV.L #00000002H, [R14] MOV.L #_c, R14 MOV.L #00000003H, [R14] RTS</pre>	<p><u>Assembly-language expanded code</u></p> <pre>_func: .STACK_func=4 MOV.L #_s1, R1 MOV.L #00000001H, [R1] MOV.L #00000002H, 04H[R1] MOV.L #00000003H, 08H[R1] RTS</pre>
<p>Code size: 28 bytes Number of cycles: 9</p>	<p>Code size: 15 bytes Number of cycles: 7</p>

4.2 Variables and the const Qualifier

Declare variables for which the values will not change with the const qualifier.

When program code includes the initialization of a global variable with a declaration, the initial value is allocated to ROM and the global variable to RAM. The global variable is initialized when the initial value is transferred from ROM to RAM when the program is started. When a variable with an initial value has been const-qualified, the variable will not be rewritten and the compiler will not reserve RAM for it. This reduces the amount of RAM in use and eliminates the need for the transfer from ROM to RAM.

Not const-Qualified	Const-Qualified
<u>C source code</u> <pre>char a[] = {1, 2, 3, 4, 5};</pre>	<u>C source code</u> <pre>const char a[] = {1, 2, 3, 4, 5};</pre>
ROM size: 5 bytes RAM size: 5 bytes	ROM size: 5 bytes RAM size: 0 bytes

4.3 Local Variables and Global Variables

Declaring variables for local use, such as temporary variables and loop counters, as local variables by declaration within the functions where they are used will improve the speed of execution.

If a variable can be used as a local variable, declare it in that way, rather than as a global variable. Since the value of a global variable may be changed by a call of a function or operations that affect a pointer, optimization will be less efficient if a variable that can be declared as local is declared as global.

Before Using a Local Variable	After Using a Local Variable
<p><u>C source code</u></p> <pre>int tmp; void func(int* a, int* b) { tmp = *a; *a = *b; *b = tmp; }</pre>	<p><u>C source code</u></p> <pre>void func(int* a, int* b) { int tmp; tmp = *a; *a = *b; *b = tmp; }</pre>
<p><u>Assembly-language expanded code</u></p> <pre>_func: .STACK_func=4 MOV.L #_tmp, R14 MOV.L [R1], [R14] MOV.L [R2], [R1] MOV.L [R14], [R2] RTS</pre>	<p><u>Assembly-language expanded code</u></p> <pre>_func: .STACK_func=4 MOV.L [R1], R14 MOV.L [R2], [R1] MOV.L R14, [R2] RTS</pre>
<p>Code size: 12 bytes Number of cycles: 13</p>	<p>Code size: 8 bytes Number of cycles: 8</p>

4.4 Offsets for Structure Members

Declaring members which are more often used higher in the code may improve code size.

Before Improvement	After Improvement
<u>C source code</u> <pre> struct str { long l1[8]; char c1; }; struct str str1; char x; void func(void) { x = str1.c1; } </pre>	<u>C source code</u> <pre> struct str { char c1; long l1[8]; }; struct str str1; char x; void func(void) { x = str1.c1; } </pre>
<u>Assembly-language expanded code</u> <pre> _func: .STACK_func=4 MOV.L #_x, R14 MOV.L #_str1, R15 MOV.B 20H[R15], [R14] RTS </pre>	<u>Assembly-language expanded code</u> <pre> _func: .STACK_func=4 MOV.L #_x, R14 MOV.L #_str1, R15 MOV.B [R15], [R14] RTS </pre>
Code size: 16 bytes Number of cycles: 8	Code size: 15 bytes Number of cycles: 8

When defining a structure, declare the members in consideration of the alignment value. Alignment means that the addresses to which variables are allocated for more efficient access to the variables.

For example, the boundary alignment value for the **long** type is 4 bytes and the **long**-type variable must be allocated to the address which is a multiple of 4. For a structure variable, alignment applies to both its members and to the structure variable itself. The boundary alignment values for members are the same as those for variables of the same type that are not structure members. The boundary alignment value of a structure variable is the highest value among those of its members. If allocating the members of a structure variable without any spaces would violate the required alignments, alignment of the addresses is obtained by including spaces. That is, padding is required. Padding is also required when the size of a structure variable is not a multiple of the highest boundary alignment value. Frequent requirements for padding lower the efficiency of allocation to memory.

Before Improvement	After Improvement
<p><u>C source code</u></p> <pre> /* The boundary alignment value is 4 bytes since the member with the maximum alignment value is of the long type. */ struct str { char c1; /* 1 byte */ /* 3 bytes of padding */ long l1; /* 4 bytes */ char c2; /* 1 byte */ char c3; /* 1 byte */ char c4; /* 1 byte */ /* 1 byte for padding */ } str1; </pre>	<p><u>C source code</u></p> <pre> /* The boundary alignment value is 4 bytes since the member with the maximum alignment value is of the long type. */ struct str { char c1; /* 1 byte */ char c2; /* 1 byte */ char c3; /* 1 byte */ char c4; /* 1 byte */ long l1; /* 4 bytes */ } str1; </pre>
<p><u>Assembly-language expanded code</u></p> <pre> .SECTION B,DATA,ALIGN=4 _str1: .blk1 3 </pre>	<p><u>Assembly-language expanded code</u></p> <pre> .SECTION B,DATA,ALIGN=4 _str1: .blk1 2 </pre>

4.5 Allocating Bit Fields

Allocate bit fields to which values are to be consecutively set to the same structure.

To set members of bit fields in different structures, access to each of the structures is required for access to the members. Such access can be kept down to a single access to the structure itself by collectively allocating related bit fields to the same structure.

The following shows an example in which the size is improved by allocating related bit fields to the same structure.

Before Allocating Bit Fields to the Same Structure	After Allocating Bit Fields to the Same Structure
<p><u>C source code</u></p> <pre> struct str { int flag1:1; } b1, b2, b3; void func(void) { b1.flag1 = 1; b2.flag1 = 1; b3.flag1 = 1; } </pre>	<p><u>C source code</u></p> <pre> struct str { int flag1:1; int flag2:1; int flag3:1; } a1; void func(void) { a1.flag1 = 1; a1.flag2 = 1; a1.flag3 = 1; } </pre>
<p><u>Assembly-language expanded code</u></p> <pre> _func: .STACK_func=4 MOV.L #_b1, R14 BSET #00H, [R14].B MOV.L #_b2, R14 BSET #00H, [R14].B MOV.L #_b3, R14 BSET #00H, [R14].B RTS </pre>	<p><u>Assembly-language expanded code</u></p> <pre> _func: .STACK_func=4 MOV.L #_a1, R14 BSET #00H, [R14].B BSET #01H, [R14].B BSET #02H, [R14].B RTS </pre>
<p>Code size: 25 bytes Number of cycles: 13</p>	<p>Code size: 13 bytes Number of cycles: 13</p>

4.6 Loop Control Variable

Declaring a loop control variable as a signed 4-byte integer type (signed int or signed long) raises the likelihood of optimization in the form of loop unrolling, which reduces the code size and increases the speed of execution.

Declaration without signed long	Declaration with signed long
<u>C source code</u> <pre> unsigned long n = 50; signed short array[100]; void func(void) { signed short i; for (i = 0; i < n; i++) { array[i+5] = 0; } } </pre>	<u>C source code</u> <pre> unsigned long n = 50; signed short array[100]; void func(void) { signed long i; for (i = 0; i < n; i++) { array[i+5] = 0; } } </pre>
<u>Assembly-language expanded code</u> <pre> _func: .STACK_func=4 MOV.L #00000000H, R15 MOV.L #_n, R14 MOV.L [R14], R14 MOV.L #_array, R5 L11: ; bb7 MOV.W R15, R15 CMP R14, R15 BGEU L13 L12: ; bb SHLL #01H, R15, R1 ADD R5, R1 ADD #01H, R15 MOV.W #0000H, 0AH[R1] BRA L11 L13: ; return RTS </pre>	<u>Assembly-language expanded code</u> <pre> _func: .STACK_func=4 MOV.L #00000000H, R5 MOV.L #_n, R15 MOV.L [R15], R15 MOV.L #_array, R14 ADD #0AH, R14 MOV.L #00000000H, R4 L11: ; bb6 CMP R15, R5 BEQ L13 L12: ; bb MOV.W R4, [R14+] ADD #01H, R5 BRA L11 L13: ; return RTS </pre>
Code size: 35 bytes Number of cycles: 511	Code size: 31 bytes Number of cycles: 362

4.7 Function Interfaces

Efficient use of the arguments of functions reduces the amount of RAM required and improves the speed of execution.

The number of arguments should be carefully selected so that all arguments can be allocated to registers (up to four). If there are too many arguments, turn them into a structure and pass the pointer to it. If the structure itself is passed instead of a pointer to the structure, the variables may not be allocated to registers. Allocating arguments to registers simplifies calling and processing at the entry and exit points of functions. This also saves space in the stack area.

The user’s manual for the compiler describes the specifications of function interfaces.

No Arguments in a Structure	Arguments in a Structure
<p><u>C source code</u></p> <pre> void func(char a, char b, char c, char d, char e, char f, char g, char h) {} void call_func(void) { func(1,2,3,4,5,6,7,8); } </pre>	<p><u>C source code</u></p> <pre> struct str { char a; char b; char c; char d; char e; char f; char g; char h; }; void func(struct str* str_arg) {} void call_func(void) { struct str arg = {1,2,3,4,5,6,7,8}; func(&arg); } </pre>
<p><u>Assembly-language expanded code</u></p> <pre> _call_func: .STACK_call_func=8 SUB #04H, R0 MOV.L #00000004H, R4 MOV.B #05H, [R0] MOV.L #00000003H, R3 MOV.L #00000002H, R2 MOV.B #08H, 03H[R0] MOV.L #00000001H, R1 MOV.B #07H, 02H[R0] MOV.B #06H, 01H[R0] BSR _func ADD #04H, R0 RTS </pre>	<p><u>Assembly-language expanded code</u></p> <pre> _call_func: .STACK_call_func=12 SUB #08H, R0 MOV.L R0, R1 MOV.L #04030201H, [R0] MOV.L #08070605H, 04H[R0] BSR _func RTSD #08H </pre>
<p>Code size: 28 bytes Number of cycles: 21</p>	<p>Code size: 22 bytes Number of cycles: 15</p>

4.8 Reducing the Number of Loops

Unrolling loops will considerably improve the speed of execution.

Unrolling loops is especially effective for inner loops. Since unrolling loops increases the sizes of programs, loops should be unrolled when fast execution is to take priority over the code size.

Before Unrolling	After Unrolling
<u>C source code</u> <pre>int a[100]; void func(void) { int i; for (i = 0; i < 100; i++) { a[i] = 0; } }</pre>	<u>C source code</u> <pre>int a[100]; void func(void) { int i; for (i = 0; i < 100; i += 2) { a[i] = 0; a[i+1] = 0; } }</pre>
<u>Assembly-language expanded code</u> <pre>_func: .STACK_func=4 MOV.L #00000064H, R15 MOV.L #_a, R14 MOV.L #00000000H, R5 L11: ; bb SUB #01H, R15 MOV.L R5, [R14+] BNE L11 L12: ; return RTS</pre>	<u>Assembly-language expanded code</u> <pre>_func: .STACK_func=4 MOV.L #00000032H, R14 MOV.L #_a, R1 L11: ; bb MOV.L #00000000H, [R1] MOV.L #00000000H, 04H[R1] ADD #08H, R1 SUB #01H, R14 BNE L11 L12: ; return RTS</pre>
Code size: 19 bytes Number of cycles: 603	Code size: 22 bytes Number of cycles: 353

Specifying the **-loop** option selects optimization in the form of unrolling loops. In the source code before improvement in the example above, when the **-loop** option is specified for compilation, assembly-language expanded code which is the same as that in the source code after the improvement is output. When the entire processing of the loop is unrolled, the loop's conditional expression is also deleted. For example, when the number of iterations is 8 and **-loop=8** is specified, the conditional expression is deleted.

4.9 Using Tables

Using tables instead of branching for **switch** statements will improve the speed of execution.

If the processing for each **case** label of a **switch** statement is almost the same, consider the use of a table.

In the example below, the character constant to be assigned to variable **ch** changes with the value of variable **i**.

switch Statement	Equivalent Table-Based Code
<p><u>C source code</u></p> <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>	<p><u>C source code</u></p> <pre> const char chbuf[] = {'a', 'x', 'b'}; char func(int i) { if ((unsigned int)i < 3) { return (chbuf[i]); } return (0); } </pre>
<p><u>Assembly-language expanded code</u></p> <pre> _func: .STACK_func=4 CMP #00H, R1 BEQ L14 L11: ; entry CMP #01H, R1 BEQ L15 L12: ; entry CMP #02H, R1 BEQ L16 L13: ; switch_clause_bb5 MOV.L #00000000H, R1 RTS L14: ; switch_break_bb MOV.L #00000061H, R1 RTS L15: ; switch_clause_bb3 MOV.L #00000078H, R1 RTS L16: ; switch_clause_bb4 MOV.L #00000062H, R1 RTS </pre>	<p><u>Assembly-language expanded code</u></p> <pre> _func: .STACK_func=4 CMP #02H, R1 BGTU L12 L11: ; if_then_bb MOV.L #_chbuf, R14 MOVU.B [R14,R1], R1 RTS L12: ; bb8 MOV.L #00000000H, R1 RTS </pre>
<p>Code size: 27 bytes Number of cycles: 7</p>	<p>Code size: 17 bytes Number of cycles: 5</p>

4.10 Branches

Changing the positions of cases for branching can improve the speed of execution. When comparison is performed in order beginning from the top, such as in an **else if** statement, the speed of execution for the cases at the end becomes slow if there are many preceding branches. Cases to which branching is frequent should be placed near the beginning of the sequence.

Before Changing the Position of a Case	After Changing the Position of a Case
<p><u>C source code</u></p> <pre> 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); } </pre>	<p><u>C source code</u></p> <pre> 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); } </pre>
<p><u>Assembly-language expanded code</u></p> <pre> _func: .STACK_func=4 CMP #01H, R1 STZ #02H, R1 BEQ L13 L11: ; if_else_bb CMP #02H, R1 STZ #04H, R1 BEQ L13 L12: ; if_else_bb9 CMP #03H, R1 SCEQ.L R1 SHLL #03H, R1 L13: ; if_break_bb17 RTS </pre>	<p><u>Assembly-language expanded code</u></p> <pre> _func: .STACK_func=4 CMP #03H, R1 STZ #08H, R1 BEQ L13 L11: ; if_else_bb CMP #02H, R1 STZ #04H, R1 BEQ L13 L12: ; if_else_bb9 CMP #01H, R1 SCEQ.L R1 SHLL #01H, R1 L13: ; if_break_bb17 RTS </pre>
<p>Number of cycles: 12 (for a=3)</p>	<p>Number of cycles: 7 (for a=3)</p>

4.11 Inline Expansion

The speed of execution can be improved by applying inline expansion to functions that are frequently called. The inline expansion of functions is specified by **#pragma inline**. However, inline expansion generally increases the sizes of programs.

When other source files do not refer to an inline-expanded function, change the function to a static function. Some code in the function will be removed and the code size may be reduced.

Before Inline Expansion	After Inline Expansion
<pre> C source code 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(void) { int i; for (i = 0; i < 10; i++) { sub(x, y, i); } } </pre>	<pre> C source code 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(void) { int i; for (i = 0; i < 10; i++) { sub(x, y, i); } } </pre>

<u>Assembly-language expanded code</u>	<u>Assembly-language expanded code</u>
<pre> __\$_sub: .STACK__\$sub=4 MOV.L [R3,R2], R14 MOV.L [R3,R1], R15 MOV.L R14, [R3,R1] MOV.L R15, [R3,R2] RTS _func: .STACK_func=16 PUSHM R6-R8 MOV.L #_y, R7 MOV.L #_x, R8 MOV.L #00000000H, R6 L12: ; bb MOV.L R8, R1 MOV.L R7, R2 MOV.L R6, R3 BSR __\$_sub ADD #01H, R6 CMP #0AH, R6 BNE L12 L13: ; return RTSD #0CH, R6-R8 </pre>	<pre> _func: .STACK_func=4 MOV.L #0000000AH, R5 MOV.L #_y, R14 MOV.L #_x, R15 L11: ; bb SUB #01H, R5 MOV.L [R14], R4 MOV.L [R15], R3 MOV.L R4, [R15+] MOV.L R3, [R14+] BNE L11 L12: ; return RTS </pre>
<p>Code size: 47 bytes Number of cycles: 189</p>	<p>Code size: 29 bytes Number of cycles: 84</p>

4.12 Moving Identical Expressions in More than One Conditional Branch Destination before the Conditional Branch

When there are identical expressions in more than one conditional branch destination, move and unify them into one section before the conditional branch.

Identical Expressions Following a Branch	Expression before the Branch
<p><u>C source code</u></p> <pre> 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); } </pre>	<p><u>C source code</u></p> <pre> 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; } </pre>

<u>Assembly-language expanded code</u>	<u>Assembly-language expanded code</u>
<pre> _call_func: .STACK_call_func=12 PUSHM R6-R7 ADD #00H, R1, R6 BN L17 L12: ; if_then_bb MOV.L #00000002H, R3 MOV.L #00000001H, R2 MOV.L #00000000H, R1 BSR _func CMP R6, R1 MOV.L #_s, R7 BGE L14 L13: ; if_then_bb9 MOV.L [R7], R14 MOV.L #00000000H, R1 ADD #01H, R14 MOV.L R14, [R7] BRA L16 L14: ; if_else_bb MOV.L #00000002H, R3 MOV.L #00000001H, R2 MOV.L #00000000H, R1 BSR _func NEG R1 CMP R1, R6 BGE L17 L15: ; if_then_bb18 MOV.L [R7], R14 MOV.L #00000000H, R1 SUB #01H, R14 MOV.L R14, [R7] L16: ; if_then_bb18 BRA L18 L17: ; if_break_bb22 MOV.L #00000000H, R1 L18: ; if_break_bb22 RTSD #08H, R6-R7 </pre>	<pre> _call_func: .STACK_call_func=8 PUSH.L R6 MOV.L R1, R6 MOV.L #00000002H, R3 MOV.L #00000001H, R2 MOV.L #00000000H, R1 BSR _func CMP #00H, R6 BN L17 L12: ; if_then_bb CMP R6, R1 MOV.L #_s, R14 BGE L14 L13: ; if_then_bb12 MOV.L #00000000H, R1 MOV.L [R14], R15 ADD #01H, R15 MOV.L R15, [R14] BRA L16 L14: ; if_else_bb NEG R1 CMP R1, R6 BGE L17 L15: ; if_then_bb21 MOV.L #00000000H, R1 MOV.L [R14], R15 SUB #01H, R15 MOV.L R15, [R14] L16: ; if_then_bb21 BRA L18 L17: ; if_break_bb25 MOV.L #00000000H, R1 L18: ; if_break_bb25 RTSD #04H, R6-R6 </pre>
<p>Code size: 65 bytes Number of cycles: 42</p>	<p>Code size: 57 bytes Number of cycles: 30</p>

4.13 Replacing a Sequence of Complicated if Statement with a Simple Statement Having the Same Logical Meaning

When a sequence of **if** statements and conditional expressions is complicated, replace them with a simple expression which has the same meaning.

Complicated Sequence	Single if Statement
<p><u>C source code</u></p> <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>	<p><u>C source code</u></p> <pre> int x; int func(int s, int t) { s &= 1; t &= 1; if (!(s ^ t)) { x = 1; } return (0); } </pre>
<p><u>Assembly-language expanded code</u></p> <pre> _func: .STACK_func=4 AND #01H, R2 BTST #00H, R1 MOV.L #_x, R14 BNE L12 L11: ; if_then_bb CMP #00H, R2 BNE L13 BRA L14 L12: ; if_else_bb CMP #00H, R2 BNE L14 L13: ; if_then_bb28 MOV.L #00000001H, [R14] L14: ; if_break_bb30 MOV.L #00000000H, R1 RTS </pre>	<p><u>Assembly-language expanded code</u></p> <pre> _func: .STACK_func=4 XOR R1, R2 BTST #00H, R2 BNE L12 L11: ; if_then_bb MOV.L #_x, R14 MOV.L #00000001H, [R14] L12: ; if_break_bb MOV.L #00000000H, R1 RTS </pre>
<p>Code size: 24 bytes Number of cycles: 12</p>	<p>Code size: 18 bytes Number of cycles: 8</p>

4.14 Converting short- or char-Type Variables into the int Type

In accord with the ANSI-C specification, the CC-RX compiler converts **short**- or **char**-type operations into the **int** type before generating code for the operations. Type conversion is also produced when an **int**-type value is substituted for a **short**- or **char**-type variable. Defining variables as the **int** type in the first place can reduce additional type conversion.

Note: When the type of a variable is converted into the **int** type, the range of variables or values obtained by the operation will be changed. If you change the type, take care that this does not affect the operation of the program.

char-Type Variables	int-Type Variables
<u>C source code</u> <pre>char func(char a, char b, char c) { char t = a + b; return (t >> c); }</pre>	<u>C source code</u> <pre>int func(int a, int b, int c) { int t = a + b; return (t >> c); }</pre>
<u>Assembly-language expanded code</u> <pre>_func: .STACK_func=4 ADD R1, R2 MOVU.B R2, R14 SHLR R3, R14 MOVU.B R14, R1 RTS</pre>	<u>Assembly-language expanded code</u> <pre>_func: .STACK_func=4 ADD R1, R2 MOV.L R2, R1 SHAR R3, R1 RTS</pre>
Code size: 10 bytes Number of cycles: 7	Code size: 8 bytes Number of cycles: 6

4.15 Unifying Common case Processing in switch Statements

When the branch destinations of multiple **case** labels have the same processing, move the **case** labels and unify the processing.

Same Processing at Multiple Destinations	Unified Processing
<p><u>C source code</u></p> <pre> int x; 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; } } </pre>	<p><u>C source code</u></p> <pre> int x; void func(void) { switch(x) { case 0: case 1: case 2: dummy1(); break; case 3: case 4: dummy2(); break; default: break; } } </pre>

<u>Assembly-language expanded code</u>	<u>Assembly-language expanded code</u>
<pre> _func: .STACK_func=4 MOV.L #_x, R14 MOV.L [R14], R14 CMP #00H, R14 BEQ L18 L13: ; entry CMP #01H, R14 BEQ L18 L14: ; entry CMP #02H, R14 BEQ L18 L15: ; entry CMP #03H, R14 BEQ L19 L16: ; entry CMP #04H, R14 BEQ L19 L17: ; return RTS L18: ; switch_clause_bb2 BRA _dummy1 L19: ; switch_clause_bb4 BRA _dummy2 </pre>	<pre> _func: .STACK_func=4 MOV.L #_x, R14 MOV.L [R14], R14 CMP #03H, R14 BLTU L15 L13: ; entry SUB #03H, R14 CMP #02H, R14 BLTU L16 L14: ; return RTS L15: ; switch_clause_bb BRA _dummy1 L16: ; switch_clause_bb1 BRA _dummy2 </pre>
<p>Code size: 30 bytes Number of cycles: 20</p>	<p>Code size: 23 bytes Number of cycles: 15</p>

4.16 Replacing for Loops with do-while Loops

Replacing a **for** statement with a **do-while** statement if it is clear that the loop is executed at least once may reduce the code size. Replacing another kind of conditional expression with an equality or inequality operator may also reduce the code size.

for Loop	do-while Loop
<p><u>C source code</u></p> <pre>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; } }</pre>	<p><u>C source code</u></p> <pre>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); }</pre>
<p><u>Assembly-language expanded code</u></p> <pre>_func: .STACK_func=4 MUL R1, R2 MOV.L #_array, R15 MOV.L #00000000H, R14 MOV.L #00000000H, R5 L11: ; bb13 CMP R2, R14 BGE L13 L12: ; bb MOV.L R5, [R15+] ADD #01H, R14 BRA L11 L13: ; return RTS</pre>	<p><u>Assembly-language expanded code</u></p> <pre>_func: .STACK_func=4 MUL R1, R2 MOV.L #_array, R15 MOV.L #00000000H, R14 MOV.L #00000000H, R5 L11: ; bb ADD #01H, R14 CMP R2, R14 MOV.L R5, [R15+] BLT L11 L12: ; return RTS</pre>
<p>Code size: 24 bytes Number of cycles: 458</p>	<p>Code size: 18 bytes Number of cycles: 324</p>

4.17 Replacing Division by Powers of Two with Shift Operations

If it is clear that the divisor in division is a power of two and the dividend is a positive value, replace the division with a shift operation.

Division by a Power of Two	Shift Operation
<u>C source code</u> <pre>int s; void func(void) { s = s / 2; }</pre>	<u>C source code</u> <pre>int s; void func(void) { s = s >> 1; }</pre>
<u>Assembly-language expanded code</u> <pre>_func: .STACK_func=4 MOV.L #_s, R14 MOV.L [R14], R15 DIV #02H, R15 MOV.L R15, [R14] RTS</pre>	<u>Assembly-language expanded code</u> <pre>_func: .STACK_func=4 MOV.L #_s, R14 MOV.L [R14], R15 SHAR #01H, R15 MOV.L R15, [R14] RTS</pre>
Code size: 15 bytes Number of cycles: 10	Code size: 11 bytes Number of cycles: 8

4.18 Changing Bit Fields with Two or More Bits to the char Type

When a bit field has two or more bits, change the bit field to the **char** type. Note, however, that this will increase the amount of ROM in use.

Bit Fields	char
<u>C source code</u> <pre> struct { unsigned char b0:1; unsigned char b1:2; } dw; unsigned char dummy; int func(void) { if (dw.b1) { dummy++; } return (0); } </pre>	<u>C source code</u> <pre> struct { unsigned char b0:1; unsigned char b1; } db; unsigned char dummy; int func(void) { if (db.b1) { dummy++; } return (0); } </pre>
<u>Assembly-language expanded code</u> <pre> _func: .STACK_func=4 MOV.L #00000006H, R15 MOV.L #_dw, R14 TST [R14].UB, R15 BNE L12 L11: ; if_break_bb MOV.L #00000000H, R1 RTS L12: ; if_then_bb MOV.L #00000000H, R1 MOV.L #_dummy, R14 MOVU.B [R14], R15 ADD #01H, R15 MOV.B R15, [R14] RTS </pre>	<u>Assembly-language expanded code</u> <pre> _func: .STACK_func=4 MOV.L #_db, R1 MOVU.B 01H[R1], R1 CMP #00H, R1 BNE L12 L11: ; if_break_bb MOV.L #00000000H, R1 RTS L12: ; if_then_bb MOV.L #00000000H, R1 MOV.L #_dummy, R14 MOVU.B [R14], R15 ADD #01H, R15 MOV.B R15, [R14] RTS </pre>
Code size: 30 bytes ROM size: 1 byte Number of cycles: 10	Code size: 29 bytes ROM size: 2 bytes Number of cycles: 9

4.19 Assigning Small Absolute Values when Referring to Constants

When referring to constants, assigning a small absolute value may reduce the code size. When constant values are used to assign IDs, use numbers with small absolute values.

Larger Value	Smaller Value
<u>C source code</u> <pre>#define ID_1 (1000) int id; void func(void) { id = ID_1; }</pre>	<u>C source code</u> <pre>#define ID_1 (1) int id; void func(void) { id = ID_1; }</pre>
<u>Assembly-language expanded code</u> <pre>_func: .STACK_func=4 MOV.L #_id, R14 MOV.L #000003E8H, [R14] RTS</pre>	<u>Assembly-language expanded code</u> <pre>_func: .STACK_func=4 MOV.L #_id, R14 MOV.L #00000001H, [R14] RTS</pre>
Code size: 11 bytes	Code size: 10 bytes

Website and Support

Renesas Electronics Website

<http://www.renesas.com/>

Inquiries

<http://www.renesas.com/contact/>

All trademarks and registered trademarks are the property of their respective owners.

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	May 17, 2018		New release

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: "Standard" and "High Quality". The intended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below.
"Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
"High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.
Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user's manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user's manuals, application notes, "General Notes for Handling and Using Semiconductor Devices" in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note 2) "Renesas Electronics product(s)" means any product developed or manufactured by or for Renesas Electronics.

(Rev.4.0-1 November 2017)



SALES OFFICES

Renesas Electronics Corporation

<http://www.renesas.com>

Refer to "<http://www.renesas.com/>" for the latest and detailed information.

Renesas Electronics America Inc.

1001 Murphy Ranch Road, Milpitas, CA 95035, U.S.A.
Tel: +1-408-432-8888, Fax: +1-408-434-5351

Renesas Electronics Canada Limited

9251 Yonge Street, Suite 8309 Richmond Hill, Ontario Canada L4C 9T3
Tel: +1-905-237-2004

Renesas Electronics Europe Limited

Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-651-700, Fax: +44-1628-651-804

Renesas Electronics Europe GmbH

Arcadiastrasse 10, 40472 Düsseldorf, Germany
Tel: +49-211-6503-0, Fax: +49-211-6503-1327

Renesas Electronics (China) Co., Ltd.

Room 1709 Quantum Plaza, No.27 ZhichunLu, Haidian District, Beijing, 100191 P. R. China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.

Unit 301, Tower A, Central Towers, 555 Langao Road, Putuo District, Shanghai, 200333 P. R. China
Tel: +86-21-2226-0888, Fax: +86-21-2226-0999

Renesas Electronics Hong Kong Limited

Unit 1601-1611, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2265-6688, Fax: +852 2886-9022

Renesas Electronics Taiwan Co., Ltd.

13F, No. 363, Fu Shing North Road, Taipei 10543, Taiwan
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.

80 Bendemeer Road, Unit #06-02 Hyflux Innovation Centre, Singapore 339949
Tel: +65-6213-0200, Fax: +65-6213-0300

Renesas Electronics Malaysia Sdn.Bhd.

Unit 1207, Block B, Menara Amcorp, Amcorp Trade Centre, No. 18, Jln Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: +60-3-7955-9390, Fax: +60-3-7955-9510

Renesas Electronics India Pvt. Ltd.

No.777C, 100 Feet Road, HAL 2nd Stage, Indiranagar, Bangalore 560 038, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.

17F, KAMCO Yangjae Tower, 262, Gangnam-daero, Gangnam-gu, Seoul, 06265 Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5338