RX-family C/C++ Compiler Package

REJ06J0099-0100
Rev.1.00
Apr 20, 2010

Application Notes: Compiler Usage Guide
Tips for Efficient Programming Edition

This document introduces techniques for efficient programming for version 1.0 of the RX-family C/C++ compiler.

Table of contents

1. Introduction 2

2. Specifying data 4
2.1 Data structures 4
2.2 Variables and the const type 5
2.3 Local variables and global variables 6
2.4 Member offsets for structure declarations 7
2.5 Bit field allocation 8
2.6 Loop control variables 9
2.7 External variable access optimization during base register specification 11
2.8 Specification order for linker section addresses during external variable access optimization 13

3. Function calls 15
3.1 Function modularization 15
3.2 Function interfaces 16

4. Calculation methods 18
4.1 Reducing loop iterations 18
4.2 Making use of tables 21

5. Branching 23

6. Interrupts 25

7. Inline expansion 27
Website and Support 29

1. Introduction

The RX-family C/C++ compiler performs its own optimizations, but programming techniques can be used to increase performance even more.

This document introduces techniques that we would like users to try in order to create efficient programs.

There are two ways to evaluate a program: by how fast it executes, and by how small it is. The principles for creating efficient programs are as follows:

(1) Principles for improving execution speed

Since execution speed depends on frequently executed and complex statements, make sure you understand and focus on what they process.

(2) Principles for reducing size

To reduce program size, factor out common processing, and refactor complex functions.

In addition to the code generated by the compiler, the execution speed in production changes due to such factors as the memory architecture and interrupts. Try running the various techniques introduced in this document, check their results, and apply what works.

The expanded assembly language code in this document can be obtained from the command line using the RX-family C/C++ compiler as follows:

ccrx△*<C-language-file>*△**-output=src**△**-cpu=rx600**

Note that expanded assembly language code may change due to future improvements to programs existing before or after the improvements, or to the compiler.

The execution speed for code in this document can be measured using the simulator debugger included with the compiler package. Note that the number of cycles for external memory access is measured as 1. Use these measurement results as values for reference.

Table 1-1 lists techniques for efficient programming.

Table 1-1 List of techniques for efficient programming

No.	Item	ROM efficiency	RAM efficiency	Execution speed	Ref.
1	Data structures	Good	--	Good	*2.1*
2	Variables and the const type	--	Good	--	*2.2*
3	Local variables and global variables	Good	--	Good	*2.3*
4	Member offsets for structure declarations	Good	--	--	*2.4*
5	Bit field allocation	Good	--	--	*2.5*
6	Loop control variables	Poor	--	Good	*2.6*
7	External variable access optimization during base register specification	Good	--	Good	*2.7*
8	Specification order for linker section addresses during external variable access optimization	Good	--	Good	*2.8*
9	Function modularization	Good	--	--	*3.1*
10	Function interfaces	--	Good	Good	*3.2*
11	Reducing loop iterations	Poor	--	Good	*4.1*
12	Making use of tables	Poor	--	Good	*4.2*
13	Branching	--	--	Good	*5*
14	Interrupts	Good	--	Good	*6*
15	Inline expansion	--	--	Good	*7*

Legend:
Good: Improves performance
Poor: May degrade performance

2. Specifying data

Table 2-1 lists items that need to be kept in mind regarding data.

Table 2-1 Precautions when specifying data

Item	Precaution	Ref.
Data type specifiers, types, and modifiers	• Attempts to decrease data size may result in increased program size. Keep the purpose of the data in mind when performing type declarations. • Keep in mind that program size may change according to whether data is signed or unsigned. • For initialization data with values that do not change within a program, add the const operator to reduce memory usage.	*2.2*
Data consistency	• Allocate data areas to prevent wasted space.	
Defining and viewing structures	• Pointer variables can be used for structures with data that is frequently accessed or changed to reduce the program size. • Bit fields can be used to shrink the data size.	*2.1*
Making use of internal ROM/RAM	• Since internal memory is much faster to access than external memory, store as many common variables as possible in internal memory.	--

2.1 Data structures

■ Overview

Related data can be declared in a structure to improve execution speed.

■ Description

When related data is referenced repeatedly in the same function, structures can be used to improve efficiency by facilitating the creation of code that uses relative access, and making it easier to pass by argument. Since the access scope is limited for relative access, aggregating frequently accessed data at the beginning of a structure is effective.

Structuring data makes it easier to perform tuning that changes the data representation.

■ Example usage

The following substitutes numbers for variables a, b, and c.

Source code before

```
int a, b, c;

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

Source code after

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

void func()
{
        register struct s *p=&s1;
        p->a = 1;
        p->b = 2;
        p->c = 3;
}
```

Expanded assembly code before	Expanded assembly code after

<pre>_ func:
        MOV.L   #_a,R4
        MOV.L   #00000001H,[R4]
        MOV.L   #_b,R4
        MOV.L   #00000002H,[R4]
        MOV.L   #_c,R4
        MOV.L   #00000003H,[R4]
        RTS</pre>
	<pre>_func:
	        MOV.L #_s1,R5
	        MOV.L #00000001H,[R5]
	        MOV.L #00000002H,04H[R5]
	        MOV.L #00000003H,08H[R5]
	        RTS</pre>

■ Code size and execution speed before and after

CPU type	Code size (in bytes)		Execution speed (in cycles)	
	Before	After	Before	After
RX610	28	15	9	7

2.2 Variables and the const type

■ Overview

Use the const type to declare variables with unchanging values.

■ Description

Variables with initial values are usually transferred from the ROM area to the RAM area at startup, and processed using the RAM area. Therefore, when programs contain initialization data within unchanging values, this secured RAM area is wasted. The const operator can be added to such initialization data to conserve used memory by preventing transfer during startup to the RAM area.

In addition, using ROM is easier when programs are created based on the premise that initial values do not change.

■ Example usage

The following sets 5 items of initialization data.

Source code before	Source code after

<pre>char a[] =
    {1, 2, 3, 4, 5};</pre>
	<pre>const char a[] =
	   {1, 2, 3, 4, 5};</pre>

Initial values are copied from ROM to RAM and then processed.	Initial values are processed as is in ROM.

2.3 Local variables and global variables

■ Overview

Locally used variables such as temporary variables and loop counters can be declared as local variables within functions to improve execution speed.

■ Description

Make sure that anything that can be used as a local variable is declared as a local variable, not as a global variable. Since the values of global variables can change due to function calls and pointer operations, they reduce the efficiency of optimizations.

Using local variables provides the following benefits:

a. They are inexpensive to access.

b. They can be allocated to a register.

c. They are efficiently optimized.

■ Example usage

The following shows cases in which a global variable (before) and a local variable (after) is used as a temporary variable.

Source code before

```
int tmp;

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

Expanded assembly code before

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

Source code after

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

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

Expanded assembly code after

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

■ Code size and execution speed before and after

CPU type	Code size (in bytes)		Execution speed (in cycles)	
	Before	After	Before	After
RX610	13	7	13	9

2.4 Member offsets for structure declarations

■ Overview

Code size can be improved by declaring frequently used members of structure at the beginning.

■ Description

Structure members are accessed by incrementing the offset of the structure address. Since smaller offsets mean smaller sizes, declare frequently used members first.

The most efficient cases are when the signed char and unsigned char types are within the first 32 bytes, short and unsigned short types are within the first 64 bytes, and int, unsigned, long, and unsigned long types are within the first 128 bytes.

■ Example usage

The following shows an example in which the code changes based on structure offset.

Source code before

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

struct str STR1;
char x;

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

Expanded assembly code before

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

Source code after

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

struct str STR1;
char x;

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

Expanded assembly code after

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

■ Code size and execution speed before and after

CPU type	Code size (in bytes)		Execution speed (in cycles)	
	Before	After	Before	After
RX610	18	17	8	8

■ Precautions

When defining a structure, keep the alignment count in mind when declaring members.

The alignment count of a structure matches the largest alignment value in the structure, so that the size of the structure is a multiple of the alignment count. This means that, because the next alignment is guaranteed the size of an unused area is included when the end of a structure does not match the alignment count of the structure itself.

Source code before

```
  /* Since the largest member is an int type, the
alignment is 4 */
  struct str {
        char C1; /* 1 byte + 3 bytes for alignment
*/
        long L1; /* 4 bytes */
        char C2; /* 1 byte  */
        char C3; /* 1 byte  */
        char C4; /* 1 byte + 1 byte for alignment
*/
  }STR1;
```

str size before

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

Source code after

```
  /* Since the largest member is an int type, the
alignment is 4 */
  struct str {
        char C1; /* 1 byte  */
        char C2; /* 1 byte  */
        char C3; /* 1 byte  */
        char C4; /* 1 byte  */
        long L1; /* 4 bytes */
  }STR1;
```

str size after

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

2.5 Bit field allocation

■ Overview

Make sure that consecutively set bit fields are allocated within the same structure.

■ Description

The data in a bit field needs to be accessed each time a different bit field member is accessed. By allocating related bit fields together within the same structure, this access can be completed in one run.

■ Example usage

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

Source code before

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

Source code after

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

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

Expanded assembly code before

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

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

Expanded assembly code after

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

■ Code size and execution speed before and after

CPU type	Code size (in bytes)		Execution speed (in cycles)	
	Before	After	Before	After
RX610	25	13	14	9

2.6 Loop control variables

■ Overview

Changing loop control variables to a 4-byte integer type (signed long/unsigned long) facilitates loop expansion optimizations, which can improve execution speed.

■ Description

Loop expansion optimizations cannot be performed when, during evaluation of a loop termination condition, a difference in size prevents a loop control variable from expressing the compared data. For example, when the loop control variable is a signed char, but the compared data is a signed long, loop expansion optimization is not performed. Accordingly, loop expansion optimization is more easily applied for signed long types than for signed char or signed short types. To take advantage of loop expansion optimization, use a 4-byte integer type for loop control variables.

■ Example usage

```
Source code before

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

void func()
{
```

```
Source code after

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

void func()
{
```

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

Expanded assembly code before
<when loop=2 is specified>

```
_func:
          MOV.L     #_array_size,R4
          MOV.L     [R4],R2
          MOV.L     #00000000H,R5
          BRA       L11
L12:
          MOV.L     #_array,R4
          MOV.L     #00000000H,R3
          MOV.B     R3,[R5,R4]
          ADD       #01H,R5
L11:
          MOV.B     R5,R5
          CMP       R2,R5
          BLT       L12
L13:
          RTS
```

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

Expanded assembly code after
<when loop=2 is specified>

```
_func:
          MOV.L     #_array_size,R5
          MOV.L     [R5],R2
          MOV.L     #00000000H,R4
          ADD       #0FFFFFFFFH,R2,R3
          CMP       R3,R2
          BLE       L12
L11:
          MOV.L     #_array,R1
          MOV.L     R1,R5
          BRA       L13
L14:
          MOV.W     #0000H,[R5]
          ADD       #02H,R5
          ADD       #02H,R4
L13:
          CMP       R3,R4
          BLT       L14
L15:
          CMP       R2,R4
          BGE       L17
L16:
          MOV.L     #00000000H,R5
          MOV.B     R5,[R4,R1]
          RTS
L12:
          MOV.L     #_array,R5
          MOV.L     #00000000H,R3
L19:
          CMP       R2,R4
          BGE       L17
L20:
          MOV.B     R3,[R5+]
          ADD       #01H,R4
          BRA       L19
```

```
L17:
        RTS
```

■ Code size and execution speed before and after

CPU type	Code size (in bytes)		Execution speed (in cycles)	
	Before	After	Before	After
RX610	32	67	171	82

2.7 External variable access optimization during base register specification

■ Overview

When a specific register is used as a base register during access to ROM/RAM sections across an entire project, it can be combined with external variable access optimization to reduce code size.

■ Description

When R13 is specified for the base register of the RAM section, access to the RAM section is performed relative to the R13 register. Also, if external variable success optimization between modules is enabled, the relative values for the R13 register are optimized, so that instruction sizes are reduced for values within the 8-bit range. The base register can be specified from the HEW menu by choosing Build -> RX Standard ToolChain -> CPU -> Base register, as shown in Figure 1.

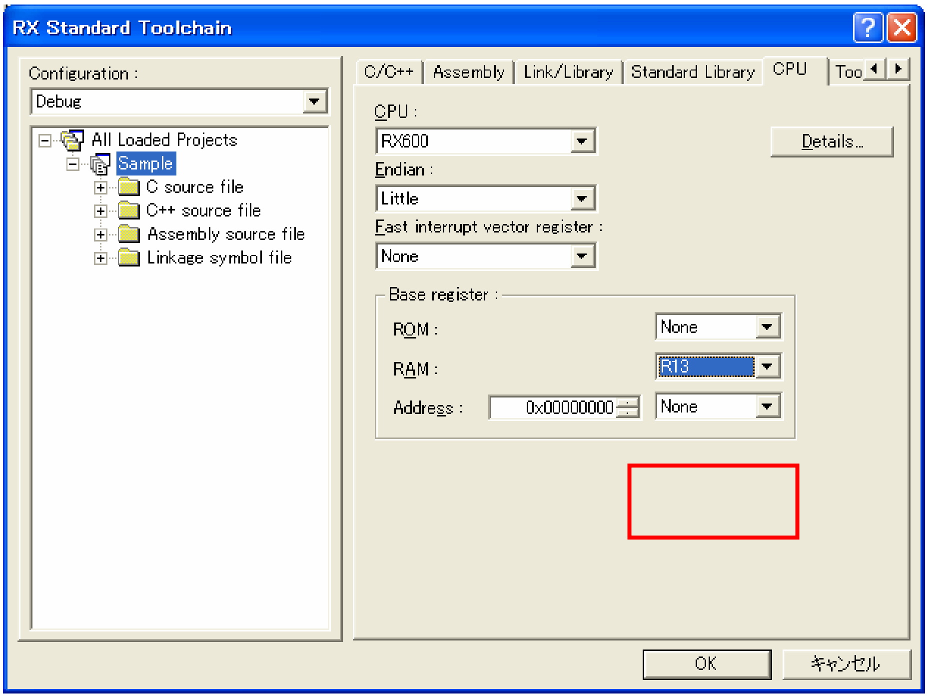

Figure 1 Base register setting

■ Example usage

Source code before

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

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

Expanded assembly code before

```
_func:
          MOV.L      #_a,R4
          MOV.L      #00000000H,[R4]
          MOV.L      #_b,R4
          MOV.L      #00000001H,[R4]
          MOV.L      #_c,R4
          MOV.L      #00000002H,[R4]
          MOV.L      #_d,R4
          MOV.L      #00000003H,[R4]
          RTS
```

Source code after

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

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

Expanded assembly code after

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

■ Code size and execution speed before and after

CPU type	Code size (in bytes)		Execution speed (in cycles)	
	Before	After	Before	After
RX610	14	10	11	7

2.8 Specification order for linker section addresses during external variable access optimization

■ Overview

When external variable access optimization is enabled, the order of section allocation in the linker can be changed to reduce code size.

■ Description

For instructions that use the relative register format to access memory, instruction size can be reduced by using smaller displacement values. The section allocation order in the linker can be changed as follows to improve code size:

- Moving sections that frequently access external variables earlier within a function.
- Moving sections with external variables that have small type sizes earlier.

Note that, since external variable access optimization requires compilation twice, the build time might be longer.

■ Example usage

Source code before

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

void func(int a){
       d11 = a;
       d12 = a;
       d13 = a;
       d14 = a;
       d21 = a;
       d22 = a;
       d23 = a;
       d24 = a;
       d41 = a;
       d42 = a;
       d43 = a;
       d44 = a;
}
```

Expanded assembly code before

<When the section allocation order is D,D_2,D_1 or D*>

Source code after

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

void func(int a){
       d11 = a;
       d12 = a;
       d13 = a;
       d14 = a;
       d21 = a;
       d22 = a;
       d23 = a;
       d24 = a;
       d41 = a;
       d42 = a;
       d43 = a;
       d44 = a;
}
```

Expanded assembly code after

<When the section allocation order is D_1,D_2,D>

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

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

■ Code size and execution speed before and after

CPU type	Code size (in bytes)		Execution speed (in cycles)	
	Before	After	Before	After
RX610	43	31	20	18

3. Function calls

Table 3-1 lists things to keep in mind regarding function calls.

Table 3-1 Precautions regarding function calls

Item	Precaution	Ref.
Function position	• Keep tightly coupled functions together in the same file.	*3.1*
Interfaces	• Be strict with regard to the number of arguments (up to 4), so that all arguments can be allocated to the register. • For functions with many arguments, put the arguments in a structure and pass them by pointer.	*3.2*
Macro substitution	• When many function calls exist, their execution speed can be improved by macro substitution. Note that macros will increase the program size, so use only as appropriate.	--

3.1 Function modularization

■ Overview

Size can be improved by grouping tightly coupled functions into a single file.

■ Description

When functions in different files are called, they are expanded into 4-byte BSR instructions, but when functions in the same file are called, they are expanded into 3-byte BSR instructions when the call scope is close, allowing compact objects to be generated.

Also, modularization facilitates corrections during tune-ups.

■ Example usage

In this example, function g is called from function f.

Source code before

```
extern void sub(void);



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

Expanded assembly code before

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

Source code after

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

Expanded assembly code after

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

■ Code size and execution speed before and after

CPU type	Code size (in bytes)		Execution speed (in cycles)	
	Before	After	Before	After
RX610	7	6	9	9

3.2 Function interfaces

■ Overview

Adjusting function arguments can decrease RAM consumption and improve execution speed.
For details, see *8.2 Function call interfaces* in the compiler manual.

■ Description

Be selective about argument counts, so that all arguments (up to 4) fit within the register. When using many arguments, put them in a structure and then pass it as a pointer. If the structure itself is passed rather than as a pointer, the structure might not be able to fit in the register when received. Making sure that arguments fit in the register simplifies processing for call and function entry and exit points, and helps to conserve stack area.

Note that registers R1 to R4 are used for arguments.

■ Example usage

In the following, function f has four more arguments than the number of in the registers available for arguments.

Source code before

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

Source code after

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

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

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

Expanded assembly code before

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

Expanded assembly code after

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

■ Code size and execution speed before and after

CPU type	Code size (in bytes)		Execution speed (in cycles)	
	Before	After	Before	After
RX610	16	8	16	4

4. Calculation methods

Table 4-1 lists items to keep in mind regarding calculation methods.

Table 4-1 Precautions regarding calculation methods

Item	Precaution	Ref.
Reducing loop iterations	■ Consider merging loop statements with identical or similar loop conditions. • Try loop expansion.	*4.1*
Using fast algorithms	■ Consider algorithms that do not require much execution time, such as quicksort for arrays.	--
Making use of tables	■ Consider using tables for switch statements in which the processing for each case is nearly identical. ■ Execution speed can often be improved by storing results calculated ahead of time in a table, and then referencing the table values when a calculation result is needed. However, note in this case that ROM space will increase, so decide based on both required execution speed and available ROM space.	*0*
Conditional expressions	Comparisons of constants to 0 helps to generate efficient code.	--

4.1 Reducing loop iterations

■ Overview

Loops can be expanded to greatly improve execution speed.

■ Description

Loop expansion is particularly effective for inner loops. Since loop expansion increases program size, apply it only to improve execution speed despite the cost in program size.

■ Example usage

The following initializes array `a[]`.

Source code before

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

Source code after

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

Expanded assembly code before

```
_func:
          MOV.L      #00000064H,R4
          MOV.L      #_a,R5
          MOV.L      #00000000H,R3
```

Expanded assembly code before

```
_func:
          MOV.L      #00000032H,R4
          MOV.L      #_a,R5
L11:
```

```
L11:                                     MOV.L     #00000000H,[R5]
          MOV.L    R3,[R5+]              MOV.L     #00000000H,04H[R5]
          SUB      #01H,R4               ADD       #08H,R5
          BNE      L11                   SUB       #01H,R4
L12:                                     BNE       L11
          RTS                        L12:
                                         RTS
```

■ Code size and execution speed before and after

CPU type	Code size (in bytes)		Execution speed (in cycles)	
	Before	After	Before	After
RX610	19	22	504	353

■ Notes

Specify loop options to perform loop expansion optimization. When the loop option is specified in the before source code below and the code is compiled, the same assembly expansion code is output as the assembly expansion code in the after source code.

Source code before

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

Expanded assembly code before

```
<loop=2 指定時>

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

Source code after

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

Expanded assembly code after

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

4.2 Making use of tables

■ Overview

Execution speed can be improved by using tables instead of branch switch statements.

■ Description

Consider using tables when the processing for each case in a switch statement is largely the same.

■ Example usage

In the following, the character constant replacing variable `ch` changes depending on the value of variable `i`.

Source code before

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

Source code after

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

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

Expanded assembly code before

```
_func:
          CMP        #00H,R1
          BEQ        L17
L16:
          CMP        #01H,R1
          BEQ        L19
L18:
          CMP        #02H,R1
          BEQ        L20
          BRA        L21
L12:
L17:
          MOV.L      #00000061H,R1
          BRA        L21
L13:
L19:
          MOV.L      #00000078H,R1
```

Expanded assembly code after

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

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

■ Code size and execution speed before and after

CPU type	Code size (in bytes)		Execution speed (in cycles)	
	Before	After	Before	After
RX610	28	10	13	6

Note: For i=2

5. Branching

■ Overview

Execution speed can be improved by changing the placement of branch cases.

■ Description

When an else if statement is used to perform comparison in order, the execution speed of the terminal case suffers when the cases grow. Therefore, place frequently branched cases first.

■ Example usage

In the following, the return value differs depending on the argument value.

Source code before

```
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);
}
```

Source code after

```
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);
}
```

Expanded assembly code before

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

Expanded assembly code after

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

`MOV.L #00000000H,R1`
`RTS`
`L11:`
`MOV.L #00000002H,R1`
`RTS`
	`MOV.L #00000000H,R1`
	`RTS`
	`L11:`
	`MOV.L #00000008H,R1`
	`RTS`

Code size and execution speed before and after

CPU type	Code size (in bytes)		Execution speed (in cycles)	
	Before	After	Before	After
RX610	22	22	11	7

Note: For a=3

6. Interrupts

■ Overview

Fast interrupt functionality can be used to reduce interrupt response times.

■ Description

Expected interrupt response times might not be achieved when many registers are saved or restored before or after interrupt processing. In such cases, the fast interrupt specification (fint) can be used along with the fint_register option to prevent register saving and restoration thereby shortening interrupt response time.

However, keep in mind that, since fewer registers can be used by other functions when the fint_register option is used, overall program efficiency might suffer.

■ Example usage

Source code before

```
#pragma interrupt int_func

volatile int count;

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

Expanded assembly code before

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

Source code after

```
#pragma interrupt int_func(fint)

volatile int count;

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

Expanded assembly code after

<When the fint_register=2 option is specified>

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

■ Code size and execution speed before and after

CPU type	Code size (in bytes)		Execution speed (in cycles)	
	Before	After	Before	After
RX610	18	14	23	14

7. Inline expansion

■ Overview

Execution speed can be improved by expanding frequently called functions inline.

■ Description

Execution speed can be improved by expanding frequently called functions inline. This is especially true for functions called within loops. However, since inline expansion can result in increased program size, apply this method only to improve execution speed despite the cost in program size.

■ Example usage

The following switches the elements in array a and array b.

Source code before

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

Expanded assembly code before

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

_func:
```

Source code after

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

Expanded assembly code after

```
; sub code has decreased due to
; inline expansion
```

```
        PUSHM      R6-R8                    _func:
        MOV.L      #00000000H,R6                    MOV.L      #0000000AH,R1
        MOV.L      #_x,R7                           MOV.L      #_y,R2
        MOV.L      #_y,R8                           MOV.L      #_x,R3
L12:                                        L11:
        MOV.L      R6,R3                            MOV.L      [R3],R4
        MOV.L      R7,R1                            MOV.L      [R2],R5
        MOV.L      R8,R2                            MOV.L      R4,[R2+]
        ADD        #01H,R6                          MOV.L      R5,[R3+]
        BSR        __$sub                           SUB        #01H,R1
        CMP        #0AH,R6                          BNE        L11
        BLT        L12                      L12:
L13:                                                RTS
        RTSD       #0CH,R6-R8
```

■ Code size and execution speed before and after

CPU type	Code size (in bytes)		Execution speed (in cycles)	
	Before	After	Before	After
RX610	47	29	119	84

Website and Support

- Renesas Electronics Website
  http://www.renesas.com/
- Inquiries
  http://www.renesas.com/inquiry

Revision Record

Rev.	Date	Description	
		Page	Summary
1.00	Apr.20.10	—	First edition issued
—	—	—	—
—	—	—	—
—	—	—	—
—	—	—	—
—	—	—	—
—	—	—	—
—	—	—	—

RENESAS

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.
2880 Scott Boulevard Santa Clara, CA 95050-2554, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

Renesas Electronics Canada Limited
1101 Nicholson Road, Newmarket, Ontario L3Y 9C3, Canada
Tel: +1-905-898-5441, Fax: +1-905-898-3220

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-585-100, Fax: +44-1628-585-900

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

Renesas Electronics (China) Co., Ltd.
7th Floor, Quantum Plaza, No.27 ZhiChunLu Haidian District, Beijing 100083, P.R.China
Tel: +86-10-8235-1155, Fax: +86-10-8235-7679

Renesas Electronics (Shanghai) Co., Ltd.
Unit 204, 205, AZIA Center, No.1233 Lujiazui Ring Rd., Pudong District, Shanghai 200120, China
Tel: +86-21-5877-1818, Fax: +86-21-6887-7858 / -7898

Renesas Electronics Hong Kong Limited
Unit 1601-1613, 16/F., Tower 2, Grand Century Place, 193 Prince Edward Road West, Mongkok, Kowloon, Hong Kong
Tel: +852-2886-9318, Fax: +852 2886-9022/9044

Renesas Electronics Taiwan Co., Ltd.
7F, No. 363 Fu Shing North Road Taipei, Taiwan, R.O.C.
Tel: +886-2-8175-9600, Fax: +886 2-8175-9670

Renesas Electronics Singapore Pte. Ltd.
1 harbourFront Avenue, #06-10, keppel Bay Tower, Singapore 098632
Tel: +65-6213-0200, Fax: +65-6278-8001

Renesas Electronics Malaysia Sdn.Bhd.
Unit 906, 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 Korea Co., Ltd.
11F., Samik Lavied' or Bldg., 720-2 Yeoksam-Dong, Kangnam-Ku, Seoul 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141