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:

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

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>
<thead>
<tr>
<th>Item</th>
<th>Precaution</th>
<th>Ref.</th>
</tr>
</thead>
</table>
| 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 \( c \).

<table>
<thead>
<tr>
<th>Source code before</th>
<th>Source code after</th>
</tr>
</thead>
</table>
| int \( a \), \( b \), \( c \); | struct \( s \) {
| void func() |  
| { |  
| \( a = 1; \) |  
| \( b = 2; \) |  
| \( c = 3; \) |  
| } |  
| } |  
| void func() |  
| { |  
| register struct \( s *p=s1; \) |  
| \( p->a = 1; \) |  
| \( p->b = 2; \) |  
| \( p->c = 3; \) |  
| } |
Expanding assembly code before \_func:

\begin{verbatim}
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
\end{verbatim}

Expanding assembly code after \_func:

\begin{verbatim}
MOV.L   #_s1,R5
MOV.L   #00000001H,[R5]
MOV.L   #00000002H,04H[R5]
MOV.L   #00000003H,08H[R5]
RTS
\end{verbatim}

- Code size and execution speed before and after

<table>
<thead>
<tr>
<th>CPU type</th>
<th>Code size (in bytes)</th>
<th>Execution speed (in cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>RX610</td>
<td>28</td>
<td>15</td>
</tr>
</tbody>
</table>

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.

\begin{verbatim}
char a[] =
    {1, 2, 3, 4, 5};
\end{verbatim}

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

\begin{verbatim}
const char a[] =
    {1, 2, 3, 4, 5};
\end{verbatim}

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.

<table>
<thead>
<tr>
<th>Source code before</th>
<th>Source code after</th>
</tr>
</thead>
<tbody>
<tr>
<td>int tmp;</td>
<td>void func(int* a, int* b)</td>
</tr>
<tr>
<td>void func(int* a, int* b)</td>
<td>{ int tmp;</td>
</tr>
<tr>
<td></td>
<td>tmp = *a;</td>
</tr>
<tr>
<td></td>
<td>*a = *b;</td>
</tr>
<tr>
<td></td>
<td>*b = tmp;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expanded assembly code before</th>
<th>Expanded assembly code after</th>
</tr>
</thead>
<tbody>
<tr>
<td>__func:</td>
<td>__func:</td>
</tr>
<tr>
<td>MOV.L  #_tmp,R4</td>
<td>MOV.L  [R1],R5</td>
</tr>
<tr>
<td>MOV.L  [R1],[R4]</td>
<td>MOV.L  [R2],[R1]</td>
</tr>
<tr>
<td>MOV.L  [R2],[R1]</td>
<td>MOV.L  R5,[R2]</td>
</tr>
<tr>
<td>MOV.L  [R4],[R2]</td>
<td>RTS</td>
</tr>
</tbody>
</table>

■ Code size and execution speed before and after

<table>
<thead>
<tr>
<th>CPU type</th>
<th>Code size (in bytes)</th>
<th>Execution speed (in cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>RX610</td>
<td>13</td>
<td>7</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Source code before</th>
<th>Source code after</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct str {</td>
<td>struct str {</td>
</tr>
<tr>
<td>long L1[8];</td>
<td>char C1;</td>
</tr>
<tr>
<td>char C1;</td>
<td>long L1[8];</td>
</tr>
<tr>
<td>};</td>
<td>};</td>
</tr>
<tr>
<td></td>
<td>struct str STR1;</td>
</tr>
<tr>
<td></td>
<td>char x;</td>
</tr>
<tr>
<td>void func() {</td>
<td>void func() {</td>
</tr>
<tr>
<td></td>
<td>x = STR1.C1;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Expanded assembly code before

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

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

<table>
<thead>
<tr>
<th>CPU type</th>
<th>Code size (in bytes)</th>
<th>Execution speed (in cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>RX610</td>
<td>18</td>
<td>17</td>
</tr>
</tbody>
</table>
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.

```c
/* 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;
```

```c
/* 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;
```

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.

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

```c
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

<table>
<thead>
<tr>
<th>CPU type</th>
<th>Code size (in bytes)</th>
<th>Execution speed (in cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>RX610</td>
<td>25</td>
<td>13</td>
</tr>
</tbody>
</table>

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()
{
```c
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
```
Code size and execution speed before and after

<table>
<thead>
<tr>
<th>CPU type</th>
<th>Code size (in bytes)</th>
<th>Execution speed (in cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>67</td>
</tr>
</tbody>
</table>

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

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

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

Expanded assembly code before

```assembly
_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

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

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

Expanded assembly code after

```assembly
_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

<table>
<thead>
<tr>
<th>CPU type</th>
<th>Code size (in bytes)</th>
<th>Execution speed (in cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>RX610</td>
<td>14</td>
<td>10</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Source code before</th>
<th>Source code after</th>
</tr>
</thead>
<tbody>
<tr>
<td>/* D_1 section */</td>
<td>/* D_1 section */</td>
</tr>
<tr>
<td>char d11=0, d12=0, d13=0, d14=0;</td>
<td>char d11=0, d12=0, d13=0, d14=0;</td>
</tr>
<tr>
<td>/* D_2 section */</td>
<td>/* D_2 section */</td>
</tr>
<tr>
<td>short d21=0, d22=0, d23=0, d24=0, dmy2[12]={0};</td>
<td>short d21=0, d22=0, d23=0, d24=0, dmy2[12]={0};</td>
</tr>
<tr>
<td>/* D section */</td>
<td>/* D section */</td>
</tr>
<tr>
<td>int d41=0, d42=0, d43=0, d44=0, dmy4[60]={0};</td>
<td>int d41=0, d42=0, d43=0, d44=0, dmy4[60]={0};</td>
</tr>
<tr>
<td>void func(int a){</td>
<td>void func(int a){</td>
</tr>
<tr>
<td>d11 = a;</td>
<td>d11 = a;</td>
</tr>
<tr>
<td>d12 = a;</td>
<td>d12 = a;</td>
</tr>
<tr>
<td>d13 = a;</td>
<td>d13 = a;</td>
</tr>
<tr>
<td>d14 = a;</td>
<td>d14 = a;</td>
</tr>
<tr>
<td>d21 = a;</td>
<td>d21 = a;</td>
</tr>
<tr>
<td>d22 = a;</td>
<td>d22 = a;</td>
</tr>
<tr>
<td>d23 = a;</td>
<td>d23 = a;</td>
</tr>
<tr>
<td>d24 = a;</td>
<td>d24 = a;</td>
</tr>
<tr>
<td>d41 = a;</td>
<td>d41 = a;</td>
</tr>
<tr>
<td>d42 = a;</td>
<td>d42 = a;</td>
</tr>
<tr>
<td>d43 = a;</td>
<td>d43 = a;</td>
</tr>
<tr>
<td>d44 = a;</td>
<td>d44 = a;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

Expanded assembly code before

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

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   R1,0108H[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   R1,05H[R4]
  MOV.W   R1,06H[R4]
  MOV.W   R1,07H[R4]
  MOV.L   R1,08H[R4]
  RTS
```

---

**Code size and execution speed before and after**

<table>
<thead>
<tr>
<th>CPU type</th>
<th>Code size (in bytes)</th>
<th>Execution speed (in cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>RX610</td>
<td>43</td>
<td>31</td>
</tr>
</tbody>
</table>
3. Function calls

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

<table>
<thead>
<tr>
<th>Item</th>
<th>Precaution</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function position</td>
<td>• Keep tightly coupled functions together in the same file.</td>
<td>3.1</td>
</tr>
<tr>
<td>Interfaces</td>
<td>• Be strict with regard to the number of arguments (up to 4), so that all arguments can be allocated to the register.</td>
<td>3.2</td>
</tr>
<tr>
<td></td>
<td>• For functions with many arguments, put the arguments in a structure and pass them by pointer.</td>
<td></td>
</tr>
<tr>
<td>Macro</td>
<td>• 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.</td>
<td>--</td>
</tr>
</tbody>
</table>

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

```c
extern void sub(void);

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

Expanded assembly code before

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

Source code after

```c
void sub(void)
{
}

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

Expanded assembly code after

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

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

### 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
```

### Source code after

```c
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 after

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

### Code size and execution speed before and after

<table>
<thead>
<tr>
<th>CPU type</th>
<th>Code size (in bytes)</th>
<th>Execution speed (in cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>RX610</td>
<td>16</td>
<td>8</td>
</tr>
</tbody>
</table>
4. Calculation methods

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

<table>
<thead>
<tr>
<th>Item</th>
<th>Precaution</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reducing loop iterations</td>
<td>Consider merging loop statements with identical or similar loop conditions.</td>
<td>4.1</td>
</tr>
<tr>
<td></td>
<td>• Try loop expansion.</td>
<td></td>
</tr>
<tr>
<td>Using fast algorithms</td>
<td>Consider algorithms that do not require much execution time, such as quicksort for arrays.</td>
<td>0</td>
</tr>
<tr>
<td>Making use of tables</td>
<td>Consider using tables for switch statements in which the processing for each case is nearly identical.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 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.</td>
<td></td>
</tr>
<tr>
<td>Conditional expressions</td>
<td>Comparisons of constants to 0 helps to generate efficient code.</td>
<td>--</td>
</tr>
</tbody>
</table>

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;
    }
} 

Expanded assembly code before
_func:
    MOV.L #00000064H,R4
    MOV.L #_a,R5
    MOV.L #00000000H,R3
    L11:

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 #00000032H,R4
    MOV.L #_a,R5
L11:
## Code size and execution speed before and after

<table>
<thead>
<tr>
<th>CPU type</th>
<th>Code size (in bytes)</th>
<th>Execution speed (in cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>RX610</td>
<td>19</td>
<td>22</td>
</tr>
</tbody>
</table>

L11:  
MOV.L R3, [R5+]
SUB  #01H, R4
BNE  L11

L12:  
RTS

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

L12:  
RTS
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 \texttt{ch} changes depending on the value of variable \texttt{i}.

**Source code before**

```c
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);
}
```

**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
```

**Source code after**

```c
char chbuf[] = { 'a', 'x', 'b' };
char func(int i)
{
    return (chbuf[i]);
}
```

**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

<table>
<thead>
<tr>
<th>CPU type</th>
<th>Code size (in bytes)</th>
<th>Execution speed (in cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>RX610</td>
<td>28</td>
<td>10</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Source code before</th>
<th>Source code after</th>
</tr>
</thead>
<tbody>
<tr>
<td>int func(int a)</td>
<td>int func(int a)</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>if (a==1)</td>
<td>if (a==3)</td>
</tr>
<tr>
<td>a = 2;</td>
<td>a = 8;</td>
</tr>
<tr>
<td>else if (a==2)</td>
<td>else if (a==2)</td>
</tr>
<tr>
<td>a = 4;</td>
<td>a = 4;</td>
</tr>
<tr>
<td>else if (a==3)</td>
<td>else if (a==1)</td>
</tr>
<tr>
<td>a = 8;</td>
<td>a = 2;</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>a = 0;</td>
<td>a = 0;</td>
</tr>
<tr>
<td>return (a);</td>
<td>return (a);</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expanded assembly code before</th>
<th>Expanded assembly code after</th>
</tr>
</thead>
<tbody>
<tr>
<td>_func:</td>
<td>_func:</td>
</tr>
<tr>
<td>CMP #01H,R1</td>
<td>CMP #03H,R1</td>
</tr>
<tr>
<td>BEQ L11</td>
<td>BEQ L11</td>
</tr>
<tr>
<td>L12:</td>
<td>L12:</td>
</tr>
<tr>
<td>CMP #02H,R1</td>
<td>CMP #02H,R1</td>
</tr>
<tr>
<td>BNE L14</td>
<td>BNE L14</td>
</tr>
<tr>
<td>L13:</td>
<td>L13:</td>
</tr>
<tr>
<td>MOV.L #00000004H,R1</td>
<td>MOV.L #00000004H,R1</td>
</tr>
<tr>
<td>RTS</td>
<td>RTS</td>
</tr>
<tr>
<td>L14:</td>
<td>L14:</td>
</tr>
<tr>
<td>CMP #03H,R1</td>
<td>CMP #01H,R1</td>
</tr>
<tr>
<td>BNE L17</td>
<td>BNE L17</td>
</tr>
<tr>
<td>L16:</td>
<td>L16:</td>
</tr>
<tr>
<td>MOV.L #00000008H,R1</td>
<td>MOV.L #00000002H,R1</td>
</tr>
<tr>
<td>RTS</td>
<td>RTS</td>
</tr>
<tr>
<td>L17:</td>
<td>L17:</td>
</tr>
</tbody>
</table>
### Code size and execution speed before and after

<table>
<thead>
<tr>
<th>CPU type</th>
<th>Code size (in bytes)</th>
<th>Execution speed (in cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>RX610</td>
<td>22</td>
<td>22</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Source code before</th>
<th>Source code after</th>
</tr>
</thead>
<tbody>
<tr>
<td>#pragma interrupt int_func</td>
<td>#pragma interrupt int_func(fint)</td>
</tr>
<tr>
<td>volatile int count;</td>
<td>volatile int count;</td>
</tr>
<tr>
<td>void int_func()</td>
<td>void int_func()</td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>count++;</td>
<td>count++;</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expanded assembly code before</th>
<th>Expanded assembly code after</th>
</tr>
</thead>
<tbody>
<tr>
<td>_int_func:</td>
<td>_int_func:</td>
</tr>
<tr>
<td>PUSHM R4-R5</td>
<td>MOV.L #_count,R12</td>
</tr>
<tr>
<td>MOV.L #_count,R4</td>
<td>MOV.L [R12],R13</td>
</tr>
<tr>
<td>MOV.L [R4],R5</td>
<td>ADD #01H,R13</td>
</tr>
<tr>
<td>ADD #01H,R5</td>
<td>MOV.L R13,[R12]</td>
</tr>
<tr>
<td>MOV.L R5,[R4]</td>
<td>RTFI</td>
</tr>
<tr>
<td>POPM R4-R5</td>
<td></td>
</tr>
</tbody>
</table>

■ Code size and execution speed before and after

<table>
<thead>
<tr>
<th>CPU type</th>
<th>Code size (in bytes)</th>
<th>Execution speed (in cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>RX610</td>
<td>18</td>
<td>14</td>
</tr>
</tbody>
</table>

Code size (in bytes) | Execution speed (in cycles)
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

```c
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

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

- Code size and execution speed before and after

<table>
<thead>
<tr>
<th>CPU type</th>
<th>Code size (in bytes)</th>
<th>Execution speed (in cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before</td>
<td>After</td>
</tr>
<tr>
<td>RX610</td>
<td>47</td>
<td>29</td>
</tr>
</tbody>
</table>
Website and Support

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

- Inquiries
  http://www.renesas.com/inquiry

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

<table>
<thead>
<tr>
<th>Rev.</th>
<th>Date</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00</td>
<td>Apr.20.10</td>
<td>—</td>
<td>First edition issued</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Note:** The table above shows the revision history for a document, with 'Rev.' indicating the revision number, 'Date' specifying the date of revision, 'Page' indicating the page number of the changes, and 'Description' providing a brief description of the changes made.
Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.

2. Renesas Electronics does not assume any liability for infringement of 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. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.

3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.

4. 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 of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.

5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may 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.

6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.

7. Renesas Electronics products are classified according to the following three quality grades: “Standard”, “High Quality”, and “Specific”. The recommended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as “Specific” without the prior written consent of Renesas Electronics. Further, you may not make any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as “Specific” or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics.

8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified range.

9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, injury or damage caused by fire in the event of the failure of a Renesas Electronics product, 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, please evaluate the safety of the final products or system manufactured by you.

10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.

12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries. (Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries. (Note 2) “Renesas Electronics products” means any product developed or manufactured by or for Renesas Electronics.