To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

RENESAS

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 use 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. The quality grade of each Renesas Electronics product is “Standard” unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   “Standard”: Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   “High Quality”: Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   “Specific”: Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
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 ranges.
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, and 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.
11. This document may not be 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, 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 product(s)” means any product developed or manufactured by or for Renesas Electronics.

SuperH RISC engine C/C++ Compiler Package

APPLICATION NOTE: [Compiler use guide]

Efficient programming techniques

This document introduces efficient programming techniques for SuperH RISC engine C/C++ Compiler V.9.

Table of contents

1. Summary

The SuperH RISC engine C/C++ compiler has provided various optimizations, but through innovations in programming even better performance can be obtained.

This document describes recommended techniques for efficient program for the user to try.

Criteria for evaluating programs include speed of program execution and program size

The following are rules for efficient program creation.

(1) Rules for improving execution speed
Execution speed is determined by statements which are frequently executed and by complex statements. These should be found, and special efforts should be made to improve them.

(2) Rules for reducing program size
In order to shrink program size, similar processing should be performed using common code, and complex functions should be revised.

The execution speed on production machines may differ depending not only on the code generated by the compiler, but also on the memory architecture, cache hit rate, interrupts, and other factors.

Make sure that you check the results of the techniques given in this document, by executing them on the production machines.

The assembly language expansion code appearing in this document is obtained using the command line

shcΔ (C language file) Δ-code=asmcodeΔ-cpu=sh2

However, the cpu option may differ the assembly language expansion code among the SH-1, SH-2, SH-2E, SH-3, and SH-4. Future improvements in the compiler and other changes may result in changes to assembly language expansion code.

Table 1-1 shows the CPU options used for code size and execution speed. The defaults are used for other options, but some specific options are used for some techniques.

Table 1-1 List of CPU Options

No.	CPU Type	CPU Option
1	SH-2	-cpu=sh2
2	SH-2A	-cpu=sh2a
3	SH-3	-cpu=sh3
4	SH-4A	-cpu=sh4AΔ-fpu=single

The execution speeds given in this document have been determined using the simulator debugger from the compiler package.

For the measuments with SH-2A, SH-3, and SH-4A, cache misss are not considered except for some measurements. The number of external memory access cycle is assumed to be 1.

These measurement results are for reference only.

Figure 1-1 lists Efficient Programming Techniques.

Figure 1-1 List of Efficient Programming Techniques

No.	Function	ROM Efficiency	RAM Efficiency	Execution speed	Referenced Section
1	Local Variable(Data Size)	O	-	O	2.1
2	Global Variable(Signs)	O	-	O	2.2
3	Data Structures	O	-	O	2.3
4	Data Alignment	-	O	-	2.4
5	Initial Values and the Const Type	-	O	-	2.5
6	Local Variables and Global Variables	O	-	O	2.6
7	Referencing Constans	O	-	-	2.7
8	Optimization of Division by Constant	X	-	O	2.8
9	Offset of Member in Structure Declaration	O	-	O	2.9
10	Allocation of Bit Fields	O	-	-	2.10
11	Loop Control Variables	X	-	O	2.11
12	Incorporation of Functions in Modules	O	-	O	3.1
13	Function Interface	-	O	O	3.2
14	Reducing the Number of Loops	X	-	O	4.1
15	Use of Tables	O	-	O	4.2
16	Conditionals	O	-	O	4.3
17	Branching	O	-	O	5

Note. In the table, circles (O) and X's have the following meanings.
O: Effective in enhancing performance
X: May detract from performance

2. Data Specification

Table 2-1 lists data-related matters that should be considered.

Table 2-1 Suggestions for Data Specification

Area	Suggestion	Referenced Sections
Data type specifiers,type modifiers	• If an attempt is made to reduce data sizes, the program size may increase as a result. Data types should be declared according to their use. • Program size may change depending on whether signed or unsigned types are used; care should be taken in selecting data types. • In the case of initialization data the values of which do not change within the program, using the const operator will reduce memory requirements.	2.1 2.2 2.5
Data adjustment	• Data should be allocated such that unused areas do not appear in memory.。	2.4
Definition and referencing of structures	• In some cases, data which is frequently referenced or modified can be incorporated into structures and pointer variables used to reduce program size. • Bit fields can be used to reduce data size.	2.3
Use of internal ROM/RAM	• Since Internal memory is accessed more rapidly than external memory common variables should be stored in internal memory.	-

2.1 Local Variable(Data Size)

Important Points:

When local variables of size four bytes are used, ROM efficiency and speed of execution can be improved in some cases.

Description:

The general-purpose registers in the Renesas Tecnology SuperH RISC engine family are four bytes, and so the basic unit of processing is four bytes.

Hence when there are operations employing one-byte or two-byte local variables, code is added to convert these to four bytes. In some cases, taking four bytes for variables, even when only one or two bytes would suffice, can result in smaller program size and faster execution.

Example of Use:

To calculate the sum of the integers from 1 to 50:

Source code (BEFORE)

```
int f(void)
{
    char a = 50;
    int c = 0;
    for ( ; a > 0; a-- )
        c += a;
    return(c);
}
```

Expanded assembly code (BEFORE)

```
_f:
          MOV         #50,R2      ; H'00000032
          MOV         #0,R6       ; H'00000000
L11:
          ADD         R2,R6
          ADD         #-1,R2
          EXTS.B      R2,R2
          CMP/PL      R2
          BT          L11
          RTS
          MOV         R6,R0
```

Source code (AFTER)

```
int f(void)
{
    long a = 50;
    int c = 0;
    for ( ; a > 0; a-- )
        c += a;
    return(c);
}
```

Expanded assembly code (AFTER)

```
_f:
          MOV         #50,R2      ; H'00000032
          MOV         #0,R6       ; H'00000000
L11:
          ADD         R2,R6
          ADD         #-1,R2
          CMP/PL      R2
          BT          L11
          RTS
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size[byte]		Execution Speed [Cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-2	18	16	353	303
SH-2A	16	14	302	252
SH-3	18	16	353	303
SH-4A	18	16	300	268

2.2 Global Variable(Signs)

Important Points:

When a statement includes a type conversion for a global variable, if it makes no difference whether an integer variable is `signed` or `unsigned`, declaring it as `signed` can improve ROM efficiency and execution speed.

Description:

When the Renesas Tecnology SuperH RISC engine family transfers one or two-byte data from memory using a MOV instruction, an EXTU instruction is added for `unsigned` data. Hence efficiency is poorer for variables declared as `unsigned` types than for `signed` types.

Note that for SH-2A and SH2A-FPU, MOV + EXTU instructions may be substituted for a MOVU instruction. Since a MOVU instruction is a 32-bit instruction, efficiency is poorer for variables declared as `unsigned` types than for `signed` types.

Example of Use:

To substitute at the sum of variable a and variable b for variable c:

Source code (BEFORE)

```
unsigned short a;
unsigned short b;
int c;
void f(void)
{
    c = b + a;
}
```

Expanded assembly code (BEFORE)

```
_f:
          MOV.L       L11,R1
          MOV.L       L11+4,R2
          MOV.W       @R1,R5
          EXTU.W      R5,R4
          MOV.L       L11+8,R5
          MOV.W       @R5,R7
          EXTU.W      R7,R7
          ADD         R7,R4
          RTS
          MOV.L       R4,@R2
L11:
          .DATA.L     _b
          .DATA.L     _c
          .DATA.L     _a
```

Source code (AFTER)

```
short a;
short b;
int c;
void f(void)
{
    c = b + a;
}
```

Expanded assembly code (AFTER)

```
_f:
          MOV.L       L11,R1
          MOV.L       L11+4,R4
          MOV.W       @R1,R5
          MOV.W       @R4,R7
          MOV.L       L11+8,R2
          ADD         R7,R5
          RTS
          MOV.L       R5,@R2
L11:
          .DATA.L     _b
          .DATA.L     _a
          .DATA.L     _c
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size[byte]		Execution Speed [Cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-2	32	28	15	11
SH-2A	32	28	8	8
SH-3	32	28	15	11
SH-4A	32	28	16	10

2.3 Data Structures

Important Points:

When related data is declared as a structure, in some cases execution speed is improved.

Description:

When data is referenced any number of times within the same function, by allocating the base address to a register and creating a data structure, efficiency is improved. Efficiency is also improved when the data is passed as a parameter. Frequently accessed data should be gathered at the beginning of the structure for best results.

When data is structured, it becomes easier to perform tuning such as modification of the data representation.

Example of Use:

To substitute numerical values into the variables a, b, and c:

Source code (BEFORE)

```
int a, b, c;
void f(void)
{
    a = 1;
    b = 2;
    c = 3;
}
```

Source code (AFTER)

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

void f(void)
{
    register struct s *p=&s1;

    p->a = 1;
    p->b = 2;
    p->c = 3;
}
```

Expanded assembly code (BEFORE)

```
_f:
          MOV.L       L11,R7      ; _a
          MOV         #1,R1       ; H'00000001
          MOV.L       R1,@R7      ; a
          MOV.L       L11+4,R1    ; _b
          MOV.L       L11+8,R2    ; _c
          MOV         #2,R4       ; H'00000002
          MOV         #3,R5       ; H'00000003
          MOV.L       R4,@R1      ; b
          RTS
          MOV.L       R5,@R2      ; c
L11:
          .DATA.L     _a
          .DATA.L     _b
          .DATA.L     _c
```

Expanded assembly code (AFTER)

```
_f:
          MOV.L       L11,R2      ; _s1
          MOV         #1,R1       ; H'00000001
          MOV         #2,R4       ; H'00000002
          MOV         #3,R5       ; H'00000003
          MOV.L       R1,@R2      ; (p)->a
          MOV.L       R4,@(4,R2)  ; (p)->b
          RTS
          MOV.L       R5,@(8,R2)  ; (p)->c
L11:
          .DATA.L     _s1
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size[byte]		Execution Speed [Cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-2	32	20	12	9
SH-2A	32	20	9	6
SH-3	32	20	14	10
SH-4A	32	20	10	8

2.4 Data Alignment

Important Points:

In some cases, the amount of RAM required can be reduced by changing the order of data declarations.

Description:

When declaring variables in types of different sizes, variables with the same size type should be declared consecutively. By aligning data in this way, empty areas in the data space are minimized.

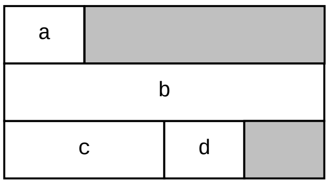
Example of Use:

To declare data totaling eight bytes:

Source code (BEFORE)

```
char a;
int b;
short c;
char d;
```

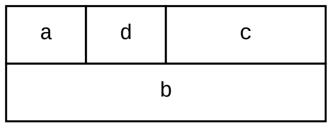
Data arrangement before optimization



Source code (AFTER)

```
char a;
char d;
short c;
int b;
```

Data arrangement after optimization



2.5 Initial Values and the Const Type

Important Points:

Initial values which do not change during program execution should be declared using const.

Description:

Initialization data is normally transferred from ROM to RAM on startup, and the RAM area is used for processing. Hence, if the values of initialization data are not changed within the program, the prepared RAM area is wasted. By using the `const` operator when declaring initialization data, transfer to RAM on startup is prevented, and the amount of memory used is reduced.

In addition, by creating programs which as a rule do not change initial values, it is easy to prepare the program for storage in ROM.

Example of Use:

To specify five pieces of initialization data:

Source code (BEFORE)	Source code (AFTER)
`char a[] = {1, 2, 3, 4, 5};`	`const char a[] = {1, 2, 3, 4, 5};`
Initial value is transferred from ROM to RAM before processing.	Initial value stored in ROM is used for processing.

2.6 Local Variables and Global Variables

Important Points:

If locally-used variables such as temporary variables or loop counters are declared as local variables, execution speed can be improved.

Description:

Variables which can be used as local variables should always be declared as local variables, as global variables. Since the values of global variables may change depending on function calls or pointer operations, they degrade optimization efficiency.

Use of local variables has the following advantages.

a. Low access cost
b. The possibility of register allocation
c. More efficient optimization

Example of Use:

Examples using global variables (BEFORE) and local variables (AFTER) as temporary variables:

Source code (BEFORE)

```
int tmp;

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

Expanded assembly code (BEFORE)

```
_f:
          MOV.L       @R4,R1      ; *(a)
          MOV.L       L11,R6      ; _tmp
          MOV.L       R1,@R6      ; tmp
          MOV.L       @R5,R7      ; *(b)
          MOV.L       R7,@R4      ; *(a)
          MOV.L       @R6,R2      ; tmp
          RTS
          MOV.L       R2,@R5      ; *(b)
L11:
          .DATA.L     _tmp
```

Source code (AFTER)

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

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

Expanded assembly code (AFTER)

```
_f:
          MOV.L       @R4,R6      ; *(a)
          MOV.L       @R5,R2      ; *(b)
          MOV.L       R2,@R4      ; *(a)
          RTS
          MOV.L       R6,@R5      ; *(b)
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size[byte]		Execution Speed [Cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-2	20	10	12	7
SH-2A	20	10	10	6
SH-3	20	10	15	7
SH-4A	20	10	11	7

2.7 Referencing Constans

Important Points:

Code size can be decreased by allowing constant values to be represented in one byte.

Description:

When 2-byte or 4-byte constant values are used, the constant value is reserved in memory as literal data, and code is generated to use a MOV instruction to load the data into the register. On the other hand, when 1-byte constant values are used, the constant data can be embedded within the MOV instruction. This reduces the memory access needed to load literal data, as well as the size of the code needed for the literal data.

Note that for SH-2A and SH2A-FPU, constant values up to 20 bits long can be embedded within code.

The `const_load=inline` option or `speed` option can be specified to expand all 2-byte constants and some 4-byte constants to instructions calculated from 1-byte constant values. Since this increases code size but reduces memory access, it can improve execution speed.

Example of Use:

Source code (BEFORE)

```
#define   CODE  (567)

int data;
void f(void)
{
    data= CODE;
}
```

Expanded assembly code (BEFORE)

```
_f:
          MOV.L       L11+4,R6    ; _data
          MOV.W       L11,R2      ; H'0237
          RTS
          MOV.L       R2,@R6      ; data
L11:
          .DATA.W     H'0237
          .RES.W      1
          .DATA.L     _data
```

Source code (AFTER)

```
#define   CODE  (123)

int data;
void f(void)
{
    data  = CODE;
}
```

Expanded assembly code (AFTER)

```
_f:
          MOV.L       L11,R6      ; _data
          MOV         #123,R2     ; H'0000007B
          RTS
          MOV.L       R2,@R6      ; data
L11:
          .DATA.L     _data
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size[byte]		Execution Speed [Cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-2	14	12	5	5
SH-2A	14	12	4	4
SH-3	14	12	5	5
SH-4A	14	12	6	5

2.8 Optimization of Division by Constant

Important Points:

Optimization of Division by Constant. Therefore, use a division by a constant wherever possible.

Description:

The optimization processing turns a division by a constant into an operation of multiplying by an approximate value of the constant's reciprocal and then fine-tuning the result. This function will drastically improve the execution speed for division compared to using the subroutine calls or the DIVS instruction.

Example of Use:

In the following example of improvement, the use of a constant as the divisor will result in an instruction string that obtains a quotient of 3 directly without calling a division routine. A similar code will be generated also for divisions by other constants:

Source code (BEFORE)

```
int x;
int z=3;
void f (int y){
    x=y/z;
}
```

Expanded assembly code (BEFORE)

```
_f:
          STS.L       PR,@-R15
          MOV.L       L11,R5      ; _z
          MOV.L       L11+4,R2    ; __divls
          MOV.L       @R5,R0      ; z
          MOV.L       L11+8,R6    ; _x
          JSR         @R2
          MOV         R4,R1
          LDS.L       @R15+,PR
          RTS
          MOV.L       R0,@R6      ; x
L11:
          .DATA.L     _z
          .DATA.L     __divls
          .DATA.L     _x
```

Source code (AFTER)

```
int x;
void f (int y){
    x=y/3;
}
```

Expanded assembly code (AFTER)

```
_f:
          STS.L       MACL,@-R15
          STS.L       MACH,@-R15
          MOV.L       L11,R1      ; H'55555556
          MOV.L       L11+4,R5    ; _x
          DMULS.L     R4,R1
          STS         MACH,R6
          MOV         R6,R0
          ROTL        R0
          AND         #1,R0
          ADD         R0,R6
          MOV.L       R6,@R5      ; x
          LDS.L       @R15+,MACH
          RTS
          LDS.L       @R15+,MACL
L11:
          .DATA.L     H'55555556
          .DATA.L     _x
```

Note: This optimization, which can drastically improve the speed, is not applied for optimizations for size because the expanded code may become too large.

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size[byte]		Execution Speed [Cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-2	32	36	74	22
SH-2A	20	36	42	16
SH-3	32	36	76	24
SH-4A	32	36	77	19

Note: y=10000

2.9 Offset of Member in Structure Declaration

Important Points:

Declare a frequently used member of a structure in the beginning of code to improve both the size and speed.

Description:

A program accesses a structure member by adding an offset to the structure address. The smaller the offset, the more advantageous both the size and speed. Therefore, declare a frequently used member in the beginning of code.

It is most effective to declare a member within less then 16 bytes from the beginning for `char` and `unsigned char` types, within less then 32 bytes from the beginning for `short` and `unsigned short` types, and within less then 64 bytes from the beginning for `int`, `unsigned`, `long`, and `unsigned long` types.

Example of Use:

In the following example, the offset of a structure changes the code.

Source code (BEFORE)

```
struct S{
    int a[100];
    int x;
};
int f(struct S *p){
    return p->x;
}
```

Expanded assembly code (BEFORE)

```
_f:
        MOV        #100,R0    ; H'00000064
        SHLL2      R0
        RTS
        MOV.L      @(R0,R4),R0; (p)->x
```

Source code (AFTER)

```
struct S{
    int x;
    int a[100];
};
int f(struct S *p){
    return p->x;
}
```

Expanded assembly code (AFTER)

```
_f:
        RTS
        MOV.L      @R4,R0     ; (p)->x
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size[byte]		Execution Speed [Cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-2	8	4	5	3
SH-2A	6	4	5	5
SH-3	8	4	5	3
SH-4A	8	4	6	5

2.10 Allocation of Bit Fields

Important Points:

The bit fields to be referenced in connection with the same expression should be allocated to the same structure.

Description:

Every time the members in different bit fields are referenced, it is necessary to load data including the bit fields. You can manage to load this data only once by allocating related bit fields to the same structure.

Example of Use:

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

Source code (BEFORE)

```
struct bits{
    unsigned int b0: 1;
} f1, f2;
int f(void){
    if (f1.b0 && f2.b0) return 1;
    else return 0;
}
```

Expanded assembly code (BEFORE)

```
_f:
          MOV.L       L15,R6      ; _f1
          MOV.B       @R6,R0      ; (part of)f1
          TST         #128,R0
          BT          L12
          MOV.L       L15+4,R6    ; _f2
          MOV.B       @R6,R0      ; (part of)f2
          TST         #128,R0
          BF          L13
L12:
          RTS
          MOV         #0,R0       ; H'00000000
L13:
          RTS
          MOV         #1,R0       ; H'00000001
L15:
          .DATA.L     _f1
          .DATA.L     _f2
```

Source code (AFTER)

```
struct bits{
    unsigned int b0: 1;
    unsigned int b1: 1;
} f1;
int f(void){
    if (f1.b0 && f1.b1) return 1;
    else return 0;
}
```

Expanded assembly code (AFTER)

```
_f:
          MOV.L       L11,R1      ; _f1
          MOV         #-64,R2     ; H'FFFFFFC0
          MOV.B       @R1,R0      ; (part of)f1
          EXTU.B      R2,R2
          AND         #192,R0
          CMP/EQ      R2,R0
          RTS
          MOVT        R0
L11:
          .DATA.L     _f1
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size[byte]		Execution Speed [Cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-2	32	20	11	9
SH-2A	32	24	12	12
SH-3	32	20	11	9
SH-4A	32	20	11	11

2.11 Loop Control Variables

Important Points:

Loop control variables can be changed to signed 4-byte integers (`signed int`/`signed long`), to facilitate loop expansion and improve execution speed.

Description:

Even when the `speed` or `loop` option is specified, loop expansion optimization is not performed when the loop control variable is one of the following types:

- unsigned char
- unsigned short
- unsigned long / signed long

Loop control variables of types other than those above are subject to loop expansion optimization, but compared to the `signed char`, `signed short`, `unsigned int`, and `unsigned long` types, loop expansion optimization is more easily performed for the `signed int` and `signed long` types. As such, use the signed 4-byte integer type for loop control variables to perform loop expansion optimization.

Example of Use:

Source code (BEFORE)

```
int ub;
char a[16];

void f2() {
    unsigned char i;

    for(i=0;i<ub;i++) {
        a[i]=0;
    }
}
```

Expanded assembly code (BEFORE)

```
When the loop option is specified
_f2:
          MOV.L       L14+2,R2    ; _ub
          MOV         #0,R6       ; H'00000000
          MOV.L       @R2,R5      ; ub
          BRA         L11
          MOV         R6,R4
L12:
          MOV.L       L14+6,R2    ; _a
          EXTU.B      R6,R0
          MOV.B       R4,@(R0,R2); a[]
          ADD         #1,R0
          MOV         R0,R6
L11:
          EXTU.B      R6,R2
          CMP/GE      R5,R2
          BF          L12
          RTS
          NOP
L14:
          .RES.W      1
          .DATA.L     _ub
          .DATA.L     _a
```

Source code (AFTER)

```
int ub;
char a[16];

void f2() {
    int i;

    for(i=0;i<ub;i++) {
        a[i]=0;
    }
}
```

Expanded assembly code (AFTER)

```
When the loop option is specified
_f2:
          MOV.L       L21+2,R2    ; _ub
          MOV.L       @R2,R4      ; ub
          MOV         R4,R5
          ADD         #-1,R5
          CMP/GT      R5,R4
          BF/S        L12
          MOV         #0,R6       ; H'00000000
          MOV.L       L21+6,R7    ; _a
          MOV         #0,R1       ; H'00000000
          BRA         L13
          MOV         R7,R2
L14:
          MOV         R1,R0
          MOV.B       R1,@R2      ; a[]
          MOV.B       R0,@(1,R2) ; a[]
          ADD         #2,R2
          ADD         #2,R6
L13:
          CMP/GE      R5,R6
          BF          L14
          CMP/GE      R4,R6
          BT          L17
          MOV         R6,R0
          RTS
          MOV.B       R1,@(R0,R7); a[]
L12:
          MOV.L       L21+6,R2    ; _a
          MOV         #0,R1       ; H'00000000
L19:
          CMP/GE      R4,R6
          BT          L17
          MOV.B       R1,@R2      ; a[]
          ADD         #1,R2
          BRA         L19
```

```
                          ADD         #1,R6
L17:
                          RTS
                          NOP
L21:
                          .RES.W      1
                          .DATA.L     _ub
                          .DATA.L     _a
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size[byte]		Execution Speed [Cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-2	38	74	204	104
SH-2A	36	72	155	77
SH-3	38	74	204	120
SH-4A	38	74	142	91

Note: ub=16

3. Function Calls

Matters that should be considered when calling functions are listed in Table 3-1.

Table 3-1 Suggestions Related to Function Calls

Area	Suggestion	Referenced Sections
Function position	• Closely-related functions should be combined in a single file.	3.1
Interface	• The number of parameters should be strictly limited (up to four) such that they are all allocated to registers. • When there are a large number of parameters, they should be incorporated in a structure, and passed using pointers.	3.2
Replacement by macros	• When a function is called frequently, it can be replaced by a macro to speed execution. However, the use of a macro increases program size, and so macros should be used according to the circumstances.	-

3.1 Incorporation of Functions in Modules

Important Points:

Closely-related functions can be combined in a single file to improve program execution speed.

Description:

When functions in different files are called, a JSR instruction is used to expand them; but if functions in the same file are called and the calling range is narrow, a BSR instruction is used, resulting in faster execution and more compact object generation.

Inline expansion can also be performed for function calls within the same file. When the `speed` option or `inline` option is specified, automatic inline expansion is performed, and high-speed object generation is possible (with the program size tending to increase).

By incorporating functions into modules, modifications for tune-up purposes are easier.

Example of Use:

To call the function g from the function f:

Source code (BEFORE)

```
#include <machine.h>
extern g(void);

int f(void)
{
    g();
    nop();
}
```

Source code (AFTER)

```
#include <machine.h>
int g(void)
{
}

int f(void)
{
     g();
     nop();
}
```

Expanded assembly code (BEFORE)

```
_f:
          STS.L       PR,@-R15
          MOV.L       L11,R2      ; _g
          JSR         @R2
          NOP
          NOP
          LDS.L       @R15+,PR
          RTS
          NOP
L11:
          .DATA.L     _g
```

Expanded assembly code (AFTER)

```
_g:
          RTS
          NOP
_f:
          STS.L       PR,@-R15
          BSR         _g
          NOP
          NOP
          LDS.L       @R15+,PR
          RTS
          NOP
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size[byte]		Execution Speed [Cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-2	20	14	15	13
SH-2A	16	12	15	12
SH-3	20	14	16	14
SH-4A	20	14	16	15

Note:

The BSR instruction can call functions within a range of ±4096 bytes (±2048 instructions).

If the file size is too large, the BSR instruction cannot be used effectively.

In such cases, it is recommended that functions which call each other frequently be positioned sufficiently closely so that the BSR instruction can be used.

3.2 Function Interface

Important Points:

By taking care in declaring the parameters of a function, the amount of RAM required can be reduced, and execution speed improved.

For details, see *9.3.2 Function Caling Interface* in the compiler documentation.

Description:

Function parameters should be selected carefully such that all parameters are allocated to registers (up to four parameters). If the structure itself is received, instead of a pointer to the structure, it does not enter the register. If all parameters fit into registers, function calls and processing at function entry and exit points are simplified. Stack use is also reduced.

The registers R0 to R3 are work registers, R4 to R7 are for parameters, and R8 to R14 are for local variables.

With SH-2E, single-precision floating-point numbers are handled in floating-point registers. `FR0` to `FR3` are for work registers, `FR4` to `FR11` are for arguments, and `FR12` to `FR14` are for local variables.
With SH2A-FPU, SH-4, and SH-4A, single-precision/double-precision floating-point numbers can be handled in floating-point registers. When double-precision floating-point numbers are handled, four registers from `DR4` to `DR10` are used for arguments.

Example of Use:

The number of parameters for function f is five, more than the number of parameter registers:

Source code (BEFORE)

```
int f(int, int, int, int, int);

void g(void)
{
    f(1, 2, 3, 4, 5);
}
```

Expanded assembly code (BEFORE)

```
_g:
          STS.L      PR,@-R15
          MOV        #5,R1        ; H'00000005
          MOV.L      R1,@-R15
          MOV.L      L11+2,R2     ; _f
          MOV        #4,R7        ; H'00000004
          MOV        #3,R6        ; H'00000003
          MOV        #2,R5        ; H'00000002
          JSR        @R2
          MOV        #1,R4        ; H'00000001
          ADD        #4,R15
          LDS.L      @R15+,PR
          RTS
          NOP
L11:
          .RES.W     1
          .DATA.L    _f
```

Source code (AFTER)

```
struct b{
    int a, b, c, d, e;
} b1 = {1, 2, 3, 4, 5};

int f(struct b *p);

void g(void)
{
    f(&b1);
}
```

Expanded assembly code (AFTER)

```
_g:
          MOV.L      L11,R4       ; _b1
          MOV.L      L11+4,R2     ; _f
          JMP        @R2
          NOP
L11:
          .DATA.L    _b1
          .DATA.L    _f
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size[byte]		Execution Speed [Cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-2	30	16	20	9
SH-2A	28	16	19	9
SH-3	30	16	22	9
SH-4A	30	16	20	12

4. Operations

Table 5.5 lists areas relating to operations that should be given consideration.

Table 4-1 Suggestions Related to Operations

Area	Suggestion	Referenced Sections
Reduction of number of loop iterations	■ The possibility of merging loop statements with conditions that are identical or similar should be studied. • Try expanding loop statements.	4.1
Use of fast algorithms	■ The use of efficient algorithms requiring little processing time, such as quick sorts of an array, should be studied.	-
Utilization of tables	■ When processing for each case of a switch statement is nearly the same, the use of tables should be studied. ■ Execution speed can sometimes be improved by performing operations in advance, storing the results in a table, and referring to values in the table when the operation results are needed. However, this method requires increased amounts of ROM, and so should be used with due attention paid to the balance between required execution speed and available ROM.	4.2
Conditionals	When making comparisons with a constant, if the value of the constant is 0, more efficient code is generated.	4.3

4.1 Reducing the Number of Loops

Important Points:

When a loop is expanded, execution speed can be improved.

Description:

Loop expansion is especially effective for inner loops. Loop expansion results in an increase in program size, and so this technique should be used only when there is a need to improve execution speed at the expense of larger program size.

Example of Use:

To initialize the array a[]:

Source code (BEFORE)

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

Source code (AFTER)

```
extern int a[100];
void f(void)
{
    int i;

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

Expanded assembly code (BEFORE)

```
_f:
          MOV         #100,R6     ; H'00000064
          MOV.L       L13+2,R2    ; _a
          MOV         #0,R5       ; H'00000000
L11:
          DT          R6
          MOV.L       R5,@R2      ; a[]
          BF/S        L11
          ADD         #4,R2
          RTS
          NOP
L13:
          .RES.W      1
          .DATA.L     _a
```

Expanded assembly code (AFTER)

```
_f:
          MOV         #50,R6      ; H'00000032
          MOV.L       L13,R2      ; _a
          MOV         #0,R5       ; H'00000000
L11:
          DT          R6
          MOV.L       R5,@R2      ; a[]
          MOV.L       R5,@(4,R2)  ; a[]
          BF/S        L11
          ADD         #8,R2
          RTS
          NOP
L13:
          .DATA.L     _a
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size[byte]		Execution Speed [Cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-2	22	24	506	356
SH-2A	20	22	403	253
SH-3	22	24	606	505
SH-4A	22	24	539	268

Note:

When the `loop` option is specified, loop expansion optimization is performed. When the BEFORE source code is compiled with the loop option specified, the same expanded assembly code is output as that for the AFTER source code.

Source code (BEFORE, with loop option specified)

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

Source code (AFTER)

```
extern int a[100];
void f(void)
{
    int i;

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

Expanded assembly code (BEFORE)

```
<-loop>
_f:
          MOV         #50,R6      ; H'00000032
          MOV.L       L13,R2      ; _a
          MOV         #0,R5       ; H'00000000
L11:
          DT          R6
          MOV.L       R5,@R2      ; a[]
          MOV.L       R5,@(4,R2) ; a[]
          BF/S        L11
          ADD         #8,R2
          RTS
          NOP
L13:
          .DATA.L     _a
```

Expanded assembly code (AFTER)

```
_f:
          MOV         #50,R6      ; H'00000032
          MOV.L       L13,R2      ; _a
          MOV         #0,R5       ; H'00000000
L11:
          DT          R6
          MOV.L       R5,@R2      ; a[]
          MOV.L       R5,@(4,R2) ; a[]
          BF/S        L11
          ADD         #8,R2
          RTS
          NOP
L13:
          .DATA.L     _a
```

4.2 Use of Tables

Important Points:

Instead of using a switch statement for branching, tables can be used to improve execution speed.

Description:

When processing by each case of a switch statement is essentially the same, the use of a table should be studied.

Example of Use:

To change the character constant to be substituted into the variable ch according to the value of the variable i:

Source code (BEFORE)

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

Expanded assembly code (BEFORE)

```
_f:
          TST         R4,R4
          BT          L17
          MOV         R4,R0
          CMP/EQ      #1,R0
          BT          L19
          CMP/EQ      #2,R0
          BT          L20
          BRA         L21
          NOP
L17:
          BRA         L21
          MOV         #97,R2      ; H'00000061
L19:
          BRA         L21
          MOV         #120,R2     ; H'00000078
L20:
          MOV         #98,R2      ; H'00000062
L21:
          RTS
          MOV         R2,R0
```

Expanded assembly code (AFTER)

```
_f:
          MOV.L       L11,R6      ; _chbuf
          MOV         R4,R0
          RTS
          MOV.B       @(R0,R6),R0; chbuf[]
L11:
          .DATA.L     _chbuf
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size[byte]		Execution Speed [Cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-2	32	12	13	5
SH-2A	30	12	11	7
SH-3	32	12	13	5
SH-4A	32	12	18	5

Note: i=2

4.3 Conditionals

Important Points:

When making comparisons with a constant, if the value of the constant is 0, more efficient code is generated.

Description:

When making comparisons with zero, an instruction to load the constant value is not generated, and so the length of the code is shorter than in comparisons with constants of value other than 0. Condionals for loops and if statements should be designed such that comparisons are with 0.

Example of Use:

To change the return value according to whether the value of an parameter is 1 or greater:

Source code (BEFORE)

```
int f (int x)
{
    if ( x >= 1 )
        return 1;
    else
        return 0;
}
```

Expanded assembly code (BEFORE)

```
_f:
          MOV         #1,R2       ; H'00000001
          CMP/GE      R2,R4
          RTS
          MOVT        R0
```

Source code (AFTER)

```
int f (int x)
{
    if ( x > 0 )
        return 1;
    else
        return 0;
}
```

Expanded assembly code (AFTER)

```
_f:
          CMP/PL      R4
          RTS
          MOVT        R0
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size[byte]		Execution Speed [Cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-2	8	6	5	4
SH-2A	8	6	6	5
SH-3	8	6	5	4
SH-4A	8	6	6	5

5. Branching

Matters pertaining to branching that should be considered are as follows.

- The same decisions should be combined.
- When switch statements and "else if" statements are long, cases which should be decided quickly and to which branching is frequent should be placed at the beginning.
- When switch and "else if" statements are long, dividing them into stages can speed program execution.

Important Points:

Switch statements with up to five or six cases can be changed to if statements to improve execution speed.

Description:

Switch statements with few cases should be replaced by if statements.

In a switch statement, the range of the variable value is checked before referring to the table of case values, for additional overhead.

On the other hand, if statements involve numerous comparisons, for decreased efficiency as the number of cases involved increases.

The code expansion method for the `switch` statement can be specified by the `case` option. When `case=ifthen` is specified, `switch` statements are expanded using the `if_then` method. When `case=table` is specified, `switch` statements are expanded using the table method. If this option is omitted, the expansion method is automatically selected by the compiler.

Example of Use:

To change the return value according to the value of the variable a:

Source code (BEFORE)

```
int x(int a)
{
    switch (a)
    {
    case 1:
        a = 2; break;
    case 10:
        a = 4; break;
    default:
        a = 0; break;
    }
    return (a);
}
```

Expanded assembly code (BEFORE)

```
_x:
          MOV         R4,R0
          CMP/EQ      #1,R0
          BT          L16
          CMP/EQ      #10,R0
          BT          L17
          BRA         L18
          NOP
L16:
          BRA         L19
          MOV         #2,R2       ; H'00000002
L17:
          BRA         L19
          MOV         #4,R2       ; H'00000004
L18:
          MOV         #0,R2       ; H'00000000
L19:
          RTS
          MOV         R2,R0
```

Source code (AFTER)

```
int x (int a)
{
    if (a==1)
        a = 2;
    else if (a==10)
        a = 4;
    else
        a = 0;
    return (a);
}
```

Expanded assembly code (AFTER)

```
_x:
          MOV         R4,R0
          CMP/EQ      #1,R0
          BF          L12
          BRA         L13
          MOV         #2,R4       ; H'00000002
L12:
          CMP/EQ      #10,R0
          BF/S        L13
          MOV         #0,R4       ; H'00000000
          MOV         #4,R4       ; H'00000004
L13:
          RTS
          MOV         R4,R0
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size[byte]		Execution Speed [Cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-2	28	22	11	9
SH-2A	22	20	8	5
SH-3	28	22	11	9
SH-4A	28	22	20	10

Note: a=1

Website and Support <website and support,ws>

Renesas Technology Website
http://japan.renesas.com/

Inquiries
http://japan.renesas.com/inquiry
csc@renesas.com

Revision Record <revision history,rh>

Rev.	Date	Description	
		Page	Summary
1.00	Jun.1.07	—	First edition issued

Notes regarding these materials

1. This document is provided for reference purposes only so that Renesas customers may select the appropriate Renesas products for their use. Renesas neither makes warranties or representations with respect to the accuracy or completeness of the information contained in this document nor grants any license to any intellectual property rights or any other rights of Renesas or any third party with respect to the information in this document.
2. Renesas shall have no liability for damages or infringement of any intellectual property or other rights arising out of the use of any information in this document, including, but not limited to, product data, diagrams, charts, programs, algorithms, and application circuit examples.
3. You should not use the products or the technology described in this document for the purpose of military applications such as the development of weapons of mass destruction or for the purpose of any other military use. When exporting the products or technology described herein, you should follow the applicable export control laws and regulations, and procedures required by such laws and regulations.
4. All information included in this document such as product data, diagrams, charts, programs, algorithms, and application circuit examples, 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 products listed in this document, please confirm the latest product information with a Renesas sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas such as that disclosed through our website. (http://www.renesas.com)
5. Renesas has used reasonable care in compiling the information included in this document, but Renesas assumes no liability whatsoever for any damages incurred as a result of errors or omissions in the information included in this document.
6. When using or otherwise relying on the information in this document, you should evaluate the information in light of the total system before deciding about the applicability of such information to the intended application. Renesas makes no representations, warranties or guaranties regarding the suitability of its products for any particular application and specifically disclaims any liability arising out of the application and use of the information in this document or Renesas products.
7. With the exception of products specified by Renesas as suitable for automobile applications, Renesas products are not designed, manufactured or tested for applications or otherwise in systems the failure or malfunction of which may cause a direct threat to human life or create a risk of human injury or which require especially high quality and reliability such as safety systems, or equipment or systems for transportation and traffic, healthcare, combustion control, aerospace and aeronautics, nuclear power, or undersea communication transmission. If you are considering the use of our products for such purposes, please contact a Renesas sales office beforehand. Renesas shall have no liability for damages arising out of the uses set forth above.
8. Notwithstanding the preceding paragraph, you should not use Renesas products for the purposes listed below:
   (1) artificial life support devices or systems
   (2) surgical implantations
   (3) healthcare intervention (e.g., excision, administration of medication, etc.)
   (4) any other purposes that pose a direct threat to human life
   Renesas shall have no liability for damages arising out of the uses set forth in the above and purchasers who elect to use Renesas products in any of the foregoing applications shall indemnify and hold harmless Renesas Technology Corp., its affiliated companies and their officers, directors, and employees against any and all damages arising out of such applications.
9. You should use the products described herein within the range specified by Renesas, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas shall have no liability for malfunctions or damages arising out of the use of Renesas products beyond such specified ranges.
10. Although Renesas endeavors to improve the quality and reliability of its products, IC products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Please be sure to implement safety measures to guard against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas 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 applicable measures. Among others, since the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
11. In case Renesas products listed in this document are detached from the products to which the Renesas products are attached or affixed, the risk of accident such as swallowing by infants and small children is very high. You should implement safety measures so that Renesas products may not be easily detached from your products. Renesas shall have no liability for damages arising out of such detachment.
12. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written approval from Renesas.
13. Please contact a Renesas sales office if you have any questions regarding the information contained in this document, Renesas semiconductor products, or if you have any other inquiries.