RL78 Family C Compiler Package (CC-RL)

Application Guide: Programming Techniques

R20AN0529EJ0100
Rev.1.0
Nov. 26, 2018

Introduction

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

Compiler Revision for which Correct Operation has been Confirmed

CC-RL V1.07.00 for the RL78 family

Contents

1. Overview

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

- Options
- Language extensions
- Coding techniques

The results of measurement and assembly code given in this application note were obtained by using V1.07 of the CC-RL compiler. The value assumed for the **-cpu** option was **-cpu=S3**.

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

2. Options

This chapter describes the effects on code size, ROM size, and speed of execution when options for CC-RL are specified. The degrees of the effects depend on the details of the source code.

2.1 Compiler Options

√: Improved, x: Worsened, Δ: Either improved or worsened, —: No effect, ( ): Default

Option	Code Size	ROM Size	Number of Clock Cycles	Remarks
-memory_model	Δ	Δ	Δ	
-far_rom	x	x	x	
-Onothing	x	—	x	
-Osize	√	—	Δ	Optimization will emphasize code size. However, specifying this option may lower efficiency in terms of the speed of execution.
-Ospeed	Δ	—	√	Optimization will emphasize execution performance. However, specifying this option may lower efficiency in terms of code size.
-Ounroll	x	—	√	The effect of specifying this option depends on its parameter.
-Odelete_static_func	(√)	—	—	
-Oinline_level	x	—	√	The effect of specifying this option depends on its parameter.
-Oinline_size	x	—	√	The effect of specifying this option depends on its parameter.
-Opipeline [V1.03 or later]	—	—	(√)	
-Otail_call	(√)	—	(√)	
-Omerge_files	√	—	√	
-Ointermodule	√	—	√	
-Owhole_program	√	—	√	
-Oalias	√	—	√	
-Osame_code [V1.02 or later]	√	—	x	
-goptimize	√	√	—	At the time of linkage, inter-module optimization is applied to files compiled with this option specified. For optimization at the time of linkage, refer to section 2.3, Linkage Options.
-dbl_size=8	x	x	x	
-signed_char	x	—	x	
-signed_bitfield	x	—	x	
-switch	Δ	Δ	Δ	
-merge_string	—	√	—	
-pack	x	√	x	

-stack_protector/ -stack_protector_all [Professional Edition only] [V1.02 or later]	x	—	x	This option generates code for detecting stack smashing at the entry and end of a function. The code for detection may lower efficiency in terms of code size and speed of execution.
-control_flow_integrity [Professional Edition only] [V1.06 or later]	x	x	x	This option selects the generation of code for checking the called functions for control flow integrity. Since extra code and data are generated for this purpose, this may lower the efficiency in terms of code size, speed of execution, and ROM size.
-unaligned_pointer_for_ca78k0r [V1.06 or later]	x	—	x	

2.1.1 -memory_model

This option specifies the type of the memory model when compiling.

When **-memory_model=small**: The default attribute of both variables and functions is **near**.

When **-memory_model=medium**: The default attribute of variables is **near** and that for functions is **far**.

The **far** area is more suitable for handling large areas but using it increases the size of code for function calls. Note that when this option is omitted, either of the following is assumed according to the setting of the **-cpu** option.

- When **-cpu=S1**: -memory_model=small
- When **-cpu=S2** or **-cpu=S3**: -memory_model=medium

C source code

```
int val;
#pragma noinline func1
void func1()
{
  ++val;
}

#pragma noinline func2
void (*func2(void)) ()
{
  return func1;
}

void main(void)
{
  func2()();
}
```

	-memory_model=small	-memory_model=medium
Code size (bytes)	18	20
Number of clock cycles (clock cycles)	30	32

2.1.2 -far_rom

This option sets the default **near**/**far** attribute of ROM data to **far**. The **far** area is more suitable for handling large areas but using it increases the size of code for data accesses. The default **near**/**far** attribute when this option is omitted is determined by the **-memory_model** option setting.

C source code

```
char* ptr;
const char* c_ptr;
void func()
{
  *ptr = *c_ptr;
}
```

	With -far_rom	Without -far_rom
Code size (bytes)	15	9
Number of clock cycles (clock cycles)	19	13

Note: When this option is specified, the pointer size depends on whether the pointer points to **const** data and the C90 and C99 standards may be violated.

```
// -far_rom is specified.
char* ptr;           // The pointer size is 2 bytes.
                     // It points to the char with __near attribute.

const char* c_ptr;   // The pointer size is 4 bytes.
                     // It points to the char with __far attribute.
```

2.1.3 -O<level>

This option specifies the optimization level from among default, size, speed, and nothing. When this option is omitted, the optimization level is **-Odefault**.

default: Default
Performs optimization that is effective in terms of both the object size and execution speed.

size: Optimization with object size precedence
Regards reducing the ROM/RAM capacity as important and performs the maximum optimization that is effective for general programs.

speed: Optimization with execution speed precedence
Regards improving the execution speed as important and performs the maximum optimization that is effective for general programs.

nothing: Optimization with debugging precedence
Regards debugging as important and suppresses all types of optimization including default optimization.

The default values of the items for optimization listed below vary with the selected optimization level.

Optimization through the optimization level setting does not precisely match the optimization result obtained by specifying each optimization item separately. For example, when the **-Odefault** level is specified and then each optimization item is separately set to match the corresponding item value for the **-Osize** level, the optimization result is not equivalent to that obtained when **-Osize** is specified from the beginning.

Optimization Item (item)	Optimization Level (level)			
	-Odefault	-Osize	-Ospeed	-Onothing
-Ounroll	1	1	2	1
-Odelete_static_func	on	on	on	off
-Oinline_level	3	3	2	—
-Oinline_size	0	0	100	—
-Otail_call	on	on	on	off
-Opipeline	on	on	on	off
-Osame_code	off	on	off	off

C source code

```
long a;

void main(void)
{
  unsigned long i = 0;
  unsigned long j = 0;
  for (i = 0; i < 5; ++i) {
    for (j = 0; j < 5; ++j) {
      a += (i + j);
      a *= (i + j);
    }
  }
  for (i = 0; i < 5; ++i) {
    for (j = 0; j < 5; ++j) {
      a += (i + j);
      a *= (i + j);
    }
  }
}
```

	-Odefault	-Osize	-Ospeed	-Onothing
Code size (bytes)	309	279	688	345
Number of clock cycles (clock cycles)	4261	5701	2944	4061

2.1.4 -Ounroll

This option specifies whether loops are to be unrolled.

Loops will be unrolled with the value specified as the parameter as the maximum rate of increase. The operation when 0 is set as the parameter is the same as that when 1 is set as the parameter. Note that if **-Onothing** is specified, the setting of this option is ignored.

Unrolling loop statements accelerates execution while increasing the code size.

C source code

```
int val;

void main(void)
{
  unsigned int i, j, k, l;
  for (i = 1; i < 7; ++i) {
    for (j = 1; j < 6; ++j) {
      for (k = 1; k < 5; ++k) {
        for (l = 1; l < 4; ++l) {
          val += (i + j + k);
          val *= (i + j + k);
        }
      }
    }
  val += (i * 10);
  }
}
```

	-Ounroll=1	**-Ounroll=2**	**-Ounroll=4294967295**
Code size (bytes)	152	214	309
Number of clock cycles (clock cycles)	11154	8670	7968

Note: **-Ounroll=4294967295** is the maximum value of the parameter.

2.1.5 -Odelete_static_func

This option specifies whether unused **static** functions are to be deleted.

If **-Odelete_static_func=on** is specified, optimization is enabled. If **-Odelete_static_func=off** is specified, optimization is disabled.

Deletion of unused **static** functions will decrease the code size.

C source code

```
int val;

static int func()
{
  return 100;
}

void main(void)
{
  val += func();
}
```

	-Odelete_static_func=on	-Odelete_static_func=off
Code size (bytes)	10	14

2.1.6 -Oinline_level

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

The level of expansion varies with the value of the parameter in the following way.

When **-Oinline_level=0**: Suppresses all inline expansion, including those functions for which **#pragma inline** was specified.

When **-Oinline_level=1**: Inline expansion only proceeds for functions for which **#pragma inline** is specified.

When **-Oinline_level=2**: Functions that are the targets of expansion are automatically distinguished and expanded.

When **-Oinline_level=3**: Functions that are the targets of expansion are automatically distinguished and expanded, while minimizing the increase in code size.

However, if 1 to 3 is specified, a function that is specified with **#pragma inline** may not be expanded according to the content of the function and the state of compilation. Note that if **-Onothing** is specified, the setting of this option is ignored.

Inline expansion of functions generally accelerates execution while increasing the code size.

C source code

```
int val, x[1000], y[1000];

static void func1(void)
{
  ++val;
}

#pragma inline func2
void func2(int a)
{
    x[a] = x[a] + a;
}

void func3(int a)
{
  if (a) {
    y[a] = a;
  }
}

void main(void)
{
  int i;
  func2(val);
  func3(val);
  for (i = 0; i < 10; ++i) {
    func1();
  }
  func2(val);
  func3(val);
}
```

	-Oinline_level=0	-Oinline_level=1	-Oinline_level=2	-Oinline_level=3
Code size (bytes)	72	88	100	80
Number of clock cycles (clock cycles)	247	229	150	155

Note: In the case of **-Oinline_level=2**, specify **-Oinline_size=200** (functions will be inline-expanded until the code size is increased by up to 200%).

#pragma inline has the same effect as an **__inline** declaration.

A function for which **#pragma noinline** is specified will not be inline-expanded.

```
#pragma inline in_func1
void in_func1(void)            // Performs inline expansion
{
}

__inline void in_func2(void)   // Performs inline expansion
{
}

#pragma noinline no_func       // Does not perform inline expansion
void no_func(void)
{
}
```

2.1.7 -Oinline_size

This option specifies the allowed increase in the code size due to the use of inline expansion of functions.

If 100 is set as the parameter, functions will be inline-expanded until the code size is increased by up to 100%.

This option is valid when **-Oinline_level=2** is specified.

Inline expansion of functions generally accelerates execution while increasing the code size.

C source code

```
int x[10];

void func1(int a)
{
  x[a] = x[a] * x[a];
}

void func2(int a)
{
  x[a] = x[a] * x[a] * x[a];
}

void func3(int a)
{
  x[a] = x[a] * x[a] * x[a] * x[a] * x[a] * x[a] * x[a] * x[a];
}

void main(void)
{
  int i;
  for (i = 0; i < 10; ++i) {
    func1(i);
    func2(i);
    func3(i);
  }
}
```

	-Oinline_size=0	**-Oinline_size=100**	**-Oinline_size=200**	**-Oinline_level=65535**
Code size (bytes)	107	126	135	165
Number of clock cycles (clock cycles)	988	922	742	612

Note: **-Oinline_level=65535** is the maximum value of the parameter.

2.1.8 -Opipeline [V1.03 or later]

This option specifies whether to enable or disable the optimization of pipeline processing.

If **-Opipeline=on** is specified, optimization is enabled. If **-Opipeline=off** is specified, optimization is disabled. Note that if **-Onothing** is specified, the setting of this option is ignored.

If the optimization of pipeline processing is enabled, the compiler will schedule instructions for more efficient execution and thus increase the speed of execution.

C source code

```
#define N 2
int a[N*N], b[N*N], c[N*N];

void main(void) {
  int i, j, k;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      c[i*N+j] = 0;
      for (k = 0; k < N; k++) {
        int tmp = a[i*N+k] * b[k*N+j];
        c[i*N+j] += ((tmp >> 2) & (~(0xffffffff << 4))) *
                    ((tmp >> 5) & (~(0xffffffff << 7)));
      }
    }
  }
}
```

	-Opipeline=on	-Opipeline=off
Code size (bytes)	197	197
Number of clock cycles (clock cycles)	279	283

2.1.9 -Otail_call

This option specifies whether a function call at the end of a function is to be replaced with a **br** instruction.

If **-Otail_call=on** is specified, optimization is enabled. If **-Otail_call=off** is specified, optimization is disabled.

If there is a function call at the end of a function and certain conditions are met, a **br** instruction will be generated for that call rather than a **call** instruction and the **ret** instruction will be removed, reducing the code size and increasing the speed of execution. However, some debugging functions cannot be used.

C source code

```
int a, b;

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

void main(void)
{
  a += b;
  func();
}
```

	-Otail_call=on	-Otail_call=off
Code size (bytes)	15	17
Number of clock cycles (clock cycles)	14	20

2.1.10 -Omerge_files

This option enables merging of multiple files before compilation.

When this option is specified, the compiler compiles multiple C source files and output the results to a single file. When this option is not specified, the C source files are not merged and each of the C source files is compiled separately.

C source code [tp1.c]

```
extern long func(long x, long y, long z);
long result;

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

C source code [tp2.c]

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

	With -Omerge_files	Without -Omerge_files
Code size (bytes)	196	217
Number of clock cycles (clock cycles)	10	55

Note: The above values are for when **-Ointermodule is** specified. The code size includes the size of the startup routine.

2.1.11 -Ointermodule

This option is used to enable global optimization.

Global optimization is mainly optimization in which inter-procedural alias analysis and the propagation of constant parameters and return values are utilized.

C source code

```
static __near int func(int x, int y, int z) {
  return z - x - y;
}

int func2(void) {
  return func(3, 4, 8);
}
```

	With -Ointermodule	Without -Ointermodule
Code size (bytes)	14	17
Number of clock cycles (clock cycles)	25	28

2.1.12 -Owhole_program

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

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

- Files outside the scope of compilation at this time will neither modify nor refer to the values and addresses of **extern** variables defined in the target source files.
- Files outside the scope of compilation at this time will not call functions within the target source files when calls of functions in files outside the scope of compilation by target source files are allowed.

If this option is specified, it is assumed that the **-Ointermodule** option is specified. If two or more C source files are input, it is assumed that the **-Omerge_files** option is specified.

C source code [tp1.c]

```
extern const int c;
extern int func(void);
int result;

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

C source code [tp2.c]

```
#pragma inline (func)
const int c = 1;
int x = 10;
int *p;

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

	With -Owhole_program	Without -Owhole_program
Code size (bytes)	205	192
Number of clock cycles (clock cycles)	194	206

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

2.1.13 -Oalias

This option is used to enable optimization in consideration of the types of data indicated by pointers.

Specifying this option improves code efficiency in terms of code size and speed of execution. However, the results of conversion may differ from the expected values if the C source code does not comply with the ISO/IEC 9899 standard.

ansi or **noansi** can be specified as the parameter. When **ansi** is specified, optimization in consideration of the types of data indicated by pointers proceeds in accord with ISO/IEC 9899. When **noansi** is specified, the types of data indicated by pointers are not considered in terms of ISO/IEC 9899.

The performance of object code is generally better when **ansi** is specified than when **noansi** is specified, but the results of execution may differ.

C source code

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

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

	-Oalias=ansi	-Oalias=noansi
Code size (bytes)	26	40
Number of clock cycles (clock cycles)	19	28

2.1.14 -Osame_code [V1.02 or later]

This option is used to enable optimization which integrates multiple instruction sequences that are found to be the same within single sections of the unit of compilation into functions.

If **-Osame_code=on** is specified, optimization is enabled. If **-Osame_code=off** is specified, optimization is disabled. Note that if **-Onothing** is specified, the setting of this option is ignored.

Optimization by this option increases the number of function calls, thus decreasing the speed of execution but reducing the code size.

C source code

```
volatile int value = 0;
int v1;
int v2;
int v3;
int v4;
int v5;

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

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

	-Osame_code=on	-Osame_code=off
Code size (bytes)	55	93
Number of clock cycles (clock cycles)	57	39

2.1.15 -dbl_size

This option is used to change the interpretation of variables of the **double** and **long double** types.

4 or 8 (only when **-cpu=S3** is specified) can be specified as the parameter.

When 4 is specified as the parameter, both the **double** type and **long double** type are handled as the **float** type. When 8 is specified as the parameter, neither the **double** type nor the **long double** type is handled as the **float** type.

Note that when this option is not specified, both the **double** type and **long double** type are handled as the **float** type (operation becomes the same as when 4 is specified as the parameter).

C source code

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

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

	-dbl_size=4	**-dbl_size=8**
Code size (bytes)	66	121
Number of clock cycles (clock cycles)	132	257

2.1.16 -signed_char

This option specifies that a **char** type without a **signed** or **unsigned** specifier is handled as a signed type.

When this option is not specified, a **char** type without a **signed** or **unsigned** specifier is handled as an unsigned type.

C source code

```
int func(char c)
{
  if (c > 10) {
    c++;
  }
  return c;
}
```

	With -signed_char	Without -signed_char
Code size (bytes)	14	8
Number of clock cycles (clock cycles)	10	4

2.1.17 -signed_bitfield

This option specifies that a bit field of a type without a **signed** or **unsigned** specifier is handled as a signed type.

When this option is not specified, a bit field of a type without a **signed** or **unsigned** specifier is handled as an unsigned type.

C source code

```
typedef struct T {
  int b1:3;
  int b2:3;
  int b3:16;
}STB;
STB stb;
int i;
unsigned int u;

void func1(STB c)
{
  i = c.b1;
  u = c.b2;
}
```

	With -signed_bitfield	Without -signed_bitfield
Code size (bytes)	22	19
Number of clock cycles (clock cycles)	11	9

2.1.18 -switch

This option specifies the format in which the code of **switch** statements will be output.

When **-switch=ifelse** is specified, the code will be output by using the **if_then** method. This is effective when there are not so many **case** labels.

When **-switch=binary** is specified, the code will be output in the binary search format. If this item is selected when many **case** labels are used, branching to any **case** label will be at almost the same speed. This is effective if you wish to reduce the data size when there are many **case** labels.

When **-switch=abs_table** is specified, the code that is output will include branch tables for the **case** labels in the **switch** statements. The absolute addresses of each of the **case** label locations are stored in the tables. The more **case** labels the statements have, the larger the ROM size will become. However, the speed of execution remains the same regardless of the number of labels. This is effective when a larger data size is not a problem.

When **-switch=rel_table** is specified, the code will be output by using the **case** branch table in the **switch** statement. The relative distances from a branch instruction to each of the **case** label locations are stored in a branch table. Though the ROM size is smaller than that for **-switch=abs_table**, a linkage error will occur if a relative distance exceeds 64 Kbytes. This is effective if the relative distance from a table branch instruction to a **case** label location does not exceed 64 Kbytes but there are many **case** labels.

When this option is not specified, the compiler automatically selects the optimum output format for each **switch** statement.

C source code

```
long val = 0;
void func(int val1)
{
  switch (val1) {
  case 21:
    val += 10;
    break;
  case 22:
    val *= 10;
    break;
  case 23:
    val /= 4;
    break;
  case 24:
    val -= 12;
    break;
  default:
    val = -1;
    break;
  }
}

void main()
{
  int i = 20;
  while (i < 25) {
    func(i);
    ++i;
  }
}
```

	-switch			
	ifelse	**binary**	**abs_table**	**rel_table**
Code size (bytes)	120	130	121	126
ROM size (bytes)	0	0	12	8
Number of clock cycles (clock cycles)	189	205	225	217

Note: In the above C source code example, the **if_then** method will be selected if **-switch** is not specified.

2.1.19 -merge_string

If a string constant with the same value occurs more than once in the source file, this option collectively allocates them to a single area, thus reducing the ROM size.

C source code

```
#include <string.h>
long val = 0;
char *a = "abcde";
char *b = "abcde";
void func(void)
{
  if (strcmp(a, b) == 0) {
    val = 1;
  }
}
```

	With -merge_string	Without -merge_string
ROM size (bytes)	6	12

2.1.20 -pack

This option performs packing of a structure (sets 1 as the boundary alignment value for a structure member).

When this option is specified, members of a structure are not aligned according to its type, but code is generated with them packed to be aligned at a 1-byte boundary. Therefore, though the ROM size decreases, specifying this option may lower efficiency in terms of code size and speed of execution.

Correct operation is not guaranteed if a structure, union, or address of those members whose alignment condition has been changed from two bytes to one byte by this option is passed as an argument of a standard library function.

Correct operation is not guaranteed if the address of a structure or union member whose alignment condition has been changed from two bytes to one byte by this option is passed to a pointer whose type has two bytes as the alignment condition and indirect reference to the pointer is performed.

C source code

```
struct{
  signed char  a;
  signed long  b;
  struct{
    signed char c;
    signed long d;
  }f;
}data, *stp;

void func()
{
  data.a = 1;
  data.b = 2;
  data.f.c = 5;
  data.f.d = 6;
  if (stp->b != stp->f.d) {
    data.b++;
  }
}
```

	With -pack	Without -pack
ROM size (bytes)	12	14
Code size (bytes)	97	65
Number of clock cycles (clock cycles)	47	37

The same effect can also be obtained by a #pragma directive. When this option and a **#pragma** directive are specified at the same time, the **#pragma** directive will take priority.

```
struct s1 {
  char a;
  long b;       // When -pack is specified, the boundary alignment value
                // is 1.
                // When -pack is not specified, the boundary alignment
                // value is 2.
} data1;

#pragma pack
struct s2 {
  char a;
  long b;       // The boundary alignment value is always 1 regardless of
                // the option setting.
} data2;

#pragma unpack
struct s3 {
  char a;
  long b;       // The boundary alignment value is always 2 regardless of
                // the option setting.
} data3;
```

2.1.21 -stack_protector/-stack_protector_all [Professional Edition only] [V1.02 or later]

This option generates code for detecting stack smashing at the entry and end of a function.

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

C source code

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

void f1()                   // Sample program in which the stack is smashed
{
  volatile char str[10];
  int i;
  for (i = 0; i <= 9; i++) {
    str[i] = i;             // The stack is smashed when i reaches 10.
  }
}

void __stack_chk_fail(void)
{
  printf("stack is broken!");
}

void main()
{
  f1();
}
```

	-stack_protector	-stack_protector_all	Without -stack_protector/ -stack_protector_all
Code size (bytes)	50	62	38
Number of clock cycles (clock cycles)	1075	1080	1073

The same effect can also be obtained by a #pragma directive. When this option and a **#pragma** directive are specified at the same time, the **#pragma** directive will take priority.

Example: With **-stack_protector** or **-stack_protector_all**

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

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

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

Example: Without **-stack_protector** or **-stack_protector_all**

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

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

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

2.1.22 -control_flow_integrity [Professional Edition only] [V1.06 or later]

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

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

C source code

```
#include <stdlib.h>
int glb;
void __control_flow_chk_fail(void)
{
  abort();
}
void func1(void) // Added to the function list
{
  ++glb;
}
void func2(void) // Not added to the function list
{
  --glb;
}
void (*pf)(void) = func1;
void main(void)
{
  pf(); // Indirect call of the function func1
  func2();
}
```

	With -control_flow_integrity	Without -control_flow_integrity
Code size (bytes)	72	60
ROM size (bytes)	59	0
Number of clock cycles (clock cycles)	91	26

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

2.1.23 -unaligned_pointer_for_ca78k0r [V1.06 or later]

Indirect references by pointers are accessed in 1-byte units. The purpose of this option is to support the porting of code written for the CA78K0R compiler.

Specifying this option increases the size of the object code and decreases the speed of execution.

C source code

```
#include <stdlib.h>

char c;
int *glbp = &c;

int func1(void)
{
  return *glbp;
}

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

	With -unaligned_pointer_for_ca78k0r	Without -unaligned_pointer_for_ca78k0r
Code size (bytes)	8	5
Number of clock cycles (clock cycles)	5	3

2.2 Assembler Option

√: Improved, x: Worsened, Δ: Either improved or worsened, —: No effect

Option	Code Size	ROM Size	Number of Clock Cycles	Remarks
-goptimize	√	√	—	At the time of linkage, inter-module optimization is applied to files compiled with this option specified. For optimization at the time of linkage, refer to 2.3, Linkage Options.

2.3 Linkage Options

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

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

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

√: Improved, x: Worsened, Δ: Either improved or worsened, —: No effect

Option	Code Size	ROM Size	Number of Clock Cycles	Remarks
-optimize=symbol_delete	√	√	—	
-optimize=branch	√	—	—	

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

2.3.1 -optimize=symbol_delete

Variables or functions to which nothing refers are deleted. Specify the **entry** symbol with **-entry** in the linkage editor.

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

C source code

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

void func1(void)
{
  value1++;
}

void func2(void)
{
  value2++;
}

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

	With -optimize=symbol_delete	Without -optimize=symbol_delete
Code size (bytes)	156	162
ROM size (bytes)	144	146

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

2.3.2 -optimize=branch

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

C source code

```
extern void sub(void);

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

	With -optimize=branch	Without -optimize=branch
Code size (bytes)	168	176

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

3. Language Extensions

This chapter describes the effects on code size, ROM size, and speed of execution by the language extensions.

3.1 Reserved Words

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

Reserved Word	Code Size	ROM Size	Number of Clock Cycles	Remarks
__saddr	√	—	√	
__callt	√	x	x	
__near __far	Δ	Δ	Δ	Whether functions or variables are allocated to the **near** area or **far** area affects the code size, ROM size, and speed of execution. Allocating frequently called functions and frequently used variables to the **near** area improves efficiency in addressing.

3.1.1 __saddr

External variables declared with **__saddr** are allocated to the **saddr** area.

Initialized variables are allocated to the **.sdata** section.

Uninitialized variables are allocated to the **.sbss** section.

The performance of object code can be improved by allocating frequently used external variables and static variables within functions to the **saddr** area. For a one-bit field especially, allocation to the **saddr** area can be expected to have a large effect.

Without __saddr

C source code

```
typedef struct {
  unsigned char b0:1;
  unsigned char b1:1;
  unsigned char b2:1;
  unsigned char b3:1;
  unsigned char b4:1;
  unsigned char b5:1;
  unsigned char b6:1;
  unsigned char b7:1;
} BITF;
BITF data0, data1;

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

Assembly-language expanded code

```
_func:
    .STACK _func = 4
    movw hl, #LOWW(_data1)
    mov1 CY, [hl].1
    movw hl, #LOWW(_data0)
    mov1 [hl].4, CY
    ret
```

Code size: 11 bytes
Number of clock cycles: 13

With __saddr

C source code

```
typedef struct {
  unsigned char b0:1;
  unsigned char b1:1;
  unsigned char b2:1;
  unsigned char b3:1;
  unsigned char b4:1;
  unsigned char b5:1;
  unsigned char b6:1;
  unsigned char b7:1;
} BITF;
__saddr BITF data0, data1;

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

Assembly-language expanded code

```
_func:
    .STACK _func = 4
    mov1 CY, _data1.1
    mov1 _data0.4, CY
    ret
```

Code size: 7 bytes
Number of clock cycles: 9

The same effect can also be obtained by **#pragma saddr**. The **#pragma** directive takes priority over the **__near** or **__far** specification.

```
#pragma saddr value
int __far value;        // Allocated to the saddr area
```

3.1.2 __callt

A function declared with **__callt** (**callt** function) is called by the **callt** instruction. The specification for the **callt** function becomes **__near**, and address reference always returns a **near** pointer.

The addresses of the functions to be called are stored in the callt table area [80H-BFH], and the functions are called with a smaller-size code than that for a normal call instruction (**call** instruction).

A table of addresses for function calls is generated and this leads to increased ROM size for the code. The speed of execution will decrease because the **callt** instruction requires more clock cycles for execution than the **call** instruction.

Without __callt	With __callt

C source code
 	
`#pragma noinline sub`
`void sub(void)`
`{`
`}`
 	
`void func(void)`
`{`
`  sub();`
`  sub();`
`  sub();`
`  sub();`
`  sub();`
`}`
	C source code
	
	`#pragma noinline sub`
	`__callt void sub(void)`
	`{`
	`}`
	
	`void func(void)`
	`{`
	`  sub();`
	`  sub();`
	`  sub();`
	`  sub();`
	`  sub();`
	`}`

Assembly-language expanded code
 	
`_func:`
`  .STACK _func = 4`
`  call $!_sub`
`  call $!_sub`
`  call $!_sub`
`  call $!_sub`
`  br $_sub`
	Assembly-language expanded code
	
	`_func:`
	`  .STACK _func = 4`
	`  callt [@_sub]`
	`  callt [@_sub]`
	`  callt [@_sub]`
	`  callt [@_sub]`
	`  br !_sub`
	
	`  .SECTION .callt0,CALLT0  (*)`
	`@_sub:`
	`  .DB2 _sub`

Code size: 14 bytes Number of clock cycles: 45	Code size: 11 bytes Number of clock cycles: 53 * The ROM size is increased (+ 2 bytes) by code for generating a table of addresses for functions.

The same effect can also be obtained by **#pragma callt**.

```
#pragma callt sub
void sub(void)          // callt function
{
}
```

3.1.3 __near/__far

The region for the allocation of a function or variable can be explicitly specified by adding the **__near** or **__far** type qualifier when the function or variable is declared. The **far** area is much larger than the **near** area but the sizes of code for function calls and code for data access are also larger. The code size can be decreased by allocating frequently called functions and frequently used variables to the **near** area.

Code Allocated to the far Area	Code Allocated to the near Area

C source code
 	
`#pragma noinline sub`
`void __far sub(void)`
`{`
`}`
 	
`int __far value;`
`void func(void)`
`{`
`  sub();`
`  value += 10;`
`}`
	C source code
	
	`#pragma noinline sub`
	`void __near sub(void)`
	`{`
	`}`
	
	`int __near value;`
	`void func(void)`
	`{`
	`  sub();`
	`  value += 10;`
	`}`

Assembly-language expanded code
 	
`_func:`
`  .STACK _func = 4`
`  call $!_sub`
`  mov es, #LOW(HIGHW(_value))`
`  movw ax, #0x000A`
`  addw ax, es:!LOWW(_value)`
`  movw es:!LOWW(_value), ax`
`  ret`
	Assembly-language expanded code
	
	`_func:`
	`  .STACK _func = 4`
	`  call !_sub`
	`  movw ax, #0x000A`
	`  addw ax, !LOWW(_value)`
	`  movw !LOWW(_value), ax`
	`  ret`

Code size: 17 bytes Number of clock cycles: 21	Code size: 13 bytes Number of clock cycles: 18

The location to allocate a function can also be specified by **#pragma near** or **#pragma far**. (V1.05 or later)

The **#pragma** directive takes priority over the **__near**, **__far**, or **__callt** specification.

```
#pragma near func1
void __far func1(void)    // Allocated to the near area
{
}
```

A declaration without the **__near** or **__far** type qualifier follows the default **near**/**far** attribute determined by the memory model. The following table shows how to determine the **near**/**far** attribute.

	Item	How to Determine near/far Attribute
(a)	-cpu	This option determines the default memory model.
(b)	-memory_model	This option overwrites the memory model determined in (a).
(c)	-far_rom	This option overwrites the determined attribute with the **far** attribute only for ROM data.
(d)	__near/__far	These settings are not affected by (a) to (c); the qualifier specification is valid unless (e) is also specified.
(e)	#pragma near #pragma far	These settings are not affected by (a) to (d); the **#pragma** directive is always valid.

3.2 #pragma Directives

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

#pragma Directive	Code Size	ROM Size	Number of Clock Cycles	Remarks
#pragma interrupt #pragma interrupt_brk	Δ	Δ	Δ	Changing interrupt specifications used with this directive can improve the performance of interrupt functions.

3.2.1 #pragma interrupt/interrupt_brk

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

Interrupt Specifications	Format	Description
Register bank	bank	Changing the register bank eliminates the need to save the values of general registers on the stack. The values of the ES and CS registers, on the other hand, still need to be saved on the stack even if the register bank is changed.
Nested interrupt enable	enable	Generates EI at the entries of functions to allow the nesting of interrupts.

C source code

```
#pragma interrupt_brk func
void func(void)
{
  sub(1, 2, 3);
}
```

The following table shows the results for comparison when particular interrupt specifications are made and not made for the C source code above.

	None	bank=RB0	enable
Code size (bytes)	32	26	35
Number of clock cycles (clock cycles)	35	28	39

4. Using a Variables/Functions Information File

A variables/functions information file is generated when the **-vfinfo** option of the linker is specified.

The variables/functions information file is in the text format and contains the declarations of those variables and functions among the variables and functions defined in the C source file to which reference is frequent as **saddr** and **callt,** respectively. Including such a file will decrease the code size.

[How to use]

First, specify the **-vfinfo** option of the linker to select the generation of a variables/functions information file.

Then, include the variables/functions information file at compilation through either of the following methods.

- Specify the file with the **-preinclude** option of the compiler.
- Use **#include** to include the file in each of the C source files.

C source code

```
int value1;
int value2;

void func1(void)
{
  value1 += 100;
}

void func2(void)
{
  value1 += 100;
}

void sub(void)
{
  func1();
  func1();
  func1();
  func1();
  func1();
}

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

Output example of the variables/functions information file

```
/* RENESAS OPTIMIZING LINKER GENERATED FILE xxxx.xx.xx */
/*** variable information ***/
#pragma saddr value1 /* count:4,size:2,near,VFINFO.obj */
/* #pragma saddr value2 */ /* count:0,size:2,near,unref,VFINFO.obj */ ※

/*** function information ***/
#pragma callt func1 /* count:5,far,VFINFO.obj */
#pragma callt main /* count:1,far,VFINFO.obj */
#pragma callt sub /* count:1,far,VFINFO.obj */
/* #pragma callt func2 */ /* count:0,far,unref,VFINFO.obj */          ※
```

Note: Variable **value2** and function **func2** will be commented out in the output file because there is no reference to either (despite the directives, **value2** does not become a **saddr** variable and **func2** does not become a **callt** function).

Treating function calls as **callt** functions will decrease the code size but slow the speed of execution. Replacing a function with a **near** function is better when the speed of execution is also to be given priority. Specifying **-vfinfo(near)** selects the output of a function as a **near** function.

	Using the Variables/Functions Information File		Not Using the Variables/Functions Information File
	-vfinfo(near)	**-vfinfo**	
Code size (bytes)	187	182	191
ROM size (bytes)	142	148	142
Number of clock cycles (clock cycles)	63	73	63

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

5. Coding Techniques

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

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

Item	Code Size	ROM Size	Number of Clock Cycles	Remarks
Variables and the **const** qualifier	—	√	—	
Local variables and global variables	√	√	√	
Allocating bit fields	√	—	√	
Function interfaces	√	—	√	
Reducing the number of loops	x	—	√	
Using tables	√	x	√	
Branches	—	—	√	
Inline expansion	Δ	—	√	
Moving identical expressions in more than one conditional branch destination before the conditional branch	√	—	√	
Replacing a sequence of complicated **if** statement with a simple statement having the same logical meaning	√	—	√	
Types of variables	√	Δ	√	
Unifying common **case** processing in **switch** statements	√	—	√	
Replacing **for** loops with **do-while** loops	√	—	√	
Replacing division by powers of two with shift operations	√	—	√	
Changing bit fields with two or more bits to the **char** type	√	x	√	
Alignment of a structure	—	√	—	

5.1 Variables and the const Qualifier

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

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

Creating programs based on the rule of not using initialized variables (**.data**) where this is possible will facilitate the creation of ROM images.

Not const-Qualified	**const-Qualified**

C source code
 	
`char a[] =`
`  {1, 2, 3, 4, 5};`
	C source code
	
	`const char a[] =`
	`  {1, 2, 3, 4, 5};`

ROM size: 5 bytes RAM size: 5 bytes	ROM size: 5 bytes RAM size: 0 bytes

5.2 Local Variables and Global Variables

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

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

Before Using a Local Variable	After Using a Local Variable

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

Assembly-language expanded code
 	
`_func:`
`  .STACK _func = 6`
`  movw de, ax`
`  push bc`
`  pop hl`
`  movw ax, [de]`
`  movw !LOWW(_tmp), ax`
`  movw ax, [hl]`
`  movw [de], ax`
`  movw ax, !LOWW(_tmp)`
`  movw [hl], ax`
`  ret`
	Assembly-language expanded code
	
	`_func:`
	`  .STACK _func = 6`
	`  movw de, ax`
	`  push bc`
	`  pop hl`
	`  movw ax, [de]`
	`  movw bc, ax`
	`  movw ax, [hl]`
	`  movw [de], ax`
	`  movw ax, bc`
	`  movw [hl], ax`
	`  ret`

Code size: 14 bytes Number of clock cycles: 15	Code size: 10 bytes Number of clock cycles: 15

5.3 Allocating Bit Fields

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

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

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

Before Allocating Bit Fields to the Same Structure	After Allocating Bit Fields to the Same Structure

C source code
 	
`struct str`
`{`
`  int flag1:1;`
`} b1, b2, b3;`
 	
`void func(void)`
`{`
`  b1.flag1 = 1;`
`  b2.flag1 = 1;`
`  b3.flag1 = 1;`
`}`
	C source code
	
	`struct str`
	`{`
	`  int flag1:1;`
	`  int flag2:1;`
	`  int flag3:1;`
	`} a1;`
	
	`void func(void)`
	`{`
	`  a1.flag1 = 1;`
	`  a1.flag2 = 1;`
	`  a1.flag3 = 1;`
	`}`

Assembly-language expanded code
 	
`_func:`
`  .STACK _func = 4`
`  set1 !LOWW(_b1).0`
`  set1 !LOWW(_b2).0`
`  set1 !LOWW(_b3).0`
`  ret`
	Assembly-language expanded code
	
	`_func:`
	`  .STACK _func = 4`
	`  mov a, #0x07`
	`  or a, !LOWW(_a1)`
	`  mov !LOWW(_a1), a`
	`  ret`

Code size: 13 bytes Number of clock cycles: 15	Code size: 9 bytes Number of clock cycles: 12

5.4 Function Interfaces

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

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

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

<table>
<tr><th>No Arguments in a Structure</th><th>Arguments in a Structure</th></tr>
<tr><td>C source code

<pre>
struct str
{
  char a;
  char b;
  char c;
  char d;
  char e;
  char f;
  char g;
  char h;
} arg;

void func(char a, char b, char c,
          char d, char e, char f,
          char g, char h){}

void call_func(void)
{
  func(arg.a, arg.b, arg.c, arg.d,
       arg.e, arg.f, arg.g, arg.h);
}
</pre></td><td>C source code

<pre>
struct str
{
  char a;
  char b;
  char c;
  char d;
  char e;
  char f;
  char g;
  char h;
} arg;

void func(struct str* str_arg){}

void call_func(void)
{
  func(&arg);
}
</pre></td></tr>
<tr><td>Assembly-language expanded code

<pre>
_call_func:
    .STACK _call_func = 8
    mov a, !LOWW(_arg+0x00007)
    shrw ax, 8+0x00000
    push ax
    mov a, !LOWW(_arg+0x00006)
    shrw ax, 8+0x00000
    push ax
    mov a, !LOWW(_arg+0x00005)
    mov d, a
    mov a, !LOWW(_arg+0x00004)
    mov e, a
    mov b, !LOWW(_arg+0x00003)
    mov c, !LOWW(_arg+0x00002)
    mov x, !LOWW(_arg+0x00001)
    mov a, !LOWW(_arg)
    call $!_func
    addw sp, #0x04
    ret
</pre></td><td>Assembly-language expanded code

<pre>
_call_func:
    .STACK _call_func = 4
    movw ax, #LOWW(_arg)
    br $_func
</pre></td></tr>
<tr><td>Code size: 38 bytes
Number of clock cycles: 31</td><td>Code size: 5 bytes
Number of clock cycles: 10</td></tr>
</table>

5.5 Reducing the Number of Loops

Unrolling loops will considerably improve the speed of execution.

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

<table>
<tr><th>Before Unrolling</th><th>After Unrolling</th></tr>
<tr><td>C source code

<pre>
int a[100];

void func(void)
{
  int i;
  for (i = 0; i < 100; i++) {
    a[i] = 0;
  }
}
</pre></td><td>C source code

<pre>
int a[100];

void func(void)
{
  int i;
  for (i = 0; i < 100; i += 2) {
    a[i] = 0;
    a[i+1] = 0;
  }
}
</pre></td></tr>
<tr><td>Assembly-language expanded code

<pre>
_func:
    .STACK _func = 4
    movw de, #LOWW(_a)
    movw bc, #0x0064
.BB@LABEL@1_1: ; bb
    clrw ax
    movw [de], ax
    movw ax, bc
    addw ax, #0xFFFF
    movw bc, ax
    incw de
    incw de
    bnz $.BB@LABEL@1_1
.BB@LABEL@1_2: ; return
    ret
</pre></td><td>Assembly-language expanded code

<pre>
_func:
    .STACK _func = 6
    push hl
    movw ax, #LOWW(_a)
    movw [sp+0x00], ax
    movw hl, #0x0032
.BB@LABEL@1_1: ; bb
    pop de
    push de
    clrw bc
    movw ax, bc
    movw [de], ax
    movw ax, de
    incw ax
    incw ax
    movw de, ax
    movw ax, bc
    movw [de], ax
    movw ax, [sp+0x00]
    addw ax, #0x0004
    movw [sp+0x00], ax
    movw ax, hl
    addw ax, #0xFFFF
    movw hl, ax
    bnz $.BB@LABEL@1_1
.BB@LABEL@1_2: ; return
    pop hl
    ret
</pre></td></tr>
<tr><td>Code size: 18 bytes
Number of clock cycles: 1106</td><td>Code size: 36 bytes
Number of clock cycles: 1059</td></tr>
</table>

5.6 Using Tables

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

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

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

<table>
<tr><th>switch Statement</th><th>Equivalent Table-Based Code</th></tr>
<tr><td>C source code

<pre>
char func(int i)
{
  char ch;
  switch (i) {
  case 0:
    ch = 'a'; break;
  case 1:
    ch = 'x'; break;
  case 2:
    ch = 'b'; break;
  default:
    ch = 0; break;
  }
  return (ch);
}
</pre></td><td>C source code

<pre>
const char chbuf[] =
  {'a', 'x', 'b'};

char func(int i)
{
  if ((unsigned int)i < 3) {
    return (chbuf[i]);
  }
  return (0);
}
</pre></td></tr>
<tr><td>Assembly-language expanded code

<pre>
_func:
    .STACK _func = 4
    cmpw ax, #0x0000
    bz $.BB@LABEL@1_4
.BB@LABEL@1_1: ; entry
    addw ax, #0xFFFF
    bz $.BB@LABEL@1_5
.BB@LABEL@1_2: ; entry
    cmpw ax, #0x0001
    bz $.BB@LABEL@1_6
.BB@LABEL@1_3: ; switch_clause_bb5
    clrb a
    ret
.BB@LABEL@1_4: ; switch_break_bb
    mov a, #0x61
    ret
.BB@LABEL@1_5: ; switch_clause_bb3
    mov a, #0x78
    ret
.BB@LABEL@1_6: ; switch_clause_bb4
    mov a, #0x62
    ret
</pre></td><td>Assembly-language expanded code

<pre>
_func:
    .STACK _func = 4
    cmpw ax, #0x0003
    bnc $.BB@LABEL@1_2
.BB@LABEL@1_1: ; if_then_bb
    movw bc, ax
    mov a, SMRLW(_chbuf)[bc]
    ret
.BB@LABEL@1_2: ; bb10
    clrb a
    ret
</pre></td></tr>
<tr><td>Code size: 59 bytes
Number of clock cycles: 82</td><td>Code size: 44 bytes
Number of clock cycles: 78</td></tr>
</table>

5.7 Branches

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

Before Changing the Position of a Case

C source code

```
int func(int a)
{
  if (a == 1) {
    a = 2;
  }
  else if (a == 2) {
    a = 4;
  }
  else if (a == 3) {
    a = 8;
  }
  else {
    a = 0;
  }
  return (a);
}
```

Assembly-language expanded code

```
_func:
    .STACK _func = 4
    cmpw ax, #0x0001
    bnz $.BB@LABEL@1_2
.BB@LABEL@1_1: ;
entry.if_break_bb17_crit_edge
    onew ax
    incw ax
    br $.BB@LABEL@1_7
.BB@LABEL@1_2: ; if_else_bb
    cmpw ax, #0x0002
    bnz $.BB@LABEL@1_4
.BB@LABEL@1_3: ;
if_else_bb.if_break_bb17_crit_edge
    movw ax, #0x0004
    br $.BB@LABEL@1_7
.BB@LABEL@1_4: ; if_else_bb9
    cmpw ax, #0x0003
    oneb a
    skz
.BB@LABEL@1_5: ; if_else_bb9
    clrb a
.BB@LABEL@1_6: ; if_else_bb9
    mov x, #0x08
    mulu x
.BB@LABEL@1_7: ; if_break_bb17
    ret
```

Number of clock cycles: 26
(for **a=3**)

After Changing the Position of a Case

C source code

```
int func(int a)
{
  if (a == 3) {
    a = 8;
  }
  else if (a == 2) {
    a = 4;
  }
  else if (a == 1) {
    a = 2;
  }
  else {
    a = 0;
  }
  return (a);
}
```

Assembly-language expanded code

```
_func:
    .STACK _func = 4
    cmpw ax, #0x0003
    bnz $.BB@LABEL@1_2
.BB@LABEL@1_1: ;
entry.if_break_bb17_crit_edge
    movw ax, #0x0008
    br $.BB@LABEL@1_7
.BB@LABEL@1_2: ; if_else_bb
    cmpw ax, #0x0002
    bnz $.BB@LABEL@1_4
.BB@LABEL@1_3: ;
if_else_bb.if_break_bb17_crit_edge
    movw ax, #0x0004
    br $.BB@LABEL@1_7
.BB@LABEL@1_4: ; if_else_bb9
    cmpw ax, #0x0001
    oneb a
    skz
.BB@LABEL@1_5: ; if_else_bb9
    clrb a
.BB@LABEL@1_6: ; if_else_bb9
    mov x, #0x02
    mulu x
.BB@LABEL@1_7: ; if_break_bb17
    ret
```

Number of clock cycles: 17
(for **a=3**)

5.8 Inline Expansion

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

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

Before Inline Expansion

C source code

```
int x[10], y[10];

static void sub(int *a, int *b,
                int i)
{
  int temp;
  temp = a[i];
  a[i] = b[i];
  b[i] = temp;
}

void func()
{
  int i;
  for (i = 0; i < 10; i++) {
    sub(x, y, i);
  }
}
```

After Inline Expansion

C source code

```
int x[10], y[10];

#pragma inline (sub)
static void sub(int *a, int *b,
                int i)
{
  int temp;
  temp = a[i];
  a[i] = b[i];
  b[i] = temp;
}

void func()
{
  int i;
  for (i = 0; i < 10; i++) {
    sub(x, y, i);
  }
}
```

Assembly-language expanded code

```
_sub@1:
   .STACK _sub@1 = 8
   push ax
   push bc
   pop hl
   movw ax, de
   addw ax, ax
   movw bc, ax
   movw ax, hl
   addw ax, bc
   movw de, ax
   movw ax, [sp+0x00]
   addw ax, bc
   movw hl, ax
   movw ax, [hl]
   movw bc, ax
   movw ax, [de]
   movw [hl], ax
   movw ax, bc
   movw [de], ax
   pop hl
   ret
_func:
   .STACK _func = 6
   push hl
   clrw ax
   movw [sp+0x00], ax
.BB@LABEL@2_1: ; bb
   movw de, ax
   movw bc, #LOWW(_y)
   movw ax, #LOWW(_x)
   call $!_sub@1
   movw ax, [sp+0x00]
   incw ax
   movw [sp+0x00], ax
   cmpw ax, #0x000A
   bnz $.BB@LABEL@2_1
.BB@LABEL@2_2: ; return
   pop hl
   ret
```

Code size: 26 bytes
Number of clock cycles: 411
(When **-Oinline_level=1** is specified)

Assembly-language expanded code

```
_func:
   .STACK _func = 6
   push hl
   movw de, #LOWW(_y)
   movw bc, #0x000A
   movw hl, #LOWW(_x)
.BB@LABEL@1_1: ; bb
   movw ax, [hl]
   movw [sp+0x00], ax
   movw ax, [de]
   movw [hl], ax
   movw ax, [sp+0x00]
   movw [de], ax
   movw ax, bc
   addw ax, #0xFFFF
   movw bc, ax
   incw de
   incw de
   incw hl
   incw hl
   bnz $.BB@LABEL@1_1
.BB@LABEL@1_2: ; return
   pop hl
   ret
```

Code size: 31 bytes
Number of clock cycles: 183
(When **-Oinline_level=1** is specified)

Use of **#pragma inline** does not guarantee inline expansion. Functions may not be inline-expanded depending on whether **-Oinline_level** is specified, the contents of the function, or the status of compilation.

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

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

Identical Expressions Following a Branch

C source code

```
int s;

int func(int a, int b, int c)
{
  return (a + b + c);
}

int call_func(int x)
{
  if (x >= 0) {
    if (x > func(0, 1, 2)) {
      s++;
    }
  }
  else {
    if (x < -func(0, 1, 2)) {
      s--;
    }
  }
  return 0;
}
```

Expression before the Branch

C source code

```
int s;

int func(int a, int b, int c)
{
  return (a + b + c);
}

int call_func(int x)
{
  int tmp = func(0, 1, 2);
  if (x >= 0) {
    if (x > tmp) {
      s++;
    }
  }
  else {
    if (x < -tmp) {
      s--;
    }
  }
  return 0;
}
```

Assembly-language expanded code

```
_call_func:
   .STACK _call_func = 6
   push ax
   bt a.7, $.BB@LABEL@2_5
.BB@LABEL@2_1: ; if_then_bb
   movw de, #0x0002
   onew bc
   clrw ax
   call $!_func
   movw bc, ax
   movw ax, [sp+0x00]
   xor a, #0x80
   movw [sp+0x00], ax
   xchw ax, bc
   xor a, #0x80
   cmpw ax, bc
   bnc $.BB@LABEL@2_3
.BB@LABEL@2_2: ; if_then_bb9
   incw !LOWW(_s)
   br $.BB@LABEL@2_5
.BB@LABEL@2_3: ; if_else_bb
   movw de, #0x0002
   onew bc
   clrw ax
   call $!_func
   movw bc, ax
   clrw ax
   subw ax, bc
   xor a, #0x80
   movw bc, ax
   movw ax, [sp+0x00]
   cmpw ax, bc
   sknc
.BB@LABEL@2_4: ; if_then_bb18
   decw !LOWW(_s)
.BB@LABEL@2_5: ; if_break_bb22
   clrw ax
   pop hl
   ret
```

Code size: 55 bytes
Number of clock cycles: 65

Assembly-language expanded code

```
_call_func:
   .STACK _call_func = 6
   push ax
   movw de, #0x0002
   onew bc
   clrw ax
   call $!_func
   movw bc, ax
   movw ax, [sp+0x00]
   bt a.7, $.BB@LABEL@2_5
.BB@LABEL@2_1: ; if_then_bb
   xor a, #0x80
   movw de, ax
   movw ax, bc
   xor a, #0x80
   cmpw ax, de
   bnc $.BB@LABEL@2_3
.BB@LABEL@2_2: ; if_then_bb12
   incw !LOWW(_s)
   br $.BB@LABEL@2_5
.BB@LABEL@2_3: ; if_else_bb
   clrw ax
   subw ax, bc
   xor a, #0x80
   movw bc, ax
   movw ax, de
   cmpw ax, bc
   sknc
.BB@LABEL@2_4: ; if_then_bb21
   decw !LOWW(_s)
.BB@LABEL@2_5: ; if_break_bb25
   clrw ax
   pop hl
   ret
```

Code size: 44 bytes
Number of clock cycles: 50

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

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

<table>
<tr><th>Complicated Sequence</th><th>Single if Statement</th></tr>
<tr><td>C source code


```
int x;

int func(int s, int t)
{
  s &= 1;
  t &= 1;
  if (!s) {
    if (t) {
       x = 1;
    }
  }
  else {
    if (!t) {
       x = 1;
    }
  }
  return 0;
}
```

</td><td>C source code


```
int x;

int func(int s, int t)
{
  s &= 1;
  t &= 1;
  if (! (s ^ t)) {
    x = 1;
  }
  return 0;
}
```

</td></tr>
<tr><td>Assembly-language expanded code


```
_func:
   .STACK _func = 4
   movw de, ax
   mov a, c
   and a, #0x01
   mov x, a
   mov a, e
   bt a.0, $.BB@LABEL@1_2
.BB@LABEL@1_1: ; bb10.thread
   cmp0 x
   bnz $.BB@LABEL@1_3
   br $.BB@LABEL@1_4
.BB@LABEL@1_2: ; if_else_bb
   cmp0 x
   bnz $.BB@LABEL@1_4
.BB@LABEL@1_3: ; if_then_bb28
   onew ax
   movw !LOWW(_x), ax
.BB@LABEL@1_4: ; if_break_bb30
   clrw ax
   ret
```

</td><td>Assembly-language expanded code


```
_func:
   .STACK _func = 4
   xor a, b
   xch a, x
   xor a, c
   xch a, x
   mov a, x
   bt a.0, $.BB@LABEL@1_2
.BB@LABEL@1_1: ; if_then_bb
   onew ax
   movw !LOWW(_x), ax
.BB@LABEL@1_2: ; if_break_bb
   clrw ax
   ret
```

</td></tr>
<tr><td>Code size: 23 bytes
Number of clock cycles: 27</td><td>Code size: 16 bytes
Number of clock cycles: 22</td></tr>
</table>

5.11 Types of Variables

When using variables, specify the types having the minimum usable sizes for the given purposes. This is because RL78 devices excel in handling small-type variables.

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

int-Type Variables

C source code

```
int func(int a, int b, int c)
{
  int t = a + b;
  return (t >> c);
}
```

Assembly-language expanded code

```
_func:
    .STACK _func = 6
    push hl
    movw hl, ax
    movw ax, de
    clrb a
    mov a, x
    mov [sp+0x00], a
    movw ax, hl
    addw ax, bc
    movw bc, ax
    mov a, [sp+0x00]
    xchw ax, bc
    cmp0 b
    bz $.BB@LABEL@1_2
.BB@LABEL@1_1: ; entry
    sarw ax, 0x01
    dec b
    bnz $.BB@LABEL@1_1
.BB@LABEL@1_2: ; entry
    pop hl
    ret
```

Code size: 23 bytes
Number of clock cycles: 32

char-Type Variables

C source code

```
char func(char a, char b, char c)
{
  char t = a + b;
  return (t >> c);
}
```

Assembly-language expanded code

```
_func:
    .STACK _func = 4
    add x, a
    mov a, x
    shrw ax, 8+0x00000
    cmp0 c
    bz $.BB@LABEL@1_2
.BB@LABEL@1_1: ; entry
    shrw ax, 0x01
    dec c
    bnz $.BB@LABEL@1_1
.BB@LABEL@1_2: ; entry
    mov a, x
    ret
```

Code size: 15 bytes
Number of clock cycles: 25

Note that when a variable is used as an element of an array, the type of the variable may be converted into the **int** type at the time of address calculation. Therefore, using a 16-bit type for such variables may improve the output code.

char-Type Variable	int-Type Variable

C source code
 	
`char array[100];`
`char index = 10;`
 	
`char func(void)`
`{`
`  return array[index-2];`
`}`
	C source code
	
	`char array[100];`
	`int index = 10;`
	
	`char func(void)`
	`{`
	`  return array[index-2];`
	`}`

Assembly-language expanded code
 	
`_func:`
`  .STACK _func = 4`
`  mov a, !LOWW(_index)`
`  shrw ax, 8+0x00000`
`  addw ax, #LOWW(_array+0x0FFFE)`
`  movw de, ax`
`  mov a, [de]`
`  ret`
	Assembly-language expanded code
	
	`_func:`
	`  .STACK _func = 4`
	`  movw ax, #LOWW(_array+0x0FFFE)`
	`  addw ax, !LOWW(_index)`
	`  movw de, ax`
	`  mov a, [de]`
	`  ret`

Code size: 11 bytes Number of clock cycles: 13	Code size: 9 bytes Number of clock cycles: 12

In cases that require comparison operations, such as using the loop counters in **for** statements, specify the **unsigned** type if it is obvious that the value of a variable will be never be less than 0. Comparison instructions of the RL78 family are for comparing unsigned values, so this simplifies the output code.

int-Type Variable	Unsigned int-Type Variable

C source code
 	
`int value;`
`void func(int num)`
`{`
`  int i;`
`  for (i = 0; i < num; i++) {`
`  value += 10;`
`  }`
`}`
	C source code
	
	`int value;`
	`void func(unsigned int num)`
	`{`
	`  unsigned int i;`
	`  for (i = 0; i < num; i++) {`
	`  value += 10;`
	`  }`
	`}`

Assembly-language expanded code
 	
`_func:`
`  .STACK _func = 4`
`  movw de, ax`
`  clrw bc`
`.BB@LABEL@1_1: ; bb6`
`  movw ax, de`
`  xor a, #0x80`
`  movw hl, ax`
`  movw ax, bc`
`  xor a, #0x80`
`  cmpw ax, hl`
`  bnc $.BB@LABEL@1_3`
`.BB@LABEL@1_2: ; bb`
`  movw ax, #0x000A`
`  addw ax, !LOWW(_value)`
`  movw !LOWW(_value), ax`
`  incw bc`
`  br $.BB@LABEL@1_1`
`.BB@LABEL@1_3: ; return`
`  ret`
	Assembly-language expanded code
	
	`_func:`
	`  .STACK _func = 4`
	`  movw bc, ax`
	`  clrw ax`
	`.BB@LABEL@1_1: ; entry`
	`  movw de, ax`
	`.BB@LABEL@1_2: ; bb6`
	`  cmpw ax, bc`
	`  bz $.BB@LABEL@1_4`
	`.BB@LABEL@1_3: ; bb`
	`  movw ax, #0x000A`
	`  addw ax, !LOWW(_value)`
	`  movw !LOWW(_value), ax`
	`  movw ax, de`
	`  incw ax`
	`  br $.BB@LABEL@1_1`
	`.BB@LABEL@1_4: ; return`
	`  ret`

Code size: 25 bytes Number of clock cycles: 168	Code size: 20 bytes Number of clock cycles: 134

5.12 Unifying Common case Processing in switch Statements

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

Same Processing at Multiple Destinations	Unified Processing

C source code
 	
`void func(void)`
`{`
`  switch(x) {`
`  case 0:`
`  dummy1();`
`  break;`
`  case 1:`
`  dummy1();`
`  break;`
`  case 2:`
`  dummy1();`
`  break;`
`  case 3:`
`  dummy2();`
`  break;`
`  case 4:`
`  dummy2();`
`  break;`
`  default:`
`  break;`
`  }`
`}`
	C source code
	
	`void func(void)`
	`{`
	`  switch(x) {`
	`  case 0:`
	`  case 1:`
	`  case 2:`
	`  dummy1();`
	`  break;`
	`  case 3:`
	`  case 4:`
	`  dummy2();`
	`  break;`
	`  default:`
	`  break;`
	`  }`
	`}`

Assembly-language expanded code	Assembly-language expanded code

`_func:`
`  .STACK _func = 4`
`  movw ax, !LOWW(_x)`
`  cmpw ax, #0x0000`
`  bz $.BB@LABEL@3_6`
`.BB@LABEL@3_1: ; entry`
`  addw ax, #0xFFFF`
`  bz $.BB@LABEL@3_6`
`.BB@LABEL@3_2: ; entry`
`  addw ax, #0xFFFF`
`  bz $.BB@LABEL@3_6`
`.BB@LABEL@3_3: ; entry`
`  addw ax, #0xFFFF`
`  bz $.BB@LABEL@3_7`
`.BB@LABEL@3_4: ; entry`
`  cmpw ax, #0x0001`
`  bz $.BB@LABEL@3_7`
`.BB@LABEL@3_5: ; return`
`  ret`
`.BB@LABEL@3_6: ; switch_clause_bb2`
`  br $_dummy1`
`.BB@LABEL@3_7: ; switch_clause_bb4`
`  br $_dummy2`
	`_func:`
	`  .STACK _func = 4`
	`  movw ax, !LOWW(_x)`
	`  cmpw ax, #0x0003`
	`  bc $.BB@LABEL@3_3`
	`.BB@LABEL@3_1: ; entry`
	`  addw ax, #0xFFFD`
	`  cmpw ax, #0x0002`
	`  bc $.BB@LABEL@3_4`
	`.BB@LABEL@3_2: ; return`
	`  ret`
	`.BB@LABEL@3_3: ; switch_clause_bb`
	`  br $_dummy1`
	`.BB@LABEL@3_4: ; switch_clause_bb1`
	`  br $_dummy2`

Code size: 33 bytes Number of clock cycles: 23	Code size: 21 bytes Number of clock cycles: 17

5.13 Replacing for Loops with do-while Loops

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

for Loop

C source code

```
int array[10][10];

void func(int nsize, int msize)
{
  int i;
  int *p;
  int s;
  p = &array[0][0];
  s = nsize * msize;
  for (i = 0; i < s; i++) {
    *p++ = 0;
  }
}
```

do-while Loop

C source code

```
int array[10][10];

void func(int nsize, int msize)
{
  int i;
  int *p;
  int s;
  p = &array[0][0];
  s = nsize * msize;
  i = 0;
  do {
    *p++ = 0;
    i++;
  } while (i != s);
}
```

Assembly-language expanded code	Assembly-language expanded code

_func:
.STACK _func = 8
push hl
push bc
pop de
movw bc, ax
movw ax, de
mulh
movw [sp+0x00], ax
clrw bc
movw de, #LOWW(_array)
.BB@LABEL@1_1: ; bb13
xor a, #0x80
movw hl, ax
movw ax, bc
xor a, #0x80
cmpw ax, hl
bnc $.BB@LABEL@1_3
.BB@LABEL@1_2: ; bb
clrw ax
movw [de], ax
incw de
incw de
incw bc
movw ax, [sp+0x00]
br $.BB@LABEL@1_1
.BB@LABEL@1_3: ; return
pop hl
ret
	_func:
	.STACK _func = 6
	push bc
	pop de
	movw bc, ax
	movw ax, de
	mulh
	movw bc, ax
	movw de, #LOWW(_array)
	.BB@LABEL@1_1: ; bb
	clrw ax
	movw [de], ax
	movw ax, bc
	addw ax, #0xFFFF
	movw bc, ax
	incw de
	incw de
	bnz $.BB@LABEL@1_1
	.BB@LABEL@1_2: ; return
	ret

Code size: 34 bytes Number of clock cycles: 1626	Code size: 23 bytes Number of clock cycles: 1112

5.14 Replacing Division by Powers of Two with Shift Operations

If the divisor in division is a power of two, replace the division with a shift operation.

Division by a Power of Two	Shift Operation

C source code
 	
`int s;`
`void func(void)`
`{`
`  s = s / 2;`
`}`
	C source code
	
	`int s;`
	`void func(void)`
	`{`
	`  s = s >> 1;`
	`}`

Assembly-language expanded code
 	
`_func:`
`  .STACK _func = 4`
`  movw ax, !LOWW(_s)`
`  onew bc`
`  incw bc`
`  call !!__COM_sidiv`
`  movw !LOWW(_s), ax`
`  ret`
	Assembly-language expanded code
	
	`_func:`
	`  .STACK _func = 4`
	`  movw ax, !LOWW(_s)`
	`  sarw ax, 0x01`
	`  movw !LOWW(_s), ax`
	`  ret`

Code size: 13 bytes Number of clock cycles: 75	Code size: 9 bytes Number of clock cycles: 9

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

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

<table>
<tr><th>Bit Fields</th><th>char Type</th></tr>
<tr><td>C source code

<pre>
struct {
  unsigned char b0:1;
  unsigned char b1:2;
} dw;

unsigned char dummy;

int func(void)
{
  if (dw.b1) {
    dummy++;
    }
  return (0);
}
</pre></td><td>C source code

<pre>
struct {
  unsigned char b0:1;
  unsigned char b1;
} dw;

unsigned char dummy;

int func(void)
{
  if (dw.b1) {
    dummy++;
    }
  return (0);
}
</pre></td></tr>
<tr><td>Assembly-language expanded code

<pre>
_func:
    .STACK _func = 4
    mov a, #0x06
    and a, !LOWW(_dw)
    bnz $.BB@LABEL@1_2
.BB@LABEL@1_1: ; if_break_bb
    clrw ax
    ret
.BB@LABEL@1_2: ; if_then_bb
    inc !LOWW(_dummy)
    br $.BB@LABEL@1_1
</pre></td><td>Assembly-language expanded code

<pre>
_func:
    .STACK _func = 4
    cmp0 !LOWW(_dw+0x00001)
    bnz $.BB@LABEL@1_2
.BB@LABEL@1_1: ; if_break_bb
    clrw ax
    ret
.BB@LABEL@1_2: ; if_then_bb
    inc !LOWW(_dummy)
    br $.BB@LABEL@1_1
</pre></td></tr>
<tr><td>Code size: 13 bytes
Number of clock cycles: 11
ROM size: 1 byte</td><td>Code size: 12 bytes
Number of clock cycles: 10
ROM size: 2 bytes</td></tr>
</table>

5.16 Alignment of a Structure

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

For example, the boundary alignment value for the **long** type is 2 bytes and the **long**-type variable must be allocated to the address which is a multiple of 2. For a structure variable, alignment applies to both its members and to the structure variable itself. The boundary alignment values for members are the same as those for variables of the same type that are not structure members. The boundary alignment value of a structure variable is the highest value among those of its members. If allocating the members of a structure variable without any spaces would violate the required alignments, alignment of the addresses is obtained by including spaces. That is, padding is required. Padding is also required when the size of a structure variable is not a multiple of the highest boundary alignment value. Frequent requirements for padding lower the efficiency of allocation to memory.

Before Alignment	After Alignment

C source code
 	
`// The boundary alignment value is`
`// 2 bytes since the member with`
`// the maximum alignment value is`
`// of the long type.`
`struct str {`
`  char c1;  // 1 byte`
`// 1 byte for padding`
`  long l1;  // 4 bytes`
`  char c2;  // 1 byte`
`  char c3;  // 1 byte`
`  char c4;  // 1 byte`
`// 1 byte for padding`
`} str1;`
	C source code
	
	`// The boundary alignment value is`
	`// 2 bytes since the member with`
	`// the maximum alignment value is`
	`// of the long type.`
	`struct str {`
	`  char c1;  // 1 byte`
	`  char c2;  // 1 byte`
	`  char c3;  // 1 byte`
	`  char c4;  // 1 byte`
	`  long l1;  // 4 bytes`
	`} str1;`

Assembly-language expanded code
 	
`_str1:`
`  .DS (10)`
	Assembly-language expanded code
	
	`_str1:`
	`  .DS (8)`

RAM size: 10 bytes	RAM size: 8 bytes

Website and Support

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

Inquiries
http://www.renesas.com/contact/

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	November 26, 2018		New release

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
5. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.

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

   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
6. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
7. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
8. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
9. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
10. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

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

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

(Rev.4.0-1 November 2017)

RENESAS

SALES OFFICES

Renesas Electronics Corporation

http://www.renesas.com

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

Renesas Electronics Corporation
TOYOSU FORESIA, 3-2-24 Toyosu, Koto-ku, Tokyo 135-0061, Japan

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

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

Renesas Electronics Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K
Tel: +44-1628-651-700

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

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

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

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

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

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

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

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

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