IAR Embedded Workbench for RL78

Programming Techniques

R01AN3735EJ0100
Rev.1.00
Apr. 14, 2017

Introduction

This application note describes how to reduce the code size, increase the execution speed, and programming techniques to avoid bugs when using IAR Embedded Workbench for RL78.

The following versions of the integrated development environments are supported.

- IAR Embedded Workbench IDE V7.4.1.4269
- IAR C/C++ Compiler for Renesas RL78 V2.21.1.1833

Target Device

RL78 Family

Contents

1. Code Size Reduction

1.1 Size of Variables

When using variables, specify the type having the minimum allowable size.
This is because the RL78 devices excel in handling small-type variables.

Before Change	After Change

void main(void)
{
    signed int i;
 	
    for (i=0; i < 10; i++)
    {
        NOP();
    }
}
	void main(void)
	{
	    signed char i;
	
	    for (i=0; i < 10; i++)
	    {
	        NOP();
	    }
	}

Figure 1.1 C Source Code

Before Change			After Change		
MOVW	AX, #0x0	2	MOV	A, #0x0	2
MOVW	HL, AX	1	MOV	X, A	1
??main_0:			??main_0:		
MOVW	AX, HL	2	MOV	A, X	1
XOR	A, #0x80	2	XOR	A, #0x80	2
CMPW	AX, #0x800A	3	SUB	A, #0x8A	2
BNC	??main_1	2	BNC	??main_1	2
NOP		1	NOP		1
INCW	HL	1	INC	X	1
BR	S:??main_0	3	BR	S:??main_0	3
		17 bytes			15 bytes

Figure 1.2 Output Assembler

1.2 Unsigned Variables

Add “unsigned” for all data that never handle negative values.
This is because the RL78 devices excel in handling unsigned variables.

Before Change	After Change

signed int data0;
signed int data1;
 	
void main(void)
{
    if (data0 > 10)
    {
        data1++;
    }
}
	unsigned int data0;
	unsigned int data1;
	
	void main(void)
	{
	    if (data0 > 10)
	    {
	        data1++;
	    }
	}

Figure 1.3 C Source Code

Before Change			After Change		
MOVW	AX, N:_data0	2	MOVW	AX, N:_data0	2
XOR	A, #0x80	2			
CMPW	AX, #0x800B	3	CMPW	AX, #0xB	3
SKC		2	SKC		2
INCW	N:_data1	4	INCW	N:_data1	4
		13 bytes			11 bytes

Figure 1.4 Output Assembler

1.3 saddr Area

Use the __saddr qualifier or #pragma saddr declaration for frequently used external variables and static variables within functions.

Allocating variables in the saddr area improves the code.

For a one-bit field especially, the __saddr qualifier or #pragma saddr declaration can be expected to have a large effect.

Alternatively, the variables/functions information file can be used to allocate variables to the saddr area.

<table>
<tr><th>Before Change</th><th>After Change</th></tr>
<tr><td><pre>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 main(void)
{
    data0.b4 = data1.b1;
}</pre></td><td><pre>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 main(void)
{
    data0.b4 = data1.b1;
}</pre></td></tr>
</table>

Figure 1.5 C Source Code

<table>
<tr><th colspan="3">Before Change</th><th colspan="3">After Change</th></tr>
<tr><td>MOVW</td><td>HL, #LWRD(_data1)</td><td>3</td><td rowspan="2">MOV1</td><td rowspan="2">CY, S:_data1.1</td><td rowspan="2">3</td></tr>
<tr><td>MOV1</td><td>CY, [HL].1</td><td>2</td></tr>
<tr><td>MOVW</td><td>HL, #LWRD(_data0)</td><td>3</td><td rowspan="2">MOV1</td><td rowspan="2">S:_data0.4, CY</td><td rowspan="2">3</td></tr>
<tr><td>MOV1</td><td>[HL].4, CY</td><td>2</td></tr>
<tr><td colspan="3">10 bytes</td><td colspan="3">6 bytes</td></tr>
</table>

Figure 1.6 Output Assembler

1.4 callt Function

Use the __callt qualifier or #pragma callt declaration for frequently called functions.

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 for direct function calls.

<table>
<tr><th>Before Change</th><th>After Change</th></tr>
<tr><td><pre>void func_sub(void)
{
    …
}

void func(void)
{
    func_sub();
        :
    func_sub();
}</pre></td><td><pre>__callt void func_sub(void)
{
    …
}

void func()
{
    func_sub();
        :
    func_sub();
}</pre></td></tr>
</table>

Figure 1.7 C Source Code

<table>
<tr><th colspan="2">Before Change</th><th colspan="2">After Change</th></tr>
<tr><td><pre>   CALL   _func_sub
    CALL   _func_sub</pre></td><td>4
4</td><td><pre>   CALLT   [__T_func_sub]
    CALLT   [__T_func_sub]

__T_func_sub:
    DW   _func_sub</pre></td><td>2
2

2</td></tr>
<tr><td colspan="2">8 bytes</td><td colspan="2">6 bytes</td></tr>
</table>

Figure 1.8 Output Assembler

Notes:
- A table of addresses for function calls in generated (.callt0).
- Due to generation of this table, code size reduction is effective for a function called only once.
- The CALLT instruction requires more clock cycles for execution that the CALL instruction.
- Alternatively, the variables/functions information file can be used to specify declarations of the functions of the functions to be called through the CALLT instruction.

1.5 Alignment of Structure Members

In the RL78family of devices, reading or writing in word units cannot start from an odd address;

Data for alignment is inserted by the default option setting so that 2-bytes or larger members are allocated to even addresses.

Therefore, take care regarding the alignment of structure members and do not leave unused space between members.

Before Change	After Change
struct	struct
{	{
signed char a;	signed char a;
signed int b;	signed char c;
signed char c;	signed int b;
struct	struct
{	{
signed int d;	signed int d;
signed int e;	signed int e;
} f;	} f;
} data;	} data;

Figure 1.9 C Source Code

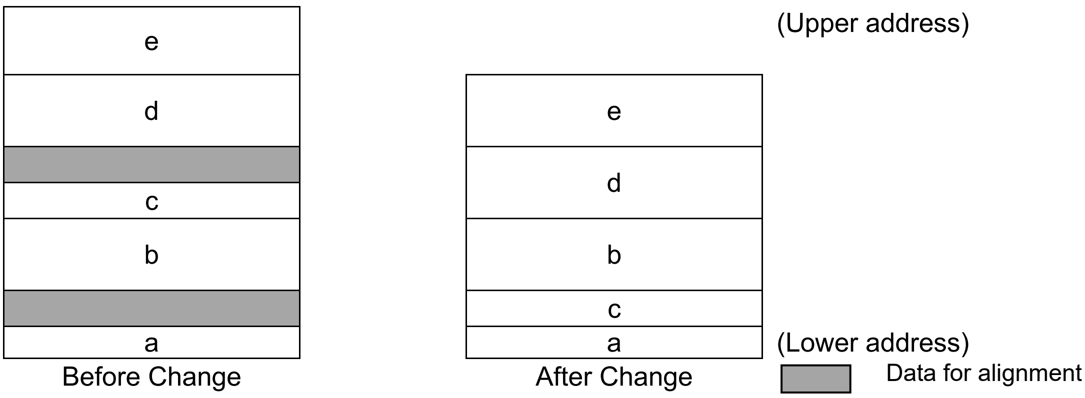

Figure 1.10 Data Allocation in Memory

1.6 Bit Fields and 1-Byte Variable

When the size of bit-field member is two or more bits, use the char type instead of a bit field (two or more bits). Note that the size of RAM area used will increase when this is done.

Before Change	After Change

struct
{
    unsigned char b0:1;
    unsigned char b1:2;
} data;
 	
unsigned char dummy;
 	
if (data.b1)
{
    dummy++;
}
	unsigned char data;
	unsigned char dummy;
	
	if (data)
	{
	    dummy++;
	}

Figure 1.11 C Source Code

Before Change			After Change		
MOV	A, N:_data	3			
SHR	A, 0x1	2			
AND	A, #0x3	2			
CMP0	A	1	CMP0	N:_data	1
SKZ		2	SKZ		2
INC	N:_dummy	3	INC	N:_dummy	3
		13 bytes			6 bytes

Figure 1.12 Output Assembler

1.7 Type Conversion

The short type and char type variables are extended to the int type when calculated, and the unsigned short type and unsigned char type variables are extended to the unsigned int type when calculated. Therefore, many instructions that perform type conversion are generated in a program that uses these variables. When type conversion is performed in programming, instructions that perform type conversion are not generated and thus the code size is reduced.

Before Change	After Change

void main(void)
{
    unsigned char i;
 	
    for (i = 0; i <4; i++)
    {
        array[2 + i] = *(p + i);
    }
}
	void main(void)
	{
	    int i;
	
	    for (i = 0; i <4; i++)
	    {
	        array[2 + i] = *(p + i);
	    }
	}

Figure 1.13 C Source Code

Remark: array[] and *p are global variables.

Before Change				After Change			
	MOV	A, #0x0	2		MOVW	AX, #0x0	4
	MOV	E, A	2		MOVW	HL, AX	1
??main_0:				??main_0:			
	MOV	A, E	2		MOVW	AX, HL	1
					XOR	A, #0x80	2
	CMP	A, #0x4	2		CMPW	AX, #0x8004	3
	BNC	??main_1	2		BNC	??main_1	2
	MOV	A, E	2				
	XCH	A, X	1				
	MOV	A, #0x0	2		MOVW	AX, HL	1
	MOVW	BC, #0x2	3		MOVW	BC, #0x2	3
	MULHU		3		MULHU		3
	MOVW	HL, N:_p	4		MOVW	DE, N:_p	4
	XCHW	AX, HL	1		XCHW	AX, DE	1
	ADDW	AX, HL	1		ADDW	AX, DE	1
	XCHW	AX, HL	1		XCHW	AX, DE	1
	MOVW	AX, [HL]	1		MOVW	AX, [DE]	1
	PUSH	AX	1		PUSH	AX	1
	MOV	A, E	1				
	XCH	A, X	2				
	MOV	A, #0x0	2		MOVW	AX, HL	1
	MOVW	BC, #0x2	3		MOVW	BC, #0x	3
	MULHU		3		MULHU		3
	ADDW	AX, #LWRD(_array+4)	2		ADDW	AX,#LWRD(_array+4)	2
	MOVW	HL, AX	1		MOVW	DE, AX	1
	POP	AX	1		POP	AX	1
	MOVW	[HL], AX	1		MOVW	[DE], AX	1
	INC	E	1		INCW	HL	1
	BR	S:??main_0	3		BR	S:??main_0	3
??main_1:				??main_1:			
	RET		1		RET		1
			51 bytes				46 bytes

Figure 1.14 Output Assembler

1.8 Deleting Induction Variables

A variable that controls a loop is called an induction variable. When loops are controlled using other variables, induction variables are eliminated, and thus the code size is reduced.

<table>
<tr><th>Before Change</th><th>After Change</th></tr>
<tr><td>

```
int main(void)
{
    int i;

    for (i = 0; *(table + i) != 0; ++i)
    {
        if (x== (*(table + i) & 0xFF))
        {
            return(*(table +i) & 0xFF00);
        }
    }
}
```

</td><td>

```
int main(void)
{
    const unsigned short *p;

    for (p = table; *p != 0; ++p)
    {
        if (x == (*p & 0xFF))
        {
            return(*p & 0xFF00);
        }
    }
}
```

</td></tr>
</table>

Figure 1.15 C Source Code

Remark: x and *table are global variables.

Before Change				After Change			
	MOVW	AX, #0x0	3		MOVW	DE, N:_table	3
	MOVW	HL, AX	1		PUSH	DE	1
					POP	HL	1
??main_0:				??main_0:			
	MOVW	AX, HL	1		MOVW	AX, [HL]	1
	MOVW	BC, #0x2	3		CMPW	AX, #0x0	3
	MULHU		3		BZ	??main_1	2
	MOVW	DE, N:_table	3				
	XCHW	AX, DE	1		MOVW	AX, [HL]	1
	ADDW	AX, DE	1		AND	A, #0x0	2
	XCHW	AX, DE	1		XCH	A, X	1
	MOVW	AX, [DE]	1		AND	A, #0xFF	2
	CMPW	AX, #0x0	3		XCH	A, X	1
	BZ	??main_1	2		MOVW	DE, N:_x	3
					CMPW	AX, DE	1
	MOVW	AX, HL	1		BNZ	??main_2	2
	MOVW	BC, #0x2	3				
	MULHU		3		MOVW	AX, [HL]	1
	MOVW	DE, N:_table	3		AND	A, #0xFF	2
	XCHW	AX, DE	1		XCH	A, X	1
	ADDW	AX, DE	1		AND	A, #0x0	2
	XCHW	AX, DE	1		XCH	A, X	1
	MOVW	AX, [DE]	1		RET		1
	AND	A, #0x0	2				
	XCH	A, X	1	??main_2:			
	AND	A, #0xFF	2		XCHW	AX, HL	1
	XCH	A, X	1		ADDW	AX, #0x2	2
	MOVW	DE, N:_x	3		XCHW	AX, HL	1
	CMPW	AX, DE	1		BR	S:??main_0	3
	BNZ	??main_2	2				
				??main_1:			
	MOVW	AX, HL	1		MOVW	AX, #0x0	3
	MOVW	BC, #0x2	3		RET		1
	MULHU		3				
	MOVW	HL, N:_table	3				
	XCHW	AX, HL	1				
	ADDW	AX, HL	1				
	XCHW	AX, HL	1				
	MOVW	AX, [HL]	1				
	AND	A, #0xFF	2				
	XCH	A, X	1				
	AND	A, #0x0	2				
	XCH	A, X	1				
	RET		1				
??main_2:							
	INCW	HL	1				
	BR	S:??main_0	3				
??main_1:							
	MOVW	AX, #0x0	3				
	RET		1				
			78bytes				43bytes

Figure 1.16 Output Assembler

1.9 Loop Fusion

Loop fusion refers to integrating different loop statements in the same function into a single one, thus reducing the number of loop statements. Loop fusion reduces the code size and makes code run faster as well by eliminating the loop iteration overhead.

<table>
<tr><th>Before Change</th><th>After Change</th></tr>
<tr><td>

```
void main(void)
{
    uint8_t i = 0;
    uint8_t total = 0;
    uint8_t test[10] = {0};

    for (i = 0; i < 10; i++)
    {
        test[i] = CSS;
    }
    for (i = 0; i < 10; i++)
    {
        total += test[i];
    }
}
```

</td><td>

```
void main(void)
{
    uint8_t i = 0;
    uint8_t total = 0;
    uint8_t test[10] = {0};

    for (i = 0; i < 10; i++)
    {
        test[i] = CSS;
        total += test[i];
    }
}
```

</td></tr>
</table>

Figure 1.17 C Source Code

Before Change			After Change		
SUBW	SP, #0xA	2	SUBW	SP, #0xA	2
MOV	C, #0x0	2	MOV	C, #0x0	2
MOV	B, #0x0	2	MOV	B, #0x0	2
MOVW	AX, SP	1	MOVW	AX, SP	1
MOVW	DE, AX	1	MOVW	DE, AX	1
MOVW	HL, #LWRD(?_0)	3	MOVW	HL, #LWRD(?_0)	3
MOVW	AX, [HL]	1	MOVW	AX, [HL]	1
MOVW	[DE], AX	1	MOVW	[DE], AX	1
MOVW	AX, [HL+0x02]	2	MOVW	AX, [HL+0x02]	2
MOVW	[DE+0x02], AX	2	MOVW	[DE+0x02], AX	2
MOVW	AX, [HL+0x04]	2	MOVW	AX, [HL+0x04]	2
MOVW	[DE+0x04], AX	2	MOVW	[DE+0x04], AX	2
MOVW	AX, [HL+0x06]	2	MOVW	AX, [HL+0x06]	2
MOVW	[DE+0x06], AX	2	MOVW	[DE+0x06], AX	2
MOVW	AX, [HL+0x08]	2	MOVW	AX, [HL+0x08]	2
MOVW	[DE+0x08], AX	2	MOVW	[DE+0x08], AX	2
MOV	A, #0x0	2	MOV	A, #0x0	2
MOV	C, A	1	MOV	C, A	1
??main_0:			??main_0:		
MOV	A, C	1	MOV	A, C	1
CMP	A, #0xA	2	CMP	A, #0xA	2
BNC	??main_1	2	BNC	??main_1	2
MOV	A, #0x0	2	MOV	A, #0x0	2
MOV1	CY, 0xFFFA4.6	2	MOV1	CY, 0xFFFA4.6	2
ROLC	A, 0x1	2	ROLC	A, 0x1	2
MOV	D, A	1	MOV	D, A	1
MOVW	AX, SP	1	MOVW	AX, SP	1
MOVW	HL, AX	1	MOVW	HL, AX	1
XCH	A, D	1	XCH	A, D	1
MOV	[HL+C], A	2	MOV	[HL+C], A	2
XCH	A, D	1	XCH	A, D	1
INC	C	1	MOVW	AX, SP	1
BR	S:??main_0	3	MOVW	HL, AX	1
			MOV	A, [HL+C]	2
??main_1:			MOV	X, A	1
MOV	A, #0x0	2	MOV	A, B	1
MOV	C, A	1	ADD	A, X	1
			MOV	B, A	1
??main_2:			INC	C	1
MOV	A, C	1	BR	S:??main_0	3
CMP	A, #0xA	2			
BNC	??main_3	2	??main_1:		
			ADDW	SP, #0xA	2
MOVW	AX, SP	2	RET		1
MOVW	HL, AX	1			
MOV	A, [HL+C]	2			
MOV	X, A	1			
MOV	A, B	1			
ADD	A, X	2			
MOV	B, A	1			
INC	C	1			
BR	S:??main_2	3			
??main_3:					
ADDW	SP, #0xA	2			
RET		1			
		80 bytes			65 bytes

Figure 1.18 Output Assembler

1.10 Memory Models

The RL78 Family includes a small model that generates code with 16-bit address length and a medium model that generates code with 20-bit address length.

Model	Size	Functions	Variables
Small model	Program: 64Kbytes or smaller; Data 64Kbytes or smaller	near	near
Medium model	Program: 64Kbytes or larger; Data: 64Kbytes or smaller	far	near

Figure 1.19 Memory Model Type

If a program exceeds 64Kbytes, select the medium model. Adding the __near modifier to a frequently called function during programming reduces the code size.

However, when the __near modifier and the __far modifier are added, a pointer variable type handling them must match.

2. Faster Execution Speed

2.1 Consecutive Access to Array

When accessing an array consecutively in a loop, use a pointer variable. Without the use of a pointer variable, the process to obtain a real address from an array subscript may be output every time and thus the execution speed may slow down.

Note: The execution times of the programs in this chapter are all measured by using the RL78 simulator in the CS+ integrated development environment.

<table>
<tr><th>Before Change</th><th>After Change</th></tr>
<tr><td>

```
int i;

sum = 0;

for (i = 0; i < 10; i++)
{
    sum += array[i];
}
```

</td><td>

```
int i;
int *p;

sum = 0;
p = &array[0];

for (i = 0; i < 10; i++)
{
    sum += *p++;
}
```

</td></tr>
</table>

Figure 2.1 C Source Code Example

Remark: sum and array[] are global variables.

<table>
<tr><th colspan="2">Before Change</th><th colspan="2">After Change</th></tr>
<tr><td>

```
        MOVW    AX, #0x0
        MOVW    N:_sum, AX
        MOVW    AX, #0x0
        MOVW    HL, AX

??main_0:
        MOVW    AX, HL
        XOR     A, #0x80
        CMPW    AX, #0x800A
        BNC     ??main_1

        MOVW    AX, HL
        MOVW    BC, #0x2
        MULHU
        ADDW    AX, #LWRD(_array)
        MOVW    DE, AX
        MOVW    AX, [DE]
        ADDW    AX, N:_sum
        MOVW    N:_sum, AX
        INCW    HL
        BR      S:??main_0

??main_1:
        RET
```

</td><td>

```
3
4
3
1


1
2
3
2

1
3
3
3
1
1
3
4
1
3


1
```

</td><td>

```
        MOVW    AX, #0x0
        MOVW    N:_sum, AX
        MOVW    AX, #LWRD(_array)
        MOVW    HL, AX
        MOVW    AX, #0x0
        MOVW    DE, AX

??main_0:
        MOVW    AX, DE
        XOR     A, #0x80
        CMPW    AX, #0x800A
        BNC     ??main_1

        MOVW    AX, [HL]
        ADDW    AX, N:_sum
        MOVW    N:_sum, AX
        XCHW    AX, HL
        ADDW    AX, #0x2
        XCHW    AX, HL
        INCW    DE
        BR      S:??main_0

??main_1:
        RET
```

</td><td>

```
3
4
3
1
3
1


1
2
3
2

1
3
4
1
3
1
1
3


1
```

</td></tr>
<tr><td colspan="2" align="right">43 bytes</td><td colspan="2" align="right">41 bytes</td></tr>
<tr><td colspan="2" align="right">Execution Time: 30 cycles</td><td colspan="2" align="right">Execution Time: 29 cycles</td></tr>
</table>

Figure 2.2 Output Assembler

2.2 End Condition for Loop

If a comparison expression with 0 is used as an end condition for a loop, the calculation of the end condition for one loop may become faster. In addition, the number of registers used may decrease.

Before Change	After Change

`int i;`
`int Height;`
`int Width;`
`int *p;`
`int s;`
 	
`p = &array[0][0];`
`s = Height * Width;`
`for (i = 0; i < s; i++)`
`{`
`    *p++ = 0;`
`}`
	`int i;`
	`int Height`
	`int Width`
	`int *p;`
	
	`p = &array[0][0];`
	`for (i = Height * Width; i > 0; i--)`
	`{`
	`    *p++ = 0;`
	`}`

Figure 2.3 C Source Code Example

Remark: array[][] is a global variable.

Before Change:

```
        SUBW    SP, #0x6              2
        MOVW    AX, #0x5              3
        MOVW    [SP+0x04], AX         2
        MOVW    AX, #0xA              3
        MOVW    [SP+0x02], AX         2
        MOVW    AX, #LWRD(_array)     3
        MOVW    DE, AX                1
        MOVW    AX, [SP+0x02]         2
        MOVW    BC, AX                1
        MOVW    AX, [SP+0x04]         2
        MULHU                         3
        MOVW    [SP], AX              2
        MOVW    AX, #0x0              3
        MOVW    HL, AX                1

??main_0:
        MOVW    AX, [SP]              2
        MOVW    BC, AX                1
        MOVW    AX, HL                1
        CALL    N:?SI_CMP_L02         3
        BNC     ??main_1              2

        MOVW    AX, #0x0              3
        MOVW    [DE], AX              1
        XCHW    AX, DE                1
        ADDW    AX, #0x2              3
        XCHW    AX, DE                1
        INCW    HL                    1
        BR      S:??main_0            3

??main_1:
        ADDW    SP, #0x6              2
        RET                           1
```

55 bytes
Execution Time: 41 cycles

After Change:

```
        SUBW    SP, #0x4              2
        MOVW    AX, #0x5              3
        MOVW    [SP+0x02], AX         2
        MOVW    AX, #0xA              3
        MOVW    [SP], AX              2
        MOVW    AX, #LWRD(_array)     3
        MOVW    DE, AX                1
        MOVW    AX, [SP]              2
        MOVW    BC, AX                1
        MOVW    AX, [SP+0x02]         2
        MULHU                         3
        MOVW    HL, AX                1

??main_0:
        MOVW    AX, HL                1
        XOR     A, #0x80              2
        CMPW    AX, #0x8001           3
        BC      ??main_1              2

        MOVW    AX, #0x0              3
        MOVW    [DE], AX              1
        XCHW    AX, DE                1
        ADDW    AX, #0x2              3
        XCHW    AX, DE                1
        DECW    HL                    1
        BR      S:??main_0            3

??main_1:
        ADDW    SP, #0x4              2
        RET                           1
```

48 bytes
Execution Time: 36 cycles

Figure 2.4 Output Assembler

2.3 Optimizing Pointer Variables

Optimizing pointer variables speeds up the calculation.

<table>
<tr><th>Before Change</th><th>After Change</th></tr>
<tr><td><pre>int i;
int *p;

p = array;
for (i = N >> 2; i > 0; i--)
{
      *p++ = 0;
      *p++ = 0;
      *p++ = 0;
      *p++ = 0;
}
for (i = N & 3; i > 0; i--)
{
      *p++ =0;
}</pre></td><td><pre>int i;
int *p;

p = array;
for (i = N >> 2; i > 0; i--)
{
      *(p+0) = 0;
      *(p+1) = 0;
      *(p+2) = 0;
      *(p+3) = 0;
}
for (i = N & 3; i > 0; i--)
{
      *p++ =0;
}</pre></td></tr>
</table>

Figure 2.5 C Source Code Example

Remark: array[] is a global variable and N is 10.

Before Change		After Change	
MOVW AX, #LWRD(_array)	3	MOVW AX, #LWRD(_array)	3
MOVW HL, AX	1	MOVW HL, AX	1
MOVW AX, N:_N	3	MOVW AX, N:_N	3
SARW AX, 0x2	2	SARW AX, 0x2	2
MOVW DE, AX	1	MOVW DE, AX	1
??main_0:		??main_0:	
MOVW AX, DE	1	MOVW AX, DE	1
XOR A, #0x80	2	XOR A, #0x80	2
CMPW AX, #0x8001	3	CMPW AX, #0x8001	3
BC ??main_1	2	BC ??main_1	2
MOVW AX, #0x0	3	MOVW AX, #0x0	3
MOVW [HL], AX	1	MOVW [HL], AX	1
XCHW AX, HL	1	MOVW AX, #0x0	3
ADDW AX, #0x2	3	MOVW [HL+0x02], AX	2
XCHW AX, HL	1	MOVW AX, #0x0	3
MOVW AX, #0x0	3	MOVW [HL+0x04], AX	2
MOVW [HL], AX	1	MOVW AX, #0x0	3
XCHW AX, HL	1	MOVW [HL+0x06], AX	2
ADDW AX, #0x2	3	DECW DE	1
XCHW AX, HL	1	BR S:??main_0	3
MOVW AX, #0x0	3		
MOVW [HL], AX	1	??main_1:	
XCHW AX, HL	1	MOVW AX, N:_N	3
ADDW AX, #0x2	3	AND A, #0x0	2
XCHW AX, HL	1	XCH A, X	1
MOVW AX, #0x0	3	AND A, #0x3	2
MOVW [HL], AX	1	XCH A, X	1
XCHW AX, HL	1	MOVW DE, AX	1
ADDW AX, #0x2	3		
XCHW AX, HL	1	??main_2:	
DECW DE	1	MOVW AX, DE	1
BR S:??main_0	3	XOR A, #0x80	2
		CMPW AX, #0x8001	3
??main_1:		BC ??main_3	2
MOVW AX, N:_N	3		
AND A, #0x0	2	MOVW AX, #0x0	3
XCH A, X	1	MOVW [HL], AX	1
AND A, #0x3	2	XCHW AX, HL	1
XCH A, X	1	ADDW AX, #0x2	3
MOVW DE, AX	1	XCHW AX, HL	1
		DECW DE	1
??main_2:		BR S:??main_2	3
MOVW AX, DE	1		
XOR A, #0x80	2	??main_3:	
CMPW AX, #0x8001	3	RET	1
BC ??main_3	2		
MOVW AX, #0x0	3		
MOVW [HL], AX	1		
XCHW AX, HL	1		
ADDW AX, #0x2	3		
XCHW AX, HL	1		
DECW DE	1		
BR S:??main_2	3		
??main_3:			
RET	1		
90 bytes		73 bytes	
Execution Time: 64 cycles		Execution Time: 52 cycles	

Figure 2.6 Output Assembler

2.4 Faster Execution Using Level of optimization in Integrated Development Environment

The execution speed can be improved by using the appropriate level of optimization in the integrated development environment.

1 Click on [Options…] from the pop-up menu produced by right-clicking on a project in the [Workspace] panel in the integrated development environment IAR Embedded Workbench IDE.

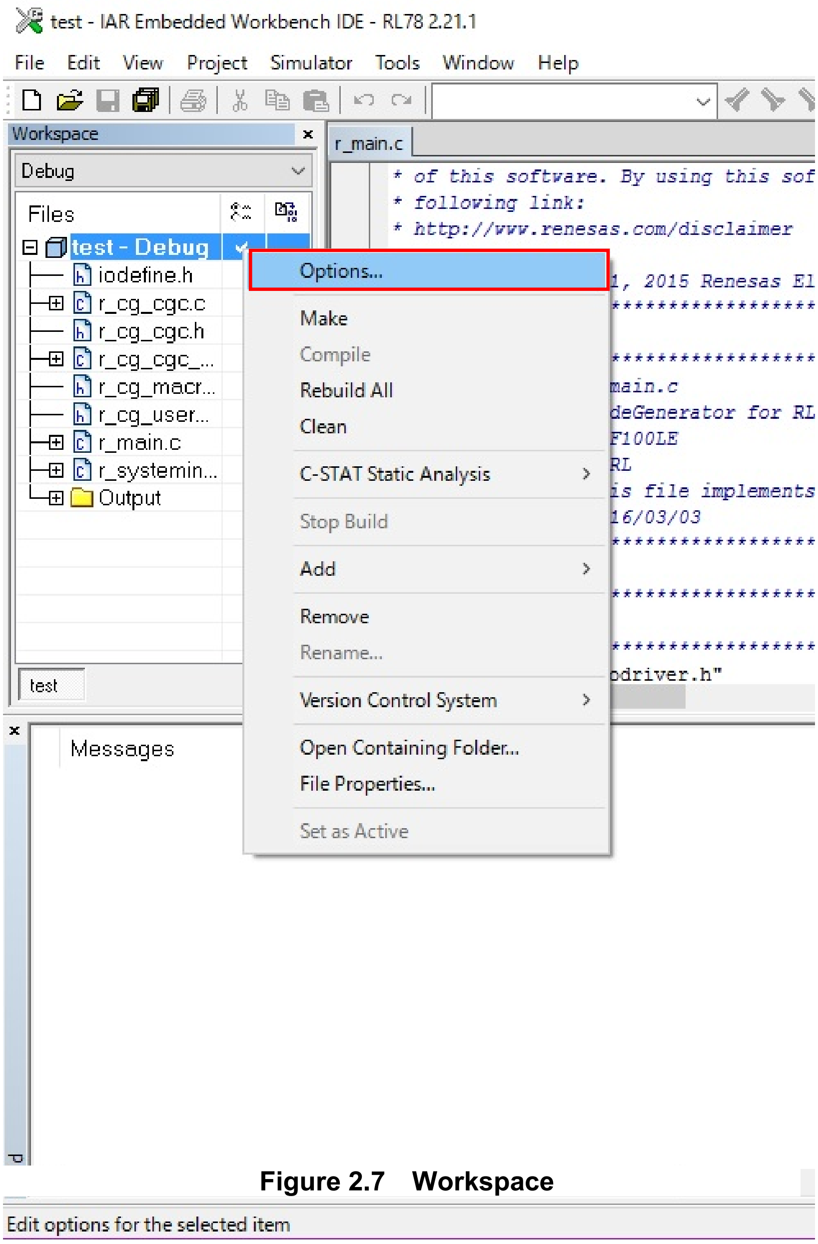

Figure 2.7 Workspace

2 Click on [C/C++ Compiler] -> [Optimizations] in order. Select [High] in [Level] area; select Blanced/Size/Speed.

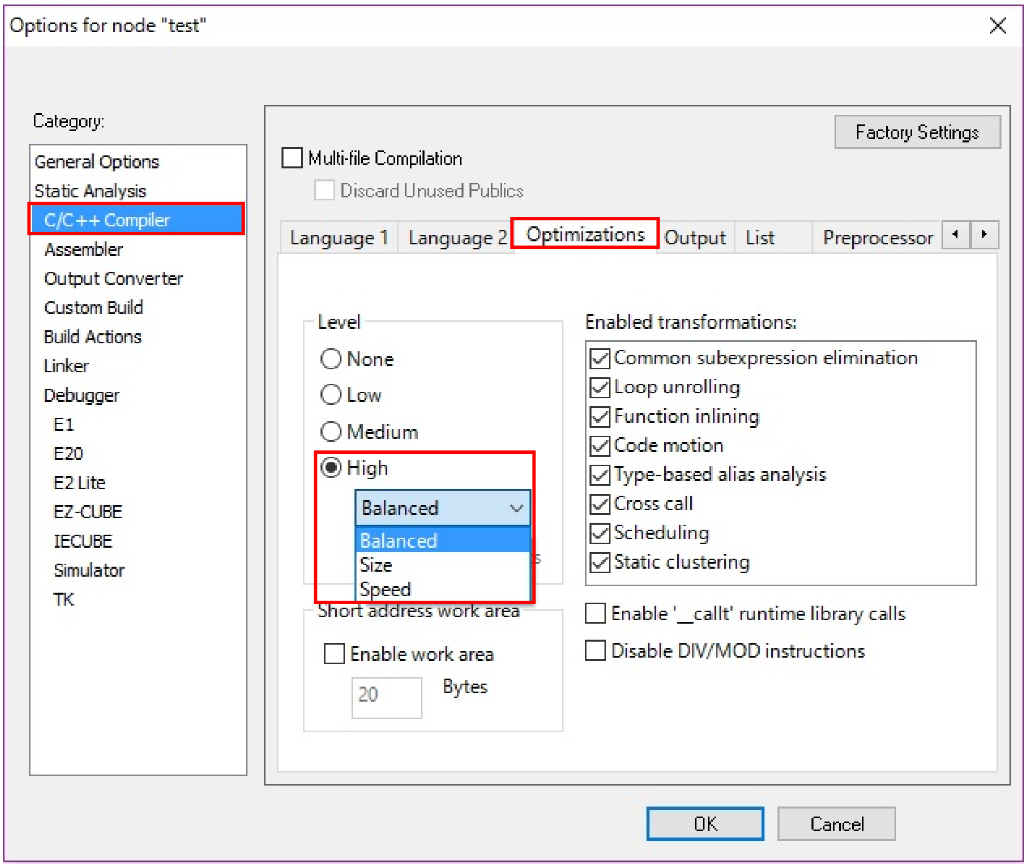

Figure 2.8 Options

3 Execution Example

The following shows the output assembler codes, as reference examples, when the level of optimization in the IAR Embedded Workbench IDE is actually changed for the same source code (Figure 2.9 C Source Code9) and the build is performed.

Source Code Used

```
int i;
int Height;
int Width;
int *p;
int s;

p = &array[0][0];
s = Height * Width;
for (i = 0; i < s; i++)
{
      *p++ = 0;
}
```

Figure 2.9 C Source Code Example

Remark: array[][] is a global variable.

No Optimizations		Speed Precedence / Balance Precedence		Size Precedence	
SUBW SP, #0x6	2	CLRW AX	1	MOVW HL, #LWRD(_array)	3
MOVW AX, #0x5	3	MOVW N:_array, AX	3	MOV A, #0x19	3
MOVW [SP+0x04], AX	2	MOVW HL, #LWRD(_array+2)	3		
MOVW AX, #0x5	3	MOV A, #0x8	3	??main_0:	
MOVW [SP+0x02], AX	2			MOVW DE, #0x0	2
MOVW AX, #LWRD(_array)	3	??main_0:		XCHW AX, DE	1
MOVW DE, AX	1	MOVW DE, #0x0	2	MOVW [HL], AX	2
MOVW AX, [SP+0x02]	2	XCHW AX, DE	1	MOVW AX, DE	2
MOVW BC, AX	1	MOVW [HL], AX	2	INCW HL	1
MOVW AX, [SP+0x04]	2	MOVW [HL+0x02], AX	2	INCW HL	1
MULHU	3	MOVW [HL+0x04], AX	2	DEC A	1
MOVW [SP], AX	2	MOVW AX, DE	2	BNZ ??main_0	2
MOVW AX, #0x0	3	XCHW AX, HL	2		
MOVW HL, AX	1	ADDW AX, #0x6	3	RET	1
		XCHW AX, HL	2		
??main_0:		DEC A	1		
MOVW AX, [SP]	2	BNZ ??main_0	2		
MOVW BC, AX	1				
MOVW AX, HL	1	RET	1		
CALL N:?SI_CMP_L02	2				
BNC ??main_1	2				
MOVW AX, #0x0	3				
MOVW [DE], AX	2				
XCHW AX, DE	2				
ADDW AX, #0x2	3				
XCHW AX, DE	2				
INCW HL	1				
BR S:??main_0	2				
??main_1:					
ADDW SP, #0x6	2				
RET	1				
56 bytes		30 bytes		19 bytes	
Execution Time: 41 cycles		Execution Time: 24 cycles		Execution Time: 19 cycles	

Figure 2.10 Output Assembler

3. Programming Techniques to Avoid Bugs

3.1 Writing Conditional Expression's Value on Left Side of Operator

It is not recommended to write a variable on the left side of the operator in the conditional expression as in Figure 3.1.

```
#define VAL_OK 1

if (ret == VAL_OK)
{
      sub();
}
```

Figure 3.1 Bad Description Example (1)

This is because a description mistake may be overlooked. As in Figure 3.2, even if the assignment operator (=) is placed instead of the equality operator (==), a compile error does not occur during compilation (a warning occurs) and an execution file is generated.

```
#define VAL_OK 1

if (ret = VAL_OK)
{
      sub();
}
```

Figure 3.2 Bad Description Example (2)

In order to avoid a case like the above, it is recommended to write a value for the conditional expression on the left side of the operator.

```
#define VAL_OK 1

if (VAL_OK == ret)
{
      sub();
}
```

Figure 3.3 Good Description Example

If written as in Figure 3.3, a change from the equality operator (==) to the assignment operator (=) can be noticed as a programming mistake because a compile error is produced during compilation.

3.2 Magic Number

It is recommended to define a constant that has meaning as a macro and use it and not to use a magic number (immediate value). The meaning of a constant can be clearly indicated by defining it as a macro. Especially, when changing a constant used in multiple locations, only one macro needs to be changed. This prevents a mistake from happening in advance.

Before Change	After Change

if (8 == cnt)
{
    cnt++;
}
	#define CNTMAX 8
	if (CNTMAX == cnt)
	{
	    cnt++:
	}

Figure 3.4 C Source Code Example

3.3 Calculation That Might Cause Information Loss

Attention is required when variables of different types are calculated. A variable value may be changed (information might be lost). When intentionally assigning a value to a different type, write a type conversion to explicitly indicate that intention.

If the calculation results in a value outside the value range that can be expressed in the type, an unintended value may be produced. It is recommended to confirm before the calculation that the calculation result is within the value range that can be expressed in the type. Or convert to a type that can handle a bigger value before the calculation.

Before Change	After Change

/* Assignment example */
short s;
long l;
 	
void main(void)
{
    s = l;
    s = s + 1;
}
 	
/* Calculation example */
unsigned int n;
unsigned int m;
 	
n = 0x8000;
m = 0x8000;
 	
if (0xffff < (n + m))
{
    …
}
	/* Assignment example */
	short s;
	long l;
	
	void main(void)
	{
	    s = (short)l;
	    s = (short)(s + 1);
	}
	
	/* Calculation example */
	unsigned int n
	unsigned int m;
	
	n = 0x8000;
	m = 0x8000;
	
	if (0xffff < ((long)n + m))
	{
	    …
	}

Figure 3.5 C Source Code Example

3.4 Type Conversion to Remove const and volatile

Because the areas modified by const or volatile are the areas that are only referenced and are not allowed to be optimized, attention is required when accessing these areas. If a cast is performed on pointer variables pointing to these areas to remove const or volatile, the compiler may not be able to check a program for erroneous descriptions, or unintended optimization may be performed.

Before Change	After Change

void sub(char *);
const char *p;
void main(void)
{
    sub((char*)p);
    …
}
	void sub(char *);
	const char *p;
	void main(void)
	{
	    sub(p);
	    …
	}

Figure 3.6 C Source Code Example

3.5 Prohibition of Recursive Call

A function is not allowed to call the function itself irrespective of whether it is direct or indirect (prohibition of recursive call). Because the stack size to be used during execution cannot be predicted for a recursive call, it may cause a stack overflow.

```
unsigned int calc(unsigned int n)
{
    if (1 >= n)
    {
        return (1);
    }
    else
    {
        return (n * calc(n-1));
    }
}
```

Figure 3.7 Bad Description Example

3.6 Localizing Access Range and Related Data

1 .Declare a variable accessed from multiple functions in the same file as a static variable.

The fewer the number of global functions is, the more the readability in understanding the entire program improves. Include a static specifier so that the number of global functions do not increase unnecessarily.

Before Change	After Change

int n;
void func1(void)
{
    …
    n = 0;
    …
}
 	
void func2(void)
{
    if (0 == n)
    {
        n++;
    }
    …
}
	static int n;
	void func1(void)
	{
	    …
	    n = 0;
	    …
	}
	
	void func2(void)
	{
	    if (0 == n)
	    {
	        n++;
	    }
	    …
	}

Figure 3.8 C Source Code Example

Remark: n is a variable that cannot be accessed from other files.

2 .If a function is referenced only by a function defined in the same file, write it as a static function.

The fewer the number of global functions is, the more the readability in understanding the entire program improves. Include a static specifier so that the number of global functions do not increase unnecessarily.

Before Change	After Change

void sub(void)
{
    …
    …
}
 	
void main(void)
{
    …
    sub();
    …
}
	static void sub(void)
	{
	    …
	    …
	}
	
	void main(void)
	{
	    …
	    sub();
	    …
	}

Figure 3.9 C Source Code Example

Remark: sub is a function that is not called by other files.

3 .When defining a related constant, use enum rather than #define.

If each related constant is defined with the enum type, an undefined usage can be checked by a compiler or other software.

The macro name defined by #define is expanded as a macro and is not turned into a name processed by a compiler. On the other hand, the enum constant defined by the enum declaration is tuned into a name processed by a compiler. The name processed by a compiler can be referenced during debugging, which makes debugging easier.

Before Change	After Change

#define JANUALY 0
#define FEBRUALY 1
#define SUNDAY 0
#define MONDAY 1
int month;
int day;
 	
…
if (JANUALY == month)
{
    …
    if (MONDAY == day)
    {
        …
        …
    }
}
 	
if (SUNDAY == month)   ← Does not cause an error
{
    …
    …
}
	typedef enum
	{
	    JANUALY, FEBRUALY, …
	} month;
	typedef enum
	{
	    SUNDAY, MONDAY, …
	} day;
	…
	if (JANUALY == month)
	{
	    …
	    if (MONDAY == day)
	    {
	        …
	        …
	    }
	}
	
	if (SUNDAY == month) ← Causes an error
	{
	    …
	    …
	}

Figure 3.10 C Source Code Example

3.7 Exception Processing of Branch Condition

1 .Place the else clause at the end of the if-else if statement. Especially when a condition of else does not usually occur, include the exception processing or a comment that was predefined by a project in the else clause.

If the else clause is not included in the if-else statement, it is not clear whether inclusion of the else clause is forgotten or whether the else clause does not occur. Even when it is known beforehand that an else condition does not occur, the program operation when an unexpected condition occurs can be predicted by including the else clause.

Before Change	After Change

if (0 == var)
{
    …
}
else if (0 < var)
{
    …
}
	if (0 == var)
	{
	    …
	}
	else if (0 < var)
	{
	    …
	}
	else
	{
	    /* Description of exception processing or comment */
	}

Figure 3.11 C Source Code Example

2 .Place the default clause at the end of the switch statement. Especially when a default condition does not usually occur, include the exception processing or a comment that was predefined by a project in the default clause.

If the default clause is not included in the switch statement, it is not clear whether inclusion of the default clause is forgotten or whether the default clause does not occur. Even when it is known beforehand that a default condition does not occur, the program operation when an unexpected condition occurs can be predicted by including the default clause.

<table>
<tr><th>Before Change</th><th>After Change</th></tr>
<tr><td>

```
switch (var)
{
    case 0:
        …
        break;

    case 1:
        …
        break;
}
```

</td><td>

```
switch (var)
{
    case 0:
        …
        break;

    case 1:
        …
        break;

    default:
        /* Description of exception processing or comment */
        break;
}
```

</td></tr>
</table>

Figure 3.12 C Source Code Example

3 .Do not use an equality operator (==) or an inequality operator (!=) for comparing a loop counter.

If the value of a loop counter change is not 1, an infinite loop may be entered.

<table>
<tr><th>Before Change</th><th>After Change</th></tr>
<tr><td>

```
void main(void)
{
    int i = 0;
    for (i = 0; i != 11; i += 2)
    {
        …
        …
    }
}
```

</td><td>

```
void main(void)
{
    int i = 0;
    for (i = 0; i < 11; i += 2)
    {
        …
        …
    }
}
```

</td></tr>
</table>

Figure 3.13 C Source Code Example

3.8 Consideration for Special Description

1 .If intentionally writing statements that do nothing, use a comment or an empty macro to indicate the intention.

Before Change	After Change

``for (;;)
{
}
 	
i = CNT;
while (0 < (--i));```
	``/* Example of using comment */
	for (;;)
	{
	    /* Interrupt wait time */
	}
	
	/* Example of using empty macro */
	#define NO_OPERATION
	i = CNT;
	while (0 < (--i))
	{
	    NO_OPERATION;
	}```

Figure 3.14 C Source Code Example

2 .Specify how to write an infinite loop.

Specify how to write an infinite loop and make the writing style consistent.

Example:

- Make an infinite loop consistent using for (;;).
- Make an infinite loop consistent using while (1).
- Make an infinite loop consistent using do … while (1).
- Use an infinite loop written as a macro.

Including infinite loops written in different writing styles in the same project may deteriorate maintainability.

3.9 Deleting Unused Description

1 .Do not define a function, variable, argument, typedef, label, macro, etc. that are not used.

As it is difficult to determine whether the definition of an unused function (such as variable/argument/label) is a description error or not, maintainability is reduced.

Before Change	After Change

void main(int n)
{
    /* n is not used in the main function */
    …
}
	void main(void)
	{
	    …
	}

Figure 3.15 C Source Code Example

2 .Do not comment out code.

Avoid use of invalid code if possible because the code readability is lost.
However, if code needs to be disabled for debugging, etc., write code according to the predefined rule (such as enclosing with #if 0) instead of commenting out.

Before Change	After Change
… // i++ …	… #if 0 /* Temporarily disabled for debugging */ i++ #endif …

Figure 3.16 C Source Code Example

4. Sample Code

Sample code can be downloaded from the Renesas Electronics website.

5. Reference Documents

RL78 family User's Manual: Software (R01US0015E)

RL78 Compiler CC-RL User's Manual (R20UT3123E)

CC-RL C Compiler for RL78 Family Coding Techniques (R02UT3569E)

(The latest information can be downloaded from the Renesas Electronics website.)

Website and Support

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

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

Revision Record	RL78 Family C/C++ Compiler for Renesas RL78 Programming Technique

Rev.	Date	Description	
		Page	Summary
1.00	Apr. 14, 2017	—	First edition issued.

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Handling of Unused Pins

   Handle unused pins in accordance with the directions given under Handling of Unused Pins in the manual.

   ¾ The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible. Unused pins should be handled as described under Handling of Unused Pins in the manual.

2. Processing at Power-on

   The state of the product is undefined at the moment when power is supplied.

   ¾ The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the moment when power is supplied.
   In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the moment when power is supplied until the reset process is completed.
   In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the moment when power is supplied until the power reaches the level at which resetting has been specified.

3. Prohibition of Access to Reserved Addresses

   Access to reserved addresses is prohibited.

   ¾ The reserved addresses are provided for the possible future expansion of functions. Do not access these addresses; the correct operation of LSI is not guaranteed if they are accessed.

4. Clock Signals

   After applying a reset, only release the reset line after the operating clock signal has become stable. When switching the clock signal during program execution, wait until the target clock signal has stabilized.

   ¾ When the clock signal is generated with an external resonator (or from an external oscillator) during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Moreover, when switching to a clock signal produced with an external resonator (or by an external oscillator) while program execution is in progress, wait until the target clock signal is stable.

5. Differences between Products

   Before changing from one product to another, i.e. to a product with a different part number, confirm that the change will not lead to problems.

   ¾ The characteristics of Microprocessing unit or Microcontroller unit products in the same group but having a different part number may differ in terms of the internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

SALES OFFICES Renesas Electronics Corporation http://www.renesas.com

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

Renesas Electronics America Inc.
2801 Scott Boulevard Santa Clara, CA 95050-2549, U.S.A.
Tel: +1-408-588-6000, Fax: +1-408-588-6130

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: +44-1628-585-100, Fax: +44-1628-585-900

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

Renesas Electronics (China) Co., Ltd.
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, P. R. China 200333
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 II Stage, Indiranagar, Bangalore, India
Tel: +91-80-67208700, Fax: +91-80-67208777

Renesas Electronics Korea Co., Ltd.
12F., 234 Teheran-ro, Gangnam-Gu, Seoul, 135-080, Korea
Tel: +82-2-558-3737, Fax: +82-2-558-5141

Colophon 6.0