CC-RL C COMPILER FOR RL78 FAMILY CODING TECHNIQUES

CC-RL V.1.02.00

Dec. 24, 2015 R20UT3569EJ0100

Microcomputer Tool Product Marketing Department,
Tool Business Division
Renesas System Design Co., Ltd.

RENESAS

AGENDA

▪ Introduction	**Page 3**
▪ Coding Techniques	**Page 4**
▪ Memory Models	**Page 16**
▪ Using Variables/Functions Information File	**Page 19**

Introduction

- This document describes coding techniques to further reduce the code size or accelerate execution even after optimization through option settings when using the CC-RL C compiler.
- Each amount of code reduction shown in this document only applies to the corresponding example; the actual reduction will vary slightly between cases.
- The output assembly-language codes shown in this document are examples compiled with the medium model and the code size precedence option (-Osize) specified. Note that the output code will differ when a different type of optimization (default optimization or speed precedence optimization) is specified.

- This document uses the following tools and versions for description.
  - CC-RL C compiler for the RL78 family V.1.02.00
  - e2 studio integrated development environment V4.2.0.012
  - CS+ integrated development environment V.3.03.00

Coding Techniques

Effects of Coding Techniques

Effects on the output code size and execution speed when applying coding techniques

Coding Technique	Code Size	Execution Speed
Size of variables	✓	✓
Unsigned variables	✓	✓
saddr area	✓	✓
callt function	✓	X
Alignment of structure variables	✓	△
Bit fields and 1-byte variables	✓	✓

✓: Effective; △: Not effective; X: Performance degraded

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.
- Example:
  - C source program

Before Change	After Change
void func(void)	void func(void)
{	{
signed int i;	**signed char i;**
for(i=0; i<10; i++)	for(i=0; i<10; i++)
__nop();	__nop();
}	}

  - Output assembly-language program

Before Change		After Change	
movw ax, #0x000A	3	**mov a, #0x0A**	2
.BB@LABEL@1_1:		.BB@LABEL@1_1:	
nop	1	nop	1
addw ax, #0xFFFF	3	**dec a**	1
bnz $.BB@LABEL@1_1	2	bnz $.BB@LABEL@1_1	2
ret	1	ret	1
	10 bytes		7 bytes

Unsigned Variables

- Add "unsigned" for all data that never handle negative values.
- This is because the RL78 devices excel in handling unsigned variables.
- Example:
  - C source program

Before Change	After Change
signed int data0,data1;	**unsigned** int data0,data1;
if(data0 > 10) data1++;	if(data0 > 10) data1++;

  - Output assembly-language program

Before Change		After Change	
movw ax, !LOWW(_data0)	3	movw ax, !LOWW(_data0)	3
xor a, #0x80	2		
cmpw ax, #0x800B	3	cmpw ax, #0x000B	3
skc	2	skc	2
incw !LOWW(_data1)	3	incw !LOWW(_data1)	3
	13 bytes		11 bytes

saddr Area (1/2)

- 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.

saddr Area (2/2)

- Example:
  - C source program

Before Change	After Change

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;
 	
data0.b4 = data1.b1;
	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;
	
	data0.b4 = data1.b1;

  - Output assembly-language program

Before Change			After Change		
movw	**hl,#LOWW (_data1)**	3			
mov1	CY,[hl].1	2	mov1	CY,_data1.1	3
movw	**hl,#LOWW (_data0)**	3			
mov1	[hl].4,CY	2	mov1	_data0.4,CY	3
		10 bytes			6 bytes

callt Function (1/2)

- 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 that for direct function calls.
- Example:
  - C source program

Before Change	After Change

void func_sub(void)
{
;
}
void func()
{
func_sub();
;
func_sub();
}
	__callt void func_sub(void)
	{
	;
	}
	void func()
	{
	func_sub();
	;
	func_sub();
	}

callt Function (2/2)

- Example:
  - Output assembly-language program

Before Change		After Change	
		.SECTION .callt0,CALLT0	
		@_func_sub:	
		.DB2 _func_sub	2
.SECTION .textf,TEXTF		.SECTION .textf,TEXTF	
_func:		_func:	
call !!_func_sub	4	**callt [@_func_sub]**	2
call !!_func_sub	4	**callt [@_func_sub]**	2
	8 bytes		6 bytes

- Notes:
  - A table of addresses for function calls is generated (.callt0).
  - Due to generation of this table, code size reduction is not effective for a function called only once.
  - The CALLT instruction requires more clock cycles for execution than the CALL instruction.
  - Alternatively, the variables/functions information file can be used to specify declarations of the functions to be called through the CALLT instruction

Alignment of Structure Members (1/2)

- In the RL78 family 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-byte 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.
- Example:
  - C source program

Before Change

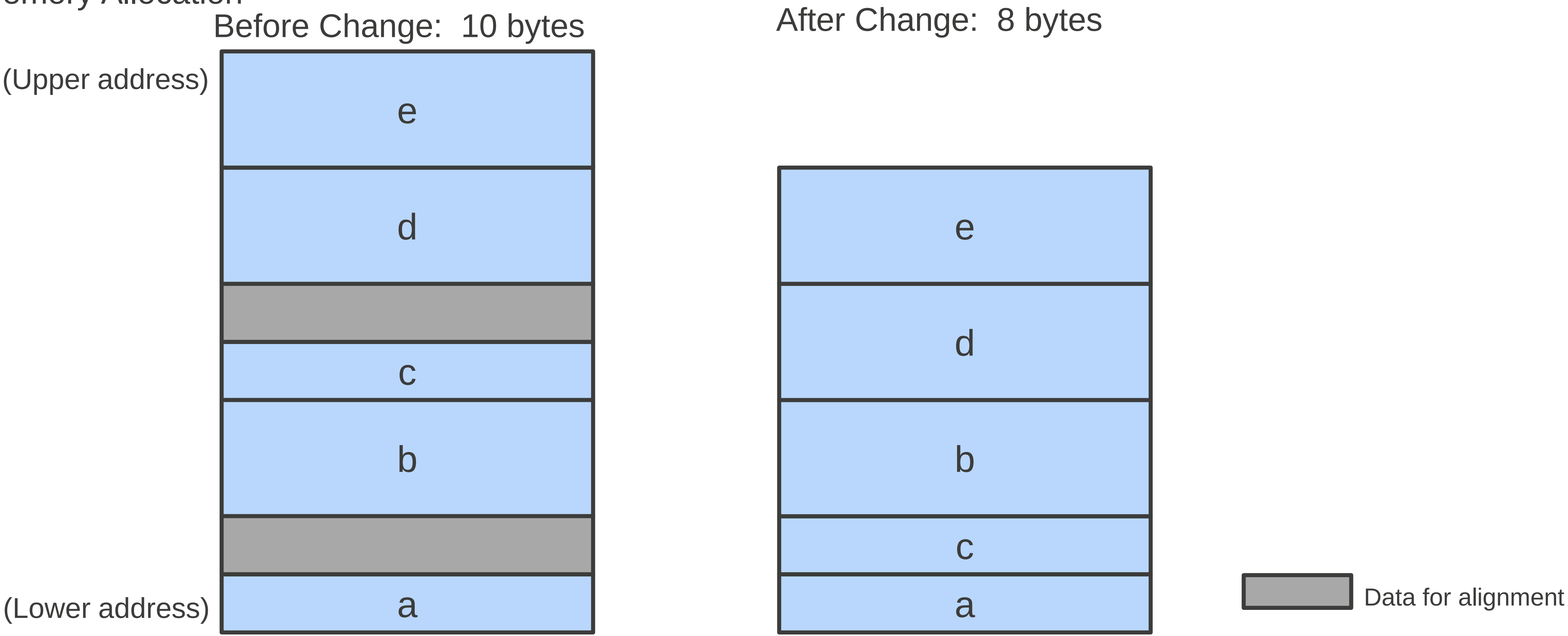
```
struct {
        signed char a;
        signed int b;
        signed char c;
        struct {
                signed int d;
                signed int e;
        } f;
} data;
```

After Change

```
struct {
        signed char a;
        signed char c;
        signed int b;
        struct {
                signed int d;
                signed int e;
        } f;
} data;
```

Alignment of Structure Members (2/2)

- Example:
  - Memory Allocation



Before Change: 10 bytes

(Upper address)

e

d

c

b

a

(Lower address)

After Change: 8 bytes

e

d

b

c

a

▨ Data for alignment

Bit Fields and 1-Byte Variable (1/2)

- When the size of a 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.
- Example:
  - C source program

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

Bit Fields and 1-Byte Variable (2/2)

- Example:
  - Output assembly-language program

Before Change		After Change	
mov a, #0x06	2	**cmp0 !LOWW(_data)**	3
and a, !LOWW(_data)	3		
sknz	2	sknz	2
ret	1	ret	1
inc !LOWW(_dummy)	3	inc !LOWW(_dummy)	3
ret	1	ret	1
	12 bytes		10 bytes

Memory Models

Memory Models (1/2)

- According to the specifications of the RL78 family, the sizes of the codes for function call and data access differ depending on whether
  - the program size is 64 Kbytes or larger
  - the data size (including ROM data) is 64 Kbytes or larger.

- CC-RL provides the following two memory models.

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

Memory Models (2/2)

- For a large program, select the medium model and add the __near qualifier to frequently called functions to reduce the code size.
- Note that when the __near or __far qualifier is added to a function, the type of the pointer variable that handles the qualified function should also be modified to match the type of the function.

Using Variables/Functions Information File

Using Variables/Functions Information File (1/3)

- Features
  - Frequently used variables are allocated to the saddr area.
  - Frequently called functions are handled as callt functions.
  - In addition to the qualifiers (__saddr and __callt) and #pragma declarations (saddr and callt) specified in the source files, the variables specified in the variables/functions information file are allocated to the saddr area and the functions specified in the file are handled as callt functions.
- How to use
  - Specify the –vfinfo linker option to generate a variables/functions information file.
  - Include the variables/functions information file at compilation in either of the following methods.
    - Specify the file through the –preinclude compiler option.
    - Use #include to include the file to each source file.

Using Variables/Functions Information File (2/3)

- Note
  - When generating a variables/functions information file through the –vfinfo linker option, check that the build process has been completed correctly and a load module file has been created.
- Linker option -vfinfo
  - This option selects variables and functions for which code reduction works most effectively based on their reference frequencies, adds declarations of saddr variables and callt functions through #pragma directives to the selected variables and functions, and outputs them to a header file (variables/functions information file).

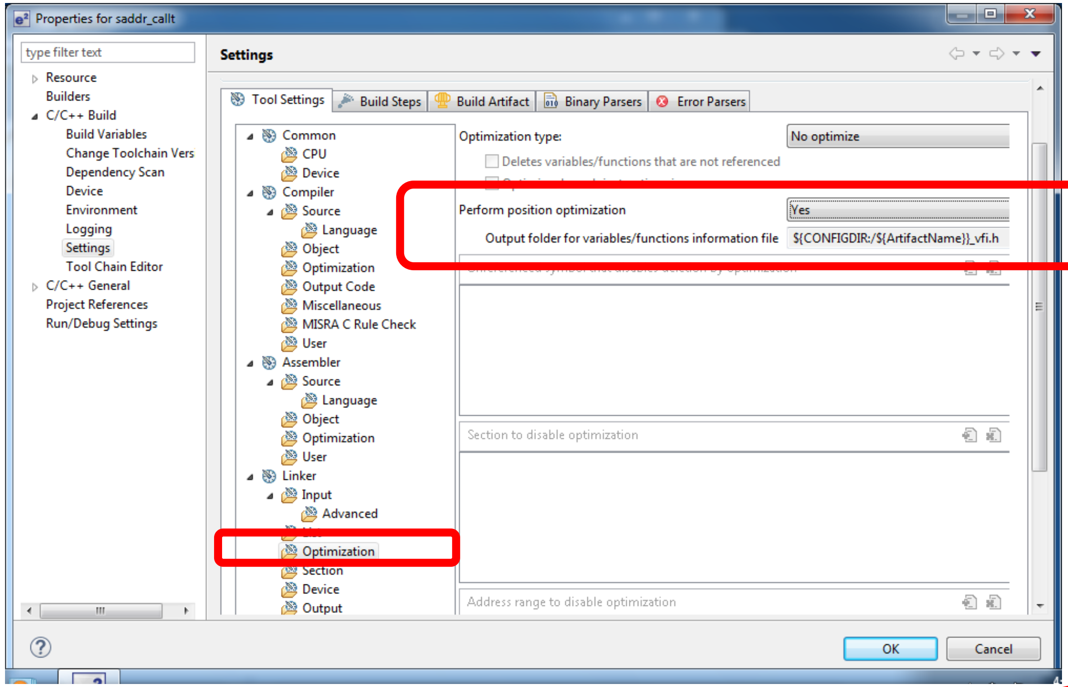
Using Variables/Functions Information File (3/3)

- Example:

```
/* RENESAS OPTIMIZING LINKER GENERATED FILE yyyy.mm.dd */
/*** variable information ***/
#pragma saddr data0 /* count:10,size:1,near,tp0.obj */
#pragma saddr data1 /* count:5,size:1,near,tp0.obj */
                    :
/* #pragma saddr datann */ /* count:1,size:1,near,tp1.obj */
                    :
/*** function information ***/
#pragma callt func_sub0 /* count:4,far,tp0.obj */
#pragma callt func_sub1 /* count:1,far,tp0.obj */
                    :
/* #pragma callt func0 */ /* count:1,far,tp1.obj */
                    :
```

Using Variables/Functions Information File (e2 studio) (1/2)

- Generating a variables/functions information file automatically
  - Enable position optimization in the linker.



Optimization type: No optimize

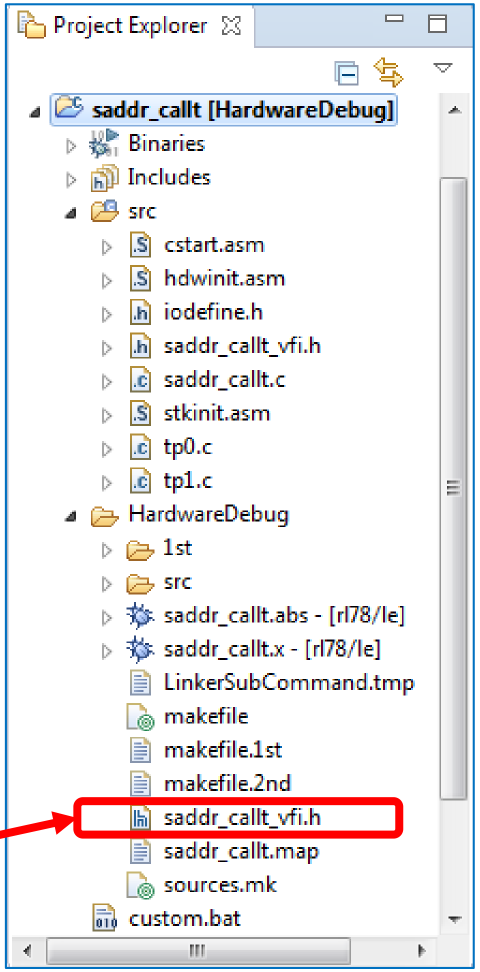
□ Deletes variables/functions that are not referenced

Perform position optimization: Yes

Output folder for variables/functions information file: ${CONFIGDIR:/${ArtifactName}}_vfi.h

Section to disable optimization

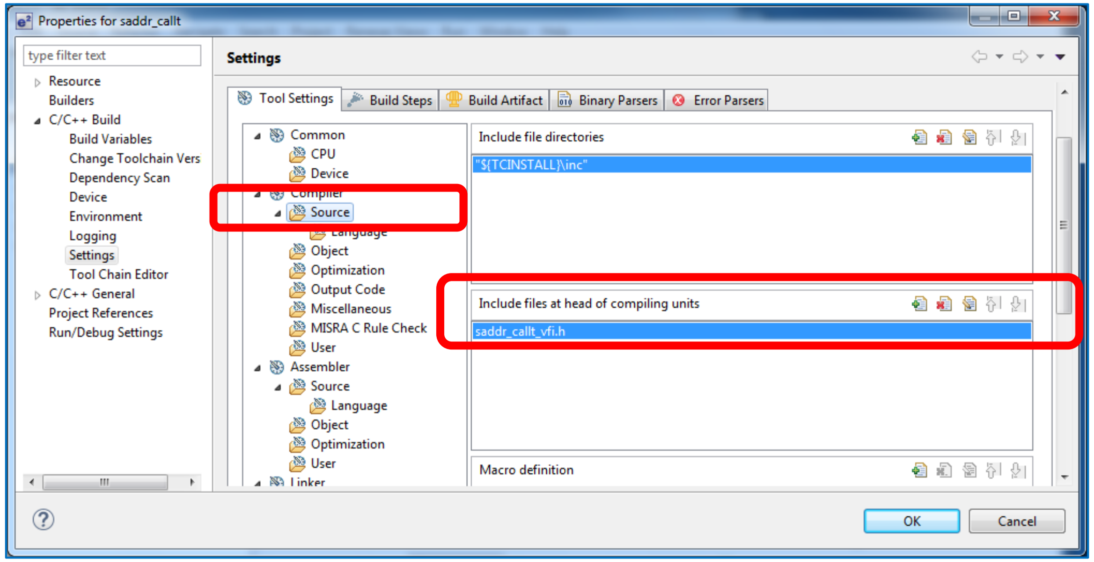
Address range to disable optimization



  - Project name.h" is registered in the project tree.

Using Variables/Functions Information File (e2 studio) (2/2)

- Editing a variables/functions information file (after automatic generation)
  - Disable position optimization that was enabled in the step shown in the previous page in the linker.
  - Import the automatically generated "Project name.h" file to the src folder.
  - Register the "Project name.h" file in [Include files at head of compiling units].



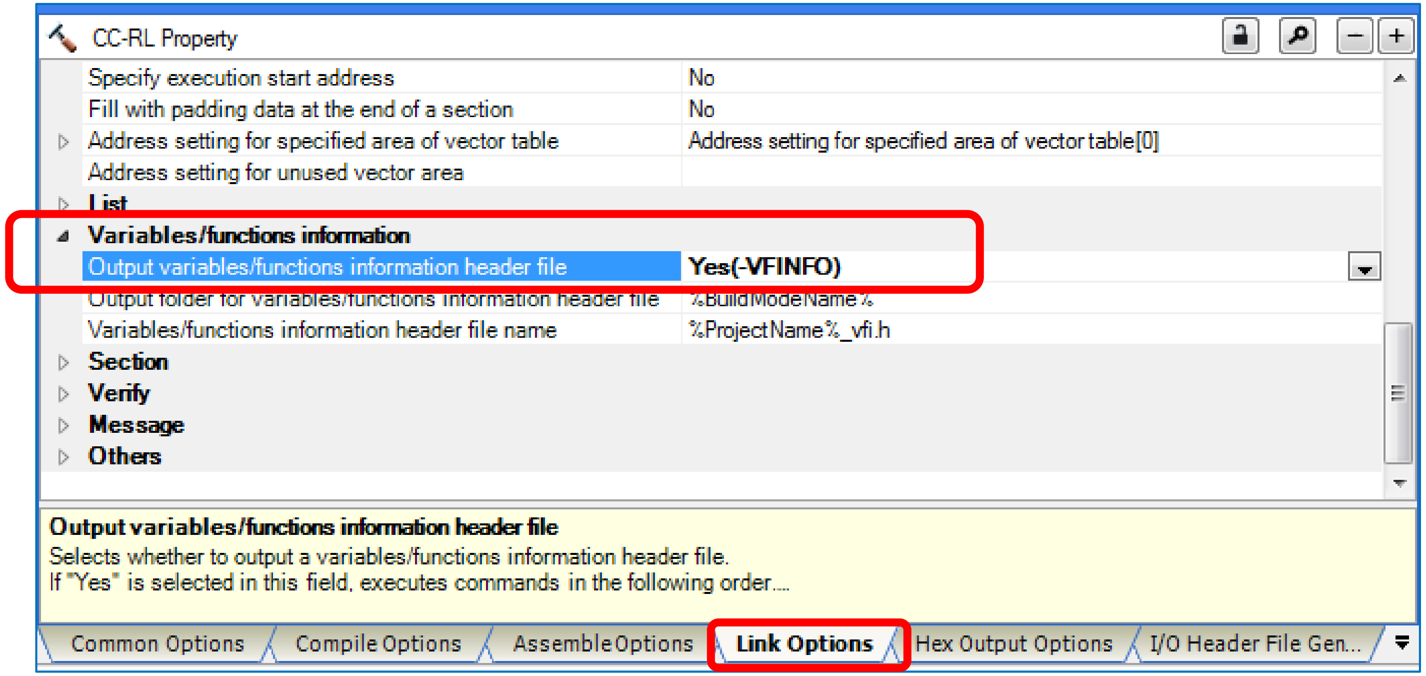
Include file directories
"${TCINSTALL}\inc"

Include files at head of compiling units
saddr_callt_vfi.h

Macro definition

Using Variables/Functions Information File (CS+) (1/2)

- Generating a variables/functions information file automatically
  - Enable output of a variables/functions information file.

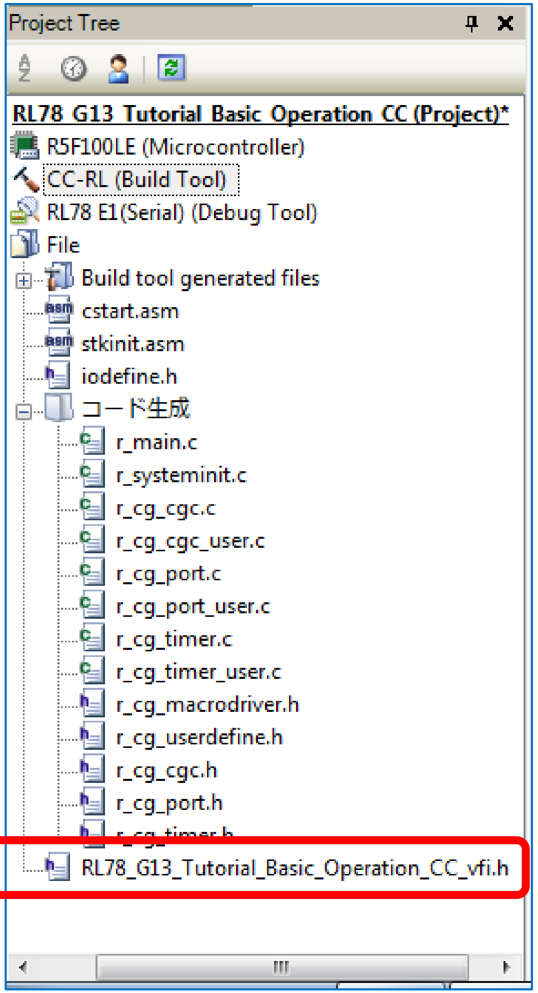


CC-RL Property

Specify execution start address	No
Fill with padding data at the end of a section	No
▷ Address setting for specified area of vector table	Address setting for specified area of vector table[0]
Address setting for unused vector area	
▷ **List**	
◢ **Variables/functions information**	
Output variables/functions information header file	**Yes(-VFINFO)**
Output folder for variables/functions information header file	%BuildModeName%
Variables/functions information header file name	%ProjectName%_vfi.h
▷ **Section**	
▷ **Verify**	
▷ **Message**	
▷ **Others**	

Output variables/functions information header file
Selects whether to output a variables/functions information header file.
If "Yes" is selected in this field, executes commands in the following order....

  - "Project name.h" is registered in the project tree.

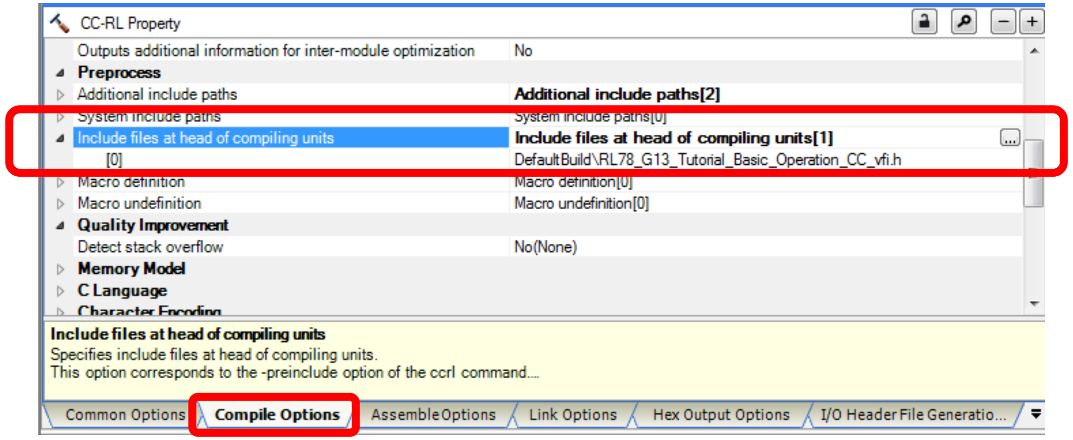


Project Tree

- RL78_G13_Tutorial_Basic_Operation_CC (Project)*
  - R5F100LE (Microcontroller)
  - CC-RL (Build Tool)
  - RL78 E1(Serial) (Debug Tool)
  - File
    - Build tool generated files
    - cstart.asm
    - stkinit.asm
    - iodefine.h
    - コード生成
      - r_main.c
      - r_systeminit.c
      - r_cg_cgc.c
      - r_cg_cgc_user.c
      - r_cg_port.c
      - r_cg_port_user.c
      - r_cg_timer.c
      - r_cg_timer_user.c
      - r_cg_macrodriver.h
      - r_cg_userdefine.h
      - r_cg_cgc.h
      - r_cg_port.h
      - r_cg_timer.h
    - RL78_G13_Tutorial_Basic_Operation_CC_vfi.h

Using Variables/Functions Information File (CS+) (2/2)

- Editing a variables/functions information file (after automatic generation)
  - Disable output of a variables/functions information file that was enabled in the step shown in the previous page.
  - Copy the "Project name.h" file to another folder (such as the source folder). (Although it can be used without copying, when output of a variables/functions information file is enabled, the tool overwrites and deletes the file.)
  - Register the "Project name.h" file in [Include files at head of compiling units].



Outputs additional information for inter-module optimization	No
Preprocess	
Additional include paths	**Additional include paths[2]**
System include paths	System include paths[0]
Include files at head of compiling units	**Include files at head of compiling units[1]**
[0]	DefaultBuild\RL78_G13_Tutorial_Basic_Operation_CC_vfi.h
Macro definition	Macro definition[0]
Macro undefinition	Macro undefinition[0]
Quality Improvement	
Detect stack overflow	No(None)
Memory Model	
C Language	
Character Encoding	

Include files at head of compiling units
Specifies include files at head of compiling units.
This option corresponds to the -preinclude option of the ccrl command....

Renesas System Design Co., Ltd.