SuperH RISC engine C/C++ Compiler Package

APPLICATION NOTE: [Reference] Additional functions

This document explains features added to each version of the SuperH RISC engine C/C++ compiler, as well as provides notes on upgrading from an earlier version.

A.1 Features Added between Ver. 1.0 and Ver. 2.0 2

A.2 Features Added between Ver. 2.0 and Ver. 3.0 3

A.3 Features Added between Ver. 3.0 and Ver. 4.1 6

A.4 Features Added between Ver. 4.1 and Ver. 5.0 9

A.5 Features Added between Ver. 5.0 and Ver. 5.1 11

A.6 Features Added between Ver. 5.1 and Ver. 6.0 13

A.7 Features Added between Ver. 6.0 and Ver. 7.0 15

A.8 Features Added between Ver. 7.0 and Ver. 7.1 28

A.9 Features Added between Ver. 7.1 and Ver. 8.0 40

A.10 Features Added between Ver. 8.0 and Ver. 9.0 41

A.11 Features Added between Ver. 9.0 and Ver. 9.1 43

A.12 Features Added between Ver. 9.1 and Ver. 9.2 52

B. Notes on Version Upgrade 74
B.1 Guaranteed Program Operation 74
B.2 Compatibility with Earlier Version 75

Website and Support <website and support,ws> 76

Revision Record <revision history,rh> 76

A.1 Features Added between Ver. 1.0 and Ver. 2.0

Table A.1 summarizes the features added to version 2.0 of the SHC compiler.

Table A.1 Summary of Features Added to Version 2.0 of the SHC Compiler

No.	Feature	Description
1	Support for SH7600 Series	In addition to the SH7000 Series, objects can be created which use instructions for the SH7600 Series as well.
2	Position-independent code	SH7600 Series objects can be created with program sections assigned to arbitrary addresses.
3	Specification of output area for character strings	An option can be used to select whether to place character string data in a constant section (ROM) or in a data section (RAM).
4	Comment nesting	An option is supported to specify whether comments are nested or not.
5	Optimize for speed or for size	An option is provided to specify whether to optimize for speed or for size at time of object creation.
6	Support for section name switching	By using #pragma instructions midway through a program, object output section names can be switched.
7	mac embedded function	An embedded function is supported for performing multiply-and-accumulate operations on two arrays using the MAC instruction.
8	Embedded functions for system calls	Embedded functions are supported for making direct system calls to the ITRON-specification OS HI-SH7.
9	Single-precision elementary function library	A single-precision elementary function library is supported.
10	char-type bit fields	char-type bit fields are supported.

A.2 Features Added between Ver. 2.0 and Ver. 3.0

Table A.2 summarizes the features added to version 3.0 of the SHC compiler.

Table A.2 Summary of Features Added to Version 3.0 of the SHC Compiler

No.	Feature	Description
1	Strengthened optimization	Optimization performance was greatly enhanced. Also, provisions were made for selective use of the option to optimize for speed or for size.
2	SH-3 support	An option was provided for generating objects for the SH-3, and the little-endian format characteristic of the SH-3 was also supported. Also, an SH-3 data prefetch instruction was supported as an embedded function.
3	Extension of compiler limits	The number of files that can be compiled at once, the maximum nesting levels for include files, and other compiler limits were extended.
4	Support for Japanese character codes in character strings	Provisions were added for character string data containing shift-JIS and EUC Japanese character codes.
5	Specification of options using files	Files can be used to specify command line options.
6	Utilization of the SH-2 divider	Division operation code is generated which makes use of the SH-2 divider.
7	Inline expansion	Specifications can be added for inline expansion of user routines written in C and assembly languages.
8	Use of short address specifications	Variables can be specified for short addressing, including two-byte addresses and GBR-relative data.
9	Control of register save/restore operations	Statements can be added to suppress register save/restore operations, to improve function speed and size.

(1) Strengthened optimization

Optimization in ver. 3.0 provides options for emphasizing speed (the -SPEED option) and size (the -SIZE option), and both types of optimization have been reinforced.

To enhance speed, loop optimization has been improved and inline expansion employed to improve execution speed by about 10%, achieving an execution speed of 1 MIPS/MHz.

In order to reduce program size, instructions which shrink code size are generated and overlapping processing is combined for significant improvements, to cut object size by approximately 20%. And, by using expansion features introduced in ver.3.0 (8. Use of short address specifications, and 9. Control of register save/restore operations), object size can be further reduced.

(2) SH-3 support

In addition to the SH-1 and SH-2, objects can now be created for the SH-3 (using the -CPU=SH3 option). Also, the following features for the SH-3 are supported.

(a) An -ENDIAN option (-ENDIAN=BIG or LITTLE) corresponding to a feature for setting the order of bits in memory is supported.

(b) A prefetch extended embedded function for generating a cache prefetch instruction (PREF) is supported.

(3) Extension of compiler limits

Compiler limits were extended as indicated in the following table.

Table A.3 Extended Compiler Limits

No.	Description	Ver.2.0	Ver.3.0
1	Number of source programs that can be compiled at once	16 files	unlimited
2	Number of source code lines per file	32,767 lines	65535 lines
3	Number of source code lines in an entire compiled unit	32,767 lines	unlimited
4	Maximum number of #include nesting levels	8 levels	30 levels

(4) Support for Japanese character codes in character strings

Shift-JIS and EUC Japanese character codes can also be included in programs as character string data.

When input codes are shift-JIS (-SJIS option), output codes are also shift-JIS; when input codes are EUC (-EUC option), output codes are also EUC.

However, the graphical user interface currently does not support display of Japanese character code data.

(5) Specification of options using files

By using the -SUBCOMMAND option to specify a file name, options can be included in the specified file rather than on the command line. As a result, numerous complex options need not be entered on the command line each time.

(6) Utilization of the SH-2 divider

The following options are supported to enable use of the SH-2 divider.

(a) Objects which do not use the divider can be generated through the -DIVISION=CPU option.
(b) Objects which use the divider can be generated by using the -DIVISION=PERIPHERAL option. During use of the divider, interrupts are disabled.
(c) Objects which use the divider can be generated through the -DIVISION=NOMASK option. This assumes that the divider will not be used during interrupt processing.

(7) Inline expansion

(a) Inline expansion of C functions
When the -SPEED option is used, the compiler automatically inline-expands small functions. Also, by using the -INLINE option, the maximum size of functions for inline expansion can be modified. Inline expansion can also be explicitly specified using a #pragma statement. The "#pragma inline" statement specifies inline expansion of a user function written in C.

Example (inline expansion of C function):

```
#pragma inline(func)
int func(int a,int b)
{
      return(a+b)/2;
}
```

```
main()
{
   i=func(10,20); /* expanded to i=(10+20)/2 */
}
```

(b) Inline expansion of an assembler function

The "#pragma inline_asm" option can be used to specify inline expansion of user functions written in assembly language. However, when using "#pragma inline_asm" for inline expansion, the output of the compiler is assembly language source code. In such cases debugging at the C language level is not possible.

Example (inline expansion of an assembler function):

```
#pragma inline_asm(rotl)
int rotl(int a)
{
      ROTL  R4
      MOV   R4,R0
}

main()
{
      i=rotl(i); /* set the variable i in the register R4, and expand the code for the function rotl
*/
}
```

(8) Use of short address specifications

(a) Specifying two-byte address variables

Using the "#pragma abs16(<variable name>)" statement, variables can be specified for assignment to an address range addressable using two bytes (-32768 to 32767). By this means, the size of an object referring to such a variable can be reduced.

(b) Specification of GBR base variables

Using the "#pragma gbr(<variable name>)" statement, a variable can be specified for referencing in GBR-relative addressing mode. By this means, the size of an object referencing this variable can be reduced, and memory-based bit manipulation instructions specific to the GBR-relative addressing mode can be employed.

(9) Control of register save/restore operations

The "#pragma noregsave(<function name>)" statement can be used to suppress register save/restore operations at the entry and exit points of functions. This can be used to produce fast, compact functions without register save/restore overhead. A function for which "#pragma noregsave" is specified cannot be called by ordinary functions, but can be called by C language functions which are specified explicitly (using "#pragma regsave") for calling a function for which "#pragma noregsave" has been specified.

By using "#pragma noregsave" with functions which are executed frequently, program size can be reduced and speed of execution increased.

A.3 Features Added between Ver. 3.0 and Ver. 4.1

The features added to version 4.1 of the SuperH RISC engine C/C++ compiler are summarized below.

(1) Register assignment of external variables

The "#pragma global_register(<variable name>=<register number>)" statement can be used to assign external variables to registers.

(2) Cache-savvy optimization

An "-align16" option is supported for assigning labels with 16-byte alignments, for efficient use of cache memory and fetch instructions.

(3) Strengthened inline expansion feature

A feature was added such that, when as a result of inline expansion a function is itself no longer used, it is deleted. Functions which are not themselves necessary after inline expansion should be declared using "static". Similarly, static functions which are not called or referenced by address are deleted.

Examples:

```
#pragma inline(func)
int a;
static int func(){
   a++;
}
main(){
   func();
}
```

```
#pragma inline(func)
   int a;
   /* func() function is itself deleted */


main(){
     a++; /* inline expansion
 }
```

(4) Recursive inline expansion

A feature was added for recursive inline expansion of functions. The depth of recursion can be specified using the "-nestinline" option.

(5) Option for loop expansion optimization

The "-loop" and "-noloop" options can be used to specify whether or not loop processing is expanded in optimization, independently of the "-speed" and "-size" options. (These options are invalid when the option to omit optimization is specified.)

(6) Option for two-byte-address variables

Previously, variables with two-byte addresses had to be specified individually using the "#pragma abs16" statement, but now an "-abs16" option enables specification for all variables at once. The option "-abs16=run" specifies two-byte addresses only for runtime routines; "-abs16=all" specifies two-byte addresses for all variables and functions, including runtime routines.

(7) Upper byte of function return value guaranteed

Previously, the upper byte of values returned by functions in the (unsigned) char and short types was not guaranteed. By specifying the "-rtnext" option, the upper byte of the return value is now guaranteed (the upper byte of R0 is sign-extended or zero-extended).

(8) More complete listing files

Compared with previous versions, the information contained in object lists and assembly lists is now more complete and easier to read.

By the simultaneously output in statement units of C source and assembly language source in a list file, the correspondence between them is easier to grasp (using the "-show=source,object" option).

(in addition, the default for the "-show" option was changed from source to nosource.)

A list of runtime routine names used in a function has been added, as information for computing the amount of stack space used by the function.

The data loaded by an instruction for data loading from a constant pool is added as a comment.

Examples:

```
1:  float x;
2:  func(){
3:     x/=1000;
4:  }

Listing file

func.c   1     float x;
func.c   2     func(){*(a) Simultaneous output of C source and assembly language
                      code
000000         _func:               ; function: func
                                    ; frame size=4

                                    ; used runtime library name:

                                    ; divs *(b) Runtime routine name
000000    4F22           STS.L   PR,@-R15
func.c    3 x/=1000;
000002    D404           MOV.L   L216+2,R4 ; x
000004    D004           MOV.L   L216+6,R0 ; H'447A0000 *(c) Load data
000006    D305           MOV.L   L216+10,R3 ; __divs
000008    430B           JSR     @R3
00000A    6142           MOV.L   @R4,R1
func.c    4       }
00000C    4F26         LDS.L    @R15+,PR
00000E    000B         RTS
000010    2402         MOV.L    R0,@R4
000012      L216:
000012    00000002       .RES.W     1
000014    <00000000>     .DATA.L  _x
000018    447A0000       .DATA.L  H'447A0000
00001C    <00000000>     .DATA.L  __divs
000000      _ x:                       ; static:      x
000000 00000004      .RES.L     1
```

(9) More complete error messages

By specifying the "-message" option to output information messages, programming errors can be checked more easily.

Examples:

```
1:  void func(){
2:        int a;
3:        a++;
4: sub(a);
5:    }
```

Information messages

```
line 3: 0011 (I) Used before set symbol: "a"    (reference of undefined auto variable)
line 4: 0200 (I) No prototype function             (no prototype declaration)
```

In addition, the identifier, token or number causing the error is added to the message to make it easier to find the error location.

Examples:

```
: 2118 (E) Prototype mismatch "identifier"
: 2119 (E) Not a parameter name "identifier"
: 2201 (E) Cannot covert parameter "number"
: 2225 (E) Undeclared name "identifier"
: 2500 (E) Illegal token "token"
```

(10) Automatic conversion of Japanese character codes

When a character string containing either EUC or shift-JIS Japanese character codes is output to an object file, the Japanese character codes are automatically converted to the encoding specified by an encoding option.

(a) An "-outcode=euc" option causes automatic conversion to EUC codes.
(b) An "-outcode=sjis" option results in automatic conversion to shift-JIS codes.

(11) Specification of CPU type by an environment variable

It is now possible to use an environment variable instead of a command line option to specify the CPU type.

Environment variable specification

```
SHCPU=SH1     (equivalent to the "-cpu=sh1" option)
SHCPU=SH2     (equivalent to the "-cpu=sh2" option)
SHCPU=SH3     (equivalent to the "-cpu=sh3" option)
```

(12) Option to treat double data types as float types

By using the "-double=float" option, data declared as the double type can be read as the float type. In programs where the precision of the double type is not required, execution speed can be improved without the need to modify the source code.

A.4 Features Added between Ver. 4.1 and Ver. 5.0

The features added to version 5.0 of the SuperH RISC engine C/C++ compiler are summarized below.

(1) Extension of the number of characters in a line

The limit on the number of characters in a single logical line was extended from 4,096 characters to 32,768 characters.

(2) Removal of the limit on compiler source lines

The limit of 65,535 lines in a single file for compiling was removed. However, that part of the file exceeding 65,535 lines cannot be debugged.

(3) Compatibility with SH-4 instructions

This compiler version is also compatible with the SH-4, to maintain compliance with the SH Family of CPUs. By using the "-cpu=sh4" option, SH-4 objects can be generated.

(4) Addition of a normalize mode

By using the "-denormalize=on|off" option, it is possible to choose whether to handle non-normalized numbers or set them to zero. This is valid only when "-cpu=sh4" is used.

However, when "-denormalize=on", if a non-normalized number is input to the FPU, an exception occurs. Hence an exception handler must be written to handle processing of non-normalized numbers.

(5) Addition of a rounding mode

By using the "-round=nearest|zero" option, it is possible to choose whether to round to zero or to the nearest number. This is valid only when "-cpu=sh4" is used.

(6) Compatibility of compiler option environment variable with SH-4

Instead of using command line options to specify the CPU, the environment variable "SHCPU" can be used to specify the SH-4, by setting "SHCPU=SH4".

(7) Compatibility with the SH-2E

By using the "-cpu-sh2e" option, objects for the SH-2E can be generated.

(8) Compatibility of compiler option environment variable with SH-2E

Instead of using command line options to specify the CPU, the environment variable "SHCPU" can be used to specify the SH-2E, by setting "SHCPU=SH2E".

(9) Use of extensions to distinguish between C and C++ files

By selective use of the shc and shcpp commands, the compiler enables determination of the syntax used. Now, C++ files can be compiled based on file extensions or an options even when using the shc command. For details refer to the table below.

Table A.4 Conditions for Determining Compiling Syntax

Command	Option	Extension of File for Compiling	Syntax Used in Compiling
shcpp	Arbitrary	Arbitrary	Compiled as C++
shc	-lang=c	Arbitrary	Compiled as C
	-lang=cpp		Compiled as C++
	No -lang option	*.c	Compiled as C
		*.cpp, *.cc, *.cp, *.CC	Compiled as C++

A.5 Features Added between Ver. 5.0 and Ver. 5.1

The features added to version 5.1 of the SuperH RISC engine C/C++ compiler are summarized below.

(1) Support for the SH3-DSP library

In addition to the older SH-DSP, support is now also available for libraries that can be applied to SH3-DSP.

(2) Support for embedded C++ language

Support is now available for embedded C++ language specification, which is the C++ specification compatible with embedded systems.

- Support for bool-type
- Multiple inheritance warnings
- Support for embedded C++ language class libraries

(3) Support for inter-module optimization functions

Implements the following optimization, and generates objects with optimal size/speed.

With this optimization, size is reduced by approximately 10%, and execution speed is improved by 7 to 8%.

- Reduction of superfluous register save/restore code
- Deletion of unreferenced variables/functions
- Routinization of common codes
- Optimization of function call codes

(4) Improved compiling speed

Fast compiling speed has been achieved through improved optimization processing.

A maximum of double speed, and an average speed increase of 130% has been achieved.

(5) Extension of limits

- The limit on command line length has been extended from 256 to 4,096.
- The limit on file name length has been extended from 128 to 251.
- The limit on character string literal length has been extended from 512 to 32,767.

(6) Strengthened optimization

The various kinds of optimization for improving object performance have been strengthened.

(7) Support for C++ comments

In the C language, use of "//" comments is now possible.

(8) Changes to the integrated environment (PC version)

The older PC integrated environment HIM (Hitachi Integration Manager) has been replaced by the new integrated environment HEW (High-performance Embedded Workshop).

The following functions have been added, as compared with HIM.

Project generator
Automatically generates header files that define peripheral I/Os for each CPU.
Combination interface with the version management tools
Supports the interface with the version management tools provided by the third party.
Hierarchy project support
Can define multiple subprojects in a project and hierarchically manage them.
Network support
Provides development environment under WindowsNT CSS.

A.6 Features Added between Ver. 5.1 and Ver. 6.0

The features added to version 6.0 of the SuperH RISC engine C/C++ compiler are summarized below.

(1) Relaxation of limits

Limits for source programs and command lines have been greatly relaxed.

- File name length: 251 bytes → No limit
- Symbol length: 251 bytes → No limit
- Number of symbols: 32,767 symbols → No limit
- Number of source program rows: C/C++: 32,767 rows, ASM: 65,535 rows → No limit
- C program character string length: 512 characters → 32,766 characters
- Assembly program row length: 255 characters → 8,192 characters
- Subcommand file row length: ASM: 300 bytes, optlnk: 512 bytes → No limit
- Number of parameters for the Optimizing Linkage Editor rom option: 64 parameters → No limit

(2) Hyphens (-) in directory names and filenames

Hyphens (-) can now be specified in directory names and filenames

(3) Elimination of copyright notice

By specifying the logo/nologo option, it is now possible to specify whether or not to display a copyright notice.

(4) Error message prefixes

Along with support for the error help function in the Integrated Development Environment, the start of error messages in the compiler and Optimizing Linkage Editor have been ascribed prefixes.

(5) Addition of fpscr options

If the cpu=sh4 option is specified, and the fpu option is not specified, it is now possible to specify whether to guarantee the FPSCR register precision mode before and after calling on the function.

(6) #pragma extensions

#pragma extensions can now be written without ( ).

(7) Addition of embedded functions

trace functions have been added.

(8) Addition of implicit declarations

_ _HITACHI_ _ and _ _HITACHI_VERSION_ _ are implicitly declared with #define.

(9) static label name

In order that static labels inside the file can be referenced by #pragma inline_asm, the label name has been changed to _ _$ (name). However, it is displayed as _(name) in the linkage list.

(10) Extension and changes to the language specification

- Errors when unions are initialized have been eliminated.

Example:

```
union{
char c[4];
} uu={{'a','b','c'}};
```

- It is now possible to substitute a structure and make a declaration at the same time.

Example:

```
struct{
int a, int b;
} s1
void test()
{
struct S s2=s1;
}
```

- The boundary alignment of bool-type data is now 4 bytes.
- Exception processing and template functions are now supported as the C++ language specification.
- The C preprocessor is now ANSI/ISO compliant.

A.7 Features Added between Ver. 6.0 and Ver. 7.0

From the SuperH RISC engine C/C++ Compiler Ver.7.0 algorithm and code generation has been greatly improved. So the options and generated codes are much different from those of Ver.6.0.
The features added to version 7.0 of the SuperH RISC engine C/C++ compiler are summarized below.

(1) External access optimization function (map option support)

This function performs optimization of external variable access and function branch instructions based on the allocated address of the variables and functions at linkage. Optimization is implemented by recompiling the external symbol allocation information files which are output (specified to map option) by the Optimizing Linkage Editor at the time of the first linkage.

(2) Automatic generation of GBR relative access code (gbr option support)

If gbr=auto is specified, the compiler automatically generates GBR settings and GBR relative access code. Before and after a function call, the GBR value is guaranteed. However, GBR-related embedded functions cannot be used.

(3) Strengthened speed/size selection options

speed/size selection options (shift, blockcopy, division, approxdiv options) have been added, and it is now possible to make finer size/speed adjustments.

(4) Strengthened functions for embedded systems

- Addition of embedded functions
  Double precision multiplication, SWAP instruction, LDTLB instruction, NOP instruction
- Addition and change of #pragma extension
  Support for #pragma entry entry function specification and SP setting
  Support for #pragma stacksize stack size specification
  Support for #pragma interrupt sp=<variable>+<constant> and sp=&<variable>+<constant>
- Support for section operators
  Supports functions of coding the size references in C language.
- Relaxation of address cast errors
  Errors of cast expressions with regard to address initialization when initializing external variables have been relaxed.

(5) Improved libraries

- Support for reentrant libraries
  If the reent option is specified with the Library Creation Tool, a reentrant library is generated.
- The units of the malloc reserve size and the number of input and output files has been made variable.
  It is now possible to specify the malloc reserve size with _sbrk_size, and the number of input and output files with _nfiles in the initial settings of the C/C++ library functions. This substantially reduces RAM capacity.
  If this specification is omitted, the malloc reserve size is 520, and the number of input and output files is 20.
- Support for easy I/O
  If the nofloat option is specified with the Library Creation Tool, floating point conversions are not supported, and a small I/O routine is generated.

(6) Addition of optimization options (V7.0.06)

- Added Options

The following shows the options added to Ver.7.0.06. Uppercase letters indicate the abbreviations and characters underlined indicate the defaults.

Table A.5 Added options

	Item	Command Line Format	Specification
1	Treatment of global variables	GLOBAL_Volatile = { 0 |	Treat global variables as non-volatile-qualified except variables which are volatile-qualified
		1 }	Treats global variables as volatile-qualified
2	Optimizing range of global variables	OPT_Range = {All |	Optimizes all the global variables in a whole function
		NOLoop |	Suppresses a motion of global variables out of a loop or optimization of a loop control variable
		NOBlock }	Suppresses an optimization of the global variables cross over a branch or a loop
3	Deletion of vacant loops	DEL_vacant_loop ={ 0 |	Suppresses a deletion of a vacant loop
		1}	Deletes a vacant loop
4	Specification of maximum unroll factor	MAX_unroll = <numeric value> <numeric value>:1-32	Specifies the maximum number of loop unroll factor Default : 1 (when the speed or loop option is specified, the default is 2)
5	Deletion of assignments before an infinite loop	INFinite_loop = {0 |	Suppresses a deletion of assignments to global variables before an infinite loop
		1 }	Deletes assignments to global variables before an infinite loop
6	Allocation of global variable	GLOBAL_Alloc = {0 |	Suppresses register allocation of global variables
		1 }	Allocates registers of global variables
7	Allocation of struct/union member	STRUCT_Alloc = {0 |	Suppresses register allocation of struct or union members
		1 }	Allocates registers to struct or union members
8	Propagation of const-qualified variable	CONST_Var_propagate = {0 |	Suppresses the propagation of variables which are const-qualified
		1 }	Propagates variables which are const-qualified
9	IInline expansion of constant load	CONST_Load = {Inline |	Performs inline expansion of constant load
		Literal }	Loads constant data from literal pool Default : When size is specified, up to two or three instructions are expanded
10	Scheduling of instructions	SChedule = {0 |	Suppresses instruction scheduling
		1 }	Schedules instructions

GLOBAL_Volatile

Optimize[Details][Global variables][Treat global variables as volatile qualified]

Command Line Format

```
GLOBAL_Volatile = { 0 | 1 }
```

Description

When **global_volatile=0** is specified, the compiler optimizes accesses of the global variables which are non-volatile-qualified. So a count or an order of accesses to global variables may differ from that of the C/C++ program.

When **global_volatile=1** is specified, all the global variables are treated as volatile-qualified. So a count or an order of accesses to global variables may be the same as that of the C/C++ program.

The default for this option is **global_volatile=**0.

Remarks

When **global_volatile=1** is specified, **schedule=0** becomes the default.

OPT_Range

Optimize[Details][Global variables][Specify optimizing range :]

Command Line Format

```
OPT_Range = { All | NOLoop | NOBlock }
```

Description

When **opt_range=all** is specified, the compiler optimizes accesses to all the global variables in a function.

When **opt_range=noloop** is specified, the compiler does not optimize accesses to the global variables which are used in a loop or a loop conditional expression.

When **opt_range=noblock** is specified, the compiler does not optimize accesses to the global variable cross over a branch or a loop.

The default for this option is **opt_range=all**.

Example

(1) Example of optimization across a branch (opt_range=all or noloop is specified)

```
        int A,B,C;
        void f(int a) {
            A = 1;
            if (a) {
                B = 1;
            }
            C = A;
        }

<source image after optimizing>
        void f(int a) {
            A = 1;
            if (a) {
                B = 1;
            }
            C = 1; /* Deletes reference of variable A and propagates A=1 */
}
```

(2) Example of optimization against loop (opt_range=all is specified)

```
        int A,B,C[100];
        void f() {
            int i;
            for (i=0;i<A;i++) {
                C[i] = B;
            }
        }

<source image after optimizing>
        void f() {
            int i;
            int temp_A, temp_B;
            temp_A = A; /* Remove reference of variable A used in loop conditional expression */
            temp_B = B; /* Remove reference of variable B in a loop */
            for (i=0;i<temp_A;i++) { /* Delete reference of variable A */
                C[i] = temp_B; /* Delete reference of variable B */
            }
}
```

Remarks

Whenever **opt_range=noloop** is specified, **max_unroll=1** becomes the default.
Whenever **opt_range=noloblock** is specified, **max_unroll**=1, **const_var_propagate**=0, and **global_alloc=0** becomes the default.

DEL_vacant_loop

Optimize[Details][Miscellaneous][Delete vacant loop]

Command Line Format

```
DEL_vacant_loop = { 0 | 1 }
```

Description

When **del_vacant_loop=0** is specified, the compiler does not delete a vacant loop.

When **del_vacant_loop=1** is specified, the compiler deletes a vacant loop.

The default for this option is **del_vacant_loop=0.**

Remarks

Note that the default differs between version 7.0.04 and 7.0.06.

Up to V7.0.04 : Delete vacant loop
V7.0.06 or later : Does not delete vacant loop

MAX_unroll

Optimize[Details][Miscellaneous][Specify maximum unroll factor :]

Command Line Format

```
MAX_unroll = <numeric value>
```

Description

Specifies the maximum unroll factor when a loop is expanded.

The <numeric value> accepts a decimal number from 1 to 32. If < numeric value > is specified out of the range, an error will occur.

When the **speed** or **loop** option is specified, the default for this option is **max_unroll=2.**

Otherwise the default for this option is **max_unroll=1.**

Remarks

Whenever **opt_range=noloop** or **opt_range=noblock** is specified, the default for this option is **max_unroll=1.**

INFinite_loop

Optimize[Details][Global variables]

[Delete assignment to global variables before an infinite loop]

Command Line Format

```
INFinite_loop = { 0 | 1 }
```

Description

When **infinite_loop=0** is specified, the compiler does not delete an assignment to a global variable before an infinite loop.

When **infinite_loop=1** is specified, the compiler deletes an assignment before an infinite loop to a global variable which is not referred to in the infinite loop.

The default for this option is **infinite_loop =0.**

Example

```
 int A;
 void f()
{
    A = 1; /* Assignment to variable A */
    while(1) {}  /* Variable A is not referred in a loop */
}
```

```
<source image when specified infinite_loop=1)
      void f()
      {
        /* Delete assignment to variable A */
          while(1) {}
}
```

Remarks

Note that the default differs between version 7.x (up to V7.0.04) and 7.0.06 or later.

Up to V7.0.04 : Deletes an assignment before an infinite loop to a global variable which is not referred to in the infinite loop

V7.0.06 or later : Does not delete an assignment to a global variable before an infinite loop

GLOBAL_Alloc

Optimize[Details][Global variables][Allocate registers to global variables :]

Command Line Format

```
GLOBAL_Alloc = { 0 | 1 }
```

Description

When **global_alloc=0** is specified, the compiler does not allocate registers to global variables.

When **global_alloc=1** is specified, the compiler allocates registers to global variables.

The default for this option is **global_alloc=1.**

Remarks

When **opt_range=noblock** is specified, **global_alloc=0** becomes the default.

When **optimize=0** is specified, note that the default differs between version 7.x (up to V.7.0.04) and 7.0.06 or later.

Up to V7.0.04 : Allocates registers to global variables

V7.0.06 or later : Does not allocate registers to global variables

STRUCT_Alloc

Optimize[Details][Miscellaneous][Allocate registers to struct/union members]

Command Line Format

```
STRUCT_Alloc = { 0 | 1 }
```

Description

When **struct_alloc=0** is specified, the compiler does not allocate registers to struct or union members.

When **struct_alloc=1** is specified, the compiler allocates registers to struct or union members.

The default for this option is **struct_alloc=1.**

Remarks

When either **opt_range=noblock** or **global_alloc=0,** and **struct_alloc=1** is specified, the compiler allocates registers only to local struct or union members.

When **optimize=0** is specified, note that the default differs between version 7.x (up to V7.0.04) and 7.0.06 or later.

Up to V7.0.04 : Allocate registers to struct or union members

V7.0.06 or later : Does not allocate registers to struct or union members

CONST_Var_propagate

Optimize[Details][Global variables][Propagate variables which are const qualified :]

Command Line Format

```
CONST_Var_propagate = { 0 | 1 }
```

Description

When **const_var_propagate=0** is specified, the compiler does not propagate global variables which are const-qualified.

When **const_var_propagate=1** is specified, the compiler propagates global variables which are const-qualified.

The default for this option is **const_var_propagate=**1.

Example

```
const int X = 1;
int A;
void f() {
    A = X;
}
```

<source image when specified const_var_propagate=1>

```
void f() {
    A = 1; /* Propagates X=1 */
}
```

Remarks

When **opt_range=noblock** is specified, the default for this option is **const_var_propagate=0.**

Variables which are const-qualified in C++ program are always propagated even if const_var_propagate=0 is specified.

CONST_Load

Optimize[Details][Miscellaneous][Load constant value as :]

Command Line Format

```
CONST_Load = { Inline | Literal }
```

Description

When **const_load=inline** is specified, the load of all the 2-byte constant data or some 4-byte constant data is expanded.

When **const_load=literal** is specified, all the 2-byte or 4-byte constant data are loaded from literal pool.

The default for this option is below.

When the **speed** option is specified:

The default is **const_load=inline**.

When the **size** or **nospeed** option is specified:

If 2-byte or 4-byte constant data can be expanded into 2 or 3 instructions respectively,

const_load=inline is applied.

Otherwise the default is **const_load=literal**.

Example

```
    int f() {
    return (257);
}
```

(1) When const_load=inline or speed option is specified:

```
    MOV #1,R0 ; R0 <- 1
    SHLL8  R0 ; R0 <- 256 (1<<8)
    RTS
    ADD #1,R0 ; R0 <- 257 (256+1)
```

(2) When const_load=literal, size or nospeed is specified:

```
    MOV.W  L11,R0
    RTS
    NOP
L11:
    .DATA.W   H'0101
```

SChedule

Optimize[Details][Global variables][Schedule instructions :]

Command Line Format

```
SChedule = { 0 | 1 }
```

Description

When **schedule=0** is specified, the compiler does not schedule instructions. They will be executed in the order written in the C/C++ program.

When **schedule=1** is specified, the compiler schedules instructions paying attention to the pipeline or superscalar (only SH-4) mechanism.

The default for this option is **schedule=1**.

Remarks

When **opt_range=noblock** is specified, **schedule=0** becomes the default.

- The default in optimize=0

When **optimize=0** is specified, the defaults of the added options are shown below.

```
global_volatile=0
opt_range=noblock
del_vacant_loop=0
max_unroll=1
infinite_loop=0
global_alloc=0
struct_alloc=0
const_var_propagate=0
const_load=literal
schedule=0
```

The defaults of the following options differ from **optimize=1**.

	optimize=0	optimize=1
opt_range	noblock	all
global_alloc	0	1
struct_alloc	0	1
const_var_propagate	0	1
const_load	literal	Depending on speed/size/nospeed
schedule	0	1

- Compatibility in V7 (up to V7.0.04)

The defaults of the following options differ between version 7.x (up to V.7.0.04) and 7.0.06 or later.

(i) Deletion of a vacant loop (del_vacant_loop)

Up to V7.0.04 : Deletes a vacant loop

V7.0.06 or later : Does not delete a vacant loop

(ii) Deletion of an assignment before an infinite loop (infinite_loop)

Up to V7.0.04 : Deletes an assignment before an infinite loop to global variable which is not referred to in the infinite loop

V7.0.06 or later : Does not deletes assignment to global variable before an infinite loop

The specification of the following with **optimize=0** differs between version 7.x (up to V.7.0.04) and 7.0.06 or later.

(i) Allocation of global variables (global_alloc)

Up to V7.0.04 : Allocates global variables to registers

V7.0.06 or later : Does not allocate global variables to registers

(ii) Allocation of struct or union members (struct_alloc)

Up to V7.0.04 : Allocates struct or union members to registers

V7.0.06 or later : Does not allocate struct or union members to registers

- System of Optimization

The levels of the optimization of global variables are shown below. When one of those levels is selected in HEW, the options related to the optimization of global variables can be controlled together.

The level is set at Optimize[Details][Level :].

(i) Level 1

All the optimizations of global variables are suppressed.

```
global_volatile=1
opt_range=noblock
infinite_loop=0
global_alloc=0
const_var_propagate=0
schedule=0
```

(ii) Level 2

The optimizations of global variables which are not volatile-qualified are done within a basic block

(sequence of instructions which have no labels or branches except at beginning or end).

```
global_volatile=0
opt_range=noblock
infinite_loop=0
global_alloc=0
```

```
const_var_propagate=0
schedule=1
```

(iii) Level 3

All the optimizations of global variables which are non-volatile-qualified are done.

```
global_volatile=0
opt_range=all
infinite_loop=0
global_alloc=1
const_var_propagate=1
schedule=1
```

(iv) Custom

User specifies these options according to the programs.

When level 1, level 2, or level 3 is specified, above-mentioned options cannot be changed separately.

- The followings are features added to Optimizing Linkage Editor.

(7) Support for wild cards

It is possible to specify wild cards for input files and start option section names.

(8) Search path

It is possible to specify search paths for multiple input files and library files with the environment variable (HLNK_DIR).

(9) Separate output of load modules

It is possible to perform separate output of absolute load module files.

(10) Changed error levels

The error level for messages for information, warnings, and error levels, and whether or not to output them can be changed individually.

(11) Support for binary and HEX

It is now possible to input and output binary files.

In addition, it is now possible to choose to output in the Intel HEX format.

(12) Output of the stack capacity usage information

With the stack option, it is possible to output data files for the stack analysis tools.

(13) Debug information deletion tool

With the strip option, it is possible to delete just the debug information within the load module files and library files.

The features added to version 7.1 of the SuperH RISC engine C/C++ Optimizing Linkage Editor are summarized below.

(14) Output external symbol allocation information files (map option support)

If the map option is specified, the compiler generates an external symbol allocation information file to be used for external variable access optimization.

A.8 Features Added between Ver. 7.0 and Ver. 7.1

- The features added to version 7.1 of the SuperH RISC engine C/C++ compiler are summarized below.

(1) Strengthened optimization

(a) Deletion of EXTU immediately after MOVT

Deletes the unnecessary EXTU immediately after MOVT.
(As nothing besides 1 or 0 can be set, EXTU is unnecessary)

Before optimization		After optimization	
_f:		_f:	
MOV.L	L12+2,R6 ;	MOV.L	L12+2,R6 ;
_a1		_a1	
MOV.B	@R6,R0	MOV.B	@R6,R0
TST	#128,R0	TST	#128,R0
MOVT	R0	MOVT	R0
EXTU.B	R0,R0		

As nothing besides 1 or 0 can be set for R0, EXTU is unnecessary.

(b) Deletion of EXTU after a right shift of a zero extended register

Even if a zero extended register is zero extended after a right shift, the value does not change so it is deleted.

Before optimization			After optimization		
f:			f:		
	MOV.L	L13+2,R2;		MOV.L	L13+2,R2;
_a2			_a2		
	MOV	#1,R5		MOV	#1,R5
	MOV.W	@R2,R6		MOV.W	@R2,R6
	EXTU.W	R6,R6		EXTU.W	R6,R6
	MOV	R6,R2		MOV	R6,R2
	SHLR2	R2		SHLR2	R2
	SHLR	R2		SHLR	R2
	EXTU.W	R2,R2		CMP/GE	R5,R2
	CMP/GE	R5,R2		:	
	:				

As the upper 2 bytes are zero-cleared with EXTU, the value does not change even if EXTU is performed again.

(c) Unifying consecutive AND

If ANDs to the same variable are made consecutively, they are grouped into 1 AND.

Before optimization	After optimization
_f: MOV.L L11+2,R6 ; _a5 MOV.B @R6,R0 AND #3,R0 RTS AND #1,R0	_f: MOV.L L11+2,R6 ; _a5 MOV.B @R6,R0 RTS AND #1,R0
Grouped into 1 AND.	

(d) Bit field comparison and combination

Unifies evaluation (TST#n, R0) of multiple bit fields.

Before optimization	After optimization

_f:
:
MOV R4,R0
TST #64,R0
BF L12
TST #32,R0
BF L12
MOV R6,R0
:
	_f:
	:
	MOV R4,R0
	TST #96,R0
	BF L12
	MOV R6,R0
	:

Unifies the criteria of the bit fields, and replaces them with 1 evaluation.	

(e) Deletion of EXTS of consecutive EXTS+EXTU

After EXTS, if EXTU of the same size is executed, EXTS is unnecessary so it is deleted.

Before optimization	After optimization
_f: : EXTS.B R6,R2 EXTU.B R2,R0 :	_f: : EXTU.B R6,R0 :
EXTU is executed on a value from EXTS, so EXTS is unnecessary.	

(f) Deletion of MOVT(+XOR)+EXTU+CMP/EQ

Deletes the unnecessary MOVT(+XOR)+EXTU+CMP/EQ after TST, and makes a conversion so as to reference the T bit with a direct branch instruction.

Before optimization	After optimization
_f: : TST #4,R0 MOVT R0 MOV.L L23+6,R6 ; _st2 XOR #1,R0 EXTU.B R0,R0 CMP/EQ #1,R0 MOV.B @R6,R0 BF L16 :	_f: : TST #4,R0 MOV.L L23+6,R6; _st2 MOV.B @R6,R0 BT L16 :
Directly references the T bit.	

(g) AND #imm, R0+CMP/EQ #imm, R0 → TST #imm, R0

Replaces AND #imm, R0+CMP/EQ #imm, R0 with TST #imm, R0.

Before optimization	After optimization
L17: MOV.B @R6,R0 AND #1,R0 CMP/EQ #1,R0 BF L19 MOV.B @R5,R0 AND #1,R0	L17: MOV.B @R6,R0 TST #1,R0 BT L19 MOV.B @R5,R0 AND #1,R0

(h) Deletion of EXTU when comparing (==) unsigned char and constant

Deletes the unnecessary EXTU when comparing the unsigned char and constant immediately after the load.

Before optimization	After optimization
_f: MOV.L L11,R6 ; _b MOV.B @R6,R2 MOV #-128,R6; H'FFFFFF80 EXTU.B R6,R6 EXTU.B R2,R2 CMP/EQ R6,R2 MOVT R2 MOV.L L11+4,R6 ; _a RTS MOV.B R2,@R6	_f: MOV.L L11,R6 ; _b MOV.B @R6,R2 MOV #-128,R6; H'FFFFFF80 CMP/EQ R6,R2 MOVT R2 MOV.L L11+4,R6 ; _a RTS MOV.B R2,@R6
Deletes the unnecessary extension.	

(i) Deletion of extension after LOAD / before STORE of bit field

Deletes the unnecessary extension of the bit field after LOAD and before STORE.

Before optimization	After optimization
_f: MOV.L L11+2,R6;_st MOV.B @R6,R2 EXTU.B R2,R0 OR #128,R0 :	_f: MOV.L L11+2,R6;_st MOV.B @R6,R2 OR #128,R0 :
Deletes the unnecessary extension.	

(j) Deletion of copy when evaluating switch-case

Deletes the copy of the value when performing each case evaluation of switch statements.

Before optimization

```
_f:
                    :
        MOV         R0,R2
        MOV         R2,R0
        CMP/EQ      #1,R0
        BT          L24
        CMP/EQ      #2,R0
        BT          L26
        MOV         R2,R0
        CMP/EQ      #3,R0
        BT          L28
        MOV         R2,R0
        CMP/EQ      #4,R0
        BT          L30
        MOV         R2,R0
                    :
```

After optimization

```
_f:
                    :
        MOV         R0,R2
        MOV         R2,R0
        CMP/EQ      #1,R0
        BT          L24
        CMP/EQ      #2,R0
        BT          L26
        CMP/EQ      #3,R0
        BT          L28
        CMP/EQ      #4,R0
        BT          L30
                    :
```

Deletes the unnecessary copy.

(k) Unifying consecutive OR

If ORs to the same variable are made consecutively, they are grouped into 1 OR.

Before optimization

```
_f:
    MOV.L       L11+2,R6    ;
_a5
    MOV.B       @R6,R0
    OR          #3,R0
    RTS
    OR          #1,R0
```

After optimization

```
_f:
    MOV.L       L11+2,R6    ;
_a5
    MOV.B       @R6,R0
    RTS
    OR          #3,,R0
```

Grouped into 1 OR.

(l) Deletion of EXTS immediately in front of AND #imm,R0 or TST #imm,R0

Deletes the unnecessary extension immediately in front of;

(i) AND #imm,R0
(ii) TST #imm,R0

Before optimization	After optimization
_f: : EXTS.B R6,R0 AND #32,R0 :	_f: : AND #32,R0 :
_f: : EXTS.B R6,R0 TST #32,R0 :	_f: : TST #32,R0 :
Deletes the unnecessary extension.	

(m) Deletion of EXTU of consecutive EXTU+EXTS

After EXTU, if EXTS of the same size is executed, EXTU is unnecessary so it is deleted.

Before optimization	After optimization
_f: : EXTU.B R6,R2 EXTS.B R2,R0 :	_f: : EXTS.B R6,R0 :
EXTS is executed on a value from EXTU, so EXTU is unnecessary.	

(n) Deletion of EXTU immediately after XOR #imm,R0(OR,AND) after MOVT

Deletes the unnecessary EXTU immediately after;

(i) XOR #imm,R0

(ii) OR #imm,R0

(iii) AND #imm,R0

after MOVT

Before optimization	After optimization
: MOVT R0 XOR #1,R0 RTS EXTU.B R0,R0	: MOVT R0 RTS XOR #1,R0
MOVT R0 OR #2,R0 RTS EXTU.B R0,R0	: MOVT R0 RTS OR #2,R0
: MOVT R0 AND #1,R0 RTS EXTU.B R0,R0	: MOVT R0 RTS AND #1,R0
Deletes the unnecessary extension.	

(o) Deletion of unnecessary EXTS when making comparison

Deletes redundant EXTS re-output when comparing registers after sign expansion.

Before optimization	After optimization
_f: : EXTS.B R6,R6 CMP/GT R6,R2 BF L13 :	_f: : CMP/GT R6,R2 BF L13 :
If R6 is already extended previously, EXTS is unnecessary.	

(p) Disabling (immediately) of allocation of constant values to registers

Disables allocation of functional parameter constants (-128 to 127) to registers.

Before optimization	After optimization
_f:	_f:
PUSH R14	
:	:
MOV.B #127,R14	
:	:
MOV.B R14,R4	MOV.B #127,R4
BSR sub	BSR sub
:	:
POP R14	

Loads directly constant values #127 to parameter registers without allocating to registers.

(q) Strengthened DT instructions

Performs DT instruction for variables allocated to registers.

Before optimization	After optimization
_f:	_f:
MOV.L L16+2,R6; _x	MOV.L L16+2,R6; _x
MOV.L @R6,R2	MOV.L @R6,R2
ADD #-1,R2	DT xxxx R2 xxxx
TST R2,R2	BT/S L12
BT/S L12	:
:	

Performs DT instruction.

(r) Improved literal output position

Precision of instruction size calculation when deciding literal data output position is improved, and it is possible to output the literal data output position later.

(s) Deletion of 1byte&=1byte redundant EXTU

Deletes the unnecessary EXTU when 1byte&=1byte.

Before optimization	After optimization
_f: : MOV.B @(R0,R7),R6 MOV.B @R5,R2 EXTU.B R6,R6 AND R6,R2 MOV.B R2,@R5 MOV.B @R14,R2 :	_f: : MOV.B @(R0,R7),R6 MOV.B @R5,R2 AND R6,R2 MOV.B R2,@R5 MOV.B @R14,R2 :
Deletes the unnecessary extension.	

(t) 2 byte literal expansion

Prevents the same code from being expanded twice.

Before optimization	After optimization

_f:
MOV.L L13+4,R4 ; _b
SHLL8 R0
ADD #-48,R0
MOV.W @(R0,R4),R2
MOV #8,R0
SHLL8 R0
ADD #-46,R0
EXTU.W R2,R6
MOV.W @(R0,R4),R2
MOV #8,R0
SHLL8 R0
ADD #-44,R0
EXTU.W R2,R5
MOV.W @(R0,R4),R2
	_f:
	MOV.L L13+4,R4 ; _b
	SHLL8 R0
	ADD #-48,R0
	MOV.W @(R0,R4),R2
	MOV #8,R0
	SHLL8 R0
	ADD #-46,R0
	EXTU.W R2,R6
	MOV.W @(R0,R4),R2
	ADD #2,R0
	EXTU.W R2,R5
	MOV.W @(R0,R4),R2

Prevents the same code from being expanded twice.	

(u) Improving expansion of loop condition determination

If size is given priority, copying of loop determination is not executed when performing loop condition determination.

Before optimization	After optimization (v7)	After optimization (v7.1)

`while (cond) {`
`  :`
`}`
	`if (cond) {`
	`  do {`
	`  :`
	`  } while (cond);`
	`}`
		`goto L1;`
		`do {`
		`  :`
		`L1:;`
		`  } while (cond);`

cond appears in one place rather than in two places.		

(v) Elimination of redundant if statement condition determination

When the result of the first if statement makes the later if statement unnecessary, the later if statement is eliminated.

Before optimization	After optimization

`if (cond)`
`  t=65;`
`else`
`  t=67;`
`if (t == 65)`
`  fx();`
`else`
`  fy();`
	`if (cond) {`
	`  t=65;`
	`  fx();`
	`} else {`
	`  t=67;`
	`  fy();`
	`}`

When the result of the first if statement makes the later if statement unnecessary, the later if statement is eliminated.	

(w) Direct operations of temporary variables

Disables substitution to redundant temp variables, and changes the operation sequence of the equation.

Before optimization	After optimization
`k = i + prime;` `p = flags + k;`	`p = i + prime + flags;`
k is not used later so superfluous substitution to temp is not executed.	

(x) Post increment addressing

Uses MOV.L @Rm+,Rn for the LOAD 4-byte variable.

Before optimization		After optimization	
	:		:
L11:		L11:	
MOV.L	@R5,R2	MOV.L	@R5+,R2
ADD	#4,R5	DT	R6
DT	R6	ADD	R2,R4
ADD	R2,R4	BF	L11
BF	L11		:
	:		

Executes MOV.L @Rm+,Rn with one instruction.

(y) Improving loop termination conditions

Relaxes conditions for performing optimization of loop termination, and makes optimization easy to apply.

Before optimization	After optimization
int a, b;	int a, b;
func() {	func() {
unsigned short sx;	
	a++;
for (sx=0; sx<1; sx++) {	b++;
a++;	f();
b++;	
f();	}
}	
}	

Performs loop termination.

(z) Optimization of 1-bit evaluation

Groups conditional expressions that reference multiple bit fields of 1-bit width into 1, and generates code that simultaneously performs fetching and comparison of values using bit AND.

Before optimization	After optimization

struct S {
  char bit0:1;
  char bit1:1;
  char bit2:1;
  char bit3:1;
}ss1;
if((ss1.bit0|ss1.bit1|ss1.bit2)!=
0){
  :
  :
}
	struct S {
	  char bit0:1;
	  char bit1:1;
	  char bit2:1;
	  char bit3:1;
	}ss1;
	if ((*(char *)&ss1 & 0xe0)!= 0){
	  :
	  :
	}

Simultaneously performs fetching and comparison using AND.	

A.9 Features Added between Ver. 7.1 and Ver. 8.0

The features added to version 8.0 of the SuperH RISC engine C/C++ compiler are summarized below.

(1) Supporting new CPUs

SH-4A and SH4AL-DSP are now supported.

(2) Expanding and changing the language specifications

- SP-C is now supported.
- The long long and unsigned long long types are now supported.

(3) Improving the built-in functions

- Adding the built-in functions for DSP
  Absolute value, MSB detection, arithmetic shift, round-off operation, bit pattern copy, modulo addressing setup, modulo addressing cancellation, and CS bit setting
- Adding the built-in functions for SH-4A and SH4AL-DSP
  Sine and cosine calculation, reciprocal of the square root, instruction cache block invalidation, instruction cache block prefetch, and synchronization of data operations
- Adding and changing the #pragma extension
  #pragma ifunc Suppressing the saving or recovery of the floating-point register
  #pragma bit_order Specifying the order of bit fields
  #pragma pack Specifying the alignment number for the structure, union, or class

(4) Automatic selection of the size of the enumerated type (supporting the auto_enum option)

The enumerated type is processed as a smallest type that can contain the enumerated type.

(5) Specifying the alignment number for the structure, union, or class members (supporting the pack option)

The alignment number for the structure, union, or class members can be specified.

(6) Specifying the order of bit fields (supporting the bit_order option)

The order of the bit field members can be specified.

(7) Changing the error level (supporting the change_message option)

The error level for information and warning messages can be changed for each message.

(8) Deregulation of limitations

The maximum allowable number of switch statements is now increased to 2048.

(9) Supporting a fixed point for the DSP library

A fixed point for the DSP library is now supported.

(10) Inter-file inline expansion functionality (supporting the File_inline option)

Inline expansion of functions between files can now be specified.

(11) Compact placement of data within sections (supporting the Data_dtuff option)

Support has been added for functionality that performs linkage by compacting the free space that occurs when section border adjustment for each compile unit is performed. This functionality can be specified to decrease the overall size for data sections.

A.10 Features Added between Ver. 8.0 and Ver. 9.0

- The features added to version 9.0 of the SuperH RISC engine C/C++ compiler are summarized below.

(1) Support for New CPUs

The SH-2A and SH2A-FPU are supported.

An option and a #pragma extension are added to use TBR in the SH-2A and SH2A-FPU.

(2) Extension and Change of Language Specifications

- The following items conform to the ANSI standard.
  - Array index

```
int iarray[10], i=3;
i[iarray] = 0; /* Same as iarray[i] = 0; */
```

  - union bit field specification enabled

```
union u {
  int a:3;
};
```

  - Constant operation

```
static int i=1||2/0; /* Error is changed to warning for zero division */
```

  - Addition of library and macro

```
strtoul, FOPEN_MAX
```

- The following items conform to the ANSI standard when the strict_ansi option is specified, which may cause a difference in results between Ver. 9 and earlier versions.
  - unsigned int and long operations
  - Associativity of floating-point operations
- The variables with register storage class specification are preferentially allocated to registers when the enable_register option is specified.

(3) Enhancement of Intrinsic Functions

- Intrinsic functions for SH-2A and SH2A-FPU are added.

Saturation operations and TBR setting and reference

- Intrinsic functions for instructions that cannot be written in C are added.

Reference and setting of the T bit, extraction of the middle of registers connected,addition with carry, subtraction with borrow, sign inversion, 1-bit division, rotation, and shift.

(4) Loosening Limits on Values

The following limits are loosened.

- Nesting level in a combination of repeat statements (while, do, and for) and select statements (if and switch): 32 levels -> 4096 levels
- Number of goto labels allowed in one function: 511 -> 2,147,483,646
- Nesting level of switch statements: 16 levels -> 2048 levels
- Number of case labels allowed in one switch statement: 511 -> 2,147,483,646
- Number of parameters allowed in a function definition or function call: 63 ->2,147,483,646
- Length of section name: 31 bytes -> 8192 bytes
- Number of sections allowed in #pragma section in one file: 64 -> 2045

(5) Extension of Memory Space Allocation

More detailed settings can be made for memory space allocation.

- abs16/abs20/abs28/abs32 option
- #pragma abs16/abs20/abs28/abs32

(6) Specification of Absolute Address for Variables (support for #pragma address)

An absolute address can be specified for an external variable.

(7) Extension of Optimization for External Variable Access (support for smap option)

Optimization is applied to access to external variables defined in the file to be compiled. Recompilation, which is required for the map option, is not necessary.

(8) Improvement in Precision of Mathematics Library

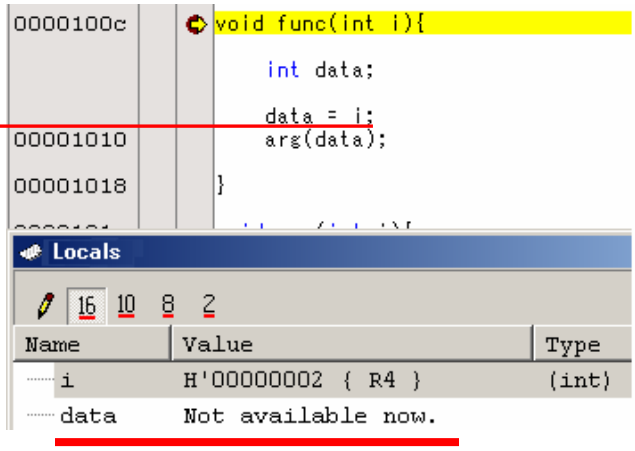
The precision of operation using the mathematics library is improved, which may cause a difference in results between Ver. 9 and earlier versions.

A.11 Features Added between Ver. 9.0 and Ver. 9.1

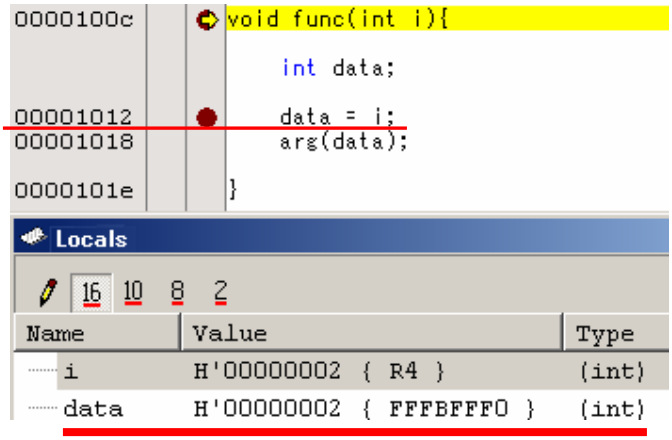
Additional output mode for debugging information (optimize=debug_only)

The optimize=debug_only option can be specified to always allow local variable information to be referenced during debugging. Optimization related to deletion of each statement can also be completely prevented, and break points can be set for each statement in the C source code. Since performance may degrade when this option is used to generate objects compared to when optimize=0 (no optimization) is specified, we recommend that it only be used temporarily during debugging.

[optimize=0]



[optimize=debug_only]



New additional specification for interrupts (SH-3, SH3-DSP, SH-4, SH-4A, SH4AL-DSP)

The following interrupt specification and embedded function have been added to allow executionally efficient interrupt functions to be coded using the C language.

Interrupt specification

#pragma interrupt sr_rts - Specifies register bank switching and RTS instruction return

This performs termination in the RTS instruction. This generates a save code for only the register used within the function, and sets the RB bit and BL bit of the SR register at the end of the function.

#pragma interrupt bank - Specifies the interrupt handling function

If the sr_jsr() embedded function exists, a save code for the SSR/SPC register is generated, and a save code is generated for the register used in the function.

#pragma interrupt rts - Specifies RTS instruction return

This performs termination in the RTS instruction, and suppresses save code output for SSR/SPC and registers R0 - R7.

A save code is generated for the register used in the function

Embedded function

sr_jsr(void * func, int imask) - Embedded function for controlling multiple interrupts

This clears the RB bit and BL bit for the SR register, sets imask for the interrupt mask, and calls the func function.

Example 1: When multiple interrupts are permitted

C source code

```
#include <machine.h>

// Handling function declaration
#pragma interrupt (func(bank))
void func(void);

// Interrupt function declaration
#pragma interrupt (sub (sr_rts))
void sub(void);

// Function definition
void func(void){
     :
 /* RB=0,RL=0
   Setting the interrupt level
   to 8 and calling sub() */

   sr_jsr(sub,8);
     :
}

void sub(void){
     :
}
```

Generated code

```
_func
                          <- When interrupt occurs, RB=1 and BL=1
  MOV.L    R14,@-R15
  STS.L    PR,@-R15
  STC      SSR,@-R15      } Save of registers other than R0 to R7 used in the
  STC      SPC,@-R15        function and SPC/SSR
     :
  STC     SR,R6
  MOV.L    L12+6,R1   ; H'CFFFFF0F
  MOV      #-128,R4   ; H'FFFFFF80
  EXTU.B   R4,R4                    } RB=0, BL=0 and IMASK=8
  MOV.L    L12+10,R14 ; _sub          changed
  AND      R1,R6
  OR       R4,R6
  LDC      R6,SR
  JSR      @R14       <- The sub() function is called, RB=1 and
  NOP                    BL=1 are set, and then returned.
     :
  LDC      @R15+,SPC
  LDC      @R15+,SSR
  LDS.L    @R15+,PR     } Restoration of registers other than R0 to R7
  MOV.L    @R15+,R14      used in the function and SPC/SSR
  RTE
  NOP

_sub:                     <- When this is called from func(), RB=0
                            and BL=0 are set.
  MOV.L    R0,@-R15
  MOV.L    R1,@-R15     } Save of only the registers used in the function
     :
  STC      SR,R0
  MOV.L    L12+2,R1 ; H'30000000
  OR       R1,R0
  MOV.L    @R15+,R1                 } RB=1 and BL=1 are set, and only
  LDC      R0,SR                      the registers used in the function
  RTS                                 are restored.
  LDC.L    @R15+,R0_BANK  <- Because RB=1 is set
```

Example 2: When multiple interrupts are not permitted

C source code

```
// Handling function declaration
#pragma interrupt (func(bank))
void func(void);

// Interrupt function declaration
#pragma interrupt (sub (rts))
void sub(void);

// Function definition
void func(void){
     :
   sub();
     :
}

void sub(void){
    :
}
```

Generated code

```
_func                     <- When interrupt occurs, RB=1 and BL=1

  STS.L    PR,@-R15     } Registers other than R0 to R7 used within the
                          function are saved. Since no sr_jsr() embedded
                          function exists, SPC/SSR is not saved.
     :
  MOV.L    L12,R14 ; _sub
  JSR      @R14
  NOP
     :
  LDS.L    @R15+,PR     } Registers other than R0 to R7 used within the
  RTE                     function are restored.
  NOP


_sub:

  MOV.L    R14,@-R15    } Registers other than R0 to R7 used within the
  MOV.L    R13,@-R15      function are saved.
     :
  MOV.L    @R15+,R13    } Registers other than R0 to R7 used within the
  RTS                     function are restored.
  MOV.L    @R15+,R14
```

Additional range checking omission function for conversion between floating-point decimals and integers - simple_float_conv option (SH-2E, SH2A-FPU, SH-4, SH-4A)

The simple_float_conv option can be specified to generate code to omit range checking for the converted value for conversion between unsigned integers and floating-point decimals.

This option can be used when the pre-conversion value is an integer from 0 to 2147483647, or a floating-point decimal from 0.0 to 2147483647.0. Keep in mind that the conversion results are not guaranteed when the pre-conversion value is out of range.

Example of generated code:

C source code

```
unsigned long func(float f)
{
      return((unsigned int)f);
}
```

When option is not used

```
  MOV      #79,R2    ; 0x0000004F
  SHLL8    R2
  SHLL16   R2        ; 0x4F000000
  LDS      R2,FPUL
  FSTS     FPUL,FR8
  FCMP/GT  FR4,FR8
  BT       L12
  FADD     FR8,FR8   ; when f ≥ 0x4F000000,
  FSUB     FR8,FR4   ; value before set is (f - 0x4F800000)

L12:
  FTRC     FR4,FPUL  ; Conversion from float to signed long
  STS FPUL,R0
```

When option is used

```
  FTRC      FR4,FPUL  ; Conversion from float to signed long
  STS       FPUL,R0
```

Additions and changes to existing functionality and specifications

(a) Extension of the division=cpu=inline option (SH-2A, SH2A-FPU)

division=cpu=inline can now be used even for cpu=sh2a|sh2afpu.

When division=cpu=inline is specified for SH-2A or SH2A-FPU, constant division is converted to multiplication and expanded inline, and variable division uses the DIVS or DIVU instruction. This is to improve the speed at which constant division is calculated, but keep in mind that object size may increase. If the speed or nospeed option (optimizing speed or size and speed) is specified, the division=cpu=inline option becomes the default.

Example of generated code:

<table>
<tr><th>C source code</th><th>cpu=sh2a, division=cpu=inline</th><th>cpu=sh2a, division=cpu=runtime</th></tr>
<tr><td rowspan="2">unsigned long A;
int func(void){
  A = A/10;
}</td><td>_func:
  MOV.L L11,R5 ; _A
  MOV.L L11+4,R1 ; H'CCCCCCCD
  MOV.L @R5,R6 ; A
  DMULU.L R6,R1
  STS MACH,R2
  SHLR2 R2
  SHLR R2
  MOV.L R2,@R5 ; A
  RTS
  MOV.L R2,@R5 ; A</td><td>_func:
  MOV.L L11,R5 ; _A
  MOV.L @R5,R6 ; A
  MOV #10,R0 ; H'0000000A
  DIVU R0,R6
  RTS
  MOV.L R6,@R5 ; A</td></tr>
<tr><td>Cycle count: 12
Size: 0x1C</td><td>Cycle count: 38
Size: 0x10</td></tr>
</table>

(b) Extension of targets for embedded functions for cache block operation (SH-4)

The embedded functions for cache block operations ocbi(), ocbp(), and ocbwb(), can now be used with SH-4.

(c) Changes to the #pragma inline specification

The specification for the inline option for functions for which #pragma inline is specified has been changed.

Old specification:

If the inline option is specified, functions with #pragma inline specified follow the value set for the inline option. (Note)

(Note) If the speed option (optimization on speed) is specified, inline=20 is specified by default.

New specification:

Inline expansion is performed for functions with #pragma inline specified, regardless of the value set for the inline option.

(d) Displaying options within sub-commands in the compile list

If a sub-command file is specified during compilation, the options specified in the sub-command file are now output to the compile list. This means that when the Renesas Integrated Development Environment (High-performance Embedded Workshop) is used to output the compile list, compile options are output to the compile list.

Performance improvements for the mathematical function library (SH-1, SH-2, SH2-DSP, SH-2A, SH-3, SH3-DSP, SH4AL-DSP)

Object sizes have been reduced and calculation speed and accuracy have been improved for the sinf, cosf, tanf, expf, logf, sqrtf, and atanf mathematical functions for floating point decimals.

Note that since the calculation accuracy has been improved, the calculation results from these mathematical functions may differ from those in Ver. 9.00.

Benchmark comparison V. 9.00 / V. 9.01 (cycle count)

Library function	V.9.00 – SH-2	V.9.01– SH-2	V.9.00 – SH-2A	V.9.01– SH-2A
sinf	2497	477	1001	169
cosf	2434	465	954	162
tanf	3196	705	1806	274
atanf	3160	515	1602	218
logf	3816	491	1720	232
sqrtf	1018	236	562	109
expf	4432	469	1463	192

Improved debugging information

(e) Referenced parameters
The case in which "Not available now" is displayed for a variable when a function is entered has been improved.

(f) Variables for which #pragma address is specified
Variables using #pragma address can now be referenced as symbols.

(g) Delection of unnecessary type information
Debugging information for types not referenced in C/C++ source files has been deleted to decrease object file size.

Improved messages

(h) Information messages for uninitialized variables
When an uninitialized variable is used in C source code, the message "C0011 (I) Used before set symbol : *variable-name* in *function-name*" is now output. "C5549 (I) Variable "*variable-name*" is used before its value is set" is output to the C++ source code.

(i) Information messages for paths without return statements
The message "C0017 (I) Missing return statement" is output now even when there are multiple function entrances and only one path is missing a return statement.

Contents added and improved in the optimization linkage editor

(j) Optimization suppression functionality by section (supported from V.9.01 of the optimization linker, and V.9.00R04(a) of the package)
The section_forbid option can now be used to suppress inter-module operation by section.

(k) Improved overlay functionality (supported from V.9.01 of the optimization linker, and V.9.00R04(a) of the package)

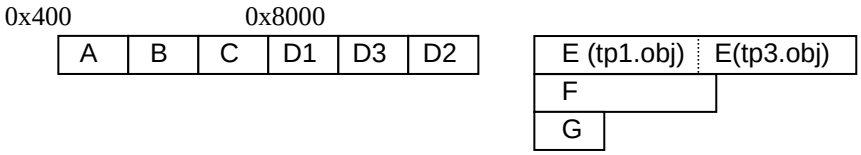
Parentheses "()" can now be used for the start option. This allows more complex overlay positioning than previous versions.

Examples:

When .obj files are input in the following order (the parentheses contain the section for each .obj)

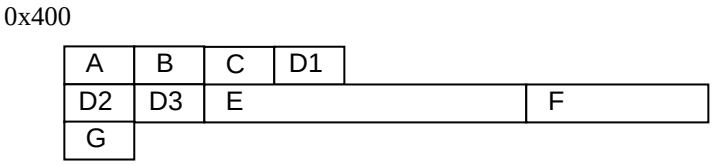
tp1.obj(A,D1,E) -> tp2.obj(B,D3,F) -> tp3.obj(C,D2,E,G)

(1) -start=A,B,C,D*/400,E:F:G/8000



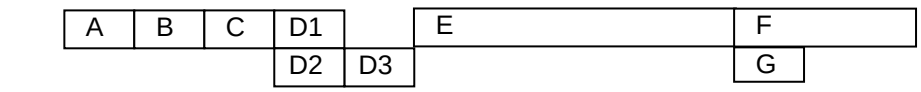
- The E, F, and G sections separated by ":" are allocated to the same address.
- Sections specified by wildcards (sections starting with "D" in this example) are allocated in the order input.
- Items within sections of the same name (section E in this example) are allocated in the order in which objects are input.

(2) -start=A,B,C,D1:D2,D3,E,F:G/400

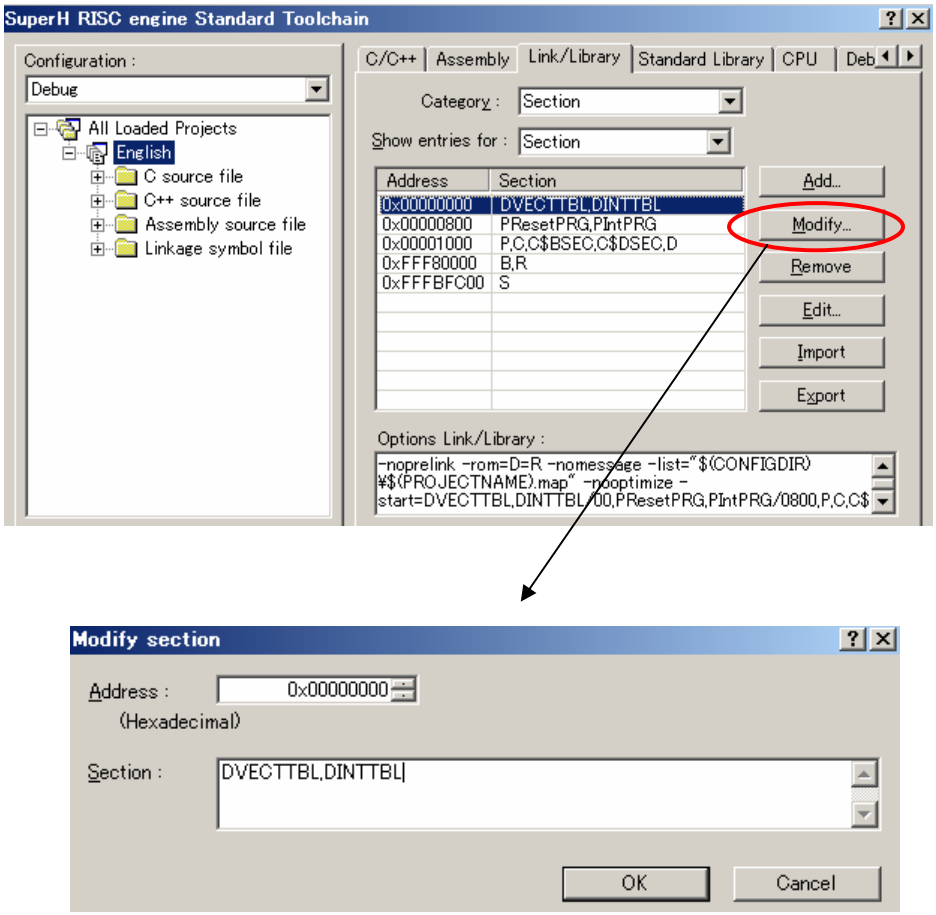


- Starting with sections immediately after separation by ":" (A, D2, and G in this example), each start is allocated to the same address.

(3) -start=A,B,C,(D1:D2,D3),E,(F:G)/400



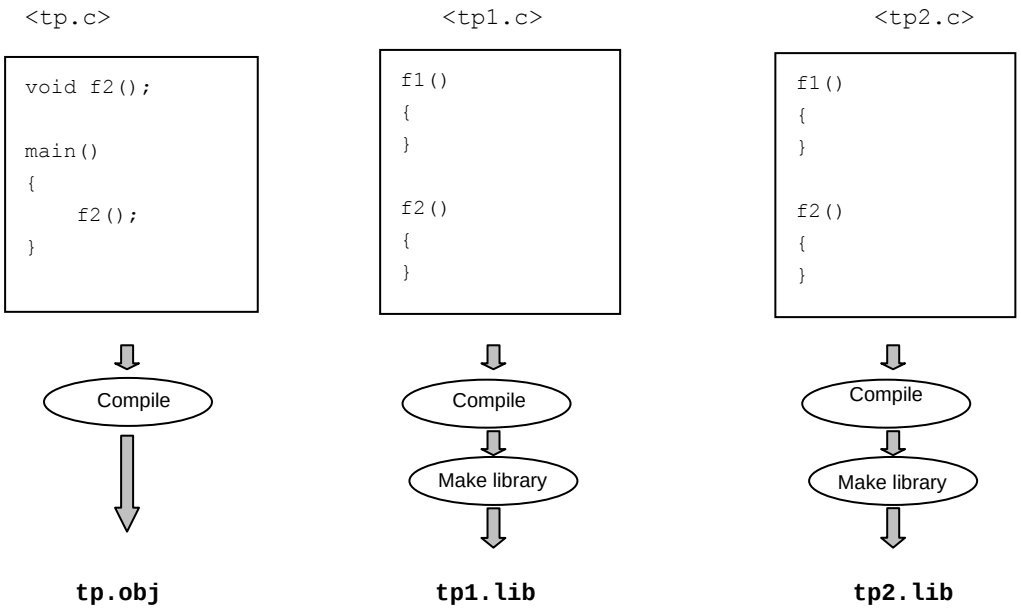
- This is a new method for specifying overlays.
- If the same address placement is enclosed in "()", the same address placement within the "()" is performed as immediately after the beginning of the section immediately before the "()" (C and E in this case).
- The section immediately after the "()" (E in this case) is placed immediately after the last section enclosed in "()".
- When the Renesas Integrated Development Environment is used, the new overlay specification method cannot be performed from **Edit**, only from **Refresh**.


SuperH RISC engine Standard Toolchain
Configuration :
Debug
All Loaded Projects
English
C source file
C++ source file
Assembly source file
Linkage symbol file
C/C++
Assembly
Link/Library
Standard Library
CPU
Deb
Category :
Section
Show entries for :
Section
Address
Section
0x00000000
DVECTTBL,DINTTBL
0x00000800
PResetPRG,PIntPRG
0x00001000
P,C,C$BSEC,C$DSEC,D
0xFFF80000
B,R
0xFFFBFC00
S
Add...
Modify...
Remove
Edit...
Import
Export
Options Link/Library :
-noprelink -rom=D=R -nomessage -list="$(CONFIGDIR)
¥$(PROJECTNAME).map" -nooptimize -
start=DVECTTBL,DINTTBL/00,PResetPRG,PIntPRG/0800,P,C,C$
Modify section
Address :
0x00000000
(Hexadecimal)
Section :
DVECTTBL,DINTTBL
OK
Cancel

(l) Notification of same-name symbols within a library (supported from V.9.01 of the optimization linker, and V.9.00R04(a) of the package)

When multiple symbols of the same name (variables or functions with the same name) exist within a library file used during linkage, the warning message "L1320 (W) Duplicate Symbol "*symbol*" in "*library* (*module*)"" is displayed. Keep in mind that this message may be output on environments to which the L1320 message wasn't previously output. Note that since this message is output for all modules containing symbols of the same name, it may be output frequently. If this makes it difficult to confirm other messages, specify the nomessage=1320 option to suppress L1320 messages.

Example of message output for V.9.01 and later:




```
Link option:
   optlnk tp.obj -lib=tp1.lib,tp2.lib -start=P/0

Message output for the above link:
  ** L1320 (W) Duplicate symbol "_f1" in "tp1.lib(tp1)"
  ** L1320 (W) Duplicate symbol "_f1" in "tp2.lib(tp2)"
  ** L1320 (W) Duplicate symbol "_f2" in "tp1.lib(tp1)"
  ** L1320 (W) Duplicate symbol "_f2" in "tp2.lib(tp2)"
```

(m)Functionality to detect redundancy during physical space placement (supported from V.9.02 of the optimization linker, and V.9.01R00 of the package)

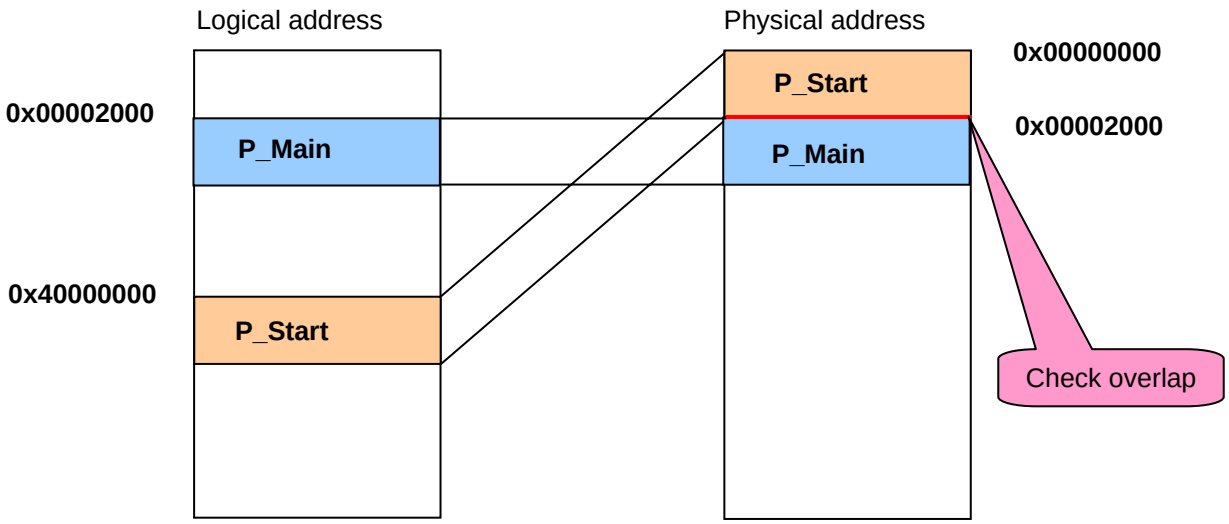
An option (ps_check) has been added to detect objects that overlap when SH3 or SH4 is used and objects overlap not for logical addresses but in actual memory. This option can be used to terminate linkage processing with an error, when redundancy is detected.

Example:

In SH4, when an invalid status exists for MMU, 4 GB address spaces are mapped to 512 MB (29 bit) external memory spaces (the 3 bits beyond the 4 GB addressing are ignored during mapping).
For example, the object overlapping in which the usable U0 space (00000000 to 0x7fffffff) is mapped to external memory (512 MB) for the user mode can be detected as follows:

-PS_check=00000000-1fffffff,20000000-3fffffff,40000000-5fffffff,60000000-7fffffff

When this option is specified, all 00000000, 20000000, 40000000, and 60000000 locations are placed in the same location in memory.



(n) Functionality to specify the byte count for data records (supported from V.9.02 of the optimization linker, and V.9.01R00 of the package)

The byte_count option can now be used to change the maximum byte count for data records in Intel HEX-format files.

(o) Functionality to fill in free areas with random numbers (supported from V.9.02 of the optimization linker, and V.9.01R00 of the package)

Functionality to fill random numbers using an option (space) for specifying free area output has been added.

(p) Extension of options to reduce amount of memory used (supported from V.9.02 of the optimization linker, and V.9.01R00 of the package)

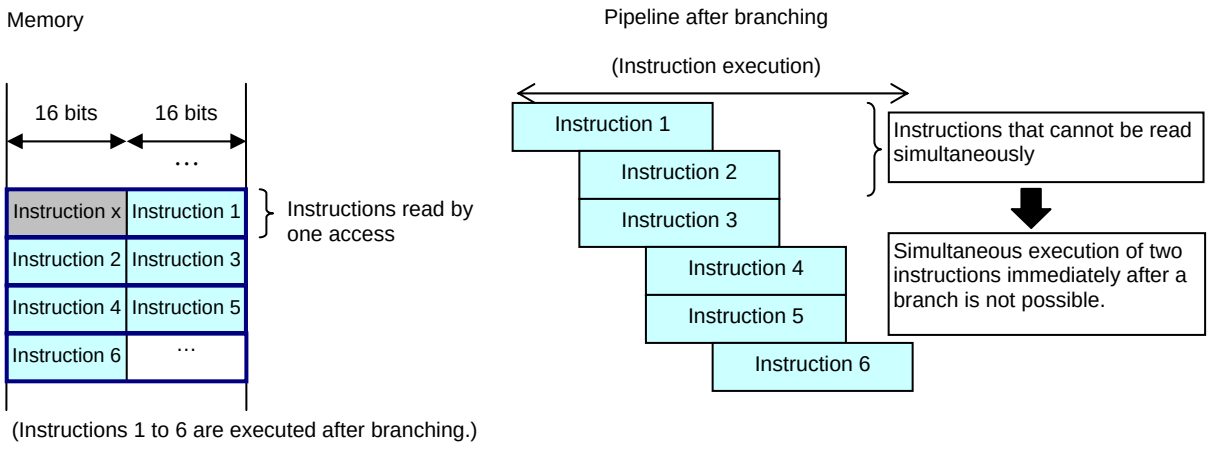
The option (memory=low) for functionality to reduce the amount of memory used can now be used during library file creation.

A.12 Features Added between Ver. 9.1 and Ver. 9.2

(1) Aligning branch destination addresses on a 4-byte boundary (#pragma align4, -align4)

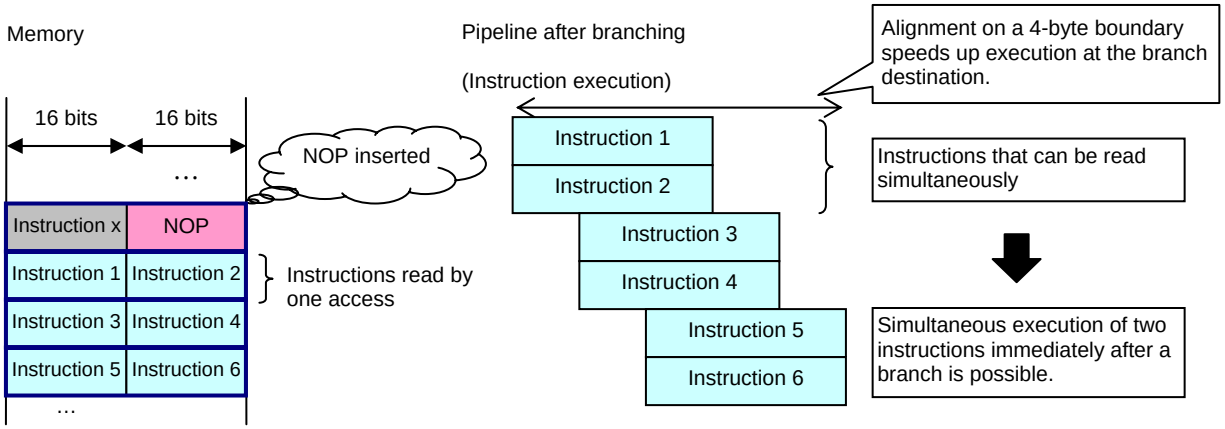
A feature that aligns branch destination addresses on a 4-byte boundary has been added. The SH CPU reads instructions in 32-bit units. Since the basic SH CPU instructions are 16 bits long, the SH CPU can read two instructions simultaneously when there are consecutive 16-bit instructions. However, if a branch instruction exists, the SH CPU can simultaneously read two instructions at the branch destination only when the branch destination addresses have been aligned on a 4-byte boundary. If the branch destination addresses have not been aligned on a 4-byte boundary, processing efficiency at the branch destination might be degraded. The feature that aligns branch destination addresses on a 4-byte boundary allows the SH CPU to always read two instructions simultaneously at a branch destination. This results in fewer memory accesses after a branch, speeding up processing. For CPUs having the superscalar architecture (e.g., SH-2A, SH2A-FPU, SH-4, SH-4A, and SH4AL-DSP), parallel execution performance of instructions at a branch destination might also be improved. This feature is particularly effective for branches executed many times, such as a frequently called function conditional statement and a loop with many iterations.

When branch destination addresses are not aligned on a 4-byte boundary (superscalar CPU):



(Instructions 1 to 6 are executed after branching.)

When branch destination addresses are aligned on a 4-byte boundary (superscalar CPU):



(Instructions 1 to 6 are executed after branching.)

Since this feature might insert NOP (No Operation) instructions to align branch destination addresses on a 4-byte boundary, processing efficiency might be adversely affected due to an increase in object size. For this reason, one of the feature's three scopes can be selected to reduce the negative impact. Since the most effective scope depends on the program, selection should be made only after trying all three in the user system.

The following shows the formats of the **align4** option and **#pragma instruction**:

- Option : ALIGN4={ALL|LOOP|INMOSTLOOP}
- #pragma : #pragma align4 [()<function name>=<scope>[,...][)]

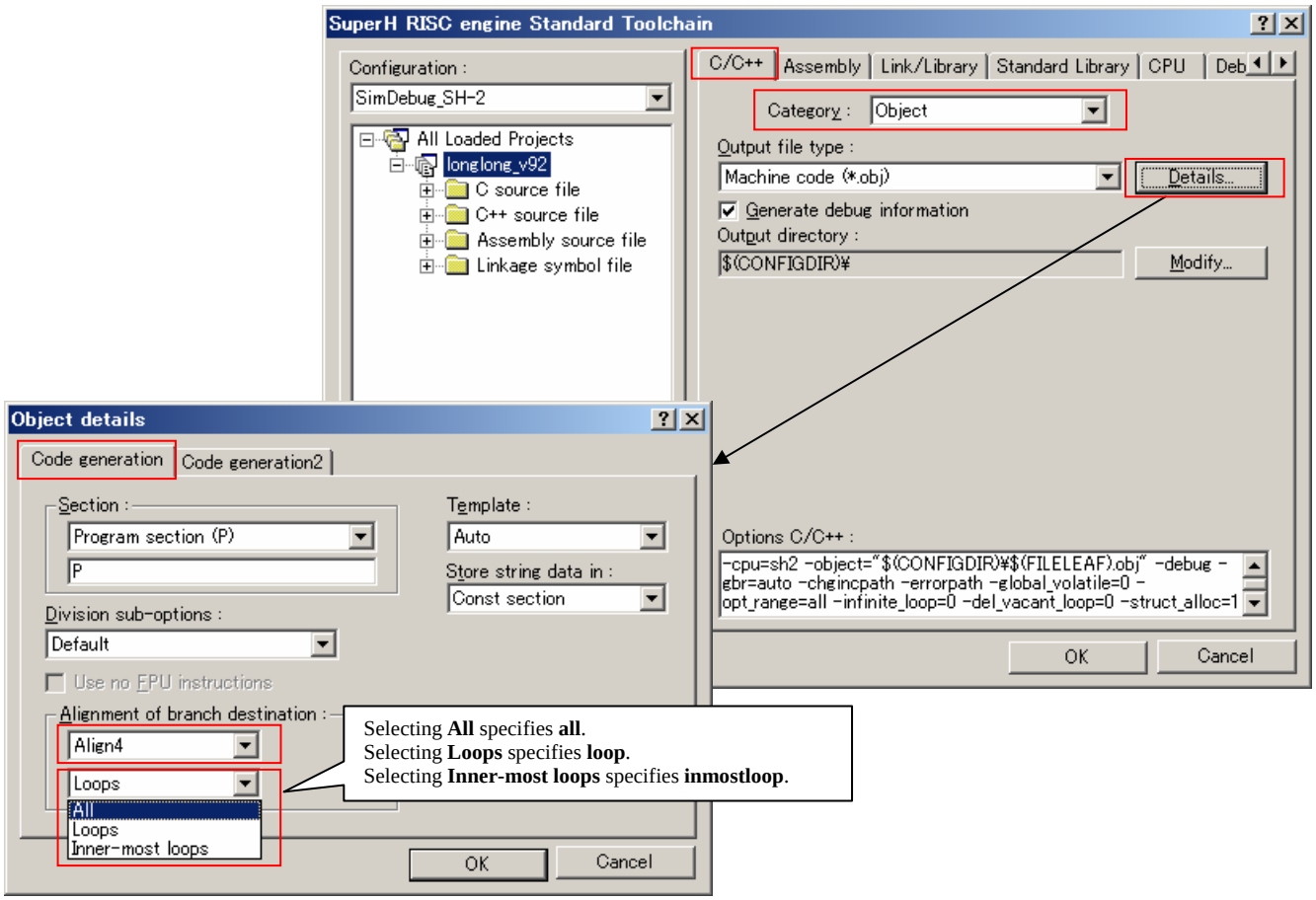
Specified keyword		Scope of alignment on a 4-byte boundary	Increase in size
Option	**#pragma**		
ALL	all	All branch destination addresses	Large
LOOP	loop	The start addresses of all loops	
INMOSTLOOP	inmostloop	The start addresses of the innermost loops	Small

To specify the **align4** option in HEW, specify the following settings on the **C/C++** page in the **SuperH RISC engine Standard Toolchain** dialog box:

Category: Select **Object**.

Details: Click this button to display the **Object details** dialog box, and specify the following setting on the **Code generation** page:

Alignment of branch destination: Select **Align4**, and a scope.



Example:

The following shows an execution example when **align4=all** is specified for SH-2A.

Sample Source:

```
int a[64],b[64];
int c;
void main() {
    int i;

    for(i=0;i<64;i++) {
       a[i] = b[i];
       c++;
    }
}
```

Disassemble List (align4 not specified):

```
_main:
          MOV.L       L13,R7      ; _c
          MOV.L       @R7,R6      ; c
          MOV         #64,R1      ; H'00000040
          MOV.L       L13+4,R4    ; _a
          MOV.L       L13+8,R5    ; _b
L11:
          MOV.L       @R5+,R0     ; b[]
          DT          R1
          ADD         #1,R6
          BF/S        L11
          MOV.L       R0,@R4+     ; a[]
          RTS
          MOV.L       R6,@R7      ; c
L13:
          .DATA.L     _c
          .DATA.L     _a
          .DATA.L     _b
```

Disassemble List (align4=all specified):

```
_main:
          MOV.L       L13+2,R7    ; _c
          MOV.L       @R7,R6      ; c
          MOV         #64,R1      ; H'00000040
          MOV.L       L13+6,R4    ; _a
          MOV.L       L13+10,R5   ; _b
          NOP
L11:
          MOV.L       @R5+,R0     ; b[]
          DT          R1
          ADD         #1,R6
          BF/S        L11
          MOV.L       R0,@R4+     ; a[]
          RTS
          MOV.L       R6,@R7      ; c
L13:
          .RES.W      1
          .DATA.L     _c
          .DATA.L     _a
          .DATA.L     _b
```

The size of the code and the execution speed during execution of the above sample source are as follows:

	Code size (bytes)		Execution speed (cycles)	
CPU type	align4 not specified	align4=all specified	align4 not specified	align4=all specified
SH-2A	36	38	393	330

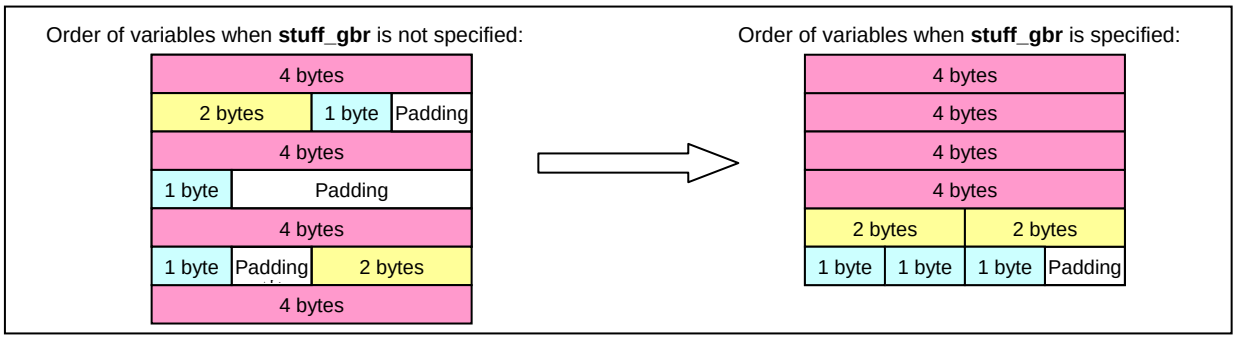
Supplementary notes:

- Functions whose branch destination addresses have been aligned on a 4-byte boundary are not optimized during linkage.
- If both the **align4**, **align16**, or **align32** option and the **#pragma align4** instruction are specified, the **#pragma align4** instruction takes precedence.
- If the **align4** option is specified together with the **align16** or **align32** option, the following error message is output: C3305 (F) Invalid command parameter "<option name>"

- If the **#pragma align4** instruction is specified incorrectly, the following error message is displayed:
  - When **#pragma align4** is specified twice with two different scopes for the same function:
    C2806 (E) Multiple #pragma for one function
  - When the function specified in **#pragma align4** is defined before **#pragma align4** is declared:
    C2857 (E) Function "<function name>" in #pragma already declared
  - When a symbol that is not a function is specified in **#pragma align4**:
    C2858 (E) Illegal #pragma "<identifier>" function type
  - When the **#pragma align4** specification contains a syntax error:
    C2859 (E) Illegal #pragma "<identifier>" declaration

(2) Allocating variables in order of data size in GBR areas ($G0 and $G1 sections) (stuff_gbr)

The **stuff_gbr** option has been added. When this option is specified, variables with **#pragma gbr_base** or **#pragma gbr_base1** specified are allocated in specific sections according to variable size. This type of allocation can reduce the unused area inserted as padding for boundary alignment, thereby conserving memory.



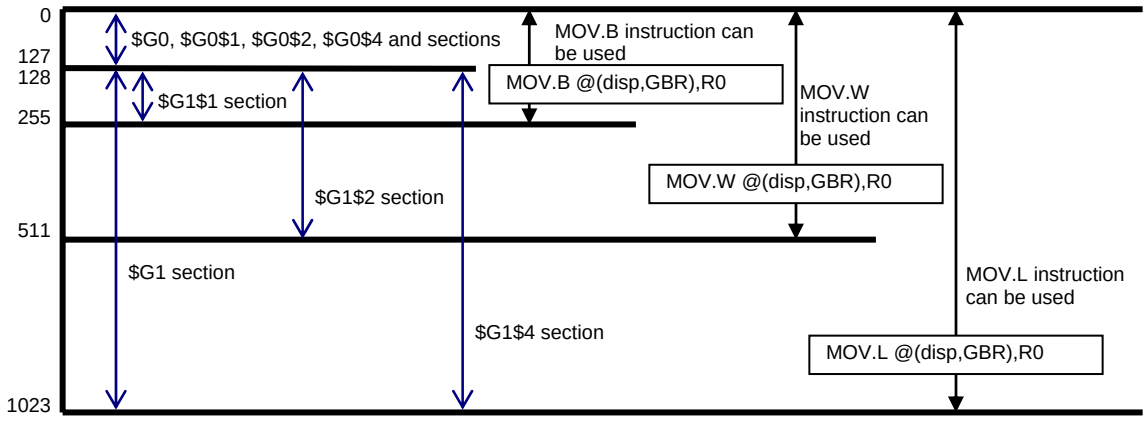
Variables with **#pragma gbr_base** or **#pragma gbr_base1** specified are allocated in different sections based on size, as shown below. If GBR-relative instructions are output, however, an empty $G0 section is generated even when the **stuff_gbr** option is specified.

	stuff_gbr specified			
	Variable size (bytes)			**stuff_gbr not specified**
	4n	***4n* + 2**	***2n* + 1**	
Variable with #pragma gbr_base specified	$G0$4	$G0$2	$G0$1	$G0
Variable with #pragma gbr_base1 specified	$G1$4	$G1$2	$G1$1	$G1

The **stuff_gbr** option takes effect only when the **gbr=user** option is specified. If the **gbr=user** option is not specified, the following warning message is displayed and the **stuff_gbr** option is ignored:

C1301 (W) "stuff_gbr" option ignored

A variable with **#pragma gbr_base** or **#pragma gbr_base1** specified is accessed using a MOV instruction based on a relative value (offset) from the address set in the GBR register. For a variable with **#pragma gbr_base** specified, the possible offset is 0 to 127 bytes. For a variable with **#pragma gbr_base1** specified, the possible offset differs depending on the type of the variable. For a variable of type "char" or "unsigned char", which is accessed using the MOV.B instruction, the possible offset is 128 to 255 bytes. For a variable of type "short" or "unsigned short", which is accessed using the MOV.W instruction, the possible offset is 128 to 511 bytes. For a variable of type "int", "unsigned int", "long", "unsigned long", "float", or "double", which is accessed using the MOV.L instruction, the possible offset is 128 to 1023 bytes.



Therefore, each section must be allocated within a specific range as shown in the following table.

Section name	Section allocation range
$G0	At the address set in the GBR register.
$G0$1, $G0$2, or $G0$4	At addresses within 127 bytes of the start address of the $G0 section.
$G1	At the address 128 bytes from the start address of the $G0 section.
$G1$1	At an address within 255 bytes of the start address of the $G0 section following the $G1 section.
$G1$2	At an address within 511 bytes of the start address of the $G0 section following the $G1 section.
$G1$4	At an address within 1023 bytes of the start address of the $G0 section following the $G1 section.

If sections are not allocated as explained in the above table, the following error message is output during linkage:

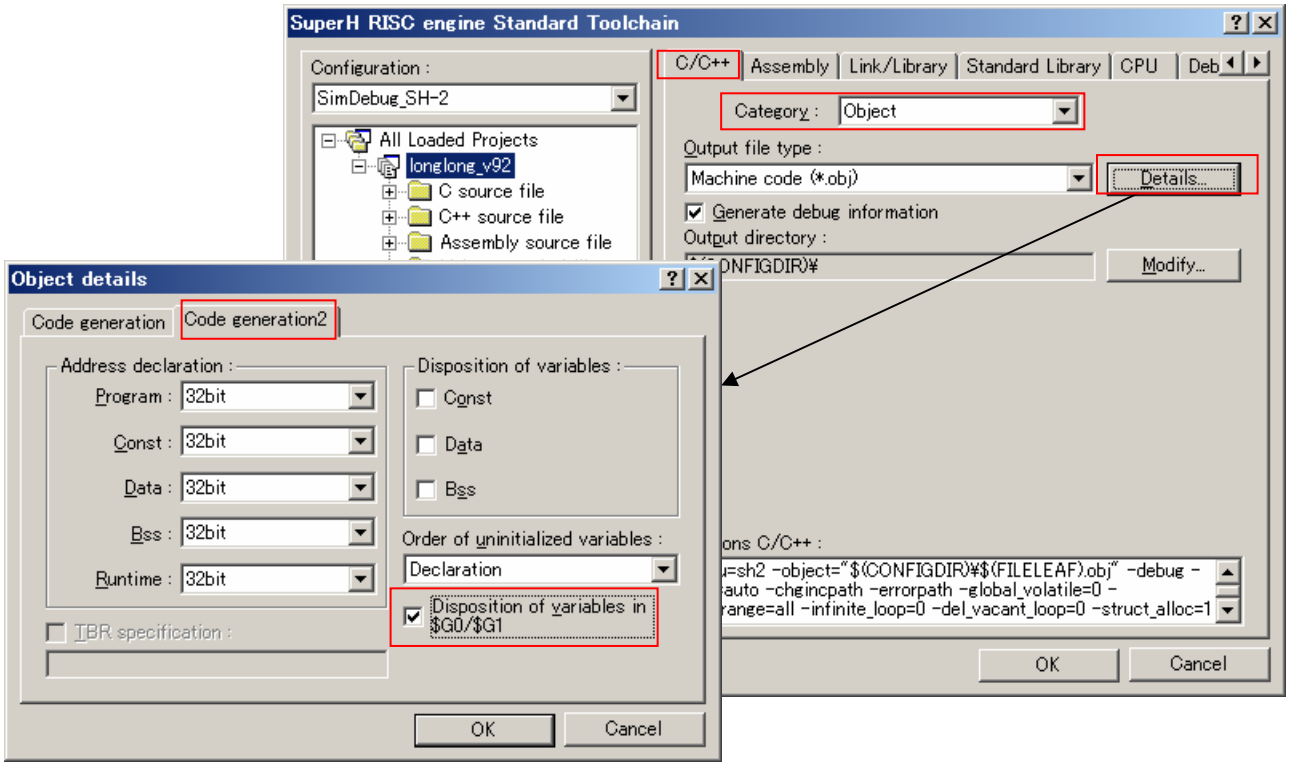
L2330 (E)Relocation size overflow

If the **stuff_gbr** option is specified in HEW, specify the following settings on the **C/C++** page in the **SuperH RISC engine Standard Toolchain** dialog box:

Category: Select **Object**.

Details: Click this button to display the **Object details** dialog box. On the **Code generation2** page, specify the following setting:

Disposition of variables in $G0/$G1: Select this check box.



Example:

Sample Source:

```
#pragma gbr_base(a,b,c,d,e,f,g,h,i)

long  a;
short b;
char  c;
long  d;
char  e;
long  f;
char  g;
short h;
long  i;
```

Disassemble List (stuff gbr not specified):

```
          .SECTION    $G0,DATA,ALIGN=4
_a:                                ; static: a
          .DATAB.L    1,0
_b:                                ; static: b
          .DATAB.W    1,0
_c:                                ; static: c
          .DATAB.B    1,0
          .RES.B      1
_d:                                ; static: d
          .DATAB.L    1,0
_e:                                ; static: e
          .DATAB.B    1,0
          .RES.B      1
          .RES.W      1
_f:                                ; static: f
          .DATAB.L    1,0
_g:                                ; static: g
          .DATAB.B    1,0
          .RES.B      1
_h:                                ; static: h
          .DATAB.W    1,0
_i:                                ; static: i
          .DATAB.L    1,0
```

Disassemble List (stuff gbr specified):

```
          .SECTION    $G0$4,DATA,ALIGN=4
_a:                                ; static: a
          .DATAB.L    1,0
_d:                                ; static: d
          .DATAB.L    1,0
_f:                                ; static: f
          .DATAB.L    1,0
_i:                                ; static: i
          .DATAB.L    1,0
          .SECTION    $G0$2,DATA,ALIGN=2
_b:                                ; static: b
          .DATAB.W    1,0
_h:                                ; static: h
          .DATAB.W    1,0
          .SECTION    $G0$1,DATA,ALIGN=1
_c:                                ; static: c
          .DATAB.B    1,0
_e:                                ; static: e
          .DATAB.B    1,0
_g:                                ; static: g
          .DATAB.B    1,0
```

Boundary adjustment area: 5 bytes

Note:

For a union, structure, or array, use care in specifying the section allocation address. If a union, structure, or array is specified in **#pragma gbr_base1**, a link error might occur even when sections have been allocated as explained above. For example, if **#pragma gbr_base1** is specified for "char x[4]" as shown in the following sample source, although 4-byte array "x" is allocated in the $G1$4 section, the MOV.B instruction is used when array "x" is referenced. Therefore, to prevent a linkage error, the offset of the $G1$4 section in which array "x" is allocated must be within 128 to 255 bytes, not within 128 to 1023 bytes.

Sample Source:

```
#pragma gbr_base1(x)

char x[4];

void func(void)
{
    x[0] = 1;
}
```

Disassemble List:

```
          .SECTION    P,CODE,ALIGN=4
_func:                            ; function: func
          MOV         #1,R0       ; H'00000001
          RTS
          MOV.B       R0,@( x-(STARTOF $G0),GBR); x[]
          .SECTION    $G1$4,DATA,ALIGN=4
_x:                               ; static: x
          .DATAB.B    4,0
          .SECTION    $G0,DATA,ALIGN=4
```

(3) Preventing inline expansion in C++ (cpp_noinline)

C++-specific inline expansion can now be prevented by specifying the **cpp_noinline** option. There are the following two types of C++-specific inline expansion:

- Inline expansion of a function with specifier inline

Sample Source:

```
int x;
inline int func(int a){
    return a * 3;
}
void g(void){
     x = func(x);
}
```

After Inline Expansion:

```
void g(void) {
     x = x * 3;
}
```

- Inline expansion of a member function defined in a class

Sample Source:

```
class A {
   int cx, cy;
public:
   int func() {
       return cx + cy;
   }
};
int g(class A a) {
    return = a.func();
}
```

After Inline Expansion:

```
int g(class A a) {
     return (a.cx + a.cy);
}
```

Specifying the **cpp_noinline** option does not prevent the following types of inline expansion, which are also effective in C:

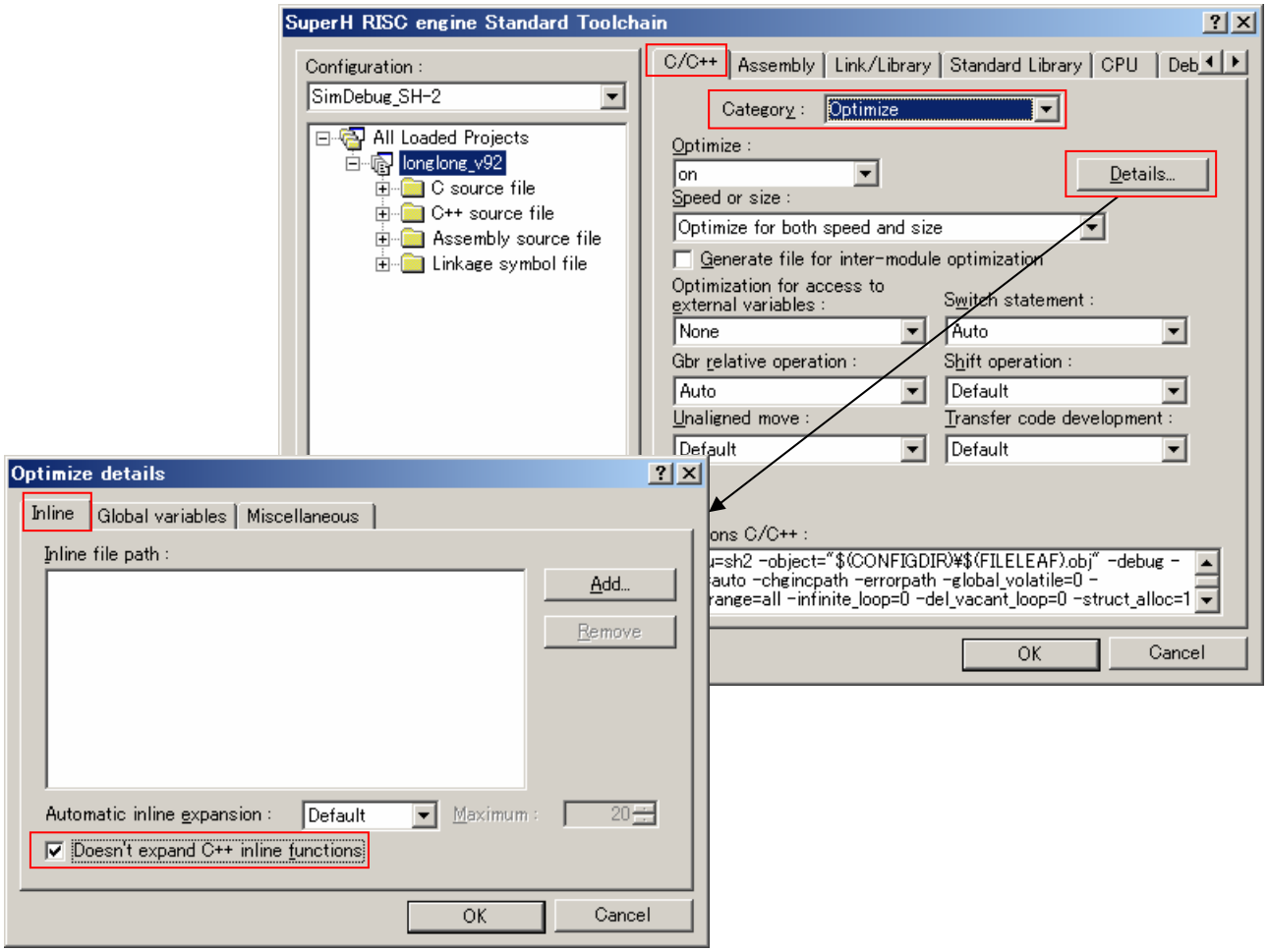
- Automatic inline expansion of a function performed by specifying the **inline** option
- Inline expansion of a function performed by specifying **#pragma inline**

To specify the **cpp_noinline** option in HEW, specify the following settings on the **C/C++** page in the **SuperH RISC engine Standard Toolchain** dialog box:

Category: Select **Optimize**.

Details: Click this button to display the **Optimize details** dialog box. On the **Inline** page, specify the following setting:

Doesn't expand C++ inline functions: Select this check box.



(4) Added or changed specifications for existing features

(a) Changes to the inline expansion suppression conditions that apply when **#pragma inline** is specified

The inline expansion suppression conditions that apply when **#pragma inline** is specified have been changed. Previously, the increase rate for the size of the function to be inline-expanded was used as a condition for suppressing inline expansion. This condition has now been abolished. However, the other conditions, such as increase of compilation time and increase of memory usage, still exist. Accordingly, functions with **#pragma inline** specified are not always inline-expanded.

Note that if the **scope** option, which takes effect by default, is used to divide the scope of optimization, inline expansion might be suppressed. In this case, specifying the **noscope** option ensures that inline expansion is performed.

(b) Prototype declaration for intrinsic functions

Prototype declaration for intrinsic functions is now also performed in C.

(5) Improved messages

(a) Changes to the message for division by zero

For a function that includes an expression whose divisor is zero, the following warning message is now output:

C1501 (W) Division by zero

```
Sample Source:

int a,b;

void main(){
    a = b/0;  /* C1501 (W) Division by zero */
}
```

```
Error Message:

test.c(4) : C1501 (W) Division by zero
```

Supplementary note:

In C++, the following warning message is output:

C5039 (W) Division by zero

(6) Enhanced optimization

(a) Instruction expansion for "long long" type operations

Optimization has been enhanced so that an "(unsigned) long long" operation that was hitherto processed by calling a runtime routine is now processed by instructions. Note that this enhancement applies conditionally. If one or more of the necessary conditions is not satisfied, a runtime routine call is used. The following shows the conditions.

Division and modulo operations

All of the following conditions must be satisfied:

- The low-order 32 bits of the divisor are 0s that are constants.
- The CPU is SH-2A or SH2A-FPU. Alternatively, the divisor is not 0.
- The high-order 32 bits of the divisor are not "-1" (not 0xFFFFFFFF00000000 in a "signed long long" operation).
- An option for which "Instruction expansion" is indicated in the following table has been specified:

CPU	Division method selection	Execution speed/ size optimization	Instruction expansion/ runtime routine
SH-1	-	-	Runtime routine
SH-2A or SH2A-FPU	-	-	Instruction expansion
Other CPUs	division=cpu=inline	-	Instruction expansion
	division=cpu	speed	Instruction expansion
		nospeed	Instruction expansion
		size	Runtime routine
	division=cpu=runtime	-	Runtime routine

Comparison operation

The expression is an inequality (<, >, <=, or >=) with 0 or an equality (== or !=).

Shift operation

The number of shifted bits is the constant 1, 8, 16, or 32.

Cast operation

The cast is from a type other than "float" and "double". Alternatively, the cast is to a type other than "float" and "double".

Example:

With the SH-2, an addition operation of type "long long" is expanded to instructions as follows.

Sample Source:

```
long long x, y;
long long  func(void) {
    return x + y;
}
```

Disassemble List Generated in V.9.1:

```
_func:
          STS.L       PR,@-R15
          MOV.L       L11+2,R4    ; _y
          MOV.L       @(4,R4),R1 ; (part of)y
          MOV.L       @R4,R2      ; (part of)y
          MOV.L       R1,@-R15
          MOV.L       R2,@-R15
          MOV.L       L11+6,R6    ; _x
          MOV.L       @(4,R6),R4 ; (part of)x
          MOV.L       @R6,R5      ; (part of)x
          MOV.L       R4,@-R15
          MOV.L       R5,@-R15
          MOV.L       @(20,R15),R7
          MOV.L       L11+10,R2   ; __add64
          JSR         @R2
          MOV.L       R7,@-R15
          ADD         #20,R15
          LDS.L       @R15+,PR
          RTS
          NOP
L11:
          .RES.W      1
          .DATA.L     _y
          .DATA.L     _x
          .DATA.L     __add64
```

Disassemble List Generated in V.9.2:

```
_func:
          MOV.L       L11+2,R1    ; _x
          MOV.L       L11+6,R5    ; _y
          MOV.L       @(4,R1),R4 ; (part of)x
          MOV.L       @(4,R5),R7 ; (part of)y
          MOV.L       @R1,R6      ; (part of)x
          MOV.L       @R5,R1      ; (part of)y
          CLRT
          ADDC        R7,R4
          MOV.L       @R15,R2
          ADDC        R1,R6
          MOV.L       R6,@R2
          RTS
          MOV.L       R4,@(4,R2)
L11:
          .RES.W      1
          .DATA.L     _x
          .DATA.L     _y
```

The following table shows the code size and execution speed of the sample source when x = 1 and y = 1.

	Code size (bytes)		Execution speed (cycles)	
CPU type	**V9.1**	**V9.2**	**V9.1**	**V9.2**
SH-2	50	34	77	27

(b) Use of CLIP instructions (SH-2A and SH2A-FPU)

The SH-2A or SH2A-FPU provides saturation value comparison instructions (CLIPS.B, CLIPS.W, CLIPU.B, and CLIPU.W). Before the enhancement, intrinsic functions (clipsb(), clipsw(), clipub(), and clipuw()) had to be used to generate these instructions. Now, these instructions are generated during saturation evaluation processing without the intrinsic functions.

Example:

In the following example, the CLIPS.B instruction is generated.

Sample Source (1):

```
void func(long a) {
    …
    if (a > 127) {
        a = 127;
    }
    else if (a <  -128) {
        a = -128;
    }
    …
}
```

Sample Source (2):

```
void func(long a) {
    …
    if (a > 127) {
        a = 127;
    }
    if (a < -128) {
        a = -128;
    }
    …
}
```

Sample Source (3):

```
void func(long a) {
    …
    a = (a < 127) ? a : 127;
    a = (a > -128) ? a : -128;
    …
}
```

Sample Source (4):

```
void func(long a) {
    …
    a = (a < 127) ? ((a > -128) ? a : -128) : 127;
    …
}
```

Disassemble List:

```
_func:
                …
          CLIPS.B     R4      ; a
                …
```

- In the following example, the CLIPU.B instruction is generated.

Sample Source (1):

```
void func(unsigned long a) {
    …
    if (a > 255) {
        a = 255;
    }
    …
}
```

Sample Source (2):

```
void func(unsigned long a) {
    …
    a = (a < 255) ? a : 255;
    …
}
```

Disassemble List:

```
_func:
                …
          CLIPU.B     R4      ; a
                …
```

- In the following example, the CLIPS.W instruction is generated.

Sample Source (1):

```
void func(long a) {
    ...
    if (a > 32767) {
        a = 32767;
    }
    else if (a <  -32768) {
        a = -32768;
    }
    ...
}
```

Sample Source (2):

```
void func(long a) {
    ...
    if (a > 32767) {
        a = 32767;
    }
    if (a < -32768) {
        a = -32768;
    }
    ...
}
```

Sample Source (3):

```
void func(long a) {
    ...
    a = (a < 32767) ? a : 32767;
    a = (a > -32768) ? a : -32768;
    ...
}
```

Sample Source (4):

```
void func(long a) {
    ...
    a = (a < 32767) ? ((a > -32768) ? a : -32768) :
32767;
    ...
}
```

Disassemble List:

```
_func:
                ...
          CLIPS.W     R4      ; a
                ...
```

- In the following example, the CLIPU.W instruction is generated.

Sample Source (1):

```
void func(unsigned long a) {
    ...
    if (a > 65535) {
        a = 65535;
    }
    ...
}
```

Sample Source (2):

```
void func(unsigned long a) {
    ...
    a = (a < 65535) ? a : 65535;
    ...
```

Disassemble List:

```
_func:
                ...
          CLIPU.W     R4      ; a
                ...
```

The following table lists the saturation comparison instructions and the variable types for which a instruction is generated during saturation evaluation.

Instruction	Variable type for which the instruction is generated
CLIPS.B	long, int, and short
CLIPU.B	unsigned long, unsigned int, and unsigned short
CLIPS.W	long and int
CLIPU.W	unsigned long and unsigned int

Supplementary notes:

- The CLIPS.B and CLIPS.W instructions are generated even when upper-limit evaluation and lower-limit evaluation are performed in reverse order.
- A saturation comparison instruction is generated even when a comparison operator that includes an equal sign (such as ">=") is used for upper-limit evaluation or lower-limit evaluation.
- A saturation comparison instruction is generated when there is assignment of the upper-limit value or lower-limit value. However, no instruction is generated when the upper-limit value or lower-limit value is directly returned by using the "return" statement.
- A saturation comparison instruction is not generated when there is processing that does not assign the upper-limit value or lower-limit value to the variable subject to saturation evaluation, such as assignment to a variable that is not related to saturation evaluation.

(7) Major improvements to the optimizing linkage editor

(a) Display of the total section size of ROM and RAM areas (**total_size** and **show=total_size**)
(Optimizing Linker 9.03 and later, and Package V.9.01 Release 01 and later)

Specifying the **total_size** option outputs the total section size information to the standard output. The output information is categorized into three types. If the total size information needs to be output to the linkage list file, use the **show=total_size** option.

RAMDATA SECTION: Total size of the RAM sections (bytes)
ROMDATA SECTION: Total size of the ROM sections other than program sections (bytes)
PROGRAM SECTION: Total size of the program sections (bytes)

A section subject to the ROM support function (**rom** option) uses areas in both the transfer source (ROM) and destination (RAM). Therefore, the size of such sections is included in both the RAMDATA SECTION and ROMDATA SECTION values.

The following table lists the compiler's default sections and their section types.

Default section name	Section	Counted as
P	Program section	PROGRAM SECTION
C	Constant section	ROMDATA SECTION
D	Initialized data section	ROMDATA SECTION
R (when HEW is used)	Initialized data section (destination) (when the ROM support function is used)	RAMDATA SECTION
B	Uninitialized data section	RAMDATA SECTION
S	Stack section	RAMDATA SECTION

The following are output examples:

— The **total_size** option (standard output)

```
RAMDATA SECTION:  00000808 Byte(s)
ROMDATA SECTION:  0000041c Byte(s)
PROGRAM SECTION:  00000358 Byte(s)
```

— The **show=total_size** option (linkage list file)

```
*** Total Section Size ***

RAMDATA SECTION:  00000808 Byte(s)
ROMDATA SECTION:  0000041c Byte(s)
PROGRAM SECTION:  00000358 Byte(s)
```

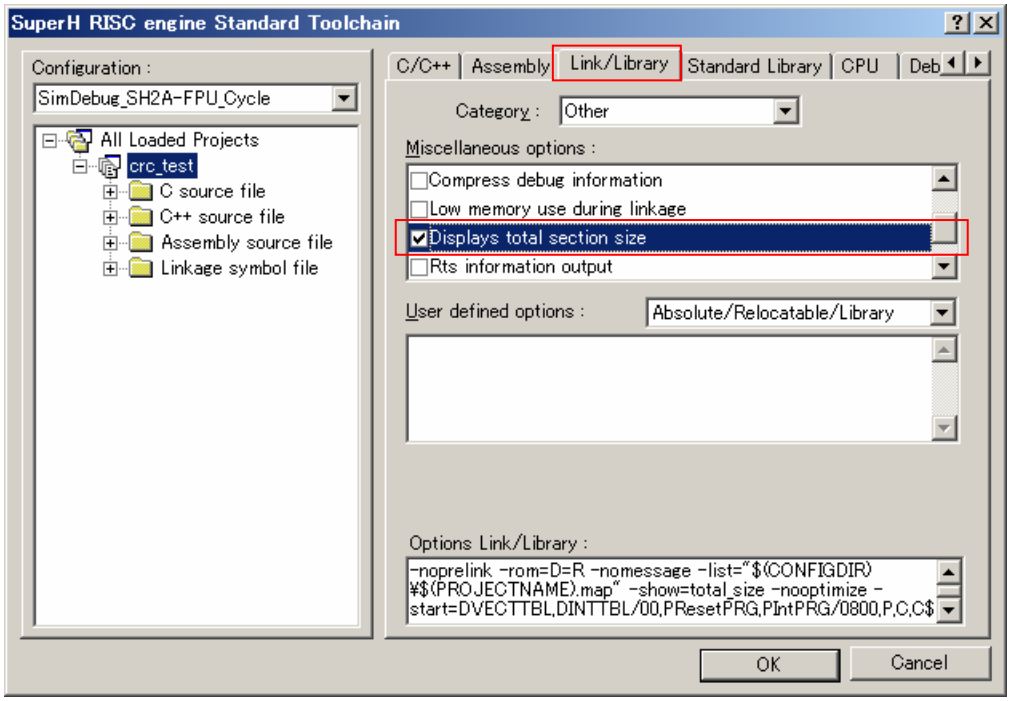
When these options are used in HEW, specify them as follows:

— The **total_size** option (standard output)

In the **SuperH RISC engine Standard Toolchain** dialog box, on the **Link/Library** page, specify the following settings:

Category: Select **Other**.

Displays total section size: Select this check box.

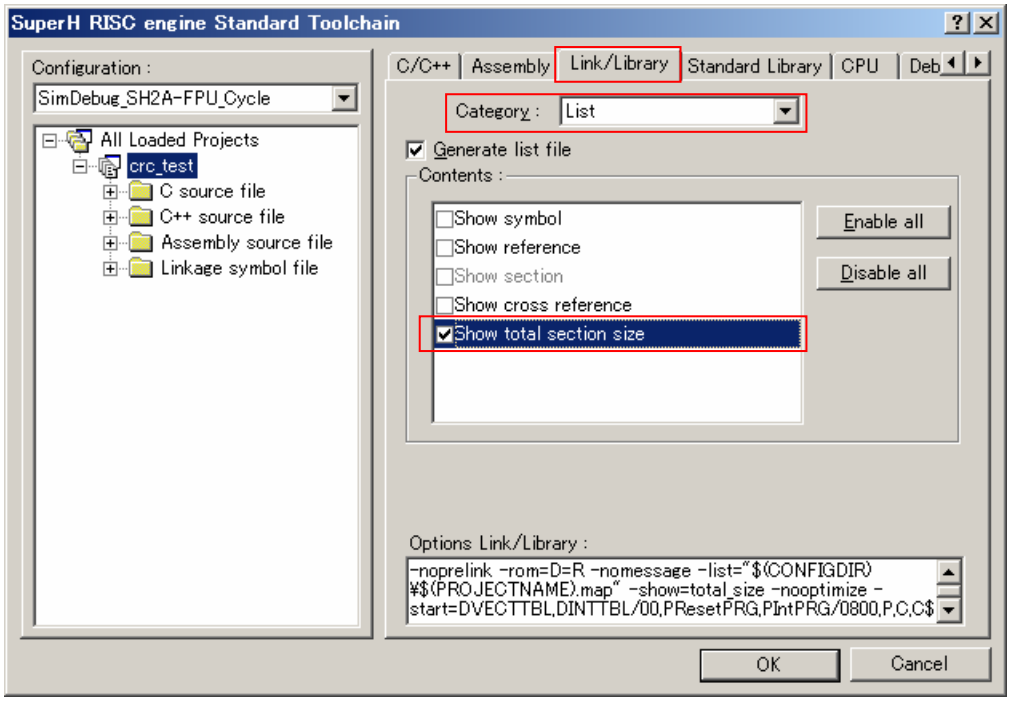


— The **show=total_size** option (linkage list file)

In the **SuperH RISC engine Standard Toolchain** dialog box, on the **Link/Library** page, specify the following settings:

Category: Select **List**.

Show total section size: Select this check box.



(b) Calculating and outputting the CRC value (crc)

The CRC (Cyclic Redundancy Check) value for the area specified in the **crc** option can now be calculated and output to the specified address. From the output CRC value, whether the data in the embedded system and the data during generation match can be checked. The following shows the format of the **crc** option:

-CRc = <suboption>

<suboption>: <address where the result is output>=<target range>[/<polynomial expression>]

<address where the result is output>: <address>

<target range>: <start address>-<end address>[,...]

<polynomial expression> : { CCITT | 16 }

A CRC is performed for the specified range of addresses from low to high, and the result is output to the specified address. For a polynomial expression, either CRC-CCITT or CRC-16 can be selected (the default is CRC-CCITT).

Polynomial expression:

CRC-CCITT

X^16+X^12+X^5+1

Bit representation: 10001000000100001

CRC-16

X^16+X^15+X^2+1

Bit representation: 11000000000000101

When this option is used with HEW, specify the following settings:

In the **SuperH RISC engine Standard Toolchain** dialog box, on the **Link/Library** page, specify the following settings:

Category: Select **Output**.

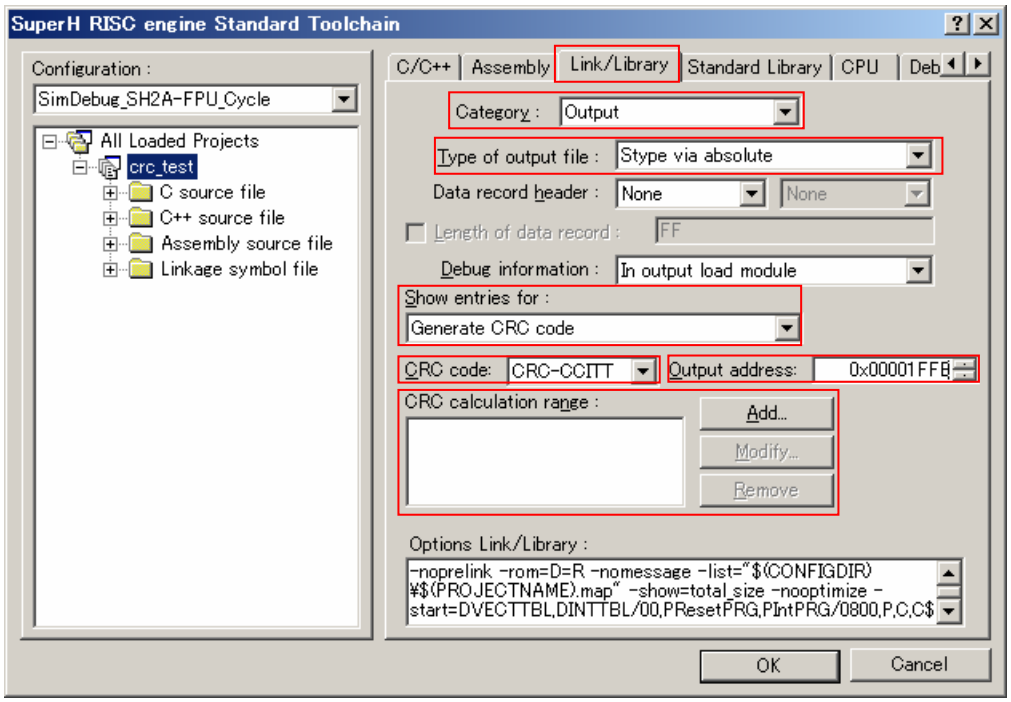
Type of output file: Select **Hew via absolute** or **Stype via absolute**.

Show entries for: Select **Generate CRC code**.

CRC code: Select **CRC-CCITT** or **CRC-16**.

Output address: Specify the address to which the CRC calculation result will be output.

CRC calculation range: Use the **Add**, **Modify**, and **Remove** buttons to specify the range of addresses for which a CRC will be performed.



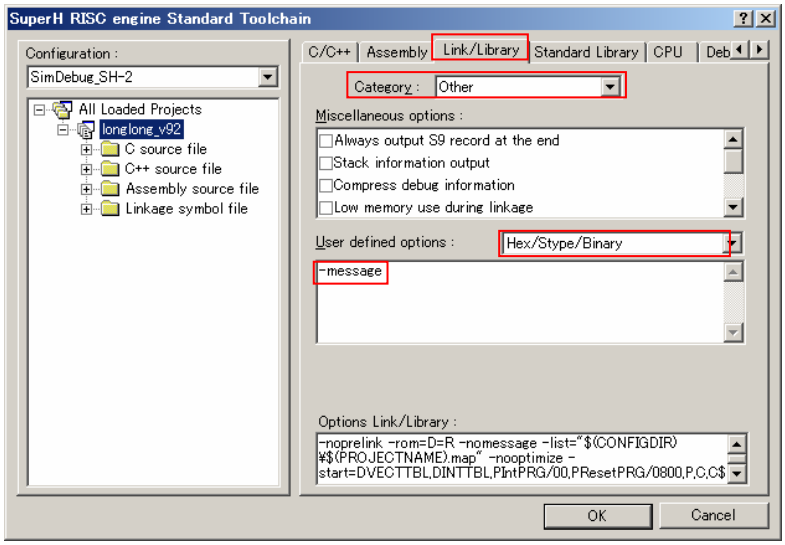
To output information about the output of the CRC result, specify the following settings on the **Link/Library** page in the **SuperH RISC engine Standard Toolchain** dialog box.

To output a message to the standard output:

Category: Select **Other**.

User defined options: Select **HEX/Stype/Binary**, and add the following entry:

-message



When the above settings are specified and the **crc** option is specified, the following message is output to the standard output:

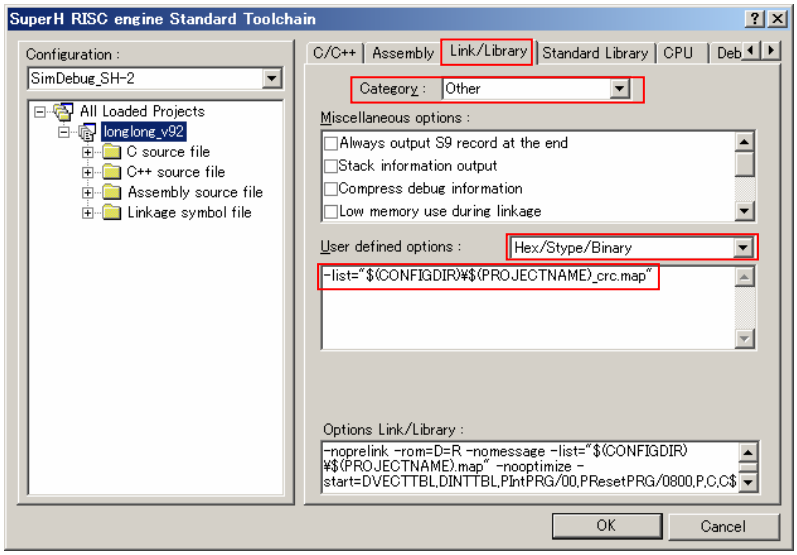
L0500 (I) Generated CRC code at "address"

To output the information to the linkage list:

Category: Select **Other**.

User defined options: Select **HEX/Stype/Binary**, and add the following entry:

-list="$(CONFIGDIR)\$(PROJECTNAME)_crc.map"



When the **crc** option is specified, the CRC result and the output destination address are output to the linkage list (project name: _crc.map) as follows:

CODE: CRC result

ADDRESS: Output destination address

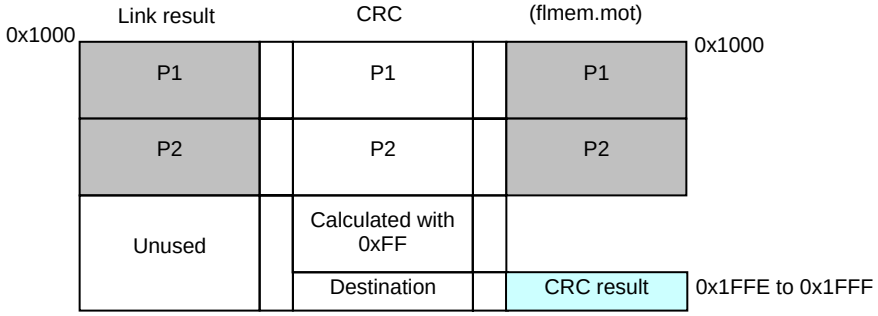
```
*** CRC code ***

CODE    : 5db6
ADDRESS : 00002ffe
```

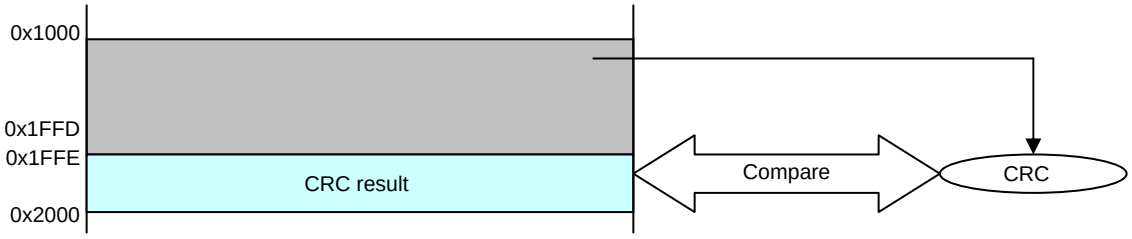
Example:

```
optlnk *.obj -form=stype -start=P1,P2/1000
  -crc=1FFE=1000-1FFD
  -output=flmem.mot
```

When options are set as shown above, the CRC result is output to "flmem.mot", which will be created.



If the ROM image generated with the above option settings is written to the computer's ROM, a CRC is performed for the range from 0x1000 to 0x1FFD, and the result is compared with the result that has been output to the ROM image. If these two results are a match, that range in the generated ROM image file and that range in the ROM image on the computer are identical.



The following shows sample programs that perform a CRC. By comparing the CRC result for the ROM image obtained by this sample program with the CRC result written to ROM, consistency of the data written to ROM can be confirmed.

— Polynomial expression CRC-CCITT

```
Sample Source:

typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
uint16_t CRC_CCITT(uint8_t *pData, uint32_t iSize)
{
    uint32_t ui32_i;
    uint8_t *pui8_Data;
    uint16_t ui16_CRC = 0xFFFFu;
    pui8_Data = (uint8_t *)pData;
    for(ui32_i = 0; ui32_i < iSize; ui32_i++)
    {
        ui16_CRC = (uint16_t)((ui16_CRC >> 8u) |
            ((uint16_t)((uint32_t)ui16_CRC << 8u)));
        ui16_CRC ^= pui8_Data[ui32_i];
        ui16_CRC ^= (uint16_t)((ui16_CRC & 0xFFu) >> 4u);
        ui16_CRC ^= (uint16_t) ((ui16_CRC << 8u) << 4u);
        ui16_CRC ^= (uint16_t)(((ui16_CRC & 0xFFu) << 4u) << 1u);
    }
    ui16_CRC = (uint16_t)( 0x0000FFFFul &
        ((uint32_t)~(uint32_t)ui16_CRC) );
    return ui16_CRC;
}
```

— Polynomial expression CRC-16

```
Sample Source:

#define POLYNOMIAL 0xa001
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;

uint16_t CRC16(uint8_t *pData, uint32_t iSize)
{
    uint16_t crcdData = (uint16_t)0;
    uint32_t data = 0;
    uint32_t i,cycLoop;
    for(i=0;i<iSize;i++){
        data = (uint32_t)pData[i];
        crcdData = crcdData ^ data;
        for (cycLoop = 0; cycLoop < 8; cycLoop++) {
            if (crcdData & 1) {
                crcdData = (crcdData >> 1) ^ POLYNOMIAL;
            } else {
                crcdData = crcdData >> 1;
            }
        }
    }
    return crcdData;
}
```

Supplementary note:

Note the following points when the **crc** option is used:

— A CRC is not performed in the order in which target ranges are specified. Instead, it is performed from the lowest address to the highest address.

— If multiple absolute files are input, the following warning message is output and the **crc** option is disabled:

L1000 (W) Option "crc" ignored

— The target ranges must not include the CRC result output address. If the target ranges include the CRC result output address, the following error message is output:

L2022 (E) Address ranges overlap in option "option" : "address range"

— The **crc** option takes effect when **form=hexadecimal** or **form=stype** is specified as the output format. If an entry other than these is specified, the following error message is output:

L2004 (E) Option "crc" cannot be combined with option "form=<output format>”

— If the output range is specified with the **output** option, the CRC result output address must be included in the output range. If the CRC result output address is not included in the output range, the following error message is output:

L1181 (W) Fail to write "CRC Code"

— The CRC calculation assumes that the unused area in a specified target range is 0xFF unless a value is specified by the **space** option. If **random** or a value of 2 bytes or more is specified, the following error message is output:

L2004 (E) Option "crc" cannot be combined with option "space=<value> "

Restrictions:

If an overlay area is used, do not specify the **crc** option.

B. Notes on Version Upgrade

This section describes notes when the version is upgraded from the earlier version (SuperH RISC engine C/C++ Compiler Package Ver. 6.x or lower).

B.1 Guaranteed Program Operation

When the version is upgraded and program is developed, operation of the program may change. When the program is created, note the followings and sufficiently test your program.

(1) Programs Depending on Execution Time or Timing

C/C++ language specifications do not specify the program execution time. Therefore, a version difference in the compiler may cause operation changes due to timing lag with the program execution time and peripherals such as the I/O, or processing time differences in asynchronous processing, such as in interrupts.

(2) Programs Including an Expression with Two or More Side Effects

Operations may change depending on the version when two or more side effects are included in one expression.

Example

```
a[i++]=b[i++];          /* i increment order is undefined.                    */
f(i++,i++) ;          /* Parameter value changes according to increment order.  */
                      /* This results in f(3, 4) or f(4, 3) when the value of i is 3.  */
```

(3) Programs with Overflow Results or an Illegal Operation

The value of the result is not guaranteed when an overflow occurs or an illegal operation is performed. Operations may change depending on the version.

Example

```
int a, b;
x=(a*b)/10;  /* This may cause an overflow depending on the value range of a and b. */
```

(4) No Initialization of Variables or Type Inequality

When a variable is not initialized or the parameter or return value types do not match between the calling and called functions, an incorrect value is accessed. Operations may change depending on the version.

File 1:

```
int f(double
d)
{
      :
}
```

File 2:

```
int g(void)
{
      f(1);
}
```

The parameter of the caller function is the int type, but the parameter of the callee function is the double type. Therefore, a value cannot be correctly referenced.

The information provided here does not include all cases that may occur. Please use this compiler prudently, and sufficiently test your programs keeping the differences between the versions in mind.

B.2 Compatibility with Earlier Version

The following notes cover situations in which the compiler (Ver. 5.x or lower) is used to generate a file that is to be linked with files generated by the earlier version or with object files or library files that have been output by the assembler (Ver. 4.x or lower) or linkage editor (Ver. 6.x or lower). The notes also covers remarks on using the existing debugger supplied with the earlier version of the compiler.

(1) Object Format

The standard object file format has been changed from SYSROF to ELF. The standard format for debugging information has also been changed to DWARF2.

When object files (SYSROF) output by the earlier version of the compiler (Ver. 5.x or lower) or assembler (Ver. 4.x or lower) are to be input to the optimizing linkage editor, use a file converter to convert it to the ELF format. However, relocatable files output by the linkage editor (extension: rel) and library files that include one or more relocatable files cannot be converted.

(2) Point of Origin for Include Files

When an include file specified with a relative directory format was searched for, in the earlier version, the search would start from the compiler's directory. In the new version, the search starts from the directory that contains the source file.

(3) C++ Program

Since the encoding rule and execution method were changed, C++ object files created by the earlier version of the compiler cannot be linked. Be sure to recompile such files.

The name of the library function for initial/post processing of the global class object, which is used to set the execution environment, has also been changed. Refer to section 9.2.2, Execution Environment Settings, and modify the name, in the SuperH RISC engine C/C++ Conpiler, Assembler, Optimizing Linkage Editor User's Manual.

(4) Abolition of Common Section (Assembly Program)

With the change of the object format, support for a common section has been abolished.

(5) Specification of Entry via .END (Assembly Program)

Only an externally defined symbol can be specified with .END.

(6) Inter-module Optimization

Object files output by the earlier version of the compiler (Ver. 5.x or earler) or the assembler (Ver. 4.x or earler) are not targeted for inter-module optimization. Be sure to recompile and reassemble such files so that they are targeted for inter-module optimization.

Website and Support <website and support,ws>

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

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

Revision Record <revision history,rh>

		Description	
Rev.	**Date**	**Page**	**Summary**
1.00	June.1.07	—	First edition issued
2.00	April.1.08	52	Addition

Notes regarding these materials

1. This document is provided for reference purposes only so that Renesas customers may select the appropriate Renesas products for their use. Renesas neither makes warranties or representations with respect to the accuracy or completeness of the information contained in this document nor grants any license to any intellectual property rights or any other rights of Renesas or any third party with respect to the information in this document.
2. Renesas shall have no liability for damages or infringement of any intellectual property or other rights arising out of the use of any information in this document, including, but not limited to, product data, diagrams, charts, programs, algorithms, and application circuit examples.
3. You should not use the products or the technology described in this document for the purpose of military applications such as the development of weapons of mass destruction or for the purpose of any other military use. When exporting the products or technology described herein, you should follow the applicable export control laws and regulations, and procedures required by such laws and regulations.
4. All information included in this document such as product data, diagrams, charts, programs, algorithms, and application circuit examples, is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas products listed in this document, please confirm the latest product information with a Renesas sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas such as that disclosed through our website. (http://www.renesas.com)
5. Renesas has used reasonable care in compiling the information included in this document, but Renesas assumes no liability whatsoever for any damages incurred as a result of errors or omissions in the information included in this document.
6. When using or otherwise relying on the information in this document, you should evaluate the information in light of the total system before deciding about the applicability of such information to the intended application. Renesas makes no representations, warranties or guaranties regarding the suitability of its products for any particular application and specifically disclaims any liability arising out of the application and use of the information in this document or Renesas products.
7. With the exception of products specified by Renesas as suitable for automobile applications, Renesas products are not designed, manufactured or tested for applications or otherwise in systems the failure or malfunction of which may cause a direct threat to human life or create a risk of human injury or which require especially high quality and reliability such as safety systems, or equipment or systems for transportation and traffic, healthcare, combustion control, aerospace and aeronautics, nuclear power, or undersea communication transmission. If you are considering the use of our products for such purposes, please contact a Renesas sales office beforehand. Renesas shall have no liability for damages arising out of the uses set forth above.
8. Notwithstanding the preceding paragraph, you should not use Renesas products for the purposes listed below:
   (1) artificial life support devices or systems
   (2) surgical implantations
   (3) healthcare intervention (e.g., excision, administration of medication, etc.)
   (4) any other purposes that pose a direct threat to human life

   Renesas shall have no liability for damages arising out of the uses set forth in the above and purchasers who elect to use Renesas products in any of the foregoing applications shall indemnify and hold harmless Renesas Technology Corp., its affiliated companies and their officers, directors, and employees against any and all damages arising out of such applications.
9. You should use the products described herein within the range specified by Renesas, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas shall have no liability for malfunctions or damages arising out of the use of Renesas products beyond such specified ranges.
10. Although Renesas endeavors to improve the quality and reliability of its products, IC products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Please be sure to implement safety measures to guard against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other applicable measures. Among others, since the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
11. In case Renesas products listed in this document are detached from the products to which the Renesas products are attached or affixed, the risk of accident such as swallowing by infants and small children is very high. You should implement safety measures so that Renesas products may not be easily detached from your products. Renesas shall have no liability for damages arising out of such detachment.
12. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written approval from Renesas.
13. Please contact a Renesas sales office if you have any questions regarding the information contained in this document, Renesas semiconductor products, or if you have any other inquiries.