To our customers,

Old Company Name in Catalogs and Other Documents

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

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

April 1st, 2010
Renesas Electronics Corporation

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

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

RENESAS

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: "Standard", "High Quality", and "Specific". The recommended applications for each Renesas Electronics product depends on the product's quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as "Specific" without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as "Specific" or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is "Standard" unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

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

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

   "Specific": Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) "Renesas Electronics" as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

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

Application Note

SuperH RISC engine C/C++ Compiler Package

Application Note

Renesas Microcomputer Development Environment System

Renesas Electronics
www.renesas-electoronics.com

Rev.4.00 2007.02

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.

Preface

The Renesas Tecnology SuperH RISC engine family of next-generation single-chip microcomputers offers high-performance processing while incorporating a variety of peripheral devices, and are designed for embedded applications, operating under low power consumption.

These application notes explain methods for the efficient creation of application programs which capitalize on the functions and performance of the Renesas Tecnology SuperH RISC engine family using the SuperH RISC engine C/C++ Compiler Package V. 9.00.

For detailed specifications of the C/C++ compiler, please refer to the SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual.

Organization of These Application Notes

These application notes consist of the following ten sections and an appendix.

Section 1 provides an overview and describes installation methods and the programming development procedure.

Section 2 illustrates the debugging process using various sample and explains program creation using the C language.

Section 3 gives warnings to be heeded when combining C language programs and assembly language programs, and when using cross-software with object files created using the C/C++ compiler and explains extended functions of the SuperH RISC engine C/C++ compiler, as well as procedures specific to software for embedded equipment.

Section 4 explains HEW options.

Section 5 and 6 explains methods for creating C language programs designed to capitalize on the performance of the Renesas Tecnology SuperH RISC engine family of microcomputers.

Section 7 illustrates the utilizing method using HEW.

Section 8 illustrates efficient C++ programming technique.

Section 9 explains useful options, as well as functions that perform lateral optimization across modules during linking.

Section 10 presents answers to questions frequently asked by users.

The appendix describes changes in each version of the SuperH RISC engine C/C++ compiler.

Related Manuals

The following are related manuals.

- Renesas Tecnology SuperH RISC engine Family, Microcomputer Hardware Manuals
- SuperH RISC engine High-performance Embedded Workshop 3 User’s Manual
- SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User’s Manual
- SuperH RISC engine High-performance Embedded Workshop 3 Tutorial

Cross-Software Versions

In order to use the SuperH RISC engine C/C++ compiler V. 9.00 the following cross-software versions should be used.

Cross Software Name	Version
SH-series cross assembler	7.00
H-series optimizing linkage editor	9.00
SH-series library generator	3.00

Symbols and Conventions used in this Application Note

[ ]: Indicates that the enclosed item can be omitted.

(RET): Indicates the Return (Enter) key is to be pressed.

Δ: Indicates one or more spaces or tabs.

abc: Boldfaced items are to be input by the user.

<>: Items enclosed in these brackets should be specified.

… : Indicates that the immediately preceding item is specified one or more times.

H': Integer constants preceded by H' are in hexadecimal.

0x: Integer constants preceded by 0x are in hexadecimal.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company limited.

MS-DOS® is a registered trademark of Microsoft Corporation in the United States and other countries.

Microsoft® WindowsNT® operating system, Microsoft®,Windows®98 and Windows 2000 operating system, Microsoft® WindowsMe® operating system, Microsoft® WindowsXp® operating system are registered trademarks of Microsoft Corporation in the United States and other countries.

IBM PC is a registered trademark of International Business Machines Corporation.

It is recommended that these application notes be read in the following manner.

No.	Circumstances	Use of these Application Notes
1	Using the SuperH RISC engine C/C++ compiler for the first time (1) You want to learn how to use the compiler to create load modules, and how to use cross-software. (2) You want to create programs which run on SH-1, SH-2, SH-2E, SH-2A, SH2A-FPU, SH2-DSP, SH-3, SH3-DSP, SH-4, SH-4A and SH4AL-DSP.	(1) The procedure for starting the compiler is described in section 1.4, Method of Execution. In section 1.5, Procedure for Program Development, operations using cross-software necessary to complete a load module are explained. (2) There are programs in sections 2.2 and 2.3, Introduction of Sample Program. These are programs provided in order to explain the bare minimum of compiler functions necessary for embedded equipment. Please refer to them in creating simple programs, and using the simulator, debugger and other tools to confirm their operation. Other compiler functions are described in section 3, Compiler. If you encounter problems in creating load modules, please refer to section 3.15, Issues Related to Cross-Software.
2	A program for embedded equipment is to be created. (1) There is a program used with other microcomputers which will be ported.	(1) Read sections 2.2 to 2.3, Introduction of Sample Program, and section 3, Compiler, to discover functions you can use, and consider whether the assembly language code cannot be rewritten in the C language. For information on combining assembly language programs with C programs, please refer to section 3.15.1, Issues Related to Assembly Language Programs.
	(2) A new program will be created.	(2) First read sections 2.2 to 2.3, Introduction of Sample Program, for a summary of program creation. Next proceed to section 3, Compiler, and learn about the extended functions of the SuperH RISC engine C/C++ compiler. In creating a program, refer to section 5, Efficient Programming Techniques, to ensure that your programs are successful from the very start.
3	Execution speed is to be improved, or program size reduced.	Refer to section 5, Efficient Programming Techniques to improve performance.
4	The program does not run as expected.	Examine the warning information following each relevant item, as well as the items in section 11. Q&A, to determine whether there is any relevant information.

Contents

Section 1 Overview

1.1 Summary

The SuperH RISC engine C/C++ compiler enables effective creation in the C language of programs which take advantage of the functions and performance of the Renesas Technology SuperH RISC engine family of single-chip microcomputers for embedded applications.

This document explains procedures for creating application programs using this C/C++ compiler.

1.2 Features

The features of the SuperH RISC engine C/C++ compiler are as follows.

(1) Full complement of functions

The following functions can be used to create efficient application programs for the Renesas Technology SuperH RISC engine family.

- C language representation of interrupt functions and special instructions for the Renesas Technology SuperH RISC engine family
- Generation of position-independent code (excluding SH-1)
- Fast floating-point operations
- Selection of compiler settings to give priority to speed of execution or efficiency of memory use

(2) Powerful optimization features

The following types of optimization are performed in order to utilize the performance of the Renesas Technology SuperH RISC engine family with its RISC (Reduced Instruction Set Computer) type instruction set.

- Automatic/optimized allocation of local variables to registers
- Alleviation of processing intensity
- Pipeline optimization
- Constant convolution
- String sharing
- Deletion of common expression/loop invariant
- Deletion of unnecessary text
- Optimization of tail recursion
- Optimization between modules

Hence programming is possible without the need to explicitly take into account the architecture of the Renesas Technology SuperH RISC engine family.

1.3 Method of Installation

1.3.1 PC Version

The operating environment for the SuperH RISC engine C/C++ compiler for Windows 98/Me/2000/XP/NT, and the procedure for installing on Windows 98/Me/2000/XP/NT, are explained below.

(1) Operating environment

- Host computer: IBM PC-compatible
  (CPU capable of running Windows 98/Me/2000/XP/NT)
- OS: Windows 98/Me/2000/XP/NT (Japanese or English)
- Memory size: Minimum 128 MB, 256 MB or more recommended
- Hard disk space: 120 MB or more free disk space required (for full installation)
- Display: SVGA or better
- I/O device: CD-ROM drive
- Others: Mouse or other pointing devices

(2) Method of installation

To install the integrated development environment on the PC, click the Setup button in the [Add/Remove Programs applet] in [Control Panel], and then follow the onscreen instructions.

(3) Using the compiler from the DOS prompt

When using the compiler from the DOS command line under Windows, certain environment variables must be set.

Explanation of Environment Variables

(a) Environment variable SHC_LIB

Indicates where the main files of the SuperH RISC engine C/C++ compiler are saved.

(b) Environment variable SHC_TMP

Specifies the path for creation of temporary files used by the C/C++ compiler. This setting cannot be omitted.

(c) Environment variable SHC_INC

This environment variable is set when reading the standard header files for the C/C++ compiler from a specified path. Several paths can be specified by separating them with commas (','). If this environment variable is not set, the standard header file is read from SHC_LIB.

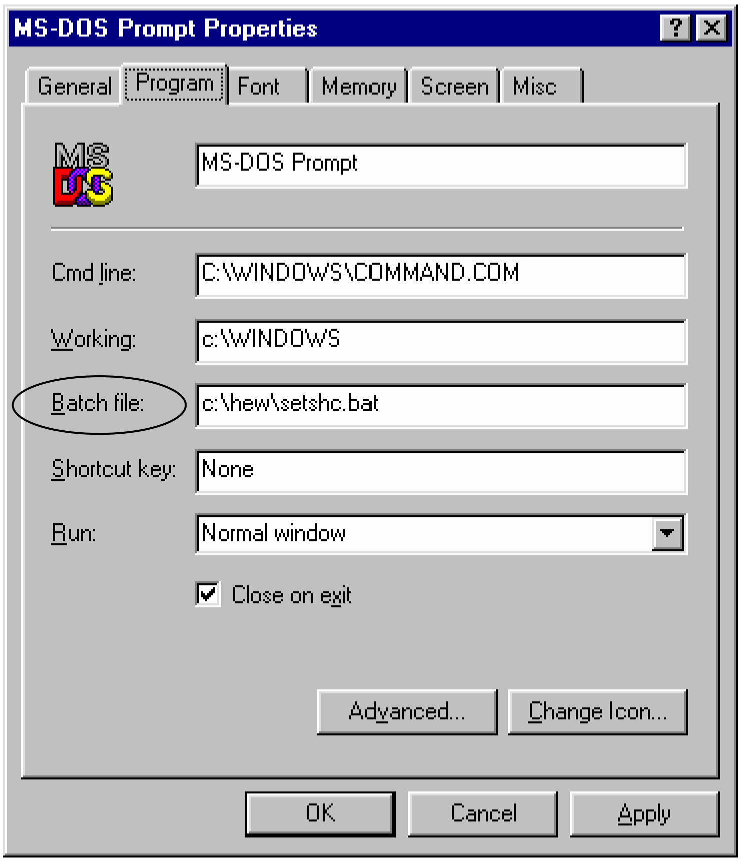
First, create a batch file with the contents shown below, which are necessary when starting with the DOS prompt. If such a batch file already exists, the items shown below should be added. The example shown below is for installation of the integrated environment to hard disk drive C.

To set the path, first use the SET command at the MS-DOS prompt to determine the current path setting, and then add to it as necessary.

The following are examples of notation for batch files.

```
PATH C:\Hew3\Tools\Renesas\Sh\9_0_0\bin; %PATH%

SET SHC_LIB=C:\Hew3\Tools\Renesas\Sh\9_0_0\bin

SET SHC_TMP=C:\tmp

SET SHC_INC=C:\Hew3\Tools\Renesas\Sh\9_0_0\include
```

Next, enter the path of the batch file below as the Batch file on the [Program] tag of the DOS prompt properties dialog.



MS-DOS Prompt

Cmd line: C:\WINDOWS\COMMAND.COM

Working: c:\WINDOWS

Batch file: c:\hew\setshc.bat

Shortcut key: None

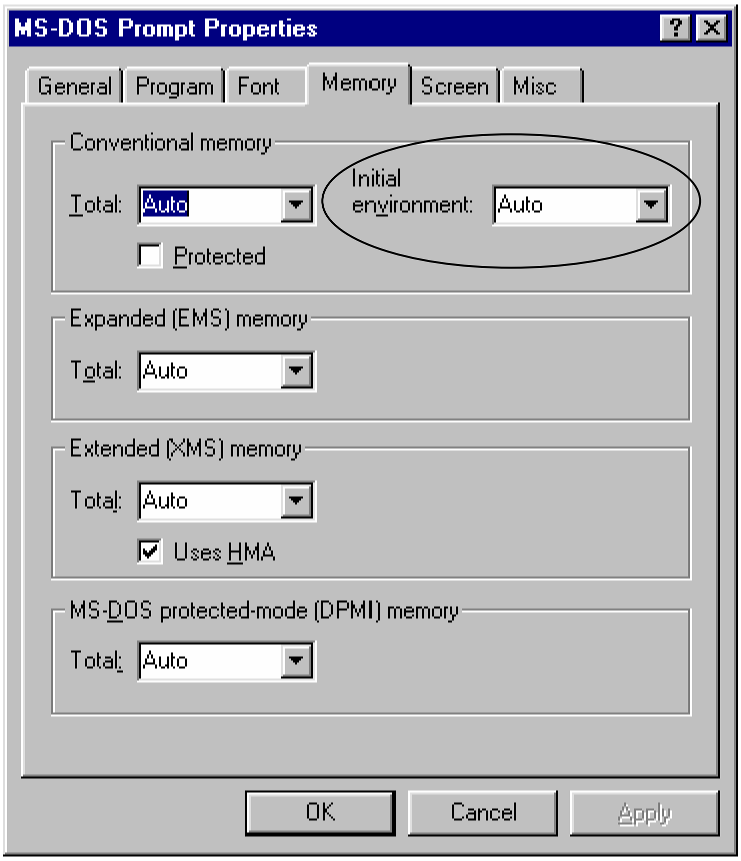
Run: Normal window

☑ Close on exit

Figure 1.1 MS-DOS Prompt Properties (1)

On completing the above settings, restart the MS-DOS prompt session.

Note: If, on running the batch file, the message "Insufficient space for environment variables" appears, please make the following change.



MS-DOS Prompt Properties

Conventional memory

Total: Auto

Initial environment: Auto

☐ Protected

Expanded (EMS) memory

Total: Auto

Extended (XMS) memory

Total: Auto

☑ Uses HMA

MS-DOS protected-mode (DPMI) memory

Total: Auto

Figure 1.2 MS-DOS Prompt Properties (2)

On the [Memory] tag of the DOS prompt properties dialog, change the "Initial environment" from Automatic to 1024.

After changing this setting, the MS-DOS prompt session must be restarted.

1.3.2 UNIX Version

The procedure for installing the C/C++ compiler on a UNIX system is described below.

Note: Do not use Japanese characters or spaces in the name for the installation directory.

(1) Installation media

The compiler is distributed on a single CD-ROM.

(2) Method of installation

Please use the following procedure to install the compiler. Wherever (RET) appears in the instructions, the Enter (Return) key is to be pressed.

(a) Compiler/simulator installation

The procedure for compiler/simulator installation is as follows.

(i) Creating a path for the compiler/simulator

Create a path for storage of the compiler files, using any arbitrary name.

```
% mkdirΔ<compiler and simulator pathname> (RET)
```

(ii) Mounting the CD-ROM

Mount the CD-ROM as indicated below. If mounting is performed automatically, the following command is not required.

[For Solaris]

```
% mountΔrΔFΔhsfsΔdev/dsk/c0t6d0s2Δ/cdrom (RET)
```

[For HP-UX]

```
% mountΔ/dev/dsk/c201d2s0Δ/cdrom (RET)
```

(iii) Copying the compiler/simulator

Move to the newly created path, and then decompress the files for the SuperH RISC engine C/C++ compiler/simulator from the CD-ROM to the path created in (i) above.

[For Solaris]

```
% cdΔ<compiler and simulator pathname> (RET)

% tarΔ-xvfΔ/cdrom/sh c sim pack sparc/Program.tar (RET)
```

[For HP-UX]

```
% cdΔ<compiler and simulator pathname> (RET)

% tarΔ-xvfΔ/cdrom/Program.tar (RET)
```

(iv) Changing environment settings

Set environment variables and pathnames as follows. (Double asterisks ** indicate an appropriate value should be specified.) For details on environment variables, refer to the SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual.

```
% setenvΔSHC_LIBΔ<compiler and simulator pathname> (RET)

% setenvΔSHC_INCΔ<compiler and simulator pathname> (RET)

% setenvΔSHC_TMPΔusr/tmp (RET)

% setenvΔSHCCPUΔSH** (RET)

% setenvΔHLNK_TMPΔ/usr/tmp (RET)

% setenvΔHLNK_LIBRARY1Δ<compiler and simulator pathname>/******.lib (RET)

% setenvΔHLNK_LIBRARY2Δ<compiler and simulator pathname>/******.lib (RET)
```

(v) Unmount the CD-ROM.

```
% umountΔ/cdrom (RET)
```

(b) Simulator installation

The procedure for installing the UNIX simulator in versions earlier than Ver 8 is as follows.

(i) Mounting the CD-ROM

Refer to the README.TXT file for the procedure to mount the CD-ROM.

When using the method described in the README.TXT file to copy cross-software, move to the directory to which it is copied, and then proceed to the explanation in (iii), Start the installer.

(ii) Load the installer from the tar file on the CD-ROM.

(The following example assumes that the CD-ROM drive device name is /cdrom.)

```
tarΔxvfΔcdrom/program.tarΔcas_install (RET)
```

(iii) Start the installer.

```
cas install (RET)
```

(iv) Display immediately after installer startup

The following message is displayed immediately after the installer is started.

Installation of the cycle-accurate simulator starts. Input parameters according to the messages.

(v) CPU selection

Select the simulator CPU to be used. (In this example, "10" selects the SH4 CPU.)

If the CPU SH4 or SH2DSP is selected, proceed to the explanation (vi) Co-verification tool selection.

If a CPU other than SH4 or SH2DSP is selected, proceed to the explanation (vii) Input name of directory for installation of definition files.

```
Target
CPU(1:SH1,2:SH2,3:SH3,4:SH3E,5:SHDSP,6:SH2E,7:SH4BSC,8:SH3DSP,9:SHDSPC,10:SH4,11:SH2DSP)
: 10
```

(vi) Co-verification tool selection

If the CPU SH4 or SH2DSP was selected in (v), select the name of the simulator co-verification tool to be used. (In this example, "1" selects “Seamless”.)

When not using a co-verification tool, select “No”.

If the CPU SH2DSP is selected in (v), Eaglei cannot be selected.

```
SH-4:Please select Co-Velification Tool(1:Seamless,2:Eaglei,3:No): 1
```

(vii) Input name of directory for installation of definition files

Input the directory for installation of definition files. The directory enclosed in parentheses ( ) is the default. The default is generated according to the following rules.

```
<current directory> + "/df_CSDSH"
```

If the default directory name is acceptable, simply enter (RET). A directory name can be entered either as an absolute path or as a relative path. (In the example, (RET) is input.)

```
Directory name for the definition files(/export/home1/cas/cassh3sim/df_CSDSH): (RET)
```

(viii) Enter the name of the host machine with the CD-ROM drive

Input the name of the host machine with the CD-ROM drive. The default is the name of the startup host. When installing from a CD-ROM drive of the startup host, press (RET).

When installing from a CD-ROM drive of another host on a network, input that host's name. However, in this case, it is assumed that login from a remote shell is possible (the /etc/hosts.equiv and $HOME/.rhosts files are set). For details on the environment settings for remote shells and related matters, refer to the manual for the startup machine.

When installing from the startup host’s CD-ROM drive, proceed to the explanation (x) Input tar file name.

When installing from a CD-ROM drive of another host on the network, proceed to the explanation in (ix), Input the login name of the host machine with the CD-ROM drive. (In the example below, the name of the host with the CD-ROM drive is sp3.)

```
Host name connected to a tape driver(sparc2): sp3 (RET)
```

(ix) Input the login name of the host machine with the CD-ROM drive

Input the login name of the host machine with the CD-ROM drive. This message is displayed when installing from a CD-ROM drive of another host. (In this example, the login name for the host machine with the CD-ROM drive is "remote.")

```
Login name of host connected to a tape driver:remote (RET)
```

(x) Input tar file name

Input the tar file name. The default is /dev/rmt/0m for the HP9000, and /dev/rmt/0 for SPARC. (The following example assumes that the CD-ROM drive device name is /cdrom.)

```
Tape driver name(/dev/rmt/0):  /cdrom/simulator.tar (RET)
```

(xi) Input (RET) before installation of definition files

Confirm that the CD-ROM with the definition files is mounted on the CD-ROM drive, and enter (RET).

```
Input return, after setting the tape including the definition files to the tape driver.
(RET)
```

(xii) Choose whether to install main files

Choose whether or not to install the main files for the interface software.

To install the main files enter "y," otherwise enter "n."

If "n" is entered, proceed to the explanation (xv) Choose whether to install setup files.

(In this example, "y" is entered.)

```
   Do you install the main files?(y/n):y (RET)
```

(xiii) Input of directory name for main files

Input the directory for installation of interface software main files.

The default is generated according to the following rule.

<current directory> + "/main"

A directory name can be entered either as an absolute path or as a relative path.

(In the following directory, (RET) is input.)

```
   Directory name for the main files(/export/home1/cas/cassh3sim/main): (RET)
```

(xiv) Input (RET) before main file installation

Confirm that the CD-ROM with the interface software main files is mounted on the CD-ROM drive, and enter (RET).

```
   Input return,after setting the tape including the main files to the tape driver. (RET)
```

(xv) Choose whether to install setup files

Choose whether or not to copy the setup sample files. To copy the setup files enter "y," otherwise enter "n."

(In this example, "y" is entered; then the installation file name is displayed.)

```
   Do you copy the setup files to current directory?(y/n): y (RET)
```

(xvi) Choose path and environment variable settings

Choose whether or not to add path and environment variable settings to the shell script.

If "y" is entered, the installer determines the login shell type from the "SHELL" environment variable, backs up an arbitrary shell script file (see table 1.1) beneath the directory specified by the "HOME" environment variable, and then sets the path and environment variable information.

Settings conform to the following specifications.

If the main files have not been installed (see (xii) Choose whether to install main files), path settings are not changed.

If a relative path was specified in (vii) Input name of directory for installation of definition files or in (xiii) Input of directory name for main files, the path and environment variable information is set using the input path information, and execution is not possible in directories other than the directory in which the installer was started.

If "n" is entered, the installer proceeds to (xviii), Installation completed message, and the installer is terminated.

Table 1.1 Filenames Used for Different Shells

No.	Shell Name	Script File Name	Backup File Name
1	Bourne shell (sh)	.profile	.profile.bak
2	C shell (csh)	.cshrc	.cshrc.bak
3	Korn shell (ksh)	.profile	.profile.bak

(In this example, "y" is entered; thereafter the shell script is displayed.)

```
Do you append the path list and the environment variables in shell script?(y/n): y (RET)

/export/home1/cas/.cshrc
```

(xvii) Choose whether to overwrite the backup file

When backing up the shell script, this message is displayed if there exists a file with the same name as the backup file. Choose whether or not to overwrite the file.

(In this example, the login shell is the C shell.)

```
Do you overwrite the backup file(.cshrc.bak)?(y/n): y (RET)
```

(xviii) Installation completed message

When all steps of the installation are completed, the following message is displayed and the installer terminates.

```
Installation of the cycle-accurate simulator completed.
```

(c) Installation of the Acrobat® Reader

The manual can be viewed from within Windows. The software used to view the manual (the Acrobat® Reader) should be installed on a computer running Windows 98/Me/2000/XP/NT.

Acrobat® Reader copyright ® 1987-2001 Adobe Systems Incorporated. All rights reserved.

Adobe and Acrobat are trademarks of Adobe Systems and are registered in specific jurisdictions.

The following procedure is used to execute installation. Before commencing the installation procedure, be sure to close all applications:

(i) Insert the CD-ROM provided into the CD-ROM drive. (Here it is assumed that the CD-ROM drive is drive D.)

(ii) From the Windows® Start menu, click on [Run …].

(iii) Specify in the [Run …] dialog box either Acrobreader51_jpn.exe (Japanese) in the [PDF_Read\Japanese] directory on the CD-ROM or Acrobreader51_eng.exe (English) in the [PDF_Read\English] directory (example:D:\PDF_Read\Japanese\ Acrobreader51_jpn.exe), and then click [OK].

(iv) Follow the onscreen installation instructions.

1.4 Method of Execution

1.4.1 Starting the Embedded Workshop

Upon completion of installation, the installer for the Embedded Workshop creates a folder named " Embedded Workshop 2" within the Programs folder in the Windows Start menu, and within this folder places shortcuts to the executable program for the Embedded Workshop and to other files.

The content of the start menu may vary depending on which tools are installed.

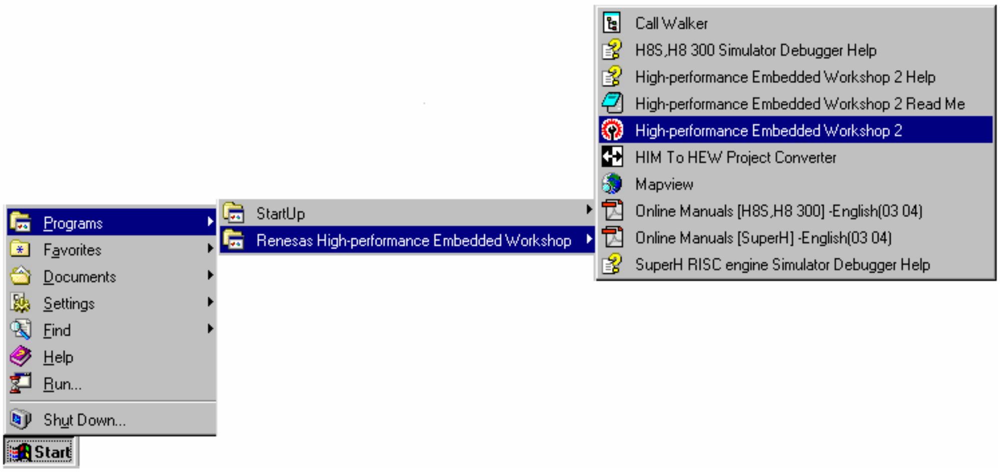
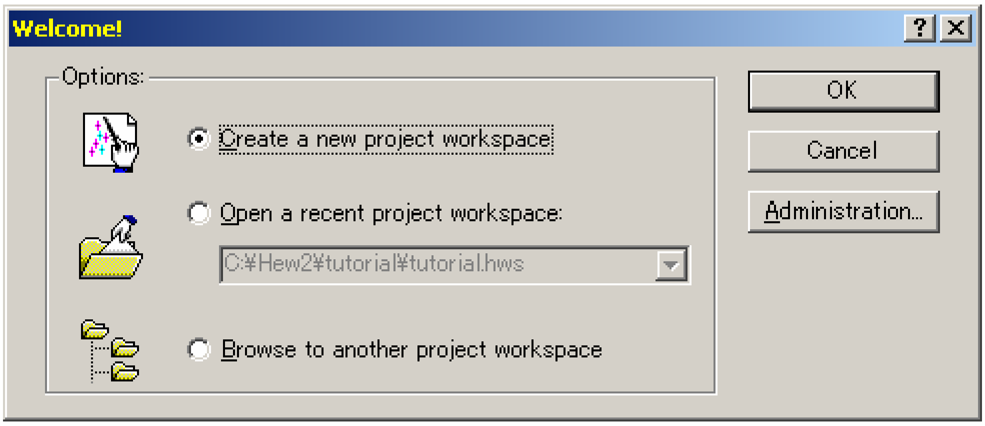

Programs
Favorites
Documents
Settings
Find
Help
Run...
Shut Down...
Start
StartUp
Renesas High-performance Embedded Workshop
Call Walker
H8S,H8 300 Simulator Debugger Help
High-performance Embedded Workshop 2 Help
High-performance Embedded Workshop 2 Read Me
High-performance Embedded Workshop 2
HIM To HEW Project Converter
Mapview
Online Manuals [H8S,H8 300] -English(03 04)
Online Manuals [SuperH] -English(03 04)
SuperH RISC engine Simulator Debugger Help

Figure 1.3 Startup of the Embedded Workshop from the Start Menu

Upon clicking on the Embedded Workshop item in the Start menu, a startup message is displayed, followed by a Welcome! dialog box (figure 1.4).


Welcome!

Options:

● Create a new project workspace

○ Open a recent project workspace:

C:¥Hew2¥tutorial¥tutorial.hws

○ Browse to another project workspace

Figure 1.4 Welcome! Dialog Box

When using the Embedded Workshop for the first time, or when beginning work on a new project, select [Create a New Project] and click [OK]. In order to resume work on a project that has already been created, select [Open an Existing Project] or [Browse to another project workspace] and click [OK]. No matter which of these is selected, clicking on [Exit] causes the Embedded Workshop to terminate. By clicking on [Administration...], system tools used with the Embedded Workshop can be registered and deleted.

1.4.2 Starting the Compiler

In this subsection, the method for executing the SuperH RISC engine C/C++ compiler is explained, along with examples. For details on compiler options, refer to the SuperH RISC Engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual. When using the PC version, refer to the Operating Manual.

Table 1.2 Table of compiling conditions

Command	Option	Extension of File(s) to be Compiled	Compiling Conditions
shcpp	Any	Any	C++-compiled
shc	-lang=c	Any	C-compiled
	-lang=cpp		C++-compiled
	No lang option specified	*.c	C-compiled
		*.cpp, *.cc, *.cp, *.CC	C++-compiled

The command shc C-compiles* or C++-compiles C programs and C++ programs, respectively, according to the lang option or the extension of the program file name. The command shcpp C++-compiles all programs, regardless of whether they are C programs or C++ programs. Compiling conditions are described in table 1.2.

Note: *C-compiling means a program is compiled based on C language syntax; C++-compiling means a program is compiled based on C++ language syntax.

Below the basic procedures for using the compiler are explained.

(1) Program compiling

To compile the "test.c" C source program:

```
shcΔtest.c (RET)
```

To compile the "test.cpp" C++ source program:

```
shcΔtest.cpp(RET)

shcppΔtest.cpp(RET)
```

(2) Displaying command input formats and compiler options

This command displays a list of command input formats and compiler options on the standard output screen.

```
shc (RET)

shcpp(RET)
```

(3) Specifying options

Options (debug, listfile, show, etc.) are prefixed with a hyphen (-), and multiple options are separated by spaces (Δ). In the PC version, a slash (/) can be used in place of the hyphen at the DOS prompt.

When specifying multiple suboptions, they should be separated by commas (,).

```
shcΔ-debugΔ-listfileΔ-show=noobject,expansionΔtest.c(RET)
```

In the PC version, parentheses can also be used to enclose suboptions.

```
shcΔ/debugΔ/listfileΔ/show=(noobject,expansion)Δtest.c(RET)
```

(4) Compiling multiple C/C++ programs

Multiple C/C++ programs can be compiled at once. The following are examples of commands for compiling C source programs.

Example 1: Specifying multiple programs for compiling

```
shcΔtest1.cΔtest2.c (RET)
```

Example 2: Specifying options (options are specified for all C source programs)

```
shcΔ-listfileΔtest1.cΔtest2.c (RET)
```

The listfile option is effective for both test1.c and test2.c.

Example 3: Specifying options (options are specified separately for each program)

```
shcΔtest1.cΔtest2.cΔ-listfile(RET)
```

Here the listfile option is effective only for test2.c. Specification of options for individual programs takes priority over specification of options for all source programs.

Notes:

(1) If, after installation, the compiler cannot be run, check the following.

- Confirm that the PATH environment variable includes the directory containing the C/C++ compiler.
- Confirm that the SHC_LIB environment variable is set to the directory containing the main C/C++ compiler files.

The SHC_LIB environment variable is used to set the directory containing the main C/C++ compiler files. Hence if the full set of C/C++ compiler files is not placed in the same directory, the compiler will not run.

(2) The compiler determines the syntax to be used at compile time according to whether the shc or the shcpp command is used; but even when the shc command is used, it will perform C++-compiling depending on file extensions and options.

1.5 Procedure for Program Development

Figure 1.5 shows the procedure used to develop a C/C++ language program. The shaded area shows the software provided as the SuperH RISC engine C/C++ compiler package.

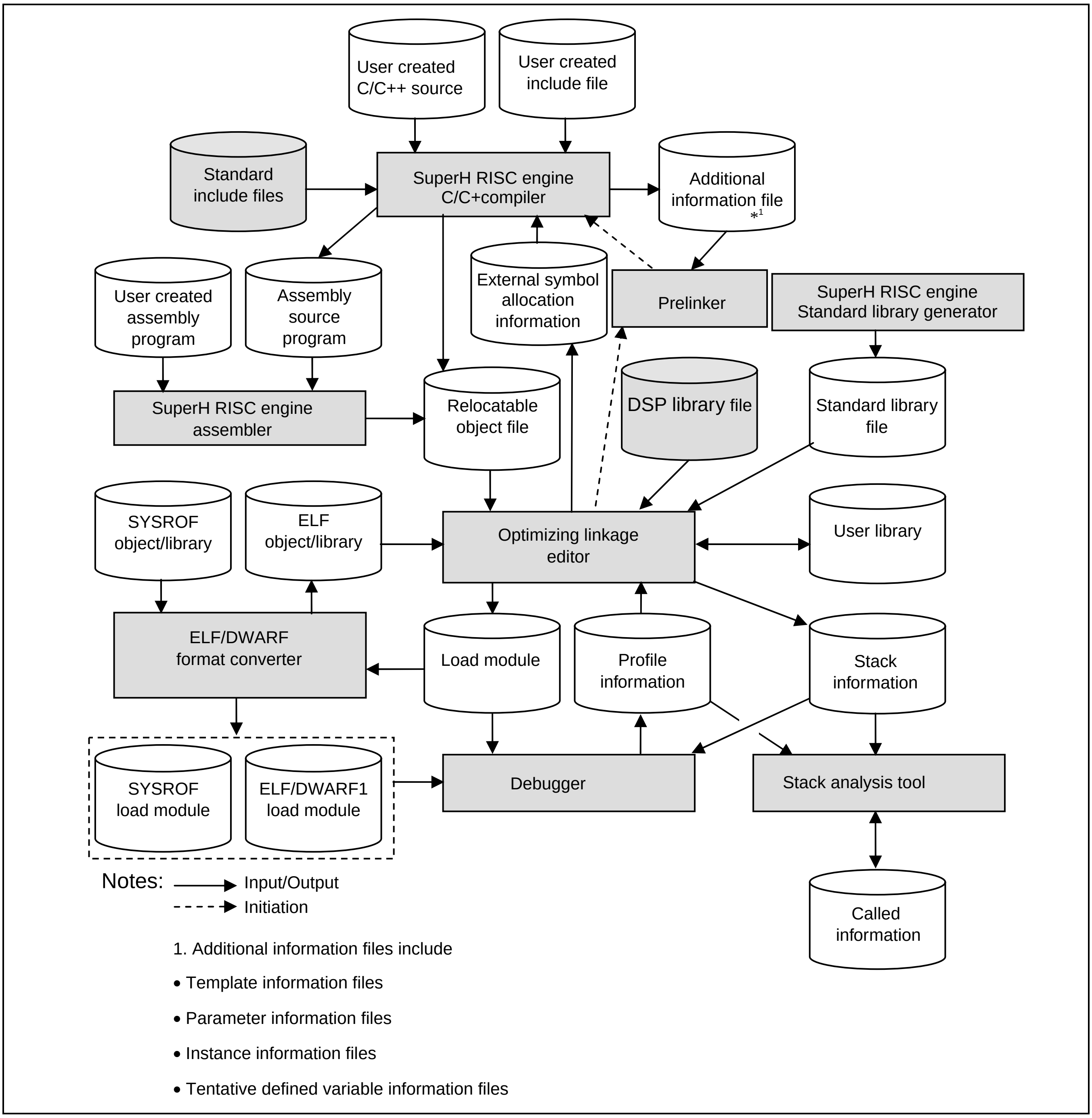

User created C/C++ source
User created include file
Standard include files
SuperH RISC engine C/C+compiler
Additional information file *1
User created assembly program
Assembly source program
External symbol allocation information
Prelinker
SuperH RISC engine Standard library generator
SuperH RISC engine assembler
Relocatable object file
DSP library file
Standard library file
SYSROF object/library
ELF object/library
Optimizing linkage editor
User library
ELF/DWARF format converter
Load module
Profile information
Stack information
SYSROF load module
ELF/DWARF1 load module
Debugger
Stack analysis tool
Called information
Notes:
Input/Output
Initiation
1. Additional information files include
• Template information files
• Parameter information files
• Instance information files
• Tentative defined variable information files

Figure 1.5 Procedure for Program Development

Below the procedure for program development is explained for the example of a source file on_motor.c. For details of use of cross-software, please refer to the user's manual for the cross-software package.

(1) Create a source file

Use the editor to create a source file.

(2) Generate a relocatable object file

Start the compiler, and compile the source file.

```
shcΔon_motor.c (RET)
```

A relocatable object file called on_motor.obj, which is optimized and without debugging information, is generated. In order to generate a list file, specify the listfile option.

(3) Generate a load module file

On including the library file sensor.lib and starting the linkage editor, an executable load module file with the name on_motor.abs is generated.

```
optlnkΔ-nooptΔ-subcommand = link.sub (RET)
```

The contents of lnk.sub are as follows.

```
Sdebug
input on_motor
library sensor.lib
Exit
```

Even if a relocatable object file contains debugging information, if the debug option is omitted when linking, no debugging information is output to the load module file.

(4) S-type format file output

In order to write to an EPROM using a ROM programmer, lnk.sub should be coded as follows.

```
Form=stype
Sdebug
input on_motor
library sensor.lib
Exit
```

An S-type format load module file with the name on_motor.mot is generated.

Section 2 Procedure for Creating and Debugging a Program

2.1 Creating a Project

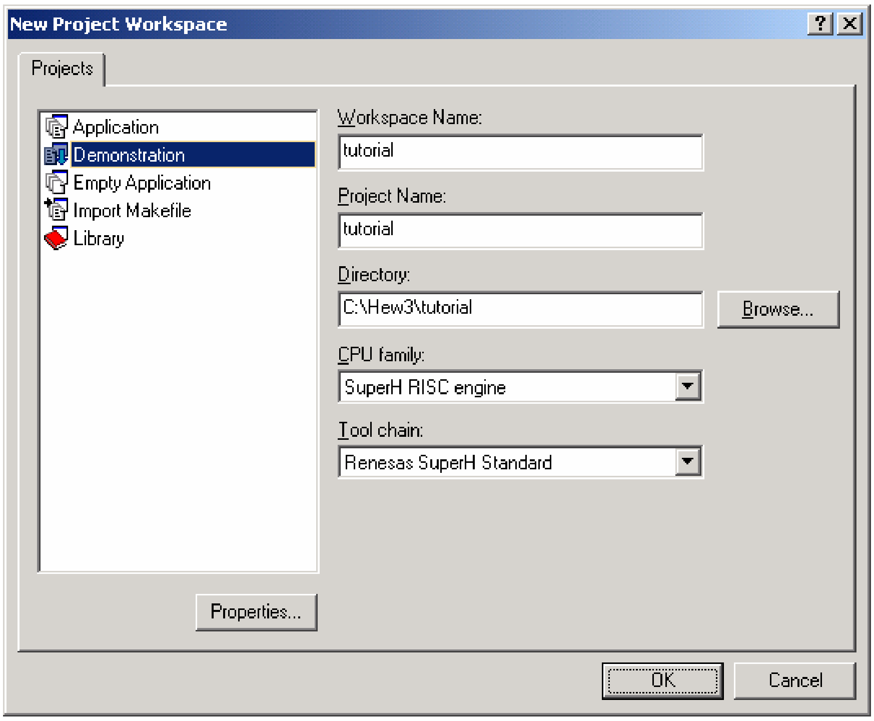
2.1.1 Creating the Project for a Simulator Debugger

(1) Specify the project

When you have selected the [Create a new project workspace] radio button and clicked [OK] on the [Welcome!] dialog box, the [New Project Workspace] dialog box (figure 2.1), which is used to create a new workspace and project, will be launched. You will specify a workspace name (when a new workspace is created, the project name is the same as the default), a CPU family, a project type, and so on, on this dialog box.

Enter “tutorial”, for example, in the [Workspace Name] field, then the [Project Name] field will show “tutorial” and the [Directory] field will show “c:\hew2\tutorial”. If you want to change the project name, enter a new project name manually in the [Project Name] field. If you want to change the directory used for the new workspace, click the [Browse…] button and specify a directory, or enter a directory path manually in the [Directory] field.

Here, specify [Demonstration] as a left-hand side Project type.



Workspace Name: tutorial

Project Name: tutorial

Directory: C:\Hew3\tutorial

CPU family: SuperH RISC engine

Tool chain: Renesas SuperH Standard

Figure 2.1 New Project Workspace Dialog Box

(2) Selecting the Target CPU

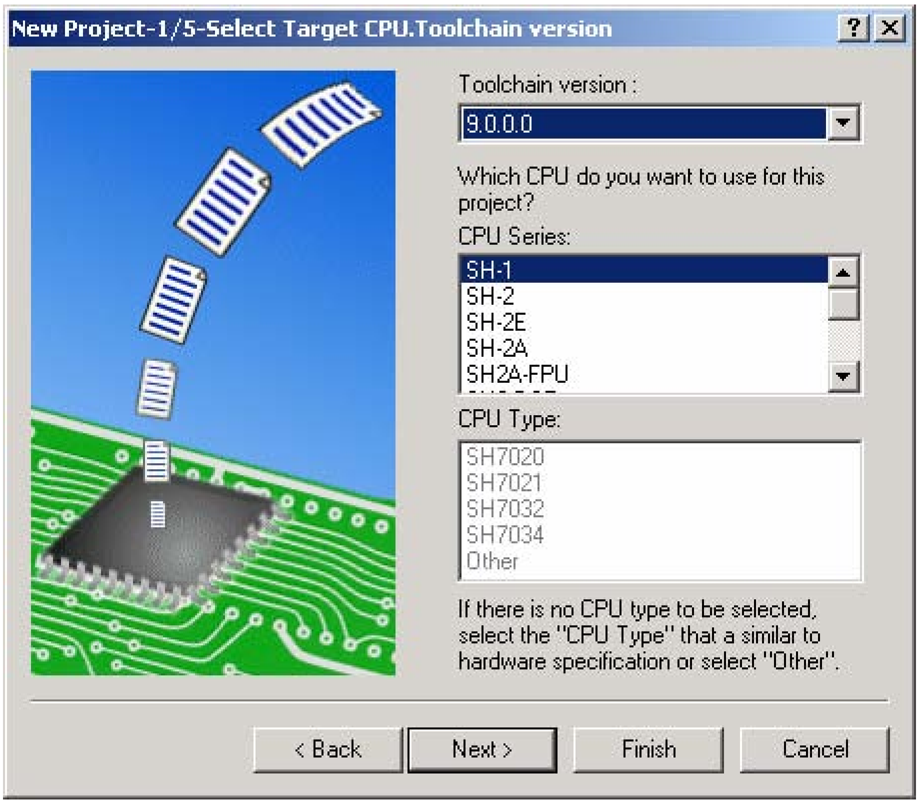
When you click [OK] on the [New Project Workspace] dialog box, the project generator will be invoked. Start by selecting the CPU that you will be using. CPU types shown in the [CPU Type] list are classified into the CPU series shown in the [CPU Series] list.The selected items in the [CPU Series:] list box and the [CPU Type:] list box specify the files to be generated. Select the CPU type of the program to be developed. If the CPU type which you want to select is not displayed in the [CPU Type:] list, select a CPU type with similar hardware specifications or select [Other].

- Clicking [Next>] moves to the next display.
- Clicking [<Back] moves to the previous display or the previous dialog box.
- Clicking [Finish] opens the [Summary] dialog box.
- Clicking [Cancel] returns the display to the [New Project Workspace] dialog box.

[<Back], [Next>], [Finish], and [Cancel] are common buttons of all the wizard dialog boxes.

In this tutorial, select [SH-1] in the [CPU Series] list (figure 2.2). Then click [Next >].

If you have selected “demonstration” , you cannot select CPU Type.



New Project-1/5-Select Target CPU.Toolchain version

Toolchain version : 9.0.0.0

Which CPU do you want to use for this project?

CPU Series: SH-1 (selected), SH-2, SH-2E, SH-2A, SH2A-FPU

CPU Type: SH7020, SH7021, SH7032, SH7034, Other

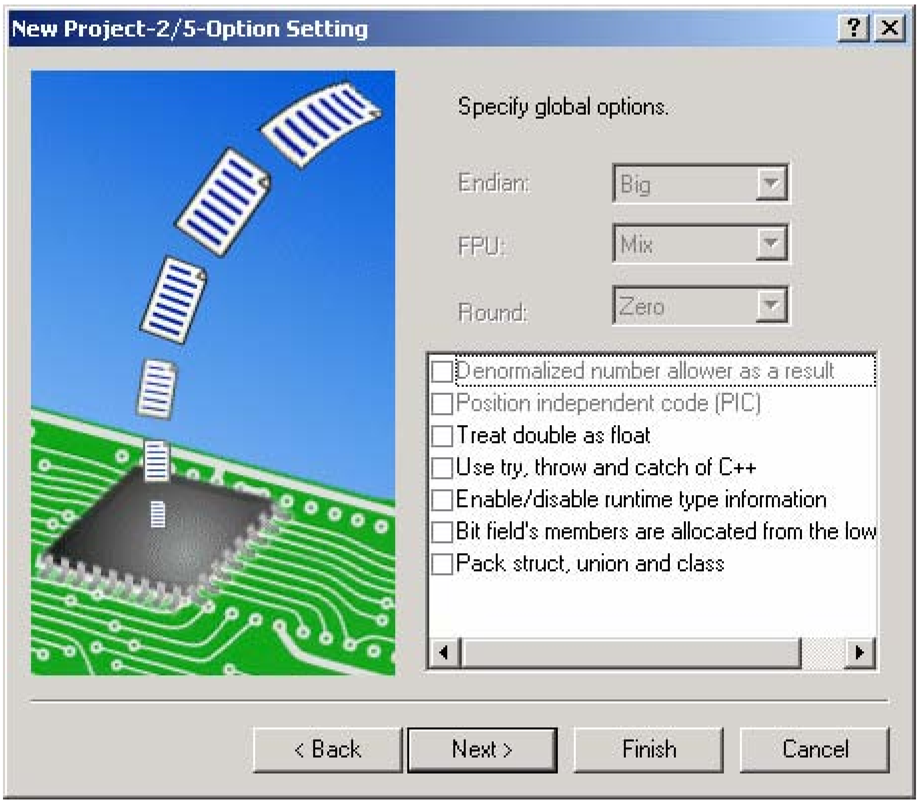
If there is no CPU type to be selected, select the "CPU Type" that a similar to hardware specification or select "Other".

Figure 2.2 New Project Step 1 Dialog Box

(3) Option Setting

Clicking the [Next>] button on the Step-1 screen opens the dialog box shown in figure 2.3. On this screen, you can specify options common to all project files. Settings for these options can be modified in correspondence with the CPU series selected in step 1 screen. To change the option settings after a project has been created, select the CPU tab from the [Options -> SuperH RISC engine Standard Toolchain] menu item of the HEW window.

Click the [Next>] button without changing the setting. The Step-3 screen will be displayed.



New Project-2/5-Option Setting

Specify global options.

Endian: Big

FPU: Mix

Round: Zero

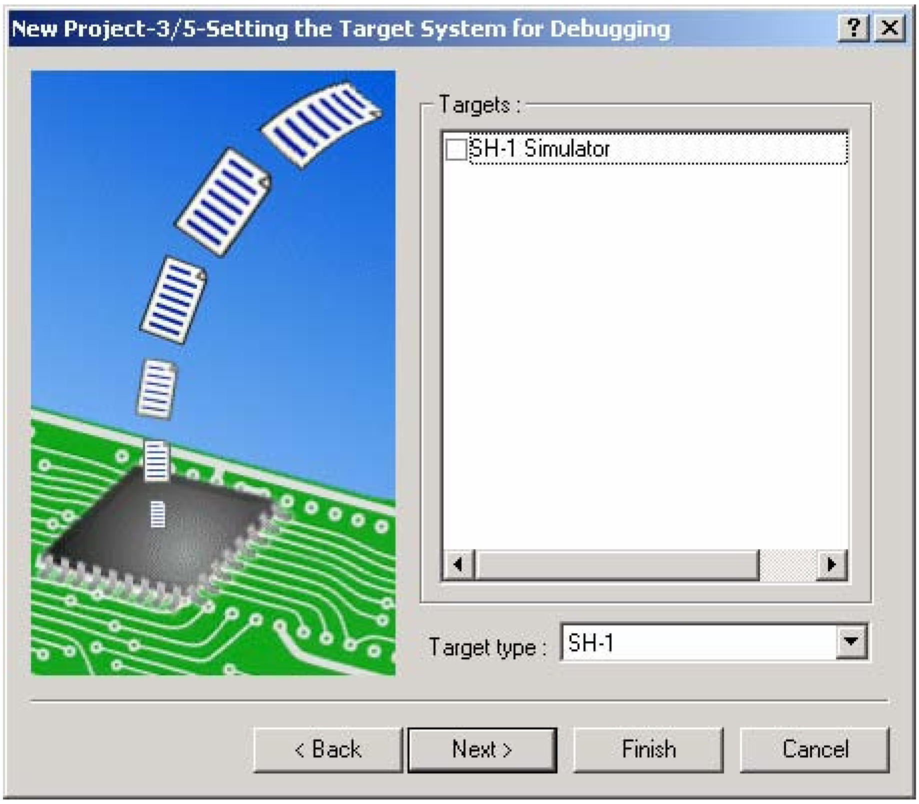
□ Denormalized number allower as a result
□ Position independent code (PIC)
□ Treat double as float
□ Use try, throw and catch of C++
□ Enable/disable runtime type information
□ Bit field's members are allocated from the low
□ Pack struct, union and class

Figure 2.3 New Project Step 2 Dialog Box

(4) Setting the Target System for Debugging

When the [Next>] button is clicked in the Step-2 screen, the screen shown in figure 2.4 is displayed. This screen is used to specify the target system for debugging. Select (check) the target for debugging from the list under [Target:]. Selection of no target or of multiple targets is allowed.

In this tutorial, select [SH-1 Simulator] and then click the [Next>] button.



New Project-3/5-Setting the Target System for Debugging

Targets :

□ SH-1 Simulator

Target type : SH-1

Figure 2.4 New Project Step 3 Dialog Box

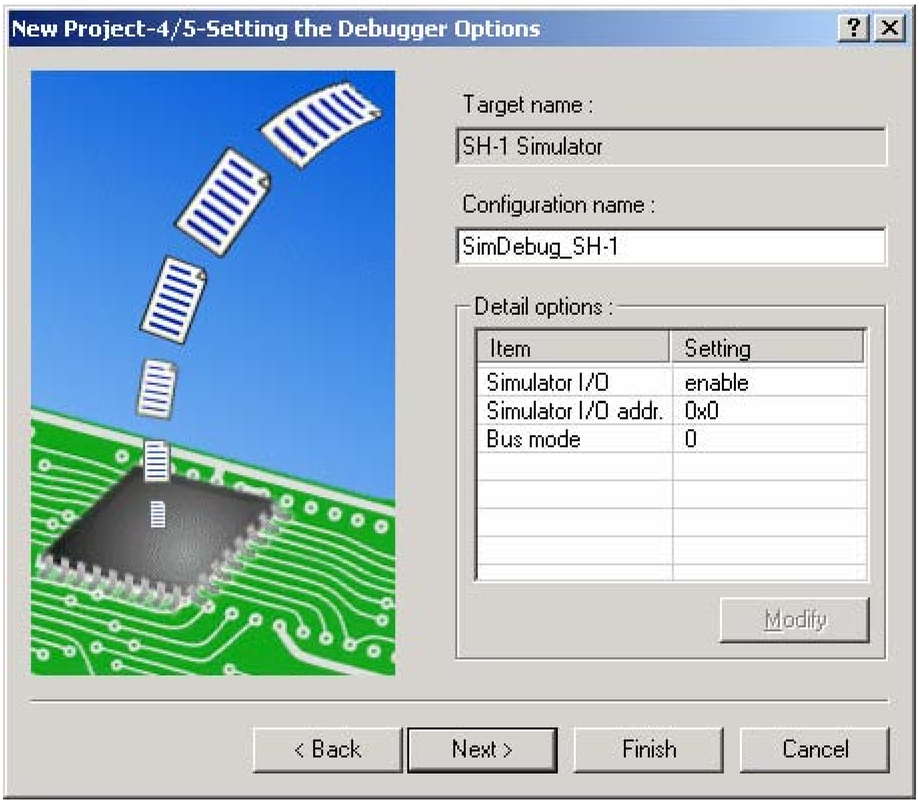
(5) Setting the Debugger Options

When the [Next>] button is clicked in the Step-3 screen, the screen shown in figure 2.5 is displayed. This screen is used to specify the optional settings for the selected target for debugging.

By default, the HEW creates two configurations, [Release] and [Debug]. When a target for debugging is selected, the HEW creates another configuration (The name of the target is included).

The name of the configuration can be modified in [Configuration name:]. Options to do with the target for debugging are displayed under [Detail options:]. To change the settings, select [Item] and then click [Modify]. When items for which modification is not possible are selected, [Modify] remains grayed even if [Item] is selected.

In this tutorial, click the [Next>] button without changing the settings.



New Project-4/5-Setting the Debugger Options

Target name : SH-1 Simulator

Configuration name : SimDebug_SH-1

Detail options :

Item	Setting
Simulator I/O	enable
Simulator I/O addr.	0x0
Bus mode	0

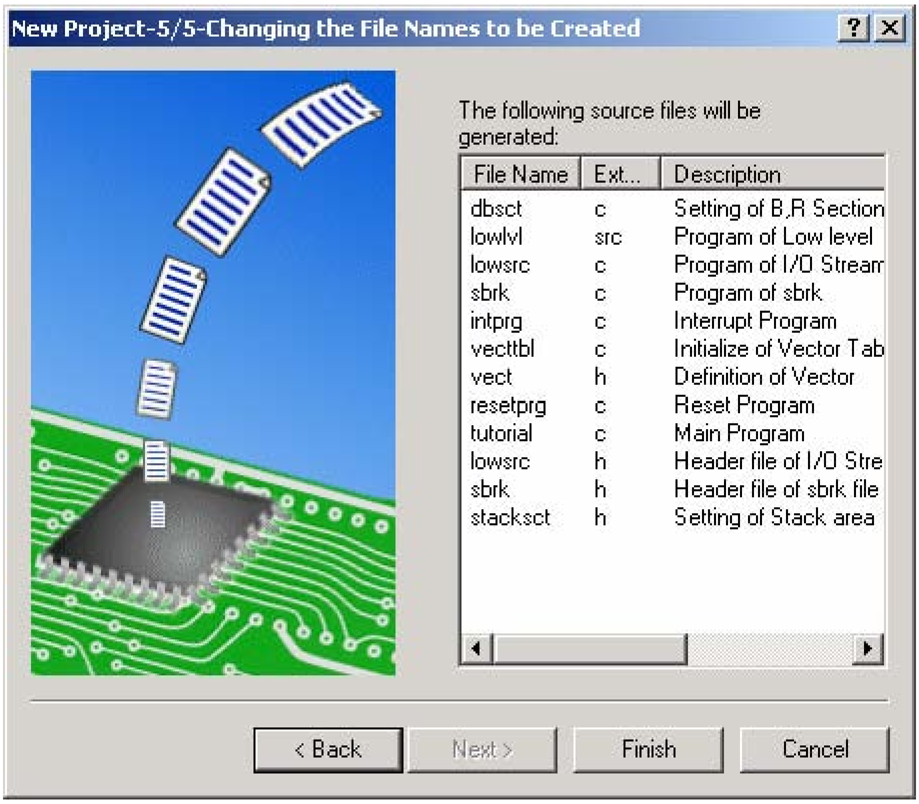
Figure 2.5 New Project Step 4 Dialog Box

(6) Confirming Settings (Summary Dialog Box)

Clicking on [Next >] on the Step-4 screen displays the screen shown in figure 2.6. On this screen, display the source file information for the project to be created. After confirmation, click [Finish].

Clicking [Finish] on the Step-5 screen, the project generator shows a list of generated files on the [Summary] dialog box (figure 2.7). Confirm the contents of the dialog box and click [OK].

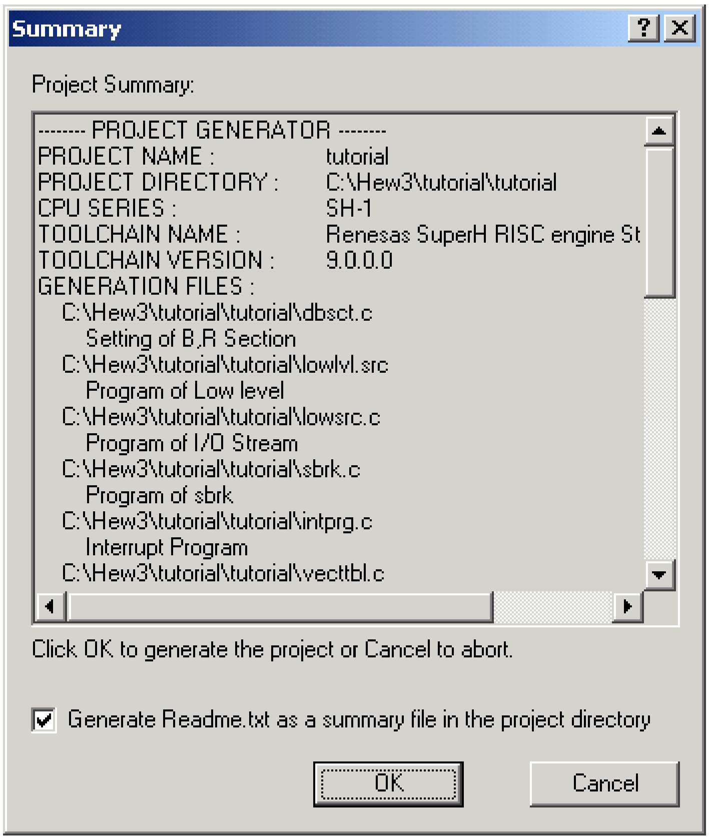
When [Generate Readme.txt as a summary file in the project directory] checkbox is checked, the project information displayed on the [Summary] dialog box will be stored in the project directory under the text file name "Readme.txt".



The following source files will be generated:

File Name	Ext...	Description
dbsct	c	Setting of B,R Section
lowlvl	src	Program of Low level
lowsrc	c	Program of I/O Stream
sbrk	c	Program of sbrk
intprg	c	Interrupt Program
vecttbl	c	Initialize of Vector Tab
vect	h	Definition of Vector
resetprg	c	Reset Program
tutorial	c	Main Program
lowsrc	h	Header file of I/O Stre
sbrk	h	Header file of sbrk file
stacksct	h	Setting of Stack area

Figure 2.6 New Project Step 5 Dialog Box



```
-------- PROJECT GENERATOR --------
PROJECT NAME :          tutorial
PROJECT DIRECTORY :     C:\Hew3\tutorial\tutorial
CPU SERIES :            SH-1
TOOLCHAIN NAME :        Renesas SuperH RISC engine St
TOOLCHAIN VERSION :     9.0.0.0
GENERATION FILES :
  C:\Hew3\tutorial\tutorial\dbsct.c
    Setting of B,R Section
  C:\Hew3\tutorial\tutorial\lowlvl.src
    Program of Low level
  C:\Hew3\tutorial\tutorial\lowsrc.c
    Program of I/O Stream
  C:\Hew3\tutorial\tutorial\sbrk.c
    Program of sbrk
  C:\Hew3\tutorial\tutorial\intprg.c
    Interrupt Program
  C:\Hew3\tutorial\tutorial\vecttbl.c
```

Click OK to generate the project or Cancel to abort.

☑ Generate Readme.txt as a summary file in the project directory

Figure 2.7 Summary dialog box

(7) Other

If demonstration is selected from Project Type, low-level library sample that can be used at simulator debugging will be embedded. The files to be embedded are as follows.

- lowlvl.src (Standard I/O Sample Assembler List)
- lowsrc.c (Low-level Library Source File)
- lowsrc.h (Low-level Library Header File)

2.2 Introduction of Sample Program (SH-1, SH-2, SH-2E, SH-2A, SH2A-FPU, SH2-DSP)

In this subsection, a sample program with the structure shown in figure 2.8 is used to explain the actual procedure for creating a program. The development environment is described in table 2.1.

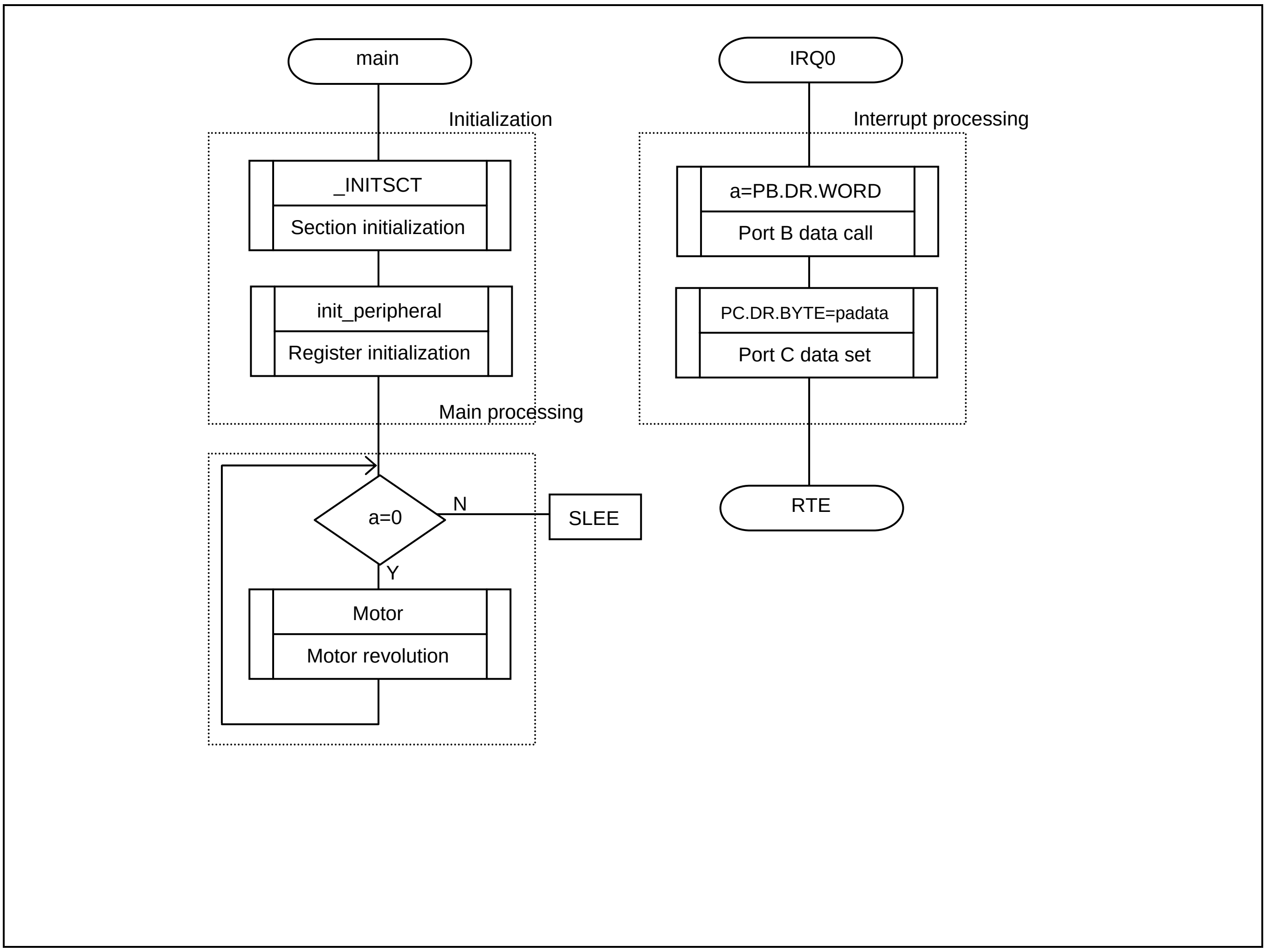


Figure 2.8 Sample Program Structure

Table 2.1 Sample Program Development Environment

OS	UNIX
CPU	SH-1

2.2.1 Creating a Vector Table

A vector table creation program appears in figure 2.9. For details on creating vector tables, refer to section 3.1.3, Creating Vector Tables.

Figure 2.9 shows the same program as in figure 2.10, written in assembly language.

```
/****************************************************************/
/*                         file name "vect.c"                  */
/****************************************************************/
      extern void main(void);
      extern void inv_inst(void);
      extern void IRQ0(void);

   void (* const vec_table[])(void)={
         main,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
         0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
         inv_inst, 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
         0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
         IRQ0
   };
```

Figure 2.9 Vector Table Creation Program (C Language Version)

The vector table for the SH-1 appears in table 2.2.

Upon power-on reset, the function main is started. At this time the stack pointer is set to 0.

The start address of the function inv_inst is set to the vector number 32, and the start address of the function IRQ0 is set to the vector number 64. These are start vector numbers for the user vector and for external interrupts, respectively.

Table 2.2 Exception Processing Vector Table

Exception Source	Vector Number		Vector Table Address Offset
Power-on reset	PC	0	H'00000000 to H'00000003
	SP	1	H'00000004 to H'00000007
Manual reset	PC	2	H'00000008 to H'0000000B
	SP	3	H'0000000C to H'0000000F
:		:	:
Trap instruction (User vector)		32	H'00000080 to H'00000083
		:	:
		63	H'000000FC to H'000000FF
Interrupt	IRQ0	64	H'00000100 to H'00000103
	:	:	:

```
.SECTION VECT,DATA,ALIGN=4
.IMPORT      _main
.IMPORT      _inv_inst
.IMPORT      _IRQ0
.DATA.L      _main                 ;_main start address set to vector number 0
.DATA.L      H'0000000             ;SP initial value set to vector number 1
.ORG         H'0080
.DATA.L      _inv_inst             ;_inv_inst start address set to vector number 32
.ORG         H'0100
.DATA.L      _IRQ0                 ;_IRQ0 start address set to vector number 64
.END
```

Figure 2.10 Vector Table Creation Program (Assembly Language Version)

In the assembly language program, an underscore "_" is prefixed to the external names of the C language program.

2.2.2 Creating a Header File

Figure 2.11 shows a header file used in common by all the sample programs. By defining IPRA and other I/O ports, the I/O ports can be accessed by name as if they were variables.

```
/***********************************************************************/
/*                    file name "7032.h" (Extract)                     */
/***********************************************************************/
/***********************************************************************/
/*                    Definitions of I/O Registers                     */
/***********************************************************************/
struct st_intc { /* struct INTC       */
                  union {     /*  IPRA        */
                             unsigned short WORD;/*Word Access       */
                  struct {    /*  Bit   Access */
                          unsigned char UU:4;        /*  IRQ0        */
                          unsigned char UL:4;        /*  IRQ1        */
                          unsigned char LU:4;        /*  IRQ2        */
                          unsigned char LL:4;        /*  IRQ3        */
                          } BIT;     /*              */
                   } IPRA;    /*              */
                  union {     /*  IPRB        */
                         unsigned short WORD;        /*  Word Access */
                         struct {     /*  Bit   Access */
                          unsigned char UU:4;        /*  IRQ4        */
                          unsigned char UL:4;        /*  IRQ5        */
                          unsigned char LU:4;        /*  IRQ6        */
                          unsigned char LL:4;        /*  IRQ7        */
                          } BIT;     /*              */
                   }  IPRB;   /*              */
};  /*     */
```

```
#define INTC  (*(volatile struct st_intc  *)0x5FFFF84)
   /* INTC Address  */
/************************************************************************/
/*                              Timer registers                        */
/************************************************************************/
struct st_itu0 { /* struct ITU0       */
                  union {     /* TCR  */
               unsigned char BYTE;    /* Byte Access  */
               struct {/* Bit  Access        */
                          unsigned char  :1; /*      */
                          unsigned char CCLR :2;     /*  CCLR        */
                          unsigned char CKEG :2;     /*  CKEG        */
                          unsigned char TPSC :3;     /*  TPSC        */
                        }   BIT;      /*      */
                  }    TCR;   /*      */
}; /* */
 #define ITU0  (*(volatile struct st_itu0  *)0x5FFFF04)
   /* ITU0 Address  */
/************************************************************************/
/*                              PORT registers                         */
/************************************************************************/
struct st_pa {   /*  struct PA        */
               union {  /*  PADR      */
                  unsigned short  WORD;       /*  Word Access */
                      struct {        /*  Byte Access */
                             unsigned char H;        /*  High        */
                             unsigned char L;        /*  Low */
                 }   BYTE;    /*      */
                                struct {     /*  Bit Access  */
                                     unsigned char B15 :1;   /*  Bit 15*/
                                     unsigned char B14 :1;   /*  Bit 14*/
                                     unsigned char B13 :1;   /*  Bit 13*/
                                     unsigned char B12 :1;   /*  Bit 12*/
                                     unsigned char B11 :1;   /*  Bit 11*/
                                     unsigned char B10 :1;   /*  Bit 10*/
                                     unsigned char B9  :1;   /*  Bit 9*/
                                     unsigned char B8  :1;   /*  Bit 8*/
                                     unsigned char B7  :1;   /*  Bit 7*/
                                     unsigned char B6  :1;   /*  Bit 6*/
                                     unsigned char B5  :1;   /*  Bit 5*/
                                     unsigned char B4  :1;   /*  Bit 4*/
                                     unsigned char B3  :1;   /*  Bit 3*/
                                     unsigned char B2  :1;   /*  Bit 2*/
                                     unsigned char B1  :1;   /*  Bit 1*/
                                     unsigned char B0  :1;   /*  Bit 0*/
```

```
                                       }   BIT;      /*      */
                            }    DR;    /*      */
}; /* */
#define PB   (*(volatile struct st_pa  *)0x5FFFFC2)
   /*  PB  Address  */
struct st_pc {   /*  struct PC        */
            union { /*  PCDR  */
                   unsigned char  BYTE;      /*  Byte Access */
                   struct {   /*  Bit Access  */
                                      unsigned char B7 :1;   /*  Bit 7       */
                                      unsigned char B6 :1;   /*  Bit 6       */
                                      unsigned char B5 :1;   /*  Bit 5       */
                                      unsigned char B4 :1;   /*  Bit 4       */
                                      unsigned char B3 :1;   /*  Bit 3       */
                                      unsigned char B2 :1;   /*  Bit 2       */
                                      unsigned char B1 :1;   /*  Bit 1       */
                                      unsigned char B0 :1;   /*  Bit 0       */
                            }  BIT;    /*      */
                   }   DR;     /*      */
}; /* */
#define PC  (*(volatile struct st_pc *)0x5FFFFD1)
   /* PC  Address   */
/*************************************************************************/
/*                              file name "sample.h"                    */
/*************************************************************************/
/*************************************************************************/
/*                              Timer registers                         */
/*************************************************************************/
 struct tcsr {   /*  */
                 short OVF  :1;      /*TCSR struct OVF bit                 */
                 short WTIT :1;      /* WTIT bit                           */
                 short   :3;  /* work area                                 */
                 short CKS2 :1;      /* CKS2 bit                           */
                 short CKS1 :1;      /* CKS1 bit                           */
                 short  :9;   /* work area    */
}; /* */
#define TCSR_FRT (*(volatile unsigned short *)0x5FFFFB8)
   /* */
#define TCSR__FRT (*(volatile struct tcsr *)0x5FFFFB8)
   /* */
    extern void motor( void );       /* motor module */
    extern void _INITSCT( void );
   /* section initialize module      */
    extern void init_peripheral(void);
   /* peripheral initialize module   */
```

Figure 2.11 Header File

2.2.3 Creating the Main Processing Program

The main processing program is shown in figure 2.12. Here the function main, which is started upon power-on reset, and the function motor, which is called continuously until an interrupt occurs, are defined.

```
/***********************************************************************/
/*                             file name "sample.c"                    */
/***********************************************************************/
 #include "7032.h"
 #include "sample.h"
 #include <machine.h>         /*Define embedded function sleep       */
const short padata=0x3;       /* C section                           */
 short a=0;                   /* D section                           */
 int work;                           /* B section                           */
/***********************************************************************/
/*                             main module                             */
/***********************************************************************/
 void main( void )
 {
       _INITSCT();            /* Initialize each section             */
        init_peripheral();
        while(!a) motor();
        sleep();
 }
/***********************************************************************/
/*                              motor module                           */
/***********************************************************************/
 void motor( void )           /*Call until an interrupt occurs       */
 {
            :
            :
        return;
 }
```

Figure 2.12 Main Processing Program

In the function main, _INITSCT and init_peripheral are called to perform section initialization and internal register initialization. Then the program waits for a change in the value of the global variable a. During this time, the function motor is continuously called. If the value of a changes from zero, the low-power consumption state is entered.

2.2.4 Creation of the Initialization Unit

Figure 2.13 shows an assembly language program which sets the values for external names used in section initialization; Figure 2.14 shows a C language program which performs section initialization and register initialization.

```
;******************************************************************
;                         file name "sct.src"                     *
;******************************************************************
                                    .SECTION B,DATA,ALIGN=4
                                    .SECTION R,DATA,ALIGN=4
                                    .SECTION D,DATA,ALIGN=4
          ; any sections to be added are listed here

                                    .SECTION C,DATA,ALIGN=4
          __B_BGN:                  .DATA.L (STARTOF B)
          __B_END:                  .DATA.L (STARTOF B)+(SIZEOF B)
          __D_BGN:                  .DATA.L (STARTOF R)
          __D_END:                  .DATA.L (STARTOF R)+(SIZEOF R)
          __D_ROM:                  .DATA.L (STARTOF D)

                                    .EXPORT __B_BGN
                                    .EXPORT __B_END
                                    .EXPORT __D_BGN
                                    .EXPORT __D_END
                                    .EXPORT __D_ROM
                                    .END
```

Figure 2.13 Initialization Program (Assembly Language Part)

The start and end addresses of the B section and D section are defined.

At compile time, if section options are not used to specify section names, the C/C++ compiler automatically assigns the following names.

Program section:	P
Constants section:	C
Initialization data section:	D
Uninitialized data section:	B

The R section shows the RAM area to which initialization data area on the ROM is copied using the ROM support functions of the linkage editor. For more information on the ROM support functions of the linkage editor, refer to section 3.15.2 (1) , ROM Support Function.

STARTOF is an operator which determines the start address of sections, using the format "STARTOF <section name>".

SIZEOF is an operator which determines the size of a section, in byte units, using the format "SIZEOF <section name>".

```
/*********************************************************************/
/*                                  file name "init.c"               */
/*********************************************************************/
   #include "7032.h"
   #include "sample.h"
/*********************************************************************/
/*                           section initialize module               */
/*********************************************************************/
   extern int *_B_BGN,*_B_END,*_D_BGN,*_D_END,*_D_ROM;
   void _INITSCT(void)
   {
          register int *p,*q;
          for (p=_B_BGN; p<_B_END; p++)
                 *p=0;
          for (p=_D_BGN; q=_D_ROM, p<_D_END; p++,q++)
                 *p=*q;
 }
/******************************************************************** /
/*                     peripheral initialize module                 * /
/******************************************************************** /
   void init_peripheral(void)
   {
          INTC.IPRA.WORD = 0x3000;                  /* Initialize IPRA */
          ITU0.TCR.BYTE = 0x02;                     /* Initialize TCR0 */
          TCSR_FRT = 0x5A01;                        /* Initialize TCSR */
          PB.DR.WORD = 0x80;                        /* Initialize PORT */
   }
```

Figure 2.14 Initialization Program (C Language Part)

In the section initialization module _INITSCT, B section zero-clearing and copying of ROM initialization data to RAM are performed based on the section address specified in sct.src. The int type specifier is used, but if the size is other than 4n bytes, char should be used.

In the internal register initialization module init_peripheral, the following settings are performed.

- In the interrupt priority level register A, the IRQ0 interrupt priority level is set to 3.
- In the timer control register 0, clearing of the timer counter 0 of the 16-bit integrated timer pulse unit is forbidden, counting is set for the rising edge, and internal clock is set to count at ϕ/4.
- The timer counter for the watchdog timer is set to 0x01.
- Port B is set to 0x80.

2.2.5 Creating Interrupt Functions

Figure 2.15 shows interrupt functions. The external interrupt handler function IRQ0 and trap instruction function inv_inst are defined.

```
/************************************************************************/
/*                           file name "int.c"                         */
/************************************************************************/
   #include "7032.h"
   #include "sample.h"
   extern const short padata;               /*      C section       */
   extern short a;                          /*      D section       */
   extern int work;                         /*      B section       */
   #pragma interrupt(IRQ0, inv_inst)
/************************************************************************/
/*                           interrupt module IRQ0                     */
/************************************************************************/
   void IRQ0(void)
   {
          a = PB.DR.WORD;
          PC.DR.BYTE = padata;
   }
/************************************************************************/
/*                           interrupt module inv_inst                 */
/************************************************************************/
   void inv_inst(void)
   {
          return;
   }
```

Figure 2.15 Interrupt Functions

The function IRQ0 sets the global variable a to PB.DR.WORD (0x80) when an IRQ0 external interrupt occurs. By this means the CPU is put into a low-power consumption state.

2.2.6 Creating a Batch File for a Load Module

Figure 2.16 shows a batch file used to create an S-type load module (sample.mot).

```
shcΔ-debugΔsample.cΔinit.cΔint.c
#Compile C programs
asmshΔsct.srcΔ–debug
#Assemble Assembly programs
shcΔ-debugΔ-section=c=VECTΔvect.c
#Compile vector table creation programs
optlnkΔ-nooptΔ-subcommand=rom.sub
#Link using a subcommand file
rmΔ*.obj
#Remove object module files
```

Figure 2.16 Batch File to Create a Load Module

Here vect.c is compiled into an independent file, and the option section=VECT is used to make it a section separate from other initialization data units. On linking it is allocated addresses starting from 0.

2.2.7 Creating a Linkage Editor Subcommand File

Figure 2.17 shows a subcommand file (filename rom.sub) for the linkage editor used when creating load modules.

```
Sdebug
input sample,init,int,vect,sct
                   ; Specify input files
library /user/unix/SHCV5.0/shclib.lib
                   ; Specify a standard library
output sample.abs   sample.abs ; Specify an output filename
rom D=R            ; Specify ROM support options
start VECT/0,P,C,D/0400,R,B/F0000000
                   ; Specify the start addresses for each section
                   ; Allocate section VECT starting from address 0
                   ; Allocate sections P, C, D to the area starting from
                     address H'400
                   ; Allocate sections R, B to the area starting from address
                     H'F0000000
form s             ; Specify s-type format
list  sample.map   ; Specify memory map information output
Exit
```

Figure 2.17 Subcommand File for Linkage Editor

2.3 Introduction of Sample Program (SH-3,SH3-DSP,SH-4,SH-4A, and SH4AL-DSP)

In this subsection, a sample program is introduced for the case of the SH7708. The sample program introduced here performs processing from reset until execution is passed to the main() function. This is an example of the smallest program necessary when the CPU is started.

2.3.1 Creating an Interrupt Handler

In contrast with the SH-1, SH-2, SH-2E, SH-2A, SH2A-FPU, and SH2-DSP, in the cases of the SH-3 , SH3-DSP, SH-4, SH-4A, and SH4 AL-DSP vector control in the event of an interrupt must in essence be specified in software.

Fixed addresses for interrupts in the SH-3 are set to the PC (program counter) for three different causes: reset, exceptions, and interrupts. Hence routines to determine the interrupt factor, and for branching to interrupt processing for each factor, must be written at each of these addresses as interrupt handlers.

Individual handlers are explained in detail below. Here an example is given in which the vector base register (VBR) is fixed at H'00000000, and the memory management unit (MMU) is unused.

(1) Reset handler (address H'00000000)

At power-on or manual reset, the PC is set to H'a0000000. Because addresses H'00000000 and H'a0000000 correspond to a common physical address, the program is placed at H'00000000. Here the following steps are performed:

- Exception judgment is performed by EXEVT
- The processing routine is called from the vector table

Processing is shown in figure 2.18.

```
;********************************************************************
;          file name "reset.src"
;********************************************************************
;  SH7708 Reset handler Routine
          .IMPORT      _vecttbl
          .IMPORT      _stacktbl
          .SECTION  VECT,CODE,LOCATE=H’0
__reset:
;************************************************************
;          You should initialize the stack RAM area by BSC
;          before set the stack pointer "R15"
;************************************************************
; exsample ) AREA1 (CS1) -> STACK RAM
; AREA1
; Bus size ->16bit
; D23-D16 ->not PORT
; wait 3 state
; BCR2>> PORTEN:A1SZ0:A1SZ0
; 0: 1 :0
; >> BCR2=0x3fff8
          MOV.L     BSCR2,R0
          MOV.L     #H'3fff8,R1
          MOV.W   R1,@R0
; WCR2>> A1-2W1:A1-2W0
; 1: 1
; >> WCR2=0xffff
```

```
          MOV.L   WCR2,R0
          MOV.L   #H'ffff,R1
          MOV.W  R1,@R0
;*************************************************************
          MOV.L  VECTadr,R1
          MOV.L  STACKadr,R2
          MOV.L     EXPEVT,R0
          MOV.L     @R0,R0
          CMP/EQ    #0,R0      ;POWER ON RESET
          BT     PON_RESET
          CMP/EQ    #H'20,R0
          BT        MANUAL_RESET
          ; if( EXPEVT != RESET)
          ; while(1);
LOOP
   BRA       LOOP
          NOP
PON_RESET
          MOV.L     @(0,R1),R1       ;set function
          MOV.L     @(0,R2),R15 ;set stack pointer
          JMP       @R1
          NOP
MANUAL_RESET
          MOV.L     @(4,R1),R1       ;set function
          MOV.L     @(4,R2),R15 ;set stack pointer
          JMP       @R1
          NOP
;
          .ALIGN       4
VECTadr   .DATA.L      _vecttbl
STACKadr  .DATA.L      _stacktbl
EXPEVT    .DATA.L      H'ffffffd4
BSCR2     .DATA.L      H'ffffff62
WCR2      .DATA.L      H'ffffff66
          .END
```

Figure 2.18 Reset Handler Program

(2) General exception-processing handler (VBR+H'100)

- The exception factor code is read from EXPEVT.
- The processing function (vector function) for this factor is read from the vector table.
- The terminate routine is set.
- Execution jumps to the vector function.

In this case, RTE instructions are used in order to jump to the vector function. Here the PR register value is changed immediately before jumping to the vector function, such that on returning from the vector function, control is passed to the terminate routine. As PR during vector function processing is the terminate routine, it is necessary to return the vector function by executing RTS. For this reason, when defining the vector function, do not use ‘#pragma interrrupt’.

(3) VBR+H'400 TLB miss exception handler

Because MMU is unused, this is not included.

(4) VBR+H'600 interrupt handler

- The interrupt factor code is read from INTEVT.
- The processing function (vector function) for this factor is read from the vector table.
- The interrupt mask level for this factor is set from the interrupt mask table.
- The terminate routine is set.
- Execution jumps to the vector function.

This processing is in essence the same as for general exception handlers; on returning from the vector function, control is passed to the terminate routine.

```
;*******************************************************************
; FILE :vhandler.src
;*******************************************************************
       .include  "env.inc"
       .include  "vect.inc"

IMASKclr: .equ   H'FFFFFF0F
RBBLclr:  .equ   H'CFFFFFFF
MDRBBLset:   .equ   H'70000000
       .import      _RESET_Vectors
       .import      _INT_Vectors
       .import      _INT_MASK
;*******************************************************************
;* macro definition
;*******************************************************************
.macro PUSH_EXP_BASE_REG
stc.l ssr,@-r15 ; save ssr
stc.l spc,@-r15 ; save spc
sts.l pr,@-r15 ; save context registers
stc.l r7_bank,@-r15
stc.l r6_bank,@-r15
stc.l r5_bank,@-r15
stc.l r4_bank,@-r15
stc.l r3_bank,@-r15
stc.l r2_bank,@-r15
stc.l r1_bank,@-r15
stc.l r0_bank,@-r15
.endm
;
.macro POP_EXP_BASE_REG
ldc.l @r15+,r0_bank ; recover registers
ldc.l @r15+,r1_bank
ldc.l @r15+,r2_bank
ldc.l @r15+,r3_bank
ldc.l @r15+,r4_bank
ldc.l @r15+,r5_bank
ldc.l @r15+,r6_bank
ldc.l @r15+,r7_bank
lds.l @r15+,pr
ldc.l @r15+,spc
ldc.l @r15+,ssr
.endm
;*******************************************************************
;      reset
```

```
;*******************************************************************
       .section  RSTHandler,code
_ResetHandler:
              mov.l  #EXPEVT,r0
              mov.l  @r0,r0
              shlr2  r0
              shlr   r0
              mov.l  #_RESET_Vectors,r1
              add    r1,r0
              mov.l  @r0,r0
              jmp    @r0
              nop
;*******************************************************************
;      exceptional interrupt
;*******************************************************************
       .section  INTHandler,code
       .export   INTHandlerPRG
INTHandlerPRG:
_ExpHandler:
              PUSH_EXP_BASE_REG
;
              mov.l  #EXPEVT,r0       ; set event address
              mov.l  @r0,r1           ; set exception code
              mov.l  #_INT_Vectors,r0 ; set vector table address
              add #-(h'40),r1         ; exception code - h'40
              shlr2  r1
              shlr   r1
              mov.l  @(r0,r1),r3      ; set interrupt function addr
;
              mov.l  #_INT_MASK,r0            ; interrupt mask table addr
              shlr2  r1
              mov.b  @(r0,r1),r1              ; interrupt mask
              extu.b r1,r1
;
              stc sr,r0                       ; save sr
              mov.l  #(RBBLclr&IMASKclr),r2   ; RB,BL,mask clear data
              and r2,r0                       ; clear mask data
              or     r1,r0                    ; set interrupt mask
              ldc r0,ssr                      ; set current status
;
              ldc.l  r3,spc
              mov.l  #__int_term,r0           ; set interrupt terminate
              lds r0,pr
;
              rte
              nop
;
              .pool
;
;*******************************************************************
;      Interrupt terminate
;*******************************************************************
```

```
             .align 4
__int_term:
             mov.l #MDRBBLset,r0             ; set MD,BL,RB
             ldc.l  r0,sr                    ;
             POP_EXP_BASE_REG
             rte                             ; return
             nop
;
             .pool
;
;*******************************************************************
;      TLB miss interrupt
;*******************************************************************
       .org   H'300
_TLBmissHandler:
             PUSH_EXP_BASE_REG
;
             mov.l  #EXPEVT,r0               ; set event address
             mov.l  @r0,r1                   ; set exception code
             mov.l  #_INT_Vectors,r0          ; set vector table address
             add #-(h'40),r1                 ; exception code - h'40
             shlr2  r1
             shlr   r1
             mov.l  @(r0,r1),r3              ; set interrupt function addr
;
             mov.l  #_INT_MASK,r0            ; interrupt mask table addr
             shlr2  r1
             mov.b  @(r0,r1),r1              ; interrupt mask
             extu.b r1,r1
;
             stc sr,r0                       ; save sr
             mov.l  #(RBBLclr&IMASKclr),r2   ; RB,BL,mask clear data
             and r2,r0                       ; clear mask data
             or     r1,r0                    ; set interrupt mask
             ldc r0,ssr                      ; set current status
;
             ldc.l  r3,spc
             mov.l  #__int_term,r0           ; set interrupt terminate
             lds    r0,pr
;
             rte
             nop
;
             .pool
;
;*******************************************************************
;      IRQ
;*******************************************************************
       .org   H'500
_IRQHandler:
             PUSH_EXP_BASE_REG
;
```

```
             mov.l  #INTEVT,r0               ; set event address
             mov.l  @r0,r1                   ; set exception code
             mov.l  #_INT_Vectors,r0         ; set vector table address
             add #-(h'40),r1                 ; exception code - h'40
             shlr2  r1
             shlr   r1
             mov.l  @(r0,r1),r3              ; set interrupt function addr
;
             mov.l  #_INT_MASK,r0            ; interrupt mask table addr
             shlr2  r1
             mov.b  @(r0,r1),r1              ; interrupt mask
             extu.b r1,r1
;
             stc sr,r0                       ; save sr
             mov.l  #(RBBLclr&IMASKclr),r2   ; RB,BL,mask clear data
             and r2,r0                       ; clear mask data
             or     r1,r0                    ; set interrupt mask
             ldc r0,ssr                      ; set current status
;
             ldc.l  r3,spc
             mov.l  #__int_term,r0           ; set interrupt terminate
             lds r0,pr
;
             rte
             nop
;
             .pool
             .end
```

Figure 2.19 Interrupt Handler Program

Note: The include files “env.inc" and "vect.inc" are automatically generated by HEW when an SH3 project is created.

2.3.2 Creating the Vector Table

(1) Vector table <vect.c>

Here the vector table, the interrupt priority table, and the TRAPA function table are described. The names for each factor are registered in this table, and actual user-created function names are given in the vect7708.h header file.

```
/*****************************************************************/
/* FILE NAME "vect.c"                                            */
/*****************************************************************/
#include "vect7708.h"

/*****************************************************************/
/* ALLOCATE STACK AREA                                           */
/*****************************************************************/
#pragma section STK /* SECTION name "BSTK" */
long stack[STACK_SIZE];
#pragma section
/*****************************************************************/
/* ALLOCATE DEFINITION TABLE                                     */
/*****************************************************************/
```

```
const void *stacktbl[]={
   STACK_PON,
   STACK_MANUAL
};
/*****************************************************************/
/* ALLOCATE VECTOR TABLE (EXPEVT or INTEVT CODE H'000-H'5a0)     */
/*****************************************************************/
void (*const vecttbl[])(void) = {
                              /* EVT KIND CODE REG
   */
   RESET_PON,                 /* PON RESET           H'000   EXPEVT          */
   RESET_MANUAL,              /* MANUAL RESET        H'020   EXPEVT          */
   TLB_MISS_READ,             /* TLB MISS(R)         H'040   EXPEVT          */
   TLB_MISS_WRITE,            /* TLB MISS(W)         H'060                   */
   TLB_1ST_PAGE,              /*                     H'080                   */
   TLB_PROTECT_READ,          /*                     H'0a0                   */
   TLB_PROTECT_WRITE,         /*                     H'0c0                   */
   ADR_ERROR_WRITE,           /*                     H'0e0                   */
   ADR_ERROR_WRITE,           /*                     H'100                   */
   RESERVED,                  /*                     H'120   ------          */
   RESERVED,                  /*                     H'140   ------          */
   TRAP,                      /*                     H'160   (with TRA)      */
   ILLEGAL_INST,              /*                     H'180   EXPEVT          */
   ILLEGAL_SLOT,              /*                     H'1a0   EXPEVT          */
   NMI,                       /*                     H'1c0   INTEVT          */
   USER_BREAK,                /*                     H'1e0   EXPEVT          */
   IRQ15,                     /*                     H'200   INTEVT          */
   IRQ14,                     /*                     H'220   INTEVT          */
   IRQ13,                     /*                     H'240   INTEVT          */
   IRQ12,                     /*                     H'260   INTEVT          */
   IRQ11,                     /*                     H'280   INTEVT          */
   IRQ10,                     /*                     H'2a0   INTEVT          */
   IRQ9,                      /*                     H'2c0   INTEVT          */
   IRQ8,                      /*                     H'2e0   INTEVT          */
   IRQ7,                      /*                     H'300   INTEVT          */
   IRQ6,                      /*                     H'320   INTEVT          */
   IRQ5,                      /*                     H'340   INTEVT          */
   IRQ4,                      /*                     H'360   INTEVT          */
   IRQ3,                      /*                     H'380   INTEVT          */
   IRQ2,                      /*                     H'3a0   INTEVT          */
   IRQ1,                      /*                     H'3c0   INTEVT          */
   RESERVED,                  /*                     H'3e0   ------          */
   TMU0_TUNI0,                /*                     H'400   INTEVT          */
   TMU1_TUNI1,                /*                     H'420   INTEVT          */
   TMU2_TUNI2,                /*                     H'440   INTEVT          */
   TMU2_TICPI2,               /*                     H'460   INTEVT          */
   RTC_ATI,                   /*                     H'480   INTEVT          */
   RTC_PRI,                   /*                     H'4a0   INTEVT          */
   RTC_CUI,                   /*                     H'4c0   INTEVT          */
   SCI_ERI,                   /*                     H'4e0   INTRVT          */
   SCI_RXI,                   /*                     H'500   INTRVT          */
   SCI_TXI,                   /*                     H'520   INTRVT          */
   SCI_TEI,                   /*                     H'540   INTRVT          */
   WDT_ITI,                   /*                     H'560   INTEVT          */
   REF_RCMI,                  /*                     H'580   INTEVT          */
   DEF_RPVI,                  /*                     H'5a0   INTEVT          */
   RESERVED
};
/***************************************************************/
/* ALLOCATE INTERRUPT PRIORITY TABLE INTEVT H'1c0-H'5a0         */
/***************************************************************/
const char imasktbl[]={
   15<<4,                     /*     NMI level 16(IMASK=0-15)                */
   IP_RESERVED,               /* ---------------
   */
   15<<4,                     /*     IRQ15 (IRL0000)                         */
   14<<4,                     /*     IRQ14 (IRL0001)                         */
   13<<4,                     /*     IRQ13 (IRL0010)                         */
   12<<4,                     /*     IRQ12 (IRL0011)                         */
   11<<4,                     /*     IRQ11 (IRL0100)                         */
   10<<4,                     /*     IRQ10 (IRL0101)                         */
    9<<4,                     /*     IRQ9 (IRL0110)                          */
    8<<4,                     /*     IRQ8 (IRL0111)                          */
    7<<4,                     /*     IRQ7 (IRL1000)                          */
    6<<4,                     /*     IRQ6 (IRL1001)                          */
    5<<4,                     /*     IRQ5 (IRL1010)                          */
    4<<4,                     /*     IRQ4 (IRL1011)                          */
    3<<4,                     /*     IRQ3 (IRL1100)                          */
    2<<4,                     /*     IRQ2 (IRL1101)                          */
    1<<4,                     /*     IRQ1 (IRL1110)                          */
   IP_RESERVED,               /*     ---------------                         */
   IP_TMU0,                   /*     TMU0 TUNI0                              */
   IP_TMU1,                   /*     TMU1 TUNI1                              */
   IP_TMU2,                   /*     TNU2 TUNI2                              */
   IP_TMU2,                   /*      TICPI2                                 */
   IP_RTC,                    /*     RTC ATI                                 */
   IP_RTC,
   IP_RTC,
   IP_SCI,                    /*     SCI      ERI                            */
   IP_SCI,
   IP_SCI,
   IP_SCI,
   IP_WDT,                    /*     WDT ITI                                 */
   IP_REF,                    /*     REF RCMI                                */
   IP_REF,                    /*     REF ROVI                                */
   IP_RESERVED
};

void (*const trap tbl[])(void)={
   TRAPA_0,
   TRAPA_1,
   TRAPA_2,
   TRAPA_3,
   TRAPA_4,
   TRAPA_5,
   TRAPA_6,
   TRAPA_7,
   TRAPA_8,
   TRAPA_9,
   TRAPA_10,
   TRAPA_11,
   TRAPA_12,
   TRAPA_13,
   TRAPA_14,
   TRAPA_15
};
```

Figure 2.20 Vector Table

(2) Vector function registration <vect7708.h>

The actual user-defined function names and other parameters are set. If interrupt processing functions are added, this area is changed.

Processing here includes:

- Definition of stack size
- Definition of vector function names for each factor
- Setting of interrupt priorities (values set to IPRA and IPRB)

A function called halt is here defined for unused vectors. The user functions themselves must be defined as interrupt functions using a #pragma interrupt declaration. Also, if a function is registered, the extern declaration for the function must appear in this file.

```
/*****************************************************************/
/* FILE NAME "vect7708.h"                                        */
/*****************************************************************/



/*****************************************************************/
/* STACK SIZE definition                                        */
/*****************************************************************/
#define STACK_SIZE           (0x4096/4)     /* 4096 byte */
#define STACK_PON            (&stack[STACK_SIZE])
#define STACK_MANUAL                 (&stack[STACK_SIZE])
extern long stack[];

/*****************************************************************/
/* RESET FUNCTION definition                                     */
/*****************************************************************/
#define    RESET_PON      init      /* PON RESET    H'000   EXPEVT  */
#define    RESET_MANUAL   init      /* MANUAL RESET H'020   EXPEVT  */
/*****************************************************************/
/* INTERRUPT FUNCTION definition                                    */
/********************************************************************/
#define   TLB_MISS_READ      halt    /* TLB MISS(R)  H'040   EXPEVT  */
#define   TLB_MISS_WRITE     halt    /* TLB MISS(W)  H'060   EXPEVT  */
#define   TLB_1ST_PAGE       halt    /*      H'080           EXPEVT  */
#define   TLB_PROTECT_READ   halt    /*      H'0a0           EXPEVT  */
```

```
#define   TLB_PROTECT_WRITE  halt    /*      H'0c0          EXPEVT  */
#define   ADR_ERROR_WRITE    halt    /*      H'0e0          EXPEVT  */
#define   ADR_ERROR_WRITE    halt    /*      H'100          EXPEVT  */
/*#define RESERVED           halt */ /*      H'120          ------  */
/*#define RESERVED           halt */ /*      H'140          ------  */
#define   TRAP               trap    /*      H'160 (with TRA)       */
#define   ILLEGAL_INST       halt    /*      H'180          EXPEVT  */
#define   ILLEGAL_SLOT       halt    /*      H'1a0          EXPEVT  */
#define   NMI                halt    /*      H'1c0          INTEVT  */
#define   USER_BREAK         halt    /*      H'1e0          EXPEVT  */
#define   IRQ15              irq15   /*      H'200          INTEVT  */
#define   IRQ14              halt    /*      H'220          INTEVT  */
#define   IRQ13              halt    /*      H'240          INTEVT  */
#define   IRQ12              halt    /*      H'260          INTEVT  */
#define   IRQ11              halt    /*      H'280          INTEVT  */
#define   IRQ10              halt    /*      H'2a0          INTEVT  */
#define   IRQ9               halt    /*      H'2c0          INTEVT  */
#define   IRQ8               halt    /*      H'2e0          INTEVT  */
#define   IRQ7               halt    /*      H'300          INTEVT  */
#define   IRQ6               halt    /*      H'320          INTEVT  */
#define   IRQ5               halt    /*      H'340          INTEVT  */
#define   IRQ4               halt    /*      H'360          INTEVT  */
#define   IRQ3               halt    /*      H'380          INTEVT  */
#define   IRQ2               halt    /*      H'3a0          INTEVT  */
#define   IRQ1               halt    /*      H'3c0          INTEVT  */
/*#define RESERVED           halt    *//*    H'3e0          ------  */
#define   TMU0_TUNI0         halt    /*      H'400          INTEVT  */
#define   TMU1_TUNI1         halt    /*      H'420          INTEVT  */
#define   TMU2_TUNI2         halt    /*      H'440          INTEVT  */
#define   TMU2_TICPI2        halt    /*      H'460          INTEVT  */
#define   RTC_ATI            halt    /*      H'480          INTEVT  */
#define   RTC_PRI            halt    /*      H'4a0          INTEVT  */
#define   RTC_CUI            halt    /*      H'4c0          INTEVT  */
#define   SCI_ERI            halt    /*      H'4e0          INTRVT  */
#define   SCI_RXI            halt    /*      H'500          INTRVT  */
#define   SCI_TXI            halt    /*      H'520          INTRVT  */
#define   SCI_TEI            halt    /*      H'540          INTRVT  */
#define   WDT_ITI            halt    /*      H'560          INTEVT  */
#define   REF_RCMI           halt    /*      H'580          INTEVT  */
#define   DEF_RPVI           halt    /*      H'5a0          INTEVT  */
#define   RESERVED           halt
extern void init(void);
extern void halt(void);
extern void _trap(void);
extern void irq15(void);

/********************************************************************/
/* INTERRUPT MASK definition                                        */
/********************************************************************/
#define IP_TMU0     (0<<4)
#define IP_TMU1     (0<<4)
#define IP_TMU2     (0<<4)
```

```
#define IP_RTC      (0<<4)
#define IP_SCI      (0<<4)
#define IP_WDT      (0<<4)
#define IP_REF      (0<<4)
#define IP_RESERVED    (15<<4)
/*******************************************************************/
/* IPRA,IPRB definition                                            */
/*******************************************************************/
#define WORD_IPRA   ((IP_TMU0<<12)|(IP_TMU1<<8)|(IP_TMU2<<4)|IP_RTC)
#define WORD_IPRB   ((IP_WDT<<12)|(IP_REF<<8)|(IP_SCI<<4)|0)
extern void set_ip(void);
extern long stack[];

/*******************************************************************/
/* TRAPA system call definition
             */
/*******************************************************************/
#define TRAPA_0     halt
#define TRAPA_1     halt
#define TRAPA_2     halt
#define TRAPA_3     halt
#define TRAPA_4     halt
#define TRAPA_5     halt
#define TRAPA_6     halt
#define TRAPA_7     halt
#define TRAPA_8     halt
#define TRAPA_9     halt
#define TRAPA_10    halt
#define TRAPA_11    halt
#define TRAPA_12    halt
#define TRAPA_13    halt
#define TRAPA_14    halt
#define TRAPA_15    halt /*#15(#0F) should be Exception routine(Illegal use )*/
```

Figure 2.21 Vector Function Name Definitions

2.3.3 Creating the Header File

The header file common to the sample program is shown below.

```
/*******************************************************************/
/*                     file name "7700s.h"       (Extract)         */
/*******************************************************************/
struct st_intc {                                    /*  struct INTC */
               union {                              /*    ICR       */
                     unsigned short WORD;           /*  Byte Access */
                     struct {                       /*  Bit  Access */
                            unsigned short NMIL:1;   /*    NMIL     */
                            unsigned short     :6;   /*             */
                            unsigned short NMIE:1;   /*    NMIE      */
                            }       BIT;             /*             */
                     }          ICR;                 /*             */
               union {                               /*    IPRA      */
```

```
                     unsigned short WORD;            /*  Word Access*/
                     struct {                        /*  Bit  Access*/
                            unsigned short UU:4;     /*    IRQ0     */
                            unsigned short UL:4;     /*    IRQ1     */
                            unsigned short LU:4;     /*    IRQ2     */
                            unsigned short LL:4;     /*    IRQ3     */
                            }       BIT;             /*             */
                     }          IPRA;                /*             */
               union {                               /*    IPRB     */
                     unsigned short WORD;            /*  Word Access*/
                     struct {                        /*  Bit  Access*/
                            unsigned short UU:4;     /*    IRQ4     */
                            unsigned short UL:4;     /*    IRQ5     */
                            unsigned short LU:4;     /*    IRQ6     */
                            unsigned short LL:4;     /*    IRQ7     */
                            }       BIT;             /*             */
                     }          IPRB;                /*             */
               char             wk1[234];            /*             */
               unsigned int   TRA;                   /*    TRA      */
               unsigned int   EXPEVT;                /*    EXPEVT   */
               unsigned int   INTEVT;                /*    INTEVT   */
};                                                   /*             */

union un_ccr {                                       /*  union CCR  */
             unsigned int       LONG;                  /*  Long Access*/
             struct {                                  /*  Bit  Access*/
                    unsigned int     :26;              /*             */
                    unsigned int RA  :1;               /*    RA       */
                    unsigned int     :1;               /*    0        */
                    unsigned int CF  :1;               /*    CF       */
                    unsigned int CB  :1;               /*    CB       */
                    unsigned int WT  :1;               /*    WT       */
                    unsigned int CE  :1;               /*    CE       */
                    }           BIT;                   /*             */
};                                                     /*             */
#define SCI   (*(volatile struct st_sci  *)0xFFFFFE80) /* SCI  Address*/
#define TMU   (*(volatile struct st_tmu  *)0xFFFFFE90) /* TMU  Address*/
#define TMU0  (*(volatile struct st_tmu0 *)0xFFFFFE94) /* TMU0 Address*/
#define TMU1  (*(volatile struct st_tmu0 *)0xFFFFFEA0) /* TMU1 Address*/
#define TMU2  (*(volatile struct st_tmu2 *)0xFFFFFEAC) /* TMU2 Address*/
#define RTC   (*(volatile struct st_rtc  *)0xFFFFFEC0) /* RTC  Address*/
#define INTC  (*(volatile struct st_intc *)0xFFFFFEE0) /* INTC Address*/
#define BSC   (*(volatile struct st_bsc  *)0xFFFFFF60) /* BSC  Address*/
#define CPG   (*(volatile struct st_cpg  *)0xFFFFFF80) /* CPG  Address*/
#define UBC   (*(volatile struct st_ubc  *)0xFFFFFF90) /* UBC  Address*/
#define MMU   (*(volatile struct st_mmu  *)0xFFFFFFE0) /* MMU  Address*/
#define CCR   (*(volatile union  un_ccr  *)0xFFFFFFEC) /* CCR  Address*/
```

Figure 2.22 Header File

2.3.4 Creating the Initialization Part

After reset, the BSC and pointer are set, and control passes to the initialization function.

The initialization function sets interrupt priorities and initializes sections, and then passes control to the start of the user function.

(1) Initialization function <init.c, cntrl.h>

- Sets interrupt priorities
- Flushes the cache
- Turns the cache on
- Initializes sections
- Sets interrupt masks
- Branches to the user function

```
/****************************************************************/
/* file name "cntrl.h" */
/****************************************************************/
#include <machine.h>
#include "7700s.h"
/****************************************************************/
/* control BL ,MD bit */
/****************************************************************/
#define BLoff()     set_cr((get_cr()&0xefffffff))
#define BLon()      set_cr((get_cr()|0x10000000))
#define USRmode()   set_cr((get_cr()|0x40000000))
/****************************************************************/
/* cache control */
/****************************************************************/
#define CacheON()           (CCR.BIT.CE=1)
#define CacheOFF()          (CCR.BIT.CE=0)
#define CacheFLASH()              (CCR.BIT.CF=1)
```

Figure 2.23 Macro Definition Program

```
/****************************************************************/
/* file name "init.c" */
/****************************************************************/
#include <machine.h>
#include "cntrl.h"
void init(void)
{
   set_ip();
   CacheOFF();
   CacheFLASH();
   CacheON();
   BLoff();         /* BLOCK BIT OFF         */

   _INITSCT();      /* section initialize    */
   set_imask(0);    /* interrput priority 0  */

   main();          /* User main() routine   */

   halt();          /* halt()                */

}
```

Figure 2. 24 Initialization Program (1)

(a) Setting interrupt priorities <ipr.c>

IPRA and IPRB are used to set interrupt priorities for each interrupt factor defined in vect7708.h.

```
/***************************************************************/
/* file name "ipr.c" */
/***************************************************************/
#include "7700s.h"
#include "vect7708.h"
void set_ip(void)
{
   INTC.IPRA.WORD=WORD_IPRA;
   INTC.IPRB.WORD=WORD_IPRB;
}
```

Figure 2.25 Program to Set Interrupt Priorities

(b) Section initialization <sect.src, initsct.c>

Initialization of sections allocated to RAM is performed.

The uninitialized data section B is cleared to 0. Initialized data items are copied from section D in ROM to section R in RAM. (initsct.c)

In addition, assembly-language code is necessary to acquire the section start address and size. (sct.src)

```
;*******************************************************************
; file name "sct.src"
;*******************************************************************
          .SECTION  B,DATA,ALIGN=4
          .SECTION  R,DATA,ALIGN=4
          .SECTION  D,DATA,ALIGN=4
;   If other section are existed , Insert here ”.SECTION XXX”,
          .SECTION  C,DATA,ALIGN=4
__B_BGN:  .DATA.L      (STARTOF B)
__B_END:  .DATA.L      (STARTOF B)+(SIZEOF B)
__D_BGN:  .DATA.L      (STARTOF R)
__D_END:  .DATA.L      (STARTOF R)+(SIZEOF R)
__D_ROM:  .DATA.L      (STARTOF D)
          .EXPORT      __B_BGN
          .EXPORT      __B_END
          .EXPORT      __D_BGN
          .EXPORT      __D_END
          .EXPORT      __D_ROM
          .END
```

Figure 2.26 Section Definition Program

```
/**************************************************************/
/* file name "initsct.c"                                      */
/**************************************************************/
extern int *_B_BGN,*_B_END,*_D_BGN,*_D_END,*_D_ROM;

void _INITSCT(void)
{
   register int *p, *q;
   for (p=_B_BGN; p<_B_END; p++){
      *p=0;
   }
   for (p=_D_BGN, q=_D_ROM; p<_D_END; p++, q++){
      *p=*q;
   }
}
```

Figure 2.27 Section Initialization Program

2.3.5 Creating the Main Processing Part and Interrupt Processing Part

Upon creating the functions main(), halt(), and irq15(), the above program can be linked.

```
void main(void)
{
/* user program description */
}
#pragma interrupt(halt, irq15)

void halt(void)
{
   while(1);           /* routine for error processing       */
                       /* here left as an infinite loop      */
}
void irq15(void)
{
                       /* IRQ15 processing program           */
}
```

Figure 2.28 Main Processing Program

2.3.6 Creating a Batch File for the Load Module

Figure 2.29 shows a batch file for creation of an S-type load module (sample.mot).

```
            shcΔ-debugΔ-cpu=sh3Δvect.cΔinit.cΔipr.cΔinitsct.cΔmain.c
                                                                  #Compile C programs
            asmshΔsct.srcΔ-debugΔ-cpu=sh3
            asmshΔintr.srcΔ-debugΔ-cpu=sh3
            asmshΔreset.srcΔ-debugΔ-cpu=sh3
                                                                  #Assemble Assembly programs
            optlnkΔ-nooptΔ-subcommand=lnk.sub
                                                                  #Link using a subcommand file
            rmΔ*.obj
                                                                  #Remove object module files
```

Figure 2.29 Batch File for Load Module Creation

2.3.7 Creating a Linkage Editor Subcommand File

Figure 2.30 shows a subcommand file (filename lnk.sub) for the linkage editor used when creating load modules.

```
  Sdebug
  input     vect, init, ipr, initsct, main, intr, sct, reset
                         ; Specify an input file
  Library   /user/unix/SHCV50/shc3npb.lib
                         ; Specify a standard library
  output    sample.abssample.abs ; Specify output files
  rom D=R                ; Specify ROM support options
  start     P,C,D/10000,R,B,BSTK/04000000
                         ; Specify the start address for each section.
                         ; Do not specify an address for section VECT because
                         ; the section VECT is assigned to absolute address section
                         ; (assigned to address 0).
                         ; Allocate sections P, C, and D to
                         ; the area starting from address H'10000.
                         ; Allocate sections R and B to the area
                         ; starting from address H'04000000.
  form      s            ; Specify s-type format
  list      sample.map; Specify memory map information output
  Exit
```

Figure 2.30 Subcommand File for Linkage Editor

2.4 Debugging using Simulator Debugger

2.4.1 Setting Configuration

Execute the Simulator Debugger using the project that was created in section 2.1.1, Creating the Project for a Simulator Debugger.

Select [Build Configurations…] from the [Option] menu to display the available environments. On the screen shown in figure 2.32, select the environment you use. In this case, select [SimDebug SH-1].

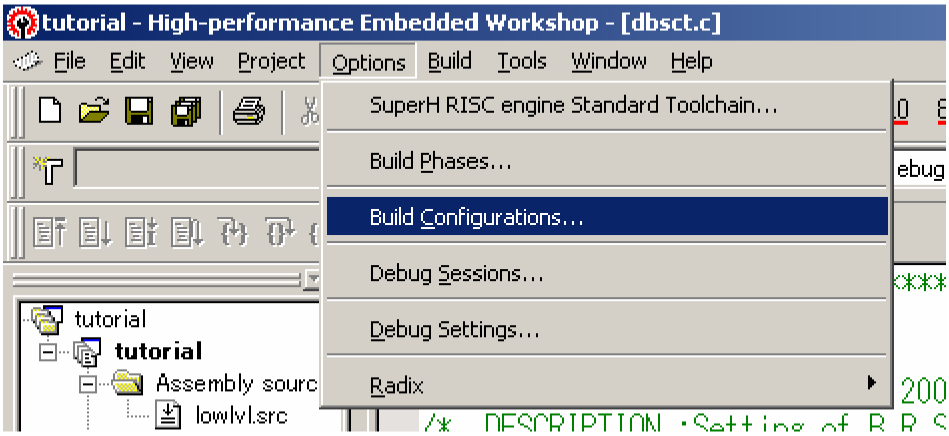
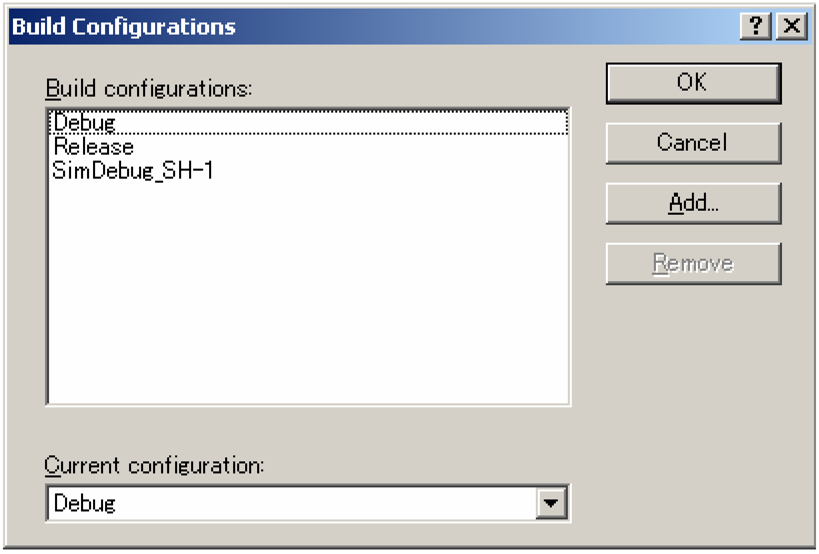


Figure 2.31 Options Menu



Build configurations:

Debug
Release
SimDebug_SH-1

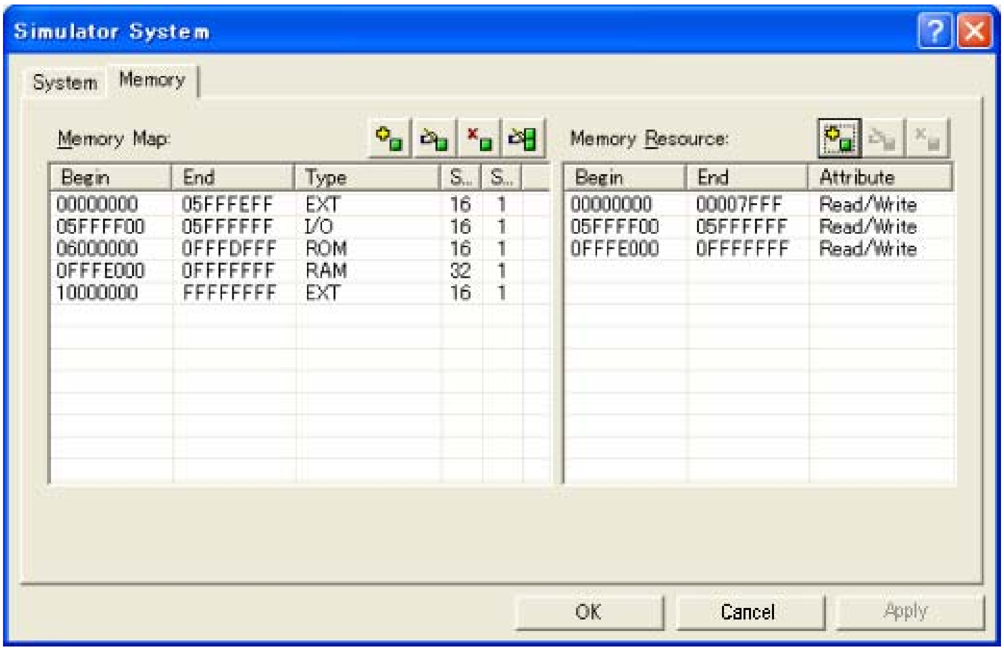
Current configuration: Debug

Figure 2.32 Build Configurations Dialog Box

2.4.2 Allocating Memory Resources

The allocation of the memory resource is necessary to run the application being developed. When using the demonstration project, the memory resource is allocated automatically, so check the setting.

- Select [Simulator->Memory Resource...] from the [Option] menu, and display the allocation of the current memory resource.



Memory Map:

Begin	End	Type	S..	S..
00000000	05FFFEFF	EXT	16	1
05FFFF00	05FFFFFF	I/O	16	1
06000000	0FFFDFFF	ROM	16	1
0FFFE000	0FFFFFFF	RAM	32	1
10000000	FFFFFFFF	EXT	16	1

Memory Resource:

Begin	End	Attribute
00000000	00007FFF	Read/Write
05FFFF00	05FFFFFF	Read/Write
0FFFE000	0FFFFFFF	Read/Write

Figure 2.33 Simulator -> Simulator System Dialog Box

The program area is allocated to the addresses H'00000000 to H'00007FFF. The stack area is allocated to the addresses H'0FFFE000 to H'0FFFFFFF, which can be read from or written to.

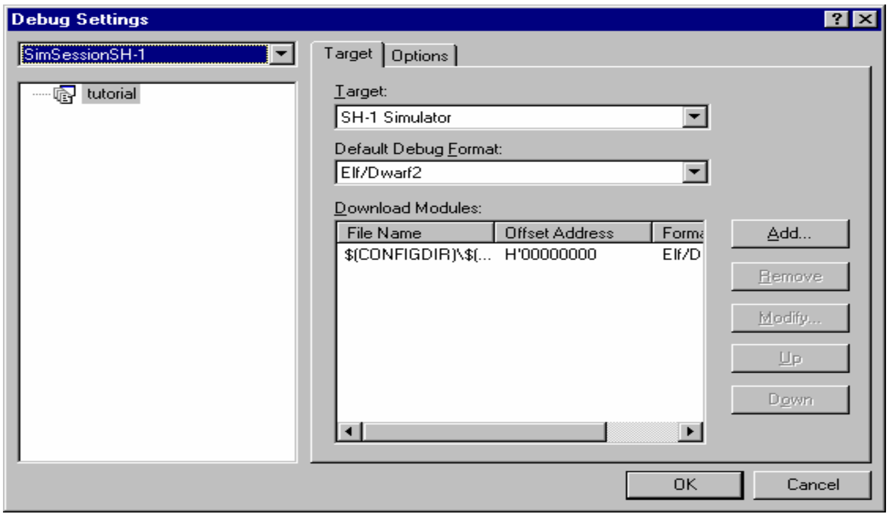
- Close the dialog box by clicking [OK].

The memory resource can also be referred to or modified by using the [Simulator] tab on the [SuperH RISC engine Standard Toolchain] dialog box. Changes made in either of the dialog boxes are reflected.

2.4.3 Downloading a Sample Program

When using the demonstration project, the sample program to be downloaded is automatically set, so check the settings.

- Open the [Debug Setting] dialog box by selecting [Debug Settings...] on the [Option] menu.



SimSessionSH-1

tutorial

Target: SH-1 Simulator

Default Debug Format: Elf/Dwarf2

Download Modules:

File Name	Offset Address	Forma
$(CONFIGDIR)\$(...	H'00000000	Elf/D

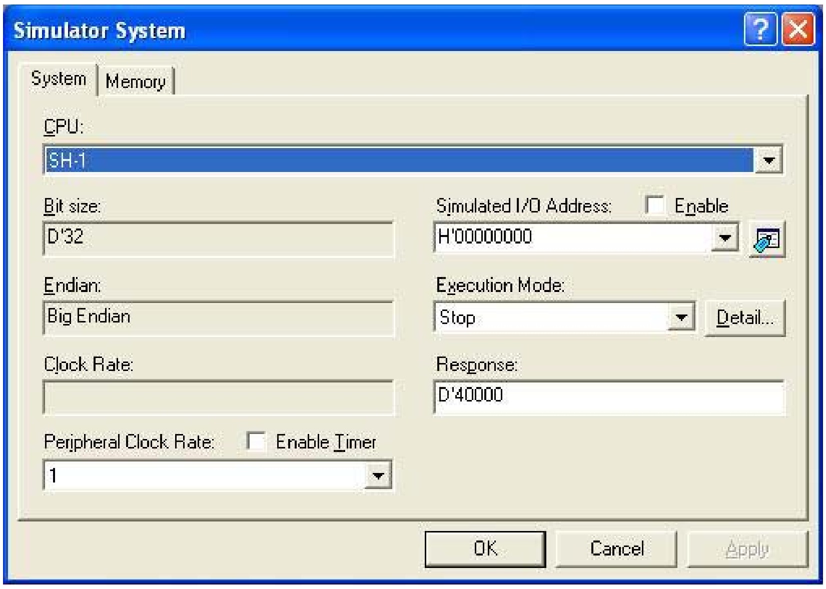
Figure 2.34 Debug Settings Dialog Box

- Files to be downloaded is set in [Download Modules].
- Close the [Debug Settings] dialog box by clicking the [OK] button.
- Download the sample program by selecting [Download Modules->All Download Modules] from the [Debug] menu

2.4.4 Setting Simulated I/O

When the demonstration project is used, the simulated I/O is automatically set, so check the setting.

- Open the [Simulator System] dialog box by selecting [Simulator->System] from the [Option] menu.



CPU: SH-1

Bit size: D'32

Endian: Big Endian

Clock Rate:

Peripheral Clock Rate: 1 □ Enable Timer

Simulated I/O Address: H'00000000 □ Enable

Execution Mode: Stop

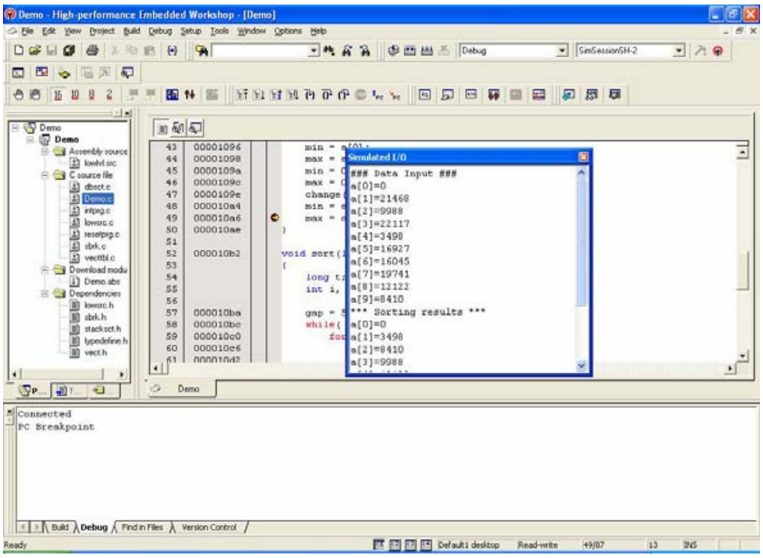
Response: D'40000

Figure 2.35 Simulator System Dialog Box

- Confirm that [Enable] in [System Call Address] is checked.
- Click the [OK] button to enable the Simulated I/O

- Select [CPU->Simulated I/O] from the [View] menu and open the [Simulated I/O] window.

The Simulated I/O will not be enabled if the [Simulated I/O] window is not open.



```
### Data Input ###
a[0]=0
a[1]=21468
a[2]=9988
a[3]=22117
a[4]=3498
a[5]=16927
a[6]=16045
a[7]=19741
a[8]=12122
a[9]=8410
*** Sorting results ***
a[0]=0
a[1]=3498
a[2]=8410
a[3]=9988
```

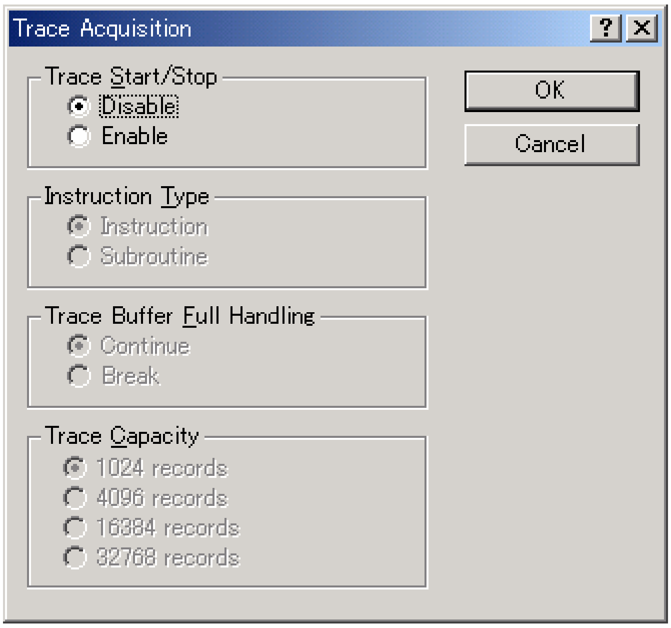
```
Connected
PC Breakpoint
```

Figure 2.36 Simulated I/O Window

2.4.5 Setting Trace Information Acquisition Conditions

- Select [Code->Trace] from the [View] menu and open the [Trace] window. Open the pop-up menu by right clicking the mouse on the [Trace] window, and select [Acquisition...] from the popup menu.

The [Trace Acquisition] dialog box below will be displayed.


Trace Acquisition

Trace Start/Stop
● Disable
○ Enable

Instruction Type
● Instruction
○ Subroutine

Trace Buffer Full Handling
● Continue
○ Break

Trace Capacity
● 1024 records
○ 4096 records
○ 16384 records
○ 32768 records

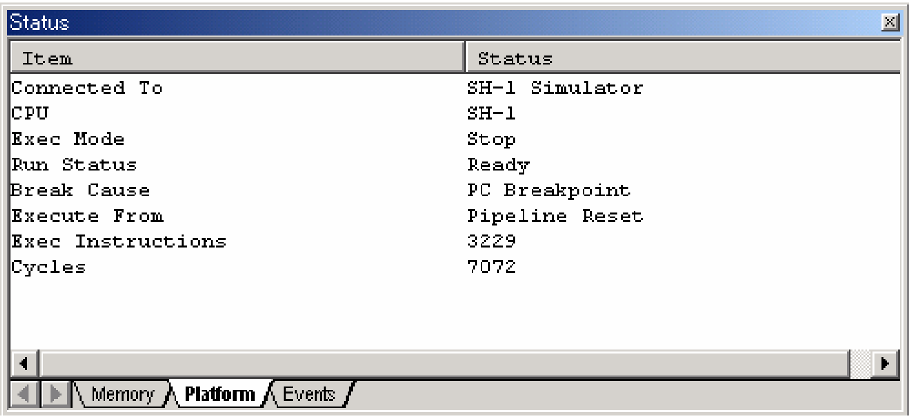
Figure 2.37 Trace Acquisition Dialog Box

- Set [Trace start/Stop] to [Enable] in the [Trace Acquisition] dialog box, and click the [OK] button to enable the acquisition of the trace information.

2.4.6 Status Window

The termination cause can be displayed in the [Status] window.

- Select [CPU->Status] from the [View] menu to open the [Status] window, and select the [Platform] sheet in the [Status] window.



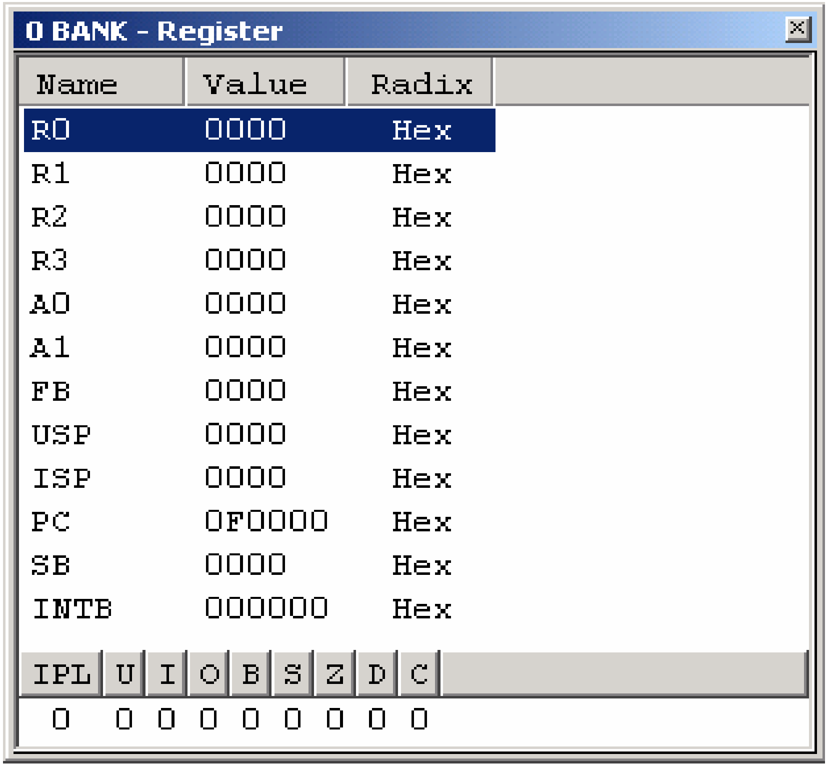
Item	Status
Connected To	SH-1 Simulator
CPU	SH-1
Exec Mode	Stop
Run Status	Ready
Break Cause	PC Breakpoint
Execute From	Pipeline Reset
Exec Instructions	3229
Cycles	7072

Figure 2.38 View->CPU->Staus Window

2.4.7 Registers Window

Register values can be checked in the [Register] window.

- Select [CPU->Registers] from the [View] menu.



Name	Value	Radix
R0	0000	Hex
R1	0000	Hex
R2	0000	Hex
R3	0000	Hex
A0	0000	Hex
A1	0000	Hex
FB	0000	Hex
USP	0000	Hex
ISP	0000	Hex
PC	0F0000	Hex
SB	0000	Hex
INTB	000000	Hex

IPL	U	I	O	B	S	Z	D	C
0	0	0	0	0	0	0	0	0

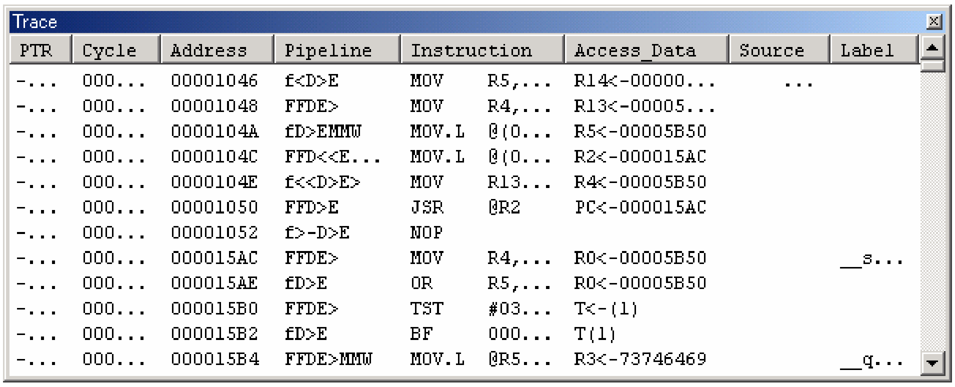
Figure 2.39 View ->CPU-> Register Window

2.4.8 Trace

(1) Trace Buffer

The trace buffer can be used to clarify the history of instruction execution.

- Select [Code->Trace] from the [View] menu and open the [Trace] window. Scroll up to the very top of the window.



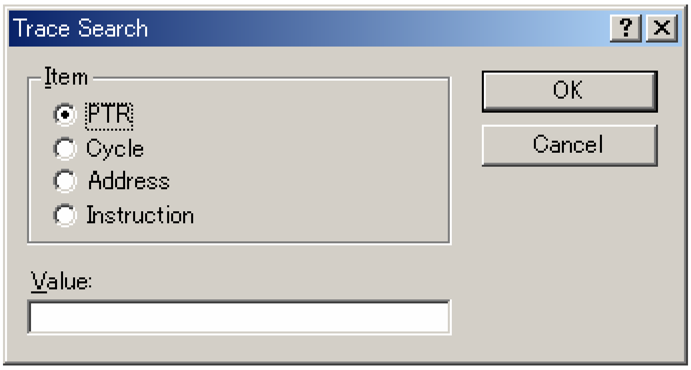
Trace

PTR	Cycle	Address	Pipeline	Instruction		Access_Data	Source	Label
-...	000...	00001046	f<D>E	MOV	R5,...	R14<-00000...	...	
-...	000...	00001048	FFDE>	MOV	R4,...	R13<-00005...		
-...	000...	0000104A	fD>EMMW	MOV.L	@(0...	R5<-00005B50		
-...	000...	0000104C	FFD<<E...	MOV.L	@(0...	R2<-000015AC		
-...	000...	0000104E	f<<D>E>	MOV	R13...	R4<-00005B50		
-...	000...	00001050	FFD>E	JSR	@R2	PC<-000015AC		
-...	000...	00001052	f>-D>E	NOP				
-...	000...	000015AC	FFDE>	MOV	R4,...	R0<-00005B50		__s...
-...	000...	000015AE	fD>E	OR	R5,...	R0<-00005B50		
-...	000...	000015B0	FFDE>	TST	#03...	T<-(1)		
-...	000...	000015B2	fD>E	BF	000...	T(1)		
-...	000...	000015B4	FFDE>MMW	MOV.L	@R5...	R3<-73746469		__q...

Figure 2.40 Trace Window (Trace Information Display)

(2) Trace Search

Click the right mouse button on the [Trace] window to launch the pop-up menu, and select [Find...] to open the [Trace Search] dialog box.



Trace Search

Item

● PTR
○ Cycle
○ Address
○ Instruction

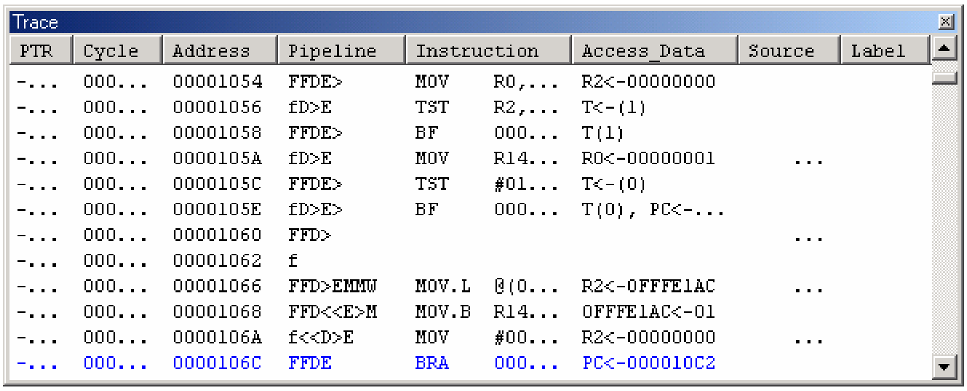
Value:

OK

Cancel

Figure 2.41 Trace Search Dialog Box

Setting the item to be searched to [Item] and the contents to be searched to [Value] and clicking the [OK] button begins the trace search. When the searched item is found, the first line is highlighted. To continue searching the same contents [Value], click the right mouse button in the [Trace] window to display the pop-up menu, and select [Find Next] from the pop-up menu. The next searched line is highlighted.



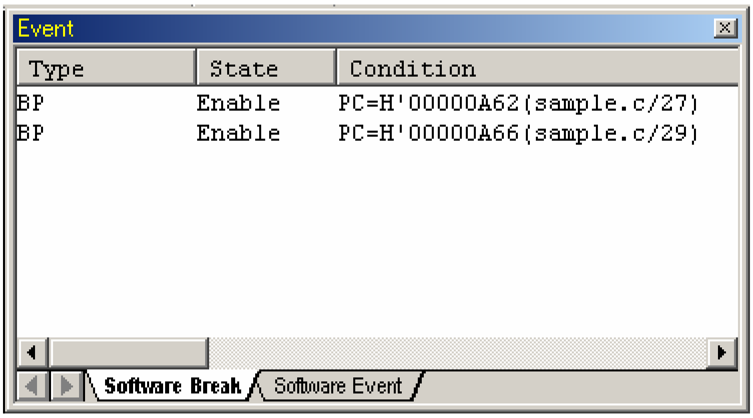
PTR	Cycle	Address	Pipeline	Instruction	Access_Data	Source	Label
-...	000...	00001054	FFDE>	MOV R0,...	R2<-00000000		
-...	000...	00001056	fD>E	TST R2,...	T<-(1)		
-...	000...	00001058	FFDE>	BF 000...	T(1)		
-...	000...	0000105A	fD>E	MOV R14...	R0<-00000001	...	
-...	000...	0000105C	FFDE>	TST #01...	T<-(0)		
-...	000...	0000105E	fD>E>	BF 000...	T(0), PC<-...		
-...	000...	00001060	FFD>			...	
-...	000...	00001062	f				
-...	000...	00001066	FFD>EMMW	MOV.L @(0...	R2<-0FFFE1AC	...	
-...	000...	00001068	FFD<<E>M	MOV.B R14...	0FFFE1AC<-01		
-...	000...	0000106A	f<<D>E	MOV #00...	R2<-00000000	...	
-...	000...	0000106C	FFDE	BRA 000...	PC<-000010C2		

Figure 2.42 Trace Window

2.4.9 Displaying Breakpoints

A list of all the breakpoints that are set in the program can be checked in the [Eventpoint] window.

- Select [Code->Eventpoint] from the [View] menu.



Type	State	Condition
BP	Enable	PC=H'00000A62(sample.c/27)
BP	Enable	PC=H'00000A66(sample.c/29)

Figure 2.43 Eventpoint Window

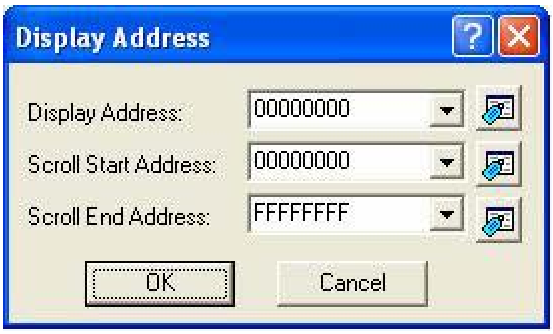
A breakpoint can be set, a new breakpoint can be defined, and a breakpoint can be deleted using the [Eventpoint] window.

- Close the [Eventpoint] window.

2.4.10 Displaying Memory Contents

The contents of memory block can be displayed on a Memory window. For example, the procedure for displaying the memory for the main column in byte size is shown as below.

Select [CPU->Memory…] from the [View] menu to enter memory area start address in the [Begin] field and end address in the [End] field.



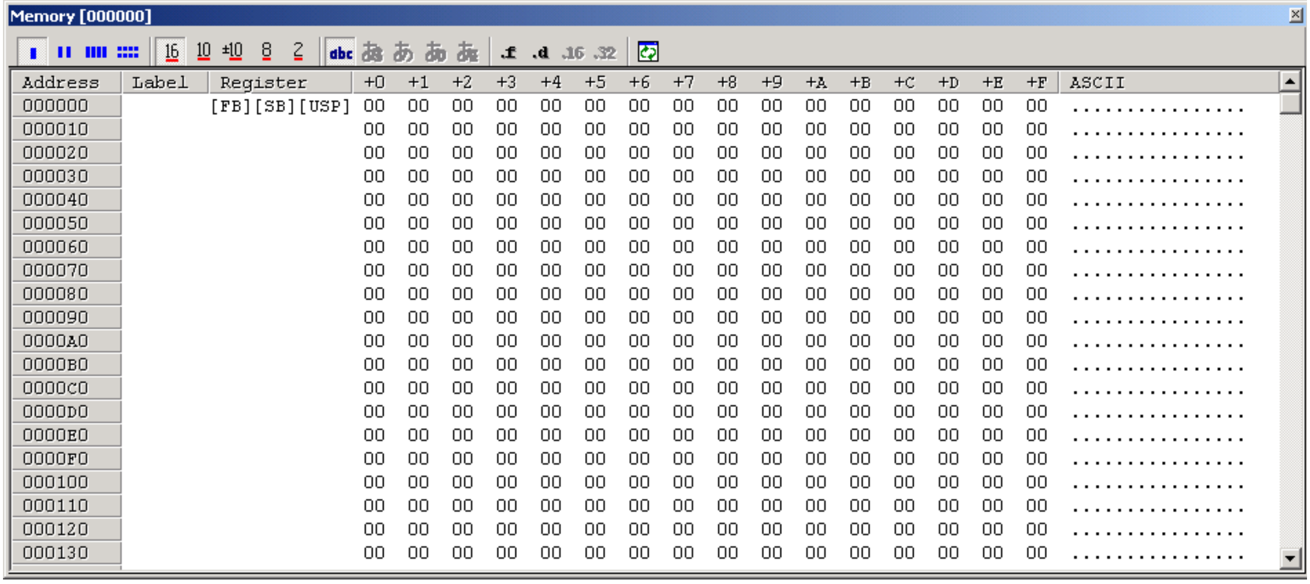
Display Address: 00000000

Scroll Start Address: 00000000

Scroll End Address: FFFFFFFF

Figure 2.44 Set Address Dialog Box

Click on the [OK] button to open the Memory window which shows the specified memory area.



```
Address  Label  Register        +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F  ASCII
000000          [FB][SB][USP]   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000010                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000020                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000030                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000040                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000050                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000060                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000070                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000080                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000090                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000A0                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000B0                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000C0                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000D0                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000E0                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000F0                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000100                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000110                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000120                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000130                          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figure 2.45 Memory Window

2.5 Standard I/O and File I/O Processing in the Simulator/Debugger

The simulator/debugger allows the user to perform the standard I/O and file I/O from the programe to be debugged. When the I/O processing is executed, the Simulation I/O window must be open. The supported I/O processing is as follows:

Table 2.3 I/O Functions

No.	Function Code	Function name	Description
1	H'21	GETC	Inputs one byte from the standard input device.
2	H'22	PUTC	Outputs one byte to the standard output device.
3	H'23	GETS	Inputs one line from the standard input device.
4	H'24	PUTS	Outputs one line to the standard output device.
5	H'25	FOPEN	Opens a file.
6	H'06	FCLOSE	Closes a file.
7	H'27	FGETC	Inputs one byte from a file.
8	H'28	FPUTC	Outputs one byte to a file.
9	H'29	FGETS	Inputs one line from a file.
10	H'2A	FPUTS	Outputs one line to a file.
11	H'0B	FEOF	Checks for end of file.
12	H'0C	FSEEK	Moves the file pointer
13	H'0D	FTELL	Returns the current position of the file pointer.

To perform I/O processing, first specify the location for I/O in the [System Call Address] in the Simulator System dialog box, check the [Enable], and then execute the program to be debugged.

When detecting a subroutine call instruction (BSR, JSR, or BSRF), that is, a system call to the specialized address during user program execution, the simulator/debugger performs I/O processing by using the R0 and R1 values as the parameters.

Therefore, before issuing a system call, set as follows in the user program:

- Set the function code (table 2.3) to the R0 register

MSB 1 byte	1 byte		LSB
H'01	Function Code	…	…

- Set the parameter block address to the R1 register

(For the parameter block, refer to each function description)

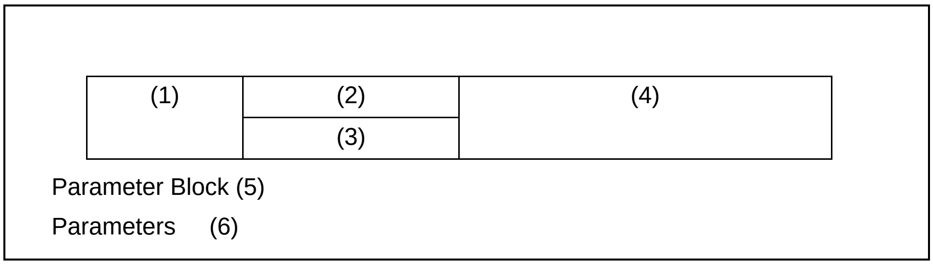
MSB LSB

Parameter block address

- Reserve the parameter block and I/O buffer areas

In case of accessing each parameter of the parameter block, after the I/O processing in the parameter size, the simulator/debugger resumes simulation from the instruction that follows the system call instruction.

Note: If the JSR, BSR, or BSRF instruction is executed as system call, an instruction following the JSR, BSR, or BSRF is as an ordinary instruction, not a slot instruction. Hence, any instruction whose result will differ depending on the slot instruction or ordinary instruction should not be used following the JSR, BSR, or BSRF.



(1)	(2)	(4)
	(3)	

Parameter Block (5)

Parameters (6)

Figure 2.46 Description Format of the I/O Function

The contents of the items are as follows:

(1) Number corresponding to table 2.3

(2) Function name

(3) Function Code

(4) I/O overview

(5) I/O parameter block

(6) I/O parameters

1	GETC	Inputs one byte from the standard input device
	H'21	

Parameter Block One byte One byte

+0 +2	Input buffer start address

Parameter

- Input buffer start address (input)

Start address of the buffer to which the input data is written to

2	PUTC	Outputs one byte to the standard output device.
	H'22	

Parameter Block One byte One byte

+0 +2	Output buffer start address

Parameter • Output buffer start address (input)

Start address of the buffer in which the output data is stored

3	GETS	Inputs one line from the standard input device.
	H'23	

Parameter Block

	One byte	One byte
+0	Input buffer start address	
+2		

Parameter • Input buffer start address (input)

Start address of the buffer to which the input data is written to

4	PUTS	Outputs one line to the standard output device.
	H'24	

Parameter Block

	One byte	One byte
+0	Output buffer start address	
+2		

Parameter • Output buffer start address (input)

Start address of the buffer in which the output data is stored

5	FOPEN	Opens a file.
	H'25	

The FOPEN opens a file and returns the file number. After this processing, the returned file number must be used to input, output, or close files. A maximum of 256 files can be open at the same time.

Parameter Block

	One byte	One byte
+0	Return Value	File number
+2	Open mode	Unused
+4	Start address of file name	
+6		

Parameter

- Return value (output)

  0: Normal completion

  -1: Error

- File number (output)

  The number to be used in all file accesses after opening

  Open mode (input)

  H'00 "r"

  H'01 "w"

  H'02 "a"

  H'03 "r+"

  H'04 "w+"

H'05 "a+"
H'10 "rb"
H'11 "wb"
H'12 "ab"
H'13 "r+b"
H'14 "w+b"
H'15 "a+b"
These modes are interpreted as follows:
"r": Open for reading.
"w": Open an empty file for writing
"a": Open for appending (write starting at the end of the file).
"r+": Open for reading and writing.
"w+": Open an empty file for reading and writing.
"a+": Open for reading and appending.
"b": Open in binary mode.

- Start address of file name (input)
  The start address of the area for storing the file name

6	FCLOSE H'06	Closes a file.

Parameter Block

	One byte	One byte
+0	Return Value	File number

Parameter
- Return value (output)
  0: Normal completion
  -1: Error
- File number (input)
  The number returned when the file was opened

7	FGETC H'27	Inputs one byte from a file

Parameter Block

	One byte	One byte
+0	Return Value	File number
+2	Unused	
+4	Start address of input buffer	
+6		

Parameter
- Return value (output)
  0: Normal completion
  -1: Error
- File number (input)
  The number returned when the file was opened
- Start address of input buffer (input)
  The start address of the buffer for storing input data

8	FPUTC H'28	Outputs one byte to a file

Parameter Block

	One byte	One byte
+0	Return Value	File number
+2	Unused	
+4	Start address of output buffer	
+6		

Parameter

- Return value (output)

  0: Normal completion

  -1: Error
- File number (input)

  The number returned when the file was opened
- Start address of output buffer (input)

  The start address of the buffer used for storing the output data

9	FGETS	Reads character string data from a file
	H'29	

Parameter Block

	One byte	One byte
+0	Return Value	File number
+2	Buffer size	
+4	Start address of input buffer	
+6		

Parameter

- Return value (output)

  0: Normal completion

  -1: EOF detected
- File number (input)

  The number returned when the file was opened
- Buffer size (input)

  The size of the area for storing the read data

  (A maximum of 256 bytes can be stored)
- Start address of input buffer (input)

  The start address of the buffer for storing input data

10	FPUTS	Writes character string data to a file
	H'2A	

Parameter Block

	One byte	One byte
+0	Return Value	File number
+2	Unused	
+4	Start address of output buffer	
+6		

Parameter

- Return value (output)

  0: Normal completion

-1: Error

- File number (input)

  The number returned when the file was opened

- Start address of output buffer (input)

  The start address of the buffer used for storing the output data

11	FEOF	Checks for end of file
	H'0B	

Parameter Block

	One byte	One byte
+0	Return Value	File number

Parameter

- Return value (output)

  0: File pointer is not at EOF

  -1: EOF detected

- File number (input)

  The number returned when the file was opened

12	FSEEK	Moves the file pointer to the specified position
	H'0C	

Parameter Block

	One byte	One byte
+0	Return Value	File number
+2	Direction	Unused
+4	Offset	
+6		

Parameter

- Return value (output)

  0: Normal completion

  -1: Error

- File number (input)

  The number returned when the file was opened

- Direction (input)

  0: The offset specifies the position as a byte count from the start of the file

  1: The offset specifies the position as a byte count from the current file pointer

  2: The offset specifies the position as a byte count from the end of the file

- Offset (input)

  The byte count from the location specified by the direction parameter

13	FTELL	Returns the current position of the file pointer.
	H'0D	

Parameter Block

	One byte	One byte
+0	Return Value	File number
+2	Unused	
+4	Offset	
+6		

Parameter
- Return value (output)
  0: Normal completion
  -1: Error
- File number (input)
  The number returned when the file was opened
- Offset (output)
  The current position of the file pointer
  (A byte count from the start of the file)

The following shows an example for inputting and outputting one character as a standard input (from a keyboard).

```
;------------------------------------------------------------------------
;
;   FILE        :lowlvl.src
;   DATE        :Tue, Mar 05, 2002
;   DESCRIPTION :Program of Low level
;   CPU TYPE    :
;
;   This file is generated by Renesas Project Generator (Ver.3.0).
;
;------------------------------------------------------------------------

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;                            lowlvl.src
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;            SH-series simulator debugger interface routine
;                         -Input/output one character-
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
     .EXPORT     _charput
     .EXPORT     _charget
SIM_IO:   .EQU      H'0000        ;Specifies TRAP_ADDRESS

     .SECTION    P, CODE, ALIGN=4

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;  _charput: One character output
;          C program interface: charput(char)
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

_charput:
```

```
          MOV.L       O_PAR,R0          ; Sets output buffer address to R0
          MOV.B       R4,@R0            ; Sets output charcter to buffer
          MOV.L       #O_PAR,R1         ; Sets parameter block address to R1
          MOV.L       #H'01220000,R0    ; Specifies function code (PUTC)
          MOV.W       #SIM_IO,R2        ; Sets system call address to R2
          JSR         @R2
     NOP
     RTS
     NOP

          .ALIGN      4
O_PAR:                                  ; Parameter block
          .DATA.L     OUT_BUF

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; _charget: One character input
;      C program interface: char charget(void)
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

          .ALIGN      4
_charget:
          MOV.L       #I_PAR,R1         ; Sets parameter block address to R1
          MOV.L       #H'01210000,R0    ; Specifies function code (GETC)
          MOV.W       #SIM_IO,R2        ; Sets system call address to R2
          JSR         @R2
     NOP
          MOV.L       I_PAR,R0          ; Sets input buffer address to R0
          MOV.B       @R0,R0            ; Returns input data
     RTS
     NOP

          .ALIGN      4
I_PAR:                                  ; Parameter block
          .DATA.L     IN_BUF

;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;            I/O buffer definition
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

     .SECTION    B,DATA,ALIGN=4

OUT_BUF:
          .RES.L      1              ; Output buffer
IN_BUF:
          .RES.L      1              ; Input buffer

     .END
```

Section 3 Compiler

3.1 Interrupt Functions

3.1.1 Definitions of Interrupt Functions (No Options)

Description:

Preprocessor directive (#pragma) can be used to create interrupt functions in the C language. A function declared using "#pragma interrupt" saves/restores all registers (except for the global base register GBR and vector base register VBR) to be used within the function before and after function processing. For this reason, interrupted functions do not need to make provisions to deal with interrupts.

- Format:

#pragma interrupt (<function name>,[<function name>...])

Example of use:

The interrupt function handler1 is declared. This function takes over the stack from the interrupted function and uses it, and after the completion of processing, returns with an RTE instruction.

<Case where GBR, VBR are not saved/restored>

C language code

```
#pragma interrupt(handler1)    /* Declares interrupt function */
void handler1(void)
   {
      :                        /* Interrupt function processing*/
      :
   }
```

Expanded into assembly language code

```
                 .EXPORT    _handler1
                 .SECTION   P,CODE,ALIGN=4
_handler1:                     ; function: handler1
                     :         ; Saves work registers
                     :         ; Interrupt function processing
                     :         ; Restores work registers
                 RTE
                 NOP
                 .END
```

<When storing or restoring the GBR and VBR>

C language code

```
#pragma interrupt(handler1)
void handler1(void)
{
    void** save_vbr;                /* Defines the VBR storage area   */
    void* save_gbr;                 /* Defines the GBR storage area   */
    save_vbr = get_vbr();           /* Saves VBR                      */
    save_gbr = get_gbr();           /* Saves GBR                      */
        :                           /* Interrupt function processing  */
        :
    set_vbr(save_vbr);              /* Restores VBR                   */
    set_gbr(save_gbr);              /* Restores GBR                   */
}
```

Expanded into assembly language code

```
         .EXPORT     handler1
         .SECTION    P,CODE,ALIGN=4
handler1:                       ; function: handler1
                                ; frame size=8
          MOV.L      R5,@-R15
          STC        GBR,R5     ; Saves GBR
          MOV.L      R4,@-R15
          STC        VBR,R4     ; Saves VBR
             :                  ; Saves work registers
             :                  ; Interrupt function processing
             :                  ; Restores work registers
          LDC        R4,VBR     ; Restores VBR
          LDC        R5,GBR     ; Restores GBR
          MOV.L      @R15+,R4
          MOV.L      @R15+,R4
          RTE
          NOP
          .END
```

Important Information:

(1) Only the void data type can be returned by an interrupt function.

Examples:

```
#pragma interrupt(f1, f2)       /* Declares interrupt function     */
void   f1(void){...}            /* Defines interrupt function f1   */
int    f2(void){...}            /* Defines interrupt function f2   */
```

The definition of the interrupt function f1 is correct, but the definition of the interrupt function f2 results in an error.

(2) The only memory class specifier that can be specified in the definition of an interrupt function is extern. Even if static is specified, it is treated as extern.

(3) function declared as an interrupt function cannot be called as an ordinary function. If a function declared as an interrupt function is called as an ordinary function, runtime operation is not guaranteed.

Examples:

- test1.c file contents

```
#pragma interrupt(f1)       /* Declares interrupt function       */
void f1(void){…}            /* Declares interrupt function f1    */
int  f2(){f1();}
```

- test2.c file contents

```
f3(){ f1(); }
```

In the file test1.c, an error occurs at function f2. In the file test2.c, an error does not occur at the function f3, but the function f1 is interpreted as extern int f1(), and runtime operation becomes unstable.

(4) In the event of an interrupt, the operation of SH-3, SH3-DSP, SH-4A and SH4AL-DSP differs from that of SH-1, SH-2 SH-2E, SH-2A, SH2A-FPU and SH2-DSP, and an interrupt handler is necessary. An example of an interrupt handler is shown below.

Example of an Interrupt Handler for SH-3

```
;*******************************************************************
;   FILE   :vhandler.src
;*******************************************************************
       .include  "env.inc"
       .include  "vect.inc"

IMASKclr:     .equ   H'FFFFFF0F
RBBLclr:      .equ   H'CFFFFFFF
MDRBBLset:    .equ   H'70000000
       .import       _RESET_Vectors
       .import       _INT_Vectors
       .import       _INT_MASK
;*******************************************************************
;*     macro definition
;*******************************************************************
                .macro PUSH_EXP_BASE_REG
        stc.l   ssr,@-r15                 ; save ssr
        stc.l   spc,@-r15                 ; save spc
        sts.l   pr,@-r15                  ; save context registers
        stc.l   r7_bank,@-r15
        stc.l   r6_bank,@-r15
        stc.l   r5_bank,@-r15
        stc.l   r4_bank,@-r15
        stc.l   r3_bank,@-r15
        stc.l   r2_bank,@-r15
        stc.l   r1_bank,@-r15
        stc.l   r0_bank,@-r15
                .endm
;
                .macro POP_EXP_BASE_REG
        ldc.l   @r15+,r0_bank           ; recover registers
        ldc.l   @r15+,r1_bank
        ldc.l   @r15+,r2_bank
        ldc.l   @r15+,r3_bank
        ldc.l   @r15+,r4_bank
        ldc.l   @r15+,r5_bank
        ldc.l   @r15+,r6_bank
        ldc.l   @r15+,r7_bank
        lds.l   @r15+,pr
        ldc.l   @r15+,spc
        ldc.l   @r15+,ssr
                .endm
;********************************************************************
;      reset
;********************************************************************
       .section  RSTHandler,code
_ResetHandler:
             mov.l  #EXPEVT,r0
             mov.l  @r0,r0
             shlr2  r0
             shlr   r0
             mov.l  #_RESET_Vectors,r1
             add    r1,r0
             mov.l  @r0,r0
             jmp    @r0
             nop
;********************************************************************
;      exceptional interrupt
;********************************************************************
       .section  INTHandler,code
       .export   INTHandlerPRG
```

```
INTHandlerPRG:
_ExpHandler:
             PUSH_EXP_BASE_REG
;
             mov.l  #EXPEVT,r0                  ; set event address
             mov.l  @r0,r1                      ; set exception code
             mov.l  #_INT_Vectors,r0            ; set vector table address
             add    #-(h'40),r1                 ; exception code - h'40
             shlr2  r1
             shlr   r1
             mov.l  @(r0,r1),r3                 ; set interrupt function addr
;
             mov.l  #_INT_MASK,r0               ; interrupt mask table addr
             shlr2  r1
             mov.b  @(r0,r1),r1                 ; interrupt mask
             extu.b r1,r1
;
             stc    sr,r0                       ; save sr
             mov.l  #(RBBLclr&IMASKclr),r2      ; RB,BL,mask clear data
             and    r2,r0                       ; clear mask data
             or     r1,r0                       ; set interrupt mask
             ldc    r0,ssr                      ; set current status
;
             ldc.l  r3,spc
             mov.l  #__int_term,r0              ; set interrupt terminate
             lds    r0,pr
;
             rte
             nop
;
             .pool
;
;*******************************************************************
;      Interrupt terminate
;*******************************************************************
             .align 4
__int_term:
             mov.l   #MDRBBLset,r0              ; set MD,BL,RB
             ldc.l  r0,sr                       ;
             POP_EXP_BASE_REG
             rte                                ; return
             nop
;
             .pool
;
```

```
;*********************************************************************
;      TLB miss interrupt
;*********************************************************************
       .org   H'300
_TLBmissHandler:
              PUSH_EXP_BASE_REG
;
              mov.l  #EXPEVT,r0                ; set event address
              mov.l  @r0,r1                    ; set exception code
              mov.l  #_INT_Vectors,r0          ; set vector table address
              add    #-(h'40),r1               ; exception code - h'40
              shlr2  r1
              shlr   r1
              mov.l  @(r0,r1),r3               ; set interrupt function addr
;
              mov.l  #_INT_MASK,r0             ; interrupt mask table addr
              shlr2  r1
              mov.b  @(r0,r1),r1               ; interrupt mask
              extu.b r1,r1
;
              stc    sr,r0                     ; save sr
              mov.l  #(RBBLclr&IMASKclr),r2    ; RB,BL,mask clear data
              and    r2,r0                     ; clear mask data
              or     r1,r0                     ; set interrupt mask
              ldc    r0,ssr                    ; set current status
;
              ldc.l  r3,spc
              mov.l  #__int_term,r0            ; set interrupt terminate
              lds    r0,pr
;
              rte
              nop
;
              .pool
;
;*********************************************************************
;      IRQ
;*********************************************************************
       .org   H'500
_IRQHandler:
              PUSH_EXP_BASE_REG
;
              mov.l  #INTEVT,r0                ; set event address
              mov.l  @r0,r1                    ; set exception code
```

```
             mov.l  #_INT_Vectors,r0          ; set vector table address
             add    #-(h'40),r1               ; exception code - h'40
             shlr2  r1
             shlr   r1
             mov.l  @(r0,r1),r3               ; set interrupt function addr
;
             mov.l  #_INT_MASK,r0             ; interrupt mask table addr
             shlr2  r1
             mov.b  @(r0,r1),r1               ; interrupt mask
             extu.b r1,r1
;
             stc    sr,r0                     ; save sr
             mov.l  #(RBBLclr&IMASKclr),r2    ; RB,BL,mask clear data
             and    r2,r0                     ; clear mask data
             or     r1,r0                     ; set interrupt mask
             ldc    r0,ssr                    ; set current status
;
             ldc.l  r3,spc
             mov.l  #__int_term,r0            ; set interrupt terminate
             lds    r0,pr
;
             rte
             nop
;
             .pool
             .end
```

Note: In the table part of the SH-3 interrupt handler example, places for which there are no corresponding addresses should be left blank.
In this case, RTE instructions are used in order to jump to the vector function. In addition, on returning from the vector function, control is passed to the terminate routine, therefore it is necessary to return the vector function by executing RTS.
For this reason, when defining the vector function, do not use ‘#pragma interrupt’
The include files “env.inc" and "vect.inc" are automatically generated by HEW when the SH3 project is created.
In the PUSH_EXP_BASE_REG and POP_EXP_BASE_REG macros described in the list above, only the R0-R7 bank registers are saved by the “stc.l rn_bank, @-R15” instructions, and restored by the “ldc.l @+R15, rn_bank” instructions.
General registers which can be accessed by the “MOV” instructions aren’t saved nor restored.
In case of using SH-3,SH3-DSP,SH-4,SH-4A or SH4AL-DSP, when an interrupt is accepted, the RB bit in the SR register is set to 1. So if you use these macros right before or after the interrupt handler, only the R0_BANK0 to R7_BANK0 registers are saved and the R0_BANK1 to R7_BANK1 registers are not saved.
So even if you use these macros at the head of the interrupt handler, as long as the program runs with RB remains set to 1, R0_BANK1 to R7_BANK1 registers were not saved and the value of these registers are destroyed.
In case of using these macros, you must run the program before interruption with RB=0, or modify these macros to save/restore R0_BANK1 to R7_BANK1 registers.

3.1.2 Definitions of Interrupt Functions (with Options)

Description:

Options available in definitions of interrupt functions are "specify stack switching" and "specify trap instruction return", and "specify register bank".
By using "specify stack switching," when an external interrupt occurs the stack pointer is switched to the specified address, and the stack is used to execute the interrupt function. On return, the stack pointer is returned to its position at the time of the interrupt (figure 3.1). In using this option, a sufficient margin must be secured for the stack of the interrupted function to be used by the interrupt function.
By using the "specify trap instruction return" option, the TRAPA instruction is used for returns. If this option is not specified, the RTE instruction is used for returns.
SH-2A and SH2A-FPU have build-in register banks for quickly saving and restoring registers during interrupt processing. By using the "specify register bank" option, the compiler generates the code for saving and restoring registers, assuming that the register bank is available.
In particular, the bank target registers (R0 to R14, GBR, MACH, MACL, and PR) are automatically saved when an interrupt exception occurs. This suppresses the generation of the save code in the interrupt function.
The RESBANK instruction is used for restoring the bank target registers.

- Format:

#pragma interrupt (<function name>[(<Interrupt specification>)][,<function name> [(<Interrupt specification>)]...])

Table 3.1 Interrupt Specifications

No.	Item	Format	Option	Specifications
1	Stack Switching specification	sp=	{ <variable> | &<variable> | <constant> | <variable>+< constant> | &<variable>+ <constant> }	Specifies an address of a new stack using variable or constant <variable>: Variable (pointer type) &<variable>: Variable address (object type) <constant>: Constant
2	TRAP instruction Return specification	tn=	<constant>	Specifies the end with the TRAPA instruction. <constant>: Constant value (TRAP vector number)
3	Register bank specification	resbank	None	Suppresses the output of the register save code for the following registers: R0 to R14, GBR, MACH, MACL, PR If "tn" is not specified, the RESBANK instruction is generated before the RTE instruction.

Example of use:

Example 1

The interrupt function handler2 is declared. This function uses the array STK as a stack (figure 3.1), and on completion of processing returns control via the TRAPA#63 instruction.

C language code

```
extern  int  STK[100];

int  *ptr  = STK + 100;

#pragma interrupt(handler2(sp=ptr, tn=63))

                                                /* Declares interrupt function            */

void  handler2(void)

{

         :                                      /* Describes interrupt function processing  */

         :


}
```

Expanded into assembly language code

```
          .IMPORT     _STK

          .EXPORT     _ptr

          .EXPORT     _handler2

          .SECTION    P,CODE,ALIGN=4

handler2:                                       ; function: handler2

                                                ; frame size=4

          MOV.L       R0,@-R15

          MOV.L       L217,R0                   ; _ptr

          MOV.L       @R0, R0

          MOV.L       R15,@-R0

          MOV         R0,R15

           :                                    ; Saves the work registers

           :                                    ; Interrupt function processing

           :                                    ; Restores the work registers

          MOV.L       @R15+,R15

          MOV.L       @R15+,R0

          TRAPA       #63

L217 :

          .DATA.L     _ptr

          .SECTION    D,DATA,ALIGN=4

_ptr:                                           ; static: ptr

          .DATA.L     H'00000190+ STK

          .END
```

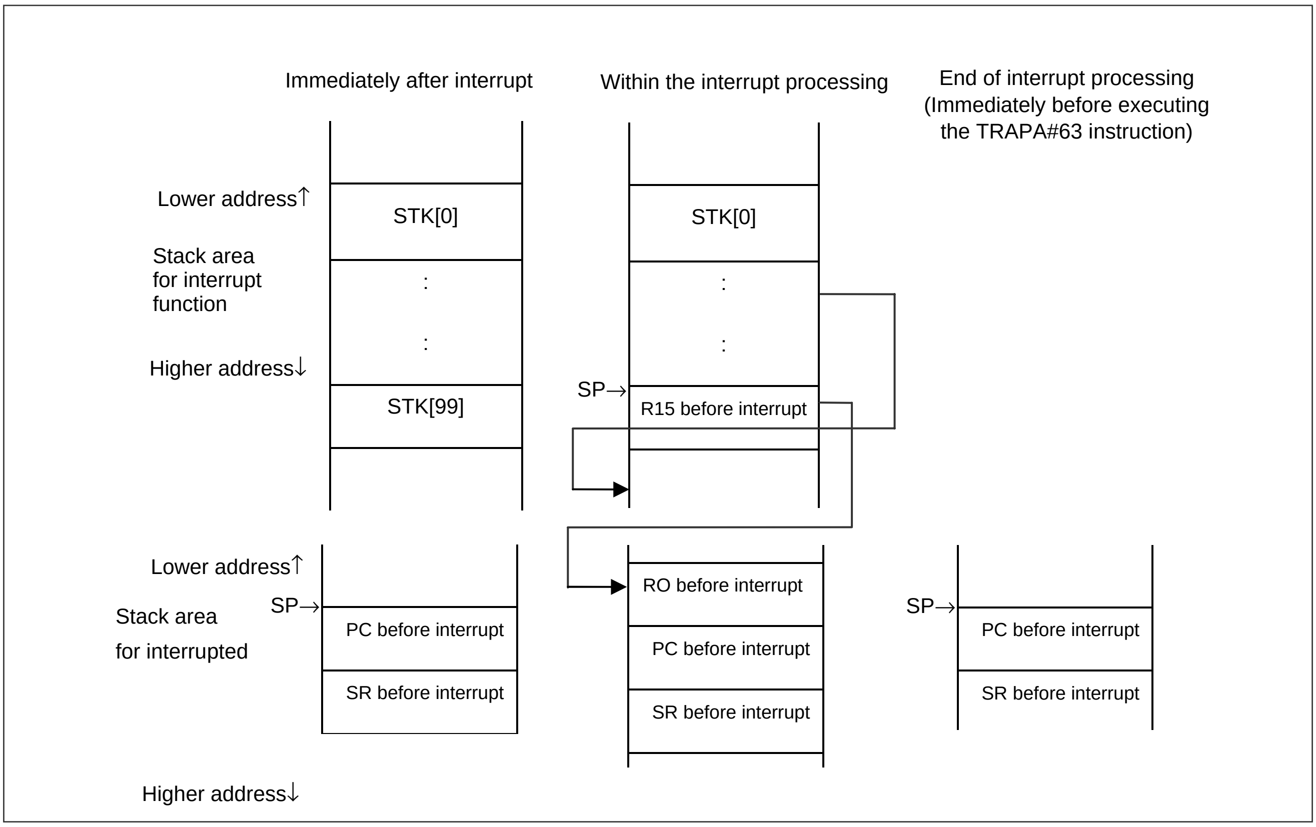


Figure 3.1 Example of Stack Use by an Interrupt Function

Example 2

C language code

```
#pragma interrupt(handler(resbank))
double flag1;
int  flag2;
void handler()
{
           flag1 = 1;
           flag2 = 1;
}
```

Expanded into assembly language code ("-cpu=sh2afpu" specified)

```
 _handler:
                                        ; Does not output the save code for the bank target
                                        ; work registers.
          FMOV.S      FR8,@-R15         ; Saves the work registers other than the bank target
          FMOV.S      FR9,@-R15         ; registers.
          MOVA        L11,R0
          MOV         #1,R4
          FMOV.S      @R0+,FR8
          MOV.L       L11+8,R1
          FMOV.S      @R0,FR9
          MOV.L       L11+12,R6
          FMOV.S      FR9,@-R1
          FMOV.S      FR8,@-R1
          MOV.L       R4,@R6
          FMOV.S      @R15+,FR9         ; Restores the work registers other than the
          FMOV.S      @R15+,FR8         ; bank target registers.
          RESBANK                       ; Restores the bank target registers.
          RTE
          NOP
```

Important Information:

(1) The "resbank" specification is valid when "cpu=sh2a|sh2afpu" is specified.

(2) The register bank must be enabled before an interrupt occurs in the function with "resbank" specified.

(3) If you specify "resbank" and "tn", neither the register save code nor the RESBANK instruction will be generated. In this case, specify that the RESBANK instruction is generated in the trap routine.

(4) When the registers are restored from the function with "resbank" specified, the variable values specified in "#pragma global_register" return to the values before the interrupt even if they were changed during interrupt processing.

3.1.3 Creating a Vector Table

Description:

A vector table can be created in the C language by making the following settings.

(1) An array for use with the vector table is prepared, and an exception processing function pointer is specified for each element in the array.

(2) After compiling the file, the start address of the vector table is specified to link the file.

Example of use:

C language code: vect_table.c

```
extern void reset(void);          /* Power-on reset processing function   */
extern void warm reset(void);     /* Manual reset processing function     */
extern void irq0(void);           /* IRQ0 interrupt processing function   */
extern void irq1(void);           /* IRQ1 interrupt processing function   */
          :
          :
void(* const vect_table[])(void) = {
             reset,               /* Start address of power-on reset    */
             0,                   /* Stack pointer of power-on reset    */
             warm reset,          /* Start address of manual reset      */
             0,                   /* Stack pointer of manual reset      */
              :
              :
             irq0,                /* Vector number 64 */
             irq1,                /* Vector number 65 */
              :
              :
};
```

Batch file

```
shcΔ-section=c=VECTΔvect_table
shcΔresetΔwarm_resetΔirq0Δirq1 …
optlnkΔ-nooptΔ-subcommand=link.sub
```

The contents of link.sub are as follows.

```
      sdebug
      input vect_table
      input reset
      input warm_reset
      input irq0,irq1
      …
      output sample.abs
      start VECT/0,P,C,D/0400,B/0F000000
      exit
```

By compiling vect_table.c, a relocatable object file vect table.obj is generated for the initialization data section (VECT) only.
The section VECT is set to the start address H'0 and linked together with other files, to obtain the load module file sample.abs.

Expanded into assembly language code: vect_table.src

```
          .IMPORT     _reset
          .IMPORT     _warm_reset
          .IMPORT     _irq0
          .IMPORT     _irq1
          .EXPORT     _vect_table
          .SECTION    VECT,DATA,ALIGN=4
_vect_table:                                      ; static: vect_table
          .DATA.L     _reset
          .DATA.L     H'00000000
          .DATA.L     _warm_reset
          .DATA.L     H'00000000
              :
              :
          .DATA.L     _irq0, _irq1
              :
              :
          .END
```

Important Information:

(1)In the event of an interrupt, the operation of SH-3, SH3-DSP, SH-4, SH-4A and SH4AL-DSP differs from that of SH-1, SH-2, SH-2E,SH-2A, SH2A-FPU, and SH2-DSP, and an interrupt handler is necessary.

(2) The vector table must be assigned to a defined absolute address, and so here it is created as an independent file; but by using section-switching functions, it can be included in a file with other modules. For details, refer to section 3.7, Section Name Specification.

3.2 Built-in Functions

The built-in functions shown in table 3.2 are provided to enable representation in C of instructions inherent to the SH-1, SH-2, SH-2E, SH-2A, SH2A-FPU, SH2-DSP, SH-3, SH3-DSP, SH-4, SH-4A, and SH4AL-DSP. When using these built-in functions, the standard header file "machine.h", "smachine.h" or "umachine.h" must be included.
The contents of <machine.h> are divided as follows according to the execution mode of SH-3, SH3-DSP, SH-4, SH-4A, and SH4AL-DSP:

(1) <machine.h>: All built-in functions

(2) <smachine.h>: Built-in functions that can be used only in the privileged mode

(3) <umachine.h>: Built-in functions other than those in (2)

Table 3.2 List of Built-in Functions (1)

No.	Item	Function	Available CPU	Available Execution Mode	See Section:
1	Status register (SR)	Sets the SR.	All CPUs	Privileged mode only	3.2.1
2		Refers to the SR.			
3		Sets the interrupt mask.			
4		Refers to the interrupt mask.			
5	Vector base register (VBR)	Sets the VBR.	All CPUs	Privileged mode only	3.2.2
6		Refers to the VBR.			
7	Global base register (GBR)	Sets the GBR.	All CPUs	All execution modes	3.2.3
8		Refers to the GBR.			3.2.4
9		Refers to byte data at the address specified by the GBR and offset.			
10		Refers to word data at the address specified by the GBR and offset.			
11		Refers to long word data at the address specified by the GBR and offset.			
12		Sets byte data at the address specified by the GBR and offset.			
13		Sets word data at the address specified by the GBR and offset.			

Table 3.2 List of Built-in Functions (2)

No.	Item	Function	Available CPU	Available Execution Mode	See Section:
14	Global base register (GBR)	Sets long word data at the address specified by the GBR and offset.	All CPUs	All execution modes	3.2.3 3.2.4
15		ANDs byte data at the address specified by GBR.			
16		ORs byte data at the address specified by GBR.			
17		XORs byte data at the address specified by GBR.			
18		Tests byte data at the address specified by GBR.			
19	System control	SLEEP instruction	All CPUs	Privileged mode only	3.2.5
20		TAS instruction		All execution modes	
21		TRAPA instruction			
22	Multiply-and-accumulate operations	Multiplication and accumulation in word units	All CPUs	All execution modes	3.2.6
23		Multiplication and accumulation in long word units	Excluding cpu=sh1		
24		Ring-buffer compatible multiplication and accumulation in word units	All CPUs	All execution modes	3.2.7
25		Ring-buffer compatible multiplication and accumulation in long word	Excluding cpu=sh1		
26	64-bit multiplication	Upper 32 bits of multiplication of signed 64-bit data	Excluding cpu=sh1	All execution modes	3.2.8
27		Lower 32 bits of multiplication of signed 64-bit data			
28		Upper 32 bits of multiplication of unsigned 64-bit data	Excluding cpu=sh1	All execution modes	3.2.9
29		Lower 32 bits of multiplication of unsigned 64-bit data			

Table 3.2 List of Built-in Functions (3)

No.	Item	Function	Available CPU	Available Execution Mode	See Section:
30	Swapping upper and lower data bits	SWAP.B instruction	All CPUs	All execution modes	3.2.10
31		SWAP.W instruction			
32		Swaps upper and lower bits of 4-byte data.			
33	System call	Performs system call.	All CPUs	All execution modes	3.2.11
34	Prefetch instruction	Prefetch instruction	Only when cpu=sh2a, sh2afpu, sh3, sh3dsp, sh4, sh4a, sh4aldsp is specified	All execution modes	3.2.12
35	LDTLB instruction	Expands the LDTLB.	Only when cpu=sh3, sh3dsp, sh4, sh4a, sh4aldsp is specified	Privileged mode only	3.2.13
36	NOP instruction	Expands the NOP.	All CPUs	All execution modes	3.2.14
37	Floating-point unit	Sets the FPSCR.	Only when cpu=sh2e, sh2afpu, sh4, sh4a is specified	All execution modes	3.2.15
38		Refers to the FPSCR.			
39	Single-precision floating point	FIPR instruction	Only when cpu=sh4, sh4a is specified	All execution modes	
40	vector operation	FTRV instruction			
41		Performs 4-dimensional vector to 4 x 4 matrix translation and 4-dimensional vector addition.			
42		Performs 4-dimensional vector to 4 x 4 matrix translation and 4-dimensional vector subtraction.			
43		Performs 4-dimensional vector addition.	Only when cpu=sh2afpu, sh4, sh4a is specified		
44		Performs 4-dimensional vector subtraction.			
45		Performs 4x4 matrix multiplication.	Only when cpu=sh4, sh4a is specified		
46		Performs 4x4 matrix multiplication and addition.			
47		Performs 4x4 matrix multiplication and subtraction.			
48	Extension register access	Loads data to extension registers.	Only when cpu=sh4, sh4a is specified	All execution modes	3.2.16
49		Restores data from extension registers.			

Table 3.2 List of Built-in Functions (4)

No.	Function	Description	Available CPU	Available Execution Mode	See Section:
50	DSP instruction	Absolute value	When sh2dsp, sh3dsp, sh4aldsp, and dspc are specified	All execution modes	3.2.17
51		MSB detection			
52		Arithmetic shift			
53		Rounding-off operation			
54		Bit pattern copy			
55		Modulo addressing setup		Privileged mode only	
56		Modulo addressing cancellation			
57		CS bit setting (DSR register)		All execution modes	
58	Sine and cosine	Sine and cosine calculation	Only when cpu=sh4a is specified	All execution modes	3.2.18
59	Reciprocal of the square root	Reciprocal of the square root	Only when cpu=sh4a is specified	All execution modes	3.2.19
60	Instruction cache invalidation	Instruction cache block invalidation	Only when cpu=sh4a, sh4aldsp is specified	All execution modes	3.2.20
61	Cache block operations	Cache block invalidation	Only when cpu=sh4a, sh4aldsp is specified	All execution modes	3.2.21
62		Cache block purge			
63		Cache block write-back			
64	Instruction cache prefetch	Prefetch of the instruction cache block	Only when cpu=sh4a, sh4aldsp is specified	All execution modes	3.2.22
65	System synchronization	Synchronizes data operations.	Only when cpu=sh4a, sh4aldsp is specified	All execution modes	3.2.23

Table 3.2 List of Built-in Functions (5)

No.	Function	Description	Available CPU	Available Execution Mode	See Section:
66	Referencing and setting the T bit	References the T bit.	All CPUs	All execution modes	3.2.24
67		Clears the T bit.			
68		Sets the T bit.			
69	Cutting out the middle of the concatenated register	Cuts out the middle 32 bits from the concatenated 64-bit data.	All CPUs	All execution modes	3.2.25
70	Addition with carry	Adds two data items and the T bit, and applies the carry to the T bit.	All CPUs	All execution modes	3.2.26
71		Adds two data items and the T bit, and references the carry.			
72		Adds two data items, and applies the carry to the T bit.			
73		Adds two data items, and references the carry.			
74	Subtraction with borrow	Subtracts data2 and the T bit from data1, and applies the borrow to the T bit.	All CPUs	All execution modes	3.2.27
75		Subtracts data2 and the T bit from data1, and references the borrow.			
76		Subtracts data2 from data1, and applies the borrow to the T bit.			
77		Subtracts data2 from data1, and references the borrow.			
78	Sign inversion	Subtracts the data and the T bit from 0, and applies the borrow to the T bit.	All CPUs	All execution modes	3.2.28
79	One-bit division	Performs a one-step division of data1 by data2, and applies the result to the T bit.	All CPUs	All execution modes	3.2.29
80		Performs the initialization for the signed division of data1 by data2, and references the T bit.			
81		Performs the initialization for the unsigned division.			

Table 3.2 List of Built-in Functions (6)

No.	Function	Description	Available CPU	Available Execution Mode	See Section:
82	Rotation	Rotates the data left by one bit, and then applies the bits that moved outside the operand to the T bit.	All CPUs	All execution modes	3.2.30
83		Rotates the data right by one bit, and then applies the bits that moved outside the operand to the T bit.			
84		Rotates the data left by one bit including the T bit, and then applies the bits that moved outside the operand to the T bit.			
85		Rotates the data right by one bit including the T bit, and then applies the bits that moved outside the operand to the T bit.			
86	Shift	Shifts the data to the left by one bit, and then applies the bits that moved outside the operand to the T bit.	All CPUs	All execution modes	3.2.31
87		Logically shifts the data to the right by one bit, and then applies the bits that moved outside the operand to the T bit.			
88		Arithmetically shifts the data to the right by one bit, and then applies the bits that moved outside the operand to the T bit.			
89	Saturation operation	Saturation operation on signed one-byte data	Only when cpu=sh2a, sh2afpu is specified	All execution modes	3.2.32
90		Saturation operation on signed two-byte data			
91		Saturation operation on unsigned one-byte data			
92		Saturation operation on unsigned two-byte data			
93	Referencing and setting the TBR	Sets the data in the TBR.	Only when cpu=sh2a, sh2afpu is specified	All execution modes	3.2.33
94		References the value of the TBR.			

3.2.1 Setting and Referencing to the Status Register

Description:

The functions shown in table 3.3 are provided for use in setting and referencing to the status register.

Table 3.3 List of Built-in Functions for the Status Register

No.	Function	Format	Description
1	Status register setting	void set_cr(int cr)	Sets cr (32 bits) in the status register.
2	Status register reference	int get_cr(void)	Refers to the status register.
3	Sets the interrupt mask	void set_imask(int mask)	Sets mask (4 bits) in the interrupt mask (4 bits).
4	Refers to the interrupt mask	int get_imask(void)	Refers to the interrupt musk status (4 bits)

Example of use:

By setting the interrupt mask to the highest value (15), the function func1 disables external interrupts during processing. After the completion of processing, the original interrupt mask level is restored.

C language code

```
#include <machine.h>

void  func1(void)
{
    int   mask;               /* Defines the storage area for interrupt mask level   */

    mask = get imask();       /* Saves interrupt mask level                          */
    set imask(15);            /* Sets the interrupt mask level to 15                 */
          :                   /* Performs processing with interrupt disabled         */
          :
    set imask(mask);          /* Restores interrupt mask level                       */
}
```

Expanded into assembly language code

```
          .EXPORT      _func1
          .SECTION     P,CODE,ALIGN=4
_func1:                                 ; function: func1
                                        ; frame size=0
          MOV.W        L216,R3          ; H'FF0F
          STC          SR,R0
          SHLR2        R0
          SHLR2        R0
          AND          #15,R0
          MOV          R0,R4
          STC          SR,R0
          AND          R3,R0
          OR           #240,R0
          LDC          R0,SR
           :
           :
          MOV          R4,R0
          AND          #15,R0
          SHLL2        R0
          SHLL2        R0
          STC          SR,R2
          MOV          R3,R1
          AND          R1,R2
          OR           R2,R0
          LDC          R0,SR
          RTS
          NOP
L216:
          .DATA.W      H'FF0F
          .END
```

3.2.2 Setting and Referencing to the Vector Base Register

Description:

Table 3.4 shows the functions provided for setting and reading the vector base register.

Table 3.4 List of Built-in Functions for the Vector Base Register

No.	Function	Format	Description
1	Vector base register setting	void set_vbr(void **base)	Sets **base (32 bits) in the vector base register.
2	Vector base register reference	void **get_vbr(void)	Refers to the vector base register.

Example of use:

Upon reset, the vector base register (VBR) is initialized to 0. If the vector table is begun at an address other than 0, the following function should be set at the start address (H'00000008) on manual reset, so that if manual reset is performed when the system is started, exception processing can be performed using the vector table.

C language code

```
#include <machine.h>
#define VBR  0x0000FC00          /* Defines the start address of vector table    */

void  warm reset(void)
{
    set_vbr((void **)VBR);
                                 /* Sets the vector base register to the start   */
                                 /* address of vector table                      */
}
```

Expanded into assembly language code

```
       .EXPORT      _warm_reset
       .SECTION     P,CODE,ALIGN=4
 _warm reset:                          ; function: warm reset
                                       ; frame size=0
       MOV.L        L215,R3            ; H'0000FC00
       LDC          R3,VBR
       RTS
       NOP
 L215:
       .DATA.L      H'0000FC00
       .END
```

Important Information:

The vector base register should be changed only after setting the vector table. If the order is reversed, and an external interrupt occurs while setting the vector table, a system crash will occur.

3.2.3 Accessing I/O Registers(1)

Description:

The functions shown in table 3.5 are provided for use in manipulating the global base register (GBR) to access I/O registers.

Table 3.5 List of Built-in Functions for Use with the Global Base Register

No.	Function	Format	Description
1	Global base register setting	void set_gbr(void **base)	Sets *base (32 bits) in the global base register
2	Global base register reference	int *get_gbr(void)	Refers to the global base register
3	Reference of byte data at the address specified by GBR and offset	unsigned char gbr_read_byte(int offset)	Refers to the byte data (8 bits) at the address specified by adding the GBR and the offset
4	Reference of word data at the address specified by GBR and offset	unsigned short gbr_read_word(int offset)	Refers to the word data (16 bits) at the address specified by adding the GBR and the offset
5	Reference of long-word data at the address specified by GBR and offset	unsigned long gbr_read_long(int offset)	Refers to the long- word data (32 bits) at the address specified by adding the GBR and the offset
6	Setting of byte data at the address specified by GBR and offset	void gbr_write_byte(int offset,unsigned char data)	Specifies the word data (8 bits) at the address specified by adding the GBR and the offset
7	Setting of word data at the address specified by GBR and offset	void gbr_write_word(int offset,unsigned short data)	Specifies the word data (16 bits) at the address specified by adding the GBR and the offset
8	Setting of long-word data at the address specified by GBR and offset	void gbr_write_long(int offset,unsigned long data)	Specifies the word data (32 bits) at the address specified by adding the GBR and the offset
9	AND operation of byte data at the address specified by GBR and offset	void gbr_and_byte(int offset,unsigned char mask)	ANDs mask and the byte data at the address specified by the GBR and the offset, and sets the result in offset
10	OR operation of byte data at the address specified by GBR and offset	void gbr_or_byte(int offset,unsigned char mask)	ORs mask and the byte data at the address specified by the GBR and the offset, and sets the result in offset
11	XOR operation of byte data at the address specified by GBR and offset	void gbr_xor_byte(int offset,unsigned char mask)	XORs mask and the byte data at the address specified by the GBR and the offset, and sets the result in offset
12	Test of byte data at the address specified by GBR and offset	void gbr_tst_byte(int offset,unsigned char mask)	ANDs mask and the byte data at the address specified by the GBR and the offset, compares the result with 0, and set or clear the T bit depending on the comparison result

Notes: (1) When the access size is a word, base should be set to a multiple of 2; when the access size is a long word, it should be set to a multiple of 4.

(2) For numbers 3 through 8 in table 3.5, the offset must be a constant. An offset of up to +255 bytes may be specified when the access size is a byte, and an offset of up to +510 bytes can be specified when the access size is a word. In addition, an offset of up to +1020 bytes can be specified when the access size is a long word.

(3) The mask must be a constant. The mask can be specified within the range 0 to +255.

(4) The global base register is a control register, and so the C/C++ compiler does not save and restore values on function entry and exit. When changing the global base register value, the user must save and restore the register value on function entry and exit.

(5) This function is invalid when gbr=auto is specified. (Ver.7 or later)

Example of use:

The following is an example of a timer driver using the SH-1internal 16-bit integrated timer pulse unit.

C language code

```
#include <machine.h>
#define IOBASE 0x05fffec0           /* Defines I/O base address                  */
#define TSR   (0x05ffff07 - IOBASE)
                                    /* Clears the compare match flag of the      */
                                    /* register                                  */
#define TSRCLR (unsigned char)0xf8
                                    /* Clears the compare match flag of the      */
                                    /* timer status flag register                */
void  tmrhdr(void)
{
    void    *gbrsave;               /* Defines the stack area for the global     */
                                    /* base register                             */
    gbrsave = get gbr();            /* Saves the global base register            */
    set gbr((void *)IOBASE);
                                    /* Specifies I/O base register in the global */
                                    /* register                                  */
    gbr read byte(TSR);             /* Reads the timer status flag register to   */
                                    /* clear it                                  */
    gbr and byte(TSR, TSRCLR);
                                    /* Clears the compare match flag of the      */
                                    /* timer status flag register                */
    set gbr(gbrsave);               /* Restores the global base register         */
}
```

Expanded into assembly language code

```
          .EXPORT      _tmrhdr
          .SECTION     P,CODE,ALIGN=4
_tmrhdr:                             ; function: tmrhdr
                                     ; frame size=4
          ADD          #-4,R15
          STC          GBR,R1
          MOV.L        L11,R5        ; H'05FFFEC0
          MOV.L        R1,@R15
          LDC          R5,GBR
          MOV.B        @(71,GBR),R0
          MOV          #71,R0        ; H'00000047
          AND.B        #248,@(R0,GBR)
          MOV.L        @R15,R2
          LDC          R2,GBR
          RTS
          ADD          #4,R15
L11:
          .DATA.L      H'05FFFEC0
```

3.2.4 Accessing I/O Registers(2)

Example of use:

By using the standard library offsetof, the need to calculate the global base register relative offset in advance is eliminated.

C language code

```
#include <stddef.h>
#include <machine.h>
struct IOTBL{
   char  cdata1;                 /* offset 0   */
   char  cdata2;                 /* offset 1   */
   char  cdata3;                 /* offset 2   */
   short sdata1;                 /* offset 4   */
   int   idata1;                 /* offset 8   */
   int   idata2;                 /* offset 12  */
} table;

void f (void)
{
     void      *gbrsave;         /* Defines the stack area for the global   */
                                 /* base register                           */
     gbrsave = get gbr();        /* Saves the global base register          */
     set gbr(&table);            /* Sets the table start address in the     */
                                 /* global base register                    */
              :
              :
     gbr and byte(offsetof(struct IOTBL, cdata2), 0x10);
                                 /* ANDs table.cdata2 and 0x10, and         */
                                 /* saves the result in table.cdata2        */
              :
              :
     set gbr(gbrsave);           /* Restores the global base register       */
}
```

Expanded into assembly language code

```
      .EXPORT     _table
      .EXPORT     _f
      .SECTION    P,CODE,ALIGN=4
 _f:                                      ; function: f
                                          ; frame size=0
      MOV.L       L217+2,R3               ; table
      MOV         #1,R0
      STC         GBR,R4
      LDC         R3,GBR
         :
         :
      AND.B       #16,@(R0,GBR)
         :
         :
      RTS
      LDC         R4,GBR
 L217 :
      .RES.W      1
      .DATA.L     _table
      .SECTION    B,DATA,ALIGN=4
 _table:                                  ; static: table
      .RES.L      4
      .END
```

3.2.5 System Control

Description:

Table 3.6 shows the functions provided as special instructions for the Renesas Technology SuperH RISC engine family.

Table 3.6 List of Built-in Functions for Special Instructions

No.	Function	Format	Description
1	SLEEP instruction	void sleep(void)	Expands into the SLEEP instruction
2	TAS instruction	int tas(char *addr)	Expands into TAS.B @addr
3	TRAPA instruction	int trapa(int trap_no)	Expands into TRAPA #trap_no
4	OS system call	-	Refer to section 3.2.11
5	PREF instruction	-	Refer to section 3.2.12

Notes: (1) In the table, trap_no must be a constant.

(2) The built-in function trapa starts an interrupt function from the C language program. Create the called function as an interrupt function.

Example of use:

A SLEEP instruction is issued to put the CPU into a low-power consumption mode. In the low-power consumption mode, the internal state of the CPU is saved, execution of the instruction immediately following is halted, and the system waits for an interrupt request. When an interrupt occurs, the CPU leaves the low-power consumption mode.

C language code

```
#include <machine.h>
void  func(void)
{
       :
       :
    sleep();                         /* Issues the SLEEP instruction */
       :
       :
}
```

Expanded into assembly language code

```
       .EXPORT    _func
       .SECTION   P,CODE,ALIGN=4
_func:                               ; function: func
                                     ; frame size=0
       SLEEP
       RTS
       NOP
       .END
```

3.2.6 Multiply-and-Accumulate Operations (1)

Description:

Table 3.7 shows the functions provided for multiply-and-accumulate operations.

Table 3.7 List of Built-in Functions for Multiply-and-Accumulate Operations

No.	Function	Format	Description
1	Multiplication and accumulation in word units	int macw(short *ptr1, short *ptr2,unsigned int count)	Performs multiplication and accumulation between word data *ptr1 (16 bits) and word data *ptr2 (16 bits) for the number of times specified by count
2	Multiplication and accumulation in long word units	int macl(int *ptr1, int *ptr2,unsigned int count)	Performs multiplication and accumulation between long-word data *ptr1 (32 bits) and long-word data *ptr2 (32 bits) for the number of times specified by count

The word multiply-and-accumulate function macw is supported in all the CPUs SH-1, SH-2, SH-2E, SH-2A, SH2A-FPU, SH2-DSP, SH-3, SH3-DSP, SH-4, SH-4A, and SH4L-DSP. The long-word multiply-and-accumulate function macl is supported in the CPUs other than SH-1.

The built-in multiple-and-accumulate functions do not perform parameter checks. Tables for data for multiply-and-accumulate operations should be aligned by two bytes for word operations, and by four bytes for long word operations.

Example of use:

A multiply-and-accumulate operation is performed. If the number of executions is 32 or less, they are performed by repetition of the MAC instruction; if 33 or more executions are performed, or if the number of executions is a variable, the MAC instruction is looped.

C language code

```
#include <machine.h>
short a[SIZE];
short b[SIZE];

Void  func(void)
{
      int x;
            :
            :
      x = macw(a,b,SIZE);
            :
            :
}
```

```
  a[0]          *     b[0]          +
  a[1]          *     b[1]          +
  a[2]          *     b[2]          +
:                         :         +
  a[SIZE-2]     *     b[SIZE-2]     +
  a[SIZE-1]     *     b[SIZE-1]
```

Expanded into assembly language code

- case of size ≤ 32 : Achieved by executing the MAC instruction repeatedly

```
         .EXPORT _func
         .SECTION P,CODE,ALIGN=4
_func:                           ; function: func
                                 ; frame size=8
         STS.L    MACH,@-R15
         STS.L    MACL,@-R15
         MOV.L    L217+2,R3      ; _b
         CLRMAC
         MOV.L    L218+6,R2      ; _a
         MAC.W    @R2+,@R3+
           :                     ; Repeats for SIZE
           :
         STS      MACL,R0
         LDS.L    @R15+,MACL
         RTS
         LDS.L    @R15+,MACH
L218 :
        .RES.W    1
        .DATA.L   _b
        .DATA.L   _a
```

- case of size > 32 or variable:Achieved by the loop function of the MAC instruction

```
        .EXPORT      _func
        .SECTION     P,CODE,ALIGN=4
_func:                                     ; function: func
                                          ; frame size=8
        STS.L        MACH,@-R15
        MOV          #33,R3
        STS.L        MACL,@-R15
        TST          R3,R3
        CLRMAC
        BT           L218
        MOV.L        L220+2,R2            ; _b
        SHLL         R3
        MOV.L        L220+6,R1            ; _a
        ADD          R1,R3
L219 :
        MAC.W        @R1+,@R2+
        CMP/HI       R1,R3
        BT           L219
L218 :
        STS          MACL,R0
        LDS.L        @R15+,MACL
        RTS
        LDS.L        @R15+,MACH
L220 :
        .RES.W       1
        .DATA.L      _b
        .DATA.L      _a
```

3.2.7 Multiply-and-Accumulate Operations (2)

Description:

Table 3.8 shows functions provided for ring buffer-compatible multiply-and-accumulate operations.

Table 3.8 List of Built-in Functions for Ring Buffer-Compatible Multiply-and-Accumulate Operations

No.	Function	Format	Description
1	Ring-buffer compatible multiplication and accumulation in word units	int macwl(short *ptr1, short *ptr2,unsigned int count, unsigned int mask)	Performs multiplication and accumulation between word data *ptr1 (16 bits) and word data *ptr2 (16 bits) for the number of times specified by count
2	Ring-buffer compatible multiplication and accumulation in long-word units	int macll(int *ptr1, int *ptr2,unsigned int count, unsigned int mask)	Performs multiplication and accumulation between long-word data *ptr1 (32 bits) and long-word data *ptr2 (32 bits) for the number of times specified by count

The ring-buffer compatible word multiply-and-accumulate function macwl is supported in all the CPUs SH-1, SH-2, SH-2E, SH-2A, SH2A-FPU, SH2-DSP, SH-3, SH3-DSP, SH-4, SH-4A, and SH4AL-DSP, in SH-2, SH-2E, SH-3, and SH-4. The ring-buffer compatible long word multiply-and-accumulate function macll is supported in the CPUs other than SH-1.
The built-in ring buffer-compatible multiple-and-accumulate functions do not perform parameter checks. The first parameter should be aligned by two bytes in the case of a word multiply-and-accumulate operation, and by four bytes for long word operations; the second parameter should be aligned for twice the size of the ring buffer mask.

Example of use:

A ring buffer-compatible multiply-and-accumulate operation is performed. The second parameter must be aligned to twice the size of the ring buffer, and so a separate file is used.

C language code:macwl.c

```
#include <machine.h>

short a[SIZE];
extern short b[];

void  func(void)
{
     int x;
         :
         :
     x = macwl(a,b,SIZE,~0x10);

         :
         :

}
```

```
  a[0]          *        b[0]        +
  a[1]          *        b[1]        +
    :                      :         +
  a[7]          *        b[7]        +
  a[8]          *        b[0]        +
  a[9]          *        b[1]        +
       :                   :         +
a[15]           *        b[7]        +
    :                      :         +
a[SIZE-8]       *        b[0]        +
a[SIZE-7]       *        b[1]        +
    :                      :         +
a[SIZE-1]       *        b[7]
```

Expanded into assembly language code:macwl.src

```
            .IMPORT   _b
            .EXPORT   _a
            .EXPORT   _func
            .SECTION P,CODE,ALIGN=4
_func:                                      ; function: func
                                            ; frame size=8
            STS.L        MACH,@-R15
            MOV          #33,R3
            STS.L        MACL,@-R15
            TST          R3,R3
            CLRMAC
            BT           L218
            MOV.L        L220+6,R1          ; _b
            SHLL         R3
            MOV.L        L220+6,R4          ; _a
            MOV          #-17,R2
            ADD          R4,R3
L219 :
            MAC.W        @R4+,@R1+
            AND          R2,R1
            CMP/HI       R4,R3
            BT           L219
L218 :
            STS          MACL,R0
            LDS.L        @R15+,MACL
            RTS
            LDS.L        @R15+,MACH
L220 :
            .RES.W       1
            .DATA.L      _b
            .DATA.L      _a
            .SECTION     B,DATA,ALIGN=4
_a :                                        ; static: a
            .RES.W       33
            .END
```

3.2.8 64-Bit Multiplication (1)

Table 3.9 shows functions provided for signed 64-bit multiplication .

Table 3.9 List of Built-in Functions for Signed 64-Bit Multiplication

No.	Function	Format	Description
1	Upper 32-bits of multiplication of signed 64-bit data	long dmuls_h(long data1, long data2)	Signed 32-bit × signed 32-bit performs multiplication of signed 64-bit data, and returns a result of the upper 32-bits
2	Lower 32-bits of multiplication of signed 64-bit data	long dmuls_l(long data1, long data2)	Signed 32-bit × signed 32-bit performs multiplication of signed 64-bit data, and returns a result of the lower 32-bits

Multiplication of signed 64-bit data is supported by all except SH-1.

Example of use:

C language code

```
#include <machine.h>
extern long data1,data2;
extern long result;
void main( void )
{
    result = dmuls_h(data1,data2);
                                                    /*  Performs multiplication of signed 64-bit data  */
}
```

Expanded into assembly language code

```
          .IMPORT    _result
          .IMPORT    _data1
          .IMPORT    _data2
          .EXPORT    _main
          .SECTION    P,CODE,ALIGN=4
_main:                                    ; function: main
                                          ; frame size=8
          STS.L       MACL,@-R15
          STS.L       MACH,@-R15
          MOV.L       L11+2,R2            ; _data1
          MOV.L       L11+6,R5            ; _data2
          MOV.L       @R2,R6
          MOV.L       @R5,R2
          DMULS.L     R6,R2
```

```
           MOV.L       L11+10,R6         ; _result
           STS         MACH,R2
           MOV.L       R2,@R6
           LDS.L       @R15+,MACH
           RTS
           LDS.L       @R15+,MACL
 L11:
           .RES.W      1
           .DATA.L    _data1
           .DATA.L    _data2
           .DATA.L    _result
           .END
```

3.2.9 64-Bit Multiplication (2)

Table 3.10 shows functions provided for unsigned 64-bit multiplication .

Table 3.10 List of Built-in Functions for Unsigned 64-Bit Multiplication

No.	Function	Format	Description
1	Upper 32-bits of multiplication of unsigned 64-bit data	long dmulu_h(long data1, long data2)	Unsigned 32-bit × unsigned 32-bit performs multiplication of unsigned 64-bit data, and returns a result of the upper 32-bits
2	Lower 32-bits of multiplication of unsigned 64-bit data	long dmulu_l(long data1, long data2)	Unsigned 32-bit × unsigned 32-bit performs multiplication of unsigned 64-bit data, and returns a result of the lower 32-bits

Multiplication of unsigned 64-bit data is supported by all except SH-1.

Example of use:

C language code

```
#include <machine.h>
extern long data1,data2;
extern long result;
void main( void )
{
  result = dmulu_h(data1,data2);
                                                /* Performs multiplication of signed 64-bit data */
}
```

Expanded into assembly language code

```
          .IMPORT     _result
          .IMPORT     _data1
          .IMPORT     _data2
          .EXPORT     _main
          .SECTION    P,CODE,ALIGN=4
_main:                                ; function: main
                                      ; frame size=8
          STS.L       MACL,@-R15
          STS.L       MACH,@-R15
          MOV.L       L11+2,R2        ; _data1
          MOV.L       L11+6,R5        ; _data2
          MOV.L       @R2,R6
          MOV.L       @R5,R2
          DMULU.L     R6,R2
          MOV.L       L11+10,R6       ; _result
          STS         MACH,R2
          MOV.L       R2,@R6
          LDS.L       @R15+,MACH
          RTS
          LDS.L       @R15+,MACL
L11:
          .RES.W      1
          .DATA.L     _data1
          .DATA.L     _data2
          .DATA.L     _result
          .END
```

3.2.10 Swapping Upper and Lower Data

Table 3.11 shows functions provided for swapping upper and lower data.

Table 3.11 List of Built-in Functions for Ring Buffer-Compatible Multiply-and-Accumulate Operations

No.	Function	Format	Description
1	SWAP.B instruction	unsigned short swapb(unsigned short data)	Swaps the upper and lower 1 byte of 2-byte data
2	SWAP.W instruction	unsigned long swapw(unsigned long data)	Swaps the upper and lower 2 bytes of 4-byte data
3	Swap upper and lower bits of 4-byte data	unsigned long end_cnvl(unsigned long data)	Arranges 4-byte data by single bytes with upper and lower in reverse order

Example of use:

C language code

```
#include <machine.h>
extern unsigned short data;
extern unsigned short result;
void main( void )
{
      result = swapb(data);
                                             /* If data = 0x1234, result = 0x3412.*/
}
```

Expanded into assembly language code

```
          .IMPORT    _result
          .IMPORT    _data
          .EXPORT    _main
          .SECTION    P,CODE,ALIGN=4
_main:                              ; function: main
                                    ; frame size=0
          MOV.L       L11,R6        ; _data
          MOV.W       @R6,R2
          SWAP.B      R2,R6
          MOV.L       L11+4,R2      ; _result
          RTS
          MOV.W       R6,@R2
L11:
          .DATA.L    _data
          .DATA.L     result
          .END
```

3.2.11 System Call

Description:

The format for built-in functions capable of issuing system calls from a C language program is indicated below. The number of Parameters for a system call is variable between 0 and 4.
However, TRAPA cannot directly call a C function with an RTE return.
In actual practice, trapa_svc (a C built-in function) should be used to register a handler function (which should be written in assembly language), and the R0 function code tested to call each routine.
Returns from this routine written in assembly language should be by RTE.

- Format:

```
ret=trapa_svc(int trap_no, int code,
   [type1 p1[, type2 p2[, type3 p3[, type4 p4]]]])
   trap_no          :trap number (specified by a constant)
   code             :function code, assigned to R0
   p1               :first parameter, assigned to R4
   p2               :second parameter, assigned to R5
   p3               :third parameter, assigned to R6
   p4               :fourth parameter, assigned to R7
   type1 to type4   :parameter types are integer types([unsigned]char,
                       [unsigned]short,[unsigned]int,
                       [unsigned]long), or the pointer type
```

Example of use:

Using this function, an OS system call is issued which can be specified using trap number #63.

C language code

```
#include <machine.h>
#define  SIG SEM  0xffc8
void  main(void)
{
       :
       :
    trapa svc(63, SIG SEM, 0x05);
       :
       :
}
```

Expanded into assembly language code

```
          .EXPORT      _main
          .SECTION     P,CODE,ALIGN=4
_main:     :                             ; function: main
          :                             ; frame size=0
          MOV.L        L215+2,R0        ; H'0000FFC8
          MOV         #5,R4
          TRAPA       #63
           :
           :
          RTS
          NOP
L215:
          .RES.W      1
          .DATA.L     H'0000FFC8
          .END
```

Vector table definition

```
void(*const vect[])(void)={
      :
      :
   HDR,.......                  /* Defines HDR in trap 63 vector */
      :
} ;
```

Handler (written in assembly language)

```
        .IMPORT   _func
HDR:
                                ; Saves the PR and R1 to R7 registers
                                ; Selects the function to be called according to the R0 function code
;         :
         MOV.L    label+2,R0
         JSR       @R0
                                ; ->If the contents of the R4 to R7 registers are
                                ; not destroyed, a correct parameter is passed.
         NOP
;         :
                                ; Restores the PR and R1 to R7 registers
         RTE                    ; ->Returns from the exception processing
                                ; The return value R0 of func is used as
                                ; the return value of trapa_svc
         NOP
label:
        .RES.W  1
        .DATA.L  _func
        .END
```

3.2.12 Prefetch Instruction

Description:

The format for the built-in function to perform cache prefetching in the SH-2A, SH2A-FPU, SH-3, SH3-DSP, SH-4, SH-4A, and SH4AL-DSP is shown below. This built-in function is valid only when "-cpu=sh2a", "-cpu=sh2afpu", "-cpu=sh3" , "-cpu=sh3dsp", "-cpu=sh4" , "-cpu=sh4a" or "-cpu=sh4aldsp" is specified.

- Format:

```
void prefetch(void *p1)
p1 : prefetch address
```

Example of use:

C language code

```
#include <umachine.h>
int a[1200];
f()
{
    int *pa = a;
        :
        :
    prefetch(pa+8);
        :
        :
}
```

Expanded into assembly language code

```
_f:                               ; function: f
            :
            :
        ADD     #32,R6
        PREF    @R6
            :
            :
```

3.2.13 LDTLB Instruction

Description:

The format for the built-in function to perform LDTLB expansion in the SH-3, SH3-DSP, SH-4, SH-4A, and SH4AL-DSP is shown below. This built-in function is valid only when "-cpu=sh3" , "-cpu=sh3dsp", "-cpu=sh4" , "-cpu=sh4a" or "-cpu=sh4aldsp" is specified.

- Format:

```
void ldtlb (void)
```

Example of use:

C language code

```
#include <machine.h>
void main(void)
{
       ldtlb();
}
```

Expanded into assembly language code

```
          .EXPORT     _main
          .SECTION    P,CODE,ALIGN=4
_main:                            ; function: main
                                  ; frame size=0
          LDTLB
          NOP
          NOP
          RTS
          NOP
            .END
```

3.2.14 NOP Instruction

Description:

The format for the built-in function to expand into the NOP instruction is shown below.

- Format:

```
void nop (void)
```

Example of use:

C language code

```
#include <machine.h>
void  main(void)
{
int a;
    if (a){
        nop();
    }
}
```

Expanded into assembly language code

```
            .EXPORT     _main
            .SECTION    P,CODE,ALIGN=4
_main:                              ; function: main
                                  ; frame size=0
            TST         R2,R2
            BT          L12
            NOP
            RTS
            NOP
L12:
            RTS
            NOP
            .END
```

3.2.15 Single-Precision Floating-Point Operations

Description:

Built-in functions for single-precision floating-point operations were added beginning with the SH-4. Table 3.12 lists the operations available. The built-in functions for the floating-point operation unit are valid when -cpu=sh2e, -cpu=sh2afpu, -cpu=sh4, or -cpu=sh4a is specified. The built-in functions for the single-precision floating-point vector operation are valid when -cpu=sh4 or -cpu=sh4a is specified. However, add4() and sub4() are effective also at the time of cpu=sh2afpu specification.

Table 3.12 List of Single-Precision Floating-Point Operations (1)

No.	Function	Format	Description
1	Floating-point operation unit	void set_fpscr(int cr)	Sets cr (32 bits) in the FPSCR.
2		int get_fpscr()	Refers to the FPSCR.
3	Single-precision floating-point vector operation	float fipr(float vect1[4], float vect2[4])	Obtains inner product of two vectors.
4		float ftrv(float vec1[4], float vec2[4])	Transforms vec1(vector) by tbl (4×4 matrix) loaded using ld_ext( ), and saves the result in vec2 (vector).
5		void ftrvadd(float vec1[4], float vec2[4], float vec3[4])	Transforms vec1(vector) by tbl (4×4 matrix) loaded using the ld_ext( ) function, adds the result to vec2 (vector), and saves the addition result in vec3 (vector).
6		void ftrvsub(float vec1[4], float vec2[4], float vec3[4])	Transforms vec1(vector) by tbl (4×4 matrix) loaded using the ld_ext( ) function, subtracts vec2 (vector) from the result, and saves the subtraction result in vec3 (vector).
7		void add4(float vec1[4], float vec2[4], float vec3[4])	Adds vec1 (vector) and vec2 (vector), and saves the result in vec3 (vector).

Table 3.12 List of Single-Precision Floating-Point Operations (2)

No.	Function	Format	Description
8	Single-precision floating-point vector operation	void sub4(float vec1[4], float vec2[4], float vec3[4])	Subtracts vec2 (vector) from vec1(vector), and saves the result in vec3 (vector).
9		void mtrx4mul(float mat1[4][4], float mat2[4][4])	Transforms mat1 (4×4 matrix) by tbl (4×4 matrix) loaded using the ld_ext() function, and saves the result in mat2.
10		void mtrx4muladd(float mat1[4][4], float mat2[4][4], float mat3[4][4])	Transforms mat1 (4×4 matrix) by tbl (4×4 matrix) loaded using theld_ext() function, adds the result and mat2 (4×4 matrix), and saves the addition results in mat3 (4×4 matrix).
11		void mtrx4mulsub(float mat1[4][4], float mat2[4][4], float mat3[4][4])	Transforms mat1 (4×4 matrix) by tbl (4×4 matrix) loaded using the ld_ext() function, subtracts mat2 (4×4 matrix) from the result, and saves the subtraction results in mat3 (4×4 matrix).

Example of use:

Perform multiplication of 4x4 matrices.
One of the matrices must be loaded in advance using the ld_ext function.

C language code

```
#include<machine.h>
float table[4][4] ={{1.0,0.0,0.0,0.0},{0.0,1.0,0.0,0.0},
                    {0.0,0.0,1.0,0.0},{0.0,0.0,0.0,1.0}} ;
float data1[4][4] ={{11.0,12.0,13.0,14.0},{15.0,16.0,17.0,18.0},
                    {11.0,12.0,13.0,14.0},{15.0,16.0,17.0,18.0}} ;
float data2[4][4] ={{0.0,0.0,0.0,0.0},{0.0,0.0,0.0,0.0},
                    {0.0,0.0,0.0,0.0},{0.0,0.0,0.0,0.0}} ;

void main()
{
  ld_ext(table) ;
  mtrx4mul(data1,data2) ;
}
```

Expanded into assembly language code

```
            .EXPORT    _table
            .EXPORT    _data1
```

```
          .EXPORT    _data2
          .EXPORT    _main
          .SECTION    P,CODE,ALIGN=4
_main:                            ; function: main
                                 ; frame size=0
          MOV.L       L11+2,R2    ; _table
          MOV.L       L11+6,R6    ; _data2
          FRCHG
          FMOV.S      @R2+,FR0
          FMOV.S      @R2+,FR1
          FMOV.S      @R2+,FR2
          FMOV.S      @R2+,FR3
          FMOV.S      @R2+,FR4
          FMOV.S      @R2+,FR5
          FMOV.S      @R2+,FR6
          FMOV.S      @R2+,FR7
          FMOV.S      @R2+,FR8
          FMOV.S      @R2+,FR9
          FMOV.S      @R2+,FR10
          FMOV.S      @R2+,FR11
          FMOV.S      @R2+,FR12
          FMOV.S      @R2+,FR13
          FMOV.S      @R2+,FR14
          FMOV.S      @R2+,FR15
          FRCHG
          ADD         #-64,R2
          MOV.L       L11+10,R2    ; _data1
          ADD         #16,R6
          FMOV.S      @R2+,FR0
          FMOV.S      @R2+,FR1
          FMOV.S      @R2+,FR2
          FMOV.S      @R2+,FR3
          FTRV        XMTRX,FV0
          FMOV.S      FR3,@-R6
          FMOV.S      FR2,@-R6
          FMOV.S      FR1,@-R6
          FMOV.S      FR0,@-R6
          FMOV.S      @R2+,FR0
          ADD         #32,R6
          FMOV.S      @R2+,FR1
          FMOV.S      @R2+,FR2
          FMOV.S      @R2+,FR3
          FTRV        XMTRX,FV0
          FMOV.S      FR3,@-R6
          FMOV.S      FR2,@-R6
          FMOV.S      FR1,@-R6
          FMOV.S      FR0,@-R6
          FMOV.S      @R2+,FR0
          ADD         #32,R6
          FMOV.S      @R2+,FR1
          FMOV.S      @R2+,FR2
          FMOV.S      @R2+,FR3
          FTRV        XMTRX,FV0
          FMOV.S      FR3,@-R6
          FMOV.S      FR2,@-R6
          FMOV.S      FR1,@-R6
          FMOV.S      FR0,@-R6
          FMOV.S      @R2+,FR0
          ADD         #32,R6
          FMOV.S      @R2+,FR1
          FMOV.S      @R2+,FR2
          FMOV.S      @R2+,FR3
          FTRV        XMTRX,FV0
          FMOV.S      FR3,@-R6
          FMOV.S      FR2,@-R6
          FMOV.S      FR1,@-R6
          RTS
          FMOV.S      FR0,@-R6
L11:
          .RES.W      1
          .DATA.L    _table
          .DATA.L    _data2
          .DATA.L    _data1
          .SECTION    D,DATA,ALIGN=4
_table:                           ; static: table
          .DATA.L     H'3F800000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'3F800000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'3F800000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'3F800000
```

```
_data1:                          ; static: data1
          .DATA.L     H'41300000
          .DATA.L     H'41400000
          .DATA.L     H'41500000
          .DATA.L     H'41600000
          .DATA.L     H'41700000
          .DATA.L     H'41800000
          .DATA.L     H'41880000
          .DATA.L     H'41900000
          .DATA.L     H'41300000
          .DATA.L     H'41400000
          .DATA.L     H'41500000
          .DATA.L     H'41600000
          .DATA.L     H'41700000
          .DATA.L     H'41800000
          .DATA.L     H'41880000
          .DATA.L     H'41900000
_data2:                          ; static: data2
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .END
```

Example of use:

Perform multiplication of a vector and a matrix. The matrix must be loaded in advance using the ld_ext function.

C language code

```
#include<machine.h>
float table[4][4]={{1.0,2.0,3.0,4.0},{5.0,6.0,7.0,8.0},
                   {8.0,7.0,6.0,5.0},{4.0,3.0,2.0,1.0}} ;
float data1[4]   = {11.0,12.0,13.0,14.0} ;
float data2[4]   = {0.0,0.0,0.0,0.0} ;

void main()
{
  ld_ext(table) ;
  ftrv(data1,data2) ;
}
```

Expanded into assembly language code

```
          .EXPORT     _table
          .EXPORT     _data1
          .EXPORT     _data2
          .EXPORT     _main
          .SECTION    P,CODE,ALIGN=4
_main:                            ; function: main
                                  ; frame size=0
          MOV.L       L11+2,R2    ; _table
          FRCHG
          FMOV.S      @R2+,FR0
          FMOV.S      @R2+,FR1
          FMOV.S      @R2+,FR2
          FMOV.S      @R2+,FR3
          FMOV.S      @R2+,FR4
          FMOV.S      @R2+,FR5
          FMOV.S      @R2+,FR6
          FMOV.S      @R2+,FR7
          FMOV.S      @R2+,FR8
          FMOV.S      @R2+,FR9
          FMOV.S      @R2+,FR10
          FMOV.S      @R2+,FR11
          FMOV.S      @R2+,FR12
          FMOV.S      @R2+,FR13
          FMOV.S      @R2+,FR14
          FMOV.S      @R2+,FR15
          FRCHG
          ADD         #-64,R2
          MOV.L       L11+6,R2    ; _data1
          FMOV.S      @R2+,FR0
          FMOV.S      @R2+,FR1
          FMOV.S      @R2+,FR2
          FMOV.S      @R2+,FR3
          MOV.L       L11+10,R2   ; _data2
          FTRV        XMTRX,FV0
          ADD         #16,R2
          FMOV.S      FR3,@-R2
          FMOV.S      FR2,@-R2
          FMOV.S      FR1,@-R2
          RTS
          FMOV.S      FR0,@-R2
L11:
          .RES.W      1
          .DATA.L     _table
          .DATA.L     _data1
          .DATA.L     _data2
          .SECTION    D,DATA,ALIGN=4
_table:                            ; static: table
          .DATA.L     H'3F800000
          .DATA.L     H'40000000
          .DATA.L     H'40400000
          .DATA.L     H'40800000
          .DATA.L     H'40A00000
          .DATA.L     H'40C00000
          .DATA.L     H'40E00000
          .DATA.L     H'41000000
          .DATA.L     H'41000000
          .DATA.L     H'40E00000
          .DATA.L     H'40C00000
          .DATA.L     H'40A00000
          .DATA.L     H'40800000
          .DATA.L     H'40400000
          .DATA.L     H'40000000
          .DATA.L     H'3F800000
_data1:                            ; static: data1
          .DATA.L     H'41300000
          .DATA.L     H'41400000
          .DATA.L     H'41500000
          .DATA.L     H'41600000
_data2:                            ; static: data2
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .DATA.L     H'00000000
          .END
```

- Obtains inner products of two vectors.

C language code

```
#include<machine.h>
float ret = 0;
float data1[]={1.0,2.0,3.0,4.0} ;
float data2[]={11.0,12.0,13.0,14.0} ;

void main()
{
   ret = fipr (data1,data2) ;
}
```

Expanded into assembly language code

```
          .EXPORT    _ret
          .EXPORT    _data1
          .EXPORT    _data2
          .EXPORT    _main
          .SECTION    P,CODE,ALIGN=4
_main:                           ; function: main
                                 ; frame size=0
          MOV.L       L11,R2     ; _data1
          FMOV.S      @R2+,FR0
          FMOV.S      @R2+,FR1
          FMOV.S      @R2+,FR2
          FMOV.S      @R2+,FR3
          MOV.L       L11+4,R2   ; _data2
          FMOV.S      @R2+,FR4
          FMOV.S      @R2+,FR5
          FMOV.S      @R2+,FR6
          FMOV.S      @R2+,FR7
          MOV.L       L11+8,R2   ; _ret
          FIPR        FV4,FV0
          RTS
          FMOV.S      FR3,@R2
L11:
          .DATA.L    _data1
          .DATA.L    _data2
          .DATA.L    _ret
          .SECTION    D,DATA,ALIGN=4
_ret:                            ; static: ret
          .DATA.L     H'00000000
```

```
_data1:                        ; static: data1
          .DATA.L     H'3F800000
          .DATA.L     H'40000000
          .DATA.L     H'40400000
          .DATA.L     H'40800000
_data2:                        ; static: data2
          .DATA.L     H'41300000
          .DATA.L     H'41400000
          .DATA.L     H'41500000
          .DATA.L     H'41600000
          .END
```

Important Information:

(1) Built-in functions for single-precision floating-point vector operations are valid only in the SH-4 and SH-4A. (add4() and sub4() are effective also at SH2A-FPU)

(2) When built-in functions for vector operations are used in interrupt functions, the following should be noted. The built-in functions ld_ext(float[4][4]) and st_ext(float[4][4]) modify the floating-point register bank bit (FR) of the floating-point status control register (FPSCR) to access the extension register; hence if either of the built-in functions ld_ext(float[4][4]) or st_ext(float[4][4]) is used within an interrupt function, the interrupt mask should be changed before and after use of the built-in vector operation function. An example is given below.

Example:

```
#pragma interrupt (intfunc)
void intfunc(){
     ...
     ld ext();
     ...
}
void normfunc(){
     ...
     int maskdata=get imask();   /*Saves the interrupt mask      */
     set imask(15);              /*Specifies the interrupt mask  */
     ld ext(mat1);
     ftrv(vec1,vec2);
     set imask(maskdata);        /*Restores the interrupt mask   */
    ...
}
```

(3) The built-in functions mtrx4mul, mtrx4muladd, mtrx4mulsub are operations on 4×4 matrices, so that the result for matrix A × matrix B may not coincide with the result for matrix B × matrix A.

Example:

```
extern float matA[][];
extern float matB[][];
int judge(){
     float data1[4][4], data2[4][4];
     set imask(15);
     ld ext(matA);
     mtrx4mul(matB,data1);       /* data1 = matBxmatA      */
     ld ext(matB);
     mtrx4mul(matA,data2);       /* data2 = matAxmatB      */
     ..../*In this case, data1[ ][ ] may not match data2[ ][ ]      */
}
```

3.2.16 Accessing the Extension Register

Description:

Table 3.13 shows the functions provided to access the extension register.

Table 3.13 List of Built-in Functions to Access the Extension Register

No.	Function	Format	Description
1	Extension register access	void ld_ext(float mat[4][4])	Loads tbl (4×4 matrix) to extension register Example: extern float tbl[4][4]; In this case In this case, ld_ext(tbl) loads tbl to extension register.
2		void st_ext(float mat[4][4])	Saves the contents of extension register to tbl (4×4 matrix). Example: extern float tbl[4][4]; In this case In this case, st_ext(tbl) saves the contents of extension register to tbl.

Notes: (1) Built-in functions to access the extension register are valid only in the SH-4 and SH-4A.

(2) When these functions are used within an interrupt function, the interrupt mask must be changed. For details, refer to (2) of Important Information in section 3.2.15, Single-Precision Floating-Point Operations.

3.2.17 DSP Instruction

Description:

Table 3.14 shows the functions provided for the DSP instruction.

Table 3.14 List of Built-in Functions for the DSP Instruction

No.	Function	Format	Description
1	Absolute value	long __fixed pabs_lf (long __fixed data) long __accum pabs_la (long __accum data)	Obtains the absolute value. Correct operation is not guaranteed if the obtained absolute value cannot be represented in the same type as the return value (that is, long __fixed type for pabs_lf() or long __accum type for pabs_la() is).
2	MSB detection	__fixed pdmsb_lf (long __fixed data) __fixed pdmsb_la (long __accum data)	Detects an MSB (obtains the amount of shift required for normalizing data).

No.	Function	Format	Description
3	Arithmetic shift	long __fixed psha_lf (long __fixed data, int count) long __accum psha_la (long __accum data, int count)	Performs an arithmetic shift. The specifiable value for count is from -32 to +32. Specifying a positive value shifts data to the left. Specifying a negative value shifts data to the right, for its absolute value. Correct operation is not guaranteed if a value outside the valid range is specified.
4	Rounding-off error	__accum rndtoa (long __accum data) __fixed rndtof (long __fixed data)	Deals with the rounding-off error.
5	Bit pattern copy	long __fixed long_as_lfixed (long data) long lfixed_as_long (long __fixed data)	Copies a bit pattern from the general register to the DSP register, or vice versa.
6	Modulo addressing setup	void set_circ_x (__X__circ __fixed array[], size_t size) void set_circ_y (__Y__circ __fixed array[], size_t size)	Sets the modulo addressing.
7	Modulo addressing cancellation	void clr_circ(void)	Cancels the modulo addressing (that is, clears the 10th and 11th bits from the right of the SR to zero).

8	CS bit setting (DSR register)	void set_cs
		(unsigned int mode)
			Sets the CS bit.
			mode=0: Carry or borrow mode
			mode=1: Negative-value mode
			mode=2: Zero-value mode
			mode=3: Overflow mode
			mode=4: Signed “larger than” mode
			mode=5: Signed “equal to or larger than” mode

Example of use:

C language code

```
#include <machine.h>
circ __X __fixed input[4] = {0.0r, 0.25r, 0.5r, 0.25r};
_Y __fixed output[8];

void  main(void)
{
      int i;
      set_circ_x(input, sizeof(input)); /* Set the modulo addressing */
      for(i=0; i < 8; ii++){
          output[i] = input[i];
      }
      clr_circ();                 /* Cancel the modulo addressing*/
}
```

Expanded into assembly language code

```
          .EXPORT     _output
          .EXPORT     _input
          .EXPORT     _main
          .SECTION    P,CODE,ALIGN=4
_main:                            ; function: main
                                  ; frame size=0
          MOV.L       L13+4,R5    ; _input
          EXTU.W      R5,R2
          MOV         R5,R6
          ADD         #6,R6
          SHLL16      R6
          ADD         R6,R2
          LDC         R2,MOD
          STC         SR,R2
          MOV.W       L13,R4      ; H'F3FF
          AND         R4,R2
          MOV         R2,R6
          MOV         #4,R2       ; H'00000004
          SHLL8       R2
          OR          R2,R6
          LDC         R6,SR
          MOV         #8,R2       ; H'00000008
          MOV.L       L13+8,R6    ; _output
L11:
          MOVX.W      @R5+,X1
          DT          R2
          PCOPY       X1,A0
          BF/S        L11
          MOVY.W      A0,@R6+
          STC         SR,R2
          AND         R4,R2
          LDC         R2,SR
          RTS
          NOP
L13:
          .DATA.W     H'F3FF
          .RES.W      1
          .DATA.L     _input
          .DATA.L     _output
          .SECTION    $XD,DATA,ALIGN=4
_input:                          ; static: input
          .DATA.W     H'0000,H'2000,H'4000,H'2000
          .SECTION    $YB,DATA,ALIGN=4
_output:                         ; static: output
          .RES.W      8
          .END
```

3.2.18 Sine and Cosine

Description:

The function obtains the approximate sine and cosine from the angles specified for angle, and then sets the results to the area indicated by sinv and cosv.

- Format:

```
void fsca(long angle, float * sinv, float * cosv)
```

Example of use:

<u>C language code</u>

```
#include <machine.h>
long angle = (45<<16)/360;  /* 45 degrees */
float  * sinv;
float  * cosv;
void  main(void)
{
      fsca(angle, sinv, cosv);
}
```

<u>Expanded into assembly language code</u>

```
          .EXPORT     _sinv
          .EXPORT     _cosv
          .EXPORT     _angle
          .EXPORT     _main
          .SECTION    P,CODE,ALIGN=4
_main:                          ; function: main
                                ; frame size=0
          MOV.L       L11+2,R6  ; _angle
          MOV.L       @R6,R2
          MOV.L       L11+6,R6  ; _sinv
          LDS         R2,FPUL
          FSCA        FPUL,DR0
          MOV.L       @R6,R2
          MOV.L       L11+10,R6 ; _cosv
          FMOV.S      FR0,@R2
          MOV.L       @R6,R2
          RTS
          FMOV.S      FR1,@R2
L11:
          .RES.W      1
          .DATA.L     _angle
          .DATA.L     _sinv
          .DATA.L     _cosv
          .SECTION    D,DATA,ALIGN=4
_angle:                          ; static: angle
          .DATA.L     H'00002000
          .SECTION    B,DATA,ALIGN=4
_sinv:                           ; static: sinv
          .RES.L      1
_cosv:                           ; static: cosv
          .RES.L      1
          .END
```

3.2.19 Reciprocal of the Square Root

Description:

This function obtains the approximate reciprocal of the square root.

- Format:

```
float fsrra(float data)
```

Example of use:

C language code

```
#include <machine.h>
float  data;
float result;
void  main(void)
{
       result=fsrra(data);
}
```

Expanded into assembly language code

```
          .EXPORT     _data
          .EXPORT     _result
          .EXPORT     _main
          .SECTION    P,CODE,ALIGN=4
_main:                            ; function: main
                                  ; frame size=0
          MOV.L       L11,R2      ; _data
          FMOV.S      @R2,FR9
          MOV.L       L11+4,R2    ; _result
          FSRRA       FR9
          RTS
          FMOV.S      FR9,@R2
L11:
          .DATA.L     _data
          .DATA.L     _result
          .SECTION    B,DATA,ALIGN=4
_data:                            ; static: data
          .RES.L      1
_result:                          ; static: result
          .RES.L      1
          .END
```

3.2.20 Invalidation of the Instruction Cache

Description:

The function invalidates the instruction cache.

- Format:

```
void icbi(void *p)
```

Example of use:

C language code

```
#include <machine.h>
extern int *p;
void  main(void)
{
   icbi(p);
}
```

Expanded into assembly language code

```
          .IMPORT     _p
          .EXPORT     _main
          .SECTION    P,CODE,ALIGN=4
_main:                            ; function: main
                                  ; frame size=0
          MOV.L       L11+2,R6    ; _p
          MOV.L       @R6,R2
          ICBI        @R2
          RTS
          NOP
L11:
          .RES.W      1
          .DATA.L     _p
          .END
```

3.2.21 Cache Block Operations

Description:

The functions perform operations on a cache block.

- Format:

```
void ocbi(void *p)     Cache block invalidation
void ocbp(void *p)     Cache block purge
void ocbwb(void *p)    Cache block write-back
```

Example of use:

C language code

```
#include <machine.h>
extern  int *p;
void  main(void)
{
    ocbi(p);
}
```

Expanded into assembly language code

```
          .IMPORT     _p
          .EXPORT     _main
          .SECTION    P,CODE,ALIGN=4
_main:                            ; function: main
                                  ; frame size=0
          MOV.L       L11+2,R6    ; _p
          MOV.L       @R6,R2
          OCBI        @R2
          RTS
          NOP
L11:
          .RES.W      1
          .DATA.L     _p
          .END
```

3.2.22 Instruction Cache Prefetch

Description:

The function reads a 32-byte instruction block that begins from the 32-byte boundary into the instruction cache.

- Format:

```
void prefi(void *p)
```

Example of use:

C language code

```
#include <machine.h>
void *pa;
void  main(void)
{
      prefi(pa);
}
```

Expanded into assembly language code

```
          .EXPORT     _pa
          .EXPORT     _main
          .SECTION    P,CODE,ALIGN=4
_main:                            ; function: main
                                  ; frame size=0
          MOV.L       L11+2,R6    ; _pa
          MOV.L       @R6,R2
          PREFI       @R2
          RTS
          NOP
L11:
          .RES.W      1
          .DATA.L     _pa
          .SECTION    B,DATA,ALIGN=4
_pa:                              ; static: pa
          .RES.L      1
          .END
```

3.2.23 System Synchronization

Description:

This function expands an instruction to the SYNCO instruction.

The SYNCO instruction synchronizes data operations. Executing the SYNCO instruction allows the instructions after the SYNCO instruction to be started when the data operation that came before the SYNCO instruction has completed.

- Format:

```
void synco(void)
```

Example of use:

C language code

```
#include <machine.h>
void  main(void)
{
    synco();
}
```

Expanded into assembly language code

```
          .EXPORT     _main
          .SECTION    P,CODE,ALIGN=4
_main:                          ; function: main
                                ; frame size=0
          SYNCO
          RTS
          NOP
          .END
```

3.2.24 Referencing and Setting the T Bit

Description:

Table 3.15 shows the functions provided for setting and referencing the T bit.

Table 3.15 List of Built-in Functions for the T Bit

No.	Function	Format	Description
1	T bit reference	int movt(void)	References the value of the T bit in the SR register.
2	T bit clear	void clrt(void)	Clears the T bit in the SR register.
3	T bit setting	void sett(void)	Sets the T bit in the SR register.

Example of use:

The function allows you to reference the value of the T bit in the SR register. The referenced value will be 0 or 1.

C language code

```
#include <machine.h>
int sr_t;
void func(void)
{
      sr_t = movt();
}
```

Expanded into assembly language code

```
_func:
          MOVT       R2
          MOV.L      L11,R6     ; _sr_t
          RTS
          MOV.L      R2,@R6
```

3.2.25 Cutting Out the Middle of the Contatenated Register

Description:

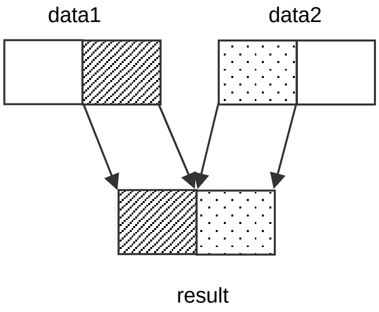
The function cuts out the middle 32 bits from two concatenated 32-bit data items.

- Format:

```
unsigned long xtrct(unsigned long data1, unsigned long data2)
```

Example of use:

In this example, data1 and data2 are concatenated, and then the middle 32 bits are cut out.



C language code

```
#include <machine.h>
unsigned long result, data1, data2;
void main(void)
{
        result = xtrct(data1,data2);
}
```

Expanded into assembly language code

```
_main:
          MOV.L        L11,R1      ; _data2
          MOV.L        L11+4,R4    ; _data1
          MOV.L        @R1,R2
          MOV.L        @R4,R5
          MOV.L        L11+8,R6    ; _result
          XTRCT        R5,R2
          RTS
          MOV.L        R2,@R6
```

3.2.26 Addition with Carry

Description:

Table 3.16 shows the functions provided for addition with carry.

Table 3.16 List of Built-in Functions for Addition with Carry

No.	Function	Format	Description
1	Addition with carry	long addc(long data1, long data2)	Adds two data items and the T bit, and applies the carry to the T bit.
2		int ovf_addc(long data1, long data2)	Adds two data items and the T bit, and references the carry.
3		long addv(long data1, long data2)	Adds two data items, and applies the carry to the T bit.
4		int ovf_addv(long data1, long data2)	Adds two data items, and references the carry.

Example of use:

In this example, data1, data2, and T bit are added, and then the carry is applied to the T bit.

C language code

```
#include <machine.h>
long result, data1, data2;
void main(void)
{
    result = addc(data1,data2);      /* result = data1 + data2 + T bit */
}
```

Expanded into assembly language code

```
_main:
          MOV.L         L11,R1       ; _data1
          MOV.L         L11+4,R2     ; _data2
          MOV.L         @R1,R4
          MOV.L         @R2,R2
          MOV.L         L11+8,R5     ; _result
          ADDC          R4,R2
          RTS
          MOV.L         R2,@R5
```

Important Information:

The addc and ovf_addc functions reference the contents of the T bit. If you specify a comparison or shift immediately before these functions, the calculation results applied to the T bit may cause incorrect operation of the functions.

3.2.27 Subtraction with Borrow

Description:

Table 3.17 shows the functions provided for subtraction with borrow.

Table 3.17 List of Built-in Functions for Subtraction with Borrow

No.	Function	Format	Description
1	Subtraction with borrow	long subc(long data1, long data2)	Subtracts data2 and the T bit from data1, and applies the borrow to the T bit.
2		int unf_subc(long data1, long data2)	Subtracts data2 and the T bit from data1, and references the borrow.
3		long subv(long data1, long data2)	Subtracts data2 from data1, and applies the borrow to the T bit.
4		int unf_subv(long data1, long data2)	Subtracts data2 from data1, and references the borrow.

Example of use:

In this example, data2 and the T bit are subtracted from data1, and then the borrow is applied to the T bit.

<u>C language code</u>

```
#include<machine.h>
long result, data1, data2;
void main(void)
{
    result = subc(data1,data2);     /* result = data1 - data2 - T bit */
}
```

<u>Expanded into assembly language code</u>

```
_main:
          MOV.L        L11,R1       ; _data1
          MOV.L        L11+4,R4     ; _data2
          MOV.L        @R1,R6
          MOV.L        @R4,R5
          MOV.L        L11+8,R1     ; _result
          SUBC         R5,R6
          RTS
          MOV.L        R6,@R1
```

Important Information:

The subc and unf_subc functions reference the contents of the T bit. If you specify a comparison or shift immediately before these functions, the calculation results applied to the T bit may cause incorrect operation of the functions.

3.2.28 Sign Inversion

Description:

The function subtracts the data and the T bit from 0, and applies the borrow to the T bit.

- Format:

```
long negc(long data)
```

Example of use:

In this example, "data" and the T bit is subtracted from 0, and the borrow is applied to the T bit.

C language code

```
#include <machine.h>
long result, data;
void main(void)
{
        result = negc(data);        /* result = 0 - data - T bit */
}
```

Expanded into assembly language code

```
_main:
          MOV.L         L11,R1       ; _data
          MOV.L         L11+4,R5     ; _result
          MOV.L         @R1,R4
          NEGC          R4,R2
          RTS
          MOV.L         R2,@R5
```

3.2.29 One-Bit Division

Description:

Table 3.18 shows the functions provided for one-bit division.

Table 3.18 List of Built-in Functions for One-Bit Division

No.	Function	Format	Description
1	One-bit division	unsigned long div1(unsigned long data1, unsigned long data2)	Performs a one-step division of data1 by data2, and applies the result to the T bit.
2		int div0s(long data1, long data2)	Performs the initialization for the signed division of data1 by data2, and references the T bit.
3		void div0u(void)	Performs the initialization for the unsigned division.

Example of use:

By repeatedly executing one-bit division, you can obtain a quotient. The following shows an example of unsigned division of d1 (32 bits) by d2 (16 bits), where the result is "ret" (16 bits).

C language code

```
#include <machine.h>
unsigned long data1,data2;
unsigned long result;
void main(void)
{
    unsigned long d1,d2;

    d1 = data1;
    d2 = data2;
    d2 <<= 16;                  /* Uses the upper 16 bits as the divisor, and sets the lower 16 bits to 0    */
    div0u();                    /* Sets the initial values for the unsigned division                        */
    d1 = div1(d1, d2);          /* Repeats the one-step division 16 times                                   */
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
    d1 = div1(d1, d2);
```

```
     d1 = div1(d1, d2);
     d1 = div1(d1, d2);
     result = rotcl(d1);           /* rotcl is a built-in function for generating the ROTCL instruction      */
                                   /* Applies the T bit as the result of the last step division to the quotient */
 }
```

Expanded into assembly language code

```
 _main:
           MOV.L     L11,R1       ; _data2
           MOV.L     @R1,R5
           SHLL16    R5
           DIV0U
           MOV.L     L11+4,R2     ; _data1
           MOV.L     @R2,R2
           DIV1      R5,R2
           DIV1      R5,R2
           DIV1      R5,R2
           DIV1      R5,R2
           MOV       R2,R6
           DIV1      R5,R6
           MOV       R6,R2
           DIV1      R5,R2
           DIV1      R5,R2
           DIV1      R5,R2
           DIV1      R5,R2
           DIV1      R5,R2
           DIV1      R5,R2
           MOV       R2,R6
           DIV1      R5,R6
           MOV       R6,R2
           DIV1      R5,R2
           DIV1      R5,R2
           DIV1      R5,R2
           DIV1      R5,R2
           ROTCL     R2
           MOV.L     L11+8,R4     ; _result
           RTS
           MOV.L     R2,@R4
```

Important Information:

(1) Although you can perform division by repeatedly using the div1 function, do not update the M, Q, and T bits during repetition. (A comparison and shift also update the T bit).

(2) You must use div0s() or div0u() immediately before the function to initialize the M, Q, and T bits.

3.2.30 Rotation

Description:

Table 3.19 shows the functions provided for rotation.

Table 3.19 List of Built-in Functions for Rotation

No.	Function	Format	Description
1	Rotation	unsigned long rotl(unsigned long data)	Rotates the data left by one bit, and then applies the bits that moved outside the operand to the T bit.
2		unsigned long rotr(unsigned long data)	Rotates the data right by one bit, and then applies the bits that moved outside the operand to the T bit.
3		unsigned long rotcl(unsigned long data)	Rotates the data left by one bit including the T bit, and then applies the bits that moved outside the operand to the T bit.
4		unsigned long rotcr(unsigned long data)	Rotates the data right by one bit including the T bit, and then applies the bits that moved outside the operand to the T bit.

Example of use:

In this example, "data" is rotated left by one bit, and then the bits that moved outside the operand are applied to the T bit.

C language code

```
#include <machine.h>
unsigned long result, data;
void main(void)
{
    result = rotl(data);
}
```

Expanded into assembly language code

```
_main:
          MOV.L        L11,R2       ; _data
          MOV.L        L11+4,R6     ; _result
          MOV.L        @R2,R2
          ROTL         R2           ; Generates the ROTL instruction
          RTS
          MOV.L        R2,@R6
```

Important Information:

The rotcl and rotcr functions reference the contents of the T bit. If you specify a comparison or shift immediately before these functions, the calculation results applied to the T bit may cause incorrect operation of the functions.

3.2.31 Shift

Description:

Table 3.20 shows the functions provided for a shift.

Table 3.20 List of Built-in Functions for Shift

No.	Function	Format	Description
1	Shift	unsigned long shll(unsigned long data)	Shifts the data to the left by one bit, and then applies the bits that moved outside the operand to the T bit.
2		unsigned long shlr(unsigned long data)	Logically shifts the data to the right by one bit, and then applies the bits that moved outside the operand to the T bit.
3		long shar(long data)	Arithmetically shifts the data to the right by one bit, and then applies the bits that moved outside the operand to the T bit.

Example of use:

In this example, "data" is shifted to the left by one bit, and then the bits that moved outside the operand are applied to the T bit.

C language code

```
#include <machine.h>
unsigned long result, data;
void main(void)
{
    result = shll(data);
}
```

Expanded into assembly language code

```
_main:
          MOV.L       L11,R2     ; _data
          MOV.L       L11+4,R6   ; _result
          MOV.L       @R2,R2
          SHLL        R2         ; Generates the SHLL instruction
          RTS
          MOV.L       R2,@R6
```

3.2.32 Saturation Operation

Description:

Table 3.21 shows the functions provided for saturation operations.

Table 3.21 List of Built-in Functions for Saturation Operations

No.	Function	Format	Description
1	Saturation operation on signed one-byte data	long clipsb(long data)	Sets the value of the data if it falls in the range from -128 to 127. If the data exceeds this range, the function sets the upper or lower limit.
2	Saturation operation on signed two-byte data	long clipsw(long data)	Sets the value of the data if it falls in the range from -32768 to 32767. If the data exceeds this range, the function sets the upper or lower limit.
3	Saturation operation on unsigned one-byte data	unsigned long clipub(unsigned long data)	Sets the value of the data if it falls in the range from 0 to 255. If the data exceeds this range, the function sets the upper limit.
4	Saturation operation on unsigned two-byte data	unsigned long clipuw(unsigned long data)	Sets the value of the data if it falls in the range from 0 to 65535. If the data exceeds this range, the function sets the upper limit.

This built-in function is valid only when "-cpu=sh2a" or "-cpu=sh2afpu" is specified.

Example of use:

In this example, the value of the data is set if it falls in the range from -128 to 127. If the data exceeds this range, the upper or lower limit is set.

C language code

```
#include <machine.h>
long result, data;
void main(void)
{
    result = clipsb(data);           /* The value of "result" is in the range from -128 to 127 */
}
```

Expanded into assembly language code

```
_main:
          MOV.L       L11,R2        ; _data
          MOV.L       @R2,R2
          MOV.L       L11+4,R6      ; _result
          CLIPS.B     R2
          RTS
          MOV.L       R2,@R6
```

3.2.33 Referencing and Setting the TBR

Description:

Table 3.22 shows the functions provided for setting and referencing the jump table base register (TBR).

Table 3.22 List of Built-in Functions for the TBR

No.	Function	Format	Description
1	TBR setting	void set_tbr(void *data)	Sets the data in the TBR.
2	TBR reference	void *get_tbr(void)	References the value of the TBR.

This built-in function is valid only when "-cpu=sh2a" or "-cpu=sh2afpu" is specified.

Example of use:

In this example, data is set in the TBR.
This function is used for setting, in the TBR, the jump table that is generated for a TBR relative function call.

C language code

```
#include <machine.h>
void main(void){
      set_tbr(__sectop("$TBR"));    /* Sets the beginning of the $TBR section to the TBR */
}
```

Expanded into assembly language code

```
_main:
          MOV.L         L11,R2      ; STARTOF $TBR
          RTS
          LDC           R2,TBR
L11:
          .DATA.L       STARTOF $TBR
```

3.3 Inline Expansion

3.3.1 Inline Expansion of Functions

Description:

Inline expansion of functions is used in order to enhance the execution speed of a program. Normally a function is called by branching to a section of code consisting of a series of operations; but this feature expands the processing of the function at the point at which it is called, eliminating the instruction at the branch point and speeding execution. Expansion of functions called within loops can be expected to have an especially great effect in speeding execution. There are two kinds of inline expansion of programs, as follows.

(1) Automatic inline expansion

By specifying the "-speed" option at compilation, the automatic function inline expansion feature is implemented, and small functions are automatically expanded. In order to exert more detailed control over the automatic function inline expansion feature, the "-inline" option can be used to specify the sizes of functions for expansion. Before Ver.6, the sizes of functions are specified in terms of the number of nodes (the number of variables, operators and other elements excluding declarations). (The default value for the "-inline" option is 20.) After Ver.7, the user is able to specify the allowed increase in the program's size doe to the use of inline expansion.

Format:

```
shc -speed [-inline=<numelic value>]…
```

(2) Inline expansion based on directive

Functions for inline expansion are specified using "#pragma inline" directives.

Format:

#pragma inline (< function name > [,< function name >…])

Example of use:

A function called within a loop is expanded inline.

(1) Automatic inline expansion

When the following program is compiled using the "-speed" option, f is expanded inline.

C language code

```
extern int *z;
int  f (int p1, int p2)              /* Function to be expanded     */
{
        if (p1 > p2)
            return p1;
        else if (p1 < p2)
            return p2;
        else
            return 0;
}
void  g (int *x, int *y, int count)
```

```
{
        for ( ; count>0; count--, z++, x++, y++)
              *z = f(*x, *y);
}
```

(2) Inline expansion

The functions f1 and f2, specified using a "#pragma inline" directive, are expanded.

C language code

```
int  v,w,x,y;
#pragma  inline(f1,f2)              /* Specifies the function to be expanded inline   */
int  f1(int a, int b)               /* Function to be expanded                        */
{
        return (a+b)/2;
}
int  f2(int  c, int  d)             /* Function to be expanded                        */
{
        return (c-d)/2;
}
void  g ()
{
        int i;
        for(i=0;i<100;i++){
        if(f1(x,y) == f2(v,w))
        sleep();
        }
}
```

Important Information:

(1) The "#pragma inline" directive should be placed before the function itself.

(2) An external definition is generated for a function specified by a "#pragma inline" directive. Hence when writing a function for inline expansion in a file to be included by multiple files, the function must be declared static.

(3) The following functions cannot be inline expanded.

— Functions with variable parameters

— Functions which refer to the addresses of parameters within the functions

— Functions for which the number and type of real and dummy parameters do not agree

— Functions which are called via addresses

(4) If a cache is installed in other than SH-1, inline expansion may result in a cache miss, so that speed is not improved.

(5) When using this feature, because code is expanded at the point at which the function is called, there is a tendency for the program size to increase. This feature should be used with due consideration paid to the balance between program size and speed of execution.

3.3.2 Inline Expansion of Assembly Language

Description:

There are times when a CPU instruction is not supported in the C language, or when assembly language code will provide enhanced performance over the equivalent code in C. At such times, the code in question can be written in assembly language and combined with the C language program. The SuperH RISC engine C/C++ compiler offers a feature for expansion of inline assembly language code to enable inclusion of inline assembly language code with the C source program.
By writing assembly language code in the same area as a C language function, placing a "#pragma inline_asm " directive before the function, the compiler expands the assembly-language code inline at the point at which it is called.
Interfaces between functions should conform to C/C++ compiler generation rules. The C/C++ compiler generates code which saves parameter values in registers R4 to R7, and places return values in R0. For the SH-2E, SH2A-FPU, SH-4, and SH-4A, set FR0 for the return values of the single-precision floating point operations. For the SH2A-FPU, SH-4, and SH-4A, set DR0 for the return values of the double-precision floating point operations.

- Format:

```
            #pragma inline_asm   (< function name > [,< function name >...])
```

Example of use:

When there is frequent exchange of upper and lower bytes, comprising a performance bottleneck, a byte-swapping function can be written in assembly language and expanded inline.

C language code

```
#pragma inline_asm (swap)              /* Specifies the assembler function to be expanded*/
short swap(short p1)                   /* Describes the function to be improved in assembly language */
{
          EXTU.W       R4,R0           ; clear upper word
          SWAP.B       R0,R2           ; swap with R0 lower word
          CMP/GT       R2,R0           ; if (R2 < R0)
          BT           ?0001           ; then goto ?0001
          NOP                          ;
          MOV          R2,R0           ; return R2
?0001:                                 ; local label   Local label is used as a label.
                                       ;
}
void f (short *x, short *y, int i)
{
      for ( ; i > 0; i--, x++, y++)
         *y = swap(*x);                /* Described in the same format as function call C */
}
```

Expanded into assembly language code (Extracted)

```
_f:
          MOV.L     R14,@-R15
          MOV       R6,R14
          MOV.L     R13,@-R15
          CMP/PL    R14
          MOV.L     R12,@-R15
          MOV       R5,R13
          MOV       R4,R12
          BT        L224
          MOV.L     L225,R3              ; L221
          JMP       @R3
          NOP
L224:
L222:
          MOV.W     @R12,R4
          BRA       L223
          NOP
L225:
          .DATA.L   L221
L223:
          EXTU.W    R4,R0
          SWAP.B    R0,R2
          CMP/GT    R2,R0
          BT        ?0001
          NOP
          MOV       R2,R0
?0001:
          .ALIGN    4
          MOV.W     R0,@R13
          ADD       #-1,R14
          ADD       #2,R12
          ADD       #2,R13
          CMP/PL    R14
          BF        L226
          MOV.L     L227+2,R3          ; L222
          JMP       @R3
          NOP
L226:
L221:
          MOV.L     @R15+,R12
          MOV.L     @R15+,R13
          RTS
          MOV.L     @R15+,R14
L227:
          .RES.W    1
          .DATA.L   L222
```

Important Information:

(1) "#pragma inline_asm " should be specified before the definition of the function itself.

(2) An external definition is generated for a function specified by a "#pragma inline_asm " directive. Hence when writing a function for inline expansion in a file to be included by multiple files, the function must be declared static.

(3) Any labels used in assembly language should be local labels.

(4) When using the registers R8 to R15 (but also including FR12 to FR15 in the cases of SH-2E, and FR12 to FR15, and DR12 to DR14 in the case of SH2A-FPU, SH-4, and SH-4A) in functions written in assembly language, these registers must be saved and restored at the start and at the end of the assembly language codes. For details, refer to section 3.15.1 (2) (c), Rule relating to register.

(5) RTS must not be included at the end of a function in assembly language.

(6) When using this feature, the option specifying the object format "-code=asmcode" should be used at compilation.

(7) Use of this feature imposes limitations on debugging at the C source level.

(8) For details on calling functions between C language programs and assembly language programs, refer to section 3.15.1 (2), Function Calling Interface.

(9) For more information on combining C programs and assembly language programs, refer to section 3.15.1, Issues Related to Assembly Language Programs.

3.3.3 Sample Program with an Inline Assembly Function

Programs which if written in C would be inefficient, and programs which cannot be written in the C language, are normally written in assembly language; but by using inline assembly function, such programs can be written mainly in C but with inline assembly language code.

Advantages to Inline Assembly Functions

Inline assembly functions can be defined as if they were C language functions.
Assembly language instructions can be incorporated directly, without any of the overhead of subroutine calls or returns generally resulting when calling functions written in assembly language.

Disadvantages to Inline Assembly Functions

At compilation, the assembly source program must be output.
Consequently C local variables cannot be referenced during debugging.
(At compilation, if the "-code=asmcode" option is specified, the assembler must then be started. At both C compile time and assembly, by specifying the debug option, step execution at the C source level becomes possible.)

Making Effective Use of Inline Assembly Function

It is recommended that inline assembly functions be used as header files in the following manner.

- Functions are declared as static.
- Only local labels are used.
- No instructions are written causing the assembler to automatically generate a literal pool.
- A RTS (return) instruction is not written at the end of a definition.

Format:

```
                                                  /* Inline function definition   */
                                                  /*  FILE: inlasm.h              */
#pragma inline_asm (rev4b)
static  unsigned long rev4b(unsigned long p)
                                                  /* function declared as static  */
{
      ; comments in definitions denoted by semicolons (;) in assembler
            SWAP.W    R4,R0
            SWAP.B    R0,R0
      ; the RTS instruction is not included at the end
}
#pragma inline_asm(ovf)
static  unsigned long ovf()
{
?LABEL001         ; within inline assembly functions, local labels are used
                  ; local labels:  within 16 characters, beginning with '?'
      MOV         R4,R0
          :
      CMP/EQ      #1,R0
      BT          ?LBABEL001
}
```

```
#pragma inline_asm (ovfadd)
#ifdef  NG INLINEASM
                                                  /* an incorrect inline assembly function definition    */
static  unsigned long ovfadd()
{
          :
      MOV.L     #H'f0000000,R0
      ; this causes the assembler to automatically generate a literal pool
      ; in this case instructions may not be correctly expanded
      ; if a literal pool is generated outside the scope of this function, alignment is disrupted
}
#else
                                                  /* correct inline assembly function definition         */
static  unsigned long ovfadd()
{
             :
      MOV.L     #H'f0000000,R0
         ; a .POOL control instruction must be included within this inline
         ; assembly function
         ; definition in this case the instruction is correctly expanded
      .POOL
         ; this .POOL directive causes a literal pool to be expanded here
         ; the actual code expansion image for this program is as follows
         ;       :
         ;       MOV.L   Lxxx,R0
         ;       BRA     Lyyy
         ;       NOP
         ; Lxxx .DATA.L H'f0000000
         ; Lyyy
}
#end
```

The operation of inline assembly functions introduced here are as follows. To perform 64-bit operations in Ver. 8 or a later version, you can use the long long type and unsigned long long type supported in Ver.8.

- 64-bit addition
- 64-bit subtraction
- 64-bit multiplication
- Bit rotation
- Endian conversions
- Multiply-and-accumulate operations
- Overflow checking

To perform 64-bit operations, the following header file is used.

```
   "longlong.h"

   typedef struct{
        unsigned long H;
        unsigned long L;
      }longlong;
```

(1) 64-bit addition

Because the integer data types in the C language are not 64-bit data, processing in C would be redundant. Hence inline assembly statements to efficiently perform 64-bit operations are presented below.

(i) Addition of 64-bit data

Format: longlong addll(longlong a,longlong b)

Parameters: a: 64-bit data

b: 64-bit data

Returned value: longlong: 64-bit data

Description: Adds a and b, returns the result

```
#include <stdio.h>
#include "longlong.h"
#pragma inline asm (addll)
static longlong addll(longlong a,longlong b)
{
    MOV     @(0,R15),R0         ; Sets the start address of return value structure c
    MOV     @(4,R15),R1         ; Sets the first parameter     (a.H)
    MOV     @(8,R15),R2         ;                              (a.L)
    MOV     @(12,R15),R3        ; Sets the second parameter    (b.H)
    MOV     @(16,R15),R4        ;                              (b.L)
    CLRT                        ; Clear T bit
    ADDC    R4,R2               ; Adds the lower 32 bits of R4 and R2 , and sets the T bit
                                ; according to the carry.
    ADDC    R3,R1               ; Adds the higher 32 bits of R3 and R1 with carry
    MOV     R1,@(0,R0)          ; Sets the return value        (c.H)
    MOV     R2,@(4,R0)          ;                              (c.L)
}
void main(void)
{
       longlong a,b,c;

       a.H=0xeffffffff;
       a.L=0xffffffff;
       b.H=0x10000000;
       b.L=0x00000000;
       c=addll(a,b);
       printf("addll = %8X %08X  \n",c.H,c.L);
}
```

(ii) Addition of 64-bit data (address specification)

Format: void addllp(longlong *pa,longlong *pb,longlong *pc)
Parameters: pa: Address of 64-bit data
pb: Address of 64-bit data
pc: Address of variable for storing result
Returned value: None
Description: Adds pa and pb, returns the result in pc

```
#include <stdio.h>
#include "longlong.h"
#pragma inline asm(addllp)
static void addllp(longlong *pa,longlong *pb,longlong *pc)
{
    MOV       @(0,R5),R0      ; Sets (pa->H) to R0
    MOV       @(4,R5),R1      ; Sets (pa->L) to R1
    MOV       @(0,R6),R2      ; Sets (pb->H) to R2
    MOV       @(4,R6),R3      ; Sets (pb->L) to R3
    CLRT                      ; Clear T bit
    ADDC      R3,R1           ; Adds the lower 32 bits of R3 and R1 , and sets
                              ; the T bit according to the carry.
    ADDC      R2,R0           ; Adds the higher 32 bits of R2 and R0 with carry
    MOV       R0,@(0,R4)      ; Sets R0 to (pc->H)
    MOV       R1,@(4,R4)      ; Sets R1 to (pc->L)
}
void main(void)
{
    longlong a,b,c;
    longlong *pa,*pb,*pc;
    b.H=0x10000000;
    b.L=0x00000000;
    c.H=0xeffffff;
    c.L=0xffffffff;

    pa=&a;
    pb=&b;
    pc=&c;
    addllp(pa,pb,pc);
    printf("addllp = %8x %08x \n",pa->H,pa->L);
}
```

(iii) Addition of 64-bit data (with address specifications)
Format: void addtoll(longlong *pa,longlong b)
Parameters: *pa: address of 64-bit data
b: 64-bit data
Returned value: None
Description: Adds b and the data specified by pa, and returns the result to pa

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm (addtoll)
static void addtoll(longlong *pa,longlong b)
{
    MOV       @(0,R4),R0          ; Sets (pa->H) to R0
    MOV       @(4,R4),R1          ; Sets (pa->L) to R1
    MOV       @(0,R15),R2         ; Sets (b.H) to R2
    MOV       @(4,R15),R3         ; Sets (b.L) to R3
    CLRT                          ; Clear T bit
    ADDC      R3,R1               ;  Adds R3 (b.L) and R1 (pa->L ) , and sets
                                  ; the T bit according to the carry.
    ADDC      R2,R0               ;  Adds R2 (b.H) and R0 (pa->H) with carry
    MOV       R0,@(0,R4)          ; Sets R0 to (pa->H)
    MOV       R1,@(4,R4)          ; Sets R1 to (pa->L)
}
void main(void)
{
    longlong *pa,b,c;

    b.H=0x10000000;
    b.L=0x00000000;
    c.H=0xefffffff;
    c.L=0xffffffff;

    pa=&c;
    addtoll(pa,b);
    printf("addtoll = %8x %08x \n",pa->H,pa->L);
}
```

(iv) Addition of 64-bit data and 32-bit data

Format: void addtoll32(longlong *pa,long b)

Parameters: *pa: address of 64-bit data
b: 32-bit data

Returned value: None

Description: Adds b and the data specified by pa, and returns the result to the address specified by pa

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm(addtoll32)
static void addtoll32(longlong *pa,long b)
{
    MOV       @(0,R4),R0       ; Sets (pa->H) to R0
    MOV       @(4,R4),R1       ; Sets (pa->L) to R1
    CLRT                       ; Clear T bit
    ADDC      R5,R1            ; Adds R1(pa->L) and R5 (b), and sets
                               ; the T bit according to the carry
    MOVT     R3             ;Sets a carry in R3
    ADD      R3,R0          ;Adds R0 (pa->H) and R3 (carry)
    MOV      R0,@(0,R4)     ;Sets R0 to (pa->H)
    MOV      R1,@(4,R4)     ;Sets R1 to (pa->L)
}

void main(void)
{
    longlong *pa,c;
    long b;
    b=0x00000001;
    c.H=0xeffffff;
    c.L=0xffffffff;
    pa=&c;
    addtoll32(pa,b);
    printf("addlltoll32 = %8x %08x  \n",pa->H,pa->L);
}
```

(2) 64-bit subtraction

(i) Subtraction of 64-bit data

Format:	longlong subll(longlong a,longlong b)
Parameters:	a: 64-bit data
	b: 64-bit data
Returned value:	longlong: 64-bit data
Description:	Subtract b from a, and returns the result

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm(subll)
static longlong subll(longlong a,longlong b)
{
    MOV      @(0,R15),R0     ; Sets the return value address to R0
    MOV      @(4,R15),R1     ;Sets (a.H) to R1
    MOV      @(8,R15),R2     ;Sets (a.L) to R2
    MOV      @(12,R15),R3    ;Sets (b.H) to R3
    MOV      @(16,R15),R4    ;Sets (b.L) to R4
    CLRT                     ;Clear T bit
    SUBC     R4,R2           ;Subtract R4 (b.L) from R2 (a.L) ,
                             ;and sets the T bit according to the borrow
    SUBC     R3,R1           ;Subtract R3 (b.H) from R1 (a.H) with borrow
    MOV      R1,@(0,R0)      ;Sets R1 to (c.H)
    MOV      R2,@(4,R0)      ;Sets R2 to (c.L)
}
void main(void)
{
    longlong a,b,c;
```

```
        a.H=0xffffffff;
        a.L=0xffffffff;
        b.H=0xffffffff;
        b.L=0xffffffff;
        c=subll(a,b);
        printf("subll = %x %08x  \n",c.H,c.L);
    }
```

(ii) Subtraction of 64-bit data (with address specification)

Format: void subtoll(longlong *pa,longlong b)
Parameters: *pa:address of 64-bit data
b : 64-bit data
Returned value: None
Description: Subtracts b from the data specified by pa, and returns the result to pa

```
    #include <stdio.h>
    #include "longlong.h"
    #pragma inline asm(subtoll)
    static void subtoll(longlong *pa,longlong b)
    {
        MOV      @(0,R4),R0         ;Sets (pa->H) to R0
        MOV      @(4,R4),R1         ;Sets (pa->L) to R1
        MOV      @(0,R15),R2        ;Sets (b.H) to R2
        MOV      @(4,R15),R3        ;Sets (b.L) to R3
        CLRT                        ;Clear T bit
        SUBC     R3,R1              ; Subtract R3 (b.L) from R1 (pa.L),
                                    ;and sets the T bit according to the borrow
        SUBC     R2,R0              ; Subtract R2 (b.H) from R0 (pa.H) with borrow
        MOV      R0,@(0,R4)         ;Sets R0 to (pa->H)
        MOV      R1,@(4,R4)         ;Sets R1 to (pa->L)
    }
    void main(void)
    {
        longlong *pa,b,c;
        b.H=0xffffffff;
        b.L=0xffffffff;
        c.H=0xffffffff;
        c.L=0xffffffff;
        pa=&c;
        subtoll(pa,b);
        printf("addtoll = %8x %08x \n",pa->H,pa->L);
    }
```

(iii) Subtraction of 32-bit data from 64-bit data

Format: void subtoll32(longlong *pa,long b)
Parameters: *pa: Address of 64-bit data
b: 32-bit data
Returned value: None
Description: Subtracts b from the data specified by pa, and returns the result to pa

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm(subtoll32)
static void subtoll32(longlong *pa,long b)
{
    MOV     @(0,R4),R0      ;Sets (pa->H) to R0
    MOV     @(4,R4),R1      ;Sets (pa->L) to R1
    CLRT                    ;Clear T bit
    SUBC     R5,R1          ; Subtracts R5 (b) from R1 (pa->L),
                            ;and sets the T bit according to the borrow
    MOVT    R3              ;Sets a borrow to R3
    SUB     R3,R0           ;Subtracts R3 (borrow) from R0 (pa->H)
    MOV     R0,@(0,R4)      ;Sets R0 to (pa->H)
    MOV     R1,@(4,R4)      ;Sets R1 to (pa->L)
}
void main(void)
{
    longlong *pa,c;
    unsigned long b;
    pa=&c;

    c.H=0xf0000000;
    c.L=0x00000000;

    b=0x00000001;

    subtoll32(pa,b);
    printf("subll = %8x %08x  \n",pa->H,pa->L);
}
```

(3) 64-bit multiplication

(i) Multiplication of 64-bit data

Format: longlong mulll(longlong a,longlong b)
Parameters: a: 64-bit data
b: 64-bit data
Returned value: longlong: 64-bit data
Description: Multiplies a and b, and returns the result

```
#include <stdio.h>
#include "longlong.h”
```

```
#pragma inline_asm(mulll)
static longlong mulll(longlong a,longlong b)
{
    MOV     @(4,R15),R0         ;Sets (a.H) to R0
    MOV     @(8,R15),R1         ;Sets (a.L) to R1
    MOV     @(12,R15),R2        ;Sets (b.H) to R2
    MOV     @(16,R15),R3        ;Sets (b.L) to R3
    MUL.L   R0,R3               ;Multiplies R0 (a.H) with R3 (b.L)
    STS     MACL,R0             ;Substitutes the result (lower 32 bits)
    MUL.L   R2,R1               ;Multiplies R1 (a.L) with R2 (b.H)
    STS     MACL,R2             ;Substitutes the result (lower 32 bits)
    ADD     R2,R0               ;
    DMULU   R1,R3               ;Multiplies R1 (a.L) with R3 (b.L)
    STS     MACH,R1             ;Substitutes the result (higher 32 bits)
    STS     MACL,R3             ;Substitutes the result (lower 32 bits)
    ADD     R1,R0               ;
    MOV     @(0,R15),R4         ;
    MOV     R0,@(0,R4)          ;Sets R0 to (c.H)
    MOV     R3,@(4,R4)          ;Sets R3 to (c.L)
}
void main(void)
{
    longlong a,b,c;
    a.H=0x7fffffff;
    a.L=0xffffffff;
    b.H=0x00000000;
    b.L=0x00000002;

    c=mulll(a,b);
    printf("mulll = %8x %08x  \n",c.H,c.L);
}
```

(ii) Multiplication of 64-bit data (address specified)

Format: void multoll(longlong *pa,longlong b)
Parameters: pa: :address of 64-bit data
b: 64-bit data
Returned value: None
Description: Multiplies b and the data specified by pa, and returns the result to the address specified by pa

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm(multoll)
static void multoll(longlong *pa,longlong b)
{
    MOV     @(0,R4),R0        ;Sets (pa->H) to R0
    MOV     @(4,R4),R5        ;Sets (pa->L) to R5
    MOV      @(4,R15),R1      ; Sets (b.L) to R1
    MUL      R0,R1            ; Multiplies R0 (pa->H) with R1 (b.L)
    STS      MACL,R3          ;
    DMULU    R5,R1            ; Multiplies R5 (pa->L) with R1 (b.L)
    STS      MACH,R0          ; Substitutes the result (higher 32 bits)
    STS      MACL,R1          ; Substitutes the result (lower 32 bits)
    ADD      R3,R0            ;
    MOV      R0,@(0,R4)       ; Sets R0 to (pa->H)
    MOV      R1,@(4,R4)       ; Sets R1 to (pa->L)

}
void main(void)
{
    longlong *pa,b,c;
    c.H=0x0000ffff;
    c.L=0xffff0000;
    b.H=0x00000000;
    b.L=0x00010000;

    pa=&c;
    multoll(pa,b);
    printf("multoll = %8x %08x \n",pa->H,pa->L);
}
```

(iii) Multiplication of 64-bit data and unsigned 32-bit data

Format: void multoll32(longlong *pa,unsigned long b)
Parameters: *pa: :address of 64-bit data
b:unsigned 32-bit data
Returned value: None
Description: Multiplies b and the data specified by pa, and returns the result to the address specified by pa

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm (multoll32)
static void multoll32(longlong *pa,unsigned long b)
{
    MOV      @(0,R4),R0       ; Sets (pa->H) to R0
    MOV      @(4,R4),R1       ; Sets (pa->L) to R1
    ADDC     R5,R1            ; Adds R1(pa->L) and R5 (b),
                              : and sets the T bit according to the carry
    MOVT     R3               ; Sets a carry in R3
    ADD      R3,R0            ;  Adds R0 (pa->H) and R3 (carry)
    MOV      R0,@(0,R4)       ; Sets R0 to (pa->H)
    MOV      R1,@(4,R4)       ; Sets R1 to (pa->L)
void main(void)
{
    longlong *pa,c;
    unsigned long b;

    b=0xffffff00;
    c.H=0x00000000;
    c.L=0x00000100;
    pa=&c;
    multoll32(pa,b);
    printf("mulltoll32 = %8x %08x \n",pa->H,pa->L);
}
```

(iv) Multiplication of unsigned 32-bit data

Format: longlong mul64(unsigned long a,unsigned long b)
Parameters: a: unsigned 32-bit data
b: unsigned 32-bit data
Returned value: longlong: 64-bit data
Description: Multiplies a and b, and returns the result

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm(mul64)
static longlong mul64(unsigned long a,unsigned long b)
{
    MOV    @(0,R15),R0  ;Sets address of c to R0
    DMULU  R4,R5        ;Multiply R4 (a) with R5 (b)
    STS    MACH,R1      ;Substitutes the result (higher 32 bits)
    MOV    R1,@(0,R0)   ;Sets R1 to (c.H)
    STS    MACL,R2      ;Substitutes the result (lower 32 bits)
    MOV    R2,@(4,R0)   ;Sets R2 to (c.L)
}
void main(void)
{
    longlong c;
    unsigned long a,b;

a=0xffffffff;
    b=0x10000000;
    c=mul64(a,b);
    printf("mul64 = %8x %08x  \n",c.H,c.L);
}
```

(v) Multiplication of signed 32-bit data

Format: longlong mul64s(signed long a,signed long b)
Parameters: a: 32-bit data
b: 32-bit data

Returned value: longlong: 64-bit data
Description: Multiplies a and b, and returns the result

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm(mul64s)
static longlong mul64s(signed long a,signed long b)
{
    MOV    @(0,R15),R0    ; Sets address of c to R0
    DMULS  R4,R5          ; Multiply R4 (a) with R5 (b) with sign
    STS    MACH,R1        ; Substitutes the result (higher 32 bits)
    MOV    R1,@(0,R0)     ; Sets R1 to (c.H)
    STS    MACL,R2        ; Substitutes the result (lower 32 bits)
    MOV    R2,@(4,R0)     ; Sets R2 to (c.L)
}
void main(void)
{
    longlong c;
    signed long a,b;
    a = -1;
    b=1;
    c=mul64s(a,b);
    printf("mul64s = %8x %08x \n",c.H,c.L);
}
```

(4) Bit rotation

(i) Rotate 8 bits of data one bit to the left

Format: short rot8l(unsigned long a)
Parameters: a: unsigned 8-bit data
Returned value: short: 8-bit data
Description: Rotate a one bit to the left, and return the result

```
#include <stdio.h>
#pragma inline_asm(rot8l)
unsigned char rot8l(unsigned char a)
{
    ROTL    R4          ; Shifts left by 1 bit
    MOV     R4,R0       ;
    SHLR8   R0          ; Shifts right by 8 bit
    OR      R4,R0       ;
}
void main(void)
{
    unsigned char a;

   a=0x12;
```

```
    a=rot8l(a);
    printf(" rot8l %x \n",a);

}
```

(ii) Rotate 8 bits of data n bits to the left

Format: short rot8ln(unsigned char a,int n)
Parameters: a: unsigned 8-bit data
n: number of shifts
Returned value: short: 8-bit data
Description: Rotate a n bits to the left, and return the result

```
#include <stdio.h>
#pragma inline_asm(rot8ln)
unsigned char rot8ln(unsigned char a,int n)
{
    MOV    #0,R1      ;Sets the counter register
?LOOP:
    ROTL   R4         ;Shifts left by 1 bit
    MOV    R4,R2      ;
    SHLR8  R2         ;Shifts right by 8 bit
    ADD    #1,R1      ;Increments the counter register by 1
    CMP/EQ R1,R5      ;If R1==R5, sets T to 1
    BF     ?LOOP      ;If T!=1, branches to LOOP
    OR     R2,R4      ;Executed before branch
    MOV    R4,R0      ;Sets the return value
}
void main(void)
{
    unsigned char a,b;
    int n;

    a=0x12;
    n=4;
    b=rot8ln(a,n);
    printf("b: %x \n",b);
}
```

(iii) Rotate 8 bits of data one bit to the right

Format: short rot8r(unsigned char a)
Parameters: a: unsigned 8-bit data
Returned value: short: 8-bit data
Description: Rotate a one bit to the right, and return the result

```
#pragma inline_asm(rot8r)
unsigned char rot8r(unsigned char a)
```

```
{
    ROTR     R4             ;Shifts right by 1 bit
    MOV      R4,R0          ;
    SHLR16   R4             ;Shifts right by 16 bit
    SHLR8    R4             ;Shifts right by 8 bit
    OR       R4,R0          ;
}
void main(void)
{
    unsigned char a;

    a=0x12;
    a=rot8r(a);
    printf(" rot8r %x \n",a);

}
```

(iv) Rotate 8 bits of data n bits to the right

Format:	short rot8rn(unsigned char a,int n)
Parameters:	a: unsigned 8-bit data
	n: number of shifts
Returned value:	short: 8-bit data
Description:	Rotate a n bits to the right, and return the result

```
#include <stdio.h>
#pragma inline_asm(rot8rn)
unsigned char rot8rn(unsigned char a,int n)
{
    MOV      #0,R1           Sets the counter register
?LOOP:
    ROTR     R4              ;Shifts right by 1 bit
    MOV      R4,R2           ;
    SHLR16   R2              ;Shifts right by 16 bit
    SHLR8    R2              ;Shifts right by 8 bit
    ADD      #1,R1           ;Increments the counter register by 1
    CMP/EQ   R1,R5           ;If R1==R5, sets T to 1
    BF       ?LOOP           ;If T!=1, branches to LOOP
    OR       R2,R4           ;Executed before branch
    MOV      R4,R0           ;Sets the return value
}
void main(void)
{
    unsigned char a,b;
    int n;

    a=0x12;
    n=4;
    b=rot8rn(a,n);
    printf(" rot8rn %x \n",b);
}
```

(v) Rotate 16 bits of data one bit to the left

Format: short rot16l(unsigned short a)
Parameters: a: unsigned 16-bit data address
Returned value: short: 16-bit data
Description: Rotate a one bit to the left, and return the result

```
#pragma inline_asm(rot16l)
unsigned short rot16l(unsigned short a)
{
    ROTL    R4          ;Shifts left by 1 bit
    MOV     R4,R0       ;
    SHLR16  R0          ;Shifts right by 16 bit
    OR      R4,R0       ;
}
void main(void)
{
    unsigned short a,b;
    a=0x1234;
    b=rot16l(a);
    printf(" rot16l = %x \n",b);
}
```

(vi) Rotate 16 bits of data n bits to the left

Format: short rot16ln(unsigned short a,int n)
Parameters: a: unsigned 16-bit data address
n: number of shifts
Returned value: short: 16-bit data
Description: Rotate a n bits to the left, and return the result

```
#include <stdio.h>
#pragma inline_asm(rot16ln)
unsigned short rot16ln(unsigned short a,int n)
{
    MOV     #0,R1       ;Sets the counter register
?LOOP:
    ROTL    R4          ;Shifts left by 1 bit
    MOV     R4,R2       ;
    SHLR16  R2          ;Shifts right by 16 bit
    ADD     #1,R1       ;Increments the counter register by 1
    CMP/EQ  R1,R5       ;If R1==R5, sets T to 1
    BF      ?LOOP       ;If T!=1, branches to LOOP
    OR      R2,R4       ;
    MOV     R4,R0       ;Sets the return value
}
void main(void)
{
    unsigned short a,b;
    int n;

    a=0x1234;
    n=8;
    b=rot16ln(a,n);
    printf("rot16ln = %x \n",b);
}
```

(vii) Rotate 16 bits of data one bit to the right

Format:	short rot16r(unsigned short a)
Parameters:	a: unsigned 16-bit data address
Returned value:	short: 16-bit data
Description:	Rotate a one bit to the right, and return the result

```
#include <stdio.h>
#pragma inline_asm(rot16r)
unsigned short rot16r(unsigned short a)
{
    ROTR    R4          ;Shifts right by 1 bit
    MOV     R4,R0       ;
    SHLR16  R0          ;Shifts right by 16 bit
    OR      R4,R0       ;
}
void main(void)
{
    unsigned short a,b;
    a=0x1234;
    b=rot16r(a);
    printf("rot16r = %x \n",b);
}
```

(viii) Rotate 16 bits of data n bits to the right

Format:	short rot16rn(unsigned short a,int n)
Parameters:	a: unsigned 16-bit data address n: number of shifts
Returned value:	short: 16-bit data
Description:	Rotate a n bits to the right, and return the result

```
#include <stdio.h>
#pragma inline_asm(rot16rn)
```

```
unsigned short rot16rn(unsigned short a,int n)
{
    MOV     #0,R1       ;Sets the counter register
?LOOP:
    ROTR    R4          ;Shifts right by 1 bit
    MOV     R4,R2       ;
    SHLR16  R2          ;Shifts right by 16 bit
    ADD     #1,R1       ;Increments the counter register by 1
    CMP/EQ  R1,R5       ;If R1==R5, sets T to 1
    BF      ?LOOP       ;If T!=1, branches to LOOP
    OR      R2,R4       ;
    MOV     R4,R0       ;Sets the return value
}
void main(void)
{
    unsigned short a,b;
    int n;

    a=0x1234;
    n=8;
    b=rot16rn(a,n);
    printf("rot16rn %x \n",b);
}
```

(ix) Rotate 32 bits of data one bit to the left

Format:	short rot32l(unsigned long a)
Parameters:	a: unsigned 32-bit data address
Returned value:	short: 32-bit data
Description:	Rotate a one bit to the left, and return the result

```
#include <stdio.h>
#pragma inline_asm(rot32l)
unsigned long rot32l(unsigned long a)
{
    ROTL    R4          ;Shifts left by 1 bit
    MOV     R4,R0       ;Sets the return value
}
void main(void)
{
    unsigned long a;
    a=0x12345678;
    a=rot32l(a);
    printf(" rot32l %8x \n",a);
}
```

(x) Rotate 32 bits of data n bits to the left

Format: short rot32ln(unsigned long a,int b)
Parameters: a: unsigned 32-bit data address
b: number of shifts
Returned value: short: 32-bit data
Description: Rotate a n bits to the left, and return the result

```
#include <stdio.h>
#pragma inline_asm(rot32ln)
unsigned long rot32ln(unsigned long a,int b)
{
    MOV     #0,R1           ; Sets the counter register
?LOOP:
    ROTL    R4              ; Shifts right by 1 bit
    ADD     #1,R1           ; Increments the counter register by 1
    CMP/EQ  R1,R5           ; If R1==R5, sets T to 1
    BF      ?LOOP           ; If T!=1, branches to LOOP
    MOV     R4,R0           ; Sets the return value
}
void main(void)
{
    unsigned long a;
    int    b;
    a=0x12345678;
    b=16;
    a=rot32ln(a,b);
    printf(" rot32ln %8x \n",a);
}
```

(xi) Rotate 32 bits of data one bit to the right

Format: short rot32r(unsigned long a)
Parameters: a: unsigned 32-bit data address
Returned value: short: 32-bit data
Description: Rotate a one bit to the right, and return the result

```
#include <stdio.h>
#pragma inline_asm(rot32r)
unsigned long rot32r(unsigned long a)
{
    ROTR    R4             ; Shifts right by 1 bit
    MOV     R4,R0          ; Sets the return value
}
void main(void)
{
    unsigned long a,b;
    a=0x12345678;
    b=rot32r(a);
    printf(" rot32r %8x \n",b);
}
```

(xii) Rotate 32 bits of data n bits to the right

Format: short rot32rn(unsigned long a,int b)
Parameters: a: unsigned 32-bit data address
b: number of shifts
Returned value: short: 32-bit data
Description: Rotate a n bits to the right, and return the result

```
#include <stdio.h>
#pragma inline_asm(rot32rn)
unsigned long rot32rn(unsigned long a,int b)
{
    MOV     #0,R1           ;Sets the counter register
?LOOP:
    ROTR    R4              ;Shifts right by 1 bit
    ADD     #1,R1           ;Increments the counter register by 1
    CMP/EQ  R1,R5           ;If R1==R5, sets T to 1
    BF      ?LOOP           ;If T!=1, branches to LOOP
    MOV     R4,R0           ;Sets the return value
}
void main(void)
{
    unsigned long a;
    int    b;
    a=0x12345678;
    b=16;
    a=rot32rn(a,b);
    printf("rot32rn %8x \n",b);
}
```

(5) Endian conversions

(i) Swap the upper 16 bits and lower 16 bits

Format: unsigned long swap(unsigned long a)
Parameters: a: unsigned 32-bit data
Returned value: unsigned long: unsigned 32-bit data
Description: Swap the upper and lower 16 bits of a

```
#include <stdio.h>
#pragma inline_asm  (swap)
static unsigned long swap(unsigned long a)
{
    SWAP.W  R4,R0    ;Swaps the upper 16 bits of R4 and lower 16 bits of R0
}
void main(void)
{
    unsigned long a,b;
    a=0xaaaabbbb;
    b=swap(a);
    printf("b: %8x \n",b);
}
```

(ii) Symmetric swap of the upper 16 bits and lower 16 bits

Format: unsigned long swapbit(unsigned long a)
Parameters: a: unsigned 32-bit data
Returned value: unsigned long: unsigned 32-bit data
Description: Individually swap the upper and lower 16 bits of a

32bit → 1bit , 1bit → 32bit
31bit → 2bit , 2bit → 31bit
:
:
18bit → 15bit , 15bit → 18bit
17bit → 16bit , 16bit → 17bit

```
#include <stdio.h>

#pragma inline_asm (swapbit)
static unsigned long swapbit(unsigned long a)
{
    MOV     #0,R0         ;Sets the counter register
?LOOP:
    ROTCL   R4            ;Rotates left
    ROTCR   R1            ;Rotates right
    ADD     #1,R0         ;Increments the counter register by 1
    CMP/EQ  #32,R0        ;If 32==R0, sets T to 1
    BF     ?LOOP          ;If T!=1, branches to LOOP
    NOP                   ;
    MOV     R1,R0         ;Sets the return value

}
void main(void)
{
    unsigned long a,b;

    a=0x1234;
    b=swapbit(a);

    printf("b: %8x \n",b);

}
```

(iii) Endian conversion

Format: unsigned long swapbyte(unsigned long a)
Parameters: a: unsigned 32-bit data
Returned value: unsigned long: unsigned 32-bit data
Description: Endian-convert a and return the result

```
#include <stdio.h>
#pragma inline_asm (swapbyte)
static unsigned long swapbyte(unsigned long a)
{

    SWAP.B  R4,R4       ; Swaps data in bits 0 to 7 of R4 and data in bits 8 to 15 of R4
    SWAP.W  R4,R4       ; Swaps upper 16 bits of R4 and lower 16 bits of R4
    SWAP.B  R4,R0       ; Swaps data in bits 0 to 7 of R4 and data in bits 8 to 15 of R0

}
void main(void)
{
    unsigned long a,b;
    a=0xaabbccdd;
    b=swapbyte(a);
    printf("b: %8x \n",b);
}
```

(6) Multiply-and-accumulate operations

(i) Multiply-and-accumulate operation on arrays of 32-bit data

Format: long macl32h(long *pa,long *pb,int size)
Parameters: *pa: Start address of 32-bit data array
*pb: Start address of 32-bit data array
size: array size
Returned value: long: 32-bit data
Description: Perform multiply-and-accumulate on the data arrays *pa and *pb; return the upper 32 bits of the 64-bit data result

```
#include <stdio.h>
#pragma inline_asm (macl32h)
static long macl32h(long *pa,long *pb,int size)
{
    MOV     #0,R1          ; Sets the counter register
    CLRMAC                 ; Initializes the MAC
?LOOP:
    MAC.L   @R4+,@R5+      ; Multiplies and accumulates
    ADD     #1,R1          ; Increments the counter register by 1
    CMP/EQ  R1,R6          ; If R1==R6, sets T to 1
    BF      ?LOOP          ; If T!=1, branches to LOOP
    NOP                    ;
    STS     MACH,R0        ;Substitutes the result
}
void main(void)
{
    int size=3;
    long c;
    long pa[3]={0x0000f000,0x000f0000,0x00f00000};
    long pb[3]={0x00000100,0x00001000,0x00010000};

    c=macl32h(pa,pb,size);
    printf("macl32h = %8x \n",c);
}
```

(ii) Multiply-and-accumulate operation on arrays of unsigned data

Format: longlong macl64(long *pa,long *pb,int size)
Parameters: *pa: Start address of 32-bit data array
*pb: Start address of 32-bit data array
size: array size
Returned value: longlong: 64-bit data
Description: Perform multiply-and-accumulate on the 32-bit data arrays *pa and *pb and saves the result.

```
#include <stdio.h>
#include "longlong.h"
#pragma inline_asm (macl64)
static longlong macl64(long *pa,long *pb,int size)
{
    MOV     #0,R0            ;Sets the counter register
    MOV     @(0,R15),R1      ;Sets the address of first parameter
    CLRMAC                   ;Initializes the MAC register
?LOOP:
    MAC.L   @R4+,@R5+        ;Performs multiplication and accumulation operation, and modifies address
    ADD     #1,R0            ;Increments the counter register by 1
    CMP/EQ  R0,R6            ;If R0==R6, sets T to 1
    BF      ?LOOP            ;If T!=1, branches to LOOP
    NOP                      ;
    STS     MACH,R2          ;Sets the upper 32 bits of the result
    MOV     R2,@(0,R1)       ;
    STS     MACL,R3          ;Sets the lower 32 bits of the result
    MOV     R3,@(4,R1)       ;
}
void main(void)
{
    longlong c;
    int size=3;
    long *pa,*pb;
    long pa[3]={0x0000f000,0x000f0000,0x00f00000};
    long pb[3]={0x00000100,0x00001000,0x00010000};
    c=macl64(pa,pb,size);
    printf("macl64 = %8X %08X  \n",c.H,c.L);
}
```

(7) Overflow check

(i) Overflow check of 32-bit data addition

Format: long addovf(long a,long b)
Parameters: a: 32-bit data for addition
b: 32-bit data for addition
Returned value: long: 32-bit data
Description: Add a and b, return the result.

If an overflow results, return the maximum value (7FFFFFFF).
If an underflow results, return the minimum value (80000000).
Judgment is based on a change in the sign bit.

```
#include <stdio.h>
#pragma inline_asm (addovf)
static long addovf(long a,long b)
{
    ADDV    R4,R5           ; Performs addition with sign,
                            ; and sets the T bit according to the change of the sign bit
    BF      ?RETURN         ; If T==0, branches to OVER
    MOV     #0,R1           ;
    CMP/GT  R4,R1           ; If R1>R4, sets the T bit
    BF      ?OVER           ; If T==0, branches to OVER
    NOP                     ;
    MOV.L   ?DATA+4,R5      ; Sets R5 to (H'7FFFFFFF)
    BRA     ?RETURN         ; Branches to RETURN
    NOP                     ;
?OVER:
    MOV.L   ?DATA,R5        ; Sets R5 to (H'80000000)
?RETURN:
    MOV     R5,R0           ; Sets R5 to R0
    BRA     ?OWARI          ; Branches to OWARI
    NOP                     ;
?DATA:
    .ALIGN  4
    .RES.L   1
    .DATA.L  H'7FFFFFFF
    .DATA.L  H'80000000
?OWARI:

}
void main(void)
```

```
{
    long a,b,c;
    a=0x3000000;
    b=0x2000000;
    c=addovf(a,b);
    printf("c: %x \n",c);
}
```

3.4 Register Specification

In some cases there is a need to improve the speed of execution of a module which frequently accesses external variables. In such cases, the feature for specifying global base variables (GBR) is used, in which the global base register (GBR) is employed in relative addressing mode to reference the frequently accessed data. GBR-referenced variables are allocated to the sections $G0, $G1 and referenced using the offset from the start address of the $G0 section saved in GBR. This results in code that is faster and more compact than code which loads addresses for referencing, thereby increasing both speed of execution and efficiency of ROM use.

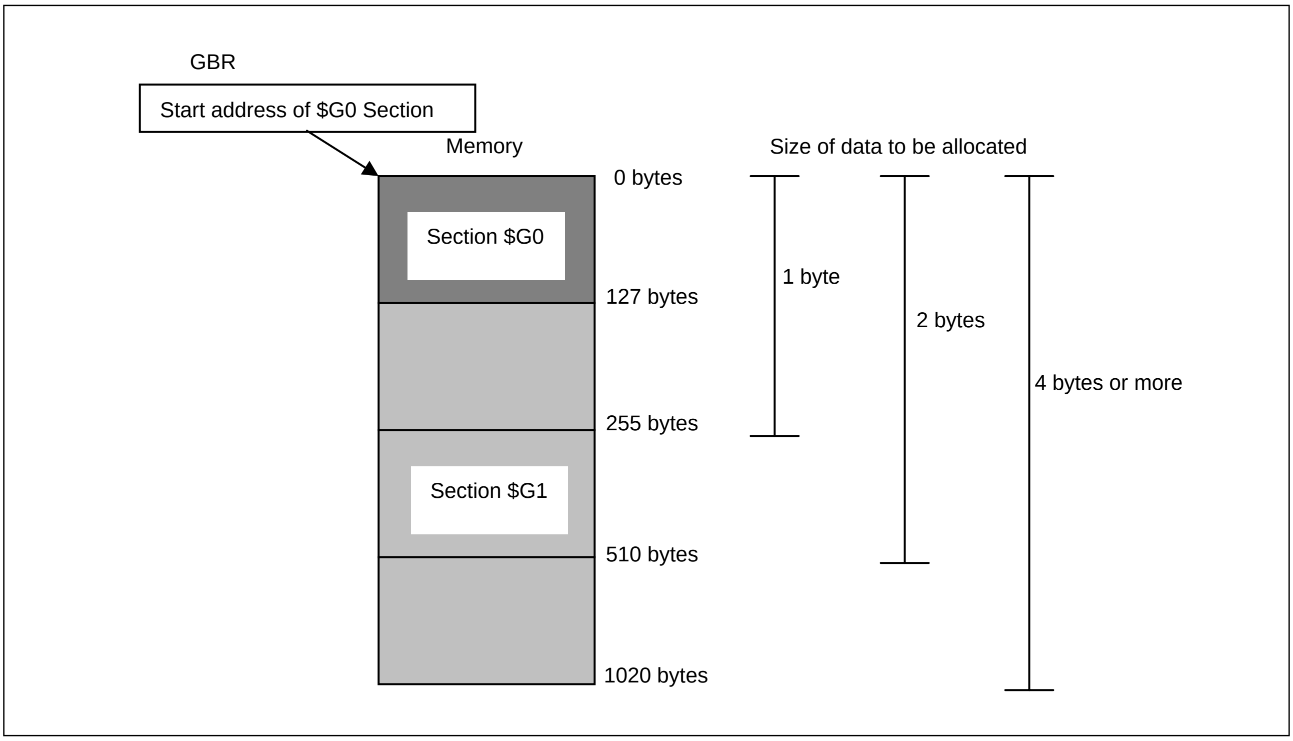


Figure 3.2 GBR Base Variable Referencing

3.4.1 Specification of GBR Base Variables

Description:

Preprocessor directives are used in order to implement GBR base referencing of external variables.
The "#pragma gbr_base"directive specifies that a variable is at an offset of 0 to 127 bytes from the address indicated by GBR. The variable specified here is allocated to the section $G1.
The "#pragma gbr_base1"directive specifies that the offset from the address indicated by GBR of a variable is, for the char type and unsigned char type, at most 255 bytes; for the short and unsigned short types, at most 510 bytes; and for the int, unsigned int, long, unsigned long, float, and double types, at most 1020 bytes. The variable specified here is allocated to the section $G1.

Format:

```
        #pragma gbr_base ( variable name > [,< variable name >…] )
        #pragma gbr_base1 ( variable name > [,< variable name >…] )
```

Example of use:

C language code

```
        #pragma gbr_base(a1,b1,c1)
        #pragma gbr_base1(a2,b2,c2)
        char a1,a2;
        short b1,b2;
        long c1,c2;
        void f()
        {
             a1 = a2;
             b1 = b2;
             c1 = c2;
        }
```

Expanded into assembly language code

```
  _f:
            MOV.B       @(_a2-(STARTOF $G0),GBR),R0
            MOV.B       R0,@(_a1-(STARTOF $G0),GBR)
            MOV.W       @(_b2-(STARTOF $G0),GBR),R0
            MOV.W       R0,@(_b1-(STARTOF $G0),GBR)
            MOV.L       @(_c2-(STARTOF $G0),GBR),R0
            RTS
            MOV.L       R0,@(_c1-(STARTOF $G0),GBR)
```

In order to use GBR base variables, it is necessary to set the start address of $G0 section as a GBR register beforehand. An example of this is shown below.

Initialization program (assembly language part)

```
                 :
                .SECTION  $G0,DATA,ALIGN=4
                 :
  __G_BGN:      .DATA.L   (STARTOF $G0)        ;Start address of $G0 section
                 :                             ;Specifies the address
                .EXPORT   __G_BGN
                 :
                 .END
```

Initialization program (C language part)

```
        #include <machine.h>
        extern int *_G_BGN;
        void  INITSCT()          /* Function executed before the main function */
        {
            :
        set gbr( G BGN);         /* Specifies the start address of $G0 section in the GBR register */
            :
        }
```

Important Information:

In using this feature, the following rules should be followed.

(1) At the start of program execution, GBR should be set to the start address of the $G0 section.

(2) The $G1 section should always be placed immediately after the $G0 section at linkage. Even when using only "#pragma gbr_base1", the $G0 section must always be created.

(3) If the total size after linking of the $G0 section exceeds 128 bytes, or if there are variables with offsets which exceed the offsets specified for the different data types in the explanation of "#pragma gbr_base1", correct operation is not guaranteed.

(4) If 2 and 3 above are not satisfied, correct operation is not guaranteed; the map list output at linkage should be checked to confirm that they are satisfied.

(5) Data which is accessed especially frequently, and data on which bit operations will be performed, should be allocated to section $G0 whenever possible. An object file is created which is more efficient in terms of both execution speed and size when data is allocated to section $G0 rather than to section $G1.

(6) Variables for which "#pragma gbr_base" or "#pragma gbr_base1" is specified are allocated to sections in the order in which the variables are declared. It should be remembered that if variables of different sizes are declared in alternation, the data size is increased.

(7) When gbr=auto is specified,the specification of #pragma gbr_base or #pragma gbr_base1 will be invalid(Ver.7 or later).

3.4.2 Register Allocation of Global Variables

Description:

The global variable specified by <variable name> is allocated to the register specified by <register name>.

Format:

```
            #pragma global_register(< variable name >=< register name >,...)
```

Example of use:

C language code

```
#pragma global_register(x=R13,y=R14)
      int    x;
      char   *y;
func1()
{
             x++;
}
func2()
{
             *y=0;
}
func(int a)
{
             x = a;
             func1();
             func2();
}
```

Expanded into assembly language code

```
          .EXPORT       _func1
          .EXPORT       _func2
          .EXPORT       _func
          .SECTION      P,CODE,ALIGN=4
_func1:                              ; function: func1
                                     ; frame size=0
          RTS
          ADD           #1,R13
_func2:                              ; function: func2
                                     ; frame size=0
          MOV           #0,R3
          RTS
          MOV.B         R3,@R14
_func:                               ; function: func
                                     ; frame size=4
```

```
            STS.L       PR,@-R15
            BSR         _func1
            MOV         R4,R13
            BRA         _func2
            LDS.L       @R15+,PR
            .SECTION    B,DATA,ALIGN=4
            .END
```

Important Information:

(1) Simple types and pointer types of variables can be used as global variables. Unless the "-double=float" option is specified, double type variables cannot be specified (except for the SH2A-FPU, SH-4, and SH-4A).

(2) Registers which can be specified are R8 to R14, FR12 to FR15 (for the SH-2E,SH2A-FPU, SH-4, and SH-4A), and DR12 to DR14 (for the SH2A-FPU, SH-4, and SH-4A).

(3) Initial values cannot be set. Also, addresses cannot be referenced.

(4) References of specified variables from linked files are not guaranteed.

(5) Static data members can be specified, but nonstatic data members cannot be specified.

Variable types which can be set in FR12 to FR15

(i) For SH-2E
float type
double type (when the double=float option is specified)

(ii) For SH2A-FPU, SH-4, and SH-4A
float type (without the fpu=double option)
double type (with the fpu=single option)

Variable types which can be set in DR12 to DR15

(i) For SH2A-FPU, SH-4, and SH-4A
float type (with the fpu=double option)
double type (without the fpu=single option)

3.5 Control of Register Save/Restore Operations

Description:

In functions called by functions which perform no other processing, there are cases when it is desirable not to save and restore registers in order to further speed program execution. In such cases, the preprocessor directives "#pragma noregsave", "#pragma noregalloc", and "#pragma regsave" are used for more complete control over register save/restore operations.

(1) The "#pragma noregsave" directive specifies that general-purpose registers are not saved and restored at the entry and exit points of functions.
(2) The "#pragma noregalloc" directive is used to create an object that does not save/restore general-purpose registers at function entry/exit points, and does not allocate registers for register variables (R8 to R14) across function calls.
(3) The "#pragma regsave" directive is used to create an object which saves and restores R8 through R14 among the general-purpose registers at function entry/exit points, and does not allocate registers for register variables (R8 to R14).
(4) "#pragma regsave" and "#pragma noregalloc" can be specified simultaneously for the same function. Such overlapping specifications causes an object to be created in which all registers for register variables (R8 to R14) are saved and restored at the function entry/exit points, and no register variable registers are allocated across function calls.

- Format:

```
#pragma noregsave(< function name > [,< function name >…])
#pragma noregalloc(< function name > [,< function name >…])
#pragma regsave(< function name > [,< function name >…])
```

Example of use:

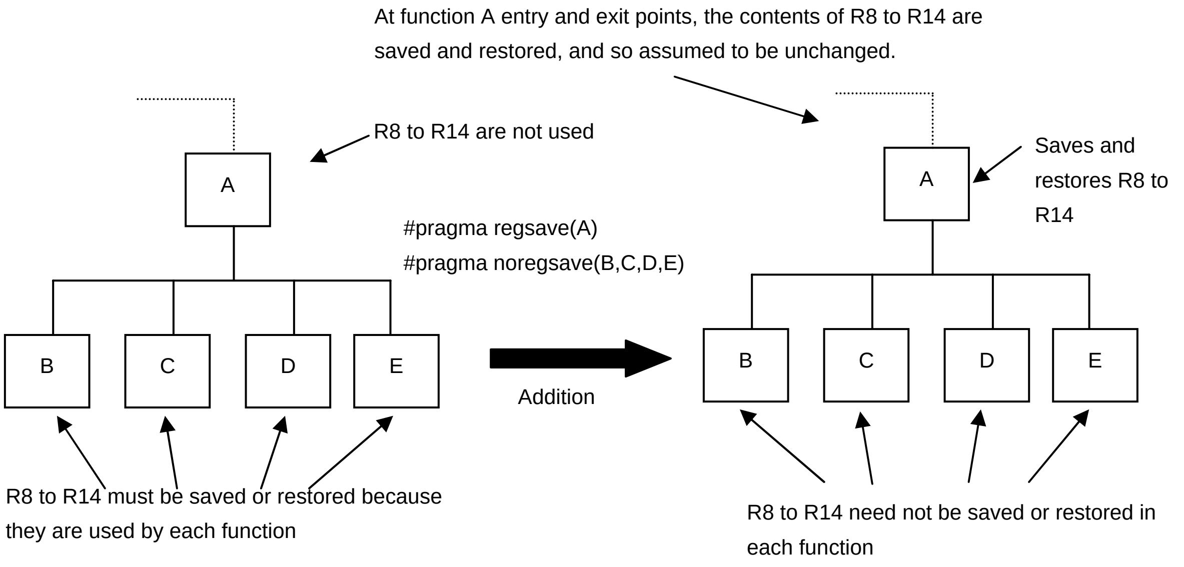
Examples of situations in which it is desirable to eliminate register storing/restoring, or to create conditions in which it can be eliminated, are shown below.

Example 1
When registers R8 to R14 are used in a function that is run at power-on, there is no need to save and restore registers, and so by specifying "#pragma noregsave" the object size is reduced, and the speed of execution improved.

Example 2
In cases where registers R8 to R14 are used in functions which put the system into low-power mode without returning control to the calling function, and in similar cases, there is no need for register save/restore operations. Hence by specifying "#pragma noregsave", the object size can be reduced and execution speed improved.

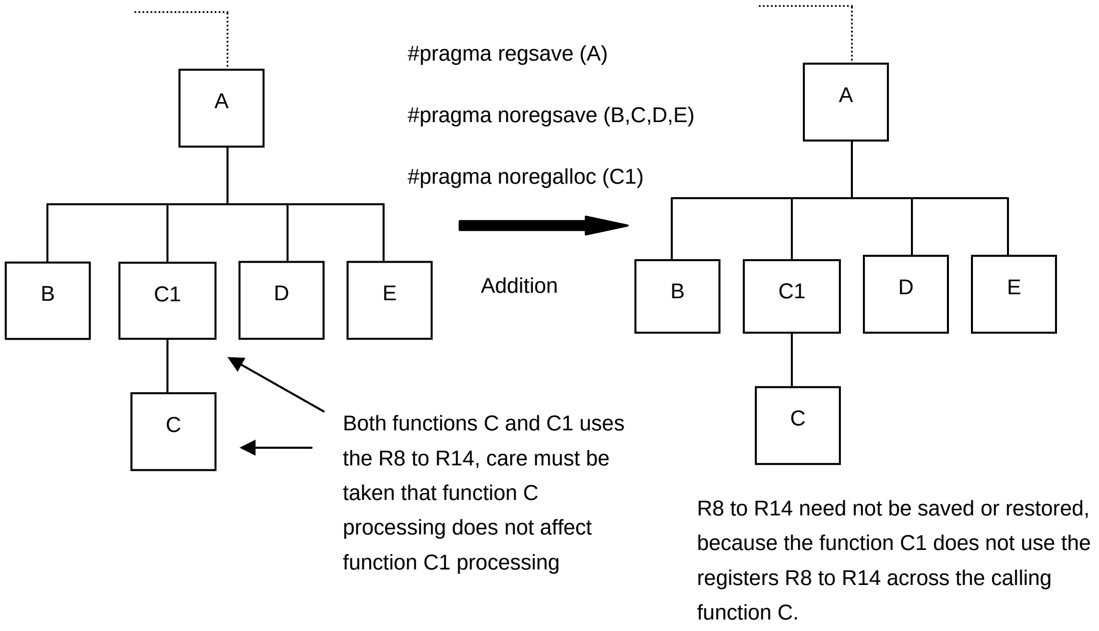
Example 3

When registers R8 to R14 are not allocated by function A, but are allocated by functions B, C, D and E, an object which performs save/restore operations for R8 to R14 is generated. Function A does not use R8 to R14, and so there are no adverse effects if a function called by function A does not save/restore the registers; but there are cases in which a function which calls function A uses the registers. In such cases directives can be added so that save/restore is performed at function A entry and exit points, but is not performed for each of the functions called by function A.


At function A entry and exit points, the contents of R8 to R14 are saved and restored, and so assumed to be unchanged.
R8 to R14 are not used
A
#pragma regsave(A)
#pragma noregsave(B,C,D,E)
B
C
D
E
Addition
A
Saves and restores R8 to R14
B
C
D
E
R8 to R14 must be saved or restored because they are used by each function
R8 to R14 need not be saved or restored in each function

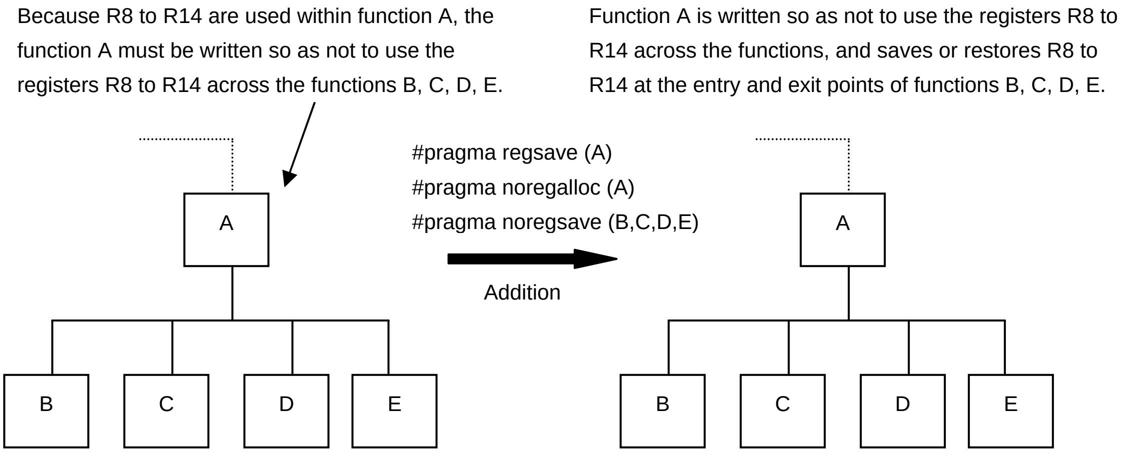
Example 4

When, using the same calling relation as in example 3 above, both functions C and C1 use the registers R8 to R14, the function C1 must not use the registers R8 to R14 across the calling function C. In such cases, if the "#pragma noregalloc" directive is used with function C1, specifying that allocation of R8 to R14 not exceed function calls, then function C can be specified using the "pragma noregsave" directive.


#pragma regsave (A)
#pragma noregsave (B,C,D,E)
#pragma noregalloc (C1)
A
B
C1
D
E
C
Addition
A
B
C1
D
E
C
Both functions C and C1 uses the R8 to R14, care must be taken that function C processing does not affect function C1 processing
R8 to R14 need not be saved or restored, because the function C1 does not use the registers R8 to R14 across the calling function C.

Example 5

When, in a calling relation similar to that of example 3, registers R8 to R14 are used within function A, the function A must be written so as not to use the registers R8 to R14 across the functions B, C, D, E. In such cases, both the specifications "#pragma regsave" and "#pragma noregalloc" are used for the function A. By specifying both "#pragma regsave" and "#pragma noregalloc", the registers R8 to R14 are saved and restored at function entry/exit points, and code is output which does not allocate R8 to R14 across function calls, so that "#pragma noregsave" can be specified for the functions B, C, D, E.


Because R8 to R14 are used within function A, the function A must be written so as not to use the registers R8 to R14 across the functions B, C, D, E.

Function A is written so as not to use the registers R8 to R14 across the functions, and saves or restores R8 to R14 at the entry and exit points of functions B, C, D, E.

Important Information:

The results of calling a function specified using "#pragma noregsave" by a method other than those listed below are not guaranteed.

(1) When used as the first function started, not called by any other function
(2) When called from a function specified using "#pragma regsave"
(3) When called from a function specified using "#pragma regsave", via a function specified using "#pragma noregalloc"

3.6 Specification of 16/20/28/32-Bit Address Areas

Description:

Preprocessor directives can be used to specify to the compiler that externally referenced variable and function addresses are 16, 20, 28, or 32 bits.
The compiler assumes that identifiers declared using "#pragma abs16" can be represented as 16-bit addresses, and allocates only a 16-bit address storage space where normally 32 bits would be allocated. In this way, the object size can be reduced, for greater efficiency of ROM use.
In addition, if memory is allocated at design time such that variables and functions referenced by multiple functions are preferentially placed in addresses represented by 16 bits, this feature can be used effectively.
A 16-bit address conversion option was added beginning with the SuperH RISC engine C/C++ compiler Ver. 4.1. This option can also be used for multiple specifications. For details, see appendix B, Added Features.
In Ver. 9.0 or a later version, a 20/28-bit address area can be specified in SH-2A and SH2A-FPU. This option can also be used for multiple specifications.

- Format:

```
<Preprocessor directive>
    #pragma abs16 (<identifier> [,<identifier>...])
    #pragma abs20 (<identifier> [,<identifier>...])
    #pragma abs28 (<identifier> [,<identifier>...])
    #pragma abs32 (<identifier> [,<identifier>...])
             identifier: variable name | function name
<Options>
    abs16 = { program | const | data | bss | run | all }[,...]
    abs20 = { program | const | data | bss | run | all }[,...]
    abs28 = { program | const | data | bss | run | all }[,...]
    abs32 = { program | const | data | bss | run | all }[,...]
             The default is abs32=all.
```

Example of use:

Example 1

Externally accessed variable and function are specified as having 16-bit addresses.

C language code

```
#pragma abs16 (x,y,z)
extern int x();
int y;
long z;
f ()
{
   z = x() + y;
}
```

Expanded into assembly language code

```
_f:
          STS.L     PR,@-R15
          MOV.W     L218,R3        ;Loads x address
          JSR       @R3
          NOP
          MOV.W     L218+2,R3      ;Loads y address
          MOV.L     @R3, R2
          MOV.W     L218+4,R1      ;Loads z address
          ADD       R2,R0
          LDS.L     @R15+,PR
          RTS
          MOV.L     R0,@R1
L218 :
          .DATA.W   _x
          .DATA.W   _y
          .DATA.W   _z
```

Example 2

Externally accessed variable and function are specified as having 20-bit addresses.

C language code

```
     #pragma abs20 (x,y,z)
     extern int x();
     int y;
     long z;
     f()
     {
          z = x() + y;
     }
```

Expanded into assembly language code

```
_f:
          STS.L         PR,@-R15
          MOVI20        #_x,R2        ;  Loads x address
          JSR/N         @R2
          MOVI20        #_y,R5        ;  Loads y address
          MOV.L         @R5,R1
          MOVI20        #_z,R4        ;  Loads z address
          ADD           R1,R0
          LDS.L         @R15+,PR
          RTS
          MOV.L         R0,@R4
_y:
          .RES.L        1
```

```
 _z:
           .RES.L        1
```

Example 3

Externally accessed variable and function are specified as having 28-bit addresses.

C language code

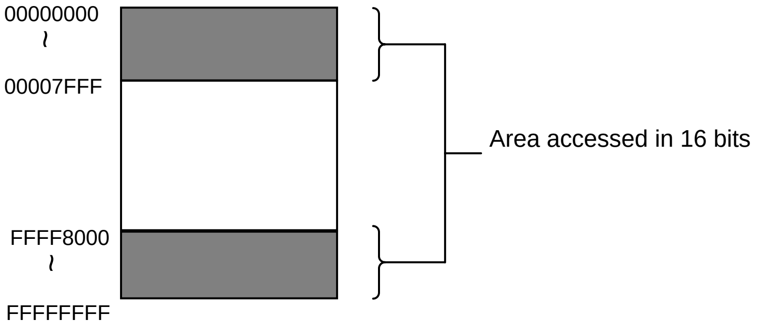
```
#pragma abs28 (x,y,z)
extern int x();
int y;
long z;
f()
{
     z = x() + y;
}
```

Expanded into assembly language code

```
 _f:
           STS.L         PR,@-R15
           MOVI20S       #_x+H'80,R2     ; Loads x address
           ADD           #Low _x,R2
           JSR/N         @R2
           MOVI20S       #_y+H'80,R5     ; Loads y address
           ADD           #Low _y,R5
           MOV.L         @R5,R1
           MOVI20S       #_z+H'80,R4     ; Loads z address
           ADD           #Low _z,R4
           ADD           R1,R0
           LDS.L         @R15+,PR
           RTS
           MOV.L         R0,@R4
 _y:
           .RES.L        1
 _z:
           .RES.L        1
```

Important Information:

(1) Variables and functions with the "abs16/20/28/32" option specified should be placed in different sections using section switching, with sections arranged such that addresses can be represented by the specified bits at linkage. If addresses which can be represented by the specified bits are not assigned, an error occurs at linkage.



The following table lists the access range available for each specification.

Table 3.23 Address Range

#pragma/option	Address range	
	Lower	Upper
abs16	0x00000000	0x00007FFF
	0xFFFF8000	0xFFFFFFFF
abs20	0x00000000	0x0007FFFF
	0xFFF80000	0xFFFFFFFF
abs28	0x00000000	0x07FFFF7F*[1]
	0xF8000000	0xFFFFFFFF
abs32	0x00000000	0xFFFFFFFF

Note:1. Note that the address is 0x07FFFF7F.

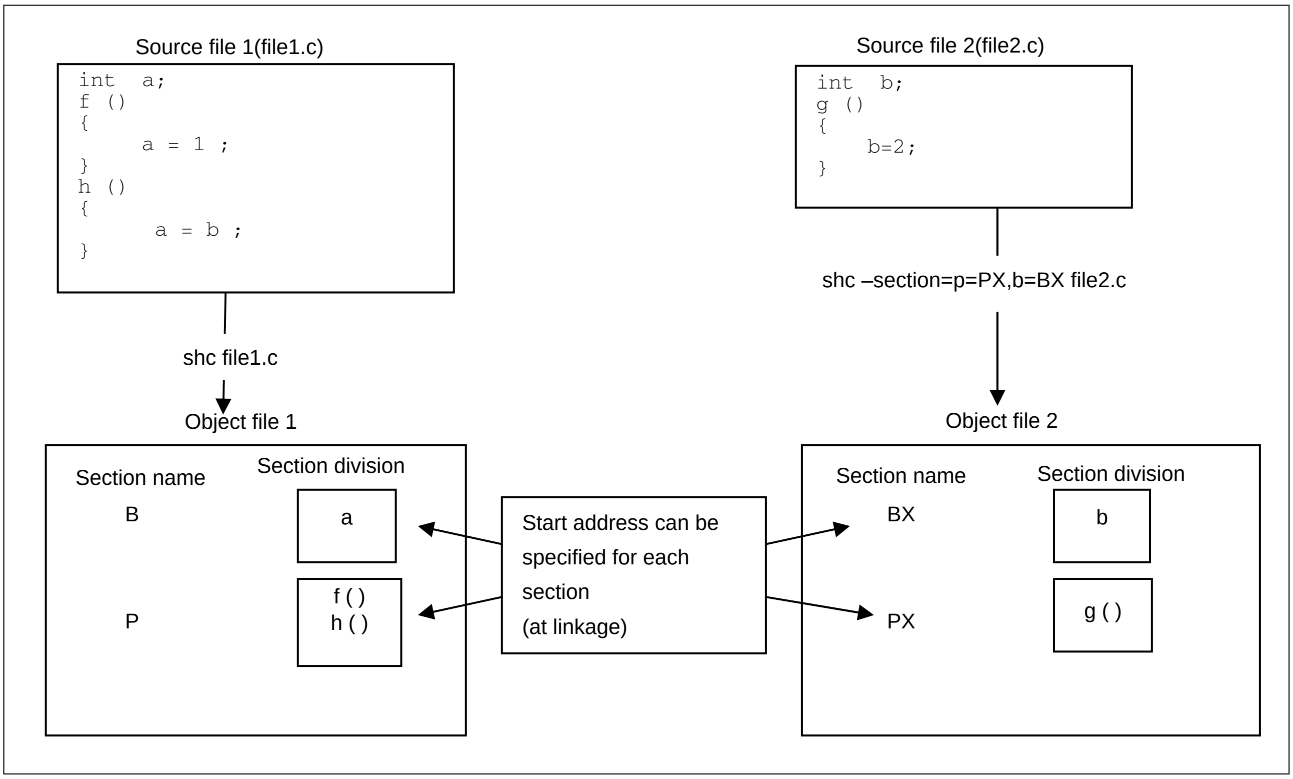
(2) If position-independent code generation is specified at compilation, function addresses having the specified number of bits are not generated.

3.7 Section Name Specification

When there is a need to allocate different sections of a system with the same attributes to different addresses (for example, a need to allocate a certain module to external RAM, and another module to internal RAM), names are assigned to the different sections, and addresses are assigned to the sections at linkage. The SuperH RISC engine C/C++ compiler provides two different methods for specifying section names. Below a methods is indicated for specifying separate section names for multiple modules. In the example of this explanation, it is assumed that modules f, g, h and data a, b are divided with f, h, a together, and g, b together.

3.7.1 Section Name Specification

The SuperH RISC engine C/C++ compiler can specify object section names at compilation through the "-section" option. Using this feature, modules and data to be separated can be incorporated into different files, different section names specified at compilation, and start addresses for each specified at linkage.



Source file 1(file1.c)

```
int  a;
f ()
{
     a = 1 ;
}
h ()
{
      a = b ;
}
```

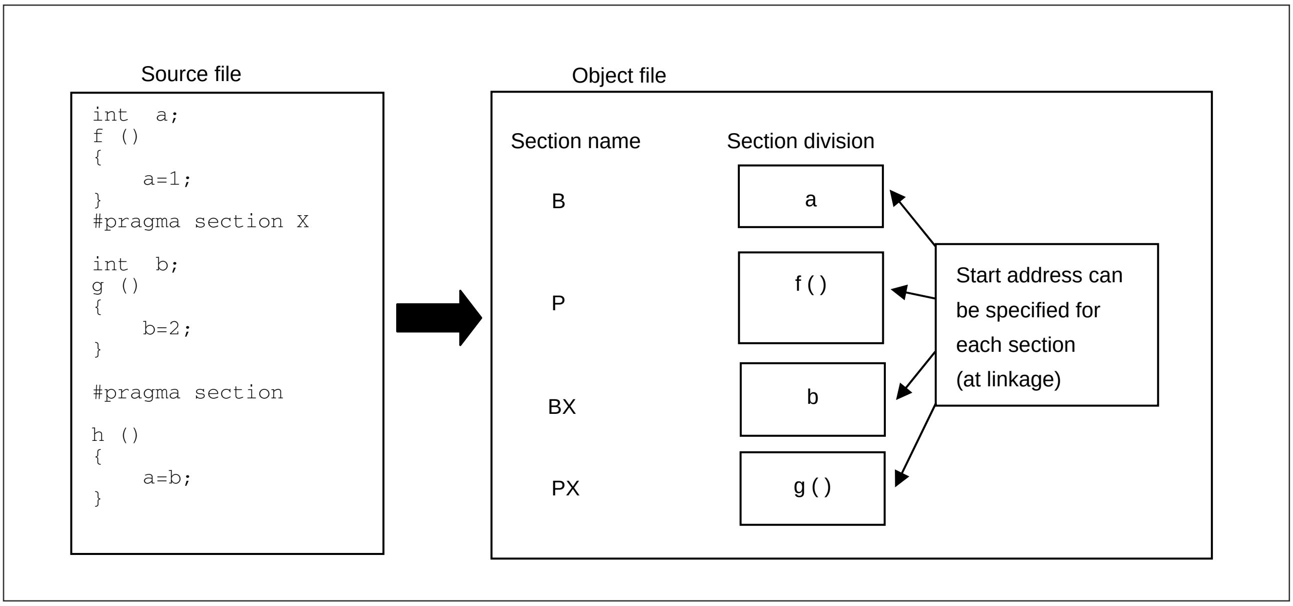
Source file 2(file2.c)

```
int  b;
g ()
{
    b=2;
}
```

Figure 3.3 Method for Specifying Section Names

3.7.2 Section Switching

Using the "-section" option, section names can be specified only in file units; but by using the "#pragma section" directive, section names with the same attributes can be switched within a single file, for finer control of memory allocation. By means of this feature, the section divisions of explained in section 3.7.1, Section Name Specification, can be described in a single file. An example of the use of this feature appears in figure 3.4.



```
int  a;
f ()
{
    a=1;
}
#pragma section X

int  b;
g ()
{
    b=2;
}

#pragma section

h ()
{
    a=b;
}
```

Figure 3.4 Section Switching Method

In this figure, by including the "#pragma section X" directive, the name of the program section from this line until the "#pragma section" directive becomes "PX", and the name of the uninitialized data section becomes "BX".
With the "#pragma section" directive, the default section name is reinstated.

3.8 Specification of Entry Functions, and SP Settings

Description:

Functions specified with “function name” are handled as entry functions. Save and restore code for registers cannot be created by entry functions in any circumstances. If the CPU is SH-3,SH3-DSP,SH-4,SH-4A, or SH4AL-DSP, and if there is a sp=<constant> specification or a #pragma stacksize declaration, stack pointer initialization code is output at the start of the function.

Format:

```
#pragma entry  (< function name >=<(sp=<constant>))
```

Example of use:

<u>C language code</u>

<u>Example 1:</u>

```
#pragma entry INIT(sp=0x10000)
void INIT() {
:
}
```

<u>Example 2:</u>

```
#pragma stacksize 100
#pragma entry INIT
void INIT() {
:
}
```

<u>Expanded into assembly language code</u>

<u>Example 1:</u>

```
.SECTION P, CODE
_INIT:
MOV.L L1, R15
:
L1: .DATA.L H’00010000
:
```

<u>Example 2:</u>

```
.SECTION S, STACK
.RES.B 100
.SECTION P, CODE
_INIT:
MOV.L L1, R15
:
L1: .DATA.L STARTOF S + SIZEOF S
:
```

Important Information:

Specify #pragma entry before the function declaration. Entry functions can only specify up to two load modules in total.
For <constant>, be sure to specify a multiple of 4.
If cpu=sh1|sh2|sh2e|sh2dsp is specified, the sp=<constant> specification is invalid.

3.9 Position-Independent Code

In order to improve execution speed, code in ROM is sometimes copied to RAM on startup and run from RAM. In order to do so, it is necessary that the program be able to load the ROM code into an arbitrary address. Such code is called position-independent code.
By specifying as a command-line option "pic=1" when compiling using the SuperH RISC engine C/C++ compiler, position-independent code is generated.

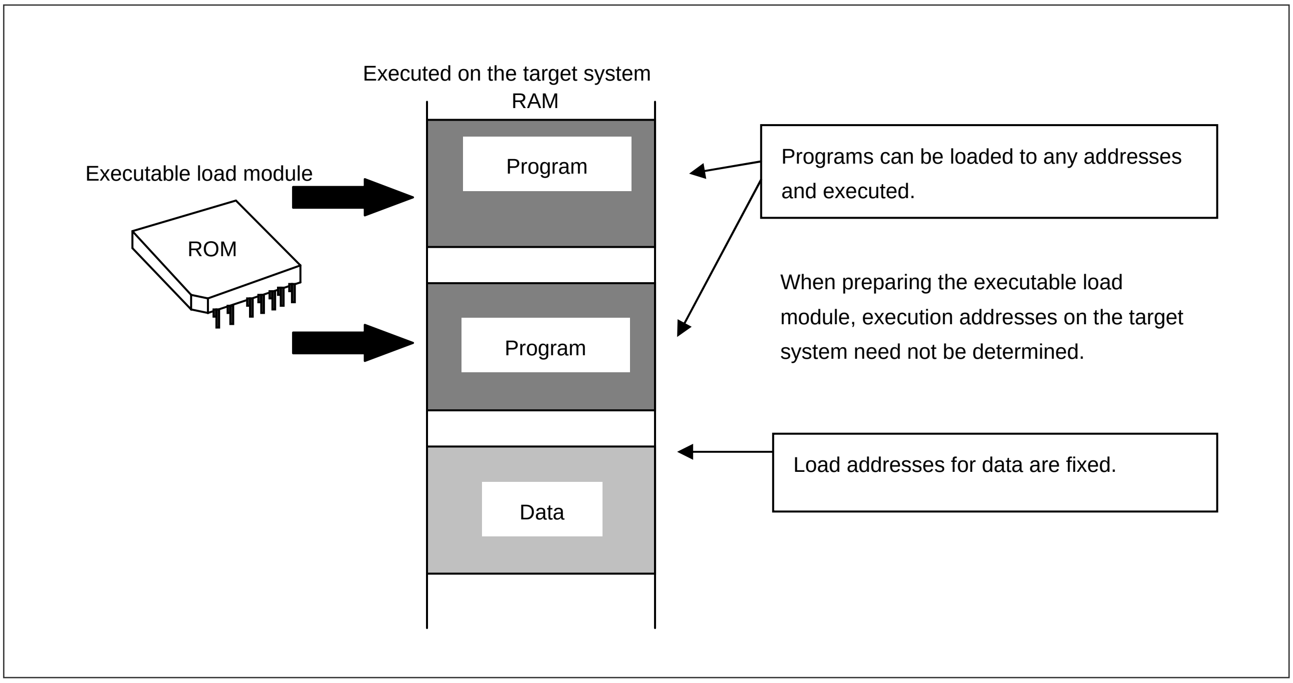

Executed on the target system
RAM
Executable load module
ROM
Program
Program
Data
Programs can be loaded to any addresses and executed.
When preparing the executable load module, execution addresses on the target system need not be determined.
Load addresses for data are fixed.

Figure 3.5 Position-Independent Code

Notes: 1. This feature cannot be used for the SH-1.
2. This feature cannot be applied to data sections.
3. In execution as position-independent code, function addresses cannot be specified as initial values.

Examples:

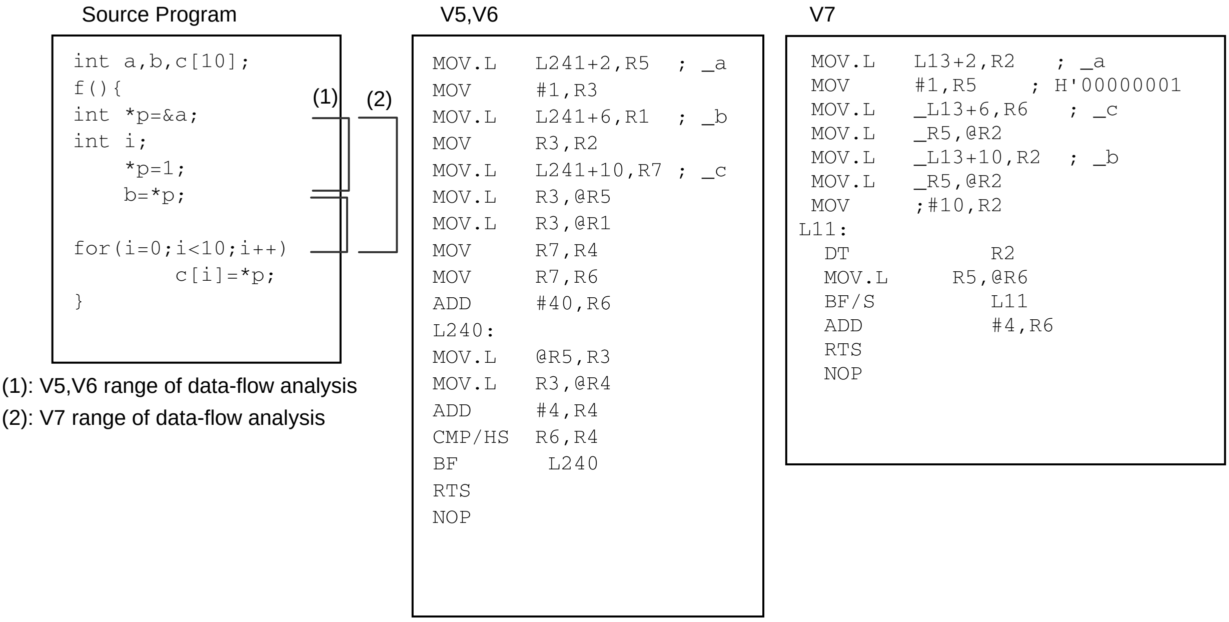
```
extern int f();
int (*fp)() = f;
```

The address of the function f is indeterminate until it is loaded into RAM, and so in this case operation is not guaranteed.

4. When using this feature, please use a standard library compatible with position-independent code. For details on creating a library, refer to the SuperH RISC engine C/C++ Compiler, Assembler, and Optimizing Linkage Editor User's Manual.

3.10 MAP Optimization

With supercomputers in mind the latest optimization processing has been applied, and with alias analysis of pointer and external variables, and data-flow analysis including of control statements, further improved optimization has been achieved.



Source Program

```
int a,b,c[10];
f(){
int *p=&a;
int i;
    *p=1;
    b=*p;

for(i=0;i<10;i++)
        c[i]=*p;
}
```

(1): V5,V6 range of data-flow analysis

(2): V7 range of data-flow analysis

V5,V6

```
MOV.L   L241+2,R5  ; _a
MOV     #1,R3
MOV.L   L241+6,R1  ; _b
MOV     R3,R2
MOV.L   L241+10,R7 ; _c
MOV.L   R3,@R5
MOV.L   R3,@R1
MOV     R7,R4
MOV     R7,R6
ADD     #40,R6
L240:
MOV.L   @R5,R3
MOV.L   R3,@R4
ADD     #4,R4
CMP/HS  R6,R4
BF       L240
RTS
NOP
```

V7

```
 MOV.L   L13+2,R2   ; _a
 MOV     #1,R5    ; H'00000001
 MOV.L   _L13+6,R6   ; _c
 MOV.L   _R5,@R2
 MOV.L   _L13+10,R2  ; _b
 MOV.L   _R5,@R2
 MOV     ;#10,R2
L11:
  DT           R2
  MOV.L     R5,@R6
  BF/S         L11
  ADD          #4,R6
  RTS
  NOP
```

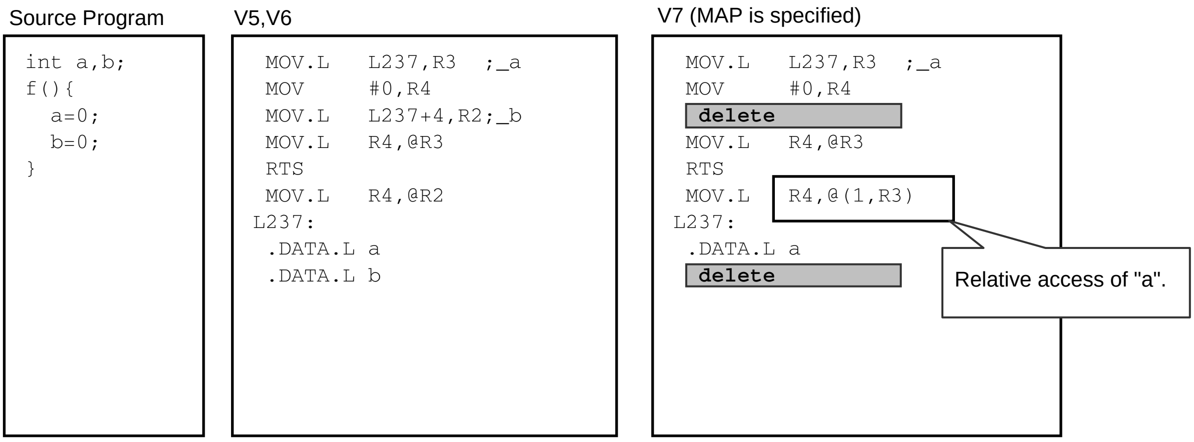
3.10.1 Procedure for Use

Recompile using symbol allocated addresses assigned by compiling and linking.
By this means, optimization reliant on allocated addresses can be achieved by the compiler.

```
Procedure for use:
First compilation and linkage:
Compilation with normal options
Linkage with -map=<file>.bls option -> outputs <file>.bls

Second compilation and linkage:
Compilation with -map=<file>.bls option
Linkage with normal options
```

3.10.2 Example of Improved External Variable Access Code (1)

Taking into account the order of variable allocation, consecutively access allocated variables in the same register relatively.



Source Program

```
int a,b;
f(){
  a=0;
  b=0;
}
```

V5,V6

```
 MOV.L   L237,R3  ;_a
 MOV     #0,R4
 MOV.L   L237+4,R2;_b
 MOV.L   R4,@R3
 RTS
 MOV.L   R4,@R2
L237:
 .DATA.L a
 .DATA.L b
```

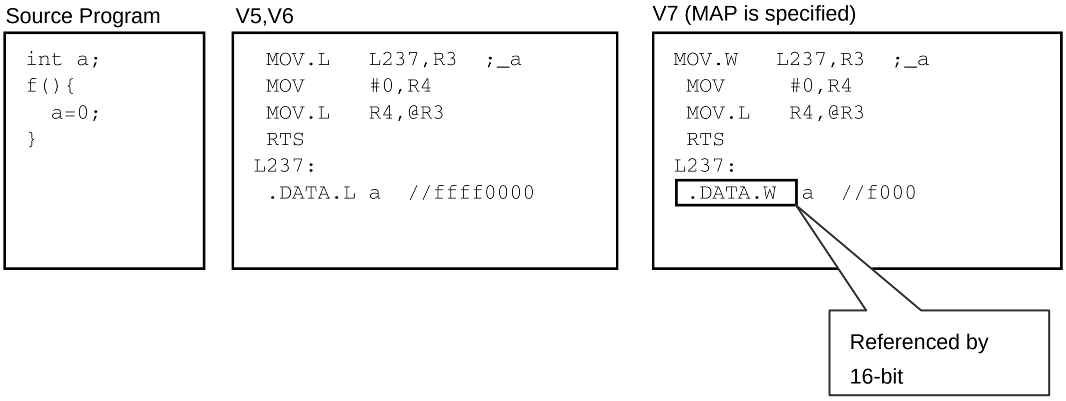
V7 (MAP is specified)

```
 MOV.L   L237,R3  ;_a
 MOV     #0,R4
 delete
 MOV.L   R4,@R3
 RTS
 MOV.L   R4,@(1,R3)
L237:
 .DATA.L a
 delete
```

Relative access of "a".

3.10.3 Example of Improved External Variable Access Code (2)

Variables allocated to 0 to7FFF, and FFFF8000 to FFFFFFFF are accessed by 16-bit literal.



Source Program

```
int a;
f(){
  a=0;
}
```

V5,V6

```
 MOV.L   L237,R3  ;_a
 MOV     #0,R4
 MOV.L   R4,@R3
 RTS
L237:
 .DATA.L a  //ffff0000
```

V7 (MAP is specified)

```
MOV.W   L237,R3  ;_a
 MOV     #0,R4
 MOV.L   R4,@R3
 RTS
L237:
 .DATA.W a  //f000
```

Referenced by 16-bit

3.10.4 Example of Improved External Variable Access Code (3)

Taking into account the branch destination address, branch using BSR/BRA.

Source Program

```
exterm g();
f(){
   g();
}
```

V5,V6

```
 MOV.L   L237,R2  ;_g
 JMP      @R2
 NOP
L237:
 .RES.W 1
 .DATA.L _g  ;within ±4096
```

V7 (MAP is specified)

```
BRA  _g
NOP
```

Branch using BRA

3.10.5 Example of Improved External Variable Access Code (4)

When the option is gbr=auto (default), GBR is used as the base for external variable access.

Source Program

```
int a[101];
int b;

void f()
{
  int i;

  for(i=0;
  i<100;
  i++)
    a[i]=0;

  a[50]=b;
}
```

V5,V6

```
_f:
  MOV.L    L239+4,R7  ; _a
  MOV      #0,R5
  MOV.W    L239,R6    ; H'0190
  MOV      R7,R4
  ADD      R7,R6
L238:
  MOV.L    R5,@R4
  ADD      #4,R4
  CMP/HS   R6,R4
  BF       L238
  MOV.L    L239+8,R2  ;
H'0C8+_a
  MOV.L    L239+12,R1 ; _b
  MOV.L    @R2,R3
  RTS
  MOV.L    R3,@R2
L239:
  .DATA.W  H'0190
  .DATA.W  0
  .DATA.L  _a
  .DATA.L  H'000000C8+_a
  .DATA.L  _b
```

V7 (MAP is specified)

```
_f:
  STC      GBR,@-R15
  MOV.L    L13+2,R0  ; _a
  LDC      R0,GBR
  MOV      #100,R6 ;
  STC      GBR,R2
  MOV      #0,R5
L11:
  ADD      #-1,R6
  MOV.L    R5,@R2
  TST      R6,R6
  ADD      #4,R2
  BF       L11
  MOV.L    @(404,GBR),R0
  MOV.L    R0,@(200,GBR)
  RTS
  LDC      @R15+,GBR
L13:
  .RES.W   1
  .DATA.L  _a
```

GBR referenced relatively

3.11 Options

3.11.1 Options for Code Generation

In order to enable the user to choose procedures for code generation, the SuperH RISC engine C/C++ compiler offers the following options.

Table 3.24 Options for Code Generation

Option	Description
-SPeed	Generates code with emphasis on execution speed
-SIze	Generates code with emphasis on reducing program size
-Goptimize	Outputs inter-module optimization add-on information.
-MAP	Sets the base address based on the external symbol allocation information generated by the Optimizing Linkage Editor, and generates code for performing external access relatively with a base address. If gbr=auto is specified, sets the GBR register as the base address depending on the conditions, and generates code for performing external access relatively with GBR.
-GBr	If gbr=auto is specified, depending on the conditions the compiler automatically generates GBR relative logic operation code. If gbr=auto and MAP=<filename> are set, sets the GBR as the base address depending on the conditions, and generates code for performing external variable access relatively with GBR.
-CAse	Specifies the code expansion method for the switch statement If case=ifthen is specified, the switch statement is expanded using the if_then method. Therefore, in this method, the object code size is increased according to the number of case labels included in the switch statement. If case=table is specified, the switch statement is expanded using the table method. In this case, the size of the jump table allocated in the constant area is increased in proportion to the number of case labels contained in switch statements, however, the execution speed is always constant. When this option is omitted, the compiler automatically judges which expansion method to use.
-SHIft	If shift=inline is specified, all shift operations are expanded with instructions. If shift=runtime is specified, a runtime routine call is made in cases where there are many expansion instructions.
-BLOckcopy	If blockcopy=inline is specified, all transfer code between memory is expanded with instructions. If blockcopy=runtime is specified, a runtime routine call is made in cases where the size of the transfer is large.
-INLine	Specifies whether to perform automatic inline expansion of functions. If the inline option is specified, automatic inline expansion is performed. It is possible to specify size.
-DIvision	Selects the method of integer-type division, and remainder calculation in the program. If division=cpu=inline is specified, constant division is converted to multiplication and inline expansion is performed, and variable division selects a runtime routine using the DIV1 instruction.
-Macsave	Specifies whether to guarantee the MACH and MACL registers before and after function calls. When 0 is specified, the MACH and MACL registers are not guaranteed before and after function calls.

3.11.2 Options for Optimization Linkage

In order to enable the user to choose procedures for optimization linkage, the SuperH RISC engine C/C++ Optimizing Linkage Editor offers the following options.

Table 3.25 Options for Linkage

Option	Sub-option	Description
-OPtimize	-	Specifies whether to execute inter-module optimization.
	-SPeed	Executes optimization except that which might possibly cause reduced object speed. Same as optimize=string_unify, symbol_delete, variable_access, register,branch
	-SAFe	Executes optimization except that which might possibly be limited by the attributes of the variable or function. Same as optimize=string_unify, register,branch
	-Branch	Based on the program allocation information, the branch instruction size is optimized. If other optimization items are executed, this is executed whether specified or not.
	-Register	The relationships between function calls are analyzed, and register reallocations and redundant register save/restore codes are deleted with this specification.
	-STring_unify	Unifies same value constants of constants with the const attribute. Constants with the const attribute include the following items. Variables declared const in C/C++ programs Initial values of string data Literal constants
	-SYmbol_delete	Variables/functions which are not referenced are deleted with this specification.
	-Variable_access	Allocates frequently accessed variables to areas accessible in 8/16-bit absolute addressing mode.
	-SAMe_code	Makes multiple similar instruction strings into subroutines.
	-Function_call	Allocates addresses of frequently accessed functions if the memory range from 0 to 0xFF has space.
-SMAESize	-	Specifies the minimum code size of code to be optimized using common code unification optimization.
-PROfile	-	Specifies the profile information file. Using inter-module optimization, optimization can be executed based on dynamic information.
-CAchesize	-	Specifies the cache size and cache line size. If the profile option is specified, it is used with branch instruction optimization.
-SYmbol_forbid	-	Disables optimization by deletion of unreferenced symbols.
-SAMECode_forbid	-	Disables optimization by unification of common code.
-Variable_forbid	-	Disables optimization by use of short-absolute addressing mode.
-FUnction_forbid	-	Disables optimization by use of indirect addressing mode.
-Absolute_forbid	-	Disables optimization of address + size range.

3.11.3 Options for Creating Standard Libraries

In order to enable the user to choose procedures for optimization when creating standard libraries, the SuperH RISC engine C/C++ Standard Library Creation Tool offers the following options.

Table 3.26 Options for Creating Standard Libraries

Option	Description
-SPeed	Generates code with emphasis on execution speed
-SIze	Generates code with emphasis on reducing program size
-Goptimize	Outputs inter-module optimization add-on information.
-MAP	Sets the base address based on the external symbol allocation information generated by the Optimizing Linkage Editor, and generates code for performing external access relatively with a base address. If gbr=auto is set, sets the GBR registers as the base address depending on the conditions, and generates code for performing external access relatively with GBR.
-GBr	If gbr=auto is specified, depending on the conditions the compiler automatically generates GBR relative logic operation code. If gbr=auto and MAP=<filename> are set, sets the GBR as the base address depending on the conditions, and generates code for performing external variable access relatively with GBR.
-CAse	Specifies the code expansion method for the switch statement If case=ifthen is specified, the switch statement is expanded using the if_then method. Therefore, in this method, the object code size is increased according to the number of case labels included in the switch statement. If case=table is specified, the switch statement is expanded using the table method. In this case, the size of the jump table allocated in the constant area is increased in proportion to the number of case labels contained in switch statements, however, the execution speed is always constant. When this option is omitted, the compiler automatically judges which expansion method to use.
-SHIft	If shift=inline is specified, all shift operations are expanded with instructions. If shift=runtime is specified, a runtime routine call is made in cases where there are many expansion instructions.
-BLOckcopy	If blockcopy=inline is specified, all transfer code between memory is expanded with instructions. If blockcopy=runtime is specified, a runtime routine call is made in cases where the size of the transfer is large.
-INLine	Specifies whether to perform automatic inline expansion of functions. If the inline option is specified, automatic inline expansion is performed. It is possible to specify size.

3.12 SH-DSP Features

The SH-DSP core is provided with a DSP unit which performs 16-bit fixed-point operations and is ideal for:

- Multiply-and-accumulate operations
- Repeated processing

It is thus capable of performing at high speed the JPEG processing, audio processing, and filter processing required for multimedia operations.

In previous SH cores (the SH-1 core example in figure 3.6), the performance of multiply-and-accumulate operations were determined by the three cycles constituting the multiplier operation time in pipeline operation. Even if the multiplier operation time were improved to a single cycle, however, stalling of the pipeline would occur due to instruction data transfer, so that the long-term average time would be 2.5 cycles.

In the SH-DSP core, the DSP unit operation time is a single cycle, and an X bus/Y bus is provided as the data bus, so that multiply-and-accumulate operations take just one cycle (figure 3.7). Here the long-term average time is also one cycle.

Code example

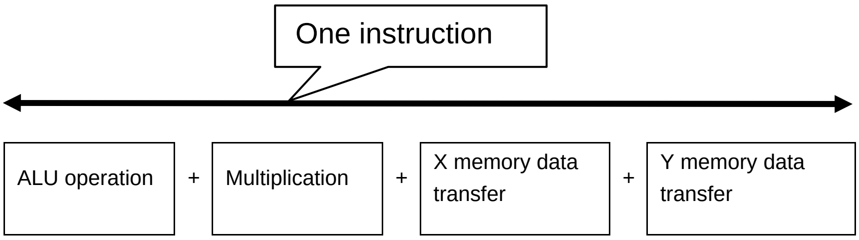
```
clrmac
mac.w @r4+,@r5+
mac.w @r4+,@r5+
mac.w @r4+,@r5+
mac.w @r4+,@r5+
rts
sts   macl,r0
```

IF	:Instruction fetch (32 bits)	MA	:Memory access
if	:Instruction fetch (with no bus cycles)	mul	:Multiplier operation
ID	:Decode	WB	:Write-back
EX	:Execution/ address calculation		

Example of pipeline operation

if	ID	EX	MA																
	IF	ID	EX	MA	▨	MA	mul	mul	mul										
		if	▨	▨	ID	EX	MA	▨	MA	mul	mul	mul							
					IF	▨	ID	EX	▨	MA	▨	MA	mul	mul	mul				
							if	▨	▨	▨	ID	EX	MA	▨	MA	mul	mul	mul	
											IF	▨	ID	▨	▨	EX			
													if	▨	▨	ID	EX	MA	WB

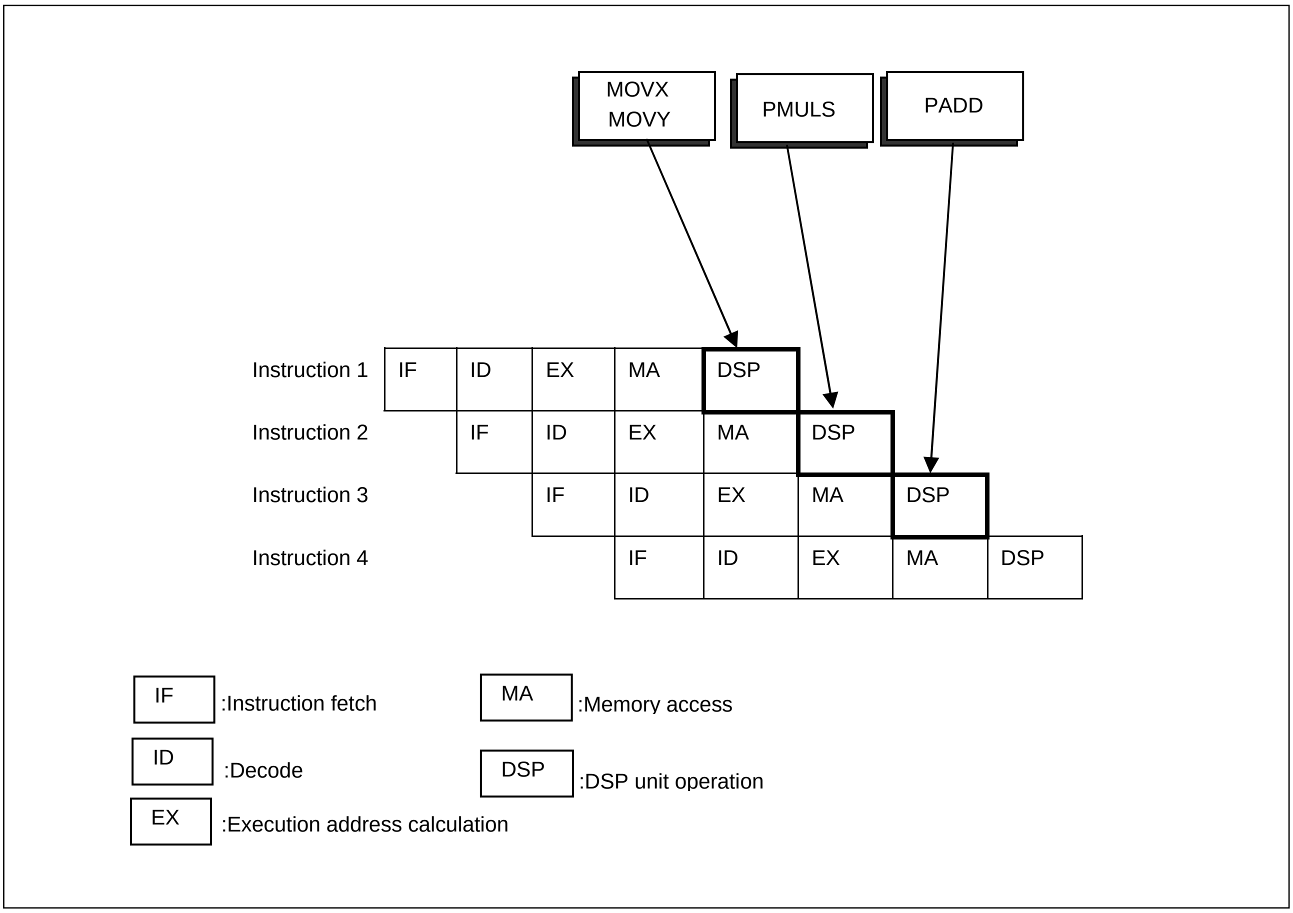
Figure 3.6 Multiple-and-Accumulate Instruction Executed in SH Core



Code example

```
Instruction 1                              MOVX.W@R4+,X0   MOVY.W@R6+,Y0
Instruction 2             PMULSX0,Y0,M0    MOVX.W@R4+,X1   MOVY.W@R6+,Y1
Instruction 3PADD A0,M0,A0PMULSX1,Y1,M1    MOVX.W@R4+,X0   MOVYW@R6+,Y0
Instruction 4PADD A0,M1,A0PMULS X0,Y0,M0   MOVX.W@R4+,X1   MOVYW@R6+,Y1
```

Example of pipeline operation



Instruction 1	IF	ID	EX	MA	DSP			
Instruction 2		IF	ID	EX	MA	DSP		
Instruction 3			IF	ID	EX	MA	DSP	
Instruction 4				IF	ID	EX	MA	DSP

IF :Instruction fetch

ID :Decode

EX :Execution address calculation

MA :Memory access

DSP :DSP unit operation

Figure 3.7 Multiply-and-Accumulate Instruction Executed in SH-DSP Core

Further, the SH-DSP core is equipped with hardware mechanisms to reduce disruption of the pipeline due to repeated processing.

In previous SH cores, conditional branching was used for loop processing. Conditional branching acts to disrupt pipelines, adding to processing overhead.

In the SH-DSP core there is a zero-overhead mechanism which reduces to zero the pipeline disruption due to this loop processing. Simply by setting the loop start and finish addresses and number of loops, loop processing is completed without performing conditional branching. Many critical software operations depend on loop processing; this is a hardware mechanism which is effective in speeding software execution.

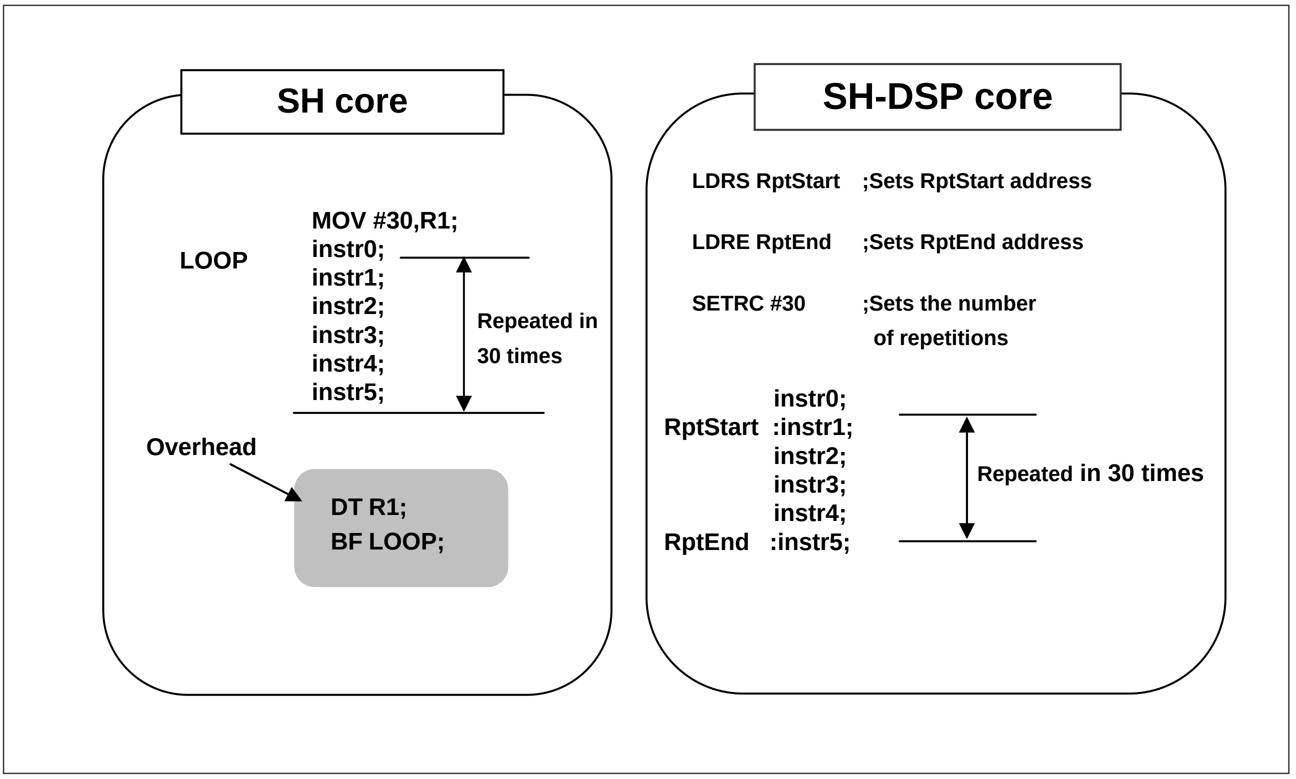


Figure 3.8 Repetition Processing

The SH-DSP core is able to execute in parallel five instructions, as shown in figure 3.9: condition evaluation, ALU operations, signed multiplication, X memory access, and Y memory access. By combining these instructions, various multiply-and-accumulate operations can be performed at high speed.

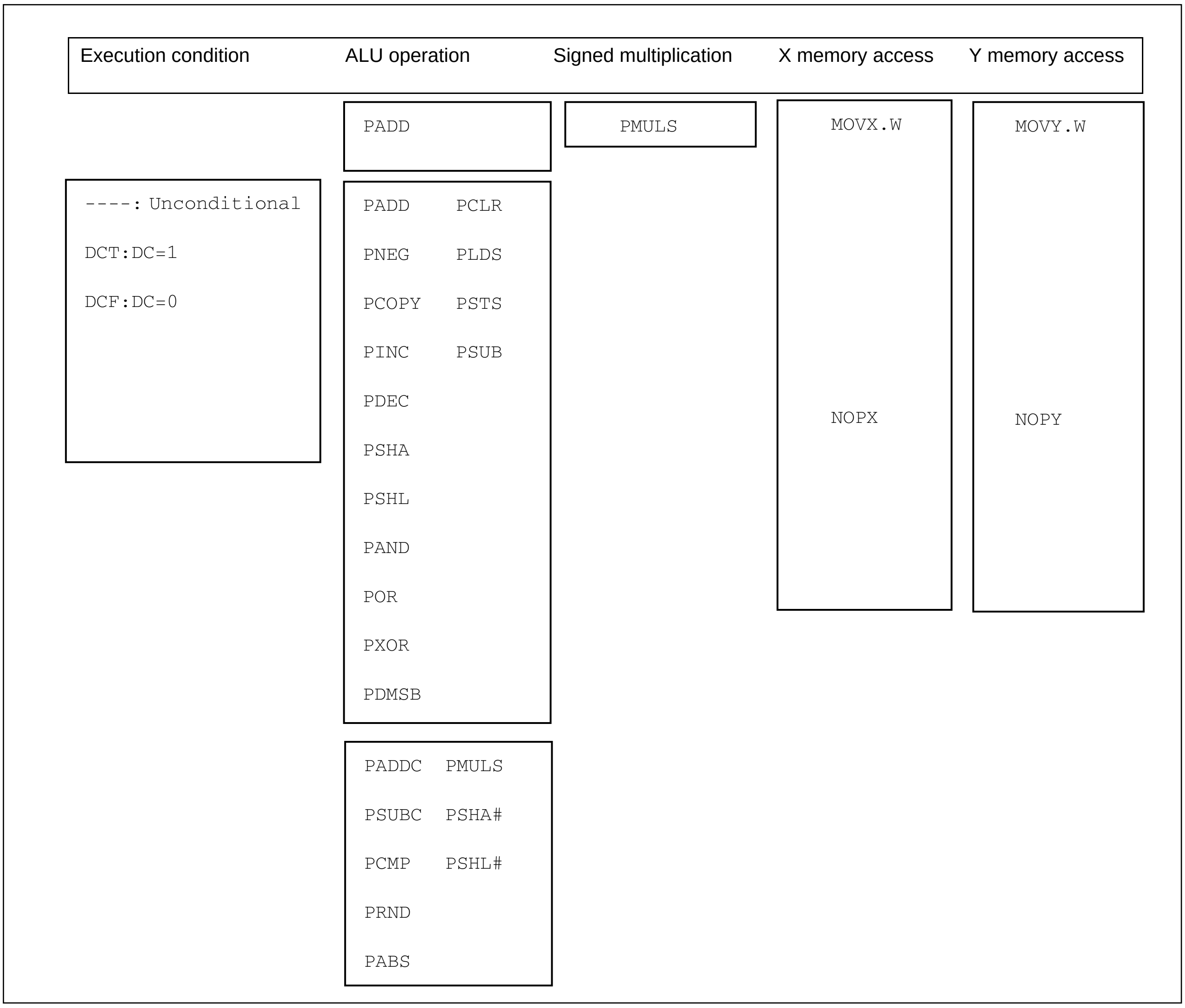


Figure 3.9 DSP Instructions (Parallel Instructions)

3.13 DSP Library

3.13.1 Summary

This section explains the digital signal processing (DSP) library that can be used with SH2-DSP and SH3-DSP (henceforward jointly referred to simply as SH-DSP) This library includes standard DSP functions, and by using them singly or consecutively, DSP operations can be performed.

This library includes the following functions.

- Fast Fourier transforms
- Window functions
- Filters
- Convolution and correlation
- Other

The functions in this library are, with the exception of fast Fourier transforms and filters, reentrant.

When using this library, include the files shown in table 3.27. In addition, as shown in table 3.28, link to the library corresponding to the CPU and compile options.

When this library is called on, if the function finishes normally, EDSP_OK is returned as the value, and if an error occurs, EDSP_BAD_ARG or EDSP_NO_HEAP is returned as the value. For the details of return values, refer to the explanation of each function.

Table 3.27 Include Files for Use with the DSP Library

Type of library	Description	Include file
DSP Library	The library performs DSP operations	<ensigdsp.h>
		<filt_ws.h>*[1]

Note: 1. When using filter functions, include them only once in the user program.

Table 3.28 DSP Library List

CPU	Option		Library Name
SH2-DSP	-pic=0		shdsplib.lib
	-pic=1		shdsppic.lib
SH3-DSP	-pic=0	-endian=big	sh3dspnb.lib
SH4AL-DSP	-pic=1	-endian=big	sh3dsppb.lib
	-pic=0	-endian=little	sh3dspnl.lib
	-pic=1	-endian=little	sh3dsppl.lib

3.13.2 Data Format

This library handles data as signed 16-bit fixed point numbers. Signed 16-bit fixed point numbers, as shown in (a) in figure 3.10, are of the data format where the point is fixed to the right side of the most significant bit (MSB), and values from
-1 to $1-2^{-15}$ can be expressed.

In this library, transfer of data uses the short type of data format. Therefore, when using this library from C/C++ programs, it is necessary to express data in signed 16-bit fixed point numbers.

Example: +0.5 expressed as a signed 16-bit fixed point number is H'4000. Therefore, the short type actual parameter passed to the library function is H'4000.

Internal operations within this library use signed 32-bit fixed point numbers and signed 40-bit fixed point numbers. Signed 32-bit fixed point numbers, are of the data format as shown in (b) in figure 3.10, and values from -1 to $1-2^{-31}$ can be expressed. Signed 40-bit fixed point numbers, are of the data format with an additional 8-bit guard bit as shown in (c) in figure 3.10, and values from -2^{8} to $2^{8}-2^{-31}$ can be expressed.

The multiplication results of signed 16-bit fixed point numbers are saved as signed 32-bit fixed point numbers. With fixed point multiplication using DSP instructions, only in the case of H'8000 x H'8000 is it necessary to be careful in case overflow occurs. In addition, the least significant bit (LSB) of multiplication results is normally 0. When the multiplication results are used in the next operation, the upper 16 bits are removed, and the result is converted to a signed 16-bit fixed point number. In this case, there is a possibility that underflow or reduced accuracy may occur.

In multiply-and-accumulate operations of this library, addition results are saved as signed 40-bit fixed point numbers. Be careful that overflow does not occur when performing addition.

If an overflow occurs when performing an operation, a correct result will not be obtained. In order to prevent overflows, it is necessary to perform scaling of coefficients or of input data. Scaling functions are built into this library. For the details of scaling, refer to the explanation of each function.

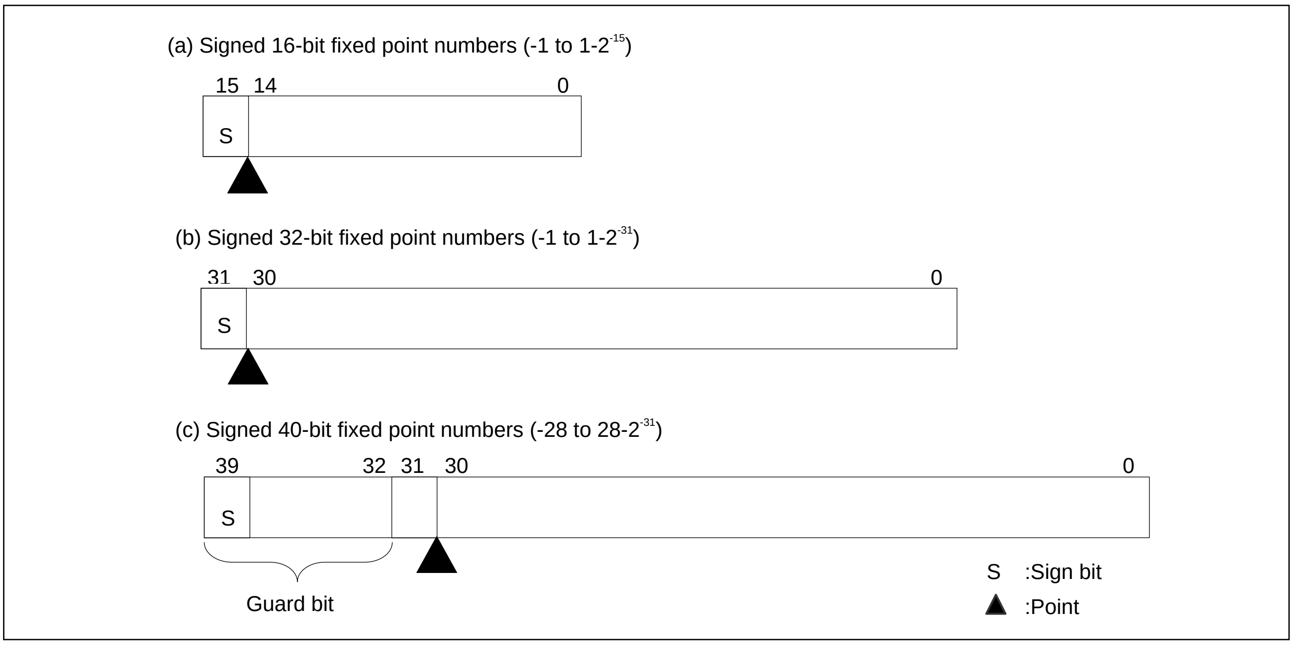

(a) Signed 16-bit fixed point numbers (-1 to 1-2-15)
15
14
0
S
(b) Signed 32-bit fixed point numbers (-1 to 1-2-31)
31
30
0
S
(c) Signed 40-bit fixed point numbers (-28 to 28-2-31)
39
32
31
30
0
S
Guard bit
S :Sign bit
▲ :Point

Figure 3.10 Data Format

3.13.3 Efficiency

The functions in this library are optimized to execute at high speed on SH-DSP.
In order to use the library efficiently, when deciding the memory map of the system in development, observe the following two recommendations as far as possible.

- Allocate memory that supports 32-bit read for 1 cycle for program code segments.
- Allocate memory that supports 16-bit (or 32-bit) read and write for 1 cycle for data segments.

If the microcomputer to be used has 32-bit memory built in of sufficient capacity to allocate the library code and data, it is best to allocate it to the 32-bit memory. If it is necessary to use other memory, follow the above recommendation as far as possible.

3.13.4 Fast Fourier transform

(1) List of functions

Table 3.29 List of DSP Library Functions (Fast Fourier Transform)

No.	Type	Function Name	Description
1	not-in-place complex number FFT	FftComplex	Performs not-in-place complex number FFT
2	not-in-place real-number FFT	FftReal	Performs not-in-place real-number FFT
3	not-in-place inverse complex number FFT	IfftComplex	Performs not-in-place inverse complex number FFT
4	not-in-place inverse real-number FFT	IfftReal	Performs not-in-place inverse real-number FFT
5	in-place complex number FFT	FftInComplex	Performs in-place complex number FFT
6	in-place real number FFT	FftInReal	Performs in-place real-number FFT
7	in-place inverse complex number FFT	IfftInComplex	Performs in-place inverse complex number FFT
8	in-place inverse real-number FFT	IfftInReal	Performs in-place inverse real-number FFT
9	logarithmic absolute value	LogMagnitude	Converts complex number data into logarithmic absolute values
10	FFT rotation factor generation	InitFft	Generates FFT rotation factors
11	FFT rotation factor release	FreeFft	Releases the memory used to store FFT rotation factors

Note: For details on not-in-place and in-place, refer to “(5) FFT structure”.

The factors use the scaling defined by the user to execute forward direction high speed Fourier transforms and reverse direction high speed Fourier transforms.
Forward direction Fourier transforms are defined using the following equations.

$$y_n = 2^{-s} \sum_{n=0}^{N} e^{-2j\pi n/N} \cdot x_n$$

Here, s represents the number of stages for performing scaling, and N represents the number of data elements.
Reverse direction Fourier transforms are defined using the following equations.

$$y_n = 2^{-s} \sum_{n=0}^{N} e^{2 j\pi n / N} \cdot x_n$$

For details on scaling, refer to “(4) Scaling”.

(2) Complex number data array format

FFT and IFFT complex number data arrays are allocated to X memory for real numbers and to Y memory for imaginary numbers. However, the allocation of real number FFT output data and real number IFFT input data differs. If the arrays in which real numbers and imaginary numbers are stored are defined as x and y respectively, the real number component of the DC component goes into x[0], and rather than the imaginary number component of the DC component, the real number component of the Fs/2 component goes into y[0] (the DC component and Fs/2 component are both real numbers, and the imaginary number component is 0).

(3) Real number data array format

There are 3 kinds of FFT and IFFT real number data array formats as follows.

- Stored in a single array, and allocated to an arbitrary memory block.
- Stored in a single array, and allocated to X memory.
- Divided into 2 arrays for storage. The size of each array is N/2, and the first half of the array is allocated to X memory, and the second half is allocated to Y memory.

Only the first specification method is available for FftReal. The user can select the second or third methods for IfftReal, FftInReal, and IfftInReal.

(4) Scaling

The signal strength of base 2 FFT doubles at each stage, and peak signal amplitude also doubles. For this reason, when converting to a high intensity signal there is a possibility that overflows may occur. However, by halving the signal at each stage (this is called ‘scaling’), overflows can be prevented. However, if excessive scaling is implemented, there is a possibility that unnecessary quantization noise may occur.

The optimal balance of scaling between overflows and quantization noise depends greatly on the characteristics of the input signals. In order to prevent overflows with spectra with large peaks in the signals, maximum scaling is necessary, but with impulse signals, scaling is hardly required at all.

Performing scaling at every stage is the safest method. If the intensity of the input data is less than 2^{30}, overflows can be prevented using this method. With this library, scaling can be specified for each stage. Therefore, by specifying scaling precisely, the impact of overflows and quantization noise can be suppressed to the minimum.

In order to specify the method of scaling, each FFT function parameter includes ‘scale’. ‘scale’ corresponds to each stage from the least significant bit to each individual bit. If the corresponding scale bit is set to 1, at every stage, division by 2 is executed.

In order to increase execution speed, base 4 FFT is used in this library. ‘scale’ corresponds to each stage from the least significant bit to each two bits. If either one bit is set to 1, division by 2 is executed. If both bits are set to 1, division by 4 is executed. In other words, this is the same as if two base 2 FFT stages are replaced with one base 4 FFT stage. However, with base 4 FFT, there is a greater possibility that quantization noise will occur than with base 2 FFT.

An example of ‘scale’ is shown below.

- When scale = H’FFFFFFFF (or size-1), scaling is performed for all base 2 FFT stages. If the intensity of all the input data is less than 2^{30}, overflow will not occur.
- When scale = H’55555555, scaling is performed for every other base 2 FFT stage.
- When scale = 0, scaling is not performed.

These scale values are defined as ensigdsp.h, EFFTALLSCALE(H’FFFFFFFF), EFFTMIDSCALE(H’55555555), and EFFTNOSCALE(0)

(5) FFT structure

The FFT structures of this library are of 2 kinds, not-in-place FFT, and in-place FFT

With not-in-place FFT, the input data is removed from RAM, FFT is executed, and the output result is stored in another place in RAM specified by the user.

On the other hand, with in-place FFT, the input data is removed from RAM, FFT is executed, and the output result is stored in the same place in RAM. If this method is used, execution time for the FFT is increased, but the memory space used can be decreased.

When using other FFT functions with input data, use not-in-place FFT. In addition, when seeking to conserve memory space, use in-place FFT.

(6) Explanation of each function

(a) not-in-place complex number FFT

Description:

- Format:

```
 int FftComplex (short op_x[], short op_y[],
                                  const short  ip_x[], const short ip_y[], long size,
 long scale)
```

- Parameters:

`op_x[]`	Real number component of output data
`op_y[]`	Imaginary number component of output data
`ip_x[]`	Real number component of input data
`ip_y[]`	Imaginary number component of input data
`size`	FFT size
`scale`	Scaling specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 4` •`size is not a power of 2` •`size > max_fft_size`

- Explanation of this function:

Executes a complex number fast Fourier transform.

- Remarks:

As this function performs not-in-place, provide input arrays and output arrays separately. For details on allocation of complex number data arrays, refer to “(2) Complex number data array format”. Before calling on this function, call on InitFft, and initialize the rotation factor and max_fft_size. For details on scaling, refer to “(4) Scaling”. ‘scale’ uses the lower $\log_2$ (size) bit. This function is not reentrant.

Example of use:

```
 #include <stdio.h>
 #include <math.h>
 #include <ensigdsp.h>
```

Include header

```
#define MAX_FFT_SAMP  64
#define MIN_CFFT_SIZE   4
long ip_scale=0xffffffff;
long size = MIN_CFFT_SIZE;

#pragma section X
short ip_x[MAX_FFT_SAMP];
short op_x[MAX_FFT_SAMP];
```

Variables placed in X or Y memory are defined by a pragma section within the section.

```
#pragma section Y
short ip_y[MAX_FFT_SAMP];
short op_y[MAX_FFT_SAMP];
#pragma section
```

Variables placed in X or Y memory are defined by a pragma section within the section.

```
/* Data for cycle counting */
#define TWOPI 6.283185307 /* data */
void main()
{
    int i,j;
    long n_samp;

    n_samp=MAX_FFT_SAMP; /* data */
    for (j = 0; j < n_samp; j++){
        ip_x[j] = cos(j * TWOPI/n_samp) * 8188;
        ip_y[j] = sin(j * TWOPI/n_samp) * 8188;
    }
```

Data creation for FFT

```
    if(InitFft(n_samp) != EDSP_OK){
        printf("Initfft != err end");
    }
```

FFT initialization function;
Initialization is performed for the number of data elements. This is required. The number of data elements is equal to the FFT data size, and must be a power of 2.

```
    if(FftComplex(op_x,op_y,ip_x,ip_y,n_samp,EFFTALLSCALE) != EDSP_OK){
        printf("FftComplex error\n");
    }
    FreeFft();
    for(i=0;i<n_samp;i++){
        printf("[%d] op_x=%d  op_y=%d  \n",i,op_x[i],op_y[i]);
    }
}
```

This frees the table used in FFT calculations. If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft.

(b) not-in-place real number FFT

Description:

- Format:

```
int FftReal (short op_x[], short op_y[], const short ip[],
                       long size, long scale)
```

- Parameters:

`op_x[]`	Real number component of positive output data
`op_y[]`	Imaginary number component of positive output data
`ip[]`	Real number input data
`size`	FFT size
`scale`	Scaling specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 8` •`size is not a power of 2` •`size > max_fft_size`

- Explanation of this function:

Executes a real number fast Fourier transform.

- Remarks:

size/2 positive output data is stored in op_x and op_y. Negative output data is the conjugate complex number of positive output data. In addition, as the values of output data of 0 and $F_s/2$ are real numbers, the real number output with $F_s/2$ is stored in op_y[0].
As this function performs not-in-place, provide input arrays and output arrays separately.
For details on allocation of complex number and real number data arrays, refer to “(2) Complex number data array format” and “(3) Real number data array format”.
Before calling on this function, call on InitFft, and initialize the rotation factor and max_fft_size.
For details on scaling, refer to “(4) Scaling”.
‘scale’ uses the lower $\log_2$ (size) bit.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define VLEN 64
#define TWOPI  6.28318530717959
/* global data declarations */
#pragma section X
short output_x[VLEN];
#pragma section Y
short output_y[VLEN];
#pragma section
void main()
{
    short i;
    int k;
    short input[VLEN];
    short output[VLEN];
/* generate two sinusoids */
    k = VLEN / 8;
    for (i = 0; i < VLEN; i++)
       input[i] = floor(16383 * cos(TWOPI * k * i / VLEN) + 0.5);
    k = VLEN * 3 / 8;
    for (i = 0; i < VLEN; i++)
        input[i] += floor(16383 * cos(TWOPI * k * i / VLEN) + 0.5);

/* do FFT */
    if (InitFft(VLEN) != EDSP_OK)
        printf("InitFft problem\n");
    if (FftReal(output_x, output_y, input, VLEN, EFFTALLSCALE) != EDSP_OK)
        printf("FftReal problem\n");
    FreeFft();
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Creation of data for FFT

FFT initialization function;

Initialization is performed for the number of data elements. This is required. The number of data elements is equal to the FFT data size, and must be a power of 2.

This frees the table used in FFT calculations. If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft.

(c) not-in-place inverse complex number FFT

Description:

- Format:

```
int IfftComplex (short op_x[], short op_y[],
                             const short ip_x[], const short ip_y[],
                             long size, long scale)
```

- Parameters:

`op_x[]`	Real number component of output data
`op_y[]`	Imaginary number component of output data
`ip_x[]`	Real number component of input data
`ip_y[]`	Imaginary number component of input data
`size`	Inverse FFT size
`scale`	Scaling specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 4` •`size is not a power of 2` •`size > max_fft_size`

- Explanation of this function:

Executes a complex number inverse fast Fourier transform.

- Remarks:

As this function performs not-in-place, provide input arrays and output arrays separately.
For details on allocation of complex number data arrays, refer to “(2) Complex number data array format”.
Before calling on this function, call on InitFft, and initialize the rotation factor and max_fft_size.
For details on scaling, refer to “(4) Scaling”.
‘scale’ uses the lower log2 (size) bit.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */
long ip_scale=8188;
#pragma section X
short ipi_x[MAX_IFFT_SIZE];          /* input array */
short opi_x[MAX_IFFT_SIZE];          /* normal output array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
short opi_y[MAX_IFFT_SIZE];
#pragma section
void main()
{
    int i,j;
    long scale;
    long max_size;
    max_size=MAX_IFFT_SIZE;/* data */

    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
        ipi_y[j] = sin(j * TWOPI/max_size) * ip_scale;
    }
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error end  \n");
    }
    else {
        if(FftInComplex(ipi_x, ipi_y, max_size,EFFTALLSCALE) != EDSP_OK){
            printf("FftInComplex err  end  \n");
        }
        for (j = 0; j < max_size; j++){
            opi_x[j]=0;
            opi_y[j]=0;
        }
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Creation of data for FFT (data used to execute FftComplex)

FFT initialization function;

Initialization is performed for the number of data elements. This is required. The number of data elements is equal to the FFT data size, and must be a power of 2.

This processing performs FFT calculations and uses the results as input values for an inverse FFT function; normally it is not necessary.

```
        if(IfftComplex(opi_x, opi_y, ipi_x, ipi_y, max_size,
                             EFFTALLSCALE)!= EDSP_OK){
            printf("IfftComplex err  end  \n");
        }
        for (j = 0; j < max_size; j++){
           printf("[%d]  opi_x=%d  op_y=%d \n",j, opi_x[j],opi_y[j]);
        }
        FreeFft();
    }
}
```

This frees the table used in FFT calculations. If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft.

(d) not-in-place real number inverse FFT

Description:

- Format:

```
int IfftReal (short op_x[], short scratch_y[],
                        const short ip_x[], const short ip_y[], long size, long
                        scale, int op_all_x)
```

- Parameters:

op_x[]	Real number output data
scratch_y[]	Scratch memory or real number output data
ip_x[]	Real number component of positive input data
ip_y[]	Imaginary number component of positive input data
size	Inverse FFT size
scale	Scaling specification
op_all_x	Allocation specification of output data

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	In any of the following cases •size < 8 •size is not a power of 2 •size > max_fft_size •op_all_x ≠ 0 or 1

- Explanation of this function:

Executes a real number inverse fast Fourier transform.

- Remarks:

Store size/2 positive input data in ip_x and ip_y. Negative input data is the conjugate complex number of positive input data. In addition, as the values of input data of 0 and F_S/2 are real numbers, store the real number input with F_S/2 in ip_y[0].|
The format of output data is specified with op_all_x. If op_all_x=1, all output data is stored in op_x. If op_all_x=0, the first size/2 output data is stored in op_x, and the remainder of the size/2 output data is stored in scratch_y.
As this function performs not-in-place, provide input arrays and output arrays separately.
For details on allocation of complex number and real number data arrays, refer to “(2) Complex number data array format” and “(3) Real number data array format”.
Store size/2 data in ip_x and ip_y respectively. size or size/2 data is stored in op_x depending on the value of op_all_x.
Before calling on this function, call on InitFft, and initialize the rotation factor and max_fft_size.
For details on scaling, refer to “(4) Scaling”.
‘scale’ uses the lower $\log_2$ (size) bit.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
```

Include header

```
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */
long ip_scale=8188;
#pragma section X
short ipi_x[MAX_IFFT_SIZE];           /* input array */
short opi_x[MAX_IFFT_SIZE];           /* normal output array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
short opi_y[MAX_IFFT_SIZE];
#pragma section
```

Variables placed in X or Y memory are defined by a pragma section within the section.

Creation of data for FFT (data used to execute FftReal)

```
void main()
{
    int i,j;
    long scale;
    long max_size;
    max_size=MAX_IFFT_SIZE;/* data */
    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
    }
    if (InitFft(max_size) != EDSP_OK){
        printf("InitFft error end  \n");
    }
    else {
        if(FftInReal(ipi_x, ipi_y, max_size,EFFTALLSCALE,1) != EDSP_OK){
            printf("FftInReal err  end  \n");
        }
if(IfftReal(opi_x, opi_y, ipi_x, ipi_y, max_size, EFFTALLSCALE,1)!=
   EDSP_OK){
            printf("IfftReal err end  \n");
        }
        for (j = 0; j < max_size; j++){
           printf("[%d]  opi_x=%d  op_y=%d \n",j, opi_x[j],opi_y[j]);
```

FFT initialization function;

Initialization is performed for the number of data elements. This is required. The number of data elements is equal to the FFT data size, and must be a power of 2. Also required for inverse FFT.

This processing performs FFT calculations and uses the results as input values for an inverse FFT function; normally it is not necessary.

```
        }
        FreeFft();
    }
}
```

This frees the table used in FFT calculations. If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft.

(e) in-place complex number FFT

Description:

- Format:

```
int FftInComplex (short data_x[], short data_y[],
                                long size, long scale)
```

- Parameters:

`data_x[]`	Real number component of input data
`data_y[]`	Imaginary number component of input and output data
`size`	FFT size
`scale`	Scaling specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 4` •`size is not a power of 2` •`size > max_fft_size`

- Explanation of this function:

Executes an in-place complex number fast Fourier transform.

- Remarks:

For details on allocation of complex number data arrays, refer to “(2) Complex number data array format”.
Before calling on this function, call on InitFft, and initialize the rotation factor and max_fft_size.
For details on scaling, refer to “(4) Scaling”.
‘scale’ uses the lower $\log_2$ (size) bit.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_FFT_SAMP  64
#define TWOPI 6.283185307 /* data */
long ip_scale=0xffffffff;


#pragma section X
short ip_x[MAX_FFT_SAMP];
#pragma section Y
short ip_y[MAX_FFT_SAMP];
#pragma section
void main()
{
    int i,j;
    long max_size;
    long n_samp;
    n_samp=MAX_FFT_SAMP;
    max_size=n_samp;/* data */
    for (j = 0; j < n_samp; j++){
        ip_x[j] = cos(j * TWOPI/n_samp) * ip_scale;
        ip_y[j] = sin(j * TWOPI/n_samp) * ip_scale;
    }
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error\n");
    }
    if(FftInComplex(ip_x, ip_y, n_samp,EFFTALLSCALE ) != EDSP_OK){
        printf("FftInComplex error\n");
    }
    FreeFft();
    for(i=0;i<max_size;i++){
        printf("[%d] ip_x=%d  ip_y=%d  \n",i,ip_x[i],ip_y[i]);
    }
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Data creation for FFT

FFT initialization function;

Initialization is performed for the number of data elements. This is required. The number of data elements is equal to the FFT data size, and must be a power of 2.

This frees the table used in FFT calculations. If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft.

(f) in-place real number FFT

Description:

- Format:

```
int FftInReal (short data_x[], short data_y[], long size,
                          long scale, int ip_all_x)
```

- Parameters:

`data_x[]`	Real number data when input, and real number component of the positive output data when output
`data_y[]`	Real number data or unused for input, and imaginary number component of the positive output data when output
`size`	FFT size
`scale`	Scaling specification
`ip_all_x`	Allocation specification of input data

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 8` •`size is not a power of 2` •`size > max_fft_size` •`ip_all_x ≠ 0 or 1`

- Explanation of this function:

Executes an in-place real number fast Fourier transform.

- Remarks:

The format of input data is specified with ip_all_x. If ip_all_x=1, all input data is removed from data_x. If ip_all_x=0, the first half of size/2 input data is removed from data_x, and the second half of size/2 input data is removed from data_y.
After execution of this function, size/2 positive output data is stored in data_x and data_y. Negative output data is the conjugate complex number of positive output data. In addition, as the values of output data of 0 and $F_S/2$ are real numbers, the real number output with $F_S/2$ is stored in data_y[0].
For details on allocation of complex number and real number data arrays, refer to “(2) Complex number data array format” and “(3) Real number data array format”.
Store size/2 data in data_y. size or size/2 data is stored in data_x depending on the value of ip_all_x.
Before calling on this function, call on InitFft, and initialize the rotation factor and max_fft_size.
For details on scaling, refer to “(4) Scaling”.
‘scale’ uses the lower $\log_2$ (size) bit.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
```

Include header

```
#define MAX_FFT_SAMP  64
#define TWOPI 6.283185307 /* data */
long ip_scale=8188;
/*long ip_scale=0xffffffff;*/

#pragma section X
short ip_x[MAX_FFT_SAMP];
#pragma section Y
short ip_y[MAX_FFT_SAMP];
#pragma section
```

Variables placed in X or Y memory are defined by a pragma section within the section.

```
void main()
{
    int i,j;
    long max_size;
    long n_samp;
    int ip_all_x;
    n_samp=MAX_FFT_SAMP;
    max_size=n_samp;/* data */

    for (j = 0; j < n_samp; j++){
        ip_x[j] = cos(j * TWOPI/n_samp) * ip_scale;
        ip_y[j] = 0;
    }
```

Data creation for FFT

```
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error\n");
    }
```

FFT initialization function;

Initialization is performed for the number of data elements. This is required. The number of data elements is equal to the FFT data size, and must be a power of 2.

```
    ip_all_x = 1;
    if(FftInReal(ip_x, ip_y, n_samp,EFFTALLSCALE ,ip_all_x) != EDSP_OK){
        printf("FftInReal error\n");
    }
    FreeFft();
    for(i=0;i<max_size;i++){
        printf("[%d] ip_x=%d  ip_y=%d  \n",i,ip_x[i],ip_y[i]);
    }
}
```

This frees the table used in FFT calculations. If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft.

(g) in-place complex number inverse FFT

Description:

- Format:

```
int IfftInComplex (short data_x[], short data_y[],
                                long size, long scale)
```

- Parameters:

`data_x[]`	Real number component of input data
`data_y[]`	Imaginary number component of input and output data
`size`	Inverse FFT size
`scale`	Scaling specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 4` •`size is not a power of 2` •`size > max_fft_size`

- Explanation of this function:

Executes an in-place complex number inverse fast Fourier transform.

- Remarks:

For details on allocation of complex number data arrays, refer to "(2) Complex number data array format".
Before calling on this function, call on InitFft, and initialize the rotation factor and max_fft_size.
For details on scaling, refer to "(4) Scaling".
'scale' uses the lower $\log_2$ (size) bit.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
```

Include header

```
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */

long ip_scale=8188;
#pragma section X
short ipi_x[MAX_IFFT_SIZE];                /* input array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
#pragma section
```

Variables placed in X or Y memory are defined by a pragma section within the section.

```
void main()
{
    int i,j;
    long scale;
    long max_size;

    max_size=MAX_IFFT_SIZE;/* data */
    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
        ipi_y[j] = sin(j * TWOPI/max_size) * ip_scale;
    }
```

Data creation for FFT (data used as input for FftInComplex)

```
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error end  \n");
    }
```

FFT initialization function;
Initialization is performed for the number of data elements. This is required. The number of data elements is equal to the FFT data size, and must be a power of 2. Also required for inverse FFT.

```
    else {
        if(FftInComplex(ipi_x, ipi_y, max_size,EFFTALLSCALE) != EDSP_OK){
            printf("FftInComplex err  end  \n");
        }
        if(IfftInComplex(ipi_x, ipi_y, max_size,EFFTALLSCALE) != EDSP_OK){
            printf("IfftInComplex err  end  \n");
        }
        for (j = 0; j < max_size; j++){
```

This processing performs FFT calculations and uses the results as input values for an inverse FFT function; normally it is not necessary.

```
           printf("[%d]  ipi_x=%d  ip_y=%d \n",j, ipi_x[j],ipi_y[j]);
        }
        FreeFft();
    }
}
```

This frees the table used in FFT calculations. If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft.

(h) in-place real number inverse FFT

Description:

- Format:

```
 int IfftInReal (short data_x[], short data_y[], long size,
                            long scale, int op_all_x)
```

- Parameters:

`data_x[]`	Real number component of positive input data when input, and real number data when output
`data_y[]`	Imaginary number component of positive input data when input, and real number data when output or unused
`size`	Inverse FFT size
`scale`	Scaling specification
`op_all_x`	Allocation specification of output data

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 8` •`size is not a power of 2` •`size > max_fft_size` •`op_all_x ≠ 0 or 1`

- Explanation of this function:

Executes an in-place real number inverse fast Fourier transform.

- Remarks:

Store size/2 positive input data in data_x and data_y. Negative input data is the conjugate complex number of positive input data. In addition, as the values of input data of 0 and $F_S/2$ are real numbers, store the real number input with $F_S/2$ in data_y[0].
The format of output data is specified with op_all_x. If op_all_x=1, all output data is stored in data_x. If op_all_x=0, the first half of the size/2 output data is stored in data_x, and the second half of the size/2 output data is stored in data_y.
For details on allocation of complex number and real number data arrays, refer to "(2) Complex number data array format" and "(3) Real number data array format".
Store size/2 data in data_y. size or size/2 data is stored in data_x depending on the value of op_all_x.
Before calling on this function, call on InitFft, and initialize the rotation factor and max_fft_size.
For details on scaling, refer to "(4) Scaling".
'scale' uses the lower $\log_2$ (size) bit.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */

long ip_scale=8188;

#pragma section X
short ipi_x[MAX_IFFT_SIZE];          /* input array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
#pragma section
void main()
{
    int i,j;
    long scale;
    long max_size;
    max_size=MAX_IFFT_SIZE;/* data */
    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
    }
    if(InitFft(max_size) != EDSP_OK){
           printf("InitFft error end  \n");
    }
    else {
        if(FftInReal(ipi_x, ipi_y, max_size,EFFTALLSCALE,1) != EDSP_OK){
            printf("FftInReal err  end  \n");
        }
        if(IfftInReal(ipi_x, ipi_y, max_size, EFFTALLSCALE,1) != EDSP_OK){
            printf("IfftInReal err  end  \n");
        }
        for (j = 0; j < max_size; j++){
           printf("[%d]  ipi_x=%d  ip_y=%d \n",j, ipi_x[j],ipi_y[j]);
        }
        FreeFft();
    }
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Data creation for FFT (data used as input for FftInReal)

FFT initialization function;

Initialization is performed for the number of data elements. This is required. The number of data elements is equal to the FFT data size, and must be a power of 2. Also required for inverse FFT.

This frees the table used in FFT calculations. If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft.

(i) Logarithmic absolute value

Description:

- Format:

```
 int LogMagnitude (short output[], const short ip_x[],
                               const short ip_y[], long no_elements,
                               float fscale)
```

- Parameters:

`output[]`	Real number output z
`ip_x[]`	Input real number component x
`ip_y[]`	Input imaginary number component y
`no_elements`	Number of output data elements N
`fscale`	Output scaling coefficient

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_elements < 1` •`no_elements > 32767` •$|\texttt{fscale}| > 2^{15}/(10\log_{10}2^{31})$

- Explanation of this function:

Calculates the logarithmic absolute value of complex number input data in decibel units, and writes the scaling results in the output array.

- Remarks:

$z(n)=10\text{fscale} \cdot \log_{10}(x(n)^2+y(n)^2) \quad 0 \le n < N$

For details on allocation of complex number data arrays, refer to “(2) Complex number data array format”.

Example of use:

```
#include <stdio.h>
#include <math.h>
#include <ensigdsp.h>
#define MAX_IFFT_SIZE  16
#define TWOPI 6.283185307 /* data */
long ip_scale=8188;
#pragma section X
short ipi_x[MAX_IFFT_SIZE];          /* input array */
#pragma section Y
short ipi_y[MAX_IFFT_SIZE];
#pragma section
void main()
{
    int i,j;
    long scale;
    long max_size;
    short output[MAX_IFFT_SIZE];
    max_size=MAX_IFFT_SIZE;/* data */

    for (j = 0; j < max_size; j++){
        ipi_x[j] = cos(j * TWOPI/max_size) * ip_scale;
    }
    if(InitFft(max_size) != EDSP_OK){
        printf("InitFft error end  \n");
    }
    else {
        if(FftInReal(ipi_x, ipi_y, max_size,EFFTALLSCALE,1) != EDSP_OK){
            printf("FftInReal err  end  \n");
        }
        if(LogMagnitude(output, ipi_x,ipi_y, max_size/2, 2) != EDSP_OK){
            printf("LogMagnitude err  end  \n");
        }
        for (j = 0; j < max_size/2; j++){
           printf("[%d]  output=%d  \n",j, output[j]);
        }
        FreeFft();
    }
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Data creation for FFT

FFT function;
Creates data used by the LogMagnitude function.

This frees the table used in FFT calculations.

If this is not done, memory resources are wasted. If FFT is to be performed again using the same number of data elements, the FFT function is used again without executing FreeFft. This is not directly related to LogMagnitude.

(j) Rotation factor generation

Description:

- Format:
  `int InitFft (long max_size)`

- Parameters:

`max_size`	Maximum size of the required FFT

- Returned value:

`EDSP_OK`	Successful
`EDSP_NO_HEAP`	The memory space that can be obtained with malloc is insufficient
`EDSP_BAD_ARG`	In any of the following cases •`max_size < 2` •`max_size is not a power of 2` •`max_size > 32,768`

- Explanation of this function:

Generates the rotation factor (1/4 size) to be used by the FFT function.

- Remarks:

The rotation factor is stored in the memory obtained by malloc.
When the rotation factor is generated, the max_fft_size global variable is updated. max_fft_size shows the maximum capacity size of the FFT.
Be sure to call on this function once before calling on the first FFT function.
Make max_size 8 or more.
The rotation factor is generated by the conversion size specified by max_size. Use the same rotation factor when executing a FFT function with a smaller size than max_size.
The address of the rotation factor is stored inside the internal variable. Do not access this with the user program.
This function is not reentrant.

(k) Rotation factor release

Description:

- Format:
  `void FreeFft (void)`

- Parameters:

None

- Returned value:

None

- Explanation of this function:

Releases the memory used to store the rotation factors.

- Remarks:

Make the max_fft_size global variable 0. When executing the FFT function again after executing FreeFft, be sure to execute InitFft first.
This function is not reentrant.

3.13.5 Window Functions

(1) List of functions

Table 3.30 DSP Library Function List (Window Functions)

No.	Type	Function Name	Description
1	Blackman window	GenBlackman	Generates a Blackman window.
2	Hamming window	GenHamming	Generates a Hamming window.
3	Hanning window	GenHanning	Generates a Hanning window.
4	Triangular window	GenTriangle	Generates a triangular window.

(2) Explanation of each function

(a) Blackman window

Description:

- Format:
  `int GenBlackman (short output[], long win_size)`

- Parameters:

`output[]`	Output data W(n)
`win_size`	Window size N

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	win_size ≤ 1

- Explanation of this function:

Generates a Blackman window, and outputs to output.

- Remarks:

Use VectorMult when applying this window to actual data.
The function to be used is shown below.

$$W(n) = \left(2^{15} - 1\right)\left[0.42 - 0.5\cos\left(\frac{2\pi n}{N}\right) + 0.08\cos\left(\frac{4\pi n}{N}\right)\right] \quad 0 \leq n < N$$

Example of use:

```
#include <stdio.h>          } Include header
#include <ensigdsp.h>
#define MAXN 10
void main()
{
    int i;
    long len;
    short output[MAXN];
    len=MAXN ;
    if(GenBlackman(output, len) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    for(i=0;i<len;i++){
        printf("output=%d \n",output[i]);
    }
}
```

(b) Hamming window

Description:

- Format:
  `int GenHamming (short output[], long win_size)`

- Parameters:

`output[]`	Output data W(n)
`win_size`	Window size N

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	win_size ≤ 1

- Explanation of this function:

Generates a Hamming window, and outputs to output.

- Remarks:

Use VectorMult when applying this window to actual data.
The function to be used is shown below.

$$W(n) = \left(2^{15} - 1\right)\left[0.54 - 0.46\cos\left(\frac{2\pi n}{N}\right)\right] \quad 0 \le n < N$$

Example of use:

```
#include <stdio.h>          Include header
#include <ensigdsp.h>
#define MAXN  10
void main()
{
    int i;
    long len;
    short output[MAXN];
    len=MAXN ;
    if(GenHamming(output, len) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    for(i=0;i<len;i++){
        printf("output=%d \n",output[i]);
    }
}
```

(c) Hanning window

Description:

- Format:
  `int GenHanning (short output[], long win_size)`

- Parameters:

`output[]` Output data W(n)

`win_size` Window size N

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	win_size ≤ 1

- Explanation of this function:

Generates a Hanning window, and outputs to output.

- Remarks:

Use VectorMult when applying this window to actual data.

The function to be used is shown below.

$$W(n) = \left(\frac{2^{15} - 1}{2}\right)\left[1 - \cos\left(\frac{2\pi n}{N}\right)\right] \quad 0 \le n < N$$

Example of use:

```
#include <stdio.h>       } Include header
#include <ensigdsp.h>
#define MAXN  10
void main()
{
    int i;
    long len;
    short output[MAXN];
```

```
     len=MAXN ;
     if(GenHanning(output, len) != EDSP_OK){
         printf("EDSP_OK not returned\n");
     }
     for(i=0;i<len;i++){
         printf("output=%d \n",output[i]);
     }
 }
```

(d) Triangular window

Description:

- Format:
  `int GenTriangle (short output[], long win_size)`

- Parameters:

`output[]`	Output data W(n)
`win_size`	Window size N

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	win_size ≤ 1

- Explanation of this function:

Generates a triangular window, and outputs to output.

- Remarks:

Use VectorMult when applying this window to actual data.
The function to be used is shown below.

$$W(n) = \left(2^{15} - 1\right)\left[1 - \left|\frac{2n - N + 1}{N + 1}\right|\right] \quad 0 \le n < N$$

Example of use:

```
 #include <stdio.h>       } Include header
 #include <ensigdsp.h>
 #define MAXN 10
 void main()
 {
     int i;
     long len;
     short output[MAXN];
     len=MAXN ;
```

```
    if(GenTriangle(output, len) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    for(i=0;i<len;i++){
        printf("output=%d \n",output[i]);
    }
}
```

3.13.6 Filters

(1) List of functions

Table 3.31 DSP Library Function List (Filters)

No.	Type	Function Name	Description
1	FIR	Fir	Performs finite impulse-response filter processing
2	FIR for single data elements	Fir1	Performs finite impulse-response filter processing for a single data element
3	IIR	Iir	Performs infinite impulse-response filter processing
4	IIR for single data elements	Iir1	Performs infinite impulse-response filter processing for a single data element
5	Double precision IIR	Diir	Performs double-precision infinite impulse-response filter processing
6	Double precision IIR for single data elements	Diir1	Performs double-precision infinite impulse-response filter processing for a single data element
7	Adaptive FIR	Lms	Performs adaptive FIR filter processing
8	Adaptive FIR for single data elements	Lms1	Performs adaptive FIR filter processing for a single data element
9	FIR work space allocation	InitFir	Allocates a work space for use by the FIR filter
10	IIR work space allocation	InitIir	Allocates a work space for use by the IIR filter
11	Double precision IIR work space allocation	InitDIir	Allocates a work space for use by the DIIR filter
12	Adaptive FIR work space allocation	InitLms	Allocates a work space for use by the LMS filter
13	FIR work space release	FreeFir	Releases the work space allocated by InitFir
14	IIR work space release	FreeIir	Releases the work space allocated by InitIir
15	Double precision IIR work space release	FreeDIir	Releases the work space allocated by InitDIir
16	Adaptive FIR work space release	FreeLms	Releases the work space allocated by InitLms

Note: When using any of these functions, include filt_ws.h only once in the user program.

(2) Coefficient scaling

When executing filter processing, there is a possibility that saturation or quantization noise may occur. These can be suppressed to the minimum by performing scaling of these filter coefficients. However, it is necessary to perform scaling giving careful consideration to the impact of saturation and quantization noise. If the coefficient is too large there is a possibility that saturation may occur. If it is too small, quantization noise may occur.

With the FIR (finite impulse response) filter, saturation will not occur if the filter coefficient is set so that the following equation is applied.

coeff[i] ≠ H’8000 (for all instances of i)

Σ|coeff| < 224

res_shift = 24

coeff is the filter coefficient, and res_shift is the number of bits shifted to the right at output.

However, when there are many input signals, even if a smaller res_shift value is used (or a bigger coeff value), the possibility of saturation is slight, and quantization noise can be reduced by a wide margin. In addition, if there is a possibility that the input value includes H’8000, set all coeff values to be in the range of H’8001 to H’7FFF.

The IIR (infinite impulse response) filter has a recursive structure. For this reason, the scaling method explained above is not suitable.

The LMS (least mean square) adaptive filter is the same as the FIR filter. However, when adapting the coefficient, there may be cases where saturation occurs. In this case, make the settings so that H’8000 is not included in the coefficient.

(3) Work space

With digital filters, there is information that must be saved between one process and the next. This information is stored in memory that can be accessed with the minimum of overhead. With this library, the Y-RAM area is used as the work space. Before executing filter processing, call on the Init function and initialize the work space.

The work space memory is accessed by the library function. Do not access the work space directly from the user program.

(4) Using memory

In order to use SH-DSP efficiently, allocate filter coefficients to X memory. Input and output data can be allocated to arbitrary memory segments.

Allocate filter coefficients to X memory using the #pragma section instruction.

Each filter is allocated to the work space from the global buffer using the Init function. The global buffer is allocated to Y memory.

(5) Explanation of each function

(a) FIR

Description:

- Format:

```
int Fir (short output[], const short input[], long no_samples,
                const short coeff[], long no_coeffs, int res_shift,
                short *workspace)
```

- Parameters:

output[]	Output data y
input[]	Input data x
no_samples	Number of input data elements N
coeff[]	Filter coefficient h
no_coeffs	Number of coefficients (filter length) K
res_shift	Right shift applied to each output.
workspace	Pointer to the work space

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	In any of the following cases •no_samples < 1 •no_coeffs ≤ 2 •res_shift < 0 •res_shift > 25

- Explanation of this function:

Performs finite impulse-response (FIR) filter processing

- Remarks:

The latest input data is saved in the work space. The results of filter processing of input are written to output.

$$y(n) = \left[\sum_{k=0}^{K-1} h(k)\ x(n-k)\right] \cdot 2^{-res\_shift}$$

The results of multiply-and-accumulate operations are saved as 39 bits. Output y(n) is the lower 16 bits fetched from the res_shift bit right shifted results. When an overflow occurs, this is the positive or negative maximum value.
For details on coefficient scaling, refer to “(2) Coefficient scaling”.
Before calling on this function, call on InitFir, and initialize the work space of the filter.
If the same array is specified for output as for input, input will be overwritten.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
```

Include header

```
#define NFN 8 /* number of functions */
#define FIL_COUNT   32  /* number of data objects */
#define N   32

#pragma section X
static short coeff_x[FIL_COUNT];
```

Set the filter coefficients in X memory. Since Y memory is used by the library as the work area to calculate filter coefficients, Y memory should not be used.

```
#pragma section
short data[FIL_COUNT] =  {
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,};
short coeff[8] =  {
         0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000,};

void main()
{
    short *work, i;
    short output[N];
    int nsamp, ncoeff, rshift;
    /* copy coeffs into X RAM */
    for(i=0;i<NFN;i++) {
        coeff_x[i] = coeff[i];/* Sets coefficient */
    }
```

Set filter coefficients in X memory as variables.

```
    for (i = 0; i < N; output[i++] = 0) ;
    ncoeff = NFN;/* Sets the number of coefficients  */
    nsamp = FIL_COUNT;/* set number of samples */
    rshift = 12;
    if (InitFir(&work, ncoeff) != EDSP_OK){
        printf("Init Problem\n");
    }
```

Filter initialization:
(1) Work area address
(2) Number of coefficients
This is necessary before Fir function execution. The work area in Y memory uses (number of coefficients)*2+8 bytes.

```
    if(Fir(output, data, nsamp, coeff_x, ncoeff, rshift, work) != EDSP_OK){
        printf("Fir Problem\n");
    }
    if (FreeFir(&work, ncoeff) != EDSP_OK){
        printf("Free Problem\n");
    }
    for(i=0;i<nsamp;i++){
        printf("#%2d output:%6d \n",i,output[i]);
    }
}
```

The FreeFir function frees the work area used for Fir calculations; The FreeFir function must always be performed after Fir execution. If this function is not executed, memory resources are wasted.

(b) FIR for single data elements

Description:

- Format:

```
int Fir1 (short *output, short input, const short coeff[],
                    long no_coeffs, int res_shift, short *workspace)
```

- Parameters:

output	Pointer to output data y(n)
input	Input data x(n)
coeff[]	Filter coefficient h
no_coeffs	Number of coefficients (filter length) K
res_shift	Right shift applied to each output.
workspace	Pointer to the work space

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	In any of the following cases •no_coeffs ≤ 2 •res_shift < 0 •res_shift > 25

- Explanation of this function:

Performs finite impulse-response (FIR) filter processing for single data elements.

- Remarks:

The latest input data is saved in the work space. The results of filter processing of input are written to *output.

$$y(n) = \left[\sum_{k=0}^{K-1} h(k)\ x(n-k)\right] \cdot 2^{-res\_shift}$$

The results of multiply-and-accumulate operations are saved as 39 bits. Output y(n) is the lower 16 bits fetched from the res_shift bit right shifted results. When an overflow occurs, this is the positive or negative maximum value.
For details on coefficient scaling, refer to “(2) Coefficient scaling”.
Before calling on this function, call on InitFir, and initialize the work space of the filter.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
```

Include header

```
#define NFN 8  /* number of functions */
#define MAXSH 25
#define N   32
#pragma section X
static short coeff_x[NFN];
```

Set the filter coefficients in X memory. Since Y memory is used by the library as the work area to calculate filter coefficients, Y memory should not be used.

```
#pragma section
short data[32] =  {
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400};
short coeff[8] =  {
         0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000};
void main()
{
    short *work, i;
    short output[N];
    int ncoeff, rshift;

    /* copy coeffs into X RAM */
    for(i=0;i<NFN;i++) {
        coeff_x[i] = coeff[i];/* Sets coefficient */
    }
```

Set filter coefficients in X memory as variables.

```
    for (i = 0; i < N; output[i++] = 0) ;
    rshift = 12;
    ncoeff = NFN;/* Sets the number of coefficients */
    if (InitFir(&work, NFN) != EDSP_OK){
        printf("Init Problem\n");
    }
```

Filter initialization:
(1) Work area address
(2) Number of coefficients
This is necessary before Fir1 function execution. The work area in Y memory uses (number of coefficients)*2+8 bytes.

Fir1 means that the number of data elements that are set to the Fir function is 1. When executing Fir1 multiple times, the Init Fir and FreeFir functions must be executed before and after the for statement respectively.

```
    for(i=0;i<N;i++) {
        if(Fir1(&output[i], data[i], coeff_x, ncoeff, rshift, work) !=
    EDSP_OK){
            printf("Fir1 Problem\n");
        }
        printf(" output[%d]=%d \n",i,output[i]);
    }
    if (FreeFir(&work, NFN) != EDSP_OK){
        printf("Free Problem\n");
    }
}
```

Fir1 means that the number of data elements that are set to the Fir function is 1. When executing Fir1 multiple times, the Init Fir and FreeFir functions must be executed before and after the for statement respectively.

(c) IIR

Description:

- Format:

```
 int Iir (short output[], const short input[], long no_samples,
                const short coeff[], long no_sections, short *workspace)
```

- Parameters:

`output[]`	Output data y_{K-1}
`input[]`	Input data x_0
`no_samples`	Number of input data elements N
`coeff[]`	Filter coefficient
`no_sections`	Number of secondary filter sections K
`workspace`	Pointer to the work space

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_samples < 1` •`no_sections < 1` •a_{0k} `< 0` •a_{0k} `> 16`

- Explanation of this function:

Performs infinite impulse-response (IIR) filter processing.

- Remarks:

This filter is configured whereby a secondary filter, or biquad, is linked to the K number tandem. Additional scaling is performed with the output of each biquad. The filter coefficient is specified with a signed 16-bit fixed point number
The output of each biquad is subject to the following equation.

$d_k(n)=[a_{1k}d_k(n-1)+a_{2k}d_k(n-2)+2_{15}x(n)] \cdot 2^{-15}$

$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)] \cdot 2^{-a0k}$

The input x_k (n) for k is the output y_{k-1} (n) of the previous section. The input of the first section (k=0) is read from input.
The output of the last section (k=K-1) is written to output.
Set coeff in the following order of coefficients.

a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}

The a_{0k} item is the number of bits for right shift to be performed on the output of the biquad for k.
Each biquad performs a 32-bit multiply-and-accumulate operation. The output of each biquad is the lower 16 bits fetched

from the 15-bit or a_{0k} right shifted results. When an overflow occurs, this is the positive or negative maximum value.
Before calling on this function, call on InitIir, and initialize the work space of the filter.
If the same array is specified for output as for input, input will be overwritten.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
```

Include header

```
#define K  4
#define NUMCOEF  (6*K)
#define N 50
#pragma section X
static short coeff_x[NUMCOEF];
```

Set the filter coefficients in X memory. Since Y memory is used by the library as the work area to calculate filter coefficients, Y memory should not be used.

```
#pragma section
static short coeff[24] = {15, 19144, -7581,  5301, 10602,  5301,
                               15, -1724,-23247, 13627, 27254, 13627,
                               15, 19144, -7581,  5301, 10602,  5301,
                               15, -1724,-23247, 13627, 27254, 13627};
```

Six filter coefficients should be set in one section. The leading element in a section is the number of right-shifts, and is not a filter coefficient.

```
static short input[50] = {32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000 };
void main()
{
    short *work, i;
    short output[N];

    for(i=0;i<NUMCOEF;i++) {
```

Set filter coefficients in X memory as variables.

```
          coeff_x[i] = coeff[i];
      }
      if (InitIir(&work, K) != EDSP_OK){
          printf("Init Problem\n");
      }
      if (Iir(output, input, N, coeff_x, K, work) != EDSP_OK){
          printf("EDSP_OK not returned\n");
      }
      if (FreeIir(&work, K) != EDSP_OK){
          printf("Free Problem\n");
      }
      for(i=0;i<N;i++){
          printf("#%2d output:%6d \n",i,output[i]);
      }
  }
```

Filter initialization:
(1) Work area address
(2) Number of filter sections
This is necessary before Iir function execution. The work area in Y memory uses ((number of filter sections)*2*2) bytes.

The FreeIir function frees the work area used for Iir calculations; The FreeIir function must always be performed after Iir execution. If this function is not executed, memory resources are wasted.

(d) IIR for single data elements

Description:

- Format:

```
int Iir1 (short *output, short input, const short coeff[],
                    long no_sections, short *workspace)
```

- Parameters:

`output`	Pointer to output data $y_{K-1}(n)$
`input`	Input data x_0 (n)
`coeff[]`	Filter coefficient
`no_sections`	Number of secondary filter sections K
`workspace`	Pointer to the work space

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_sections < 1` •a_{ok} `< 0` •a_{ok} `> 16`

- Explanation of this function:

Performs infinite impulse-response (IIR) filter processing for single data elements.

- Remarks:

This filter is configured whereby a secondary filter, or biquad, is linked to the K number tandem. Additional scaling is performed with the output of each biquad. The filter coefficient is specified with a signed 16-bit fixed point number. The output of each biquad is subject to the following equation.

$d_k(n)=[a_{1k}d_k(n-1)+a_{2k}d_k(n-2)+2^{15}x(n)] \cdot 2^{-15}$
$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)] \cdot 2^{-a0k}$
The input x_k (n) for k is the output y_{k-1} (n) of the previous section. The input of the first section (k=0) is read from input.
The output of the last section (k=K-1) is written to output.
Set coeff in the following order of coefficients.
a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}
The a0k item is the number of bits for right shift to be performed on the output of the biquad for k.
Each biquad performs a 32-bit saturation operation. The output of each biquad is the lower 16 bits fetched from the 15-bit or a_{0k} right shifted results. When an overflow occurs, this is the positive or negative maximum value.
Before calling on this function, call on InitIir, and initialize the work space of the filter.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
#define K   4
#define NUMCOEF  (6*K)
#define N   50
#pragma section X
static short coeff_x[NUMCOEF];
#pragma section
static short coeff[24] = {15, 19144, -7581,  5301, 10602,  5301,
                               15, -1724,-23247, 13627, 27254, 13627,
                               15, 19144, -7581,  5301, 10602,  5301,
                               15, -1724,-23247, 13627, 27254, 13627};

static short input[50] = {32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000,
                    32000, 32000, 32000, 32000, 32000 };
short keisu[5]={ 1,2,20,4,5 };
```

Include header

Set the filter coefficients in X memory. Since Y memory is used by the library as the work area to calculate filter coefficients, Y memory should not be used.

Six filter coefficients should be set in one section. The leading element in a section is the number of right-shifts, and is not a filter coefficient.

```
void main()
{
    short *work, i;
    short output[N];
    for(i=0;i<NUMCOEF;i++) {
        coeff_x[i] = coeff[i];
 }
    if (InitIir(&work, K) != EDSP_OK){
        printf("Init Problem\n");
    }
    for(i=0;i<N;i++){
        if (Iir1(&output[i], input[i], coeff_x, K, work) != EDSP_OK){
            printf("EDSP_OK not returned\n");
        }
        printf("output[%d]:%d \n" ,i,output[i]);
    }
    if (FreeIir(&work, K) != EDSP_OK){
        printf("Free Problem\n");
    }
}
```

Set filter coefficients in X memory as variables.

Filter initialization:
(1) Work area address
(2) Number of filter sections
This is necessary before Iir1 function execution. The work area in Y memory uses (number of filter sections)*2*2 bytes.

Iir1 means that the number of data elements that are set to the Iir function is 1. When executing Iir1 multiple times, the Init Iir and FreeIir functions must be executed before and after the for statement.

(e) Double precision IIR

Description:

- Format:

```
int DIir (short output[], const short input[], long no_samples,
                  const long coeff[], long no_sections, long *workspace)
```

- Parameters:

`output[]`	Output data y_{K-1}
`input[]`	Input data x
`no_samples`	Number of input data elements N
`coeff[]`	Filter coefficient
`no_sections`	Number of secondary filter sections K
`workspace`	Pointer to the work space

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_samples < 1` •`no_sections < 1` •a_{0k} `< 3` •`k < K-1` and a_{0k} `> 32` •`k = K-1` and a_{0k} `> 48`

- Explanation of this function:

Performs double-precision infinite impulse-response filter processing

- Remarks:

This filter is configured whereby a secondary filter, or biquad, is linked to the K number tandem. Additional scaling is performed with the output of each biquad. The filter coefficient is specified with a signed 32-bit fixed point number.
The output of each biquad is subject to the following equation.
$d_k(n)=[a_{1k}d_k(n-1)+a_{2k}d_k(n-2)+2^{29}x(n)] \cdot 2^{-31}$
$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)] \cdot 2^{-a0k} \cdot 2^{2}$
The input x_k (n) for k is the output y_{k-1} (n) of the previous section. The input of the first section (k=0) is read from the 16-bit left shifted value of input. The output of the last section (k=K-1) is written to output.
Set coeff in the following order of coefficients.
a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}
The a_{0k} item is number of bits for right shift to be performed on the output of the biquad for k.
DIir differs from Iir in that the filter coefficient is specified with a 32-bit value rather than a 16-bit value. The results of multiply-and-accumulate operations are saved as 64 bits. The output of intermediate stages is the lower 32 bits fetched from the a_{0k} bit right shifted results. When an overflow occurs, this is the positive or negative maximum value. At the last stage, the lower 16 bits are fetched from the a_{0k-1} bit right shifted results. When an overflow occurs, this is the positive or negative maximum value.
Before calling on this function, call on InitDIir, and initialize the work space of the filter.
The delayed node d_k (n) is rounded off to 30 bits, and when an overflow occurs, this is the positive or negative maximum value.
When using DIir, specify the coefficient with a signed 32-bit fixed point number. In this case, when a_{0k} is k < K-1 set it as 31, and when k=K-1 set it as 47.
As the speed of execution of Iir is faster than that of DIir, if double precision calculation is required, use Iir.
If the same array is specified for output as for input, input will be overwritten.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <filt_ws.h>
#include <ensigdsp.h>
```

Include header

```
#define K  5
#define NUMCOEF  (6*K)
#define N  50
#pragma section X
static long coeff_x[NUMCOEF];
```

Set the filter coefficients in X memory.

Since Y memory is used by the library as the work area to calculate filter coefficients, Y memory should not be used.

```
#pragma section
static long coeff[60] =
 {31,1254686956, -496866304, 347415747, 694831502, 347415746,
  31,-113001278,-1523568505, 893094203,1786188388, 893094206,
  31,1254686956, -496866304, 347415747, 694831502, 347415746,
  31,-113001278,-1523568505, 893094203,1786188388, 893094206,
  47,1254686956, -496866304, 347415747, 694831502, 347415746};
```

Six filter coefficients should be set in one section. The leading element in a section is the number of right-shifts, and is not a filter coefficient.

The number of right-shifts is 31 except for the last section; the last section is 47.

```
static short input[100] = {
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000,
                           32000, 32000, 32000, 32000, 32000 };

void main()
{
    short i;
    short output[N];
    long *work;
    long nsamp;

    for(i=0;i<NUMCOEF;i++)
```

Set filter coefficients in X memory as variables.

```
        coeff_x[i] = coeff[i];
    if(InitDIir(&work,K) != EDSP_OK){
        printf("InitDIir Problem\n");
    }
    if(DIir(output, input, N, coeff_x, K, work) != EDSP_OK){
        printf("DIir Problem\n");
    }
    if(FreeDIir(&work, K) != EDSP_OK){
        printf("FreeDIir Problem\n");
    }
    for(i=0;i<N;i++){
        printf("output[%d]=%d\n",i,output[i]);
    }
}
```

Filter initialization:
(1) Work area address
(2) Number of filter sections
This is necessary before DIir function execution. The work area in Y memory uses (number of filter sections)*4*2 bytes.

The FreeDIir function frees the work area used for DIir calculations; The FreeDIir function must always be performed after DIir execution. If this function is not executed, memory resources are wasted.

(f) Double precision IIR for single data elements

Description:

- Format:

```
int DIir1  (short output[], const short input[], long no_samples,
                    const long coeff[], long no_sections,
                    long *workspace)
```

- Parameters:

`output`	Pointer to output data $y_{K-1}(n)$
`input`	Input data x_0 (n)
`coeff[]`	Filter coefficient
`no_sections`	Number of secondary filter sections K
`workspace`	Pointer to the work space

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_sections < 1` •a_{0k} `< 3` •`k < K-1` and a_{0k} `> 32` •`k = K-1` and a_{0k} `> 48`

- Explanation of this function:

Performs double precision infinite impulse-response filter processing for single data elements.

- Remarks:

This filter is configured whereby a secondary filter, or biquad, is linked to the K number tandem. Additional scaling is performed with the output of each biquad. The filter coefficient is specified with a signed 32-bit fixed point number.
The output of each biquad is subject to the following equation.
$d_k(n)=[a_{1k}d_k(n-1)+a_{2k}dk(n-2)+2^{29}x(n)] \cdot 2^{-31}$
$y_k(n)=[b_{0k}d_k(n)+b_{1k}d_k(n-1)+b_{2k}d_k(n-2)] \cdot 2^{-a0k} \cdot 2^{2}$
The input x_k (n) for k is the output y_{k-1} (n) of the previous section. The input of the first section (k=0) is read from the 16-bit left shifted value of input. The output of the last section (k=K-1) is written to output.
Set coeff in the following order of coefficients.
a_{00}, a_{10}, a_{20}, b_{00}, b_{10}, b_{20}, a_{01}, a_{11}, a_{21}, b_{01} ... b_{2K-1}
The a_{0k} item is number of bits for right shift to be performed on the output of the biquad for k.
DIir1 differs from Iir1 in that the filter coefficient is specified with a 32-bit value rather than a 16-bit value. The results of multiply-and-accumulate operations are saved as 64 bits. The output of intermediate stages is the lower 32 bits fetched from the a_{0k} bit right shifted results. When an overflow occurs, this is the positive or negative maximum value. At the last stage, the lower 16 bits are fetched from the a_{0k-1} bit right shifted results. When an overflow occurs, this is the positive or negative maximum value.
Before calling on this function, call on InitDIir, and initialize the work space of the filter.
The delayed node d_k (n) is rounded off to 30 bits, and when an overflow occurs, this is the positive or negative maximum value.
When using DIir1, specify the coefficient with a signed 32-bit fixed point number. In this case, when a_{0k} is k < K-1 set it as 31, and when k=K-1 set it as 47.
As the speed of execution of Iir1 is faster than that of DIir1, if double precision calculation is required, use Iir1.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
#define K  5
#define NUMCOEF  (6*K)
#define N   50
#pragma section X
static long coeff_x[NUMCOEF];
#pragma section
static long coeff[60] =
 {31,1254686956, -496866304, 347415747, 694831502, 347415746,
  31,-113001278,-1523568505, 893094203,1786188388, 893094206,
  31,1254686956, -496866304, 347415747, 694831502, 347415746,
  31,-113001278,-1523568505, 893094203,1786188388, 893094206,
  47,1254686956, -496866304, 347415747, 694831502, 347415746};
static short input[N]   = {32000, 32000, 32000, 32000, 32000,
                              32000, 32000, 32000, 32000, 32000,
                              32000, 32000, 32000, 32000, 32000,
                              32000, 32000, 32000, 32000, 32000,
                              32000, 32000, 32000, 32000, 32000,
                              32000, 32000, 32000, 32000, 32000,
                              32000, 32000, 32000, 32000, 32000,
                              32000, 32000, 32000, 32000, 32000,
                              32000, 32000, 32000, 32000, 32000,
                              32000, 32000, 32000, 32000, 32000 };
void main()
{
    short i;
    short output[N];
    long *work;

    for(i=0;i<NUMCOEF;i++)
        coeff_x[i] = coeff[i];
    if(InitDIir(&work, K) != EDSP_OK){
        printf("Init Problem\n");
```

Include header

Six filter coefficients should be set in one section. The leading element in a section is the number of right-shifts, and is not a filter coefficient.

Set the filter coefficients in X memory. Since Y memory is used by the library as the work area to calculate filter coefficients, Y memory should not be used.

The number of right-shifts is 31 except for the last section; the last section is 47.

Filter initialization:
(1) Work area address
(2) Number of filter sections
This is necessary before DIir1 function execution. The work area in Y memory uses (number of filter sections)*4*2 bytes.

Set filter coefficients in X memory as variables.

DIir1 means that the number of data elements that are set to the DIir function is 1. When executing DIir1 multiple times, the InitDIir and FreeDIir functions must be executed before and after the for statement respectively.

```
    }
    for(i=0;i<N;i++){
        if(DIir1(&output[i], input[i], coeff_x, K, work) !=EDSP_OK){
            printf("DIir1 error\n");
        }
        printf("output[%d]:%d \n" ,i,output[i]);
    }
    if(FreeDIir(&work, K) != EDSP_OK){
        printf("Free DIir error\n");
    }
}
```

DIir1 means that the number of data elements that are set to the DIir function is 1. When executing DIir1 multiple times, the InitDIir and FreeDIir functions must be executed before and after the for statement respectively.

(g) Adaptive FIR

Description:

- Format:

```
int Lms  (short output[], const short input[],
                 const short ref_output[], long no_samples,
                 short coeff[], long no_coeffs, int res_shift,
                 short conv_fact, short  *workspace)
```

- Parameters:

`output[]`	Output data y
`input[]`	Input data x
`ref_output[]`	Desired output value d
`no_samples`	Number of input data elements N
`coeff[]`	Adaptive filter coefficient h
`no_coeffs`	Number of coefficients K
`res_shift`	Right shift applied to each output
`conv_fact`	Convergence coefficient 2µ
`workspace`	Pointer to the work space

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_samples < 1` •`no_coeffs ≤ 2` •`res_shift < 0` •`res_shift > 25`

- Explanation of this function:

Using a least mean square (LMS) algorithm, executes real number adaptive FIR filter processing.

- Remarks:

FIR filters are defined using the following equations.

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k)\ x(n-k)\right] \cdot 2^{-res\_shift}$$

The results of multiply-and-accumulate operations are saved as 39 bits. Output y(n) is the lower 16 bits fetched from the res_shift bit right shifted results. When an overflow occurs, this is the positive or negative maximum value.
Update of filter coefficients is performed using the Widrow-Hoff algorithm.
$h_{n+1}(k)=h_n(k)+2\mu e(n)x(n-k)$
Here, e(n) is the margin of error between the desired signal and the actual output.
e(n)=d(n)-y(n)
With the 2µe(n)x(n-k) calculation, multiplication of 16 bits x 16 bits is performed 2 times. The upper 16 bits of both multiplication results are saved, and when an overflow occurs, this is the positive or negative maximum value. If the value of the updated coefficient is H’8000, there is a possibility that overflow may occur with the multiply-and-accumulate operation. Set the value of the coefficient to be in the range of H'8001 to H'7FFF.
For details on coefficient scaling, refer to “(2) Coefficient scaling”. As the coefficient is adapted using an LMS filter, the safest scaling method is to set less than 256 coefficients and to set res_shift to 24.
conv_fact should normally be set to positive. Do not set it to H’8000.
Before calling on this function, call on InitLms, and initialize the filter.

If the same array is specified for output as for input or for ref_output, input or ref_output will be overwritten.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
```

Include header

Set the filter coefficients in X memory. Since Y memory is used by the library as the work area to calculate filter coefficients, Y memory should not be used.

```
#define K   8
#define N   40
#define TWOMU 32767
#define RSHIFT 15
#define MAXSH 25
#pragma section X
static short coeff_x[K];
#pragma section
short data[N] =  {
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400};
short coeff[K] =  {
         0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000};

static short ref[N] = { -107, -143, 998, 1112, -5956,
                             -10781, 239, 13655, 11202, 2180,
                             -687, -2883, -7315, -6527, 196,
                             4278, 3712, 3367, 4101, 2703,
                             591, 695, -1061, -5626, -4200,
                             3585, 9285, 11796, 13416, 12994,
                             10231, 5803, -449, -6782, -11131,
                             -10376, -2968, 2588, -1241, -6133};
void main()
{
    short *work, i, errc;
    short output[N];
    short twomu;
    int nsamp, ncoeff, rshift;

    /* copy coeffs into X RAM */
    for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
```

Set filter coefficients in X memory as variables.

```
    }
    nsamp = 10;
    ncoeff = K;
    rshift = RSHIFT;
    twomu = TWOMU;
    for (i = 0; i < N; output[i++] = 0) ;
    ncoeff = K;/* Sets the number of coefficients */
    nsamp = N;/* Sets the number of samples */


for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
    if (InitLms(&work, K) != EDSP_OK){
```

Filter initialization:
(1) Work area address
(2) Number of coefficients
This is necessary before LMS function execution. The work area in Y memory uses (number of coefficients)*2+8 bytes.

```
        printf("Init Problem\n");
    }
    if(Lms(output, data, ref, nsamp, coeff_x, ncoeff, RSHIFT,TWOMU, work) !=
   EDSP_OK){
        printf("Lms Problem\n");
    }
    if (FreeLms(&work, K) != EDSP_OK){
```

The FreeLms function frees the work area used for Lms calculations; the FreeLms function must always be executed after Lms execution. If this function is not executed, memory resources are wasted.

```
        printf( "Free Problem\n");
    }
    for (i = 0; i < N; i++){
        printf("#%2d output:%6d \n",i,output[i]);
    }
}
```

(h) Adaptive FIR for single data elements

Description:

- Format:

```
int Lms1  (short *output, short input, short ref_output,
                   short coeff[], long no_coeffs, int res_shift,
                   short conv_fact, short  *workspace)
```

- Parameters:

output	Pointer to output data y(n)
input	Input data x (n)
ref_output	Desired output value d(n)
coeff[]	Adaptive filter coefficient h
no_coeffs	Number of coefficients K
res_shift	Right shift applied to each output.
conv_fact	Convergence coefficient 2μ
workspace	Pointer to the work space

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	In any of the following cases •no_coeffs ≤ 2 •res_shift < 0 •res_shift > 25

- Explanation of this function:

Using a least mean square (LMS) algorithm, executes real number adaptive FIR filter processing for single data elements.

- Remarks:

FIR filters are defined using the following equation.

$$y(n) = \left[\sum_{k=0}^{K-1} h_n(k)x(n-k)\right] \cdot 2^{-res\_shift}$$

The results of multiply-and-accumulate operations are saved as 39 bits. Output y(n) is the lower 16 bits fetched from the res_shift bit right shifted results. When an overflow occurs, this is the positive or negative maximum value.
Update of filter coefficients is performed using the Widrow-Hoff algorithm.
$h_{n+1}(k)=h_n(k)+2\mu e(n)x(n-k)$
Here, e(n) is the margin of error between the desired signal and the actual output.
e(n)=d(n)-y(n)
With the 2μe(n)x(n-k) calculation, multiplication of 16 bits x 16 bits is performed 2 times. The upper 16 bits of both multiplication results are saved, and when an overflow occurs, this is the positive or negative maximum value. If the value of the updated coefficient is H’8000, there is a possibility that overflow may occur with the multiply-and-accumulate operation. Set the value of the coefficient to be in the range of H'8001 to H'7FFF.
For details on coefficient scaling, refer to “(2) Coefficient scaling”. As the coefficient is adapted using an LMS filter, the safest scaling method is to set less than 256 coefficients and to set res_shift to 24.
conv_fact should normally be set to positive. Do not set it to H’8000.
Before calling on this function, call on InitLms, and initialize the filter.
This function is not reentrant.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#include <filt_ws.h>
```

Include header

```
#define K   8
#define N   40
#define TWOMU 32767
#define RSHIFT 15
#define MAXSH 25
```

Variables placed in X or Y memory are defined by a pragma section within the section.

```
#pragma section X
static short coeff_x[K];
#pragma section
short data[N] =  {
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400,
         0x0000, 0x07ff, 0x0c00, 0x0800, 0x0200, 0xf800, 0xf300, 0x0400};
short coeff[K] =  {
         0x0c60, 0x0c40, 0x0c20, 0x0c00, 0xf600, 0xf400, 0xf200, 0xf000};
static short ref[N] = { -107, -143, 998, 1112, -5956,
                             -10781, 239, 13655, 11202, 2180,
                             -687, -2883, -7315, -6527, 196,
                             4278, 3712, 3367, 4101, 2703,
                             591, 695, -1061, -5626, -4200,
                             3585, 9285, 11796, 13416, 12994,
                             10231, 5803, -449, -6782, -11131,
                             -10376, -2968, 2588, -1241, -6133};
void main()
{
    short *work, i, errc;
    short output[N];
    short twomu;
    int nsamp, ncoeff, rshift;
    /* copy coeffs into X RAM */
    for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
```

Set filter coefficients in X memory as variables.

```
    nsamp = 10;
    ncoeff = K;
    rshift = RSHIFT;
    twomu = TWOMU;
    for (i = 0; i < N; output[i++] = 0) ;



    ncoeff = K;/* Sets the number of coefficients */
    nsamp = N;/* Sets the number of samples */

    for (i = 0; i < K; i++){
        coeff_x[i] = coeff[i];
    }
    if (InitLms(&work, K) != EDSP_OK){
        printf("Init Problem\n");
    }
```

Filter initialization:
(1) Work area address
(2) Number of coefficients
This is necessary before LMS1 function execution. The work area in Y memory uses (number of coefficients)*2+8 bytes.

```
    for(i=0;i<nsamp;i++){
        if(Lms1(&output[i], data[i], ref[i], coeff_x, ncoeff, RSHIFT, TWOMU,
                                                                 work) !=
EDSP_OK){
            printf("Lms1 Problem\n");
        }
    }
    if (FreeLms(&work, K) != EDSP_OK){
        printf( "Free Problem\n");
    }
    for (i = 0; i < N; i++){
        printf("#%2d output:%6d
\n",i,output[i]);
    }
}
```

The FreeLms function frees the work area used for Lms calculations; The FreeLms function must always be performed after Lms execution. If this function is not executed, memory resources are wasted.

(i) FIR work space allocation

Description:

- Format:

```
int InitFir  (short **workspace, long no_coeffs)
```

- Parameters:

`workspace`	Pointer to the work space
`no_coeffs`	Number of coefficients K

- Returned value:

`EDSP_OK`	Successful
`EDSP_NO_HEAP`	The memory space that can be used by the work space is insufficient
`EDSP_BAD_ARG`	no_coeffs ≤ 2

- Explanation of this function:

Allocates the work space to be used by Fir and Fir1.

- Remarks:

Initializes all previously input data to 0.
Only Fir, Fir1, Lms or Lms 1 can operate the work space allocated with InitFir. Do not access the work space directly from the user program.
This function is not reentrant.

(j) IIR work space allocation

Description:

```
int InitIir  (short **workspace, long no_sections)
```

- Parameters:

`workspace`	Pointer to the work space
`no_sections`	Number of secondary filter sections K

- Returned value:

`EDSP_OK`	Successful
`EDSP_NO_HEAP`	The memory space that can be used by the work space is insufficient
`EDSP_BAD_ARG`	no_sections < 1

- Explanation of this function:

Allocates the work space to be used by Iir and Iir1.

- Remarks:

Initializes all previously input data to 0.
Only Iir and Iir1 can operate the work space allocated with InitIir. Do not access the work space directly from the user program.
This function is not reentrant.

(k) Double precision IIR work space allocation

Description:

- Format:
  `int InitDIir  (long **workspace, long no_sections)`

- Parameters:

`workspace`	Pointer to the work space
`no_sections`	Number of secondary filter sections K

- Returned value:

`EDSP_OK`	Successful
`EDSP_NO_HEAP`	The memory space that can be used by the work space is insufficient
`EDSP_BAD_ARG`	no_sections < 1

- Explanation of this function:

Allocates the work space to be used by DIir and DIir1.

- Remarks:

Initializes all previously input data to 0.
Only DIir and DIir1 can operate the work space allocated with InitDIir.
This function is not reentrant.

(l) Adaptive FIR work space allocation

Description:

- Format:
  `int InitLms  (short **workspace, long no_coeffs)`

- Parameters:

`workspace`	Pointer to the work space
`no_coeffs`	Number of coefficients K

- Returned value:

`EDSP_OK`	Successful
`EDSP_NO_HEAP`	The memory space that can be used by the work space is insufficient
`EDSP_BAD_ARG`	`no_coeffs ≤ 2`

- Explanation of this function:

Allocates the work space to be used by Lms and Lms1.

- Remarks:

Initializes all previously input data to 0.
Only Fir, Fir1, Lms or Lms 1 can operate the work space allocated with InitLms. Do not access the work space directly from the user program.
This function is not reentrant.

(m) FIR work space release

Description:

- Format:
  `int FreeFir  (short **workspace, long no_coeffs)`

- Parameters:

`workspace`	Pointer to the work space
`no_coeffs`	Number of coefficients K

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	`no_coeffs ≤ 2`

- Explanation of this function:

Releases the work space allocated by InitFir

- Remarks:

This function is not reentrant.

(n) IIR work space release

Description:

- Format:
  `int FreeIir  (short **workspace, long no_sections)`

- Parameters:

`workspace`	Pointer to the work space
`no_sections`	Number of secondary filter sections K

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	`no_sections < 1`

- Explanation of this function:

Releases the work space allocated by InitIir

- Remarks:

This function is not reentrant.

(o) Double precision IIR work space release

Description:

- Format:

```
int FreeDIir  (long **workspace, long no_sections)
```

- Parameters:

`workspace`	Pointer to the work space
`no_sections`	Number of secondary filter sections K

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	`no_section ≤ 2`

- Explanation of this function:

Releases the work space memory allocated by InitDIir.

- Remarks:

This function is not reentrant.

(p) Adaptive FIR work space release

Description:

- Format:

```
int FreeLms  (short **workspace, long no_coeffs)
```

- Parameters:

`workspace`	Pointer to the work space
`no_coeffs`	Number of coefficients K

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	`no_coeffs < 1`

- Explanation of this function:

Releases the work space memory allocated by InitLms

- Remarks:

This function is not reentrant.

3.13.7 Convolution and Correlation

(1) List of functions

Table 3.32 List of DSP Library Functions (Convolution)

No.	Type	Function Name	Description
1	Complete convolution	ConvComplete	Calculates complete convolution for two arrays
2	Periodic convolution	ConvCyclic	Calculates periodic convolution for two arrays
3	Partial convolution	ConvPartial	Calculates partial convolution for two arrays
4	Correlation	Correlate	Calculates correlation for two arrays
5	Periodic correlation	CorrCyclic	Calculates periodic correlation for two arrays

When using these functions, allocate one of the two input arrays to X memory, and the other to Y memory. The output array can be allocated to either memory.

(2) Explanation of each function

(a) Complete convolution

Description:

- Format:

```
int ConvComplete  (short output[], const short ip_x[], const short ip_y[], long
                                  x_size,
                                  long y_size, int res_shift)
```

- Parameters:

`output[]`	Output z
`ip_x[]`	Input x
`ip_y[]`	Input y
`x_size`	Size X of ip_x
`y_size`	Size Y of ip_y
`res_shift`	Right shift applied to each output.

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`x_size < 1` •`y_size < 1` •`res_shift < 0` •`res_shift > 25`

- Explanation of this function:

Complete convolves the two input arrays x and y, and writes the results to the output array z.

- Remarks:

$$z(m) = \left[\sum_{i=0}^{X-1} x(i)\, y(m-i)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < X+Y-1$$

Data external to the input array is read as 0.

ip_x is allocated to X memory, ip_y is allocated to Y memory, and output is allocated to arbitrary memory.

In addition, it is necessary to ensure that the array output size is more than (xsize+ysize-1).

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define NX   8
#define NY   8
#define NOUT  NX+NY-1
#pragma section X
static short datx[NX];
#pragma section Y
static short daty[NY];
#pragma section
short w1[5] = {-1, -32768, 32767, 2, -3, };
short x1[5] = {1, 32767, -32767, -32767, -2, };
void main()
{
    short i;
    short output[NOUT];
    int xsize, ysize, rshift;
    /* copy data into X and Y RAM */
    for(i=0;i<NX;i++){
        datx[i] = w1[i%5];
    }
    for(i=0;i<NY;i++){
        daty[i] = x1[i%5];
    }
    xsize = NX;
    ysize = NY;
    rshift = 15;
    if(ConvComplete(output, datx, daty, xsize, ysize, rshift) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    for(i=0;i<NX;i++){
        printf("#%3d  dat_x:%6d  dat_y:%6d \n",i,datx[i],daty[i]);
    }
    for(i=0;i<NOUT;i++){
        printf("#%3d  output:%d \n",i,output[i]);
    }
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data for use in convolution calculations.

(b) Periodic convolution

Description:

- Format:

```
int ConvCyclic  (short output[], const short ip_x[],
                                const short ip_y[], long size,
                                int res_shift)
```

- Parameters:

`output[]`	Output z
`ip_x[]`	Input x
`ip_y[]`	Input y
`size`	Size N of the array
`res_shift`	Right shift applied to each output.

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 1` •`res_shift < 0` •`res_shift > 25`

- Explanation of this function:

Periodically convolves the two input arrays x and y, and writes the results to the output array z.

- Remarks:

$$z(m) = \left[\sum_{i=0}^{N-1} x(i)\, y\left(\left|m - i + N\right|_N\right)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < N$$

Here, $|i|_N$ means the remainder (i % N).
ip_x is allocated to X memory, ip_y is allocated to Y memory, and output is allocated to arbitrary memory.
In addition, it is necessary to ensure that the array output size is more than ‘size’.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
short x2[5] = {1, 32767, -32767, -32767, -2, };
short w2[5] = {-1, -32768, 32767, 2, -3, };
#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    short i;
    short output[N];
    int size, rshift;
    /* copy data into X and Y RAM */
    for(i=0;i<N;i++){
        datx[i] = w2[i];
        daty[i] = x2[i];
    }
    size = N ;
    rshift = 15;
    if(ConvCyclic(output, datx, daty, size, rshift) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }

    for(i=0;i<N;i++){
        printf("#%2d  ip_x:%6d  ip_y:%6d  output:%6d \n",i,datx[i],daty[i],
output[i]);
    }
}
```

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data for use in convolution calculations.

(c) Partial convolution

Description:

- Format:

```
int ConvPartial  (short output[], const short ip_x[],
                                     const short ip_y[], long x_size,
                                     long y_size,  int res_shift)
```

- Parameters:

`output[]`	Output z
`ip_x[]`	Input x
`ip_y[]`	Input y
`x_size`	Size x of ip_x
`y_size`	Size y of ip_y
`res_shift`	Right shift applied to each output.

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`x_size < 1` •`y_size < 1` •`res_shift < 0` •`res_shift > 25`

- Explanation of this function:

This function convolves the two input arrays x and y, and writes the results to the output array z.

- Remarks:

Output fetched from data external to the input array is not included.

$$z(m) = \left[\sum_{i=0}^{A-1} a(i)\ b(m + A - 1 - i)\right] \cdot 2^{-res\_shift} \quad 0 \le m \le |A\text{-}B|$$

However, the number of arrays is a < b, and A is a size and B is b size.
Data external to the input array is read as 0.
ip_x is allocated to X memory, ip_y is allocated to Y memory, and output is allocated to arbitrary memory.
In addition, it is necessary to ensure that the array output size is more than (|xsize-ysize|+1).

Example of use:

```
#include <stdio.h>
                          Include header
#include <ensigdsp.h>

#define NX   5

#define NY   5

short x3[5] = {1, 32767, -32767, -32767, -2, };

short w3[5] = {-1, -32768, 32767, 2, -3, };
```

```
#pragma section X
static short datx[NX];
#pragma section Y
static short daty[NY];
#pragma section
void main()
{
    short i;
    short output[NY+NX];
    int ysize, xsize, rshift;
    /* copy data into X and Y RAM */
    for(i=0;i<NX;i++){
        datx[i] = w3[i];
    }
    for(i=0;i<NY;i++){
        daty[i] = x3[i];
    }
    xsize = NX;
    ysize = NY;
    rshift = 15;
    if(ConvPartial(output, datx, daty, xsize, ysize, rshift) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    for(i=0;i<NX;i++){
        printf("ip_x=%d  \n",datx[i]);
    }
    for(i=0;i<NY;i++){
        printf("ip_y=%d  \n",daty[i]);
    }
    for(i=0;i<(NY+NX);i++){
        printf("output=%d \n",output[i]);
    }
}
```

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data for use in convolution calculations.

(d) Correlation

Description:

- Format:

```
int Correlate  (short output[], const short ip_x[],
                           const short ip_y[], long x_size, long y_size,
                           long no_corr, int x_is_larger,
                           int res_shift)
```

- Parameters:

output[]	Output z
ip_x[]	Input x
ip_y[]	Input y
x_size	Size x of ip_x
y_size	Size y of ip_y
no_corr	Number of correlations M for calculation
x_is_larger	Array specification when x=y
res_shift	Right shift applied to each output.

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	In any of the following cases •x_size < 1 •y_size < 1 •no_corr < 1 •res_shift < 0 •res_shift > 25 •x_is_larger ≠ 0 or 1

- Explanation of this function:

Finds the correlation of the two input arrays x and y, and writes the results to the output array z.

- Remarks:

In the following equation, the number of arrays is a < b, and A is a size. If x_is_larger=0 make x to be a, and if x_is_larger=1 make x to be b.
Operation is not guaranteed when the b array is smaller than the a array.
Set the sizes of the input arrays x and y, as well as x_is_larger, so that no conflict exists.

$$z(m) = \left[\sum_{i=0}^{A-1} a(i)\ b(i + m)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < M$$

There is no obstacle to having A < X + M. In this case, use 0 for data external to the array.
res_shift=0 corresponds to normal integer calculation, and res_shift=15 corresponds to decimal calculation.
ip_x is allocated to X memory, ip_y is allocated to Y memory, and output is allocated to arbitrary memory.
In addition, it is necessary to ensure that the array output size is more than no_corr.

Example of use:

```
#include <stdio.h>
                        } Include header
#include <ensigdsp.h>

#define NY   5

#define NX   5
```

```
#define M    4
#define MAXM  NX+NY
short x4[5] = {1, 32767, -32767, -32767, -2, };
short w4[5] = {-1, -32768, 32767, 2, -3, };
#pragma section X
static short datx[NX];
#pragma section Y
static short daty[NY];
```

Variables placed in X or Y memory are defined by a pragma section within the section.

```
#pragma section
void main()
{
    short i;
    int ysize, xsize, ncorr, rshift;
    short output[MAXM];
    int x_is_larger;
    /* copy data into X and Y RAM */
    for(i=0;i<NX;i++){
         datx[i] = w4[i%5];
    }
    for(i=0;i<NY;i++){
        daty[i] = x4[i%5];
    }
```

Sets data for use in calculations.

```
    /* test working of stack */
    ysize = NY;
    xsize = NX;
    ncorr = M;
    rshift = 15;
    x_is_larger=0;
    for (i = 0; i < MAXM; output[i++] = 0);
    if (Correlate(output, datx, daty, xsize, ysize, ncorr,x_is_larger,rshift)
                                                                              !=
EDSP_OK){
       printf("EDSP_OK not returned\n");
    }
    for(i=0;i<MAXM;i++){
        printf("[%d]:output=%d\n",i,output[i]);
    }
}
```

(e) Periodic correlation

Description:

- Format:

```
int CorrCyclic  (short output[], const short ip_x[],
                              const short ip_y[], long size, int reverse,
                              int res_shift)
```

- Parameters:

`output[]`	Output z
`ip_x[]`	Input x
`ip_y[]`	Input y
`size`	Size N of the array
`reverse`	Reverse flag
`res_shift`	Right shift applied to each output.

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`size < 1` •`res_shift < 0` •`res_shift > 25` •`reverse ≠ 0 or 1`

- Explanation of this function:

Finds the correlation of the two input arrays x and y periodically, and writes the results to the output array z.

- Remarks:

$$z(m) = \left[\sum_{i=0}^{N-1} x(i)\, y\left(|i + m|_N\right)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < N$$

Here, $|i|_N$ means the remainder (i % N). If reverse=1, the output data is reversed, and the actual calculation is as follows.

$$z(m) = \left[\sum_{i=0}^{N-1} y(i)\, x\left(|i + m|_N\right)\right] \cdot 2^{-res\_shift} \qquad 0 \le m < N$$

ip_x is allocated to X memory, ip_y is allocated to Y memory, and output is allocated to arbitrary memory.
In addition, it is necessary to ensure that the array output size is more than ‘size’.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
short x5[5] = {1, 32767, -32767, -32767, -2, };
short w5[5] = {-1, -32768, 32767, 2, -3, };

#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    short i;
    short output[N];
    int size, rshift;
    int reverse;
    int result;
    /* TEST CYCLIC CORRELATION OF X WITH Y */
    reverse=0;
    /* copy data into X and Y RAM */
    for(i=0;i<N;i++){
        datx[i] = w5[i];
        daty[i] = x5[i];
    }
    /* test working of stack */
    size = N;
    rshift = 15;
    if (CorrCyclic(output, datx, daty, size, reverse, rshift) != EDSP_OK){
        printf("EDSP_OK not returned - this one\n");
    }
    for(i=0;i<N;i++){
        printf("output[%d]=%d\n",i,output[i]);
    }
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data for use in calculations.

3.13.8 Other

(1) List of functions

Table 3.33 DSP Library Function List (Miscellaneous)

No.	Type	Function Name	Description
1	H’8000 → H’8001 replacement	Limit	Replaces H'8000 data with H'8001
2	X memory → Y memory copy	CopyXtoY	Copies an array from X memory to Y memory.
3	Y memory → X memory copy	CopyYtoX	Copies an array from Y memory to X memory.
4	Copy to X memory	CopyToX	Copies an array from a specified location to X memory.
5	Copy to Y memory	CopyToY	Copies an array from a specified location to Y memory.
6	Copy from X memory	CopyFromX	Copies an array from X memory to a specified location.
7	Copy from Y memory	CopyFromY	Copies an array from Y memory to a specified location.
8	Gaussian white noise	GenGWnoise	Generates Gaussian white noise.
9	Matrix multiplication	MatrixMult	Multiplies two matrices.
10	Multiplication	VectorMult	Multiplies two data elements.
11	RMS value	MsPower	Determines RMS power.
12	Mean	Mean	Determines mean.
13	Mean and variance	Variance	Determines mean and variance.
14	Maximum value	MaxI	Determines maximum value of integer array.
15	Minimum value	MinI	Determines minimum value of integer array.
16	Maximum absolute value	PeakI	Determines maximum absolute value of integer array.

(2) Explanation of each function

(a) H’8000 → H’8001 replacement

Description:

- Format:
  `int Limit (short data[], long no_elements, int data_is_x)`

- Parameters:

`data[]`	Data array
`no_elements`	Number of data elements
`data_is_x`	Data location specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_elements < 1` •`data_is_x ≠ 0 or 1`

- Explanation of this function:

Replaces input data with a value of H'8000 with H'8001. In this way, when fixed point multiplication is performed with the DSP instruction, overflow will not occur.

- Remarks:

Even when the process is performed there is a possibility that overflow may occur with addition in the multiply-and-accumulate operation.
When data_is_x=1 allocate data to X memory, and when data_is_x=0 allocate data to Y memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    short i;
    int size;
    int src_x;
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

```
      /* copy data into X and Y RAM */
      for(i=0;i<N;i++) {
          datx[i] = dat[i%4];
          daty[i] = dat[i%4];
          printf("BEFORE NO %d datx daty :%d:%d \n",i,datx[i], daty[i]);
      }
      size = N;
      src_x = 1;

      if (Limit(datx, size, src_x) != EDSP_OK){
          printf( "EDSP_OK not returned\n");
      }
      src_x = 0;
      if (Limit(daty, size, src_x) != EDSP_OK){
          printf( "EDSP_OK not returned\n");
      }
      for(i=0;i<N;i++) {
          printf("After NO %d datx daty :%d:%d\n",i,datx[i], daty[i]);
      }
  }
```

Sets data.

If using X memory

If using Y memory

(b) X memory → Y memory copy

Description:

- Format:
  `int CopyXtoY  (short op_y[], const short ip_x[], long n)`

- Parameters:

`op_y[]`	Output array
`ip_x[]`	Input array
`n`	Number of data elements

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	`n < 1`

- Explanation of this function:

The array is copied from ip_x to op_y.

- Remarks:

Allocate ip_x to X memory, and allocate op_y to Y memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    int i;

    for(i=0;i<N;i++){
        daty[i]=0;
        datx[i]=dat[i%4];
    }
    if(CopyXtoY(daty, datx, N) != EDSP_OK){
        printf("CopyXtoY Problem\n");
    }
    printf("no_elements:%d \n",N);
    for(i=0;i<N;i++){
        printf("#%2d  op_x:%6d  ip_y:%6d \n",i,datx[i],daty[i]);
    }
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data.

(c) Y memory → X memory copy

Description:

- Format:
  `int CopyYtoX  (short op_x[], const short ip_y[], long n)`
- Parameters:

`op_x[]`	Output array
`ip_y[]`	Input array
`n`	Number of data elements

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	n < 1

- Explanation of this function:

The array is copied from ip_y to op_x.

- Remarks:

Allocate ip_y to Y memory, and allocate op_x to X memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  5
static short dat[N] = { -32768, 32767, -32768, 0,3};
#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
void main()
{
    int i;

    for(i=0;i<N;i++){
        daty[i]=dat[i];
    }
    if(CopyYtoX(datx, daty, N)!= EDSP_OK){
      printf("CopyYtoX error!\n");
    }
    printf("no_elements %d \n",N);
    for(i=0;i<N;i++){
        printf("#%2d  po_x:%6d  ip_y:%6d \n",i,datx[i],daty[i]);
    }
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data.

(d) Copy to X memory

Description:

- Format:
  `int CopyToX  (short op_x[], const short input[], long n)`

- Parameters:

`op_x[]`	Output array
`input[]`	Input array
`n`	Number of data elements

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	`n < 1`

- Explanation of this function:

The array input is copied to op_x.

- Remarks:

Allocate op_x to X memory, and allocate input to arbitrary memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
#pragma section X
static short datx[N];
#pragma section
void main()
{
    int   i;
    short data[N];

    for(i=0;i<N;i++){
       data[i]=dat[i];
    }
    if(CopyToX(datx, data, N) !=EDSP_OK){
        printf("CopyToX Problem\n");
    }
    printf("no_elements %d\n",N);
    for(i=0;i<N;i++){
        printf("#%2d  op_x:%6d  input:%6d \n",i,datx[i],data[i]);
```

Include header

Variables placed in X memory are defined by a pragma section within the section.

Sets data.

```
    }
}
```

(e) Copy to Y memory

Description:

- Format:
  `int CopyToY (short op_y[], const short input[], long n)`

- Parameters:

`op_y[]`	Output array
`input[]`	Input array
`n`	Number of data elements

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	`n < 1`

- Explanation of this function:

The array input is copied to op_y.

- Remarks:

Allocate op_y to Y memory, and allocate input to arbitrary memory.

Example of use:

```
#include <stdio.h>     }
                       }  Include header
#include <ensigdsp.h>  }
#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
#pragma section Y
static short daty[N]; <--------  Variables placed in Y memory are defined by a
#pragma section                  pragma section within the section.
void main()
{
    int i;
    short data[N] ;

    for(i = 0; i < N; i++){
        data[i] = dat[i%4] ; <----  Sets data.
    }
    if(CopyToY(daty, data, N) != EDSP_OK){
        printf("CopyToY Problem\n");
    }
    printf("no_elements %ld \n",N);
    for(i = 0; i < N; i++){
        printf("#%2d  op_y:%6d  input:%6d \n",i,daty[i],data[i]);
    }
}
```

(f) Copy from X memory

Description:

- Format:
  `int CopyFromX  (short output[], const short ip_x[], long n)`

- Parameters:

output[]	Output array
ip_x[]	Input array
n	Number of data elements

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	n < 1

- Explanation of this function:

The array ip_x is copied to output.

- Remarks:

Allocate ip_x to X memory, and allocate output to arbitrary memory.

Example of use:

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

Include header

```
#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
static short out_dat[N] ;

#pragma section X
static short datx[N];
#pragma section
void main()
```

Variables placed in X memory are defined by a pragma section within the section.

```
{
    int i;

    for(i=0;i<N;i++){
        datx[i]=dat[i];
    }
    if(CopyFromX(out_dat,datx, N) != EDSP_OK){
        printf("CopyFromX Problem\n");
    }
    for(i=0;i<N;i++){
        printf("#%3d output:%6d  ip_x:%6d \n",i,out_dat[i],datx[i]);
    }
    printf("no_elements:%ld\n",N);
}
```

Sets data.

(g) Copy from Y memory

Description:

- Format:
  `int CopyFromY  (short output[], const short ip_y[], long n)`

- Parameters:

`output[]`	Output array
`ip_y[]`	Input array
`n`	Number of data elements

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	`n < 1`

- Explanation of this function:

The array ip_y is copied to output.

- Remarks:

Allocate ip_y to Y memory, and allocate output to arbitrary memory.

Example of use:

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

Include header

```
#define N  4
static short dat[N] = { -32768, 32767, -32768, 0};
static short out_dat[N] ;


#pragma section Y
static short daty[N];
```

Variables placed in Y memory are defined by a pragma section within the section.

```
#pragma section
void main()
{
    int i;

    for(i=0;i<N;i++){
        daty[i]=dat[i];
```

Sets data.

```
    }
    if(CopyFromY(out_dat,daty, N)!= EDSP_OK){
         printf("CopyFormY Problem\n");
    }
    printf("no_elements:%d \n",N);
    for(i=0;i<N;i++){
        printf("#%2d  output:%6d  ip_y:%6d \n",i,out_dat[i],daty[i]);
    }
}
```

(h) Gaussian white noise

Description:

- Format:
  `int GenGWnoise  (short output[], long no_samples, float variance)`

- Parameters:

`output[]`	Outputs white noise data
`no_samples`	Number of output data elements
`Variance`	Variance of noise distribution σ^2

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases

- `no_samples < 1`
- `variance ≤ 0.0`

- Explanation of this function:

With a mean of 0, Gaussian white noise is generated with the variance specified by the user.

- Remarks:

One set of two output data elements are generated. In order to generate 1 set of output data, use a rand function, and until a set of less than 1 is found by the sum of the square of x, 1 set of random numbers, γ_1 and γ_2, between –1 and 1 is generated. Then 1 set of output data, o_1 and o_2, is calculated using the following equations.

$$o_1 = \sigma\gamma_1\sqrt{-2\ln(x)/x}$$

$$o_2 = \sigma\gamma_2\sqrt{-2\ln(x)/x}$$

If the number of data elements is set to an odd number, the second data element of the last set is nullified.
As the rand function of the standard library called on by this function is not reentrant, the order of the random numbers γ_1 and γ_2 generated will not necessarily always be the same. However, there will be no impact on the characteristics of the white noise o_1 and o_2 generated.
This function uses a floating point operation. As the processing speed of floating point operations is slow, it is recommended that this function is used for evaluation.

Example of use:

```
#include <stdio.h>      }  Include header
#include <ensigdsp.h>   }
#define MAXG  4.5 /* approx. saturating level for N(0,1) random variable */
#define N_SAMP  10  /* number of samples generated in a frame        */
void main()
{
    short out[N_SAMP];
    float var;
    int   i;
    var = 32768 / MAXG * 32768 / MAXG;
    if(GenGWnoise(out, N_SAMP, var) !=EDSP_OK){
        printf("GenGWnoise Problem\n");
    }
    for(i=0;i<N_SAMP;i++){
        printf("#%2d  out:%6d  \n",i,out[i]);
    }
}
```

(i) Matrix multiplication

Description:

- Format:

```
int MatrixMult  (void *op_matrix, const void *ip_x,
                           const void *ip_y, long m, long n, long p,
                           int x_first, int res_shift)
```

- Parameters:

op_matrix	Pointer to the first data element of output
ip_x	Pointer to the first data element of input x
ip_y	Pointer to the first data element of input y
m	Number of rows in matrix 1
n	Number of columns in matrix 1, number of rows in matrix 2
p	Number of rows in matrix 2
x_first	Order specification for matrix multiplication
res_shift	Right shift applied to each output.

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	In any of the following cases •m, n, or p < 1 •res_shift < 0 •res_shift > 25 •x_first ≠ 0 or 1

- Explanation of this function:

Performs multiplication of the two matrices x and y, and allocates the result to op_matrix.

- Remarks:

When x_first=1, calculates x · y In this case, ip_x is m x n, ip_y is n x p, and op_matrix is m x p.
When x_first=0, calculates y · x. In this case, ip_y is m x n, ip_x is n x p, and op_maxtrix is m x p.
The results of multiply-and-accumulate operations are saved as 39 bits. Output y(n) is the lower 16 bits fetched from the res_shift bit right shifted results. When an overflow occurs, this is the positive or negative maximum value.
Each matrix is allocated to a normal C format (row major order).

$$\begin{matrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \end{matrix}$$

In order to be able to specify an arbitrary array size, specify void* for the array parameters. Make these parameters point to short variables.
Provide input arrays ip_x and ip_y, and output array op_matrix separately.
Allocate ip_x to X memory, allocate ip_y to Y memory, and allocate op_matrix to arbitrary memory.

Example of use:

```
#include <stdio.h>
                        Include header
#include <ensigdsp.h>

#define N   4

#define NN  N*N
```

```
short m1[16] = {  1, 32767, -32767, 32767,
                  1, 32767, -32767, 32767,
                  1, 32767, -32767, 32767,
                  1, 32767, -32767, 32767, };
short m2[16] = {  -1, 32767, -32767, -32767,
                  -1, 32767, -32767, -32767,
                  -1, 32767, -32767, -32767,
                  -1, 32767, -32767, -32767, };
#pragma section X
static short datx[NN];
#pragma section Y
static short daty[NN];
```

Variables placed in X or Y memory are defined by a pragma section within the section.

```
#pragma section
void main()
{
    short i, j;
    short output[NN];
    int m, n, p, rshift, x_first;
    long sum;
    for (i = 0; i < NN; output[i++] = 0) ;
    /* copy data into X and Y RAM */
    for(i=0;i<NN;i++) {
        datx[i] = m1[i%16];
        daty[i] = m2[i%16];
    }
```

Sets data.

```
    m = n = p = N;
    rshift = 15;
    x_first = 1;
    if (MatrixMult(output, datx, daty, m, n, p, x_first, rshift) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    for(i=0;i<NN;i++) {
        printf("output[%d]=%d\n",i,output[i]);
    }
}
```

(j) Multiplication

Description:

- Format:

```
int VectorMult  (short output[], const short ip_x[],
                            const short ip_y[],long no_elements,
                            int res_shift)
```

- Parameters:

output[]	Output
ip_x[]	Input 1
ip_y[]	Input 2
no_elements	Number of data elements
res_shift	Right shift applied to each output.

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	In any of the following cases •no_elements < 1 •res_shift < 0 •res_shift > 16

- Explanation of this function:

Data is fetched one element at a time from ip_x and ip_y and multiplication is performed, with the results being allocated to output.

- Remarks:

Output is the lower 16 bits fetched from the res_shift bit right shifted results.
When an overflow occurs, this is the positive or negative maximum value.
This function performs multiplication of the data. To calculate the inner product, use the MatrixMult function, setting 1 for m (the number of rows of matrix 1) and for p (the number of columns of matrix 2).
ip_x is allocated to X memory, ip_y is allocated to Y memory, and output is allocated to arbitrary memory.

Example of use:

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

Include header

```
#define N  4
#define RSHIFT  15
short y[4] = {1, 32767, -32767, 32767, };
short x[4] = {-1, 32767, -32767, -32767, };


#pragma section X
static short datx[N];
#pragma section Y
static short daty[N];
#pragma section
```

Variables placed in X or Y memory are defined by a pragma section within the section.

```
void main()
{
    short i, n ;
    short output[N];
    int size, rshift;

    /* copy data into X and Y RAM */
    for(i=0;i<N;i++) {
        datx[i] = x[i];
        daty[i] = y[i];
    }
    size = N;
    rshift = RSHIFT;
    for (i = 0; i < N; output[i++] = 0) ;
    if (VectorMult(output, datx, daty, size, rshift) != EDSP_OK) {
        printf("EDSP_OK not returned\n");
    }
    for(i=0;i<N;i++){
        printf("#%2d  output:%6d  ip_x:%6d  ip_y:%6d \n",i,output[i],datx[i],

daty[i]);
    }
}
```

Sets data.

(k) RMS value

Description:

- Format:

```
 int MsPower  (long *output, const short input[],long no_elements, int src_is_x)
```

- Parameters:

`output`	Pointer to output
`input[]`	Input x
`no_elements`	Number of data elements N
`src_is_x`	Data location specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_elements < 1` •`src_is_x ≠ 0 or 1`

- Explanation of this function:

Determines the RMS value of input data.

- Remarks:

$$\mathrm{RMS} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)^2$$

Rounds off the division result to the nearest integer.
The result of the operation is saved as 63 bits.
If no_elements is 2^{32}, overflow may occur.
When src_is_x=1 allocate input to X memory, and when src_is_x=0 allocate data to Y memory.
Allocate output to arbitrary memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};

#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    int    i;
    long   output[1];
    int    src_x;

    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
    src_x = 1;
    if (MsPower(output, datx, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("MsPower:x=%d\n",output[0]);
    src_x = 0;
    if (MsPower(output, daty, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("MsPower:y=%d\n",output[0]);
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data.

When X memory is used, src_x=1.

When Y memory is used, src_x=0.

(l) Mean

Description:

- Format:

```
int Mean  (short *mean, const short input[], long no_elements,
                     int src_is_x)
```

- Parameters:

mean	Pointer to mean value of input
input[]	Input x
no_elements	Number of data elements N
src_is_x	Data location specification

- Returned value:

EDSP_OK	Successful
EDSP_BAD_ARG	In any of the following cases •no_elements < 1 •src_is_x ≠ 0 or 1

- Explanation of this function:

Determines the mean of input data.

- Remarks:

$$\overline{x} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)$$

Rounds off the division result to the nearest integer.
The operation result is saved as 32 bits. If no_elements is greater than 2^{16}-1, overflow may occur.
When src_is_x=1 allocate input to X memory, and when src_is_x=0 allocate data to Y memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N  5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};
#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    short     i,output[1];
    int       size;
    int       src_x;
    int       flag = 1;
    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
    /* test working of stack */
    src_x = 1;
    if (Mean(output, datx, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("Mean:x=%d\n",output[0]);
    src_x = 0;
    if (Mean(output, daty, N, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("Mean:y=%d\n",output[0]);
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

When X memory is used, src_x=1.

When Y memory is used, src_x=0.

(m) Mean and variance and

Description:

- Format:

```
int Variance  (long *variance, short *mean, const short input[],
                        long no_elements, int src_is_x)
```

- Parameters:

`Variance`	Pointer to the variance σ^2 of input
`mean`	Pointer to data mean $\overline{x}$
`input[]`	Input x
`no_elements`	Number of data elements N
`src_is_x`	Data location specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_elements < 1` •`src_is_x ≠ 0` or `1`

- Explanation of this function:

Determines mean and variance of input.

- Remarks:

$$\overline{x} = \frac{1}{N}\sum_{i=0}^{N-1} x(i)$$

$$\sigma^2 = \frac{1}{N}\sum_{i=0}^{N-1} x(i)^2 - \overline{x}^2$$

Rounds off the division result to the nearest integer.
$\overline{x}$ is saved as 32 bits. There is no check for overflow.
If no_elements is greater than 2^{16}-1, overflow may occur.
σ^2 is saved as 63 bits. There is no check for overflow.
When src_is_x=1 allocate input to X memory, and when src_is_x=0 allocate data to Y memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};
#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    long         size,var[1];
    short        mean[1];
    int          i ;
    int          src_x;

    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }

    /* test working of stack */
    size = N;
    src_x = 1;
    if (Variance(var, mean, datx, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("Variance:%d  mean:%d \n ",var[0],mean[0]);
    src_x = 0;
    if (Variance(var, mean, daty, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("Variance:%d  mean:%d \n ",var[0],mean[0]);
}
```

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data.

When X memory is used, src_x=1.

When Y memory is used, src_x=0.

(n) Maximum value

Description:

- Format:

```
int MaxI  (short **max_ptr, short input[], long no_elements,
                  int src_is_x)
```

- Parameters:

`max_ptr`	Pointer to the maximum data
`input[]`	Input
`no_elements`	Number of data elements
`src_is_x`	Data location specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_elements < 1` •`src_is_x ≠ 0 or 1`

- Explanation of this function:

Searches for the maximum value in the array input, and returns its address to max_ptr.

- Remarks:

If several data elements have the same maximum value, the address of the data with the start closest to input is returned. When src_is_x=1 allocate input to X memory, and when src_is_x=0 allocate data to Y memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
static short dat[131] = {-16384, -32767, 32767, 14877, 8005};
#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    short    *outp,**outpp;
    int       size,i;
    int       src_x;
 /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
    /* MAXI */
    size = N;
    outpp = &outp;
    src_x = 1;
    if (MaxI(outpp, datx, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("Max:x = %d\n",**outpp);
    src_x = 0;
    if (MaxI(outpp, daty, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("Max:y = %d\n",**outpp);
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data.

When X memory is used, src_x=1.

When Y memory is used, src_x=0.

(o) Minimum value

Description:

- Format:

```
int MinI  (short **min_ptr, short input[], long no_elements, int src_is_x)
```

- Parameters:

`min_ptr`	Pointer to the minimum data
`input[]`	Input
`no_elements`	Number of data elements
`src_is_x`	Data location specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_elements < 1` •`src_is_x ≠ 0 or 1`

- Explanation of this function:

Searches for the minimum value in the array input, and returns its address to min_ptr.

- Remarks:

If several data elements have the same minimum value, the address of the data with the start closest to input is returned. When src_is_x=1 allocate input to X memory, and when src_is_x=0 allocate data to Y memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 10
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};
#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    short     *outp,**outpp;
    int       size,i;
    int       src_x;
    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
    /* MINI */
    /* test working of stack */
    size = N;
    outpp = &outp;
    src_x = 1;
    if (MinI(outpp, datx, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("Min:x=%d\n",**outpp);
    src_x = 0;
    if (MinI(outpp, daty, size, src_x) != EDSP_OK){
        printf("EDSP_OK not returned\n");
    }
    printf("Min:y=%d\n",**outpp);
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data.

When X memory is used, src_x=1.

When Y memory is used, src_x=0.

(p) Maximum absolute value

Description:

- Format:

```
int PeakI  (short **peak_ptr, short input[], long no_elements, int src_is_x)
```

- Parameters:

`peak_ptr`	Pointer to the maximum absolute value data
`input[]`	Input
`no_elements`	Number of data elements
`src_is_x`	Data location specification

- Returned value:

`EDSP_OK`	Successful
`EDSP_BAD_ARG`	In any of the following cases •`no_elements < 1` •`src_is_x ≠ 0 or 1`

- Explanation of this function:

Searches for the maximum absolute value in the array input, and returns its address to peak_ptr.

- Remarks:

If several data elements have the same maximum absolute value, the address of the data with the start closest to input is returned.
When src_is_x=1 allocate input to X memory, and when src_is_x=0 allocate data to Y memory.

Example of use:

```
#include <stdio.h>
#include <ensigdsp.h>
#define N 5
static short dat[5] = {-16384, -32767, 32767, 14877, 8005};
#pragma section X
static short    datx[N];
#pragma section Y
static short    daty[N];
#pragma section
void main()
{
    short      *outp,**outpp;
    int        size,i;
    int        src_x;
    /* copy data into X and Y RAM */
    for (i = 0; i < N; i++) {
        datx[i] = dat[i];
        daty[i] = dat[i];
    }
    size = N;
    outpp = &outp;
    src_x = 1;
    if (PeakI(outpp, datx, size, src_x) != EDSP_OK)
    {
        printf("EDSP_OK not returned\n");
    }
    printf("Peak:x=%d\n",**outpp);
    src_x = 0;
    if (PeakI(outpp, daty, size, src_x) != EDSP_OK)
    {
        printf("EDSP_OK not returned\n");
    }
    printf("Peak:y=%d\n",**outpp);
}
```

Include header

Variables placed in X or Y memory are defined by a pragma section within the section.

Sets data.

When X memory is used, src_x=1.

When Y memory is used, src_x=0.

3.14 Performance of the DSP Library

(1) Number of execution cycles of the DSP library

The number of execution cycles required by functions in the DSP library are indicated below. Measurements were performed using an emulator (SH-DSP, 60 MHz), with the program section allocated to X-ROM or to Y-ROM.

Table 3.34 List of Execution Cycles for DSP Library Functions (1)

Category	DSP Library Function Name	Number of Execution Cycles (Cycle)	Notes
Fast Fourier transforms	FftComplex	29,330	Size: 256 Scaling: 0xFFFFFFFF
	FftReal	25,490	
	IfftComplex	30,380	
	IfftReal	29,240	
	FftInComplex	26,540	
	FftInreal	25,260	
	IfftInComplex	27,590	
	IfftInReal	27,470	
	LogMagnitude	1,778,290	
	InitFft	3,116,640	
	FreeFft	780	
Filter functions	Fir	23,010	Number of coefficients: 64 Number of data items: 200 Convergence coefficient 2μ = 32767
	Fir1	280	
	Lms	97,710	
	Lms1	790	
	InitFir	1,400	
	InitLms	1,400	
	FreeFir	90	
	FreeLms	90	
	Iir	23,530	Number of data items: 200 Number of filter sections: 5
	Iir1	360	
	DIir	309,010	
	DIir1	1,860	
	InitIir	280	
	InitDIir	280	
	FreeIir	90	
	FreeDIir	270	

Table 3.34 List of Execution Cycles for DSP Library Functions (2)

Category	DSP Library Function Name	Number of Execution Cycles (Cycle)	Notes
Window functions	GenBlackman	789,950	Number of data items: 100
	GenHamming	418,330	
	GenHanning	447,250	
	GenTriangle	744,220	
Convolution functions	ConvComplete	21,890	Number of data items: 100
	ConvCyclic	14,790	
	ConvPartial	370	
	Correlate	11,930	
	CorrCylic	15,790	
Other functions	Limit	480	Number of data items: 100
	CopyXtoY	130	
	CopyYtoX	130	
	CopyToX	1,270	
	CopyToY	1,270	
	CopyFromX	1,320	
	CopyFromY	1,320	
	GenGWnoise	2,878,410	
	MatrixMult	2,337,460	
	VectorMult	1,500	
	MsPower	370	
	Mean	270	
	Variance	820	
	MaxI	540	
	MinI	520	
	PeakI	740	

(2) Comparison of C language and DSP library source code

Here source code is presented in C language and from the DSP library, for some of the FFT-related functions (those performing butterfly calculations).

In the DSP library source code, the DSP-specific instructions such as movx, movy, and padd are used to improve the performance of the DSP library.

C source code

```
void R4add(short *arp, short *brp, short *aip, short *bip, int grpinc, int numgrp)
{
short  tr,ti;
int    grpind;
       for(grpind=0;grpind<numgrp;grpind++) {
                 tr  =  *brp;
                 ti  =  *bip;
                 *brp = sub(*arp,ti);
                 *bip = add(*aip,tr);
                 *arp = add(*arp,ti);
                 *aip = sub(*aip,tr);
                 arp += grpinc;
                 aip += grpinc;
                 brp += grpinc;
                 bip += grpinc;
       }
}
```

DSP library source code

```
_R4add:

       MOV.L  Ix,@-R15
       MOV.L  Iy,@-R15

       MOV.L  @(2*4,R15),Ix
       SHLL   Ix
       MOV    Ix,Iy
       MOV.L  @(3*4,R15),R1

       REPEAT r4alps,r4alpe
       ADD    #-1,R1
       SETRC  R1
                                 movx.w  @ar,X0     movy.w @bi,Y0
       padd   X0,Y0,A0
       psub   X0,Y0,A1           movx.w  @br,X0     movy.w @ai,Y0
       padd   X0,Y0,A0           movx.w  A0,@ar+Ix
       pneg   X0,X0              movx.w  A1,@br+Ix
       padd   X0,Y0,A1           movy.w A0,@bi+Iy
```

```
        movx.w @ar,X0            movy.w @bi,Y0
        .ALIGN 4
 r4alps padd   X0,Y0,A0          movy.w A1,@ai+Iy
        psub   X0,Y0,A1          movx.w  @br,X0     movy.w @ai,Y0
        padd   X0,Y0,A0          movx.w  A0,@ar+Ix
        pneg   X0,X0             movx.w  A1,@br+Ix
        padd   X0,Y0,A1          movy.w A0,@bi+Iy
 r4alpe                          movx.w  @ar,X0     movy.w @bi,Y0
                                 movy.w A1,@ai+Iy
        MOV.L  @R15+,Iy
        RTS
        MOV.L  @R15+,Ix
```

(3) Performance of individual FFT functions

Fourier transform functions are classified as follows.

Table 3.35 Fast Fourier Transform Functions

	Not-in-place function	In-place function
Complex Fourier transform	FftComplex	FftInComplex
Real Fourier transform	FftReal	FftReal

Table 3.36 Inverse Fast Fourier Transform Functions

	Not-in-place function	In-place function
Complex Fourier transform	IfftComplex	IfftInComplex
Real Fourier transform	IfftReal	IfftInReal

Differences between In-Place and Not-In-Place Functions

In-place functions use the array of input data as the array for output data. Hence the input data is overwritten by the output data, and is not saved.

When using not-in-place functions, the input and output data must be prepared separately before calling on a function. The input data and output data are separate, and so the input data is saved even after the function is called on.

There is almost no difference in the performance of in-place and not-in-place functions, and so the type of function to be used should be determined based on the amount of memory available.

Compared with not-in-place functions, in-place functions require half the amount of memory.

- About scaling

In each stage of FFT calculations, calculations are executed in multiply-and-accumulate form, so overflows tend to occur. If an overflow occurs, all values become maxima or minima, so that calculation results cannot be evaluated correctly.

In order to prevent overflow, scaling is performed at each stage of FFT calculations; the scaling is 2 by which values are divided (right-shifted).

Table 3.37 Scaling Values and Features

Scaling Value	Features
FFTNOSCALE	No shifting whatsoever; overflow tends to occur
EFFTMIDSCALE	Shifting at every other stage
EFFTALLSCALE	Shifting at all stages; overflow does not occur readily

Scaling does not have a large effect on performance. Hence when deciding on a scaling, the features of the data, rather than performance, should be considered.

(4) Filter functions

Using Fir and Lms

The relation between the number of coefficients and cycles for the Fir and Lms filters are shown in figure 3.11.

Because the Lms filter uses an adaptive algorithm, speed of calculation is slower than for the Fir filter. In a system with stable data waveforms, Lms should be used to determine filter coefficients, after which it should be replaced by the Fir filter.

The number of right-shifts can be specified for data scaling. Because multiply-and-accumulate operations are used internally in SH-DSP library functions, depending on the input data, overflows may occur. In such cases the number of right-shifts should be modified appropriately, and should be selected referring to output values.

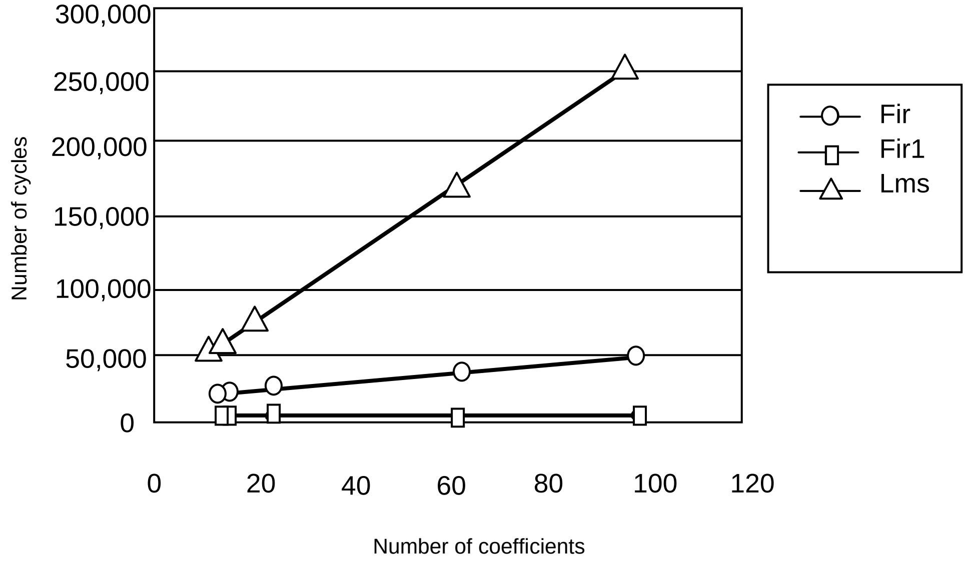


Figure 3.11 Relation between Number of Coefficients and Number of Cycles

- Iir and DIir

When performance is given priority, Iir should be used instead of DIir. Because multiply-and-accumulate operations are used internally in SH-DSP library functions, depending on the input data, overflows may occur. In such cases the number of right-shifts should be modified appropriately, and should be selected referring to output values.

The number of right-shifts can be specified for data scaling. However, the number of right-shifts is specified as part of the array of filter coefficients. For details, refer to section 3.13.6, (5)(c) IIR and (e) Double precision IIR.

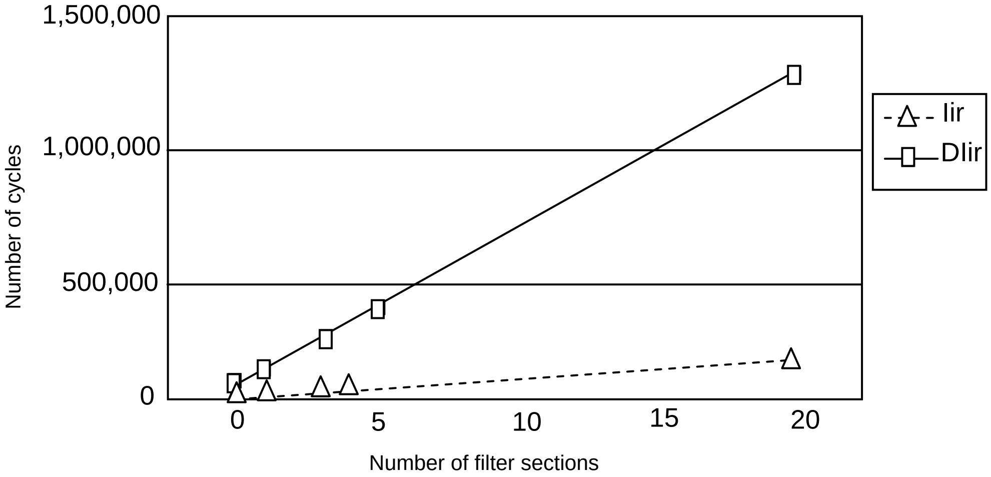


Figure 3.12 Relation between Number of Filter Sections and Number of Cycles

- Selective Use of Filter Functions

The Fir filter has a linear phase response and is always stable, making it suitable for use in audio, video and other applications where phase distortion cannot be tolerated. On the other hand, the Iir filter includes feedback, and can obtain results using fewer coefficients than Fir, for faster execution; it is suitable for applications where time constraints are imposed. However, the Iir filter may be unstable in some situations, and proper care should be taken in its use.

3.15 Issues Related to Cross-Software

3.15.1 Issues Related to Assembly Language Programs

Because the SuperH RISC engine C/C++ compiler supports special instructions of Renesas Technology SuperH RISC engine family as well as standard instructions, almost any kind of program can be written in C language. However, when there is a need to extend performance, often critical sections of code are written in assembly language and combined with C language programs.
This section reviews a number of issues that should be born in mind when combining C language programs with assembly language code, among them:

- Mutual referencing of external names
- function calling Interface

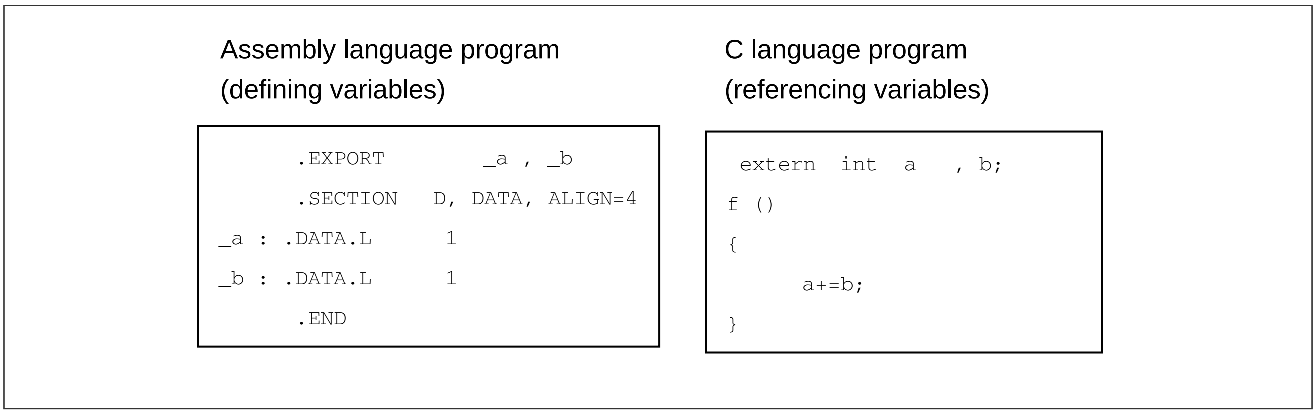
For further details, refer to the SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual.

(1) Mutual Referencing of External Names

(a) Referencing an externally defined name in an assembly language program from a C language program

The following procedure is used to reference an externally defined name in an assembly language program from a C language program.

- In the assembly language program, a symbol name (within 32 characters) which begins with the underscore ("_") is declared as an external definition using the ".EXPORT" or ".GLOBAL" assembler directive.
- In the C language program, the "extern" storage class specifier is used to declare the symbol name, without the leading "_", for external reference.



Assembly language program
(defining variables)

```
       .EXPORT        _a , _b
       .SECTION   D, DATA, ALIGN=4
_a : .DATA.L      1
_b : .DATA.L      1
       .END
```

C language program
(referencing variables)

```
 extern  int  a   , b;
f ()
{
      a+=b;
}
```

Figure 3.13 Example of Use of a C Language Program to Reference Variables Defined Externally in an Assembly Language Program

(b) Referencing an externally defined name in a C language program from an Assembly language program

In the C language program, externally defined names include the following.

- Global variables which are not static storage class variables
- Variable names declared using extern storage class
- Function names for which static storage class is not specified

The externally defined name of a C language program is referenced from an Assembly language program as follows.

- The symbol name (without a leading "_") is externally defined (as a global variable) in the C language program.
- In the assembly language program, the ".IMPORT" or ".GLOBAL" assembler directive is used to declare an external reference of the symbol name, with a leading "_".

C language program
(defining variable)

```
int      a;
```

Assembly language program
(referencing variable)

```
         .IMPORT     _a
         .SECTION P,CODE, ALIGN=2
         MOV.L       A_a, R1
         MOV.L       @R1, R0
         ADD            #1,R0
         RTS
         MOV.L       R0,@R1
A_a:  .DATA.L     _a
         .END
```

Figure 3.14 Example of Use of an Assembly Language Program to Reference Variables Defined Externally in a C Language Program

Note: Function names and external names created from static data members are translated by the C++ compiler using fixed rules. When there is a need to know the external name generated by the compiler, compiler option code=asm or listfile should be used to view the external names generated by the compiler. If C++ functions are defined by adding extern C, external names are generated using rules similar to those for C functions. However, overloading of such functions is not possible.

(2) Function Calling Interface

When either a C language program or an assembly language program calls functions in the other language, the following four rules should be observed by the assembly language program.

(i) Rule relating to the stack pointer
(ii) Rule for allocating and deallocating the stack frame
(iii) Rule relating to registers
(iv) Rule relating to setting and referencing parameters and return values

Here rules (i) to (iii) are explained. For information on (iv), refer to section 3.15.1 (3), Setting and Referencing Parameters and Return Values.

(a) Rule relating to the stack pointer

Valid data should not be saved in the stack area lower than (in the direction of address 0) the address of the stack pointer. Any data saved at addresses lower than the stack pointer may be corrupted as a result of interrupt processing.

(b) Rule for allocating and freeing the stack frame

When a function is called (immediately after execution of a JSR or BSR instruction), the stack pointer points to the lowest address in the stack being used by the calling function. Allocating and setting data at addresses above this (in the direction of address H'FFFFFFFF) is the role of the calling function.

Normally, the RTS instruction is used to return control to the calling function after the area used by the called function is freed. The area at higher addresses than this (the return value address and parameter area) are freed by the calling function.

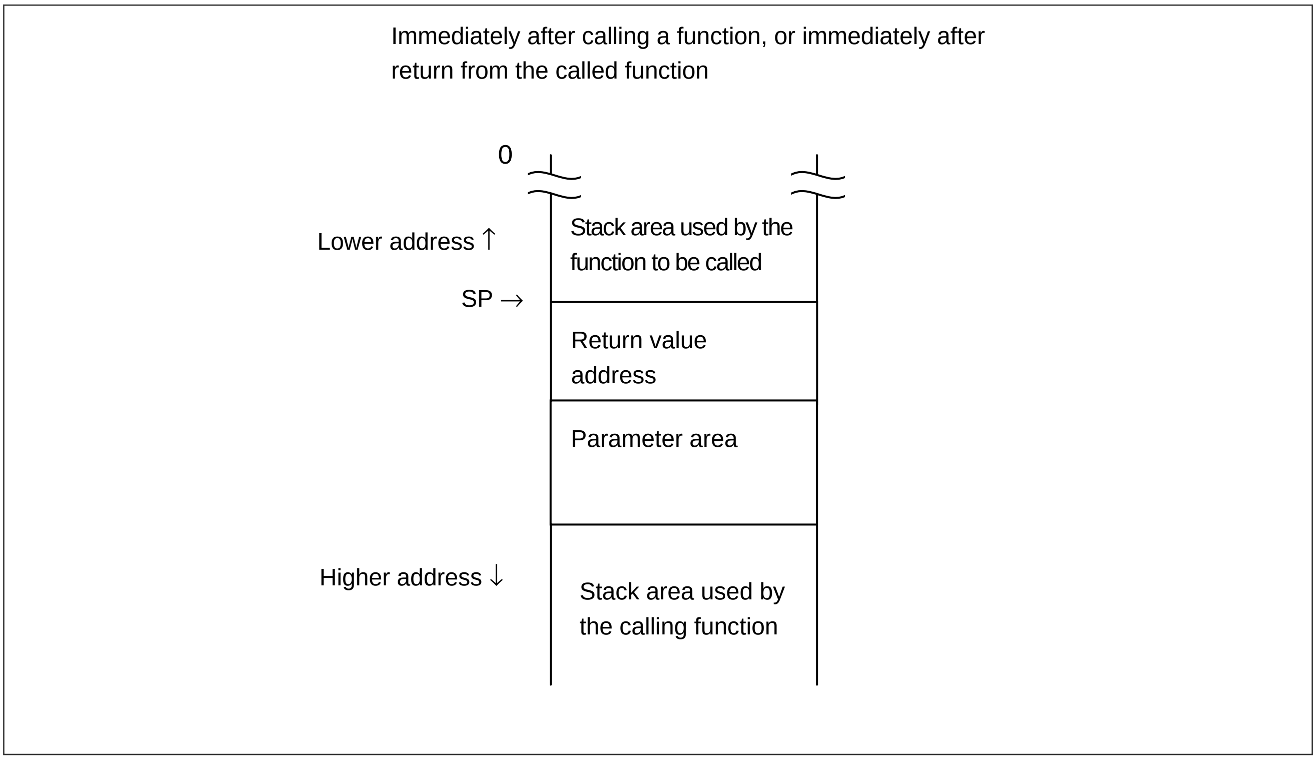

Immediately after calling a function, or immediately after return from the called function
0
Lower address ↑
SP →
Stack area used by the function to be called
Return value address
Parameter area
Higher address ↓
Stack area used by the calling function

Figure 3.15 Allocating and Freeing the Stack Frame

(c) Rule relating to registers

There are registers for which the C/C++ compiler does and does not guarantee that values will be preserved after a function call. Rules for preservation of register contents are indicated in table 3.38.

Table 3.38 Rules for Preservation of Register Contents After Function Calls by a C Language Program

No.	Type	Registers	Notes Regarding Assembly Language Code
1	Registers for which contents are not guaranteed	R0 to R7, FR0 to FR11*1,DR0 to DR10*2, FPUL*1*2, FPSCR*1*2, A0*3, A0G*3, A1*3, A1G*3, M0*3, M1*3, X0*3, X1*3, Y0*3, Y1*3, DSR*3, MOD*3,RS*3, and RE*3	If registers used in a function contain valid data when a program calls the function, the caller must save the data onto the stack or into the register before calling the function. The callee function can use the registers without saving the contained data. However, when **fpscr=safe** is specified, the contents of **FPSCR** are guaranteed.
2	Registers for which contents are guaranteed	R8 to R15, MACH, MACL, PR, FR12 to FR15*1, and DR12 to DR14*2	The data in registers used in functions is saved onto the stack at function entry, and restored from the stack at function exit. Note that data in the MACH and MACL registers are not guaranteed if **macsave=0** is specified. When **gbr=auto** is specified, the contents of **GBR** are guaranteed.

Notes: 1. Single-precision floating point registers for SH-2E, SH2A-FPU, SH-4, and SH-4A.
2. Double-precision floating point registers for SH2A-FPU, SH-4, and SH-4A.
3. DSP registers for SH2-DSP, SH3-DSP, and SH4AL-DSP.

Calling between C language programs and assembly language programs should be as follows.

(i) Calling an assembly language function from a C program

- When an assembly language function is called from a different module, the contents of the PR register should be saved on the stack at the entry point of the assembly language function, and restored from the stack at the exit point.
- When using registers R8 to R15, MACH and MACL within an assembly language function, the register contents should be saved on the stack before use, and restored from the stack after use.
- For details on the parameters passed to an assembly language function, refer to section 3.15.1 (3), Setting and Referencing Parameters and Return Values.

(ii) Calling a C language function from an assembly program

- If there are valid values in the registers R0 to R7, they should be saved on empty registers or the stack before calling the C function.
- For details on the return value passed to an assembly language function, refer to section 3.15.1 (3), Setting and Referencing Parameters and Return Values.

Figure 3.16 shows an example in which an assembly language function g is called from a C language function f, and the assembly language function g in turn calls a C language function h.

C language function f

```
extern  void  g( );


f( )
{
  g( );
}
```

Assembly language function g

```
         .EXPORT     _g                     Declaration of externally defined function g
         .IMPORT        _h                  Declaration of externally referenced function h
         .SECTION    P, CODE, ALIGN=2
_g :  STS.L          PR ,@-R15              Saves the PR register value
MOV.L          R14,@-R15                    Saves the registers used by the function g
MOV.L          R13,@-R15
:
         MOV.L          R2,@R15             Saves the registers used by the function h
         MOV.L          R1,@R15
MOV.L          L_h,R0                       Calls the function h
JSR               @R0
NOP
:                                           Restores the registers used by the function g
MOV.L          @R15+,R13
MOV.L          @R15+,R14
RTS                                         Restores the PR register value
LDS.L          @R15+,PR

L_h : .DATA.L        _h
         .END
```

C language function h

```
h( )
{
      :
      :
}
```

Figure 3.16 Example of Mutual Function Calling between a C Language Program and an Assembly Language Program

(3) Setting and Referencing Parameters and Return Values

The rule imposed by the C/C++ compiler for setting and referencing parameters and return values differs depending on whether the types of the parameters or return value are explicitly declared in the function declaration. In C language, a function declaration in which the types of the parameters and the return value are made explicit is called a prototype function declaration.

In the following, after first discussing the general rules for parameters and return values in C language, the area for parameter allocation and method of allocation, as well as setting of the return value location will be explained.

(a) General rules for parameters and return values in C language programs

(i) Passing parameters

A function must be called only after parameter values have been copied to areas allocated for parameters in registers or on the stack. After control is returned to the calling function, the calling function never references the areas allocated to parameters, and so the called function can modify the values of parameters without any direct effect on processing by the calling function.

(ii) Rules for type conversion

When passing parameters, or when returning a value, automatic type conversions are performed.
The rules for type conversions are shown in table 3.39.

Table 3.39 Rules for type conversions

Type Conversion	Conversion Method
Type conversion of a parameter whose type has been declared	If the parameter type is declared by a prototype declaration, the parameter is converted to the declared type.
Type conversion of a parameter whose type has not been declared	If the parameter type has not been declared by a prototype declaration, the parameter is converted according to the following rules. • Parameters of type char, unsigned char, short, and unsigned short are converted to the int type. • Parameters of type float are converted to the double type. • Types other than the above are not converted.
Type conversion of return values	Return values are converted to the type returned by the function.

Example 1: Type is declared by a prototype declaration

```
 long f();
 long f()
{
       float x;
             :
             :
       return x;
 }
```

The return value x is converted to the long type according to the prototype declaration.

Example 2: Similar to Example 1, but type is not declared by a prototype declaration

```
void p(int,...);

long f()
{
     char c;
           :
   p(1.0, c);
           :
}
```

Because the type of the corresponding parameter is int, the first parameter is converted to the type int.
There is no type for the parameter corresponding to the second parameter, so it is converted to the type int.

Example 3: Similar to Example 2; types are not declared by a prototype declaration

When parameter types are not declared by a prototype declaration, the same types should be specified on the called and calling sides, to ensure that parameters are passed correctly. If types are not in agreement, correct operation is not guaranteed.

```
void f(x)
float x;
{
        :
        :
}

void main()
{
     float x;
     f(x);
}
```

In this example, there is no prototype declaration for the parameters of function f, and so when function f is called by the function main, the parameter x is converted to the type double. On the other hand, the parameter is declared as the float type by the function f, and so correct passing of the parameter is not possible. Either parameter types should be declared by a prototype declaration, or else the parameter declaration by function f should be changed to the double type.
Parameter types can be correctly declared using a prototype declaration as follows.

```
void f(float x)
{
        :
        :
}
```

```
void main()
{
     float x;
     f(x);
}
```

(b) Allocating area for parameters in a C language program

Registers may be allocated to parameters, or, if this is not possible, stack area may be used for parameters. Areas for allocation to parameters are shown in figure 3.17; general rules for allocating memory for parameters appear in table 3.40.

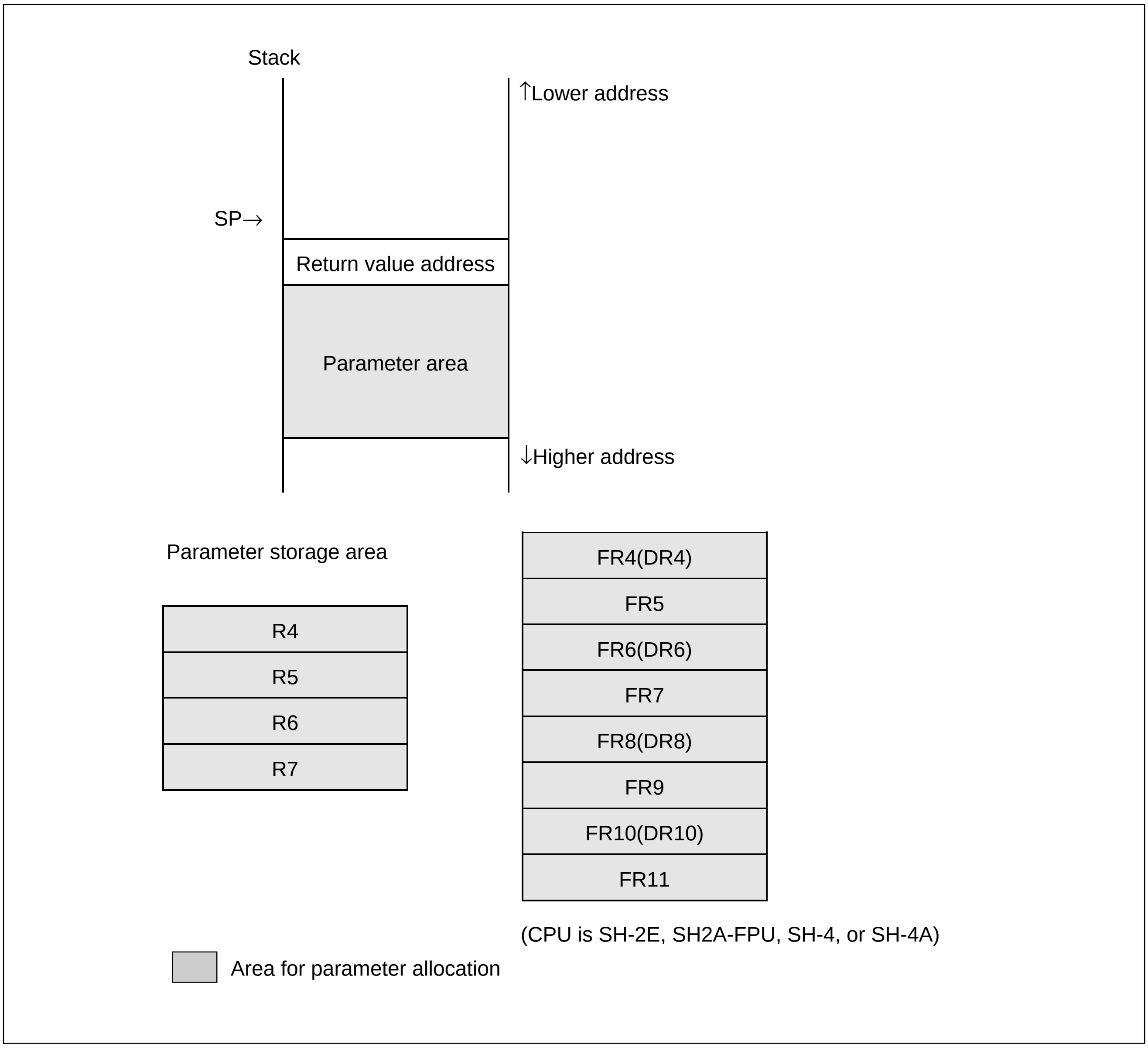


Figure 3.17 Areas for Allocation to Parameters in C Language Programs

Table 3.40 General Rules for Allocating Memory to Parameters in C Language Programs

Allocating Rules		
Parameters Passed through Registers		
Registers Used for Parameter Storage	**Available Parameter Type**	**Parameter Passed through Stack**
R4 to R7	char, unsigned char, bool, short, unsigned short, int, unsigned int, long, unsigned long, float (when CPU is other than SH-2E, SH2A-FPU, SH-4, or SH-4A), pointer, pointer to a data member, and reference	(1) Parameters whose types are other than target types for register passing (2) Parameters of a function which has been declared by a prototype declaration to have variable-number parameters*3 (3) When other parameters are already allocated to R4 to R7. (4) When other parameters are already allocated to FR4 (DR4) to FR11(DR10). (5) long long type and unsigned long long type parameters (6) _ _fixed type, long _ _fixed type, _ _accum type, and long _ _accum type parameters
FR4 to FR11*1	For SH-2E • Parameter is float type. • Parameter is double type and double=float is specified. For SH2A-FPU, SH-4, or SH-4A • Parameter type is float type and fpu=double is not specified. • Parameter type is double type and fpu=single is specified.	
DR4 to DR10*2	For SH2A-FPU, SH-4, or SH-4A • Parameter type is double type and fpu=single is not specified. • Parameter type is float type and fpu=double is specified.	

Notes: 1. Single-precision floating-point registers for SH-2E, SH2A-FPU, SH-4, and SH-4A.

2. Double-precision floating-point registers for SH2A-FPU, SH-4, and SH-4A.

3. If a function has been declared to have variable parameters by a prototype declaration, parameters which do not have a corresponding type in the declaration and the immediately preceding parameter are allocated to a stack.

Examples:

```
int f2(int, int, int, int,...);
f2(a, b, c, x, y, z)
{
                :
}
```

Up until the fourth parameter, normally register space is allocated; but here stack area is allocated for x, y, and z as well.

(i) Allocating registers for parameter storage

Registers are allocated for parameter storage in the order of declarations in the source program, starting from the register with the smallest number. An example of allocation of registers for parameter storage is shown in example 1.

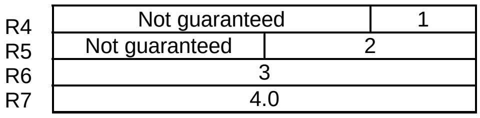
(ii) Allocation of stack area for parameters

Stack area is allocated for parameter storage in the order of declarations in the source program, starting with the lowest address. Examples of allocation of stack area for parameter storage are shown in examples 2 through 8.

[Important information regarding parameters with structure and shared types]
When preparing parameters with structure and shared types, these types are always aligned with four-byte boundaries, and memory areas in multiples of four bytes are always used for them. This is because the stack pointer in SuperH microcomputers changes in four-byte units.

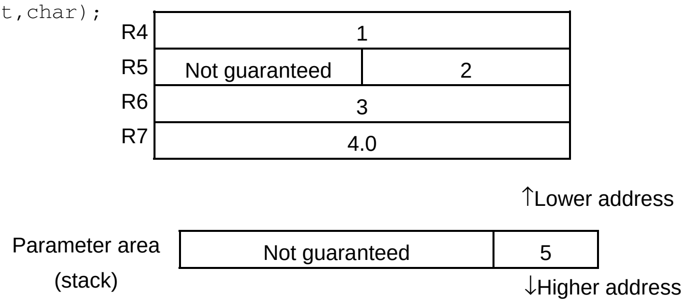
Example 1: The registers R4 through R7 are allocated, in the order of declaration, to parameters with the types of the registers.

```
int f(char,short,int,float);
    :
f(1,2,3,4.0);
    :
```



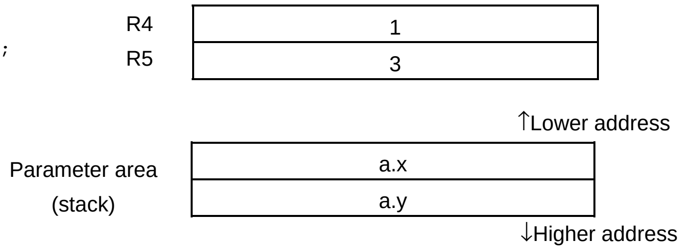
Example 2: Stack area is allocated for parameters for which register allocation is not possible. When allocating stack area for parameters of type (unsigned) char or (unsigned) short, these are expanded to four bytes for allocation.

```
int f(int,short,long,float,char);
    :
f(1,2,3,4.0,5);
    :
```



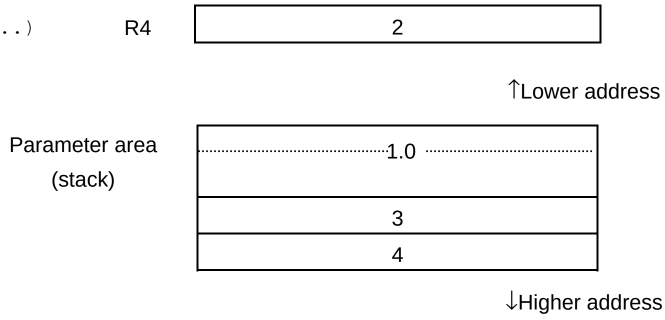
Example 3: Stack area is allocated for parameters of types that cannot be assigned to registers.

```
struct s{int x,y;}a;
int f(int,struct s,int);
    :
f(1,a,3);
    :
```



Example 4: When a prototype declaration is used to declare a function having a variable number of parameters, parameters with no declared type, and the immediately preceding parameters, are allocated on the stack in the order of declaration.

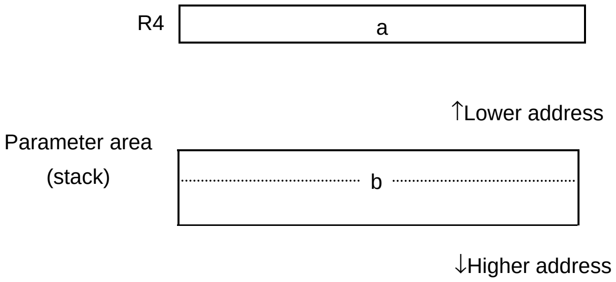
```
int f(double, int, int...)
    :
f(1.0, 2, 3, 4)
    :
```



Example 5: Case where there is no prototype declaration

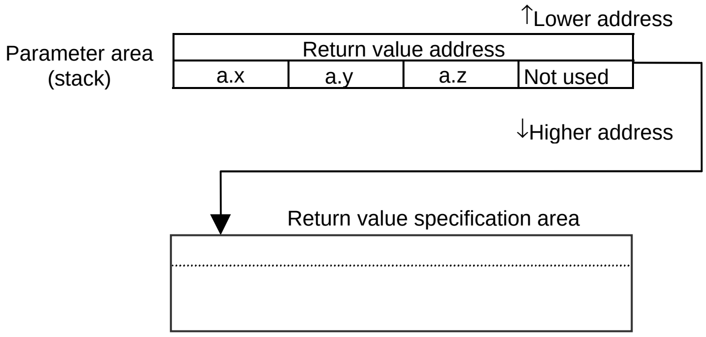
→ char types are expanded to int types, and float types to double types for allocation.

```
 int f ( )
 char  a ;
 float  b;


      f (a ,b)
```



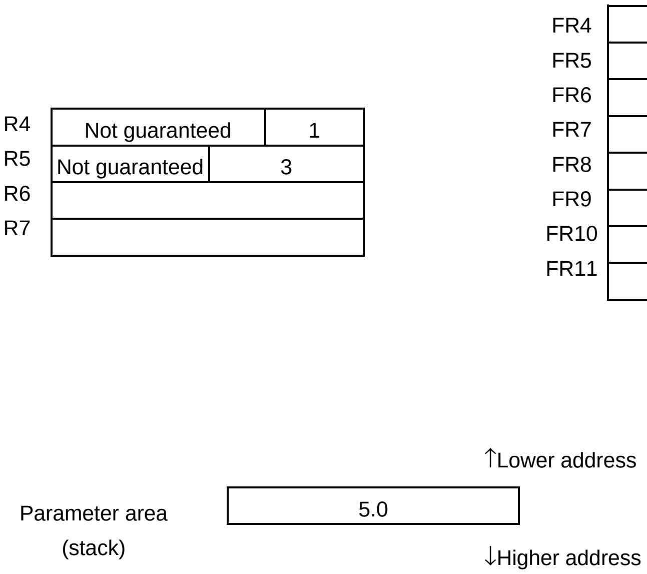
Example 6: When the type returned by a function exceeds four bytes or is a class, a return value address is set immediately before the parameter area. Also, when the class size is not a multiple of four bytes, an empty area occurs.

```
struct s{char x,y,z;}a;
double f(struct s);
    :
f(a);
```



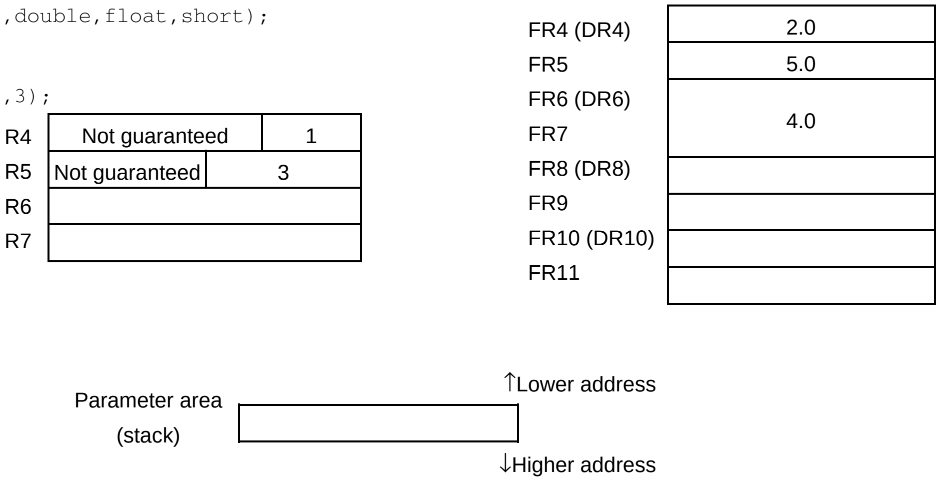
Example 7: When the CPU is SH-2E, float type parameters are assigned to the FPU registers.

```
int f(char,float,short,float,double);
    :
f(1,2.0,3,4.0,5.0);
    :
```



Example 8: When the CPU is the SH2A-FPU, SH-4,SH-4A and there is no -fpu option specified, float/double type parameters are assigned to FPU registers.

```
int f(char,float,double,float,short);
    :
f(1,2.0, 4.0,5.0,3);
    :
```



(c) Location for setting return values in C language programs

Depending on the type of a function's return value, the return value is placed in a register or on the stack. The relation between the return value type and the returned location is described in table 3.41.

When the return value of a function is placed on the stack, the return value is set in the area pointed to by the return value address. On the calling side, in addition to securing area for parameters, an area for the return value is also secured, and after setting the address as the return value address, the function is called (cf. figure 3.18). When the function return value is of type void, no return value is set.

Table 3.41 Types and Locations of Return Values in C Language Programs

No	Return Value Type	Return Value Storage Area
1	(signed) char, unsigned char, (signed) short, unsigned short, (signed) int, unsigned int, long, unsigned long, float, pointer, bool, reference, pointer to data member	R0: 32 bits The contents of the upper three bytes of (signed) char, or unsigned char and the contents of the upper two bytes of (signed) short or unsigned short are not guaranteed. However, when the **rtnext** option is specified, sign extension is performed for (signed) char or (signed) short type, and zero extension is performed for unsigned char or unsigned short type. FR0: 32 bits (1) For SH-2E • Return value is float type. • Return value is double type and **double=float** is specified. (2) For SH2A-FPU, SH-4, or SH-4A • Return value is float type and **fpu=double** is not specified. • Return value is floating-point type and **fpu=single** is specified
2	double, long double s tructure, union, class, pointer to function member	Return value setting area (memory) DR0: 64 bits For SH2A-FPU, SH-4, or SH-4A • Return value is double type and **fpu=single** is not specified. • Return value is floating-point type and **fpu=double** is specified.
3	(signed) long long and unsigned long long	Return value setting area (memory)
4	_ _fixed, long _ _fixed, _ _accum, and long _ _accum	Return value setting area (memory)

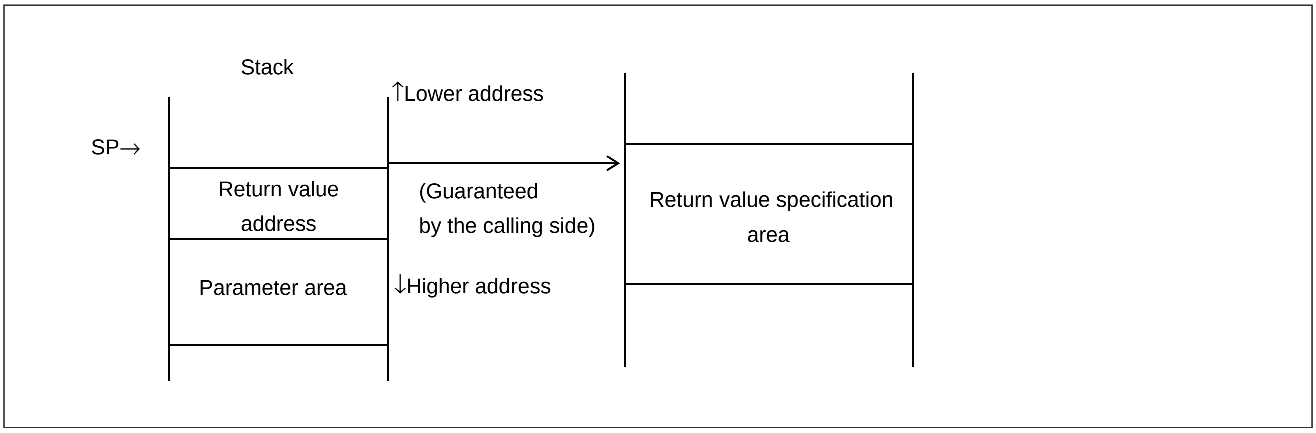


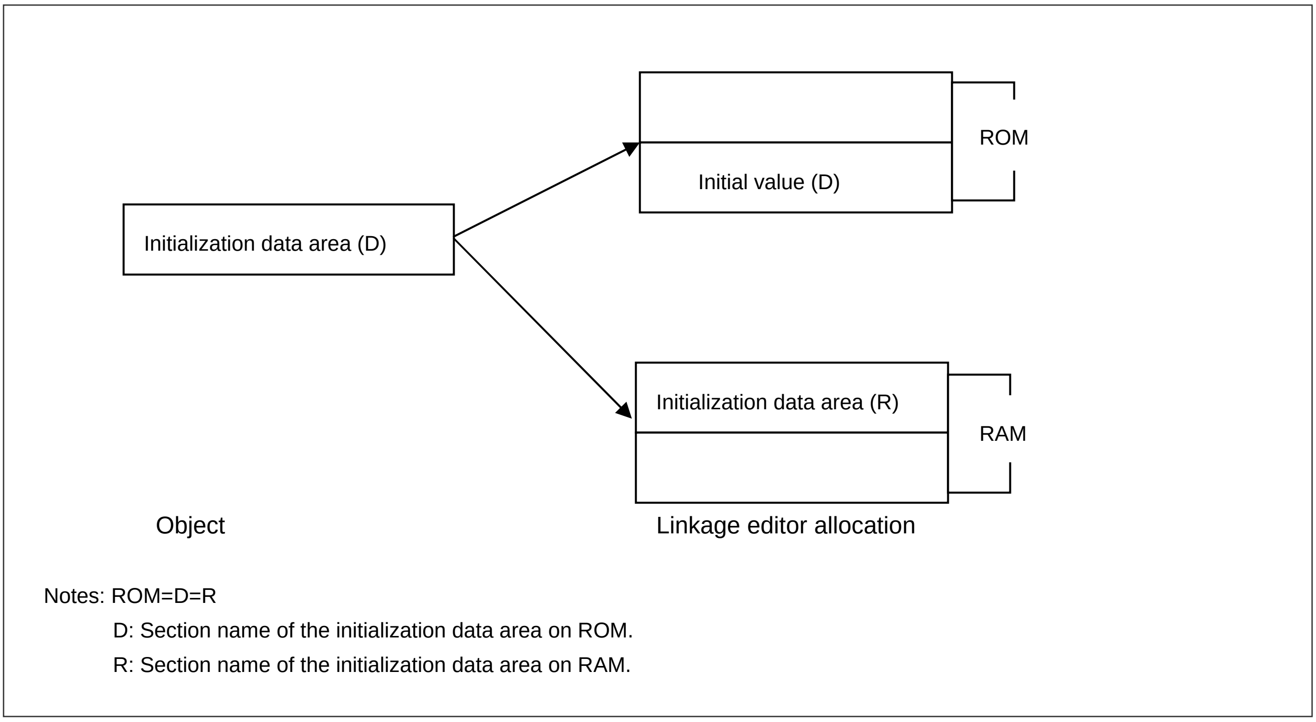
Figure 3.18 Area for Return Values When Using the Stack in a C Program

3.15.2 Use With the Optimization Linkage Editor

(1) ROM Support Function

When writing a load module to ROM, the initialization data area is also written to ROM. However, actual data operations must be performed in RAM, and so on startup the initialization data area must be copied from ROM to RAM. By using the ROM support function of the linkage editor, this processing is simplified.
In order to use the ROM support function, at linkage the option "ROM=D=R" (where D is the section name of the initialization data area in ROM, and R is the section name of the initialization data area in RAM) must be specified.
The ROM support function performs the following operations.

(a) An area in RAM of the same size as the initialization data area in ROM must be secured. Figure 3.19 illustrates the procedure to copy the initialization data area from ROM to RAM.


ROM
Initial value (D)
Initialization data area (D)
Initialization data area (R)
RAM
Object
Linkage editor allocation

Notes: ROM=D=R
D: Section name of the initialization data area on ROM.
R: Section name of the initialization data area on RAM.

Figure 3.19 Memory Allocation Using the ROM Support Function

(b) Address resolution is performed automatically so that symbols declared in the initialization data area are referenced using addresses in RAM.

The user must include, in the startup routine, processing to copy data in ROM to RAM. For an example of this, refer to section 2.2.4, Creation of the Initialization Unit.
For further information on the ROM support function, refer to the SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual.
This function is supported in ver.4.0 or later of the H series linkage editor.

(2) Important Information on Linking

Table 3.42 describes procedures for dealing with error messages output when linking the relocatable object files generated by the C/C++ compiler.

Table 3.42 Responses to Error Messages at Linkage

No.	Error message	Areas to check	Countermeasures
1	At linkage, error no. L1100(314)*, cannot find section, is output.	Have the compiler output section names been specified, in capital letters, in the start option of the linkage editor.	Be sure to specify the correct section names.
2	At linkage, error no. L1160(105)*, undefined external symbol, is output.	When there is mutual referencing of variables between a C/C++ program and an assembly language program, check whether names are preceded by an underscore in the assembly language program.	The correct variable names must be used in referencing.
		Check whether C library functions are not used in the C/C++ program.	At linkage, the standard library should be specified as an input library.
		Check whether undefined symbol names do not begin with an underscore. (used by runtime routines in the standard library)	
		Check whether the standard I/O library is used for C library functions.	Create low-level interface routines for linking.
3	C/C++ source-level debugging is not possible.	Check whether the debug option has been specified at compile and linkage.	Be sure to specify the debug option when compiling and linking. When specifying the sdebug option during linking, be sure to also load the debugging information file into the debugger.
		Check whether version 5.3 or later of the linkage editor is being used.	Version 5.3 or later of the linkage editor should be used.
4	At linkage, error no. L2330(108)*, relocation size overflow, is output.	Check whether, in GBR base variable specification, the offset of the specified variable is within limits.	For variables which exceed limits, delete any #pragma gbr_base/gbr_base1 declarations.
5	At linkage, error no. L2300(104)*, duplicate symbol, is output.	Check whether there are external definitions of variables or functions with the same name in multiple files.	Change the name, or use the static specification.
		Check for external definitions of variables or functions within a header file included by multiple files. (Similarly for functions specified using #pragma inline/inline_asm)	Use the static specification.

Note: * The error numbers before the parentheses belong to Linker ver. 7 and later, while those enclosed in the parentheses belong to ver. 6 and earlier.

3.15.3 Use With the Simulator-Debugger

When the simulator-debugger is used to execute a load module, a "MEMORY ACCESS ERROR" may be generated. For security, one of the following methods should be used to avoid this error.

(a) Use the same memory mapping as in the actual CPU even when using the simulator-debugger (the total number of bytes in one section should always be a multiple of four).

(b) At linkage, link the dummy section, created by using the following assembly language program, after all sections except for the P section.

Assembly language program:

```
.SECTION DM,DUMMY,ALIGN=1
.RES.B      3
.END
```

Examples of linking

- When using command options:

```
-START=P,C,DM/0400,B,DM,D,DM/01000000
```

- When using a subcommand file:

```
START P,C,DM/0400,B,DM,D,DM/01000000
```

The following is important information related to source-level debugging using the simulator-debugger.

(a) Ver. 6.0 or later of the linkage editor should be used.

(b) When compiling, the -debug option should be used, and when linking, the sdebug option should be specified.

(c) In some cases, local symbols of a function cannot be referenced within that function.

(d) When multiple statements are included in a single line of source code, only a single statement can be displayed.

(e) Source code lines which have been eliminated through optimization cannot be debugged.

(f) When swapping of lines and other changes are made as a result of debugging, the order of program execution and disassembled display may differ from the order in the source list.

Example:
C language program

```
12 for (i=0; i<6; i++)
13 {
14        j = i+1;
15       j++;
16 }
17 j++;
```

Disassembled display by the simulator/debugger

```
14 j = i+1;
12 for (i=0; i<6; i++)
17         j++;
```

(g) In for or while statement, disassembled display may be displayed twice; at loop statement entry and exit points.

(1) Profile function

(a) Using the Profile function

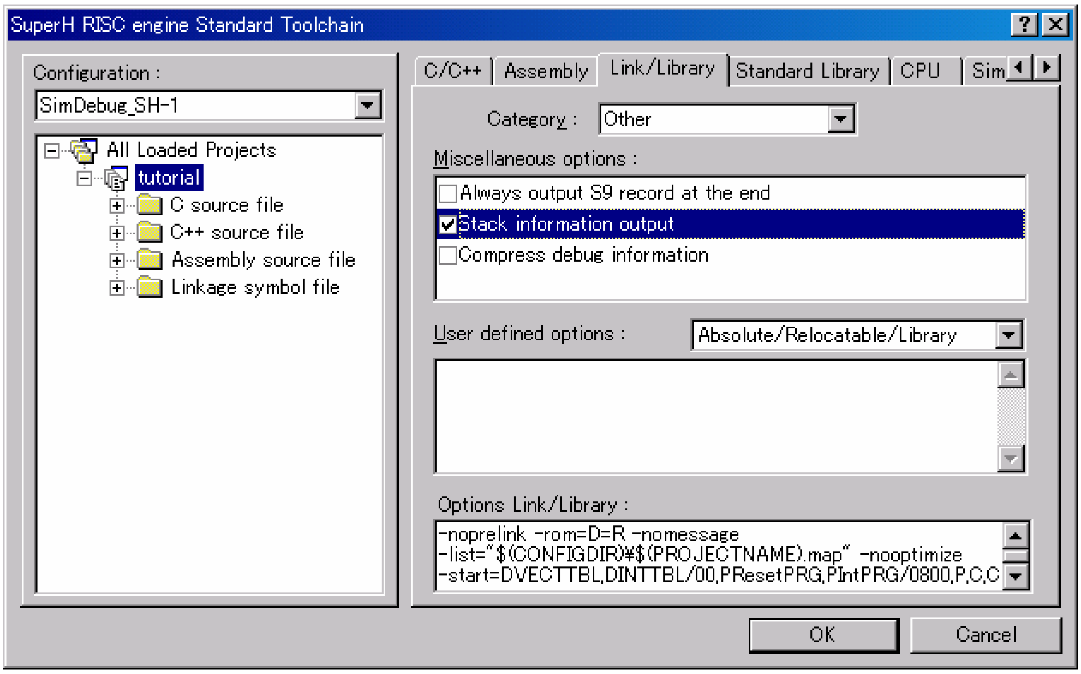
(i) Stack information files

The profile function allows the HEW to read the stack information files (extension: ".SNI") which are output by the Optimizing Linker (ver. 7.0 or later). Each of these files contains information related to the calling of static functions in the corresponding source file. Reading the stack information file makes it possible for the HEW to display this information to do with the calling of functions without executing the user application (i.e. before measuring the profile data). (However, this feature is not available when [Show Only Executed Function] is checked.)

When the HEW does not read any stack information files, the data about the functions executed during measurement will be displayed by the profile function.

Whether to read or not the stack information file can be specified by turning on or off the [Load Stack Information file (SNI file)] checkbox in the Load Program dialog box.

To make the linker create a stack information file, select “Other” from the “Category:” list box and check the “Stack information output” box in the “Link/Library" pane of the Standard Toolchain dialog box specified by HEW linker option. Then build the information file.



SuperH RISC engine Standard Toolchain

Configuration : SimDebug_SH-1

Category : Other

Miscellaneous options :

☐ Always output S9 record at the end
☑ Stack information output
☐ Compress debug information

User defined options : Absolute/Relocatable/Library

Options Link/Library :

```
-noprelink -rom=D=R -nomessage
-list="$(CONFIGDIR)¥$(PROJECTNAME).map" -nooptimize
-start=DVECTTBL,DINTTBL/00,PResetPRG,PIntPRG/0800,P,C,C
```

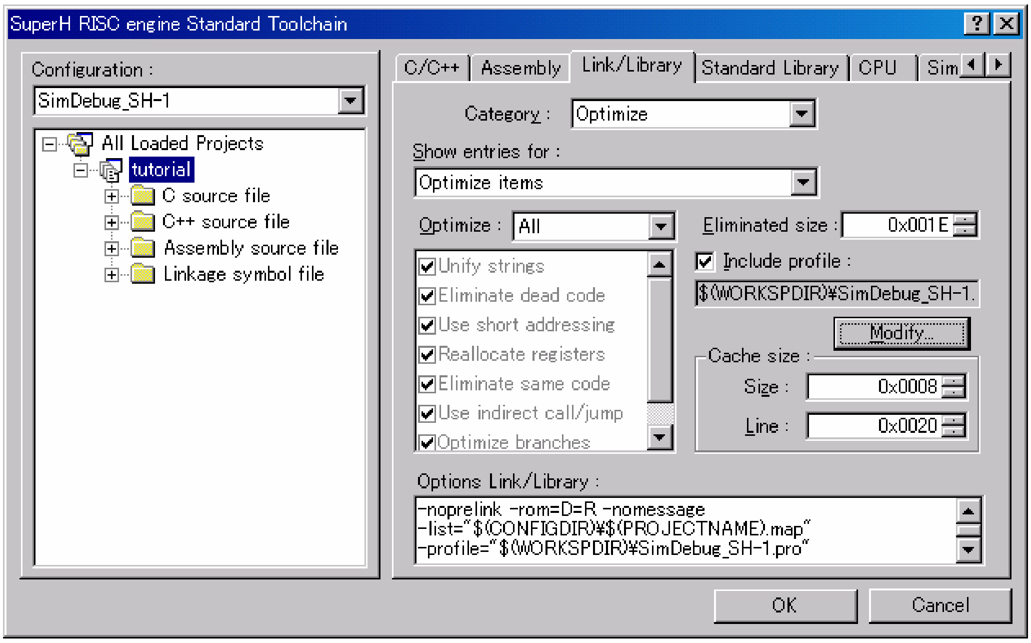
Figure 3.20 The Category:[Other] Dialog Box

(ii) Profile information files

To create a profile information file, select the “Output Profile Information Files…” menu option from the pop-up menu of the Profile window and specify the file name, after measuring a profile data of the application program.

This file contains information on the number of times functions are called and global variables are accessed. The Optimizing Linker (ver. 7.0 or later) is capable of reading the profile information file and optimizing the allocation of functions and variables in correspondence with the status of the actual operation of the program.

To input the profiler information file to the linker, select “Optimize” from the “Category:” list box and check the “Include Profile:” box in the “Link/Library” pane of the Standard Toolchain dialog box, and specify the name of the profile information file.



SuperH RISC engine Standard Toolchain

Configuration : SimDebug_SH-1

- All Loaded Projects
  - tutorial
    - C source file
    - C++ source file
    - Assembly source file
    - Linkage symbol file

Category : Optimize

Show entries for : Optimize items

Optimize : All

Eliminated size : 0x001E

☑ Unify strings
☑ Eliminate dead code
☑ Use short addressing
☑ Reallocate registers
☑ Eliminate same code
☑ Use indirect call/jump
☑ Optimize branches

☑ Include profile : $(WORKSPDIR)¥SimDebug_SH-1.

Cache size :
Size : 0x0008
Line : 0x0020

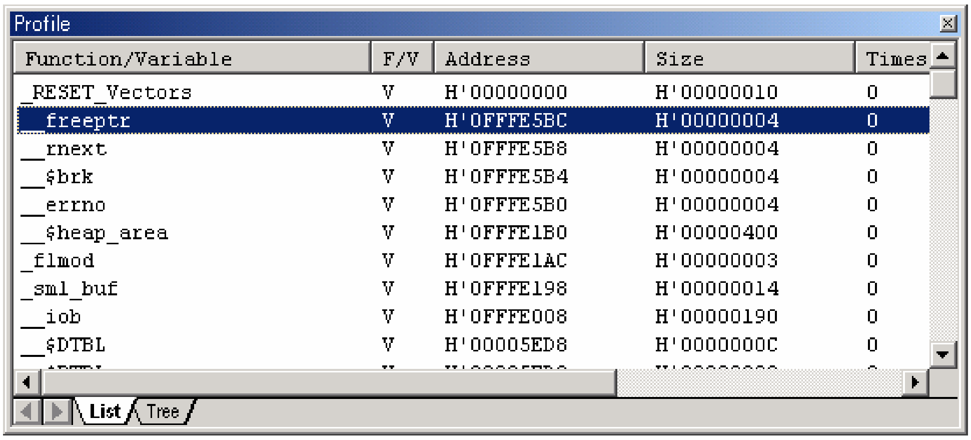
Options Link/Library :

```
-noprelink -rom=D=R -nomessage
-list="$(CONFIGDIR)¥$(PROJECTNAME).map"
-profile="$(WORKSPDIR)¥SimDebug_SH-1.pro"
```

Figure 3.21 Category:[optimize] Dialog Box

(iii) Profile window

Select [View-> Performance->Profile] to open the Profile window. This menu item is displayed when a load module is loaded.



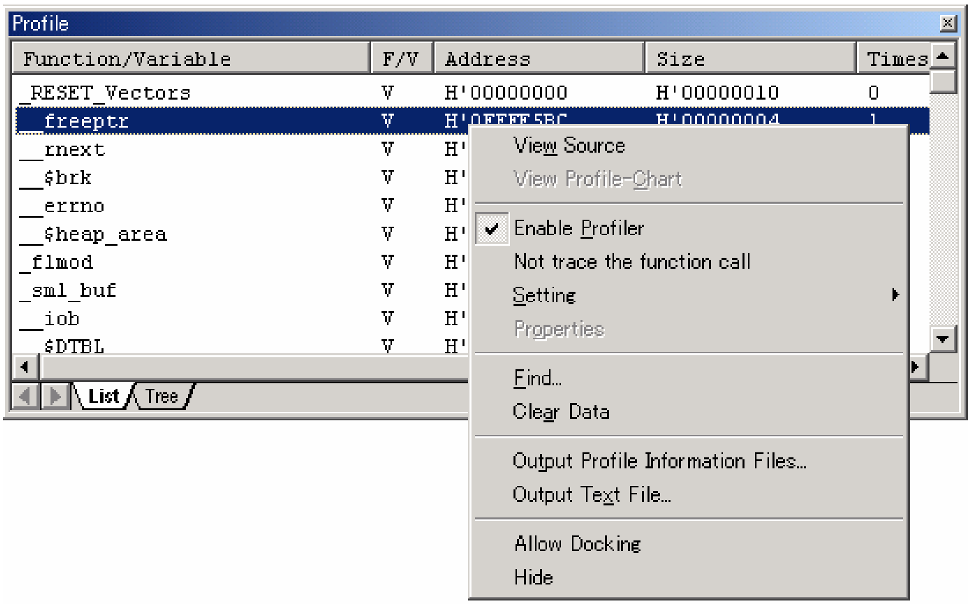
Profile

Function/Variable	F/V	Address	Size	Times
_RESET_Vectors	V	H'00000000	H'00000010	0
__freeptr	V	H'0FFFE5BC	H'00000004	0
__rnext	V	H'0FFFE5B8	H'00000004	0
__$brk	V	H'0FFFE5B4	H'00000004	0
__errno	V	H'0FFFE5B0	H'00000004	0
__$heap_area	V	H'0FFFE1B0	H'00000400	0
_flmod	V	H'0FFFE1AC	H'00000003	0
_sml_buf	V	H'0FFFE198	H'00000014	0
__iob	V	H'0FFFE008	H'00000190	0
__$DTBL	V	H'00005ED8	H'0000000C	0

Figure 3.22 Profile Window

(iv) Profile window menu

Select “Enable” from the pop-up menu of the Profile window. (The item on the menu will be checked.)



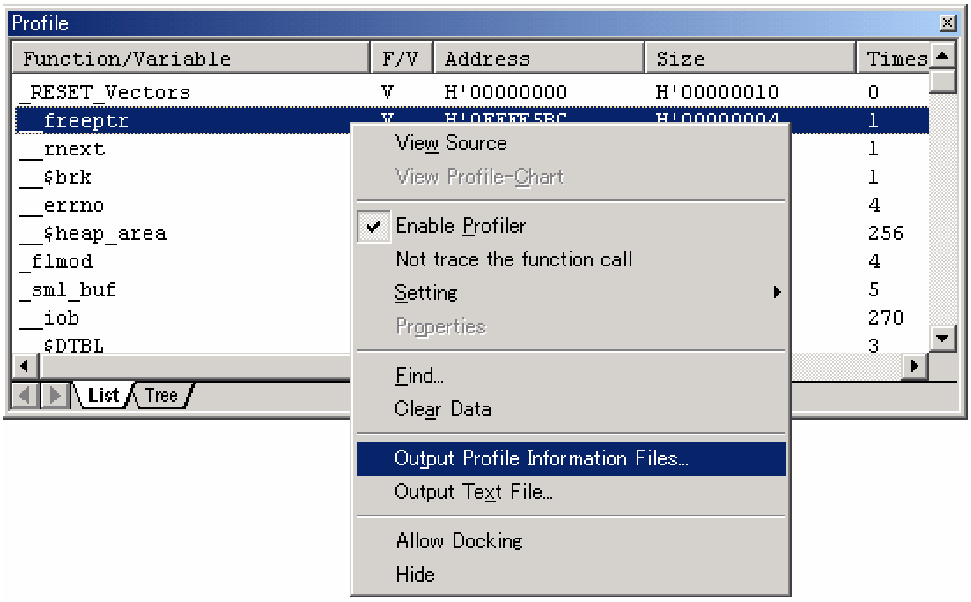
Function/Variable	F/V	Address	Size	Times
_RESET_Vectors	V	H'00000000	H'00000010	0
__freeptr	V	H'0FFFF5BC	H'00000004	1
__rnext	V	H'		
__$brk	V	H'		
__errno	V	H'		
__$heap_area	V	H'		
_flmod	V	H'		
_sml_buf	V	H'		
__iob	V	H'		
__$DTBL	V	H'		

Figure 3.23 Profile Window Menu (Enable Profiler)

(v) Set a breakpoint with a condition that the Profile measument be stopped. (The Profile measument can be manually stopped without setting the condition.)

(vi) If the stop condition set in (v) above is satisfied, or the execution is stopped manually or for some other reason, the measument results are displayed in the Profile window.

(vii) To create a profile information file, select the “Save Profile Information Files…” menu option from the pop-up menu.



Function/Variable	F/V	Address	Size	Times
_RESET_Vectors	V	H'00000000	H'00000010	0
__freeptr	V	H'0FFFF5BC	H'00000004	1
__rnext				1
__$brk				1
__errno				4
__$heap_area				256
_flmod				4
_sml_buf				5
__iob				270
__$DTBL				3

Figure 3.24 Profile Window Menu (Save Profile Information Files…)

(b) Notes:

(i) The number of executed cycles for an application program as measured by the profile function includes a margin of error. The profile function only allows the measurement of the proportions of execution time that the functions occupy in the overall execution of the application program. Use the Performance Analysis function to precisely measure the numbers of executed cycles.

(ii) The names of the corresponding functions may not be displayed when the profile information on a load module with no debug information is measured.

(iii) The stack information file (extension: ".SNI") must be in the same directory as the load module file (extension: ".ABS").

(iv) It is not possible to store the results of measurement.

(v) It is not possible to edit the results of measurement.

(c) Overview of the Profile function

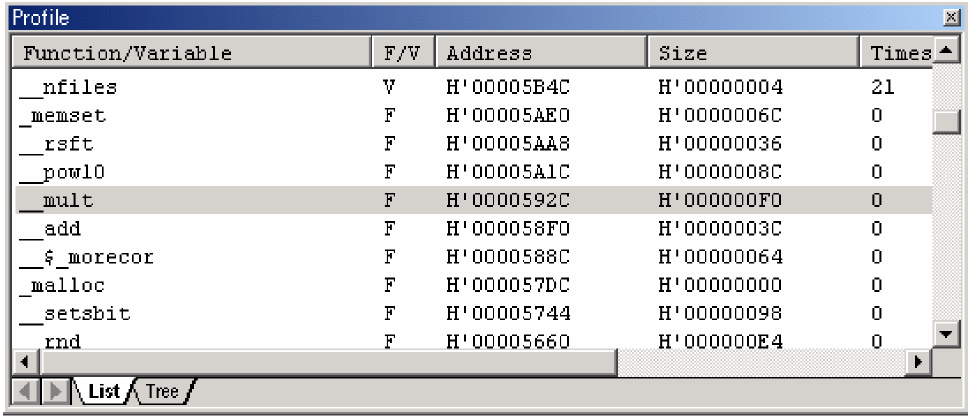
The Profile function measures the execution performance of an application program in terms of the execution count of functions in it. The Profile function allows you to identify the parts of the program causing performance degradation and the causes of the degradation.

(i) Profile window

The Profile window has two tabs; a "List" tab and a "Tree" tab.

- List Tab

  This tab lists functions and global variables and displays the profile data for each function and variable.

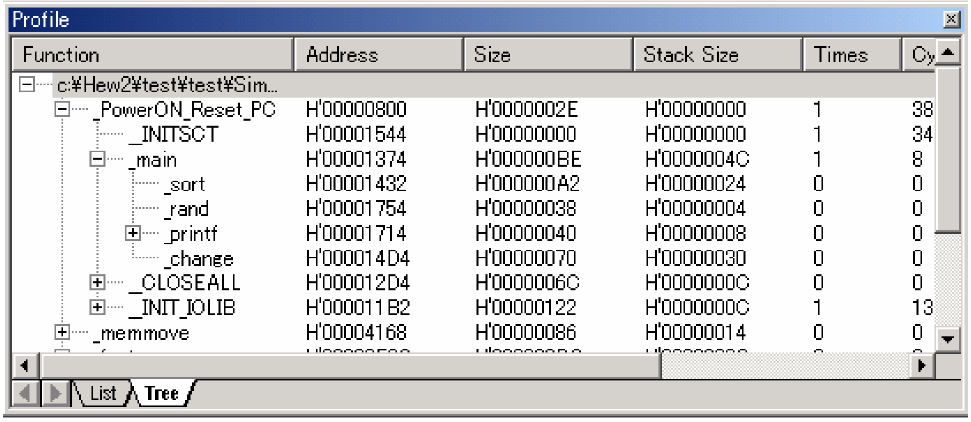


Function/Variable	F/V	Address	Size	Times
__nfiles	V	H'00005B4C	H'00000004	21
_memset	F	H'00005AE0	H'0000006C	0
__rsft	F	H'00005AA8	H'00000036	0
__pow10	F	H'00005A1C	H'0000008C	0
__mult	F	H'0000592C	H'000000F0	0
__add	F	H'000058F0	H'0000003C	0
__$_morecor	F	H'0000588C	H'00000064	0
_malloc	F	H'000057DC	H'00000000	0
__setsbit	F	H'00005744	H'00000098	0
rnd	F	H'00005660	H'000000E4	0

Figure 3.25 List Tab

- Tree Tab

  This tab displays the relation of function calls as a tree diagram along with the profile data that are values when the function is called.



Profile

Function	Address	Size	Stack Size	Times	Cy
c:¥Hew2¥test¥test¥Sim...					
_PowerON_Reset_PC	H'00000800	H'0000002E	H'00000000	1	38
_INITSCT	H'00001544	H'00000000	H'00000000	1	34
_main	H'00001374	H'000000BE	H'0000004C	1	8
_sort	H'00001432	H'000000A2	H'00000024	0	0
_rand	H'00001754	H'00000038	H'00000004	0	0
_printf	H'00001714	H'00000040	H'00000008	0	0
_change	H'000014D4	H'00000070	H'00000030	0	0
_CLOSEALL	H'000012D4	H'0000006C	H'0000000C	0	0
_INIT_IOLIB	H'000011B2	H'00000122	H'0000000C	1	13
_memmove	H'00004168	H'00000086	H'00000014	0	0

Figure 3.26 Profile-Tree Window

- Profile-Chart Window

  The Profile-Chart window displays the relation of calls for a specific function. This window displays the specified function in the middle, with the callers of the function on the left and the callees of the function on the right. The numbers of times the function calls the called functions or is called by the calling functions are also displayed in this window.

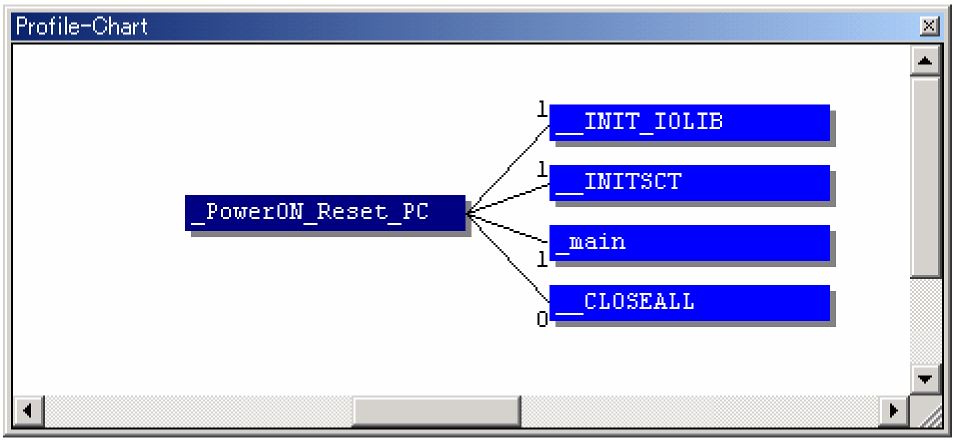


Figure 3.27 Profile-Chart Window

(ii) Types and Purposes of Displayed Data

- Address

  Function: Displays the addresses of functions (global variables).
  Use: You can see the locations in memory to which the functions are allocated. Sorting the list of functions and global variables in order of their addresses allows the user to view the way the items are allocated in the memory space.

  Note: The sorted display is only available on the “List” tab.

- Size

  Function: Displays the sizes of functions (global variables).
  Use: Sorting in order of size makes it easy to find small functions that are frequently called. Setting such functions as inline may reduce the overhead of function calls.
  If you are using a microcomputer, which incorporates a cache memory, more of the cache memory will need to be updated when you execute larger functions. This information allows you to check if those functions that may cause cache misses are frequently called.

  Note: The sorted display is only available on the “List” tab.

- Stack Size

  Function: Displays the sizes of the stack used by functions.
  Use: When there is deep nesting of function calls, pursue the route of the function calls and obtain the total stack size for all of the functions on that route to estimate the amount of stack being used.

  Note: This is displayed with the “Tree” tab.
  The sizes of the stacks used by functions are set in the stack information file. If the stack information file is not read, all the stack sizes are displayed as 0. If you include the output profile information file (extension:”.PRO”) in the stack analysis tool (H series Call Walker) in such a case, correct values will not be displayed.

- Times

  Function: Displays the number of calls to each function or the number of accesses made to variables.
  Use: Sorting by the number of calls or accesses makes it easy to identify the frequently called functions and frequently accessed global variables.

  Note: The sorted display is only available on the “List” tab.

- Others

  Measurement of a variety of target-specific data is also available. For details, refer to the simulator or emulator manual for the target platform that you are using.

(iii) Display setting

If you select the “Setting…” from the pop-up menu, the Setting list appears. Any part with a problem can be easily detected by customizing the display with this list.

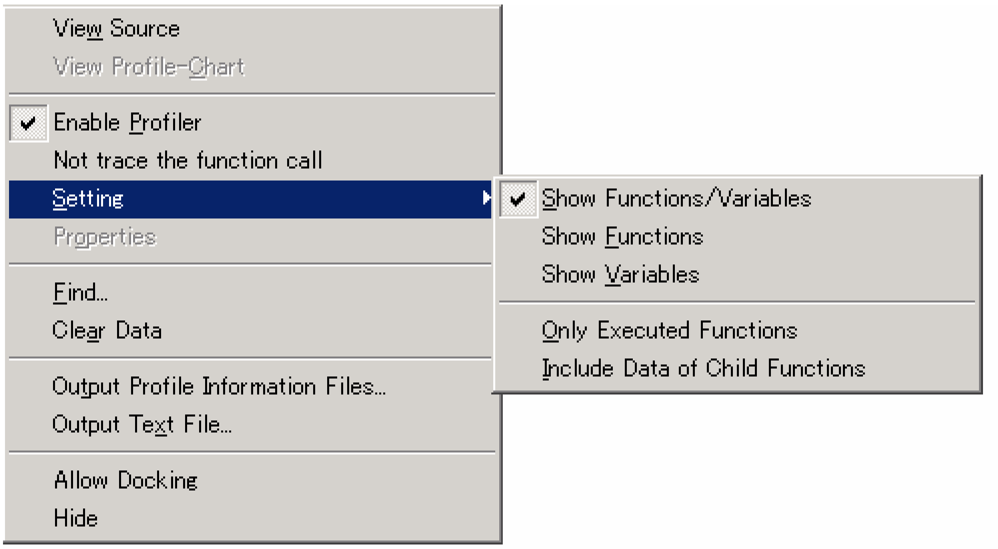
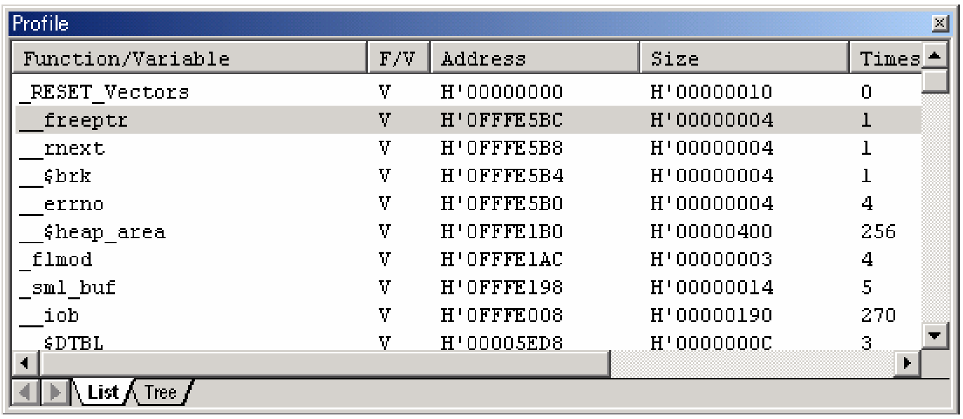


Figure 3.28 Profile Window Pop-up Menu

- Show Functions/Variables

  Function: Displays the information on functions and variables in the window.

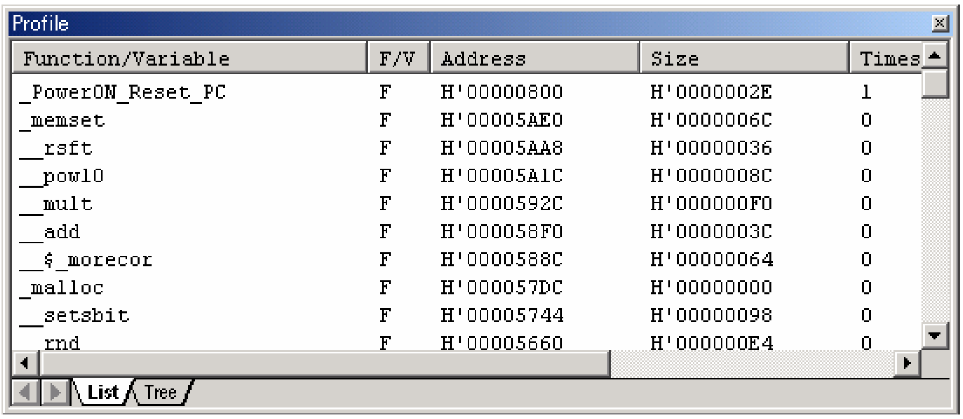


Function/Variable	F/V	Address	Size	Times
_RESET_Vectors	V	H'00000000	H'00000010	0
__freeptr	V	H'0FFFE5BC	H'00000004	1
__rnext	V	H'0FFFE5B8	H'00000004	1
__$brk	V	H'0FFFE5B4	H'00000004	1
__errno	V	H'0FFFE5B0	H'00000004	4
__$heap_area	V	H'0FFFE1B0	H'00000400	256
_flmod	V	H'0FFFE1AC	H'00000003	4
_sml_buf	V	H'0FFFE198	H'00000014	5
__iob	V	H'0FFFE008	H'00000190	270
$DTBL	V	H'00005ED8	H'0000000C	3

Figure 3.29 Profile Window (Show Functions/Variables)

- Show Functions

  Function: Displays the information on functions in the window.

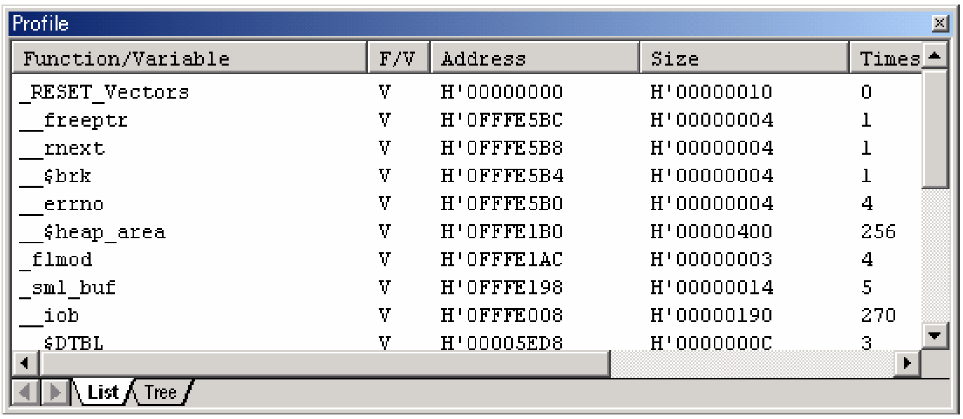


Function/Variable	F/V	Address	Size	Times
_PowerON_Reset_PC	F	H'00000800	H'0000002E	1
_memset	F	H'00005AE0	H'0000006C	0
__rsft	F	H'00005AA8	H'00000036	0
__pow10	F	H'00005A1C	H'0000008C	0
__mult	F	H'0000592C	H'000000F0	0
__add	F	H'000058F0	H'0000003C	0
__$_morecor	F	H'0000588C	H'00000064	0
_malloc	F	H'000057DC	H'00000000	0
__setsbit	F	H'00005744	H'00000098	0
rnd	F	H'00005660	H'000000E4	0

Figure 3.30 Profile Window (Show Functions)

- Show Variables

  Function: Displays the information on varialbles in the window.



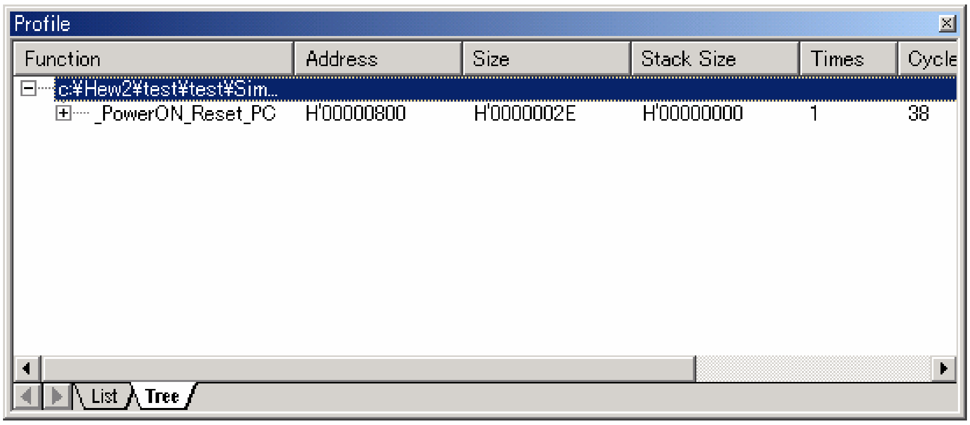
Function/Variable	F/V	Address	Size	Times
_RESET_Vectors	V	H'00000000	H'00000010	0
__freeptr	V	H'0FFFE5BC	H'00000004	1
__rnext	V	H'0FFFE5B8	H'00000004	1
__$brk	V	H'0FFFE5B4	H'00000004	1
__errno	V	H'0FFFE5B0	H'00000004	4
__$heap_area	V	H'0FFFE1B0	H'00000400	256
_flmod	V	H'0FFFE1AC	H'00000003	4
_sml_buf	V	H'0FFFE198	H'00000014	5
__iob	V	H'0FFFE008	H'00000190	270
$DTBL	V	H'00005ED8	H'0000000C	3

Figure 3.31 Profile Window (Show Variables)

- Only executed function(s) checkbox

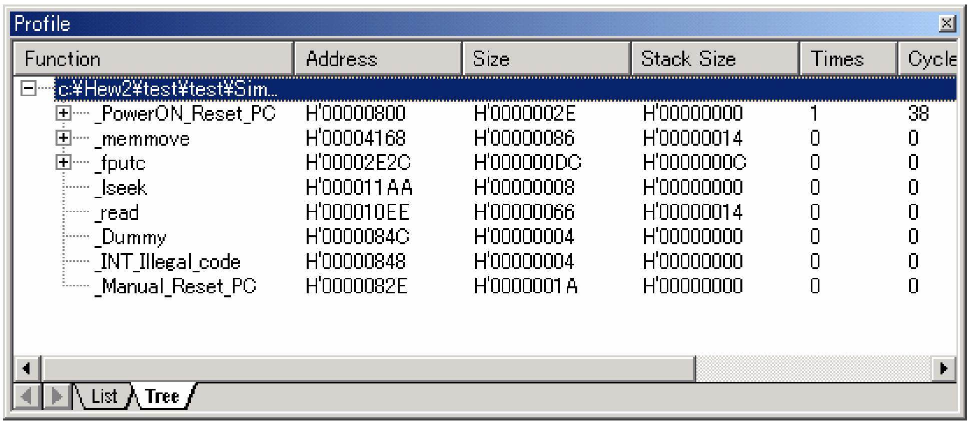
  Function: Specifies whether to display only the functions executed (or the variables accessed) during a Profile data measurement or to display the unexecuted functions (or unaccessed) as well.
  Use: To make the display simple by displaying only the functions executed during a data measurement and not displaying the other functions.
  If all the functions are displayed, It is possible to determine the rate of the functions executed during a data measurement in all the functions, and it is also possible to determine how satisfactorily the application program was executed during a data measument.

  Note: If the stack information file is not read, only the executed functions (or accessed variables) are displayed regardless of this specification.



Function	Address	Size	Stack Size	Times	Cycle
c:¥Hew2¥test¥test¥Sim...					
_PowerON_Reset_PC	H'00000800	H'0000002E	H'00000000	1	38

Figure 3.32 Display Example When the Only Executed Function(s) Checkbox Is On



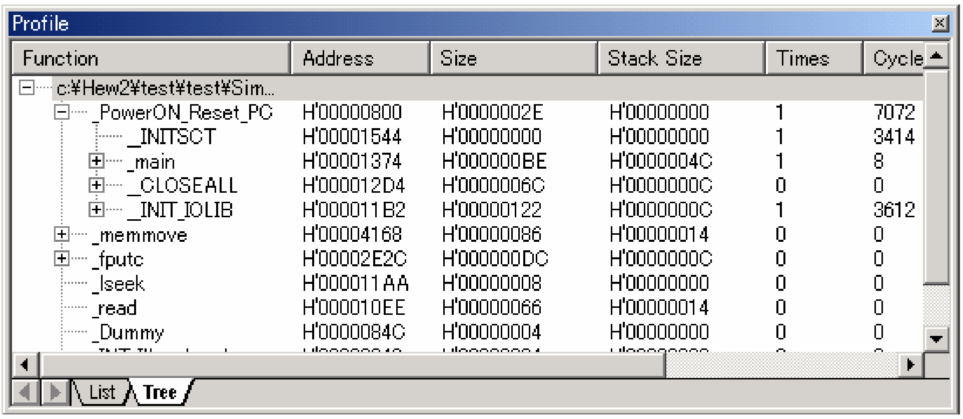
Function	Address	Size	Stack Size	Times	Cycle
c:¥Hew2¥test¥test¥Sim...					
_PowerON_Reset_PC	H'00000800	H'0000002E	H'00000000	1	38
_memmove	H'00004168	H'00000086	H'00000014	0	0
_fputc	H'00002E2C	H'000000DC	H'0000000C	0	0
_lseek	H'000011AA	H'00000008	H'00000000	0	0
_read	H'000010EE	H'00000066	H'00000014	0	0
_Dummy	H'0000084C	H'00000004	H'00000000	0	0
_INT_Illegal_code	H'00000848	H'00000004	H'00000000	0	0
_Manual_Reset_PC	H'0000082E	H'0000001A	H'00000000	0	0

Figure 3.33 Display Example When the Only Executed Function(s) Checkbox Is Off

- Include data of child function(s) checkbox

  Function: Specifies whether to include the data on the child functions in the measurement data to be displayed.
  Use: For example, when you are measuring the Cycle with the SH1 Simulator, checking this checkbox will display the number of cycles from the call to that function to return from it (the number of cycles of the child function called by that function is also added). This is useful when you determine the rate of execution time for each module.

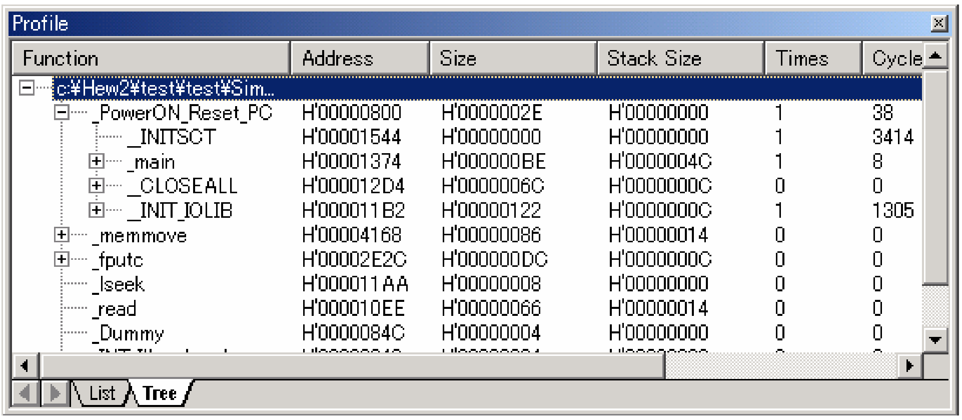
  Note: The value displayed in each column of the Address, Size, Stack Size, and Times does not change.



Profile

Function	Address	Size	Stack Size	Times	Cycle
c:¥Hew2¥test¥test¥Sim...					
_PowerON_Reset_PC	H'00000800	H'0000002E	H'00000000	1	7072
_INITSCT	H'00001544	H'00000000	H'00000000	1	3414
_main	H'00001374	H'000000BE	H'0000004C	1	8
_CLOSEALL	H'000012D4	H'0000006C	H'0000000C	0	0
_INIT_IOLIB	H'000011B2	H'00000122	H'0000000C	1	3612
_memmove	H'00004168	H'00000086	H'00000014	0	0
_fputc	H'00002E2C	H'000000DC	H'0000000C	0	0
_lseek	H'000011AA	H'00000008	H'00000000	0	0
_read	H'000010EE	H'00000066	H'00000014	0	0
_Dummy	H'0000084C	H'00000004	H'00000000	0	0

Figure 3.34 Display Example When the Include Data Of Child Function(s) Checkbox Is On



Profile

Function	Address	Size	Stack Size	Times	Cycle
c:¥Hew2¥test¥test¥Sim...					
_PowerON_Reset_PC	H'00000800	H'0000002E	H'00000000	1	38
_INITSCT	H'00001544	H'00000000	H'00000000	1	3414
_main	H'00001374	H'000000BE	H'0000004C	1	8
_CLOSEALL	H'000012D4	H'0000006C	H'0000000C	0	0
_INIT_IOLIB	H'000011B2	H'00000122	H'0000000C	1	1305
_memmove	H'00004168	H'00000086	H'00000014	0	0
_fputc	H'00002E2C	H'000000DC	H'0000000C	0	0
_lseek	H'000011AA	H'00000008	H'00000000	0	0
_read	H'000010EE	H'00000066	H'00000014	0	0
_Dummy	H'0000084C	H'00000004	H'00000000	0	0

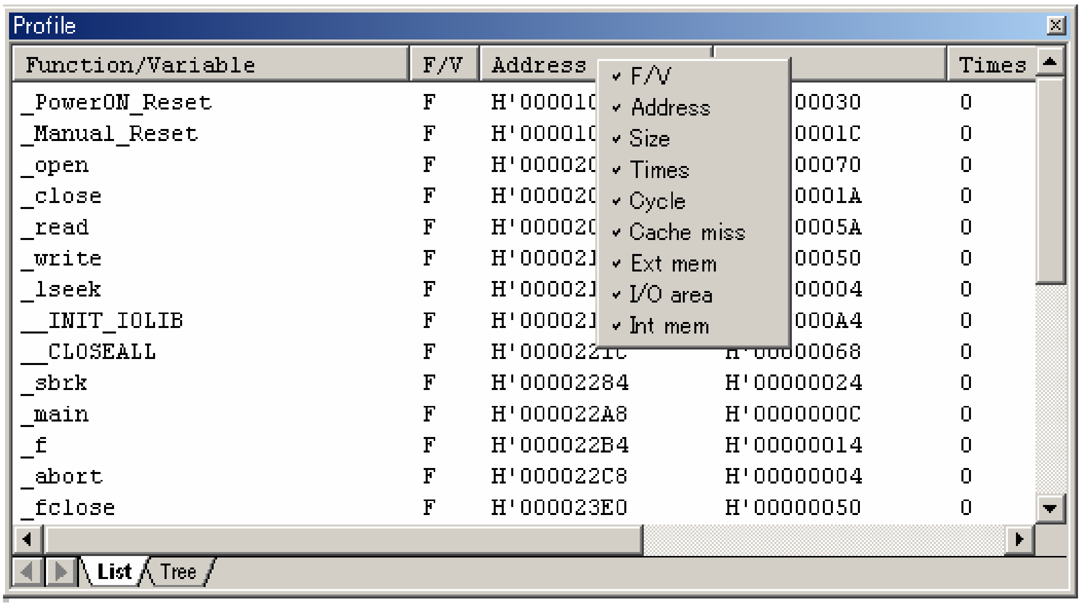
Figure 3.35 Display Example When the Include Data Of Child Function(s) Checkbox Is Off

(iv) Column setting

Right-click on the displayed column in the Profile window to display the pop-up menu.

Function: Selects the information to be displayed in the window.

Use: To display only the required information for simpler window display.



Profile

Function/Variable	F/V	Address		Times
_PowerON_Reset	F	H'00001C	00030	0
_Manual_Reset	F	H'00001C	0001C	0
_open	F	H'00002C	00070	0
_close	F	H'00002C	0001A	0
_read	F	H'00002C	0005A	0
_write	F	H'000021	00050	0
_lseek	F	H'000021	00004	0
__INIT_IOLIB	F	H'000021	000A4	0
__CLOSEALL	F	H'0000221C	H'00000068	0
_sbrk	F	H'00002284	H'00000024	0
_main	F	H'000022A8	H'0000000C	0
_f	F	H'000022B4	H'00000014	0
_abort	F	H'000022C8	H'00000004	0
_fclose	F	H'000023E0	H'00000050	0

Figure 3.36 Profile Window Pop-up Window

3.16 Changing the Alignment Number for the Structure

Description:

Use the pack option (-pack={1 | 4}) or the #pragma pack extension (pack 1 | pack 4 | unpack) to change the alignment number for the structure.
If you specify both the option and the extension, the specification of the extension has priority.
The alignment number for the structure, unit, and class will be the same as the maximum alignment number for the members.
The default is pack=4.
The following shows the specifications and the alignment number.

Table 3.43 Alignment number for the structure, union, and class when the pack option is specified

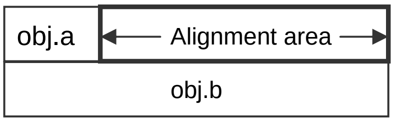
Specification	pack=1	pack=4	No specification
[unsigned]char	1	1	1
[unsigned]short, _ _fixed	1	2	2
[unsigned]int, [unsigned]long, [unsigned]long long, long_ _fixed, _ _accum, long_ _accum, floating-point, pointer	1	4	4
Structure, union, and class for which the alignment number is 1	1	1	1
Structure, union, and class for which the alignment number is 2	1	2	2
Structure, union, and class for which the alignment number is 4	1	4	4

Allocating structure data

(1) When you allocate structure members, a blank area may be inserted between members because each member is aligned by the alignment number for the data type of that member.

Example:

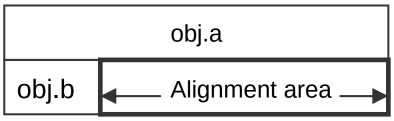
```
struct {
char  a;
int   b;
} obj;
```



(2) If the alignment number of a structure is 4 bytes and the last member ends at the first, second, or third byte, the next bytes are also handled as a structure-type area.

Example:

```
struct {
int  a;
char  b;
} obj;
```

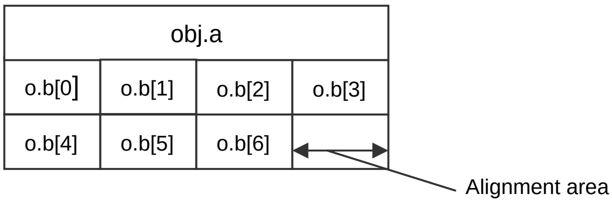


Allocating unit data

(1) If the alignment number of a unit is 4 bytes and the maximum size of the member is not a multiple of 4 (bytes), the area including the remaining bytes up to a multiple of 4 are handled as the unit-type area, until the size reaches a multiple of 4.

Example:

```
union {
int     a;
char    b[7];
} o;
```



Changing the alignment number

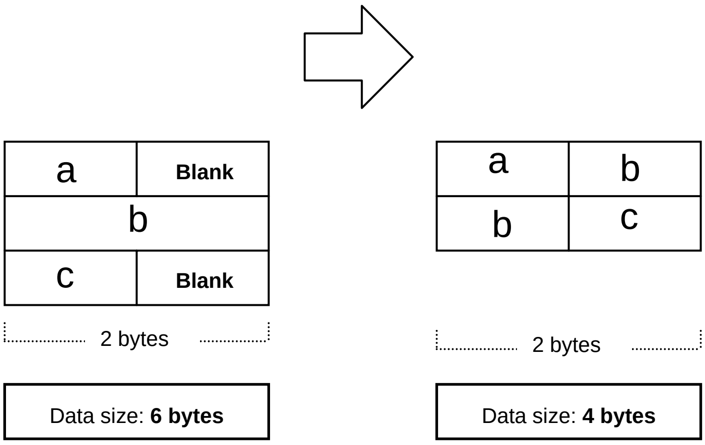
When `#pragma pack 1` is specified, blanks for alignment may not be inserted because data other than one-byte data can also be allocated to an odd address. This may reduce the data size.

C/C++ program,

```
struct S1{
    char a;
    short b;
    char c;
}
```

```
#pragma pack 1
struct S1{
    char  a;
    short b;
    char  c;
}
```

Data image



Note: If the alignment number is 1, each member is accessed in byte units. The members cannot be accessed by using a pointer.
This specification reduces the data size, which is efficient for data block transfer. However, if you change the alignment number to 1, members of a word or long word structure will be accessed byte by byte. This increases the code size.

C/C++ program

```
struct S {
   char x;
   int y;
} s;
int *p=&s.y;     ←s.y may be an odd address.
void test()
{
   s.y=1;        ←Can be accessed

   *p =7;        ←Cannot be accessed
}
```

3.17 long long type

Description:

The long long and unsigned long long data types are supported.
A signed integer is described as long long, and an unsigned integer is described as unsigned long long.
To create an integer constant of the long long type, add the suffix LL to the integer.To create an integer constant of the unsigned long long type, add the suffix ULL to the integer.

Table 3.44 Integer types and the range of values

Type	Range of values	Data size
char	-128 to 127	1 byte
signed char	-128 to 127	1 byte
unsigned char	0 to 255	1 byte
short	-32768 to 32767	2 bytes
unsigned short	0 to 65535	2 bytes
int	-2147483648 to 2147483647	4 bytes
unsigned int	0 to 4294967295	4 bytes
long	-2147483648 to 2147483647	4 bytes
unsigned long	0 to 4294967295	4 bytes
long long	-9223372036854775808 to 9223372036754775807	8 bytes
unsigned long long	0 to 18446744073709551615	8 bytes

3.18 DSP-C Specifications

Description:

The DSP-C language is supported.
This specification is valid when the compiler option “dspc” is specified for the SuperH RISC engine C/C++ compiler.

3.18.1 Fixed-Point Data Type

Previously, the integer type has been used to represent a fractional value. You can now use the fixed-point data type to code a fractional value without modification.
The SuperH RISC engine C/C++ compiler generates DSP instructions appropriate to the fixed-point data type being used.
Table 3.45 shows the internal representation of the fixed-point data type.

Table 3.45 Internal Representation of the Fixed-point Data Type

Type	Size (Size on memory)	Align-ment number (bytes)	Range of data		Constant index
			Min. value	Max. value	
_ _fixed	16 bits (16 bits)	2	-1.0	$1.0-2^{-15}$ (0.999969482421875)	r
long _ _fixed	32 bits (32 bits)	4	-1.0	$1.0-2^{-31}$ (0.9999999995343387126922607421875)	R
_ _accum	24 bits (32 bits)	4	-256.0	$256.0-2^{-15}$ (255.999969482421875)	a
long _ _accum	40 bits (64 bits)	4	-256.0	$256.0-2^{-31}$ (255.9999999995343387126922607421875)	A

Important Information:

(1) The _ _accum and long _ _accum data stored in memory is right justified, with sign extension added at the beginning part.

Example: `(_ _accum)128.5a is stored as ″00 40 40 00″.`
Example: `(long _ _accum)(-256.0A) is stored as ″FF FF FF 80 00 00 00 00″.`

(2) Comparing DSP-C and the previous method

C function [Previous method]

```
// -cpu=sh3

#include <stdio.h>

#define NUM 8

short input[NUM] = {0x1000, 0x2000, 0x4000,
                    0x6000,

                  0xf000, 0xe000, 0xc000,
                  0xa000};

short result[NUM];

void func(void)

{

    int i;

    for (i = 0; i < NUM; i++) {

   result[i] = input[i] + 0x1000;

    }

}

void main(void)

{

    int i;

    func();

    for (i = 0; i < NUM; i++) {

   printf("%f\n", result[i]/32768.0);

    }

}
```

[DSP-C]

```
// -cpu=sh3dsp -dspc

#include <stdio.h>

#define NUM 8

__fixed input[8] = { 0.125r, 0.25r, 0.5r, 0.75r,

            -0.125r,  -0.25r, -0.5r, -0.75r};

__fixed result[NUM];

void func()

{

    int i;

    for (i = 0; i < NUM; i++) {

   result[i] = input[i] + 0.125r;

    }

}

void main(void)

{

    int i;

    func();

    for (i = 0; i < NUM; i++) {

   printf("%r\n", result[i]);

    }

}
```

(3) Example of multiply-and-accumulation operations

If the integer type is used as a substitute for a fractional value, the products must be aligned to the fixed number of digits. This alignment is unnecessary for the fixed-point data type.

C function [Previous method]

```
// -cpu=sh3

#include <stdio.h>

#define NUM 8

short x_input[NUM] = {0x1000, 0x2000, 0x4000,
0x6000, 0xf000, 0xe000, 0xc000, 0xa000};

short y_input[NUM] = {0x1000, 0x2000, 0x4000,
0x6000, 0xf000, 0xe000, 0xc000, 0xa000};

int result;

int func(short *x_input, short *y_input)

{

    int i;

    int temp = 0;

    for (i = 0; i< NUM ;i++) {

      temp += (x_input[i] * y_input[i]) >> 15;

    }

    return (temp);

}

void main()

{

    result = func(x_input, y_input);

    printf("%f\n", result/32768.0);

}
```

[DSP-C]

```
// -cpu=sh3dsp -dspc -fixed_noround

#include <stdio.h>

#define NUM 8

__X __fixed x_input[NUM] = { 0.125r,   0.25r,
  0.5r, 0.75r, -0.125r, -0.25r, -0.5r, -0.75r};

__Y __fixed y_input[NUM] = { 0.125r,   0.25r,
0.5r, 0.75r, -0.125r, -0.25r, -0.5r, -0.75r};

__accum result;

void func(__accum *result_p,

         __X __fixed *x_input,

        __Y __fixed *y_input)

{

    int i;

    __accum temp = 0.0a;

    for (i = 0; i< NUM ;i++) {

      temp += x_input[i] * y_input[i];

    }

    *result_p = temp;

}

void main()

{

    func(&result, x_input, y_input);

    printf("%a\n", result);

}
```

3.18.2 Memory Qualifier

Adding the X/Y memory qualifier to variables promotes generation of X/Y memory-dedicated access instructions which are more efficient than ordinary memory access instructions.
Use the following qualifier to explicitly specify the X or Y memory for storing data.

_ _X: Store data in the X memory.

_ _Y: Store data in the Y memory.

The SuperH RISC engine C/C++ compiler outputs objects that have the _ _X or _ _Y memory qualifier to the sections shown in table 3.46. You must allocate these sections to the X or Y memory during linking.

Table 3.46 Memory Qualifier Specifications

Name	Section	Description
Constant area	$XC	const data (Stored in the X memory)
	$YC	const data (Stored in the X memory)
Initialized data area	$XD	Data with an initial value (Stored in the X memory)
	$YD	Data with an initial value (Stored in the Y memory)
Uninitialized data area	$XB	Data without an initial value (Stored in the X memory)
	$YB	Data without an initial value (Stored in the Y memory)

However, X or Y memory may exist only on RAM. You must be careful when creating ROM from such memory.

Examples of use:

(1) Storing data in memory by using the _ _X or _ _Y memory qualifier

```
_ _X  int            a;     //Store in the X memory.
      int  _ _X      b;     //Store in the X memory.
_ _Y  int          * c;     //Pointer to the int data in the Y memory (Memory is undefined.)
      int   _ _Y   * d;     //Pointer to the int data in the Y memory (Memory is undefined.)
      int   *_ _Y    e;     //Pointer to the int data (Stored in the Y memory)
_ _X  int   *_ _Y    f;     //Pointer to the int data in the X memory (Stored in the Y memory)
```

(2) Copying the constant area and initialized data area from ROM to X/Y RAM

In this example, the data that was stored in ROM during linking is copied to X/Y RAM when the program starts. You need to use the VOW option of the optimizing linkage editor to allocate the same space twice in ROM and in X/Y RAM.

Example of the subcommand during linking:

```
rom=$XC=XC,$XD=XD,$YC=YC,$YD=YD start
P,C,D,$XC,$XD,$YC,$YD/400,$XB,XC,XD/05007000,$YB,YC,YD/05017000
```

The standard library function INITSCT() allows you to easily copy data from ROM to X/Y RAM.

Example of use: _INITSCT()

```
#include <_h_c_lib.h>

void PowerON_Reset(void)

{

      _INITSCT();

    main();

    sleep();

}


#pragma section $DSEC

static const struct {

   void *rom_s;

   void *rom_e;

   void *ram_s;

} DTBL[] = { {__sectop("$XC"), __secend("$XC"), __sectop("XC")},

           {__sectop("$XD"), __secend("$XD"), __sectop("XD")},

           {__sectop("$YC"), __secend("$YC"), __sectop("YC")},

           {__sectop("$YD"), __secend("$YD"), __sectop("YD")}};

#pragma section
```

(3) Not using the constant area or initialized area

By specifying that neither a const specification nor initialized data is to be added to an object with the X/Y memory qualifier, you do not have to allocate the same space twice in ROM and in X/Y RAM.

For example, you can eliminate initialized data by specifying dynamic initialization as shown in the following example.

Example of use

```
#define NUM 8

__X __fixed x_input[NUM];

__Y __fixed y_input[NUM];

__fixed x_input[NUM] = {  0.125r,   0.25r,  0.5r,  0.75r,  -0.125r,  -0.25r,  -0.5r,  -0.75r};

__fixed y_input[NUM] = {  0.125r,   0.25r,  0.5r,  0.75r,  -0.125r,  -0.25r,  -0.5r,  -0.75r};

void xy_init()

{

       int i;


       for (i = 0; i< NUM; i++) {

           x_input[i] = x_init[i];

           y_input[i] = y_init[i];

       }

}


void main()

{

        xy_init();

           :

           :

}
```

(4) Comparing DSP-C and the previous method

C function [Previous method]

```
// -cpu=sh3

#include <stdio.h>

#define NUM 8

short x_input[NUM] = {0x1000, 0x2000, 0x4000,
                  0x6000,

    0xf000, 0xe000, 0xc000, 0xa000};

short y_input[NUM] = {0x2000, 0x4000, 0xe000,
                  0xf000,

    0x6000, 0x2000, 0xe000, 0xf000};

short result[NUM];

void func(void)

{

    int i;

    for (i = 0; i < NUM; i++) {

      result[i] = x_input[i] - y_input[i];

    }

}


void main(void)

{

    int i;

    func();

    for (i = 0; i < NUM; i++) {

      printf("%f\n", result[i]/32768.0);

    }

}
```

[DSP-C]

```
// -cpu=sh3dsp -dspc

#include <stdio.h>

#define NUM 8

__X __fixed x_input[NUM] = { 0.125r,  0.25r,
              0.5r,  0.75r,

           -0.125r, -0.25r, -0.5r, -0.75r};

__Y __fixed y_input[NUM] = {0.25r, 0.5r, -0.25r,
            -0.125r,

     0.75r, 0.25r, -0.25r, -0.125r};

__fixed result[NUM];

void func(void)

{

    int i;

    for (i = 0; i < NUM; i++) {

      result[i] = x_input[i] - y_input[i];

    }

}


void main(void)

{

    int i;

    func();

    for (i = 0; i < NUM; i++) {

      printf("%r\n", result[i]);

    }

}
```

3.18.3 Saturation Qualifier

If the operation results in an overflow, saturation operation replaces the result with the largest or smallest representable value. For DSP-C, simply adding a saturation qualifier enables the saturation operation.
Use the following qualifier to specify the saturation operation:

```
_ _sat
```

You can specify the saturation qualifier only for _ _fixed or long _ _fixed data. Specifying the saturation qualifier for any other data type causes an error.
Saturation operation will be performed if an expression contains data piece for which at least one saturation qualifier (_ _sat) is specified.

Examples of use:

(1) Example of sat specification

```
_ _fixed         a;
_ _sat  _ _fixed b;
_ _fixed         c;

a = -0.75r ;
b = -0.75r ;
c = a + b  ; // c = -1.0r will result.
```

(2) Comparing DSP-C and the previous method

C function [Previous method]

```
// -cpu=sh3
#include <stdio.h>
#define NUM 8
short x_input[NUM] = {0x1000, 0x2000, 0x4000,
   0x6000, 0xf000, 0xe000, 0xc000, 0xa000};
short y_input[NUM] = {0x1000, 0x2000, 0x4000,
   0x6000, 0xf000, 0xe000, 0xc000, 0xa000};
short result[NUM];
void func(void)
{
    int i;
    int temp;
    for (i = 0; i < NUM; i++) {
      temp = x_input[i] + y_input[i];
      if (temp > 32767) {
          temp = 32767;
      }
      else if (temp < -32768) {
         temp = -32768;
   }
   result[i] = temp;
    }
}
void main(void)
{
    int i;
    func();
     :
```

[DSP-C]

```
// -cpu=sh3dsp -dspc
#include <stdio.h>
#define NUM 8
__sat __X __fixed x_input[NUM] = { 0.125r,
0.25r,  0.5r,  0.75r,
  -0.125r, -0.25r, -0.5r, -0.75r};
__sat __Y __fixed y_input[NUM] = { 0.125r,
0.25r, 0.5r, 0.75r,                  -0.125r,
-0.25r, -0.5r, -0.75r};
__fixed result[NUM];


void func(void)
{
    int i;
    for (i = 0; i < NUM; i++) {
      result[i] = x_input[i] + y_input[i];
    }
}
void main(void)
{
    int i;
    func();
    for (i = 0; i < NUM; i++) {
      printf("%r\n", result[i]);
    }
}
```

3.18.4 Circular Qualifier

Use the following qualifier to specify the modulo addressing:

```
_ _circ
```

You can specify the modulo addressing for _ _fixed type one-dimensional arrays and pointers for which the memory qualifier (_ _X/_ _Y) is specified. Specifying the modulo addressing for any other conditionscauses an error.

Examples of use:

(1) Comparing DSP-C and the previous method

C function [Previous method]

```
// -cpu=sh3

#include <stdio.h>

#define NUM 8

#define BUFFER_SIZE 4

short x_input[NUM] = {0x1000, 0x2000, 0x4000,
0x6000,, 0xf000, 0xe000, 0xc000, 0xa000};

short y_input[BUFFER_SIZE] = {0x2000, 0x4000,
0x2000, 0x1000};

short result[NUM];


void func()

{

    int i;

    for (i = 0; i < NUM; i++) {

      result[i] = x_input[i] +
      y_input[i%(BUFFER_SIZE)];

    }

}


void main()

{

    int i;

    func();

    for (i = 0; i < NUM; i++) {

     printf("%f\n", result[i]/32768.0);

    }

}
```

[DSP-C]

```
// -cpu=sh3dsp -dspc

#include <stdio.h>

#include <machine.h>

#define NUM 8

#define BUFFER_SIZE 4

__X __fixed x_input[NUM] = { 0.125r,  0.25r,
   0.5r,  0.75r, -0.125r, -0.25r, -0.5r,
   -0.75r};

__circ __Y __fixed y_input[BUFFER_SIZE] =
   {0.25r, 0.5r, 0.25r, 0.125r};

__fixed result[NUM];


void func()

{

    int i;

    set_circ_y(y_input, sizeof(y_input));

    for (i = 0; i < NUM; i++) {

     result[i] = x_input[i] + y_input[i];

    }

    clr_circ();

}


void main()

{

    int i;

    func();

    for (i = 0; i < NUM; i++) {

     printf("%r\n", result[i]);

    }

}
```

Important Information:

(1) The modulo addressing is applicable to one-dimensional arrays and pointers that exist between the built-in functions clr_circ() and set_circ_x() or set_circ_y().

(2) Correct operation is not guaranteed if you specify the modulo addressing for multiple arrays concurrently or if you reference an array or pointer with _ _circ specified in other than between the built-in functions shown above.

(3) Correct operation is not guaranteed if you specify the modulo addressing in a negative direction.

(4) Data subject to modulo addressing must be aligned so that the higher 16 bits will be the same during liking. You cannot directly reference the contents of an array.

(5) Correct operation is not guaranteed if one of the following occurs (a warning may be output):

- optimize=0 is specified.
- The _ _circ pointer is specified for other than a local variable.
- volatile is specified for the _ _circ pointer.
- The _ _circ pointer is updated but is not referenced.
- There is a function all between the built-in functions clr_circ and set_circ_x or set_circ_y.

3.18.5 Type Conversion

Table 3.47 shows the rules for type conversion.

Table 3.47 Rules for Type Conversion

Conversion	Specifications
_ _fixed -> long _ _fixed	Lower 16 bits are cleared to zero. The value remains unchanged.
_ _accum -> long _ _accum	
long _ _fixed -> _ _fixed	Lower 16 bits are truncated. Precision of the fractional part is degraded.
long _ _accum -> _ _accum	
_ _fixed -> _ _accum	Sign expansion is performed for higher 8 bits. The value remains unchanged.
long _ _fixed -> long _ _accum	
_ _fixed -> long _ _accum	Sign expansion is performed for higher 8 bits. Lower 16 bits are cleared to zero. The value remains unchanged.
long _ _fixed -> _ _accum	Sign expansion is performed for higher 8 bits. Lower 16 bits are truncated. Precision of the fractional part is degraded.
_ _accum -> _ _fixed	Higher 8 bits are truncated. The 9th bit must be the sign bit. The value remains unchanged if the integer part is zero.
long _ _accum -> long _ _fixed	
_ _accum -> long _ _fixed	
long _ _accum -> _ _fixed	Higher 8 bits and lower 16 bits are truncated. The 9th bit must be the sign bit. The value remains unchanged if the integer part is zero. Precision of the fractional part is degraded.
_ _fixed -> signed integer type	The value is -1 for -1.0r and -1.0R, or 0 for other cases.
long _ _fixed -> signed integer type	
_ _accum -> signed integer type	The fractional part is truncated. The value after conversion is an integer from -256 to 255.
long _ _accum -> signed integer type	
_ _fixed -> unsigned integer type	For -1.0r and -1.0R, the maximum value for the type after conversion is assumed. For other cases, 0 is assumed.
long _ _fixed -> unsigned integer type	
_ _accum -> unsigned integer type	The fractional part is truncated. For a positive value, the value after conversion is an integer from 0 to 255. For a negative value, (the value before conversion + 1 + the maximum value for the type after conversion) is assumed.
long _ _accum -> unsigned integer type	
signed integer type -> _ _fixed	The highest bit before conversion must be the highest bit after conversion. All the other bits will be zero.
signed integer type ->long _ _fixed	
signed integer type -> _ _accum	Lower 9 bits of the value must be the integer part. The fractional part must be zero.
signed integer type ->long _ _accum	
unsigned integer type -> _ _fixed	All the bits after conversion must be zero.
unsigned integer type ->long _ _fixed	

Conversion	Specifications
unsigned integer type -> _ _accum	Lower 9 bits of the value must be the integer part. The fractional part must be zero.
unsigned integer type ->long _ _accum	
Fixed-point -> floating-point	A value representable in the type after conversion will be the same as the original value. The value that cannot be represented is rounded to a nearest value.
Floating-point -> fixed point	The handling of the fractional part is the same as for the conversion from fixed-point to floating point. The handling of the integer part is the same as for the conversion from floating-point to integer. If the integer part is the representable range for the fixed-point, the value remains unchanged. If the integer part exceeds the range, the lowest bit of the overflow must be a sign bit. The saturation processing is not performed even if it is specified for the type after conversion.

Important Information:

(1) Conversion from (long)_ _fixed to the integer type, and vice versa
Integers that can be represented in the (long)_ _fixed type are 0 and -1.
This means that the above conversion causes missing information.

(2) Conversion from (long)_ _accum to the integer type, and vice versa
Integers in the range from -256 to 255 can be represented in the (long)_ _accum type. Integers within this range retain information after they are converted.
However, note that converting a negative value to the unsigned integer type causes an overflow.
For a series of operations that only require the integer type, conversion to the integer type may improve performance.

(3) Bit pattern copy
If you use a substitute operator to copy a bit pattern, a type conversion occurs and the expected results cannot be acquired. In this case, use the built-in functions such as long_as_lfixed and lfixed_as_long.

3.18.6 Arithmetic Conversion

If an operation contains two different types of operands, the type shown in the upper column in figure 3.37 will be used. An error occurs if you specify an operation between types which are not related in figure 3.37 (for example, between the integer and floating-point, or between _ _accum and long_ _fixed). In this case, you must use a cast to perform explicit type conversion. However, as long as the result values are guaranteed, some operation may not follow the above conversion rules for sake of efficiency.

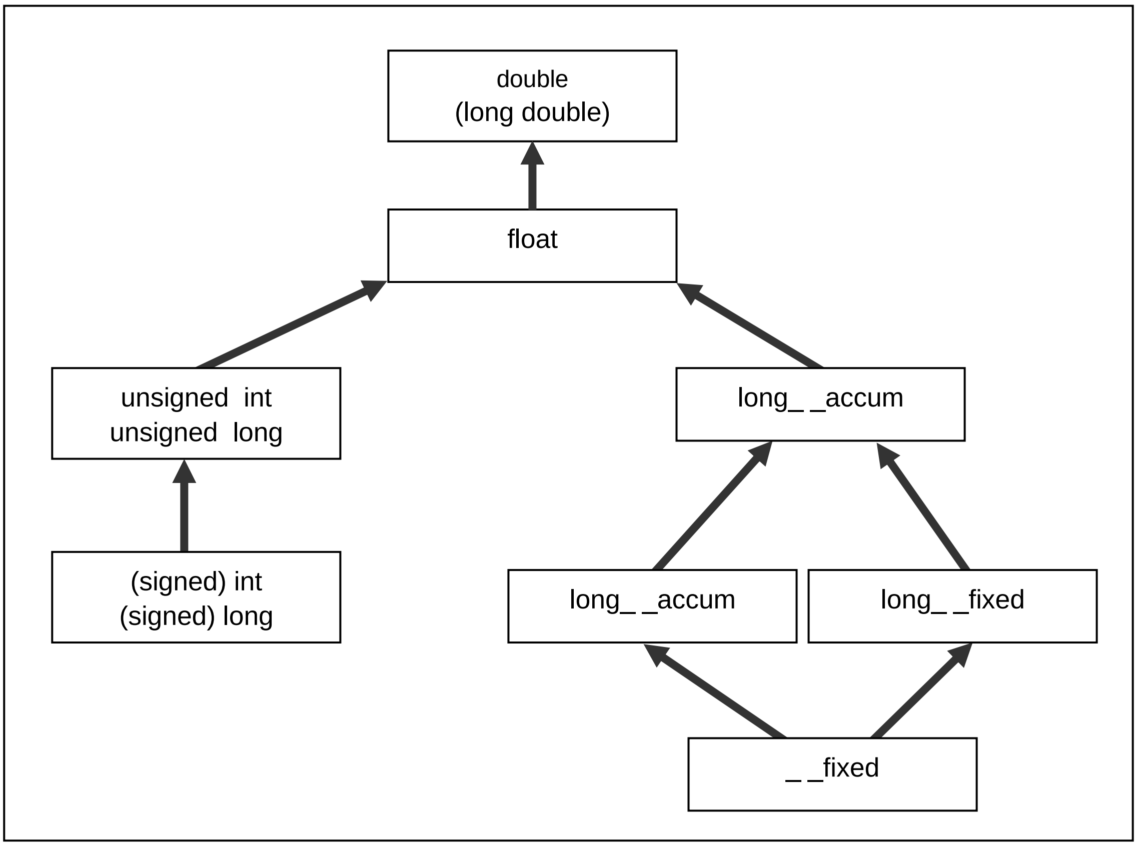


Figure 3.37 Rules for Arithmetic Conversion

3.19 MAP Optimization Extended Option

Description:

This option performs MAP optimization, without using information about symbol allocated addresses that were assigned by linking. As such, recompiling is unnecessary.
However, since optimization is only applicable to static variables defined within files, extern variables cannot be optimized.

3.19.1 Usage

Specify the "-smap" option at compile-time.

3.19.2 Example of Improved External Variable Access Code (1)

Taking into account the order of variable allocation within the same section, access the consecutively allocated variables relatively, for the same register.

aiueo

```
int a,b;
extern int c;
f(){
    a = 0;
    b = 0;
    c = 0;
}
```

```
        MOV       #0,R2
        MOV.L     L11,R6      ; _a
        MOV.L     R2,@R6
        MOV.L     L11+4,R6    ; _b
        MOV.L     R2,@R6
        MOV.L     L11+8,R6    ; _c
        RTS
        MOV.L     R2,@R6
L11:
        .DATA.L   _a
        .DATA.L   _b
        .DATA.L   _c
```

```
        MOV       #0,R2
        MOV.L     L11+2,R6    ; _a
        MOV.L     R2,@R6
        <<< delete >>>
        MOV.L     R2,@(4,R6)
        MOV.L  L11+6,R6    ; _c
        RTS
        MOV.L     R2,@R6
L11:
        .DATA.L   _a
        <<< delete >>>
        .DATA.L   _c
```

Access using a relative address from "a"

The extern variable is ignored.

3.19.3 Example of Improved External Variable Access Code (2)

When the "gbr=auto" option (default) is specified, GBR is used as the base for external variable access.

Source Program

```
int a[100];
f(){
    a[0]=0;
    a[50]=0;
    a[51]=0;
    a[52]=0;
}
```

"smap" not specified

```
       MOV.L      L11+2,R5   ; _a
       MOV        #-56,R0
       MOV        #0,R4
       EXTU.B     R0,R0
       MOV.L      R4,@R5
       MOV.L      R4,@(R0,R5)
       ADD        #4,R0
       MOV.L      R4,@(R0,R5)
       ADD        #4,R0
       RTS
       MOV.L      R4,@(R0,R5)
L11:
       .RES.W     1
       .DATA.L    _a
```

"smap" specified

```
      STC       GBR,@-R15
      MOV.L     L11,R0       ; _a
      LDC       R0,GBR
      MOV       #0,R0
      MOV.L     R0,@(0,GBR)
      MOV.L     R0,@(200,GBR)
      MOV.L     R0,@(204,GBR)
      MOV.L     R0,@(208,GBR)
      RTS
      LDC       @R15+,GBR
L11:
      .DATA.L     _a
```

Reference relatively with GBR

3.20 TBR-Relative Function Call

Description:

For SH-2A and SH2A-FPU, the jump table base register (TBR) is used for calling functions through the use of table-reference subroutine call instructions.
A relative value from the TBR is an offset in the jump table whose base address is contained by the TBR. This value equals to the distance between the beginning of the $TBR section and the address data label of the function to be called.
You can use the "-tbr" option to specify that TBR-relative calling is to be used for all functions. You can also use the preprocessor directive "#pragma tbr" to specify TBR-relative calling for an individual function.
To perform a TBR-relative function call, you must set the start address of the $TBR section to the TBR.
To use a TBR-relative call to call standard library functions, perform the following:

(1) In the "tbr.h" system include file, add "#pragma tbr", followed by the library for which TBR-relative calling is to be used.
(2) Use the standard library creation tool "lbgsh" to create a library.
(3) In the source program that calls the library for which you want to perform TBR-relative calls, use an #include directive to include "tbr.h".

- Format:

```
<Options>

-tbr[=<section name>]


<Preprocessor directive>

#pragma tbr (<function name>[(sn=<section name>|ov=<offset value>)][,...])

<offset value> must be a multiple of 4, from 0 to 1020.
```

Example of use:

Example 1

If the TBR-relative function call is specified, the compiler uses TBR-relative calling for all functions, and generates a jump table for the functions defined in the file. This jump table consists of the function address data and their labels.
The label name of a function address in the jump table is the function name, preceded by "$_". For a static function, the label name is the function name, preceded by "$__$".

C language code

```
/* -cpu=sh2a -size -tbr */
#include <machine.h>
void f1(){}
void f2(){}
static void f3(){}


main()

{

    set_tbr(__sectop("$TBR")); /* Sets the beginning of the $TBR section to the TBR */

    f1();

    f2();
    f3();
}
```

Expanded into assembly language code

```
_main:
          STS.L       PR,@-R15
          MOV.L       L14+2,R2    ; STARTOF $TBR
          LDC         R2,TBR
          JSR/N       @@($_f1-(STARTOF $TBR),TBR)       ; TBR-relative function call
          JSR/N       @@($_f2-(STARTOF $TBR),TBR)       ; TBR-relative function call
          JSR/N       @@($__$f3-(STARTOF $TBR),TBR)     ; TBR-relative function call
          LDS.L       @R15+,PR
          RTS/N
L14:
          .RES.W      1
          .DATA.L     STARTOF $TBR
          .SECTION    $TBR,DATA,ALIGN=4                 ; TBR-relative jump table
$_f1:
          .DATA.L     _f1                               ; Function address data
$_f2:
          .DATA.L     _f2                               ; Function address data
$_main:
          .DATA.L     _main                             ; Function address data
$__$f3:
          .DATA.L     __$f3                             ; Static function address data
```

Example 2

In addition to the "-tbr" option, which specifies that TBR-relative calling is to be used for all functions, you can use the "#pragma tbr" to specify TBR-relative calling for an individual function.
The functions specified in <function name> will be called using TBR-relative calls.
If you specify "sn=<section name>", function address data is generated in the section indicated by the section name preceded by "$TBR".
If you specify "ov=<offset value>", the TBR-relative value will be the indicated offset value.

C language code

```
/* -cpu=sh2a -size */
#pragma tbr (f1(sn=X))
#pragma tbr (f2(ov=0))
f1(){}
```

```
f2(){}

main(){

f1();

f2();

}
```

Expanded into assembly language code

```
    _main:

          STS.L       PR,@-R15

          JSR/N       @@($_f1-(STARTOF $TBRX),TBR)

          JSR/N       @@(0,TBR)                  ; The TBR-relative value is 0

          LDS.L       @R15+,PR

          RTS/N

          .SECTION    $TBRX,DATA,ALIGN=4         ; Section name "$TBRX"

$_f1:

          .DATA.L     _f1

                                                 ; The function address data is not generated

                                                 ; for function (f2) for which "ov=<offset value>"

                                                 ; is specified (see Example 3).
```

Example 3

For the functions for which "ov=<offset value>" is specified, you must create the function address data in the TBR-relative jump table.
If the function definition is not found in the same file, you must set the same specification in the file for the function definitions, or create the function address data in the TBR-relative jump table.

C language code

```
/* -cpu=sh2a */

#pragma tbr (func1(ov=0))          /* Specifies offset 0 in the jump table */

#pragma tbr (func2(ov=4))          /* Specifies offset 4 in the jump table */

#pragma tbr (func3(ov=8))          /* Specifies offset 8 in the jump table */

extern void func1();

extern void func2();

extern void func3();


#pragma tbr (func4(sn=NEW))        /* Specifies "$TBRNEW" for the section in the jump table */

#pragma tbr (func5(sn=NEW))

#pragma tbr (func6(sn=NEW))
```

```
extern void func4();

extern void func5();

extern void func6();


#include<machine.h>

void main()

{

     set_tbr(__sectop("$TBR"));       /* Sets the beginning of the $TBR section to the TBR */

     func1();

     func2();

     func3();

     set_tbr(__sectop("$TBRNEW"));    /* Switches the table to "$TBRNEW" */

     func4();

     func5();

     func6();

}
```

Expanded into assembly language code

```
_main:

           STS.L      PR,@-R15

           MOV.L      L11+2,R1    ; STARTOF $TBR

           LDC        R1,TBR

           JSR/N      @@(0,TBR)

           JSR/N      @@(4,TBR)

           JSR/N      @@(8,TBR)

           MOV.L      L11+6,R4    ; STARTOF $TBRNEW

           LDC        R4,TBR

           JSR/N      @@($_func4-(STARTOF $TBRNEW),TBR)

           JSR/N      @@($_func5-(STARTOF $TBRNEW),TBR)

           JSR/N      @@($_func6-(STARTOF $TBRNEW),TBR)

           LDS.L      @R15+,PR

           RTS/N

L11:

         .RES.W     1

           .DATA.L    STARTOF $TBR

           .DATA.L    STARTOF $TBRNEW
```

To specify a TBR-relative function call, you must set the TBR. Since the compiler, when calling functions, references the data on the jump table to find the function address, this can reduce the size of the data.

The following shows the code for calling functions, without using the TBR.

```
_main:
           STS.L     PR,@-R15
           MOV.L      L11,R1     ; _func1
           JSR/N      @R1
           MOV.L      L11+4,R4   ; _func2
           JSR/N      @R4
           MOV.L      L11+8,R5   ; _func3
           JSR/N      @R5
           MOV.L      L11+12,R6  ; _func4
           JSR/N      @R6
           MOV.L      L11+16,R7  ; _func5
           JSR/N      @R7
           MOV.L      L11+20,R2  ; _func6
           JMP        @R2
           LDS.L      @R15+,PR
L11:
           .DATA.L    _func1
           .DATA.L    _func2
           .DATA.L    _func3
           .DATA.L    _func4
           .DATA.L    _func5
           .DATA.L    _func6
```

Assembly language code (jump table 1)

Create a jump table for function address data in the "$TBR" section according to the "ov=<offset value>" specification in "pragma tbr".

```
           .SECTION   $TBR,DATA,ALIGN=4    ;
           .DATA.L    _func1               ;  The offset in the jump table should be 0.
           .DATA.L    _func2               ;  The offset should be 4
           .DATA.L    _func3               ;  The offset should be 8
```

Assembly language code (jump table 2)

If the function definition is not found in the same file, use the same specification in the file for the function definitions, or create the following jump table:

```
            .EXPORT     $_func4
            .EXPORT     $_func5
            .EXPORT     $_func6
            .SECTION    $TBRNEW,DATA,ALIGN=4
$_func4:                                  ; The label name should be "$_"+<function name>
            .DATA.L     _func4            ; Function address data
$_func5:
            .DATA.L     _func5
$_func6:
            .DATA.L     _func6
```

Example 4

For SH-2A, the CPU calls printf functions relatively with the TBR:

(1) Specify "#pragma tbr printf" in "tbr.h".

```
     :
#if (defined(_SH2A) || defined(_SH2AFPU)) && !defined(_PIC)
     :
#pragma tbr printf     // ← Added
     :
#endif  /* #if (defined(_SH2A) || defined(_SH2AFPU)) && !defined(_PIC) */
     :
```

(2) Create a standard library containing a TBR-relative table.

```
lbgsh -cpu=sh2a
```

(3) Specify "tbr.h" to be included in the program that is using printf.

```
#include <tbr.h>       // ← Added
#include <stdio.h>

main()
{
    printf("tbr\n");
}
```

Important Information:

(1) The compiler does not use the TBR-relative calling if the BSR instruction can be used to call functions. However, the compiler does use TBR-relative calling if the "-size" option is specified.

(2) If you specify any option other than "-cpu=sh2a" or "-cpu=sh2afpu", TBR-relative function calling is disabled.

(3) If you specify the "-pic=1" option, the TBR-relative function calling is disabled, because the absolute addresses of functions cannot be determined.

(4) If "$TBR" is used to indicate a section name for the jump table that is specified for the section name in the "-section" option, malfunction may occur during execution of objects.

(5) You can specify up to 255 functions for each section in the entire program.

(6) You cannot specify "sn=<section name>" and "ov=<offset value> at the same time for the same function.

(7) An error occurs if you specify the following #pragma extensions at the same time, for the same function:

```
#pragma interrupt

#pargma inline

#pragma inline_asm

#pragma entry
```

3.21 Generating a GBR-Relative Logic Operation Instruction

Description:

When the "-gbr=user" option is specified with the "-logic_gbr" option, logical operation instructions relative to the GBR are used for external variables other than the GBR base variables specified in "#pragma gbr_base".

- Format:

```
-logic_gbr
```

Example of use:

C language code

```
char a,b,c;

main(){

     a &= 0x0f;

     b |= 0x01;

     c ^= 0x01;

}
```

Expanded into assembly language code ("-gbr_user" specified, "-logic_gbr" not specified)

```
          MOV.L       L11+2,R6   ; _a

          MOV.B       @R6,R0

          AND         #15,R0

          MOV.B       R0,@R6

          MOV.L       L11+6,R6   ; _b

          MOV.B       @R6,R0

          OR          #1,R0

          MOV.B       R0,@R6

          MOV.L       L11+10,R6  ; _c

          MOV.B       @R6,R0

          XOR         #1,R0

          RTS

          MOV.B       R0,@R6

L11:

          .RES.W      1

          .DATA.L     _a

          .DATA.L     _b

          .DATA.L     _c
```

Expanded into assembly language code ("-gbr_user" specified, "-logic_gbr" specified)

```
          MOV.L       L11+2,R0   ; _a-(STARTOF $G0)
          AND.B       #15,@(R0,GBR)                   ; GBR-relative operation instruction
          MOV.L       L11+6,R0   ; _b-(STARTOF $G0)
          OR.B        #1,@(R0,GBR)                    ; GBR-relative operation instruction
          MOV.L       L11+10,R0  ; _c-(STARTOF $G0)
          RTS
          XOR.B       #1,@(R0,GBR)                    ; GBR-relative operation instruction
 L11:
          .RES.W      1
          .DATA.L     _a-(STARTOF $G0)
          .DATA.L     _b-(STARTOF $G0)
          .DATA.L     _c-(STARTOF $G0)
```

To use the GBR as the base address, you must specify the GBR for the start address of the $G0 section beforehand, just as you would for "#pragma gbr_base".

Important Information:

(1) When you specify the "-logic_gbr" option, you must map the $G0 section.
(2) If you do not specify the "-gbr=user" option, the "-logic_gbr" option is disregarded.

3.22 Enabling Register Declarations

Description:

The compiler allocates registers to variables in order, based on the analysis results in the compiler, regardless of whether or not the registers are declared.
When the "-enable_register" option is specified, the registers are allocated first to the variables with the register declaration.

- Format:

```
-enable_register
```

Example of use:

C language code

```
int sum[10],input1[10],input2[10];

int b;


void func()

{

     register int a = 0;

     int i;


     while(b) {

     a++;

     for (i = 0; i < 10; i++) {

     sum[i] = input1[i] + input2[i];

       }

       b--;

     }


     printf("%d\n",a);          // Since the value of 'a' is passed to printf via R5,

                                // allocating R5 to 'a' improves efficiency.

}
```

Expanded into assembly language code ("-enable_register" not specified)

```
_func:

          MOV.L       R12,@-R15

          MOV.L       R13,@-R15

          MOV.L       R14,@-R15
          MOV.L       L16+2,R12
          MOV         #0,R13          ;  Since R5 was allocated to another variable with higher priority,
                                      ;  R13 is allocated to variable a.
                  :          :
          MOV.L       L16+22,R2       ; _printf
          MOV.L       R14,@R12
          MOV         R13,R5          ;  Copies the value of variable a (R13) to R5
          MOV.L       @R15+,R14
          MOV.L       @R15+,R13
          JMP         @R2             ;  Calls printf()
          MOV.L       @R15+,R12
```

Expanded into assembly language code ("-enable_register" specified)

```
 _func:
          MOV.L       R12,@-R15
          MOV.L       R13,@-R15
          MOV.L       R14,@-R15
          MOV.L       L16,R12         ; _b
          MOV         #0,R5           ;  Since variable a gives higher priority, R5 is allocated.
              :          :
          MOV.L       L16+20,R2       ; _printf
          MOV.L       R13,@R12
          MOV.L       @R15+,R14
          MOV.L       @R15+,R13
          JMP         @R2             ;  Calls printf()
          MOV.L       @R15+,R12
```

Important Information:

If a register is not allocated, the following information message appears:
C0102 (I) Register is not allocated to "variable name" in "function name"
However, this message does not appear if an argument is not allocated to any register.

3.23 Specifying Absolute Addresses of Variables

Description:

You can specify the absolute addresses of variables that are referenced externally, using a preprocessor directive. The compiler allocates the variables declared in the #pramga address directive to the corresponding absolute addresses. This feature enables easier access via variables to I/O allocated to a specific address.

- Format:

`#pragma address (<variable name> = <address value>[,<variable name> = <address value> ...] )`

Example of use:

`Variable”io” is allocated to the absolute address 0x100.`

C language code

```
#pragma address (io=0x100)

int io;

f()

{

     io = 10;

}
```

Expanded into assembly language code

```
_func:

          MOV          #1,R2

          SHLL8        R2

          MOV          #10,R6

          RTS

          MOV.L        R6,@R2

          .SECTION     $ADDRESS$B100,DATA,LOCATE=H'100

_io:

          .RES.L       1
```

Important Information:

(1) You must specify "#pragma address" before the variable declaration.

(2) An error will occur if you specify a compound type member or other than a variable.

(3) An error will occur if you specify an odd address for a variable or structure whose alignment number is 2. An error will also occur if you specify an address other than a multiple of four for a variable or structure whose alignment number is 4.

(4) An error will occur if you specify "#pragma address" more than once for the same variable.

(5) An error will occur if you specify the same address for different variables or if you specify the same variable address more than once.

(6) An error will occur if you specify the following #pragma extensions at the same time, for the same variable:

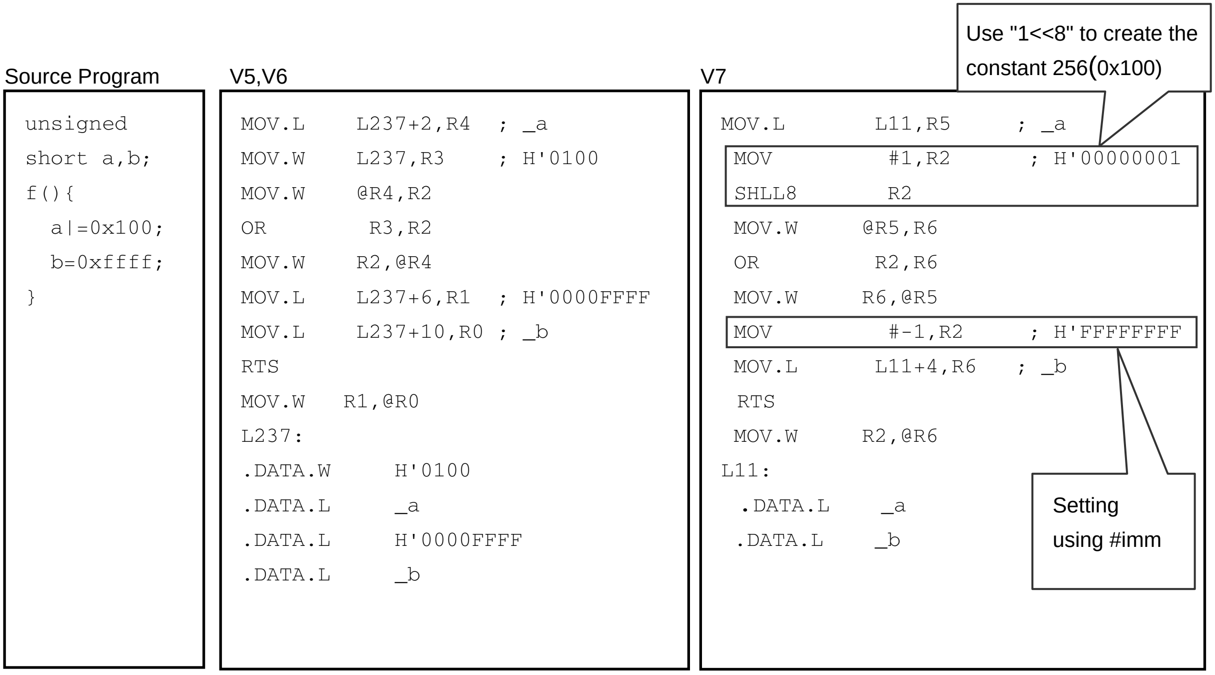
```
#pragma section
```

```
#pragma abs16/abs20/abs28/abs32

#pragma gbr_base/gbr_base1

#pragma global_register
```

3.24 Strengthened optimization

3.24.1 Improved Literal Data (1)

Constant data optimization has been strengthened.



Source Program

```
unsigned
short a,b;
f(){
  a|=0x100;
  b=0xffff;
}
```

V5,V6

```
MOV.L     L237+2,R4   ; _a
MOV.W     L237,R3     ; H'0100
MOV.W     @R4,R2
OR         R3,R2
MOV.W     R2,@R4
MOV.L     L237+6,R1   ; H'0000FFFF
MOV.L     L237+10,R0 ; _b
RTS
MOV.W   R1,@R0
L237:
.DATA.W      H'0100
.DATA.L      _a
.DATA.L      H'0000FFFF
.DATA.L      _b
```

V7

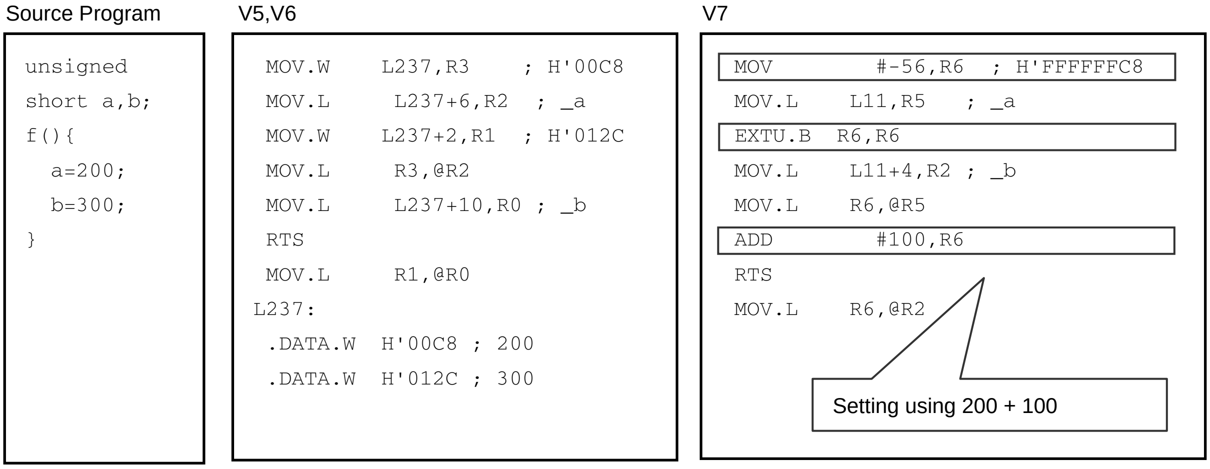
```
MOV.L       L11,R5      ; _a
 MOV          #1,R2       ; H'00000001
 SHLL8        R2
 MOV.W      @R5,R6
 OR          R2,R6
 MOV.W      R6,@R5
 MOV          #-1,R2      ; H'FFFFFFFF
 MOV.L       L11+4,R6   ; _b
 RTS
 MOV.W      R2,@R6
L11:
 .DATA.L    _a
 .DATA.L    _b
```

Use "1<<8" to create the constant 256(0x100)

Setting using #imm

3.24.2 Improved Literal Data (2)

Constant values of 2-bytes or more are reused.



Source Program

```
unsigned
short a,b;
f(){
  a=200;
  b=300;
}
```

V5,V6

```
 MOV.W    L237,R3    ; H'00C8
 MOV.L     L237+6,R2   ; _a
 MOV.W    L237+2,R1   ; H'012C
 MOV.L     R3,@R2
 MOV.L     L237+10,R0 ; _b
 RTS
 MOV.L     R1,@R0
L237:
 .DATA.W  H'00C8 ; 200
 .DATA.W  H'012C ; 300
```

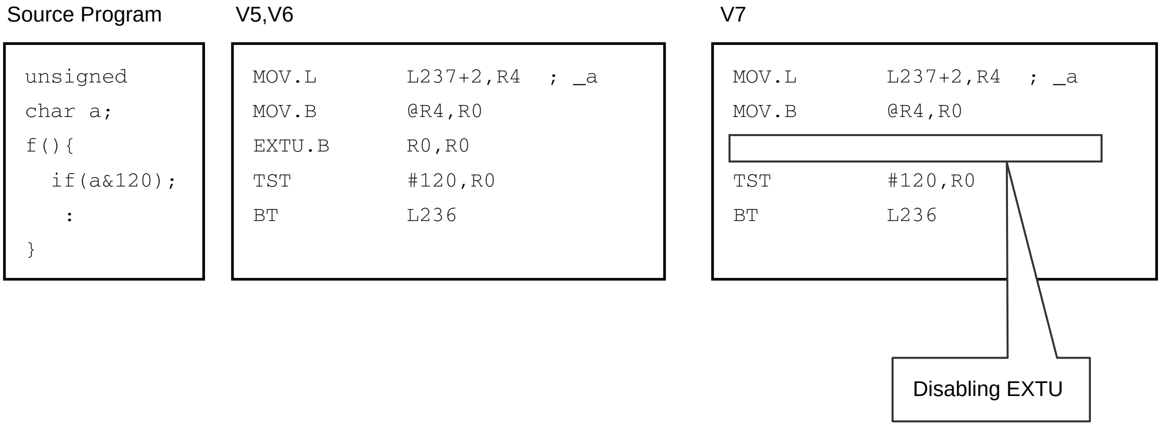
V7

```
MOV          #-56,R6   ; H'FFFFFFC8
MOV.L    L11,R5    ; _a
EXTU.B  R6,R6
MOV.L    L11+4,R2 ; _b
MOV.L    R6,@R5
ADD          #100,R6
RTS
MOV.L    R6,@R2
```

Setting using 200 + 100

3.24.3 Disabling EXTU (1)

Disables EXTU with regard to AND results in conditional expressions.



Source Program

```
unsigned
char a;
f(){
  if(a&120);
   :
}
```

V5,V6

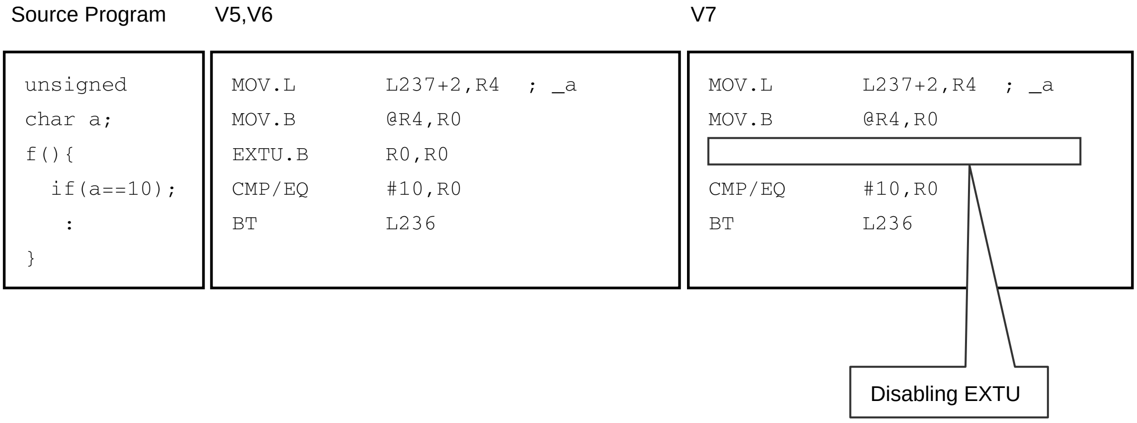
```
MOV.L       L237+2,R4  ; _a
MOV.B       @R4,R0
EXTU.B      R0,R0
TST         #120,R0
BT          L236
```

V7

```
MOV.L       L237+2,R4  ; _a
MOV.B       @R4,R0

TST         #120,R0
BT          L236
```

3.24.4 Disabling EXTU (2)

Disables EXTU when making comparisons with constants.



Source Program

```
unsigned
char a;
f(){
  if(a==10);
   :
}
```

V5,V6

```
MOV.L       L237+2,R4  ; _a
MOV.B       @R4,R0
EXTU.B      R0,R0
CMP/EQ      #10,R0
BT          L236
```

V7

```
MOV.L       L237+2,R4  ; _a
MOV.B       @R4,R0

CMP/EQ      #10,R0
BT          L236
```

3.24.5 Improved Bit Operations (1)

Improve comparison code for 1-bit data

Source Program

```
struct S{
unsigned char p0:1;
unsigned char p1:1;
unsigned char p2:1;
unsigned char p3:1;
unsigned char p4:1;
unsigned char p5:1;
unsigned char p6:1;
unsigned char p7:1;
}data;
     :
if(data.p7)
```

V5,V6

```
MOV.L     L239+2,R0   ; _data
MOV.B     @R0,R0
AND       #1,R0
EXTU.B    R0,R0
TST       R0,R0
BT        L238
```

V7

```
MOV.L     L239+2,R6   ; _data
MOV.B     @R6,R0

TST       #1,R0
BT        L238
```

TST with #1

3.24.6 Improved Bit Operations (2)

Improve substitution code for 1-bit data

Source Program

```
struct S{
unsigned char p0:1;
unsigned char p1:1;
unsigned char p2:1;
unsigned char p3:1;
unsigned char p4:1;
unsigned char p5:1;
unsigned char p6:1;
unsigned char p7:1;
}data1,data2;
     :
data1.p7=data2.p6;
```

V5,V6

```
 STS.L      PR,@-R15
 MOV.L      L239+4,R0   ; _data2
 MOV.L      L239+8,R2   ; _data1
 MOV.B      @R0,R0
 MOV.W      L239,R1     ; H'0701
 TST        #2,R0
 MOV.L     L239+12,R3 ; __bfsbu
 MOVT       R0
 ADD        #-1,R0
 JSR        @R3
 NEG        R0,R0
 LDS.L      @R15+,PR
 RTS
 NOP
L239:
 .DATA.W    H'0701
 .DATA.L    __bfsbu0
```

V7

```
MOV.L    L14+2,R6   ; _data2
MOV.B    @R6,R0
MOV.L    L14+6,R6   ; _data1
TST       #2,R0
MOV.B    @R6,R0
BF        L12
BRA      L13
AND      #254,R0
L12:
OR       #1,R0
L13:
RTS
  MOV.B   R0,@R6
```

Inline expansion without using a runtime routine

3.24.7 Improved Bit Operations (3)

Improve logic operation code for bit fields

Source Program

```
struct S{
unsigned char p0:4;
unsigned char p1:4;
}data1;
      :
data1.p1|=1;
```

V5,V6

```
STS.L       PR,@-R15
MOV.L    L236+4,R0  ; _data1
MOV.L    L236+4,R2  ; _data1
MOV.B    @R0,R0
MOV.W    L236,R1    ; H'0404
AND      #15,R0
MOV.L    L236+8,R3  ; __bfsbu
JSR       @R3
OR        #1,R0
LDS.L       @R15+,PR
RTS
NOP
L236:
.DATA.W  H'0404
.DATA.W  0
.DATA.L   _data1
.DATA.L   __bfsbu
```

V7

```
MOV.L     L11,R5   ; _data1
MOV.B     @R5,R2
MOV       R2,R0
AND       #15,R0
OR        #1,R0
AND       #15,R0
MOV       R0,R6
MOV       R2,R0
AND       #240,R0
OR        R6,R0
RTS
MOV.B     R0,@R5
L11:
  .DATA.L     _data1
```

Inline expansion without using a runtime routine

3.24.8 Improved Bit Operations (4)

Improve consecutive decision processing of the same bit field

Source Program

```
struct S{
unsigned char p0:1;
unsigned char p1:1;
unsigned char p2:1;
unsigned char p3:1;
unsigned char p4:1;
unsigned char p5:1;
unsigned char p6:1;
unsigned char p7:1;
}data;
      :
if(data.p7==1 &&
      data.p6==1)
```

V5,V6

```
MOV.L       L239+2,R4  ; _data
MOV         R4,R0
MOV.B       @R0,R0
AND         #1,R0
CMP/EQ      #1,R0
BF          L238
MOV         R4,R0
MOV.B       @R0,R0
TST         #2,R0
MOVT        R0
ADD         #-1,R0
NEG         R0,R0
CMP/EQ      #1,R0
BF          L238
```

V7

```
MOV.L      L14+2,R6   ; _data
MOV.B       @R6,R0
AND         #3,R0
CMP/EQ      #3,R0
BF          L12
```

Evaluate 2 bits simultaneously

3.24.9 Improved Bit Operations (5)

Improve consecutive substitution of the same bit field

Source Program

```
struct S{
unsigned char p0:1;
unsigned char p1:1;
unsigned char p2:1;
unsigned char p3:1;
unsigned char p4:1;
unsigned char p5:1;
unsigned char p6:1;
unsigned char p7:1;
}data;
    :
data.p0=0;
data.p1=0;

:
data.p7=0;
```

V5,V6

```
MOV.L      L240,R4    ; _data1
MOV.B      @R4,R0
AND        #127,R0
MOV.B      R0,@R4
MOV.B      @R4,R0
AND        #191,R0
MOV.B      R0,@R4
  :
MOV.B      @R4,R0
AND        #254,R0
RTS
MOV.B      R0,@R4
```

V7

```
MOV.L    L11,R2  ; _data1
MOV      #0,R3   ; H'00000000
RTS
MOV.B    R3,@R2
```

Set all bits simultaneously

3.25 Controlling the Output Order of Uninitialized Variables

Description

The "-bss_order" option can be used to allocate uninitialized variables by declaration order or definition order.

Specify -bss_order=declaration to allocate uninitialized variables by declaration order, or -bss_order=definition to allocate uninitialized variables by definition order. If this option is omitted, -bss_order=declaration is used.

Format

```
-bss_order={ declaration | definition }
```

Example of use

C language code

```
extern int a1;
extern int a2;
int a3;
extern int a4;
int a5;
int a2;
int a1;
int a4;
```

Expanded into assembly language code

When <-bss_order=declaration is specified:>

```
        .SECTION B,DATA,ALIGN=4
_a1:
        .RES.L 1
_a2:
        .RES.L 1
_a3:
        .RES.L 1
_a4:
        .RES.L 1
_a5:
        .RES.L 1
```

When <-bss_order=definition is specified:>

```
        .SECTION B,DATA,ALIGN=4
 _a3:
        .RES.L 1
 _a5:
        .RES.L 1
 _a2:
        .RES.L 1
 _a1:
        .RES.L 1
 _a4:
        .RES.L 1
```

Remarks

When the "stuff" option is specified, bss_order=definition always takes effect.

3.26 Specifying the Placement of Variables

Description:

The "-stuff" option can be used to place variables in different sections, by boundary alignment adjustment number. This reduces padding, to conserve memory.

A section type can be specified in the "-stuff" option. The variables belonging to the specified section type are placed in the sections for boundary alignment adjustment number 4, 2, and 1, based on the size of the data. If the section type is omitted, the option targets all variables.

Data within each section is output in order of definition. bss_order=declaration is disregarded if specified.

If "-nostuff" is specified, all variables are placed in the section with boundary alignment adjustment number 4.

Data within each section follows the definition order for the C and D sections, and bss_order for the B section.

If this option is omitted, "nostuff" is used.

Table 3.48 Relationship between Variable Size and Section Name

	Section Type	**Variable Size Size of Variable(in Bytes)**		
		4n	**4n+2**	**2n+1**
Const-type variables	const	C$4	C$2	C$1
Initialized vVariables with initial values	data	D$4	D$2	D$1
Uninitialized Variables variables without initial values	bss	B$4	B$2	B$1

- Format:

```
-stuff [=section-type[,...]]

-nostuff

section-type:{ Bss | Data | Const }
```

Example of use:

C language code

```
int a;

char b=0;

const short c=0;

struct {

char x;

char y;

} ST;
```

Expanded into assembly language code

```
               .SECTION C$2,DATA,ALIGN=2
 _c:
               .DATA.W  H’0000
               .SECTION D$1,DATA,ALIGN=1
 _b:
               .DATA.B  H’00
               .SECTION B$4,DATA,ALIGN=4
 _a:
               .RES.L  1
               .SECTION B$2,DATA,ALIGN=2
 _sr:
               .RES.B  2
```

- Remarks:

Variables for which #pragma gbr_base|gbr_base1 or #pragma global_register is specified are not affected by this option.

Section 4 HEW

4.1 Specifying options in HEW2.0 or later

You can specify options from the Build menu. Here is how to specify options from Renesas Integrated Development Environment. Select "SuperH RISC engine Standard Toolchain" from the build menu.

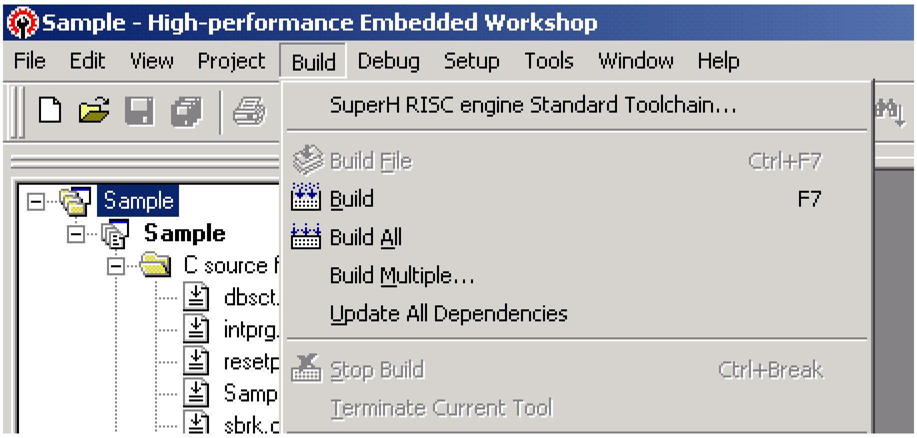


Figure 4.1 HEW Build Menu

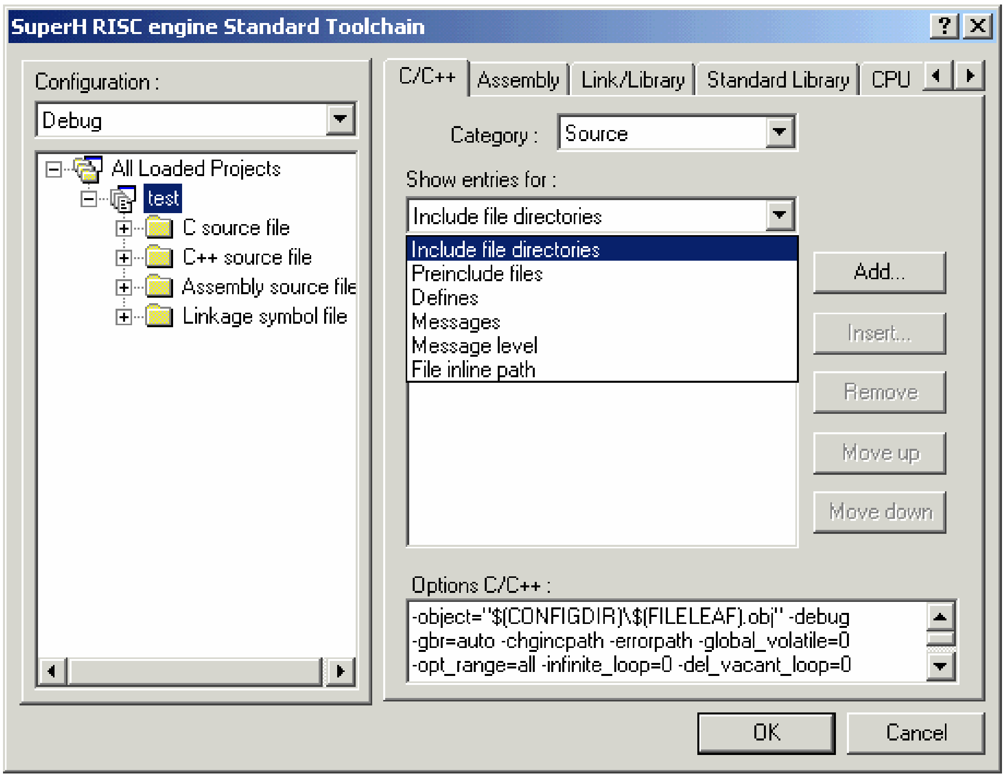
4.1.1 C/C++ Compiler Options

Select the C/C++ tab from the SuperH RISC engine Standard Toolchain dialog box.

(1) Category:[Source]

Table 4.1 Correspondence between Items on the Category:[Source] and Compiler Options

Dialog Box	Option
Show entries for :	
Include files directories	Include = <path name>[,…]
Preinclude files	PREInclude = <file name>[,…]
Defines	DEFine = <sub>[,…]
	<sub> : <macro name> [= <string>]
Messages	MEssage
Message leve	CHAnge_message = <sub>[,…]
	<sub> : <level>[=<n>[-m],…]
	<level> : {Information | Warning | Error}
File inline path	FILE_INLINE_PATH = <path name>[,…]
Display information level messages	NOMEssage [= <error number> [- <error number>[,…]]



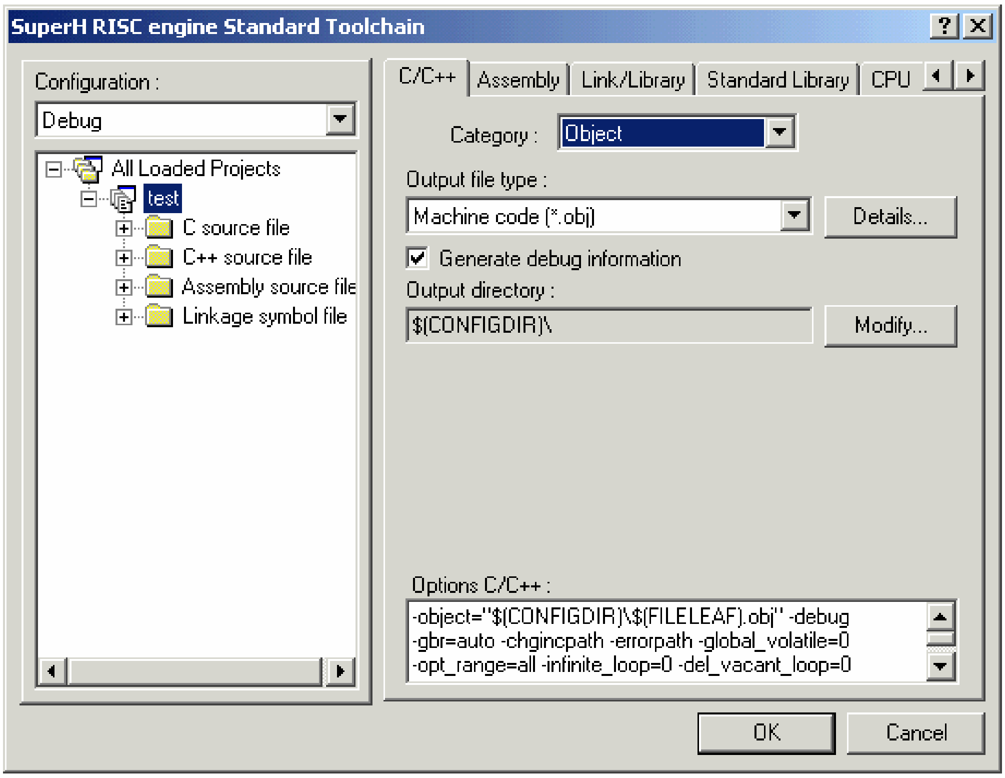
```
-object="$(CONFIGDIR)\$(FILELEAF).obj" -debug
-gbr=auto -chgincpath -errorpath -global_volatile=0
-opt_range=all -infinite_loop=0 -del_vacant_loop=0
```

Figure 4.2 Category:[Source] Dialog Box

(2) Category:[Object]

Table 4.2 Correspondence between Items on the Category:[Object] and Compiler Options

Dialog Box	Option
Output file type :	
Machine code (*.obj)	Code = Machinecode
Assembly source code (*.src)	Code = Asmcode
Preprocessed source file (*.p/*.pp)	PREProcessor [= <file name>]
Suppress #line in preprocessed source file	NOLINe
Generate debug information	DEBug / NODEBug
Output directory :	OBjectfile = <file name>



Configuration : Debug

Category : Object

Output file type : Machine code (*.obj)

☑ Generate debug information

Output directory : $(CONFIGDIR)\

Options C/C++ :

```
-object="$(CONFIGDIR)\$(FILELEAF).obj" -debug
-gbr=auto -chgincpath -errorpath -global_volatile=0
-opt_range=all -infinite_loop=0 -del_vacant_loop=0
```

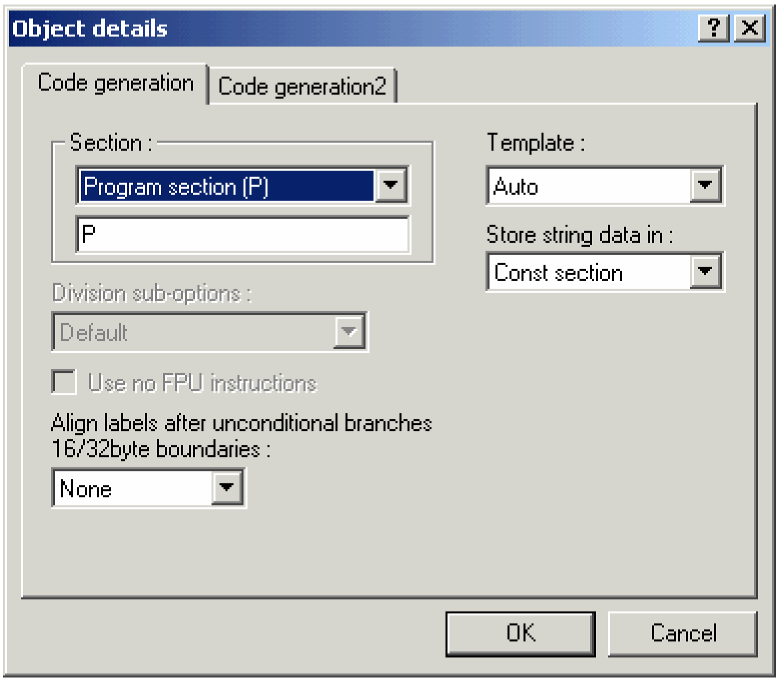
Figure 4.3 Category:[Object] Dialog Box

Clicking on [Details…] opens the "Optimize details" dialog box.

(a) Code generation tab

Table 4.3 Correspondence between Items on the Optimize details Dialog Box and Compiler Options

Dialog Box	Option
Section :	SEction = <sub>[,…]
Program section (P)	<sub> : Program = <section name>
Const section (C)	<sub> : Const = <section name>
Data section (D)	<sub> : Data = <section name>
Uninitialized data section (B)	<sub> : Bss = <section name>
	Default: (p=P, c=C, d=D, b=B)
Template :	Template =
	{ None | Static | Used |
	ALl | AUto }
Store string data in :	STring = { Const | Data }
Division sub-options :	DIvision = Cpu [= { Inline | Runtime }]
Use no FPU instructions	IFUnc
Align labels after unconditional branches	ALIGN16
16/32 byte boundaries :	ALIGN32
	NOALign



Object details

Section : Program section (P)

P

Template : Auto

Store string data in : Const section

Division sub-options : Default

□ Use no FPU instructions

Align labels after unconditional branches 16/32byte boundaries : None

Figure 4.4 Code Generation Tab Dialog Box

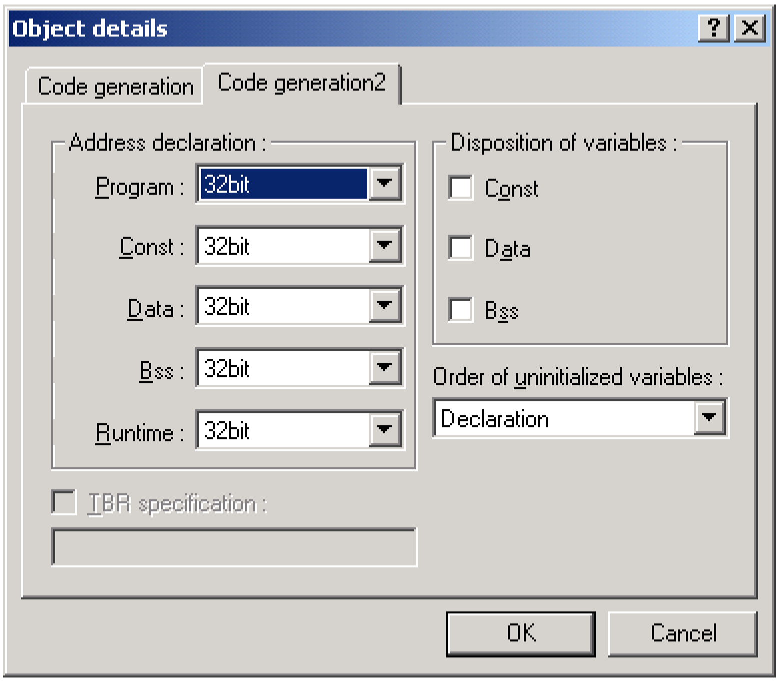
(b) Code generation2 tab

Table 4.4 Correspondence between Items on the Optimize details Dialog Box and Compiler Options

Dialog Box	Option
Address declaration :	<ABS> = <sub>[,…] <ABS> : { ABs16 | ABS20 | ABS28 | ABS32 } <sub> : { Program | Const | Data | Bss | Run | All }
TBR specification :	TBR [= <section name>]

Disposition of variable :	STUff=<sub>[,...]
	<sub>:
	{Bss|Data|Const}

Order of uninitialized variables :	BSs_order=<sub>
	<sub>:
	{DEClaration|DEFinition}



Program : 32bit
Const : 32bit
Data : 32bit
Bss : 32bit
Runtime : 32bit

□ Const
□ Data
□ Bss

Order of uninitialized variables : Declaration

□ TBR specification :

Figure 4.5 Code Generation2 Tab Dialog Box

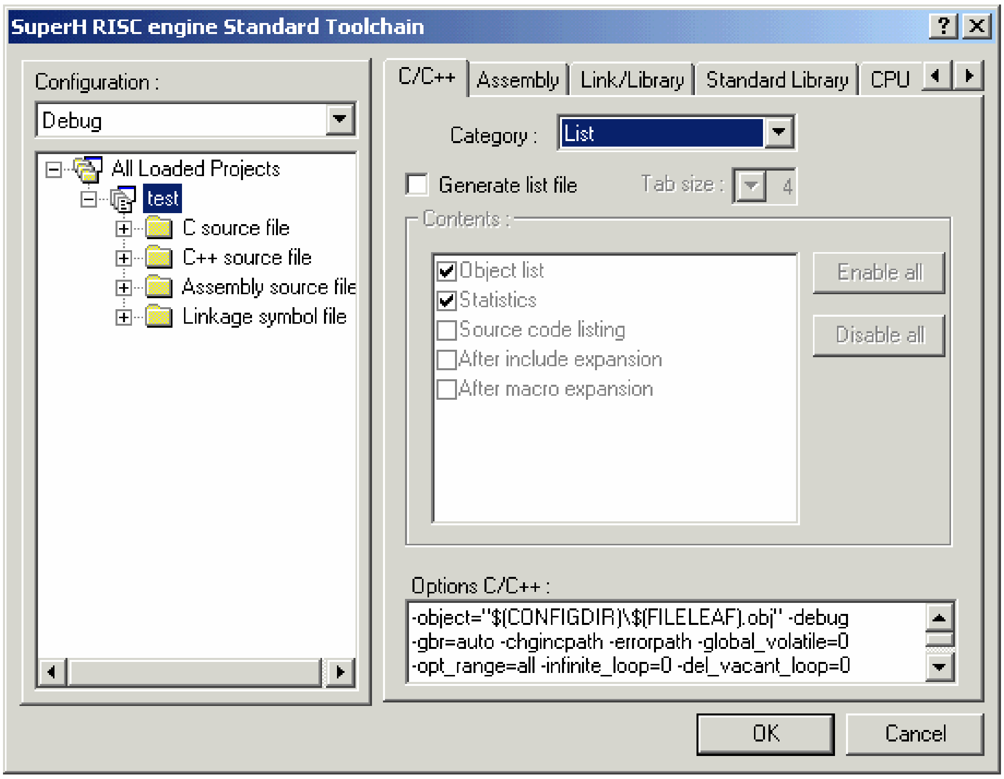
(3) Category:[List]

Table 4.5 Correspondence between Items on the Category:[List] and Compiler Options

Dialog Box	Option
Generate list file	Listfile [= <file name>] / NOListfile

Tab size :	SHow = <sub>[,…]
	<sub> : Tab = { 4 | 8 }

Contents :	SHow = <sub>[,…]
Object list	<sub> : Object / NOObject
Statistics	<sub> : STatistics / NOSTatistics
Source code listing	<sub> : SOurce / NOSOurce
After include expansion	<sub> : Include / NOInclude
After macro expansion	<sub> : Expansion / NOExpansion


SuperH RISC engine Standard Toolchain

Configuration : Debug

All Loaded Projects
- test
  - C source file
  - C++ source file
  - Assembly source file
  - Linkage symbol file

Category : List

□ Generate list file Tab size : 4

Contents :
- ☑ Object list
- ☑ Statistics
- □ Source code listing
- □ After include expansion
- □ After macro expansion

Options C/C++ :

```
-object="$(CONFIGDIR)\$(FILELEAF).obj" -debug
-gbr=auto -chgincpath -errorpath -global_volatile=0
-opt_range=all -infinite_loop=0 -del_vacant_loop=0
```

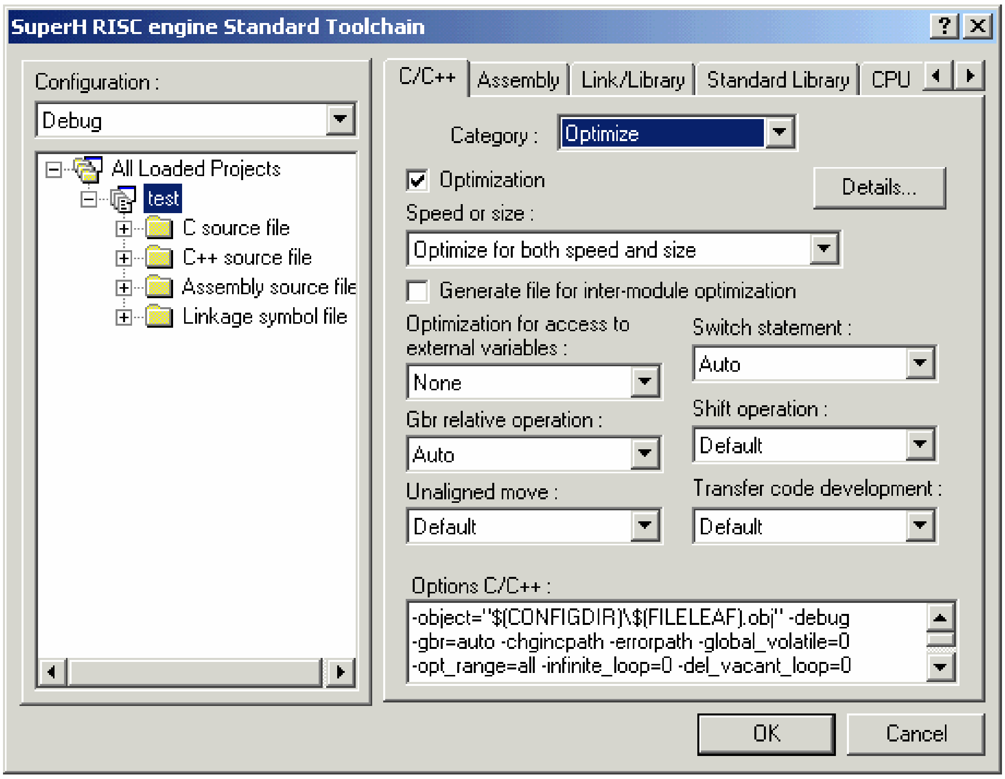
Figure 4.6 Category:[List] Dialog Box

When both the -nolist and -show options are specified, the -nolist option takes precedence.

(4) Category:[Optimize]

Table 4. 6 Correspondence between Items on the Category:[Optimize] and Compiler Options

Dialog Box	Option
Optimization	OPtimize = 1 / OPtimize = 0
Speed or size :	
Optimize for speed	SPeed
Optimize for size	SIze
Optimize for both speed and size	NOSPeed
Generate file for inter-module optimization	Goptimize
Optimization for access to external valriables :	MAP = <file name>
GBR relative operation :	GBr = { Auto | User }
Unaligned move :	Unaligned = { Inline | Runtime }
Switch statement :	CAse = { Ifthen | Table }
Shift operation :	SHIft = { Inline | Runtime }
Transfer code development :	BLOckcopy = { Inline | Runtime }



Configuration : Debug

Category : Optimize

☑ Optimization

Speed or size : Optimize for both speed and size

☐ Generate file for inter-module optimization

Optimization for access to external variables : None

Gbr relative operation : Auto

Unaligned move : Default

Switch statement : Auto

Shift operation : Default

Transfer code development : Default

Options C/C++ :

```
-object="$(CONFIGDIR)\$(FILELEAF).obj" -debug
-gbr=auto -chgincpath -errorpath -global_volatile=0
-opt_range=all -infinite_loop=0 -del_vacant_loop=0
```

Figure 4.7 Category:[Optimize] Dialog Box

- For the Speed or Size option, select the "Optimize for both speed and size".

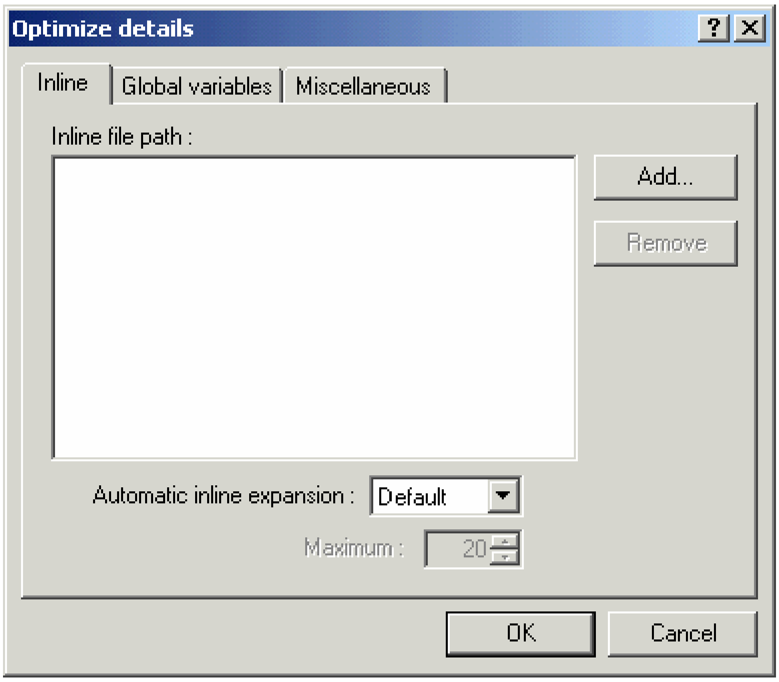
Clicking on [Details…] opens the "Optimize details" dialog box.

The options that have been added in the V.7.0.06 should be specified in this dialog box.

(a) Inline tab

Table 4.7 Correspondence between Items on the Optimize details Dialog Box and Compiler Options

Dialog Box	Option
Inline	-
Inline file path :	File_inline = <file name>[,…]
Automatic inline expansion :	INLine [= <numeric value>] / NOINLine



Inline file path :

Automatic inline expansion : Default

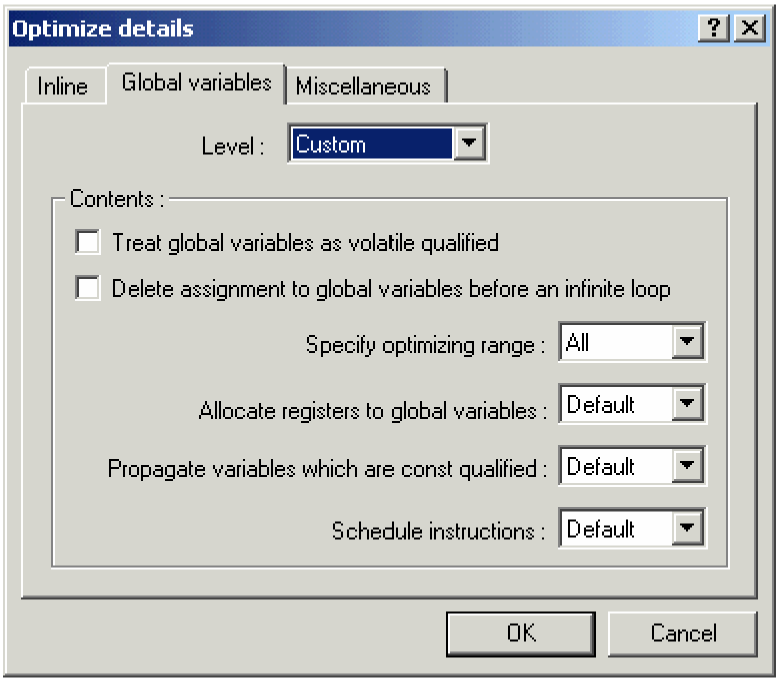
Maximum : 20

Figure 4.8 Inline Tab Dialog Box

(b) Global variables tab

Table 4.8 Correspondence between Items on the Optimize details Dialog Box and Compiler Options

Dialog Box	Option
Level :	-
Contents :	
Treat Global variables as voaltile qualified	GLOBAL_Volatile = 1 / GLOBAL_Volatile = 0
Delete assignment to global variables before an infinite loop	INFinite_loop = 1 / INFinite_loop = 0
Specify optimizing range :	OPT_Range = { All | NOLoop | NOBlock }
Allocate registers to global variables :	
Disable	GLOBAL_Alloc = 0
Enable	GLOBAL_Alloc = 1
Default	-
Propagate variables which are const qualified :	
Disable	CONST_Var_propagate = 0
Enable	CONST_Var_propagate = 1
Default	-
Schedule instructions :	
Disable	SChedule = 0
Enable	SChedule = 1
Default	-



Level : Custom

Contents :

☐ Treat global variables as volatile qualified

☐ Delete assignment to global variables before an infinite loop

Specify optimizing range : All

Allocate registers to global variables : Default

Propagate variables which are const qualified : Default

Schedule instructions : Default

Figure 4.9 Global Variables Tab Dialog Box

By setting Level, the optimizations for external variables can be controlled collectively.

Level1
Disable the optimization of external variables.
gloal_volatile = 1
opt_range = noblock
infinite_loop = 0
global_alloc = 0
const_var_propagate = 0
schedule = 0

Level2
Optimize external variables without volatile specification within the range of a branch (including a loop).
gloal_volatile = 0
opt_range = noblock
infinite_loop = 0
global_alloc = 0
const_var_propagate = 0
schedule = 1

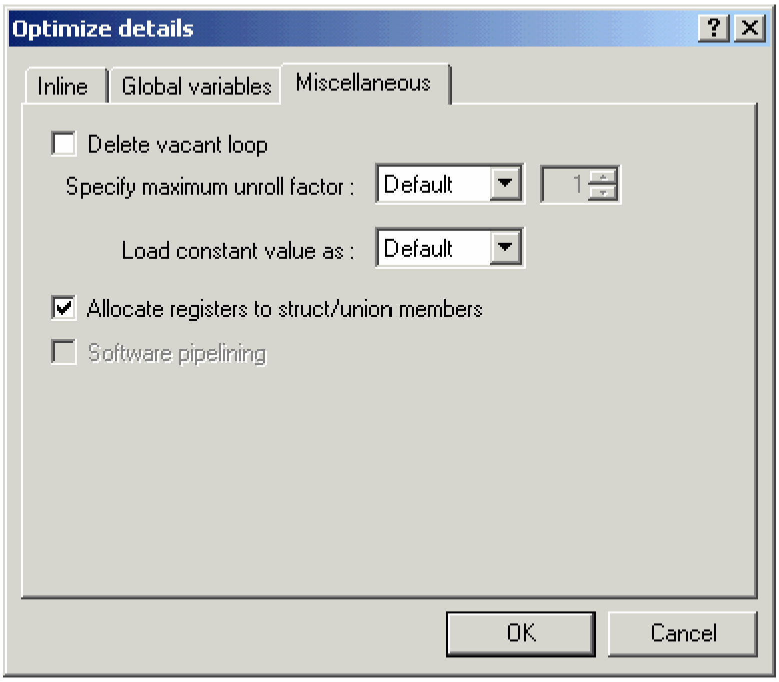
Level3
Optimize all the external variables without volatile specification.
gloal_volatile = 0
opt_range = all
infinite_loop = 0
global_alloc = 1
const_var_propagate = 1
schedule = 1

Custom
The user specifies the optimization of external variables according to the program.

(c) Miscellaneous tab

Table 4.9 Correspondence between Items on the Miscellaneous Tab Dialog Box and Compiler Options

Dialog Box	Option
Delete vacant loop	DEL_vacant_loop = 1 / DEL_vacant_loop = 0
Specify maximum unroll factor :	MAX_unroll = <numeric value> : 1 to 32
Load constant value as :	
Inline	CONST_Load = Inline
Literal	CONST_Load = Literal
Default	-
Allocate registers to struct/union members :	STRUCT_Alloc = 1 / STRUCT_Alloc = 0
Software pipelining :	SOftpipe



□ Delete vacant loop

Specify maximum unroll factor : Default 1

Load constant value as : Default

☑ Allocate registers to struct/union members

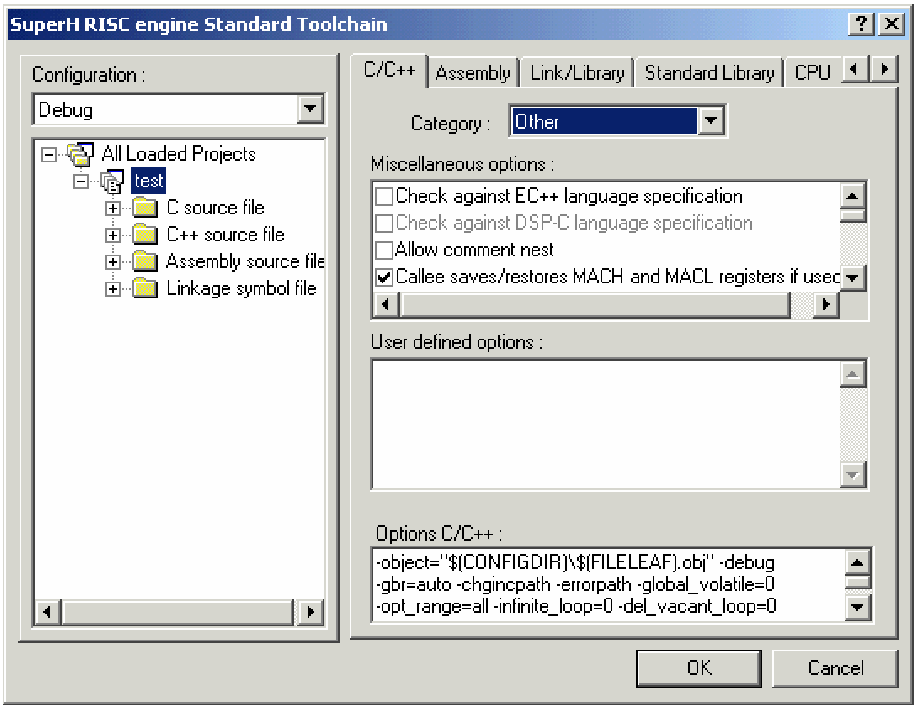
□ Software pipelining

Figure 4.10 Miscellaneous Tab Dialog Box

(5) Category:[Other]

Table 4.10 Correspondence between Items on the Category:[Other] and Compiler Options

Dialog Box	Option
Miscellaneous options :	
Check against EC++ language specification	ECpp
Check against DSP-C language specification	DSpc
Allow comment nest	COMment = Nest / COMment = NONest
Callee saves/restores MACH and MACL registers if used	Macsave = 1 / Macsave = 0
Saves/restores SSR and SPC registers	SAve_cont_reg = 0 / SAve_cont_reg = 1
Expand return value to 4 byte	RTnext / NORTnext
Loop unrolling	LOop / NOLOop
Approximate a floating-point constant division	APproxdiv
Avoid illegal SH7055 instructions	PAtch = 7055
Change FPSCR register if double data used	FPScr = Safe / FPScr = Aggressive
Treats loop condition as volatile qualified	Volatile_loop
enum size is made the smallest	AUto_enum
Floating-point contant is handled as a fixed-point constant	FIXED_Const
Treats 1.0 as maximum number of fixed type	FIXED_Max
Delete type conversion after fixed multiple	FIXED_Noround
DSP repeat loop is used	REPeat
Enable register declaration	ENAble_register declaration
Obey ansi specifications more strictly	STRIct_ansi
Change integer division into floating-point	FDIv



Configuration : Debug

Category : Other

Miscellaneous options :

☐ Check against EC++ language specification
☐ Check against DSP-C language specification
☐ Allow comment nest
☑ Callee saves/restores MACH and MACL registers if usec

User defined options :

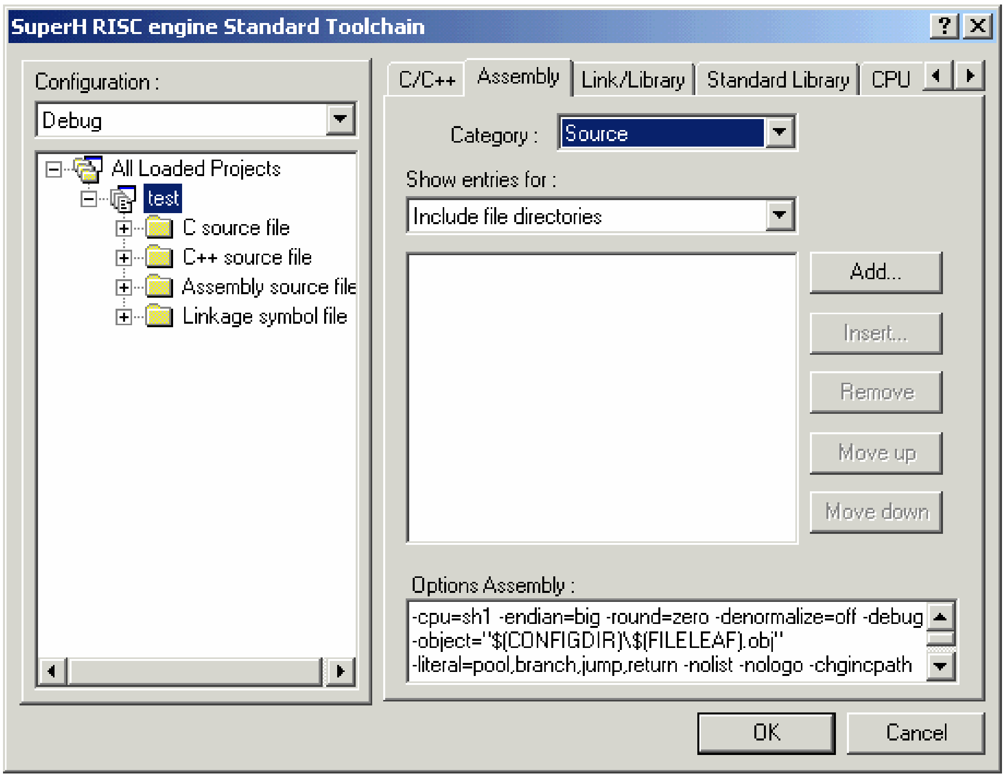
Options C/C++ :

```
-object="$(CONFIGDIR)\$(FILELEAF).obj" -debug
-gbr=auto -chgincpath -errorpath -global_volatile=0
-opt_range=all -infinite_loop=0 -del_vacant_loop=0
```

Figure 4.11 Category:[Other] Dialog Box

4.1.2 Assembly Options

Select the Assemby tab from the SuperH RISC engine Standard Toolchain dialog box.



Configuration : Debug

Category : Source

Show entries for : Include file directories

Options Assembly :

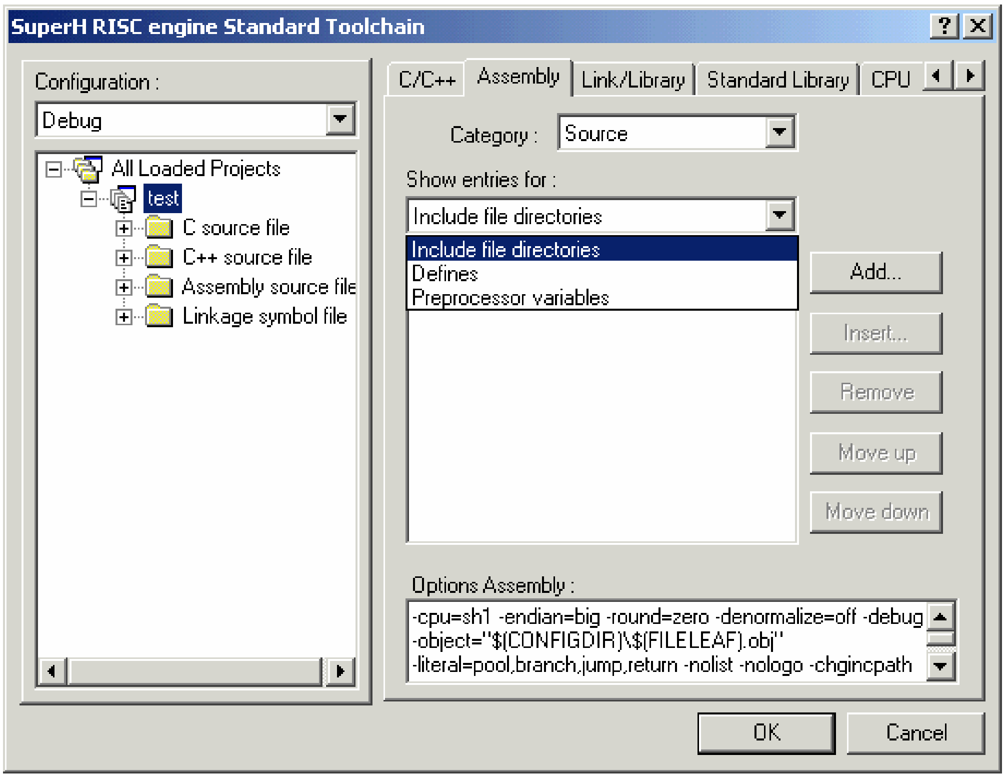
```
-cpu=sh1 -endian=big -round=zero -denormalize=off -debug
-object="$(CONFIGDIR)\$(FILELEAF).obj"
-literal=pool,branch,jump,return -nolist -nologo -chgincpath
```

Figure 4.12 Assembly Tab Dialog Box

(1) Category:[Source]

Table 4.11 Correspondence between Items on the Category:[Source] and Assembler Options

Dialog Box	Option
Show entries for :	
Include file directories	Include = <path name>[,…]
Defines	DEFine = <sub>[, …]
	<sub> :
	<replacement symbol> = "<string literal>"
Preprocessor variables	ASsignA = <sub>[, …]
	<sub> :
	<variable name> = <Integer constant>
	ASsignC = <sub>[, …]
	<sub> :
	<variable name> = "<string literal>"



Configuration : Debug

Category : Source

Show entries for : Include file directories

Options Assembly :

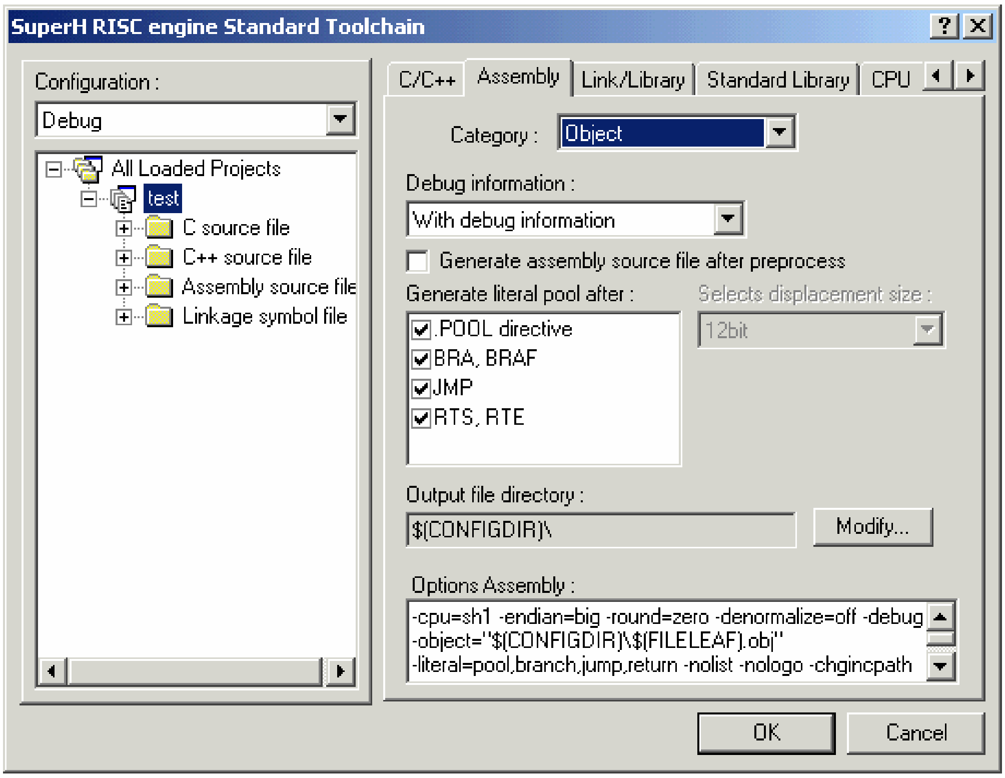
```
-cpu=sh1 -endian=big -round=zero -denormalize=off -debug
-object="$(CONFIGDIR)\$(FILELEAF).obj"
-literal=pool,branch,jump,return -nolist -nologo -chgincpath
```

Figure 4.13 Category:[Source] Dialog Box

(2) Category:[Object]

Table 4.12 Correspondence between Items on the Category:[Object] and Assembler Options

Dialog Box	Option
Debug information :	
Default	-
With debug information	Debug
Without debug information	NODebug
Geberate assembly source file after preprocess	EXPand [= <output file name>]
Generate literal pool after :	LITERAL = <point>[,…]
.POOL directive	<point> : Pool
BRA, BRAF	<point> : Branch
JMP	<point> : Jump
RTS, RTE	<point> : Return
Selects displacement size :	DIspsize = { 4 | 12 }
Output file directory: :	Object [= <output file name>] / NOObject



Configuration : Debug

Category : Object

Debug information : With debug information

☐ Generate assembly source file after preprocess

Generate literal pool after :

☑ .POOL directive
☑ BRA, BRAF
☑ JMP
☑ RTS, RTE

Selects displacement size : 12bit

Output file directory : $(CONFIGDIR)\

Options Assembly :

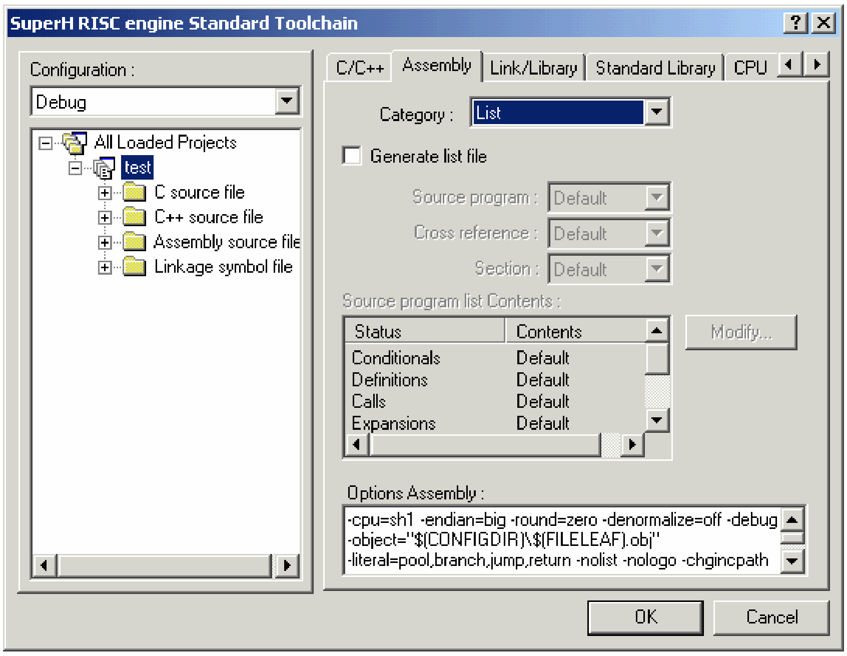
```
-cpu=sh1 -endian=big -round=zero -denormalize=off -debug
-object="$(CONFIGDIR)\$(FILELEAF).obj"
-literal=pool,branch,jump,return -nolist -nologo -chgincpath
```

Figure 4.14 Category:[Object] Dialog Box

(3) Category:[List]

Table 4.13 Correspondence between Items on the Category:[List] and Assembler Options

Dialog Box	Option
Generate list file	LISt [= <output file name>] / NOLISt
Source program :	
Default	-
Shown	SOurce
Not shown	NOSOurce
Cross reference :	
Default	-
Shown	CRoss_reference
Not shown	NOCRoss_reference
Section :	
Default	-
Shown	SEction
Not shown	NOSEction
Source program list Contents :	
Contents	
Default / Shown / Not shown	- / SHow [= <item>[,…]] / NOSHow [= <item>[,…]]
Status	
Conditions	<item> : CONditionals
Definitions	<item> : Definitions
Calls	<item> : CAlls
Expansions	<item> : Expansions
Code	<item> : CODe
Tab Size	<item> : TAB = { 4 / 8 }



Configuration : Debug

- All Loaded Projects
  - test
    - C source file
    - C++ source file
    - Assembly source file
    - Linkage symbol file

Category : List

□ Generate list file

Source program : Default

Cross reference : Default

Section : Default

Source program list Contents :

Status	Contents
Conditionals	Default
Definitions	Default
Calls	Default
Expansions	Default

Options Assembly :

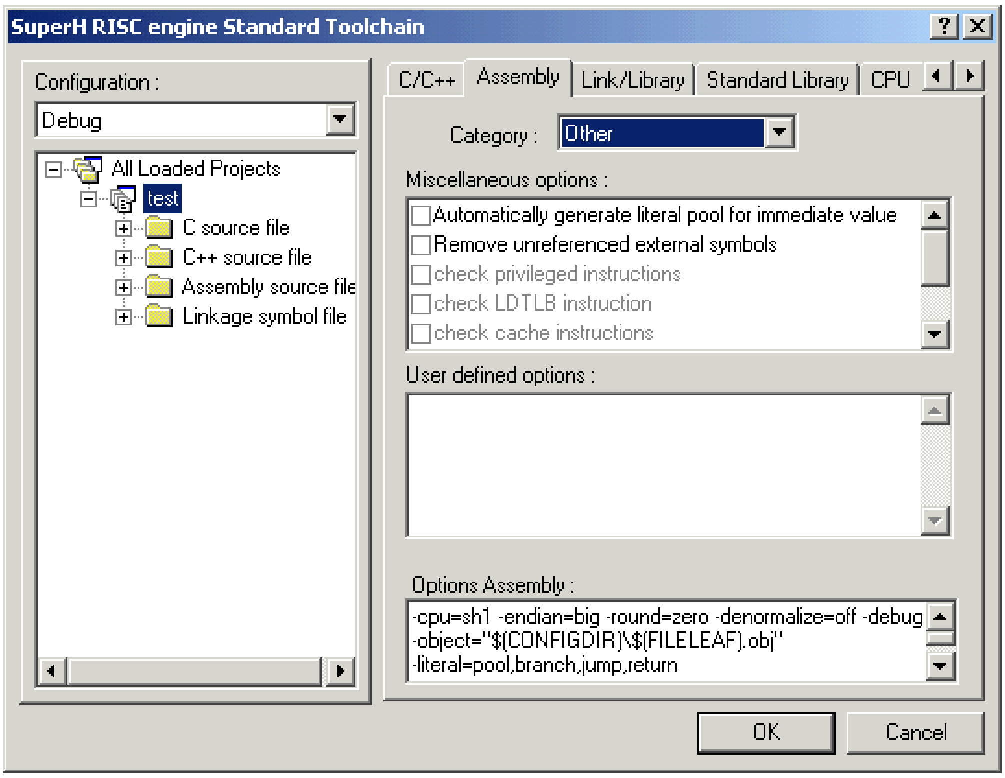
```
-cpu=sh1 -endian=big -round=zero -denormalize=off -debug
-object="$(CONFIGDIR)\$(FILELEAF).obj"
-literal=pool,branch,jump,return -nolist -nologo -chgincpath
```

Figure 4.15 Category:[List] Dialog Box

(4) Category:[Other]

Table 4.14 Correspondence between Items on the Category:[Other] and Assembler Options

Dialog Box	Option
Miscellaneous options :	
Automatically generate literal pool for immediate value	AUTO_literal
Remove unrefernced external symbols	Exclude / NOExclude
check privileged instructions	CHKMd
check LDTLB instruction	CHKTlb
check cache instructions	CHKCache
check DSP instructions	CHKDsp
check FPU instructions	CHKFpu
check 8-byte alignment of FDATA	CHKAlign8



Category : Other

Miscellaneous options :

□ Automatically generate literal pool for immediate value
□ Remove unreferenced external symbols
□ check privileged instructions
□ check LDTLB instruction
□ check cache instructions

User defined options :

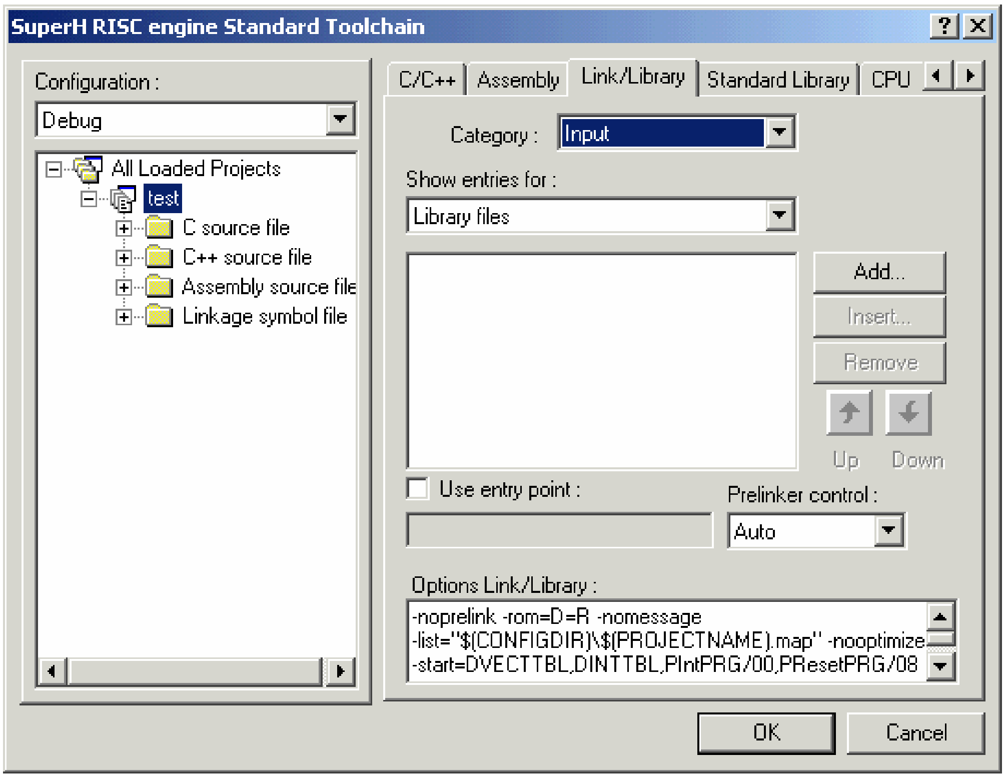
Options Assembly :

```
-cpu=sh1 -endian=big -round=zero -denormalize=off -debug
-object="$(CONFIGDIR)\$(FILELEAF).obj"
-literal=pool,branch,jump,return
```

Figure 4.16 Category:[Other] Dialog Box

4.1.3 Optimizing Linkage Editor Options

Select the Link/Library tab from the SuperH RISC engine Standard Toolchain dialog box.



SuperH RISC engine Standard Toolchain

Configuration : Debug

All Loaded Projects
- test
  - C source file
  - C++ source file
  - Assembly source file
  - Linkage symbol file

Category : Input

Show entries for : Library files

□ Use entry point :

Prelinker control : Auto

Options Link/Library :

```
-noprelink -rom=D=R -nomessage
-list="$(CONFIGDIR)\$(PROJECTNAME).map" -nooptimize
-start=DVECTTBL,DINTTBL,PIntPRG/00,PResetPRG/08
```

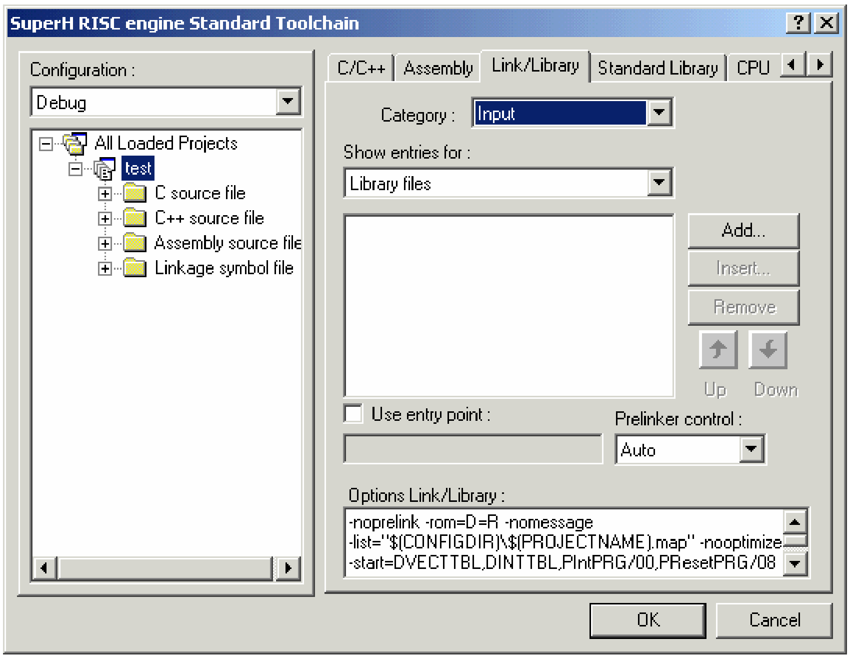
Figure 4.17 Link/Library Tab Dialog Box

(1) Category:[Input]

Table 4.15 Correspondence between Items on the Category:[Input] and Linkage Editor Options

Dialog Box	Option
Show entries for :	
Library files	LIBrary = <file name>[,…]
Relocatable files and object files*[1]	Input = <sub> [{,|Δ}...]
	<sub> :
	<file name>[(<module name>[,...])]
Binary files	Binary = <sub>[,...]
	<sub> :
	<file name>(<section name>
	[:<boundary alignment>]
	[,<symbol name>])
Defines	DEFine = <sub>[,...]
	<sub> :
	<symbol name> = { <symbol name>
	| <numerical value> }
Use entry point :	ENTry = { <symbol name> | <address> }
Prelinker control :	
Auto	NOPRElink
Skip prelinker	NOPRElink
Run prelinker	-

*1 Files which are included in the project need not be added explicitly; this is specified when linking objects which are not compiled/assembled.



Configuration : Debug

Category : Input

Show entries for : Library files

☐ Use entry point :

Prelinker control : Auto

Options Link/Library :

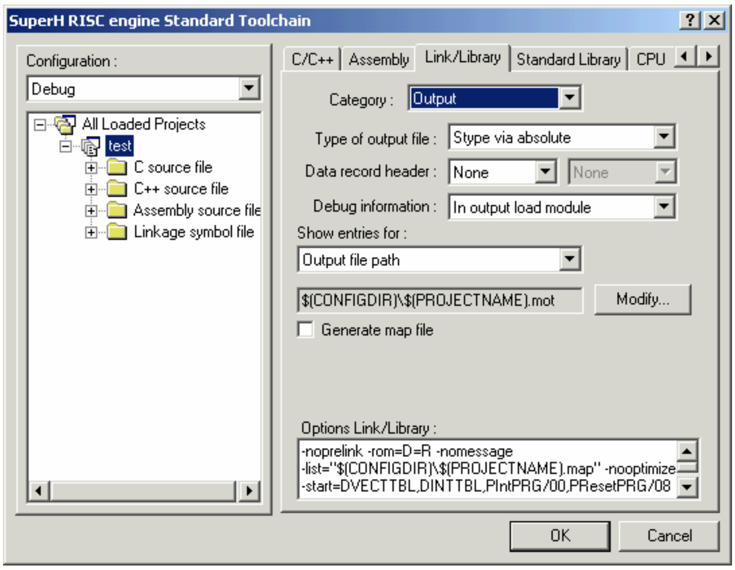
```
-noprelink -rom=D=R -nomessage
-list="$(CONFIGDIR)\$(PROJECTNAME).map" -nooptimize
-start=DVECTTBL,DINTTBL,PIntPRG/00,PResetPRG/08
```

Figure 4.18 Category:[Input] Dialog Box

(2) Category:[Output]

Table 4.16 Correspondence between Items on the Category:[Output] and Optimizing Linkage Editor Options

Dialog Box	Option
Type of output file :	
Absolute(ELF/DWARF)	FOrm = Absolute
Absolute(SYSROF)	FOrm = Absolute
Relocatable	FOrm = Relocate
System library	FOrm = Library = S
User library	FOrm = Library = U
Hex via absoulte	FOrm = Hexadecimal
S type bia absoulte	FOrm = Stype
Binary via absoulte	FOrm = Binary
Data record header :	REcord = { H16 | H20 | H32 | S1 | S2 | S3 }
Debug information :	
None	NODEBug
In output load module	DEBug
In separate debug file (*.dbg)	SDebug
Show entries for :	
Output file path/message	
ROM to RAM mapped sections	ROm = <sub>[,…]
	<sub> : <ROM section name>=<RAM section name>
Divided output files	OUtput = <sub>[,…]
	<sub> : <file name>[=<output range>]
	<output range> :
	{ <start address> - <end address> |
Output padding data	<section name>[: ...] }
Repressed information level messages	SPace = [<numerical value>]
	NOMessage [= <sub>[,…]] / Message
	<sub> : <error code> [- <error code>]
Generate map file	MAp [= <file name>]



Configuration : Debug

Category : Output

Type of output file : Stype via absolute

Data record header : None None

Debug information : In output load module

Show entries for : Output file path

$(CONFIGDIR)\$(PROJECTNAME).mot

□ Generate map file

Options Link/Library :

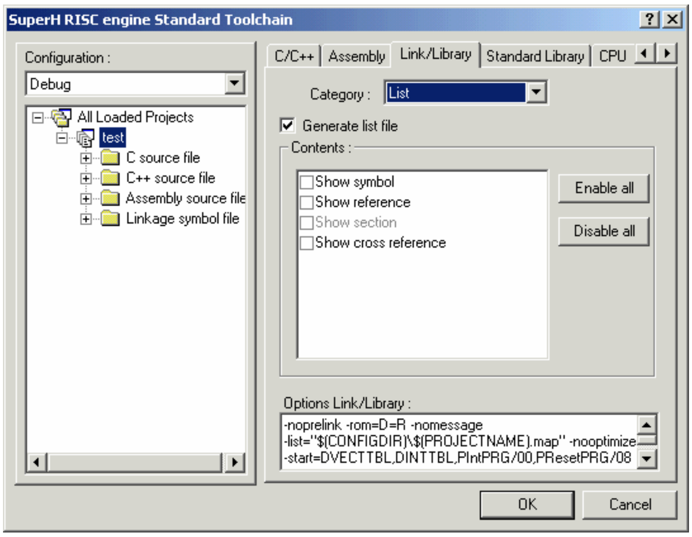
```
-noprelink -rom=D=R -nomessage
-list="$(CONFIGDIR)\$(PROJECTNAME).map" -nooptimize
-start=DVECTTBL,DINTTBL,PIntPRG/00,PResetPRG/08
```

Figure 4.19 Category:[Output] Dialog Box

(3) Category:[List]

Table 4.17 Correspondence between Items on the Category:[List] and Optimizing Linkage Editor Options

Dialog Box	Option
Generate list file	LISt [= <file name>] / -
Contents :	SHow [= <sub>[,...]]
Show symbol	<sub> : SYmbol
Show reference	<sub> : Reference
Show section	<sub> : SEction
Show cross reference	<sub> : Xreference



Configuration : Debug

Category : List

☑ Generate list file

Contents :

□ Show symbol
□ Show reference
□ Show section
□ Show cross reference

Options Link/Library :

```
-noprelink -rom=D=R -nomessage
-list="$(CONFIGDIR)\$(PROJECTNAME).map" -nooptimize
-start=DVECTTBL,DINTTBL,PIntPRG/00,PResetPRG/08
```

Figure 4.20 Category:[List] Dialog Box

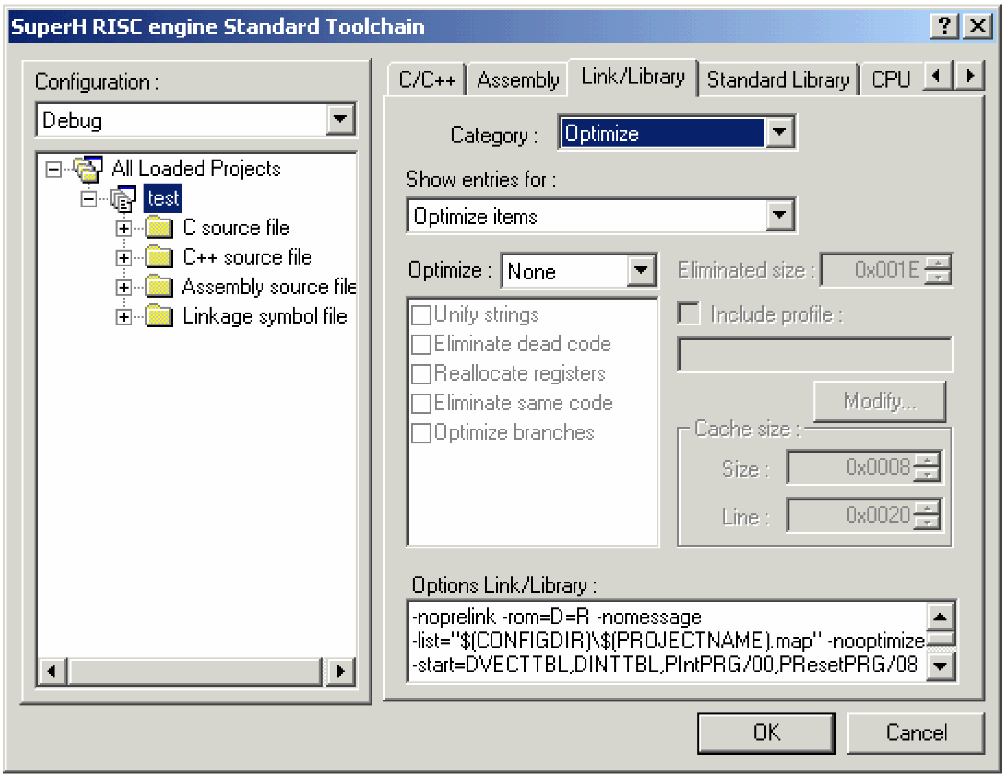
(4) Category:[Optimize]

Table 4.18 Correspondence between Items on the Category:[Optimize] and Optimizing Linkage Editor Options

Dialog Box	Option
Show entries for :	
Optimize items	
Optimize :	OPtimize [= <sub>[,...]]
All	<sub> : STring_unify,SYmbol_delete, Variable_access,Register, SAMe_code,SHort_format, Function_call,Branch
Speed	<sub> : SPeed
Safe	<sub> : SAFe
Custom	Optionally specify the folloing:
Unify strings	<sub> : STring_unify
Eliminate dead code	<sub> : SYmbol_delete
Reallocate registers	<sub> : Register
Eliminate same code	<sub> : SAMe_code
Optimize branches	<sub> : Branch
None	NOOPtimize
Eliminated size :	SAMESize = <size> (default:sames=1e)
Include profile :	PROfile = <file name>

Cache size :	CAchesize = Size = <size>,
	Align = <line size>
	(default:ca=s=8,a=20)

Show entries for :	
Forbid item	
Elimination of dead code	SYmbol_forbid = <symbol name>[,...]
Elimination of same code	SAMECode_forbid = <function name>[,...]
Memory allocation in	Absolute_forbid = <address> [+ <size>] [,...]



Category : Optimize

Show entries for : Optimize items

Configuration : Debug

Optimize : None

Eliminated size : 0x001E

□ Include profile :

□ Unify strings
□ Eliminate dead code
□ Reallocate registers
□ Eliminate same code
□ Optimize branches

Cache size :
Size : 0x0008
Line : 0x0020

Options Link/Library :

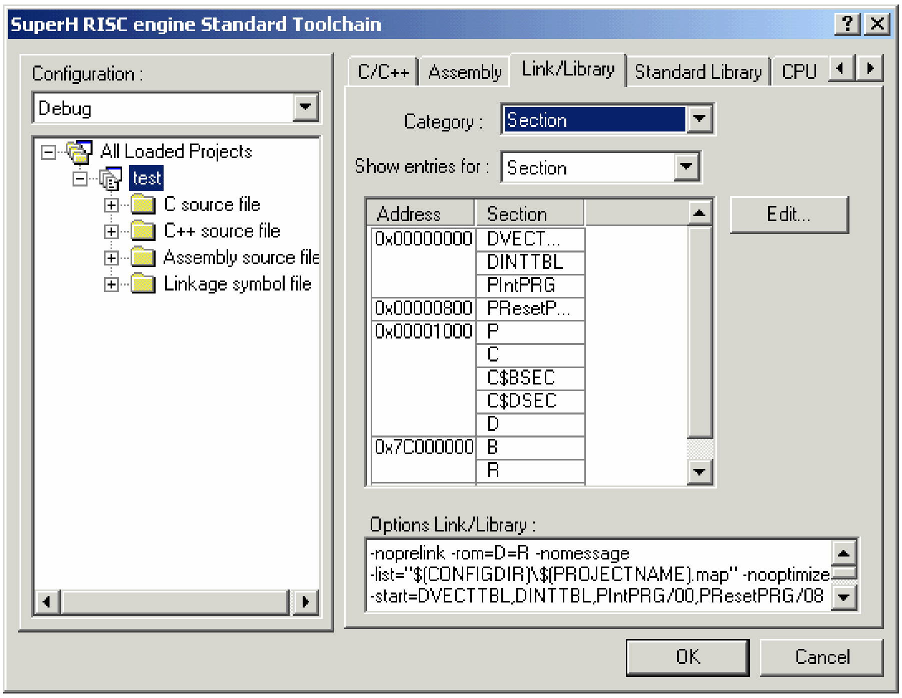
```
-noprelink -rom=D=R -nomessage
-list="$(CONFIGDIR)\$(PROJECTNAME).map" -nooptimize
-start=DVECTTBL,DINTTBL,PIntPRG/00,PResetPRG/08
```

Figure 4.21 Category:[Optimize] Dialog Box

(5) Category:[Section]

Table 4.19 Correspondence between Items on the Category:[Section] and Optimizing Linkage Editor Options

Dialog Box	Option
Show entries for :	
Section	STAR t= <sub>[,...] <sub> : <section name> [{: | ,} <section name>[,...]] [/<address>]
Symbol file	FSymbol = <section name>[,...]



Configuration : Debug

Category : Section

Show entries for : Section

Address	Section
0x00000000	DVECT...
	DINTTBL
	PIntPRG
0x00000800	PResetP...
0x00001000	P
	C
	C$BSEC
	C$DSEC
	D
0x7C000000	B
	R

Options Link/Library :

```
-noprelink -rom=D=R -nomessage
-list="$(CONFIGDIR)\$(PROJECTNAME).map" -nooptimize
-start=DVECTTBL,DINTTBL,PIntPRG/00,PResetPRG/08
```

Figure 4.22 Category:[Section] Dialog Box

An additional section can be specified with [Edit] button.
Specified section names and addresses can be added with [Add] button.
Already specified section names and addresses can be edited with [Modify] button.
Multiple sections can be allocated to the same address with [New Overlay] button.
Already specified sections can be removed with [Remove] button.
The order of sections can be altered with [UP] and/or [DOWN] button.

If the contents of the dialog box on the previous page are written to the subcommand file of the Linkage Editor:

```
START DVECTTBL, DINTTBL, PIntPRG
START PRestPRG/1000

START P,C,C$BSEC, C$DSEC,D/1000
START RAM_sct1:RAM_sct2/F00000
 START B,R/7F000000
 START Stack/7FFFFBF0
```

These are shown in the figure 4.22.

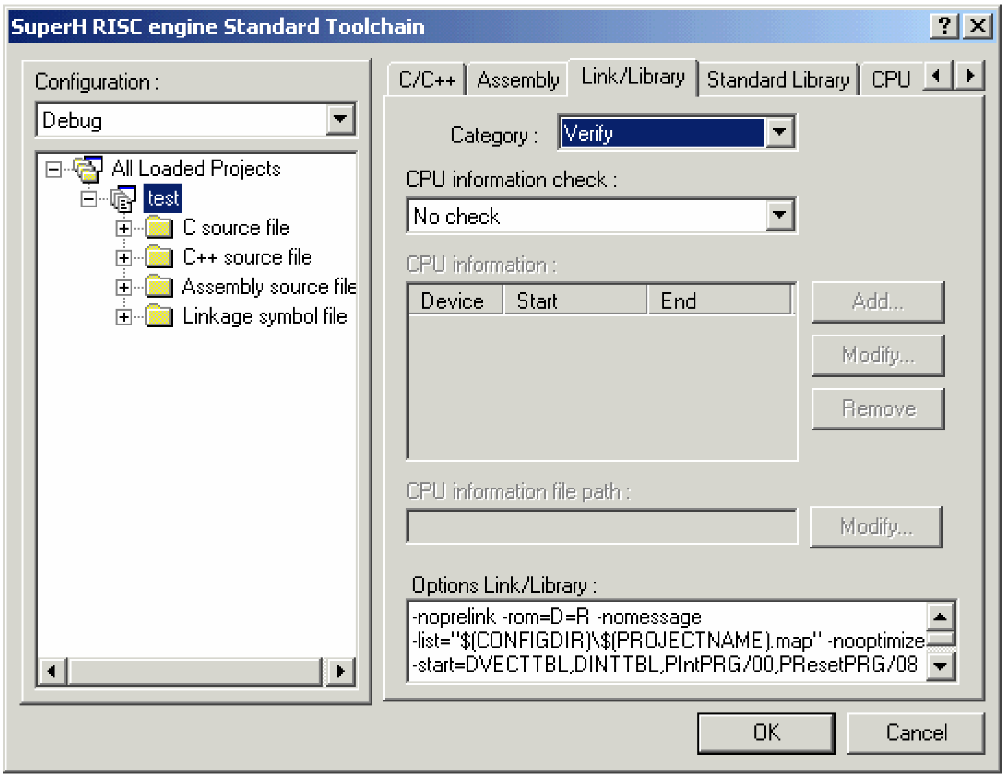
RAM_sct1 and RAM_sct2 are allocated to the same section.

Note: For details about creating the subcommand file for the Linkage Editor, refer to the SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual.

(6) Category:[Verify]

Table 4.20 Correspondence between Items on the Category:[Verify] and Optimizing Linkage Editor Options

Dialog Box	Option
CPU information check :	
No check	-
Check	CPu = {<cpu information file name> | {<memory type>} = <address range>[,…]} <memory type> : {ROm | Ram | XROm | XRAm | YROm | YRAm } <address range> : <start address> - <end address>
Use CPU information file	CPu = <cpu information file name>



Configuration : Debug

Category : Verify

CPU information check : No check

Options Link/Library :

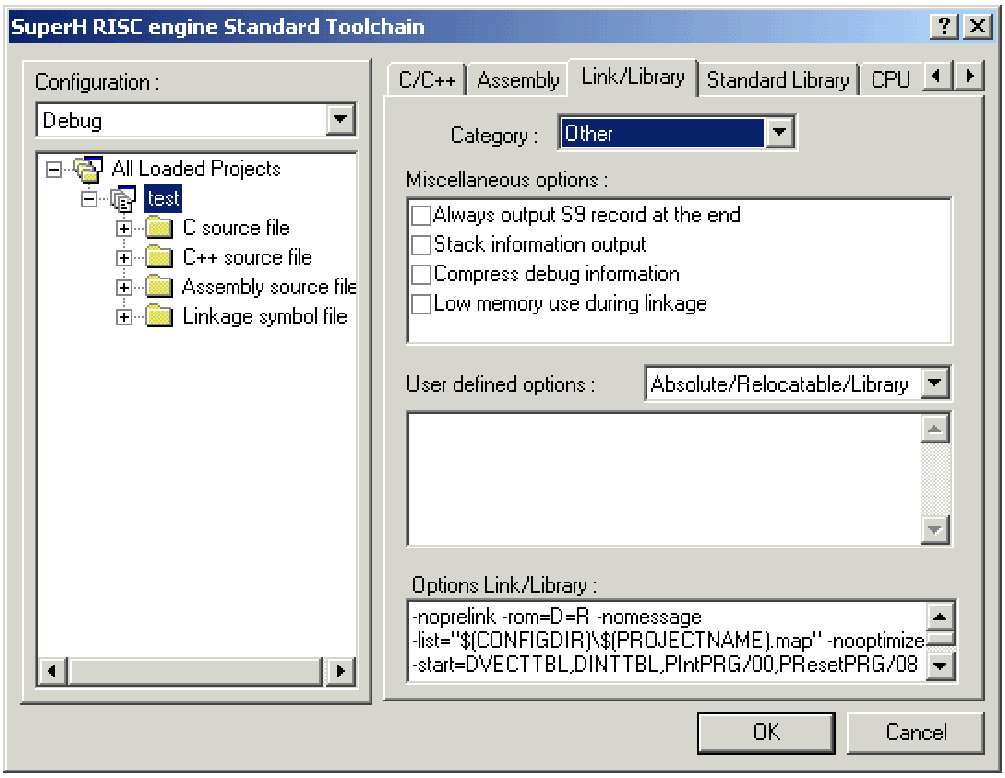
```
-noprelink -rom=D=R -nomessage
-list="$(CONFIGDIR)\$(PROJECTNAME).map" -nooptimize
-start=DVECTTBL,DINTTBL,PIntPRG/00,PResetPRG/08
```

Figure 4.23 Category:[Verify] Dialog Box

(7) Category: [Other]

Table 4.21 Correspondence between Items on the Category:[Other] and Optimizing Linkage Editor Options

Dialog Box	Option
Miscellaneous options :	
Always output S9 record at the end	S9
Stack information output	STACk
Compress debug information	COmpress / NOCOmpress
Low memory use during linkage	MEMory = [High | Low]
User defined options :	
Absolute/Relocatable/Library	
Hex/SType/Binary	



Configuration : Debug

Category : Other

Miscellaneous options :

- □ Always output S9 record at the end
- □ Stack information output
- □ Compress debug information
- □ Low memory use during linkage

User defined options : Absolute/Relocatable/Library

Options Link/Library :

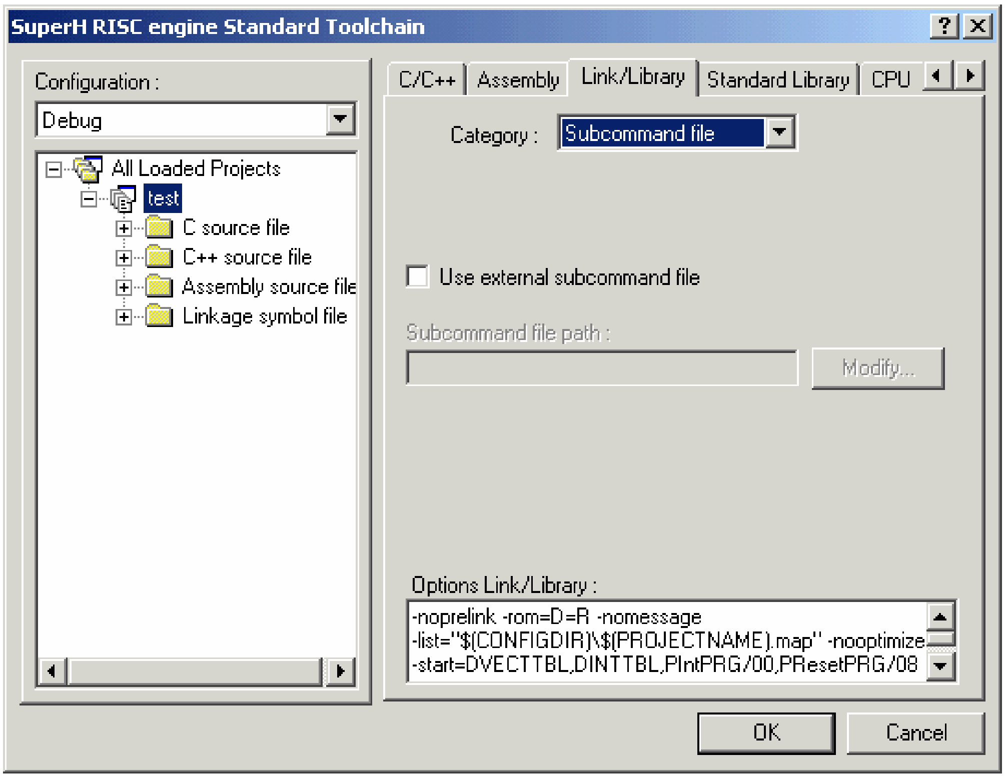
```
-noprelink -rom=D=R -nomessage
-list="$(CONFIGDIR)\$(PROJECTNAME).map" -nooptimize
-start=DVECTTBL,DINTTBL,PIntPRG/00,PResetPRG/08
```

Figure 4.24 Category:[Other] Dialog Box

(8) Category:[Subcommand file]

Table 4.22 Correspondence between Items on the Category:[Subcommnad file] and Optimizing Linkage Editor Options

Dialog Box	Option
Use external subcommand file	SUbcommand = <file name>



Configuration : Debug

Category : Subcommand file

□ Use external subcommand file

Subcommand file path :

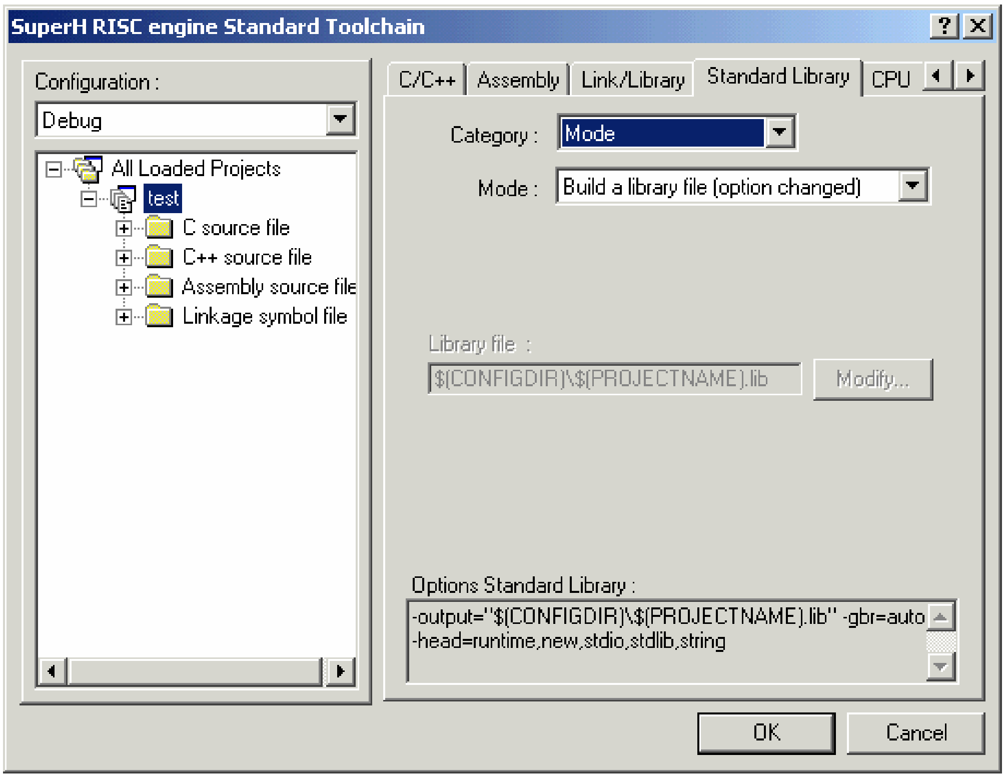
Options Link/Library :

```
-noprelink -rom=D=R -nomessage
-list="$(CONFIGDIR)\$(PROJECTNAME).map" -nooptimize
-start=DVECTTBL,DINTTBL,PIntPRG/00,PResetPRG/08
```

Figure 4.25 Category:[Subcommand file] Dialog Box

4.1.4 Standard Library Generator Options

Select the Standard Library tab from the SuperH RISC engine Standard Toolchain dialog box.



Configuration : Debug

Category : Mode

Mode : Build a library file (option changed)

Library file : $(CONFIGDIR)\$(PROJECTNAME).lib

Options Standard Library :

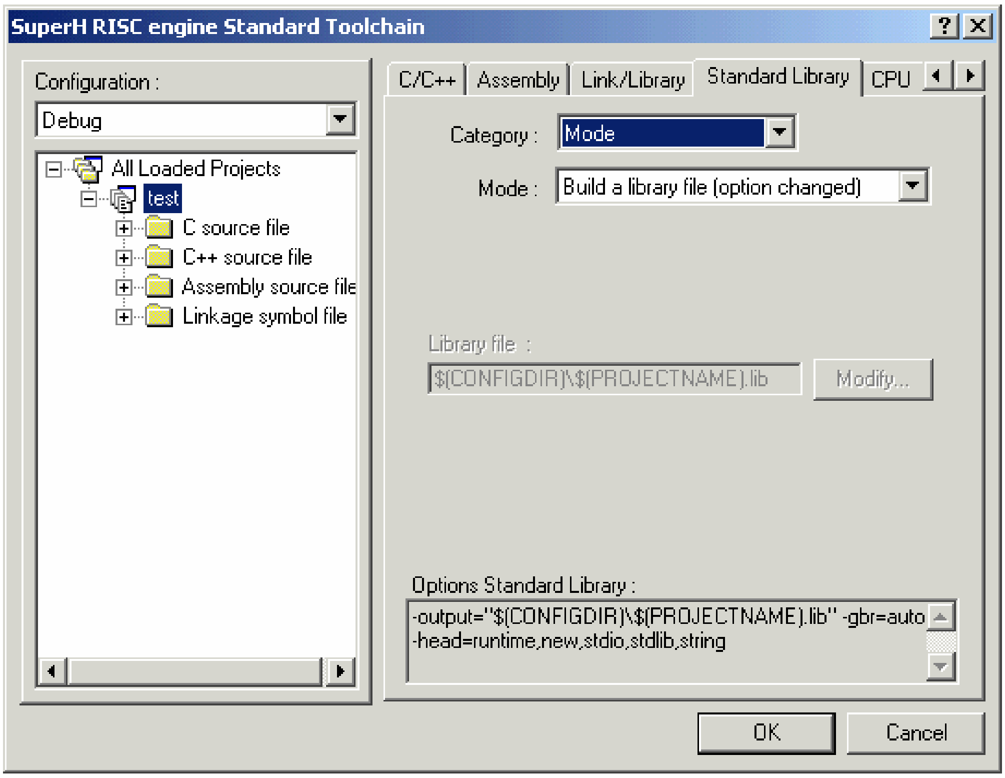
```
-output="$(CONFIGDIR)\$(PROJECTNAME).lib" -gbr=auto
-head=runtime,new,stdio,stdlib,string
```

Figure 4.26 Standard Library Tab Dialog Box

(1) Category:[Mode]

Table 4.23 Correspondence between Items on the Category:[Mode] and Functions

Dialog Box	Function
Mode :	
Build a library file(anytime)	Creates a current standard library
Build a library file(option changed)	Creates a current standard library when options have been changed.
Use an existing library file	Links an existing standard library.
Do not add a library file	Does not link a standard library



Configuration : Debug

Category : Mode

Mode : Build a library file (option changed)

Library file : $(CONFIGDIR)\$(PROJECTNAME).lib

Options Standard Library :

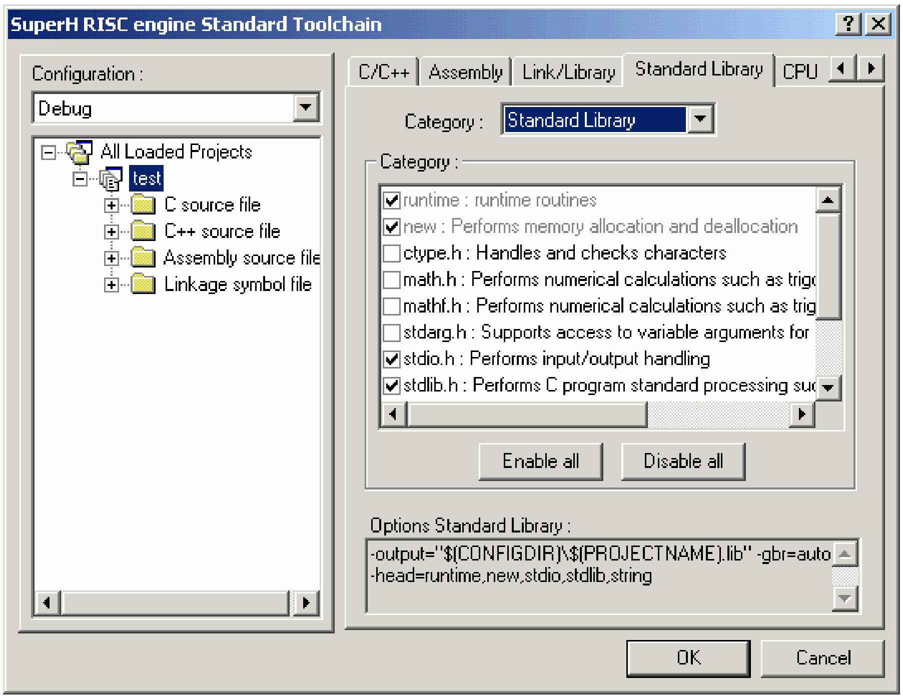
```
-output="$(CONFIGDIR)\$(PROJECTNAME).lib" -gbr=auto
-head=runtime,new,stdio,stdlib,string
```

Figure 4.27 Category:[Mode] Dialog Box

(2) Category:[Standard Library]

Table 4.24 Correspondence between Items on the Category:[Standard Library] and Standard Library Generator Options

Dialog Box	Option
Category :	Head = <sub>[,….]
runtime	<sub> : RUNTIME
new	<sub> : NEW
ctype.h	<sub> : CTYPE
math.h	<sub> : MATH
mathf.h	<sub> : MATHF
stdarg.h	<sub> : STDARG
stdio.h	<sub> : STDIO
stdlib.h	<sub> : STDLIB
string.h	<sub> : STRING
ios(EC++)	<sub> : IOS
complex(EC++)	<sub> : COMPREX
string(EC++)	<sub> : CPPSTRING



Configuration : Debug

Category : Standard Library

Category :

☑ runtime : runtime routines
☑ new : Performs memory allocation and deallocation
□ ctype.h : Handles and checks characters
□ math.h : Performs numerical calculations such as trig
□ mathf.h : Performs numerical calculations such as trig
□ stdarg.h : Supports access to variable arguments for
☑ stdio.h : Performs input/output handling
☑ stdlib.h : Performs C program standard processing suc

Options Standard Library :

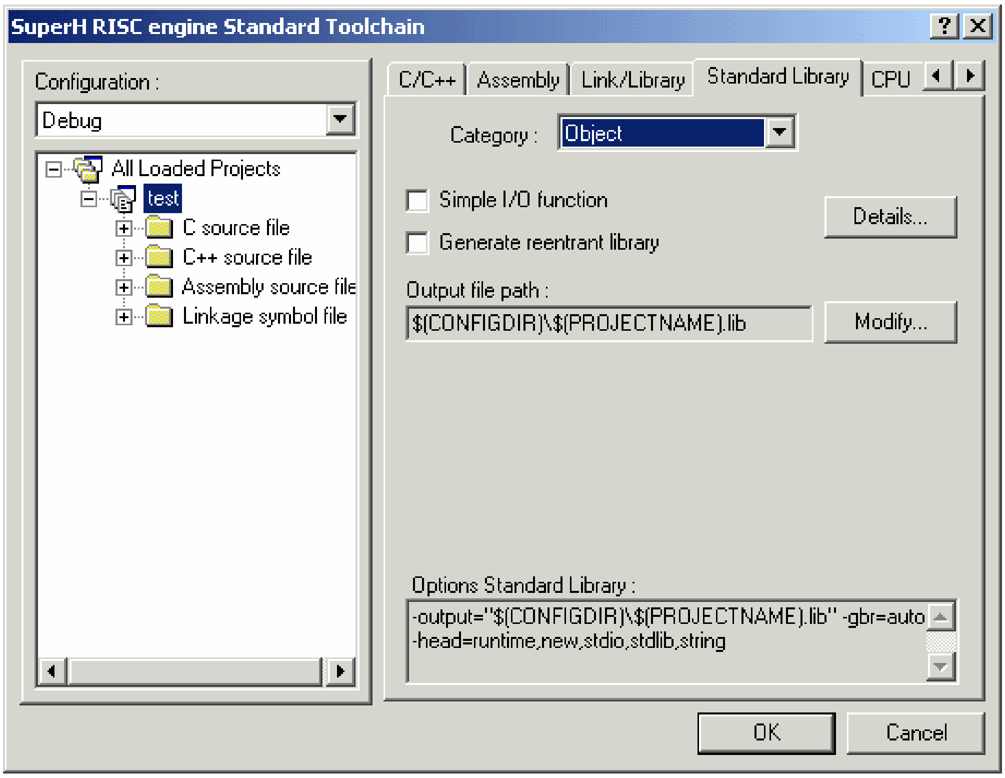
```
-output="$(CONFIGDIR)\$(PROJECTNAME).lib" -gbr=auto
-head=runtime,new,stdio,stdlib,string
```

Figure 4.28 Category:[Standard Library] Dialog Box

(3) Category:[object]

Table 4.25 Correspondence between Items on the Category:[Object] and Options

Dialog Box	Option
Simple I/O function	NOFLoat
Generate reentrant Library	REent



Category : Object

□ Simple I/O function

□ Generate reentrant library

Output file path : $(CONFIGDIR)\$(PROJECTNAME).lib

Options Standard Library :

```
-output="$(CONFIGDIR)\$(PROJECTNAME).lib" -gbr=auto
-head=runtime,new,stdio,stdlib,string
```

Figure 4.29 Category:[Object] Dialog Box

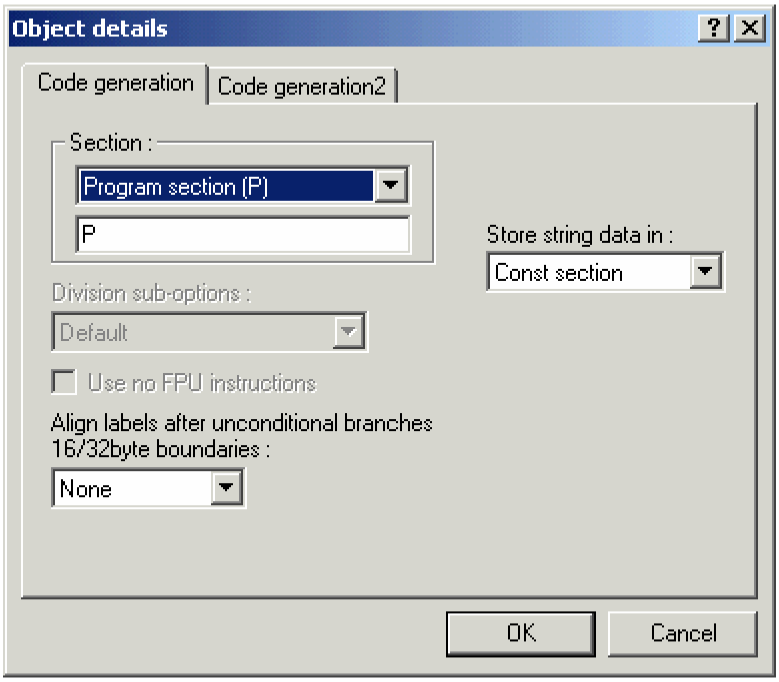
Clicking on [Details…] opens the "Optimize details" dialog box.

(a) Code generation tab

Table 4.26 Correspondence between Items on the Optimize details Dialog Box and Compiler Options

Dialog Box	Option
Section	SEction = <sub>[,…]
Program section(P)	<sub> : Program = <section name>
Const section(C)	<sub> : Const = <section name>
Data section(D)	<sub> : Data = <section name>
Uninitialized data section(B)	<sub> : Bss = <section name>
	Default: (p=P, c=C, d=D, b=B)
Store string data in :	STring = { Const | Data }
Division sub-options :	DIvision = Cpu [= { Inline | Runtime }]
Use no FPU instructions :	IFUnc
Align labels after unconditional branches	ALIGN16
16/32 byte boundaries :	ALIGN32
	NOALign

NOFLoat, REent : Standard Library Generator options
Others: Compiler options



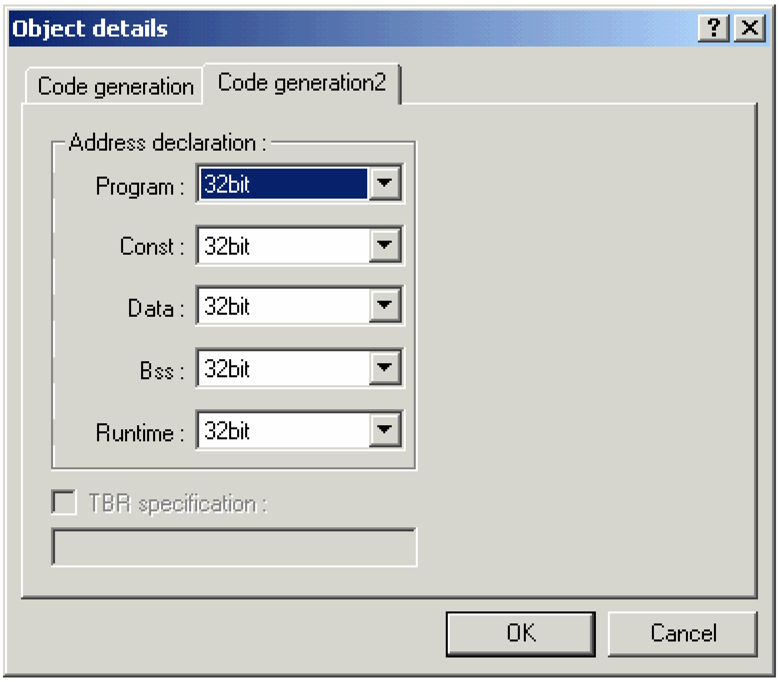
Section :
Program section (P)
P
Division sub-options :
Default
□ Use no FPU instructions
Align labels after unconditional branches 16/32byte boundaries :
None
Store string data in :
Const section

Figure 4.30 Category:[Object] Dialog Box

(b) Code generation2 tab

Table 4.27 Correspondence between Items on the Optimize details Dialog Box and Compiler Options

Dialog Box	Option
Address declaration :	<ABS> = <sub>[,…] <ABS> : { ABs16 | ABS20 | ABS28 | ABS32 } <sub> : { Program | Const | Data | Bss | Run | All }
TBR specification :	TBR [= <section name>]



Address declaration :

Program : 32bit

Const : 32bit

Data : 32bit

Bss : 32bit

Runtime : 32bit

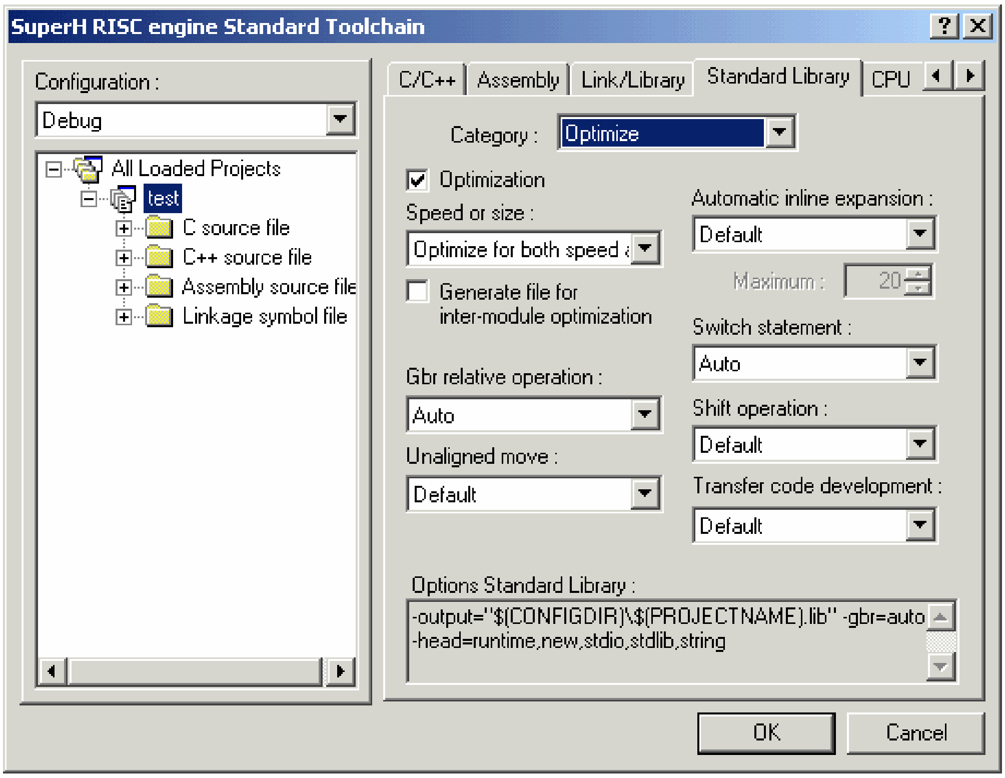
□ TBR specification :

Figure 4.31 Category:[Object] Dialog Box

(4) Category:[Optimize]

Table 4.28 Correspondence between Items on the Category [Optimize] and Compiler Options

Dialog Box	Option
Optimization	OPtimize = 1 / OPtimize = 0
Speed or size :	
Optimize for speed	SPeed
Optimize for size	SIze
Optimize for both speed and size	NOSPeed
Generate file for inter-module optimization	Goptimize
Gbr relative operation :	GBr = { Auto | User }
Unaligned move :	Unaligned = { Inline | Runtime }
Automatic inline expansion	INLine [= <data>] / NOINLine
Switch statement :	CAse = { Ifthen | Table }
Shift operation :	SHIft = { Inline | Runtime }
Transfer code development :	BLOckcopy = { Inline | Runtime }



Configuration : Debug

Category : Optimize

☑ Optimization

Speed or size : Optimize for both speed a

□ Generate file for inter-module optimization

Gbr relative operation : Auto

Unaligned move : Default

Automatic inline expansion : Default

Maximum : 20

Switch statement : Auto

Shift operation : Default

Transfer code development : Default

Options Standard Library :

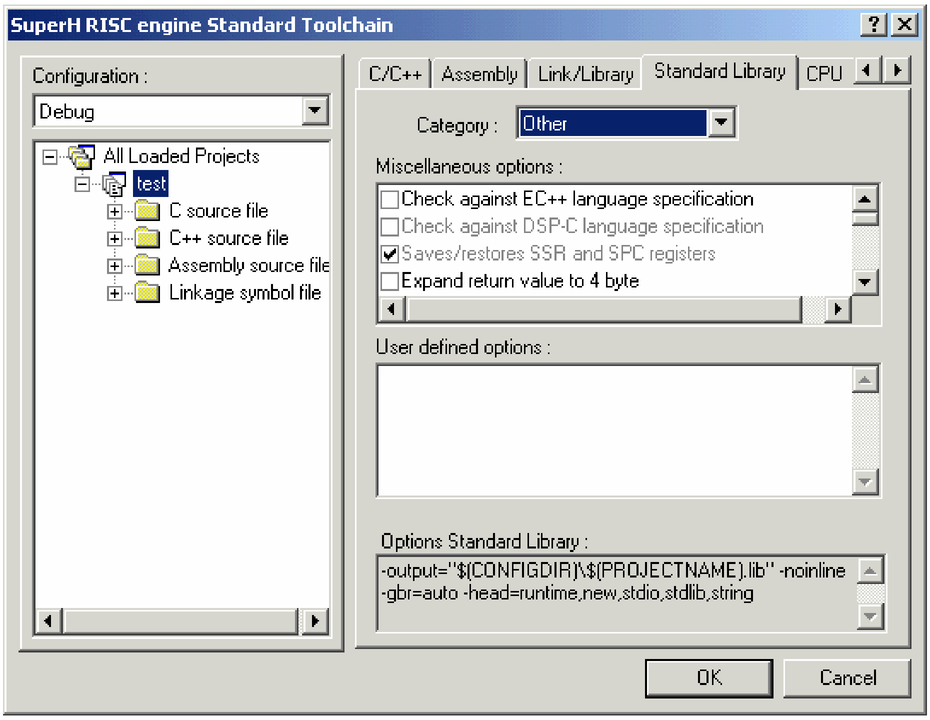
```
-output="$(CONFIGDIR)\$(PROJECTNAME).lib" -gbr=auto
-head=runtime,new,stdio,stdlib,string
```

Figure 4.32 Category:[Optimize] Dialog Box

(5) Category:[Other]

Table 4.29 Correspondence between Items on the Category:[Other] and Compiler Options

Dialog Box	Option
Miscellaneous options :	
Check against EC++ language specification	ECpp
Check against DSP-C language specification	DSpc
Saves/restores SSR and SPC registers	SAve_cont_reg = { 0 | 1 }
Expand return value to 4 bytes	RTnext/NORTnext
Loop unrolling	LOop/NOLOop
Approximate a floating-point constant division	APproxdiv
Avoid illegal SH7055 instructions	PAtch = 7055
Change FPSCR register if double data used	FPScr = Safe/FPScr = Aggressive
Treats loop condition as volatile qualified	Volatile_loop
Enum size is made the smallest	AUto_enum
Floating-point cnstant is handled as a fixed-point constant	FIXED_Const
Treats 1.0 as maximum number of fixed type	FIXED_Max
Delete type conversion sfter fixed multiple	FIXED_Noround
DSP repeat loop is used	REPeat
Enable register declaration	ENAble_register
Obey ANSI specifications more strictly	STRIct_ansi
Change integer division into floating-point	FDIv



Configuration : Debug

Category : Other

Miscellaneous options :

□ Check against EC++ language specification
□ Check against DSP-C language specification
☑ Saves/restores SSR and SPC registers
□ Expand return value to 4 byte

User defined options :

Options Standard Library :

-output="$(CONFIGDIR)\$(PROJECTNAME).lib" -noinline -gbr=auto -head=runtime,new,stdio,stdlib,string

Figure 4.33 Category:[Other] Dialog Box

4.1.5 CPU Options

Select the CPU tab from the SuperH RISC engine Standard Toolchain dialog box.

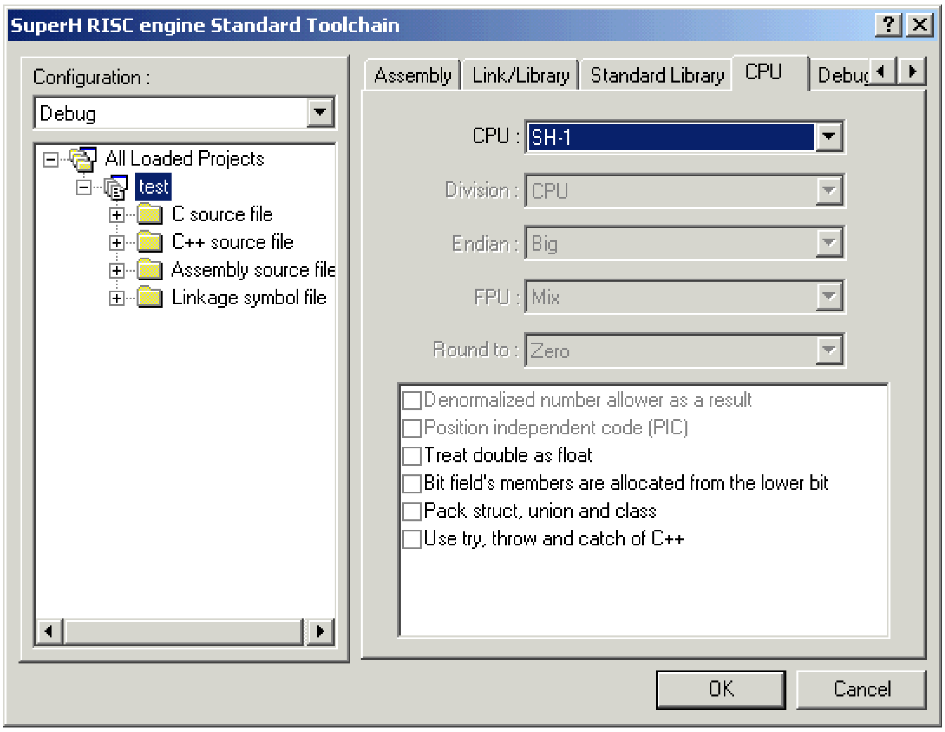
Table 4.30 Correspondence between Items on the [CPU] Tab and Compiler Options

Dialog Box	Option

CPU :	CPu = { SH1 | SH2 | SH2E | SH2A |
	SH2AFPU | SH2DSP |SH3 |
	SH3DSP | SH4 | SH4A |
	SH4ALDSP }

Division :	DIvision = { Cpu = [= { Inline | Runtion}] |
	Peripheral | Nomask }

Endian :	ENdian = { Big | Little }
FPU :	Fpu = { Single | Double }
Round to :	Round = { Zero | Nearest }
Denormalized number allower as a result	DENormalize = ON / DENormalize = OFF
Position independent code (PIC)	Pic = 1 / Pic = 0
Treat double as float	DOuble = Float
Bit field's members are allocated from the lower bit	Bit_order = { Left | Right }
Pack struct, union and class	PACK = 1 / PACK = 4
Use try,throw and catch of C++	EXception / NOEXception
Enable/disable runtime type information	RTTI = ON / RTTI = OFF



Configuration : Debug

CPU : SH-1

Division : CPU

Endian : Big

FPU : Mix

Round to : Zero

□ Denormalized number allower as a result
□ Position independent code (PIC)
□ Treat double as float
□ Bit field's members are allocated from the lower bit
□ Pack struct, union and class
□ Use try, throw and catch of C++

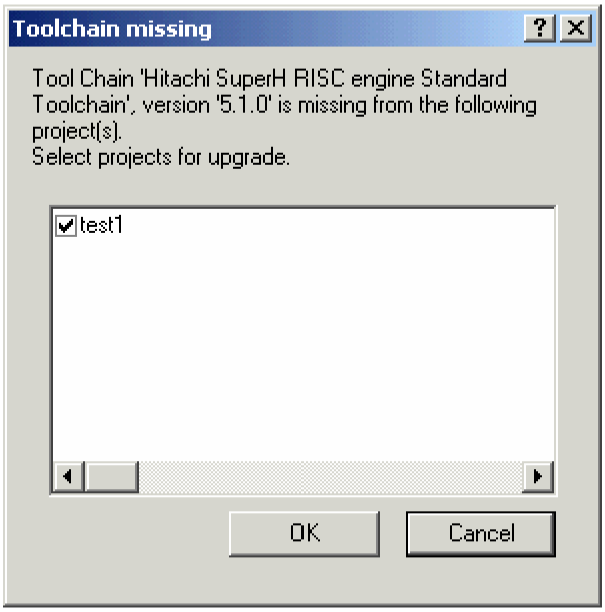
Figure 4.34 [CPU] Tab Dialog Box

4.2 Specifying the Compiler Version from the Renesas Integrated Development Environment

Here, the method for specifying the compiler version within the Renesas Integrated
Development Environment is explained. Compiler versions can be specified by upgrading the Renesas Integrated Development Environment.
If the workspace created in an old version (such as HEW1.1 or SH5.1B) is opened in a new version (such as HEW3.01 or SH9.0), the following dialog box appears.

(1) Checking the project to be upgraded.

Check the name of the project to be upgraded.



Toolchain missing

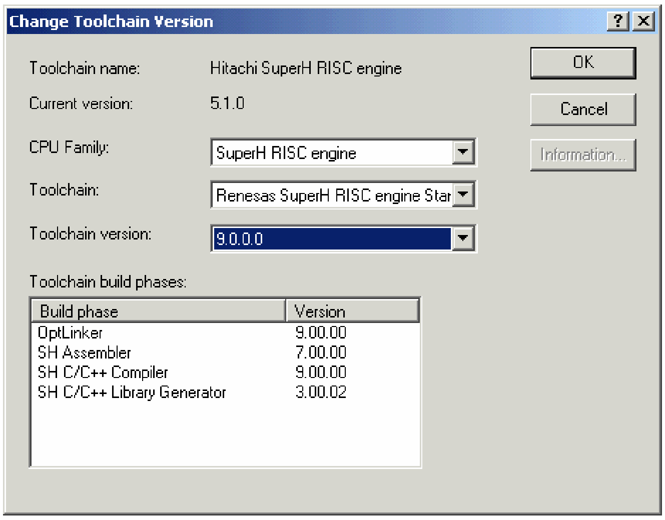
Tool Chain 'Hitachi SuperH RISC engine Standard Toolchain', version '5.1.0' is missing from the following project(s).
Select projects for upgrade.

☑ test1

Figure 4.35 High-performance Embedded Workshop

(2) Specifying the Compiler Version

Select the Compiler version which can be upgraded.



Change Toolchain Version

Toolchain name: Hitachi SuperH RISC engine

Current version: 5.1.0

CPU Family: SuperH RISC engine

Toolchain: Renesas SuperH RISC engine Star

Toolchain version: 9.0.0.0

Toolchain build phases:

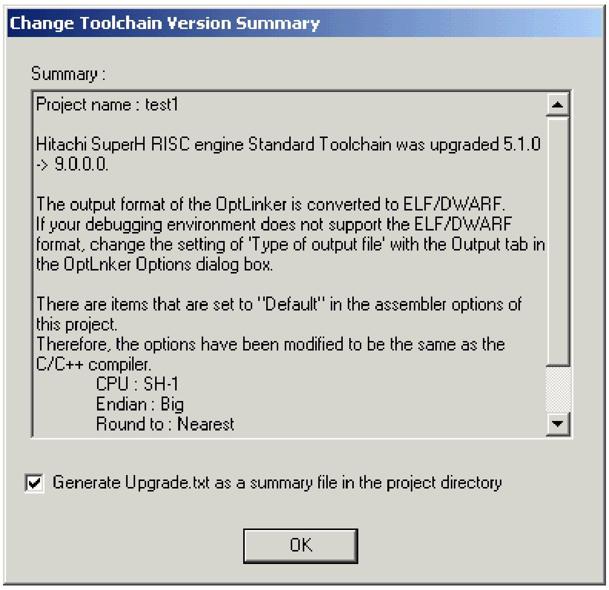
Build phase	Version
OptLinker	9.00.00
SH Assembler	7.00.00
SH C/C++ Compiler	9.00.00
SH C/C++ Library Generator	3.00.02

Figure 4.36 Change Toolchain Version Dialog Box

(3) Confirmation message

The C/C++ Compiler Ver7.1 and later versions support only the file format ELF/DWARF for the object to be output.

The file format is changed to ELF/DWARF format at upgrading. If the current debugging environment does not support the ELF/DWARF format, convert the ELF/DWARF format to the format supported by the debugging environment after upgrading.



Change Toolchain Version Summary

Summary :

Project name : test1

Hitachi SuperH RISC engine Standard Toolchain was upgraded 5.1.0 -> 9.0.0.0.

The output format of the OptLinker is converted to ELF/DWARF.
If your debugging environment does not support the ELF/DWARF format, change the setting of 'Type of output file' with the Output tab in the OptLnker Options dialog box.

There are items that are set to "Default" in the assembler options of this project.
Therefore, the options have been modified to be the same as the C/C++ compiler.
  CPU : SH-1
  Endian : Big
  Round to : Nearest

☑ Generate Upgrade.txt as a summary file in the project directory

Figure 4.37 Confirmation Message Dialog Log

Section 5 Efficient Programming Techniques

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

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

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

The SuperH RISC engine C/C++ compiler can be instructed to perform optimizations which emphasize speed of execution, by specifiying the "-speed" compile option.

The following are rules for efficient program creation.

(1) Rules for improving execution speed

Execution speed is determined by statements which are frequently executed and by complex statements. These should be found, and special efforts should be made to improve them.

(2) Rules for reducing program size

In order to shrink program size, similar processing should be performed using common code, and complex functions should be revised.

Because of compiler optimization, sometimes the actual execution speed differs from the theoretical speed. Various techniques should be utilized, checking performance by actually running the program with the compiler, in order to enhance performance.

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

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

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

The code size and execution speed values shown in this section were measured with the SH-1, SH-2, SH-2A, SH2A-FPU, SH-2E, SH2-DSP (SH7065), SH-3, SH3-DSP, SH-4, SH4A, and SH4AL-DSP. Table 5.1 shows the CPU options during compilation.

Table 5.1 List of CPU Options

No.	CPU Type	CPU Option
1	SH-1	-cpu=sh1
2	SH-2	-cpu=sh2
3	SH-2A	-cpu=sh2a
4	SH2A-FPU	-cpu=sh2afpu
5	SH-2E	-cpu=sh2e
6	SH-DSP(SH7065):	-cpu=sh2
7	SH-3	-cpu=sh3
8	SH3-DSP	-cpu=sh3
9	SH-4	-cpu=sh4Δ-fpu=single
10	SH-4A	-cpu=sh4Δ-fpu=single
11	SH4AL-DSP	-cpu=sh4aldsp

For the measuments with SH-2A, SH2A-FPU, SH3, SH3-DSP, SH-4, SH-4A, and SH4AL-DSP, cache misss are not considered except for some measurements. The number of external memory access cycle is assumed to be 1.

Table 5.2 lists Efficient Programming Techniques.

Table 5.2 List of Efficient Programming Techniques

No.	Function	ROM Efficiency	RAM Efficiency	Execution speed	Referenced Section
1	Local variables (data size)	○	-	○	5.1.1
2	Global variables(sign)	○	-	○	5.1.2
3	Data size (multiplication)	○	-	○	5.1.3
4	Data structures	○	-	○	5.1.4
5	Data alignment	-	○	-	5.1.5
6	Initial values and the const type	-	○	-	5.1.6
7	Local variables and global variables	○	-	○	5.1.7
8	Use of pointer variables	○	-	○	5.1.8
9	Referencing constants (1)	○	-	-	5.1.9
10	Referencing constants (2)	○	-	-	5.1.10
11	Variables which remain constant (1)	-	-	-	5.1.11
12	Variables which remains constant (2)	-	-	-	5.1.12
13	Incorporation of functions in modules	○	-	○	5.2.1
14	Calling functions using pointer varialbles	○	-	○	5.2.2
15	Function interface	-	○	○	5.2.3
16	Tail recursion	○	-	○	5.2.4
17	Using the FSQRT and FABS Instructions	○	-	○	5.2.5
18	Movement of invariant expressions within loops	-	-	○	5.3.1
19	Reducing the number of loops	x	-	○	5.3.2
20	Use of multiplication and division	-	-	-	5.3.3
21	Application of identities	-	-	○	5.3.4
22	Use of tables	○	-	○	5.3.5
23	Conditionals	○	-	○	5.3.6
24	Eliminating load/store instruction	○	-	○	5.3.7
25	Branching	○	-	○	5.4
26	Inline expansion of functions	x	-	○	5.5.1
27	Inline expansion with embeded assembly-language code	-	-	○	5.5.2
28	Offset Reference Using the Global Base Register (GBR)	○	-	○	5.6.1
29	Selective Use of Global Base Register (GBR) Area	○	-	○	5.6.2
30	Control of registe save/restore opetation	○	-	○	5.7
31	Specification using two-byte addresses	○	-	-	5.8
32	Prefetch instruction	-	-	○	5.9.1
33	Tiling	x	-	○	5.9.2
34	Matrix operations	○	-	○	5.10
35	Software pipelines	-	-	○	5.11

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

5.1 Data Specification

Table 5.3 lists data-related matters that should be considered.

Table 5.3 Suggestions for Data Specification

Area	Suggestion	Referenced Sections
Data type specifiers,type modifiers	If an attempt is made to reduce data sizes, the program size may increase as a result. Data types should be declared according to their use. Program size may change depending on whether signed or unsigned types are used; care should be taken in selecting data types. In the case of initialization data the values of which do not change within the program, using the const operator will reduce memory requirements.	5.1.1 to 5.1.3, 5.1.6
Data adjustment	Data should be allocated such that unused areas do not appear in memory.	5.1.5
Definition and referencing of structures	In some cases, data which is frequently referenced or modified can be incorporated into structures and pointer variables used to reduce program size. Bit fields can be used to reduce data size.	5.1.4
Local variables and global variables	Local variables are more efficient; all variables which can be used as local variables should be declared as local variables, rather than as global variables.	5.1.7
Use of pointer types	Programs which use array types should be modified to use pointer types.	5.1.8
Use of internal ROM/RAM	Since Internal memory is accessed more rapidly than external memory common variables should be stored in internal memory.	-

5.1.1 Local Variable (Data Size)

Important Points:

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

Description:

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

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

Example of Use:

To calculate the sum of the integers from 1 to 10:

Code before optimization

```
int f (void)
{
     char a = 10;
   int  c = 0;

     for ( ; a > 0; a-- )
            c += a;

     return(c);
}
```

Expanded into assembly language code (before optimization)

```
_f:
          MOV           #10,R4
          MOV           #0,R5
L217:
          EXTS.B        R4,R3
          ADD           R3,R5
          ADD           #-1,R4
          EXTS.B        R4,R2
          CMP/PL        R2
          BT            L217
          RTS
          MOV           R5,R0
```

Code after optimization

```
int f(void)
{
     long a = 10;
     int  c = 0;

     for ( ; a > 0; a-- )
            c += a;

     return(c);
}
```

Expanded into assembly language code (after optimization)

```
_f:
          MOV           #10,R4
          MOV           #0,R5
L217:
          ADD           R4,R5
          ADD           #-1,R4
          CMP/PL        R4
          BT            L217
          RTS
          MOV           R5,R0
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	18	16	73	63
SH-2	18	16	73	63
SH-2A	16	14	62	52
SH2AL-FPU	16	14	62	52
SH-2E	18	16	73	63
SH2-DSP(SH7065)	18	16	73	63
SH-3	18	16	73	63
SH3-DSP	18	16	73	63
SH-4	18	16	64	54
SH-4A	18	16	54	44
SH4AL-DSP	18	16	54	44

5.1.2 Global Variables (Signs)

Important Points:

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

Description:

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

Example of Use:

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

Code before optimization

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

Expanded into assembly language code (before optimization)

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

Code after optimization

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

Expanded into assembly language code (after optimization)

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

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	32	28	15	11
SH-2	32	28	15	11
SH-2A	32	28	8	8
SH2A-FPU	32	28	15	11
SH-2E	32	28	15	11
SH2-DSP(SH7065)	32	28	15	11
SH-3	32	28	15	11
SH3-DSP	32	28	15	13
SH-4	32	28	13	9
SH-4A	32	28	10	8
SH4AL-DSP	32	28	10	8

5.1.3 Data Size (Multiplication)

Important Points:

In multiplication, when the multiplier or multiplicand is declared as an [unsigned] char or an [unsigned] short, execution speed is improved.

Description:

In the SH-2, SH-2E, SH2-DSP,SH-3, SH-3DSP, SH-4, SH2A, SH2AL-DSP, and SH4AL-DSP when the multiplier and multiplicand in multiplication are one or two bytes, the operation is expanded into MULS.W or MULU.W instructions; but when either is four bytes, a MUL.L instruction is used.

In the case of the SH-1, when multiplier and multiplicand are one or two bytes, a MULS or MULU instruction is used; but if they are four bytes, the run-time library is called.

Example of Use:

To take the product of the variables a and b, and return the result:

Note: In this example, the compile option is –cpu=sh1.

Code before optimization

```
int f( long a, long b )
{
   return( a * b );
}
```

Expanded into assembly language code (before optimization)

```
_f:
          MOV.L       L11,R2
          MOV         R5,R1
          JMP         @R2
          MOV         R4,R0
L11:
          .DATA.L     _muli
```

Code after optimization

```
int f( short a, short b )
{
   return( a * b );
}
```

Expanded into assembly language code (after optimization)

```
_f:
          STS.L         MACL,@-R15
          MULS          R5,R4
          STS           MACL,R0
          RTS
          LDS.L         @R15+,MACL
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	12	10	23	8

Note: a=1, b=2

5.1.4 Data Structures

Important Points:

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

Description:

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

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

Example of Use:

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

Code before optimization

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

Expanded into assembly language code (before optimization)

```
_f:
          MOV.L       L11,R7
          MOV         #1,R1
          MOV.L       R1,@R7
          MOV.L       L11+4,R1
          MOV.L       L11+8,R2
          MOV         #2,R4
          MOV         #3,R5
          MOV.L       R4,@R1
          RTS
          MOV.L       R5,@R2
L11:
          .DATA.L     _a
          .DATA.L     _b
          .DATA.L     _c
```

Code after optimization

```
struct s{
    int a;
    int b;
    int c;
} s1;
void f(void)
{
    register struct s *p=&s1;

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

Expanded into assembly language code (after optimization)

```
_f:
          MOV.L       L11,R2
          MOV         #1,R1
          MOV         #2,R4
          MOV         #3,R5
          MOV.L       R1,@R2
          MOV.L       R4,@(4,R2)
          RTS
          MOV.L       R5,@(8,R2)
L11:
          .DATA.L     _s1
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	32	20	12	9
SH-2	32	20	12	9
SH-2A	32	20	9	6
SH2A-FPU	32	20	9	6
SH-2E	32	20	12	9
SH2-DSP(SH7065)	32	20	12	9
SH-3	32	20	14	10
SH3-DSP	32	20	15	11
SH-4	32	20	8	7
SH-4A	32	20	10	8
SH4AL-DSP	32	20	10	8

5.1.5 Data Alignment

Important Points:

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

Description:

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

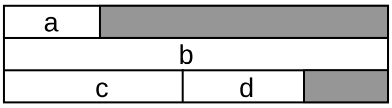
Example of Use:

To declare data totaling eight bytes:

Code before optimization

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

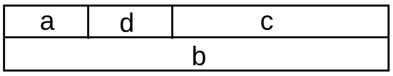
Data arrangement before optimization



Code after optimization

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

Data arrangement after optimization



5.1.6 Initial Values and the Const Type

Important Points:

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

Description:

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

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

Example of Use:

To specify five pieces of initialization data:

Code before optimization	Code after optimization

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

Initial value is transferred from ROM to RAM before processing.	Initial value stored in ROM is used for processing.

5.1.7 Local Variables and Global Variables

Important Points:

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

Description:

Variables which can be used as local variables should always be declared as local variables, never as global variables. The value of a global variable may change as the result of a function call or a pointer operation, and so global variables are not subject to global optimization.

Use of local variables has the following advantages.

a. Low access cost

b. The possibility of register allocation

c. Optimization

Example of Use:

To perform ten loop repetitions:

Code before optimization

```
int i;

void f(void)
{
   for ( i = 0; i < 10; i++ );
}
```

Expanded into assembly language code (before optimization)

```
_f:
          MOV.L         L218+2,R4
          MOV           #0,R3
          MOV           #10,R5
          BRA           L216
          MOV.L         R3,@R4
L217:
          MOV.L         @R4,R1
          ADD           #1,R1
          MOV.L         R1,@R4
L216:
          MOV.L         @R4,R3
          CMP/GE        R5,R3
          BF            L217
          RTS
          NOP
L218:
          .DATA.W    0
          .DATA.L    _i
```

Code after optimization

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

   for ( i = 0; i < 10; i++ );
}
```

Expanded into assembly language code (after optimization)

```
_f:
          MOV          #10,R4
L216:
          DT           R4
          BF           L216
          RTS
          NOP
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	20	12	54	52
SH-2	20	10	45	42
SH-2A	20	8	42	42
SH2A-FPU	20	8	42	42
SH-2E	20	10	45	42
SH2-DSP(SH7065)	20	10	54	51
SH-3	20	10	45	42
SH3-DSP	20	10	53	51
SH-4	20	10	34	33
SH-4A	20	10	25	23
SH4AL-DSP	20	10	50	46

5.1.8 Use of Pointer Variables

Important Points:

In some cases, rewriting a program which uses arrays into that uses pointer types can improve execution speed. (Ver.6)

Description:

In referencing an array element a[i], code is generated which adds the address of the ith element to the address of a[0]. By using pointer variables, the number of variables and operations can sometimes be reduced.

Example of Use:

To calculate the total for an array:

Code before optimization

```
int f1( int data[], int count )
{
         int ret = 0, i;

         for (i = 0; i < count; i++)
                ret += data[i]*i;
         return ret;
}
```

Expanded into assembly language code
(before optimization)

```
_f1:
          STS.L         MACL,@-R15
          MOV           #0,R7
          CMP/PL        R5
          BF/S          L219
          MOV           R7,R6
L220:
          MOV           R6,R0
          SHLL2         R0
          MOV.L         @(R0,R4),R3
          MUL.L         R6,R3
          ADD           #1,R6
          STS           MACL,R3
          CMP/GE        R5,R6
          BF/S          L220
          ADD           R3,R7
L219:
          MOV           R7,R0
          RTS
          LDS.L         @R15+,MACL
```

Code after optimization

```
int f2( int *data, int count )
{
        int ret = 0, i;

        for (i = 0; i < count; i++)
                ret += *data++ *i;
         return ret;
}
```

Expanded into assembly language code
(after optimization)

```
_f2:
          STS.L         MACL,@-R15
          MOV           #0,R7
          CMP/PL        R5
          BF/S          L221
          MOV           R7,R6
L222:
          MOV.L         @R4+,R3
          MUL.L         R6,R3
          ADD           #1,R6
          STS           MACL,R3
          CMP/GE        R5,R6
          BF/S          L222
          ADD           R3,R7
L221:
          MOV           R7,R0
          RTS
          LDS.L         @R15+,MACL
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	40	36	318	318
SH-2	34	30	179	159
SH-2E	34	30	178	158
SH2-DSP(SH7065)	34	30	207	187
SH-3	34	30	149	129
SH3-DSP	34	30	168	148
SH-4	34	30	117	97

Note: The number of cycles is for count=10.

5.1.9 Referencing Constants (1)

Important Points:

Code size can be reduced by using a single byte to represent immediate values wherever possible.

Description:

When a single-byte immediate value is used, it is embedded in the code. On the other hand, two-byte and four-byte immediate values are placed in memory and then accessed.

Example of Use:

To substitute an immediate value into the variable:

Source code (1)

```
int i;
void f(void)
{
     i = 0x10000;
}
```

Expanded into assembly language code (1)

```
_f:
          MOV          #1,R2
          MOV.L        L12+2,R6
          SHLL16       R2
          RTS
          MOV.L        R2,@R6
L12:
          .RES.W       1
          .DATA.L     _i
```

Source code (2)

```
int i;
void f(void)
{
     i = 0x01;
}
```

Expanded into assembly language code (2)

```
_f:
          MOV.L        L12+2,R6
          MOV          #1,R2
          RTS
          MOV.L        R2,@R6
L12:
          .RES.W       1
          .DATA.L     _i
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Source code (1)	Source code (2)	Source code (1)	Source code (2)
SH-1	16	12	7	5
SH-2	16	12	7	5
SH-2A	16	12	4	4
SH2AL-FPU	16	12	4	4
SH-2E	16	12	7	5
SH2-DSP(SH7065)	16	12	7	6
SH-3	16	12	7	5
SH3-DSP	16	12	6	6
SH-4	16	12	5	4
SH-4A	16	12	5	4
SH4AL-DSP	16	12	5	4

5.1.10 Referencing Constants (2)

Important Points:

Expressions using constants may be combined without an increase in the size of the code generated.

Description:

The constant convolution feature is available. Even if a constant is represented as an expression, it is calculated at compilation and is not reflected in the code generated.

Example of Use:

To substitute a constant for the variable a:

Code before optimization

```
#define MASK1  0x1000
#define MASK2  0x10

int a = 0xffffffff;

void f(void)
{
      int x;

    x = MASK1;
    x |= MASK2;
    a &= x;
}
```

Expanded into assembly language code
(before optimization)

```
_f:
          MOV.W         L217,R4
          MOV.L         L217+4,R5
          MOV.L         @R5,R3
          AND           R4,R3
          RTS
          MOV.L          R3,@R5
L217:
          .DATA.W       H'1010
          .DATA.W      0
          .DATA.L       _a
```

Code after optimization

```
#define MASK1  0x1000
#define MASK2  0x10

int a = 0xffffffff;

void f(void)
{
      a &= MASK1 | MASK2;
}
```

Expanded into assembly language code
(after optimization)

```
_f:
          MOV.L         L216+4,R4
          MOV.W         L216,R3
          MOV.L         @R4,R2
          AND           R3,R2
          RTS
          MOV.L          R2,@R4
L216:
          .DATA.W       H'1010
          .DATA.W      0
          .DATA.L       _a
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	20	20	9	9
SH-2	20	20	9	9
SH-2A	20	20	7	7
SH2A-FPU	20	20	7	7
SH-2E	20	20	9	9
SH2-DSP(SH7065)	20	20	9	9
SH-3	20	20	9	9
SH3-DSP	20	20	9	9
SH-4	20	20	8	8
SH-4A	20	20	6	6
SH4AL-DSP	20	20	6	6

5.1.11 Variables Which Remain Constant (1)

Important Points:

When the value of a variable remains constant, it is treated as a constant; there is no effect on memory efficiency or execution speed even if the variable is not calculated in advance.

Description:

The constant convolution feature applies to variables which behave as constants also, tracing the value of the variable and performing constant calculations. Hence there is no increase in generated code size even if the source code is written so as to be easily readable.

Example of Use:

To change a return value according to the result for the variable rc:

```
Calculate the variable value in      Have the C compiler calculate the
advance                              value
Source code (1)                      Source code (2)

#define ERR       -1                 #define ERR       -1
#define NORMAL     0                 #define NORMAL     0

int f(void)                          int f(void)
{                                    {
     int rc, code;                        int rc, code;

     rc = 0;                              rc = 0;
     code = NORMAL;                       if ( rc ) code = ERR;
     return( code );                          else  code = NORMAL;
}                                         return( code );
                                     }

Expanded into assembly language code (1)   Expanded into assembly language code (2)

_f:                                  _f:
          RTS                                  RTS
          MOV         #0,R0                    MOV         #0,R0
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Source code (1)	Source code (2)	Source code (1)	Source code (2)
SH-1	4	4	3	3
SH-2	4	4	3	3
SH-2A	4	4	4	4
SH2A-FPU	4	4	4	4
SH-2E	4	4	3	3
SH2-DSP(SH7065)	4	4	3	3
SH-3	4	4	3	3
SH3-DSP	4	4	3	3
SH-4	4	4	3	3
SH-4A	4	4	2	2
SH4AL-DSP	4	4	2	2

5.1.12 Variables Which Remain Constant (2)

Important Points:

When the value of a variable remains constant, it is treated as a constant; there is no effect on memory efficiency or execution speed even if the variable is not calculated in advance.

Description:

The constant convolution feature applies to variables which behave as constants also, tracing the value of the variable and performing constant calculations. Hence there is no increase in generated code size even if the source code is written so as to be easily readable.

Example of Use:

To calculate the product of the variables a and c, and substitute the result into the variable b.

Calculate the variable value in advance

Source code (1)

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

     a = 3;
     b = 15;
     return b;
}
```

Expanded into assembly language code (1)

```
_f:
          RTS
          MOV          #15,R0
```

Have the C compiler calculate the value

Source code (2)

```
int f(void)
{
     int a, b, c;

     a = 3;
     c = 5;
     b = c * a;
     return b;
}
```

Expanded into assembly language code (2)

```
_f:
          RTS
          MOV          #15,R0
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Source code (1)	Source code (2)	Source code (1)	Source code (2)
SH-1	4	4	3	3
SH-2	4	4	3	3
SH-2A	4	4	4	4
SH2A-FPU	4	4	4	4
SH-2E	4	4	3	3
SH2-DSP(SH7065)	4	4	3	3
SH-3	4	4	3	3
SH3-DSP	4	4	3	3
SH-4	4	4	3	3
SH-4A	4	4	2	2
SH4AL-DSP	4	4	2	2

5.2 Function Calls

Matters that should be considered when calling functions are listed in table 5.4.

Table 5.4 Suggestions Related to Function Calls

Area	Suggestion	Referenced Sections
Function position	Closely-related functions should be combined in a single file.	5.2.1
Interface	The number of parameters should be strictly limited (up to four) such that they are all allocated to registers. When there are a large number of parameters, they should be incorporated in a structure, and passed using pointers.	5.2.3
Function division	In some cases, various types of optimization are not performed effectively for extremely large functions. Using a feature called tail recursion, functions should be divided until they are sufficiently small that optimization can be performed effectively.	5.2.4
Replacement by macros	When a function is called frequently, it can be replaced by a macro to speed execution. However, the use of a macro increases program size, and so macros should be used according to the circumstances.	-

5.2.1 Incorporation of Functions in Modules

Important Points:

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

Description:

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

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

Example of Use:

To call the function g from the function f:

Code before optimization

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

Code after optimization

```
int g(void)
{
}

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

Expanded into assembly language code
(before optimization)

```
_f:
          MOV.L         L216+2,R3
          JMP           @R3
          NOP
L216:
          .DATA.W       0
          .DATA.L       _g
```

Expanded into assembly language code
(after optimization)

```
_g:
          RTS
          NOP
_f:
          BRA          _g
          NOP
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	12	4	8	6
SH-2	12	4	8	6
SH-2A	12	4	8	6
SH2A-FPU	12	4	8	6
SH-2E	12	4	8	6
SH2-DSP(SH7065)	12	4	9	6
SH-3	12	4	8	6
SH3-DSP	12	4	9	6
SH-4	12	4	8	5
SH-4A	12	4	5	4
SH4AL-DSP	12	4	5	4

Comments:

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

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

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

5.2.2 Calling Functions Using Pointer Variables

Important Points:

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

Description:

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

Example of Use:

To change the called function according to the value of the variable a:

Code before optimization

```
extern void nop(void);
extern void stop(void);
extern void play(void);

void f(int a)
{
     switch (a)
      {
     case 0:
                 nop(); break;
     case 1:
                  stop(); break;
     case 2:
                  play(); break;
      }
}
```

Expanded into assembly language code
(before optimization)

```
_f:
          MOV           R4,R0
          CMP/EQ        #0,R0
          BT            L220
          CMP/EQ        #1,R0
          BT            L221
          CMP/EQ        #2,R0
          BT            L222
          BRA           L223
          NOP
L220:
          MOV.L         L224,R3
          JMP           @R3
          NOP
L221:
          MOV.L         L224+4,R3
          JMP           @R3
          NOP
```

Code after optimization

```
extern void nop(void);
extern void stop(void);
extern void play(void);

static int (*key[3])() =
            {nop, stop, play};
void f(int a)
{
       (*key[a])();
}
```

Expanded into assembly language code
(after optimization)

```
_f:
          MOV.L       R4,@-R15
          MOV         R4,R3
          MOV.L       L241+2,R0
          SHLL2       R3
          MOV.L       @(R0,R3),R3
          JMP         @R3
          ADD         #4,R15
L241:
          .DATA.W     0
          .DATA.L     _$key
          .SECTION    D,DATA,ALIGN=4
_$key:
          .DATA.L     _nop,_stop,_play
```

```
L222:
          MOV.L         L224+8,R3
          JMP           @R3
          NOP

L223:
          RTS
          NOP
L224:
          .DATA.L    _nop
          .DATA.L    _stop
          .DATA.L    _play
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	52	16	15	11
SH-2	52	16	15	11
SH-2A	52	16	14	10
SH2A-FPU	52	16	14	10
SH-2E	52	16	15	11
SH2-DSP(SH7065)	52	16	16	12
SH-3	52	16	15	11
SH3-DSP	52	16	16	13
SH-4	52	16	13	10
SH-4A	52	16	10	8
SH4AL-DSP	52	16	10	8

5.2.3 Function Interface

Important Points:

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

For details, refer to section, 3.15.1 (2), Function calling interface.

Description:

Function parameters should be selected carefully such that all parameters are allocated to registers (up to four parameters). If numerous parameters must be used, they should be incorporated in a structure and passed using pointers. If all parameters fit into registers, function calls and processing at function entry and exit points are simplified. Stack use is also reduced.

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

In the SH-2E, SH-4, and SH-4A the floating point registers are used to handle floating point data. Registers FR0 to FR3 are work registers, FR4 to FR11 are for parameters, and FR12 to FR14 are for local variables.

Example of Use:

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

Code before optimization

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

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

Expanded into assembly language code
(before optimization)

```
_g:
          STS.L         PR,@-R15
          MOV           #5,R3
          MOV.L         L216+2,R2
          MOV           #4,R7
          MOV.L         R3,@-R15
          MOV           #3,R6
          MOV           #2,R5
          JSR           @R2
          MOV           #1,R4
          ADD           #4,R15
          LDS.L         @R15+,PR
          RTS
```

Code after optimization

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

int f(struct b *p);

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

Expanded into assembly language code
(after optimization)

```
_g:
          MOV.L         L217,R4
          MOV.L         L217+4,R3
          JMP           @R3
          NOP
L217:
          .DATA.L       _b1
          .DATA.L       _f
          NOP
L216:
          .DATA.W       0
          .DATA.L       _f
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	32	16	17	7
SH-2	32	16	20	10
SH-2A	28	16	17	9
SH2A-FPU	28	16	17	9
SH-2E	32	16	20	10
SH2-DSP(SH7065)	32	16	28	14
SH-3	32	16	22	10
SH3-DSP	32	16	25	15
SH-4	32	16	18	10
SH-4A	32	16	15	6
SH4AL-DSP	32	16	15	6

5.2.4 Tail Recursion

Important Points:

Execution speed does not suffer even if large functions are broken up into a series of smaller functions, with the next function being called at the end of the previous function.

Description:

When the function funk3() is called within the function funk2(), which is itself called by function funk1(), control is passed to function funk3() by a BRA or JMP instruction. Normally, after the completion of processing by function funk3(), an RTS instruction returns control to function funk2(), and when processing by function funk2() is completed, another RTS instruction returns control to function funk1(). (See to left side of figure 5.1.)

Here, when funk3() is called at the end of function funk2(), control is transferred to funk3() by a BSR or JSR instruction, and on completion of processing by funk3(), control can be returned directly to the function funk1() by an RTS instruction. (See right side of figure 5.1.) This feature is called tail recursion.

In some cases, various types of optimization are not performed effectively for extremely large modules. By making use of tail recursion, larger modules can be broken up into modules small enough that optimization is effective, for enhanced performance.

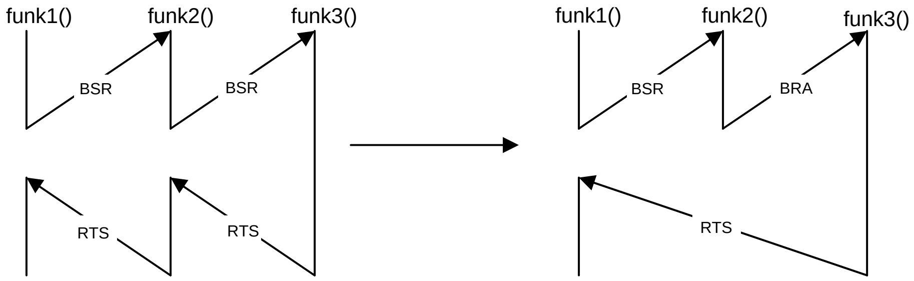


Figure 5.1 Tail Recursion

Example of Use:

To call the functions g and h from the function f: When returning from g and h, control is passed directly to the function which called f, bypassing f itself.

Source code before application (Ver.2.0)

```
void f(int x)
{
      if (x==1)
            g();
      else
            h();
}
```

Expanded into assembly language code (before application)

```
_f:
```

Source code after application (Ver.3.0 or later)

```
void f(int x)
{
      if (x==1)
            g();
      else
            h();
}
```

Expanded into assembly language code (after application)

```
_f:
```

```
          STS.L         PR,@-R15
          MOV           R4,R0
          CMP/EQ        #1,R0
          BF            L207
          BSR        _g
          NOP
          BRA            L208
          NOP
L207:
          BSR           _h
          NOP
L208:
          LDS.L          @R15+,PR
          RTS
          NOP
```

```
               MOV            R4,R0
               CMP/EQ         #1,R0
               BF             L12
               BRA            _g
            NOP
L12:
              BRA              _h
              NOP
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Application	After Application	Before Application	After Application
SH-2	24	14	14	8

Note: x = 2

5.2.5 Using the FSQRT and FABS Instructions

Important Points:

Instead of calling the mathematical functions sqrt and fabs from the library,

if targeting the SH-4, use the FSQRT and FABS instructions in the instruction sets for those processors.

Description:

The fabs (floating point absolute value) function is part of the mathematical function library; but the library is not necessary in programs without function addresses, and so a direct FABS instruction is used instead.

However, use of this instruction requires inclusion of <math.h> or of <mathf.h>.

If these are not included, the compiler calls fabs as an ordinary function, and the resulting library call detracts from performance.

There is no need for the user to define a macro.

<Macro example>

`#define fabs(a) ((a)>=0?0:(-(a)))` /* Not expanded into a FABS instruction */

Example of Use:

Below is shown the difference when <math.h> is not included (a library call results) and when it is included (a FABS instruction is used).

Note: In this example, the following compile option is used.
-cpu=sh4Δ-fpu=single

When using fabsf(), <mathf.h> must be included.

Code before optimization

```
float a,b;
f()
{
     :
     :
b=fabs(a);
   :
}
```

Expanded into assembly language code
(before optimization)

```
_f:
       STS.L      PR,@-R15
       MOV.L      L12,R6
       MOV.L      L12+4,R1
       JSR        @R1
       FMOV.S     @R6,FR4
       MOV        R0,R4
       LDS        R4,FPUL
```

Code after optimization

```
#include <math.h>
float a,b;
f()
{
     :
     :
b=fabs(a);
   :
}
```

Expanded into assembly language code
(after optimization)

```
_f:
         MOV.L      L12,R1
         MOV.L      L12+8,R4
         FMOV.S     @R1,FR9
         FABS       FR9
         RTS
         FMOV.S     FR9,@R4
_L12:
       FLOAT      FPUL,FR8
       MOV.L      L12+8,R5
       LDS.L      @R15+,PR
       RTS
       FMOV.S     FR8,@R5

L12:
       .DATA.L   _a
       .DATA.L   _fabs
       .DATA.L   _bW
```

```
       .DATA.L    _a
       .DATA.L    _fabs
       .DATA.L    _bW
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH2A-FPU	68	40	49	21
SH-2E	36	20	33	9
SH-4	36	20	29	8
SH-4A	36	20	22	6

5.3 Operations

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

Table 5.5 Suggestions Related to Operations

Area	Suggestion	Referenced Section
Unification, movement of invariant or common expressions	The possibility of substituting into the temporary variables of partial equations common to a function should be studied. Any invariant expressions within a for statement should be moved outside the for statement.	5.3.1
Reduction of number of loop iterations	The possibility of merging loop statements with conditions that are identical or similar should be studied. Try expanding loop statements.	5.3.2
Optimization of operations	Combine identical operations to reduce the number of operation iterations.	5.3.3
Use of identities	The possibility of using mathematical identities to reduce the number of operations should be studied.	5.3.4
Use of fast algorithms	The use of efficient algorithms requiring little processing time, such as quick sorts of an array, should be studied.	-
Utilization of tables	When processing for each case of a switch statement is nearly the same, the use of tables should be studied. Execution speed can sometimes be improved by performing operations in advance, storing the results in a table, and referring to values in the table when the operation results are needed. However, this method requires increased amounts of ROM, and so should be used with due attention paid to the balance between required execution speed and available ROM.	5.3.5
Conditionals	When making comparisons with a constant, if the value of the constant is 0, more efficient code is generated.	5.3.6
Load/store elimination	By eliminating memory access (load, store) instructions, the number of execution cycles can be reduced.	5.3.7

5.3.1 Movement of Invariant Expressions within Loop

Important Points:

When, in a loop, there is an expression the value of which does not change, execution speed can be improved by calculating the expression before the loop starts. (Ver.6)

Description:

By calculating the value of an expression which does not change within a loop prior to the start of the loop, calculations during each iteration can be omitted, and the number of execution instructions can be reduced.

Example of Use:

To substitute the array element b[5] into the array a[ ]:

Code before optimization

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

       j = 5;
       for ( i=0; i < 100; i++)
        a[i] = b[j];
}
```

Expanded into assembly language code
(before optimization)

```
_f:
          MOV.L       L240+4,R5
          MOV         R5,R4
          MOV.W       L240,R6
          ADD         R5,R6
          MOV.L       L240+8,R5
L239:
          MOV.L       @R5,R3
          MOV.L       R3,@R4
          ADD         #4,R4
          CMP/HS      R6,R4
          BF          L239
          RTS
          NOP
L240:
          .DATA.W     H'0190
          .DATA.W     0
          .DATA.L     _a
          .DATA.L     H'00000014+_b
```

Code after optimization

```
extern int a[100], b[100];
void f(void)
{
       int i,j,t;

       j = 5;
       for ( i=0, t=b[j];i < 100; i++)
        a[i] = t;
}
```

Expanded into assembly language code
(after optimization)

```
_f:
          MOV.L       L241+4,R5
          MOV.L       @R5,R5
          MOV.L       L241+8,R7
          MOV         R7,R4
          MOV.W       L241,R6
          ADD         R7,R6
L240:
          MOV.L       R5,@R4
          ADD         #4,R4
          CMP/HS      R6,R4
          BF           L240
          RTS
          NOP
L241:
          .DATA.W     H'0190
          .DATA.W     0
          .DATA.L     H'00000014+_b
          .DATA.L     _a
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	36	36	809	611
SH-2	36	36	809	611
SH-2E	36	36	809	611
SH2-DSP(SH7065)	36	36	908	611
SH-3	36	36	909	711
SH3-DSP	36	36	1008	711
SH-4	36	36	608	407

5.3.2 Reducing the Number of Loops

Important Points:

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

Description:

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

Example of Use:

To initialize the array a[]:

Code before optimization

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

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

Expanded into assembly language code
(before optimization)

```
_f:
          MOV.L         L238+2,R7
          MOV           #0,R5
          MOV.W         L238,R6
          MOV           R7,R4
          ADD           R7,R6
L237:
          MOV.L         R5,@R4
          ADD           #4,R4
          CMP/HS        R6,R4
          BF            L237
          RTS
          NOP
L238:
          .DATA.W    H'0190
          .DATA.L    _a
```

Code after optimization

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

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

Expanded into assembly language code
(after optimization)

```
_f:
          MOV.L        L239+2,R7
          MOV          #0,R5
          MOV.W        L239,R0
          MOV          R7,R6
          ADD          #4,R6
          MOV          R7,R4
          ADD          R7,R0
L238:
          MOV.L        R5,@R4
          MOV.L        R5,@R6
          ADD          #8,R4
          CMP/HS       R0,R4
          BF/S         L238
          ADD          #8,R6
          RTS
          NOP
L239:
          .DATA.W      H'0190
          .DATA.L      _a
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	24	28	805	455
SH-2	24	24	506	356
SH-2A	20	24	403	253
SH2A-FPU	20	24	403	253
SH-2E	24	24	506	356
SH2-DSP(SH7065)	24	24	605	605
SH-3	24	24	606	407
SH3-DSP	24	24	705	503
SH-4	24	24	305	204
SH-4A	24	24	405	255
SH4AL-DSP	24	24	405	255

5.3.3 Use of Multiplication and Division

Important Points:

When unsure whether to use multiplication/division or shift operations, try using multiplication and division.

Description:

Programs should always be written to make them easy to read. In multiplication and division operations, when the multiplier/divisor and the multiplicand/dividend are unsigned, these operations are replaced by a combination of shift operations as a result of compiler optimization.

Example of Use:

To execute multiplication and division operations:

Source code(multiplication)

```
unsigned int a;

int f(void)
{
      return(a*4);
}
```

Expanded into assembly language code
(after optimization)

```
_f:
          MOV.L         L217,R3
          MOV.L         @R3,R0
          RTS
          SHLL2         R0
L217:
          .DATA.L       _a
```

Source code (division)

```
unsigned int b;

int f(void)
{
      return(b/2);
}
```

Expanded into assembly language code
(after optimization)

```
_f:
          MOV.L       L217,R3
          MOV.L       @R3,R0
          RTS
          SHLR         R0
L217:
          .DATA.L     _b
```

5.3.4 Application of Identities

Important Points:

By applying mathematical identities, the number of operations can sometimes be reduced, improving execution speed.

Description:

Caution should be exercised, since numerical identities, while analytically simple, can result in an increased number of operations in actual numerical application.

Example of Use:

To calculate the sum of integers from 1 to n:

Code before optimization

```
int f( long n )
{
      int i, s;

      for (s = 0, i = 1;
              i <= n; i++)
              s += i;
      return( s );
}
```

Expanded into assembly language code
(before optimization)

```
_f:
          MOV            #1,R5
          CMP/GT         R4,R5
          BT/S           L218
          MOV            #0,R6
L219:
          ADD            R5,R6
          ADD            #1,R5
          CMP/GT         R4,R5
          BF             L219
L218:
          RTS
          MOV            R6,R0
```

Code after optimization

```
int f( long n )
{
         return( n*(n+1) >> 1 );
}
```

Expanded into assembly language code
(after optimization)

```
_f:
          STS.L          MACL,@-R15
          MOV            R4,R0
          ADD            #1,R0
          MUL.L          R4,R0
          STS            MACL,R0
          SHAR           R0
          RTS
          LDS.L          @R15+,MACL
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	18	24	609	31
SH-2	18	16	609	21
SH-2A	16	12	608	10
SH2A-FPU	16	12	608	10
SH-2E	18	16	609	14
SH2-DSP(SH7065)	18	16	710	15
SH-3	18	16	609	18
SH3-DSP	18	16	710	18
SH-4	18	16	507	14
SH-4A	18	16	407	8
SH4AL-DSP	18	16	407	8

Note: Number of cycles n = 100

5.3.5 Use of Tables

Important Points:

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

Description:

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

Example of Use:

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

Code before optimization

```
char f (int i)
{
      char ch;

      switch (i)
      {
          case 0:
               ch = ‘a’; break;
          case 1:
               ch = ‘x’; break;
          case 2:
               ch = ‘b’; break;
      }
      return (ch);
}
```

Expanded into assembly language code
(before optimization)

```
_f:
          MOV           R4,R0
          CMP/EQ        #0,R0
          BT            L218
          CMP/EQ        #1,R0
          BT            L219
          CMP/EQ        #2,R0
          BT            L220
          BRA           L221
          NOP
L218:
          BRA           L221
          MOV           #97,R4
L219:
          BRA           L221
          MOV           #120,R4
L220:
          MOV           #98,R4
```

Code after optimization

```
char chbuf[] = { ‘a’, ‘x’, ‘b’ };

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

Expanded into assembly language code
(after optimization)

```
_f:
          MOV.L         L218+2,R0
          RTS
          MOV.B         @(R0,R4),R0
L218:
          .DATA.W       0
          .DATA.L       _chbuf
```

```
L221:
          RTS
          MOV            R4,R0
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	32	12	13	5
SH-2	32	12	13	5
SH-2A	30	12	11	7
SH2A-FPU	30	12	11	7
SH-2E	32	12	13	5
SH2-DSP(SH7065)	32	12	14	5
SH-3	32	12	13	5
SH3-DSP	32	12	14	6
SH-4	32	12	10	4
SH-4A	32	12	10	4
SH4AL-DSP	32	12	10	4

Note: i = 2

5.3.6 Conditionals

Important Points:

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

Description:

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

Example of Use:

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

Code before optimization

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

Expanded into assembly language code
(before optimization)

```
_f:
          MOV         #1,R3
          CMP/GE      R3,R4
          MOVT        R0
          RTS
          NOP
```

Code after optimization

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

Expanded into assembly language code
(after optimization)

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

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	8	6	5	4
SH-2	8	6	5	4
SH-2A	8	6	6	5
SH2A-FPU	8	6	6	5
SH-2E	8	6	5	4
SH2-DSP(SH7065)	8	6	5	4
SH-3	8	6	5	4
SH3-DSP	8	6	5	4
SH-4	8	6	5	4
SH-4A	8	6	4	3
SH4AL-DSP	8	6	4	3

5.3.7 Eliminating Load/Store Instructions

Important Points:

By eliminating memory access (load, store) instructions, the number of execution cycles can be reduced.

Description:

In coordinate calculations, loading/storing x, y, and z values from/into memory for each iteration is a major factor detracting from performance. Whenever possible, coordinate calculations should be performed in the FPU register, not in structures, thereby reducing the number of memory load and store instructions and improving execution speed.

Example of Use:

To calculate each of the vertex distances (squares) between a fixed point P and a plane formed by the points P0, P1, and P2, and make a decision based on the distances:

Note: In this example, the compile option is -cpu=sh4Δfpu=single.

Code before optimization

```
#define SCAL2(v)  ((v)->x*(v)->x \
               +(v)->y*(v)->y\
               +(v)->z*(v)->z)
#define SubVect(a,b)
               ((a)->x-=(b)->x,\
               (a)->y -= (b)->y,\
               (a)->z -= (b)->z)

typedef  struct {
    float x,y,z;
} POINT3;
typedef  struct  {
   POINT3*   v;
} POLI;

int f(POINT3 *p, POLI *poli,
                            float rad)
{
    float  dst2;
    POINT3 dv;

    dv=poli->v[0];
    SubVect(&dv,p);
    dst2=SCAL2(&dv);
    if (dst2>rad) return 0;

    dv=poli->v[1];
    SubVect(&dv,p);
    dst2=SCAL2(&dv);
    if (dst2>rad) return 0;
```

Code after optimization

```
typedef  struct {
    float x,y,z;
} POINT3;
typedef  struct  {
   POINT3*   v;
} POLI;

float scal2(POINT3 *p1, POINT3 *q1)
{
    float a,b,c;
    float d,e,f;
    float *p=(float *)p1,*q=(float *)q1;

    a=*p++; d=*q++;
    b=*p++; e=*q++; a-=d;
    c=*p++; f=*q++; b-=e;
                       c-=f;
    return a*a+b*b+c*c;
}


int f(POINT3 *p,POLI *poli, float rad)
{
    float  d;
    POINT3 *q;

    q=poli->v;
    d2=scal2(q++,p);
    if (d2>rad) return 0;

    d2=scal2(q++,p);
    dv=poli->v[2];
    SubVect(&dv,p);
    dst2=SCAL2(&dv);
    if (dst2>rad) return 0;

    return 1;
}
```

Expanded into assembly language code
(before optimization)

```
_f:
          ADD         #-16,R15
          MOV.L       @R5,R1
          MOV         #4,R0
          FMOV.S      FR4,FR5
          MOV.L       @R1,R6
          MOV.L       @(4,R1),R5
          MOV.L       @(8,R1),R7
          MOV.L       R6,@R15
          MOV.L       R5,@(4,R15)
          MOV.L       R7,@(8,R15)
          FMOV.S      @R4,FR4
          FMOV.S      @R15,FR8
          FMOV.S      @(R0,R4),FR7
          FSUB        FR4,FR8
          MOV.L       R1,@(12,R15)
          FMOV.S      FR8,@R15
          FMOV.S      @(R0,R15),FR8
          FSUB        FR7,FR8
          FMOV.S      FR8,@(R0,R15)
          MOV         #8,
          FMOV.S      @(R0,R4),FR6
          FMOV.S      @(R0,R15),FR8
          FMOV.S      @R15,FR0
          FSUB        FR6,FR8
          FMOV.S      FR8,@(R0,R15)
          MOV         #4,R0
          FMOV.S      @(R0,R15),FR9
          FMOV.S      @(R0,R15),FR8
          MOV         #8,R0
          FMUL        FR8,FR9
          FMOV.S      @(R0,R15),FR8
          FMAC        FR0,FR0,FR9
          FMOV.S      @(R0,R15),FR0
          FMAC        FR0,FR8,FR9
          FCMP/GT     FR5,FR9
          BT          L12
          MOV.L       @(12,R1),R6
```

```
    if (d2>rad) return 0;

    d2=scal2(q++,p);
    if (d2>rad) return 0;

    return 1;
}
```

Expanded into assembly language code
(after optimization)

```
_scal2:
          FMOV.S      @R4,FR0
          FMOV.S      @R5,FR8
          ADD         #4,R4
          ADD         #4,R5
          FSUB        FR8,FR0
          FMOV.S      @R4,FR9
          FMOV.S      @R5,FR8
          MOV         #4,R0
          FMOV.S      @(R0,R4),FR6
          FSUB        FR8,FR9
          FMOV.S      @(R0,R5),FR8
          FSUB        FR8,FR6
          FMOV.S      FR9,FR8
          FMUL        FR9,FR8
          FMAC        FR0,FR0,FR8
          FMOV.S      FR6,FR0
          FMAC        FR0,FR6,FR8
          RTS
          FMOV.S      FR8,FR0
_f:
          MOV.L       R13,@-R15
          MOV.L       R14,@-R15
          STS.L       PR,@-R15
          FMOV.S      FR14,@-R15
          MOV.L       @R5,R14
          MOV          R4,R13
          FMOV.S      FR4,FR14
          MOV          R4,R5
          MOV          R14,R4
          BSR          _scal2
          ADD          #12,R14
          FCMP/GT     FR14,FR0
          BT           L18
          MOV          R13,R5
          BSR        _ _scal2
          MOV          R14,R4
          FCMP/GT     FR14,FR0
          MOV         #4,R0
          MOV.L       @(16,R1),R4
          MOV.L       @(20,R1),R5
          MOV.L       R6,@R15
          MOV.L       R4,@(4,R15)
          MOV.L       R5,@(8,R15)
          FMOV.S      @R15,FR8
          FSUB        FR4,FR8
          FMOV.S      FR8,@R15
          FMOV.S      @(R0,R15),FR8
          FSUB        FR7,FR8
          FMOV.S      FR8,@(R0,R15)
          MOV         #8,R0
          FMOV.S      @(R0,R15),FR8
          FSUB        FR6,FR8
          FMOV.S      FR8,@(R0,R15)
          MOV         #4,R0
          FMOV.S      @(R0,R15),FR9
          FMOV.S      @(R0,R15),FR8
          MOV         #8,R0
          FMOV.S      @R15,FR0
          FMUL        FR8,FR9
          FMOV.S      @(R0,R15),FR8
          FMAC        FR0,FR0,FR9
          FMOV.S      @(R0,R15),FR0
          FMAC        FR0,FR8,FR9
          FCMP/GT     FR5,FR9
          BT           L12
          MOV.L       @(24,R1),R6
          MOV         #4,R0
          MOV.L       @(28,R1),R7
          MOV.L       @(32,R1),R4
          MOV.L       R6,@R15
          MOV.L       R7,@(4,R15)
          MOV.L       R4,@(8,R15)
          FMOV.S      @R15,FR8
          FSUB        FR4,FR8
          FMOV.S      FR8,@R15
          FMOV.S      @(R0,R15),FR8
          FSUB        FR7,FR8
          FMOV.S      FR8,@(R0,R15)
          MOV         #8,R0
          FMOV.S      @(R0,R15),FR8
          FSUB        FR6,FR8
          FMOV.S      FR8,@(R0,R15)
          MOV         #4,R0
          FMOV.S      @(R0,R15),FR9
          FMOV.S      @(R0,R15),FR8
          MOV         #8,R0
          FMOV.S      @R15,FR0
          FMUL        FR8,FR9
          FMOV.S      @(R0,R15),FR8
```

```
          BT/S         L18
          ADD          #12,R14
          MOV          R13,R5
          BSR          _scal2
          MOV          R14,R4
          FCMP/GT     FR14,FR0
          BF/S         L20
          MOV          #1,R0
L18:
          MOV          #0,R0
L20:
          FMOV.S      @R15+,FR14
          LDS.L       @R15+,PR
          MOV.L       @R15+,R14
          RTS
          MOV.L       @R15+,R13
          FMAC        FR0,FR0,FR9
          FMOV.S      @(R0,R15),FR0
          FMAC        FR0,FR8,FR9
          FCMP/GT     FR5,FR9
          BF/S        L14
          MOV         #1,R0
L12:
          MOV         #0,R0
L14:
          RTS
          ADD         #16,R15
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH2A-FPU	196	90	115	106
SH-2E	196	62	141	128
SH-4	196	62	123	87
SH-4A	196	62	97	93

Analysis of the Program before and after Optimization:

The number of load and store instructions for both cases are compared below.

For a single iteration in x, y, z:

Before optimization, we have

```
dv=poli->v[0];              One LOAD instruction
SubVect(&dv,p);             Two LOAD and one STORE instruction
dst2=SCAL2(&dv);            Two LOAD instructions
if (dst2>rad) rerun 0;
```

This is repeated three times, for a total of 18 load/store instructions.

After optimization, we have

```
a=*p++; d=*q++;
b=*p++; e=*q++; a-=d;
c=*p++; f=*q++; b-=e;
                c-=f;
return a*a+b*b+c*c;
```

Here p and q are loaded, for two instructions, times three iterations for a total of six load/store instructions.

In this way, memory access instructions can be reduced to 1/3. Because the instruction set for the SuperH microcomputers includes essentially no instructions that can directly calculate memory data, the number of instructions is increased compared with operations in the FPU register.

In addition, storing to memory can disrupt pipelines. Thus reduction of the number of memory access operations can also result in smoother pipeline operations.

Addendum:

In the optimized program, the fixed point P is loaded three times.

If this is improved so that only one load operation is needed, an even greater performance improvement is obtained.

Considering that, for fixed points in general, loop processing is performed for multiple planes, the fixed point should be loaded into an FPU register variable instead of a structure to perform operations.

5.4 Branching

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

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

Important Points:

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

Description:

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

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

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

Example of Use:

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

Code before optimization

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

Expanded into assembly language code
(before optimization)

```
_x:
          MOV          R4,R0
          CMP/EQ       #1,R0
          BT           L16
          CMP/EQ       #10,R0
          BT           L17
```

Code after optimization

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

Expanded into assembly language code
(after optimization)

```
_x:
          MOV          R4,R0
          CMP/EQ       #1,R0
          BF           L22
          BRA          L23
          MOV          #2,R4
          BRA         L18
          NOP
L16:
          BRA         L19
          MOV         #2,R2
L17:
          BRA         L19
          MOV         #4,R2
L18:
          MOV         #0,R2
L19:
          RTS
          MOV         R2,R0
```

```
L22:
          CMP/EQ      #10,R0
          BF/S        L23
          MOV         #0,R4
          MOV         #4,R4
L23:
          RTS
          MOV         R4,R0
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	28	22	11	9
SH-2	28	22	11	9
SH-2A	22	20	8	5
SH2A-FPU	22	20	8	5
SH-2E	28	22	11	9
SH2-DSP(SH7065)	28	22	12	10
SH-3	28	22	11	9
SH3-DSP	28	22	12	10
SH-4	28	22	8	7
SH-4A	28	22	7	7
SH4AL-DSP	28	22	7	7

Note: a=1

5.5 Inline Expansion

Matters related to inline expansion that should be considered are listed in table 5.6.

Table 5.6 Suggestions Relating to Inline Expansion

Area	Suggestion	Sections
Inline expansion of functions	Advantages may be gained by inline expansion of functions that are called frequently. However, function expansion results in larger program sizes. This feature should be selected in consideration of the balance between speed of execution and available ROM.	5.5.1
Inline expansion with embedded assembly language	Code written in assembly language can be called using the same interface as for C language functions.	5.5.2

5.5.1 Inline Expansion of Functions

Important Points:

Functions that are called frequently can be inline-expanded to improve execution speed.

Description:

Through inline expansion of functions that are called frequently, speed of execution can be improved. Expansion of functions called within a loop can have a particularly great effect. This option should be used only when there is a need to improve speed of execution even at the expense of increasing the program size.

Example of Use:

To exchange the elements of the array a and the array b:

Code before optimization

```
 int x[10], y[10];
 static void g(int *a, int *b, int i)
{
          int temp;

          temp = a[i];
          a[i] = b[i];
          b[i] = temp;
}

void f (void)
{
          int i;

          for (i=0;i<10;i++)
                        g(x, y, i);
}
```

Code after optimization

```
int x[10], y[10];
#pragma inline (g)
static void g(int *a, int *b, int i)
{
         int temp;

           temp = a[i];
           a[i] = b[i];
           b[i] = temp;
}

void f (void)
{
           int i;

           for (i=0;i<10;i++)
                          g(x, y, i);
}
```

Expanded into assembly language code
(before optimization)

```
_$g:
          SHLL2       R6
          MOV         R6,R0
          MOV.L       @(R0,R4),R1
          MOV.L       @(R0,R5),R2
          MOV.L       R2,@(R0,R4)
          RTS
          MOV.L       R1,@(R0,R5)
_f:
          MOV.L       R11,@-R15
          MOV.L       R12,@-R15
          MOV.L       R13,@-R15
          MOV.L       R14,@-R15
          STS.L       PR,@-R15
          MOV         #0,R14
          MOV.L       L14+2,R12
          MOV.L       L14+6,R13
          MOV         #10,R11
L12:
          MOV         R14,R6
          MOV         R12,R4
          MOV         R13,R5
          BSR         _$g
          ADD         #1,R14
          CMP/GE      R11,R14
          BF          L12
          LDS.L       @R15+,PR
          MOV.L       @R15+,R14
          MOV.L       @R15+,R13
          MOV.L       @R15+,R12
          RTS
          MOV.L       @R15+,R11
L14:
          .RES.W      1
          .DATA.L     _x
          .DATA.L     _y
```

Expanded into assembly language code
(after optimization)

```
_f:
          MOV         #10,R1
          MOV.L       L13+2,R4
          MOV.L       L13+6,R5
L11:
          MOV.L       @R5,R6
          MOV.L       @R4,R2
          DT           R1
          MOV.L       R2,@R5
          MOV.L       R6,@R4
          ADD         #4,R5
          BF/S        L11
          ADD         #4,R4
          RTS
          NOP
L13:
          .RES.W     1
          .DATA.L    _y
          .DATA.L    _x
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	54	36	210	137
SH-2	54	36	210	118
SH-2A	38	32	164	74
SH2A-FPU	50	32	187	74
SH-2E	54	36	210	118
SH2-DSP(SH7065)	52	36	305	138
SH-3	54	36	234	147
SH3-DSP	52	36	294	156
SH-4	54	36	203	97
SH-4A	54	36	155	85
SH4AL-DSP	52	36	185	85

5.5.2 Inline Expansion with Embedded Assembly Language

Important Points:

Assembly language code can be included within a C program to speed execution.

Description:

Sometimes it is desirable to write code in assembly language for enhanced performance, and in particular for improving speed of execution. In such cases, it is possible to write only critical code in assembly language, and call it in the same way one would call a C language function. This feature must be used with the -code=asmcode option.

Example of Use:

To swap the upper and lower bytes of elements in the array big, and store the result in the array little.

Code before optimization

```
#define A_MAX 10
typedef unsigned char UChar;
short big[A_MAX],little[A_MAX];
short swap(short p1)
{
      short ret;

     *((UChar *)(&ret)+1) =
      *((UChar *)(&p1));
     *((UChar *)(&ret)) =
      *((UChar *)(&p1)+1);
    return ret;
}

void f (void)
{
      int  i;
      short *x, *y;

      x = little;
      y = big;
      for(i=0; i<A_MAX; i++,
                           x++, y++){
            *x = swap(*y);
      }
}
```

Expanded into assembly language code
(before optimization)

```
_swap:
          ADD          #-8,R15
          MOV          R4,R0
          MOV.L        R4,@R15
          MOV.W        R0,@(2,R15)
```

Code after optimization

```
#define A_MAX 10
#pragma inline_asm (swap)
typedef unsigned char UChar;
short big[A_MAX],little[A_MAX];
short swap(short p1)
{
     SWAP.B R4,R0
}

void f (void)
{
     int  i;
     short *x, *y;

     x = little;
     y = big;
     for(i=0; i<A_MAX; i++, x++, y++){
                    *x = swap(*y);
    }
}
```

Expanded into assembly language code
(after optimization)

```
_swap:
          SWAP.B R4,R0
          .ALIGN      4
          RTS
          NOP
          MOV.B        @(2,R15),R0
          MOV.B        R0,@(5,R15)
          MOV.B        @(3,R15),R0
          MOV.B        R0,@(4,R15)
          MOV.W        @(4,R15),R0
          RTS
          ADD          #8,R15
_f:
          MOV.L        R12,@-R15
          MOV.L        R13,@-R15
          MOV.L        R14,@-R15
          STS.L        PR,@-R15
          MOV.L        L14+2,R13
          MOV.L        L14+6,R14
          MOV          #10,R12
L12:
          BSR          _swap
          MOV.W        @R14+,R4
          DT           R12
          MOV.W        R0,@R13
          BF/S         L12
          ADD          #2,R13
          LDS.L        @R15+,PR
          MOV.L        @R15+,R14
          MOV.L        @R15+,R13
          RTS
          MOV.L        @R15+,R12
L14:
          .RES.W       1
          .DATA.L      _little
          .DATA.L      _big
```

```
_f:
          MOV.L        R12,@-R15
          MOV.L        R13,@-R15
          MOV.L        R14,@-R15
          MOV.L        L15,R13
          MOV.L        L15+4,R14
          MOV          #10,R12
L12:
          BRA          L14
          MOV.W        @R14+,R4
L15:
          .DATA.L      _little
          .DATA.L      _big
L14:
           SWAP.B      R4,R0
          .ALIGN       4
          DT           R12
          MOV.W        R0,@R13
          BT/S         L17
          ADD          #2,R13
          MOV.L        L16+2,R3
          JMP          @R3
          NOP
L17:
          MOV.L        @R15+,R14
          MOV.L        @R15+,R13
          RTS
          MOV.L        @R15+,R12
L16:
          .RES.W       1
          .DATA.L      L12
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	46	68	285	172
SH-2	46	64	286	171
SH-2A	30	60	202	126
SH2A-FPU	42	76	215	148
SH-2E	46	64	286	171
SH2-DSP(SH7065)	46	64	318	193
SH-3	46	64	113	185
SH3-DSP	46	64	112	177
SH-4	46	64	62	108
SH-4A	46	64	182	117
SH4AL-DSP	46	64	182	117

5.6 Use of the Global Base Register (GBR)

5.6.1 Offset Reference Using the Global Base Register (GBR)

Important Points:

By using the GBR to reference external variables using offsets, performance can be improved.

Description:

By using offsets to reference frequently accessed external variables with the GBR as a base register, more compact object code can be generated. In addition, the number of instructions is reduced, for improved speed of execution.

Example of Use:

To substitute the contents of a structure y into a structure x:

Note: In this example, the compile option is –cpu=sh2 –gbr=user.

Code before optimization

```
struct {
      char  c1;
      char  c2;
      short s1;
      short s2;
      long  l1;
      long  l2;
} x, y;

void f (void)
{
    x.c1 = y.c1;
    x.c2 = y.c2;
    x.s1 = y.s1;
    x.s2 = y.s2;
    x.l1 = y.l1;
    x.l2 = y.l2;
}
```

Expanded into assembly language code
(before optimization)

```
_f:
          MOV.L       L12,R5
          MOV.L       L12+4,R6
          MOV.B       @(1,R5),R0
          MOV.B       @R5,R1
          MOV.B       R0,@(1,R6)
          MOV.W       @(2,R5),R0
          MOV.L       @(8,R5),R4
          MOV.W       R0,@(2,R6)
```

Code after optimization

```
#pragma gbr_base(x,y)
struct {
      char  c1;
      char  c2;
      short s1;
      short s2;
      long  l1;
      long  l2;
} x, y;

void f (void)
{
      x.c1 = y.c1;
      x.c2 = y.c2;
      x.s1 = y.s1;
      x.s2 = y.s2;
      x.l1 = y.l1;
      x.l2 = y.l2;
}
```

Expanded into assembly language code
(after optimization)

```
_f:
   MOV.B  @(_y2-(STARTOF $G0),GBR),R0
   MOV.B  R0,@(_x2-(STARTOF $G0),GBR)
   MOV.B  @(_y2-(STARTOF $G0)+1,GBR),R0
   MOV.B  R0,@(_x2-(STARTOF $G0)+1,GBR)
   MOV.W  @(_y2-(STARTOF $G0)+2,GBR),R0
   MOV.W  R0,@(_x2-(STARTOF $G0)+2,GBR)
   MOV.W  @(_y2-(STARTOF $G0)+4,GBR),R0
   MOV.W  R0,@(_x2-(STARTOF $G0)+4,GBR)
```

```
          MOV.W       @(4,R5),R0         MOV.L  @(_y2-(STARTOF $G0)+8,GBR),R0
          MOV.L       @(12,R5),R7        MOV.L  R0,@(_x2-(STARTOF $G0)+8,GBR)
          MOV.B       R1,@R6             MOV.L  @(_y2-(STARTOF $G0)+12,GBR),R0
          MOV.W       R0,@(4,R6)         RTS
          MOV.L       R4,@(8,R6)         MOV.L  R0,@(_x2-(STARTOF $G0)+12,GBR)
          RTS
          MOV.L       R7,@(12,R6)
L12:
        .DATA.L     _y
        .DATA.L     _x
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	40	26	22	25
SH-2	40	26	22	25
SH-2A	40	26	17	18
SH2A-FPU	40	26	17	18
SH-2E	40	26	22	25
SH2-DSP(SH7065)	40	26	22	25
SH-3	40	26	26	27
SH3-DSP	40	26	36	31
SH-4	40	26	18	21
SH-4A	40	26	15	13
SH4AL-DSP	40	26	15	13

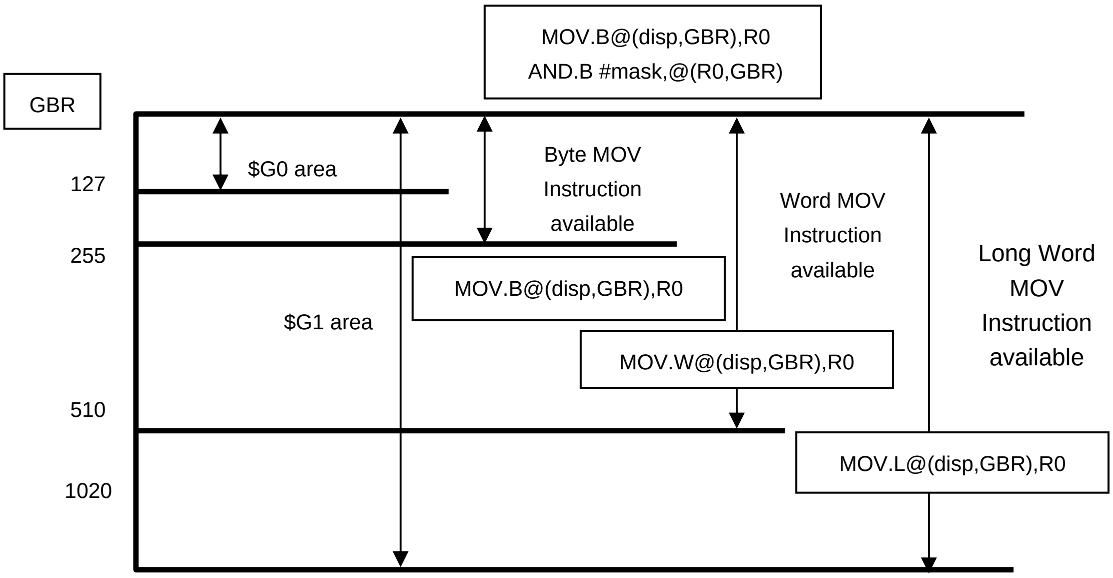
5.6.2 Selective Use of Global Base Register (GBR) Area

Important Points:

Selective use of the GBR0 and GBR1 areas will improve performance.

Description:

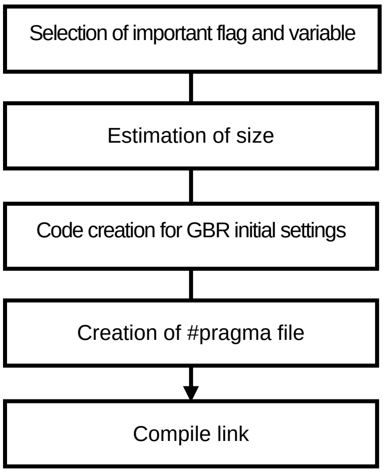
- #pragma gbr_base areas



- The characteristics and applications of GBR base addressing:

Section (area)	Characteristic	Application
$G0 (bgr_base)	The bit processing, setting, and referencing of byte data are efficiently made.	Flag data of byte
$G1 (bgr_base1)	Setting and referencing of data are efficiently made.	General variables

- The procedure for using the GBR base:


Selection of important flag and variable
Estimation of size
Code creation for GBR initial settings
Creation of #pragma file
Compile link

Select important flag and variable, and nominate them for a candidate of #pragma gbr_base.

Check the size of each section for $G0 and $G1, and make sure that the section size of $G1 falls within the area size of byte, word, and long word respectively.

```
MOV #STARTOF $G0,R0
LDC  R0,GBR
```

Initial settings are required before initial access to gbr_base.

At linkage, the $G1 section should be allocated immediately after the $G0 section.

Example of Use:

To access the bit field:

Note: In this example, the compile option is –cpu=sh2 –gbr=user.

```
C source

#pragma gbr_base (bitf)

struct BitField {
  unsigned char a : 1 ;
  unsigned char b : 1 ;
  unsigned char c : 1 ;
  unsigned char d : 1 ;
  unsigned char e : 1 ;
  unsigned char f : 1 ;
  unsigned char g : 1 ;
  unsigned char h : 1 ;
} bitf ;

main()
{
  bitf.a = 1 ; // bit set
  bitf.b = 0 ; // bit
clear

  if (bitf.c)
    bitf.d = 1 ;
  else
    bitf.e = 1 ;
}
```

```
#pragma not specified

.EXPORT      _bitf
.EXPORT      _main
.SECTION     P,CODE,ALIGN=4
_main:       ; function: main
               ; frame size=0
MOV.L        L241,R4   ; _bitf
MOV.B        @R4,R0
OR           #128,R0
MOV.B        R0,@R4
MOV.B        @R4,R0
AND          #191,R0
MOV.B        R0,@R4
MOV          R4,R0
MOV.B        @R0,R0
TST          #32,R0
BT           L238
MOV.B        @R4,R0
BRA          L240
OR           #16,R0
L238:
MOV.B        @R4,R0
OR           #8,R0
L240:
RTS
MOV.B        R0,@R4
L241:
.DATA.L      _bitf
.SECTION     B,DATA,ALIGN=4
_bitf:       ; static: bitf
               .RES.B      1
               .END
```

```
#pragma specified

.EXPORT      _bitf
.EXPORT      _main
.SECTION     ,CODE,ALIGN=4
_main:         ;function:main
                 ; frame size=0
  MOV #_bitf-(STARTOF$G0),R0
  OR.B    #128,@(R0,GBR)
  AND.B   #191,@(R0,GBR)
  TST.B    #32,@(R0,GBR)
  BT           L238
  BRA          L239
  OR.B     #16,@(R0,GBR)
L238:
  OR.B     #8,@(R0,GBR)
L239:
  RTS
  NOP
.SECTION     G0,DATA,ALIGN=4
_bitf:         ; static: bitf
.DATAB.B     1,0
.END
```

5.7 Control of Register Save/Restore Operations

Important Points:

Through innovations to register save/restore operations, execution speed can be improved. (Ver.8)

Description:

By eliminating save and restore operations for variable registers at the entry and exit points of end functions, execution speed and efficiency of ROM use can be improved. However, one of the following types of processing must be performed, possibly resulting in decreased, rather than increased, performance. Hence the code to which this method is applied should be carefully studied.

(1) The registers for register variables must be saved/restored at the function calling the function for which register save/restore operations are omitted.

(2) The object must not allocate registers for register variables spanning function calls.

Example of Use:

To combine stack save/restores using a function table:

Code before optimization

```
#define LISTMAX 2
typedef
   int ARRAY[LISTMAX][LISTMAX][LISTMAX];
ARRAY ary1, ary2, ary3;
void init(int, ARRAY);
void copy(ARRAY, ARRAY);
void sum(ARRAY, ARRAY, ARRAY);
void table (void)
{
    init(74755, ary1);
    copy(ary1, ary2);
    sum(ary1, ary2, ary3);
}

void init (int seed, ARRAY p)
{
    int    i, j, k;

    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
            for ( k = 0; k < LISTMAX; k++ ){
                seed = ( seed * 1309 ) & 16383;
                p[i][j][k] = seed;
            }
}

void copy (ARRAY p, ARRAY q)
{
    int    i, j, k;
```

Code after optimization

```
#pragma regsave (table)
#pragma noregalloc (table)
#pragma noregsave (init, copy, sum)
#define LISTMAX 2
typedef
   int ARRAY[LISTMAX][LISTMAX][LISTMAX];
ARRAY  ary1, ary2, ary3;
void init(int, ARRAY);
void copy(ARRAY, ARRAY);
void sum(ARRAY, ARRAY, ARRAY);
void table (void)
{
    init(74755, ary1);
    copy(ary1, ary2);
    sum(ary1, ary2, ary3);
}

void init (int seed, ARRAY p)
{
    int    i, j, k;

    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
            for ( k = 0; k < LISTMAX; k++ ){
                seed = ( seed * 1309 ) & 16383;
                p[i][j][k] = seed;
            }
}

void copy (ARRAY p, ARRAY q)
```

```
    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
            for ( k = 0; k < LISTMAX; k++ )
                q[k][i][j] = p[i][j][k];
}

void sum (ARRAY p, ARRAY q, ARRAY r)
{
    int    i, j, k;

    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
            for ( k = 0; k < LISTMAX; k++ )
                r[i][j][k] = p[i][j][k] +
                                   q[i][j][j];
}
```

Expanded into assembly language code
(before optimization)

```
_table:
          MOV.L        R14,@-R15
          STS.L        PR,@-R15
          MOV.L        L270+6,R14
          MOV.L        L270+10,R4
          BSR          _init
          MOV          R14,R5
          MOV.L        L270+14,R5
          BSR          _copy
          MOV          R14,R4
          MOV          R14,R4
          LDS.L        @R15+,PR
          MOV.L        L270+18,R6
          MOV.L        L270+14,R5
          BRA          _sum
          MOV.L        @R15+,R14
_init:
          MOV          #2,R6
          MOV.L        R13,@-R15
          MOV          #0,R13
          MOV.L        R12,@-R15
          MOV          R5,R12
          MOV.L        R10,@-R15
          MOV.L        R9,@-R15
          MOV.L        R8,@-R15
          MOV          R5,R8
          STS.L        MACL,@-R15
          ADD          #32,R8
          MOV.W        L270,R9
          MOV.W        L270+2,R10
```

```
{
    int    i, j, k;

    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
            for ( k = 0; k < LISTMAX; k++ )
                q[k][i][j] = p[i][j][k];
}

void sum (ARRAY p, ARRAY q, ARRAY r)
{
    int    i, j, k;

    for ( i = 0; i < LISTMAX; i++ )
        for ( j = 0; j < LISTMAX; j++ )
            for ( k = 0; k < LISTMAX; k++ )
                r[i][j][k] = p[i][j][k] +
                                   q[i][j][j];
}
```

Expanded into assembly language code
(after optimization)

```
_table:
          MOV.L        R14,@-R15
          MOV.L        R13,@-R15
          MOV.L        R12,@-R15
          MOV.L        R11,@-R15
          MOV.L        R10,@-R15
          MOV.L        R9,@-R15
          MOV.L        R8,@-R15
          FMOV.S       FR15,@-R15
          FMOV.S       FR14,@-R15
          FMOV.S       FR13,@-R15
          FMOV.S       FR12,@-R15
          STS.L        PR,@-R15
          MOV.L        L270+10,R4
          MOV.L        L270+6,R5
          STS.L        MACH,@-R15
          STS.L        MACL,@-R15
          BSR          _init
          NOP
          MOV.L        L270+6,R4
          MOV.L        L270+14,R5test3
          BSR          _copy
          NOP
          MOV.L        L270+6,R4
          MOV.L        L270+14,R5test3
          MOV.L        L270+18,R6
          BSR          _sum
          NOP
          LDS.L        @R15+,MACL
          LDS.L        @R15+,MACH
```

```
L261:
          MOV         R13,R1
          MOV         R12,R0
L262:
          MOV         R13,R7
          MOV         R0,R5
L263:
          MUL.L       R10,R4
          ADD         #1,R7
          STS         MACL,R3
          MOV         R3,R4
          AND         R9,R4
          CMP/GE      R6,R7
          MOV.L       R4,@R5
          BF/S        L263
          ADD         #4,R5
          ADD         #1,R1
          CMP/GE      R6,R1
          BF/S        L262
          ADD         #8,R0
          ADD         #16,R12
          CMP/HS      R8,R12
          BF           L261
          LDS.L       @R15+,MACL
          MOV.L       @R15+,R8
          MOV.L       @R15+,R9
          MOV.L       @R15+,R10
          MOV.L       @R15+,R12
          RTS
          MOV.L       @R15+,R13
_copy:
          MOV.L       R14,@-R15
          MOV.L       R13,@-R15
          MOV         #2,R13
          MOV.L       R11,@-R15
          MOV.L       R10,@-R15
          MOV.L       R9,@-R15
          MOV         #0,R9
          MOV.L       R8,@-R15
          MOV         R9,R14
          ADD         #-4,R15
          MOV         R5,R8
          ADD         #32,R8
L264:
          MOV         R9,R7
          MOV         R14,R10
          SHLL2       R10
          SHLL        R10
          MOV         R14,R3
          SHLL2       R3
          SHLL2       R3
          ADD         R4,R3
```

```
          LDS.L       @R15+,PR
          FMOV.S      @R15+,FR12
          FMOV.S      @R15+,FR13
          FMOV.S      @R15+,FR14
          FMOV.S      @R15+,FR15
          MOV.L       @R15+,R8
          MOV.L       @R15+,R9
          MOV.L       @R15+,R10
          MOV.L       @R15+,R11
          MOV.L       @R15+,R12
          MOV.L       @R15+,R13
          RTS
          MOV.L       @R15+,R14
_init:
          MOV.W       L270+2,R10
          MOV         R5,R8
          MOV.W       L270,R9
          ADD         #32,R8
          MOV         #2,R6
          MOV         R5,R12
          MOV         #0,R13
L261:
          MOV         R12,R0
          MOV         R13,R1
L262:
          MOV         R0,R5
          MOV         R13,R7
L263:
          MUL.L       R10,R4
          ADD         #1,R7
          CMP/GE      R6,R7
          STS         MACL,R3
          MOV         R3,R4
          AND         R9,R4
          MOV.L       R4,@R5
          BF/S        L263
          ADD         #4,R5
          ADD         #1,R1
          CMP/GE      R6,R1
          BF/S        L262
          ADD         #8,R0
          ADD         #16,R12
          CMP/HS      R8,R12
          BF          L261
          RTS
          NOP
_copy:
          ADD         #-4,R15
          MOV         R5,R8
          MOV         #0,R9
          ADD         #32,R8
          MOV         R9,R14
          MOV.L       R3,@R15
L265:
          MOV.L       @R15,R3
          MOV         R5,R6
          MOV         R7,R11
          SHLL2       R11
          SHLL        R11
          ADD         R3,R11
          MOV         R7,R1
          SHLL2       R1
L266:
          MOV.L       @R11+,R3
          MOV         R6,R0
          ADD         R10,R0
          ADD         #16,R6
          CMP/HS      R8,R6
          BF/S        L266
          MOV.L       R3,@(R0,R1)
          ADD         #1,R7
          CMP/GE      R13,R7
          BF          L265
          ADD         #1,R14
          CMP/GE      R13,R14
          BF          L264
          ADD         #4,R15
          MOV.L       @R15+,R8
          MOV.L       @R15+,R9
          MOV.L       @R15+,R10
          MOV.L       @R15+,R11
          MOV.L       @R15+,R13
          RTS
          MOV.L       @R15+,R14
L270:
          .DATA.W     H'3FFF
          .DATA.W     H'051D
          .DATA.W     0
          .DATA.L     _ary1
          .DATA.L     H'00012403
          .DATA.L     _ary2
          .DATA.L     _ary3
_sum:
          MOV.L       R14,@-R15
          MOV.L       R13,@-R15
          MOV.L       R12,@-R15
          MOV         #0,R12
          MOV.L       R11,@-R15
          MOV         #2,R11
          MOV.L       R10,@-R15
          MOV.L       R9,@-R15
          MOV.L       R8,@-R15
          ADD         #-4,R15
          MOV         R12,R8
```

```
          MOV         #2,R13
L264:
          MOV         R14,R3
          SHLL2       R3
          SHLL2       R3
          MOV         R14,R10
          ADD         R4,R3
          MOV         R9,R7
          SHLL2       R10
          MOV.L       R3,@R15
          SHLL        R10
L265:
          MOV         R7,R11
          MOV.L       @R15,R3
          SHLL2       R11
          MOV         R7,R1
          SHLL        R11
          MOV         R5,R6
          SHLL2       R1
          ADD         R3,R11
L266:
          MOV         R6,R0
          ADD         #16,R6
          MOV.L       @R11+,R3
          CMP/HS      R8,R6
          ADD         R10,R0
          BF/S        L266
          MOV.L       R3,@(R0,R1)
          ADD         #1,R7
          CMP/GE      R13,R7
          BF          L265
          ADD         #1,R14
          CMP/GE      R13,R14
          BF          L264
          RTS
          ADD         #4,R15
_sum:
          ADD         #-4,R15
          MOV         #0,R12
          MOV         R12,R8
          MOV         #2,R11
L267:
          MOV         R8,R13
          SHLL2       R13
          SHLL2       R13
          MOV         R12,R10
L268:
          MOV         R10,R14
          MOV         R10,R3
          SHLL2       R3
          MOV         R12,R9
          SHLL2       R14
```

```
L267:
          MOV         R12,R10
          MOV         R8,R13
          SHLL2       R13
          SHLL2       R13
L268:
          MOV         R12,R9
          MOV         R12,R7
          MOV         R10,R14
          SHLL2       R14
          SHLL        R14
          MOV         R10,R3
          SHLL2       R3
          MOV.L       R3,@R15
L269:
          MOV         R13,R0
          ADD         R6,R0
          ADD         R14,R0
          ADD         R7,R0
          MOV         R13,R3
          MOV.L       R0,@-R15
          MOV         R13,R2
          MOV.L       @(4,R15),R0
          ADD         #1,R9
          ADD         R5,R3
          ADD         R14,R3
          MOV.L       @(R0,R3),R3
          ADD         R4,R2
          ADD         R14,R2
          ADD         R7,R2
          MOV.L       @R2,R1
          CMP/GE      R11,R9
          MOV.L       @R15+,R2
          ADD         R1,R3
          MOV.L       R3,@R2
          BF/S        L269
          ADD         #4,R7
          ADD         #1,R10
          CMP/GE      R11,R10
          BF          L268
          ADD         #1,R8
          CMP/GE      R11,R8
          BF          L267
          ADD         #4,R15
          MOV.L       @R15+,R8
          MOV.L       @R15+,R9
          MOV.L       @R15+,R10
          MOV.L       @R15+,R11
          MOV.L       @R15+,R12
          MOV.L       @R15+,R13
          RTS
          MOV.L       @R15+,R14
```

```
          MOV.L       R3,@R15
          SHLL        R14
          MOV         R12,R7
L269:
          MOV         R13,R0
          ADD         R6,R0
          ADD         R14,R0
          MOV         R13,R2
          ADD         R7,R0
          MOV         R13,R3
          ADD         R4,R2
          MOV.L       R0,@-R15
          ADD         R14,R2
          MOV.L       @(4,R15),R0
          ADD         R5,R3
          ADD         R7,R2
          ADD         R14,R3
          MOV.L       @R2,R1
          MOV.L       @(R0,R3),R3
          ADD         #1,R9
          MOV.L       @R15+,R2
          CMP/GE      R11,R9
          ADD         R1,R3
          MOV.L       R3,@R2
          BF/S        L269
          ADD         #4,R7
          ADD         #1,R10
          CMP/GE      R11,R10
          BF          L268
          ADD         #1,R8
          CMP/GE      R11,R8
          BF          L267
          RTS
          ADD         #4,R15
L270:
          .DATA.W     H'3FFF
          .DATA.W     H'051D
          .DATA.W     0
          .DATA.L     _ary1
          .DATA.L     H'00012403
          .DATA.L     _ary2
          .DATA.L     _ary3
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	292	288	684	669
SH-2	238	242	446	426
SH-2E	238	258	446	438
SH2-DSP(SH7065)	236	252	490	470
SH-3	238	242	476	458
SH3-DSP	236	252	489	487
SH-4	238	258	301	313

5.8 Specification Using Two-Byte Addresses

Important Points:

By using two bytes to represent the addresses of variables and functions, efficiency of ROM use is improved.

Description:

When variables and functions are located at addresses which can be represented by two bytes and addresses of the code on the referring side are specified with two bytes, the code size can be shrunk.

Example of Use:

To call the external function g when the value of the variable x is 1:

Code before optimization

```
extern int  x;
extern void g(void);

void f (void)
{
   if (x == 1)
        g();
}
```

Expanded into assembly language code
(before optimization)

```
_f:
         MOV.L       L218+2,R3
         MOV.L       @R3,R0
         CMP/EQ      #1,R0
         BF          L219
         MOV.L       L218+6,R2
         JMP         @R2
         NOP
L219:
         RTS
         NOP
L218:
         .DATA.W      0
         .DATA.L     _x
         .DATA.L     _g
```

Code after optimization

```
#pragma abs16(x,g)
extern int  x;
extern void g(void);

void f (void)
{
   if (x == 1)
        g();
}
```

Expanded into assembly language code
(after optimization)

```
_f:
          MOV.W       L238+2,R3
          MOV.L       @R3,R0
          CMP/EQ      #1,R0
          BF          L239
          MOV.W       L238,R2
          JMP         @R2
          NOP
L239:
          RTS
          NOP
L238:
          .DATA.W     _g
          .DATA.W     _x
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-1	28	28	15	11
SH-2	28	28	15	11
SH-2A	24	24	13	11
SH2A-DSP	24	24	13	11
SH-2E	28	28	15	11
SH2-DSP(SH7065)	28	28	16	12
SH-3	28	28	15	11
SH3-DSP	28	28	17	12
SH-4	28	28	13	10
SH-4A	28	28	9	6
SH4AL-DSP	28	28	9	6

Note: x =1, function g is void g( ){ }

5.9 Cache Use

Performance can be improved through effective cache use.

5.9.1 Prefetch Instruction

Important Points:

When accessing array variables, by executing a prefetch instruction prior to use, the execution speed can be improved. (SH-2A, SH2A-FPU, SH-3, SH3-DSP, SH-4, SH-4A, and SH4AL-DSP only)

Description:

When successively accessing an array in a loop, by performing a prefetch prior to referencing an array member, execution speed is improved. And, by expanding the loop even more effective prefetch operations are possible.

However, continuous execution of prefetch instructions do not result in improved speed. Subsequent prefetch instructions should be executed only after the previous prefetch instruction has completed.

Example of Use:

To store the result of an operation performed using the elements a, b, and c of the array "data" in the element d (SH-4).

Note: In this example, the compile option is –cpu=sh4 –fpu=single.

Code before optimization

```
typedef struct {
    float a, b, c, d;
} data_t;

data_t data[2048];

int f( void )
{
    data_t *p1, *p2;
    data_t *end = &data[2048];
    float a1, b1, c1, t11, t12;
    float a2, b2, c2, t21, t22;

    for( p1=data, p2=data+1; p1<end;
 p1+=2, p2+=2 ){
        a1 = p1->a;
        b1 = p1->b;
        t11 = a1 * a1;
        t12 = b1 * b1;
        t11 += t12;
        c1 = 1/t11;
        p1->c = c1;
        a1 += b1;
        a1 += c1;
        p1->d = a1;

        a2 = p2->a;
```

Code after optimization

```
#include <machine.h>

typedef struct {
    float a, b, c, d;
} data_t;

data_t data[2048];

int f( void )
{
    data_t *p1, *p2;
    data_t *end = &data[2048];
    float a1, b1, c1, t11, t12;
    float a2, b2, c2, t21, t22;
    data_t *next = data+4;

    for( p1=data, p2=data+1; p1<end;
 p1+=2, p2+=2 ){
        prefetch(next);
        next += 2;

        a1 = p1->a;
        b1 = p1->b;
        t11 = a1 * a1;
        t12 = b1 * b1;
        t11 += t12;
        c1 = 1/t11;
        b2 = p2->b;
        t21 = a2 * a2;
        t22 = b2 * b2;
        t21 += t22;
        c2 = 1/t21;
        p2->c = c2;
        a2 += b2;
        a2 += c2;
        p2->d = a2;
    }
}
```

Expanded into assembly language code
(before optimization)

```
_f:
          MOV.L       L252+6,R6
          MOV.L       L252+2,R7
          MOV         R7,R5
          MOV         R7,R4
          ADD         R7,R6
          CMP/HS      R6,R4
          ADD         #16,R5
          BT/S        L250
          FLDI1       FR5
L251:
          MOV         #4,R0
          FMOV.S      @R4,FR4
          FMOV.S      @(R0,R4),FR6
          MOV         #8,R0
          FMOV.S      FR4,FR7
          FMUL        FR4,FR7
          FMOV.S      FR6,FR8
          FMUL        FR6,FR8
          FADD        FR6,FR4
          FADD        FR8,FR7
          FMOV.S      FR7,FR3
          FMOV.S      FR5,FR7
          FDIV        FR3,FR7
          FADD        FR7,FR4
          FMOV.S      FR7,@(R0,R4)
          MOV         #12,R0
          FMOV.S      FR4,@(R0,R4)
          MOV         #4,R0
          FMOV.S      @(R0,R5),FR6
          MOV         #8,R0
```

```
        p1->c = c1;
        a1 += b1;
        a1 += c1;
        p1->d = a1;

        a2 = p2->a;
        b2 = p2->b;
        t21 = a2 * a2;
        t22 = b2 * b2;
        t21 += t22;
        c2 = 1/t21;
        p2->c = c2;
        a2 += b2;
        a2 += c2;
        p2->d = a2;
    }
}
```

Expanded into assembly language code
(after optimization)

```
_f:
          MOV.L       L253+6,R7
          MOV.L       L253+2,R0
          MOV         R0,R4
          MOV         R0,R5
          ADD         R0,R7
          MOV         R0,R6
          CMP/HS      R7,R4
          ADD         #16,R5
          ADD         #64,R6
          BT/S        L251
          FLDI1       FR5
L252:
          PREF        @R6
          MOV         #4,R0
          FMOV.S      @R4,FR4
          FMOV.S      @(R0,R4),FR6
          MOV         #8,R0
          FMOV.S      FR4,FR7
          FMUL        FR4,FR7
          FMOV.S      FR6,FR8
          FMUL        FR6,FR8
          FADD        FR6,FR4
          ADD         #32,R6
          FADD        FR8,FR7
          FMOV.S      FR7,FR3
          FMOV.S      FR5,FR7
          FDIV        FR3,FR7
          FADD        FR7,FR4
          FMOV.S      FR7,@(R0,R4)
          MOV         #12,R0
          FMOV.S      @R5,FR4                       FMOV.S      FR4,@(R0,R4)
          FMOV.S      FR6,FR8                       MOV         #4,R0
          FMUL        FR6,FR8                       FMOV.S      @(R0,R5),FR6
          FMOV.S      FR4,FR7                       MOV         #8,R0
          FMUL        FR4,FR7                       FMOV.S      @R5,FR4
          FADD        FR6,FR4                       FMOV.S      FR6,FR8
          FADD        FR8,FR7                       FMUL        FR6,FR8
          FMOV.S      FR7,FR3                       FMOV.S      FR4,FR7
          FMOV.S      FR5,FR7                       FMUL        FR4,FR7
          FDIV        FR3,FR7                       FADD        FR6,FR4
          FADD        FR7,FR4                       FADD        FR8,FR7
          FMOV.S      FR7,@(R0,R5)                  FMOV.S      FR7,FR3
          ADD         #32,R4                        FMOV.S      FR5,FR7
          MOV         #12,R0                        FDIV        FR3,FR7
          CMP/HS      R6,R4                         FMOV.S      FR7,@(R0,R5)
          FMOV.S      FR4,@(R0,R5)                  FADD        FR7,FR4
          BF/S        L251                          ADD         #32,R4
          ADD         #32,R5                        MOV         #12,R0
L250:                                               CMP/HS      R7,R4
          RTS                                       FMOV.S      FR4,@(R0,R5)
          NOP                                       BF/S        L252
L252:                                               ADD         #32,R5
          .DATA.W     0                   L251:
          .DATA.L     _data                         RTS
          .DATA.L     H'00008000                    NOP
                                          L253:
                                                    .DATA.W     0
                                                    .DATA.L     _data
                                                    .DATA.L     H'00008000
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-3	18	24	609	31
SH3-DSP	18	16	609	21
SH-4	16	12	608	10

Notes: 1. For SH-3, SH3-DSP, and SH-4, load the program into the external memory.
2. For SH-3 and SH3-DSP, perform the measurement with the number of external memory access cycle set to 16.
3. For SH-4, perform the measurement with the number of memory access wait cycle set to 15.
4. Cache miss should be considered

5.9.2 Tiling

Important Points:

In this method, a program is created with concentration of data access such that data cache misses are reduced.

In other words, in this technique, calculations which can be performed while the cache is hit are performed first.

Description:

A simple example, consider the creation of an array which takes the sum of the differences of two arrays A and B.

By creating the program with the order of access varied, data cache misses can be reduced.

Example of Use:

For array members a, b, c, and d, a structure performs the calculation

$$di=\Sigma j \;\; bj-aj$$

Code before optimization

```
typedef struct {
    float a,b,c,d;
} data_t;

f(data_t data[], int n)
{
    data_t *p,*q;
    data_t *p_end = &data[n];
    data_t *q_end = p_end;
    float a,d;

    for (p = data; p < p_end; p++){
        a = p->a;
        d = 0.0f;
        for (q = data; q < q_end; q++){
            d += q->b -a;
        }
        p->d=d;
   }
}
```

Expanded into assembly language code (before optimization)

Code after optimization

```
#define STRIDE 512
typedef struct {
    float a,b,c,d;
} data_t;

f(data_t data[], int n)
{
    data_t *p,*q, *end=&data[n];
    data_t *pp, *qq;
    data_t *pp_end, *qq_end;
    float a,d;


    for (p = data; p < end; p = pp_end){
        pp_end = p + STRIDE;
        for (q = data; q < end; q = qq_end){
            qq_end = q + STRIDE;
            for (pp = p; pp < pp_end && pp

               a = pp->a;
               d = pp->d;
               for (qq = q; qq < qq_end
                        && qq < end; qq++){
                   d += qq->b -a;
               }
               p->d = d;
           }
       }
   }
}
```

Expanded into assembly language code (after optimization)

```
_f:
          MOV           R5,R1
          SHLL2         R1
          SHLL2         R1
          FLDI0         FR6
          ADD           R4,R1
          BRA           L244
          MOV           R4,R6
L245:
          MOV           R4,R5
          FMOV.S        @R6,FR5
          CMP/HS        R1,R5
          BT/S          L246
          FMOV.S        FR6,FR4
L247:
          STS           FPSCR,R3
          MOV.L         L248,R2
          MOV           #4,R0
          FMOV.S        @(R0,R5),FR3
          ADD           #16,R5
          AND           R2,R3
          CMP/HS        R1,R5
          LDS           R3,FPSCR
          FSUB          FR5,FR3
          BF/S          L247
          FADD          FR3,FR4
L246:
          MOV           #12,R0
          FMOV.S        FR4,@(R0,R6)
          ADD           #16,R6
L244:
          CMP/HS        R1,R6
          BF            L245
          RTS
          NOP
L248:
            .DATA.L     H'FFE7FFFF
            .END
```

```
_f:
          MOV.L         R14,@-R15
          MOV           R5,R7
          MOV.L         R13,@-R15
          SHLL2         R7
          MOV.L         R11,@-R15
          SHLL2         R7
          MOV.L         R10,@-R15
          ADD           R4,R7
          MOV.W         L259,R11
          BRA           L249
          MOV           R4,R13
L250:
          MOV           R13,R10
          ADD           R11,R10
          BRA           L251
          MOV           R4,R14
L252:
          MOV           R14,R1
          ADD           R11,R1
          BRA           L253
          MOV           R13,R6
L254:
          MOV           #12,R0
          FMOV.S        @R6,FR5
          FMOV.S        @(R0,R6),FR4
          BRA           L255
          MOV           R14,R5
L256:
          STS           FPSCR,R3
          MOV.L         L259+2,R2
          MOV           #4,R0
          FMOV.S        @(R0,R5),FR3
          ADD           #16,R5
          AND           R2,R3
          LDS           R3,FPSCR
          FSUB          FR5,FR3
          FADD          FR3,FR4
L255:
          CMP/HS        R1,R5
          BT            L257
          CMP/HS        R7,R5
          BF            L256
L257:
          MOV           #12,R0
          ADD           #16,R6
          FMOV.S        FR4,@(R0,R13)
L253:
          CMP/HS        R10,R6
          BT            L258
          CMP/HS        R7,R6
          BF            L254
```

```
L258:
          MOV           R1,R14
L251:
          CMP/HS        R7,R14
          BF            L252
          MOV           R10,R13
L249:
          CMP/HS        R7,R13
          BF            L250
          MOV.L         @R15+,R10
          MOV.L         @R15+,R11
          MOV.L         @R15+,R13
          RTS
          MOV.L         @R15+,R14
L259:
          .DATA.W       H'2000
          .DATA.L       H'FFE7FFFF
          .END
          .END
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-3	76	132	931×10^3	725×10^3
SH3-DSP	76	132	940×10^3	697×10^3
SH-4	52	104	315×10^3	43×10^3

Notes: 1. n = 4096, STRIDE = 512
2. Cache miss should be considered.

Analysis of the Program before and after Optimization:

After optimization, there is a fourfold nesting of loops, and processing is complex, with increased code size. However, by means of this processing the overhead associated with cache misses can be reduced. This technique is not effective when there is little data to be processed, but is more effective for larger data sizes.

Prior to optimization, the values of data[0] through data[n-1] are referenced in succession in order to calculate data[0]->d.

Then, in order to calculate data[1]->d, the values of data[0] through data[n-1] are again referenced; but when the size of the array data is large compared with the size of the cache, the value of data[0] will no longer be in the cache, resulting in a cache miss.

Because the data in a large area is referenced in succession, by the time the same data is next referenced, it is no longer present in the cache.

In the optimized program, data is divided into smaller areas for accessing, and so during this data accessing there are fewer cache misses. The order of calculations is changed so that, during cache hits, other calculations using the same data are also performed.

5.10 Matrix Operations

Important Points:

If intrinsic functions are used in matrix operations, execution speed can be improved.

Here an array acting as a multiplier must be stored in the floating point extended register in advance.

Description:

The product of an array of four rows and four columns is normally calculated by successive operations using loops, resulting in processing complexity, with little expectation of fast execution. However, the SH-4 supports intrinsic functions for matrix operations; by using these functions, a significant improvement in execution speed is possible.

Example of Use:

To store the product of the array data and the array tbl in the array ret:

Note: In this example, the compile option is –cpu=sh4 –fpu=single.

Code before optimization

```
void mtrx4mul1 (float data[4][4],
  float tbl[4][4], float ret[4][4])
{
int i,j,k;

for(i=0;i<4;i++){
  for(j=0;j<4;j++){
    for(k=0;k<4;k++){
      ret[i][j]+=
        data[i][k]*tbl[k][j];
      }
    }
  }
}
```

Code after optimization

```
#include <machine.h>
void _mtrx4mul (float data[4][4],
  float tbl[4][4],float ret[4][4])
{
   ld_ext(tbl);
   mtrx4mul(data,ret);
}
```

Expanded into assembly language code
(before optimization)

```
_mtrx4mul1:
          MOV.L       R14,@-R15
          MOV.L       R13,@-R15
          MOV.L       R11,@-R15
          MOV.L       R10,@-R15
          MOV.L       R9,@-R15
          MOV.L       R8,@-R15
          ADD         #-4,R15
          MOV         #0,R8
          MOV.L       R8,@R15
          MOV         #4,R14
L244:
          MOV.L       @R15,R11
          MOV         R8,R9
```

Expanded into assembly language code
(after optimization)

```
_mtrx4mul:
    ADD         #-12,R15
    MOV.L       R4,@(8,R15)
    MOV.L       R5,@(4,R15)
    MOV.L       R6,@R15
    MOV.L       @(8,R15),R2
    FRCHG
    FMOV.S      @R2+,FR0
    FMOV.S      @R2+,FR1
    FMOV.S      @R2+,FR2
    FMOV.S      @R2+,FR3
    FMOV.S      @R2+,FR4
    FMOV.S      @R2+,FR5
    FMOV.S      @R2+,FR6
```

```
          SHLL2        R11
          SHLL2        R11
L245:
          MOV          R9,R1
          MOV          #0,R7
          SHLL2        R1
          MOV          R8,R10
          MOV          #0,R13
          ADD          R5,R7
L246:
          MOV          R11,R0
          ADD          R6,R0
          ADD          R1,R0
          MOV          R11,R3
          MOV.L        R0,@-R15
          ADD          R4,R3
          MOV.L        @R15+,R2
          MOV          R1,R0
          ADD          R13,R3
          FMOV.S       @(R0,R7),FR0
          FMOV.S       @R2,FR2
          ADD          #1,R10
          FMOV.S       @R3,FR3
          CMP/GE       R14,R10
          ADD          #16,R7
          FMAC         FR0,FR3,FR2
          FMOV.S       FR2,@R2
          BF/S         L246
          ADD          #4,R13
          ADD          #1,R9
          CMP/GE       R14,R9
          BF           L245
          MOV.L        @R15,R3
          ADD          #1,R3
          CMP/GE       R14,R3
          BF/S         L244
          MOV.L        R3,@R15
          ADD          #4,R15
          MOV.L        @R15+,R8
          MOV.L        @R15+,R9
          MOV.L        @R15+,R10
          MOV.L        @R15+,R11
          MOV.L        @R15+,R13
          RTS
          MOV.L        @R15+,R14
```

```
          FMOV.S       @R2+,FR7
          FMOV.S       @R2+,FR8
          FMOV.S       @R2+,FR9
          FMOV.S       @R2+,FR10
          FMOV.S       @R2+,FR11
          FMOV.S       @R2+,FR12
          FMOV.S       @R2+,FR13
          FMOV.S       @R2+,FR14
          FMOV.S       @R2+,FR15
          FRCHG
          MOV.L        @(4,R15),R3
          MOV.L        @R15,R1
          FMOV.S       @R3+,FR0
          FMOV.S       @R3+,FR1
          FMOV.S       @R3+,FR2
          FMOV.S       @R3+,FR3
          FTRV           XMTRX,FV0
          ADD            #16,R1
          FMOV.S       FR3,@-R1
          FMOV.S       FR2,@-R1
          FMOV.S       FR1,@-R1
          FMOV.S       FR0,@-R1
          FMOV.S       @R3+,FR0
          FMOV.S       @R3+,FR1
          FMOV.S       @R3+,FR2
          FMOV.S       @R3+,FR3
          FTRV           XMTRX,FV0
          ADD            #32,R1
          FMOV.S       FR3,@-R1
          FMOV.S       FR2,@-R1
          FMOV.S       FR1,@-R1
          FMOV.S       FR0,@-R1
          FMOV.S       @R3+,FR0
          FMOV.S       @R3+,FR1
          FMOV.S       @R3+,FR2
          FMOV.S       @R3+,FR3
          FTRV           XMTRX,FV0
          ADD            #32,R1
          FMOV.S       FR3,@-R1
          FMOV.S       FR2,@-R1
          FMOV.S       FR1,@-R1
          FMOV.S       FR0,@-R1
          FMOV.S       @R3+,FR0
          FMOV.S       @R3+,FR1
          FMOV.S       @R3+,FR2
          FMOV.S       @R3+,FR3
          FTRV           XMTRX,FV0
          ADD            #32,R1
          FMOV.S       FR3,@-R1
          FMOV.S       FR2,@-R1
          FMOV.S       FR1,@-R1
          FMOV.S       FR0,@-R1
```

```
ADD          #12,R15
RTS
NOP
```

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte]		Execution Speed [cycle]	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-4	110	118	603	113

5.11 Software Pipelines

Important Points:

By designing a program so as to eliminate waits for the results of operations, smooth pipeline flow is achieved.

Description:

Software pipelining involves the elimination of waits for instructions accompanying data flow (definition and use of values). For example, in code to take a sum of values, when addition by an ADD instruction immediately follows the definition of a load instruction, a wait occurs. If the load instruction is issued earlier, however, this wait can be eliminated. If the processing is within a loop, loading of data for the next iteration is performed during the current iteration.

Typical examples of this method are division and square-root calculations. The SH-4 are provided with FDIV and FSQRT instructions; but because of a large latency (the cycle from issue of an instruction until generation of the result; 12 cycles in the case of the SH-4), in programs which use the result immediately, wait cycles until execution of the next instruction occur.

Example of Use:

Example of a loop which takes the sum of square roots

Note: In this example, the compile option is –cpu=sh4 –fpu=single.

Code before optimization

```
#include <mathf.h>

float func1(float *p, int cnt){
    float ret=0.0f;
    do {
       ret+=sqrtf(*p++);
       x();

    } while(cnt--);
    return ret;
 }
```

Expanded into assembly language code
(before optimization)

```
_func1:
          MOV.L     R14,@-R15
          FMOV.S    FR15,@-R15
          STS.L     PR,@-R15
          ADD       #-8,R15
          MOV.L     L262,R14
          FLDI0     FR15
          MOV.L     R4,@(4,R15)
          MOV.L     R5,@R15
L260:
          MOV.L     @(4,R15),R3
          ADD       #4,R3
          MOV.L     R3,@(4,R15)
```

Code after optimization

```
#include <mathf.h>

float func21(float *p, int cnt){
    float ret=0.0f;
    float sq=0.0f;
    do {
       ret+=sq;
       sq=sqrtf(*p++);
       x();
    } while (cnt--);
    return ret;
 }
```

Expanded into assembly language code
(after optimization)

```
_func21:
          MOV.L     R14,@-R15
          FMOV.S    FR15,@-R15
          FMOV.S    FR14,@-R15
          STS.L     PR,@-R15
          ADD       #-8,R15
          FLDI0     FR4
          MOV.L     L263,R14
          FMOV.S    FR4,FR15
          MOV.L     R4,@(4,R15)
          MOV.L     R5,@R15
          FMOV.S    FR4,FR14
L261:
```

```
          ADD         #-4,R3
          FMOV.S      @R3,FR3
          FSQRT       FR3
          JSR         @R14
          FADD        FR3,FR15
          MOV.L       @R15,R3
          ADD         #-1,R3
          MOV.L       R3,@R15
          ADD         #1,R3
          TST         R3,R3
          BF          L260
          FMOV.S      FR15,FR0
          ADD         #8,R15
          LDS.L       @R15+,PR
          FMOV.S      @R15+,FR15
          RTS
          MOV.L       @R15+,R14
L262:
          .DATA.L     _x
```

```
          MOV.L       @(4,R15),R3
          FADD        FR15,FR14
          ADD         #4,R3
          MOV.L       R3,@(4,R15)
          ADD         #-4,R3
          FMOV.S      @R3,FR15
          JSR         @R14
          FSQRT       FR15
          MOV.L       @R15,R3
          ADD         #-1,R3
          MOV.L       R3,@R15
          ADD         #1,R3
          TST         R3,R3
          BF          L261
          FMOV.S      FR14,FR0
          ADD         #8,R15
          LDS.L       @R15+,PR
          FMOV.S      @R15+,FR14
          FMOV.S      @R15+,FR15
          RTS
          MOV.L       @R15+,R14
L263:
          .DATA.L     _x
```

Analysis of the Program before and after Optimization:

Before optimization, the FADD instruction is executed immediately after FSQRT, and so wait cycles occur until FSQRT is completed and FADD can be executed.

In the optimized program, after execution of FSQRT, an FADD instruction is issued in the next loop, and so waiting until FADD is eliminated.

Code Size and Execution Speed before and after Optimization:

CPU Type	Code Size [byte] (one loop)		Execution Speed [cycle] (Number of wait cycles in FSQRT)	
	Before Optimization	After Optimization	Before Optimization	After Optimization
SH-4	28	28	9	0

5.12 About Cache Memory

The SuperH series includes the products with on-chip cache memory.

Caching is a mechanism which reduces the frequency of accesses of program and data in memory and speeds program operation.

By using cache memory, the speed of program execution is improved. However, there are various types of caches, and a thorough understanding of their structure and functioning will enable more effective programming.

Here a number of cache structures present in the SuperH series are explained, and suggestions for programming which makes full use of cache memory are presented.

5.12.1 Description of Terms

Cache hit

When the CPU attempts to access external memory, it checks to see if the data to be retrieved already exists in cache memory or not. Cases in which the data is present in cache memory are called cache hits.

In essence, when a cache hit occurs, high-speed cache memory is accessed, eliminating the need to access slower external memory.

Cache miss

When the CPU attempts to access external memory, it checks to see if the data to be retrieved already exists in cache memory or not. Cases in which the data is not present in cache memory are called cache misses.

Cache fill

When a cache miss occurs, the CPU stores the contents of the memory accessed in the cache. This is called cache filling.

Cache line leng

When the CPU performs a cache fill, it does not store only the contents of the memory accessed in the cache; rather, in cache filling it stores the contents of a continuous area of memory including the areas preceding and following the accessed data. The size of this region is called the cache line length. The line length is a fixed length (size) for a given CPU. This line length is the unit when storing data in the cache.

Cache size, number of entries number of lines

The capacity of data stored in the cache is called the cache size.

The number of entries (or number of lines) is determined by the cache line length and the cache size as follows.

(cache size) = (number of entries) × (cache line length)

Write-backand write-through

When, in the cache hit state, an attempt is made to overwrite the memory contents, there are two choices for performing the overwrite.

(1) The contents of cache memory and the contents of external memory are overwritten simultaneously.

(2) Only the cache memory is overwritten.
In the case of (1), the contents of the cache memory and the contents of external memory always coincide. This method is called write-through.

In the case of (2), the most recent data remains in cache memory only; external memory is not overwritten, but retains old data. When this method is used, before the contents of the cache entry are discarded, the data in the entry is written back to the external memory. This method is called write-back. (Ordinarily a flag indicates whether there has been writing one or more times to an entry in the cache, and write-back to external memory is performed only when the flag indicates that cache writing has occurred.)

Cache coherency

This refers to the coincidence of the contents of external memory and the contents of cache memory.

In other words, when the cache is used with the write-back method, there is the possibility that the contents of cache memory do not agree with the contents of external memory, and if a device other than the CPU accesses external memory, because this data has not been updated, erroneous software operation occurs. When some other device accesses the same memory (when common memory is used), either the write-through method should be used, or else write-back of the applicable cache entries for the memory area should be performed prior to accessing by the other device.

Direct mapping

This is another caching method.

In essence, the address of cache memory for storage of data is determined uniquely based on the address in external memory. The offset of the external memory is used to store the data in the address with the same offset in cache memory.

When checking the data location with a given address in cache, the stored location is determined unambiguously by the memory address; hence this method alleviates the burden on the hardware. However, when the offset addresses of frequently used memory locations coincide, the same entry is replaced frequently, and on the other hand there are entries which are used hardly at all.

Hence in some circumstances this method results in inefficient use of cache memory, and so the program design should be considered carefully when opting for this method.

Full-associative method

This is another caching method.

In contrast with direct mapping, all addresses and data for external memory are stored in entries. Entries are replaced starting with those which have not been accessed for the longest time (LRU method), and so the cache can be used with maximum efficiency. However, in order to determine in which cache entry an address of external memory is stored, all cache entries must be checked, and the hardware mechanism becomes complex.

Set-associative method

This method lies midway between direct mapping and the full-associative method; here a number of direct-mapping caches are used (the number used is called the number of ways). As in direct mapping, the cache entry to be used is determined from the offset value of the address in external memory; but of the number of caches present, the way which has not been accessed for the longest time is used.

When searching for the area of the cache in which the address is stored, instead of searching all cache entries, only searches equal to the number of ways are necessary, so that the hardware operations involved are not so complex.

The SH7604, SH7708, SH7707, SH7709, and SH7718 use 4-way set-associative caching.

5.13 SuperH Family Caches

Each of the caches used in the SuperH Series is explained below.

1. SH7032, SH7034, SH7020, and SH7021 Groups (SH-1)

This series is not provided with cache memory. These products are based on CPUs with internal ROM/RAM. If execution is limited to internal ROM/RAM, performance equal to or surpassing that using cache memory is possible.

2. SH704x Group (SH-2)

This series is based on CPUs with internal ROM/RAM. They have CPUs with performance superior to the SH7034, and provide a function enabling part of internal RAM to be used as an instruction cache.

Caching method: Instruction cache (direct mapping)

Cache size: 1 kbyte (in use, 2 kbytes of internal RAM)
Cache line length: 4 bytes (two instructions)
Number of entries: 256

The cache is for instructions only. Because of the low overhead of cache filling, this cache is highly effective for loop processing within 1 kbyte. The effective range of the cache is external memory; it is not effective for internal ROM/RAM. (Internal ROM/RAM can be accessed rapidly, and so a cache need not be used.)

However, 2 kbytes of the 4 kbytes of internal RAM is used, as 1 kbyte of cache memory; and so when using the cache, the internal RAM size is 2 kbytes.

Because this is not a data cache, in some cases overall performance is improved by using all 4 kbytes of RAM for data without using a cache, and using internal ROM preferentially for frequently used code.

3. SH7604 Group (SH-2)

This series is based on a ROM-less processor-type CPU with cache.

Caching method: 4-way set-associative cache (mixed instructions and data)

Cache size: 4 kbytes
Cache line length: 16 bytes
Number of entries: 256 (64×4)
Other: Write-through method

4. SH7707, SH7708, and SH7709 Groups (SH-3)

This series is based on a ROM-less processor-type CPU with cache.

Caching method: 4-way set-associative cache (mixed instructions and data)
Cache size: 8 kbytes
Cache line length: 16 bytes
Number of entries: 512 (128×4)
Other: Selectable among write-through or write-back method

5. SH7750 Group (SH-4)

Caching method: Direct mapping (mixed instructions and data)

Instruction cache

Cache size: 8 kbytes
Cache line length: 32 bytes
Number of entries: 256
Other: 4 kbytes × 2 index modes possible

Data cache (Operand cache)

Cache size: 16 kbytes
Cache line length: 32 bytes
Number of entries: 512
Other: 8 kbytes can be used as internal RAM

Storage queue

Two storage queues for 32 bytes are provided for high-speed transfer to external memory.

The storage queues are buffers for high-speed transfer to external memory.

By using it, high-speed transfer to external memory is possible.

The data cache cannot perform cache blocking, and so there is the possibility of cache replacement.

The storage queue is a reliable mechanism for high-speed transfer which does not cause drops in performance due to cache replacement or other issues.

5.14 Techniques for Cache Utilization

The following describes the techniques for efficient use of the cache.

(1) Reasons for poor performance

Table 5.7 lists the most possible reasons that prevent better performance.

The generally-believed methods and measures required for the best use of the cache are as follows:

[1] Use a debugger or profile tool to check the dependency of functions and execution frequency.
[2] Place the caches in locations having close dependency to reduce cache misses.
[3] Perform size-oriented optimization to make the frequently-executed part into a function.

Table 5.7 Primary Reasons for Poor Performance

Item	Possible reasons	Action
Shifts in addresses on the cache	Shifts in addresses on the cache or in entries may change the contention relationship of the cache.	Change the alignment.
Increase in the program size	The cache hit rate may be degraded due to increase in the dummy area for alignment and increase in the program size.	Perform size-oriented optimization

The following describes details of the actions you must take. However, note that these actions do not always work.

(2) Corrective action for shifts in addresses on the cache

The effective method is to change the alignment of the program to the length of a cache line. By default, the alignment of programs output by the SuperH RISC engine C/C++ compiler is four bytes. Use the “align16” compiler option for the SuperH RISC engine C/C++ compiler to change the alignment to 16 bytes that is equal to the length of one line on the cache. This ensures that the addresses are placed from the beginning of the cache line.

However, note that this method is valid when the length of a cache line is 16 bytes. The program size will increase.

[1] Most efficient method of specifying the “align16” option

- Specify the “align16” option for compact function groups (that use only few lines on the cache).
- Specify the “align16” option for a small function group, if any, within a general common routine groups in the program.
  (Defining the small function group adjacent to the same module reduces cache entry contention.)

(3) Increase in the program size

Increase in the dummy area for alignment and in the program size degrades the cache hit ratio rate.

[1] How to cope with the increase in the program size

You can suppress an increase in the program size as follows:

- Perform optimization by specifying “Size-Oriented” for optimization option for the SuperH RISC engine C/C++ compiler.
- Re-create the program by using functions of smaller size.
- Create a special general-purpose routine.

(4) How to use cache entries

You must check whether the entries in the cache are used uniformly and whether the number of replacements is not concentrated on particular entries. This check is especially important for direct-mapping caching.

Checking these items requires some means for tracing the cache contents.

However, actual tracing is impossible in many cases.

Hence, the means for improving overall performance include the following:

[1] Select functions which are frequently executed.

[2] Determine the start and end addresses of each function from the map file of the linkage editor.

[3] Check the entries of the cache used by the function.

[4] Compile statistics for the entries used by each function.

[5] Check whether particular entries are not concentrated on a cache.

In this stage, if multiple functions are using the same cache line, change the function addresses to eliminate concentration.

You can change addresses by changing the section names at compilation or by changing the input order during linking.

The method in [3] for checking the used entries varies depending on the CPU and caching method. Generally, if direct-mapping caching is used, the entry number is determined by the offset value of the absolute address. (The offset range depends on the cache size.)

If you find it difficult to check for contention by using the procedures [3] through [5], use the following method:

Change the section names of the functions selected in [1] and recompile.

This allocates the functions selected in [1] to continuous addresses. In other words, cache contention does not occur between these functions. This means that the total size of the functions to change the section names must be equal to or smaller than the cache size.

(5) Programming techniques

There are effective programming techniques for efficiently using the cache. See the following for programming:

- Tilig programs

  See section 5.9.2, Tiling.

- Prefetching

  See section 5.9.1, Prefetch Instruction.

Section 6 Efficient Programming Techniques (Supplement)

6.1 How to Specify Options

To create an efficient program, it is effective to specify an optimization option. You can achieve increased effectiveness depending on the selection of an optimization option or the use of an option in combination with others.

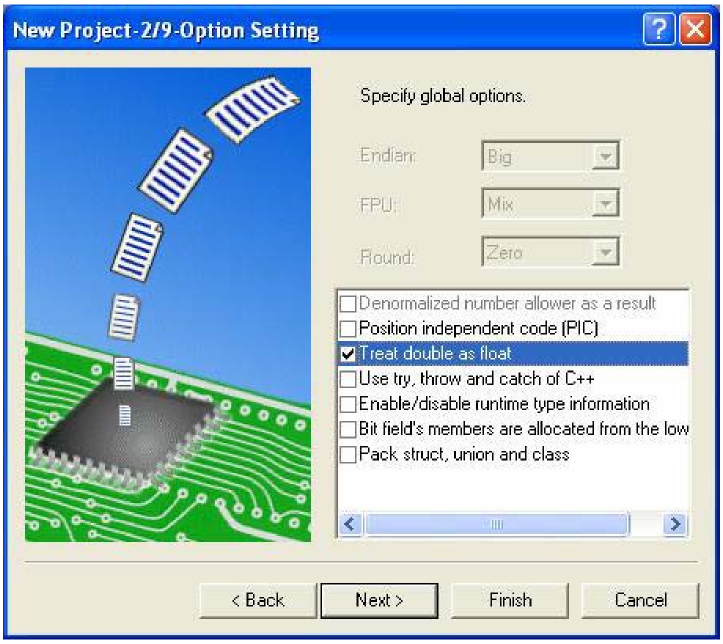
6.1.1 Options for Starting HEW (Floating Point Setting)

Options that can be specified for starting HEW set the environment related to the use of a CPU in general including the handling of endian and floating points.

When a program uses floating points, the floating point setting greatly influences the performance.Concerning both the size and speed, the single-precision floating point mode (32-bit) is more efficient than the double-precision floating point mode.Use the single-precision floating point mode if it is sufficient for your application field.The single precision gives about seven decimal digits of precision while the double precision gives about 17 decimal digits of precision.

Note: The floating point mode specified as described in this section will influence an entire project. Therefore, you cannot specify different modes for each file.

(1) For SH-1, SH-2, SH-2E, SH-2A, SH2-DSP, SH3, SH3-DSP, and SH4AL-DSP
Click the "Treat double as float" check box shown in figure 6.1. This will cause all the floating points (including those declared as double) to be handled in single-precision mode.



New Project-2/9-Option Setting

Specify global options.

Endian: Big

FPU: Mix

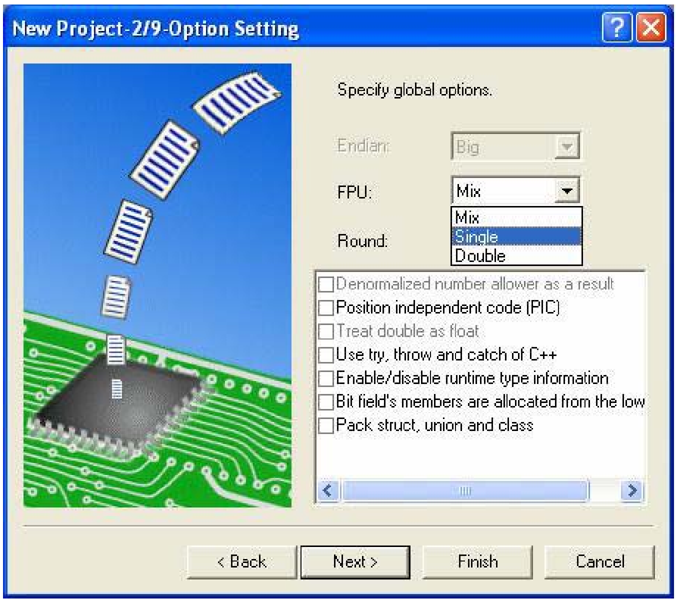
Round: Zero

- □ Denormalized number allower as a result
- □ Position independent code (PIC)
- ☑ Treat double as float
- □ Use try, throw and catch of C++
- □ Enable/disable runtime type information
- □ Bit field's members are allocated from the low
- □ Pack struct, union and class

Figure 6.1 How to Specify Single-Precision Mode
(SH-1, SH-2, SH-2E, SH-2A, SH2-DSP, SH3, SH3-DSP, and SH4AL-DSP)

(2) For SH2A-FPU, SH-4, and SH-4A

Specify "Single" in the FPU menu shown in figure 6.2.



New Project-2/9-Option Setting

Specify global options.

Endian: Big

FPU: Mix

Round:

□Denormalized number allower as a result
□Position independent code (PIC)
□Treat double as float
□Use try, throw and catch of C++
□Enable/disable runtime type information
□Bit field's members are allocated from the low
□Pack struct, union and class

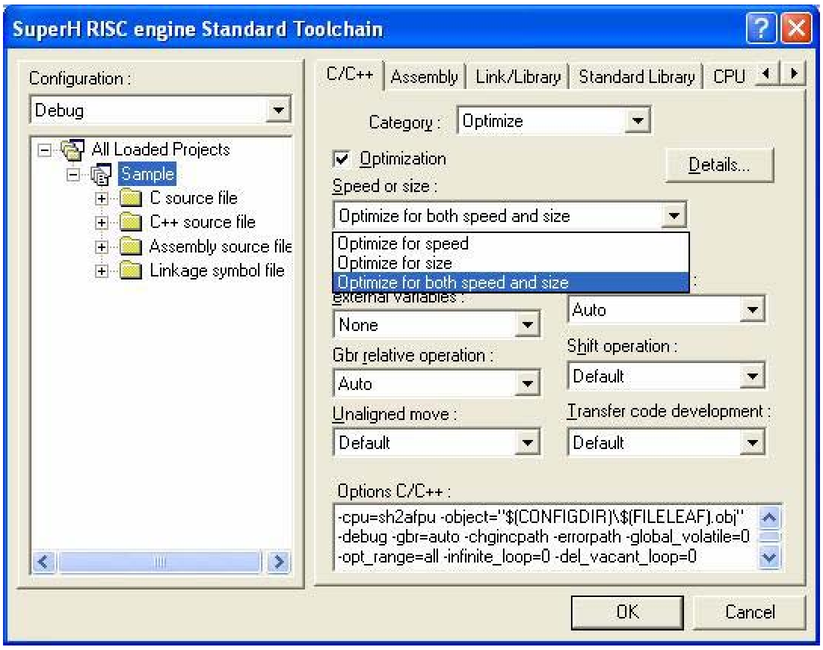
Figure 6.2 How to Specify Single-Precision Mode (SH2A-FPU, SH-4, and SH-4A)

Specify "Double" on this screen to perform all the operations in double-precision mode. Specify "Mix" to calculate float in single-precision mode and double in double-precision mode as described in a program. (This is the same operation as when "Treat double as float" is not checked for SH-1, SH-2, SH-2E, SH-2A, SH2-DSP, SH3, SH3-DSP, or SH4AL-DSP. However, a program may have poorer performance than when double is specified because it is executed while switching between the single- and double-precision modes of the FPU).

6.1.2 How to Specify Optimization Options (Speed and Size)

There are the following major optimization options (figure 6.3).

(a) Optimize for size and speed (default)
(b) Optimize for size
(c) Optimize for speed



Configuration : Debug

Category : Optimize

☑ Optimization

Speed or size : Optimize for both speed and size

External variables : None

Auto

Gbr relative operation : Auto

Shift operation : Default

Unaligned move : Default

Transfer code development : Default

Options C/C++ :

```
-cpu=sh2afpu -object="$(CONFIGDIR)\$(FILELEAF).obj"
-debug -gbr=auto -chgincpath -errorpath -global_volatile=0
-opt_range=all -infinite_loop=0 -del_vacant_loop=0
```

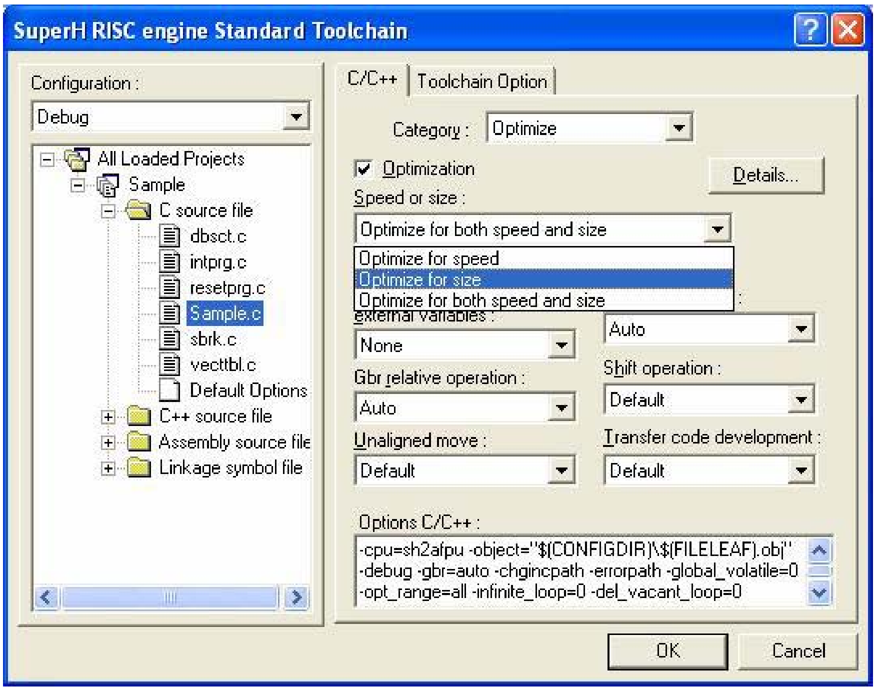
Figure 6.3 Optimization Options

"Optimize for size and speed" (default) performs only optimizations that improve both the size and speed. "Optimize for size" performs optimizations that reduce the size at the expense of the speed in addition to those performed for "Optimize for size and speed". "Optimize for speed" performs optimizations that improve the speed at the expense of the size reduction in addition to those performed for "Optimize for size and speed".

To give priority to either the size or speed, specify either "Optimize for size" or "Optimize for speed" respectively.

In many of the systems, the speed is important only in a limited part of a program. If this is the case, it is effective to optimize the files that require speed using "Optimize for speed" and the others using "Optimize for size". Different optimization options can be specified for each file because they do not change the interface between files.

To specify different options for each file, select a target source file in the directory tree shown on the left and then specify an option (figure 6.4).



Configuration : Debug

Category : Optimize

☑ Optimization

Speed or size : Optimize for both speed and size

external variables : None

Auto

Gbr relative operation : Auto

Shift operation : Default

Unaligned move : Default

Transfer code development : Default

Options C/C++ :

```
-cpu=sh2afpu -object="$(CONFIGDIR)\$(FILELEAF).obj"
-debug -gbr=auto -chgincpath -errorpath -global_volatile=0
-opt_range=all -infinite_loop=0 -del_vacant_loop=0
```

Figure 6.4 How to Specify Different Options for Each File

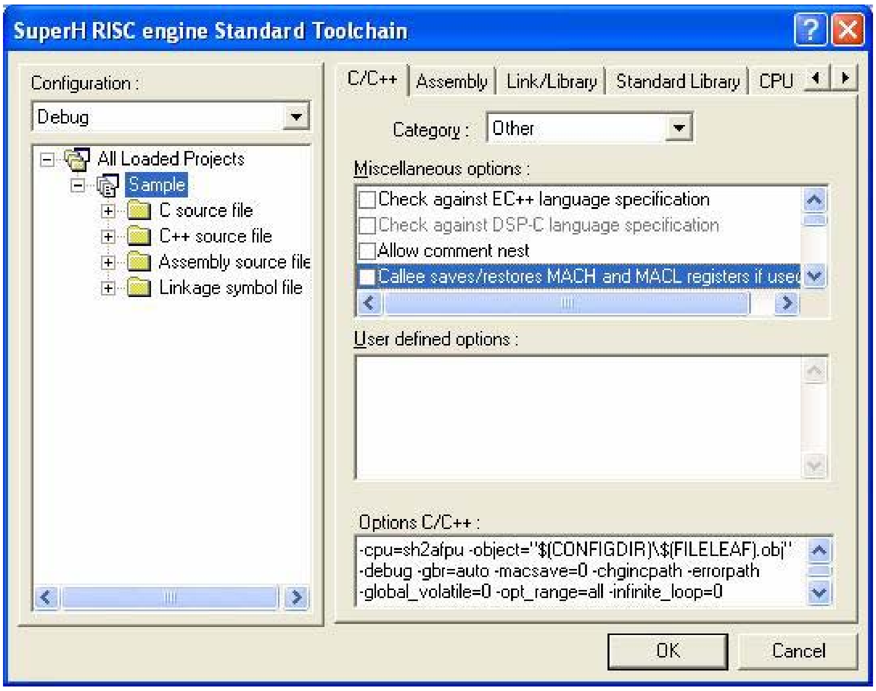
6.1.3 Options Needing Attention for Program Compatibility (Function Interface)

Some of the options that change the interface of a function can improve the execution efficiency of a program. These options, by default supporting the earlier versions of the Compiler for the sake of compatibility, can improve the efficiency without a compatibility problem if you change an entire project. These options are located in the Others menu of the Compiler options.

Note: If a project includes an assembler-written program, you must check which of the conventions is used to create it.

(1) Callee saves/restores MACH and MACL registers if used

Specify this option to have a function that uses the MACH and MACL registers save and restore these registers at the function entry and exit points.These registers, being saved and restored by default, are mostly used as work registers. Thus, you can improve both the size and speed by selecting not to save and restore them (unchecking the check box) (figure 6.5).



Configuration : Debug

Category : Other

Miscellaneous options :

☐ Check against EC++ language specification
☐ Check against DSP-C language specification
☐ Allow comment nest
☐ Callee saves/restores MACH and MACL registers if used

User defined options :

Options C/C++ :

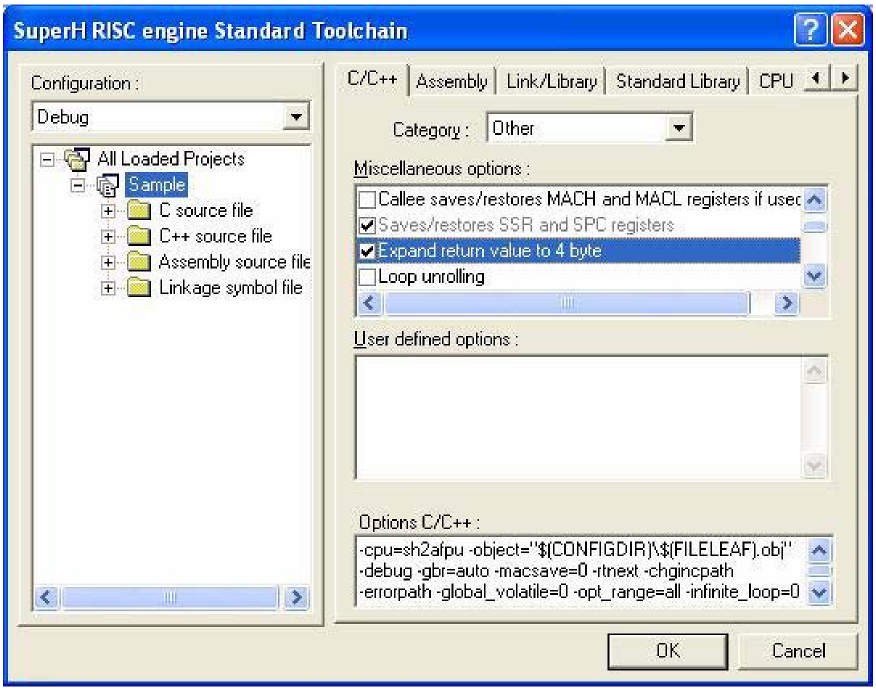
```
-cpu=sh2afpu -object="$(CONFIGDIR)\$(FILELEAF).obj"
-debug -gbr=auto -macsave=0 -chgincpath -errorpath
-global_volatile=0 -opt_range=all -infinite_loop=0
```

Figure 6.5 MACH and MACL Register Save and Restore Option

(2) Expand return value to 4 byte

If the function return type is either char, unsigned char, short, or unsigned short, the sign extension (or zero extension) is performed by default not by the called function but by the calling function (this is a specification compatible with the earlier versions of the Compiler).

If a function is called more than once, it is advantageous in terms of size reduction to have the called function sign-extend a return value (check the check box) instead of having the calling function do so because the extension code needs to be included only once.



Configuration : Debug

Category : Other

Miscellaneous options :

- □ Callee saves/restores MACH and MACL registers if usec
- ☑ Saves/restores SSR and SPC registers
- ☑ Expand return value to 4 byte
- □ Loop unrolling

Options C/C++ :

```
-cpu=sh2afpu -object="$(CONFIGDIR)\$(FILELEAF).obj"
-debug -gbr=auto -macsave=0 -rtnext -chgincpath
-errorpath -global_volatile=0 -opt_range=all -infinite_loop=0
```

Figure 6.6 Return Value Extended Option

Note: The SH register size is 4 bytes. You can create an efficient program by declaring the return values of a function as well as data to be put in the register such as function parameters and local variables as either of the four-byte types, int, unsigned, long, and unsigned long because no sign extension processing is required.
(For details, please refer to 5.1.1, Local Variable.)

6.1.4 Options for Handling Variables with volatile Declaration (volatile Variable)

Specify a volatile declaration to disable the optimization of access to a variable.Declare volatile for an external variable to be used in a program, mainly for the following two purposes:

(a) Disabling the optimization of access to a peripheral input-output register

(b) Disabling the optimization of access to a variable to be shared in different tasks or interrupt processing.

V6.0 and earlier versions of the SH C/C++ Compiler performed hardly any optimization of access to an external variable. However, V7.0 and later versions perform drastically enhanced optimizations.If a program with no volatile declaration for a variable that meets either of the above conditions (1) and (2) was developed using the Compiler V6.0 or earlier, it is recommended to add a volatile declaration to the program when it is ported to the Compiler V7.0 or later.

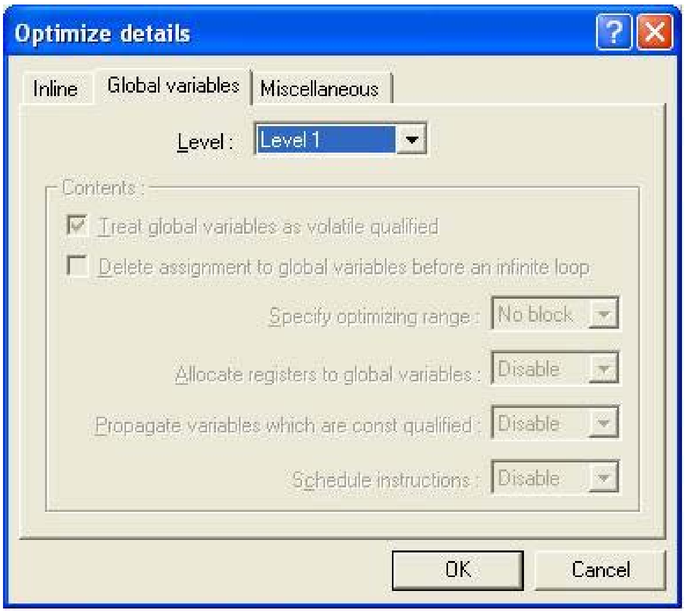
If you do not modify the program, click the "Details..." button in the Optimize menu of the Compiler option to set up the Details option.

Use these options, which will disable optimizations, to disable a minimum range of optimizations without impairing the functionality of a program developed in V6.0 or earlier.

(1) If volatile is not declared for a peripheral input-output register

A peripheral input-output register may have a different operation, depending on the register specifications, if you optimize two consecutive accesses for writing or reading to one access.

To disable such optimizations, specify Level 1 in the "Global variables" tab of the Optimize Details option (figure 6.7).



Level : Level 1

Contents :

☑ Treat global variables as volatile qualified

□ Delete assignment to global variables before an infinite loop

Specify optimizing range : No block

Allocate registers to global variables : Disable

Propagate variables which are const qualified : Disable

Schedule instructions : Disable

Figure 6.7 Optimize Details Option Level 1

However, Level 1 disables other optimizations as well. Retain the effects of a volatile declaration and perform a maximum range of other optimizations as shown below (figure 6.8).

(a) Set Level to "Level 1."

(b) Then, set Level to "Custom" (the Level 1 settings will remain).

(c) Set "Specify optimizing range" to "All."

(d) Set "Allocate registers to global variables" to "Enable."

(e) Set "Propagate variables which are const qualified" to "Enable."

(f) Set "Schedule instructions" to "Enable."

Example:

Source code

```
extern int  x;

void f (void)
{
  x=1;
   x=2;
}
```

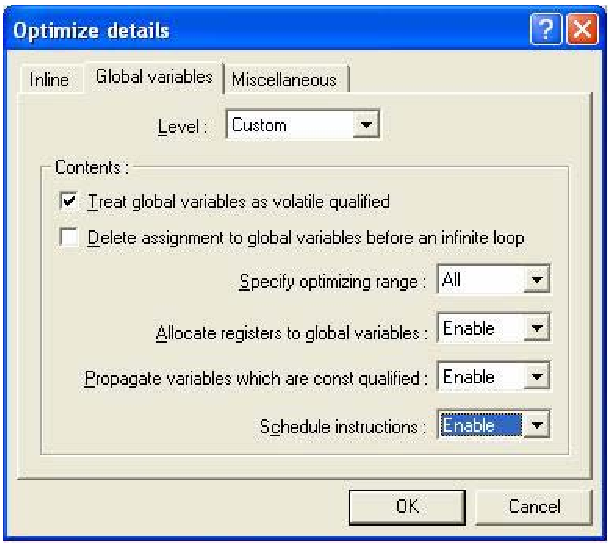
Assembler expansion code

```
_f:
          MOV.L        L11,R6
          MOV          #2,R2
          RTS
          MOV.L        R2,@R6
```

Assembler expansion code (with the option specified)

```
_f:
          MOV.L        L11,R6
          MOV          #1,R2
          MOV.L        R2,@R6
          MOV          #2,R2
          RTS
          MOV.L        R2,@R6
```

In this example, not specifying the option results in combining two accesses to an input-output register into one. Thus, you may have a different effect on the peripheral input-output register.



Level : Custom

Contents :

☑ Treat global variables as volatile qualified

□ Delete assignment to global variables before an infinite loop

Specify optimizing range : All

Allocate registers to global variables : Enable

Propagate variables which are const qualified : Enable

Schedule instructions : Enable

Figure 6.8 Specifying Volatile for a Peripheral Input-Output Register

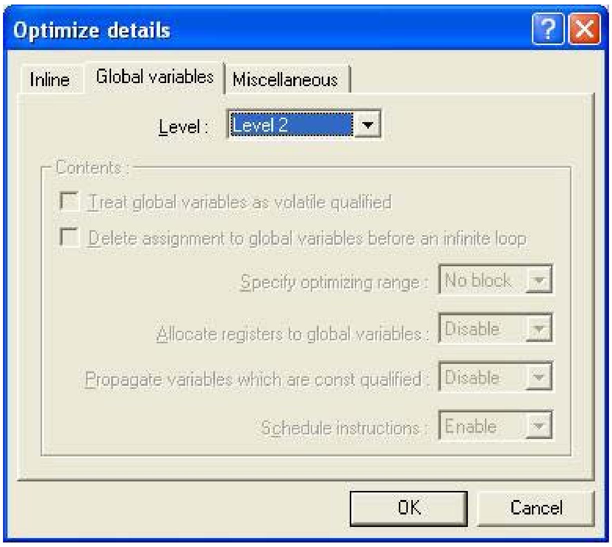
Notes: 1. In a header file created by HEW for each model, volatile is declared for a peripheral input-output register. If external variables without a volatile declaration are only those shared in tasks and interrupts, specify an option described in (2).

2. If two references are made to an external variable in one expression, the order of them is not ensured. The result is the same if volatile is declared for an external variable. If more than one reference must be made to a peripheral input-output register, make the references in different expressions.

(2) If volatile is not declared for an external variable to be shared in tasks and interrupts, such an external variable is a variable on the memory.

Combining consecutive accesses does not change the effect of a program. If, however, an external variable to be referenced in a loop is allocated to the register, changing the variable in other tasks and interrupt processing may not influence the loop processing and may consequently change the operation.

To disable such optimizations, specify Level 2 in the "Global variables" tab of the Optimize Details option (figure 6.9).



Level : Level 2

Contents :

□ Treat global variables as volatile qualified

□ Delete assignment to global variables before an infinite loop

Specify optimizing range : No block

Allocate registers to global variables : Disable

Propagate variables which are const qualified : Disable

Schedule instructions : Enable

Figure 6.9 Optimize Details Option Level 2

However, Level 2 disables other optimizations as well. Retain the effects of a volatile declaration and perform a maximum range of other optimizations as shown below (figure 6.10).

(a) Set Level to "Level 2."

(b) Then, set Level to "Custom" (the Level 2 settings will remain).

(c) Set "Specify optimizing range" to "No loop" (i.e., disable the optimization of external variables in a loop.If the default setting "No block" is retained, optimizations are disabled on those in all the structures including a loop).

(d) Set "Allocate registers to global variables" to "Enable."

(e) Set "Propagate variables which are const qualified" to "Enable."

Example:

Source code

```
extern int  x; /* This may be
                     changed due to
                     an interrupt. */

void f (void)
{
  x=1;
  while (1){
    if (x!=1) break;
  }
}
```

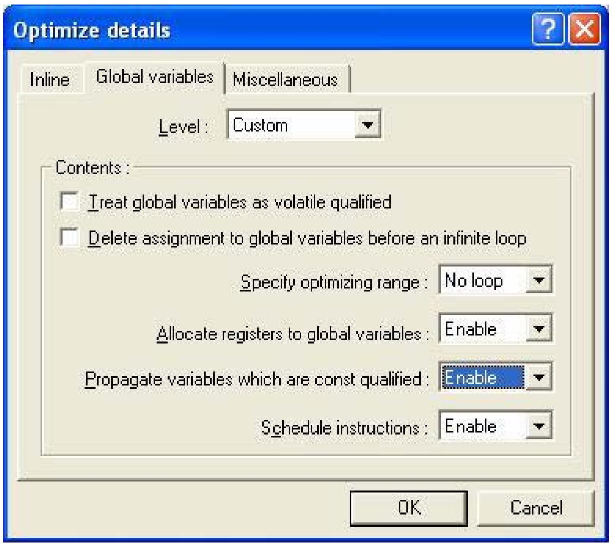
Assembler expansion code

```
_f
L10:
          BRA         L10
          NOP
     RTS
          NOP
```

Assembler expansion code (with the option specified)

```
_f
          MOV.L       L13,
          MOV         #1,R2
          MOV.L       R2,@R1
          MOV.L       @R1,R0
          CMP/EQ      #1,R0
          BT          L11
          RTS
          NOP
```

The source code assumes that, if the option is not specified, the loop will be broken when an interrupt changes external variable x. However, optimizations may make it an infinite loop because volatile is not declared for x.



Optimize details

Level : Custom

Contents :

□ Treat global variables as volatile qualified

□ Delete assignment to global variables before an infinite loop

Specify optimizing range : No loop

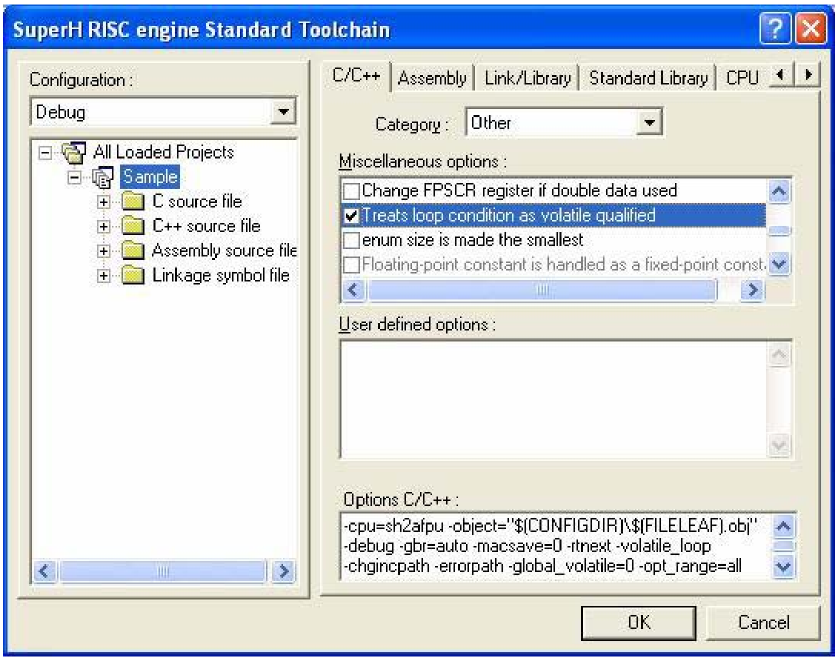
Allocate registers to global variables : Enable

Propagate variables which are const qualified : Enable

Schedule instructions : Enable

Figure 6.10 Volatile Specification for a Variable to be Shared in Tasks and Interrupts

If a reference to a variable to be shared in tasks and interrupts is limited to a loop conditional expression, keep the Optimize Details option as default (Level 3) and specify "Treat loop condition as volatile qualified" in the Other option to achieve an equivalent effect to the above and minimize the range of optimizations to be disabled (figure 6.11).



SuperH RISC engine Standard Toolchain

Configuration : Debug

Category : Other

Miscellaneous options :

□ Change FPSCR register if double data used

☑ Treats loop condition as volatile qualified

□ enum size is made the smallest

□ Floating-point constant is handled as a fixed-point const

User defined options :

Options C/C++ :

```
-cpu=sh2afpu -object="$(CONFIGDIR)\$(FILELEAF).obj"
-debug -gbr=auto -macsave=0 -rtnext -volatile_loop
-chgincpath -errorpath -global_volatile=0 -opt_range=all
```

Figure 6.11 Declaring Volatile for Loop Conditions

Example:

Source code

```
extern int  x; /* This may be
                     changed due to
                     an interrupt. */

void f (void)
{
  x=1;
  while (x){    /* A shared variable
                   is accessed in a
       conditional expression. */
  }
}
```

Assembler expansion code

```
_f:
L10:
          BRA          L10
          NOP
          RTS
          NOP
```

Assembler expansion code (with the option specified)

```
_f:
          MOV.L        L13,R1
          MOV          #1,R2
          MOV.L        R2,@R1
L11:
          MOV.L        @R1,R2
          TST          R2,R2
          BF           L11
          RTS
          NOP
```

In this example, a variable to be shared in a task is evaluated in a loop conditional expression. Thus, define the loop condition as volatile to avoid an infinite loop.

Note: The method described in item (2) of this section ensures that a variable to be shared in tasks and interrupts is referenced every time a loop is entered. This will cause a variable to be referenced in each loop iteration and consequently the value of a variable changed in other tasks or interrupts to be correctly reflected.
However, two references to a variable in a section not including a loop may be optimized at this setting. If you have such references and need to correctly reflect the value of a variable changed in other tasks or interrupts, specify an option described in item (1), "If volatile is not declared for a peripheral input-output register."

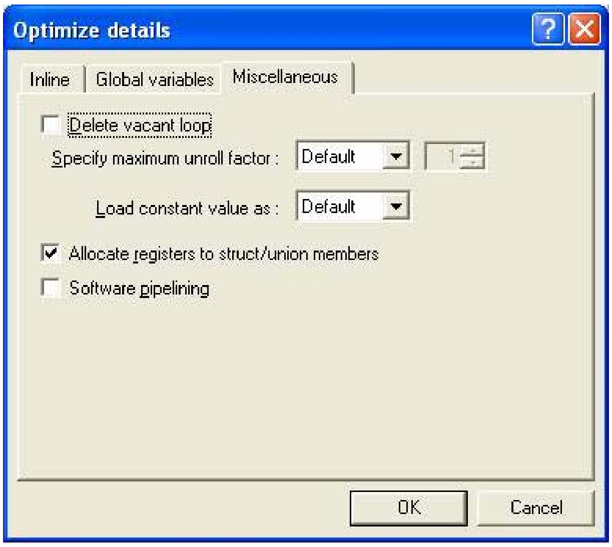
Example:

```
extern int x;
void f(void){
  int a;
  a=x;
  /* Long processing without a loop */
  a=x;
}
```

If you need to detect, in such a program, that a task or interrupt has changed x between the two references to it, specify an option described in item (1), "If volatile is not declared for a peripheral input-output register."

6.1.5 Disabling Deletion of Empty Loops

Empty loops that you wrote in a program to provide timing may be deleted in V7.0 or later due to optimizations. To disable deletion, click the "Details..." button in the Optimize menu of the Compiler option. In the Details option dialog box, select the "Miscellaneous" tab and make sure that "Delete vacant loop" checkbox is OFF. (It is OFF by default.) (Figure 6.12)



□ Delete vacant loop

Specify maximum unroll factor : Default ▼ 1

Load constant value as : Default ▼

☑ Allocate registers to struct/union members

□ Software pipelining

Figure 6.12 Disabling Deletion of Empty Loops

Example:

Source code

```
void f (void)
{
  int x;
  for (x=0; x<100; x++){
    /* Timing loop */
  }
}
```

Assembler expansion code

```
_f:
          RTS
    NOP
```

Assembler expansion code (with the option specified)

```
_f:
          MOV          #100,R2
L11:
          DT           R2
          BF           L11
          RTS
          NOP
```

Since, in this example, there is no processing inside the timing loop, the code may be deleted unless you disable deletion of empty loops.

Note: To avoid deletion of a loop, you can either access in the loop a variable with a volatile declaration or call in the loop built-in function nop().In these cases, check this option to enhance loop optimization.

6.1.6 Disabling Optimization of const Variables

The optimization processing may optimize variables for which const is declared by replacing it with a constant. This will not change the program operations. However, changing the value of a variable with a const declaration during debugging, for example, will not influence the program.
Since the optimization processing replaces const-declared variable *a* with its initial value, changing the value of a during debugging will not change the program operations.

To disable such optimizations, click the "Details..." button in the Optimize menu of the Compiler option. In the Details option dialog box, select the "Global variables" tab and specify as shown below (figure 6.13).

(a) Set Level to "Level 3" (default).

(b) Then, set Level to "Custom" (the Level 3 settings will remain).

(c) Set "Propagate variables which are const qualified" to "Disable."

Example:

Source code

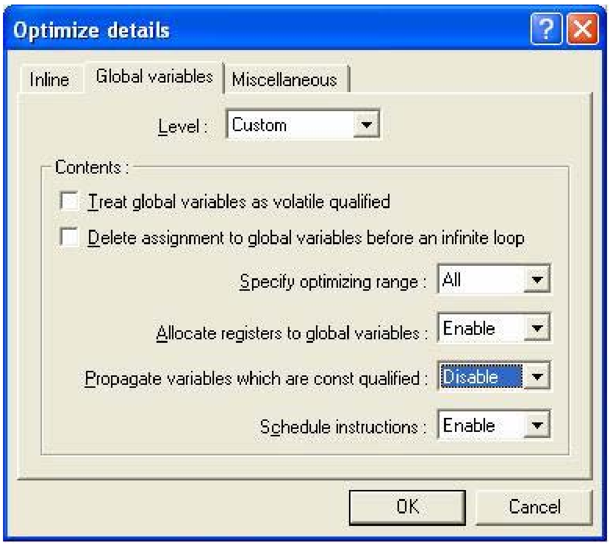
```
extern int x;
const int a=1;

void f (void)
{
  x=a;
}
```

Assembler expansion code

```
_f:
          MOV.L       L11,R6
          MOV         #1,R2
          RTS
          MOV.L       R2,@R6
```

Assembler expansion code (with the option specified)

```
_f:
          MOV.L       L11+2,R6
          MOV.L       @R6,R2
          MOV.L       L11+6,R6
          RTS
          MOV.L       R2,@R6
```



Level : Custom

Contents :

□ Treat global variables as volatile qualified

□ Delete assignment to global variables before an infinite loop

Specify optimizing range : All

Allocate registers to global variables : Enable

Propagate variables which are const qualified : Disable

Schedule instructions : Enable

Figure 6.13 Disabling Optimization of Const Variables

6.1.7 Options Effective for Enhancing Execution Efficiency of Floating Points

(1) Optimization that replaces divisions by floating-point constants with multiplications

This optimization replaces divisions by floating-point constants with multiplications by reciprocals of the constants.

This optimization is provided not in the Optimize menu but in the Other menu of the C/C++ Compiler because the resultant value may be different (although it is within the range of error). In this menu, select "Approximate floating-point constant division" (figure 6.14).

Example:

Source code

```
float x;

void f (float y)
{
  x=y/3.0;
}
```

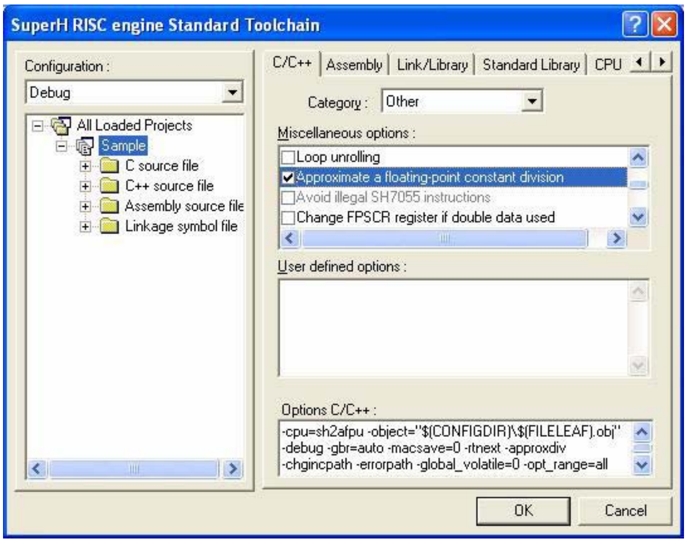
Assembler expansion code

```
_f:
          MOVA          L11,R0
          MOV.L         L11+4,R2
    FMOV.S      @R0,FR8
          FDIV          FR8,FR4
          RTS
          FMOV.S        FR4,@R2
```

Assembler expansion code (with the option specified)

```
_f:
          MOVA          L11,R0
          MOV.L         L11+4,R2
          FMOV.S        @R0,FR8
          FMUL          FR8,FR4
          RTS
          FMOV.S        FR4,@R2
```

An operation of division by 3.0 is replaced with a faster multiplication operation. (The result may be different although it is within the range of error.)



SuperH RISC engine Standard Toolchain

Configuration : Debug

- All Loaded Projects
  - Sample
    - C source file
    - C++ source file
    - Assembly source file
    - Linkage symbol file

C/C++ | Assembly | Link/Library | Standard Library | CPU

Category : Other

Miscellaneous options :

□ Loop unrolling
☑ Approximate a floating-point constant division
□ Avoid illegal SH7055 instructions
□ Change FPSCR register if double data used

User defined options :

Options C/C++ :

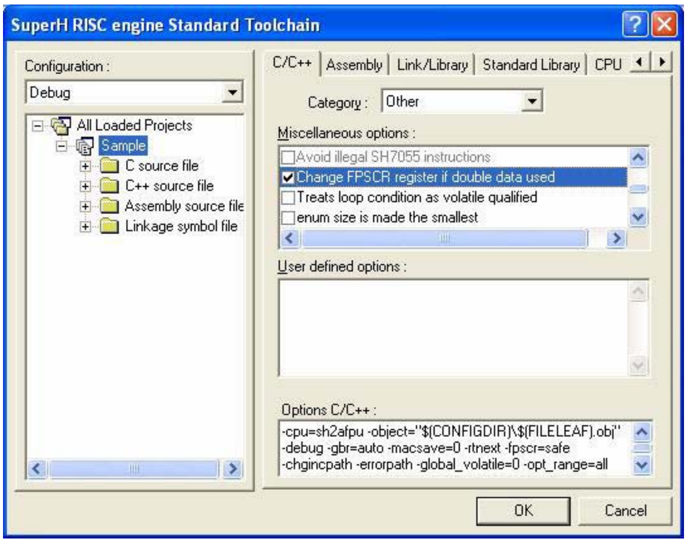
```
-cpu=sh2afpu -object="$(CONFIGDIR)\$(FILELEAF).obj"
-debug -gbr=auto -macsave=0 -rtnext -approxdiv
-chgincpath -errorpath -global_volatile=0 -opt_range=all
```

Figure 6.14 Optimization of Divisions by Floating-Point CSonstants

(2) Precaution for when Mix is selected in the floating-point setting of SH2A-FPU, SH-4, and SH-4A

If you specify Mix in the floating-point setting of SH2A-FPU, SH-4, and SH-4A, the same calling interfaces as the earlier versions of the Compiler will be used for the sake of compatibility. Since, on this interface, the floating-point setting will become undefined upon the return from a function so that FPSCR must be reset every time this happens.

Specify "Change FPSCR register if double data is used" in the Other option of the C/C++ Compiler to limit the switching of FPSCR to before and after double-precision operations and thus improve the size and speed of a program.



SuperH RISC engine Standard Toolchain

Configuration : Debug

- All Loaded Projects
  - Sample
    - C source file
    - C++ source file
    - Assembly source file
    - Linkage symbol file

C/C++ | Assembly | Link/Library | Standard Library | CPU

Category : Other

Miscellaneous options :

□ Avoid illegal SH7055 instructions
☑ Change FPSCR register if double data used
□ Treats loop condition as volatile qualified
□ enum size is made the smallest

User defined options :

Options C/C++ :

```
-cpu=sh2afpu -object="$(CONFIGDIR)\$(FILELEAF).obj"
-debug -gbr=auto -macsave=0 -rtnext -fpscr=safe
-chgincpath -errorpath -global_volatile=0 -opt_range=all
```

Figure 6.15 Recommended Option for when Mix Is Selected in the Floating-Point Setting of SH-4

Note: Since this option changes the interface of a function, the change must be made on all the files at the same time.

6.2 Optimization of Division by Constant

- Important Points:

The optimization processing turns a division by a constant into an operation other than a division. Therefore, use a division by a constant wherever possible.

- Description:

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

- Example of Use:

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

Source code before optimization

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

Assembly code before optimization

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

Source code after optimization

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

Assembly code after optimization

```
_f:
          MOV.L       L11,R2
          DMULS.L     R4,R2
          STS         MACH,R6
          MOV         R6,R0
          ROTL        R0
          AND         #1,R0
          ADD         R6,R0
          MOV.L       L11+4,R6
          RTS
          MOV.L       R0,@R6
L11:
          .DATA.L     H'55555556
          .DATA.L     _x
```

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

6.3 Size of Division by Integer

● Important Points:

A division by an integer can be executed faster as a data type (char or short) than as an int type (32-bit) because the former is shorter.

● Description:

For a division, a different runtime routine is available for each of the 32-bit, 16-bit, and 8-bit sizes.A division can be executed faster as a smaller-sized type if the range of values is limited.

● Example of Use:

If, as shown in the following example of improvement, the divisor, dividend, and result are in a 16-bit range, declaring the division operand and result as short types will call a 16-bit division routine (divws), not a 32-bit division routine (divls).

Source code before optimization

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

Assembly code before optimization

```
_f:
          STS.L        PR,@-R15
          MOV.L        L11+2,R6
          MOV.L        L11+6,R4
          MOV.L        L11+10,R2
          MOV.L        @R6,R0
          JSR          @R2
          MOV.L        @R4,R1
          MOV.L        L11+14,R6
          LDS.L        @R15+,PR
          RTS
          MOV.L        R0,@R6
L11:
          .RES.W       1
          .DATA.L      _z
          .DATA.L      _y
          .DATA.L      __divls
          .DATA.L      _x
```

Source code after optimization

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

Assembly code after optimization

```
_f
          STS.L        PR,@-R15
          MOV.L        L11+2,R6
          MOV.L        L11+6,R4
          MOV.L        L11+10,R2
          MOV.W        @R6,R0
          JSR          @R2
          MOV.W        @R4,R1
          MOV.L        L11+14,R6
          LDS.L        @R15+,PR
          RTS
          MOV.W        R0,@R6
L11:
          .RES.W       1
          .DATA.L      _z
          .DATA.L      _y
          .DATA.L      __divws
          .DATA.L      _x
```

6.4 Register Declaration

● Important Points:

The SH C/C++ Compiler Ver.7 or later allocates variables in the same way whether or not there are register declarations.

● Description:

The Compiler weighs the frequencies in use of local variables (by giving a higher priority to ones that appear in a loop, etc.) and accordingly allocates registers. Register declarations for variables are ignored.

● Example of Use:

The following shows an example of compiling programs without and with register declarations. Both the programs generate a code with the same register allocation.

Source code without a register declaration

```
int a[10];
int f(){
  int i;
  int s=0;
  for (i=0; i<10; i++)
    s+=a[i];
  return s;
}
```

Assembly code

```
_f:
          MOV         #0,R2
          MOV.L       L13,R5
          MOV         #5,R4
L11:
          MOV.L       @R5,R6
          DT          R4
          ADD         R6,R2
          MOV.L       @(4,R5),R6
          ADD         #8,R5
          BF/S        L11
          ADD         R6,R2
          RTS
          MOV         R2,R0
L13:
          .DATA.L     _a
```

Source code with a register declaration

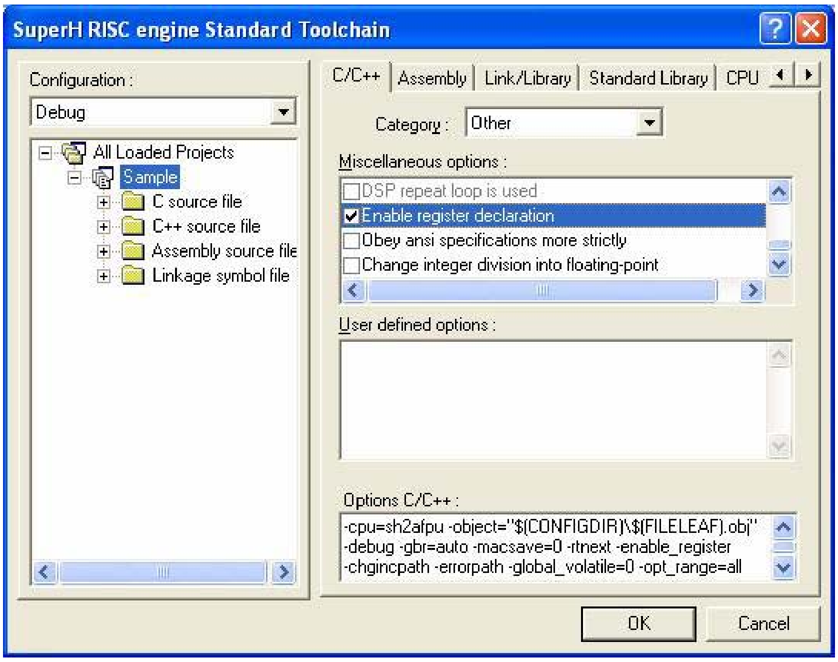
```
int a[10];
int f(){
  register int i;
  register int s=0;
  for (i=0; i<10; i++)
    s+=a[i];
  return s;
}
```

Assembly code

```
_f
          MOV         #0,R2
          MOV.L       L13,R5
          MOV         #5,R4
L11:
          MOV.L       @R5,R6
          DT          R4
          ADD         R6,R2
          MOV.L       @(4,R5),R6
          ADD         #8,R5
          BF/S        L11
          ADD         R6,R2
          RTS
          MOV         R2,R0
L13:
          .DATA.L     _a
```

Notes: 1. When the Compiler automatically allocates variables to registers, effective register allocation will be difficult if too many local variables are used in a function. It is recommended to split up functions appropriately until about eight or fewer local variables are used in a loop.

2. In the Ver.9 compiler or later, by selecting the enable_register option, variables with the register storage class specification can be allocated preferentially to the registers. (The enable_register option isn't selected in default in Ver.9 compiler or later.)



Configuration : Debug

Category : Other

Miscellaneous options :

□ DSP repeat loop is used
☑ Enable register declaration
□ Obey ansi specifications more strictly
□ Change integer division into floating-point

Options C/C++ :

```
-cpu=sh2afpu -object="$(CONFIGDIR)\$(FILELEAF).obj"
-debug -gbr=auto -macsave=0 -rtnext -enable_register
-chgincpath -errorpath -global_volatile=0 -opt_range=all
```

Figure 6.16 enable_register Option Specification

6.5 Offset of Member in Structure Declaration

● Important Points:

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

● Description:

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

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

● Example of Use:

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

Source code before optimization

```
struct S{
  int a[100];
  int x;
};

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

Assembly code before optimization

```
_f:
          MOV         #100,R0
          SHLL2       R0
          RTS
          MOV.L       @(R0,R4),R0
```

Source code after optimization

```
struct S{
  int x;
  int a[100];
};

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

Assembly code after optimization

```
_f:
          RTS
          MOV.L       @R4,R0
```

6.6 Allocation of Bit Fields

● Important Points:

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

● Description:

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

● Example of Use:

In the following example, related bit fields are allocated to the same structure, thus improving both the speed and size.

Source code before optimization

```
struct bits{
  unsigned int b0: 1;
} f1, f2;

int f(void){
  if (f1.b0 && f2.b0) return 1;
  else return 0;
}
```

Assembly code before optimization

```
_f:
          MOV.L       L15,R6
          MOV.B       @R6,R0
          TST         #128,R0
          BT          L12
          MOV.L       L15+4,R6
          MOV.B       @R6,R0
          TST         #128,R0
          BT          L12
          RTS
          MOV         #1,R0
L12:
          RTS
          MOV         #0,R0
L15:
          .DATA.L     _f1
          .DATA.L     _f2
```

Source code after optimization

```
struct bits{
  unsigned int b0: 1;
  unsigned int b1: 1;
} f1;

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

Assembly code after optimization

```
_f:
          MOV.L       L11,R6
          MOV         #-64,R3
          EXTU.B      R3,R3
          MOV.B       @R6,R0
          AND         #192,R0
          CMP/EQ      R3,R0
          RTS
          MOVT        R0
L11:
          .DATA.L     _f1
```

6.7 Software Pipeline (Floating-Point Table Search)

● Important Points:

You can improve the execution speed of the table search code by designing it to compare data items referenced from a table with the data items loaded in the previous loop iteration, instead of immediately compare them.

● Description:

Pipeline optimization cannot perform sufficient optimization of a loop with a few instructions such as table search because there is no little room for rearranging the instructions. For floating-point table search, for example, the FCMP operation must wait until the completion of load if the program performs comparison immediately after loading data from the table. To work around this problem, design the program so that it loads comparison data into local variables in one loop iteration and then compares them in the next iteration.

● Example of Use:

The code before improvement compares the loaded floating-point data (FR8) using the FCMP instruction written immediately after it. If the code is improved so that the data referenced in one loop iteration will be compared in the next iteration, the load is executed in parallel with a branch instruction of the loop.

Source code before optimization

```
float a[100];

int f(float b){
  int i=0;
  float *p=a;
  while (i<100){
    if (*p==b) return i;
   i++;
   p++;
  }
  return -1;
}
```

Assembly code before optimization

```
_f:
          MOV         #0,R5
          MOV.L       L16,R2
          MOV         #100,R6
L11:
          FMOV.S      @R2,FR8
          FCMP/EQ     FR4,FR8
          BT          L12
          DT          R6
          ADD         #1,R5
          BF/S        L11
```

Source code after optimization

```
float a[100];

int f(float b){
  int i=0;
  float *p=a;
  float tmp=*p;
  while (i<100){
    if (tmp==b) return i;
   i++;
   p++;
   tmp=*p;
  }
  return -1;
}
```

Assembly code after optimization

```
_f:
          MOV.L       L16+2,R2
          MOV         #0,R5
          MOV         #100,R6
          FMOV.S      @R2,FR8
L11:
          FCMP/EQ     FR4,FR8
          BT          L12
          ADD         #4,R2
          DT          R6
          FMOV.S      @R2,FR8
          ADD         #4,R2
          RTS
          MOV         #-1,R0
L12:
          RTS
          MOV         R5,R0
L16:
          .DATA.L     _a
```

```
          BF/S        L11
          ADD         #1,R5
          RTS
          MOV         #-1,R0
L12:
          RTS
          MOV         R5,R0
L16:
          .RES.W      1
          .DATA.L     _a
```

6.8 Ensuring of Data Access Size

● Important Points:

Declare volatile to ensure the size (byte, word, long word) of a memory access instruction used to access a peripheral register.

● Description:

Declare volatile to ensure the size of an instruction that accesses global variables and pointers. This will cause the instruction to load and store data at the size of the data type. Declare volatile to access bit fields in order to access them using the data type used when the bit fields are declared. Unless volatile is declared, the access to the bit fields will be optimized, possibly causing accesses with other type than the declared one.

● Example of Use:

If volatile is not declared, member x is accessed as byte access. If volatile is declared, it is accessed as the declared type (word).

Source code without volatile specification

```
struct S{
  short x: 8;
  short y: 8;
} *p;

int f(){
  return p->x;
}
```

Assembly code

```
_f:
          MOV.L       L11+2,R2
          MOV.L       @R2,R6
          MOV.B       @R6,R2
          RTS
          EXTS.W      R2,R0
L11:
          .RES.W      1
          .DATA.L     _p
```

Source code with volatile specification

```
volatile struct S{
  short x: 8;
  short y: 8;
} *p;

int f(){
  return p->x;
}
```

Assembly code

```
_f:
          MOV.L       L11,R2
          MOV.L       @R2,R6
          MOV.W       @R6,R2
          SHLR8       R2
          RTS
          EXTS.B      R2,R0
L11:
          .DATA.L     _p
```

6.9 Use of Floating-Point Instructions

● Important Points:

To use single-precision floating-point instructions FABS (SH2-E, SH2A-FPU, SH-4, and SH-4A) and FSQRT (SH2A-FPU, SH-4, SH-4A), include the include file <mathf.h> and call single-precision floating-point functions fabsf and sqrtf.

● Description:

Use single-precision floating-point instructions FABS (SH2-E, SH2A-FPU, SH-4, and SH-4A) and FSQRT (SH2A-FPU, SH-4, SH-4A) as follows:

(a) Include <math.h>.

(b) Call fabsf function (FABS) and sqrtf function (FSQRT).

● Example of Use:

In the example before improvement, <mathf.h> is not included and thus the Compiler calls the *fabsf* function from the library, not recognizing it as a standard function. If <mathf.h> is included, the Compiler recognizes it as a function corresponding to the FABS instruction and thus directly generates the FABS instruction.

Source code before optimization

```
float fabsf(float);

float f(float x, float y){
  return fabsf(x)+fabsf(y);
}
```

Assembly code before optimization

```
_f:
          STS.L       PR,@-R15
          FMOV.S      FR14,@-R15
          FMOV.S      FR15,@-R15
          MOV.L       L12+2,R2
          JSR         @R2
          FMOV.S      FR5,FR15
          MOV.L       L12+2,R2
          FMOV.S      FR0,FR14
          JSR         @R2
          FMOV.S      FR15,FR4
          FADD        FR0,FR14
          FMOV.S      FR14,FR0
          FMOV.S      @R15+,FR15
          FMOV.S      @R15+,FR14
          LDS.L       @R15+,PR
          RTS
          NOP
L12:
```

Source code after optimization

```
#include <mathf.h>

float f(float x, float y){
  return fabsf(x)+fabsf(y);
}
```

Assembly code after optimization

```
_f:
          FABS        FR4
          FABS        FR5
          FADD        FR5,FR4
          RTS
          FMOV.S      FR4,FR0
          .RES.W       1
          .DATA.L      _fabsf
```

Note: Header <mathf.h> is not a standard C library function of ANSI.

Section 7 Using HEW

This chapter describes the use of HEW for build- and simulation-related processes.
Note that the supported functions and methods vary from one HEW version to another.
The appropriate version is indicated under "• Note:" for each topic.

The following table shows a list of the items relating to the use of HEW.

No.	Category	Item	Section
1	Builds	Regenerating and Editing Automatically Generated Files	7.1.1
2		Makefile Output	7.1.2
3		Makefile Input	7.1.3
4		Creating Custom Project Types	7.1.4
5		Multi-CPU Feature	7.1.5
6		Networking Feature	7.1.6
7		Converting from Old HEW Version	7.1.7
8		Converting a HIM Project to a HEW Project	7.1.8
9		Add Supported CPUs	7.1.9
10	Simulations	Pseudo-interrupts	7.2.1
11		Convenient Breakpoint Functions	7.2.2
12		Coverage Feature	7.2.3
13		File I/O	7.2.4
14		Debugger Target Synchronization	7.2.5
15		How to Use Timers	7.2.6
16		Examples of Timer Usage	7.2.7
17		Reconfiguration of Debugger Target	7.2.8
18	Call Walker	Creating a Stack Information File	7.3.1
19		Starting Call Walker	7.3.2
20		Call Walker Window and Opening a File	7.3.3
21		Editing Stack Information	7.3.4
22		Stack Area Size of Assembly Program	7.3.5
23		Merging Stack Information	7.3.6
24		Other Features	7.3.7

7.1 Builds

7.1.1 Regenerating and Editing Automatically Generated Files

- Description:

HEW will automatically generate I/O register definition, interrupt function, and other various files if you select Application for the project type when creating a new workspace.

However, when creating a new project, you may sometimes skip this automatic file generation process because you then believe that the files are unnecessary.

You may also forget to edit or set such files.

If you do, you can use this feature to automatically generate and edit files after creating a project.

However, this feature is only available when you select Application for the project type when creating a new workspace.

- Usage:

HEW Menu: **Project > Edit Project Configuration...**

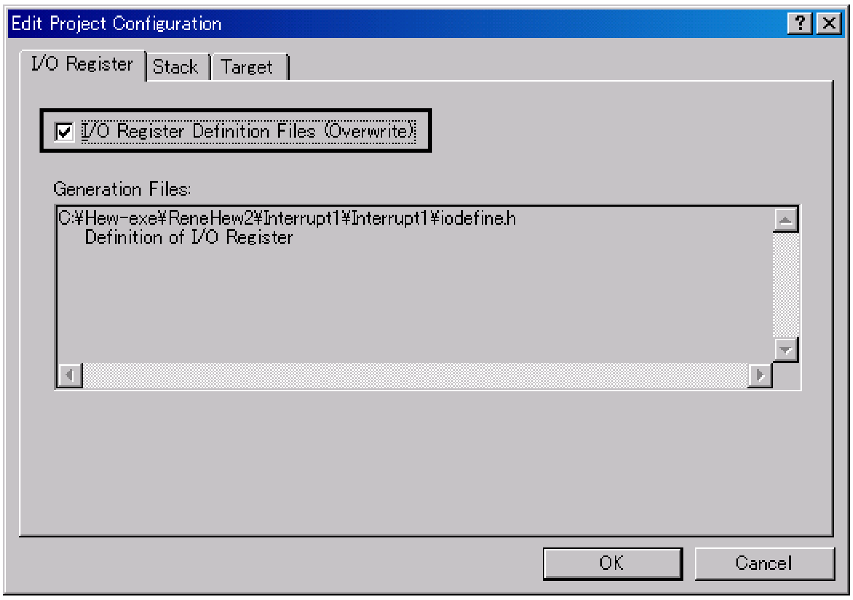
- Files that can be regenerated:

I/O Register Definition Files: iodefine.h

[Generation method]

You can regenerate iodefine.h by checking [I/O Register Definition Files (overwrite)] on the [I/O Register] tab in the [Edit Project Configuration] dialog box.

If you modify iodefine.h inadvertently, you can regenerate it and overwrite it on the modified file.



☑ I/O Register Definition Files (Overwrite)

Generation Files:

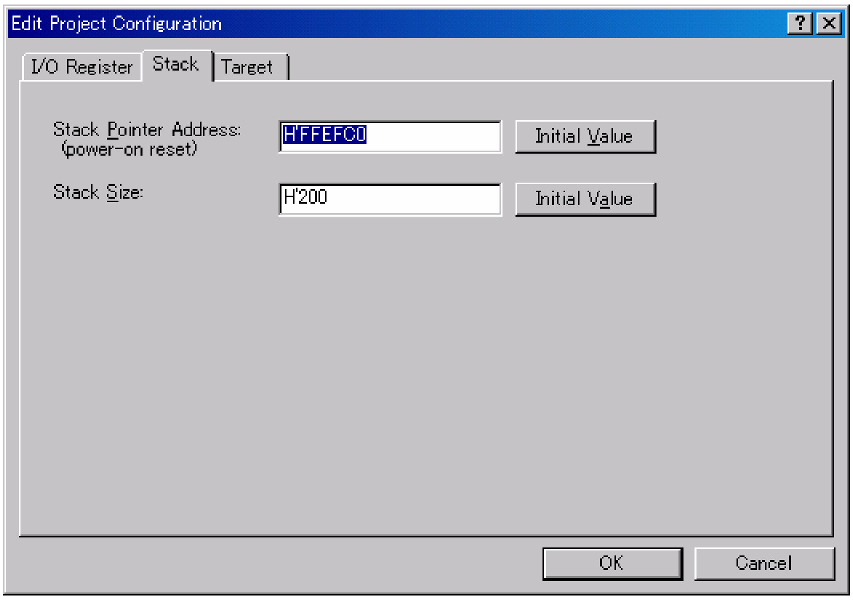
```
C:¥Hew-exe¥ReneHew2¥Interrupt1¥Interrupt1¥iodefine.h
   Definition of I/O Register
```

- Files that can be re-edited:

Stack size setting file: stacksct.h

[Editing method]

You can edit the initial values of [Stack Pointer Address] and [Stack Size] on the [Stack] tab in the [Edit Project Configuration] dialog box.



Stack Pointer Address: (power-on reset) H'FFEFC0

Stack Size: H'200

- Note:

Regenerating and re-editing files are supported by HEW 2.0 or later.

7.1.2 Makefile Output

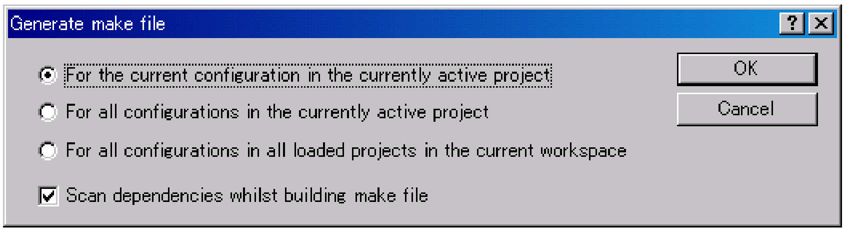
- Description:

HEW allows you to create a makefile based on the current option settings.

By using the makefile, you can build the current project without having to install HEW completely. This is convenient when you send a project to a person who has not installed HEW or manage the version of an entire build, including the makefile.

- Makefile production method:

1. Make sure that the project that generates the makefile is the current project.
2. Make sure that the build configuration that builds the project is the current configuration.
3. Choose [Build > Generate make file].
4. You will see the following dialog box. In this dialog box, select one of the makefile generation methods.



Generate make file

● For the current configuration in the currently active project
○ For all configurations in the currently active project
○ For all configurations in all loaded projects in the current workspace
☑ Scan dependencies whilst building make file

- Makefile generation directory:

HEW creates a [make] subdirectory in the current workspace directory and generates makefiles in this subdirectory. The makefile name is the current project or configuration name followed by the extension .mak (debug.mak, for example). HEW-generated makefiles can be executed by the executable file HMAKE.EXE contained in the directory where HEW is installed. However, user-modified makefiles cannot be executed.

- Makefile execution method:

1. Open the [Command] window and move to the [make] subdirectory that contains the generated makefile.
2. Execute HMAKE.On the command line, enter HMAKE.EXE <makefile-name>.

- Note:

This feature is supported by HEW 1.1 or later.

7.1.3 Makefile Input

- Description:

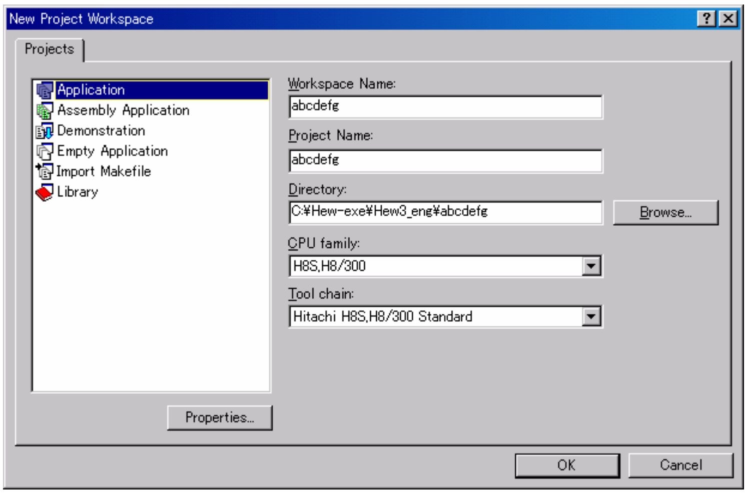
HEW allows to input the makefiles that were generated by HEW or used by UNIX environment.

From the makefile, you can automatically obtain the file structure of the project.

(However, you cannot obtain option settings or similar specifications.) This facilitates the migration from the command line to HEW.

- Makefile input method:

1. When creating a new workspace, select [Import Makefile] from the project type options in the [New Project Workspace] dialog box.



Workspace Name: abcdefg

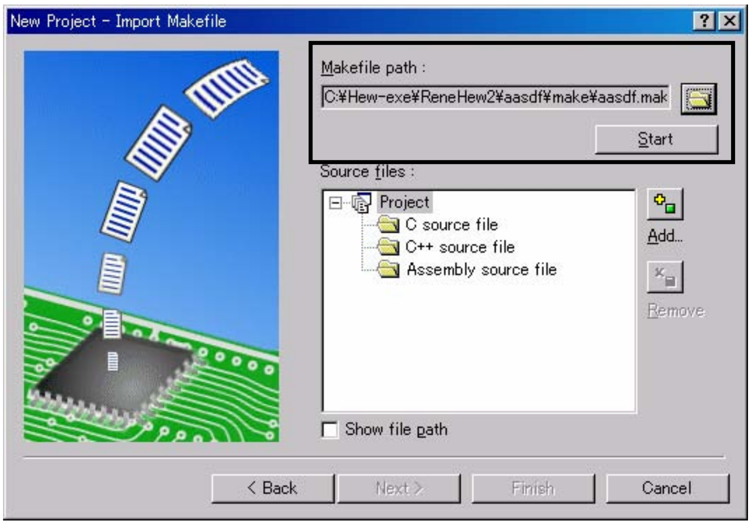
Project Name: abcdefg

Directory: C:¥Hew-exe¥Hew3_eng¥abcdefg

CPU family: H8S,H8/300

Tool chain: Hitachi H8S,H8/300 Standard

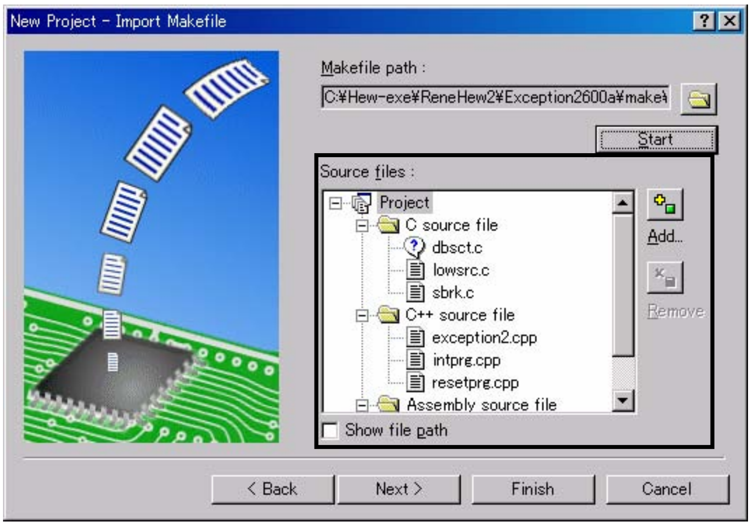
2. Specify the makefile path in the [Makefile path] field in the [New Project-Import Makefile] dialog box and click on the [Start] button.



Makefile path : C:¥Hew-exe¥ReneHew2¥aasdf¥make¥aasdf.mak

□ Show file path

3. The [Source files] pane displays the makefile source file structure. In this structure chart, any file marked ? is a file that has been proved through an analysis to contain no entity. This file will not be added to the project. (It is ignored.)



New Project - Import Makefile

Makefile path :
C:¥Hew-exe¥ReneHew2¥Exception2600a¥make¥

Source files :

```
Project
  C source file
    dbsct.c
    lowsrc.c
    sbrk.c
  C++ source file
    exception2.cpp
    intprg.cpp
    resetprg.cpp
  Assembly source file
```

□ Show file path

4. By following the wizard, specify CPU and other options and open the workspace. You can then begin a development work.

• Note:

This feature is supported by HEW 3.0 or later.

7.1.4 Creating Custom Project Types

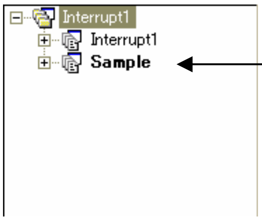
• Description:

This feature allows a project created by a user to be used by another user as a template for program development on another machine.

Information that can be contained in the template may concern the project file structure, build options, debugger settings, and anything else relating to the project.

• Project type storing method:

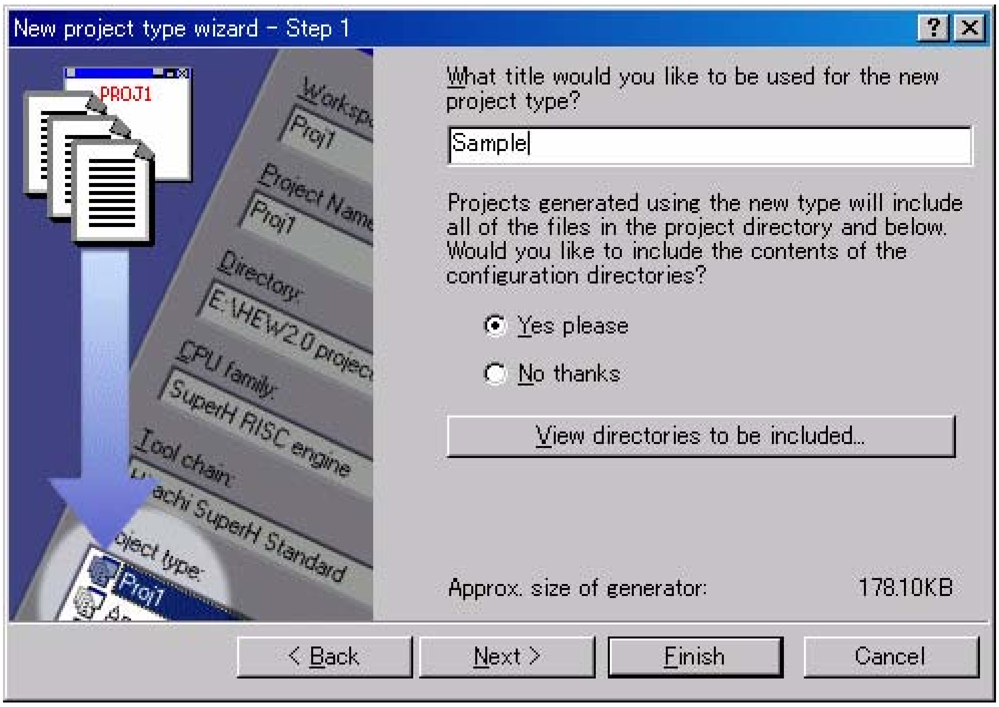
1. Activate the project you want to store project information in because the active project accepts project information when the workspace is open. To activate a project, select the project by choosing [Project -> Set Current Project].



Interrupt1
Interrupt1
Sample

The active project is identified by boldface characters.

2. Open the following project type wizard by choosing [Project -> Create Project Type...], assign a name to the project type you will use as the template and specify whether to include the configuration directory containing the post-build executable files and other resources in the template.
   You can quit the project type wizard here by clicking on the [Finish] button.



New project type wizard - Step 1

What title would you like to be used for the new project type?

Sample

Projects generated using the new type will include all of the files in the project directory and below. Would you like to include the contents of the configuration directories?

● Yes please

○ No thanks

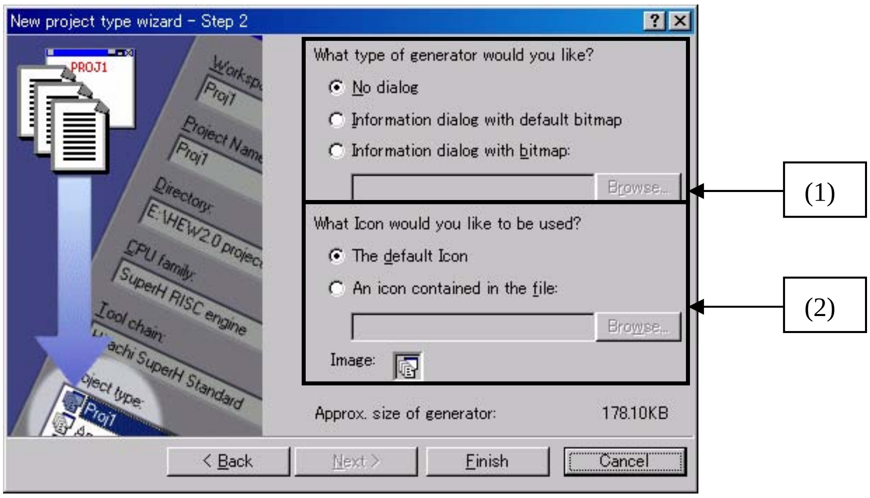
View directories to be included...

Approx. size of generator: 178.10KB

3. At [New project type wizard – Step 1], click on the [Next] button to open the following wizard: When opening the project type templateat step (1), specify whether to display project information and bitmaps.

   At step (2), you can change the project type icon to a user-specified icon. Click on the [Finish] button.

   These settings are not mandatory.



New project type wizard - Step 2

What type of generator would you like?

● No dialog

○ Information dialog with default bitmap

○ Information dialog with bitmap:

What Icon would you like to be used?

● The default Icon

○ An icon contained in the file:

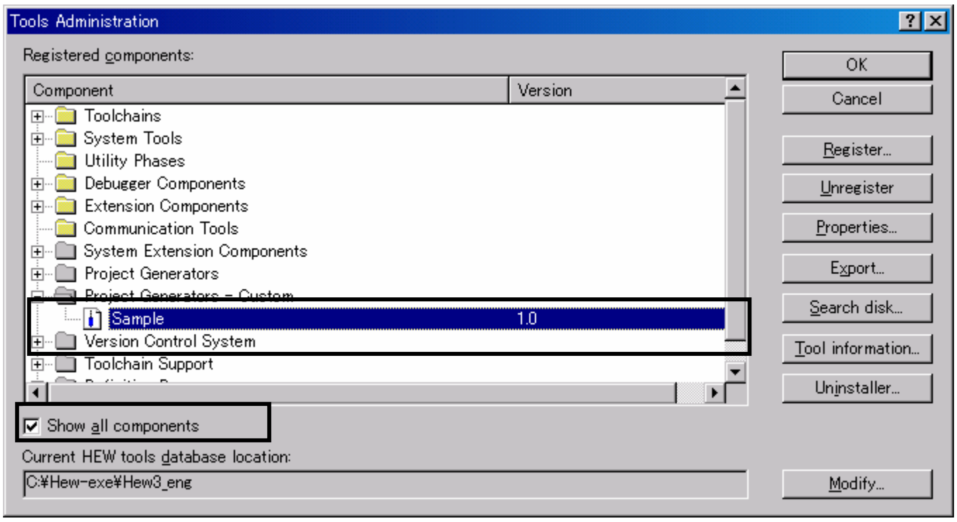
Image:

Approx. size of generator: 178.10KB

4. A project type template named “Custom Project Generator” has thus been created. To use this template on another machine, choose [Tools -> Administration...] to open the following dialog box:

   When you check the following [Show all components] check box, you will see [Project Generators – Custom].

   Click on the created project type and click on the [Export...] button.



Tools Administration

Registered components:

Component	Version
Toolchains	
System Tools	
Utility Phases	
Debugger Components	
Extension Components	
Communication Tools	
System Extension Components	
Project Generators	
Project Generators – Custom	
Sample	1.0
Version Control System	
Toolchain Support	

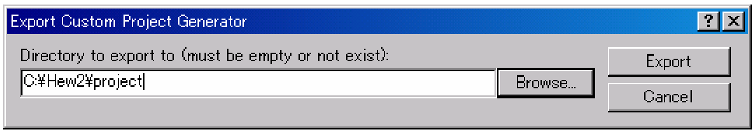
☑ Show all components

Current HEW tools database location:

C:¥Hew-exe¥Hew3_eng

5. The following dialog box opens. Select a directory in which the Custom Project Generator template will be stored. The directory must be empty.

   The project type storage process is now complete.



Export Custom Project Generator

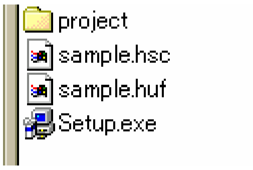
Directory to export to (must be empty or not exist): C:¥Hew2¥project

• Installing Custom Project Generator:

Use the following procedure to install the Custom Project Generator template created by the above project type storage method on another machine.

1. The following installation environment is created for the directory that was created at step 5 of the project type storage method:

   (Installation environment directory)

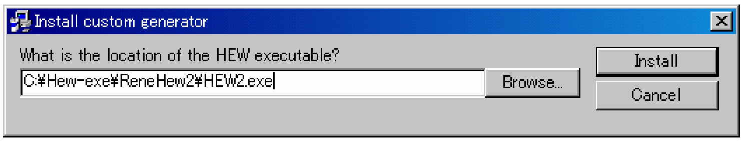


project
sample.hsc
sample.huf
Setup.exe

2. Copy the above installation environment and install the copy on another machine.

   When you run Setup.exe, the following dialog box opens. Specify the location in which HEW2.exe is installed and click on the [Install] button.

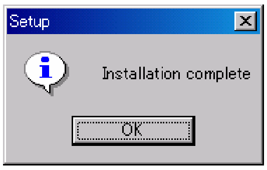
   (Directory example: c:¥Hew2¥HEW2.exe)



Install custom generator

What is the location of the HEW executable? C:¥Hew-exe¥ReneHew2¥HEW2.exe

3. The environment has been built up completely.



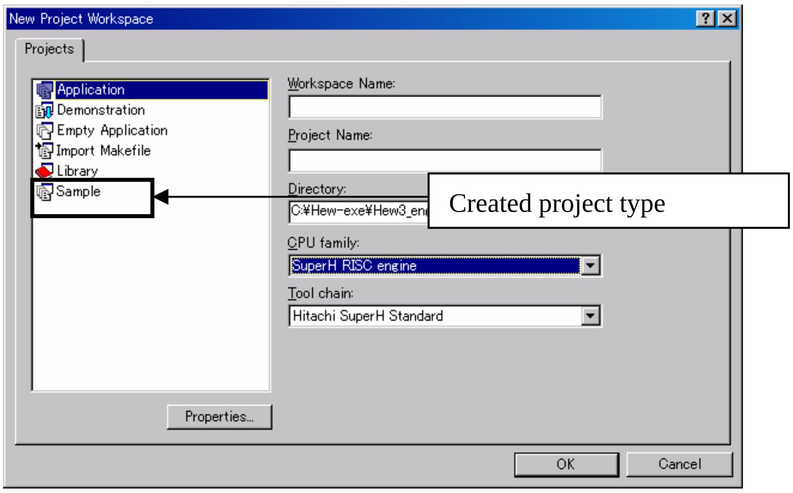
Setup

Installation complete

- Custom Project Generator usage example:

An example of using the installed Custom Project Generator template is provided below.

1. Start HEW and choose [Create a new project workspace] in the [Welcome!] dialog box. The installed project type is added to the [Projects] list. Click on the project type and click on the [OK] button.
   You can now proceed with program development using the stored project template for any new project.



New Project Workspace

Projects

Application
Demonstration
Empty Application
Import Makefile
Library
Sample

Workspace Name:

Project Name:

Directory: C:¥Hew-exe¥Hew3_en

CPU family: SuperH RISC engine

Tool chain: Hitachi SuperH Standard

Created project type

- Note:

This feature is supported by HEW 2.0 or later.

7.1.5 Multi-CPU Feature

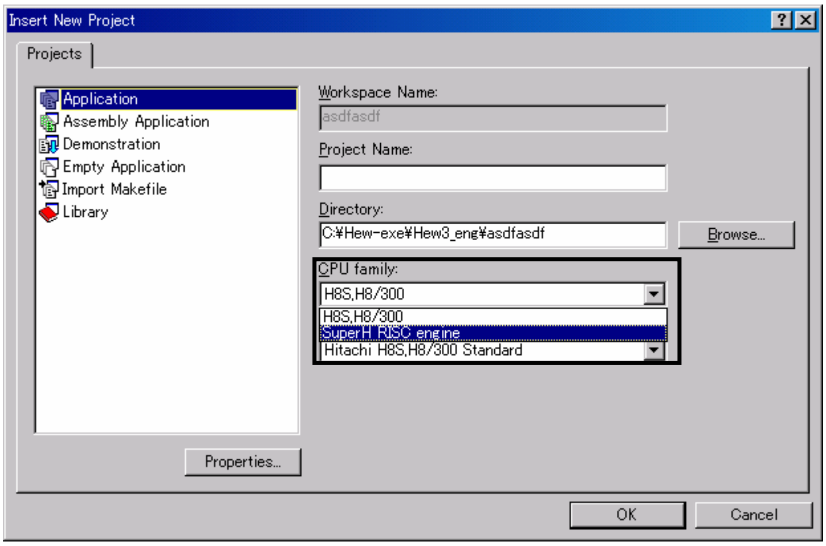
- Description:

When inserting a new project in the workspace, you can insert a CPU of another type. This enables SH and H8 projects to be managed in a single workspace.

- Example of inserting a different CPU family:

1. When an H8 (SH) project is open, click on [Project -> Insert Project...]. In the [Insert Project] dialog box, select a new project and click on the [OK] button.

2. The following [Insert New Project] dialog box appears: Select a project name, select SH (H8) as the CPU type, and click on the [OK] button. You can place different CPU types in addition to the current CPU types in the workspace.



Insert New Project

Workspace Name: asdfasdf

Project Name:

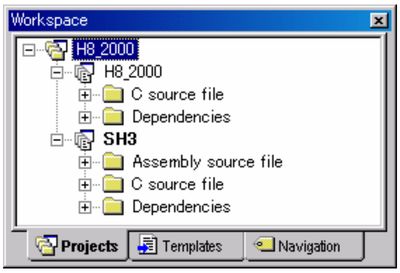
Directory: C:¥Hew-exe¥Hew3_eng¥asdfasdf

CPU family: H8S,H8/300

H8S,H8/300
SuperH RISC engine

Hitachi H8S,H8/300 Standard

3. With the procedure above, you can mix SH and H8 projects in a single workspace.



- Note:

This feature is supported by HEW 3.0 or later.

7.1.6 Networking Feature

• Description:

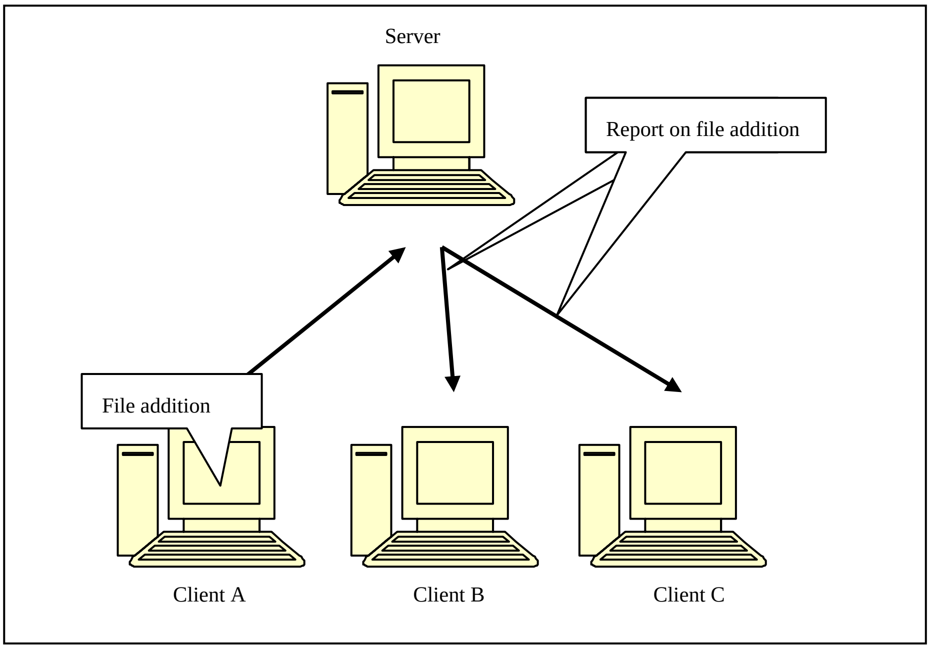
HEW allows workspaces and projects to be shared by different users via a network.

Therefore, users can learn changes that other users have made, by manipulating the shared project at the same time.

This system uses one computer as its server.

For example, if a client adds a new file to a project, the server machine is notified, and then notifies the other clients of the addition.

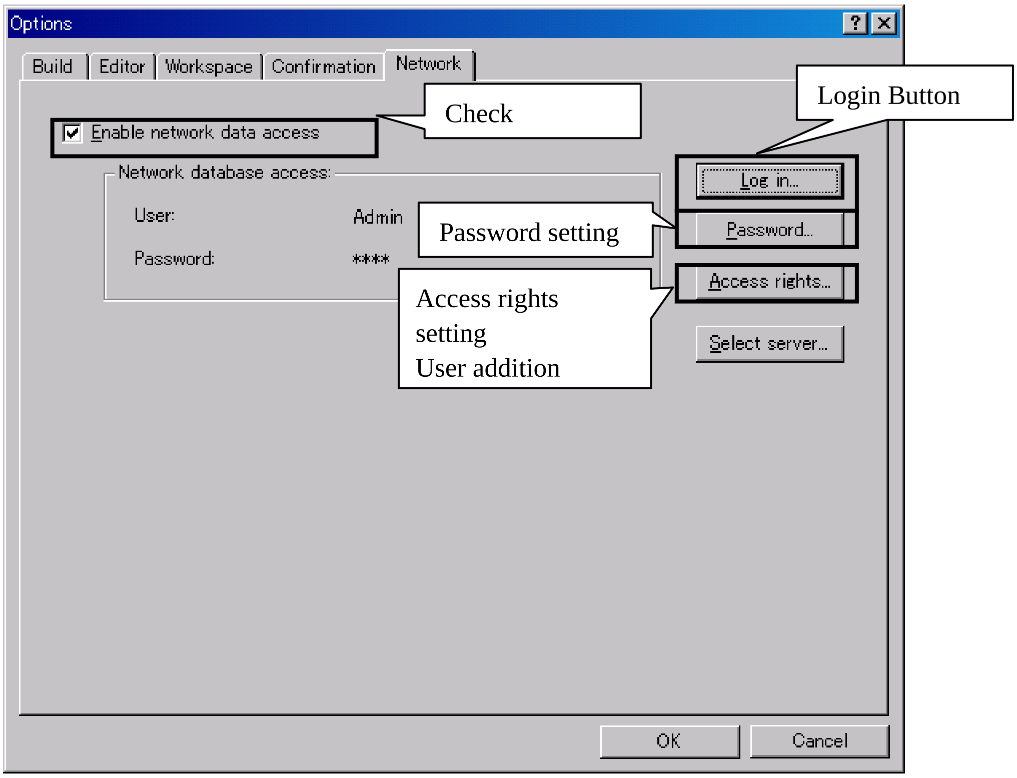
In addition, users can be granted rights for access to specific projects or files.



• Network access setup:

1. Choose [Tools -> Options...] and select the [Network] tab. Check the [Enable network data access] check box.
2. An administrator is added. Since the administrator does not have a password initially, you need to specify a password. The administrator should be granted the highest access right.
3. Click on the [Password…] button and specify a password for the administrator.
4. Click on the [OK] button. This allows the administrator access to the network.

[Network] Tab of the [Options] dialog box



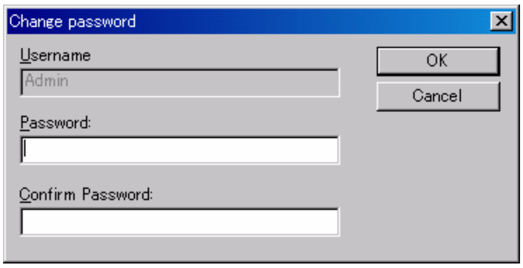
☑ Enable network data access

Network database access:

User: Admin

Password: ****

[Change password] dialog box



Username: Admin

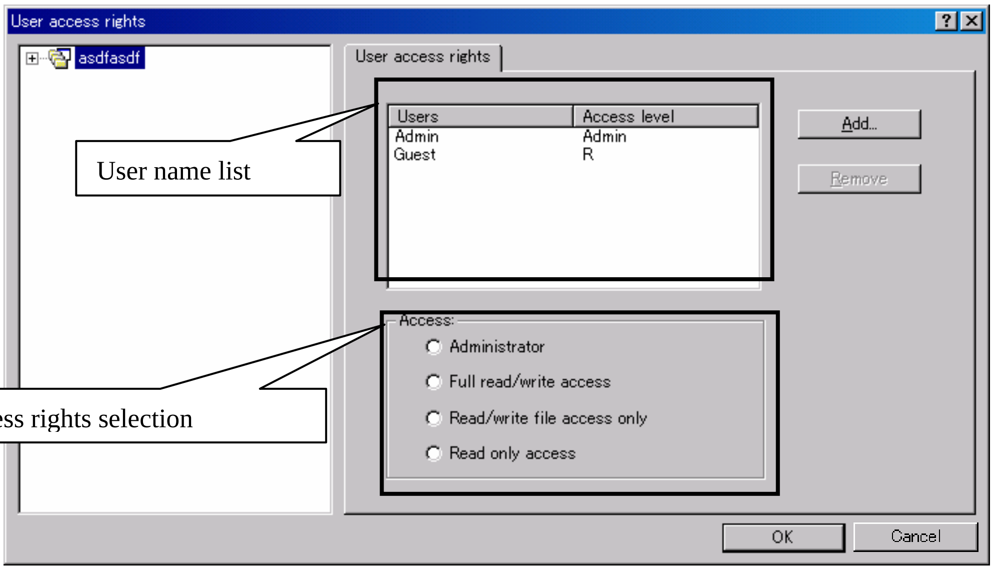
Password:

Confirm Password:

- Adding a new user:

By default, an administrator and a guest have been added. You can register new users.

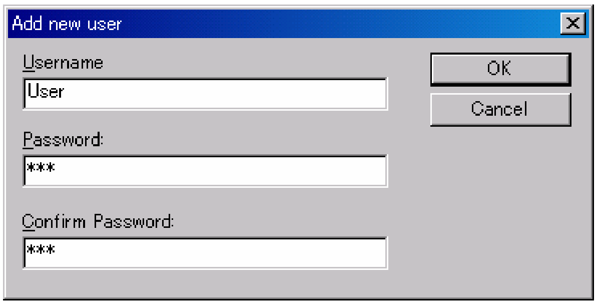
1. Click on the [Log in...] button shown on the previous page. Log in as a user granted administrator access right.
2. Click on the [Access rights…] button to open the following [User access rights] dialog box.



Users	Access level
Admin	Admin
Guest	R

Access:
- ○ Administrator
- ○ Full read/write access
- ○ Read/write file access only
- ○ Read only access

3. Click on the [Add…] button to open the [Add new user] dialog box.
4. Enter a new user name and password. (Password specification is mandatory.)



Username: User

Password: ***

Confirm Password: ***

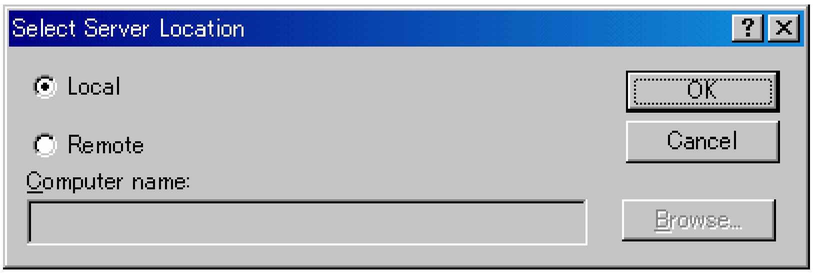
5. The new user name is then added to the user list. Select the user name and specify access right for the user.
6. Click on the [OK] button. Your specification will be put into effect.

- Selecting the server machine

Select the machine that will work as the server. If you make your own machine the server, you do not have to do anything.

If you specify another machine as the server, click on the [Select server…] button in the [Options] dialog box. Choose [Remote] in the following dialog box, and then specify a computer name.

Click on the [OK] button. Your specification will be put into effect.



- Note:

This feature is supported by HEW 3.0 or later.

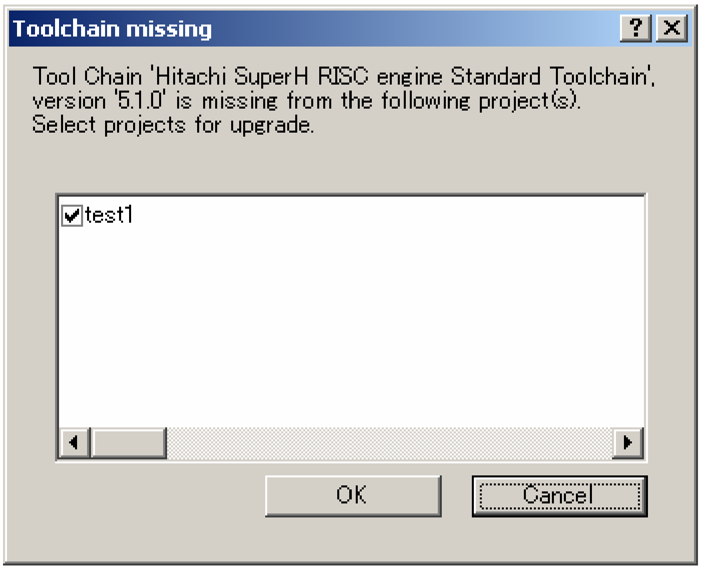
Use of this feature will lower the HEW performance.

7.1.7 Converting from Old HEW Version

Here, the method for specifying the compiler version within the Renesas Integrated Development Environment is explained. Compiler versions can be specified by upgrading the Renesas Integrated Development Environment.

If the workspace created in an old version (such as HEW1.1 or SHC5.1B) is opened in a new version (such as HEW3.0 or SHC8.0), the following dialog box appears.

(1) Checking the project to be upgraded.
Check the name of the project to be upgraded.



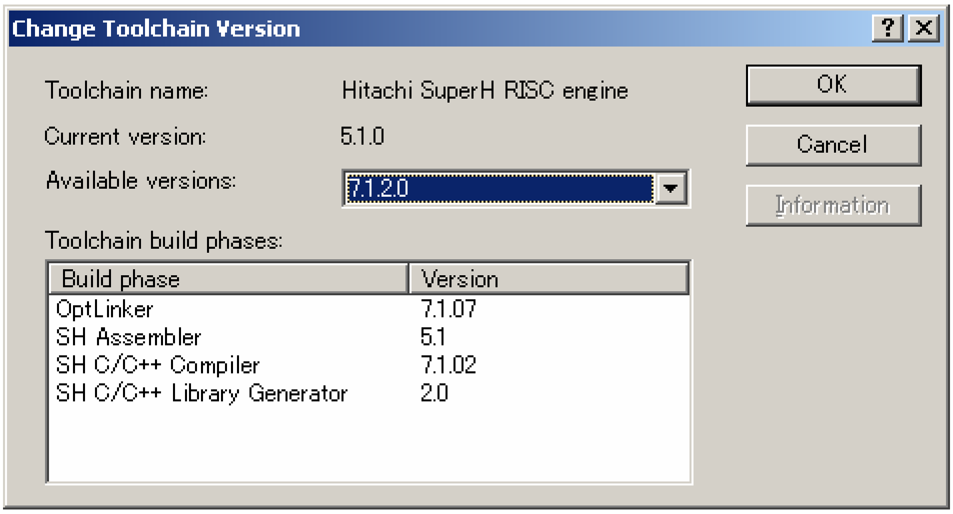
Toolchain missing

Tool Chain 'Hitachi SuperH RISC engine Standard Toolchain', version '5.1.0' is missing from the following project(s). Select projects for upgrade.

☑ test1

High-performance Embedded Workshop

(2) Specifying the Compiler Version
Select the Compiler version which can be upgraded.



Change Toolchain Version

Toolchain name: Hitachi SuperH RISC engine

Current version: 5.1.0

Available versions: 7.1.2.0

Toolchain build phases:

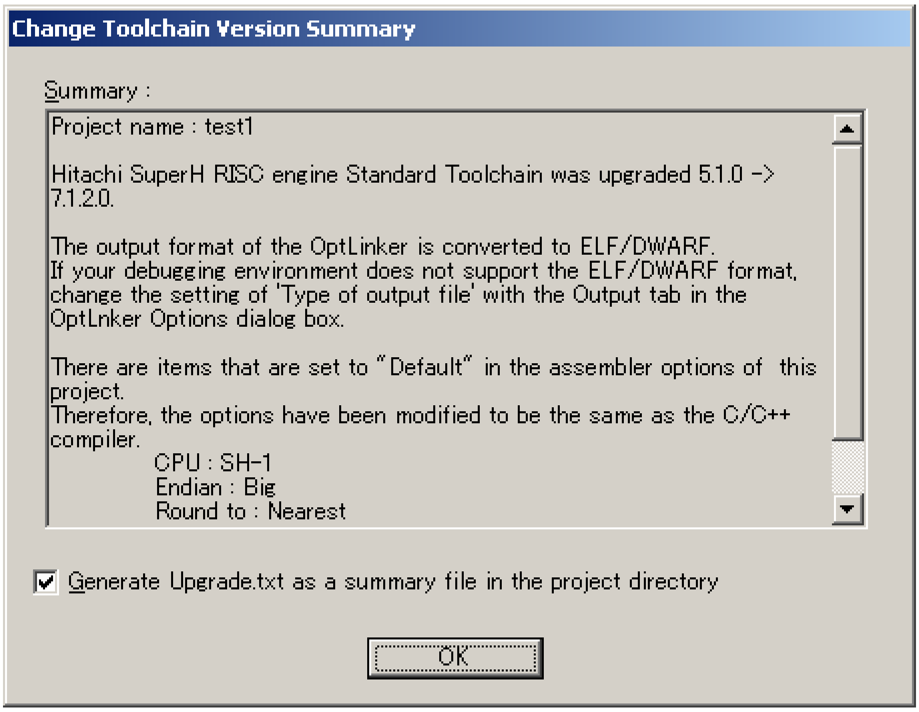
Build phase	Version
OptLinker	7.1.07
SH Assembler	5.1
SH C/C++ Compiler	7.1.02
SH C/C++ Library Generator	2.0

Change Toolchain Version Dialog Box

(3) Confirmation message

The C/C++ Compiler Ver6.0 or later versions support only the file format ELF/DWARF for the object to be output.

The file format is changed to ELF/DWARF format at upgrading. If the current debugging environment does not support the ELF/DWARF format, convert the ELF/DWARF format to the format supported by the debugging environment after upgrading.



Project name : test1

Hitachi SuperH RISC engine Standard Toolchain was upgraded 5.1.0 -> 7.1.2.0.

The output format of the OptLinker is converted to ELF/DWARF.
If your debugging environment does not support the ELF/DWARF format, change the setting of 'Type of output file' with the Output tab in the OptLnker Options dialog box.

There are items that are set to "Default" in the assembler options of this project.
Therefore, the options have been modified to be the same as the C/C++ compiler.

```
            CPU : SH-1
            Endian : Big
            Round to : Nearest
```

☑ Generate Upgrade.txt as a summary file in the project directory

Confirmation Message Dialog Log

(4) Standard Library Generator Options

After upgrading, **Standard Library** Tab Category: [**Mode**] in the **SuperH RISC engine Standard Toolchain** dialog box is changed to **Build a library file(anytime)**, so should be careful.

7.1.8 Converting a HIM Project to a HEW Project

By using the HimToHew tool supplied with the HEW system, you can convert HIM projects into HEW projects.

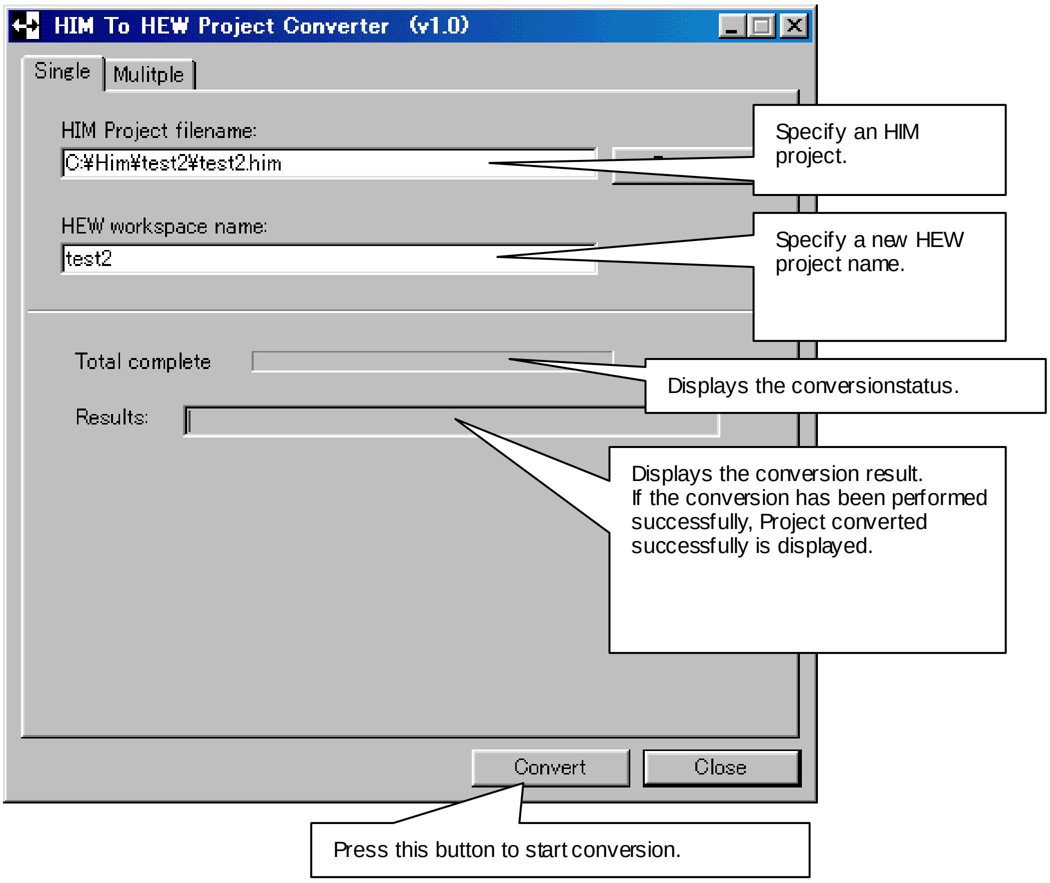
In the [Programs (P)] on the Windows® [Start Menu], select [Him To Hew Project Converter] from [Renesas High-performance Embedded Workshop].

You will find Single and Multiple tabs.

Select the Single tab when generating an HEW workspace and an HEW project from one HIM project.

Select the Multiple tab when converting multiple HIM projects into HEW projects and registering them in an HEW workspace in batch.

(1) Single tab



HIM Project filename: C:¥Him¥test2¥test2.him

HEW workspace name: test2

Total complete

Results:

Specify an HIM project.

Specify a new HEW project name.

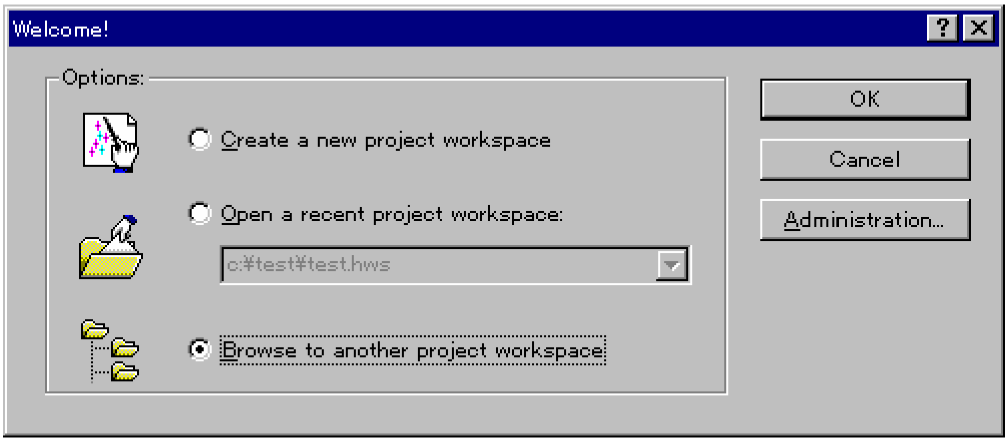
Displays the conversionstatus.

Displays the conversion result.
If the conversion has been performed successfully, Project converted successfully is displayed.

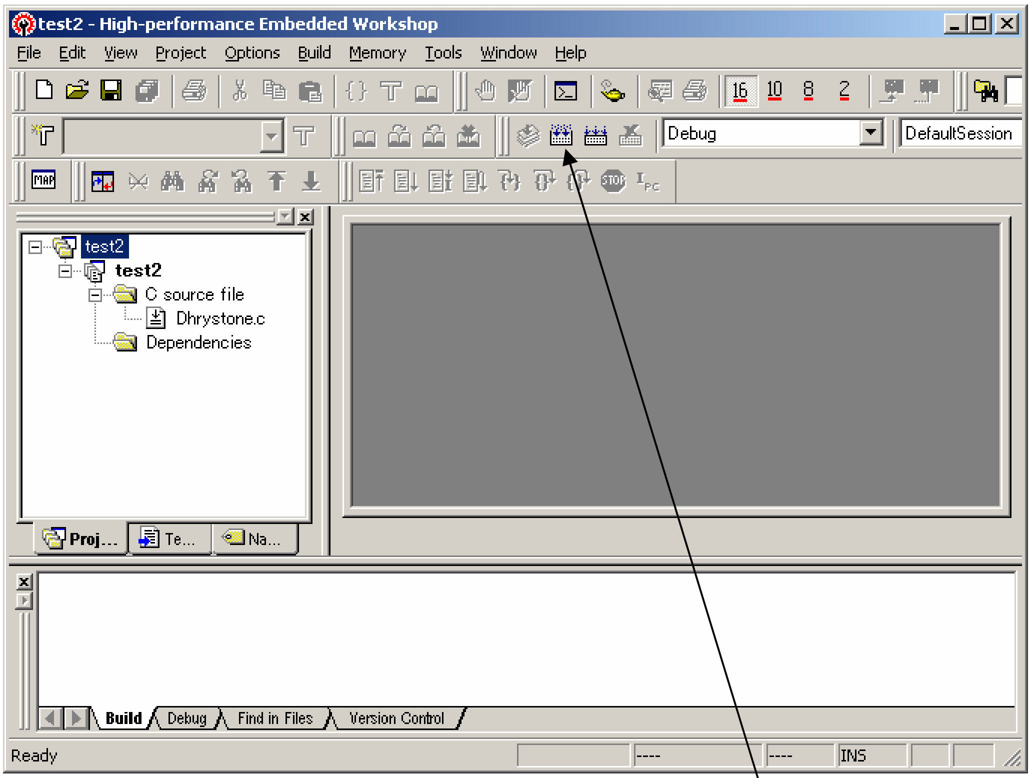
Press this button to start conversion.

In the next step, start the HEW.

Select **Browse to another project workspace**, click on the [OK] button, and specify the HEW project that has been converted.


Welcome!
Options:
○ Create a new project workspace
○ Open a recent project workspace:
c:¥test¥test.hws
● Browse to another project workspace
OK
Cancel
Administration...

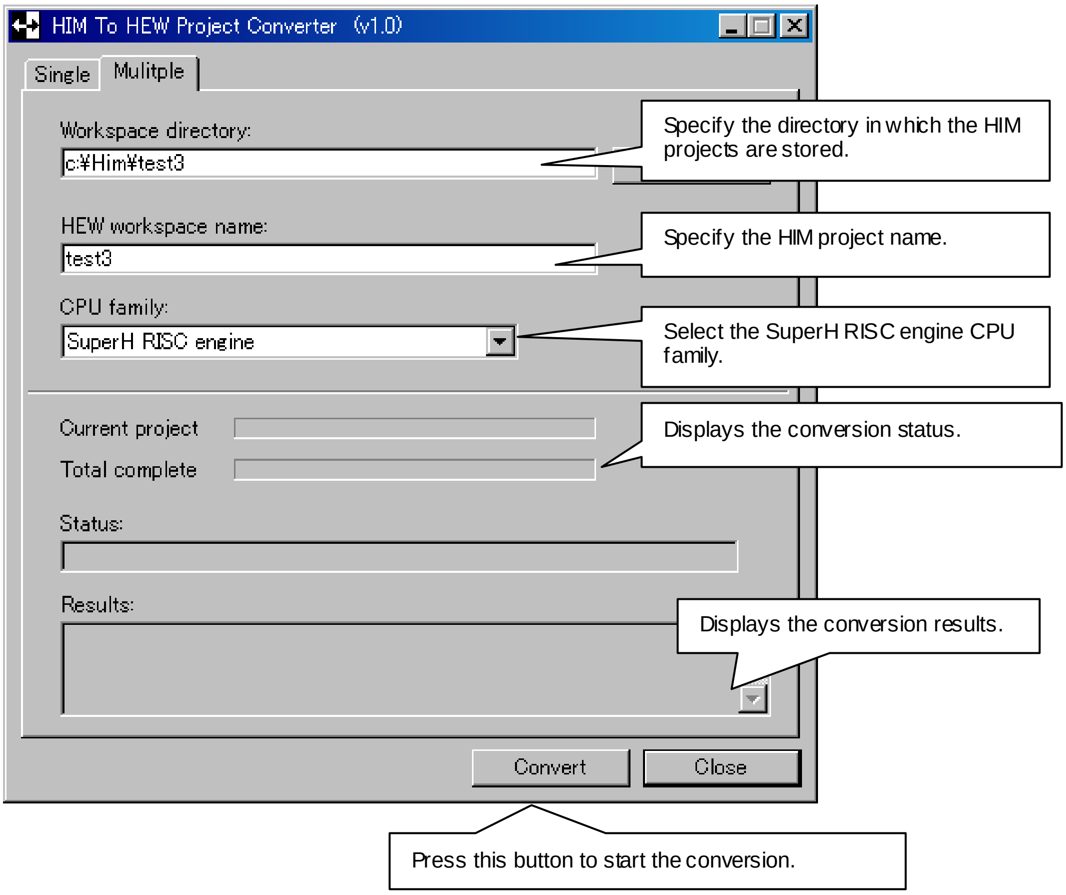
The HEW project is opened as shown below:



Specify [Build → Build] to execute the building process. On the command menu, click here.

(2) Multiple tab

This tab converts multiple HIM projects into HEW projects.



Workspace directory: c:¥Him¥test3

HEW workspace name: test3

CPU family: SuperH RISC engine

Specify the directory in which the HIM projects are stored.

Specify the HIM project name.

Select the SuperH RISC engine CPU family.

Displays the conversion status.

Displays the conversion results.

Press this button to start the conversion.

After the conversion, start the HEW as in the case of the Single tab in order to build the converted HEW workspace.

7.1.9 Add Supported CPUs

- Description:

HEW can automatically generate I/O register definition and vector table files, but HEW cannot support new CPUs which are released after HEW release.

In this case, the tool **DeviceUpdater** can make HEW support new CPUs.

And this tool can update generated files to bug fixed version.

- How to get **DeviceUpdater**

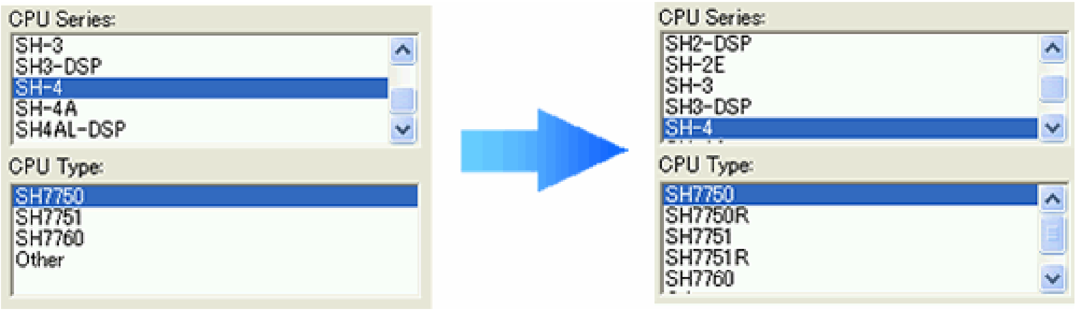
Download from the following URL of Renesas Technology Corp.

Please refer to Notes of this page, too.

http://www.renesas.com/eng/products/mpumcu/tool/crosstool/support_tool/device_updater.html

- Execution Results of **DeviceUpdater**

CPU types are added as follows.



CPU Series:
SH-3
SH3-DSP
SH-4
SH-4A
SH4AL-DSP

CPU Type:
SH7750
SH7751
SH7760
Other

CPU Series:
SH2-DSP
SH-2E
SH-3
SH3-DSP
SH-4

CPU Type:
SH7750
SH7750R
SH7751
SH7751R
SH7760

- Notes

This feature is supported by HEW 2.2 or later.

7.2 Simulations

7.2.1 Pseudo-interrupts

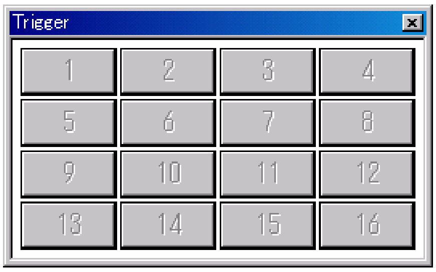
• Description:

Pseudo-interrupt buttons, which simulate certain interrupt causes, when clicked on, can cause pseudo-interrupts manually.

For each button, specify an interrupt priority and interrupt condition.

• Usage:

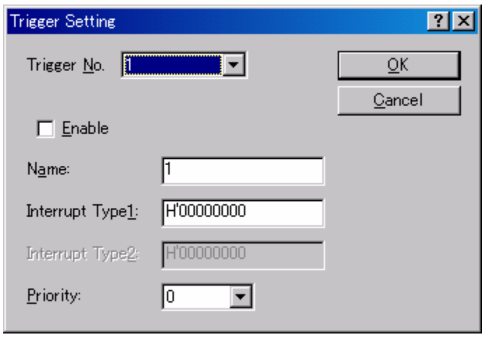
1. When you choose [View -> CPU -> Trigger], the following view appears:



2. Click the right mouse button on this view and choose [Setting…]. The [Trigger Setting] dialog box appears.
   If you check the [Enable] check box, the interrupt identified by trigger number 1 is enabled.

   In addition, specify an interrupt name, interrupt priority, and interrupt condition (vector number).

   The interrupt button identified by trigger number 1 becomes active.



Trigger No. 1

□ Enable

Name: 1

Interrupt Type1: H'00000000

Interrupt Type2: H'00000000

Priority: 0

3. The setting is now complete. When one of the buttons that was set during the above procedure is clicked on, the program will stop as specified by the pertinent vector table.

• Note:

This feature is supported by HEW 2.1 or later.

7.2.2 Convenient Breakpoint Functions

- Description:

The HEW breakpoint facility includes the following convenient functions, which will be activated not only upon ordinary breaks, but when a break condition is satisfied.

File input

File output

Interrupt

- How to display a breakpoint view:

HEW 2.2 or earlier: Choose [View -> Code -> Breakpoints]

HEW 3.0 or later: Choose [View -> Code -> Eventpoints]

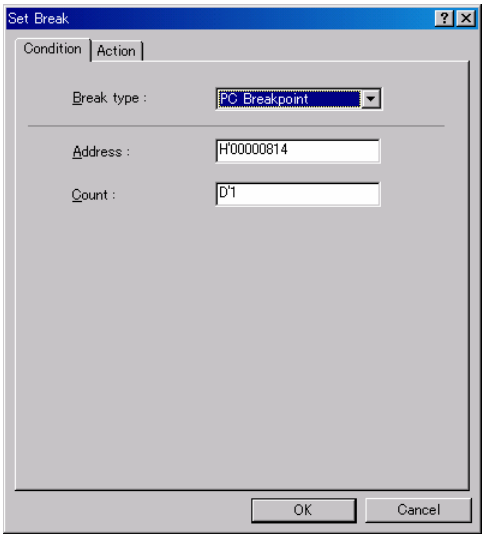
Note: For HEW 3.0 or later, go to the [Breakpoints] view and click on the [Software Event] tab.

- File input setting example:

Right-click on the [Breakpoints] view and choose [Setting…] to open the following [Set Break] dialog box. As shown below, PC breakpoint is used so that a break condition is considered as satisfied when the PC reaches the following address. The setting method is similar for other breakpoint types.

Click on the [Action] tab, select [File Input] in the [Action type] field, specify an input file name, an input address, and other items, and then click on the [OK] button.

([Condition]

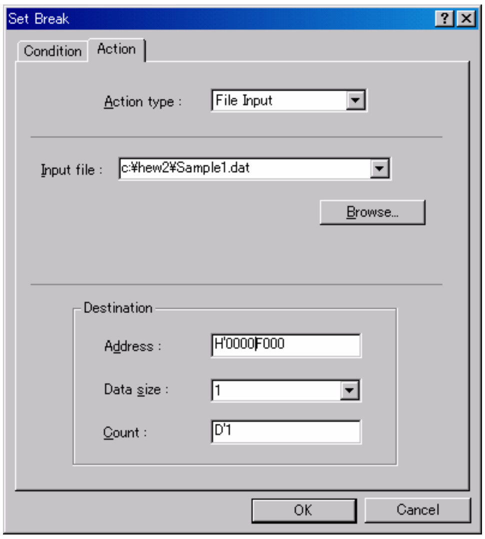


Break type : PC Breakpoint

Address : H'00000814

Count : D'1

([Action] tab)



Action type : File Input

Input file : c:¥hew2¥Sample1.dat

Destination

Address : H'0000F000

Data size : 1

Count : D'1

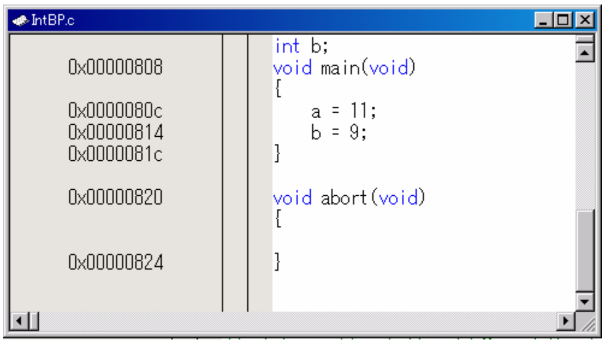
- File input action example:

Let’s see the following practical action example:

As the result of the above setting, the breakpoint is at [H'00000814] and the input file contains [H'FF].

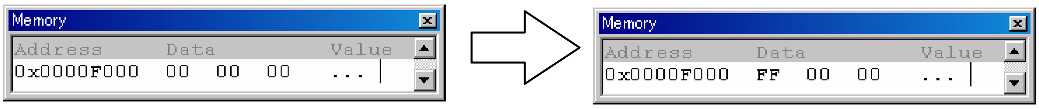
Run the program using the Go command or similar method.

(Source code fragment)



```
             int b;
0x00000808   void main(void)
             {
0x0000080c       a = 11;
0x00000814       b = 9;
0x0000081c   }

0x00000820   void abort(void)
             {

0x00000824   }
```

You can see that, when the PC reaches [H'00000814], the break condition is satisfied and, as a consequence, the memory contents of address H'F000 change.



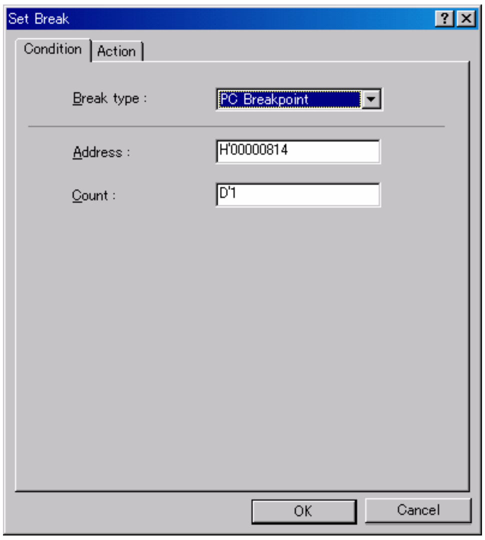
```
Memory
Address      Data          Value
0x0000F000   00  00  00    ...
```

```
Memory
Address      Data          Value
0x0000F000   FF  00  00    ...
```

- **File output** setting example:

The method for file output setting in the [Set Break] dialog box is similar to the method for file input setting. For file output breakpoints, PC breakpoint is also used so that a break condition is considered as satisfied when the PC reaches the following address. Click on the [Action] tab, select [File Output] in the [Action Type] field, specify an output file name, an output address, and other items, and then click on the [OK] button.

([Condition] tab)

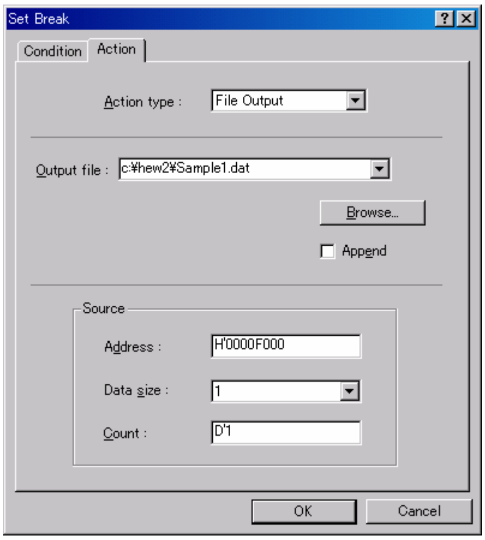


Break type : PC Breakpoint

Address : H'00000814

Count : D'1

([Action] tab)



Action type : File Output

Output file : c:¥hew2¥Sample1.dat

□ Append

Source

Address : H'0000F000

Data size : 1

Count : D'1

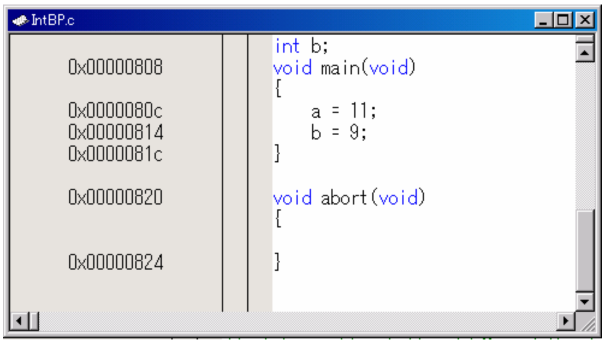
• **File output** action example:

Let's see the following practical action example:

As the result of the above setting, the breakpoint is at [H'00000814] and the contents of address H'F000 are [H'FF].

Run the program using the Go command or similar method.

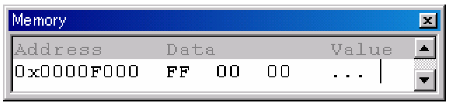
(Source code fragment)



IntBP.c

```
              int b;
0x00000808    void main(void)
              {
0x0000080c        a = 11;
0x00000814        b = 9;
0x0000081c    }

0x00000820    void abort(void)
              {

0x00000824    }
```

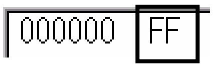
You can see that, when the PC reaches [H'00000814], the break condition is satisfied and, as a consequence, the contents of address H'F000 are output to the file.



Memory

```
Address     Data           Value
0x0000F000  FF   00   00   ...
```

(Sample.dat contents as seen on a binary editor)



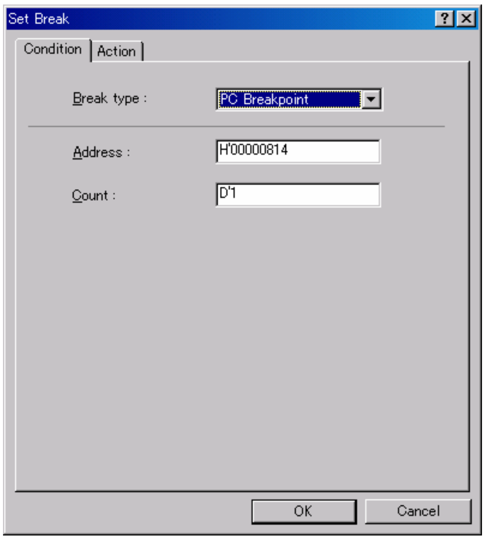
```
000000  FF
```

• **Interrupt** setting example:

The method for file output setting in the [Set Break] dialog box is similar to the method for file input setting. As shown below, PC breakpoint is used so that a break condition is considered as satisfied when the PC reaches the following address. The setting method is similar for other breakpoint types.

Click on the [Action] tab, select [Interrupt] in the [Action Type] field, specify an interrupt priority and an interrupt type (vector number 7), and click on the [OK] button.

([Condition] tab)



Set Break

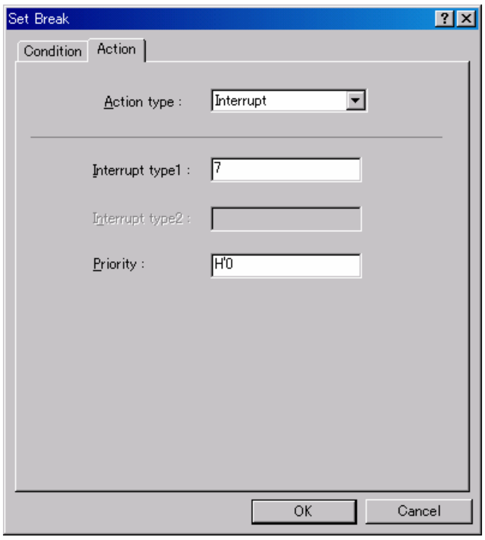
Condition | Action

Break type : PC Breakpoint

Address : H'00000814

Count : D'1

([Action] tab)



Set Break

Condition | Action

Action type : Interrupt

Interrupt type1 : 7

Interrupt type2 :

Priority : H'0

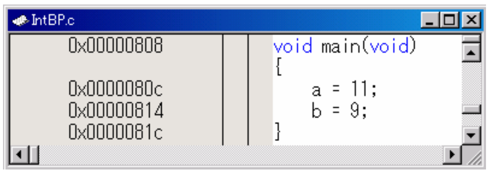
• **Interrupt** action example:

Let’s see the following practical action example:

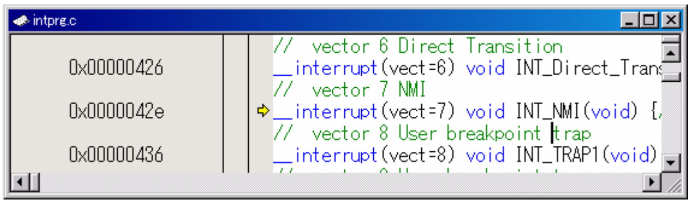
While the breakpoint is set at [H'00000814] as the result of the above setting, run the program by the Go command or similar method.

You can see that, when the PC reaches [H'00000814], a non-maskable interrupt (NMI) of vector number 7 will occur.

(Source code fragment)



```
0x00000808        void main(void)
                  {
0x0000080c            a = 11;
0x00000814            b = 9;
0x0000081c        }
```



```
                  //  vector 6 Direct Transition
0x00000426        __interrupt(vect=6) void INT_Direct_Trans
                  //  vector 7 NMI
0x0000042e        __interrupt(vect=7) void INT_NMI(void) {
                  //  vector 8 User breakpoint trap
0x00000436        __interrupt(vect=8) void INT_TRAP1(void)
```

7.2.3 Coverage Feature

- Description:

HEW allows users to collect statement coverage information within a user-specified address range during program execution. By using statement coverage information, you can observe how each statement is being executed. In addition, you can easily identify program code that has not been executed.

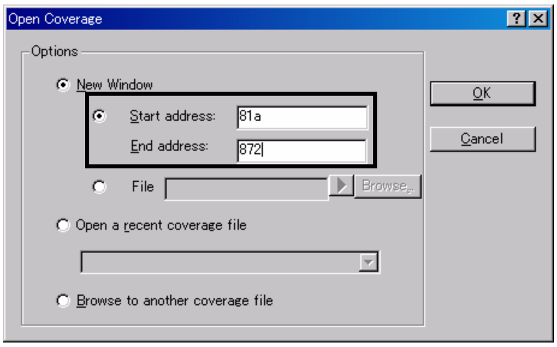
- How to open the [Open Coverage] dialog box:

[View -> Code -> Coverage...]

- How to collect new coverage information:

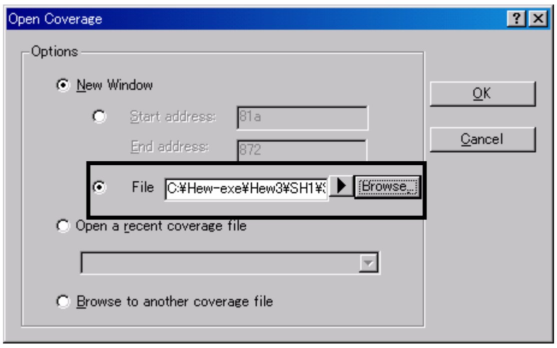
1. Open the [Open Coverage] dialog box, choose [New Window], and enter the start and end addresses that identify the range from which you want to obtain coverage information. If the HEW version is 3.0 or later, you can specify a C or C++ source file name to identify the information you want to collect.
   To complete the above specification, click on the [OK] button.

(Address specification)



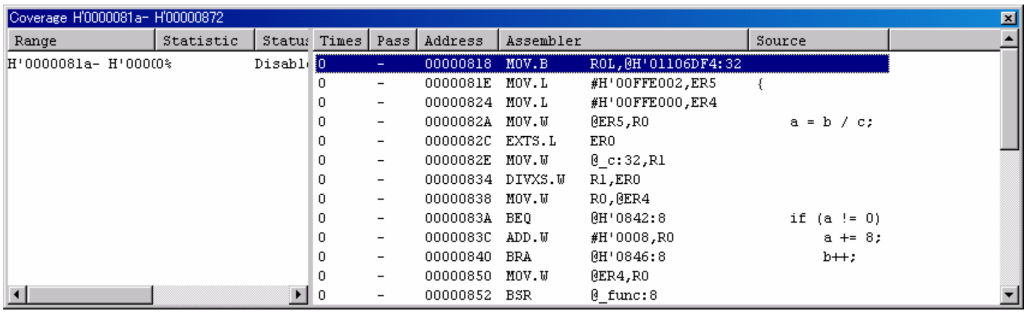
● New Window
● Start address: 81a
End address: 872
○ File

(File name specification) * Supported by HEW 3.0 or later



● New Window
○ Start address: 81a
End address: 872
● File C:¥Hew-exe¥Hew3¥SH1¥

2. When you click on the [OK] button, the following coverage view appears:
   On the right view, click the right mouse button and choose [Enable]. The coverage is enabled.

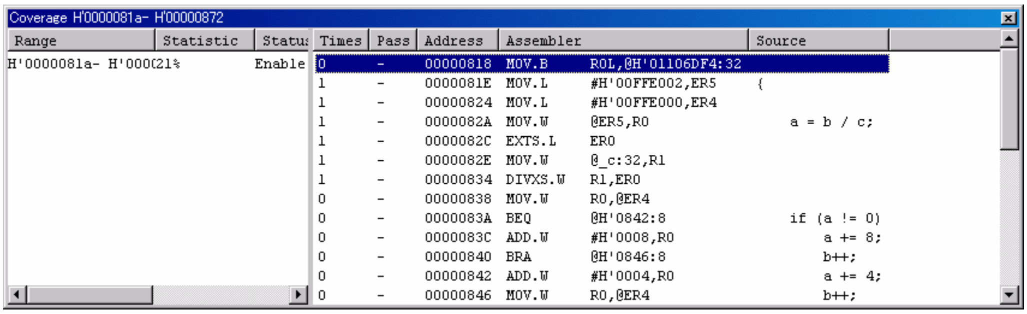


Coverage H'0000081a- H'00000872

Range	Statistic	Status
H'0000081a- H'000(0%		Disabl

Times	Pass	Address	Assembler		Source
0	-	00000818	MOV.B	ROL,@H'01106DF4:32	
0	-	0000081E	MOV.L	#H'00FFE002,ER5	{
0	-	00000824	MOV.L	#H'00FFE000,ER4	
0	-	0000082A	MOV.W	@ER5,R0	a = b / c;
0	-	0000082C	EXTS.L	ER0	
0	-	0000082E	MOV.W	@_c:32,R1	
0	-	00000834	DIVXS.W	R1,ER0	
0	-	00000838	MOV.W	R0,@ER4	
0	-	0000083A	BEQ	@H'0842:8	if (a != 0)
0	-	0000083C	ADD.W	#H'0008,R0	a += 8;
0	-	00000840	BRA	@H'0846:8	b++;
0	-	00000850	MOV.W	@ER4,R0	
0	-	00000852	BSR	@_func:8	

3. Let's run the program. Notice that the right coverage view contains a line with the [Times] column changed to 1. This indicates that the statement at the address corresponding to this line has been executed.
   On the left view, the C0 coverage value within the address range is displayed.



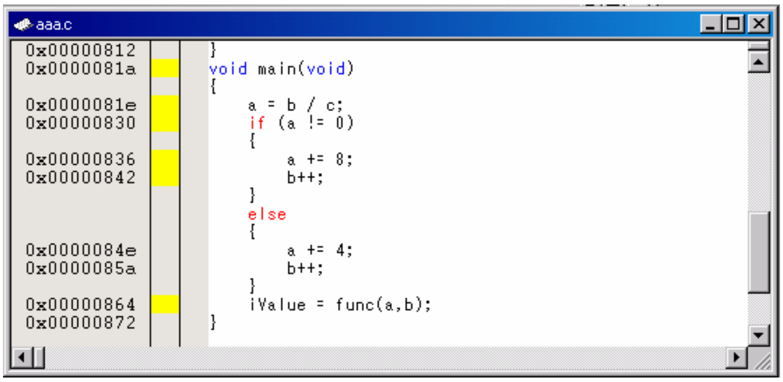
Coverage H'0000081a- H'00000872

Range	Statistic	Status
H'0000081a- H'000(21%		Enable

Times	Pass	Address	Assembler		Source
0	-	00000818	MOV.B	ROL,@H'01106DF4:32	
1	-	0000081E	MOV.L	#H'00FFE002,ER5	{
1	-	00000824	MOV.L	#H'00FFE000,ER4	
1	-	0000082A	MOV.W	@ER5,R0	a = b / c;
1	-	0000082C	EXTS.L	ER0	
1	-	0000082E	MOV.W	@_c:32,R1	
1	-	00000834	DIVXS.W	R1,ER0	
0	-	00000838	MOV.W	R0,@ER4	
0	-	0000083A	BEQ	@H'0842:8	if (a != 0)
0	-	0000083C	ADD.W	#H'0008,R0	a += 8;
0	-	00000840	BRA	@H'0846:8	b++;
0	-	00000842	ADD.W	#H'0004,R0	a += 4;
0	-	00000846	MOV.W	R0,@ER4	b++;

Note: The left coverage view exists when the HEW version is 3.0 or later.

4. In addition to the coverage view, you can use another method to see coverage information. A left column on the editor screen indicates whether program execution has passed a particular source line.

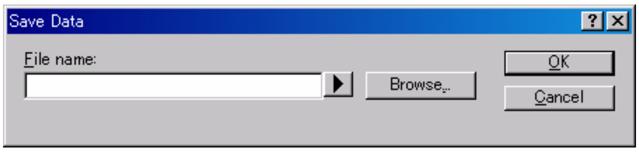


aaa.c

```
0x00000812       }
0x0000081a       void main(void)
                 {
0x0000081e           a = b / c;
0x00000830           if (a != 0)
                     {
0x00000836               a += 8;
0x00000842               b++;
                     }
                     else
                     {
0x0000084e               a += 4;
0x0000085a               b++;
                     }
0x00000864           iValue = func(a,b);
0x00000872       }
```

• Save Data:

To save coverage information, click the right mouse button on the right coverage view and enter a file name with the extension* .cov.

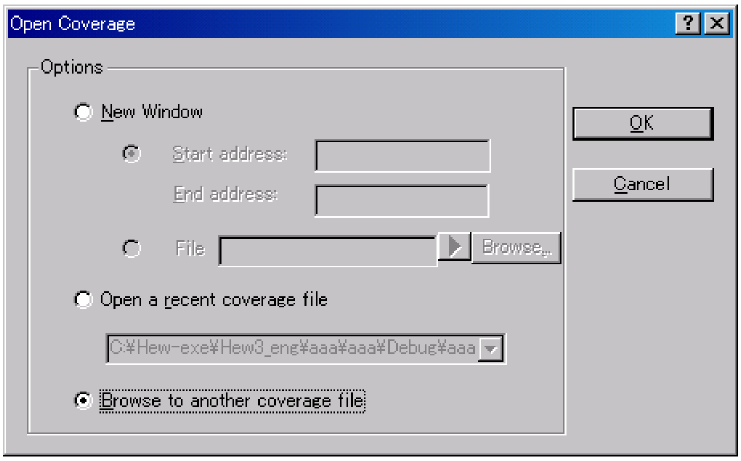


- Information collection using existing coverage information:

You can rarely obtain a single collection of coverage information that covers the entire program.

You may want to increase the coverage percentage while repeating coverage collection steps, each of which is performed under a different test condition.

For this purpose, specify a file that has been saved in the [Save Data] and select [Open a recent coverage file] or [Browse to another coverage file] in the [Open Coverage] dialog box. Then click on the [OK] button.



Open Coverage

Options

○ New Window

● Start address:

End address:

○ File

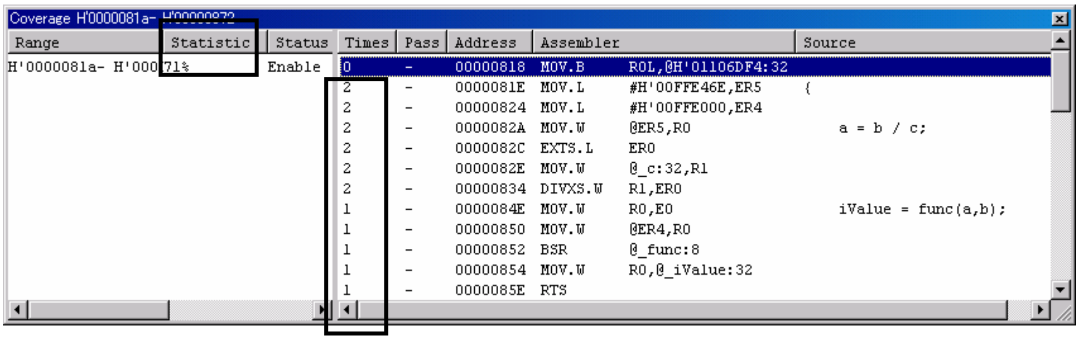
○ Open a recent coverage file

C:¥Hew-exe¥Hew3_eng¥aaa¥aaa¥Debug¥aaa

● Browse to another coverage file

The coverage view opens. Run the program again under a new condition.

As shown below, the coverage view and the editor display new information reflecting the current run, such as the number of runs and the new C0 coverage value.



```
Coverage H'0000081a- H'00000872

Range             Statistic  Status   Times  Pass  Address   Assembler                       Source
H'0000081a- H'000 71%        Enable   0      -     00000818  MOV.B    R0L,@H'01106DF4:32
                                      2      -     0000081E  MOV.L    #H'00FFE46E,ER5       {
                                      2      -     00000824  MOV.L    #H'00FFE000,ER4
                                      2      -     0000082A  MOV.W    @ER5,R0                   a = b / c;
                                      2      -     0000082C  EXTS.L   ER0
                                      2      -     0000082E  MOV.W    @_c:32,R1
                                      2      -     00000834  DIVXS.W  R1,ER0
                                      1      -     0000084E  MOV.W    R0,E0                     iValue = func(a,b);
                                      1      -     00000850  MOV.W    @ER4,R0
                                      1      -     00000852  BSR      @_func:8
                                      1      -     00000854  MOV.W    R0,@_iValue:32
                                      1      -     0000085E  RTS
```

7.2.4 File I/O

• Description:

HEW used to rely on the I/O simulation feature in order to simulate file I/O operations instead of actually performing file I/O.

However, HEW now allows actual files to be input or output if the following files are replaced.

• How to obtain files:

Download the files from the "Guideline for File Operatable Low-Level Interface Routines for Simulator and Debugger" page on the following URL of Renesas Technology Corp.

http://www.renesas.com/

• How to create the environment:

(1) Create a project by HEW.
Select [Application] or [Demonstration] as the project type.
A number of files are created automatically under the created project.
(If you have selected [Application] as the project type, check the [Use I/O Library] check box at project creation step 3.
The value specified in the [Number of I/O Stream] field must be at least the number of actually handled files + 3 (number of standard I/O files.

(2) Of the created files, replace "lowsrc.c" and "lowlvl.src".*[1]

(3) Create the "C:\Hew2\stdio" directory.*[2]

(4) Perform a rebuild to create a simulator/debugger environment in which file I/O is possible.

Notes: 1. -lowsrc.c-
These files are common to SH and H8.
Replace the file with the "lowsrc.c" file contained in the project.
-lowlvl.src-
This file varies from one CPU to another.
Replace this file with the "lowlvl.src" file contained in the folder corresponding to the CPU that has created the project.

2. In the created environment, standard I/O files will be actually opened when program code for file I/O processing is encountered, unlike the practice performed so far – simulation of file opening.
Therefore, files named "stdin", "stdout", and "stderr", which will be actually opened for standard I/O processing are automatically generated when the program is first executed.
Since these files are defined so that they should be created in "C:\Hew2\stdio", you must create the directory as explained in Item (3). If this directory does not exist, HEW will not work normally.
When the simulator runs, these files are opened by INIT_IOLIB() in the "lowsrc.c" file contained in the project.
stdin = 0
stdout = 1
stderr = 2

- Example of Use:

As in the following example, consider the use of printf or a similar method to output characters to the standard output (stdout):

(Sample program code)

```
void main(void)
{
          printf("***** ID-1 OK *****\n");
}
```

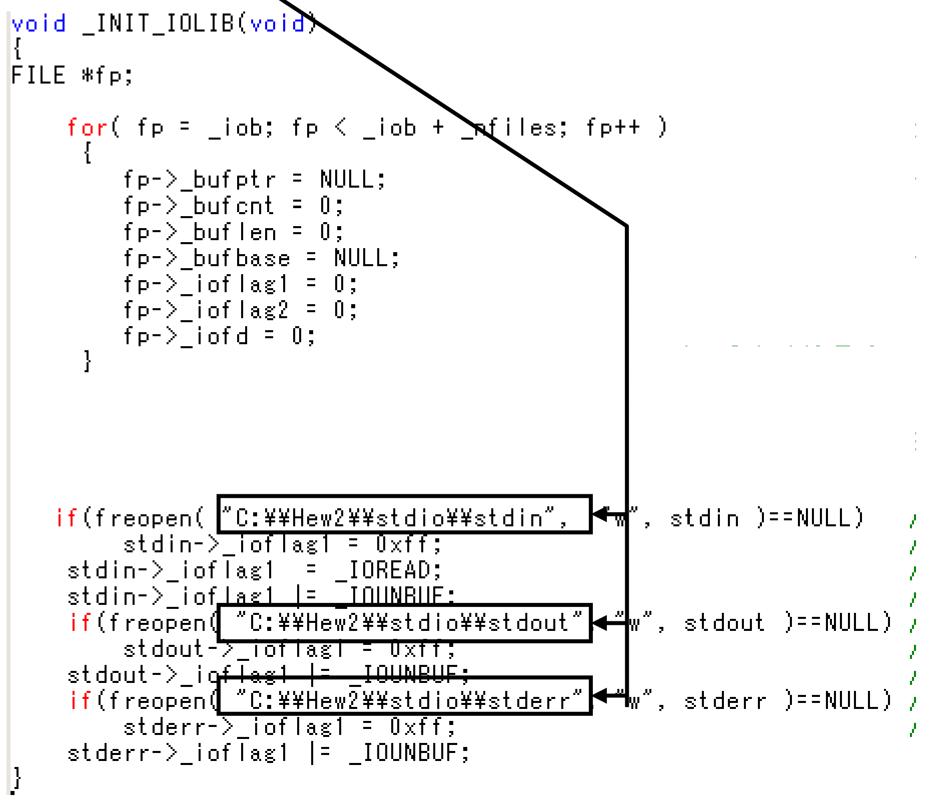
When you run this program, it creates a file named stdout in the "c:\Hew2\stdio" directory you have already created. The file contents are as follows:

(Contents of stdout)

```
***** ID-1 OK *****
```

- How to redirect I/O:

To redirect I/O, change this in the _INIT_IOLIB function in the lowsrc.c file.



```
void _INIT_IOLIB(void)
{
FILE *fp;

    for( fp = _iob; fp < _iob + _nfiles; fp++ )
     {
        fp->_bufptr = NULL;
        fp->_bufcnt = 0;
        fp->_buflen = 0;
        fp->_bufbase = NULL;
        fp->_ioflag1 = 0;
        fp->_ioflag2 = 0;
        fp->_iofd = 0;
     }




   if(freopen( "C:¥¥Hew2¥¥stdio¥¥stdin", "w", stdin )==NULL)    /
        stdin->_ioflag1 = 0xff;                                 /
    stdin->_ioflag1  = _IOREAD;                                 /
    stdin->_ioflag1 |= _IOUNBUF;                                /
    if(freopen( "C:¥¥Hew2¥¥stdio¥¥stdout" "w", stdout )==NULL) /
        stdout->_ioflag1 = 0xff;                                /
    stdout->_ioflag1 |= _IOUNBUF;                               /
    if(freopen( "C:¥¥Hew2¥¥stdio¥¥stderr" "w", stderr )==NULL) /
        stderr->_ioflag1 = 0xff;                                /
    stderr->_ioflag1 |= _IOUNBUF;
}
```

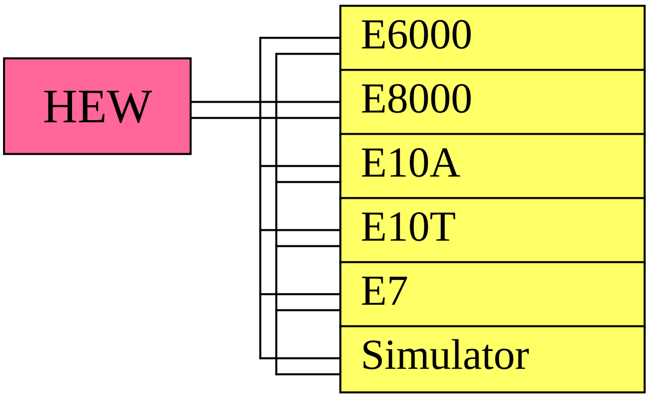
7.2.5 Debugger Target Synchronization

- Description:

HEW allows you to debug multiple targets on a single instance of HEW.

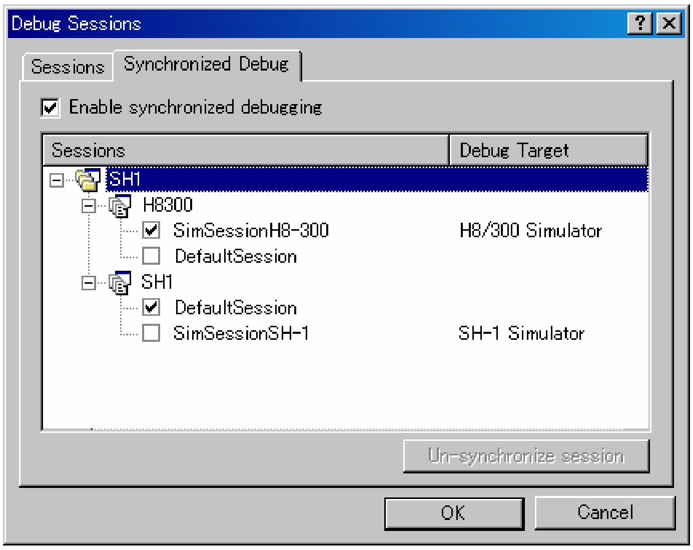
This means that you can debug multiple targets at the same time while synchronizing them with each other.

In addition, you can raise an event (such as a step or Go) in one session in synchronization with the same event in another session.



- How to synchronize debugger targets:

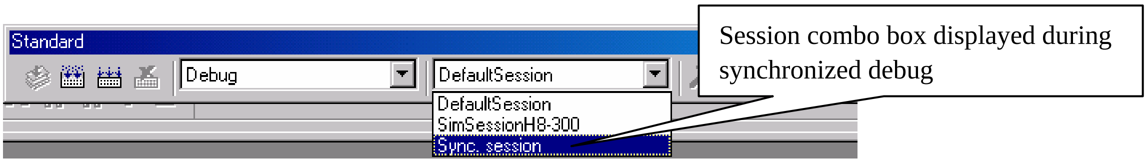
1. Choose [Options -> Debug sessions...] to open the following dialog box and click the [Synchronized Debug] tab. Check any session you want to synchronize and check the [Enable synchronized debugging] check box.



☑ Enable synchronized debugging

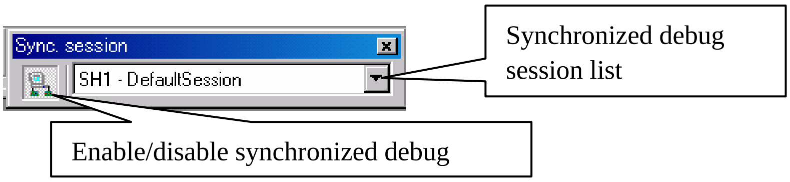
Sessions	Debug Target
SH1	
H8300	
☑ SimSessionH8-300	H8/300 Simulator
□ DefaultSession	
SH1	
☑ DefaultSession	
□ SimSessionSH-1	SH-1 Simulator

2. Select [Sync. session] from the session combo box on the [Standard] tool bar.



Session combo box displayed during synchronized debug

3. The [Sync. session] tool bar appears in the tool bar. The setting is now complete.



Synchronized debug session list

Enable/disable synchronized debug

- Available commands:

When synchronized debug is enabled, you can perform the following actions in synchronized mode:

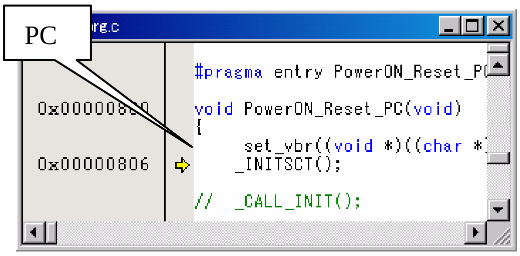
User action	Target debugger session 1	Target debugger session 2
[Run] during one of the sessions	"Run"	"Run"
[Step] during one of the sessions	"Step"	"Step"
ESC pressed during one of the sessions	"Stop"	"Stop"
-	"Stop" due to a breakpoint or user program error	Stop (same as when ESC is pressed)
-	Stop (same as when ESC is pressed)	"Stop" due to a breakpoint or user program error
[CPU reset] during one of the sessions	"CPU reset"	"CPU reset"

- Synchronized debug example

An example of executing the step command is provided below.

1. Execute the step during [SH1 – SimSessionSH-1].The following condition results:

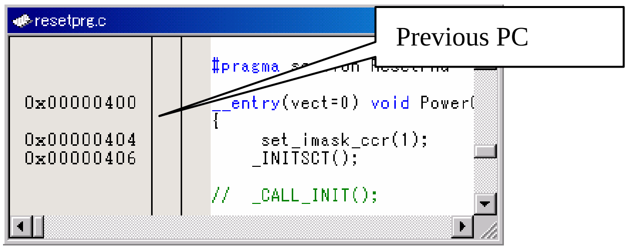
SH – SimSessionSH-1 state



PC

```
            #pragma entry PowerON_Reset_PC
0x00000800  void PowerON_Reset_PC(void)
            {
                set_vbr((void *)((char *
0x00000806      _INITSCT();

            //  _CALL_INIT();
```

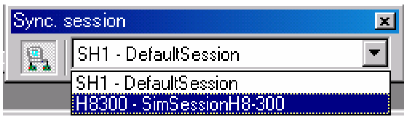
H8300 - SimSessionH8-300 state



Previous PC

```
            #pragma section ResetPRG
0x00000400  __entry(vect=0) void Power(
            {
0x00000404      set_imask_ccr(1);
0x00000406      _INITSCT();

            //  _CALL_INIT();
```

2. Change the session using the [Sync. session] tool bar.

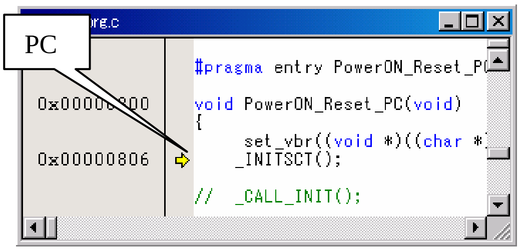


SH1 - DefaultSession

SH1 - DefaultSession
H8300 - SimSessionH8-300

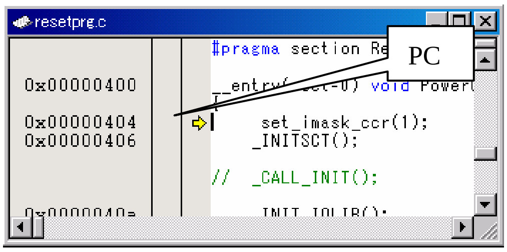
3. As shown below, you can see that the PC has also moved to the next line during the [H8300 – SimSessionH8-300] session.

SH – SimSessionSH-1 state



```
             #pragma entry PowerON_Reset_P
0x0000 00    void PowerON_Reset_PC(void)
             {
                 set_vbr((void *)((char *
0x00000806       _INITSCT();

             //  _CALL_INIT();
```

H8300 - SimSessionH8-300 state



```
             #pragma section Re
0x00000400   __entry(     ) void Power
0x00000404       set_imask_ccr(1);
0x00000406       _INITSCT();

             //  _CALL_INIT();
0x0000040a       INIT_IOLIB();
```

• Note:

This feature is supported by HEW 3.0 or later.

7.2.6 How to Use Timers

- Description:

HEW supports prioritization of timers and interrupts.

For each timer, only channel 0 is supported.

HEW support is limited to overflow, underflow, and compare match interrupts. HEW does not support interrupts that involve terminal I/O, such as input capture interrupts.

- Supported timer control registers in each CPU

In the Supported column on the following table, ○ indicates that the register is supported and Δ indicates that only the bits associated with the feature described in the paragraph under [Description] are supported.

Debug platform name	Timer name	Supported control register	Supported
SH-1	ITU0	TSTR	Δ
		TCR	Δ
		TIER	○
		TSR	○
		TCNT	○
		GRA	○
		GRB	○
SH-2/SH-2E/ SH2-DSP(SH7065)	CMT0	CMSTR	○
		CMCSR	○
		CMCNT	○
		CMCOR	Δ
SH-3/SH3-DSP/ SH3-DSP(Core)/ SH-4/SH-4BSC/ SH-4(SH7750R)	TMU0	TCR	Δ
		TCNT	○
		TSTR	○
		TCOR	○
SH2-DSP(Core)	FRT0	TIER	Δ
		FTCSR	Δ
		FRC	○
		OCRA	○
		OCRB	○
		TCR	Δ
		TOCR	Δ

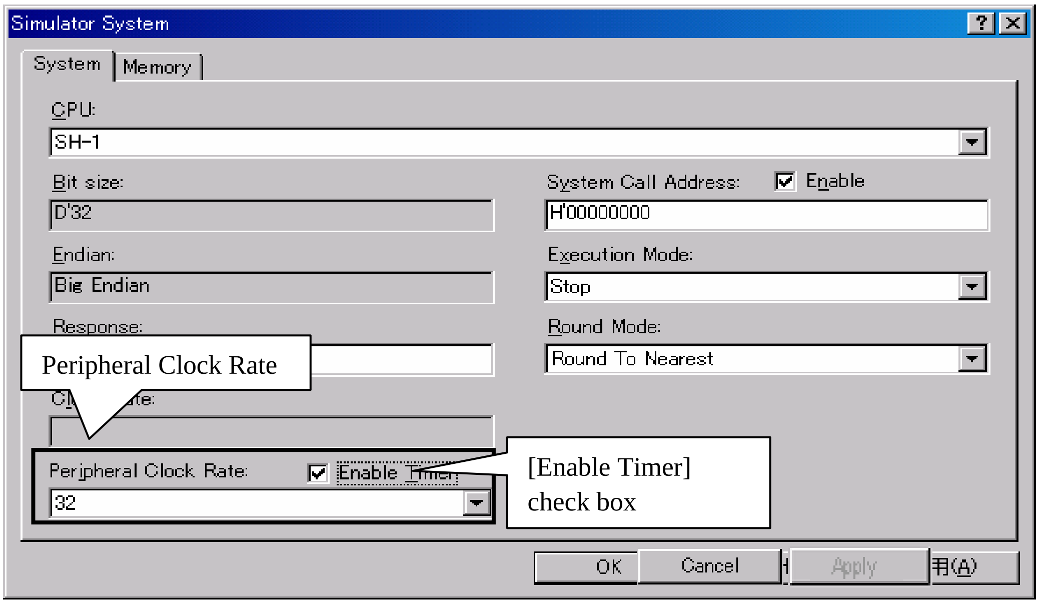
- Supported interrupt priority level setting registers in each CPU

In the Supported column on the following table, ○ indicates that the register is supported and Δ indicates that only the bits associated with the feature described in the paragraph under [Description] are supported.

Debug platform name	Supported control register	Supported
SH-1	IPRC	Δ
SH-2	IPRG	Δ
SH-2E	IPRJ	Δ
SH2-DSP(SH7065)	IPRL	Δ
SH-3/SH3-DSP/ SH3-DSP(Core)/ SH-4/SH-4BSC/ SH-4(SH7750R)	IPRA	Δ
SH2-DSP(Core)	INTPRI0B	Δ

- Timer simulation method:

Choose [Options -> Simulator -> System...] to open the following [Simulator System] dialog box, check the [Enable Timer] check box, and specify a ratio between the external clock and the peripheral module clock.



CPU: SH-1

Bit size: D'32

System Call Address: ☑ Enable

H'00000000

Endian: Big Endian

Execution Mode: Stop

Response:

Round Mode: Round To Nearest

Peripheral Clock Rate: ☑ Enable Timer

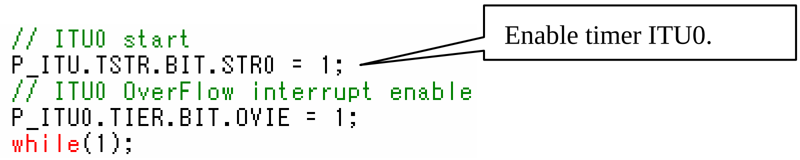
32

Peripheral Clock Rate

[Enable Timer] check box

In addition, you can use timer control registers and write program code to enable them as shown below.

If you create a clock that drives timers via a peripheral module, specify the frequency division ratio using an appropriate timer control register.



```
// ITU0 start
P_ITU.TSTR.BIT.STR0 = 1;
// ITU0 OverFlow interrupt enable
P_ITU0.TIER.BIT.OVIE = 1;
while(1);
```

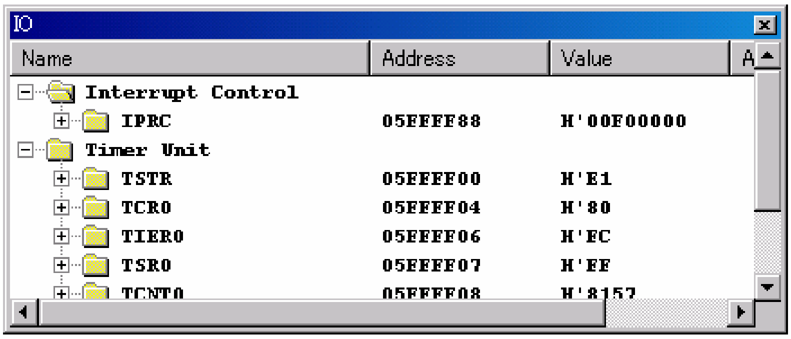
Enable timer ITU0.

Note: Before setting the value to the timer control registers, confirm that the access to the timer register is permitted in the "memory" tab of the "Simulator System" dialog box.

If the access is not permitted, you can neither set the value to the control register nor use the timer.

• How to view timer register settings:

To view settings on timer registers and interrupt priority level setting registers, choose [View -> CPU -> I/O] to open the following I/O window.



Name	Address	Value	A
Interrupt Control			
IPRC	05FFFF88	H'00F00000	
Timer Unit			
TSTR	05FFFF00	H'E1	
TCR0	05FFFF04	H'80	
TIER0	05FFFF06	H'FC	
TSR0	05FFFF07	H'FF	
TCNT0	05FFFF08	H'8157	

• Note:

This feature is supported by HEW 3.0 or later.

7.2.7 Examples of Timer Usage

- Description:

This subsection outlines how to use compare match and cyclic handler interrupts, using ITU in the SH7034 (SH-1) as an example.

- HEW setup:

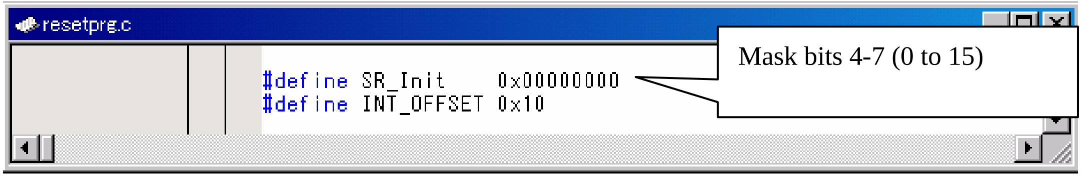
Enable the timers by referring to the paragraph entitled “Timer simulation method” in subsection 7.2.6, How to Use Timers.

- Sample program containing code that raises a compare match interrupt:

The following sample program contains code that raises a compare match interrupt.

Before a compare match interrupt can occur, the interrupt priority level specified by IPRC (interrupt priority register) must be equal to or higher than the value specified by the interrupt mask bits in the SR (status register).

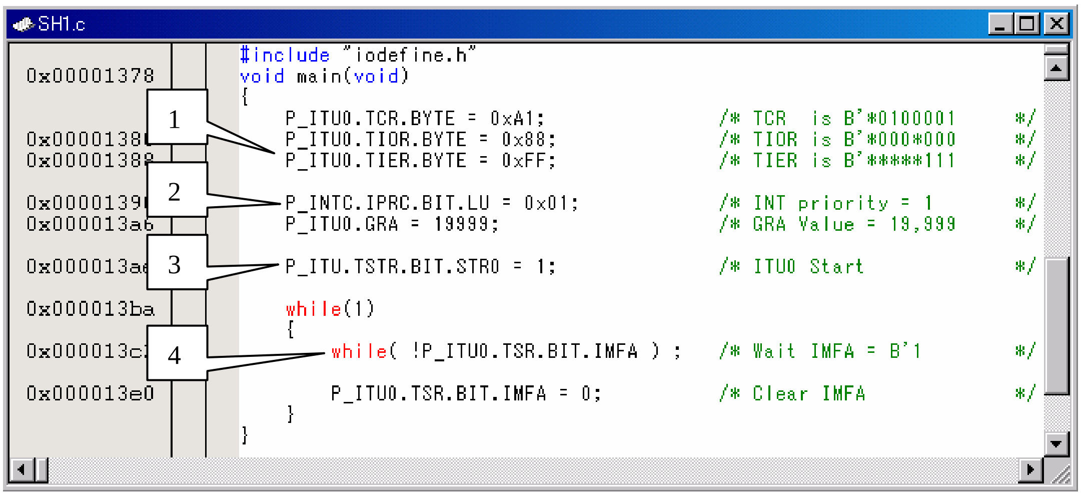
[Setting SR interrupt mask bits]
Set bits 4-7 in the SR to one of the values from 0 to 15 using a file that contains the following reset routine.



```
#define SR_Init     0x00000000
#define INT_OFFSET  0x10
```

Mask bits 4-7 (0 to 15)

[Explanation of an interrupt generation program]

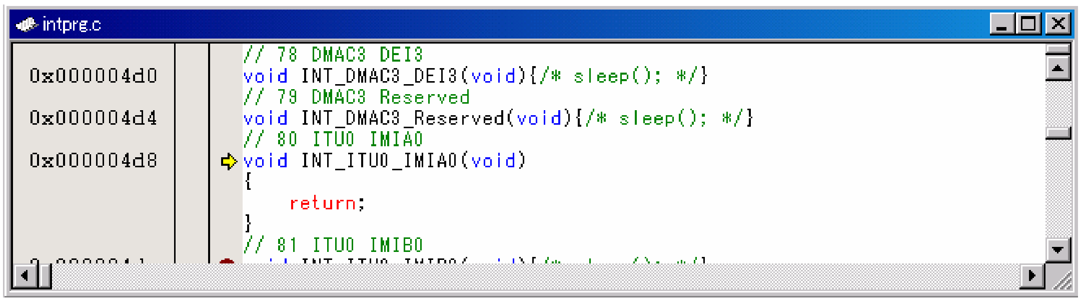


```
0x00001378   #include "iodefine.h"
             void main(void)
             {
                 P_ITU0.TCR.BYTE = 0xA1;              /* TCR  is B'*0100001     */
                 P_ITU0.TIOR.BYTE = 0x88;             /* TIOR is B'*000*000     */
                 P_ITU0.TIER.BYTE = 0xFF;             /* TIER is B'*****111     */

                 P_INTC.IPRC.BIT.LU = 0x01;           /* INT priority = 1       */
                 P_ITU0.GRA = 19999;                  /* GRA Value = 19,999     */

                 P_ITU.TSTR.BIT.STR0 = 1;             /* ITU0 Start             */

0x000013ba       while(1)
                 {
                     while( !P_ITU0.TSR.BIT.IMFA ) ;  /* Wait IMFA = B'1        */

0x000013e0           P_ITU0.TSR.BIT.IMFA = 0;         /* Clear IMFA             */
                 }
             }
```

1. When the IMFA (compare match flag A) bit in TIER (Timer Interrupt Enable register) becomes 1, the interrupt is enabled.
2. Set an interrupt priority in IMFA.
3. Start the ITU0 timer.
4. Wait until the IMFA bit becomes 1. (Wait for a compare match.)

- Program execution:

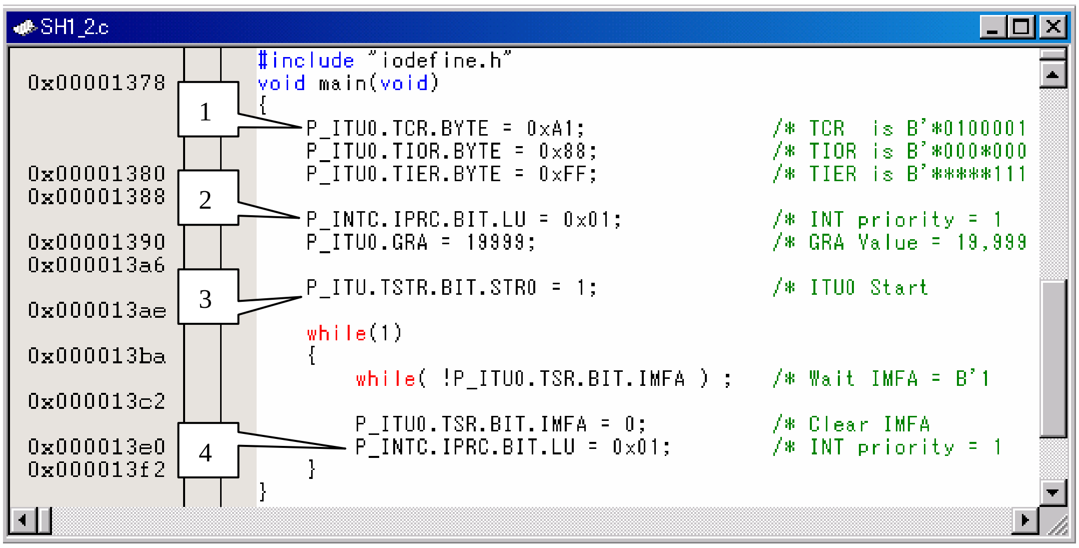
Wait until TCNT0 (timer counter 0) and GRA (general register A) match (a compare match occurs) at step 4 in the paragraph entitled “Explanation of an interrupt generation program.”
When the two match, a compare match interrupt occurs, with the result of calling the following interrupt routine:
For further information, refer to the pertinent hardware manual.



```
             // 78 DMAC3 DEI3
0x000004d0   void INT_DMAC3_DEI3(void){/* sleep(); */}
             // 79 DMAC3 Reserved
0x000004d4   void INT_DMAC3_Reserved(void){/* sleep(); */}
             // 80 ITU0 IMIA0
0x000004d8 ⇨ void INT_ITU0_IMIA0(void)
             {
                 return;
             }
             // 81 ITU0 IMIB0
```

- Sample program containing code for a cyclic handler

The following sample program contains code for a cyclic handler.
When a compare match occurs, the program clears the timer, and then branches control to an interrupt handler.
After the interrupt is serviced, the program lowers the interrupt priority in IPRC (interrupt priority register).
Control then returns to the code that caused the interrupt. The program raises the interrupt priority to ensure that the IMFA bit can be set.
For information on SR interrupt mask bit setting, refer to the compare match sample program.



```
             #include "iodefine.h"
0x00001378   void main(void)
             {
                 P_ITU0.TCR.BYTE = 0xA1;                /* TCR  is B'*0100001
0x00001380       P_ITU0.TIOR.BYTE = 0x88;               /* TIOR is B'*000*000
0x00001388       P_ITU0.TIER.BYTE = 0xFF;               /* TIER is B'*****111

                 P_INTC.IPRC.BIT.LU = 0x01;             /* INT priority = 1
0x00001390       P_ITU0.GRA = 19999;                    /* GRA Value = 19,999
0x000013a6
                 P_ITU.TSTR.BIT.STR0 = 1;               /* ITU0 Start
0x000013ae
                 while(1)
0x000013ba       {
                     while( !P_ITU0.TSR.BIT.IMFA ) ;    /* Wait IMFA = B'1
0x000013c2
                     P_ITU0.TSR.BIT.IMFA = 0;           /* Clear IMFA
0x000013e0           P_INTC.IPRC.BIT.LU = 0x01;         /* INT priority = 1
0x000013f2       }
             }
```

1. Set TCR (Timer Control register) to ensure that the timer counter (TCNT) will be cleared when the IMFA (compare match flag A) bit becomes 1.
2. Set an interrupt priority in IMFA.
3. Start the ITU0 timer.
4. After a compare match occurs, the interrupt priority level is raised.

- Program execution:

The program waits until a compare match occurs. When a compare match occurs, the program passes control to the following interrupt routine.

The interrupt routine services the interrupt, lowers the interrupt priority level in IMFA, and returns control to the program.

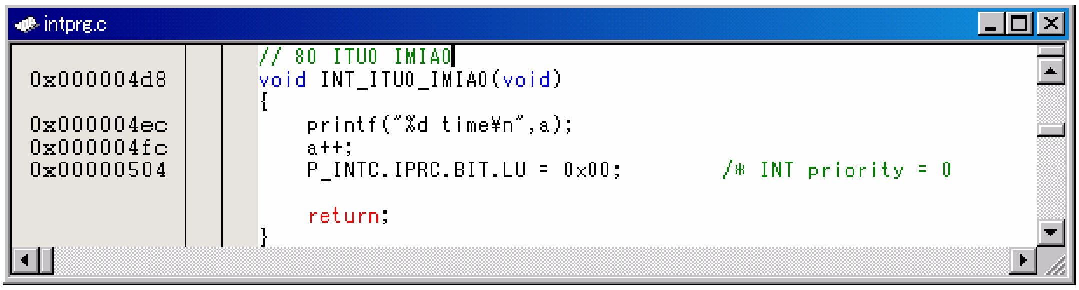
Interrupt processing can be completed in this way.

The program can then be ready to accept the next compare match interrupt.

For further information, refer to the pertinent hardware manual.

In accordance with the HEW specification, when an interrupt occurs, the PC stops at the beginning of the function that has caused the interrupt.

When simulating a cyclic handler, you need to advance the PC at each cycle by using the Go command or similar method.



```
                      // 80 ITU0 IMIA0
 0x000004d8           void INT_ITU0_IMIA0(void)
                      {
 0x000004ec               printf("%d time¥n",a);
 0x000004fc               a++;
 0x00000504               P_INTC.IPRC.BIT.LU = 0x00;        /* INT priority = 0

                          return;
                      }
```

7.2.8 Reconfiguration of Debugger Target

- Description:

HEW can configure Debugger Target, if you select Application for the project type when creating a new workspace.

However, when creating a new project, you may sometimes not configure this, because you then believe that this is unnecessary.

If you do, you can use this feature to reconfigure Debugger Target after creating a project.

However, this feature is only available when you select Application for the project type when creating a new workspace.

- Usage:

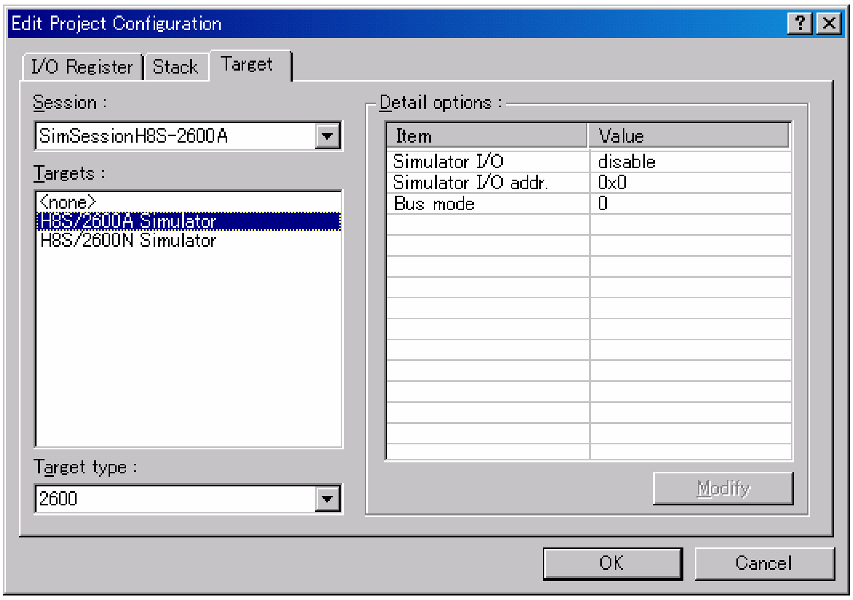
HEW Menu: **Project > Edit Project Configuration...**

- Functions that can be reconfigured:

[Setting method]

You can set a simulator and other debugger targets on the [Target] tab in the [Edit Project Configuration] dialog box.

If a debugger is already connected to the session, you will see a message saying, “This target has already existed. It does not support duplicated targets” and cannot connect to the debugger target.



Session : SimSessionH8S-2600A

Targets :
<none>
H8S/2600A Simulator
H8S/2600N Simulator

Target type : 2600

Detail options :

Item	Value
Simulator I/O	disable
Simulator I/O addr.	0x0
Bus mode	0

- Note:

Reconfiguring a file is supported by HEW 2.1 or later.

7.3 Call Walker

• Description:

Call Walker reads the stack information files (*.sni) that are output by the optimizing linkage editor or the profile information files (*.pro) that are output by the simulator debugger. Call Walker also displays the sizes of the stacks that are used statically.

Although the sizes of the stacks used by assembly language programs cannot be output to stack information files, you can add the information by using the editing feature and obtain the sizes of the stacks used in the entire system.

Once you edit information about the sizes of the used stacks, you can save the modified information in a call information file (*.cal) or read the modified information from the file.

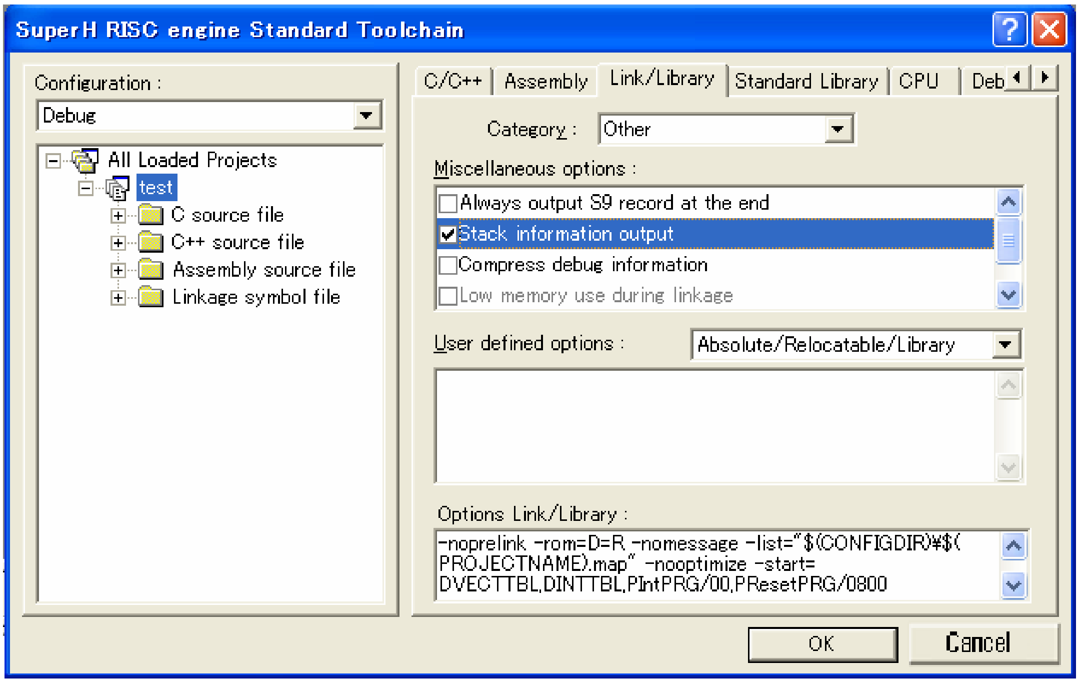
You can also merge multiple call information files.

7.3.1 Creating a Stack Information File

Follow the procedure below to create a stack information file or a profile information file.

- How to create a stack information file (*.sni)

To create a stack information file, select the following option in the [Link/Library] page.



SuperH RISC engine Standard Toolchain

Configuration : Debug

All Loaded Projects
test
C source file
C++ source file
Assembly source file
Linkage symbol file

C/C++ | Assembly | Link/Library | Standard Library | CPU | Deb

Category : Other

Miscellaneous options :

□ Always output S9 record at the end
☑ Stack information output
□ Compress debug information
□ Low memory use during linkage

User defined options : Absolute/Relocatable/Library

Options Link/Library :

```
-noprelink -rom=D=R -nomessage -list="$(CONFIGDIR)¥$(
PROJECTNAME).map" -nooptimize -start=
DVECTTBL,DINTTBL,PIntPRG/00,PResetPRG/0800
```

In this dialog box: Choose the [Link/Library] tab. Then select [Other] in the [Category] text box and select [Stack information output] in the [Miscellaneous options] list.

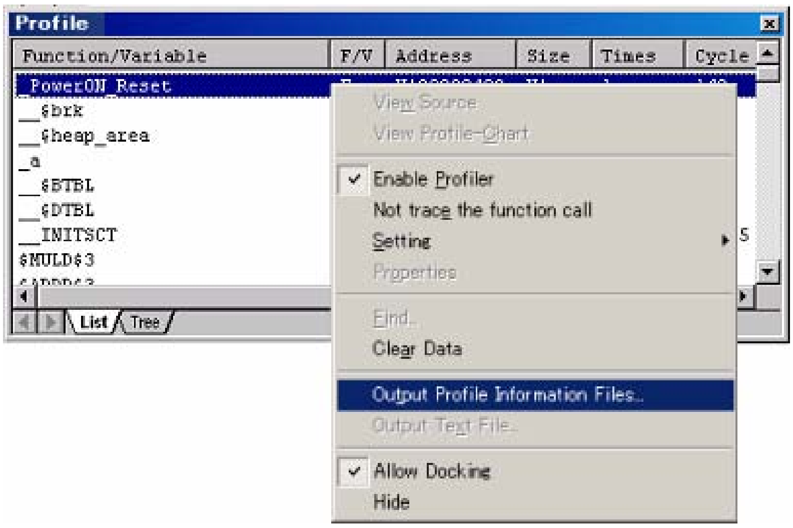
Command line: STACk

• How to create a profile information file (*.pro)

Use the profile feature to execute a desired user program.
When you complete executing the user program, right-click on the Profile window to save the profile information and create a profile information file (*.pro).

For details about how to create profile information, see section 4.14, Viewing the Profile Information, in the Simulator/Debugger Part in the High-performance Embedded Workshop 3 User's Manual.

[Profile window]
[View]->[Performance]->[Profile]



7.3.2 Starting Call Walker

You can start Call Walker in two ways.

- From the [Start] menu

Choose [Programs]->[Renesas High-performance Embedded Workshop]->[Call Walker].

- From HEW

Choose [Tools]->[Call Walker].

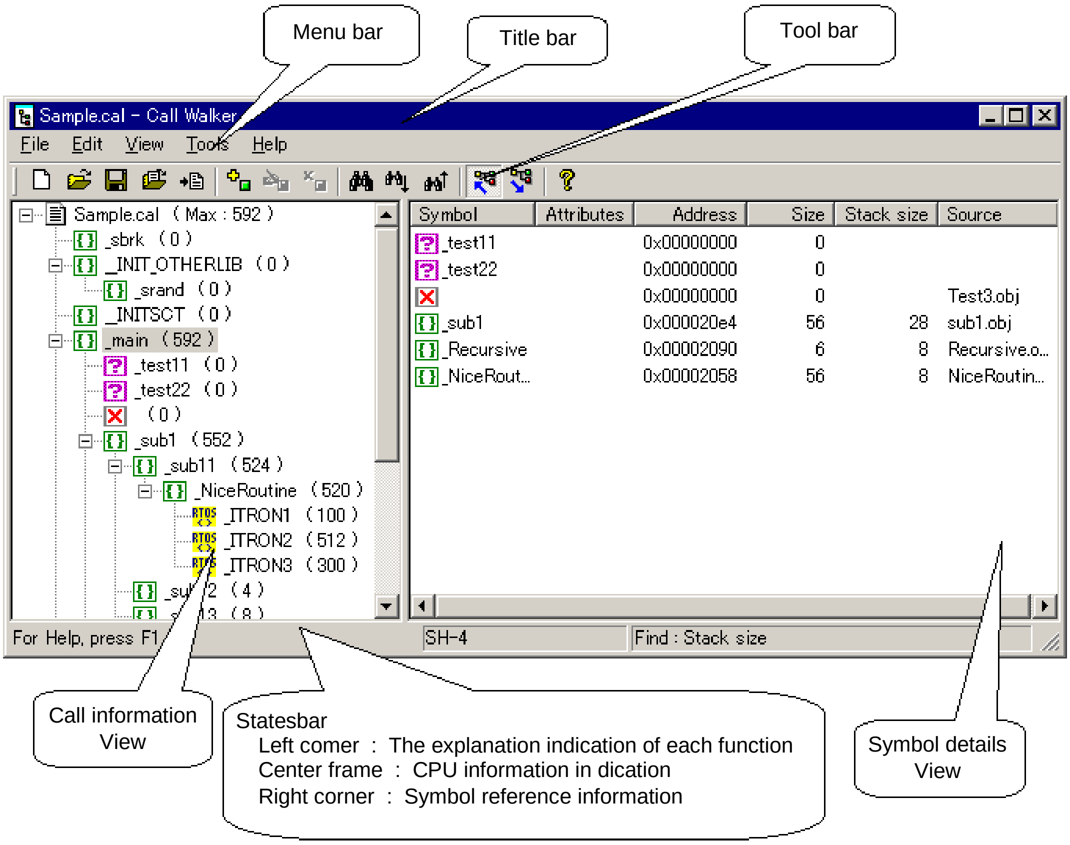
7.3.3 Call Walker Window and Opening a File

When you start Call Walker, you open a desired stack information file (*.sni) or profile information file (*.pro) by choosing **[File]->[Import Stack File...]**.

You choose **[File]->[Open...]** to open an existing edited file (*.cal).

When you open a file, the following window appears.

Note: For assembler functions other than those in the standard library, the stack size is shown as 0. See section 7.3.4, Editing Stack Information and set the appropriate stack size.



```
Sample.cal （Max : 592）
  _sbrk （0）
  _INIT_OTHERLIB （0）
    _srand （0）
  _INITSCT （0）
  _main （592）
    _test11 （0）
    _test22 （0）
     （0）
    _sub1 （552）
      _sub11 （524）
        _NiceRoutine （520）
          _ITRON1 （100）
          _ITRON2 （512）
          _ITRON3 （300）
      _su 2 （4）
          13 （8）
```

Symbol	Attributes	Address	Size	Stack size	Source
_test11		0x00000000	0		
_test22		0x00000000	0		
		0x00000000	0		Test3.obj
_sub1		0x000020e4	56	28	sub1.obj
_Recursive		0x00002090	6	8	Recursive.o...
_NiceRout...		0x00002058	56	8	NiceRoutin...

For Help, press F1 | SH-4 | Find : Stack size

Menu bar

Title bar

Tool bar

Call information View

Statesbar
Left comer : The explanation indication of each function
Center frame : CPU information in dication
Right corner : Symbol reference information

Symbol details View

- Call information view

This view shows the link hierarchy of the symbols.
The number on the right of each symbol name indicates the required stack size.

(1) Details about the symbols

The icon on the left of each symbol name indicates the type of the symbol.
The following types are available:

File being edited

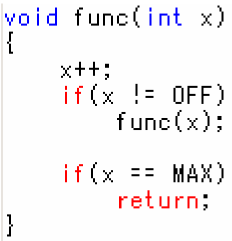
Assembler

C/C++ function

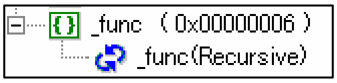
Direct or indirect recursive functions

(a) Direct recursive function
This icon indicates that the indicated function directly calls itself.

[Example]



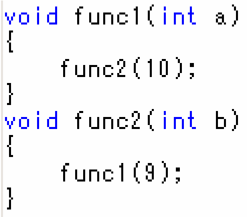
```
void func(int x)
{
    x++;
    if(x != OFF)
        func(x);

    if(x == MAX)
        return;
}
```



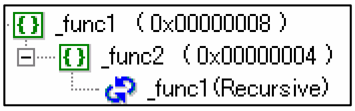
```
⊟ {} _func  (0x00000006 )
     _func(Recursive)
```

(b) Indirect recursive function
This icon also indicates that the indicated function indirectly calls itself.

[Example]



```
void func1(int a)
{
    func2(10);
}
void func2(int b)
{
    func1(9);
}
```



```
{} _func1  (0x00000008 )
⊟ {} _func2  (0x00000004 )
       _func1(Recursive)
```

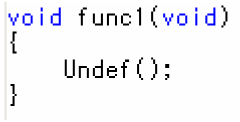
RTOS function (function of a realtime operating system such as ITRON)

[?] Unknown reference source function

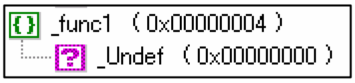
In the following example, the func1() function calls the Undef() function. However, if the Undef() function really does not exist, this icon is displayed for the Undef() function.

Calling a non-existing function results in a linkage error. However, by using the change_ message link option, you can change error messages to warning messages.
You can create load modules even if warning messages exist. Therefore, you can create stack information files as well.

[Example]



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

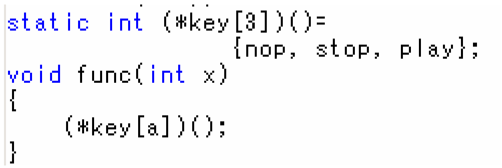


```
{} _func1 (0x00000004)
   └ [?] _Undef (0x00000000)
```

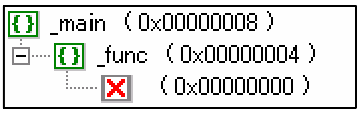
☒ Function with unresolved reference address

This icon is displayed when the indicated function is called from a table as shown below.

[Example]



```
static int (*key[3])()=
                    {nop, stop, play};
void func(int x)
{
    (*key[a])();
}
```



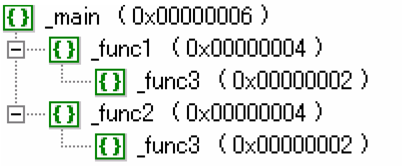
```
{} _main (0x00000008)
⊟ {} _func (0x00000004)
      └ ☒ (0x00000000)
```

Abbreviation icon

This tool displays all the link levels. If the user application is large, the number of link levels to be displayed is enormous.
Therefore, only the first symbols are displayed and other same symbols are abbreviated using the abbreviation icon.
To show all the symbols, choose [View]->[Show All Symbols].
To show part of the symbols, choose [View]->[Show Simple Symbols].

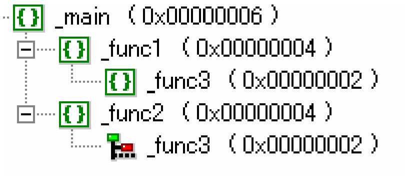
[Example]

Show All



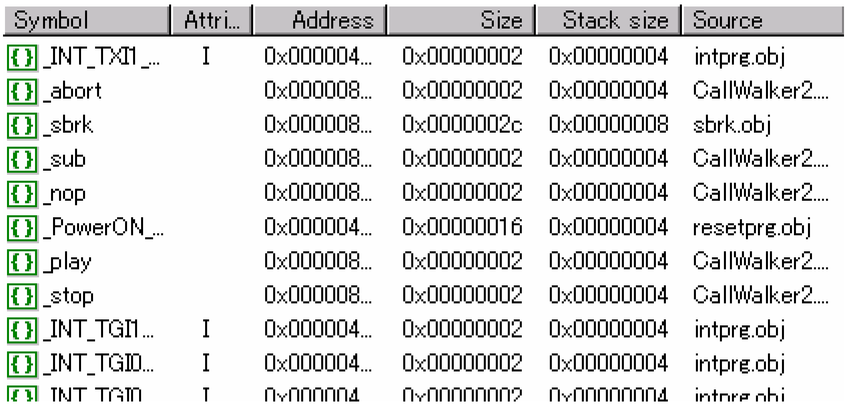
```
{} _main (0x00000006)
⊟ {} _func1 (0x00000004)
      └ {} _func3 (0x00000002)
⊟ {} _func2 (0x00000004)
      └ {} _func3 (0x00000002)
```

Show Simple



```
{} _main (0x00000006)
⊟ {} _func1 (0x00000004)
      └ {} _func3 (0x00000002)
⊟ {} _func2 (0x00000004)
      └ _func3 (0x00000002)
```

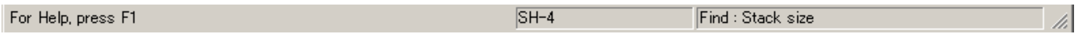
- Detailed symbol view



Symbol	Attri...	Address	Size	Stack size	Source
_INT_TXI1_...	I	0x000004...	0x00000002	0x00000004	intprg.obj
_abort		0x000008...	0x00000002	0x00000004	CallWalker2....
_sbrk		0x000008...	0x0000002c	0x00000008	sbrk.obj
_sub		0x000008...	0x00000002	0x00000004	CallWalker2....
_nop		0x000008...	0x00000002	0x00000004	CallWalker2....
PowerON...		0x000004...	0x00000016	0x00000004	resetprg.obj
_play		0x000008...	0x00000002	0x00000004	CallWalker2....
_stop		0x000008...	0x00000002	0x00000004	CallWalker2....
_INT_TGI1...	I	0x000004...	0x00000002	0x00000004	intprg.obj
_INT_TGI0...	I	0x000004...	0x00000002	0x00000004	intprg.obj
_INT_TGI0	I	0x000004	0x00000002	0x00000004	intprg.obj

This view shows the address, attributes, stack size, and other details about each symbol.
Click a symbol and then right-click to execute editing commands.

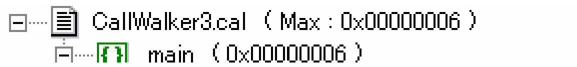
- Status bar



For Help, press F1 | SH-4 | Find : Stack size

The status bar shows the CPU type and other information about the stack information file (at the time of creation) that is currently open.

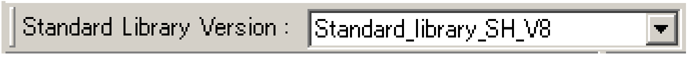
- Maximum stack size



CallWalker3.cal (Max : 0x00000006)
main (0x00000006)

"Max" indicates the maximum size of the statically-used stack in the currently open stack information file.

- Standard library version selection



Standard Library Version : Standard_library_SH_V8

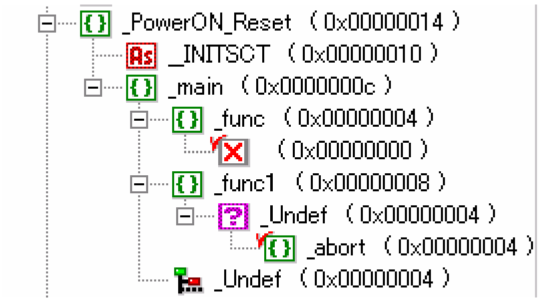
Select the standard library version that is used when you create the currently open stack information file.
The stack size used by the assembler functions in the standard library is determined by the version of the standard library.
You do not need to select any version when you install only one HEW package.

7.3.4 Editing Stack Information

While a file is open, you can select a desired symbol name from the detailed symbol view on the right to add, change, or delete the symbol using the Add..., Modify..., or Delete... command in the Edit menu.
You can also perform the same operations by right-clicking in the detailed symbol view.

Although this tool calculates the maximum size of the statically-used stack, the user needs to edit the information file to determine the maximum size of the dynamically-used stack due to multiple interrupts and other reasons.

You can change the positions of symbols by dragging and dropping the desired symbol in the call information view on the left.
When you move or edit a symbol, a check mark appears next to the corresponding symbol in the call information view in the left.



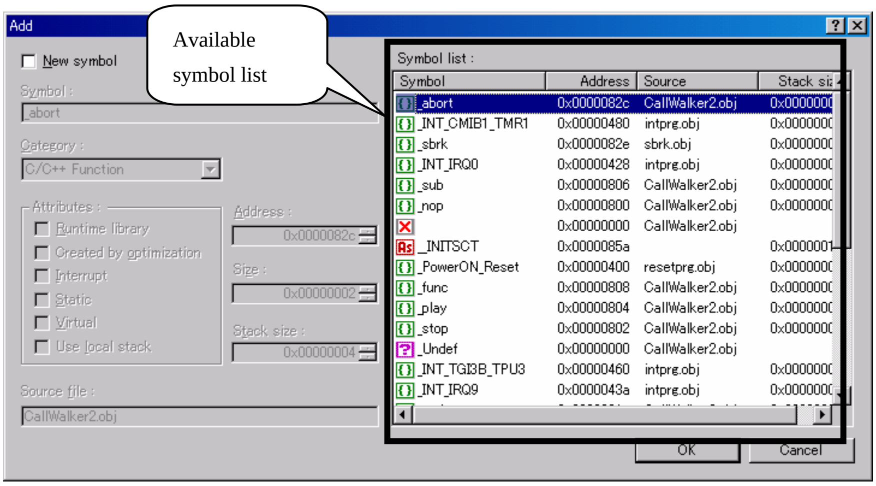
```
_PowerON_Reset (0x00000014)
    _INITSCT (0x00000010)
    _main (0x0000000c)
        _func (0x00000004)
            (0x00000000)
        _func1 (0x00000008)
            _Undef (0x00000004)
                _abort (0x00000004)
        _Undef (0x00000004)
```

The following sections describe the available commands.

- Add... command

(1) Adding an existing symbol

When you click the Add... command, the following dialog box appears. The list on the right shows the symbols in the current file. To add an existing symbol, select a desired symbol from the list and click the [OK] button.



□ New symbol

Symbol : _abort

Category : C/C++ Function

Attributes :
□ Runtime library
□ Created by optimization
□ Interrupt
□ Static
□ Virtual
□ Use local stack

Address : 0x0000082c

Size : 0x00000002

Stack size : 0x00000004

Source file : CallWalker2.obj

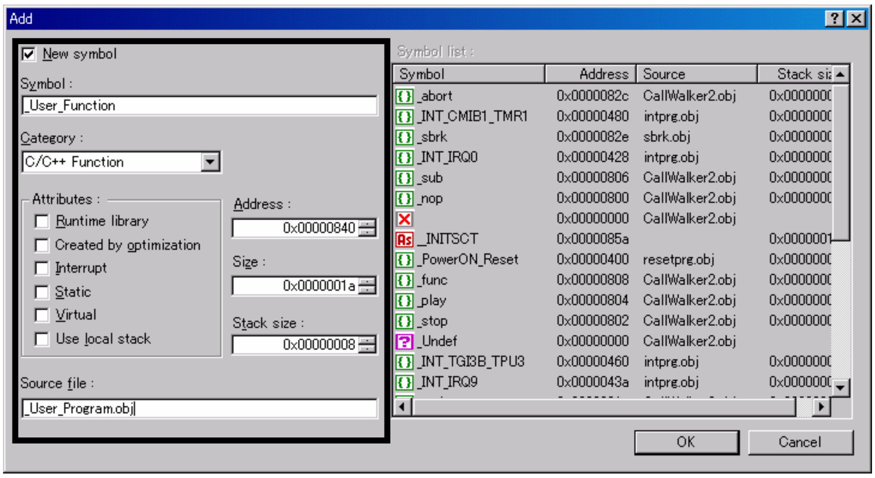
Symbol list :

Symbol	Address	Source	Stack si
_abort	0x0000082c	CallWalker2.obj	0x0000000
_INT_CMIB1_TMR1	0x00000480	intprg.obj	0x0000000
_sbrk	0x0000082e	sbrk.obj	0x0000000
_INT_IRQ0	0x00000428	intprg.obj	0x0000000
_sub	0x00000806	CallWalker2.obj	0x0000000
_nop	0x00000800	CallWalker2.obj	0x0000000
☒	0x00000000	CallWalker2.obj	
_INITSCT	0x0000085a		0x0000001
_PowerON_Reset	0x00000400	resetprg.obj	0x0000000
_func	0x00000808	CallWalker2.obj	0x0000000
_play	0x00000804	CallWalker2.obj	0x0000000
_stop	0x00000802	CallWalker2.obj	0x0000000
_Undef	0x00000000	CallWalker2.obj	
_INT_TGI3B_TPU3	0x00000460	intprg.obj	0x0000000
_INT_IRQ9	0x0000043a	intprg.obj	0x0000000

(2) Adding a new symbol

When you select the [New symbol] check box on the left, you can create a new symbol.

At the same time, you can define the symbol name, symbol category, attributes, address, stack size, and other details.



☑ New symbol

Symbol : User_Function

Category : C/C++ Function

Attributes :
□ Runtime library
□ Created by optimization
□ Interrupt
□ Static
□ Virtual
□ Use local stack

Address : 0x00000840

Size : 0x0000001a

Stack size : 0x00000008

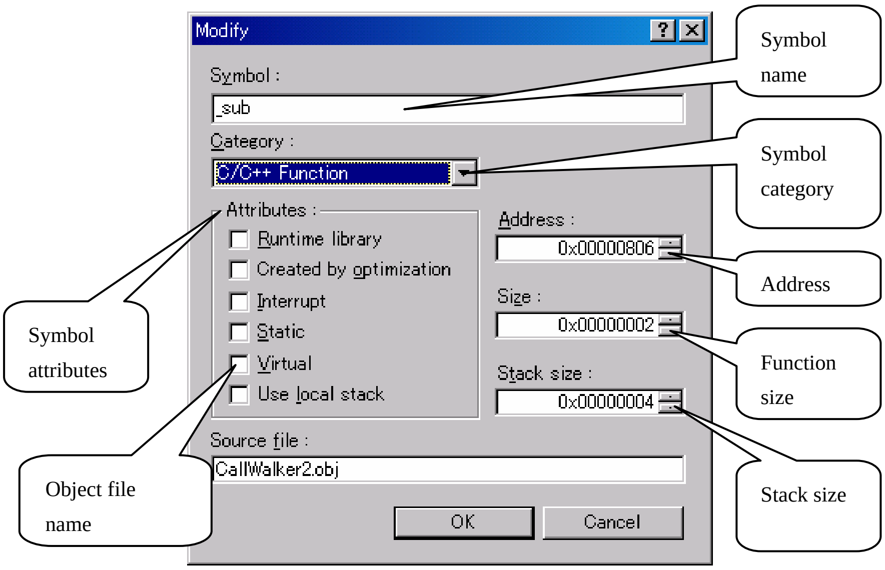
Source file : User_Program.obj

Symbol list :

Symbol	Address	Source	Stack si
_abort	0x0000082c	CallWalker2.obj	0x0000000
_INT_CMIB1_TMR1	0x00000480	intprg.obj	0x0000000
_sbrk	0x0000082e	sbrk.obj	0x0000000
_INT_IRQ0	0x00000428	intprg.obj	0x0000000
_sub	0x00000806	CallWalker2.obj	0x0000000
_nop	0x00000800	CallWalker2.obj	0x0000000
☒	0x00000000	CallWalker2.obj	
_INITSCT	0x0000085a		0x0000001
_PowerON_Reset	0x00000400	resetprg.obj	0x0000000
_func	0x00000808	CallWalker2.obj	0x0000000
_play	0x00000804	CallWalker2.obj	0x0000000
_stop	0x00000802	CallWalker2.obj	0x0000000
_Undef	0x00000000	CallWalker2.obj	
_INT_TGI3B_TPU3	0x00000460	intprg.obj	0x0000000
_INT_IRQ9	0x0000043a	intprg.obj	0x0000000

- Modify... command

Select the symbol whose information you want to change and click the [Modify...] command. The following dialog box appears. You can modify several information items.



Modify

Symbol : _sub

Category : C/C++ Function

Attributes :
- □ Runtime library
- □ Created by optimization
- □ Interrupt
- □ Static
- □ Virtual
- □ Use local stack

Address : 0x00000806

Size : 0x00000002

Stack size : 0x00000004

Source file : CallWalker2.obj

- Delete... command

To delete the symbols that are unnecessary for determining the stack size, select such a symbol (in the left or right view) and click the [Delete...] command.

7.3.5 Stack Area Size of Assembly Program

Unlike by C/C++ program, the stack area size used by assembly program cannot be calculated automatically in assembling. Therefore the stack area size used by assembly functions should be edited by using Call Walker.
But the stack area size is specified in the assembly function by using **.STACK** directive. Call Walker displays the value specified by **.STACK** directive.

- Description of **.STACK** directive

Defines the stack amount for a specified symbol referenced by using Call Walker.

The stack value for a symbol can be defined only one time; the second and later specifications for the same symbol are ignored. A multiple of 2 in the range from H'00000000 to H'FFFFFFFE can be specified for the stack value, and any other value is invalid.

The stack value must be specified as follows:
- A constant value must be specified.
- Forward reference symbol, external reference symbol and relative address symbol must not be used.

- Specification Method of **.STACK** assembler directive

  •.STACK <symbol> = <stack value>

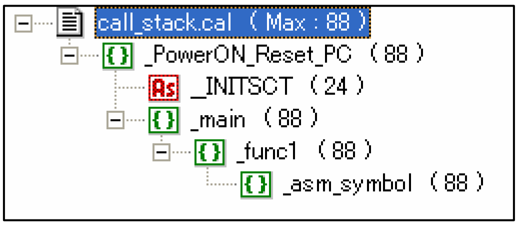
- Example of assembly program

```
        .EXPORT     _asm_symbol
        .SECTION    P,CODE,ALIGN=4
_asm_symbol:
        .STACK      _asm_symbol=88
          :
        RTS
        NOP
        .END
```

← Stack Size of **_asm_symbol** function

- Displayed Example by Call Walker

As the following example, the stack area size used by _asm_symbol function is displayed “88” in Call Walker.



```
⊟ call_stack.cal （ Max : 88 ）
  ⊟ {} _PowerON_Reset_PC （ 88 ）
       As _INITSCT （ 24 ）
     ⊟ {} _main （ 88 ）
        ⊟ {} _func1 （ 88 ）
             {} _asm_symbol （ 88 ）
```

- Remarks

(1) **.STACK** assembler directive can only make Call Walker display stack size, and does NOT affect the behavior of program.

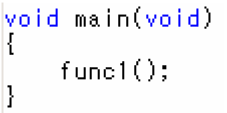
(2) This assembler directive is supported in SuperH RISC engine Assembler Ver.7.00 or later.

7.3.6 Merging Stack Information

You can merge a stack information file that is saved or being edited with another stack information file. By doing so, the edited stack information is not overwritten by the post-build stack information.

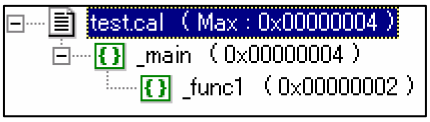
- Merge example

(1) Contents of test.c



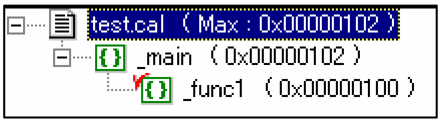
```
void main(void)
{
    func1();
}
```

(2) Open a stack information file from Call Walker.



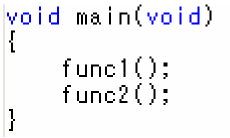
```
⊟ test.cal （ Max : 0x00000004 ）
  ⊟ {} _main （ 0x00000004 ）
       {} _func1 （ 0x00000002 ）
```

(3) Change the contents of the file (change the stack size of func1 to 100).



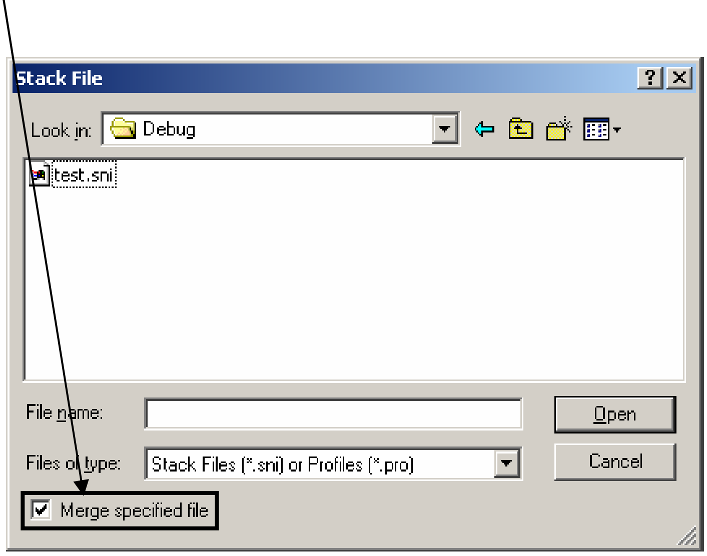
```
test.cal （Max : 0x00000102 ）
    _main （0x00000102 ）
        _func1 （0x00000100 ）
```

(4) Change the contents of test.c and perform build (**add a call for func2**).



```
void main(void)
{
    func1();
    func2();
}
```

(5) Open test.sni while test.cal is open in Call Walker.
Select here and choose the [Open] button.



Stack File

Look in: Debug

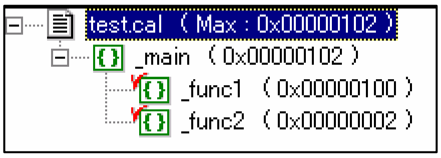
test.sni

File name:

Files of type: Stack Files (*.sni) or Profiles (*.pro)

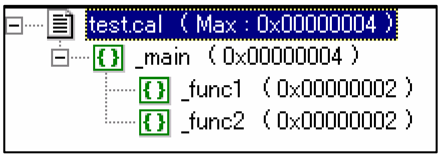
☑ Merge specified file

(6) The information of func2 is added while keeping the stack size of func1 changed in step (3). This is merging of stack information.



```
test.cal （Max : 0x00000102 ）
    _main （0x00000102 ）
        _func1 （0x00000100 ）
        _func2 （0x00000002 ）
```

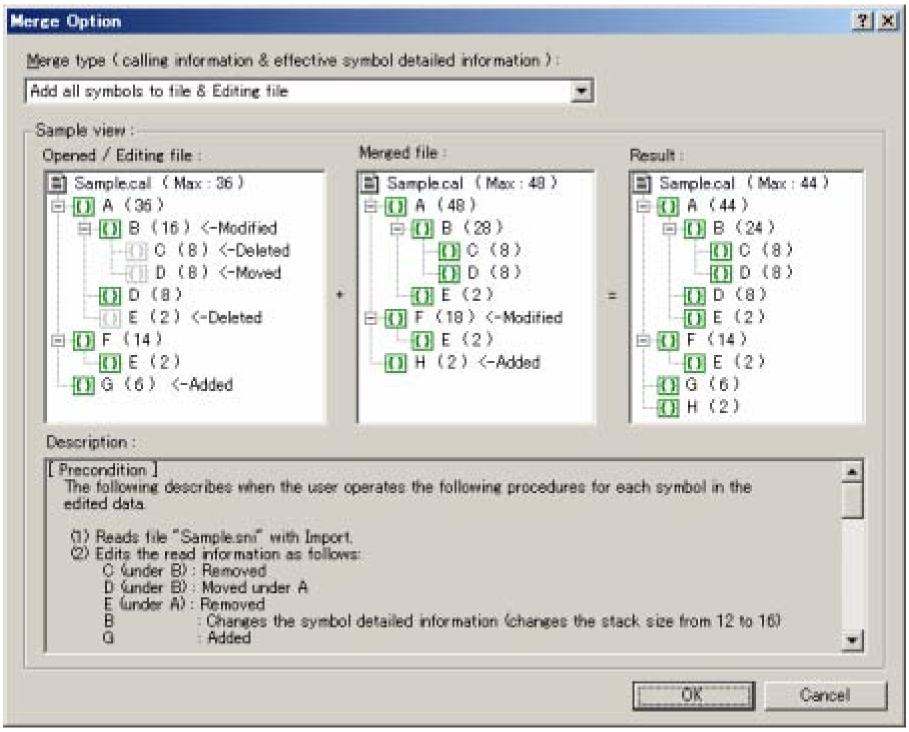
If you do not select the [Merge specified file] check box in step (5), the stack size of func1 changed in step (3) returns to the previous value.



```
test.cal （Max : 0x00000004 ）
    _main （0x00000004 ）
        _func1 （0x00000002 ）
        _func2 （0x00000002 ）
```

- Detailed merge options

You can change the method of merging. Five methods are available.
For details about merge methods, read [Description] in the following dialog box.
How to specify a merge method
[Tools]-> [Merge Option...]



Merge type (calling information & effective symbol detailed information) : Add all symbols to file & Editing file

Sample view :

```
Opened / Editing file :                 Merged file :                    Result :
Sample.cal ( Max : 36 )            +    Sample.cal ( Max : 48 )     =    Sample.cal ( Max : 44 )
  A ( 36 )                                A ( 48 )                         A ( 44 )
    B ( 16 ) <-Modified                     B ( 28 )                         B ( 24 )
      C ( 8 ) <-Deleted                       C ( 8 )                          C ( 8 )
      D ( 8 ) <-Moved                         D ( 8 )                          D ( 8 )
    D ( 8 )                                 E ( 2 )                          D ( 8 )
    E ( 2 ) <-Deleted                     F ( 18 ) <-Modified                E ( 2 )
  F ( 14 )                                  E ( 2 )                        F ( 14 )
    E ( 2 )                               H ( 2 ) <-Added                    E ( 2 )
  G ( 6 ) <-Added                                                          G ( 6 )
                                                                           H ( 2 )
```

Description :

```
[ Precondition ]
  The following describes when the user operates the following procedures for each symbol in the
  edited data.

   (1) Reads file "Sample.sni" with Import.
   (2) Edits the read information as follows:
        C (under B) : Removed
        D (under B) : Moved under A
        E (under A) : Removed
        B             : Changes the symbol detailed information (changes the stack size from 12 to 16)
        G             : Added
```

- Note

The merge feature is available in Call Walker version 1.3 or later.

7.3.7 Other Features

- Realtime operating system icon

You can show the icon of the realtime operating system as RTOS in the call information view in the left of the window.

[How to specify]
[Tools]-> [Realtime OS Option...]

This file with the csv extension is packaged in each realtime operating system product.

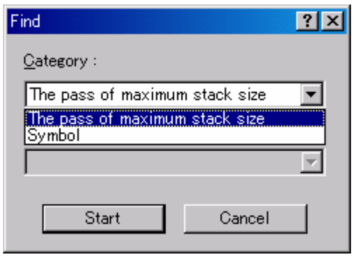
- Outputting lists

You can output stack information in text format in a file.

[How to output]
[File]->[Output List...]

- Search feature

You can find the following two items from the call information view by specifying the desired target in the following dialog box.


Find
Category :
The pass of maximum stack size
The pass of maximum stack size
Symbol
Start
Cancel

(1) Pass with the maximum stack size

(2) Symbol name

[How to specify]|
[Edit]->[Find...]
[Edit]->[Find Next...] (find the next item)
[Edit]->[Find Previous...] (find the previous item)

- Setting the display format for the call information view

You can use the following two commands to select the format for displaying stack sizes:

(1) Show Required Stack

The largest stack size is shown at the top and the smallest stack size is shown at the bottom.

(2) Show Used Stack

The smallest stack size is shown at the top and the largest stack size is shown at the bottom.

[How to specify]
[View]->[Show Required Stack] or [Show Used Stack]

Section 8 Efficient C++ Programming Techniques

The Compiler supports the C++ and C languages.
This chapter describes in detail the options of an object-oriented language C++ and how to use the various C++ functions.

Code a C++ program for an embedded system with caution. Otherwise, the program will have a larger object size or a lower processing speed than expected.
Therefore, this chapter presents some cases in which the performance of a C++ program is deteriorated compared with C as well as codes with which you can work around such performance deterioration.

The following table shows a list of efficient C++ programming techniques:

No.	Category	Item	Section
1	Initialization Processing/Post-processing	Initialization Processing and Post-Processing of Global Class Object	8.1.1
2	Introduction to C++ Functions	How to Reference a C Object	8.2.1
3		How to Implement *new* and *delete*	8.2.2
4		Static Member Variable	8.2.3
5	How to Use Options	C++ Language for Embedded Applications	8.3.1
6		Run-Time Type Information	8.3.2
7		Exception Handling Function	8.3.3
8		Disabling Startup of Prelinker	8.3.4
9	Advantages and Disadvantages of C++ Coding	Constructor (1)	8.4.1
10		Constructor (2)	8.4.2
11		Default Parameter	8.4.3
12		Inline Expansion	8.4.4
13		Class Member Function	8.4.5
14		*operator* Operator	8.4.6
15		Function Overloading	8.4.7
16		Reference Type	8.4.8
17		Static Function	8.4.9
18		Static Member Variable	8.4.10
19		Anonymous *union*	8.4.11
20		Virtual Function	8.4.12

8.1 Initialization Processing/Post-processing

8.1.1 Initialization Processing and Post-Processing of Global Class Object

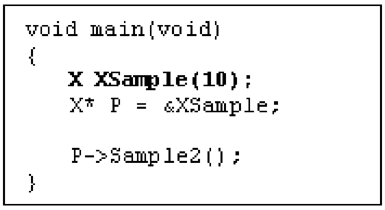
- Important Points:

To use a global class object in C++, you need to call the initialization processing function (_CALL_INIT) and the post-processing function (_CALL_END) before and after the *main* function, respectively.

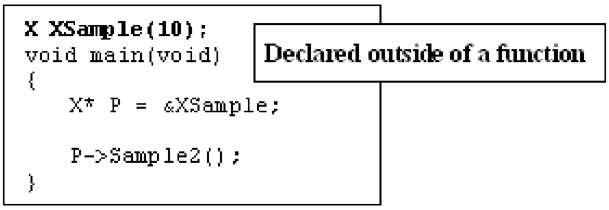
- What is a global class object?

A global class object is a class object that is declared outside of a function.

(Class object declaration inside a function) (Global class object declaration)



```
void main(void)
{
    X XSample(10);
    X* P = &XSample;

    P->Sample2();
}
```



```
X XSample(10);
void main(void)
{
    X* P = &XSample;

    P->Sample2();
}
```

Declared outside of a function

- Why is initialization processing/post-processing necessary?

If a class object is declared inside a function as shown above, the constructor of class *X* is called when function *main* is executed.
In contrast, a global class object declaration is not executed even when a function is executed.
Thus, you need to call _CALL_INIT before calling the *main* function in order to explicitly call the constructor of class *X*.
Likewise, call _CALL_END after calling the *main* function in order to call the destructor of class *X*.

- Operations when using and not using _CALL_INIT/_CALL_END:

The following shows the values obtained when the value of member variable *x* of class *X* is referenced.
When not using _CALL_INIT/_CALL_END, no correct value can be obtained and no expression in the *while* statement will be executed as follows:

(Value of member variable *x*)
  When using **_CALL_INIT   --> 10**

  When not using **_CALL_INIT --> 0**

```
class X{
    int x;
public:
      X(int n){x = n}; // constructor
      ~X(){}           // destructor
    void Sample2(void);
};
X XSample(10); // global class object
void X::Sample2(void)
{
    while(x == 10)
    {
    }
}
void main(void)
{
    X* P = &XSample;

    P->Sample2();
}
```

<-- Reference

- How to call _CALL_INIT/_CALL_END:

Provide the following code before and after calling the *main* function.

```
void INIT(void)
{
   _INITSCT();
   _CALL_INIT();
   main();
   _CALL_END();
}
```

If HEW is used, remove the comment characters in the section for calling _CALL_NIT/_CALL_END of *resetprg.c*.

(*PowerON_Reset* function of *resetprg.c*)

```
__entry(vect=0) void PowerON_Reset(void)
{
   set_imask_ccr(1);
   _INITSCT();

// _CALL_INIT();      // Remove the comment when you use global class object

// _INIT_IOLIB();     // Remove the comment when you use SIM I/O

// errno=0;           // Remove the comment when you use errno
// srand(1);          // Remove the comment when you use rand()
// _s1ptr=NULL;       // Remove the comment when you use strtok()

   HardwareSetup();  // Use Hardware Setup
   set_imask_ccr(0);

   main();

// _CLOSEALL();       // Remove the comment when you use SIM I/O

// _CALL_END();       // Remove the comment when you use global class object

   sleep();
}
```

8.2 Introduction to C++ Functions

8.2.1 How to Reference a C Object

• Important Points:

Use an '*extern* "*C*"' declaration to directly use in a C++ program the resources in an existing C object program. Likewise, the resources in a C++ object program can be used in a C program.

• Example of Use:

1. Use an '*extern* "*C*"' declaration to reference a function in a C object program.

```
(C++ program)

extern "C" void CFUNC();
void main(void)
{
    X XCLASS;
    XCLASS.SetValue(10);

    CFUNC();
}
```

```
(C program)

extern void CFUNC();
void CFUNC()
{
    while(1)
    {
        a++;
    }
}
```

2. Use an 'extern "C"' declaration to reference a function in a C++ object program.

```
(C program)

void CFUNC()
{
    CPPFUNC();
}
```

```
(C++ program)

extern "C" void CPPFUNC();
void CPPFUNC(void)
{
    while(1)
    {
        a++;
    }
}
```

• Important Information:

1. A C++ object generated by a previous-version(Ver.5) compiler cannot be linked because the encoding and executing methods have been changed.
   Be sure to recompile it before using it.
2. A function called in the above method cannot be overloaded.

8.2.2 How to Implement new and delete

- Important Points:

To use *new*, implement a low-level function.

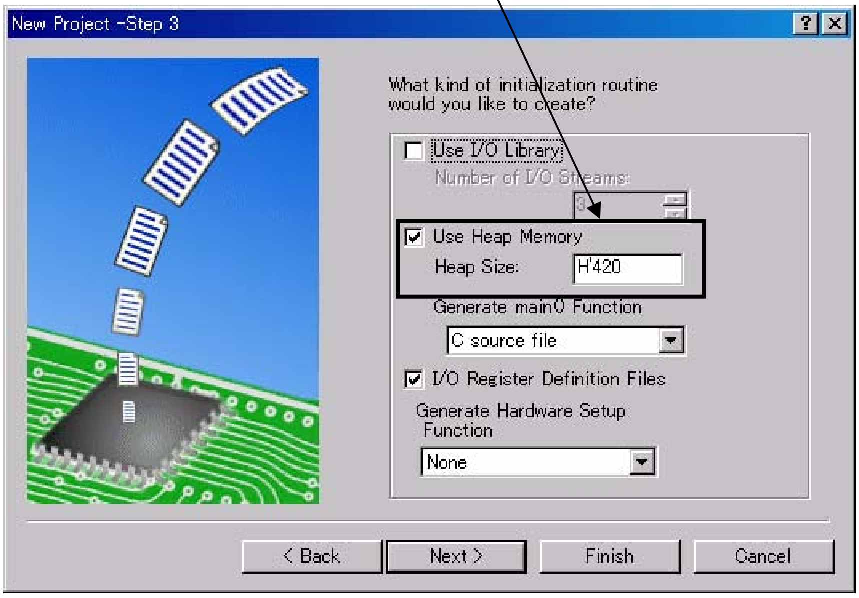
- Description:

If *new* is used in an embedded system, the dynamic allocation of actual heap memory is realized using *malloc*.
Thus, implement a low-level interface routine (*sbrk*) to specify the size of heap memory to be allocated just as when using *malloc*.

- Implementation Method:

To use HEW, make sure that [Use Heap Memory] is checked when a workspace is created.
If this option is checked, *sbrk.c* and *sbrk.h* shown on the next page will be automatically created.
Specify the size of heap memory to be allocated in Heap Size.
To change the size after creating a workspace, change the value defined in HEAPSIZE in *sbrk.h*.

If HEW is not used, create a file shown on the next page and implement it in a project.



What kind of initialization routine would you like to create?

□ Use I/O Library

Number of I/O Streams: 3

☑ Use Heap Memory

Heap Size: H'420

Generate main() Function: C source file

☑ I/O Register Definition Files

Generate Hardware Setup Function: None

```
(sbrk.c)

#include <stdio.h>
#include "sbrk.h"

//const size_t _sbrk_size=  /* Specifies the minimum unit of    */
                            /* the defined heap area            */

static  union  {
   long  dummy ;          /* Dummy for 4-byte boundary          */
   char heap[HEAPSIZE];   /* Declaration of the area managed    */
                          /*                           by sbrk */
 }heap_area ;

static  char  *brk=(char *)&heap_area;/* End address of area assigned     */

/**************************************************************************/
/*     sbrk:Data write                                                    */
/*     Return value:Start address of the assigned area (Pass)             */
/*                    -1                (Failure)                         */
/**************************************************************************/
char  *sbrk(size_t size)    /* Assigned area size               */
{
   char  *p;

   if(brk+size>heap_area.heap+HEAPSIZE) /* Empty area size      */
     return (char *)-1 ;

   p=brk ;                  /* Area assignment                   */
   brk += size ;            /* End address update                 */
   return p ;
}
```

```
(sbrk.h)

/* size of area managed by sbrk */
#define HEAPSIZE 0x420
```

8.2.3 Static Member Variable

• Description:

In C++, a class member variable with the *static* attribute can be shared among multiple objects of a class type.
Thus, a static member variable comes in handy because it can be used, for example, as a common flag among multiple objects of the same class type.

• Example of Use:

Create five class-A objects within the main function.
Static member variable *num* has an initial value of 0. This value will be incremented by the constructor every time an object is created.
Static member variable num, shared among objects, will have a value of 5 at the maximum.

• FAQ:

The following lists some frequently asked questions on using a static member variable.

[L2310 Error Occurred]
When a static member variable is used, message "** L2310 (E) Undefined external symbol "class-name::static-member-variable-name" referenced in "file-name"" is output at linkage.

[Solution]

This error occurs because the static member variable is not defined.
Add either of the following definition as shown on the next page:

If there is an initial value: `int A::num = 0;`
If there is no initial value: `int A::num = 0;`

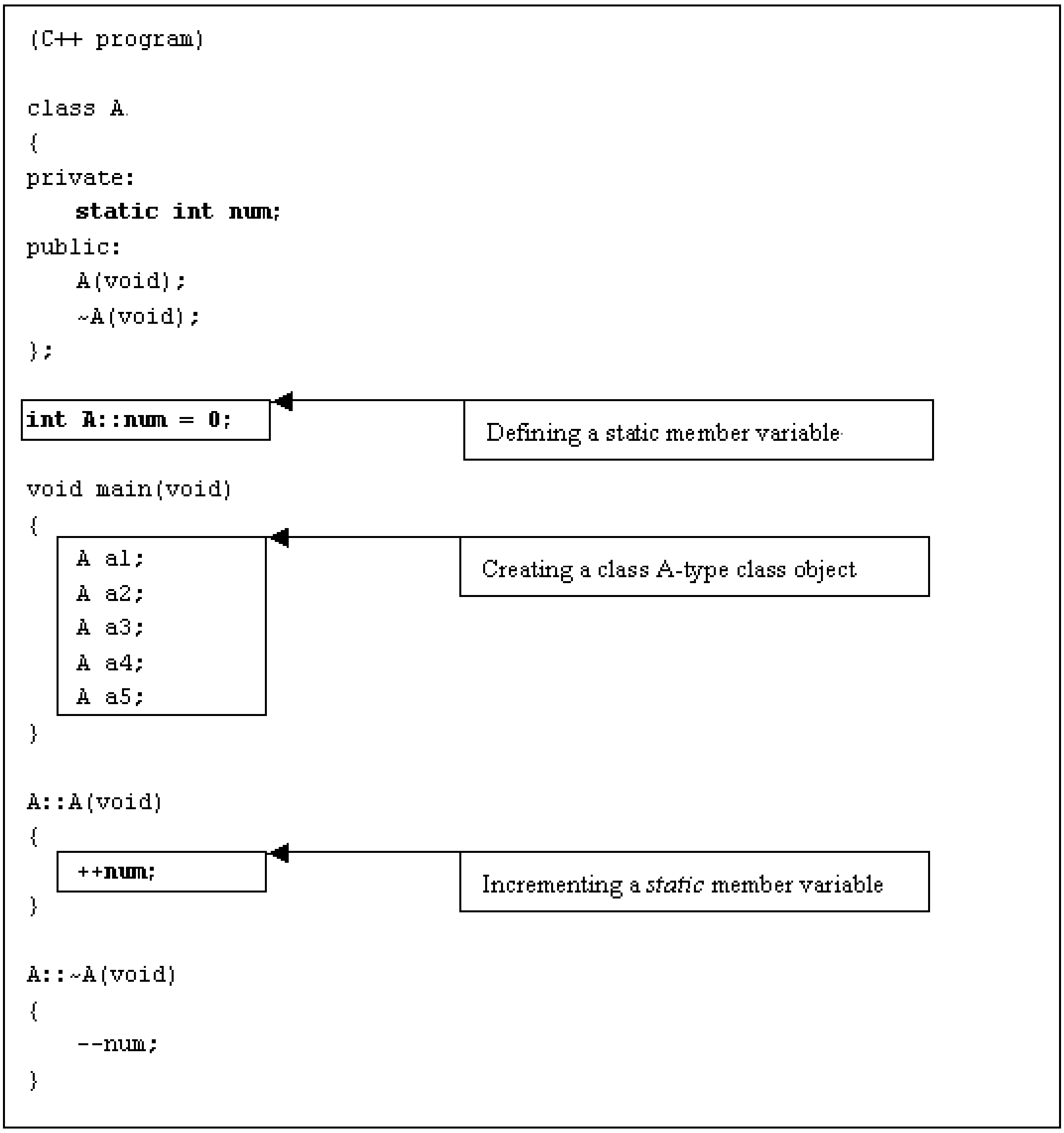
[Unable to assign an initial value]

No initial value is assigned to a *static* member variable to be initialized.

[Solution]

A *static* member variable to be initialized, handled as a variable with an initial value, is created in the D-section by default. Thus, specify the ROM implementation support option of the optimization linkage editor and, in the initial routine, copy the D-section from the ROM to the RAM using the *_INITSCT* function*.

Note: * This solution is not required if HEW automatically creates an initial routine.



```
(C++ program)

class A
{
private:
    static int num;
public:
    A(void);
    ~A(void);
};

int A::num = 0;

void main(void)
{
    A a1;
    A a2;
    A a3;
    A a4;
    A a5;
}

A::A(void)
{
    ++num;
}

A::~A(void)
{
    --num;
}
```

Defining a static member variable
Creating a class A-type class object
Incrementing a *static* member variable

8.3 How to Use Options

8.3.1 C++ Language for Embedded Applications

- Description:

The ROM/RAM sizes and the execution speed are important for an embedded system.
The C++ language for embedded applications (EC++) is a subset of the C++ language. For EC++, some of the C++ functions not appropriate for an embedded system have been removed.
Using EC++, you can create an object appropriate for an embedded system.

- Specification method:

Dialog menu: C/C++ tab Category: Other tab, Check against EC++ language specification
Command line: *eccp*

- Unsupported keywords:

An error message will be output if either of the following keywords is included.

catch, const_cast, dynamic_cast, explicit, mutable, namespace, reinterpret_cast, static_cast, template, throw, try, typeid, typename, using

- Unsupported language specifications:

A warning message will be output if either of the following language specifications is included.

Multiple inheritance, virtual base class

8.3.2 Run-Time Type Information

- Description:

In C++, a class object with a virtual function may have a type identifiable only at run-time.
A run-time identification function is available to provide support in such a situation.
To use this function in C++, use the *type_info* class, *typeid* operator, and *dynamic_cast* operator.
For the Compiler, specify the following option to use run-time type information.
Additionally, specify the following option at linkage to start up the prelinker.

- Specification method:

Dialog menu: CPU tab, Enable/disable runtime type information
Command line: *rtti=on* | *off*

Dialog menu: Link/Library tab, Category: Input tab, Prelinker control
Then, select Auto or Run prelinker.
Command line: *Do not specify noprelink (default).*

- Example of Use of *type_info* Class and *typeid* Operator:

The *type_info* class is intended to identify the run-time type of an object.
Use the *type_info* class to compare types at program execution or acquire a class type.
To use the *type_info* class, specify a class object with a virtual function using the typeid operator.

```
#include <typeinfo.h>                         Must be included
#include <string>
class Base{
protected:
    string *pname1                            Base class
public:
    Base() {
       pname1 = new string;
       if (pname1)                            Virtual function
           *pname1 = "Base";
    }
    virtual string Show() {return *pname1}
    virtual ~Base() {
       if (pname1                             Virtual destructor
           delete pname1;
    }
};                                   Derived class
class Derived : public Base{
   string *pname2;
public:
   Derived() {
      pname2 = new string;
      if (pname2)                             Virtual function
          *pname2 = "Derived";
   }
   string Show() {return *pname2;}
   ~Derived() {;
      if (pname2)                             Virtual destructor
          delete pname2;
   }
};
void main(void)
{
   Base* pb = new Base;
   Derived* pd = new Derived;                 Specifying a class
                                              object
   const type_info& t = typeid(pb);
   const type info& t1 = typeid(pd);
   t.name();                                  Acquiring type name [Base*]
   t1 name();                                 Acquiring type name [Derived*]
}
```

- Example of Use of *dynamic_cast* Operator:

Use the *dynamic_cast* operator, for example, to cast at run-time a pointer or reference of the derived-class type to a pointer or reference of the base-class type between a class including a virtual function and its derived class.

```
#include <string>
class Base{
protected:
    string *pname1;
public:
    Base() {
        pname1 = new string;
        if (pname1)
            *pname1 = "Base";
    }
    virtual string Show() {return *pname1;}
    virtual ~Base() {
        if (pname1)
            delete pname1;
    }
};
class Derived : public Base{
    string *pname2;
public:
    Derived() {
        pname2 = new string;
        if (pname2)
            *pname2 = "Derived";
    }
    string Show() {return *pname2;}
    ~Derived() {
        if (pname2)
            delete pname2;
    }
};
void main(void)
{
    Derived *pderived = new Derived;
    Base *pbase = dynamic_cast<Base *> (pderived);

    string ddd;
    ddd = pbase-> Show();

    delete pbase;
}
```

Base class

Virtual function

Virtual destructor

Derived class

Virtual function

Virtual destructor

Cast to Base * at run-time

Acquiring class name Base

8.3.3 Exception Handling Function

- Description:

Unlike C, C++ has a mechanism for handling an error called an exception.
An exception is a means for connecting an error location in a program with an error handling code.
Use the exception mechanism to put together error handling codes in one location.
For the Compiler, specify the following option to use the exception mechanism.

- Specification method:

Dialog menu: CPU tab, Use try, throw and catch of C++
Command line: *exception*

- Example of Use:

If opening of file "INPUT.DAT" fails, initiate the exception handling and display an error in the standard error output.

```
 (C++ program example for exception handling)

void main(void)
{
    try
    {
        if ((fopen("INPUT.DAT","r"))==NULL){
            char * cp = "cannot open input file\n";
            throw cp;
        }
    }
    catch(char *pstrError)
    {
        fprintf(stderr,pstrError);
        abort();
    }
    return;
}
```

- Important Information:

The coding performance may deteriorate.

8.3.4 Disabling Startup of Prelinker

- Description:

Starting up the Prelinker will reduce the link speed. The Prelinker need not be running unless the template function or run-time type conversion of C++ is used.
To use the Linker from a command line, specify the following *noprelink* option.
If Hew is used and the *Prelinker control* list box is set to Auto, the output of the *noprelink* option will be automatically controlled.

- Specification method:

Dialog menu: Link/Library tab, Category: Input tab, Prelinker control
Command line: *noprelink*

8.4 Advantages and Disadvantages of C++ Coding

The Compiler, when compiling a C++ program, internally converts the C++ program to a C program to create an object. This chapter compares a C++ program and a C program after conversion and describes the influences on coding efficiency of each function.

No.	Function	Development and maintenance	Size Reduction	Speed	Section
1	Constructor (1)	◎	Δ	Δ	8.4.1
2	Constructor (2)	◎	Δ	Δ	8.4.2
3	Default parameter	◎	○	○	8.4.3
4	Inline expansion	○	Δ	○	8.4.4
5	Class member function	◎	Δ	Δ	8.4.5
6	*operator* Operator	◎	Δ	Δ	8.4.6
7	Function overloading	◎	○	○	8.4.7
8	Reference type	◎	○	○	8.4.8
9	Static function	◎	○	○	8.4.9
10	Static member variable	◎	○	○	8.4.10
11	Anonymous *union*	◎	○	○	8.4.11
12	Virtual function	◎	Δ	Δ	8.4.12

◎Same as C
○Requiring caution in use
ΔPerformance decrease

8.4.1 Constructor (1)

Development and Maintenance	◎	Size Reduction	Δ	Speed	Δ

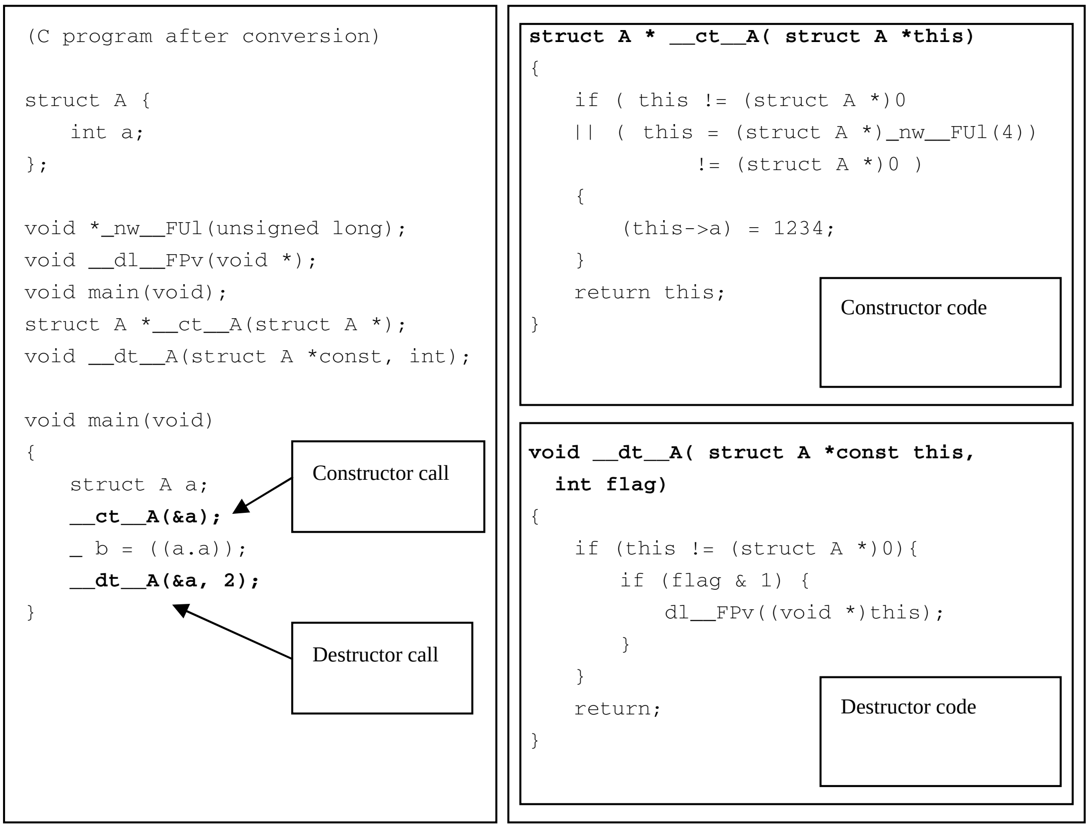
- Important Points:

Use a constructor to automatically initialize a class object. However, use it with caution because it will influence the object size and processing speed as follows:

- Example of Use:

Create a class-A constructor and destructor and compile them.The size and processing speed will be influenced because the constructor and destructor will be called in the class declaration and decisions will be made in the constructor and destructor codes.

```
(C++ program)

class A
{
private:
    int a;
public:
    A(void);
    ~A(void);
    int getValue(void){ return a; }
};

void main(void)
{
    A a;
    b = a.getValue();
}

A::A(void)
{
    a = 1234;
}

A::~A(void)
{
}
```



```
(C program after conversion)

struct A {
    int a;
};

void *_nw__FUl(unsigned long);
void __dl__FPv(void *);
void main(void);
struct A *__ct__A(struct A *);
void __dt__A(struct A *const, int);

void main(void)
{
    struct A a;
    __ct__A(&a);
    _ b = ((a.a));
    __dt__A(&a, 2);
}
```

```
struct A * __ct__A( struct A *this)
{
    if ( this != (struct A *)0
    ||  ( this = (struct A *)_nw__FUl(4))
             != (struct A *)0 )
    {
        (this->a) = 1234;
    }
    return this;
}
```

```
void __dt__A( struct A *const this,
  int flag)
{
    if (this != (struct A *)0){
        if (flag & 1) {
            dl__FPv((void *)this);
        }
    }
    return;
}
```

8.4.2 Constructor (2)

Development and Maintenance	◎	Size Reduction	Δ	Speed	Δ

- Important Points:

To declare a class in an **array**, use a constructor to automatically initialize a class object. However, use it with caution because it will influence the object size and processing speed as follows:

- Example of Use:

Create a class-A constructor and destructor and compile them.The memory needs to be dynamically allocated and deallocated because the constructor and destructor are called in the class declaration but are declared in the array.
Use *new* and *delete* to dynamically allocate and deallocate the memory.
This requires implementation of a low-level function. (For details on the implementation method, refer to section 8.1.2, Execution Environment Settings, in the SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual.)
The size and processing speed will be influenced because decisions and the low-level function processing are added in the constructor and destructor codes.

```
(C++ program)
class A
{
private:
    int a;
public:
    A(void);
    ~A(void);
    int getValue(void){ return a; }
};

void main(void)
{
    A a[5];
    b = a[0].getValue();
}

A::A(void)
{
    a = 1234;
}

A::~A(void)
{
}
```

```
(C program after conversion)
struct A {
    int a;
};

void *__nw__FUl(unsigned long);
void __dl__FPv(void *);
void main(void);
void *__vec_new();
void __vec_delete();
struct A *__ct__A(struct A *);
void __dt__A(struct A *const, int);

void main(void)
{
    struct A a[5];
    __vec_new( (struct A *)a, 5, 4,
__ct__A);
    _ b = ((a.a));
    __vec_delete( &a, 5, 4, __dt__A, 0,
0);
}
```

Constructor call

Destructor call

```
struct A *__ct__A( struct A *this)
{
    if((this != (struct A *)0)
     || ( (this = (struct A
*)__nw__FUl(4)) != (struct A *)0) )
    {
        (this->a) = 1234;
    }
    return this;
}
```

Constructor code

```
void __dt__A( struct A *const this,
 int flag)
{
    if (this != (struct A *)0){
        if (flag & 1){
            __dl__FPv((void *)this);
        }
    }
    return;
}
```

Destructor code

8.4.3 Default Parameter

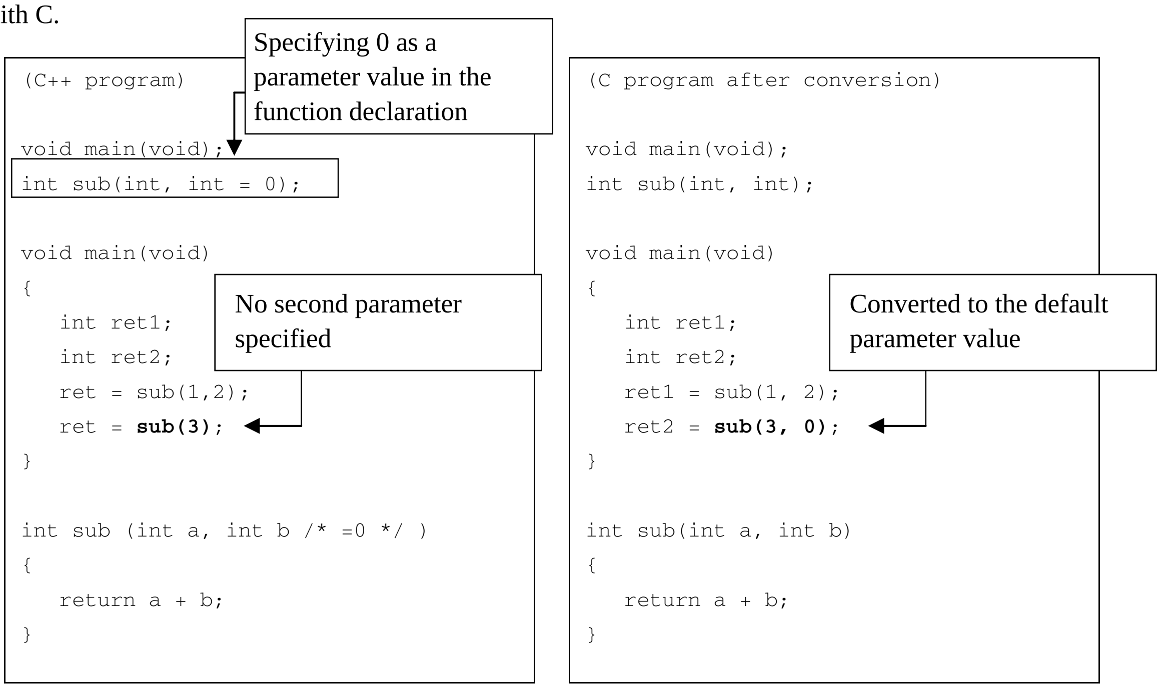
Development and Maintenance	◎	Size Reduction	○	Speed	○

• Important Points:

In C++, a default parameter can be used to set a default used when calling a function.
To use a default parameter, specify a default value for parameters of a function when declaring the function.
This will eliminate the need of specifying a parameter in many of the function calls and enable the use of a default parameter instead, thus improving the development efficiency.
A parameter value can be changed if a parameter is specified.

• Example of Use:

The following shows an example of calling function *sub* when 0 is specified as a default parameter value in the declaration of function *sub*.
As shown below, no parameter needs to be specified if the default parameter value is acceptable when calling function *sub*.
Moreover, the efficiency of a program is not deteriorated even when it is converted into C.
In sum, a default parameter ensures superior development and maintenance efficiency and has no disadvantage compared with C.



```
(C++ program)

void main(void);
int sub(int, int = 0);

void main(void)
{
   int ret1;
   int ret2;
   ret = sub(1,2);
   ret = sub(3);
}

int sub (int a, int b /* =0 */ )
{
   return a + b;
}
```

Specifying 0 as a parameter value in the function declaration

No second parameter specified

```
(C program after conversion)

void main(void);
int sub(int, int);

void main(void)
{
   int ret1;
   int ret2;
   ret1 = sub(1, 2);
   ret2 = sub(3, 0);
}

int sub(int a, int b)
{
   return a + b;
}
```

Converted to the default parameter value

8.4.4 Inline Expansion

Development and Maintenance	○	Size Reduction	Δ	Speed	○

• Important Points:

When coding the definition of a function, specify *inline* in the beginning to cause inline expansion of the function. This will eliminate the overhead of a function call and improve the processing speed.

• Example of Use:

Specify function *sub* as an inline function and inline-expand it in the main function.Then, remove the function *sub* code. However, function *sub* cannot be reference from other files.
Use inline expansion with caution because, although the processing speed is certain to improve, the program size will become too large unless only small functions are used.

```
(C++ program)

int a;

inline int sub(int x, int y)
{
    return (x+y);
}

void main(void)
{
    a = sub(1,2);
}
```

```
(C program after conversion)

int a;
void main(void)
{
    a = 3;
    return;
}
```

Expanding the content of function *sub*

8.4.5 Class Member Function

Development and Maintenance	◎	Size Reduction	Δ	Speed	Δ

• Important Points:

Defining a class will enable information hidingand improve the development and maintenance efficiency.
However, use this technique with caution because it will influence the size and processing speed.

• Example of Use:

In the following example, class member functions *set* and *add* are used to access *private* class member variables *a*, *b*, and *c*.
When calling a class member function, the parameter specification in a C++ program either has only a value or no parameter.
As shown in the C program after conversion, however, the address of class A (struct A) is also passed as a parameter.
Additionally, *private* class member variables *a*, *b*, and *c* are accessed in the class member function code.
However, the *this* pointer is used to access them.
In sum, use a class member function with caution because it will influence the size and processing speed.

```
(C++ program)

class A
{
private:
    int a;
    int b;
    int c;
public:
    void set(int, int, int);
    int add();
};

int main(void)
{
    A a;
    int ret;

    a.set(1,2,3);
    ret = a.add();

    return ret;
}
void A::set(int x, int y, int z)
{
    a = x;
    b = y;
    c = z;
}
int A::add()
{
    return (a += b + c);
}
```

```
(C program after conversion)

struct A {
    int a;
    int b;
    int c;
};
void set__A_int_int(struct A *const, int, int, int);
int add__A(struct A *const);

int main(void)
{
    struct A a;
    int ret;

    set__A_int_int(&a, 1, 2, 3);
    ret = add__A(&a);

    return ret;
}
void set__A_int_int(struct A *const this, int x, int y, int z)
{
    this->a = x;
    this->b = y;
    this->c = z;
    return;
}
int add__A(struct A *const this)
{
    return (this->a += this->b + this->c);
}
```

8.4.6 operator Operator

Development and Maintenance	○	Size Reduction	Δ	Speed	Δ

• Important Points:

In C++, use the keyword, *operator* to overload an operator.
This will enable simple coding of the user's operations such as matrix operations and vector calculations.
However, use *operator* with caution because it will influence the size and processing speed.

• Example of Use:

In the following example, unary operator "+" is overloaded using the *operator* keyword.
If the *Vector* class is declared as shown below, unary operator "+" can be changed to the user's operation.
However, the size and processing speed will be influenced because, as shown in the C program after conversion, reference using the *this* pointer is made.

```
(C++ program)

class Vector
{
private:
    int x;
    int y;
    int z;
public:
    Vector & operator+ (Vector &);
};

void main(void)
{
    Vector a,b,c;

    a = b + c;
}

Vector & Vector::operator+ (Vector & vec)
{
    static Vector ret;

    ret.x = x + vec.x;
    ret.y = y + vec.y;
    ret.z = z + vec.z;

    return ret;
}
```

User's operation (addition)

```
(C program after conversion)

struct Vector {
    int x;
    int y;
    int z;
};

void main(void);
struct Vector *__plus__Vector_Vector(struct Vector *const, struct Vector *);

void main(void)
{
    struct Vector a;
    struct Vector b;
    struct Vector c;

    a = *__plus__Vector_Vector(&b, &c);
    return;
}

struct Vector *__plus__Vector_Vector( struct Vector *const this,  struct Vector
*vec)
{
    static struct Vector ret;

    ret.x = this->x + vec->x;
    ret.y = this->y + vec->y;
    ret.z = this->z + vec->z;

    return &ret;
}
```

Reference using the this

8.4.7 Overloading of Functions

Development and Maintenance	◎	Size Reduction	○	Speed	○

• Important Points:

In C++, you can "overload" functions, i.e., give the same name to different functions.
Specifically, this feature is effective when you use functions with the same processing but with different types of arguments.
Be careful not to give the same name to functions with no commonality because it is sure to cause malfunctions.
The use of this function will not influence the size or processing speed.

• Example of Use:

In the following example, the first and second parameters are added and the resultant value is used as a return value.
All the functions have the same name, *add* but different parameter and return value types.
As shown in the C program after conversion, the call of the add functions or the code of the add functions do not increase the code size.
Thus, the use of this feature will not influence the size and processing speed.

```
(C++ program)
void main(void);
int add(int,int);
float add(float,float);
double add(double,double);
void main(void)
{
    int    ret_i = add(1, 2);
    float  ret_f = add(1.0f, 2.0f);
    double ret_d = add(1.0, 2.0);
}

int add(int x,int y)
{
    return x+y;
}

float add(float x,float y)
{
    return x+y;
}

double add(double x,double y)
{
    return x+y;
}
```

```
(C program after conversion)

void main(void);
int add__int_int(int, int);
float add__float_float(float, float);
double add__double_double(double, double);

void main(void)
{
   auto int ret_i;
   auto float ret_f;
   auto double ret_d;

   ret_i = add__int_int(1, 2);
   ret_f = add__float_float(1.0f, 2.0f);
   ret_d = add__double_double(1.0, 2.0);
}

int add__int_int( int x,  int y)
{
   return x + y;
}

float add__float_float( float x,  float y)
{
   return x + y;
}

double add__double_double( double x,  double y)
{
   return x + y;
}
```

8.4.8 Reference Type

Development and Maintenance	◎	Size Reduction	○	Speed	○

- Important Points:

The use of a reference-type parameter will enable simple coding of a program and improve the development and maintenance efficiency.
Additionally, the use of the reference type will not influence the size or processing speed.

- Example of Use:

As shown below, reference-type passing instead of pointer passing will enable simple coding.
In a reference type, not the values but the addresses of *a* and *b* are passed.
The use of a reference type, as shown in the C program after conversion, will not influence the size and processing speed.

```
(C++ program)

void main(void);
void swap(int&, int&);

void main(void)
{
    int a=100;
    int b=256;

    swap(a,b);
}

void swap(int &x, int &y)
{
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

```
(C program after conversion)

void main(void);
void swap(int *, int *);

void main(void)
{
    int a=100;
    int b=256;

    swap(&a, &b);
}

void swap(int *x,  int *y)
{
    int tmp;
    tmp = *x;
    *x = *y;
    *y = tmp;
}
```

8.4.9 Static Function

Development and Maintenance	◎	Size Reduction	○	Speed	○

- Important Points:

If the class configuration becomes complex due to derived classes, etc., it will be increasingly more difficult to access *static* class member variables with the *private* attribute until they need to be changed to the *public* attribute.
To access a *static* class member variable without changing the *private* attribute in such a case, create a member function to be used as an interface and specify the *static* variable in the function.
A *static* function is thus used to access only static class member variables.

- Example of Use:

As shown on the next page, use a static function to access a static member variable.
Although the use of a class will influence the code efficiency, the use of a static function itself will not influence the size and processing speed.

- Note:

For details on a static member variable, refer to section 8.2.3, Static Member Variable.

```
(C++ program)

class A
{
private:
   static int num;
public:
   static int getNum(void);
   A(void);
   ~A(void);
};

int A::num = 0;
void main(void)
{
   int num;

   num = A::getNum();

   A a1;
   num = a1.getNum();

   A a2;
   num = a2.getNum();
}

A::A(void)
{
   ++num;
}

A::~A(void)
{
   --num;
}

int A::getNum(void)
{
   return num;
}
```

Static member variable

Static function

Accessing the static member variable

```
(C program after conversion)
struct A
{
    char __dummy;
};
void *__nw__FUl(unsigned long);
void __dl__FPv(void *);
int getNum__A(void);          <----------   Static member variable
struct A *__ct__A(struct A *);
void __dt__A(struct A *const, int);
int num__1A = 0;              <----------   Static function
void main(void)
{
    int num;
    struct A a1;
    struct A a2;

    num = getNum__A();

    __ct__A(&a1);
    num = getNum__A();

    __ct__A(&a2);
    num = getNum__A();

    __dt__A(&a2, 2);
    __dt__A(&a1, 2);
}
int getNum__A(void)
{
    return num__1A;           <----------   Accessing the static member variable
}
struct A *__ct__A( struct A *this)
{
    if ( (this != (struct A *)0)
    || ( (this = (struct A *)__nw__FUl(1)) != (struct A *)0) ){
        ++num__1A;
    }
    return this;
}
void __dt__A( struct A *const this,  int flag)
{
    if (this != (struct A *)0){
        --num__1A;
        if(flag & 1){
            __dl__FPv((void *)this);
        }
    }
    return;
}
```

8.4.10 Static Member Variable

Development and Maintenance	◎	Size Reduction	○	Speed	○

- Important Points:

In C++, a class member variable with the static attribute can be shared among multiple objects of a class type.
Thus, a static member variable comes in handy because it can be used, for example, as a common flag among multiple objects of the same class type.

- Example of Use:

Create five class-A objects within the *main* function.
Static member variable *num* has an initial value of 0. This value will be incremented by the constructor every time an object is created.
Static member variable num, shared among objects, will have a value of 5 at the maximum.
Additionally, the use of a class will influence the code efficiency.
However, the use of a static member variable itself will not influence the size and processing speed because the Compiler internally handles member variable *num* as if it is an ordinary global variable.

- Note:

For details on a static member variable, refer to section 8.2.3, Static Member Variable.

```
(C++ program)

class A
{
private:
   static int num;
public:
   A(void);
   ~A(void);
};

int A::num = 0;

void main(void)
{
   A a1;
   A a2;
   A a3;
   A a4;
   A a5;
}

A::A(void)
{
   ++num;
}

A::~A(void)
{
   --num;
}
```

Creating a class A-type class object

Incrementing a *static* member variable

```
(C program after conversion)

struct A
{
    char __dummy;
};
void *__nw__FUl(unsigned long);
void __dl__FPv(void *);
struct A *__ct__A(struct A *);
void __dt__A(struct A *const, int);
int num__1A = 0;
void main(void)
{
    struct A a1;
    struct A a2;
    struct A a3;
    struct A a4;
    struct A a5;
    __ct__A(&a1);
    __ct__A(&a2);
    __ct__A(&a3);
    __ct__A(&a4);
    __ct__A(&a5);
    __dt__A(&a5, 2);
    __dt__A(&a4, 2);
    __dt__A(&a3, 2);
    __dt__A(&a2, 2);
    __dt__A(&a1, 2);
}
struct A *__ct__A( struct A *this)
{
    if( (this != (struct A *)0)
    || ( (this = (struct A *)__nw__FUl(1)) != (struct A *)0) ){
        ++num__1A;
    }
    return this;
}
void __dt__A( struct A *const this,  int flag)
{
    if(this != (struct A *)0){
        --num__1A;
        if (flag & 1){
            __dl__FPv((void *)this);
        }
    }
    return;
}
```

Handled by the Compiler as if it is an ordinary global variable

Creating class A-type class objects

Calling constructors

Calling destructors

Incrementing a static member variable

8.4.11 Anonymous Union

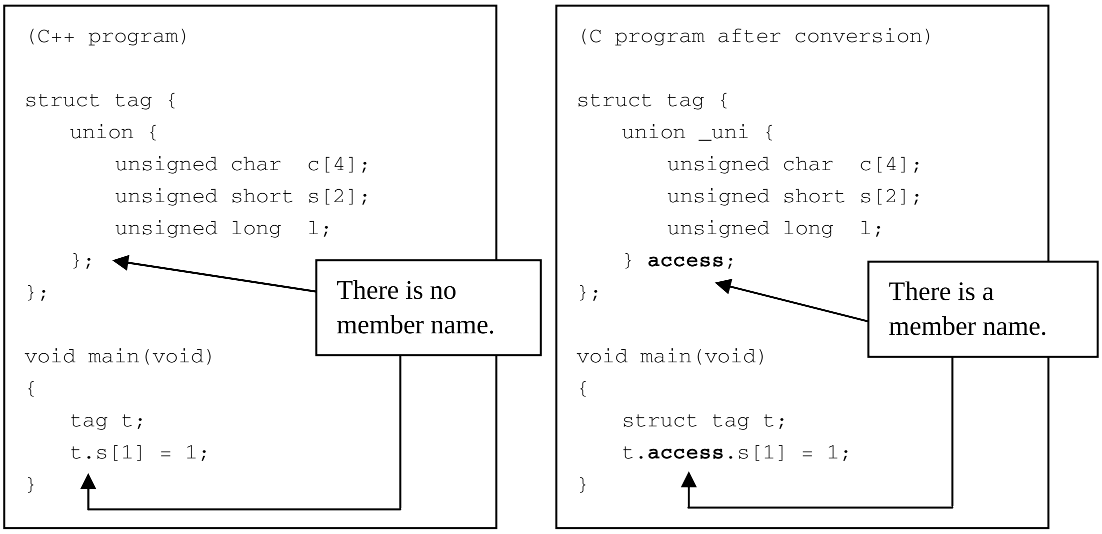
Development and Maintenance	◎	Size Reduction	○	Speed	○

- Important Points:

In C++, use an anonymous *union* to directly access a member without, like in C, having to specify the member name. This will improve the development efficiency. Additionally, it will not influence the size and processing speed.

- Example of Use:

In the following example, function main is used to access *union* member variable *s*.
In the C++ program, member variable *s* is directly accessed. In the C program after conversion, it is accessed using a member name that the Compiler has automatically created.
The use of this simple code enables access to a member variable without influencing the object efficiency.



```
(C++ program)

struct tag {
    union {
        unsigned char  c[4];
        unsigned short s[2];
        unsigned long  l;
    };
};

void main(void)
{
    tag t;
    t.s[1] = 1;
}
```

There is no member name.

```
(C program after conversion)

struct tag {
    union _uni {
        unsigned char  c[4];
        unsigned short s[2];
        unsigned long  l;
    } access;
};

void main(void)
{
    struct tag t;
    t.access.s[1] = 1;
}
```

There is a member name.

8.4.12 Virtual Function

Development and Maintenance	◎	Size Reduction	Δ	Speed	Δ

- Important Points:

A virtual function must be used if, as shown in the following program, there is a function with the same name in each of a base class and a derived class. Otherwise, the function call cannot be properly made as intended.
If a virtual function is declared, these calls can be properly made as intended.
Use a virtual function to improve the development efficiency. However, use it with caution because it will influence the size and processing speed.

- Example of Use:

In the *main3* function call, two pointers store class-B addresses.
If *virtual* is declared, the class-B *foo* function is properly called.
If *virtual* is not declared, one of the pointers calls the class-A *foo* function.
The use of a virtual function, resulting in creation of a table, etc. as shown on the next page, will influence the size and speed.

```
(C++ program)

class A
{
private:
    int a;
public:
    virtual void foo(void);
};

class B : public A
{
private:
    int b;
public:
    virtual void foo(void);
};

void A::foo(void)
{
}

void B::foo(void)
{
}
```

Virtual function declaration

Virtual function declaration (Can be omitted)

```
void main1(void)
{
    A a;
    a.foo();
}

void main2(void)
{
    B b;
    b.foo();
}

void main3(void)
{
    B b;
    A * pa = &b;
    B * pb = &b;

    pa->foo();
    pb->foo();
}
```

Virtual function call

- C program after conversion (tables, etc. for virtual functions):

```
struct __T5585724;
struct __type_info;
struct __T5584740;
struct __T5579436;
struct A;
struct B;
extern void main1__Fv(void);
extern void main2__Fv(void);
extern void main3__Fv(void);
extern void foo__1AFv(struct A *const);
extern void foo__1BFv(struct B *const);
struct __T5585724
{
    struct __T5584740 *tinfo;
    long offset;
    unsigned char flags;
};
struct __type_info
{
    struct __T5579436 *__vptr;
};
struct __T5584740
{
    struct __type_info tinfo;
    const char *name;
    char *id;
    struct __T5585724 *bc;
};
struct __T5579436
{
    long d;                           // this-pointer offset
    long i;                           // Unassigned
    void (*f)();                      // For virtual function call
};
struct A {                            // Class-A declaration
    int a;
    struct __T5579436 *__vptr;        // Pointer to a virtual function table
};
struct B {                            // Class-B declaration
    struct A __b_A;
    int b;
};
static struct __T5585724 __T5591360[1];
#pragma section $VTBL
extern const struct __T5579436 __vtbl__1A[2];
extern const struct __T5579436 __vtbl__1B[2];
extern const struct __T5579436 __vtbl__Q2_3std9type_info[];
#pragma section
extern struct __T5584740 __T_1A;
extern struct __T5584740 __T_1B;
```

```
static char __TID_1A;                      // Unassigned
static char __TID_1B;                      // Unassigned
static struct __T5585724 __T5591360[1] =   // Unassigned
{
    {
        &__T_1A,
        0L,
        ((unsigned char)22U)
    }
};
#pragma section $VTBL
const struct __T5579436 __vtbl__1A[2] =     // Virtual function table for class-A
{
    {
        0L,                                 // Unassigned area
        0L,                                 // Unassigned area
        ((void (*)())&__T_1A)               // Unassigned area
    },
    {
        0L,                                 // this-pointer offset
        0L,                                 // Unassigned area
        ((void (*)())foo__1AFv)             // ((void (*)())foo__1AFv) // Pointer to A::foo()
    }
};
const struct __T5579436 __vtbl__1B[2] =     // Virtual function table for class-B
{
    {
        0L,                                 // Unassigned area
        0L,                                 // Unassigned area
        ((void (*)())&__T_1B)               // Unassigned area
    },
    {
        0L,                                 // this-pointer offset
        0L,                                 // Unassigned area
        ((void (*)())foo__1BFv)             // ((void (*)())foo__1BFv) // Pointer to B::foo()
    }
};
#pragma section
struct __T5584740 __T_1A =                  // Type information for class-A (unassigned)
{
    {
        (struct __T5579436 *)__vtbl__Q2_3std9type_info
    },
    (const char *)"A",
    &__TID_1A,
    (struct __T5585724 *)0
};
struct __T5584740 __T_1B =                  // Type information for class-B (unassigned)
{
    {
        (struct __T5579436 *)__vtbl__Q2_3std9type_info
    },
    (const char *)"B",
    &__TID_1B,
    __T5591360
};
```

- C program after conversion (virtual function calls):

```
void main1__Fv(void)
{
    struct A _a;
    _a.__vptr = __vtbl__1A;
    foo__1AFv( &_a );                                                    // Call A::foo()
    return;
}
void main2__Fv(void)
{
    struct B _b;
    _b.__b_A.__vptr = __vtbl__1A;
    _b.__b_A.__vptr = __vtbl__1B;
    foo__1BFv( &_b );                                                    // Call B::foo()
    return;
}
void main3__Fv(void)
{
    struct __T5579436 *_tmp;
    struct B _b;
    struct A *__pa;
    struct B *__pb;

    (*((struct A*)(&_b))).__vptr = __vtbl__1A;
    (*((struct A*)(&_b)))._vptr = __vtbl__lB;
    _pa = (struct A *)&_b;
    _pb = &_b;

    _tmp = _pa->__vptr + 1;
    (  (void (*)(struct A *const))  _tmp->f  )  (  (struct A *)_pa + _tmp->d );
    // Call to B::foo() (The object what is pointed by _pa  is B)

    _tmp = _pb->__b_A.__vptr + 1;
    (  (void (*)(struct B *const))  _tmp->f  )  (  (struct B *)_pb + _tmp->d );
    // Call to B::foo()

    return;
}
```

Section 9 Optimizing Linkage Editor

This chapter describes the use of effective options at linkage, and the Inter-Module Optimization at linkage.

The following table shows a list of the items relating to the use of Optimizing Linkage Editor.

No.	Category	Item	Section
1	Input/Output Options	Input Options	9.1.1
		Output Options	9.1.2
2	List Options	Symbol Information List	9.2.1
3		Symbol Reference Count	9.2.2
4		Cross-Reference Information	9.2.3
5	Effective Options	Output to Unused Area	9.3.1
6		End code of S Type File	9.3.2
7		Debug Information Compression	9.3.3
8		Link Time Reduction	9.3.4
9		Notification of Unreferenced Symbol	9.3.5
10		Reduce Empty Areas of Boundary Alignment	9.3.6
11	Optimize Options	Optimization at Linkage	9.4.1
12		Sub Options of Optimize Option	
13		Unifies Constants/Strings	9.4.2
14		Eliminates Unreferenced Symbols	9.4.3
15		Optimizes Register Save/Restore Codes	9.4.4
16		Unifies Common Codes	9.4.5
17		Optimizes Branch Instructions	9.4.6
18		Optimization Partially Disabled	9.4.7
19		Confirm Optimization Results	9.4.8

9.1 Input/Output Options

9.1.1 Input Options

• Description

The optimizing linkage editor can input the following four files according to user usage.
This is one of the convenient features.

• Specification Method

Dialog menu: **Link/Library Tab Category: [Input] Show entries for :**
Command line: *Input <suboption>:<file name>*
Library<file name>
Binary<suboption>:<file name>

• Available Input Files

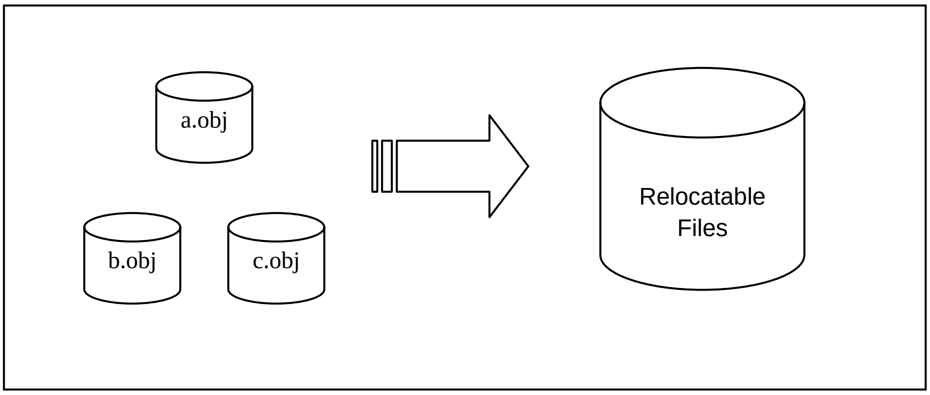
Kind of Files	Command line
Object Files	input
Relocatable Files	input
Library Files	library
Binary Files	binary

(1) Object Files

Ordinary files output from the compiler or the assembler.

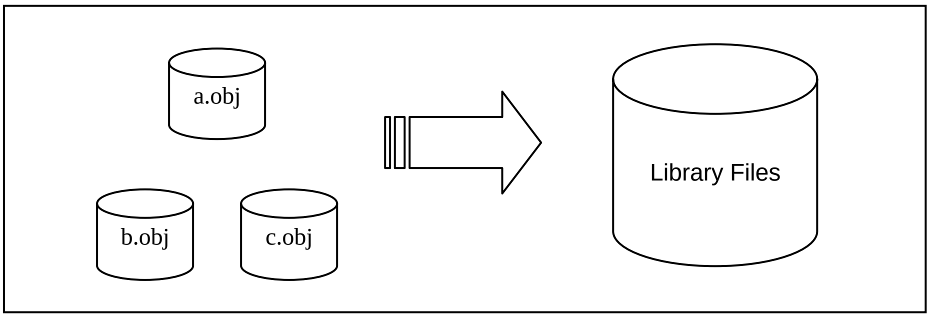
(2) Relocatable Files

Relocatable (Address Unresolved) Files.
This file consists of one or more object files, and is generated from the optimizing linkage editor with output options.
Symbols in relocatable files **are linked**, even if other files **don't refer** to them.
So, in case of using Relocatable Files, be careful not to increase ROM size by linking unnecessary files.



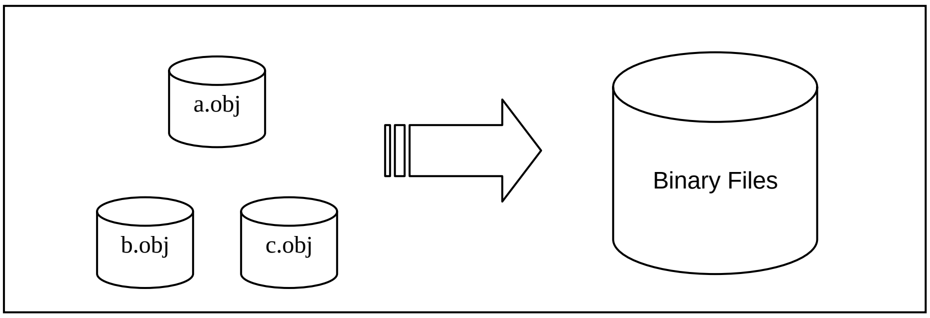
(3) Library Files

Relocatable (Address Unresolved) Files.
This file consists of one or more object files, and is generated from the optimizing linkage editor with output options.
Symbols in relocatable files **are not linked**, if other files **don't refer** to them.



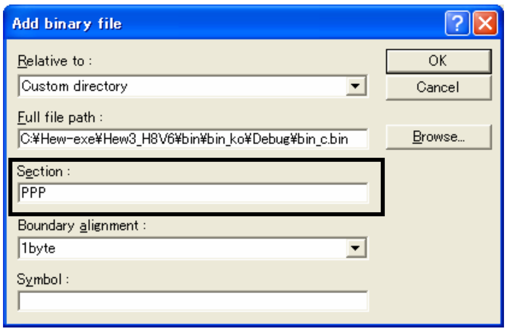
(4) Binary Files

Binary Files are available to input.
This file consists of one or more object files, and is generated from the optimizing linkage editor with output options.
When input binary files, section name should be specified. This section name is located with the **start** option.
As binary files have no debug information, C/C++ source level debugger can't be used.



[Specification Method 1]

Section name should be specified.
Dialog menu: **Link/Library Tab Category: [Input] Show entries for :**
Binary files
Command line: ***binary=bin_c.bin(PPP)***

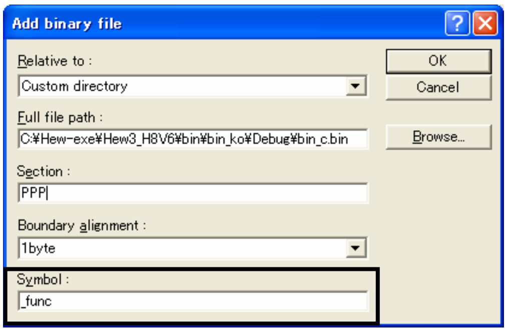


Relative to : Custom directory
Full file path : C:¥Hew-exe¥Hew3_H8V6¥bin¥bin_ko¥Debug¥bin_c.bin
Section : PPP
Boundary alignment : 1byte
Symbol :

[Specification Method 2]

Symbol can be defined at the head of the binary files.
Specify symbol name with section name, to do this.
For a variable name referred by a C/C++ program, add an underscore (_) at the head of the symbol name.

Dialog menu: **Link/Library Tab Category: [Input] Show entries for :**
Binary files
Command line: ***binary=bin_c.bin(PPP,_func)***


Add binary file

Relative to : Custom directory

Full file path : C:¥Hew-exe¥Hew3_H8V6¥bin¥bin_ko¥Debug¥bin_c.bin

Section : PPP

Boundary alignment : 1byte

Symbol : _func

[Specification Method 3]

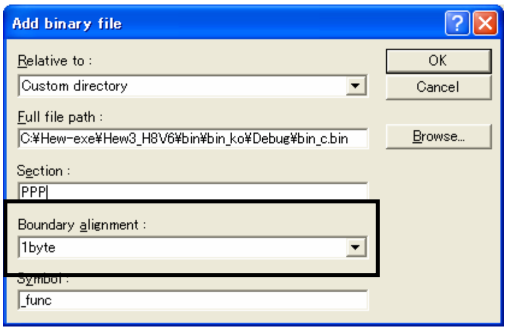
When input binary files, boundary alignment value can be specified.

When the boundary alignment specification is omitted, 1 is used as the default for the compatibility with earlier versions. This boundary alignment specification is valid in the Optimizing Linkage Editor Ver.9.0 or later.

Dialog menu: **Link/Library Tab Category: [Input] Show entries for : Binary files**

Command line: ***binary=bin_c.bin(PPP:<boundary alignment>,_func)***

<boundary alignment>: 1 | 2 | 4 | 8 | 16 | 32 (default: 1)


Add binary file

Relative to : Custom directory

Full file path : C:¥Hew-exe¥Hew3_H8V6¥bin¥bin_ko¥Debug¥bin_c.bin

Section : PPP

Boundary alignment : 1byte

Symbol : _func

9.1.2 Output Options

• Description

Some type of ROM writer can input only HEX files or only S-type files.
The optimizing linkage editor can output the following eight files according to user usage.
User can change the kind of output file, if necessary.

• Specification Method

Dialog menu: **Link/Library Tab Category: [Output] Type of output file :**
Command line: *form{ absolute* | *relocate* | *object* | *library=s* | *library=u* |
hexadecimal | *stype* | *binary }*

• Available Output Files

No.	Kind of Files	Command line
1	Absolute Files	form absolute
2	Relocatable Files	form relocate
3	Object Files	form object
4	User Library Files	form library=s
5	System Library Files	form library=u
6	HEX Files	form hexadecimal
7	S-type Files	form stype
8	Binary Files	form binary

(1) Absolute Files

Address resolved Files by the optimizing linkage editor.
As this file has debug information, C/C++ source level debugger can be used.
When writing to ROM, this file should be transformed to either S-type format or HEX or Binary.

(2) Relocatable Files

Relocatable (Address Unresolved) Files.
As this file has debug information, C/C++ source level debugger can be used.
To execute this file, this file should be transformed to absolute file by linking again.

(3) Object Files

This file is used when a module (object) is extracted as an object file from a library with the extract option.

When specifying by command line, a needed object file can be extracted from the library file specified by this option.
When using HEW, specify the following options at **Link/Library Tab Category: [Other] User defined options :**
[Extract Options]
form=object
extract=<module name>

(4) User Library/System Library

Output Library Files.

(5) HEX Files

Output HEX Files.
As this files have no debug information, C/C++ source level debugger can’t be used.
For details of HEX file, please refer to “SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User’s Manual” 18.1.2 HEX File Format.

(6) S-type Files

Output S-type Files.
As this files have no debug information, C/C++ source level debugger can’t be used.
For details of S-type file, please refer to “SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User’s Manual” 18.1.1 S-Type File Format.

(7) Binary Files

Output Binary Files.
As binary files have no debug information, C/C++ source level debugger can’t be used.

9.2 List Options

9.2.1 Symbol Information List

• Description

The optimizing linkage editor can output symbol address, size and optimization information in addition to linkage map information, by specifying additional sub-options.

— symbol address -**ADDR**

— size -**SIZE**

— optimization -**OPT** (**ch**- changed, **cr**- created, **mv**- moved)

• Specification Method

Dialog menu: **Link/Library Tab Category: [List] Contents : Show symbol**
Command line: *list [=<file name>]*
Show symbol

```
<*.map file>

   *** Options ***
           :
   *** Error information ***
          :
   *** Mapping List ***
         :
   *** Symbol List ***

   SECTION=
   FILE=                    START      END    SIZE
     SYMBOL                 (1)ADDR    (2)SIZE  INFO  COUNTS  (3)OPT

   SECTION=P
   FILE=C:\Hew-exe\Hew3_SHV9\bin\bin\Debug\bin.obj

                            00000800  00000821      22

        _main
                                 00000800      6        func ,g         *   ch
        _abort
                                 00000806      4        func ,g         *   ch
        _com_opt1
                                 0000080a      18       func ,g         *   cr ch

   *** Delete Symbols ***
            :
   *** Variable Accessible with Abs8 ***
            :
   *** Variable Accessible with Abs16 ***
            :
   *** Function Call ***
        :
```

9.2.2 Symbol Reference Count

• Description

The optimizing linkage editor can output static symbol reference count in addition to linkage map information, by specifying additional sub-options.

— symbol reference count -COUNTS

• Specification Method

Dialog menu: **Link/Library Tab Category: [List] Contents : Show reference**
Command line: *list [=<file name>]*
Show reference

```
<*.map>
*** Options ***
         :
*** Error information ***
         :
*** Mapping List ***
         :
*** Symbol List ***

SECTION=
FILE=                    START       END    SIZE
  SYMBOL                 ADDR     SIZE  INFO       (1)COUNTS  OPT

SECTION=P
FILE=C:\Hew-exe\Hew3_SHV9\bin\bin\Debug\bin.obj

                          00000800         00000821       22
    _main
                          00000800         6              func ,g      1    ch
    _abort
                          00000806         4              func ,g      0    ch
    _com_opt1
                          0000080a         18             func ,g      2    cr ch
*** Delete Symbols ***
         :
*** Variable Accessible with Abs8 ***
         :
*** Variable Accessible with Abs16 ***
         :
*** Function Call ***
```

9.2.3 Cross-Reference Information

• Description

The optimizing linkage editor can output cross-reference information in addition to linkage map information, by specifying additional sub-options. Cross-reference information makes it possible to search where a global symbol is referenced.
Local symbols and static symbols are not output.

- **Specification Method**

Dialog menu: **Link/Library Tab Category: [List] Contents : Show cross reference**

Command line: *list [=<file name>]*
Show xreference

```
<*.map file>
*** Cross Reference List ***

No   Unit Name   Global.Symbol   Location  External Information
(1)      (2)           (3)          (4)            (5)
---- ----------- --------------- -------- ---------------------
0001 test1
     SECTION=P
                _main
                                00000100
     SECTION=B
                _sl1
                                00007000 0001(0000011a:P)
                _sl2
                                00007004 0001(0000010e:P)
                _ret
                                00007008 0001(00000128:P)
     SECTION=D
0002 test2
     SECTION=P
                _func1
                                0000015c 0001(00000124:P)
                _func2
                                00000164 0001(0000013c:P)
                _func3
                                00000170 0001(00000150:P)
```

- **Description of Each Item**

(1) Unit number, which is an identification number in object units, displayed in External Information (5)
(2) Object name, which specifies the input order at linkage
(3) Symbol name output in ascending order for every section
(4) Symbol allocation address, which is a relative value from the beginning of the section when relocatable format is specified for output file format (form=relocate).
(5) Address from which an external symbol is referenced
Output format: <Unit number> (<address or offset in section>:<section name>)

- **Remarks**

This option is valid for the Optimizing Linkage Editor Ver.9.0 or later.

9.3 Effective Options

9.3.1 Output to Unused Area

• **Description**

The optimizing linkage editor can output any data to unused area.
This is useful for ROM transfer, and this is useful to detect an abnormal interrupt by executing unused area with no data, when program hangs.
A 1-, 2-, or 4-byte value is valid for output data size. If an odd number of digits are specified, the upper digits are extended with 0 to use it as an even number of digits.
The maximum size of output data is 4-byte. If a value over 4-byte is specified, the lower 4-byte is used.
This option is available only when output file is S-type file, Binary or HEX.

• **Specification Method**

Dialog menu: **Link/Library Tab Category: [Output] Show entries for :**
Specify value filled in unused area
Command line: *space [=<numerical value>]*

• **Examples**

(1) Divide file and specify the range to fill unused area with data by
Link/Library Tab Category: [Output] Show entries for : Divided output files
-output="C:\bin\Debug\a.bin"=00-0FFFF

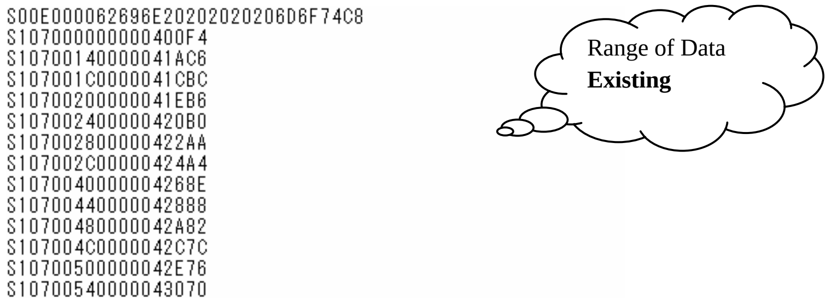
(2) Specify the filling data by
Link/Library Tab Category: [Output] Show entries for : Specify value filled in unused area
-space=FF

The example of the following page <Specify value filled in unused area [H’FF]> shows how unused area is filled with data.

- **Examples of S-type Files**

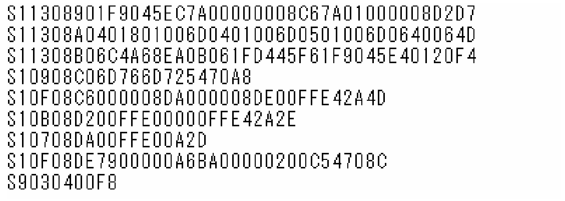
As the following examples, 0xFF records are added to the unused areas in the range of data existing.
If this option is not specified, the records in the range of data not existing are not output.
If this option is specified, 0xFF records are added to the area in the range of data not existing, according to the output range specification in the output option **Divided output files**.

<NOT Specify value filled in unused area>



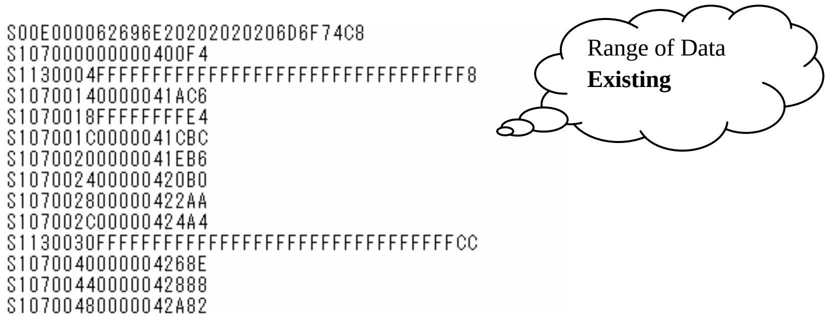
```
S00E000062696E20202020206D6F74C8
S107000000000400F4
S10700140000041AC6
S107001C0000041CBC
S10700200000041EB6
S107002400000420B0
S107002800000422AA
S107002C00000424A4
S10700400000004268E
S107004400000042888
S10700480000042A82
S107004C0000042C7C
S10700500000042E76
S107005400000043070
```

…



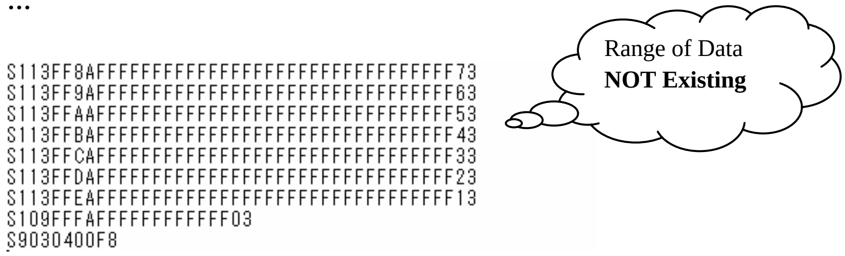
```
S11308901F9045EC7A00000008C67A01000008D2D7
S11308A0401801006D0401006D0501006D06400640
S11308B06C4A68EA0B061FD445F61F9045E40120F4
S10908C06D766D725470A8
S10F08C6000008DA000008DE00FFE42A4D
S10B08D200FFE00000FFE42A2E
S10708DA00FFE00A2D
S10F08DE7900000A6BA00000200C54708C
S9030400F8
```

<Specify value filled in unused area [H’FF]>



```
S00E000062696E20202020206D6F74C8
S107000000000400F4
S1130004FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF8
S10700140000041AC6
S1070018FFFFFFFFE4
S107001C0000041CBC
S10700200000041EB6
S107002400000420B0
S107002800000422AA
S107002C00000424A4
S1130030FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFCC
S10700400000004268E
S107004400000042888
S10700480000042A82
```

…

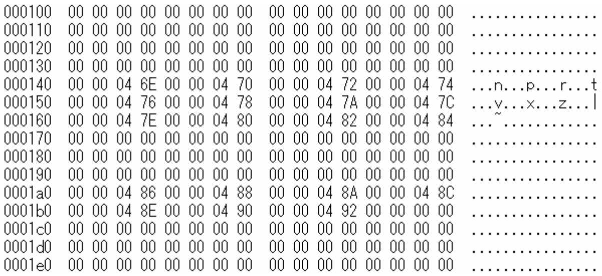


```
S113FF8AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF73
S113FF9AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF63
S113FFAAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF53
S113FFBAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF43
S113FFCAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF33
S113FFDAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF23
S113FFEAFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF13
S109FFFAFFFFFFFFFFFF03
S9030400F8
```

- **Examples of Binary Files**

As the following examples, the unused areas in the range of data existing are changed from 0x00 to 0xFF.
If this option is not specified, the records in the range of data not existing are not output.
If this option is specified, 0xFF records are added to the area in the range of data not existing, according to the output range specification in the output option **Divided output files**.

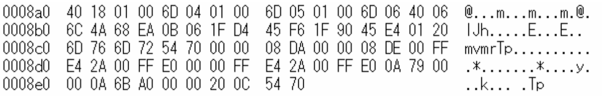
<NOT Specify value filled in unused area>



```
000100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000120  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000130  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000140  00 00 04 6E 00 00 04 70  00 00 04 72 00 00 04 74  ...n...p...r...t
000150  00 00 04 76 00 00 04 78  00 00 04 7A 00 00 04 7C  ...v...x...z...|
000160  00 00 04 7E 00 00 04 80  00 00 04 82 00 00 04 84  ...~............
000170  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
000190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0001a0  00 00 04 86 00 00 04 88  00 00 04 8A 00 00 04 8C  ................
0001b0  00 00 04 8E 00 00 04 90  00 00 04 92 00 00 00 00  ................
0001c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0001d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0001e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

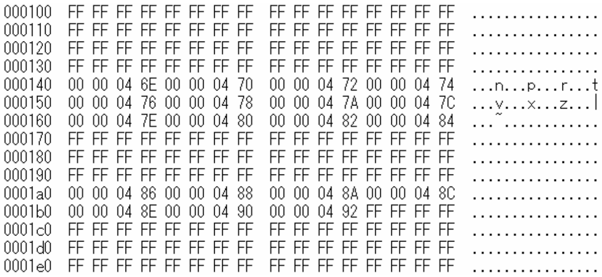
Range of Data
Existing

…



```
0008a0  40 18 01 00 6D 04 01 00  6D 05 01 00 6D 06 40 06  @...m...m...m.@.
0008b0  6C 4A 68 EA 0B 06 1F D4  45 F6 1F 90 45 E4 01 20  lJh.....E...E..
0008c0  6D 76 6D 72 54 70 00 00  08 DA 00 00 08 DE 00 FF  mvmrTp..........
0008d0  E4 2A 00 FF E0 00 00 FF  E4 2A 00 FF E0 0A 79 00  .*.......*....y.
0008e0  00 0A 6B A0 00 00 20 0C  54 70                    ..k... .Tp
```

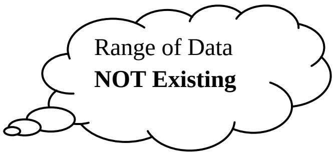
<Specify value filled in unused area [H’FF]>

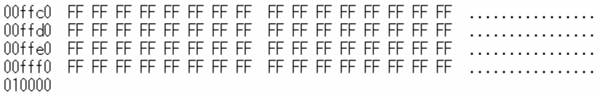


```
000100  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000110  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000120  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000130  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000140  00 00 04 6E 00 00 04 70  00 00 04 72 00 00 04 74  ...n...p...r...t
000150  00 00 04 76 00 00 04 78  00 00 04 7A 00 00 04 7C  ...v...x...z...|
000160  00 00 04 7E 00 00 04 80  00 00 04 82 00 00 04 84  ...~............
000170  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000180  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
000190  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
0001a0  00 00 04 86 00 00 04 88  00 00 04 8A 00 00 04 8C  ................
0001b0  00 00 04 8E 00 00 04 90  00 00 04 92 FF FF FF FF  ................
0001c0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
0001d0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
0001e0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
```

Range of Data
Existing

…



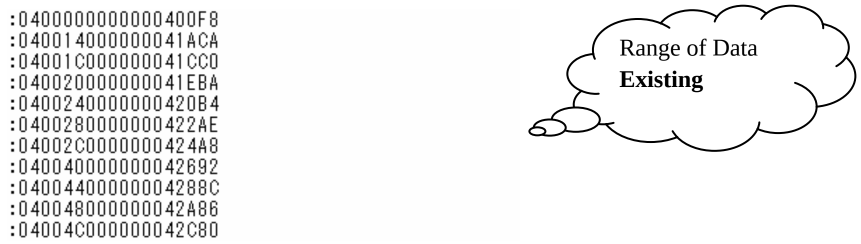


```
00ffc0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00ffd0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00ffe0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
00fff0  FF FF FF FF FF FF FF FF  FF FF FF FF FF FF FF FF  ................
010000
```

- **Examples of HEX Files**

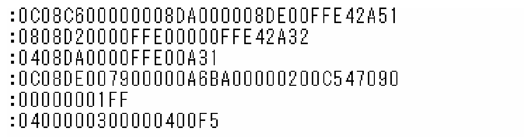
As the following examples, 0xFF records are added to the unused areas in the range of data existing.
If this option is not specified, the records in the range of data not existing are not output.
If this option is specified, 0xFF records are added to the area in the range of data not existing, according to the output range specification in the output option **Divided output files**.

<NOT Specify value filled in unused area>



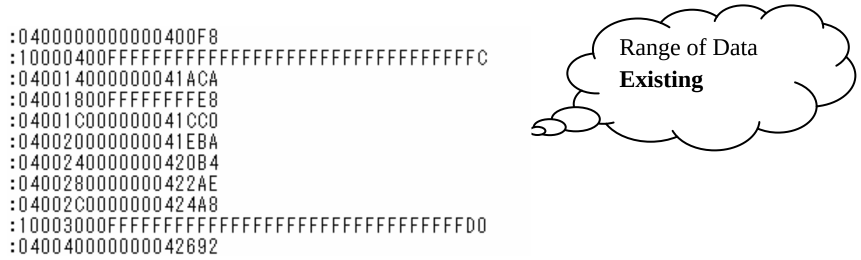
```
:0400000000000400F8
:04001400000041ACA
:04001C000000041CC0
:04002000000041EBA
:0400240000000420B4
:0400280000000422AE
:04002C00000000424A8
:04004000000000042692
:0400440000000004288C
:0400480000000042A86
:04004C000000042C80
```

…



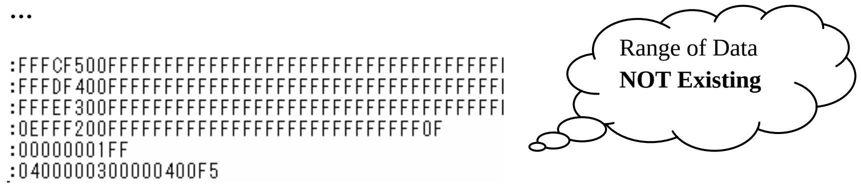
```
:0C08C600000008DA000008DE00FFE42A51
:0808D20000FFE00000FFE42A32
:0408DA0000FFE00A31
:0C08DE00790000A6BA0000200C547090
:00000001FF
:0400000300000400F5
```

<Specify value filled in unused area [H’FF]>



```
:0400000000000400F8
:10000400FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC
:04001400000041ACA
:04001800FFFFFFFFE8
:04001C000000041CC0
:04002000000041EBA
:0400240000000420B4
:0400280000000422AE
:04002C00000000424A8
:10003000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD0
:04004000000000042692
```

…



```
:FFFCF500FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFI
:FFFDF400FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFI
:FFFEF300FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFI
:0EFFF200FFFFFFFFFFFFFFFFFFFFFFFFFFFF0F
:00000001FF
:0400000300000400F5
```

- **Remarks**

This option is valid for the optimizing linkage editor Ver.8 or later.

9.3.2 End code of S-Type File

- **Description**

In some type of ROM writer, run time error may occur during input to ROM writer, when the end code of S-type file is not s9 record.
This is because end code is s7 or s8, if the entry address exceeds 0x10000.
By specifying this option, the end code can be always s9.

- **Specification Method**

Dialog menu: **Link/Library Tab Category: [Other] Miscellaneous options :**
Always output S9 record at the end
Command line: *S9*

- **Remarks**

For details of S-type file, please refer to section 18.1.1, S-Type File Format in the SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual.

9.3.3 Debug Information Compression

- **Description**

By specifying this option, the loading time is reduced when loading files to debugger.
But on the contrary, the link time is increased.

- **Specification Method**

Dialog menu: **Link/Library Tab Category: [Other] Miscellaneous options :**
Compress debug information
Command line: *compress*
uncompress

- **Remarks**

This option is valid only when output file is absolute file.

9.3.4 Link Time Reduction

- **Description**

When this option is specified, the linkage editor loads the necessary information at linkage in smaller units to reduce the memory occupancy.
As a result, the link time may be reduced.
Try this option when processing is slow because a large project is linked and the memory size occupied by the linkage editor exceeds the available memory in the machine used.

- **Specification Method**

Dialog menu: **Link/Library Tab Category: [Other] Miscellaneous options :**
Low memory use during linkage
Command line: *memory={high | low}*

- **Examples**

The following example is the comparison of the link time when this option is specified or not.
At the following case, the link time is reduced by 34 %.

<Measurement Conditions>
- 1,000 files
- 100 symbols per each file
- 1,000 function symbols
- Specifies the same options, except this option

<memory=high>
111 seconds

<memory=low>
73 seconds

- **Remarks**

This option is valid for the optimizing linkage editor Ver.8 or later.

9.3.5 Notification of Unreferenced Symbol

- **Description**

When project is large, it is difficult to find the externally defined symbol which is defined but not referenced.

When this option is specified, the external symbol which is not referenced can be notified through an output message at linkage.

To output a notification message, the message option* must also be specified.

Note : * **Link/Library Tab Category: [Output] [Show entries for :] [Output messages] Repressed information level messages :**

- **Specification Method**

Dialog menu: **Link/Library Tab Category: [Output] [Show entries for :] [Output messages] Notify unused symbol**

Command line: *msg_unused*

- **Output Message**

L0400 (I) Unused symbol “file”-“symbol”

The symbol named **symbol** in **file** is not used.

- **Remarks**

(1) This option is valid for the optimizing linkage editor Ver.9 or later.

(2) In any of the following cases, references are not correctly analyzed so that information shown by output messages will be incorrect.

- **–goptimize** is not specified at assembly and there are branches to the same section within the same file.
- There are references to constant symbols within the same file.
- There are branches to immediate subordinate functions when optimization is specified at compilation.
- Optimization is specified at linkage and constants are unified.

9.3.6 Reduce Empty Areas of Boundary Alignment

• Description

When this option is specified, the empty areas, which are generated as the boundary alignment of sections for each object file, are filled at linkage.

As a result, the unnecessary empty areas generated by boundary alignment are filled, reducing the size of the data sections as a whole.

This option affects constant area (C section), initialized data area (D section) and uninitialized data area (B section).

• Specification Method

Dialog menu: **Link/Library Tab Category: [Output] [Show entries for :]**

Reduce empty areas of boundary alignment

Command line: *data_stuff*

• Examples

The following example shows how empty areas of boundary alignment are reduced.

```
(file1.c)
short s1;
char c1;
```

```
(file2.c)
char c2;
```

<When **data_stuff** is not specified>

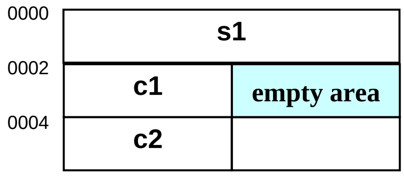
When **data_stuff** is not specified, one byte empty area of boundary alignment is generated between **file1.c** and **file2.c**, because boundary alignment value is 4 for SH CPU specification.

In this example, if the size of the top data which is linked next is one byte, there is no need of this boundary alignment.

But the top data of the next file is 2 bytes or more, boundary alignment at the end of this file (**file1.c**) should be performed.

As a result, data alignment and data size are

s1(2 bytes) + c1(1 byte) + empty area(1 byte) + c2(1 byte) = 5 bytes



<When **data_stuff** is specified>

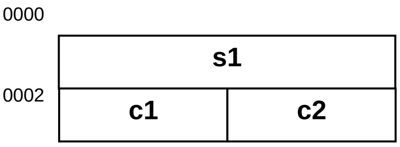
When **data_stuff** is specified, empty area of boundary alignment is not generated, if the size of the top data which is linked next is one byte as this example.

As a result, data alignment and data size are

s1(2 bytes) + c1(1 byte) + c2(1 byte) = 4 bytes

Here, the data size is reduced to 4 bytes.

As this program example, empty areas generated as the boundary alignment of sections are filled at linkage. However, the order of data allocation is not changed.



• Remarks

(1) This option is valid for the optimizing linkage editor Ver.8.00.06 or later.
(2) The function of this option is not applicable to object files generated by the assembler.
(3) Specification of this option is invalid in any of the following cases:
- **library** or **object** is specified as output format of the optimizing linkage editor
- **absolute** is specified as input format of the optimizing linkage editor
- **memory=low** is specified
- optimization at linkage (**optimize**) is specified

(4) Optimization will not be applied in the linkage of a relocatable file that was generated with this option specified.

9.4 Optimize Options

9.4.1 Optimization at Linkage

• Description

Compiler outputs the supplement information to each module when generating object files.
According to this supplement information, the optimizing linkage editor performs the inter-module optimization which is impossible at compile and links.
As a result, both ROM size and execution speed are improved.

• Specification Method

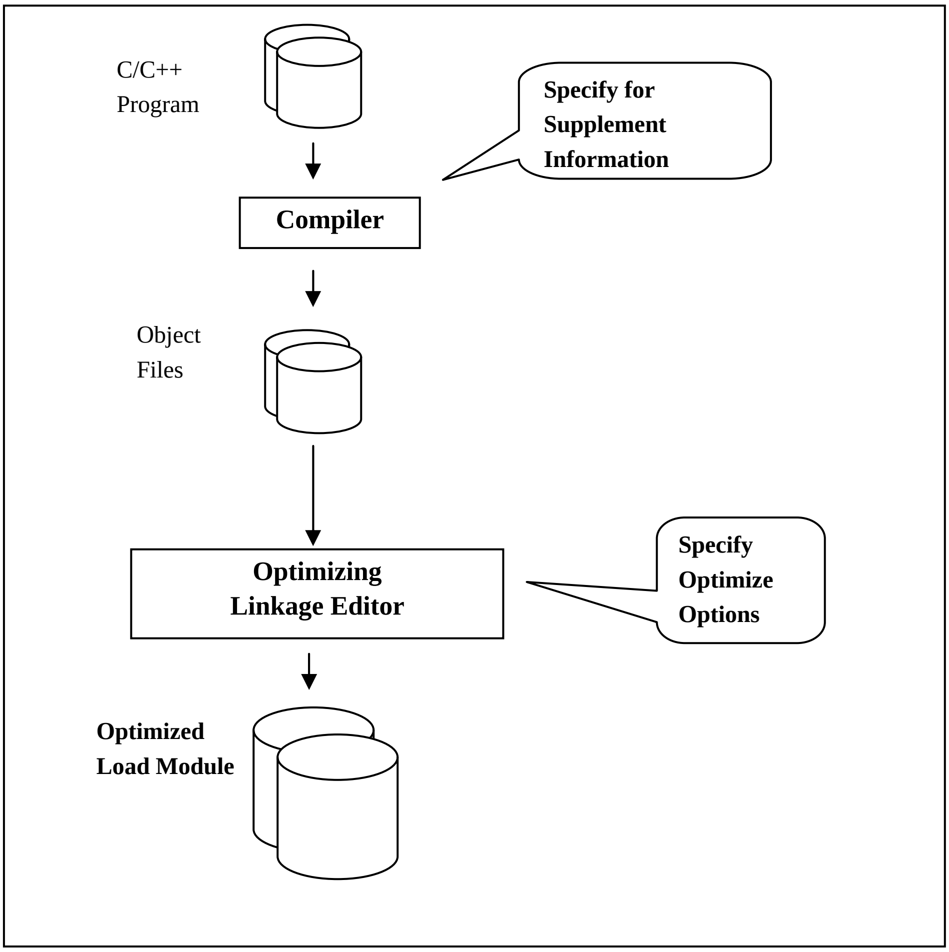
Dialog menu: **Link/Library Tab Category: [Optimize] Optimize items**
Command line: *optimize=<suboption>*
<suboption> is described in 9.4.2, Unifies Constants/Strings, to 9.4.6, Optimizes Branch Instructions.

The following specification for supplement information is necessary at compile, even if optimization at linkage is specified. Without the following specification, optimization at linkage is not available.

• Specification Method for Supplement Information

Dialog menu: **C/C++ Tab Category: [Optimize] Generate file for inter-module optimization**
Command line: *goptimize*

- **Inter-Module Optimization Flow**



9.4.2 Unifies Constants/Strings

Size	○	Speed	-

• Description

The same value constants and the same strings having the const attribute are unified across the modules.
This option deletes const section to improve Size.
Speed is not changed.

• Specification Method

Dialog menu: **Link/Library Tab Category: [Optimize] Optimize items**
Unify strings
Command line: *optimize=string_unify*

• Examples of the sane value constants

The **const long** variables “**cl1**, **cl2**” which have the same constant value are unified to one constant.
This reduces ROM size by 4 bytes.

```
(file1.c)
#include <machine.h>
const long cl1=100;
void main(void);
void func01(long);
long g_max;
void main(void)
{
        func01(cl1+1);
        func02(cl1+2);
        func03(cl1+3);
}
void func01(long c_litr)
{
        g_max = c_litr++;
}
```

Deleted

```
(file2.c)
#include <machine.h>
const long cl2=100;
void main(void);
void func02(long);
void func03(long);
extern long g_max;

void func02(long c_litr)
{
        func03(cl2+c_litr);
        nop();
}
void func03(long c_litr)
{
        g_max = c_litr;
}
```

9.4.3 Eliminates Unreferenced Symbols

Size	○	Speed	-

- **Description**

Variables/functions which are never referred are deleted. When specifying this optimization, an entry function should be specified. Without an entry function specification, this optimization is not performed.
This is because CPU jumps from vector table to entry function, and the optimization of entry functions or the functions whose address is before entry functions changes the jump address.

- **Specification Method**

Dialog menu: **Link/Library Tab Category: [Optimize] Optimize items**
Eliminate dead code
Command line: *optimize=symbol_delete*

- **Specification Method for Entry Functions**

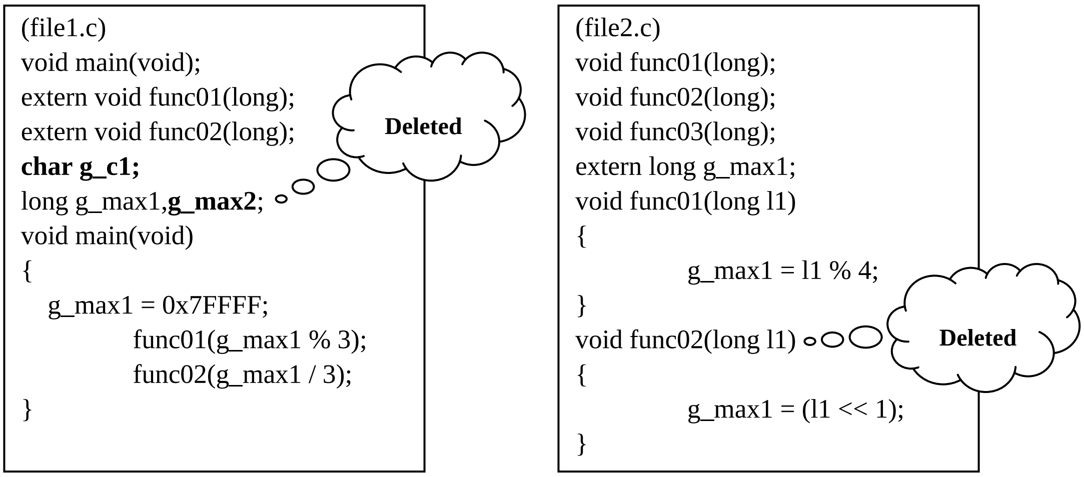
Dialog menu: **Link/Library Tab Category: [Input] Use entry point**
Command line: *entry=<symbol name>* | *<address>*

When specify symbol name, add an underscore (_) at the head of the name.

Example: main -> _main

- **Examples of eliminates unreferenced variables/functions**

Variable **g_max2** and function **func03** which are never referred are deleted.

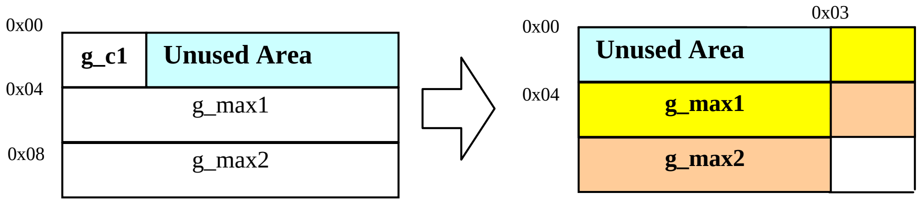


```
(file1.c)
void main(void);
extern void func01(long);
extern void func02(long);
char g_c1;
long g_max1,g_max2;
void main(void)
{
   g_max1 = 0x7FFFF;
            func01(g_max1 % 3);
            func02(g_max1 / 3);
}
```

```
(file2.c)
void func01(long);
void func02(long);
void func03(long);
extern long g_max1;
void func01(long l1)
{
            g_max1 = l1 % 4;
}
void func02(long l1)
{
            g_max1 = (l1 << 1);
}
```

The **char** type variable **g_c1** is never referred, but is not deleted.
This is because SH is 4-byte boundary alignment, and if **g_c1** is deleted, the address of next variable is not multiples of four.
The access for the odd address symbol occurs an address error because of the CPU specification

[If 1-byte variable is deleted]



If optimization is performed, 4-byte variable **g_max1** is accessed by address 0x03.

9.4.4 Optimizes Register Save/Restore Codes

Size	○	Speed	○

- **Description**

The relationships between function calls are analyzed and redundant register save/restore codes are deleted with this specification. In addition, depending on the register state before and after the function call, the register numbers to be used are modified.

- **Specification Method**

Dialog menu: **Link/Library Tab Category: [Optimize] Optimize items**
Reallocate registers
Command line: *optimize=register*

- **Examples of Optimizes register save/restore codes**

Function **func1** calls function **func2** and **func3**.

```
(file1.c)
void func1(int i1,int i2,int i3,int i4,int i5,long *l6)
{
   a = 0 * i1;
   b = 1 * i2;
   d = 4 * i4;
   h = 8 * i5;
   i = 9
   *l6 = b;
   func2(i,h,3000,200,100,l6);
   func3(i,h,3000,200,100,l6);
}
```

```
(file2.c)
void func2(int i1,int i2,int i3,int i4,int i5,long *l6)
{
   a = 0 * i1;
   b *= 1 * i2;
   d *= 4 * i4;
   f *= 6 / i2;
   g = 7 / i3;
   h = 8 * i5;
   i *= 9 / b ;
   *l6 = b * g;
}
```

```
(file3.c)
void func3(int i1,int i2,int i3,int i4,int i5,long *l6)
{
    a = 0 * i1;
    b *= 1 * i2;
    c = 2 * i3;
    d *= 4 * i4;
    e *= 5 * i1;
    f *= 6 / i2;
    g *= i2 / i3;
    h *= 8 * i5;
    i *= (9 / b) * ((*l6)++);
    *l6 *= b * g;
}
```

- **Examples of Codes by Optimizes register save/restore codes**

Examples of codes before and after this optimization are as follows.
Due to the addition of register save/restore codes in the parent function, register save/restore codes in the child function are reduced.

In the following example, which is SH-1,
ROM Size: 532 bytes to 524 bytes
Execution Speed: 718 cycles to 711 cycles

(Before Optimization)
save/restore R13-R14 (**2 registers**)

```
_func1:
        MOV.L       R13,@-R15
        MOV.L       R14,@-R15
           …
        MOV.L       @R15+,R14
        RTS
        MOV.L       @R15+,R13
```

(After Optimization)
save/restore ER2-ER4 (**7 registers**)

```
_func1:
        MOV.L       R8,@-R15
        MOV.L       R9,@-R15
        MOV.L       R10,@-R15
        MOV.L       R11,@-R15
        MOV.L       R12,@-R15
        MOV.L       R13,@-R15
        MOV.L       R14,@-R15
           …
        MOV.L       @R15+,R14
        MOV.L       @R15+,R13
        MOV.L       @R15+,R12
        MOV.L       @R15+,R11
        MOV.L       @R15+,R10
        MOV.L       @R15+,R9
        RTS
        MOV.L       @R15+,R8
```

save/restore R10,R11, R12, R14
(**4 registers**)

```
_func2:
        MOV.L       R10,@-R15
        MOV.L       R11,@-R15
        MOV.L       R12,@-R15
        MOV.L       R14,@-R15
           …
        MOV.L       @R15+,R14
        MOV.L       @R15+,R12
        MOV.L       @R15+,R11
        RTS
        MOV.L       @R15+,R10
```

NO save/restore (**0 registers**)

```
_func2:
        STS.L      PR,@-R15
           …
        LDS.L       @R15+,PR
        NOP
        RTS
        NOP
```

save/restore R8-R14 (**7 registers**)

```
_func3:
        MOV.L     R8,@-R15
        MOV.L     R9,@-R15
        MOV.L     R10,@-R15
        MOV.L     R11,@-R15
        MOV.L     R12,@-R15
        MOV.L     R13,@-R15
        MOV.L     R14,@-R15
             …
        MOV.L     @R15+,R14
        MOV.L     @R15+,R13
        MOV.L     @R15+,R12
        MOV.L     @R15+,R11
        MOV.L     @R15+,R10
        MOV.L     @R15+,R9
        RTS
        MOV.L     @R15+,R8
```

save/restore R13,R14 (**2 registers**)

```
_func3:
        MOV.L     R13,@-R15
        MOV.L     R14,@-R15
             …
        MOV.L     @R15+,R14
        RTS
        MOV.L     @R15+,R13
```

9.4.5 Unifies Common Codes

Size	O	Speed	-

• Description

Multiple strings representing the same instruction are unified into a subroutine and the code size is reduced with this specification. This optimization increases the overhead of function call and decreases execution speed, so should be careful.

The minimum code size for the optimization with the same-code unification can be specified.

When inline expansion of functions is specified at compile, this optimization is not performed, as execution speed is decreased.

• Specification Method

Dialog menu: **Link/Library Tab Category: [Optimize] Optimize items**
Eliminate same code
Command line: *optimize=same_code*

• Specification Method for Unification Size

Dialog menu: **Link/Library Tab Category: [Optimize] Eliminated size**
Command line: *samesize=<size>*

- **Examples: C Source Programs**

Function **func00** and **func01** have the same lines of expressions.

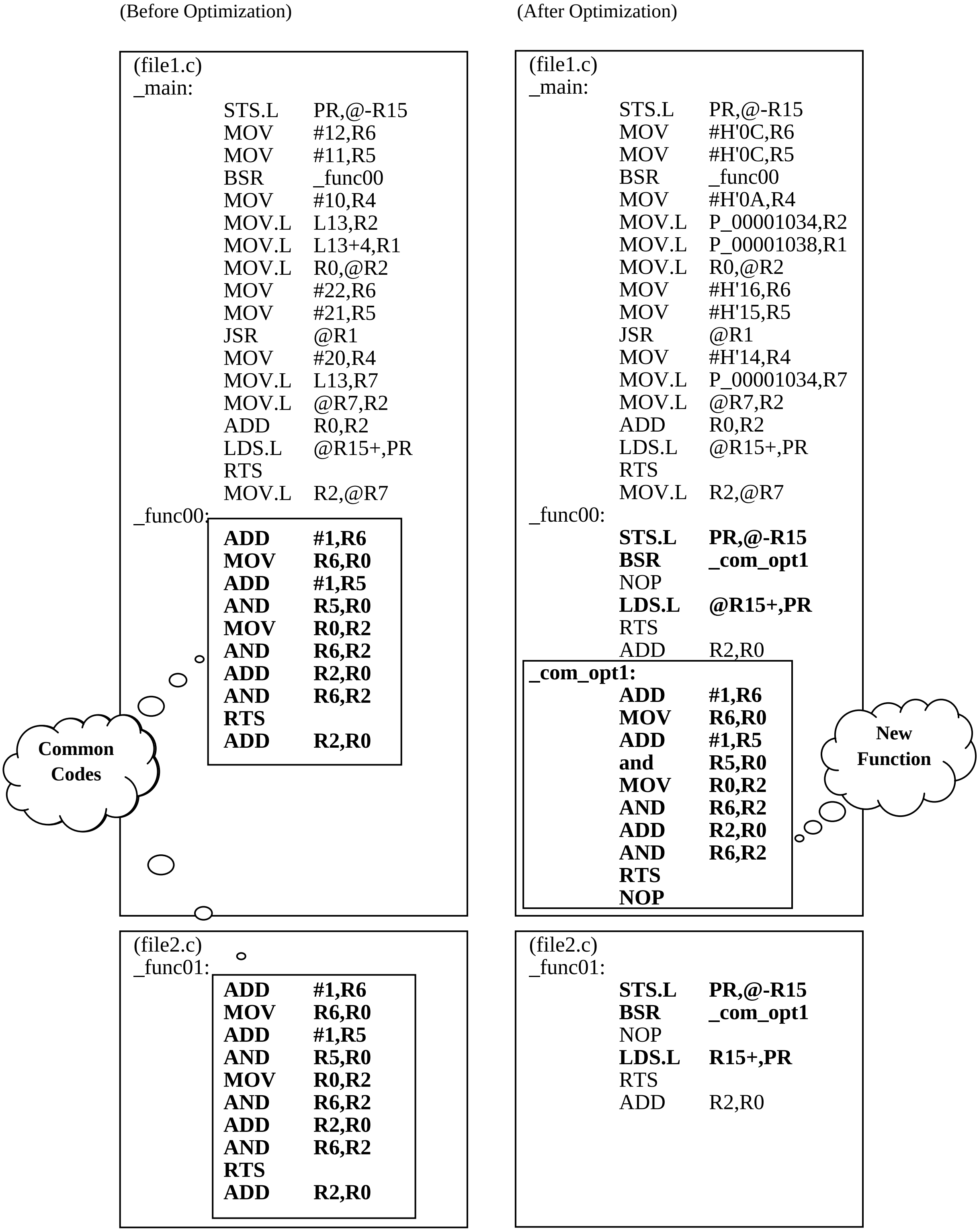
```
(file1.c)
void main(void);
int func00(int,int,int);
extern int func01(int,int,int);
int ret;
void main(void)
{
        ret =  func00(10,11,12);
        ret += func01(20,21,22);
}
int func00(int i1,int i2,int i3)
{
        i1++;
        i2++;
        i3++;
        i1 = i3 & i2;
        i2 = i1 & i3;
        i3 = i2 & i3;
        return i1+i2+i3;
}
```

```
(file2.c)
void func01(void);
int func01(int,int,int);
int func01(int i1,int i2,int i3)
{
        i1++;
        i2++;
        i3++;
        i1 = i3 & i2;
        i2 = i1 & i3;
        i3 = i2 & i3;
        return i1+i2+i3;
}
```

• Examples: Codes

Examples of codes before and after this optimization are as follows.
Common codes are unified into a new function **_com_opt1**, which is called from the original positions.

In the following example, which is SH-1,
ROM Size: 40 bytes to 24 bytes
Execution Speed: 46 cycles to 60 cycles



(Before Optimization)

```
(file1.c)
_main:
          STS.L    PR,@-R15
          MOV      #12,R6
          MOV      #11,R5
          BSR      _func00
          MOV      #10,R4
          MOV.L    L13,R2
          MOV.L    L13+4,R1
          MOV.L    R0,@R2
          MOV      #22,R6
          MOV      #21,R5
          JSR      @R1
          MOV      #20,R4
          MOV.L    L13,R7
          MOV.L    @R7,R2
          ADD      R0,R2
          LDS.L    @R15+,PR
          RTS
          MOV.L    R2,@R7
_func00:
          ADD      #1,R6
          MOV      R6,R0
          ADD      #1,R5
          AND      R5,R0
          MOV      R0,R2
          AND      R6,R2
          ADD      R2,R0
          AND      R6,R2
          RTS
          ADD      R2,R0
```

```
(file2.c)
_func01:
          ADD      #1,R6
          MOV      R6,R0
          ADD      #1,R5
          AND      R5,R0
          MOV      R0,R2
          AND      R6,R2
          ADD      R2,R0
          AND      R6,R2
          RTS
          ADD      R2,R0
```

(After Optimization)

```
(file1.c)
_main:
          STS.L    PR,@-R15
          MOV      #H'0C,R6
          MOV      #H'0C,R5
          BSR      _func00
          MOV      #H'0A,R4
          MOV.L    P_00001034,R2
          MOV.L    P_00001038,R1
          MOV.L    R0,@R2
          MOV      #H'16,R6
          MOV      #H'15,R5
          JSR      @R1
          MOV      #H'14,R4
          MOV.L    P_00001034,R7
          MOV.L    @R7,R2
          ADD      R0,R2
          LDS.L    @R15+,PR
          RTS
          MOV.L    R2,@R7
_func00:
          STS.L    PR,@-R15
          BSR      _com_opt1
          NOP
          LDS.L    @R15+,PR
          RTS
          ADD      R2,R0
_com_opt1:
          ADD      #1,R6
          MOV      R6,R0
          ADD      #1,R5
          and      R5,R0
          MOV      R0,R2
          AND      R6,R2
          ADD      R2,R0
          AND      R6,R2
          RTS
          NOP
```

```
(file2.c)
_func01:
          STS.L    PR,@-R15
          BSR      _com_opt1
          NOP
          LDS.L    R15+,PR
          RTS
          ADD      R2,R0
```

9.4.6 Optimizes Branch Instructions

Size	○	Speed	○

- **Description**

C/C++ Compiler calls functions by the absolute addressing mode (JSR), when access functions in other files, and when access over the address range* which can be accessed by the PC relative addressing mode (BSR).
As the optimizing linkage editor performs optimization at linkage, it can recalculate the branch range of which the branch destination is in other file.
The branch instruction can be changed to the PC relative addressing mode (BSR), if possible.

Though the original branch range exceeds the address range which can be accessed by the PC relative addressing mode, the branch instruction can be also changed to BSR, if the branch range is reduced by other optimization.

If any other optimization item is executed, this optimization is always performed regardless of whether it is specified or not.

Note: *. The address range which can be accessed by the PC relative addressing mode: –4096 to 4094 byte

- **Specification Method**

Dialog menu: **Link/Library Tab Category: [Optimize] Optimize items**
Optimize branches
Command line: *optimize=branch*

- **Examples: C Source Programs**

Function **main** calls function **func01**.

```
(file1.c)
long func01(long,long);
void main(void);
long g_l1,g_l2;
void main(void)
{
   g_l1 = 100;
   g_l2 = 200;
            g_l1 = func01(g_l1,g_l2);
}
                         :
                         :
long func01(long l1,long l2)
{
            return l1 + l2;
}
```

- **Examples: Codes**

Examples of codes before and after this optimization are as follows.
Function **func01** is called by **BSR**.

In the following example, which is SH-1,
ROM Size: 46 bytes to 42 bytes
Execution Speed: 22 cycles to 21 cycles

(Before Optimization)

```
_main:
        STS.L     PR,@-R15
        MOV.L     L13,R1
        MOV       #-56,R5
        MOV.L     L13+4,R2
        MOV       #100,R4
        EXTU.B    R5,R5
        MOV.L     R4,@R1
        MOV.L     L13+8,R3
                JSR       @R3
        MOV.L     R5,@R2
        MOV.L     L13,R7
        LDS.L     @R15+,PR
        RTS
        MOV.L     R0,@R7
L13:
        .DATA.L   _g_l1
        .DATA.L   _g_l2
        .DATA.L   _func01
```

```
_func01:
        ADD       R5,R4
        RTS
        MOV       R4,R0
```

(After Optimization)

```
_main:
        STS.L     PR,@-R15
        MOV.L     L13,R1
        MOV       #--56,R5
        MOV.L     L13+4,R2
        MOV       #100,R4
        EXTU.B    R5,R5
        MOV.L     R4,@R1
        NOP
      BSR       _func01
        MOV.L     R5,@R2
        MOV.L     L13,R7
        LDS.L     @R15+,PR
        RTS
        MOV.L     R0,@R7
L13:
        .DATA.L   _g_l1
        .DATA.L   _g_l2
```

```
_func01:
        ADD       R5,R4
        RTS
        MOV       R4,R0
```

9.4.7 Optimization Partially Disabled

- **Description**

When don’t want to optimize some variables or functions by the optimizing linkage editor, that variables or functions can be specified as follows.
Disablements by the symbol name and by the address range are available.

- **Disables elimination of unreferenced symbols**

- **Specification Method**

Dialog menu: **Link/Library Tab Category: [Optimize] Forbid item**
Elimination of dead code
Command line: *symbol_forbid=<symbol name>*

- **Disables unification of common codes**

- **Specification Method**

Dialog menu: **Link/Library Tab Category: [Optimize] Forbid item**
Elimination of same code
Command line: *samecode_forbid=<function name>*

- **Address Range where optimization is disabled**

- **Specification Method**

Dialog menu: **Link/Library Tab Category: [Optimize] Forbid item**
Memory allocation in
Command line: *absolute_forbid=<address>[+size]*

9.4.8 Confirm Optimization Results

- **Description**

Optimization results by the optimizing linkage editor can be confirmed as follows.

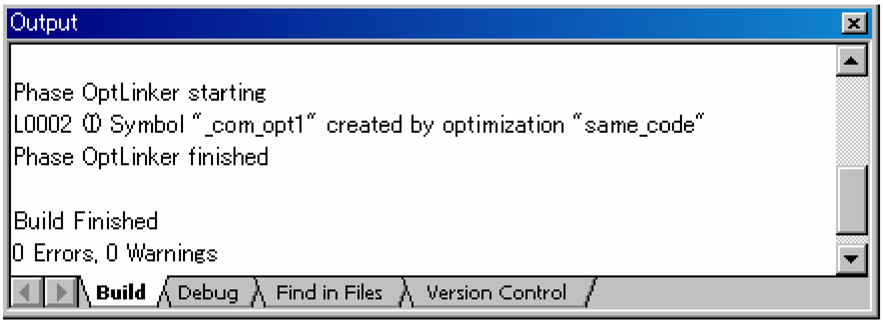
- **Confirmation by message**

When using HEW, optimization results are output by not checking in the following dialog.

Dialog menu: **Link/Library Tab Category: [Output] Show entries for:**
Repressed information level messages
Command line: *message[=<error number>]>*
: *nomessage*

- **Example of message output**

The following example shows that a new function has been created by the unification of common codes.



```
Phase OptLinker starting
L0002 (I) Symbol "_com_opt1" created by optimization "same_code"
Phase OptLinker finished

Build Finished
0 Errors, 0 Warnings
```

- **Confirmation by list**

Optimization results are confirmed by specifying the following options.
For more details, please refer to section 9.2.1, Symbol Information List.

Dialog menu: **Link/Library Tab Category: [List] Contents : Symbol**
Command line: *list [=<file name>]*
Show symbol

Section 10 MISRA C

10.1 MISRA C

10.1.1 What Is MISRA C?

MISRA C refers to the usage guidelines for the C language that were issued by the Motor Industry Software Reliability Association (MISRA) in 1998, as well as the C coding rules standardized by those guidelines. The C language itself is very useful, but suffers from some particular problems. The MISRA C guideline divides these problems into five types: programmer errors, misconceptions about the language, unintended compiler operations, errors at execution, and errors in the compiler itself. The purpose of MISRA C is to overcome these problems, while promoting safe usage of the C language. MISRA C contains 127 rules of two types: *required* and *advisory*. Code development should aim to conform to all of these rules, but as this is sometimes difficult to accomplish, there is also a process to confirm and document times when the rule conformance is not followed. Compliance to various issues is also required separate from these rules, such as when software metrics need to be measured.

10.1.2 Rule Examples

This subsection introduces some actual MISRA C rules. Figure 10.1 shows Rule 62, that all switch statements shall contain a final default clause. This is categorized as a programmer error. In a switch statement, if the "default" label is misspelled as "defalt", the compiler will not treat this as an error. If the programmer does not notice this error, the expected default operation will never be executed. This problem can be avoided through the application of Rule 62.

Example:

```
switch(x) {
    :
defalt:   ⬅ Misspelled
    err = 1;
    break;
}
```

Figure 10.1 Rule 62

Figure 10.2 shows Rule 46, that the value of an expression shall be the same under any order of evaluation that the standard permits. This is categorized as a misconception about the language. Namely, if ++i is evaluated first, the expression becomes 2+2, but if i is evaluated first, the expression becomes 2+1. Likewise, since no provision exists for the evaluation order of function arguments, if ++j is evaluated first, the expression becomes f(2,2), but if j is evaluated first, the expression becomes f(1,2). This problem can be avoided through the application of Rule 46.

Example:

```
i = 1;
x = ++i + i;        x = 2 + 2? x = 2 + 1?

j = 1;
func(j, ++j);       func(1, 2)? func(2, 2)?
```

Figure 10.2 Rule 46

Figure 10.3 shows Rule 38, that the right hand operand of a shift operator shall lie between zero and one less than the width in bits of the left hand operand. This is categorized as an unintended compiler operation. In ANSI, if the shift number of the bit-shift operator is a negative number or larger than the size of the object to be shifted, the calculation results are undefined. In Figure 10.3, if the shift number when us is shifted is not between 0 and 15, the results are undefined and the value will differ depending on the compiler. This problem can be avoided through the application of Rule 38.

Example:

```
unsigned short us;

  us << 16;  ⬅ Undefined action
  us >> -1   ⬅ Undefined action
```

Figure 10.3 Rule 38

Figure 10.4 shows Rule 51, that the evaluation of constant unsigned integer expressions should not lead to wrap-around. This is categorized as an error at execution. When the result of an unsigned integer calculation is theoretically negative, it is unclear whether a theoretically negative value is expected, or a result based on a calculation without the sign will suffice. This situation could lead to a malfunction. Also, the results of an addition calculation may cause an overflow, resulting in a very small value. This problem can be avoided through the application of Rule 51.

Example:

```
   if( 1UL - 2UL )  ⬅ What is intended: -1 or 0xFFFFFFFF?

   *(char*)(0xfffffffeUL + 2);  ⬅ Results in a 0 address.
```

Figure 10.4 Rule 51

10.1.3 Compliance Matrix

With MISRA C, source code is checked for compliance with all 127 rules. In addition, a table as the one shown in Table 10.1 needs to be made, showing whether or not each rule is upheld. This is called a *compliance matrix*. Given the difficulty of visually checking all rules, we recommend that you use a static check tool. The MISRA C guideline also indicates such, stating that the use of a tool to adhere to rules is of utmost importance. As not every rule can be checked using such a tool, you will need to perform a visual review to check such rules visually.

Table 10.1 Compliance Matrix

Rule number	Compiler	Tool 1	Tool 2	Review (visual)
1	Warning 347			
2		Violation 38		
3			Warning 97	
4				Pass
...	...	...	...	...

10.1.4 Rule Violations

Rule violations can consist of those that are known to be safe, and those that may have more effects. Rule violations such as the former should be accepted, but some degree of safety is lost when rule violations are accepted too easily. This is why MISRA C states a special procedure for accepting rule violations. Such violations require a valid reason, as well as verification that the violation is safe. As such, locations and valid reasons for all accepted rules are documented. So that violations are not accepted too easily, the signature of an individual with appropriate authority within the organization is added to such documentation after consultation with an expert. This means that when a rule that is the same as one already accepted is violated, it is deemed as an "accepted rule violation", and can be treated as accepted, without performing the above procedures again. Of course, such violations need to be reviewed regularly.

10.1.5 MISRA C Compliance

To encourage MISRA C compliance, code needs to be developed in compliance with the rules, and rule violation problems need to be resolved. To show whether code complies with the rules, documentation for the compliance matrix and accepted rule violations is needed, along with signatures for each rule violation. To prevent future problems, you should train programmers to make the most of the C language and tools used, implement policies regarding coding style, choose adequate tools, and measure software metrics of various kinds. Such efforts should be officially standardized, along with the appropriate documentation. MISRA C compliance requires more than just development of individual products according to the guidelines, but rather of the organization itself.

10.2 SQMlint

10.2.1 What Is SQMlint?

SQMlint is a package that provides the Renesas C compiler with the additional function for checking whether it conforms to the MISRA C rules. SQMlint statically checks the C source code, and reports the areas that violate the rules. SQMlint runs as part of the C compiler in the Renesas product development environment. SQMlint can be started simply by adding an option at compile-time, as shown in figure 10.5. It in no way affects the code generated by the compiler.

Table 10.2 lists the rules supported by SQMlint.

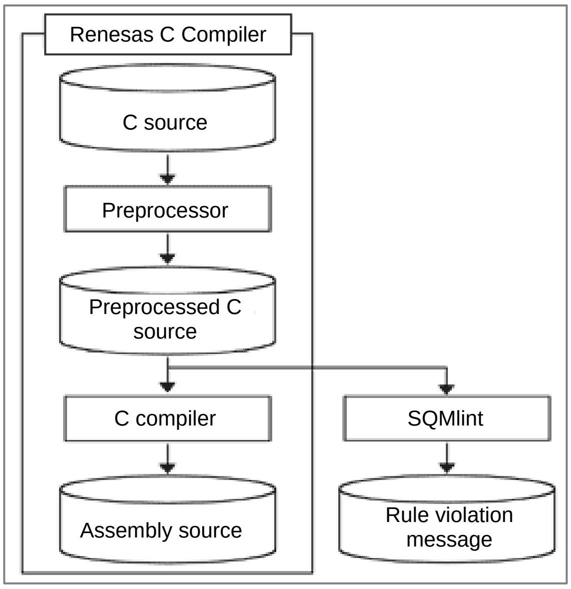


Figure 10.5 SQMlint Positioning

Table 10.2 Rules Supported by SQMlint

Rule	Test	Rule	Test	Rule	Test	Rule	Test	Rule	Test	Rule	Test
1	○	26	×	51	○*	76	○	101	○	126	○
2	×	27	×	52	×	77	○	102	○	127	○
3	×	28	○	53	○	78	○	103	○		
4	×	29	○	54	○*	79	○	104	○		
5	○	30	×	55	○	80	○	105	○		
6	×	31	○	56	○	81	×	106	○*		
7	×	32	○	57	○	82	○	107	×		
8	○	33	○	58	○	83	○	108	○		
9	×	34	○	59	○	84	○	109	×		
10	×	35	○	60	○	85	○	110	○		
11	×	36	○	61	○	86	×	111	○		
12	○	37	○	62	○	87	×	112	○		
13	○	38	○	63	○	88	×	113	○		
14	○	39	○	64	○	89	×	114	×		
15	×	40	○	65	○	90	×	115	○		
16	×	41	×	66	×	91	×	116	×		
17	○*	42	○	67	×	92	×	117	×		
18	○	43	○	68	○	93	×	118	○		
19	○	44	○	69	○	94	×	119	○		
20	○	45	○	70	○*	95	×	120	×		
21	○*	46	○*	71	○	96	×	121	○		
22	○*	47	×	72	○*	97	×	122	○		
23	×	48	○	73	○	98	×	123	○		
24	○	49	○	74	○	99	○	124	○		
25	×	50	○	75	○	100	×	125	○*		

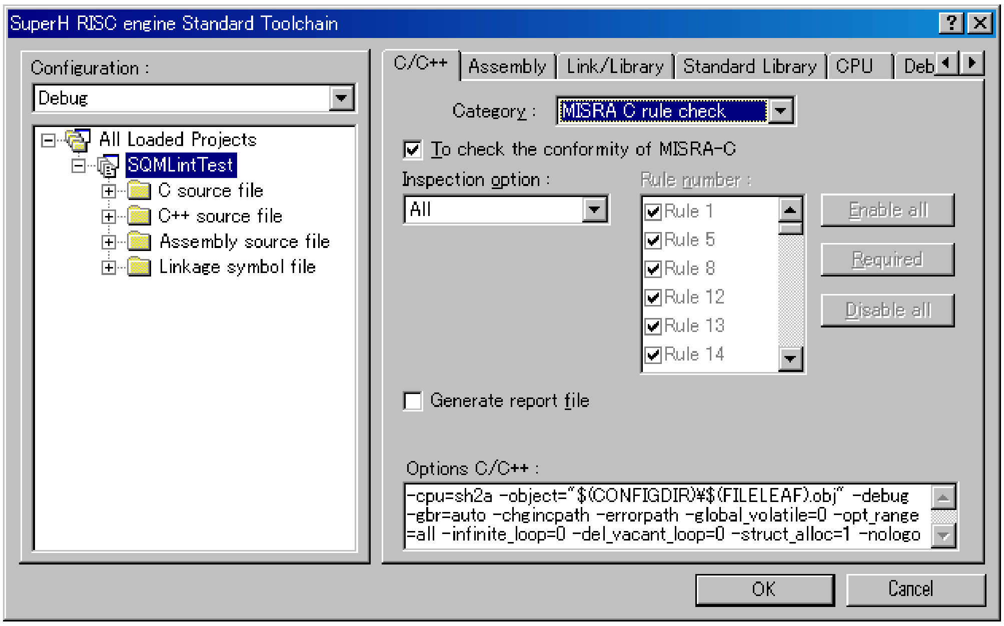
○: Testable ×: Not testable *: Testable with limitations

Table 10.3 Number of Rules Supported by SQMlint

Rule category	Number of testable rules (Supported by SQMlint / Total)
Required	67/93
Advisory	19/34
Total	86/127

10.2.2 Using SQMlint

SQMlint start options can be set easily from the window for setting the HEW Compile Options. Figure 10.6 shows the dialog box for specifying HEW options, in which [MISRA C rule check] should be selected from [Category].



Configuration : Debug

Category : MISRA C rule check

☑ To check the conformity of MISRA-C

Inspection option : All

Rule number :
☑ Rule 1
☑ Rule 5
☑ Rule 8
☑ Rule 12
☑ Rule 13
☑ Rule 14

☐ Generate report file

Options C/C++ :

```
-cpu=sh2a -object="$(CONFIGDIR)¥$(FILELEAF).obj" -debug
-gbr=auto -chgincpath -errorpath -global_volatile=0 -opt_range
=all -infinite_loop=0 -del_vacant_loop=0 -struct_alloc=1 -nologo
```

Figure 10.6 HEW Options Window

Thus, SQMlint will start at compile-time. The meaning of [Inspection Option] in this dialog is:

- [All]: Performs testing for all rules.
- [Required]: Performs testing only for rules necessary according to the MISRA C rule.
- [Custom]: Performs testing for the rules specified by the user. Please select the rules by using the check box and the buttons of the right-side.

10.2.3 Viewing Test Results

Test results can be output in the following three ways:

(a) Standard error output

Messages are output the same as HEW compile errors. A tag jump can be performed by double-clicking the message, or right-clicking the message and choosing [Jump]. . The source code can be easily corrected by the same operation as the compile error.

Note that an explanation is displayed by right-clicking the message and choosing [Help].

(b) CSV file

A file format that can be read by spreadsheet software, allowing reviews to be performed more easily.

(c) SQMmerger

SQMmerger is a tool for merging a C source file with CSV-formatted report file generated by SQMlint into a file that contains C source lines and their associated report messages.

To execute SQMmerger, use the following command entry format:

```
sqmmerger -src <c-source-file-name> -r <report-file-name> -o <output-file-name>
```

Displays both the source file and test results, as shown in figure 10.7.

```
1 : void func(void);
2 : void func(void)
3 : {
4 : LABEL:
  [MISRA(55) Complain] label ('LABEL') should not be used
5 :
6 : goto LABEL;
   [MISRA(56) Complain] the 'goto' statement shall not be used
7 : }
```

Figure 10.7 SQMmerger

10.2.4 Development Procedures

Figure 10.8 shows how to perform development using SQMlint.

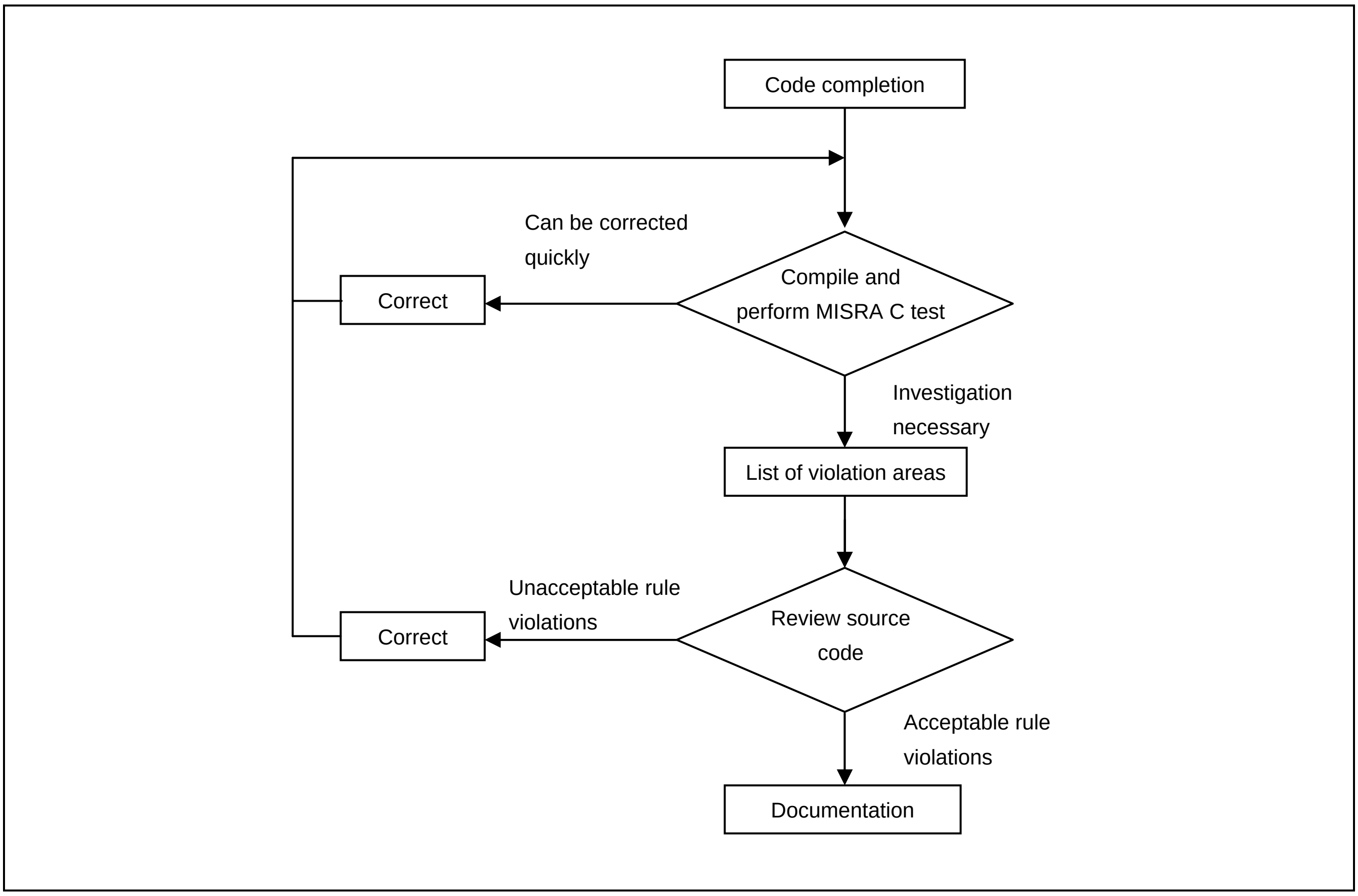


Figure 10.8 Development Procedure Using SQMlint

- Collect all compile errors. SQMlint assumes that the C source code is valid.
- Find errors detected by SQMlint.
- Correct the errors that can be easily corrected.
- Create a list of the locations of rule violations that require investigation, and perform a review.
- Perform corrections for rules deemed unacceptable upon review.
- Document rules deemed acceptable upon review, to leave a record.

10.2.5 Supported Compilers

The following compilers are supported by SQMlint:

- SH C/C++ Compiler Package V.9.00 Release00 and later

10.2.6 Rules That Can Be Checked by the SH C/C++ Compiler

The following rules cannot be checked by SQMlint, but violations of the rules can be detected via SH C/C++ compiler messages.

Table 10.4 Rules That Can Be Checked with the SH C/C++ Compiler

Rule Number	Rule Description	SH C/C++ Compiler Message
9	Comments shall not be nested.	`C5009 (I) Nested comment is not allowed` A nested comment exists.
26	When an object or function is declared more than once, the declarations shall be compatible.	`C2136 (E) Type mismatch` A variable or function with the extern or static memory class has been declared more than once, but the types do not match.
52	All statements shall be reachable.	`C0003 (I) Unreachable statement` A statement exists that will not be executed.

Section 11 Q & A

This section presents answers to questions frequently asked by users.

11.1 C/C++ Compiler/Assembler

11.1.1 const Declaration

Question

I performed a const declaration, but cannot assign it to a constant area (C) section.

Answer

Declaring a symbol using const has the following effect.

(1) const char msg[]="sun";
C section assignment: character string "sun"

(2) const char *msg[]=("sun", "moon");
C section assignment: character strings "sun" and "moon"
D section assignment: msg[0] and msg[1]
(leading addresses of *msg[0] and *msg[1])

(3) const char *const msg[]=("sun", "moon");
C section assignment: character strings "sun" and "moon", msg[0] and msg[1]
(leading addresses of *msg[0] and *msg[1])

(4) char *const msg[]=("sun", "moon");
C section assignment: character strings "sun" and "moon", msg[0] and msg[1]
(leading addresses of *msg[0] and *msg[1])

11.1.2 Correct Evaluation of Single-Bit Data

Question

I tried to determine whether a single bit in a bit field was set or not, but in some cases was not able to evaluate the bit correctly.

Answer

When single-bit data is declared as signed, that bit is interpreted as the sign bit.

Hence values represented using the single bit are "0" and "-1".

In order to represent the values "0" and "1", be sure to declare the data as unsigned.

Examples:

Evaluation always incorrect

```
struct{
        char  p7:1;
        char  p6:1;
        char  p5:1;
        char  p4:1;
        char  p3:1;
        char  p2:1;
        char  p1:1;
        char  p0:1;
}s1;

if(s1.p0 == 1){
    s1.p1 = 0;
}
```

Correct evaluation

```
struct{
        unsigned char  p7:1;
        unsigned char  p6:1;
        unsigned char  p5:1;
        unsigned char  p4:1;
        unsigned char  p3:1;
        unsigned char  p2:1;
        unsigned char  p1:1;
        unsigned char  p0:1;
}s1;

 if(s1.p0 == 1){
    s1.p1 = 0;
}
```

Note: When writing the condition for an if statement, the resulting code is more efficient if comparison is with 0.

11.1.3 Installation

Question

I input commands for the compiler, assembler or linker, but they would not start.

Answer

Check whether the installation directories for the compiler, assembler and linker are included in the "PATH" environment variable.

To start the compiler from the DOS window, set the following environment:

(1) Setting the PATH

Set the PATH option to the place where the tool to be used is located.

```
Example:
  c:\> PATH=%PATH%; C:\Hew3\Tools\Renesas\Sh\9_0_0\bin (RET)
             This should be added to an existing PATH.
```

(2) Setting SHC_LIB

Indicates where the main files of the SuperH RISC engine C/C++ compiler are saved. This setting cannot be omitted.

```
Example:
  c:\> set SHC_LIB=C:\Hew3\Tools\Renesas\Sh\9_0_0\bin (RET)
```

(3) Setting SHC_TMP

Specifies the path for creation of temporary files used by the C/C++ compiler. This setting cannot be omitted.

```
Example:
  c:\> set SHC_TMP=C:\tmp
```

(4) Setting SHC_INC

This environment variable is set when reading the standard header files for the C/C++ compiler from a specified path. Several paths can be specified by separating them with commas (','). If this environment variable is not set, the standard header file is read from SHC_LIB.

```
Example:
  c:\> set SHC_INC=C:\Hew3\Tools\Renesas\Sh\9_0_0\include
```

11.1.4 Runtime Routine Specifications and Execution Speed

Question

Tell me about the speed of the runtime routines provided by the compiler.

Answer

The following is a list of runtime routine speeds/FPL speeds when using internal ROM and RAM. For rules for naming runtime routines, please refer to appendix A, Rules for Naming Runtime Routines. The options for creating a library are as follows:

Table 11.1 Library Creation Options

	cpu	Pic	endian	denormaliaztion	round	fpu	double=float
SH-1	sh1	-	big	-	-	-	None
SH-2	sh2	1	big	-	-	-	None
SH-2A	sh2a	1	big	−	−	−	None
SH-3	sh3	1	big	−	−	−	None
SH-4	sh4	0	big	off	zero	None	−
SH-4A	sh4a	0	big	off	zero	None	−

Table 11.2 List of Runtime Routine Speeds/FPL Speeds (1)

No.	Type		Function Name	Stack Size	Number of Execution Cycles					
					SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
1.1	Integer operations	Multiply	_muli	12	38	-	-	-	-	-
2.1		Divide	_divbs	4	38	38	-	26	24	24
2.2			_divbu	0	28	28	-	19	18	18
2.3			_divws	4	49	50	-	34	31	31
2.4			_divwu	0	39	39	-	26	25	26
2.5			_divls	8	37 / 109	39 / 109	-	26 / 73	20 / 50	21 /61
2.6			_divlsp	12	-	84	-	-	-	-
2.7			_divlspnm	8	-	57	-	-	-	-
2.8			_divlu	4	31 / 82	33 / 84	-	22 /56	17 / 50	19 / 50
3.1		Remainder	_modbs	8	57	60	-	40	33	33
3.2			_modbu	4	39	40	-	27	23	25
3.3			_modws	8	66	69	-	46	39	39
3.4			_modwu	4	49	50	-	34	29	31
3.5			_modls	12	45 / 95	47 / 97	-	31 / 65	23 / 57	23 / 56
3.6			_modlsp	12	-	84	-	-	-	-
3.7			_modlspnm	8	-	57	-	-	-	-
3.8			_modlu	8	34 / 72	36 / 71	-	24 / 48	18 /43	20 /46

Notes: 1. The unit is cycles. Measured values include error.

2. Maximum and minimum pattern values are indicated [maximum / minimum] for routines for which processing differs significantly depending on the input values.

Table 11.2 List of Runtime Routine Speeds/FPL Speeds (2)

No.	Type		Function Name	Stack Size	Number of Execution Cycles					
					SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
4.1	Floating point operations	Add	_adds	24	129	139	60	80	-	-
4.2			_addd_a	44	320	297	147	195	-	-
5.1		Post Increment	_poas	44	135	145	64	84	-	-
5.2			_poad	84	327	303	150	199	-	-
6.1		Substract	_subs	24	144	125	62	86	-	-
6.2			_subdr	44	383	308	149	213	-	-
7.1		Post Decrement	_poss	44	175	192	93	120	-	-
7.2			_posd	84	570	550	302	365	-	-
8.1		Multiply	_muls	24	144	17	9	11	-	-
8.2			_muld_a	64	383	108	50	69	-	-
9.1		Divide	_divs	20	175	17	16	11	-	-
9.2			_divdr	60	570	108	50	69	-	-
10.1		Compare	_eqs	20	16	36	16	24	-	-
10.2			_eqd_a	32	90	108	50	70	-	-
10.3			_nes	20	16	36	16	24	-	-
10.4			_ned_a	32	90	108	50	70	-	-
10.5			_gts	20	33	36	16	24	-	-
10.6			_gtd_a	32	90	108	50	70	-	-
10.7			_lts	20	33	36	16	24	-	-
10.8			_ltd_a	32	90	108	50	70	-	-
10.9			_ges	20	33	36	16	24	-	-
10.10			_ged_a	32	90	108	50	70	-	-
10.11			_les	20	33	36	16	24	-	-
10.12			_led_a	32	90	108	50	70	-	-

Notes: The unit is cycles. Measured values include error.

Table 11.2 List of Runtime Routine Speeds/FPL Speeds (3)

No.	Type	Function Name	Stack Size	Number of Execution Cycles					
				SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
11.1	Convert sign	_negs	0	7	7	4	5	-	-
11.2		_negd_a	12	30	39	18	26	-	-
12.1	Convert	_stod_a	12	66	73	35	50	-	-
12.2		_dtos_a	20	122	128	61	82	-	-
12.3		_stoi	12	50	63	21	31	-	-
12.4		_dtoi_a	20	148	141	72	86	-	-
12.5		_stou	12	50	63	21	31	-	-
12.6		_dtou_a	20	148	141	72	86	-	-
12.7		_itos	12	88	91	45	59	-	-
12.8		_itod_a	12	189	179	96	110	-	-
12.9		_utos	8	81	82	46	52	-	-
12.10		_utod_a	8	99	96	51	61	-	-

Notes: The unit is cycles. Measured values include error.

Table 11.2 List of Runtime Routine Speeds/FPL Speeds (4)

No.	Type	Function Name	Stack Size	Number of Execution Cycles					
				SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
13.1	Move area	_quick_evn_mvn	4	12+3*(n/4)					
13.2		_quick_mvn	8	17+3*(n/4) (n<=64)					
				24+1.625*(n/4) (n>=68)					
13.3		_quick_odd_mvn	4	12+3*(n/4)					
13.4		_slow_mvn	12	21+5*n+3*((n-1)/4)					
14.1	Compare character string	_quick_strcmp1	0	26+7*(n/4)+5*((n-1)%4)					
14.2		_slow_strcmp1	0	35+7*n					
15.1	Copy character string	_quick_strcpy	16	30+6*(n/4)+4*((n-1)%4)					
15.2		_slow_strcpy	24	24+6*n+2*((n-1)/4)					
16.1	Left-shift	_sftl	4	19 / 42	21 / 39	-	-	-	-
17.1	Right shift	_sftrl	0	19 / 42	21 / 39	-	-	-	-
17.2		_sftra	4	20 /43	22 / 47	-	-	-	-
17.3		_sta_sftr6	0	8	9	-	-	-	-
17.4		_sta_sftr7	0	10	11	-	-	-	-
17.5		_sta_sftr10	0	7	8	-	-	-	-
17.6		_sta_sftr11	0	8	9	-	-	-	-
17.7		_sta_sftr12	0	9	10	-	-	-	-
17.8		_sta_sftr13	0	10	11	-	-	-	-
17.9		_sta_sftr21	0	10	11	-	-	-	-
17.10		_sta_sftr27	0	10	11	-	-	-	-
17.11		_sta_sftr28	0	10	11	-	-	-	-
17.12		_sta_sftr29	0	10	11	-	-	-	-
18.1	Packed structure	_pack1_st16	4	12	13	5	10	6	8
18.2		_pack1_st32	4	18	19	8	16	8	12
18.3		_pack1_st64	4	33	35	16	30	16	22
18.4		_pack1_ld16	4	17	18	10	13	11	14
18.5		_pack1_ld32	4	29	30	17	22	18	-
18.6		_pack1_ld64	8	67	73	38	52	39	53
18.7		_bfs64sp1	60	289 / 599	333 / 580	174 / 339	205 / 392	141 / 295	163 / 266
18.8		_bfs64up1	60	289 / 599	333 / 580	174 / 339	205 / 392	141 / 295	163 / 266
18.9		_bfx64sp1	36	239 / 591	276 / 563	144 / 334	194 / 385	130 / 289	147 / 256
18.10		_bfx64up1	40	227 / 588	264 / 550	144 / 332	186 / 377	124 / 282	149 / 266

Notes: 1. The unit is cycles. Measured values include error.

2. Maximum and minimum pattern values are indicated [maximum / minimum] for routines for which processing differs significantly depending on the input values.

Table 11.2 List of Runtime Routine Speeds/FPL Speeds (5)

No.	Type	Function Name	Stack Size	Number of Execution Cycles					
				SH-1	SH-2	SH-2A	SH-3	SH-4	SH-4A
19.1	longlong	add64	8	32	42	21	27	18	25
19.2		_sub64	8	32	42	21	27	18	25
19.3		_mul64	36	134	92	40	64	48	45
19.4		_div64s	64	148 / 601	165 / 351	87 / 183	108 / 245	72 / 195	64 / 161
19.5		_div64u	60	121 / 527	137 / 326	74 / 169	90 / 227	59 / 182	51 / 152
19.6		_mod64s	64	142 /550	158 / 342	80 / 179	105 / 241	65 / 190	61 / 155
19.7		_mod64u	60	117 / 569	132 / 312	70 / 165	87 / 223	55 / 178	48 / 147
19.8		_neg64	8	26	33	17	24	15	19
19.9		_not64	8	24	31	16	21	15	19
19.10		_and64	8	32	42	19	28	18	26
19.11		_or64	8	32	42	19	28	18	26
19.12		_xor64	8	32	42	19	28	18	26
19.13		_shlld64	20	86	96	35	45	27	35
19.14		_shlrd64	20	85	94	37	48	29	40
19.15		_shard64	24	93	105	38	49	29	39
19.16		_bfs64s	52	133 / 446	157 / 404	82 / 241	79 / 266	51 /205	59 / 160
19.17		_bfs64u	52	133 / 446	157 / 404	82 / 241	79 / 266	51 /205	59 / 160
19.18		_bfx64s	24	89 / 441	105 / 392	47 / 238	71 / 262	43 / 202	42 / 151
19.19		_bfx64u	24	77 / 428	93 / 379	49 / 238	63 / 254	37 / 195	38 / 148
19.20		_cmplt64	4	23	26	12	16	13	16
19.21		_cmplt64u	4	23	26	12	16	13	16
19.22		_cmpgt64	4	23	26	12	16	13	16
19.23		_cmpgt64u	4	23	26	12	16	13	16
19.24		_cmple64	4	23	26	12	16	13	16
19.25		_cmple64u	4	23	26	12	16	13	16
19.26		_cmpge64	4	23	26	12	16	13	16
19.27		_cmpge64u	4	23	26	12	16	13	16
19.28		_cmpeq64	4	23	27	12	17	13	17
19.29		_cmpne64	4	24	28	14	18	14	18
19.30		_convi64	8	21	26	11	20	11	13
19.31		_convu64	8	18	23	9	18	10	13
19.32		_convs64	20	146	147	81	97	-	-
19.33		_convs64u	20	146	147	81	97	-	-
19.34		_convf64	20	-	-		-	74	67
19.35		_convf64u	20	-	-		-	74	67
19.36		_convw64	20	175	161	86	102	-	-
19.37		_convw64u	20	175	161	86	102	-	-
19.38		_convd64	20	-	-	-	-	75	77
19.39		_convd64u	20	-	-	-	-	75	77
19.40		_conv64i	0	4	4	3	3	3	4
19.41		_conv64u	0	4	4	3	3	3	4
19.42		_conv64s	24	258	260	141	166	-	-
19.43		_conv64us	24	242	246	136	156	-	-
19.44		_conv64f	28	-	-	-	-	78	75
19.45		_conv64uf	28	-	-	-	-	71	65
19.46		_conv64w	20	164	168	88	111	-	-
19.47		_conv64uw	20	133	140	72	93	-	-
19.58		_conv64d	20	-	-	-	-	80	84
19.59		_conv64ud	20	-	-	-	-	67	70

Notes: 1. The unit is cycles. Measured values include error.

2. Maximum and minimum pattern values are indicated [maximum / minimum] for routines for which processing differs significantly depending on the input values.

Table 11.3 List of Runtime Routine Speeds/FPL Speeds

No.	Type	Function Name	Stack Size	Number of Execution Cycles		
				SH2-DSP	SH3-DSP	SH4AL-DSP
1.1	DSP	_padd24	8	50	33	32
1.2		_padd40	8	60	38	36
1.3		_pdiv16	24	830	514	442
1.4		_pdiv32	36	1164	742	625
1.5		_pdiv24	36	2279	1446	1246
1.6		_pdiv40	36	2750	1696	1439
1.7		_pmul32	16	51	35	32
1.8		_pmul24	24	143	94	87
1.9		_pmul40	44	188	135	105
1.10		_psub24	24	50	33	32
1.11		_psub40	8	60	38	36
1.12		_pconv16s	12	19 / 199	12 / 123	20 / 102
1.13		_pconv16w	16	57 / 212	37 / 126	39 / 115
1.14		_pconv32s	12	20 /340	12 / 196	19 / 140
1.15		_pconv32w	16	53 / 381	34 / 233	37 / 148
1.16		_pconv24s	12	18 / 280	11 / 171	19 / 116
1.17		_pconv24w	16	58 / 286	38 / 168	33 / 172
1.18		_pconv40s	16	29 / 568	18 / 339	24 / 220
1.19		_pconv40w	16	41 / 515	29 / 316	25 / 231
1.20		_pconvs16	16	71 / 1597	47 / 937	50 / 459
1.21		_pconvs32	16	70 / 1341	48 / 809	44 / 457
1.22		_pconvs24	16	104 / 1633	68 / 958	60 / 482
1.23		_pconvs40	16	106 / 1618	70 / 951	64 / 467
1.24		_pconvw16	16	86 / 12374	56 / 7223	49 / 3156
1.25		_pconvw32	20	106 / 3160	68 / 1848	59 / 853
1.26		_pconvw24	20	135 / 10354	86 / 6215	77 / 3172
1.27		_pconvw40	20	142 / 10338	91 / 6207	84 / 3160
1.28		_pcmplt40	4	30	19	17
1.29		_pcmple40	4	30	19	20
1.30		_pcmpgt40	4	30	19	20
1.31		_pcmpge40	4	30	19	20
1.32		_pcmpeq40	4	28	18	16
1.33		_pcmpne40	4	29	18	20
1.34		_pdiv16_sat	28	859	530	459
1.35		_pdiv32_sat	40	1262	790	625
1.36		_pmul32_sat	16	66	42	38

Notes: 1. The unit is cycles. Measured values include error.

2. Maximum and minimum pattern values are indicated [maximum / minimum] for routines for which processing differs significantly depending on the input values.

11.1.5 SH Series Object Compatibility

Question

Are there any problems with linking an object compiled with the compile options "-cpu=sh1" (or sh2, sh2e, sh3, sh4) and "-pic=1"?

Answer

In essence the microcomputers are upward-compatible, so that an SH-1 object and an SH-3 object can be linked and then executed on the SH-3. This means that previous resources can continue to be used without modification.

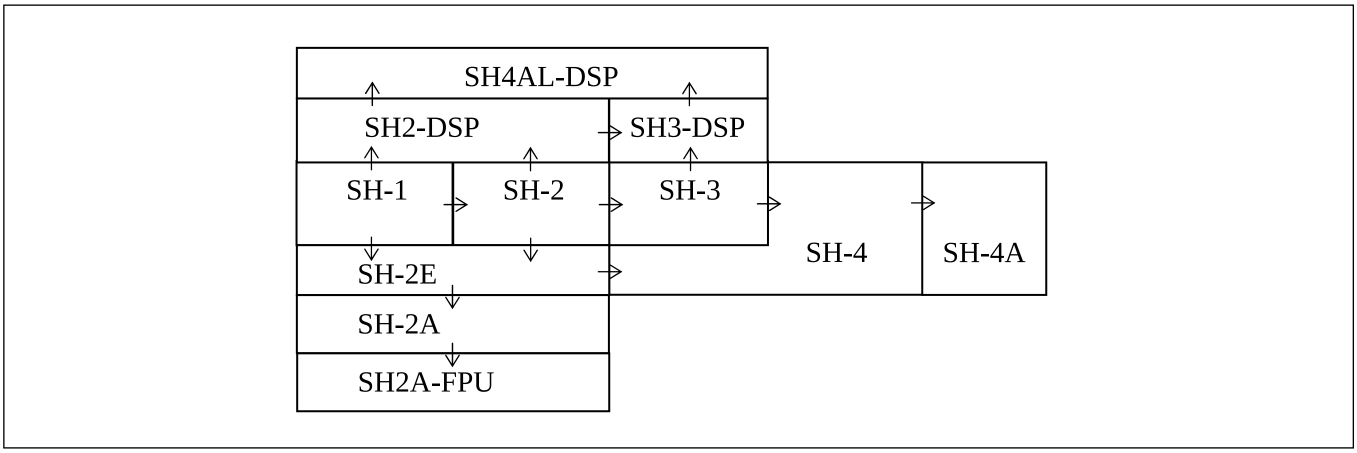


Figure 11.1 Object Compatibility

Note: (1) SH-1, SH-2, SH-2E, SH2-DSP, SH-2A, and SH2A-FPU are big-endian; when objects for these models are used with the SH-3, SH3-DSP, SH4AL-DSP, SH-4, and SH-4A, they should be used as big-endian.
(2) Objects compiled with the "=pic=1" option and objects compiled with the "=pic=0" option can be linked; however, the resulting program will not be position-independent.
(3) Operation during interrupts is different for the SH-3, SH3-DSP, SH4AL-DSP, SH-4, and SH-4A than for the SH-1, SH-2, SH-2E, SH2-DSP, SH-2A, and SH2A-FPU, and interrupt handlers are necessary.

For information on the "-endian" option, refer to section 11.1.15, Data Endian Assignment.

11.1.6 Executing Host Machine and OS

Question

What are executing host machines and OSes?

Answer

The following table lists machines and OSes on which the SuperH RISC engine C/C++ Compiler (ver. 9.0) can run.

Table 11.4 List of Executing Machines and OSes

System name	OS	Notes
HP9000/700 HITACHI9000 HITACHI9000V	HP-UX ver.10.2	
IBM-PC/AT	Windows98/Me/2000/XP/NT	Pentium® processor
SPARC	Solaris ver. 2.5 Solaris ver. 8	

11.1.7 C/C++ Source-Level debugging Not Possible.

Question

I used the "-debug" compiler option, but still can't perform debugging at the C source level.

Answer 1

To output debugging information during linking, as well as at compile-time, you need to specify the appropriate option.

Note that if the directory that contains the source program differs from the one that existed at compile-time, debugging cannot be performed on the C source level. In this case, either return the source program to its original directory, or recompile the program.

For Linker Ver.7 or later:

When specifying the output range during linking, so that output is divided among several files, debugging information will not be appended to each file, but to only one separate file. As such, debugging cannot be performed on the C source level unless the debugging information files are loaded into the debugger.

For Linker Ver.6:

During linking, you can specify a combination of options to output the object format for several types, but some of these cannot be used by the debugger.

From the following table, choose the object format appropriate to the debugger used.

Table 11.5 Options/Subcommands and Compatible Debuggers

Compatible Debuggers	Options/Subcommands	
	Object Format	Debug Information Output
3rd-party debugger supporting ELF/DWARF format	ELF	DEBUG
Hitachi Integration Manager (ver. 4), +E7000	SYSROFPLUS	SDEBUG
Hitachi Integration Manager (ver. 3), +E7000	SYSROF	SDEBUG
Hitachi Debugging Interface (ver. 2), +E6000	SYSROF	DEBUG
Hitachi Debugging Interface (ver. 3), +E6000	ELF	SDEBUG

Answer 2

When -code=asm is specified, debugging cannot be performed at the C source level.

If you use an inline assembler, specify -code=asm.

To perform debugging at the C source level for a project using an inline assembler, specify -code=asm only for files for which the inline assembler is used.

11.1.8 Warning Occurs on Inline Expansion

Question

(1) On attempting inline expansion, the warning "Function (function name) in #pragma inline is not expanded" appeared.

(2) On attempting inline expansion, the warning "Function not optimized" appeared.

Answer

These warning messages do not prevent program execution.

(1) Check whether the function specified by #pragma inline satisfies the conditions for inline expansion.

Functions with the function name specified by #pragma inline and functions specified with the function specifier inline (C++ language) are inline-expanded when they are called. However, in the following circumstances they are not expanded.

- When the function is defined before the #pragma inline specifier
- When the function has a variable parameter
- When a parameter address is referenced within the function
- When calling is performed via the address of the function to be expanded
- From the second operator of a conditional/logical operator

Example:

```
    #pragma inline(A,B)
    int A(int a)
    {
            if(a>10) return 1;
              else return 0;
    }
    int B(int a)
    {
              if(a<25) return 1;
              else return 0;
    }
    void main()
    {
            int a;
              if( A(a)==1 && B(a)==1 )
{
   .....

}
    }
```

A( ) is inline-expanded, but B( ) is not.
(Since there are cases in which the evaluation B(a)==1 need not be performed)

(2) This is due to insufficient memory. When the SuperH RISC engine C/C++ Compiler performs inline expansion, the function size increases, and partway through optimization processing there may be insufficient memory, so that optimization in larger than expression units is no longer possible. To remedy this situation, try the following.

- Do not expand large functions
- Do not expand functions called at numerous locations
- Reduce the number of expanded functions
- Increase the amount of memory available

11.1.9 A "Function not optimized" Warning Appears at Compilation

Question

When I used the "-optimize=1" option to compile, I received a "Function not optimized" warning. Previously I was able to compile this program in the same system environment, using the same compile options, without problem. Why did I receive this warning?

Answer

This warning does not prevent program execution.

The following are possible causes of the warning message.

(1) A compiler limit has been exceeded

During optimization, the compiler generates new internal variables, and in some cases a compiler limit is exceeded. In such cases, functions should be divided into smaller functions.

For more information on compiler limits, refer to section 16.1, Limitations of the Compiler, in the SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual.

(2) Memory is insufficient

If memory is insufficient during optimization processing, the SuperH RISC engine C/C++ compiler issues a warning and ceases optimization of expression and larger units. Compiling is continued, but the end result is the same as if the optimize=0 optimization level was selected. In order to avoid this warning, large functions in the C source program should be broken up into smaller functions.

If this is not possible, the only other choice is to increase the amount of memory available to the compiler.

(3) Cases of inline expansion

Refer to section 11.1.8, Warning Occurs on Inline Expansion.

11.1.10 A "compiler version mismatch" Message Appears at Compilation

Question

On compiling, a fatal error message, "compiler version mismatch", appeared. Why is this?

Answer

Check whether the directories specified by the environment variables "PATH" and "SHC_LIB" are not erroneous.

Examples:

If the environment variables are set as follows, the above error message is output.

`PATH =`(path for SHC ver.8.0)

`SHC_LIB =` (path name for C compiler for SHC ver 6.0)

11.1.11 A "memory overflow" Error Occurs at Compilation

Question

On compiling, the fatal error "memory overflow" occurred. Why is this?

Answer

The following are possible causes of a memory overflow error.

(1) Insufficient memory

(2) Not all the C/C++ compiler files are present in the directory specified by the path name set in the "SHC_LIB" environment variable.

Example:

When the following settings are used, the above error message will appear.
The environment variable is set to SHC_LIB=/SHC/BIN
Files are saved in both /SHC/BIN and in /SHC/MSG
In this case, all files must be present in /SHC/BIN.

(3) Environment variables are not set correctly.

In the case of the PC version, the environment variable "SHC_LIB" should be set not to the directory with the libraries, but to the directory containing SHC.EXE. The batch file SETSHC.BAT created on compiler installation sets the "SHC_LIB" variable to the directory "C:\SHC\BIN" containing the file SHC.EXE.

11.1.12 Precedence of Include Specification

Question

I don't understand all the various options for including files.

Tell me how they're used and the order of precedence.

Answer

The search path for include files is specified using an option or an environment variable.

Files surrounded by "<" and ">" are read from a directory specified using the "-include" option; if multiple directories are specified, they are searched in the order in which they were specified. When a file is not found in the directories specified using the "-include" option, the directory specified by the SHC_INC environment variable is searched, and then the system directory (SHC_LIB) is searched.

Searches for files surrounded by quotes (") begin from the current directory. If they are not found in the current directory, then the above rules are followed for searching.

Briefly, the order of precedence when searching directories for include files is as follows:

```
-inc > SHC_INC > SHC_LIB
```

There is also a "-preinclude" option for forced reading of a file, separate from the above rules. When this option is used, the file specified by this option is placed at the beginning of all files for compiling, and compiling is executed.

By using this option to read a file intended for only temporary use, such as a file containing #pragma statements and test data, recompiling is possible without modifying source files.

11.1.13 Compile Batch Files

Question

There are many options that need to be set when compiling, and it's troublesome to repeat them each time.

Is there some more convenient method?

Answer

At compilation, the "-subcommand" option ("-subcommand=<filename>") can be used.

The "-subcommand" option can be used multiple times on the command line. A subcommand file can contain command line parameters, separated by spaces, carriage returns or tabs. The contents of the subcommand file are expanded into the command line parameters at the position of the subcommand specification.

However, the "-subcommand" option cannot itself be specified within a subcommand file.

Examples:

In the following example, the command line is expanded to be equivalent to

shcΔ-optimize=1Δ-listfileΔ-debugΔ-cpu=sh2Δ-pic=1Δ-sizeΔ-euc

Δ-endian=bigΔtest.c

<u>Command line</u>

shcΔ-sub=test.subΔtest.c

<u>Contents of test.sub</u>

```
-optimize=1
-listfile
-debug
-cpu=sh2
-pic=1
-size
-euc
-endian=big
```

11.1.14 Japanese Text within Programs

Question

I have developed the source code for a program on a workstation and a PC, but the Japanese codes on the workstation and on the PC are different, and it's difficult to manage the source files. Is there an easier way to do this?

Answer

When shift-JIS format is used for Japanese codes, if compiling on a workstation (which uses the EUC encoding for Japanese), the "-sj" compiler option should be used. Conversely, when EUC code is used in a program to be compiled on a PC, the "-euc" compile option should be specified. Even in a workstation network environment in which EUC and shift-JIS codes are intermixed, by setting the appropriate compile option, compiling using either Japanese encoding is possible.

Compiling can be performed using the Japanese code employed on the target machine.

Table 11.6 System and Japanese Code Correspondence

Host	Default
SPARC	EUC
HP9000/700	shift-JIS
PC9800 series	shift-JIS
IBM-PC	shift-JIS

Examples:

When source code is written on a workstation (SPARC) and compiled on a PC (IBM PC), the "-euc" option can be used in compiling, to prevent misinterpretation of Japanese codes in character strings.

11.1.15 Data Endian Assignment

Question

Do the SH models use big-endian or little-endian data?

Answer

The Renesas Tecnology SuperH RISC engine family are big-endian systems.

However, the SH-3, SH3-DSP, SH-4, SH-4A, and SH4AL-DSP support an "-endian=Big(Little)" option to enable CPU big/little-endian switching.

Note:

(1) The "-endian" option can be combined with any arbitrary suboption of the "-cpu" option, but little-endian object programs cannot be executed on products other than the SH-3, SH3-DSP, SH-4, SH-4A, or SH4AL-DSP.

(2) Big-endian objects and little-endian objects cannot be used together.

(3) Differences in endian type may influence the results of program execution.
Example: Code which is affected by endian type

```
f( ){
    int a=0x12345678;
    char *p;

    p=((char *)(&a));

    if(*p==0x12){ (1) }
    else{ (2) }
}
```

In this case, if data is big-endian (1) is executed, but if little-endian, then *p is 0x78, and so (2) is executed.

For more information on data assignment, refer to section 10.1.2 (4), Memory Allocation in Little Endian, in the SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual.

(4) The "-denormalize=on|off" option can be used to select whether to handle non-normalized numbers or treat them as 0. (when -cpu=sh4 or -cpu=sh4a only)

However, when "-denormalize=on", if non-normalized numbers are input to the FPU, an exception occurs. Hence exception processing must be written on software for processing of non-normalized numbers.

11.1.16 Assembling Using "#pragma inline_asm"

Question

When assembling a program using "#pragma inline_asm", an "ILLEGAL DATA AREA ADDRESS" (error no. 452) error occurs.

Answer

(1) Check whether you are compiling with the "-code=asmcode" option.

(2) Check whether there is a data table in the assembly language code.

The following is one possible cause of this situation.

```
#pragma inline_asm(bar)
int bar()
{
        MOV.L    #160,R9
}
```

In the above code, the line

```
        MOV.L    #160,R9
```

is not interpreted by SuperH microcomputers as an instruction to move the value "160" directly to the register.

Normally a data pool must be created and loaded. The assembler automatically recognizes this and creates the data pool; but this generated data does not have the alignment of the assembly language source output by the compiler, and an error results. Such an instance of automatic data generation by an assembler is not anticipated by today's compilers, and so it is not possible to write code in the assembly language source for an inline_asm function which causes the assembler to automatically generate a data pool. However, the code of the above example can be modified as follows to avoid this problem.

Example of modified code

< Prior to modification>

```
        MOV.L    #160,R9
```

< After modification >

```
        MOV         #100,R9
        ADD         #60,R9
```

11.1.17 Privileged Mode

Question

The embedded functions "set_cr" and "get_cr" do not work correctly.

Answer

The above embedded function can only be used in privileged mode in the SH-3 and SH-4.

For general information refer to section 10.3.3, Intrinsic Functions, in the SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual; for information on the priviledged mode, see the hardware manual of each device. Verify that privileged mode is set at the time these embedded functions are called. (In priviledged mode, the SR register MD bit is set.) In order to cause a transition from a non-privileged mode to privileged mode, a TRAPA instruction must be issued.

11.1.18 Regarding Object Generation

Question

When generating an object directly from the compiler, and when generating the object via an assembler, the following occur.

(1) The program sizes are different.
(2) Symbol types are DAT rather than ENT.

Answer

Due to differences in the method of object generation used when directly generating an object and when using an assembler, the resulting load modules are generally different. This is not erroneous operation.

An object output by an assembler does not distinguish between ENT and DAT; this likewise is not erroneous behavior.

11.1.19 About the #pragma gbr_base Feature

Question

When I use the "#pragma gbr_base" feature and load the program into the emulator or try to write it to ROM, an error occurs.

Answer

The sections $G0, $G1 should be treated as initialization data sections.

Normally variables are assigned as follows:

(1) Variables without an initial value specified are assigned to the uninitialized data section (by default, section name "B")
(2) Variables with an initial value specified are assigned to the initialized data section (by default, section name "D")
(3) Variables with a "const" specification are assigned to the constant section (by default, section "C")
However, for variables specified using "#pragma gbr_base" (or gbr_base1) no such distinction is made, and all are assigned to section $G0 (or $G1); hence the compiler treats $G0, $G1 as initialized data areas, and generates an object assuming that "0" is specified in the case of variables for which no initial value was specified.

11.1.20 Compiling Programs Containing Japanese Codes

Question

On compiling a program on a PC that was confirmed to compile correctly on a SPARC workstation, an error occurred.

Answer

Check whether Japanese codes are not included in the source program. The SuperH RISC engine C/C++ compiler supports Japanese codes in both EUC and shift-JIS encodings, but the default encoding is different for different host machines. On a SPARC workstation, the default Japanse encoding is EUC, but on a PC it is shift-JIS. When compiling a program which uses EUC Japanese codes on a PC, the -euc option should be specified. For more information on the default Japanese codes for different host machines, please refer to section 11.11.14, Japanese Text within Programs.

11.1.21 Speed of Floating Point Operations

Question

Tell me about the speed of execution of floating point operations.

Answer

The speed of execution of elementary functions using the standard libraries are shown in table11.8 (for the SH-1, SH-2, SH-3), table 11.9 (SH-2E), table 11.10 (SH-4), table 11.11(SH-4A), and table 11.12(for the SH-2A, SH2A-FPU). For information on the performance of arithmetic operations and other floating point operations, please refer to section 11.1.4, Runtime Routine Specifications and Execution Speed.

Table 11.7 shows the conditions for creating a standard library.

Table 11.7 The Conditions for Creating a Standard Library

Condition	Options for Creating Library						
	Cpu	pic	endian	denormal	round	fpu	double= float
1	sh1	–	big	–	–	–	None
2	sh2	0	big	–	–	–	None
3	sh3	0	big	–	–	–	None
4	sh2e	0	big	–	–	–	None
5	sh4	0	big	off	zero	None	–
6	sh4	0	big	off	zero	single	–
7	sh4	0	big	off	zero	double	–
8	sh4a	0	big	off	zero	None	–
9	sh4a	0	big	off	zero	single	–
10	sh4a	0	big	off	zero	double	–
11	sh2a	0	big	–	–	–	None
12	sh2afpu	0	big	off	zero	None	–
13	sh2afpu	0	big	off	zero	single	–
14	sh2afpu	0	big	off	zero	double	–

Table 11.8 Execution Speed of Floating Point Library Functions (SH-1, SH-2, SH-3)

CPU		SH-1	SH-2	SH-3
Conditions for Creating Library		**1**	**2**	**3**
Single-precision	Sinf	2,438	2,497	1,632
	Cosf	2,384	2,434	1,599
	Tanf	3,120	3,196	2,091
	asinf	5,176	5,418	3,526
	acosf	5,355	5,622	3,659
	atanf	2,924	3,160	2,054
	logf	3,710	3,816	2,490
	sqrtf	3,252	1,018	661
	expf	4,327	4,432	2,873
	powf	4,649	4,824	3,139
Double-precision	sin	5,297	4,964	3,282
	cos	5,289	4,918	3,279
	tan	7,460	7,087	4,673
	asin	13,898	13,788	9,004
	acos	14,158	14,084	9,196
	atan	5,583	5,687	3,712
	log	8,756	8,368	5,535
	sqrt	2,903	2,946	1,803
	exp	9,501	8,952	5,912
	pow	9,337	8,943	5,918

Notes: The unit is cycles. Measured values include error.

Table 11.9 Execution Speed of Floating Point Library Functions (SH-2E)

CPU		SH-2E
Conditions for Creating Library		**4**
Single-precision	sinf	307
	cosf	302
	tanf	343
	asinf	1,267
	acosf	1,289
	atanf	468
	logf	213
	sqrtf	648
	expf	299
	powf	472
Double-precision	sin	3,005
	cos	3,002
	tan	4,339
	asin	8,544
	acos	8,717
	atan	3,434
	log	5,144
	sqrt	1,896
	exp	5,475
	pow	5,437

Notes: The unit is cycles. Measured values include error.

Table 11.10 Execution Speed of Floating Point Library Functions (SH-4)

CPU		SH-4		
Conditions for Creating Library		**5**	**6**	**7**
Single-precision	Sinf	63	59	139
	Cosf	62	59	135
	Tanf	80	78	186
	Asinf	75	71	264
	Acosf	72	72	269
	Atanf	104	72	155
	Logf	86	85	192
	Sqrtf	—*	—*	—*
	Expf	119	100	193
	Powf	387	366	213
Double-precision	Sin	331	70	139
	Cos	310	66	135
	Tan	408	71	186
	Asin	523	71	206
	Acos	616	72	253
	Atan	393	58	145
	Log	403	85	192
	Sqrt	—*	—*	—*
	Exp	403	90	193
	Pow	1,032	366	213

Notes: The unit is cycles. Measured values include error.

* The SH-4 supports the sqrt instruction, and so the sqrt function was omitted.

Table 11.11 Execution Speed of Floating Point Library Functions (SH-4A)

CPU		SH-4A		
Conditions for Creating Library		**8**	**9**	**10**
Single-precision	Sinf	100	95	195
	Cosf	113	96	188
	Tanf	139	134	277
	Asinf	117	113	336
	Acosf	124	123	344
	Atanf	148	122	205
	Logf	131	130	233
	Sqrtf	—*	—*	—*
	Expf	169	146	219
	Powf	408	388	194
Double-precision	Sin	305	110	194
	Cos	288	107	187
	Tan	377	128	247
	Asin	466	113	267
	Acos	558	122	324
	Atan	331	97	191
	Log	344	130	133
	Sqrt	—*	—*	—*
	Exp	387	133	256
	Pow	877	388	219

Notes: The unit is cycles. Measured values include error.

* The SH-4 supports the sqrt instruction, and so the sqrt function was omitted.

Table 11.12 Execution Speed of Floating Point Library Functions (SH-2A,SH2A-FPU)

CPU		SH-2A	SH2A-FPU		
Conditions for Creating Library		**11**	**12**	**13**	**14**
Single-precision	Sinf	1,001	68	65	139
	Cosf	954	68	64	135
	Tanf	1,806	83	82	188
	Asinf	1,545	79	75	273
	Acosf	1,699	74	73	277
	Atanf	1,602	98	73	156
	Logf	1,720	92	93	196
	Sqrtf	562	—*	—*	—*
	Expf	1,463	121	102	208
	Powf	2,140	407	386	246
Double-precision	Sin	3,431	302	77	140
	Cos	3,387	288	72	135
	Tan	4,425	385	75	188
	Asin	5,550	463	75	208
	Acos	5,949	544	73	259
	Atan	3,641	348	59	145
	Log	4,557	401	93	196
	Sqrt	1,622	—*	—*	—*
	Exp	4,137	410	93	208
	Pow	4,086	903	386	246

Notes: The unit is cycles. Measured values include error.

* The SH-4 supports the sqrt instruction, and so the sqrt function was omitted.

11.1.22 Using the PIC Option

Question

I want to program using position-independent code; how do I proceed?

Detailed questions:

(1) I want to transfer multiple applications dynamically to available RAM for execution.
(2) I want to know how to perform initialization processing.
(3) Tell me about practical limits and things to watch out for.

Answer

In order to transfer a program from ROM to a fixed address in RAM for execution, do not use the -PIC option; instead, use the procedure described in 11.2.4, Transfer to RAM and Execution of a Program.

In order to dynamically transfer code to RAM, the -PIC option is convenient; but this option is valid only for program sections, and does not result in position independence for data. Hence data areas can only be loaded to a fixed address.

Because of this limitation, in order to make an entire program (including data) position-independent, special measures must be taken in writing the program.

The following explains the procedure for programming when no data section is included.

- Programming procedure when no data section is included

Example of a program configuration

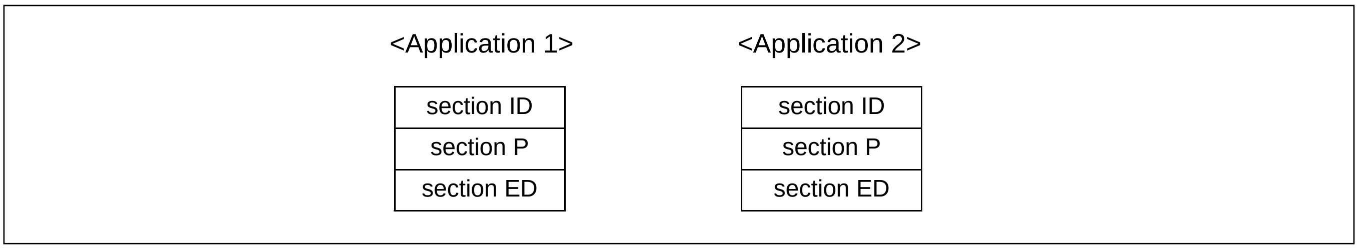


Figure 11.2 Section

C language program

```
<main.c>
main()
{
     int i;
     for (i=0;i<10;i++){
        sub(i);
     }
}
```

```
<sub.c>
sub(int p)
{
    int i;
    for (i=0;i<p;i++){
          ;
    }
}
```

Assembly language program

```
<pic.src>
        .import      _main
        .section     ED,DATA,ALIGN=4    ;  generate the ending section ED
        .section     ID,DATA,ALIGN=4    ;  data section for header
        .data.l      (STARTOF ED)
        .data.l      _main
        .end

<lnk.sub>
        input main
        input sub
        input pic
        start        ID,P,ED/0          ;   assigned starting from address 0; ID at beginning, ED at end
        list  pic
        exit
```

A header (ID section) is added to each program.

The contents of the ID section are:

Offset 0 address	Program size
Offset 4 address	Entry point (main address)

In this manner programs are generated, and the program controlling these calculates the load address and execution address, taking into account ID.

The following shows an example fo the control program.

```
<control.c>
void load_program(int ID){
      char *p;
size=load_ID( ID);               /* load program ID header data           */
                                 /* return value is program size          */
     p=malloc(size);
      if(p!=NULL){
          mload(p,ID);           /* write program data to heap            */
          go((*(long**)p+1)+(long*)p);
                                 /* set PC at leading address of program  */
                                 /* and execute                           */
    }
    else {
       error("Insufficient Memory");
       }
}
```

This is a program image; the method of execution will differ depending on the OS used. The above example should be regarded as a flow-level example when the program is run dynamically.

11.1.23 Optimization Causes Large Amounts of Code to be Deleted

Question

After compiling, large amounts of code are deleted.

Answer

It is possible that the following kinds of optimization are being performed.

(1) Deletion of empty loops

An empty loop, provided to make the program wait for a fixed amount of time, may be deleted through optimization.

Example

```
set_param();                  /* set parameter                                          */
   for(i=0;i<10000;i++);      /* after setting parameter, set result                    */
                              /* empty loop to make the program wait a fixed amount of time */
                              /* the compiler deletes the loop itself                   */
                              /* as being meaningless                                   */
   read_data();               /* acquire result                                         */
                              /* because the loop is deleted, the wait time is eliminated, */
                              /* and an attempt to read the result before it is obtained fails */
```

(2) Deletion of substitution into local variables

Despite the fact that a value is substituted into a local variable, if the value is not referenced, the substitution operation is itself deleted.

Example

```
int data1, data2, data3;
func()
{
     int   res1,res2,res3;

     res1=data1*data2;
     res2=data2*data3;       /*res2 is not referenced after this, and so the expression is deleted    */
     res3=data3*data1;
     sub(res1,res1,res3);  /*      mistake in specifying the second parameter                  */
                           /* if res2 is written instead of res1, the above expression is not deleted    */
}
```

Local variables are valid up to the end of the function, and so normally values are not substituted into local variables within a function and then not referenced. Hence deletion may occur when programming mistakes like the above are made.

11.1.24 Values of Local Variables Cannot be Displayed during Debugging

Question

I can't see the values of local variables.

During debugging, the code references a local variable, but its value cannot be referenced, or is incorrect.

Answer

It is possible that the following kind of optimization has been performed.

(1) Constant operation at compile time

At compile time, any values that are already determined are calculated when compiling and not at runtime, and so variables may themselves be eliminated.

Example 1

```
int x;
func()
{
    int a;
    a=3;
    x=x+a;          /*  here, at compile time this expression becomes x=x+3            */
                    /*  if a is not used elsewhere, then there is no reason to treat a as   */
                    /*  a variable, and it is deleted from debugging information as well  */
}

Example 2
func(int a,int b)
{
    int tmp;
    int len;

    tmp=a*a+b*b;
    len=sq(tmp);    /*  this becomes len=sq(a*a+b*b); and tmp is deleted              */

        :
}
```

These kinds of cases are conceivable, but they have no effect on actual program operation.

(2) Deletion of unreferenced variables

Example 3

```
int data1, data2, data3;
func()
{
     int   res1,res2,res3;

     res1=data1*data2;
     res2=data2*data3;          /* this expression is deleted, and res2 is itself deleted as well */
     res3=data3*data1;
     sub(res1,res1,res3);       /* error in writing the second parameter                          */
                                /* deletion does not occur if res1 is changed to res2             */
}
```

Local variables are valid up to the end of the function, and so normally values are not substituted into local variables within a function and then not referenced. Hence deletion may occur when programming mistakes like the above are made.

11.1.25 Interrupt Inhibit/Enable Macros

Question

I want to use macros for interrupt inhibit/enable processing; how do I proceed?

Answer

This is possible using embedded functions, as in the following example. For further details on embedded functions, refer to section 10.3.3, Intrinsic Functions, in the SuperH RISC engine C/C++ Compiler, Assemble, Optimizing Linkage Editor User's Manual.

Example

```
    #include <machine.h>
    #define disable() { save_cr=get_cr(); set_imask(0x0f); }
    #define enable()  { set_cr(save_cr); }

    function()
{
        int save_cr;

        disable();
        sub();
        enable();
    }
```

11.1.26 Interrupt Functions in SH-3 and Later Models

Question

Are there any differences in the procedure for writing interrupt functions for SuperH microcomputers starting with SH-3?

(1) I want to use multiple interrupts, but using a function with #pragma interrupt specified,

(a) The SSR, SPC save instructions are not available.

(b) The instructions to clear the RB and BL bits of the SR cannot be used.

(c) The SSR, SPC restore instructions are not available.

(2) I want to use a TRAP number specification with a #pragma interrupt statement, but the BL bit of the SR remains 1, and so when a TRAPA instruction is issued an instruction exception occurs.

Answer

The compiler does not output SSR or SPC save/restore instructions. Either they should be written explicitly using the "#pragma inline_asm" feature, or the program should be written using assembler. SR settings can be written using the embedded functions set_cr, get_cr.

In the SH-3 and later models, interrupt processing is greatly changed from processing in the SH-1, SH-2, and SH-2E. In the latter microcomputers, when an interrupt occurs the vector table is referenced, and control branches to the corresponding interrupt routine. In the SH-3 and later, however, branching is to a fixed address. Hence normally an interrupt handler must be placed at the interrupt branching destination in order to inhibit/enable multiple interrupts, evaluate the interrupt factor and start processing for different interrupt factors. Ordinarily such interrupt handlers are written in assembly language.

Refer to sections 2.2 and 2.3, Introduction of Sample Program.

11.1.27 An Operated Result by the Floating Point of SH4

Question

An operated result by the floating point of SH4 does not match an expectated value.

Answer

Compiler has FPU option. There are three patterns which are FPU=single/double/No specification in FPU option. The followings show differences:

FPU=single: All floating point expression is treated as single precision.

FPU=double: All floating point expression is treated as double precision.

FPU=No specification: The precision of a floating point expression follows the type of C description.

According to the FPU option, PR bit setting at FPSCR register is different.

(1) This value (PR bit) is set [0(=single precision)] in the initial condition.
(2) If No specification at FPU option, C compiler generates code which change PR bits at each FPU operation. But a PR changing code is not generated at all when specifying FPU=double/single.
(3) Therefore, option is operated correctly without considering of an above bit when specifying FPU=No specification and FPU=single specification, but specifying [1=(double)] explicitly to PR bit at the users side when specifying FPU=double is required.

11.1.28 Regarding Optimization Options

Question

What will be changed by optimization option (speed, size)?

Answer

Generated codes are changed by specified optimization option. (Do not change Algorithm of User program by optimization.) By optimization, optimize codes like inline expansion of a function and loop unrolling, so the number of times of run-time cycles is changed. Thereby, the timing of operation is also changed. First of all, please verify enough about timing of operation. Moreover, optimization of variable access is also considered as concern matters other than the above. The case that an instruction of data can be realized between registers without memories, and it is corresponded to optimization of variable access, it may be said [Timing verification]. If you want [Do not want to optimize] variable, please confirm including a necessity of an addition of volatile declaration.

11.1.29 An argument of function is not transferred correctly.

Question

An argument of function is not transferred correctly.

Answer

Please confirm whether a prototype of function is declared.

If a prototype of function is not declared, arguments (char, unsigned, char, float) are become an object of automatic type translation. At the time, declaring a function side to call as a changed type is required.

Recommend to declare a prototype of function.

An existence of declaration of a prototype of function can be confirmed by message option of compiler.

11.1.30 How to Check Coding Which May Cause Incorrect Operation

Question

Is there any function to check for potential problem code, such as a missing prototype declaration for a function?

Answer

When coding a program, note that there are some kinds of codes which are not errors in language specifications but may produce incorrect operation results. These codes can be checked by outputting information messages using an option.

```
(example)
  shc Δ -message Δ test.c (RET)

(C language program)

/*  /* COMMENT */              →5009 : String “/*” in a comment
int ;                          →0002 : A declaration without a declarator
void func(int);
void main(void)
{
  long a;
  func(a+1);                   →0006 : Function parameter expression is converted
                                       into the parameter type specified in
                                       prototype declaration
  sub();                       →0200 : No prototype declaration for called function
}
```

[Specification method]
Dialog menu: **C/C++**tab **Category: [Source] Messages**, **Display information level message**
Command line:***message***

Remarks

In the dialog menu, removing the left-side checkmark from a message disables the output of the message. In the command line, specifying an error number in a sub-option of the nomessage option disables the output of the message. For details on error numbers, refer to section 12, Compiler Error Messages, in the SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User’s Manual.

After generating information messages, the compiler performs an error recovery and generates an object program. Check that the error recovery performed by the compiler conforms with the aims of the program.

11.1.31 Comment Coding

Question

(1) How can I nest comments?

(2) How can I code C++ comments in a C language program?

Answer

(1) There is an option that allows you to nest comments without generating an error. In this case, note that these comments are interpreted as described below.

[Specification method]
Dialog menu: C/C++ tab Category: [Other] Miscellaneous options: Allow comment nest
Command line**: *comment***

Table 11.13 Nest of comment.

C/C++ Source Code	Nested Comments Not Allowed	Nested Comments Allowed
/* comment */	Recognized as a comment statement	Recognized as a comment statement
/* /* comment */ */	Coding error	Recognized as a comment statement
/* /* /* comment */	Recognized as a comment statement	Coding error

(2) The C++ comment code “//” can be used. There is the following relationship between the “//” and the C comment code (/* */). The parts that can be recognized as comments are underlined:

```
void func()
{
    abc=0;        //  /* comment  */

    def=1;        /* comment
    ghi=2;        // comment        */
}
```

←Code after // is recognized as a comment

←Code enclosed in /**/ is recognized as a comment

11.1.32 How to Build Programs When the Assembler Is Embedded

Question

A warning message is output at compiling when the assembler intrinsic is performed using #pragma inline_asm.

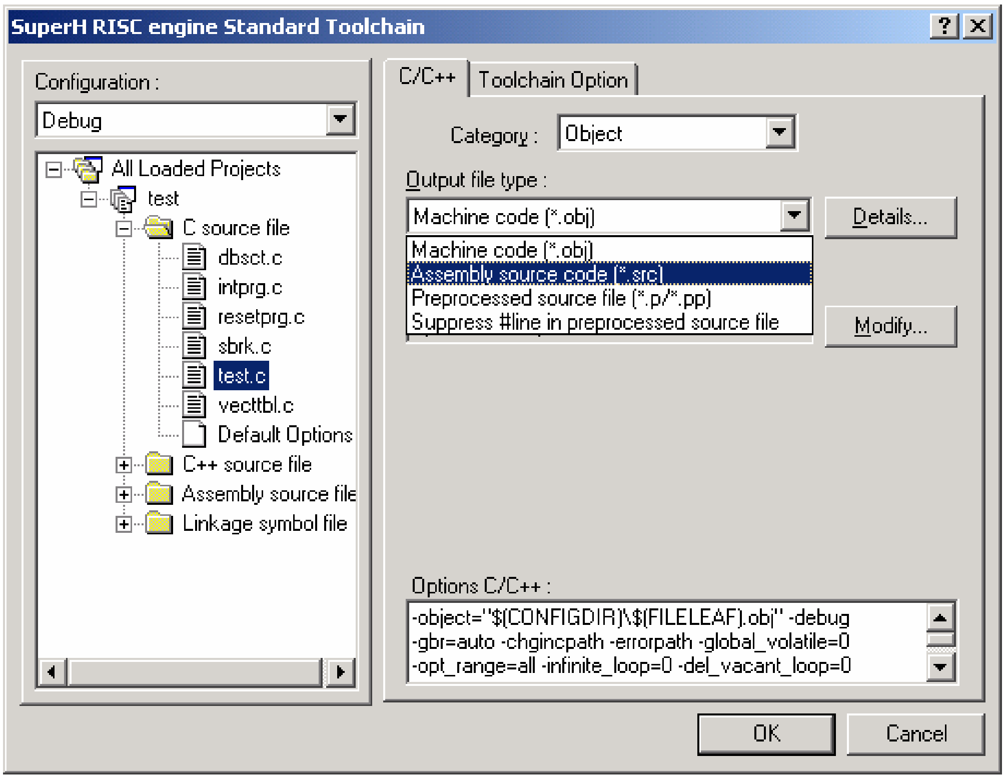
Answer

Assembler embedded files should be output in the Assembly language and then be assembled.

To build a file on the HEW, specify the file containing the Assembler embedding to the Assembly output, How to Specify Options for Each File. When built in this manner, the file that has been Assembly output will automatically be assembled.

In the following example, the file test.c containing an Assembly embedding is specified:

<HEW2.0 or later>



SuperH RISC engine Standard Toolchain

Configuration : Debug

Category : Object

Output file type : Machine code (*.obj)

Machine code (*.obj)
Assembly source code (*.src)
Preprocessed source file (*.p/*.pp)
Suppress #line in preprocessed source file

Options C/C++ :

```
-object="$(CONFIGDIR)\$(FILELEAF).obj" -debug
-gbr=auto -chgincpath -errorpath -global_volatile=0
-opt_range=all -infinite_loop=0 -del_vacant_loop=0
```

Figure 11.3 Compiler Dialog Box

Select Assembly source code (*.src) from C/C++ Tab Category: [Object] Output file type: .

Files are built normally with this specification.

Note that this specification disables C source debugging.

11.1.33 C++ Language Specifications

Question

Are there any function supporting the development of programs in the C++ language?

Answer

The SuperH RISC engine C/C++ compiler supports the following functions to support program development in C++:

(1) Support of EC++ class libraries

As EC++ class libraries are supported, the intrinsic C++ class libraries can be used from a C++ program without any specification.

The following four-type libraries are supported:

- Stream I/O class library
- Memory manipulation library
- Complex number calculation class library
- Character string manipulation class library

For details, refer to section 10.4.2, EC++ Class Libraries, in the SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User’s Manual.

(2) EC++ language specification syntax check function

Syntaxes are checked on C++ programs, based upon the EC++ language specifications, using a compiler option.

[Specification method]
Dialog menu: **C/C++** Tab **Category: [Other] Miscellaneous options: Check against EC++ language specification**
Command line: ***ecpp***

(3) Other functions

The following functions are supported for efficient coding of C++ programs:

<Better C functions>

- Inline expansion of functions
- Customization of operators such as +, -,<<
- Simplification of names through the use of multiple definition functions
- Simple coding of comments

<Object-oriented functions>

- Classes
- Constructors
- Virtual functions

For a description of how to set the execution environment at using library functions in a C++ program, refer to section 9.2.2(4), C/C++ library function initial settings(_INILIB), in the SuperH RISC engine C/C++ Compiler, Assembler, Optimizing Linkage Editor User's Manual.

11.1.34 How to View Source Programs after Pre-Processor Expansion

Question

How can I review a program after macros are expanded?

Answer

The output of the source program expanded by the Pre-Processor is specified with the compiler option.

If the source program before expansion was a C language program, it is output with the extension <filename>.p. For a C++ program, the extension is <filename>.pp.

In this case, no object program is created. Therefore, any optimization option specifications are not available.

[Specification method]
Dialog menu: **C/C++** Tab **Category: [Object] Output file type: Preprocessed source file (*.p/*.pp)**
Command line: ***preprocessor***

11.1.35 The Program Runs Correctly on the ICE But Fails When Installed on a Real Chip

Question

The program runs correctly at debugging on the ICE but fails when operated on a real chip.

Answer

If a program contains the initialization data area (D section), it uses emulation memory on the ICE. Therefore, read/write operation can be performed on the ICE, however, only read operation can be performed on a real chip because memory on a real chip is ROM. This causes the malfunction of the program execution whenever a write operation is attempted.

The initialization data area should be copied from the ROM area to the RAM area at the power-on reset.

Secure an area for each of ROM and RAM using the ROM implementation support option of the HEW2.0 or later optimizing linkage editor and the HEW1.2 inter-module optimizer.

For a description of how to copy data from a ROM area to a RAM area, refer to section 2.3.4, Creating the Initialization Part.

11.1.36 How to Use C language Programs Developed for H8 Microcomputers

Question

What points should I confirm when using a C language program developed for an H8S, H8/300 microcomputer on an SH microcomputer?

Answer

Be careful on the following points for the program:

(1) int-type data are treated as 4-byte data.

On the H8S, H8/300, int type data are treated as 2-byte data, however, on the SH Family, they are treated as 4-byte data. Confirm that there is not any problem on the range of values.

(2) Some expanded functions cannot be used.

Functions on the SH Family C/C++ compiler and the H8S and H8/300 Series C/C++ compiler are compatible by using the #pragma statement, for example, however there are some differences between them in the expanded functions and specifications.

Note that built-in functions are CPU-specific.

(3) Notes on assembler embedding

Because of differences in architecture, the SH series cannot handle any code in which an H8S, H8/300 Series assembly source is embedded.

If you wish to use C source files created in the M32R development environment in the SH development environment, Translation Helper is available.

This is a support tool to translate smoothly the all C source files created in the M32R development environment to the SH development environment.

Translation Helper can be free downloaded from Renesas Development Environment site.

11.1.37 Optimizations That Cause Infinite Loops

Question

Why do infinite loops occur when I upgrade the compiler, or turn optimization on?

Answer

Infinite loops may occur due to compiler optimization, such as in the following common source, in which substitution for a is read from the register instead of from memory, preventing the value of *d from being reflected when changed via interrupt. This optimization is part of the compiler specification, and can be prevented by using the volatile-type specifier.

Example

C source

```
void f( int *d)
{
        int a;
        do
        {
                a=*d;
        }while(a!=0);
}
```

Assembler source with optimization

```
_f:                                ; function: f
                                   ; frame size=0
          .STACK      _f=0
          MOV.L       @R4,R2
L11:
          TST         R2,R2        ; not read from memory
          BF          L11
          RTS
          NOP
          .END
```

Modified C source

```
void f( volatile int *d)
{
        int a;
        do
        {
                a=*d;
        }while(a!=0);
}
```

Modified assembler source with optimization

```
_f:                                ; function: f
                                   ; frame size=0
          .STACK      _f=0
L10:
          MOV.L       @R4,R2     ; read from memory
          TST         R2,R2
          BF          L10
          RTS
          NOP
.END
```

11.1.38 Precautions Regarding the DSP Library

Question

When using the DSP library, I sometimes experience abnormal termination, and don't get the results I expect. Are there any precautions regarding DSP library usage of which I should be aware?

Answer

Check the following:

1. Memory corruption

Since the DSP library uses heap memory, if the memory has been corrupted, proper calculation results cannot be obtained.

Heap memory corruption can also lead to abnormal operation.

2. Proper usage of DSP memory (X and Y memory)

There are some DSP library functions that require input and output data to be placed in X/Y memory. For such a DSP library function, allocate sections containing input and output data to X/Y memory, according to the description of the function. You can use #pragma section to separate sections on a finer level. For details, see section 3.7.2, Section Switching.

Also, when using the "-dspc" option, you can use the X/Y memory modifier to easily separate X/Y memory sections.

Note that the workspace must be allocated to Y-RAM when the filter function is used. If the "-dspc" option is not specified, allocate the DY and BY sections to Y-RAM during linkage. If the "-dspc" option is specified, allocate the $YD and $YB sections to Y-RAM during linkage.

3. Usage methods for DSP library functions

When DSP library functions are used, special pre- or post-processing may be needed.

Check that the corresponding library functions, as well as the pre- and post-processing, are being used properly.

For details about how to use each library function, see section 3.13, DSP Library.

4. Scaling errors

Since the DSP library functions perform scaling processing, such processing may cause errors to occur.

For details about scaling, see section 3.13, DSP Library.

11.1.39 Maximum Sampling Data Count for a DSP Library Function

Question

What is the maximum sampling data count for a DSP library function?

Answer

The maximum sampling data count for a DSP library function depends largely on two things: DSP memory (X and Y memory) capacity, and whether the function is an in-place or non-in-place function.

For the in-place function:

maximum-sampling-count = *x-or-y-memory-size* / 2 (short type size)

For the not-in-place functions, the maximum sampling count shall be half the result of the calculation, because the input and output areas need to be separate.

When the X-RAM and Y-RAM are 8K:

- `FftComplex`

  Maximum sampling count: 2048

  Size of heap used: 17334

- `FftReal`

  - When input data is placed outside of X/Y memory

    Maximum sampling count: 4096

    Size of heap used: 18358

  - When input data is placed in X/Y memory

    Maximum sampling count: 2048

    Size of heap used: 17334

- `IfftComplex`

  Maximum sampling count: 2048

  Size of heap used: 17334

- `IfftReal`

  Maximum sampling count: 2048

  Size of heap used: 19382 (17334 + 2048)

  (this is because even in `IfftReal()`, `malloc` is used to allocate the area.)

- `FftInComplex`

  Maximum sampling count: 4096

  Size of heap used: 18358

- `FftInReal`

  Maximum sampling count: 4096

  Size of heap used: 18358

- `IfftInComplex`

  Maximum sampling count: 4096

  Size of heap used: 18358

- `IfftInReal`

  Maximum sampling count: 4096

  Size of heap used: 18358

11.1.40 Read/write Instructions for Bit Fields

Question

```
struct bit{
 unsigned short int b0 : 1;
 unsigned short int b1 : 1;
 unsigned short int b2 : 1;
 unsigned short int b3 : 1;
 unsigned short int b4 : 1;
 unsigned short int b5 : 1;
 unsigned short int b6 : 1;
 unsigned short int b7 : 1;
 unsigned short int b8 : 1;
 unsigned short int b9 : 1;
 unsigned short int b10 : 1;
 unsigned short int b11 : 1;
 unsigned short int b12 : 1;
 unsigned short int b13 : 1;
 unsigned short int b14 : 1;
 unsigned short int b15 : 1;
}  ;
```

In the above code, I'd like to define a bit field, and access the bits of a specific register for a 16 bit width, but I end up performing access by byte and bit operation instruction. For registers that can only be accessed for 16 bits, when a byte access or bit operation instruction is generated, I can't properly read the register value. What should I do?

Answer

As long as there are no particular specifications in the program, bit field members are accessed by compiler-optimized instructions. SH-2A and SH2A-FPU generate bit access instructions, while other CPUs generate byte access instructions. As a result, access may be performed by unintended instructions. Specify volatile to perform access using the type set for the member variable.

To prevent changes to access methods and multiple accesses by the compiler, specify volatile explicitly for variables for which you would like to prevent such changes.

C source without volatile

```
struct bit reg;

void f()
{
        reg.b6=1;
}
```

Assembler source without volatile
(other than SH-2A and SH2A-FPU)

```
_f:                       ; function: f
                         ; frame size=0
          .STACK      _f=0
          MOV.L       L11+2,R6   ; _reg
          MOV.B       @R6,R0
          OR          #2,R0
          RTS
          MOV.B       R0,@R6
```

Assembler source without volatile
(for SH-2A SH2A-FPU)

```
    f:             ;function:f
   .STACK          _f=0
    MOV.L          L11,R2    ; reg
    BSET.B         #1,@(0,R2)
    RTS/N
```

C source with volatile

```
volatile struct bit reg;

void f()
{
        reg.b6=1;
}
```

Assembler source with volatile
(other than SH-2A and SH2A-FPU)

```
_f:                             ; function: f
                               ; framesize=0
          .STACK      _f=0
          MOV.L       L11+2,R6   ; _reg
          MOV         #2,R5      ; H'00000002
          MOV.W       @R6,R2
          SHLL8       R5
          OR          R5,R2
          RTS
          MOV.W       R2,@R6
```

Assembler source with volatile
(for SH-2A SH2A-FPU)

```
    f:                 ;function: f
                      ;framesize=0;
   .STACK    _f=0
   MOV.L     L11+2,R6;reg
   MOV.W     @R6,R2
   MOVI20    #512,R5  ;H’00000200
   OR        R5,R2
   RTS
   MOV.W     R2,@R6
```

Note that bit fields whose type is long long are always accessed using run-time routines.

C source

```
struct bit{
 unsigned long long int b0 : 1;
 unsigned long long int b1 : 1;
 unsigned long long int b2 : 1;
 unsigned long long int b3 : 1;
 unsigned long long int b4 : 1;
 unsigned long long int b5 : 1;
 unsigned long long int b6 : 1;
 unsigned long long int b7 : 1;
};

struct bit reg;

void f()
{
        reg.b6=1;
}
```

Assembler source

```
_f:                             ; function: f
                                ; frame size=12
          .STACK      _f=12
          STS.L       PR,@-R15
          MOV         #1,R1     ; H'00000001
          MOV.L       R1,@-R15
          MOV         #0,R4     ; H'00000000
          MOV.L       R4,@-R15
          MOV.L       L11+4,R1  ; _reg
          MOV.L       L11+8,R5  ; __bfs64u_p
          MOV.W       L11,R0    ; H'0601
          JSR         @R5
          MOV         R15,R2
          ADD         #8,R15
          LDS.L       @R15+,PR
          RTS
           NOP
```

11.1.41 Specifying Interrupt Processing

Question

I want to specify interrupt processing. What should I do?

Answer

To specify interrupt processing, be sure to first check the vector table definition when setting up a HEW project. Since a file containing the template for the interrupt processing function is generated, edit this file. Note also the interrupt processing format for SH-1 and SH-2 is different than that for SH-3 and SH-4, and the files generated by HEW are different.

- For SH-1 and SH-2

Interrupt processing requires: 1) the interrupt processing function, 2) the vector table, and 3) initialization of the interrupt mask bit for the status register. As an example for SH-1 and SH-2, processing for the IRQ0 interrupt cause is specified in the SH7020 project.

1. Interrupt processing function

HEW comes with an empty interrupt processing function. The intprg.c file contains a definition for void INT_IRQ0(void). You can use this to specify IRQ0 processing. Note that #pragma interrupt needs to be specified for the interrupt processing function. This is performed using vect.h, which does not need to be changed.

```
//intprg.c
// 64 Interrupt IRQ0
void INT_IRQ0(void)
{
/* Specify processing here */
}
```

```
// vect.h
// 64 Interrupt IRQ0
#pragma interrupt INT_IRQ0
extern void INT_IRQ0(void);
```

2. Vector table

This can be used as generated by HEW, and does not need to be edited. The vector table is void *INT_Vectors[], in vecttbl.c. With SH-1 and SH-2, when an interrupt occurs, control moves to one of the functions registered in the vector table. The vector number for IRQ0 is 64, as can be confirmed by checking the hardware documentation. When an interrupt occurs due to the IRQ0 interrupt cause, a function in INT_Vectors[60] is called (60 = 64 - 4). Since a function named INT_IRQ0 is registered in INT_Vectors[60], INT_IRQ0 is executed when an interrupt is caused by IRQ0.

```
void *INT_Vectors[] = {
// 4 Illegal code
    (void*) INT_Illegal_code,
...
// 64 Interrupt IRQ0
    (void*) INT_IRQ0,
...
};
```

3. Initializing the interrupt mask bit of the status register

For interrupt processing to be used, the interrupt mask bit of the status register must be properly initialized. In resetprg.c, set SR_Init to an appropriate value from 0x000000F0. In PowerON_Reset_PC, set_cr can be used to set the interrupt mask bit of the status register.

```
#define SR_Init    0x000000F0
```

- For SH-3 and SH-4

For SH-3 and SH-4, interrupt processing also requires: 1) the interrupt processing function, 2) the vector table, and 3) initialization of the interrupt mask bit for the status register. As an example for SH-3 and SH-4, processing for the IRQ0 interrupt cause is specified in the SH7705 project.

1. Interrupt processing function

Since HEW comes with an empty interrupt function, delete it and define a new function. Because _INT_IRQ0 is defined in intprg.src, delete it, as well as the .global INT_IRQ0 specification in vect.inc. Then, use the C language as usual to define void INT_IRQ0(void). You do not need to specify #pragma interrupt.

```
;intprg.src
...
;H'5E0 H-UDI
_INT_H_UDI
;H'600 IRQ0   ; Delete this
_INT_IRQ0     ; Delete this
;H'620 IRQ1
_INT_IRQ1
...
```

```
;vect.h
...
;H'5E0 H-UDI
.global         _INT_H_UDI
;H'600 IRQ0                 ; Delete this
.global         _INT_IRQ0   ; Delete this
;H'620 IRQ1
.global         _INT_IRQ1
...
```

2. Vector table

This can be used as generated by HEW, and does not need to be edited. The vector table is _INT_Vectors in vecttbl.src. With SH-3 and SH-4, when an interrupt occurs, control moves to IRQHandler in vhandler.src. The address of the interrupt processing routine is calculated from the value of the interrupt event register, and then control moves to the routine. The exception code for IRQ0 is H'600, as can be confirmed by checking the hardware documentation. The offset from INT_Vectors is H'B8 obtained from the expression: {(H'600 - H'40)} / 4. Since the element size of INT_Vectors is 4, INT_IRQ0, the 46th element (H'B6 / 4 = 46) of INT_Vectors, is called as the interrupt routine. IRQHandler processing is as follows:

(1) An exception code is obtained from the interrupt event register.
(2) The INT_Vectors address is obtained.
(3) The address of the interrupt processing routine is calculated.
(4) The interrupt mask is obtained.
(5) The interrupt mask is set in ssr.
(6) The address of the interrupt processing routine is set in spc.
(7) Jump is performed to the interrupt processing routine, using rte.

```
      .org      H'500
_IRQHandler:
      PUSH_EXP_BASE_REG
;
      mov.l     #INTEVT,r0               ; set event address            -(1)
      mov.l     @r0,r1                   ; set exception code
      mov.l     #_INT_Vectors,r0         ; set vector table address     -(2)
      add       #-(h'40),r1              ; exception code - h'40
      shlr2     r1
      shlr      r1
      mov.l     @(r0,r1),r3              ; set interrupt function addr  -(3)
;
      mov.l     #_INT_MASK,r0            ; interrupt mask table addr
      shlr2     r1
      mov.b     @(r0,r1),r1              ; interrupt mask
      extu.b    r1,r1                                                   -(4)
;
      stc       sr,r0                    ; save sr
      mov.l     #(RBBLclr&IMASKclr),r2   ; RB,BL,mask clear data
      and       r2,r0                    ; clear mask data
      or        r1,r0                    ; set interrupt mask
      ldc       r0,ssr                   ; set current status           -(5)
;
      ldc.l     r3,spc                                                  -(6)
      mov.l     #__int_term,r0           ; set interrupt terminate
      lds       r0,pr
;
      rte                                                               -(7)
      nop
;
      .pool
      .end
```

3. Initializing the interrupt mask bit of the status register

Like SH-1 and SH-2, the interrupt mask bit of the status register must be properly initialized for SH-3 and SH-4. In resetprg.c, set SR_Init to an appropriate value from 0x000000F0. In PowerON_Reset_PC, set_cr can be used to set the interrupt mask bit of the status register.

```
#define SR_Init    0x000000F0
```

11.1.42 Common Invalid Instruction Exceptions That Occur When Programs Are Run for an Extended Period of Time

Question

Once the device has been running for 10 minutes to 2 hours, a common invalid instruction exception occurs, and a reset is necessary. Is there some way to analyze from where the problem is occurring?

Answer

Ultimately, this means that a common invalid instruction is occurring, but the system may lose control and cause a common invalid instruction exception due to the following reasons. If the system loses control after an extended period of operation, (2) is very likely.

(1) An unintended interrupt is being performed.
(2) A stack overflow is corrupting valid RAM data.
(3) A problem exists with the board environment (such as a data conflict or memory software error).

To find the cause of the problem, perform the following and operate the device:

- Enable instruction tracing.
- Set breakpoints for the interrupt function jumped to during the common invalid instruction exception.

Once the device is operating and the common invalid instruction exception occurs, processing will stop at the breakpoint set for the interrupt function. When this occurs, analyze the status of the instruction trace, and determine the cause of the problem.

Use the following analysis method when a stack overflow is causing the problem:

- Set read/write break access for the address immediately before the address of the start of the stack area.

Once the device is operating and an access occurs that overflows the stack, processing will stop at the breakpoint set above. When this occurs, if the access instruction is a stack access instruction, the cause of the problem is most likely a stack overflow.

11.1.43 When the Result of an Integer Calculation Differs from the Expected Value

Question

Sometimes, when the results of an integer multiplication are substituted for a variable of the long long type, an unexpected value is returned.

When I change 60000*70000 to 60000*30000, the correct value is obtained.

Why is an incorrect value obtained when the results of multiplication exceed the int value, even when substitution is performed to a long long type variable?

Example1:

```
long long l_max;
        :
        l_max=60000*70000;
```

Example2:

```
long long l_max;
        :
int test=70000;
        :
        l_max=60000*test;
```

Answer

Even when the variable substituted is of the long long type, the calculated integer is handled as an int type (4 bytes) when specified as a constant.

As such, 60000*70000 becomes 0xFA56EA00 during multiplication, but when substitution to the long long type is performed, sign extension occurs, and it becomes 0xFFFFFFFFFA56EA00.

Since 60000*30000 becomes 0x6B49D200, no sign extension occurs, it becomes 0x000000006B49D200, and the proper value is obtained.

To obtain the expected calculation results, you need to specify LL after the constant value so that the compiler explicitly recognizes that the value is a long long type.

Example1:

```
long long l_max;
       :
    l_max = 60000LL * 70000LL;    // Specify LL after one or both constants.
```

Example2:

```
long long l_max;
       :
int test=70000;
       :
    l_max = 60000LL * test;    // Specify LL after constants.
```

11.2 Linkage Editor

11.2.1 An "Undefined symbol" Message Appears on Linking

Question

At linkage, an "undefined symbol" message appears. Why is this?

What does it mean?

Answer

Please check to make sure that libraries are being linked. Also check whether functions which have been declared or used actually exist in code. For details, refer to section 3.15.2 (2), Important Information on Linking.

11.2.2 A "RELOCATION SIZE OVERFLOW" Message Appears at Linkage

Question

On linking, I receive a "RELOCATION SIZE OVERFLOW" warning message. Also, how do I go about checking for missing section address specifications?

Answer

Check whether limits are not exceeded by the specifications #pragma abs16, #pragma gbr_base, or #pragma gbr_base1.

Section addresses are specified by section name using the START command; sections for which no specification is made are placed after the last section for which an address is specified.

Such errors in programming tend to occur particularly frequently when there are numerous section names.

If there are sections not specified by the START command, this command causes a warning to be output.

(1) Message example

The following is an example of options and message output by the linkage editor.

```
input sample.obj
input low/__main.obj
input low/__exit.obj
library lib/shclib.lib
library low/shclow.lib
output sample.abs
form a
entry _$main
start C,B,D,P/0400                        (specifications for $G0 and $G1 are missing)
;start C,B,D,$G0,$G1,P(0400)              (parameter specification for normal termination)
```

```
** L1120 (W) Section address is not assigned to "$G0"
** L1120 (W) Section address is not assigned to "$G1"

LINKAGE EDITOR COMPLETED
```

Warning messages are output because the $G0 and $G1 section names are not defined.

11.2.3 A "SECTION ATTRIBUTE MISMATCH" Message Appears at Linkage

Question

On linking, a "SECTION ATTRIBUTE MISMATCH" warning message appears. What should I do about it?

Answer

This error may be caused by any of the following.

(1) Different alignments are specified for the same section.

Check whether different alignments have not been specified for the same section name.

(2) An attempt is made to link an object compiled with the "-cpu=sh4" option and an object compiled using a different cpu option.

On compiling using the cpu=sh4 option*, each section is unconditionally set to aligndata8. Consequently alignment is different with objects compiled using other cpu options. In such cases also, the ALIGN_SECTION option/subcommand of the linkage editor can be used to avoid this.

(3) The modification shown in Answer 2 in 11.2.4, Transfer to RAM and Execution of a Program, meets all of the following conditions. Note that you can ignore any warnings output.

(a) The name of the program section P was changed by using the section option of the C/C++ compiler or other means.

(b) The section in (a) above is specified as the transfer source section.

Note: * On compiling using the cpu=sh4 option, each section is unconditionally set to aligndata8 (Ver.5 or lower).
Memory areas may increase between sections as a result of the eight-byte alignment.

11.2.4 Transfer to RAM and Execution of a Program

Question

I want to transfer my program to RAM, from which execution is faster; how do I proceed?

<Operating environment>

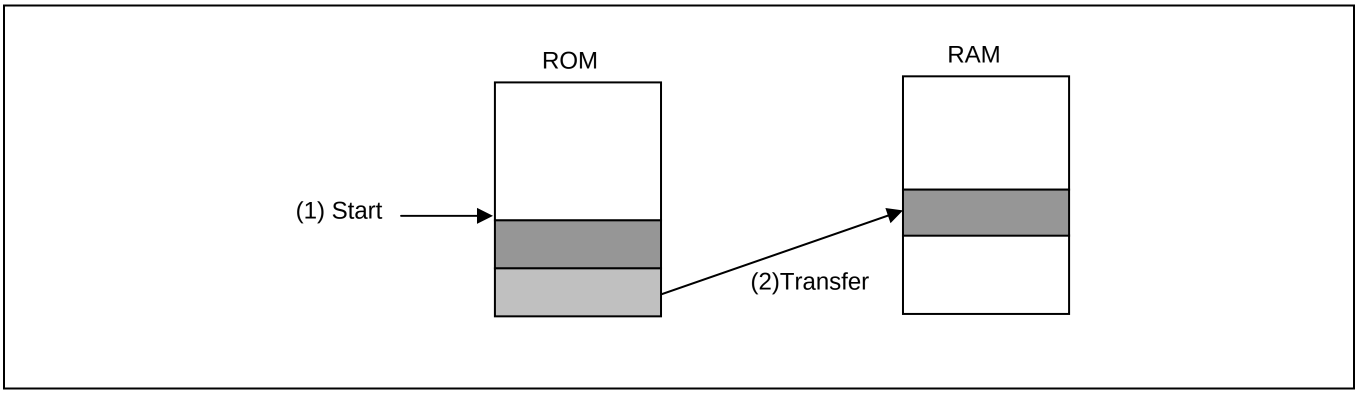


Figure 11.4 Transferring a Program from ROM to RAM

<Details>

(1) Start the program resident in ROM.

(2) Transfer some of the sections of the program's own code to RAM.

Answer1

When the program code must be copied to a fixed address in RAM, as with initialization data, the ROM support function of the linker can be used to execute the program from RAM (at linkage, address resolution occurs, and so it is not possible to determine the address in RAM and copy the program code at runtime).

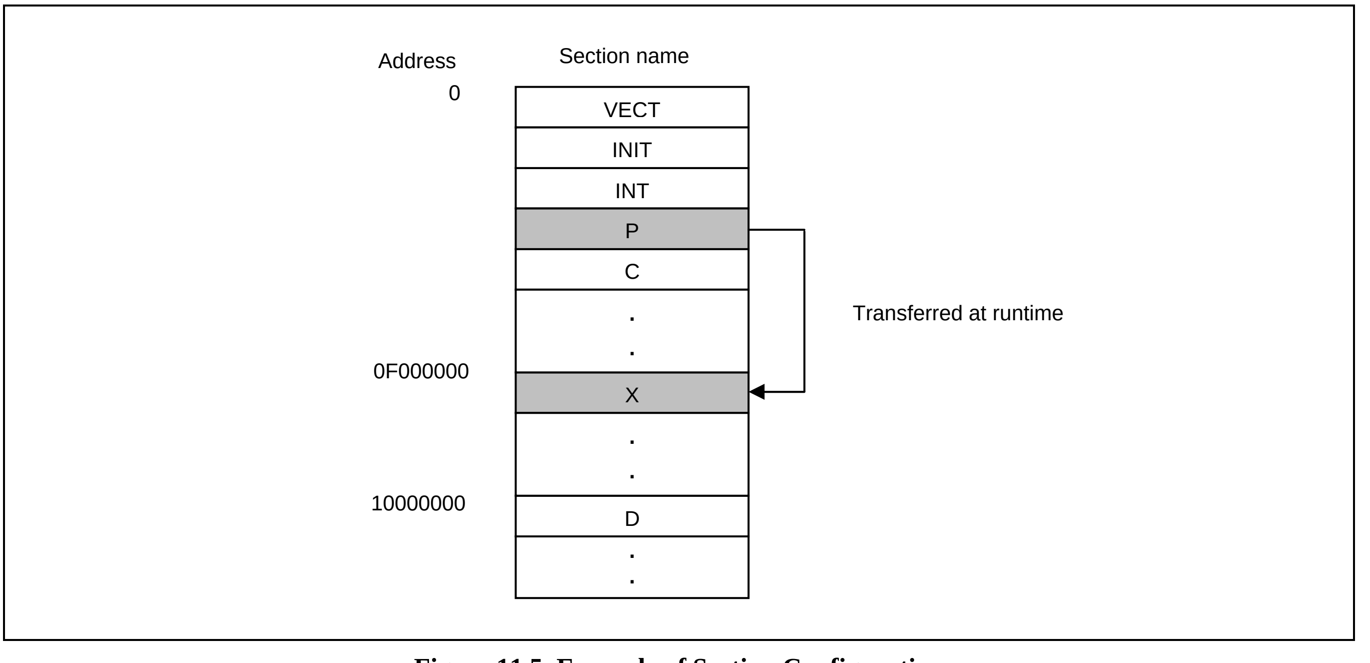


Figure 11.5 Example of Section Configuration

An example of a program with section configuration as in figure 11.5 appears below.

C language part

```
/******************************************************************/
/*                     file name "init.c"                         */
/*----------------------------------------------------------------*/
/*      Compile option initializes program section names          */
/******************************************************************/
#include "sample.h"  /* Includes the sample.h file of  section 2       */
extern int *_B_BGN,*_B_END;
extern int *_P_BGN;  /* Start address of section P                     */
extern int *_X_BGN;  /* Start address of section X                     */
extern int *_X_END;  /* End address of section X                       */
extern void _INITSCT(void);
extern void _INIT();
extern void main();

void _INIT()
{
        _INITSCT();
        main();
        for ( ; ; )
            ;
}

void _INITSCT(void)
{
        int *p,*q;

        for ( p = _B_BGN; p < _B_END; p++ )
            *p = 0;

        /* copy from section P to section X */
        for ( p = _X_BGN, q = _P_BGN; p < _X_END; p++, q++ )
            *p = *q;
}

/****************************************************************/
/*                     file name "main.c"                       */
/*--------------------------------------------------------------*/
/*            Program section name is "P" by default            */
/****************************************************************/
int a = 1;
int b;
```

```
const int c = 100;
void main(void)
{
        /* this routine is executed from the copy destination (RAM) */
        for ( ; ; )
            ;
}

/******************************************************************/
/*                       file name "int.c"                        */
/******************************************************************/
#include "sample.h"        /* Includes the sample.h file of section 2  */
#include "7032.h"          /* Includes the 7032.h file of section 2    */
extern int a;              /* section D code                           */
extern int b;              /* section B code                           */
extern const int c;        /* section C code                           */
#pragma interrupt(IRQ0, inv_inst)

/******************************************************************/
/*                     interrupt module IRQ0                      */
/******************************************************************/
extern void IRQ0(void)
{
        a = PB.DR.WORD;
        PC.DR.BYTE = c;
}




/******************************************************************/
/*                   interrupt module inv_inst                    */
/******************************************************************/
extern void inv_inst(void)
{
        return;
}
```

Assembly language code part

```
;****************************************************************
;*                    file name "sct.src"                       *
;****************************************************************
                .SECTION        P,CODE,ALIGN=4
                .SECTION        X,CODE,ALIGN=4
                .SECTION        B,DATA,ALIGN=4
                .SECTION        C,DATA,ALIGN=4

__P_BGN:     .DATA.L (STARTOF P)                ; Start address of section P
__X_BGN:     .DATA.L (STARTOF X)                ; Start address in RAM of section P
__X_END:     .DATA.L (STARTOF X)+(SIZEOF X)     ; End address in RAM of section P
__B_BGN:     .DATA.L (STARTOF B)                ; Start address of section BBS
__B_END:     .DATA.L (STARTOF B)+(SIZEOF B)     ; End address of section BBS

                .EXPORT __P_BGN
                .EXPORT __X_BGN
                .EXPORT __X_END
                .EXPORT __B_BGN
                .EXPORT __B_END
                .END


;****************************************************************
;*                    file name "vect.src"                      *
;****************************************************************
        .SECTION        VECT,DATA,ALIGN=4

        .IMPORT __INIT
        .IMPORT _inv_inst
        .IMPORT _IRQ0

        .DATA.L __INIT
        .DATA.L H'FFFFFFC
        .ORG     H'0080
        .DATA.L _inv_inst
        .ORG     H'0100
        .DATA.L _IRQ0
        .END
```

Command line commands are as follows.

Command specifications

shcΔ-debugΔ-section=P=INITΔinit.c
shcΔ-debugΔ-section=P=INTΔint.c
shcΔ-debugΔmain.c
asmshΔsct.srcΔ-debug
asmshΔvect.srcΔ-debug
optlnkΔ-nooptimizeΔ-sub=rom.sub

Linker option file

```
;*****************************************************************
;*                    file name "rom.sub"                        *
;*****************************************************************
sdebug
input vect, sct, init, int, main
ROM (P,X)                  ;  Address resolved so that section P is assigned to X
start VECT/0,INIT,INT,P,C,D/10000000,X/0f000000
                           ;  VECT, INIT, INT, P, C, D are in ROM, X is in RAM
output sample.abs
list sample.map
exit
```

By means of the above code, the program of section P is copied to section X and executed.

The section INIT is the routine which performs the copying, and so must be separate from the routine to be copied. Here the main program (section P) is run from the copy destination.

Answer 2

With HEW version 2.0 or later, you can use the ROM support function of the optimization linkage editor to ease copying a program section during execution to a fixed address in RAM (decided during linkage), and execute the program from RAM.

First, to transfer the program section to be executed from RAM during startup, specify the address of the section. This processing is added to the dbsct.c file generated by HEW. At this point, the code in the PXX section is transferred to the XX section. Add specifications as follows.

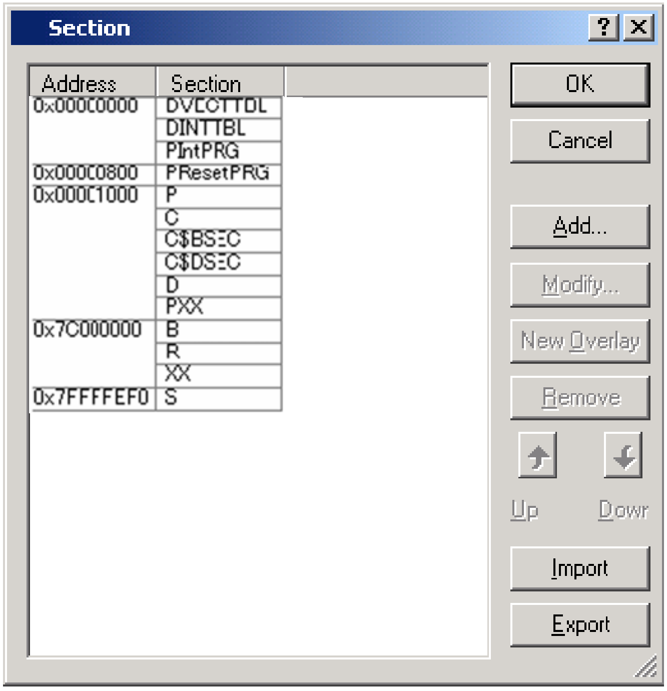
```
#pragma section $DSEC
static const struct {
             char *rom_s;       /* Start address in ROM of the initialized data section */
             char *rom_e;       /* End address in ROM of the initialized data section */
             char *ram_s;       /* Start address in RAM of the initialized data section */
}DTBL[]= {{__sectop("D"), __secend("D"), __sectop("R")},
         {__sectop("PXX"),__secend("PXX"),__sectop("XX")}};
#pragma section $BSEC
static const struct {
             char *b_s;         /* Start address of the uninitialized data section */
             char *b_e;         /* End address of the uninitialized data section */
}BTBL[]= {__sectop("B"), __secend("B")};
```

The above are settings for the PXX section and XX section

After this processing is performed, at startup a copy is sent from the PXX section to the XX section.

Use the optimization linkage editor to specify the start address of the transfer destination section XX.

Choose [Build -> SuperH RISC engine Standard Toolchain... -> Optimization Linker]. On the opened page, select a category section, and click the [Edit] button to display a dialog box for the section settings.



Section

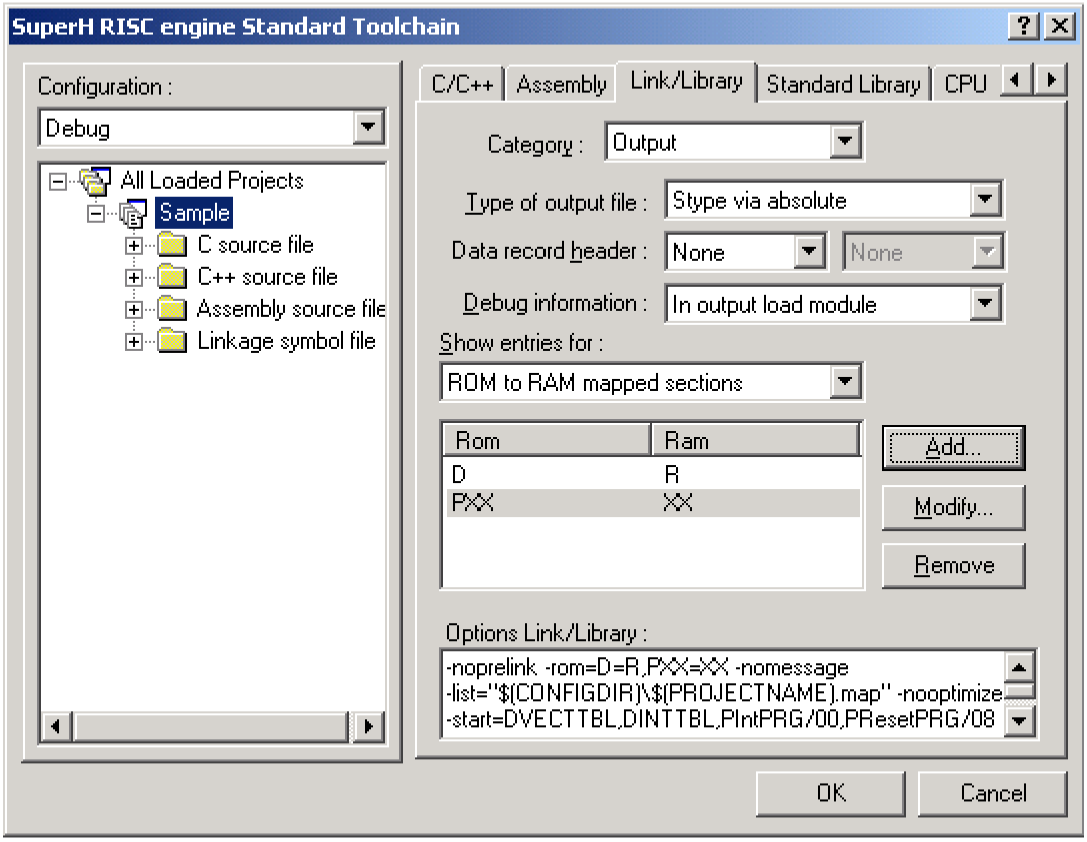
Address	Section
0x000C0000	DVECTTBL
	DINTTBL
	PIntPRG
0x000C0800	PResetPRG
0x000C1000	P
	C
	C$BSEC
	C$DSEC
	D
	PXX
0x7C000000	B
	R
	XX
0x7FFFFEF0	S

Figure 11.6 Section Settings Dialog Box

Here, set up the PXX section and XX section.

Choose [Build -> SuperH RISC engine Standard Toolchain -> Optimization linker]. On the opened page, select [Output] from [Category] and [Sections for mapping from ROM to RAM] from [Option item] to set up the mapping from PXX to XX.

With these settings, the program can be executed from RAM.



Configuration : Debug

Category : Output

Type of output file : Stype via absolute

Data record header : None None

Debug information : In output load module

Show entries for : ROM to RAM mapped sections

Rom	Ram
D	R
PXX	XX

Options Link/Library :

```
-noprelink -rom=D=R,PXX=XX -nomessage
-list="$(CONFIGDIR)\$(PROJECTNAME).map" -nooptimize
-start=DVECTTBL,DINTTBL,PIntPRG/00,PResetPRG/08
```

Figure 11.7 Optimization Linker Dialog Box

Notes:

In the Ver.9 compiler or later, by selecting the enable_register option, variables with the register storage class specification can be allocated preferentially to the registers. (The enable_register option isn't selected in Ver.9 compiler or later)

With improvements to HEW2.0 or later, messages are no longer output ordinarily, but in the following cases, the same warning message as HEW1.2 (L1323 (W) Section attribute mismatch: "FXX") may be output.

When the above setting is done in HEW 1, 2, the Inter-Module Optimization Tool may output a warning message (1300 SECTION ATTRIBUTE MISMATCH IN ROM OPTION/SUBCOMMAND(XX)).

This reason is that a problem section is specified in the __sectop and __secend operations.
This warning can be ignored.

(1) The name of the program section P was changed by using the section option of the C/C++ compiler or other means.

(2) The section in (1) above is specified as the transfer source section.

11.2.5 Fixing Symbol Addresses in Certain Memory Areas for Linking

Question

After fixing a program in internal ROM, I want to develop the program for external memory, and in future want to update only the external memory program.

Answer

When fixing a program in internal ROM, the link command fsymbol can be used to output a definition file of externally defined labels for the internal ROM.

A definition file is created by the assembler EQU statement, and so when creating an external memory program, this file can be assembled and input to reference a fixed address in ROM.

Example of Use:

Figure 11.8 illustrates an example in which the feature A of a product A is modified to the feature B, to develop the product B. Using this, by resolving the addresses of symbols in shared ROM, the common ROM can be used.

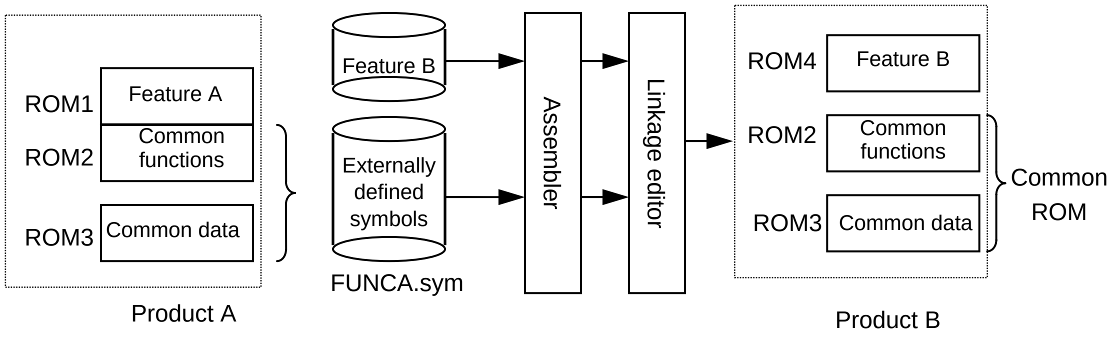


Figure 11.8 Example of Use of the Feature for Output of Symbol Addresses

Example of specification of externally defined symbol file output

optlnkΔROM1,ROM2,ROM3Δ-output=FUNCAΔ-fsymbol=sct2,sct3

The externally defined symbols sct2 and sct3 are output to a file.

Example of file output (FUNCA.sym)

```
;H SERIES LINKAGE EDITOR GENERATED FILE   1997.10.10
;fsymbol = sct2, sct3

;SECTION NAME = sct1
.export sym1
sym1:  .equ     h'00FF0080
.export sym2
sym2:  .equ     h'00FF0100
;SECTION NAME = sct2
```

```
.export sym3
sym3:  .equ     h'00FF0180
    .end
```

Example of specification of assembly and relinking

```
asmshΔROM4
asmshΔFUNCA.sym
optlnkΔROM4,FUNCA
```

The externally referenced symbols in ROM4 can be resolved without linking the object files ROM2, ROM3.

Note: When using this procedure, the symbols within feature A cannot be referenced from common functions.

11.2.6 Using Overlays

Question

I want to use an overlay in my program.

At runtime, I want to transfer a program from ROM to RAM for execution, but I want to execute two or more routines that will not be executed simultaneously at the same RAM address.

Answer

For information on transferring programs from ROM to RAM for execution, refer to section 11.2.4, Transfer to RAM and Execution of a Program.

The essence of the program is as follows, but the following procedure is required.

- Example

The following is an example of transfer of multiple programs or data sets, which do not exist simultaneously, from external ROM to faster internal RAM for execution.

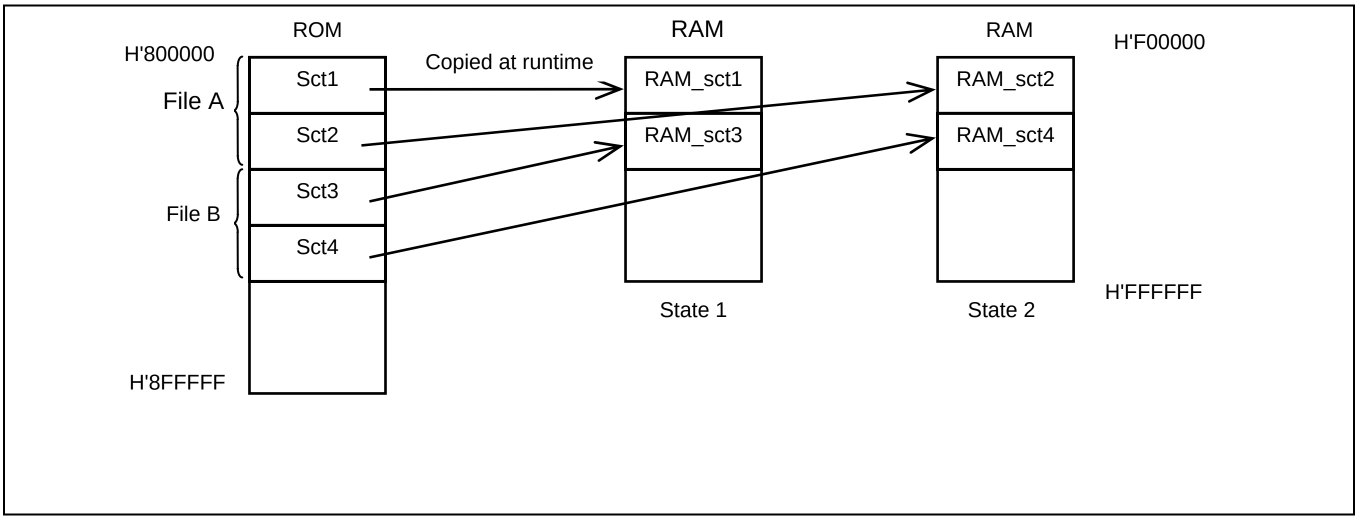


Figure 11.9 Assigning Multiple Sections to the Same Address

Command example

optlnkΔ-subcommand=test.sub

Contents of test.sub

```
INPUT  A,B
ROM    Sct1=RAM_sct1
ROM    Sct3=RAM_sct3
ROM    Sct2=RAM_sct2
ROM    Sct4=RAM_sct4
START  Sct1,Sct2,Sct3,Sct4/800000
START  RAM_sct1,RAM_sct3:RAM_sct2,RAM_sct4/0F00000
```

Explanation

RAM_sct1 and RAM_sct2 are assigned from the same address. RAM_sct3 is concatenated with RAM_sct1, and RAM_sct4 with RAM_sct2.

11.2.7 Specifying Error Output for Undefined Symbols

Question

I want to have an error message output, and prevent output of the load module, if there are undefined symbols at link time.

Answer

The UDFCHECK option should be specified at link time.

By this means, if there are any undefined symbols present, error message 221 will be output and output of the load module will be suppressed.

(If the UDFCHECK option/subcommand is not specified, the warning message 105 is displayed, and the load module is generated.)

In the Linkage Editor Ver7 or later, however, the UDFCHECK option is eliminated and the UDFCHECK is always enabled.

11.2.8 Unify Output Forms of S-Type File

Question

I would like to unify mixed output forms S1, S2, S3 of S type file.

Answer

These can be output by specific data record (S1, S2, S3) irrespective of load address by options.

Example: optlnk test.abs -form=stype -output=test.mot -record=s2 ; All data records are output by S2.

11.2.9 Dividing an Output File

Question

I would like to divide an output file for each ROM devices into some files.

Answer

If specify a start address and termination address in the end of an output file name, an object of specified area can be output. An output file name can be specified more than two.

Example: An area of 0x0-0xFFFF is output into optlnk test.abs -form=stype -output=test1. mot=0-FFFF test2.mot=10000-1FFFF; test1.mot, an area of 0x10000-0x1FFFF is output into test2.mot.

11.2.10 Execution of optlinksh.exe on Windows 2000

Question

If "optlnksh.exe" is executed in Windows2000, [2020 SYNTAX ERROR] is output.

Answer

Please confirm whether there is a space in environment variable SHC_TMP.

It can be operated correctly in shc even SHC_TMP has a space, but an error (2020 SYNTAX ERROR) occurs in optlnksh. Temporarily directory in Windows 2000 is C:\Documents and Settings\foo\Local Settings\Temp (the foo is user name).

11.2.11 Output File Format of Optimizing Linkage Editor

Question

Tell me about the load module file format available to a ROM Programmer.

Answer

The load modules output by the optimizaton linkage editor are shown below:

When creating a load module for a ROM Programmer, output it in the hexdecimal or SType format. In this case, no debugging information is output.

- Optimization linkage editors supporting the C/C++ Compiler V7.1, V8.0 output load modules in the ELF/DWARF2 format at debugging. The load modules created by earlier versions is output in either the SYSROF or ELF/DWARF1 format, and so the format should be changed with the ELF/DWARF format converter to use in the latest version.

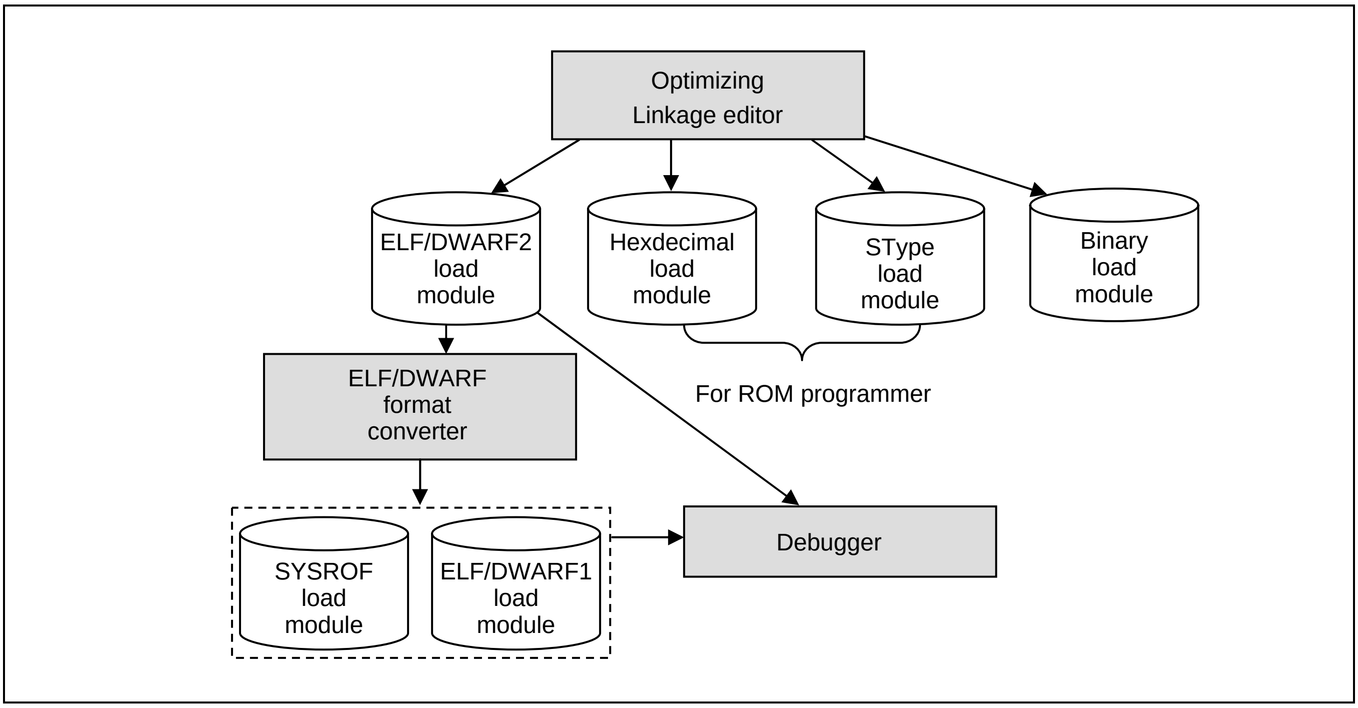


Figure 11.10 Optimizing Linkage Editor Output Load Module

11.2.12 Method for Calculating the Program Sizes (ROM and RAM)

Question

Can you tell me how to measure the ROM and RAM sizes properly?

Answer

You can check them in the list file output by the optimization linkage editor.

Specification method

From the dialog box menu: [Optimization Linker] tab -> [Category]: [List] linkage list output

From the command line:-list=*file-name*

Check method

Specify this option to output the following list file (*.map).

In this case, since the code attribute section is from DVECTTBL, DINTTBL, PIntPRG, PResetPRG, P, C$BSEC, C$DSEC, and D, the ROM size is 0x00006a8.

The RAM area is from B, R and S, and therefore its size is 0x0000052c.

```
*** Mapping List ***

SECTION                            START      END         SIZE   ALIGN

DVECTTBL
                                  00000000  0000000f        10       4
DINTTBL
                                  00000010  000003ff       3f0       4
PIntPRG
                                  00000400  00000557       158       4
PResetPRG
                                  00000800  00000833        34       4
P
                                  00001000  000010db        dc       4
C$BSEC
                                  000010dc  000010e3         8       4
C$DSEC
                                  000010e4  000010ef         c       4
D
                                  000010f0  0000111b        2c       4
B
                                  7c000000  7c0003ff       400       4
R
                                  7c000400  7c00042b        2c       4
S
                                  7c000500  7c0005ff        100      4
```

11.2.13 When Section Alignment Mismatch Is Output

Question

When I input a binary file like the following, and reference the section name of the binary file via a section address operator, the L1322 warning is output. What can I do to avoid this?

[Option specified]

```
binary=project.bin(BIN_SECTION)
```

[C/C++ program]

```
void main(void)
{
    unsigned char *s_ptr;
    s_ptr = __sectop("BIN_SECTION");
    dummy(s_ptr);
}
```

Answer

When the section address operators (__sectop and __second) are used, a section with the size of 0 and with the boundary alignment number of 4 is created in the code generated by the compiler, as shown below.

In this case, a binary section is input, but the boundary alignment number for the entity of the binary section is 1. Since there is more than one boundary alignment number for the same section name, the L1322 warning message is output.

Note that despite this warning message being output, program operation is not affected.

This warning message can be avoided by specifying a boundary alignment number when the binary file is input with the optimization linker.

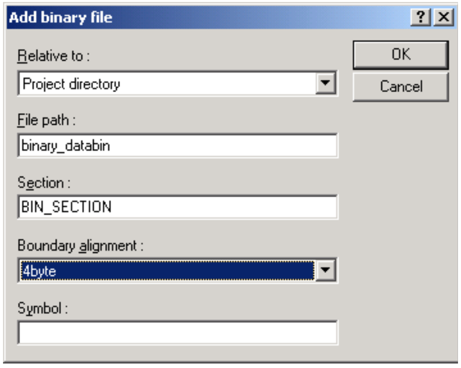
[Code when __sectop is used]

```
_main:                          ; function: main
                                ; frame size=0
          .STACK      _main=0
          MOV.L       L13+2,R4  ; STARTOF BIN_SECTION
          BRA         _dummy
          NOP
…
         .SECTION   . BIN_SECTION,DATA,ALIGN=4      ; Section with the size of 0, and with the
                                                      boundary alignment number of 4
          .END
```

Example of how to avoid the warning

From the dialog box menu: [Optimization Linker] tab -> [Category]: [Input] option item: binary file

From the command line: binary=binary_data.bin(BIN_SECTION:4)



Relative to : Project directory

File path : binary_databin

Section : BIN_SECTION

Boundary alignment : 4byte

Symbol :

Figure 11.11 Add binary file Dialog Box

Remarks

Specification of a boundary alignment number when a binary file is input is supported by the Linkage Editor of version 9 or later.

For details, see section 9.1.1(4) Binary files.

11.3 Standard Library

11.3.1 Reentrant Function and the Standard Library

Question

Are there any precautions of which I need to be aware for making a function reentrant?

Answer

Functions which use global variables are not reentrant.

Moreover, even when a function is created as a reentrant function, if the standard library is used employing the following standard include file, global variables are used and so the function is no longer reentrant.

Below a list of reentrant library functions is indicated. In the table, the _errno variable is set for functions denoted by triangles; if _errno is not referenced within the program, reentrant execution is possible.

You can also make the standard library reentrant. For details about how to make the standard library reentrant, see section 11.3.2, I would like to use reentrant library function in standard library file.

Table 11.14 List of Reentrant Library Functions (1)

Reentrant column O: Reentrant; X: Non-reentrant; Δ: _errno variable set

No.	Standard Include File		Function Name	Reentrant
1	stddef.h	1	offsetof	O
2	assert.h	2	assert	x
3	ctype.h	3	isalnum	O
		4	isalpha	O
		5	iscntrl	O
		6	isdigit	O
		7	isgraph	O
		8	islower	O
		9	isprint	O
		10	ispunct	O
		11	isspace	O
		12	isupper	O
		13	isxdigit	O
		14	tolower	O
		15	toupper	O
4	math.h	18	acos	Δ
		17	asin	Δ
		18	atan	Δ
		19	atan2	Δ
		20	cos	Δ
		21	sin	Δ
		22	tan	Δ
		23	cosh	Δ
		24	sinh	Δ
		25	tanh	Δ
		26	exp	Δ
		27	frexp	Δ
		28	ldexp	Δ
		29	log	Δ
		30	log10	Δ

Table 11.14 List of Reentrant Library Functions (2)

No.	Standard Include File	Function Name		Reentrant
4	math.h	31	modf	Δ
		32	pow	Δ
		33	sqrt	Δ
		34	ceil	Δ
		35	fabs	Δ
		36	floor	Δ
		37	fmod	Δ
5	setjmp.h	38	setjmp	○
		39	longjmp	○
6	stdarg.h	40	va_start	○
		41	va_arg	○
		42	va_end	○
7	stdio.h	43	fclose	x
		44	fflush	x
		45	fopen	x
		46	freopen	x
		47	setbuf	x
		48	setvbuf	x
		49	fprintf	x
		50	fscanf	x
		51	printf	x
		52	scanf	x
		53	sprintf	Δ
		54	sscanf	Δ
		55	vfprintf	x
		56	vprintf	x
		57	vsprintf	Δ
		58	fgetc	x
		59	fgets	x
		60	fputc	x
7	stdio.h	61	fputs	x
		62	getc	x
		63	getchar	x
		64	gets	x
		65	putc	x
		66	putchar	x
		67	puts	x
		68	ungetc	x
		69	fread	x
		70	fwrite	x
		71	fseek	x
		72	ftell	x
		73	rewind	x
		74	clearerr	x
		75	feof	x
		76	ferror	x
		77	perror	x
8	stdlib.h	78	atof	Δ
		79	atoi	Δ
		80	atol	Δ
		81	strtod	Δ
		82	strtol	Δ
		83	rand	x
		84	srand	x
		85	calloc	x
		86	free	x
		87	malloc	x
		88	realloc	x
		89	bsearch	○
		90	qsort	○

Table 11.14 List of Reentrant Library Functions (3)

No.	Standard Include File		Function Name	Reentrant
8	stdlib.h	91	abs	○
		92	div	Δ
		93	labs	○
		94	ldiv	Δ
9	string.h	95	memcpy	○
		96	strcpy	○
		97	strncpy	○
		98	strcat	○
		99	strncat	○
		100	memcmp	○
		101	strcmp	○
		102	strncmp	○
9	string.h	103	memchr	○
		104	strchr	○
		105	strcspn	○
		106	strpbrk	○
		107	strrchr	○
		108	strspn	○
		109	strstr	○
		110	strtok	x
		111	memset	○
		112	strerror	○
		113	strlen	○
		114	memmove	○

11.3.2 I would like to use reentrant library function in standard library file.

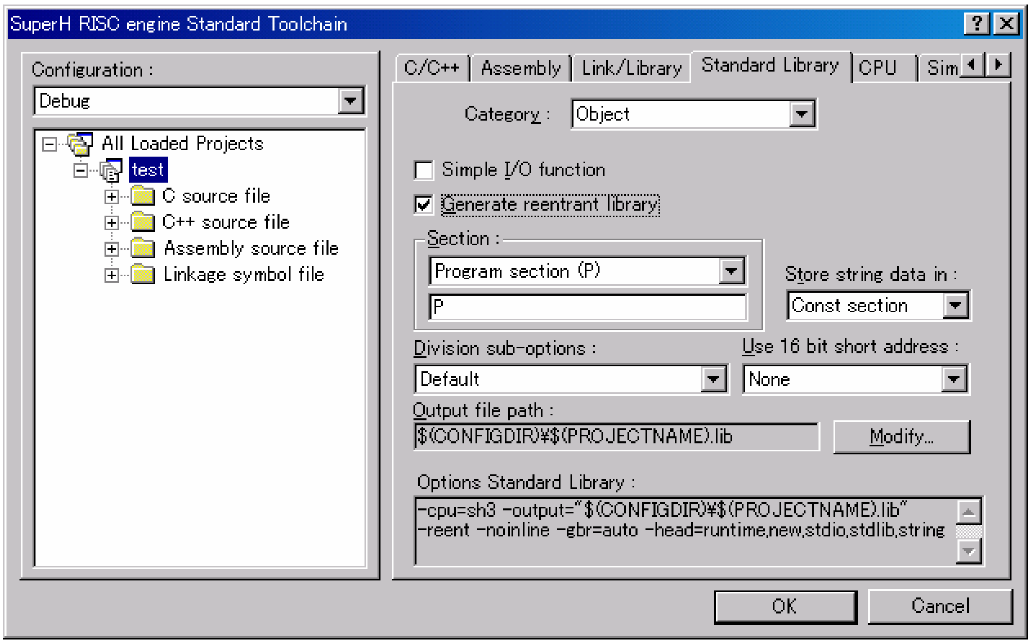
Question

I would like to use reentrant library function in standard library file.

Answer

There are reentrant function lists on [11.3.1 reentrant library]. Reentrant function can be generated by setting of library generator in SHC V7.0 or later.

- On command line, use the lbgsh-reent option.
- The setting in the HEW is shown in figure 11.12.



Configuration : Debug

Category : Object

☐ Simple I/O function

☑ Generate reentrant library

Section : Program section (P)

P

Store string data in : Const section

Division sub-options : Default

Use 16 bit short address : None

Output file path : $(CONFIGDIR)¥$(PROJECTNAME).lib

Options Standard Library :

```
-cpu=sh3 -output="$(CONFIGDIR)¥$(PROJECTNAME).lib"
-reent -noinline -gbr=auto -head=runtime,new,stdio,stdlib,string
```

Figure 11.12 Standard Library Dialog Box

11.3.3 There is no standard library file. (SHC V6, 7, 8)

Question

There is no standard library file which is supported in SHC V6 or later. (SHC V6, 7, 8).

Answer

Since SHC V6, the specification of the standard library was changed. and the options become to be able to be specified. This enabled the user to have the standard libraries turned by the options. Please generate a standard library file by using a library generator since a standard library file has not been attached to a product in SHC V6 or later.

11.3.4 Warning Message on Building Standard Library

Question

[L1200(W) Backed up file "a.lib" into "b.lbk"] may be output when generate a standard library file.

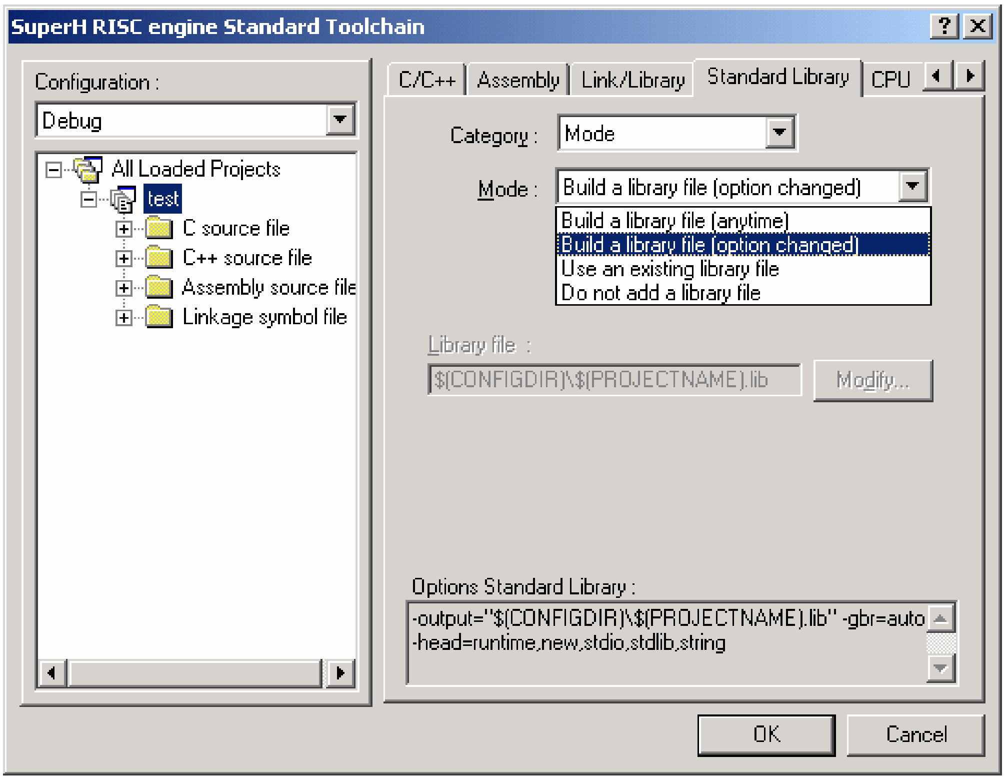
Answer

This is just warning message which HEW will make backup files when it generates new library files.

If you select "Build a library file (option changed)" at [Standard Library] mode: in HEW/[OPTIONS]/[SuperH RISC engine Standard Toolchain], the warning will not be issued. When you select "BUILD ALL" in HEW, Linkage editor generates a standard library at first. For the first project you created, it is necessary to build a standard library, and so you must select the “Build a library file” in the [Standard Library] mode of the HEW/[OPTIONS]/[SuperH RISC engine Standard Toolchain.].

However, a standard library is already created in the file for which BUILD ALL is once specified, and so the automatic generation of a standard library is not necessary for this file. In this case, since a standard library is automatically generated for each BUILD ALL specification, the existing library is backed up.

If you select the “Build a library file (option changed)”, this warning message can be avoided. Also, this can save the time required for automatically generating a standard libray on BUILD ALL.



Configuration : Debug

Category : Mode

Mode : Build a library file (option changed)

Library file : $(CONFIGDIR)\$(PROJECTNAME).lib

Options Standard Library :

```
-output="$(CONFIGDIR)\$(PROJECTNAME).lib" -gbr=auto
-head=runtime,new,stdio,stdlib,string
```

Figure 11.13 Standard Library Dialog Box

11.3.5 Size of Memory Used as Heap

Question

Tell me how to calculate the size of the memory used as heap.

Answer

The size of the memory used as heap is the total of memory areas assigned by the memory management library functions (calloc, malloc, ralloc, new) in a C/C++ program. However, these functions use four bytes as management area each time they are called. Calculate the heap size by adding this size to the size of the actually assigned area.

The compiler manages the heap in 1024 byte unit. Calculate the size of the area allocated as heap (HEAPSIZE) as follows:

HEAPSIZE = 1024 x n (n≥1)

(area size allocated by memory management library) + (Management area size≤HEAPSIZE)

The I/O library functions use the memory management library functions in internal processing. The size of the area allocated during I/O is 516 bytes x maximum number of concurrently open files.

Note: The area freed by the memory management library function free or delete is reused by a memory management library function for allocation. Even if the total size of the free area is sufficient, repeating allocations causes the free area to be divided into smaller ones, making the allocation of newly requested large areas impossible. To prevent this situation, use the heap area according to following suggestions.

a. Large sized areas should be allocated immediately after the program starts to run.

b. The size of the data area to be freed and reused should be constant.

11.3.6 Editing Library Files

Question

How can I edit an existing library file, so that I can re-use it?

Answer

Existing library files can be edited by using the options for the optimization linkage editor. The following explains each editing function.

The H Series Librarian Interface is provided to launch the optimization linkage editor from the GUI.

Starting the H Series Librarian Interface

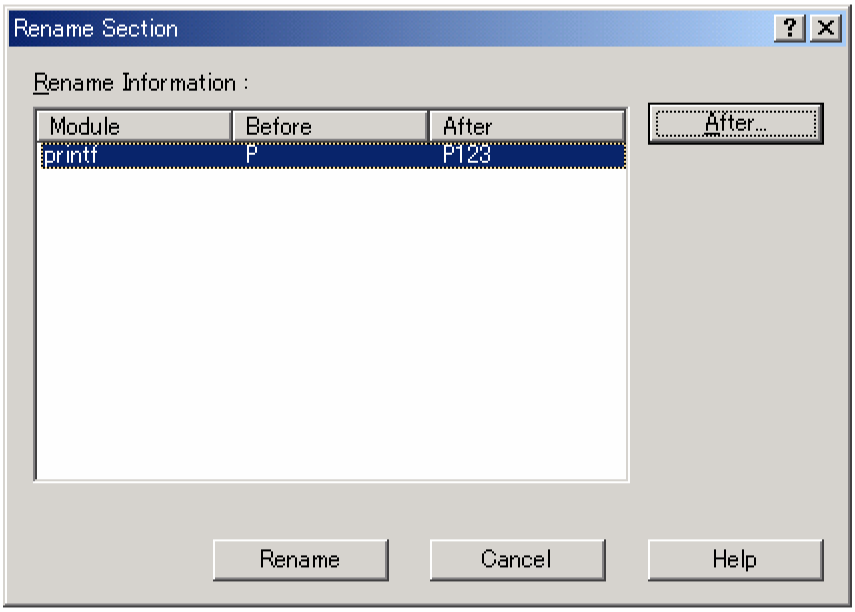
To start the H Series Librarian Interface, from HEW, choose [Tools -> H Series Librarian Interface].

(A) Changing the section names of the modules in the library

You can change the section names and place sections at a specific address for specific modules in the library.

(1) Open the appropriate library, and select the module that you would like to allocate to a specific address.

(2) Choose [Action -> Rename Section...] to display the following dialog box, and click the [After] button to change the section name.



Rename Section

Rename Information :

Module	Before	After
printf	P	P123

Figure 11.14 Rename Section Dialog Box

[For the command line]

optlnk –form=lib –lib=*library-file-name* -rename=*name-of-the-module-in-the-library* (P=P123)

(B) Swapping modules in the library and adding new modules to the library

You can swap library modules, as well as add new ones.

(1) Open the appropriate library, and choose [Action -> Add/Replace...].

(2) Open the module to be swapped, of the same name. If a module with a different name is opened, the module is added.

[For the command line]

optlnk –form=lib –lib=*library-file-name* -replace=*name-of-the-module-in-the-library*

(C) Deleting modules in the library

You can delete library modules.

(1) Open the appropriate library, and select the module or modules you would like to delete.

(2) Choose [Action -> Delete...] to display the Delete dialog box, and the click the [Delete] button.

[For the command line]

optlnk –form=lib –lib=*library-file-name* -delete=*name-of-the-module-in-the-library*

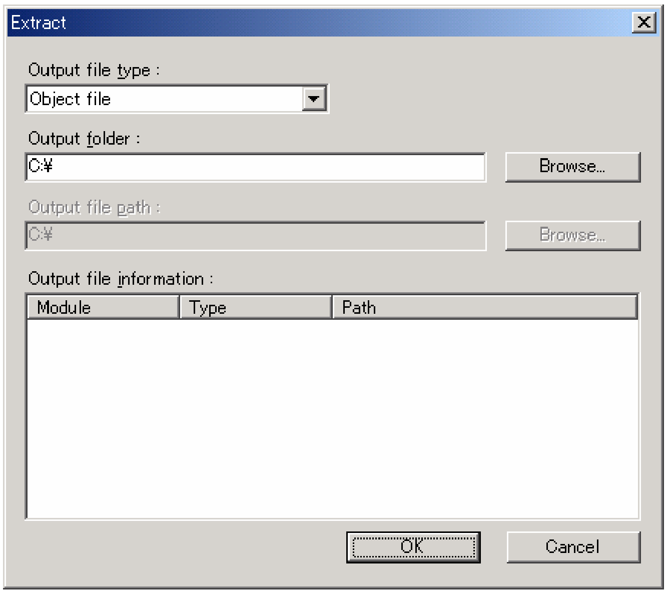
(D) Extracting modules from the library

You can extract library modules.

(1) Open the appropriate library, and select the module or modules you would like to extract.

(2) Choose [Action -> Extract...] to display the following dialog box, set the output destination, and then click the [OK] button.

(3) The module or modules are output to the set output destination (this is C:\ in the following example).



Extract

Output file type :

Object file

Output folder :

C:¥

Output file path :

C:¥

Output file information :

Module	Type	Path

Figure 11.15 Extract Dialog Box

[For the command line]

optlnk –lib=*library-file-name* -extract=*name-of-the-module-in-the-library* -form=*output-file-format*

Note that the output format for this example is object.

11.4 HEW

11.4.1 Failure to Display Dialog Menu

Question

Tool option dialog boxes are not displayed correctly with the HIM and the HEW.

Answer

If an old release (such as 400.950a) of Windows®95 is used, an application error occurs when options in the C/C++ compiler, the Assembler, or the IM OptLinker are opened, and the HEW may aborts the operation abnormally or option dialog boxes may not be displayed correctly. This problem is caused when the version of the COMCTL32.DLL file that is located in the System directory of the Windows directory is too old. In this case, upgrade the Windows®95.

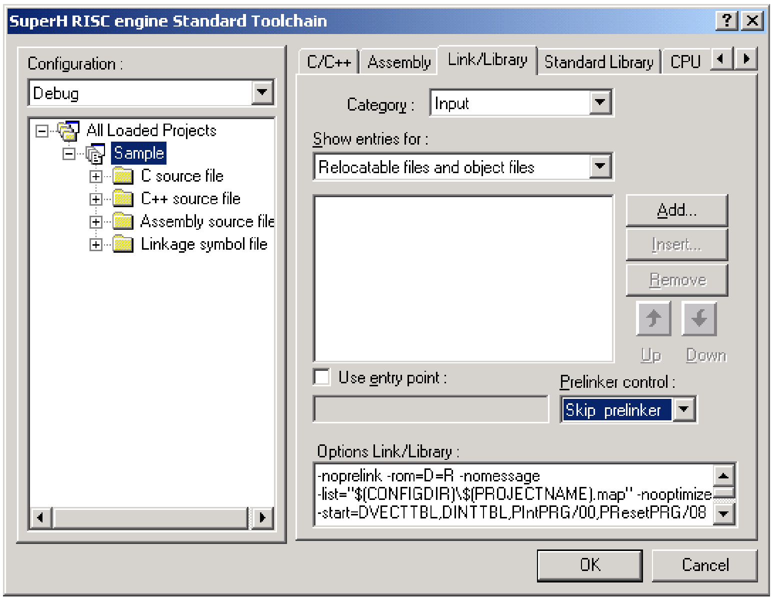
11.4.2 Linkage Order of Object Files

Question

I would like to specify an order of link of an object file on HEW.

Answer

Please add an object file by pushing [Add] and select the Show entry for: [Relocatable files and object files] from the category [Input] in the Link/Library tab of the SuperH RISC engine Standard Toolchain. An object is linked in order specified in this time.



Configuration : Debug

Category : Input

Show entries for : Relocatable files and object files

□ Use entry point :

Prelinker control : Skip prelinker

Options Link/Library :

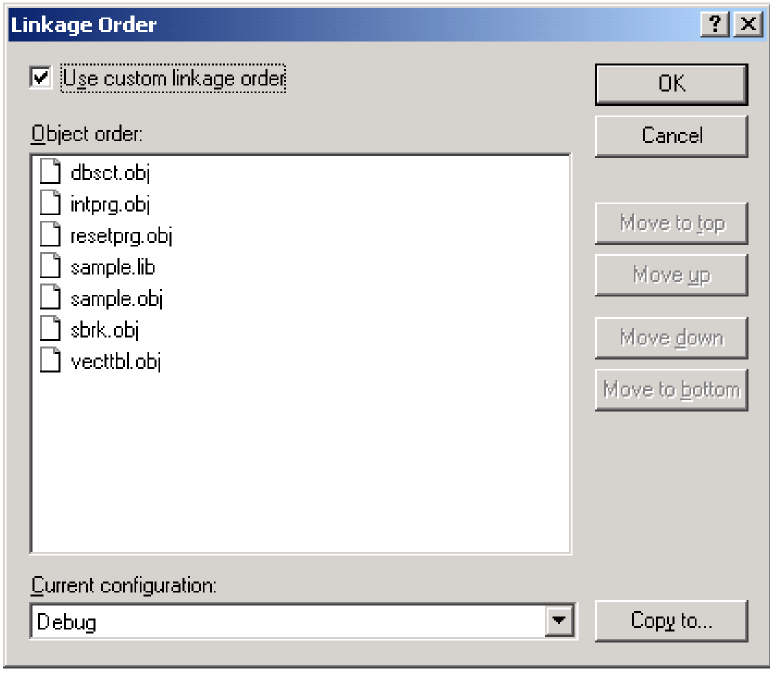
```
-noprelink -rom=D=R -nomessage
-list="$(CONFIGDIR)\$(PROJECTNAME).map" -nooptimize
-start=DVECTTBL,DINTTBL,PIntPRG/00,PResetPRG/08
```

Figure 11.16 Link/Library Dialog Box

SHC V.8.00 Release02 or later eases specifying the link order.

To display the dialog box for customizing the link order, choose [Build], and then [Specify link order].

Here, specify the link order. The items higher on the list are linked first.



☑ Use custom linkage order

Object order:

- dbsct.obj
- intprg.obj
- resetprg.obj
- sample.lib
- sample.obj
- sbrk.obj
- vecttbl.obj

Current configuration: Debug

Figure 11.17 Linkage Order Dialog Box

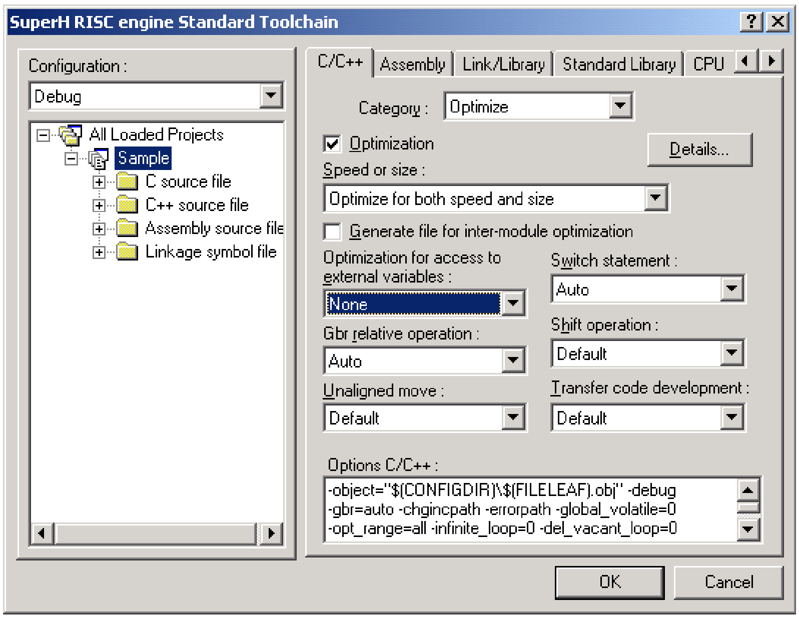
11.4.3 Specifying the MAP Optimization

Question

Specifying the MAP optimization will cause a warning message to appear.

Answer

If you check the “Include map file” in the C/C++ tab’s category: [Optimize] of the SuperH RISC engine Standard Toolchain, the warning message shown in figure 11.18 appears. This is to automatically enable the “Generate map file” in the Link/Library tab’s category: [Output].


SuperH RISC engine Standard Toolchain

Configuration : Debug

- All Loaded Projects
  - Sample
    - C source file
    - C++ source file
    - Assembly source file
    - Linkage symbol file

Category : Optimize

☑ Optimization

Speed or size : Optimize for both speed and size

☐ Generate file for inter-module optimization

Optimization for access to external variables : None

Switch statement : Auto

Gbr relative operation : Auto

Shift operation : Default

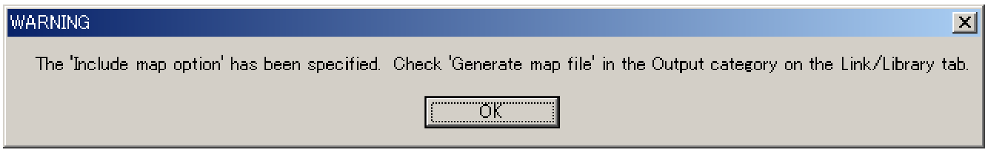
Unaligned move : Default

Transfer code development : Default

Options C/C++ :

```
-object="$(CONFIGDIR)\$(FILELEAF).obj" -debug
-gbr=auto -chgincpath -errorpath -global_volatile=0
-opt_range=all -infinite_loop=0 -del_vacant_loop=0
```

Figure 11.18 C/C++ Dialog Box


WARNING

The 'Include map option' has been specified. Check 'Generate map file' in the Output category on the Link/Library tab.

Figure 11.19 Warning Message

11.4.4 Excluding a project file

Question

I would like to eliminate a project file from Build temporarily.

Answer

The file is eliminated from Build if choose [Exclude Build <file>] by pressing a right button of mouse onto the file of "Projects" tab on work space window. If sending a file back to Build again, please choose [Include Build <file>] by pressing a right button of mouse on the file of "Projects" tab on work space window.

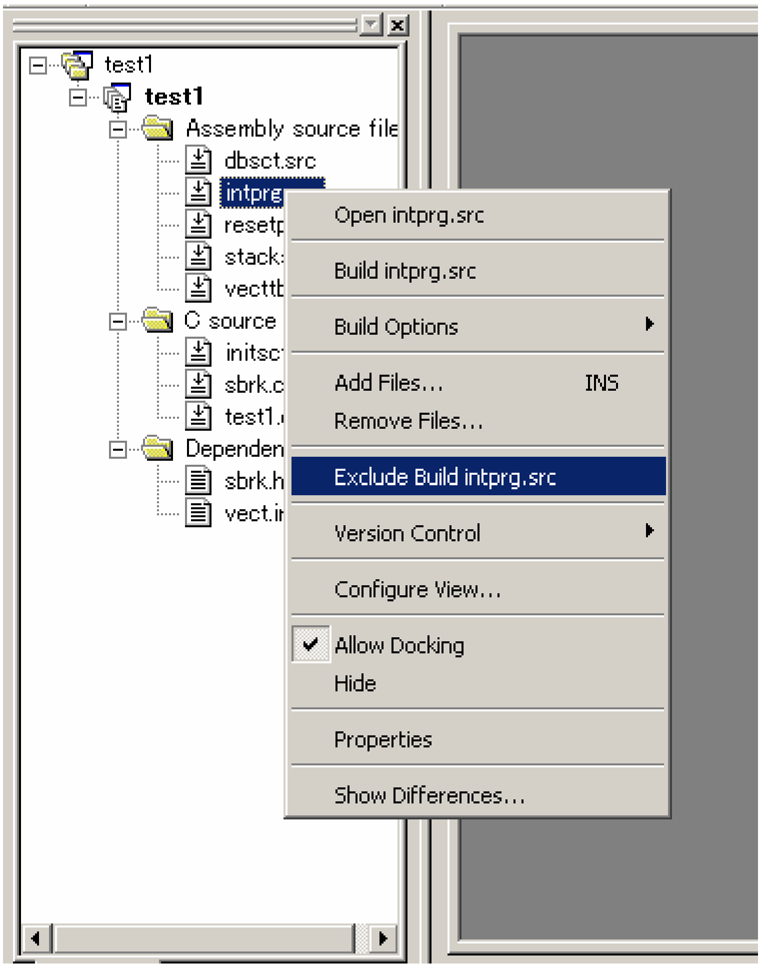


Figure 11.20 Exclude Build Menu

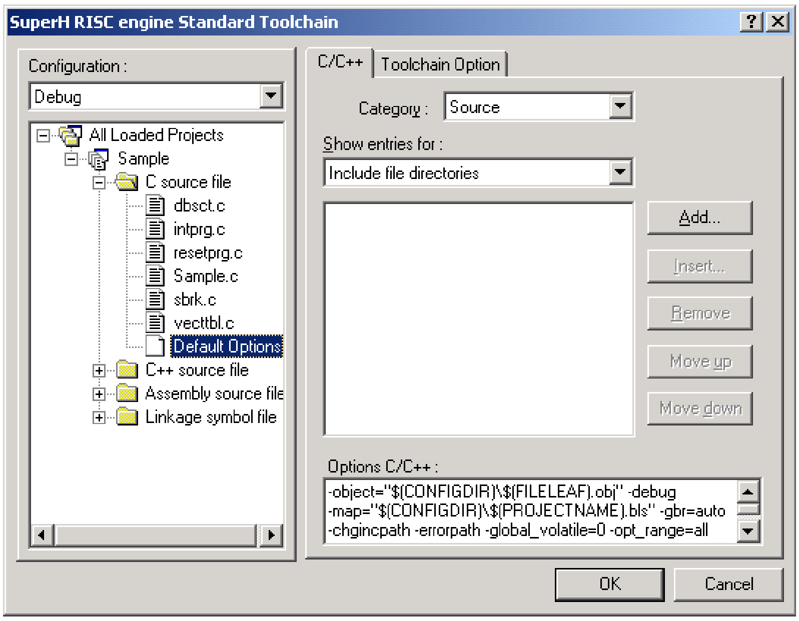
11.4.5 Specifying the Default Options for Project Files

Question

I would like to automatically specify a default option into file when adding a project into file.

Answer

The list of files is displayed on the left of the SuperH RISC enging Standard Toolchain (see figure 11.21). Please open the folder in file group in which Default Option is to be specified by the file list. "Default Options" icon is displayed in the folder. Please choose an icon and click "OK" by specifying an option in the right side of an option dialog box. This option can be applied when a file of the file group is first added to the project.



SuperH RISC engine Standard Toolchain

Configuration : Debug

Category : Source

Show entries for : Include file directories

Options C/C++ :

```
-object="$(CONFIGDIR)\$(FILELEAF).obj" -debug
-map="$(CONFIGDIR)\$(PROJECTNAME).bls" -gbr=auto
-chgincpath -errorpath -global_volatile=0 -opt_range=all
```

Figure 11.21 Default Options

11.4.6 Changing Memory Map

Question

A memory map can not be changed.

Answer

When a memory source of the memory window has been mapped, a memory map can not be changed in the system configuration window. Please change a memory map after mapping of a memory resource was released.

11.4.7 How to Use HEW on Network

Question

(1) Can the HEW be installed on a network?
(2) Can projects and programs be installed on a network?

Answer

(1) The HEW system itself cannot be installed on a network.
(2) No problem. Be careful not to access a single file by plural users.

11.4.8 Limitations on File and Directory Names Created in HEW

Question

The message ”Error has occurred whilst saving file <filename>” is displayed at the HEW system startup. Why is it?

Answer

Files and directories created on the HEW system have limitations.

For the specifications of the following items, only half-width alphanumeric characters and half-width underlines can be used:

- Names of the directories to be installed
- Names of the directories in which projects are to be created
- Project names

11.4.9 Failure of Japanese Font Display with the HEW Editor or HDI

Question

(1) Japanese fonts are not displayed with the HEW editor.

(2) Japanese characters are rotated 90 degrees with the HEW editor.

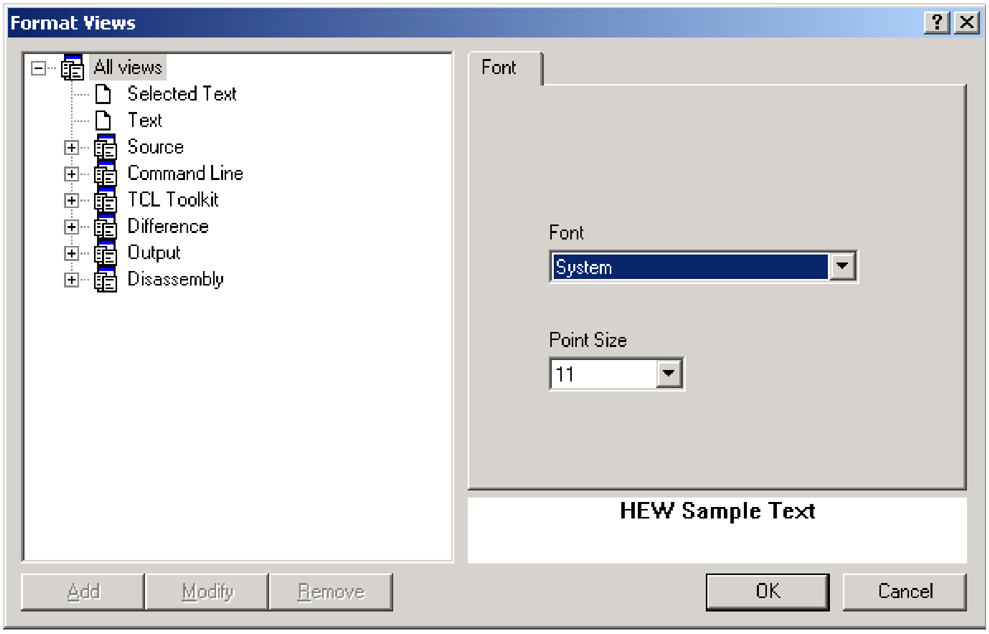
(3) The inter-module optimizer generates SYNTAX ERROR messages.

Answer

When coding Japanese with the HEW editor, specify Japanese font as follows:

<HEW2.0 or later>

Use Font of the Font tab in Tools-> Format Views.:



Format Views

Font

System

Point Size

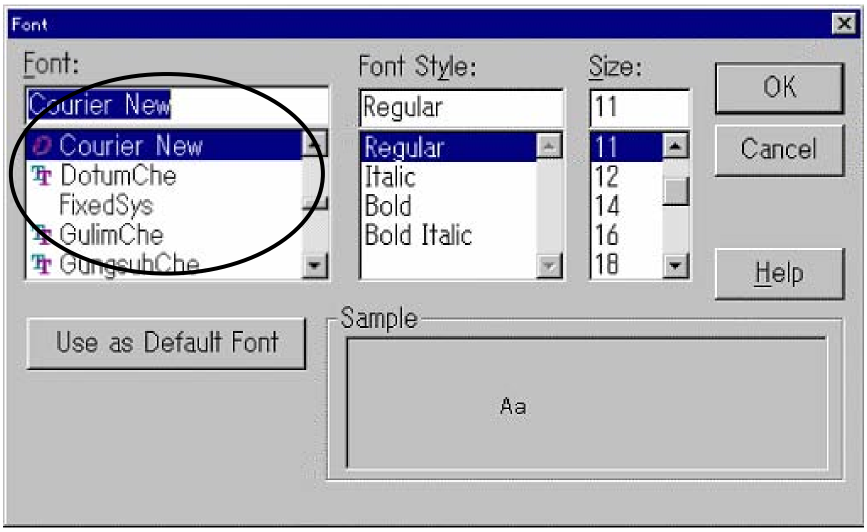
11

HEW Sample Text

Figure 11.22 Font Dialog Box

If Japanese fonts are not correctly displayed with the HDI, modify as follows:

[Setup->Customize->Font…]



Font: Courier New

Font Style: Regular

Size: 11

Figure 11.23 Font Dialog Box

11.4.10 How to Convert Programs from HIM to HEW

Question

How can I use a project created under HIM (Hitachi Integration Manager) on the HEW?

Answer

Projects can be converted from HIM to HEW using a tool called " HIM To HEW Project Converter" that is supplied with the HEW system.

For details on this tool, refer to section 3, Converting a Project from HIM to HEW, in the Renesas High-performance Embedded Workshop Release Notes.

11.4.11 Corresponding Device Not Available during HEW Project Setup

Question

When I attempted to use HEW to set up a project, the device I want to select was not available for selection. What should I do?

Answer

Download Device Updater from the Renesas web site.

Device Updater is a tool for updating project files generated by HEW. Project support for new CPUs is performed in order.

If the corresponding device still cannot be selected after the project files have been updated by Device Updater, the files generated by HEW will have to be rewritten manually. For example, take the case of setting up the SH7018 project for the SH-2 core. First, create a new project in HEW, with SH-2 selected for the CPU series, and Other selected for the CPU type. Download the I/O register definition file from the Renesas web site, and add SH7018.H to the project. If Other was not selected for the CPU type during project setup, rename SH7018.H to iodefine.h, overwriting the file of the same name, as generated by HEW.

Also, when interrupt functions are used, files generated by HEW must be changed. For details, see section 11.1.41, Specifying Interrupt Processing.

11.4.12 I want to use an old compiler (tool chain) in the latest HEW.

Question

I have an old compiler package. When I bought an Emulator, new HEW was bundled.

In order to Build and Debug with new HEW, I want to use an old tool chain in the new HEW.

Can I do that?

Answer

It depends on the version of the compiler package you are using. See below.

[SHC V.4 or previous]

< Build >

The tool chain cannot be registered in the latest HEW. Therefore, building by new HEW is not available.

< Debug >

Absolute file (*.abs) cannot be used. You can only use S-type format file.

Moreover, debugging at C source level is not available. Only at assembler level is available.

[SHC V.5.0]

< Build >

The tool chain cannot be registered in the latest HEW. Therefore, building by new HEW is not available.

(Note)

"HIM to HEW Project Converter" is usable if you have SHC V.5.1 compiler package.

By using this tool, you can convert HIM project into HEW project. You can use SHC V.5.1 with new HEW after conversion.

< Debug >

Absolute file (*.abs) cannot be used. You can only use S-type format file.

Moreover, debugging program at C source level is not available. Only at assembler level is available.

[SHC V.5.1]

< Build >

The tool chain can be registered in the latest HEW. Therefore, building by new HEW is available.

But you cannot create new project with the latest HEW.

In case of creating new project, you must use HEW V.1 bundled with the older compiler package.

Once you create project by HEW V.1, you can open it with in new HEW.

< Debug >

Absolute file (*.abs) cannot be used. You can only use S-type format file.

Moreover, debugging program at C source level is not available. Only at assembler level is available.

[SHC V.6]

< Build >

The tool chain can be registered in the latest HEW. Therefore, building by new HEW is available.

But you cannot create new project with the latest HEW.

In case of creating new project, you must use HEW V.1 bundled with the older compiler package.

Once you create project by HEW V.1, you can open it in new HEW.

< Debug >

Absolute file (*.abs) can be used.

By registering absolute file, debugging at C source level is available.

[SHC V.7 or later]

<Build & Debug>

There is no limitation. You can use all functions of new HEW.

Appendix A Rules for Naming Runtime Routines

The rules for naming function names of runtime routines are as follows.

(1) Rules for naming integer operations, floating point operations, sign conversion, and bit field functions

[operation name] [size] [sign] [r] [p] [nm]

[size] : b …1 byte
: w …2 bytes
: l …4 bytes
: s …4 bytes (single-precision floating point)
: d …8 bytes (double-precision floating point)

[sign] : s …signed
: u …unsigned

[r] : _subdr, _divdr only; only when the order of parameter stack pushing is different from _subd, _divd respectively

[p] : Added only in peripheral processing

[nm] : No mask; added only in peripheral processing when there is no interrupt mask

exception: _muli

Note: The [sign] identifier is added only for integer operations.

(2) Rules for naming conversion functions

_[size]to[size]

[size] : i …Signed, 4 bytes
: u …Unsigned, 4 bytes
: s …Single-precision floating point
: d …Double-precision floating point

(3) Rules for naming shift functions

[sta] sft [direction] [sign] [number of bits]

[sta_] : Added only when the number of bits is added

[direction] : l …Left-shift
: r …Right-shift

[sign]*1 : l …Logical shift
: a …Arithmetic shift

[number of bits]*2: 0 to 31

Notes: 1. [sign] added only when [direction] is r.
2. [number of bits] added only when [sta_] is added.

(4) Rules for naming other functions

Memory area movement, character string comparison, and character string copy functions are special cases.

Appendix B Added Features

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

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

Table B.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.

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

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

Table B.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 B.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.

B.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)        #pragma inline(func)
int a;                         int a;
static int func(){             /* func() function is itself deleted */
   a++;
}
main(){                     main(){
   func();                       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.

B.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 B.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++

B.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.

B.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.

B.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 B.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.

Deletion of vacant loops

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

Specification of maximum unroll factor

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.**

Deletion of assignments before an infinite loop

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

Allocation of global variable

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

Allocation of struct/union member

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

Propagation of const-qualified variable

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.

Inline expansion of constant load

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=inlin**e.

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=litera**l.

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
```

Scheduling of instructions

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.

B.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    ; _a1            MOV.L        L12+2,R6    ; _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; _a2                MOV.L        L13+2,R2; _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

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

After optimization

```
_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

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

After optimization

```
_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

```
_f:
                  :
    EXTS.B       R6,R2
    EXTU.B       R2,R0
                  :
```

After optimization

```
_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:`	`_f:`
`:`	`:`
`TST      #4,R0`	`TST      #4,R0`
`MOVT     R0`	`MOV.L    L23+6,R6; _st2`
`MOV.L    L23+6,R6   ; _st2`	`MOV.B    @R6,R0`
`XOR      #1,R0`	`BT  L16`
`EXTU.B   R0,R0`	`:`
`CMP/EQ   #1,R0`	
`MOV.B    @R6,R0`	
`BF       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:`	`L17:`
`MOV.B       @R6,R0`	`MOV.B       @R6,R0`
`AND         #1,R0`	`TST         #1,R0`
`CMP/EQ      #1,R0`	`BT          L19`
`BF          L19`	`MOV.B       @R5,R0`
`MOV.B       @R5,R0`	`AND         #1,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:	_f:
MOV.L L11,R6 ; _b	MOV.L L11,R6 ; _b
MOV.B @R6,R2	MOV.B @R6,R2
MOV #-128,R6; H'FFFFFF80	MOV #-128,R6; H'FFFFFF80
EXTU.B R6,R6	CMP/EQ R6,R2
EXTU.B R2,R2	MOVT R2
CMP/EQ R6,R2	MOV.L L11+4,R6 ; _a
MOVT R2	RTS
MOV.L L11+4,R6 ; _a	MOV.B R2,@R6
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:	_f:
MOV.L L11+2,R6;_st	MOV.L L11+2,R6;_st
MOV.B @R6,R2	MOV.B @R6,R2
EXTU.B R2,R0	OR #128,R0
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	MOVT R0
XOR #1,R0	RTS
RTS	XOR #1,R0
EXTU.B R0,R0	
MOVT R0	:
OR #2,R0	MOVT R0
RTS	RTS
EXTU.B R0,R0	OR #2,R0
:	:
MOVT R0	MOVT R0
AND #1,R0	RTS
RTS	AND #1,R0
EXTU.B R0,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:	_f:
:	:
EXTS.B R6,R6	CMP/GT R6,R2
CMP/GT R6,R2	BF L13
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

```
_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
                      :
```

After optimization

```
_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

```
_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
```

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
   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) {`	`goto L1;`
`:`	`do {`	`do {`
`}`	`:`	`:`
	`} while (cond);`	`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)`	`if (cond) {`
`t=65;`	`t=65;`
`else`	`fx();`
`t=67;`	`} else {`
`if (t == 65)`	`t=67;`
`fx();`	`fy();`
`else`	`}`
`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 = i + prime + flags;`
`p = flags + k;`	
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 {	struct S {
char bit0:1;	char bit0:1;
char bit1:1;	char bit1:1;
char bit2:1;	char bit2:1;
char bit3:1;	char bit3:1;
}ss1;	}ss1;
if((ss1.bit0|ss1.bit1|ss1.bit2)!= 0){	if ((*(char *)&ss1 & 0xe0)!= 0){
:	:
:	:
}	}
Simultaneously performs fetching and comparison using AND.	

B.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.

B.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.

Appendix C 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).

C.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.

C.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.

Appendix D ASCII Code Table

Table D.1 ASCII Code Table

Upper four bits								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Lower four bits								
0	NULL	DLE	SP	0	@	P	′	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	•	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Renesas Microcomputer Development Environment System
Application Note
SuperH RISC engine C/C++ Compiler Package

Publication Date: Rev.1.00, November 9, 2004
Rev.4.00, February 2, 2007
Published by: Sales Strategic Planning Div.
Renesas Technology Corp.
Edited by: Customer Support Department
Global Strategic Communication Div.
Renesas Solutions Corp.

Renesas Technology Corp. Sales Strategic Planning Div. Nippon Bldg., 2-6-2, Ohte-machi, Chiyoda-ku, Tokyo 100-0004, Japan

RENESAS SALES OFFICES

http://www.renesas.com

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

Renesas Technology America, Inc.
450 Holger Way, San Jose, CA 95134-1368, U.S.A
Tel: <1> (408) 382-7500, Fax: <1> (408) 382-7501

Renesas Technology Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K.
Tel: <44> (1628) 585-100, Fax: <44> (1628) 585-900

Renesas Technology (Shanghai) Co., Ltd.
Unit 204, 205, AZIACenter, No.1233 Lujiazui Ring Rd, Pudong District, Shanghai, China 200120
Tel: <86> (21) 5877-1818, Fax: <86> (21) 6887-7898

Renesas Technology Hong Kong Ltd.
7th Floor, North Tower, World Finance Centre, Harbour City, 1 Canton Road, Tsimshatsui, Kowloon, Hong Kong
Tel: <852> 2265-6688, Fax: <852> 2730-6071

Renesas Technology Taiwan Co., Ltd.
10th Floor, No.99, Fushing North Road, Taipei, Taiwan
Tel: <886> (2) 2715-2888, Fax: <886> (2) 2713-2999

Renesas Technology Singapore Pte. Ltd.
1 Harbour Front Avenue, #06-10, Keppel Bay Tower, Singapore 098632
Tel: <65> 6213-0200, Fax: <65> 6278-8001

Renesas Technology Korea Co., Ltd.
Kukje Center Bldg. 18th Fl., 191, 2-ka, Hangang-ro, Yongsan-ku, Seoul 140-702, Korea
Tel: <82> (2) 796-3115, Fax: <82> (2) 796-2145

Renesas Technology Malaysia Sdn. Bhd
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No.18, Jalan Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: <603> 7955-9390, Fax: <603> 7955-9510

Colophon 6.0

SuperH RISC engine C/C++ Compiler Package
Application Note

Renesas Electronics Corporation
1753, Shimonumabe, Nakahara-ku, Kawasaki-shi, Kanagawa 211-8668 Japan