

C Compiler for M16C Family

Application Notes

M3T-NC308WA/M3T-NC30WA

Application Note

Rev.2.00
Jun. 16, 2005

Renesas Technology
www.renesas.com

Keep safety first in your circuit designs!

1. Renesas Technology Corp. puts the maximum effort into making semiconductor products better and more reliable, but there is always the possibility that trouble may occur with them. Trouble with semiconductors may lead to personal injury, fire or property damage. Remember to give due consideration to safety when making your circuit designs, with appropriate measures such as (i) placement of substitutive, auxiliary circuits, (ii) use of non-flammable material or (iii) prevention against any malfunction or mishap.

Notes regarding these materials

1. These materials are intended as a reference to assist our customers in the selection of the Renesas Technology Corp. product best suited to the customer's application; they do not convey any license under any intellectual property rights, or any other rights, belonging to Renesas Technology Corp. or a third party.
2. Renesas Technology Corp. assumes no responsibility for any damage, or infringement of any third-party's rights, originating in the use of any product data, diagrams, charts, programs, algorithms, or circuit application examples contained in these materials.
3. All information contained in these materials, including product data, diagrams, charts, programs and algorithms represents information on products at the time of publication of these materials, and are subject to change by Renesas Technology Corp. without notice due to product improvements or other reasons. It is therefore recommended that customers contact Renesas Technology Corp. or an authorized Renesas Technology Corp. product distributor for the latest product information before purchasing a product listed herein.
The information described here may contain technical inaccuracies or typographical errors. Renesas Technology Corp. assumes no responsibility for any damage, liability, or other loss rising from these inaccuracies or errors.
Please also pay attention to information published by Renesas Technology Corp. by various means, including the Renesas Technology Corp. Semiconductor home page (<http://www.renesas.com>).
4. When using any or all of the information contained in these materials, including product data, diagrams, charts, programs, and algorithms, please be sure to evaluate all information as a total system before making a final decision on the applicability of the information and products. Renesas Technology Corp. assumes no responsibility for any damage, liability or other loss resulting from the information contained herein.
5. Renesas Technology Corp. semiconductors are not designed or manufactured for use in a device or system that is used under circumstances in which human life is potentially at stake. Please contact Renesas Technology Corp. or an authorized Renesas Technology Corp. product distributor when considering the use of a product contained herein for any specific purposes, such as apparatus or systems for transportation, vehicular, medical, aerospace, nuclear, or undersea repeater use.
6. The prior written approval of Renesas Technology Corp. is necessary to reprint or reproduce in whole or in part these materials.
7. If these products or technologies are subject to the Japanese export control restrictions, they must be exported under a license from the Japanese government and cannot be imported into a country other than the approved destination.
Any diversion or reexport contrary to the export control laws and regulations of Japan and/ or the country of destination is prohibited.
8. Please contact Renesas Technology Corp. for further details on these materials or the products contained therein.

Preface

These application notes explain methods for the efficient creation of application programs which capitalize on the function and performance of the Renesas Technology M16C family using the NC308WA V.5.20 Release 02 and NC30WA V.5.30 Release 02.

For Detailed specifications of the C compiler please refer to the NC308 User's Manual and NC30 User's Manual.

Related Manuals

- Renesas Technology M16C Family Hardware Manuals
- High-performance Embedded Workshop User's Manual
- NC308 User's Manual
- NC30 User's Manual
- M16C/80 & M32C/80 Series Programming Guidelines <C Language>

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 followed by H are in hexadecimal.

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

0b: Integer constants preceded by 0b are in binarydecimal.

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.

Linux is a trademark of Linus Torvalds.

Turbolinux and its logo are trademarks of Turbolinux, Inc.

Solaris is a registered trademark of Sun Microsystems, Inc.

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

Contents

SECTION 1.OVERVIEW	1-1
1.1SUMMARY	1-1
1.1.1SPECIFICATION SUMMARY	1-1
1.2FEATURES	1-2
1.3METHOD OF INSTALLATION	1-3
1.4METHOD OF EXECUTION	1-4
1.4.1STARTING THE COMPILER.....	1-4
1.4.2STARTING THE EMBEDDED WORKSHOP	1-5
1.5PROCEDURE FOR PROGRAM DEVELOPMENT	1-6
SECTION 2.PROCEDURE FOR CREATING A PROGRAM.....	2-1
2.1CREATING A PROJECT	2-1
2.2START-UP PROGRAMS.....	2-10
2.2.1PURPOSE OF STARTUP PROGRAMS.....	2-10
2.2.2SETTING UP A STARTUP PROGRAM	2-12
SECTION 3.COMPILER.....	3-1
3.1INTERRUPT FUNCTIONS.....	3-1
3.1.1CODING INTERRUPT PROCESSING FUNCTIONS.....	3-1
3.1.2CODING FAST INTERRUPT PROCESSING FUNCTIONS.....	3-4
3.1.3CODING FUNCTIONS FOR SOFTWARE INTERRUPT (INT INSTRUCTION) PROCESSING	3-6
3.1.4RETISTERING INTERRUPT PROCESSING FUNCTIONS	3-8
3.1.5CODING EXAMPLE OF INTERRUPT PROCESSING FUNCTIONS.....	3-9
3.2ASSEMBLER MACRO.....	3-11
3.2.1..... ASSEMBLY LANGUAGE INSTRUCTIONS THAT CAN BE SPECIFIED USING ASSEMBLER MACRO FUNCTIONS.....	3-11
3.2.2DECIMAL ADDITION USING THE ASSEMBLER MACRO FUNCTION "DADD_B"...	3-14
3.2.3TRANSFERRING STRINGS USING THE ASSEMBLER MACRO FUNCTION "SMOVF_B"	3-15
3.2.4SUM-OF-PRODUCTS OPERATION USING THE ASSEMBLER MACRO FUNCTION "RMPA_W"	3-16
3.2.5#PRAGMA __ASMMACRO.....	3-17
3.3PRAGMA FUNCTIONS AND OPTIONS FOR REDUCING ROM AREA.....	3-18
3.3.1#PRAGMA SBDATA.....	3-18
3.3.2#PRAGMA SB16DATA	3-19
3.3.3#PRAGMA BIT.....	3-20
3.3.4#PRAGMA SPECIAL.....	3-21
3.3.5-FJSRW.....	3-22
3.3.6-OR	3-23
3.3.7-FNO_ALIGN.....	3-24
3.3.8-WNO_USED_FUNCTION.....	3-25
3.4PRAGMA FUNCTION AND OPTIONS FOR SPEEDING UP PROCESSING	3-26
3.4.1#PRAGMA STRUCT.....	3-27
3.4.2-OSTACK_FRAME_ALIGN	3-31

3.4.3-OS.....	3-32
3.4.4-OLoop_UNROLL[=COUNT].....	3-33
3.4.5-Ofloat_TO_INLINE.....	3-34
3.4.6-OSTatic_TO_INLINE.....	3-35
3.5PRAGMA FUNCTIONS AND OPTIONS FOR REDUCING ROM AREA AND SPEEDING UP PROCESSING.....	3-36
3.5.1-O[1-5].....	3-37
3.5.2-OSP_ADJUST.....	3-39
3.5.3-FUSE_DIV.....	3-40
3.5.4-WNO_USED_ARGUMENT.....	3-41
3.5.5-FSMALL_ARRAY.....	3-42
3.5.6-FDOUBLE_32.....	3-43
3.6OTHER PRAGMA FUNCTIONS AND OPTIONS.....	3-44
3.6.1OTHER PRAGMA FUNCTIONS.....	3-44
3.6.2OTHER OPTIONS.....	3-48
3.7SECTIONS.....	3-52
3.7.1SECTIONS MANAGED BY NC308.....	3-52
3.8ISSUES RELATED TO CROSS-SOFTWARE.....	3-55
3.8.1ISSUES RELATED TO ASSEMBLY LANGUAGE PROGRAMS.....	3-55
3.9LONG LONG TYPE.....	3-63
3.10"NEAR/FAR" TYPE.....	3-64
3.10.1NEAR AND FAR AREAS.....	3-64
3.10.2DEFAULTS OF THE "NEAR" AND "FAR" ATTRIBUTES.....	3-64
3.10.3"NEAR" AND "FAR" SPECIFICATION FOR FUNCTIONS.....	3-64
3.10.4"NEAR" AND "FAR" SPECIFICATION FOR VARIABLES.....	3-65
3.10.5"NEAR" AND "FAR" SPECIFICATION FOR POINTERS.....	3-67
3.10.6DIFFERENCE IN POINTER'S "NEAR/FAR" SPECIFICATION BETWEEN NC308 AND NC30.....	3-69
3.10.7ASSIGNING VARIABLE ADDRESS IN THE FAR AREA TO THE "NEAR" POINTER.....	3-69
3.11INLINE EXPANSION.....	3-70
3.11.1OVERVIEW OF THE INLINE STORAGE CLASS.....	3-70
3.11.2FORMAT OF AN INLINE STORAGE CLASS DECLARATION.....	3-70
3.11.3RULES FOR THE INLINE STORAGE CLASS.....	3-72
SECTION 4.HIGH-PERFORMANCE EMBEDDED WORKSHOP.....	4-1
4.1SPECIFYING OPTIONS IN HIGH-PERFORMANCE EMBEDDED WORKSHOP.....	4-1
4.1.1C COMPILER OPTIONS.....	4-2
4.1.2ASSEMBLER OPTIONS.....	4-13
4.1.3LINKAGE EDITOR OPTIONS.....	4-19
4.1.4LIBRARIAN OPTIONS.....	4-26
4.1.5LOAD MODULE CONVERTER OPTIONS.....	4-30
4.1.6CONFIGURATION OPTIONS.....	4-34
4.1.7CPU OPTIONS.....	4-37
4.2BUILDS.....	4-38
4.2.1MAKEFILE OUTPUT.....	4-38
4.2.2MAKEFILE INPUT.....	4-39
4.2.3CREATING CUSTOM PROJECT TYPES.....	4-41

4.2.4MULTI-CPU FEATURE	4-45
4.2.5NETWORKING FEATURE	4-46
SECTION 5.EFFICIENT PROGRAMMING TECHNIQUES	5-1
5.1REGISTER PASSING FOR ARGUMENTS	5-2
5.2USING REGISTER VARIABLES.....	5-3
5.3USING M16C-SPECIFIC INSTRUCTIONS	5-5
5.4USING THE "CARRY" FLAG FOR BIT OPERATION BRANCHING	5-6
5.5MOVING DETERMINATE EXPRESSIONS WITHIN A LOOP TO OUTSIDE OF THE LOOP	5-7
5.6 .SBDATA DECLARATION AND SPECIAL PAGE FUNCTION DECLARATION UTILITY	5-8
5.7USING "SWITCH" INSTEAD OF "ELSE IF"	5-9
5.8COMPARISON OPERATORS FOR LOOP COUNTERS	5-10
5.9RESTRICT.....	5-10
5.10USING _BOOL.....	5-11
5.11EXPLICITLY INITIALIZING AUTO VARIABLES.....	5-11
5.12INITIALIZING ARRAYS	5-12
5.13INCREMENTS / DECREMENTS.....	5-13
5.14"SWITCH" STATEMENTS	5-14
5.15IMMEDIATE FLOATING-POINTS	5-14
5.16ZERO CLEARING EXTERNAL VARIABLES	5-15
5.17ORGANIZING STARTUP	5-16
5.18USING TEMPORARY VALUES WITHIN LOOPS	5-18
5.19USING 32-BIT MATHEMATICAL FUNCTIONS	5-19
5.20USING UNSIGNED WHENEVER POSSIBLE	5-20
5.21ARRAY INDEX TYPES	5-21
5.22USING PROTOTYPE DECLARATIONS.....	5-21
5.23USING THE CHAR TYPE FOR FUNCTIONS THAT RETURN ONLY CHAR TYPE VALUES	5-23
5.24COMMENTING OUT CLEAR PROCESSING FOR BSS AREAS	5-24
5.25REDUCING GENERATED CODE.....	5-25
SECTION 6.USING THE SIMULATOR DEBUGGER.....	6-1
6.1 USING THE VIRTUAL INTERRUPT FUNCTION	6-2
6.1.1 INSERTING A VIRTUAL INTERRUPT BY BUTTON CLICK	6-2
6.1.2 INSERTING A VIRTUAL INTERRUPT AT A REGULAR INTERVAL	6-4
6.1.3 INSERTING A VIRTUAL INTERRUPT AT A SPECIFIED CYCLE	6-6
6.1.4 INSERTING A VIRTUAL INTERRUPT WHEN AN INSTRUCTION AT A SPECIFIED ADDRESS IS EXECUTED.....	6-9
6.2 USING THE VIRTUAL PORT INPUT/OUTPUT FUNCTION.....	6-11
6.2.1 ENTERING DATA BY BUTTON CLICK	6-11
6.2.2 ENTERING DATA FROM A VIRTUAL PORT WHEN A SPECIFIED ADDRESS IS READ	6-13
6.2.3 ENTERING DATA FROM A VIRTUAL PORT AT A SPECIFIED CYCLE	6-17
6.2.4 ENTERING DATA FROM A VIRTUAL PORT WHEN A VIRTUAL INTERRUPT OCCURS	6-19

6.2.5 CHECKING DATA OUTPUT TO A VIRTUAL OUTPUT PORT	6-21
6.3 USING A VIRTUAL LED OR LABEL TO CHECK THE MEMORY CONTENTS.....	6-26
6.4 USING PRINTF FOR DEBUGGING	6-29
6.5 USING I/O SCRIPTS	6-31
SECTION 7.MISRA C	7-1
7.1MISRA C.....	7-1
7.1.1WHAT IS MISRA C?.....	7-1
7.1.2RULE EXAMPLES	7-1
7.1.3COMPLIANCE MATRIX.....	7-3
7.1.4RULE VIOLATIONS.....	7-3
7.1.5MISRA C COMPLIANCE	7-3
7.2SQLMINT.....	7-4
7.2.1WHAT IS SQLMINT?.....	7-4
7.2.2USING SQLMINT	7-6
7.2.3VIEWING TEST RESULTS.....	7-7
7.2.4DEVELOPMENT PROCEDURES	7-8
7.2.5SUPPORTED COMPILERS	7-8
SECTION 8.FREQUENTLY ASKED QUESTIONS.....	8-1
8.1C COMPILER (M3T-NC308WA)	8-1
8.1.1BIT FIELDS	8-1
8.1.2MEMORY MANAGEMENT FUNCTIONS.....	8-2
8.1.3"-ONBSD2 OPTION.....	8-3
8.1.4PRIORITY OF OPTIMIZATION OPTIONS.....	8-4
8.1.5ADDING FUNCTIONS TO THE LIBRARY	8-5
8.1.6PLACING CONST DECLARATIONS IN THE ROM SECTION.....	8-6
8.1.7PASSING PARAMETERS VIA REGISTERS	8-7
8.1.8HOW FUNCTION PARAMETERS ARE PASSED.....	8-8
8.1.9PROTOTYPE DECLARATIONS	8-9
8.1.10MEMBER PLACEMENT IN A STRUCTURED BIT FIELD.....	8-10
8.1.11INCREMENT AND DECREMENT OPERATORS	8-12
8.1.12PLACING EXTERNAL VARIABLES.....	8-13
8.1.13PLACING AN ARRAY IN THE FAR AREA	8-14
8.1.14PLACING A FUNCTION AT A FIXED ADDRESS	8-15
8.1.15SPECIFYING AN ABSOLUTE ADDRESS USING #PRAGMA ADDRESS	8-16
8.1.16USING #DEFINE TO DEFINE A STRING.....	8-17
8.1.17TYPES OF BIT FIELD MEMBERS	8-18
8.1.18DUPLICATE VARIABLE DEFINITIONS.....	8-19
8.1.19PROTOTYPE DECLARATIONS FOR A FUNCTION	8-20
8.1.20EXTERNAL REFERENCES FOR FUNCTIONS WITHOUT AN EXTERN DECLARATION	8-21
8.1.21CODE DELETION DURING OPTIMIZATION	8-22
8.1.22CONSOLIDATING BIT ACCESS	8-23
8.1.23PLACING A LIBRARY FUNCTION AT A ROM ADDRESS	8-24
8.1.24PROCESSING FOR NEGATIVE INTEGER CALCULATIONS.....	8-25
8.1.25INT TYPE SIZES.....	8-26

8.1.26	CONTROLLING THE ENTER INSTRUCTION.....	8-28
8.1.27	PERFORMANCE FOR THE FLOATING-POINT LIBRARY	8-29
8.2	LINKER.....	8-30
8.2.1	"-LOC" OPTION FOR LN308 AND LN30.....	8-30
8.2.2	WARNINGS DURING LINKING	8-31
8.2.3	CHANGING A START ADDRESS	8-32
8.3	STK VIEWER.....	8-33
8.3.1	STK VIEWER STACK SIZE.....	8-33
8.4	SQMLINT.....	8-34
8.4.1	SELECTING TEST RULES.....	8-34
8.4.2	OUTPUTTING REPORT FILES.....	8-35
8.4.3	REPORT MESSAGES (1)	8-37
8.4.4	REPORT MESSAGES (2)	8-38
8.5	HIGH-PERFORMANCE EMBEDDED WORKSHOP	8-39
8.5.1	LINK ORDER FOR FILES.....	8-39
8.5.2	LINK ORDER FOR RELOCATABLE FILES	8-41
8.5.3	GENERATING MOTOROLA S FORMAT FILES	8-42
8.5.4	INSTALLING HIGH-PERFORMANCE EMBEDDED WORKSHOP (1)	8-43
8.5.5	INSTALLING HIGH-PERFORMANCE EMBEDDED WORKSHOP (2)	8-44
8.5.6	CANCELLING A BUILD.....	8-45
8.5.7	SELECTING A BUILD TARGET.....	8-46
8.5.8	BUILD CONFIGURATION.....	8-47
8.5.9	OUTPUTTING DEBUGGING INFORMATION	8-48
8.6	SBDATA DECLARATION UTILITY.....	8-49
8.6.1	SBDATA DECLARATION UTILITY.....	8-49
APPENDIX.....		A-1
APPENDIX A.ADDED FEATURES.....		A-1
A.1	FEATURES ADDED BETWEEN VER 1.00 RELEASE 1 AND VER 2.00 RELEASE 1 ...	A-1
A.2	FEATURES ADDED BETWEEN VER 2.00 RELEASE 1 AND VER 2.00 RELEASE 2 ..	A-3
A.3	FEATURES ADDED BETWEEN VER 2.00 RELEASE 2 AND VER 3.00 RELEASE 1 ..	A-4
A.4	FEATURES ADDED BETWEEN VER 3.00 RELEASE 1 AND VER 3.10 RELEASE 1 ..	A-7
A.5	FEATURES ADDED BETWEEN VER 3.10 RELEASE 1 AND VER 3.10 RELEASE 2 ..	A-8
A.6	FEATURES ADDED BETWEEN VER 3.10 RELEASE 2 AND VER 3.10 RELEASE 3 ..	A-9
A.7	FEATURES ADDED BETWEEN VER 3.10 RELEASE 3 AND VER 5.00 RELEASE 1	A-10
A.8	FEATURES ADDED BETWEEN VER 5.00 RELEASE 1 AND VER 5.10 RELEASE 1	A-12
A.9	FEATURES ADDED BETWEEN VER 5.10 RELEASE 1 AND VER 5.20 RELEASE 1	A-14

Section 1. Overview

1.1 Summary

The NC30WA and NC308WA compilers enable effective creation in the C language of programs that take advantage of the functions and performance of the Renesas Technology M16C family of single-chip microcomputers for embedded applications.

This document explains procedures for creating application programs using these C compilers.

1.1.1 Specification Summary

Language specification	ANSI standard
INT type	16 bits
Supported data types	8, 16, 32, and 64 bit integer types 32 and 64 bit floating-point types
Features	Multi-memory model Japanese character support High ROM efficiency

1.2 Features

The following table shows the features of the M16C family compilers NC30WA and NC308WA.

Performance	Top-class ROM efficiency.
Memory model	Memory can be specified in detail, using "near/far" specifications.
Extensibility	Robust support for embedded functionality, such as interrupt function and address declarations, using #pragma.
Regional characters	The EUC character encoding is supported, allowing string constants and comments to be specified in European and Asian languages.
Utilities	ROM compression utilities are supported.
Tools included	Integrated development environments (TM and High-performance Embedded Workshop) An assembler that supports structured code A simulator

1.3 Method of Installation

PC Version

To perform installation, start the installer and follow the instructions displayed. Be sure to check the license ID before starting installation, as it needs to be entered during installation.

The data entered during the installation process is used to create a user registration file (this file is only created for the PC version).

1.4 Method of Execution

1.4.1 Starting the Compiler

- Command input format for compile drivers

A compile driver runs the compiler commands, assembler commands, and link commands, and creates a machine language data file. The following information (input parameters) is needed to run a compile driver:

1. C source file
2. Assembly source file
3. Relocatable object file
4. Command line options (items specified as necessary)

Enter these items on the command line. Enter at least one item among 1, 2, and 3. Figure 1.1 shows the input format, and Figure 1.2 shows an input example. In the input example, the following operations are performed, and the absolute module file "sample.x30" is created:

1. The startup program "ncrt0.a30" is assembled.
2. The C source program "sample.c" is compiled/assembled.
3. The relocatable object file "ncrt0.a30" is linked to "sample.r30".

The start options are as follows:

- Option for specifying the name of the absolute module file "sample.x30": "-o"
- Option for specifying the list file (*.lst) output during assembly: "-as30 "-l"
- Option for specifying the map file (*.map) output during linking: "-ln30 "-ms"

```
% nc30{Δ}[start-options]{Δ}<[name-of-assembly-language-source-file]{Δ}
      [name-of-relocatable-object-file]{Δ}[name-of-C-source-file]>
```

%; Indicates the command prompt.

< >: Indicates required items.

[]: Indicates optional items.

Δ: Indicates a space.

Figure 1.1 Input Format for Compile Drivers

```
% nc30 -osample -as30 "-l" -ln30 "-ms" ncrt0.a30 sample.c{return}
```

{return}: Indicates the Return key being entered.

Be sure to specify the start-up program first when linking.

Figure 1.2 Input Example for the Compile Driver Command

1.4.2 Starting the Embedded Workshop

When installation is performed correctly, the Embedded Workshop is placed in a folder named [Renesas High-performance Embedded Workshop], in the [Programs] folder of the Windows [Start] menu, along with shortcuts to each executable program of the Embedded workshop. Note that the items displayed in the [Start] menu differ depending on which tools are installed.

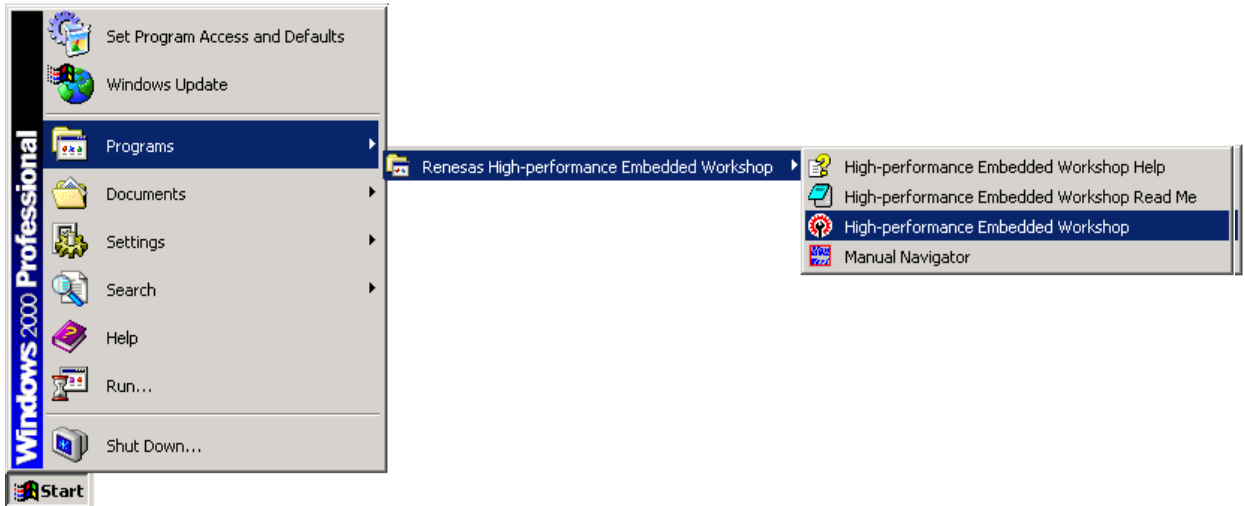


Figure 1.3 Starting the Embedded Workshop from the Start Menu

From the Start menu, when you click the Embedded Workshop, a start-up message is displayed, followed by the [Welcome!] dialog box (as shown in Figure 1.4).

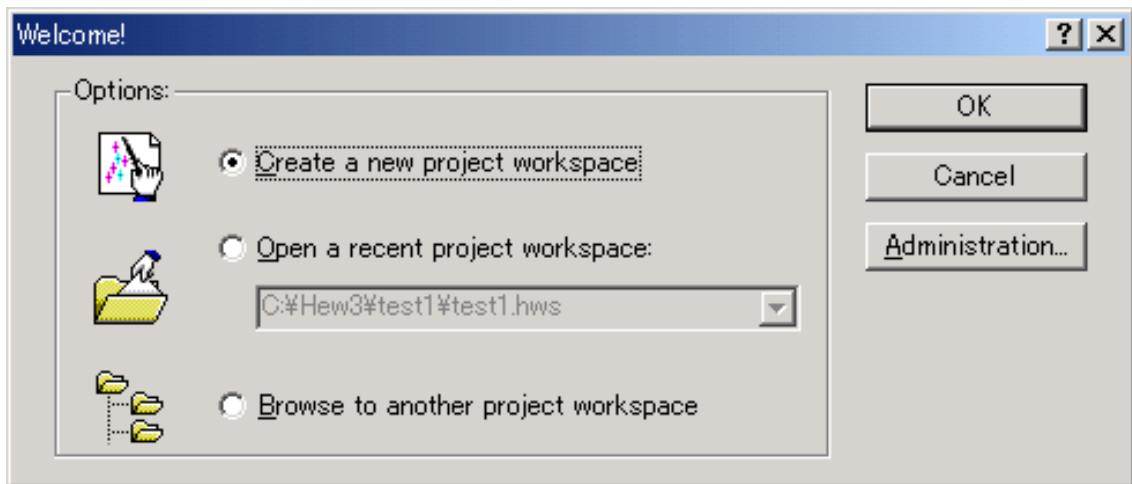


Figure 1.4 [Welcome!] Dialog Box

When using the Embedded Workshop for the first time, or creating a new project, select [Create a new workspace], and then click the [OK] button. Otherwise, to work on a project already created, select [Open a recent project workspace] or [Browse to another project workspace], and then click the [OK] button. You can also click the [Administration...] button to add or remove the system tools used with the Embedded Workshop.

1.5 Procedure for Program Development

Figure 1.5 shows an example flow for program development using NC308. The following is an overview of this program (items 1 through 4 correspond to those in Figure 1.5).

- (1) The C source program ("AA.c") is compiled by nc308 and assembled by assembler as308, and the relocatable object file ("AA.r30") is created.
- (2) The settings for section location, section size, and interrupt vector table are changed to match systems in which the include file "sect308.inc" is embedded. This file contains code for startup program "ncrt0.a30", as well as section information.
- (3) The changed startup program is assembled, and as a result, the relocatable object file "ncrt0.r30" is created.
- (4) The two relocatable object files, "AA.r30" and "ncrt0.r30", are linked by linkage editor ln308, which is executed from nc308. The absolute module file ("AA.x30") is then created.

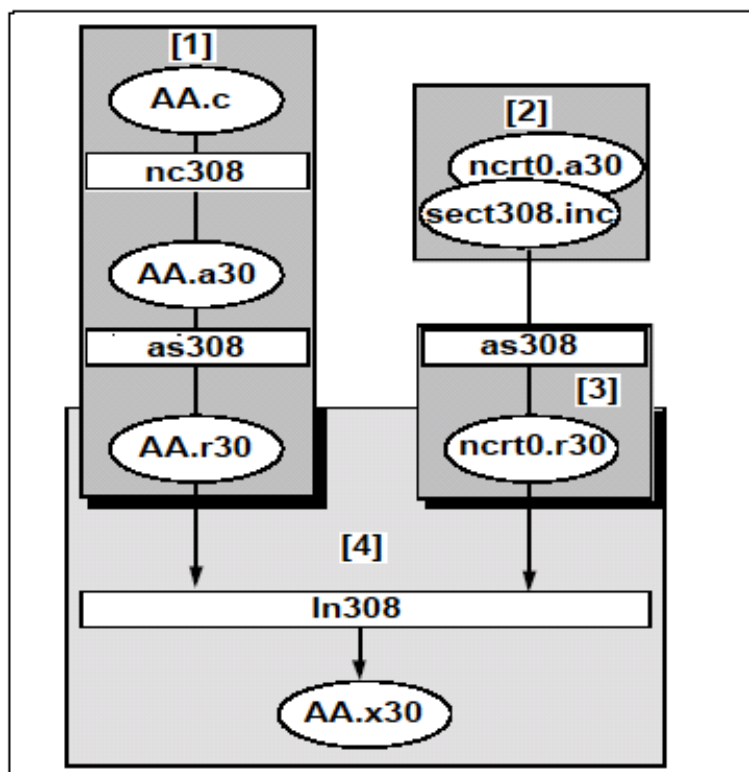


Figure 1.5 Flow of Program Development

MEMO

Section 2. Procedure for Creating a Program

2.1 Creating a Project

(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 can 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, in this dialog box. For example, when you enter "tutorial", in the [Workspace Name] field, then the [Project Name] field will show "tutorial" and the [Directory] field will show "C:\Hew3\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.

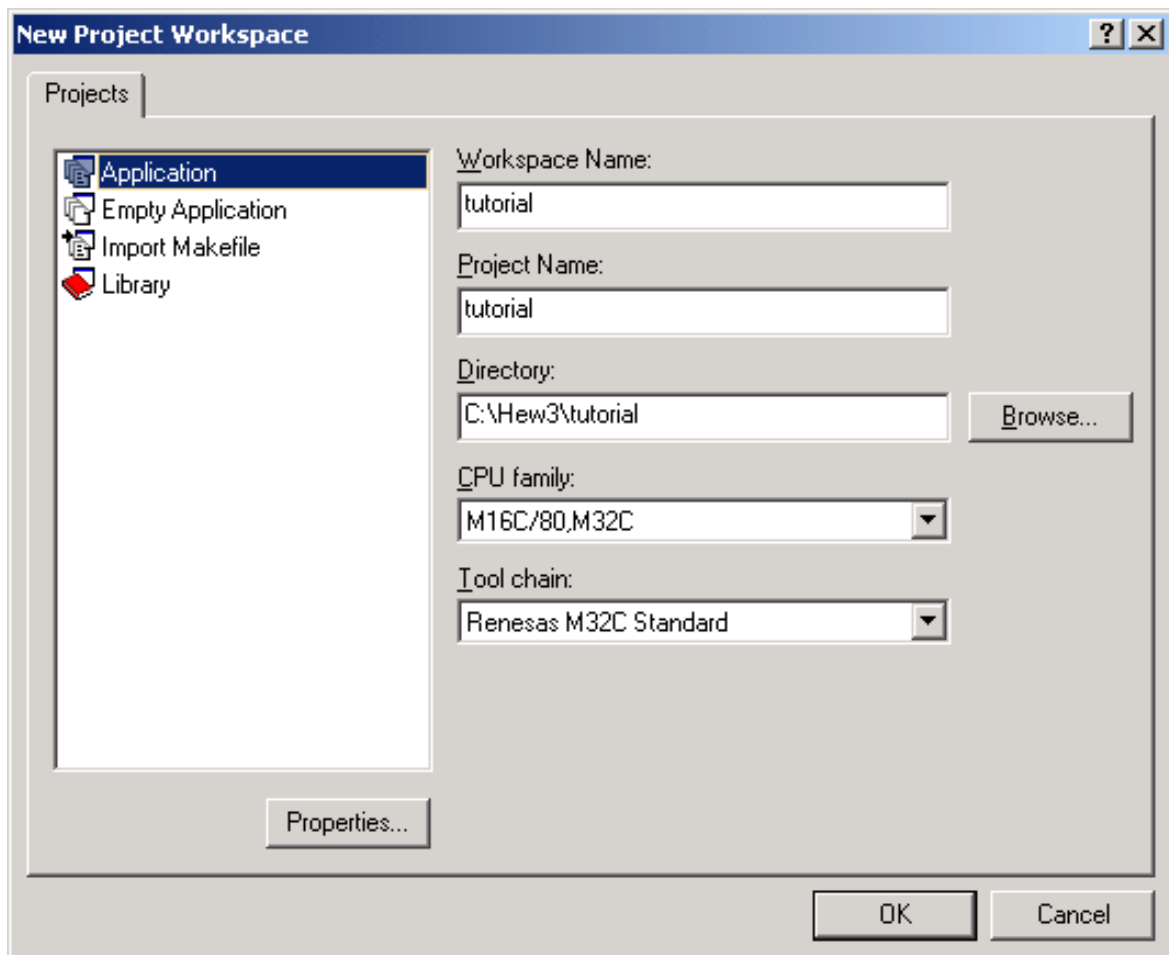


Figure 2.1 New Project Workspace Dialog Box

(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.

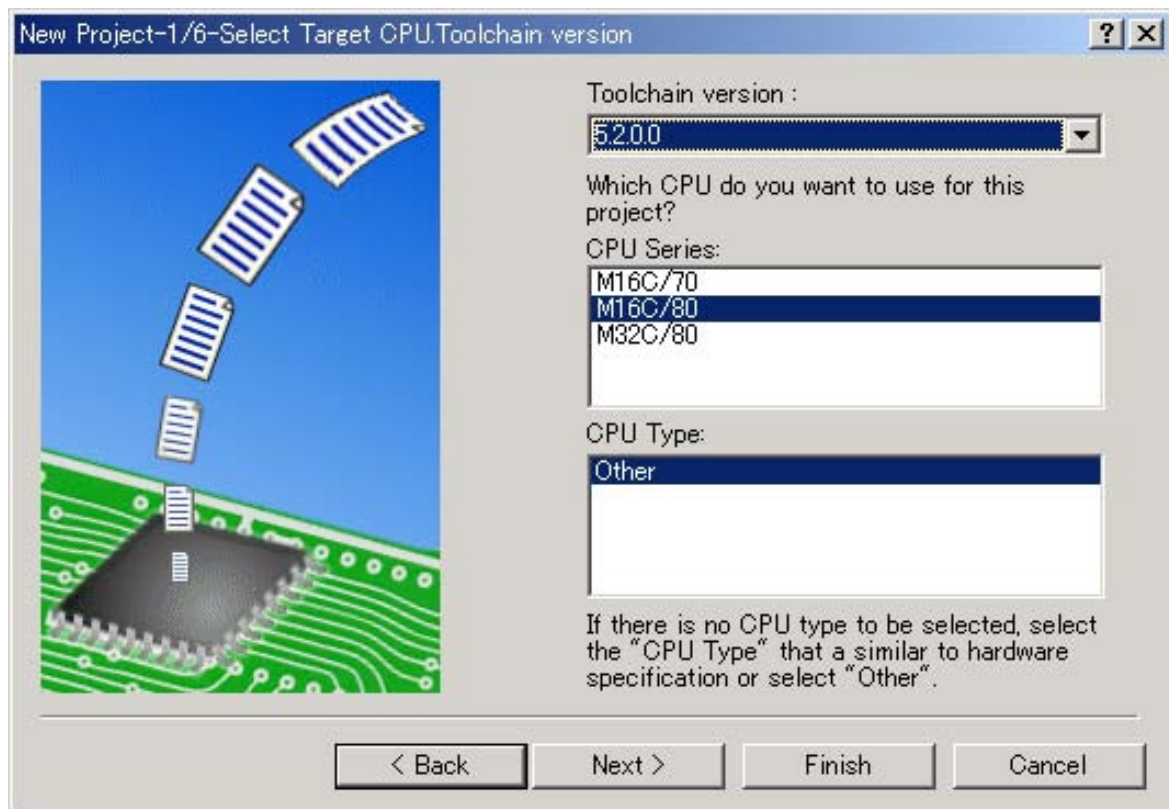


Figure 2.2 New Project Step 1 Dialog Box

(3) Selecting the RTOS

Clicking the [Next>] button on the Step-1 screen opens the dialog box shown in Figure 2.3. In this window, you can select whether or not to use an RTOS, as well as whether to use the default startup file or a user-defined file.

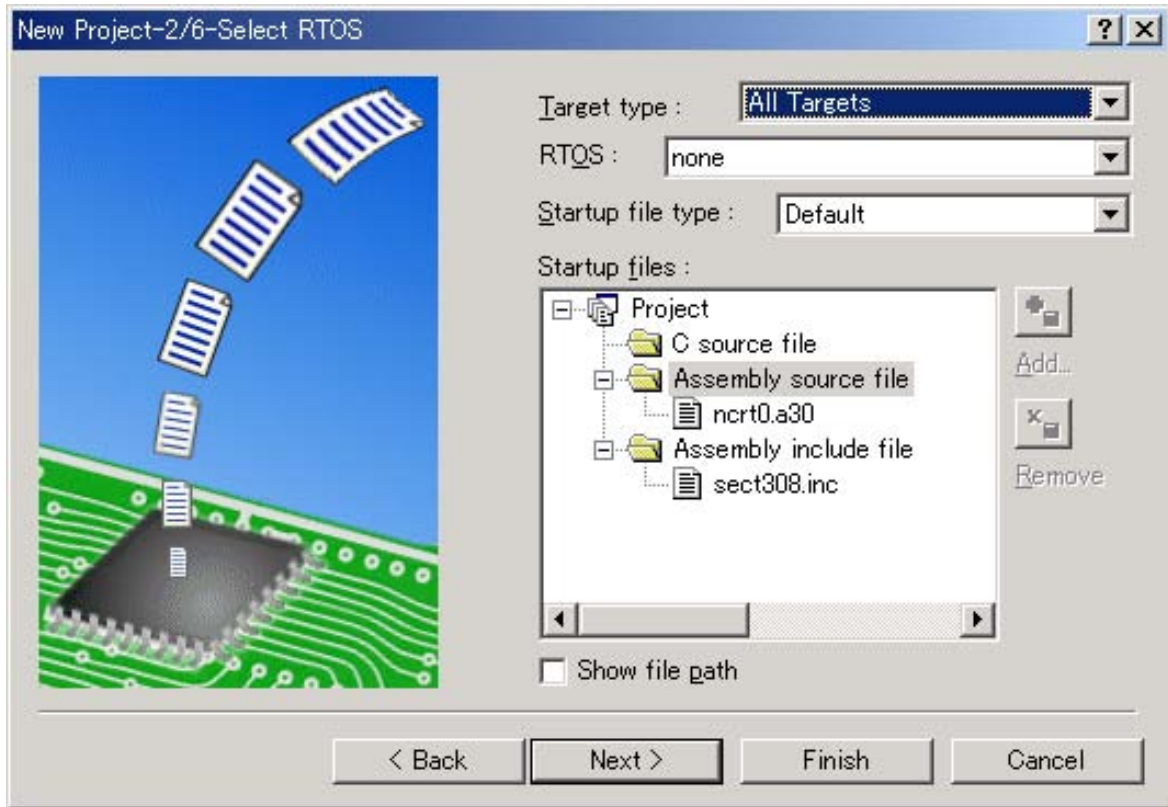


Figure 2.3 New Project Step 2 Dialog Box

(4) Setting the input/output library, and heap size

Clicking the [Next>] button on the Step-2 screen opens the dialog box shown in Figure 2.4. In this window, you can select whether or not to use the I/O library, which whether or not to generate a "main" function file, and the size of the heap area.

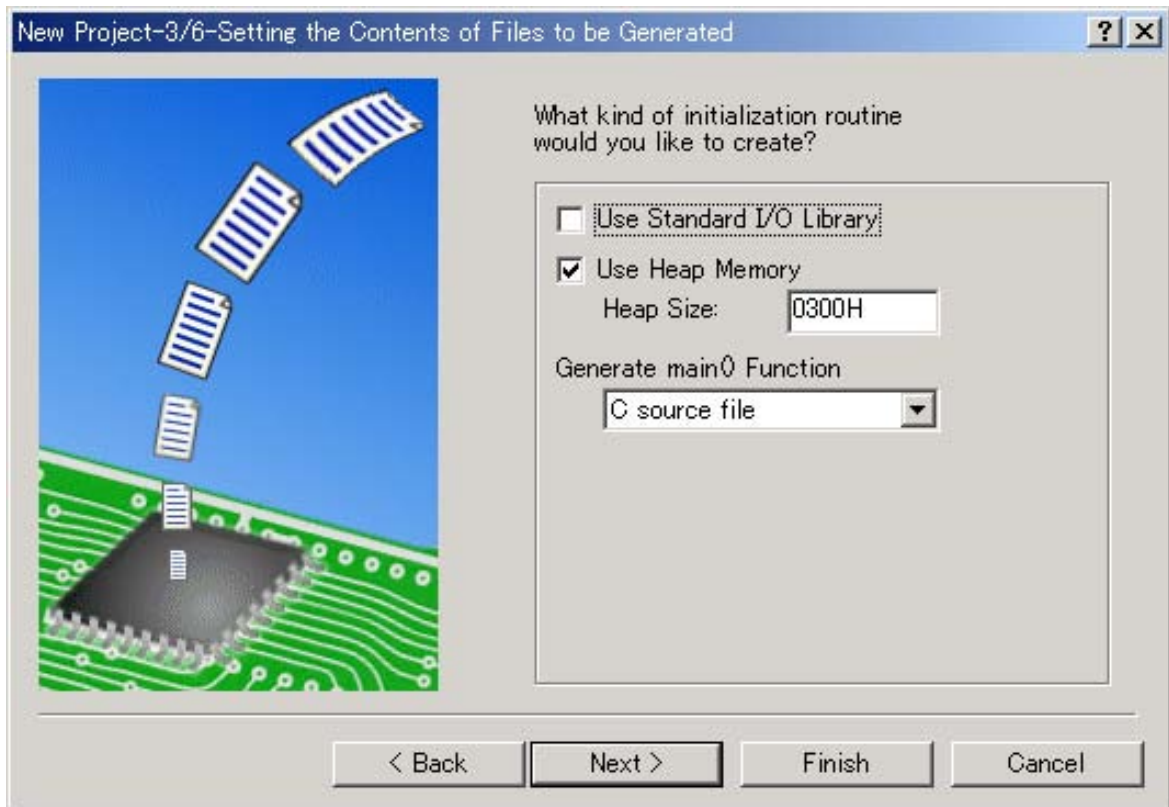


Figure 2.4 New Project Step 3 Dialog Box

(5) Setting the stack area

Clicking the [Next>] button on the Step-3 screen opens the dialog box shown in Figure 2.5. You can use this window to set the stack size and interrupt stack size.

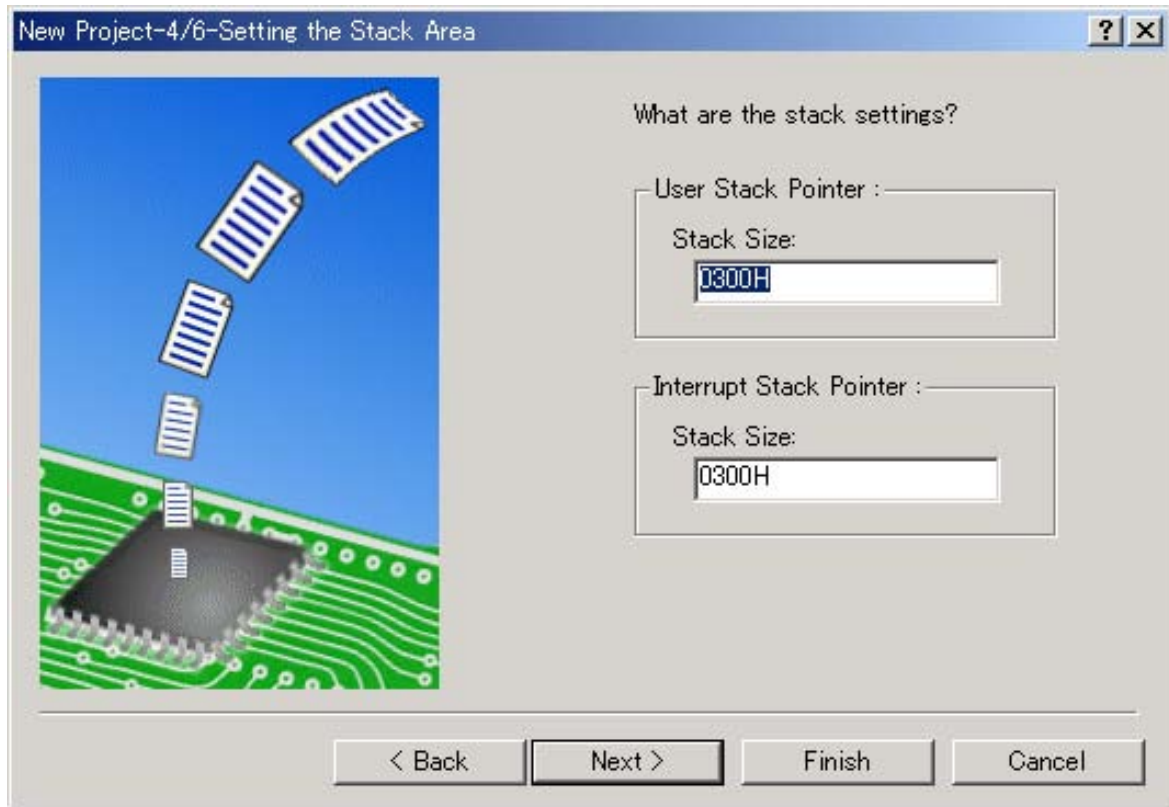


Figure 2.5 New Project Step 4 Dialog Box

(6) Setting the debugger options

When the [Next>] button is clicked in the Step-4 screen, the screen shown in figure 2.6 is displayed. You can use this window to perform settings for external package debuggers. Use this window to specify the debugger target, by selecting from [Targets:] the debugger target to be used. You may also leave the debugger target unselected.

Note that you can also select a debugger for an external package, if any exist.

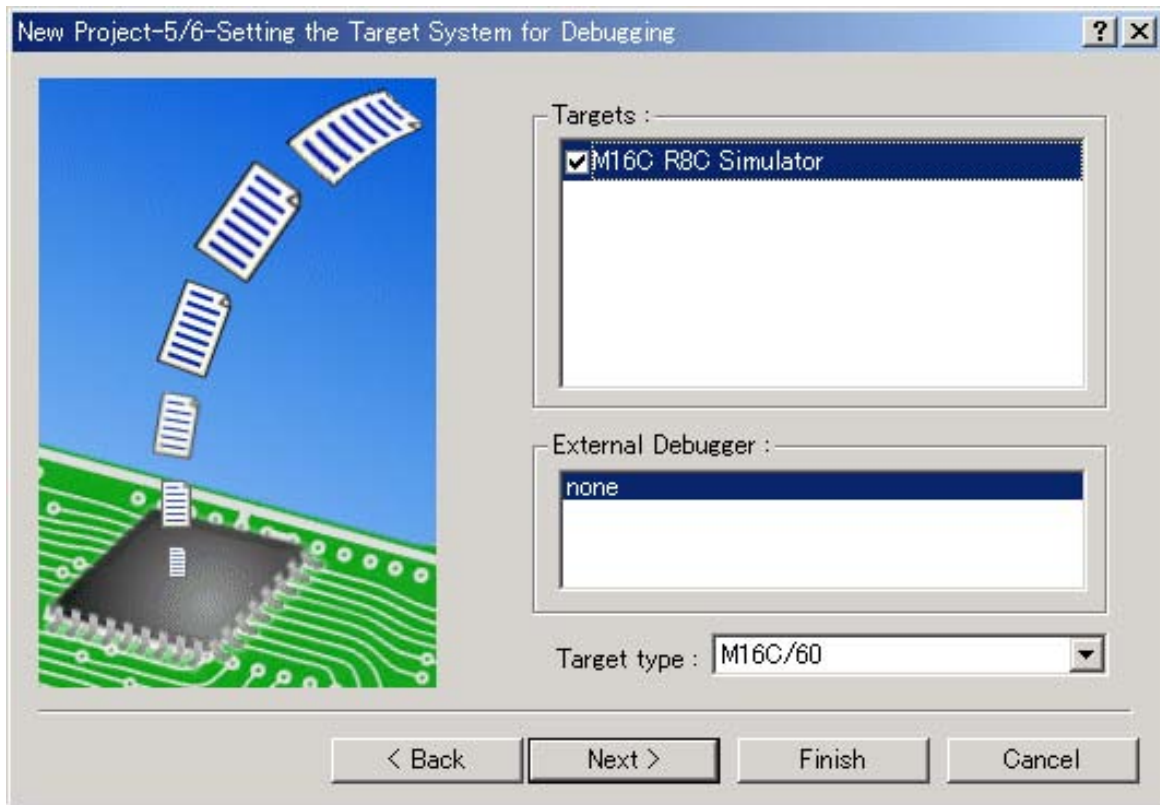


Figure 2.6 New Project Step 5 Dialog Box

(7) Setting the configuration file name

In the window for Step 5, click the [Next >] button to display the window shown in Figure 2.7.

In this window, set a configuration file name for each target selected.

A configuration refers to a file in which non-target High-performance Embedded Workshop statuses are saved.

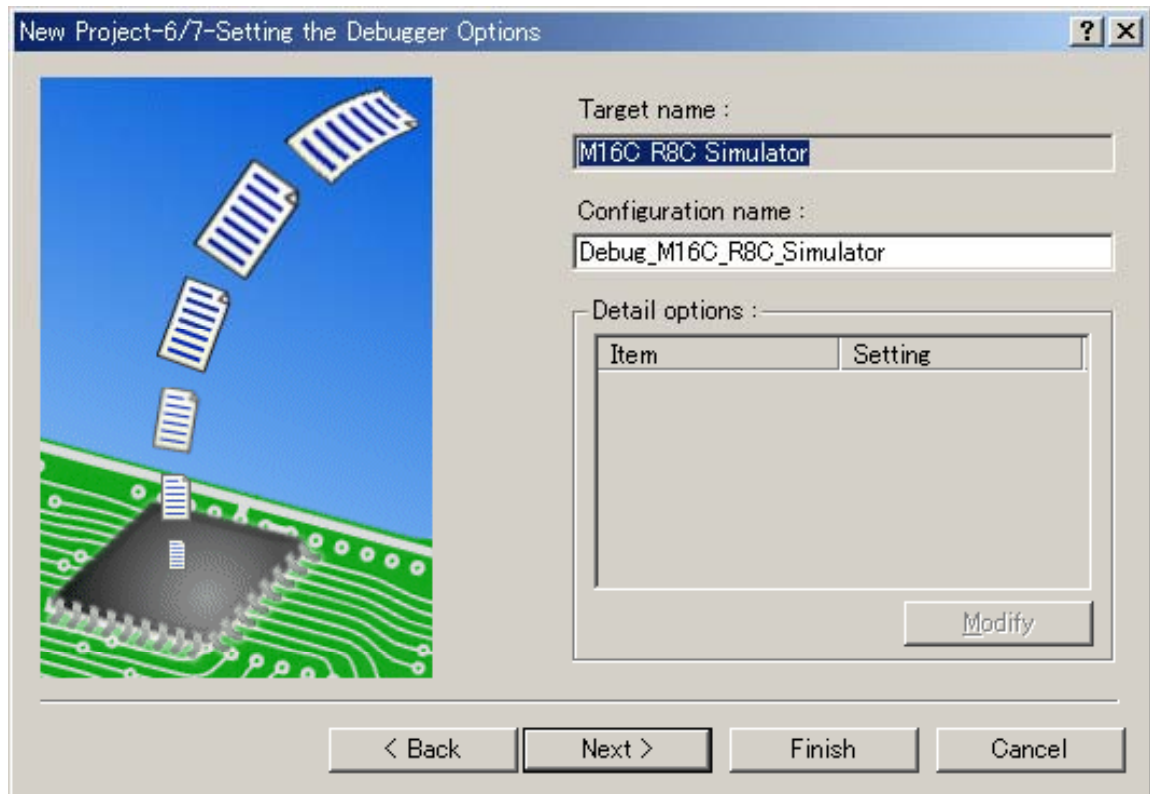


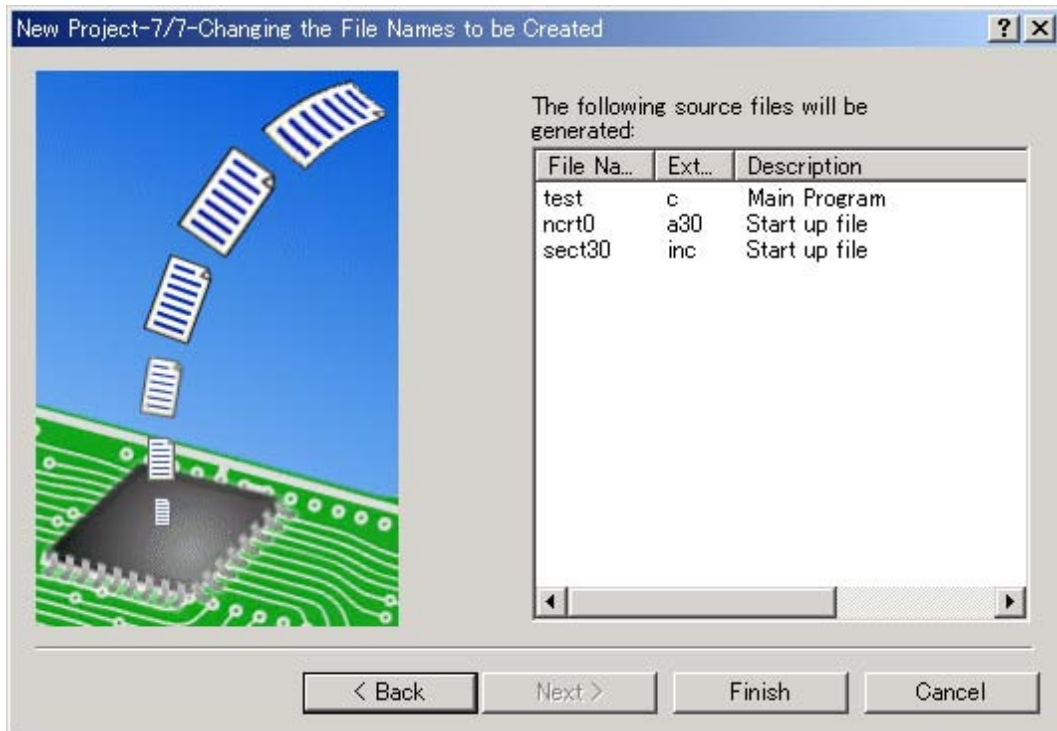
Figure 2.7 New Project Step 6 Dialog Box

(8) Confirming settings (Summary dialog box)

Clicking on [Next >] on the Step-6 screen displays the screen shown in Figure 2.8. This Window, displays the source file information for the project to be created. After confirmation, click [Finish].

When you click [Finish] on the screen in Figure 2.8, the project generator shows a list of generated files on the [Summary] dialog box (Figure 2.9). Confirm the contents of the dialog box and click [OK].

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



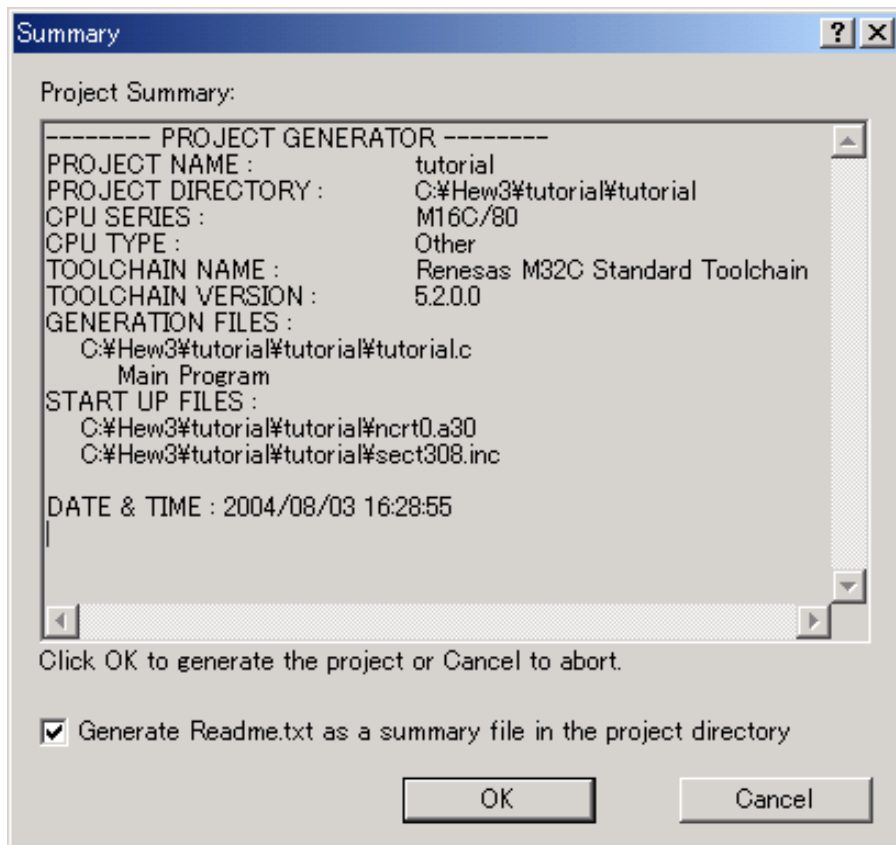


Figure 2.9 Summary Dialog Box

2.2 Start-up Programs

2.2.1 Purpose of Startup Programs

For built-in programs to run correctly, before processing, the microcomputer must be initialized, and the stack area must be set. Since such processing cannot usually be performed using C code, a program separate from the C source program is used that performs initialization and settings via assembly code. This is called a *startup program*. The following explains the sample startup programs "ncrt0.a30" and "sect308.inc", as prepared by NC308.

Purposes of the startup programs:

- ① Securing the stack area
- ② Performing initial settings for the microcomputer
- ③ Initializing the static variable space
- ④ Setting up the interrupt table register INTB
- ⑤ Calling the `main` function
- ⑥ Setting up the interrupt vector table

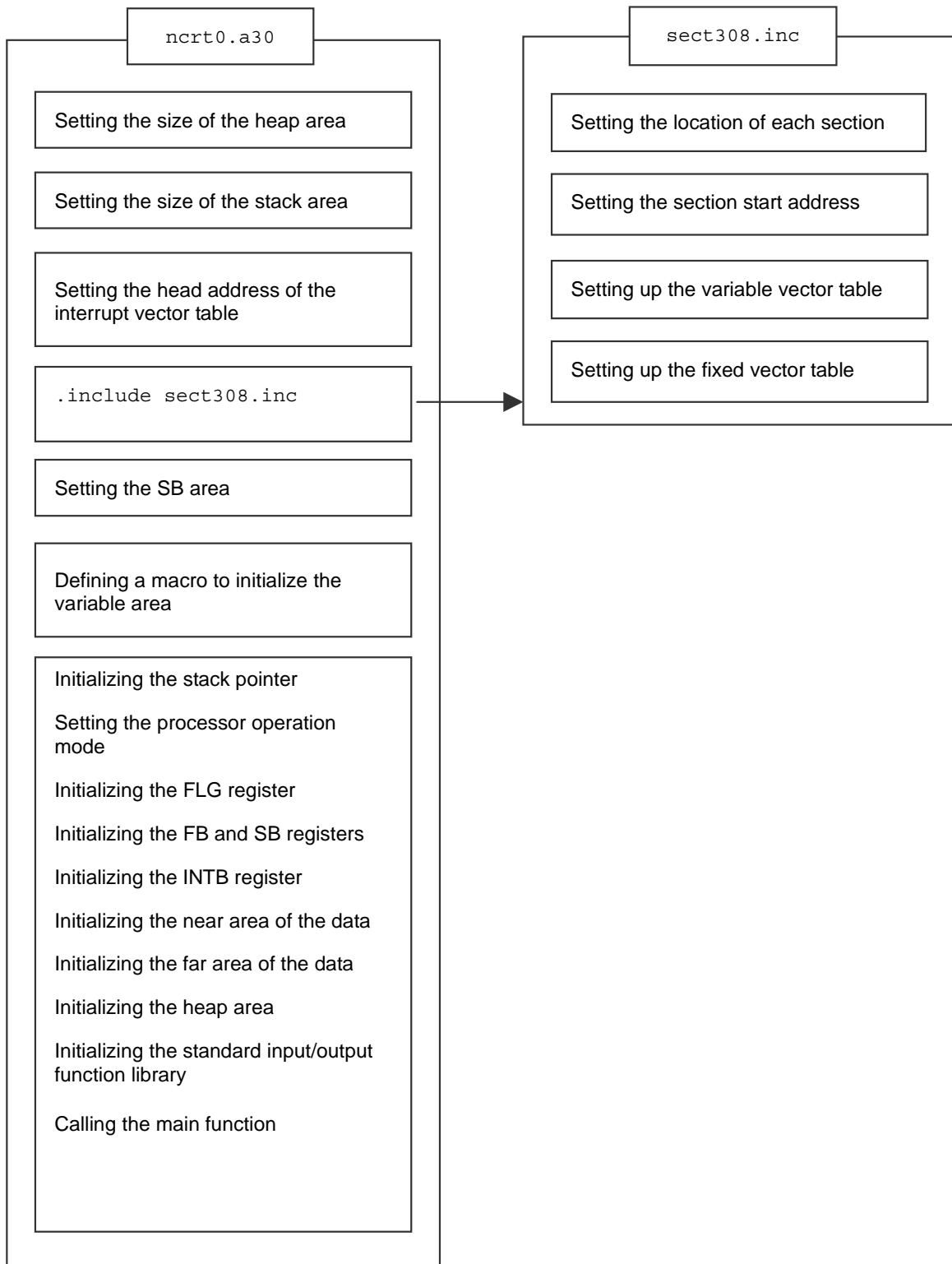


Figure 2.10 The Structure of a Startup Program

2.2.2 Setting Up a Startup Program

(1) Adding a section name

The sections created by NC308 are defined in the section definition file "sect308.inc". When you change a section name in "#pragma SECTION", the section base name created by NC308 is added. As a result, you must add such definitions in the section definition file "sect308.inc".

```
;-----  
; Arrangement of section  
;-----  
;-----  
; Near RAM data area  
;-----  
; SBDATA area  
    .section data_SE,DATA  
    .org 400H  
data_SE_top:  
;  
    .section bss_SE,DATA  
bss_SE_top:  
...  
    .section data_NO,DATA  
data_NO_top:  
;  
    .section new_data_NE,DATA  
new_data_NE_top:  
;  
    .section new_bss_NE,DATA  
new_bss_NE_top:  
...  
;-----  
; code area  
;-----  
    .section interrupt  
;  
    .section program  
;  
    .section new_program  
...  
;
```

Defining an additional section name changed in "#pragma SECTION."

Defining an additional section name changed in "#pragma SECTION."

Defining an additional section name changed in "#pragma SECTION."

Figure 2.11 Adding Section Names (sect308.inc)

When adding a data section or bss section, in addition to adding a definition of the section name, you need to include initial value transfer processing and zero clear processing for each area. So, be sure to add these to "ncrt0.a30".

```
=====
; NEAR area initialize.
;-----
; bss zero clear
;-----
    BZERO  bss_SE_top,bss_SE
    BZERO  bss_SO_top,bss_SO
    BZERO  bss_NE_top,bss_NE
    BZERO  bss_NO_top,bss_NO"
    BZERO  new_bss_NE_top,new_bss_NE
;-----
; initialize data section
;-----
    BCOPY  data_SEI_top,data_SE_top,data_SE
    BCOPY  data_SOI_top,data_SO_top,data_SO
    BCOPY  data_NEI_top,data_NE_top,data_NE
    BCOPY  data_NOI_top,data_NO_top,data_NO
    BCOPY  new_data_NEI_top,new_data_NE_top,new_data_NE
;
;
```

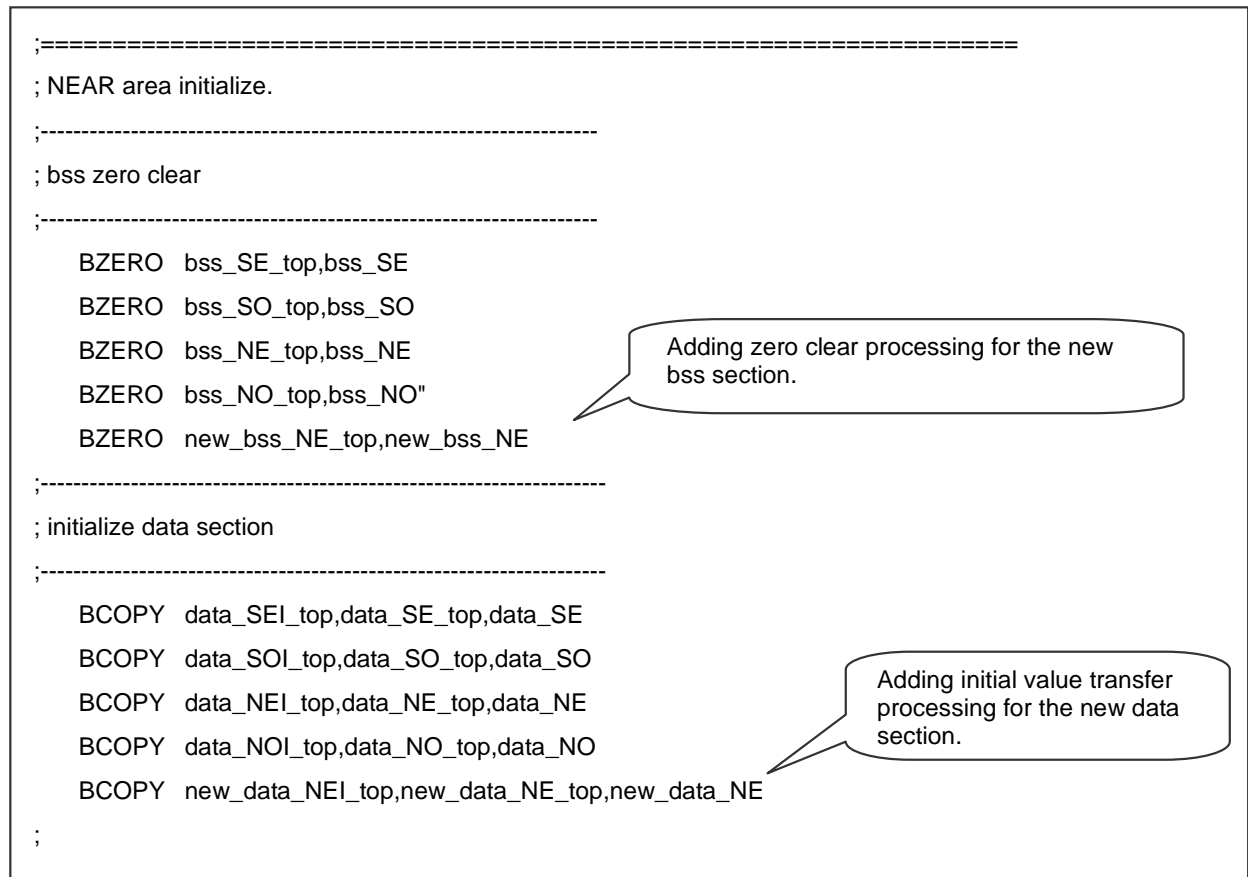


Figure 2.12 Adding Initialization Processing for the Added Sections

(2) Registering interrupt functions

To use interrupts correctly, you need to specify an interrupt processing function, and register it in the interrupt vector table. The following explains how to perform registration in the interrupt vector table.

To specify an interrupt processing function, perform registration by changing the interrupt vector table in the sample startup program "sect308.inc".

Perform the following changes to the interrupt vector table:

Use the ".glb" instruction command to make an external reference declaration for the interrupt processing function.

For the interrupt to be used, change the dummy function name dummy_int to the name of the interrupt processing function.

```
-----  
;   variable vector section  
-----  
    .section vector ; variable vector table  
    .org VECTOR_ADR  
;  
    .lword dummy_int ; vector (BRK)  
    .org ( VECTOR_ADR + 32 )  
    .lword dummy_int ; DMA0 (software int 8)  
    .lword dummy_int ; DMA1 (software int 9)  
    .lword dummy_int ; DMA2 (software int 10)  
    .lword dummy_int ; DMA3 (software int 11)  
    .glb _ta0  
    .lword _ta0 ; TIMER A0 (software int 12)  
    .lword dummy_int ; TIMER A1 (software int 13)  
    .lword dummy_int ; TIMER A2 (software int 14)  
    .lword dummy_int ; TIMER A3 (software int 15)  
    .lword dummy_int ; TIMER A4 (software int 16)  
    ...
```

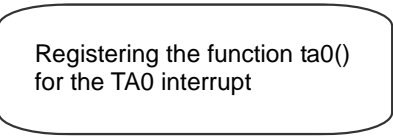


Figure 2.13 Interrupt Vector Table (sect308.inc)

Section 3. Compiler

3.1 Interrupt Functions

3.1.1 Coding Interrupt Processing Functions

NC308 allows you to code interrupt processing as C functions. The procedure consists of four steps.

- ① Coding interrupt processing functions
- ② Registering the functions on the interrupt vector table
- ③ Setting the interrupt enable flag (I flag)
 - Use the inline assembling.
- ④ Setting the priority level of the interrupt
 - Set the interrupt priority level before enabling the interrupt.

This subsection describes how to code the functions according to the type of interrupt processing.

- (1) Coding the hardware interrupt (#pragma INTERRUPT)

```
#pragma INTERRUPT interrupt-function-name
```

This declaration causes the compiler to generate the following instructions at the entry and exit points of the specified function, in addition to the regular function procedure: The instruction for saving and restoring all the registers used in the function and the reit instruction. Only the void type is available for parameters and return values of the interrupt processing function. If the declaration contains any other type, a warning will be output during compilation.

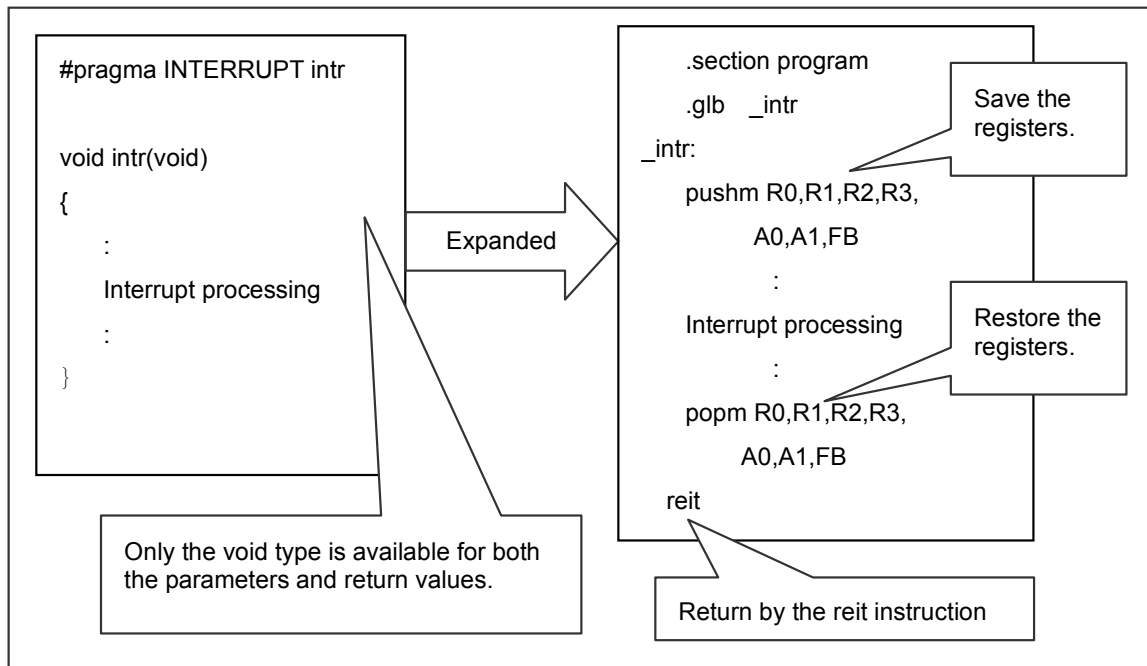


Figure 3.1 Expansion of the Interrupt Processing Function

(2) Coding the interrupt using the register bank (#pragma INTERRUPT/B)

For the M16C/80 Series, you can switch the register bank to reduce the time for starting the interrupt processing, while maintaining the contents of the registers. To use this functionality, declare the following coding:

```
#pragma INTERRUPT/B interrupt-function-name
```

The above coding causes the compiler to generate the instruction for switching the register bank, rather than the instructions for saving and restoring registers. However, you can only specify one interrupt because the M16C/80 Series product provides the register banks 0 and 1. Use this functionality for an interrupt that requires a faster startup.

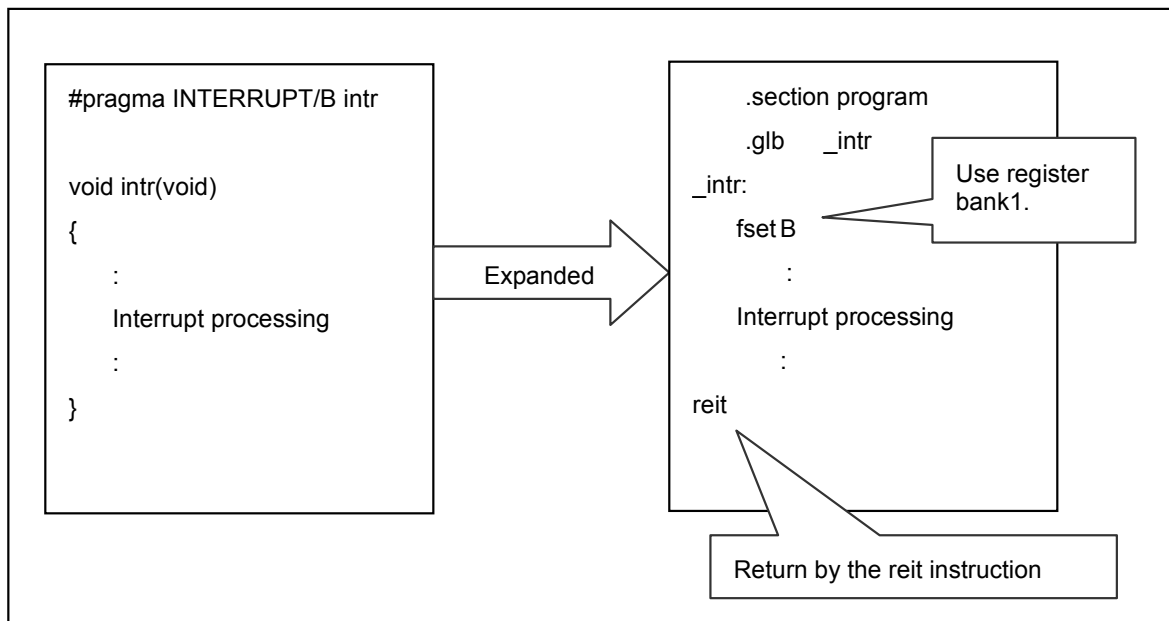


Figure 3.2 Expansion of the Interrupt Processing Function Using the Register Bank

(3) Coding the interrupt for enabling multiple interrupts (#pragma INTERRUPT/E)

When the M16C/80 Series compiler accepts an interrupt request, the interrupt enable flag (I flag) is set to "0", disabling interrupts. By setting the I flag to "1" at the entry point of the interrupt processing function (immediately after entering the interrupt processing function) to enable multiple interrupts, you can improve interrupt response. To use this functionality, code as follows:

```
#pragma INTERRUPT/E interrupt-function-name
```

The above coding causes the compiler to generate the instruction for setting the I flag to "1" at the entry point of the interrupt processing function (immediately after entering the interrupt processing function). However, if you want to enable multiple interrupts in the middle of the interrupt processing function, declare "#pragma INTERRUPT", and then use the asm() function in the middle of the interrupt processing function to set the I flag to "1".

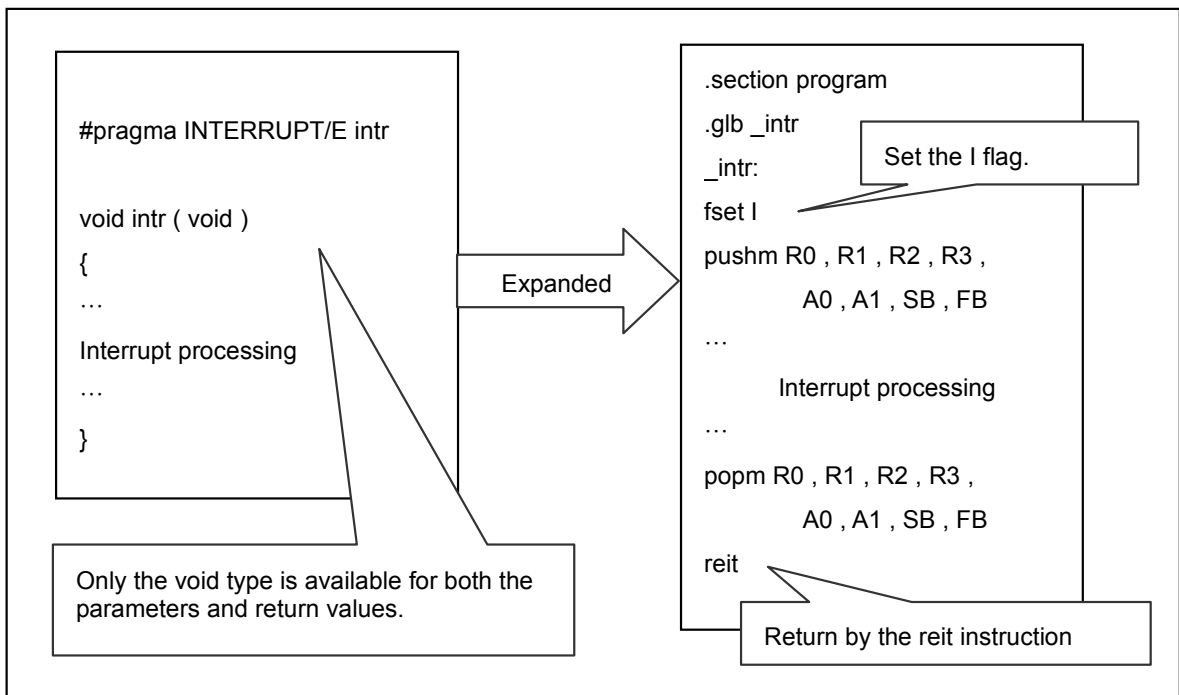


Figure 3.3 Expansion of the Interrupt Processing Function that Enables Multiple Interrupts

3.1.2 Coding Fast Interrupt Processing Functions

NC308 allows you to code fast interrupt processing as C functions that respond to an interrupt in five cycles, and return from the interrupt in three cycles. However, you can only set a single interrupt at the interrupt priority level 7 for the fast interrupt. The procedure consists of five steps.

- ① Coding fast interrupt processing functions
- ② Setting the interrupt priority level of the fast interrupt
 - Set the interrupt priority level before enabling the interrupt.
- ③ Setting the fast interrupt bit
- ④ Setting the vector register (VCT).
- ⑤ Setting the interrupt enable flag (I flag)
 - Use the inline assembler.

This subsection describes how to code the functions according to the type of fast interrupt processing.

- (1) Coding the fast hardware interrupt (`#pragma INTERRUPT/F`)

```
#pragma INTERRUPT/F interrupt-function-name
```

The above declaration causes the compiler to generate the following instructions at the entry and exit points of the specified function, in addition to the regular function procedure: Instructions for saving and restoring all the registers used in the function and the `freit` instruction for returning from the fast interrupt routine. Only the void type is available for parameters and return values of the interrupt processing function. If the declaration contains any other type, a warning will be output during compilation.

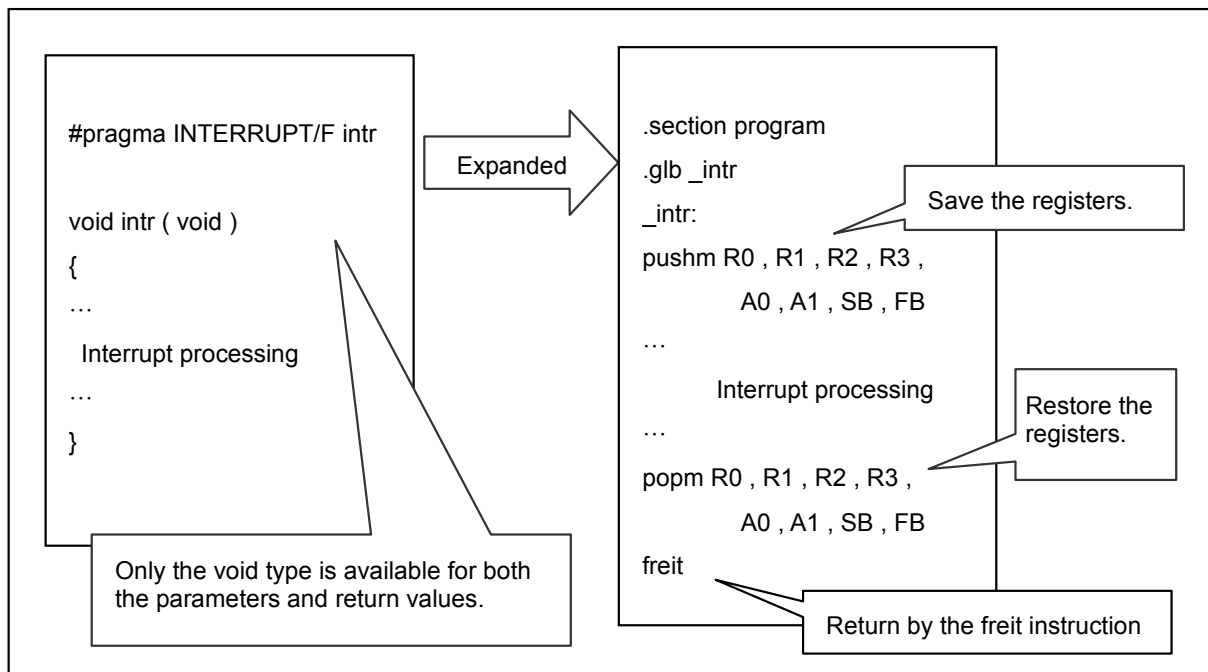


Figure 3.4 Expanding the Fast Interrupt Processing Function

(2) Coding the fast interrupt using the register bank (#pragma INTERRUPT/F/B)

For the M16C/80 Series, you can switch the register bank to reduce the time for starting the fast interrupt processing, while maintaining the contents of the registers. To use this functionality, declare the following coding:

```
#pragma INTERRUPT/F/B interrupt-function-name
```

The above coding causes the compiler to generate the instruction for switching the register bank, rather than the instructions for saving and restoring registers. However, you can only specify one interrupt because the M16C/80 Series product provides the register banks 0 and 1. Use this functionality for an interrupt that requires the fastest startup.

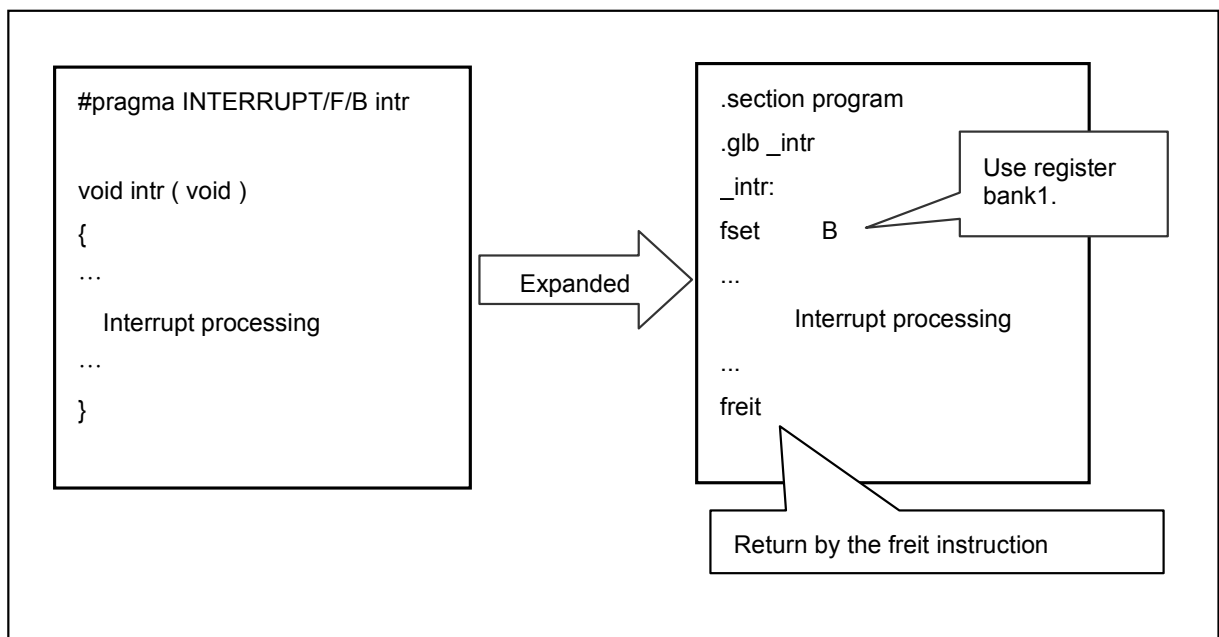


Figure 3.5 Expansion of the Fast Interrupt Processing Function Using the Register Bank

3.1.3 Coding Functions for Software Interrupt (INT Instruction) Processing

- (1) Coding the software interrupt that calls assembly language functions (#pragma INTCALL)

To use a software interrupt (INT instruction) for the M16C/80 Series, specify "#pragma INTCALL". This function allows you to generate a pseudo interrupt during debugging.

The coding method differs depending on whether the body of the function to be called by the software interrupt is written in the assembly language or in C.

If the body of the function to be called is written in the assembly language, code as follows:

```
#pragma INTCALL software-interrupt-number assembly-language-function-name
(register-name, register, ...)
```

If the body of the function to be called is written in the assembly language, you can pass the parameters via registers. You can also receive return values of other than the structure or union type.

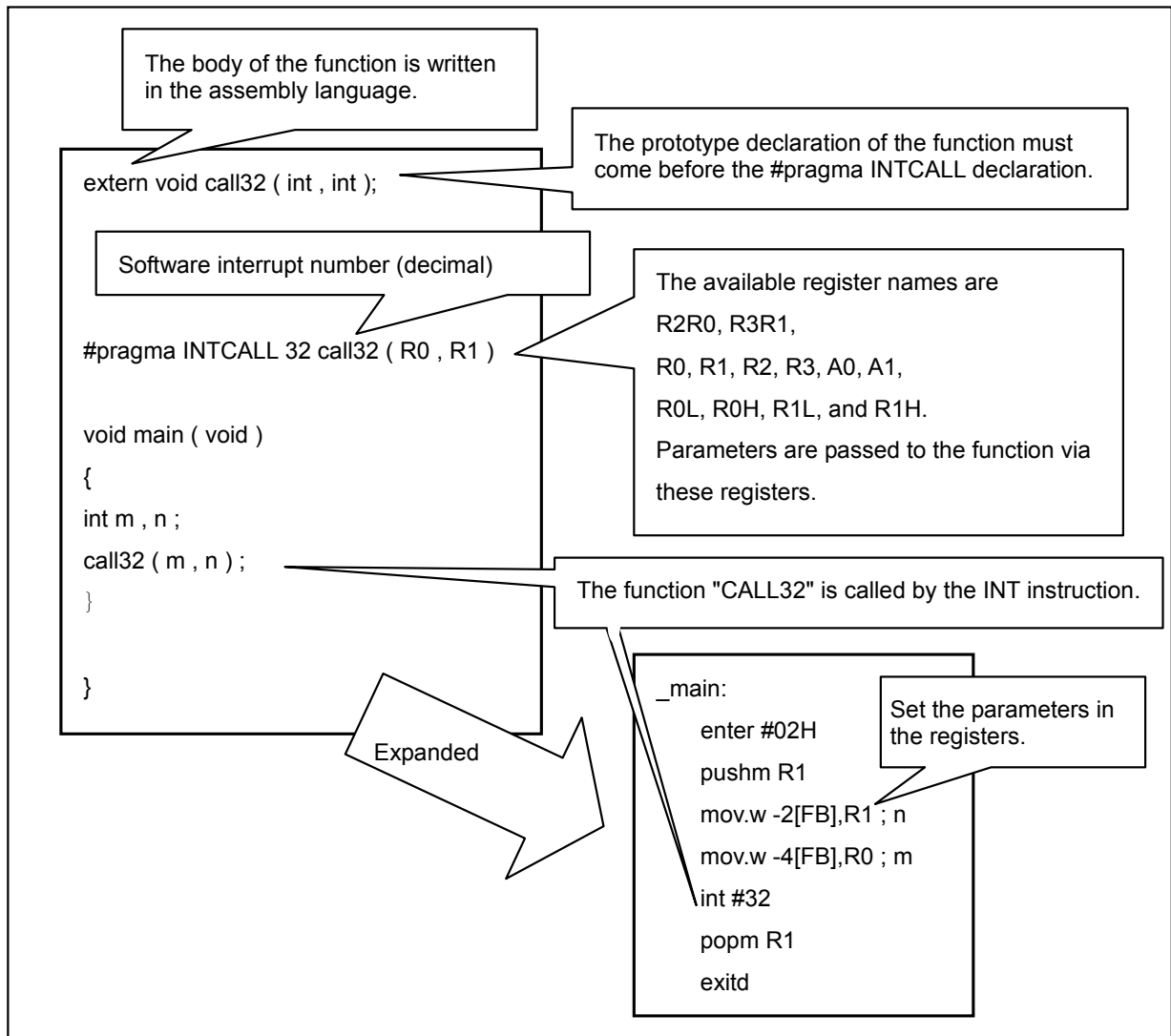


Figure 3.6 Coding Example of #pragma INTCALL that Calls the Assembly Language Function

(2) Coding the software interrupt that calls C functions (#pragma INTCALL)

If the body of the function to be called by the software interrupt (INT instruction) is written in C, code as follows:

```
#pragma INTCALL software-interrupt-number C-function-name ()
```

If the body of the function to be called is written in C, you can only specify the functions in which all parameters are passed via the registers, according to the parameter passing rules. The coding cannot include any parameter of the functions that declare "#pragma INTCALL". Return values of other than the structure or union type can be received.

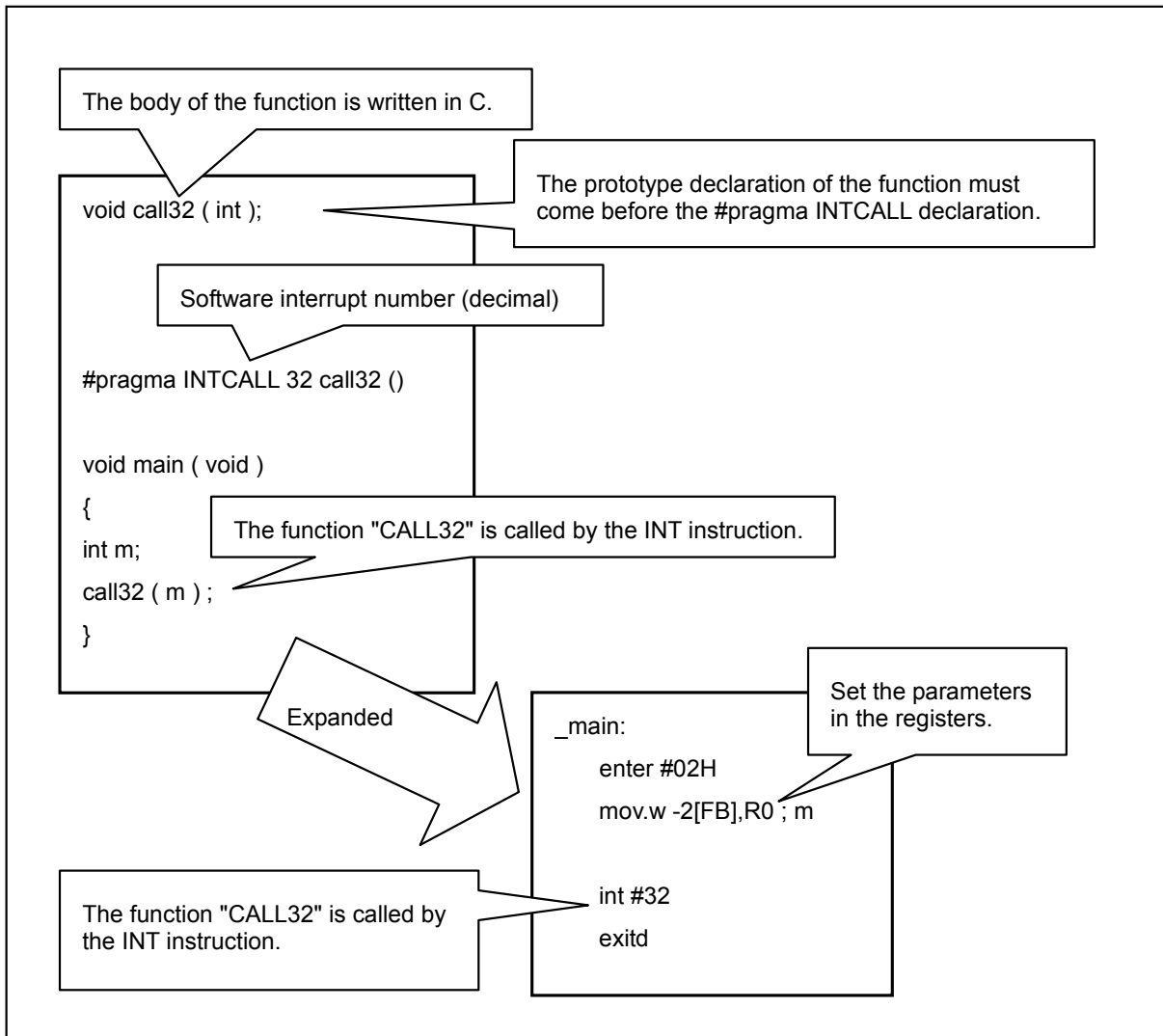


Figure 3.7 Coding #pragma INTCALL that Calls the C Function

3.1.4 Registering Interrupt Processing Functions

To use interrupts normally, you must specify interrupt processing functions and register them to an interrupt vector table.

This subsection describes how to register interrupt processing functions to the interrupt vector table.

(1) Registering the functions in the interrupt vector table

When you specify an interrupt processing function, register it by changing the interrupt vector table in the sample startup program "sect308.inc".

To change the interrupt vector table:

- ① Use the .glb instruction to declare the name of the interrupt processing function for external reference.
- ② Change the dummy function name "dummy_int" for the interrupt to the name of the interrupt processing function.

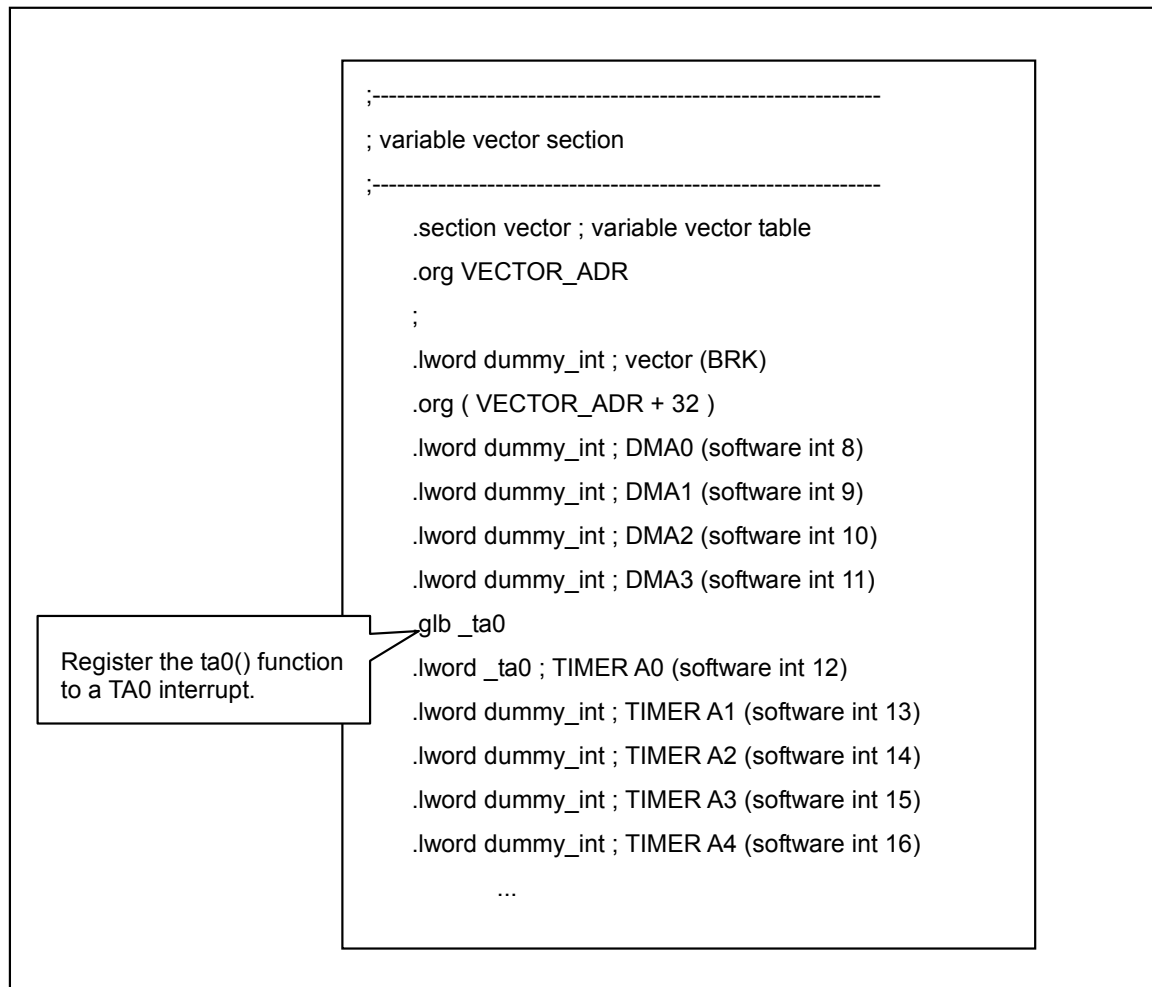


Figure 3.8 Interrupt Vector Table (sect308.inc)

3.1.5 Coding Example of Interrupt Processing Functions

This subsection provides an example of coding a program that clears the contents of "counter" to zero each time an INT0 interrupt occurs, and increases the contents of "counter" each time an INT1 interrupt occurs.

- (1) Coding example of interrupt processing functions

```
/* Prototype declaration *****/
void int0 ( void );
void int1 ( void );
#pragma INTERRUPT/F int0
#pragma INTERRUPT int1
/*****/
unsigned int counter ;
void int0 ( void ) /* Fast interrupt function */
{
    counter = 0 ;
}
void int1 ( void ) /* Interrupt function */
{
    if ( counter < 9 ) {
        counter ++ ;
    }
    else {
        counter = 0 ;
    }
}

void main ( void )
{
    INT0IC = 0x07; /* Setting the fast interrupt priority level */
    RLVL = 0x08; /* Fast interrupt specification */
    asm ( " LDC #_int0,VCT " ); /* Setting the vector register */
    INT1IC = 0x01 ; /* Setting the interrupt priority level */
    asm ( " fset i " ); /* Interrupt enabled */
    while (1) ; /* Interrupt wait loop */
}
```

Figure 3.9 Coding Example of Interrupt Processing Functions

(2) Registering the functions in the interrupt vector table

Figure 3.10 shows an example of registering the functions in the interrupt vector table.

```
;-----  
; variable vector section  
;-----  
.section vector ; variable vector table  
.org VECTOR_ADR  
...  
.org ( VECTOR_ADR + 32 )  
.lword dummy_int ; DMA0 (software int 8)  
.lword dummy_int ; DMA1 (software int 9)  
.lword dummy_int ; DMA2 (software int 10)  
.lword dummy_int ; DMA3 (software int 11)  
.lword dummy_int ; TIMER A0 (software int 12)  
.lword dummy_int ; TIMER A1 (software int 13)  
.lword dummy_int ; TIMER A2 (software int 14)  
.lword dummy_int ; TIMER A3 (software int 15)  
.lword dummy_int ; TIMER A4 (software int 16)  
.lword dummy_int ; uart0 trance (software int17)  
.lword dummy_int ; uart0 receive (software int18)  
.lword dummy_int ; uart1 trance (software int19)  
.lword dummy_int ; uart1 receive (software int 20)  
.lword dummy_int ;TIMER B0 (software int 21)  
.lword dummy_int ;TIMER B1 (software int 22)  
.lword dummy_int ;TIMER B2 (software int 23)  
.lword dummy_int ;TIMER B3 (software int 24)  
.lword dummy_int ;TIMER B4 (software int 25)  
.lword dummy_int ; INT5 (software int 26)  
.lword dummy_int ; INT4 (software int 27)  
.lword dummy_int ; INT3 (software int 28)  
.lword dummy_int ; INT2 (software int 29)  
.glb _int1  
.lword _int1 ; INT1 (software int 30)  
.glb _int0  
.lword _int0 ; INT0 (software int 31)  
.lword dummy_int ; TIMER B5 (software int 32)  
...
```

Figure 3.10 Example of Registering the Functions in the Interrupt Vector Table

3.2 Assembler Macro

NC308 allows you to code some assembly language instructions as C functions (these functions are called assembler macro functions).

The assembler macro functions let you directly code, in C programs, assembly language instructions that NC308 does not expand in ordinary C coding. This enables easier tuning of programs.

This subsection describes how to specify and use the assembler macro functions.

3.2.1 Assembly Language Instructions that can Be Specified Using Assembler Macro Functions

NC308 allows you to use assembler macro functions to specify 18 types of assembly language instructions.

The assembler macro function names are the same as the assembly language instructions shown in lowercase letters. These function names are followed by "_b", "_w", or "_l" indicating the bit length for operations. Tables 3.1 and 3.2 show the assembly language instructions that can be specified using the assembler macro functions.

Table 3.1 Assembly Language Instructions that can Be Specified Using Assembler Macro Functions (1)

Assembly language instruction	Assembler macro function name	Description	Format
DADD	dadd_b	Returns the result of decimal addition on val1 plus val2.	char dadd_b(char val1,char val2)
	dadd_w		int dadd_w(int val1,int val2)
DADC	dadc_b	Returns the result of decimal addition with carry on val1 plus val2.	char dadc_b(char val1,char val2)
	dadc_w		int dadc_w(int val1,int val2)
DSUB	dsub_b	Returns the result of decimal subtraction on val1 minus val2.	char dsub_b(char val1, char val2);
	dsub_w		int dsub_w(int val1, int val2);
DSBB	dsbb_b	Returns the result of decimal subtraction with borrow on val1 minus val2.	char dsbb_b(char val1, char val2);
	dsbb_w		int dsbb_w(int val1, int val2);
RMPA	rmpa_b	Returns the result of a sum-of-products operation, using init as the initial value, count as the number of times, and p1 and P2 as the start addresses where multipliers are stored.	long rmpa_b(long init, int count, char *p1, char *p2);
	rmpa_w		long rmpa_w(long init, int count,int *p1, int *p2);
MAX	max_b	Returns the value val1 or val2 whichever is found larger.	char max_b(char val1, char val2);
	max_w		int max_w(int val1, int val2);

Assembly language instruction	Assembler macro function name	Description	Format
MIN	min_b	Returns the value val1 or val2 whichever is found smaller.	char min_b(char val1, char val2);
	min_w		int min_w(int val1, int val2);

Table 3.2 Assembly Language Instructions that can Be Specified Using Assembler Macro Functions (2)

Assembly language instruction	Assembler macro function name	Description	Format
SMOVB	smovb_b	Transfers strings from the address p1 to the address p2 as many times as indicated by count in the backward direction.	void smovb_b(char *p1, char *p2, unsigned int count);
	smovb_w		void smovb_w(int *p1, int *p2, unsigned int count);
SMOVF	smovf_b	Transfers strings from the address p1 to the address p2 as many times as indicated by count in the forward direction.	void smovf_b(char *p1, char *p2, unsigned int count);
	smovf_w		void smovf_w(int *p1, int *p2, unsigned int count);
SMOVU	smovu_b	Transfers strings from the address p1 to the address p2 as many times as indicated by count in the forward direction until zero is detected.	void smovu_b(char *p1, char *p2);
	smovu_w		void smovu_w(int *p1, int *p2);
SIN	sin_b	Transfers strings from the fixed address p1 to the address p2 as many times as indicated by count in the forward direction.	void sin_b(char *p1, char *p2, unsigned int count);
	sin_w		void sin_w(int *p1, int *p2, unsigned int count);
SOUT	sout_b	Transfers strings from the address p1 to the address p2 as many times as indicated by count in the backward direction	void sout_b(char *p1, char *p2, unsigned int count);
	sout_w		void sout_w(int *p1, int *p2, unsigned int count);
SSTR	sstr_b	Stores strings, using data val to be stored, address p, and count as the number of times to transfer data.	void sstr_b(char val, char *p, unsigned int count);
	sstr_w		void sstr_w(int val, int *p, unsigned int count);
ROLC	rolc_b	Returns the value of val after rotating it left by 1 bit including carry.	unsigned char rolc_b(unsigned char val);
	rolc_w		unsigned int rolc_w(unsigned int val);

Assembly language instruction	Assembler macro function name	Description	Format
RORC	rorc_b	Returns the value of val after rotating it right by 1 bit including carry.	unsigned char rorc_b(unsigned char val);
	rorc_w		unsigned int rorc_w(unsigned int val);
ROT	rot_b	Returns the value of val after rotating it as many times as indicated by count.	unsigned char rot_b(signed char count,unsigned char val);
	rot_w		unsigned int rot_w(signed char count,unsigned int val);
SHA	sha_b	Returns the value of val after arithmetically shifting it as many times as indicated by count.	unsigned char sha_b(signed char count, unsigned char val);
	sha_w		unsigned int sha_w(signed char count, unsigned int val);
	sha_l		unsigned long sha_l(signed char count, unsigned longval);
SHL	shl_b	Returns the value of val after logically shifting it as many times as indicated by count.	unsigned char shl_b(signed char count, unsigned char val);
	shl_w		unsigned int shl_w(signed char count, unsigned int val);
	shl_l		unsigned long shl_l(signed char count, unsigned longval);

3.2.2 Decimal Addition Using the Assembler Macro Function "dadd_b"

When you call and use the NC308 assembler macro functions, you must include the "asmmacro.h" file that defines the assembler macro functions.

Figure 3.11 shows an example of decimal addition using the assembler macro function "dadd_b".

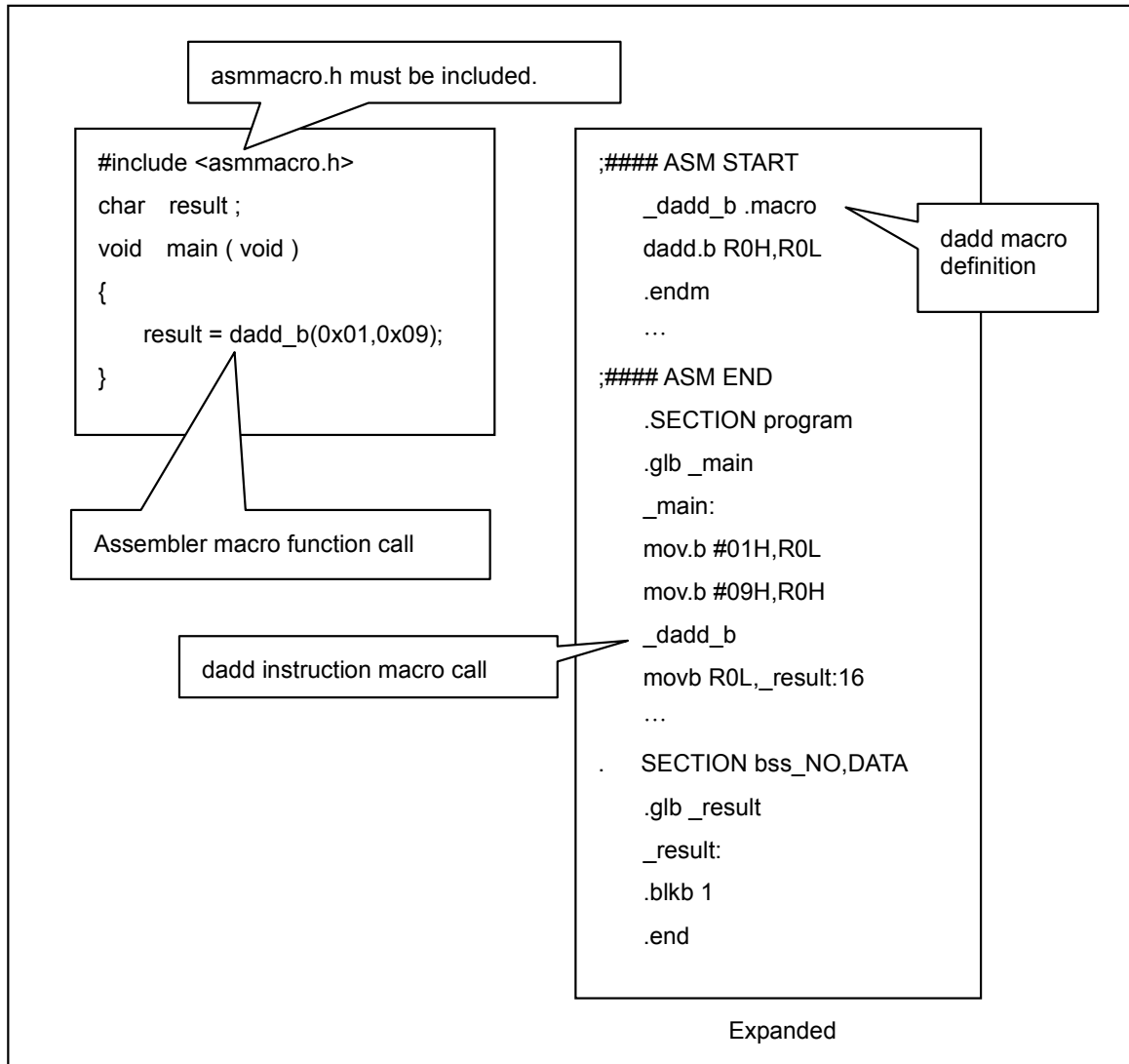


Figure 3.11 Decimal Addition Using the Assembler Macro Function "dadd_b"

3.2.3 Transferring Strings Using the Assembler Macro Function "smovf_b"

Figure 3.12 shows an example of transferring strings using the assembler macro function "smovf_b".

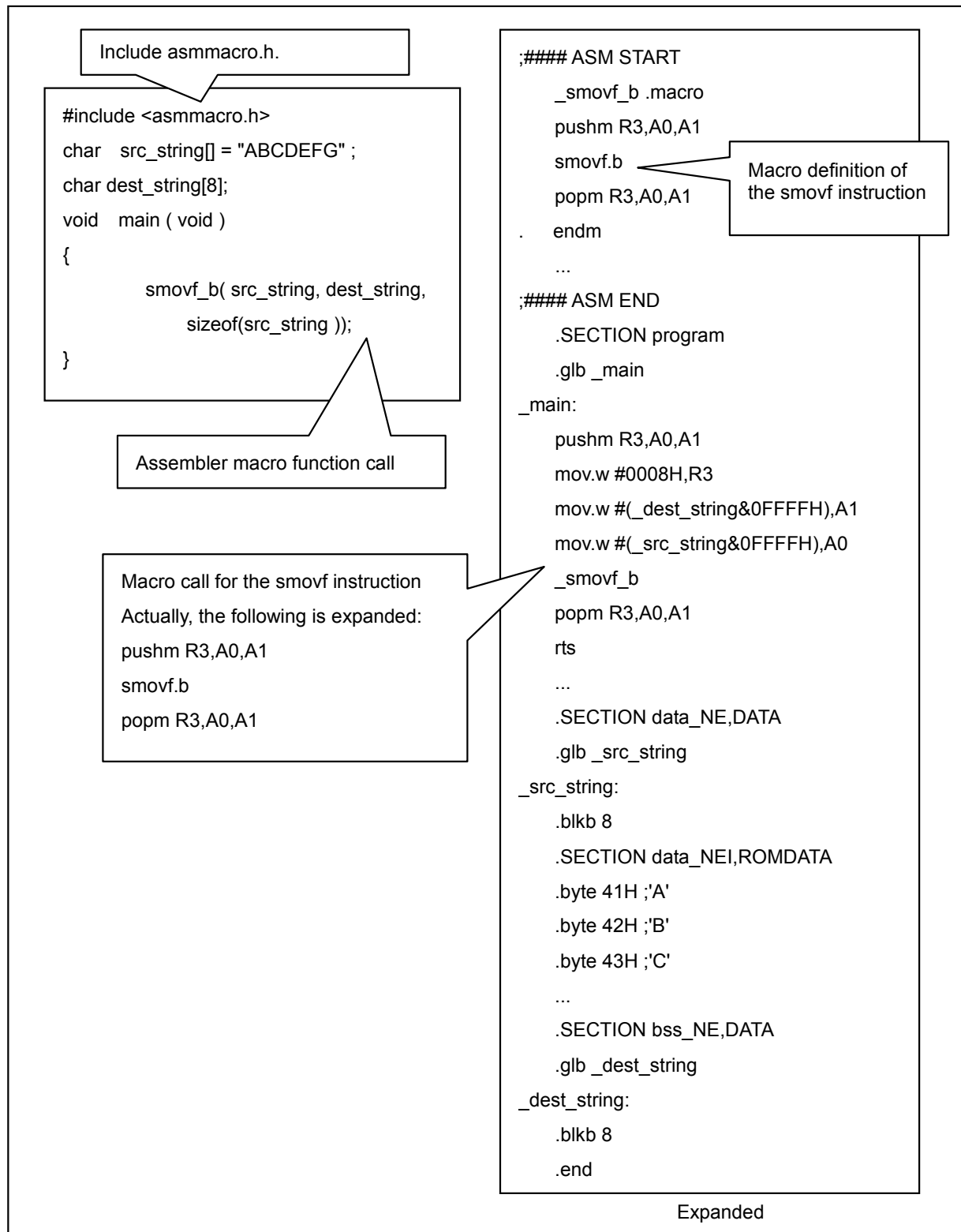


Figure 3.12 Transferring Strings Using the smovf_b Assembler Macro Function

3.2.4 Sum-of-Products Operation Using the Assembler Macro Function "rmpa_w"

Figure 3.12 shows an example of sum-of-products operation using the assembler macro function "rmpa_w".

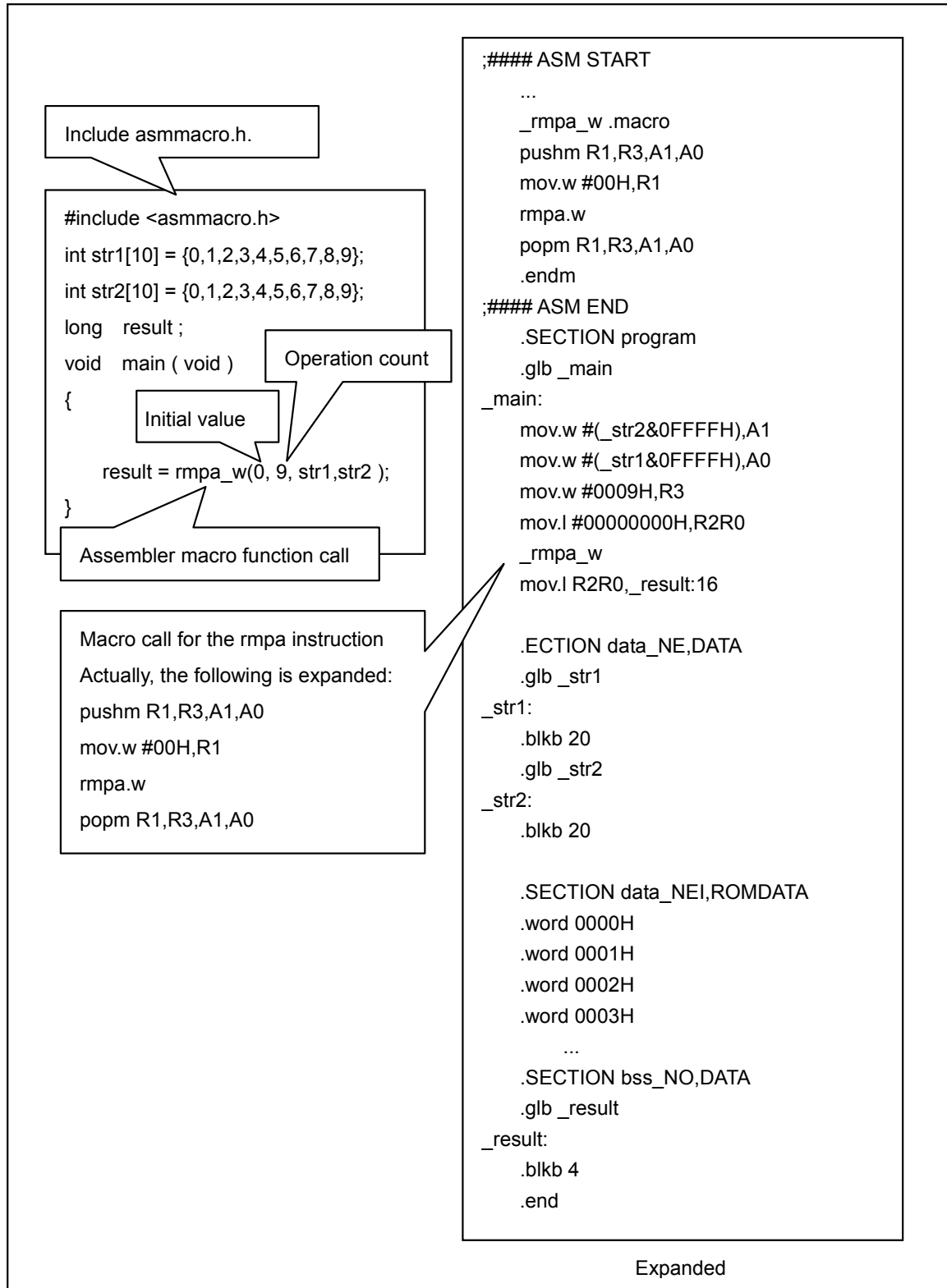


Figure 3.13 Sum-of-Products Operation Using the Assembler Macro Function "rmpa_w"

3.2.5 #pragma __ASMMACRO

For NC308, you can use #pragma __ASMMACRO to make any assembler instruction string into assembler macro functions.

Function

Declares a function defined by the assembler macro.

Syntax

```
#pragma __ASMMACRO function-name (register-name,...)
```

Rules

1. You must enter a prototype declaration of the function before the #pragma __ASMMACRO declaration. You must declare the assembler macro functions as static.
2. You cannot declare a function that does not have any parameter. Parameters are passed via registers. Specify the register matching the parameter type (based on #pragma PARAMETER).
3. When you define an assembler macro, the macro name must be the declared function name prefixed by an underscore (_).
4. The return values are set as shown below according to the function calling rules. You cannot declare the compound types (structure and union types) as a return value.
char and _Bool types: R0L float type: R2R0
int and short types: R0 double type: R3R2R1R0
long type: R2R0 long long type: R3R1R2R0
5. Registers whose contents will be changed in the assembler macro must be saved at the beginning of the assembler macro. The registers must be restored immediately before the return. (You do not need to save or restore the register storing the return values).

Example

```
static long mul( int, int ); /* Be sure to declare "static". */
#pragma __ASMMACRO mul( R0, R2 )
#pragma ASM
_mul .macro
mul.w R2,R0 ; The return value is set in R2R0.
.endm
#pragma ENDASM
long l;
void test_func( void )
{
    l=mul(2,3);
}
```

Figure 3.14 Example of Using Assembler Macro

3.3 Pragma Functions and Options for Reducing ROM Area

Table 3.3 Pragma Functions and Options for Reducing ROM Area

Subsection	Title	Description
3.3.1	#pragma SBDATA	Uses SB relative addressing to access variables.
3.3.2	#pragma SB16DATA	Uses 2-byte SB relative addressing to access variables.
3.3.3	#pragma BIT	Generates one-bit manipulation instructions in the 16-bit absolute addressing mode.
3.3.4	#pragma SPECIAL	Compresses a jump subroutine instruction from four bytes to two bytes.
3.3.5	-fjsrw	Uses the JSR.W instruction to call a function.
3.3.6	-OR	Performs maximum optimization of ROM efficiency.
3.3.7	-fno_align	Does not align the start address of the function.
3.3.8	-Wno_used_function	Outputs a warning for unused functions.

3.3.1 #pragma SBDATA

This option accesses the specified variables in the relative addressing mode using the SB register. By changing the access mode for frequently accessed variables to the SB relative addressing, you can improve the efficiency of the code. The variables specified for the SB relative addressing are assigned to the SBDATA attribute section, and are referenced using the offset from the start address of the SBDATA attribute section saved in the SB register. This causes the expanded code to be more compact than the one that loads addresses for referencing, and helps improve the ROM efficiency. The format is as follows:

```
#pragma SBDATA variable-name
```

The maximum access area for the SB register-based addressing is 256 bytes from the SB register. The address specification only requires one byte. Specifying this type of addressing for frequently-used variables will reduce the ROM area.

Before using SBDATA	Using SBDATA
int a; a=1;	#pragma SBDATA a int a; a=1;
.GLB __SB__ .SB __SB__ .FB 0 ... mov.w #0001H,_a	.GLB __SB__ .SB __SB__ .FB 0 .SBSYM _a ... mov.w #0001H,_a

Figure 3.15 Example of Using Addressing Mode with SBDATA

3.3.2 #pragma SB16DATA

SB16DATA specifies addressing that uses one-byte offset from SB register. SB16DATA specifies addressing that uses two-byte offset. For this type of addressing, the maximum access area is 64 Kbytes from the SB register. The address specification only requires two bytes. Specifying #pragma SB16DATA for frequently-used variables will reduce the ROM area. The format is as follows:

#pragma SB16DATA *variable-name*

Before using SB16DATA	Using SB16DATA
<pre>int a; a=1;</pre>	<pre>#pragma SB16DATA a int a; a=1;</pre>
<pre>.GLB __SB__ .SB __SB__ .FB 0 ... mov.w #0001H,_a</pre>	<pre>.GLB __SB__ .SB __SB__ .FB 0 .SBSYM16 _a ... mov.w #0001H,_a</pre>

Figure 3.16 Example of Using Addressing Mode with SB16DATA

3.3.3 #pragma BIT

This option declares that the specified external variable is in the range of addresses (from 00000H to 01FFFH) available for one-bit manipulation instructions in the 16-bit absolute addressing mode. This allows you to generate one-bit manipulation instructions in the 16-bit absolute addressing mode. (This function is only available for NC30WA).

The format is as follows:

#pragma BIT *variable-name*

Before using #pragma BIT	Using #pragma BIT
int sym sym=0x01 sym	#pragma BIT sym int sym sym=0x01 sym
or.w #01H,_sym	bset 0,_sym

Figure 3.17 Bit Operation Using #pragma BIT

3.3.4 #pragma SPECIAL

The special page compresses a jump subroutine instruction from four bytes of JSR.A_func to two bytes of JSRS number. This reduces the ROM area.

The format is as follows:

```
#pragma SPECIALΔ[/ C]Δcall-numberΔfunction-name()
```

```
#pragma SPECIALΔ[/ C]Δfunction-name(vect=call-number)
```

The functions declared in #pragma SPECIAL are mapped to the addresses created by adding 0FF0000H to the address set in the special page vector tables, and are therefore subject to special page subroutine calls. You may specify the following switch in the declaration:

[/C]

This switch generates the code for saving the required registers when the declared function is called.

You can specify a call number in the declaration.

Specify a call number and the compile option -fmake_special_table (-fMST) for compiling source files. The compiler will automatically generate a special page vector table.

Functions declared using #pragma SPECIAL are mapped to the program_S section.

You must map the program_S section between 0FF0000H and 0FFFFFFH.

You can specify call numbers from 18 to 255 in decimal only. As a label, "_SPECIAL_calling-number:" is output to the start address of functions declared using #pragma SPECIAL. Set this label in the special page subroutine table in the startup file.

The above setting is unnecessary if the -fmake_special_table (-fMST) option is specified.

If you specify different call numbers for a function, the call number declared later takes effect.

Example:

```
#pragma SPECIAL func(vect=20)
```

```
#pragma SPECIAL func(vect=30)// Call number 30 takes effect
```

If functions are defined in one file and function calls are defined in another file, you must specify this declaration in both files.

Before using #pragma SPECIAL	Using #pragma SPECIAL
<pre>void func(unsigned int, unsigned int); void main() { int i, j; i = 0x7FFD; j = 0x007F; func(i, j); }</pre>	<pre>#pragma SPECIAL 20 func() void func(unsigned int, unsigned int); void main() { int i, j; i = 0x7FFD; j = 0x007F; func(i, j); }</pre>
<pre>push.w -2[FB] ; j mov.w -4[FB],R0 ; i jsr \$func</pre>	<pre>push.w -2[FB] ; j mov.w -4[FB],R0 ; i jsrs #20</pre>

Figure 3.18 Example of Using the #pragma SPECIAL Declaration

3.3.5 -fjsrw

The compiler uses the JSR.A instruction to call a function that has been defined outside the file. However, most functions can be called by the JSR.W instruction if the program is not so large.

In this case, you can reduce the amount of code in the ROM as follows :

Compile with the source files with the -fJSRW option. Then, use "#pragma JSRA *function-name*" to declare the functions that caused an error during linking.

Note: When you use the -OGJ option, the most suitable jmp instruction is selected during linking.

C source	Without -fjsrw	With -fjsrw
<pre>/*file 1*/ void f() {} /* file 2*/ int main() { f(); }</pre>	<pre>.GLB __SB__ .SB __SB__ .FB 0 ... jsr_f</pre>	<pre>.OPTJ JSRW .GLB __SB__ .SB __SB__ .FB 0 ... jsr_f</pre>

Figure 3.19 Example of Using -fjsrw

3.3.6-OR

This option performs the maximum optimization to minimize the amount of code in the ROM, although the speed may be compromised. This option can be specified with the -g and -O options. Optimization with this option may partly modify the source line information. This may cause the program to behave differently during debugging. If you do not want to modify the source line information, use the -Ono_break_source_debug (-ONBSD) option to suppress optimization.

Figure 3.20 shows an example of optimization using the -OR option. The common expressions are unified to reduce the amount of code in the ROM.

C source	Without optimization	With optimization
<pre> if(b==1){ sub(); return a; } else if(b==2){ sub() return a; } else { return 0; } </pre>	<pre> ;## # C_SRC : if(b==1) cmp.w #0001H,_b:16 jne L1 ;## # C_SRC : sub(); jsr _sub ;## # C_SRC : return a; mov.w _a:16,R0 rts ;## # C_SRC : else if(b==2) L1: cmp.w #0002H,_b:16 jne L11 ;## # C_SRC : sub(); jsr _sub ;## # C_SRC : return a; mov.w _a:16,R0 rts ;## # C_SRC : else L11: ;## # C_SRC : return 0; mov.w #0000H,R0 rts </pre>	<pre> ;## # C_SRC : if(b==1) mov.w _b:16,R0 cmp.w #0001H,R0 jne L5 ;## # C_SRC : sub(); L31: jsr _sub ;## # C_SRC : return a; mov.w _a:16,R0 rts ;## # C_SRC : else if(b==2) L5: cmp.w #0002H,R0 jeq L31 ;## # C_SRC : return 0; mov.w #0000H,R0 rts </pre>

Figure 3.20 Example of Optimization Using -OR: Unifying Common Expressions

3.3.7-fno_align

This option does not align the start address of the function. This option prevents .align from being inserted to the beginning of the function, thus reducing the ROM area.

C source	Without -fno_align	With -fno_align
<pre>int f() { return 0; }</pre>	<pre>### # FUNCTION f ### # ARG Size(0) Auto Size(0) Context Size(4) .SECTION program, CODE, ALIGN .inspect 'U', 2, "program", "program", 0 .file 'C:/Hew3/fno_align/fno_align/fno_align.c' .type 256,'x',16,0 .func 'f','G',0,256,_f,0 .inspect 'F','s',"f","_f",'G',4 .align ### # C_SRC : { .glb_f _f:</pre>	<pre>### # FUNCTION f ### # ARG Size(0) Auto Size(0) Context Size(4) .SECTION program, CODE .inspect 'U', 2, "program", "program", 0 .file 'C:/Hew3/fno_align/fno_align/fno_align.c' .type 256,'x',16,0 .func 'f','G',0,256,_f,0 .inspect 'F','s',"f","_f",'G',4 ### # C_SRC : { .glb_f _f:</pre>

Figure 3.21 Example of Using -fno_align

3.3.8 -Wno_unused_function

This function displays unused global functions during linking. Deleting unused functions will reduce the ROM area.

C source	Warning message
<pre>void f() { } int main() { }</pre>	C:\Hew3\test2\test2\test2.c(20) : Warning (In308): Global function 'f' is never used

Figure 3.22 Example of Using -Wno_unused_function

3.4 Pragma Function and Options for Speeding UP Processing

Table 3.4 Pragma Function and Options for Speeding Up Processing

Subsection	Title	Description
3.4.1	#pragma STRUCT	Performs the alignment for the structure to improve the access speed.
3.4.2	-Ostack_frame_align	Performs the alignment for the stack frame to improve the access speed.
3.4.3	-OS	Performs maximum optimization of speed.
3.4.4	-Oloop_unroll[= <i>count</i>]	Unrolls the loop.
3.4.5	-Ofloat_to_inline	Performs inline expansion of the routine at runtime for floating point operations.

3.4.1 #pragma STRUCT

This option performs the alignment for the structure to improve the access speed.

Format 1. #pragma STRUCT *structure-tag-name* unpack

2. #pragma STRUCT *structure-tag-name* arrange

In the compiler, structures are packed. For example, the members of the structure in Figure 3.23 are arranged without any padding in the order they are declared.

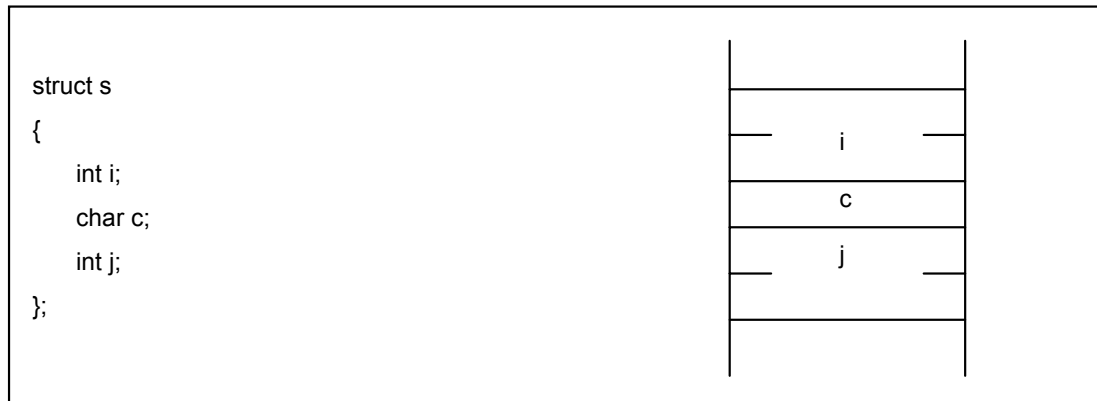


Figure 3.23 Example of Mapping Structure Members (1)

The extended function of the compiler allows you to control the mapping of structure members. Figure 3.24 shows an example of mapping the structure members when packing of the structure shown in Figure 3.23 is inhibited using #pragma STRUCTunpack.

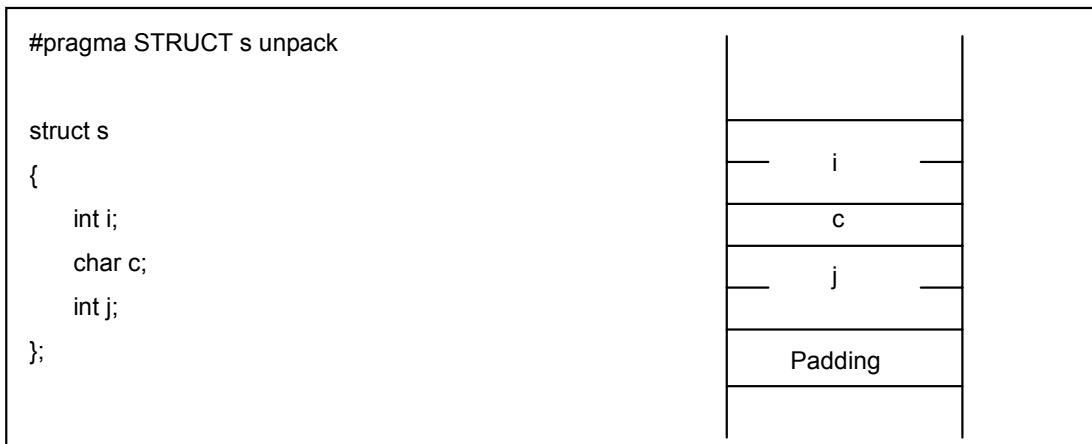


Figure 3.24 Example of Mapping Structure Members (2)

As shown in Figure 3.24, if the total size of the structure members is an odd number of bytes, #pragma STRUCTunpack adds 1 byte as padding after the last member. Therefore, if you use #pragma STRUCTunpack to inhibit padding, the size of all structures will be an even number of bytes.

The extended function of the compiler allows you to map all the odd-sized structure members first, followed by even-sized members. Figure 3.25 shows an example of mapping when the structure shown in Figure 3.23 is arranged using #pragma STRUCT arrange.

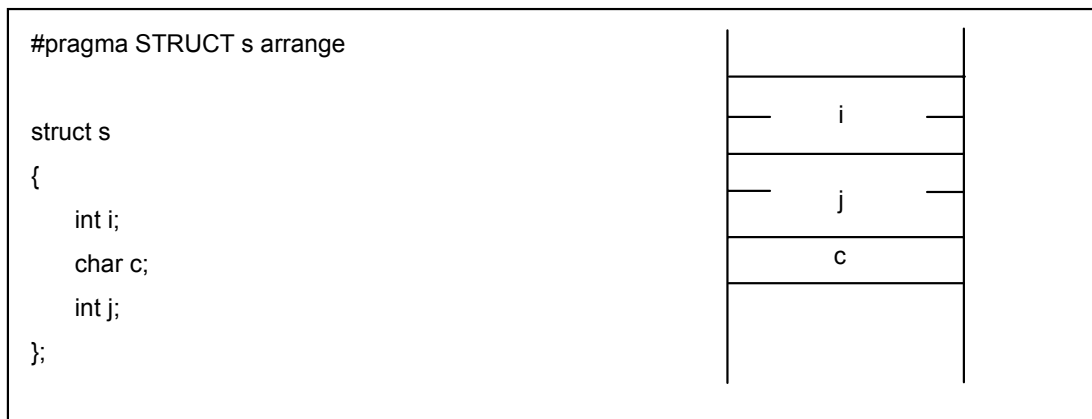


Figure 3.25 Example of Mapping Structure Members (3)

Using #pragma STRUCT unpack and #pragma STRUCT arrange together aligns the even-sized members.

Default	unpack
<pre> struct A { int a; char b; int c; }; f() { struct A a,b; a.a=1; a.b=2; a.c=3; b.a=4; b.b=5; b.c=6; } </pre>	<pre> #pragma STRUCT A unpack struct A { int a; char b; int c; }; f() { struct A a,b; a.a=1; a.b=2; a.c=3; b.a=4; b.b=5; b.c=6; } </pre>
<pre> ;## # C_SRC : a.a=1; mov.w #0001H,-10[FB]; a ;## # C_SRC : a.b=2; mov.b #02H,-8[FB] ; a ;## # C_SRC : a.c=3; mov.w #0003H,-7[FB] ; a ;## # C_SRC : b.a=4; mov.w #0004H,-5[FB] ; b ;## # C_SRC : b.b=5; mov.b #05H,-3[FB] ; b ;## # C_SRC : b.c=6; mov.w #0006H,-2[FB] ; b </pre>	<pre> ;## # C_SRC : a.a=1; mov.w #0001H,-12[FB]; a ;## # C_SRC : a.b=2; mov.b #02H,-10[FB] ; a ;## # C_SRC : a.c=3; mov.w #0003H,-9[FB] ; a ;## # C_SRC : b.a=4; mov.w #0004H,-6[FB] ; b ;## # C_SRC : b.b=5; mov.b #05H,-4[FB] ; b ;## # C_SRC : b.c=6; mov.w #0006H,-3[FB] ; b </pre>

Figure 3.26 Example of Using #pramga STRUCT (1)

arrange	arrange+unpack
<pre>#pragma STRUCT A arrange struct A { int a; char b; int c; }; f() { struct A a,b; a.a=1; a.b=2; a.c=3; b.a=4; b.b=5; b.c=6; }</pre>	<pre>#pragma STRUCT A arrange #pragma STRUCT A unpack struct A { int a; char b; int c; }; f() { struct A a,b; a.a=1; a.b=2; a.c=3; b.a=4; b.b=5; b.c=6; }</pre>
<pre>;;# # C_SRC : a.a=1; mov.w #0001H,-10[FB]; a ;;# # C_SRC : a.b=2; mov.b #02H,-6[FB] ; a ;;# # C_SRC : a.c=3; mov.w #0003H,-8[FB] ; a ;;# # C_SRC : b.a=4; mov.w #0004H,-5[FB] ; b ;;# # C_SRC : b.b=5; mov.b #05H,-1[FB] ; b ;;# # C_SRC : b.c=6; mov.w #0006H,-3[FB] ; b</pre>	<pre>;;# # C_SRC : a.a=1; mov.w #0001H,-12[FB]; a ;;# # C_SRC : a.b=2; mov.b #02H,-8[FB] ; a ;;# # C_SRC : a.c=3; mov.w #0003H,-10[FB]; a ;;# # C_SRC : b.a=4; mov.w #0004H,-6[FB] ; b ;;# # C_SRC : b.b=5; mov.b #05H,-2[FB] ; b ;;# # C_SRC : b.c=6; mov.w #0006H,-4[FB] ; b</pre>

Figure 3.27 Example of Using #pragma STRUCT (2)

3.4.2 -Ostack_frame_align

If an even-sized auto variable is mapped to an odd address, memory access requires one more cycle than when the variable is mapped to an even address. This option causes the alignment that maps the even-sized auto variable to the even address. This enables fast memory access. (This option is available only for NC30WA).

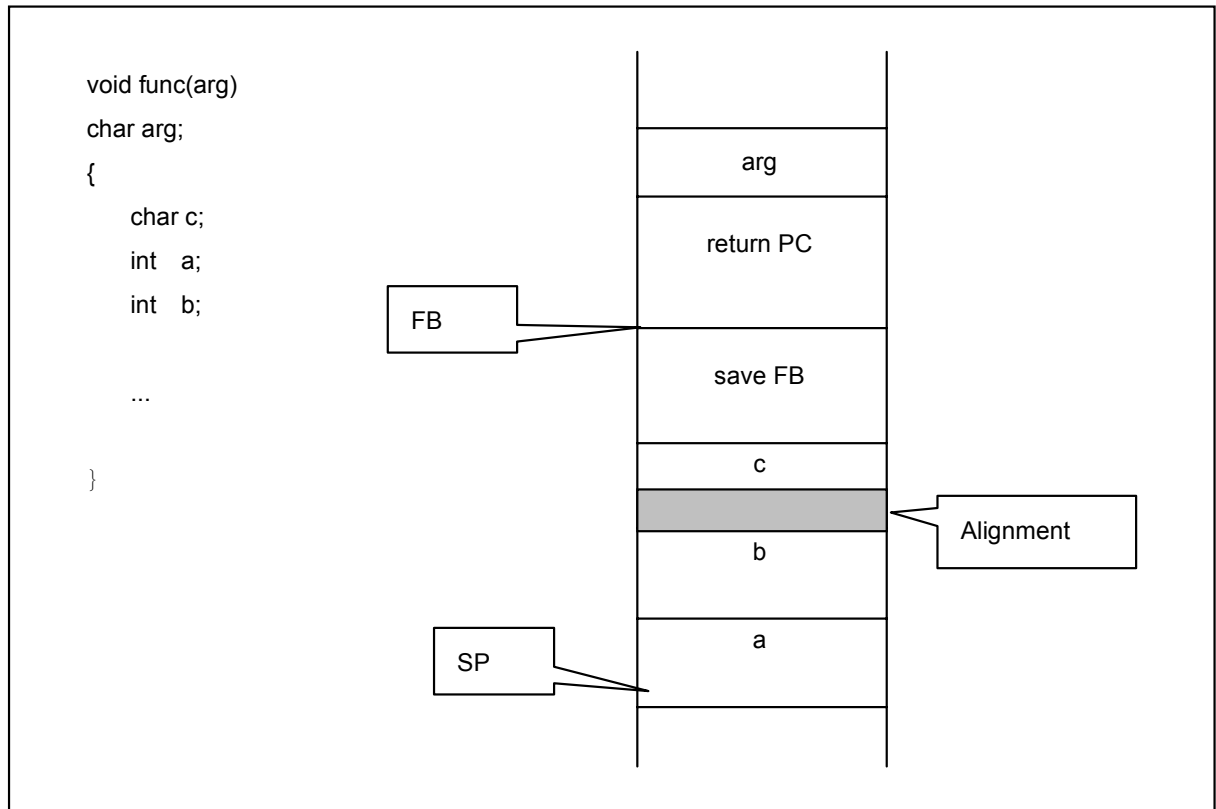


Figure 3.28 Example of Alignment by `-Ostack_frame_align`

C source	Without <code>-Ostack_frame_align</code>	With <code>-Ostack_frame_align</code>
<pre> void f() { int a; int b; a=1; b=2; ... } </pre>	<pre> ;### # C_SRC : a=1; mov.w #0001H,-2[FB] ; a ;### # C_SRC : b=2; mov.w #0002H,-4[FB] ; b </pre>	<pre> ;### # C_SRC : a=1; mov.w #0001H,-5[FB] ; a ;### # C_SRC : b=2; mov.w #0002H,-3[FB] ; b </pre>

Figure 3.29 Example of `-Ostack_frame_align`

3.4.3 -OS

This option performs the maximum optimization to obtain the fastest speed as possible, although the amount of code in the ROM may increase. This option can be specified along with the -g and -O options.

C source	With optimization	Without optimization
<pre>for(i=0;i<100;i++) a[i]=i*4;</pre>	<pre>;;# # C_SRC : for(i=0;i<100;i++) mov.w #0000H,_i:16 L1: ;;# # C_SRC : for(i=0;i<100;i++) cmp.w #0064H,_i:16 jge L5 ;;# # C_SRC : a[i]=i*4; mov.w _i:16,R0 shl.w #2,R0 indexwd.w _i:16 mov.w R0,_a:16 add.w #0001H,_j:16 jmp L1 L5:</pre>	<pre>mov.w #0000H,_i:16 mov.w _i:16,R0 shl.w #2,R0 L3: _line 26 ;;# # C_SRC : a[i]=i*4; indexwd.w _i:16 mov.w R0,_a:16 add.w #0001H,_i:16 cmp.w #0064H,_i:16 jlt L3</pre>

Figure 3.30 Example of Optimization by -OS

3.4.4 -Oloop_unroll[=*count*]

This option unrolls the code as many times as the loop count without revolving the loop statement. You may omit the loop count. If you omit the loop count, this option is applied to up to five loop statements. This option removes branches and counter calculations to improve the execution speed. However, this reduces the ROM efficiency.

C source	Without optimization	With optimization
<pre>for(i=0;i<3;i++) { a[i]=i; }</pre>	<pre>;;# # C_SRC : for(i=0;i<3;i++) mov.w #0000H,-2[FB] ; i L1: ;;# # C_SRC : for(i=0;i<3;i++) cmp.w #0003H,-2[FB] ; i jge L5 ;;# # C_SRC : a[i]=i; indexwd.w -2[FB] ; i mov.w -2[FB],_a:16 ; i add.w #0001H,-2[FB] ; i jmp L1 L5:</pre>	<pre>mov.w #0000H,-2[FB] ; i mov.w #0002H,A0 mul.w #0000H,A0 mov.w A0,A0 mov.w #0000H,_a:16[A0] mov.w #0001H,-2[FB] ; i mov.w #0002H,A0 mul.w #0001H,A0 mov.w A0,A0 mov.w #0001H,_a:16[A0] mov.w #0002H,-2[FB] ; i ;;# # C_SRC : a[i]=i; mov.w #0002H,R0 indexwd.w R0 mov.w #0002H,_a:16 mov.w #0003H,-2[FB] ; i</pre>

Figure 3.31 Example of Unrolling a Loop

3.4.5 -Ofloat_to_inline

This option performs inline expansion of floating-point runtime libraries to speed up the processing of floating-point operations (only for comparison and multiplication). This option is available only for the M32C/80 Series. When using this option, you must also specify the compile option "-M82".

Source code	Without -Ofloat_to_inline	With -Ofloat_to_inline
float f; long l; l=f;	push.l _f:16 jsr.a __f4toi4 add.l #04H,SP mov.l R2R0,_l:16	;;# # C_SRC : l=f; mov.w _f+2:16,R0 mov.w _f:16,R2 btst 7,R0H scc R3 mov.w R0,R1 shl.w #-07H,R1 and.w #0ffH,R1 xchg.w R0,R2 and.w #07fH,R2 mov.w R1,R1 jne ?+ mov.l R2R0,R2R0 jeq M1 ?: btst 7,R1L jc ?+ cmp.w #07fH,R1 jne M1 ?: add.w #062H,R1 tst.w #0ff00H,R1 jeq M2 mov.w R3,R3 jeq ?+ mov.l #80000000H,R2R0 jmp M3 ?: mov.l #7ffffffH,R2R0 jmp M3 M2: dec.w R1 or.w #0080H,R2 shlnc.l #8H,R2R0 ?: inc.w R1 cmp.w #0100H,R1 jeq M4 shlnc.l #-1,R2R0 jmp ?- M4: cmp.w #1,R3 jne M3 not.w R0 not.w R2 add.l #01H,R2R0 jmp M3 M1: mov.l #0,R2R0 M3:

Figure 3.32 Example Optimization with -Ofloat_to_inline

3.4.6 -Ostatic_to_inline

This option treats a static function (a function declared to be static) as an inline function (a function declared to be inline), and generates an inline-expanded assembling code.

The compiler treats the static function as an inline function and generates an inline-expanded assembling code when the following conditions are satisfied:

- (1) This option is applicable to a static function whose entity is specified before a function call.
(The function call and the entity of that function must be contained in the same source file.)
(Ignore this condition if you specify the -Oforward_function_to_inline option.)
- (2) Address acquisition for the target static function is omitted in the program.
- (3) The recursive call of the target static function is not performed.
- (4) The construction of a frame (reservation of an auto variable, and so on) is not performed in the assembling code output of a compiler. (Whether the frame construction is performed depends on the contents of the description of the target function, and another optimization option.)

(Ignore this condition if you specify the -Oforward_function_to_inline option.)

C source	Without -Ostatic_to_inline	With -Ostatic_to_inline
<pre>static int f(int a,int b) { return a+b; } int c; void main() { c=f(2,3); }</pre>	<pre>push.w #0003H mov.w #0002H,R0 jsr \$f add.l #02H,SP mov.w R0,_c:16</pre>	<pre>mov.w #0005H,_c:16</pre>

Figure 3.33 Example of Using -Ostatic_to_inline

3.5 Pragma Functions and Options for Reducing ROM Area and Speeding Up Processing

Table 3.5 Pragma Functions and Options for Reducing ROM Area and Speeding Up Processing

Subsection	Title	Description
3.5.1	-O[1-5]	Performs optimization.
3.5.2	-Osp_adjust	Performs optimization that corrects the stack pointer at a time.
3.5.3	-fuse_DIV	Uses the div instruction to perform division.
3.5.4	-Wno_unused_argument	Outputs a warning for unused arguments.
3.5.5	-fsmall_array	Calculates subscripts of arrays in 16 bits.
3.5.6	-fdouble_32	Handles double data as float data.

3.5.1-O[1-5]

This option perform the maximum optimization for faster processing and reduced amount of code in the ROM. This option can be specified with the -g option. The system assumes -O3 if you do not specify any number (level).

-O1: Same as when -O3, -Ono_bit, -Ono_break_source_debug, -Ono_float_const_fold, and -Ono_stdlib are valid.

-O2: Same as -O1.

-O3: Performs the maximum optimization for faster processing and reduced amount of code in the ROM.

-O4: Validates -O3 and -Oconst.

-O5: Performs the maximum optimization that improves common subexpressions (when the -OR option is concurrently specified), transfer of character strings, and comparison (when the -OS option is concurrently specified).

However, a normal code may be unable to be output when the following conditions are satisfied:

- Different variables point to the same memory position simultaneously.
- These variables are used within the same function.

```
Example:
int a=3;
int *p=&a;

test()
{
    int b;
    *p=9;
    a=10;
    b=*p; //Optimization will replace "p" with "9".
    print("b=%d(expect b=10)¥n",b);
}

Result)
b=9(expect b=10)
```

Figure 3.34 Example of the Source Specification with "-O5" that Causes Incorrect Operations

Note:

You cannot use the BTSTC or BTSTS bit manipulation instructions to write to or read data from registers in the SFR area.

If you use the optimization option (-O5), the compiler may generate the bit manipulation instructions (BTSTC and BTSTS) for the assembler code. If you use the -O5 optimization option in the following compile specification, an interrupt request bit cannot be determined correctly, resulting in unexpected operations.

```

[Example: C source that cannot be used with the optimization option]
#pragma ADDRESS TA0IC 006Ch /* M16C/80 Timer A0 interrupt control register
*/
struct {
    char ILVL : 3;
    char IR : 1; /* An interrupt request bit */
    char dmy : 4;
} TA0IC;
void wait_until_IR_is_ON(void)
{
    while (TA0IC.IR == 0) /* Wait for 1 */
    {
        ;
    }
    TA0IC.IR = 0; /* Return 1 to 0 */
}

```

Figure 3.35 Example when Optimization Options Cannot be Used

If you find that the bit manipulation instructions (BTSTC and BTSTS) are output to the SFR area, take the following measures before performing the compilation. Make sure that the generated code does not have any problem in these settings:

- Use an optimization option other than -O5.
- Use the ASM function to directly specify the instruction in the program.
- Add the -Ono_asmopt (or -ONA) option.

3.5.2-Osp_adjust

This option performs optimization by combining stack correction codes after function calls. Usually, the compiler corrects a stack pointer to release the parameter area for functions each time a function is called. When this option is specified, the stack pointer is corrected collectively, rather than for each function call. The -Osp_adjust option reduces the amount of code used in the ROM and speeds up the processing. However, the amount of stack used may increase.

C source	Without -Osp_adjust	With -Osp_adjust
<pre>main() { f(1.1); g(1.1); }</pre>	<pre>_main: ;## # C_SRC : f(1.1); push.l #3ff19999H push.l #9999999aH jsr _f add.l #08H,SP ;## # C_SRC : g(1.1); push.l #3ff19999H push.l #9999999aH jsr _g add.l #08H,SP ;## # C_SRC : } rts</pre>	<pre>_main: ;## # C_SRC : f(1.1); push.l #3ff19999H push.l #9999999aH jsr _f ;## # C_SRC : g(1.1); push.l #3ff19999H push.l #9999999aH jsr _g add.l #010H,SP ;## # C_SRC : } rts</pre>

Figure 3.36 Example of Using -Osp_adjust

3.5.3 -fuse_DIV

This option changes generated code for division.

The compiler generates div.w (divu.w) and div.b (divu.b) microcomputer instructions for the following divisions:

- The dividend is a 4-byte value, the divisor is a 2-byte value, and the result is a 2-byte value.
- The dividend is a 2-byte value, the divisor is a 1-byte value, and the result is a 1-byte value.

If the division results in an overflow when this option is specified, the compiler may operate differently from stipulated in ANSI. If the division results in an overflow, For the div instruction of the M16C, the result is unpredictable if an overflow occurs.

Therefore, when NC308 compiles the program in default settings, it calls a runtime library to correct the result for this problem even in cases where the dividend is 4-byte, the divisor is 2-byte, and the result is 2-byte.

Source program	Default	When using -fuse_DIV
<pre>int k,j; long l; k=l/j;</pre>	<pre>mov.w _j:16,R1 exts.w R1 push.l R3R1 mov.l _l:16,R2R0 glb __i4div jsr.a __i4div add.l #4H,SP mov.w R0,_k:16</pre> <p>Output the code, considering an overflow.</p>	<pre>mov.l _l:16,R2R0 div.w _j:16 mov.w R0,_k:16</pre> <p>Use the div instruction.</p>

Figure 3.37 Example of Using -fuse_DIV

3.5.4 -Wno_unused_argument

When a function having arguments is defined, this option outputs a warning for unused arguments. By correcting the source program based on the warning, you can save the memory space and speed up the processing.

C source	Warning message
<pre>int f(int a,int b,int c) { return a+c; }</pre>	C:\Hewlett-Packard\test1\test1.c(68) : [Warning(ccom)] function "f()" has no-used argument(b).

Figure 3.38 Example of Executing -Wno_unused_argument

3.5.5 -fsmall_array

When referencing a far-type array whose total size is unknown during compiling, this option calculates subscripts in 16 bits, assuming that the array's total size is within 64 Kbytes. When referencing elements of a far-type array of unknown size, the compiler calculates subscripts in 32 bits by default, so that arrays of 64 Kbytes or longer can be handled. See the following example:

```
extern int array[];  
int i = array[j];
```

In this case, because the total size of the array is not known to the compiler, the subscript "j" is calculated in 32 bits.

When this option is specified, the compiler assumes that the total size of the array is 64 Kbytes or less and calculates the subscript "j" in 16 bits. This can increase the processing speed and reduce the amount of code.

We recommend you use this option whenever the size of one array does not exceed 64 Kbytes.

Source program	Without -fsmall_array	With -fsmall_array
extern far int a[]; int i,j; i=a[j];	mov.w -2[FB],R0 ; j exts.w R0 mov.l R2R0,A0 shl.l #1,A0 mov.w _a[A0],-2[FB] ; i	indexws.w -4[FB] ; j mov.w:g_a,-2[FB] ; i

Figure 3.39 Example of Using -fsmall_array

3.5.6 -fdouble_32

This option causes the compiler to process the double type as the float type.

Notes:

1. **When you specify this option, you must declare a function prototype. Without a prototype declaration, the compiler may generate an invalid code.**
2. **When you specify this option, the debug information of the double type is processed as the float type. Therefore, the double data is displayed as the float type on the C watch window and global window of debugger PD308 and simulator PD308SIM.**

C source	Without -fdouble_32	With -fdouble_32
<pre>float data; data=data+123.456;</pre>	<pre>push.l __data:16 .glb __f4tof8 jsr.a __f4tof8 add.l #04H,SP pushm R3,R2,R1,R0 push.l #405edd2fH push.l #1a9fbe77H .glb __f8add jsr.a __f8add add.l #010H,SP pushm R3,R2,R1,R0 .glb __f8tof4 jsr.a __f8tof4 add.l #08H,SP mov.l R2R0,__data:16</pre>	<pre>push.l __data:16 push.l #42f6e979H .glb __f4add jsr.a __f4add add.l #08H,SP mov.l R2R0,__data:16</pre>

Figure 3.40 Example of Using the -fdouble_32 Option

3.6 Other Pragma Functions and Options

3.6.1 Other Pragma Functions

(1) Extended functions for memory mapping

Extended function	Description
#pragma ROM	Maps the specified variable to the rom section. Syntax: #pragma ROM Δ <i>variable-name</i> Example: #pragma ROM val Note: This facility is provided to maintain compatibility with NC77 and NC79. The variable normally must be located in the rom section using the const qualifier.
#pragma SECTION	Changes the section name generated by the compiler. Syntax: #pragma SECTION Δ <i>existing-section-name</i> Δ <i>new-section-name</i> Example: #pragma SECTION bss nonval_data

(2) Extended functions for use with target devices

Extended function	Description
#pragma ADDRESS (#pragma EQU)	Assigns a variable to the absolute address. Syntax: #pragma ADDRESS Δ <i>variable-name</i> Δ <i>absolute-address</i> Example: #pragma ADDRESS port0 2H
#pragma BITADDRESS	Assigns a variable to the bit position of the specified absolute address. Syntax: #pragma BITADDRESS Δ <i>variable-name</i> Δ <i>bit-position</i> , <i>absolute-address</i> Example: #pragma BITADDRESS io 1,100H
#pragma DMAC	Assigns a DMAC register for the external variable. (Only for NC308WA) Syntax: #pragma DMAC Δ <i>variable-name</i> Δ <i>DMAC-register-name</i> Example: #pragma DMAC dma0 DMA0
#pragma INTCALL	Declares the function to be called by software interrupt (int instruction). Switch [/c] generates the code for saving the required register when calling the function. Syntax 1: #pragma INTCALL Δ [/C] Δ <i>INT-number</i> Δ <i>assembler-function-name</i> (<i>register-name</i>) Example 1:

Extended function	Description
	<pre>#pragma INTCALL 25 func(R0, R1) #pragma INTCALL /C 25 func(R0, R1) Syntax 2: #pragma INTCALLΔ INT-numberΔC-function-name() Example 2: #pragma INTCALL 25 func() #pragma INTCALL /C 25 func() Note: You must declare the prototype of the function before entering this declaration.</pre>
<pre>#pragma INTERRUPT (#pragma INTF)</pre>	<p>Declares the interrupt processing function written in C. This declaration causes the compiler to generate the code, at the entry and exit points of the function, that performs a procedure for the interrupt processing function.</p> <p>Syntax:</p> <pre>#pragma INTERRUPTΔ[B E F]Δinterrupt-processing-function-name #pragma INTERRUPTΔ[B E F]Δinterrupt-vector-numberΔinterrupt-processing -function-name #pragma INTERRUPTΔ[B E F]Δinterrupt-processing-function-name(vect=inte rrupt-vector-number) Example: #pragma INTERRUPT int_func #pragma INTERRUPT /B int_func #pragma INTERRUPT 10 int_func #pragma INTERRUPT /E 10 int_func #pragma INTERRUPT int_func (vect=10) #pragma INTERRUPT /F int_func (vect=20) Note: You can also use #pragma INTF for maintaining compatibility with C77.</pre>
<pre>#pragma PARAMETER</pre>	<p>Declares that the parameters are passed via the specified registers when calling an assembler function.</p> <p>Switch [/C] generates the code for saving the required register when calling the function.</p> <p>Syntax:</p> <pre>#pragma PARAMETERΔ[C]Δfunction-name(register-name) Example: #pragma PARAMETER asm_func(R0, R1) #pragma PARAMETER /C asm_func(R0, R1)</pre>

Extended function	Description
	Note: You must declare the prototype of the function before entering this declaration.

(3) Extended functions for MR308 support

Extended function	Description
#pragma ALMHANDLER	Declares the name of the MR308 alarm handler. Syntax: #pragma ALMHANDLER Δ <i>function-name</i> Example: #pragma ALMHANDLER alm_func
#pragma CYCHANDLER	Declares the name of the MR308 cyclic start handler. Syntax: #pragma CYCHANDLER Δ <i>function-name</i> Example: #pragma CYCHANDLER cyc_func
#pragma INTHANDLER #pragma HANDLER	Declares the name of the MR308 interrupt handler. Syntax 1: #pragma INTHANDLER Δ [/E] Δ <i>function-name</i> Syntax 2: #pragma HANDLER Δ [/E] Δ <i>function-name</i> Example: #pragma INTHANDLER int_func
#pragma TASK	Declares the name of the MR308 task start function. Syntax: #pragma TASK Δ <i>task-start-function-name</i> Example: #pragma TASK task1

(4) Other extended functions

Extended function	Description
#pragma ASM #pragma ENDASM	Specifies the area in which coding is made in assembly language. Syntax: #pragma Δ ASM #pragma Δ ENDASM Example: #pragma ASM mov.w R0,R1 add.w R1,02H #pragma ENDASM
#pragma JSRA	Calls a function by using JSR.A as the JSR instruction. Syntax: #pragma JSRA Δ <i>function-name</i> Example: #pragma JSRA func

Extended function	Description
#pragma JSRW	<p data-bbox="571 282 1251 315">Calls a function by using JSR.W as the JSR instruction.</p> <p data-bbox="571 338 1082 371">Syntax: #pragma JSRWΔ<i>function-name</i></p> <p data-bbox="571 394 954 427">Example: #pragma JSRW func</p>
#pragma PAGE	<p data-bbox="571 439 1246 472">Specifies a new-page point in the assembler listing file.</p> <p data-bbox="571 495 895 528">Syntax: #pragmaΔPAGE</p> <p data-bbox="571 551 887 584">Example: #pragma PAGE</p>
#pragma __ASMMACRO	<p data-bbox="571 595 1246 629">Declares the function defined by the assembler macro.</p> <p data-bbox="571 651 1198 707">Syntax: #pragma __ASMMACROΔ<i>function-name</i> (<i>register-name</i>)</p> <p data-bbox="571 730 1158 763">Example: #pragma __ASMMACRO mul(R0,R2)</p>

3.6.2 Other Options

(1) Options for controlling the compile driver

Option	Function
-c	Creates a relocatable file (extension .r30), and ends processing.
-D <i>identifier</i>	Defines an identifier. This option has the same function as #define.
-I <i>directory-name</i>	Specifies the name of the directory that contains files to be referenced by the #include preprocess command. You can specify up to 16 directories.
-E	Processes only the preprocess commands and outputs the result to standard output.
-P	Starts only the preprocess commands and creates a file (extension .i).
-S	Creates an assembly source file (extension .a30), and ends processing.
-U <i>predefined-macro-name</i>	Makes the predefined macro to undefined one.
-silent	Suppresses the copyright message at startup.
-dsource	Generates an assembly source file (extension ".a30") with a C source list output as a comment. (This file is not deleted even after assembling.)
-dsource_in_list	Generates an assembly language list file (extension .lst), in addition to performing the "-dsource" function.

(2) Options for specifying output files

Option	Function
-o file-name	Specifies the name of a file (absolute module file, map file, and so on) generated by ln308. You can also specify a pathname including a directory name. Do not specify file name extensions.
-dir directory-name	Specifies the directory to which files (absolute module file, map file, and so on) generated by ln308 are output.

(3) Options for displaying version information and command line

Option	Function
-v	Displays the name of the command program being executed and the command line.
-V	Displays the startup messages of the compiler programs, then ends processing (without compiling anything).

(4) Options for debugging

Option	Function
-g	Outputs debug information to an assembly source file (extension .a30). This enables C-language level debugging.
-genter	Always outputs an enter instruction during function call. You must specify this option when using the debugger's stack trace function.
-gno_reg	Suppresses the output of debug information for register variables.

(5) Optimization options

Option	Function
-Oconst	Performs optimization by replacing references to const-modified variables with constants.
-Ono_bit	Suppresses optimization based on grouping of bit manipulations.
-Ono_break_source_debug	Suppresses optimization that affects source line information.
-Ono_float_const_fold	Suppresses constant folding processing of floating-point numbers.
-Ono_stdlib	Suppresses inline padding of standard library functions and modification of library functions.
-Ono_logical_or_combine	Suppresses optimization that puts consecutive ORs together.
-Ono_asmopt	Suppresses optimization by the assembler optimizer "aopt30."
-Ocompare_byte_to_word	Compares consecutive bytes of data at contiguous addresses in words.
-Ofoward_function_to_inline	Performs inline expansion for all the inline functions.
-Oglb_jump	Optimizes external references to jump instructions.

(6) Options for modifying generated codes

Option	Function
-fansl	Validates -fnot_reserve_far_and_near, -fnot_reserve_asm, -fnot_reserve_inline, and -fextend_to_int.

Option	Function
-fnot_reserve_asm	Exclude asm from reserved words (only "_asm" is valid).
-fnot_reserve_far_and_near	Exclude far and near from reserved words (only "_far" and "_near" are valid).
-fnot_reserve_inline	Exclude inline from reserved words (only "_inline" will be a reserved word).
-fextend_to_int	Performs operation after extending char-type data to int-type (use the expansion conforming to the ANSI standards.)
-fchar_enumerator	Handles the enumerator type as the unsigned char-type, not as the int-type.
-fno_even	Allocates all output data to the odd attribute section without separating odd data from even data.
-ffar_RAM	Changes the default attribute of RAM data to "far."
-fnear_ROM	Changes the default attribute of ROM data to "near."
-fnear_pointer	Changes the default attribute of the pointers and addresses to "near."
-fconst_not_ROM	Does not handle the types specified by const as ROM data.
-fnot_address_volatile	Does not regard the variables specified by #pragma ADDRESS (#pragma EQU) as variables specified by volatile.
-fenable_register	Validates the register storage class.
-finfo	Outputs the information required for the Inspector, Stk Viewer, Map Viewer, and utl30.
-M82	Generates the code for the M32C/80 Series.
-fswitch_other_section	Outputs a table jump for the switch statement to a section other than the program section.
-ferase_static_function= <i>function-name</i>	Does not generate any code if the function specified in this option is a static function.
-fno_switch_table	Generates the code that branches after making comparison for the switch statement.
-fmake_vector_table	Automatically generates a variable vector table.
-fmake_special_table	Automatically generates a special page vector table.

(7)Library specification option

Option	Function
-l library-file-name	Specifies the library to be used during linking.

(8) Warning options

Option	Function
-Wnon_prototype	Outputs a warning if a function without prototype declaration is used.
-Wunknown_pragma	Outputs a warning if unsupported #pragma is used.
-Wno_stop	Does not stop the compilation even if an error occurs.
-Wstdout	Outputs an error messages to standard output (stdout) of the host machine.
-Werror_file<filename>	Outputs tag files.
-Wstop_at_warning	Stops compiling if a warning occurs during compiling.
-Wnesting_comment	Outputs a warning for a comment including "/*."
-Wccom_max_warnings = <i>warning-count</i>	Allows you to specify the maximum number of times ccom308 can output a warning.
-Wall	Displays all detectable warnings (however, not including warnings output by "-Wlarge_to_small" and "-Wno_used_argument").
-Wmake_tagfile	Outputs a tag file for each file if an error and warning occurs.
-Wuninitialize_variable	Outputs a warning for auto variable that have not been initialized.
-Wlarge_to_small	Outputs a warning for implicit assignment of variables in descending sequence of size.
-Wno_warning_stdlib	This option specified with "-Wnon_prototype" or "-Wall" inhibits the "warning for standard libraries that do not have prototype declaration."
-Wno_used_static_function	Displays the static function name that does not require code generation.
-Wundefined_macro	Displays a warning if an undefined macro is used in #if.
-Wstop_at_link	Suppress the generation of an absolute module file if a warning occurs during linking.

(9) Assembly and link options

Option	Function
-as308Δ<option>	Specifies the options for the as308 assemble command. To pass two or more options, enclose them in double quotes ("").
-ln308Δ<option>	Specifies the options for the ln308 link command. To pass two or more options, enclose them in double quotes ("").

3.7 Sections

3.7.1 Sections Managed by NC308

NC308 manages the data and code mapping areas as sections.

This subsection describes the types of sections managed by NC308, and how to manage them.

(1) Structure of sections

NC308 manages data according to the type as individual sections. Table 3.6 shows the structure of sections that NC308 manages.

Table 3.6 Structure of Sections for NC308

Section base name	Contents
data	Stores static variables with initial values.
bss	Stores static variables without initial values.
rom	Stores character strings and constants.
program	Stores programs.
program_s	Stores programs specified in #pragma SPECIAL.
vector	Variable vector area (The compiler does not generate this area.)
fvector	Fixed vector area (The compiler does not generate this area.)
stack	Stack area (The compiler does not generate this area.)
heap	Heap area (The compiler does not generate this area.)

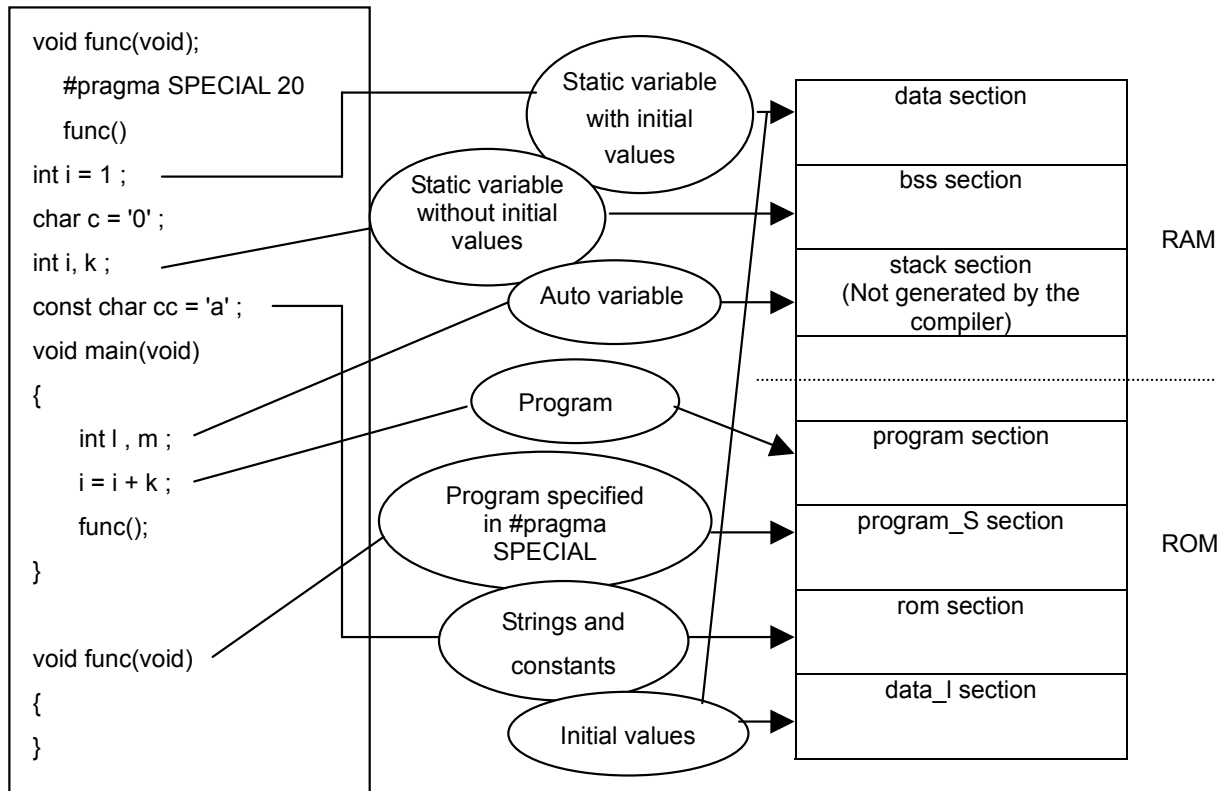


Figure 3.41 Mapping Sections according to Type of Data

(2) Section attributes

The sections generated by NC308 are further classified according to their attributes, including whether they have initial values, which area they are mapped to, and their data size.

Table 3.7 shows the symbols indicating the attributes.

Table 3.7 Section Attributes

Attribute	Contents	Applicable section base name
I	Section containing initial values of data	data
N/F/S	N: "near" attribute (area from 000000H to 00FFFFH) F: "far" attribute (area from 000000 to FFFFFFFH)	data, bss, rom
	S: SBDATA attribute (area available for SB relative addressing)	data, bss
E/O	E: Data size is an even number.	data, bss, rom
	O: Data size is an odd number.	

(3) Naming rules of sections

The names of sections generated by NC308 are determined using the section base names and attributes.

Figure 3.42 shows the combination of the section base name and attribute.

Section name = Section base name_Attribute

		Section base name			
		data	bss	rom	program
Attribute	Meaning				
N	"near" attribute				
F	"far" attribute				
S	SBDATA attribute				
E	Even data size				
O	odd data size				
I	With initial values				

Figure 3.42 Section Name Naming Rules

3.8 Issues Related to Cross-Software

3.8.1 Issues Related to Assembly Language Programs

Almost any kind of program can be written in C. However, you use the assembly language when you want to improve performance or use special instructions. This section reviews the issues you must keep in mind when linking C programs to assembly language programs.

(1) Calling assembler functions from C programs

(a) Assembler function without parameters

When calling assembler functions from C programs, use the name of the assembler functions the same way for calling functions written in C.

The first label name in an assembler function must be preceded by an underscore (_). To call the assembler function from the C program, use the first label name without the underscore. The calling C program must include a prototype declaration of the assembler function.

Figure 3.43 shows an example of calling the assembler function `asm_func`.

```
extern void asm_func( void );      ← Prototype declaration of the assembler function
void main()
{
    :
    (Omitted)
    :
    asm_func();                    ← Calling the assembler function
}
```

Figure 3.43 Example of Calling the Assembler Function without Parameters (smp1.c)

```
.glob _main
_main:
:
(Omitted)
:
jsr _asm_func ← Calling the assembler function (preceded by "_")
rts
```

Figure 3.44 Compiled Results of smp1.c (Excerpt) (smp1.a30)

(b) Passing parameters to assembler functions

When passing parameters to assembler functions, use the extended function `#pragma PARAMETER`.

`#pragma PARAMETER` passes parameters to the assembler functions via 32-bit general-purpose registers (R2R0, R3R1), 16-bit general-purpose registers (R0, R1, R2, R3), 8-bit general-purpose registers (ROL, ROH, R1L, R1H), and address registers (A0, A1).

The following shows the procedure for calling an assembler function using `#pragma PARAMETER`:

- ① Enter a prototype declaration of the assembler function before the `#pragma PARAMETER` declaration.
You must also declare the parameter types.
- ② For `#pragma PARAMETER`, declare the name of the registers to be used in the parameter list of the assembler function.

Figure 3.45 shows an example using `#pragma PARAMETER` to call the assembler function `asm_func`.

```
extern unsigned int asm_func(unsigned int, unsigned int);
#pragma PARAMETER asm_func(R0, R1) ← Pass the parameters to the assembler function
                                   via R0 and R1 registers.

void main()
{
    int i = 0x02;
    int j = 0x05;
    asm_func(i, j); ← Calling the assembler function
}
```

Figure 3.45 Example of Calling the Assembler Function with Parameters (smp2.c)

```

        .glob _main
_main:
        enter #04H
        pushm R1
        .line6
;## # C_SRC : int i = 0x02;
        mov.w #0002H,-4[FB]; i
        .line7
;## # C_SRC : int j = 0x05;
        mov.w #0005H,-2[FB]; j
        .line9
;## # C_SRC : asm_func(i, j);
        mov.w -2[FB],R1 ; j ← Pass the parameters to the assembler function
        mov.w -4[FB],R0 ; i   via R0 and R1 registers.
        jsr _asm_func ← Calling the assembler function (preceded by "_")
        .line10
;## # C_SRC : }
        popm R1
        exitd

```

Figure 3.46 Compiled Results of smp2.c (Excerpt) (smp2.a30)

(c) Limits on parameters in the #pragma PARAMETER declaration

You cannot declare the following parameter types in a #pragma PARAMETER declaration:

- Parameters of structure type and union type
- 64-bit integer type (long long) parameters
- Double precision floating-point type (double) parameters

You cannot define return values of the structure or union type for assembler functions.

(2) Coding assembler functions

(a) Coding the assembler function to be called

The following describes the procedure for coding the entry processing of an assembler function:

- ① Use the assembler pseudo instruction .SECTION to specify the section name.
- ② Use the assembler pseudo instruction .GLB to specify the function name label as global.
- ③ Add an underscore (_) to the function name to write it as a label.
- ④ If you want to modify the B or U flag, save the flag register into the stack.
- ⑤ Save the registers that may be destroyed within the function.

The following describes the procedure for coding the exit processing of an assembler function:

- ⑥ Restore the registers that have been saved during entry processing of the function.
- ⑦ If you modified the B and U flags within the function, restore the flag register from the stack.
- ⑧ Code the RTS instruction.

Do not change the contents of the SB and FB registers within the assembler function.

If you change the contents of the SB and FB registers, save them in the stack at the entry to the function, and then restore them from the stack at the exit of the function.

Figure 3.47 shows an example of coding the assembler function. In this example, the section name is "program", which is the same as the section name output by the compiler.

```

.SECTION program      ← ①
.GLB _asm_func       ← ②
_asm_func:           ← ③
  PUSHC FLG          ← ④
  PUSHM R3,R1        ← ⑤
  MOV.L SYM1, R3R1
  POPM R3,R1         ← ⑥
  POPC FLG           ← ⑦
  RTS                ← ⑧
.END
* ① to ⑧ correspond to the steps described above.

```

Figure 3.47 Example Coding of an Assembler Function

(b) Returning values from an assembler function

Values of the integer, pointer, and floating-point types can be returned from an assembler function to a C program via registers. Table 3.8 shows the calling rules for return values. Figure 3.48 shows a coding example of an assembler function to return a value.

Table 3.8 Calling Rules for Return Values

Return value type	Rule
_Bool type char type	R0L register
int type near pointer type	R0 register
float type long type	When the value is returned, the 16 low-order bits are stored in the R0 register and the 16 high-order bits are stored in the R2 register.
double type long double type	When the value is returned, it is stored in 16 bits each, beginning with the MSB, in order of registers R3, R2, R1, and R0.
long long type	When the value is returned, it is stored in 16 bits each, beginning with the MSB, in order of registers R3, R1, R2, and R0.

Return value type	Rule
structure type union type	Immediately before calling the function, the "far" address indicating the area for storing the return value is pushed to the stack. Before the return to the calling program, the called function writes the return value to the area indicated by the "far" address pushed to the stack.

```

.SECTION program
.GLB _asm_func
_asm_func:

(Omitted)

MOV.I #01A000H, R2R0
RTS
.END

```

Figure 3.48 Example of Coding Assembler Function to Return long-type Return Value

(c) Referencing C variables

Since assembler functions are written in different files from the C program, only the C global variables can be referenced.

To include the names of C variables in an assembler function, precede them with an underscore (_). You also need to use the assembler pseudo instruction .GLB to declare variables for external reference in the assembly language program.

Figure 3.49 shows an example of referencing the C program global variable "counter" from the assembler function asm_func.


```

[C program]
unsigned int counter;      ←C program global variable

main()
{
    :
    (omitted)
    :
}
[Assembler function]
.GLB _counter             ← Declare the C program global variable for external reference
asm_func:
    :
    (omitted)
    :
    MOV.W _counter, R0    ← Reference

```

Figure 3.49 Referencing a C Global Variable

(d) Notes on specifying interrupt processing in assembler functions

The following must be performed at the entry and exit points of a program (function) for interrupt processing:

1. Save the registers (R0, R1, R2, R3, A0, A1, and FB) at the entry point.
2. Restore the registers (R0, R1, R2, R3, A0, A1, and FB) at the exit point.
3. Use the REIT instruction to return from the function.

Figure 3.50 shows an example coding of an assembler function for interrupt processing.

```

.section program
.glob _func
_func:
pushm R0,R1,R2,R3,A0,A1,FB    ← Save all the registers.
MOV.B #01H, R0L
:
(Omitted)
:
popm R0,R1,R2,R3,A0,A1,FB    ← Restore all the registers.
reit                          ← Return to the C program.
.END

```

Figure 3.50 Example Coding of an Interrupt Processing Assembler Function

(e) Notes on calling C functions from the assembler

Not the following when calling functions written in C from an assembly language program:

- ① To call a C function, use a label name preceded by an underscore (_) or a dollar (\$).
- ② For NC308, the R0 register and registers used for return values are not saved during entry processing of the function. You must save these registers before calling the C function from the assembler.

For NC30, C functions do not save or restore registers. Before calling a C function, save the registers used in the assembler function. Restore them after returning from the C function.

(3) Notes on coding assembler functions

Note the following when coding assembly language functions (subroutines) that are called from a C program.

(a) Notes on handling the B and U flags

When returning from an assembler function to a C program, resume the B and U flags to the same condition as they were when the function was called.

(b) Notes on handling the FB register

If you modified the value of a FB (frame base register) in an assembler function, you cannot return to the calling C program normally. Therefore, do not modify the FB value in the assembler functions. If you need to change the FB register due to system design, save it at the start of the function and restore it when returning to the function from which it was called.

(c) Notes on handling the general-purpose registers and address registers

When modifying the contents of the general-purpose registers (R1, R2, and R3, except for R0) and address registers (A0, A1) in an assembler function, you must save them at the entry processing of the assembler function and restore them at the exit processing.

However, if the assembler function is declared using `#pragma PARAMETER /C`, the codes for saving and restoring the registers are generated in the calling program. Therefore, it is unnecessary to save and restore the registers in the assembler function. (The amount of code becomes somewhat larger.)

(d) Notes on passing parameters to an assembler function

If you want to pass parameters to a function written in assembly language, use the `#pragma PARAMETER` function to pass these parameters via registers. Figure 3.51 shows the format ("`asm_func`" is the name of the assembler function).

```
unsigned int near asm_func(unsigned int, unsigned int);
                                     ↑ Prototype declaration of the assembler function
#pragma PARAMETER asm_func(R0, R1)
```

Figure 3.51 Coding Example of an Assembler Function

`#pragma PARAMETER` passes parameters to assembler functions via the 16-bit general-purpose registers (R0, R1, R2, R3), 8-bit general-purpose registers (R0L, R0H, R1L, R1H) and address registers (A0, A1). In addition, the 16-bit general-purpose registers are combined with the address registers to form 32-bit registers (R3R1, R2R0, A1A0) through which parameters are passed to the assembler functions.

You must enter the prototype declaration of the assembler function before the `#pragma PARAMETER`

declaration.

However, you cannot declare the following parameter types in the `#pragma PARAMETER` declaration:

- Parameters of structure type and union type
- 64-bit integer type (long long) parameters
- Double precision floating-point type (double) parameters

You cannot define return values of the structure or union type for assembler functions.

3.9 long long Type

The compiler supports long long and unsigned long long types data.

This data type is represented as "long long" for signed integer, and "unsigned long long" for unsigned integers.

To create a long long type integer constant, append the suffix LL after the integer value. To create an unsigned long long type integer constant, append the suffix ULL after the integer value.

Table 3.9 Integer Types and Range of Values

Type	Range of values	Data size
char	0 to 255	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	-32768 to 32767	2 bytes
unsigned int	0 to 65535	2 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

3.10 "near/far" Type

The maximum access area for the M16C/80 Series is 16M bytes. NC308 manages this area divided into a near area (from 000000H to 00FFFFH) and a far area (from 000000H to FFFFFFFH). This subsection describes how to map variables and functions to these areas and how to access these areas.

3.10.1 Near and Far Areas

To manage the access area of a maximum of 16M bytes, NC308 divides this area into near and far areas. Table 3.10 shows the characteristics of these areas.

Table 3.10 Near and Far Areas

Area	Description
Near area	This area helps the M16C/80 Series access data efficiently. This 64-Kbyte area extends over absolute addresses from 000000H to 00FFFFH. Stacks and internal RAM are mapped to this area.
Far area	Full memory space accessible from the M16C/80 Series. This 16-Mbyte area extends over absolute addresses from 000000H to FFFFFFFH. Internal ROM is mapped to this area.

3.10.2 Defaults of the "near" and "far" Attributes

For NC308, variables and functions mapped to the near area have the "near" attribute and those mapped to the far area have the "far" attribute. Table 3.11 shows the default attributes of the variables and functions.

Table 3.11 Defaults of the "near" and "far" Attributes

Category	Attribute
Program	Fixed to "far"
RAM data	near (However, the pointer types has the "far" attribute.)
ROM data	"far"
Stack data	Fixed to "near"

To change the defaults of near and far attributes, specify the following options when starting NC308:

- ffar_RAM (-fFRAM): Change the default attribute for the RAM data to "far".
- fnear_ROM (-fNROM): Change the default attribute for the ROM data to "near".
- fnear_pointer (-fNP): Change the default attribute for the pointer type data to "near".

3.10.3 "near" and "far" Specification for Functions

Due to the architecture of the M16C/80 Series, the attribute for NC308 functions is fixed to the far area. If you specify "near", NC308 outputs a warning during compiling and forcibly maps the functions to the far area.

3.10.4 "near" and "far" Specification for Variables

`[storage-class] type-specifier near / far variable-name;`

If you do not specify "near" or "far" in the type declaration, RAM data will be mapped to the near area. Data with const modifier and ROM data will be mapped to the far area.

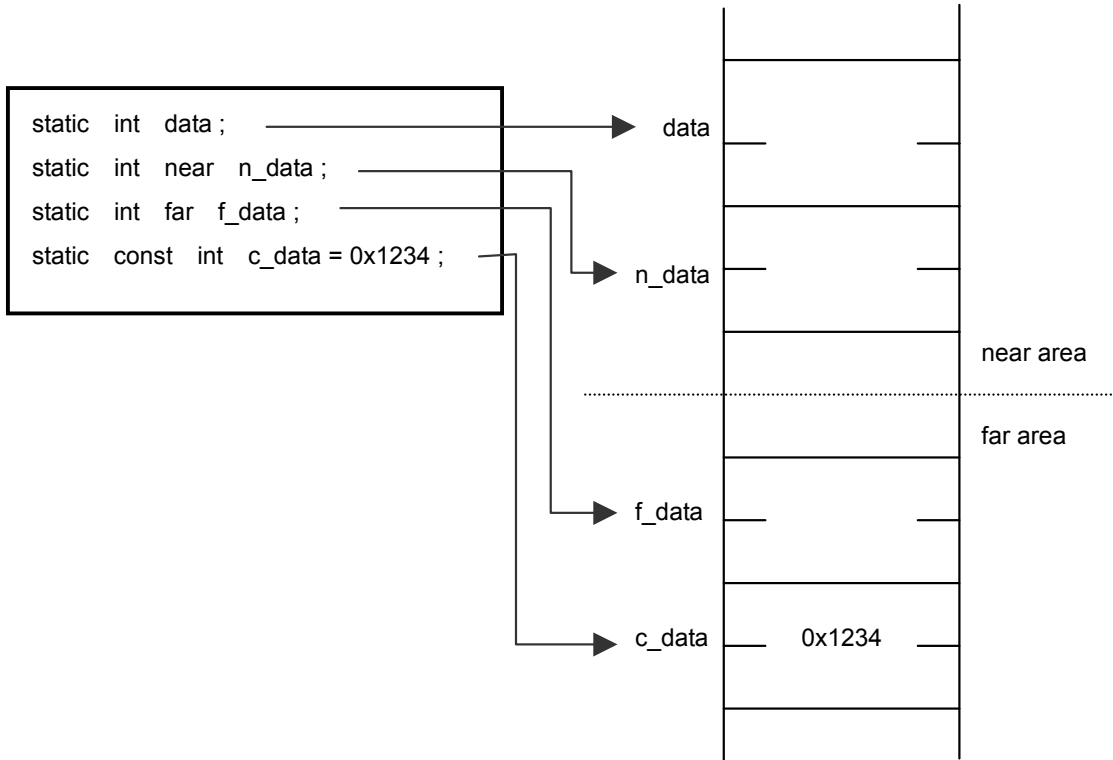


Figure 3.52 Static Variable's near/far

The specification of "near" or "far" does not affect the mapping of auto variable because they are all mapped to the stack area.)

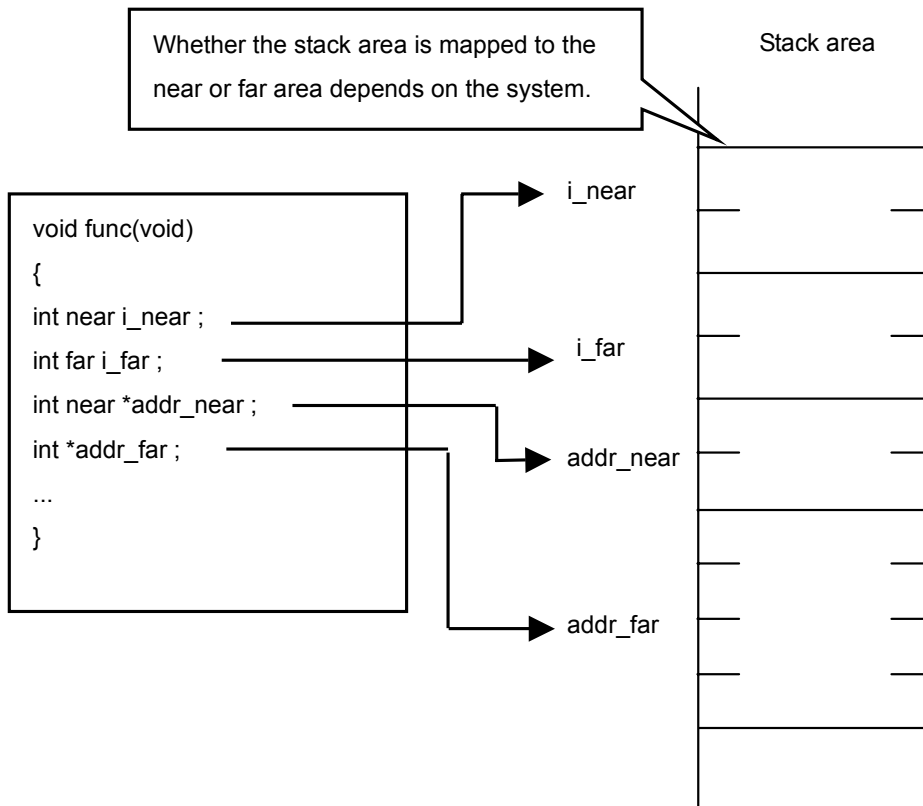


Figure 3.53 Auto Variable's near/far

3.10.5 "near" and "far" Specification for Pointers

Specify "near" or "far" for a pointer to specify the size of the address to be stored in the pointer and the area to which the pointer is mapped.

(1) Specifying the size of the address to be stored in the pointer

If you do not specify anything, the pointer is handled as a 32-bit (4-byte) pointer variable pointing to the variable in the far area.

```
[storage-class] type-specifier near / far * variable-name;
```

near: The size of the address to be stored in the pointer variable is 16 bits.

far: The size of the address to be stored in the pointer variable is 32 bits.

```
int near *near_data ;  
int far *far_data ;
```

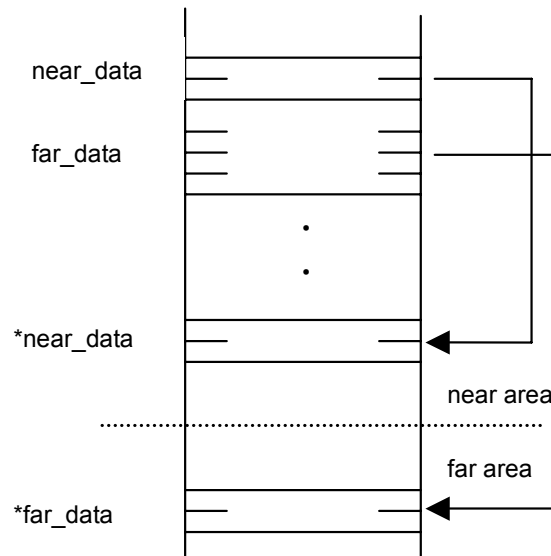


Figure 3.54 Specifying the Address Size Stored in the Pointer

(2) Specifying the area to which the pointer is mapped

If you do not specify anything, the pointer variable is mapped to the near area.

```
[storage-class] type-specifier * near/far variable-name;
```

near: Map the area for the pointer variable to the near area.

far: Map the area for the pointer variable to the far area.

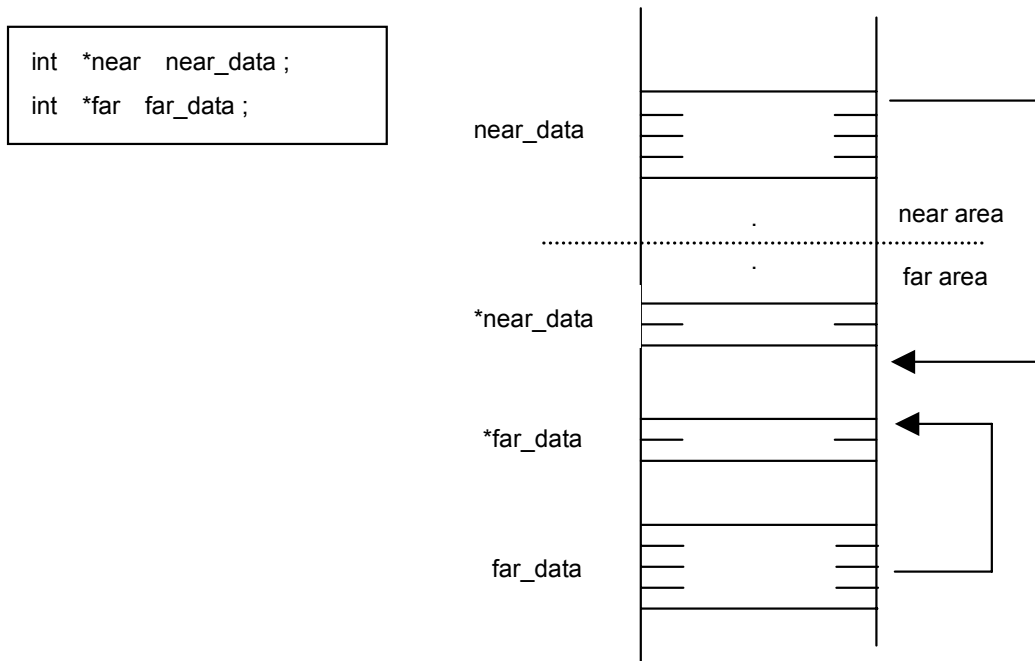


Figure 3.55 Specifying the Pointer Mapping Area

3.10.6 Difference in Pointer's "near/far" Specification between NC308 and NC30

For the NC30 C compiler for the M16C/60 and M16C/20 Series, the "near" attribute is assumed for all pointers if "near" or "far" is not specified. For NC308, if nothing is specified for the size of the address to be stored in the pointer, the size of the pointer variable is assumed to be 32 bits (4-bytes). The pointer is then handled as a pointer variable pointing to the variable in the far area.

3.10.7 Assigning Variable Address in the Far Area to the "near" Pointer

If an attempt is made to assign a variable address in the far area to the "near" pointer, NC308 outputs a warning message, indicating that the pointer will be assigned, ignoring higher-order part of the address.

NC308 also outputs a warning message indicating that the "far" pointer is explicitly or implicitly converted to the "near" pointer.

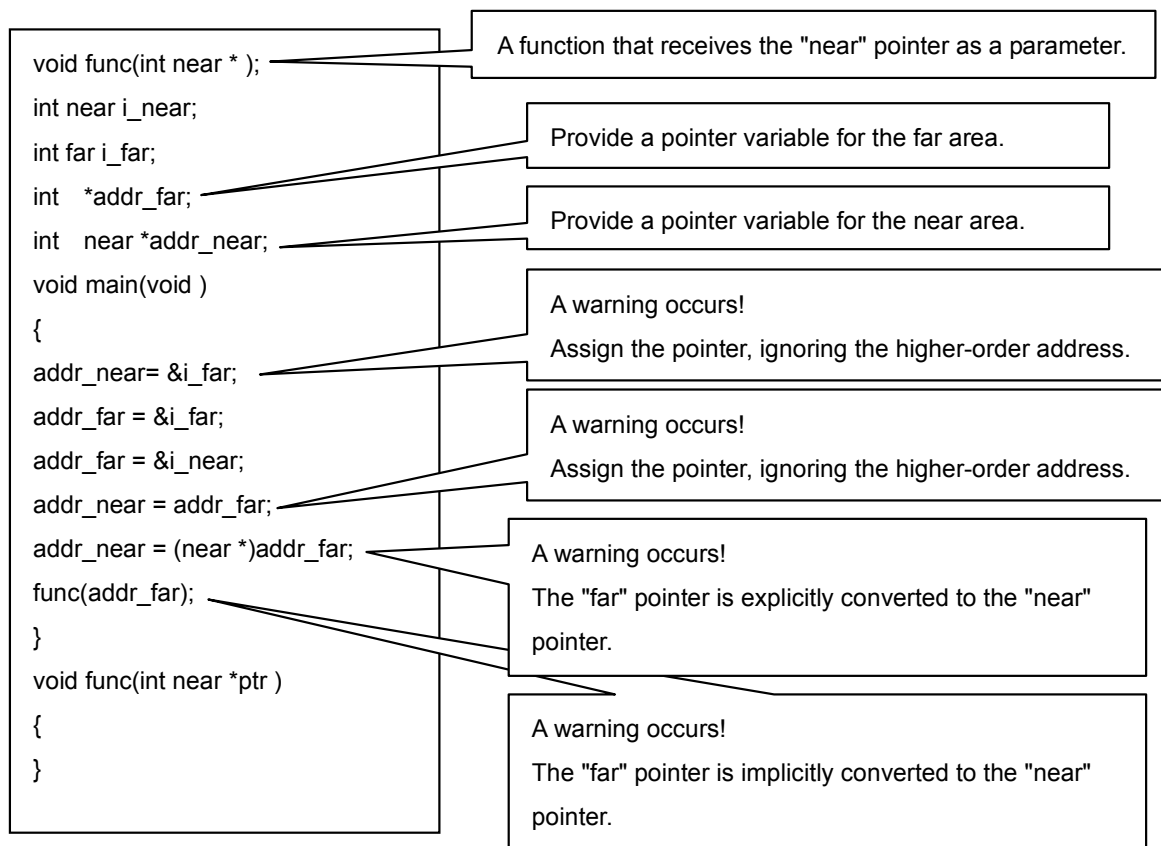


Figure 3.56 Assigning a Variable Address in the far Area to the near Pointer

3.11 Inline Expansion

You can specify an inline storage class the way as you would for C++. By specifying the inline storage class for functions, you can perform inline expansion for the functions.

3.11.1 Overview of the Inline Storage Class

The inline storage class specifier declares that the function is subject to inline expansion. For the functions specified for the inline storage class, the code is directly embedded at the assembly language level.

3.11.2 Format of an Inline Storage Class Declaration

In a declaration, enter an inline storage class specifier using the same syntax as for the static, extern type storage class specifier. Figure 3.57 shows the declaration format.

```
inline $\Delta$ type-specifier $\Delta$ function;
```

Figure 3.57 Declaration of an inline Storage Class

Figure 3.58 shows an example of a function declaration. Figure 3.59 shows the compiled result.

```
inline int func(int i)           ← Declaration and definition part of the inline function
{
    return i++;
}
void main()
{
    int s;
    s = func(s);                 ← Calling part of the inline function
}
```

Figure 3.58 Sample Program for the Inline Function

```

_LANG 'C','X.XX.XX','REV.X'
;## NC308 C Compiler OUTPUT
;## ccom308 Version X.XX.XX
;## COPYRIGHT(C) XXXX(XXXX-XXXX) RENESAS TECHNOLOGY CORPORATION
;## ALL RIGHTS RESERVED AND RENESAS SOLUTIONS CORPORATION ALL RIGHTS
RESERVED
;## Compile Start Time Thu April 10 18:40:11 1995,1996,1997,1998,1999,
2000,2001,2002,2003
;## COMMAND_LINE: ccom308 D:¥MTOOL¥nc308wa5¥TMP¥sss.i -o .¥smp.a30
d S
;## Normal Optimize O F F
;## ROM size Optimize O F F
;## Speed Optimize O F F
;## Default ROM is f a r
;## Default RAM is near
.GLB __SB__
.SB __SB__
.FB 0
;## # FUNCTION func
;## # FUNCTION main
;## # FRAME AUTO ( s) size 2, offset -4
;## # FRAME AUTO ( i) size 2, offset -2
;## # ARG Size(0) Auto Size(4) Context Size(8)
.SECTION program,CODE,ALIGN
.file 'smp.c'
.align
.line 7
;## # C_SRC : {
.glb _main
_main:
    enter #04H
    pushm R1
    .line 9
;## # C_SRC : s = func(s);
    mov.w -4[FB],R0 ; s
    .line 2
;## # C_SRC : {
    mov.w R0,-2[FB] ; i
    .line 3
;## # C_SRC : return i++;
    mov.w R0,R1
    add.w #0001H,R0
    .line 9
;## # C_SRC : s = func(s);
    mov.w R1,-4[FB] ; s
    .line 10
;## # C_SRC : }
    popm R1
    exitd
E1:
.END
;## Compile End Time Tue Jul 16 13:12:00 20xx

```

← The inline function is embedded.

Figure 3.59 Compiled Results of the Sample Program

3.11.3 Rules for the Inline Storage Class

Note the following when specifying the inline storage class:

- About inline function parameters
You cannot use the structure or union types for parameters of the inline functions.
If you use these types, a compile error occurs.
- About the indirect call of inline functions
You cannot perform the indirect call of an inline function. A specification of an indirect call causes a compile error.
- About the recursive call of inline functions
You cannot the recursive call of an inline function. A specification of a recursive call causes a compile error.
- About the definition of an inline function
When you specify the inline storage class for a function, you must define the body of the function in addition to the declaration. This body definition must be included in the same file as the functions.

Figure 3.60 shows a coding that the compiler handles as an error.

```
inline void func(int i);
void main( void )
{
    func(1);
}

[Error message]
[Error(ccom):smp.c,line 5] inline function's body is not declared previously
==> func(1);
Sorry, compilation terminated because of these errors in main().
```

Figure 3.60 Example of Inappropriate Coding of Inline Function (1)

If you used a function as an ordinary one and then define it as an inline function, the inline specification is disabled and all functions are handled as static functions. In this case, the compiler outputs a warning.

```
int func(int i);
void main( void ){
    func(1);
}
inline int func(int i){
    return i;
}
[Warning message]
[Warning(ccom):smp.c,line 9] inline function is called as normal function before
,change to static function
```

Figure 3.61 Example of Inappropriate Coding of Inline Function (2)

- About the address of an inline function

Since the inline function itself does not have an address, using the & operator for the inline function causes an error.

```
inline int func(int i)
{
    return i;
}
main()
{
    int (*f)(int);
    f = &func;
}
[Error message]
[Error(ccom):smp.c,line 10] can't get inline function's address by '&' operator
===> f = &func;
Sorry, compilation terminated because of these errors in main().
```

Figure 3.62 Example of Inappropriate Coding of Inline Function (3)

- Declaration of static data

If static data is declared in an inline function, the body of the declared static data is allocated in units of files. For this reason, if the inline function extends over two or more files, different areas are to be accessed.

The static data to be used in the inline function must be declared outside the function. The compiler outputs a warning if a static declaration is found in the inline function. We do not recommend a static declaration in the inline function.

```
inline int func( int j)
{
static int i = 0;
i++;
return i + j;
}
[Warning message]
[Warning(ccom):smp.c,line 3] static valuable in inline function
===> static int i = 0;
```

Figure 3.63 Example of Inappropriate Coding of Inline Function (4)

- Debug information

The compiler does not output C-language level debug information for inline functions.

This means that the inline functions are debugged at the assembly language level.

Section 4. Using the High-performance Embedded Workshop

4.1 Specifying Options in the High-performance Embedded Workshop

You can specify options from the Options menu. The following shows how to specify options from Renesas Integrated Development Environment. Select "Renesas M32C Standard Toolchain" from the Options menu.

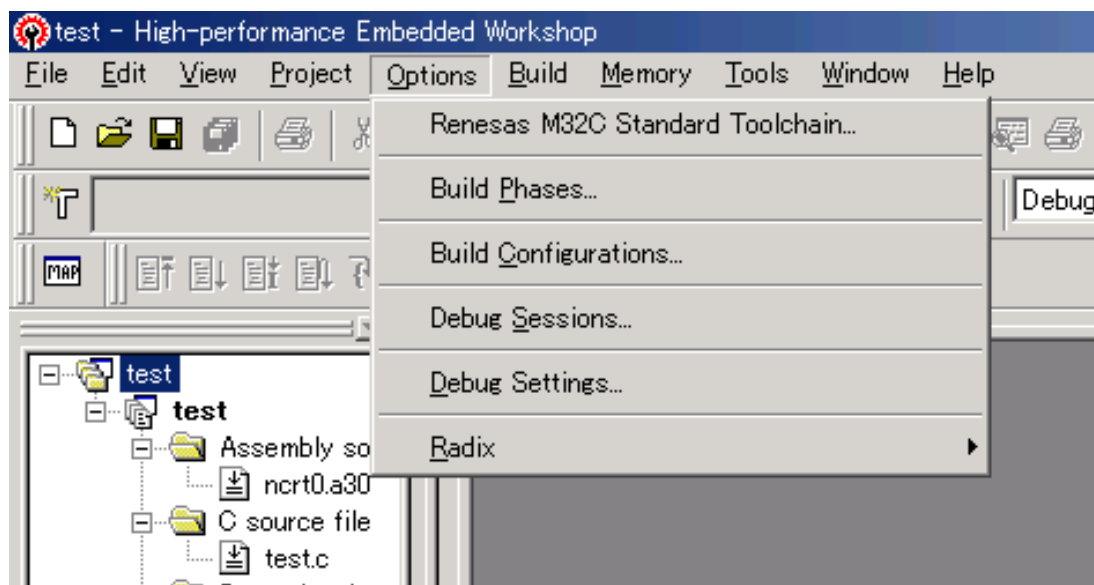


Figure 4.1 High-performance Embedded Workshop Options Menu

4.1.1 C Compiler Options

Select the [C] tab in the [Renesas M32C Standard Toolchain] dialog box.

- Category:[Source]

Table 4.1 Correspondence between Items on the Category:[Source] Dialog Box and Compiler Options

Dialog Box	Option
Show entries for :	
Include files directories	I<directory name>
Defines	D<sub> <sub> : <macro name> [= <string>]
Predefines	U<sub> <sub> : <predefined macro name>

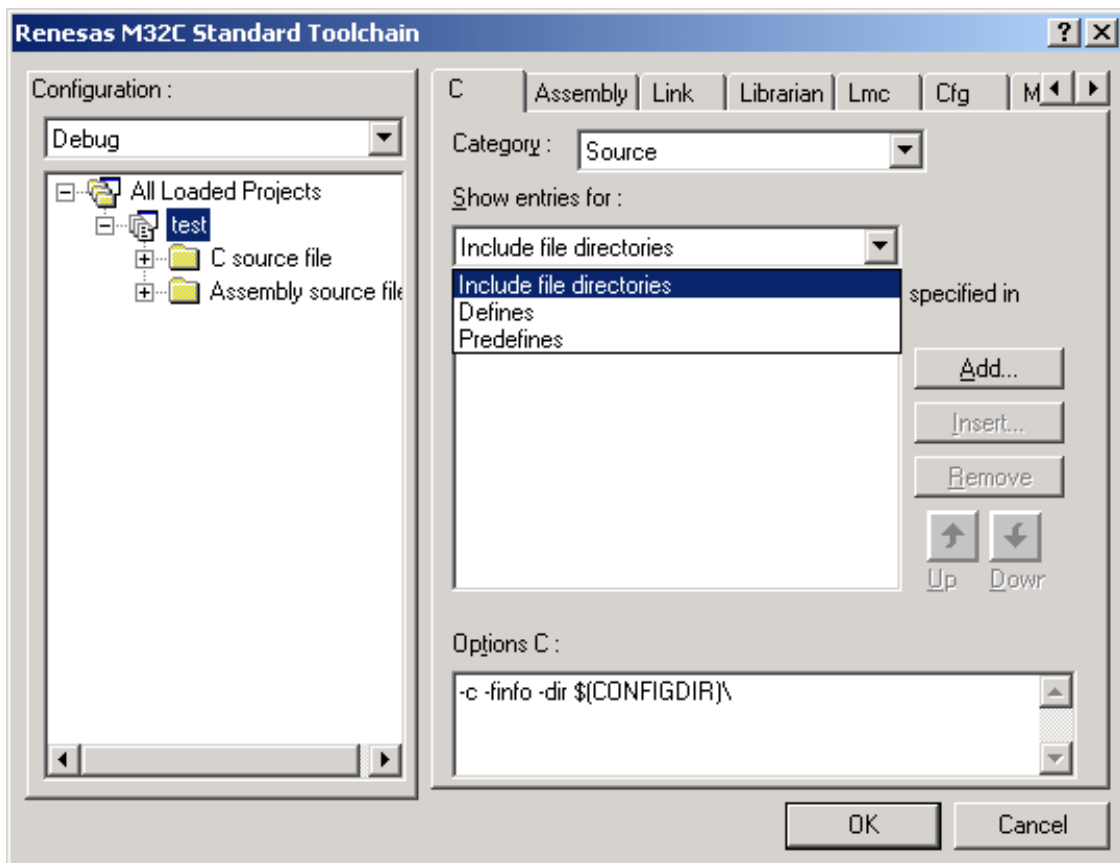


Figure 4.2 Category:[Source] Dialog Box

– Category:[Object]

Table 4.2 Correspondence between Items on the Category:[Object] Dialog Box and Compiler Options

Dialog Box	Option
Output file type : [-c] Relocatable file (*.r30) [-S] Assembly language source file (*.a30) [-P] Preprocessed source file (*.i) [-E] Preprocessed output	c S P E
Debug options : [-finfo] Outputs information needed for Inspector, Stk Viewer, and utl30 [-g] Outputs debugging information. Therefore you can perform C-language-level debugging [-genter] Always outputs an enter instruction when calling a function. Be sure to specify this option when using the debugger's stack trace function [-gno_reg] Suppresses the output of debugging information for register variables	finfo g genter gno_reg
[-dir] Specifies the directory to output the file(s) to :	dir<directory name>

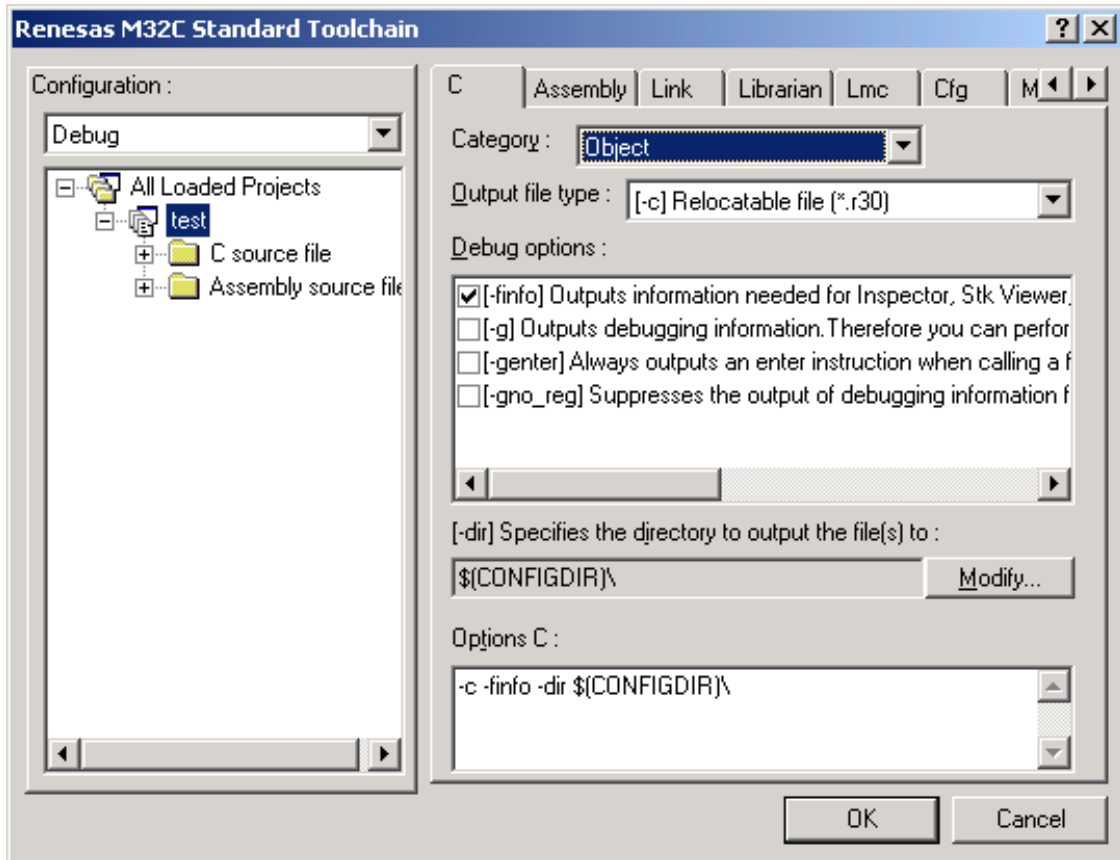


Figure 4.3 Category:[Object] Dialog Box

- Category:[List]

Table 4.3 Correspondence between Items on the Category:[List] Dialog Box and Compiler Options

Dialog Box	Option	Shortcut
[-dS] Outputs C source code as comments in the output assembly language source list.	dsource	dS
[-dSL] Outputs C source code as comments in the output assemble list.	dsource_in_list	dSL

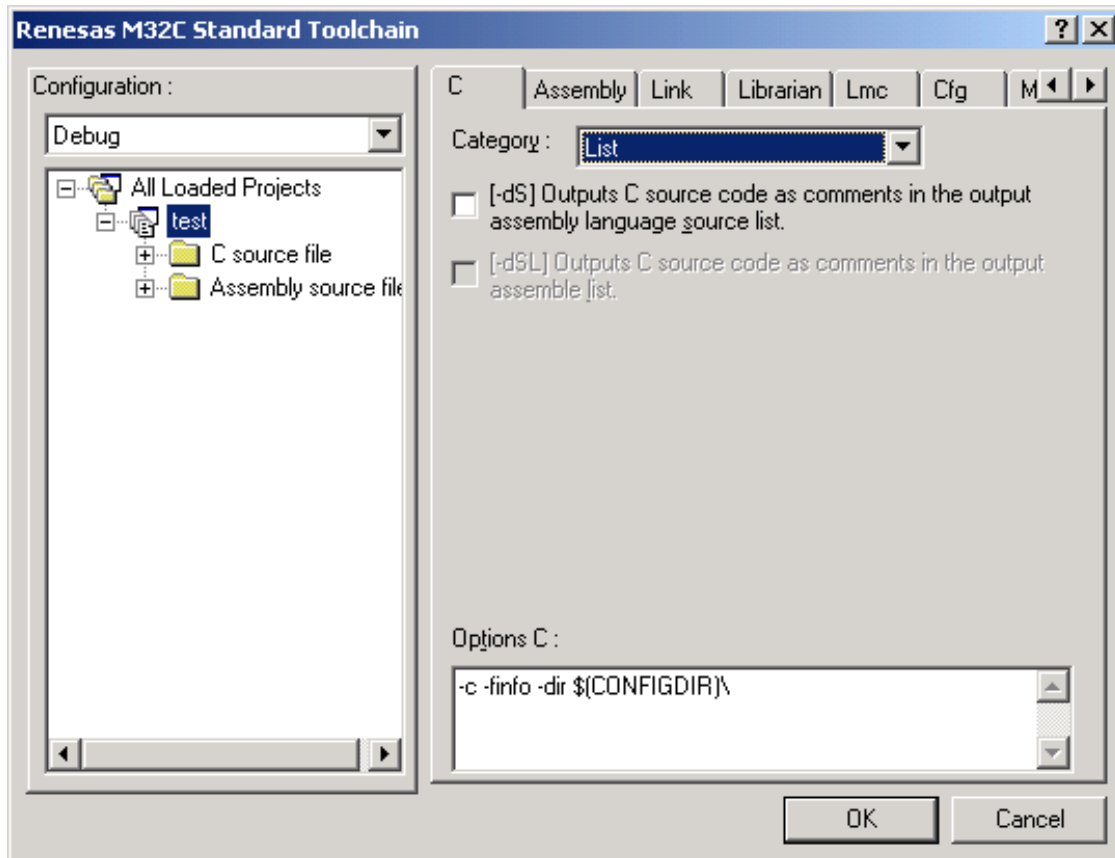


Figure 4.4 Category:[List] Dialog Box

– Category:[Optimize]

Table 4.4 Correspondence between Items on the Category:[Optimize] Dialog Box and Compiler Options

Dialog Box	Option	Shortcut
Optimization level :		
[-O1] Makes -O3,-ONB,-ONBSD,-ONFCF,and -ONS valid	O1	None
[-O2] Makes no difference with -O1	O2	None
[-O3] Optimizes speed and ROM size to the maximum	O3	None
[-O4] Makes -O3 and Oconst valid	O4	None
[-O5] Effect the best possible optimization	O5	None
Size or speed :		
[-OR] ROM size followed by speed	OR	None
[-OS] Speed followed by ROM size	OS	None

Dialog Box	Option	Shortcut
<p>Miscellaneous options :</p> <p>[-OC] Performs optimization by replacing references to theconst-qualified external variables with constants</p> <p>[-OCBTW] Compares consecutive bytes of data at contiguous addresses in words</p> <p>[-OFFTI] In line deployment is performed to the function described ahead.</p> <p>[-OFTI] A floating point runtime library function is developed.</p> <p>[-OGJ] Optimizes the branch instruction which refers to the global label.</p> <p>[-ONA] Suppresses execution of assembler optimizer aopt308</p> <p>[-ONB] Suppresses optimization based on grouping of bit manipulations</p> <p>[-ONBSD] Suppresses optimization that affects source line information</p> <p>[-ONFCF] Suppresses the constant folding processing of floating point numbers</p> <p>[-ONLOC] Suppresses the optimization that puts consecutive ORs together</p> <p>[-ONS] Inhibits inline padding of standard library functions and modification of library functions</p> <p>[-OSA] Performs optimization to remove stack correction code after calling a function</p> <p>[-OSTI] A static function is treated as an inline function</p>	<p>Oconst</p> <p>Ocompare_byte_to_word</p> <p>Oforward_function_to_inline</p> <p>Ofloat_to_inline</p> <p>Oglb_jump</p> <p>Ono_asmopt</p> <p>Ono_bit</p> <p>Ono_break_source_debug</p> <p>Ono_float_const_fold</p> <p>Ono_logical_or_combine</p> <p>Ono_stdlib</p> <p>Osp_adjust</p> <p>Ostatic_to_inline</p>	<p>OC</p> <p>OCBTW</p> <p>OFFTI</p> <p>OFTI</p> <p>OGJ</p> <p>ONA</p> <p>ONB</p> <p>ONBSD</p> <p>ONFCF</p> <p>ONLOC</p> <p>ONS</p> <p>OSA</p> <p>OSTI</p>
<p>[-OLU] Expands sentences the number of times to loop without loop :</p>	<p>Oloop_unroll=<numeric value></p>	<p>OLU</p>

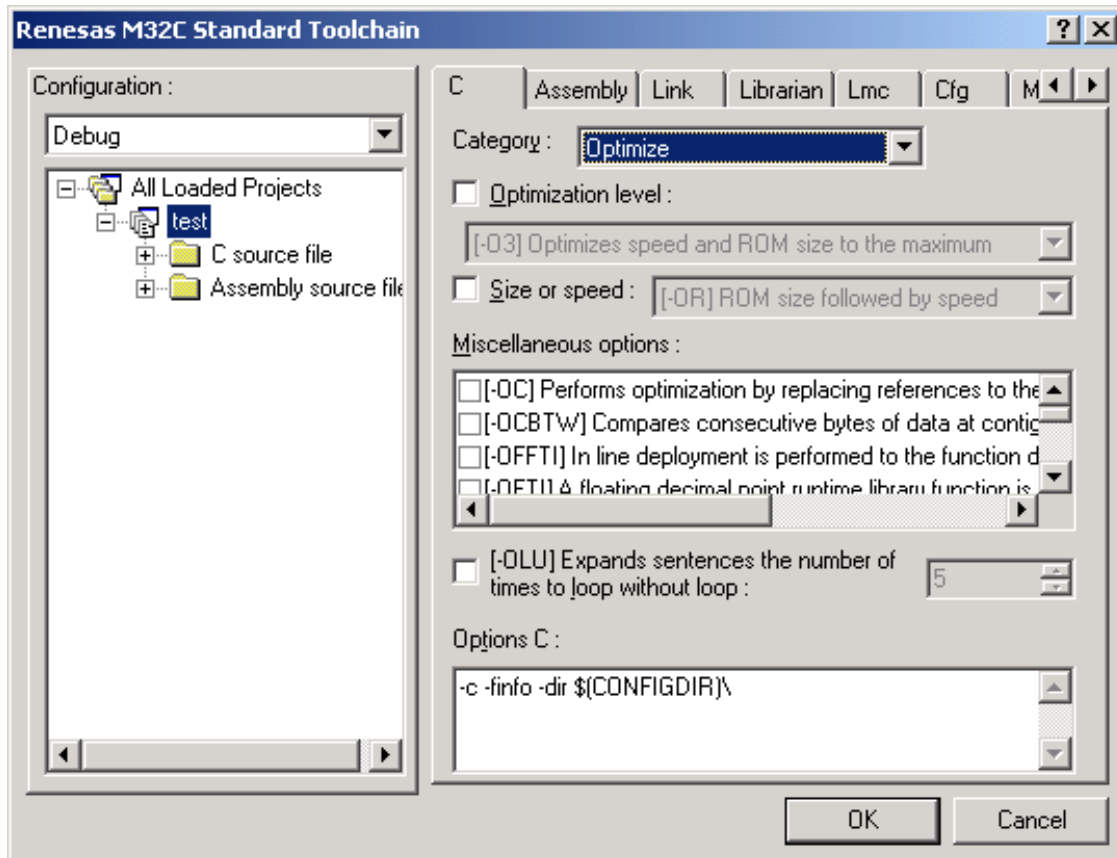


Figure 4.5 Category:[Optimize] Dialog Box

– Category:[Code Modification]

Table 4.5 Correspondence between Items on the Category:[Code Modification] Dialog Box and Compiler Options

Dialog Box	Option	Shortcut
Miscellaneous options :		
[-fansi] Makes -fNRA,-fNRFAN,-fNRI,and -fETI valid	fansi	None
[-fCE] Handles the enumerator type as an unsigned char, not as an int type	fchar_enumerator	fCE
[-fCNR] Does not handle the types specified by const as ROM data	fconst_not_ROM	fCNR
[-fD32] Handles the double type as the float type	fdouble_32	fD32
[-fER] Make register storage class available	fenable_register	fER
[-fETI] Performs operation after extending char-type data to the int type.	fextend_to_int	fETI

Dialog Box	Option	Shortcut
(Extended according to ANSI standards.)		
[-fFRAM] Changes the default attribute of RAM data to far	ffar_RAM	fFRAM
[-fJSRW] Changes the default instruction for calling functions to JRSW	fJSRW	None
[-fMST] Generates special page vector table.	fmake_special_table	fMST
[-fMVT] Generates variable vector table.	fmake_vector_table	fMVT
[-fNA] Does not align the starting address of functions	fno_align	fNA
[-fNAV] Does not regard the variables specified by #pragma ADDRESS as those specified by volatile	fnot_address_volatile	fNAV
[-fNE] Allocate all data to the odd section, with no separating odd data from even data when outputting	fno_even	fNE
[-fNP] Changes the default type of pointer data to near.	fnear_pointer	fNP
[-fNRA] Exclude asm from reserved words.	fnot_reserve_asm	fNRA
(Only _asm is valid.)		
[-fNRFAN] Exclude far and near from reserved words.	fnot_reserve_far_and_near	fNRFAN
(Only _far and _near are valid.)		
[-fNRI] Exclude inline a reserved words.	fnot_reserve_inline	fNRI
(Only _inline is valid.)		
[-fNROM] changes the default attribute of ROM data to near	fnear_ROM	fNROM
[-fNST] To a switch sentence, it always compares and branched code is generated.	fno_switch_table	fNST
[-fSA] When referencing a far-type array, this option calculates subscripts in 16 bits if the total size of the array is within 64K bytes	fsmall_array	fSA
[-fSOS] Outputs the from ROM table corresponding to the section differing from the program section	fswitch_other_section	fSOS
[-fUD] Ignores an overflow when using a divide operation.	fuse_DIV	fUD
[-fESF] If the function specified is a static function, no codes are generated.	ferase_static_function=<function name>	fESF=<function name>

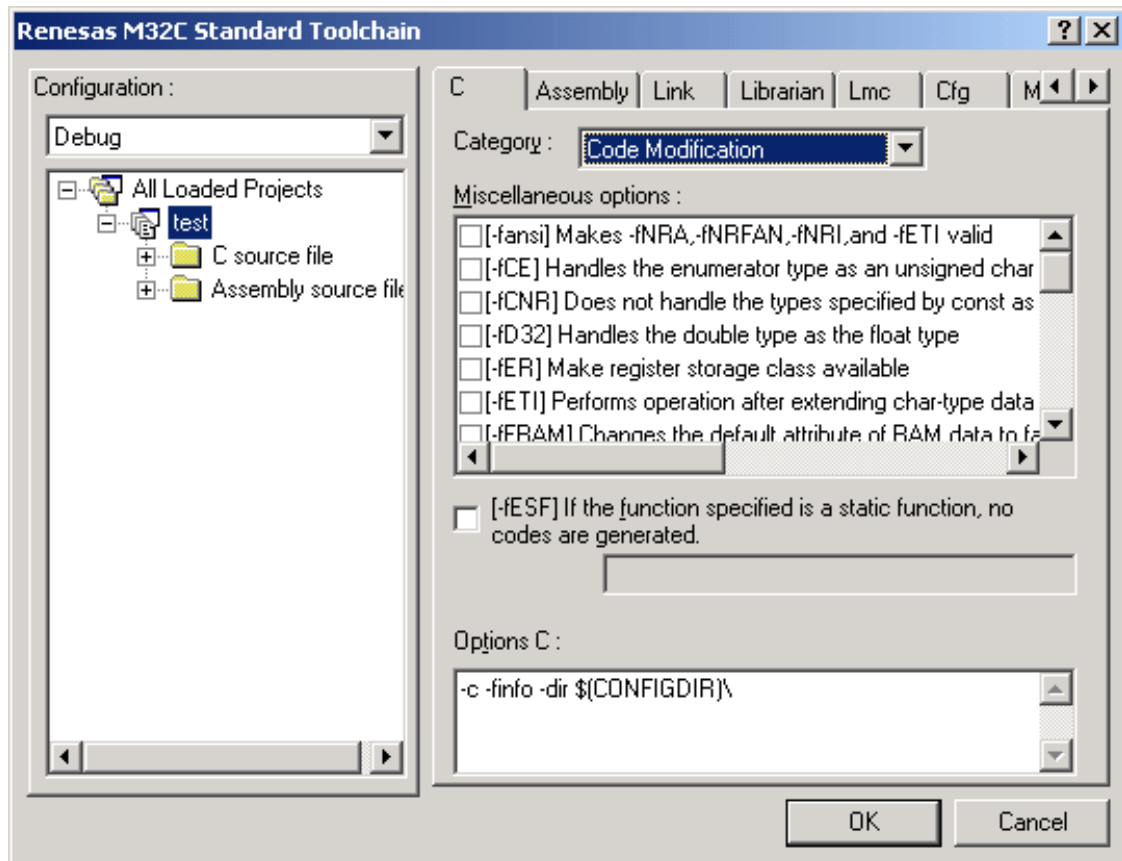


Figure 4.6 Category:[Code Modification] Dialog Box

– Category:[Warning]

Table 4.6 Correspondence between Items on the Category:[Warning] Dialog Box and Compiler Options

Dialog Box	Option	Shortcut
Miscellaneous options :		
[-Wall] Become effective all options for warning	Wall	None
[-WLTS] Outputs an alarm for implicit transfers from large size to smaller size	Wlarge_to_small	WLTS
[-WMT] Outputs error messages to every file	Wmake_tagfile	WMT
[-WNC] Outputs a warning for a comment including /*	Wnesting_comment	WNC
[-WNP] Outputs warning messages for functions without prototype declarations	Wnon_prototype	WNP
[-WNS] Prevents the compiler stopping when an error occurs	Wno_stop	WNS
[-WNUA] Outputs a warning to a function having an unused argument	Wno_used_argument	WNUA
[-WNUF] Outputs a warning for the unused function names.	Wno_used_function	WNUF
[-WNUSF] A static function name is output that does not require code generation	Wno_used_static_function	WNUSF
[-WNWS] Suppresses the warning for missing include file using standard library	Wno_warning_stdlib	WNWS
[-WSAL] Link processing is stopped when warning occurs at the time of a link.	Wstop_at_link	WSAL
[-WSAW] Stops the compiling process when a warning occurs	Wstop_at_warning	WSAW
[-Wstdout] Outputs error messages to the host machine's standard output (stdout)	Wstdout	None
[-WUM] Output the warning for undefined macro in #if .	Wundefined_macro	WUM
[-WUP] Outputs warning messages for non-supported #pragma	Wunknown_pragma	WUP
[-WUV] Outputs the warning for uninitialized auto variables	Wuninitialize_variable	WUV
[-WCMW] Specifies the maximum number of warnings output by ccom30 :	Wccom_max_warning s=<numeric value>	WCMW=<numeric value>
[-WEF] Outputs error messages to the specified file :	Werror_file=<file name>	WEF=<file name>

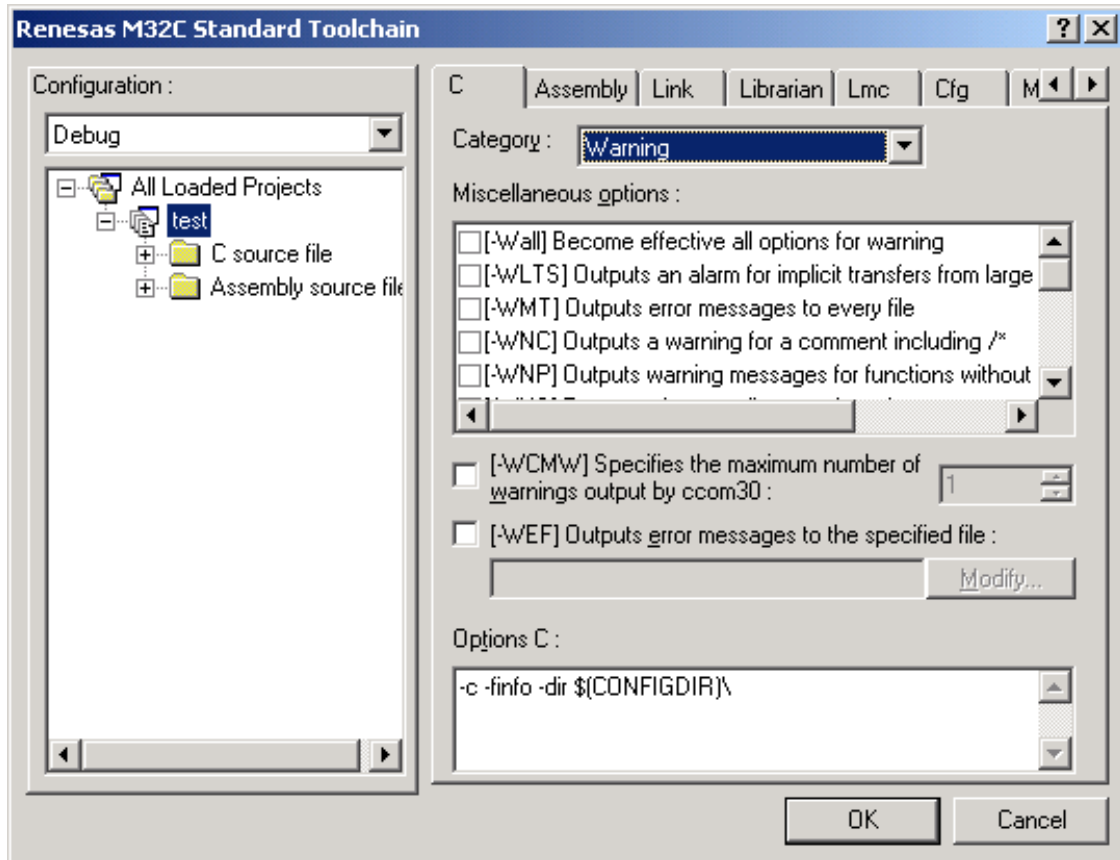


Figure 4.7 Category:[Warning] Dialog Box

– Category:[Other]

Table 4.7 Correspondence between Items on the Category:[Other] Dialog Box and Compiler Options

Dialog Box	Option
Miscellaneous options :	
[-silent] Suppresses the copyright message display at startup	silent
[-v] Displays the name of the command program and the command line during execution	v
[-V] Displays the startup messages of the compiler programs, then finishes processing (without compiling)	V
[-l] Library file :	l<file name>
User defined options :	

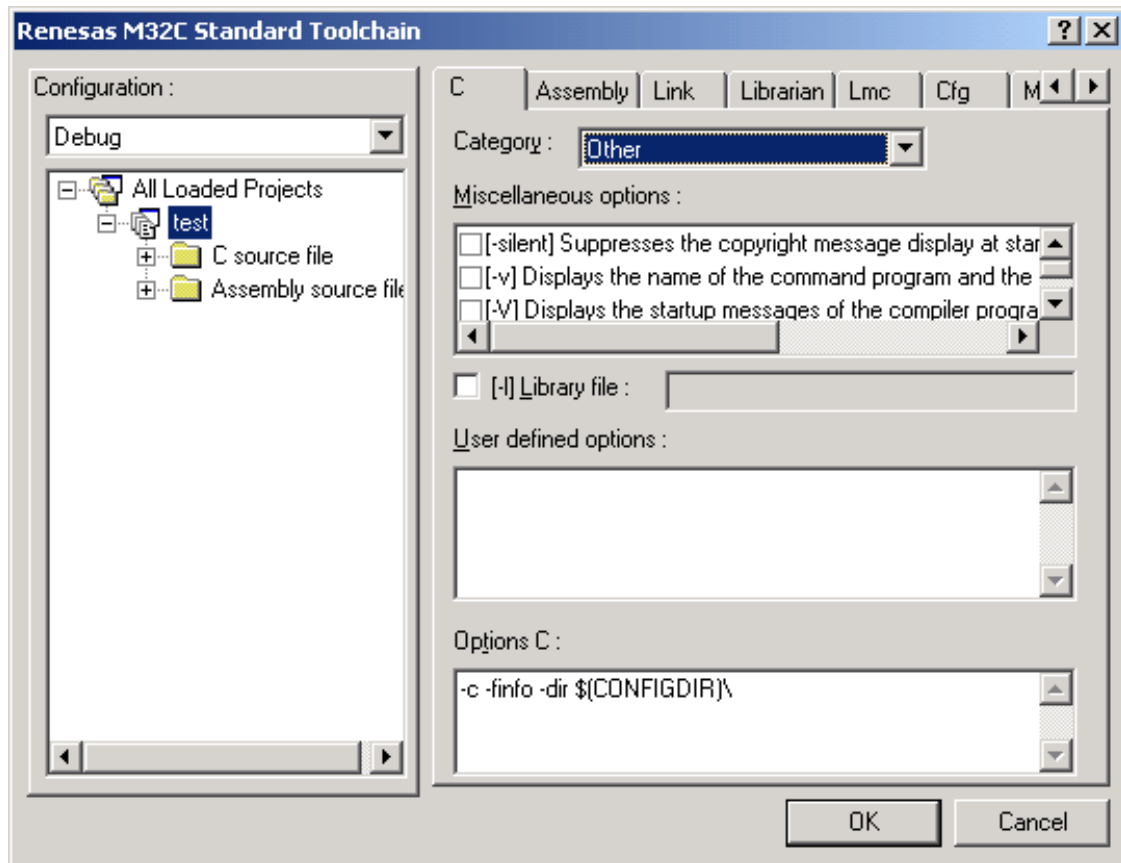


Figure 4.8 Category:[Other] Dialog Box

4.1.2 Assembler Options

Select the [Assembly] tab in the [Renesas M32C Standard Toolchain] dialog box.

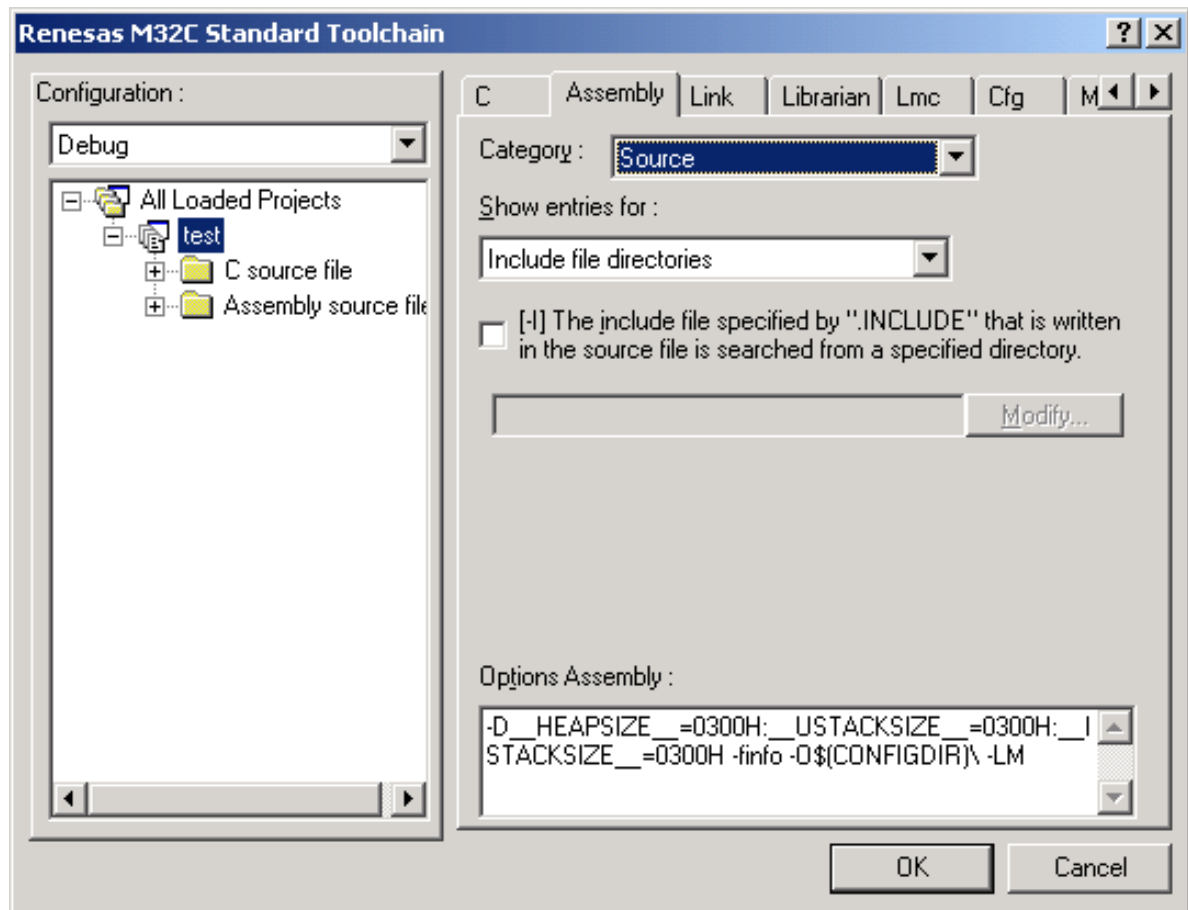


Figure 4.9 [Assembly] Tab Dialog Box

– Category:[Source]

Table 4.8 Correspondence between Items on the Category:[Source] Dialog Box and Assembler Options

Dialog Box	Option
<p>Show entries for :</p> <p>Include file directories</p> <p>[-I] The include file specified by ".INCLUDE" that is written in the source file is searched from a specified directory.</p> <p>Defines</p> <p>[-D__HEAP__=1] Disable heap are in startup (nct0.a30).</p> <p>[-D__STANDARD_IO__=1] Enable initialization for standard I/O library.</p> <p>[-D] Sets constants to symbols :</p>	<p>I<directory name></p> <p>D__HEAP__=1</p> <p>D__STANDARD_IO__=1</p> <p>D<sub></p> <p><sub> : <macro name> [= <string>]</p>

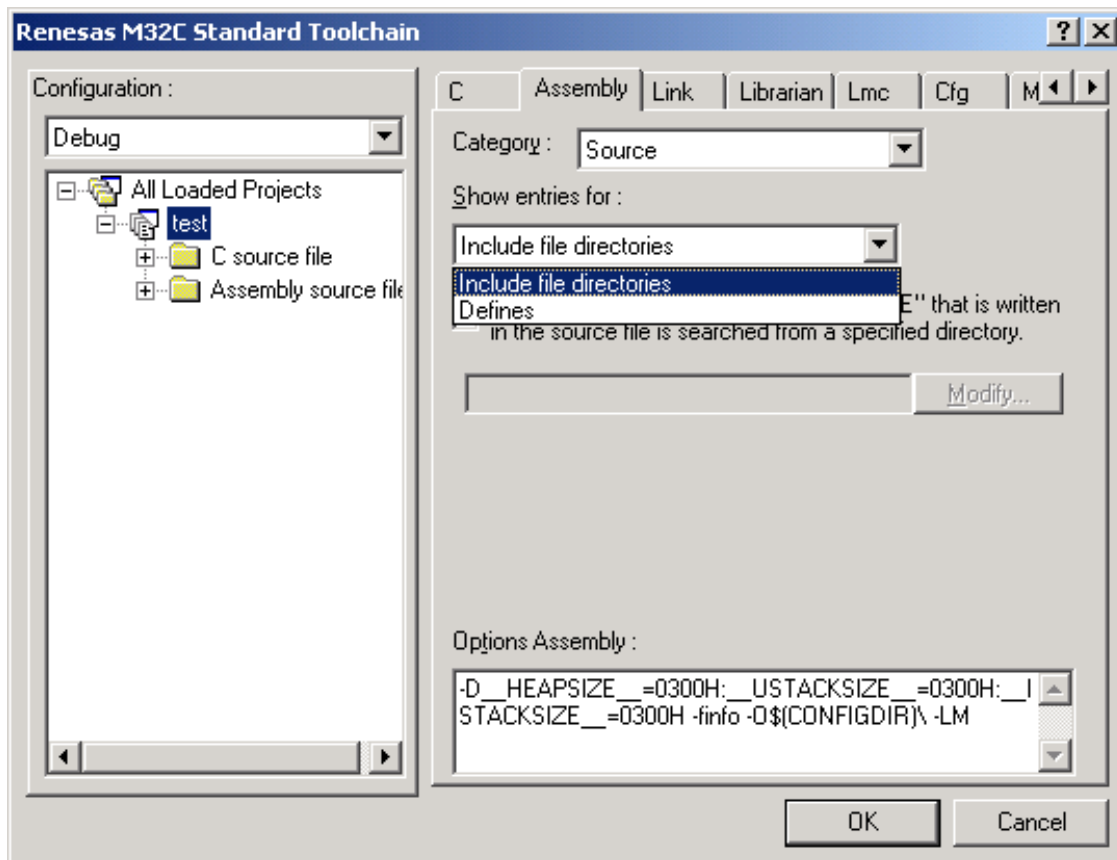


Figure 4.10 Category:[Source] Dialog Box

– Category:[Object]

Table 4.9 Correspondence between Items on the Category:[Object] Dialog Box and Assembler Options

Dialog Box	Option
[-S] Specifies the local symbol information be output.	S
[-SM] Specifies system label and local symbol information output.	SM
[-finfo] Generates inspector information.	finfo
[-N] Disables output of macro command line information.	N
[-mode60] Running AS308 with this parameter to process a program written in AS30 allows some code to be assembled by AS308.	mode60
[-mode60p] Runs structured processor(pre30) and processes parameter -mode60.	mode60p
[-M] Generates structured description command variables in byte type.	M
[-O] Output file directory :	O<directory name>

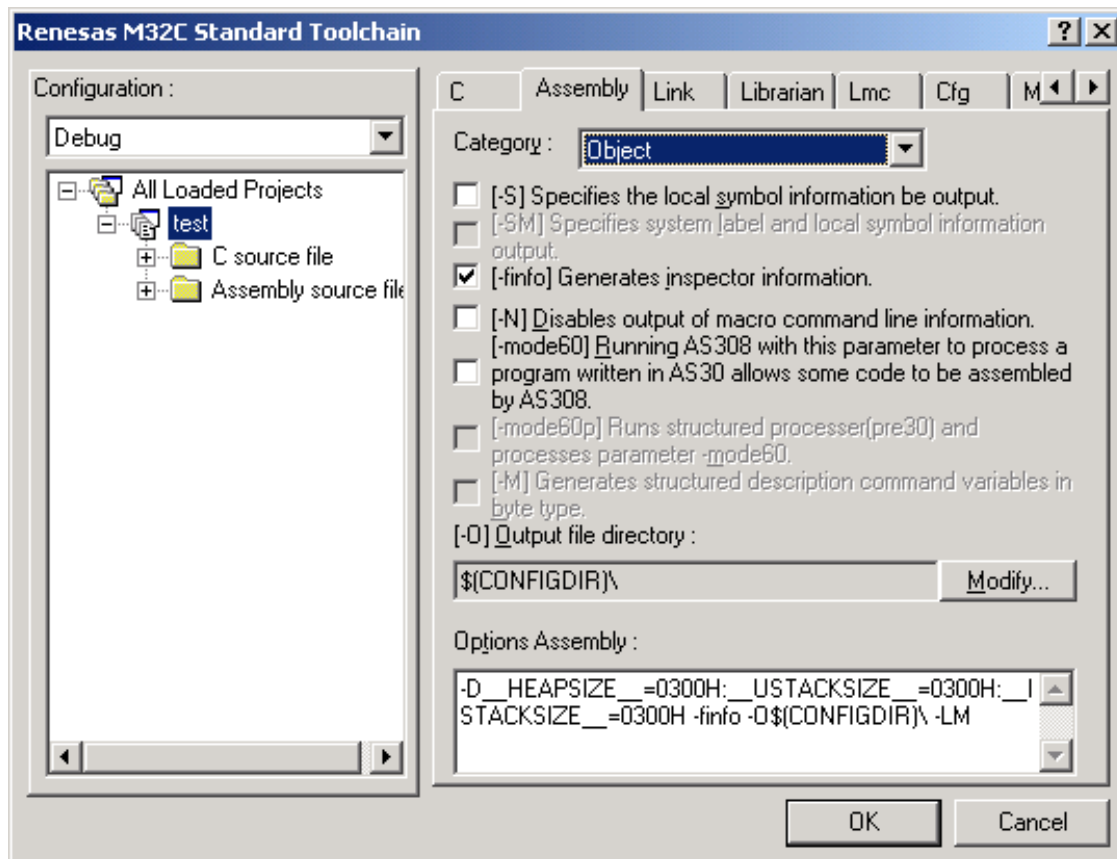


Figure 4.11 Category:[Object] Dialog Box

– Category:[List]

Table 4.10 Correspondence between Items on the Category:[List] Dialog Box and Assembler Options

Dialog Box	Option
[-L] Generates assembler list file.	L
File format: ^{*1}	
[+C] Line concatenation is output directly as is to a list file	LC
[+D] Information before .DEFINE is replaced is output to a list file	LD
[+I] Even program sections in which condition assemble resulted in false conditions are output to the assembler list file	LI
[+M] Even macro description expansion sections are output to the assembler list file	LM
[+S] Even structured description expansion sections are output to the assembler list file	LS
[-H] Header information is not output to an assembler list file.	H
*1: "L" is prefixed to the items selected for the option. Example: When the items [+C], [+D], [+I] are selected, the option is -LCDI.	

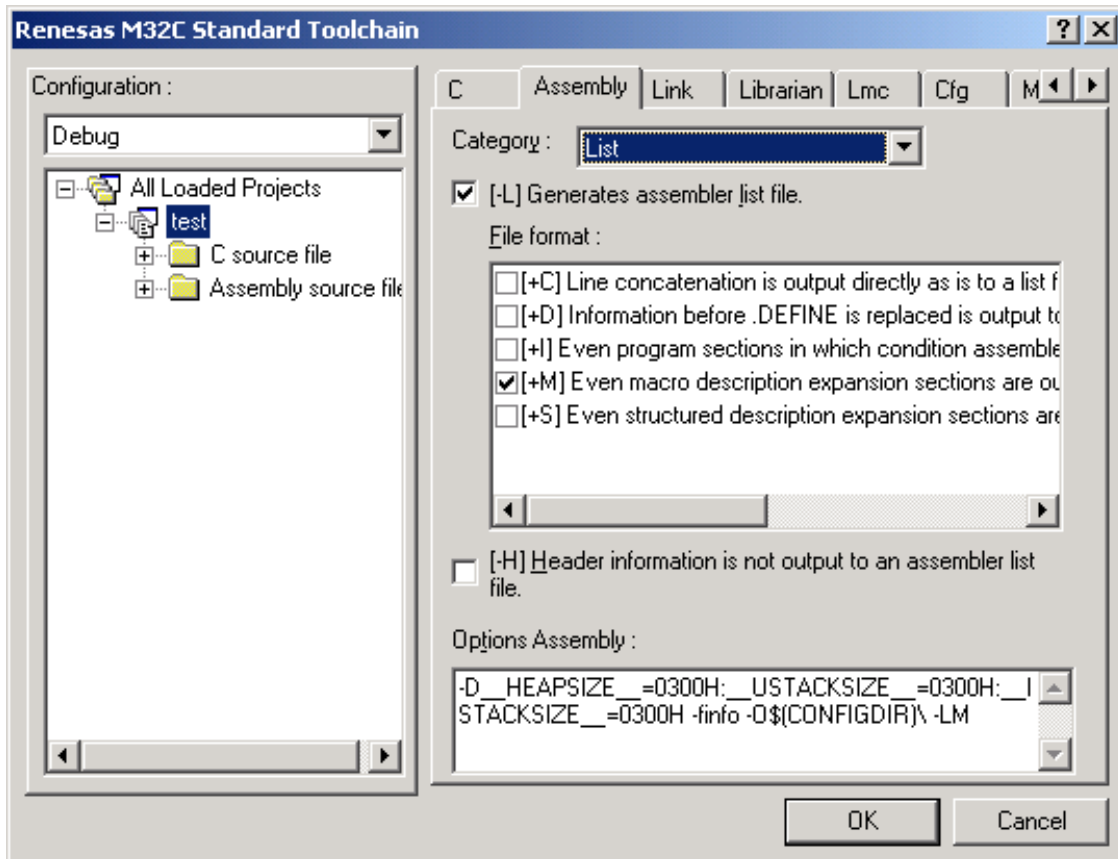


Figure 4.12 Category:[List] Dialog Box

- Category:[Tuning]

Table 4.11 Correspondence between Items on the Category:[Tuning] Dialog Box and Assembler Options

Dialog Box	Option
[-fMST] Generates special page vector table.	fMST
[-fMVT] Generates variable vector table.	fMVT
[-abs16] Selects 16-bit absolute addressing.	abs16
[-JOPT] Optimizes the branch instrument which refers to the global label.	JOPT
[-PATCH_TA] Escaping precautions No.1 on the timer functions for three-phase motor control : Number :	PATCH_TA[n] [n] : Numeric value specified in Number

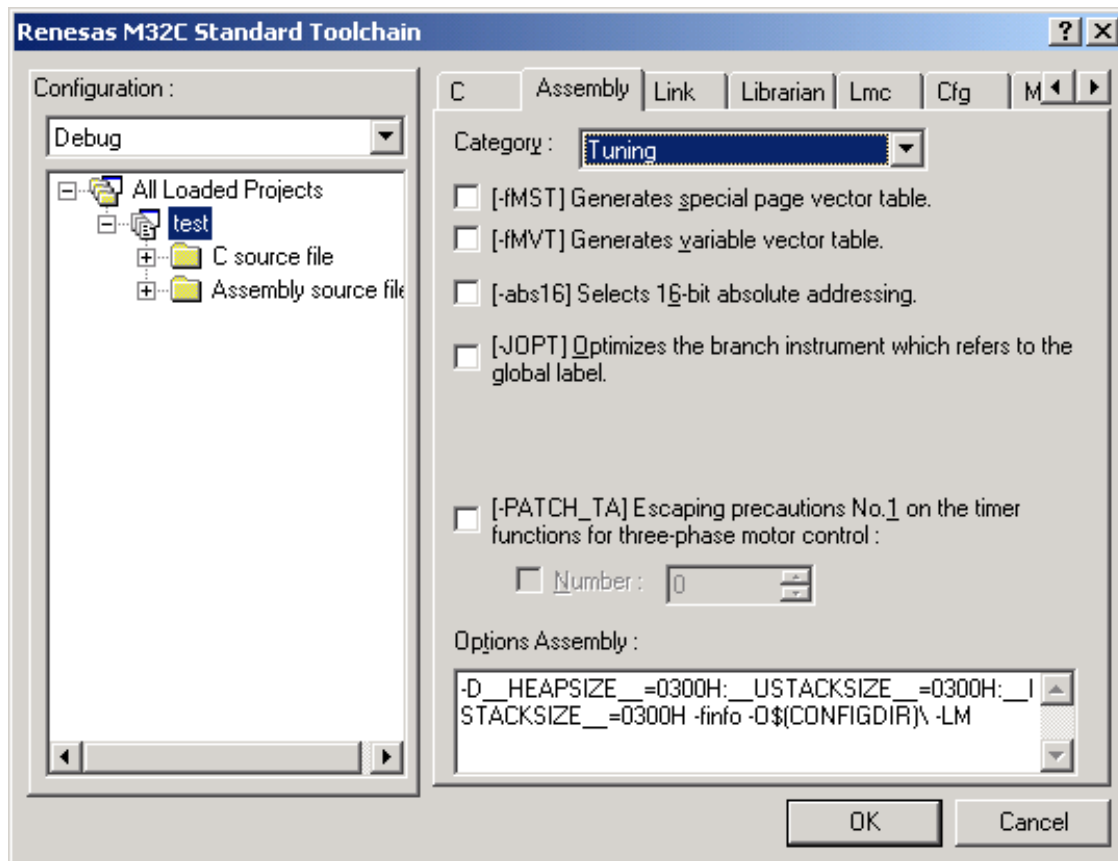


Figure 4.13 Category:[Tuning] Dialog Box

- Category:[Other]

Table 4.12 Correspondence between Items on the Category:[Other] Dialog Box and Assembler Options

Dialog Box	Option
Miscellaneous options :	
[-.] Disables message output to a display screen	.
[-C] Indicates contents of command lines when as30 has invoked mac30 and asp30	C
[-F] Fixes the file name of ..FILE expansion to the source file name	F
[-T] Generates assembler error tag file	T
[-V] Indicates the version of the assembler system program	V
User defined options :	

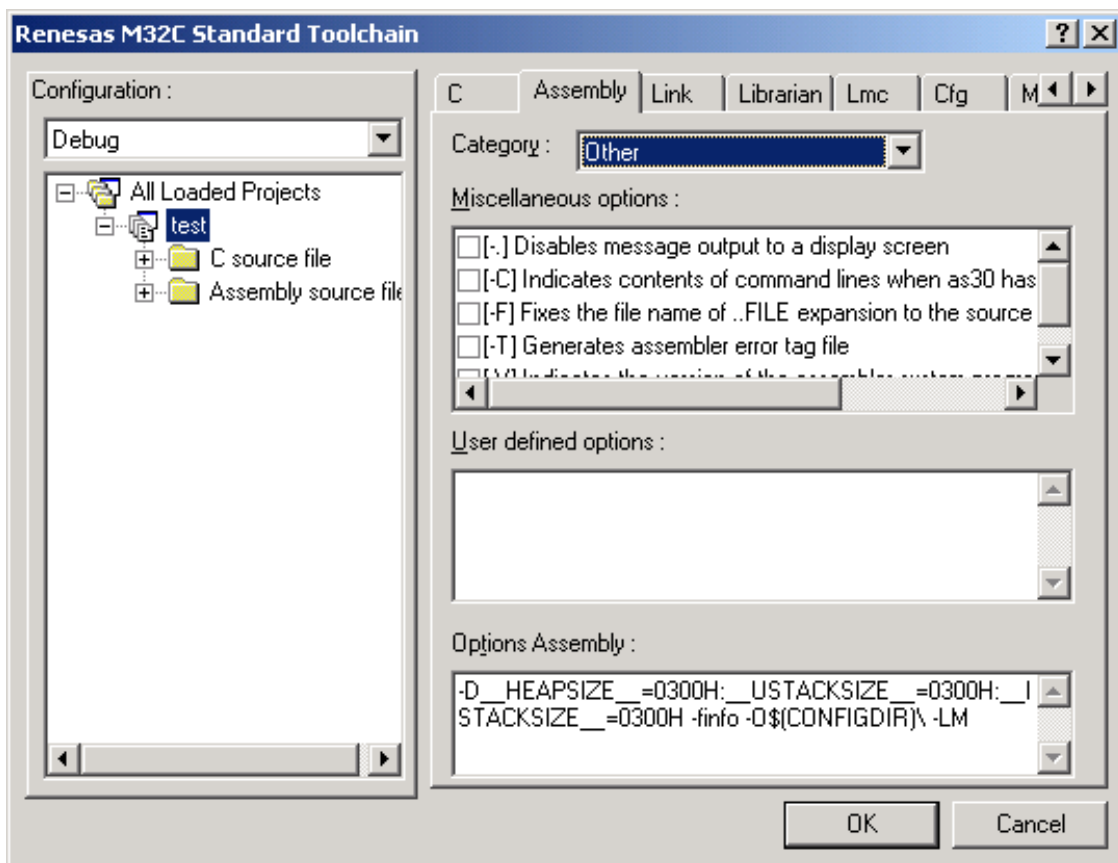


Figure 4.14 Category:[Other] Dialog Box

4.1.3 Linkage Editor Options

Select the [Link] tab in the [Renesas M32C Standard Toolchain] dialog box.

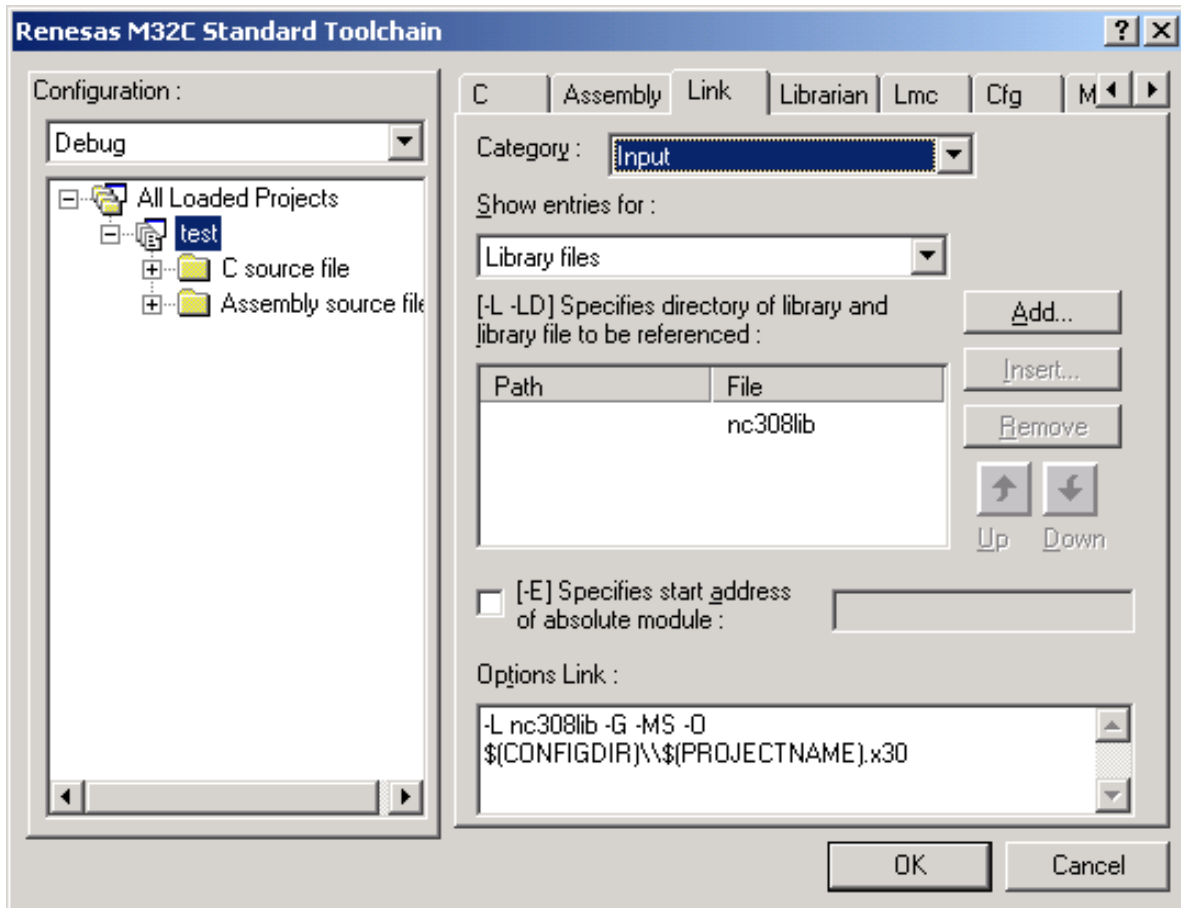


Figure 4.15 [Link] Tab Dialog Box

- Category:[Input]

Table 4.13 Correspondence between Items on the Category:[Input] Dialog Box and Linkage Editor Options

Dialog Box	Option
Show entries for : Library files	LΔ<file name> LDΔ<directory name>
[-E] Specifies start address of absolute module :	EΔ<sub> <sub> : <numeric value> <label name>

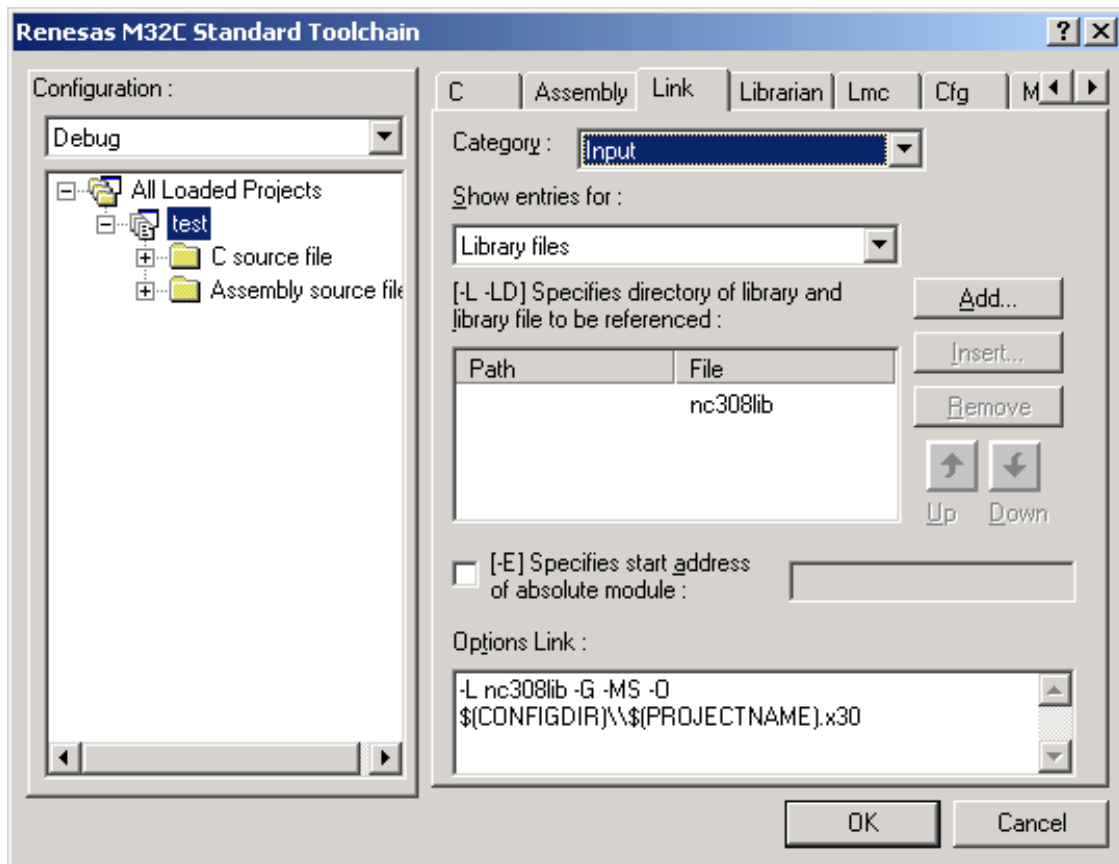


Figure 4.16 Category:[Input] Dialog Box

– Category:[Output]

Table 4.14 Correspondence between Items on the Category:[Output] Dialog Box and Linkage Editor Options

Dialog Box	Option
[-G] Outputs source debug information to absolute module file.	G
[-U] Outputs a warning for the unused function names.	U
[-W] Link processing is stopped when warning occurs at the time of a link.	W
Generate map file :	
None	-
[-M] Generates map file	M
[-MS] Includes symbol information	MS
[-MSL] Includes the fullname of symbol information	MSL
[-O] Specifies absolute module file name :	OΔ<file name>

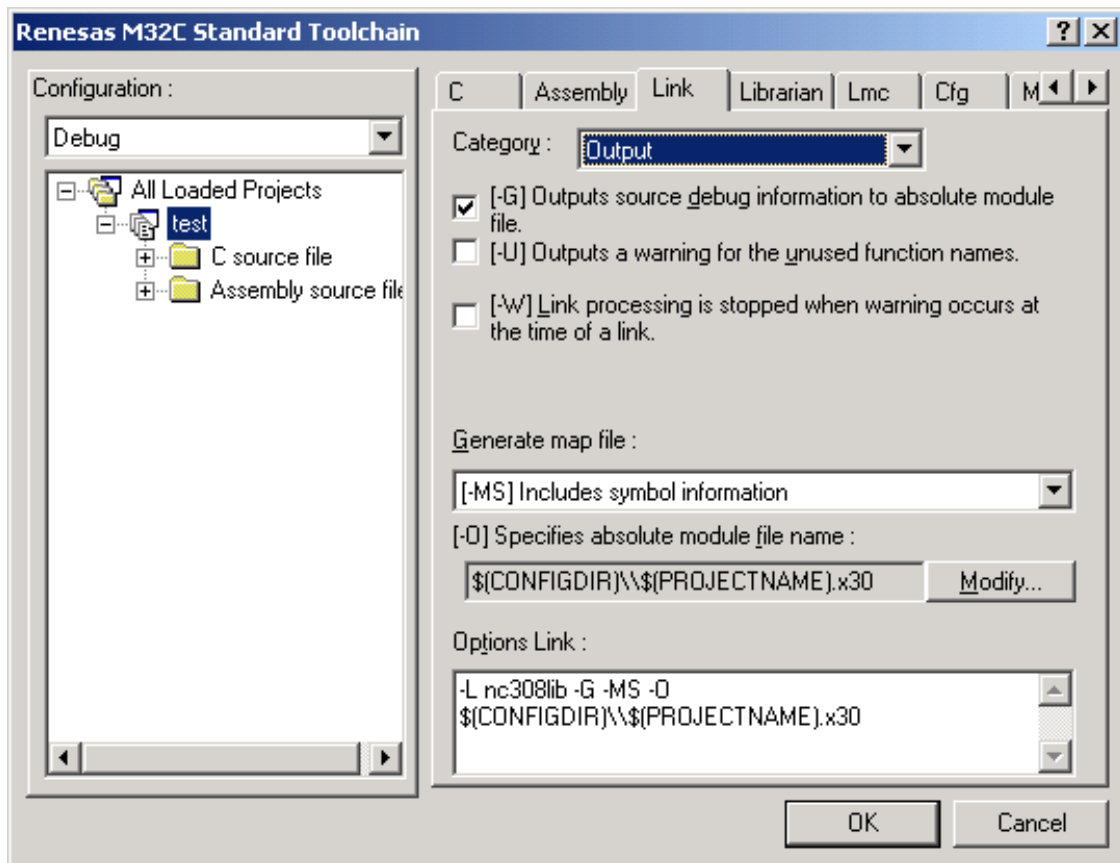


Figure 4.17 Category:[Output] Dialog Box

– Category:[Tuning]

Table 4.15 Correspondence between Items on the Category:[Tuning] Dialog Box and Linkage Editor Options

Dialog Box	Option
[-fMST] Generates special page vector table.	fMST
[-fMVT] Generates variable vector table.	fMVT
[-VECT] Sets the address to the free area at the result of performing automatic generation of a variable vector table.	VECT<sub> <sub> : <numeric value> <label name>
[-JOPT] Optimizes the branch instrument which refers to the global label.	JOPT

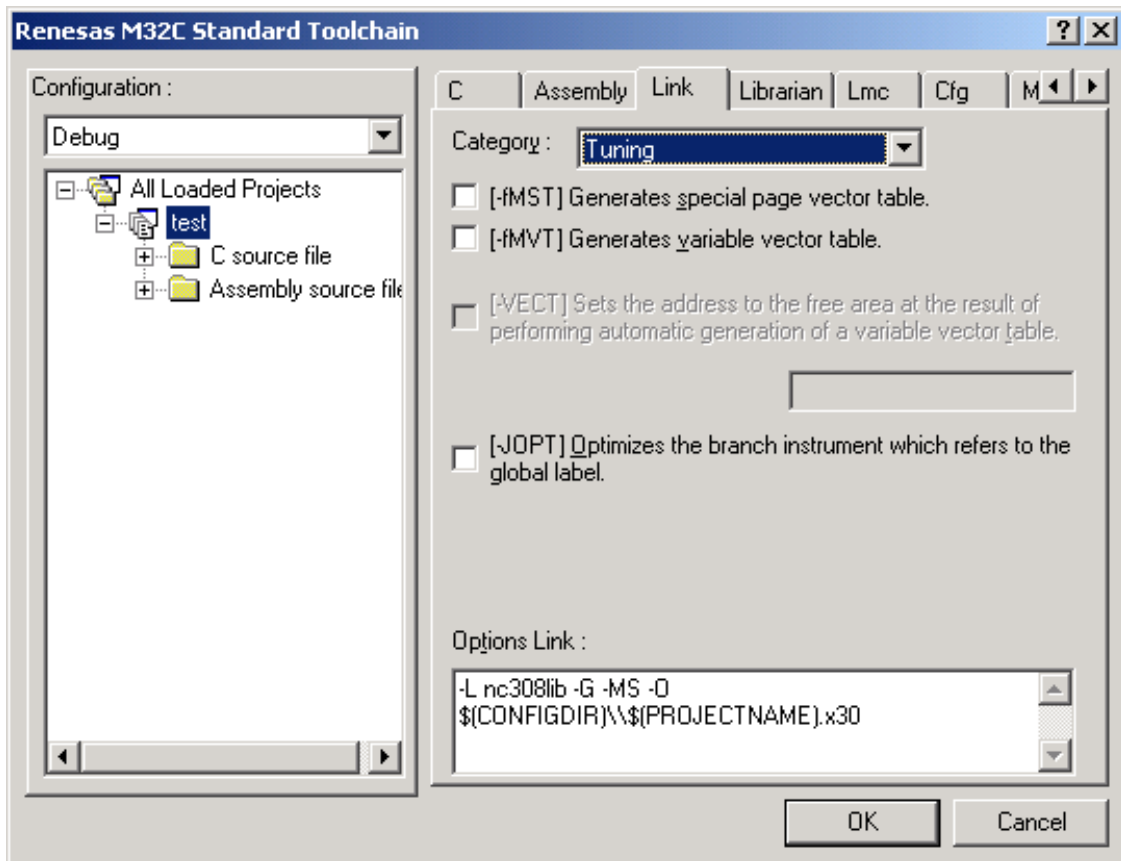


Figure 4.18 Category:[Tuning] Dialog Box

- Category:[Section]

Table 4.16 Correspondence between Items on the Category:[Section] Dialog Box and Linkage Editor Options

Dialog Box	Option
Show entries for :	
Section Order	ORDERΔ<sub>>[,...] <sub> : <section name>[=address]
Section Location	LOCΔ<sub>>[,...] <sub> : <section name>=<address>

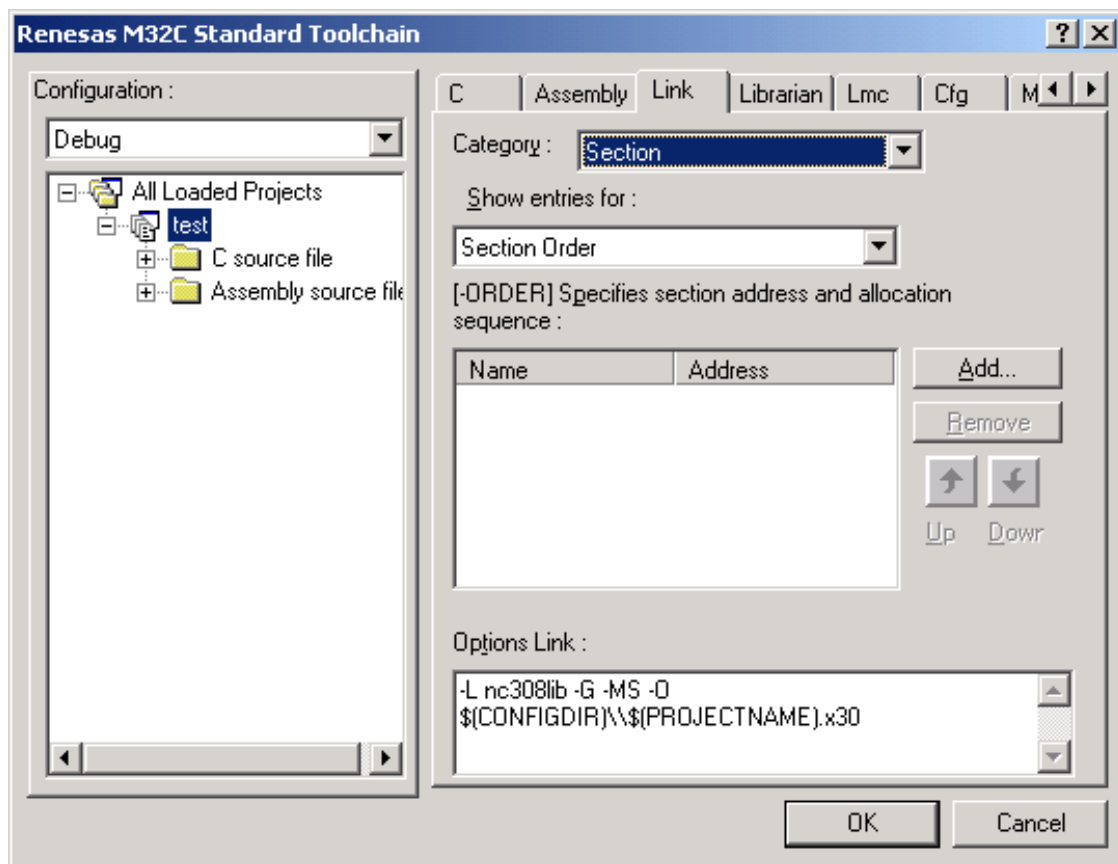


Figure 4.19 Category:[Section] Dialog Box

- Category:[Other]

Table 4.17 Correspondence between Items on the Category:[Other] Dialog Box and Linkage Editor Options

Dialog Box	Option
Miscellaneous options :	
[-.] Disables message output to screen	.
[-NOSTOP] Outputs all encountered errors to screen	NOSTOP
[-T] Generates link error tag file	T
[-V] Indicates version number of linkage editor	V
User defined options :	

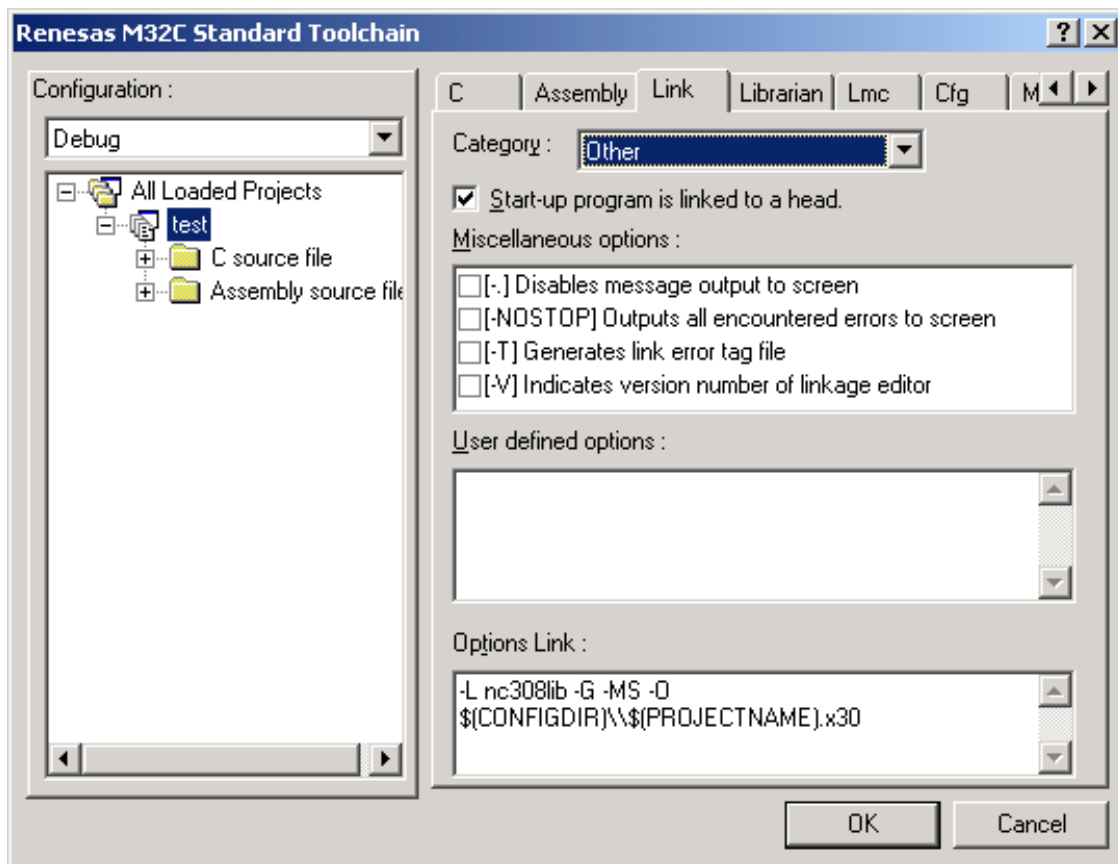


Figure 4.20 Category:[Other] Dialog Box

- Category:[Subcommand file]

Table 4.18 Correspondence between the Item on the Category:[Subcommand file] Dialog Box and Linkage Editor Option

Dialog Box	Option
[@] Use external subcommand file.	@<file name>

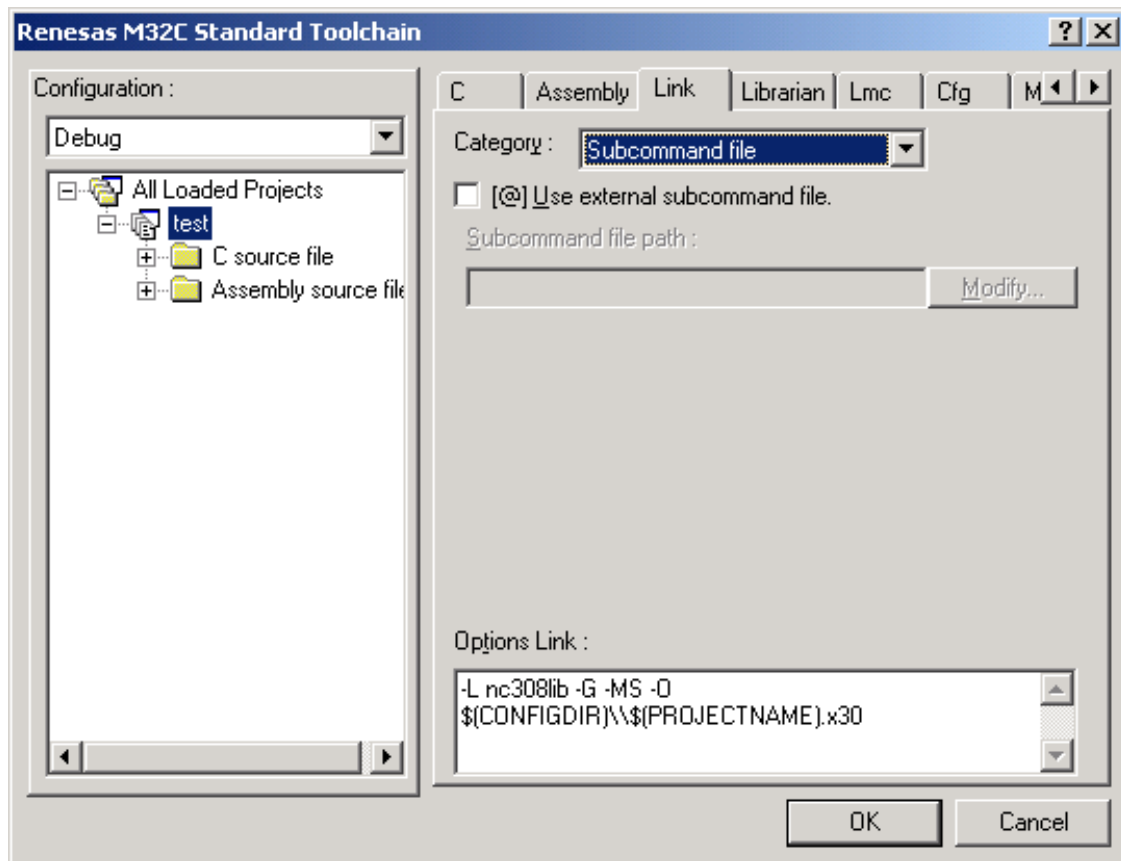


Figure 4.21 Category:[Subcommand file] Dialog Box

4.1.4 Librarian Options

Select the [Librarian] tab in the [Renesas M32C Standard Toolchain] dialog box.

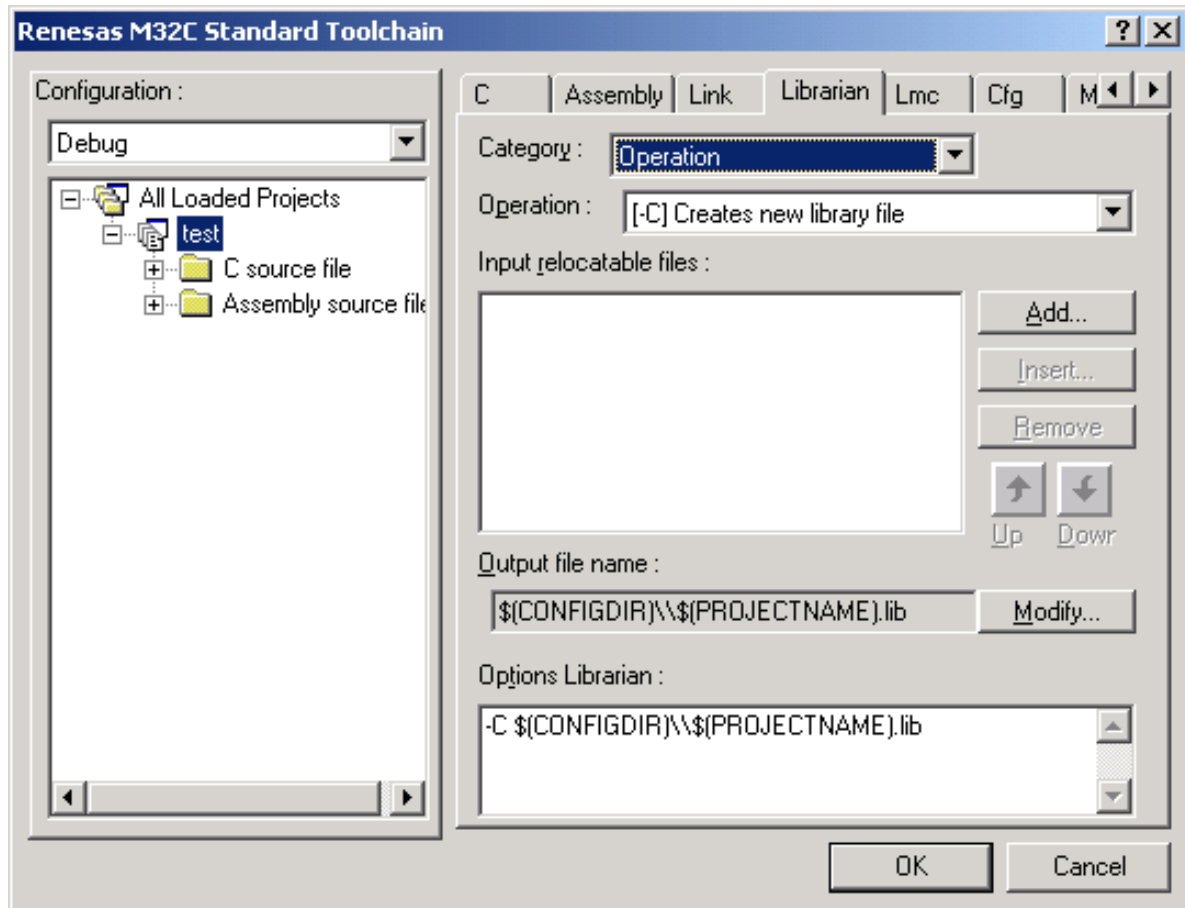


Figure 4.22 [Librarian] Tab Dialog Box

– Category:[Operation]

Table 4.19 Correspondence between Items on the Category:[Operation] Dialog Box and Librarian Options

Dialog Box	Option
Operation	
[-A] Adds modules to library file	A
[-C] Creates new library file	C
[-D] Deletes modules from library file	D
[-L] Generates library list file	L
[-R] Replaces modules	R
[-U] Updates modules	U
[-X] Extract	X

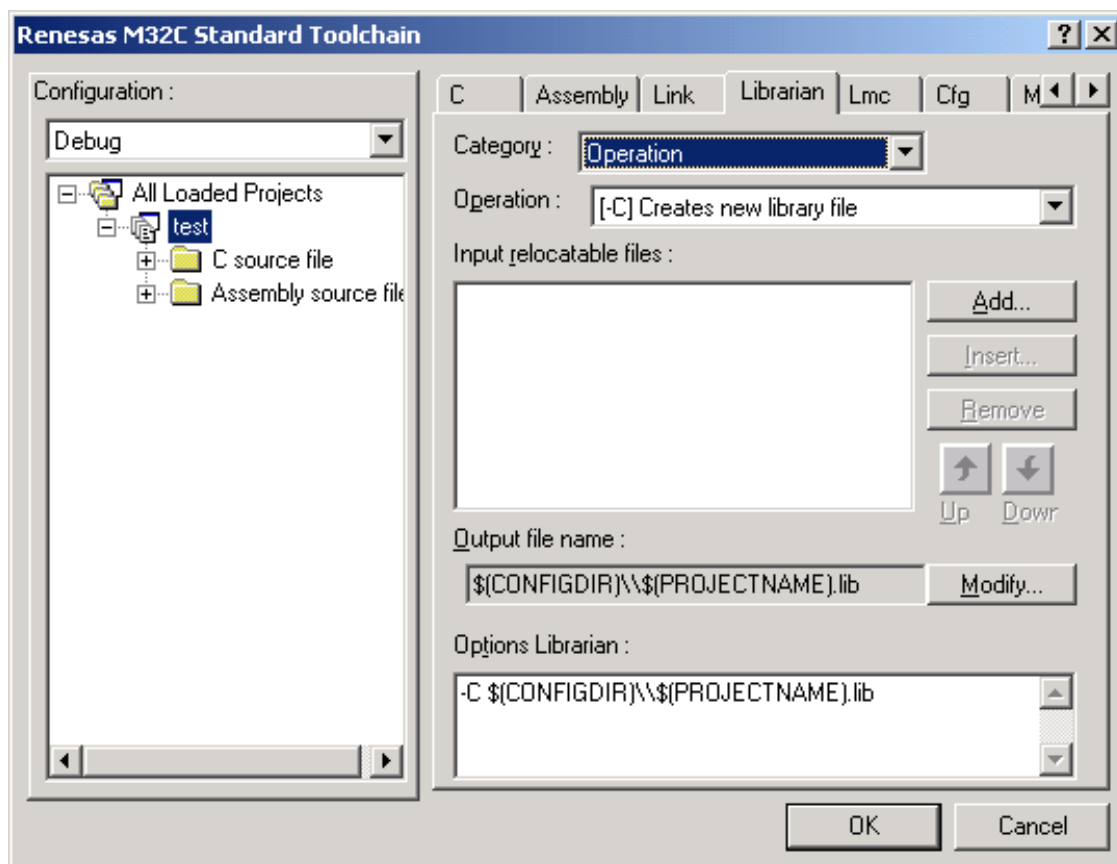


Figure 4.23 Category:[Operation] Dialog Box

– Category:[Other]

Table 4.20 Correspondence between Items on the Category:[Other] Dialog Box and Librarian Options

Dialog Box	Option
Miscellaneous options :	
[-.] Disables message output to screen	.
[-V] Indicates version of librarian	V
User defined options :	

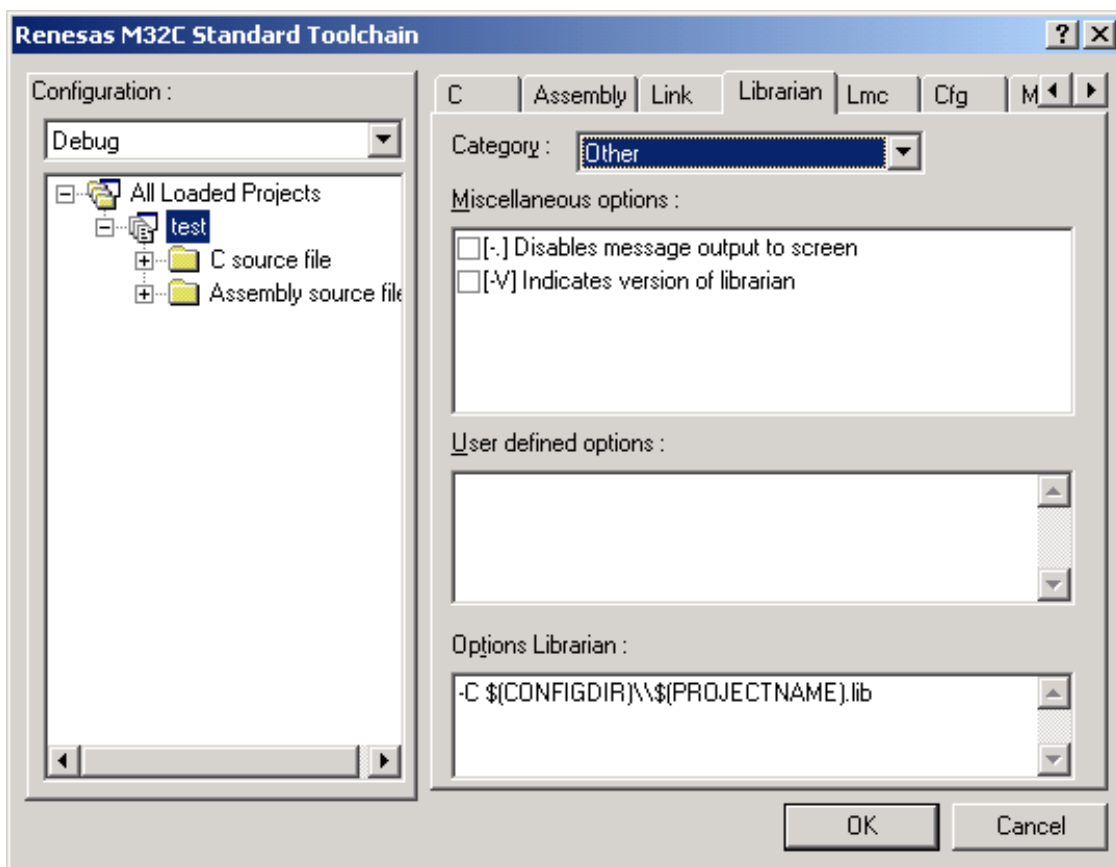


Figure 4.24 Category:[Other] Dialog Box

– Category:[Subcommand file]

Table 4.21 Correspondence between the Item on the Category:[Subcommand file] Dialog Box and Librarian Option

Dialog Box	Option
[@] Use external subcommand file.	@ <file name>

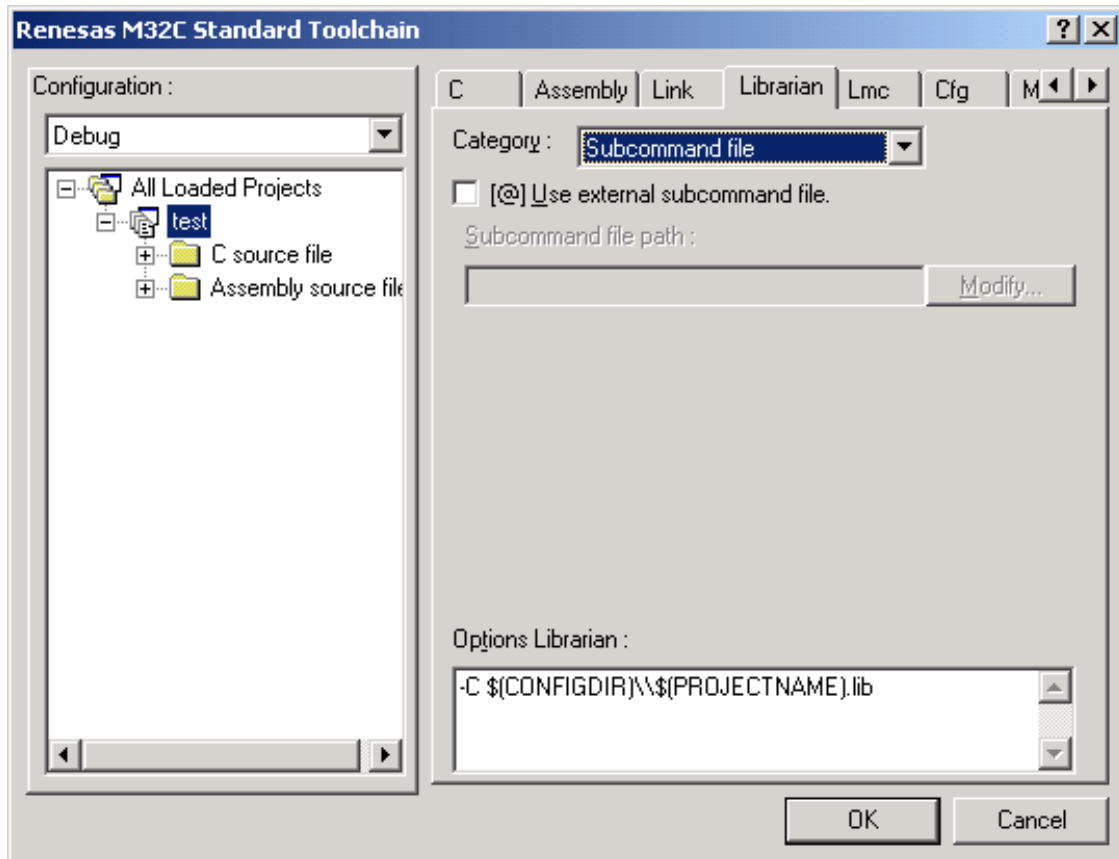


Figure 4.25 Category:[Subcommand file] Dialog Box

4.1.5 Load Module Converter Options

Select the [Lmc] tab in the [Renesas M32C Standard Toolchain] dialog box.

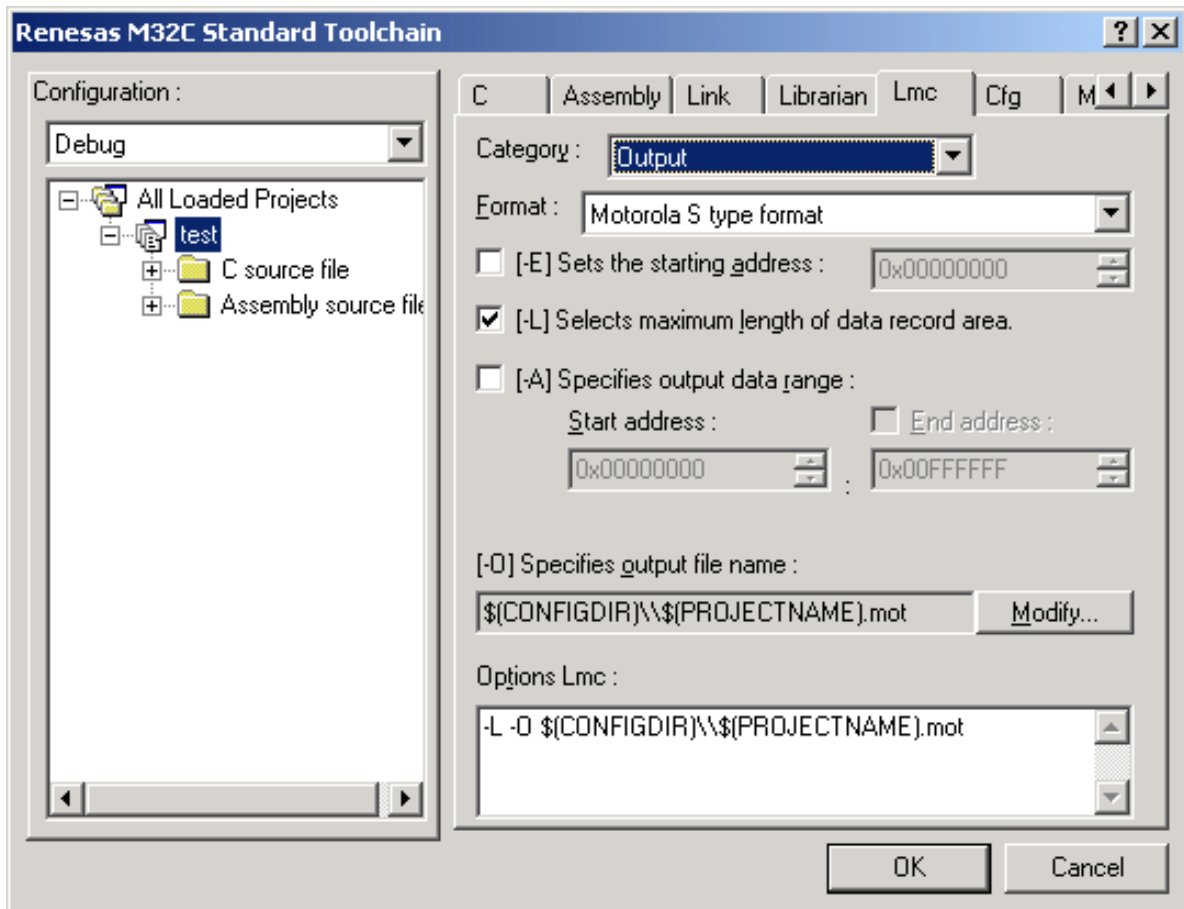


Figure 4.26 [Lmc] Tab Dialog Box

- Category:[Output]

Table 4.22 Correspondence between Items on the Category:[Output] Dialog Box and Load Module Converter Options

Dialog Box	Option
Format :	
Motorola S type format	-
[-H] Converts file info Intel HEX format	H
[-E] Sets the starting address :	EΔ<address>
[-L] Selects maximum length of data record area.	L
[-A] Specifies output data range :	AΔ<start address>[:end address]
[-O] Specifies output file name :	OΔ<file name>

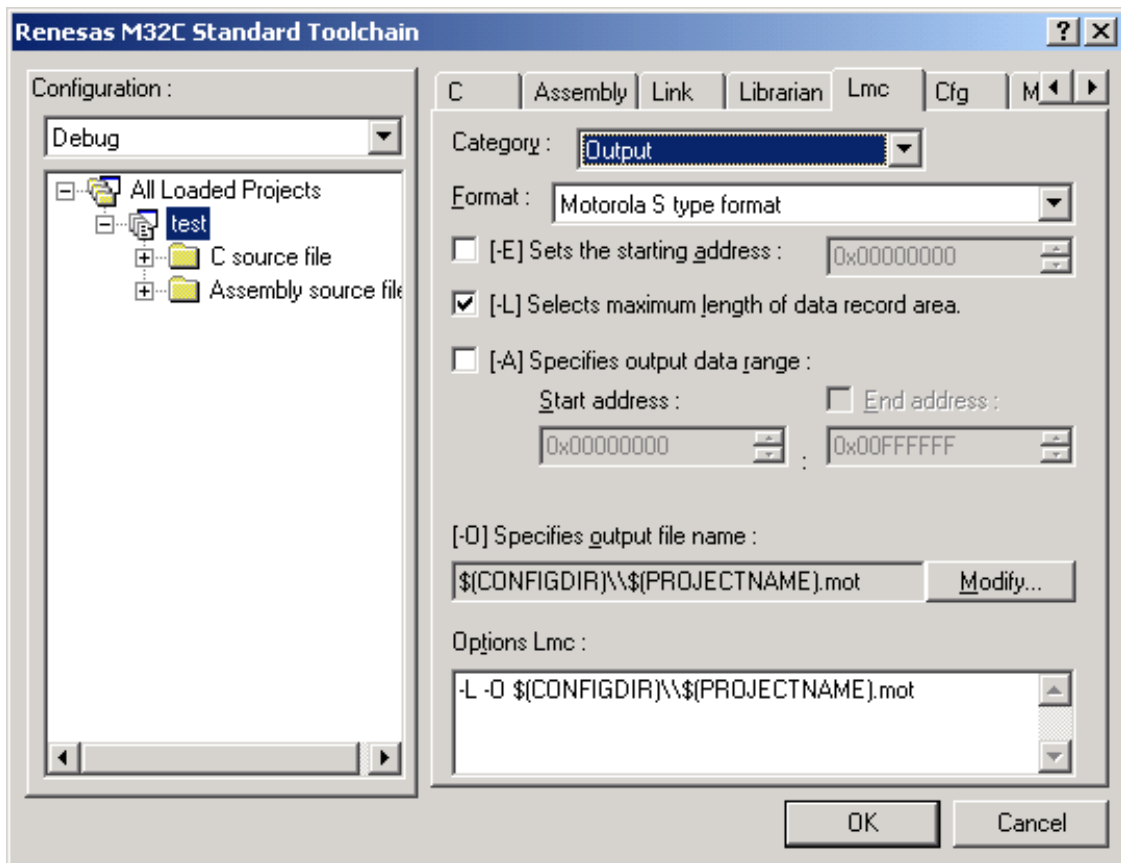


Figure 4.27 Category:[Output] Dialog Box

- Category:[Code]

Table 4.23 Correspondence between Items on the Category:[Code] Dialog Box and Load Module Converter Options

Dialog Box	Option
[-ID] ID code check ID code setting :	ID[sub] [sub] : <code protect setting> <#numeric value>
ROM code protect function : [-protect1] Level 1 setting [-protect2] Level 2 setting [-protectx] Set the ROM code protect value	protect1 protect2 protectxΔ<numeric value>
[-F] Sets the fill data in the free area :	FΔ<data value set in the free area>[sub] [sub] : <:start address>[:end address]

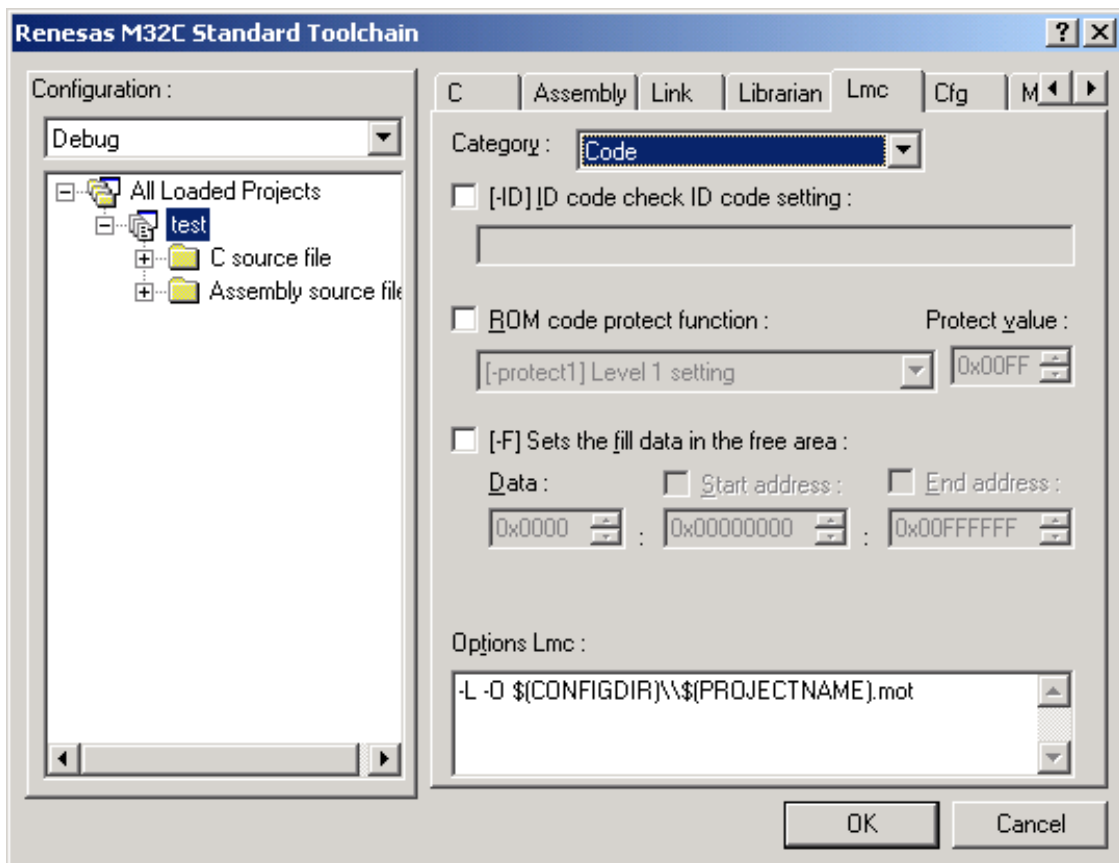


Figure 4.28 Category:[Code] Dialog Box

– Category:[Other]

Table 4.24 Correspondence between Items on the Category:[Other] Dialog Box and Load Module Converter Options

Dialog Box	Option
Miscellaneous options :	
[-.] Disables message output to screen	.
[-V] Indicates version of load module converter	V
User defined options :	

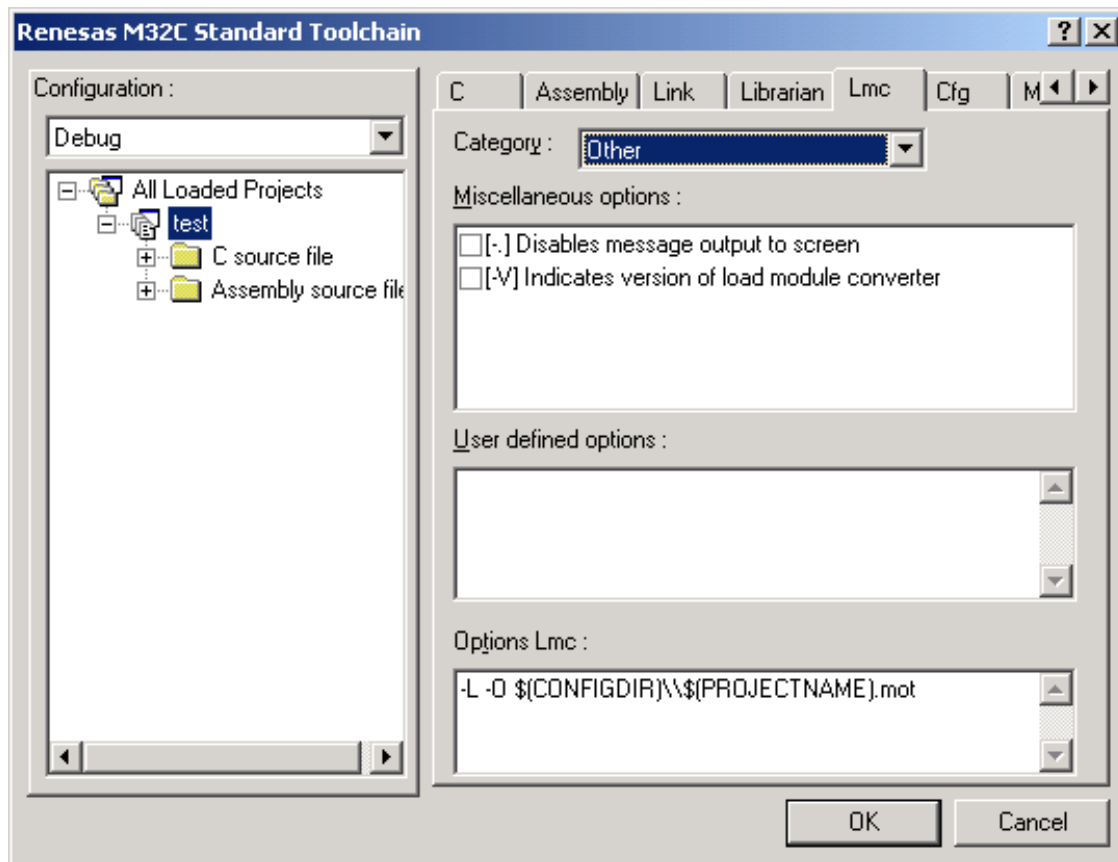


Figure 4.29 Category:[Other] Dialog Box

4.1.6 Configuration Options

Select the [Cfg] tab in the [Renesas M32C Standard Toolchain] dialog box.

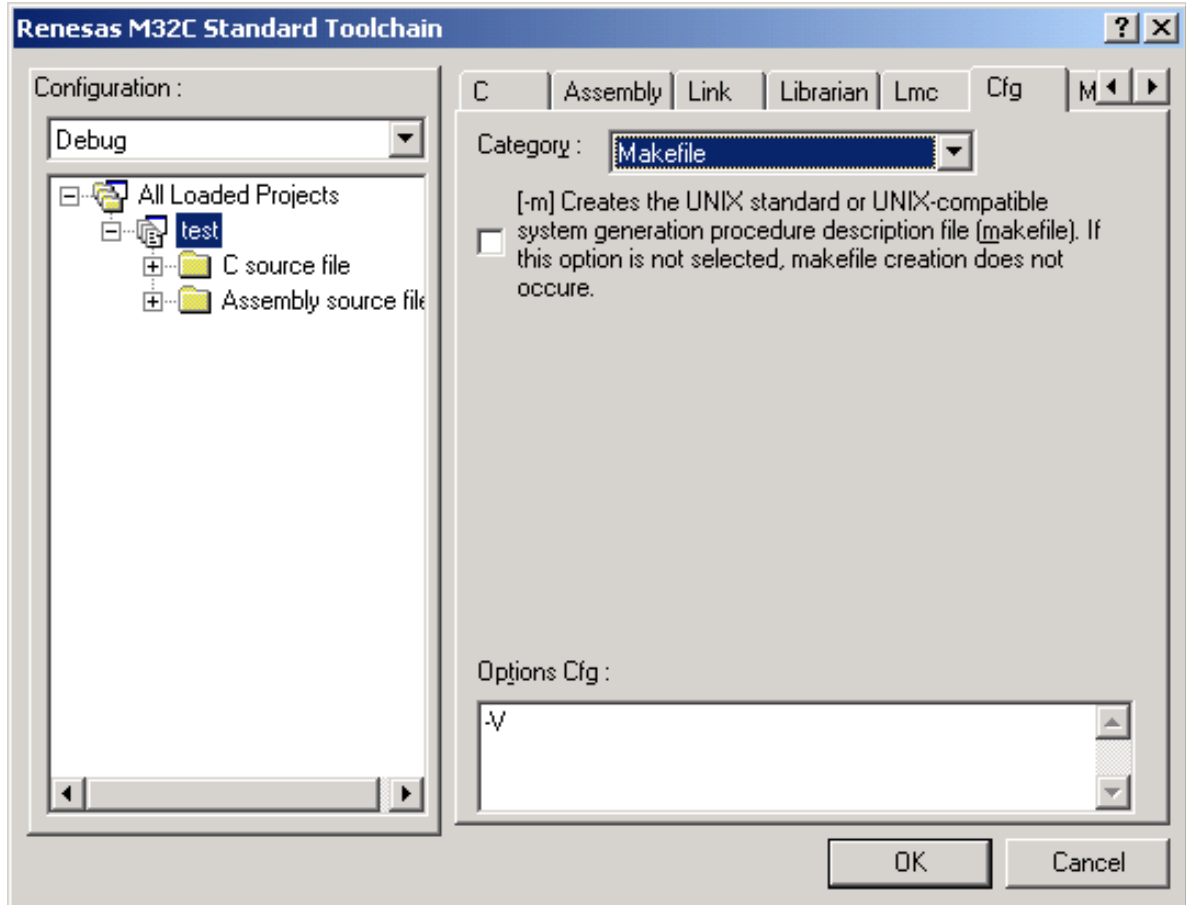


Figure 4.30 [Cfg] Tab Dialog Box

- Category:[Makefile]

Table 4.25 Correspondence between the Item on the Category:[Makefile] Dialog Box and Configurator Option

Dialog Box	Option
[-m] Creates the UNIX standard or UNIX-compatible system generation procedure description file (makefile). If this option is not selected, makefile creation does not occur.	m

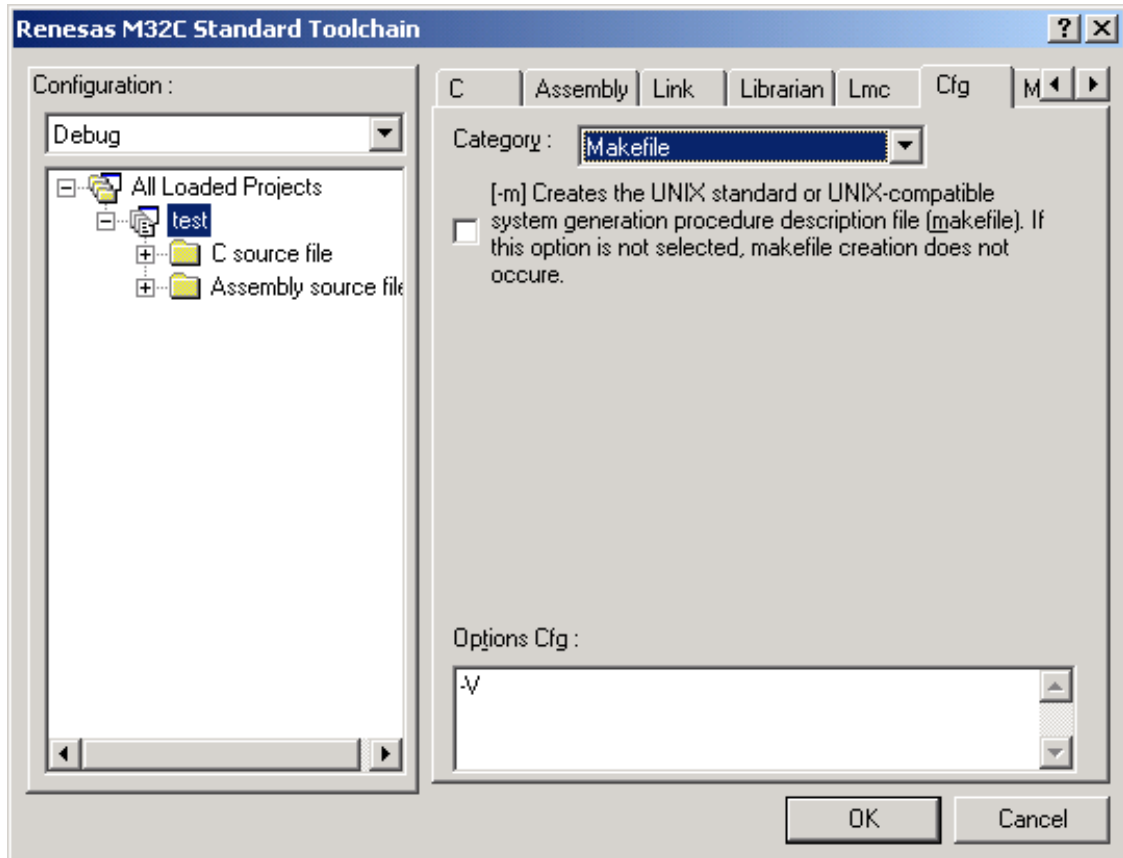


Figure 4.31 Category:[Makefile] Dialog Box

– Category:[Other]

Table 4.26 Correspondence between Items on the Category:[Other] Dialog Box and Configurator Options

Dialog Box	Option
Miscellaneous options :	
[-V] Displays the information on the files generated by the command	V
[-v] Displays the command option descriptions and detailed information on the version	v
User defined options :	

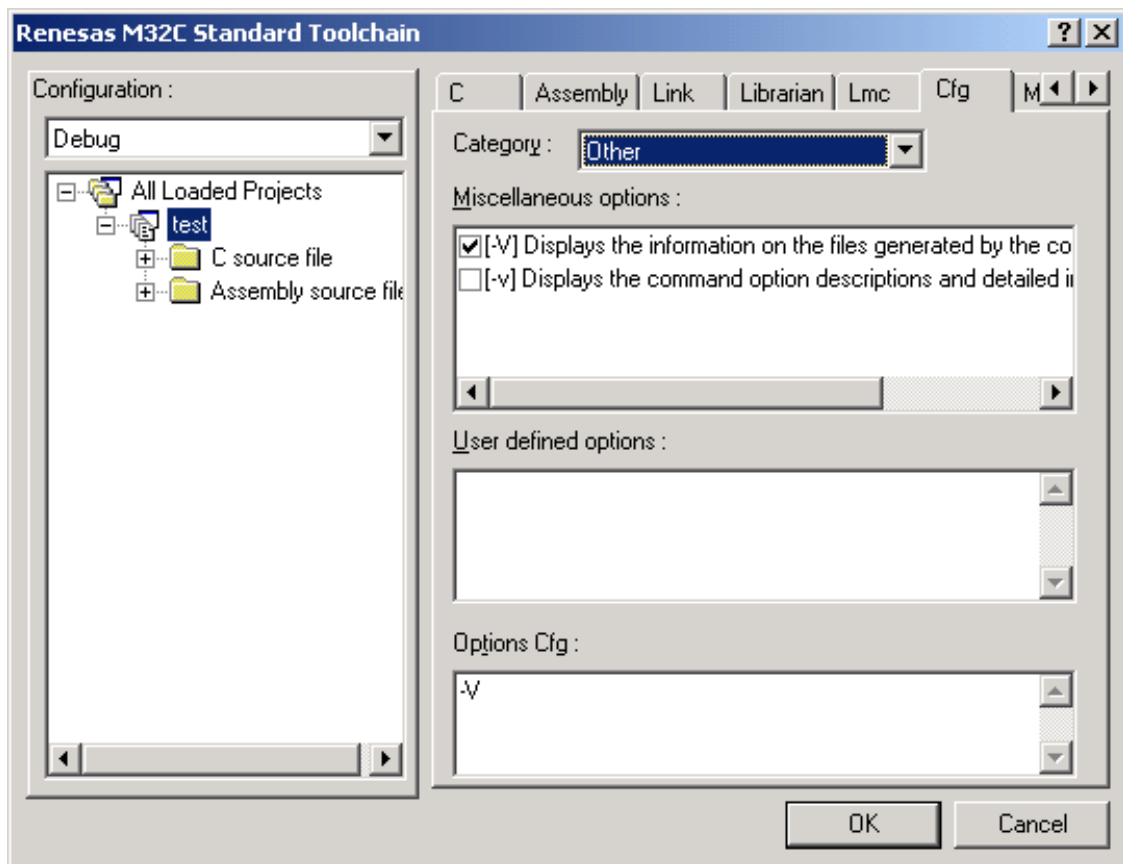


Figure 4.32 Category:[Other] Dialog Box

4.1.7 CPU Options

Select the [CPU] tab in the [Renesas M32C Standard Toolchain] dialog box.

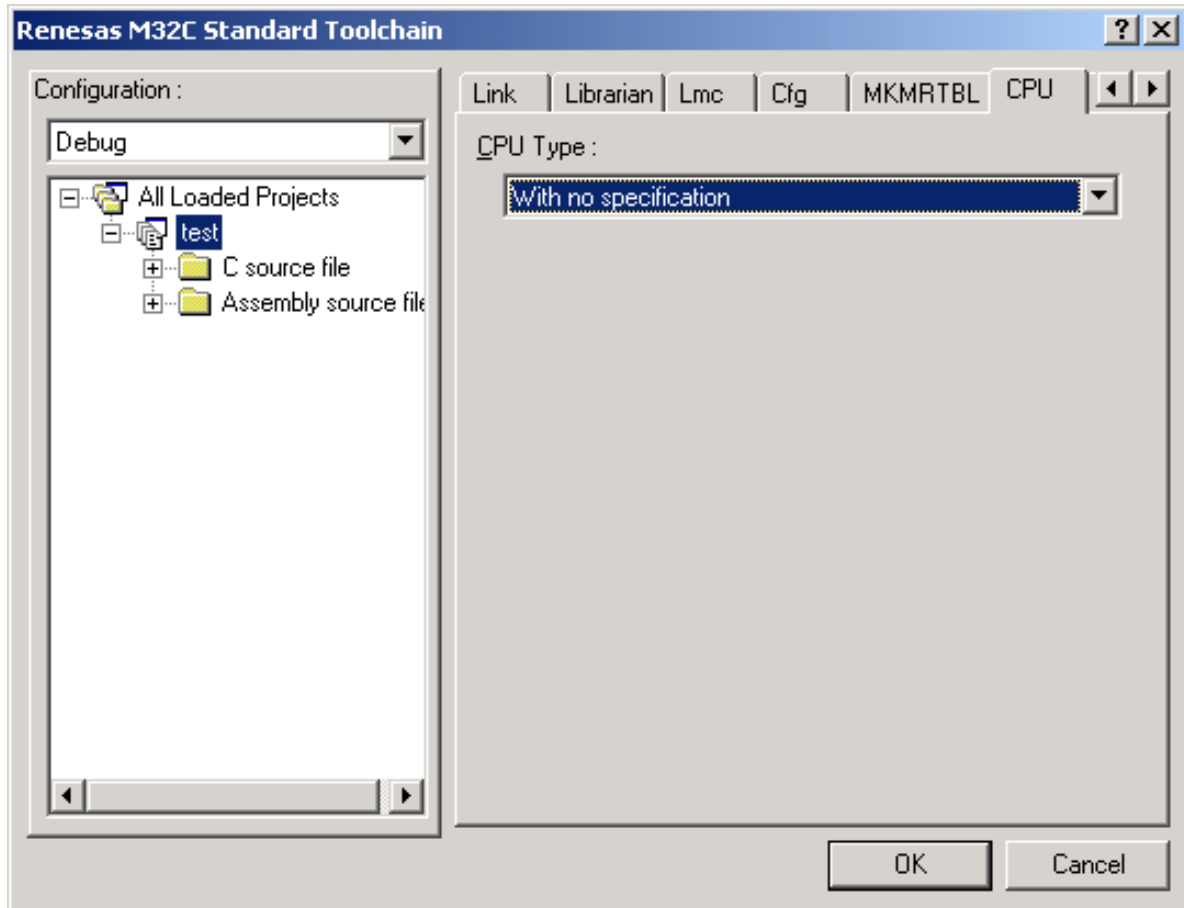


Figure 4.33 [CPU] Tab Dialog Box

Table 4.27 Correspondence between Items on the [CPU] Tab Dialog Box and Compiler Options

Dialog Box	Option
CPU Type :	
With no specification	-
Generates code for M32C/80 series	M82

4.2 Builds

4.2.1 Makefile Output

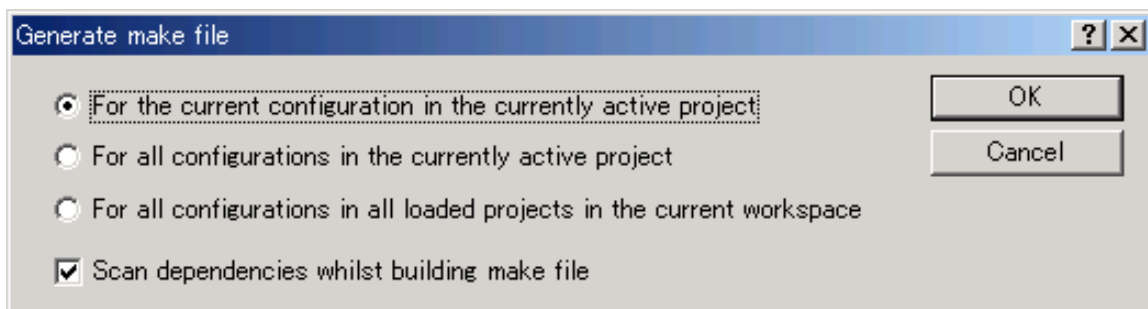
■ Description:

The High-performance Embedded Workshop 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 the High-performance Embedded Workshop completely. This is convenient when you want to send a project to a person who has not installed the High-performance Embedded Workshop or manage the version of an entire build, including the makefile.

■ How to generate a makefile:

- Make sure that the project that generates the makefile is the current project.
- Make sure that the build configuration that builds the project is the current configuration.
- Choose [Build > Generate Makefile].
- The following dialog box appears. In this dialog box, select one of the makefile generation methods.



■ Makefile generation directory:

The High-performance Embedded Workshop creates a "make" subdirectory in the current workspace directory and generates makefiles in this subdirectory. The name of a makefile is the current project or configuration name followed by the extension .mak (debug.mak, for example). The High-performance Embedded Workshop-generated makefiles can be executed by using the executable file HMAKE.EXE contained in the directory where the High-performance Embedded Workshop is installed (c::\¥hew3, for example). However, user-modified makefiles cannot be executed.

■ How to execute a makefile:

1. Open the [Command] window and move to the "make" directory that contains the generated makefile.
2. Execute HMAKE. On the command line, enter HMAKE.EXE <makefile-name>.

4.2.2 Makefile Input

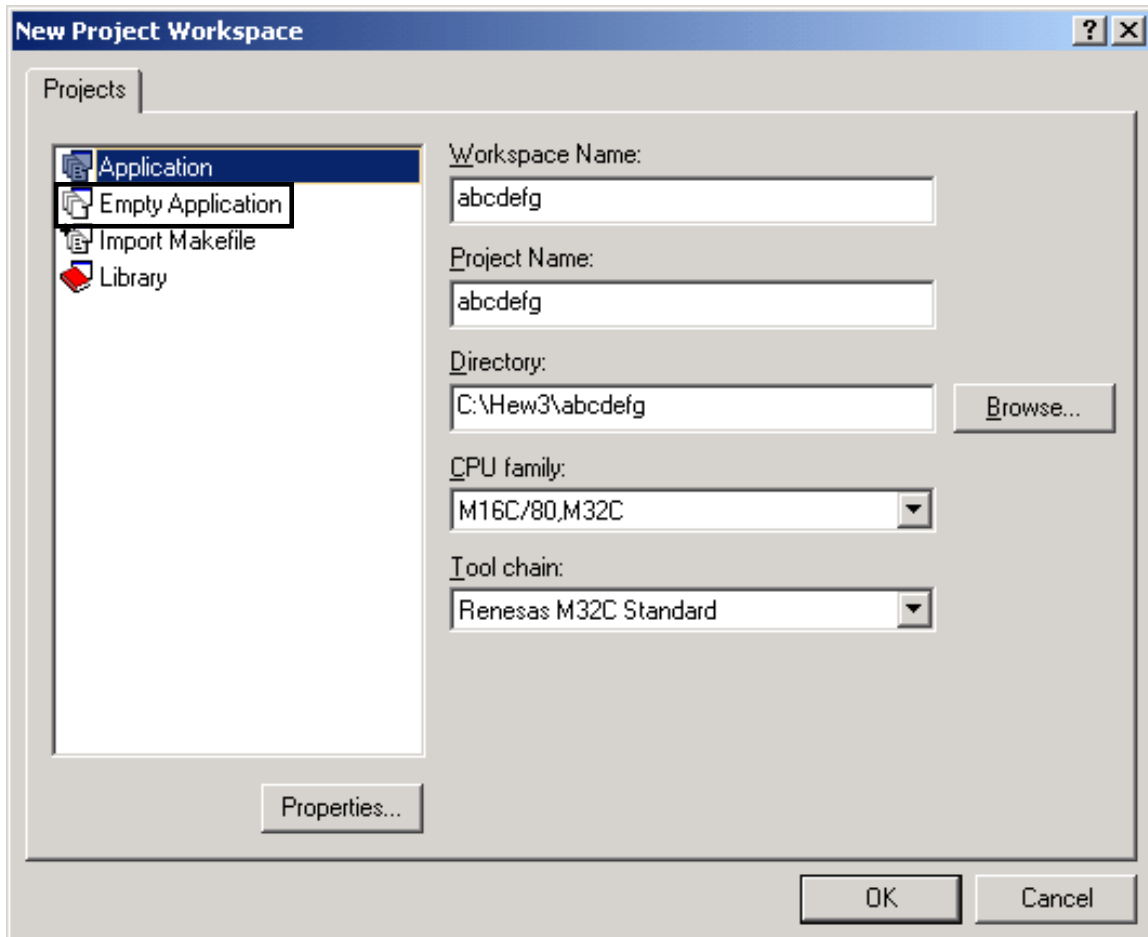
■ Description:

The High-performance Embedded Workshop allows you to input the makefiles that were generated by the High-performance Embedded Workshop or used in the UNIX environment.

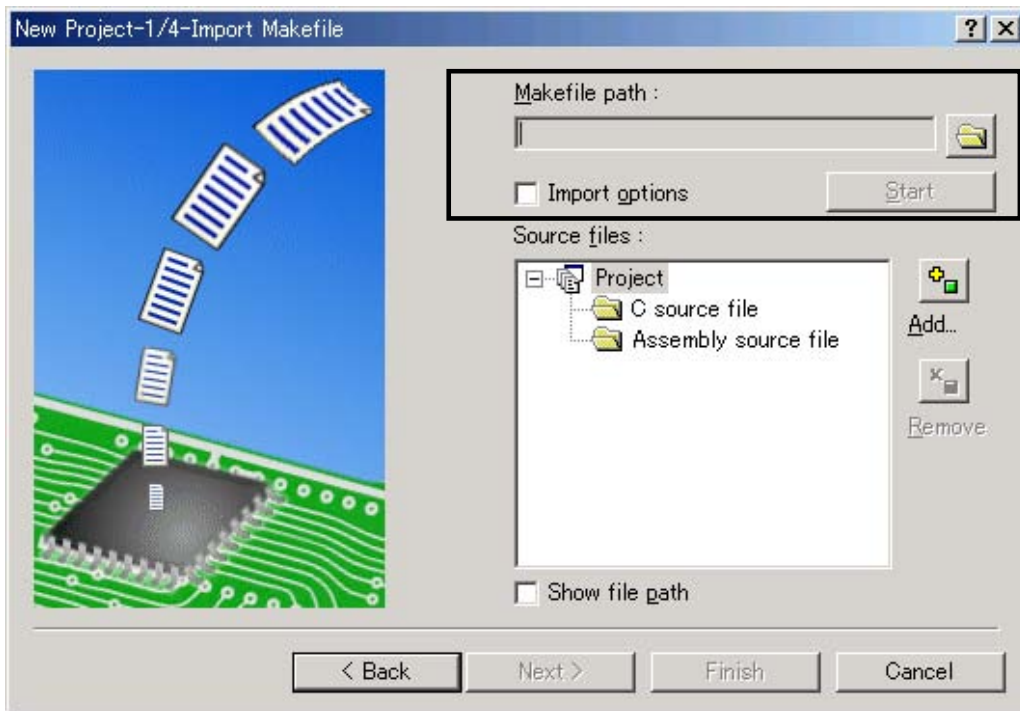
From the makefile, you can automatically obtain the **file structure** of the project. (However, you cannot obtain option settings or similar specifications.) This facilitates the migration from the command line to the High-performance Embedded Workshop.

■ How to input the makefile:

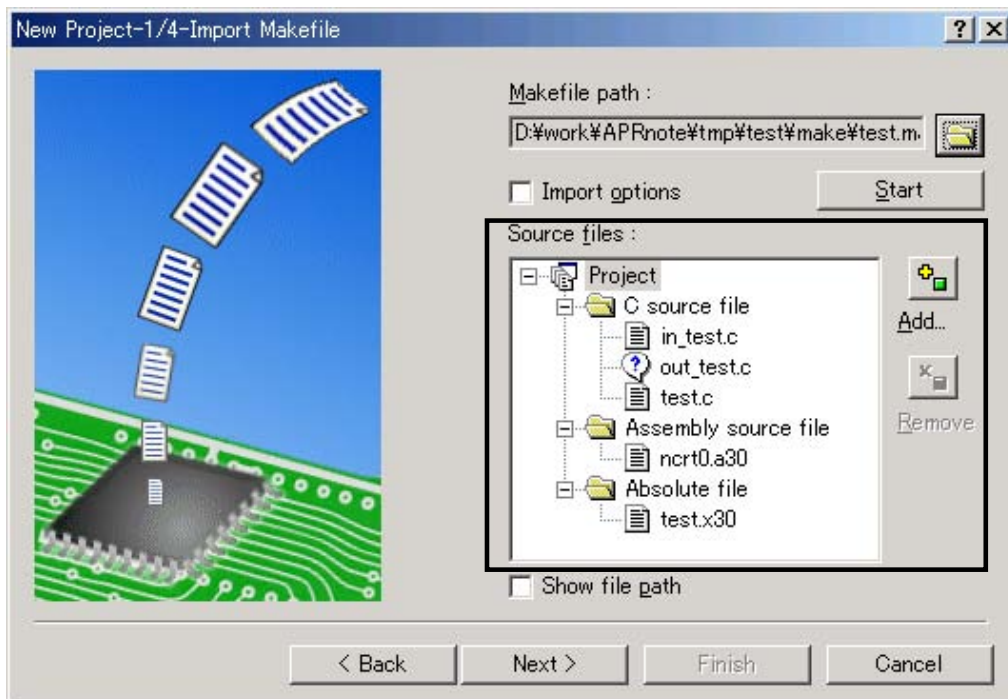
1. When creating a new workspace, select [Import Makefile] from the project type options in the [New Project Workspace] dialog box.



2. In the [New Project-Import Makefile] dialog box, specify the makefile path in the [Makefile path] field, and then click the [Start] button.



3. The [Source files] pane shows the source file structure of the makefile. In this structure chart, any file marked with a question mark icon has been analyzed to make sure it does not contain any entity. This file will not be added to the project. (It is ignored.)



4. By following the wizard, specify the CPU and other options and open the workspace. You can then begin a development work.

4.2.3 Creating Custom Project Types

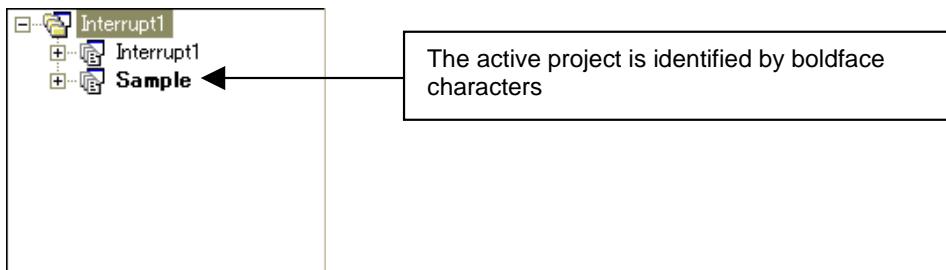
■ Description:

This feature allows a user to use a project created by another user, as a template for developing a program on another machine.

The template may contain all information about the project, including the project file structure, build options, and debugger settings.

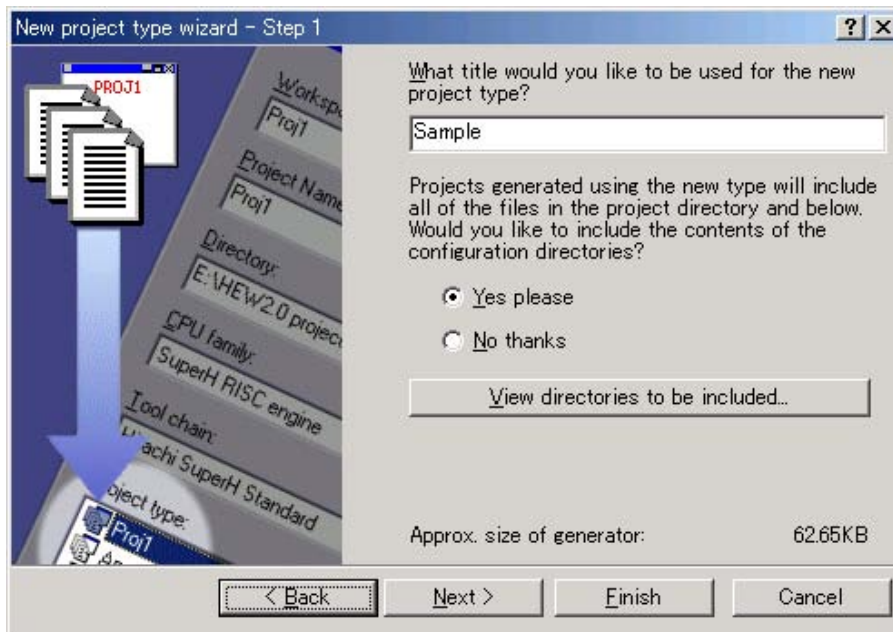
■ How to save the project type:

1. Activate the target project. This is because information about the project that is active when the workspace is open will be saved. To activate a project, choose [Project -> Set Current Project] and select the project.



2. Choose [Project -> Create Project Type]. The following project type wizard appears. Enter the name of 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.

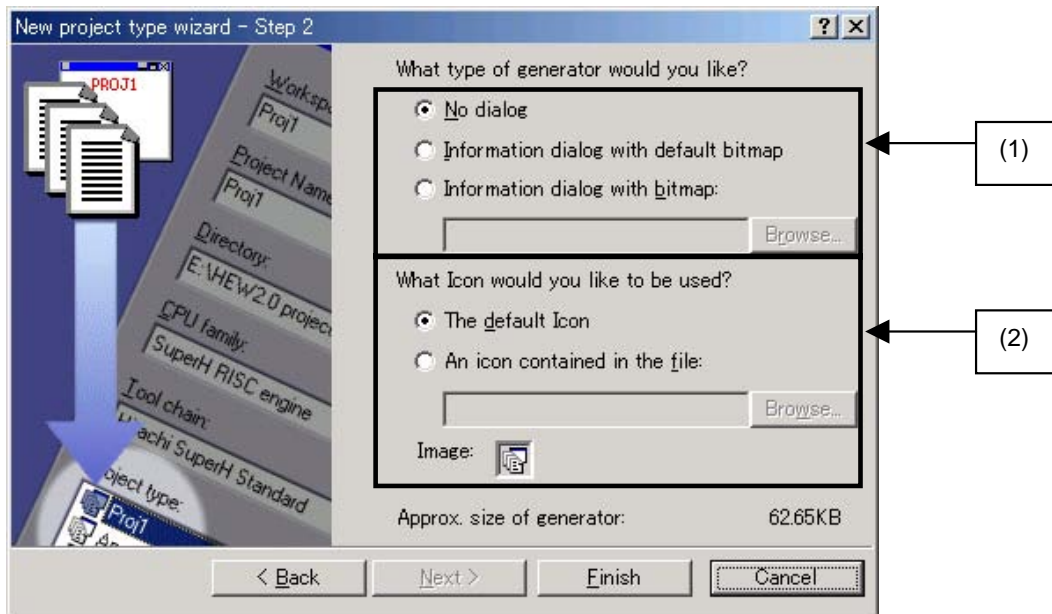
Here, you can click the [Finish] button to quit the project type wizard.



3. In the [New project type wizard – Step 1] dialog box, click the [Next >] button. The following wizard appears. In (1) below, specify whether to display project information and bitmaps when the project type template opens.

In (2), you can change the project type icon to a user-specified icon. Click the [Finish] button.

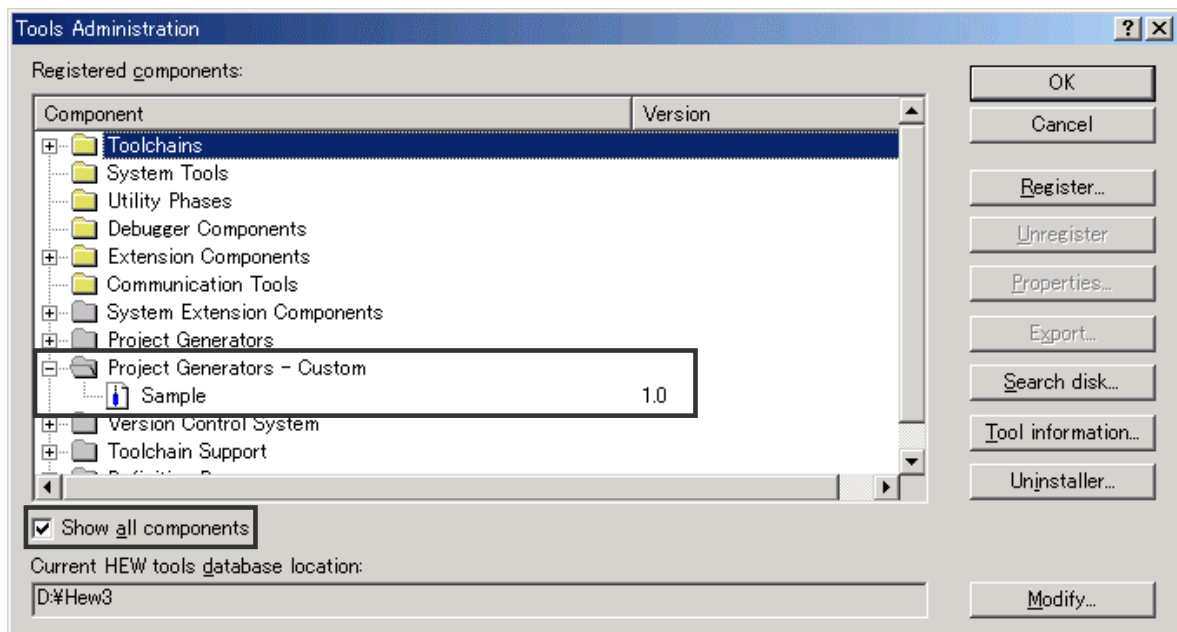
These settings are not mandatory.



4. The above actions create a project type template named "Custom Project Generator". To use this template on another machine, choose [Tools -> Administration]. The following dialog box appears.

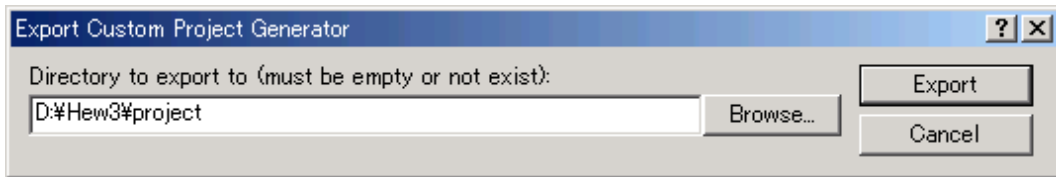
Select the [Show all components] check box to display the [Project Generators – Custom] folder.

In this folder, click the created project type and click the [Export...] button.



The following dialog box appears. Select a directory to which you want to output the Custom Project Generator template. The directory must be empty.

This completes the saving of the project type.

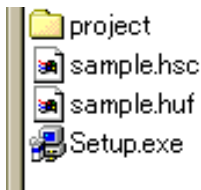


■ Installing Custom Project Generator:

The following shows how to install the Custom Project Generator template created above on another machine.

1. The following installation environment is created in the directory that was created at step 5 in *How to save the project type*:

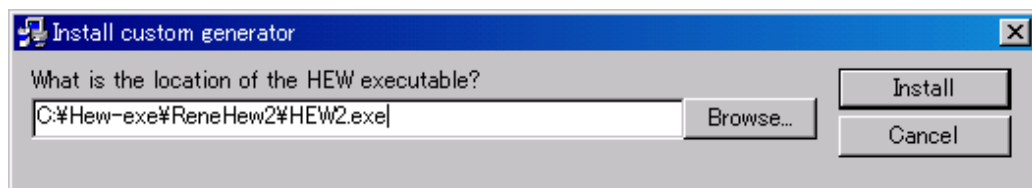
(Installation environment directory)



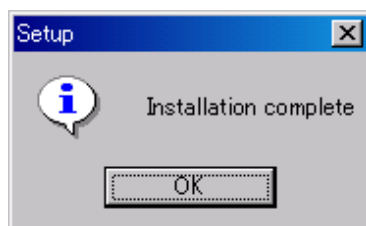
2. Copy the above installation environment and install the copy on another machine.

When you run Setup.exe, the following dialog box appears. Specify the location in which the High-performance Embedded Workshop is installed, and then click the [Install] button.

(Directory example: C:\Hew-exe\ReneHew2\HEW2.exe)



3. This completes the building of the environment.

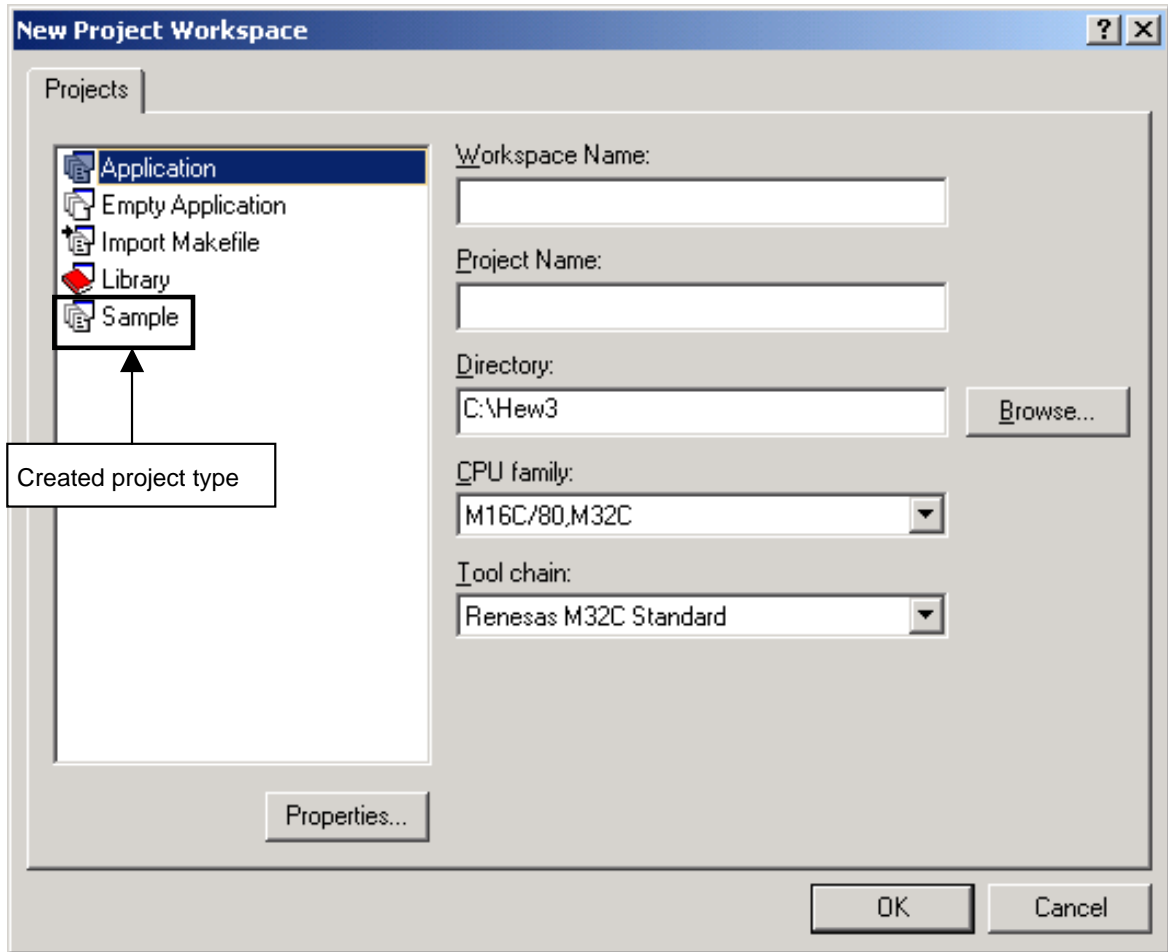


■ Example of using the Custom Project Generator:

The following gives an example of using the installed Custom Project Generator template.

1. Start the High-performance Embedded Workshop and choose [Create a new project workspace] in the [Welcome!] dialog box. The installed project type is added to the [Projects] list. Click the project type and click the [OK] button.

You can now proceed with program development using the stored project template for any new project.



4.2.4 Multi-CPU Feature

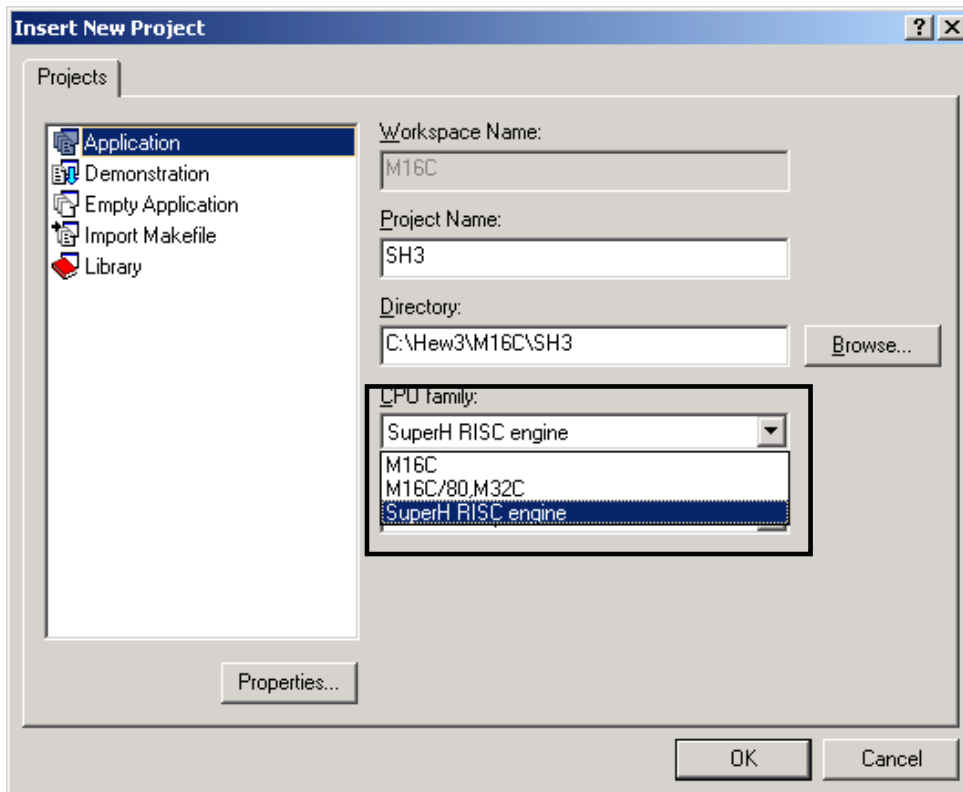
■ Description:

When inserting a new project in the workspace, you can insert a CPU of another type.

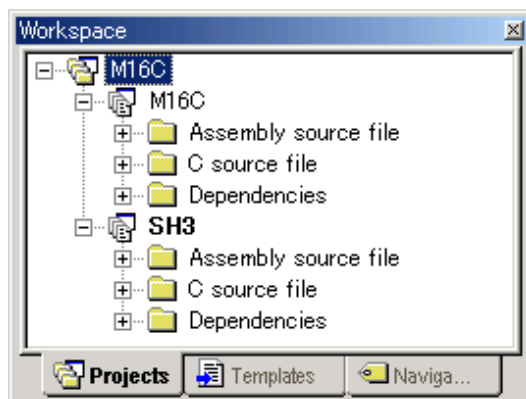
This enables you to manage different projects, such as M16C and SH, in a single workspace.

■ Example of inserting a different CPU family:

1. When an M16C (SH) project is open, click [Project -> Insert Project...]. In the [Insert Project] dialog box, select a new project and click the [OK] button.
2. The [Insert New Project] dialog box appears. Select M16C (SH) for the project name and CPU family, and click the [OK] button. You can place different CPU types in addition to the current CPU types in the workspace.



3. With the procedure above, you can install M16C and SH projects in a single workspace.



4.2.5 Networking Feature

■ Description:

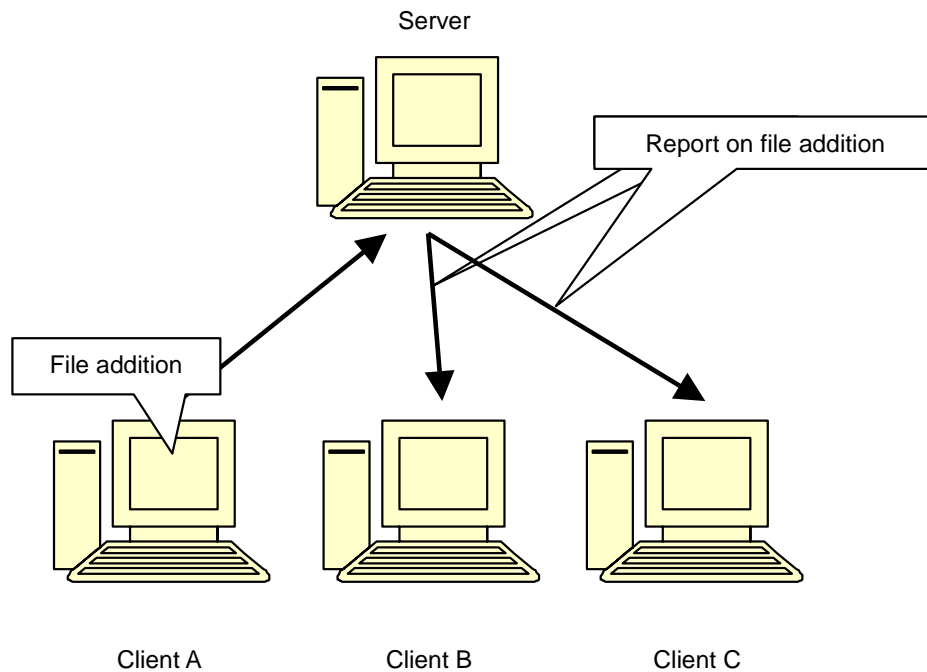
The High-performance Embedded Workshop allows different users to share workspaces and projects via a network.

This allows the users to 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. Then the server notifies the other clients of the addition.

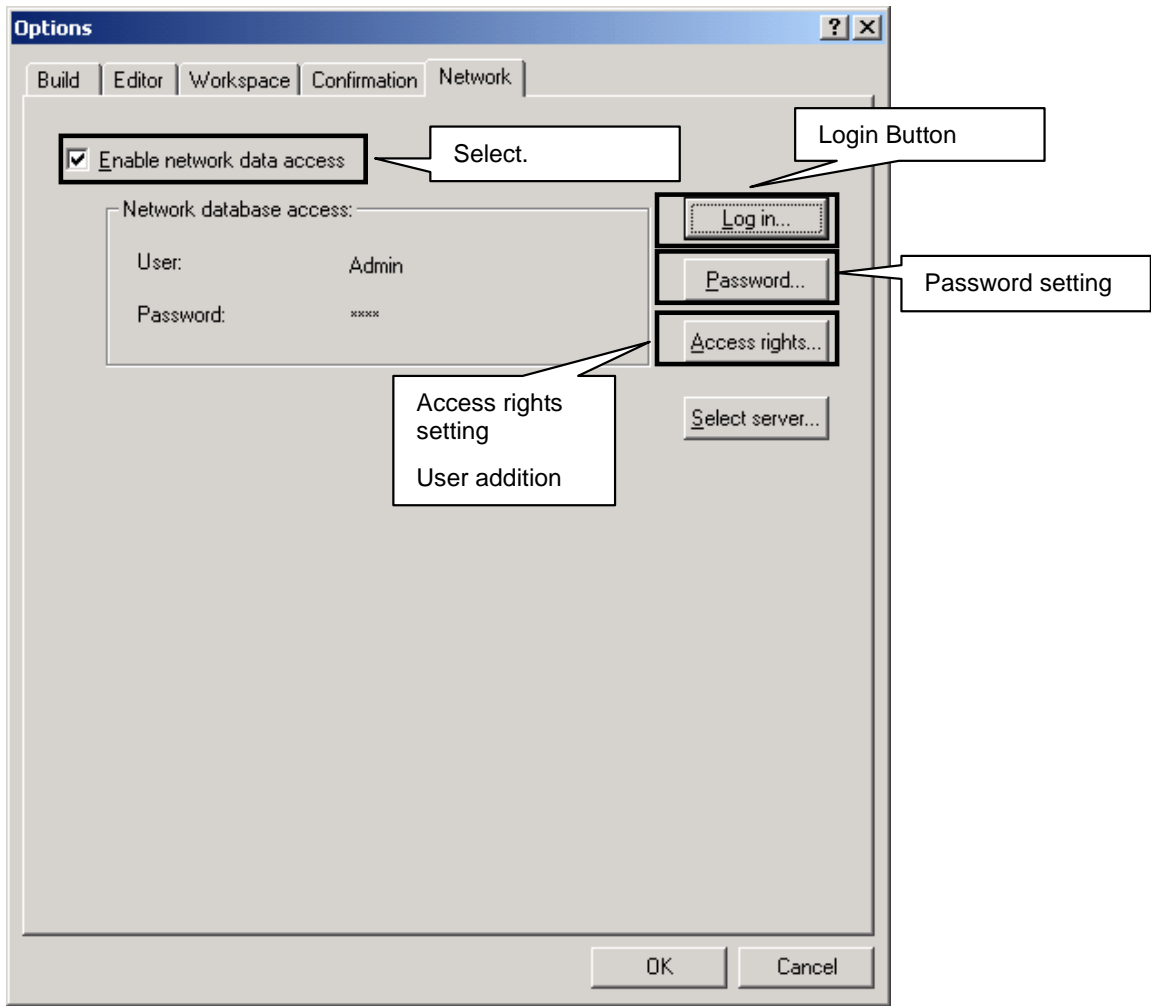
In addition, users can be granted rights for access to specific projects or files.



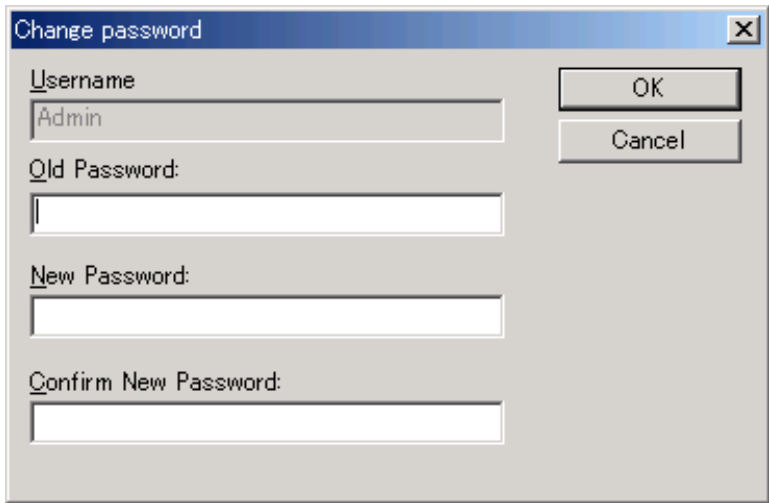
■ Setting the network access:

1. Choose [Tools -> Options], and then select the [Network] tab. Select 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 the [Password] button and specify a password for the administrator.
4. Click the [OK] button. This allows the administrator to access the network.

[Network] Tab of the [Options] dialog box



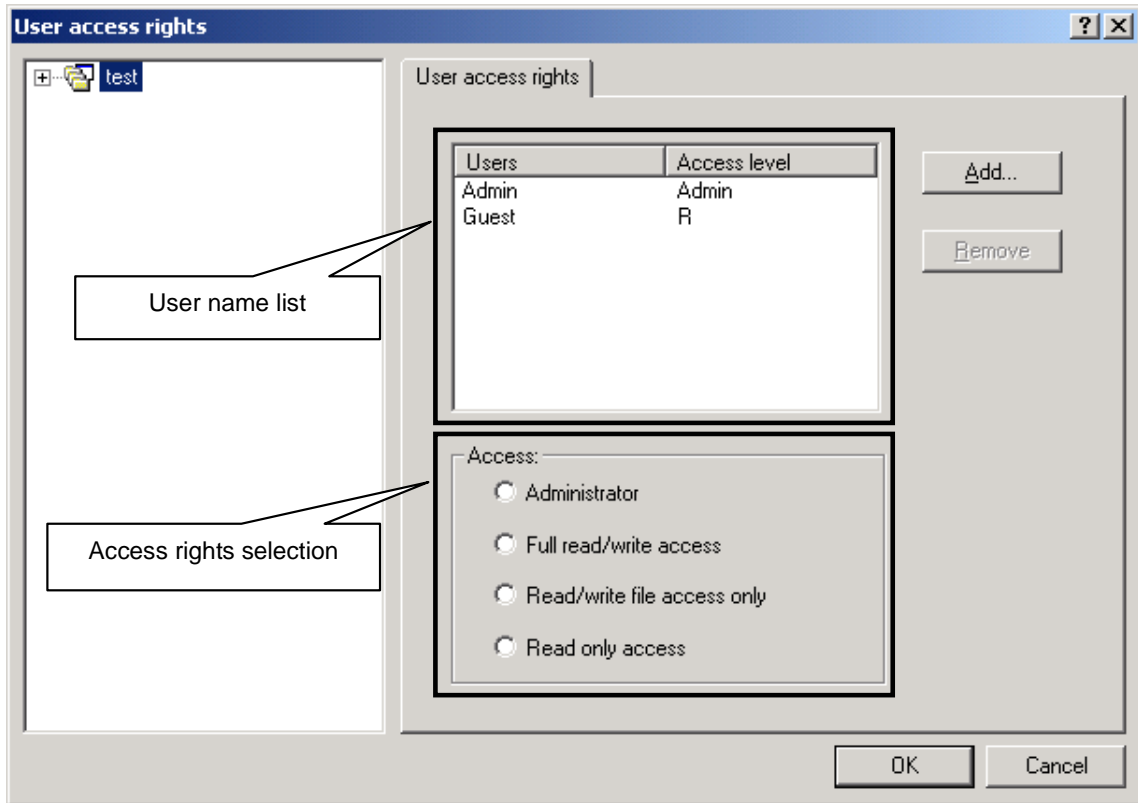
[Change password] dialog box



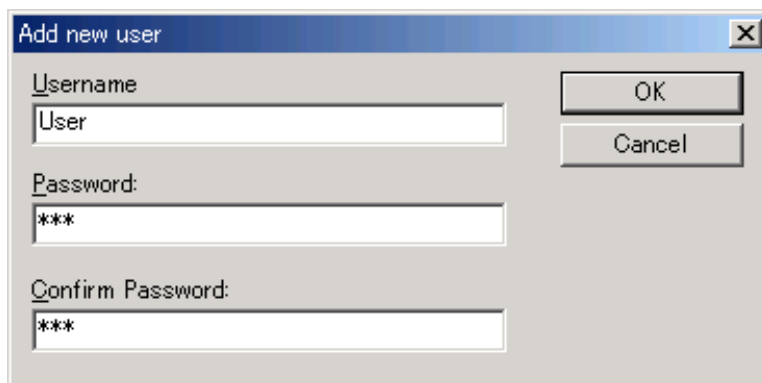
■ Adding a new user:

By default, an administrator and a guest have been added. You can register new users.

1. Click the [Log in...] button shown on the previous page. Log in as a user who has administrator access right.
2. Click the [Access rights] button. The [User access rights] dialog box appears.



3. Click the [Add...] button. The [Add new user] dialog box appears.
4. Enter a new user name and password. (Password specification is mandatory.)



5. The new user name is then added to the user list. Select the user name and specify access right for the user.

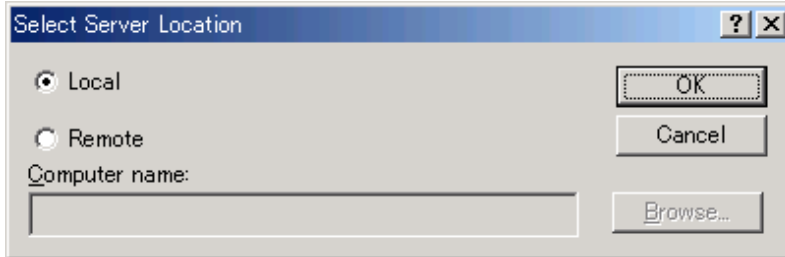
Click the [OK] button to enable your specification.

■ Selecting the server machine:

Select the machine that will work as the server. If you want to make your own machine the server, you do not have to do anything.

If you want to specify another machine as the server, click the [Select server] button in the [Network] tab. Select [Remote] in the following dialog box, and then specify the computer name.

Click the [OK] button to enable your specification.



■ **Remarks:**

Use of this feature will lower the High-performance Embedded Workshop performance.

MEMO

Section 5. Efficient Programming Techniques

While the NC308WA compiler performs its own optimization, resourceful programming can also yield increased performance. This chapter describes several techniques that users can employ to create more efficient programs. A program can be evaluated using two criteria: how fast it executes, and how small it is. The following are important principles for creating efficient programs:

(1) Maximizing execution speed

Execution speed is determined by both frequently executed statements and complex statements. It is important to understand how these statements are processed, and to improve them selectively.

(2) Minimizing program size

To keep the program as small as possible, similar processing sections should be shared, and complex functions should be simplified whenever possible.

Due to optimization by the compiler, results regarding execution speed may differ from what they would theoretically be. As such, make use of various methods to improve performance, testing them on the compiler as you go.

Table 5.1 lists the efficient programming techniques explained in this chapter.

Table 5.1 List of Efficient Programming Techniques

No.	Item		RAM efficiency	ROM efficiency	Execution speed
5.1	Register passing for arguments		✓	✓	✓
5.2	Using register variables		✓	✓	✓
5.3	Using M16C-specific instructions		--	✓	✓
5.4	Using the "carry" flag for bit operation branching		--	✓	✓
5.5	Moving determinate items within a loop to outside of the loop		--	--	✓
5.6	SBDDATA declaration and SPECIAL page function declaration utility	SBDDATA declarations	--	✓	--
		SPECIAL page function declarations	--	✓	✗
5.7	Using "switch" instead of "else if"		--	--	✓
5.8	Comparison operators for loop counters		--	✓	✓
5.9	restrict		--	✓	✓
5.10	Using _Bool		--	✓	--
5.11	Explicitly initializing auto variables		✓	✓	✓
5.12	Initializing arrays		--	--	✓
5.13	Increments / decrements		✓	✓	✓
5.14	"switch" statements		--	--	✓
5.15	Immediate floating-points		--	✓	✓
5.16	Zero clearing external variables		--	✓	✓

No.	Item	RAM efficiency	ROM efficiency	Execution speed
5.17	Organizing start-up	--	✓	✓
5.18	Using temporary values within loops	×	--	✓
5.19	Using 32-bit mathematical functions	--	✓	✓
5.20	Using unsigned wherever possible	--	✓	✓
5.21	Array index types	✓	✓	✓
5.22	Using prototype declarations	✓	✓	✓
5.23	Using the char type for functions that return only char type values	--	✓	--
5.24	Commenting out clear processing for bss areas	--	✓	✓
5.25	Reducing generated code	✓	✓	--

5.1 Register Passing for Arguments

There are two ways to pass an argument to a function: *stack passing*, where the argument is passed by being pushed to the stack, and *register passing*, where the argument is passed by being assigned to the register. Whereas stack passing entails the cost of pushing to and popping from the stack, register passing entails so such cost, and can be performed quickly. Register passing is used under the following three conditions:

- ① A prototype declaration exists for the function.
- ② The ... variable argument is not used in the prototype declaration.
- ③ The function argument type matches one listed in the following table.

Figure 5.1 shows a condition under which the existence of a prototype declaration changes whether register passing is used or stack passing is used.

Table 5.2 Types for Register Passing

Compiler	First argument	Second argument
NC30WA	_Bool type char type int type near pointer type	int type near pointer type
NC308WA	_Bool type char type int type near pointer type	None.

Note that allocation to the register is as follows when register passing is used.

Table 5.3 Argument Allocation for Register Passing

Argument type	Compiler	First argument	Second argument	Third and subsequent arguments
_Bool type	NC30WA	R1L	Stack	Stack
char type	NC308WA	R0L		
int type	NC30WA	R1	R2	Stack
near pointer type	NC308WA	R0	Stack	
Other types	NC30WA	Stack	Stack	Stack
	NC308WA			

Before	After
<pre>int main() { f(3); } int f(a) int a; { ... }</pre>	<pre>int f(int a); int main() { f(3); } int f(int a) { ... }</pre>
<pre>push.w #0003H jsr _f</pre>	<pre>mov.w #0003H,R0 jsr \$f</pre>

Figure 5.1 Example of Register Passing for an Argument

5.2 Using Register Variables

You can allocate a frequently used variable to the register by adding the register modifier to the variable declaration, to speed up the program considerably. However, if you use the register modifier too much, register space may become insufficient, which can actually slow down the program. Also, with NC30WA, when a variable remaining from a function call is allocated to the register, register save/restore instructions are created before and after the function call, which can also slow down the program. The "-fER" option is required at compile-time for the register modifier to take effect. Figure 5.4 gives an example of such an improvement.

```

int f()
{
    register int i;
    for(i=0;i<100;i++)
    {
        ...
    }
}

```

The variable i is force placed in the register.

Figure 5.2 Declaring a Register Variable

```

int a;
int f()
{
    register int i;
    i=a;
    g();
    i=a+1;
}

```

Inefficient, as register save/restore instructions are generated for NC30WA.

Figure 5.3 Register Save/Restore

Before	After
<pre> int i; sum=0; for(i=0;i<100;i++) { sum+=a[i]; } </pre>	<pre> register int i; sum=0; for(i=0;i<100;i++) { sum+=a[i]; } </pre>
<pre> ;## # C_SRC : sum=0; mov.w #0000H,-4[FB] ; sum ;## # C_SRC : for(i=0;i<100;i++) mov.w #0000H,-4[FB] ; i L1: ;## # C_SRC : for(i=0;i<100;i++) cmp.w #0064H,-4[FB] ; i jge L5 ;## # C_SRC : sum+=a[i]; mov.w -4[FB],A0 ; i shl.w #1,A0 add.w _a:16[A0],-2[FB] ; sum add.w #0001H,-4[FB] ; i jmp L1 L5: </pre>	<pre> ;## # C_SRC : sum=0; mov.w #0000H,-2[FB] ; sum ;## # C_SRC : for(i=0;i<100;i++) mov.w #0000H,R0 L1: ;## # C_SRC : for(i=0;i<100;i++) cmp.w #0064H,R0 ; i jge L5 ;## # C_SRC : sum+=a[i]; mov.w R0,A0 ; i i shl.w #1,A0 add.w _a:16[A0],-2[FB] ; sum add.w #0001H,R0 ; i jmp L1 L5: </pre>

Figure 5.4 Using Register Variables

5.3 Using M16C-specific Instructions

By replacing code that uses an "if" statement to assign a value to a variable with code that uses both an "if" clause and an "else" clause to assign an immediate value to the same variable, you can reduce branching by expanding the "STZX" instruction, and increase ROM efficiency. This is shown below.

Before	After
<pre>void main(void) { int i=2; int port; if(port == 1){ i = 3; } }</pre>	<pre>void main(void) { int i; int port; if(port == 1){ i = 3; }else{ i = 2; } }</pre>
<pre>mov.w #0002H,R0 ; i cmp.w #0001H,-2[FB] ; port jne L3 mov.w #0003H,R0 ; i L3:</pre>	<pre>cmp.w #0001H,-2[FB] ; port stzx.w #0003H,#0002H,R0 ; i</pre>

Figure 5.5 Using M16C-specific Instructions

5.4 Using the "Carry" Flag for Bit Operation Branching

In code such as the following, & (|) should be used instead of && (||).

Before	After
<pre> struct A { int a:1; int b:1; int c:1; } a; main() { if(a.a && a.b && a.c) func(); } </pre>	<pre> struct A { int a:1; int b:1; int c:1; } a; main() { if(a.a & a.b & a.c) func(); } </pre>
<pre> btst 00H,-2[FB] ; a jz L1 btst 01H,-2[FB] ; a jz L45 btst 02H,-2[FB] ; a jz L47 jsr _func L47: L45: L1: </pre>	<pre> btst 00H,-2[FB] ; a band 01H,-2[FB] ; a band 02H,-2[FB] ; a jnc L29 jsr _func L29: </pre>

Figure 5.6 Using the "Carry" Flag for Bit Operation Branching

5.5 Moving Determinate Expressions Within a Loop to Outside of the Loop

In code such as the following, by moving determinate expressions that are within a loop so that they are outside the loop, you can reduce the number of calculations required, and thus speed up the program. This can be done automatically by enabling the optimization option in the compiler.

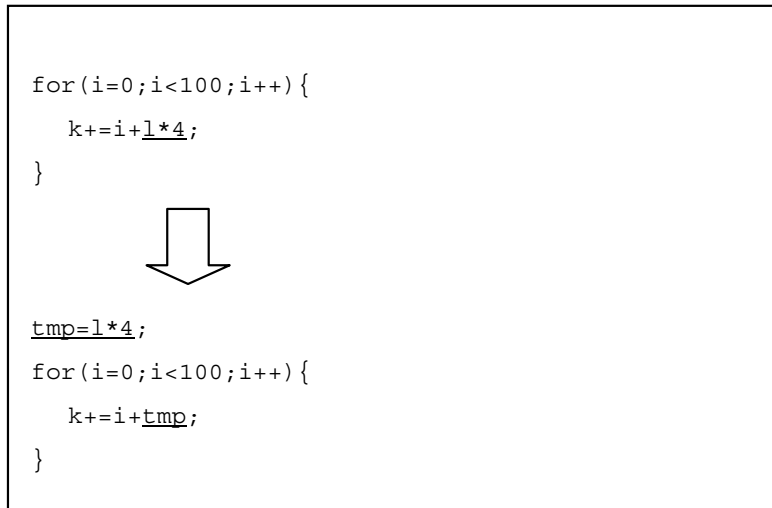


Figure 5.7 Moving Determinate Expressions Within a Loop to Outside of the Loop

C source	Before optimization	After optimization
<pre> for(i=0;i<100;i++) a[i]=1*4; </pre>	<pre> ;## # C_SRC : for(i=0;i<100;i++) mov.w #0000H,_i:16 L1: .line 18 ;## # C_SRC for(i=0;i<100;i++) cmp.w #0064H,_i:16 jge L5 ;## # C_SRC : a[i]=1*4; mov.w _l:16,R0 shl.w #2,R0 indexwd.w _i:16 mov.w R0,_a:16 add.w #0001H,_i:16 jmp L1 L5: </pre>	<pre> ;## # C_SRC :for(i=0;i<100;i++) mov.w #0000H,_i:16 mov.w l:16,R0 shl.w #2,R0 L1: ;## # C_SRC : a[i]=1*4; indexwd.w _i:16 mov.w R0,_a:16 add.w #0001H,_i:16 cmp.w #0064H,_i:16 jlt L1 </pre>

Figure 5.8 Performing Optimization to Move Determinate Items Within a Loop to Outside of the Loop

5.6 SBDATA Declaration and SPECIAL Page Function Declaration Utility

utl308, the SBDATA declaration and SPECIAL page function declaration utility, processes absolute module files (*.x30), and outputs the following:

1. SBDATA declaration

This declaration is for performing allocation from frequently used variables, to the SB area.

(#pragma SBDATA)

2. SPECIAL page function declaration

This declaration is for performing allocation from frequently used functions, to the special page area.

(#pragma SPECIAL)

To use utl308, at compile-time, specify the command line option "-finfo" in the compile driver to generate absolute module files (*.x30).

Figure 5.9 shows the processing flow for NC308. You can use this tool for optimum usage of SBDATA functionality and SPECIAL page functionality. For details, in the NC308 User's Manual, see *Appendix G*.

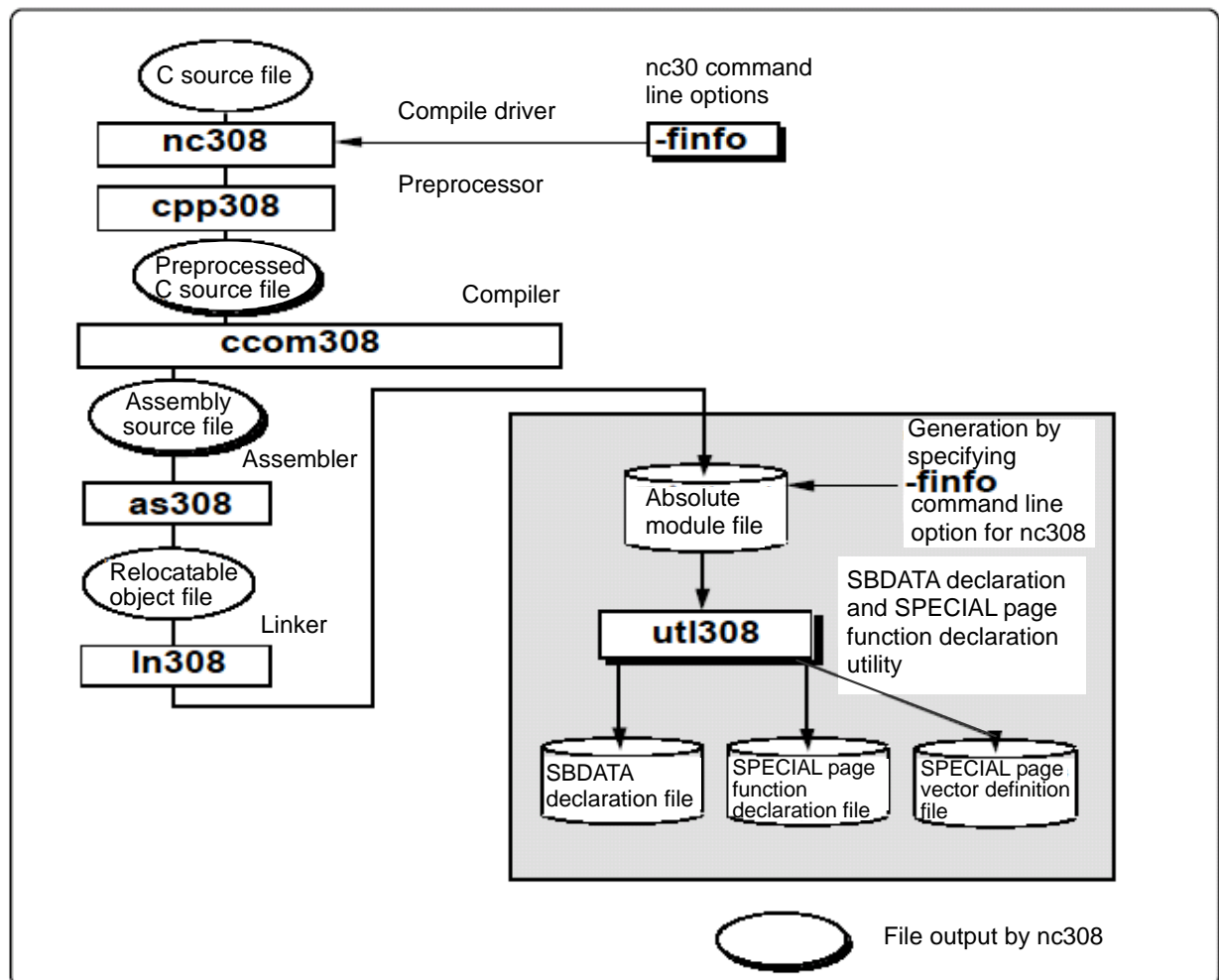


Figure 5.9 SBDATA Declaration and SPECIAL Page Function Declaration Utility

5.7 Using "switch" Instead of "else if"

When comparing arrays or other structures multiple times, it is faster to use a "switch" statement than an "else if" statement. This is because "else if" statements perform comparisons through indirect addressing, while "switch" statements secure space and perform comparisons within the register.

Before	After
<pre> if(a[i]==0){ ... } else if(a[i]==1){ ... } else if(a[i]==2){ ... } else { ... } </pre>	<pre> switch(a[i]) { case 0: ... break; case 1: ... break; case 2: ... break; default: ... beak; } </pre>
<pre> ;## # C_SRC : if(a[i]==0) mov.w R0,R1 ; i i shl.w #1,R0 mov.w R0,A0 mov.w _a:16[A0],R0 jne L1 ... ;## # C_SRC : else if(a[i]==1) L1: mov.w R1,A0 ; i i shl.w #1,A0 cmp.w #0001H,_a:16[A0] jne L11 ... ;## # C_SRC : else if(a[i]==2) L11: mov.w R1,A0 ; i i shl.w #1,A0 cmp.w #0002H,_a:16[A0] ... </pre>	<pre> indexws.w R0 ; i mov.w:g _a:16,R0 cmp.w #0000H,R0 jeq L3 cmp.w #0001H,R0 jeq L5 cmp.w #0002H,R0 jeq L7 jmp L9 ... </pre>

Figure 5.10 Using "switch" Instead of "else if"

5.8 Comparison Operators for Loop Counters

Within a loop, comparisons to 0 both use less ROM, and execute faster.

Before	After
<pre>for(i=0;i<10;i++){ } ...</pre>	<pre>for(i=10;i!=0;i--){ } ...</pre>
<pre>add.w #0001H,_i cmp.w #000aH,_i jlt label</pre>	<pre>adjnz.w #-1,_i,label</pre>

Figure 5.11 Comparison Operators for Loop Counters

5.9 restrict

You can facilitate optimization by using the restrict modifier, as long as you are sure that you do not point to the same areas as variables. Note that if you use the restrict modifier when pointing to the same areas as variables, a malfunction may occur.

Before	After
<pre>int a; int *q; void f() { a=10; *q=20; sub(a); }</pre> <p style="text-align: center;">a needs to be loaded.</p>	<pre>int a; int * restrict q; void f() { a=10; *q=20; sub(a); }</pre> <p style="text-align: center;">This is replaced with a=10.</p>
<pre>_f: ;## # C_SRC : a=10; mov.w #000aH,_a:16 ;## # C_SRC : *q=20; mov.w #0014H,[_q:16] ;## # C_SRC : sub(a); mov.w _a:16,R0 jsr \$sub ;## # C_SRC : } rts</pre>	<pre>_f: ;## # C_SRC : a=10; mov.w #000aH,_a:16 ;## # C_SRC : *q=20; mov.w #0014H,[_q:16] ## # C_SRC : sub(a); mov.w #000aH,R0 jsr \$sub ;## # C_SRC : } rts</pre>

Figure 5.12 Example Usage of restrict

5.10 Using _Bool

_Bool is a Boolean type, which can be either 0 or 1. You can use the _Bool flag to prevent unnecessary exception processing.

Before	After
<pre>char flag; if(flag==0){ ... } else if(flag==1) { ... } else{ // exception processing }</pre>	<pre>_Bool flag; if(flag==0){ ... } else { ... }</pre>

Figure 5.13 Example Usage of _Bool

5.11 Explicitly Initializing auto Variables

When you initialize an auto variable, it is allocated to the register. Otherwise, it is stored in the stack. The optimization option is required for this optimization.

Before	After
<pre>extern unsigned int *p, max_val, min_val; void func(void) { unsigned int max = 0; unsigned int min; // Pushed to // stack while (1) { if (max == 0) min = max = *p; if (max < *p) max = *p; if (min > *p) min = *p; p++; if (*p == 0) break; } max_val = max; min_val = min; }</pre>	<pre>extern unsigned int *p, max_val, min_val; void func(void) { unsigned int max = 0; unsigned int min=*p; // Allocated // to register while (1) { if (max == 0) min = max = *p; if (max < *p) max = *p; if (min > *p) min = *p; p++; if (*p == 0) break; } max_val = max; min_val = min; }</pre>

<pre> .glb_func _func: enter #02H pushm R1 ;## # C_SRC : unsigned int max = 0; mov.w #0000H,R0 ; max ;## # C_SRC : while (1) L3: ;## # C_SRC : if (max == 0) min = max = *p; cmp.w #0000H,R0 ; max jne L7 mov.w [_p:16],R0 mov.w R0,R1 mov.w R1,-2[FB] ; min L7: ;## # C_SRC : if (max < *p) max = *p; cmp.w [_p:16],R0 ; max jgeu L17 mov.w [_p:16],R0 L17: ;## # C_SRC : if (min > *p) min = *p; cmp.w [_p:16],-2[FB] ; min mov.w [_p:16],-2[FB] ; min L27: ;## # C_SRC : p++; add.l #00000002H,_p:16 ;## # C_SRC : if (*p == 0) break; mov.w [_p:16],R1 jne L3 ;## # C_SRC : max_val = max; mov.w R0,_max_val:16 ; max ;## # C_SRC : min_val = min; mov.w -2[FB],_min_val:16 ; min ;## # C_SRC : } popm R1 exitd </pre>	<pre> .glb_func _func: pushm R1 ;## # C_SRC : unsigned int max = 0; mov.w #0000H,R0 ; max ;## # C_SRC : unsigned int min=*p; mov.w [_p:16],R1 ; min ;## # C_SRC : while (1) L3: ;## # C_SRC : if (max == 0) min = max = *p; cmp.w #0000H,R0 ; max jne L7 mov.w [_p:16],R0 mov.w R0,R1 L7: ._line 27 ;## # C_SRC : if (max < *p) max = *p; cmp.w [_p:16],R0 ; max jgeu L17 mov.w [_p:16],R0 L17: ._line 28 ;## # C_SRC : if (min > *p) min = *p; cmp.w [_p:16],R1 ; min jleu L27 mov.w [_p:16],R1 L27: ;## # C_SRC : p++; add.l #00000002H,_p:16 ;## # C_SRC : if (*p == 0) break; cmp.w #0000H,[_p:16] jne L3 ;## # C_SRC : max_val = max; mov.w R0,_max_val:16 ; max ;## # C_SRC : min_val = min; mov.w R1,_min_val:16 ; min ;## # C_SRC : } popm R1 rts </pre>
--	--

Figure 5.14 Explicitly Initializing auto Variables

5.12 Initializing Arrays

When two arrays are initialized within a loop statement, move one of them to a separate loop. This allows the loops to be expanded using the sstr instruction, to increase execution speed.

However, when initializing three or more arrays, perform initialization within the same loop to improve ROM efficiency. This is because the sstr instruction requires an initial setup for each register, degrading ROM efficiency when 3 or more arrays are initialized.

Before	After
<pre> /* Initialization within the same loop */ void loop2_1(void) { int i; for(i = 0; i < 10; i++){ a[i] = 0x0a; b[i] = 0x0b; } } </pre>	<pre> /* Initialization within a separate loop */ void loop2_2(void) { int i; for(i = 0; i < 10; i++) a[i] = 0x0a; for(i = 0; i < 10; i++) b[i] = 0x0b; } </pre>
<pre> _loop2_1: pushm A0 mov.w #0000H,R0 L3: mov.w R0,A0 mov.b #0aH,_a:16[A0] mov.b #0bH,_b:16[A0] add.b #01H,A0 mov.w A0,R0 cmp.w #000aH,R0 jlt L3 popm A0 rts </pre>	<pre> _loop2_2: pushm R3,A1 mov.b #0aH,R0L mov.w #(_a&0FFFFH),A1 mov.w #0aH,R3 sstr.b mov.b #0bH,R0L mov.w #(_b&0FFFFH),A1 mov.w #0aH,R3 sstr.b popm R3,A1 rts </pre>

Figure 5.15 Initializing Arrays

5.13 Increments / Decrements

Separate increments and decrements from expressions. The compiler will attempt to keep the value before the increment/decrement.

Before	After
<pre> a[i++] = b[j++]; </pre>	<pre> a[i] = b[j]; i++; j++; </pre>
<pre> mov.w -22[FB],R0 ; j add.w #0001H,-22[FB] ; j indexws.w R0 mov.w:g -22[FB],R0 ; b indexwd.w -2[FB] ; i mov.w R0,-22[FB] ; a add.w #0001H,-2[FB] ; i </pre>	<pre> indexws.w -4[FB] ; j mov.w:g -24[FB],R0 ; b indexwd.w -2[FB] ; i mov.w R0,-24[FB] ; a add.w #0001H,-2[FB] ; i add.w #0001H,-4[FB] ; j </pre>

Figure 5.16 Increments / Decrements

5.14 "Switch" Statements

For more efficient code, in "switch" statements with several case values, reduce the intervals between the case values when possible. This is because code for large intervals is expanded to "if else" formats, while that for small intervals is expanded to branch tables. It is also a good idea to use the enum type for case values.

Before	After
<pre>switch(a) { case 100: ... case 200: ... case 300: ... }</pre>	<pre>switch(a) { case 1: ... case 2: ... case 3: ... }</pre>
<pre>cmp.w #0064H,R0 jeq L5 cmp.w #00c8H,R0 jeq L7 cmp.w #012cH,R0 jeq L9 ...</pre>	<pre>S2: jmp.i.w L47 L47: .word L45-S2&0ffffH .word L23-S2&0ffffH .word L25-S2&0ffffH ...</pre>

Figure 5.17 "switch" Statements

5.15 Immediate Floating-points

When precision is not an issue, append the f suffix to floating-point numbers. A number without the f suffix is processed as a double type.

Before	After
<pre>float f; f=f+123.456;</pre>	<pre>float f; f=f+123.456f;</pre>
<pre>push.l _f:16 .glb __f4tof8 jsr.a __f4tof8 add.l #04H,SP pushm R3,R2,R1,R0 push.l #405edd2fH push.l #1a9fbe77H jsr.a __f8add add.l #010H,SP pushm R3,R2,R1,R0 jsr.a __f8tof4 add.l #08H,SP mov.l R2R0, f:16</pre>	<pre>push.l _f:16 push.l #42f6e979H jsr.a __f4add add.l #08H,SP mov.l R2R0, _f:16</pre>

Figure 5.18 Immediate Floating-points

5.16 Zero Clearing External Variables

During startup, the initial value of all external variables is zero cleared, even when not explicitly set as such. When you explicitly zero clear a variable, since a 0 is secured in ROM and transferred during startup, ROM space is wasted. As such, do not zero clear the initial value of an external variable.

Before	After
<pre>int i=0</pre>	<pre>int i;</pre>
<pre>.SECTION data_NE,DATA ._inspect 'U', 1, "data_NE", "data_NE", 0 .glb_i _i: .blkb 2 .SECTION data_NEI,ROMDATA ._inspect 'U', 1, "data_NEI", "data_NEI", 0 ;## # init data of i. .word 0000H</pre>	<pre>.SECTION bss_NE,DATA ._inspect 'U', 3, "bss_NE", "bss_NE", 0 .glb_i _i: .blkb 2</pre>

Figure 5.19 Zero Clearing External Variables

5.17 Organizing Startup

If you know that no standard I/O functions or memory management functions will be used, you can speed up the program by omitting their initial setup. By specifying "-D__STANDARD_IO__=1" and "-D__HEAP__=1" at assembly-time for ncr0.a30, you can disable initialization processing for these functions. Figure 5.21 shows the High-performance Embedded Workshop dialog box for setting "-D__STANDARD_IO__=1" and "-D__HEAP__=1".

```
-----  
; HEAP SIZE definition  
-----  
.if __HEAP__ == 1 ; for HEW  
  
HEAPSIZE .equ 0h  
  
.else  
.if __HEAPSIZE__ == 0  
HEAPSIZE .equ 300h  
.else ; for HEW  
HEAPSIZE .equ __HEAPSIZE__  
.endif  
.endif  
  
~~~~~  
=====  
; Initialize standard I/O  
-----  
.if __STANDARD_IO__ != 1  
.glb _init  
.call _init,G  
.jsr.a _init  
.endif
```

Figure 5.20 Organizing Startup

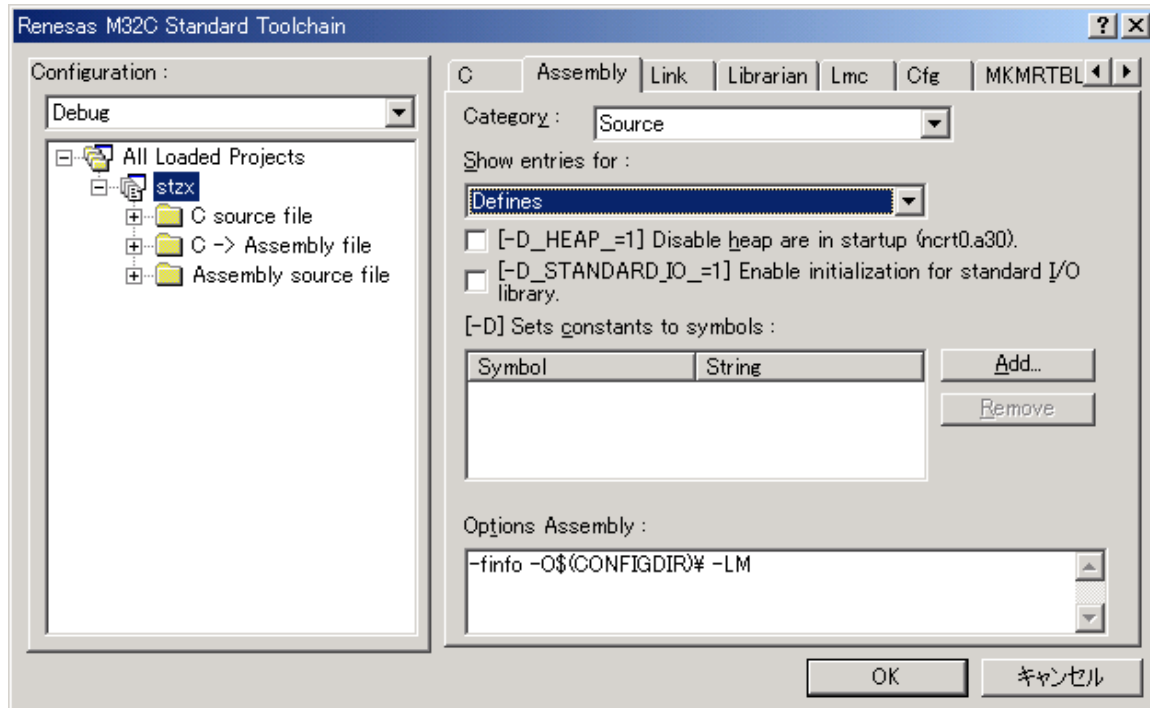


Figure 5.21 The High-performance Embedded Workshop Dialog Box for Organizing Startup

5.18 Using Temporary Values within Loops

When an external variables is used within a loop, memory is referenced for each iteration. You can facilitate calculation in the register by using temporary variables.

Before	After
<pre>int total; void func(int p[]) { int i; total=0; for(i=0;i<100;i++){ total+=p[i]; // Memory referenced // each time } }</pre>	<pre>int total; void func(int p[]) { int i,tmp; tmp=0; for(i=0;i<100;i++){ tmp+=p[i]; // Calculation performed // in register. } total=tmp; }</pre>
<pre>_func: enter #02H pushm R2,A0 ;## # C_SRC : total=0; mov.w #0000H,_total:16 ;## # C_SRC : for(i=0;i<100;i++){ mov.w #0000H,-2[FB] ; i L1: ;## # C_SRC : for(i=0;i<100;i++){ cmp.w #0064H,-2[FB] ; i jge L5 ;## # C_SRC : total+=p[i]; //Memory referenced each time. mov.w -2[FB],R0 ; i exts.w R0 mov.l R2R0,A0 shl.l #1,A0 add.l 8[FB],A0 ; p add.w [A0],_total:16 add.w #0001H,-2[FB] ; i jmp L1 L5: ;## # C_SRC : } popm R2,A0 exitd</pre> <div data-bbox="598 1099 794 1234" style="border: 1px solid black; border-radius: 15px; padding: 5px; display: inline-block;"> <p>Memory is referenced for each iteration in the loop.</p> </div>	<pre>_func: enter #00H pushm R1,R2,R3,A0,A1 mov.l 8[FB],A0 ; p p ;## # C_SRC : tmp=0; mov.w #0000H,R1 ; tmp ;## # C_SRC : for(i=0;i<100;i++){ mov.w #0000H,R0 ; i L1: ;## # C_SRC :tmp+=p[i]; // Calculation performed in register. mov.w R0,R3 ; i i exts.w R0 mov.l R2R0,A1 shl.l #1,A1 add.l A0,A1 ; p add.w [A1],R1 add.w #0001H,R3 ; i mov.w R3,R0 ; i i cmp.w #0064H,R0 ; i jlt L1 ;## # C_SRC : total=tmp; mov.w R1,_total:16 ; tmp ;## # C_SRC : } popm R1,R2,R3,A0,A1 exitd</pre> <div data-bbox="1182 1081 1385 1200" style="border: 1px solid black; border-radius: 15px; padding: 5px; display: inline-block;"> <p>Replaced by a calculation in the register.</p> </div>

Figure 5.22 Using Temporary Values within Loops

5.19 Using 32-bit Mathematical Functions

You can use 32-bit mathematical functions for variables of the float type to increase execution speed. When the standard mathematical functions are used, the arguments are first converted up to the double type, and the resulting double type value is converted back to a float type value when substituted for the variable. When 32-bit mathematical operations are used, all calculations are performed using the float type.

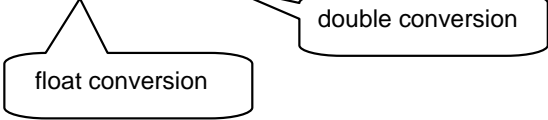
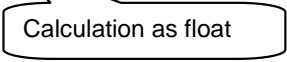
Before	After
<pre data-bbox="215 600 470 667">#include<math.h> float fdata,result; result=sin(fdata);</pre> 	<pre data-bbox="815 600 1066 667">#include<mathf.h> float fdata,result; result=sinf(fdata)</pre> 
<pre data-bbox="252 813 671 1084">;## # C_SRC : result=sin(fdata); push.l _fdata:16 .glb __f4tof8 jsr.a __f4tof8 add.l #04H,SP pushm R0,R1,R2,R3 jsr _sin add.l #08H,SP pushm R3,R2,R1,R0 .glb __f8tof4 jsr.a __f8tof4 add.l #08H,SP mov.l R2R0, result:16</pre>	<pre data-bbox="850 813 1283 913">;## # C_SRC : result=sinf(fdata); push.l _fdata:16 jsr _sinf add.l #04H,SP mov.l R2R0,_result:16</pre>

Figure 5.23 Using 32-bit Mathematical Functions

5.20 Using unsigned Whenever Possible

Use unsigned integers whenever possible, to improve code efficiency for branching instructions (note, however, that this can worsen code efficiency for only the M16C/20 and M16C/60 series, as adding "unsigned" in NC308 incurs sign extension). Also, when comparing signed integers, use the != and == operators instead of the <=, >=, <, > operators to improve code efficiency whenever possible.

Before	After
<pre>int data[100]; void main() { int i; int sum=0; for(i=0;i<100;i++) sum+=data[i]; }</pre>	<pre>int data[100]; void main() { unsigned int i; int sum=0; for(i=0;i!=100;i++) sum+=data[i]; }</pre>
<pre>;;# # C_SRC : int sum=0; mov.w #0000H,-4[FB] ; sum ;;# # C_SRC :for(i=0;i<100;i++)sum+= data[i]; mov.w #0000H,-2[FB] ; i L1: ;;# # C_SRC : for(i=0;i<100;i++)sum+= data[i]; cmp.w #0064H,-2[FB] ; i jge L5 mov.w -2[FB],A0 ; i shl.w #1,A0 add.w _data:16[A0],-4[FB] ; sum add.w #0001H,-2[FB] ; i jmp L1 L5:</pre>	<pre>;;# # C_SRC : int sum=0; mov.w #0000H,-4[FB] ; sum ;;# # C_SRC :for(i=0;i!=100;i++)sum+= data[i]; mov.w #0000H,-2[FB] ; i L1: ;;# # C_SRC :for(i=0;i!=100;i++)sum+= data[i]; cmp.w #0064H,-2[FB] ; i jeq L5 mov.w -2[FB],A0 ; i shl.w #1,A0 add.w _data:16[A0],-4[FB] ; sum add.w #0001H,-2[FB] ; i jmp L1 L5:</pre>

Figure 5.24 Example Usage of Unsigned

5.21 Array Index Types

Type expansion is performed during calculation for array indexes, based on the size of an element in the array.

- (1) For sizes 2 bytes or larger (for types other than the char type or signed char type):

Calculation is performed by extending the index to the int type.

- (2) For far-type arrays of 64K or larger:

Calculation is performed by extending the index to the long type.

As such, when you use a char-type variable as the array index, it must be extended to the int type, making for inefficient code. So, use the int type, instead of the char type, for array indexes. Note that this optimization is for NC30WA only.

Before	After
<pre>char i; int a[10]; a[i]=0;</pre>	<pre>int i; int a[10]; a[i]=0;</pre>
<pre>mov.b -1[FB],R0L ; i mov.b #00H,R0H shl.w #01H,R0 mova -21[FB],A0 ; a add.w R0,A0 mov.w #0000H,[A0]</pre>	<pre>mov.w -2[FB],R0 ; i shl.w #01H,R0 mova -22[FB],A0 ; a add.w R0,A0 mov.w #0000H,[A0]</pre>

Figure 5.25 Array Index Types

5.22 Using Prototype Declarations

With this compiler, you can use function prototype declarations to perform more efficient function calls. In other words, if you do not use function prototype declarations with this compiler, when such functions are called, their arguments are pushed to the stack area by the rules shown in the following table.

Table 5.4 Stack Usage Rules Pertaining to Arguments

Data type	Rule when pushed to a stack
char type signed char type	Extended to the int type.
float type	Extended to the double type.
Other types	Not extended.

This means that when function prototype declarations are not used, redundant type extension may be performed.

You can use function prototype declarations to improve efficiency of function calls, to avoid redundant type extension and allow arguments to be allocated to the register.

Before	After
<pre> int main() { char data1; char data2; char data3; int data; data=f(data1,data2,data3); } int f(i, j, k) char i,j,k; { ... } </pre>	<pre> int f(char ,char ,char); int main() { char data1; char data2; char data3; int data; data=f(data1,data2,data3); } int f(i, j,k) char i,j,k; { ... } </pre>
<pre> extz -2[FB],R0 ; data3 push.w R0 extz -2[FB],R0 ; data2 push.w R0 extz -2[FB],R0 ; data1 push.w R0 jsr _f </pre>	<pre> push.b -2[FB] ; data3 push.b -2[FB] ; data2 mov.b -2[FB],R0L ; data1 jsr \$f </pre>

Figure 5.26 Using Prototype Declarations

5.23 Using the char Type for Functions that Return only char Type Values

You can decrease the amount of ROM space used by using the char type for the return value of functions that only return values of this type.

Also, use smaller data types whenever possible.

Before	After
<pre>int func2(void) { switch (x) { case 0: return 255; case 1: return 254; default: return 253; } }</pre>	<pre>char func2(void) { switch (x) { case 0: return 255; case 1: return 254; default: return 253; } }</pre>
<pre>.glob _func2 _func2: ;## # C_SRC : switch (x) { mov.w _x:16,R0 jeq L3 cmp.w #0001H,R0 jeq L5 jmp L7 ;## # C_SRC : case 0: L3: ;## # C_SRC : return 255; mov.w #00ffH,R0 rts ;## # C_SRC : case 1: L5: ;## # C_SRC : return 254; mov.w #00feH,R0 rts ;## # C_SRC : default: L7: ;## # C_SRC : return 253; mov.w #00fdH,R0 rts</pre>	<pre>.glob _func2 _func2: ;## # C_SRC : switch (x) { mov.w _x:16,R0 jeq L3 cmp.w #0001H,R0 jeq L5 jmp L7 ;## # C_SRC : case 0: L3: ;## # C_SRC : return 255; mov.b #0ffH,R0L rts ;## # C_SRC : case 1: L5: ;## # C_SRC : return 254; mov.b #0feH,R0L rts ;## # C_SRC : default: L7: ;## # C_SRC : return 253; mov.b #0fdH,R0L rts</pre>

Figure 5.27 Using the char Type for Functions that Return only char Type Values

5.24 Commenting Out Clear Processing for bss Areas

The startup program `ncrt0.a30` contains bss area clear processing. This processing exists to satisfy the C language specification that uninitialized variables should have 0 as their initial value.

For example, since the code shown in Figure 5.28 contains a variable with no initial value, processing (bss area clear processing) is required during startup to set 0 as the initial value.

```
static int i;
```

Figure 5.28 Example Declaration for a Variable that Has no Initial Value

Depending on the application, variables without an initial value may not need to be zero cleared. In this case, you can speed up startup processing by commenting out the part of the startup program where bss area clear processing takes place.

```
=====
; NEAR area initialize.
;-----
; bss zero clear
;-----
;         N_BZERO bss_SE_top,bss_SE
;         N_BZERO bss_SO_top,bss_SO
;         N_BZERO bss_NE_top,bss_NE
;         N_BZERO bss_NO_top,bss_NO
;
; (omitted)
;
;-----
; FAR area initialize.
;-----
; bss zero clear
;-----
;         BZERO bss_FE_top,bss_FE
;         BZERO bss_FO_top,bss_FO
```

Figure 5.29 Example of Commenting Out of Clear Processing for bss Areas

5.25 Reducing Generated Code

You can reduce the amount of generated code for data that is declared using the int type, and is within the following ranges:

From 0 to 255: Change to the unsigned char type.

From -128 to 127: Change to the signed char type.

By decreasing the size of the variable type, you can reduce the size of comparison, addition, and other instructions, and thus increase ROM efficiency.

Before	After
<pre>int type int data[128]; void main(void) { int cnt = 128; int i; int sum=0; for(i = 0 ; i < cnt ; i++){ sum += data[i]; } }</pre>	<pre>unsigned char type void main(void) { unsigned char cnt = 128; unsigned char i; int data[128]; int sum; for(i = 0 ; i < cnt ; i++){ sum += data[i]; } }</pre>
<pre>## # C_SRC : int cnt = 128; mov.w #0080H,-6[FB] ; cnt ## # C_SRC : for(i = 0 ; i < cnt ; i++) mov.w#0000H,-4[FB] ; i L1: ## # C_SRC : for(i = 0 ; i < cnt ; i++) cmp.w -6[FB],-4[FB] ; cnt i jge L5 ## # C_SRC : sum += data[i]; mov.w -4[FB],A0 ; i shl.w #1,A0 mova -262[FB],A1 ; data add.l A1,A0 add.w [A0],-2[FB] ; sum add.w #0001H,-4[FB] ; i jmp L1 L5: ## # C_SRC : } popm A0,A1 exitd</pre>	<pre>## # C_SRC : unsigned char cnt = 128; mov.b #80H,-2[FB] ; cnt ## # C_SRC : for(i = 0 ; i < cnt ; i++) mov.b #00H,-1[FB] ; i L1: ## # C_SRC : for(i = 0 ; i < cnt ; i++) cmp.b -2[FB],-1[FB] ; cnt i jgeu L5 ## # C_SRC : sum += data[i]; mov.b -1[FB],A0 ; i shl.w #1,A0 mova -260[FB],A1 ; data add.l A1,A0 add.w [A0],-4[FB] ; sum add.b #01H,-1[FB] ; i jmp L1 L5: ## # C_SRC : } popm A0,A1 exitd</pre>

Figure 5.30 Reducing Generated Code (1)

You can reduce the amount of generated code for data that is declared using the long type, and is within the following ranges and meets the following criteria:

From 0 to 65535: Change to the unsigned int type.

From -32768 to 32767: Change to the signed int type.

By decreasing the size of the variable type, you can reduce the size of comparison, addition, and other instructions, and thus increase ROM efficiency.

Before	After
<pre> unsigned int data[65535]; void main(void) { long cnt = 65535; long i; int sum; for(i = 0 ; i < cnt ; i++){ sum += data[i]; } } </pre>	<pre> unsigned int data[65535]; void main(void) { unsigned int cnt = 65535; unsigned int i; int sum; for(i = 0 ; i < cnt ; i++){ sum += data[i]; } } </pre>
<pre> ;## # C_SRC : long cnt = 65535; mov.l #0000ffffH,-10[FB]; cnt ;## # C_SRC : for(i = 0 ; i < cnt ; i++) mov.l #00000000H,-6[FB] ; i L1: ;## # C_SRC : for(i = 0 ; i < cnt ; i++) cmp.l -10[FB],-6[FB] ; cnt i jge L5 ;## # C_SRC : sum += data[i]; mov.l -6[FB],A0 ; i shl.l #1,A0 add.w _data:16[A0],-2[FB] ; sum add.l #00000001H,-6[FB] ; i jmp L1 L5: ;## # C_SRC : } popm A0 exitd </pre>	<pre> ;## # C_SRC : unsigned int cnt = 65535; mov.w #0ffffH,-6[FB] ; cnt ;## # C_SRC : for(i = 0 ; i < cnt ; i++) mov.w #0000H,-4[FB] ; i L1: ;## # C_SRC : for(i = 0 ; i < cnt ; i++) cmp.w -6[FB],-4[FB] ; cnt i jgeu L5 ;## # C_SRC : sum += data[i]; mov.w -4[FB],A0 ; i shl.l #1,A0 add.w _data:16[A0],-2[FB] ; sum add.w #0001H,-4[FB] ; i jmp L1 L5: ;## # C_SRC : } popm A0 exitd </pre>

Figure 5.31 Deleting Generated Code (2)

Section 6. Using the Simulator Debugger

This chapter explains how to use the simulator debugger supplied with the NC30WA effectively, and consists of the following contents:

No.	Category	Item	Section
1	Using the Virtual Interrupt Function	Inserting a Virtual Interrupt by Button Click	6.1.1
2		Inserting a Virtual Interrupt at a Regular Interval	6.1.2
3		Inserting a Virtual Interrupt at a Specified Cycle	6.1.3
4		Inserting a Virtual Interrupt When an Instruction at a Specified Address Is Executed	6.1.4
5	Using the Virtual Port Input/Output Function	Entering Data by Button Click	6.2.1
6		Entering Data from a Virtual Port When a Specified Address Is Read	6.2.2
7		Entering Data from a Virtual Port at a Specified cycle	6.2.3
8		Entering Data from a Virtual Port When a Virtual Interrupt Occurs	6.2.4
9		Checking Data Output to a Virtual Output Port	6.2.5
10	Using a Virtual LED or Label to Check the Memory Contents		6.3
11	Using printf for Debugging		6.4
12	Using I/O Scripts		6.5

6.1 Using the Virtual Interrupt Function

6.1.1 Inserting a Virtual Interrupt by Button Click

■ Description:

An interrupt can be generated manually by clicking a virtual interrupt button as if the button click is a cause of the interrupt.

Set the interrupt priority and interrupt condition for the button.

■ How to create a button for generating a virtual interrupt manually:

1. From the menu, choose [View -> Graphics -> GUI I/O] to display the GUI window.
2. Either click the button creation icon, or right-click and choose [Create button] from the menu displayed to create a button for generating a virtual interrupt.

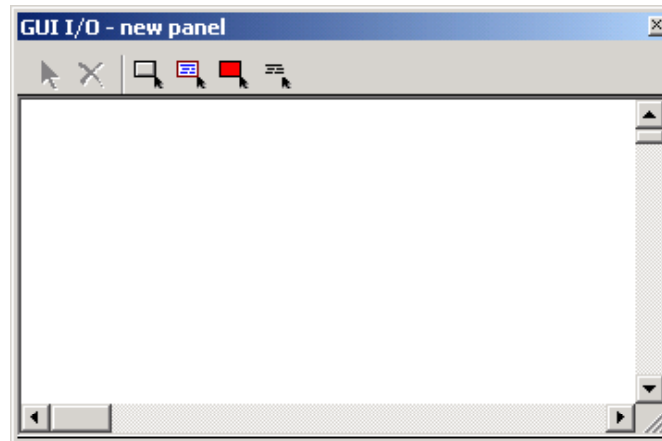


Figure 6.1 GUI Window

3. Click the created button to display the settings window for the button.

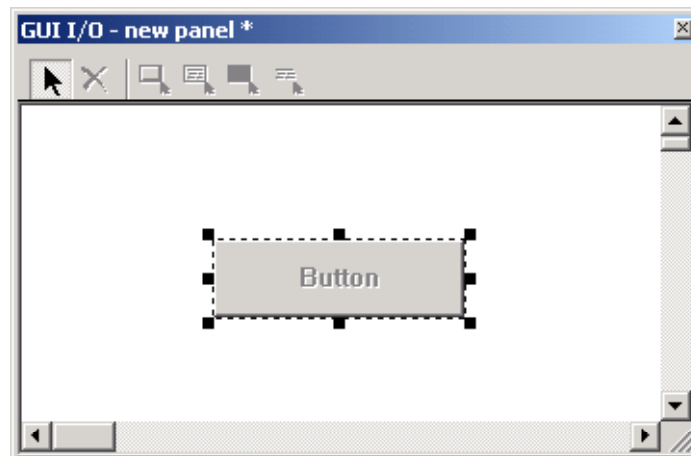


Figure 6.2 GUI Window (After Button Placement)

- For the button type, select [Interrupt], and then set the interrupt vector and IPL (interrupt priority level).

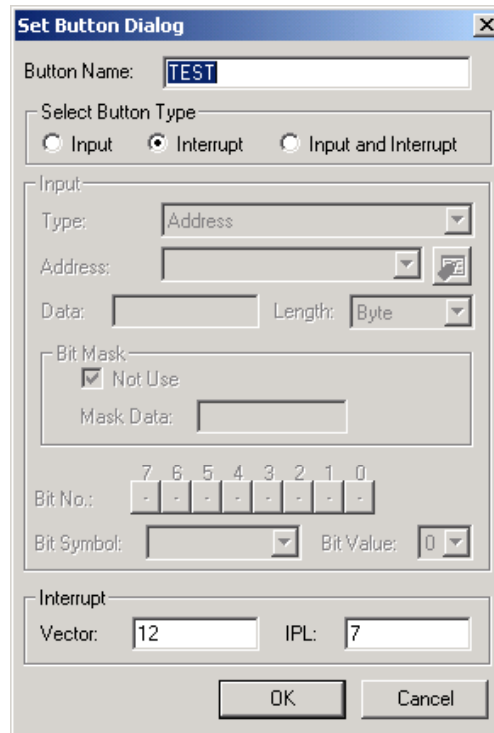


Figure 6.3 Button Settings Window

- In the file for which the vector table is set, set the interrupt vector. In this case, in the [vector 12] line, [.lword dummy_int] is changed to [.lword _test], and the test function in the source code is called when the button is pressed.

```

;-----
; variable vector section
;-----
.section vector,ROMDATA ; variable vector table
.org VECTOR_ADR

.lword dummy_int ; vector 0
.lword dummy_int ; vector 1

:

.glob _test
.lword _test ; vector 12

:

```

- During debugging, click the button created in step 2 to generate a virtual interrupt.

6.1.2 Inserting a Virtual Interrupt at a Regular Interval

■ Description:

A virtual interrupt that occurs in sync with a set time interval can be set up in the [I/O Timing Setting] window.

■ How to set up:

1. From the menu, choose [View -> CPU -> I/O Timing Setting] to display the [I/O Timing Setting] window.
2. Either click the time interval sync interrupt icon, or right-click and choose [Timer] from the menu displayed to display the window for setting the interval sync interrupt.

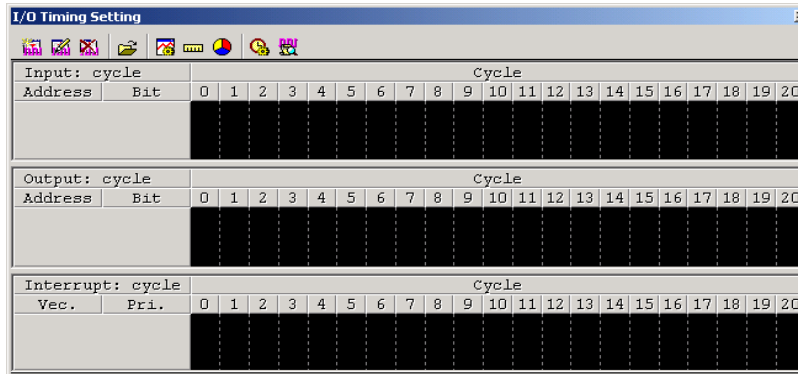


Figure 6.4 I/O Timing Setting Window

3. Click the [Load...] button to import a settings file.
4. Unless you are importing a settings file, enter the time interval, vector, and priority, and then click the [Add] button.

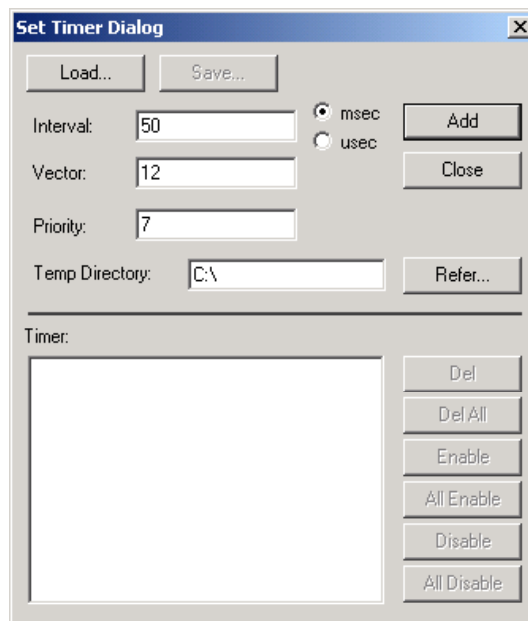


Figure 6.5 Set Timer Dialog Box

5. The input settings are displayed as entered.
6. Click the [Save...] button to save the current settings.

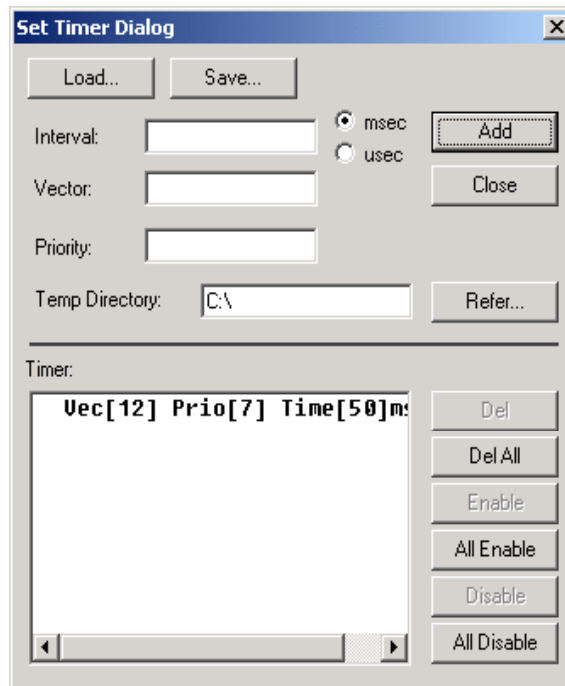


Figure 6.6 Set Timer Dialog Box (After Settings Are Input)

7. You can add multiple settings as necessary.
8. You can also toggle each setting, to enable or disable it, as necessary.

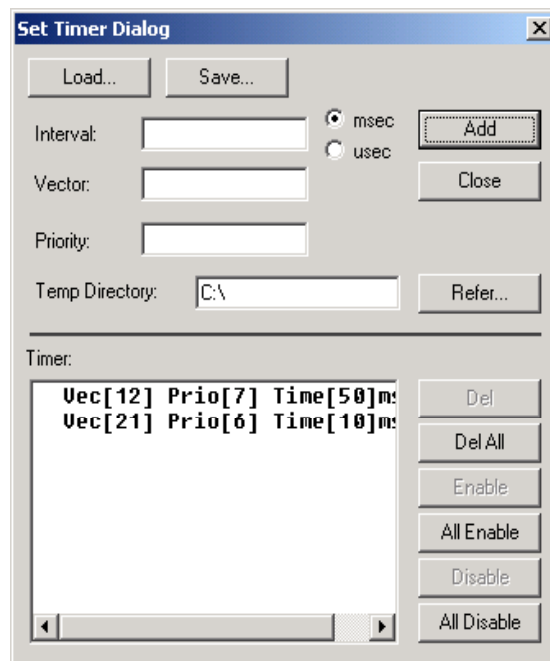


Figure 6.7 Set Timer Dialog Box (Multiple Settings)

6.1.3 Inserting a Virtual Interrupt at a Specified Cycle

■ Description:

A virtual interrupt that occurs at a specified cycle can be set up in the [I/O Timing Setting] window.

■ How to set up:

1. From the menu, choose [View -> CPU -> I/O Timing Setting] to display the [I/O Timing Setting] window.
2. Either click the data settings icon, or right-click and choose [Setup] from the menu displayed.

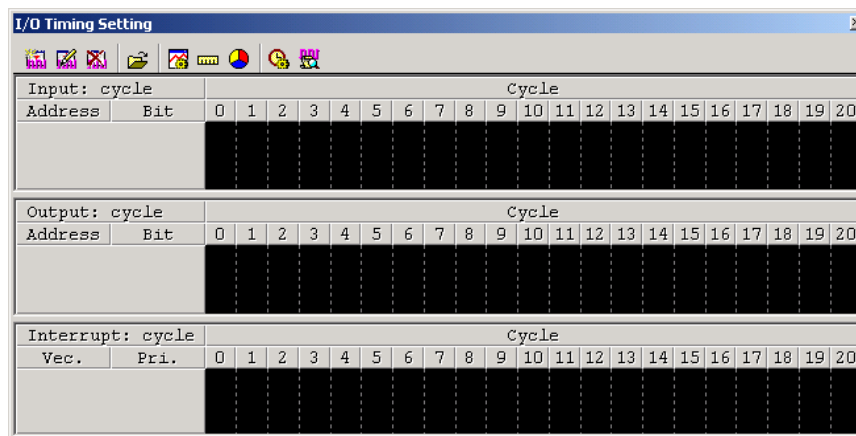


Figure 6.8 I/O Timing Setting Window

3. Select [Set a virtual interrupt], and then click the [Next>] button.

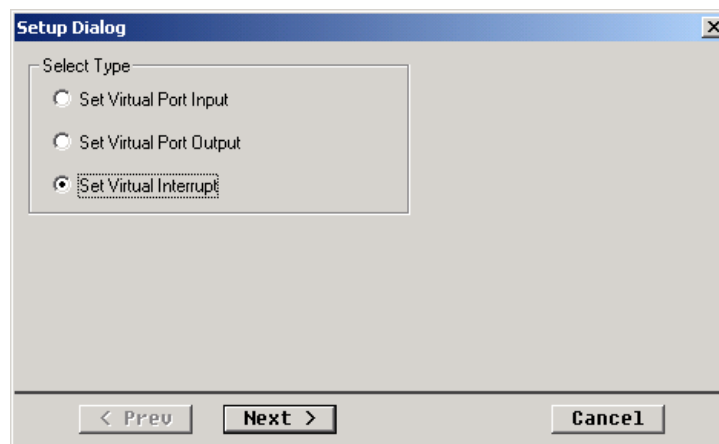


Figure 6.9 Setting a Virtual Interrupt

4. For the timing at which the interrupt is to occur, select [Cycle].
5. Set the start cycle, end cycle, vector, and priority, and then click the [Next>] button.

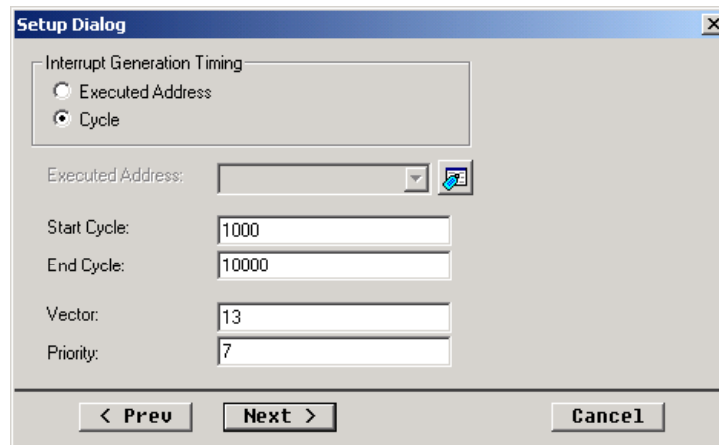


Figure 6.10 Setting a Cycle and Vector

6. Position the mouse pointer at the cell of the desired cycle at which you would like to set a virtual interrupt, and left click to set an interrupt to occur.
7. Once all interrupts to occur are set, click the [Next>] button.

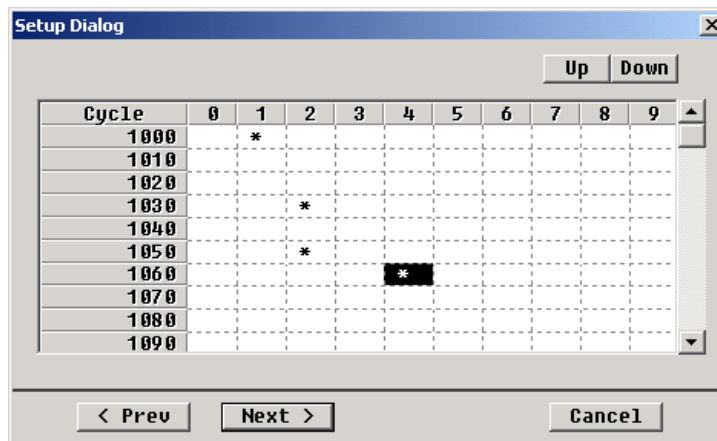


Figure 6.11 Setting Interrupts to Occur

8. Save the I/O script file, as it is modified automatically.

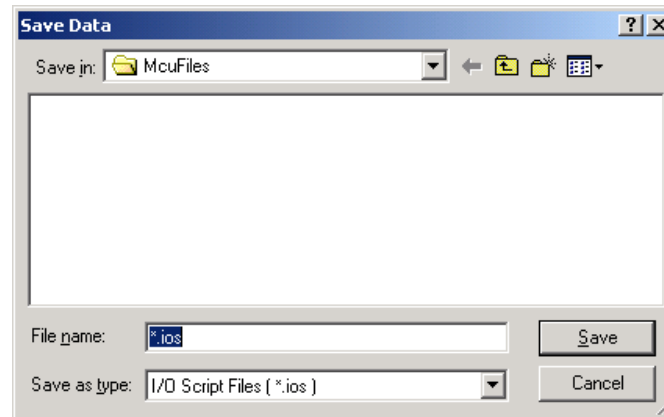


Figure 6.12 Save Dialog Box

9. The I/O script file is as follows.

```
; IOSSCRIPT FILE FOR I/O WINDOW (INT WAITC)
{
cycle 1001
int 13 , 7
waitc 31
int 13 , 7
waitc 20
int 13 , 7
waitc 12
int 13 , 7
}
```

6.1.4 Inserting a Virtual Interrupt When an Instruction at a Specified Address Is Executed

■ Description:

A virtual interrupt that occurs when an instruction at a specified address is executed can be set up in the [I/O Timing Setting] window.

■ How to set up:

1. Perform steps 1 to 3 in 6.1.3 Inserting a Virtual Interrupt at a Specified Cycle, to display a dialog box for setting up a virtual interrupt.
2. For the timing at which the interrupt is to occur, select [Execution address].
3. Set the execution address, vector, and priority, and then click the [Next>] button.

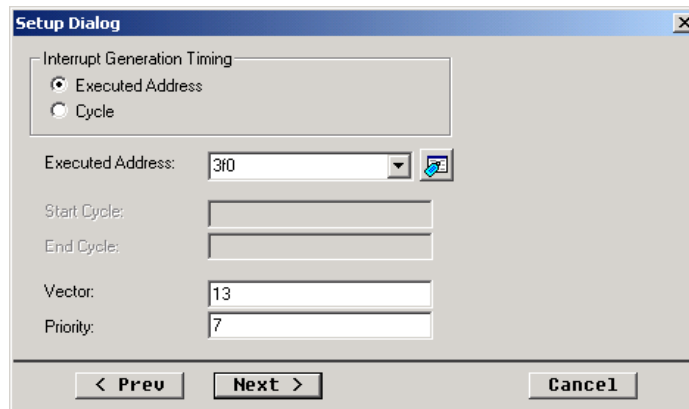


Figure 6.13 Setting the Execution Address and Vector

4. Set the number of times the instruction at the execution address is to be executed for the virtual interrupt to occur. Position the mouse pointer at the cell of the desired execution count, and left click to set the interrupt to occur.
5. Once all interrupts to occur are set, click the [Next>] button.

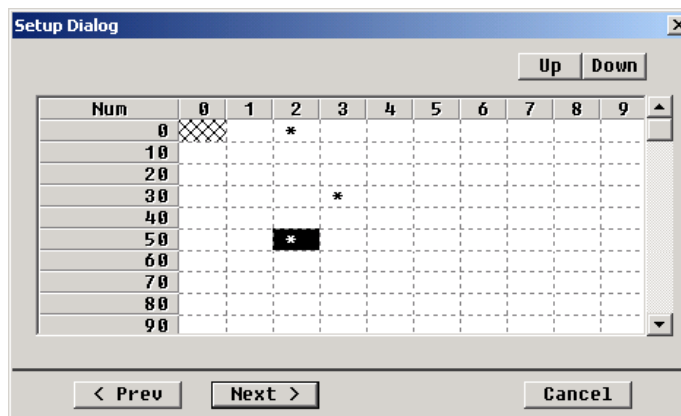


Figure 6.14 Setting Interrupts to Occur

6. Save the I/O script file, as it is modified automatically.

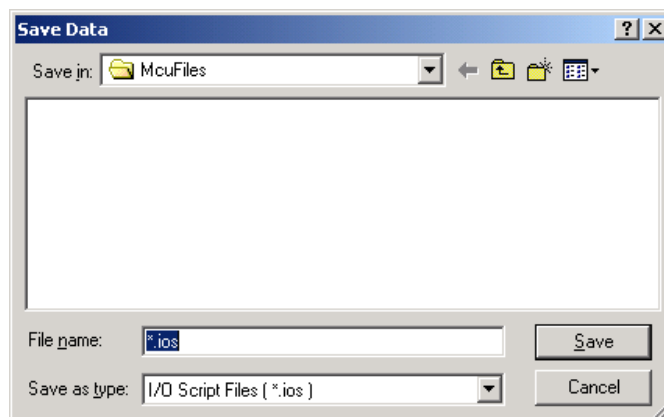


Figure 6.15 Save Dialog Box

7. The I/O script file is as follows.

```
; IOSCRIPT FILE FOR I/O WINDOW (INT ISFETCH)
{
pass #isfetch:0x3f0 , 2
int 13 , 7
pass #isfetch:0x3f0 , 31
int 13 , 7
pass #isfetch:0x3f0 , 19
int 13 , 7
}
```

6.2 Using the Virtual Port Input/Output Function

6.2.1 Entering Data by Button Click

■ Description:

A virtual port input can be generated manually by clicking a button.

Specify the address or the bit symbol for the button.

■ How to create a button for generating a virtual port input manually:

1. Place a button in the GUI window. For details, in 6.1.1 Inserting a Virtual Interrupt by Button Click, see steps 1 to 3.
2. For the button type, select [Input].
3. For the type, select [Address], [Address & Bit No.], or [Bit Symbol].
4. If [Address] is selected, specify the address, data, and data size.

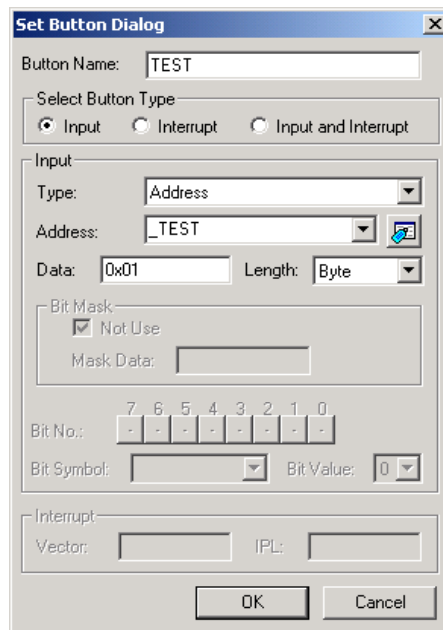


Figure 6.16 Button Settings

5. If [Address & Bit No.] is selected, set the address and bit number if no mask is to be used on a bit basis. If a mask is to be used on a bit basis, specify the address, data, data size, and mask value.

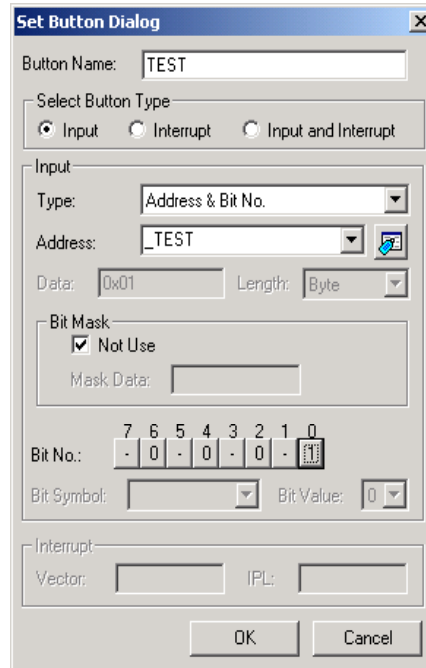


Figure 6.17 Button Settings

6. If [Bit Symbol] is selected, set the bit symbol and the value.

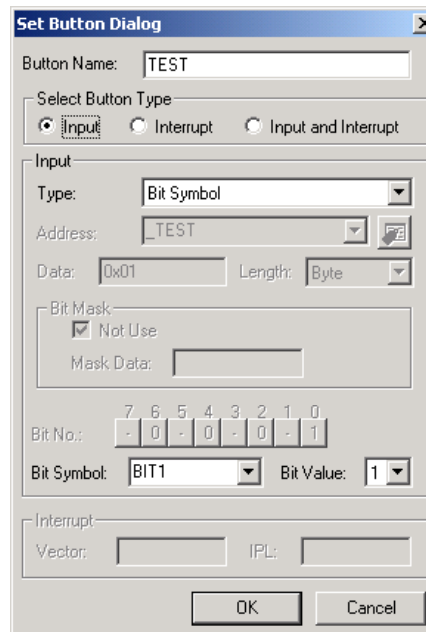


Figure 6.18 Button Settings

7. When the button is clicked, the value of the specified address or bit symbol is entered.

6.2.2 Entering Data from a Virtual Port When a Specified Address Is Read

■ Description:

Input that occurs from a virtual port when a specified address is read can be set up in the [I/O Timing Setting] window.

■ How to set up:

1. From the menu, choose [View -> CPU -> I/O Timing Setting] to display the [I/O Timing Setting] window.
2. Either click the data settings icon, or right-click and choose [Setup] from the menu displayed.

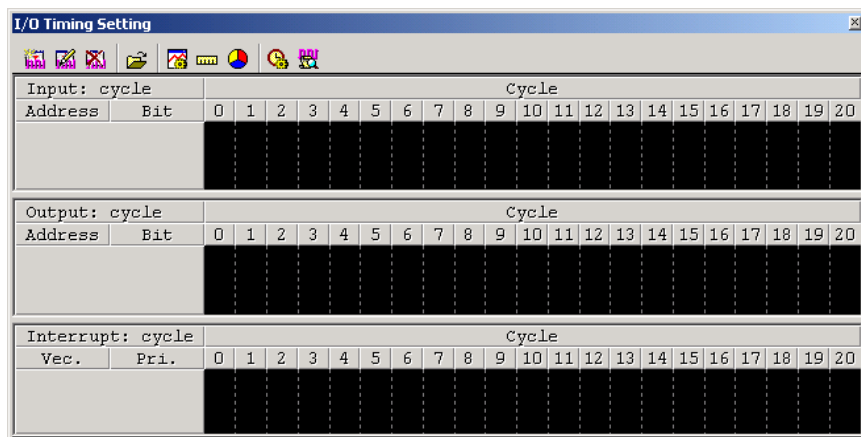


Figure 6.19 I/O Timing Setting Window

3. Select [Set a virtual input port], and click the [Next>] button.

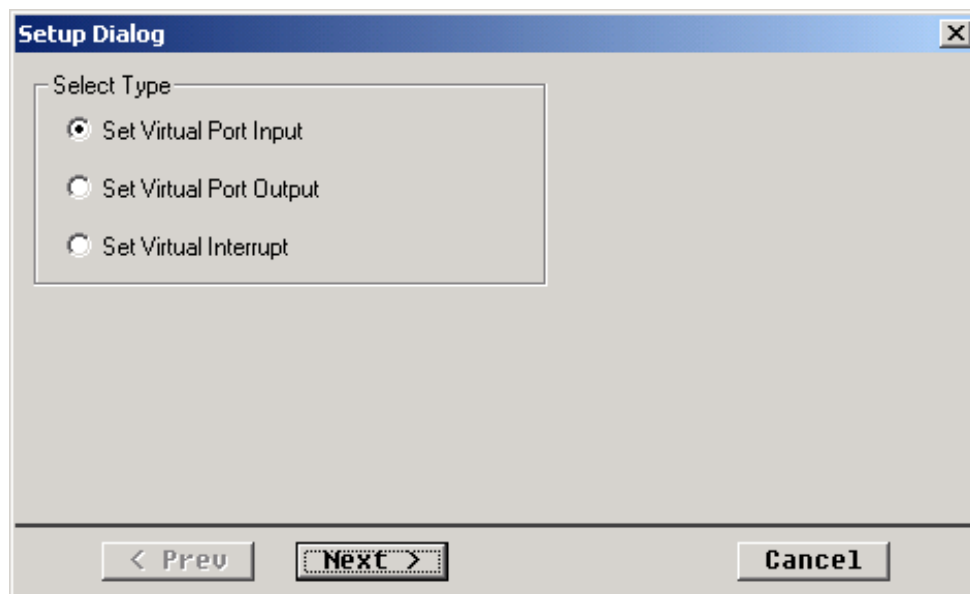


Figure 6.20 Setting the Processing Type

4. Set the data input timing to [Read access].
5. In the [Input address] field, use a hexadecimal number to enter the address for which virtual port input is to be performed.
6. In the [Read address] field, use a hexadecimal number to enter a memory address (virtual port input is performed when read access occurs for the memory address specified here).
7. When the same address is used for the input address and read address, the next piece of data is referenced when the address is accessed.
8. Click the [Next>] button.

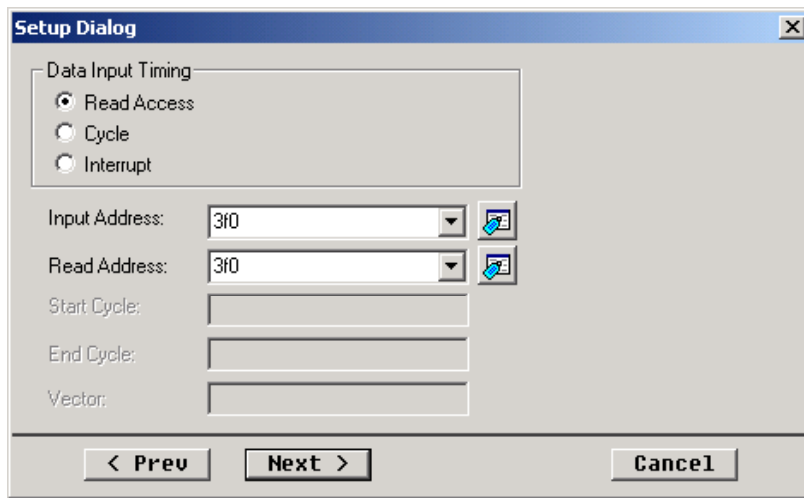


Figure 6.21 Read Access Settings

9. Set the data entered from the virtual input port. Position the mouse pointer at the cell of the desired read access occurrence count for setting the data, and double-click the left button to display the input console, and use a hexadecimal number to input the data.
10. Once the data is input, click the [Next>] button.

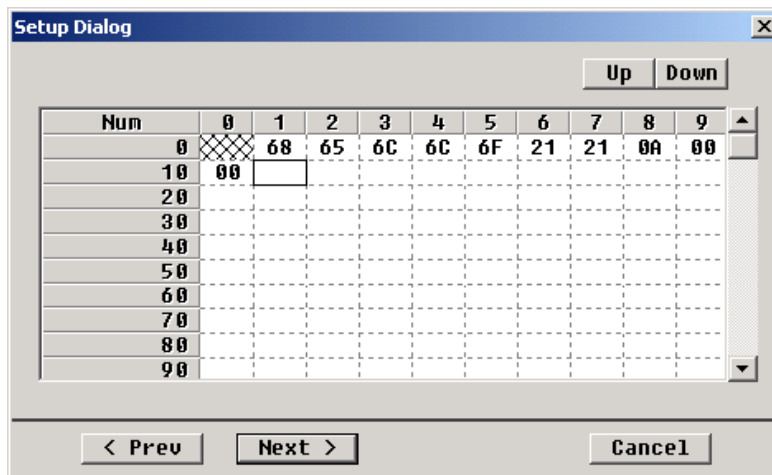


Figure 6.22 Input Data Settings

11. Save the I/O script file, as it is modified automatically.

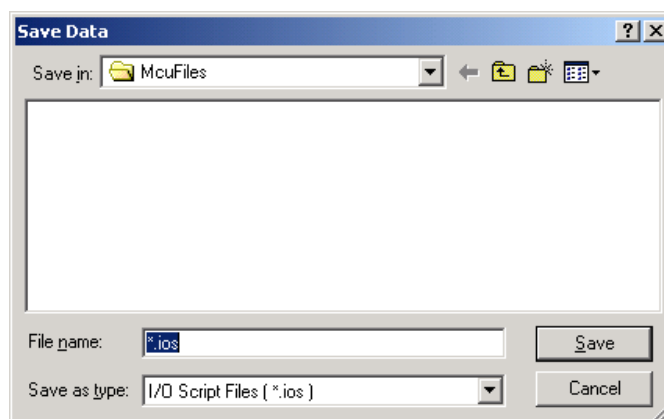


Figure 6.23 Save Dialog Box

12. The I/O script file is as follows.

```
; IOSRIPT FILE FOR I/O WINDOW (SET ISREAD)
{
pass #isread:0x3f0 , 1
set [0x3f0] = 0x68
pass #isread:0x3f0 , 1
set [0x3f0] = 0x65
pass #isread:0x3f0 , 1
set [0x3f0] = 0x6c
pass #isread:0x3f0 , 1
set [0x3f0] = 0x6c
pass #isread:0x3f0 , 1
set [0x3f0] = 0x6f
pass #isread:0x3f0 , 1
set [0x3f0] = 0x21
pass #isread:0x3f0 , 1
set [0x3f0] = 0x21
pass #isread:0x3f0 , 1
set [0x3f0] = 0xa
pass #isread:0x3f0 , 1
set [0x3f0] = 0x0
pass #isread:0x3f0 , 1
set [0x3f0] = 0x0
}
```

13. The following code is an example of virtual port input.

```
#include <stdio.h>
char *PORT_IN;
int i;
char buf[10];

void main(void)
{
    PORT_IN = (char *)0x3f0;           → Address specified in [Input address]
                                       or [Read address]
    for(i=0; i<10; i++){              → for statement
        buf[i] = *PORT_IN;           → Next piece of data referenced
                                       when read occurs
        if(buf[i] == '\0')           → 0x00 is used as the end code
            break;
    }

    printf("%s", buf);                → Outputs the input data to the standard output
}
}
```

6.2.3 Entering Data from a Virtual Port at a Specified Cycle

■ Description:

Data entry from a virtual port input at a specified cycle can be set up in the [I/O Timing Setting] window.

■ How to set up:

1. The data settings dialog box is displayed from the [I/O Timing Setting] window. Perform steps 1 to 3 in 6.2.2 Entering Data from a Virtual Port When a Specified Address Is Read.
2. Set the data input timing to [Cycle].
3. In the input address field, use a hexadecimal number to enter the address for which virtual port input is to be performed.
4. Set the start cycle and end cycle, and then click the [Next>] button.

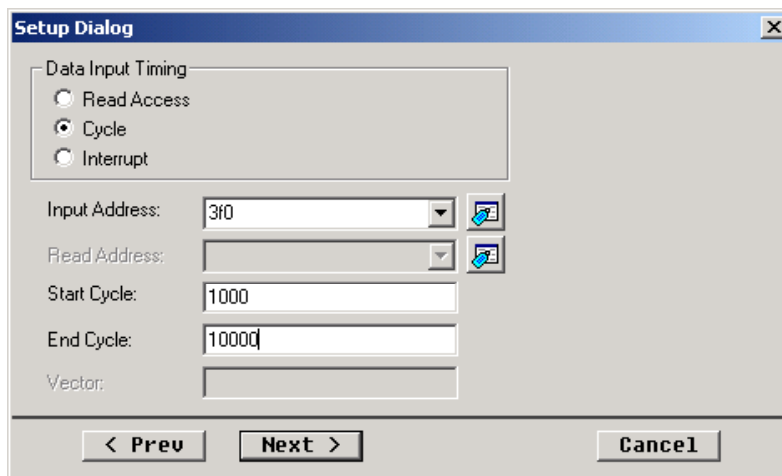


Figure 6.24 Cycle Settings

5. Set the data entered from the virtual input port. Position the mouse pointer at the cell of the desired read access occurrence count for setting the data, and double-click the left button to display the input console, and use a hexadecimal number to input the data.
6. Once the data is input, click the [Next>] button.

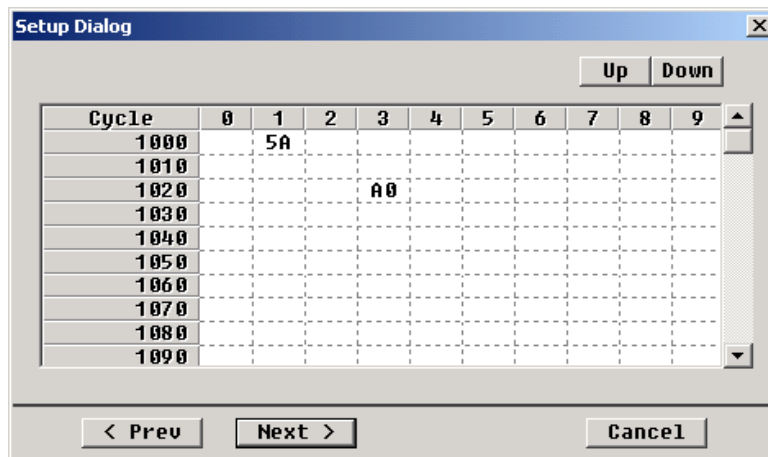


Figure 6.25 Input Data Settings

7. Save the I/O script file, as it is modified automatically.

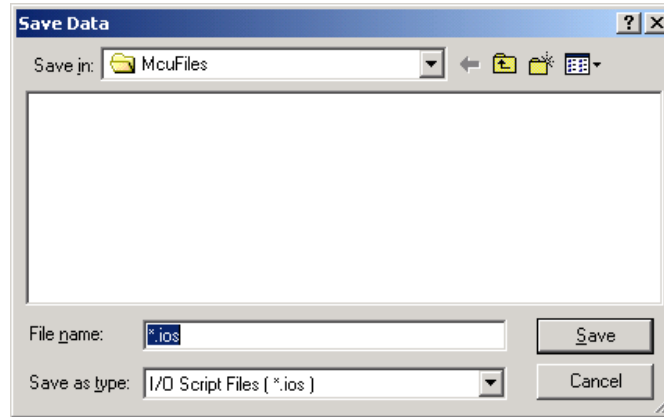


Figure 6.26 Save Dialog Box

The I/O script file is as follows.

```
; IOSCRIPT FILE FOR I/O WINDOW (SET WAITC)
{
cycle 1001
set [0x3f0] = 0x5a
waitc 22
set [0x3f0] = 0xa0
}
```

6.2.4 Entering Data from a Virtual Port When a Virtual Interrupt Occurs

■ Description:

A port input that occurs with a virtual interrupt can be set up in the [I/O Timing Setting] window.

■ How to set up:

1. The data settings dialog box is displayed from the [I/O Timing Setting] window. Perform steps 1 to 3 in 6.2.2 Entering Data from a Virtual Port When a Specified Address Is Read.
2. Set the data input timing to [Interrupt].
3. In the [Input address] field, use a hexadecimal number to enter the address for which virtual port input is to be performed.
4. Specify the vector of the interrupt to be monitored, and click the [Next>] button.

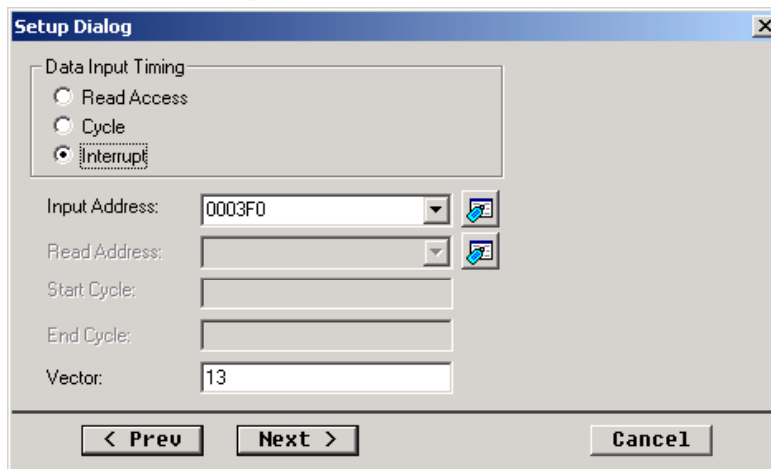


Figure 6.27 Interrupt Settings

5. Set the data entered from the virtual input port. Position the mouse pointer of the cell of the desired read access occurrence count for setting the data, and double-click the left button to display the input console, and use a hexadecimal number to input the data.
6. Once the data is input, click the [Next>] button.

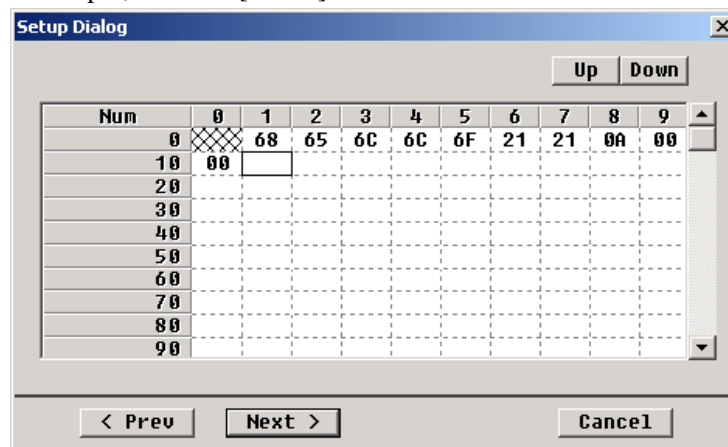


Figure 6.28 Input Data Settings

Save the I/O script file, as it is modified automatically.

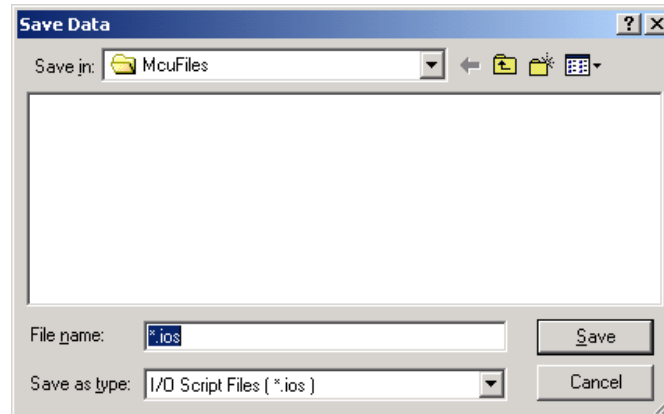


Figure 6.29 Save Dialog Box

7. The I/O script file is as follows.

```
; IOSCRIPT FILE FOR I/O WINDOW (SET ISINT)
{
pass #isint:13 , 1
set [0x3f0] = 0x68
pass #isint:13 , 1
set [0x3f0] = 0x65
pass #isint:13 , 1
set [0x3f0] = 0x6c
pass #isint:13 , 1
set [0x3f0] = 0x6c
pass #isint:13 , 1
set [0x3f0] = 0x6f
pass #isint:13 , 1
set [0x3f0] = 0x21
pass #isint:13 , 1
set [0x3f0] = 0x21
pass #isint:13 , 1
set [0x3f0] = 0xa
pass #isint:13 , 1
set [0x3f0] = 0x0
pass #isint:13 , 1
set [0x3f0] = 0x0
}
```

6.2.5 Checking Data Output to a Virtual Output Port

■ Description:

Data output to a virtual output port can be checked from the [I/O Timing Setting] window and Output Port window.

- (1) [I/O Timing Setting] window
 1. From the menu, choose [View -> CPU -> I/O Timing Setting] to display the [I/O Timing Setting] window.
 2. Either click the data settings icon, or right-click and choose [Setup] from the menu displayed.

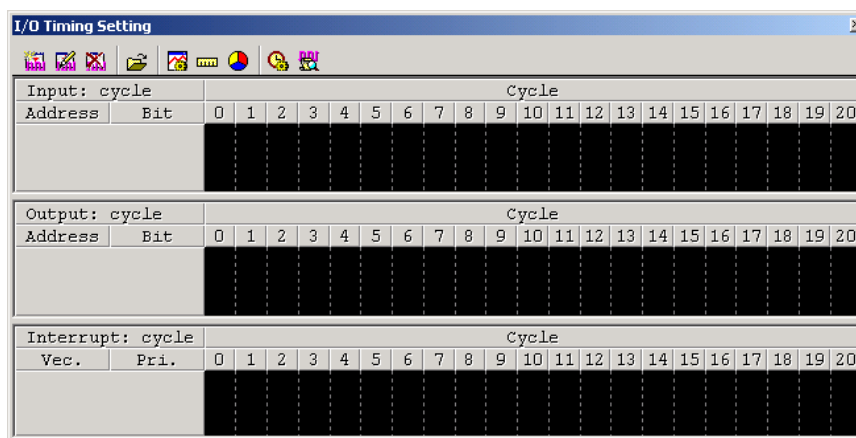


Figure 6.30 I/O Timing Setting Window

3. Select [Set a virtual port output], and click the [Next>] button.

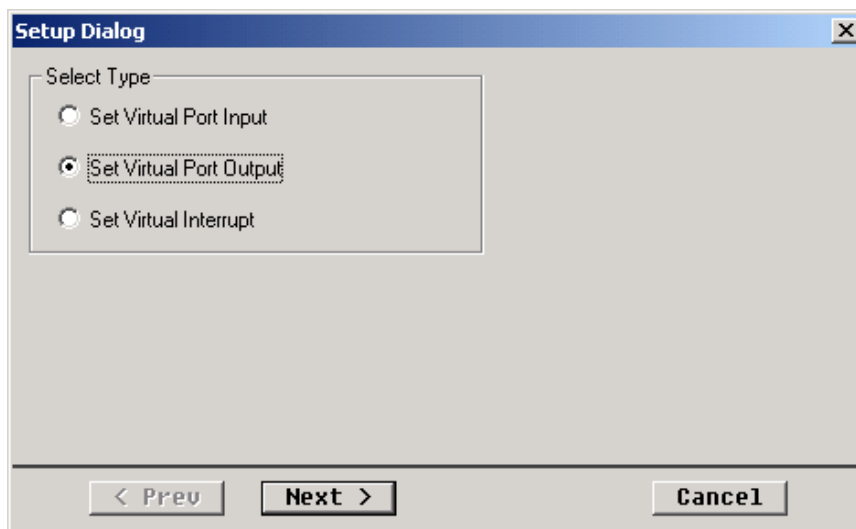


Figure 6.31 Selecting the Processing Type

4. Set the output address, and click the [Next>] button.

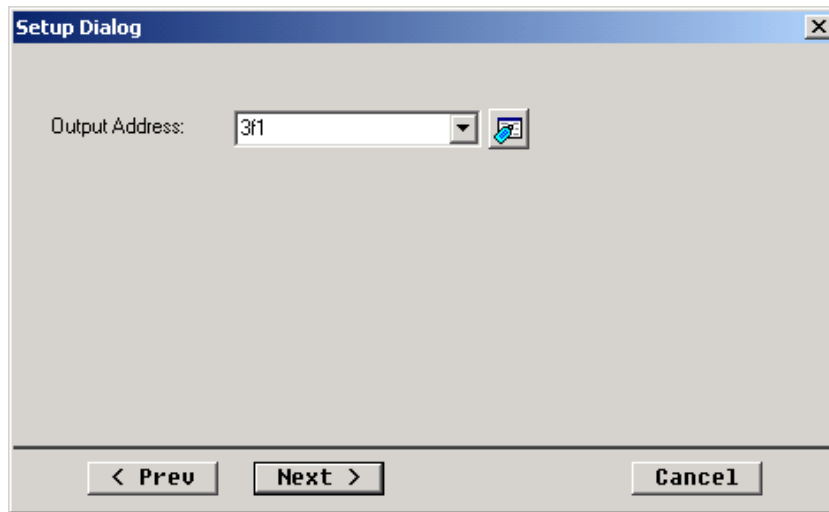


Figure 6.32 Setting the Output Address

5. A dialog box is displayed, allowing you to specify the I/O script file to which the virtual port output results are to be saved. Specify a file name, and save the file.

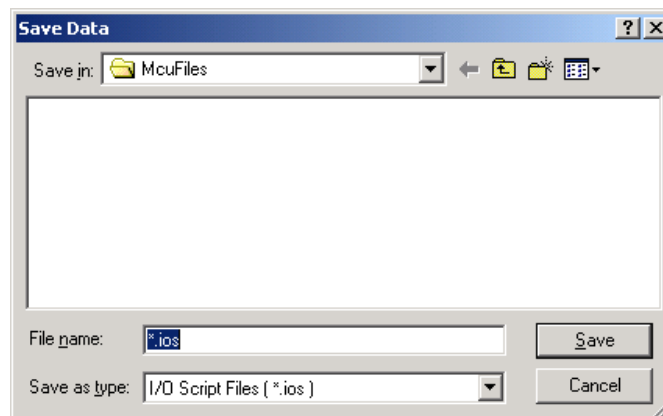


Figure 6.33 Save Dialog Box

```

#include <stdio.h>
char *PORT_OUT;
static int i;
char buf[10];

void main(void)
{
    PORT_OUT = (char *)0x3f1;           → Address specified for the output address

    sprintf(buf, "hello!!");          → Data output

    for(i = 0; i < 10; i++) {         → for statement
        *PORT_OUT = buf[i];           → Output is performed for the port
        if(buf[i] == '\0')            → 0x00 is used as the end code
            break;
    }
}

```

7. The [I/O Timing Setting] window for the output results is as follows.

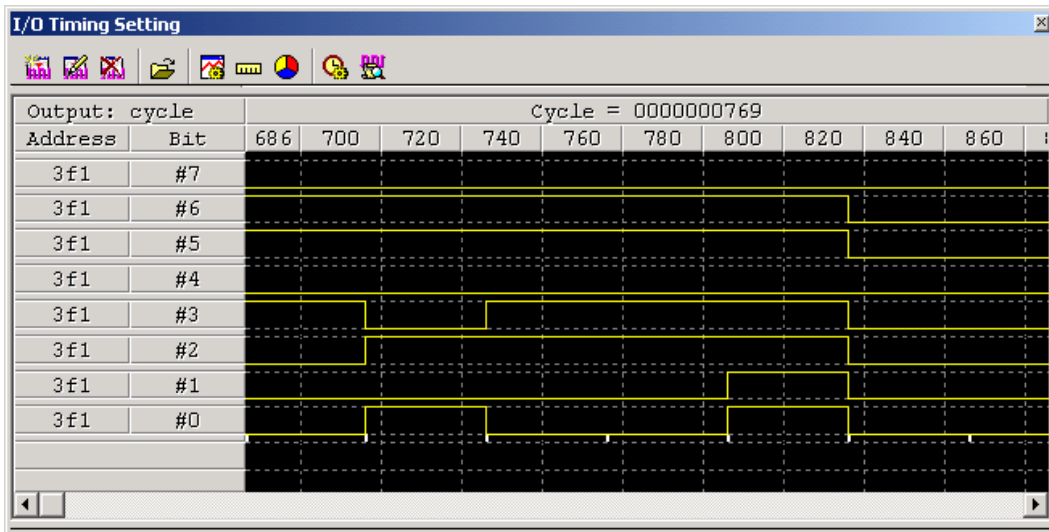


Figure 6.34 I/O Timing Setting Window

8. When the code in step 6 is executed, the I/O script specified in step 5 is as follows.

```

; IOSCRIPT FILE FOR I/O WINDOW (SET WAITC)
{
waitc 1145
set [0x3f1] = 0x68
waitc 30
set [0x3f1] = 0x65
waitc 30
set [0x3f1] = 0x6c
waitc 30
set [0x3f1] = 0x6c
waitc 30
set [0x3f1] = 0x6f
waitc 30
set [0x3f1] = 0x21
waitc 30
set [0x3f1] = 0x21
waitc 30
set [0x3f1] = 0x0
}

```

(2) Output Port window

1. From the menu, choose [View -> CPU -> Output Port] to display the [Output Port] window.
2. Either click the port settings icon, or right-click and choose [Set...] from the menu displayed.

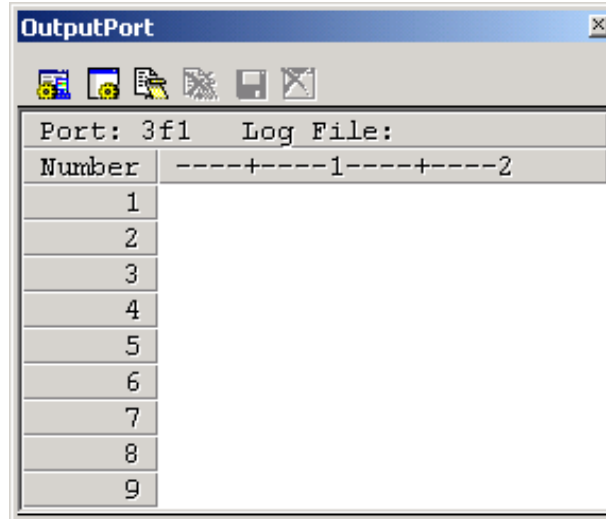


Figure 6.35 Output Port Window

3. The port settings dialog box is displayed.
4. Select [Address], and enter a label or address.
5. Click the [OK] button to complete the settings.

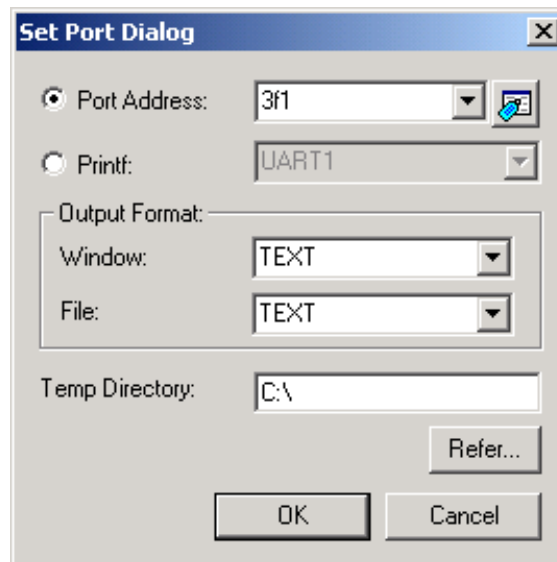


Figure 6.36 Port Settings

- When output is performed to the specified address while the program is running, the contents output to the port are displayed in the [Output Port] window.

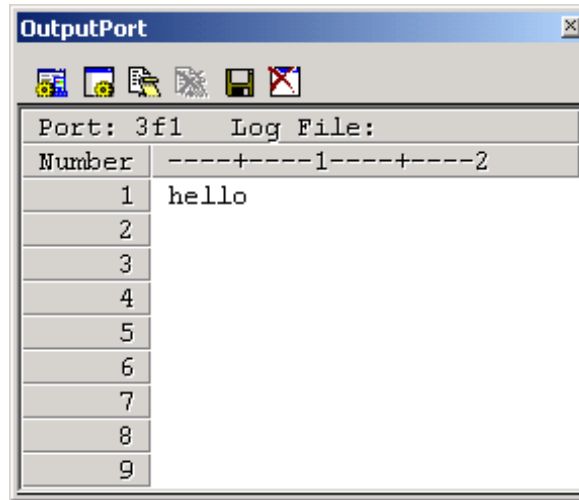


Figure 6.37 Output Port Window

- Click the log start icon to output also to a log file the contents output to the port.

6.3 Using a Virtual LED or Label to Check the Memory Contents

■ Description:

The color of a virtual LED or displayed text of a label can be changed in real-time, according to the contents of the memory being monitored.

■ How to check the memory contents by using a virtual LED or label:

1. From the menu, choose [View -> Graphics -> GUI I/O] to display the GUI window.
2. Either click the LED creation icon or label creation icon, or right-click and choose [Create LED] or [Create Label] from the menu displayed. Then, place the virtual LED or label.

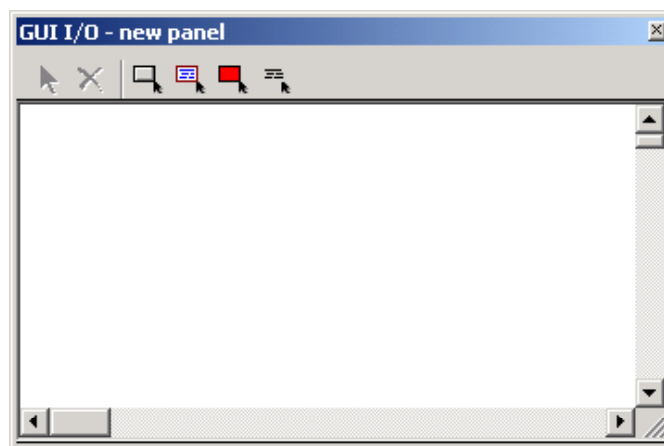


Figure 6.38 GUI Window

3. Click the placed object to display the settings window.

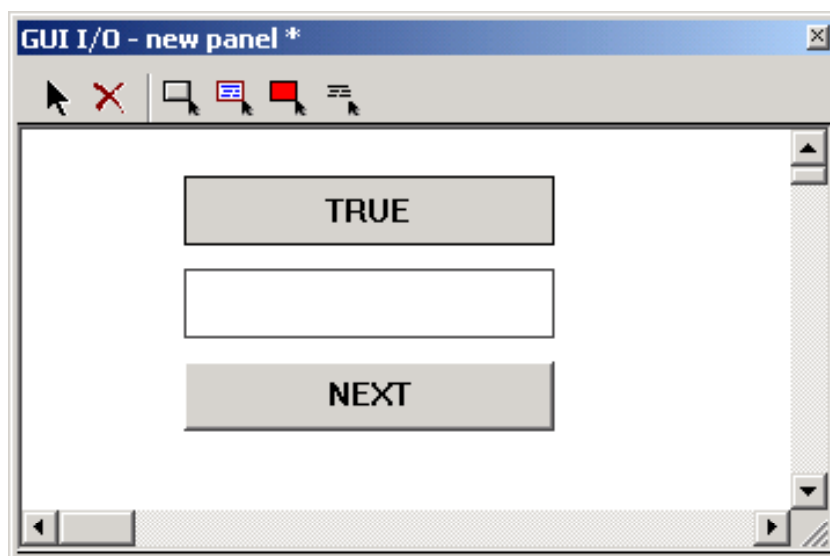


Figure 6.39 GUI Window (After Placement)

4. Set the address, and select either [Bit] or [Data] for the data type.
5. For a virtual LED, set the display color. For a label, set the displayed string.
6. For the [Bit] data type, select either [Positive] (display color 1 or displayed string 1 is used when the condition evaluates to positive) or [Negative] (display color 1 or displayed string 1 is used when the condition evaluates to negative).

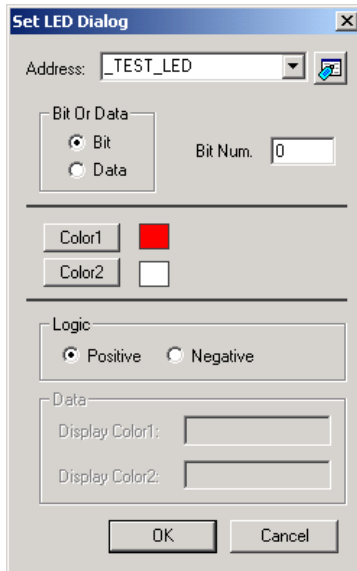


Figure 6.40 Set LED Dialog

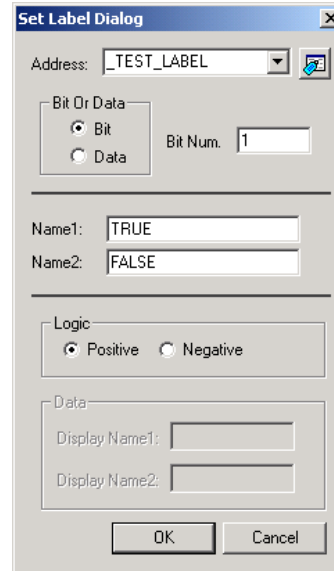


Figure 6.41 Set Label Dialog

7. The following is a code example. In this case, the virtual LED and label change by referencing the 0 bit and 1 bit of an integer from 0 to 200.

```
void main(void)
{
    for ( i = 0; i < 200; i++){
        TEST_LABEL = i;
        TEST_LED = i;
        while ( NEXT != 1);
        NEXT = 0;
    }
}
```

8. For the [Data] data type, enter the data corresponding to Display 1 and Display 2.

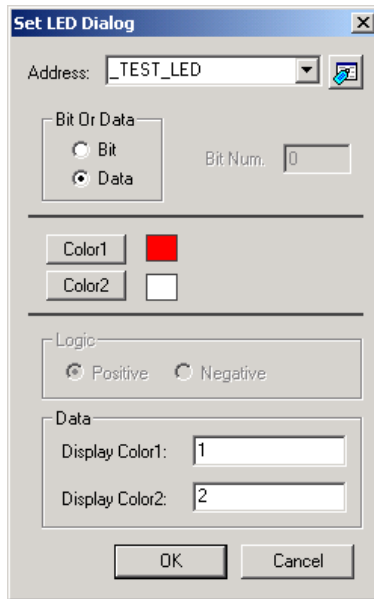


Figure 6.42 Set LED Dialog

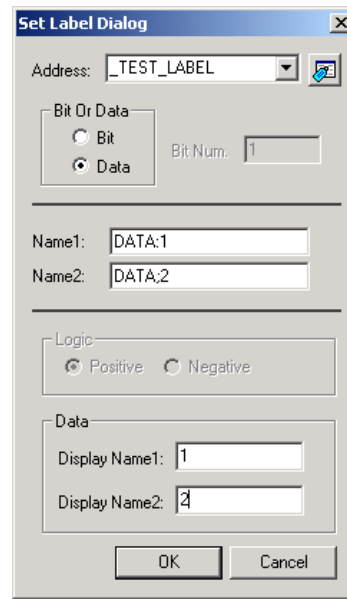


Figure 6.43 Set Label Dialog

9. When the following code is executed with the settings in step 8, the operations are the same as those in step 7.

```
void main(void)
{
    for ( i = 0; i < 200; i++){
        TEST_LABEL = ((i >> 1) & 1) ? 1 : 2;
        TEST_LED = ( i & 1) ? 1 : 2;
        while ( NEXT != 1);
        NEXT = 0;
    }
}
```

6.4 Using printf for Debugging

- Description:

The contents output via printf in the source code can also be output to the [Output Port] window and log file.

- How to output:

1. From the menu, choose [View -> CPU -> Output Port] to display the [Output Port] window.
2. Either click the port settings icon, or right-click and choose [Set...] from the menu displayed.

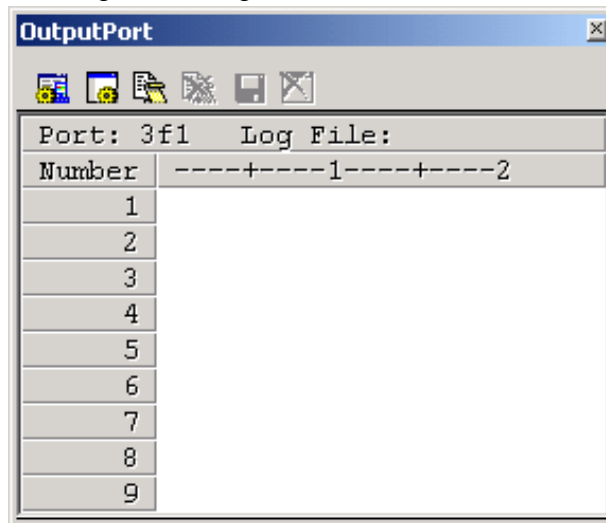


Figure 6.44 Output Port Window

3. The port settings dialog box is displayed.
4. Select [printf], and then [UART1].
5. Click the [OK] button to complete the settings.

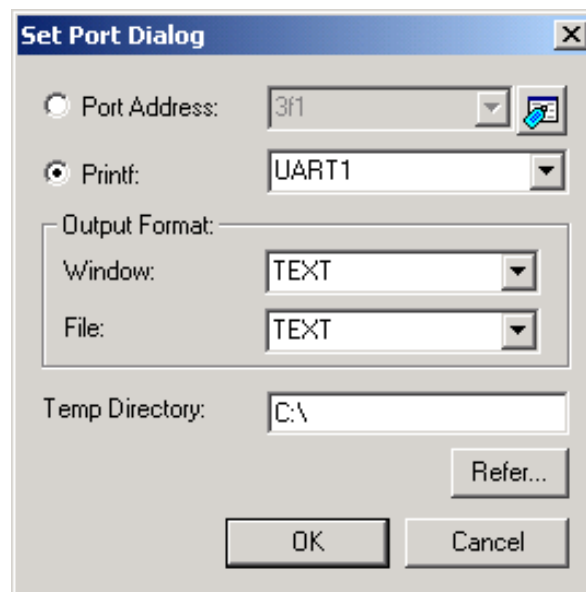


Figure 6.45 Port Settings

6. When printf is used for output while the program is running, the contents output to the port are displayed in the [Output Port] window. The output for when the following code is executed is shown below.

```

for(i = 0; i < 10; i++) {
    :
#ifdef DEBUG
    printf ( "i=%d\r\n",i);
#endif
    :
}
    
```

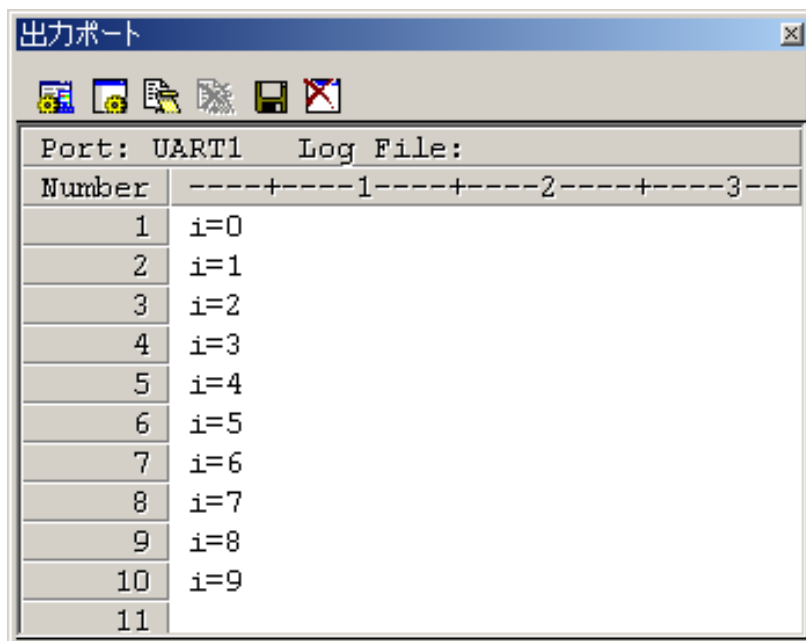


Figure 6.46 Output Port Window

7. Click the log start icon to output also to a log file the contents output to the port.

6.5 Using I/O Scripts

■ Description:

Settings for virtual port input and virtual interrupts can be specified in script format, in files. Such scripts are called *I/O scripts*, and the files in which they exist are called *I/O script files*. Although I/O script files can be created automatically using the [I/O Timing Setting] window, they can also be edited directly, for even more flexibility in settings. For example, the following kinds of settings can be performed outside of the [I/O Timing Setting] window.

- If you want to generate a cyclic virtual interrupt like timer interrupts, you can use the while statement to specify a repetition of virtual interrupt generation
- You can specify that the priority levels set in the interrupt control register's interrupt priority level select bits be referenced to resolve the interrupt priority of virtual interrupts generated.
- As conditions for entering virtual port inputs or generating virtual interrupts, you can specify a combination of program fetch, memory access for read/write, or memory comparison.

■ Method for using an I/O script file.

1. Use a text editor to create an I/O script file in advance. Set the file extension to ios.
2. In the [I/O Timing Setting] window, either click the import icon, or right-click and choose [Load] from the menu displayed to display the dialog box for importing I/O script files.
3. Select the I/O script file to be imported.

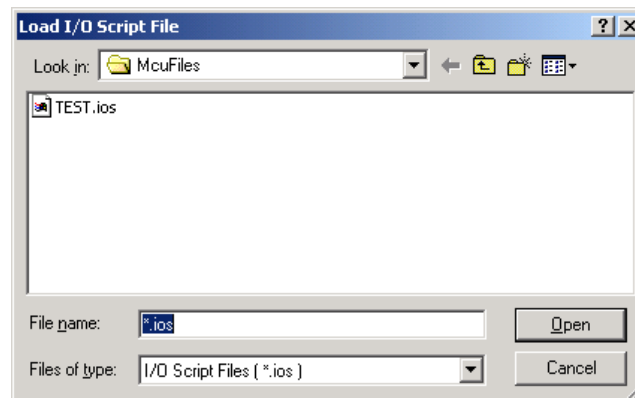


Figure 6.47 Load I/O Script File Dialog Box

■ Example definition for the timer A0 timer mode

The procedure in the following example is an example definition of the timer A0 timer mode.

In this example, the timer A0 interrupt occurs every divide-by-ratio (number of cycles) specified for timer A0. The value specified in the interrupt control register is referenced to determine the priority of this timer interrupt.

```

; Virtual interrupt example
{
while(1){
    if([0x380].b & 0x01) {
        waitc[0x386].w+1
        int 21,[0x55] & 0x7
    } else {
        waiti 100
    }
}

```

→ Procedure start
→ while statement
→ Checks timer A0's count start flag
→ Waits before I/O script execution, for the number of cycles for the frequency set for timer A0
→ Makes timer A0 interrupt occur (The interrupt control register is referenced to determine priority)
→ Waits before I/O script execution, for 100 instructions

■ Example definition for a virtual port input in sync with a cycle

The procedure in the following example is an example definition for a virtual port input in sync with a cycle.

In this example, virtual port input is performed when 5000 cycles are executed in the program. The [I/O Timing Setting] window can only be used to set virtual port input on a byte basis, but the I/O script file can be used to set virtual port input on a word or long word basis.

```

; Virtual port input example
{
    waitc 5000
    set[0x3e0] = 0x34
    waitc 5000
    set[0x3e0].w = 0x4126
}

```

→ Procedure start
→ Waits before I/O script execution, for 5000 cycles
→ Inputs 0x34 to the 0x3e0 address
→ Inputs 2-byte data 0x4126 from the 0x3e0 address
→ Procedure end

Section 7. MISRA C

7.1 MISRA C

7.1.1 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.

7.1.2 Rule Examples

This subsection introduces some actual MISRA C rules. Figure 7.1 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) {
    :
    default: ← Misspelled
        err = 1;
        break;
}
```

Figure 7.1 Rule 62

Figure 7.2 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 7.2 Rule 46

Figure 7.3 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 7.3, 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 7.3 Rule 38

Figure 7.4 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*)(0xffffffffeUL + 2); ← Results in a 0 address.

```

Figure 7.4 Rule 51

7.1.3 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 7.1 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 7.1 Compliance Matrix

Rule number	Compiler	Tool 1	Tool 2	Review (visual)
1	Warning 347			
2		Violation 38		
3			Warning 97	
4				Pass
...

7.1.4 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.

7.1.5 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.

7.2 SQMlint

7.2.1 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 7.5. It in no way affects the code generated by the compiler.

Table 7.2 lists the rules supported by SQMlint.

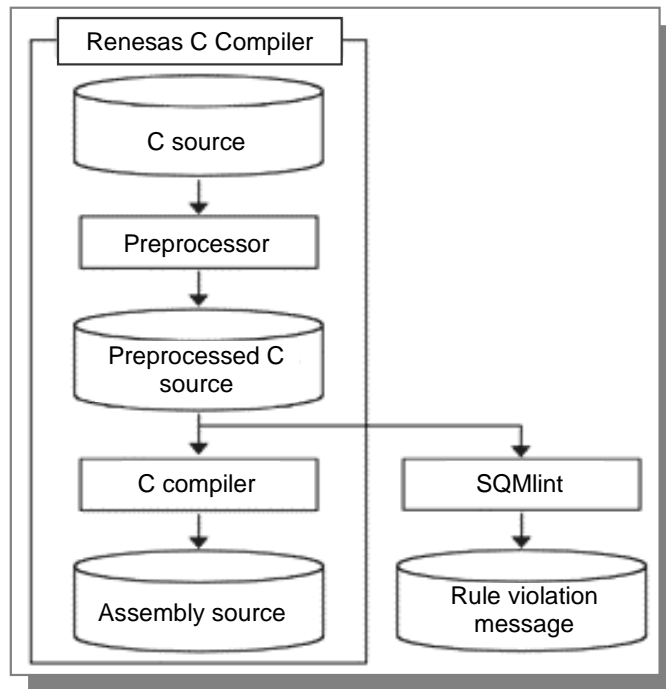


Figure 7.5 SQMlint Positioning

Table 7.2 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 7.3 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

7.2.2 Using SQMLint

SQMLint start options can be set easily from the window for setting the High-performance Embedded Workshop Compile Options. Figure 7.6 shows the dialog box for specifying the High-performance Embedded Workshop options, in which [MISRA C rule check] should be selected from [Category].

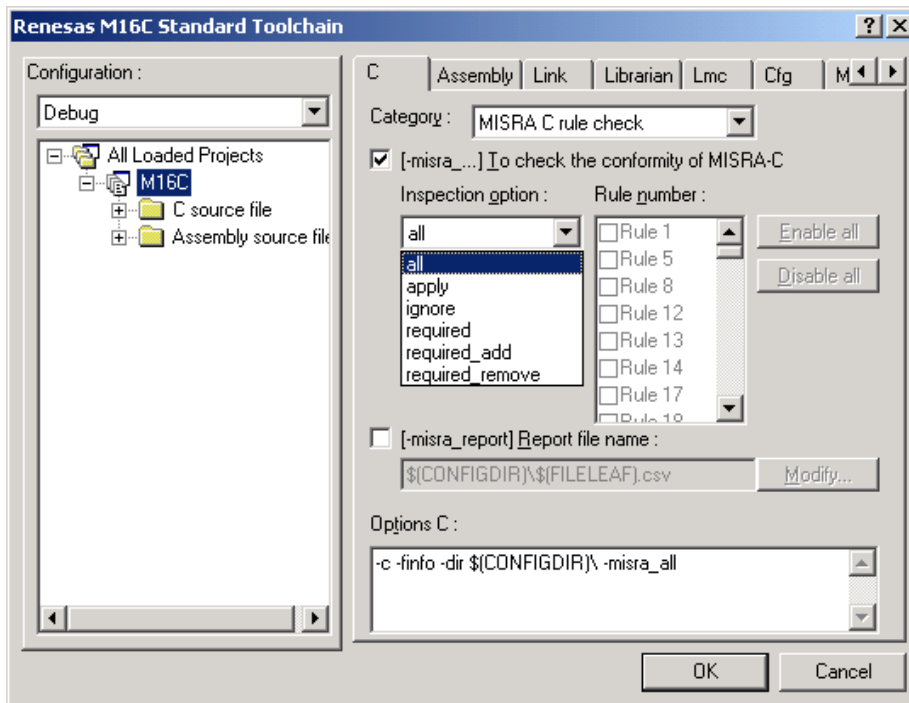


Figure 7.6 High-performance Embedded Workshop Options Window

Thus, SQMLint will start at compile-time. The following gives the meaning of each option:

- [all]: Performs testing for all rules.
- [apply]: Performs testing only for specified rules.
- [ignore]: Performs testing for all rules other than those of the specified numbers.
- [require]: Performs testing only for rules necessary according to the MISRA C rule.
- [require_add]: Performs testing for all rules necessary according to the MISRA C rule, as well as for those of subsequent numbers only.
- [require_remove]: Performs testing only for rules necessary according to the MISRA C rule, except for those of the specified numbers.

7.2.3 Viewing Test Results

Test results can be output in the following three ways:

(a) Standard error output

Messages are output the same as the High-performance Embedded Workshop compile errors. Tag jumping can be performed, allowing source code to be corrected easily using the same operations as for compile errors.

(b) CSV file

A file format that can be read by spreadsheet software, allowing reviews to be performed more easily.

(c) SQMmerger

Displays both the source file and test results, as shown in Figure 7.7.

```
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 7.7 SQMmerger

7.2.4 Development Procedures

Figure 7.8 shows how to perform development using SQMlint.

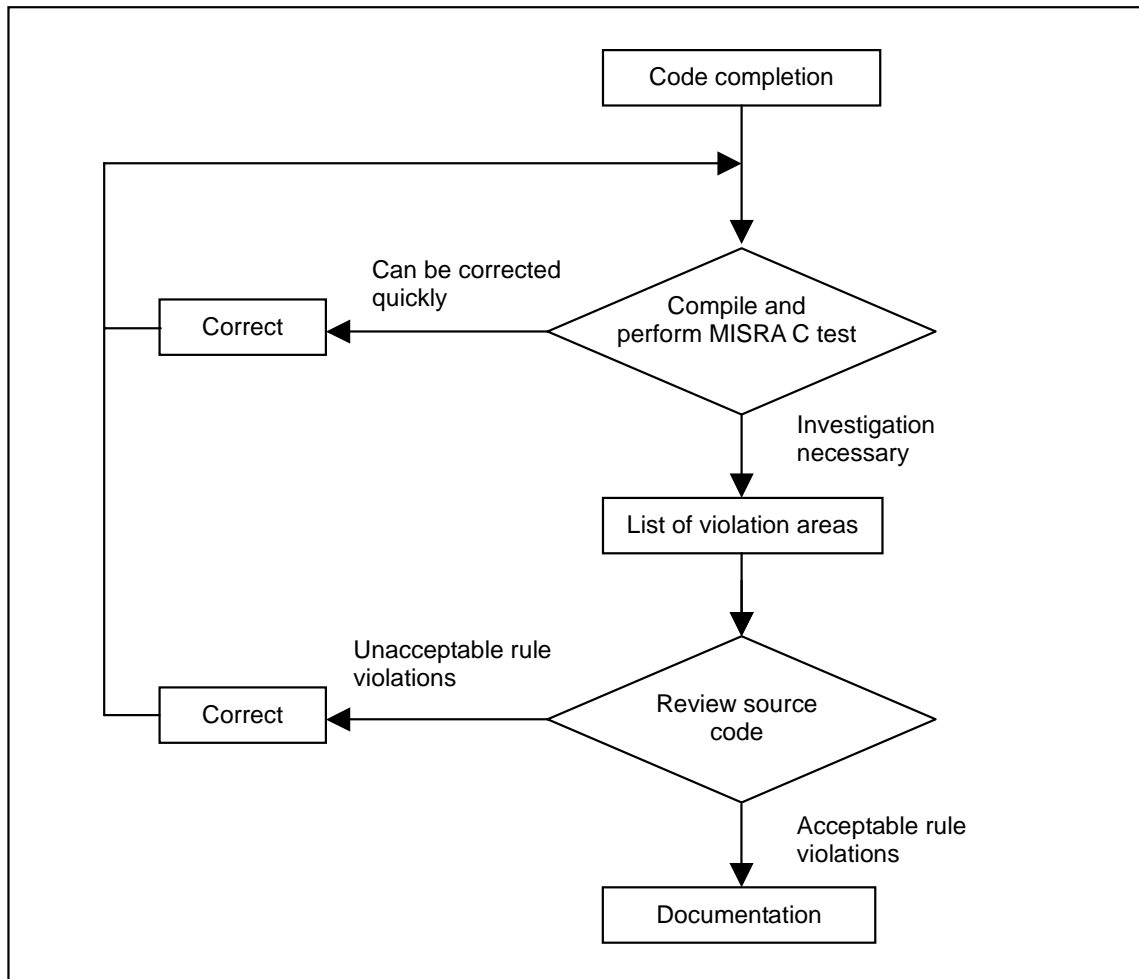


Figure 7.8 Development Procedure Using SQMlint

- (1) Collect all compile errors. SQMlint assumes that the C source code is valid.
- (2) Find errors detected by SQMlint.
- (3) Correct the errors that can be easily corrected.
- (4) Create a list of the locations of rule violations that require investigation, and perform a review.
- (5) Perform corrections for rules deemed unacceptable upon review.
- (6) Document rules deemed acceptable upon review, to leave a record.

7.2.5 Supported compilers

The following compilers are supported by SQMlint:

- M3T-NC30WA, Version 5.20 Release 1 and later
- M3T-NC308WA, Version 5.10 Release 1 and later
- M3T-CC32R, Version 4.10 Release 1 and later

Section 8. Frequently Asked Questions

This chapter contains answers to questions often asked by users.

8.1 C Compiler (M3T-NC308WA)

8.1.1 Bit Fields

■ Question:

How can I declare a bit field combining a 10-bit width, 11-bit width, and 3-bit width, without any empty bits (padding) between the fields?

■ Answer:

By declaring the entire sequence of fields as the same type, where the type contains more bits than the sum of the bits in the fields, you can create packed bit fields with no empty bits.

Example program:

```
typedef struct {
    unsigned long  BIT0_9    :10;
    unsigned long  BIT10_20 :11;
    unsigned long  BIT21_23 : 3;
} BIT0_23;
```

8.1.2 Memory Management Functions

■ Question:

How can I prevent functions such as `MALLOC()` and `MEMCPY()`, which are not needed for an application, from being linked in `NCxxWA`?

■ Answer:

The `MALLOC()` and `MEMCPY()` functions are used to secure the HEAP area set using the "nrt0.a30" and "sectxx.inc" startup programs. When not using memory management functions such as `MALLOC()` and `MEMCPY()`, specify the "-D__HEAP__=1" assembly option when assembling "nrt0.a30" and "sectxx.inc".

8.1.3 -ONBSD Option

■ Question:

What kind of optimization is suppressed by the compiler optimization option "-ONBSD"? Is it possible to perform more detailed settings for optimization suppressed by "-ONBSD"?

■ Answer:

The following kinds of optimization are suppressed:

- (1) Optimization in which common expressions are factored (so that the same expression is not performed multiple times)
- (2) Optimization in which common instruction blocks are grouped (such as for branches to identical series of instructions)
- (3) Optimization in which per-byte storage to consecutive areas is performed by word
- (4) Optimization in which consecutive push operations for 1-byte constants are performed by word
- (5) Optimization in which consecutive shifts are grouped (such as turning $b=a\ll 2$; $b\ll =2$; into $b=a\ll 4$;))
- (6) Optimization in which bit operations are grouped into OR and AND calculations
- (7) Optimization in which comparison operators in the for statements are moved to the end of the loop to reduce the number of times evaluation is performed)
- (8) Optimization in which condition-less branches to return statements are replaced with return (for NC308 only)
- (9) Optimization in which exponential calculations using an exponent of 2 are changed to shift operations
- (10) Optimization in which auto variables are automatically allocated to the register
- (11) Optimization in which constants are folded in
 - Note that these optimizations are not always applied, based on the code immediately before or after the corresponding location.

There are no options to perform more detailed settings for optimization suppression.

8.1.4 Priority of Optimization Options

■ Question:

When multiple optimization options are specified, with what priority are the options processed?
Also, which takes priority: options that perform optimization, or options that suppress it?

■ Answer:

When multiple optimization options are specified, those that take effect are as follows:

- (1) When multiple "-O1" to "-O5" options are specified, the last one specified takes effect.
- (2) When an "-Onumber" option and the "-OR" option are specified, both take effect.
- (3) When an "-Onumber" option and the "-OS" option are specified, both take effect.

- Note that the "-OR" and "-OS" options cannot be specified together.

For example, if both "-OR" and "-O1" are specified, -OR optimization is performed, but optimization suppressed by -O1 is not.

When both options that perform optimization and options that suppress optimization are specified, the latter take priority, and optimization is performed based on the specified optimization suppression options.

8.1.5 Adding Functions to the Library

■ Question:

How can I add a function to the compiler library?

■ Answer:

You can add a function to the compiler library as shown in the following example, which is based on NC308WA.

Compile the C source file to generate a relocatable file.

To generate "new.r30":

```
>nc308 -c new.c
```

Use librarian lb308 to add the relocatable file to the library.

To add the "new.r30" file to the "nc308" library file:

```
>lb308 -a nc308lib.lib new.r30
```

Note:

For details about how to use lb308, see the documentation for AS308, which can be found in the nc308wa/manual directory, on the product CD-ROM. For Windows, this documentation is also installed during product installation, so you can also find it from the Windows [Start] menu.

8.1.6 Placing const Declarations in the ROM Section

■ Question:

I thought that anything declared using const was placed in the ROM section, but with the following codes, it is placed in the data section. What should I do to have it placed in the ROM section?

```
const S_TBL *sp_tbl[]={
    p00, /* pointer to the structure */
    p01, /* pointer to the structure */
    p02, /* pointer to the structure */
};
```

■ Answer:

Code such as the above can be changed as follows to place sp_tbl in ROM:

```
S_TBL *const sp_tbl[]={
    p00, /* pointer to the structure */
    p01, /* pointer to the structure */
    p02, /* pointer to the structure */
};
```

Also, you can use the following to place p00, p01, and p02 in ROM, in addition to sp_tbl:

```
const S_TBL *const sp_tbl[]={
    p00, /* pointer to the structure */
    p01, /* pointer to the structure */
    p02, /* pointer to the structure */
};
```

Note:

When using the const declaration for a pointer, what is placed in ROM differs depending on the const position.

For the following example, a, b, and c are placed in ROM:

```
const int *i={a,b,c};
```

For the following example, i is placed in ROM:

```
int * const i={a,b,c};
```

8.1.7 Passing Parameters via Registers

■ Question:

When are function parameters passed via registers?

Also, what is the difference in size and speed when function parameters are passed via registers, as opposed to being stack-passed?

■ Answer:

Function parameters are passed via registers under the following conditions:

- (1) A prototype declaration exists for the function, and the parameter type is stated at the time of the function call.
- (2) The prototype declaration does not use the ... variable parameter.
- (3) The types of function parameters match that shown in the following table:

For NC30:

Parameter	Parameter type	Register used
First parameter	char type	R1L register
First parameter	int type or near pointer type	R1 register
Second parameter	int type or near pointer type	R2 register

For NC308:

Parameter	Parameter type	Register used
First parameter	char type	R0L register
First parameter	int type or near pointer type	R0 register

Also, passing parameters via registers is better in terms of both size and speed.

8.1.8 How Function Parameters are Passed

■ Question:

Does the way in which function parameters are passed affect program portability?

■ Answer:

If a prototype declaration exists for the function, the compiler decides how function parameters are passed. As such, the portability of the program is not affected.

8.1.9 Prototype Declarations

■ Question:

Does the prototype declaration decide how function parameters are passed?

■ Answer:

Prototype declarations not only decide how function parameters are passed, but also specify such properties as the parameter size. As a result, when a function is called for which no prototype declaration exists, there may be consistency issues with parameters, for functions existing in a separate file. To prevent such a problem, we recommend using the prototype declaration.

8.1.10 Member Placement in a Structured Bit Field

■ Question:

I have defined a structured bit field in a C program, but when I compile the program, the order of the bit field members is changed. How can I prevent this?

■ Answer:

The placement of members in a structured bit field is determined as follows:

- (1) "unpack" and "arrange" are not used in #pragma STRUCT^{#1}.
- (2) If consecutive bit fields of the same type exist, the members are placed consecutively.
- (3) If bit fields of different types exist, and there is a previous bit field of the same type, the members are placed in succession next to this previous bit field.
- (4) If no previous bit field of the same type exists, the members are placed from the next address.
- (5) Members are not placed consecutively across non-bit-field members.

To place members that meet the above conditions in the order in which they are specified, use one of the following countermeasures:

Countermeasure 1: Enclose each bit field of a different type in its own struct { }.

Countermeasure 2: Use the same type when declaring fields that you would like to place consecutively.

#1 This cannot be changed using options or #pragma STRUCT.

The following is an example in which the order of members is changed, and examples for each of the above countermeasures.

Example in which the member order is changed:

```
struct tagData {
    unsigned char    c0a : 5; /* (1) */
    unsigned char    c0b : 3; /* (2) */
    unsigned char    c1a : 6; /* (3) */
    unsigned char    c1b : 2; /* (4) */
    unsigned int     i2a : 10; /* (5) */
    unsigned int     i2b : 6; /* (6) */
    unsigned char    c4a : 3; /* (7) */
    unsigned char    c4b : 5; /* (8) */
    unsigned char    c5; /* (9) */
    unsigned char    c6a : 5; /* (10) */
    unsigned char    c6b : 3; /* (11) */
} s;
```

For the above code, members are placed as follows:

(1), (2), (3), and (4) are placed consecutively.

Since the type differs for (5) and (6), members of the same type, (7) and (8), are placed first.

Since (9) is a non-bit-field member, (10) and (11) cannot continue from (7) and (8).

(5) and (6) are placed from the address after (7) and (8), as their type is different.

(9) is placed from the address after (5) and (6), as it is a non-bit-field member.

(10) and (11) are placed from the address after (9).

Countermeasure 1: Enclose each bit field of a different type in its own struct { }.

```
struct tagData {
    struct {
        unsigned char    c0a : 5; /* (1) */
        unsigned char    c0b : 3; /* (2) */
        unsigned char    c1a : 6; /* (3) */
        unsigned char    c1b : 2; /* (4) */
    } s1;
    struct {
        unsigned int     i2a : 10; /* (5) */
        unsigned int     i2b : 6; /* (6) */
    } s2;
    struct {
        unsigned char    c4a : 3; /* (7) */
        unsigned char    c4b : 5; /* (8) */
    } s3;
    unsigned char    c5; /* (9) */
    struct {
        unsigned char    c6a : 5; /* (10) */
        unsigned char    c6b : 3; /* (11) */
    } s4;
} s;
```

- Note that areas referencing this structure need to be changed.

Before: s.c0a = 1;

After: s.s1.c0a = 1;

Countermeasure 2: Use the same type when declaring fields that you would like to place consecutively.

```
struct tagData {
    unsigned int    c0a : 5; /* (1) */
    unsigned int    c0b : 3; /* (2) */
    unsigned int    c1a : 6; /* (3) */
    unsigned int    c1b : 2; /* (4) */
    unsigned int    i2a : 10; /* (5) */
    unsigned int    i2b : 6; /* (6) */
    unsigned int    c4a : 3; /* (7) */
    unsigned int    c4b : 5; /* (8) */
    unsigned char   c5; /* (9) */
    unsigned char   c6a : 5; /* (10) */
    unsigned char   c6b : 3; /* (11) */
} s;
```

Members (1) to (8) and (10) to (11) are placed consecutively.

Using this method may increase the amount of code slightly.

8.1.11 Increment and Decrement Operators

■ Question:

I have programmed the C code shown in (1).

When looking at the generated code, it appears that the variable x is compared to 5 before it is incremented. To have the variable x compared to 5 after it is incremented, do I need to change this code to something like that shown in (2)?

```
(1)  if (x++ == 5) {
      aaasub();
    }
(2)  x++;
      if (x == 5) {
          aaasub();
      }
```

■ Answer:

There are two ways to use the ++ increment operator and -- decrement operator: before the variable, and after the variable.

After the variable: increment/decrement is performed after the variable is used.

Before the variable: increment/decrement is performed before the variable is used.

Since the increment operator used in your program is placed after the variable, the variable x is compared to 5 before increment is performed. To achieve the expected results, place the increment operator before the variable as follows:

```
if (++x == 5) {
    aaasub();
}
```

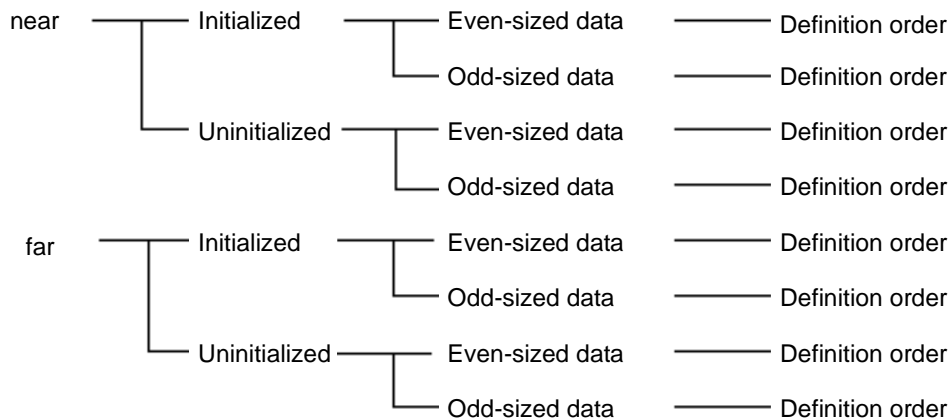
8.1.12 Placing External Variables

■ Question:

When writing a C program, how can I have external variables placed in the order in which they are defined?

■ Answer:

With the standard NCxxWA, external variables are placed as grouped into each of the following attributes:



You cannot group initialized and uninitialized data together, but you can specify the "-fno_even" command option at compile time to group even-sized data and odd-sized data, placing external variables as follows:



8.1.13 Placing an Array in the far Area

■ Question:

What declaration in NC30WA should I use to place an array in the "far" area, but the pointer that references it in the "near" area?

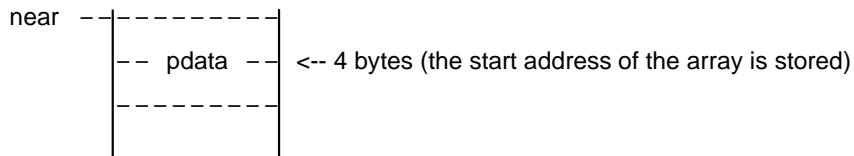
■ Answer:

As an example, use a declaration like the following to declare a 64-byte array of the short type:

```
/* (1) declaring the array itself */
short far data[64];

/* (2) declaring the pointer to the array */
short far *pdata;
```

The pointer in (2) is stored in the "near" area as shown below. This pointer points to the "far" area.



You can also change the area in which the pointer to the array is placed, and the area to which the pointer points, as shown below.

```
short * far pdata
```

In the above, pdata is placed in the "far" area, and that to which it points is placed in the "near" area. If "far" is omitted, this is handled as "near".

```
short far * far pdata
```

In the above, pdata is placed in the "far" area, and that to which it points is also placed in the "far" area.

```
short far * pdata
```

In the above, pdata is placed in the "near" area, and that to which it points is also placed in the "far" area.

```
short * pdata
```

In the above, pdata is placed in the "near" area, and that to which it points is also placed in the "near" area.

8.1.14 Placing a Function at a Fixed Address

■ Question:

How can I place the functions in a C program at an absolute (fixed) address?

■ Answer:

NC30WA places functions in the program section. As such, to place functions in an absolute address, specify the location of the program section.

Since addresses in the program section come after the .section program line in the "sect30.inc" file, they can be specified using the .org pseudo-instruction.

To place a function at the 10000H address, specify the following in the "sect30.inc" file:

```
.section program
.org 10000H
```

To place each function at a separate address, use #pragma SECTION for each function to create a section with a name other than that of the program section, and change the code so that the address of the section is specified.

To place the func() function in a section called program1, and place the section at 20000H:

(1) You can specify the name of the section in which to place the function as follows:

```
#pragma SECTION program program1
func()
{
}
```

(2) In the "sect30.inc" file, specify the section address as follows:

```
.section program1
.org 20000H
```

8.1.15 Specifying an Absolute Address Using #pragma ADDRESS

■ Question:

How can I use #pragma ADDRESS to specify the same absolute address as the following?

```
#define AAA (*(volatile unsigned char *)0x000406)
```

■ Answer:

To use #pragma ADDRESS to specify 0x0406 as shown above, use the following:

```
#pragma ADDRESS AAA 000406h
unsigned char AAA;
```

Note that the variables declared by #pragma ADDRESS have a volatile attribute without its specification.

8.1.16 Using #define to Define a String

■ Question:

When I use the preprocessing command #define to define a numerical value in a string, why do expressions using the defined string return unexpected results?

For example, in the following, the value of cul is not 0x04AAAA, as I would expect.

```
#define V1 0x040000
#define V2 0x0AAAA + V1

cul = (WORD *)V2
```

■ Answer:

For the above program, the expression for the results of the #define expansion are as follows (note that #define expansion involves only string replacement):

```
cul = ( WORD *)0x0AAAA+ 0x040000;
```

In this expression, (WORD *) is casted only to 0x0AAAA, to which 0x040000 is added. Since this ends up being added to the pointer to int, it becomes 0x80000.

To obtain the expected value, enclose in parentheses the expression defined by #define, as follows:

```
#define V2 (0x0AAAA + V1)
```

8.1.17 Types of Bit Field Members

■ Question:

When executing the following program, I thought that the long-type variable l would be substituted with -1, but instead it was replaced with 0xF (15). How can I set it to -1?

```
void main( void )
{
    struct
    {
        short r : 10 ;
        short i : 4 ;
        short s : 2 ;
    } buf ;
    long l ;

    buf.i = -1 ;
    l = (long)buf.i ;
}
```

■ Answer:

The bit field member is processed with unsigned as its type. As a result, variables with large types are zero-extended when stored or transferred.

As shown below, to handle such variables as sign extensions, declare the type of the bit field member explicitly, as "signed".

```
struct
{
    signed short r : 10 ;
    signed short i : 4 ;
    signed short s : 2 ;
} buf ;
```

8.1.18 Duplicate Variable Definitions

■ Question:

In a C program, if a variable of the same name and type is declared more than once in the same file, will an error occur?

■ Answer:

No error or warning will occur when a variable of the same name is declared more than once in the same file.

This is due to the adoption of the following ANSI specification:

Note:

"As long as the type of variable allocated to a given does not change within the same file, any number of external declarations can exist for that name."

No error occurs for the following:

```
int i;
int i=1;

main()
{
}
```

A duplicate definition error occurs for the following:

```
int i;
char i;

main()
{
}
```

8.1.19 Prototype Declarations for a Function

■ Question:

Why is the error message "*function-name*: value is undefined" output during linking, even though the function exists?

■ Answer:

This happens when the function name on either the side calling the function or the side being called is *_function-name*, while the other side is *\$function-name*. This is often due to the fact that no prototype declaration exists for the function, or the types of the parameters for the functions do not match. Check the prototype definition.

8.1.20 External References for Functions Without an extern Declaration

■ Question:

Does a compiler error occur when a function performs an external reference without performing an extern declaration?

■ Answer:

If no extern declaration (prototype declaration) exists, the ANSI specification dictates that the return value of the function is interpreted as an int value. As such, when you use the int type for an external reference, no error occurs.

To detect functions that have no extern declaration (prototype declaration), specify "-Wno_prototype (-WNP)" or "-Wall" at compile-time. This will output a warning for any function without a prototype declaration.

8.1.21 Code Deletion During Optimization

■ Question:

Why is no code output for expressions such as reading a port when an optimization option is used?

■ Answer:

This is because the code is deleted as meaningless during optimization. For operations that have a purpose such as reading, use the volatile declaration to declare any variables.

8.1.22 Consolidating Bit Access

■ Question:

Since consecutive bit access operations are consolidated into one operation, when an interrupt occurs during bit access, the program sometimes malfunctions. What should I do?

■ Answer:

Sometimes malfunction occurs because consecutive bit calculations are consolidated into one calculation, due to optimization. To avoid such malfunction, you can suppress this kind of optimization by specifying the "-ONB (-Ono_bit)" option at compile time.

8.1.23 Placing a Library Function at a ROM Address

■ Question:

How can I specify a ROM address at which to place a library function during linking? Also, is there a section for library functions so that I can specify section placement during linking?

■ Answer:

Unfortunately, there is no section for library functions.

To change the section of any library function, in the source program of the library function, specify the section name in the `#pragma SECTION` extension function. Both NC30WA and NC308WA come with the source for each library.

At the start of the library function, add the following `#pragma SECTION` line.

```
#pragma SECTION program library
```

The name of the section in which the library function will be placed is "library". You can specify the section name as you wish.

Then, after compiling and assembling the library function as you would for a regular function, place the "library" section at the intended address, and perform linking.

Notes:

#pragma SECTION cannot be used to change the name of a section for libraries in the assembler. Change .section program directly.

The source for library functions is stored in `src30¥lib(src308¥lib)`, in the NC30WA(NC308WA) installation directory, if selected as such during installation.

8.1.24 Processing for Negative Integer Calculations

■ Question:

Regarding how processing is performed for the following calculations:

- (1) In what direction are the results of negative integer division using the / operator rounded?
- (2) What is the sign of the result of modulo operation using the % operator?
- (3) When a negative integer is bit-shifted using the >> operator, which is performed: an arithmetic shift (where the MSB becomes the sign bit) or a logical shift (where the MSB becomes 0)?

■ Answer:

- (1) The results of negative integer division using the / operator are rounded to 0.

Expression	Result
$(-10) / 3$	-3
$(-10) \% 3$	-1
$10 / (-3)$	-3
$10 \% (-3)$	1
$(-10) / (-3)$	3
$(-10) \% (-3)$	-1

- (2) The sign of the result of modulo operation using the % operator is the sign of the dividend.

Expression	Result
$(-10) \% 3$	-1
$10 \% (-3)$	1
$(-10) \% (-3)$	-1

Note:

If the "-fround_under_div(-fRUD)" option is specified, the sign of the divisor is used.

- (3) When a negative integer is bit-shifted using the >> operator, an arithmetic shift is performed, where the MSB becomes the sign bit.

8.1.25 int Type Sizes

■ Question:

In the following program, why does the right-hand side of the `aaa = (signed long)(BBB << CCC);` expression yield `0x0000f000`? In VISUAL C++, it yields `0x0fff000`.

Example program	Generated assembly code
<pre>#define BBB (unsigned short)0xffff #define CCC (unsigned char)12 signed long aaa; test(void) { aaa = (signed long)(BBB << CCC); }</pre>	<pre>_test: mov.w #0f000H, _aaa mov.w #00000H, _aaa+2 rts</pre>

■ Answer:

The compiler processes the expressions in the above program in the following order:

- (1) BBB is converted to the int type.
(if the former value can be expressed as the int type, signed int is used)
(if the former value cannot be expressed as the int type, unsigned int is used)
- (2) Shift is performed with CCC.
- (3) The result is converted to the signed long type, and stored in aaa.

Since NC30 and NC308 handle conversion to the int type in 16 bits, the expression `aaa = (signed long)(BBB << CCC);` is processed as follows:

- (1) `0xffff -> 0xffff` (this cannot be expressed as the int type, and is converted to the unsigned int type)
- (2) `0xffff<<12 -> 0xf000`
- (3) Since `0xf000` has no sign, it remains `0x0000f000` even when casted to a signed long. -> aaa.

The result is `0x0000f000`.

In NC30 and NC308, to have the result of the expression `aaa = (signed long)(BBB << CCC);` be `0x0fff000`, specify the code to cast the BBB variable instead of casting the shifted results to the signed long type.

```
#define BBB (unsigned short)0xffff
#define CCC (unsigned char)12

signed long aaa;

test( void )
{
    aaa = (signed long)BBB<<CCC;
}
```

The following assembly code is generated when the above program is compiled.

```
;;# # C_SRC :      aaa = (signed long)BBB << CCC;  
      mov.w      #0f000H, _aaa  
      mov.w      #00fffH, _aaa+2
```

Note:

VISUAL C++ handles the int type as 32 bits.

As such, in VISUAL C++ the expression `aaa = (signed long)(BBB << CCC);` is processed as follows:

(1) `0xffff` -> `0x0000ffff`

(2) `0x0000ffff << 12` -> `0x0fff000`

(3) `0x0fff000` -> `aaa`

The result of this calculation is `0x0fff000`.

8.1.26 Controlling the enter Instruction

■ Question:

With NC308WA Version 3.00 Release 1, the enter instruction is output at the beginning of functions. But since this is not output with Version 3.10 Release 2, stack parameters cannot be referenced with inline assembly code. Is there any way of controlling the enter instruction?

■ Answer:

With NC308WA Version 3.10, optimization has been strengthened not to output unnecessary enter instructions. However, since processing within inline assembly code is not taken into account, when parameters and automatic variables are not referenced outside the range from #pragma ASM to #pragma ENDASM or other than asm(), the enter instructions are deemed unnecessary and not output.

NC308WA supports the use of asm("... \$\$ or \$@ ...", variable-name); to use inline assembly code to reference C variables.

Since this method allows you to properly check from the compiler whether the value is used, you can generate enter instructions as necessary. If you use another method to reference C variables, operation may become unpredictable, or compatibility between versions may be lost. When using such a method to reference variables and parameters, do not set function return values.

We recommend that you use code such as that in the following examples.

Example 1:

```
void memset(char *p, char c, unsigned short n)
{
    asm(" PUSHM  A1,R0,R3  ");    // The registers used must be saved
    asm(" MOV.L  $$[FB],A1  ", d); // Parameter d is transferred to A1
    asm(" MOV.B  $$[FB],R0L ", c); // Parameter c is transferred to R0L
    asm(" MOV.W  $$[FB],R3  ", n); // Parameter n is transferred to R3
    asm(" SSTR.B                ");
    asm(" POPM   A1,R0,R3  ");    // The registers used are restored
}
```

Example 2:

```
void memset(char *p, char c, unsigned short n)
{
    asm(" PUSHM  A1,R0,R3  ");    // The registers used must be saved
    asm(" MOV.L  $@,A1     ", d); // Parameter d is transferred to A1
    asm(" MOV.B  $@,R0L    ", c); // Parameter c is transferred to R0L
    asm(" MOV.W  $@,R3     ", n); // Parameter n is transferred to R3
    asm(" SSTR.B                ");
    asm(" POPM   A1,R0,R3  ");    // The registers used are restored
}
```

For automatic variables, \$\$ is replaced with the offset from the FB register value. For external variables, symbols, or register variables, \$@ is replaced with the register name.

\$@ is replaced with an operand indicating an automatic variable, external variable, or register variable. The type of variable corresponding to \$@ is automatically determined by the compiler.

Optimization may change the function parameters and automatic variables to register variables, but not variables specified using asm("\$\$ or \$@", variable-name). As such, no situation exists where, for example, a \$\$ intended to be written for the offset from an automatic variable FB becomes a register variable, and is accidentally expanded to the register name.

8.1.27 Performance for the Floating-point Library

■ Question:

What can you tell me about performance for the floating-point library?

■ Answer:

Performance for the floating-point library is as follows:

Measurement conditions

- (1) Compiler used: M3T-NC30WA V.5.30 Release 02
- (2) Emulator used: Compact emulator for the M16C/Tiny series (M30290T2-CPE)

Clock 20 MHz High-speed mode

Measurement results

- Arithmetic operations

	float	double
Addition (7.1+2.9)	33 μ sec	37 μ sec
Subtraction (7.9-2.9)	35 μ sec	40 μ sec
Division (3.5/0.5)	79 μ sec	164 μ sec
Multiplication (7.5*10.2)	22 μ sec	37 μ sec

- Mathematical functions

acos(0.5)	9.718 msec
asin(0.5)	10.682 msec
atan(0.5)	7.984 msec
atan2(1.5,0.5)	13.119 msec
ceil(10.8)	0.104 msec
cos(0.5)	2.775 msec
cosh(0.5)	1.934 msec
exp(10.4)	1.533 msec
fabs(x)	0.010 msec
fmod(10.5,1.5)	0.218 msec
ldexp(10.4,3)	0.006 msec
log(2.4)	4.587 msec
log10(100.3)	5.295 msec
pow(10.3,4.5)	7.061 msec
sin(0.4)	3.047 msec
sinh(0.6)	1.938 msec
sqrt(100.1)	2.112 msec
tan(0.5)	2.717 msec
tanh(0.9)	1.966 msec

8.2 Linker

8.2.1 "-LOC" Option for ln308 and ln30

■ Question:

In what situations should I use the "-LOC" option for ln308 or ln30?

■ Answer:

This option can be used for applications in which the program runs in RAM. In this case, the RAM address of the program running in RAM is defined using the "-ORDER" option. Then, the ROM address registering the program transferred to RAM is defined using the "-LOC" option.

Note that the "-LOC" option functionality only registers the defined program at the specified address, and does not have any functionality to transfer the program to an address area during execution.

8.2.2 Warnings During Linking

■ Question:

The warning "16-bits unsigned value is out of range 0 -- 65535. address='xxxx'" that is output during linking, does not occur when the RAM size is decreased. What is wrong?

■ Answer:

The above warning is output when bit operations are output for areas not reachable by bit operations (outside of 0H to 1FFFH). The reason that decreasing the RAM size prevents the warning from being output is that variables in areas other than from 0H to 1FFFH are fit into the area from 0H to 1FFFH, and can be reached by bit operations. The address at which variables are placed cannot be determined at compile-time, as this is determined during linking.

Perform the following to remove settings for outputting bit operations:

- (1) If `-fbit` (or `-fB`) is specified during compile-time, clear this specification.
- (2) If the corresponding variable is specified in `#pragma BIT`, clear this specification.

8.2.3 Changing a Start Address

■ Question:

How can I convert a program for which the start address is coded as 0xFC0000 to the Motorola S format with a start address of 0x000000?

Also, why doesn't the start address change when I run lmc308 as follows?

```
>lmc308 -E 00 test.x30
```

■ Answer:

lmc308 cannot change start addresses. The "-E" command option is for registering an execution start address, which is registered in digits 5 to 8 of the S8 record on the last line of the Motorola S format. However, keep in mind that this option cannot change the start address.

8.3 Stk Viewer

8.3.1 Stk Viewer Stack Size

■ Question:

The stack size displayed in Stk Viewer was secured, but an overflow occurred nonetheless. Is the stack size displayed in Stk Viewer not sufficient?

■ Answer:

The stack size displayed in Stk Viewer is for reference only.

Stk, which is called by Stk Viewer, calculates the stack size based on stack information in the absolute module files output by the compiler, but since this size is only theoretical, it is not the actual size as calculated from the tracing the program.

Also note that Stk does not take interrupt functions into account.

To calculate the stack size with interrupt functions taken into account, add the stack size of the functions or assembly routines for which interrupts occur as displayed in Stk Viewer, to the stack size of the interrupt function.

8.4 SQMLint

8.4.1 Selecting Test Rules

■ Question:

Is there a way to select only the necessary rules from those that can be tested with SQMLint?

■ Answer:

You can select specific rules to be tested, from those that can be tested with SQMLint. You can select these rules as follows:

- (1) Test all rules that can be tested with SQMLint.
- (2) Of the rules that can be tested with SQMLint, test only those deemed "required" by MISRA C.
- (3) Of the rules that can be tested with SQMLint, test all deemed "required", as well as those deemed "advisory" whose numbers are specified.
- (4) Of the rules that can be tested with SQMLint, test only those whose numbers are specified.

For details about rules that can be tested with SQMLint and their numbers, in Chapter 7, see *Table 7.2 Rules Supported by SQMLint*.

8.4.2 Outputting Report Files

■ Question:

I want to compile multiple C source files while outputting the results, as checked with SQLint, to a report file. How can I output a separate report file for each source file?

■ Answer:

Use the High-performance Embedded Workshop or TM Embedded Workshop.

● For High-performance Embedded Workshop:

For M3T-NC30WA or M3T-NC308 WA:

- (1) Choose the [Options] menu, and then [Renesas M16C Standard Toolchain].
- (2) In the [C] tab of the displayed dialog box, select [MISRA C Rule Check] for [Category].
- (3) Set [[-r] Report file name:] as follows:

```
$ (CONFIGDIR) ¥$ (PROJECTNAME) .csv
```

For M3T-CC32R

- (1) Choose the [Options] menu, and then [Renesas M32R Standard Toolchain].
- (2) In the [C] tab of the displayed dialog box, select [MISRA C Rule Check] for [Category].
- (3) Set [-misra_report Report file name:] as follows:

```
$ (CONFIGDIR) ¥$ (PROJECTNAME) .csv
```

● For TM:

For M3T-NC30 WA or M3T-NC308 WA:

- (1) Choose the [Project] menu, and then [Option Browser].
- (2) Choose [CFLAGS], and then [Edit...].
- (3) For the option change category, select [MISRA-C check option].
- (4) Select the [-sqlint] option, and set the parameter string as follows:

```
-misra all -r $*.csv
```

For M3T-CC32R

- (1) Choose the [Project] menu, and then [Option Browser].
- (2) Choose [CFLAGS], and then [Edit...].
- (3) For the option change category, select [MISRA-C check option].
- (4) Select the [-misra_report] option, and set the parameter string as follows:

```
$*.csv
```

Note:

When entering commands from the command line, specify the name of the report file for each compiled file, as follows:

```
>nc308 -c test.c -sq -sqlint "-misra all -r test.csv"
```

In other words, compile one C source file for each command.

For example, when the following is executed on the command line, only the report results for test2.c are saved in the "report.csv" file:

```
>nc308 test1.c test2.c -sq -sqlint "-misra all -r report.csv"
```

8.4.3 Report Messages (1)

■ Question:

When I specify the enum type for the parameter of a prototype declaration, and specify an enumerator as the actual parameter when calling the function, the following message is output even though they are of the same type. Why is this message output?

Example C source:

```
enum E { A, B, C };
void func1(enum E);

void func2()
{
    func1(A);
}
```

Message output:

```
Rule 77 (Complaining) "parameter type shall be compatible with prototype,
the 1st parameter"
```

■ Answer:

The type for enumerators is int.

Since SQMLint performs strict comparison for dummy parameters and their corresponding actual parameters, the enum E type and int type are treated as distinct, and the report message is output. Disregard any such messages.

8.4.4 Report Messages (2)

■ Question:

When I specify an enumerator on both sides of the `:` ternary operator, and substitute the resulting return value with an enumeration variable, the following message is output even though they are of the same type. Why is this message output?

Example C source:

```
enum E { A, B, C };

void func(int i)
{
    enum E e;
    e = (i == 0) ? A : B;
}
```

Messages output:

```
Rule 43 (Complaining) "information loss conversion (from 'signed int' to
'enum E') in assignment operation"
```

```
Rule 29 (Complaining) "enum type object to which has not been assigned own
enumerator"
```

■ Answer:

When you specify enumerators for both sides of the `:` ternary operator, the operator returns a value of the `int` type. As such, this is the same as substituting an `int`-type constant for an enumeration variable, and the report message is output. Disregard any such messages.

8.5 High-performance Embedded Workshop

8.5.1 Link Order for Files

■ Question:

I want to create a project in the High-performance Embedded Workshop, but the links for the files end up being alphabetized by file name. How can I have the startup file (ncrt0.r30) linked first?

■ Answer:

Perform the following:

- For M3T-NC30WA Version 5.20 R1:
 - (1) Choose the [Options] menu, and then click [Renesas M16C Standard Toolchain...].
(The [Renesas M16C Standard Toolchain] dialog box appears.)
 - (2) Click the [Link] tab.
 - (3) For [Category:], select [Input].
 - (4) For [Show entries for:], select [Relocatable files].
 - (5) Click the [Add] button.
(The [Add Relocatable Files] dialog box appears.)
 - (6) For [Relative to:], select [Configuration directory].
 - (7) For [File path:], enter "ncrt0.r30".
 - (8) In the [Add Relocatable Files] dialog box, click the [OK] button.
 - (9) In the [Renesas M16C Standard Toolchain] dialog box, click the [OK] button.

This will allow you to link "ncrt0.r30" first.

- For M3T-NC30WA Version 5.30 R1, M3T-NC308WA Version 5.20 R1, and M3T-NC8C Version 5.30 R1:

When a new project workspace is created, the file will automatically be linked first only if the name of the startup program is "ncrt0.a30".

Otherwise, this can be accomplished as follows:

- (1) Choose the [Options] menu, and then click [Renesas M16C Standard Toolchain...]^{#1}.
(The [Renesas M16C Standard Toolchain] dialog box appears.)
#1
For M3T-NC308WA, this is [Renesas M32C Standard Toolchain...].
For M3T-NC8C, this is [Renesas R8C Standard Toolchain...].
- (2) Click the [Link] tab.
- (3) From [Category:], select [Other].

Select the [Start-up program is linked to a head.] option to have the startup file linked first.

For startup programs whose file name is not "ncrt0.a30", use the first one of the above methods.

Overview of the [Relocatable files] list:

When files registered in the workspace are added to the [Relocatable files] list, they are linked before the files registered in the workspace.

When files not registered in the workspace are added to the [Relocatable files] list, they are linked after the files registered in the workspace.

Files added to the [Relocatable files] list are linked in list order.

8.5.2 Link Order for Relocatable Files

■ Question:

How can I change the order in which relocatable files are linked?

■ Answer:

Relocatable files are linked in the following order:

- (1) Files registered in both the workspace and the [Relocatable files] list are linked in the order of the [Relocatable files] list.
- (2) Files registered only in the workspace are linked in alphabetical order.
- (3) Files registered only in the [Relocatable files] list are linked in list order.

Add files to the [Relocatable files] list as necessary.

8.5.3 Generating Motorola S Format Files

■ Question:

How can I generate Motorola S format files?

■ Answer:

Perform the following:

- (1) Choose the [Options] menu, and then click [Build Phase].

The [Build Phase] dialog box appears.

- (2) Click the [Build Order] tab.

The order list of the build phase appears.

For M3T-NC30WA Version 5.20 Release 1, select [M16C Stype Converter].

For M3T-NC30WA Version 5.30 Release 1; select [M16C Load Module Converter].

For M3T-NC308WA Version 5.20 Release 1; select [M32C Load Module Converter].

For M3T-NC8C Version 5.30 Release 1; select [R8C Load Module Converter].

For M3T-CC32R, select [M32R Stype Converter].

- (3) Click the [OK] button to close the dialog box.

- (4) Choose the [Options] menu, and then click [Renesas M16C Standard Toolchain...]^{#1}.

The [Renesas M16C Standard Toolchain] dialog box appears.

#1

For M3T-NC308WA, this is [Renesas M32C Standard Toolchain...].

For M3T-NC8C, this is [Renesas R8C Standard Toolchain...].

For M3T-CC32R, this is [Renesas M32R Standard Toolchain...].

- (5) In the [Renesas M16C Standard Toolchain] dialog box, click the [Lmc] tab.

The [Option Settings for Loading Module Converters] window appears.

- (6) Set the necessary options, and click the [OK] button to close the dialog box.

This completes these settings.

From now on, the loading module converter is executed at build-time.

8.5.4 Installing High-performance Embedded Workshop (1)

■ Question:

I installed the version of the High-performance Embedded Workshop that comes with M3T-NC30WA, on a machine on which the High-performance Embedded Workshop environment was already installed. But even when I start the High-performance Embedded Workshop, M16C is not displayed as a toolchain.

■ Answer:

When installing the version of the High-performance Embedded Workshop that comes with M3T-NC30WA, be sure to install it in the same directory as the already installed the High-performance Embedded Workshop, overwriting the previous installation. Also, since the High-performance Embedded Workshop environment needs to be installed before M3T-NC30WA, be sure to use the installer for the M3T-NC30WA compiler.

For those using SHC Version 7 and H8C Version 5:

Be sure to download and install the High-performance Embedded Workshop revisions for the updated compiler package from the following URL, and then install the M3T-NC30WA compiler:

http://download.renesas.com/eng/mpumcu/upgrades/IDEs_and_project_managers/hew/index.html

8.5.5 Installing High-performance Embedded Workshop (2)

■ Question:

When I start the High-performance Embedded Workshop and try to create a new project workspace, I cannot, since nothing is displayed in the project type field. Why is this?

■ Answer:

For the version of the High-performance Embedded Workshop environment that comes with M3T-NC30WA, be sure to use the M3T-NC30WA installer. Also, when installing M3T-NC30WA for the High-performance Embedded Workshop on a machine on which the High-performance Embedded Workshop environment is already installed, be sure to perform installation by overwriting the existing the High-performance Embedded Workshop environment.

If this does not solve the problem, perform the following for toolchain registration:

- (1) Start the High-performance Embedded Workshop.
- (2) When the [Welcome!] dialog box appears, click the [Cancel] button.
- (3) In the High-performance Embedded Workshop menu, choose [Tools], and then click [Administration]. The [Tool Administration] dialog box appears.
- (4) In the [Tool Administration] dialog box, click the [Registration] button. The [Choose HEW Registration File] dialog box appears.
- (5) In the [Choose HEW Registration File] dialog box, for the file location, navigate to the M3T-NC30WA installation directory (which is "%MTOOL" by default).
- (6) Select "nc30wa.hrf", and click the [Select] button.
- (7) When you return to the [Tool Administration] dialog box, click the [OK] button.

8.5.6 Cancellling a Build

■ Question:

How can I cancel a build when a compiler error occurs while the build is being performed?

■ Answer:

With the default High-performance Embedded Workshop settings, even when an error occurs during a build, the build is not cancelled, and processing is performed through to linking. You can perform the following settings to cancel a build whenever an error occurs.

- (1) From the [Tools] menu, click [Options]. The [Options] dialog box is displayed.
- (2) In the [Options] dialog box, click the [Build] tab. The settings for build processing are displayed.
- (3) Select [Stop build if the number of errors is exceeded:].
(You can specify this number of errors in the adjacent test box.)
- (4) Click the [OK] button to close the dialog box.

To cancel a build when a warning occurs, select [Stop build processing if the number of warnings is exceeded:]. Just as with errors, you can specify this number of warnings in the adjacent test box.

8.5.7 Selecting a Build Target

■ Question:

When I perform a build with the High-performance Embedded Workshop3, the file built is not the one specified (the file selected in the tree of the [Project] tab, in the workspace window), but the one open in the editor. Why is this?

■ Answer:

When the file open in the editor has focus (you can see the cursor), it becomes the build target. This is because source files are often built after being edited. If the editor does not have focus, the file selected in the tree of the [Project] tab, in the workspace window, is the build target.

8.5.8 Build Configuration

■ Question:

Why is it that even if a build error occurs when [Release] is selected for the build configuration, compile can be performed properly when [Debug] is selected?

■ Answer:

This is because the options set for [Debug] and [Release] are different.

About the utility of build configurations:

When a project is created in the High-performance Embedded Workshop, two build configurations are created: [Release] and [Debug]. Different option patterns can be set for each configuration, so that these option patterns can be easily switched by switching the build configuration.

- Note that when the High-performance Embedded Workshop first creates these build configurations, the option patterns of each are identical.

For the above reasons, when option patterns are changed for one of the configurations in the High-performance Embedded Workshop, the changes are not automatically applied to the other configuration.

8.5.9 Outputting Debugging Information

■ Question:

I am using NC8C Version 5.30 Release 1. Is debugging information output by default when a project is created?

■ Answer:

Debugging information is not set to be output by default when a work space is created.

To output debugging information, set the corresponding options as follows:

- (1) From the [Options] menu, click [Renesas R8C Standard Toolchain].
- (2) The [Renesas R8C Standard Toolchain] dialog box appears.
- (3) Click the [C] tab.
- (4) From [Category], select [Object].
- (5) From the [Debug options] list, select [-g].
- (6) Click the [OK] button.

8.6 SBDATA Declaration Utility

8.6.1 SBDATA Declaration Utility

■ Question:

Why are some variables commented out when I use the SBDATA declaration utility (utlxx) to generate "sbddata.h"?

■ Answer:

Variables already declared as SBDATA in a program are mapped first to the SBDATA area. The SBDATA declaration utility maps the remaining parts to variables. As such, variables not mapped to the SBDATA area are output as comments. There are two kinds of comments output:

(1) Comments for variables already declared in #pragma SBDATA.

The @ character is appended to the end of the comment.

```
*/ //pragma SBDATA ***** /* size=( 1) / ref=[ 22] @  
*/
```

(2) Comments for variables that could have been but were not mapped to SBDATA.

```
*/ //pragma SBDATA ***** /* size=( 1) / ref=[ 22] */
```

MEMO

Appendix

Appendix A. Added Features

A.1 Features Added between Ver 1.00 Release 1 and Ver 2.00 Release 1

(1) SB308: SBADATA declaration utility

This utility analyzes information on variable usage, and outputs the SBADATA declarations (#pragma SBADATA) in order of access frequency.

You can reduce code size by including the output header file, and then recompiling.

(2) SP308: Special page declaration utility

This utility analyzes information on function calls, and outputs special page declarations (#pragma SPECIAL) in order of number of times functions are called.

You can reduce code size by including the output header file, and then recompiling.

(3) Levels for optimization options

- Optimization options have been divided into five levels, "-O1" through "-O5", to facilitate specification.

-O1	Only performs optimization that does not affect debugging information (line information).
-O2	With Version 2.00 Release 2, this is the same as -O1.
-O3	Performs optimization, including that which affects debugging information. Specifying -O is has the same effect as specifying -O3.
-O4	In addition to the optimizations performed with -O3, performs optimization whereby references to the const external variable are replaced with a constant.
-O5	In addition to the optimizations performed with -O4, performs common sub-expression optimization whereby variable aliases (such as indirect references) are disregarded. Note that when the same variable is used, if operation involves both direct and pointer indirect references, or multiple pointers are used indirectly, code may be created that behaves unexpectedly. As such, use this option only after taking generated code into careful consideration.

- -Ono_logical_or_combine (-ONLOC)

Prevents optimization in which consecutive logical ORs are combined. This takes effect when the level specified is "-O3" or greater.

- -Ocompare_byte_to_word (-OCBTW)

Performs comparison by word for sequential bytes in sequential memory.

This option takes effect regardless of the other options specified.

(4)Warning options

- -Wlarge_to_small (-WLTS)

Issues a warning for implicit type conversions from large sizes to small sizes.

This option does not take effect unless explicitly specified, even when the "-Wall" option is specified.

- -Wuninitialized_variable (-WUV)

Issues a warning when an uninitialized auto variable is used.

When the "-Wall" option is specified, this option takes effect automatically, even when not specified.

Note that in a user application, when initialization is performed by branching by means of such conditions as "if" or "for", the compiler will evaluate this as initialization not performed, and a warning will be output.

Example:

```
main()
{
    int i;
    int val;
    for(i=0;i<2;i++){
        f();
        val = 1; //Initialization always performed here
    }
    ff( val );
}
```

(5)Options to change output code

- -finfo

Outputs information files for SB308 and SP308.

- -fuse_CLIP (-fUC)

Generates code, using CLIP instructions.

- -fuse_MAX (-fUM)

Generates code, using MIN and MAX instructions.

(6)Variable name expansion processing for the asm function

Support has been added for variable name expansion, using \$@. When \$@ is specified, the compiler performs output after evaluating whether the corresponding variable is an auto variable, register variable, or external variable.

Example: `asm(" mov.w #10H, $@" , I);`

(7)AS308 functionality

Operations that contain strings can now be specified in assembler directions and mnemonic operands.

(8)XRF308 functionality

The maximum number of files that xrf308 can open concurrently has been changed to 600.

A.2 Features Added between Ver 2.00 Release 1 and Ver 2.00 Release 2

None.

A.3 Features Added between Ver 2.00 Release 2 and Ver 3.00 Release 1

(1) aopt308: assembler optimizer

The assembler optimizer aopt308 has been added, allowing conditional branches using the `adjnz` command to be optimized. Note that aopt308 is run automatically when compilation is performed with compile driver nc308, with one of "-O", "-O3", "-O4", "-O5", "-OR", or "-OS" options specified.

(2) utl308: SBDATA declaration and special page declaration utility

The SBDATA declaration utility and special page function declaration utility have been merged into utl308, and the existing sb308 and sp308 utilities have been discontinued.

You can reduce code size by including the output header file of this utility.

(3) STK viewer (for W95J and Solaris versions only)

A GUI has been created for the stack usage calculation utility, to increase usability and viewing.

(4) Map viewer (W95J version only)

A utility for viewing map information has been added, allowing map information to be viewed more easily.

(5) Support for the integrated development environment TM Version 3.00 (W95J version only)

Support has been added for integrated development environment TM Version 3.00.

Note that TM version 2.01 and earlier is no longer supported.

(6) Merging of stack information and utility information

Until now, stack information (.stk) and information for the SBDATA declaration and special page declaration utility (.utl) have been output individually for each source file. These have been merged with inspector information, so that separate data files no longer need to be created.

Note that inspector information is used by integrated development environment TM Version 3.00, and is stored in relocatable files (.r30) and absolute module files (.x30).

(7) NC308 functionality

(a) Optimization options

- The `"-Oloop_unroll[=maximum-number-of-loops](-OLU)"` command option

Improves execution speed by expanding, during compilation, "for" loops for which the number of loops is clear.

If the maximum number of loops is specified, loops that loop within the specified number of times will be expanded.

If the maximum number of loops is omitted, "for" loops that loop five or fewer times will be expanded.

(b) Strengthened warning functionality and suppression options

- Warnings for missing includes in the header files of standard library functions

When the `"-Wno_prototype"` or `"-Wall"` option is specified, the standard library functions are used. If header files needed for these functions are not included, a warning message will be output.

- The `"-Wno_warning_stdlib(-WNWS)"` command option

Suppresses the warning messages output when the above header files are not included.

This option takes effect when the `"-Wnon_prototype (-WNP)"` or `"-Wall"` option is specified.

- (c) Function start alignment by default, and corresponding suppression option
 Even alignment for instruction positions of function starts can now be performed by default.
 With this, the "-fno_align" option has been added to suppress even alignment for instruction positions of function starts, and the "-falign" option to perform even alignment has been discontinued.
- (d) Changes to utility information output options
- Changes in functionality for the "-finfo" command option
 As stack information and utility information have been merged, inspector information (including stack information and utility information) can now be output using the "-finfo" option.
 The previous "-fshow_stack_usage(-fSSU)" option has been discontinued.
- (e) CLIP, MAX, and MIN instructions used by default
 The "-fuse_CLIP(-fUC)" and "-fuse_MAX(-fUM)" options to use the CLIP, MAX and MIN instructions have been discontinued, so that these instructions can be used without specifying anything.
- (8) AS308 functionality
- (a) Functionality to create inspector information
- The "-finfo" command option
 The "-finfo" command option has been added, to create inspector information.
 - Directives
 The following directives have been added.
 - .INSF: Shows function (subroutine) start information for the inspector information.
 - .EINSF: Shows function (subroutine) end information for the inspector information.
 - .CALL: Shows function (subroutine) call destination information for the inspector information.
 - .STK: Shows stack information for the inspector information.
- (b) Branch optimization implementation
 Branch optimization can be performed using the adjnz and sbjnz instructions. Note these are equivalent to conditional branch instructions for optimal choice rules.
- (9) LMC308 functionality
- The "-A" command option
 The "-A" command option was added, to specify the address range of machine-language data output to the created file.
 - The "-F" command option
 The "-F" command option was added, to output optional data for addresses not registered in the specified absolute module file.
 - Extension functionality for the output file name specification option
 The output file name specification option "-O" can now be used to specify the extension of the output file name.
 - Changes to functionality for generation conditions of original Mitsubishi HEX format files

Previously, when the "-H" command option was specified, files were created in the original Mitsubishi HEX format for all sections registered in the specified absolute module file, based on the maximum address value of the set data. With this version, this only happens when addresses for sections that have the CODE or ROMDATA attribute exceed 1MB. Note that when an original Mitsubishi HEX format file is created, the "Original HEX format for mitsubishi microcomputers is generated" warning message is output.

- Changes to functionality when no ROM data exists

Previously, when sections in the specified absolute module file that had the CODE or ROMDATA attribute contained no data, a machine-language file containing no data was created. With this version, an error will occur.

- Execution path display

The execution status of lmc308 is now displayed as a path, the same as with the assembler.

A.4 Features Added between Ver 3.00 Release 1 and Ver 3.10 Release 1

(1) NC308 functionality

(a) Strengthened optimization

Optimization for conditional branching, bit operations, and register variables in the compiler architecture has been strengthened.

Optimization for the assembler has also been improved, including that for combining like logical operations for sequential areas and registers, and for machine-fixed instructions such as btsts.

(b) Strengthened warnings and additional options

The "-Wno_used_argument(-WNUA)" command option has been added.

When this option is specified, warnings are issued for unused parameters during function definition.

This function does not take effect, even when "-Wall" is specified, and needs to be specified separately.

(c) Options for assembling list files

The "-dsourc_in_list(-dSL)" command option has been added.

When this option is specified, whenever a relocatable file is created, an assembling list file is also created, and output as commented C source lines.

Note that this has no effect when the "-P", "-E", or "-S" option is specified, as assembling is not performed.

(d) Changes to "-dsourc(-dS)" option functionality

When this option is specified and commented C source lines are output to the generated assembly source, unnecessary relocatable files (.r30) will no longer be remain undeleted.

(2) AS308 functionality

When absolute addressing "base:19" is specified for the "BTST" operand of a bit operation instruction, code is now generated in S format.

(3) Strengthened functionality for utl308: SBDATA declaration and special page function declaration utility

- The "-sb308" and "-sp308" command options can now be specified simultaneously.

SBDATA processing and special page processing can now be performed concurrently.

- The "-fsection" command option has been added.

When this option is specified, variables and functions used in #pragma SECTION to change location sections are also processed.

- Processing for the "-o" command option has changed, and "-fover_write(-fOW)" has been added.

When an output file is specified with the "-o" option, if the file already exists, the results will be output to the standard output, and the file will not be overwritten.

When the "-fover_write(-fOW)" option is specified simultaneously, if an existing file is specified for "-o", the file will be force overwritten.

A.5 Features Added between Ver 3.10 Release 1 and Ver 3.10 Release 2

(1) NC308 functionality

(a) Increased maximum nesting for #include

The maximum number of nested files that can be loaded using the #include directive has been increased from 8 to 40.

(b) Changes in how comments are processed

Comment processing has been changed to match that used for common C++ processing.

In previous versions, portions between // and line feed codes were processed after portions enclosed between /* and */ were processed. In this version, comment processing takes place from the comment delineator character first.

As a result, keep in mind that comment processing has changed as shown in the following example.

```
i = 4 /* comment */
      +2;
```

In previous versions, everything between /* and */ was treated as a comment, leaving `i = 4 / +2;`.

In this version, everything between // and the end of the line is treated as a comment, leaving `i = 4 +2;`.

(c) Support for lowercase #pragma extension function names

The designated extension function words following the #pragma directive (such as ADDRESS, INTERRUPT, SBADATA, and ASM) can now be specified using lowercase letters. The functionality does not change regardless of the case used.

(d) Changes to #pragma ASM, #pragma ENDASM

In previous versions, if non-paired quote characters (' or ") were specified between #pragma ASM and #pragma ENDASM, an error would occur during token analysis. With this version, this is allowed.

```
#pragma ASM
nop ; Insert "NOP" // The double quote characters are paired, and no problem exists.
nop ; Don't care // Only one quote character exists.
// This would lead to an error in previous versions,
// but is now allowed.
#pragma ENDASM
```

(e) Strengthened warning functionality

In previous versions, when the "-Wno_used_argument(-WNUA)" command option was specified, warnings would be issued for unused stack passing arguments. With this version, warnings are also issued for unused register passing arguments.

A.6 Features Added between Ver 3.10 Release 2 and Ver 3.10 Release 3

None.

A.7 Features Added between Ver 3.10 Release 3 and Ver 5.00 Release 1

(1) Linux version

A Linux (supporting Japanese Turbolinux 7 workstation) version has been added.

(2) NC308 functionality

(a) Support for the "long long" type, "_Bool" type, and "restrict" modifier

Support has been added for the newly created types ("long long" type, and "_Bool" type) and "restrict" modifier, for ISO/IEC 9899:1999 (ANSI C99).

(b) Extension function #pragma BITADDRESS

The extension function #pragma BITADDRESS has been added, allowing "_Bool" type external variables to be allocated to 1 bit of a specified absolute address.

(c) Extension function #pragma SB16DATA

The extension function #pragma SB16DATA has been added, allowing external variables to be accessed using dsp16[SB] addressing.

(d) Extension function #pragma DMAC

The extension function #pragma DMAC has been added, allowing external variables to be allocated to the DMAC register, so that C can be used to access the DMAC register. This functionality supports DMAC channels 0 and 1.

(e) Strengthened optimization functionality

Optimization (especially that improving execution speed) including the following has been strengthened:

- Inline function functionality
- Constant transmission optimization
- Optimization of analysis for branching, such as using "if" statements
- Optimizations such as those for arithmetic calculations

(f) Changes to how extension function #pragma SECTION is processed

Processing for #pragma SECTION has been changed to allow the "data" and "rom" section names within a given source file to be changed multiple times.

(g) Changes to predefined macros for the M32C/80 series

When the "-M82" code generation option for the M32C/80 series is used, "M32C80", and not "M16C80", is now defined as the predefined macro.

(h) Faster interrupt processing functions

Register save and restore processing for interrupt processing functions has been made faster.

(i) asm function extension

Up to two \$\$ and \$@ can now be used within the asm function.

(j) Binary support for integer constants

Binary numbers can now be specified for integer constants. Integer constants starting with the "0b" prefix are treated as binary numbers. For example, you can specify "0b00010010" in binary to represent the hexadecimal number "0x12".

(3) Inline expansion macro for standard library functions

An inline expansion macro for the strcpy, strcmp, memcpy, and memset standard library functions has been added to the standard header "string.h".

(4) Strengthened AS308 functionality

(a) Optimization for address register relative addressing

When the address register relative value 0 is used for standard instructions and bit instructions, address register indirect addressing is selected.

```
mov.w #1234h,0[A0] → mov.w #1234h,[A0]
bclr      1,0[A0]   →   bclr      1,[A0]
```

(b) The .SBSYM16 directive

When symbols defined with this directive are referenced, dsp16[SB] addressing is selected.

```
.glb      sym
.sbsym16  sym
abs.b sym      →   dsp:16[SB] is selcted
```

(5) Strengthened LN308 functionality

(a) Changes to specification rules for command files

The number of characters that can be specified on one line has been increased from 255 to 2,048.

(6) Strengthened map viewer functionality

The following functionality has been added to the map viewer.

- Map information printing
- Scrolling in the left window
- Displayed expansion/reduction of the memory size image

A.8 Features Added between Ver 5.00 Release 1 and Ver 5.10 Release 1

(1) NC308 functionality

(a) The "-fdouble_32[-fD32]" command option

Specifies that double types be handled as 32-bit data lengths, the same as float types.

Note:

When using this option, be sure to indicate the function prototype. If no prototype declaration exists, code may not be generated properly.

(b) The "-Wno_used_static_function[-WNUSF]" command option

When the "-Ostatic_to_inline[-OSTI]" option is specified, the following warning message will be displayed when the static function is expanded inline, or is not referenced from anywhere:

```
[Warning(ccom):xxx.c,line xx] Code generation for static function
(function-name) can be suppressed by using -ferase_static_function(-fESF)
option.
```

(c) The "-ferase_static_function=*name-of-the-static-function*[-fESF=*name-of-the-static-function*]" command option

Prevents code from being generated for the specified static function.

Note:

Specify this option for static functions detected by the "-Wno_used_static_function[-WNUSF]" command option.

(d) Strengthened "-Oconst" command option

Replaces references to array data initialized by a constant with references to the constant.

(e) Strengthened processing for sum calculations

The rmpa instruction is now generated for sum calculation processing within "for" statements.

Program example:

```
signed char ch1[10];
signed char ch2[10];
int ih1[10];
int ih2[10];

#define LOOP_MAX 9

signed char *pc1,*pc2;
int *pi1,*pi2;

int i;
long l;

void func_c1(void)
{
    int j;

    i = 0;
    for( j=0; j<LOOP_MAX; j++ ){
        i = i + (int)*pc1 * (int)*pc2;
        pc1++;
        pc2++;
    }
}

void func_c2(void)
```

```

{
    int    j;

    i = 0;
    for( j=0; j<LOOP_MAX; j++ ){
        i = i + (int)ch1[j] * (int)ch2[j];
    }
}

```

Generated code example:

```

_func_c1:
    pushm    R1,R2,R3,A0,A1
    mov.w   #0000H,_i:16
    mov.w   _i:16,R0
    mov.l   _pc1:16,A0
    mov.l   _pc2:16,A1
    mov.w   #0009H,R3
    rmpa.b
    mov.w   R0,_i:16
    add.l   #00000009H,_pc1:16
    add.l   #00000009H,_pc2:16
    popm    R1,R2,R3,A0,A1
    rts

_func_c2:
    pushm    R1,R2,R3,A0,A1
    mov.w   #0000H,_i:16
    mov.w   _i:16,R0
    mov.w   #(_ch1&0FFFFH),A0
    mov.w   #(_ch2&0FFFFH),A1
    mov.w   #0009H,R3
    rmpa.b
    mov.w   R0,_i:16
    popm    R1,R2,R3,A0,A1
    rts

```

Notes:

- One of the "-O[3-to-5]", "-OS", or "-OR" optimization options needs to be specified.
- The rmpa instruction may not be generated, depending on the processing for the sum calculation in the "for" statement.

(2) AS308 functionality

(a) AS308 options "-PATCH_TA" and "-PATCH_TAn"

Generates code to save precautions for timer functionality, for controlling three-phase motors.

(b) LN308 option "-U"

Outputs a warning message output for unused functions.

(c) LMC308 option "-F"

Ranges can now be specified for processing to embed arbitrary data in free areas.

A.9 Features Added between Ver 5.10 Release 1 and Ver 5.20 Release 1

(1) NC308 functionality

(a) High-performance Embedded Workshop

The High-performance Embedded Workshop, an integrated development environment that combines and batch manages various tools, including editors and debuggers, has been added.

(b) The "-fno_switch_table [-fNST]" command option

Generates branching code that always performs comparisons for "switch" statements

Note:

Specify this option to prevent code from being created that uses table jumps for "switch" statements.

(c) The "-fswitch_other_section [-fSOS]" command option

Specifies table codes for "switch" statements in a section other than the program section.

Note:

This option does not take effect when the "-fno_switch_table [-fNST]" option is specified.

(d) The "-fmake_vector_table [-fMVT]" command option

Automatically generates interrupt vector tables.

(e) The "-fmake_special_table [-fMST]" command option

Automatically generates special page tables.

(f) The "-Ofoward_function_to_inline [-OFFTI]" command option

Performs inline expansion for all inline functions.

(g) The "-Ofloat_to_inline [-OFTI]" command option

Performs inline expansion of floating-point runtime libraries, to increase speed for floating-point calculation processing.

Note:

This option takes effect when specified simultaneously with the "-M82" compile option.

(h) The "-Oglb_jump [-OGJ]" command option

Selects the optimum branching instruction, based on the branching distance during linking.

(i) The "-Wno_used_function [-WNUF]" command option

Outputs a warning when an unused formula is detected.

(j) The "-Wstop_at_link [-WSAL]" command option

Prevents an absolute module file from being created when a warning occurs during linking.

(k) The "-Wundefined_macro [-WUM]" command option

Outputs a warning when an undefined macro is used within an "#if" statement.

(l) Options for the extension function #pragma INTHANDLER(#pragma HANDLER)

Multiple interrupts are now allowed immediately after an interrupt handler is entered.

Note:

This functionality takes effect when the "/E" option is specified in #pragma INTHANDLER (#pragma HANDLER).

- (2) AS308 functionality
 - (a) The "-fMVT" command option
Automatically generates variable vector tables.
 - (b) The "-fMST" command option
Automatically generates special page tables.
 - (c) The "-JOPT" command option
Optimizes branching instructions that reference global labels.
 - (d) The .ID directive
Sets the ID code for the ID code check function.
 - (e) The .PROTECT directive
Sets a value for the control address for ROM code protection.
 - (f) The .RVECTOR directive
Sets the software interrupt number and software interrupt name.
 - (g) The .SVECTOR directive
Sets the special page number and special page name.
- (3) LN308 functionality
 - (a) The "-fMVT" command option
Automatically generates variable vector tables.
 - (b) The "-fMST" command option
Automatically generates special page tables.
 - (c) The "-VECT" command option
Sets addresses for free areas when automatic generation is performed for variable vector tables.
 - (d) The "-JOPT" command option
Optimizes branching instructions that reference global labels.
 - (e) The "-W" command option
Prevents an absolute module file from being created when a warning occurs.
 - (f) Removal of precautions for the "-L" command option
The following precautions from the previous version have been removed:
When multiple library files are specified, the linker references them in the order they are specified. As a result, an undefined symbol error will occur when the following conditions are satisfied:
 - The relocatable file (sample.r30) references a global symbol registered in the library file (A.LIB).
 - The relocatable module in the library file (A.LIB) linked in (1) references a global symbol registered in another library file (B.LIB).
 - In the "-L" option, the library file (B.LIB) from (2) is specified before the library file (A.LIB) from (1), such as in the following example:

```
>ln308 sample.r30 -L B.LIB A.LIB
```

(4) LMC308 functionality

(a) The "--protectx" command option

Sets a value for the control address for ROM code protection.

C Compiler for M16C Family Application Notes

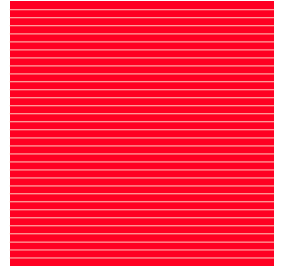
Publication Date: Jun. 16, 2005 Rev.2.00

Published by: Sales Strategic Planning Div.
Renesas Technology Corp.

Edited by: Microcomputer Tool Development Department
Renesas Solutions Corp.

© 2005. Renesas Technology Corp. and Renesas Solutions Corp., All rights reserved. Printed in Japan.

C Compiler for M16C Family Application Notes



Renesas Technology Corp.

2-6-2, Ote-machi, Chiyoda-ku, Tokyo, 100-0004, Japan