Hitachi Microcomputer Development Environment System

SuperH RISC engine Family

C Compiler

HITACHI

7/31/96
Hitachi Micro Systems, Inc.
Thomas Mayer

Notice

When using this document, keep the following in mind:

1. This document may, wholly or partially, be subject to change without notice.
2. All rights are reserved: No one is permitted to reproduce or duplicate, in any form, the whole or part of this document without Hitachi's permission.
3. Hitachi will not be held responsible for any damage to the user that may result from accidents or any other reasons during operation of the user unit according to this document.
4. Circuitry and other examples described herein are meant merely to indicate the characteristics and performance of Hitachi's semiconductor products. Hitachi assumes no responsibility for any intellectual property claims or other problems that may result from applications based on the examples described herein.
5. No license is granted by implication or otherwise under any patents or other rights of any third party or Hitachi, Ltd.
6. MEDICAL APPLICATIONS: Hitachi's products are not authorized for use in MEDICAL APPLICATIONS without the written consent of the appropriate officer of Hitachi's sales company. Such use includes, but is not limited to, use in life support systems. Buyers of Hitachi's products are requested to notify the relevant Hitachi sales offices when planning to use the products in MEDICAL APPLICATIONS.

Preface

The Hitachi SuperH RISC (reduced instruction set computer) engine family is a new generation series of single-chip RISC microprocessors that not only realize high-performance operation processing, but contain several types of on-chip peripheral devices and can be incorporated into devices operating with low power consumption.

These application notes describe how to create application programs making effective use of the Hitachi SuperH RISC engine family functions and capabilities, using the SH Series C Compiler Version 3.0.

For detailed specifications of the C compiler, refer to the *SH Series C Compiler User Manual*.

The Hitachi SuperH RISC engine family is made up of the SH-1, SH-2, and SH-3.

Application Notes Configuration

These application notes consist of five sections and several appendices, as summarized below:

- Section 1 describes methods of creating C language programs.
- Section 2 describes techniques specific to the SH series C compiler extended functions and intrinsic functions.
- Section 3 describes methods of creating C programs that make good use of the Hitachi SuperH RISC engine family capabilities.
- Section 4 describes cautions when linking C language and assembly language programs, and during use of cross software with object files generated by the C compiler.
- Section 5 is a listing of answers to questions commonly asked by users.
- The appendices list the SH series C compiler options and differences between Version 2.0 and Version 3.0.

Related Manuals

Related manuals are as follows:

- The hardware manuals of each of the SH7000 series, SH7600 series, and SH7700 series microcomputers.
- *SH Series C Compiler User Manual*
- *SH Series Cross Assembler User Manual*
- *H Series Linkage Editor User Manual*
- *H Series Librarian User Manual*
- *SH Series Simulator/Debugger User Manual*

Contents

Figures

Tables

Section 1 Introduction

1.1 Overview

The SH series C compiler enables the creation of effective C programs making use of the functions and capabilities of single-chip RISC microprocessor Hitachi SuperH RISC engine family with onboard peripherals. This manual describes the methods of creating application programs using this C compiler.

1.2 Features

Functions: The following functions allow creation of effective Hitachi SuperH RISC engine family application programs:

- C language specification for interrupt functions and Hitachi SuperH RISC engine family dedicated special instructions
- Position independent code generation (SH-2, SH-3 only)
- High-speed floating-point operations
- Selection of optimized execution speed priority, memory efficiency priority

Optimizations: The following optimizations give full capability to the Hitachi SuperH RISC engine family with its RISC type instruction set:

- Automatic/optimized allocation of local variables to registers
- Operation reduction
- Pipeline optimization
- Fold-in of constants
- Commonality of character strings
- Deletion of common format/loop constant format
- Deletion of unnecessary statements
- Tail recursion optimization

The above features make efficient programming possible for individuals not familiar with the Hitachi SuperH RISC engine family architecture.

1.3 Installation Method

1.3.1 UNIX Version

File Format: Archive file format (tar format).

Table 1.1 shows the C compiler file organization.

Table 1.1 C Compiler File Organization (UNIX Version)

Item	File Names
C compiler unit	shc, shcfrt, shcmdl, shcgen, shcpep, shcasm, shcprm, shctil, shcerr.msg, shcerr.off, shchlp.msg
Standard include files	assert.h, ctype.h, errno.h, float.h, limits.h, machine.h, math.h, mathf.h, setjmp.h, stdarg.h, stddef.h, stdio.h, stdlib.h, string.h. smachine.h, umachine.h
Standard library files	shclib.lib, shcnpic.lib, shcpic.lib, shc3npb.lib, shc3pb.lib, shc3npl.lib, shc3pl.lib
Sample files	7032.h, 7032.c

Installation: Perform the following steps to install the SH series C compiler onto the UNIX system ("%" within the explanation indicates a shell prompt).

1. Create a path that stores each file of the C compiler.

   % mkdirΔ<C compiler pathname>(RETURN)

2. Input the following commands to copy the C compiler files in the path just created (input device is assumed to be /dev/rst0). Caution: Store all of the C compiler unit files in the same directory.

   % cdΔ<C compiler pathname>(RETURN)

   % tarΔxvfΔ/dev/rst0(RETURN)

3. Set the path in which the C compiler is installed.

- For the C shell, add the following settings to the login path file (.login):

  setΔpath=(<C compiler pathname>Δ<pathname string being used>)(RETURN)*[1]

  setenvΔSHC_LIBΔ<C compiler pathname>(RETURN)*[2]

  Note 1: Add the path in which the C compiler is stored to the head of the path list within parentheses. Example: When setΔpath=(.Δ/user/binΔ/bin) is already established, designate as follows:

  setΔpath=(<C compiler use pathname>Δ.Δ/user/binΔ/bin)(RETURN)

  Note 2: Set the environment variables indicating the C compiler path. Example: When the C compiler is stored in /ex/shcV3/bin, designate as follows:

  setenvΔSHC_LIBΔ/ex/shcV3/bin (RETURN)

- For the Bourne shell, add the following settings to the login path file (.profile).

```
PATH=<C compiler use pathname>Δ<pathname string being used>)(RETURN)*1
exportΔPATH (RETURN)*1
SHC_LIB=<C compiler use pathname>(RETURN)*2
exportΔSHC_LIB (RETURN)*2
```

Note 1: Add the path in which the C compiler is stored to the head of the path list. Example: When PATH=.:/user/bin:/bin) is already established, designate as follows.

```
PATH=(<C compiler use pathname>:.:/user/bin:/bin (RETURN)
```

Note 2: Set the environment variables indicating the C compiler use path. Example: When the C compiler is stored in /ex/shcV3/bin, designate as follows.

```
SHC_LIB=/ex/shcV3/bin (RETURN)
```

Explanation of Environment Variables:

1. SHC_LIB. Indicates the storage location of the SHC compiler unit. Consequently, the C compiler will not operate unless all of the C compiler unit files are placed in the same directory beforehand.
2. SHC_TMP. The C compiler creates a temporary file in a path called either /usr/tmp or /tmp for the internal data necessary during compilation. Confirm that the path exists. If it does not, create a path for storing the temporary file. If a path is established in a location other than /usr/tmp or /tmp, set the path for storing temporary files with the environment variable SHC_TMP. Temporary files are deleted after completion of the compilation process.
3. SHC_INC. Designated when the SHC compiler standard header file is retrieved from a specified path. Multiple designations can be made for this path by using commas (,) as delimiters. When this is not designated, the standard header file is retrieved from SHC_LIB.

1.3.2 PC Version

File Format: The files are MS-DOS file format. (The provided medium is 1.2-Mbyte format with the PC-98 version, and 1.44-Mbyte format with the IBM-PC version.)

Table 1.2 shows the C compiler file organization.

Table 1.2 C Compiler File Organization (PC Version)

Item	File Names
C compiler unit	SHC.EXE, SHCPRM.EXE, SHCTIL.EXE, SHCFRT.EXE, SHCMDL.EXE, SHCGEN.EXE, SHCPEP.EXE, SHCASM.EXE, DOS4G.EXE, SHCERR.MSG, SHCERR.OFF, SHCHLP.MSG
Standard include files	ASSERT.H, CTYPE.H, ERRNO.H, FLOAT.H, LIMITS.H, MATH.H, MATHF.H, SETJMP.H, STDARG.H, STDDEF.H, STDIO.H, STDLIB.H, STRING.H, MACHINE.H, SMACHINE.H, UMACHINE.H
Standard library files	SHCLIB.LIB, SHCPIC.LIB, SHCNPIC.LIB, SHC3NPB.LIB, SHC3PB.LIB, SHC3NPL.LIB, SHC3PL.LIB
Sample files	7032.C, 7032.H

Installation: Use the installer (install) to perform the installation onto the machine being used. "A>" within the explanation indicates a prompt.

1. SHC compiler directory organization: When the installer is run, directories are created with the following organization.
   - A:\SHC\BIN C compiler unit, standard include files
   - A:\SHC\LIB standard library files
   - A:\SHC\SAMPLE sample files
2. Running the installer: Insert the first floppy disk (1/2) into the drive (drive name is assumed to be B:) and input the following command:

```
A>B:\install(RETURN)

SH SERIES C Compiler Installation Program
Copyright (C) 1995 Hitachi, Ltd., Hitachi Software Engineering Co., Ltd.
Licensed Material of Hitachi, Ltd., Hitachi Software Engineering Co., Ltd.

        Menu:
            1 - Default installation
            2 - Custom installation
            3 - Quit
            Input number: 1                                    *1
```

Perform the installation with the default settings.

```
Installation parameters
(1) Files to be installed                                  Displays the files to
    Execution files, include files,                        be installed
    library files, sample programs
(2) Library type                                           Displays CPU type for
    SH1, SH2, SH3                                          library to be installed
(3) Directory                                              Displays installation
    Execution files directory [A:\SHC\BIN]                 destination directory for
    Include files directory [A:\SHC\BIN]                   each file
    Library files directory [A:\SHC\LIB]
    Sample programs directory [A:\SHC\SAMPLE]
    Compiler work directory [A:\TMP]                       Displays directory
                                                           containing temporary file
                                                           created when compiler is
                                                           being used

Menu:
    1 - Start installation
    2 - Select files to be installed
    3 - Select library type
    4 - Change install directories
    5 - Quit

Input number: 1                                            *2

Start installation: (Y: Yes, N: No)? Y
```

Start the installation.

```
MKDIR A:\SHC\BIN                                           Creates installation
MKDIR A:\SHC\LIB                                           destination directories
MKDIR A:\SHC\SAMPLE
EXPAND ASSERT.H A:\SHC\BIN                                 Displays expanded file
EXPAND CTYPE.H A:\SHC\BIN                                  names and installation
EXPAND ERRONO.H A:\SHC\BIN                                 destination directories
EXPAND FLOAT.H A:\SHC\BIN
           :
           :
Change floppy disk number 2 and press any key:
```

Insert the second floppy disk (2/2) and input any key.

```
EXPAND SHC.EXE A:\SHC\BIN
EXPAND SHCPRM.EXE A:\SHC\BIN
EXPAND SHCTIL.EXE A:\SHC\BIN
EXPAND SHCFRT.EXE A:\SHC\BIN
          :
          :
Installation completed. SETSHC.BAT is created in A:\SHC\BIN
```

Displays expanded file names and installation destination directories

This completes the installation. An environment setting usage sample batch file has been created in the directory where the compiler unit is installed. The environment setting usage sample batch file (SETSHC.BAT) has been created with the following contents in combination with the installation directory settings.

```
PATH A:\SHC\BIN;A:\;A:\DOS;A:\TOOL
SET SHC_LIB=A:\SHC\BIN
SET SHC_INC=A:\SHC\BIN
SET SHC_TMP=A:\TMP
SET DOS16M=1@1M-4M
```

The underlined section indicates the currently established pathname

Note that the DOS16M settings differ depending on the amount of memory installed in the machine being used; this should be confirmed by the user. To modify the installation files, CPU type for the libraries, or installation directories, select custom installation at the point marked *1 or else 2-4 at *2. Each of the settings can be modified.

An example of setting modification is given below. The modifications are: sample programs not installed; only SH-3 usage library installed; and installation destination directory name modified. [Y] indicates files installed; [N] indicates files not installed.

Installation file modification:

1. `Files to be installed`

   When not modifying the contents within [], just press RETURN

   ```
   Execution files [Y]: (RETURN)
   Include files [Y]: (RETURN)
   Library files [Y]: (RETURN)
   ```

   ```
   Sample programs [Y]: N
   ```

   Modification occurs with “N”

Library type modification:

```
2.  Library type                          Input “N” for items not to be installed
    SH1[Y]:N
    SH2[Y]:N
    SH3[Y]:(RETURN)                       Press RETURN when not modifying
```

Installation directory name modification:

```
3.  Directory
    Execution files directory             Input directory names for those to be
    [A:\SHC\BIN]:A\SHC3\BIN(RETURN)       modified
    Include files directory
    [A:\SHC\BIN]:A\SHC3\INC(RETURN)
    Library files directory
    [A:\SHC\LIB]:A\SHC3\LIB(RETURN)
    Compiler work directory               Press RETURN when not modifying
    [A:\TMP]:(RETURN)
```

After all setting modifications have been completed, or else when modification processing is discontinued with the [ESC] key, the installation information is displayed.

Installation file modification:

```
Installation parameters
1.  Files to be installed
    Execution files, include files, library files
2.  Library type
    SH3
3.  Directory
    Execution files directory[A:\SHC3\BIN]
    Include files directory [A:\SHC3\INC]
    Library files directory [A:\SHC3\LIB]
    Compiler work directory [A:\TMP]
```

Continue the installation operation following directions on the screen and confirming the modification locations.

Explanation of Environment Variables: Modify the contents of AUTOEXEC.BAT for the environment variable settings while referring to the environment setting usage sample batch file (SETSHC.BAT).

- SHC_LIB: Indicates the storage location of the SHC compiler unit.

- SHC_TMP: Designates the path where a temporary file used by the SHC compiler during operation is created. This setting cannot be omitted.
- SHC_INC: Designated when the SHC compiler standard header file is retrieved from a specified path. Multiple designations can be made for this path by using commas (,) as delimiters. When this is not designated the standard header file is retrieved from SHC_LIB.
- DOS16M: The protected memory area used by the compiler is designated with the environment variable DOS16M for the SHC compiler to use additional extended memory.

```
SET DOS16M=<switch_mode>[@<start_address>][:size]                    or
SET DOS16M=<switch_mode>[@<start_address>[-final address]]
```

Contents within [ ] can be omitted. 1-16M can be designated for both the address and size.

```
switch_mode
PC-98 series or compatible machine: 1
IBM-PC/AT series or compatible machine: 0
```

- Start address: Designates the first address of the memory area used by the compiler.
- End address: Designates the last address of the memory area used by the compiler.
- Size: Designates the amount of protected memory used by the compiler.

The setting of items is performed in decimal or hexadecimal (for hexadecimal, the prefix 0x is necessary). Numbers can be designated in kbyte/Mbyte units. Numbers without a suffix are considered to be kbytes.

Example: PC-98 series settings: 6 Mbytes of extended memory are installed, and the 5 Mbytes from 1 Mbyte to 6 Mbyte are used:

```
A> SET DOS16M=1@1M–6M (RETURN)          or
A> SET DOS16M=1@1M:5M (RETURN)
```

When using 4,096 kbytes of extended memory:

```
A> SET DOS16M=1:4096K (RETURN)
```

The DOS16M memory area setting can be omitted when the EMS driver corresponds to the VCPI (Virtual Control Program Interface) standards, or the DPMI (DOS protected mode interface) standards.

The EMS drivers (EMM.SYS, EMM386.EXE) for DOS Version 6.2 and earlier do not conform with the VCPI or DPMI standards, so the SHC compiler cannot be used when they are installed. For this reason, take the following countermeasures:

- When using MS-DOS Version 3.3:
  — Install an EMS driver that corresponds to the VCPI or DPMI standards, such as the Melco Company’s Melware or the I/O Data Company’s Memory Server II.

— Refer to the individual product manuals for the specifications of these drivers.

- When using MS-DOS Version 5.0:
  — When EMS memory is not necessary, do not install the EMS driver.
  — When EMS memory is necessary (using the EMM386.EXE included with DOS Version 5.0), execute DPMI.EXE before using the SHC compiler (DPMI.EXE is included with DOS Version 5.0).
  — General purpose EMS drivers (EMM.SYS) cannot be used.
- When using MS-DOS Version 6.2:
  — When EMS memory is not necessary, do not install the EMS driver.
  — When EMS memory is necessary (using the EMM386.EXE included with DOS Version 6.2), the compiler can be used by installing the included EMS driver (EMM386.EXE).
  — General purpose EMS drivers (EMM.SYS) cannot be used.

Memory Requirements and Disk Space Occupied (for both PC-98 and IBM-PC versions):

- CPU: The CPU must be an 80386SX or later.
- Memory Used: 640 kbytes of main memory and 5 Mbytes or more of protected memory are necessary to operate this system. 8 Mbytes or more of protected memory is recommended.
- Disk Space Occupied: Approximately 3,500 kbytes (when all libraries are installed).

1.4 Startup Method

This section explains the startup method for the SH series C compiler and gives an example of its use. For information on the compiler options, refer to Appendix A, Compiler Options.

The general command line is:

```
shc[Δ<option>Δ...][Δ<file name>[Δ<option>Δ...]...]
```

1. Single file compilation (file name: send_msg.c): An object file with the name send_msg.obj is generated.
   ```
   >  shcΔsend_msg.c(RETURN)
   ```
2. Single file compilation with compiler options designated (file name: send_msg.c): Both types of command line perform the same function.
   ```
   >  shcΔ-listfileΔ-show=noobject,expansionΔsend_msg.c(RETURN)
   >  shcΔsend_msg.cΔ-listfileΔ-show=noobject,expansion(RETURN)
   ```
   An object file named send_msg.obj and a list file named send_msg.lis without an object list, and with a source list after macro expansion, are generated.
   Add a hyphen (-) before compiler options. Use a comma (,) to delimit suboptions. In the PC version, a slash mark (/) can be designated instead of a hyphen before the compiler options, and suboptions can be bundled within parentheses (from Version 1.0 on). Consequently, the following description is also possible in the PC version.

```
> shc Δ/listfile Δ/show=(noobject,expansion) Δsend_msg.c(RETURN)
```

3. Multiple file compilation with compiler options designated (file names: send_msg.c, get_msg.c)

```
> shc Δ-cpu=sh2 Δsend_msg.c Δget_msg.c(RETURN)
```

The two SH-2 object files named send_msg.obj and get_msg.obj are generated. When compiler options are designated at the start of all the files, the compiler options become effective for every file.

```
> shc Δsend_msg.c Δ-debug Δget_msg.c(RETURN)
```

Two object files are generated; one named send_msg.obj that has debug information, and one named get_msg.obj that has no debug information. When compiler options are designated after a file, the compiler options become effective for that file only.

4. Input the startup command

```
shc(RETURN)
```

The command line format and compiler option list are output.

5. Cautions: If the compiler cannot be started up after installation, reconfirm the following items:

— Is the environment variable “PATH” set to a C compiler directory?

— Is the environment variable “SHC_LIB” set to a C compiler unit directory?

The environment variable “SHC_LIB” is used to designate the directory where the SHC compiler unit resides. Consequently, the compiler will not operate unless all of the C compiler unit files are placed in the same directory beforehand.

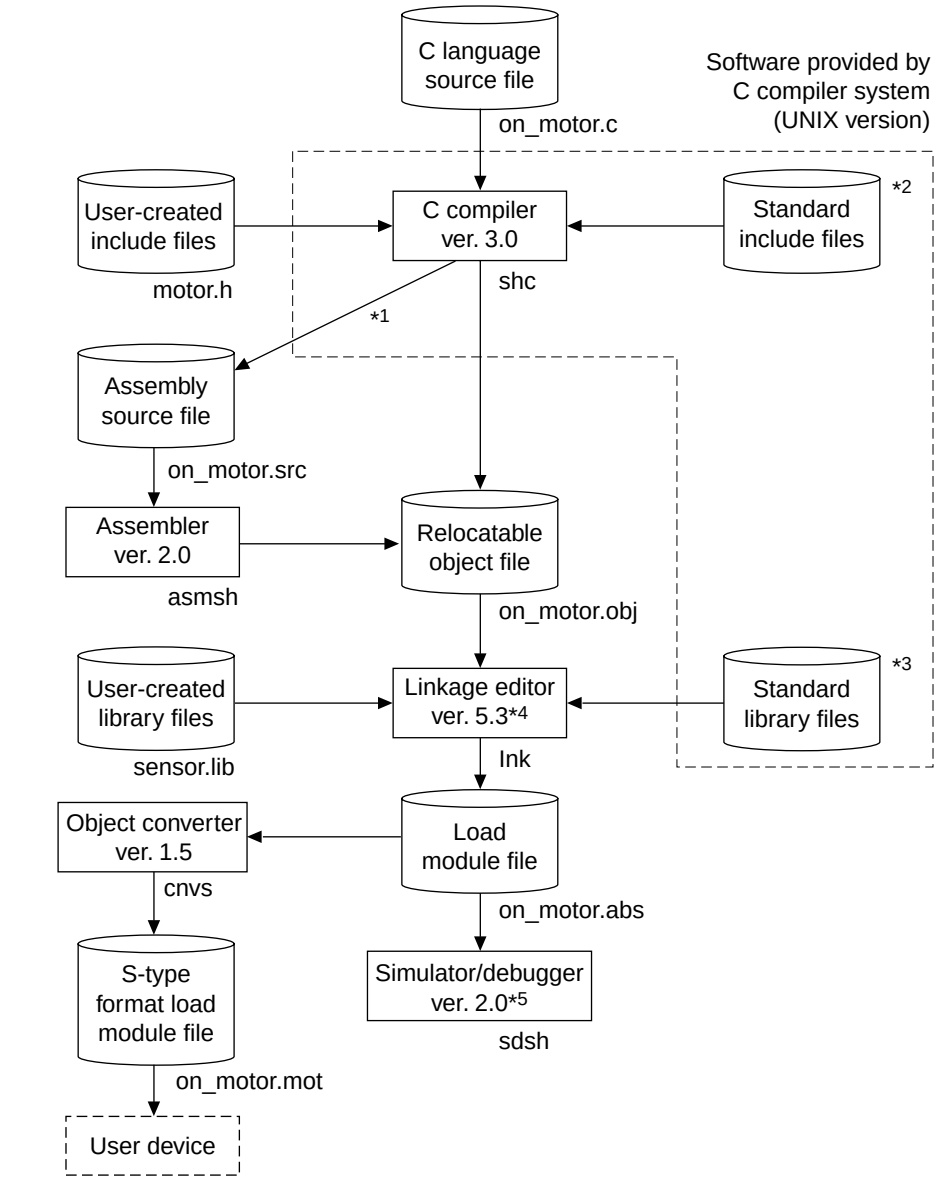
— For the PC version, are the environment variables “SHC_LIB”, “SHC_TMP”, and “DOS16M” correctly set?

— For the PC version, is an EMS driver installed? Or else, is execution occurring under the MS-Windows environment?

There are cases in which the compiler cannot be activated if an EMS driver is installed. Remove the EMS driver from CONFIG.SYS and restart. Additionally, there are cases in which coexistence with EMS applications is not possible. Activate the compiler from a DOS prompt in the MS-Windows environment, Version 3.1 or later. Refer to section 1.3, Installation Method, and attached software materials for details.

1.5 Program Development Procedures

Figure 1.1 shows program development features.



Notes:
1. Assembly source files can be output through option designation.
2. Define the C library functions and the names of macros necessary to use them.
3. A group of standard functions that can be used in C language programs. Includes C library functions and rountines used during execution.
4. See section 4.2, Relation to the Linkage Editor.
5. See section 4.3, Relation to the Simulator/Debugger.

zrisi01.eps

Figure 1.1 Program Development Features

The program development procedure will be explained below, using as an example the source file on_motor.c, which includes the header file motor.h. Refer to the individual cross software user manuals for details on cross software usage.

1.5.1 Source File Creation

Using the editor, create the source file.

1.5.2 Relocatable Object File Generation

Activate the compiler and compile the source file.

```
shc∆on_motor.c(RETURN)
```

An optimized relocatable object file named on_motor.obj that has no debug information is generated. Designate the listfile option to generate a list file.

1.5.3 Load Module File Generation

When the linkage editor is activated with the library file sensor.lib included as noted below, an executable load module file named on_motor.abs is generated.

```
lnk∆on_motor.obj∆-library=sensor.lib(RETURN)
```

Please note that even if the relocatable object file has debug information attached, no debug information will be output in the load module file if the debug option is omitted during linkage.

1.5.4 Load Module File Conversion to S-Type Format

When using a ROM writer to write into an EPROM, activate the file converter as follows.

```
cnvs∆on_motor.abs(RETURN)
```

An S-type format load module file named on_motor.mot is generated.

1.6 Sample Program Introduction

Figure 1.2 shows sample program introduction.

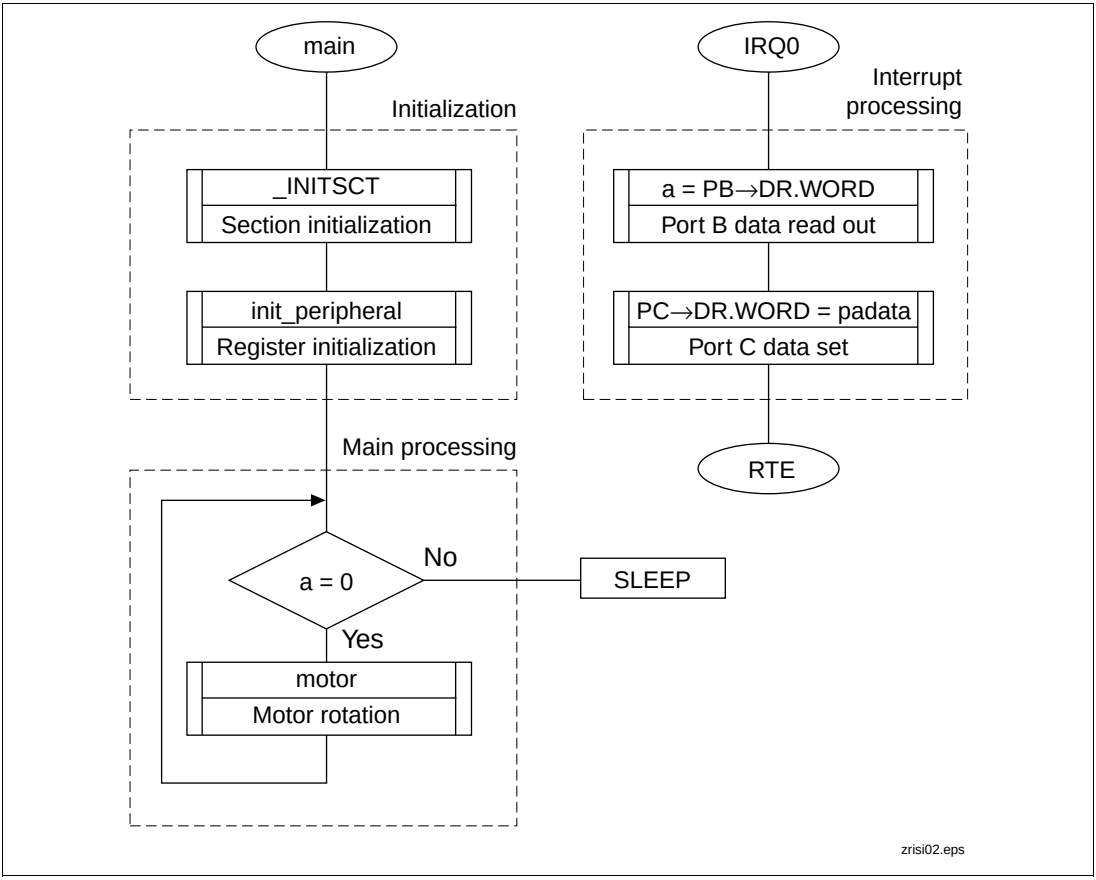


Figure 1.2 Sample Program Introduction

This section explains the actual creation of a program, using a sample program. The development environment is indicated in table 1.3.

Table 1.3 Sample Program Development Environment

OS	CPU
UNIX	SH-1

1.6.1 Vector Table Creation

The vector table creation program is shown below. Refer to section 2.1.3, Vector Table Creation, for details.

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

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

The SH-1 vector table created is shown in table 1.4. The function main is activated by a power on reset. The stack pointer is set to 0 at this time. The start address of function inv_inst is set to vector number 32; the start address of function IRQ0 is set to vector number 64. These are, respectively, the user vector and external interrupt start vector numbers.

Table 1.4 Exception Processing Vector Table

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

The above written in assembly language is as follows:

Vector Table Creation Program (Assembly Language Version):

```
.SECTION   VECT,DATA,ALIGN=4
.IMPORT    _main
.IMPORT    _inv_inst
.IMPORT    _IRQ0
.DATA.L    _main                ;_ main start address set in vector 0
.DATA.L    H'0000000            ;SP initial value set in vector  1
.ORG       H'0080
.DATA.L    _inv_inst            ;_inv_inst start address set in vector 32
.ORG       H'0100
.DATA.L    _IRQ0                ;IRQ0 start address set in vector 64
.END
```

Add an underscore (_) to the beginning of the C program external names in the assembly language program.

1.6.2 Header File Creation

The header file used throughout the sample program is shown below. By defining I/O ports such as IPRA, those I/O ports can be accessed by name in the same manner as variables.

```
/*********************************************************************/
/*                        file name “7032.h”                         */
/*********************************************************************/
/*********************************************************************/
/*                    Definitions of I/O Registers                   */
/*********************************************************************/
struct st_intc {                                /*struct INTC          */
       union {                                  /*IPRA                 */
              unsigned short WORD;              /*       Word Access   */
              struct {                          /*       Bit Access    */
                     unsigned short UU:4;       /*       IRQ0          */
                     unsigned short UL:4;       /*       IRQ1          */
                     unsigned short LU:4;       /*       IRQ2          */
                     unsigned short LL:4;       /*       IRQ3          */
                     }          BIT;            /*                     */
              }          IPRA;                  /*                     */
       union  {                                 /*IPRB                 */
              unsigned short     WORD;          /*       Word Access   */
              struct{                           /*       Bit Access    */
                     unsigned short UU:4;       /*       IRQ4          */
                     unsigned short UL:4;       /*       IRQ5          */
                     unsigned short LU:4;       /*       IRQ6          */
                     unsigned short LL:4;       /*       IRQ7          */
                     }          BIT;            /*                     */
              }          IPRB;                  /*                     */
};
#define INTC      (*(volatile struct st_intc     *)0x5FFFF84)
                                                /*INTC          Address*/
```

```
/*****************************************************************/
/*                      Timer registers                          */
/*****************************************************************/
struct st_itu0 {                                /*struct  ITU0          */
       union  {                                 /*TCR                   */
              unsigned char BYTE;               /*      Byte Access     */
              struct {                          /*      Bit Access      */
                     unsigned char wk    :1;    /*                      */
                     unsigned char CCLR  :2;    /*      CCLR            */
                     unsigned char CKEG  :2;    /*      CKEG            */
                     unsigned char TPSC  :3;    /*      TPSC            */
                     }    BIT;                 /*                      */
              }          TCR;                   /*                      */
};
#define ITU0      (*(volatile struct st_itu0     *)0x5FFFF04)
                                                /*ITU0          Address*/
```

```
/************************************************************************/
/*                      PORT registers                                  */
/************************************************************************/
struct st_pa {                                  /*struct PA             */
       union {                                  /*PADR                  */
              unsigned short      WORD;         /*      Word Access     */
              struct {                          /*      Bit Access      */
                     unsigned short B15:1;      /*      Bit 15          */
                     unsigned short B14:1;      /*      Bit 14          */
                     unsigned short B13:1;      /*      Bit 13          */
                     unsigned short B12:1;      /*      Bit 12          */
                     unsigned short B11:1;      /*      Bit 11          */
                     unsigned short B10:1;      /*      Bit 10          */
                     unsigned short B9:1;       /*      Bit 9           */
                     unsigned short B8:1;       /*      Bit 8           */
                     unsigned short B7:1;       /*      Bit 7           */
                     unsigned short B6:1;       /*      Bit 6           */
                     unsigned short B5:1;       /*      Bit 5           */
                     unsigned short B4:1;       /*      Bit 4           */
                     unsigned short B3:1;       /*      Bit 3           */
                     unsigned short B2:1;       /*      Bit 2           */
                     unsigned short B1:1;       /*      Bit 1           */
                     unsigned short B0:1;       /*      Bit 0           */
                     }           BIT;           /*                      */
          }              DR;                    /*                      */
};                                              /*                      */
#define PB     (*(volatile struct st_pa   *)0x5FFFFC2)
                                                /*PB            Address*/
```

```
struct st_pc  {                                  /*struct PC              */
       union {                                   /*PCDR                   */
              unsigned short      WORD;          /*Word Access            */
              struct {                           /*Bit Access             */
                     unsigned short wk:8;        /*       Bit 8           */
                     unsigned short B7:1;        /*       Bit 7           */
                     unsigned short B6:1;        /*       Bit 6           */
                     unsigned short B5:1;        /*       Bit 5           */
                     unsigned short B4:1;        /*       Bit 4           */
                     unsigned short B3:1;        /*       Bit 3           */
                     unsigned short B2:1;        /*       Bit 2           */
                     unsigned short B1:1;        /*       Bit 1           */
                     unsigned short B0:1;        /*       Bit 0           */
                     }           BIT;            /*                       */
              }           DR;                    /*                       */
};                                               /*                       */
#define PC    (*(volatile struct st_pc   *)0x5FFFFD0)
                                                 /*PC            Address*/

/************************************************************************/
/*                    file name “sample.h”                              */
/************************************************************************/
/************************************************************************/
/*                        Timer registers                               */
/************************************************************************/
struct     tcsr   {
           short OVF  :1;                        /*TCSR struct OVF bit    */
           short WTIT:1;                         /*WTIT bit               */
           short      :3;                        /*work area              */
           short CKS2:1;                         /*CKS2 bit               */
           short CKS1:1;                         /*CKS1 bit               */
           short      :9;                        /*work area              */
};

#define TCSR_FRT   (*(volatile unsigned short *)0x5FFFFB8)
#define TCSR__FRT  (*(volatile struct tcsr*)0x5FFFFB8)
```

```
extern void motor( void );                    /*motor module
*/
extern void _INITSCT ( void );                /*section initialize module
*/
extern void init_peripheral ( void );         /*peripheral initialize module*/
```

1.6.3 Main Processing Module Creation

The main processing program is shown below. The function main, activated by a power on reset, and the function motor, called continuously until an interrupt occurs, are defined.

```
/*********************************************************************/
/*                        file name "sample.c"                       */
/*********************************************************************/
#include "7032.h"
#include "sample.h"
#include <machine.h>                      /*Defines the intrinsic function sleep*/
const short padata=0x3                    /*C section                           */
short a=0;                                /*D section                           */
int work;                                 /*B section                           */

/*********************************************************************/
/*                          main module                              */
/*********************************************************************/
void main(void)
{
       _INITSCT();                        /*Initialization of each section*/
       init_peripheral();
       while(!a)     motor();
       sleep();
}
```

```
/*********************************************************************/
/*                        motor module                               */
/*********************************************************************/
void motor (void)                          /*Called until an interrupt occurs*/
{
            .
            .
            .
            .
       return;
}
```

In the function main, _INITSCT and init_peripheral are called and the sections and internal registers are initialized, then a wait occurs until the value of global variable a is modified. The function motor is called continuously during that interval. Low power state is entered when the value of a becomes anything other than 0.

1.6.4 Initializing Module Creation

The assembly language program for setting the values of external names used in section initialization is shown below.

```
;*********************************************************************/
;                        file name “sct.src”                          */
;*********************************************************************/
              .SECTION B,DATA.ALIGN=4
              .SECTION R,DATA,ALIGN=4
              .SECTION D,DATA,ALIGN=4
When sections are to be added, describe them here.
```

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

The B section and D section start and end addresses are defined. When section names are not designated by the section option at the time of compilation, the C compiler assigns the various names as follows:

- Program area section: P
- Constant area section: C
- Initialized data area section: D
- Uninitialized data area section: B

The R section indicates the RAM area for copying the initialized data area in ROM using the ROM conversion support function of the linkage editor. Refer to section 4.2.1, ROM Conversion Support Function, for details on the ROM conversion support function of the linkage editor.

STARTOF is an operator for obtaining the section collective start address, using the “STARTOF<section name>” description. SIZEOF is an operator for obtaining the section collective size in byte units, using the “SIZEOF<section name>” description.

The C language program that performs the section and register initializations is shown below.

```
/*********************************************************************/
/*                        file name “init.c”                         */
/*********************************************************************/
#include “7032.h”
#include “sample.h”
/*********************************************************************/
/*                    section initialize module                      */
/*********************************************************************/
extern int *_B_BGN,*_B_END,*_D_BGN,*_D_END,*_D_ROM
```

```
void _INITSCT(void)
{
        register int *p,*q;

        for (p=_B_BGN; p<_B_END; p++)
            *p=0;
        for (p=_D_BGN,q=_D_ROM; p<_D_END; p++,q++)
            *p=*q;
}

/*********************************************************************/
/*                    peripheral initialize module                   */
/*********************************************************************/
void init_peripheral(void)
{
        INTC→IPRA.WORD = 0x3000                    /*IPRA initialization*/
        ITU0→TCR.BYTE = 0x02                       /*TCR0 initialization*/
        TCSR_FRT = 0x5A01                          /*TCSR initialization*/
        PB→DR.WORD = 0x80                          /*PORT initialization*/
}
```

The section initialization module _INITSCT performs a zero clear of the B section based on the section address designated by sct.src, and copies the initialized data area in ROM into RAM. The format designator used is int, but char should be used when the size is any other than 4n bytes.

The internal register initialization module init_peripheral performs each of the following settings:

- The IRQ0 interrupt priority level is set to 3 in interrupt priority level setting register A
- Clear prohibition of 16-bit integrated timer pulse unit timer counter 0, counting on the rising edge, and counting by $\phi/4$ of the internal clock are set in timer control register 0
- Timer counter of the watchdog timer is set to 0×01
- Port B is set to 0×80

1.6.5 Interrupt Function Creation

The interrupt function is shown below. The external interrupt processing function IRQ0 and the trap instruction function inv_inst are defined.

```
/*********************************************************************/
/*                        file name “int.c”                          */
/*********************************************************************/
#include “7032.h”
#include “sample.h”
extern const short padata;                      /*C section          */
extern short a;                                 /*D section          */
extern int work;                                /*B section          */
#pragma interrupt(IRQ0, inv_inst)

/*********************************************************************/
/*                        interrupt module IRQ0                      */
/*********************************************************************/
void IRQ0(void)
{
       a = PB→DR.WORD;
       PC→.DR.WORD = padata;
}

/*********************************************************************/
/*                    interrupt module inv_inst                      */
/*********************************************************************/
void inv_inst(void)
{
       return;
}
```

The function IRQ0 sets PB→DR.WORD (0×80) in the global variable a when an IRQ0 external interrupt occurs. This causes the CPU to enter the low power state.

1.6.6 Load Module Batch File Creation

The batch file for creating an S-type format load module (sample.mot) is shown below.

shc	-debug sample.c init.c int.c	Compile the C language program
asmsh	sct.src -debug	Assemble assembly language program
shc	-debug -section=c=VECT vect.c	Compile vector table creation program
lnk	-subcommand=rom.sub	Link using the subcommand file
cnvs	sample.abs	Create S-type format load module
rm	*.obj *.abs	Delete temporary file

The program vect.c is compiled as an independent file and becomes a section different from the other initialized data areas with option section=VECT attached. During linkage it is allocated from address 0.

1.6.7 Linkage Editor Subcommand File Creation

The linkage editor subcommand file (file name: rom.sub) used during load module creation is shown below.

```
debug
input     sample,init,int,vect,sct                ;Designate input files
library   /user/unix/SHCV3.0/shclib.lib           ;Designate standard  libraries
output    sample.abs                              ;Designate output file  name
rom       (D,R)                                   ;Designate ROM conversion
                                                  support option
start     VECT(0),P,C,D(0400),R,B,(0F000000)      ;Designate start address
                                                  of each section
                                                  ;Allocate section VECT
                                                  from address 0
                                                  ;Allocate sections P,C,D
                                                  in order from address  H'400
                                                  ;Allocate sections R,B in
                                                  order from address
                                                  H'0F000000
form      a                                       ;Designate absolute format
print     sample,map                              ;Designate output of
                                                  memory map information
exit
```

Section 2 Functions

This section describes the SH series C compiler extended functions and specific programming techniques for software intrinsic to the individual machines. The assembly language code is obtained through the following command line:

```
shcΔ<C language file>Δ-code=asmcode
```

The assembly language code may change in the future due to improvements in the compiler.

2.1 Interrupt Functions

2.1.1 Interrupt Function Definition (Without Options)

Interrupt functions can be created in C using the preprocessor control statement #pragma. Functions declared with “#pragma interrupt” save/restore all registers (excepting the global base register GBR and vector base register VBR) used within the function before and after the function processing. For this reason, it is not necessary to provide interrupt processing for interrupted functions.

Description:

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

Example: The interrupt function handler1 is declared. This function operates using the stack handed over from the interrupted function and returns with an RTE instruction after processing is completed.

C Language Code when GBR, VBR Are Not Saved/Restored:

```
#pragma interrupt(handler1)                /* Interrupt function declaration     */

        void handler1(void)
        {
            :                              /* Interrupt function processing      */
        }
```

Assembly Language Code:

```
                .EXPORT     _handler1
                .SECTION    P,CODE,ALIGN=4
handler1:                           ;function: handler1
                .                   ;Save registers used in processing
                .                   ;Interrupt function processing
                .                   ;Restore registers used in processing
                RTE
                NOP
                .END
```

C Language Code when GBR, VBR Are Saved/Restored:

```
#pragma interrupt(handler1)
void handler1(void)
{
    void** save_vbr;            /*VBR save area                 */
    void* save_gbr;             /*GBR save area                 */

    save_vbr = get_vbr();       /*Save VBR                      */
    save_gbr = get_gbr();       /*Save GBR                      */
           :                    /*Interrupt function processing */
    set_vbr(save_vbr);          /*Restore VBR                   */
    set_gbr(save_gbr);          /*Restore GBR                   */
}
```

Assembly Language Code:

```
                .EXPORT         handler1
                .SECTION        P,CODE,ALIGN=4
_handler1:                                      ;function: handler3
                                                ;frame size=16
                MOV.L           R5,@-R15
                STC             GBR,R5          ;Save GBR
                MOV.L           R4,@-R15
                STC             VBR,R4          ;Save VBR
                :                               ;Save registers used in processing
                :                               ;Interrupt function processing
                :                               ;Restore registers used in
                                                processing
                LDC             R4,VBR          ;Restore VBR
                LDC             R5,GBR          ;Restore GBR
                MOV.L           @R15+,R4
                MOV.L           @R15+,R5
                RTE
                NOP
L211:
                .DATA.W         H'FF0F
                .END
```

Precautions:

1. The only data format returned by interrupt functions is the void format. Example:

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

   The interrupt function f1 definition is correct, but the interrupt function f2 definition results in an error.
2. The only memory class designator that can be designated in interrupt function definitions is extern. Even if static is designated, the processing will be as extern.
3. Functions declared as interrupt functions cannot be called as ordinary functions. Operation during execution cannot be guaranteed when functions declared as interrupt functions are called from ordinary functions.

Example:

- test1.c file contents

```
#pragma interrupt(f1)                    /*interrupt function declaration*/
void    f1(void){...}                    /*interrupt function f1 definition*/
int     f2(){  f1(); }
```

- test2.c file contents

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

In the test1.c file, an error results in function f2. In the test2.c file, no error results in function f3, but function f1 is interpreted as being extern int f1() and operation during execution becomes undefined.

4. Operation upon an interrupt is different with the SH-3 from SH-1 and SH-2; an interrupt handler is required. An example of an interrupt handler is given below.

```
;*********************************************************************;
;   SH-3 Interrupt Starter Routine
;*********************************************************************;
        .SECTION        inthandl,CODE,ALIGN=4
        .ORG            H'600
        .EXPORT         __int_start
        .EXPORT         __int_term
__int_start:
        STC.L           SSR,@-R15           ; save ssr
        STC.L           SPC,@-R15           ; save spc
;
        MOV.L           R8,@-R15            ; save work register
        ADD             #-4,R15             ; sr stack area
        MOV.L           R0,@-R15            ; save work register
        MOV.L           R1,@-R15            ; save work register
        MOV.L           R2,@-R15            ; save work register
;
        MOV.L           INTEVT,R0           ; set INTEVT address to r0
        MOV.L           @R0,R1              ; set exception code to r1
        MOVA            vcttbl,R0           ; set vector table address to r0
        SHLR2           R1                  ; 3-bit shift-right exception code
```

```
SHLR        R1
ADD         #-(h'1c0>>3),R1     ; exception code –h'1c0
MOV.L       @(R0,R1),R8         ; set interrupt function addr to r8

MOVA        imasktbl,R0         ; set interrupt mask table addr to r0
SHLR2       R1                  ; 2-bit shift-right exception code
MOV.B       @(R0,R1),R1         ; set interrupt mask to r1
EXTU.B      R1,R1

STC         SR,R0               ; save sr to r0
LDC         R0,SSR              ; set current status to ssr
MOV.L       IMASKclr,R2         ; set IMASK clear data to r1
AND         R2,R0               ; clear interrupt mask
OR          R1,R0               ; set interrupt mask
MOV.L       RBBLclr,R1          ; set RB,BL clear data to r1
AND         R1,R0               ; (RB = BL = 0)
MOV.L       R0,@(12,R15)        ; push sr
;
MOVA        __int_term,R0       ; set __int_term addr to spc
LDC.L       R0,SPC
;
MOV.L       @R15+,R2            ; restore work register
MOV.L       @R15+,R1            ; restore work register
MOV.L       @R15+,R0            ; restore work register
LDC.L       @R15+,SR            ; restore sr
JMP         @R8                 ; jump to interrupt function
MOV.L       @R15+,R8            ; restore work register
;
```

```
;*********************************************************************;
;   SH-3 Interrupt Terminator Routine
;*********************************************************************;
            .ALIGN 4
__int_term
            LDC.L       @R15+,SPC           ; load spc
            LDC.L       @R15+,SSR           ; load ssr
            RTE                             ; rte
            NOP
;
            .ALIGN      4
RBBLclr     .DATA.L     H'4FFFFFFF
IMASKclr    .DATA.L     H'FFFFFF0F
INTEVT      .DATA.L     H'FFFFFFD8
;
vcttbl                                      ; Interrupt Vector Table
            .DATA.L     H'00000000          ; NMI
            .DATA.L     H'00000000          ; IRL = 0
            .DATA.L     H'00000000          ; IRL = 1
;                       :
;                       :
            .RES.L      26
;                       :
;                       :
            .DATA.L     H'00000000          ; RCVI
;
imasktbl                                    ; Interrupt Mask Table
            .DATA.B     H'F0                ; NMI
            .DATA.B     H'F0                ; IRL = 0
            .DATA.B     H'E0                ; IRL = 1
;                       :
;                       :
            .RES.B      26
;                       :
;                       :
            .DATA.B     H'00                ; RCVI
            .END
```

Note: Make the interrupt priority rank of the imasktbl on-chip peripheral module the same as that established by the interrupt level setting registers A–B (IPRA–IPRB).

2.1.2 Interrupt Function Definition (With Options)

The interrupt function definition options consist of the stack switching designation and the trap instruction return designation. With the stack switching designation, the stack pointer is switched to a designated address when an external interrupt occurs, and the interrupt function operates using this stack. The stack pointer before the interrupt occurrence is returned to upon restoration after the interrupt routine. It becomes unnecessary to secure any extra interrupt function stack beforehand for functions interrupted with this designation.

With the trap instruction return designation, the return is performed with a TRAPA instruction. When not designated, the return is by an RTE instruction.

Description:

```
#pragma interrupt
(<function_name>[(<interrupt_specification>)][,<function_name>[
(<interrupt_specification>)]...])
```

Table 2.1 Interrupt Specification List

Item	Format	Option	Contents of Designation
Stack switching designation	sp=	<variable> | & <variable> | <constant>	New stack address designated with a variable or constant <variable>: variable (object type) value & <variable>: variable (pointer type) address <constant>: constant value
Trap instruction return designation	tn=	<constant>	End designated with a TRAPA instruction <constant>: constant value (trap vector number)

The interrupt function handler2 is declared. This function uses the array STK as a stack and returns with a “TRAPA #63” instruction after completion of processing.

C Language Code:

```
extern int STK[100];
int *ptr = STK + 100;
#pragma interrupt(handler2(sp=ptr,tn=63))      /*Interrupt function  declaration*/
void handler2(void)
{
    :                                          /*Interrupt function processing description*/
}
```

Assembly Language Code:

```
                .IMPORT         _STK
                .EXPORT         _ptr
                .EXPORT         _handler2
                .SECTION        P,CODE,ALIGN=4
_handler2                                      ; function: handler2
                MOV.L           R0,@-R15
                MOV.L           L211,R0
                MOV.L           @R0,R0
                MOV.L           R15,@-R0
                MOV             R0,R15
                :                              ; Save registers used in  processing
                :                              ; Interrupt function processing
                :                              ; Restore registers used in
                                               processing
                MOV.L           @R15+,R15
                MOV.L           @R15+,R0
                TRAPA           #63
L211:
                .DATA.L         _ptr
                .SECTION        D,DATA,ALIGN=4
_ptr:                                          ; static: ptr
                .DATA.L         H'00000190+_STK
                .END
```

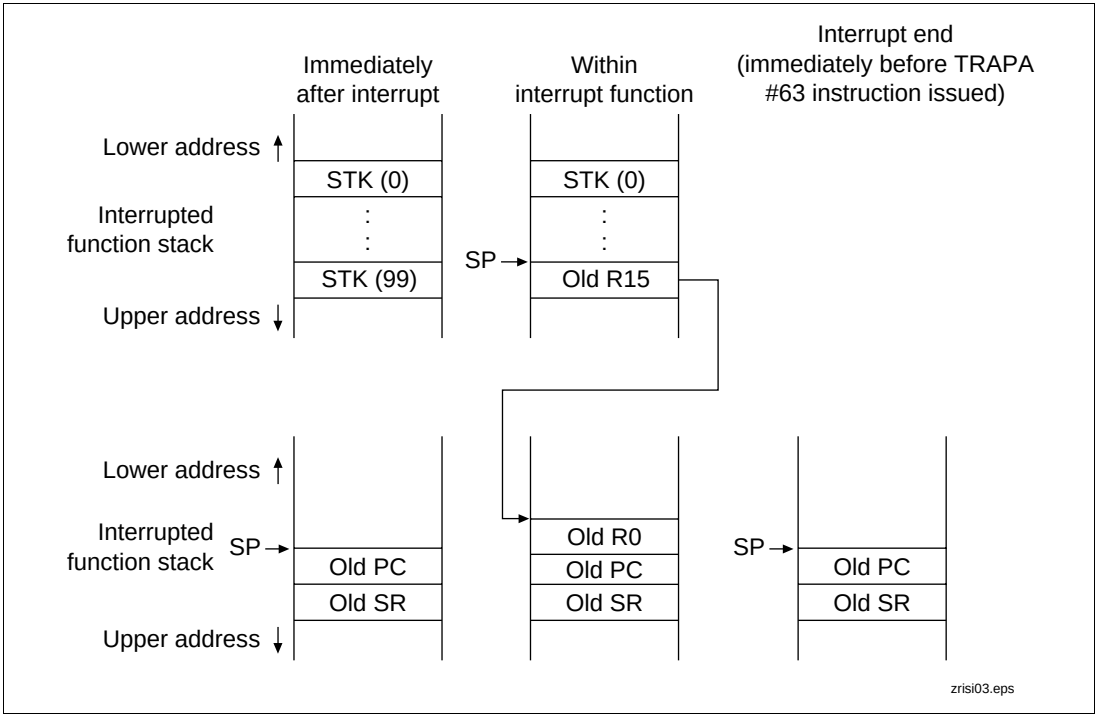


Figure 2.1 Example of Stack Use by an Interrupt Function

2.1.3 Vector Table Creation

Vector tables can be created in C as follows:

1. Provide a vector table usage array and designate an exception processing function pointer for each element.
2. After compiling this file, designate and link the vector table start address.

C Language Code: vect_table.c:

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

Batch File:

```
shc -section=c=VECT vect_table
shc reset warm_reset irq0 irq1...
lnk vect_table,reset,warm_reset,irq0,irq1,...-output=sample.abs-
start=VECT(0),P,C,D(0400),B(0F000000)
```

Compiling vect_table.c generates the relocatable object file vect_table.obj for the initialized data section (VECT) only.

The section VECT is designated with a start address of H'0 and linked along with the other files, and the load module sample.abs is obtained.

Assembly Language Code: vect.table.src:

```
                .IMPORT         _reset
                .IMPORT         _warm_reset
                .IMPORT         _irq0
                .IMPORT         _irq1
                .EXPORT         _vect_table
                .SECTION        VECT,DATA,ALIGN=4
_vect_table:                                            ;static: vect_table
                .DATA.L         _reset
                .DATA.L         H'00000000
                .DATA.L         _warm_reset
                .DATA.L         H'00000000
                    :
                    :
                .DATA.L         _irq0,irq1
                    :
                    :
                .END
```

Precautions:

1. Operation upon an interrupt for the SH-3 is different from SH-1 and SH-2 in that a vector table is not used and an interrupt handler is necessary.
2. Since the vector table must be allocated to a fixed absolute address, it was created here as an independent file, but by using the section switching function it is possible to make it a file identical to that of other modules. Refer to section 2.7, Section Name Designation, for details.

2.2 Intrinsic Functions

The intrinsic functions indicated in table 2.2 are provided to enable C language description of the instructions inherent to the SH-1, SH-2, and SH-3. The standard header file “machine.h” must be included when using intrinsic functions. Also, “machine.h” is partitioned in response to the SH-3 execution mode for each function that can be used with the respective mode. Include “smachine.h” when using functions usable only when in privileged mode, and “umachine.h” when using all other functions.

Table 2.2 Intrinsic Function List

Item	Function	Usable Execution Mode (SH-3)
Status register (SR)	SR setting	Privileged mode only
	SR referencing	
	Interrupt mask setting	
	Interrupt mask referencing	
Vector base register (VBR)	VBR setting	Privileged mode only
	VBR referencing	
Global base register (GBR)	GBR setting	No restrictions
	GBR referencing	
	GBR-base byte referencing	
	GBR-base word referencing	
	GBR-base longword referencing	
	GBR-base byte setting	
	GBR-base word setting	
	GBR-base longword setting	
	GBR-base byte AND	
	GBR-base byte OR	
	GBR-base byte XOR	
	GBR-base byte TEST	
System control	SLEEP instruction	Privileged mode only
	TAS instruction	No restrictions
	TRAPA instruction	
Multiply/accumulate operation	Word multiply/accumulate	No restrictions
	Longword multiply/accumulate	
	Ring buffer corresponding word multiply/accumulate	
	Ring buffer corresponding longword multiply/accumulate	
System call	System call execution	No restrictions
Prefetch instruction	Prefetch instruction	No restrictions

2.2.1 Status Register Setting/Referencing

The functions indicated in table 2.3 are provided for status register setting/referencing.

Table 2.3 Status Register Usage Intrinsic Functions

Item	Description	Explanation
Status register setting	void set_cr (int cr)	Sets cr (32 bit) in the status register
Status register referencing	int get_cr (void)	References the status register
Interrupt mask setting	void set_imask(int mask)	Sets mask (4 bit) in the interrupt mask (4 bit)
Interrupt mask referencing	int get_imask(void)	References the interrupt mask (4 bit)

The function func1 performs processing after prohibiting external interrupts by setting the interrupt mask to its maximum (15). After completion of processing, the original interrupt mask level is restored and the function ends.

C Language Code:

```
include <machine.h>

void   func1(void)
{
    int        mask;        /*Interrupt mask level storage location      */

    mask = get_imask();     /*Store the interrupt mask level             */
    set_imask(15);          /*Set the interrupt mask level to 15         */
        :                   /* Prohibit interrupts and execute  processing*/
        :
    set_imask(mask);        /* Restore the interrupt mask level          */
    }
```

Assembly Language Code:

```
        .EXPORT         _func1
        .SECTION        P,CODE,ALIGN=4
_func1                                  ; function: func1
        MOV.W           L210,R3
        STC             SR,R0
        SHLR2           R0
        SHLR2           R0
        AND             #15,R0
        MOV             R0,R4
        STC             SR,R0
        AND             R3,R0
        OR              #240,R0
        LDC             R0,SR
            :
            :
        MOV             R4,R0
        AND             #15,R0
        SHLL2           R0
        SHLL2           R0
        STC             SR,R2
        MOV             R3,R1
        AND             R1,R2
        OR              R2,R0
        LDC             R0,SR
        RTS
        NOP
L210:
        .DATA.W         H'FF0F
        .END
```

2.2.2 Vector Base Register Setting/Referencing

The functions indicated in table 2.4 are provided for vector base register setting/referencing.

Table 2.4 Vector Base Register Usage Intrinsic Functions

Item	Description	Explanation
Vector base register setting	void set_vbr (void **base)	Sets **base (32 bit) in the vector base register
Vector base register referencing	void **get_vbr (void)	References the vector base register

The vector base register (VBR) is initialized to 0 by a reset. When the vector table starts from an address other than address 0, if the next function is established in the start address (H'00000008) for a manual reset and a manual reset occurs at the time of system startup, exception processing can be executed using the established vector table.

C Language Code:

```
#include <machine.h>
#define VBR    0x0000FC00          /*Vector table start address          */

void   warm_reset(void)
{
       set_vbr((void**)VBR);
                                   /*Vector the vector base  register    */
                                   /*Establish in the table's start address*/
}
```

Assembly Language Code:

```
            .EXPORT         _warm_reset
            .SECTION        P,CODE,ALIGN=4
_warm_start:                                    ; function: warm_reset
            MOV.L           L209,R3
            LDC             R3,VBR
            RTS
            NOP
L209:
            .DATA.L         H'0000FC00
            .END
```

Precautions: Perform modifications of the vector base register after establishing the vector table. If this order is reversed, an external interrupt occurrence during vector table establishment will cause a system failure.

2.2.3 Accessing I/O Registers (1)

The functions in table 2.5 are provided for global base register (GBR) manipulation to allow access to I/O registers.

Table 2.5 Global Base Register Usage Intrinsic Function

Item	Description	Explanation
Global base register setting*2	void set_gbr(void *base)	Sets *base (32 bit) in the global base register
Global base register referencing*2	int *get_gbr(void)	References the global base register
Global base register base byte referencing*2	unsigned char gbr_read_byte(int offset)	References global base register relative offset byte data (8 bit)
Global base register base word referencing*2	unsigned short gbr_read_word(int offset)	References global base register relative offset word data (16 bit)
Global base register base longword referencing*2	unsigned long gbr_read_long(int offset)	References global base register relative offset longword data (32 bit)
Global base register base byte setting*2	void gbr_write_byte(int offset, unsigned char data)	Sets data (8 bit) in the global base register relative offset
Global base register base word setting*2	void gbr_write_word(int offset, unsigned short data)	Sets data (16 bit) in the global base register relative offset
Global base register base longword setting*2	void gbr_write_long(int offset, unsigned long data)	Sets data (32 bit) in the global base register relative offset
Global base register base byte AND	void gbr_and_byte(int offset, unsigned char mask)	Takes the AND of the global base register relative offset byte data and mask, sets it in offset
Global base register base byte OR	void gbr_or_byte(int offset, unsigned char mask)	Takes the OR of the global base register relative offset byte data and mask, sets it in offset
Global base register base byte XOR	void gbr_xor_byte(int offset, unsigned char mask)	Takes the XOR of the global base register relative offset byte data and mask, sets it in offset
Global base register base byte TEST	int gbr_tst_byte(int offset, unsigned char mask)	Takes the AND of the global base register relative offset byte data and mask; judges that value as 0. Result is set in the T bit

Notes: 1. Establish base as a multiple of 2 when the access size is word, and as a multiple of 4 when the access size is longword.

2. The offset must be a constant for these items. The range that can be designated for offset is +255 bytes when the access size is byte, +510 bytes when it is word, and +1020 bytes when it is longword.
3. The mask must be a constant. The range that can be designated for mask is 0 to +255.
4. The global base register is a control register, so saving and restoring of values at function entrances and exits is not executed by the C compiler. When modifying the global base register value, the user must carry out the save/restore of the value at the function entrance/exit.

The following is an example of a timer driver using the SH7034 on-chip 16 bit integrated timer pulse unit.

C Language Code:

```
#include <machine.h>
#define IOBASE 0x05fffec0           /*I/O base address                    */
#define TSR       (0x05ffff07 - IOBASE)
                                    /*Timer status flag register offset  address*/
#define TSRCLR (unsigned char)0xf8
                                    /*Timer status flag register clear value*/


void    tmrhdr(void)
{
    void       *gbrsave;            /*Global base register value storage
                                    location*/
    gbrsave = get_gbr();            /*Store the global base register value*/
    set_gbr((void*)IOBASE);         /*Set the I/O base address in the global
                                    register*/
    gbr_read_byte(TSR);             /*Dummy read to clear the timer status
                                    flag register*/
    gbr_and_byte(TSR, TSRCLR);      /*Clear the timer status flag register
                                    compare match flag*/
    set_gbr(gbrsave);               /*Restore the global base register value*/
}
```

Assembly Language Code:

```
                .EXPORT         _tmrhdr
                .SECTION        P,CODE,ALIGN=4
_tmrhdr:                                        ;function: tmrhdr
                MOV.L           L210,R3
                STC             GBR,R4
                LDC             R3,GBR
                MOV.B           @(71,GBR),R0
                MOV             #71,R0
                AND.B           #248,@(R0,GBR)
                RTS
                LDC             R4,GBR
L210:
                .DATA.L         H'05FFFEC0
                .END
```

2.2.4 Accessing I/O Registers (2)

Use of the standard library offsetof eliminates the need to calculate the value of the global base register relative offset beforehand.

C Language Code:

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

void    f(void)
{
        void          *gbrsave;     /*Global base register value storage
                                    location*/
        gbrsave = get_gbr();        /*Store the global base register value*/
        set_gbr(&table);
                                    /*Set the table start address in the global
                                    base register*/
            :
            :
        gbr_and_byte(offsetof(struct IOTBL, cdata2),0x10);
                                    /*Take the AND of the table.cdata2 value
                                    and 0x10 and set it in table.cdata2   */
            :
            :
        set_gbr(gbrsave);           /*Restore the global base register value*/
}
```

Assembly Language Code:

```
                .EXPORT         _table
                .EXPORT         _f
                .SECTION        P,CODE,ALIGN=4
_f:                                             ; function: f
                MOV.L           L211+2,R3
                MOV             #1,R0
                STC             GBR,R4
                LDC             R3,GBR
                    :
                    :
                AND.B           #16,@(R0,GBR)
                    :
                    :
                RTS
                LDC             R4,GBR
L211:
                .RES.W          1
                .DATA.L         _table
                .SECTION        B,DATA,ALIGN=4
_table:                                         ; static: table
                .RES.L          4
                .END
```

2.2.5 System Control

The functions indicated in table 2.6 are provided as Hitachi SuperH RISC family engine dedicated special instructions.

Table 2.6 Special Instruction Usage Intrinsic Functions

Item	Description	Explanation
SLEEP instruction	void sleep(void)	Compiles to the SLEEP instruction
TAS instruction	void tas(char *addr)	Compiles to TAS.B @addr
TRAPA instruction	void trapa(int trap_no)	Compiles to TRAPA #trap_no

Notes: 1. The trap_no in the table must be a constant.
2. The trapa intrinsic function activates an interrupt function from the C program. Create the called function as an interrupt function.

In the following example, a SLEEP instruction is issued and the CPU is placed in the low power state. In the low power state, execution of the next instruction is halted and the internal status of the CPU is maintained while the occurrence of an interrupt request is awaited. Low power state is exited when an interrupt occurs.

C Language Code:

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

Assembly Language Code:

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

2.2.6 Multiply/Accumulate Operations (1)

The functions indicated in table 2.7 are provided for multiply/accumulate operations.

Table 2.7 Multiply/Accumulate Operation Usage Intrinsic Functions

Item	Description	Explanation
Word multiply/accumulate	int macw(short *ptr1, short *ptr2, unsigned int count)	Multiply/accumulate word data *ptr1 (16 bit) with word data *ptr2 (16 bit) number of times indicated by count
Longword multiply/accumulate	int macl(int *ptr1, int *ptr2, unsigned int count)	Multiply/accumulate longword data *ptr1 (32 bit) with longword data *ptr2 (32 bit) number of times indicated by count

The word multiply/accumulate function macw is supported by SH-1, SH-2, and SH-3, but the longword multiply/accumulate function macl is only supported by SH-2 and SH-3.

The multiply/accumulate operation intrinsic function does not perform an argument check. Adjust both of the data tables on which multiply/accumulate operations are performed so that the boundaries are 2-byte for word multiply/accumulate functions and 4-byte for longword multiply/accumulate functions.

In the example below, the multiply/accumulate operation is performed. When the number of multiply/accumulate operation executions is 32 times or fewer, they are realized by repeating the MAC instruction, but when the number is 33 times or more, or else when the number of iterations is a variable, they are realized with a MAC instruction loop.

C Language Code:

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

void   func(void)
{                                       a[0] * b[0]
          :                       +     a[1] * b[1]
          :                       +     a[2] * b[2]
   macw(a,b,SIZE);                +       :     :
          :                       + a[SIZE-2] * b[SIZE-2]
          :                       + a[SIZE-1] * b[SIZE-1]
}
```

Assembly Language Program:

- For SIZE ≤ 32: Repeat the MAC instruction

```
            .EXPORT     _func
            .SECTION    P,CODE,ALIGN=4
_func:                                          ; function: func
            STS.L       MACH,@-R15
            STS.L       MACL,@-R15
                :
                :
            MOV.L       L211+2,R3
            CLRMAC
            MOV.L       L211+6,R2
            MAC.W       @R2+,@R3+               ;Repeat according to SIZE
                :
                :
            STS         MACL,R0
            LDS.L       @R15+,MACL
            RTS
            LDS.L       @R15+,MACH
L211:
            .RES.W      1
            .DATA.L     _b
            .DATA.L     _a
            .END
```

- For SIZE > 32, or variable: Realize through a MAC instruction loop

```
            .EXPORT     _func
            .SECTION    P,CODE,ALIGN=4
_func:                                          ; function: func
            STS.L       MACH,@-R15
            MOV         #SIZE,R3
            STS.L       MACL,@-R15
                :
                :
            TST         R3,R3
            CLRMAC
            BT          L211
            MOV.L       L213+2,R2
            SHLL        R3
            MOV.L       L213+6,R1
            ADD         R1,R3
L212:
            MAC.W       @R1+,@R2+
            CMP/HI      R1,R3
            BT          L212
L211:
            STS         MACL,R0
                :
                :
            LDS.L       @R15+,MACL
            RTS
            LDS.L       @R15+,MACH
L213:
            .RES.W      1
            .DATA.L     _b
            .DATA.L     _a
            .END
```

2.2.7 Multiply/Accumulate Operations (2)

The functions indicated in table 2.8 are provided for multiply/accumulate operations which correspond to the link buffer.

Table 2.8 Link Buffer Related Multiply/Accumulate Operation Intrinsic Functions

Item	Description	Explanation
Link buffer related word multiply/accumulate	int macwl(short *ptr1, short *ptr2,unsigned int count, unsigned int mask)	Multiply/accumulate word data *ptr1 (16 bit) with the word data *ptr2 (16 bit) designated by mask number of times indicated by count
Link buffer related longword multiply/accumulate	int macll(int *ptr1, int *ptr2,unsigned int count, unsigned int mask)	Multiply/accumulate longword data *ptr1 (32 bit) with the longword data *ptr2 (32 bit) designated by mask number of times indicated by count

The link buffer related word multiply/accumulate function macwl is supported by SH-1, SH-2, and SH-3, but the link buffer related longword multiply/accumulate function macll is only supported by SH-2 and SH-3.

The link buffer related multiply/accumulate operation intrinsic function does not perform an argument check.

Use 2-byte boundaries when the first argument is a word multiply/accumulate function, 4-byte when it is a longword multiply/accumulate function, and make the second argument twice that of the link buffer size.

In the example below, the link buffer related multiply/accumulate operation is performed. Because the second argument must be adjusted so that the boundary is twice that of the link buffer size, it is treated as a separate file.

C Language Source Code: macwl.c:

```
    #include <machine.h>
    short a[SIZE];
    extern short b[];                                   a[0] * b[0]
                                                  +     a[1] * b[1]
    void   func(void)                             +           :
:
    {                                             +     a[7] * b[7]
            :                                     +     a[8] * b[0]
            :                                     +     a[9] * b[1]
       macwl(a,b,SIZE,-0x10);                     +     a[15] * b[7]
                                                  +           :
:
            :                                     + a[SIZE-8] * b[0]
            :                                     + a[SIZE-7] * b[1]
    }                                             +           :
:
                                                  + a[SIZE-1] * b[7]
```

Assembly Language Source Code: buffer.src:

```
        .EXPORT     _b
        .SECTION    B,DATA,ALIGN=32
_b:                                     ; static: b
        .RES.W      8
        .END
```

Assembly Language Code: macwl.src:

```
        .IMPORT     _b
        .EXPORT     _a
        .EXPORT     _func
        .SECTION    P,CODE,ALIGN=4
_func:                                  ; function: func
        STS.L       MACH,@-R15
        MOV         #SIZE,R3
        STS.L       MACL,@-R15
            :
            :
        TST         R3,R3
        CLRMAC
        BT          L211
        MOV.L       L213+2,R1
        SHLL        R3
        MOV.L       L213+6,R4
        MOV         #-17,R2
        ADD         R4,R3
L212:
        MAC.W       @R4+,@R1+
        AND         R2,R1
        CMP/HI      R4,R3
        BT          L212
```

```
L211:
        STS         MACL,R0
            :
            :
        LDS.L       @R15+,MACL
        RTS
        LDS.L       @R15+,MACH
L213:
        .RES.W      1
        .DATA.L     _b
        .DATA.L     _a
        .SECTION    B,DATA,ALIGN=4
_a:
        .RES.W      SIZE
        .END
```

2.2.8 System Call

The description of the intrinsic function that allows issuance of system calls from C programs is noted below. The number of arguments for system calls is variable from 0 to 4.

Description:

```
ret=trapa_svc(int trap_no, int code,
                    [type1 p1[, type2 p2[, type3 p3[, type4 p4]]]])
```

- trap_no: trap number (designated by a constant)
- code: function code, allocated to R0
- p1: first argument, allocated to R4
- p2: second argument, allocated to R5
- p3: third argument, allocated to R6
- p4: fourth argument, allocated to R7
- type1–type4: argument formats are general integer format ([unsigned]char, [unsigned]short, [unsigned]int, [unsigned]long), or else pointer format

In the example below, a system call of an OS that can be designated by trap number 63 is issued using this function.

C Language Code:

```
#include <machine.h>
#define SIG_SEM 0xffc8

void   func(void)
{
          :
          :
       trapa_svc(63, SIG_SEM, 0x05);
          :
          :
}
```

Assembly Language Code:

```
                .EXPORT     _func
                .SECTION    P,CODE,ALIGN=4
_func:                                          ; function: func
                   :
                   :
                MOV.L       L209+2,R0
                MOV         #5,R4
                TRAPA       #63
                   :
                   :
                RTS
                NOP
L209:
                .RES.W      1
                .DATA.L     H'0000FFC8
                .END
```

2.2.9 Prefetch Instruction

The description of the intrinsic function that performs prefetches of the cache for the SH-3 is noted below. This intrinsic function is effective only when -cpu=sh3 is designated.

Description:

```
void prefetch(void *p1)
```

p1: address for which prefetch is performed

C Language Code:

```
#include <umachine.h>
int a[1200];

f ()
{
        int *pa = a;
            :
            :
        prefetch(pa+8);
            :
            :
}
```

Assembly Language Code:

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

2.3 Inline Expansion

2.3.1 Inline Expansion of Functions

The function inline expansion capability is used to increase program execution speed. Ordinarily, function calls take the form of a branch to a section with a series of processes and implementing that processing. However, with this capability, the function processing is inserted at the function call position and the branch section instructions are deleted to increase the speed. This can have a large effect, particularly when functions called from within loops are expanded.

There are two types of inline expansion of functions, as follows:

1. Automatic Inline Expansion:
   When the -speed option is designated during compilation, automatic inline development of functions takes effect and small functions are automatically expanded. The size of functions to be expanded can be designated with the -inline option to more precisely control the automatic inline development. Moreover, the node count (the number of variable, operator statements excepting the declaration section) designates the function size (the -inline option default value is 20).

Description:

```
shc -speed[-inline=<node count>]...
```

2. Inline Expansion by a Control Statement:
   Functions to be expanded inline can be designated with a #pragma inline statement.

Description:

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

A function called from within a loop is expanded inline.

An example of automatic inline expansion is below. When the following program is compiled after adding the -speed option, f is expanded inline.

C Language Code:

```
extern int *z;
int f  (int pl,int p2)                    /* Expanded function      */
{
    if (p1 > p2)
           return p1;
    else if (p1 < p2)
           return p2;
    else
           return 0;
}
void g (int *x, int *y, int count)
{
    for ( ; count>0; count--, z++, x++, y++)
              *z = f(*x, *y);
}
```

An example of inline expansion by a control statement is shown below. Functions f1 and f2 designated by #pragma inline are expanded inline.

C Language Code:

```
int     v,w,x,y;
#pragma inline(f1,f2)              /* Designation of functions for inline
                                   expansion*/
int     f1(int a, int b)           /* Expanded function              */
{
                return (a+b)/2;
}
int     f2(int c, int d)           /* Expanded function              */
{
                return (c-d)/2;
}
void g ()
{
                int i;
                for(i=0;i<100;i++){
                                if(f1(x,y) == f2(v,w))
                                                sleep();
                }
}
```

Precautions:

1. Designate #pragma inline before the function body definition.
2. Functions designated by #pragma inline also generate external definitions, so when writing inline functions within files where multiple files are included, always designate static in the function declaration.
3. The following functions are not expanded inline:
    - Functions with variable parameters
    - Functions that reference parameter addresses within functions
    - Functions for which the number and format of actual arguments and temporary arguments are not in agreement
    - Functions called by an address
    - Functions called from inline expanded functions
4. With the SH-2 and SH-3, there are cases in which speed is not increased by inline expansion because of cache errors.

5. When this capability is used, there is a tendency for program size to increase because equivalent code is expanded in the function call position. Consideration should be given to finding a balance between execution speed and program size.

2.3.2 Embedded Assembler Inline Expansion Notation Method

There are cases in which one wants to use CPU instructions not supported by the C language, or wants to improve performance by making statements in assembly language rather than in C. In such cases, there is a method of making the statements in assembly language and joining that code with the C program, but the SH series C compiler can incorporate such code into the C source program by using the embedded assembler inline expansion capability.

Code written in assembly language is generally written in the same form as that of C language functions, and when those functions are declared as functions written by the assembler by placing a “#pragma inline_asm” before them, the compiler expands the assembler code in the function call position.

Follow the C compiler creation rules concerning the interface between functions. The C compiler stores parameter values in registers R4 to R7, and generates code assuming that return values are stored in R0.

Description:

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

In the following example, when upper and lower byte switching occurs frequently, it is the key to performance, so a byte swap function is written by the assembler and embedded inline expansion is used.

C Language Code:

```
pragma inline_asm(swap)                 /*Designation of assembler function to be
                                         expanded*/
short swap(short p1)                    /*Write with the assembler the function for
                                         improved performance*/
{
        EXTU.W      R4,R0               ;clear upper word
        SWAP.B      R0,R2               ;swap with R0 lower word
        CMP/GT      R2,R0               ;if (R2 < R0)
        BT          ?0001               ;          then goto ?0001
        NOP                             ;
        MOV         R2,R0               ;return R2
?0001:                                  ;local label
}

void f (short *x, short *y, int i)
{
        for ( ; i > 0; i--, x++, y++)
                *y = swap(*x);          /*Written in the same manner as a C
                                         function call*/
}
```

Assembly Language Expansion Code (Partial):

```
_f:
                MOV.L       R14,@-R15
                MOV         R6,R14
                MOV.L       R13,@-R15
                CMP/PL      R14
                MOV.L       R12,@-R15
                MOV         R5,R13
                MOV         R4,R12
                BT          L218
                MOV.L       L219,R3
                JMP         @R3
```

```
                NOP
L218:
L216:
                MOV.W           @R12,R4
                BRA             L217
                NOP
L219:
                .DATA.L         L215
L217:
                EXTU.W          R4,R0
                SWAP.B          R0,R2
                CMP/GT          R2,R0
                BT              ?0001
                NOP
                MOV             R2,R0
?0001:
                .ALIGN          4
                MOV.W           R0,@R13
                ADD             #-1,R14
                ADD             #2,R12
                ADD             #2,R13
                CMP/PL          R14
                BF              L220
                MOV.L           L221+2,R3
                JMP             @R3
                NOP
L220:
L215:
                MOV.L           @R15+,R12
                MOV.L           @R15+,R13
                RTS
                MOV.L           @R15+,R14
L221:
                .RES.W          1
                .DATA.L         L216
```

Precautions:

1. Designate #pragma inline_asm before the function body definition.
2. Functions designated by #pragma inline_asm also generate external definitions, so when writing inline expansion functions within files where multiple files are included, always designate static in the function declaration.
3. When using labels within assembler notation, always use local labels.
4. When using registers R8 to R15 in an assembler notation function, it is necessary to save and restore those registers at the beginning and end, respectively, of the assembler notation function.
5. Do not write an RTS at the end of an assembler notation function.
6. Compile using the object format designation option -code=asmcode.
7. When this capability has been used, C source level debugging is subject to restrictions.
8. Refer to section 4.1.2, Function Call Interface, for details on performing function calls between C language and assembly language programs.
9. Refer to the user manual for details on combining C programs and assembly programs.

2.4 GBR Base Variable Designation

There are cases when one might wish to increase the execution speed of modules that often access external variables. The GBR base variable designation capability is used in such cases to reference frequently accessed data by the relative addressing mode, using the global base register (GBR). GBR referenced variables are allocated to the $G0, $G1 sections and are referenced by offset from the start address of the $G0 section stored in the GBR. For this reason, the developed code is more compact and faster than code by which an address is loaded to make the reference. This is effective in improving both execution speed and ROM efficiency.

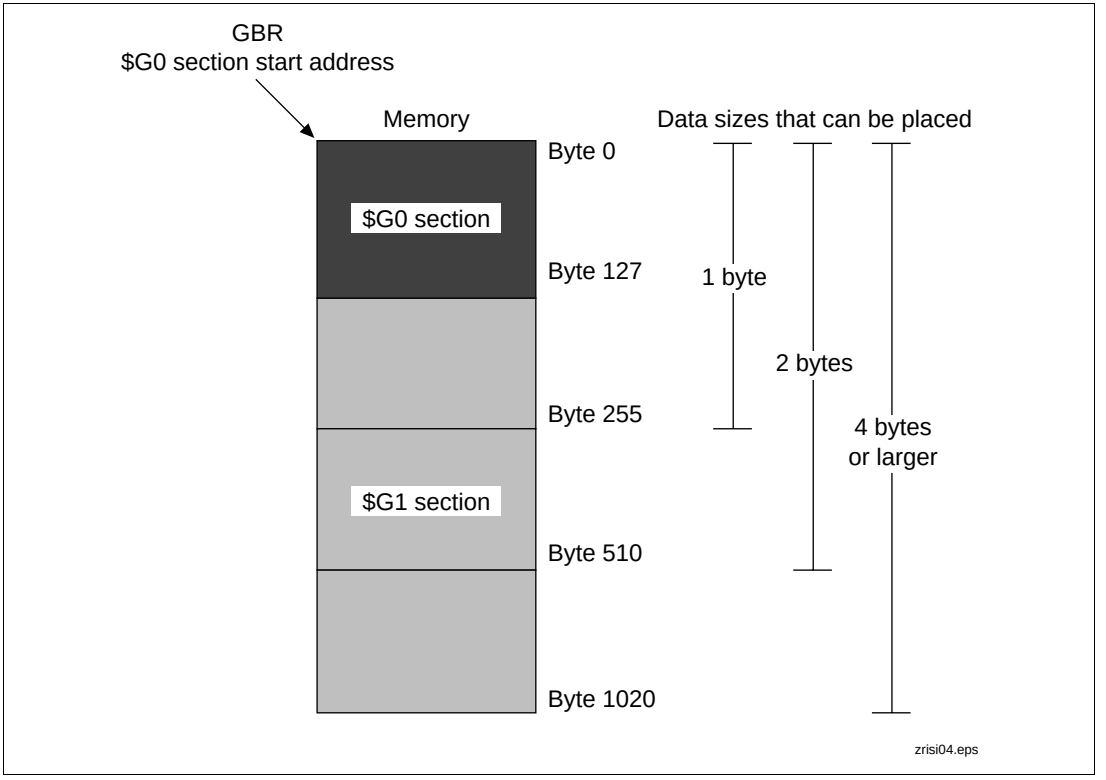


Figure 2.2 GBR Base Variable Referencing

GBR base referencing of external variables is performed by using a preprocessor control statement.

The “#pragma gbr_base” designates that the variable is in an offset of 0 to 127 bytes from the address pointed to by the GBR. Variables designated here are allocated to the “$G0” section.

The “#pragma gbr_base1” designates that the offset for the variable from the address pointed to by the GBR is a maximum of 255 bytes for char format or unsigned char format, a maximum of 510 bytes for short format or unsigned short format, and a maximum of 1020 bytes for int format, unsigned int format, long format, unsigned long format, float format, or double format. Variables designated here are allocated to the “$G1” section.

Description:

```
#pragma gbr_base(<function name>[,<function name>...])
#pragma gbr_base1(<function name>[,<function name>...])
```

C Language Code:

```
#pragma gbr_base(a1,b1,c1)
#pragma gbr_basel(a2,b2,c2)
char   a1,a2;
short  b1,b2;
long   c1,c2;
void   f  ()
{
          a1 = a2;
          b1 = b2;
          c1 = c2;
}
```

Assembly Language Code:

```
_f:
          MOV.B          @(_a2-(STARTOF $G0),GBR),R0
          MOV.B          R0,@(_a1-(STARTOF $G0),GBR)
          MOV.W          @(_b2-(STARTOF $G0),GBR),R0
          MOV.W          R0,@(_b1-(STARTOF $G0),GBR)
          MOV.L          @(_c2-(STARTOF $G0),GBR),R0
          RTS
          MOV.L          R0,@(_c1-(STARTOF $G0),GBR)
```

When using GBR base variables it is necessary to establish beforehand the $G0 section start address in the GBR. An example is given below.

Initializing Program (Assembly Language Section):

```
              :
              :
          .SECTION $G0,DATA,ALIGN=4
              :
              :
__G_BGN:  .DATA.L (STARTOF $G0)        ;Establish the $G0 section start address
              :
              :
          .EXPORT __G_BGN
              :
              :
          .END
```

Initializing Program (C Language Section):

```
#include <machine.h>
extern int *_G_BGN;
void _INITSCT ()                /* Function executed before the main function*/
{
                :
                :
        set_gbr(_G_BGN);        /* Establish the $G0 section start in the GBR
                                   register*/
                :
                :
}
```

Precautions:

1. Establish the $G0 section start address in GBR at the start of program execution.
2. Always place the $G1 section immediately after the $G0 section during linkage, and always create a $G0 section, even when using only #pragma gbr_base1.
3. Operation cannot be guaranteed when the total size exceeds 128 bytes after section $G0 linkage, or when data have offsets within section $G1 greater than those of the individual formats indicated for “#pragma gbr_base1”.
4. Incorrect operation will result if items 2 and 3 above are not fulfilled, so confirm these items with the map list that is output during linkage.
5. As much as possible, allocate frequently accessed data and data on which bit operations are performed to the $G0 section. Accessing data allocated to the $G0 section results in the generation of objects that are more efficient in size and that allow faster execution speed than when data is allocated to the $G1 section.
6. Variables designated by “#pragma gbr_base” or “#pragma gbr_base1” are allocated to the individual sections in the order that they are declared. Keep in mind that the data size will increase if variables of different sizes are alternately declared.

2.5 Register Save/Restore Control

For functions called from functions that perform only function call processing, there are cases in which one might wish to increase execution speed by not performing register saves and restorations. The preprocessor control statements #pragma noregsave, #pragma noregalloc, and #pragma regsave are used in such cases for finer control of register saves/restores.

- #pragma noregsave designates that save/restore of general registers is not performed at function entry and exit.

- #pragma noregalloc designates that save/restore of general registers is not performed at function entry and exit, and that objects are generated without allocating register variable usage registers (R8 to R14) for cases of exceeding the function calls.
- #pragma regsave designates that save/restore of R8 to R14 from among the general registers is performed at function entry and exit.
- Multiple designations of #pragma noregsave and #pragma noregalloc are possible for the same function. When there are multiple designations, all register variable usage registers (R8 to R14) are saved/restored at function entry/exit, and objects are generated without allocating register variable usage registers for cases of exceeding the function calls.

Description:

```
#pragma noregsave(<function name>[,<function name>...])
#pragma noregalloc(<function name>[,<function name>...])
#pragma regsave(<function name>[,<function name>...])
```

Conditions under which register save/restore can be deleted or reduced are indicated below.

Example 1: In a case such as when registers R8 to R14 are used by a function activated at power on, it is not necessary to save/restore the registers, so the object size and execution speed can be improved by designating “#pragma noregsave”.

Example 2: In a case such as when registers R8 to R14 are used in a function through which low power mode ensues without returning to the calling source, it is not necessary to save/restore the registers, so the object size and execution speed can be improved by designating “#pragma noregsave”.

Example 3: When registers R8 to R14 are not allocated in function A, but are allocated in functions B, C, D, and E, objects are generated that the save/restore R8 to R14 at the entry/exit of functions B, C, D, and E. Because R8 to R14 are not used in function A, there is no effect if register saves/restores are not performed by functions called by function A, but since there are cases in which they will be used by functions that have called function A, it is possible to perform the save/restore at the entry/exit of function A, to avoid performing saves/restores in individual functions called from function A.

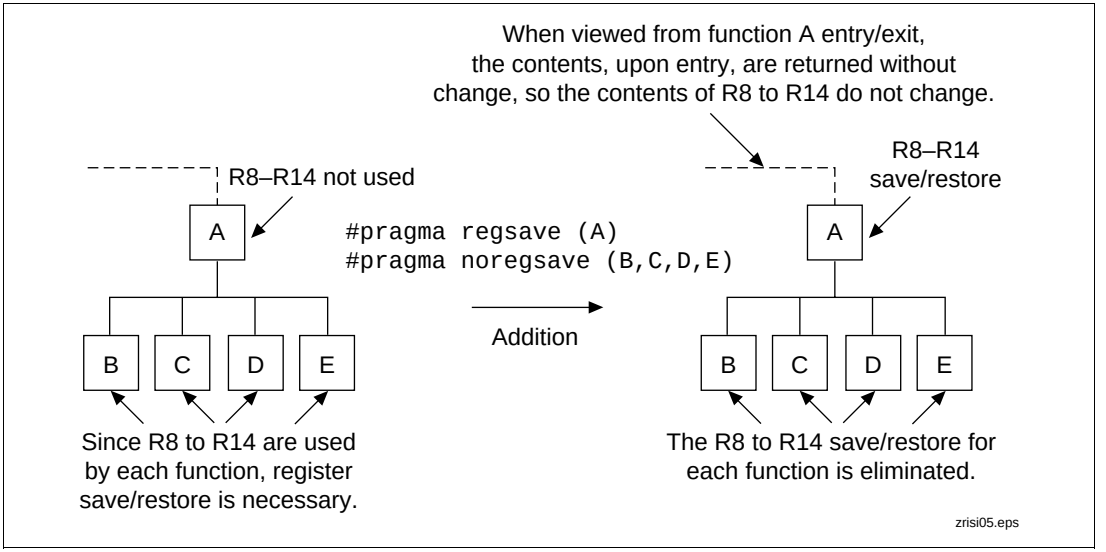


Figure 2.3 Register Save/Restore Control (1)

Example 4: For the same kind of calling relationship as in example 3, when functions C and C1 both use registers R8 to R14, it is necessary to insure that the use of R8 to R14 in function C1 does not cross over into the function C call. In such cases, it is possible to designate function C with a "#pragma noregsave" if a directive is given by designating function C1 with a "#pragma noregalloc" so that R8 to R14 are not allocated to exceed the function call.

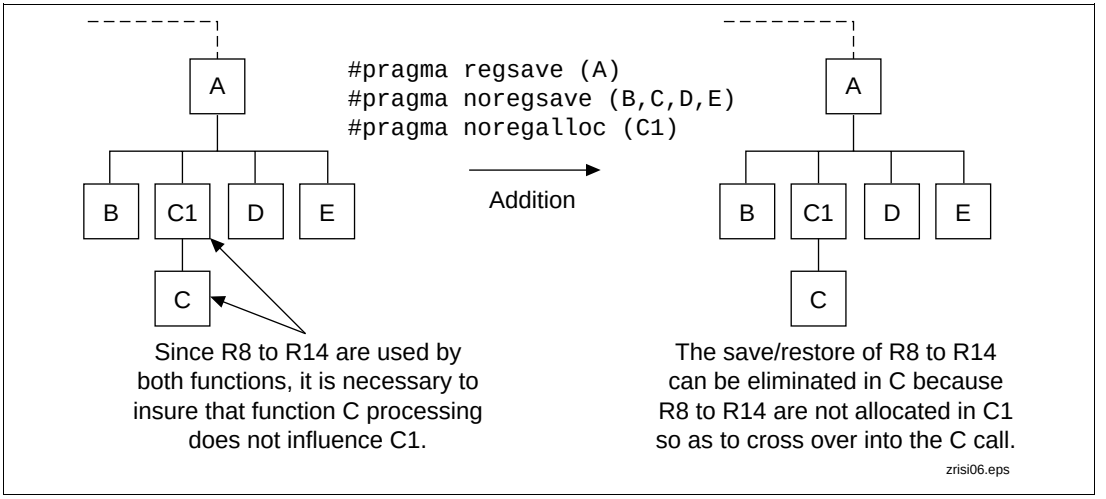


Figure 2.4 Register Save/Restore Control (2)

Example 5: For the same kind of calling relationship as that in example 3, when registers R8 to R14 are also used in function A, it is necessary to insure that use of R8 to R14 in function A does not cross over into the function B, C, D, and E calls. In such cases, the multiple designations of

“#pragma regsave” and “#pragma noregalloc” are used for function A. When the multiple designations of “#pragma regsave” and “#pragma noregalloc” are made, the R8 to R14 saves/restores are performed at function entry/exit, and code is output in which there is no cross over allocation of R8 to R14 in function calls, so it becomes possible to designate functions B, C, D, and E with “#pragma noregsave”.

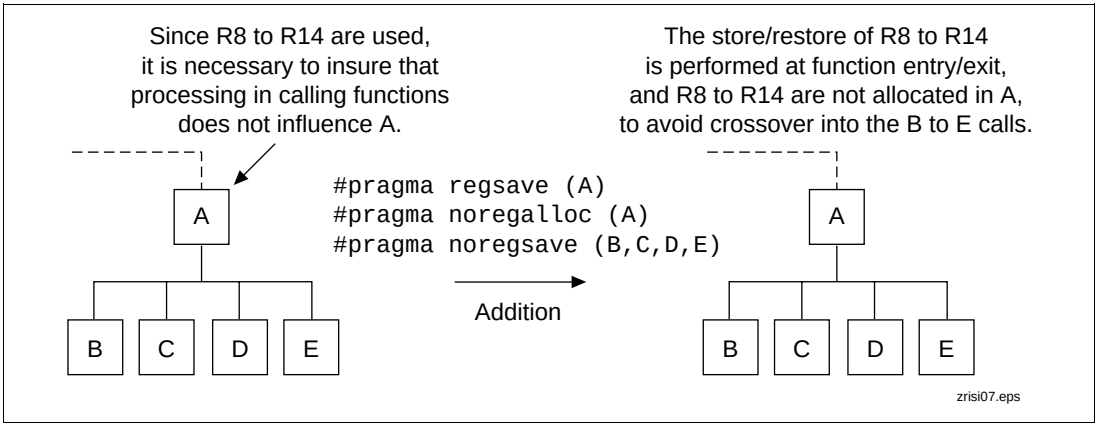


Figure 2.5 Register Save/Restore Control (3)

Precautions: The results of calling functions with a #pragma noregsave designation can not be guaranteed for any cases other than the following:

1. Functions that are not called by other functions, but that are used as first activated functions.
2. Calls from functions with a #pragma regsave designation.
3. Calls from functions with a #pragma regsave designation, by way of functions with #pragma noregalloc designations.

2.6 2-Byte Address Variable Designation

By using a preprocessor control statement, it is possible to indicate to the compiler that externally referenced variables or function addresses are 2-byte.

The compiler regards identifiers declared with “#pragma abs16” as addresses that can be expressed as 2-byte, and always allocates only a 2-byte portion to the storage area allocated for 4-byte addresses. Using this process, it is possible to improve ROM efficiency by decreasing the object size.

This function can be used to great effect if memory placement is arranged during design so that variables and functions referenced by multiple functions are given priority placement in addresses that can be expressed as 2-byte.

Description:

```
#pragma abs16 (<identifier> [,<identifier>...])
```

Identifier: variable name | function name

External access variables and function addresses are established as 2-byte.

C Language Code:

```
#pragma abs16 (x,y,z)
extern int x();
int y;
long z;
f ()
{
        z = x() + y;
}
```

Assembly Language Code:

```
_f:
        STS.L       PR,@-R15
        MOV.W       L212,R3     ;Load the x address
        JSR         @R3
        NOP
        MOV.W       L212+2,R3   ;Load the y address
        MOV.L       @R3,R2
        MOV.W       L212+4,R1   ;Load the z address
        ADD         R2,R0
        LDS.L       @R15+,PR
        RTS
        MOV.L       R0,@R1
L212:
        .DATA.W     _x
        .DATA.W     _y
        .DATA.W     _z
```

Precautions:

1. Set variables and functions designated as 2-byte address in a separate section with the section switching function, and place the section so that the address can be expressed as 2 bytes during

linkage (figure 2.6). An error will occur during linkage if they are not placed in addresses expressed as 2 bytes.

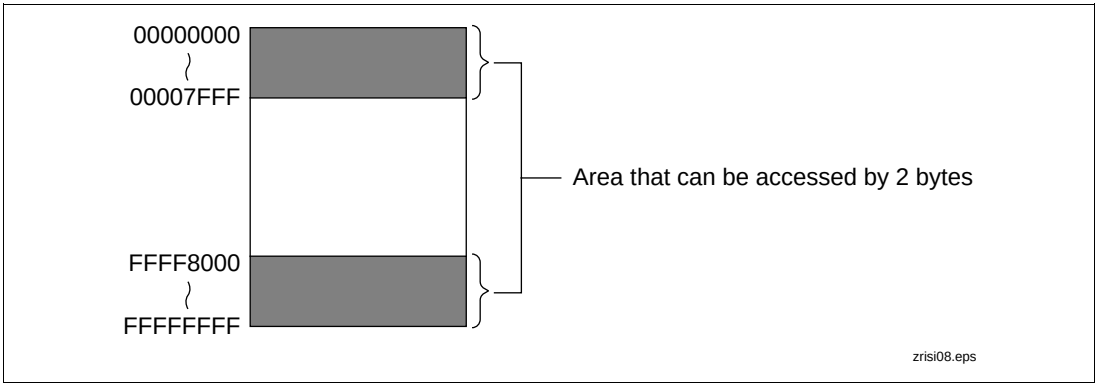


Figure 2.6 Byte Address Variable Designation

2. Function addresses will not be generated as 2-byte if position independent code generation is designated during compilation.

2.7 Section Name Designation

Methods of allocating sections with the same attributes within one system to various addresses (for example, when wishing to allocate certain modules to external RAM and other modules to on-chip RAM), assigning different names to the partitioned sections and designating the addresses at which the various sections are to be placed during linkage are described. The SH-series C compiler provides two different methods of designating section names. In the explanatory example below, modules f, g, h, and data a, b are allocated to f, h, a, and g, b respectively.

With the SH series C compiler, it is possible to designate section names for objects by designating the -section option during compilation. Using this capability, it becomes possible to group both modules and data one wishes to partition into separate files, designate different section names during compilation, and designate the individual start addresses during linkage (figure 2.7).

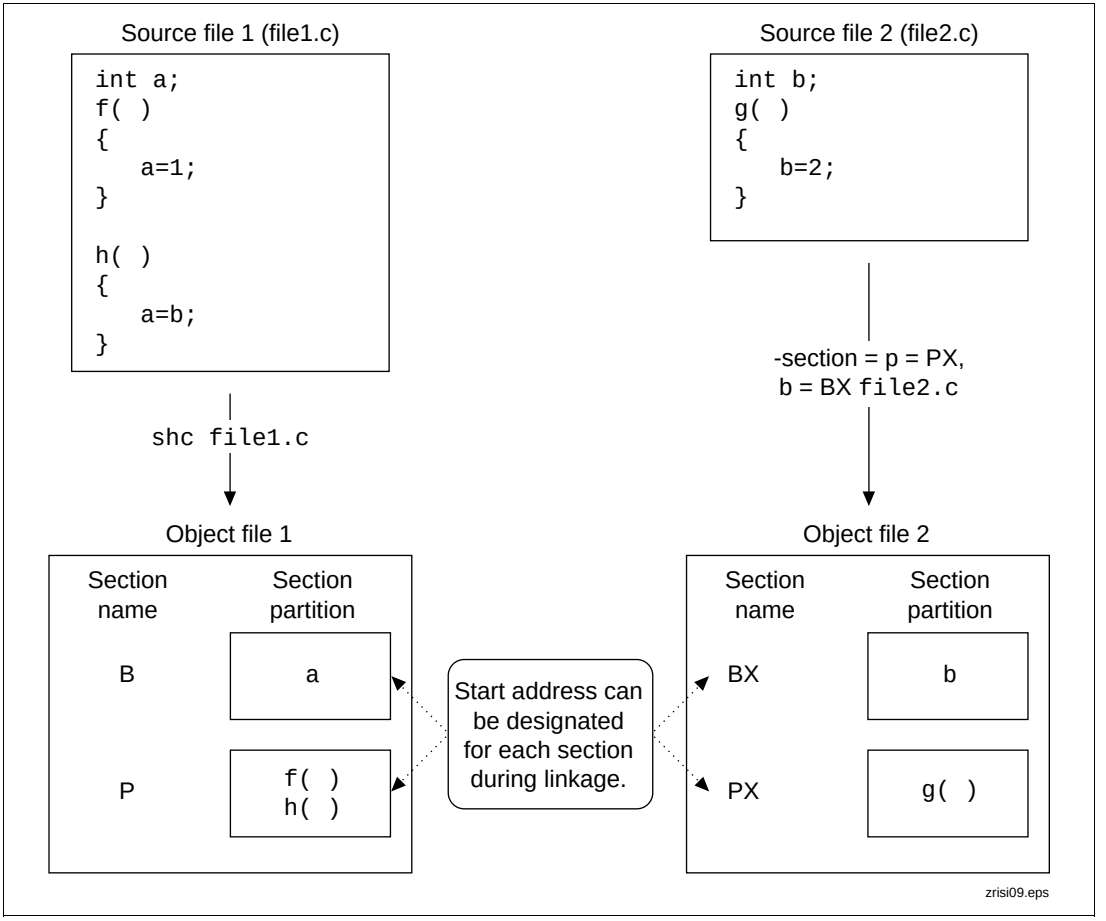


Figure 2.7 Section Name Designation Method

2.8 Section Switching

The -section option only allows designation of section names in file units. However, by using “#pragma section”, it becomes possible to switch section names with the same attributes within a single file, and makes memory allocation more precise. Using this function, it is possible to describe even the section partitions indicated in section 2.7.1 within one file. Figure 2.8 shows an example of this capability.

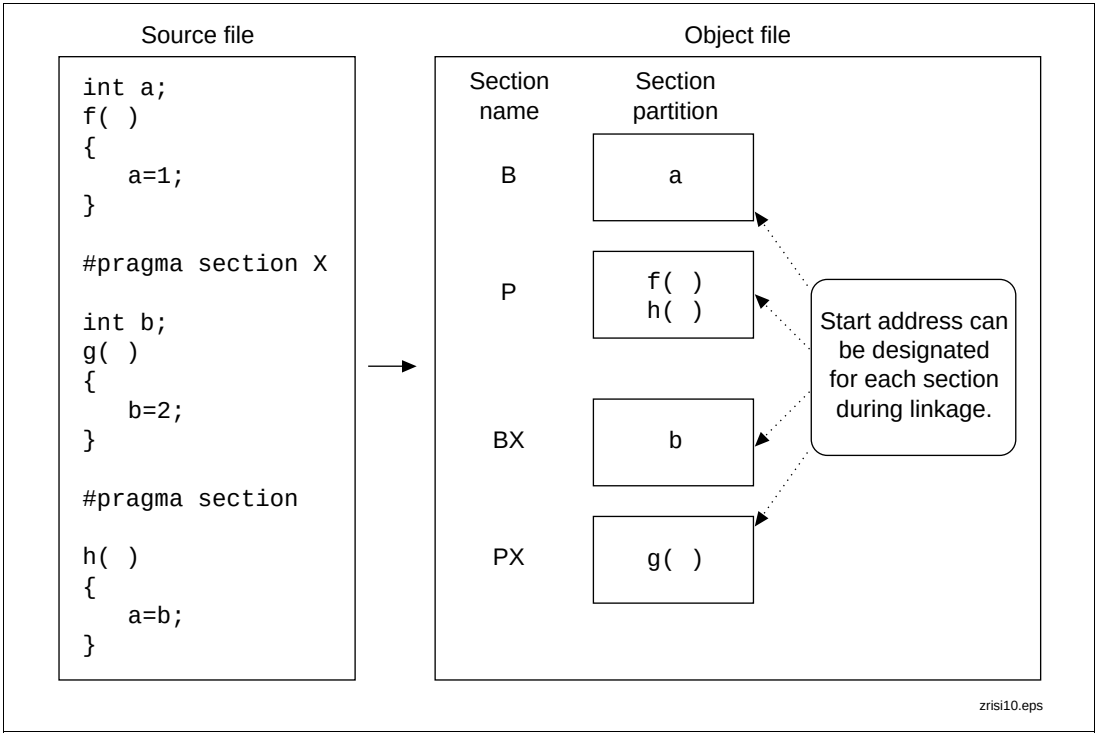


Figure 2.8 Section Switching Method

In this figure, the designation “#pragma section X” causes the program area section name from this line to the line designated by “#pragma section” becomes “PX” and the uninitialized data section name becomes “BX”. A “#pragma section” designation causes a return to the default section name.

2.9 Position Independent Code

There are cases in which code in ROM is transferred to RAM upon startup and operation from RAM is implemented to increase execution speed. To realize this capability, it is necessary that the program allow loading to arbitrary addresses. Coding that allows this is called position independent code (figure 2.9).

The SH series C compiler can generate position independent code if “pic=1” is designated in the command line option during compilation.

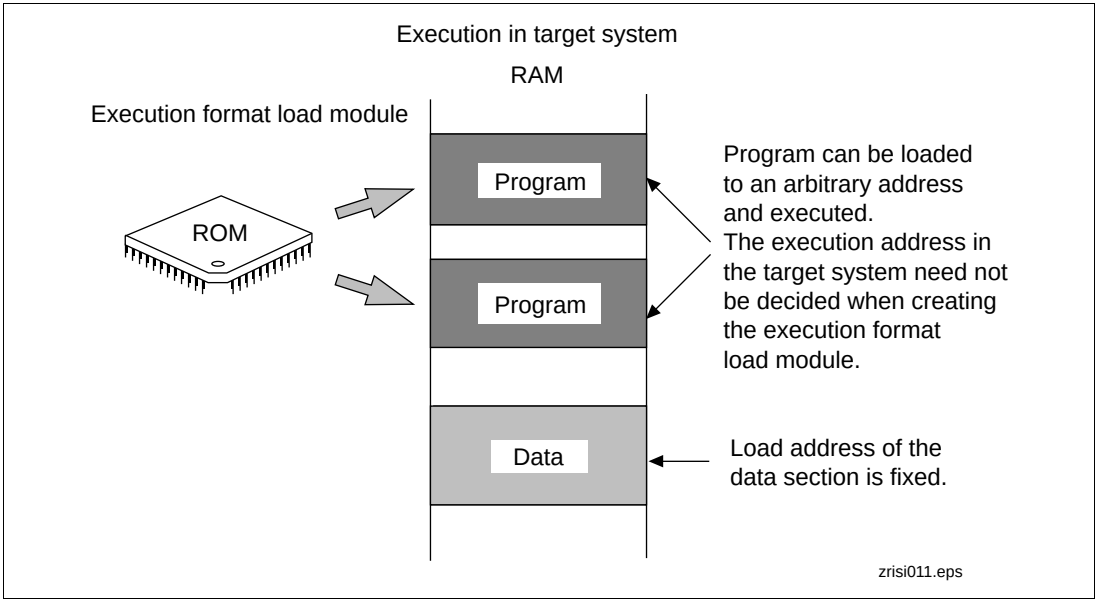


Figure 2.9 Position Independent Code

Precautions:

- Position independent coding can only be used with the SH-2 or SH-3, not with the SH-1.
- Position independent coding cannot be applied to data sections.
- When executing as position independent code, function addresses cannot be designated as initial values. For example:

```
extern int f();
int (*fp)() = f;
```

  Operation cannot be guaranteed in this case because it is not certain that the function f address has been loaded in RAM.
- When using position independent coding, link to the standard libraries “shcpic.lib” for SH-2, and “shc3pb.lib” or “shc3pl.lib” for SH-3. When position independent coding is not used, link to the standard libraries “shcnpic.lib” for SH-2, and “shc3npb.lib” or “shc3npl.lib” for SH-3.

2.10 Options

The options described in table 2.9 are provided with the SH series C compiler so that users can select the policies for code generation.

Table 2.9 Options for Code Generation

Option	Explanation
-speed	Generates optimized for speed code.
-size	Generates code giving priority to size reduction.
-divsion	Selects the method of division. Three methods can be selected, which are, in order of speed, using the CPU division instruction (cpu), using the divider with an interrupt mask (peripheral), and using the divider without an interrupt mask (nomask). However, because the method selected with this option chooses whether to make the CPU or the divider process the division, it is only effective for the SH7604, which includes a divider (even if SH-1 or SH-3 are selected with the -cpu option, code to cause use of the divider will not be executed, though it will be generated).
-macsave	Selects whether to save/restore the contents of the MACL and MACH registers, which store multiplication results, at function entry/exit. The compiler performs MACL, MACH register save/restore as the default, but as long as the MACL, MACH register contents are not referenced extending beyond a jump instruction or not being used in place or general registers it is possible to eliminate unnecessary register saves and restores by not performing MACL, MACH register save/restore. Further, with the code generated by the compiler, multiplication results are immediately stored in general registers and there is no MACL, MACH register referencing which extends into function calls. Consequently, when using only objects output by the compiler it is possible to eliminate the MACL, MACH register save/restore.

Section 3 Effective Programming Techniques

The SH series C compiler performs optimization, but it is possible to improve performance further through skillful programming. This section describes techniques for the user to create more efficient programs. The two standards for program evaluation are that the execution speed be as fast as possible, and that the program size be as small as possible. The SH series C compiler can perform optimization giving priority to execution speed. Designate “speed” in the compiler options to make this so. The basic rules for creating efficient programs are as follows:

- Improve Execution Speed: Because execution speed is determined by statements that are frequently executed and by complex statements, the processing of such should be adequately improved.
- Reduce Size: Similar processes should be standardized and complex functions revised in order to reduce the program size.

As a result of compiler optimization, execution speed will sometimes differ from that found through investigation on the desktop. Use a variety of methods and confirm actual compiler execution in order to obtain better performance. Assembly language development code in this section is obtained by using the following command line:

```
shcΔ<C_language_file>Δ-code=asmcode
```

This section mentions only cases such as assembly language development code differences between the SH-1, SH-2, and SH-3. The assembly language development code may change in the future due to improvements in the compiler.

Table 3.1 is a listing of effective program creation techniques.

Table 3.1 Effective Program Creation Techniques

Item	ROM Efficiency	RAM Efficiency	Execution Speed
Local variables (data size)	O		O
Global variables (sign)	O		O
Multiplication data size	O		O
Data struct conversion	O		O
Data consolidation		O	
Initial values and const format		O	
Local and global variables	O		O
Use of pointer variables	O		O
Constant referencing (1)	O		
Constant referencing (2)	O		
Variables that become fixed values (1)			
Variables that become fixed values (2)			
Module conversion of functions	O		O
Function calls by pointer variable	O		O
Function interface		O	O
Tail recursion	O		O
Movement of constant expressions within loops			O
Loop iteration reduction	X		O
Replacing arithmetic operations with logical operations	O		O
Multiplication/division usage			
Application of formulas			O
Practical use of tables	O		O
Conditional expressions	O		O
Floating point operation speed			
switch statement and if statement	O		O
Inline assembly of functions	X		O
Inline assembly of asm code			O
Practical use of the global base register (GBR)	O		O
Register save/restore control	O		O
2-byte address designation	O		
Prefetch instruction			O

Note: O: improves performance; X: could worsen performance

3.1 Data Designation

Table 3.2 is a listing of items to be given consideration concerning data.

Table 3.2 Cautions on Data Designation

Item	Execution Speed
Data format designators, format modifiers	• There are cases in which program size increases when one tries to reduce the data size. Make format declarations considering the data usage. • Program size can sometimes change depending on the presence/absence of signs, so be careful when making such selections. • For initialized data with unchanging values within the program, the amount of memory used will be reduced if the const operator is attached beforehand.
Data consolidation	Allocate data so that no wasted areas are produced in the data area.
Struct definition/referencing	• Program size can sometimes be reduced by placing frequently accessed/modified data in a struct and using a pointer variable. • Data size can be reduced by using bit fields.
Local variables and global variables	Local variables are more efficient, so always declare any one that can be used as a local variable as such, and not as a global variable.
Use of the pointer format	Check to see whether or not programs using array format can be rewritten using pointer format.
Use of on-chip ROM/RAM	Because accessing on-chip memory is faster than accessing external memory, common variables should be stored in on-chip memory.

3.1.1 Local Variables (Data Size)

Improvements: ROM efficiency and execution speed can sometimes be improved if the local variable size is taken as 4 bytes.

Explanation: Since the Hitachi SuperH RISC engine family general registers are 4-byte, the basis of processing is 4 bytes. Consequently, if there are operations using 1-byte/2-byte local variables, code is added for conversion to 4-byte format. If variables for which 1 byte or 2 bytes are sufficient are also taken as 4-byte, program size is reduced and execution speed can sometimes be improved.

Example: The total sum of the numbers from 1 to 10 is obtained.

Source Code before Improvement:

```
int f( void)
{
    char a = 10;
    int  c = 0;

    for( ; a > 0; a-- )
           c += a;

    return(c);
}
```

Assembled Code before Improvement:

```
_f:
    MOV           #10,R4
    MOV           #0,R5
L211:
    EXTS.B        R4,R3
    ADD           R3,R5
    ADD           #-1,R4
    EXTS.B        R4,R2
    CMP/PL        R2
    BT            L211
    RTS
    MOV           R5,R0
```

Source Code after Improvement:

```
int f( void )
{
    long a = 10;
    int  c = 0;

    for( ; a > 0; a-- )
        c += a;

    return(c);
}
```

Assembled Code after Improvement:

```
_f:
    MOV         #10,R4
    MOV         #0,R5
L211:
    ADD         R4,R5
    ADD         #-1,R4
    CMP/PL      R4
    BT          L211
    RTS
    MOV         R5,R0
```

Item	Before Improvement	After Improvement
Code size	20 bytes	16 bytes
Execution speed	84 cycles	64 cycles

3.1.2 Global Variables (Sign)

Improvements: When global variable format conversions are included within expressions, ROM efficiency and execution speed can be improved if integers are declared as signed when either signed or unsigned is acceptable for the integer format.

Explanation: With the Hitachi SuperH RISC engine family, when 1-byte/2-byte data is transferred from memory with a MOV instruction, the EXTU instruction is added for unsigned data. Consequently, unsigned format integers are less efficient than signed format integers.

Example: The value of variable a is substituted into variable b.

Source Code before Improvement:

```
unsigned short a;
unsigned short b;
int            c;

void f(void)
{
    c = b + a;
}
```

Assembled Code before Improvement:

```
_f:
    MOV.L       L212,R2
    MOV.W       @R2,R3
    MOV.L       L212+4,R0
    EXTU.W      R3,R3
    MOV.W       @R0,R1
    EXTU.W      R1,R1
    ADD         R1,R3
    MOV.L       L212+8,R1
    RTS
    MOV.L       R3,@R1
L212:
    .DATA.L     _b
    .DATA.L     _a
    .DATA.L     _c
```

Source Code after Improvement:

```
short a;
short b;
int    c;

void f(void)
{
    c = b + a;
}
```

Assembled Code after Improvement:

```
_f:
    MOV.L       L212,R2
    MOV.W       @R2,R3
    MOV.L       L212+4,R0
    MOV.W       @R0,R1
    ADD         R1,R3
    MOV.L       L212+8,R1
    RTS
    MOV.L       R3,@R1
L212:
    .DATA.L     _b
    .DATA.L     _a
    .DATA.L     _c
```

Item	Before Improvement	After Improvement
Code size	32 bytes	28 bytes
Execution speed	15 cycles	14 cycles

3.1.3 Data Size (Multiplication)

Improvements: Execution speed can be improved during multiplication if the multiplicand/multiplier are declared as (unsigned)char or (unsigned)short.

Explanation: In SH-2, SH-3 multiplication, the multiplicand/multiplier are implemented with MULS.W/MULU.W instructions when they are 1-byte/2-byte, but with the MUL.L instruction when they are 4-byte.

In SH-1 multiplication, the multiplicand/multiplier are implemented with MULS.W/MULU.W instructions when they are 1-byte/2-byte, but the runtime library is called when they are 4-byte.

Example: The product of variable a and variable b is obtained and returned (SH-1).

Source Code before Improvement:

```
int f( long a, long b )
{
    return( a * b );
}
```

Assembled Code before Improvement:

```
_f:
        STS.L       PR,@-R15
        MOV         R4,R1
        MOV.L       L212,R3
        JSR         @R3
        MOV         R5,R0
        LDS.L       @R15+,PR
        RTS
        NOP
L212:
        .DATA.L     __muli
```

Source Code after Improvement:

```
int f( short a, short b )
{
        return( a * b );
}
```

Assembled Code after Improvement:

```
_f:
        STS.L       MACL,@-R15
        MULS        R5,R4
        STS         MACL,R0
        RTS
        LDS.L       @R15+,MACL
```

Item	Before Improvement	After Improvement
Code size	20 bytes	10 bytes
Execution speed	31 cycles	8 cycles

Note: For a = 1, b = 2.

3.1.4 Data Struct Conversion

Improvements: Execution speed can sometimes be improved if related data are declared with a struct.

Explanation: When references are made many times within the same function, a struct becomes more efficient if the base address is allocated to a register. Efficiency is also improved for passing as arguments. Assembling frequently accessed data at the head of the struct is effective.

Such fine tuning as the modification of data expressions is simplified when data is converted into a struct.

Example: Numerical values are substituted into variables a, b, and c.

Source Code before Improvement:

```
int a, b, c;

void f(void)
{
        a=1;
        b=2;
        c=3;
}
```

Assembled Code before Improvement:

```
_f:
    MOV.L       L212,R2
    MOV         #2,R1
    MOV.L       L212+4,R0
    MOV         #1,R3
    MOV.L       R3,@R2
    MOV         #3,R3
    MOV.L       R1,@R0
    MOV.L       L212+8,R1
    RTS
    MOV.L       R3,@R1
L212:
    .DATA.L     _a
    .DATA.L     _b
    .DATA.L     _c
```

Source Code after Improvement:

```
struct s{
    int a;
    int b;
    int c;
} sl;

void f(void)
{
        register struct s *p=&sl;
        p→a=1;
        p→b=2;
        p→c=3;
}
```

Assembled Code after Improvement:

```
_f:
    MOV.L       L211,R4
    MOV         #1,R3
    MOV.L       R3,@R4
    MOV         #2,R2
    MOV.L       R2,@(4,R4)
    MOV         #3,R3
    RTS
    MOV.L       R3,@(8,R4)
L211:
    .DATA.L     _s1
```

Item	Before Improvement	After Improvement
Code size	32 bytes	20 bytes
Execution speed	12 cycles	10 cycles

3.1.5 Data Consolidation

Improvements: The amount of RAM used can sometimes be reduced by rearranging the order of data declarations.

Explanation: When declaring variables with different size formats, variables with the same size format should be grouped together and declared. Data consolidation in this manner minimizes vacant space in the data area.

Example: A total of 8 bytes of data are placed in memory.

Source Code before Improvement:

```
char   a;
int    b;
short  c;
char   d;
```

Source Code after Improvement:

```
char   a;
char   d;
short  c;
int    b;
```

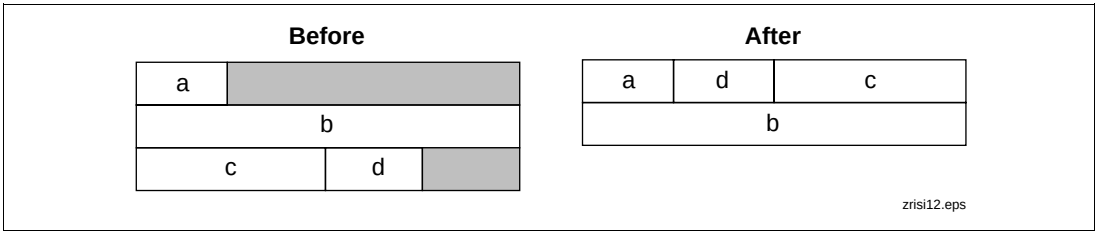


Figure 3.1 Data Placement before and after Improvement

3.1.6 Initial Values and const Format

Improvements: Initial values for which there are no modifications should be declared with the const format.

Explanation: Initialized data is usually transferred from the ROM area to the RAM area during startup, and processing is carried out using the RAM area. For this reason, the secured RAM area is wasted when initialized data with unchanging values exist in the program. If the const operator is added to the initialized data, the transfer to the RAM area during startup is suppressed, resulting in a reduction of the memory used.

Additionally, ROM conversion is simplified if programs are created following the rule that initial values are not modified.

Example: Five initialized data are established.

Source Code before Improvement:

```
char a[] =
    {1, 2, 3, 4, 5};
```

The initial values are transferred from ROM to RAM and processing is performed.

Source Code after Improvement:

```
const char a[] =
    {1, 2, 3, 4, 5};
```

Processing is performed using the initial values in ROM.

3.1.7 Local and Global Variables

Improvements: Execution speed can be improved if locally used variables such as temporary variables, loop counters, etc. are declared as local variables within the functions.

Explanation: For variables that can be used as local variables, always declare them as such, and never as global variables. The values of global variables can end up changing due to such things as function calls or pointer manipulations, so they do not become objects of global optimization.

Use of local variables provides the following advantages:

- The access cost is cheap.
- They can be allocated to registers.
- They become objects of optimization.

Example: A loop of 10 iterations is effected.

Source Code before Improvement:

```
int i;

void f(void)
{
    for( i = 0; i < 10; i++ );
}
```

Assembled Code before Improvement:

```
_f:
        MOV.L           L212+2,R4
        MOV             #0,R3
        MOV             #10,R5
        BRA             L210
        MOV.L           R3,@R4
L211:
        MOV.L           @R4,R1
        ADD             #1,R1
        MOV.L           R1,@R4
L210:
        MOV.L           @R4,R3
        CMP/GE          R5,R3
        BF              L211
        RTS
        NOP
L212:
        .RES.W          1
        .DATA.L         _i
```

Source Code after Improvement:

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

        for( i = 0; i < 10; i++ );
}
```

Assembled Code after Improvement:

```
_f:
        MOV             #10,R5
        MOV             #0,R4
L210:
        ADD             #1,R4
        CMP/GE          R5,R4
        BF              L210
        RTS
        NOP
```

Item	Before Improvement	After Improvement
Code size	32 bytes	15 bytes
Execution speed	125 cycles	54 cycles

3.1.8 Use of Pointer Variables

Improvements: Execution speed can sometimes be improved if programs using array format are rewritten using pointer format.

Explanation: For an array reference a[i], code is generated to add i to the address of a[0]. There are cases where the number of variables and operations can be reduced if a pointer variable is used.

Example: The sum total of 10 (= count) integers is obtained.

Source Code before Improvement:

```
int f( int data[], int count )
{
        int ret = 0, i;

        for(i = 0; i < count; i++)
            ret += data[i];
        return ret;
}
```

Assembled Code before Improvement:

```
_f:
        MOV             #0,R0
        MOV             R0,R7
        MOV             R0,R6
        CMP/GE          R5,R0
        BT              L213
L214:
        ADD             #1,R6
        MOV.L           @R4,R2
        CMP/GE          R5,R6
        ADD             R2,R7
        ADD             #4,R4
        BF              L214
L213:
        RTS
        MOV             R7,R0
```

Source Code after Improvement:

```
int f( int *data, int count )
{
        int ret = 0

        for( ; count > 0; count--)
                ret += *data++;
        return ret;
}
```

Assembled Code after Improvement:

```
_f:
        MOV         #0,R6
        CMP/PL      R5
        BF          L212
L213:
        ADD         #-1,R5
        MOV.L       @R4+,R3
        CMP/PL      R5
        ADD         R3,R6
        BT          L213
L212:
        RTS
        MOV         R6,R0
```

Item	Before Improvement	After Improvement
Code size	26 bytes	20 bytes
Execution speed	87 cycles	75 cycles

3.1.9 Constant Referencing (1)

Improvements: Code size can be reduced if immediate values are expressed beforehand, as much as possible, as 1 byte.

Explanation: When 1-byte immediate values are used, they are embedded in the code. In contrast, when 2-byte or 4-byte immediate values are used, they are generally placed in memory, and an accessing format results.

Example: An immediate value is substituted into variable i.

Source Code (1):

```
void f(void)
{
    i = 0x10000;
}
```

Assembly Development Code (1):

```
_f:
        MOV.L           L210,R3
        MOV.L           L210+4,R2
        RTS
        MOV.L           R3,@R2
L210:
        .DATA.L         H'00010000
        .DATA.L         _i
```

Source Code (2):

```
        void f(void)
        {
                i = 0x01;
        }
```

Assembly Development Code (2):

```
_f:
        MOV.L           L210,R2
        MOV             #1,R3
        RTS
        MOV.L           R3,@R2
L210:
        .DATA.L         _i
```

Item	(1)	(2)
Code size	16 bytes	12 bytes
Execution speed	6 cycles	6 cycles

3.1.10 Constant Referencing (2)

Improvements: The generated code will not be larger if notation of constants in the source code is made easier to read.

Explanation: There is a function that allows the fold-in of constants. Even if constants are expressed in formulas, they will not be reflected in the generated code because they are calculated during compilation.

Example: A constant is substituted into variable a.

Source Code before Improvement:

```
#define MASK1       0x1000
#define MASK2       0x10

int a = 0xffffffff;

void f(void)
{
       int x;

       x = MASK1;
       x |= MASK2;
       a &= x;
}
```

Assembled Code before Improvement:

```
_f:
       MOV.W       L211,R4
       MOV.L       L211+4,R5
       MOV.L       @R5,R3
       AND         R4,R3
       RTS
       MOV.L       R3,@R5
L211:
       .DATA.W     H'1010
       .RES.W      1
       .DATA.L     _a
```

Source Code after Improvement:

```
#define MASK1       0x1000
#define MASK2       0x10

int a = 0xffffffff;

void f(void)
{
        a &= MASK1 | MASK2;
}
```

Assembled Code after Improvement:

```
_f:
        MOV.L       L210+4,R4
        MOV.W       L210,R3
        MOV.L       @R4,R2
        AND         R3,R2
        RTS
        MOV.L       R2,@R4
L210:
        .DATA.W     H'1010
        .RES.W      1
        .DATA.L     _a
```

Item	Before Improvement	After Improvement
Code size	20 bytes	20 bytes
Execution speed	10 cycles	10 cycles

3.1.11 Variables with Fixed Values (1)

Improvements: When variables have fixed values they are handled as constants, so memory efficiency and execution speed will not change even if they are not calculated beforehand.

Explanation: The function that allows the fold-in of constants also operates on variables that become constants; the values of such variables are traced, and constant calculation is performed. Because of this, the generated code will not become larger if notation of the source code is made easier to read.

Example: A return value is changed according to the results of variable rc.

Source Code (1) with Variable Value Calculated Beforehand:

```
#define ERR       -1
#define NORMAL    0

int f(void)
{
       int rc, code;

       rc = 0;
       code = NORMAL;
       return( code );
}
```

Assembly Development Code (1):

```
_f:
       RTS
       MOV          #0,R0
```

Source Code (2) with C Compiler Performing Calculation:

```
#define ERR       -1
#define NORMAL    0

int f(void)
{
       int rc,code;

       rc = 0;
       if( rc )   code = ERR;
       else       code = NORMAL;
       return( code );
}
```

Assembly Development Code (2):

```
_f:
       RTS
       MOV          #0,R0
```

Item	Source Code (1)	Source Code (2)
Code size	4 bytes	4 bytes
Execution speed	4 cycles	4 cycles

3.1.12 Variables with Fixed Values (2)

Improvements: When variables have fixed values they are handled as constants, so memory efficiency and execution speed will not change even if they are not calculated beforehand.

Explanation: The function that allows the fold-in of constants also operates on variables that become constants; the values of such variables are traced, and constant calculation is performed. Because of this, the generated code will not become larger if notation of the source code is made easier to read.

Example: The product of variables a and c is obtained, then substituted into variable b.

Source Code (1) with Variable Value Calculated Beforehand:

```
int f(void)
{
        int a, b;

        a = 3;
        b = 15;
        return b;
}
```

Assembly Development Code (1):

```
_f:
        RTS
        MOV          #15,R0
```

Source Code (2) with C Compiler Performing Calculation:

```
int f(void)
{
        int a, b, c;

        a = 3;
        c = 5;
        b = c * a;
        return b;
}
```

Assembly Development Code (2):

```
_f:
        RTS
        MOV          #15,R0
```

Item	Source Code (1)	Source Code (2)
Code size	4 bytes	4 bytes
Execution speed	4 cycles	4 cycles

3.2 Function Calls

Table 3.3 is a list of cautions concerning function calls.

Table 3.3 Cautions on Function Calls

Item	Cautions
Function placement	Group closely related functions within one file.
Interface	• Strictly choose the number of arguments (up to 4) so that all are allocated to registers. • When there are many arguments, use a struct and pass them with a pointer.
Module partitioning	For very large modules, there are cases in which the various optimizations will not be effectively performed. Use the function called tail recursion to partition into modules with sizes for which optimization can be effectively executed.
Replacement by macros	When there are many function calls, the execution speed can be improved by use of macros. However, the program size increases when macros are used, so select this according to the circumstances.

3.2.1 Module Conversion of Functions

Improvements: Execution speed can be improved by grouping closely related functions within one file.

Explanation: Calling functions in different files is implemented with a JSR instruction, but function calls within the same file are implemented with a BSR instruction if the calling range is close. This allows high speed and compact objects to be generated.

Additionally, modifications during tune-up are simplified by module conversion.

Example: Function g is called from function f.

Source Code before Improvement:

```
extern int g(void);

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

Assembled Code before Improvement:

```
_f:
            MOV.L           L210+2,R3
            JMP             @R3
            NOP
L210:
            .RES.W          1
            .DATA.L         _g
```

Source Code after Improvement:

```
int g(void)
{
}

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

Assembled Code after Improvement:

```
_g:
        RTS
        NOP
_f:
        BRA         _g
        NOP
```

Item	Before Improvement	After Improvement
Code size	12 bytes	4 bytes
Execution speed	5 cycles	3 cycles

Note: The range that can be called with a BSR instruction is ±4096 bytes (±2048 instructions). If the file size becomes too large, use of BSR loses its effectiveness.

3.2.2 Function Calls by Pointer Variable

Improvements: Execution speed can be improved by using a table instead of branching with a switch statement.

Explanation: If the processing for each switch statement case is nearly the same, check to see whether a table can be used.

Example: The function called is changed according to the value of function a.

Source Code before Improvement:

```
void f(int a)
{
        switch(a)
        {
                case0:
                    nop(); break;
                case 1:
                    stop(); break;
                case 2:
                    play(); break;
        }
}
```

Assembled Code before Improvement:

```
_f:
            MOV             R4,R0
            CMP/EQ          #0,R0
            BT              L214
            CMP/EQ          #1,R0
            BT              L215
            CMP/EQ          #2,R0
            BT              L216
            BRA             L217
            NOP
L214:
            MOV.L           L218,R3
            JMP             @R3
            NOP
L215:
            MOV.L           L218+4,R3
            JMP             @R3
            NOP
L216:
            MOV.L           L218+8,R3
            JMP             @R3
            NOP
L217:
            RTS
            NOP
L218:
            .DATA.L         _nop
            .DATA.L         _stop
            .DATA.L         _play
```

Source Code after Improvement:

```
static int (*key[3])()=
          {nop, stop, play};
void f(int a)
{
       (*key[a])();
}
```

Assembled Code after Improvement:

```
_f:
          MOV.L       L215,R0
          ADD         #-4,R15
          MOV.L       R4,@R15
          MOV         R4,R3
          SHLL2       R3
          MOV.L       @(R0,R3),R3
          JMP         R3
          ADD         #4,R15
L215:
          .DATA.L     L210
          .SECTION    D,DATA,ALIGN=4
L210:
          .DATA.L     _nop,_stop,_play
```

Item	Before Improvement	After Improvement
Code size	52 bytes	20 bytes
Execution speed	14 cycles	10 cycles

3.2.3 Function Interface

Improvements: The amount of RAM used can be reduced and execution speed improved through management of function arguments (see section 4.1.2).

Explanation: Strictly limit the number of arguments (up to 4) so that all can be placed in registers. When there are many arguments, use a struct and pass them with a pointer. Calls and function entry/exit processing are simplified when the arguments are in registers. Also, the stack area can be economized. Registers R0 to R3 are work registers, R4 to R7 are for arguments, and R8 to R14 are for local variables.

Example: The function f has five arguments, which is more than the number of registers for argument.

Source Code before Improvement:

```
int f(int, int, int, int, int);

void g(void)
{
    f(1, 2, 3, 4, 5);
}
```

Assembled Code before Improvement:

```
_g:
            STS.L           PR,@-R15
            MOV             #5,R3
            MOV.L           L210+2,R2
            MOV             #4,R7
            MOV.L           R3,@-R15
            MOV             #3,R6
            MOV             #2,R5
            JSR             @R2
            MOV             #1,R4
            ADD             #4,R15
            LDS.L           @R15+,PR
            RTS
            NOP
L210:
            .RES.W          1
            .DATA.L         _f
```

Source Code after Improvement:

```
struct b{
        int a, b, c, d, e;
        } b1={1, 2, 3, 4, 5};

        int f(struct b *p)

        void g(void)
        {
                f(&b1);
        }
```

Assembled Code after Improvement:

```
_g:
            MOV.L       L211,R4
            MOV.L       L211+4,R3
            JMP         @R3
            NOP
L211:
            .DATA.L     _b1
            .DATA.L     _f
```

Item	Before Improvement	After Improvement
Code size	32 bytes	16 bytes
Execution speed	16 cycles	6 cycles

3.2.4 Tail Recursion

Improvements: For large functions, execution speed will not suffer when programs are broken into small modules with function calls one after another at the tail of the large function.

Explanation: When function func3() has been called in function func2() called from function func1(), a transfer to function func3() occurs with a BSR instruction/JSR instruction, and a return to function func2() occurs with an RTS instruction upon completion of function func3() processing, and then a return to function func1() occurs with an RTS instruction upon completion of function func2() processing (figure 3.2, left side).

In this case, when calling function func3() at the tail of function func2(), it is possible to transfer to function func3() with a BSR instruction/JSR instruction and then return directly to function

func1() with an RTS instruction upon completion of function func3() processing (figure 3.2, right side). This capability is called tail recursion.

For very large modules, there are cases in which the various optimizations will not be effectively performed. Performance can be improved by using this capability to partition into modules with sizes for effective optimization.

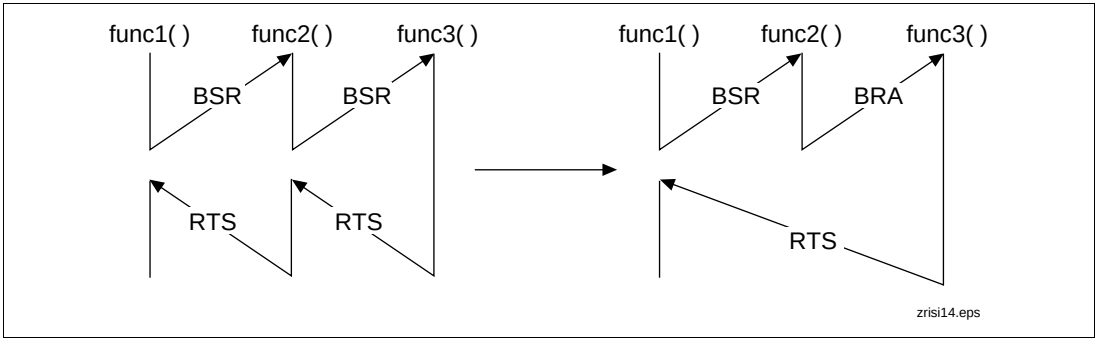


Figure 3.2 Tail Recursion

Example: Functions g and h are called from function f. Returns from g and h are direct returns to the function that called f without returning by way of f.

Source Code before Application (Version 2.0):

```
void f(int x)
{
   if (x==1)
      g();
   else
      h();
}
```

Assembled Code before Application:

```
_f:
        STS.L       PR,@-R15
        MOV         R4,R0
        CMP/EQ      #1,R0
        BF          L207
        BRA         _g
        LDS.L       @R15+,PR
L207:
        BRA         _h
        LDS.L       @R15+,PR
```

Source Code after Application (Version 3.0):

```
void f(int x)
{
    if (x==1)
        g();
    else
        h();
}
```

Assembled Code after Application:

```
_f:
        MOV         R15,R0
        CMP/EQ      #1,R0
        BT          _g
        BRA         _h
        NOP
```

Item	Before Application	After Application
Code size	16 bytes	10 bytes
Execution speed	9 cycles	6 cycles

Note: When x = 2.

3.3 Operation Methods

Table 3.4 is a list of cautions concerning the form of operations.

Table 3.4 Cautions on Operation Methods

Item	Cautions
Constant expressions/unification of common expressions/movement	• Investigate replacing partial expressions used in common within functions with temporary variables. • Place constant expressions used within for statements outside of the for statements.
Loop iteration reduction	• Investigate merging loop statements for which the loop conditions are identical or similar. • Test the loop implementation.
Management of operation methods	• Reduce the number of operations by grouping identical operations. • Investigate whether the same results can be obtained by using logical operators for operations using arithmetic operators.
Application of formulas	Investigate whether the number of operations can be reduced by application of mathematical formulas.
High speed algorithm usage	Investigate such algorithms as quick sorts in arrays that are completed in a shorter calculation time.
Use of tables	• If the processing for each switch statement case is nearly the same, investigate whether a table can be used. • A method exists for improving execution speed by substituting previous operation results into a table and referencing the table values when those operation results become necessary. However, this method increases the amount of ROM used, so select it after balancing the required execution speed with the amount of leeway in ROM capacity.

3.3.1 Movement of Constant Expressions Within Loops

Improvements: Execution speed can be improved if expressions within loops with unchanging values are calculated before the start of the loop.

When expressions with unchanging values in a loop are calculated before the start of the loop, the calculation can be omitted in each iteration and the execution instruction count can be reduced.

Example: Array element b[5] is substituted into array a[].

Source Code before Improvement:

```
void f(void)
{
    int i, j;

    j = 5;
    for ( i=0; i < 100; i++ )
        a[i] = b[j] ;
}
```

Assembled Code before Improvement:

```
_f:
        MOV.L;      L214+4,R5
        MOV         R5,R4
        MOV.W       L214,R6
        ADD         R5,R6
        MOV.L       L214+8,R5
L213:
        MOV.L       @R5,R3
        MOV.L       R3,@R4
        ADD         #4,R4
        CMP/HS      R6,R4
        BF          L213
        RTS
        NOP
L214:
        .DATA.W     H'0190
        .RES.W      1
        .DATA.L     _a
        .DATA.L     H'00000014+_b
```

Source Code after Improvement:

```
void f(void)
{
    int i, j, t;

    j = 5;
    for ( i=0, t=b[j]; i < 100; i++ )
        a[i] = t;
}
```

Assembled Code after Improvement:

```
_f:
        MOV.L       L215+4,R5
        MOV.L       @R5,R5
        MOV.L       L215+8,R7
        MOV         R7,R4
        MOV.W       L215,R6
        ADD         R7,R6
L214:
        MOV.L       R5,@R4
        ADD         #4,R4
        CMP/HS      R6,R4
        BF          L214
        RTS
        NOP
L215:
        .DATA.W     H'0190
        .RES.W      1
        .DATA.L     H'00000014+_b
        .DATA.L     _a
```

Item	Before Improvement	After Improvement
Code size	36 bytes	36 bytes
Execution speed	810 cycles	612 cycles

3.3.2 Loop Iteration Reduction

Improvements: Execution speed can be greatly improved if loops are expanded.

Explanation: Loop expansion is particularly effective for inside loops. The program size increases due to loop expansion, so this should be applied in order to improve execution speed even if it means program size is sacrificed.

Example: Array a[] is initialized.

Source Code before Improvement:

```
void f(void)
{
       int i;
       for ( i = 0; i < 100; i++ )
          a[i] = 0;
}
```

Assembled Code before Improvement:

```
_f:
       MOV.L       L212+2,R7
       MOV         #0,R5
       MOV.W       L212,R6
       MOV         R7,R4
       ADD         R7,R6
L211:
       MOV.L       R5,@R4
       ADD         #4,R4
       CMP/HS      R6,R4
       BF          L211
       RTS
       NOP
L212:
       .DATA.W     H'0190
       .DATA.L     _a
```

Source Code after Improvement:

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

    for ( i = 0; i < 100; i += 2 )
    {
        a[i] = 0;
        a[i+1] = 0;
    }
}
```

Assembled Code after Improvement:

```
_f:
        MOV.L       L213+2,R7
        MOV         #0,R5
        MOV.W       L213,R6
        MOV         R7,R4
        ADD         R7,R6
L212:
        MOV         R4,R7
        MOV.L       R5,@R4
        ADD         #8,R4
        MOV.L       R5,@(4,R7)
        CMP/HS      R6,R4
        BF          L212
        RTS
        NOP
L213:
        .DATA.W     H'190
        .DATA.L     _a
```

Item	Before Improvement	After Improvement
Code size	28 bytes	32 bytes
Execution speed	707 cycles	407 cycles

3.3.3 Replacing Arithmetic Operations with Logical Operations

Improvements: There are cases in which execution speed can be improved if arithmetic operations are replaced by logical operations.

Explanation: Because the execution time is longer for such operations as division, efficiency will be improved when logical operations can be substituted.

Example: A return value is changed according to variable i being odd or even.

Source Code before Improvement:

```
int f(int i)
{
    if (i % 2) code = -1;
    else       code = 0;
    return(code);
}
```

Assembled Code before Improvement:

```
_f:
        STS.L       PR,@-R15
        MOV         R4,R1
        MOV.L       L214+2,R3
        JSR         @R3
        MOV         #2,R0
        TST         R0,R0
        BT          L211
        BRA         L212
        MOV         #-1,R4
L211:
        MOV         #0,R4
L212:
        LDS.L       @R15+,PR
        RTS
        MOV         R4,R0
L214:
        .RES.W      1
        .DATA.L     __modls
```

Source Code after Improvement:

```
int f(int i)
{
        if (i & 1) code = -1;
        else       code = 0;
        return(code);
}
```

Assembly Development Code after Improvement:

```
_f:
        MOV         #1,R3
        TST         R3,R4
        BT          L211
        BRA         L212
        MOV         #-1,R4
L211:
        MOV         #0,R4
L212:
        RTS
        MOV         R4,R0
```

Item	Before Improvement	After Improvement
Code size	32 bytes	16 bytes
Execution speed	112 cycles	10 cycles

Note: When i = 1.

3.3.4 Multiplication/Division Use

Improvements: When uncertain whether to apply multiplication/division or a shift operation, use multiplication/division.

Explanation: Write programs so that they are easy to read. For multiplication/division, when the multiplier/divisor and multiplicand/dividend are unsigned, compiler optimization results in replacement by a combination of shift operations.

Example: Multiplication/division is executed.

Source Code (Multiplication):

```
unsignd int a;

int f(void)
{
        return(a*4);
}
```

Assembled Code for the Above:

```
_f:
        MOV.L       L211,R3
        MOV.L       @R3,R0
        RTS
        SHLL2       R0
L211:
        .DATA.L     _a
```

Source Code (Division):

```
unsignd int b;

int f(void)
{
    return(b/2);
}
```

Assembled Code for the Above:

```
_f:
        MOV.L       L211,R3
        MOV.L       @R3,R0
        RTS
        SHLR        R0
L211:
        .DATA.L     _b
```

3.3.5 Application of Formulas

Improvements: Execution speed can be improved if the number of operations can be reduced through application of mathematical formulas.

Explanation: Analysis is simplified due to mathematical formulas, but be aware that the number of operations sometimes increases when arithmetic is applied.

Example: The sum total from 1 to 100 is obtained (for the SH-2 and SH-3, development is with a MUL.L instruction, so execution speed can be even further improved).

Source Code before Improvement:

```
int f( long n )
{
       int i, s;

       for( s = 0, i = 1;
                   i <= n; i++ )
          s += i;
       return( s );
}
```

Assembled Code before Improvement:

```
_f:
       MOV        #0,R6
       MOV        #1,R5
       CMP/GT     R4,R5
       BT         L212
L213:
       ADD        R5,R6
       ADD        #1,R5
       CMP/GT     R4,R5
       BF         L213
L212:
       RTS
       MOV        R6,R0
```

Source Code after Improvement:

```
int f( long n )
{
        return( n*(n+1)>>1 );
}
```

Assembled Code after Improvement:

```
_f:
        STS.L       PR,@-R15
        MOV         R4,R1
        MOV.L       L211+2,R3
        ADD         #1,R1
        JSR         @R3
        MOV         R4,R0
        LDS.L       @R15+,PR
        RTS
        SHAR        R0
L211:
        .RES.W      1
        .DATA.L     __muli
```

Item	Before Improvement	After Improvement
Code size	20 bytes	24 bytes
Execution speed	606 cycles	32 cycles

3.3.6 Practical Use of Tables

Improvements: Execution speed can be improved by using tables instead of branching with switch statements.

Explanation: If the processing for each switch statement case is nearly the same, check to see whether a table can be used.

Example: The character constant substituted into variable ch is changed according to the value of variable i.

Source Code before Improvement:

```
char f( int i )
{
        char ch;

        switch( i )
        {
                case 0:
                    ch = 'a'; break;
                case 1:
                    ch = 'x'; break;
                case 2:
                    ch = 'b'; break;
        }
        return( ch );
}
```

Assembled Code before Improvement:

```
_f:
        MOV         R4,R0
        CMP/EQ      #0,R0
        BT          L212
        CMP/EQ      #1,R0
        BT          L213
        CMP/EQ      #2,R0
        BT          L214
        BRA         L215
        NOP
L212:
        BRA         L215
        MOV         #97,R4
L213:
        BRA         L215
        MOV         #120,R4
L214:
        MOV         #98,R4
L215:
        RTS
        MOV         R4,R0
```

Source Code after Improvement:

```
char chbuf[] = { 'a', 'x', 'b' };

char f( int i )
{
        return( chbuf[i] );
}
```

Assembled Code after Improvement:

```
_f:
        MOV.L       L212+2,R0
        RTS
        MOV.B       @(R0,R4),R0
L212:
        .RES.W      1
        .DATA.L     _chbuf
```

Item	Before Improvement	After Improvement
Code size	32 bytes	12 bytes
Execution speed	14 cycles	5 cycles

Note: When i = 2.

3.3.7 Conditional Expressions

Improvements: Efficient code is generated when 0 is used in performing comparisons with constants.

Explanation: When comparing with 0, no instruction is generated to load the constant value, so shorter code is generated than when comparing with any value other than 0. Establish such conditional expressions as loops and if statements so that comparisons are made with 0.

Example: A return value is changed according to the argument value being one or greater, or not one or greater.

Source Code before Improvement:

```
int f (int x)
{
        if ( x >= 1 )
                return 1;
        else
                return 0;
}
```

Assembled Code before Improvement:

```
_f:
        MOV         #1,R3
        CMP/GE      R3,R4
        BF          L210
        RTS
        MOV         #1,R0
L210:
        MOV         #0,R0
L211:
        RTS
        NOP
```

Source Code after Improvement:

```
int f (int x)
{
        if ( x > 0)
            return 1;
        else
            return 0;
}
```

Assembly Development Code after Improvement:

```
_f:
        CMP/PL      R4
        BF          L210
        RTS
        MOV         #1,R0
L210:
        MOV         #0,R0
L211:
        RTS
        NOP
```

Item	Before Improvement	After Improvement
Code size	16 bytes	14 bytes
Execution speed	7 cycles	6 cycles

3.3.8 Floating Point Operation Speed

Table 3.5 shows the speeds of the four arithmetical operations. Table 3.6 shows the operation speeds of the elementary functions using standard libraries. In all cases, the clock frequency is 20 MHz, the values represent execution with on-chip ROM/RAM, and the units are μseconds.

Table 3.5 Floating Point Four Arithmetical Operation Speeds

	Double Accuracy Format		Single Accuracy Format	
Operation	Average Value	Worst Value	Average Value	Worst Value
Addition	8.8	16.8	5.3	6.6
Subtraction	10.0	18.1	5.4	7.4
Multiplication (SH-1)	13.6	14.0	6.0	6.0
Multiplication (SH-2)	9.9	10.1	5.3	5.5
Division	41.8	45.2	7.0	7.0

Table 3.6 Floating Point Library Operation Speed Average Values

Function	Double Accuracy Elementary Function Library	Single Accuracy Elementary Function Library
sin	185	95
cos	185	90
tan	260	120
asin	500	200
acos	500	210
atan	325	135
log	315	145
sqrt	97	33
exp	345	165
pow	680	325

3.4 Branching

Keep in mind the following items concerning branching:

- Group identical judgments together.
- When switch statements or else-if statements are long, place cases to be processed quickly and frequently branching cases near the beginning.
- When switch statements or else-if statements are long, execution speed can be improved by dividing into stages and judging.

3.4.1 switch Statement and if Statement

Improvements: For switch statements with up to 5 or 6 cases, execution speed can be improved by using an if statement instead.

Explanation: Replace switch statements having small numbers of cases with if statements. The switch statement has an overhead because it checks the value range of the variable before consulting the case value table. On the other hand, because the if statement makes comparisons over and over again, the efficiency drops if the number of cases increases.

Example: A return value is changed according to the value of variable a.

Source Code before Improvement:

```
int x( int a )
{
    switch( a )
    {
        case 1:
            a = 2; break;
        case 10:
            a = 4; break;
        default:
            a = 0; break;
    }
    return(a);
}
```

Assembled Code before Improvement:

```
_x:
        MOV         R4,R0
        CMP/EQ      #1,R0
        BT          L211
        CMP/EQ      #10,R0
        BT          L212
        BRA         L213
        NOP
L211:
        BRA         L214
        MOV         #2,R4
L212:
        BRA         L214
        MOV         #4,R4
L213:
        MOV         #0,R4
L214:
        RTS
        MOV         R4,R0
```

Source Code after Improvement:

```
int x( int a )
{
        if( a == 1 )
            a = 2;
        else if( a == 10 )
            a = 4;
        else
            a = 0;
        return(a);
}
```

Assembled Code after Improvement:

```
_x:
        MOV         R4,R0
        CMP/EQ      #1,R0
        BF          L210
        BRA         L211
        MOV         #2,R4
L210:
        MOV         R4,R0
        CMP/EQ      #10,R0
        BF          L212
        BRA         L211
        MOV         #4,R4
L212:
        MOV         #0,R4
L211:
        RTS
        MOV         R4,R0
```

Item	Before Improvement	After Improvement
Code size	28 bytes	26 bytes
Execution speed	12 cycles	10 cycles

Note: When a = 1.

3.5 Inline Expansion

Table 3.7 is a listing of items to be given consideration concerning inline expansion.

Table 3.7 Cautions on Inline Development

Item	Cautions
Inline expansion of functions	Try out the inline development of frequently called functions. However, program size increases when functions are expanded, so select for a balance between execution speed and ROM capacity.
Use of inline assembly code	Programs written in assembler code can be called with the same interface as for C language functions.

3.5.1 Inline Expansion of Functions

Improvements: Execution speed can be improved if frequently called functions are expanded inline.

Explanation: Inline expansion of frequently called functions can improve execution speed. Expansion will in some cases allow great effectiveness, particularly for functions called within loops. However, program size tends to increase when inline expansion is used, so it should be applied to improve the execution speed despite the fact program size will be sacrificed.

Example: The elements of array a and array b are exchanged.

Source Code before Improvement:

```
int x[10], y[10];
void g (int *a, int *b, int i)
{
       int temp;

       temp = a[i];
       a[i] = b[i];
       b[i] = temp;
}

void f (void)
{
       int i;
       for ( i = 0; i < 10; i++ )
           g(x, y, i);
}
```

Assembled Code before Improvement:

```
_g:
        ADD         #-4,R15
        MOV         R6,R7
        SHLL2       R7
        MOV.L       R7,@R15
        ADD         R4,R7
        MOV.L       @R7,R6
        MOV.L       @R15,R4
        ADD         R5,R4
        MOV.L       @R4,R3
        MOV.L       R3,@R7
        MOV.L       R6,@R4
        RTS
        ADD         #4,R15
_f:
        MOV.L       R14,@-R15
        MOV.L       R13,@-R15
        MOV         #0,R14
        MOV.L       R12,@-R15
        MOV         #10,R13
        MOV.L       R11,@-R15
        STS.L       PR,@-R15
        MOV.L       L219+2,R11
        MOV.L       L219+6,R12
L218:
        MOV         R14,R6
        MOV         R12,R5
        BSR         _g
        MOV         R11,R4
        ADD         #1,R14
        CMP/GE      R13,R14
        BF          L218
        LDS.L       @R15+,PR
        MOV.L       @R15+,R11
        MOV.L       @R15+,R12
        MOV.L       @R15+,R13
```

```
        RTS
        MOV.L       @R15+,R14
L219:
        .RES.W      1
        .DATA.L     _x
        .DATA.L     _y
```

Source Code after Improvement:

```
#pragma inline (g)
int x[10], y[10];
void g (int *a, int *b, int i)
{
        int temp;

        temp = a[i];
        a[i] = b[i];
        b[i] = temp;
}

void f (void)
{
        int i;
        for ( i = 0; i < 10; i++)
            g(x, y, i);
}
```

Assembled Code after Improvement:

```
_g:
        ADD         #-4,R15
        MOV         R6,R7
        SHLL2       R7
        MOV.L       R7,@R15
        ADD         R4,R7
        MOV.L       @R7,R6
        MOV.L       @R15,R4
        ADD         R5,R4
        MOV.L       @R4,R3
        MOV.L       R3,@R7
        MOV.L       R6,@R4
        RTS
        ADD         #4,R15
_f:
        MOV         #0,R4
        MOV.L       R12,@-R15
        MOV         #10,R12
        MOV.L       R11,@-R15
        MOV.L       R10,@-R15
        MOV.L       L225+2,R10
        MOV.L       L225+6,R11
```

```
L224:
        MOV         R4,R0
        MOV         R11,R1
        MOV         R10,R6
        SHLL2       R0
        MOV         R0,R7
        ADD         R6,R7
        MOV.L       @R7,R6
        MOV         R0,R5
        ADD         R1,R5
        ADD         #1,R4
        MOV.L       @R5,R3
        CMP/GE      R12,R4
        MOV.L       R3,@R7
        MOV.L       R6,@R5
        BF          L224
        MOV.L       @R15+,R10
        MOV.L       @R15+,R11
        RTS
        MOV.L       @R15+,R12
L225:
        .RES.W      1
        .DATA.L     _x
        .DATA.L     _y
```

Item	Before Improvement	After Improvement
Code size	80 bytes	88 bytes
Execution speed	310 cycles	194 cycles

3.5.2 Embedded Inline Assembler Development

Improvements: Execution speed can be improved by writing assembler code into C programs.

Explanation: For the sake of performance, and particularly to improve execution speed, there are times when one might wish to write programs in assembler code. In such cases, it is possible to write only the required section in assembly code, and call that section in the same manner as a C language function is called.

Example: The upper and lower bytes of the elements of array big are swapped, and then stored in array little.

Source Code before Improvement:

```
#define A_MAX 10
typedef unsigned char UChar;
short big[A_MAX],little[A_MAX];
short swap(short p1)
{
        short ret;

        *((UChar *)(&ret)+1) = *((UChar *)(&p1));
        *((UChar *)(&ret)) = *((UChar *)(&p1)+1);
        return ret;
}

void f (void)
{
        int i;
        short *x, *y;

        x = little;
        y = big;
        for(i=0; i<A_MAX; i++, x++, y++){
                *x = swap(*y);
        }
}
```

Assembled Code before Improvement:

```
_swap:
        ADD         #-8,R15
        MOV         R15,R3
        ADD         #6,R3
        MOV         R15,R2
        MOV.W       R4,@R3
        MOV         R15,R0
        ADD         #6,R0
        MOV         R15,R3
        MOV.B       @R0,R0
        MOV.B       R0,@(1,R2)
        MOV         R15,R2
        ADD         #6,R2
        MOV.B       @(1,R2),R0
        MOV.B       R0,@R3
        MOV.W       @R15,R0
        RTS
        ADD         #8,R15
_f:
        MOV.L       R14,@-R15
        MOV.L       R13,@-R15
        MOV         #0,R14
        MOV.L       R12,@-R15
        MOV.L       R11,@-R15
        STS.L       PR,@-R15
        MOV         #10,R11
        MOV.L       L221,R13
        MOV.L       L221+4,R12
L220:
        BSR         _swap
        MOV.W       @R12,R4
        MOV.W       R0,@R13
        ADD         #1,R14
        ADD         #2,R13
        ADD         #2,R12
        CMP/GE      R11,R14
```

```
        BF          L220
        LDS.L       @R15+,PR
        MOV.L       @R15+,R11
        MOV.L       @R15+,R12
        MOV.L       @R15+,R13
        RTS
        MOV.L       @R15+,R14
L221:
        .DATA.L     _little
        .DATA.L     _big
```

Source Code after Improvement:

```
#pragma inline_asm (swap)
#define A_MAX 10
typedef unsigned char UChar;
short big[A_MAX],little[A_MAX];
short swap(short p1)
{
        SWAP.B R4,R0
}

void f (void)
{
        int i;
        short *x, *y;

        x = little;
        y = big;
        for(i=0; i<A_MAX; i++, x++, y++){
                *x = swap(*y);
        }
}
```

Assembled Code after Improvement:

```
_swap:
        SWAP.B      R4,R0
        .ALIGN      4
        RTS
        NOP
_f:
        MOV.L       R14,@-R15
        MOV         #0,R14
        MOV.L       R13,@-R15
        MOV.L       R12,@-R15
        MOV.L       R11,@-R15
        MOV         #10,R11
        MOV.L       L220+2,R13
        MOV.L       L220+6,R12
L218:
        MOV.W       @R12,R4
        BRA         L219
        NOP
L220:
        .RES.W      1
        .DATA.L     _little
        .DATA.L     _big
```

```
L219:
        SWAP.B      R4,R0
        .ALIGN      4
        MOV.W       R0,@R13
        ADD         #1,R14
        ADD         #2,R13
        ADD         #2,R12
        CMP/GE      R11,R14
        BT          L221
        MOV.L       L222,R2
        JMP         @R2
        NOP
L221:
        MOV.L       @R15+,R11
        MOV.L       @R15+,R12
        MOV.L       @R15+,R13
        RTS
        MOV.L       @R15+,R14
L222:
        .DATA.L     L218
```

Item	Before Improvement	After Improvement
Code size	88 bytes	64 bytes
Execution speed	358 cycles	185 cycles

3.6 Practical Use of the Global Base Register (GBR)

Improvements: Performance can be improved by using the GBR to reference external variables with an offset.

Explanation: Compact objects are generated when frequently accessed external variables are referenced by offset from the GBR used as a base register. Additionally, execution speed is sometimes improved due to a related reduction in the number of execution instructions.

Example: The contents of struct y are substituted into struct x.

Source Code before Improvement:

```
struct {
        char   c1;
        char   c2;
        short  s1;
        short  s2;
        long   l1;
        long   l2;
} x, y;
void f (void)
{
        x.c1 = y.c1;
        x.c2 = y.c2;
        x.s1 = y.s1;
        x.s2 = y.s2;
        x.l1 = y.l1;
        x.l2 = y.l2;
}
```

Assembled Code before Improvement:

```
f:
        MOV.L       L211+2,R5
        MOV.L       L211+6,R4
        MOV.B       @R5,R3
        MOV.B       R3,@R4
        MOV.B       @(1,R5),R0
        MOV.B       R0,@(1,R4)
        MOV.W       @(2,R5),R0
        MOV.W       R0,@(2,R4)
        MOV.W       @(4,R5),R0
        MOV.W       R0,@(4,R4)
        MOV.L       @(8,R5),R3
        MOV.L       R3,@(8,R4)
        MOV.L       @(12,R5),R2
        RTS
        MOV.L       R2,@(12,R4)
L211:
        .RES.W      1
        .DATA.L     _y
        .DATA.L     _x
```

Source Code after Improvement:

```
#pragma gbr_base(x,y)
struct {
        char    c1;
        char    c2;
        short   s1;
        short   s2;
        long    l1;
        long    l2;
} x, y;
void f (void)
{
        x.c1 = y.c1;
        x.c2 = y.c2;
        x.s1 = y.s1;
        x.s2 = y.s2;
        x.l1 = y.l1;
        x.l2 = y.l2;
}
```

Assembled Code after Improvement:

```
f:
        MOV.B       @(_y-(STARTOF $G0),GBR),R0
        MOV.B       R0,@(_x-(STARTOF $G0),GBR)
        MOV.B       @(_y-(STARTOF $G0)+1,GBR),R0
        MOV.B       R0,@(_x-(STARTOF $G0)+1,GBR)
        MOV.W       @(_y-(STARTOF $G0)+2,GBR),R0
        MOV.W       R0,@(_x-(STARTOF $G0)+2,GBR)
        MOV.W       @(_y-(STARTOF $G0)+4,GBR,R0
        MOV.W       R0,@(_x-(STARTOF $G0)+4,GBR)
        MOV.L       @(_y-(STARTOF $G0)+8,GBR),R0
        MOV.L       R0,@(_x-(STARTOF $G0)+8,GBR)
        MOV.L       @(_y-(STARTOF $G0)+12,GBR),R0
        RTS
        MOV.L       R0,@(_x-(STARTOF $G0)+12,GBR)
```

Item	Before Improvement	After Improvement
Code size	40 bytes	26 bytes
Execution speed	28 cycles	25 cycles

3.7 Register Save/Restore Control

Improvements: Execution speed can be improved by managing the method of saving and restoring registers.

Explanation: Execution speed and ROM efficiency can be improved by eliminating both the save and restore of the register variable usage register performed at the entry and exit of a final function. However, the opposite effect could possibly result because of the need to either save and restore the register variable usage register in the function that called the function in which the save/restore was eliminated, or to bypass the function call and use an object in which a register variable usage register is not allocated. Select this only after carefully investigating the location where it is to be applied.

Example: The save and restore of the stack are performed together by the function table.

Source Code before Improvement:

```
typedef int
        ARRAY[LISTMAX][LISTMAX][LISTMAX];
ARRAY ary1, ary2, ary3;

void table (void)
{
        init(74755, ary1);
        copy(ary1, ary2);
        sum(ary1, ary2, ary3);
}
void init (int seed, ARRAY p)
}
        int i, j, k;
        for (i=0; i<LISTMAX; i++)
            for (j=0; j<LISTMAX; j++)
                for (k=0; k<LISTMAX; k++){
                    seed = (seed*1309)&16383;
                    p[i][j][k] = seed;
                }
}
void copy (ARRAY p, ARRAY q)
{
        int i, j, k;
        for (i=0; i<LISTMAX; i++)
            for (j=0; j<LISTMAX; j++)
                for (k=0; k<LISTMAX; k++)
                    q[k][i][j] = p[i][j][k];
}
void sum (ARRAY p, ARRAY q, ARRAY r)
{
        int i, j, k;
        for (i=0; i<LISTMAX; i++)
            for (j=0; j<LISTMAX; j++)
                for (k=0; k<LISTMAX; k++)
                    r[i][j][k]=p[i][j][k] + q[i][j][k];
}
```

Assembled Code before Improvement (Partial):

```
_table:
        MOV.L       R14,@-R15
        STS.L       PR,@-R15
        MOV.L       L244+6,R14
        MOV.L       L244+10,R4
        BSR         _init
        MOV         R14,R5
        MOV.L       L244+14,R5
        BSR         _copy
        MOV         R14,R4
        MOV         R14,R4
        LDS.L       @R15+,PR
        MOV.L       L244+18,R6
        MOV.L       L244+14,R5
        BRA         _sum
        MOV.L       @R15+,R14
_init:
        MOV         #2,R6
        MOV.L       R14,@-R15
        MOV.L       R13,@-R15
        MOV.L       R12,@-R15
        MOV         #0,R13
        MOV.L       R11,@-R15
        MOV         R5,R12
        MOV.L       R10,@-R15
        MOV.L       R9,@-R15
        MOV.L       R8,@-R15
            :
            :
        LDS.L       @R15+,PR
        MOV.L       @R15+,R8
        MOV.L       @R15+,R9
        MOV.L       @R15+,R10
        MOV.L       @R15+,R11
        MOV.L       @R15+,R12
        MOV.L       @R15+,R13
        RTS
```

```
        MOV.L       @R15+,R14
_copy:
        MOV.L       R14,@-R15
        MOV.L       R13,@-R15
        MOV.L       R12,@-R15
        MOV.L       R9,@-R15
        MOV         #2,R12
        MOV.L       R8,@-R15
            :
            :
        MOV.L       @R15+,R8
        MOV.L       @R15+,R9
        MOV.L       @R15+,R12
        MOV.L       @R15+,R13
        RTS
        MOV.L       @R15+,R14
            :
            :
_sum:
        MOV.L       R14,@-R15
        MOV         #0,R7
        MOV.L       R13,@-R15
        MOV.L       R12,@-R15
        MOV.L       R11,@-R15
        MOV         #2,R12
        MOV.L       R10,@-R15
        MOV.L       R9,@-R15
        MOV.L       R8,@-R15
            :
        MOV.L       @R15+,R8
        MOV.L       @R15+,R9
        MOV.L       @R15+,R10
        MOV.L       @R15+,R11
        MOV.L       @R15+,R12
        MOV.L       @R15+,R13
        RTS
        MOV.L       @R15+,R14
```

Source Code after Improvement:

```
#pragma regsave (table)
#pragma noregalloc (table)
#pragma noregsave (init, copy, sum)
typedef int
       ARRAY[LISTMAX][LISTMAX][LISTMAX];
ARRAY ary1, ary2, ary3;
void table (void)
{
       init(74755, ary1);
       copy(ary1, ary2);
       sum(ary1, ary2, ary3);
}
void init (int seed, ARRAY p)
{
    int i, j, k;
    for (i=0; i<LISTMAX; i++)
        for (j=0; j<LISTMAX; j++)
               for (k=0; k<LISTMAX; k++){
                   seed = (seed*1309)&16383;
                   p[i][j][k] = seed;
               }
}
void copy (ARRAY p, ARRAY q)
{
       int i, j, k;
for (i=0; i<LISTMAX; i++)
    for (j=0; j<LISTMAX; j++)
        for (k=0; k<LISTMAX; k++)
           q[k][i][j] = p[i][j][k];
```

```
}
void sum (ARRAY p, ARRAY q, ARRAY r)
{
    int i, j, k;
        for (i=0; i<LISTMAX; i++)
            for (j=0; j<LISTMAX; j++)
                for (k=0; k<LISTMAX; k++)
                    r[i][j][k]=p[i][j][k] + q[i][j][k];
}
```

Assembled Code after Improvement (Partial):

```
_table:
        MOV.L       R14,@-R15
        MOV.L       R13,@-R15
        MOV.L       R12,@-R15
        MOV.L       R11,@-R15
        MOV.L       R10,@-R15
        MOV.L       R9,@-R15
        MOV.L       R8,@-R15
        STS.L       PR,@-R15
        STS.L       MACH,@-R15
        STS.L       MACL,@-R15
        MOV.L       L244+4,R5
        MOV.L       L244+8,R4
        BSR         _init
        NOP
        MOV.L       L244+12,R5
        MOV.L       L244+4,R4
        BSR         _copy
        NOP
        MOV.L       L244+16,R6
        MOV.L       L244+12,R5
        MOV.L       L244+4,R4
        BSR         _sum
        NOP
        LDS.L       @R15+,MACL
        LDS.L       @R15+,MACH
        LDS.L       @R15+,PR
        MOV.L       @R15+,R8
        MOV.L       @R15+,R9
        MOV.L       @R15+,R10
        MOV.L       @R15+,R11
        MOV.L       @R15+,R12
        MOV.L       @R15+,R13
        RTS
        MOV.L       @R15+,R14
_init:
```

```
        STS.L       PR,@-R15
        MOV         R5,R12
        MOV.W       L244,R9
        MOV         #0,R13
        MOV.W       L244+2,R10
        MOV         #2,R6
        MOV         R5,R8
        ADD         #32,R8
            :
            :
        CMP/GE      R6,R11
        BF          L236
        ADD         #16,R12
        CMP/HS      R8,R12
        BF          L235
        LDS.L       @R15+,PR
        RTS
        NOP
_copy:
        ADD         #-4,R15
        MOV         #0,R9
        MOV         R9,R13
        MOV         #2,R12
        MOV         R5,R8
        ADD         #32,R8
L238:
        MOV         R9,R14
            :
            :
        ADD         #1,R14
        CMP/GE      R12,R14
        BF          L239
        ADD         #1,R13
        CMP/GE      R12,R13
        BF          L238
        RTS
        ADD         #4,R15
_sum:
```

```
        ADD         #-8,R15
        MOV         #2,R12
        MOV         #0,R7
        MOV.L       R7,@R15
L241:
        MOV         R7,R10
        MOV.L       @R15,R13
        SHLL2       R13
            :
            :
        BF          L242
        MOV.L       @R15,R2
        ADD         #1,R2
        CMP/GE      R12,R2
        MOV.L       R2,@R15
        BF          L241
        RTS
        ADD         #8,R15
L244:
        .DATA.W     H'3FFF
        .DATA.W     H'051D
        .DATA.L     _ary1
        .DATA.L     H'00012403
        .DATA.L     _ary2
        .DATA.L     _ary3
        .DATA.L     __muli
```

Item	Before Improvement	After Improvement
Code size	370 bytes	332 bytes
Execution speed	819 cycles	795 cycles

3.8 2-Byte Address Designation

Improvements: ROM efficiency can be improved by expressing variable and function addresses as 2 bytes.

Explanation: When variables or functions are placed in addresses that can be expressed in 2 bytes, code size can be reduced by making the code on the referencing side 2-byte.

Example: The external function g is called when the value of function x is 1.

Source Code before Improvement:

```
extern int x;
extern void g(void);

void f (void)
{
    if (x == 1)
        g();
}
```

Assembled Code before Improvement:

```
_f:
        MOV.L       L212+2,R3
        MOV.L       @R3,R0
        CMP/EQ      #1,R0
        BF          L213
        MOV.L       L212+6,R2
        JMP         @R2
        NOP
L213:
        RTS
        NOP
L212:
        .RES.W      1
        .DATA.L     _x
        .DATA.L     _g
```

Source Code after Improvement:

```
#pragma abs16(x,g)
extern int x;
extern void g(void);

void f (void)
{
       if (x == 1)
           g();
}
```

Assembled Code after Improvement:

```
_f:
       MOV.W       L212,R3
       MOV.L       @R3,R0
       CMP/EQ      #1,R0
       BF          L213
       MOV.W       L212+2,R2
       JMP         @R2
       NOP
L213:
       RTS
       NOP
L212:
       .DATA.W     _x
       .DATA.W     _g
```

Item	Before Improvement	After Improvement
Code size	28 bytes	22 bytes
Execution speed	16 cycles	16 cycles

Note: When x = 1 and function g is void g(){}.

3.9 Prefetch Instruction

Improvements: When accessing array variables, an improvement in execution speed can be expected if a prefetch instruction is executed before the access (valid only for SH-3).

Explanation: When performing sequential accesses of an array with a loop, execution speed is improved by performing a prefetch before referencing the array members. Additionally, prefetches can be performed more effectively through loop expansion. However, no increase in speed can be expected when prefetch instructions are executed consecutively, so be certain to execute with an amount of separation sufficient to allow completion of a previous prefetch instruction.

Example: The product of each element of array a and array b are stored in array c.

Source Code before Improvement:

```
int a[1200], b[1200], c[1200];
void f (void)
{
       int i;
       int *pa, *pb, *pc;

       for (pa=a, pb=b, pc=c,
              i=0; i<1200; i+=4){
           *pc++ = *pa++ * *pb++;
           *pc++ = *pa++ * *pb++;
           *pc++ = *pa++ * *pb++;
           *pc++ = *pa++ * *pb++;
       }
}
```

Assembled Code before Improvement:

```
_f:
        STS.L       MACL,@-R15
        MOV         #0,R7
        MOV.W       L218,R0
        MOV.L       L218+2,R6
        MOV.L       L218+6,R4
        MOV.L       L218+10,R5
L217:
        MOV.L       @R4+,R2
        ADD         #4,R7
        MOV.L       @R6+,R1
        CMP/GE      R0,R7
        MUL.L       R2,R1
        STS         MACL,R2
        MOV.L       R2,@R5
        ADD         #4,R5
        MOV.L       @R4+,R2
        MOV.L       @R6+,R1
        MUL.L       R2,R1
        STS         MACL,R2
        MOV.L       R2,@R5
        ADD         #4,R5
        MOV.L       @R4+,R2
        MOV.L       @R6+,R1
        MUL.L       R2,R1
        STS         MACL,R2
        MOV.L       R2,@R5
        ADD         #4,R5
        MOV.L       @R4+,R2
        MOV.L       @R6+,R1
        MUL.L       R2,R1
        STS         MACL,R2
        MOV.L       R2,@R5
        BF/S        L217
        ADD         #4,R5
        RTS
```

```
        LDS.L       @R15+,MACL
L218:
        .DATA.W     H'04B0
        .DATA.L     _a
        .DATA.L     _b
        .DATA.L     _c
```

Source Code after Improvement:

```
#include <umachine.h>

int a[1200], b[1200], c[1200];

void f (void)
{
        int i;
        int *pa, *pb, *pc;

        for (pa=a, pb=b, pc=c, i=0, i<1200; i+=4){
#ifdef PREF1
            prefetch(pa+8);
#endif
            *pc++ = *pa++ * *pb++;
            *pc++ = *pa++ * *pb++;
#ifdef PREF2
            prefetch(pb+8);
#endif
            *pc++ = *pa++ * *pb++;
            *pc++ = *pa++ * *pb++;
    }
}
```

Assembly Development Code after Improvement (When PREF1, 2 Are Valid):

```
_f:
        STS.L       MACL,@-R15
        MOV         #0,R7
        MOV.W       L218,R0
        MOV.L       L218+2,R5
        MOV.L       L218+6,R4
        MOV.L       L218+10,R6
L217:
        MOV         R5,R3
        ADD         #32,R3
        PREF        @R3
        MOV.L       @R4+,R3
        MOV.L       @R5+,R1
        MUL.L       R3,R1
        STS         MACL,R3
        MOV.L       R3,@R6
        MOV.L       @R4+,R3
        ADD         #4,R6
        MOV.L       @R5+,R1
        MOV         R4,R2
        MUL.L       R3,R1
        ADD         #32,R2
        STS         MACL,R3
        MOV.L       R3,@R6
        ADD         #4,R6
        PREF        @R2
```

```
        MOV.L       @R4+,R2
        ADD         #4,R7
        MOV.L       @R5+,R1
        CMP/GE      R0,R7
        MUL.L       R2,R1
        STS         MACL,R2
        MOV.L       R2,@R6
        ADD         #4,R6
        MOV.L       @R4+,R2
        MOV.L       @R5+,R1
        MUL.L       R2,R1
        STS         MACL,R2
        MOV.L       R2,@R6
        BF/S        L217
        ADD         #4,R6
        RTS
        LDS.L       @R15+,MACL
L218:
        .DATA.W     H'04B0
        .DATA.L     _a
        .DATA.L     _b
        .DATA.L     _c
```

Item	Before Improvement	After Improvement 1 (PREF1 only)	After Improvement 2 (PREF1, 2)
Code size	84 bytes	92 bytes	96 bytes
Execution speed	61650 cycles	60300 cycles	57930 cycles

Section 4 Relation to Assembly Language Programs and Cross Software

4.1 Relation to Assembly Language Programs

Because the SH series C compiler also supports Hitachi SuperH RISC engine family dedicated special instructions, it allows most programs to be written in the C language. However, for better performance, some sections require assembly language. Then those sections can be joined with the C language programs.

This section outlines the following points one should be careful of when joining C language and assembly language programs.

- External name reciprocal referencing methods
- Function call interface

Refer to the *SH Series C Compiler User Manual* for details.

4.1.1 External Name Reciprocal Referencing Methods

Referencing Assembly From C Language Programs: Referencing assembly language program external definition names from C language programs is as follows:

- Make an external definition declaration using the ".EXPORT" or ".GLOBAL" assembler control instructions of the symbol names (32 characters or fewer) for assembly language programs with "_" added to the beginning of their names.
- In the C program, make an external reference declaration of the symbol names without "_" added at the beginning, using an "extern" memory class designator.

Assembly Language Program (Defining Side):

```
        .EXPORT     _a, _b
        .SECTION    D,DATA,ALIGN=4
_a:     .DATA.L     1
_b:     .DATA.L     1
        .END
```

C Language Program (Referencing Side):

```
extern int a, b;

f()
{
        a += b;
}
```

Referencing C Programs from Assembly: Referencing C language program external definition names from assembly language programs is as follows:

- Make an external definition of the symbol names (31 characters or fewer) in the C language program.
- In the assembly language program, make an external reference declaration of the symbol names with an underscore (_) added to the beginning, using an ".IMPORT" or ".GLOBAL" assembler control instruction.

External definition names for C language programs are as follows:

- Those that are global variables, and further, are not static memory class
- Function names declared as extern memory class
- Function names for which static memory class is not designated

C Language Program (Defining Side):

```
int a;
```

Assembly Language Program (Referencing Side):

```
                .IMPORT     _a
                .SECTION    P, CODE, ALIGN=2
                MOV.L       A_a, R1
                MOV.L       @R1, R0
                ADD         #1, R0
                RTS
                MOV.L       R0, @R1
A_a:            .DATA.L     _a
                .END
```

4.1.2 Function Call Interface

When performing reciprocal function calls between C language programs and assembly language programs, the following four rules should be observed on the assembly language program side:

1. Rule concerning the stack pointer
2. Rule concerning stack frame allocation/release
3. Rule concerning the registers
4. Rule concerning the setting/referencing of arguments and return values

Items 1 through 3 are explained below. Refer to section 4.1.3, Argument and Return Value Setting/Referencing, for item 4.

Stack Pointer: Do not store valid data in the stack area lower (in the direction of the 0 address) than the address pointed to by the stack pointer. Data stored in addresses lower than that of the stack pointer could possibly be destroyed by interrupt processing.

Stack Frame Allocation/Release: At the point of a function call being performed (immediately after execution of a JSR or BSR instruction), the stack pointer indicates the final address of the stack used on the calling function side. Data allocation/setting for addresses higher than this area (in the direction of address H'FFFFFFFF) is the duty of the calling side function.

At the time of the function return, the area established by the called function is released, then returned to the calling function, usually through use of an RTS instruction. The area with higher addresses than this (return value address and argument area) is released by the calling side function (figure 4.1).

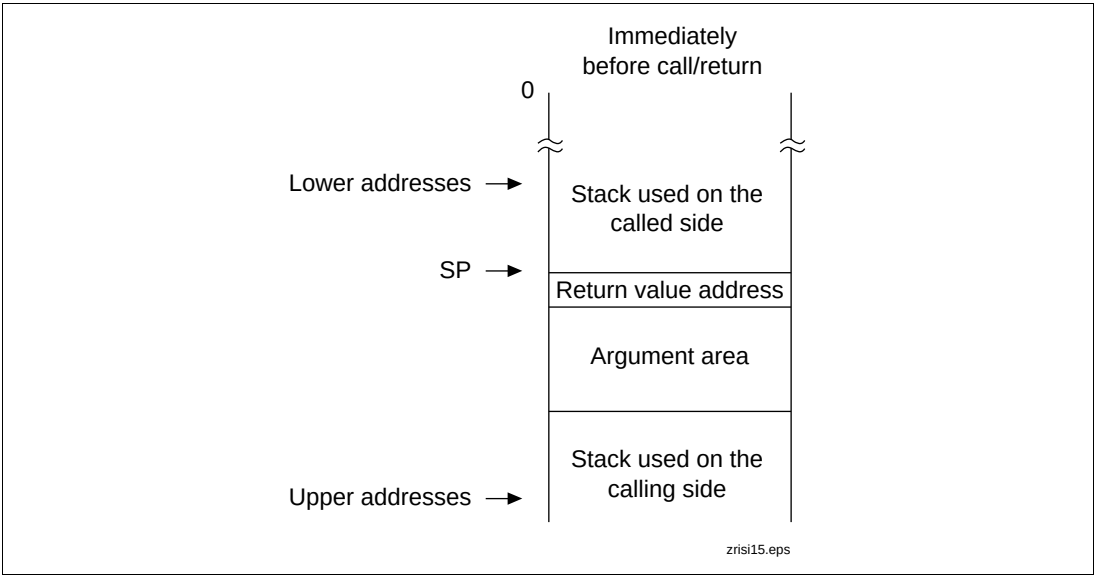


Figure 4.1 Stack Frame Allocation/Release

Registers: Immediately after a function call, there are registers for which the C compiler preserves the values, and others for which it does not. The rule for register preservation is listed in table 4.1.

Table 4.1 Rule for Register Preservation Immediately after Function Calls in C Programs

Item	Object Registers	Cautions on Assembly Language Programming
Registers not preserved	R0 to R7	If there are valid values in the object registers at the time of a function call, those values are saved on the calling side. They can be used by functions on the called side without saving.
Registers preserved	R8 to R15, MACH, MACL*, PR	Among the object registers, the values of those used within functions are saved, then restored upon the return.

Note: MACH, MACL are not preserved when -macsave = 0.

Establish the connection between C language program and assembly language program functions as follows:

- When calling assembly language functions from C language programs:

  When the object assembly language function calls a different module, save the PR register value to the stack at the assembly language function entrance, and restore it from the stack at the exit.

  When the R8 to R15, MACH, or MACL registers are used within the assembly language function, save the register values to the stack before using, and restore them from the stack after using.

  Refer to section 4.1.3, Argument and Return Value Setting/Referencing, for details on how arguments are passed to assembly language functions.

- When calling C language functions from assembly language programs:

  If there are valid values in the R0 to R7 registers, save the values to empty registers or to the stack before the C language function call.

  Refer to section 4.1.3, Argument and Return Value Setting/Referencing, for details on how return values are passed to assembly language functions.

  The following is an example of how the assembly language function g is called by the C language function f, and how the C language function h is called by the assembly language function g.

C Language Function f:

```
extern void g();

f()
{
        g();
{
```

Assembly Language Function g:

```
        .EXPORT     _g              Function g external definition
                                    declaration
        .IMPORT     _h              Function h external reference
                                    declaration
        .SECTION    P, CODE, ALIGN=2
_g:     STS.L       PR,@-R15        PR register value preservation
        MOV.L       R14,@-R15       Preservation of registers used by
                                    function g
        MOV.L       R13,@-R15
            :
        MOV.L       R2,@-R15        Preservation of registers used by
                                    function h
        MOV.L       R1,@-R15
        MOV.L       L_h,R0          Call function h
        JSR         @R0
        NOP
            :
        MOV.L       @R15+,R13       Restoration of registers used by
                                    function g
        MOV.L       @R15+,R14
        RTS
        LDS.L       @R15+,PR        PR register value restoration

L_h:    .DATA.L     _h
        .END
```

C Language Function h:

```
        h()
        {
                :
                :
        {
```

4.1.3 Argument and Return Value Setting/Referencing

The C compiler rules concerning argument and return value setting/referencing differ depending on whether or not the individual argument and return value formats are clearly declared in the

function declaration. When the argument and return value formats are clearly declared in the C language program the basic format declaration is used for the function.

In the following explanation, general rules concerning argument and return values in C programs are described first, followed by an explanation of the allocation area for arguments, the method of allocating arguments, and the location of return value establishment.

General Rules Concerning Arguments and Return Values in C Programs: Method of passing arguments: Always call functions only after copying argument values to registers or to the argument allocation area in the stack. The argument allocation area is not referenced by the calling side function after a return, so even if the argument values are changed by functions on the called side, the calling side processing is not directly effected.

Rules for format conversion: When arguments are passed or return values returned, there are cases in which the formats are automatically converted. Table 4.2 indicates the rules concerning format conversion.

Table 4.2 Rules for Format Conversion

Format Conversion	Conversion Method
Format conversion of arguments with declared formats	Arguments with formats declared with a basic format declaration are converted to the declared format.
Format conversion of arguments without declared formats	Arguments without formats declared with a basic format declaration are converted according to the following rules: • char format, unsigned char format, short format, and unsigned short format arguments are converted to int format. • float format arguments are converted to double format. • Formats other than the above are not converted.
Format conversion of return values	Return values are converted to the format returned by the function.

Example 1: When the format is declared with the basic format declaration:

```
long f();
long f()
{
        float x;
            :
            :
        return x;
}
```

The return value x is converted to long format in accordance with the basic format declaration.

Example 2: When the format is not declared with the basic format declaration:

```
void p(int,...);

long f()
{
    char c;
        :
    p(1.0, c);
        :
}
```

The first argument is converted to int format since the format of the corresponding argument is int format. The second argument is converted to int format since the corresponding argument has no format.

Example 3: When the format is not declared with the basic format declaration:

When the argument format is not declared with the basic format declaration, the same format should be designated on the called side and calling side so that the argument will be correctly passed. Operation is not guaranteed if the formats do not match.

```
void f(x)
float x;
{
       :
       :
}

void main()
{
       float x:
       f(x);
}
```

In this example, there is no basic format declaration for the function f argument, so argument x is converted to double format when it is called on the function main side. However, the argument has been declared as float format on the function f side. Consequently, the argument cannot be correctly handed over. The argument format must either be declared with a basic format declaration, or the argument declaration must be double format on the function f side.

Proper declaration of the argument format with a basic format declaration is shown below.

```
void f(float x)
{
        :
        :
}

void main()
{
        float x:
        f(x);
}
```

Argument Allocation Method in C Programs: Arguments are sometimes allocated to registers and sometimes to an argument area in the stack. Figure 4.2 shows the argument allocation area, and table 4.3 lists the general rules for argument allocation.

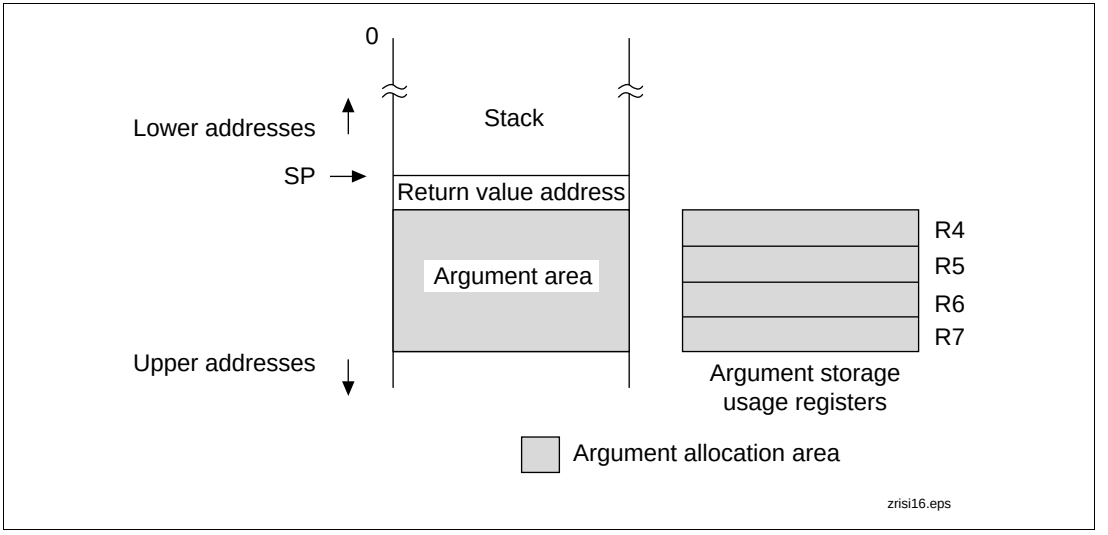


Figure 4.2 Argument Allocation Area for C Language Programs

Table 4.3 General Rules for Argument Allocation in C Programs

Arguments		Allocation Rules
Arguments passed using registers	Argument storage usage registers	R4 to R7
Arguments passed using registers	Object formats	char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, pointer
Arguments passed using the stack		• Arguments with formats other than those that are objects of register passing • Functions declared as having variable arguments with the basic format declaration* • When other arguments have already been allocated to registers R4 to R7

Note: When functions have been declared as having variable arguments with the basic format declaration, arguments without corresponding formats within the declaration and the arguments immediately preceding them are allocated to the stack. For example:

```
int f2(int, int, int, int,...);
f2(a, b, c, x, y, z)
{
        :
}
```

Arguments up until the fourth argument are allocated to registers as usual, but x here is also allocated to the stack.

Allocation to argument storage usage registers: Allocation to the argument storage usage registers occurs from the lowest numbered register in the order of the source program declarations. Figure 4.3 shows an example of argument storage usage register allocation.

Allocation to the argument area in the stack: Allocation to the argument area in the stack occurs from the lower addresses in the order declared in the source program. Figures 4.4 through 4.8 show examples of argument storage usage stack allocation.

Note: When structs and common element format arguments are established, they are allocated to 4-byte boundaries regardless of the boundary adjustments of their formats, and additionally, byte areas that are multiples of 4 are used for their areas. This is because the SH stack pointer changes in 4-byte units.

Example 1: All arguments are register passing format → Allocated to R4 to R7 in order of declaration (figure 4.3)

```
int f(char, short, int, float);
       :
   f(1, 2, 3, 4.0);
       :
```

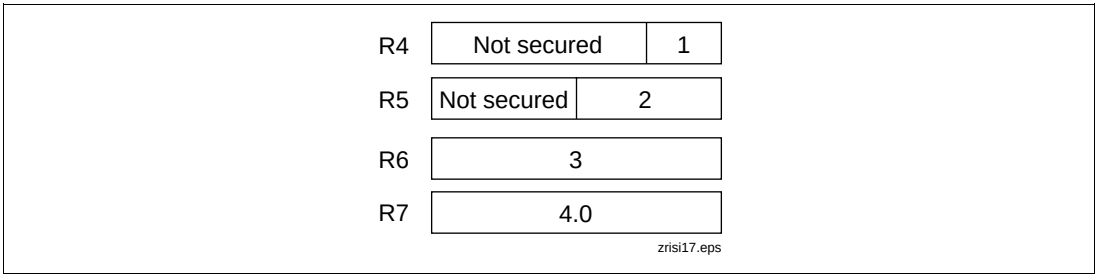


Figure 4.3 C Language Program Argument Allocation (Example 1)

Example 2: There are many arguments and not all are allocated to registers → Arguments not allocated to registers are allocated to the stack (figure 4.4)

```
int f(int, short, long, float,char);
       :
   f(1, 2, 3, 4.0, 5);
       :
```

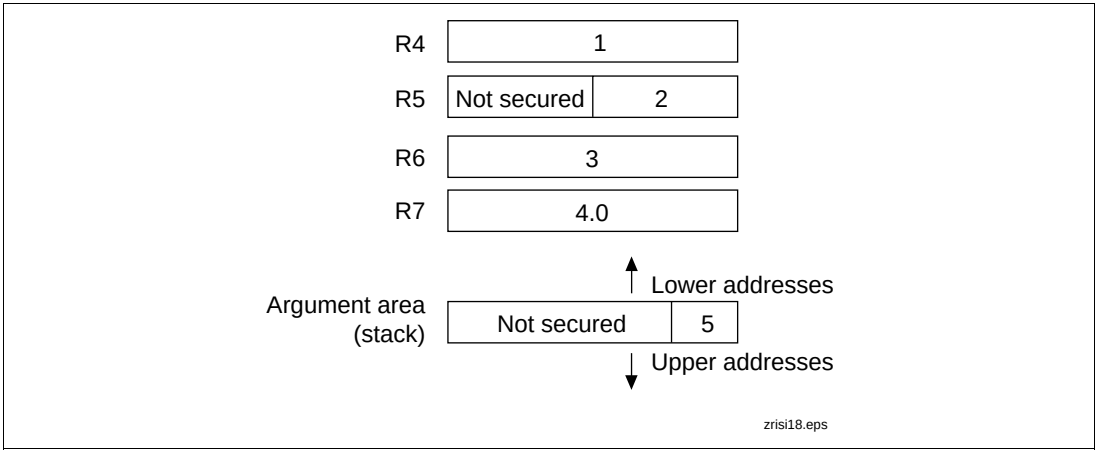


Figure 4.4 C Language Program Argument Allocation (Example 2)

Example 3: There are arguments with formats not allocated to registers → Arguments not allocated to registers are allocated to the stack (figure 4.5)

```
struct s { int x, y;} a;
int f(int, struct s, int);
       :
f(1, a, 3);
       :
```

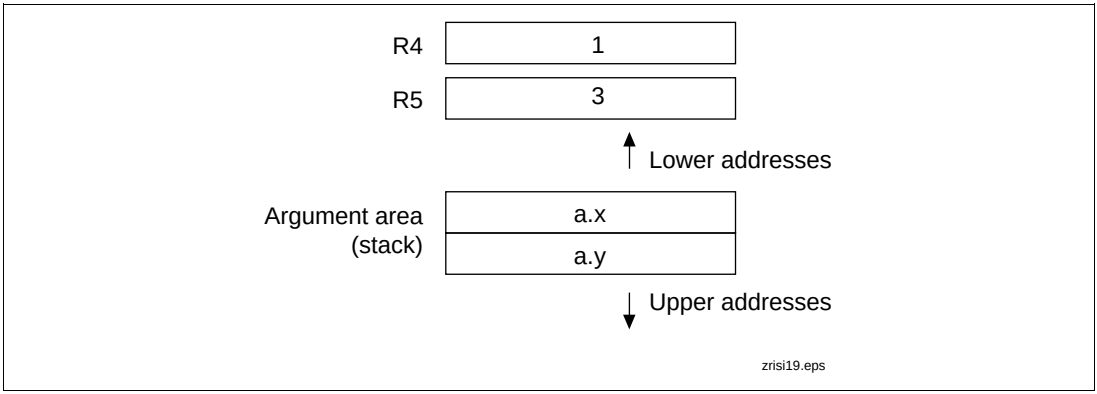


Figure 4.5 C Language Program Argument Allocation (Example 3)

Example 4: Functions have been declared as having variable arguments with the basic format declaration → Arguments without corresponding formats and the arguments immediately preceding them are allocated to the stack in the order of declaration (figure 4.6)

```
int f(double, int, int,  . . .);

    :
 f(1.0, 2, 3, 4);
    :
```

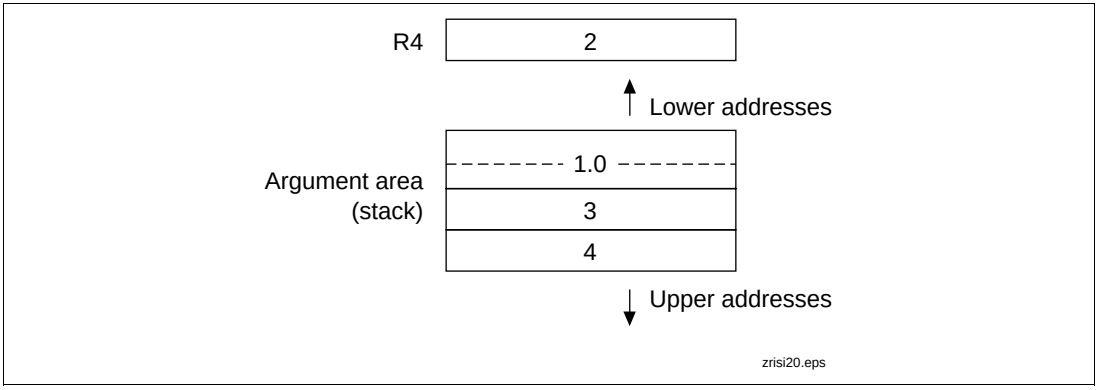


Figure 4.6 C Language Program Argument Allocation (Example 4)

Example 5: There are no basic format declarations → Char format expanded to int format, float format expanded to double format before allocation (figure 4.7)

```
int f();
char a;
float b;
       :
       f(a,b);
       :
```

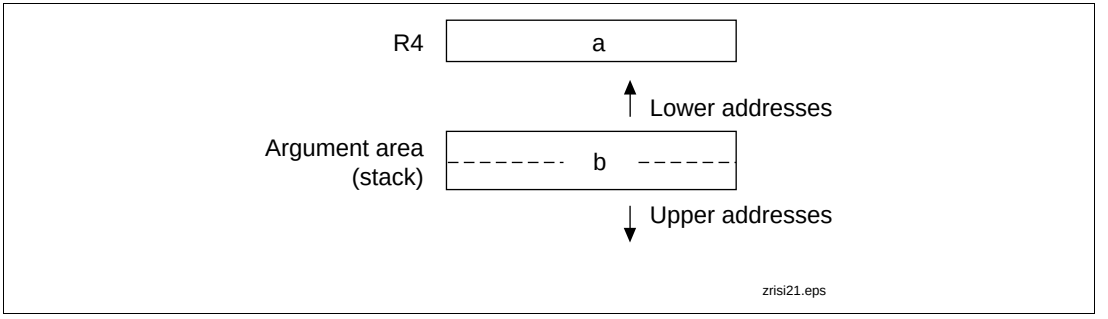


Figure 4.7 C Language Program Argument Allocation (Example 5)

Example 6: The function's return format exceeds 4 bytes, or it is a struct→ Return value address established immediately before the argument area (figure 4.8)

```
struct s{char x, y, z}a, b;
double f(struct s);
       :
       f(a);
       :
```

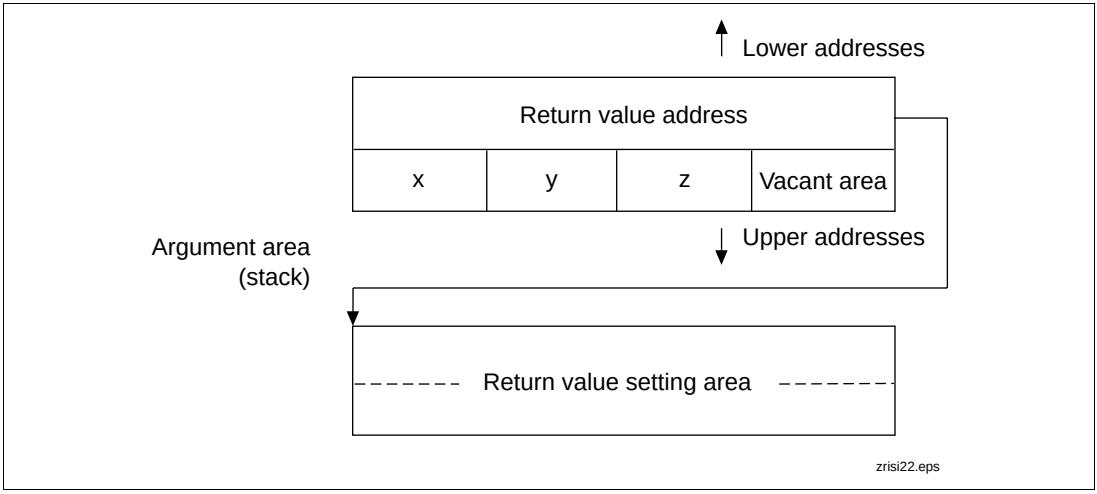


Figure 4.8 C Language Program Argument Allocation (Example 6)

Return Value Setting Location in C Programs: Return values are sometimes set in registers and sometimes in the stack, depending on the function's return value format. Refer to table 4.4 for the relationship between return value format and setting location.

When the function return value is set in the stack, that return value is established in the area pointed to by the return value address. The calling side secures not only the argument area, but the return value setting area as well and sets that address in the return value address before making the function call (see figure 4.9). The return value is not established if the function return value is void format.

Table 4.4 Return Value Formats and Setting Locations in C Programs

Return Value Format	Return Value Setting Location
char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, pointer	R0: 32 bit (contents of the char, unsigned char upper 3 bytes, and short, unsigned short upper 2 bytes are not secured)
double, long double, structs, common elements	Return value setting area (stack)

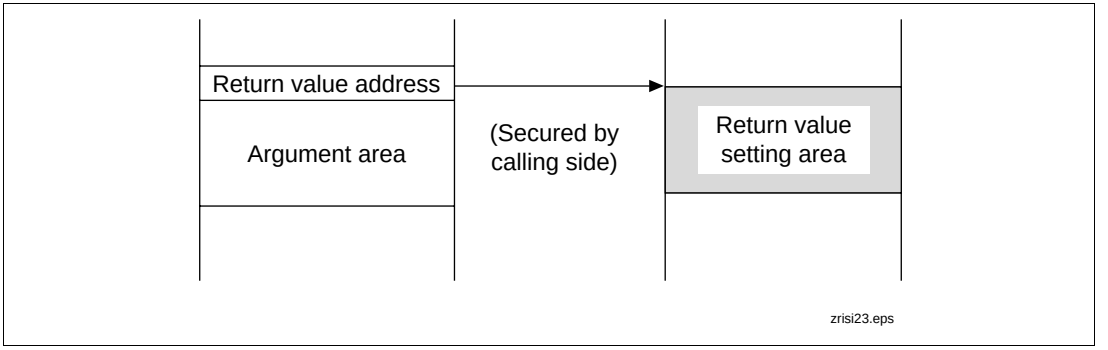


Figure 4.9 C Language Program Return Value Setting Area when Return Values Are Set in the Stack

4.2 Relation to the Linkage Editor

4.2.1 ROM Conversion Support Function

When writing load modules into ROM, the initialized data area is also written in. However, because data manipulation must be performed in RAM, the initialized data area must be copied from ROM to RAM during startup. This processing can be easily executed by using the linkage editor's ROM conversion support function.

To use the ROM conversion support function, designate the option "ROM (D,R)" (D: initialized data area section name in ROM, R: initialized data area section name in RAM) during linkage.

The following processes are carried out by the ROM conversion support function:

1. An area in RAM with the same size as the initialized data area in ROM is secured. Figure 4.10 shows the method of dual allocation to memory.

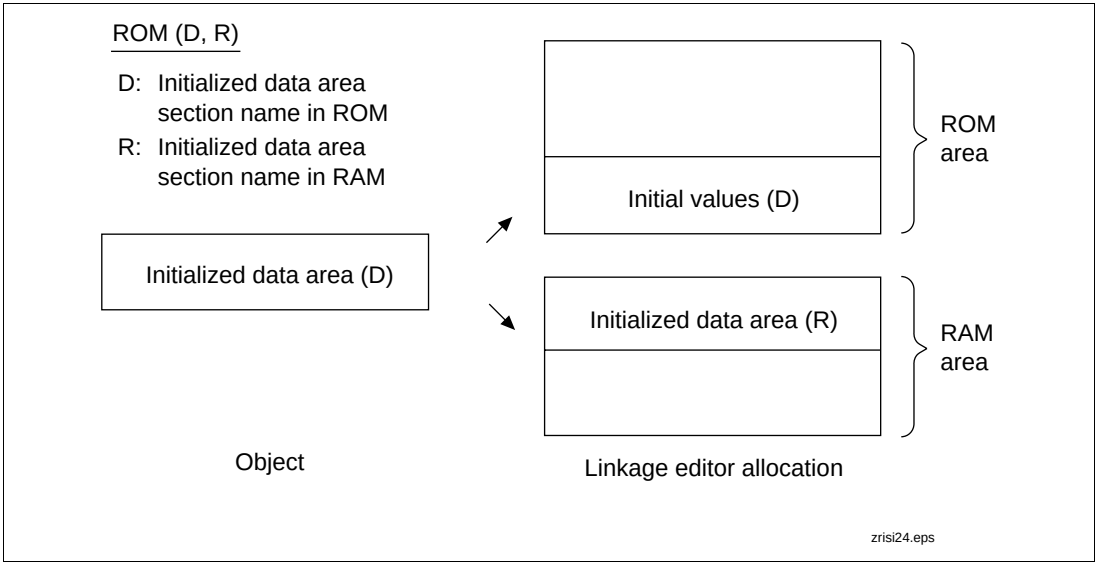


Figure 4.10 Memory Allocation by the ROM Conversion Support Function

2. Referencing of symbols declared in the initialized data area is performed automatically by resolving addresses as pointing to the RAM area address.

The user incorporates processing to copy data in ROM to RAM into the startup routine. Refer to section 1.6.4, Initializing Module Creation, for an example. Refer to the *H Series Linkage Editor User Manual* for details on the ROM conversion support function. This function is supported by the H Series Linkage Editor Version 4 and later.

4.2.2 Precautions on Linkage

Table 4.5 lists methods of dealing with error messages output when linking relocatable object files generated by the C compiler.

Table 4.5 Treating Error Messages During Linkage

Error Message	Manner of Confirmation	Method of Treatment
"CANNOT FIND SECTION" is output.	Are C compiler output section names designated with upper case characters in the linkage editor start option?	Designate a proper section name.
"UNDEFINED EXTERNAL SYMBOL" is output, even though the referenced function has been defined.	Is defined function name correct?	If function name is incorrectly defined, the C compiler judges it to be a new function and does not output an error message. User should correct the function name.
"UNDEFINED EXTERNAL SYMBOL" is output, even though the referenced function has been defined.	Is basic format declaration correct?	If basic format declaration is in error, compiler judges that a function not existing in the program has been referenced. User should correct the basic format declaration.
"UNDEFINED EXTERNAL SYMBOL" is output for an assembly program external definition symbol referenced by a C program.	Is symbol name defined with "_" at the beginning in the assembly program?	Add "_" to the beginning of the assembly program external definition symbol name.
"UNDEFINED EXTERNAL SYMBOL" is output for a C program external definition symbol referenced by an assembly program.	Is symbol name referenced with "_" at the beginning in the assembly program?	Add "_" to the beginning of the assembly program external reference symbol name.
"UNDEFINED EXTERNAL SYMBOL" is output for a symbol beginning with "_ _" (double underline).	—	Designate a standard library for the library file during linkage.
"UNDEFINED EXTERNAL SYMBOL" is output for a symbol other than those mentioned above.	Is C library function designated?	Include a standard library include file in the C program. Also designate a standard library for the library file during linkage.
"UNDEFINED EXTERNAL SYMBOL" is output for a symbol other than those mentioned above.	Is C library function standard I/O library being used?	Create a low standard interface routine and link.

Even if object files have debug information attached to them, that debug information will not be output in the load module file if the -debug option is not designated during linkage. In such cases, source level debugging will not be performed in the simulator/debugger.

4.3 Relation to the Simulator/Debugger

When load modules are executed using the simulator/debugger, there is a possibility that a "MEMORY ACCESS ERROR" will be generated. As a preventative measure, apply one of the following methods:

1. Use the same kind of memory mapping as for the actual machine when using the simulator/debugger (the total byte count for any one section always becomes a multiple of 4).
2. During linkage, link the dummy section created by the following assembly language program after all sections except the P section.

Assembly Language Program:

```
.SECTION DM,DUMMY,ALIGN=1
.RES.B     3
.END
```

Example of Combination during Linkage:

For the command line option:

```
-START=P,C,DM/0400,B,DM,D,DM/01000000
```

For subcommand files:

```
START   P,C,DM(0400),B,DM,D,DM(01000000)
```

Cautions on performing source level debugging using the simulator/debugger are as follows:

1. Use Linkage Editor Version 5.3 or later.
2. Designate the -debug option during both compilation and linkage.
3. There are times when the local symbols of a concerned function cannot be referenced within the function.
4. Only one statement can be displayed when multiple statements have been written into a single source line.
5. Source lines eliminated due to optimization cannot be debugged.
6. Because of the occurrence of line switching, etc. due to optimization, there are cases in which the program execution order or disassemble display will differ from the source listing order.

Example:

C Language Program:

```
12  for (i=0; 1<6; i++)
13  {
14          j = i+1;
15          j++;
16      }
17      j++
```

Simulator/Debugger Disassemble Display:

```
14          j = i+1;
12  for (i=0; 1<6; i++)
17  j++
```

7. There are cases in which the for or while statement will be displayed by a disassemble twice at the entry and exit of loop statements.

Section 5 Questions and Answers

This section is a collection of answers to questions often asked by users.

5.1 const Declaration

Question: After making a const declaration, there was no allocation to the constant area (C) section. Why?

Answer: When const declarations are made for symbols, be aware that the following meanings result:

1. const char msg[]=“sun”;
   Allocation to C section: character string “sun”
2. const char *msg[]={“sun”,”moon”};
   Allocation to C section: character strings “sun” and “moon”
   Allocation to D section: msg[0] and msg[1] (*msg[0] and *msg[1] start addresses)
3. const char *const msg[]={“sun”,”moon”};
   Allocation to C section: character strings “sun” and “moon”, msg[0] and msg[1] (*msg[0] and *msg[1] start addresses)
4. char *const msg[]={“sun”,”moon”};
   Allocation to C section: character strings “sun” and “moon”, msg[0] and msg[1] (*msg[0] and *msg[1] start addresses)

5.2 Reentrants and Standard Libraries

Question: What cautions are necessary when making functions reentrant?

Answer: Functions that use global variables do not become reentrant. Also, even when the intention was to create a reentrant function, if standard libraries are used in the course of using the following standard include files, the function does not become a reentrant because global variables are used.

A reentrant library listing is shown in the table below. Functions marked with a Δ in the table establish _errno variables, and can therefore be executed as reentrant as long as _errno is not referenced in the program.

Table 5.1 Reentrant Library

Standard Include File	Function Name	Reentrant
stddef.h	offsetof	O
assert.h	assert	X
ctype.h	isalnum	O
	isalpha	O
	iscntrl	O
	isdigit	O
	isgraph	O
	islower	O
	isprint	O
	ispunct	O
	isspace	O
	isupper	O
	isxdigit	O
	tolower	O
	toupper	O
math.h	acos	Δ
	asin	Δ
	atan	Δ
	atan2	Δ
	cos	Δ
	sin	Δ
	tan	Δ
	cosh	Δ
	sinh	Δ
	tanh	Δ
	exp	Δ
	frexp	Δ
	ldexp	Δ
	log	Δ
	log10	Δ
	modf	Δ
	pow	Δ

Table 5.1 Reentrant Library (cont)

Standard Include File	Function Name	Reentrant
math.h (cont.)	sqrt	Δ
	ceil	Δ
	fabs	Δ
	floor	Δ
	fmod	Δ
setjmp.h	setjmp	O
	longjmp	O
stdarg.h	va_start	O
	va_arg	O
	va_end	O
stdio.h	fclose	X
	fflush	X
	fopen	X
	freopen	X
	setbuf	X
	setvbuf	X
	fprintf	X
	fscanf	X
	printf	X
	scanf	X
	sprintf	X
	sscanf	X
	vfprintf	X
	vprintf	X
	vsprintf	X
	fgetc	X
	fgets	X
	fputc	X
	fputs	X
	getc	X
	getchar	X
	gets	X

Table 5.1 Reentrant Library (cont)

Standard Include File	Function Name	Reentrant
stdio.h (cont)	putc	X
	putchar	X
	puts	X
	ungetc	X
	fread	X
	fwrite	X
	fseek	X
	ftell	X
	rewind	X
	clearerr	O
	feof	O
	ferror	O
	perror	X
stdlib.h	atof	X
	atoi	X
	atol	X
	strtod	X
	strtol	X
	rand	X
	srand	X
	calloc	X
	free	X
	malloc	X
	realloc	X
	bsearch	O
	qsort	X
	abs	O
	div	X
	labs	O
	ldiv	X

Table 5.1 Reentrant Library (cont)

Standard Include File	Function Name	Reentrant
string.h	memcpy	O
	strcpy	O
	strncpy	O
	strcat	O
	strncat	O
	memcmp	O
	strcmp	O
	strncmp	O
	memchr	O
	strchr	O
	strcspn	O
	strpbrk	O
	strrchr	O
	strspn	O
	strstr	O
	strtok	X
	memset	O
	strerror	O
	strlen	O
	memmove	O

Note: O: Reentrant; X: Non-reentrant; Δ: _errno established.

5.3 Method of Correctly Judging 1-Bit Data

Question: It is difficult to judge whether data with a 1-bit size in the bit field is set or not. Why is this?

Answer: When 1-bit data have been declared as signed, those 1-bit data are interpreted as being sign bits. Consequently, values that can be expressed as 1-bit data become 0 and –1. In order to express 0 and 1, always declare as unsigned.

Example Where Judgment Is Always Wrong:

```
struct{
          char p7:1;
          char p6:1;
          char p5:1;
          char p4:1;
          char p3:1;
          char p2:1;
          char p1:1;
          char p0:1;
}s1;

if(s1.p0 == 1){
       s1.p1 = 0;
}
```

Example of Correct Judgment:

```
struct{
          unsigned char p7:1;
          unsigned char p6:1;
          unsigned char p5:1;
          unsigned char p4:1;
          unsigned char p3:1;
          unsigned char p2:1;
          unsigned char p1:1;
          unsigned char p0:1;
}s1;

if(s1.p0 == 1){
    s1.p1 = 0;
}
```

Note: The generated code will be more efficient when if statement conditional expressions are compared with 0.

5.4 Installation

Question: Startup wasn’t possible despite the fact that compiler, assembler, and linker commands were input. Why?

Answer: Confirm that compiler, assembler, and linker installed directories are included in the environment variable “PATH” designation. Refer to section 1.3, Installation Method, and the software attached materials.

5.5 Specifications and Speeds for Execution Routines

Question: What are the speeds of the execution routines provided by the compiler?

Answer: Below is a table of the execution routine speeds/FPL speeds when on-chip ROM and RAM are used.

Table 5.2 Execution Routine Speeds/FPL Speeds

Classification		Function Name	Stack Volume	No. of Execution Cycles SH-1	No. of Execution Cycles SH-2
Constant operations	Multiplication	_muli	12	24–44	—
	Division	_divbs	4	32	32
		_divbu	0	25	25
		_divls	8	88	88
		_divlu	4	81	81
		_divws	4	39	39
		_divwu	0	32	32
	Remainder	_modbs	8	44–45	44–45
		_modbu	4	30–33	30–33
		_modls	12	99–100	99–100
		_modlu	8	83–85	83–85
		_modws	8	50–51	50–51
		_modwu	4	37–40	37–40
Floating point operations	Addition	_adds	24	88–114	86–114
		_addd	24	147–306	147–196
	Post-increment	_poas	44	15+_adds	15+adds
		_poad	84	38+_addd	38+_addd
	Subtraction	_subs	24	95–134	93–135
		_subd	44	174–336	171–215
		_subdr	44	3+_subd	3+_subd
	Post-decrement	_poss	44	15+_subs	15+_subs
		_posd	84	38+_subd	38+_subd
	Multiplication	_muls	24	108–111	94–98
		_muld	64	263–271	191–197
	Division	_divs	20	135–136	134–135
		_divd	60	743–816	741–766
		_divdr	60	3+_divd	3+_divd
	Comparison	_eqs	20	46	46
		_eqd	32	65	65
		_nes	20	53	53
		_ned	32	67	67

Table 5.2 Execution Routine Speeds/FPL Speeds (cont)

Classification		Function Name	Stack Volume	No. of Execution Cycles SH-1	No. of Execution Cycles SH-2
Floating point operations (cont)	Comparison (cont)	_gts	20	53	53
		_gtd	32	67	67
		_lts	20	53	53
		_ltd	32	67	67
		_ges	20	53	53
		_ged	32	67	67
		_les	20	53	53
		_led	32	67	67
Sign conversion		_negs	0	11	11
		_negd	12	28	28
Conversion		_stod	12	52	52
		_dtos	20	90	90
		_stoi	12	42–225	42–225
		_dtoi	20	61–174	61–174
		_stou	12	42–225	42–225
		_dtou	20	61–174	61–174
		_itos	12	39–194	39–194
		_itod	12	48–203	48–203
		_utos	8	31–186	31–186
		_utod	8	37–192	37–192
Bit field setting		_bfsbs	24	79–147	79–147
		_bfsbu	20	49–97	49–97
		_bfsls	24	79–435	79–435
		_bfslu	20	49–263	49–263
		_bfsws	24	79–243	79–243
		_bfswu	20	49–151	49–151
Bit field referencing		_bfxbs	8	30–99	30–99
		_bfxbu	8	30–99	30–99
		_bfxls	8	30–338	30–338
		_bfxlu	8	30–338	30–338
		_bfxws	8	30–179	30–179

Table 5.2 Execution Routine Speeds/FPL Speeds (cont)

Classification	Function Name	Stack Volume	No. of Execution Cycles	
			SH-1	SH-2
Bit field referencing (cont)	_bfxwu	8	30-179	30-179
Area movement	_quick_evn_mvn	4	12+3*(n/4)	12+3*(n/4)
	_quick_mvn	8	17+3*(n/4)(n ≤ 64)	17+3*(n/4)(n ≤ 64)
			24+1.625*(n/4)(n ≥ 68)	24+1.625*(n/4)(n ≥ 68)
	_quick_odd_mvn	4	12+3*(n/4)	12+3*(n/4)
	_slow_mvn	12	21+5*n+3*((n-1)/4)	21+5*n+3*((n-1)/4)
Character string comparison	_quick_strcmp	12	26+7*(n/4)+5*((n-1)%4)	26+7*(n/4)+5*((n-1)%4)
	_slow_strcmp	20	35+7*n	35+7*n
Character string copy	_quick_strcpy	16	30+6*(n/4)+4*((n-1)%4)	30+6*(n/4)+4*((n-1)%4)
	_slow_strcpy	24	24+6*n+2*((n-1)/4)	24+6*n+2*((n-1)/4)
Left shift	_sftl	4	26–33	26–33
	_sta_sftl0–31	0	9–14	9–14
Right shift	_sftra	4	37–46	37–46
	_sftrl	4	26–33	26–33
	_sta_sftr0–31	0	9–14	9–14
	_sta_sftra0–31	0	9–23	9–23

Note: n = number of bytes

5.6 SH Series Object Compatibility

Question: Are there any problems linking objects when using such compile options as -cpu=sh1 (or sh2, sh3), and -pic=1?

Answer: Fundamentally, there is upward compatibility, so it is possible to link SH-1 objects and SH-3 objects and then execute with the SH-3. In this manner, previous assets can still be used.

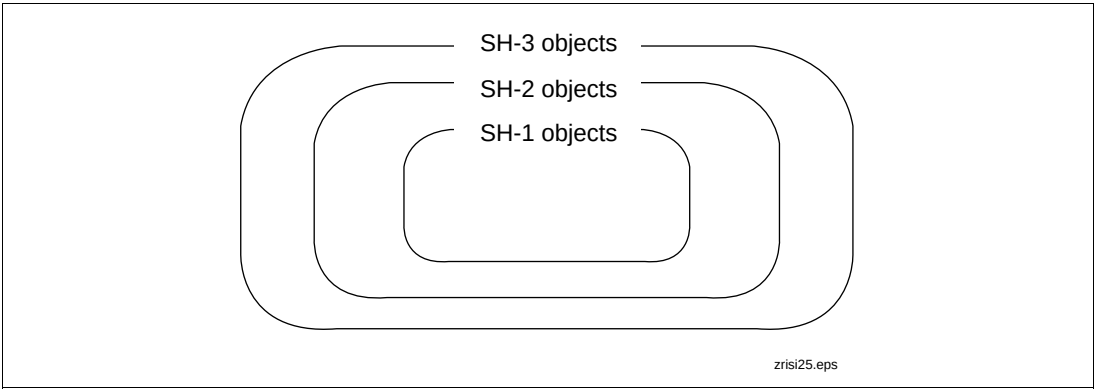


Figure 5.1 Object Compatibility Relationship

Notes: 1. SH-1 and SH-2 are Big Endian. When using them with SH-3, use Big Endian format.
2. Objects compiled with the -pic=1 option can be linked with objects compiled with the -pic=0 option. However, they do not become position independent in this case.
3. For the SH-3, operation upon an interrupt is different from that of the SH-1 and SH-2, and an interrupt handler is necessary.

Refer to section 5.20, Data Allocation, “Endian” Format, concerning the -endian option.

5.7 Concerning Operating Host Machines and OS

Question: What are the available host machines and OS?

Answer: The table below indicates the SH series C compiler (Version 3.0) available machines and OS.

Table 5.3 Host Machines and OS

System Name	OS	Notes
HP9000/700	HP-UX Version 9.0	—
NEWS-RISC	NewsOS Release 4.01R or later	—
PC9801	MS-DOS Version 3.3 or later	Must be able to operate with i386 CPU or later (using DOS-EXTENDER)
IBM-PC/AT	DOS Version 3.3 or later	Must be able to operate with i386 CPU or later (using DOS-EXTENDER)
SPARC	SunOS Release 4.1.1 or later	—
SPARC	Solaris Version 2.1 or later	—

5.8 C Source Level Debugging Not Possible

Question: C source level debugging wasn't possible even though -debug was designated in the compiler options. Why?

Answer: Has -debug been designated as an option both during compilation and during linkage? Or have directories with source programs for compilation been modified?

5.9 Warnings Appear during Inline Development

Question: The warning "Function "function name" in #pragma inline is not expanded" was output during an inline development. Why?

Answer: This warning message does not hinder execution in any way. Refer to the User Manual, Programming Edition, to confirm whether or not the function with the #pragma inline designation fulfills the conditions for inline development. Additionally, the second and later condition/logical operators cannot be inline expanded. Confirm whether or not they have been inline designated.

Example:

```
#pragma inline(A,B)
int A(int a)
{
          if(a<10) return 1;
          else return 0;
}
int B(int a)
{
          if(a<25) return 1;
          else return 0;
}
main()
{
          if ( A(a)==1 && B(a)==1 )
          {         .....
          }
}
```

A(0) is inline expanded, but B(0) is not. (This is because there are cases where the statement evaluates even without evaluating B(a)==1.)

Question: The warning “Function not optimized” was output during an inline development. Why?

Answer: This is due to memory insufficiency. This warning message does not hinder execution in any way. Function sizes become larger when the SH C compiler does inline expansion, and it is conceivable that the amount of memory becomes insufficient during optimization processing, and consequently optimizing processing on a level higher than the expression unit can no longer be performed. Try the following possible countermeasures:

- Do not inline expand large functions.
- Do not inline expand functions called from a large number of locations.
- Reduce the number of functions inline expanded.
- Increase the amount of memory.

5.10 “FUNCTION NOT OPTIMIZED” Appears during Compilation

Question: The warning “Function not optimized” was output during a compile with the “-optimize=1” option. This compile worked before, with the same system environment and same compile option, without any problem. Why?

Answer: This warning message does not hinder execution in any way. The following are possible causes of the message being displayed:

- Compiler limit values have been exceeded

  There are cases in which the compiler exceeds its limit values during optimization processing due to the generation of new internal variables. Such cases can be dealt with by partitioning functions. Refer to the User Manual, Programming Edition 1 concerning the compiler’s limit values.
- Memory is insufficient

  If the SH series C compiler runs out of memory during optimization processing, it halts optimizing above the expression unit level, and outputs this warning. Compilation then continues, but the results obtained for the optimization level are the same as when optimize=0. To avoid this warning, rewrite so as to partition large functions within the C source program. When that is not possible, the only other solution is to increase the memory available to the compiler.

  There is inline expansion. Refer to section 5.9, Warnings Appear During Inline Development.

5.11 “COMPILER VERSION MISMATCH” Appears during Compilation

Question: The message “compiler version mismatch” is output during compilation. Why?

Answer: Confirm that the directories designated in the environment variables “PATH” and “SHC_LIB” are in agreement.

Example: The above message will be output for the following kind of environment variable setting.

```
PATH = (SHC Version 2 path)
SHC_LIB = (SHC Version 3 C compiler unit pathname)
```

5.12 “MEMORY OVERFLOW” Appears during Compilation

Question: The message “memory overflow” is output during compilation. Why?

Answer: Are all of the C compiler unit files in the directory of the pathname designated in the environment variable “SHC_LIB”?

Example: The above message will be output for the following kind of setting. When the environment variable is set to SHC_LIB = /SHC/BIN, the directory organization shown on the left side of figure 5.2 results.

In this case, files under /SHC/BIN and files under /SHC/MSG must all be under the single directory /SHC/BIN.

The proper organization is shown on the right side of figure 5.2.

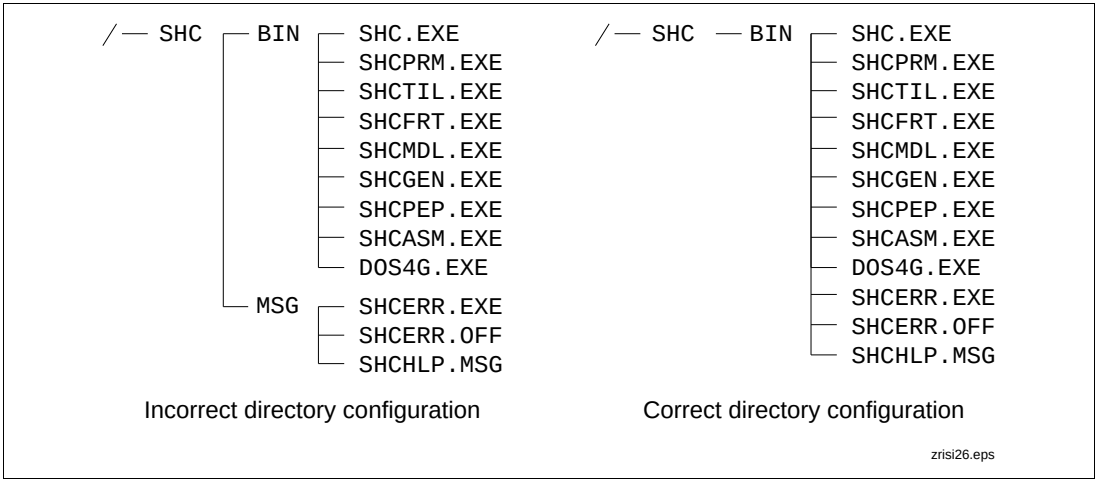


Figure 5.2 Incorrect Directory Configuration vs. Correct Directory Configuration

5.13 “UNDEFINED SYMBOL” Appears during Linkage

Question: The message “UNDEFINED SYMBOL” is output during linkage. Why, and what does it mean?

Answer: Confirm whether or not the libraries are linked. Also, do the declared functions or functions being used actually exist? Refer to section 4.2.2, Precautions on Linkage, for details.

5.14 “RELOCATION SIZE OVERFLOW” Appears during Linkage

Question: The warning message “RELOCATION SIZE OVERFLOW” (error number 108) is output during linkage. How should this be dealt with?

Answer: Confirm that #pragma abs16, #pragma gbr_base and #pragma gbr_base1 have not been designated so as to exceed the area limits.

5.15 “SECTION ATTRIBUTE MISMATCH” Appears during Linkage

Question: The warning message “SECTION ATTRIBUTE MISMATCH” (error number 107) is output during linkage. How should this be dealt with?

Answer: Confirm that there are not different alignments for identical section names.

However, in the Linkage Editor Version 5.3 and later, this warning can be avoided by attaching the ALIGN_SECTION option/subcommand. The ALIGN_SECTION option/subcommand allocates addresses so that sections with identical names but differing boundary adjustment numbers (designated with the align operand of the assembler’s SECTION control instruction) are regarded as being the same section.

5.16 Executing the Transfer of Programs to RAM

Question: With the SH-1, I want to place programs in RAM with a fast execution speed. How should this be done?

Answer: See figure 5.3.

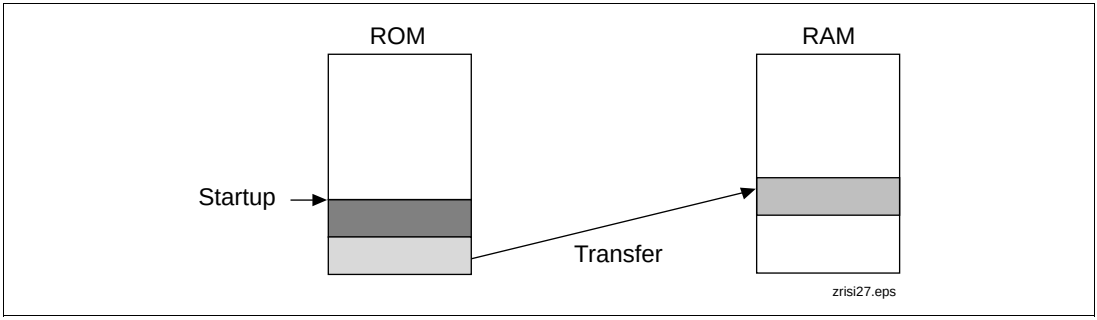


Figure 5.3 Operating Environment

1. Run a program residing in ROM.
2. Transfer to RAM a section that is part of the same program’s code.

When outputting SH-2 or SH-3 usage objects, relocatable load modules can be created if the option to output position independent code (-pic=1) is designated, but this method cannot be used with the SH-1. However, when always copying program code to a fixed address in RAM, it is also possible to execute programs in RAM with the SH-1 by using the linker's ROM conversion support function in the same manner as with initialized data. It is not possible to decide the RAM addresses and copy the program code during execution because addresses are resolved during linkage.

Following is a program example for the kind of section configuration shown in figure 5.4 below.

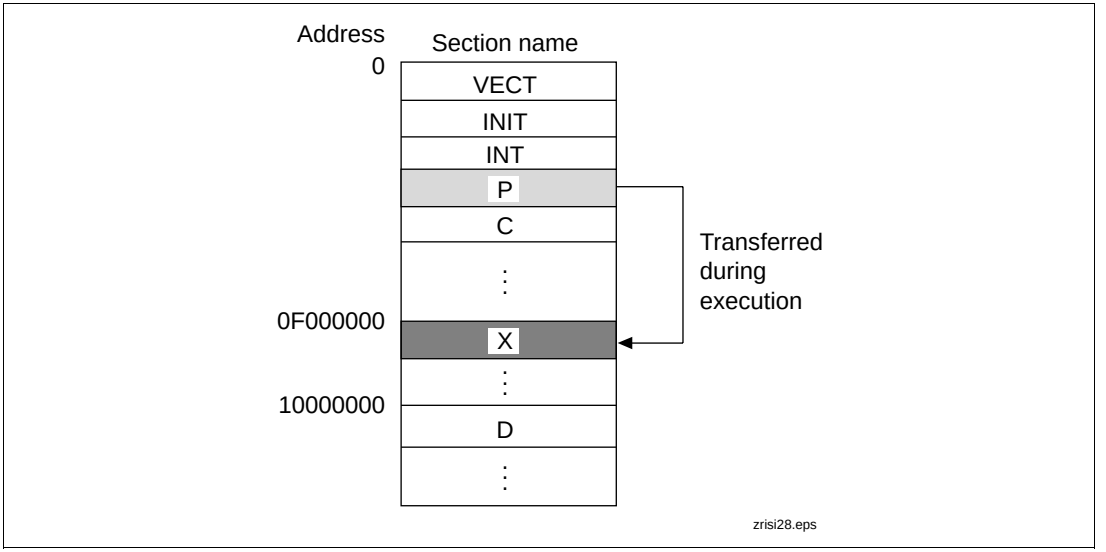


Figure 5.4 Section Configuration

C Language Section:

```
/*********************************************************************/
/*                        file name “init.c”                         */
/*-------------------------------------------------------------------*/
/*     Program section name is made “INIT” with the compile option   */
/*********************************************************************/
#include "sample.h"                        /* Include the Section 1 sample.h   */
extern int *_B_BGN,*_B_END;
extern int *_P_BGN;                        /* P section start address          */
extern int *_X_BGN;                        /* X section start address          */
extern int *_X_END;                        /* X section final address          */
extern void _INITSCT(void);
extern void _INIT();
extern void main();
void _INIT()
{
          _INITSCT();
          main();
          for ( ; ; )
              ;
}
void _INITSCT(void)
{
          int *p,*q;
          for (p = _B_BGN; p <_B_END; p++ )
               *p = 0;
                                   /* Copy from P section to X section         */
          for ( p = _X_BGN, q = _P_BGN; p < _X_END; p++, q++ )
               *p = *q;
}
```

```
/*********************************************************************/
/*                        file name “main.c”                         */
/*-------------------------------------------------------------------*/
/*         Program section name is made the default “P”               */
/*********************************************************************/
int a = 1;
int b;
const int c = 100;
void main(void)
}
                        /* This routine is executed in the copy destination (RAM)*/
          for (; ; )
               ;
}
/*********************************************************************/
/*                        file name “int.c”                          */
/*********************************************************************/
#include "sample.h"                    /* Include the Section 1 sample.h   */
extern int a;                          /* section D code                   */
extern int b;                          /* section B code                   */
extern const int c;                    /* section C code                   */
#pragma interrupt(IRQ0, inv_inst)
/*********************************************************************/
/*                        interrupt module IRQ0                      */
/*********************************************************************/
extern void IRQ0(void)
{
          a = PB_DR;
          PC_DR = c;
}
/*********************************************************************/
/*                        interrupt module inv_inst                  */
/*********************************************************************/
extern void inv_inst(void)
{
          return;
}
```

Assembly Language Section:

```
/*********************************************************************/
/*                      file name “sct.src”                          */
/*********************************************************************/
          .SECTION      P,CODE,ALIGN=4
          .SECTION      X,CODE,ALIGN=4
          .SECTION      B,DATA,ALIGN=4
          .SECTION      C,DATA,ALIGN=4
__P_BGN:  .DATA.L       (STARTOF P)                 ;P section start address
__X_BGN:  .DATA.L       (STARTOF X)                 ;Start address of the P
                                                    section in  RAM
__X_END:  .DATA.L       (STARTOF X) + (SIZEOF X)    ;Final address of the P
                                                    section in  RAM
__B_BGN:  .DATA.L       (STARTOF B)                 ;BBS section start address
__B_END:  .DATA.L       (STARTOF B) + (SIZEOF B)    ;BBS section final address
          .EXPORT       __P_BGN
          .EXPORT       __X_BGN
          .EXPORT       __X_END
          .EXPORT       __B_BGN
          .EXPORT       __B_END
          .END
/*********************************************************************/
/*                      file name “vect.src”                         */
/*********************************************************************/
          .SECTION      VECT,DATA,ALIGN=4
          .IMPORT       _main
          .IMPORT       _inv_inst
          .IMPORT       _IRQ0

          .DATA.L       _main
          .DATA.L       H'FFFFFFC
          .ORG          H'0080
          .DATA.L       _inv_inst
          .ORG          H'0100
          .DATA.L       _IRQ0
          .END
```

Command Designation: Set the command lines as follows:

```
shc -debug -section=P=INIT init.c
shc -debug -section=P=INT int.c
shc -debug main.c
asmsh  sct.src -debug
asmsh  vect.src  -debug
lnk -sub=rom.sub
```

Linker Option File:

```
/*********************************************************************/
/*                        file name "rom.sub"                        */
/*********************************************************************/
debug
input vect, sct, init, int, main
ROM    (P,X)                        ;Resolve address so that P section is allocated to X
start VECT(0),INIT,INT,P,C,D(10000000),X(0f000000)
                                    ;VECT, INIT, INT, P, C, D are stationed in ROM, X
                                    is  stationed in RAM
output sample.abs
print sample.map
exit
```

Due to the above programming, the section P program is copied into section X and executed. Because section INIT is the copying routine, it must be a separate section from the copied routine. Through this, the main program (section P) is executed in the copy destination.

Caution: C source level debugging is not possible for programs copied from ROM to RAM.

5.17 Priority of Include Designations

Question: The fact that there are a variety of options for including files is confusing. What are the purposes and priority rankings of the options?

Answer: Designation of include file reference paths is performed by options or environment variables.

Files enclosed within "<" and ">" are read in from the directory designated by the -include option, and when multiple directories have been designated, they are referenced in the order of designation. When files are not found in the directories designated by the -include option, each directory designated by first the environment variable SHC_INC, and then the system directory, (SHC_LIB) is referenced in order.

Files enclosed within quotation marks (“ ”) are referenced beginning from the current directory. When there is no current directory, referencing occurs in accordance with the rules noted above.

The include file reference path priority, ranked intuitively, is as follows:

```
-inc > SHC_INC > SHC_LIB
```

Additionally, there is a -preinclude option for the compulsory reading of files it designates in a manner different from those noted above. When this option is designated, the files designated by it are inserted at the head of the compiled files and compilation is performed.

If the contents of such items as #pragma or test data, which one wishes to read in on a temporary basis, are read in as a separate file with this option, recompiling is possible without having to treat the source file.

5.18 Compilation Batch Files

Question: When there are many items designated with compile options, designating the same items each time is bothersome. Isn’t there a better way?

Answer: The -subcommand option (-subcommand=<file_name>) is used during compilation. This option can be designated multiple times within a command line. List command line arguments in a subcommand file delimited with spaces, returns or tabs. The contents of subcommand files are developed in the command line argument subcommand designation position.

Note that the -subcommand option cannot be designated within subcommand files.

Example:

The example below is equivalent to inputting the following command line:

```
shc -optimize=1 -listfile -debug -cpu=sh2 -pic=1 -size -euc -endian=big
test.c
```

Command Line:

```
shc -sub=test.sub test.c
```

Contents of test.c:

```
-optimize
-listfile
-debug
-cpu=sh2
-pic=1
-size
-euc
-endian=big
```

5.19 Notation of Japanese within Programs

Question: Source files have been developed on both workstations and personal computers, but management of those source files is difficult because the Kanji character codes are different for workstations and personal computers. Is there a good method for managing this?

Answer: When the Kanji character code notation is shift-JIS, use the compiler option -sj when compiling on a workstation (which uses EUC code). In the opposite case, when the notation is in EUC code and compiling on a personal computer, designate the compiler option -euc and compile. Even in a workstation network environment where EUC and shift-JIS exist together, compiling with either Kanji character code is possible through use of the compile option designation. Compilation is possible with the Kanji character code used in the target (installed machine).

Table 5.4 System and Kanji Character Code Correspondence

Host	Default
SPARC	EUC
HP9000/700	Shift-JIS
NEWS	Character code indicated by the environment variable “LANG”
PC-9801	Shift-JIS
IBM-PC	Shift-JIS

Example: When the source is written on a workstation (SPARC) and the compiling is done on a personal computer (IBM-PC), if compilation is done with the -sj option added, there is no need to be concerned about character conversions of Kanji character codes within character strings.

5.20 Data Allocation, “Endian” Format

Question: Is SH data allocation Big Endian or Little Endian?

Answer: The Hitachi SuperH RISC engine family is Big Endian. However, the SH-3 supports the -endian=Big(Little) option corresponding to the CPU Big/Little switching function. Compatibility with Little Endian CPU is increased through this.

Caution:

1. The -endian option can be combined with -cpu option suboptions at will (but Little Endian object programs can only be executed with the SH-3).
2. Big Endian objects and Little Endian objects cannot be used together.
3. Program execution results will sometimes be affected by differences in Endian.

Example: Coding for which differences in Endian format will have an effect:

```
f(){
int a=0x12345678;
char *p;

        p=((char*)(&a));

        if(*p==0x12){       (1)     }
        else{  (2)   }
}
```

In this case, processing (1) is executed for Big Endian, but for Little Endian, *p is 0x78, so processing (2) is executed. (Refer to the User Manual, Programming Edition, for details on data allocation.) There are seven kinds of standard libraries, as listed below. Link the libraries shown in table 5.5 by using combinations of the -cpu option, -pic option and -endian option.

- shclib.lib (SH-1 usage)
- shcnpic.lib (SH-2 usage position independent code non-corresponding)
- shcpic.lib (SH-2 usage position independent code corresponding)
- shc3npb.lib (SH-3 usage position independent code non-corresponding, Big Endian)
- shc3pb.lib (SH-3 usage position independent code corresponding, Big Endian)
- shc3npl.lib (SH-3 usage position independent code non-corresponding, Little Endian)
- shc3pl.lib (SH-3 usage position independent code corresponding, Little Endian)

Table 5.5 Relationship between Standard Libraries and Compile Options

	Big Endian		Little Endian	
CPU	**pic = 0**	**pic = 1**	**pic = 0**	**pic = 1**
SH-1	shclib.lib	—	—	—
SH-2	shcnpic.lib	shcpic.lib	—	—
SH-3	shc3npic.lib	shc3pb.lib	shc3npl.lib	shc3pl.lib

Appendix A Compiler Options

A.1 Compiler Options

Table A.1 indicates the compiler option formats and abbreviated formats, and interpretation when options are omitted. Characters with an underscore (_) indicate the abbreviated format, and bold case characters (**abc**) are interpreted as being indicated when an item is omitted.

Table A.1 Compiler Options

Item	Format	Suboptions	Contents of Designation
Optimizing level	optimize=	0	Unoptimized objects output
		1	Optimized objects output
List content and format*1	show=	**so**urce | nosource	Source list exists/doesn't
		object | noobject	Object list exists/doesn't
		statistics | nostatistics	Statistical information exists/doesn't
		include | **noi**include	List after include development exists/doesn't
		expansion | **noe**xpansion	List after macro development exists/doesn't
		width=<numerical value>*2	Maximum number of characters per line, numerical value: 0, 80 to 132
		length=<numerical value>*2	Maximum number of characters per page, numerical value: 0, 40 to 255
		When abbrieviated: (w = 132, l = 66)	—
List file	listfile [=<listfile name>]*3		Output
	nolistfile		No output
Object file	objectfile=<objectfile name>		Output
Object format	code=	**m**achincode	Machine language program output
		asmcode	Assembly source program output
Debug information	debug	—	Output
	nodebug	—	No output

Table A.1 Compiler Options (cont)

Item	Format	Suboptions	Contents of Designation
Macro name definition	define=	<macro name> =<name>	<name> defined as <macro name>
		<macro name> =<constant>	<constant> defined as <macro name>
		<macro name>*4	Assumed as defining a <macro name>
Include file	include=<pathname>*5		Designates the destination pathname for include file inclusion (multiple designations possible)
Section name	section=	program=<section name>	Program area section name designation
		const=<section name>	Constant area section name designation
		data=<section name>	Initialized data area section name designation
		bss=<section name>	Uninitialized data area section name designation
		When abbreviated: (p = P, c = C, d = D, b = B)	—
Help message	help*6	—	Output
CPU classification	cpu=	sh1*7	SH-1 objects generated
		sh2	SH-2 objects generated
		sh3	SH-3 objects generated*8
Position independent code	pic=	0	Position independent code not generated
		1	Position independent code generated*9
Character string output area	string=	**c**onst	Output to constant section (C)
		data	Output to initializing data section (D)*10
Comment nesting	comment=	nest	Allow comment nesting
		nonest	Do not allow comment nesting*11

Table A.1 Compiler Options (cont)

Item	Format	Suboptions	Contents of Designation
Optimization method selection	speed	—	Execution speed priority code generation
	nospeed	—	Implement optimization balancing execution speed and size
	size	—	Size priority code generation*12
Selection of Japanese code in character strings	euc*13	—	Selects euc code
	sjis	—	Selects sjis code
Subcommand file designation	subcommand=<file name>		Include command options from the file designated by <file name>*14
Form of division	division=	cpu	Use the CPU division instructions
		peripheral	Use the divider (with interrupt mask)
		nomask	Use the divider (without interrupt mask)*15
Memory bit lineup order designation	endian=	big	Big Endian
		little	Little Endian*16
Inline development specifications	inline	—	Designates whether or not to perform inline development
	inline=	<numerical value>	Develop when performing inline development
	noinline	—	Designates the function size limit*17
Default header file designation	preinclude=<file name>		Include the contents of designated file at the head of the compile unit
MACH, MACL register preservation	macsave=	0	Do not preserve MACH, MACL registers with a function call
		1	Preserve MACH, MACL registers with a function call*18

Notes:
1. The show option becomes effective when listfile is designated.
2. When show=width=0 or show=length=0 are designated, they are interpreted as follows:
   show=width=0: regarded as being 1 line until line return code is output.
   show=length=0: maximum number of lines is not set; new pages are not carried out.
3. When file name designation has been omitted, files with standard expanders added are generated with the same file name as the source file.
4. The specifications for macro names that can be designated by option are shown in table A.2 below.
5. Refer to the User Manual Appendix A.1.13, Preprocessor Specifications, for the include file referencing methods.
6. Other options become ineffective when this option is designated.
7. In Version 2.0, cpu=7000 designated the SH-1 and cpu=7600 designated the SH-2. In order to preserve compatibility, these suboptions can also be designated.

8. The linked libraries will differ, depending on the cpu, pic, and endian options. Refer to section 5.20, Data Allocation “Endian” Format, concerning the correspondence between options and standard libraries, for details.
9. Cautions concerning the use of position independent code:
   a. When pic=1 is designated, after linkage program sections can be placed in arbitrary addresses and executed (data sections cannot be placed in addresses other than those decided during linkage).
   b. When executing as position independent code, function addresses cannot be designated as initial values.

      Example:

      ```
      extern int f();
      int(*fp)()=f;                ← can be designated
      ```

   c. When cpu=SH1 is designated, the pic=1 designation is ignored.
10. When string=const has been designated, the same character string can be jointly owned.
11. Comment nesting example:

    ```
    /* comment
           int i; /* nest1 /* nest2 */ */
    */
    ```

    When comment=nest is designated: the underlined section becomes a nested comment

    When comment=nonest is designated: the comments are judged as ending with “nest2 */” and the following “*/” causes an error.
12. Program execution speed will improve if the speed option is designated, but there are cases where the size will increase.
13. Abbreviated characters differ depending on the host machine. Refer to section 5.19, Notation of Japanese Within Programs, for details.
14. The subcommand option can be designated multiple times within a command line. List command line arguments in a subcommand file delimited with spaces, returns, or tabs. The subcommand option cannot be designated within subcommand files. Refer to section 5.18, Compilation Batch Files, for details.
15. The execution routine for integer format division or remainder operations within the C source is selected from among the three following items by the compiler option. This option can be combined with the cpu option suboptions at will, but object programs with peripheral or nomask designations can only be executed with the SH-2.
    a. cpu: Selects an execution routine with DIV1 instructions.
    b. peripheral: Selects an execution routine using the divider. Interrupts other than NMI are prohibited during execution of this routine (15 is set in the interrupt mask).
    c. nomask: Selects an execution routine using the divider (the interrupt mask does not change).

    Be careful of the following points when peripheral or nomask are designated:

    a. Zero division checks and errno setting are not performed.
    b. Operation is not guaranteed if an interrupt occurs during divider operation and the divider is used in the interrupt routine.
    c. Overflow interrupts are not supported.

d. There are cases when the operation results for zero division, overflow interrupts, etc., in accordance with the divider specifications, will be different from when there are cpu suboption designations.

16. This option can be combined with the cpu option suboptions at will, but Little Endian object programs can only be executed with the SH-3.
17. This designates whether or not to perform automatic inline development of functions. The suboption numerical value indicates the maximum size of the inline developed function by the function's node count (total number of argument, operator, etc. expressions excepting the declaration section).

    When the speed option is designated the default is inline=20. When the nospeed, size options are designated, or when the optimize=0 option is designated the default is noinline.
18. Functions compiled with macsave=0 cannot be called from functions compiled with macsave=1. Conversely, functions compiled with macsave=1 can be called from functions compiled with macsave=0.

Table A.2 Macro Names, Names, and Constants That Can Be Designated with the Define Option

Item	Explanation
Macro names	Character strings beginning with an English character or an underline, then followed by 0 or more English characters, underlines or numbers
Names	Character strings beginning with an English character or an underline, then followed by 0 or more English characters, underlines or numbers
Constants	Character strings that are a repetition of one or more numbers (0 to 9), or a repetition of one or more numbers followed by a period, and then followed by 0 or more numbers. Or else, hexadecimal numbers beginning with 0x.

Appendix B Changes in Version 3.0

This section describes the changes from Version 2.0 to Version 3.0 of the SH Series C Compiler.

B.1 Additions and Improvements

A summary of the functions added to the SHC Compiler Version 3.0 is given below.

B.1.1 Optimization Strengthened

The Version 3.0 optimization has two options for giving importance to speed (-speed option) or to size (-size option), and each of these optimizing functions has been strengthened. For the purpose of speed, strengthening of loop optimization and support of inline development have been realized. Concerning size, such strengthening as size importance instruction generation and the joining of redundant processes have been realized.

B.1.2 SH-3 Support

In addition to the SH-1 and SH-2, SH-3 object generation can be designated (-cpu=sh3 option). The following are supported as SH-3 usage functions:

1. The -endian option (-endian=big or -endian=little) corresponding to the function for setting bit order in memory is supported.
2. The extended intrinsic function prefetch for generating cache prefetch instructions (PREF) is supported.

B.1.3 Compiler Limit Value Extension

The compiler limit values have been extended for the items in table B.1.

Table B.1 Compiler Limit Values

Item	Version 2.0	Version 3.0
Source programs that can be compiled at a time	16 files	No limit
Source lines per one file	32,767 lines	65,535 lines
Source lines per one total compile unit	32,767 lines	No limit
#include nest levels	8 levels	30 levels

B.1.4 Support of Japanese Language Code in Character Strings

Shift-JIS and EUC Japanese language codes can be written into programs as character string data. When the input code is shift-JIS (-sjis option) the output code is also shift-JIS, and when the input code is EUC (-euc option) the output code is also EUC. However, the GUI does not handle Japanese code data display at present.

B.1.5 Option Designation by File

By designating file names with the -subcommand option, it is now possible to include options from within files. This eliminates the need to designate a complex group of options each time with a command line.

B.1.6 Use of the SH-2 Divider

The -division option is supported to allow use of the SH-2 divider.

B.1.7 Inline Expansion of C Functions

If the -speed option is designated, the compiler automatically performs the inline expansion of small functions. This option can, through use of the -inline option, change the size conditions for functions to be inline developed. The object functions for inline development can be clearly stated with #pragma designations.

Example (Inline Development of C Functions):

```
#pragma inline (func)
int func(int a, int b)
{
return (a+b)/2;
}
main()
{
i=func(10, 20);                    /* Expanded into i = (10+20)/2       */
}
```

B.1.8 Inline Assembler Functions

#pragma inline_asm designates the inline assembler notation for user functions. However, when performing inline embedding with #pragma inline_asm, make the compiler output assembler source (-code=asmcode option).

Example (Inline Assembler Function):

```
#pragma inline_asm(rotl)
int rotl(int a)
{
        ROTL  R4
        MOV   R4,R0
}
main()
{
        i=rotl(i);          /* variable i is set in register R4 and the rotl
                               function code is used                    */
}
```

B.1.9 Use of the Short Address Designation (2-Byte Address Variables)

#pragma abs16(<variable name>|<function name>,...) can designate that variables or functions be allocated to the address range (–32,768 to 32,767) that can be designated in 2 bytes. This designation enables a reduction in the size of objects referencing such variables or functions.

B.1.10 Use of GBR Relative Addressing

#pragma gbr_base(<variable name>,...) can designate that variables be allowed to be referenced using the GBR relative addressing mode. This designation not only enables a reduction in the size of objects referencing variables, but also allows use of memory bit manipulation instructions unique to the GBR relative addressing mode.

B.1.11 Register Save/Restore Control

#pragma noregsave(<function name>,...) can designate the suppression of register saves and restores at the entrances and exits of functions. This can be used to create high speed, compact functions without register saves and restores. Functions with a #pragma noregsave designation cannot be called from ordinary functions, but they can be called from C language functions (#pragma regsave) that are clearly designated to call #pragma noregsave. Program size can be decreased and execution speed improved by designating #pragma noregsave for frequently executed functions.

B.1.12 Improvement of Differences with ANSI Specifications

1. The standard header file <errno.h> is supported.
2. Pointer values to void format can now be designated in the initial values of pointer format data to other than void format.
3. When there is no left parenthesis immediately following a macro name that has attached arguments, processing is now possible as an ordinary name instead of as a macro name.

4. Even if typedef names and struct tag names are within the same scope, no error due to name conflict will result.
5. enum names can be designated as case labels, bit fields, or array sizes.
6. Comparison operations between two groups of void* format data will no longer result in an error.
7. When initializing character arrays with character strings, no error will result even if the initialization is enclosed within {}.
8. The ANSI standard library function memmove has been added to the library functions.

   ```
   #include <string.h>
   void *memmove(void *s1, const void *s2, size_t n);
   ```

   An area of n bytes from the address pointed to by s2 is copied to the area beginning at the address pointed to by s1. In this case, the results are guaranteed even if the copy source area and copy destination area overlap. FILE, size_t, and ptrdiff_t, were defined as macros with the #define, but they were modified to definition by typedef.

B.1.13 Referencing from Interrupt Functions

For functions declared as interrupt functions, referencing of functions within the same file was not possible, but this limitation has been removed.

B.2 Additions to the Compiler Options

Only additions to the compiler options are listed here. Refer to Appendix A, Compiler Options, for details.

- SH-3 object generation
- Selection of Japanese language code in character strings
- Subcommand file designation
- Division method (use or not of an interrupt mask) designation
- Size priority code generation
- Memory bit order designation
- Inline development
- Default header file designation
- Preservation of MACH, MACL registers

Appendix C ASCII Codes

Lower 4 Bits	Upper 4 Bits							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	NULL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	‘	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL