To our customers,

Old Company Name in Catalogs and Other Documents

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

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

April 1st, 2010
Renesas Electronics Corporation

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

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

RENESAS

Notice

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

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

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

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

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

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

Application Note

16

HI SeriesOS

Application Note

Renesas Microcomputer Development Environment System

Renesas Electronics
www.renesas-electoronics.com

Rev.3.00 2005.01

Keep safety first in your circuit designs!

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

Notes regarding these materials

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

Preface

The HI series OS (operating system) is a machine-installed realtime multitasking OS manufactured by Renesas Technology Corp. based on the µITRON specifications.

This application note is targeted towards the following engineers.

Targeted Engineers	Requirements
Engineers who understand the ITRON specifications	Must know the terms used in the ITRON specifications
Engineers who understand the outline of the HI series OS	Must understand the functions provided by the HI-series OS
Engineers who plan to develop systems using the HI series OS	Must have programming experience in assembly language and C language and understand written programs

This application note gives supplementary information about the development of applications using the HI series OS and answers questions frequently asked by users of the HI series OS.

Application Note Structure:

This application note contains the following four sections:

Section	Contents
Section 1 Functions of the HI series OS	Describes the functions and objects of the HI series OS and answers related FAQs.
Section 2 Creation of application programs	Describes creation of application programs using the HI series OS and answers related FAQs.
Section 3 Configuration	Describes configuration of the HI series OS and answers related FAQs.
Section 4 Device-dependent specifications	Answers FAQs related to the device-dependent specifications of the HI series OS.
Section 5 Debugging	Describes debugging of application programs using the HI series OS and answers related FAQs.

For details of each OS in the HI series, refer also to the user's manual of the OS to fully understand it.

Related Manuals:

Please read also the following manuals related to this application note.

- User's manual of the HI series OS used
- Hardware manual and programming manual of the microcomputer used
- User's manual of the cross compiler used
- High-performance Embedded Workshop (HEW) User's Manual

Terms and Symbols Used in this Application Note

Term or Symbol	Description
H' and D'	"H'" is a prefix for a hexadecimal value and "D'" is for decimal. A value without a prefix is a decimal value.
Copy-back	A caching method used in the SH-4 series microcomputer. In the SH-3 and SH3-DSP series microcomputers, the equivalent method is called "write-back", but both are collectively called "copy-back" in this application note.

Descriptions of Product Names

Product Name	Description
HI7000/4	OS based on the μITRON4.0 specifications for the SH-1, SH-2, and SH2-DSP series microcomputers manufactured by Renesas Technology Corp.
HI7700/4	OS based on the μITRON4.0 specifications for the SH-3, SH3-DSP, and SH4AL-DSP series microcomputers manufactured by Renesas Technology Corp.
HI7750/4	OS based on the μITRON4.0 specifications for the SH-4 and SH-4A series microcomputers manufactured by Renesas Technology Corp.
HI7000/4 series	Collective name for HI7000/4, HI7700/4, and HI7750/4.
HI2000/3	OS based on the μITRON3.0 specifications for the H8S family microcomputers manufactured by Renesas Technology Corp.
HI1000/4	OS based on the μITRON4.0 specifications for the H8SX family microcomputers manufactured by Renesas Technology Corp.
HEW	Abbreviation of High-performance Embedded Workshop, an integrated system development environment manufactured by Renesas Technology Corp.

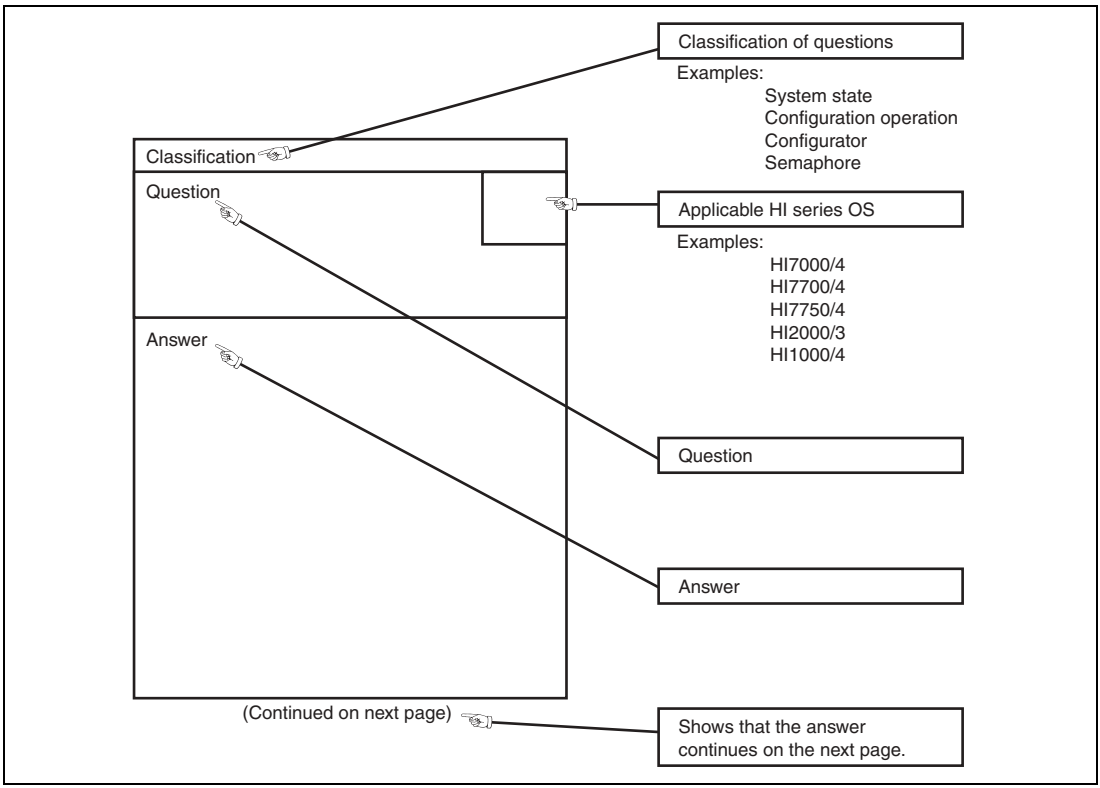
μITRON Specifications Referred to in this Application Note

This application note uses the terms of the μITRON4.0 specifications. When using the OS based on the μITRON3.0 specifications, note the following differences in terms.

Term	Description
Service call	A term used in the μITRON4.0 specifications. In the μITRON3.0 specifications, the equivalent is called a "system call", but both are collectively called a "service call" in this application note.
Task context	Name of a system state in the μITRON4.0 specifications. The name depends on the version of the μITRON specifications (for example, it is called task portion in the μITRON3.0 specifications), but all are collectively called "task context" in this application note.
Non-task context	Name of a system state in the μITRON4.0 specifications. The name depends on the version of the μITRON specifications (for example, it is called non-task portion or task-independent portion in the μITRON3.0 specifications), but all are collectively called "non-task context" in this application note.
Interrupt mask bits	Collective name for all interrupt mask bits in the status register (SR) of the SuperH™ family microcomputers and in the condition code register (CCR) and extended register (EXR) of the H8S family microcomputers and H8SX family microcomputers.
Object	Targets to be manipulated by service calls are collectively called "objects"; these include tasks, semaphores, event flags, mailboxes, message buffers, fixed-length memory pools, variable-length memory pools, cyclic handlers, alarm handlers, and overrun handlers.
Cyclic handler	An object in the μITRON4.0 specifications. In the μITRON3.0 specifications, the equivalent object is called a "cyclic start handler", but both are collectively called a "cyclic handler" in this application note.
Initialization routine	A term used in the μITRON4.0 specifications. In the μITRON3.0 specifications, the equivalent is called a "system initialization handler", but both are collectively called an "initialization routine" in this application note.

FAQ Description Format:

This application note answers FAQs in the following format:



Contents

Figures

Section 1 Functions of the HI Series OS

Section 2 Application Program Creation

Section 3 Configuration

Section 4 Device-Dependent Specifications

Section 5 Debugging

Section 1 Functions of the HI Series OS

1.1 System State

The state of the HI series OS system is classified into one of the following two contexts.

Table 1.1 System State

Name	System State
Task context (including task portion)	State or context in which a task is being executed.
Non-task context (including non-task portion or task-independent portion)	State or context in which an interrupt handler, an interrupt service routine, or a time event handler, which is not a task, is being executed.

When issuing a service call, note the system state. When specialized service calls are provided for the task context and non-task context, respectively, check the system state and issue an appropriate service call.

Table 1.2 Difference in Service Calls Due to System State

Context	Service Call	Description
Task context	xxx_yyy*	Wait state can be entered.
Non-task context	ixxx_yyy*	Wait state cannot be entered.

Note: * Some service calls use the same name for the task context and non-task context (such as sns_yyy). For details on service calls, refer to the user's manual of the HI series OS used.

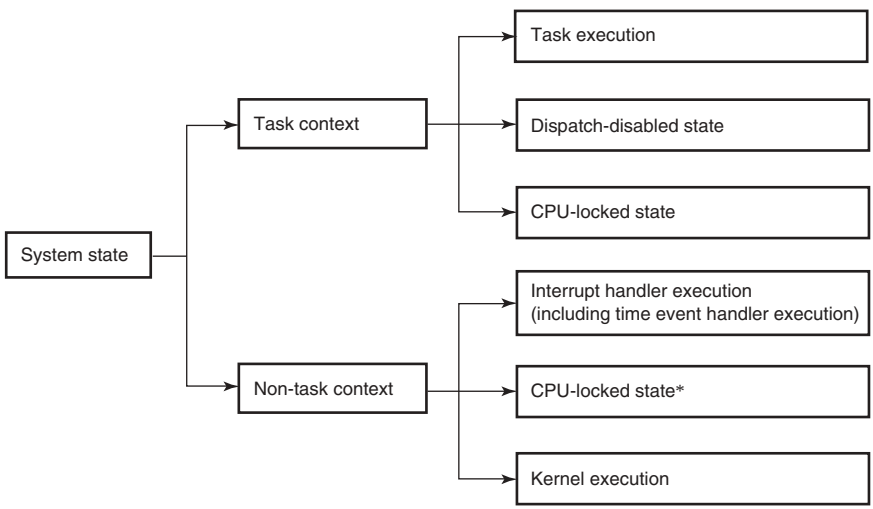
The system state can be checked according to the value of the interrupt mask bits (IMASK value).

The state of the HI series OS system is also classified in a different way as follows.

Table 1.3 Dispatch-Disabled State and CPU-Locked State

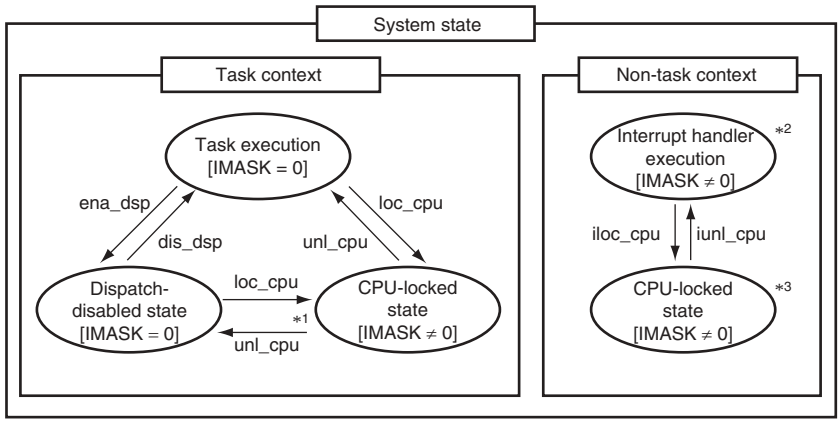
Name		System State
Dispatch-disabled/ dispatch enabled state	Dispatch-disabled state	Task context state in which an interrupt can be accepted but dispatch processing is not performed (task switching is not generated).
	Dispatch-enabled state	The dispatch-disabled state is canceled.
CPU-locked/ CPU-unlocked state	CPU-locked state	No interrupt is accepted or no dispatch processing is performed.
	CPU-unlocked state	The CPU-locked state is canceled.

These states cannot be determined through the value of the interrupt mask bits (IMASK value). They may be recognized as the task context even when the interrupt mask bit value (IMASK value) is not 0. See figure 1.1 for the state of the HI series OS system.



Note: * Not supported in the HI2000/3.

Figure 1.1 State of the HI Series OS System



Notes: 1. In the HI2000/3, the task execution state is entered when unl_cpu is issued. The dispatch-disabled state is not entered.
2. The interrupt handler execution state includes timer handler execution and initialization handler execution states.
3. This state is not supported in the HI2000/3.

Figure 1.2 System State and Interrupt Mask Bit Value

For the relationship between the application program and the system state, refer to section 2.6, Application Program Types.

Table 1.4 shows the priority of processing among tasks, the dispatcher (during kernel execution), and interrupt handlers.

Table 1.4 Priority of Processing

Priority	Processing
High	Interrupt handler, time event handler, CPU exception handler, etc.
↕	Dispatcher (part of processing performed by the HI series OS)
Low	Task

- The interrupt handler takes priority over the dispatcher.
- The priority of a time event handler (including the cyclic handler, alarm handler, and overrun handler) is equal to or lower than the priority of the timer interrupt handler which performs time management processing, and is higher than that of the dispatcher.
- The CPU exception handler takes priority over both the dispatcher and the processing that generated the CPU exception.
- The task has a lower priority than the dispatcher.

1.1.1 FAQs about System State

This section answers questions about system state which are frequently asked by users of the HI series OS.

FAQ Contents:

(1) Common Subroutine in Task Context or Non-Task Context

Classification: System state

Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4

Please explain how to distinguish the system state, between the task context and non-task context, in which a common subroutine is called?

Answer	HI7000/4	HI7700/4	HI7750/4		HI1000/4

The system state in which a service call is issued can be checked by the sns_ctx service call (referring to the context). When TRUE (= 0) is passed as return parameter "BOOL state", the subroutine calling state is the non-task context. When FALSE (≠ 0) is returned, the state is the task context. Figure 1.3 shows a sample program for checking the context.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

void Common_Sub_Routine(VP_INT exinf)
{

BOLL state;

  (description omitted)

   state = sns_ctx();
   if(state == TURE){                    /* Call from non-task context */
      (processing description omitted)
   }
   else{                                 /* Call from task context */
      (processing description omitted)
   }

}
```

Figure 1.3 Sample Code for Context Check (HI7000/4 Series and HI1000/4)

(Continued on next page)

(Continued from previous page)

Answer				HI2000/3	

The system state in which a subroutine is called can be checked by the ref_ims system call (referring to the interrupt mask level). When 0 is passed as return parameter "UINT imask", the subroutine calling state is the task portion.
When the return value is not 0, the state is the non-task portion. Figure 1.4 shows a sample program for checking the context.

```
#include "hi2000.h"

void Common_Sub_Routine(INT stacd)
{

ER   ercd;
UINT imask;

    (description omitted)

    ercd = ref_ims(&imask);
    if(imask != 0){
        /* Processing when a subroutine is called from a non-task context */
        /* or from CPU-locked state                                       */
    }
    else{
        /* Processing when a subroutine is called from a task context */
    }

}
```

Figure 1.4 Sample Code for Context Check (HI2000/3)

When a ref_ims system call is issued in the CPU-locked state during task portion execution, the value passed through return parameter UINT imask is not 0 and the non-task context is recognized.

Since the non-task context and CPU-locked state cannot be distinguished in the HI2000/3 even when the ref_ims system call is used, the application must prepare a means for distinguishing them (for example, using a specialized parameter in common subroutines).

(2) Using the CPU Exclusively

Classification: System state

Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4

What is the best way to disable all tasks (including the kernel) during execution of a specific task?

Answer

The loc_cpu service call should be used.

After loc_cpu is executed, interrupts or task switching below the kernel interrupt mask level are disabled. Note the kernel interrupt mask level setting because interrupts equal to or higher than the kernel interrupt mask level are accepted.

After required processing to exclusively use the CPU is completed, be sure to cancel the CPU-locked state by the unl_cpu service call.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

#pragma noregsave(task)

void task(VP_INT exinf)
{

BOLL state;

    (description omitted)

   loc_cpu();                /* Enters the CPU-locked state */
     /* Starts processing in the CPU-locked state */

    (processing description omitted)

     /* Terminates processing in the CPU-locked state */
   unl_cpu();                /* Cancels the CPU-locked state */

    (description omitted)

}
```

Figure 1.5 Sample Code Using loc_cpu

1.2 Objects

1.2.1 What Is an Object?

The targets of manipulation by service calls, such as tasks, are collectively called objects.

Multiple objects can be created for each object type, and these are identified by ID numbers.

1.2.2 ID Assignment

An ID number is assigned for each object when the object is created through the following methods.

Table 1.5 ID Assignment for Objects

HI Series OS	ID Assignment Method
HI7000/4 series	Assignment by the configurator
	Assignment by a service call
HI2000/3	Assignment by a setup table
HI1000/4	Assignment by the configurator

Because the HI2000/3 and HI1000/4 do not provide the dynamic assignment method (assignment by a service call), the IDs must be assigned by a setup table or configurator in advance.

1.2.3 FAQs about Objects

This section answers questions about objects which are frequently asked by users of the HI series OS.

FAQ Contents:

(1) Registered Task and Task ID

Classification: Object

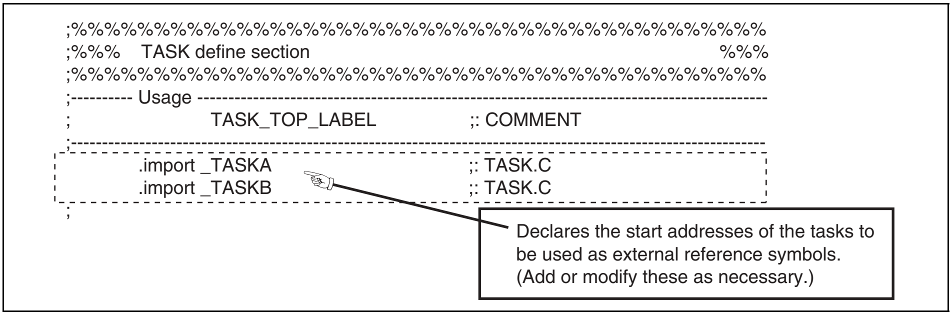
Question				HI2000/3	

Are IDs automatically assigned to tasks in the order of task registration starting from ID 1, or can any ID (value) be assigned to a task in a special method?

Answer

IDs starting from 1 are automatically assigned to tasks in the order of task registration.

Because the HI2000/3 does not provide the dynamic task creation function, tasks must be defined in advance. Figure 1.6 shows a sample setup table.



```
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;%%%    TASK define section                                          %%%
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;---------- Usage ----------------------------------------------------------------------------------------
;                       TASK_TOP_LABEL                ;: COMMENT
;---------------------------------------------------------------------------------------------------------
            .import _TASKA                            ;: TASK.C
            .import _TASKB                            ;: TASK.C
;
```

Declares the start addresses of the tasks to be used as external reference symbols. (Add or modify these as necessary.)

Figure 1.6 Sample Setup Table (2655asup.src for H8S/2655) (1/2)

(Continued on next page)

(Continued from previous page)

Answer

```
;------ Usage ----------------------------------------------------------------------------------------
;                       .res.b  SIZE + TSKSTKSIZ ;  [RANGE] ;:   COMMENT
;TSK?_SP_LABEL:  .equ    $                                   ;:   COMMENT
;----------------------------------------------------------------------------------------------------
TSKSTKSIZ:          .equ     50+(10*2)+(6*1)+6+8; [50...] ;:  Task minimum stack size
            .section            h2sstack, stack, align = 2
                    .res.b   (36) + TSKSTKSIZ ; [50...]     ;:  tskid1 stack area
TSK1_SP:            .equ     $
                    .res.b   8
                    .res.b   (36) + TSKSTKSIZ ; [50...]     ;:  tskid2 stack area
TSK2_SP:            .equ     $
                    .res.b   8
                    .res.b   (32) + TSKSTKSIZ ; [50...]
TSK3_SP:            .equ     $
                    .res.b   8
                    .res.b   (32) + TSKSTKSIZ ; [50...]
TSK4_SP:            .equ     $
                    .res.b   8
;
            .section            h2ssetup, code, align = 2
_HI_H8S:            .res.b   10                                 ;: System Area
;------ Usage ----------------------------------------------------------------------------------------
;LABEL               .data.b IMOD, ITSKPRI
;                    .data.l  ITSKADR, ITSKSP
;----------------------------------------------------------------
NOEXS:              .assign 0
RDY:                .assign 1
DMT:                .assign (-1)
TDTLEN:             .assign 10; <- Not Change !
            .section            h2ssetup, code, align
_HI_TDT:            .equ     $-TDTLEN
TDT_TOP:            .equ     $
tdt_id1:            .data.b  DMT, 1
                    .data.l  _TASKA, TSK1_SP
tdt_id2:            .data.b  DMT, 2
                    .data.l  _TASKB, TSK2_SP
tdt_id3:            .data.b  NOEXS, 3
                    .data.l  0, TSK3_SP
tdt_id4:            .data.b  NOEXS, 4
                    .data.l  0, TSK4_SP
tdt_id5:            .data.b  NOEXS, 5
                    .data.l  0, TSK4_SP
TDT_BTM:
TSKCNT:             .equ     (TDT_BTM-TDT_TOP
                                        ;:[0...255]
;
```

<Task stack area definition>
Line 1: Defines the stack size to be used.
Line 2: Defines the stack label (task stack bottom)
Line 3: Defines the shared stack management area. (this can be omitted when the shared stack is not used)

<Task definition>
Format:

```
LABEL :  .data.b  IMOD, ITSKPRI
         .data.l   ITSKADR, ITSKSP
```

(1) LABEL
A label can be specified (this can be omitted).
(2) IMOD (task initial state)
Defines a task and initial state after initiation.
a) NOEXS (= 0): Not registered
b) RDY (= 1): Task is ready after initiation
c) DMT (= -1): Task is in dormant state after initiation.
(3) ITSKPRI (initial priority of task)
Defines the initial priority of the task.
(4) ITSKADR (initial start address of task)
Defines the start address of the task.
* The start address defined by the external reference symbol must be specified here.
(5) ITSKSP (initial stack pointer of task)
Defines the stack pointer (bottom address) when the task is initiated.
* The stack label defined in the task stack area definition section must be specified here.

Figure 1.6 Sample Setup Table (2655asup.src for H8S/2655) (2/2)

(2) Static Definition by Configurator

Classification: Object					
Question	HI7000/4	HI7700/4	HI7750/4		

The configurator has views (setting items) for defining (creating) tasks or event flags.

Should these items be defined (specified) only when objects are statically created?

Should they not be defined (specified) when objects are dynamically created (through cre_tsk, etc.) in the code?

Answer

Definition in each object creation view is not always necessary.

It is not necessary when objects are dynamically created in the code (program). When an object is created by defining it in the creation view for that object, it does not need to be created in the code (program), and the object can be used immediately after the system is started.

Note that the maximum object ID (maximum number of objects to be used) must always be defined for each object type (such as the task or event flag) through the configurator. If these definitions are omitted, objects may not be created dynamically in the code (program) in some cases.

1.3 Service Call Parameter Check

In the HI series OS, the context or parameters to be checked when a service call is issued are classified into the following two types.

- Dynamic parameters: Parameters which change dynamically during system operation
  - — Whether objects such as tasks or semaphores are used,
  - — Context when a service call is issued,
  - — Status of the target task, etc.
- Static parameters
  - — Maximum value for the specified ID, etc.

Table 1.6 shows differences in operation depending on whether the parameter check function is enabled.

Table 1.6 Differences Depending on the Parameter Check Function

Parameter Check Function	Check Targets	Advantages	Disadvantages
Not installed (parameter check function is disabled)	Dynamic parameters	• Fast processing • Small program size	If the system goes out of control because of an error in a service call parameter, it is difficult to determine the error.
Installed (parameter check function is enabled)	• Static parameters • Dynamic parameters	Easy debugging	• Slow processing • Large program size

For installation (enabling) of the parameter check function in the HI series OS, refer to the appropriate section for each OS in this application note.

1.3.1 Installation in HI7000/4 Series

In the HI7000/4 series, the parameter check function is installed by settings in the Kernel Extension Function view of the configurator.

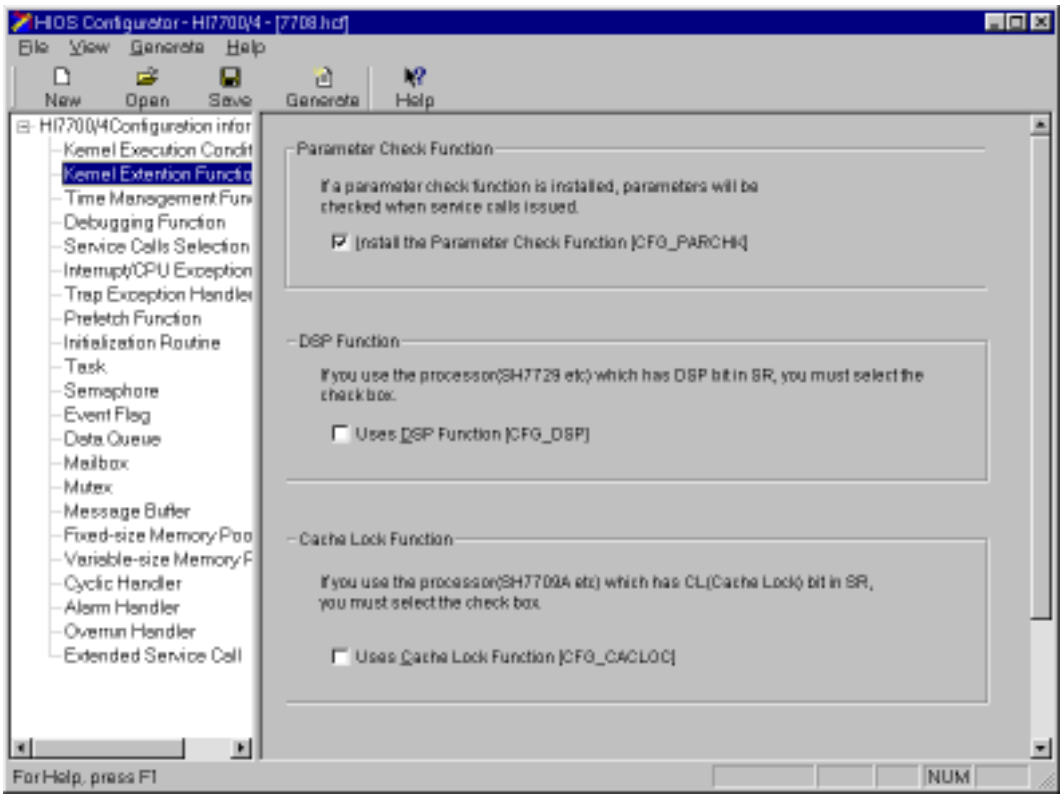


Figure 1.7 Kernel Extension Function View

Select the [Install the Parameter Check Function [CFG_PARCHK](C)] check box for [Parameter Check Function] in the kernel extended function view to install the parameter check function (this check box is selected at default in the configurator).

1.3.2 Installation in HI2000/3 and HI1000/4

In the HI2000/3 and HI1000/4, the parameter check function is installed by selecting the library files including the parameter check function when the system is configured.

Specify the library files including the parameter check function in the HEW project file during configuration to install the parameter check function. (The library files with this function are selected by default in the HEW project file provided as standard.)

The following describes an example of a library file definition procedure when using the H8S, H8/300 Series C/C++ Compiler Package V6.0.00 in the H8S/2655 advanced mode of HI2000/3 V1.10r1.

In the active HEW workspace, select [H8S, H8/300 Standard Toolchain...] from [Options] in the header menu.

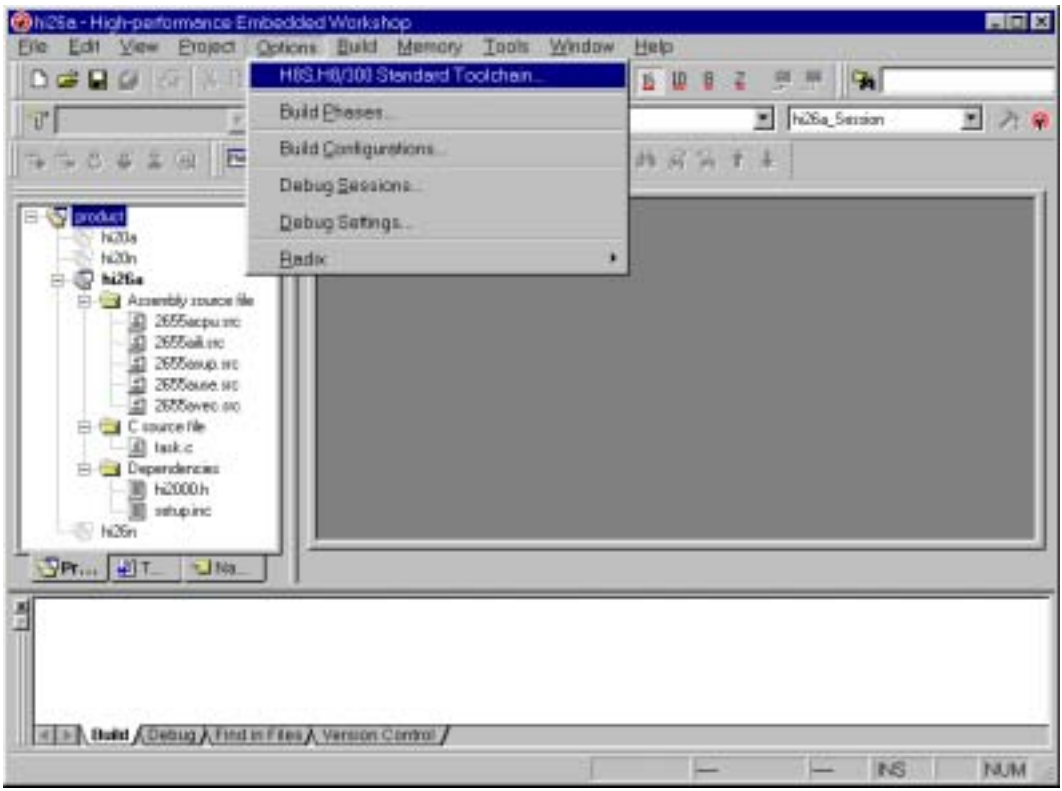


Figure 1.8 Library File Definition (1)

Select the [Link/Library] tag in the [H8S, H8/300 Standard Toolchain] dialog box to see the current settings. Figure 1.9 shows the displayed current settings.

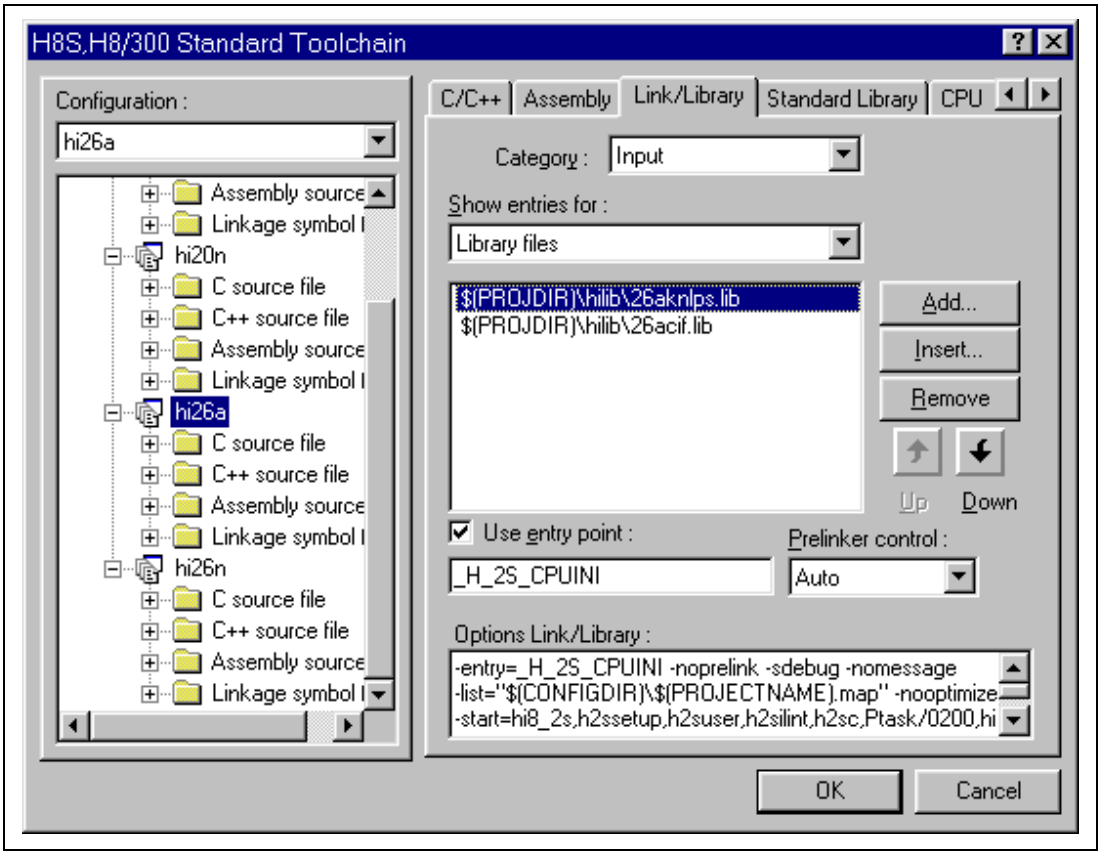


Figure 1.9 Library File Definition (2)

The default kernel library file includes the parameter check function. This example shows the procedure for switching from the default library file to a library file that does not include the parameter check function.

Select the current library file and click the [Insert...] button.

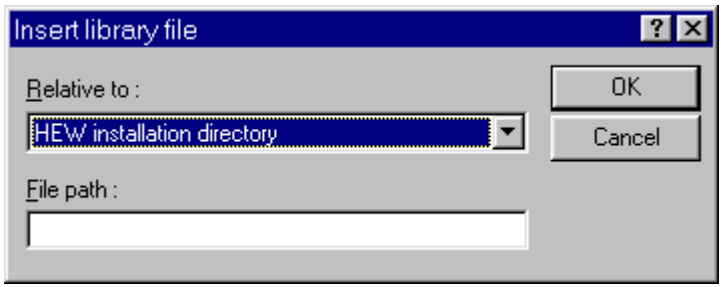


Figure 1.10 Library File Definition (3)

Specify [Relative to:] and [File path:] and click the [OK] button.

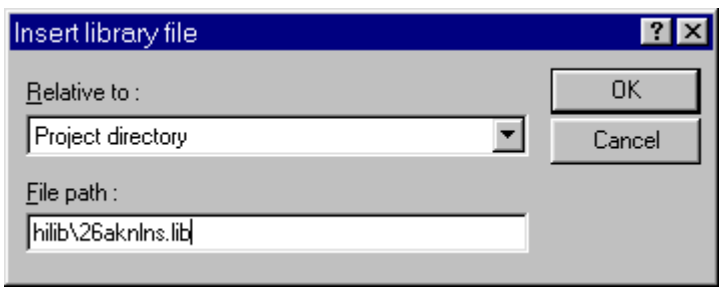


Figure 1.11 Library File Definition (4)

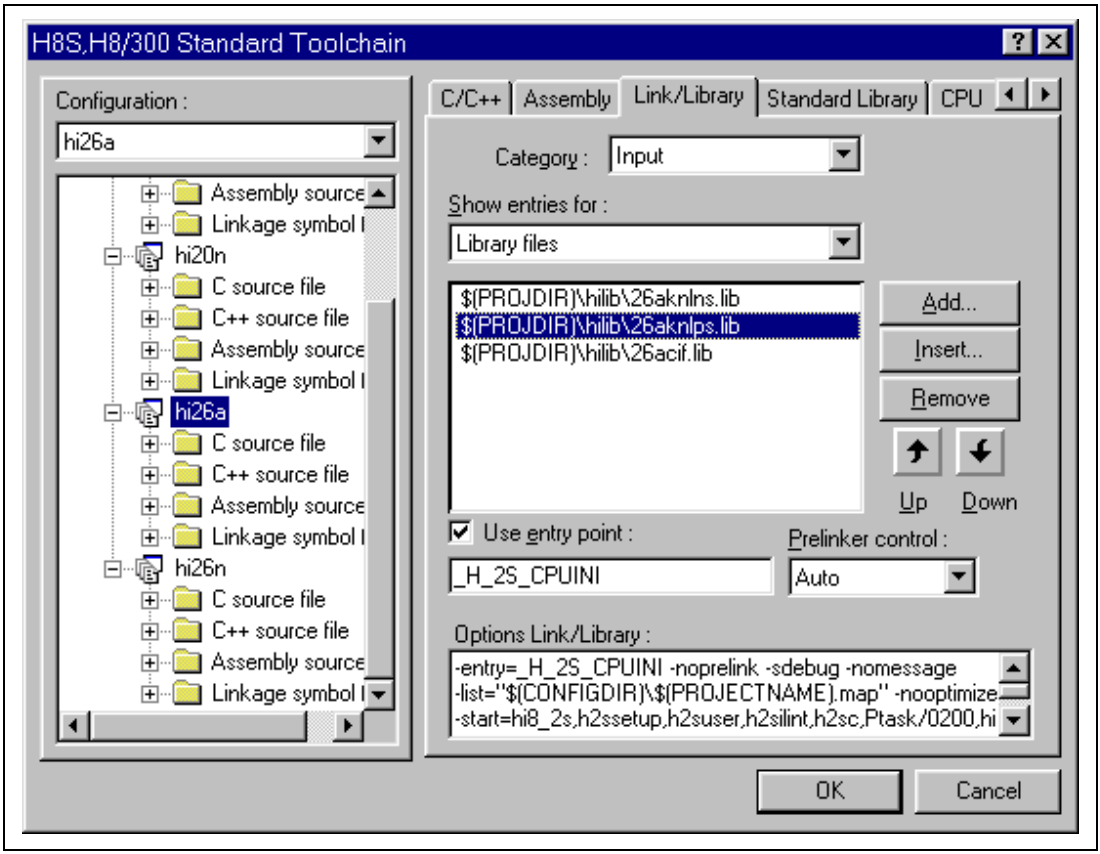


Figure 1.12 Library File Definition (5)

Select the default library file in the [H8S, H8/300 Standard Toolchain] dialog box, and click the [Remove] button.

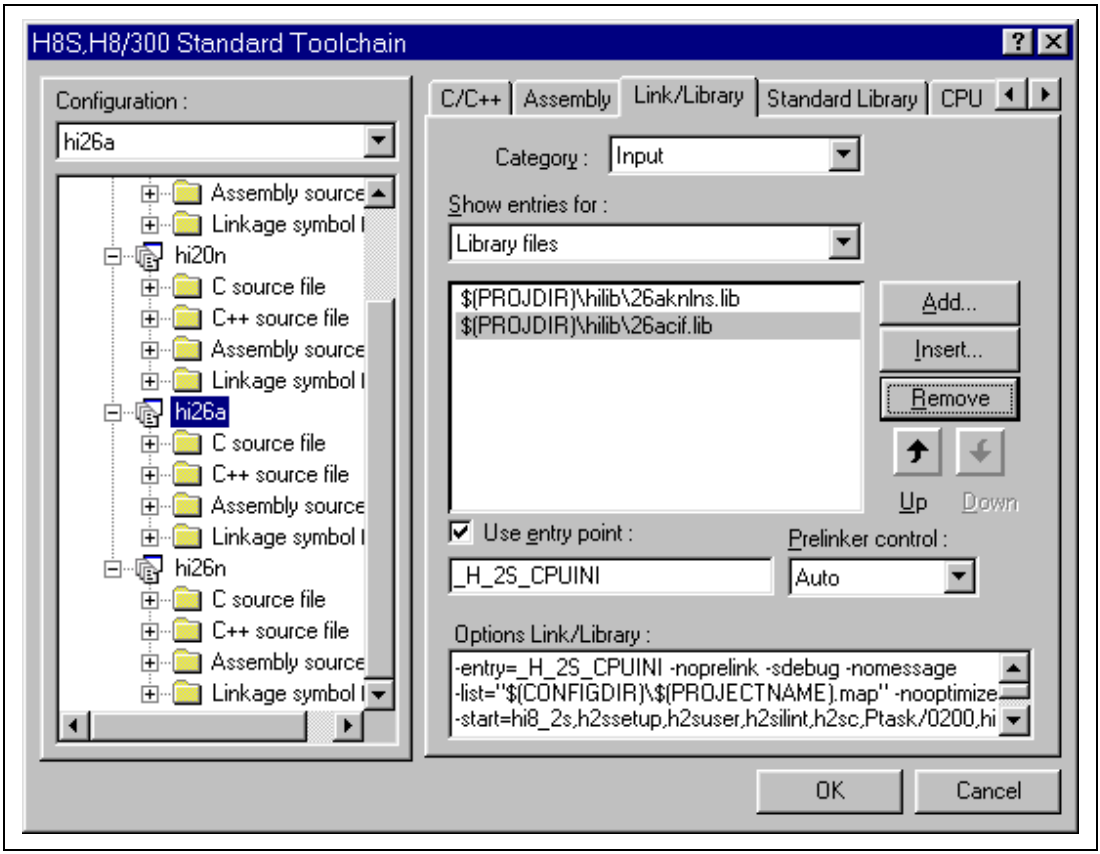


Figure 1.13 Library File Definition (6)

Click the [OK] button in the [H8S, H8/300 Standard Toolchain] dialog box to reflect the new settings in the HEW workspace. This completes switching of the kernel library.

1.3.3 FAQ about Service Call Parameter Check

This section answers a question about service call parameter check which is frequently asked by users of the HI series OS.

FAQ Contents:

(1) Parameter Check Enabled/Disabled

Classification: Service call parameter check					
Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4

The functional libraries provided by the OS are classified into those with the parameter check function and those without the function.

What purpose should they be used for?

Answer

The libraries with the parameter check function are provided by the OS to be used for debugging. They check errors in coding (parameter correctness) of the user-created application programs.

When the libraries with the parameter check function are used, the overhead of the parameter check function increases the processing time and the amount of processing code in comparison with usual service calls.

After the debugging step is completed, we recommend that the libraries without the parameter check function be used to generate load modules to be included in the final product.

Note: Dynamic parameters are always checked even when the parameter check function is not installed.

1.4 Tasks

1.4.1 Tasks and Functions

Table 1.7 shows the differences between tasks and functions.

Table 1.7 Differences between Tasks and Functions

Item	Task	Function
Program description	No difference in program description	
	A task may be configured with one function (or a group of functions) in some cases.	
Initiation	The OS determines the task to initiate according to the priority and specified initiation order.	The main function initiates each function.
Management	OS	Function
Interface	OS functionality (such as service calls)	Parameters
Dependency and coupling	Tasks are loosely coupled and independent	Functions are tightly coupled and dependent on each other

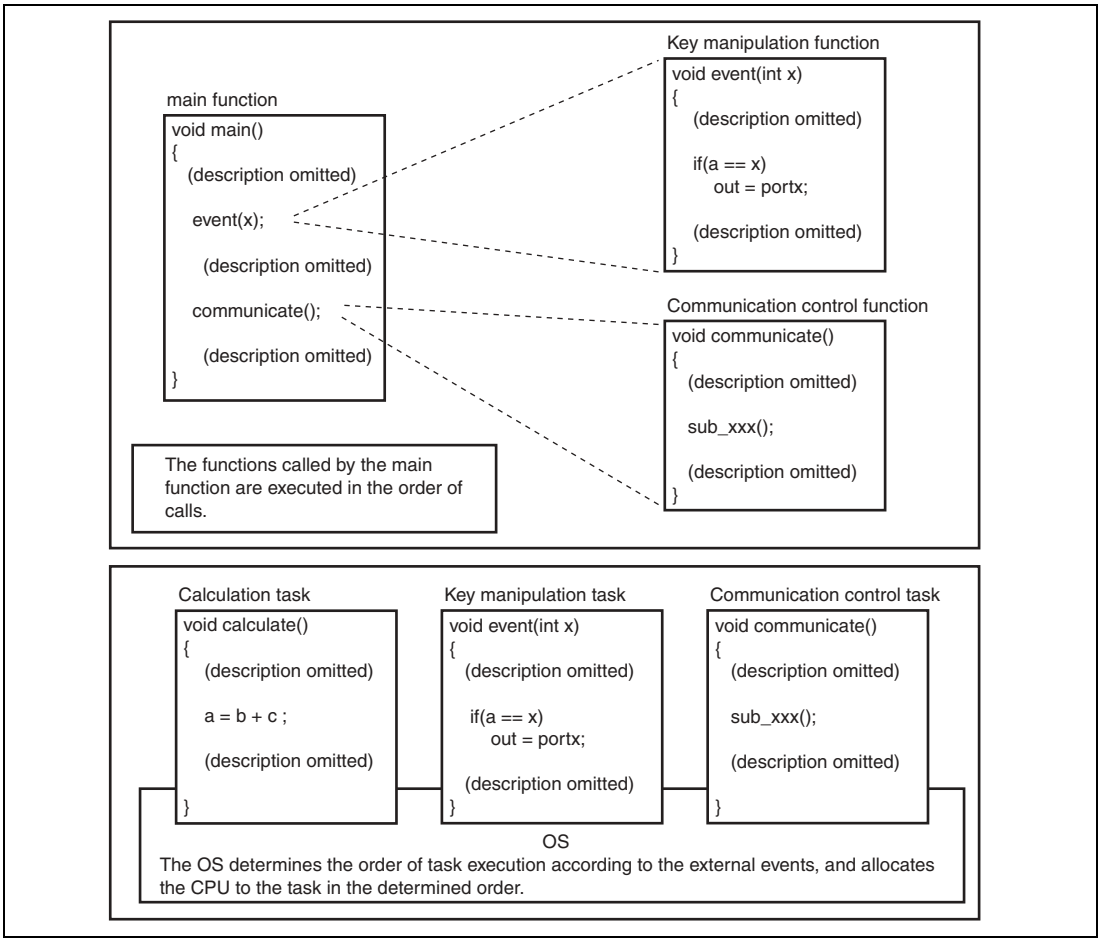


Figure 1.14 Differences between Tasks and Functions

1.4.2 Task Initiation

Figure 1.15 shows the procedure to initiate a task.

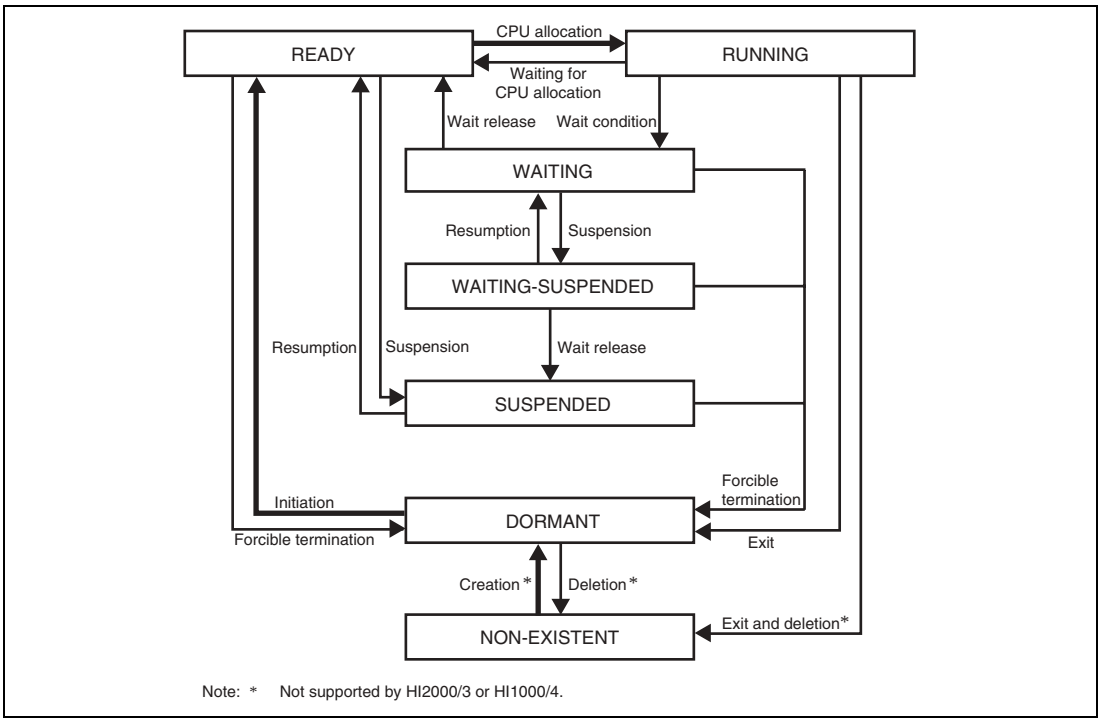


Figure 1.15 Task State Transitions

Task State	Description
NON-EXISTENT (does not exist in the system)	The task has not been registered in the kernel.
↓ cre_tsk()*, acre_tsk()*, etc.	
DORMANT (inactive)	The task has been registered in the kernel, but has not yet been initiated.
↓ sta_tsk(), act_tsk(), etc.	
READY (executable)	The task is ready to be executed and is waiting for CPU resource allocation.
RUNNING (executing)	The CPU is allocated to the task and the task is being executed.

Note: * Not supported by the HI2000/3 or HI1000/4.

1.4.3 Task Stacks

Table 1.8 shows the stacks used by tasks.

Table 1.8 Task Stacks Available in HI Series OS

Stack	HI7000/4 Series	HI2000/3	HI1000/4
Dynamic stack	Available	Not available	Not available
Static stack	Available	Available	Available
(Shared stack)	(Available)	(Available)	(Available)

For details on the stack allocation and shared stack function, refer to the user's manual of the HI series OS used.

(1) Types of Task Stack

Table 1.9 shows the types of task stack.

Table 1.9 Types of Task Stack

Stack Type	Description
Dynamic stack	This type of stack area is allocated in the space managed by the OS for the required size, and a stack is actually assigned for a task when the task is initiated.
Static stack	This type of stack area is allocated for each task, and a stack is actually assigned for a task when the task is initiated.

(2) Shared Stack Function

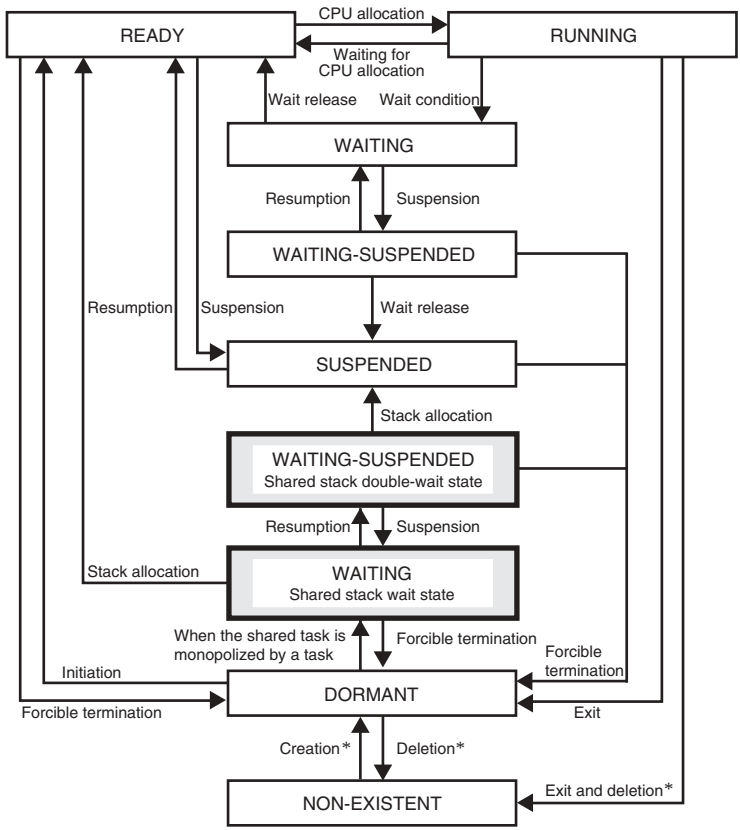
Multiple tasks that use static stacks can share one stack area. This shared stack function reduces the task stack size.

Table 1.10 shows the required memory used by dynamic stacks, static stacks, and shared stacks.

Table 1.10 Stack Types and Required Memory

Stack Type	Required Memory
Dynamic stack	The total size (Σ) of all task stacks does not need to be allocated.
Static stack (shared stack function)	• The total size (Σ) of all task stacks must be allocated. • When the shared stack function is used, multiple tasks can share one task stack, which reduces the required memory size.

Figure 1.16 shows the task state transitions for the shared stack function.



Note: * Not supported by the HI2000/3 or HI1000/4.

Figure 1.16 Task State Transitions for Shared Stack Function

Note: Tasks that use dynamic stacks cannot use the shared stack function.

1.4.4 CPU Allocation to Tasks

The CPU resource is allocated to tasks according to the priority levels defined for tasks. For the task priority, a smaller value indicates a higher priority level, and a larger value indicates a lower priority level.

The priority among tasks is determined by the priority level of each task. This section describes task priority control as illustrated in the accompanying figures.

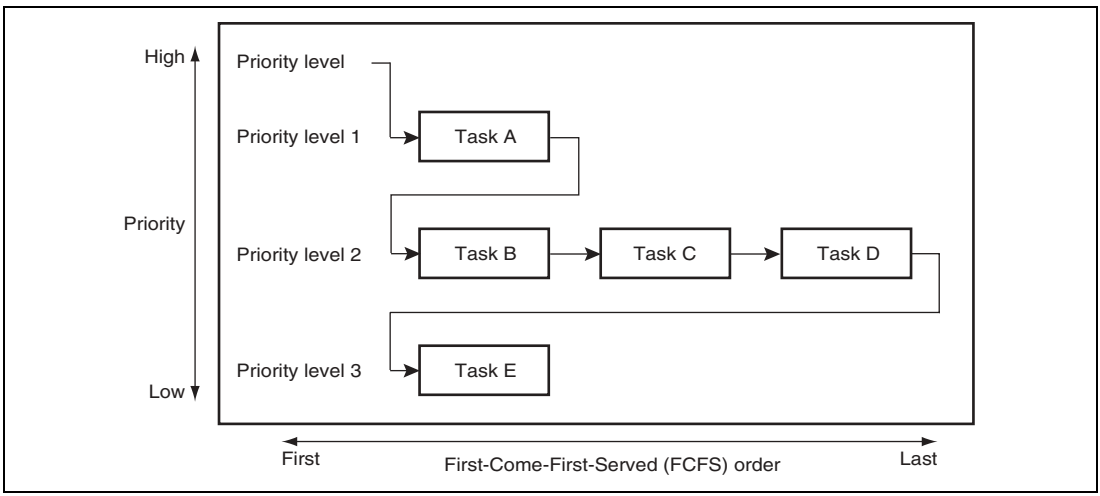


Figure 1.17 Task Priority (1)

Figure 1.18 shows the priority after task A releases the right of execution by issuing a task terminating or deleting service call or by entering the event wait state because of a service call.

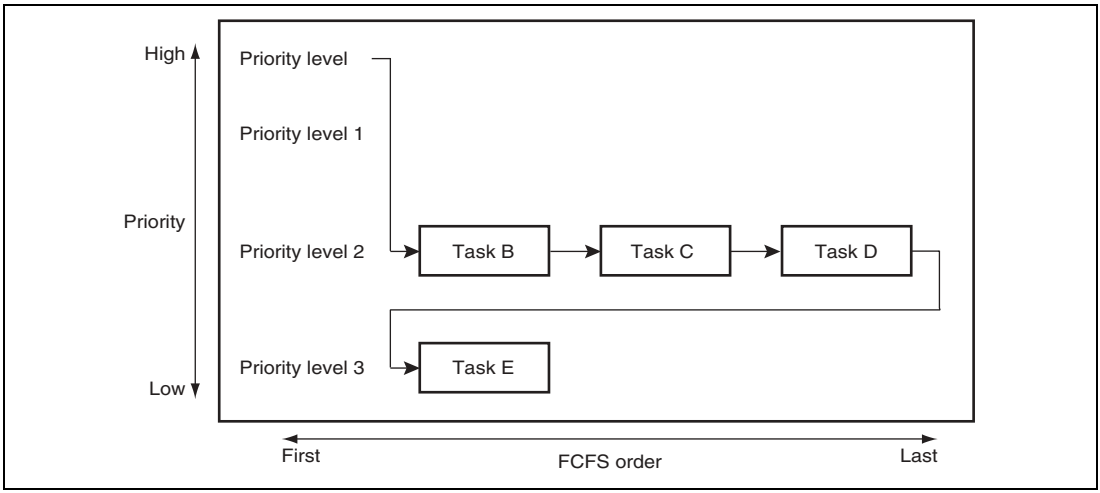


Figure 1.18 Task Priority (2)

After task B releases the right of execution by entering the event wait state because of a service call, task C enters the READY state. Figure 1.19 shows the priority after task B exits from the WAITING state.

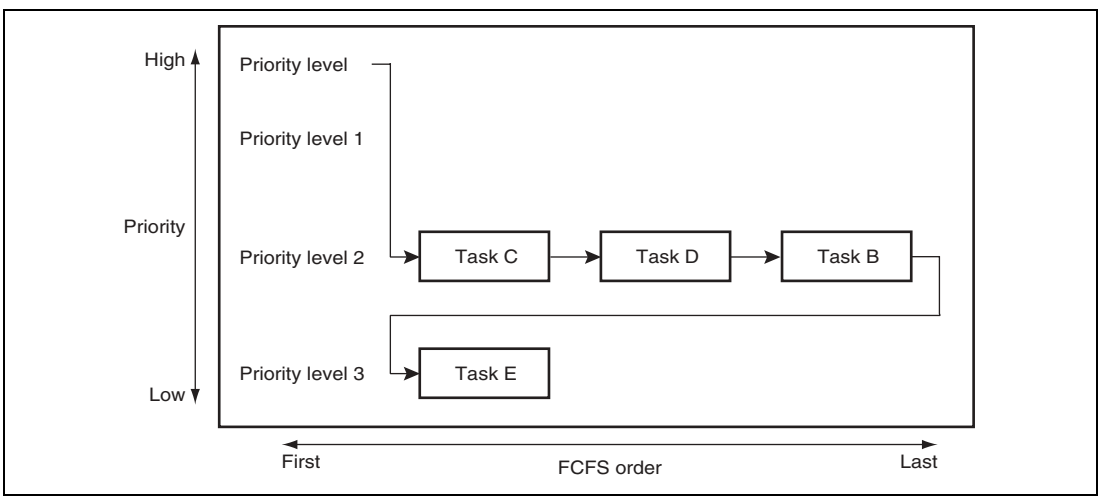


Figure 1.19 Task Priority (3)

Note: A task is scheduled to be executed last for the same priority level (placed at the end of the queue for the same priority level) on a First-Come-First-Served (FCFS) basis.

If a higher-priority task becomes ready while a lower-priority task is being executed, the lower-priority task execution is suspended (moves from the RUNNING state to the READY state) and the higher-priority task is executed first.

In the μITRON specifications, suspending a lower-priority task in favor of a higher-priority task is called preempting.

The following describes the priority change when a task issues a service call to other tasks with different priority levels or when a service call is issued to the current task.

(1) Service Call to Other Tasks

The following describes task execution control when a task issues service calls.

The initial state before a service call is issued is assumed to be as follows.

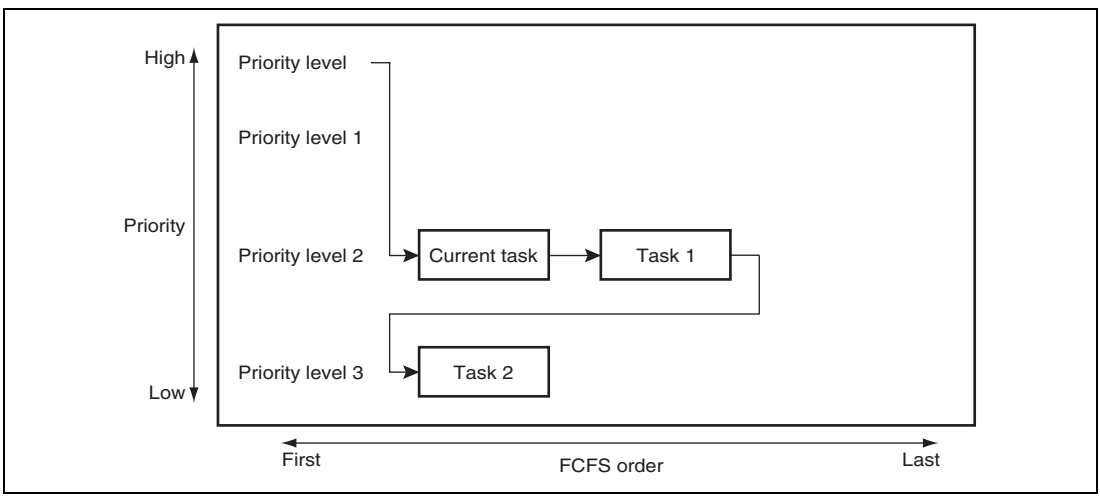


Figure 1.20 Priority Before a Service Call Is Issued to Other Tasks

The task which issues a service call is called the current task. Figure 1.21 shows the state after the current task issues an initiating service call (sta_tsk or act_tsk) to task A of the same priority level as the current task and to task B of a lower priority level than the current task, and tasks A and B enter the READY state.

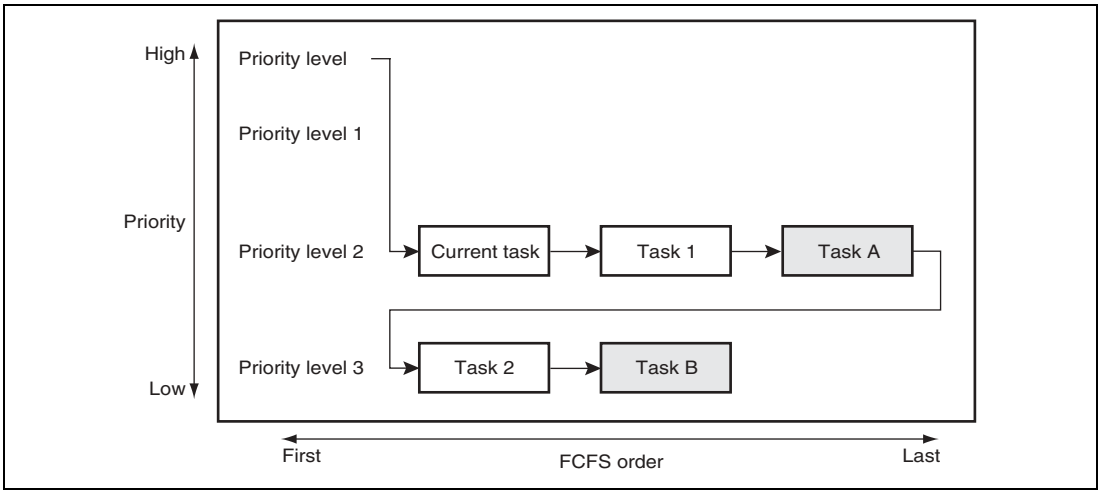


Figure 1.21 Priority After a Service Call Is Issued to Other Tasks (1)

Figure 1.22 shows the state after the current task issues an initiating service call to task C of higher priority than the current task and task C enters the READY state.

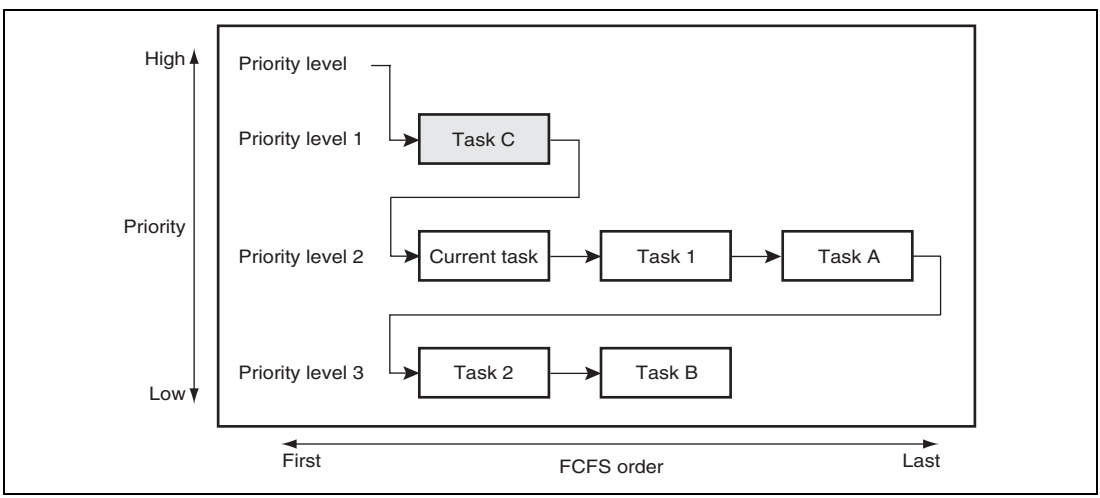


Figure 1.22 Priority After a Service Call Is Issued to Other Tasks (2)

As shown, the priority of the current task is changed by a service call to task C of higher priority than the current task, and the current task is immediately preempted when the service call is issued.

(2) Service Call to Current Task

The following describes the priority control when a service call is issued to the current task.

The initial state before a service call is issued is assumed to be as follows.

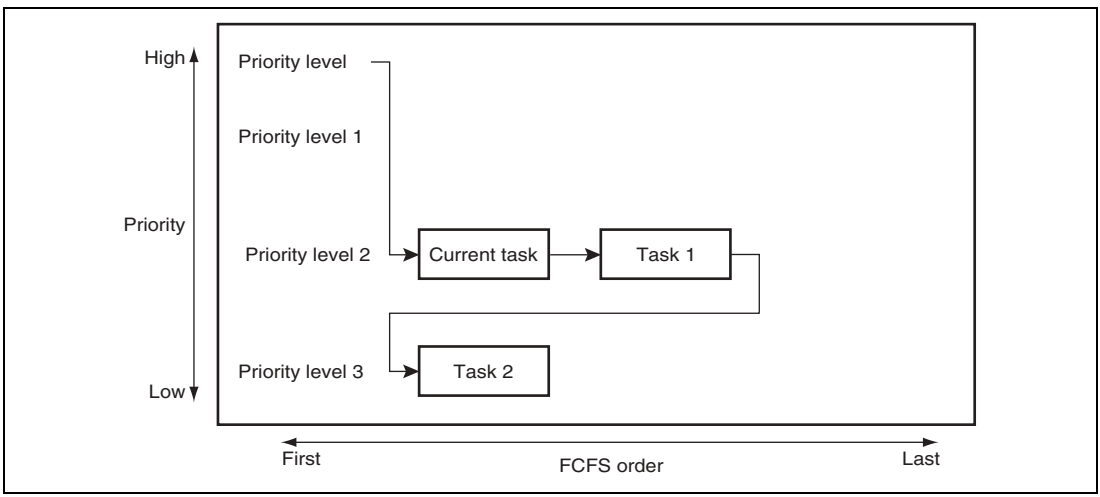


Figure 1.23 Priority Before a Service Call Is Issued to Current Task

Figure 1.24 shows the state after the priority level of the current task is modified to be higher (modified from priority level 2 to 1).

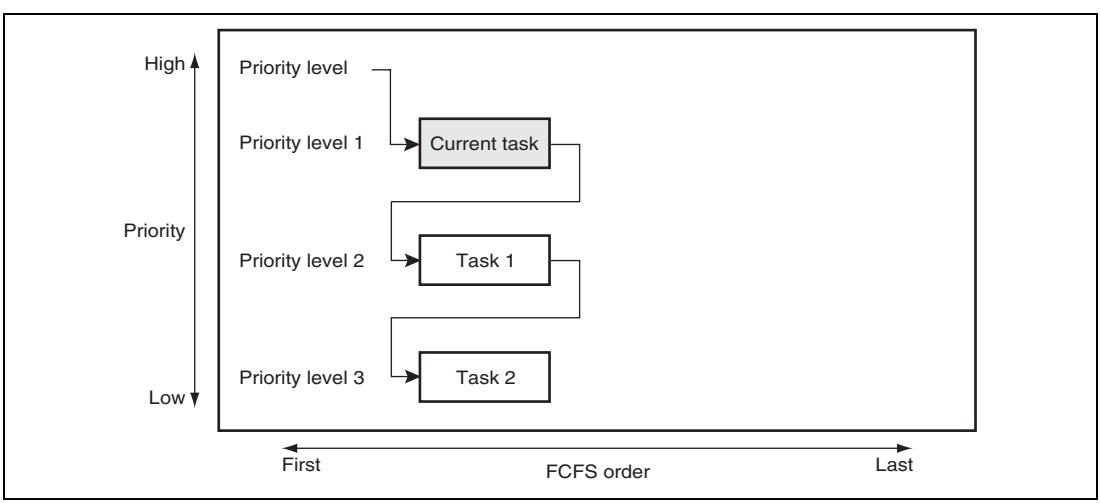


Figure 1.24 Priority After a Service Call Is Issued to Current Task (1)

Figure 1.25 shows the state after the priority level of the current task is modified back to its original level (modified from priority level 1 to 2).

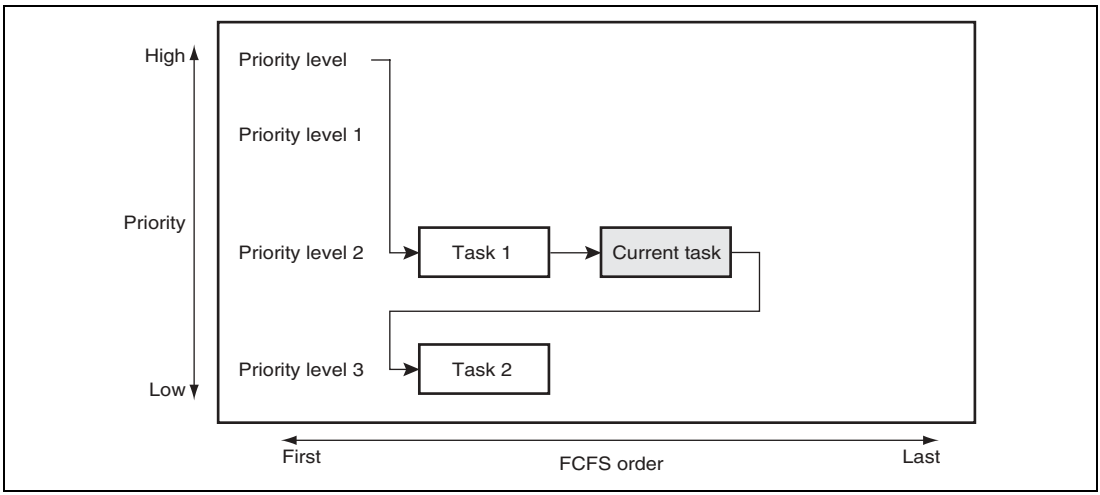


Figure 1.25 Priority After a Service Call Is Issued to Current Task (2)

In this case, the processing after the priority change depends on whether there is a task of the same priority level as the current task as shown below.

Table 1.11 Differences in Processing after Priority Change

Same-Priority Task	Current Task Processing	Current Task Execution
When a same-priority task exists	The current task is placed at the end of the same-priority queue according to the scheduling rule (FCFS basis).	Preempted
When no same-priority task exists	The current task is placed at the beginning of the same-priority queue.	Execution is continued

1.4.5 Polling

Service calls for waiting for events on objects are classified into three types: general wait, wait with timeout, and wait with polling. This section describes the differences in processing of these service calls (using an event flag in this example).

Figure 1.26 gives an overview of wai_flg service call processing as an example of general event wait service calls.

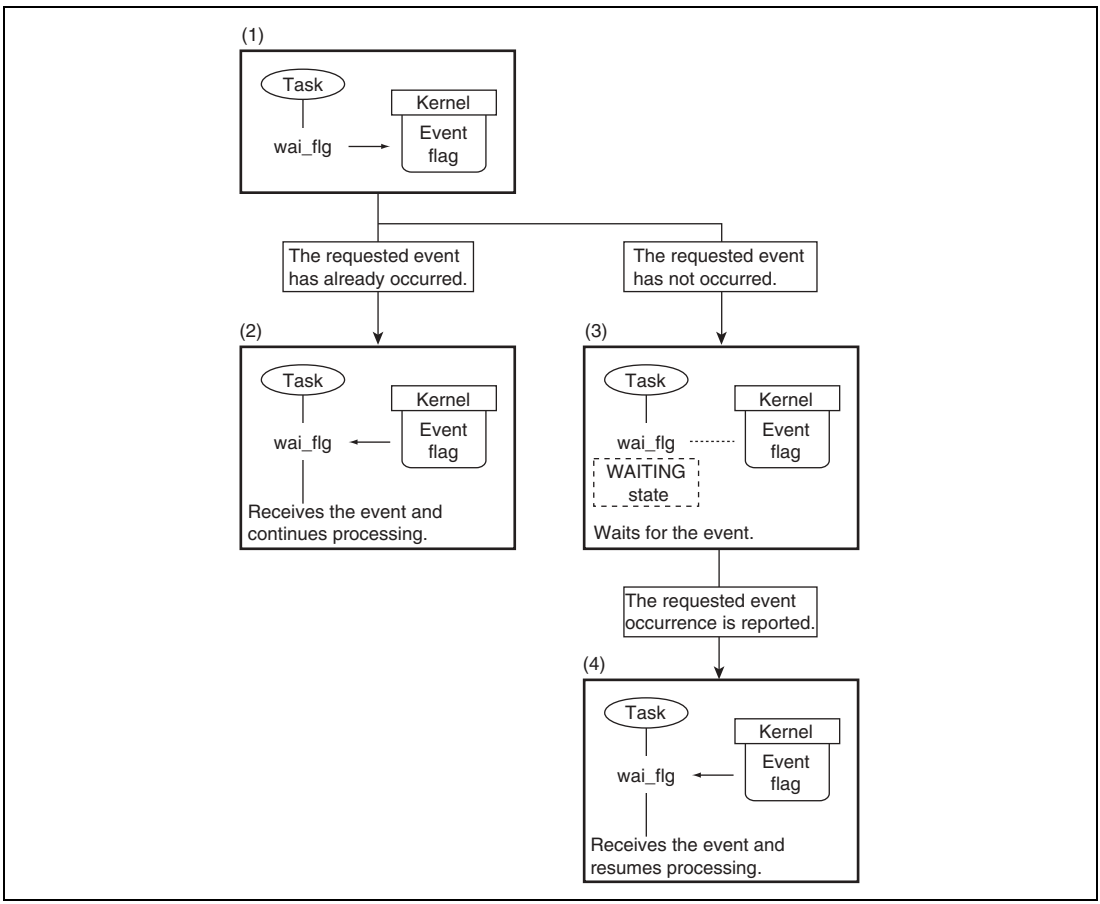


Figure 1.26 Overview of General Event Wait Service Call Processing

(1) The task issues a wai_flg service call for an event flag.

(2) When the specified event has already occurred, the return code shows normal termination (E_OK) and the task processing continues.

(3) When the specified event has not occurred, the task processing is suspended and the task enters the WAITING state until the event occurrence is reported.

(4) When the specified event is reported by a set_flg service call from a task or an interrupt handler, the return code shows normal termination (E_OK) and the task processing resumes.

Figure 1.27 gives an overview of twai_flg service call processing as an example of an event wait service call with timeout.

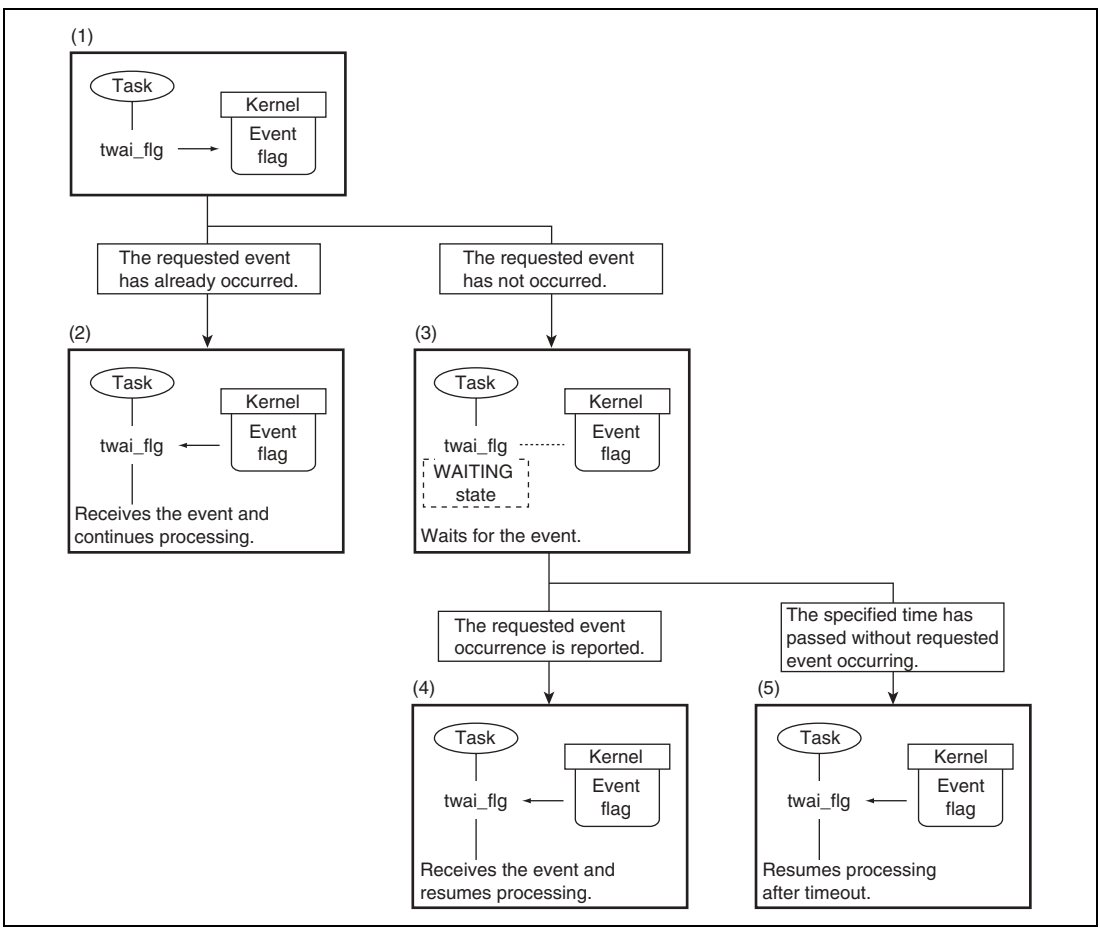


Figure 1.27 Overview of Event Wait Service Call Processing with Timeout

(1) The task issues a twai_flg service call for an event flag.

(2) When the specified event has already occurred, the return code shows normal termination (E_OK) and the task processing continues.

(3) When the specified event has not occurred, the task processing is suspended and the task enters the WAITING state for the specified time until the event occurrence is reported.

(4) When the specified event is reported by a set_flg service call from a task or an interrupt handler, the return code shows normal termination (E_OK) and the task processing resumes.

(5) When the specified event is not reported within the specified time, the return code shows time out (E_TMOUT) and the task processing resumes.

Figure 1.28 gives an overview of pol_flg service call processing as an example of an event wait service call with polling.

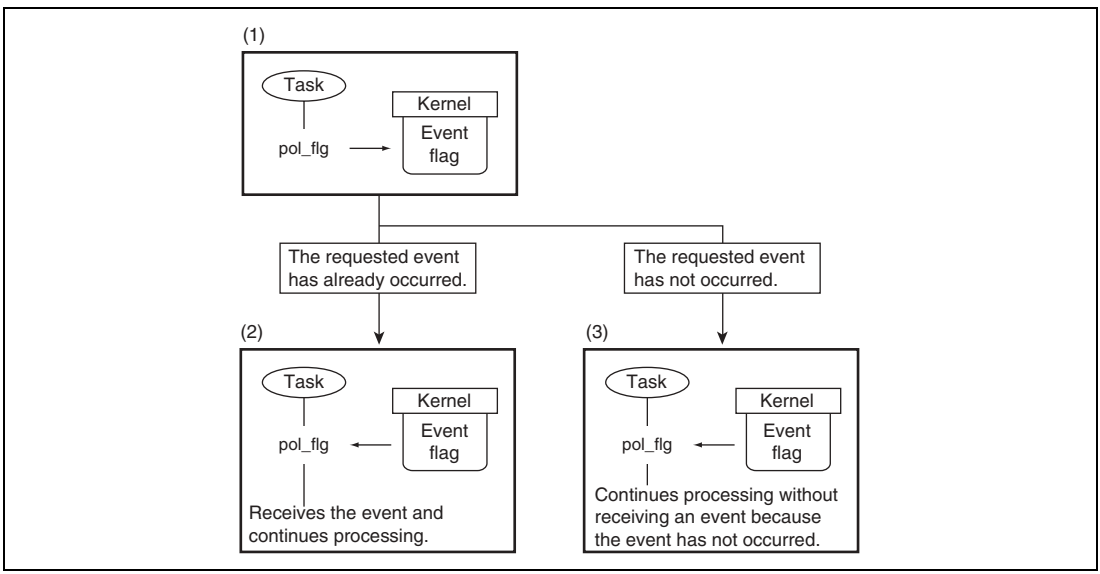


Figure 1.28 Overview of Event Wait Service Call Processing with Polling

(1) The task issues a pol_flg service call for an event flag.

(2) When the specified event has already occurred, the return code shows normal termination (E_OK) and the task processing continues.

(3) When the specified event has not occurred, the return code shows polling failed (E_TMOUT) and the task processing continues.

Table 1.12 shows the differences among general event wait, wait with timeout, and wait with polling.

Table 1.12 Differences Among General Event Wait, Wait With Timeout, and Wait With Polling

Wait Service Call	WAITING State	Wait Time
General wait	Entered	Not specified
Wait with timeout	Entered	Specified
Wait with polling	Not entered	Not specified

1.4.6 FAQs about Tasks

This section answers questions about tasks which are frequently asked by users of the HI series OS.

FAQ Contents:

(1) Initialization and Task Initiation

Classification: Task and task initiation					
Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4
Please explain in detail the relationship between the main() function and tasks.					

Answer

The µITRON specifications have no concept of the main() function (the system start function). The system using the µITRON specifications determines which task to initiate according to the task priority defined in the system and the order of task initiation requests.

System initialization or task initiation can be specified in the main() function. In this case, the main() function must be defined as the initial start task or the system initialization routine. Each task is initiated by a service call issued in the main() function.

Refer also to section 1.4.1, Tasks and Functions in this application note.

(2) Defining and Initiating Tasks in a Configuration File

Classification: Task and task initiation					
Question	HI7000/4	HI7700/4	HI7750/4		HI1000/4
What should be done to execute Main_Task() after defining it in the create and initiate mode in the task list through the configurator?					
Answer No other definition related to task creation and initiation is necessary through the configurator. Specify [Start Task after Creation (TA_ACT)] for the attribute of the initial start task in the window displayed after "Create" (or "Modify") is selected in the task list in the task view of the configurator. The kernel initialization processing makes the task enter the READY state.					

(Continued on next page)

(Continued from previous page)

Answer

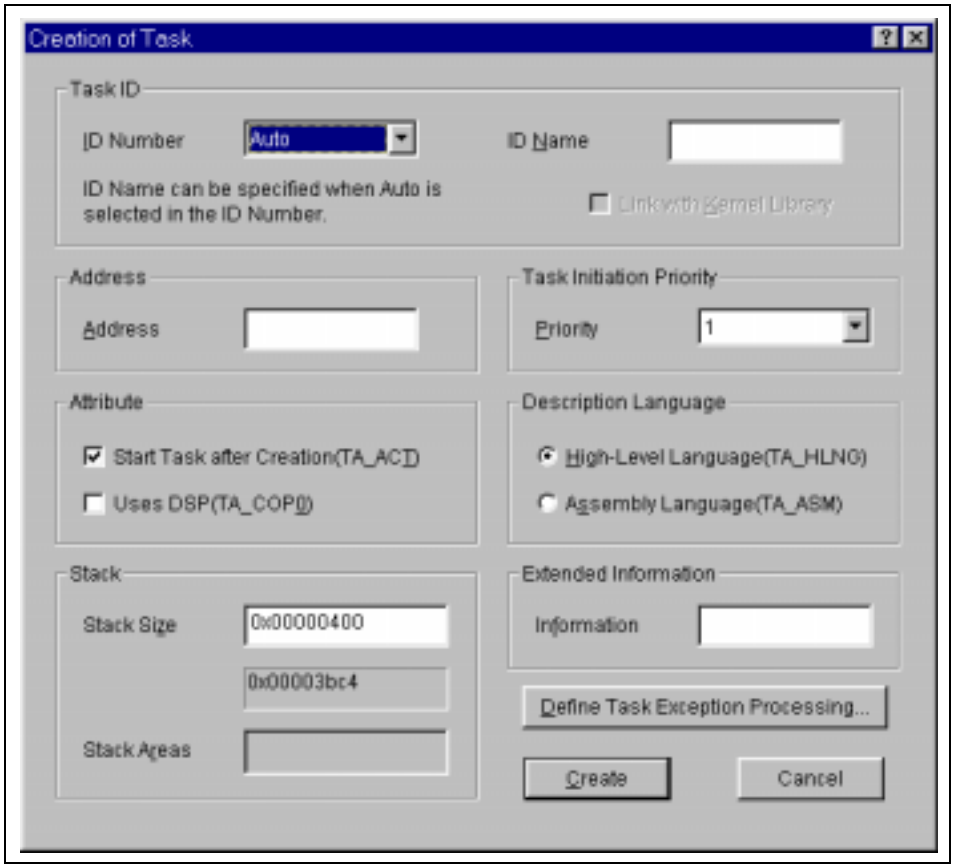


Figure 1.29 Task Creation Window

(3) Initiating Tasks

Classification: Task and task initiation

Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4

If all tasks are set to the DORMANT state, how should they be initiated?

Answer	HI7000/4	HI7700/4	HI7750/4		HI1000/4

When all tasks are defined in the DORMANT state, they can be initiated as follows:

1. Define an initialization routine and initiate tasks through service calls.
2. Define an interrupt handler or a time event handler (cyclic handler or alarm handler) and initiate tasks through service calls.

We recommend that tasks be defined with [Start Task after Creation (TA_ACT)] specified for general usage.

Answer				HI2000/3	

When all tasks are defined in the DORMANT state, they can be initiated as follows:

1. Define a system initialization handler and initiate tasks through service calls.
2. Define an interrupt handler or a cyclic handler and initiate tasks through service calls.

We recommend that the task initial state should be defined as [READY state after initiation (RDY)] instead of [DORMANT state after initiation (DMT)] in the setup table for general usage.

(4) Stack for Initial Start Task

Classification: Task and task initiation

Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4

Which stack area does the task initiated immediately after initialization use?

Answer

The initial start task uses the task stack assigned at creation. The kernel assigns an actual stack to the task according to the task creation information.

The following section areas are used as the task stack areas.

- HI7000/4 series
  - Static stack: B_histstk
  - Dynamic stack: B_hidystk
- HI2000/3
  - Static stack: h2sstack (dynamic stack is not supported)
- HI1000/4
  - Static stack: B_histack (dynamic stack is not supported)

For the static stack, an area in the above section area is assigned during task initiation processing for the size defined at creation.

For the dynamic stack, the kernel allocates an area in the above section area for the specified size, and actual stack area is assigned for a task when the task is initiated.

The section areas for the stacks can be determined by the user. For the allocation of the stack section areas, refer to the following.

HI Series OS	Reference
HI7000/4 series	Section describing "Changing Linkage Address" in the appropriate configuration guide.
HI2000/3	Section 3.4.4 in this application note
HI1000/4	Section 3.4.5 in this application note

(5) Managing Tasks for the DSP Coprocessor

Classification: Task and task initiation					
Question	HI7000/4	HI7700/4			
What should be kept in mind when using the DSP unit in the HI series OS?					

Answer

When a task uses DSP functions, TA_COP0 should be specified for the task attribute parameter when the task is created. The task with TA_COP0 specified saves and restores the DSP registers in the same way as for the general registers.

To specify TA_COP0 for the task registered through the configurator, specify the [Uses DSP (TA_COP0)] check box under [Attribute] in the Creation of Task window.

(Continued on next page)

(Continued from previous page)

Answer

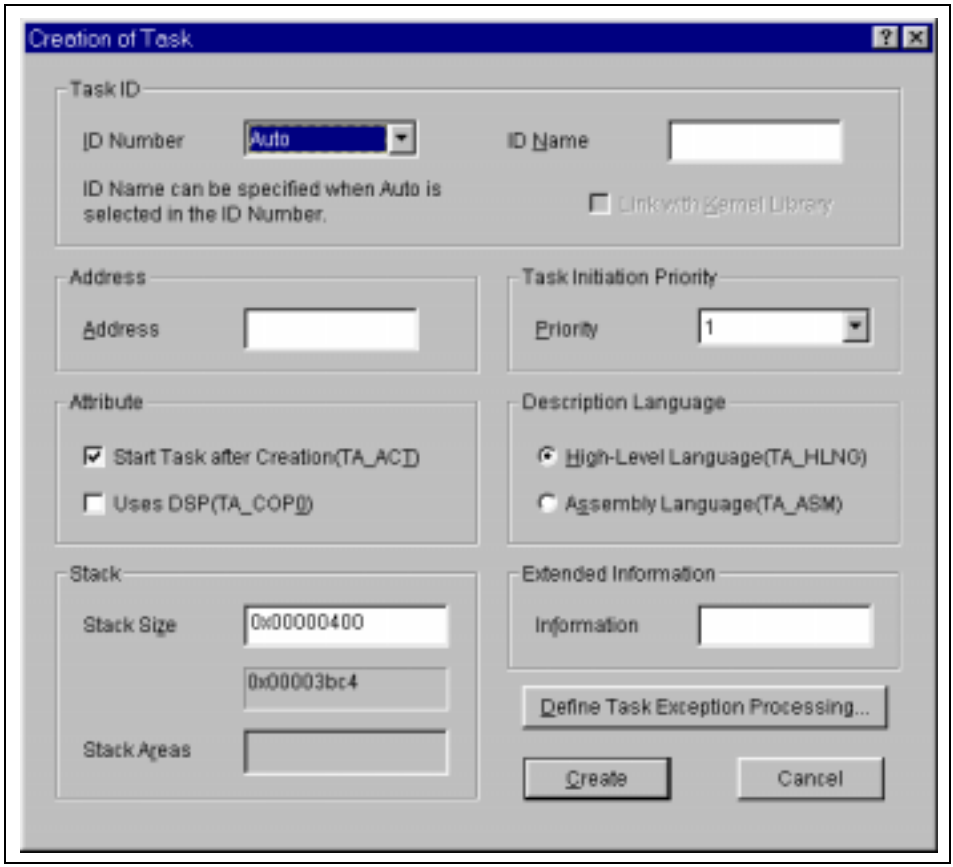


Figure 1.30 DSP Selection in Configurator

(Continued on next page)

(Continued from previous page)

Answer

To create a task by a service call during system operation, specify TA_COP0 as the task attribute parameter in the cre_tsk service call.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

#pragma noregsave(MainTask)

void MainTask(VP_INT stacd)
{
  ER        ercd;
  T_CTSK pk_ctsk;

    (processing description omitted)

    pk_ctsk.tskatr  = (TA_HLNG | TA_COP0)        /* Task attribute = high-level language description,
                                                    DSP coprocessor used                                */
    pk_ctsk.exinf   = 0;                         /* Extended information = 0                            */
    pk_ctsk.task    = (FP)task_A;                /* Task initiation address                             */
    pk_ctsk.itskpri = 1;                         /* Priority at task initiation                         */
    pk_ctsk.stksz   = 264;                       /* Task stack size                                     */
    pk_ctsk.stk     = (VP)sp_taskA;              /* Start address of task stack area                    */

    ercd=cre_tsk(TASK_A, &pk_ctsk);              /* Create task A                                       */
    ercd=sta_tsk(TASK_A, (VP_INT)0x00000001);    /* Initiate task A (initiation code = 0x1)             */

    (processing description omitted)

}
```

Figure 1.31 DSP Selection for Task Creation by Service Call (Sample Code)

For details on the task attribute parameter during task creation, refer to the HI7000/4 Series User's Manual.

When a non-task program (such as an interrupt handler or time event handler) uses DSP functions, each program must save and restore the DSP registers. For details, refer to the HI7000/4 Series User's Manual.

(6) Managing Tasks for the FPU Coprocessor

Classification: Task and task initiation

Question			HI7750/4		

What should be kept in mind when using the FPU functions in the HI series OS?

Answer

When using FPU functions, specify TA_COP1 or TA_COP2 for the task attribute parameter when creating a task. Table 1.13 shows each parameter meaning.

Table 1.13 TA_COP1 and TA_COP2 Meaning

Task Attribute	Meaning
TA_COP1	The task uses FPU register bank 0
TA_COP2	The task uses FPU register bank 1

The task with TA_COP1 or TA_COP2 specified saves and restores the FPU registers in the same way as for the general registers. Table 1.14 shows which of the TA_COP1 and TA_COP2 attributes should be specified.

Table 1.14 TA_COP1 and TA_COP2 Specifications

Case	Attribute Specification	Remarks
Matrix operation is necessary	[TA_COP1|TA_COP2]	Both FPU register banks are used
Floating-point operation is necessary	[TA_COP1]*	Only one FPU register bank is used for general floating-point operation
No floating-point operation is necessary	None	

Note: * TA_COP2 is not recommended because if it is specified, FR in FPSCR must be set to 1 in the beginning of the task and task exception processing routine.

The following describes how to specify the attribute.

(Continued on next page)

(Continued from previous page)

Answer

When the task registered in the configurator uses FPU bank 0, specify the [Uses FPU (Bank 0) (TA_COP1)] check box under [Attribute] in the Creation of Task window.

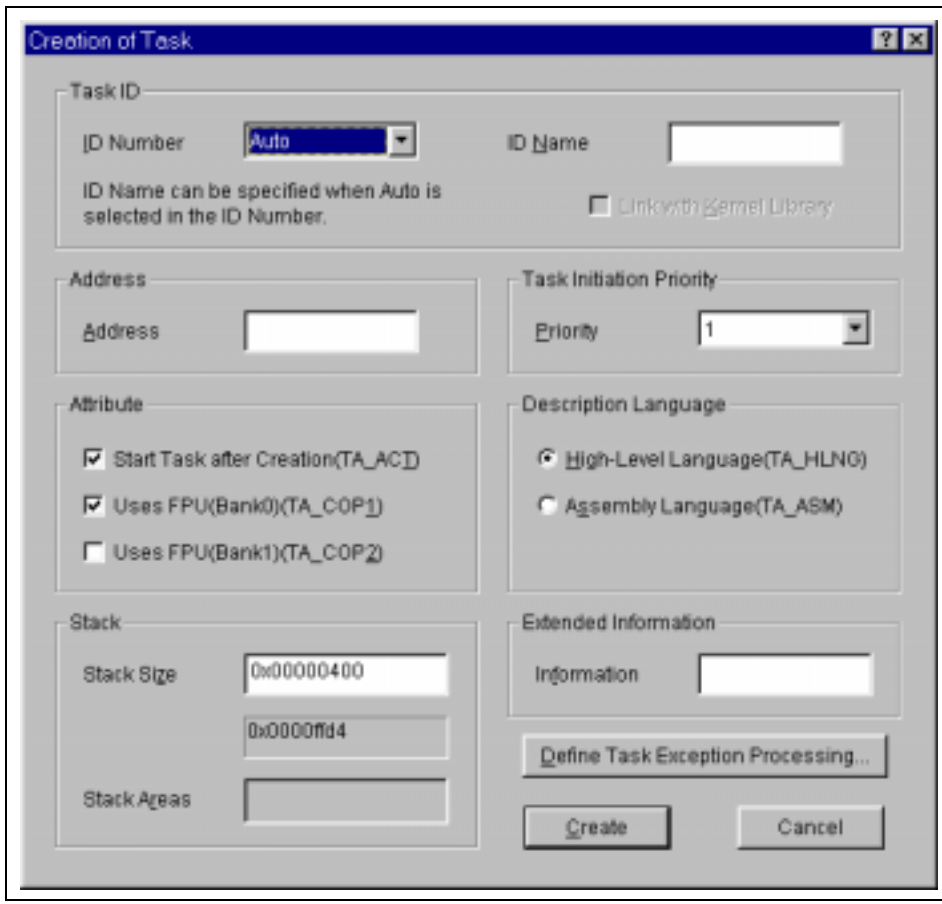


Figure 1.32 FPU Selection in Configurator (TA_COP1)

(Continued on next page)

(Continued from previous page)

Answer

When the task registered in the configurator uses FPU bank 1, specify the [Use FPU (Bank 1) (TA_COP2)] check box under [Attribute] in the Creation of Task window.

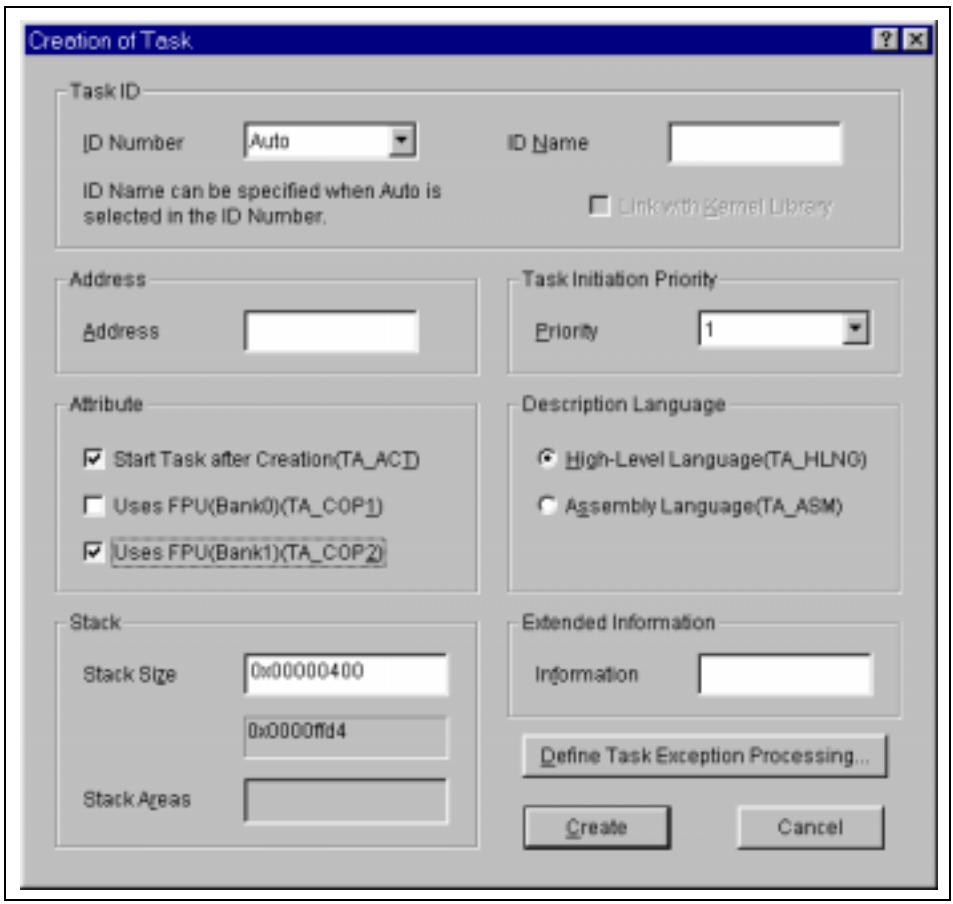


Figure 1.33 FPU Selection in Configurator (TA_COP2)

(Continued on next page)

(Continued from previous page)

Answer

When the task registered in the configurator uses both FPU banks 0 and 1, specify the [Use FPU (Bank 0) (TA_COP1)] and [Use FPU (Bank 1) (TA_COP2)] check boxes for [Attribute] in the Creation of Task window.

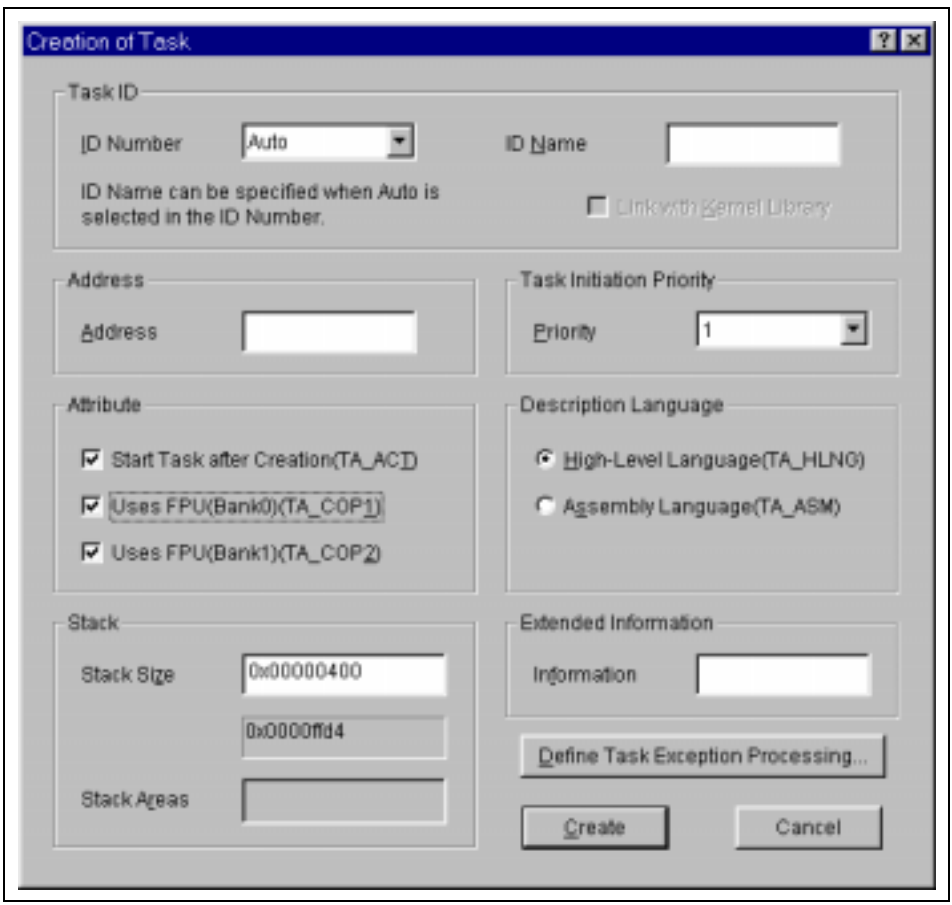


Figure 1.34 FPU Selection in Configurator (TA_COP1 and TA_COP2)

(Continued on next page)

(Continued from previous page)

Answer

To create a task by a service call during system operation, specify TA_COP1 or TA_COP2 as the task attribute parameter in the cre_tsk service call.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

#pragma noregsave(MainTask)

void MainTask(VP_INT stacd)
{
  ER         ercd;
  T_CTSK pk_ctsk;

     (processing discription omitted)

    pk_ctsk.tskatr = (TA_HLNG | TA_COP1)                /* Task attribute = high-level language description,
                                                           FPU coprocessor bank 0 used                  */ ← (1)
//  pk_ctsk.tskatr = (TA_HLNG | TA_COP2)                /* Task attribute = high-level language description,
                                                           FPU coprocessor bank 1 used                  */ ← (2)
//  pk_ctsk.tskatr = (TA_HLNG | TA_COP1 | TA_COP2)      /*Task attribute = high-level language description,
                                                           FPU coprocessor banks 0 and 1 used           */ ← (3)
    pk_ctsk.exinf  = 0;                                 /* Extended infomation = 0                      */
    pk_ctsk.task   = (FP)task_A;                        /* Task initiation address                      */
    pk_ctsk.itskpri = 1;                                /* Priority at task initiation                  */
    pk_ctsk.stksz  = 264;                               /* Task stack size                              */
    pk_ctsk.stk     = (VP)sp_taskA;                     /* Start address of task stack area             */

    ercd=cre_tsk(TASK_A, &pk_ctsk);                     /* Create task A                                */
    ercd=sta_tsk(TASK_A,( VP_INT)0x00000001);           /* Initiate task A (initiation code = 0x1)      */

     (processing discription omitted)

}
```

Figure 1.35 FPU Selection for Task Creation by Service Call (Sample Code)

(1) Task attribute specification when the FPU functions are used in bank 0.

(2) Task attribute specification when the FPU functions are used in bank 1.

(3) Task attribute specification when the FPU functions are used in banks 0 and 1.

For details on the task attribute parameter during task creation, refer to the HI7000/4 Series User's Manual.

When a non-task program (such as an interrupt handler or time event handler) uses FPU functions, each program must save and restore the FPU registers. For details, refer to the HI7000/4 Series User's Manual.

1.5 Interrupts

1.5.1 Processing before Handler Initiation after Interrupt Occurrence

This section gives an overview of the processing for an interrupt generated during task execution.

(1) H8S and H8SX Family Microcomputers

Figure 1.36 gives an overview of the processing before an interrupt handler is initiated after an interrupt occurs.

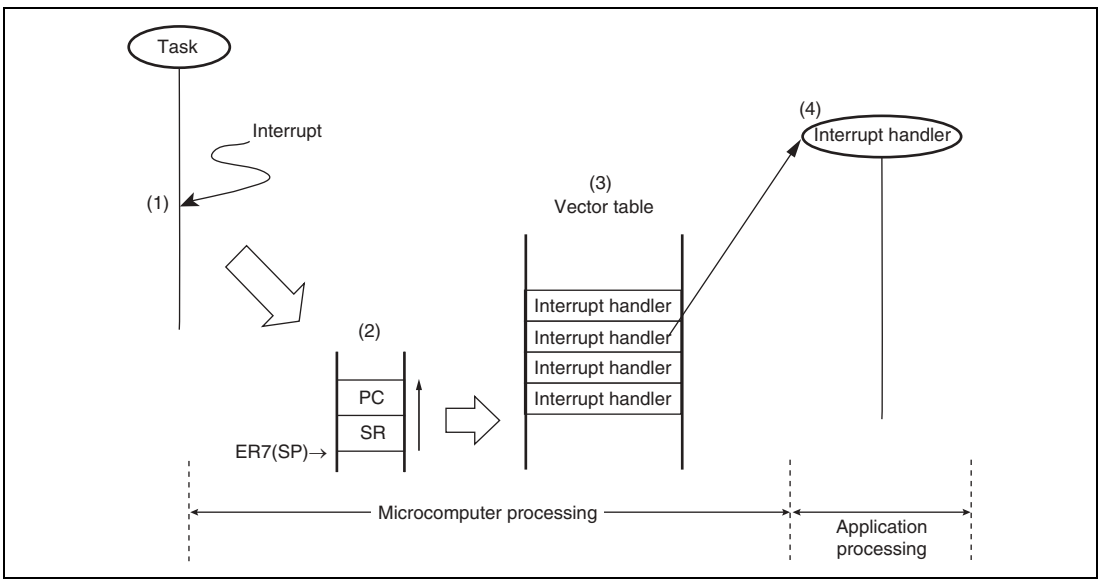


Figure 1.36 Overview of Processing before Handler Initiation after Interrupt Occurrence (1)

1. The microcomputer detects an interrupt generated during task (or interrupt handler) execution.
2. The microcomputer saves the SR and PC register information in the current stack.
3. The microcomputer analyzes the interrupt source and obtains the address of the corresponding interrupt handler registered in the vector table.
4. The interrupt handler registered in the vector table is initiated.

(2) SH-1, SH-2, and SH2-DSP Series Microcomputers

Figure 1.37 gives an overview of the processing before an interrupt handler is initiated after an interrupt occurs.

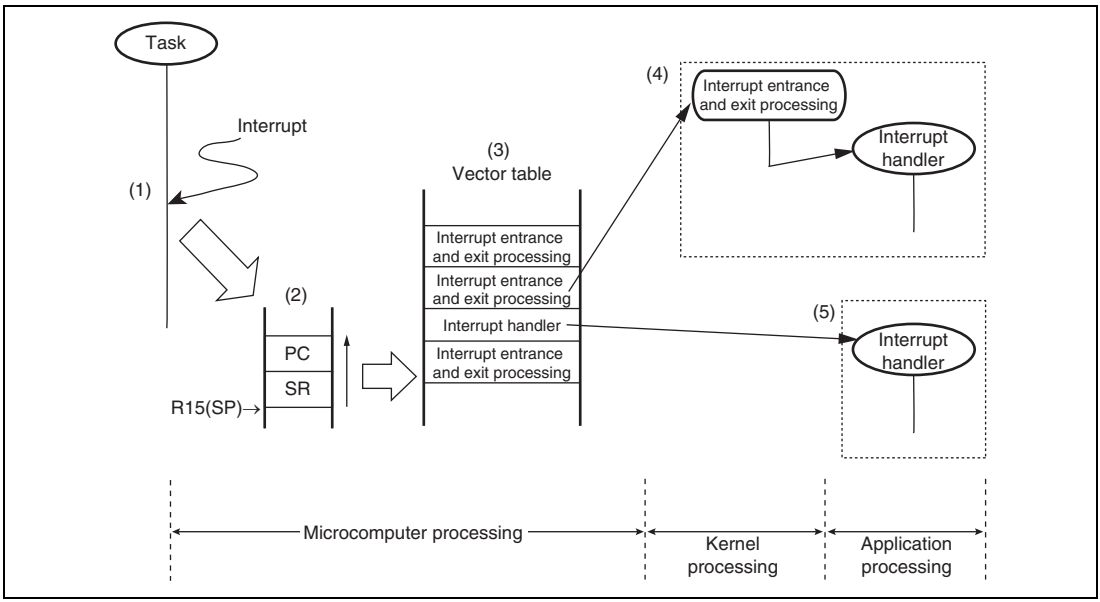


Figure 1.37 Overview of Processing before Handler Initiation after Interrupt Occurrence (2)

1. The microcomputer detects an interrupt generated during task (or interrupt handler) execution.
2. The microcomputer saves the SR and PC register information in the current stack.
3. The microcomputer analyzes the interrupt source and obtains the address of the corresponding interrupt handler registered in the vector table.
4. When the address registered in the vector table points to the interrupt entrance and exit processing, the interrupt entrance and exit processing provided by the kernel is performed, and then the interrupt handler is initiated.
   Note: The interrupt handler initiated through the interrupt service routine (kernel) is the usual interrupt handler.
5. When the address registered in the vector table points to an interrupt handler, the interrupt handler is directly initiated without involving kernel management.
   Note: The interrupt handler directly initiated without involving the interrupt service routine (kernel) is called a direct interrupt handler.

Note: The direct interrupt handler is only supported by the HI7000/4.

The interrupt entrance and exit processing is called the interrupt service routine.

(3) SH-3, SH3-DSP, and SH-4 Series Microcomputers

Figure 1.38 gives an overview of the processing before an interrupt handler is initiated after an interrupt occurs.

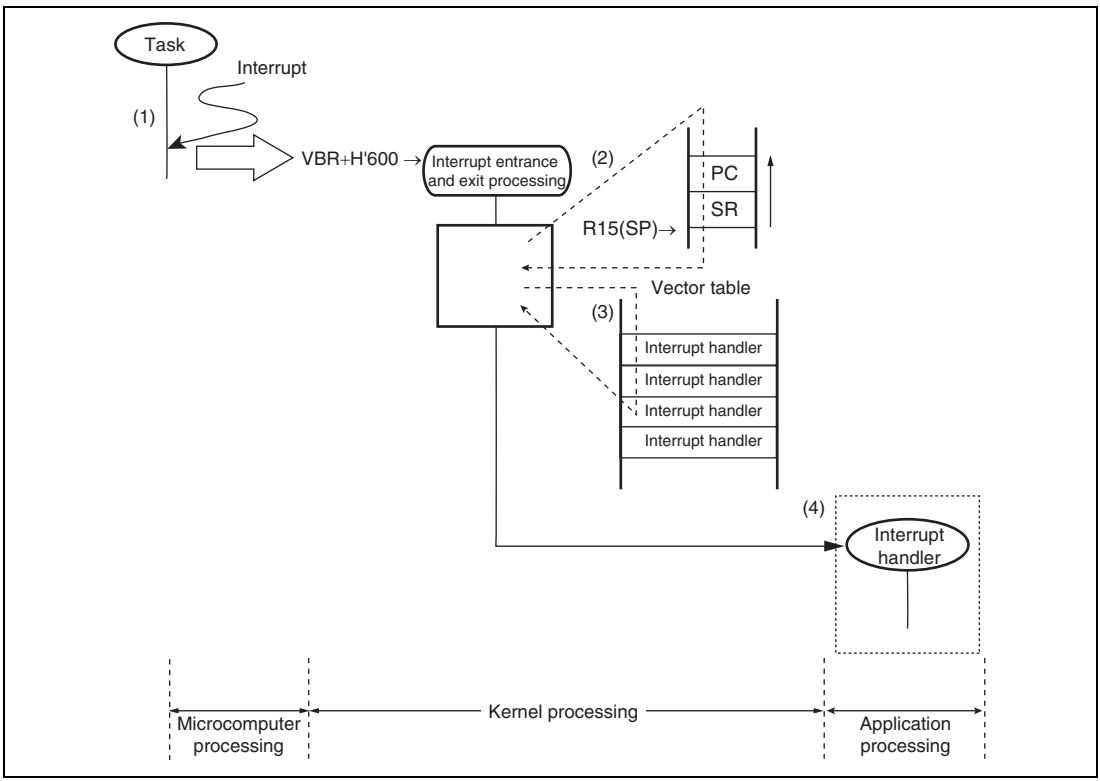


Figure 1.38 Overview of Processing before Handler Initiation after Interrupt Occurrence (3)

1. The microcomputer detects an interrupt generated during task (or interrupt handler) execution and modifies the PC value to a specified address (VBR value + H'600).
   Note: In the HI series OS, the interrupt entrance and exit processing (interrupt service routine) is located at this address (VBR value + H'600) in advance.
2. The microcomputer saves the SR and PC register information in the current stack.
3. The microcomputer analyzes the interrupt source and obtains the address of the corresponding interrupt handler registered in the vector table.
4. The interrupt handler is initiated.

1.5.2 Kernel Interrupt Mask Level

The kernel has a critical section where execution is performed with interrupts masked to prevent conflict occurring in kernel internal information.

- Acceptance of an interrupt generated during execution of the critical section in the kernel is delayed until execution of the critical section finishes.
- The critical section is processed at the kernel interrupt mask level.

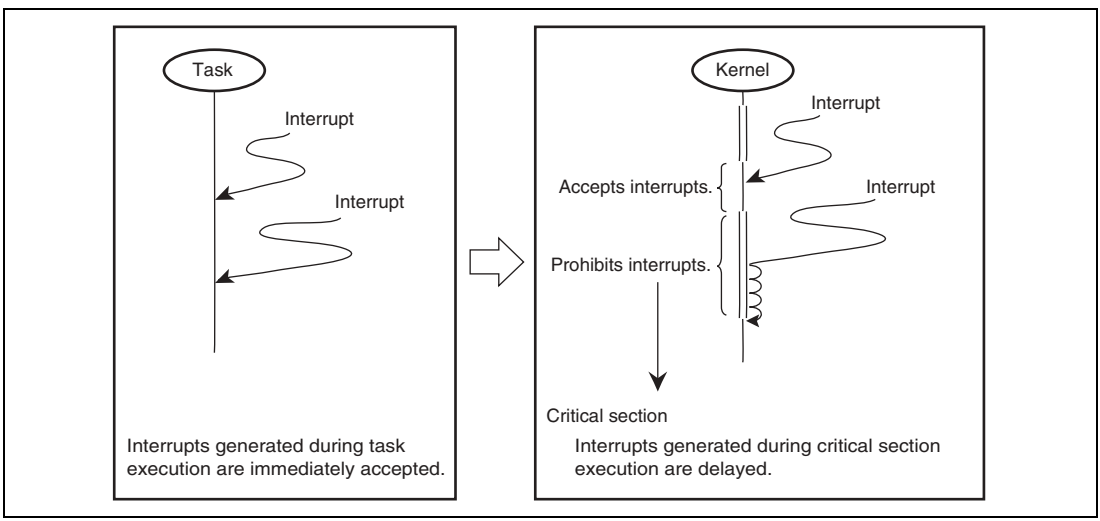


Figure 1.39 Overview of Interrupt Mask by Kernel

Note: Interrupts with interrupt levels higher than the kernel interrupt mask level are accepted immediately even during execution of the critical section.

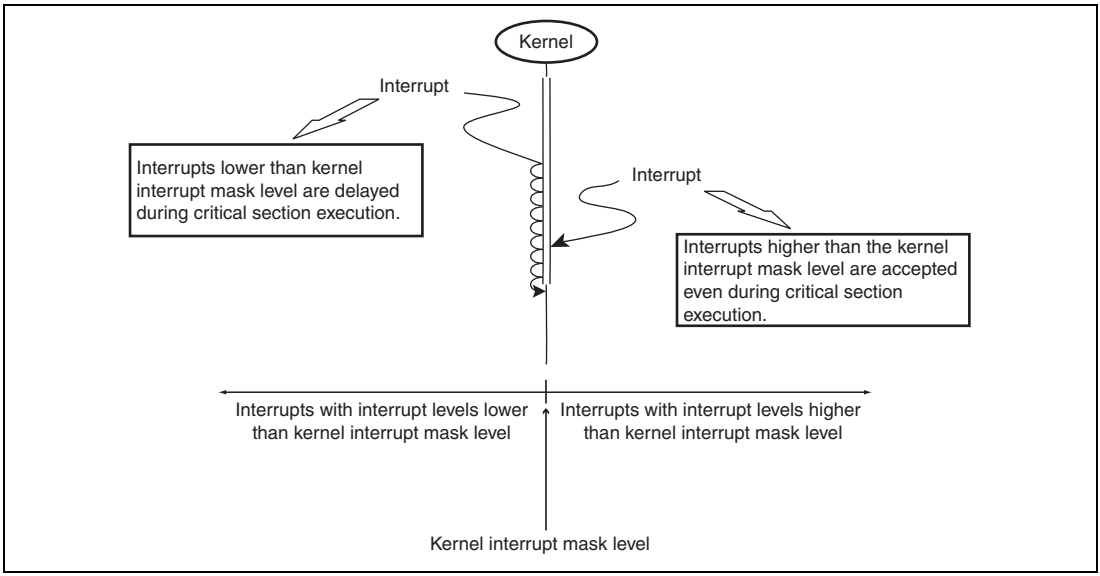


Figure 1.40 Kernel Interrupt Mask Level and Interrupt Levels

Notes on Interrupt Handlers with Higher Levels than Kernel Interrupt Mask Level:

- Service calls cannot be issued by interrupt handlers with interrupt levels higher than the kernel interrupt mask level. If called, normal system operation cannot be guaranteed.
- Execute the RTE instruction to return from an interrupt handler with an interrupt level higher than the kernel interrupt mask level.

1.5.3 Notes When Using an H8S or H8SX Family Microcomputer

When using an H8S or H8SX family microcomputer, note that the acceptable interrupts depend on the combination of the interrupt control mode and the mask level value. The HI series OS can be used in the four interrupt control modes of the H8S family microcomputers and in the two interrupt control modes of the H8SX family microcomputers.

The following tables show the relationship between the interrupt mask levels in each interrupt control mode and the acceptable interrupts (either 0 or 1 can be specified for the shaded sections in the tables).

Table 1.15 Interrupt Mask Levels in Interrupt Control Mode 0

Interrupt Mask Level (imask)	CCR Value		EXR Value			Acceptable Interrupts
	I	UI	I2	I1	I0	
1	1					Only NMI
0	0					All

Table 1.16 Interrupt Mask Levels in Interrupt Control Mode 1

Interrupt Mask Level (imask)	CCR Value		EXR Value			Acceptable Interrupts
	I	UI	I2	I1	I0	
3	1	1				Only NMI
2	1	0				Control level 1
1	0	1				All
0	0	0				All

Table 1.17 Interrupt Mask Levels in Interrupt Control Mode 2

Interrupt Mask Level (imask)	CCR Value		EXR Value			Acceptable Interrupts
	I	UI	I2	I1	I0	
7			1	1	1	Only NMI
6			1	1	0	Priority level 7
5			1	0	1	Priority levels 6 to 7
4			1	0	0	Priority levels 5 to 7
3			0	1	1	Priority levels 4 to 7
2			0	1	0	Priority levels 3 to 7
1			0	0	1	Priority levels 2 to 7
0			0	0	0	All

Table 1.18 Interrupt Mask Levels in Interrupt Control Mode 3

Interrupt Mask Level (imask)	CCR Value		EXR Value			Acceptable Interrupts
	I	UI	I2	I1	I0	
8	1	1	1	1	1	Only NMI
7	1	0				Control level 1
6	0	0	1	1	0	Priority level 7 at control levels 0 and 1
5	0	0	1	0	1	Priority levels 6 to 7 at control levels 0 and 1
4	0	0	1	0	0	Priority levels 5 to 7 at control levels 0 and 1
3	0	0	0	1	1	Priority levels 4 to 7 at control levels 0 and 1
2	0	0	0	1	0	Priority levels 3 to 7 at control levels 0 and 1
1	0	0	0	0	1	Priority levels 2 to 7 at control levels 0 and 1
0	0	0	0	0	0	All

Note: If level 7 is used as the kernel interrupt mask level in interrupt control mode 3, service calls cannot be issued by an interrupt handler of control level 1.

1.5.4 Notes on Interrupt Handler Creation

Note the following when creating interrupt handlers.

Table 1.19 Notes on Interrupt Handler Creation

Item	Note
Interrupt handler execution (processing) time	A long execution time degrades the system throughput.
	The execution time strongly affects the system response.
Service calls from interrupt handler*[1]	Interrupt handlers with interrupt levels higher than the kernel interrupt mask level cannot issue service calls.
	The NMI interrupt handler cannot issue service calls.
Return from interrupt handler*[2]	Issue the ret_int service call*[3] to return from an interrupt handler with an interrupt level equal to or lower than the kernel interrupt mask level. Use the RTE instruction to return from an interrupt handler with an interrupt level higher than the kernel interrupt mask level.

Notes:
1. If an ext_tsk (exd_tsk) service call is issued, execution is shifted to the system termination routine.
2. If a method other than the ret_int service call is used, correct system operation cannot be guaranteed.
3. The HI7000/4 series does not support the ret_int service call; therefore, it is not necessary in the HI7000/4 series.

1.5.5 FAQs about Interrupts

This section answers questions about interrupts which are frequently asked by users of the HI series OS.

FAQ Contents:

(1) Modifying Interrupt Mask

Classification: Interrupt					
Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4
Is the use of set_imask() to modify the interrupt mask level prohibited?					

Answer

set_imask() does not process the internal OS information. Accordingly, if a service call of the OS is issued after set_imask() is called, correct operation cannot be guaranteed.

The OS recognizes the system state according to the interrupt mask information. The OS not only distinguishes between the task context and the non-task context but also manages the dispatch-disabled state and CPU-locked state in the task context, and the CPU-locked state in the non-task context. Service calls are used for processing of the internal information under the OS control in addition to interrupt mask processing.

For this reason, we recommend that the service call provided by the OS should be used to modify the interrupt mask level.

(2) Multiple Interrupts

Classification: Interrupt					
Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4
Does the number of interrupts that occur during system operation affect system performance?					

Answer

The number of interrupts used does affect system performance. Specifically, though, the levels of interrupts, when there are multiple interrupts, rather than the total number of interrupts has the greatest effect on system performance.

For example, if an interrupt occurs whose level is higher than the target interrupt function (hereafter called the interrupt handler), the higher-level interrupt handler is processed first and the target interrupt handler is suspended until the higher-level interrupt handler processing ends. The order of processing is determined by the interrupt levels rather than the order of occurrence, and in this way the interrupt levels affect system operation.

If all interrupt handlers used in the system are set to the same level, the interrupt handler for emergency use may not be initiated immediately (interrupt acceptance may be delayed) in some cases.

Accordingly, the interrupt levels or processing priority of the interrupt handlers used in the system must be carefully considered.

(Continued on next page)

(Continued from previous page)

Answer

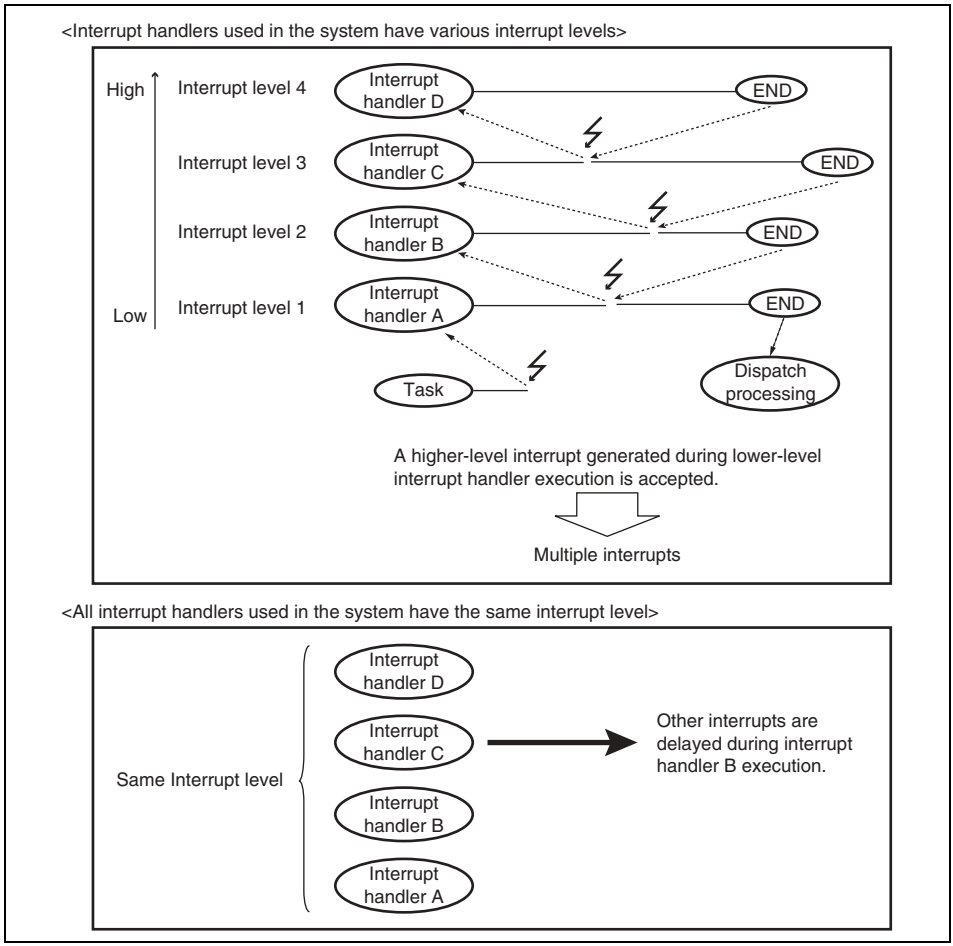


Figure 1.41 Multiple Interrupts

(3) Processing before Initiating Interrupt Handler

Classification: Interrupt

Question		HI7700/4	HI7750/4		

It takes an extremely long time before an interrupt handler is initiated after an interrupt occurs.

Please explain the processing before interrupt handler execution after an interrupt occurrence.

Answer

Figure 1.42 gives an overview of the processing before the interrupt handler is initiated after an interrupt occurs.

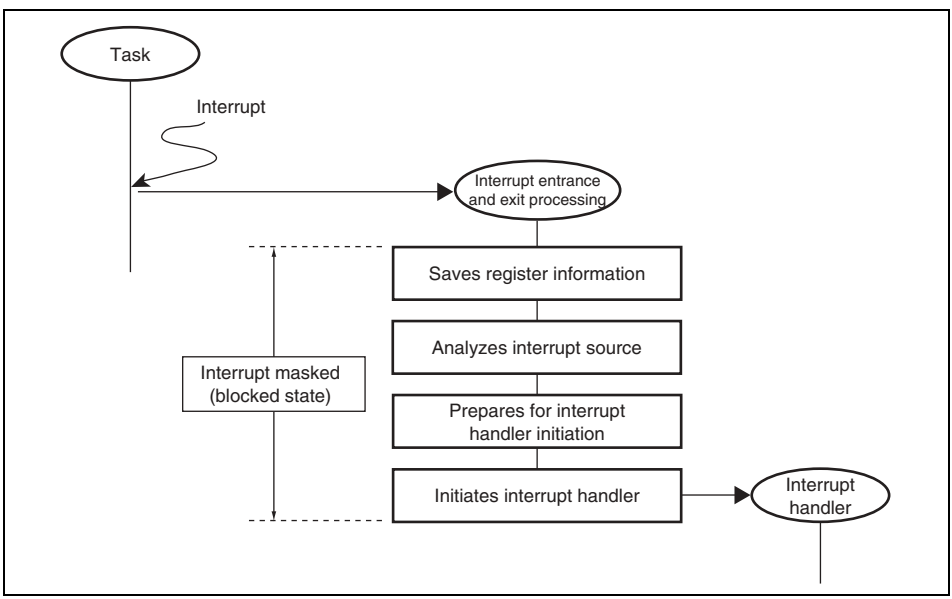


Figure 1.42 Overview of Processing before Interrupt Handler Initiation after Interrupt Occurrence

(Continued on next page)

(Continued from previous page)

Answer

If it takes an extremely long time before an interrupt handler is initiated after an interrupt occurs, the following possible causes should be checked.

- A higher-level interrupt occurs when the interrupt handler is initiated.
- A higher-level interrupt occurs immediately before the interrupt handler is initiated.
- When the interrupt handler is initiated, interrupts for the current processing are masked with a higher-level than the interrupt level of the interrupt handler.

(4) Terminating Interrupt Handler

Classification: Interrupt					
Question				HI2000/3	HI1000/4
Should any interrupt handler whose level is not higher than the kernel interrupt mask level be terminated by ret_int even when it issues no service call?					

Answer

After interrupt handler processing, the ret_int service call should be used for the following purposes.

- To recognize the interrupt nesting
- To recognize the task switching

By using ret_int for the above purposes, correct return processing will be done.

If the RTE instruction is used, execution returns to the interrupt generating function, and task switching due to a service call by the interrupt handler or the timeout task due to the timer handler cannot be recognized. This will result in a contradiction in the system status. To avoid such a contradiction, the HI series OS provides the ret_int service call.

When the system uses the timeout function, interrupt handlers whose interrupt levels are lower than the timer driver interrupt level must be terminated by ret_int regardless of whether they issue service calls to recognize the timeout task due to the timer handler and avoid any contradiction in the system status.

When the system does not use the timeout function, the termination of interrupt handlers depends on whether the handlers issue service calls.

When the system does not use the timeout function and if all interrupt handlers in the system do not issue service calls affecting task switching, they can be terminated by the RTE instruction regardless of the interrupt nesting levels.

(Continued on next page)

(Continued from previous page)

Answer

When the system does not use the timeout function, and if some interrupt handlers issue service calls affecting task switching but interrupts are never nested, the interrupt handlers must be terminated in the following ways.

- The interrupt handlers that issue service calls affecting task switching must be terminated by ret_int.
- The interrupt handlers that do not issue service calls affecting task switching must be terminated by the RTE instruction.

If interrupts are nested, the interrupt handlers must be terminated in the following ways.

- The interrupt handlers whose interrupt levels are not higher than any interrupt handlers that issue service calls affecting task switching must be terminated by ret_int regardless of whether they issue service calls affecting task switching (because whether task switching is required must be recognized).
- The interrupt handlers that do not issue service calls affecting task switching and whose interrupt levels are higher than any interrupt handlers that issue service calls affecting task switching must be terminated by the RTE instruction.

Figure 1.43 shows a sample code of an interrupt handler.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

extern VP int_stk001;                                   ← (1)
                                                        ← (2)
static const VP p_stk = (VP) & int_stk001;              ← (3)
                                                        ← (4)
#pragma interrupt (Inhhdr (sp = p_stk, sy = $ret_int))

void Inhhdr (void){

   /* Interrupt handler processing */
}
```

Figure 1.43 Sample Code of Interrupt Handler

(Continued on next page)

(Continued from previous page)

Answer

(1) Specify the allocated interrupt stack.

(2) Define the initial value of the stack pointer as a const type value.

(3) Declare the interrupt handler as an interrupt function by #pragma interrupt.

— Specify stack switching (sp=p_stk)

— Specify interrupt function end (sy=$ret_int)

(4) Describe the interrupt handler as a void type function.

(5) Interrupt Handlers that Are Not Managed by the OS

Classification: Interrupt					
Question	HI7000/4	HI7700/4	HI7750/4		
To process a specific interrupt handler prior to any other processing, an interrupt should be made without involving OS management. How can this be done?					

Answer

Interrupt handlers higher than the kernel interrupt mask level are processed outside of the kernel management and are suitable when a specific interrupt handler should be processed without involving OS management. Note, however, that such interrupt handlers cannot issue any service call.

In another way, the HI7000/4 provides the direct interrupt handler function, which initiates interrupt handlers without involving kernel operation. The direct interrupt handlers are also processed outside of the kernel management and cannot issue any service call, but this function is suitable for top-priority processing of a specific interrupt handler without involving OS management.

(6) Restrictions on Direct Interrupt Handler Usage

Classification: Interrupt					
Question	HI7000/4				
Are there any restrictions on direct interrupt handler usage?					

Answer

Note the following restrictions when using direct interrupt handlers.

- No service call can be issued from direct interrupt handlers.
- Direct interrupt handlers must be defined through the configurator.
  (Dynamic creation is not available for direct interrupt handlers.)
- The stack must be switched to that for interrupt handlers.
- TRAPA #25 must be used to return from a direct interrupt handler.

For details, also refer to the user's manual of the OS used.

(7) Sample Definition File Information

Classification: Interrupt					
Question				HI2000/3	HI1000/4

Our system has the following configuration:

[Interrupt mode: 2]

NMI: Not used
Interrupt level 7: Not used
Interrupt level 6: Kernel and cyclic handler (TPU0)
Interrupt level 5: Timer interrupts (TPU1, 2, 3, 4, and 5)
Interrupt level 4: External interrupts (IRQ0, 10, and 15)
Interrupt level 3: External interrupts (IRQ1 and 11)
Interrupt level 2: External interrupts (IRQ4 and 5)
Interrupt level 1: DMAC (DMTEND0A)
Interrupt level 0: Not used

If interrupts can be nested at the maximum level, please answer the following questions.

1. What level is the nesting of interrupts which are equal to or lower than the kernel interrupt mask level?
2. What level is the nesting of interrupts which are higher than the kernel interrupt mask level (including NMI)?
3. What level is the nesting of interrupts which are equal to or lower than the kernel interrupt mask level and higher than the timer interrupt level?
4. What level is the nesting of interrupts which are equal to or lower than the kernel interrupt mask level and higher than the TPU0 interrupt level?
5. What level is the nesting of interrupts which are equal to or lower than the kernel interrupt mask level and higher than the IRQ0 interrupt level?

(Continued on next page)

(Continued from previous page)

Answer

A1:
Interrupts equal to or lower than the kernel interrupt mask level will be nested when a level-1 interrupt occurs during task execution or when a level-2 interrupt handler is executed during level-1 interrupt handler processing. As the kernel interrupt mask level is 6, interrupts equal to or lower than level 6 will be nested unconditionally. Therefore, the nesting level for interrupts equal to or lower than the kernel interrupt mask level is 6.

A2:
As no interrupt is defined as higher than the kernel interrupt mask level, the nesting level for interrupts higher than the kernel interrupt mask level (including NMI) is 0.

A3:
The cyclic handler (TPU0) satisfies the condition for interrupts equal to or lower than the kernel interrupt mask level (level 6 in this case) and higher than the timer interrupt level (level 5 in this case). Therefore, the nesting level is 1.

A4:
There is no interrupt handler defined as being equal to or lower than the kernel interrupt mask level (level 6 in this case) and higher than the TPU0 interrupt level (level 6 in this case). Therefore, the nesting level is 0.

A5:
The timer interrupts (TPU1, 2, 3, 4, and 5) and cyclic handler (TPU0) satisfy the condition for interrupts equal to or lower than the kernel interrupt mask level (level 6 in this case) and higher than the IRQ0 interrupt level (level 4 in this case). Therefore, the nesting level is 2.

(8) Task Switching from Interrupt Handler

Classification: Interrupt

Question				HI2000/3	HI1000/4

After irot_rdq(0) is executed in an interrupt handler, task switching does not immediately occur. Why is this?

Answer

Task switching occurs when the dispatcher is initiated after the interrupt handler processing is completed. However, the dispatcher may not be initiated for the following reasons. Check the description of the interrupt handler that issues irot_rdq(0) and the system status when an interrupt occurs.

1. Descriptions in the interrupt handler

   For the HI2000/3, the interrupt handler is written by using an assembly directive of the cross compiler as follows.

```
#pragma interrupt (parameter1 (sp = parameter2, sy = parameter3) )

1. parameter1: Start address of the interrupt handler
2. parameter2: Bottom address of the stack area for the interrupt handler
3. parameter3: Interrupt handler termination processing
```

Figure 1.44 Example of #pragma interrupt Usage

parameter3 should be specified as follows according to the combination of the interrupt level of the interrupt handler and the kernel interrupt mask level.

(1) No service call can be issued by an interrupt handler whose interrupt level is higher than the kernel interrupt mask level. Because such an interrupt handler must be terminated by the RTE instruction, parameter3 should not be specified (written).

(Continued on next page)

(Continued from previous page)

Answer

(2) An interrupt handler whose interrupt level is equal to or lower than the kernel interrupt mask level must issue the ret_int service call during interrupt handler termination processing. Therefore, parameter3 should be specified as sy = $ret_int.

If termination processing is not specified for an interrupt handler equal to or lower than the kernel interrupt mask level, task scheduling will not occur after interrupt handling.

2. System state when the interrupt handler processing ends

   Task scheduling may not occur depending on the system state when the interrupt cause is generated as follows.

   (1) If the system is in the dispatch-disabled state when the interrupt cause is generated, which means that task switching is disabled, the task being executed when the interrupt occurs continues processing after the interrupt handler processing ends.

   (2) Even if the system is in the task RUNNING state, if the interrupt mask level is set to a value other than 0 by the chg_ims service call issued by the task being executed at that time, the task being executed when the interrupt occurs continues processing after the interrupt handler processing ends.

1.6 Event Flags

1.6.1 Specification of Event Flag Clearing

The specification of event flag clearing (TA_CLR attribute setting) differs among the HI series OS specifications as follows.

Table 1.20 Differences in Specification of Event Flag Clearing

HI Series OS	Specification of Event Flag Clearing
HI2000/3	Specified for each task as a parameter (the fourth parameter) when a service call is issued
HI7000/4 and HI1000/4	Specified for each event flag when an event flag is created

Figure 1.45 gives an overview of the event flag processing when event flag clearing is not specified.

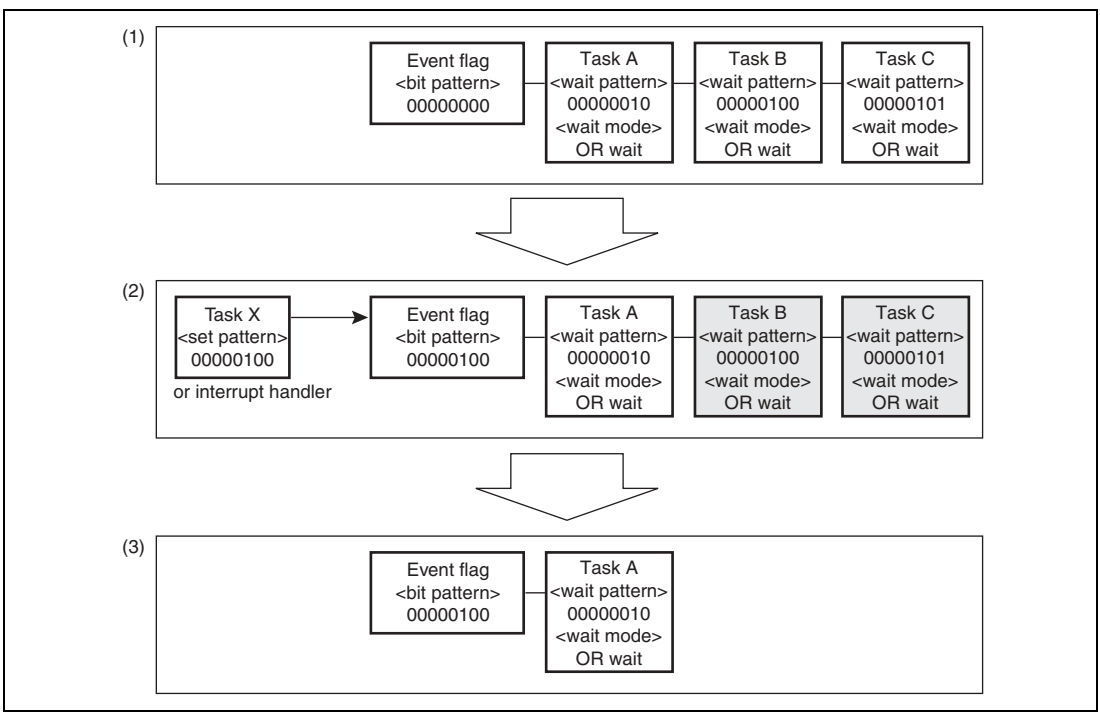


Figure 1.45 Overview of Event Flag Processing without Clearing

1. Tasks A, B, and C wait for an event flag.
2. Task X (or an interrupt handler) reports an event (sets a bit pattern) for the event flag.
3. The event flag clears the WAITING state of the tasks whose condition is satisfied (tasks B and C).

(1) Specification of TA_CLR Attribute in HI2000/3

Figure 1.46 gives an overview of the event flag processing in the HI2000/3 when event flag clearing is specified.

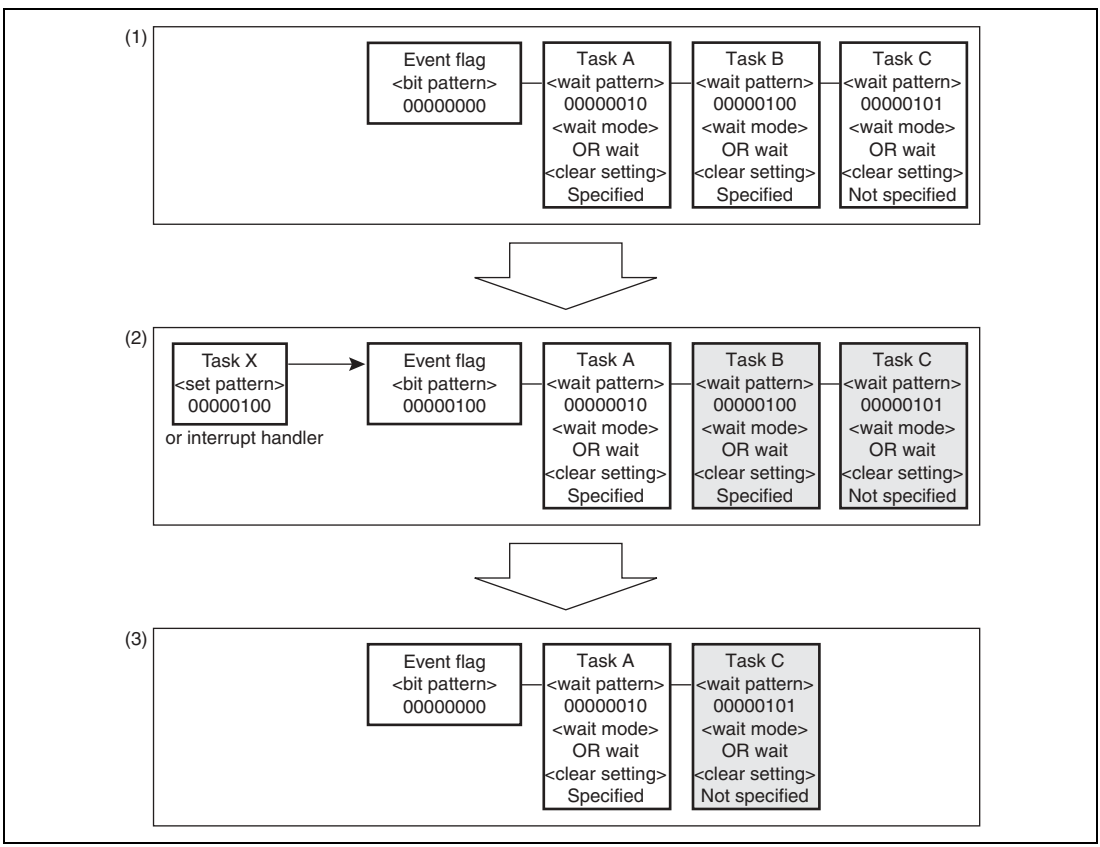


Figure 1.46 Overview of Processing with Clearing (HI2000/3)

1. Tasks A, B, and C wait for an event flag.
2. Task X (or an interrupt handler) reports an event (sets a bit pattern) for the event flag.
3. The event flag clears the WAITING state of the task whose condition is satisfied (task B) and immediately clears the bit pattern of the event flag.

When the TA_CLR attribute is specified, all bits in the bit pattern for the event flag are cleared when one waiting task is released from the WAITING state, and no more tasks are released from the WAITING state. The bit pattern before clearing is returned as the event flag bit pattern information at WAITING state clearing.

(2) Specification of TA_CLR Attribute in HI7000/4 Series and HI1000/4

Figure 1.47 gives an overview of the event flag processing in the HI7000/4 series and HI1000/4 when event flag clearing is specified.

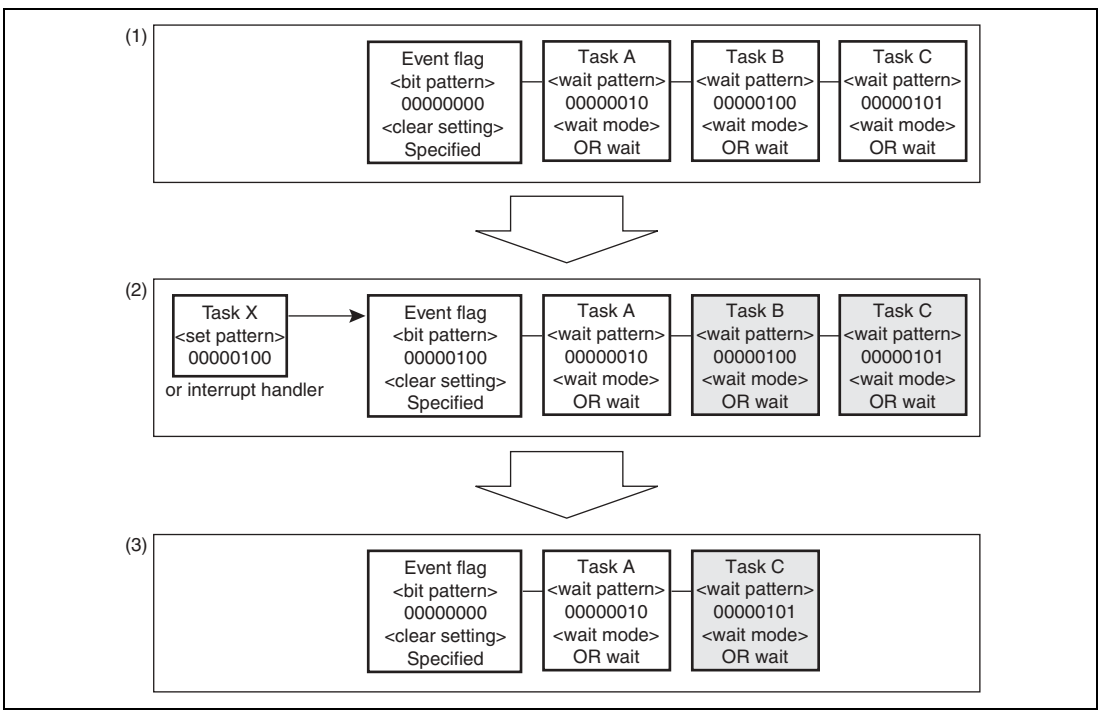


Figure 1.47 Overview of Processing with Clearing (HI7000/4 Series and HI1000/4)

1. Tasks A, B, and C wait for an event flag.
2. Task X (or an interrupt handler) reports an event (sets a bit pattern) for the event flag.
3. The event flag clears the WAITING state of the task whose condition is satisfied (task B) and immediately clears the bit pattern of the event flag.

The HI7000/4 series and HI1000/4 differ from the HI2000/3 in that event flag clearing is specified for the event flag itself.

1.6.2 FAQ about Event Flags

This section answers a question about event flags which is frequently asked by users of the HI series OS.

FAQ Contents:

(1) Clearing Event Flags

Classification: Event flag

Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4

Is there any way to clear an event flag after multiple tasks waiting for the same flag pattern are made ready?

Answer

A flag pattern can be cleared after multiple tasks waiting for the same flag pattern are released from the WAITING state. The following describes how to clear the flag in two cases: when setting the flag through a task and when setting it through an interrupt handler.

1. Setting the Flag through a Task

   Specify a higher priority level to the task that sets the event flag pattern than the tasks waiting for the event flag. After the service call processing is completed, execution returns to the task that sets the flag pattern. Perform the flag pattern clearing processing after the setting processing. Figure 1.48 shows a sample code.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

#pragma noregsave(EventFlag_Set_Task)

void EventFlag_Set_Task(VP_INT exinf)
{

  ER ercd;

    (processing description omitted)

    ercd = set_flg((ID)flgid, (FLGPTN)setptn);        /* Sets event flag */
    if(ercd != E_OK){
        /* Error processing */
    }else{
        ercd = clr_flg((ID)flgid, (FLGPTN)clrptn);    /* Clears event flag pattern */
        if(ercd != E_OK){
            /* Error processing */
        }
    }
    (processing description omitted)
}
```

Figure 1.48 Sample Code when a Task Sets the Event Flag

(Continued on next page)

(Continued from previous page)

Answer

2. Setting the Flag through an Interrupt Handler

   As the interrupt handler processing takes priority over the task and dispatcher processing, an event flag can be cleared after multiple tasks waiting for the flag pattern are released from the WAITING state by successively setting and clearing the event flag. Figure 1.49 shows a sample code.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

void EventFlag_Set_Interrupt(void)
{

ER ercd;

   (processing description omitted)

   ercd = iset_flg((ID)flgid, (FLGPTN)setptn);      /* Sets event flag */
   if(ercd != E_OK){
      /* Error processing */
   }else{
      ercd = iclr_flg((ID)flgid, (FLGPTN)clrptn);    /* Clears event flag pattern */
      if(ercd != E_OK){
        /* Error processing */
      }
   }
   (processing description omitted)
}
```

Figure 1.49 Sample Code when an Interrupt Handler Sets the Event Flag

1.7 Semaphore

1.7.1 Task Deadlock by Using Semaphore

A semaphore is used to manage resources that require exclusive control (such resources include software resources such as shared memory or non-reentrant functions in addition to hardware resources).

Figure 1.50 shows an example of semaphore usage.

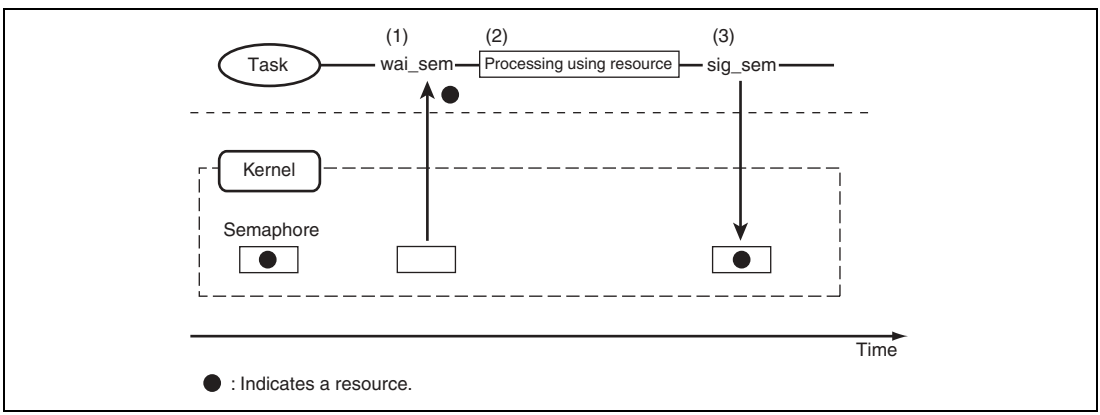


Figure 1.50 Semaphore Usage Example

1. A task obtains the semaphore.
2. Processing is performed by using the obtained resource.
3. The task releases the resource after completing the processing.

To use a resource that requires exclusive control, first obtain the semaphore, and then perform processing by using the resource. After completing the processing, release the semaphore.

The kernel does not provide a function to automatically release the obtained resource when the task completes processing; the task must always release the obtained semaphore when completing its processing.

Figure 1.51 shows an example of deadlock (tasks cannot operate).

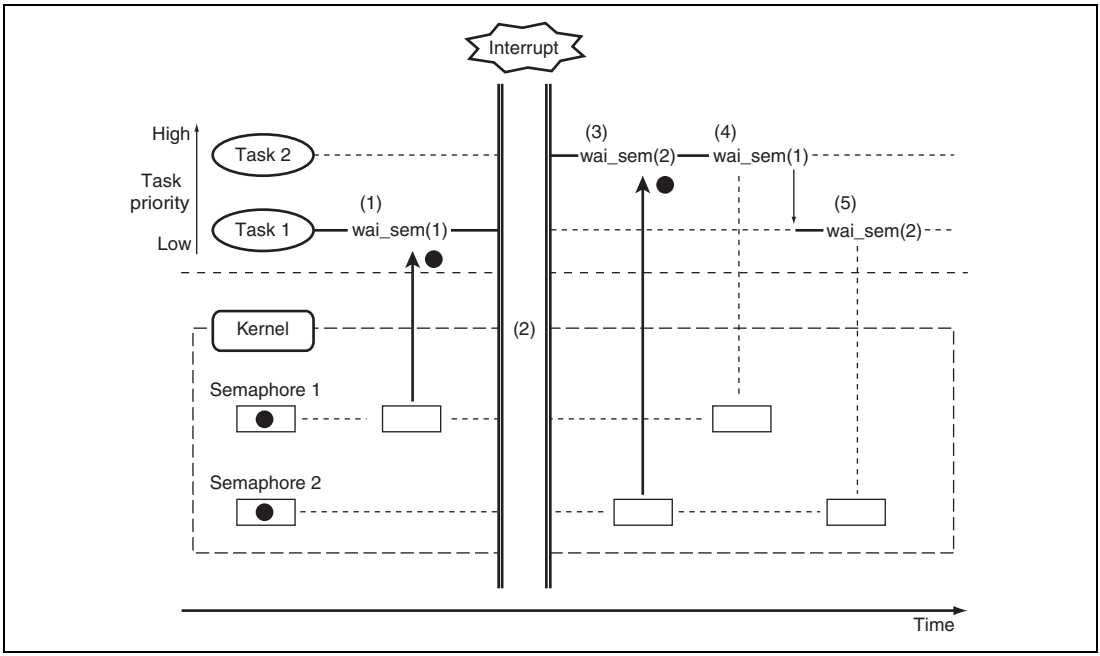


Figure 1.51 Deadlock Example (Tasks Cannot Operate)

1. Task 1 obtains semaphore 1.
2. An interrupt occurs, and the interrupt handler processing switches tasks (from task 1 to task 2).
3. Task 2 obtains semaphore 2.
4. Task 2 requests semaphore 1 but cannot obtain it because the resource (semaphore) has been obtained by task 1. Task 2 enters the WAITING state for release of the resource and tasks are switched (from task 2 to task 1).
5. Task 1 requests semaphore 2 but cannot obtain it because the resource (semaphore) has been obtained by task 2. Task 1 enters the WAITING state for release of the resource.

As a result, tasks 1 and 2 both wait for a semaphore which has been obtained by the other, and they will never be released from the WAITING state. This state is called deadlock.

Such deadlock cases cannot be avoided within the OS, and must be examined and solved during the design steps of the application (user system).

1.8 Mutex

1.8.1 Priority Inversion

Figure 1.52 gives an overview of priority inversion.

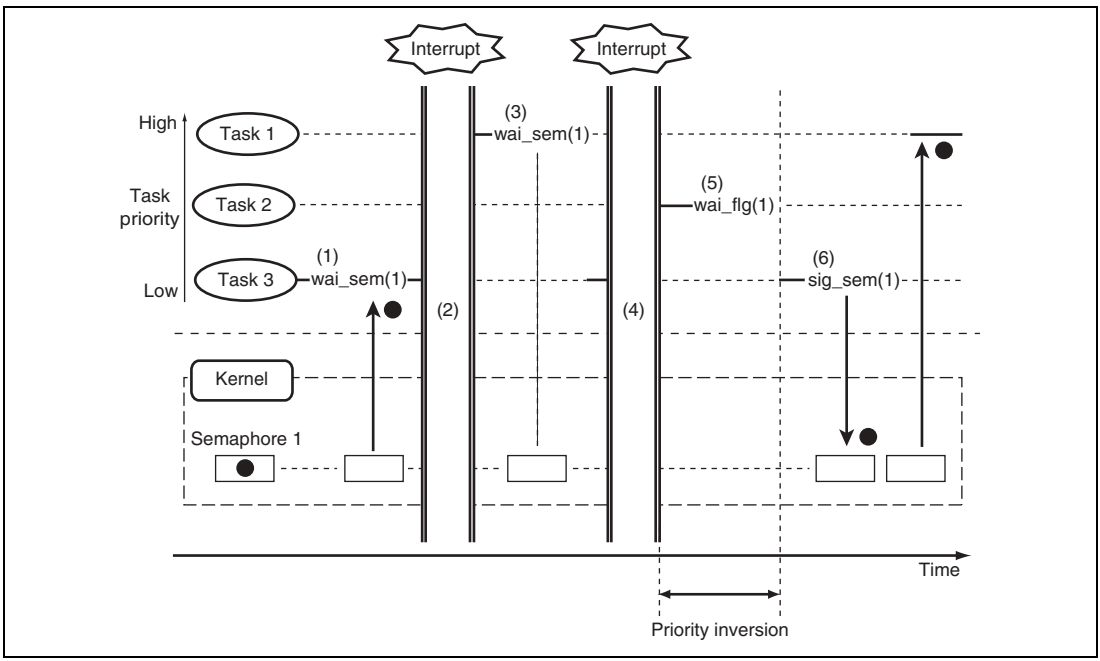


Figure 1.52 Overview of Priority Inversion

1. Task 3 obtains semaphore 1 and continues processing.
2. An interrupt occurs and the interrupt handler processing switches tasks (from task 3 to task 1).
3. Task 1 requests semaphore 1 but cannot obtain it because the resource (semaphore) has been obtained by task 3. Task 1 enters the WAITING state for release of the resource, and tasks are switched (from task 1 to task 3).
4. An interrupt occurs and the interrupt handler processing switches tasks (from task 3 to task 2).
5. Task 2 issues an event wait request and tasks are switched (from task 2 to task 3).
6. Task 3 completes the processing that uses the resource, and releases semaphore 1. At this time, task 1, which has been waiting for release of the resource, obtains semaphore 1 and resumes processing.

Higher-priority task 1 should be executed instead of task 2, but it cannot be executed because the resource (semaphore) needed for task 1 processing has been obtained by lower-priority task 3. Such a problem where a higher-priority task is kept pending because of the lower-priority task processing is called priority inversion.

1.8.2 Overview of Mutex Processing

Figure 1.53 gives an overview of mutex processing.

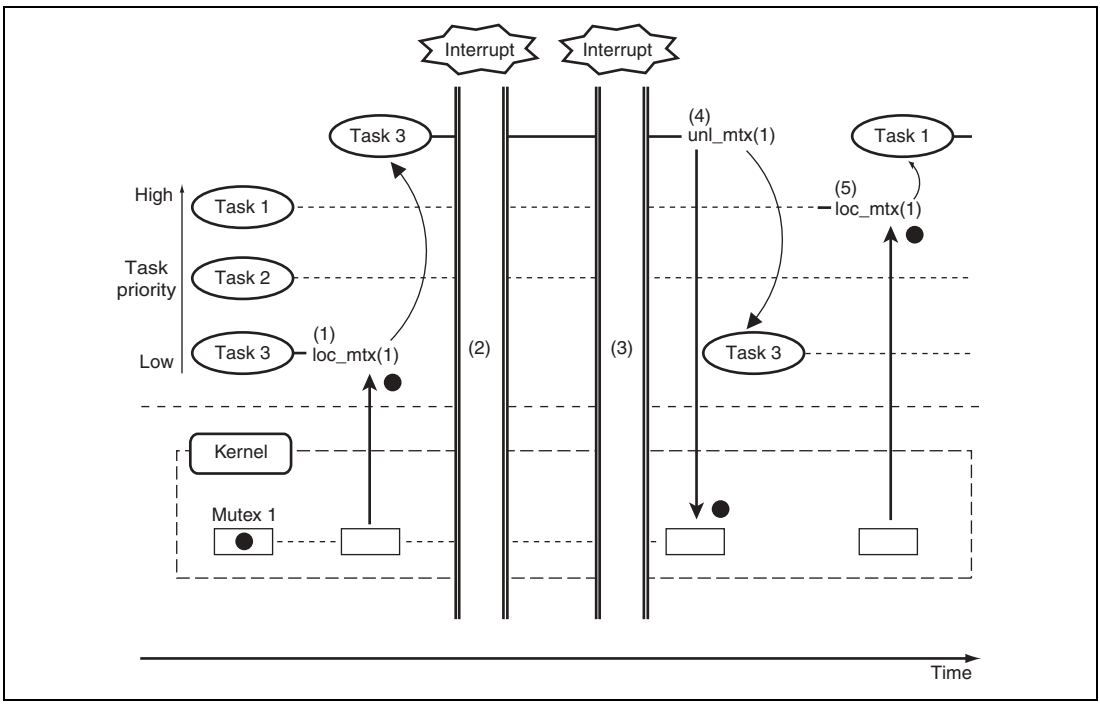


Figure 1.53 Overview of Mutex Processing

1. Task 3 obtains mutex 1 and continues processing. (At this time, the priority of task 3 is raised from 3 to 1 (ceiling priority).)
2. An interrupt occurs and the interrupt handler processing wakes up task 1. However, task switching does not occur because task 3 has the highest priority (task 3 is held at priority 1 and continues processing).
3. An interrupt occurs and the interrupt handler processing wakes up task 2. However, task switching does not occur because task 3 has the highest priority (task 3 is held at priority 1 and continues processing).

4. Task 3 completes the processing that uses the resource, and releases mutex 1. (At this time, the priority of task 3 is restored from 1 to 3, and control is switched to task 1.)
5. Task 1 obtains mutex 1 and continues processing. (At this time, the priority of task 1 is raised from 1 to 1 (ceiling priority).)

The task that obtains a mutex (locks a mutex) is executed by being automatically raised to the ceiling priority specified for the mutex, and can continue processing without entering the WAITING state even when task 1 or 2 becomes ready.

When task 3 releases the mutex (unlocks the mutex), it is modified back to the previous priority and tasks 1 and 2 are executed in that order.

1.9 Mailbox

1.9.1 Overview of Mailbox Processing

Table 1.21 summarizes the advantages and disadvantages of using mailboxes.

Table 1.21 Advantages and Disadvantages of Using Mailboxes

Advantages	Disadvantages
• Small overhead because only the message storing address is transferred. • No limitation on the message amount because the messages are managed by using a link list. • A large amount of message can be sent.	Shared memory (or a shared address space) must be prepared.

Figure 1.54 gives an overview of mailbox processing.

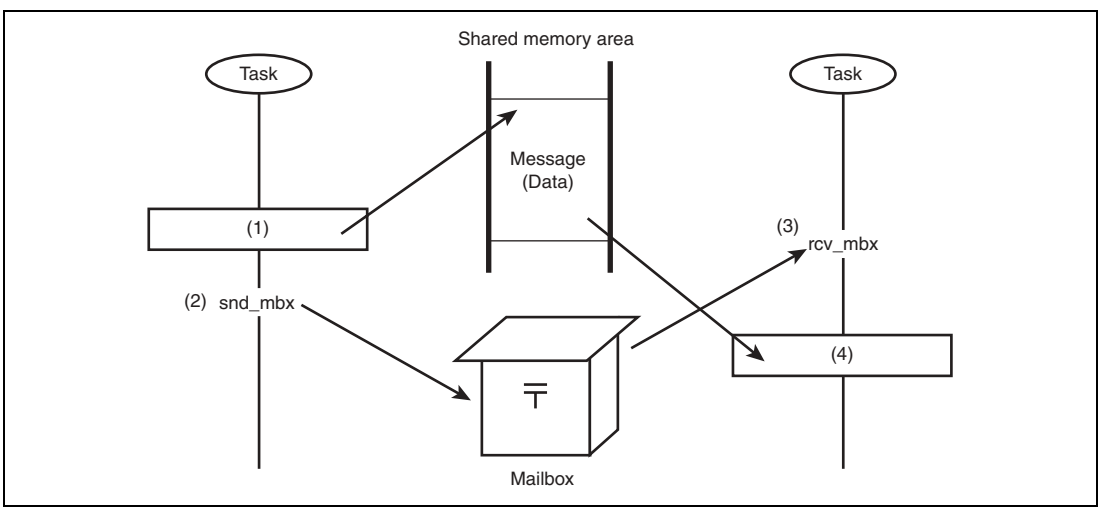


Figure 1.54 Overview of Mailbox Processing

1. Allocate a memory area where a message is to be stored, and write a message in that area.
2. Issue a snd_mbx service call to send the message address to the mailbox.
3. Issue a rcv_mbx service call to receive the message address from the mailbox.
4. Read the information in the area indicated by the received message address.

1.9.2 Overview of Sending a Message Using Mailbox

Figure 1.55 gives an overview of sending a message using a mailbox.

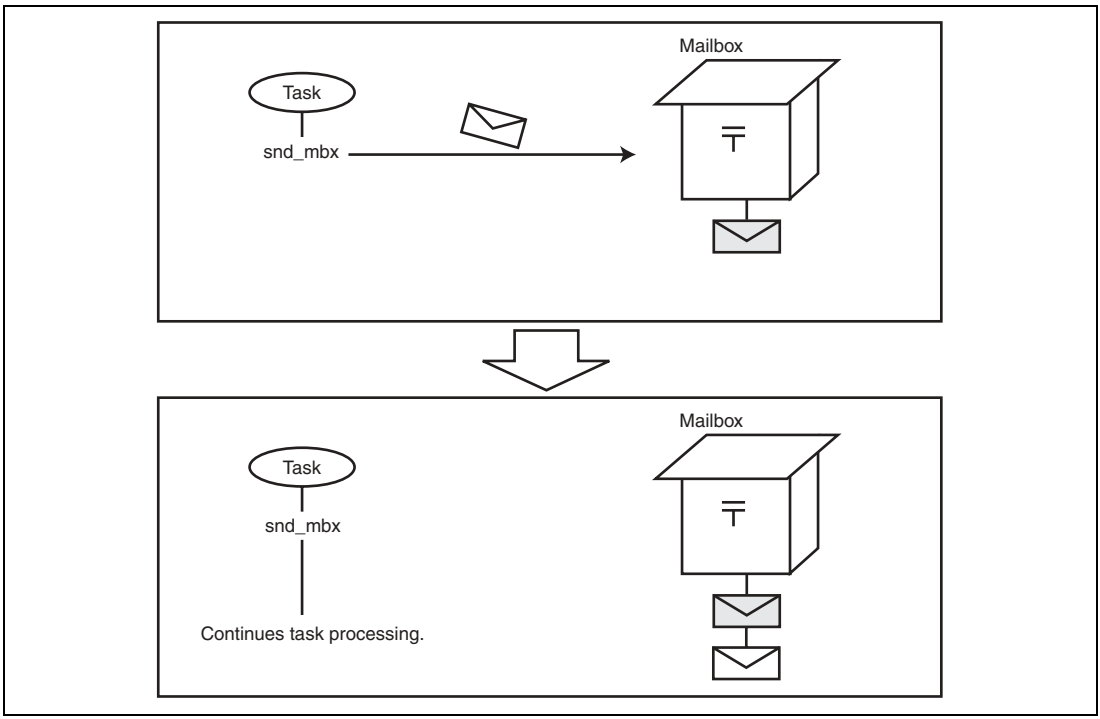


Figure 1.55 Overview of Sending a Message Using Mailbox

At the head of each message, a kernel management area must be allocated to manage the link list. This area is called a message header.

As the managing method, the FIFO (first-in first-out) method or message priority method can be selected. Accordingly, the message header format to be sent differs depending on the mailbox message managing method.

Figure 1.56 shows the message header formats for these two methods.

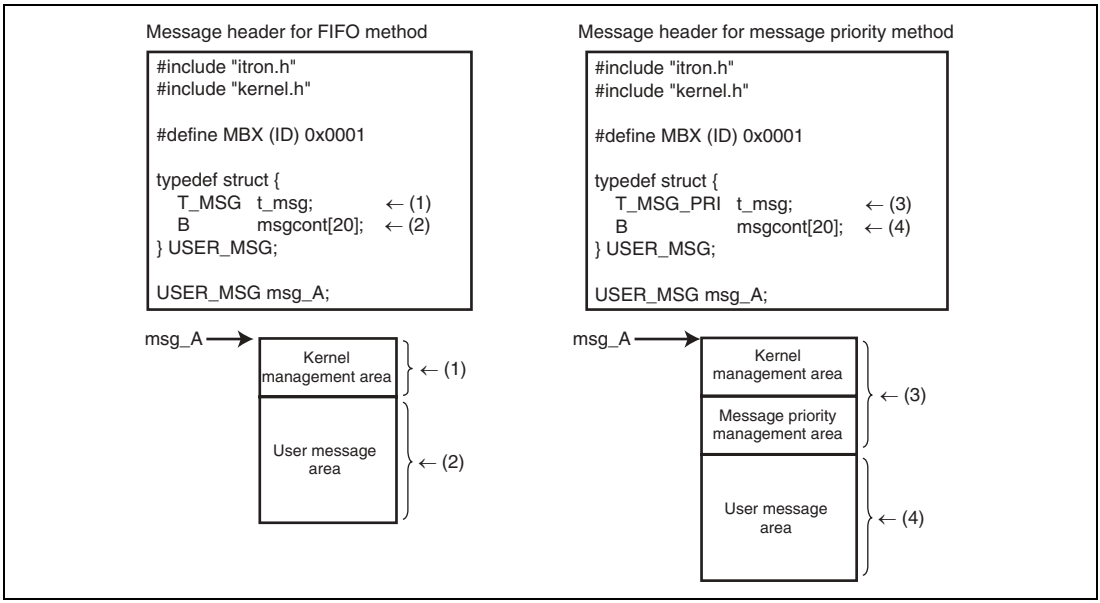


Figure 1.56 Message Header Formats

As the HI series OS cannot distinguish between these message header formats, note the combinations of the mailbox attribute and message header shown below.

Table 1.22 Combinations of Mailbox Attribute and Message Header

Message Managing Method	Message Header	
	FIFO	Message Priority
FIFO	Handled correctly.	No effect on processing but memory space is wasted.
Message priority	First 4 bytes of the user message area is handled as the priority area.*	Handled correctly.

Note: * Some messages may not be sent (an error may occur) because of the information in the first 4 bytes of the user message area in some cases.

In addition, the following notes must be observed when sending message data.

Notes when sending message data

(1) Do not modify the kernel management area after sending the message data.

(2) When sending message data for the first time, send it with the kernel management area cleared to 0. Figure 1.57 shows a sample code for sending message.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

#pragma noregsave (Task)

typedef struct user_msg{                                  }
   T_MSG  t_msg;                                          }← (1)
   B          data[10];                                   }
} USER_MSG;

// typedef struct user_primsg{                            }
//     T_MSG_PRI  t_pri_msg;                              }← (2)
//     B                 data[10];                        }
// } USER_PRIMSG;

void Task(VP_INT exinf)
{

ER ercd;
USER_MSG *message;

    (description omitted)

   ercd = get_mpf((ID)mpfid, (VP)message);                ← (3)
   if(ercd != E_OK){
      /* Error processing */
   }

   /* User message storing processing */

   message->t_msg.msghead = 0;                            ← (4)
//  message->t_pri_msg.msghead = 0;                       ← (5)

   ercd = snd_mbx((ID)mbxid, (T_MSG *)message);           ← (6)
   if(ercd != E_OK){
      /* Error processing */
   }

    (description omitted)

   ext_tsk();
}
```

Figure 1.57 Sample Code for Sending Message

1. Declares a user message (message header for FIFO management).
2. Declares a user message (message header for message priority management).
3. Allocates a memory area for the message.
4. Clears the kernel management area in the message to 0 (for FIFO management).
5. Clears the kernel management area in the message to 0 (for message priority management).
6. Sends the message.

1.9.3 Overview of Receiving a Message Using Mailbox

The following gives an overview of receiving a message using a mailbox.

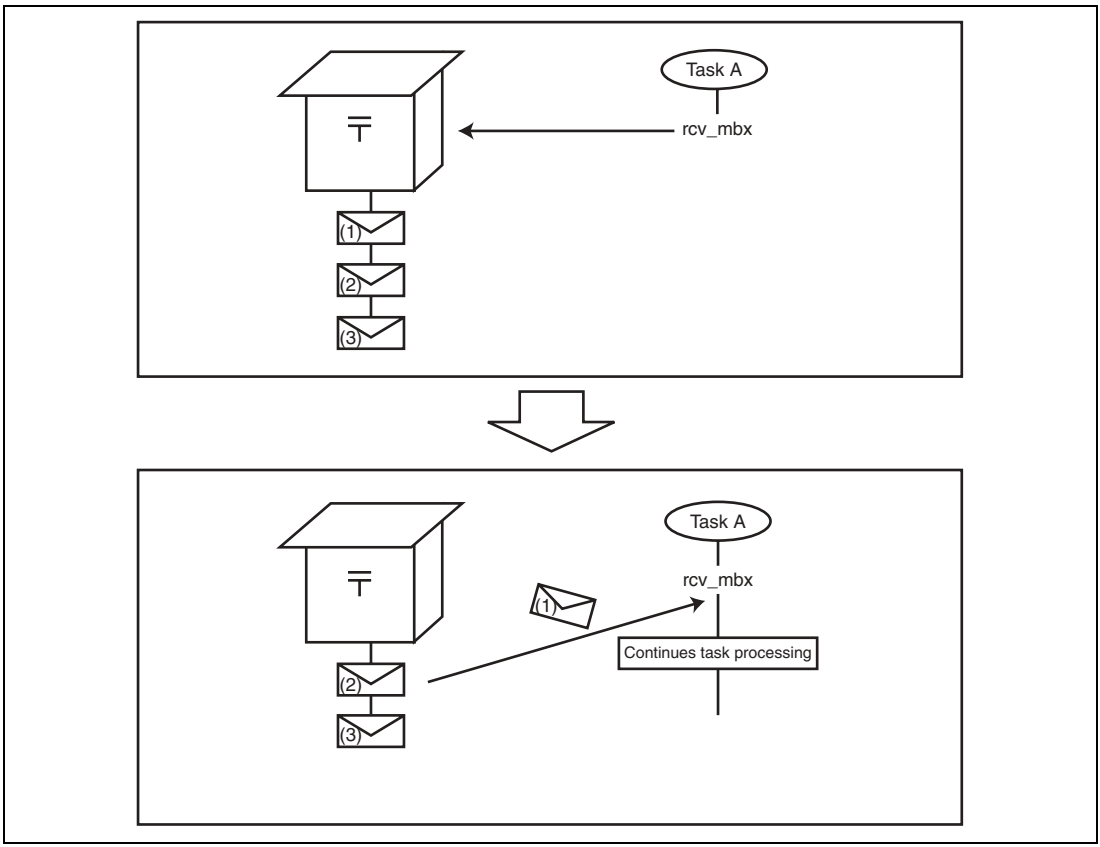


Figure 1.58 Overview of Receiving Message for Mailbox with Messages

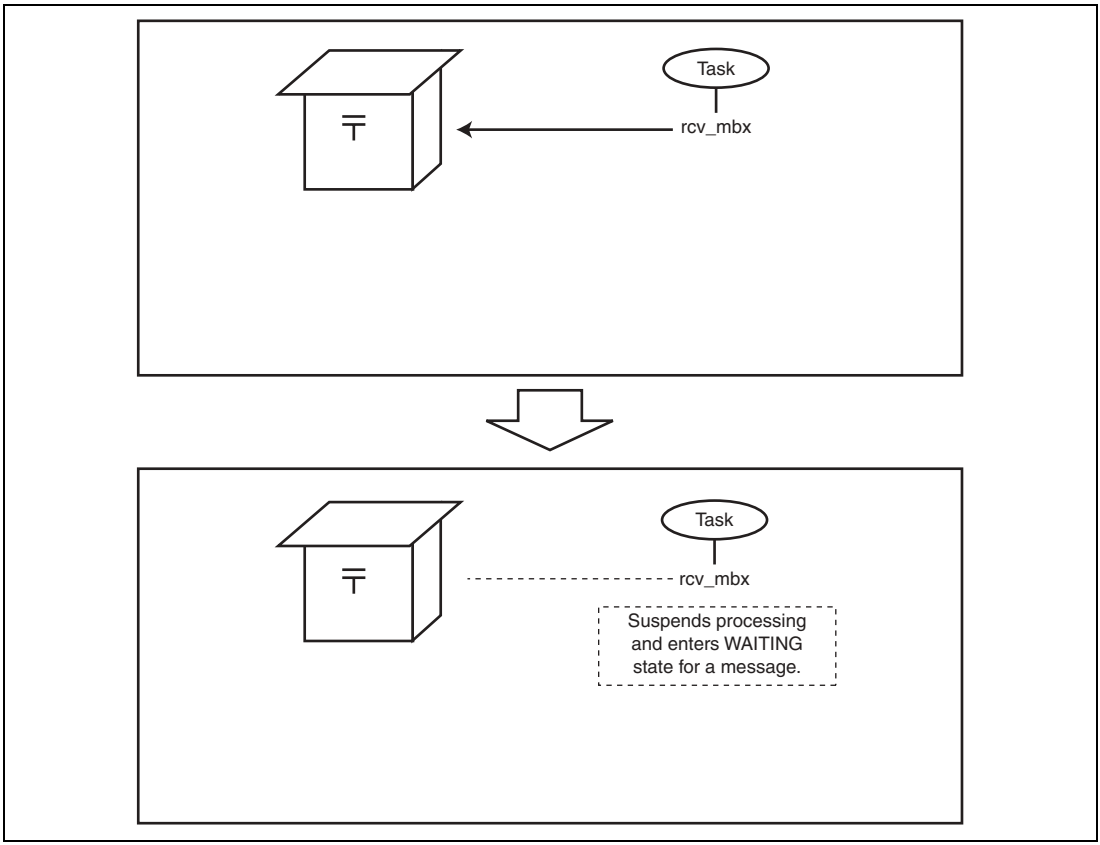


Figure 1.59 Overview of Receiving Message for Mailbox with No Messages

1.9.4 FAQ about Mailbox

This section answers a question about mailbox which is frequently asked by users of the HI series OS.

FAQ Contents:

(1) Sequential Transfer to Mailbox

Classification: Mailbox

Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4

Is it possible to send the same message sequentially to a mailbox?

Answer

The same message must not be sequentially sent to a mailbox. If attempted while the sent message has not yet been received, the message management information(link list) of the mailbox will be damaged.

The same message can be sent again only after confirming that the previously sent message has been received by the target task. Figure 1.60 shows this procedure.

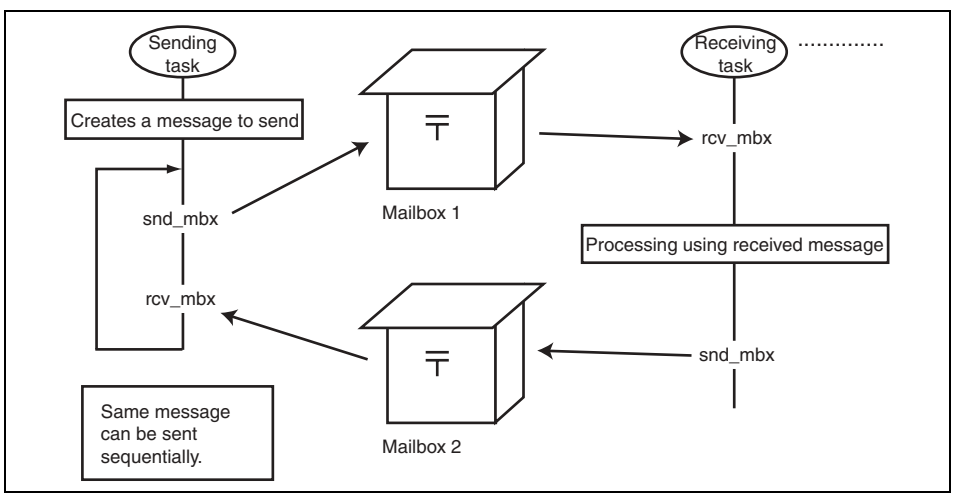


Figure 1.60 Example of Checking that Message is Received

1.10 Message Buffer

1.10.1 Overview of Message Buffer Processing

Table 1.23 summarizes the advantages and disadvantages of using message buffers.

Table 1.23 Advantages and Disadvantages of Using Message Buffers

Advantages	Disadvantages
No shared memory (nor shared address space) is required	Large overhead because a message itself is sent.

Figure 1.61 gives an overview of message buffer processing.

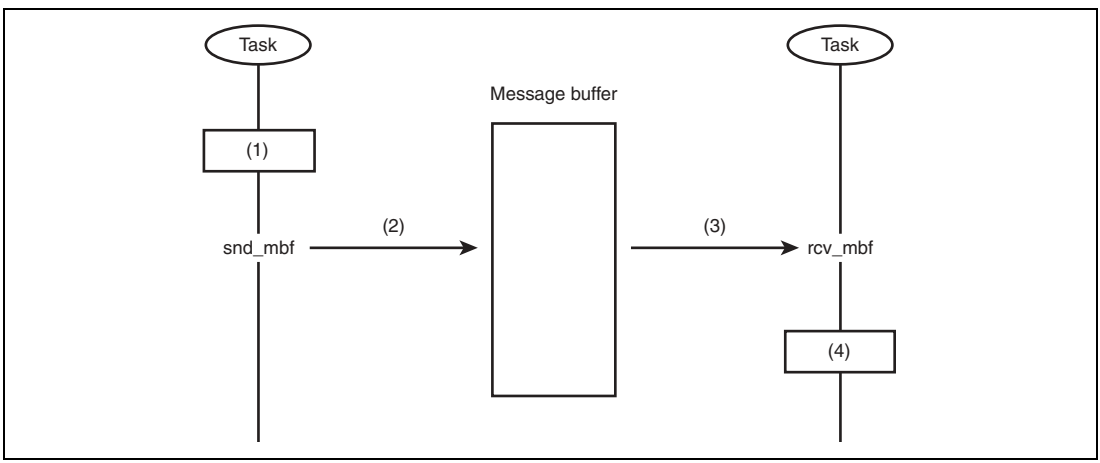


Figure 1.61 Overview of Message Buffer Processing

1. Allocate a memory area where a message is to be stored, and write a message in that area.
2. Issue a snd_mbf service call to send the message to the message buffer.
3. Issue a rcv_mbf service call to receive the message from the message buffer.
4. Read the received information.

In the HI series OS, a message buffer with buffer size = 0 can be created. Note the following in this case.

- No message can be stored in a message buffer with buffer size = 0, and the receiving task completely synchronizes with the sending task.
- A message is copied from the sending task to the receiving task at one time, which can reduce the copying steps through the message buffer.

1.10.2 Overview of Sending a Message Using Message Buffer

The message buffer processing differs depending on the sufficiency of free space in the message buffer as follows.

Table 1.24 Message Sending Processing Depending on Free Space in Message Buffer

Free Space Found in Message Buffer	Insufficient Free Space in Message Buffer
A sent message is stored in the message buffer and the sending task continues processing.	The sending task is placed in the WAITING state for message sending until sufficient space to store the sent message is created in the message buffer.

Figure 1.62 gives an overview of sending a message when the message buffer has sufficient free space.

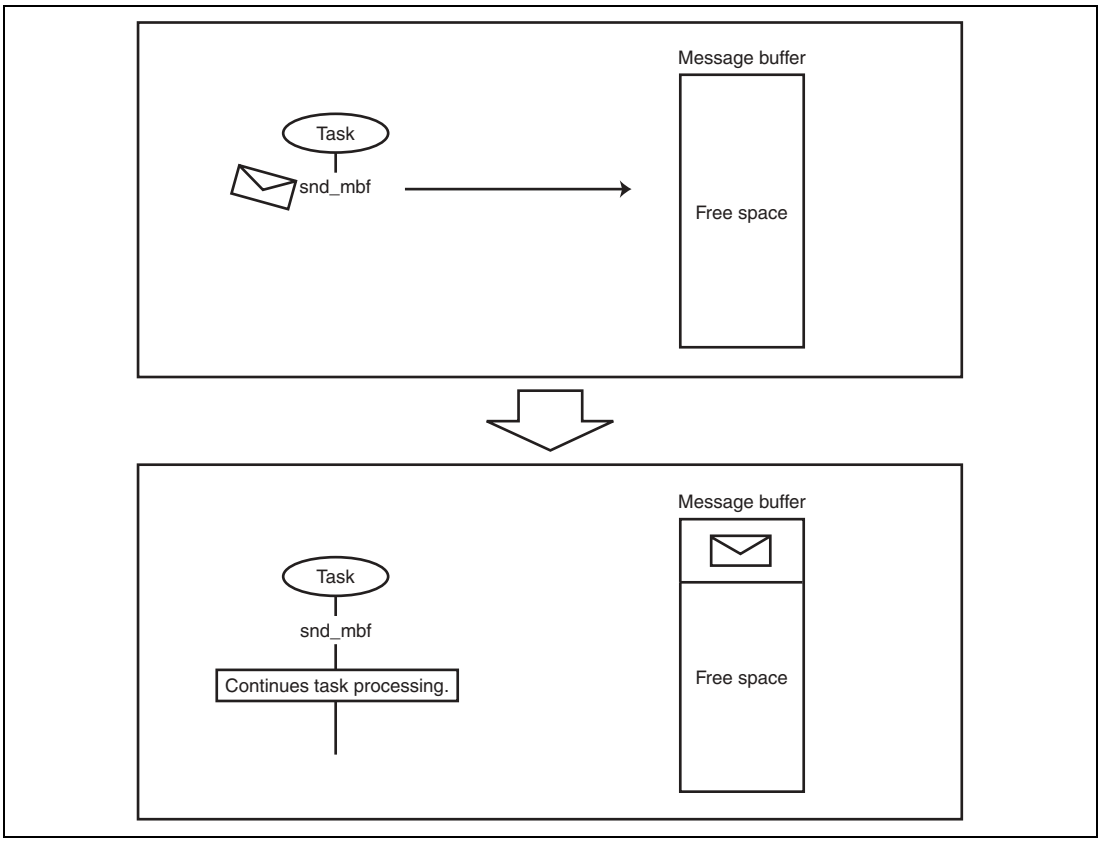


Figure 1.62 Overview of Sending a Message for Message Buffer with Enough Free Space

Figure 1.63 gives an overview of sending a message when the message buffer does not have sufficient free space.

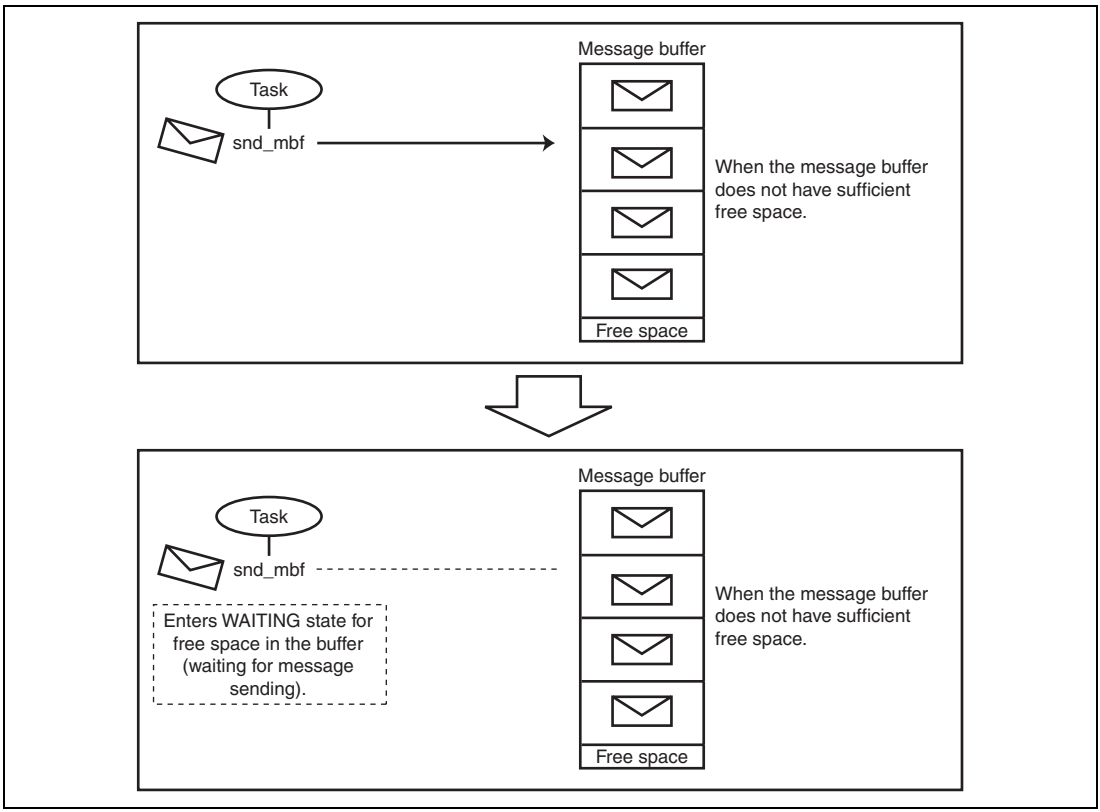


Figure 1.63 Overview of Sending a Message for Message Buffer with Insufficient Free Space

1.10.3 Overview of Receiving a Message Using Message Buffer

The following gives an overview of receiving a message using a message buffer.

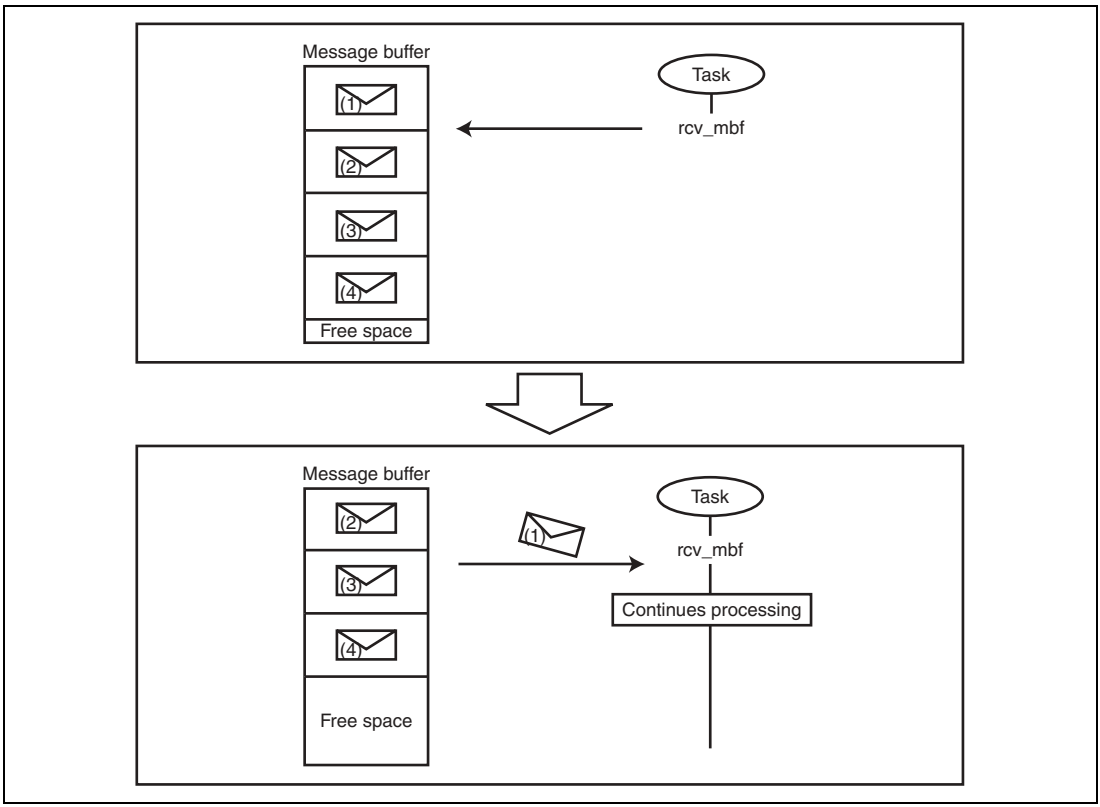


Figure 1.64 Overview of Receiving Message for Message Buffer with Messages

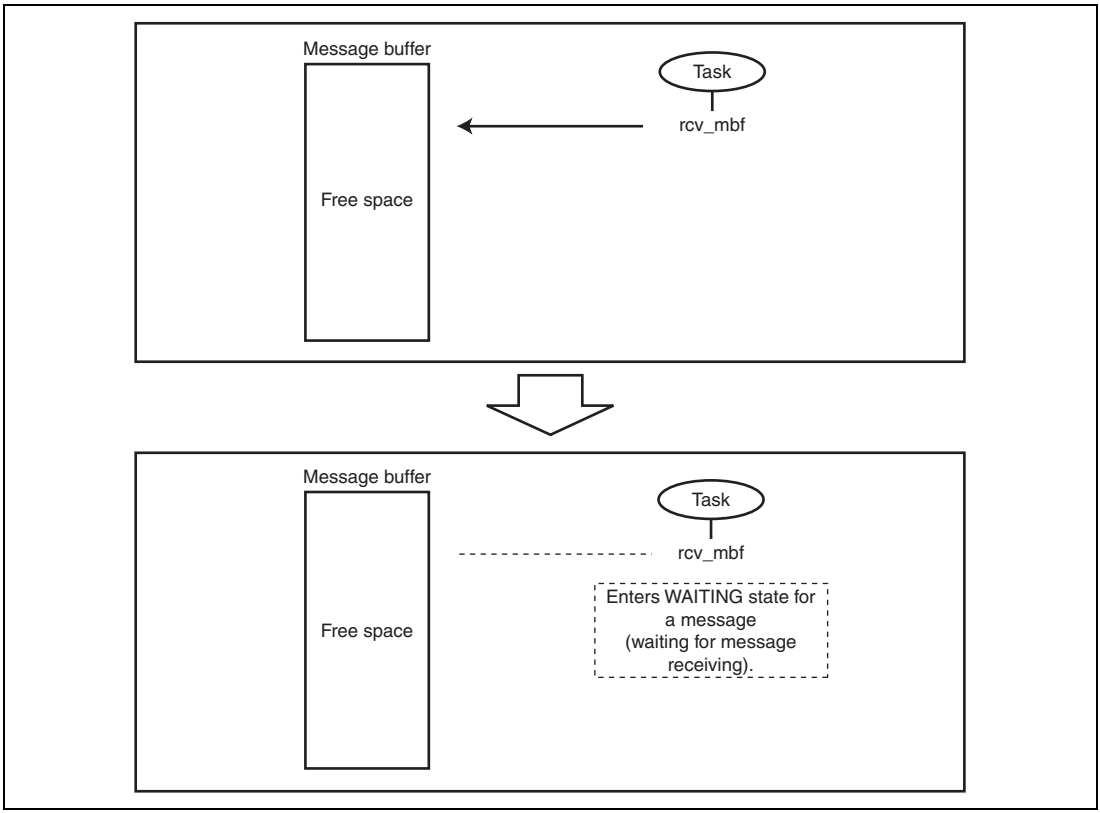


Figure 1.65 Overview of Receiving Message for Message Buffer with No Messages

1.11 Data Queue

1.11.1 Overview of Data Queue Processing

Table 1.25 summarizes the advantages and disadvantages of using data queues.

Table 1.25 Advantages and Disadvantages of Using Data Queues

Advantages	Disadvantages
• No shared memory (nor shared address space) is required • A message itself is copied, but its size is fixed at 4 bytes (the overhead is small).	A large amount of message cannot be sent because the message size is fixed.

Figure 1.66 gives an overview of data queue processing.

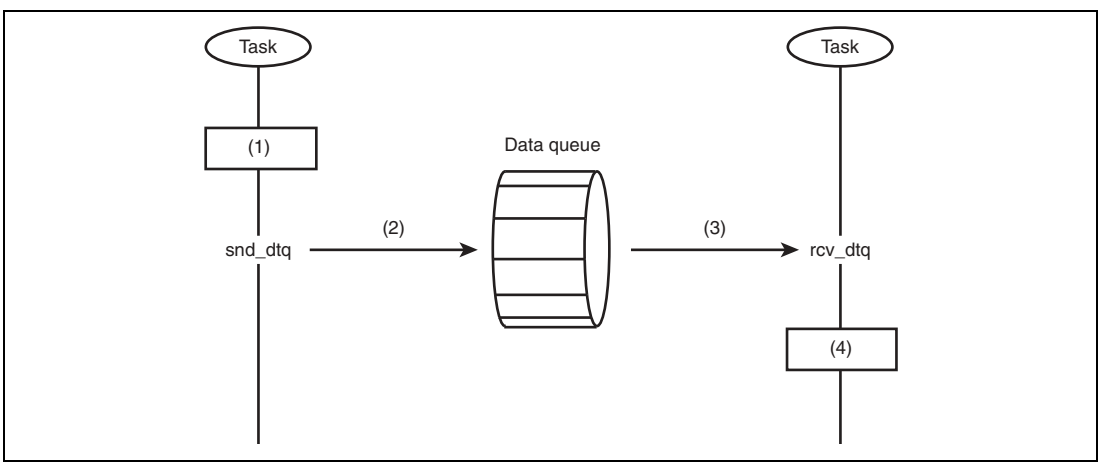


Figure 1.66 Overview of Data Queue Processing

1. Allocate a memory area where a message is to be stored, and write a message in that area.
2. Issue a snd_dtq service call to send the message to the data queue.
3. Issue a rcv_dtq service call to receive the message from the data queue.
4. Read the received information.

1.11.2 Overview of Sending a Message Using Data Queue

The data queue processing differs depending on the sufficiency of free space in the data queue as follows.

Table 1.26 Message Sending Processing Depending on Free Space in Data Queue

Free Space Found in Data Queue	Insufficient Free Space in Data Queue
A sent message is stored in the data queue and the sending task continues processing.	The sending task is placed in the WAITING state for message sending until sufficient space to store the sent message is created in the data queue.

Figure 1.67 gives an overview of sending message when the data queue has free space.

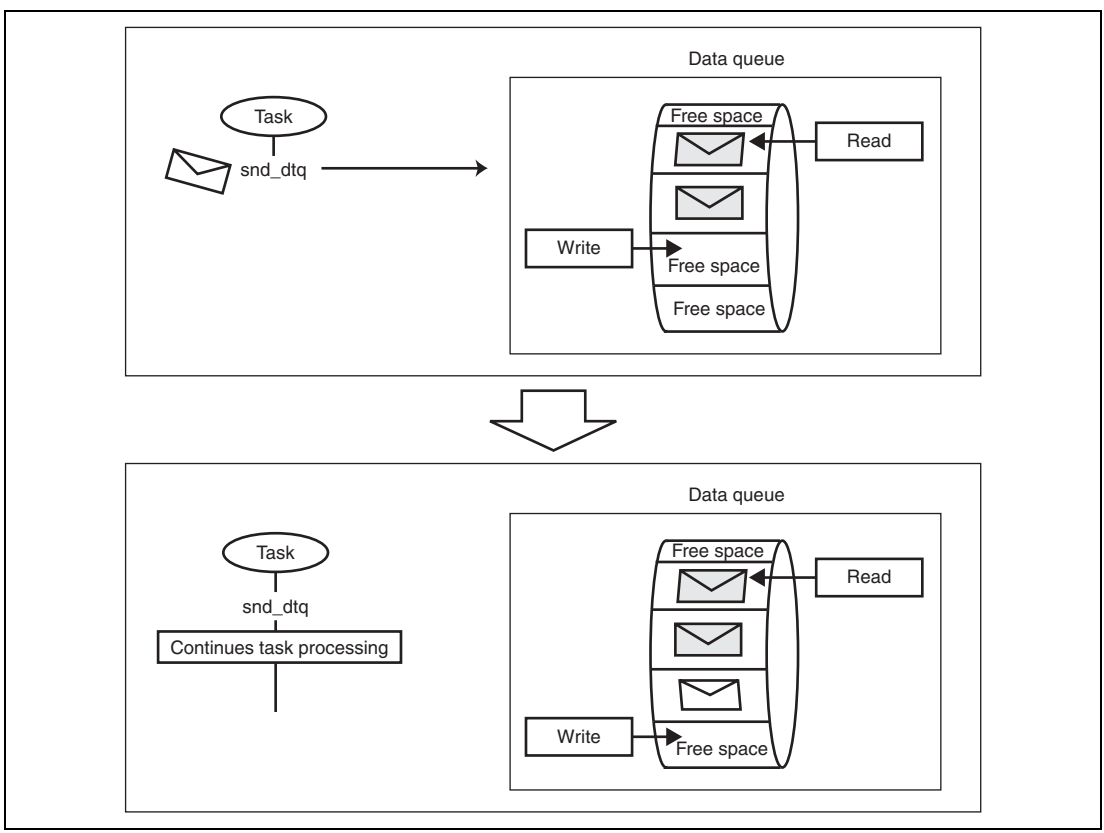


Figure 1.67 Overview of Sending a Message for Data Queue with Enough Free Space

Figure 1.68 gives an overview of sending message when the data queue does not have sufficient free space.

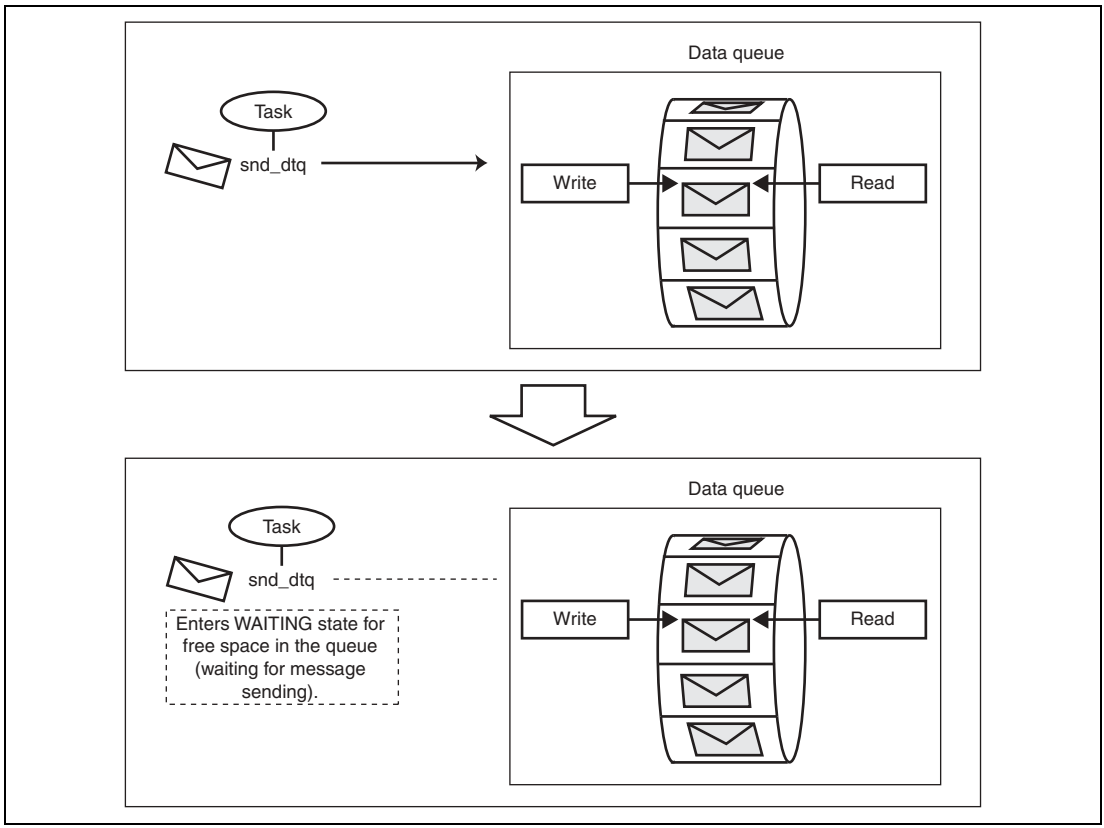


Figure 1.68 Overview of Sending a Message for Data Queue with Insufficient Free Space

The data queue has the forcible send function.

The forcible send function overwrites the oldest data in the data queue with the sent data when the data queue area does not have sufficient free space to store the sent message data. Note that the overwritten data is managed as the latest data, and thus is read last.

Figure 1.69 gives an overview of the forcible send processing.

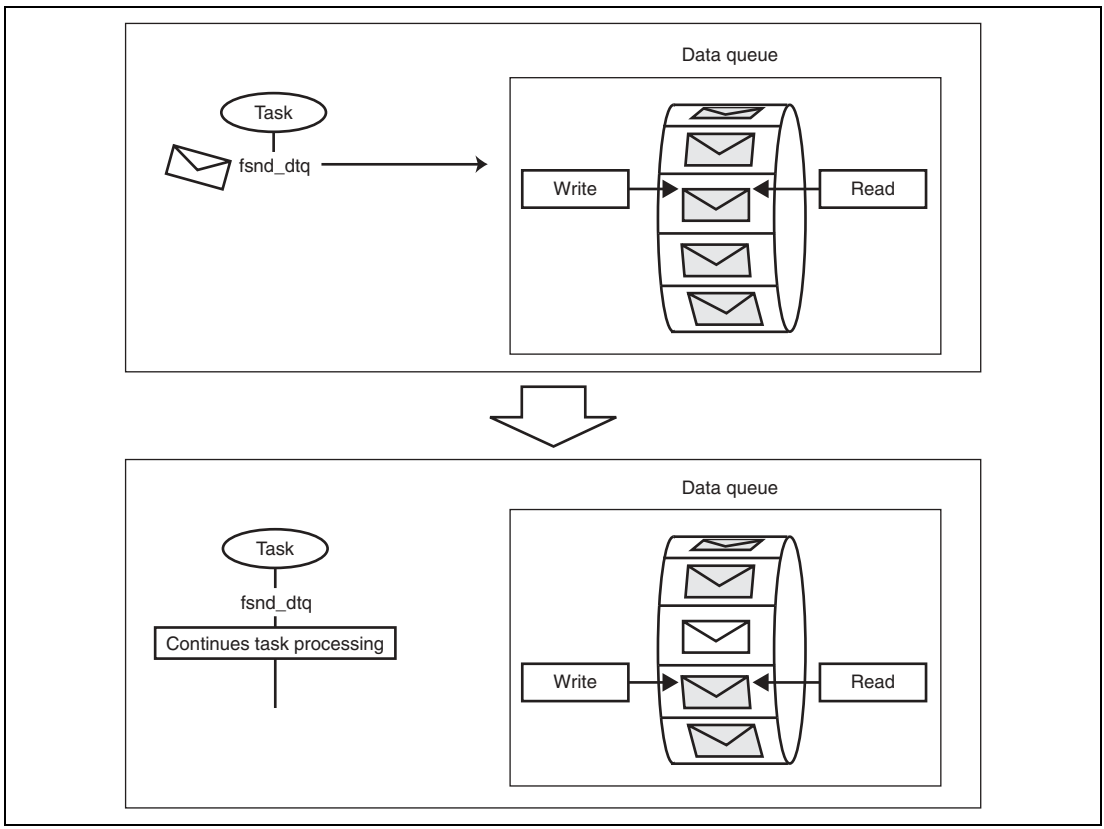


Figure 1.69 Overview of Forcible Send Processing by Data Queue

1.11.3 Overview of Receiving a Message Using Data Queue

The following gives an overview of receiving a message using a data queue.

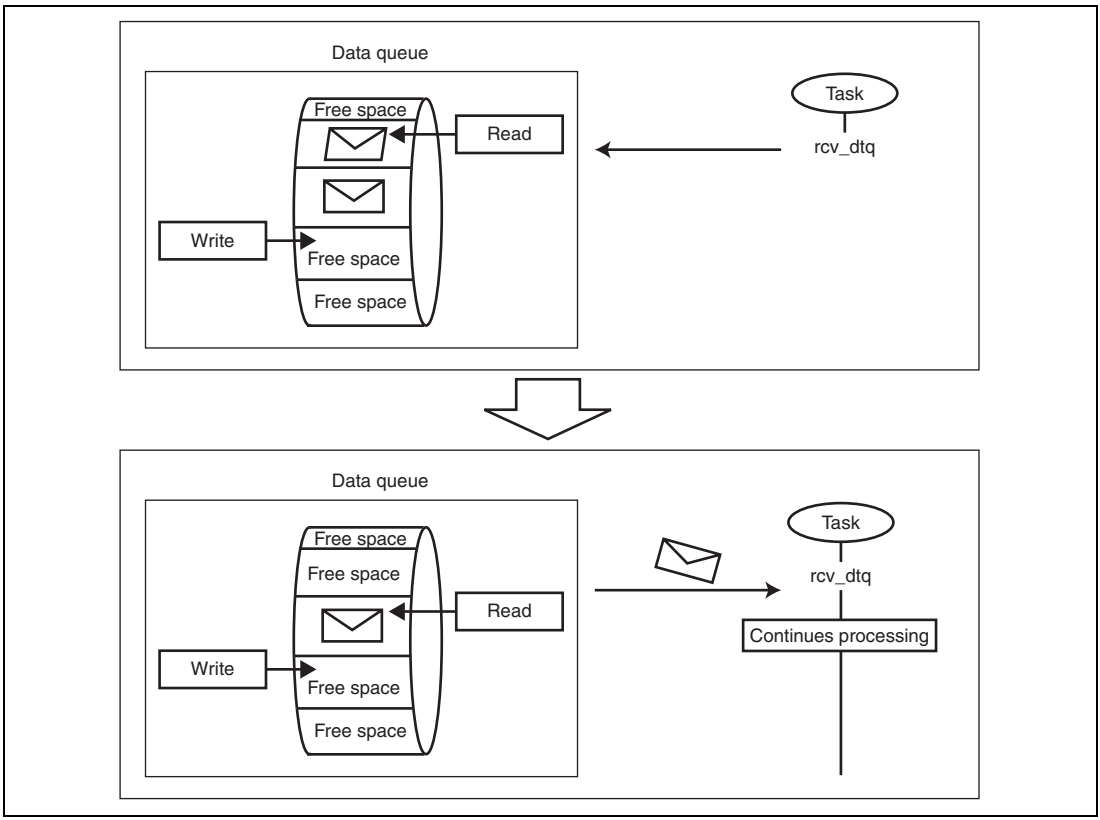


Figure 1.70 Overview of Receiving Message for Data Queue with Messages

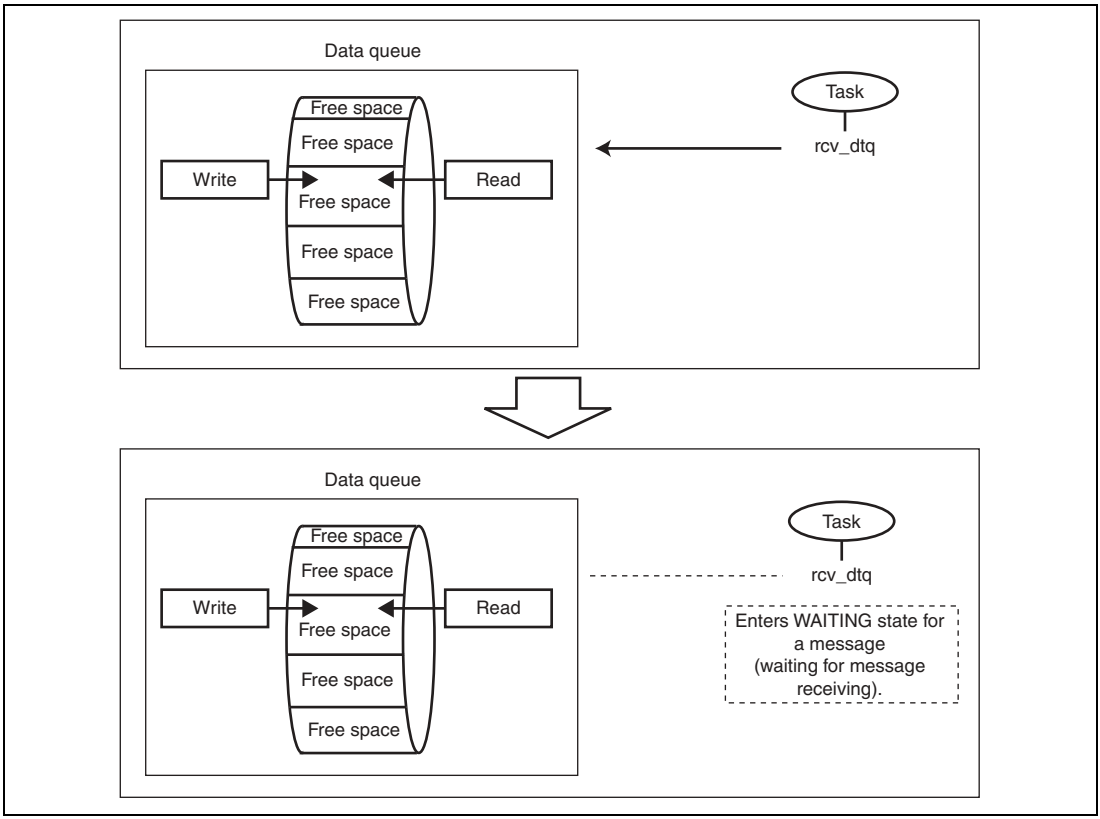


Figure 1.71 Overview of Receiving Message for Data Queue with No Messages

1.12 Memory Pool

1.12.1 Fragmentation

Fragmentation means that the used area in memory is divided into small non-contiguous pieces. Figure 1.72 gives an overview of fragmentation.

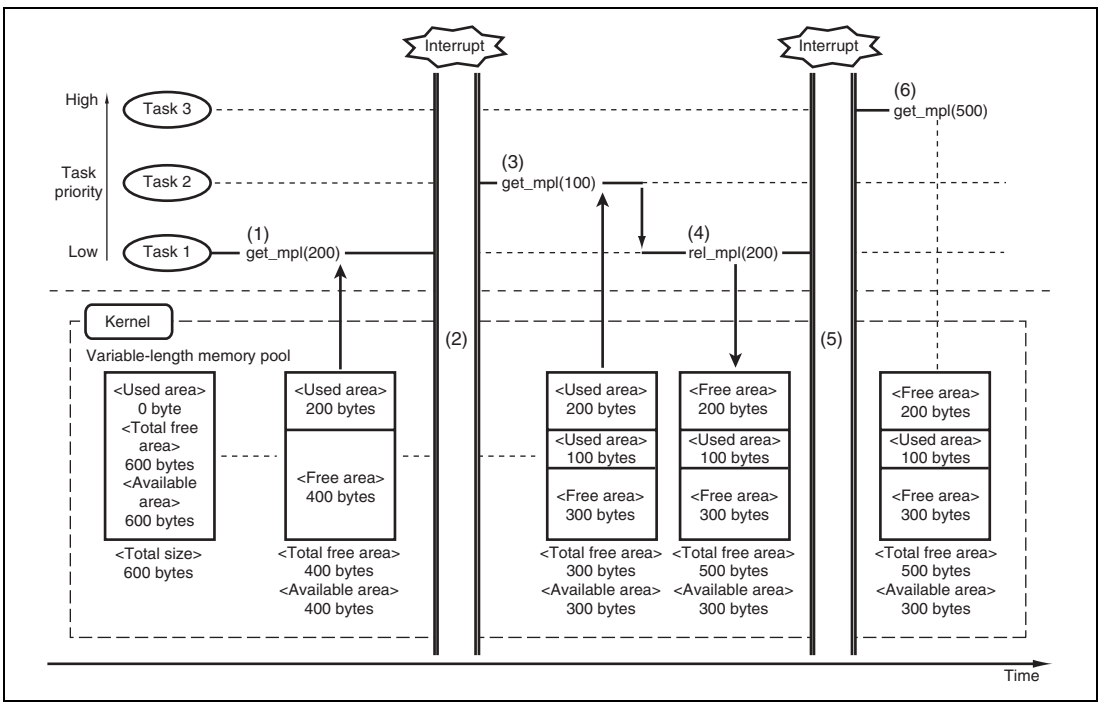


Figure 1.72 Overview of Fragmentation

1. Task 1 requests and obtains a 200-byte area.
2. An interrupt occurs and the interrupt handler switches tasks (from task 1 to task 2).
3. Task 2 requests and obtains a 100-byte area. (The processing using the obtained memory area switches tasks (from task 2 to task 1).)
4. Task 1 returns the previously obtained 200-byte area.
5. An interrupt occurs and the interrupt handler switches tasks (from task 1 to task 3).
6. Task 3 requests a 500-byte area, but enters the WAITING state for a free area because the maximum contiguous free area is 300 bytes even though the total free area is 500 bytes.

Such a condition, as shown above, is called fragmentation.

The HI series OS does not provide garbage collection, which solves fragmentation problems.

Fragmentation of the memory pool area must be solved through an application (user system).

1.12.2 FAQ about Memory Pool

This section answers a question about memory pool which is frequently asked by users of the HI series OS.

FAQ Contents:

(1) Use of malloc() function

Classification: Memory pool					
Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4
Is it possible to use the malloc() function in the system using the μITRON-based OS?					

Answer

The malloc() function cannot be used in the system using the μITRON-based OS.

The OS cannot recognize the area allocated by the malloc() function.

If the area allocated by the malloc() function overlaps the area allocated by the memory pool functions, data may be damaged.

Accordingly, when the system must manage memory, use the memory pool functions provided by the OS.

1.13 Time Management

1.13.1 Concept of Time Management

Table 1.27 shows the meaning of parameter tmout used in the time management function.

Table 1.27 Meaning of Parameter tmout

HI Series OS	tmout Meaning
HI7000/4 series and HI1000/4	tmout value (ms)
HI2000/3	tmout value × hardware timer cycle time

Figure 1.73 shows an example of processing when tslp_tsk(3) is executed with the hardware timer cycle specified as 1 ms (CFG_TICNUME = 1 and CFG_TICDENO = 1).

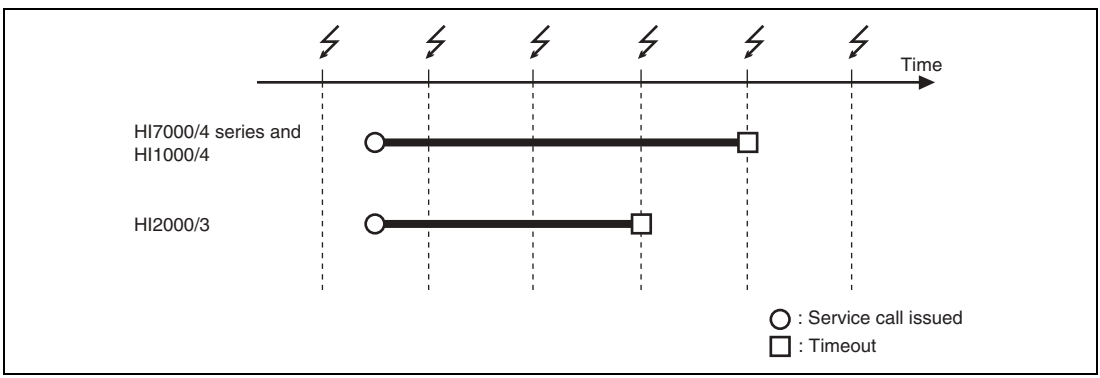


Figure 1.73 Overview of tslp_tsk(3) Processing

Table 1.28 describes the error between the tmout value and the obtained timeout period shown in the above figure.

Table 1.28 Error when Issuing tslp_tsk(3)

HI Series OS	tmout Value	Error
HI7000/4 series and HI1000/4	tmout value = 3 Wait time is 3 ms	Period after the service call is issued to register the task under the timer control and until the next time tick is supplied (X)
HI2000/3	tmout value = 3 Third hardware timer cycle	Period after a hardware timer cycle is started and until the service call is issued to register the task under the timer control (Y)

See figure 1.74 for errors (X) and (Y).

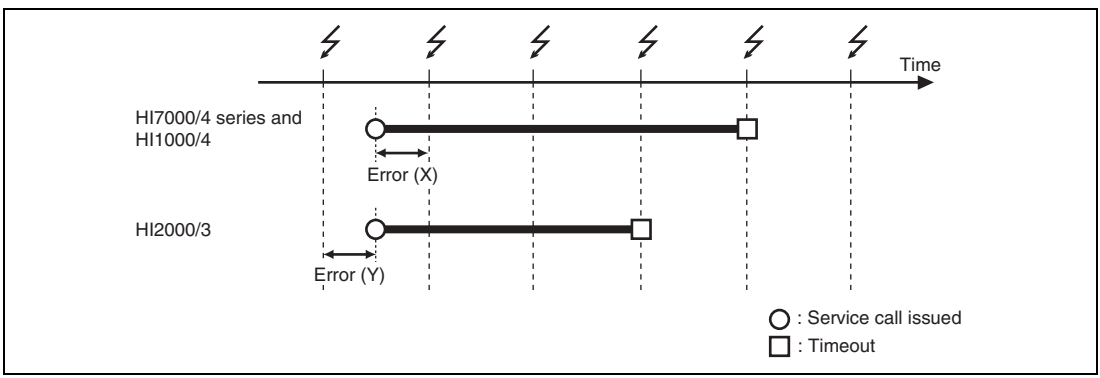


Figure 1.74 Error in tslp_tsk(3) Processing

The error affects the hardware timer cycle time. Table 1.29 shows the relationship between the hardware timer cycle time and the error.

Table 1.29 Hardware Timer Cycle Time and Error

Hardware Timer Cycle Time	Advantages	Disadvantages
When a shorter time is specified	The error in time management is reduced.	As the hardware timer cycle processing is increased, the time that can be assigned to task processing is reduced.
When a longer time is specified	As the hardware timer cycle processing is reduced, the time that can be assigned to task processing is increased.	The error in time management is increased.

1.13.2 Modification of Hardware Timer Cycle Unit

This section describes how to modify the hardware timer cycle unit by using the following means.

- HI7000/4 series and HI1000/4: Configurator
- HI2000/3: Header file for timer driver

(1) HI7000/4 Series and HI1000/4

The hardware timer cycle time (time of the time tick supplying cycle, hereinafter called the time tick cycle time) is set as 1 ms at default and can be modified through the configurator.

Figure 1.75 shows the configurator window for time management settings.

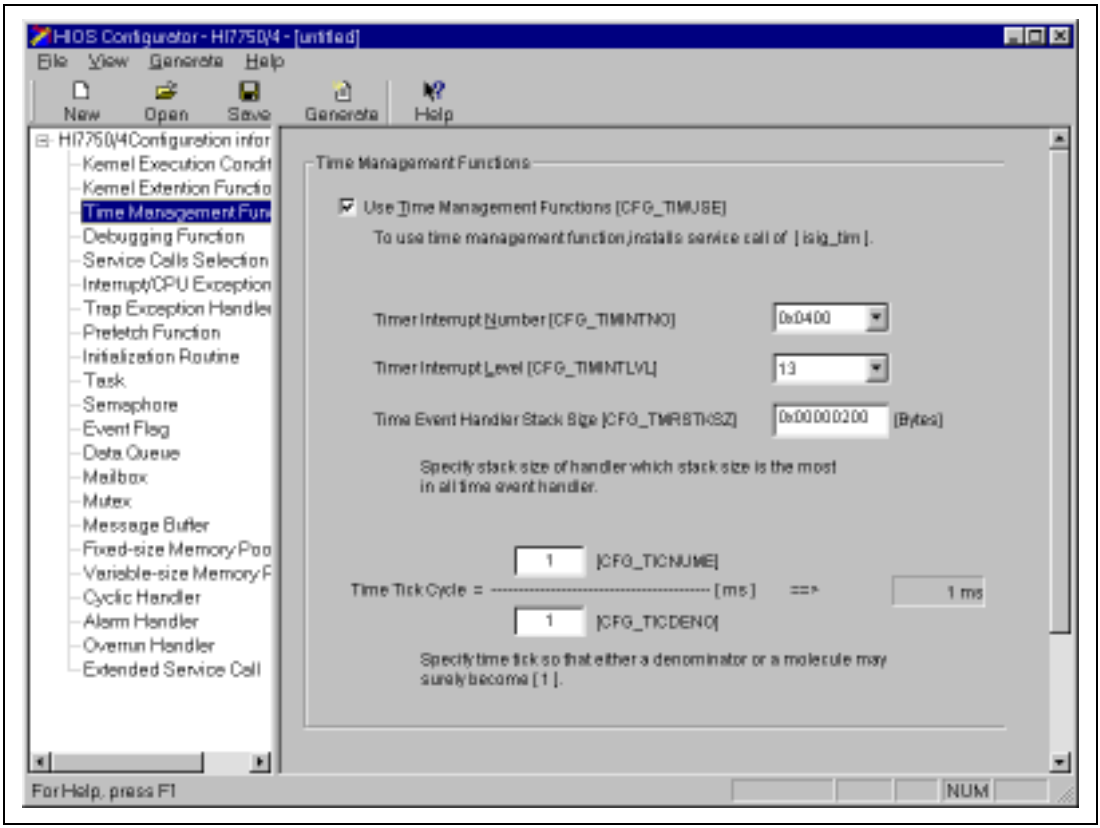


Figure 1.75 Configurator Window for Time Management Settings

As shown in the window, the time tick cycle is expressed by CFG_TICNUME (the numerator of the time tick cycle) and CFG_TICDENO (the denominator of the time tick cycle) in the following expression.

$$\boxed{\text{Time tick cycle}} = \left\{ 1 \times \left[\frac{\text{TIC\_NUME (numerator of time tick cycle)}}{\text{TIC\_DENO (denominator of time tick cycle)}} \right] \right\}$$

Figure 1.76 Calculation of Time Tick Cycle

This setting controls the time tick cycle (1 ms at default) so that it can be longer or shorter. In the default settings, the 1-ms time tick cycle base is defined as divided into 1/1, that is, the parameters are specified as CFG_TICNUME = 1 and CFG_TICDENO = 1. The default time tick cycle is used for time management of the whole system.

CFG_TICNUME and CFG_TICDENO can be set to the following values.

- CFG_TICNUME (numerator of the time tick cycle): 1 to 65,535
- CFG_TICDENO (denominator of the time tick cycle): 1 to 100

Accordingly, the 1-ms time tick cycle can be modified into a minimum of 0.01 ms (10 μs: CFG_TICNUME = 1 and CFG_TICDENO = 100, that is, 1/100) and a maximum of 65,535 ms (65 s: CFG_TICNUME = 65,535 and CFG_TICDENO = 1, that is, 65,535/1).

(2) HI2000/3

The hardware timer cycle time is set as 1 ms in the standard sample program and can be modified in the definition in the header file for the timer driver.

Figure 1.77 shows the header file for the timer driver in the standard sample program.

```
;*****************************************************************************
;* specifications ;                                                          *
;* name        = _HIPRG_TIMINI : H8S/2655 TPU0 initialize handler  ;         *
;* function    =                                                   ;         *
;* notes       =                                                   ;         *
;* date        = 99/02/22                                          ;         *
;* author      = Hitachi, Ltd.                                     ;         *
;* attribute   = public                                            ;         *
;* class       = system                                            ;         *
;* linkage     =                                                   ;         *
;* input       = ccr(B): interrupt disable                         ;         *
;*             = exr(B): interrupt disable                         ;         *
;* output      = all register unchanged                            ;         *
;* end of specifications ;                                                   *
;*****************************************************************************

   (description omitted)

;###### setting data ##################;:
;
TGRA_DATA:     .assign h'30d3;: d'12500-1;: (10000us (p/16))-1: 10ms = 10000us, p = 20MHz
```

Hardware timer cycle time

Figure 1.77 Header File for Timer Driver in Standard Sample Program (2655ause.src)

An example of the hardware timer cycle time calculation is shown below.

Reference: Calculation of Hardware Timer Cycle Time

This example describes how to obtain the 10-ms hardware timer cycle time when the H8S/2655 (whose operating frequency is 20 MHz) is used in the HI2000/3.

The hardware timer cycle time (T) is determined by the counter clock cycle time (t) and counter value (n) as follows.

$T = \{t \times (n + 1)\}$

Value t is determined by the counter clock (ϕ/1, ϕ/4, ϕ/16, or ϕ/64) selected in the timer control register (TCR).

When the CPU clock (ϕ) is 20 MHz, value t becomes as follows.

- Counter clock = ϕ/1: t = 50 ns
- Counter clock = ϕ/4: t = 200 ns
- Counter clock = ϕ/16: t = 800 ns
- Counter clock = ϕ/64: t = 3.2 μs

Value n is determined by setting a value from 0x0000 to 0xFFFF in output compare match A (TGRA). Accordingly, when the CPU clock (ϕ) is 20 MHz, value T falls within the following ranges.

- Counter clock = ϕ/1: T = 50 ns to 3.27 ms
- Counter clock = ϕ/4: T = 200 ns to 13.1 ms
- Counter clock = ϕ/16: T = 800 ns to 52.4 ms
- Counter clock = ϕ/64: T = 3.2 μs to 209.7 ms

[Calculation of 10-ms cycle]

Output compare match A (TGRA) = Timer cycle time (s) × n – 1

In the above formula, timer cycle time (s) = 10×10^{-3} to specify a 10-ms timer cycle time. When the CPU clock (ϕ) is 20 MHz and ϕ/16 is selected as the counter clock, value n is obtained as follows.

$n = 20 \times 10^6 \div 16$

Accordingly, output compare match (TGRA) becomes as follows:

Output compare match A (TGRA) = Timer cycle time (s) × n – 1
$= (10 \times 10^{-3}) \times (20 \times 10^6 \div 16) - 1$
= 12,499 (0x30D3)

To obtain a 10-ms timer cycle time (s) when using 20-MHz CPU clock (ϕ), the value set to output compare match A (TGRA) should be 12,499 (0x30D3).

1.13.3 Cyclic Handler

(1) HI7000/4 Series and HI1000/4

Figure 1.78 shows an example of cyclic handler initiation when the initiation phase is 2 ms, the initiation cycle is 3 ms, and the hardware timer cycle is set to 1 ms (CFG_TICNUME = 1 and CFG_TICDENO = 1).

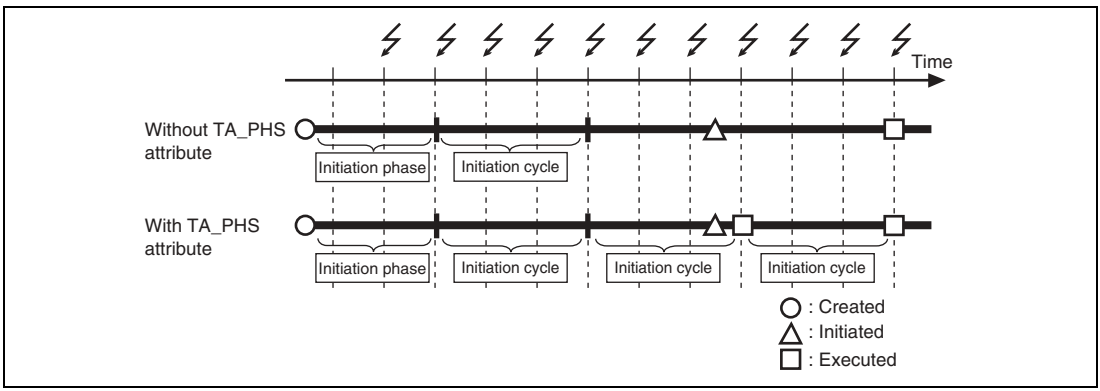


Figure 1.78 Overview of Cyclic Handler Initiation (HI7000/4 Series and HI1000/4)

(2) HI2000/3

Figure 1.79 shows an example of cyclic handler initiation when the cyclic initiation interval is 3 ms and the hardware timer cycle is set to 1 ms.

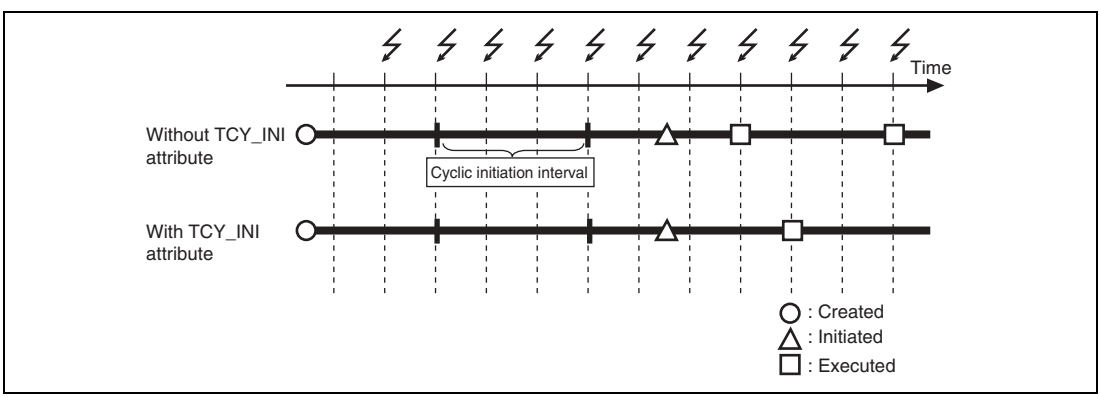


Figure 1.79 Overview of Cyclic Handler Initiation (HI2000/3)

1.13.4 Overview of Timer Management Processing

The following gives an overview of timer management processing.

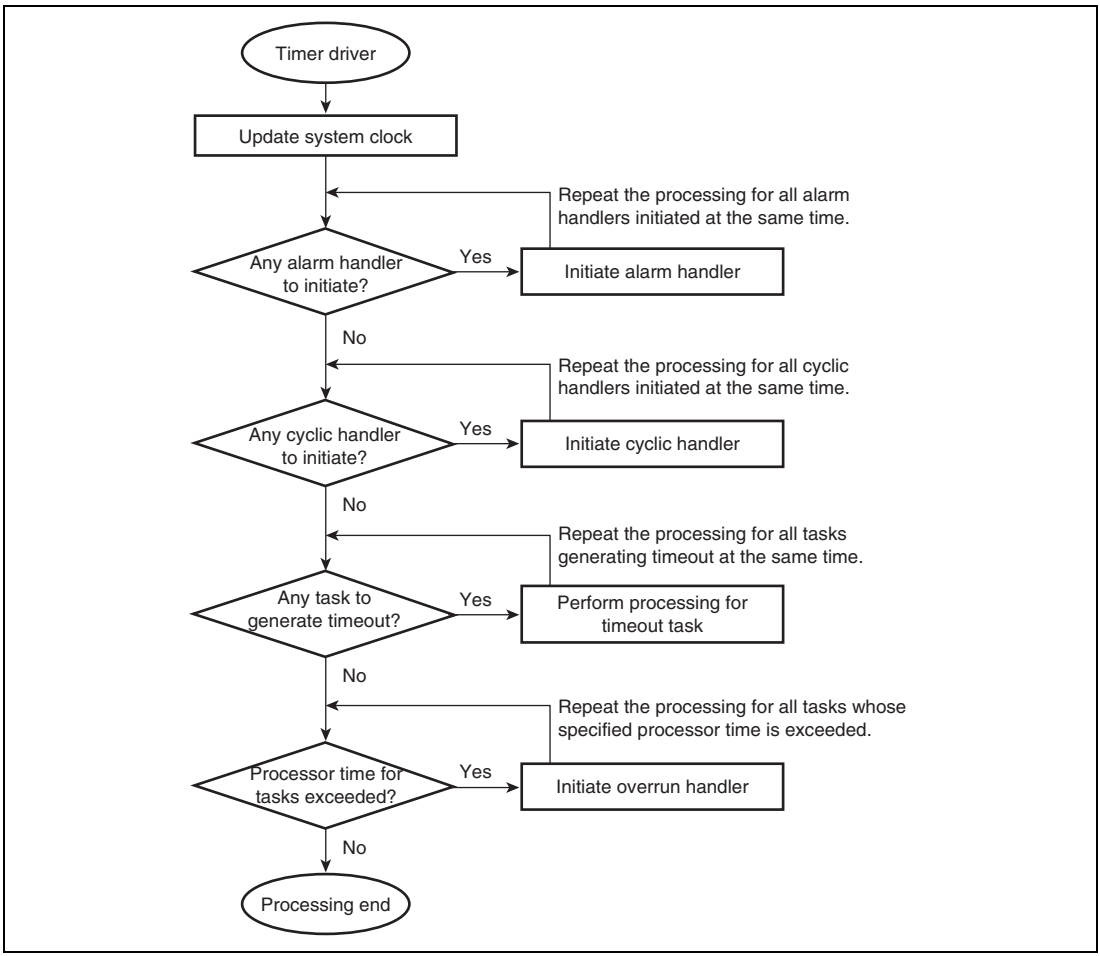


Figure 1.80 Overview of Timer Driver Processing (HI7000/4 Series)

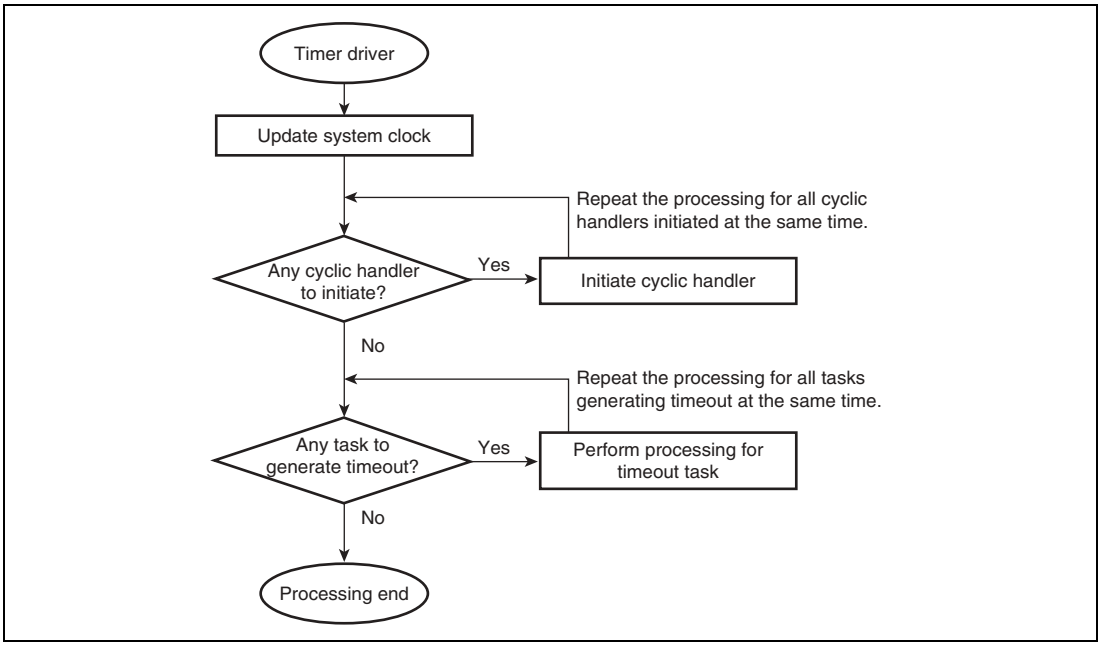


Figure 1.81 Overview of Timer Driver Processing (HI2000/3 and HI1000/4)

The following items also affect the time for timer driver processing.

- Number of alarm handlers to be initiated at the same time (only for HI7000/4 series)
- Number of cyclic handlers to be initiated at the same time
- Number of tasks to generate timeout at the same time
- Number of tasks to initiate overrun handler at the same time (only for HI7000/4 series)

If the number of tasks to generate timeout or the number of handlers (cyclic handlers and alarm handlers) to be initiated at the same time becomes large, the corresponding service processing should be repeated more times, which will result in increased timer driver processing time. If the timer driver processing time is increased, the following problems will arise.

- Degradation in response to other interrupts
- Delay in system time

Section 2 Application Program Creation

2.1 Overview of Processing from Reset to Task Initiation

Figure 2.1 gives an overview of the processing after a CPU reset (including a power-on reset) and until task initiation in the HI series OS.

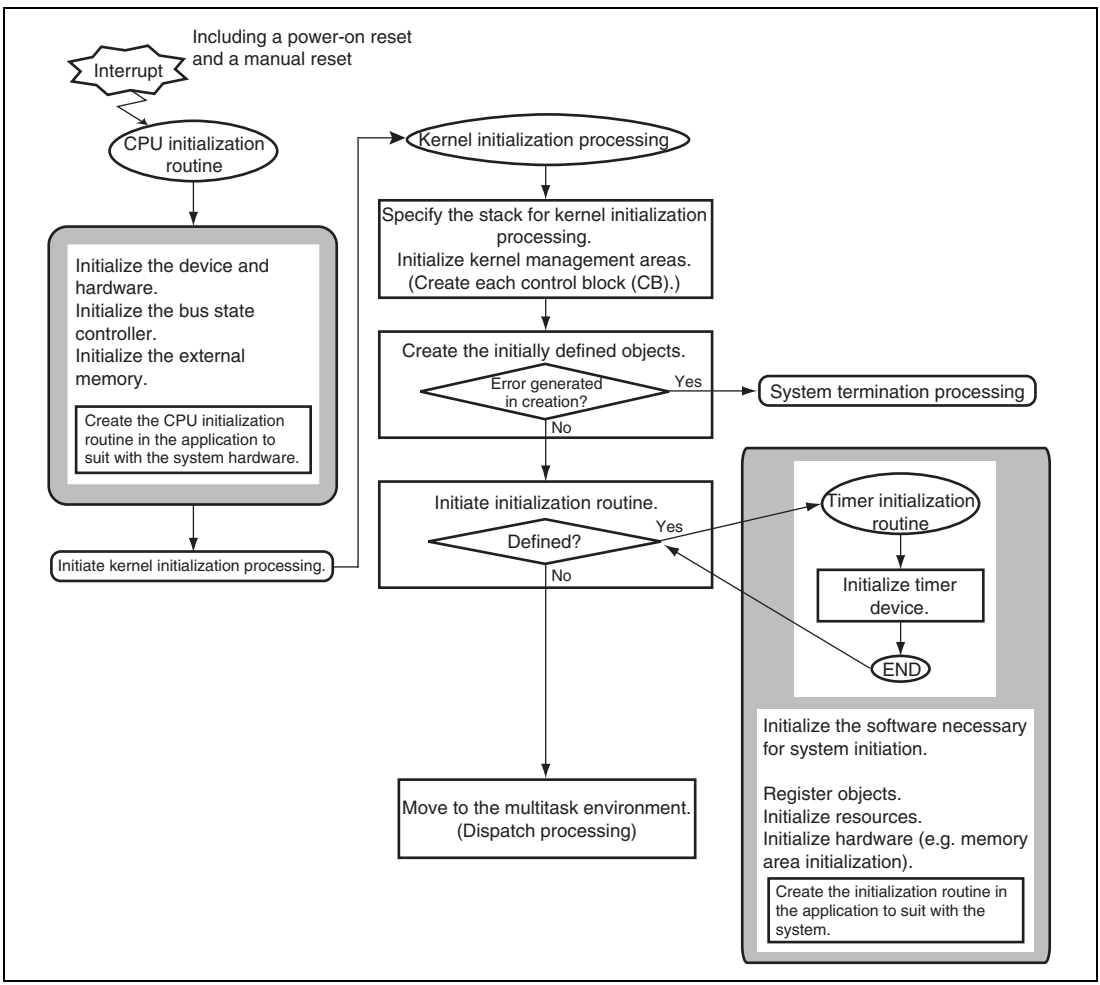


Figure 2.1 Procedure after CPU Reset and Until Task Initiation

When a CPU reset signal is input, the CPU initialization routine defined at the reset vector is initiated.

2.2 Overview of CPU Initialization Routine

The CPU initialization routine carries out the processing needed for the entire software, including the kernel, to operate. To be more specific, the CPU initialization routine includes the following processing.

- Sets up the bus state controller (BSC) to enable external memory (such as SDRAM or SRAM).
- Specifies the stack pointer for the CPU initialization routine.
- Initializes the sections.

The CPU initialization routine carries out the initialization necessary for the microcomputer and hardware used, and thus the CPU initialization routine must be created in the application in accordance with the microcomputer and hardware.

The CPU initialization routine cannot be written entirely in C language; part of it must be written in assembly language.

A C program accesses the stack (memory). If the stack area is accessed before the necessary settings are completed, a CPU exception may occur (a CPU exception causes abnormal system termination (system down)). Accordingly, the CPU initialization routine must be written in assembly language until the stack settings are completed.

The HI series OS provides sample files of the CPU initialization routine. Refer to it and create the CPU initialization routine in accordance with the hardware and microcomputer used.

Table 2.1 summarizes the sample CPU initialization routine.

Table 2.1 Overview of CPU Initialization Routine Processing

	CPU Initialization Routine	
HI Series OS	**Assembly-Language Descriptions**	**C-Language Descriptions**
HI7000/4 series	• BSC settings to enable external memory (such as SDRAM or SRAM) • Settings of stack pointer	• Initialization of sections • Enabling of cache
HI2000/3	• Settings of stack pointer • Settings of interrupt control mode • Settings of peripheral modules	See note below.
HI1000/4	• Settings of stack pointer • BSC settings to enable external memory (such as SDRAM or SRAM) • Settings of interrupt control mode • Settings of peripheral modules	See note below.

Note The HI2000/3 and HI1000/4 do not provide a C-language sample file of the CPU initialization routine. Create the routine be referring to section 2.6.3, CPU Initialization Routine Creation Example.

The following shows the sample CPU initialization routine provided by each HI series OS.

```
;*****************************************************************************;
;*                HI7000/4 CPU initialize routine                            *;
;*                Copyright (c)  Hitachi, Ltd. 2000.                         *;
;*                Licensed Material of Hitachi, Ltd.                         *;
;*                HI7000/4(HS0700ITI41SR) V1.0                               *;
;*****************************************************************************;
;*****************************************************************************;
;*****************************************************************************;
;*  FILE        = 7604_cpuasm.src ;                                          *;
;*  CPU type  = SH7604                                                       *;
;*****************************************************************************;
        .program        _hi_cpuasm
        .heading        "hi_cpuasm : CPU initialize ro
        .export         _hi_cpuasm
        .import         _hi_cpuini
        .section        P_hicpuasm, code, align = 4
;
;*****************************************************************************;
;*  BSC address                                                              *;
;*****************************************************************************;
BSC_BASE        .assign  h'fffffe0              ;  BSC      base address
BCR1            .assign  h'fffffe0-BSC_BASE     ;  BCR1     address offset
BCR2            .assign  h'fffffe4-BSC_BASE     ;  BCR2     address offset
WCR             .assign  h'fffffe8-BSC_BASE     ;  WCR      address offset
MCR             .assign  h'fffffec-BSC_BASE     ;  MCR      address offset
RTCSR           .assign  h'ffffff0-BSC_BASE     ;  RTCSR    address offset
RTCNT           .assign  h'ffffff4-BSC_BASE     ;  RTCNT    address offset
RTCOR           .assign  h'ffffff8-BSC_BASE     ;  RTCOR    address offset
;
MD_REG_BASE     .assign  h'ffff8000             ;  mode register base address of SDRAM
;
CMF_BIT         .assign  h'0080                 ;  CMF bit in RTCSR
;
;*****************************************************************************;
;*  BSC initial data                                                         *;
;*  After reset, you must initialize BSC for memory (stack) access at first. *;
;*  Please modify these definition in order to your hardware.                *;
;*****************************************************************************;
BCR1_DATA       .assign  h'a55a0000 + h'03f0    ;  BCR1     initial data
BCR2_DATA       .assign  h'a55a0000 + h'00fc    ;  BCR2     initial data
WCR_DATA        .assign  h'a55a0000 + h'aaff    ;  WCR      initial data
MCR_DATA        .assign  h'a55a0000 + h'0000    ;  MCR      initial data
RTCSR_DATA      .assign  h'a55a0000 + h'0000    ;  RTCSR    initial data
RTCNT_DATA      .assign  h'a55a0000 + h'0000    ;  RTCNT    initial data
RTCOR_DATA      .assign  h'a55a0000 + h'0000    ;  RTCOR    initial data
;
STP_REFRESH     .assign  h'a55a0000             ;  RTCSR initial data (stop count-up)
;
MODE_DATA       .assign  h'0000                 ;  data of SDRAM mode register
MODE_ADDRESS    .assign  MD_REG_BASE+MODE_DATA ; address to set MODE_DATA
IDLE_TIME       .assign  566                    ;  loop counter for idle-time
REFRESH_CNT     .assign  h'8                    ;  counter for dummy refresh
;
```

Defines data for initialization processing.
Modify the values or add data as necessary.

Figure 2.2 HI7000/4 CPU Initialization Routine: Assembly Language (SH7604) (1/2)

```
;*****************************************************************************;
;*  NAME          = _hi_cpuasm                                               ;*;
;*  FUNCTION      = CPU initialize routine ;
;*****************************************************************************;
puasm:
;***** Initialize BSC
;       mov.l   #BSC_BASE, r0           ; set BCR base address to gbr
;       ldc     r0, gbr
;
;       mov.l   #BCR1_DATA, r0          ; initialize BCR1
;       mov.l   r0, @(BCR1, gbr)
;
;       mov.l   #BCR2_DATA, r0          ; initialize BCR2
;       mov.l   r0, @(BCR2, gbr)
;
;       mov.l   #WCR_DATA, r0           ; initialize WCR
;       mov.l   r0, @(WCR, gbr)
;
;       mov.l   #MCR_DATA, r0           ; initialize MCR
;       mov.l   r0, @(MCR, gbr)
;
;       mov.l   @(RTCSR, gbr), r0       ; dummy read for CMF off
;       mov.l   #STP_REFRESH, r0        ; stop refresh
;       mov.l   r0, @(RTCSR, gbr)
;
;       mov.l   #RTCNT_DATA, r0         ; initialize RTCNT
;       mov.l   r0, @(RTCNT, gbr)
;
;       mov.l   #RTCOR_DATA, r0         ; initialize RTCOR
;       mov.l   r0, @(RTCOR, gbr)
;
;*** Initialize SDRAM
;
;       mov.l   #IDLE_TIME, r0          ; loop for i
;hi_cpuasm010:
;       add     #-1, r0
;       cmp/eq  #0, r0
;       bf      hi_cpuasm010
;
;       mov.w   #MODE_DATA, r0          ; set mode register
;       mov.l   #MODE_ADDRESS, r1
;       mov.w   r0, @r1
;
;       mov.l   #RTCSR_DATA, r0         ; initialize RTCSR
;       mov.l   r0, @(RTCSR, gbr)
;
;       mov     #0, r1                  ; loop for dummy refresh
;       mov.w   #REFRESH_CNT, r2
;hi_cpuasm020:
;       mov.l   @(RTCSR, gbr), r0
;       tst     #CMF_BIT, r0            ; check CMF bit
;       bt      hi_cpuasm020
;
;       add     #1, r1                  ; loop counter up
;       cmp/eq  r1, r2                  ; if end dummy refresh
;       bt      hi_cpuasm030            ; then goto hi_cpuasm030
;       mov.l   #RTCSR_DATA, r0         ; clear CMF bit
;       bra     hi_cpuasm020
;       mov.l   r0, @(RTCSR, gbr)
;
;hi_cpuasm030:
;
        mov.l   #_hi_cpuini, r0         ; get hi_cp
         jmp    @r0                     ; jump to h
        nop                             ; never return to this point
;
                .pool
        .end
```

Initializes the bus state controller.
Remove comment characters (;) as necessary.

Initializes external memory (SDRAM).
Remove comment characters (;) as necessary.

After completing the CPU initialization processing written in assembly language, branches to the initialization processing written in C language.

Figure 2.2 HI7000/4 CPU Initialization Routine: Assembly Language (SH7604) (2/2)

```
/*****************************************************************************/
/*                HI7000/4 CPU initialize routine                            */
/*                Copyright (c)  Hitachi, Ltd. 2000.                         */
/*                Licensed Material of Hitachi, Ltd.                         */
/*                HI7000/4(HS0700ITI41SR) V1.0                               */
/*****************************************************************************/
/*****************************************************************************/
/*  FILE        = 7604_cpuini.c ;                                            */
/*  CPU type  = SH7604                                                       */
/*****************************************************************************/
#include   <machine.h>
#include   "itron.h"
#include   "kernel.h"

/* extern  void    _INITSCT(void);  */       /* section-initialize routine                    */

#pragma section _hicpuini
#pragma noregsave(hi_cpuini)

void        hi_cpuini(void)
{

/*** Initialize Hardware Environment ***/

/*** Initialize Software Environment ***/

/*  _INITSCT();    */                        /* Call section-initialize routine               */

    vsta_knl();                              /* Start kernel                                  */
}
```

Calls the section expanding processing. Remove comment characters (/* and */) as necessary.

Calls the kernel initialization processing. After completing the CPU initialization processing, be sure to call the kernel initialization processing.

Figure 2.3 HI7000/4 CPU Initialization Routine: C Language (SH7604)

```
;*****************************************************************************;
;*          HI7700/4 CPU initialize routine                                  ;*
;*          Copyright (c) 2000 (2003) Renesas Technology Corp.               ;*
;*          and Renesas Solutions Corp. All Rights Reserved.                 ;*
;*          HI7700/4 (HS0770ITI41SR) V1.0                                    ;*
;*****************************************************************************;
;*****************************************************************************;
;* FILE       = 7708_cpuasm.src ;                                            ;*
;* CPU type   = SH7708                                                       ;*
;*****************************************************************************;
        .program        _hi_cpuasm
        .export         _hi_cpuasm
        .import         _hi_cpuini
        .import         __kernel_pon_sp
        .import         __kernel_man_sp
        .section        P_hicpuasm, code, align = 4
;
;*****************************************************************************;
;*          EXPEVT address, data                                             ;*
;*****************************************************************************;
CCN_BASE        .assign  h'ffffffd0                ; INTC
EXPEVT          .assign  h'ffffffd4-CCN_BASE       ; EXPEVT address offset
;
PON_CODE        .assign  h'000                     ; power-on reset exception code
;
;*****************************************************************************;
;*          BSC address                                                      ;*
;*****************************************************************************;
BSC_BASE        .assign  h'ffffff60                ; BSC          base address
BCR1            .assign  h'ffffff60-BSC_BASE       ; BCR1         address offset
BCR2            .assign  h'ffffff62-BSC_BASE       ; BCR2         address offset
WCR1            .assign  h'ffffff64-BSC_BASE       ; WCR1         address offset
WCR2            .assign  h'ffffff66-BSC_BASE       ; WCR2         address offset
MCR             .assign  h'ffffff68-BSC_BASE       ; MCR          address offset
DCR             .assign  h'ffffff6a-BSC_BASE       ; DCR          address offset
PCR             .assign  h'ffffff6c-BSC_BASE       ; PCR          address offset
RTCSR           .assign  h'ffffff6e-BSC_BASE       ; RTCSR        address offset
RTCNT           .assign  h'ffffff70-BSC_BASE       ; RTCNT        address offset
RTCOR           .assign  h'ffffff72-BSC_BASE       ; RTCOR        address offset
RFCR            .assign  h'ffffff74-BSC_BASE       ; RFCR         address offset
SDMR_CS2        .assign  h'ffffd000                ; SDMR (CS2)   base address
SDMR_CS3        .assign  h'ffffe000                ; SDMR (CS3)   base address
CMF_BIT         .assign  h'0080                    ; CMF bit in RTCSR
;
;*****************************************************************************;
;*          BSC initial data                                                 ;*
;*****************************************************************************;
;* After reset, you must initialize BSC for memory (stack) access at first.  ;*
;* Please modify these definition in order to your hardware.                ;*
;*****************************************************************************;
BCR1_DATA       .assign  h'0000                    ; BCR1         initial data
BCR2_DATA       .assign  h'3ffc                    ; BCR2         initial data
WCR1_DATA       .assign  h'3fff                    ; WCR1         initial data
WCR2_DATA       .assign  h'ffff                    ; WCR2         initial data
MCR_DATA        .assign  h'0000                    ; MCR          initial data
DCR_DATA        .assign  h'0000                    ; DCR          initial data
PCR_DATA        .assign  h'0000                    ; PCR          initial data
RTCSR_DATA      .assign  h'a500 + h'00             ; RTCSR        initial data
RTCNT_DATA      .assign  h'a500 + h'00             ; RTCNT        initial data
RTCOR_DATA      .assign  h'a500 + h'00             ; RTCOR        initial data
RFCR_DATA       .assign  h'a400 + h'000            ; RFCR         initial data
STP_REFRESH     .assign  h'a500                    ; RTCSR initial data(stop count-up)
SDMR2_DATA      .assign  h'0230                    ; SDMR_CS2     initial data
SDMR3_DATA      .assign  h'0230                    ; SDMR_CS3     initial data
IDLE_TIME       .assign  h'566                     ; loop counter for idle-time
REFRESH_CNT     .assign  h'8                       ; counter for dummy refresh
;
```

Defines data for initialization processing.
Modify the values or add data as necessary.

Figure 2.4 HI7700/4 CPU Initialization Routine: Assembly Language (SH7708) (1/3)

```
;*****************************************************************************;
;*  NAME        = _hi_cpuasm                                                 ;*;
;*  FUNCTION    = CPU initialize routine                                     ;*;
;*****************************************************************************;
_hi_cpuasm:
;***** Initialize BSC
;       mov.l   #BSC_BASE, r0           ; set BCR base address to gbr
;        ldc    r0, gbr
;
;       mov.w   #BCR1_DATA, r0          ; Initialize BCR1
;       mov.w   r0, @(BCR1, gbr)
;
;       mov.w   #BCR2_DATA, r0          ; Initialize BCR2
;       mov.w   r0, @(BCR2, gbr)
;
;       mov.w   #WCR1_DATA, r0          ; Initialize WCR1
;       mov.w   r0, @(WCR1, gbr)
;
;       mov.w   #WCR2_DATA, r0          ; Initialize WCR2
;       mov.w   r0, @(WCR2, gbr)
;
;       mov.w   #MCR_DATA, r0           ; Initialize MCR
;       mov.w   r0, @(MCR, gbr)
;
;       mov.w   #DCR_DATA, r0           ; Initialize DCR
;       mov.w   r0, @(DCR, gbr)
;
;       mov.w   #PCR_DATA, r0           ; Initialize PCR
;       mov.w   r0, @(PCR, gbr)
;
;       mov.w   #STP_REFRESH, r0        ; stop refresh
;       mov.w   r0, @(RTCSR, gbr)
;
;       mov.w   #RTCNT_DATA, r0         ; Initialize RTCNT
;       mov.w   r0, @(RTCNT, gbr)
;
;       mov.w   #RTCOR_DATA, r0         ; Initialize RTCOR
;       mov.w   r0, @(RTCOR, gbr)
;
;       mov.w   #RFCR_DATA, r0          ; Initialize RFCR
;       mov.w   r0, @(RFCR, gbr)
;
```

Initializes the bus state controller.
Remove comment characters (;) as necessary.

Figure 2.4 HI7700/4 CPU Initialization Routine: Assembly Language (SH7708) (2/3)

Initializes external memory (SDRAM). Remove comment characters (;) as necessary.

```
;*** Initialize SDRAM
;
;       mov.l   #IDLE_TIME, r0          ; loop for idle-time
;hicpuasm010:
;       add     #-1, r0
;       cmp/eq  #0, r0
;       bf      hicpuasm010
;
;       mov.l   #SDMR_CS2, r0           ; Initialize SDMR(CS2)
;       mov.l   #SDMR2_DATA*4, r2
;       mov.b   r1, @(r0, r2)           ; write dummy data(r1)
;
;       mov.l   #SDMR_CS3, r0           ; Initialize SDMR(CS3)
;       mov.l   #SDMR3_DATA*4, r2
;       mov.b   r1, @(r0, r2)           ; write dummy data(r1)
;
;       mov.w   #RTCSR_DATA, r0         ; Initialize RTCSR
;       mov.w   r0, @(RTCSR, gbr)
;
;       mov.w   #REFRESH_CNT, r2
;hi_cpuasm020:
;       mov.w   @(RFCR,gbr), r0         ; read RFCR
;       cmp/ge  r2, r0                  ; if end dummy refresh
;       bf      hi_cpuasm020            ; else goto hi_cpuasm020
;
;hi_cpuasm030:
;
;***** Initialize sp and jump to hi_cpuini() written by C-language
        mov.l   #CCN_BASE, r2           ; get CCN base address
        mov.l   #PON_CODE, r3           ; get exception code to power-on
        mov.l   @(EXPEVT, r2), r0       ; get exception code
        cmp/eq  r3, r0                  ; if exception != power-on
        bf      hi_cpuasm050            ; then hi_cpuasm050
;
        mov.l   #__kernel_pon_sp, r2    ; get stack address
;
hi_cpuasm040:
        mov     r2, r15                 ; set SP
;
        mov.l   #_hi_cpuini, r0         ; get hi_cpuini address
        jmp     @r0                     ; jump to hi_cpuini()
        nop                             ; never return to this point
;
hi_cpuasm050:
        mov.l   #__kernel_man_sp, r2    ; get stack address
        bra     hi_cpuasm040
        nop
;
                .pool
;
        .end
```

After completing the CPU initialization processing written in assembly language, branches to the initialization processing written in C language.

Figure 2.4 HI7700/4 CPU Initialization Routine: Assembly Language (SH7708) (3/3)

```
/*****************************************************************************/
/*          HI7700/4 CPU initialize routine                                  */
/*          Copyright (c) 2000(2003) Renesas Technology Corp.                */
/*          and Renesas Solutions Corp. All Rights Reserved.                 */
/*          HI7700/4(HS0770ITI41SR) V1.0A                                    */
/*****************************************************************************/
/*****************************************************************************/
/*  FILE          = 7708_cpuini.c ;                                          */
/*  CPU type      = SH7708                                                   */
/*****************************************************************************/
#include   <machine.h>
#include   "itron.h"
#include   "kernel.h"

/*****************************************************************************/
/*          environment data                                                 */
/*****************************************************************************/
#define IOBASE          0xffffe80                /* I/O base address    = 0xffffe80 */
#define CCR             (0xffffffec - IOBASE)    /* CCN CCR         address offset */

#define CACHE_ON    0x00000001                   /* CACHE enable  data          */
#define CACHE_OFF   0x00000000                   /* CACHE disable data          */

/* extern  void    _INITSCT(void); */            /* section-initialize routine  */

#pragma section _hicpuini
/*****************************************************************************/
/*  NAME          = hi_cpuini                                                */
/*  FUNCTION      = CPU initialize routine                                   */
/*****************************************************************************/
#pragma noregsave(hi_cpuini)

void    hi_cpuini(void)
{

/*** Initialize Hardware Environment ***/
    set_gbr((VP)IOBASE);
    gbr_write_long(CCR, CACHE_OFF);

/*** Initialize Software Environment ***/

/*  _INITSCT();    */                            /* Call section-initialize routine     */

    vsta_knl();                                  /* Start kernel                        */
}
```

Calls the section expanding processing. Remove comment characters (/* and */) as necessary.

Calls the kernel initialization processing. After completing the CPU initialization processing, be sure to call the kernel initialization processing.

Figure 2.5 HI7700/4 CPU Initialization Routine: C Language (SH7708)

```
;*****************************************************************************;
;*              HI7750/4 CPU initialize routine                              ;*;
;*              Copyright (c) 2000(2003) Renesas Technology Corp.            ;*;
;*              and Renesas Solutions Corp. All Rights Reserved.             ;*;
;*              HI7750/4(HS0775ITI41SR) V1.0                                 ;*;
;*****************************************************************************;
;*****************************************************************************;
;*****************************************************************************;
;*  FILE        = 7750_cpuasm.src ;                                          ;*;
;*  CPU type    = SH7750                                                     ;*;
;*****************************************************************************;
        .program        _hi_cpuasm
        .heading        "hi_cpuasm : CPU initialize routine"
        .export         _hi_cpuasm
        .import         _hi_cpuini
        .import         __kernel_pon_sp
        .import         __kernel_man_sp
        .section        P_hicpuasm, code, align = 4
;
;*****************************************************************************;
;*  EXPEVT address, data                                                     ;*;
;*****************************************************************************;
CCN_BASE        .assign  h'ff000020                 ; C
EXPEVT          .assign  h'ff000024-CCN_BASE        ; E
;
PON_CODE        .assign  h'000                      ; power-on reset exception code
;
;*****************************************************************************;
;*  BSC address                                                              ;*;
;*****************************************************************************;
BSC_BASE        .assign  h'ff800000                 ; BSC       base address
BCR1            .assign  h'ff800000-BSC_BASE        ; BCR1      address offset
BCR2            .assign  h'ff800004-BSC_BASE        ; BCR2      address offset
WCR1            .assign  h'ff800008-BSC_BASE        ; WCR1      address offset
WCR2            .assign  h'ff80000c-BSC_BASE        ; WCR2      address offset
WCR3            .assign  h'ff800010-BSC_BASE        ; WCR3      address offset
MCR             .assign  h'ff800014-BSC_BASE        ; MCR       address offset
PCR             .assign  h'ff800018-BSC_BASE        ; PCR       address offset
RTCSR           .assign  h'ff80001c-BSC_BASE        ; RTCSR     address offset
RTCNT           .assign  h'ff800020-BSC_BASE        ; RTCNT     address offset
RTCOR           .assign  h'ff800024-BSC_BASE        ; RTCOR     address offset
RFCR            .assign  h'ff800028-BSC_BASE        ; RFCR      address offset
SDMR2           .assign  h'ff900000                 ; SDMR2     address
SDMR3           .assign  h'ff940000                 ; SDMR3     address
CMF_BIT         .assign  h'0080                     ; CMF bit in RTCSR
;
;*****************************************************************************;
;*  BSC initial data                                                         ;*;
;*  After reset, you must initialize BSC for memory(stack) access at first.  ;*;
;*  Please modify these definition in order to your hardware.                ;*;
;*****************************************************************************;
BCR1_DATA       .assign  h'00000000                 ; BCR1      initial data
BCR2_DATA       .assign  h'3ffc                     ; BCR2      initial data
WCR1_DATA       .assign  h'77777777                 ; WCR1      initial data
WCR2_DATA       .assign  h'fffeefff                 ; WCR2      initial data
WCR3_DATA       .assign  h'07777777                 ; WCR3      initial data
MCR_DATA        .assign  h'00000000                 ; MCR       initial data
PCR_DATA        .assign  h'0000                     ; PCR       initial data
RTCSR_DATA      .assign  h'a500 + h'00              ; RTCSR     initial data
RTCNT_DATA      .assign  h'a500 + h'00              ; RTCNT     initial data
RTCOR_DATA      .assign  h'a500 + h'00              ; RTCOR     initial data
RFCR_DATA       .assign  h'a400 + h'000             ; RFCR      initial data
STP_REFRESH     .assign  h'a500                     ; RTCSR initial data(stop count-up)
SDMR2_DATA      .assign  h'0230                     ; SDMR2     initial data
SDMR3_DATA      .assign  h'0230                     ; SDMR3     initial data
IDLE_TIME       .assign  h'1000                     ; loop counter for idle-time
REFRESH_CNT     .assign  h'8                        ; counter for dummy refresh
;
```

Defines data for initialization processing.
Modify the values or add data as necessary.

Figure 2.6 HI7750/4 CPU Initialization Routine: Assembly Language (SH7750) (1/3)

```
;*****************************************************************************;
;*  NAME        = _hi_cpuasm                                                 ;*;
;*  FUNCTION    = CPU initialize routine ;                                   ;*;
;*****************************************************************************;
_hi_cpuasm:
;***** Initialize BSC
;       mov.l   #BSC_BASE, r0           ; set BSC base address to gbr
;        ldc    r0, gbr
;
;       mov.l   #BCR1_DATA, r0          ; Initialize BCR1
;       mov.l   r0, @(BCR1, gbr)
;
;       mov.w   #BCR2_DATA, r0          ; Initialize BCR2
;       mov.w   r0, @(BCR2, gbr)
;
;       mov.l   #WCR1_DATA, r0          ; Initialize WCR1
;       mov.l   r0, @(WCR1, gbr)
;
;       mov.l   #WCR2_DATA, r0          ; Initialize WCR2
;       mov.l   r0, @(WCR2, gbr)
;
;       mov.l   #WCR3_DATA, r0          ; Initialize WCR3
;       mov.l   r0, @(WCR3, gbr)
;
;       mov.l   #MCR_DATA, r0           ; Initialize MCR
;       mov.l   r0, @(MCR, gbr)
;
;       mov.w   #PCR_DATA, r0           ; Initialize PCR
;       mov.w   r0, @(PCR, gbr)
;
;       mov.w   #STP_REFRESH, r0        ; stop refresh
;       mov.w   r0, @(RTCSR, gbr)
;
;       mov.w   #RTCNT_DATA, r0         ; Initialize RTCNT
;       mov.w   r0, @(RTCNT, gbr)
;
;       mov.w   #RTCOR_DATA, r0         ; Initialize RTCOR
;       mov.w   r0, @(RTCOR, gbr)
;
;       mov.w   #RFCR_DATA, r0          ; Initialize RFCR
;       mov.w   r0, @(RFCR, gbr)
;
```

Initializes the bus state controller.
Remove comment characters (;) as necessary.

Figure 2.6 HI7750/4 CPU Initialization Routine: Assembly Language (SH7750) (2/3)

Initializes external memory (SDRAM).
Remove comment characters (;) as necessary.

```
;*** Initialize SDRAM
;
;           mov.l     #IDLE_TIME, r0          ; loop for idle-time
;hicpuasm010:
;           add       #-1, r0
;           cmp/eq    #0, r0
;           bf        hicpuasm010
;
;           mov.l     #SDMR2, r0              ; Initialize SDMR(CS2)
;           mov.l     #SDMR2_DATA*4, r2
;           mov.b     r1, @(r0, r2)           ; write dummy data(r1)
;
;           mov.l     #SDMR3, r0              ; Initialize SDMR(CS3)
;           mov.l     #SDMR3_DATA*4, r2
;           mov.b     r1, @(r0, r2)           ; write dummy data(r1)
;
;           mov.w     #RTCSR_DATA, r0         ; Initialize RTCSR
;           mov.w     r0, @(RTCSR, gbr)
;
;           mov.w     #REFRESH_CNT, r2
;hi_cpuasm020:
;           mov.w     @(RFCR, gbr), r0        ; read RFCR
;           cmp/ge    r2, r0                  ; if end dummy refresh
;           bf        hi_cpuasm020            ; else goto hi_cpuasm020
;
;hi_cpuasm030:
;
;***** Initialize sp and jump to hi_cpuini() written by C-language
            mov.l     #CCN_BASE, r2           ; get CCN base address
            mov.l     #PON_CODE, r3           ; get exception code to power-on
            mov.l     @(EXPEVT, r2), r0       ; get exception code
            cmp/eq    r3, r0                  ; if exception != power-on
            bf        hi_cpuasm050            ; then hi_cpuasm050
;
            mov.l     #__kernel_pon_sp, r2    ; get stack address
;
hi_cpuasm040:
            mov       r2, r15                 ; set SP
;
            mov.l     #_hi_cpuini, r0         ; get hi_cpuini address
            jmp       @r0                     ; jump to hi_cpuini()
            nop                               ; never return to this point
;
hi_cpuasm050:
            mov.l     #__kernel_man_sp, r2    ; get stack address
            bra       hi_cpuasm040
            nop
;
                      .pool
;
            .end
```

After completing the CPU initialization processing written in assembly language, branches to the initialization processing written in C language.

Figure 2.6 HI7750/4 CPU Initialization Routine: Assembly Language (SH7750) (3/3)

```
/*****************************************************************************/
/*                HI7750/4 CPU initialize routine                            */
/*                Copyright (c) 2000(2003) Renesas Technology Corp.          */
/*                and Renesas Solutions Corp. All Rights Reserved.           */
/*                HI7750/4(HS0775ITI41SR) V1.0A                              */
/*****************************************************************************/
/*****************************************************************************/
/*  FILE        = 7750_cpuini.c ;                                            */
/*  CPU type  = SH7750                                                       */
/*****************************************************************************/
#include    <machine.h>
#include    "itron.h"
#include    "kernel.h"

#define CCR_DATA 0x0000090d                    /* CACHE enable data          */

/* extern  void    _INITSCT(void); */          /* section-initialize routine */

#pragma section _hicpuini
#pragma noregsave(hi_cpuini)

void    hi_cpuini(void)
{

/*** Initialize Hardware Environment ***/
/*  vini_cac((UW)CCR_DATA); */

/*** Initialize Software Environment ***/

/*   _INITSCT();    */                        /* Call section-initialize routine  */

    vsta_knl();                                /* Start kernel                     */
}
```

Calls the section expanding processing.
Remove comment characters (/* and */) as necessary.

Calls the kernel initialization processing.
After completing the CPU initialization processing, be sure to call the kernel initialization processing.

Figure 2.7 HI7750/4 CPU Initialization Routine: C Language (SH7750)

```
;*****************************************************************************
;***                                                                       ***
;***     HI2000/3 Version (uITRON V3.0)                                    ***
;***     HI2000/3 user/system application file                             ***
;***                                                                       ***
;***     Copyright (c)  Hitachi, Ltd. 1998.                                ***
;***     Licensed Material of Hitachi, Ltd.                                ***
;***                                                                       ***
;*****************************************************************************
;
        .program        _2655acpu
        .heading        "### 2655acpu.src : H8S/2655 initialize module ###"
;
        .section        h2susr_ram, data, align = 2
        .res.b          18
CPUINI_SP:      .equ    $
;
        .section        h2suser, code, align = 2
;
        .export         _H_2S_CPUINI
        .import         _H_2S_INIT
;
        .aifdef DX
        .import         _HI_DEAMON_INI
        .aendi
;
;*****************************************************************************
;* specifications ;                                                          *
;* name     = _H_2S_CPUINI : H8S/2655 initialize module                      *
;* function =                                                                *
;* notes    =                                                                *
;* date     = 99/02/22                                                       *
;* author   = Hitachi, Ltd.                                                  *
;* attribute = public                                                        *
;* class    = system                                                         *
;* linkage  =                                                                *
;* input    = none                                                           *
;* output   = none                                                           *
;* end of specifications ;                                                   *
;*****************************************************************
;
;
        .radix  d                       ;;xxxxx -> d
;
;###### interrupt register address #####;;
SYSCR:          .assign h'00ffff39      ;;system control register
MSTPCRH:        .assign h'00ffff3c      ;;module stop control register H
MSTPCRL:        .assign h'00ffff3d      ;;module stop control register L
;
;###### system control register #######;;SYSCR
RAME:           .assign b'00000001      ;;RAM enable
NMIEG:          .assign b'00001000      ;;NMI edge select
INTM0:          .assign b'00010000      ;;interrupt mode 0
INTM1:          .assign b'00100000      ;;interrupt mode 1
MACS:           .assign b'10000000      ;;MAC register saturation
;
;### module stop control register H ####;;MSTPCRH
A_D:            .assign b'11111101      ;;A/D module select
D_A:            .assign b'11111011      ;;D/A module select
PPG:            .assign b'11110111      ;;PPG module select
TMR:            .assign b'11101111      ;;TMR module select
TPU:            .assign b'11011111      ;;TPU module select
DTC:            .assign b'10111111      ;;DTC module select
DMAC:           .assign b'01111111      ;;DMAC module select
;
;### module stop control register L ####;;MSTPCRL
SCI0:           .assign b'11011111      ;;SCI0 module select
SCI1:           .assign b'10111111      ;;SCI1 module select
SCI2:           .assign b'01111111      ;;SCI2 module select
;
```

Defines data for initialization processing.
Modify the values or add data as necessary.

Figure 2.8 HI2000/3 CPU Initialization Routine (H8S/2655) (1/2)

```
_H_2S_CPUINI:
        mov.l   #CPUINI_SP:32, sp           ;:get CPUINI_SP
        mov.b   @SYSCR:32, r0L              ;:get SYSCR
        and.b   #low~(INTM0 | INTM1):8, r0L ;:clear interrupt mode bit
        or.b    #low  (INTM0 | INTM1):8, r0L ;:set interrupt mode = 3
        mov.b   r0L, @SYSCR:32              ;:set SYSCR
;
;
        mov.b   @MSTPCRH:32, r0L            ;:get MSTPCRH
        and.b   #low TPU:8, r0L             ;:set TPU bit off
        mov.b   r0L, @MSTPCRH:32            ;:set I
;
        .aifdef DX
        jsr     @_HI_DEAMON_INI             ;:call
        .aendi
;
        jmp     @_H_2S_INIT                 ;:goto HI2000/3 initialize module
;
```

Add the following processing as necessary.
- Initialization of the bus state controller
- Initialization of external memory (SDRAM)

Calls the kernel initialization processing.
After completing the CPU initialization processing, be sure to call the kernel initialization processing.

Figure 2.8 HI2000/3 CPU Initialization Routine (H8S/2655) (2/2)

```
;*****************************************************************************
;*                                                                           *
;*    HI1000/4 Version (uITRON V4.0)                                         *
;*    HI1000/4 user/system application file                                  *
;*                                                                           *
;*    Copyright (C) 1998, 2003  Renesas Technology Corp. All right reserved  *
;*                                                                           *
;*****************************************************************************
;
            .program        _1650cpu
            .heading        "### 1650cpu.src : H8SX/1650 initialize module ###"
;
            .section        P_hicpuini, code, align = 2
;
            .export         _KERNEL_H_CPUINI
            .import         _KERNEL_HI_OS_SP
            .import         _vsta_knl
;
;*****************************************************************************
;* specifications ;                                                          *
;* name      = 1650cpu.src                    : H8SX/1650 initialize module  ;  *
;* function  = CPU Initialize routine                                       ;  *
;* notes     =                                                              ;  *
;* input     = none                                                         ;  *
;* output    = none                                                         ;  *
;* end of specifications ;
;*****************************************************
;
;
            .radix  d                                   ;;
;
;############# register address #############;;
INTCR:          .assign h'00FFFF32              ;;interrupt control register
MSTPCRA:        .assign h'00FFFDC8              ;;module stop control register A
ABWCR:          .assign h'00FFFD84              ;;bus width control register
ASTCR:          .assign h'00FFFD86              ;;access state control register
WTCRA:          .assign h'00FFFD88              ;;wait control register A
WTCRB:          .assign h'00FFFD8A              ;;wait control register B
;
;######### interrupt control register ##########;;INTCR
INTM0:          .assign b'00010000              ;;interrupt mode bit0
INTM1:          .assign b'00100000              ;;interrupt mode bit1
;
;####### module stop control register A #######;;MSTPCRA
MSTPA0:         .assign h'FFFE                  ;;TPU ch 5 - 0
VBR_ADR:        .assign 0                       ;;VBR address
;
_KERNEL_H_CPUINI:
            mov.l   #_KERNEL_HI_OS_SP:32, sp    ;;SP
            mov.l   #VBR_ADR, er0               ;;
             ldc.l  er0, vbr                    ;;se
            mov.l   #h'ffffff00, er0            ;;ini
             ldc.l  er0, sbr                    ;;initial SBR
;
;           mov.w   #h'00ff,   @ABWCR:32        ;;set ABWCR
;           mov.w   #h'0000, @ASTCR:32          ;;set ASTCR
;           mov.w   #h'0000, @WTCRA:32          ;;set WTCRA
;           mov.w   #h'0000, @WTCRB:32          ;;set WTCRB
;
            mov.b   #INTM1, r0L
            mov.b   r0L, @INTCR:32
;
            mov.w   @MSTPCRA:32, r0
            and.w   #MSTPA0:16, r0
            mov.w   r0, @MSTPCRA:32
;
            jmp     @_vsta_knl                  ;;goto vsta_knl
;
```

Defines data for initialization processing.
Modify the values or add data as necessary.

Add the following processing as necessary.
- Initialization of the bus state controller
- Initialization of external memory (SDRAM)

Calls the kernel initialization processing.
After completing the CPU initialization processing, be sure to call the kernel initialization processing.

Figure 2.9 HI1000/4 CPU Initialization Routine (H8SX/1650)

2.2.1 FAQs about CPU Initialization Routine

This section answers questions about CPU initialization routine which are frequently asked by users of the HI series OS.

FAQ Contents:

(1) Transferring Programs

Classification: CPU initialization routine

Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4

Please explain how to transfer all sections from ROM to RAM by using the ROM support function (ROM to RAM mappedsections in the Optlinker).

Answer

To transfer P_xxx sections (code sections) from ROM to RAM and execute them in RAM, the section initialization processing must be done in the CPU initialization routine, that is, the P_xxx section contents must be copied to the R sections.

B_xxx sections should be placed in RAM; they do not need to be placed in ROM first and then transferred to RAM.

When the ROM support function is used, they are transferred to RAM and execution can be started with the kernel initialization by simply issuing vsta_knl in the CPU initialization routine.

For details on program transfer, refer to the following descriptions in the compiler application notes.

- Application note for SuperH™ RISC engine Family C/C++ Compiler Package
  Q&A: Transfer to RAM and Execution of a Program
- Application note for H8S, H8/300 Series C/C++ Compiler Package
  Q&A: How to Run Programs in RAM

The following shows an example of program transfer in the SH7770.

(Continued on next page)

(Continued from previous page)

Answer

```
/*****************************************************************************/
/*                HI7750/4 CPU initialize routine                            */
/*                Copyright (c) 2000(2003) Renesas Technology Corp.          */
/*                and Renesas Solutions Corp. All Rights Reserved.           */
/*                HI7750/4(HS0775ITI41SR) V1.1.00                            */
/*****************************************************************************/
/*****************************************************************************/
/*  FILE        = 7770_cpuini.c ;                                            */
/*  CPU type    = SH7770                                                     */
/*****************************************************************************/
#include    <machine.h>
#include    "itron.h"
#include    "kernel.h"

/* extern  void    _INITSCT(void); */                   /* section-initialize routine        */

#pragma section _hicpuini
#pragma noregsave(hi_cpuini)

void    hi_cpuini(void)
{
/*    ER  ercd; */

/*** Initialize Hardware Environment ***/
/* ercd = vini_cac((ATR)(TCAC_IC_ENABLE | TCAC_OC_ENABLE | TCAC_P1_CB)); */

/*** Initialize Software Environment ***/
/*  _INITSCT();    */                                  /* Call section-initialize routine   */

    vsta_knl();                                         /* Start kernel                      */
}
```

Remove comment characters (/* and */) to call the section initialization processing.

Figure 2.10 Definition in CPU Initialization Routine

(Continued on next page)

(Continued from previous page)

Answer

```
/*****************************************************************************/
/*          HI7750/4 section initialize routine                              */
/*          Copyright (c) 2000(2003) Renesas Technology Corp.                */
/*          and Renesas Solutions Corp. All Rights Reserved.                 */
/*          HI7750/4(HS0775ITI41SR) V1.1.00                                  */
/*****************************************************************************/
/*****************************************************************************/
/*  FILE          = 7770_initsct.c  ;                                        */
/*****************************************************************************/
#include    <machine.h>
#include    "itron.h"

extern int *B_BGN, *B_END, *D_BGN, *D_END, *D_ROM;
extern void _INITSCT(void);

#pragma section _hicpuini
/*****************************************************************************/
/*  NAME          = _INITSCT ;                                               */
/*  FUNCTION  = Section Initialize routine ;                                 */
/*****************************************************************************/
void _INITSCT(void)
{
    register int *p, *q;
    for(p = B_BGN; p<B_END; p++)                    /* 0 clear B-section          */
        *p = 0;
    for(p = D_BGN, q = D_ROM; p<D_END; p++, q++) /* Copy D-section -> R-section */
        *p = *q;
}
```

Figure 2.11 INITSCT() Processing

(2) Defining Initial Stack Pointer

Classification: CPU initialization routine					
Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4
Is the stack pointer defined in the project file used for system creation a temporary stack pointer used until the kernel starts execution?					

Answer

This stack pointer is used until the kernel starts execution, that is, it is used by the CPU initialization routine.

The specified stack area must be set as accessible when the CPU initialization routine is initiated. Before the stack pointer is specified by the CPU initialization routine, the stack area must be enabled (necessary settings must be made in the bus state controller (BSC) to enable external memory such as SDRAM or SRAM).

In kernel initialization processing initiated after the CPU initialization routine is completed, the stack pointer is switched to point to the kernel stack allocated through the configurator.

(3) Hang-up after Initialization

Classification: CPU initialization routine					
Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4
Is it possible that execution will hang up during CPU initialization?					

Answer

After the CPU initialization routine processing, the kernel initialization processing is called, but after the kernel initialization processing, execution does not return to the CPU initialization routine.

Control is passed to the initial start task after the kernel initialization processing.

Therefore, if execution hangs up without control being passed to the initial start task, any of the following may be the cause; check the system for each possibility.

- The stack area used during kernel initialization is insufficient, and another area is overwritten and damaged.
- The RAM area used during kernel initialization cannot be accessed.
- The target board generates an illegal interrupt or an undefined exception.
- Initially defined information is incorrect, and an error occurs in kernel initialization.

For an overview of the processing after the CPU initialization routine is initiated, refer to section 2.1, Overview of Processing from Reset to Task Initiation in this application note.

2.3 Overview of Kernel Initialization Processing

The kernel initialization processing includes the following.

- Switching to the kernel stack pointer
- Creating and initializing the kernel management areas (management tables)
- Creating and initializing the initially defined objects
- Calling the system initialization routine

The kernel initialization processing creates and initializes the necessary information for kernel operation.

2.3.1 Initialization Routine

The initialization routine can be created as a C-language function.

Figure 2.12 shows a sample of the initialization routine code.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

void inirtn(VP_INT exinf)
{

  /* Initialization routine processing */

}
```

Includes standard header files.

Figure 2.12 Sample Initialization Routine Code

The initialization routine must be created in accordance with the application programs.

Refer to the provided sample initialization routine (timer initialization routine) and create the routine in accordance with the application programs used.

2.3.2 Shifting to Multitask Environment

After kernel initialization processing is completed, the dispatcher is initiated. The dispatcher schedules tasks as follows.

- When tasks are READY

  The dispatcher assigns the CPU to the task which has the highest priority among the READY tasks (the task which has the highest priority level and which received an initiation request first among the tasks having the same priority level).
- When no tasks are READY

  The dispatcher passes control to system idling processing, which causes the system to enter the idle state (SUSPENDED state) until a task enters the READY state (initiated).

2.3.3 FAQ about Kernel Initialization Processing

This section answers a question about kernel initialization processing which is frequently asked by users of the HI series OS.

FAQ Contents:

(1) Initializing Kernel Work Area

Classification: Kernel initialization processing					
Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4
Should the kernel work area be initialized (cleared to 0) in the CPU initialization routine?					
Answer The kernel work area does not need to be initialized in the CPU initialization routine. For the kernel work area (B_hiwrk section area), the kernel initialization processing creates and initializes the necessary information for kernel operation.					

2.4 Overview of System Idling Processing

When no task should be executed (no task is READY), the kernel enters the system idle state (to be more specific, interrupt masks are canceled and an infinite loop is entered).

2.4.1 System Idling Processing Using SLEEP Instruction

(1) HI7000/4 Series

To use the power-down mode of the microcomputer in the system idling processing, create a task of the lowest priority level; in that task, make the necessary settings and execute the SLEEP instruction.

Figure 2.13 shows a sample code.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

#pragma inline_asm(sleep)
static void sleep(void)
{
    sleep
}

#pragma noregsave(IdleTask)

void IdleTask(VP_INT exinf)
{

    while(1){
        /* Make necessary settings in SBYCR. */
        sleep();
    }

}
```

Includes standard header files.

Uses #pragma inline_asm to execute a SLEEP instruction.

Specifies #pragma noregsave because the registers do not need to be saved when a task is initiated.

Figure 2.13 System Idling Processing Using SLEEP Instruction (HI7000/4 Series)

(2) HI2000/3

Figure 2.14 shows the system idling processing provided as a sample file.

```
;*****************************************************************************
;* specifications ;                                                          *
;* name     = _H_SYSTEM_IDLE : HI2000/3 SYSTEM IDLING DEFINE    ;            *
;* function =                                                   ;            *
;* notes    =                                                   ;            *
;* date     = 99/02/22                                          ;            *
;* author   = Hitachi, Ltd.                                     ;            *
;* attribute = public                                           ;            *
;* class    = system                                            ;            *
;* linkage  =                                                   ;            *
;* input    =                                                   ;            *
;* output   =                                                   ;            *
;* end of specifications ;                                                   *
;*****************************************************************************
_H_SYSTEM_IDLE:
;       bra     $                       ;:forever loop
;
        sleep                           ;:sleep define
        bra     _H_SYSTEM_IDLE:8        ;:branch _H_SYSTEM_IDLE
;
```

Figure 2.14 System Idling Processing Using SLEEP Instruction (HI2000/3)

(3) HI1000/4

Figure 2.15 shows the system idling processing provided as a sample file.

```
;*************************************************************************
;*                                                                       *
;*    HI1000/4 Version (uITRON V4.0)                                     *
;*    HI1000/4 kernel idle routine                                       *
;*                                                                       *
;*    Copyright (C) 1998, 2003  Renesas Technology Corp. All right reserved *
;*                                                                       *
;*************************************************************************
        .program        _1650idle
        .heading        "### 1650idle.src : kernel idle routine ###"
        .section        P_hiidle, code, align = 2
;
        .export         _KERNEL_H_SYSTEM_IDLE
;
;*************************************************************************
; *specifications ;                                                      *
; *name     = _KERNEL_H_SYSTEM_IDLE : HI1000/4 kernel idle routine       *
; *function  =                                                           ;  *
; *notes     =                                                           ;  *
; *input     =                                                           ;  *
; *output    =                                                           ;  *
; *end of specifications ;                                               *
;*************************************************************************
_KERNEL_H_SYSTEM_IDLE:
;       bra     $                                   ;;forever loop
;
        sleep                                       ;;sleep define
        bra     _KERNEL_H_SYSTEM_IDLE:8 ;;branch _KERNEL_H_SYSTEM_IDLE
;
;*****************************************;
        .end; of 1650idle.src
```

Figure 2.15 System Idling Processing Using SLEEP Instruction (HI1000/4)

2.4.2 FAQs about System Idling Processing

This section answers questions about system idling processing which are frequently asked by users of the HI series OS.

FAQ Contents:

(1) Return from Idle State

Classification: System idling processing					
Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4
The kernel remains in the idle state after slp_tsk is executed. What could cause this?					

Answer

This may be caused by any of the following.

- The task that issued slp_tsk cannot be made READY.
  - There is no task to wake up the task that issued slp_tsk.
  - The task to wake up the task that issued slp_tsk is not initiated.
  - The interrupt handler to wake up the task that issued slp_tsk is not initiated.
- There is no other task that should be executed than the task that issued slp_tsk.

The kernel enters the system idle state when no task is in the READY state.

Create a task or interrupt handler to wake up the task that issued slp_tsk. This will cause execution to return from the system idle state.

(2) SLEEP Instruction Execution in the Idle State

Classification: System idling processing					
Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4
When the kernel detects the system idle state, it enters the sleep state by executing the SLEEP instruction. Please explain in detail this OS processing.					
Answer The kernel simply executes a SLEEP instruction. The kernel does not control SBYCR. It must be controlled through the application when a SLEEP instruction is executed.					

2.5 Overview of System Termination Processing

If an abnormal state is found in the system, the system termination processing is initiated. The following is a list of the causes of system termination (system down).

- The system termination processing is forcibly called from an application program
- An error or conflict is found in the initially-defined object information
- An error is detected within the kernel
- An undefined interrupt or exception is detected

The system termination processing must be prepared as an application program by the user. Refer to the provided sample file and create the program in accordance with the application programs.

Various items of error information are passed to the system termination processing. At debugging, the error information passed through parameters when an abnormal state is found in the system can be checked by specifying breakpoints through the emulator or the ICE; this is useful for system error analysis.

For details on the parameters passed to the system termination processing, refer to the user's manual of the HI series OS used or section 5, Debugging, in this application note.

2.5.1 Sample System Termination Processing

(1) HI7000/4

Figure 2.16 shows the system termination processing provided as a sample file.

```
/*****************************************************************************/
/*                    HI7000/4 System down routine                           */
/*                    Copyright (c)  Hitachi, Ltd. 2000.                     */
/*                    Licensed Material of Hitachi, Ltd.                     */
/*                    HI7000/4(HS0700ITI41SR) V1.0                           */
/*****************************************************************************/
/*****************************************************************************/
/*  FILE          = 7604_sysdwn.c ;                                          */
/*****************************************************************************/
#include    <machine.h>
#include    "itron.h"
#include    "kernel.h"
#include    "kernel_id.h"

#pragma section _hisysdwn
/* #pragma interrupt (_kernel_sysdwn) */
/*****************************************************************************/
/*  NAME          = _kernel_sysdwn ;                                         */
/*  FUNCTION  = System down routine ;                                        */
/*****************************************************************************/
void          _kernel_sysdwn(type, ercd, inf1, inf2)
W type;     /*system down type */
                        /* type >=  1  : system down of user program        */
                        /* type ==  0  : initial information error          */
                        /* type == -1  : context error of ext_tsk           */
                        /* type == -2  : context error of exd_tsk           */
                        /* type == -16: undefined interrupt / exception     */
ER ercd;   /* error code  */
                        /* type >=  0  : error code of user program         */
                        /* type ==  0  : error code of initial information  */
                        /* type == -1  : error code of ext_tsk              */
                        /* type == -2  : error code of exd_tsk              */
                        /* type == -16: interrupt vector number             */
VW inf1;   /* information-1  */
                        /* type >=  0  : information of user program        */
                        /* type ==  0  : indicator of initial information error */
                        /* type == -1  : address of ext_tsk call            */
                        /* type == -2  : address of exd_tsk call            */
                        /* type == -16: address of interrupt occurrence     */
VW inf2;   /* information-2  */
                        /* type >=  0  : information of user program        */
                        /* type ==  0  : number of error initial information */
                        /* type == -16: SR of interrupt occurrence          */
{
    set_imask(SR_IMS15);    /* mask all interrupt                           */
    while(TRUE);            /* endless loop                                 */
}
```

Figure 2.16 System Termination Processing (HI7000/4)

(2) HI7700/4 and HI7750/4

Figure 2.17 shows the system termination processing provided as a sample file.

```
/*****************************************************************************/
/*              HI7700/4 System down routine                                 */
/*              Copyright (c) 2000(2003) Renesas Technology Corp.            */
/*              and Renesas Solutions Corp. All Rights Reserved.             */
/*              HI7700/4(HS0770ITI41SR) V1.0                                 */
/*****************************************************************************/
/*****************************************************************************/
/*  FILE          = 7708_sysdwn.c ;                                          */
/*****************************************************************************/
#include    <machine.h>
#include    "itron.h"
#include    "kernel.h"
#include    "kernel_id.h"

/*****************************************************************************/
/*              environment data                                             */
/*****************************************************************************/
#define MD_BIT 0x40000000  /* SR.MD bit                                     */

#pragma section _hisysdwn
/*#pragma interrupt(_kernel_sysdwn) */
/*****************************************************************************/
/*  NAME          = _kernel_sysdwn ;                                         */
/*  FUNCTION  = System down routine ;                                        */
/*****************************************************************************/
void        _kernel_sysdwn(type, ercd, inf1, inf2)
W type;     /* system down type */
                        /* type >=  1  : system down of user program        */
                        /* type ==  0  : initial information error          */
                        /* type == -1  : context error of ext_tsk           */
                        /* type == -2  : context error of exd_tsk           */
                        /* type == -16: undefined interrupt/exception       */
ER ercd;    /* error code */
                        /* type >=  0  : error code of user program         */
                        /* type ==  0  : error code of initial information  */
                        /* type == -1  : error code of ext_tsk              */
                        /* type == -2  : error code of exd_tsk              */
                        /* type == -16: interrupt vector number             */
VW inf1;    /* information-1  */
                        /* type >=  0  : information of user program        */
                        /* type ==  0  : indicator of initial information error */
                        /* type == -1  : address of ext_tsk call            */
                        /* type == -2  : address of exd_tsk call            */
                        /* type == -16: address of interrupt occurrence     */
VW inf2;    /* information-2  */
                        /* type >=  0  : information of user program        */
                        /* type ==  0  : number of error initial information */
                        /* type == -16: SR of interrupt occurrence          */
{
   set_cr(MD_BIT | (SR_IMS15 << 4));     /* mask all interrupt              */
   while(TRUE);                          /* endless loop                    */
}
```

Figure 2.17 System Termination Processing (HI7700/4 and HI7750/4)

(3) HI2000/3

Figure 2.18 shows the system termination processing provided as a sample file.

```
;
;*****************************************************************************
;* specifications ;                                                          *
;* name     = _HIPRG_ABNOML : abnormal quit handler                          *
;* function =                                                                *
;* notes    =                                                                *
;* date     = 99/02/22                                                       *
;* author   = Hitachi, Ltd.                                                  *
;* attribute = public                                                        *
;* class    = system                                                         *
;* linkage  =                                                                *
;* input    =                                                                *
;* output   =                                                                *
;* end of specifications ;                                                   *
;*****************************************************************************
;
_HIPRG_ABNOML:
        orc     #HIDEF_IMASK_CCR:8, ccr  ;;interrupt mask for CCR register
        orc     #HIDEF_IMASK_EXR:8, exr  ;;interrupt mask for EXR register
        bra     $                        ;;forever loop
;
```

Figure 2.18 System Termination Processing (HI2000/3)

(4) HI1000/4

Figure 2.19 shows the system termination processing provided as a sample file.

```
;
;*****************************************************************************;
;                                                                             ;
;* NAME = vsys_dwn                                                           *;
;* FILE   = vsys_dwn.src                                                     *;
;                                                                             ;
;* FUNC = System down routine                                                *;
;* NOTE =                                                                    *;
;                                                                             ;
;* INPU  = none       :                                                      *;
;                                                                             ;
;* OUTP = none        :                                                      *;
;*****************************************************************************;
;                                                                             ;
;
;
        .section        P_hisysdwn, code, align = 2
;
        .export         _vsys_dwn
        .export         _ivsys_dwn
_vsys_dwn:
_ivsys_dwn:
        bra   _vsys_dwn:8
        rts
;
        .end; of vsys_dwn.src
```

Figure 2.19 System Termination Processing (HI1000/4)

2.5.2 FAQ about System Termination Processing

This section answers a question about system termination processing which is frequently asked by users of the HI series OS.

FAQ Contents:

(1) System-Down Causes

Classification: System termination processing					
Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4
The system goes down after initialization processing. Please explain how to determine the cause of this.					

Answer

The following is a list of the causes of system down.

- The system termination processing is forcibly called from an application program
- An error or conflict is found in the initially-defined object information
- An error is detected within the kernel
- An undefined interrupt or exception is detected

Set a breakpoint to the beginning of the system termination processing to obtain parameters at system-down to analyze the cause of this.

For details on the parameters passed to the system termination processing, refer to the user's manual of the HI series OS used or section 5, Debugging, in this application note.

2.6 Application Program Types

Table 2.2 application programs are necessary to develop a system by using the HI series OS.

Table 2.2 Application Program Types and Necessity

Type	Necessity	Remarks
Task	Always	
Interrupt handler	Always	
CPU initialization routine	Always	
System termination processing routine	Always	
System idling processing routine	Always	
Initialization routine	Optional	
Timer interrupt routine (including timer initialization routine)	*1	
Task exception processing routine	Optional	*2
Extended service call routine	Optional	*3
CPU exception handler	Optional	*2
Cyclic handler	Optional	
Alarm hander	Optional	*2
Overrun handler	Optional	*2

Always: Must always be prepared.

Optional: Must be prepared when necessary.

Notes: 1. Not necessary when the system does not use the time management function.
2. Supported by the HI7000/4 series; not supported by the HI2000/3 or HI1000/4.
3. Supported by the HI7000/4 and HI1000/4 series; not supported by the HI2000/3.

Table 2.3 shows the relationships among these application programs, the system state, and the service call types that can be issued.

Table 2.3 Application Programs and System State

Application Program	System State	Service Call Type that Can be Issued
Task	Task context	Service calls for task context
Interrupt handler	Non-task context	Service calls for non-task context
Initialization routine	Non-task context	Service calls for non-task context
Task exception processing routine	Task context	Service calls for task context
Extended service call routine	Issuing context*[1]	Issuing context*[1]
CPU exception handler	*2	*3
Cyclic handler	Non-task context	Service calls for non-task context
Alarm hander	Non-task context	Service calls for non-task context
Overrun handler	Non-task context	Service calls for non-task context

Notes: 1. The context when the service call is issued is inherited.
2. The issuing context in the HI7000/4 series and the non-task context in the HI1000/4. The CPU exception handler is not supported by the HI2000/3.
3. For details on the service calls that can be issued, refer to the user's manual for the HI series OS.

2.6.1 Task Creation Example

A task should be created as a C-language function. Read the following notes before terminating a task.

Table 2.4 Service Call for Task Termination and Notes

HI Series OS	Service Call	Notes
HI7000/4 series	ext_tsk() or exd_tsk() service call	The task terminating service call can be omitted (the ext_tsk() service call is assumed when omitted).
HI2000/3	ext_tsk() system call	The task terminating service call must not be omitted (a task must always be terminated by an ext_tsk() system call). When execution is returned from the task to its caller, correct system operation cannot be guaranteed.
HI1000/4	ext_tsk() service call	The task terminating service call can be omitted (the ext_tsk() service call is assumed when omitted).

For the value of each context register when a task is initiated, refer to the user's manual for the HI series OS used.

Figure 2.20 shows a sample of the code for a task.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

#pragma noregsave(Task)

void Task(VP_INT exinf)
{

    /* Task processing */

    ext_tsk();
/*  exd_tsk(); */

}
```

Includes standard header files.

Specifies #pragma noregsave because the registers do not need to be saved when the task is initiated.

When the task is initiated by an act_tsk() service call, the exinf value defined at task registration is passed through a parameter.
When the task is initiated by a sta_tsk() service call, the stacd value specified by the service call is passed through a parameter.

Defines termination of task processing.

Figure 2.20 Sample Task Code

Note: For the standard header files that should be included, refer to the user's manual for the HI series OS used.

2.6.2 Interrupt Handler Creation Example

The following shows a sample of the interrupt handler code for each HI series OS.

(1) Sample Interrupt Handler Code for HI7000/4 Series

Figure 2.21 shows a sample of an interrupt handler code.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

void Inh(void)
{

    /* Interrupt handler processing */

}
```

Includes standard header files.

Writes an interrupt handler as a void-type function.

Figure 2.21 Sample Interrupt Handler Code (HI7000/4 Series)

Notes: 1. For the standard header files that should be included, refer to the user's manual of the HI series OS used.
2. When using a coprocessor, all of its registers must be saved and restored in the interrupt handler.

By using IRL interrupts, two interrupt sources of different levels can be assigned to one vector table. When using IRL interrupts, write the interrupt handler as shown in the following example.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

#define I_HILEVEL 15

void vec071_handler14(void)
{
    /* IRL14 interrupt processing */
}

void vec071_handler15(void)
{
    /* IRL15 interrupt processing */
}

void vec071(void)
{

    if((get_imask()) == I_HILEVEL)
        vec071_handler15();
    else
        vec071_handler14();

}
```

Includes standard header files.

Defines the higher level.

Writes the interrupt processing for each level.

Writes the interrupt handler to be registered for an interrupt source as a void-type function.

Figure 2.22 Sample of Interrupt Handler Code when Using IRL Interrupts (HI7000/4 Series)

Note the following when using the direct interrupt handler in the HI7000/4.

- The interrupt handler is initiated without involving kernel management when an interrupt occurs.
- The direct interrupt handler cannot issue service calls.

Figure 2.23 shows a sample of a direct interrupt handler code.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

#define stksz 512
VW stk[stksz / sizeof(VW)];
static const VP p_stk = (VP) & stk[stksz / sizeof(VW)];

#pragma interrupt(DirectInh(sp = p_stk, tn = 25))
/* #pragma interrupt(high(sp = p_stk)) */
/* #pragma interrupt(nmi()) */

void DirectInh(void)
{

    /* Interrupt handler processing */

}
```

Includes standard header files.

Defines the stack size for the interrupt handler.

Allocates the stack area for the interrupt handler.

Defines the stack pointer for the interrupt handler.

*

Declares the interrupt handler as an interrupt function by using #pragma interrupt.

Writes the interrupt handler as a void-type function.

Figure 2.23 Sample Direct Interrupt Handler Code (HI7000/4)

Note: * Specify the following in #pragma interrupt.

- Stack switch setting (sp=)

  Stacks must not be switched in the NMI interrupt handler.

- Trap return setting (tn = 25)

  Specify tn = 25 for the interrupt handler that is lower than the kernel interrupt mask level. The interrupt handler (including NMI) that is higher than the kernel interrupt mask level must be terminated by the RTE instruction, and the trap return setting must not be made.

The direct interrupt handler is not supported by the HI7700/4 or HI7750/4.

(2) Sample Interrupt Handler Code for HI2000/3 and HI1000/4

The interrupt handler must save and restore the register values when an interrupt occurs. Create the interrupt handler through the following procedure.

Table 2.5 Interrupt Handler Creation Procedure

Processing	Description
Saving registers used in the interrupt handler	• Saves stack pointer. The stack pointer must be modified to point to the stack area dedicated to the interrupt handler (this processing can be omitted when the interrupt handler does not use a stack). • Saves register contents.
Interrupt processing	Processing performed in the interrupt handler
Restoring registers used in the interrupt handler	• Restores register contents. The stack pointer must be modified (this processing can be omitted when the interrupt handler does not use a stack).
Terminating the interrupt handler	Calls the ret_int routine when the interrupt level is lower than the kernel interrupt mask level or executes the RTE instruction when the interrupt level is higher than the kernel interrupt mask level.

The interrupt handler should be created by using the interrupt function creation directive (#pragma interrupt) of the C compiler. Figure 2.24 shows a sample of an interrupt handler code.

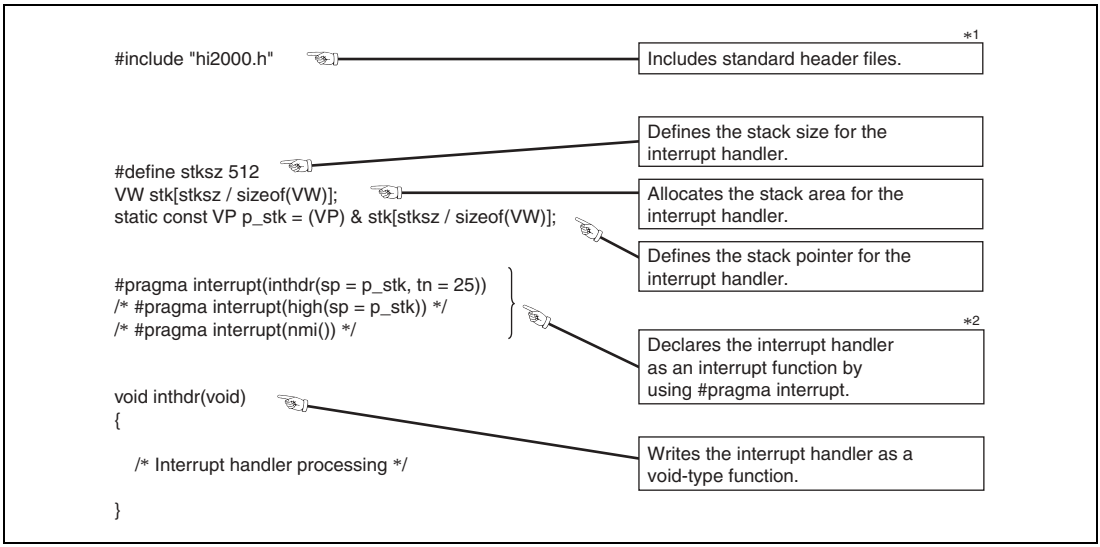


Figure 2.24 Sample Interrupt Handler Code (HI2000/3)

Notes: 1. For the standard header files that should be included, refer to the user's manual of the HI series OS used.

2. Stack switching and interrupt function termination must be specified in #pragma interrupt. For details, refer to the user's manual of the HI series OS used.

2.6.3 CPU Initialization Routine Creation Example

The HI2000/3 and HI1000/4 provide sample files written in assembly language. To use a CPU initialization routine written in C language, a call for the C-language CPU initialization routine should be added to the assembly-language CPU initialization routine.

The following shows sample modifications of the CPU initialization routine written in assembly language and samples of the CPU initialization routine code written in C language for the HI2000/3 and HI1000/4, respectively.

(1) HI2000/3

```
_H_2S_CPUINI:
        mov.l    #CPUINI_SP:32, sp          ;:get CPUINI_SP
        mov.b    @SYSCR:32, r0L             ;:get SYSCR
        and.b    #low~(INTM0|INTM1):8, r0L  ;:clear interrupt mode bit
        or.b     #low (INTM0|INTM1):8, r0L  ;:set interrupt mode = 3
        mov.b    r0L, @SYSCR:32             ;:set SYSCR
;
        mov.b    @MSTPCRH:32, r0L           ;:get MSTPCRH
        and.b    #low TPU:8, r0L            ;:set TPU bit off
        mov.b    r0L, @MSTPCRH:32           ;:set MSTPCRH
;
        .aifdef DX
        jsr      @_HI_DEAMON_INI            ;:call to init deamon code
        .aendi
;
        bsr      @_h_cpuini_c               ;:call to C-language initialize routine
;
        jmp      @_H_2S_INIT                ;:goto HI2000/3 initialize module
;
```

Calls the C-language CPU initialization routine. After the assembly-language CPU initialization routine is completed, call the C-language CPU initialization routine.

Note: This example assumes the h_cpuini_c is the name of the C-language CPU initialization routine.

Figure 2.25 Sample Modification of Assembly-Language CPU Initialization Routine (HI2000/3)

```
void h_cpuini_c(void)
{

/*** Initialize Hardware Environment ***/

/*** Initialize Software Environment ***/

//    _INITSCT();                /* Call section-initialize routine  */

}
```

Figure 2.26 Sample C-Language CPU Initialization Routine Code (HI2000/3)

(2) HI1000/4

```
_KERNEL_H_CPUINI:
      mov.l    #_KERNEL_HI_OS_SP:32, sp  ;:SP <- OS stack
      mov.l    #VBR_ADR, er0             ;:
      ldc.l    er0, vbr                  ;:set VBR address
      mov.l    #h'ffffff00, er0          ;:initial SBR
      ldc.l    er0, sbr                  ;:initial SBR
;
;     mov.w    #h'00ff, @ABWCR:32        ;:set ABWCR
;     mov.w    #h'0000, @ASTCR:32        ;:set ASTCR
;     mov.w    #h'0000, @WTCRA:32        ;:set WTCRA
;     mov.w    #h'0000, @WTCRB:32        ;:set WTCRB
;
      mov.b    #INTM1, r0L               ;:set interrupt mode 2
      mov.b    r0L, @INTCR:32            ;:set INTCR
;
      mov.w    @MSTPCRA:32, r0           ;:get MSTPCRA
      and.w    #MSTPA0:16, r0            ;:set TPU bit off
      mov.w    r0, @MSTPCRA:32           ;:set MSTPCRA
;
      jmp      @_h_cpuini                ;:goto _h_cpuini
;
```

Calls the C-language CPU initialization routine.
After the assembly-language CPU initialization routine is completed, call the C-language CPU initialization routine.

Note: This example assumes that h_cpuini is the name of the C-language CPU initialization routine.

Figure 2.27 Sample Modification of Assembly-Language CPU Initialization Routine (HI1000/4)

```
void h_cpuini(void)
{

/*** Initialize Hardware Environment ***/

/*** Initialize Software Environment  ***/

//    _INITSCT();              /* Call section-initialize routine  */

      vsta_knl();              /* Start kernel                     */
}
```

Calls the kernel initialization processing.
Be sure to call the kernel initialization processing after the CPU initialization processing is completed.

Figure 2.28 Sample C-Language CPU Initialization Routine Code (HI1000/4)

Refer also to section 2.2, Overview of CPU Initialization Routine.

2.6.4 System Termination Processing Creation Example

The HI2000/3 and HI1000/4 provide sample files written in assembly language. To write the system termination processing in C language, refer to the following sample code.

```
                                                    *1
#include "hi2000.h"          ---------- Includes standard header files.

                        *2
void HIPRG_ABNOML(void)      ---------- Writes the system termination processing as
{                                       a void-type function.

    set_imask_ccr(0xC0);    /* All interrupts masked */
    set_imask_exr(0x07);    /* All interrupts masked */
    while(1);               /* endless loop          */

}
```

Figure 2.29 Sample System Termination Processing Code (HI2000/3)

Notes: 1. For the standard header files that should be included, refer to the user's manual of the HI series OS used.

2. The function must be named HIPRG_ABNOML in the HI2000/3 or vsys_dwn in the HI1000/4 because the kernel refers to the function by these respective names.

2.6.5 System Idling Routine Creation Example

The HI2000/3 and HI1000/4 provide sample files written in assembly language. To write the system idling routine in C language, refer to the following sample code.

```
                                                                    *1
#include "hi2000.h"          ---------- Includes standard header files.

                    *2       ---------- Writes the system idling routine as a void-type
void HIPRG_IDLE(void)                    function.
{

    set_imask_ccr(0x00);   /* All interrupt masks canceled */
    set_imask_exr(0x00);   /* All interrupt masks canceled */
    while(1);              /* endless loop                 */

}
```

Figure 2.30 Sample System Idling Routine Code (HI2000/3)

Notes: 1. For the standard header files that should be included, refer to the user's manual of the HI series OS used.

2. The function must be named HIPRG_IDLE in the HI2000/3 or KERNEL_H_SYSTEM_IDLE in the HI1000/4 because the kernel refers to the function by these respective names.

2.6.6 Initialization Routine Creation Example

The HI2000/3 and HI1000/4 provide sample files written in assembly language. To write the initialization routine in C language, refer to the following sample code.

```
#include "hi2000.h"                    *1   Includes standard header files.

void inirtn(VP_INT exinf)   *2
{

  /* Initialization routine processing */

}
```

Figure 2.31 Sample Initialization Routine Code

Notes: 1. For the standard header files that should be included, refer to the user's manual of the HI series OS used.

2. In the HI2000/3, the extended information (exinf) is not passed to the initialization routine; do not create a code for receiving this information (the HI2000/3 does not provide a parameter for this information).

2.6.7 Timer Interrupt Routine Creation Example

Figure 2.32 shows a sample of a timer interrupt routine code.

```
#include "itron.h"
#include "kernel.h"

void _kernel_tmrint(void)
{
    /* Timer interrupt routine processing */

}
```

*1 Includes standard header files.

*2 Writes the timer interrupt routine as a void-type function.

Figure 2.32 Sample Timer Interrupt Routine Code

Notes: 1. For the standard header files that should be included, refer to the user's manual of the HI series OS used.

2. The function must be named as follows because the kernel refers to the function by these respective names.

HI Series OS	Function Name
HI7000/4 series	_kernel_tmrint
HI2000/3	Any user-defined name
HI1000/4	_KERNEL_H_TIM

2.6.8 Task Exception Processing Routine Creation Example

The task exception processing routine is only supported by the HI7000/4 series OS. Figure 2.33 shows a sample of a task exception processing routine code.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

#pragma noregsave(Texrtn)

void Texrtn(TEXPTN texptn, VP_INT exinf)
{
    /* Task exception processing routine processing */
}
```

Includes standard header files.

Specifies #pragma noregsave because the task exception processing routine does not need to save register contents when a task is initiated.

Writes the timer interrupt routine as a void-type function.
The exception source and extended information are passed through parameters.

Figure 2.33 Sample Task Exception Processing Routine Code

2.6.9 Extended Service Call Routine Creation Example

The extended service call routine is only supported by the HI7000/4 series OS. Figure 2.34 shows a sample of a task exception processing routine code.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

ER_UINT Svcrtn(VP_INT par1, VP_INT par2)
{
    /* Extended service call routine processing */

    return E_OK;

}
```

Includes standard header files.

The parameter values specified by cal_svc are passed to the extended service call routine. Specify the same number of parameters as those specified by cal_svc.

Sends a return value to the caller.

Figure 2.34 Sample Extended Service Call Routine Code

2.6.10 CPU Exception Handler Creation Example

The CPU exception handler is supported by the HI7000/4 series and HI1000/4. Figure 2.35 shows a sample of the CPU exception handler code.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

void cpuexphdr(void)
{

    /* CPU exception handler processing */

}
```

Includes standard header files.

Writes the CPU exception handler as a void-type function in the same way as the interrupt handler.

Figure 2.35 Sample CPU Exception Handler Code

2.6.11 Time Event Handler Creation Example

(1) Cyclic Handler Example

(a) Sample cyclic handler code for the HI7000/4 series and HI1000/4

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

void cychdr(VP_INT exinf)
{

    /* Cyclic handler processing */

}
```

Includes standard header files.

Writes the cyclic handler as a void-type function in the same way as the general interrupt handler. The exinf value defined at creation is returned through a parameter.

Figure 2.36 Sample Cyclic Handler Code (HI7000/4 Series and HI1000/4)

(b) Sample cyclic handler code for the HI2000/3

```
#include "hi2000.h"

void cychdr(void)

#pragma asm
   stm.l (er0-er1), @-sp    ;: Saves er0 and er1 in the stack.
   bsr    cychdr_main       ;: Calls the main processing.
   ldm.l @sp+, (er0-er1)    ;: Restores er0 and er1.
   rts
#pragma endasm

void cychdr_main(void)
{
   /* Cyclic handler processing */
}
```

Includes standard header files.

Writes the cyclic handler as a void-type function.

Figure 2.37 Sample Cyclic Handler Code (HI2000/3)

(2) Alarm Handler Example (Supported Only in the HI7000/4 Series)

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

void almhdr(VP_INT exinf)
{

   /* Alarm handler processing */

}
```

Includes standard header files.

Writes the alarm handler as a void-type function in the same way as the general interrupt handler. The exinf value defined at creation is returned through a parameter.

Figure 2.38 Sample Alarm Handler Code (Only in HI7000/4 Series)

(3) Overrun Handler Example (Supported Only in the HI7000/4 Series)

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

void ovrhdr(ID tskid, VP_INT exinf)
{

   /* Overrun handler processing */

}
```

Includes standard header files.

Writes the overrun handler as a void-type function in the same way as the general interrupt handler. The tskid value of the task that caused initiation of the overrun handler and the exinf value defined at creation are returned through parameters.

Figure 2.39 Sample Overrun Handler Code (Only in HI7000/4 Series)

2.7 Development Procedures for Application Programs

A system using the HI series OS can be developed though either of two approaches:

(1) The system is newly developed

(2) Programs of an existing system are used

In approach (1), the programs listed in section 2.5, Application Program Types are created, and integrated into the final form of the system.

As this approach newly creates every application program, optimum programs to embed the HI series OS can be developed.

(1) Dividing the functions in a top-down manner

The functions must be divided as far as possible. This step determines the functions that can be simultaneously processed in parallel. The divided functions are defined as tasks or interrupt handlers.

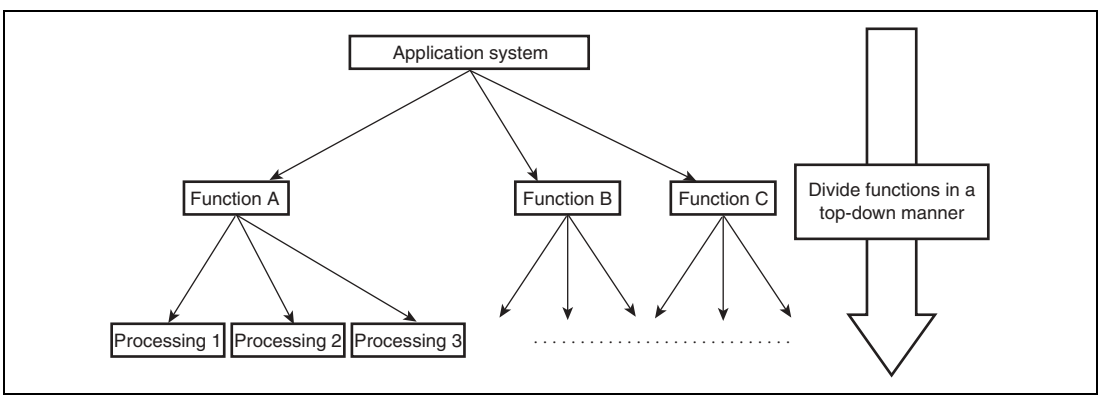


Figure 2.40 Dividing Functions in a Top-Down Manner

(2) Combining tasks (functions) for the same processing after divided

The action to combine the tasks for the same processing is called a task merge (no task merge is needed for interrupt handlers, because a handler is defined for each interrupt source). This step defines the tasks for which functional dependency is eliminated.

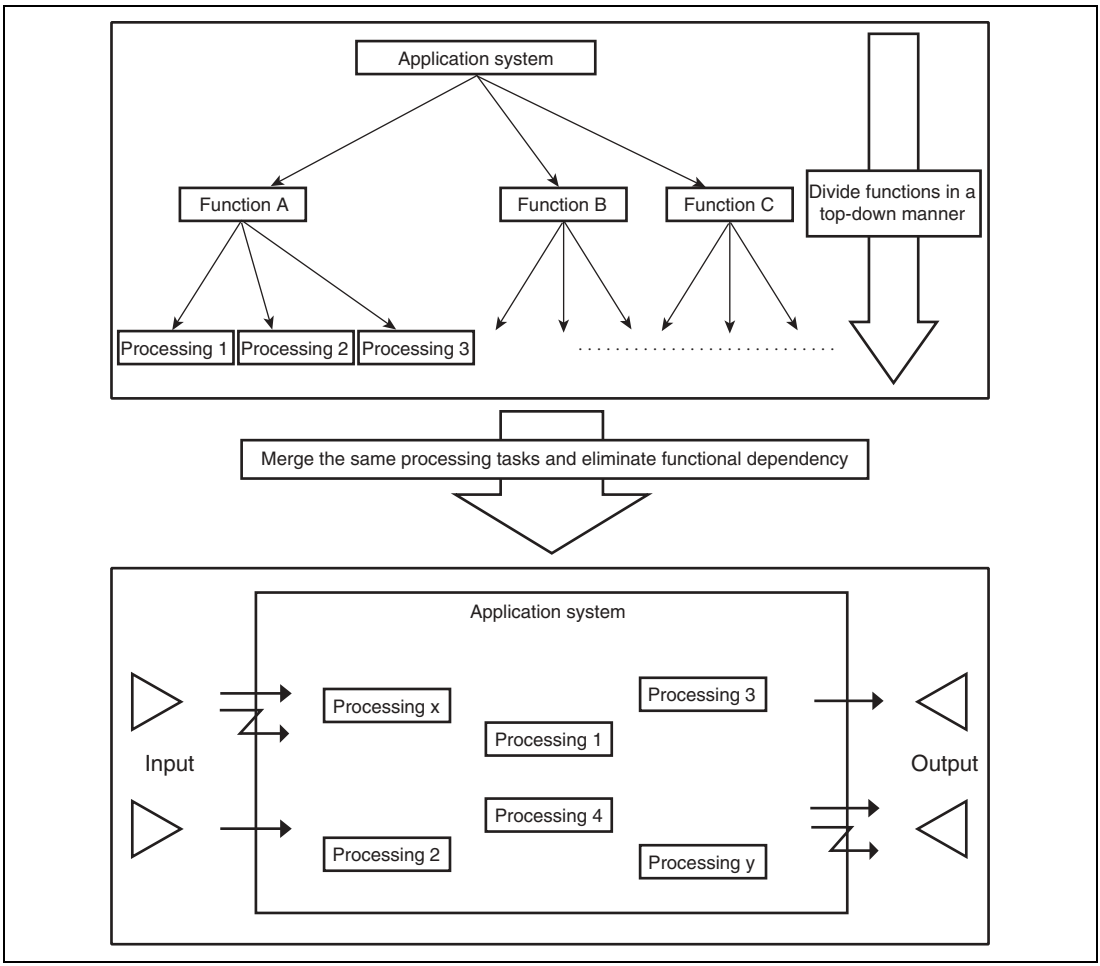


Figure 2.41 Merging Same Functions and Eliminating Functional Dependency

After this step, objects of the HI series OS are assigned to the interfaces (synchronization and communication) between multiple tasks or between a task and an interrupt handler.

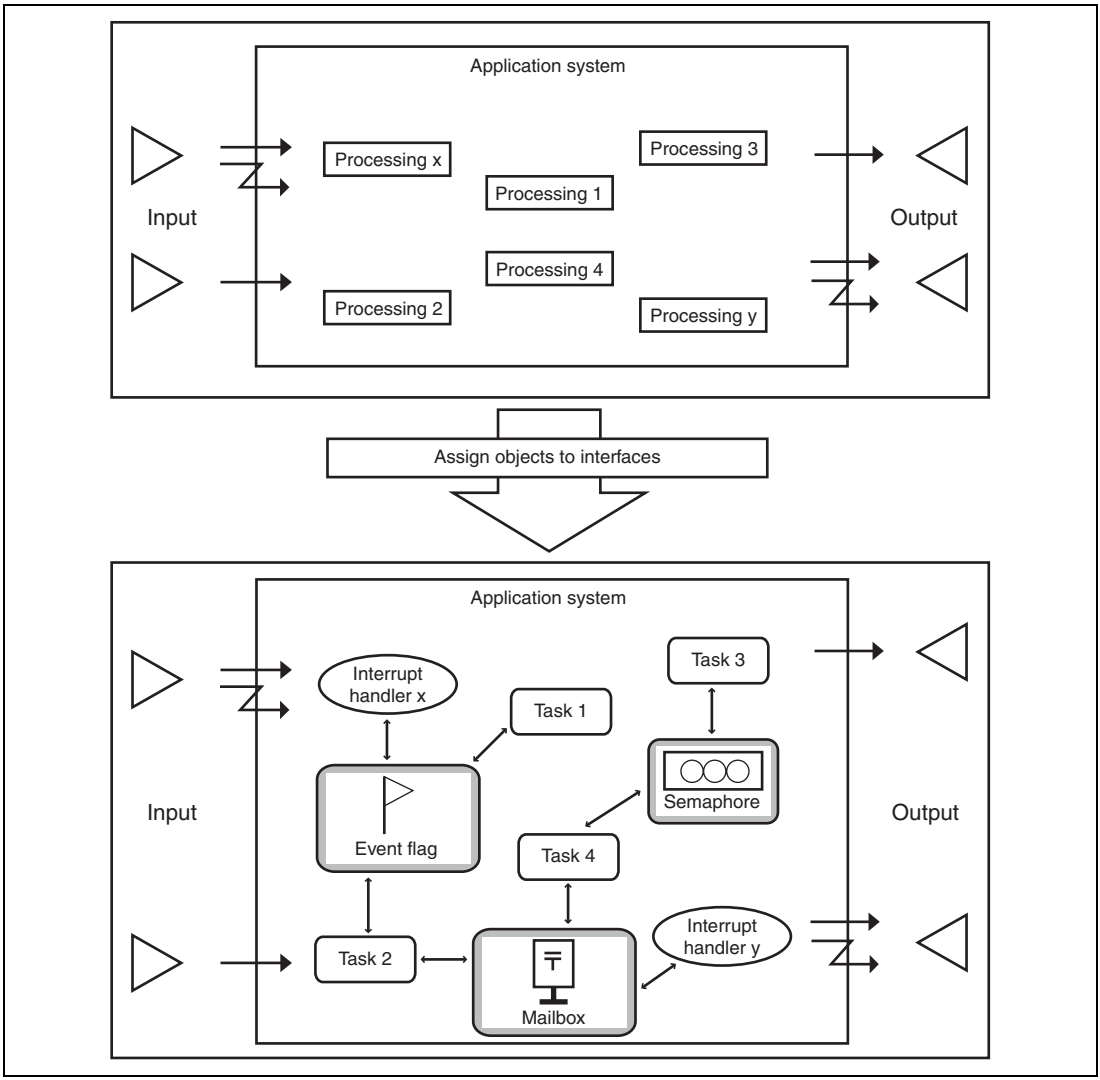


Figure 2.42 Example of ITRON Objects Assigned to Interfaces

These steps embed the HI series OS into the existing product's application programs that do not include RTOS.

Section 3 Configuration

3.1 Configuration Procedure Outline

The procedure for configuring a system using the HI series OS is described below.

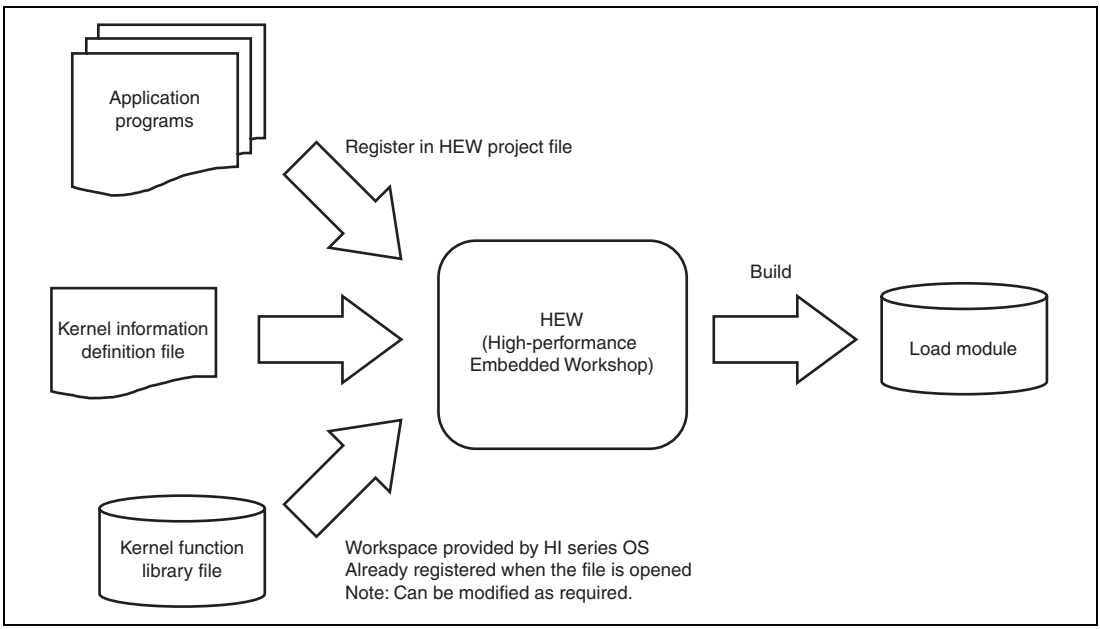


Figure 3.1 Configuration Procedure Outline

System configuration is to create, by means of the HEW, a load module from the user-created application programs, kernel information definition file (setup table or configuration file), and kernel function library file provided by the HI series OS.

For details of the user-created application programs, refer to section 2.6, Application Program Types.

For details of the kernel information definition file (setup table or configuration file), refer to section 3.2, Defining Kernel Environment.

For details of the kernel function library file provided by the HI series OS, refer to the user's manual of the HI series OS used.

For details of the HEW, refer to the online help of the compiler package used or the user's manual.

A system can be configured in the following two modes.

Table 3.1 System Configuration Modes

Configuration Mode	Overview	Supporting OS
Whole linkage*[1]	The kernel, configuration file, and application programs are linked into a single load module (called a "whole load module").	HI7000/4 series, HI2000/3, and HI1000/4
Separate linkage*[2]	The code portion and data portion of the kernel are linked into separate load modules.	HI7000/4 series
	The code portion of the kernel is called the "kernel load module", and the linkage unit for the kernel load module is called the "kernel side".	
	The data portion of the kernel is called the "kernel environment load module", and the linkage unit for the kernel environment load module is called the "kernel environment side".	

Notes: 1. The application programs can be included in the whole load module or can be linked into another load module (called the "application load module").
2. The application programs can be included in the kernel load module or kernel environment load module, or can be linked into another application load module.

The outlines of whole linkage and separate linkage are shown in figures 3.2 and 3.3, respectively.

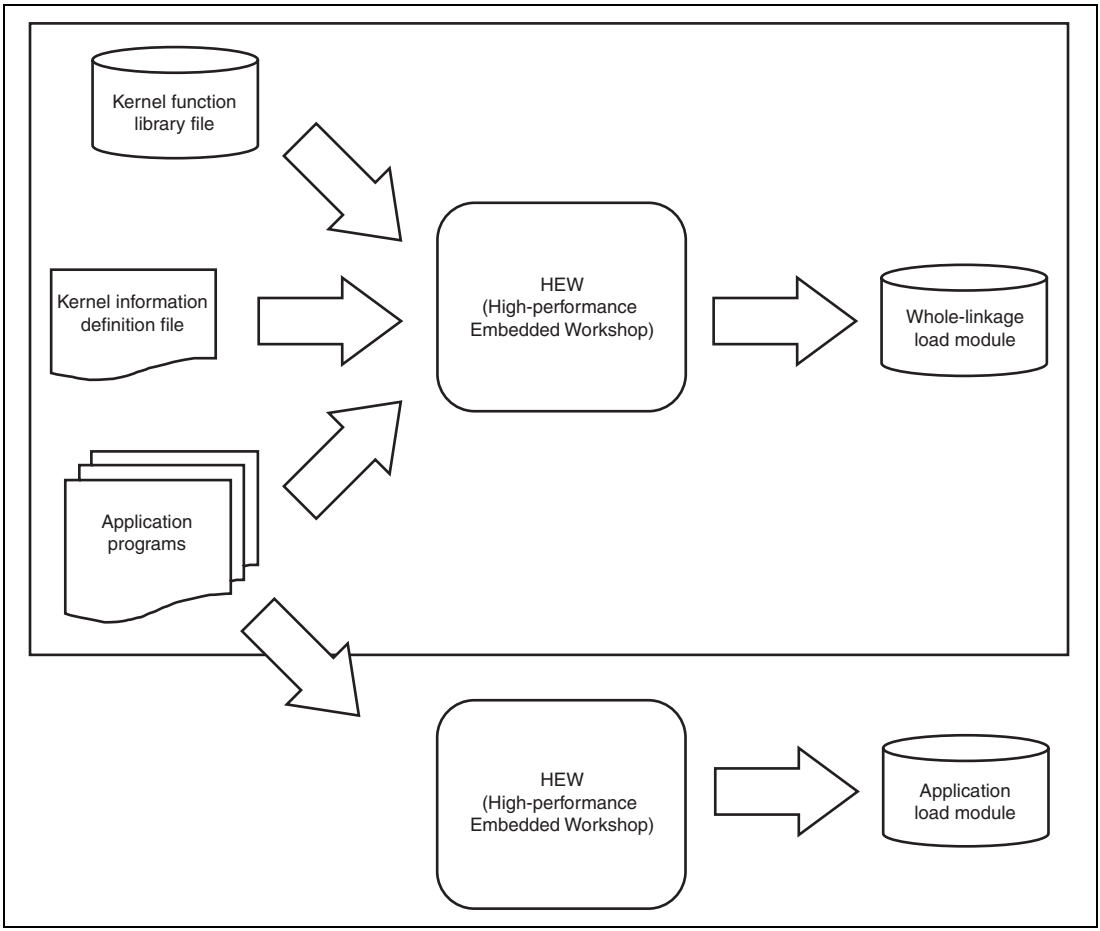
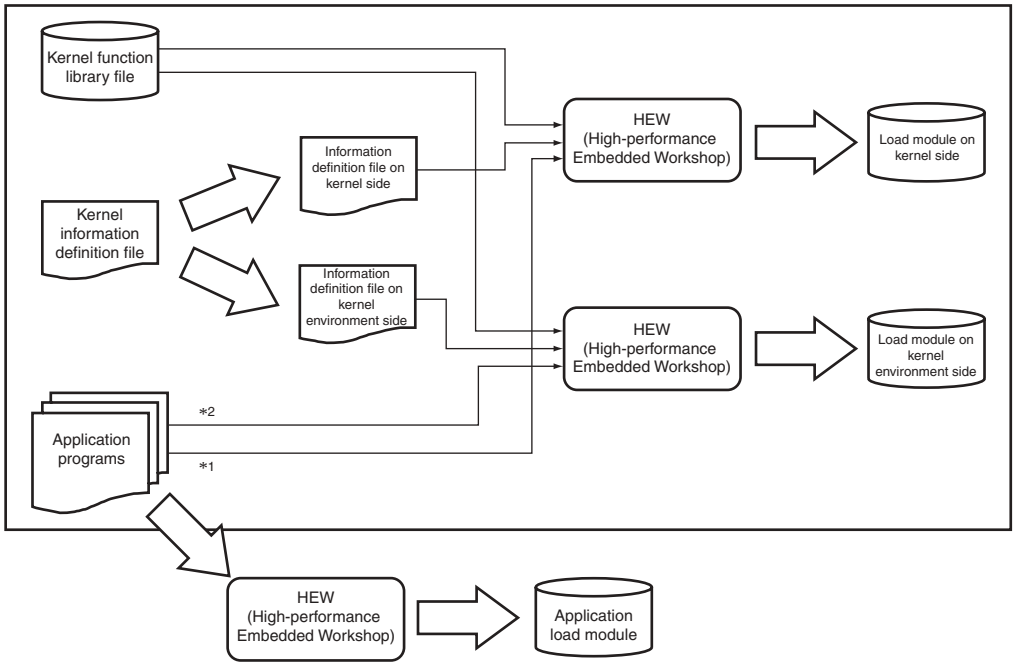


Figure 3.2 Whole Linkage Outline



Notes: 1. The application programs specified to be on the kernel side in the configurator are necessary.
2. The application programs specified to be on the kernel environment side in the configurator are necessary.

Figure 3.3 Separate Linkage Outline

The advantages and disadvantages of separate linkage, compared to whole linkage, are listed below.

Advantages:

- Since a load module can be created with only the kernel, the load module does not need to be re-created every time an application file or kernel environment file is changed.
- Even after the kernel load module is saved in ROM, the kernel environment load module can be re-created by changing configuration parameters, such as the maximum task ID (CFG_MAXTSKID) without updating the kernel load module.

Disadvantages:

- Since the kernel references the kernel environment file information during operation, the address where to locate the kernel environment file information needs be determined in advance and this address has to be defined at linkage.
- The above address cannot be changed unless the kernel load module is re-linked.

3.2 Defining Kernel Environment

The kernel environment can be defined by two methods: setup table and configurator.

- HI7000/4 series and HI1000/4: Kernel environment is defined by the configurator
- HI2000/3: Kernel environment is defined by the setup table

Each definition method is described in the following sections.

3.2.1 Definition by Configurator (HI7000/4 Series and HI1000/4)

Table 3.2 lists the files output from the configurator (kernel environment definition files; hereafter referred to as the configuration files).

Table 3.2 Files Output from Configurator (HI7000/4 Series)

No.	File Name	Contents	Remarks
1	kernel_def_main.h	Kernel function definition, such as embedded service calls	
2	kernel_def_inidata.def	Object initial definition data on the kernel load module side	
3	kernel_def_vct.inc	Vector information (written in assembly language)	HI7000/4 only
4	kernel_cfg_main.h	Kernel environment information definition, such as maximum task ID	
5	kernel_cfg_inidata.def	Object initial definition data on the kernel environment load module side	
6	kernel_id.h	Automatic ID assignment result corresponding to kernel_cfg_inidata.def	
7	kernel_macro.h	Header file defining kernel configuration macro	

Table 3.3 Files Output from Configurator (HI1000/4)

No.	File Name	Contents	Remarks
1	kernel_setup.src	Setup file	
2	kernel_id.h	Header file with automatic ID assignment result	For C language
	kernel_id.inc	Header file with automatic ID assignment result	For assembly language
3	kernel_macro.h	Header file defining kernel configuration constants	For C language
	kernel_macro.inc	Header file defining kernel configuration constants	For assembly language
4	kernel_sysini.src	File defining system initialization routine	
5	kernel_vector.src	File defining vector table creation information	

For details of the above files, refer to the HI7000/4 Series User's Manual or the HI1000/4 User's Manual for the HI series OS used.

(1) Overview of configurator operations

This section describes the construction and operations of the configurator with HI7000/4 as an example.

(a) Configurator window

The initiation window of the configurator is shown in figure 3.4.

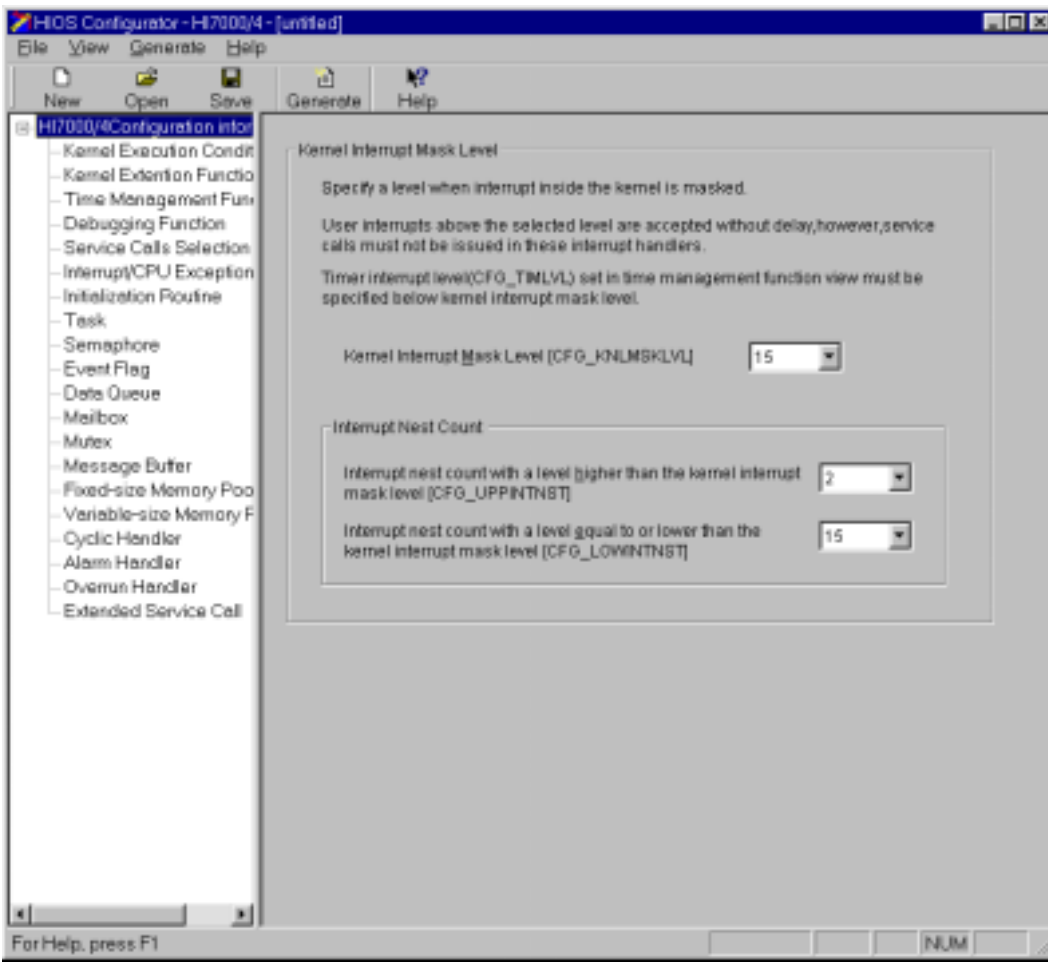


Figure 3.4 Configurator Initiation

The configurator construction is as follows:

- Left side of window: Configuration information view list window
- Right side of window: Configuration information input window

The initiation window of the configurator is different for each HI series OS. For details, refer to the online help of the configurator.

(b) Saving configurator information

After necessary definitions by the configurator are completed, save the registered contents and create configuration files. [Save] and [Generate] in the configurator header menu function as shown in table 3.4.

Table 3.4 [Save] and [Generate] Contents of Configurator

Button	Contents
[Save] 	Creates a file with extension hcf in the sample folder which saves the definitions made by the configurator.
[Generate] 	Creates the configuration files based on the definitions made by the configurator.

After definitions are modified by the configurator, be sure to perform the following:

— Update the definitions by using [Save].

— Make the configuration files reflect the modifications by using [Generate].

(c) Operating configurator definitions

How to operate the definitions is described below using “Task” in the configuration information view list window as an example.

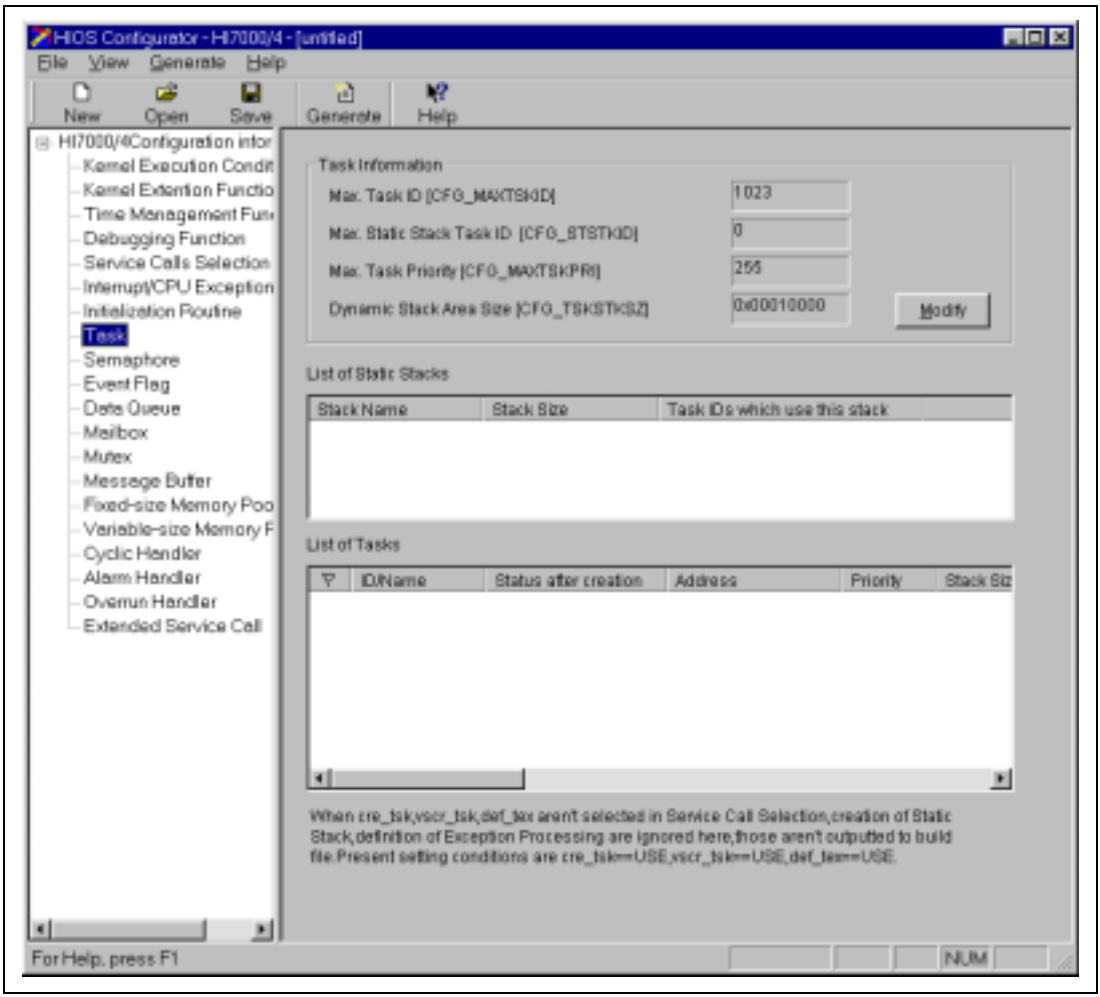


Figure 3.5 Task View

The Task View is a window for inputting various information and creating/deleting tasks. The contents displayed in the configuration information input window in the Task View are listed in table 3.5.

Table 3.5 Contents of Configuration Information Input Window in Task View

No.	Configuration Information Input Window	Contents
1	Task Information*[1]	The current definitions of the following items are displayed. • Maximum task ID • Maximum task priority • Maximum task ID using static stack • Dynamic stack area size
2	List of Static Stacks*[2]	The current definitions of the following items relevant to static stacks are displayed. • Stack area name • Stack area size • Task ID using stack area
3	List of Tasks*[2]	The current definitions of the following items relevant to tasks are displayed. • Linkage with kernel library enabled/disabled • Task ID/task name • Status after creation • Task start address • Initial task priority • Stack size/area • Description language • Coprocessor attribute • Extended information • Task exception processing routine definitions — Start address of task exception processing routine — Coprocessor attribute of task exception processing routine — Description language of task exception processing routine

Notes: 1. To modify the task information, click the [Modify] button to open the [Modification of Task Information] dialog box.
2. To modify [List of Static Stacks] or [List of Tasks], open the pop-up menu (displayed by right-clicking).

[Task Information] in the Task View is modified as shown below.

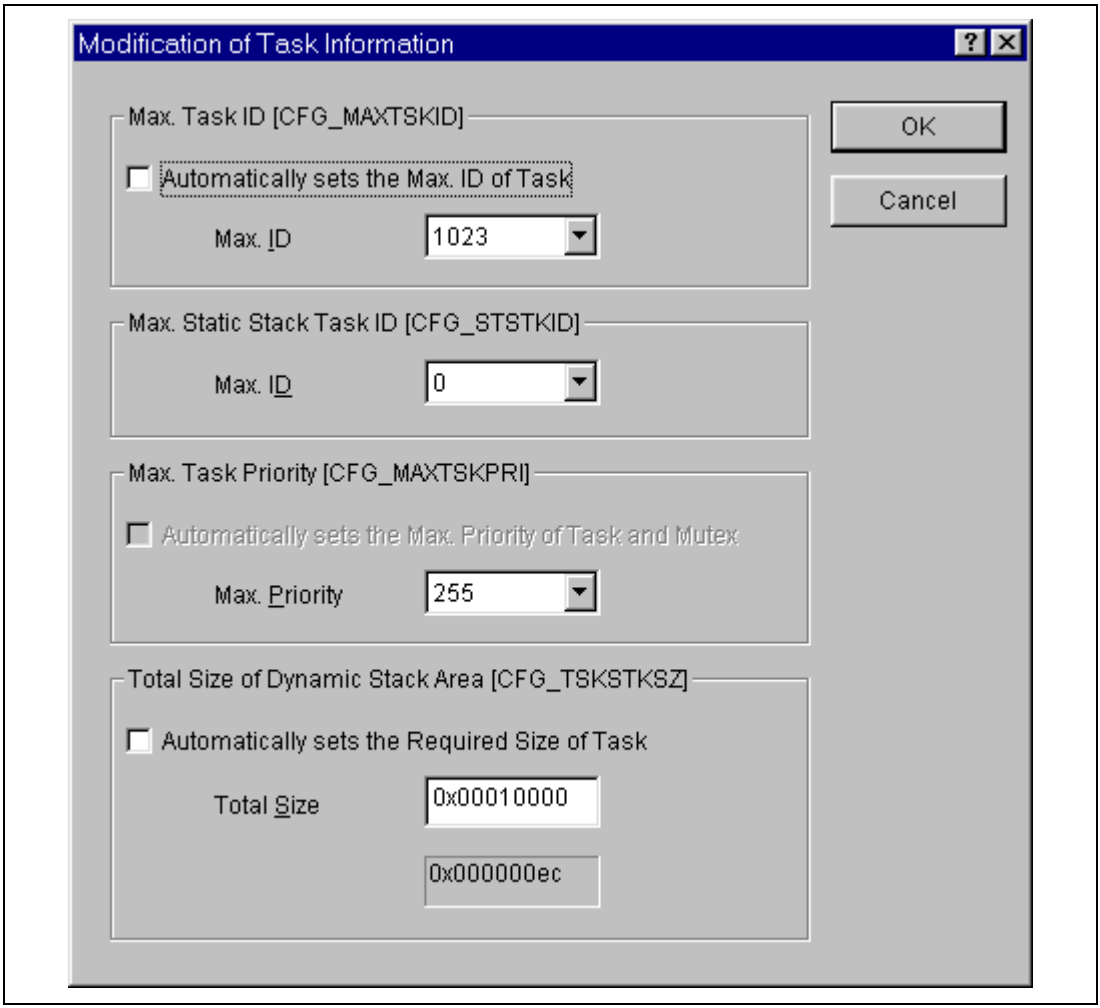


Figure 3.6 Modification of Task Information

Table 3.6 Contents of Task Information Modification

No.	Item	Displayed Contents
1	Max. Task ID	Maximum value of tasks registered in the system Setting methods: • Select [Automatically sets the Max. ID of Task]. The setting of the [Max. ID] box is ignored and the minimum value is automatically calculated in answer to the tasks created by the configurator. • Select from the pull-down menu of the [Max. ID] box.
2	Max. Static Stack Task ID	Maximum task ID among the tasks using the static stack Setting method: Select from the pull-down menu of the [Max. ID] box. Note: If the [OK] button is clicked with a value other than 0 specified, the [Definition of Stack Area] dialog box is opened.
3	Max. Task Priority	Maximum value of priorities assigned for the tasks registered in the system Setting method: Select from the pull-down menu of the [Max. Priority] box.
4	Total Size of Dynamic Stack Area	Total size of dynamic stack area Setting methods: • Select [Automatically sets the Required Size of Task]. The setting of the [Total Size] box is ignored and the minimum value is automatically calculated in answer to the tasks created by the configurator. • Input the total size of the dynamic stack area in the [Total Size] box. Note: The size displayed below the [Total Size] box is the value calculated from the size used by the tasks currently registered.

The [Definition of Stack Area] dialog box is described next.

When setting [Max. Static Stack Task ID], if the [OK] button is clicked with a value other than 0 specified, the [Definition of Stack Area] dialog box in figure 3.7 is opened.

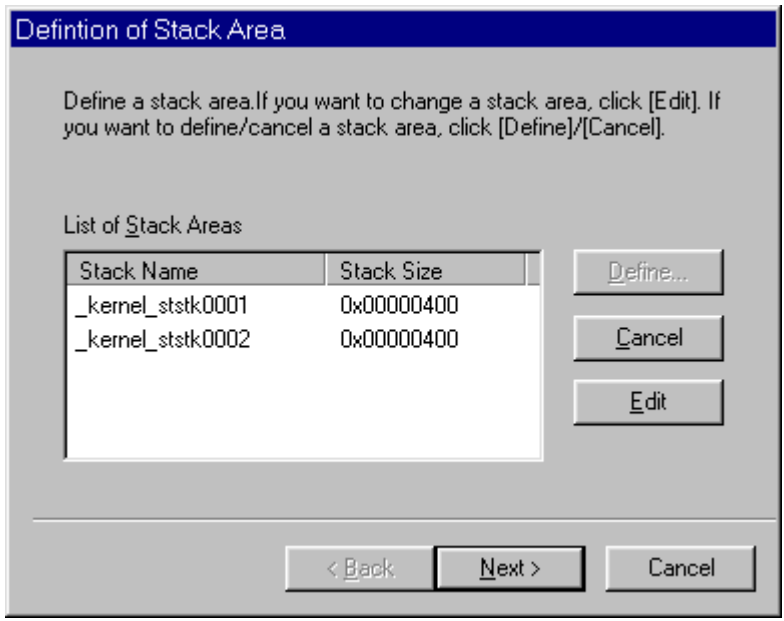


Figure 3.7 Definition of Stack Area

Clicking a stack displayed below [Stack Name] and then clicking the [Edit] button allows the stack size to be modified. The window for modification is shown in figure 3.8.

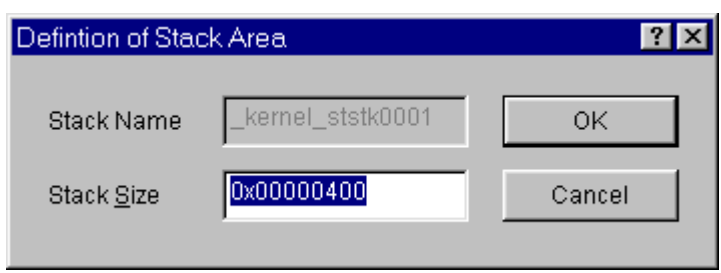


Figure 3.8 Modification of Static Stack Size

After entering the necessary size for the static stack area in [Stack Size], click the [OK] button for the modification to take effect.

On completing to set each static stack size, click the [Next >] button to define the task ID that uses each static stack. The [Task Registration] dialog box where the task ID is to be defined is shown in figure 3.9.

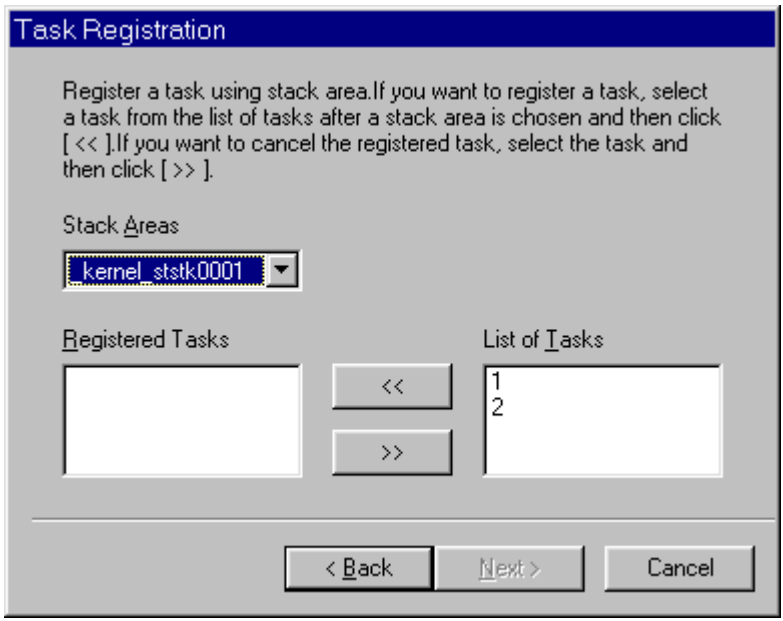


Figure 3.9 Registration of Task ID to Use Static Stack

Setting Procedure:

1. Select a static stack whose task ID is to be defined from the [Stack Areas] pull-down menu.
2. Select the task ID that uses the static stack selected in [Stack Areas] from [List of Tasks] and click the [<<] button to register it.

   Note: Registration can be cleared by selecting a task ID displayed in [Registered Tasks] and clicking the [>>] button.
3. When using the shared stack function, definition is done by registering more than one task ID to use the static stack selected in [Stack Areas].

After registration of the task IDs has finished for all static stacks, click the [Next >] button. The window in figure 3.10 is displayed to complete making settings relevant to static stacks.

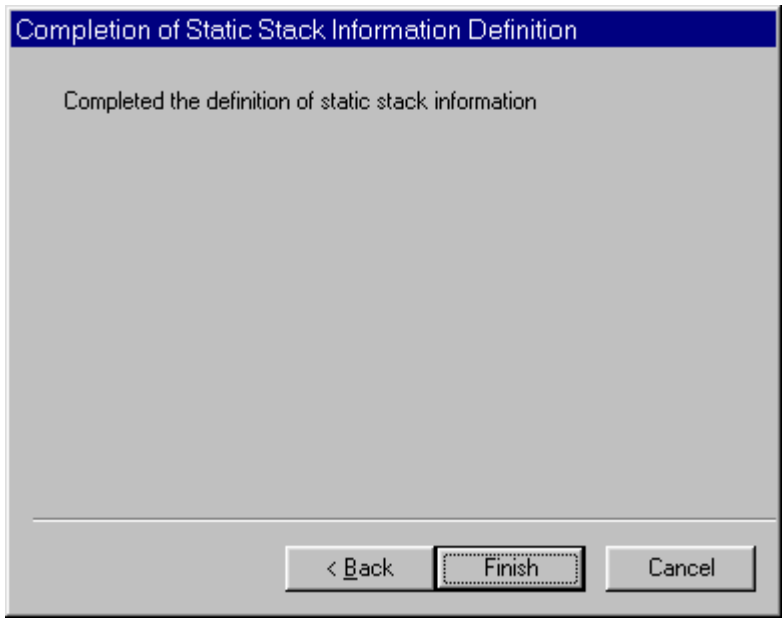


Figure 3.10 Completion of Static Stack Information Definition

Clicking the [Finish] button reflects the contents defined in [List of Static Stacks] in the Task View.

Modifying [List of Tasks] in the Task View is described next.

Modification is performed by selecting an item from the pop-up menu displayed by right-clicking in [List of Tasks]. The pop-up menu is shown in figure 3.11.

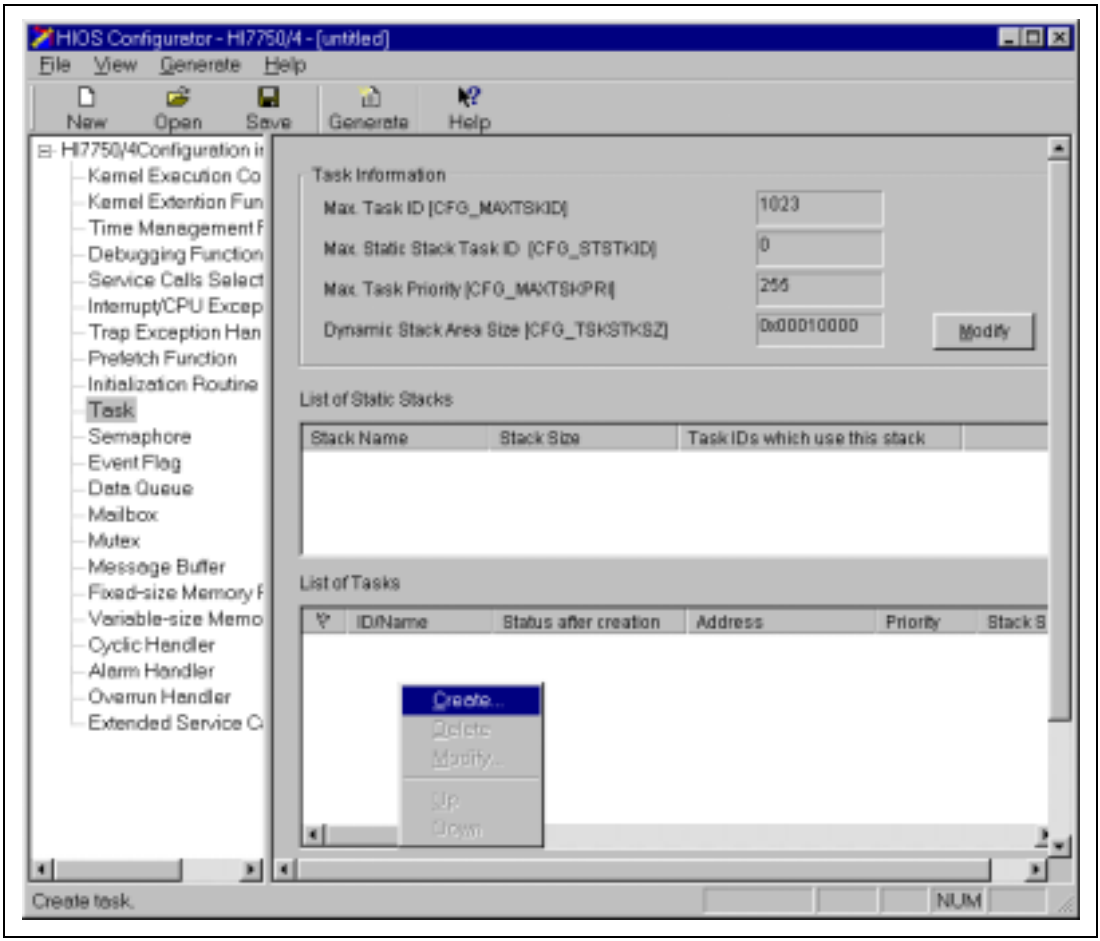


Figure 3.11 Pop-up Menu

Table 3.7 Pop-up Menu Contents

No.	Menu Item	Contents
1	Create	Opens the [Creation of Task] dialog box to define the contents for task creation.
2	Delete	Deletes the task creation information at the selected location.
3	Modify	Opens the [Modification of Task Information] dialog box to modify the creation information for the selected task.
4	Up	Selection moves up by one task.*
5	Down	Selection moves down by one task.*

Note: Since creation and initiation is processed in the display order, this is used for changing the creation order or initiation order at system activation.

When [Create] in the pop-up menu is selected, the [Creation of Task] dialog box is displayed. Settings in the [Creation of Task] dialog box are shown in table 3.8.

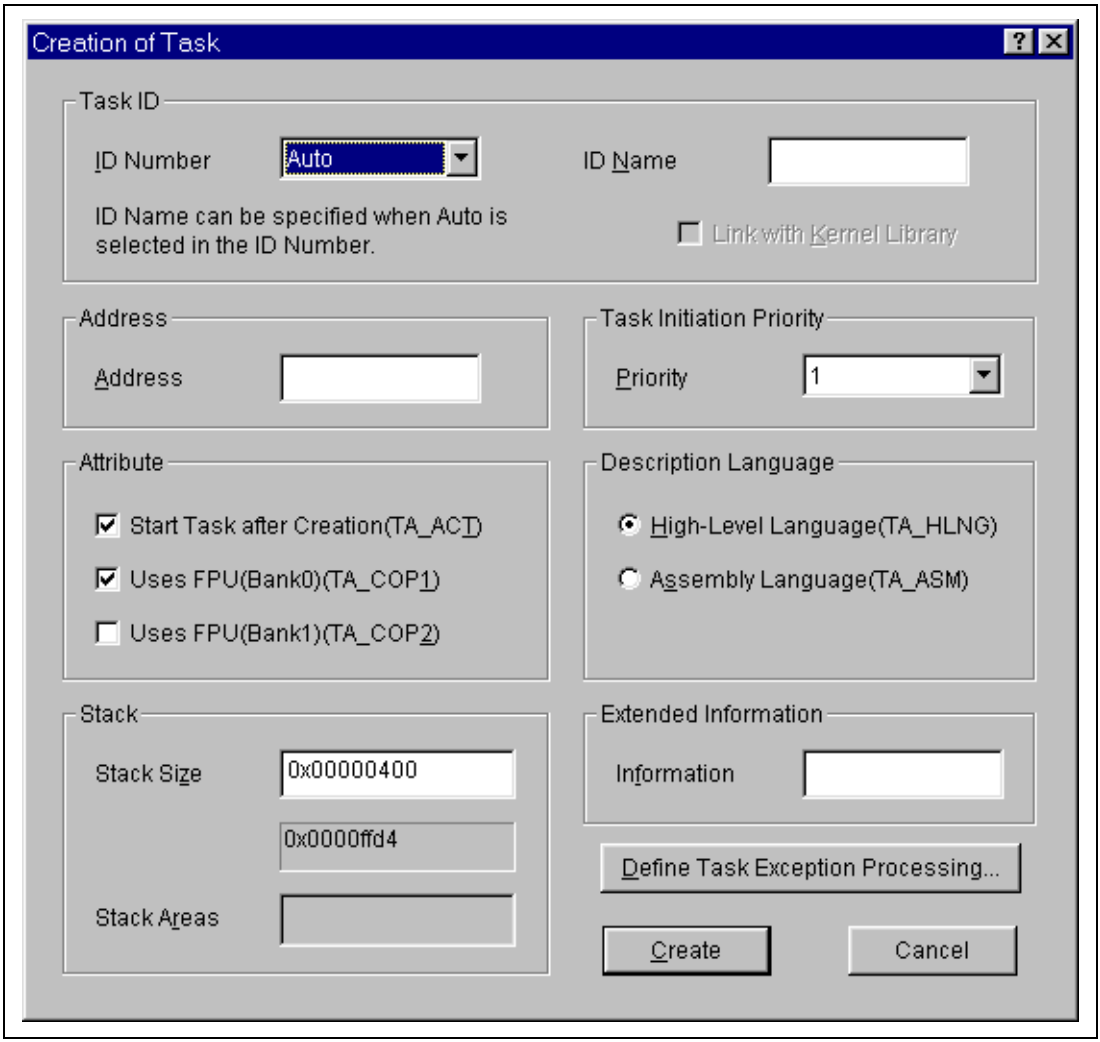


Figure 3.12 [Creation of Task] Dialog Box

Table 3.8 [Creation of Task] Dialog Box Contents

No.	Item	Contents
1	ID Number	Specify the ID number of the created task. Setting method: • When automatic ID assignment is specified, the configurator automatically assigns an unused ID when creating configuration files. • Select from the pull-down menu.
2	ID Name	When automatic ID assignment is specified, input the ID name of the created task.
3	Link with Kernel Library*[1]	Select the check box when the created task is to be linked with the kernel library.
4	Address	Input the start address of the created task as a symbol or numeric value.
5	Priority	Specify the priority when the created task is initiated.
6	Attribute*[2]	Specify the task state at creation. Setting method: When the task is to be created in the executable state, select the [Start Task after Creation (TA_ACT)] check box.
7	Description Language	Specify the description language for the created task. • Select [High-Level Language (TA_HLNG)] when the task is written in a high-level language. • Select [Assembly Language (TA_ASM)] when the task is written in assembly language.
8	Stack Size	Input the stack size the created task uses. Note: The size displayed below the [Stack Size] box is the specifiable size that was calculated from the remaining size of the dynamic stack area.
9	Stack Areas*[3]	The stack area used by the created task is displayed.
10	Extended Information	Input the extended information as a symbol or numeric value.

Notes:
1. Cannot be defined when automatic ID assignment is not selected in the [ID Number] box.
2. An item for defining the coprocessor attribute is also available. For details, refer to the online help of the configurator.
3. Displayed only when a task ID using the static stack has been specified in the [ID Number] box.

After making all settings in the [Creation of Task] dialog box, click the [Create] button to define them.

On completing definition for the task to be registered, click the [Cancel] button to finish definition.

To define a task exception processing routine for the created task, click the [Define Task Exception Processing...] button to display the [Definition of Task Exception Processing Routine] dialog box.

Settings in the [Definition of Task Exception Processing Routine] dialog box are shown in table 3.9.

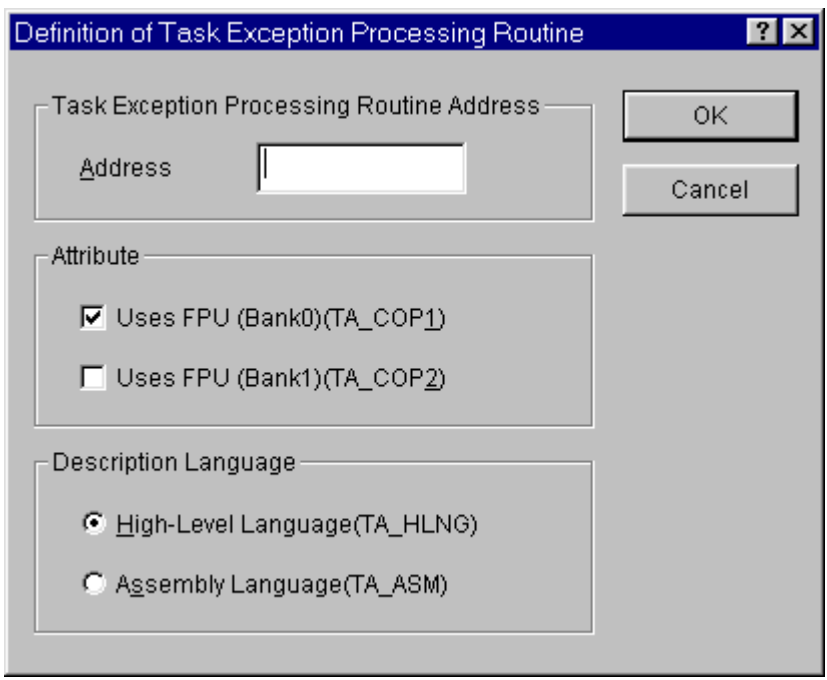


Figure 3.13 [Definition of Task Exception Processing Routine] Dialog Box

Table 3.9 [Definition of Task Exception Processing Routine] Dialog Box Contents

No.	Item	Contents
1	Address	Input the address of the task exception processing routine to be defined as a symbol or numeric value.
2	Attribute*	Select the coprocessor attribute to be used.
3	Description Language	Specify the description language for the created task. • Select [High-Level Language (TA_HLNG)] when the task is written in a high-level language. • Select [Assembly Language (TA_ASM)] when the task is written in assembly language.

Note: For details of the item relevant to defining the coprocessor attribute, refer to the online help of the configurator.

After making all settings in the [Definition of Task Exception Processing Routine] dialog box, click the [OK] button to define them.

The necessary information for the configurator is defined in this manner.

Next, each configuration information view of the configurator is described.

(2) Configuration information views of configurator

The initiation window is shown in figures 3.14 to 3.16.

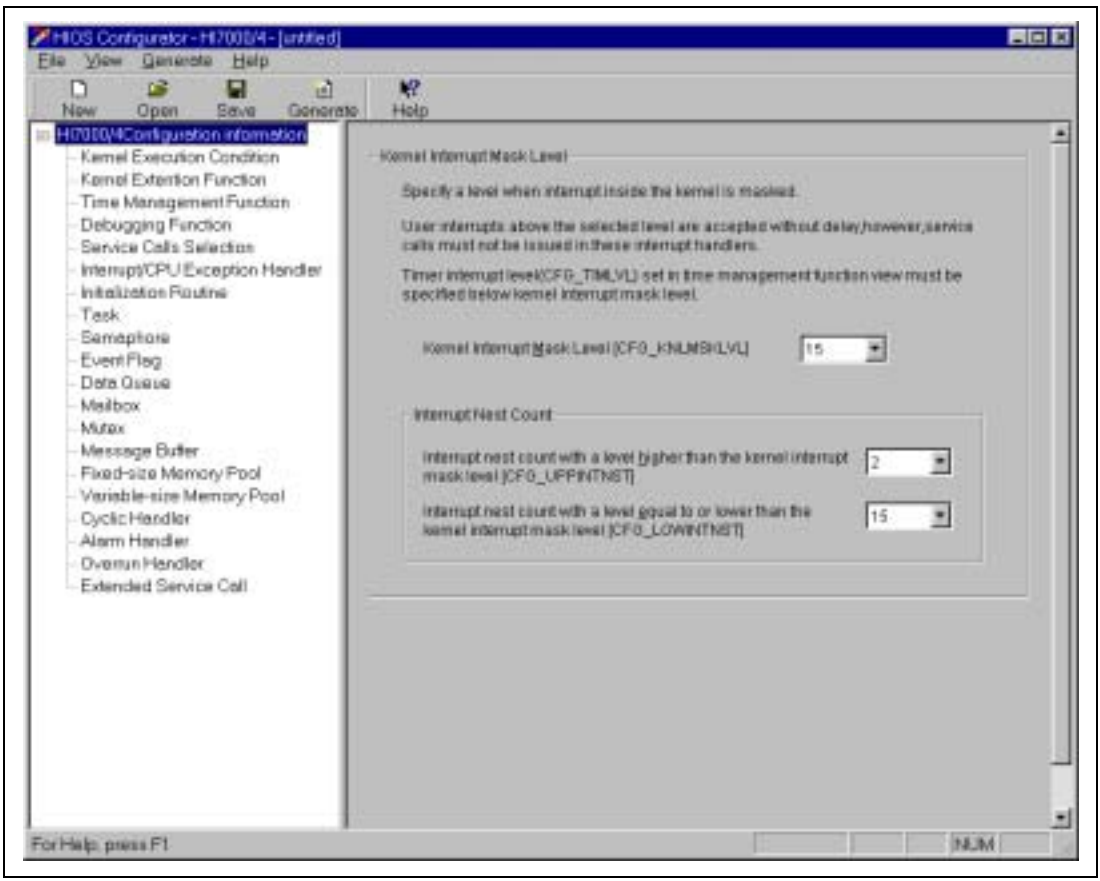


Figure 3.14 Configurator Initiation (HI7000/4)

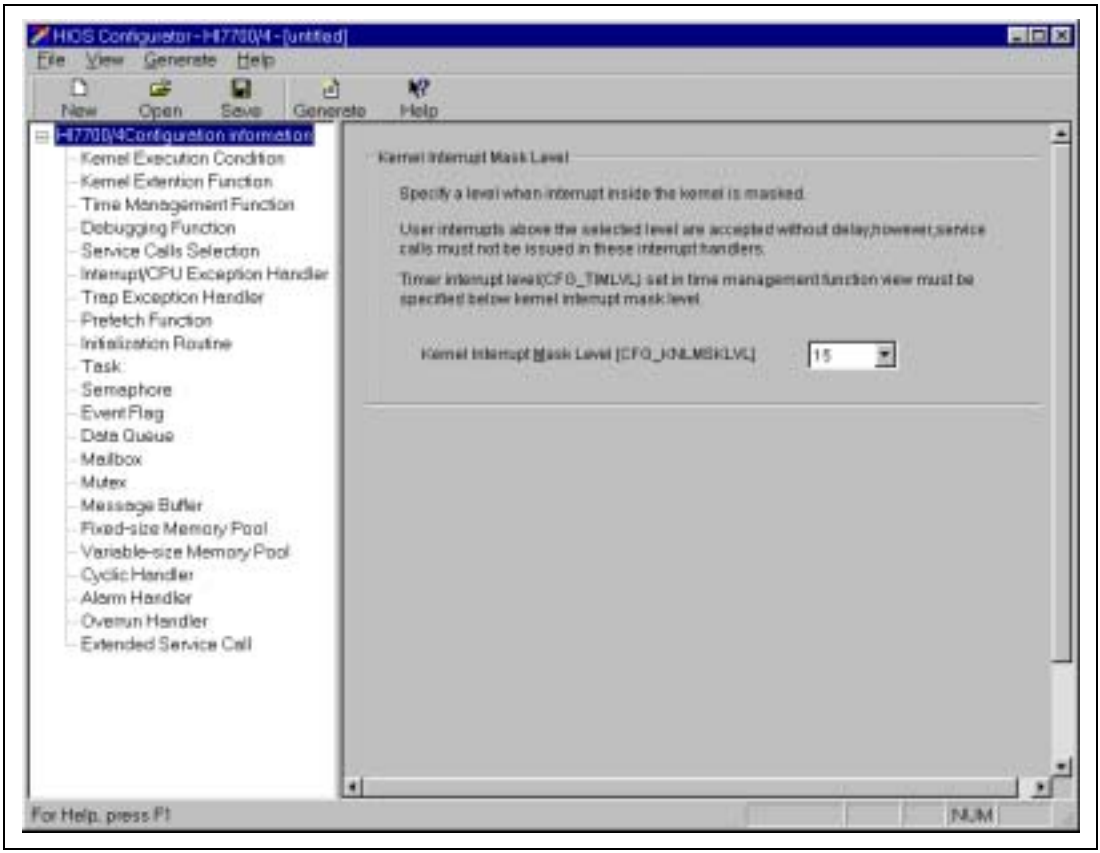


Figure 3.15 Configurator Initiation (HI7700/4 and HI7750/4)

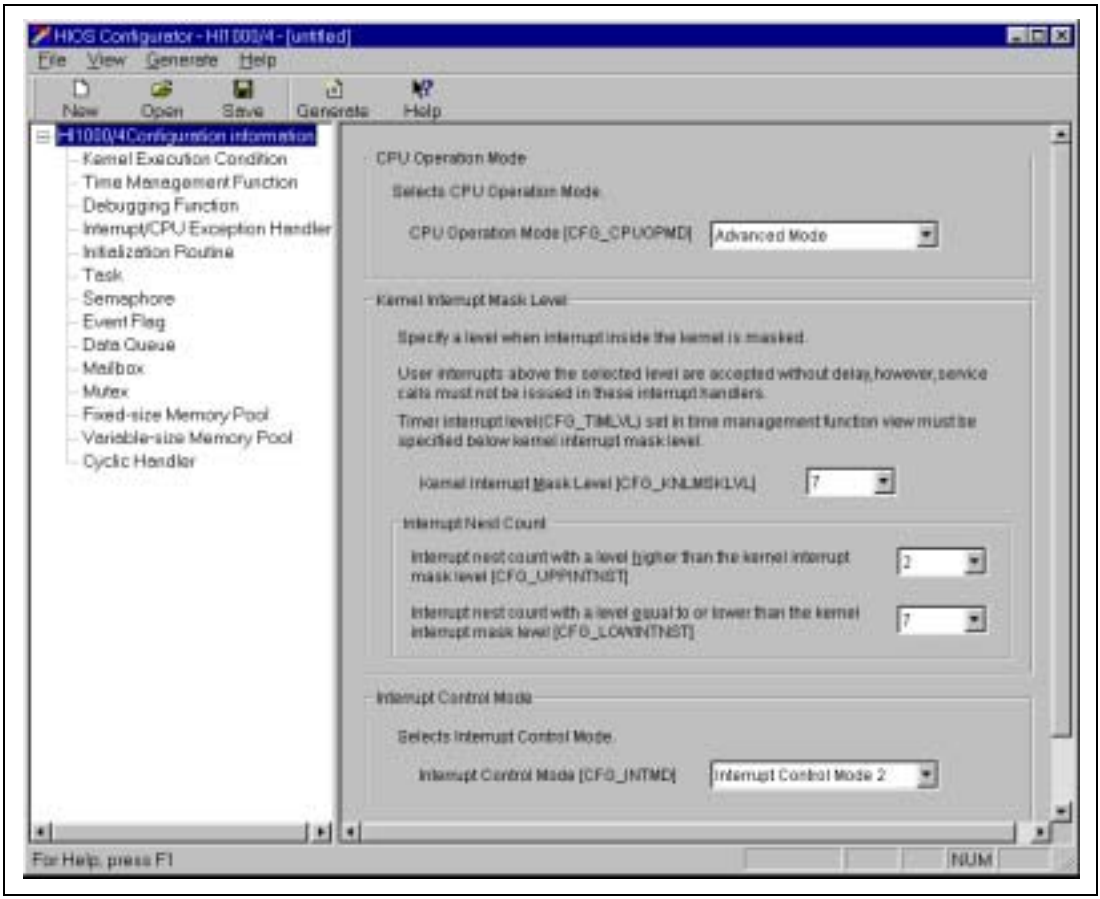


Figure 3.16 Configurator Initiation (HI1000/4)

The configurator consists of a configuration information view list window (on the left side), and a configuration information input window (on the right side).

(a) Kernel Execution Condition View

The initiation window is shared with the Kernel Execution Condition View.

The items to be set in the Kernel Execution Condition View are shown in table 3.10.

Table 3.10 Setting Items in Kernel Execution Condition View

No.	Menu Item	Contents	Target OS
1	Kernel Interrupt Mask Level	Define the mask level for masking interrupts inside the kernel.	HI7000/4, HI7700/4, HI7750/4, and HI1000/4
2	Interrupt Nest Count	Define [Interrupt nest count with a level higher than the kernel interrupt mask level] and [Interrupt nest count with a level equal to or lower than the kernel interrupt mask level].	HI7000/4 and HI1000/4
3	CPU Operation Mode	Select the CPU operating mode.	HI1000/4
4	Interrupt Control Mode	Select the interrupt control mode.	HI1000/4

Set the items by pressing the button prepared for each item to make a selection from the displayed pull-down menu.

(b) Kernel Extension Function View

The Kernel Extension Function View is shown in figures 3.17 and 3.18.

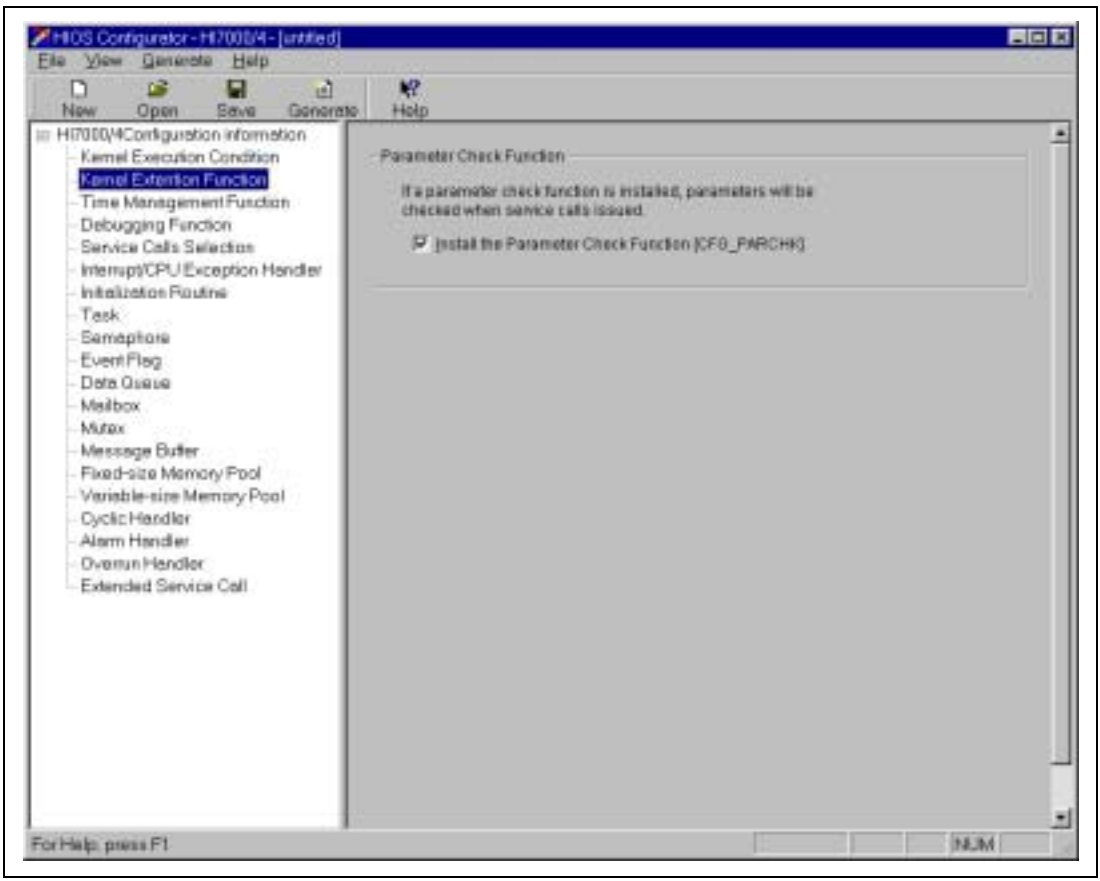


Figure 3.17 Kernel Extension Function View (HI7000/4)

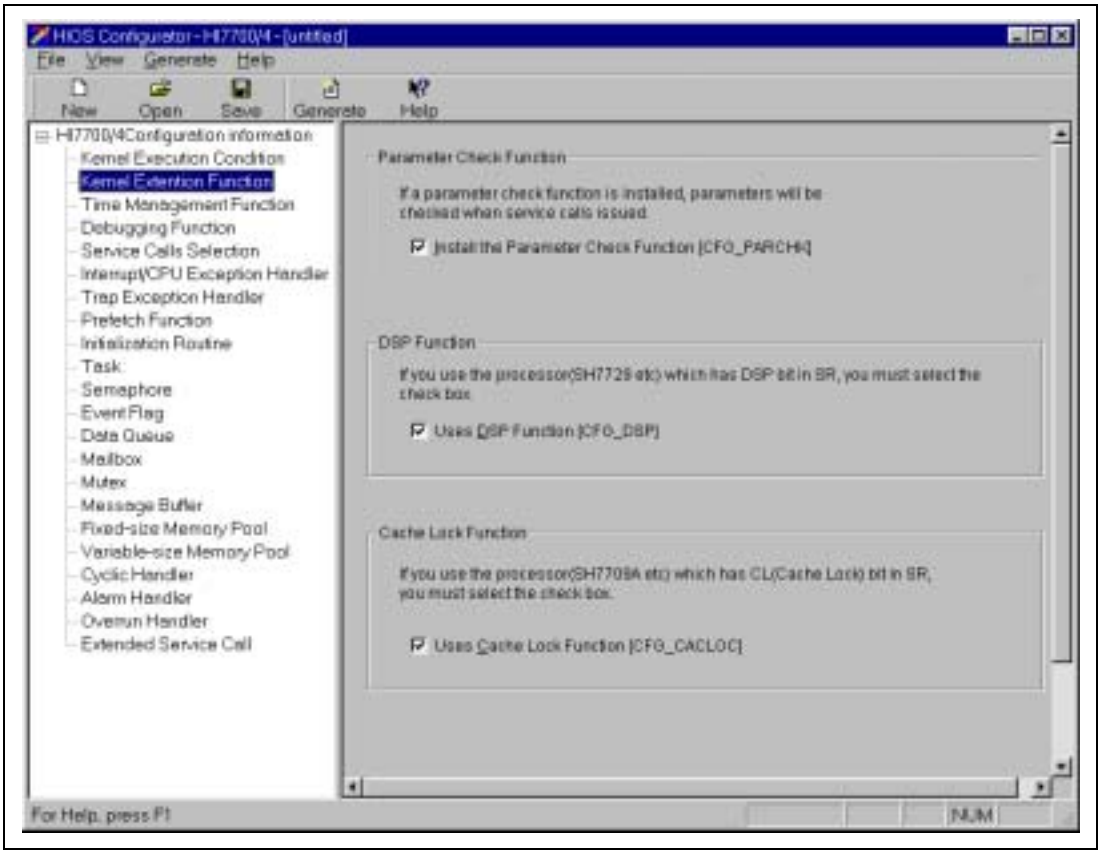


Figure 3.18 Kernel Extension Function View (HI7700/4 and HI7750/4)

The HI1000/4 configurator does not have the Kernel Extension Function View.

The items to be set in the Kernel Extension Function View are shown in table 3.11.

Table 3.11 Setting Items in Kernel Extension Function View

No.	Setting Item	Contents	Target OS
1	Parameter Check Function	Select when installing the parameter check function.	HI7000/4, HI7700/4, and HI7750/4
2	DSP Function*	Select when using the DSP function.	HI7700/4
3	Cache Lock Function*	Select when using the cache lock function.	HI7700/4

Note: Must be set when using a processor that has the DSP function or cache lock function.

Each setting is made by selecting the check box for each item.

(c) Time Management Function View

The Time Management Function View is shown in figures 3.19 and 3.20.

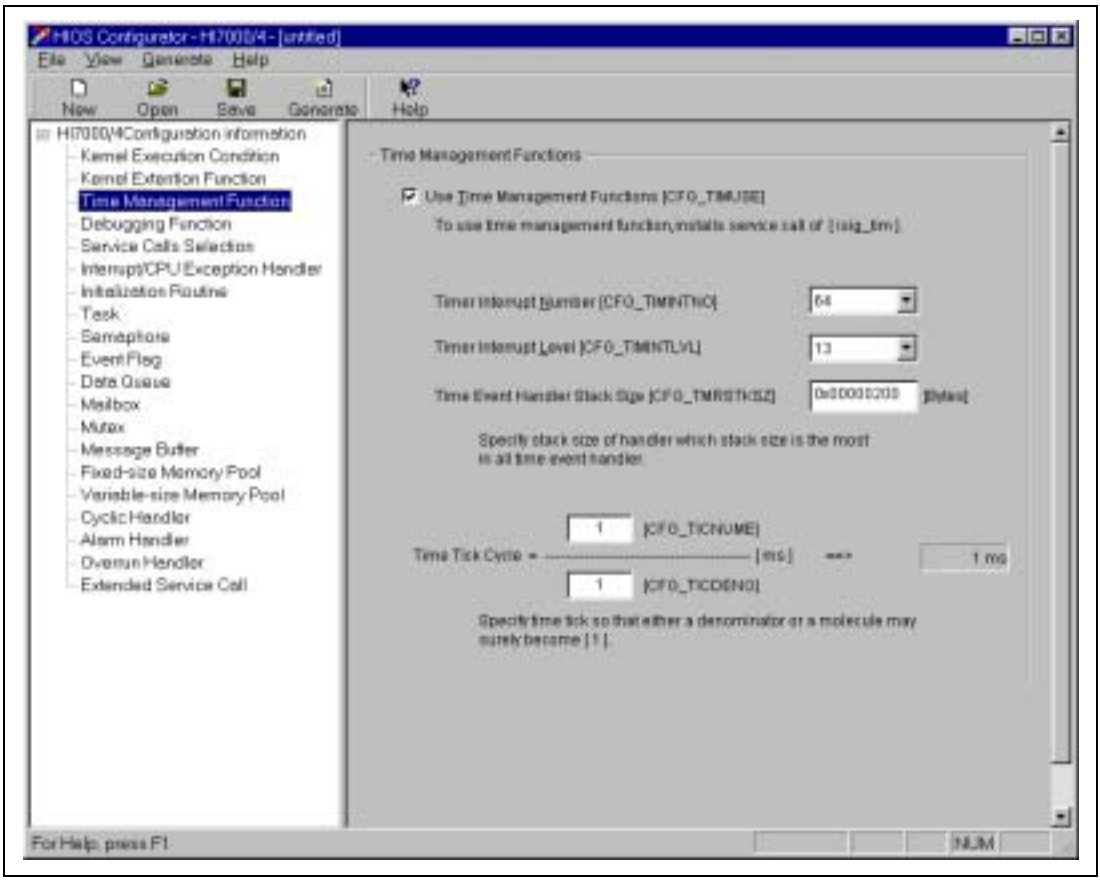


Figure 3.19 Time Management Function View (HI7000/4, HI7700/4, and HI7750/4)

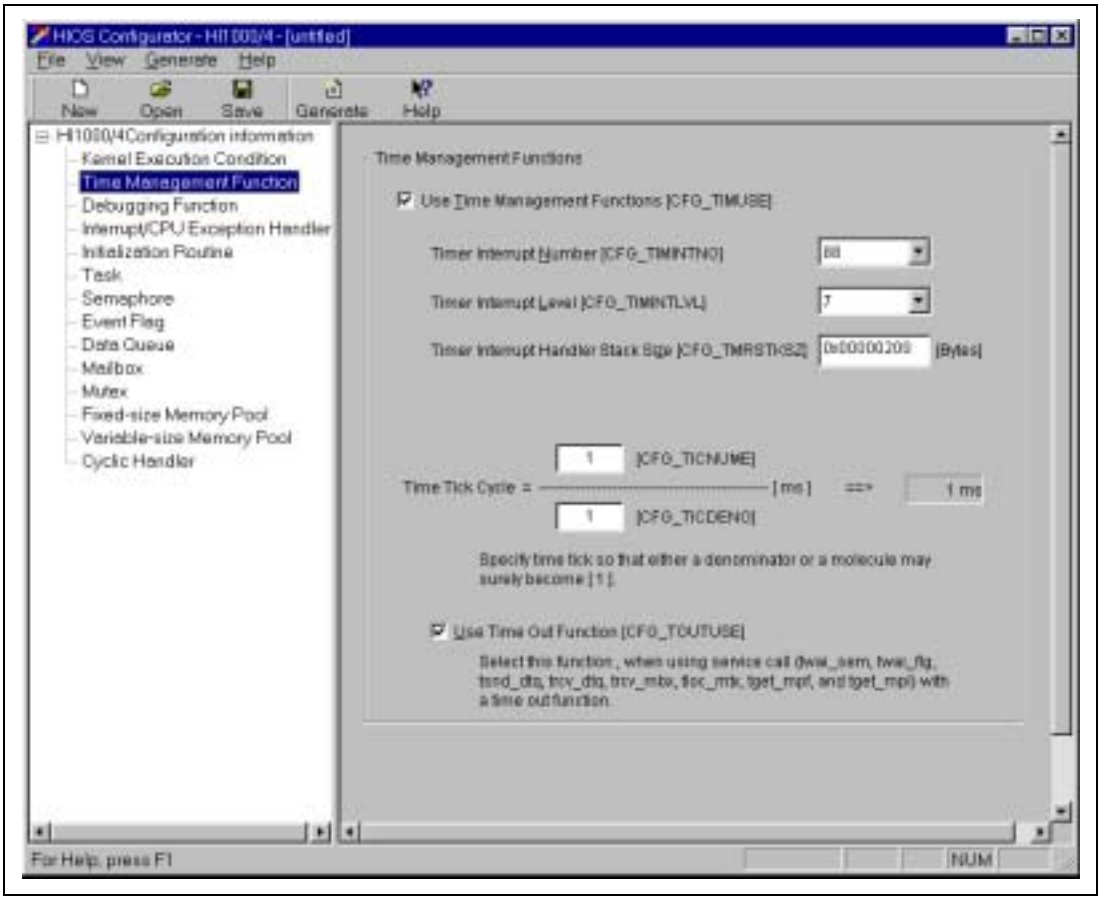


Figure 3.20 Time Management Function View (HI1000/4)

The items to be set in the Time Management Function View are shown in table 3.12.

Table 3.12 Setting Items in Time Management Function View

No.	Setting Item	Contents	Target OS
1	Time Management Functions	Select when installing the time management function.	HI7000/4, HI7700/4, HI7750/4, and HI1000/4
2	Timer Interrupt Number	Define the timer interrupt vector number (or INTEVT code).	HI7000/4, HI7700/4, HI7750/4, and HI1000/4
3	Timer Interrupt Level	Define the interrupt level of the timer interrupt.	HI7000/4, HI7700/4, HI7750/4, and HI1000/4
4	Time Event Handler Stack Size	Define the stack size used by the time event handler	HI7000/4, HI7700/4, and HI7750/4
	Timer Interrupt Handler Stack Size	Define the stack size used by the timer interrupt handler	HI1000/4
5	Time Tick Cycle	Define when changing the precision of the time tick supply cycle.	HI7000/4, HI7700/4, HI7750/4, and HI1000/4
6	Use Time Out Function	Select when using a service call with the timeout function	HI1000/4

Note: In the time tick cycle specification, either the numerator or denominator must be 1.

Set the items by pressing the button prepared for each item to make a selection from the displayed pull-down menu, or by directly entering a value for each item.

(d) Debugging Function View

The Debugging Function View is shown in figures 3.21 and 3.22.

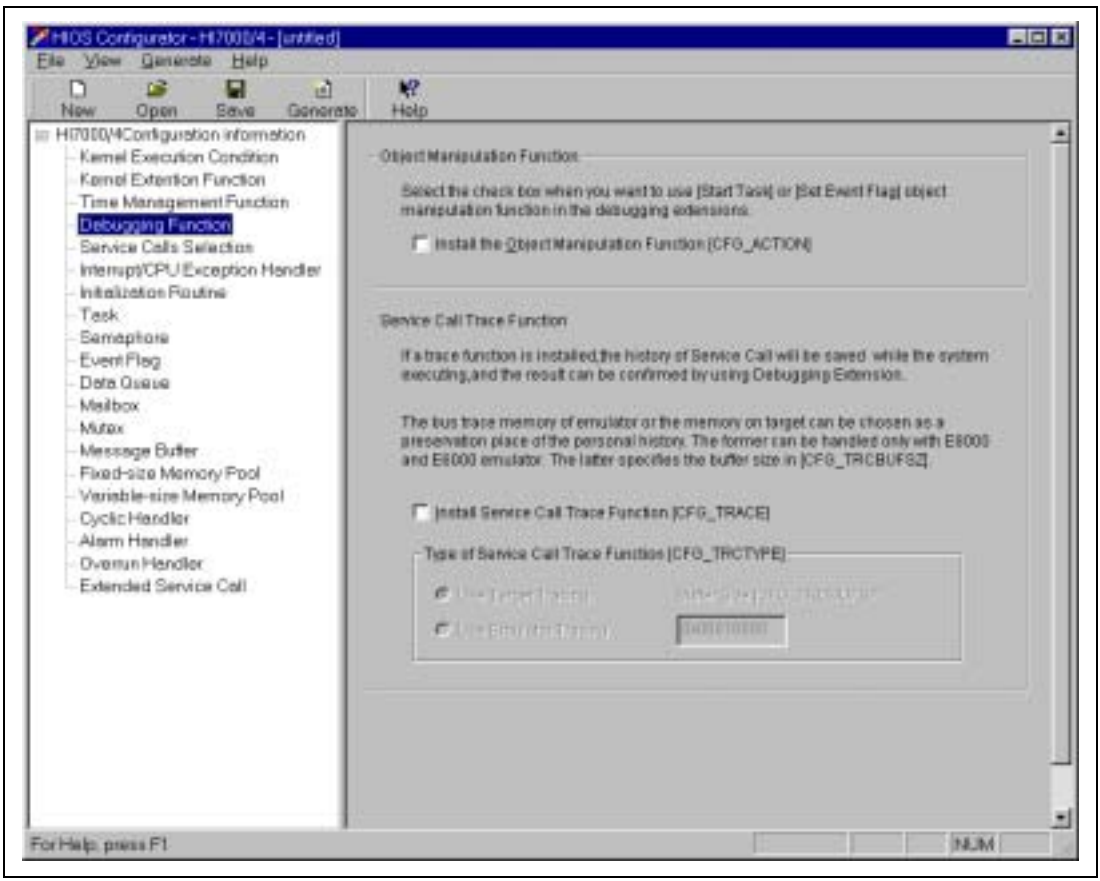


Figure 3.21 Debugging Function View (HI7000/4, HI7700/4, and HI7750/4)

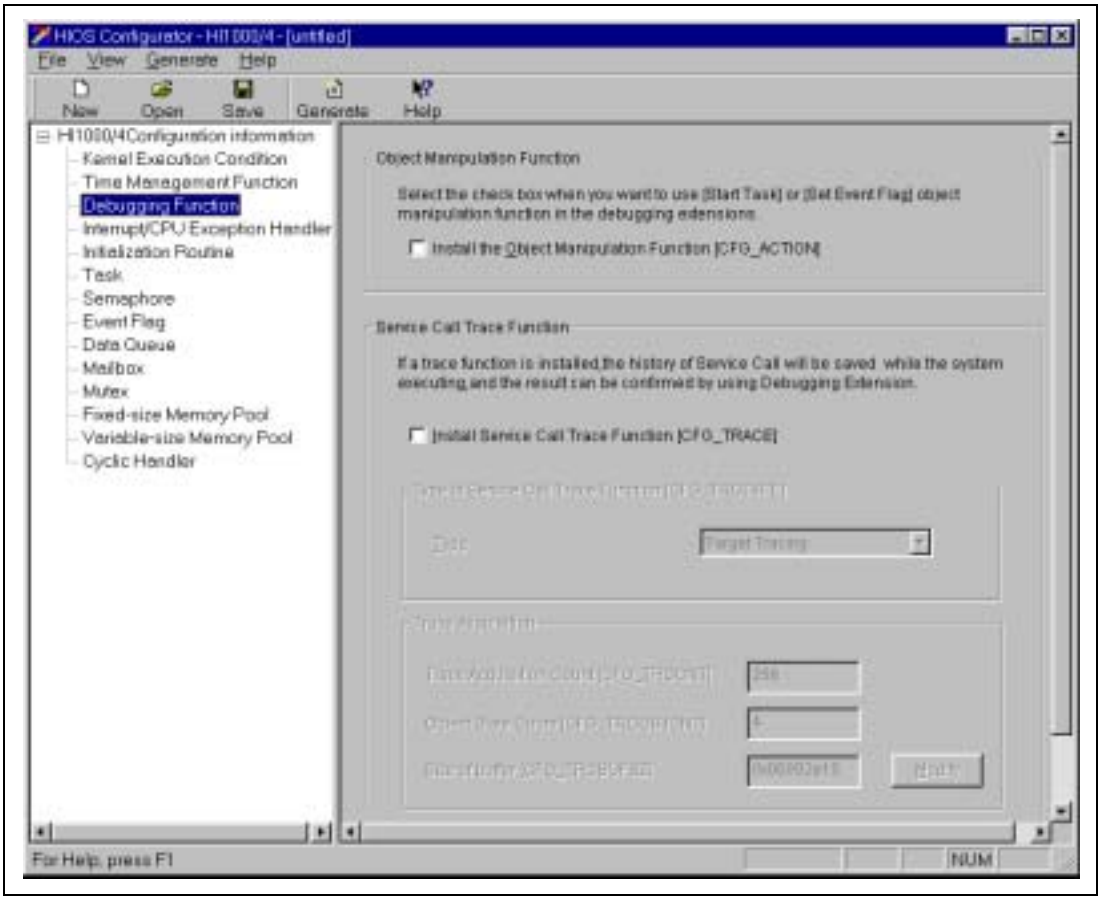


Figure 3.22 Debugging Function View (HI1000/4)

The items to be set in the Debugging Function View are shown in table 3.13.

Table 3.13 Setting Items in Debugging Function View

No.	Setting Item	Contents	Target OS
1	Object Manipulation Function	Select when using the object manipulation function, such as [Start Task] and [Set Event Flag], in the debugging extensions.	HI7000/4, HI7700/4, HI7750/4, and HI1000/4
2	Service Call Trace Function	Select when installing the service call trace function.	HI7000/4, HI7700/4, HI7750/4, and HI1000/4

Set the items by pressing the button prepared for each item to make a selection from the displayed pull-down menu, by selecting check boxes or radio boxes, or by directly entering a value for the necessary items.

(e) Service Calls Selection View

The Service Calls Selection View is shown in figure 3.23.

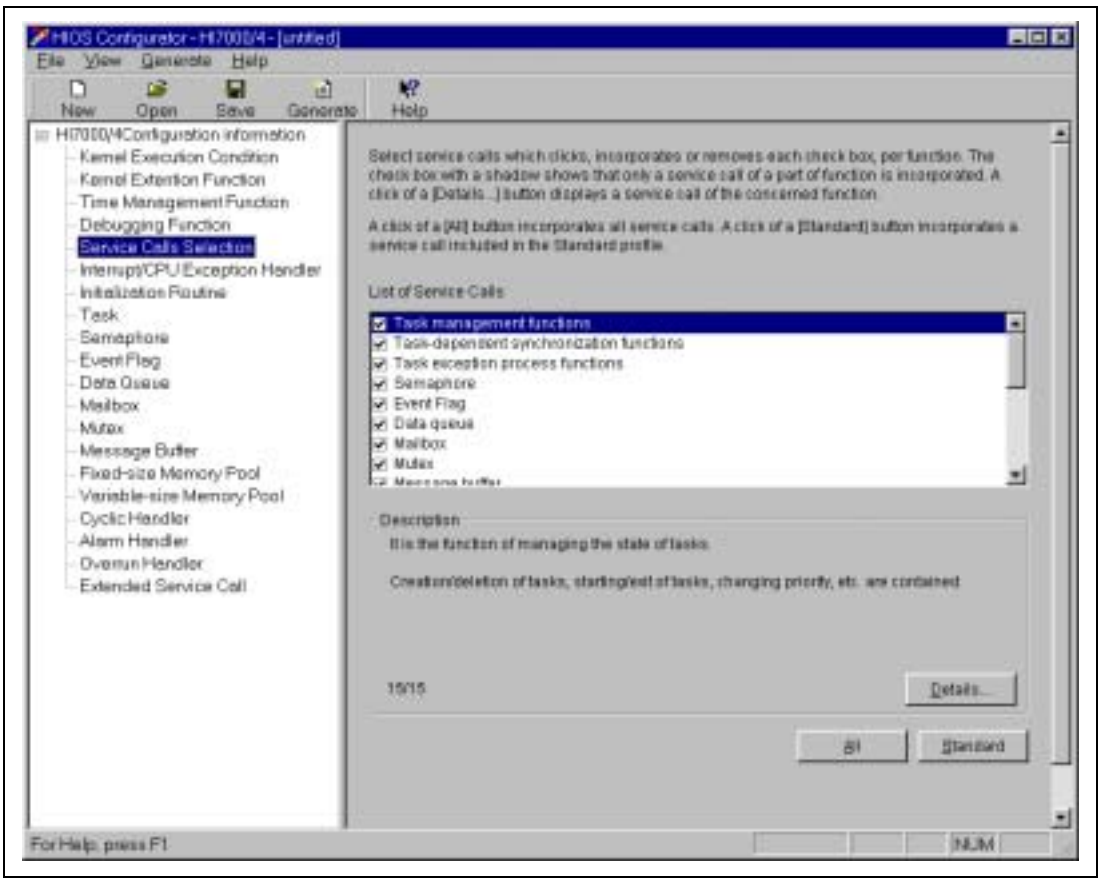


Figure 3.23 Service Calls Selection View (HI7000/4, HI7700/4, and HI7750/4)

The HI1000/4 configurator does not have the Service Calls Selection View.

In the Service Calls Selection View, the service calls to be embedded or removed can be selected in function units from the [List of Service Calls] dialog.

To select service calls in service call units, click the [Details...] button in the [Description] frame.

Clicking the [All] button embeds all service calls. Clicking the [Standard] button embeds only the service calls supported with the standard profile.

(f) Interrupt/CPU Exception Handler View

The Interrupt/CPU Exception Handler View is shown in figures 3.24 to 3.26.

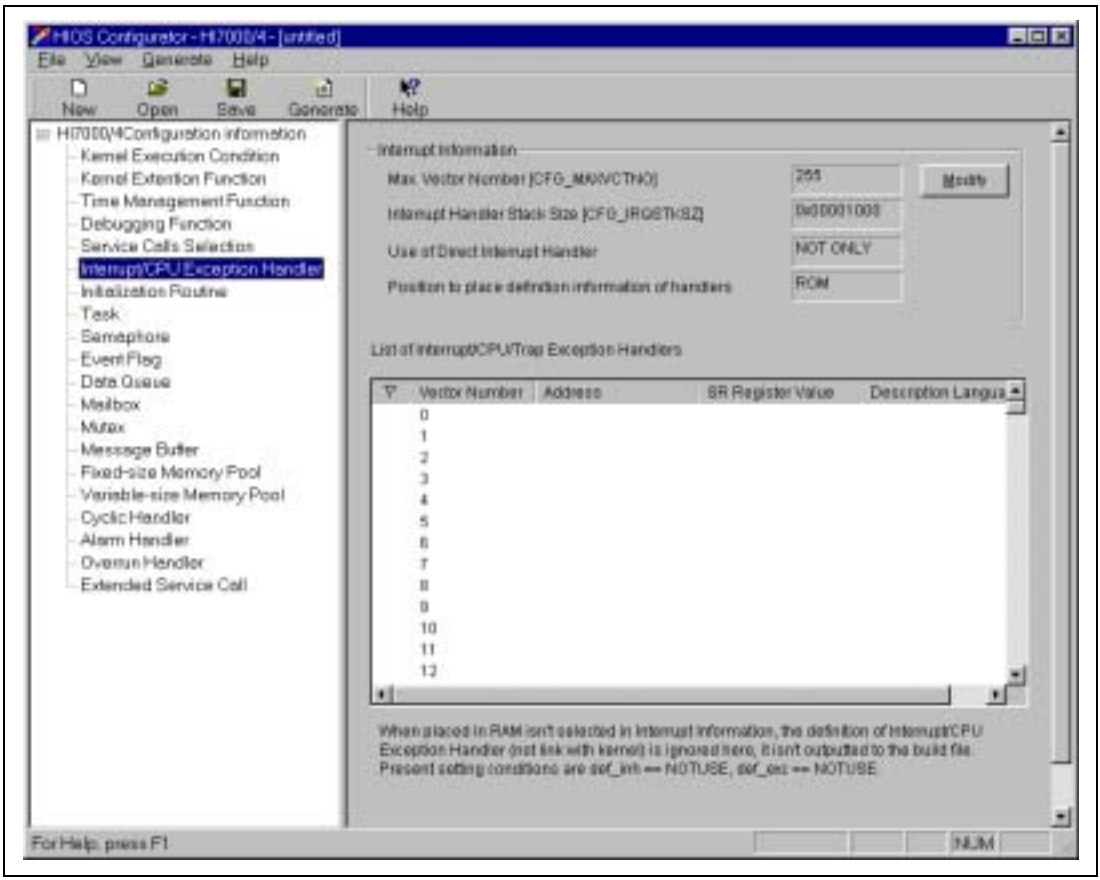


Figure 3.24 Interrupt/CPU Exception Handler View (HI7000/4)

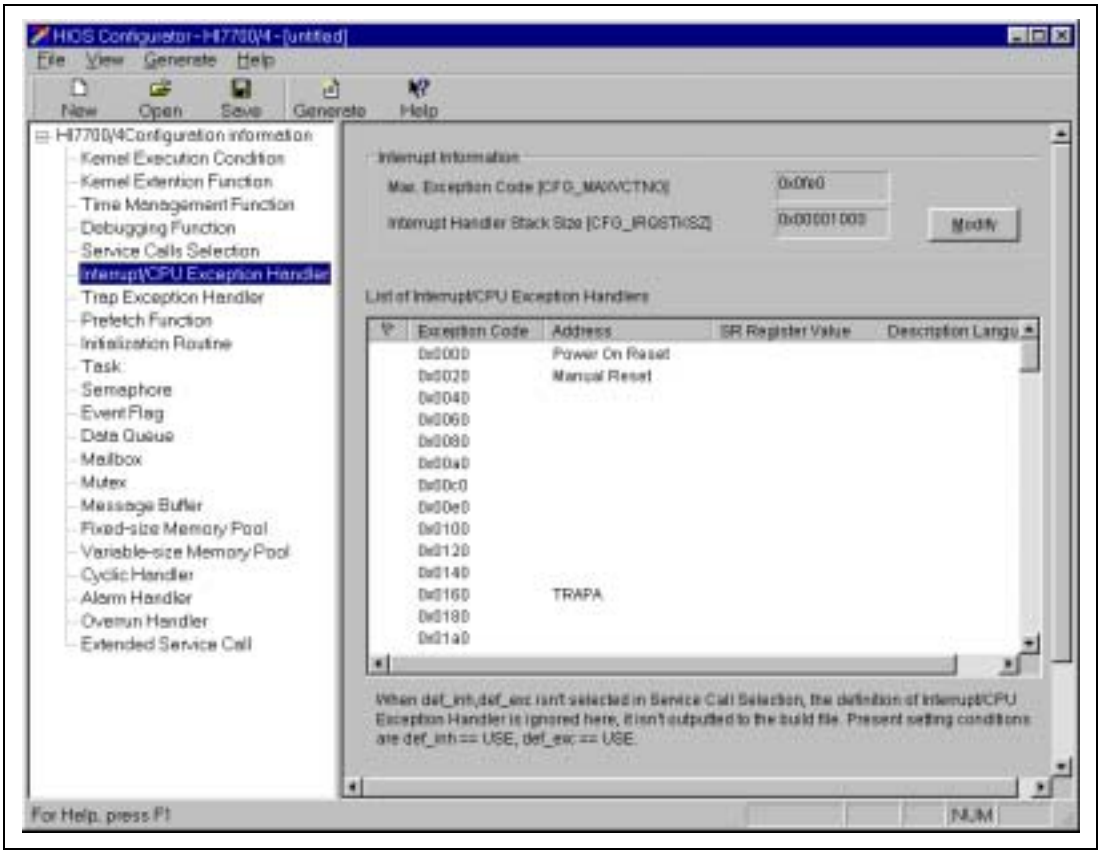


Figure 3.25 Interrupt/CPU Exception Handler View (HI7700/4 and HI7750/4)

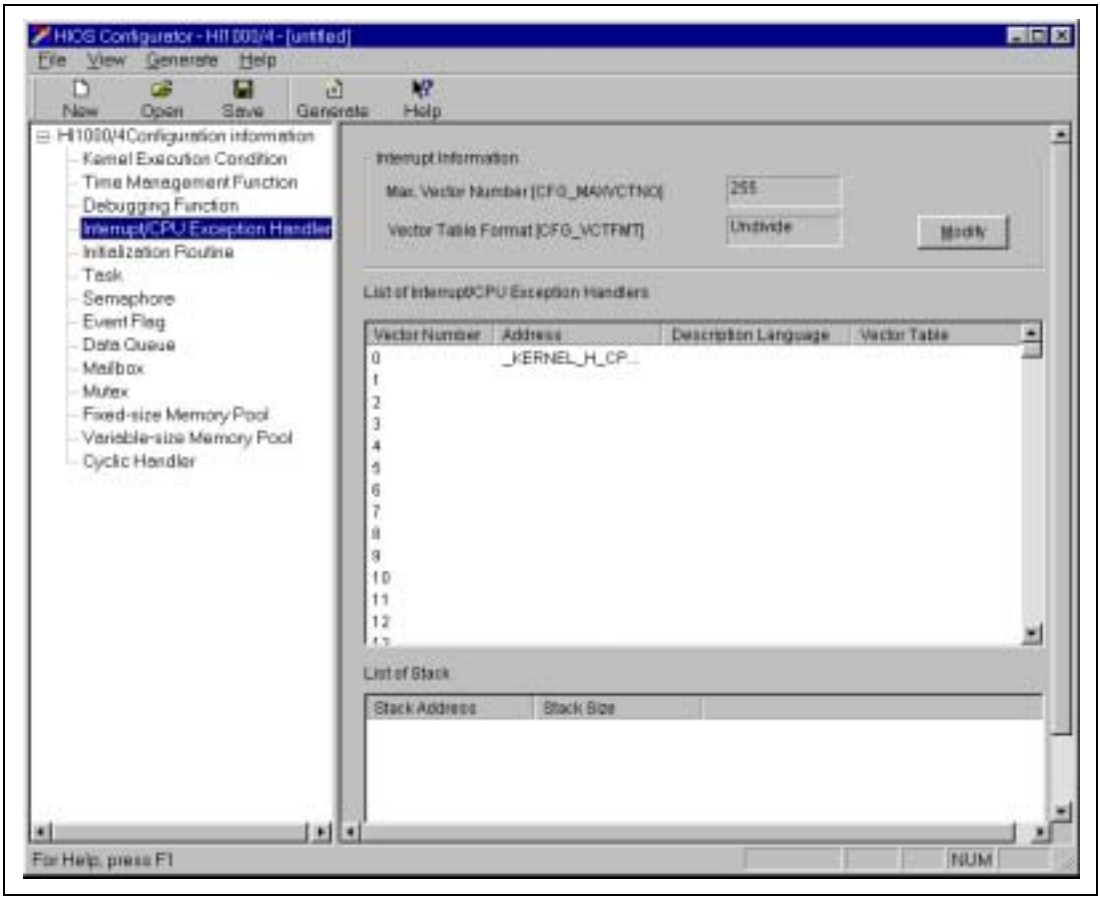


Figure 3.26 Interrupt/CPU Exception Handler View (HI1000/4)

The items to be set in the Interrupt/CPU Exception Handler View are shown in table 3.14.

Table 3.14 Setting Items in Interrupt/CPU Exception Handler View

No.	Setting Item	Contents	Target OS
1	Interrupt Information	Define information relevant to the interrupt handler.	HI7000/4, HI7700/4, HI7750/4, and HI1000/4
	Interrupt information: Maximum vector number, total size of interrupt handler stacks, whether direct interrupt handler is enabled or not, and whether interrupt handler dynamically created is embedded or not		HI7000/4
	Interrupt information: Maximum exception code and total size of interrupt handler stacks		HI7700/4 and HI7750/4
	Interrupt information: Maximum vector number and vector table format		HI1000/4
2	List of Interrupt/CPU/Trap Exception Handlers	Define the handler initiated by each vector source.	HI7000/4
	List of Interrupt/CPU Exception Handlers	Define the handler initiated by each exception source.	HI7700/4 and HI7750/4
	List of Interrupt/CPU Exception Handlers	Define the handler initiated by each vector source.	HI1000/4
3	List of Stack	Define information for the stack used by the interrupt handler.	HI1000/4

The procedure for registering a handler, such as the interrupt handler or CPU exception handler, is described below.

Handler Registering Procedure:

1. Select a vector number (or exception code) for registering a handler.
2. Select [Define] from the sub-menu displayed by right-clicking.
3. Set the necessary data in the displayed definition window and complete registration by pressing the [OK] button.

(g) Trap Exception Handler View

The Trap Exception Handler View is shown in figure 3.27.

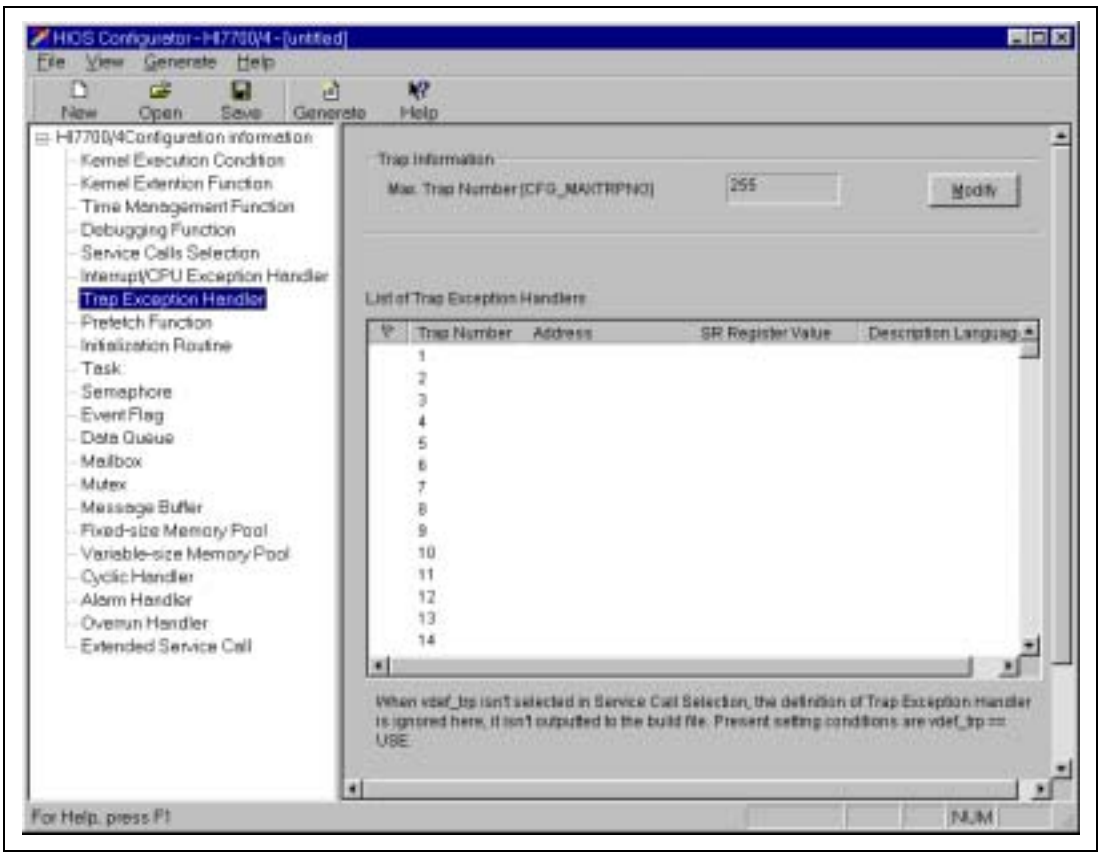


Figure 3.27 Trap Exception Handler View (HI7700/4 and HI7750/4)

The HI7000/4 and HI1000/4 configurators do not have the Trap Exception Handler View.

The items to be set in the Trap Exception Handler View are shown in table 3.15.

Table 3.15 Setting Items in Trap Exception Handler View

No.	Setting Item	Contents	Target OS
1	Trap Information	Define the maximum trap number.	HI7700/4 and HI7750/4
2	List of Trap Exception Handlers	Define the handler initiated by the trap exception source.	HI7700/4 and HI7750/4

The procedure for registering a trap exception handler is described below.

Trap Exception Handler Registering Procedure:

1. Select a trap number for registering a handler.
2. Select [Define] from the sub-menu displayed by right-clicking.
3. Set the necessary data in the displayed definition window and complete registration by pressing the [OK] button.

(h) Prefetch Function View

The Prefetch Function View is shown in figure 3.28.

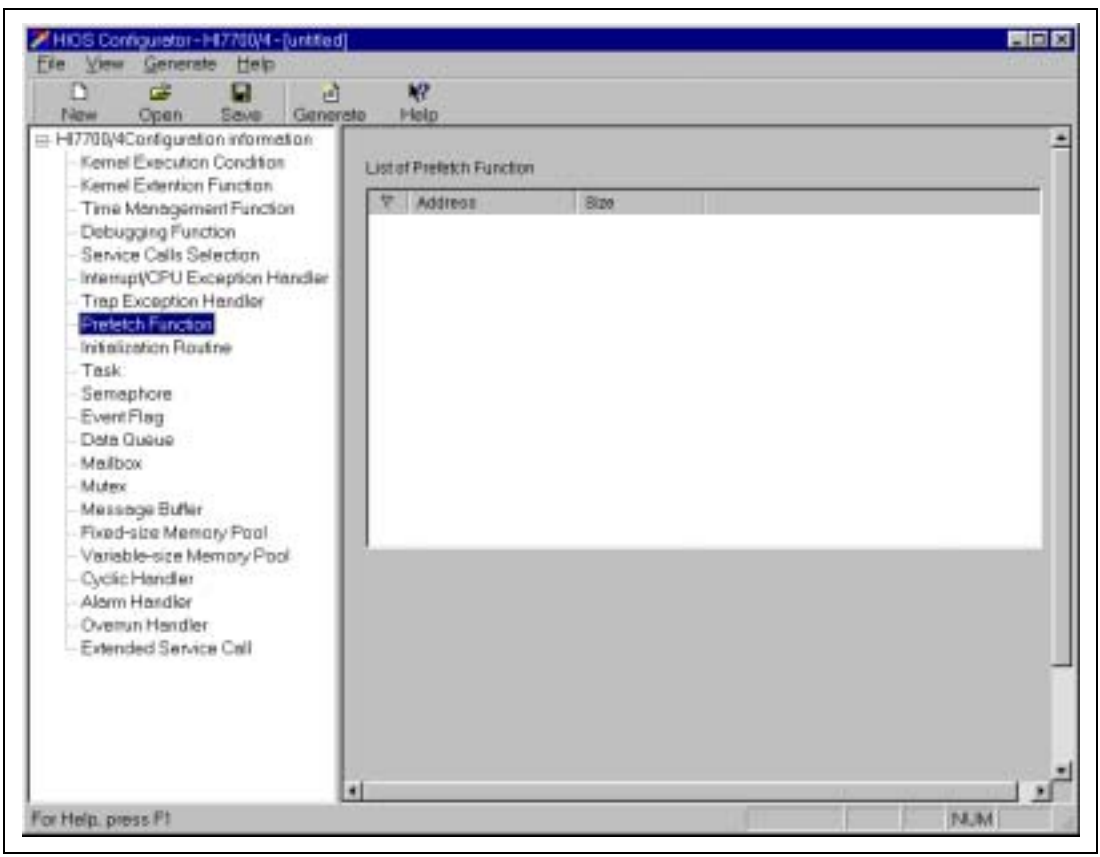


Figure 3.28 Prefetch Function View (HI7700/4 and HI7750/4)

The HI7000/4 and HI1000/4 configurators do not have the Prefetch Function View.

The item to be set in the Prefetch Function View is shown in table 3.16.

Table 3.16 Setting Item in Prefetch Function View

No.	Setting Item	Contents	Target OS
1	List of Prefetch Function	Define the start address of the area to be prefetched when the kernel is idle.	HI7700/4 and HI7750/4

The procedure for setting the prefetch functions is described below.

Prefetch Function Setting Procedure:

1. Select [Register] from the sub-menu displayed by right-clicking in [List of Prefetch Function].
2. Set the necessary data in the displayed registration window and complete registration by pressing the [Register] button.
3. Since registration can be performed continuously, after pressing the [OK] button, the next prefetch function can be registered.

   On completing all registrations, click the [Cancel] button to finish registration.

(i) Initialization Routine View

The Initialization Routine View is shown in figure 3.29.

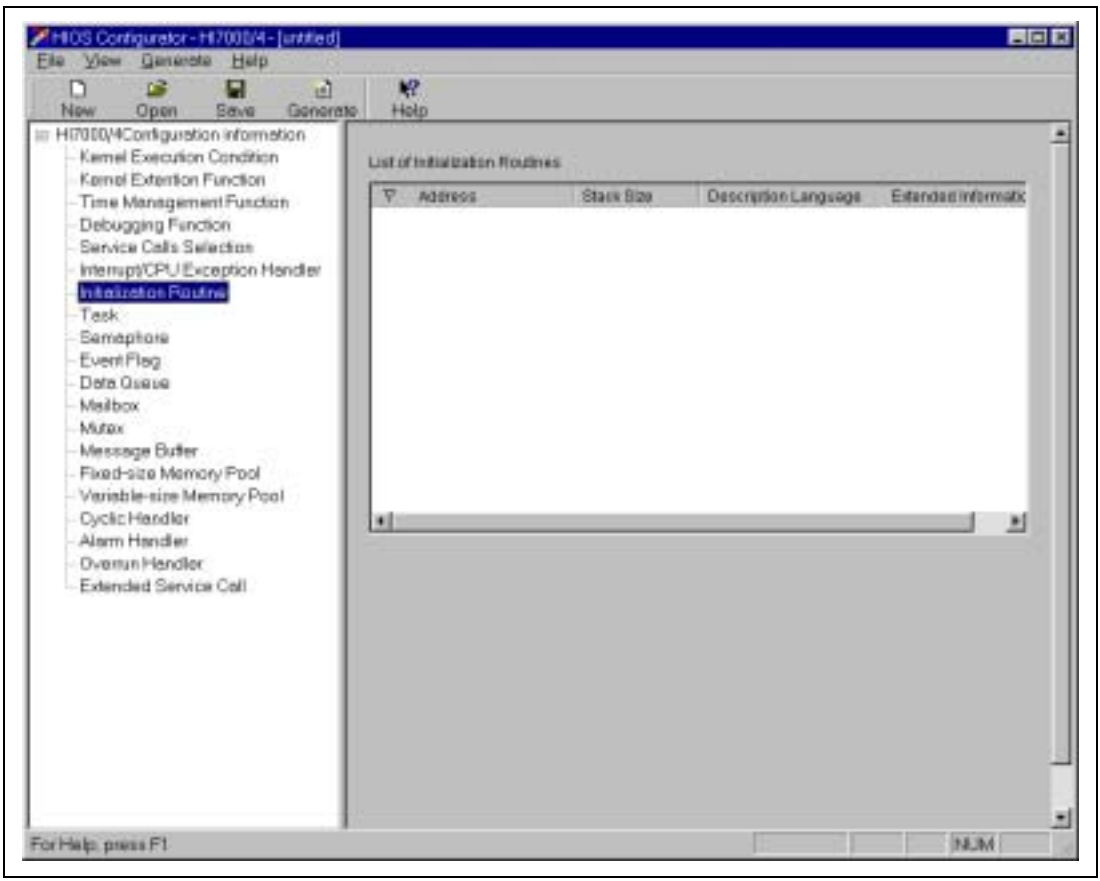


Figure 3.29 Initialization Routine View

The item to be set in the Initialization Routine View is shown in table 3.17.

Table 3.17 Setting Item in Initialization Routine View

No.	Setting Item	Contents	Target OS
1	List of Initialization Routines	Define the initialization routine called from the kernel initialization processing.	HI7000/4, HI7700/4, HI7750/4, and HI1000/4

The procedure for registering an initialization routine is described below.

Initialization Routine Registering Procedure:

1. Select [Register] from the sub-menu displayed by right-clicking in [List of Initialization Routines].
2. Set the necessary data in the displayed registration window and complete registration by pressing the [Register] button.
3. Since registration can be performed continuously, after pressing the [OK] button, the next initialization routine can be registered.

   On completing all registrations, click the [Cancel] button to finish registration.

(j) Task View

The Task View is shown in figures 3.30 and 3.31.

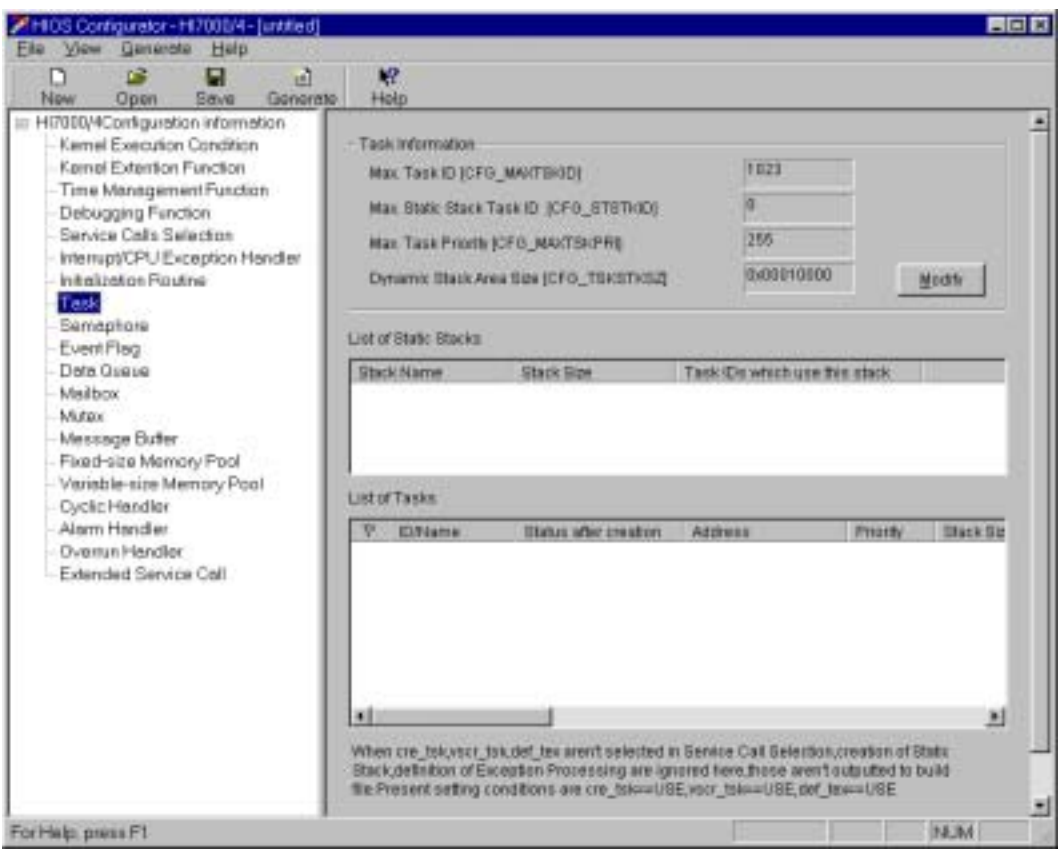


Figure 3.30 Task View (HI7000/4, HI7700/4, and HI7750/4)

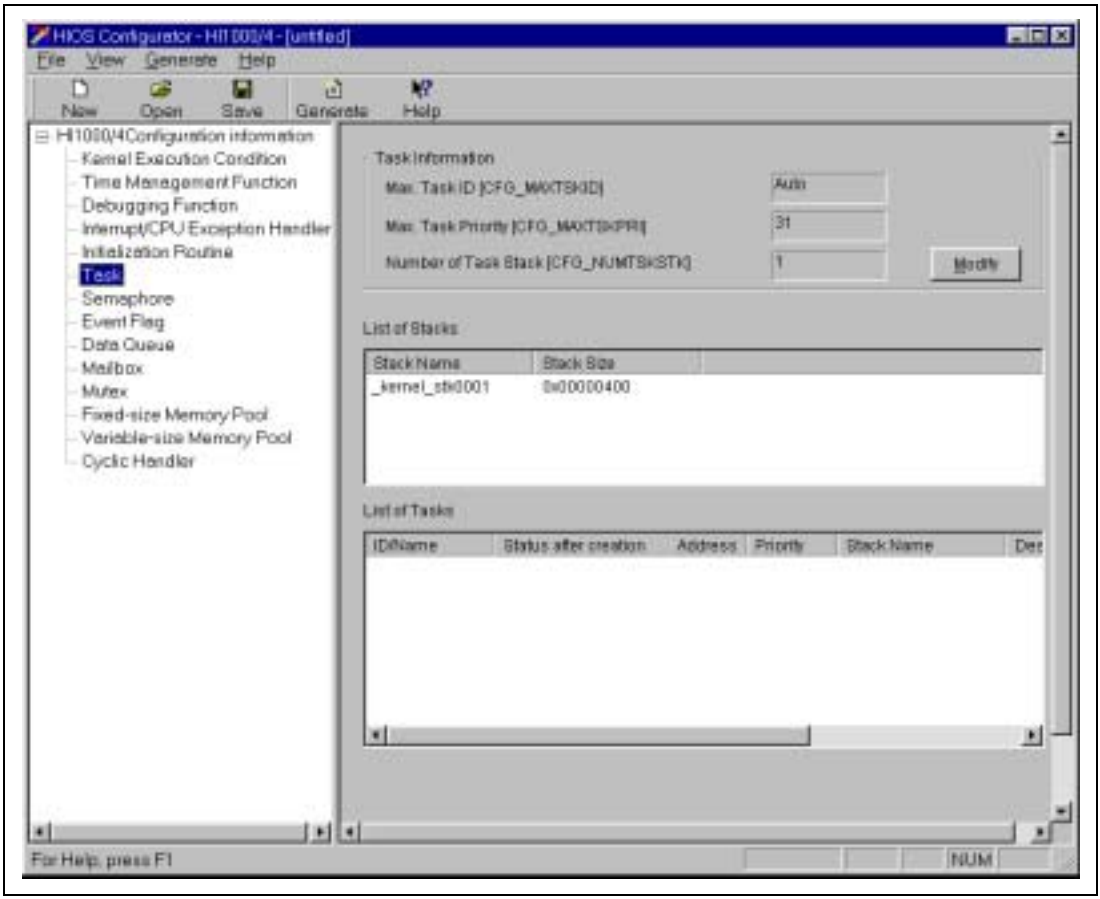


Figure 3.31 Task View (HI1000/4)

The items to be set in the Task View are shown in table 3.18.

Table 3.18 Setting Items in Task View

No.	Setting Item	Contents	Target OS
1	Max. Task ID	Define the maximum task ID to be registered in the kernel.	HI7000/4, HI7700/4, HI7750/4, and HI1000/4
2	Max. Static Stack Task ID	Define the maximum task ID using the static stack	HI7000/4, HI7700/4, and HI7750/4
3	Max. Task Priority	Define the maximum task priority to be registered in the kernel.	HI7000/4, HI7700/4, HI7750/4, and HI1000/4
4	Dynamic Stack Area Size	Define the total used size of the dynamic stack	HI7000/4, HI7700/4, and HI7750/4
5	Number of Task Stack	Number of stacks	HI1000/4
6	List of Static Stacks	Registered static stack information	HI7000/4, HI7700/4, and HI7750/4
7	List of Tasks	Registered task information	HI7000/4, HI7700/4, HI7750/4, and HI1000/4
8	List of Stacks	Registered task stack information	HI1000/4

The procedure for registering a task is described below.

Task Registering Procedure:

1. Select [Create] from the sub-menu displayed by right-clicking in [List of Tasks].
2. Set the necessary data in the displayed creation window and complete registration by pressing the [Create] button.
3. Since registration can be performed continuously, after pressing the [Create] button, the next task can be registered.

   On completing all registrations, click the [Cancel] button to finish registration.

(k) Views for objects other than a task

For the view of each object, such as the Semaphore View and Event Flag View, the structure and setting items are the same, except for those for the Task View. Therefore, the view of each object other than a task is described with the Semaphore View as an example. The Semaphore View is shown in figure 3.32.

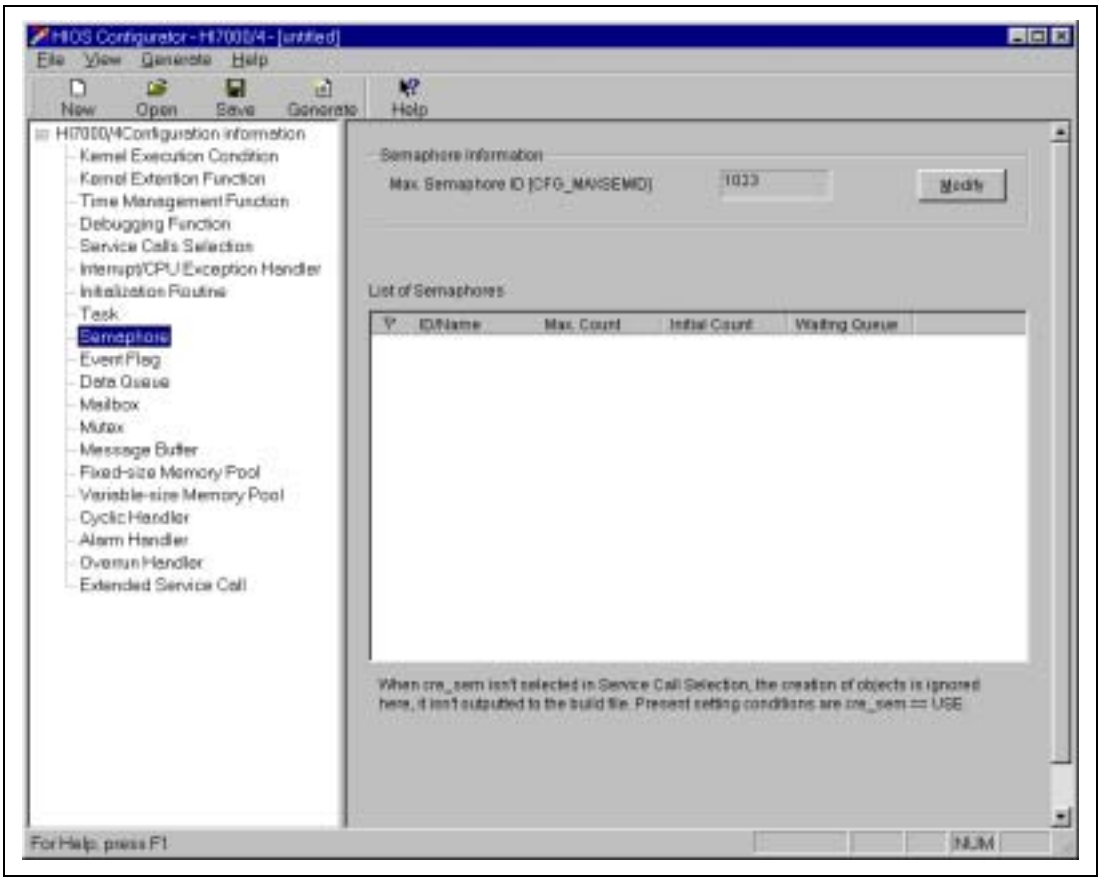


Figure 3.32 Semaphore View

The items to be set in the Semaphore View are shown in table 3.19.

Table 3.19 Setting Items in Semaphore View

No.	Setting Item	Contents	Target OS
1	Max. Semaphore ID	Define the maximum semaphore ID to be registered in the kernel.	HI7000/4, HI7700/4, HI7750/4, and HI1000/4
2	List of Semaphores	Registered semaphore information	HI7000/4, HI7700/4, HI7750/4, and HI1000/4

Refer to views of the objects, except for the Task View, with the above setting items replaced with each object name (e.g. event flag or mailbox).

The procedure for registering an object is described below with a semaphore as an example.

Semaphore Registering Procedure:

1. Select [Create] from the sub-menu displayed by right-clicking in [List of Semaphores].
2. Set the necessary data in the displayed creation window and complete registration by pressing the [Create] button.
3. Since registration can be performed continuously, after pressing the [Create] button, the next semaphore can be registered.

   On completing all registrations, click the [Cancel] button to finish registration.

3.2.2 FAQ about Configurator

This section answers a question about the configurator which is frequently asked by users of the HI series OS.

FAQ Contents:

(1) Multiple Interrupt Setting by the Configurator

Classification: Configuration, kernel environment definition, and configurator					
Question	HI7000/4	HI7700/4	HI7750/4		HI1000/4
When multiple interrupts are enabled, what items should be specified by the configurator? What descriptions are necessary for the defined interrupt handlers to implement multiple interrupts?					
Answer In the Interrupt/CPU Exception Handler View of the configurator, specify the contents of the exception code of each interrupt to be used. For each exception code, specify an address and a value to be set to SR. As this SR setting is used as the SR value when the corresponding interrupt handler is initiated, specify appropriate values according to the interrupt level. Multiple interrupts are implemented by simply specifying these SR values.					

3.2.3 Definition by Setup Table (HI2000/3)

In the HI2000/3, the kernel environment is defined with the setup table.

The setup table consists of the definition fields listed in table 3.20.

Table 3.20 Setup Table Structure

Definition Field Name	Defined Contents
Constant definition field	Defines information required for the kernel functions (synchronization and communication function, time management function, etc.).
Task registration field	Defines information required for task execution.
Fixed-length memory pool registration field	Defines information required for fixed-length memory pools.
Variable-length memory pool registration field	Defines information required for variable-length memory pools.
Cyclic handler registration field	Defines information required for cyclic handlers.
System call trace function registration field	Defines information required for system call trace functions.
Extended information registration field	Defines information required for extended information for tasks, event flags, semaphores, mailboxes, fixed-length and variable-length memory pools, and cyclic handlers.

All of the above setting items must be set regardless of whether the item is registered or not or used or not. If not, an undefined error will occur at system linkage.

(1) Constant definition field

This field defines information required for the kernel functions (such as synchronization-and-communication and time-management functions). The constant definition field of the setup table is shown in figure 3.33.

```
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;%%%    VALUE define section                                    %%%
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;------ Usage -------------------------------------------------------------------------------------------------
;LABEL                  VALUE   ;:[ RANGE ]     ;: COMMENT
;------------------------------------------------------------------------------------------------------------------
CPUINTM:      .assign 3         ;:[0.......3]        ;: CPU interrupt mode                  ← (1)
IMASK:        .assign 6         ;:[0.......8]        ;: Max interrupt level                 ← (2)
MAXPRI:       .assign 31        ;:[0......31]        ;: Max low priority                    ← (3)
FLGCNT:       .assign 4         ;:[0.....255]        ;: Eventflag definition count          ← (4)
SEMCNT:       .assign 4         ;:[0.....255]        ;: Semaphore definition count          ← (5)
MBXCNT:       .assign 4         ;:[0.....255]        ;: Mailbox definition count            ← (6)
;
OSSTKSIZ:     .equu   18+(10*2)+(6*1)+8  ;:[18...]      ;: OS stack size                    ← (7)
TIMSTKSIZ:    .equ    40+(10*1)+(6*1)+8  ;:[0, 40...]  ;: Timer stack size                 ← (8)
TRCSTKSIZ:    .equ    26+(6*1)+8         ;:[0, 26...]  ;: Trace stack size                 ← (9)
;
TTMOUT:       .assign USE     ;:[USE / NOTUSE]       ;: Time-out Function define            ←(10)
;
```

Figure 3.33 Constant Definition Field of Setup Table

(1) CPUINTM (Interrupt control mode)

Specifies the interrupt control mode used.

(2) IMASK (Kernel interrupt mask level)

Specifies the mask level for masking interrupts inside the kernel.

(3) MASKPRI (Maximum task priority)

Specifies the lowest task priority.

(4) FLGCNT (Number of event flags registered)

Specifies the maximum event flag ID to be registered in the kernel.

(5) SEMCNT (Number of semaphores registered)

Specifies the maximum semaphore ID to be registered in the kernel.

(6) MBXCNT (Number of mailboxes registered)

Specifies the maximum mailbox ID to be registered in the kernel.

(7) OSSTKSIZ (Kernel stack size)

Specifies the stack size used by the kernel (OS).

(8) TIMSTKSIZ (Timer interrupt handler stack size)

Specifies the stack size used by the timer interrupt handler.

(9) TRCSTKSIZ (System call trace function stack size)

Specifies the stack size used for processing when the system call trace function is used.

(10) TTMOUT (Timeout function enabled/disabled)

Specifies whether a system call with timeout can be used.

Note: Do not modify or delete symbols used in the constant definition field.

For the calculation methods of OSSTKSIZ, TIMSTKSIZ, and TRCSTKSIZ, refer to the HI2000/3 User's Manual.

(2) Task registration field

This field defines various information for registering tasks. The task registration field of the setup table is shown in figure 3.34.

```
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;%%%    TASK define section                                          %%%
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;------ Usage ---------------------------------------------------------------------------
;                TASK_TOP_LABEL                       ;: COMMENT
;----------------------------------------------------------------------------------------
        .import _TASKA                                ;: TASK.C
        .import _TASKB                                ;: TASK.C          (1)
;
;------ Usage ---------------------------------------------------------------------------
;                    .res.b  SIZE + TSKSTKSIZ ;[RANGE] ;: COMMENT
;TSK?_SP_LABEL:  .equ    $                             ;: COMMENT
;----------------------------------------------------------------------------------------
TSKSTKSIZ:           .equ    50+(10*2)+(6*1)+6+8; [50...]    ;: Task minimum stack size
        .section             h2sstack, stack, align = 2
                     .res.b  (36) +TSKSTKSIZ ;[50...]        ;: tskid1 stack area
TSK1_SP:             .equ    $                                                          (3)
                     .res.b  8
                     .res.b  (36) +TSKSTKSIZ ;[50...]        ;: tskid2 stack area
TSK2_SP:             .equ    $
                     .res.b  8
                     .res.b  (32) +TSKSTKSIZ ;[50...]        ;: tskid3 stack area      (2)
TSK3_SP:             .equ    $
                     .res.b  8
                     .res.b  (32) +TSKSTKSIZ ;[50...]        ;: tskid4 stack area
TSK4_SP:             .equ    $
                     .res.b  8
;
        .section             h2ssetup, code, align = 2
_HI_H8S:             .res.b  10                              ;: System Area
;------ Usage ---------------------------------------------------------------------------
;LABEL               .data.b IMOD, ITSKPRI                   ;: COMMENT
;                    .data.l  ITSKADR, ITSKSP                ;: COMMENT
;----------------------------------------------------------------------------------------
NOEXS:               .assign 0                               ;: initial mode = NO EXIST
RDY:                 .assign 1                               ;: initial mode = READY
DMT:                 .assign (-1)                            ;: initial mode = DORMANT
TDTLEN:              .assign 10;<- Not Change !              ;: TDT Length
        .section             h2ssetup, code, align = 2
_HI_TDT:             .equ     $-TDTLEN                       ;: Task define table
TDT_TOP:             .equ     $                              ;:
tdt_id1:             .data.b DMT, 1                          ;: init. mode, init. priority      (5)
                     .data.l  _TASKA, TSK1_SP                ;: top address, stack pointer
tdt_id2:             .data.b DMT, 2                          ;: init. mode, init. priority
                     .data.l  _TASKB, TSK2_SP                ;: top address, stack pointer
tdt_id3:             .data.b  NOEXS, 3                       ;: init. mode, init. priority     (4)
                     .data.l  0, TSK3_SP                     ;: top address, stack pointer
tdt_id4:             .data.b NOEXS, 4                        ;: init. mode, init. priority
                     .data.l  0, TSK4_SP                     ;: top address, stack pointer
tdt_id5:             .data.b  NOEXS, 5                       ;: init. mode, init. priority
                     .data.l  0, TSK4_SP                     ;: top address, stack pointer
TDT_BTM:
TSKCNT:              .equ     (TDT_BTM-TDT_TOP) / TDTLEN
                                        ;:[0...255]          ;: Task definition count
;
```

Figure 3.34 Task Registration Field of Setup Table

(1) Declares the start address of the task to be used as an external reference symbol.

(2) Task stack definition field

Allocates the stack area used by each task.

(3) Definition of task stack area

Defines the stack area for each task.

(4) Task definition field

Defines the tasks to be registered in the kernel.

(5) Definition of task

Defines information for each task to be registered in the kernel.

Note: Do not modify or delete symbols TDTLEN, _HI_TDT, TDT_TOP, TDT_BTM, and TSKCNT, which are used in the task registration field.
Do not modify or delete the line where TSKCNT is defined.

The details of defining a task stack area are as follows:

Line 1:	Defines the stack size used.
Line 2:	Defines the stack label (task stack bottom).
Line 3:	Defines the shared-stack-management area. (If the shared stack function is not used, this area need not be defined.)

The details of defining a task are as follows:

[Format]

```
LABEL:  .data.b  IMOD, ITSKPRI
        .data.l  ITSKADR, ITSKSP
```

— LABEL: Can be freely defined (can be omitted).

— IMOD (task initial state): Defines each task's initial state at task registration and system initiation as follows:
(1) NOEXS (= 0): Unregistered
(2) RDY (= 1): READY state when initiated
(3) DMT (= –1): DORMANT state when initiated

— ITSKPRI (task initial priority): Defines each task's initial priority.

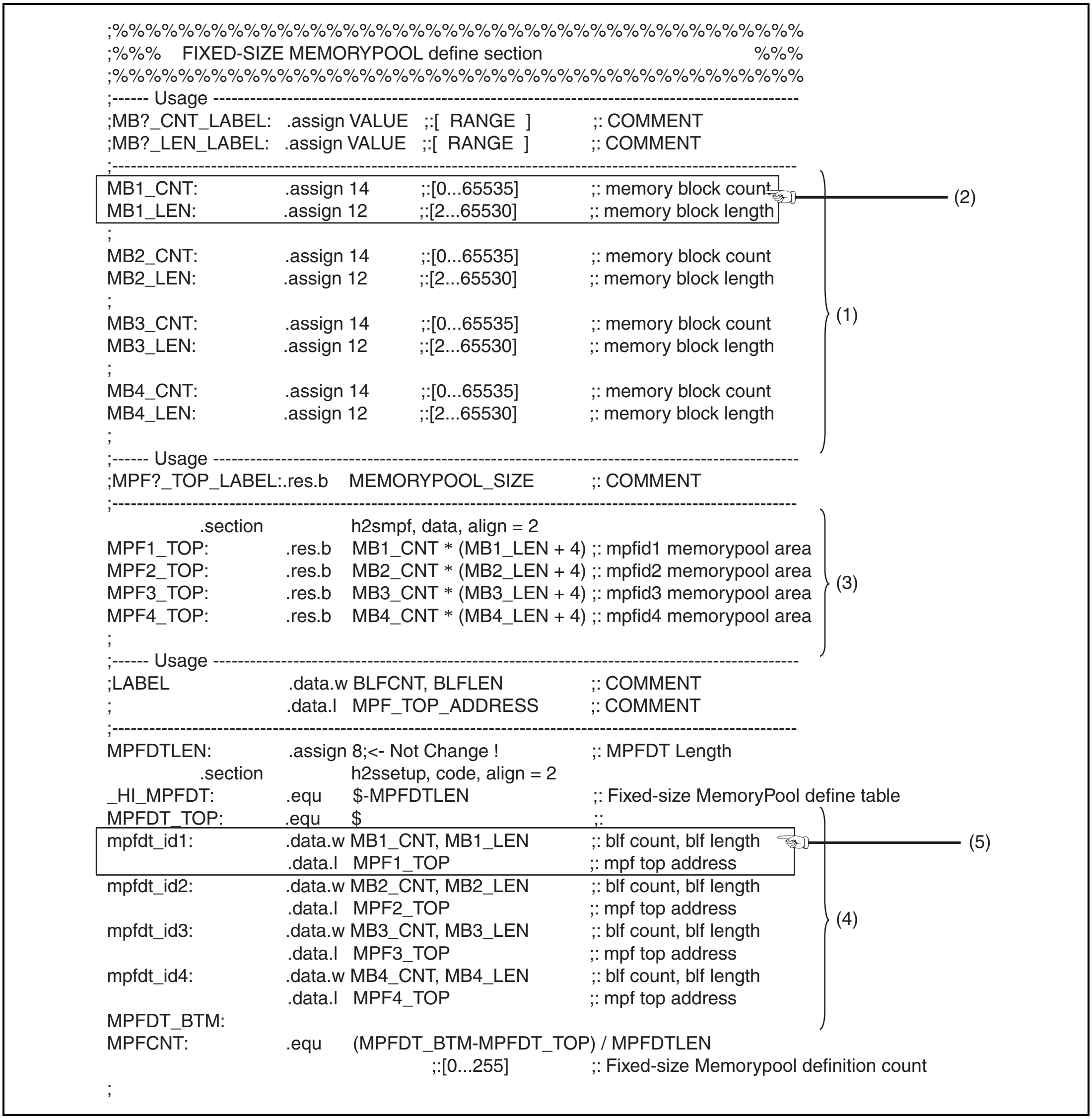
— ITSKADR (task start address): Defines the start address of the task. (Defines the start address to be defined as an external reference symbol.)

— ITSKSP (task stack pointer): Defines the stack pointer to be used at task initiation (stack label defined in the task stack area definition field).

When adding a task to be registered, insert the definition data before TDT_BTM.

(3) Fixed-length memory pool registration field

This field defines various information for registering fixed-length memory pools. The fixed-length memory pool registration field of the setup table is shown in figure 3.35.



```
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;%%%    FIXED-SIZE MEMORYPOOL define section                    %%%
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;------ Usage ------------------------------------------------------------------------------------------
;MB?_CNT_LABEL:   .assign VALUE   ;:[  RANGE  ]           ;: COMMENT
;MB?_LEN_LABEL:   .assign VALUE   ;:[  RANGE  ]           ;: COMMENT
;-------------------------------------------------------------------------------------------------------
MB1_CNT:               .assign 14        ;:[0...65535]          ;: memory block count
MB1_LEN:               .assign 12        ;:[2...65530]          ;: memory block length
;
MB2_CNT:               .assign 14        ;:[0...65535]          ;: memory block count
MB2_LEN:               .assign 12        ;:[2...65530]          ;: memory block length
;
MB3_CNT:               .assign 14        ;:[0...65535]          ;: memory block count
MB3_LEN:               .assign 12        ;:[2...65530]          ;: memory block length
;
MB4_CNT:               .assign 14        ;:[0...65535]          ;: memory block count
MB4_LEN:               .assign 12        ;:[2...65530]          ;: memory block length
;
;------ Usage ------------------------------------------------------------------------------------------
;MPF?_TOP_LABEL:.res.b    MEMORYPOOL_SIZE          ;: COMMENT
;-------------------------------------------------------------------------------------------------------
              .section           h2smpf, data, align = 2
MPF1_TOP:              .res.b    MB1_CNT * (MB1_LEN + 4) ;: mpfid1 memorypool area
MPF2_TOP:              .res.b    MB2_CNT * (MB2_LEN + 4) ;: mpfid2 memorypool area
MPF3_TOP:              .res.b    MB3_CNT * (MB3_LEN + 4) ;: mpfid3 memorypool area
MPF4_TOP:              .res.b    MB4_CNT * (MB4_LEN + 4) ;: mpfid4 memorypool area
;
;------ Usage ------------------------------------------------------------------------------------------
;LABEL                 .data.w BLFCNT, BLFLEN             ;: COMMENT
;                      .data.l   MPF_TOP_ADDRESS          ;: COMMENT
;-------------------------------------------------------------------------------------------------------
MPFDTLEN:              .assign 8;<- Not Change !          ;: MPFDT Length
              .section           h2ssetup, code, align = 2
_HI_MPFDT:             .equu     $-MPFDTLEN               ;: Fixed-size MemoryPool define table
MPFDT_TOP:             .equ      $                        ;:
mpfdt_id1:             .data.w MB1_CNT, MB1_LEN           ;: blf count, blf length
                       .data.l   MPF1_TOP                 ;: mpf top address
mpfdt_id2:             .data.w MB2_CNT, MB2_LEN           ;: blf count, blf length
                       .data.l   MPF2_TOP                 ;: mpf top address
mpfdt_id3:             .data.w MB3_CNT, MB3_LEN           ;: blf count, blf length
                       .data.l   MPF3_TOP                 ;: mpf top address
mpfdt_id4:             .data.w MB4_CNT, MB4_LEN           ;: blf count, blf length
                       .data.l   MPF4_TOP                 ;: mpf top address
MPFDT_BTM:
MPFCNT:                .equ      (MPFDT_BTM-MPFDT_TOP) / MPFDTLEN
                                          ;:[0...255]      ;: Fixed-size Memorypool definition count
;
```

Figure 3.35 Fixed-Length Memory Pool Registration Field of Setup Table

(1) Definition field for memory block size and number of memory blocks

Defines the memory block size and number of memory blocks which are used by the fixed-length memory pools to be registered in the kernel. (The symbols used here are used in the subsequent area allocation and definition table information.)

(2) Definition of memory block size and number of memory blocks

Defines the memory block size and number of memory blocks which are used by the fixed-length memory pools.

(3) Allocation of fixed-length memory pool areas

Allocates each fixed-length memory pool area based on the memory block size and number of memory blocks.

(4) Fixed-length memory pool definition field

Defines the fixed-length memory pools to be registered in the kernel.

(5) Definition of fixed-length memory pool

Defines information for each fixed-length memory pool to be registered in the kernel.

Note: Do not modify or delete symbols MB?_CNT, MB?_SIZ, MPF?_TOP, MPFDTLEN, MPFDT_TOP, and MPFDT_BTM, which are used in the fixed-length memory pool registration field.
Do not modify or delete the line where MPFCNT is defined.

The details of defining a fixed-length memory pool are as follows:

[Format]

```
LABEL:  .data.w  BLFCNT, BLFLEN
        .data.l  MPF_TOP_ADDRESS
```

— LABEL: Can be freely defined (can be omitted).
— BLFCNT (number of blocks): Defines the number of memory blocks in the fixed-length memory pool.
— BLFLEN (block size): Defines the memory block size of the fixed-length memory pool.
— MPF_TOP_ADDRESS (fixed-length memory pool address): Defines the start address of the fixed-length memory pool.

When adding a fixed-length memory pool to be registered, insert the definition data before MPFDT_BTM.

(4) Variable-length memory pool registration field

This field defines various information for registering variable-length memory pools. The variable-length memory pool registration field of the setup table is shown in figure 3.36.

```
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;%%%    VARIABLE-SIZE MEMORYPOOL define section                %%%
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;------ Usage -----------------------------------------------------------------------------------------
;MPL?_SIZ_LABEL:  .assign VALUE    ;:[  RANGE  ]               ;: COMMENT
;-----------------------------------------------------------------------------------------------------
MPL1_SIZ:            .assign 380       ;:[18.....]                     ;: memorypool size
MPL2_SIZ:            .assign 380       ;:[18.....]                     ;: memorypool size
MPL3_SIZ:            .assign 380       ;:[18.....]                     ;: memorypool size     (1)
MPL4_SIZ:            .assign 380       ;:[18.....]                     ;: memorypool size
;
;------ Usage -----------------------------------------------------------------------------------------
;MPL?_TOP_LABEL:.res.b   VARIABLE_MEMORYPOOL_SIZE ;: COMMENT
;-----------------------------------------------------------------------------------------------------
        .section                    h2smpl, data, align = 2
MPL1_TOP:            .res.b  MPL1_SIZ                    ;: mplid1 memorypool area
MPL2_TOP:            .res.b  MPL2_SIZ                    ;: mplid2 memorypool area
MPL3_TOP:            .res.b  MPL3_SIZ                    ;: mplid3 memorypool area     (2)
MPL4_TOP:            .res.b  MPL4_SIZ                    ;: mplid4 memorypool area
;
;------ Usage -----------------------------------------------------------------------------------------
;LABEL               .data.l  BLKSIZ                          ;: COMMENT
;                    .data.l  VARIABLE_MEMORYPOOL_TOP  ;: COMMENT
;-----------------------------------------------------------------------------------------------------
MPLDTLEN:            .assign 8;<- Not Change !                ;: MPLDT Length
        .section                    h2ssetup, code, align = 2
_HI_MPLDT:           .equ     $-MPLDTLEN                      ;: Variable-size MemoryPool define table
MPLDT_TOP:           .equ     $                               ;:
mpldt_id1:           .data.l  MPL1_SIZ                        ;: mpl size ☜            (4)
                     .data.l  MPL1_TOP                        ;: mpl top address
mpldt_id2:           .data.l  MPL2_SIZ                        ;: mpl size
                     .data.l  MPL2_TOP                        ;: mpl top address
mpldt_id3:           .data.l  MPL3_SIZ                        ;: mpl size             (3)
                     .data.l  MPL3_TOP                        ;: mpl top address
mpldt_id4:           .data.l  MPL4_SIZ                        ;: mpl size
                     .data.l  MPL4_TOP                        ;: mpl top address
MPLDT_BTM:
MPLCNT:              .equ    (MPLDT_BTM-MPLDT_TOP) / MPLDTLEN
                                           ;:[0...255]           ;: Variable-size Memorypool definition count
;
```

Figure 3.36 Variable-Length Memory Pool Registration Field of Setup Table

(1) Memory pool size definition field

Defines the memory pool sizes that are used by the variable-length memory pools to be registered in the kernel. (The symbols used here are used in the subsequent area allocation and definition table information.)

(2) Allocation of variable-length memory pool areas

Allocates each variable-length memory pool area based on the memory pool size.

(3) Variable-length memory pool definition field

Defines the variable-length memory pools to be registered in the kernel.

(4) Definition of variable-length memory pool

Defines information for each variable-length memory pool to be registered in the kernel.

Note: Do not modify or delete symbols MPL?_SIZ, MPL?_TOP, MPLDTLEN, MPLDT_TOP, and MPLDT_BTM, which are used in the variable-length memory pool registration field.
Do not modify or delete the line where MPLCNT is defined.

The details of defining a variable-length memory pool are as follows:

[Format]

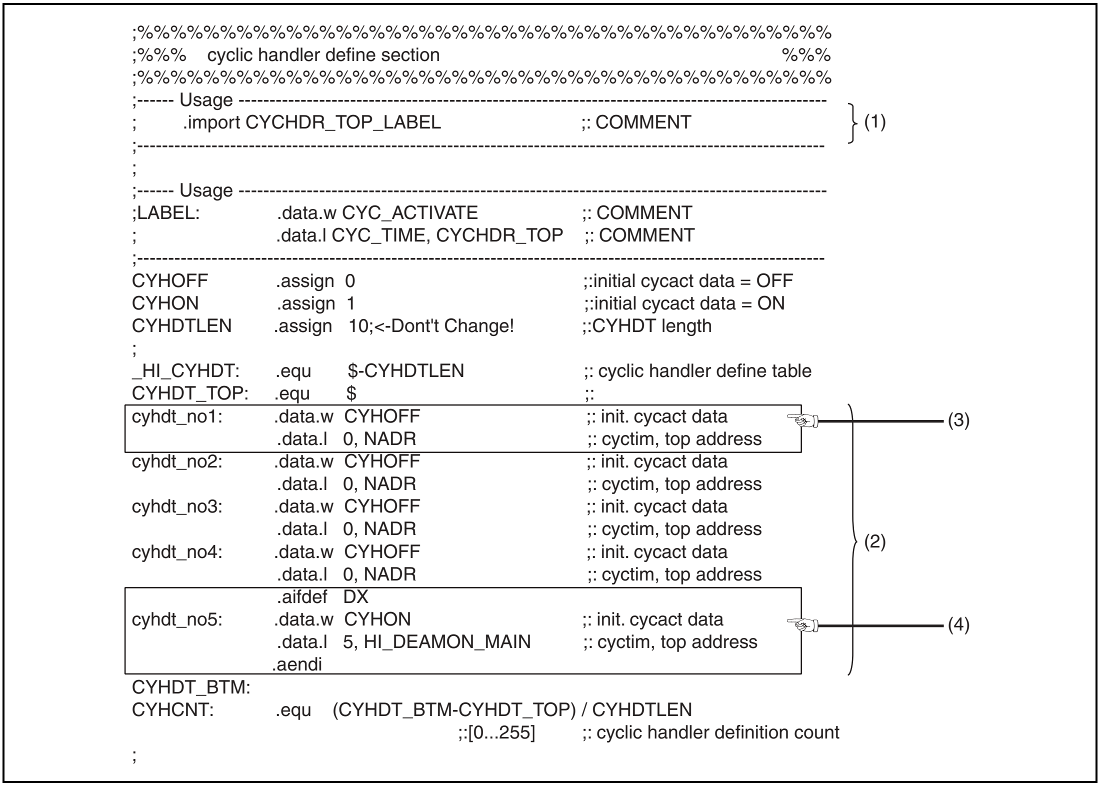
```
LABEL:  .data.l  MPL?_SIZ
        .data.l  MPL?_TOP
```

— LABEL: Can be freely defined (can be omitted).
— BLKSIZ (block size): Defines the size of the variable-length memory pool.
— VARIABLE_MEMORYPOOL_TOP (variable-length memory pool address): Defines the start address of the variable-length memory pool.

When adding a variable-length memory pool to be registered, insert the definition data before MPLDT_BTM.

(5) Cyclic handler registration field

This field defines various information for registering cyclic handlers. The cyclic handler registration field of the setup table is shown in figure 3.37.



```
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;%%%    cyclic handler define section                                      %%%
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;------ Usage ------------------------------------------------------------------------------------------
;        .import CYCHDR_TOP_LABEL                        ;: COMMENT
;-----------------------------------------------------------------------------------------------------------
;
;------ Usage ------------------------------------------------------------------------------------------
;LABEL:              .data.w CYC_ACTIVATE                ;: COMMENT
;                    .data.l CYC_TIME, CYCHDR_TOP        ;: COMMENT
;-----------------------------------------------------------------------------------------------------------
CYHOFF               .assign  0                          ;:initial cycact data = OFF
CYHON                .assign  1                          ;:initial cycact data = ON
CYHDTLEN             .assign   10;<-Dont't Change!       ;:CYHDT length
;
_HI_CYHDT:           .equ      $-CYHDTLEN                ;: cyclic handler define table
CYHDT_TOP:           .equ      $                         ;:
cyhdt_no1:           .data.w  CYHOFF                     ;: init. cycact data
                     .data.l   0, NADR                   ;: cyctim, top address
cyhdt_no2:           .data.w  CYHOFF                     ;: init. cycact data
                     .data.l   0, NADR                   ;: cyctim, top address
cyhdt_no3:           .data.w  CYHOFF                     ;: init. cycact data
                     .data.l   0, NADR                   ;: cyctim, top address
cyhdt_no4:           .data.w  CYHOFF                     ;: init. cycact data
                     .data.l   0, NADR                   ;: cyctim, top address
                     .aifdef   DX
cyhdt_no5:           .data.w  CYHON                      ;: init. cycact data
                     .data.l   5, HI_DEAMON_MAIN         ;: cyctim, top address
                    .aendi
CYHDT_BTM:
CYHCNT:              .equ    (CYHDT_BTM-CYHDT_TOP) / CYHDTLEN
                                          ;:[0...255]    ;: cyclic handler definition count
;
```

Figure 3.37 Cyclic Handler Registration Field of Setup Table

(1) Declares the start address of the cyclic handler to be used as an external reference symbol.

(2) Cyclic handler definition field

Defines the cyclic handlers to be registered in the kernel.

(3) Definition of cyclic handler

Defines information for each cyclic handler to be registered in the kernel.

(4) When the debugging extension is used, the debug daemon handler is registered as a cyclic handler.

Note: Do not modify or delete symbols _HI_CYHDT, CYHDTLEN, CYHDT_TOP, and CYHDT_BTM, which are used in the cyclic handler registration field.
Do not modify or delete the line where CYHCNT is defined.

The details of defining a cyclic handler are as follows:

[Format]
```
LABEL:  .data.w  CYC_ACTIVATE
        .data.l  CYC_TIME, CYCHDR_TOP
```

- LABEL: Can be freely defined (can be omitted).
- CYC_ACTIVATE (cyclic handler activation state): Defines the cyclic handler activation state as follows:
  (1) CYCOFF (= 0): Not initiated (not activated)
  (2) CYCON (= 1): Initiated (activated)
- CYC_TIME (cyclic time interval): Defines the cycle time to initiate the cyclic handler.
- CYCHDR_TOP (cyclic handler address): Defines the start address of the cyclic handler.

When adding a cyclic handler to be registered, insert the definition data before CYHDT_BTM.

(6) System call trace function registration field

This field defines various information for registering system call trace functions. The system call trace function registration field of the setup table is shown in figure 3.38.

```
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;%%%    SVC trace define section                                    %%%
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;------ Usage ---------------------------------------------------------------------------------------
;TRC_CNT:.assign   TRACE COUNT                         ;: COMMENT
;TRC_BUF:.assign   TRACE BUFFER ADDRESS                ;: COMMENT
;---------------------------------------------------------------------------------------------------
                .section     h2strc, data, align = 2
TRC_CNT:        .assign   4                            ;: trace count              ← (1)
TRC_BUF:        .res.b    16 + (TRC_CNT*28)            ;: trace buffer address     ← (2)
;
;------ Usage ---------------------------------------------------------------------------------------
;INITRC          .data.l   TRACE BUFFER ADDRESS        ;: COMMENT
;                .data.w   TRACE COUNT                 ;: COMMENT
;---------------------------------------------------------------------------------------------------
                .section     h2ssetup, code, align = 2
INITRC:          .equ      $                           ;:                         }
                 .data.l   TRC_BUF                     ;: trace buffer address    } ← (3)
                 .data.w   TRC_CNT                     ;: trace count             }
;
```

Figure 3.38 System Call Trace Function Registration Field of Setup Table

(1) TRC_CNT (maximum amount of trace information)

Defines the maximum amount of trace information that can be acquired by the system call trace function.

(2) TRC_BUF (allocation of trace buffer area)

Allocates the area for storing trace information that can be acquired by the system call trace function.

(3) Definition of system call trace function

Defines information for the system call trace function.

Note: Do not modify or delete symbols used in the system call trace function registration field.

The details of defining the system call trace function are as follows:

[Format]

```
INITRC:  .data.l  TRACE BUFFER ADDRESS
         .data.l  TRACE_COUNT
```

— INITRC: Symbol for defining system call trace function information

— TRACE BUFFER ADDRESS (trace buffer address for system call trace function): Defines the start address of the trace information acquisition area used by the system call trace function.

— TRACE_COUNT (amount of trace information for system call trace function): Defines the amount of trace information acquired by the system call trace function.

(7) Extended information registration fields

These fields define various information for registering extended information. The extended information registration fields of the setup table are shown in figures 3.39 to 3.45.

```
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;%%%    Task Extended Information define section                              %%%
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;------ Usage -----------------------------------------------------------------------------------------
;LABEL                .data.l  TSK?_EXINF          ;: COMMENT
;-----------------------------------------------------------------------------------------------------
                 .section        h2ssetup, code, align = 2
_HI_TSKEXINF:    .equ     $-EXLEN                  ;: TSK exinf define area
TSKE_TOP:         .equ    $                         ;:
tsk1_exinf:          .data.l  00000000              ;: tskid = 1 exinf
tsk2_exinf:          .data.l  00000000              ;: tskid = 2 exinf
tsk3_exinf:          .data.l  00000000              ;: tskid = 3 exinf           (1)
tsk4_exinf:          .data.l  00000000              ;: tskid = 4 exinf
tsk5_exinf:          .data.l  00000000              ;: tskid = 5 exinf
TSKE_BTM:
TSKECNT:            .equ    (TSKE_BTM-TSKE_TOP) / EXLEN
                                       ;:[0...255]     ;: tsk exinf count
;
```

Figure 3.39 Task Extended Information Registration Field of Setup Table

```
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;%%%    Event Flag Extended Information define section                          %%%
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;------ Usage ---------------------------------------------------------------------------------------------
;LABEL               .data.l  FLG?_EXINF          ;: COMMENT
;---------------------------------------------------------------------------------------------------------
                  .section       h2ssetup, code, align = 2
_HI_FLGEXINF:   .equ     $-EXLEN                  ;: FLG exinf define area
FLGE_TOP:         .equ    $                         ;:
flg1_exinf:          .data.l  00000000              ;: flgid = 1 exinf
flg2_exinf:          .data.l  00000000              ;: flgid = 2 exinf
flg3_exinf:          .data.l  00000000              ;: flgid = 3 exinf           (1)
flg4_exinf:          .data.l  00000000              ;: flgid = 4 exinf
FLGE_BTM:
FLGECNT:            .equ    (FLGE_BTM-FLGE_TOP) / EXLEN
                                       ;:[0...255]     ;: flg exinf count
;
```

Figure 3.40 Event Flag Extended Information Registration Field of Setup Table

```
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;%%%   Semaphore Extended Information define section                    %%%
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;------ Usage ------------------------------------------------------------------
;LABEL              .data.l  SEM?_EXINF       ;: COMMENT
;-------------------------------------------------------------------------------
                .section     h2ssetup, code, align = 2
_HI_SEMEXINF:   .equ     $-EXLEN              ;: SEM exinf define area
SEME_TOP:       .equ     $                    ;:
sem1_exinf:     .data.l  00000000             ;: semid = 1 exinf        }
sem2_exinf:     .data.l  00000000             ;: semid = 2 exinf        } (1)
sem3_exinf:     .data.l  00000000             ;: semid = 3 exinf        }
sem4_exinf:     .data.l  00000000             ;: semid = 4 exinf        }
SEME_BTM:
SEMECNT:        .equ     (SEME_BTM-SEME_TOP) / EXLEN
                                   ;:[0...255]   ;: sem exinf count
;
```

Figure 3.41 Semaphore Extended Information Registration Field of Setup Table

```
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;%%%   Mailbox Extended Information define section                      %%%
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;------ Usage ------------------------------------------------------------------
;LABEL              .data.l  MBX?_EXINF       ;: COMMENT
;-------------------------------------------------------------------------------
                .section     h2ssetup, code, align = 2
_HI_MBXEXINF:   .equ     $-EXLEN              ;: MBX exinf define area
MBXE_TOP:       .equ     $                    ;:
mbx1_exinf:     .data.l  00000000             ;: mbxid = 1 exinf        }
mbx2_exinf:     .data.l  00000000             ;: mbxid = 2 exinf        } (1)
mbx3_exinf:     .data.l  00000000             ;: mbxid = 3 exinf        }
mbx4_exinf:     .data.l  00000000             ;: mbxid = 4 exinf        }
MBXE_BTM:
MBXECNT:        .equ    (MBXE_BTM-MBXE_TOP) / EXLEN
                                   ;:[0...255]   ;: mbx exinf count
;
```

Figure 3.42 Mailbox Extended Information Registration Field of Setup Table

```
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;%%%    Fixed-size MemoryPool Extended Information define section          %%%
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;------ Usage -------------------------------------------------------------------------------------------
;LABEL              .data.l  MPF?_EXINF        ;: COMMENT
;-------------------------------------------------------------------------------------------------------------
                  .section       h2ssetup, code, align = 2
_HI_MPFEXINF:   .equ     $-EXLEN               ;: MPF exinf define area
MPFE_TOP:        .equ     $                     ;:
mpf1_exinf:         .data.l  00000000             ;: mpfid = 1 exinf
mpf2_exinf:         .data.l  00000000             ;: mpfid = 2 exinf                     (1)
mpf3_exinf:         .data.l  00000000             ;: mpfid = 3 exinf
mpf4_exinf:         .data.l  00000000             ;: mpfid = 4 exinf
MPFE_BTM:
MPFECNT:          .equ    (MPFE_BTM-MPFE_TOP) / EXLEN
                                             ;:[0...255]     ;: mpf exinf count
;
```

Figure 3.43 Fixed-Length Memory Pool Extended Information Registration Field of Setup Table

```
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;%%%    Variable-size MemoryPool Extended Information define section       %%%
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;------ Usage -------------------------------------------------------------------------------------------
;LABEL              .data.l  MPL?_EXINF        ;: COMMENT
;-------------------------------------------------------------------------------------------------------------
                 .section       h2ssetup, code, align = 2
_HI_MPLEXINF:   .equ     $-EXLEN               ;: MPL exinf define area
MPLE_TOP:        .equ     $                     ;:
mpl1_exinf:         .data.l  00000000             ;: mplid = 1 exinf
mpl2_exinf:         .data.l  00000000             ;: mplid = 2 exinf                     (1)
mpl3_exinf:         .data.l  00000000             ;: mplid = 3 exinf
mpl4_exinf:         .data.l  00000000             ;: mplid = 4 exinf
MPLE_BTM:
MPLECNT:          .equ     (MPLE_BTM-MPLE_TOP) / EXLEN
                                             ;:[0...255]     ;: mpl exinf count
;
```

Figure 3.44 Variable-Length Memory Pool Extended Information Registration Field of Setup Table

```
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;%%%    Cyclic Handler Extended Information define section                %%%
;%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
;------ Usage ---------------------------------------------------------------------------------------------
;LABEL              .data.l  CYH?_EXINF         ;: COMMENT
;----------------------------------------------------------------------------------------------------------
                  .section       h2ssetup, code, align = 2
_HI_CYCEXINF:   .equ      $-EXLEN               ;: CYH exinf define area
CYHE_TOP:        .equ     $                     ;:
cyh1_exinf:        .data.l  00000000             ;: cyhno = 1 exinf            }
cyh2_exinf:        .data.l  00000000             ;: cyhno = 2 exinf            } (1)
cyh3_exinf:        .data.l  00000000             ;: cyhno = 3 exinf            }
cyh4_exinf:        .data.l  00000000             ;: cyhno = 4 exinf            }
                   .aifdef  DX
cyh5_exinf:        .data.l  00000000             ;: cyhno = 5 exinf        ———— (2)
                   .aendi
CYHE_BTM:
CYHECNT:          .equ     (CYHE_BTM-CYHE_TOP) / EXLEN
                                       ;:[0...255]     ;: cyh exinf count
;
```

Figure 3.45 Cyclic Handler Extended Information Registration Field of Setup Table

(1) Definition of extended information

Defines extended information to be registered in each object.

(2) When the debugging extension is used, defines extended information to be registered in the debug daemon cyclic handler.

Note: Do not modify or delete symbols used in the extended information registration fields.

When adding extended information to be registered, insert the definition data before each ???E_BTM.

3.2.4 FAQ about Setup Table

This section answers a question about the setup table which is frequently asked by users of the HI series OS.

FAQ Contents:

(1) Optimizing Setup Table

Classification: Configuration, kernel environment definition, and setup table					
Question				HI2000/3	
When the system is created by using the files generated when the OS is installed without change, an error occurs and a correct system is not created. What causes this problem?					
Answer An error occurs because the setup table is specified for optimization. Do not specify the setup table for optimization. The setup table creates information (data table) required for the kernel according to the defined contents, as well as allocation of the memory area used by the kernel according to the defined value (such as calculation of stack size used by the kernel). Since no code (program) is described, the setup table does not affect the code size or speed (performance) even if it is specified for optimization. If assembly is performed with the setup table specified for optimization, an error occurs during optimization.					

3.3 Stack Size Calculation

Calculate the task or interrupt handler stack size using the following procedures.

1. Calculate the stack size for each function in a task or interrupt handler
2. Calculate the stack size considering program nesting

3.3.1 Stack Size Calculation from Stack Frame Size

A C function allocates a stack frame in the stack area when the function is initiated.

The stack frame is used as a local variable area for the function or as a parameter area for a function call.

The stack frame size can be determined from the frame size in the compile listing output by the C compiler.

As the C compiler cannot determine the stack size when service calls of the HI series OS are used, such extra stack size must be added to the frame size in the compile listing.

3.3.2 Stack Size Calculation by CallWalker

The stack size can be calculated using the "CallWalker", a tool supplied with the C compiler.

A calculation example of the task stack size using the CallWalker is shown below.

The following calculation example uses the HI7750/4, SuperH™ RISC engine Series C/C++ Compiler Package Ver. 8.0.01, and SH7770 whole linkage project (7770_mix) as the sub-project of the HEW workspace.

(1) Starting HEW

Initiate the HEW, open "\kernel\for_shc8\hios\hios.hws" in the HI7750/4 install folder, and select 7770_mix as the current project.

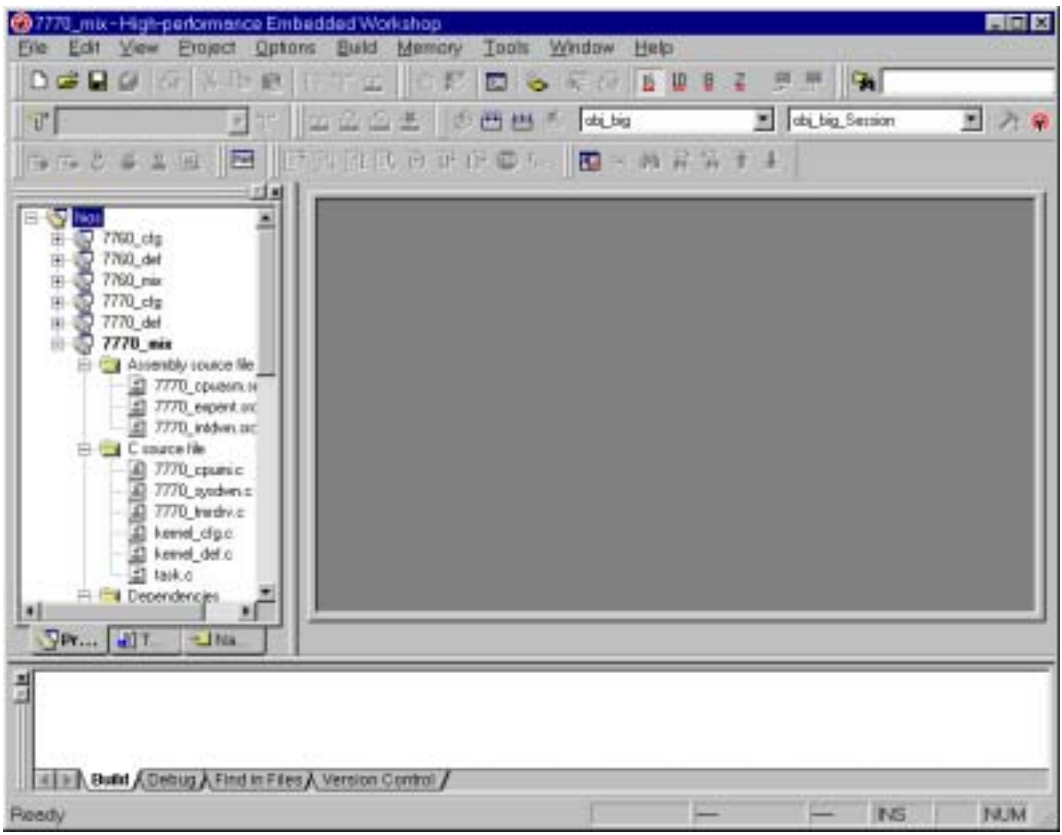


Figure 3.46 HEW Startup

In the window after setting the current project, select [SuperH RISC engine Standard Toolchain...] from [Options] in the header menu to display the HEW option setting menu.

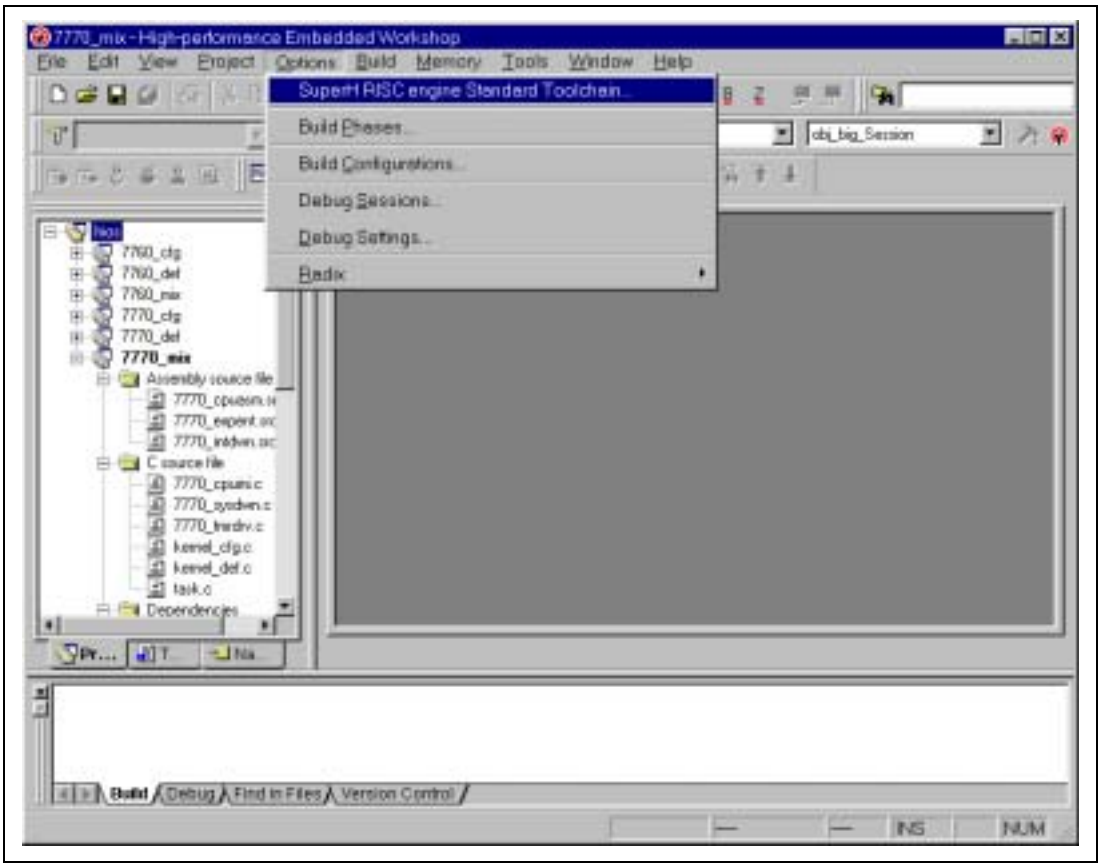


Figure 3.47 Menu Selection

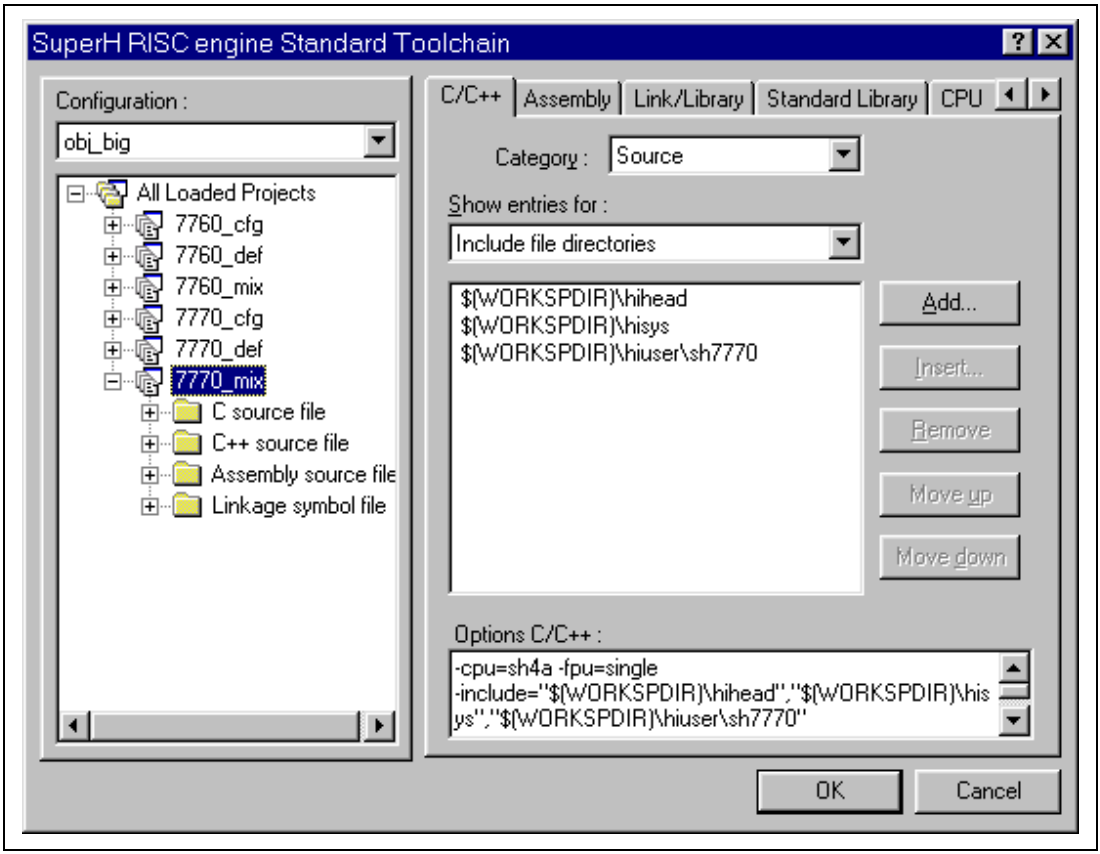


Figure 3.48 HEW Option Selection

Select "Other" for [Category] in the [Link/Library] tab and select [Stack information output].

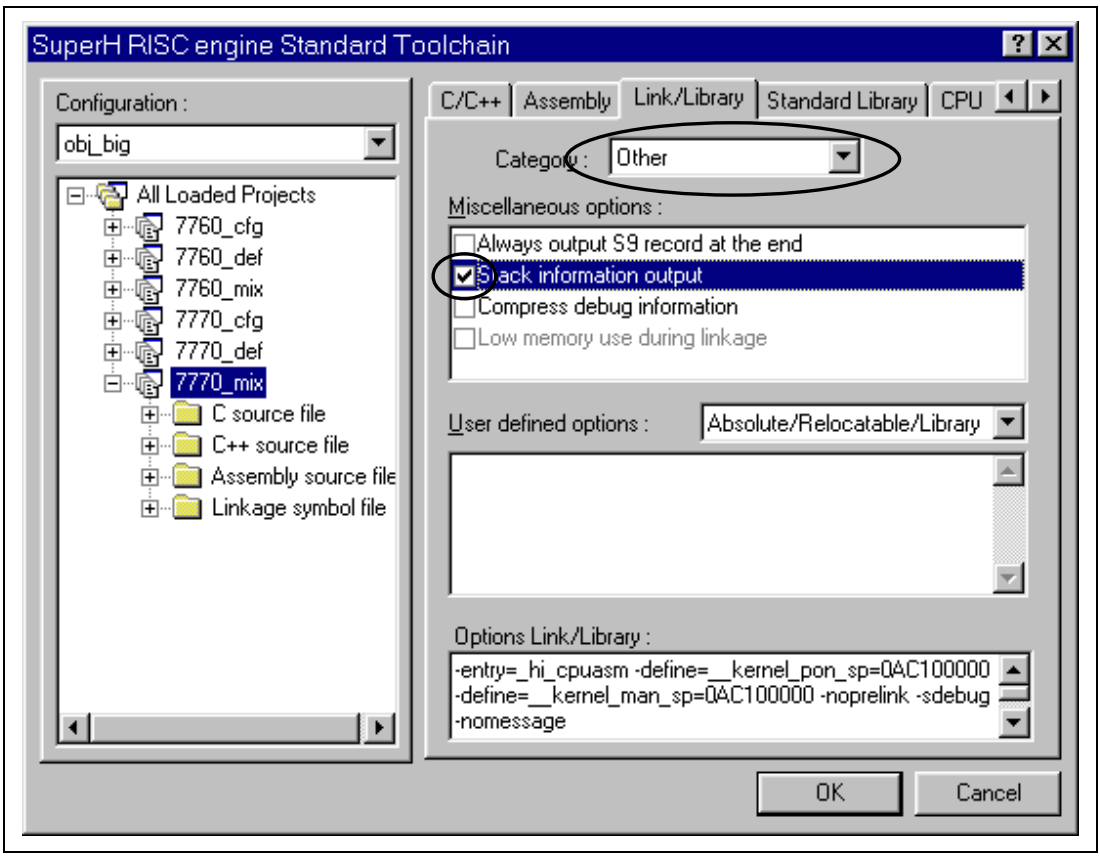


Figure 3.49 HEW Option Settings

Click the [OK] button to finish setting and execute build.

(2) Starting CallWalker

Select [Program] -> [Renesas High-performance Embedded Workshop] -> [CallWalker] to initiate the CallWalker.

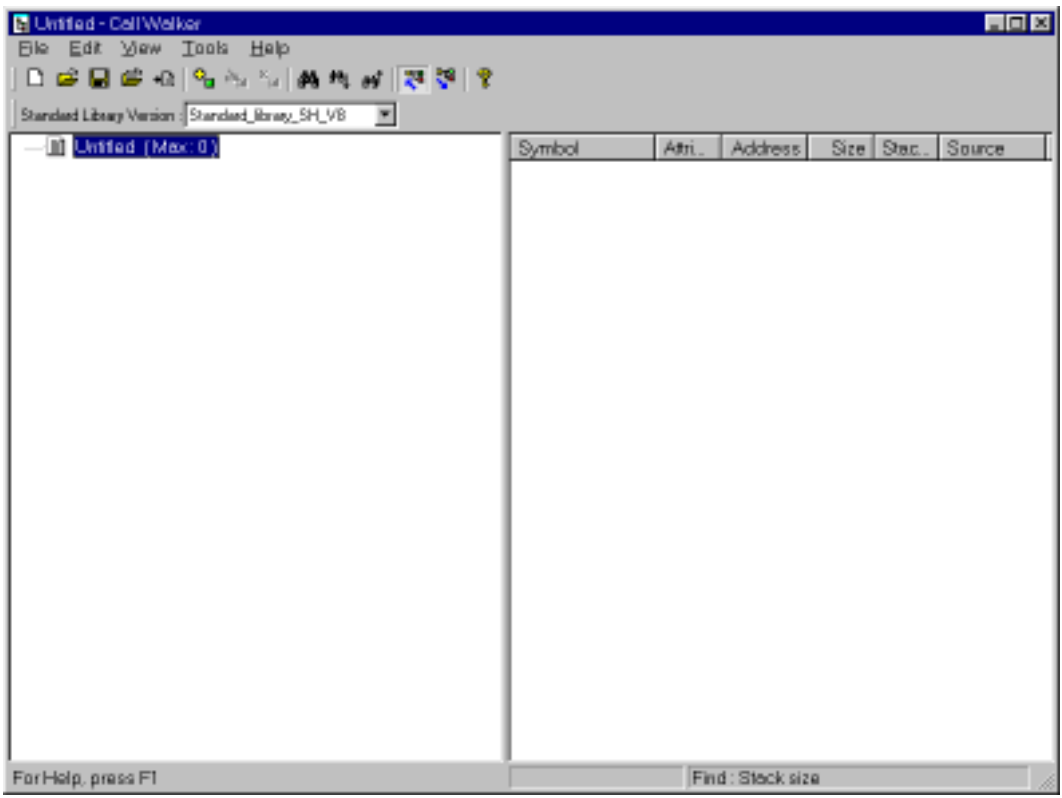


Figure 3.50 CallWalker Startup

Select [Import Stack File...] from [File] in the header menu of the startup window to open the created stack information file.

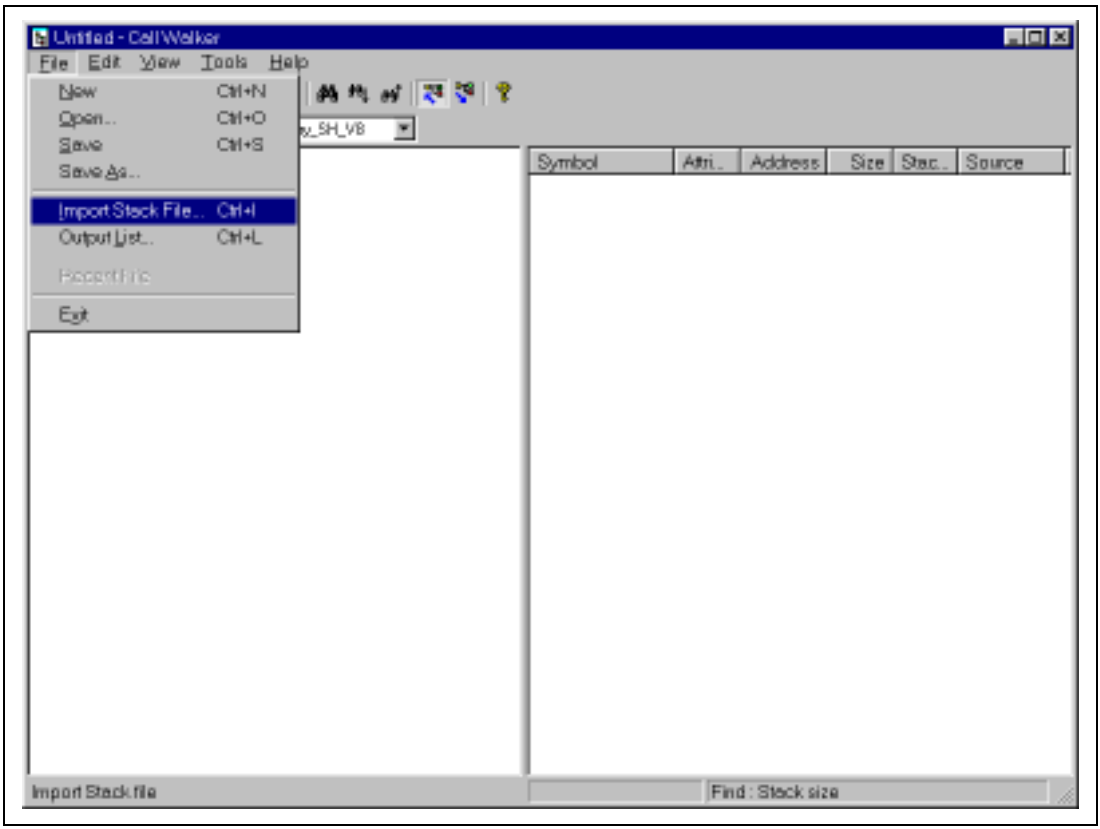


Figure 3.51 File Reading

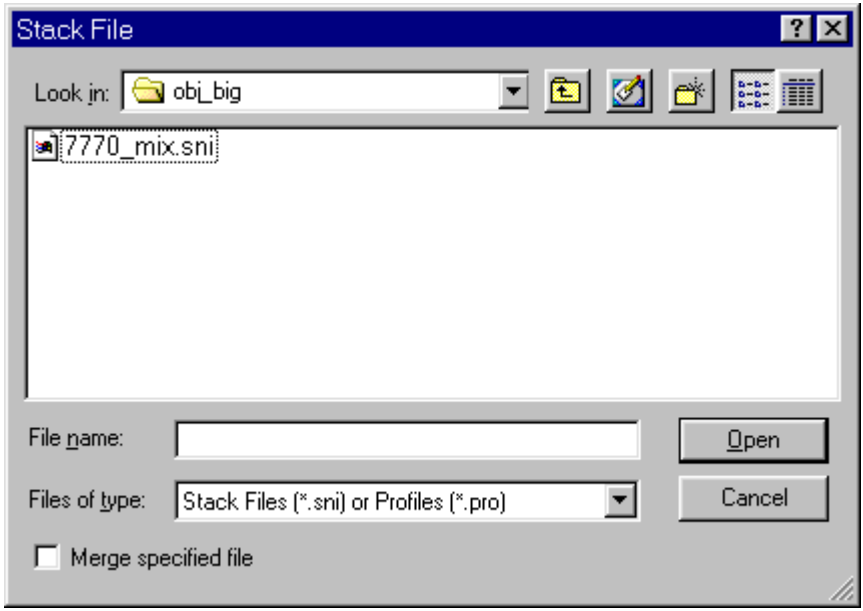


Figure 3.52 Read File Selection

(3) Calculation example of task stack size

In this example, the system consists of the application programs listed in table 3.21.

Table 3.21 Configuration of Sample System

Function Name	Application Type	Remarks
_hi_cpuini	CPU initialization routine	
_kernel_reset	(Calls vsta_knl)	Stack size is calculated as 0
_inithdr1	Initialization routine	
_MainTask	Task	
_texrtn1	Task exception processing routine for _MainTask	
_sub1	Function called from _MainTask	
_Task7	Task	
_svchdr1	Extended service call handler	Called from _MainTask
_inthdr_level1	Interrupt handler (interrupt level 1)	
_inthdr_level5	Interrupt handler (interrupt level 5)	
_kernel_tmrini	Timer driver (timer initialization routine)	Initialization routine
_kernel_tmrint	Timer driver (timer interrupt routine)	Interrupt handler
_cychdr1	Cyclic handler	
_kernel_sysdwn	System down	

In the above application, the static stack and coprocessor are assumed not to be used and the CFG_TRACE check disabled.

An example of stack size display by the CallWalker for the above application is shown in figure 3.53.

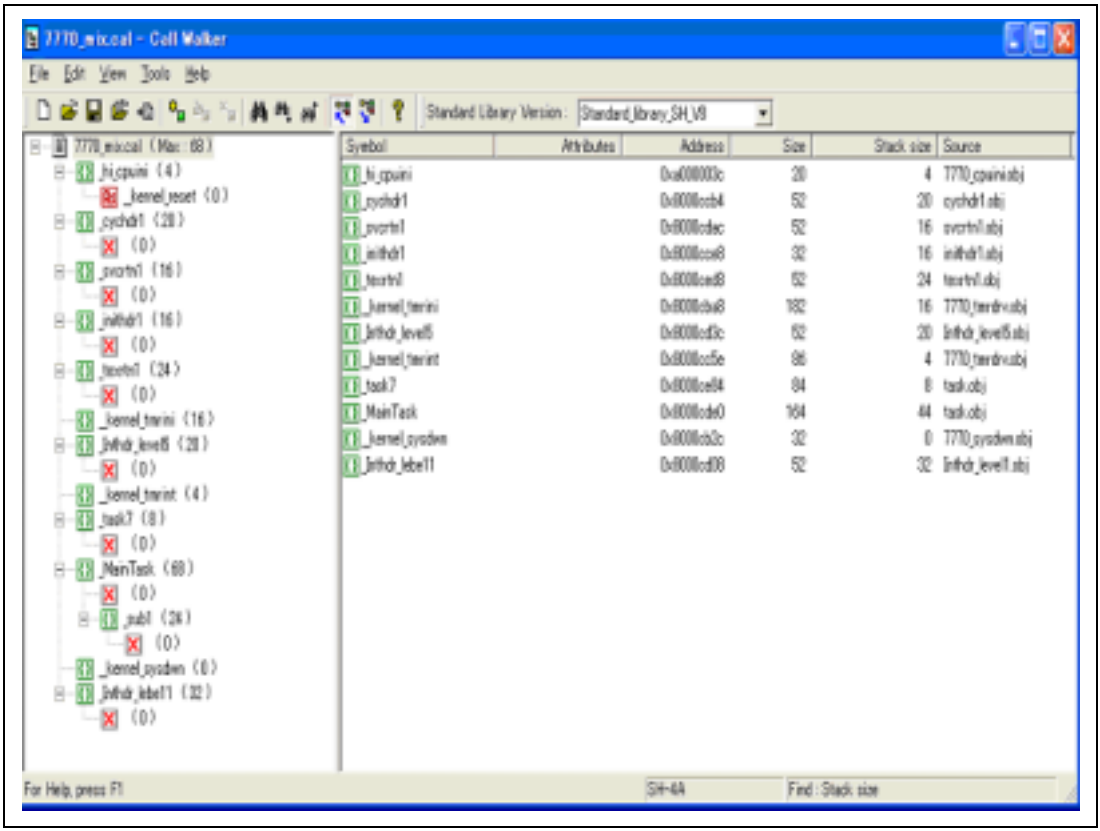


Figure 3.53 Stack Size Display Example by CallWalker

The stack size is calculated from the displayed information. The stack size displayed by the CallWalker is the stack size which a task or interrupt handler can use independently. The stack size can be obtained by adding the necessary size of the kernel to this displayed size. Each stack size is calculated below based on the displayed stack size example by the CallWalker.

The stack size of the "_MainTask" task is calculated as an example.

_MainTask calls the following function and service call. It also defines a task exception processing routine.

— _sub1

— Extended service call routine (_svcrtn1)

— Task exception processing routine (_texrtn1)

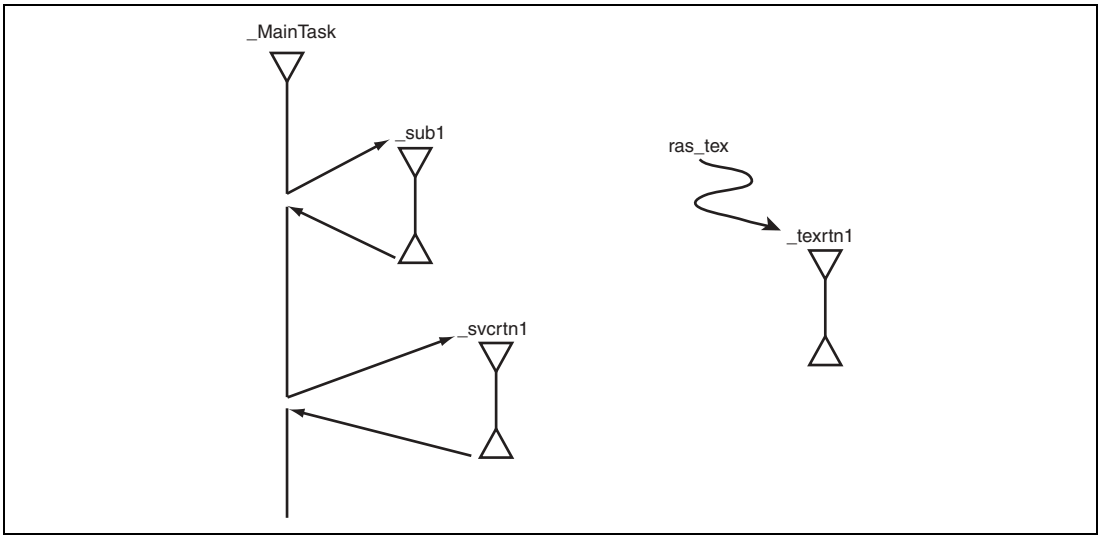


Figure 3.54 Overview of Sample Task Processing

The Call Information View of the CallWalker indicates that _MainTask calls the _sub1 function. However, information for the other calls (e.g. service call) is not available. Since the CallWalker cannot display information for the other two calls, calculation must be performed manually.

The stack sizes of _MainTask and _sub1 can be obtained from the Call Information View and Symbol Detail View, respectively. Add manually the stack sizes of the other extended service call routine and task exception processing routine to these stack sizes.

The stack size of the "_MainTask" task alone becomes as shown in table 3.22.

Table 3.22 Stack Size of _MainTask Itself

No.	Function Name	Stack Size
1	_MainTask	44 bytes
2	_sub1	24 bytes
3	_svcrtn1	16 bytes
4	_texrtn1	24 + 152 bytes*
Total		260 bytes

Note: Added size (necessary size) of call routine and handler. For details, refer to the HI7000/4 Series User's Manual.

The value determined here is the stack size of the "_MainTask" task itself.

Substitute this value into item 1 in table C.5, Task Stack Size, in the HI7000/4 Series User's Manual. The stack size of the "_MainTask" task is determined as shown in table 3.23.

Table 3.23 Task Stack Size Calculation

No.	Item		Stack Size
1	Obtained size		260 bytes
2	Necessary size		196 bytes
3	Tasks	TA_COP0 attribute	—
4		TA_COP1 attribute	—
5		TA_COP2 attribute	—
6		Static stack usage	—
7	Checks CFG_TRACE		—
8	Addition considering nested interrupts		—
Total			456 bytes

(4) Calculation example of interrupt handler stack size

In this example, there are two interrupt handlers.

— inthdr_level1

— inthdr_level5

In addition, a timer is used. The stack size of each interrupt handler needs to be determined because these interrupt handlers have different interrupt levels. Accordingly, nesting does not need to be considered for these interrupts.

Substitute each stack size into item 1 in table C.6, Interrupt Handler Stack Size, in the HI7000/4 Series User's Manual.

Table 3.24 Interrupt Handler Stack Size Calculation

No.	Item	Stack Size		
		_inthdr_level1	_inthdr_level5	_kernel_tmrint
1	Obtained size	32 bytes	20 bytes	4 bytes
2	Calls service call from the interrupt handlers	192 bytes	192 bytes	192 bytes
3	Checks CFG_TRACE	—	—	—
4	Addition considering nested interrupts	—	—	—
Total		224 bytes	212 bytes	196 bytes

The interrupt handler stack size to be specified is determined from these values.

Substitute these values into the following formula provided in the HI7000/4 Series User's Manual to obtain the interrupt handler stack size.

CFG_IRQSTKSZ = ∑ (The stack area of the handler that uses the largest stack area) + 28 + (stack size used by the NMI interrupt handler calculated as shown in appendixes C.4 and C.5 + 48) × NMI nest count

The result is as follows:

CFG_IRQSTKSZ = 224 + 212 + 196 + 28 + 0 (no NMI nesting)
= 660 bytes

(5) Calculation example of time event handler stack size

In this example, only one cyclic handler (_cychdr1) is used.

Substitute this value into item 1 in table C.7, Time Event Handler Stack Size, in the HI7000/4 Series User's Manual.

Table 3.25 Time Event Handler Stack Size Calculation

No.	Item	Stack Size
1	Obtained size	20 bytes
2	Necessary size	192 bytes
3	Calls service call from the time event handlers	144 bytes
4	Checks CFG_TRACE	—
5	Addition considering nested interrupts	—
6	Addition when the NMI is used	—
Total		356 bytes

Only one cyclic handler is used as the time event handler in this example. When more than one time event handler is used, calculate the stack size using the maximum size of all time event handlers that use the stack.

(6) Calculation example of initialization routine stack size

In this example, one initialization routine (inithdr1) is used.

However, since a timer driver is used, the timer initialization handler "_kernel_tmrini" of the timer driver is actually used, resulting in a total of two initialization routines being used.

Therefore, use the greater stack size among these two for calculating the initialization routine stack size.

Table 3.26 Initialization Routine Stack Size Calculation

No.	Item	Stack Size
1	Obtained size	16 bytes
2	Necessary size	192 bytes
3	Checks CFG_TRACE	—
4	Addition when the NMI is used	—
Total		208 bytes

(7) Notes on using CallWalker

The notes when using the CallWalker are listed below.

— [RealTime OS Option...] in the Tools menu of the CallWalker is currently not supported.

— Assembly-language functions will not be calculated by the CallWalker, so they need to be calculated manually.

— The following functions will also not be calculated by the CallWalker, so they need to be calculated manually.

- Recursive function
- Circular function
- Function having an unclear source symbol
- Function having an address still not referenced

Note that when the function at the beginning of an application program, such as the starting function of a task or task exception processing routine is written in the assembly language, it may not be displayed in the Call Information View of the CallWalker.

3.4 System Configuration Procedure

A system using the HI series OS is configured using the HEW (High-performance Embedded Workshop).

The overview of system configuration is shown in figure 3.55.

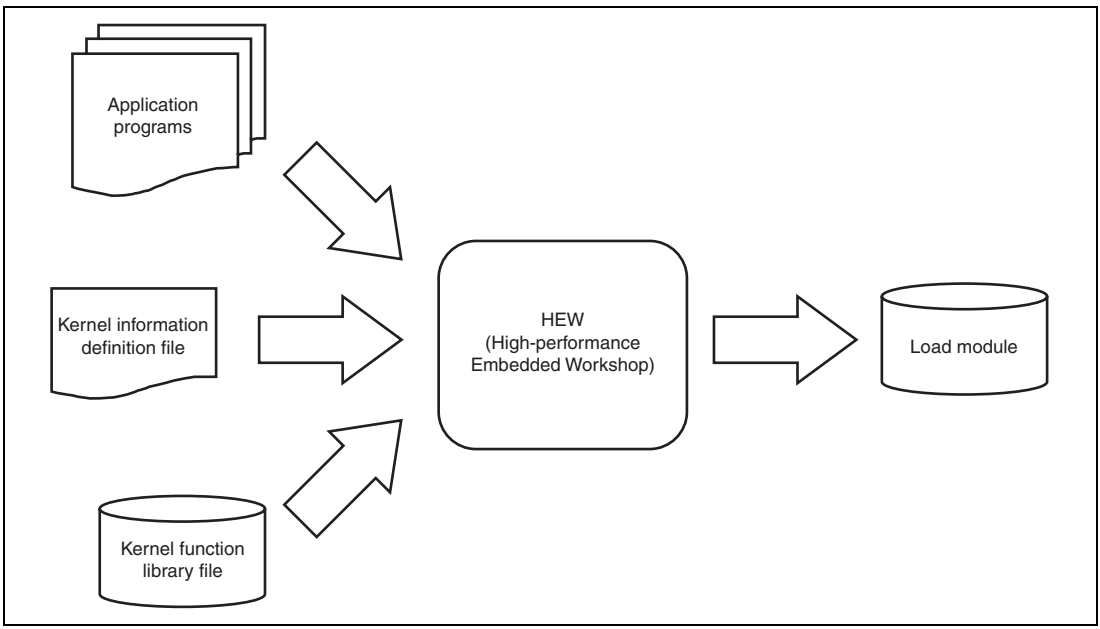


Figure 3.55 System Configuration Procedure

Each HI series OS has a HEW configuration file (HEW workspace) for the supplied standard sample programs.

The configuration procedure using the supplied standard HEW configuration file is described below.

3.4.1 HI7000/4

For the configuration procedure of the HI7000/4, the SuperH™ RISC engine Series C/C++ Compiler Package Ver. 6.0AR2, SH7612 HEW configuration file (referred to as HEW workspace), and Configuration Guide using whole linkage are provided. These can be downloaded from the website of Renesas Technology Corp.

3.4.2 HI7700/4

For the configuration procedure of the HI7700/4, the SuperH™ RISC engine Series C/C++ Compiler Package Ver. 6.0AR2, SH7729 HEW configuration file (referred to as HEW workspace), and Configuration Guide using whole linkage are provided. These can be downloaded from the website of Renesas Technology Corp.

3.4.3 HI7750/4

For the configuration procedure of the HI7750/4, the SuperH™ RISC engine Series C/C++ Compiler Package Ver. 6.0AR2, SH7750 HEW configuration file (referred to as HEW workspace), and Configuration Guide using whole linkage are provided. These can be downloaded from the website of Renesas Technology Corp.

3.4.4 HI2000/3

The configuration procedure using the HEW is shown below.

In this example, the H8S, H8/300 Series C/C++ Compiler Package Ver. 4.0AR2 is used.

Double-clicking the sample workspace file "product.hws" in the HI2000/3 installation folder "product" launches the HEW for configuring the HI2000/3. The HEW startup window is shown in figure 3.56.

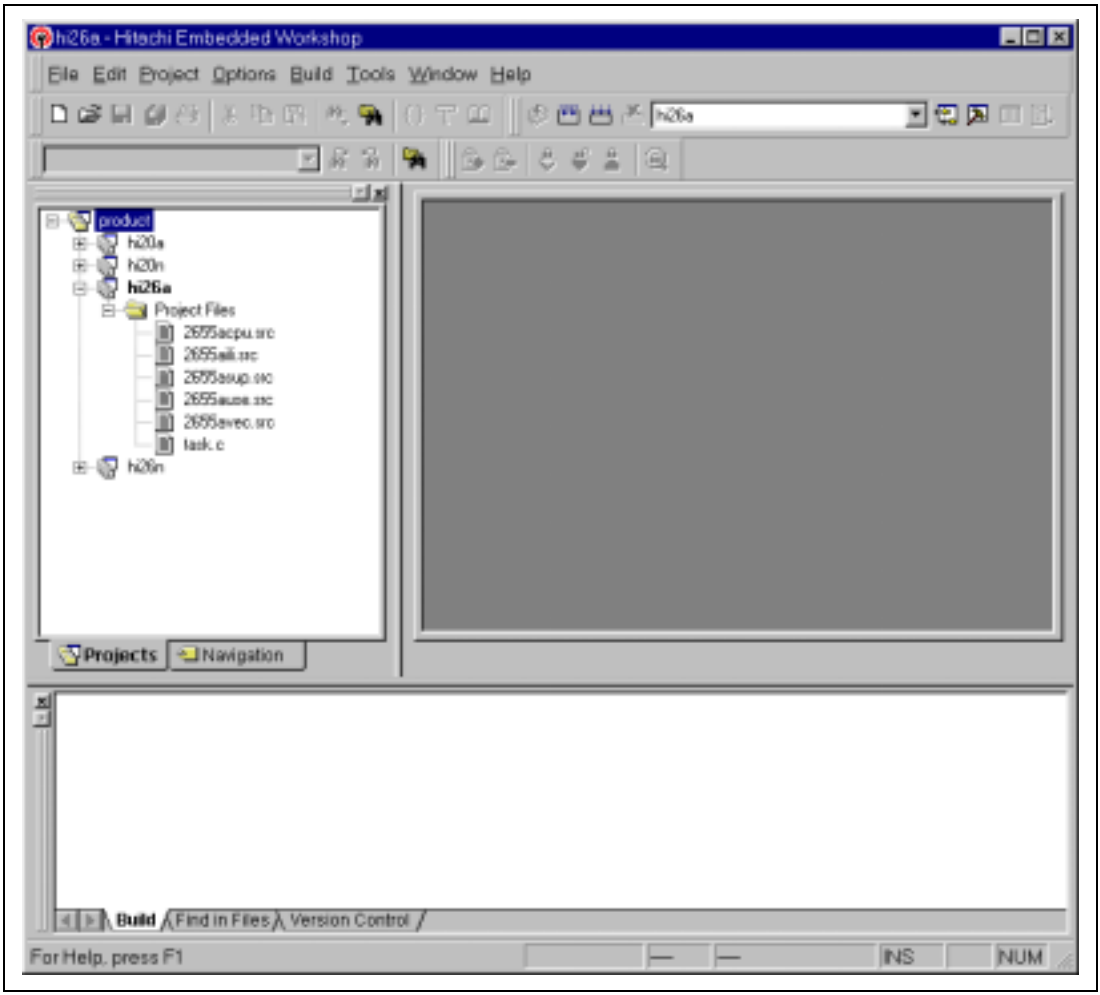


Figure 3.56 HEW Startup

Sample projects corresponding to each device are already registered in the workspace file "product.hws".

There are four sample projects corresponding to the CPU and operating modes as shown in table 3.27.

Select a project that matches the user environment (CPU and operating mode) and change the settings with reference to the subsequent descriptions.

Table 3.27 Standard Sample Projects

No.	Project Name	Configuration File*	Contents
1	hi26a	hi26a	Project to create a load module for the H8S/2600 CPU in advanced mode (already registered for H8S/2655)
2	hi26n	hi26n	Project to create a load module for the H8S/2600 CPU in normal mode (already registered for H8S/2655)
3	hi20a	hi20a	Project to create a load module for the H8S/2000 CPU in advanced mode (already registered for H8S/2655)
4	hi20n	hi20n	Project to create a load module for the H8S/2000 CPU in normal mode (already registered for H8S/2655)

Note: A setting is made to create a load module in the configuration file.

To select a sample project, select a project from the HEW workspace window and select [Set as Current Project] from the pop-up menu.

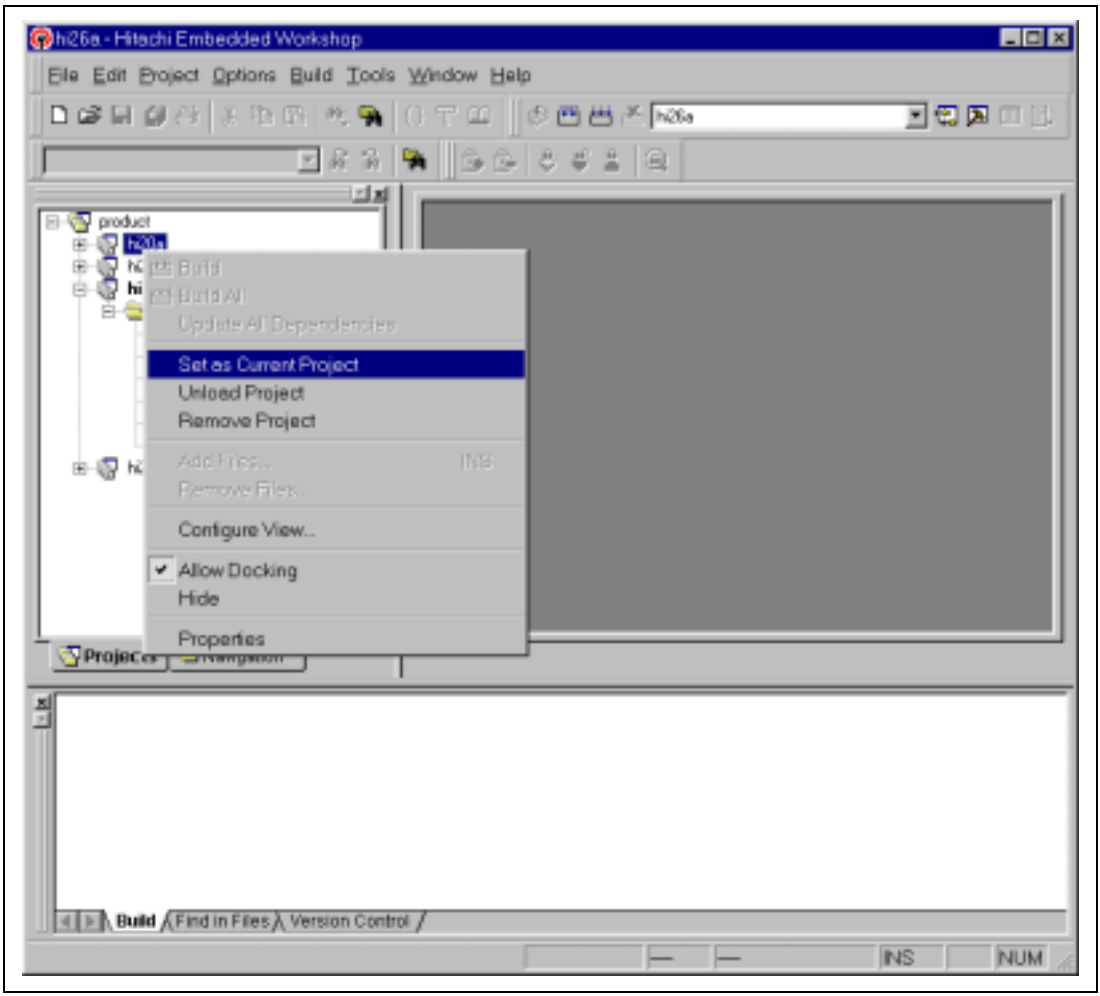


Figure 3.57 Project Selection from Pop-up Menu

Note that projects for unused environments can be deleted.

When using a device other than the H8S/2655 or H8S/2245, after selecting a project, change the system configuration file already registered to that for the CPU used.

Define (add) the application programs created in section 2, Application Program Creation, in the project file. The procedure for adding files is shown below.

Select [Add Files...] from [Project] in the header menu in the window after setting the current project, and add the created application program files to the project file.

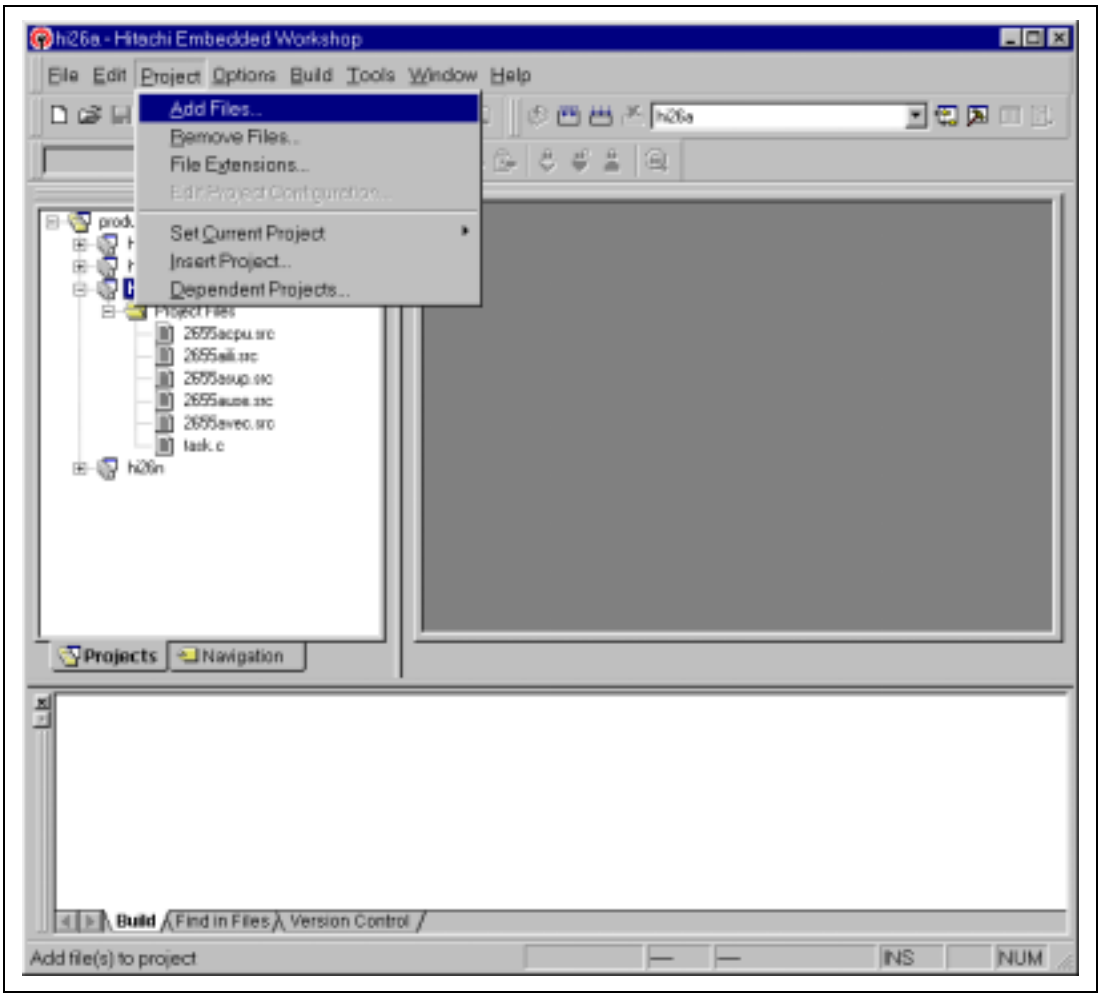


Figure 3.58 File Addition Menu

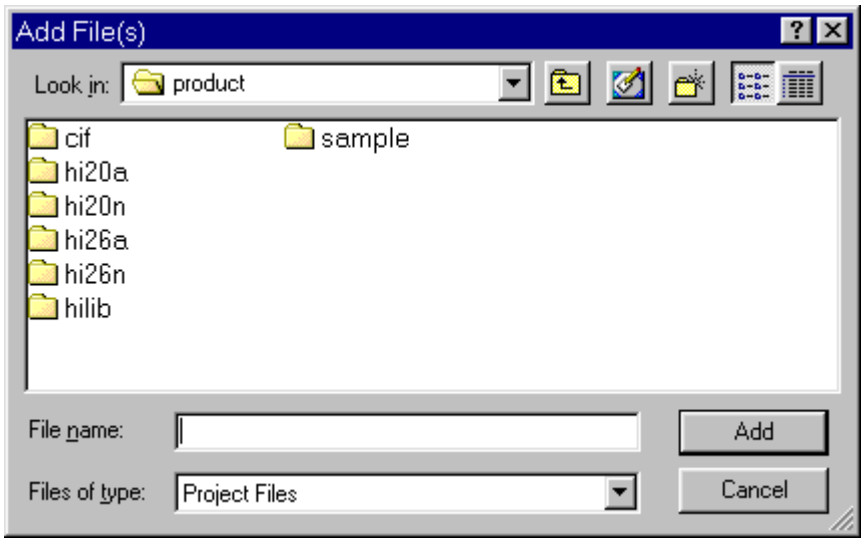


Figure 3.59 Additional File Selection

In the additional file selection window, more than one file can be selected simultaneously by moving to the folder containing the files to be added and then selecting the files with the Shift key pressed down.

Define the section information of the added files.

Select [OptLinker...] from [Options] in the header menu, select the [Section] tab of the [OptLinker options (hi26a)] dialog box, and make settings to add the section information.

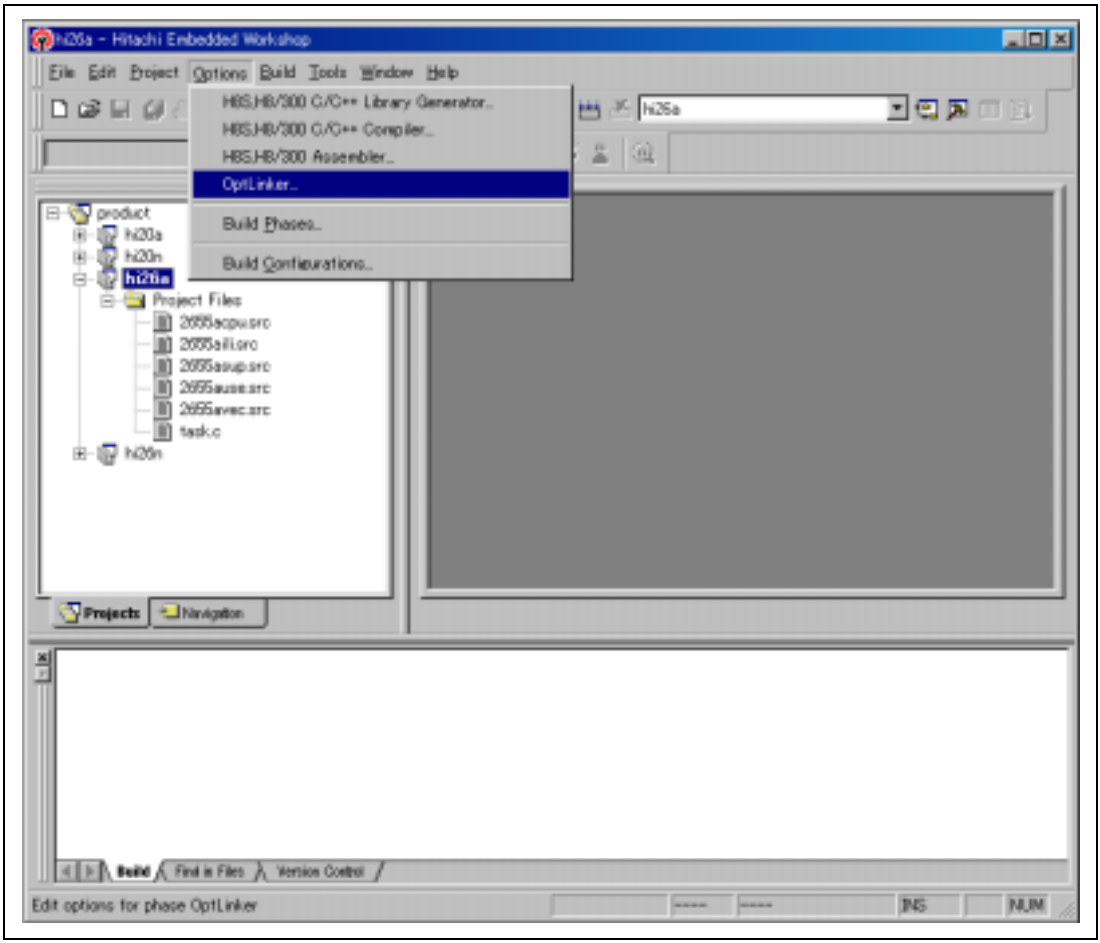


Figure 3.60 OptLinker Selection Menu

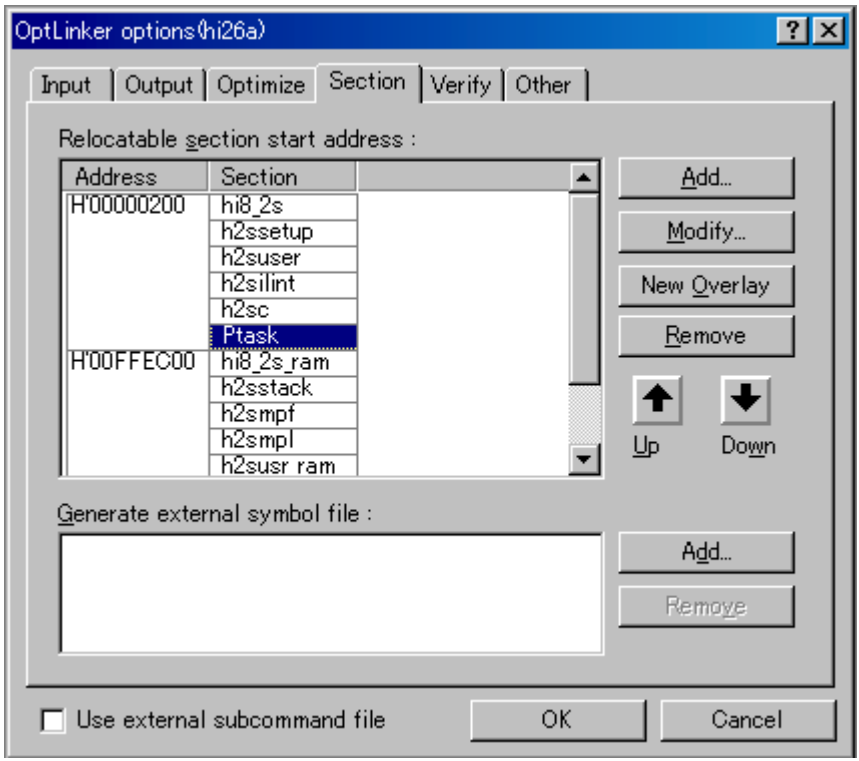


Figure 3.61 Section Information Addition

How to add a section is described next. Adding program section "P_section" of the added application file is shown as an example.

Select [Ptask] and press the [Add...] button.

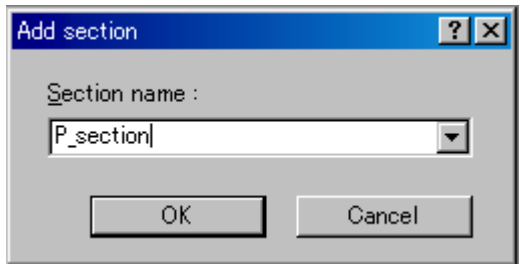


Figure 3.62 Additional Section Information Input

Input "P_section" in [Section name :] in the [Add section] dialog box and press the [OK] button. The added "P_section" section will be displayed below the "Ptask" section.

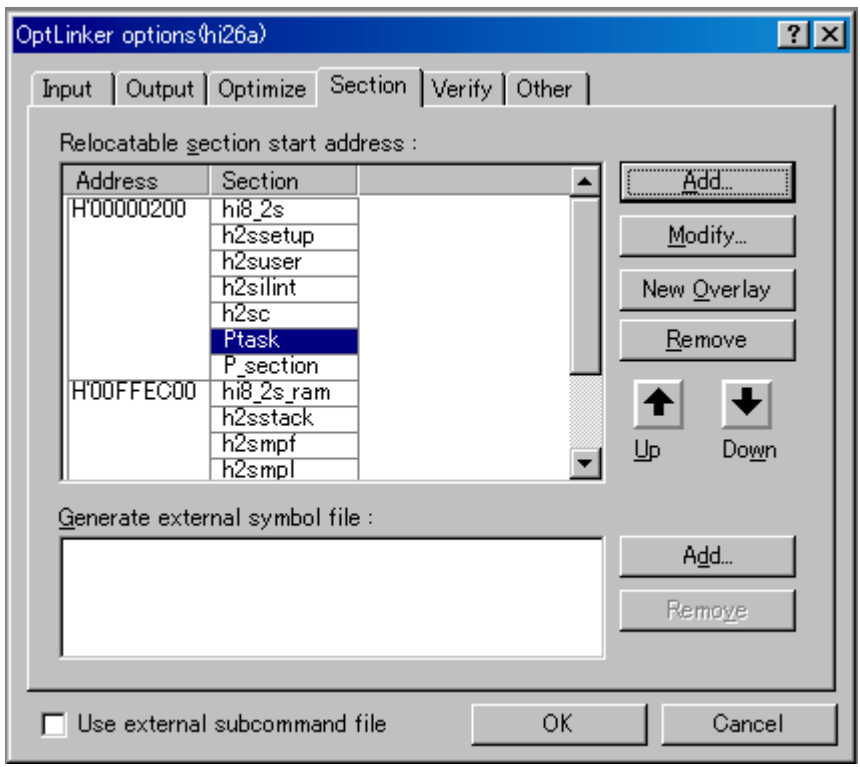


Figure 3.63 Added Section Information Confirmation

To reflect the updated section information, press the [OK] button.

Next, select [Build] from [Build] in the header menu to configure (build) the system.

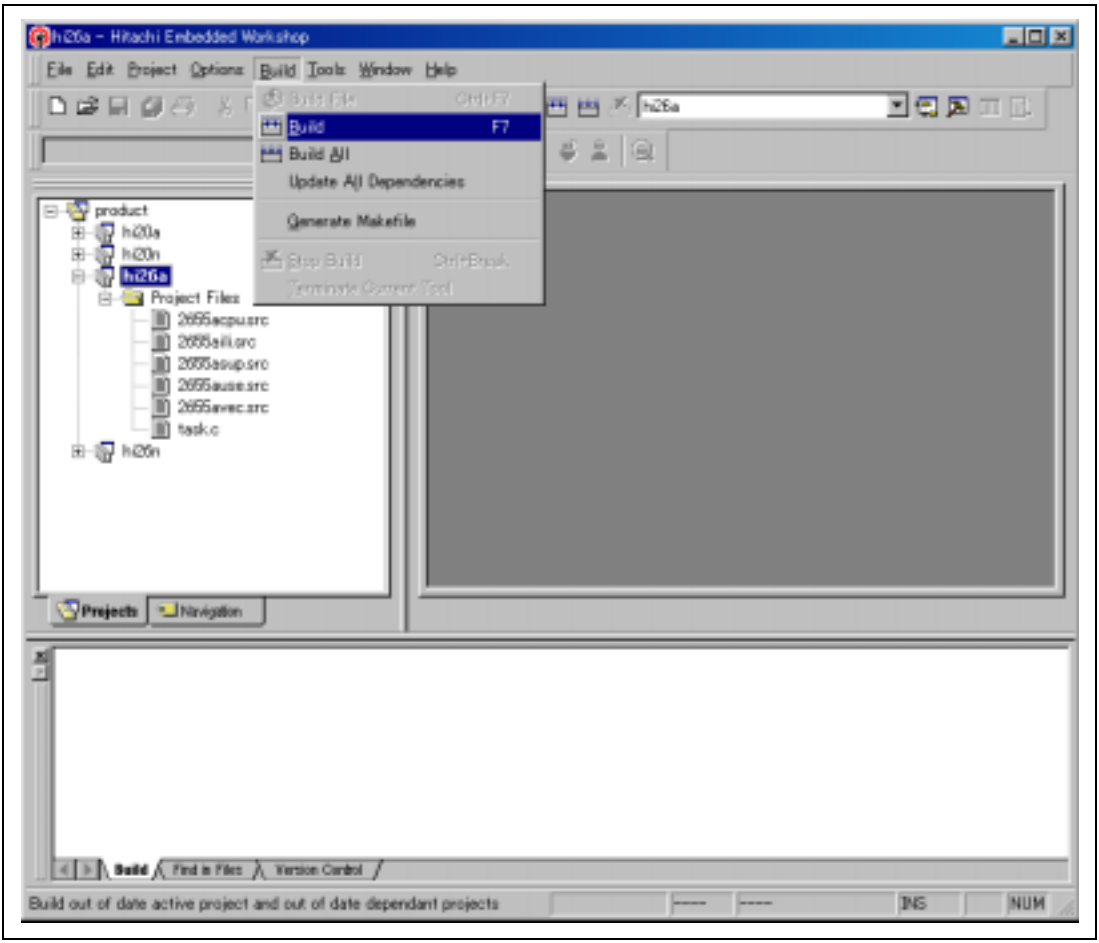


Figure 3.64 Build Execution

The above operations create an executable file.

Note that the result of compile, assemble, and linkage is displayed in the lowest part of the window. If an error occurs, after correcting the corresponding source program, re-execute build.

The executable file (extension of abs) is created in the folder (folder with the relevant project name under the [product] folder) specified by the configuration file selected in the relevant project.

Build by standard configuration uses a kernel library in which the parameter check function and shared-stack function are enabled.

After the application program has been debugged and it has reached a level to be embedded in a product, the parameter check executed at the beginning of the system call becomes a useless routine. Accordingly, this parameter check function can be removed in the HI2000/3.

For details on the how to remove the parameter check function, refer to section 1.3.2, Installation in HI2000/3 and HI1000/4.

3.4.5 HI1000/4

The configuration procedure using the HEW is shown below.

In this example, the H8S, H8/300 Series C/C++ Compiler Package Ver. 6.0.00 is used.

Double-clicking the sample workspace file "product.hws" in the HI1000/4 installation folder "product" launches the HEW for configuring the HI1000/4. The HEW startup window is shown in figure 3.65.

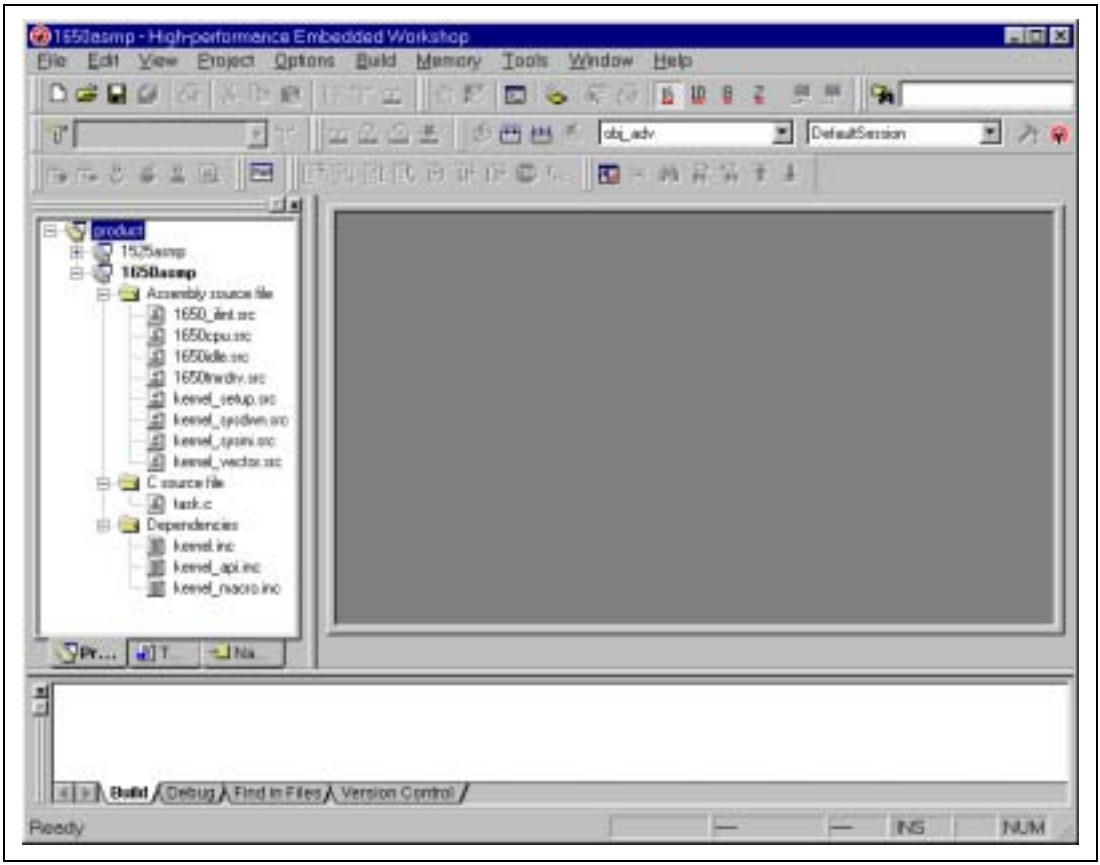


Figure 3.65 HEW Startup

Sample projects corresponding to each device are already registered in the workspace file "product.hws". There are two sample projects corresponding to the CPU and operating modes as shown below. Select a project that matches the user environment (CPU and operating mode) and change the settings with reference to the subsequent descriptions.

- 1650asmp: Project to create a load module for the H8SX/1650 CPU in advanced mode
- 1525asmp: Project to create a load module for the H8SX/1525 CPU in advanced mode

To select a sample project, select a project from the HEW workspace window and select [Set as Current Project] from the pop-up menu.

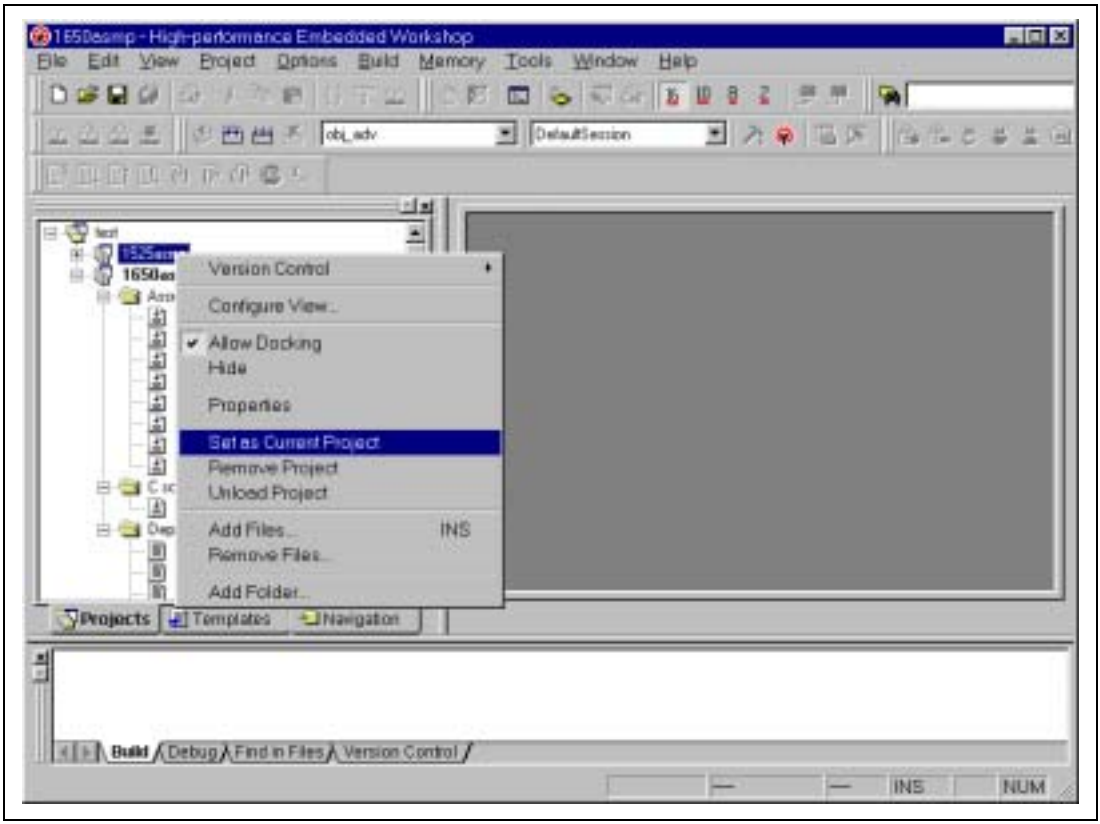


Figure 3.66 Project Selection from Pop-up Menu

Note that projects for unused environments can be deleted.

Define (add) the application programs created in section 2, Application Program Creation, in the project file. The procedure for adding files is shown below.

Select [Add Files...] from [Project] in the header menu in the window after setting the current project, and add the created application program files to the project file.

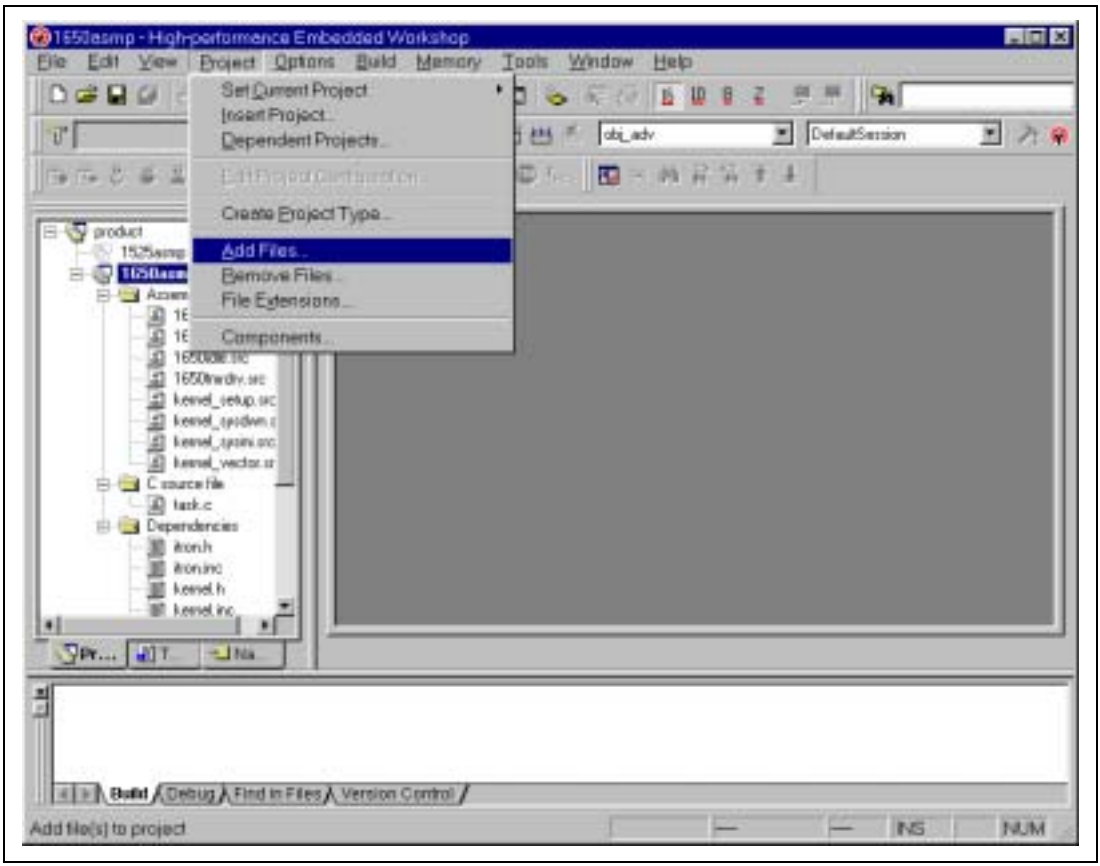


Figure 3.67 File Addition Menu

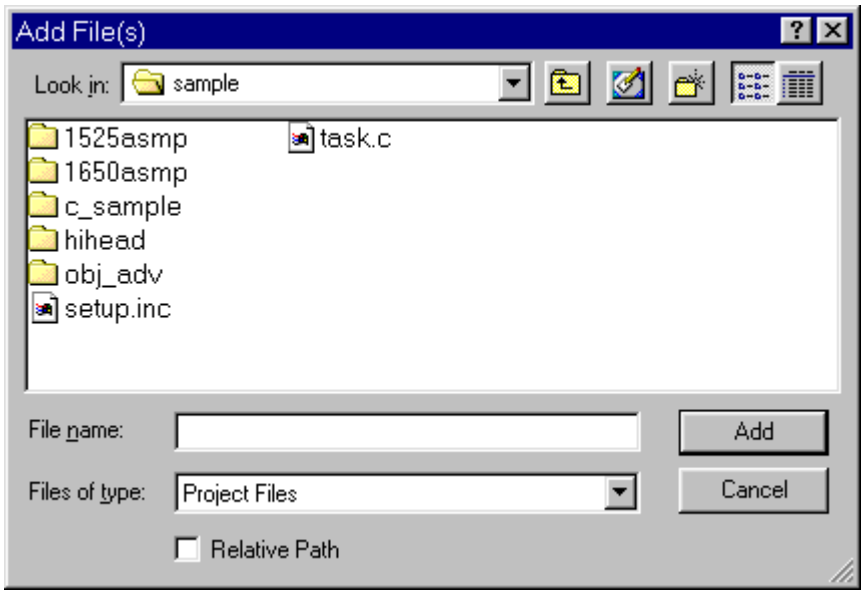


Figure 3.68 Additional File Selection

In the additional file selection window, more than one file can be selected simultaneously by moving to the folder containing the files to be added and then selecting the files with the Shift key pressed down.

Define the section information of the added files.

Select [H8S, H8/300 Standard Toolchain...] from [Options] in the header menu, select the [Link/Library] tab of the [H8S, H8/300 Standard Toolchain] dialog box, and make settings to add the section information.

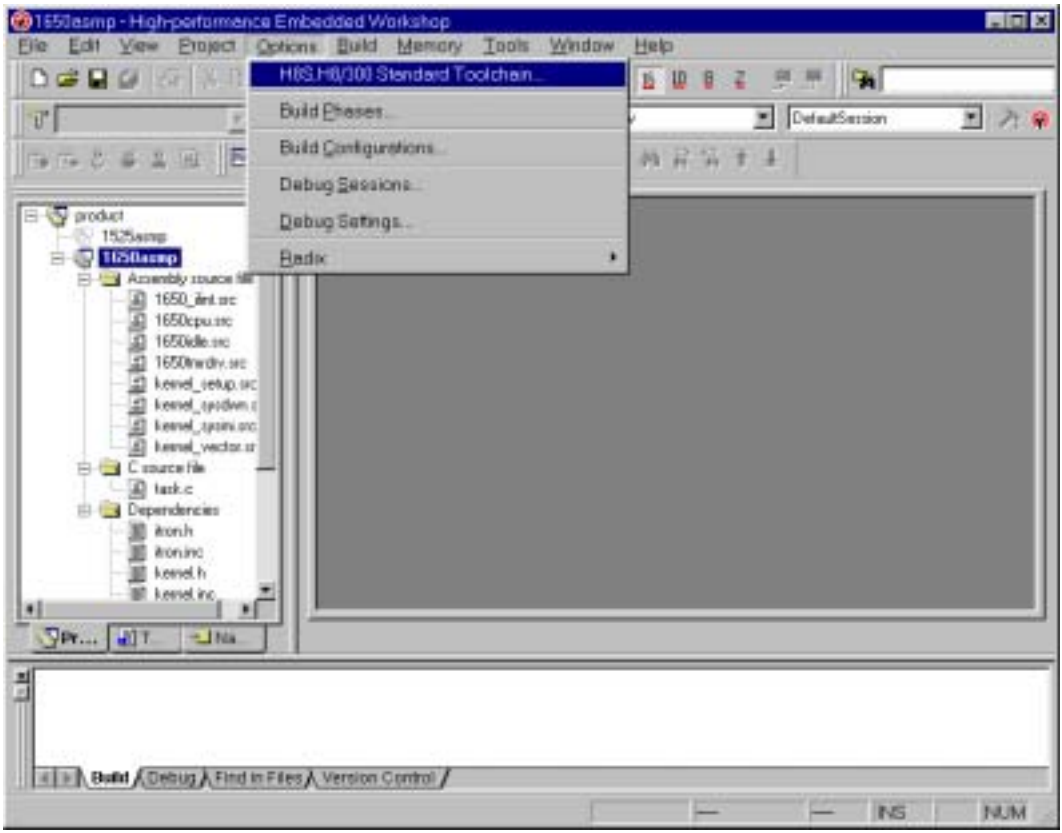


Figure 3.69 H8S, H8/300 Standard Toolchain Selection Menu

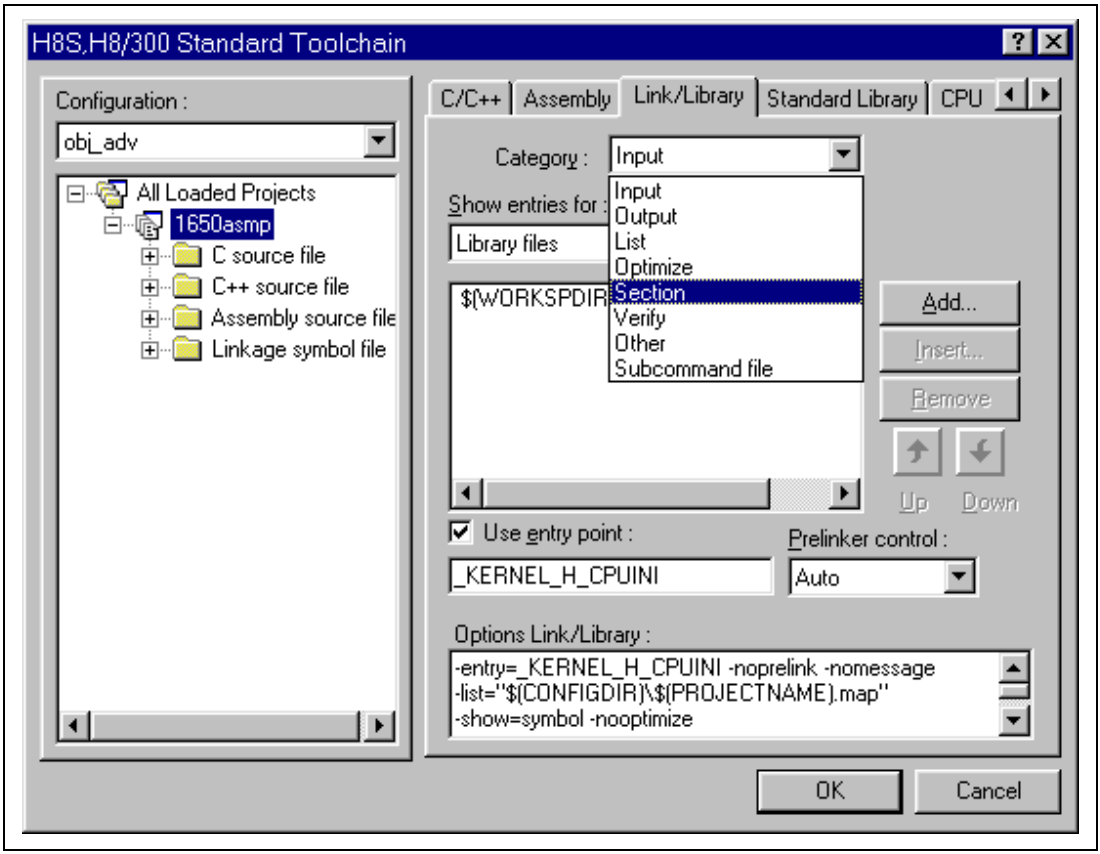


Figure 3.70 Section Setting Menu

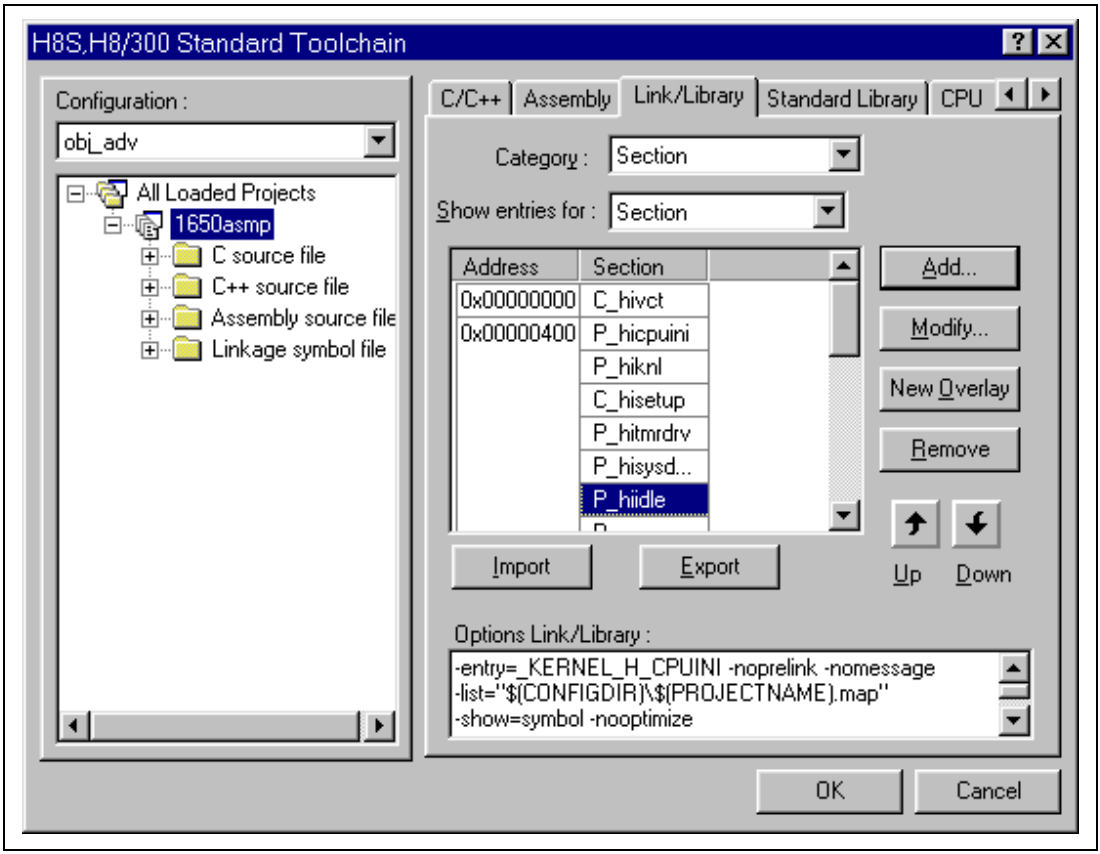


Figure 3.71 Section Information Addition

How to add a section is described next. Adding program section "P_section" of the added application file is shown as an example.

Select [P_hiidle] and press the [Add...] button.

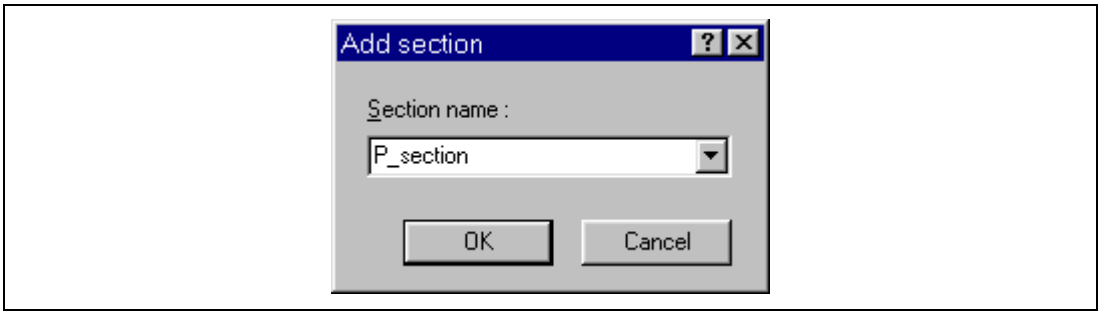


Figure 3.72 Additional Section Information Input

Input "P_section" in [Section name :] in the [Add section] dialog box and press the [OK] button. The added "P_section" section will be displayed below the "P_hiidle" section.

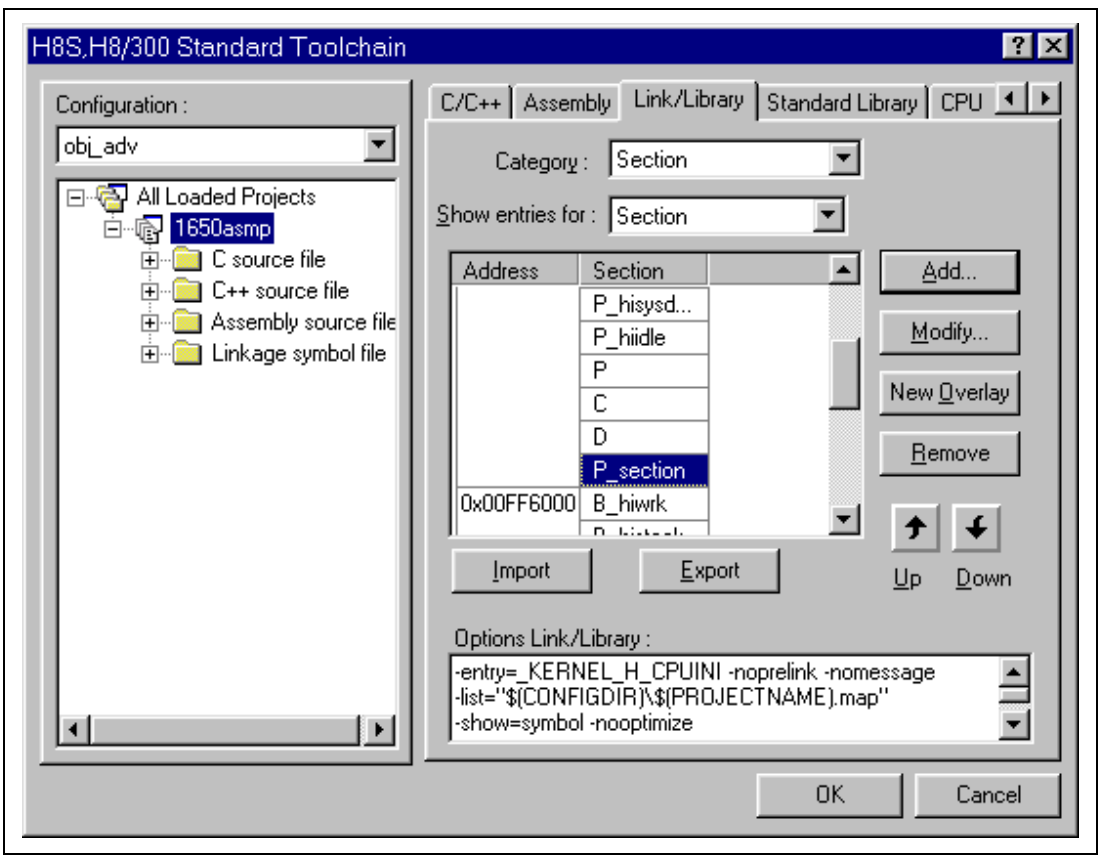


Figure 3.73 Added Section Information Confirmation

To reflect the updated section information, press the [OK] button.

Next, select [Build] from [Build] in the header menu to configure (build) the system.

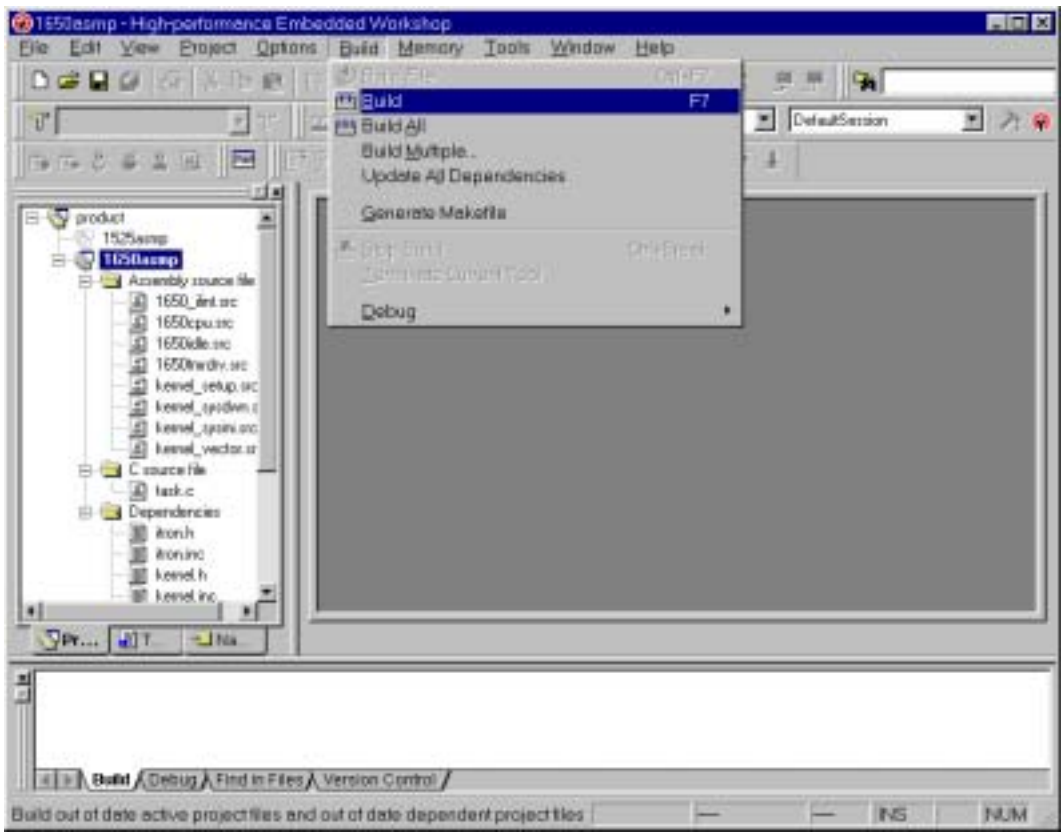


Figure 3.74 Build Execution

The above operations create an executable file.

Note that the result of compile, assemble, and linkage is displayed in the lowest part of the window. If an error occurs, after correcting the corresponding source program, re-execute build.

The executable file (extension of abs) is created in the folder ("obj_adv" folder under the [product] folder) specified by the relevant project.

Build by standard configuration uses a kernel library in which the parameter check function and shared-stack function are enabled.

After the application program has been debugged and it has reached a level to be embedded in a product, the parameter check executed at the beginning of the system call becomes a useless routine. Accordingly, this parameter check function can be removed in the HI1000/4.

For details on the how to remove the parameter check function, refer to section 1.3.2, Installation in HI2000/3 and HI1000/4.

3.4.6 FAQs about System Configuration

This section answers questions about system configuration which are frequently asked by users of the HI series OS.

FAQ Contents:

(1) Stack Size Used for Service Calls

Classification: Configuration					
Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4
Does the necessary size in stack size calculation described in the HI7000/4 Series User's Manual include the stack used for service calls?					

Answer

The necessary size in stack size calculation described in the HI7000/4 Series User's Manual does include the stack used for service calls.

Note that some service calls involve task switching, and some do not. The stack size for a service call which involves task switching is included in the necessary stack size for calculation described in the HI7000/4 Series User's Manual. On the other hand, as the latter type of service calls do not involve task switching, they are executed at a high speed without switching task stacks. This processing is possible because there is no task switching. In this case too, the stack size is included in the necessary stack size for calculation.

(2) Calculation of OS Stack Size

Classification: Configuration					
Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4
When calculating the OS stack size, should the calls of a run-time library from another run-time library be counted as the interrupt nesting level?					
Answer Interrupt nesting does not mean nesting of function calls. Only an interrupt that occurs in the same interrupt processing should be counted as the nesting level.					

(3) Definitions for Separate Linkage

Classification: Configuration

Question	HI7000/4	HI7700/4	HI7750/4		

We are considering using separate linkage for system creation.

Though updating the load modules will change the addresses of the application programs unless they are intentionally managed, do such programs operate correctly even after the addresses have changed? In addition, what should be kept in mind when separate linkage is used for task creation?

Answer

Though the addresses of the application programs will be changed by updating the load modules, the application programs are operated without problems.

The points that should be kept in mind when separate linkage is used for task creation are shown below.

When the application programs are not saved in ROM, objects must be created dynamically by service calls. Therefore, include the service calls needed to dynamically create objects by the configurator.

If service calls to dynamically create objects are not included by the configurator, note the following when separate linkage is used.

(1) The programs must be linked to the kernel side.
(2) While the [Link with Kernel Library] check box is selected in the configurator, handlers cannot be defined.

When separate linkage is used, note the following for task creation.

- When creating tasks that use the static stack by the configurator, always select the [Link with Kernel Library] check box.
- When automatic task ID assignment is specified in the configurator, never select the [Link with Kernel Library] check box.

(Continued on next page)

(Continued from previous page)

Answer

For separate linkage, when each load module is created, one symbol of the other load module must be referenced.

When the kernel environment load module is created, the address of __kernel_cnfgtbl (service call interface information: start address of the C_hibase section) must be defined. This defined address must be the same as the address where the C_hibase section is allocated.

When the kernel load module is created, the address of__kernel_sysmt (kernel environment information: start address of the C_hisysmt section) must be defined. This defined address must be the same as the address where the C_hisysmt section is allocated.

As described above, the OS does not require that the start address of the C_hibase section be the same as that of the C_hisysmt section. Please see the following table for a summary of this information.

Kernel Side	Kernel Environment Side
C_hibase section	Symbol _kernel_cnfgtbl is forcibly defined to the start address of the C_hibase section.
<Actual description>	<Reference>
Service call interface information is allocated.	Issues service calls according to the address of symbol _kernel_cnfgtbl.
Symbol _kernel_sysmt is forcibly defined to the start address of the C_hisysmt section.	C_hisysmt section
<Reference>	<Actual description>
Refers to the kernel information at the address of symbol _kernel_sysmt.	Kernel environment information is allocated.

(4) Calculation of Interrupt Nesting Level

Classification: Configuration

Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4

How many levels are interrupts to be nested in the following case? (The specified interrupt levels are not sequential values).

[Interrupt source level]

- Interrupt_IRQ0: Interrupt level 15
- Interrupt_IRQ1: Interrupt level 14
- Interrupt_IRQ2: Interrupt level 12
- Interrupt_IRQ3: Interrupt level 10
- DMAC DEIO: Interrupt level 10
- CMT: Interrupt level 08

- Kernel interrupt mask level: Interrupt mask level 12

How should the number of the following interrupts be determined? Should it be determined by simply counting the nesting level, or by calculating the difference between the highest interrupt level and mask level and the difference between the lowest interrupt level and mask level?

1. Interrupts higher than the kernel interrupt mask level
2. Interrupts equal to or lower than the kernel interrupt mask level

Answer

It can be determined by simply counting the nesting level. It does not depend on whether the interrupt level settings are sequential values.

See the following for the above example.

- Interrupts higher than the kernel interrupt mask level: 2
- Interrupts equal to or lower than the kernel interrupt mask level: 3

(5) Section Information

Classification: Configuration					
Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4
When arbitrary functions are created and the system is configured, sections P, C, D, and B are not generated. Is it necessary to prepare these sections in addition to the sections defined in advance by the OS?					
Answer The sections of the user-created programs can be freely allocated by the user. User programs do not need to be added to the OS section names. The OS does not provide the function to add them to the OS section names. User programs can be allocated with arbitrary section names.					

Section 4 Device-Dependent Specifications

4.1 FAQs about Device-Dependent Specifications

This section answers questions frequently asked by users of the HI series OS about device-dependent specifications.

FAQ Contents:

4.1.1 Cache Enabling Setting

Classification: Device-dependent specifications

Question		HI7700/4	HI7750/4		

What settings are needed to enable the cache?

Answer

The cache should be enabled (initialized) in the CPU initialization process.

The OS provides a service call specialized for cache initialization (vini_cac service call), which must be added to the CPU initialization processing.

A coding example using the HI7700/4 CPU initialization routine for the SH7708 is shown below.

```
/*****************************************************************************/
/*  NAME        = hi_cpuini                                                  */
/*  FUNCTION    = CPU initialize routine                                     */
/*****************************************************************************/
#pragma noregsave(hi_cpuini)

void    hi_cpuini(void)
{

/*** Initialize Hardware Environment ***/
    set_gbr((VP)IOBASE);                    /* set I/O base address to GBR      */
    vini_cac(9, 128, 4);                    /* CACHE disable                    */   (1)

/*** Initialize Software Environment ***/

/*  _INITSCT();    */                       /* Call section-initialize routine  */

    vsta_knl();                             /* Start kernel                     */
}
```

Figure 4.1 CPU Initialization Routine When Using Cache (SH7708)

The following table shows examples of the vini_cac service call specification ((1) shown in the above figure) for each CPU type.

(Continued on next page)

(Continued from previous page)

Answer

Table 4.1 vini_cac Specification Example for Each CPU

CPU Type		Description
SH7708 series	vini_cac (9, 128, 4);	Internal RAM mode is not used, writing mode for P0, U0, and P3 areas is copy-back mode, number of entries is 128, and number of ways is 4
	vini_cac (0x2E, 128, 2);	Internal RAM mode is used, writing mode for P0, U0, and P3 areas is write-through mode, number of entries is 128, and number of ways is 2
SH7709	vini_cac (0xF, 128, 4);	Internal RAM mode is not used, writing mode for P0, U0, and P3 areas is write-through mode, number of entries is 128, and number of ways is 4
SH7706, SH7709S, SH7727, SH7641, SH7660	vini_cac (0xB, 256, 4);	Writing mode for P0, U0, and P3 areas is write-through mode, writing mode for P1 area is copy-back mode, number of entries is 256, and number of ways is 4
SH7290, SH7294, SH7300, SH7705, SH7710	vini_cac (0xF, 512, 4);	Writing mode for P0, U0, and P3 areas is write-through mode, writing mode for P1 area is copy-back mode, number of entries is 512, and number of ways is 4
	vini_cac (0xF, 256, 4);	Writing mode for P0, U0, and P3 areas is write-through mode, writing mode for P1 area is copy-back mode, number of entries is 256, and number of ways is 4

For notes on using cache, refer to the HI7000/4 Series User's Manual.

4.1.2 Cache Usage

Classification: Device-dependent specifications

Question		HI7700/4	HI7750/4		

What should be kept in mind when using the cache?

Answer

- Separating the areas in which data will be cached or not cached

  To separate the areas in which data will be cached or not cached, allocate programs and data at linkage as follows.

  — Programs and data that should be cached: P0, P1, or P3 area

  — Programs and data that should not be cached: P2 area

  Note that data in the P2 area will not be cached even when the cache is enabled.

  To enable or disable the cache dynamically, use the following examples of procedures for HI7700/4 and HI7750/4.

- HI7700/4

  (1) To disable the cache

```
/* Setting SR.BL = 1 is recommended for interrupt masking.                    */

  old_sr = get_cr();
  set_cr(old_sr|0x10000000);      /* Set BL = 1.                              */
  vini_cac(0, entnum, waynum);  /* Disable the cache and clear the CF bit to 0 .  */
  vfls_cac(0, 0x1bffff);       /* Write the necessary area back to the actual memory. */
  /*  At this point, all cache entries can be safely destroyed.               */

  vini_cac(8, entnum, waynum);  /* Disable the cache and set the CF bit to 1.  */
  /*  This step invalidates all cache entries.                                */

  set_cr(old_sr);
```

Figure 4.2 Coding Example for Disabling Cache (HI7700/4)

(Continued on next page)

(Continued from previous page)

Answer

(2) To enable the cache

```
vini_cac(9, entnum, waynum);  /* Set CE = 1 and CF = 1   */
/* CF = 0 is allowable, but CF = 1 is safer.              */
```

Figure 4.3 Coding Example for Enabling Cache (HI7700/4)

— Note on vfls_cac

The address specified by vfls_cac must be in a physical address range from H'0 to H'1BFFFFFF (the upper three bits of the address must be 0).

For details, refer to the HI7000/4 Series User's Manual.

— Notes on vini_cac

The entnum and waynum parameters of vini_cac must be specified as follows.

(1) When the 16-kbyte cache is provided in the device such as SH7709S or SH7729

entnum = 256 and waynum = 4

(2) When the 32-kbyte mode is used for a device such as SH7705 or SH7290

After selecting the 32-kbyte mode with the CCR3 register, set entnum = 512 and waynum = 4, and issue a vini_cac service call.

(Continued on next page)

(Continued from previous page)

Answer

- HI7750/4

  (1) To disable the cache

```
/* Setting SR.BL = 1 is recommended for interrupt masking.                    */

 old_sr = get_cr();
 set_cr(old_sr|0x10000000);   /* Set BL = 1.                                 */
 vini_cac(0x00000000);        /* Set ICE = off and OCE = off.                */
 vfls_cac(0x80000000, 0x9bffffff);
                              /* Write the necessary area back to the actual memory. */
 /* At this point, all cache entries can be safely destroyed.                 */

 vini_cac(0x00000808);        /* Set ICE = off, OCE = off, ICI = 1, and OCI = 1. */
 /* This step invalidates all cache entries.                                  */

 set_cr(old_sr);
```

Figure 4.4 Coding Example for Disabling Cache (HI7750/4)

  (2) To enable the cache

```
vini_cac(0x0000090d); /* Set ICI = 1, ICE = 1, OCI = 1, CB = 1, and OCE = 1. */
/* ICI = 0 and OCI = 0 are allowable, but ICI = 1 and OCI = 1 are safer.      */
```

Figure 4.5 Coding Example for Enabling Cache (HI7750/4)

— Note on vfls_cac

The address specified by vfls_cac must be in a physical address range from H'80000000 to H'9BFFFFFF.

For details, refer to the HI7000/4 Series User's Manual.

4.1.3 Restrictions on Write-Back Mode (1)

Classification: Device-dependent specifications

Question		HI7700/4	HI7750/4		

What should be kept in mind when setting the cache to the write-back mode? Is there any restriction on cache settings for the HI7000/4?

Answer

There is nothing that needs special care, except for the coherency.

For example, when writing data through the program and then transferring the data through the DMAC, use either of the following procedures.

(1) Allocate the address where data is to be written through the program to a cache through area (write data by bypassing the cache).
(2) Create a function to write the cache contents back to the memory, call the function after data is written, and then perform DMA transfer.

When transferring data through the DMAC and then reading the data through the program, use either of the following procedures.

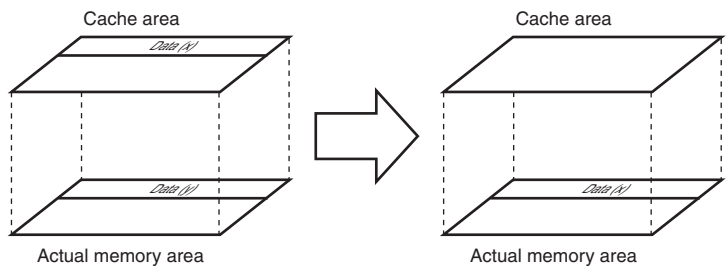
(1) Read data from an address allocated to a cache through area (read data by bypassing the cache).
(2) Create a function to invalidate the cache contents, call the function, and then read the data transferred by the DMAC from an address allocated to a cache through area.

The overview of write-back mode is shown in figure 4.6.

(Continued on next page)

(Continued from previous page)

Answer



Note: When write-back (flushing data from cache) occurs, the cache area data is reflected into the actual memory area.

Figure 4.6 Overview of Write-Back Mode

4.1.4 Restrictions on Write-Back Mode (2)

Classification: Device-dependent specifications

Question			HI7750/4		

When data is transferred through the DMA after the cache is disabled in an acquired variable-length memory block (vinv_cac service call), the data at the beginning of the variable-length memory block becomes invalid. What causes this problem?

Answer

This problem occurs when the memory block contents are stored in the cache. It occurs only in the SH-4 which uses 32-byte cache lines, and does not occur in the SH-3 which uses 16-bytes cache lines.

When variable-length memory blocks are allocated, 16-byte management areas are also allocated in the memory pool as shown in the following figure.

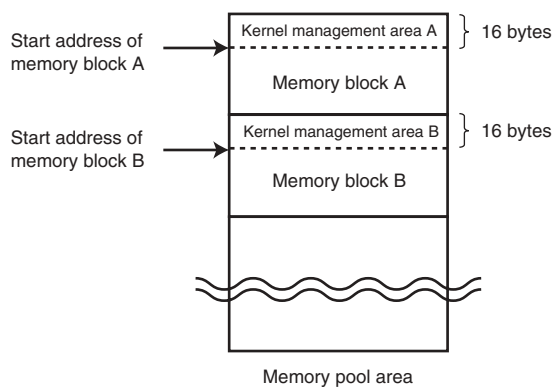


Figure 4.7 Configuration of Variable-Length Memory Blocks

(Continued on next page)

(Continued from previous page)

Answer

If the cache line size is 32 bytes and acquired memory block A is allocated to address 32n + 16 (n is an integer), the first 16 bytes of memory block A is stored in the cache when the kernel accesses management area A. The following shows an example of storing memory block contents in the cache.

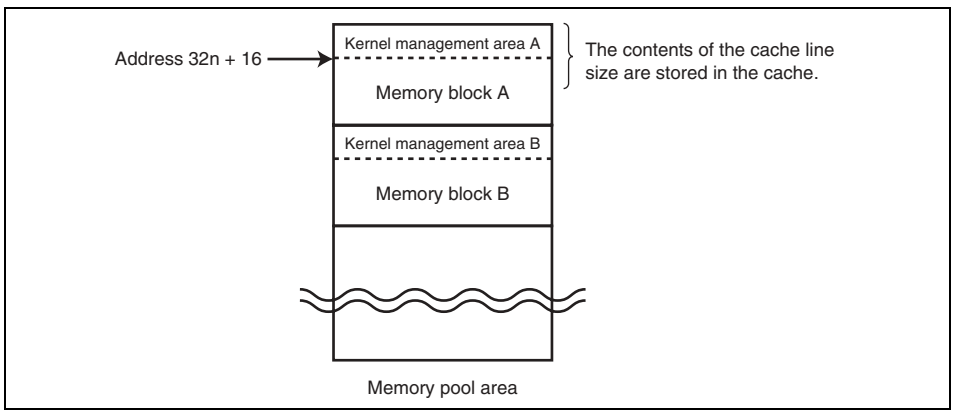


Figure 4.8 Example of Storing Variable-Length Memory Block Contents in Cache

1. When the kernel accesses management area A before DMA transfer, the data before DMA transfer is stored in the cache.
2. When the cache is flushed after DMA transfer, the first 16 bytes of memory block A is overwritten with the cache data and the contents are lost.

To prevent this problem, the start address of the memory block to be acquired must always be set to 32n as shown below.

- Specify the variable-length memory block size to be acquired to (actual required size) + 28.
- When accessing an acquired memory block, round up the start address passed from the kernel to 32n (round up to a higher address) and use the result as the start address of the memory block.

4.1.5 Cache Support

Classification: Device-dependent specifications					
Question		HI7700/4	HI7750/4		
When using cache-support service calls which manipulate the CCR, what should be kept in mind about memory allocation?					
Answer The cache-support service calls access the CCR or address-mapped cache array. During this access, the kernel internally corrects the program counter (PC) value to point to the P2 area (non-cacheable).					

4.1.6 X/Y Memory Usage

Classification: Device-dependent specifications					
Question		HI7700/4			
What should be kept in mind when using the X/Y memory of the SH7729R?					

Answer

The following addresses must be accessed by a program (the section addresses to be specified at linkage).

In P2/Uxy,

- X-RAM: H'A5007000 to H'A5008FFF
- Y-RAM: H'A5017000 to H'A5018FFF

When the following addresses are used,

- X-RAM: H'05007000 to H'05008FFF
- Y-RAM: H'05017000 to H'05018FFF

note the restriction on X/Y memory usage that 2-cycle accesses must always be ensured when the cache is enabled.

4.1.7 Support of MMU

Classification: Device-dependent specifications

Question		HI7700/4	HI7750/4		

Is there any restriction on MMU usage?

Answer

The HI series OS does not assume that the MMU is enabled, but it can be used under the following restrictions.

(1) Allocation of the kernel sections to areas where addresses are not to be translated (P1 or P2)

During kernel processing, some areas are accessed with SR.BL = 1. If a TLB miss occurs while SR.BL = 1, the CPU execution moves to the reset vector. Such areas must be allocated to areas where addresses are not to be translated (P1 or P2). This restriction is applied to the following sections.

P_hiknl, P_hireset,
C_hivct, C_hitrp, C_hibase, C_hisysmt, C_hicfg,
B_hitrcbuf, B_hitrceml, B_hiwrk, B_hidystk, B_histstk,
B_hiirqstk, P_hisysdwn, P_hiintdwn

(2) Address of the service call parameters to be passed through the pointer (such as pk_xxx)

The kernel accesses the parameter address specified by a service call with the SR.BL bit in the same state as when the service call is issued (0). If a TLB miss might occur at this point, no service call should be issued in the TLB miss handler. If it can be ensured that no TLB miss occurs at this point, the TLB miss handler can issue service calls. Note that in the HI7700/4 specifications, no service call should be issued while SR.BL = 1.

(3) Privileged/user mode

In the HI7700/4 specifications, all programs such as tasks and handlers are initiated in the privileged mode. The application program cannot move the state to the user mode.

(4) Write a program to the location of symbol __kernel_tlb_ent in TLB miss handler nnnn_expent.src.

4.1.8 Timer Driver

Classification: Device-dependent specifications

Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4

To obtain a hardware timer cycle of 1 ms when the crystal resonator on the board generates 33.333 MHz, is 33.333 the correct value to use for calculation?

Answer

Use 33.333×10^6 for calculation instead of 33.333.

If 33.333 is used, it will not affect task switching but will affect time management by the OS.

[Reference] Timer driver cycle time calculation
A calculation example of a 1-ms timer cycle time using the SH7604 in the HI7000/4 is shown below.

The hardware timer cycle time (T) is determined by the counter clock cycle time (t) and counter value (n) as follows:

$$T = \{t \times (n + 1)\}$$

t is determined by the counter clock (ϕ/8, ϕ/32, or ϕ/128) selected by the timer control register (TCR).
When ϕ (CPU clock) is 28.6364 MHz, t becomes as follows:

- Counter clock = ϕ/8: t = 279 ns
- Counter clock = ϕ/32: t = 1.11 μs
- Counter clock = ϕ/128: t = 4.46 μs

(Continued on next page)

(Continued from previous page)

Answer

n is determined by setting a value from 0x0000 to 0xFFFF in output compare match register A (OCRA).
When ϕ (CPU clock) is 28.6364 MHz, T is between the following ranges:

- Counter clock = ϕ/8: t = 279 ns to 18.2 ms
- Counter clock = ϕ/32: t = 1.11 μs to 72.7 ms
- Counter clock = ϕ/128: t = 4.46 μs to 292 ms

Calculation of 1-ms cycle:
Output compare match register A (OCRA) = Timer cycle time (s) × n – 1

In the above formula, timer cycle time (s) = 1×10^{-3} to specify a 1-ms timer cycle time.
If ϕ/8 is selected as the counter clock, when ϕ = 28.6364 MHz, n = $28.6364 \times 10^{6} \div 8$.

Accordingly, output compare match register A (OCRA) becomes as follows:

$$\begin{aligned} \text{Output compare match register A (OCRA)} &= \text{Timer cycle time (s)} \times n - 1 \\ &= (1 \times 10^{-3}) \times (28.6364 \times 10^{6} \div 8) - 1 \\ &= 3578.55\ (0x0DFA) \end{aligned}$$

To obtain a 1-ms timer cycle time (s) with ϕ (CPU clock) set to 28.6364 MHz, the value set to output compare match register A (OCRA) should be 3578.55 (0x0DFA).

4.1.9 Control of Timer Used by OS

Classification: Device-dependent specifications

Question	HI7000/4	HI7700/4	HI7750/4		

How should the timer be controlled?

Answer

How to control the timer is described below with using the SH7751 in the HI7750/4 as an example.

Open the 7751_tmrdef.h file in the supplied SH7751 folder.

Change the “Peripheral clock” value on line 19 to the value used in the actual environment, reconfigure the system, and then check the result.

Only the corresponding file can be used to control the timer in the OS.

```
/*****************************************************************************/
/*          HI7750/4 header file for timer driver                            */
/*          Copyright (c) 2000(2003) Renesas Technology Corp.                */
/*          and Renesas Solutions Corp. All Rights Reserved.                 */
/*          HI7750/4(HS0775ITI41SR) V1.0                                     */
/*****************************************************************************/
/*****************************************************************************/
/*  FILE        = 7751_tmrdef.h ;                                            */
/*  CPU type    = SH7751                                                     */
/*  Module      = TMU                                                        */
/*                INTC                                                       */
/*****************************************************************************/
/*****************************************************************************/
/* TMU, IPR setting data                                                     */
/*  Condition:                                                               */
/*    (1) Peripheral clock:      42MHz                                      */
/*    (2) Timer interrupt level: 13                                         */
/*****************************************************************************/
#define PCLOCK  41666667      /* Peripheral clock (Hz)                       */
```

(1)

Figure 4.9 7751_tmrdef.h File

Change the part of (1) in figure 4.9 to match the operating frequency of the device used.

4.1.10 CPU Initialization Routine Written in C Language

Classification: Device-dependent specifications					
Question				HI2000/3	HI1000/4
How should a CPU initialization routine be written in C?					

Answer

The CPU initialization routine can be written in the C language.

However, a C program accesses the stack (memory). A CPU exception may occur if the stack area is accessed before the necessary settings for stack access are completed. (A CPU exception causes system termination.) Accordingly, the CPU initialization routine must be written in the assembly language until the stack settings are completed.

After the necessary settings for stack access have been completed, a CPU initialization routine written in the C language may be executed.

For the method of changing the provided sample CPU initialization routine (written in the assembly language) to enable execution of a CPU initialization routine written in the C language, refer to section 2, Application Program Creation.

4.1.11 Location of Interrupt Entry/Exit Processing Routine

Classification: Device-dependent specifications

Question	HI7000/4	HI7700/4	HI7750/4		

Which address should the interrupt entry/exit processing routine (P_hiexpent section) be allocated to? (Which address is the initial value when the kernel initializes the VBR?)

Answer

The interrupt entry/exit processing routine (P_hiexpent section) can be allocated to any address; the user can choose the address.

When the kernel initializes the VBR, H'100 should be subtracted from the address where P_expent is located (kernel initialization processing automatically calculates it).

For the contents of the sample, refer to the description of exception processing in the user's manual of the device. The description includes the exception processing vector addresses.

Table 4.2 Interrupt or Exception Entry/Exit Processing

Symbol Name	Allocation Address	Processing Contents
__kernel_exp_ent	P_hiexpent section	VBR + H'100 is the general exception vector address
__kernel_tlb_ent	P_hiexpent section + H'300	VBR + H'400 is the TLB miss exception vector address
__kernel_int_ent	P_hiexpent section + H'500	VBR + H'600 is the interrupt vector address

If a general exception occurs, processing starts from VBR + H'100, so necessary processing must be located at the corresponding address.

Therefore, the above settings are necessary.

4.1.12 Initialization of External Memory

Classification: Device-dependent specifications					
Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4
When the stack area of a task is allocated to the external area, why can the task not be woken up?					

Answer

When using the external RAM area, the I/O ports must be set (initialized).

Before starting the OS, initialize the I/O ports. After a reset, the kernel initialization processing accesses external addresses to initialize the task stack area.

An example using the H8S microcomputer is shown below.

For example, in mode 6, ports A, B, and C work as input ports immediately after a reset. They must be set to address output pins by setting PFCR1 (pin function control register 1) for ports A and B to 1 and DDR (data direction register) for port C to 1.

4.1.13 Transition to Power-Down Mode

Classification: Device-dependent specifications

Question	HI7000/4	HI7700/4	HI7750/4		

Does any problem arise when the software standby mode is entered while the system timer is operating? What should be kept in mind when entering the software standby mode?

Answer

When the software standby mode is entered, the timer device used for the OS system timer stops. Accordingly, the following errors will occur.

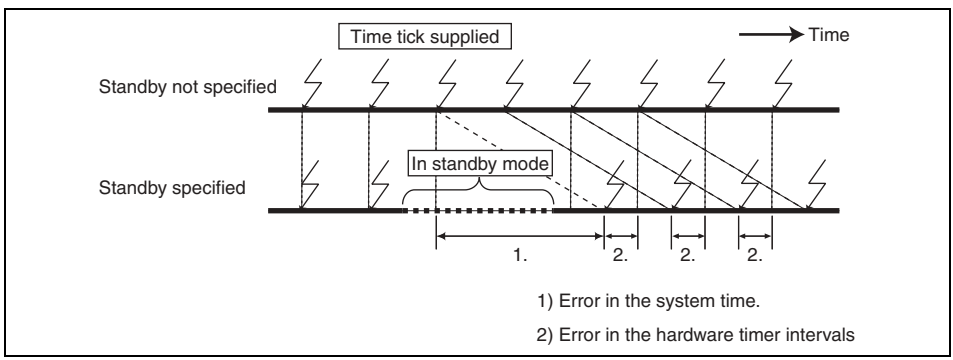


Figure 4.10 Errors in System Time in Standby Mode

(Continued on next page)

(Continued from previous page)

Answer

Note the following when a register of the timer device used for the OS system timer is initialized in software standby mode.

1. Stop the system time in software standby mode and resume it when the software standby mode is canceled.

   For example, if 0.6 ms has passed before the software standby mode is entered since the last timer interrupt, the following processing should be done to generate a timer interrupt 0.4 ms after the software standby mode is cancelled (when the time tick cycle is 1 ms).

   — Save the value of the timer counter, which is a register in the timer device, when the software standby mode is entered.

   — Restore the timer counter to the saved value when the software standby mode is canceled.

2. Stop the system time in software standby mode and initialize the timer counter value when the software standby mode is canceled.

   For example, even if 0.6 ms has passed before the software standby mode is entered since the last timer interrupt, the following processing should be done to generate a timer interrupt 1 ms after the software standby mode is cancelled (when the time tick cycle is 1 ms).

   — Initialize the timer device registers (call timer initialization routine _kernel_tmrini()) when the software standby mode is canceled.

Section 5 Debugging

5.1 Overview of Debugging

In a system incorporating the HI series OS, the system down routine is initiated when the kernel finds an abnormal state such as an error in an object that was initially defined through the configurator or an undefined interrupt or exception. The system down routine can also be initiated through the application program when necessary.

This section describes how to debug the system using the system down routine and how to analyze the cause of an error when the system down routine is initiated.

When an abnormal state is found in the system, perform the following steps to solve the problem.

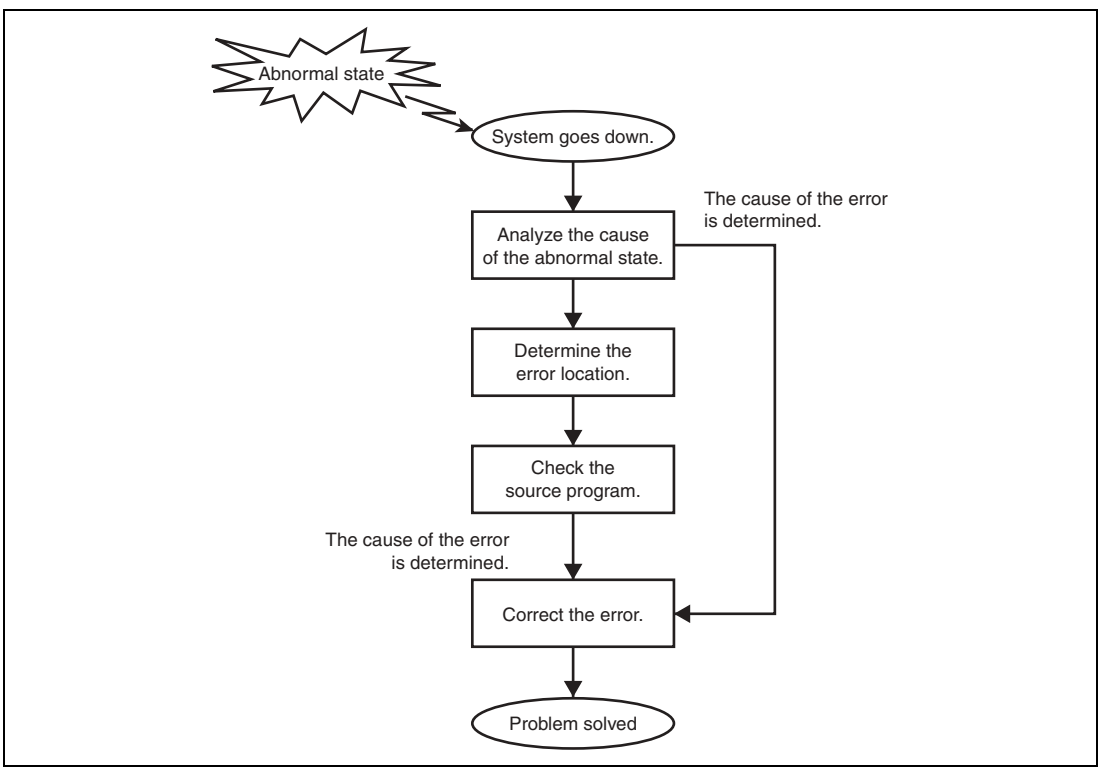


Figure 5.1 Procedure for Debugging Abnormal State in the System

Note: The system down routine is a term used in the HI7000/4 series and HI1000/4. The equivalent routine is called the system termination routine in the HI2000/3. In this section, both are collectively called the system down routine.

5.2 HI7000/4 Series

5.2.1 Preparation for Debugging

(1) Enabling Parameter Check Function

During debugging, the function for checking service call parameters should be enabled. For details on the function, refer to section 1.3, Service Call Parameter Check.

(2) Adding Debugging Code

Add a code for calling the system down routine to the application program so that the system down routine is called if a service call returns a fatal error code, such as a parameter error, and the processing cannot be continued. As this debugging code is unnecessary in the final version of the system, it is efficient to generate the code only when necessary through a macro and compiler's preprocessor directives.

The following shows the interface for calling the system down routine and a coding example.

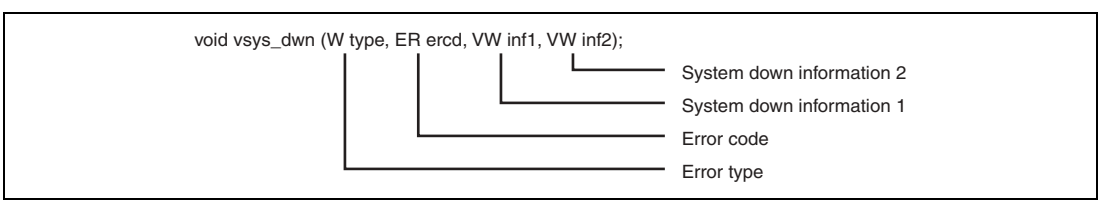


Figure 5.2 System Down Routine Calling Interface (HI7000/4 Series)

```
#define _DEBUG

#ifdef _DEBUG
#define CHK_SYSDWN(cd) if(cd) vsys_dwn((W)1, ercd, (VW)__FILE__, (VW)__LINE__)
#else
#define CHK_SYSDWN(cd)
#endif

ER ercd;

  (Processing omitted)

ercd = set_flg((ID)flgid, (FLGPTN)setptn);          /* Set the event flag */
CHK_SYSDWN(ercd != E_OK);

  (Processing omitted)
```

The error type must be 1 or a larger value when the system down routine is called from the application program. For the other parameters, any values can be selected by the user.

This example generates the debugging code only when the _DEBUG symbol is valid.

Figure 5.3 Debugging Code Example (HI7000/4 Series)

(3) Setting a Breakpoint

Set a breakpoint at the line shown in each example below through an emulator or an ICE and execute the application program.

```
/*****************************************************************/
/*  NAME      = _kernel_sysdwn ;                                 */
/*  FUNCTION  = System down routine ;                            */
/*****************************************************************/
void    _kernel_sysdwn(type, ercd, inf1, inf2)
W type; /* system down type */
            /* type >=  1 : system down of user program          */
            /* type ==  0 : initial information error            */
            /* type == -1 : context error of ext_tsk             */
            /* type == -2 : context error of exd_tsk             */
            /* type == -16: undefined interrupt/exception        */
ER ercd; /* error code  */
            /* type >=  0 : error code of user program           */
            /* type ==  0 : error code of initial information    */
            /* type == -1 : error code of ext_tsk                */
            /* type == -2 : error code of exd_tsk                */
            /* type == -16: interrupt vector number              */
VW inf1; /* information-1  */
            /* type >=  0 : information of user program          */
            /* type ==  0 : indicator of initial information error */
            /* type == -1 : address of ext_tsk call              */
            /* type == -2 : address of exd_tsk call              */
            /* type == -16: address of interrupt occurrence      */
VW inf2; /* information-2  */
            /* type >=  0 : information of user program          */
            /* type ==  0 : number of error initial information  */
            /* type == -16: SR of interrupt occurrence           */
{
  set_imask(SR_IMS15); /* mask all interrupt                     */
  while(TRUE);    /* endless loop                               */
}
```

Set a breakpoint at this line.

Figure 5.4 Example of Setting a Breakpoint (HI7000/4)

```
/*****************************************************************/
/*  NAME     = _kernel_sysdwn ;                                  */
/*  FUNCTION  = System down routine ;                            */
/*****************************************************************/
void    _kernel_sysdwn(type, ercd, inf1, inf2)
W type; /* system down type */
            /* type >=  1 : system down of user program          */
            /* type ==  0 : initial information error            */
            /* type == -1 : context error of ext_tsk             */
            /* type == -2 : context error of exd_tsk             */
            /* type == -16: undefined interrupt/exception        */
ER ercd; /* error code  */
            /* type >=  0 : error code of user program           */
            /* type ==  0 : error code of initial information    */
            /* type == -1 : error code of ext_tsk                */
            /* type == -2 : error code of exd_tsk                */
            /* type == -16: interrupt vector number              */
VW inf1; /* information-1  */
            /* type >=  0 : information of user program          */
            /* type ==  0 : indicator of initial information error */
            /* type == -1 : address of ext_tsk call              */
            /* type == -2 : address of exd_tsk call              */
            /* type == -16: address of interrupt occurrence      */
VW inf2; /* information-2  */
            /* type >=  0 : information of user program          */
            /* type ==  0 : number of error initial information  */
            /* type == -16: SR of interrupt occurrence           */
{
  set_cr(MD_BIT | (SR_IMS15 << 4));  /* mask all interrupt      */
   while(TRUE);    /* endless loop                              */
}
```

Set a breakpoint at this line.

Figure 5.5 Example of Setting a Breakpoint (HI7700/4, HI7750/4)

5.2.2 System Going Down

When the system goes down, the program execution stops at the breakpoint set as described in section 5.2.1 (3), Setting a Breakpoint. In the HI7000/4 series, the error information obtained when the system went down is passed through registers.

The error information parameters are stored in the following format.

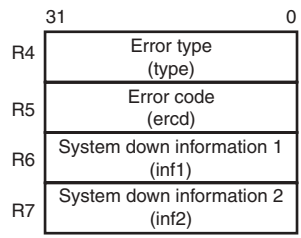


Figure 5.6 System Down Information Parameter Format (HI7000/4 Series)

5.2.3 Types of System Down Causes

The HI7000/4 series system goes down due to the following types of causes.

Table 5.1 Types of System Down Causes (HI7000/4 Series)

No.	Error Type (R4)	Description
1	0	Initially defined object error
2	H'FFFFFFFF (-1)	Context error (ext_tsk service call)
3	H'FFFFFFFE (-2)	Context error (exd_tsk service call)
4	H'FFFFFFF0 (-16)	Undefined interrupt or exception
5	1 or larger (selectable by the user) *1	vsys_dwn or ivsys_dwn service call

Note: *1 The error type value depends on the value specified by the application program.

The error information for each error cause is described below.

(1) Initially Defined Object Error

This error is found in the information about an object initially defined by the configurator. The following values are returned as error information.

Table 5.2 List of Error Information (Initially Defined Object Error)

Item	Register for Storing Information	Description
Error type (type)	R4	H'0
Error code (ercd)	R5	Code for the generated error
System down information 1 (inf1)	R6	0 (kernel side) or 1 (kernel environment side)
System down information 2 (inf2)	R7	Number for the initially defined object that has generated the error

The error code (ercd) indicates the code for the generated error (service call error code).

For system down information 1 (inf1), 0 is passed when the error occurred during object definition in the kernel side, or 1 when the error occurred during object definition in the kernel environment side. For the difference between the kernel side and the kernel environment side, see the following table.

Table 5.3 Difference between Kernel Side and Kernel Environment Side

Item	Description
Kernel side	An object which is included in the kernel load module and for which the "Link with Kernel Library" check box has been selected in the object generating dialog box of the configurator.
Kernel environment side	An object which is included in the kernel environment load module and for which the "Link with Kernel Library" check box has not been selected in the object generating dialog box of the configurator.

System down information 2 (inf2) indicates the ordinal number of the error object in definition processing. Note that the kernel side is processed first and the kernel environment side is then processed.

The following shows examples of values for system down information 1 and 2.

Initial Definitions in the Kernel Side	Initial Definitions in the Kernel Environment Side
• Task A • Cyclic handler A • Extended service call A	• Task B • Task C • Semaphore A • Event flag A

1) When an error occurred during initial definition of task A ..... inf1 = 0, inf2 = 1
2) When an error occurred during initial definition of cyclic handler A ..... inf1 = 0, inf2 = 2
3) When an error occurred during initial definition of extended service call A .... inf1 = 0, inf2 = 3
4) When an error occurred during initial definition of task B ..... inf1 = 1, inf2 = 1
5) When an error occurred during initial definition of task C ..... inf1 = 1, inf2 = 2
6) When an error occurred during initial definition of semaphore A ..... inf1 = 1, inf2 = 3
7) When an error occurred during initial definition of event flag A ..... inf1 = 1, inf2 = 4

Figure 5.7 Examples of System Down Information 1 and 2

Check the definitions using the configurator according to the ordinal number of the error object.

For details on processing for each object, refer to the HI7000/4 Series User's Manual.

(2) Context Error (ext_tsk Service Call)

This error occurs when a non-task context issues an ext_tsk service call. The following values are passed as the error information.

Table 5.4 List of Error Information (Context Error)

Item	Register for Storing Information	Description
Error type (type)	R4	H'FFFFFFFF (-1)
Error code (ercd)	R5	H'FFFFFFE7 (-25)
System down information 1 (inf1)	R6	Address where ext_tsk was called
System down information 2 (inf2)	R7	Undetermined

Check the application program line corresponding to the address passed as system down information 1, and correct the program so that the ext_tsk service call is issued from a task context.

For how to determine the program module corresponding to the error address, refer to section 5.5, Determining System Down Location.

(3) Context Error (exd_tsk Service Call)

This error occurs when a non-task context issues an exd_tsk service call. The following values are passed as the error information.

Table 5.5 List of Error Information (Context Error)

Item	Register for Storing Information	Description
Error type (type)	R4	H'FFFFFFFE (-2)
Error code (ercd)	R5	H'FFFFFFE7 (-25)
System down information 1 (inf1)	R6	Address where exd_tsk was called
System down information 2 (inf2)	R7	Undetermined

Check the application program line corresponding to the address passed as system down information 1, and correct the program so that the exd_tsk service call is issued from a task context.

For how to determine the program module corresponding to the error address, refer to section 5.5, Determining System Down Location.

(4) Undefined Interrupt or Exception

This error occurs when an undefined interrupt or undefined general exception is generated. The following values are passed as the error information.

Table 5.6 List of Error Information (Undefined Interrupt or Exception)

Item	Register for Storing Information	Description	
		HI7000/4	HI7700/4, HI7750/4
Error type (type)	R4	H'FFFFFFF0 (-16)	
Error code (ercd)	R5	Vector number	Exception code
System down information 1 (inf1)	R6	PC information when the interrupt or exception occurred[*1*2*3]	
System down information 2 (inf2)	R7	SR information when the interrupt or exception occurred[*3]	

Note: *1 For a slot illegal instruction exception, the address of the undefined code or delayed branch instruction placed in a delay slot is passed as the PC information (or the address of the next instruction is passed only for the HI7000/4).

*2 For a trap instruction exception, the address of the next instruction after the TRAPA instruction is passed.

*3 For a CPU address error or DMAC address error in the HI7000/4, if the stack pointer (SP) value is not a multiple of four, undetermined values are passed as the PC and SR information.

The error code (ercd) indicates the vector number of the generated undefined interrupt or exception in the HI7000/4, or the generated exception code in the HI7700/4 or HI7750/4. Determine the generated interrupt or exception according to the error code (ercd). For details on the vector number or exception code, refer to the hardware manual of the target microcomputer.

(a) When an Undefined Interrupt Occurred

If the generated interrupt is necessary, create and register an interrupt handler for it. If it is not an intended interrupt, determine the cause, and correct the program so that the interrupt will not occur.

An unintended interrupt may occur due to the following reasons.

- A register is set up incorrectly in the interrupt source (an external device or an on-chip peripheral module in the microcomputer).
- The IRQ or IRL mode is set up incorrectly in the interrupt controller.
- The interrupt priority is set up incorrectly in the interrupt controller and an incorrect-level interrupt is detected.
- A noise is misinterpreted as an interrupt request signal.
- A failure or incorrect setting in the hardware circuit.

(b) When an Undefined General Exception Occurred

If the generated exception is necessary, create and register a CPU exception handler or a trap exception handler for it. If it is not an intended exception, determine the error location according to the PC value passed as system down information 1 (inf1), and analyze the cause.

According to the SR value passed as system down information 2 (inf2), the CPU operating mode or interrupt mask level when the exception occurred can be determined.

For how to determine the program module corresponding to the PC address passed as system down information 1 (inf1), refer to section 5.5, Determining System Down Location.

For how to check the cause of an undefined exception , refer to section 5.6, Examples and Solutions of CPU Exception.

(5) vsys_dwn or ivsys_dwn Service Call

This error occurs when the application program issues a vsys_dwn or ivsys_dwn service call. The passed error information indicates the parameters for the issued vsys_dwn or ivsys_dwn service call.

The debugging code shown in section 5.2.1 (2), Adding Debugging Code, passes the following values as error information.

Table 5.7 List of Error Information (vsys_dwn or ivsys_dwn Service Call)

Item	Register for Storing Information	Description
Error type (type)	R4	1
Error code (ercd)	R5	Error code for the issued service call
System down information 1 (inf1)	R6	Address of the path to the source program file where the error occurred
System down information 2 (inf2)	R7	Line number of the source program where the error occurred

Determine the error cause according to the error information, and correct the application program.

For the error code for the service call, refer to the HI7000/4 Series User's Manual.

5.3 HI2000/3

5.3.1 Preparation for Debugging

(1) Enabling Parameter Check Function

During debugging, the function for checking service call parameters should be enabled. For details on the function, refer to section 1.3, Service Call Parameter Check.

(2) Adding Debugging Code

Add a code for calling the system down routine to the application program so that the system down routine is called if a service call returns a fatal error code, such as a parameter error, and the processing cannot be continued. As this debugging code is unnecessary in the final version of the system, it is efficient to generate the code only when necessary through a macro and compiler's preprocessor directives.

The following shows the interface for calling the system down routine and a coding example.

```
void HIPRG_ABNOML(void);
```

Figure 5.8 Example of System Down Routine Calling Interface (HI2000/3)

```
extern void HIPRG_ABNOML(void);

#define _DEBUG

#ifdef _DEBUG
#define CHK_SYSDWN(cd) if(cd) HIPRG_ABNOML()
#else
#define CHK_SYSDWN(cd)
#endif

ER ercd;

  (Processing omitted)

ercd = set_flg((ID)flgid, (UINT)setptn);          /* Set the event flag */
CHK_SYSDWN(ercd != E_OK);

  (Processing omitted)
```

This example generates the debugging code only when the _DEBUG symbol is valid.

Figure 5.9 Debugging Code Example (HI2000/3)

(3) Setting a Breakpoint

Set a breakpoint at the line shown in each example below through an emulator or an ICE and execute the application program.

```
;*******************************************************************
;*specifications ;                                                  *
;*name      = _HIPRG_ABNOML : abnormal quit handler          ;     *
;*function  =                                                ;     *
;*notes     =                                                ;     *
;*date      = 99/02/22                                       ;     *
;*author    = Hitachi, Ltd.                                  ;     *
;*attribute = public                                         ;     *
;*class     = system                                         ;     *
;*linkage   =                                                ;     *
;*input     =                                                ;     *
;*output    =                                                ;     *
;*end of specifications ;                                           *
;*******************************************************************
_HIPRG_ABNOML:
    orc     #HIDEF_IMASK_CCR:8,ccr   ;: interrupt mask for CCR register
    orc     #HIDEF_IMASK_EXR:8,exr   ;: interrupt mask for EXR register
    bra     $                        ;: forever loop
;
```

Set a breakpoint at this line.

Figure 5.10 Example of Setting a Breakpoint (HI2000/3)

5.3.2 System Going Down

When the system goes down, the program execution stops at the breakpoint set as described in section 5.3.1 (3), Setting a Breakpoint. In the HI2000/3, the error information obtained when the system went down is passed through the stack.

The error information parameters are stored in the following format.

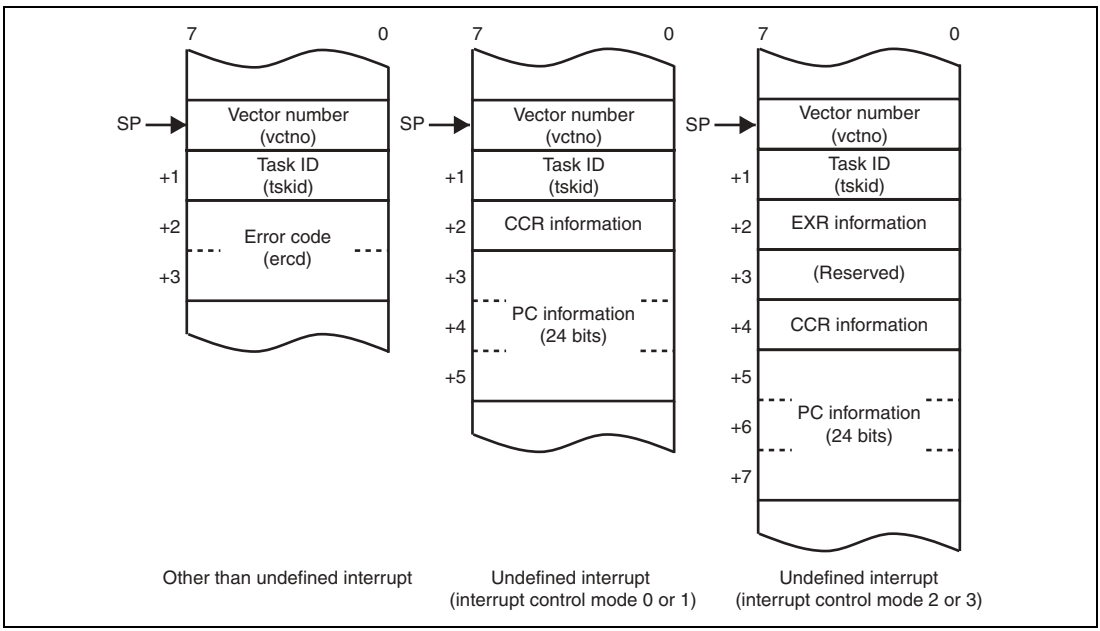


Figure 5.11 System Down Information Parameter Format (HI2000/3)

5.3.3 Types of System Down Causes

The HI2000/3 system goes down due to the following types of causes.

Table 5.8 Types of System Down Causes (HI2000/3)

	Error Type		
No.	**Vector Number (SP + 0)**	**Error Code (SP + 2, SP + 3)**	**Description**
1	0	H'0 to H'0FFF	Setup information error
2		H'F9ED	Unsupported timer
3		H'FFEB	Context error (ext_tsk service call)
4		H'FFBB	Context error (ret_int service call)
5	0 or larger	—	Undefined interrupt
6	—*1	—*1	Call from the application program

Note: *1 The error type value depends on the value specified by the application program. For details, refer to section 5.3.3 (6), Call from the Application Program.

The error information for each error cause is described below.

(1) Setup Information Error

This error is found in the setup table. The following values are passed as the error information.

Table 5.9 List of Error Information (Setup Information Error)

Item	Stack for Storing Information	Description
Vector number (vecno)	SP + 0	0
Task ID (tskid)	SP + 1	0
Error code (ercd)	SP + 2 SP + 3	Setup information error code (H'0 to H'0FFF)

The error code (ercd) indicates the code (H'0000 to H'0FFF) for the invalid setting in the setup table. Check the setup table setting corresponding to the error code. For details on the error code, refer to the HI2000/3 User's Manual.

(2) Unsupported Timer

This error occurs when an attempt is made to use the timeout function while the timeout function is disabled in the setup table. The following values are passed as the error information.

Table 5.10 List of Error Information (Unsupported Timer)

Item	Stack for Storing Information	Description
Vector number (vecno)	SP + 0	0
Task ID (tskid)	SP + 1	0
Error code (ercd)	SP + 2 SP + 3	H'F9ED

Specify "USE" for the timeout function in the setup table or correct the application program so that the timeout function is not specified for service calls.

(3) Context Error (ext_tsk Service Call)

This error occurs when a non-task context issues an ext_tsk service call. The following values are passed as the error information.

Table 5.11 List of Error Information (Context Error)

Item	Stack for Storing Information	Description
Vector number (vecno)	SP + 0	0
Task ID (tskid)	SP + 1	0
Error code (ercd)	SP + 2 SP + 3	H'FFEB

Check the application program line where ext_tsk is used, and correct the program so that the ext_tsk service call is always issued from a task context.

(4) Context Error (ret_int Service Call)

This error occurs when a ret_int service call is issued in task execution state or CPU-locked state. The following values are passed as the error information.

Table 5.12 List of Error Information (Context Error)

Item	Stack for Storing Information	Description
Vector number (vecno)	SP + 0	0
Task ID (tskid)	SP + 1	0
Error code (ercd)	SP + 2 SP + 3	H'FFBB

Check the application program line where ret_int is used, and correct the program so that the ret_int service call is always issued from an interrupt handler.

(5) Undefined Interrupt

This error occurs when an undefined interrupt is generated. The following values are passed as the error information.

Table 5.13 List of Error Information (Undefined Interrupt)

	Stack for Storing Information		
Item	Interrupt Mode 0 or 1	Interrupt Mode 2 or 3	Description
Vector number (vecno)	SP + 0	SP + 0	Vector number
Task ID (tskid)	SP + 1	SP + 1	Task ID or 0
EXR	—	SP + 2	EXR information when the interrupt occurred
CCR	SP + 2	SP + 4	CCR information when the interrupt occurred
PC	(SP + 3)*1 SP + 4 SP + 5	(SP + 5)*1 SP + 6 SP + 7	PC information when the interrupt occurred

Note: *1 This value is only valid in advanced mode; it has no means in normal mode (only the lower 16 bits of the PC are valid).

The vector number (vecno) indicates the vector number for the generated interrupt. Determine the generated interrupt according to the vector number. For details on the vector number, refer to the hardware manual of the target microcomputer.

If the generated interrupt is necessary, create and register an interrupt handler for it. If it is not an intended interrupt, determine the cause, and correct the program so that the interrupt will not occur.

An unintended interrupt may occur due to the following reasons.

- A register is set up incorrectly in the interrupt source (an external device or an on-chip peripheral module in the microcomputer).
- The IRQ or IRL mode is set up incorrectly in the interrupt controller.
- The interrupt priority is set up incorrectly in the interrupt controller and an incorrect-level interrupt is detected.
- A noise is misinterpreted as an interrupt request signal.
- A failure or incorrect setting in the hardware circuit.

The EXR and CCR information indicates the interrupt mask level when the interrupt occurred.

If an undefined interrupt occurred in a task context, the task ID (tskid) indicates the ID of the task being executed when the interrupt occurred.

For how to determine the program module corresponding to the PC information, refer to section 5.5, Determining System Down Location.

(6) Call from the Application Program

When the system down routine (_HIPRG_ABNOML) provided as a sample is called from an application program written in the C language, the return address is stored in the stack and error information cannot be passed through the stack.

When calling the sample system down routine from the application program, the user must analyze the cause of the error.

To pass error information through the stack in the same way as when other system down causes are generated, modify the system down routine and change the symbol name called from the application program (any name can be selected by the user; _HIPRG_ABNOML_CSUB in the following example) as shown below.

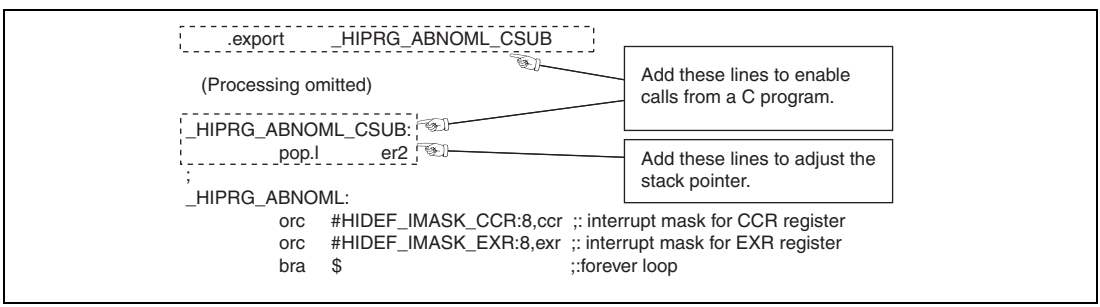


Figure 5.12 Example of System Down Routine Modification (HI2000/3)

The following shows an example of debugging code for the system down routine modified as shown above.

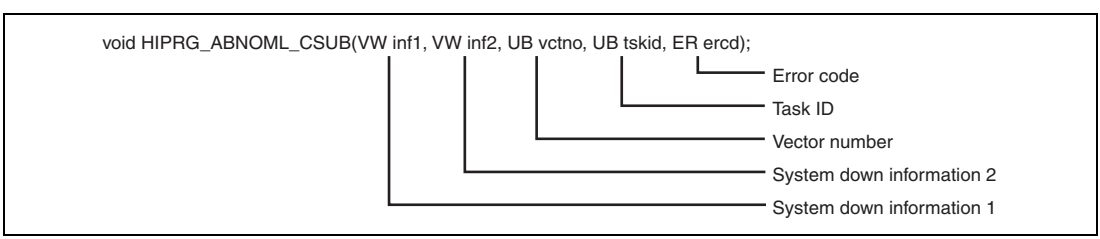


Figure 5.13 Example of System Down Routine Calling Interface (HI2000/3)

```
extern void __regparam2 HIPRG_ABNOML_CSUB(VW inf1, VW inf2, UB vctno, UB tskid, ER ercd);

#define _DEBUG
#ifdef _DEBUG
#define CHK_SYSDWN(cd) if(cd) HIPRG_ABNOML_CSUB(__FILE__, __LINE__, 255, 255, ercd)
#else
#define CHK_SYSDWN(cd)
#endif

ER ercd;

   (Processing omitted)

ercd = set_flg((ID)flgid, (FLGPTN)setptn);        /* Set the event flag */
CHK_SYSDWN(ercd != E_OK);

   (Processing omitted)
```

vctno and tskid must be 255 when the system down routine is called from the application program. For the other parameters, any values can be selected by the user.

This example generates the debugging code only when the _DEBUG symbol is valid.

Figure 5.14 Debugging Code Example (HI2000/3)

When the system down routine is called from the application program after the above debugging code is added, the following values are passed as the error information.

Table 5.14 List of Error Information (Call from the Application Program)

Item	Stack or Register for Storing Information	Description
Vector number (vecno)	SP + 0	H'FF (255)
Task ID (tskid)	SP + 1	H'FF (255)
Error code (ercd)	SP + 2 SP + 3	Error code for the issued service call
System down information 1	ER0	Address of the path to the source program file where the error occurred
System down information 2	ER1	Line number of the source program where the error occurred

When the system goes down due to a call from the application program, determine the error cause according to the error information, and correct the application program.

For the error code for the service call, refer to the HI2000/3 User's Manual.

5.4 HI1000/4

5.4.1 Preparation for Debugging

(1) Enabling Parameter Check Function

During debugging, the function for checking service call parameters should be enabled. For details on the function, refer to section 1.3, Service Call Parameter Check.

(2) Adding Debugging Code

Add a code for calling the system down routine to the application program so that the system down routine is called if a service call returns a fatal error code, such as a parameter error, and the processing cannot be continued. As this debugging code is unnecessary in the final version of the system, it is efficient to generate the code only when necessary through a macro and compiler's preprocessor directives.

The following shows the interface for calling the system down routine and a coding example.

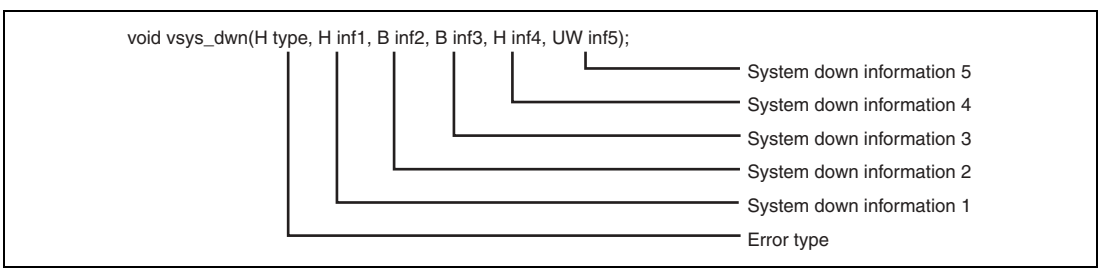


Figure 5.15 Example of System Down Routine Calling Interface (HI1000/4)

```
extern void __regparam3 vsys_dwn(H type, H inf1, B inf2, B inf3, H inf4, UW inf5);

#define _DEBUG

#ifdef _DEBUG
#define CHK_SYSDWN(cd) if(cd) vsys_dwn((H)1, ercd, 0, 0, __LINE__, __FILE__)
#else
#define CHK_SYSDWN(cd)
#endif

ER ercd;

   (Processing omitted)

ercd = set_flg((ID)flgid, (FLGPTN)setptn);          /* Set the event flag */
CHK_SYSDWN(ercd != E_OK);

   (Processing omitted)
```

The error type must be 1 or a larger value when the system down routine is called from the application program. For the other parameters, any values can be selected by the user.

This example generates the debugging code only when the _DEBUG symbol is valid.

Figure 5.16 Debugging Code Example (HI1000/4)

(3) Setting a Breakpoint

Set a breakpoint at the line shown in each example below through an emulator or an ICE and execute the application program.

```
;*****************************************************************;
;* NAME = vsys_dwn                                               *;
;* FILE   = vsys_dwn.src                                         *;
;* FUNC = System down routine                                    *;
;* NOTE =                                                        *;
;* INPU  = none      :                                           *;
;* OUTP = none       :                                           *;
;*****************************************************************;
;
            .section          P_hisysdwn, code, align = 2
;
            .export           _vsys_dwn
            .export           _ivsys_dwn
_vsys_dwn:
_ivsys_dwn:
        bra     _vsys_dwn:8
        rts
;
        .end; of vsys_dwn.src
```

Set a breakpoint at this line.

Figure 5.17 Example of Setting a Breakpoint (HI1000/4)

5.4.2 System Going Down

When the system goes down, the program execution stops at the breakpoint set as described in section 5.4.1 (3), Setting a Breakpoint. In the HI1000/4, the error information obtained when the system went down is passed through registers.

The error information parameters are stored in the following format.

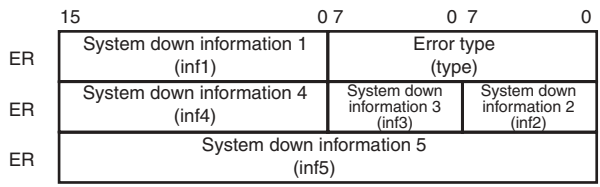


Figure 5.18 System Down Information Parameter Format (HI1000/4)

5.4.3 Types of System Down Causes

The HI1000/4 system goes down due to the following types of causes.

Table 5.15 Types of System Down Causes (HI1000/4)

No.	Error Type (R0)	Description
1	H'FFFB (-5)	Initially defined object error
2	H'FFFD (-3)	Context error 1
3	H'FFFE (-2)	Context error 2
4	H'FFFF (-1)	Undefined interrupt or exception
5	1 or larger (selectable by the user)	vsys_dwn or ivsys_dwn service call

The error information for each error type is described below.

(1) Initially Defined Object Error

This error is found in the information defined by the configurator. The following values are passed as the error information.

Table 5.16 List of Error Information (Initially Defined Object Error)

Item	Register for Storing Information	Description
Error type (type)	R0	H'FFFB
System down information 1 (inf1)	E0	Error number (H'0000 to H'0FFF)
System down information 2 (inf2)	R1L	0
System down information 3 (inf3)	R1H	0
System down information 4 (inf4)	E1	0
System down information 5 (inf5)	ER2	0

System down information 1 (inf1) indicates the error number (H'0000 to H'0FFF) corresponding to the invalid setting in the setup information. Check the setting in the setup information corresponding to the error number using the configurator. For details on the error number, refer to the HI1000/4 User's Manual.

(2) Context Error 1

This error occurs when the kernel finds a context error in a service call (ext_tsk). The following values are passed as the error information.

Table 5.17 List of Error Information (Context Error 1)

Item	Register for Storing Information	Description
Error type (type)	R0	H'FFFD
System down information 1 (inf1)	E0	H'FFE7
System down information 2 (inf2)	R1L	CCR information when the error occurred
System down information 3 (inf3)	R1H	EXR information when the error occurred
System down information 4 (inf4)	E1	0
System down information 5 (inf5)	ER2	PC information when the error occurred

Check the application program line corresponding to the address where the error occurred, and correct the program so that the ext_tsk service call is always issued from a task context.

For how to determine the program module corresponding to the PC value passed through system down information 5 (inf5), refer to section 5.5, Determining System Down Location.

(3) Context Error 2

This error occurs when the kernel finds a context error in a ret_int routine call. The following values are passed as the error information.

Table 5.18 List of Error Information (Context Error 2)

Item	Register for Storing Information	Description
Error type (type)	R0	H'FFFE
System down information 1 (inf1)	E0	0
System down information 2 (inf2)	R1L	Task ID
System down information 3 (inf3)	R1H	0
System down information 4 (inf4)	E1	0
System down information 5 (inf5)	ER2	0

Check the application program line where the ret_int routine is used, and correct the program so that the ret_int routine is always called from an interrupt handler or an exception handler.

(4) Undefined Interrupt or Exception

This error occurs when an undefined interrupt or exception is generated. The following values are passed as the error information.

Table 5.19 List of Error Information (Undefined Interrupt or Exception)

Item	Register for Storing Information	Description
Error type (type)	R0	H'FFFF
System down information 1 (inf1)	E0	Interrupt vector number
System down information 2 (inf2)	R1L	CCR information when the interrupt occurred
System down information 3 (inf3)	R1H	EXR information when the interrupt occurred
System down information 4 (inf4)	E1	Task ID or 0
System down information 5 (inf5)	ER2	PC information when the interrupt occurred

System down information 1 (inf1) indicates the vector number for the generated interrupt or exception. Determine the generated interrupt or exception according to the vector number. For details on the vector number, refer to the hardware manual of the target microcomputer.

(a) When an Undefined Interrupt Occurred

If the generated interrupt is necessary, create and register an interrupt handler for it. If it is not an intended interrupt, determine the cause, and correct the program so that the interrupt will not occur.

An unintended interrupt may occur due to the following reasons.

- A register is set up incorrectly in the interrupt source (an external device or an on-chip peripheral module in the microcomputer).
- The IRQ or IRL mode is set up incorrectly in the interrupt controller.
- The interrupt priority is set up incorrectly in the interrupt controller and an incorrect-level interrupt is detected.
- A noise is misinterpreted as an interrupt request signal.
- A failure or incorrect setting in the hardware circuit.

(b) When an Undefined General Exception Occurred

If the generated exception is necessary, create and register a CPU exception handler or a trap exception handler for it. If it is not an intended exception, determine the error location according to the PC value passed as system down information 5 (inf5), and analyze the cause.

The interrupt mask level can be determined according to system down information 2 (inf2) and system down information 3 (inf3).

If an undefined exception occurred in a task context, system down information 4 (inf4) indicates the ID of the task being executed when the exception occurred.

For how to determine the program module corresponding to the PC information passed as system down information 5 (inf5), refer to section 5.5, Determining System Down Location.

For how to check the cause of an undefined exception, refer to section 5.6, Examples and Solutions of CPU Exception.

(5) vsys_dwn or ivsys_dwn Service Call

This error occurs when the application program issues a vsys_dwn or ivsys_dwn service call. The passed error information indicates the parameters for the issued service call.

The debugging code shown in section 5.4.1 (2), Adding Debugging Code, passes the following values as the error information.

Table 5.20 List of Error Information (vsys_dwn, ivsys_dwn Service Call)

Item	Register for Storing Information	Description
Error type (type)	R0	H'1
System down information 1 (inf1)	E0	Error code for the issued service call
System down information 2 (inf2)	R1L	0
System down information 3 (inf3)	R1H	0
System down information 4 (inf4)	E1	Line number of the source program where the error occurred
System down information 5 (inf5)	ER2	Address of the path to the source program file where the error occurred

Determine the error cause according to the error information, and correct the application program.

For the error code for the service call, refer to the HI1000/4 User's Manual.

5.5 Determining System Down Location

The PC information is passed as system down information. To determine the system down location in a program according to the PC information, use the source-level debugging function of an emulator or an ICE, or check the map file output from the linker to determine the approximate location.

5.5.1 Determining the Location of a Program Module through Mapview

This section describes how to determine the location of a program module according to the PC information using the Mapview, an accessory tool of the C compiler. In the following example, HI7700/4, SuperH™ RISC engine series C/C++ compiler package Ver. 8.0.01, and the SH7641 whole linkage project (7641_mix) as a subproject in the HEW workspace are used.

When the Mapview is used, a map file including symbol information must be output through the linkage editor. Specify output of a map file including the symbol information through the optimizing linkage editor option setting window of the HEW.

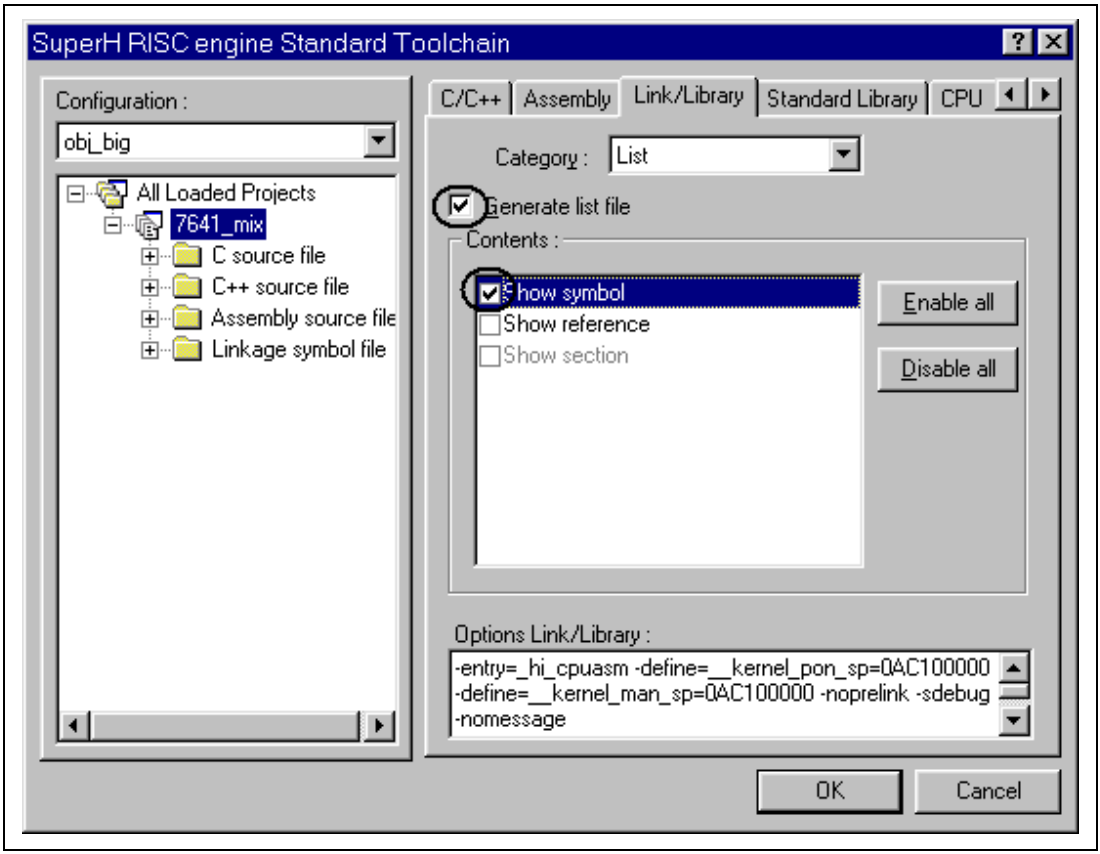


Figure 5.19 List Output Setting for Optimizing Linkage Editor

(1) Initiating the Mapview

Select [Program (P)] -> [Renesas High-performance Embedded Workshop] -> [Mapview] from the Start menu to initiate the Mapview.

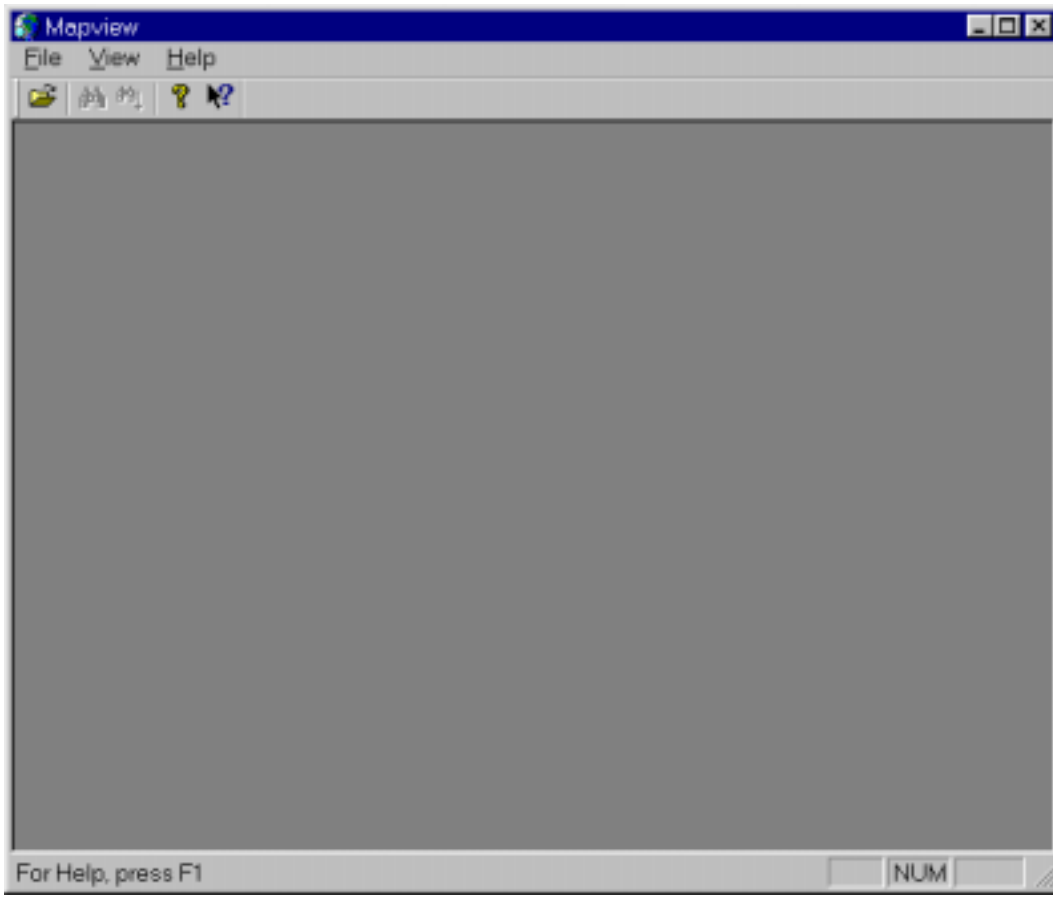


Figure 5.20 Initiated Mapview Window

Select [File] -> [Open...] from the header menu in the initiated window and open the map file output from the linkage editor.

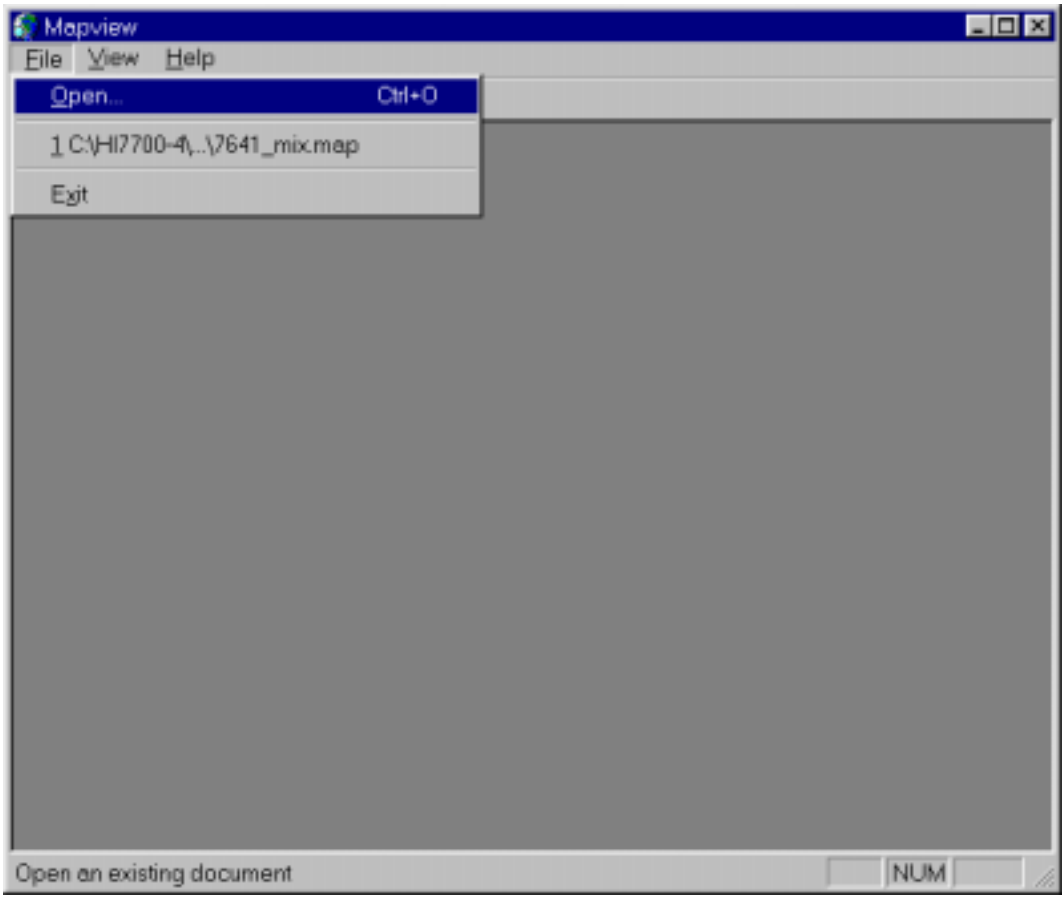


Figure 5.21 Window for Reading a File

(2) Searching for an Address

Clicking a section name displays a list of symbols used in the clicked section in the symbol information view. Check the displayed addresses and sizes and search for the symbol where the PC value is included and determine the system down location in the program.

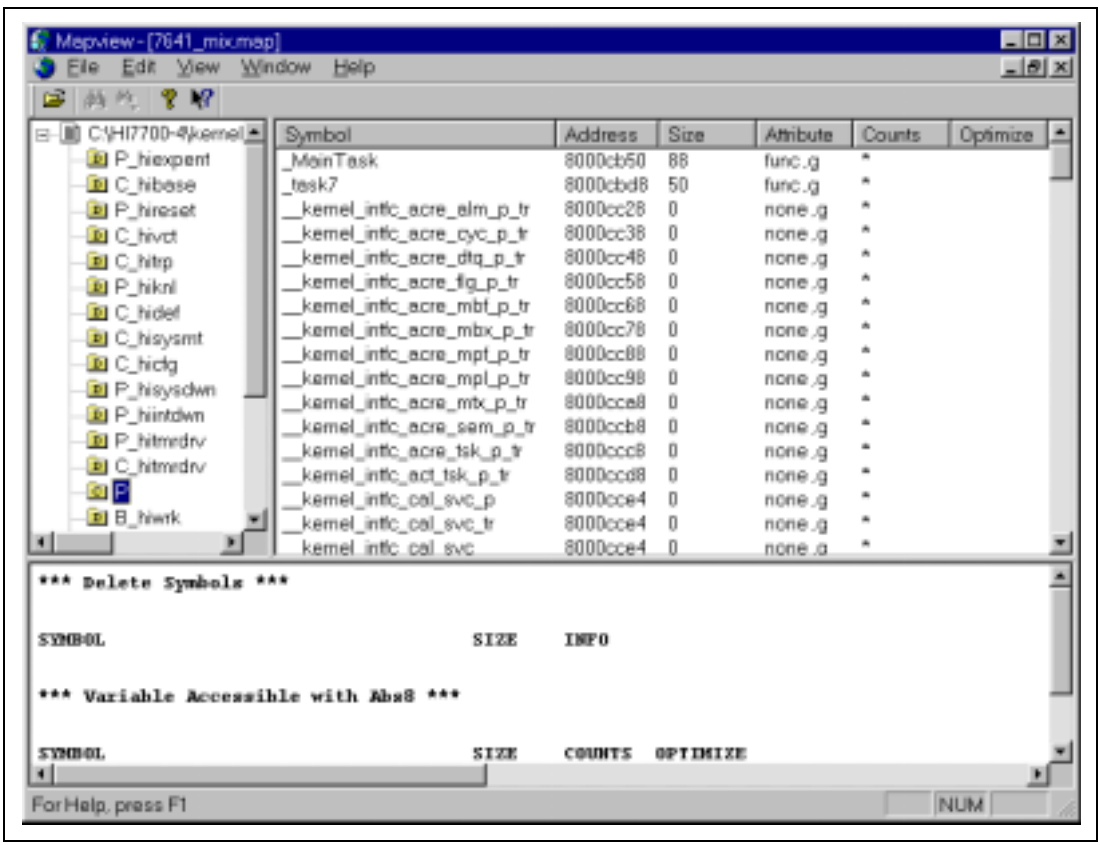


Figure 5.22 Window for Listing Symbols

5.6 Examples and Solutions of CPU Exception

This section describes examples of how the system goes down due to a CPU exception and how the problem should be solved. The following shows the main causes of CPU exceptions. Note that it is assumed in this section that neither the memory management unit (MMU) nor the user break controller (UBC) is used and there is no trap instruction.

Table 5.21 Main Error Causes

Exception Cause	Exception Location	Probable Cause	
General illegal instruction, slot illegal instruction	User program	Direct cause	• Incorrect CPU option setting through the compiler (5.6.2) • Damaged program area (5.6.3) • Failure in hardware (5.6.1)
		Indirect cause	• Stack overflow (5.6.2)
	Kernel	Direct cause	• Damaged program area (5.6.3) • Failure in hardware (5.6.1)
		Indirect cause	• Damaged kernel management area (5.6.3) • Stack overflow (5.6.2)
	Other location, outside the program area	Indirect cause	• Incorrect function call using a pointer variable (5.6.3) • Stack overflow (5.6.2)
CPU address error, DMAC or DTC address error	User program	Direct cause	• Access violation at a data boundary (5.6.3) • Access violation in the physical address space (5.6.3) • Incorrect DMAC or DTC register setting • Failure in hardware (5.6.1)
		Indirect cause	• Incorrect section information setting through the linkage editor (5.6.2) • Stack overflow (5.6.2)
	Kernel	Direct cause	• Failure in hardware (5.6.1)
		Indirect cause	• Damaged kernel management area (5.6.3) • Incorrect section information setting through the linkage editor (5.6.2) • Stack overflow (5.6.2)
	Other location, outside the program area	Indirect cause	• Incorrect function call using a pointer variable (5.6.3) • Stack overflow (5.6.2)

Direct cause: Directly causes the system to go down.

Indirect cause: Causes malfunction of the program, which results in system going down.

Note: The number in parenthesis shows the reference section in this application note.

5.6.1 Failure in Hardware

(1) Failure in Memory Initialization

When using external memory devices (such as SDRAM or SRAM), check that the bus state controller (BSC) is correctly set up and all areas in the memory to be used can be correctly accessed (read and written to). When using emulation memory, check that the emulator is correctly set up.

Hardware must be initialized before the kernel initialization processing is called. For details, refer to section 2.2, Overview of CPU Initialization Routine.

5.6.2 Incorrect Configuration

(1) Incorrect CPU Option Setting through the Compiler

Check that the CPU options (CPU type or endian) set through the compiler matches the target hardware specifications. In particular, check whether the target hardware uses big endian or little endian in the SH-2, SH-3, SH-3DSP, and SH-4 series microcomputers. The following shows the window for specifying the CPU options in the compiler.

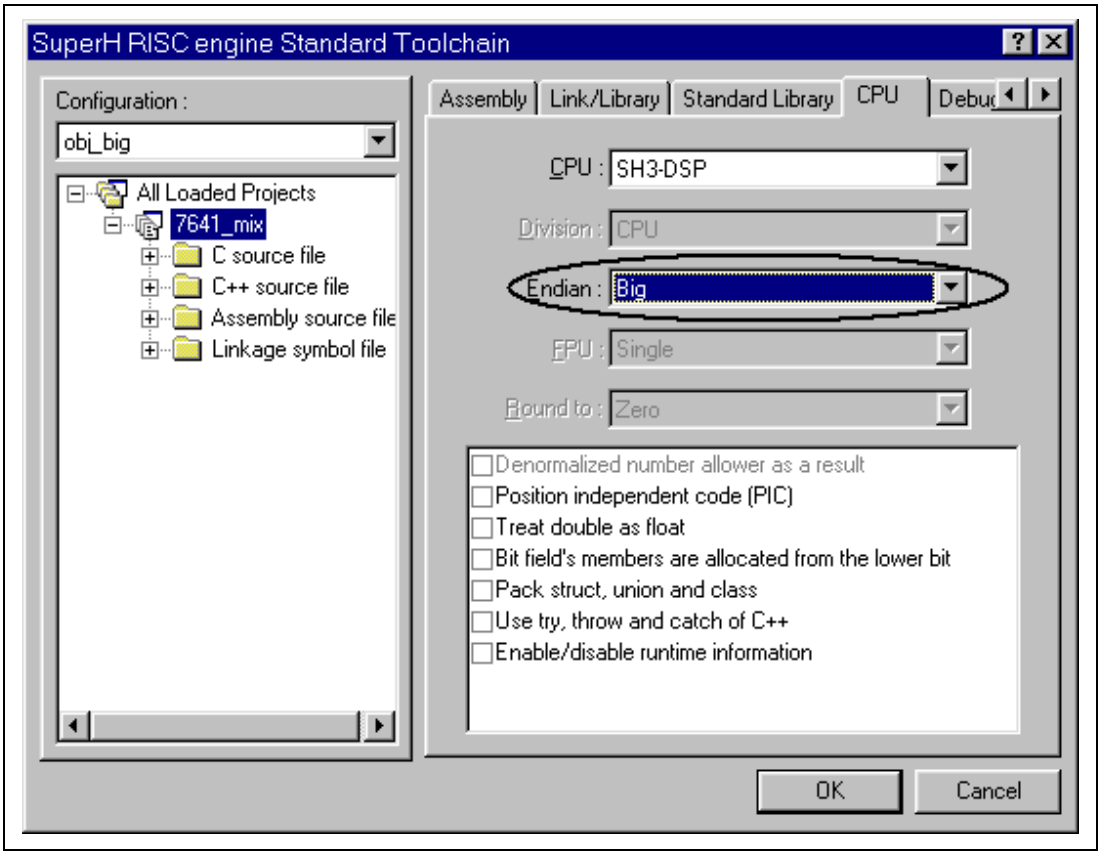


Figure 5.23 Window for Specifying CPU Options

(2) Incorrect Section Information Setting through the Linkage Editor

Check that the work spaces (such as B_hixxxx, B, and R sections) used by the HI series OS and the application program are allocated in the available RAM area and they do not exceed the RAM capacity.

To check that the sections do not exceed the available RAM area, use the map file output from the linkage editor. For output of a map file, refer to the user's manual of the compiler used.

The following shows an example of a map file output from the SuperH™ RISC engine series C/C++ compiler package Ver. 8.0.01.

```
Optimizing Linkage Editor (Ver. 8.0.02.000)      03-Sep-2004 10:35:31

(Processing omitted)

*** Mapping List ***

SECTION                  START      END        SIZE    ALIGN

P_hiexpent
                         80000100   800007df   6e0     4
C_hibase
                         80001000   80001363   364     4
P_hireset
                         80001364   80001530   1cd     4
C_hivct
                         80001534   80001933   400     4
C_hitrp
                         80001934   80002133   800     4
P_hiknl
                         80002134   8000c7a7   a674    4
C_hidef
                         8000c7a8   8000c7ef   48      4
C_hisysmt
                         8000c7f0   8000c9c3   1d4     4
C_hicfg
                         8000c9c4   8000ca2f   6c      4
P_hisysdwn
                         8000ca30   8000ca4f   20      4
P_hiintdwn
                         8000ca50   8000cab3   64      4
P_hitmrdrv
                         8000cab4   8000cb4b   98      4
C_hitmrdrv
                         8000cb4c   8000cb4d   2       4
P
                         8000cb50   8000d6cf   b80     4
B_hiwrk
                         8c000000   8c009ddb   9ddc    4
B_himpl
                         8c009ddc   8c021ddb   18000   4
B_hidystk
                         8c021ddc   8c025ddb   4000    4
B_histstk
                         8c025ddc   8c026ddb   1000    4
B_hiirqstk
                         8c026ddc   8c027fdb   1200    4
B_hitrcbuf
                         8c027fdc   8c037fdb   10000   4
P_hicpuasm
                         a0000000   a000002f   30      4
P_hicpuini
                         a0000030   a0000057   28      4
```

Check that the work spaces of the OS and application program do not exceed the available RAM area.

Figure 5.24 Mapping List in a Map File

(3) Stack Overflow

Check that there is enough stack size for each task, interrupt handler, initialization routine, and time event handler.

For calculation of each stack size, refer to section 3.3, Stack Size Calculation.

For the stack, the specified area is used from the highest address in descending order. If the stack runs out of space, the contents of the lower addresses (nearer to address 0) will be damaged. The following shows an example of an exception caused by a stack overflow.

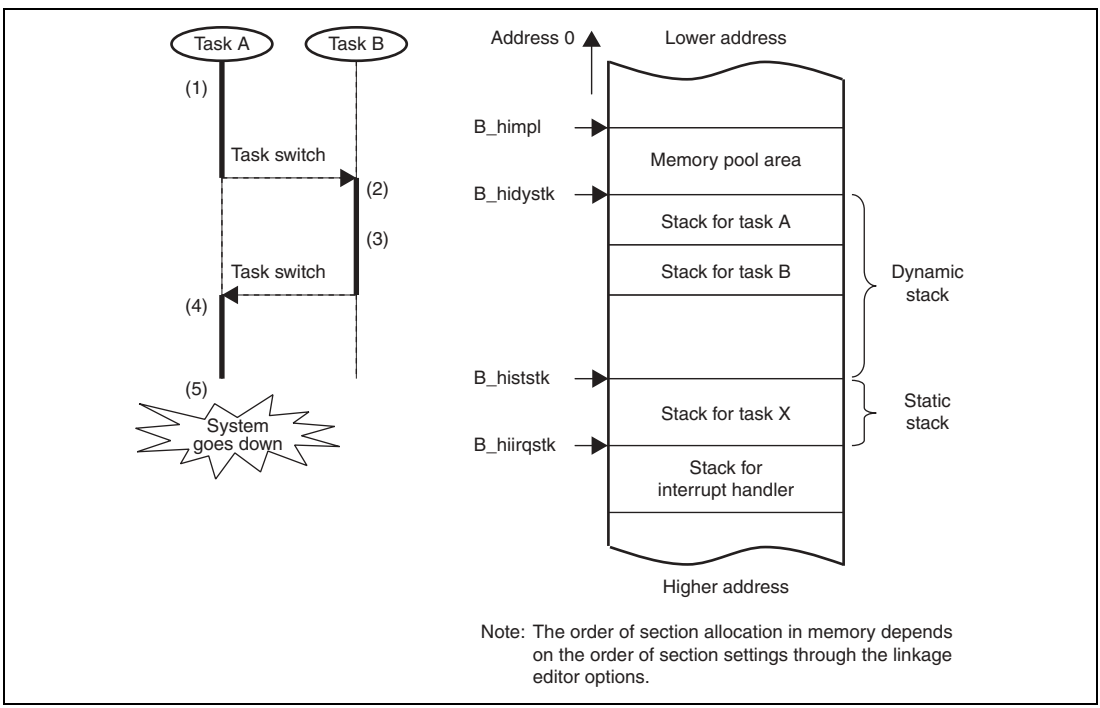


Figure 5.25 Example of Task Operation and Stack Allocation

(1) When task A is executed, the stack for task A is used.

(2) When tasks are switched and task B is executed, the stack for task B is used.

(3) If the stack for task B runs out of space, the stack area for task A, which is allocated to lower addresses, is damaged.

(4) When tasks are switched and task A is resumed, task A uses the contents of the stack. In this case, the stack contents have been overwritten and a malfunction occurs in the program.

(5) The malfunction in the program causes a CPU exception or a hang-up.

The area to be damaged depends on the section allocated to lower addresses than the stack; for example, another program area or a memory pool area may be damaged. Depending on the damaged area, the type of malfunction differs.

5.6.3 Error in Program Description

(1) Damaged Kernel Management Area

Check if the kernel management area is damaged due to an error in program description. When either of the following functions is used, special care must be taken so that the kernel management area is not damaged.

- Mailbox
- Variable-length memory pool

The following shows a bad coding example using a mailbox.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

typedef struct user_msg{
    T_MSG t_msg;
    B     data[10];
} USER_MSG;                                  (1) Declare the user message type.

void Task_sub_sndmsg(VP_INT exinf)
{
    ER ercd;
    USER_MSG message;                        (2) Declare the user message.

      (Processing omitted)

    /* Processing to store the user message */

    message->t_msg.msghead = 0;              (3) Clear the kernel management area in the message to 0.

    ercd = snd_mbx((ID)mbxid, (T_MSG *)&message);   (4) Send the message.
    if(ercd != E_OK){
        /* Error processing */
    }

    return;
}
```

Figure 5.26 Bad Coding Example for Sending a Message

If the priority of the task sending a message is higher than that of the task receiving the message, the area for local variable "message" becomes invalid when execution returns from the

Task_sub_sndmsg function. When the kernel accesses the management area in the message after that, a malfunction occurs in the program and the system goes down.

Write a program so that the contents of the message data area for a mailbox is retained until it is received; for example, allocate the message data area in the memory pool area.

(2) Damaged Program Area

When the program area (including the user program and OS) is allocated in the RAM, it may be overwritten due to an error in user program description or a failure in hardware, and the system may go down.

The following shows an example for determining the cause of the damage in the program area.

(1) Verify the program area contents in the RAM with the loaded executable file to determine whether the program area has been overwritten.
(2) Specify a hardware break so that a break occurs when a write access is made to the overwritten location.
(3) Load the program and reexecute it.
(4) If program execution stops due to a hardware break, it is confirmed that the program area has been overwritten by the program execution. Check the program code where execution stops.
(5) If program execution does not stop due to a hardware break but the same location is damaged, there may be a failure in hardware.

(3) Access Violation at a Data Boundary

When memory is manipulated through pointer variables in the SH-2, SH-3, SH-3DSP, or SH-4 series microcomputer, check if the program contains either of the following descriptions.

- Word data read or write at an address other than a word boundary (address 2n+1)
- Longword data read or write at an address other than a longword boundary (address 4n+1, 4n+2, or 4n+3)

When either of the above program code is executed, the system may go down (a CPU address error). The following shows a bad coding example.

```
#include "itron.h"
#include "kernel.h"
#include "kernel_id.h"

UB buf[16];                    ← (1) Allocate a 16-byte area (buf) for a global variable.

void Task_sub1(void)
{
    UW *ptr;
    int i;

    ptr = (UW *)&buf;          ← (2) Set pointer variable ptr to the start address of buf.

    for(i=0; i < 4; i++) {
        *ptr++ = 0;            ← (3) Clear the contents of buf to 0.
    }

      (Processing omitted)
}
```

Figure 5.27 Bad Coding Example Causing System-Down

If the buf area is allocated to address 4n through the linkage editor, the program is correctly executed. If it is allocated to an odd-valued address or address 2n, the system goes down at location 3) in the above example.

To solve this problem, modify statement 1) in the example to "UW buf[4]", then the buf area is always allocated at a longword boundary and the system-down problem can be avoided.

(4) Access Violation in the Physical Address Space

When memory is manipulated through pointer variables, check if the program contains the following description.

- Access to an unintended area due to an attempt to use an uninitialized global or local variable

When the above program code is executed, the system goes down (a CPU address error).

When the uninitialized data area (B section) for global or static variables should be cleared to 0, the section must be initialized by the CPU initialization routine. For the CPU initialization routine, refer to section 2.2, Overview of CPU Initialization Routine.

Use the information message output from the compiler to check whether uninitialized local variables are used. Note that, in some cases, this cannot be checked through the information message depending on the coding method. In this case, the user must check it through other means.

The following shows the window for specifying information message output when the HEW is used.

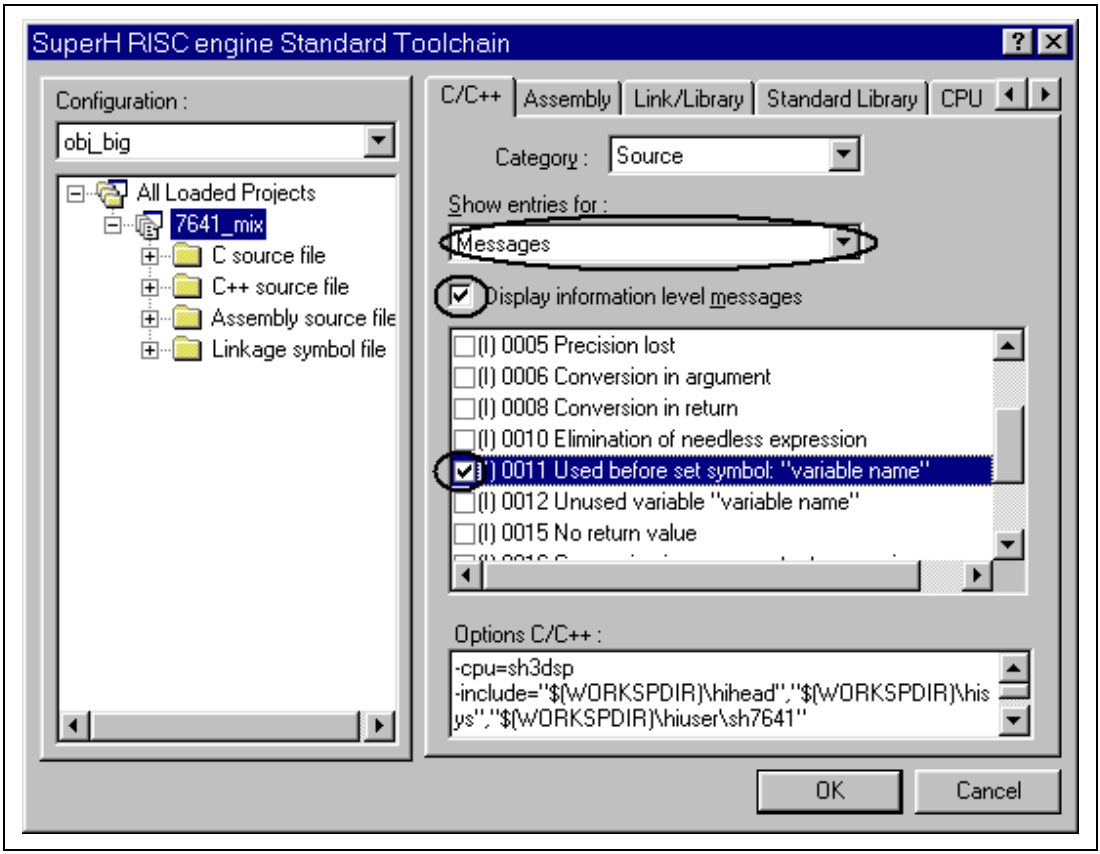


Figure 5.28 Window for Specifying Output of Compiler Information Messages

(5) Incorrect Function Call Using a Pointer Variable

When a pointer variable value becomes illegal during a function call through the pointer variable, the program execution address in the called function becomes illegal and the system may go down or may be reset. When a function should be called through a pointer variable, be sure to confirm that the source code is correct.

If the system-down cause is outside the program area and the caller of the target function cannot be determined, use the trace function of an ICE or an emulator to check the program execution flow.

The following shows an example of a function call through an illegal pointer variable.

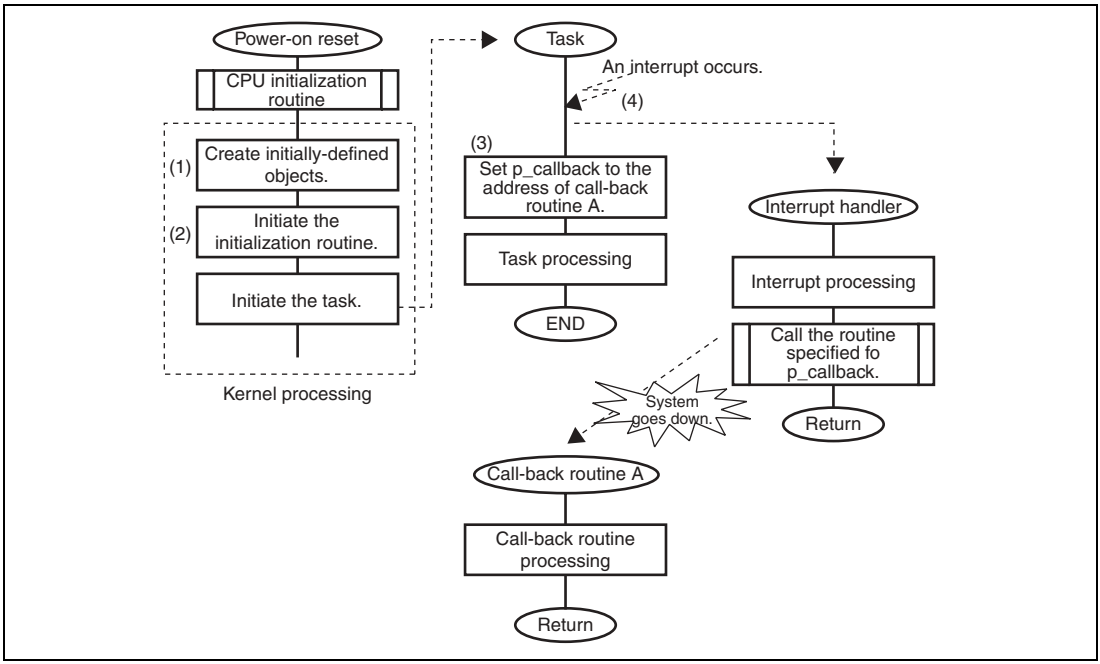


Figure 5.29 Example of a Function Call through an Illegal Pointer Variable

(1) Define an interrupt handler during initial definition.

(2) Enable hardware interrupts through the initialization routine.

(3) If no interrupt is generated, the pointer variable (p_callback) is set to the call-back address value in the task context.

(4) If an interrupt is generated before the pointer variable is set to the call-back routine address, a call is made to an illegal address, that is, the execution address is illegal and the system goes down.

In the above case, take appropriate measures so that the call-back routine is not called until the call-back routine address is determined or no interrupt is generated before the pointer variable (p_callback) is set up.

5.7 FAQs about Debugging

This section answers questions about debugging which are frequently asked by users of the HI series OS.

FAQ Contents:

5.7.1 Saving a Program in ROM

Classification: Debugging

Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4

My program correctly runs on an ICE but cannot run correctly after it is stored in ROM.

What causes this problem?

Answer

The sections must be initialized when program execution is started.

The initialized data area (D section) in a program written in the C language must be copied from ROM to RAM when program execution is started. Therefore, the initialized data area must be allocated to both ROM and RAM. This allocation can be done by using the ROM support function of the linkage editor. For the ROM support function, refer to the user's manual of the cross compiler used.

The sections must be initialized by the CPU initialization routine.

The following shows how to initialize the sections, using the CPU initialization routine provided together with each HI series OS as an example.

(Continued on next page)

(Continued from previous page)

Answer

```
/*****************************************************************/
/*  FILE        = 7604_cpuini.c ;                                */
/*  CPU type  = SH7604                                           */
/*****************************************************************/
#include    <machine.h>
#include    "itron.h"
#include    "kernel.h"

/* extern  void    _INITSCT(void); */   /* section-initialize routine          */

#pragma section _hicpuini
#pragma noregsave(hi_cpuini)

void            hi_cpuini(void)
{

/*** Initialize Hardware Environment ***/

/*** Initialize Software Environment  ***/

/*  _INITSCT();    */                    /* Call section-initialize routine  */

      vsta_knl();                        /* Start kernel                     */
}
```

Delete the comment characters (/* */) to call the section initialization processing.

Figure 5.30 Example of CPU Initialization Routine (HI7000/4 Series)

(Continued on next page)

(Continued from previous page)

Answer

```
/*****************************************************************************/
/*  FILE          = 7604_initsct.c  ;                                        */
/*****************************************************************************/
#include    <machine.h>
#include    "itron.h"

extern int *B_BGN, *B_END, *D_BGN, *D_END, *D_ROM;
extern void _INITSCT(void);

#pragma section _hicpuini
/*****************************************************************************/
/*  NAME          = _INITSCT ;                                               */
/*  FUNCTION  = Section Initialize routine ;                                 */
/*****************************************************************************/
void _INITSCT(void)
{
    register int *p, *q;
    for(p=B_BGN; p<B_END; p++)          /* 0 clear B-section                */
        *p = 0;
    for(p=D_BGN,q=D_ROM; p<D_END; p++, q++) /* Copy D-section -> R-section */
        *p = *q;
}
```

Figure 5.31 Example of Section Initialization Processing (HI7000/4 Series)

(Continued on next page)

(Continued from previous page)

Answer

```
_H_2S_CPUINI:
        mov.l   #CPUINI_SP:32,sp            ;: get CPUINI_SP
        mov.b   @SYSCR:32,r0L               ;: get SYSCR
        and.b   #low~(INTM0|INTM1):8,r0L    ;: clear interrupt mode bit
        or.b    #low (INTM0|INTM1):8,r0L    ;: set interrupt mode = 3
        mov.b   r0L,@SYSCR:32               ;: set SYSCR
;
        mov.b   @MSTPCRH:32,r0L             ;: get MSTPCRH
        and.b   #low TPU:8,r0L              ;: set TPU bit off
        mov.b   r0L,@MSTPCRH:32             ;: set MSTPCRH
;
        .aifdef DX
        jsr     @_HI_DEAMON_INI             ;: call to init deamon code
        .aendi
;
        jsr     @_h_cpuini_c                ;: call to C-language initialize routine
;
        jmp     @_H_2S_INIT                 ;: goto HI2000/3 initialize module
;
```

Add a call to the CPU initialization routine written in C.

Note: In this example, h_cpuini_c is assumed as the CPU initialization routine written in C.

Figure 5.32 Example of CPU Initialization Routine (HI2000/3)

```
void h_cpuini_c(void)
{

/*** Initialize Hardware Environment ***/

/*** Initialize Hardware Environment ***/

  _INITSCT();                    /* Call section-initialize routine  */

}
```

Add the following as necessary.
- Initialization of the bus state controller
- Initialization of the external memory (SDRAM)

Call the section initialization processing in the standard library to clear uninitialized data to 0 and to copy the uninitialized data from ROM to RAM.

Figure 5.33 Example of a Call to Section Initialization Processing (HI2000/3)

(Continued on next page)

(Continued from previous page)

Answer

```
_KERNEL_H_CPUINI:
        mov.l   #_KERNEL_HI_OS_SP:32,sp     ;: SP <- OS stack
        mov.l   #VBR_ADR,er0                ;:
        ldc.l   er0,vbr                     ;: set VBR address
        mov.l   #h'fffff00,er0              ;: initial SBR
        ldc.l   er0,sbr                     ;: initial SBR
;
;       mov.w   #h'00ff,@ABWCR:32           ;: set ABWCR
;       mov.w   #h'0000,@ASTCR:32           ;: set ASTCR
;       mov.w   #h'0000,@WTCRA:32           ;: set WTCRA
;       mov.w   #h'0000,@WTCRB:32           ;: set WTCRB
;
        mov.b   #INTM1,r0L                  ;: set interrupt mode 2
        mov.b   r0L,@INTCR:32               ;: set INTCR
;
        mov.w   @MSTPCRA:32,r0              ;: get MSTPCRA
        and.w   #MSTPA0:16,r0               ;: set TPU bit off
        mov.w   r0,@MSTPCRA:32              ;: set MSTPCRA
;
        jmp     @_h_cpuini_c
;
```

Add a call to the CPU initialization routine written in C.

Note: In this example, h_cpuini_c is assumed as the CPU initialization routine written in C.

Figure 5.34 Example of CPU Initialization Routine (HI1000/4)

```
void h_cpuini_c(void)
{

/*** Initialize Hardware Environment ***/

/*** Initialize Hardware Environment ***/

  _INITSCT;                    /* Call section-initialize routine */

  vsta_knl();                  /* Start kernel */
}
```

Add the following as necessary.
- Initialization of the bus state controller
- Initialization of the external memory (SDRAM)

Call the section initialization processing in the standard library to clear uninitialized data to 0 and to copy the uninitialized data from ROM to RAM.

Figure 5.35 Example of a Call to Section Initialization Processing (HI1000/4)

5.7.2 System-Down When Memory Pool is Used

Classification: Debugging

Question	HI7000/4	HI7700/4	HI7750/4	HI2000/3	HI1000/4

When a memory block is acquired and released in a variable-length memory pool, the system goes down. What causes this problem?

Answer

The user program seems to use memory beyond the memory block acquired from a variable-length memory pool.

In a variable-length memory pool, when a memory block is acquired, a 16-byte kernel management area is allocated in the memory pool. The following shows the configuration of the variable-length memory blocks in a memory pool.

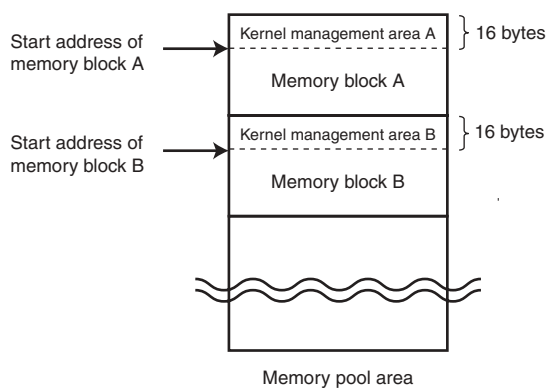


Figure 5.36 Configuration of Variable-Length Memory Blocks

(Continued on next page)

(Continued from previous page)

Answer

If the kernel management area is accidentally overwritten with the user program, the system goes down or hangs up when the kernel accesses the kernel management area to release a memory block.

Use an ICE or an emulator in the following procedure to check whether the user program uses memory beyond the acquired memory block.

(1) Specify the required size + 4 as the memory block size when acquiring a memory block.
(2) Set a hardware break at the end address of the acquired memory block (start address + required size + 1) so that a break occurs when this address is read or written to.
(3) Execute the program.

If program execution stops due to the specified hardware break, the user program has attempted to use memory beyond the available memory block range.

HI Series OS Application Note

Publication Date: Rev.1.00, Dec 19, 2003
Rev.3.00, Jan 12, 2005
Published by: Sales Strategic Planning Div.
Renesas Technology Corp.
Edited by: Technical Documentation & Information Department
Renesas Kodaira Semiconductor Co., Ltd.

Renesas Technology Corp. Sales Strategic Planning Div. Nippon Bldg., 2-6-2, Ohte-machi, Chiyoda-ku, Tokyo 100-0004, Japan

RENESAS SALES OFFICES

http://www.renesas.com

Refer to "**http://www.renesas.com/en/network**" for the latest and detailed information.

Renesas Technology America, Inc.
450 Holger Way, San Jose, CA 95134-1368, U.S.A
Tel: <1> (408) 382-7500, Fax: <1> (408) 382-7501

Renesas Technology Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K.
Tel: <44> (1628) 585-100, Fax: <44> (1628) 585-900

Renesas Technology Hong Kong Ltd.
7th Floor, North Tower, World Finance Centre, Harbour City, 1 Canton Road, Tsimshatsui, Kowloon, Hong Kong
Tel: <852> 2265-6688, Fax: <852> 2730-6071

Renesas Technology Taiwan Co., Ltd.
10th Floor, No.99, Fushing North Road, Taipei, Taiwan
Tel: <886> (2) 2715-2888, Fax: <886> (2) 2713-2999

Renesas Technology (Shanghai) Co., Ltd.
Unit2607 Ruijing Building, No.205 Maoming Road (S), Shanghai 200020, China
Tel: <86> (21) 6472-1001, Fax: <86> (21) 6415-2952

Renesas Technology Singapore Pte. Ltd.
1 Harbour Front Avenue, #06-10, Keppel Bay Tower, Singapore 098632
Tel: <65> 6213-0200, Fax: <65> 6278-8001

Colophon 2.0

HI SeriesOS
Application Note

Renesas Electronics Corporation
1753, Shimonumabe, Nakahara-ku, Kawasaki-shi, Kanagawa 211-8668 Japan

REJ05B0364-0300