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.

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.

User's Manual

RENESAS

HI1000/4 V.1.04

User's Manual
(H8SX, H8S Family Realtime OS)

Renesas Microcomputer Development Environment System

R0R41600TRW01E

Renesas Electronics
www.renesas.com

Rev.2.00 2006.06

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.

Regarding trademarks:

1. TRON is an acronym of “The Realtime Operating system Nucleus”. ITRON is an acronym of the “Industrial TRON”. µITRON is an acronym of the “Micro Industrial TRON”.
2. TRON, ITRON, and µITRON are not a name which indicates a specific group of the commodity or the commodity.
3. The µITRON4.0 specification is open realtime operating system specification. The µITRON specification was designed by the ITRON Committee in the TRON association. The µITRON4.0 specification can be downloaded from ITRON project homepage (http://www.ertl.jp/ITRON/home-e.html).
4. Microsoft® Windows® 98 operating system, Microsoft® Windows® Millennium Edition (Windows® ME) operating system, Microsoft® Windows NT® operating system, Microsoft® Windows® 2000 operating system, and Microsoft® Windows® XP operating system are registered trademarks of Microsoft Corporation in the United States and/or other countries.
5. All other product names are trademarks or registered trademarks of the respective holders.

Preface

This manual describes how to use the HI1000/4. Before using the HI1000/4, please read this manual to fully understand the operating system.

Notes on Descriptions

HEW	Abbreviation of High-performance Embedded Workshop, which is an integrated development tool.
H', 0x, and D'	For hexadecimal integers, prefix H' or 0x is attached. For decimal integers, prefix D' is attached. If no prefix is attached, a decimal integer is assumed.
nnnn	Bold-faced-italic ***nnnn*** is the CPU name used for the sample file name. Example: The timer driver file name is ***nnnnz***_tmrdrv.c in this manual, but the actual timer driver file for the H8SX/1650 is 1650tmrdrv.src.
CFG_MAXTSKID	A variable name beginning with CGF_ is specified for the configurator. For details, refer to section 7.4.4, Configurator Settings, or online help for the configurator.

Renesas Technology Homepage

Various support information are available on the following Renesas Technology homepage:

http://www.renesas.com/en/tools/

Contents

Section 1 Configuration of This Manual

This manual consists of the following sections:

Section 2 ‘Introduction’: Overview of HI1000/4

Section 3 ‘Introduction to Kernel’: Basic concept of kernel

Section 4 ‘Kernel Functions’: All the functions of kernel

Section 5 ‘Service Calls’: Specifications of service calls

Section 6 ‘Application Program Creation’: Methods for creating a task or a handler

Section 7 ‘Configuration’: Position, functions, and usage of the configurator

Section 8 ‘HEW Workspace and Projects’: Description of sample programs and methods to generate a load module using these programs

Section 9 ‘Calculation of Work Area Size’: Methods for calculating stack size

Section 10 ‘Information during System Down’: Methods for handling errors such as a system failure

Section 11 ‘Reference Listing’: References for service calls or error codes

Section 12 ‘Appendix’: Definition files dependent on the device

Section 2 Introduction

2.1 Overview

The HI1000/4 V.1.04, which is a realtime multitasking OS for microcomputers, runs on the CPUs of the H8SX family, H8S family, and AE5-series for use in IC cards and is based on the μITRON4.0 specifications.

The HI1000/4 supports the following interrupt control modes and CPU operating modes.

H8SX Family and AE5 Series

- Interrupt control mode: 0 or 2
- CPU operating mode: Advanced mode

H8S Family

- Interrupt control mode: 0, 1, 2 or 3
- CPU operating mode: Normal or advanced mode

2.2 Features

Realtime and Multitasking Processing: The HI1000/4 kernel is based on the μITRON4.0 specifications.

- Priority-based task scheduling
- Task management, including the initiation and termination of tasks, and modification of the priority
- Task synchronization, including suspension and resumption of tasks, and delay of current task
- Inter-task synchronization and communication using semaphores, event flags, data queues, and mailboxes
- Extended inter-task synchronization and communication using mutexes
- Memory pool management, including control over the allocation and return of memory blocks
- Control over timing, such as setting and referring to the system clock, and controlling the cyclic handler
- System management
- Interrupt management
- System configuration management

Compact Kernel with Optional Selection of Kernel Functions: The sizes of the kernel program and its work area are reduced to minimize the ROM and RAM sizes required by the user system. By linking the application and kernel library at the same time, only the service calls required for the user system will be included in the kernel.

Configurator: The configurator is supported to allow easy kernel configuration on the GUI screen.

Sample Programs: The following samples are provided.

- System down routines
- Timer driver for on-chip timers of H8S™, H8SX™, and AE5™ microcomputers
- Various handlers and routines
- CPU initialization handlers
- Configurator setting file
- HEW workspaces for load module creation

Debugging Extension (Optional): A debugging extension for adding multitasking debugging functions to the integrated development environment "HEW" is available. This debugging extension supports the following functions.

- Viewing the states of objects including tasks
- Operating objects (e.g. initiating tasks or setting event flags)
- Displaying service call history

The free debugging extension can be downloaded from our website.

2.3 Operating Environment

The operating environment is shown in Table 2.1.

Table 2.1 Operating Environment

Item	Requirement
Target microcomputer	H8S™, H8SX™, or AE5™ microcomputer
Host computer	Personal computer operated under Windows® 98, Windows® Millennium Edition (Windows® Me), Windows NT®4.0, Windows® 2000, or Windows® XP
Sample HEW workspace and project	HEW version 4.0 or later (H8SX, H8S and H8 Family C/C++ compiler package version 6.01 Release 01 or later)

Section 3 Introduction to Kernel

3.1 Principles of Kernel Operation

The kernel is the nucleus program of a realtime operating system.

The kernel enables one microcomputer to appear as if multiple microcomputers are operating. How does the kernel do this?

As shown in figure 3.1, the kernel operates multiple tasks in a time-division manner; the kernel switches (schedules) running tasks at intervals to make it appear as if multiple tasks are running at the same time.

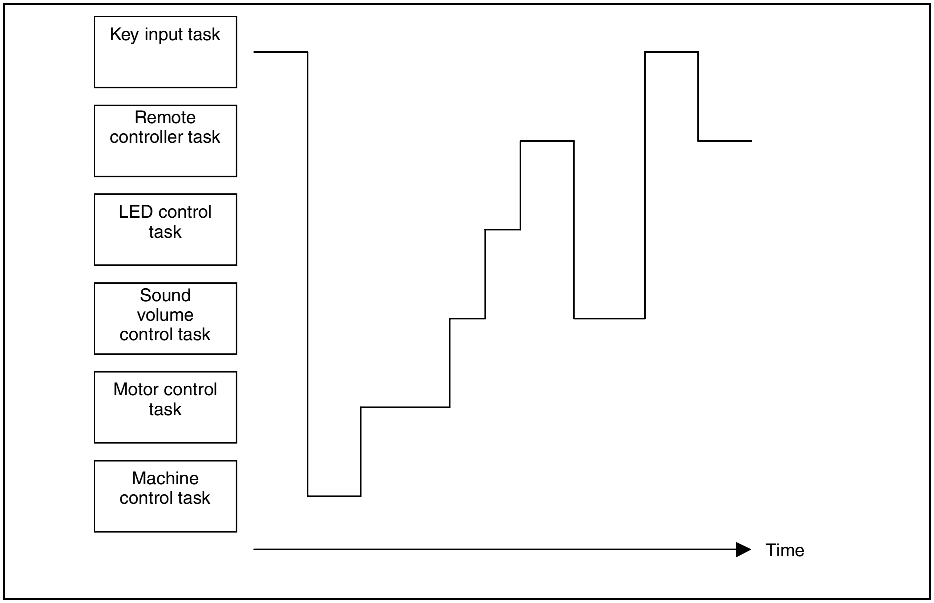


Figure 3.1 Time-Division Operation of Tasks

This task scheduling is also called task dispatch.

The kernel schedules (dispatches) tasks in the following cases.

- When a task itself requests a dispatch
- When an event (such as an interrupt) outside the current task requests a dispatch

This means that tasks are not switched at determined intervals as in a time-sharing system. This type of scheduling is generally called event-driven scheduling.

After tasks are scheduled, a task is resumed from the point where it is suspended (figure 3.2).

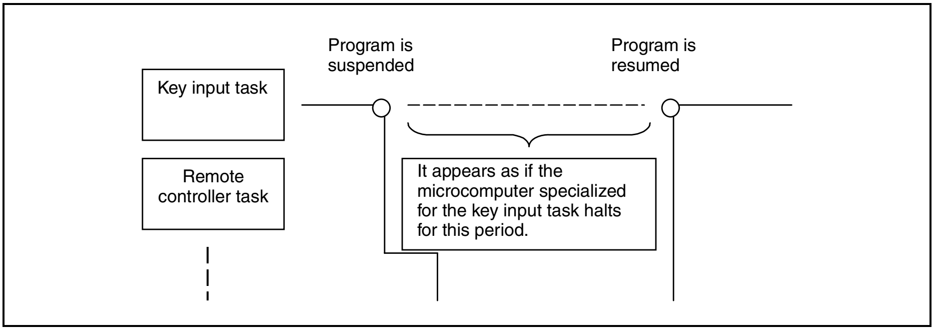

Program is suspended
Program is resumed
Key input task
Remote controller task
It appears as if the microcomputer specialized for the key input task halts for this period.

Figure 3.2 Suspending and Resuming a Task

In figure 3.2, while a task is running after obtaining control from the key input task, it appears to the programmer as if the microcomputer specialized for that program has halted.

The kernel restores the register contents stored when the task is suspended, and resumes the task in the state where it was suspended. In other words, task scheduling means saving the register contents for the current task in a memory area prepared for that task management and restoring the register contents of the next task to be resumed (figure 3.3).

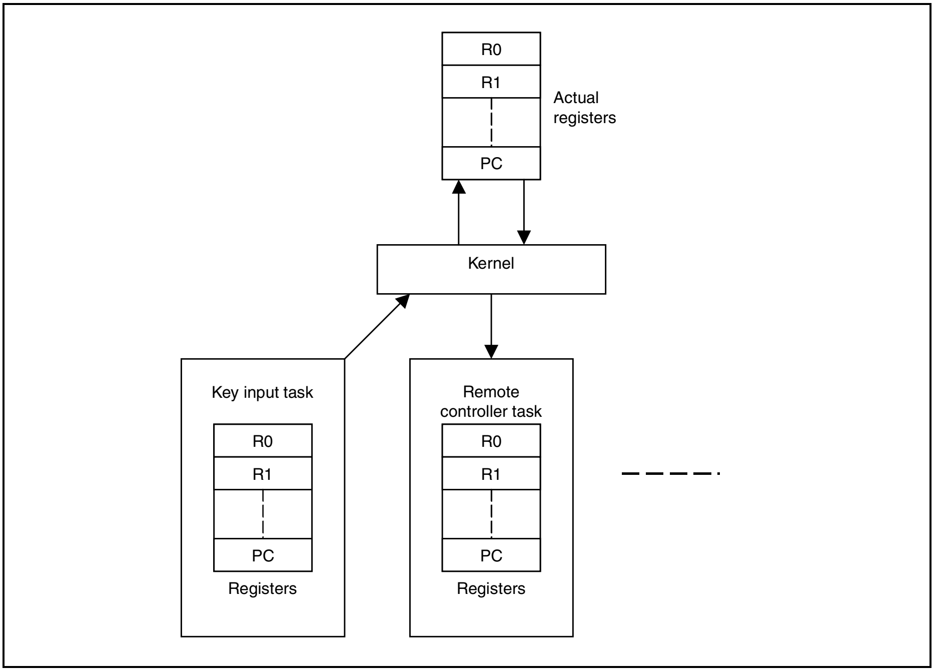


Figure 3.3 Task Scheduling

To execute tasks, stack areas are required in addition to registers. A separate stack area must be allocated for each task.

3.2 Service Calls

How should the programmer use kernel functions in a program?

To use kernel functions, they must be called in a program. This call is a service call. Through service calls, requests for various operations such as task initiation or wait for an event can be sent to the kernel.

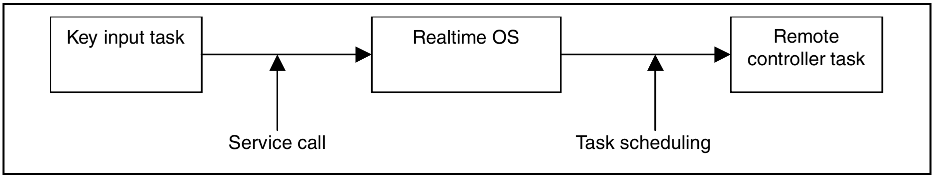


Figure 3.4 Service Call

In actual programs, a service call is issued as a C-language or an assembly-language function.

C-language function	Assembly-language function

`act_tsk(1);`	`MOV.W #TSKID,R0`
	`JSR @_act_tsk`

3.3 Objects

The targets of operation through service calls, such as tasks or semaphores, are called objects. Objects are distinguished by their IDs.

```
act_tsk(1);   /* Initiates the task with ID 1. */
```

Generally, IDs should be specified by the programmer when objects are created.

IDs can also be assigned automatically when objects are created through the configurator. In this case, the automatically assigned IDs are defined in header files (kernel_id.h and kernel_id.inc) output from the configurator as shown below.

```
#define ID_TASK1  1
```

By using this example definition, the above task initiating service call is described as follows.

```
act_tsk(ID_TASK1);   /* Initiates the task with ID "ID_TASK1". */
```

3.4 Tasks

The following describes how the kernel manages tasks.

3.4.1 Task State

The kernel checks the task state to control whether to start execution of a task. For example, figure 3.5 shows the state of the key input task and its execution control. When a key input is detected, the kernel must execute the key input task; that is, the key input task enters the running state. While waiting for a key input, the kernel does not need to execute the key input task; that is, the key input task is in the waiting state.

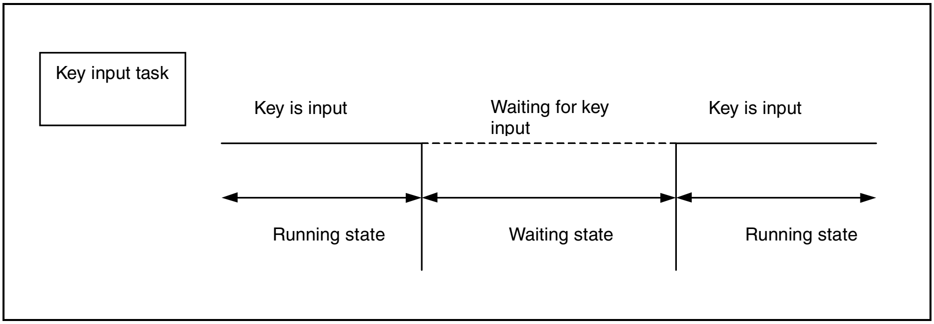


Figure 3.5 Task State

In the µITRON4.0 specifications, a task transits the seven states shown in figure 3.6.

For details on state transitions of this kernel, see Figure 4.1, Task State Transition Diagram, and Figure 4.2, Task-State Transitions for the Shared Stack Function.

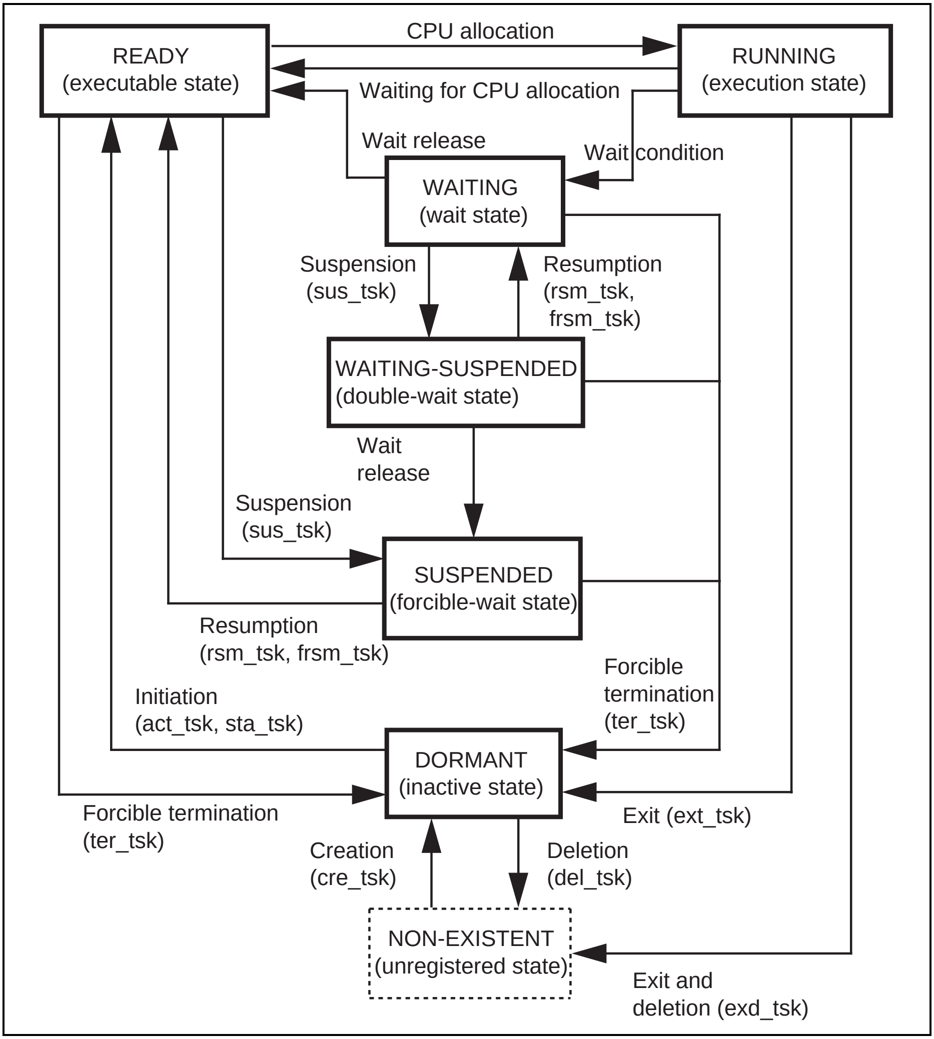


Figure 3.6 Task State Transition Diagram

(1) NON-EXISTENT (Unregistered) State

A task has not been registered in the kernel. It is a virtual state.

(2) DORMANT (Inactive) State

A task has been registered in the kernel, but has not yet been initiated, or has already been terminated.

(3) READY (Executable) State

A task is ready for execution, but cannot be executed because another higher priority task is currently running.

(4) RUNNING (Execution) State

A task is currently running in the CPU. The kernel puts the READY task with the highest priority in the RUNNING state.

(5) WAITING (Wait) State

A task issues a service call such as tslp_tsk to put itself to sleep when it can no longer continue execution. The task is released (awakened) from the WAITING (sleep) state when the wup_tsk service call is issued, and it then makes the transition to the READY state.

(6) SUSPENDED (Forcible-Wait) State

A task has been forcibly suspended by another task by the sus_tsk service call.

(7) WAITING-SUSPENDED (Double-Wait) State

This state is a combination of the WAITING state and SUSPENDED state.

3.4.2 Task Scheduling (Priority and Ready Queue)

For each task, a task priority is assigned to determine the priority of processing. A smaller value indicates a higher priority level and level 1 is the highest priority.

The kernel selects the highest-priority task from the READY tasks and puts it in the RUNNING state.

The same priority can be assigned for multiple tasks. When there are multiple READY tasks with the highest priority, the kernel selects the first task that became READY and puts it in the RUNNING state. To implement this behavior, the kernel has ready queues, which are READY task queues waiting for execution.

Figure 3.7 shows the ready queue configuration. A ready queue is provided for each priority level, and the kernel selects the task at the head of the ready queue for the highest priority and puts it in the RUNNING state.

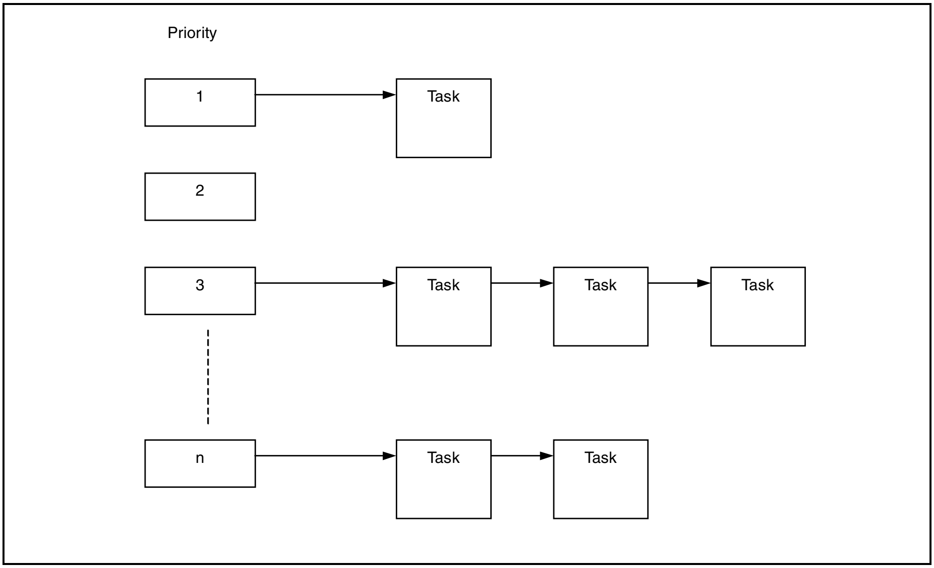


Figure 3.7 Ready Queues (Waiting for Execution)

Section 4 Kernel Functions

4.1 Overview

The kernel, which is the nucleus of the operating system, enables realtime multitasking.

The three major roles of the kernel used by the user are as follows:

- Response to events
  Recognizes events generated asynchronously, and immediately executes a task to process the event.
- Task scheduling
  Schedules task execution on a priority basis.
- Service call execution
  Accepts various requests for processing (service calls) from tasks and performs the appropriate processing.

4.2 Kernel Functions

An application program can issue service calls to use almost any kernel function.

Task Management: When a task is executed, the CPU is allocated to the task. The kernel controls the order of CPU allocation, and of the start and end of tasks. Multiple tasks can share one stack by using the shared stack function.

Task Synchronization Management: Performs basic synchronous processing for tasks, such as suspension and resumption of task execution.

Synchronization and Communication Management: Uses event flags, semaphores, data queues, and mailboxes for inter-task synchronization and communication.

Extended Synchronization and Communication Management: Uses mutexes for inter-task synchronization and communication.

Memory Pool Management: The size of the memory pool can be fixed or variable.

Time Management: Manages time-related information for the system and monitors task execution times for control purposes.

System State Management: Performs system state management functions, such as modifying or referencing the context or system states.

Interrupt Management: Initiates the appropriate interrupt handlers in response to external interrupts. The interrupt handler performs appropriate interrupt processing, and notifies tasks of interrupts.

System Configuration Management: Performs system configuration management functions, such as reading the kernel version number.

4.3 Processing Units

An application program is executed in the following processing units.

Task: A task is a unit controlled by multitasking.

Interrupt Handler: An interrupt handler is executed when an interrupt occurs.

CPU Exception Handler: A CPU exception handler is executed when a CPU exception occurs.

Time Event Handler (Cyclic Handler): A time event handler is executed when a specified cycle has been reached.

4.4 System State

The system state is classified into the following orthogonal states.

- Task context state/non-task context state
- Dispatch-disabled state/dispatch-enabled state
- CPU-locked state/CPU-unlocked state

The system operations and available service calls are determined based on the above system states.

4.4.1 Task Context State and Non-Task Context State

The system is executed in either the task context state or non-task context state. The difference between the task and non-task context states is described in Table 4.1.

Table 4.1 Task Context State and Non-Task Context State

Item	Task Context State	Non-Task Context State
Available service calls	Service calls that can be called from the task context	Service calls that can be called from the non-task context
Task scheduling	Refer to Sections 4.4.2 and 4.4.3	Does not occur

The following processing is executed in non-task context.

- Interrupt handler
- CPU exception handler
- Time event handler (cyclic handler)
- A part where the interrupt mask is changed to a value other than 0 by the chg_ims service call

4.4.2 Dispatch-Disabled State/Dispatch-Enabled State

The system is placed in either the dispatch-disabled state or dispatch-enabled state. In the dispatch-disabled state, task scheduling is not allowed and service calls that place a task in the WAITING state cannot be used.

Issuing the dis_dsp service call during task execution changes the system state to dispatch-disabled state. Issuing the ena_dsp service call will return the system state to the dispatch-enabled state. Issuing the sns_dsp service call will check whether the system state is in the dispatch-disabled state or not.

4.4.3 CPU-Locked State/CPU-Unlocked State

The system is placed in either the CPU-locked state or CPU-unlocked state. In the CPU-locked state, interrupts are disabled and task scheduling is not performed. Note however that interrupts with interrupt levels higher than the kernel interrupt mask level (CFG_KNLMSKLVL) defined in the system configuration file can be accepted.

Service calls that can be called from the CPU-locked state are listed below. Normal system operation cannot be guaranteed when service calls other than these are called from the CPU-locked state (E_CTX error detection is omitted). Note however that when service calls that shift tasks to the WAITING state are called, an E_CTX error is returned.

- ext_tsk
- loc_cpu, iloc_cpu
- unl_cpu, iunl_cpu
- sns_ctx
- sns_loc
- sns_dsp
- sns_dpn
- vsta_knl, ivsta_knl
- vsys_dwn, ivsys_dwn

Issuing the loc_cpu or iloc_cpu service call during task execution changes the system state to the CPU-locked state. Issuing the unl_cpu or iunl_cpu service call will return the system state to the CPU-unlocked state. In addition, issuing the sns_loc service call will check whether the system state is in the CPU-locked state or not.

4.5 Priority

In the HI1000/4, each processing unit is processed with the following priority.

(1) Interrupt handlers, time event handlers (cyclic handlers), and CPU exception handlers

(2) Dispatcher (part of kernel processing)

(3) Tasks

The dispatcher is a kernel processing that switches the task to be executed.

The priority of an interrupt handler becomes higher when the interrupt level is higher.

The priority of a time event handler is the same as the timer interrupt level.

The priority of a CPU exception handler is higher than that of the dispatcher, and also lower than that of other processings which have higher priorities than the processing where the CPU exception occurred.

The priority between tasks depends on the priority of these tasks.

When the following service calls are called, a priority which does not apply to the above description can be temporarily generated:

(a) When dis_dsp is called, the priority will be between (1) and (2) above. The state returns to the former state by calling ena_dsp.
(b) When loc_cpu or iloc_cpu is called, the priority will be the same as that of the interrupt handler whose interrupt level is the same as the kernel interrupt mask level. The state returns to the former state by calling unl_cpu or iunl_cpu.
(c) While the interrupt mask level is changed to other than 0 by chg_ims, the priority is the same as an interrupt handler of the same level.

4.6 Objects

Objects such as tasks and semaphores are manipulated by service calls. Objects are identified by ID numbers or object numbers.

The maximum number can be specified for almost all objects at system configuration.

4.7 Tasks

In a realtime multitasking system, the user prepares an application program in terms of a set of tasks that can be processed independently and in parallel.

A task communicates with other tasks by using service calls. Such service calls can be used to have the kernel process events generated asynchronously.

Table 4.2 and Table 4.3 list the service calls that operate tasks.

Table 4.2 Task Management Service Calls

Service Call	Description
act_tsk, iact_tsk	Starts task
can_act	Cancels task start request
sta_tsk, ista_tsk	Starts task (specifies start code)
ext_tsk	Exits current task
ter_tsk	Forcibly terminates a task
chg_pri	Changes task priority
get_pri	Refers to task priority
ref_tsk, iref_tsk	Refers to task state
ref_tst, iref_tst	Refers to task state (simple version)

Table 4.3 Task Synchronization Service Calls

Service Call	Description
slp_tsk	Sleep task
tslp_tsk	Sleep task with timeout
wup_tsk, iwup_tsk	Wakeup task
can_wup	Cancel wakeup task
rel_wai, irel_wai	Release WAITING state forcibly
sus_tsk	Suspend task
rsm_tsk	Resume task
frsm_tsk	Resume task forcibly
dly_tsk	Delay current task

4.7.1 Task State and Transition

A task can be in any of the following six states because of external or internal events, as shown in Figure 4.1.

DORMANT (Inactive) State: A task has been registered in the kernel, but has not yet been initiated, or has already been terminated.

READY (Executable) State: A task is ready for execution, but cannot be executed because another higher priority task is currently running.

RUNNING (Execution) State: A task is currently running in the CPU. The kernel puts the READY task with the highest priority in the RUNNING state.

WAITING (Wait) State: A task issues a service call such as tslp_tsk to put itself to sleep when it can no longer continue execution. The task is released (awakened) from the WAITING (sleep) state when the wup_tsk service call is issued, and it then makes the transition to the READY state.

SUSPENDED (Forcible-Wait) State: A task has been forcibly suspended by another task by the sus_tsk service call.

WAITING-SUSPENDED (Double-Wait) State: This state is a combination of the WAITING state and SUSPENDED state.

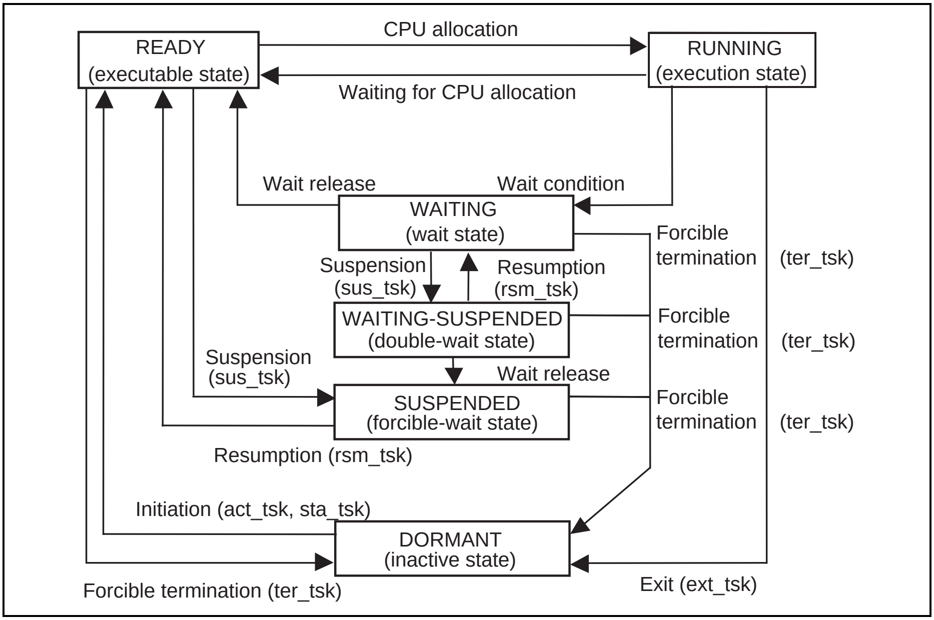


Figure 4.1 Task State Transition Diagram

4.7.2 Task Initiation

A task is initiated when it moves from the DORMANT state to become the READY state. A task can be initiated by one of the following two methods:

- By issuing the act_tsk or sta_tsk service call for the target task
- By specifying the READY state for the task's initial state at system configuration

4.7.3 Task Scheduling

Scheduling: Task scheduling means that the kernel determines the order of execution of executable tasks, that is, the order of allocating the CPU to a task that is READY. The kernel selects one READY task and puts it in the RUNNING state. If no tasks are READY, the kernel enters the idle state, cancels interrupt masking and enters an infinite loop, waiting for a task to be awoken by an interrupt. When more than one task is READY, the order of execution is determined by a CPU allocation wait queue, which is called a ready queue.

There is a ready queue for each maximum level (CFG_MAXTSKPRI) of task priority, and each queue operates on a first-come, first-served (FCFS) basis. The lower the number, the higher the priority.

When the system is in the non-task context state, task scheduling is delayed until the system returns to the task context state.

Round-Robin Scheduling: The kernel also supports round-robin scheduling, where the CPU allocates equal time to tasks with a given priority by rotating the ready queue at specific intervals.

The round-robin scheduling can be achieved by issuing the rot_rdq service call that manipulates the ready queues.

- Round-robin scheduling with rot_rdq service call

  By issuing rot_rdq at specific cycles, execution can be switched at specific intervals to a task that has the same priority as the executing task.

Limitations of Scheduling: When the system enters the dispatch-disabled state by the dis_dsp service call, task scheduling is disabled. Task scheduling is enabled when the system enters the dispatch-enabled state by the ena_dsp service call.

When the system enters the CPU-locked state by the loc_cpu service call, both task scheduling and all interrupts (except for interrupts not managed by the kernel) are disabled. Task scheduling and interrupts are enabled when the system enters the CPU-unlocked state by the unl_cpu service call.

4.7.4 Task Termination

Task termination means that a task is finished and can enter the DORMANT state.

- ext_tsk is issued
- ter_tsk is issued for the target task

When a task is terminated and then re-initiated, or when the number of initiation request queues specified in the act_tsk service call is other than 0, the task starts from the initial state. If the number of initiation request queues specified in the act_tsk service call is other than 0 and a target task is in the shared-stack WAITING state, the task that was in the shared-stack WAITING state is released first and the original task then enters the shared-stack WAITING state (tasks are switched).

A task must release its resources before execution is completed. Note that when a task is terminated, the current task unlocks the mutexes it was locking.

4.7.5 Task Stack

In this kernel, a task stack is statically allocated at system configuration. A shared stack function, which allows more than one task to use a single stack, is available.

4.7.6 Shared Stack Function

The shared stack function allows more than one task to share a single stack. This reduces the total stack area used.

A shared stack is defined by the configurator. However, only one task in a task group that shares a stack can be executed at a time.

A shared stack is released from a task when the task becomes DORMANT. When there are tasks waiting for the shared stack, the task at the head of the wait queue will use the stack and enter the READY state.

Tasks in the shared-stack WAITING state are managed on a first-in first-out (FIFO) basis, regardless of their priority. Tasks are sent to the shared-stack wait queue in the order in which they were initiated.

When multiple tasks that were defined to use the same stack are initiated simultaneously, the task that was initiated first uses the stack first and the remaining tasks enter the shared-stack WAITING state.

Figure 4.2 shows the task-state transitions for the shared stack function.

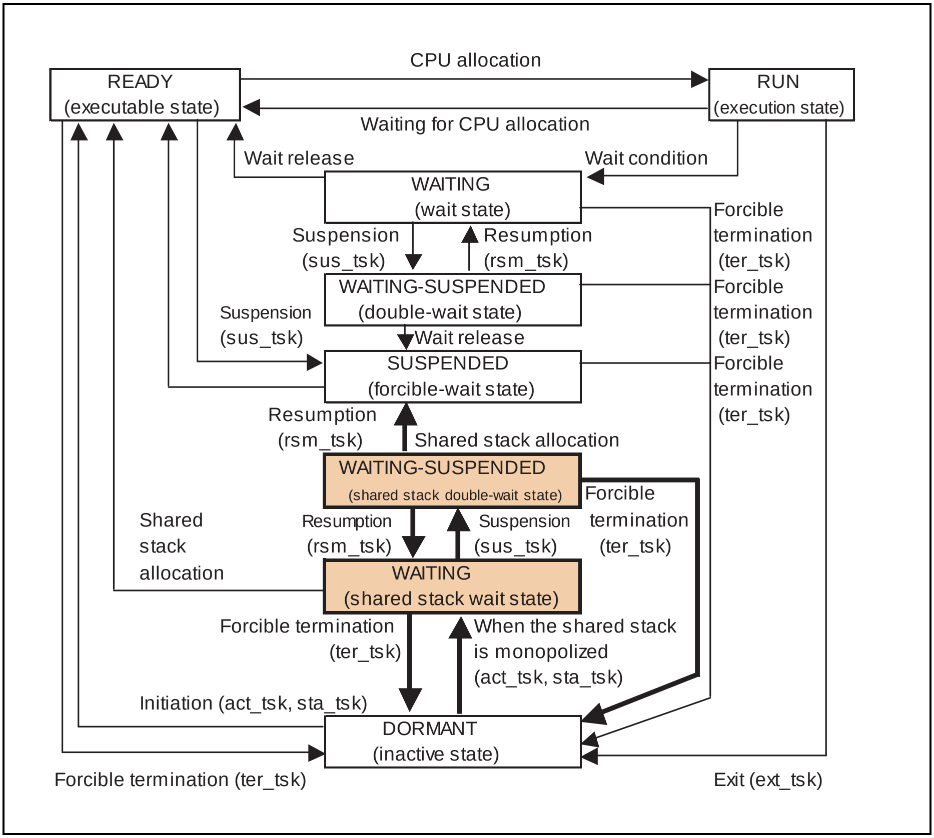


Figure 4.2 Task-State Transitions for the Shared Stack Function

4.7.7 Exclusive Control

In some cases, during execution of one task, the task may need to be executed exclusively with another program. For example, when task A and interrupt handler B refer to and modify the same global variable, reference and modification must be exclusive. The target program to control exclusivity can be an interrupt handler, specific task, or any other task.

Table 4.4 shows the way to ensure that tasks execute exclusively.

Table 4.4 Exclusive Control

Exclusive Control	Disabled Interrupts	Task Scheduling
Enter the CPU-locked state by issuing loc_cpu	Equal to or lower than the kernel interrupt mask level	None
Mask interrupts by issuing chg_ims (When chg_ims is issued from the task context state, the non-task context state is entered.)	Equal to or lower than the specified interrupt mask level	None
Enter the dispatch-disabled state by issuing dis_dsp	None	None
Exclusive control by semaphore	None	The kernel schedules tasks; however, in tasks using the same semaphore, the number of tasks entering the READY state simultaneously is limited to the semaphore initial count value.
Exclusive control by mutex	None	The kernel schedules tasks; however, tasks using the same mutex cannot enter the READY state simultaneously. In addition, for tasks using the same mutex, the task priority inversion will not occur.

4.8 Synchronization and Communication

For synchronization and communication purposes, the kernel has the following objects which are independent of tasks.

- Semaphore: Exclusively controls multiple resources.
- Event flag: Waits for several events and synchronizes task operations.
- Data queue: Transmits and receives data (passes 1-word data).
- Mailbox: Transfers data (passes data address).
- Mutex: Exclusively controls a single resource (with a function to avoid priority inversion).

4.8.1 Semaphore

The elements required for task execution are called resources. They include I/O or shared memory.

A semaphore is an object that provides exclusive control and a synchronization function by expressing the existence and the number of resources by a counter. In this case, a task must be created with a region of the task where the resources are to be exclusively accessed separated by using the wai_sem and sig_sem service calls. Usually, the semaphore counter is used so to correspond with the number of resources that can be used.

The semaphores are controlled by the service calls listed in Table 4.5.

Table 4.5 Service Calls for Semaphore Control

Service Call Name	Function
sig_sem, isig_sem	Releases a resource
wai_sem	Gets a resource
pol_sem, ipol_sem	Gets a resource (polling)
twai_sem	Gets a resource (with timeout)
ref_sem, iref_sem	Refers to the semaphore status

A semaphore is created by initially defining it at system configuration.

Figure 4.3 shows an example of using a semaphore.

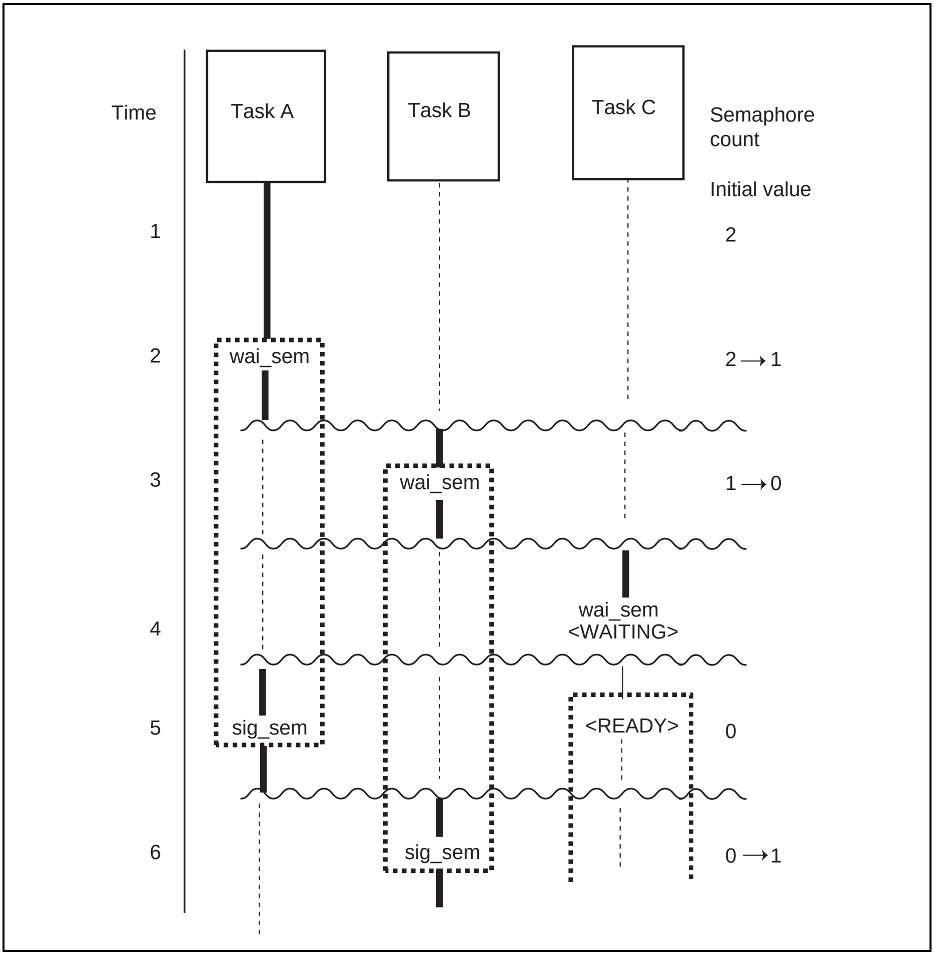


Figure 4.3 Example of Using Semaphore

Description:

Bold lines represent executed processing, and dotted lines represent the region where tasks can exclusively access resources. The following describes the semaphore operation with respect to time.

1. A semaphore with the initial value of 2 as the semaphore count is initially defined.
2. Task A issues wai_sem and gets a semaphore, decrementing the semaphore count by 1 (semaphore counter = 1). Task A continues execution.
3. Task B issues wai_sem.
4. Task C issues wai_sem, but cannot get a semaphore because the semaphore counter is 0, and it enters the WAITING state.
5. Task A releases a semaphore by issuing sig_sem. The released semaphore is allocated to task C, and task C is released from the WAITING state.
6. Task B releases a semaphore by issuing sig_sem. There is no task waiting for a semaphore, and so the semaphore counter is incremented by 1.

4.8.2 Event Flag

An event flag is a bit-group corresponding to events. One event corresponds to one bit. More than one task can wait for a specified bit to be set in an event flag, that is, tasks can wait until the specified event occurs.

The event flags are controlled by the service calls listed in Table 4.6.

Table 4.6 Service Calls for Event Flag Control

Service Call Name	Function
set_flg, iset_flg	Sets an event flag
clr_flg, iclr_flg	Clears an event flag
wai_flg	Waits for event occurrence
pol_flg, ipol_flg	Waits for event occurrence (polling)
twai_flg	Waits for event occurrence (with timeout)
ref_flg, iref_flg	Refers to the event flag status

An event flag is created by initially defining it at system configuration.

Figure 4.4 shows an example of using an event flag.

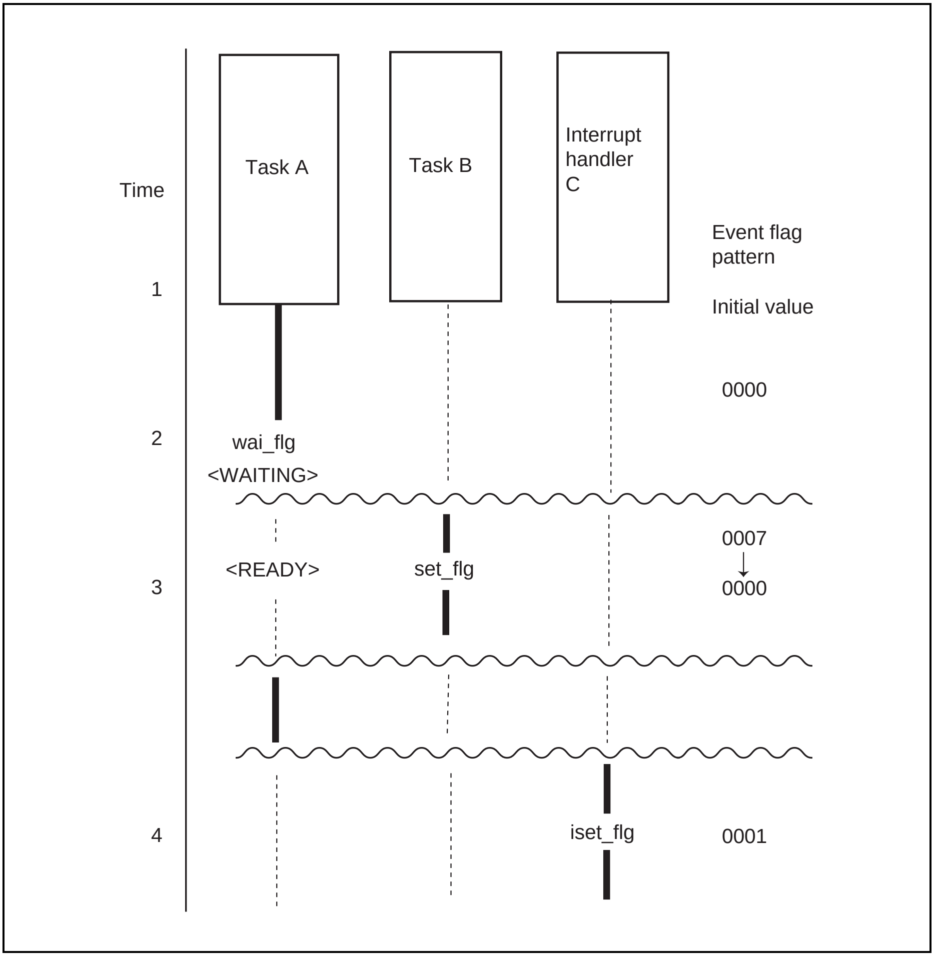


Figure 4.4 Example of Using Event Flag

Description:

Bold lines represent executed processing. The following describes event flag operation with respect to time.

1. An event flag that has the TA_CLR attribute (clear flag pattern to 0 when the WAITING state is released) is initially defined.
2. Task A issues wai_flg (waiting pattern = 3, AND wait) to wait for an event.
3. Task B issues set_flg (set pattern = 7). Since all bits that task A was waiting for have been set, task A is released from the WAITING state. In addition, since the TA_CLR attribute has been specified, the event flag is cleared to 0.
4. Interrupt handler C sets the event flag by issuing iset_flg (set pattern = 1). In this case there is no task waiting for an event, and the event flag is ORed with the pattern specified by iset_flg.

4.8.3 Data Queue

Data queues are used to send or receive 1-word data (4-byte data) between tasks. High-speed data transfer can be achieved using data queues, as communication using a data queue copies the 1-word data itself. In addition, pointers can also be specified as data.

The data queues are controlled by the service calls listed in Table 4.7.

Table 4.7 Service Calls for Data Queue Control

Service Call Name	Function
snd_dtq	Sends data to a data queue
psnd_dtq, ipsnd_dtq	Sends data to a data queue (polling)
tsnd_dtq	Sends data to a data queue (with timeout)
fsnd_dtq, ifsnd_dtq	Forcibly sends data to a data queue
rcv_dtq	Receives data from a data queue
prcv_dtq	Receives data from a data queue (polling)
trcv_dtq	Receives data from a data queue (with timeout)
ref_dtq, iref_dtq	Refers to the data queue status

A data queue is created by initially defining it at system configuration.

Figure 4.5 shows an example of using a data queue.

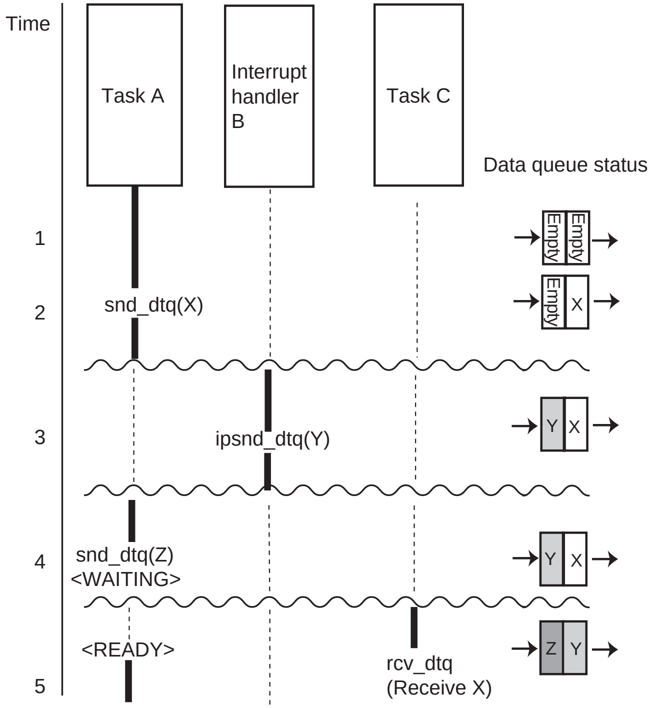


Figure 4.5 Example of Using Data Queue

Description:

Bold lines represent executed processing. The following describes the data queue operation with respect to time.

1. A data queue with a size of two words is initially defined.
2. Task A sends data X by issuing snd_dtq. Data X is copied to the data queue and task A continues execution.
3. Interrupt handler B sends data Y by issuing ipsnd_dtq.
4. Task A attempts to send data Z. At this time, since there is no empty entry in the data queue, task A enters the WAITING state.
5. Task C receives data from a data queue by issuing rcv_dtq. Task C gets data X, which is initially copied. At this time, since one entry in the data queue is released, data Z, which task A has attempted to send, is copied to a data queue, and task A is released from the WAITING state.

4.8.4 Mailbox

Mailboxes are used to send or receive message data between tasks. Since the communication using a mailbox sends and receives the message start address, it is fast regardless of the message size.

The mailboxes are controlled by the service calls listed in Table 4.8.

Table 4.8 Service Calls for Mailbox Control

Service Call Name	Function
snd_mbx, isnd_mbx	Sends a message to a mailbox
rcv_mbx	Receives a message from a mailbox
prcv_mbx, iprcv_mbx	Receives a message from a mailbox (polling)
trcv_mbx	Receives a message from a mailbox (with timeout)
ref_mbx, iref_mbx	Refers to the mailbox status

A mailbox is created by initially defining it at system configuration.

Figure 4.6 shows an example of using a mailbox.

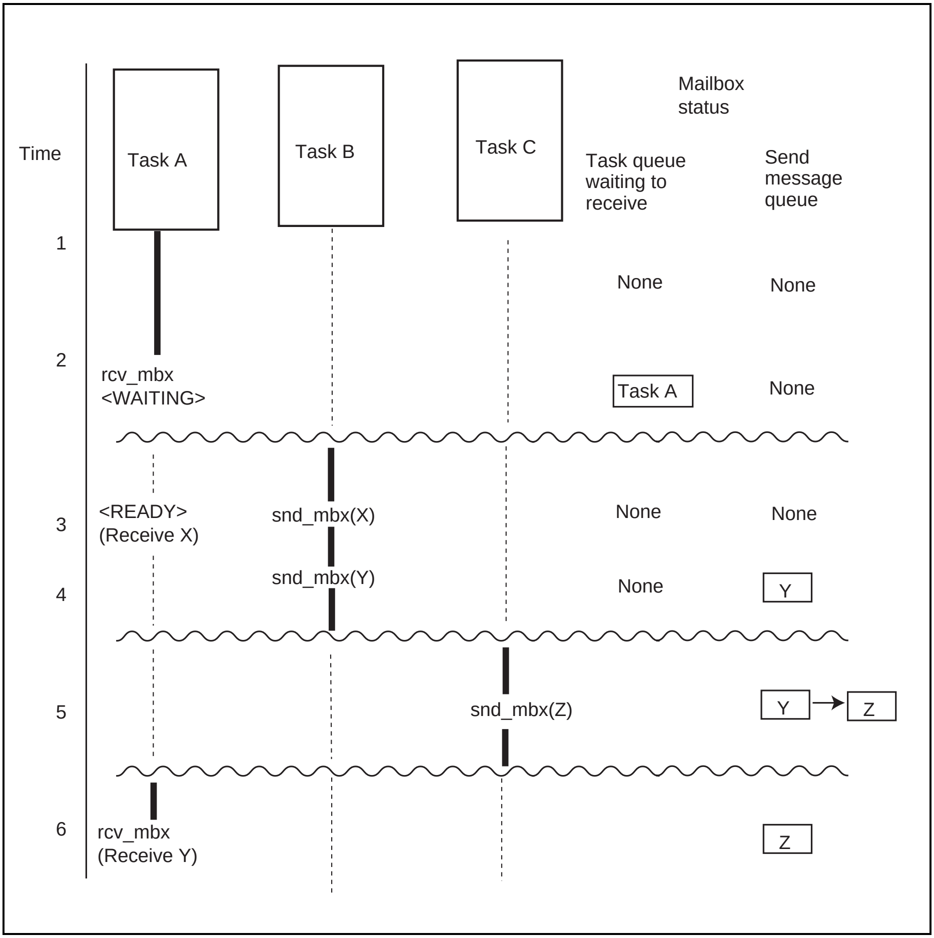


Figure 4.6 Example of Using Mailbox

Description:

Bold lines represent executed processing. The following describes the mailbox operation with respect to time.

1. A mailbox that has the TA_TFIFO attribute (tasks waiting to receive are queued in FIFO) and the TA_MFIFO attribute (sent messages are queued in FIFO) is initially defined.
2. Task A attempts to receive a message by using rcv_mbx. Since no message is stored in the mailbox, task A enters the WAITING state.
3. Task B sends message X to the mailbox using snd_mbx, and stores a message in the mailbox. At this time, task A is released from the WAITING state, and task A receives the address of message X.
4. Task B sends message Y to the mailbox using snd_mbx.
   At this time, since no tasks are waiting for a message, message Y is stored in a message queue.
5. Task C sends message Z to the mailbox using snd_mbx. In this case, message Z is also stored in a message queue (FIFO) because the TA_MFIFO attribute has been specified.
6. Task A issues rcv_mbx. At this time, task A receives the address of message Y, which is placed at the top of the message queue.

4.8.5 Mutex

The mutex is used to achieve exclusive control by providing a priority ceiling protocol to avoid priority inversion. In this protocol, the task that acquires a mutex is executed at a priority equal to the ceiling priority specified in the mutex.

The mutex is controlled by the service calls listed in Table 4.9.

Table 4.9 Service Calls for Mutex Control

Service Call Name	Function
loc_mtx	Locks a mutex
ploc_mtx	Locks a mutex (polling)
tloc_mtx	Locks a mutex (with timeout)
unl_mtx	Unlocks a mutex
ref_mtx	Refers to the mutex status

A mutex is created by initially defining it at system configuration.

Figure 4.7 shows an example of using a mutex.

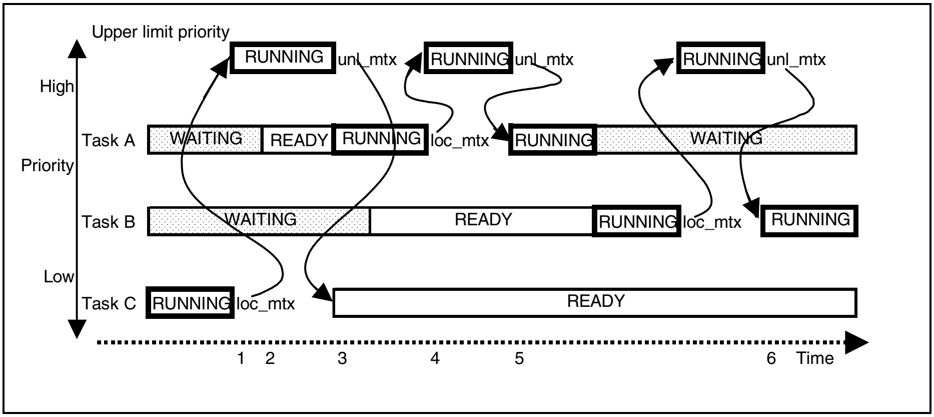


Figure 4.7 Example of Using Mutex

Description:

In Figure 4.7, the priority of tasks A, B, and C are defined as task A highest and task C lowest. Note that the ceiling priority specified by the mutex is specified as higher than the priority of the task that locks the mutex. The following describes the mutex operation with respect to time.

1. Task C locks a mutex by issuing loc_mtx. At this time, the priority of task C is pushed up to the ceiling priority specified by the mutex.
2. Task A enters the READY state while task C is executed at a priority equal to the ceiling priority specified by the mutex. Although the priority of task A is higher than that of task C at initial specification, task C now locks a mutex to be executed at the ceiling priority which is higher than task A and task A cannot enter the RUNNING state. In other words, while task C locks a mutex, task C continues execution even if the initial task priority of task A is higher than task C.
3. Task C unlocks the mutex by issuing unl_mtx. At this time, the priority of task C returns to the initial priority and task A enters the RUNNING state.
4. Task A issues loc_mtx to push its priority up to the ceiling priority specified in the mutex.
5. Task A issues unl_mtx to return its priority to the initial priority.
6. Task B issues loc_mtx to push its priority up to the ceiling priority specified in the mutex.
7. Task B issues unl_mtx to return its priority to the initial priority.

4.9 Memory Pool

The memory pools allow memory space to be used efficiently. Memory pools include fixed-size memory pools and variable-size memory pools.

4.9.1 Fixed-Size Memory Pool

A task can acquire and use a fixed-size memory block, whose fixed-size is determined for each memory pool, from the fixed-size memory pool. The memory block size is specified when the memory pool is initially defined.

The fixed-size memory pools are controlled by the service calls listed in Table 4.10.

Table 4.10 Service Calls for Fixed-Size Memory Pool Control

Service Call Name	Function
get_mpf	Gets a fixed-size memory block
pget_mpf, ipget_mpf	Gets a fixed-size memory block (polling)
tget_mpf	Gets a fixed-size memory block (with timeout)
rel_mpf	Releases a fixed-size memory block
ref_mpf, iref_mpf	Refers to the fixed-size memory pool status

A fixed-size memory pool is created by initially defining it at system configuration.

Figure 4.8 shows an example of using a fixed-size memory pool.

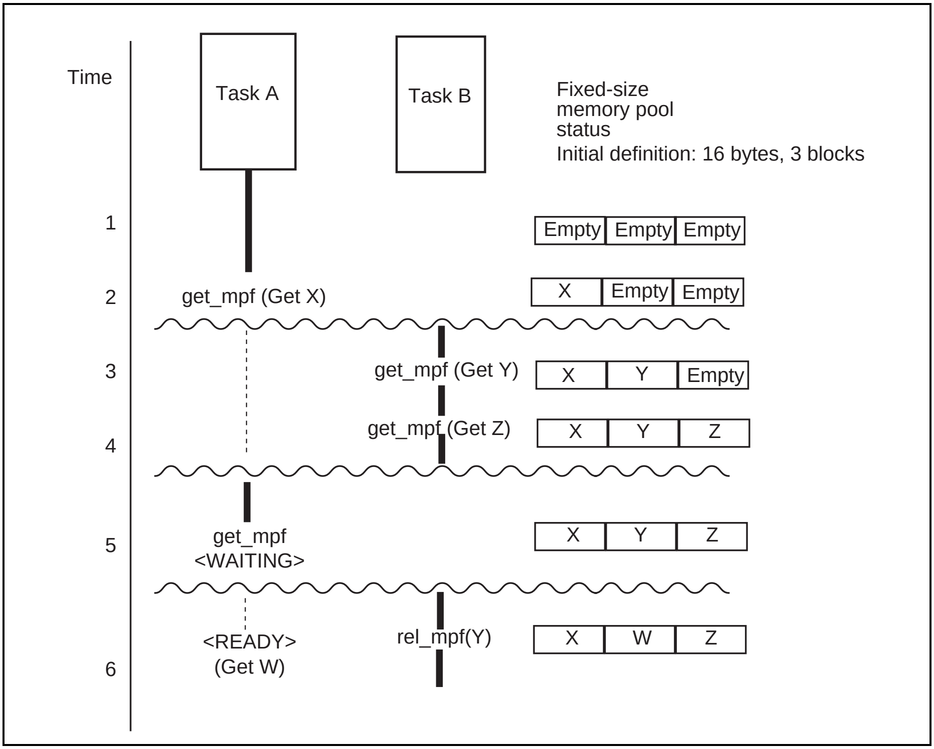


Figure 4.8 Example of Using Fixed-Size Memory Pool

Description:

Bold lines represent executed process. The following describes the fixed-size memory pool operation with respect to time.

1. A fixed-size memory pool that has three 16-byte memory blocks is initially defined.
2. Task A gets block X by issuing get_mpf.
3. Task B gets block Y by issuing get_mpf.
4. Task B gets block Z by issuing get_mpf.
5. Task A attempts to get a block by issuing get_mpf. At this time, no memory blocks are available and task A enters the WAITING state.
6. Task B returns block Y by issuing rel_mpf. At this time, task A is released from the WAITING state and the released block W is allocated to task A.

4.9.2 Variable-Size Memory Pool

A task can acquire a variable-size memory block from the variable-size memory pool.

Although the variable-size memory pool is more flexible than fixed-size memory pool, the overhead is large when acquiring or releasing variable-size memory block. Also, in a variable-size memory pool, there is a possibility of fragmentation. This means that even if there is enough total space to acquire a variable-size memory block, it cannot be acquired if the area is not contiguous.

The variable-size memory pools are controlled by the service calls listed in Table 4.11.

Table 4.11 Service Calls for Variable-Size Memory Pool Control

Service Call Name	Function
get_mpl	Gets a variable-size memory block
pget_mpl, ipget_mpl	Gets a variable-size memory block (polling)
tget_mpl	Gets a variable-size memory block (with timeout)
rel_mpl	Releases a variable-size memory block
ref_mpl, iref_mpl	Refers to the variable-size memory pool status

A variable-size memory pool is created by initially defining it at system configuration.

Figure 4.9 shows an example of using a variable-size memory pool.

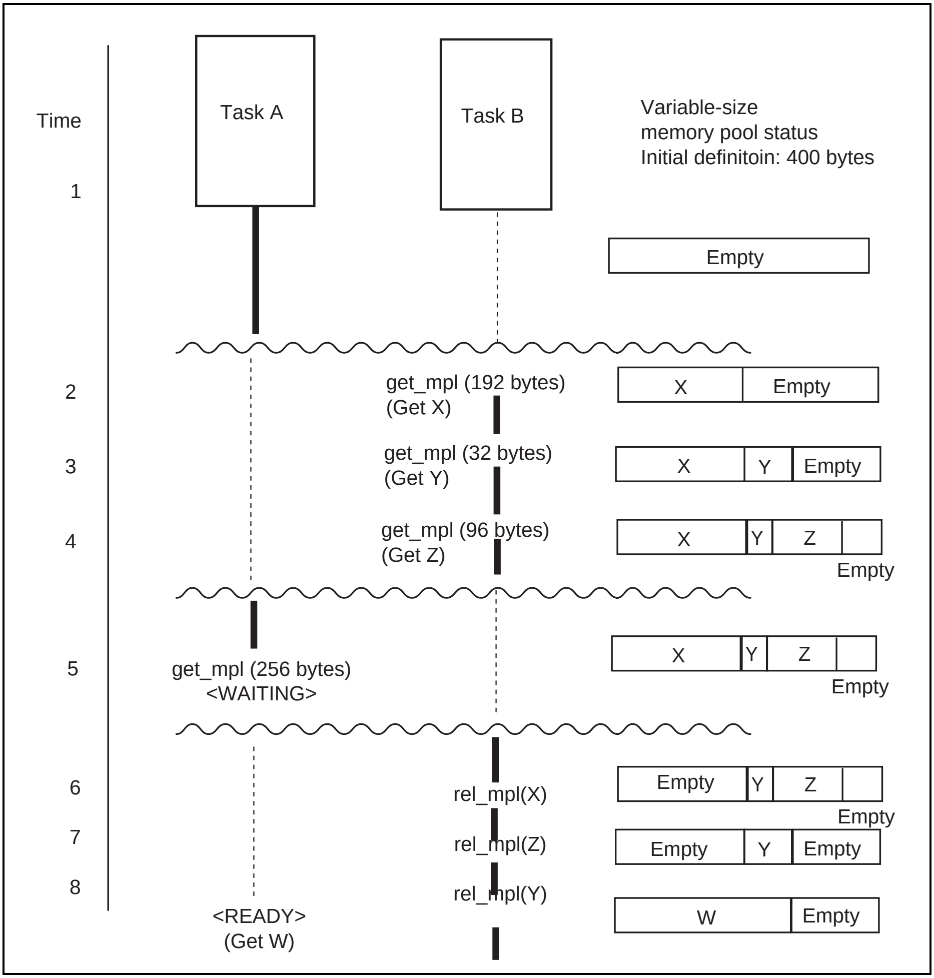


Figure 4.9 Example of Using Variable-Size Memory Pool

Description:

Bold lines represent executed process. The following describes the variable-size memory pool operation with respect to time.

1. A 400-byte variable-size memory pool is initially defined.
2. Task B acquires 192-byte memory block X by issuing get_mpl. At this time, the kernel uses 16 bytes in the memory pool. This is not indicated in Figure 4.9.

3. Task B also acquires 32-byte memory block Y by issuing get_mpl.
4. Task B also acquires 96-byte memory block Z by issuing get_mpl.
5. Task A attempts to acquire a 256-byte memory block by issuing get_mpl. However, the available memory block is insufficient to assign a 256-byte memory block to task A, so task A enters the WAITING state.
6. Task B returns 192-byte memory block X by issuing rel_mpl. At this time, since there is not 256 bytes of contiguous memory in the memory pool, task A remains in the WAITING state.
7. Task B returns 96-byte memory block Z by issuing rel_mpl. At this time, the total available memory blocks is more than 256 bytes, however there is not 256 bytes of contiguous memory in the memory pool, so task A remains the WAITING state.
8. Task B returns 32-byte memory block Y by issuing rel_mpl. At this time, since there is more than 256 bytes of contiguous memory in the memory pool, task A is released from the WAITING state and 256-byte memory block W is assigned to task A.

4.10 Time Management

The kernel provides the following functions related to time management:

- Reference to and setting of system clock
- Time event handler (cyclic handler) execution control
- Task execution control such as timeout

The kernel uses a counter called the system clock to perform the above functions. The unit of a time parameter used in the service calls is 1 ms. For the system clock cycle, a value other than 1 ms can be set by specifying the numerator of the time tick cycle (TIC_NUME) and the denominator of the time tick cycle (TIC_DENO) at system configuration.

To use the time management function, several preparations are required. For details, refer to Section 6.8, Timer Driver.

The system clock is controlled by the service calls listed in Table 4.12. In the HI1000/4, the system provides an original function to update the system clock.

Table 4.12 Service Calls for System Clock Control

Service Call Name	Function
set_tim, iset_tim	Sets system clock
get_tim, iget_tim	Refers to system clock

4.10.1 Cyclic Handler

The cyclic handler is a time event handler that is initiated at every initiation cycle after the specified initiation phase has elapsed.

Cyclic handlers are controlled by the service calls listed in Table 4.13.

Table 4.13 Service Calls for Cyclic Handler Control

Service Call Name	Function
sta_cyc, ista_cyc	Starts cyclic handler operation
stp_cyc, istp_cyc	Stops cyclic handler operation
ref_cyc, iref_cyc	Refers to the cyclic handler status

A cyclic handler is created by initially defining it at system configuration.

There are two methods to initiate the cyclic handler; storing the initiation phase, and not storing the initiation phase. In storing the initiation phase, the cyclic handler is initiated based on the timing when the cyclic handler is initially defined. In not storing the initiation phase, the cyclic handler is initiated based on the timing when operation of the cyclic handler is started. Figure 4.10 shows an example of using a cyclic handler.

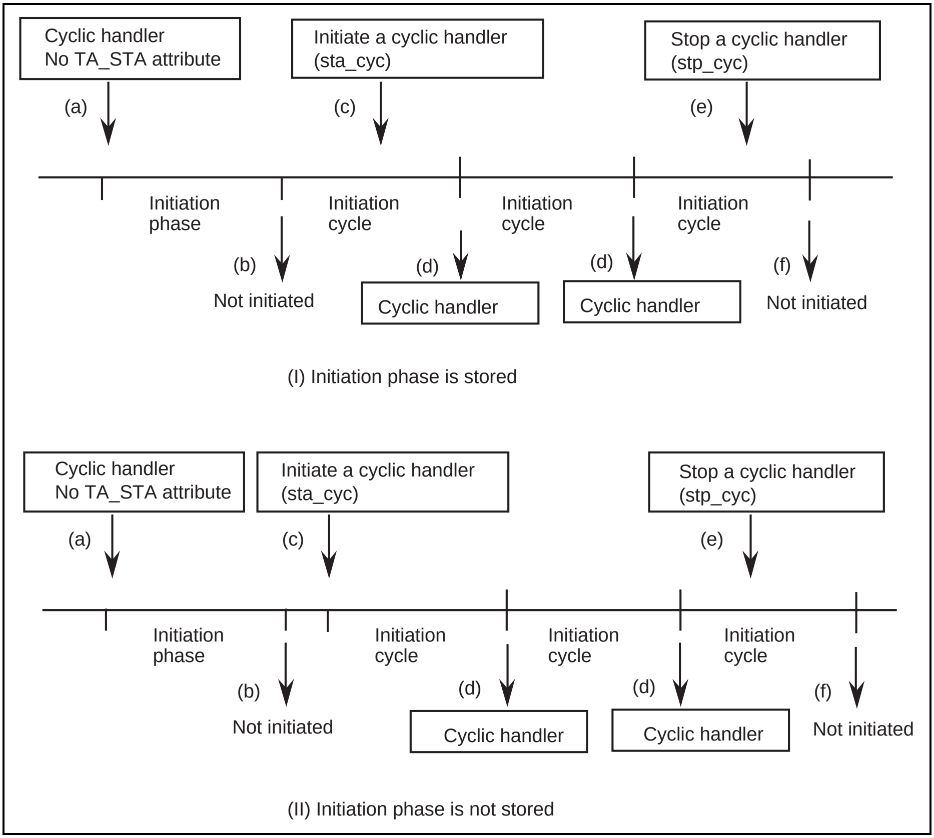


Figure 4.10 Example of Using Cyclic Handler

Description:

(a) A cyclic handler (without TA_STA attribute specification) is initially defined.

(b) The cyclic handler is not initiated after the initiation phase or cycle time has passed since the cyclic handler operation has not been initiated.

(c) The cyclic handler operation is initiated by issuing sta_cyc.

(d) When the initiation phase is stored as shown in (I) in Figure 4.10, the cyclic handler is initiated based on the initiation cycle after the cyclic handler has been initially defined. When the initiation phase is not stored as shown in (II) in Figure 4.10, the cyclic handler is initiated based on the initiation cycle after the sta_cyc service call has been issued.

(e) The cyclic handler is terminated by issuing the stp_cyc service call.

(f) The cyclic handler is not initiated after cycle time has passed since the cyclic handler operation has been terminated.

4.10.2 Notes on Time Management

(1) Drawbacks due to the repeated use

The following is performed by the kernel when a timer interrupt occurs:

a. System clock is updated.

b. All cyclic handlers that reached the cycle time are initiated and executed.

c. Timeout processing is performed after service calls such as tslp_tsk with the timeout function, have been issued and the specified timeout time has elapsed.

The processes from a to c are performed with the timer interrupt level masked.

Among these processes, b and c may overlap for multiple tasks and handlers. In that case, the processing time of the kernel becomes very long and results in the following defects.

— Delay of the response to interrupts

— Delay of system clocks

To avoid these problems, the following steps must be taken:

— The time event handler processing time must be as short as possible.

— The time event handler cycle and the timeout value specified by a service call with timeout must be set to values as large as possible. As a far-fetched example, if the cycle time of a cyclic handler is 1 ms and the handler's processing time takes longer than 1 ms, that cyclic handler will be executed forever; and the system will hang.

(2) Time management method

Time parameters specified in service calls are specified using relative time. For example, if the relative time is specified as 1 ms, the corresponding event processing is initiated at the time tick when 1 ms has elapsed since the service call was issued. If the relative time is specified as 0 ms, the corresponding event processing is initiated at the first time tick after the service call is issued.

The system clock can be changed by issuing the set_tim service call. Note, however, that the system clock for an event to which a time management request is issued before the set_tim service call is not affected.

(3) Time tick cycle setting

In a system where the numerator of the time tick cycle (TIC_NUME) is set to a value other than 1, if a value equal to or less than TIC_NUME is specified as the cycle time or initiation phase of the cyclic handler, a carry occurs in the initiation time of the cyclic handler. This causes the cyclic handler to be initiated more than once in the same time tick processing, and the drawbacks listed in (1) may occur.

As a specific example, if the cycle time of a cyclic handler is set to 5 in a system where the numerator of the time tick cycle (TIC_NUME) is set to 10, the same cyclic handler is initiated twice in a single time tick processing, so the drawbacks listed in (1) may occur.

Accordingly, it is recommended to set a multiple of the numerator of the time tick cycle (TIC_NUME) for the timeout time, cycle time, and initiation phase.

4.11 System State Management

The service calls listed in Table 4.14 can be used to control the system state.

Table 4.14 Service Calls for System State Management

Service Call Name	Function
rot_rdq, irot_rdq	Rotates ready queue
get_tid, iget_tid	Refers to the task ID
loc_cpu, iloc_cpu	Enters CPU-locked state
unl_cpu, iunl_cpu	Releases CPU-locked state
dis_dsp	Disables dispatch
ena_dsp	Enables dispatch
sns_ctx	Refers to the context
sns_loc	Refers to the CPU-locked state
sns_dsp	Refers to the dispatch-disabled state
sns_dpn	Refers to the dispatch-enabled state
vsta_knl, ivsta_knl	Initiates the kernel
vsys_dwn, ivsys_dwn	System down
ivbgn_int	Acquires trace information on interrupt handler initiation
ivend_int	Acquires trace information on interrupt handler termination

4.11.1 System Down

When an error occurs, control is passed to the system down routine.

Errors can be classified into the following three types:

1. When the vsys_dwn or ivsys_dwn service call was issued from the application program.
2. When an error was detected inside the kernel.
3. When an undefined interrupt or exception occurred.

The user must create the system down routine. For details, refer to Section 6.10, System Down Routine.

4.11.2 Service Call Trace Function

The service call trace function is used to acquire the history of the service calls that are issued during system execution and store it in the trace buffer. Basically, for a single service call, information of two service calls, the one issued first and the one issued to return, are acquired. This information is called an event.

To use the service call trace function, the service call trace function must be enabled by the configurator at system configuration. In addition, the trace function must be selected and the trace count must be specified appropriately according to the user environment.

When the trace function is enabled, after the system is initiated, all events starting from the initialization routine are acquired. The trace data is stored in the trace buffer when the normal-version or simple-version target trace is selected. The trace buffer has a ring-buffer structure in which old information is sequentially overwritten with new information.

If the ivbgn_int and ivend_int service calls are written at the beginning and end of the interrupt handler, respectively, the trace information on interrupt handler initiation and termination can also be acquired.

History Information Storage Area: History information can be stored in either the target memory or debugger (such as the E6000H emulator or the simulator). The former is called the target trace and the latter is called the tool trace. Though the tool trace limits the environment (E6000H emulator or simulator), it allows the service call trace area to be hardly required in the target system. For the environment in which the tool trace is available, refer to the manual of the debugging extension or the online help.

Preparation for Service Call Trace Function: The service call trace function is specified in the debugging function view of the configurator. The service call trace function is classified into the following four types.

- Target trace

  All service call trace information (attribute, task ID, event information, parameter, and caller's EXR, CCR, and PC) is stored in the trace buffer area allocated in the system.
- Tool trace

  All service call trace information (attribute, task ID, event information, parameter, and caller's EXR, CCR, and PC) is stored in the trace memory in the tool (simulator/debugger or emulator).
- Target trace (simple version)

  Minimal service call trace information (attribute, task ID, and event information) is stored in the trace buffer area allocated in the system.
- Tool trace (simple version)

  Minimal service call trace information (attribute, task ID, and event information) is stored in the trace memory in the tool (simulator/debugger or emulator).

Notes on Service Call Trace Function: The following must be noted when using the service call trace function:

a. Degradation of performance

When the service call trace function is used, the performance of the kernel is degraded a little.

b. Service call information not traced

The following service calls cannot be traced.

— vsta_knl, ivsta_knl

— vsys_dwn, ivsys_dwn

Service calls for non-task context such as ixxx_yyy are all acquired as service calls for task context such as xxx_yyy.

4.12 Interrupt Management and System Configuration Management

In this kernel, interrupts and exceptions are classified as follows:

- Reset: CPU reset. A program executed at a reset is called the CPU initialization routine.
- Interrupt: An interrupt is generated from external interrupt pins and peripheral modules. When an interrupt occurs, an interrupt handler is executed.
- CPU exception: An exception generated by a general illegal instruction. A CPU exception also includes a trap generated by a TRAPA instruction. When a CPU exception occurs, a CPU exception handler is executed.

If an exception or interrupt occurs, a CPU exception handler or interrupt handler defined in the vector table is directly initiated.

Interrupts and system configuration are controlled by the service calls listed in Table 4.15 and Table 4.16, respectively.

Table 4.15 Service Calls for Interrupt Control

Service Call Name	Function
chg_ims, ichg_ims	Changes interrupt mask
get_ims, iget_ims	Refers to interrupt mask

Table 4.16 Service Calls for System Configuration Control

Service Call Name	Function
ref_ver, iref_ver	Refers to version information

Interrupt handlers and CPU exception handlers (including the CPU exception handler for the TRAPA instruction) are defined in the vector table at system configuration. When an undefined interrupt or exception occurs, the system down routine will be initiated.

4.12.1 CPU Reset and Kernel Initiation

The procedure to reset the CPU and initiate the kernel is usually as shown in Figure 4.11.

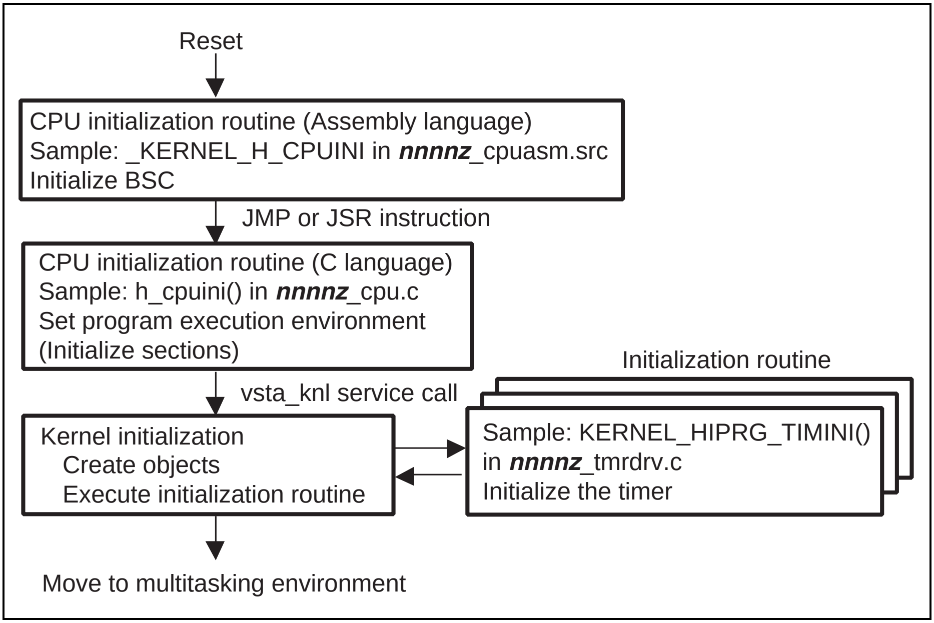

Reset
CPU initialization routine (Assembly language)
Sample: _KERNEL_H_CPUINI in ***nnnnz***_cpuasm.src
Initialize BSC
JMP or JSR instruction
CPU initialization routine (C language)
Sample: h_cpuini() in ***nnnnz***_cpu.c
Set program execution environment
(Initialize sections)
Initialization routine
vsta_knl service call
Kernel initialization
Create objects
Execute initialization routine
Sample: KERNEL_HIPRG_TIMINI()
in ***nnnnz***_tmrdrv.c
Initialize the timer
Move to multitasking environment

Figure 4.11 Flowchart from CPU Reset to Kernel Initiation

The CPU initialization routine carries out processing needed for the entire software, including the kernel, to operate.

The CPU initialization routine should set the bus state controller (BSC) so that programs can correctly access memory. A C-language program accesses stacks; therefore, the stacks must be ready to be accessed before the C-language program is executed. The CPU initialization routine should also set the C program execution environment, such as initializing sections. For details, refer to the H8S, H8/300 Series C/C++ Compiler User's Manual.

Then, vsta_knl or ivsta_knl is called to initiate the kernel. When the kernel is initiated, control does not return to the caller of vsta_knl and ivsta_knl.

The following is performed by the vsta_knl or ivsta_knl service call.

a. The interrupt mask level is set to the kernel interrupt mask level.
b. The kernel work area is initialized.
c. The initial defined objects specified in the configurator are created.
d. The initialization routine specified by the configurator is called.

e. The multitasking environment is entered.

For details on creating a CPU initialization routine, refer to Section 6.9, CPU Initialization Routine.

4.12.2 Interrupts

An interrupt handler is executed in the non-task context state. Tasks are scheduled after the interrupt handler has completed execution; tasks are not scheduled even when a task with high priority is in the READY state due to the service call issued while the interrupt handler was being executed.

Change of Interrupt Mask Level: The interrupt mask level can be changed in one of the following ways.

(a) chg_ims service call

(b) Directly change the interrupt mask bits in the CCR or EXR register (C compiler intrinsic functions: set_imask_ccr( ), set_ccr( ), set_imask_exr( ), and set_exr( )).

To change the interrupt mask level from 0 to a value other than 0, and to change it from a value other than 0 to 0, the method of (a) must be used. In other cases, the method of either (a) or (b) can be used.

When the interrupt mask level is first changed from 0 to x with the method of (a), and then changed from x to another value with the method of (b), the interrupt mask level must be returned to x before it is returned to 0 with the method of (a).

Note that when the interrupt handler is executing, the interrupt mask level must not be changed to a level lower than the current level.

Kernel Interrupt Mask Level: The kernel interrupt mask level (CFG_KNLMSKLVL) that specifies the interrupt level to mask interrupts during kernel execution is specified at system configuration. Though interrupts having an interrupt level higher than the kernel interrupt mask level are immediately accepted even during kernel execution, these interrupt handlers are not allowed to issue a service call.

When the interrupt mask level is set to a level higher than the kernel interrupt mask level, service calls cannot be issued, except when modifying the level of the interrupt mask to be equal to or lower than the kernel interrupt mask level by using the chg_ims service call.

4.12.3 CPU Exception

The CPU exception handler (including the TRAPA instruction exception) is executed in the non-task context state. The CPU exception handler uses the same stack as that used by the exception source. Since the CPU exception handler is in the non-task context state and its priority is higher than that of the dispatcher, task switching cannot be performed during the CPU exception processing.

Note that the CPU exception handler can call only the following service calls:

- sns_ctx
- sns_loc
- sns_dsp
- sns_dpn
- iget_tid
- ivsta_knl
- ivsys_dwn

Section 5 Service Calls

5.1 Overview

Service calls are classified as shown in Table 5.1.

Table 5.1 Service Call Classification

Classification	Description
Task management function	Initiates and terminates tasks
Task synchronization function	Suspends and resumes task execution
Synchronization and communication function	Manages semaphores, event flags, data queues, and mailboxes
Extended synchronization and communication function	Manages mutexes
Memory pool management function	Allocates memory dynamically
Time management function	Notifies the time to the kernel, sets and references the system clock, and manages cyclic handlers
System status management function	Shifts to the CPU-locked state or dispatch-disabled state
Interrupt management function	Changes and references the interrupt mask
System configuration management function	References the version information

5.2 Service Call Interface

Service calls can be called from programs written in C or assembly language. This section describes how to call service calls.

5.2.1 Header File

Certain header files must be included in order to use service calls. For details on the header file, refer to Section 6.1, Header Files.

5.2.2 C Language API

All service calls are described in the following C language function call format.

```
ercd = act_tsk(1);
```

The data types and sizes of the parameters are shown below. These are defined in the C language header file.

```
typedef char           B;        /* 8-bit signed integer                            */
typedef short          H;        /* 16-bit signed integer                           */
typedef long           W;        /* 32-bit signed integer                           */
typedef unsigned char  UB;       /* 8-bit unsigned integer                          */
typedef unsigned short UH;       /* 16-bit unsigned integer                         */
typedef unsigned long  UW;       /* 32-bit unsigned integer                         */
typedef char           VB;       /* 8-bit value of void type                        */
typedef short          VH;       /* 16-bit value of void type                       */
typedef long           VW;       /* 32-bit value of void type                       */
typedef void           *VP;      /* Pointer to void type                            */
typedef void           (*FP)();  /* Program initiation address                      */
typedef H              INT;      /* 16-bit signed integer                           */
typedef UH             UINT;     /* 16-bit unsigned integer                         */
typedef H              BOOL;     /* True/false value (TRUE or FALSE)                */
typedef H              FN;       /* Function code (signed integer)                  */
typedef H              ER;       /* Error code (signed integer)                     */
typedef H              ID;       /* Object ID number (signed integer)               */
typedef UH             ATR;      /* Object attribute (unsigned integer)             */
typedef UH             STAT;     /* Object state (unsigned integer)                 */
typedef UH             MODE;     /* Service call operating mode (unsigned integer) */
typedef H              PRI;      /* Priority (signed integer)                       */
typedef UW             SIZE;     /* Memory area size (unsigned integer)             */
typedef W              TMO;      /* Timeout specification (signed integer)          */
typedef UW             RELTIM;   /* Relative time (unsigned integer)                */
typedef W              VP_INT;   /* Pointer to void type or signed integer          */
typedef H              ER_BOOL;  /* Error code or true/false value                  */
typedef H              ER_ID;    /* Error code or ID number                         */
typedef H              ER_UINT;  /* Error code or unsigned integer                  */
typedef UH             FLGPTN;   /* Bit pattern of event flag (unsigned integer)    */
typedef UH             INHNO;    /* Interrupt handler number                        */
typedef UH             IMASK;    /* Interrupt mask level                            */
typedef UH             EXCNO;    /* CPU exception handler number                    */
```

5.2.3 Assembly Language API

In most cases, a service call can be called from an assembly-language program, as shown in Figure 5.1.

For details on the parameters when calling a service call from an assembly-language program, refer to the description of each service call.

```
    .INCLUDE "kernel.inc"          ← (a)
; .....
    .import  _act_tsk
;
TSKID: .assign 1
;
Task_a:
    MOV.W  #TSKID,R0               ← (b)
    JSR    @_act_tsk               ← (c)
; .....                            ← (d)
    .end
```

Figure 5.1 Example of Service Call from an Assembly-Language Program

(a) Standard header file kernel.inc is included.

(b) Parameters are specified.

(c) Service call function is called.

(d) After service call processing, execution returns to the address saved in the stack by jsr except for the service calls that do not return to the calling program. In this example, execution returns to this location. When execution returns, normal termination (E_OK) or an error code is set in R0.

5.2.4 Number of Parameter Storage Registers

The number of registers used for storing parameters is assumed to be 3 (keyword _ _regparam3 setting) in the HI1000/4 service calls (functions). Therefore, the keyword _ _regparam3 should be specified in a service call. In the sample header files supplied, 3 is specified for the number of parameter storage registers in the prototype declaration for the service calls.

Normal system operation cannot be guaranteed when a service call is issued with the number of parameter storage registers not specified to 3 in the service call.

5.2.5 Register Contents Guaranteed after Calling Service Call

Some registers guarantee the contents after a service call is called but some do not. This rule follows the H8S, H8/300H C/C++ compiler. The details are shown in Table 5.2.

Table 5.2 Register Contents Guaranteed after Calling Service Call

Registers	Guarantee on Register Contents*
ER3 to ER6 and SBR (H8SX only)	The register contents will be guaranteed.
R0	Normal termination (E_OK) or an error code is set.
ER0 to ER2	The register contents will be guaranteed only when indicated as a return parameter clearly.

Note: These rules on the guaranteed register contents are applied only when the keyword _ _regparam3 has been specified.
For details on guaranteeing the register contents when creating tasks and handlers, refer to Section 6, Application Program Creation.

5.2.6 Return Value of Service Call and Error Code

For service calls that have return codes, a positive value or 0 (E_OK) indicates normal termination, and a negative value indicates an error code. However, for service calls that have a BOOL-type return value, this is not the case. The meaning of the return value at normal termination differs according to the service call; however, only E_OK is returned at normal termination for many service calls.

An error code consists of main error codes (lower 8 bits) and sub error codes (upper bits above lower 8 bits). The sub error code of this kernel is always set to –1.

The following macros are set to the standard header itron.h. The SERCD macro of this kernel always returns –1 except for normal termination.

- ER mercd = MERCD(ER ercd); returns the main error code from the error code
- ER sercd = SERCD(ER ercd); returns the sub error code from the error code

5.2.7 Parameters and Return Parameters

In a service call that has a return parameter, the return parameter is stored in the area specified by the parameter. In a service call whose first parameter is the return parameter, the return parameter is passed to the program through ER0. However, after the service call processing completes, the return parameter ER0 is overwritten because the error code is returned to R0. This does not need to be considered in a C language program. However, in an assembly-language program, the return parameter needs to be first saved in a register that guarantees an area to store the return parameter before a service call is issued, and then acquired from the area pointed to by the register where it is saved after the service call completes.

5.2.8 System State and Service Calls

Whether a service call can be called or not depends on the system state (refer to Section 4.4, System State).

(1) Task context and non-task context

Service calls can be classified as dedicated to task context, dedicated to the non-task context, and service calls that can be called from all contexts.

(a) Names of service calls dedicated to non-task context start with “i”.

(b) Names of service calls that can be called from all context start with “sns”.

(c) Service calls other than the above are dedicated to task context.

Normal system operation cannot be guaranteed when a service call is called from a context that differs from the above description. However, in special cases, for example when a service call that shifts to the WAITING state is called from non-task context, an E_CTX error is returned.

(2) Dispatch-disabled/-enabled state

All service calls can be called from the dispatch-enabled state. Some service calls that shift to WAITING state cannot be called from the dispatch-disabled state. When those service calls are called from the dispatch-disabled state, an E_CTX error is returned.

(3) CPU-locked/-unlocked state

All service calls can be called from the CPU-unlocked state. Service calls that can be called from the CPU-locked state are listed below. Normal system operation cannot be guaranteed when service calls other than these are called from the CPU-locked state (detection of an E_CTX error is omitted). When service calls that shift to the WAITING state are called, an E_CTX error is returned.

- ext_tsk
- loc_cpu, iloc_cpu
- unl_cpu, iunl_cpu
- sns_ctx
- sns_loc
- sns_dsp
- sns_dpn
- vsta_knl, ivsta_knl
- vsys_dwn, ivsys_dwn

(4) CPU exception handler

Service calls that can be called from the CPU exception handler are listed below. Normal system operation cannot be guaranteed when service calls other than these are called from the CPU exception handler (detection of an E_CTX error is omitted). When service calls that shift to the WAITING state are called, an E_CTX error is returned.

- sns_ctx
- sns_loc
- sns_dsp
- sns_dpn
- iget_tid

— ivsta_knl
— ivsys_dwn

(5) Before kernel initiation

The following service calls can be called even before kernel initiation (vsta_knl service call).

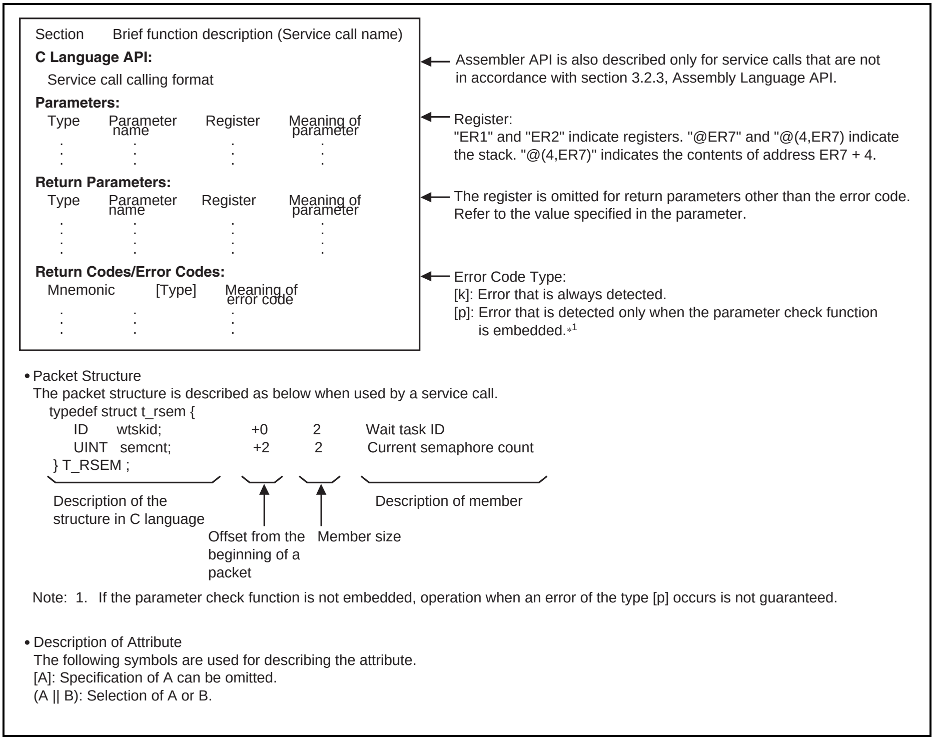
— vsta_knl, ivsta_knl
— vsys_dwn, ivsys_dwn

5.2.9 Service Calls not in the μITRON4.0 Specification

Service calls whose name start with “v” or “iv”, such as ivbgn_int, are service calls that are not in the μITRON4.0 specification.

5.3 Service Call Description Form

Service calls are described in details as shown in Figure 5.2 in this section.



Section Brief function description (Service call name)

C Language API:

Service call calling format

Parameters:

Type	Parameter name	Register	Meaning of parameter
:	:	:	:

Return Parameters:

Type	Parameter name	Register	Meaning of parameter
:	:	:	:

Return Codes/Error Codes:

Mnemonic	[Type]	Meaning of error code
:	:	:

Assembler API is also described only for service calls that are not in accordance with section 3.2.3, Assembly Language API.

Register:
"ER1" and "ER2" indicate registers. "@ER7" and "@(4,ER7) indicate the stack. "@(4,ER7)" indicates the contents of address ER7 + 4.

The register is omitted for return parameters other than the error code. Refer to the value specified in the parameter.

Error Code Type:
[k]: Error that is always detected.
[p]: Error that is detected only when the parameter check function is embedded.*[1]

- Packet Structure

The packet structure is described as below when used by a service call.

```
typedef struct t_rsem {
    ID     wtskid;          +0    2    Wait task ID
    UINT   semcnt;          +2    2    Current semaphore count
} T_RSEM ;
```

Description of the structure in C language

Offset from the beginning of a packet

Member size

Description of member

Note: 1. If the parameter check function is not embedded, operation when an error of the type [p] occurs is not guaranteed.

- Description of Attribute

The following symbols are used for describing the attribute.
[A]: Specification of A can be omitted.
(A || B): Selection of A or B.

Figure 5.2 Service Call Description Form

5.4 Task Management

Tasks are managed by the service calls listed in Table 5.3.

Table 5.3 Service Calls for Task Management

			System State*2
Service Call*1		**Description**	**T/N/E/D/U/L/C**
act_tsk	[S]	Initiates task	T/E/D/U
iact_tsk	[S]		N/E/D/U
can_act	[S]	Cancels task initiation request	T/E/D/U
sta_tsk		Initiates task (specifies start code)	T/E/D/U
ista_tsk			N/E/D/U
ext_tsk	[S]	Exits current task	T/E/D/U/L
ter_tsk	[S]	Forcibly terminates a task	T/E/D/U
chg_pri	[S]	Changes task priority	T/E/D/U
get_pri	[S]	Refers to task priority	T/E/D/U
ref_tsk		Refers to task state	T/E/D/U
iref_tsk			N/E/D/U
ref_tst		Refers to task state (simple version)	T/E/D/U
iref_tst			N/E/D/U

Notes: 1. [S]: Standard profile service calls

2. T: Can be called from task context
N: Can be called from non-task context
E: Can be called from dispatch-enabled state
D: Can be called from dispatch-disabled state
U: Can be called from CPU-unlocked state
L: Can be called from CPU-locked state
C: Can be called from CPU exception handler

Task management specifications are listed in Table 5.4.

Table 5.4 Task Management Specifications

Item	Description
Task ID	1 to CFG_MAXTSKID (255 max.)
Task priority	1 to CFG_MAXTSKPRI (31 max.)
Maximum count of task initiation request	255
Task attribute	TA_HLNG: The task is written in a high-level language TA_ASM: The task is written in assembly language
Task stack	Shared stack function is available
Shared-stack wait queue (when the shared stack function is used)	First-in first-out (FIFO) basis

5.4.1 Initiate Task (act_tsk, iact_tsk)

C-Language API:

```
ER ercd = act_tsk(ID tskid);
ER ercd = iact_tsk(ID tskid);
```

Parameters:

```
ID              tskid       R0    Task ID
```

Return Parameters:

```
ER              ercd        R0    Normal termination (E_OK) or error code
```

Error Codes:

```
E_ID          [p]  Invalid ID number (tskid ≤ 0, tskid > MAXTSKID, or
                   tskid = TSK_SELF(0) is specified in a non-task context)
E_NOEXS       [p]  Undefined (Task indicated by tskid does not exist)
E_QOVR        [k]  Queuing overflow (actcnt > H'ff)
E_CTX         [p]  Context error (Called from disabled system state)
```

Function:

Each service call initiates the task indicated by the parameter tskid. The initiated task makes a transition from the DORMANT state to the READY state.

By specifying tskid = TSK_SELF (0), the current task is specified.

Extended information of the task specified when the task is initially defined will be passed to the task as the parameter.

In using the shared stack function, if the stack of the task indicated by tskid is not being used by any task, the task indicated by tskid occupies the shared stack and shifts to the READY state. If the stack is being used by another task, the task indicated by tskid shifts to the shared-stack WAITING state and is placed in the shared-stack wait queue since the stack area cannot be used. The wait queue is managed on a first-in first-out (FIFO) basis.

When the task is not in the DORMANT state, up to 255 task initiation requests from the service calls act_tsk and iact_tsk can be stored.

5.4.2 Cancel Task Initiation Request (can_act)

C-Language API:

```
ER_UINT actcnt = can_act(ID tskid);
```

Parameters:

```
ID              tskid     R0   Task ID
```

Return Parameters:

```
ER_UINT         actcnt    R0   Number of cached initiation requests (positive
                               value or 0), or error code
```

Error Codes:

```
E_ID          [p]   Invalid ID number (tskid ≤ 0, tskid > CFG_MAXTSKID)
E_NOEXS       [p]   Undefined (Task indicated by tskid is not created)
E_CTX         [p]   Context error (Called from disabled system state)
```

Function:

The number of initiation requests queued for the task specified by tskid is determined, the result is returned as the return parameter, and at the same time the initiation requests are all cancelled.

By specifying tskid=TSK_SELF(0), the current task is specified.

A task in a DORMANT state can also be specified; in this case the return parameter is 0.

5.4.3 Start Task (Start Code Specified) (sta_tsk, ista_tsk)

C-Language API:

```
ER ercd = sta_tsk(ID tskid, VP_INT stacd);
ER ercd = ista_tsk(ID tskid, VP_INT stacd);
```

Parameters:

ID	tskid	R0	Task ID
VP_INT	stacd	ER1	Task start code

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (tskid ≤ 0, tskid > CFG_MAXTSKID)
E_NOEXS	[p]	Undefined (Task indicated by tskid does not exist)
E_OBJ	[k]	Object state error (The task specified by tskid is not in the DORMANT state or the current task is specified)
E_CTX	[p]	Context error (Called from disabled system state)

Function:

Each service call initiates the task indicated by the parameter tskid. The initiated task makes a transition from the DORMANT state to the READY state.

The task initiation code indicated by the parameter stacd will be passed to the initiated task as the parameter.

In using the shared stack function, if the stack of the task indicated by tskid is not being used by any task when the service calls sta_tsk and ista_tsk are called, the task indicated by tskid occupies the shared stack and shifts to the READY state. If the stack is being used by another task, the task indicated by tskid shifts to the shared-stack WAITING state and is placed in the shared-stack wait queue since the stack area cannot be used. The wait queue is managed on a first-in first-out (FIFO) basis.

5.4.4 Exit Current Task (ext_tsk)

C-Language API:

```
void ext_tsk( );
```

Parameters:

```
None
```

Return Parameters:

```
The service call ext_tsk does not return to the current task.
The service call ext_tsk may generate the following error, and in this case,
control is passed to the system down routine.
E_CTX        [p]   Context error (Called from disabled system state)
```

Function:

The service call ext_tsk exits the current task normally. After the execution of the service call ext_tsk, the current task makes a transition from the RUNNING state to the DORMANT state. When initiation request is queued, the service call ext_tsk exits the current task and then restarts the task. Note however that if there is a task in the shared-stack WAITING state, the task that was in the shared-stack WAITING state is released and the current task enters the shared-stack WAITING state.

In addition, if a task is locking a mutex, the locked mutex is unlocked. If there is a task waiting for that mutex to be unlocked, the WAITING state of the task waiting for unlock is cancelled, and that task makes a transition to the READY state.

The service call ext_tsk does not automatically release resources other than mutexes (such as semaphores and memory blocks) acquired before the task is exited. Therefore, the user must call service calls to release resources before exiting the task.

If the task that calls the service call ext_tsk shares the stack with other tasks, the task at the head of the stack wait queue is removed, the WAITING state is cancelled, and the task makes a transition to the READY state.

The service call ext_tsk can be called while task dispatch is disabled or the CPU is locked. After either of the service calls is called, the dispatch-disabled state or CPU-locked state is cancelled.

5.4.5 Terminate Task (ter_tsk)

C-Language API:

```
ER ercd = ter_tsk(ID tskid);
```

Parameters:

ID	tskid	R0	Task ID

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (tskid ≤ 0 or tskid > CFG_MAXTSKID)
E_NOEXS	[p]	Undefined (Task indicated by tskid does not exist)
E_OBJ	[k]	Object state error (Task indicated by tskid is in the DORMANT state)
E_ILUSE	[k]	Illegal use of service call (The current task is specified as the target task)
E_CTX	[p]	Context error (Called from disabled system state)

Function:

The service call ter_tsk forces a task specified by tskid to terminate. The terminated task enters the DORMANT state.

When the initiation request is queued, the target task enters the READY state.

If the task specified by tskid shares the stack with other tasks, the task at the head of the stack wait queue is removed and released from the WAITING state, and the task makes a transition to the READY state.

In addition, if a task is locking a mutex, the locked mutex is unlocked. If there is a task waiting for that mutex to be unlocked, the WAITING state of the task waiting for unlock is cancelled, and that task makes a transition to the READY state.

5.4.6 Change Task Priority (chg_pri)

C-Language API:

```
ER ercd = chg_pri(ID tskid, PRI tskpri);
```

Parameters:

ID	tskid	R0	Task ID
PRI	tskpri	E0	Base priority of task

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (tskpri < 0 or tskpri > CFG_MAXTSKPRI)
E_ID	[p]	Invalid ID number (tskpri < 0, tskid > CFG_MAXTSKID)
E_NOEXS	[p]	Undefined (Task specified by tskid does not exist)
E_ILUSE	[k]	Illegal use of service call (Ceiling priority is exceeded)
E_OBJ	[k]	Object state error (Task is in the DORMANT state)
E_CTX	[p]	Context error (Called from disabled system state)

Function:

Each service call changes the base task priority specified by the parameter tskid to the value specified by the parameter tskpri. The current task priority is also changed. By specifying tskid = TSK_SELF (0), the current task can also be specified.

Specifying tskpri = TPRI_INI (0) returns the task priority to the priority at task initiation that was specified when the task was initially defined.

A priority changed by the service calls is valid until the task is terminated or until the service calls are called again. When a task makes a transition to the DORMANT state, the task priority before termination becomes invalid and returns to the priority at task initiation that was specified at task definition.

If the task specified by tskid is in the WAITING state and TA_TPRI is specified for the object attribute, the wait queue can be changed by the service calls and as a result, the task at the head of the wait queue may be released from the WAITING state.

If the base priority specified in the parameter tskpri is higher than the ceiling priority of one of the mutexes when the target task locks or waits to lock the mutexes with the TA_CEILING attribute, E_ILUSE is returned.

5.4.7 Refer to Task Priority (get_pri)

C-Language API:

```
ER ercd = get_pri(ID tskid, PRI *p_tskpri);
```

Parameters:

ID	tskid	R0	Task ID
PRI	*p_tskpri	ER1	Pointer to the area where the current priority of the target task is to be returned

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code
PRI	*p_tskpri	--	Pointer to the area where the current priority of the target task is stored

Error Codes:

E_PAR	[p]	Parameter error (p_tskpri is 0)
E_ID	[p]	Invalid ID number (tskid < 0, tskid > CFG_MAXTSKID)
E_NOEXS	[p]	Undefined (Task specified by tskid does not exist)
E_OBJ	[k]	Object state error (Task is in the DORMANT state)
E_CTX	[p]	Context error (Called from disabled system state)

Function:

Each service call refers to the current priority of the task specified by the parameter tskid, and returns it to area indicated by parameter tskpri.

By specifying tskid = TSK_SELF (0), the current task is specified.

5.4.8 Refer to Task State (ref_tsk, iref_tsk)

C-Language API:

```
ER ercd = ref_tsk(ID tskid, T_RTSK *pk_rtsk);
ER ercd = iref_tsk(ID tskid, T_RTSK *pk_rtsk);
```

Parameters:

ID	tskid	R0	Task ID
T_RTSK	*pk_rtsk	ER1	Pointer to the packet where the task state is to be returned

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code
T_RTSK	*pk_rtsk	--	Pointer to the packet where the task state is stored

Packet Structure:

```
typedef    struct    t_rtsk {
           STAT      tskstat;  +0    2    Task state
           PRI       tskpri;   +2    2    Current priority of the task
           PRI       tskbpri;  +4    2    Base priority of the task
           STAT      tskwait;  +6    2    Wait cause
           ID        wobjid;   +8    2    Wait object ID
           TMO       lefttmo;  +10   4    Time to timeout
           UINT      actcnt;   +14   2    Number of queued initiation requests
           UINT      wupcnt;   +16   2    Number of queued wakeup requests
           UINT      suscnt;   +18   2    Suspend request nest count
}T_RTSK;
```

Error Codes:

E_PAR	[p]	Parameter error (pk_rtsk is 0)
E_ID	[p]	Invalid ID number (tskid < 0, tskid > CFG_MAXTSKID, or tskid = TSK_SELF(0) is specified in a non-task context)
E_NOEXS	[p]	Undefined (Task indicated by tskid does not exist)

Function:

Each service call refers to the state of the task indicated by the parameter tskid, and then returns it to the area indicated by parameter pk_rtsk.

By specifying tskid = TSK_SELF(0), the current task is specified.

The following values are returned to the area indicated by pk_rtsk. Note that data with an asterisk (*) is invalid when the task is in the DORMANT state. If referenced information is related to a function that is not installed, the referenced information will be undefined.

- tskstat

  Indicates the current task state. The following values are returned.

Table 5.5 Current Task State (tskstat)

tskstat	Code	Description
TTS_RUN	H'0001	RUNNING state
TTS_RDY	H'0002	READY state
TTS_WAI	H'0004	WAITING state
TTS_SUS	H'0008	SUSPENDED state
TTS_WAS	H'000c	WAITING-SUSPENDED state
TTS_DMT	H'0010	DORMANT state
TTS_STK	H'4000	Shared-stack WAITING state

  If the task is in both the SUSPENDED state and shared-stack WAITING state, TTS_SUS | TTS_STK (H’4008) is returned.

  H’0000 may be returned as the value of tsksts. This indicates kernel execution. In such a case, other information returned by pk_rtsk becomes undefined.

- tskpri

  Indicates the current task priority. When the task is in the DORMANT state, the priority at task initiation is returned.

- tskbpri

  Indicates the base priority of the task. When the task is in the DORMANT state, the priority at task initiation is returned.

- tskwait*

  Valid only when TTS_WAI or TTS_WAS is returned to tskstat and the following values are returned.

Table 5.6 Cause of WAITING State (tskwait)

tskwait	Code	Description
TTW_SLP	H'0001	Shifted to the WAITING state by slp_tsk or tslp_tsk
TTW_DLY	H'0002	Shifted to the WAITING state by dly_tsk
TTW_SEM	H'0004	Shifted to the WAITING state by wai_sem or twai_sem
TTW_FLG	H'0008	Shifted to the WAITING state by wai_flg or twai_flg
TTW_SDTQ	H'0010	Shifted to the WAITING state by snd_dtq or tsnd_dtq
TTW_RDTQ	H'0020	Shifted to the WAITING state by rcv_dtq or trcv_dtq
TTW_MBX	H'0040	Shifted to the WAITING state by rcv_mbx or trcv_mbx
TTW_MTX	H'0080	Shifted to the WAITING state by loc_mtx or tloc_mtx
TTW_MPF	H'2000	Shifted to the WAITING state by get_mpf or tget_mpf
TTW_MPL	H'4000	Shifted to the WAITING state by get_mpl or tget_mpl

- wobjid*

  Valid only when TTS_WAI or TTS_WAS is returned to tskstat and the waiting target object ID is returned.

- lefttmo*

  The time until the target task times out is returned. Note that when the target task is in the WAITING state according to the service call dly_tsk, the value is undefined.

- actcnt*

  The current initiation request queue count is returned.

- wupcnt*

  The current wakeup request queue count is returned.

- suscnt*

  The current suspend request nesting count is returned.

5.4.9 Refer to Task State (Simple Version) (ref_tst, iref_tst)

C-Language API:

```
ER ercd = ref_tst(ID tskid, T_RTST *pk_rtst);
ER ercd = iref_tst(ID tskid, T_RTST *pk_rtst);
```

Parameters:

ID	tskid	R0	Task ID
T_RTST	*pk_rtst	ER1	Start address of the packet where the task state is to be returned

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code
T_RTST	*pk_rtst	--	Start address of the packet where the task state is stored

Packet Structure:

```
typedef    struct    t_rtst {
           STAT      tskstat;  +0    2    Task state
           STAT      tskwait;  +2    2    Wait cause
}T_RTST;
```

Error Codes:

E_PAR	[p]	Parameter error (pk_rtst is 0)
E_ID	[p]	Invalid ID number (tskid < 0, tskid > CFG_MAXTSKID, or tskid = TSK_SELF(0) is specified in a non-task context)
E_NOEXS	[p]	Undefined (Task indicated by tskid does not exist)

Function:

Each service call refers to the minimum state of the task indicated by the parameter tskid, and then returns it to the area indicated by parameter pk_rtst. By specifying tskid = TSK_SELF (0), the current task can be specified.

The following values are returned to the area indicated by pk_rtst. Note that data with an asterisk (*) is invalid when the task is in the DORMANT state. If referenced information is related to a function that is not installed, the referenced information will be undefined.

- tskstat

  Indicates the current task state. The following values are returned.

Table 5.7 Current Task State (tskstat)

tskstat	Code	Description
TTS_RUN	H'0001	RUNNING state
TTS_RDY	H'0002	READY state
TTS_WAI	H'0004	WAITING state
TTS_SUS	H'0008	SUSPENDED state
TTS_WAS	H'000c	WAITING-SUSPENDED state
TTS_DMT	H'0010	DORMANT state
TTS_STK	H'4000	Shared-stack WAITING state

If the task is in both the SUSPENDED state and shared-stack WAITING state, TTS_SUS | TTS_STK (H'4008) is returned.

H'0000 may be returned as the value of tsksts. This indicates kernel execution. In such a case, other information returned by pk_rtsk becomes undefined.

- tskwait*

  Valid only when TTS_WAI or TTS_WAS is returned to tskstat and the following values are returned.

Table 5.8 Cause of WAITING State (tskwait)

tskwait	Code	Description
TTW_SLP	H'0001	Shifted to the WAITING state by slp_tsk or tslp_tsk
TTW_DLY	H'0002	Shifted to the WAITING state by dly_tsk
TTW_SEM	H'0004	Shifted to the WAITING state by wai_sem or twai_sem
TTW_FLG	H'0008	Shifted to the WAITING state by wai_flg or twai_flg
TTW_SDTQ	H'0010	Shifted to the WAITING state by snd_dtq or tsnd_dtq
TTW_RDTQ	H'0020	Shifted to the WAITING state by rcv_dtq or trcv_dtq
TTW_MBX	H'0040	Shifted to the WAITING state by rcv_mbx or trcv_mbx
TTW_MTX	H'0080	Shifted to the WAITING state by loc_mtx or tloc_mtx
TTW_MPF	H'2000	Shifted to the WAITING state by get_mpf or tget_mpf
TTW_MPL	H'4000	Shifted to the WAITING state by get_mpl or tget_mpl

5.5 Task Synchronization

The service calls for task synchronization are listed in Table 5.9.

Table 5.9 Task Synchronization Service Calls

Service Call[*1]		Description	System State[*2] T/N/E/D/U/L/C
slp_tsk	[S]	Shifts current task to the WAITING state	T/E/U
tslp_tsk	[S]	Shifts current task to the WAITING state with timeout function	T/E/U
wup_tsk	[S]	Wakes up task	T/E/D/U
iwup_tsk	[S]		N/E/D/U
can_wup	[S]	Cancel Wakeup Task	T/E/D/U
rel_wai	[S]	Cancels the WAITING state forcibly	T/E/D/U
irel_wai	[S]		N/E/D/U
sus_tsk	[S]	Shifts to the SUSPENDED state	T/E/D/U
rsm_tsk	[S]	Resumes the execution of a task in the SUSPENDED state	T/E/D/U
frsm_tsk	[S]	Forcibly resumes the execution of a task in the SUSPENDED state	T/E/D/U
dly_tsk	[S]	Delays the current task	T/E/U

Notes: 1. [S]: Standard profile service calls

2. T: Can be called from task context
N: Can be called from non-task context
E: Can be called from dispatch-enabled state
D: Can be called from dispatch-disabled state
U: Can be called from CPU-unlocked state
L: Can be called from CPU-locked state
C: Can be called from CPU exception handler

The task synchronization specifications are listed in Table 5.10.

Table 5.10 Task Synchronization Specifications

Item	Description
Maximum number of task wake-up request count	255
Maximum number of task suspend request nesting	1

5.5.1 Sleep Task (slp_tsk, tslp_tsk)

C-Language API:

```
ER ercd = slp_tsk( );
ER ercd = tslp_tsk(TMO tmout);
```

Parameters:

```
<tslp_tsk>
TMO              tmout        ER0  Timeout specification
```

Return Parameters:

```
ER               ercd         R0   Normal termination (E_OK) or error code
```

Error Codes:

```
E_NOSPT      [p]   Unsupported function (Time management function is not used
                   (tslp_tsk))
E_PAR        [p]   Parameter error (tmout ≤ -2)
E_CTX        [p]   Context error (Called from disabled system state)
             [k]   (tslp_tsk is called from dispatch-disabled state or CPU-
                   locked state while tmout in tslp_tsk is not TMO_POL (0))
E_TMOUT      [k]   Timeout
E_RLWAI      [k]   WAITING state is forcibly cancelled (rel_wai was accepted
                   in the WAITING state)
```

Function:

Each service call shifts the current task to the wake-up WAITING state. However, if wake-up requests are queued for the current task, the wake-up request count is decremented by one and task execution continues. The WAITING state is cancelled by the service calls wup_tsk and iwup_tsk.

The parameter tmout specified by service call tslp_tsk specifies the timeout period.

If a positive value is specified for parameter tmout, the WAITING state is released and error code E_TMOUT is returned when the tmout period has passed without the wait release conditions being satisfied. If tmout = TMO_POL (0) is specified, the task continues execution by decrementing the wake-up request count by one if the count is a positive value. If the wake-up request count is 0, error code E_TMOUT is returned. If tmout = TMO_FEVR (–1) is specified, the same operation as for service call slp_tsk will be performed. In other words, timeout will not be monitored.

If a value larger than 1 is specified for the denominator of time tick cycles (CFG_TICDENO), the maximum value that can be specified for tmout is H'7fffffff/denominator for time tick cycles. If a value larger than this is specified, operation is not guaranteed.

5.5.2 Wakeup Task (wup_tsk, iwup_tsk)

C-Language API:

```
ER ercd = wup_tsk(ID tskid);
ER ercd = iwup_tsk(ID tskid);
```

Parameters:

ID	tskid	R0	Task ID

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (tskid < 0, tskid > CFG_MAXTSKID, or tskid = TSK_SELF(0) is specified in a non-task context)
E_NOEXS	[p]	Undefined (Task indicated by tskid does not exist)
E_OBJ	[k]	Object state error (Task indicated by tskid is in the DORMANT state)
E_QOVR	[k]	Queuing overflow (wupcnt > H'ff)
E_CTX	[p]	Context error (Called from disabled system state)

Function:

Each service call releases a task from the WAITING state after the task was assigned to the WAITING state by calling the service call slp_tsk or tslp_tsk. If the target task did not enter the WAITING state by calling the service call slp_tsk or tslp_tsk, up to 255 requests to wake up a task can be stored.

By specifying tskid = TSK_SELF (0), the current task can be specified.

5.5.3 Cancel Wakeup Task (can_wup)

C-Language API:

```
ER_UINT wupcnt = can_wup(ID tskid);
```

Parameters:

ID	tskid	R0	Task ID

Return Parameters:

ER_UINT	wupcnt	R0	Number of queued task wake-up requests (0 or a positive value) or error code

Error Codes:

E_ID	[p]	Out of ID range (tskid < 0, tskid > CFG_MAXTSKID)
E_NOEXS	[p]	Undefined (Task indicated by tskid is not created)
E_OBJ	[k]	Object state error (Task indicated by tskid is in the DORMANT state)
E_CTX	[p]	Context error (Called from disabled system state)

Function:

Each service call calculates the number of wake-up requests queued for the task specified by tskid, then returns the result as a return parameter and invalidate all of those requests.

By specifying tskid = TSK_SELF (0), the current task can be specified.

5.5.4 Release WAITING State Forcibly (rel_wai, irel_wai)

C-Language API:

```
ER ercd = rel_wai(ID tskid);
ER ercd = irel_wai(ID tskid);
```

Parameters:

ID	tskid	R0	Task ID

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (tskid ≤ 0, tskid > CFG_MAXTSKID)
E_NOEXS	[p]	Undefined (Task indicated by tskid is not created)
E_OBJ	[k]	Object state error (Task indicated by tskid is in the DORMANT state or the current task ID is specified)
E_CTX	[p]	Context error (Called from disabled system state)

Function:

When the task specified by tskid is in some kind of WAITING state (not including a SUSPENDED state or shared-stack release WAITING state), it is forcibly cancelled. E_RLWAI is returned as the error code for the task for which the WAITING state is cancelled by the service call rel_wai or irel_wai.

If the service calls rel_wai and irel_wai are called for a task in a WAITING-SUSPENDED state, the task enters the SUSPENDED state. Thereafter, if the service call rsm_tsk or frsm_tsk is called and the SUSPENDED state is cancelled, E_RLWAI is returned as the error code for the task.

Note that the service calls rel_wai and irel_wai cannot cancel the shared-stack WAITING state.

5.5.5 Suspend Task (sus_tsk)

C-Language API:

```
ER ercd = sus_tsk(ID tskid);
```

Parameters:

```
ID          tskid       R0    Task ID
```

Return Parameters:

```
ER          ercd        R0    Normal termination (E_OK) or error code
```

Error Codes:

```
E_ID         [p]   Invalid ID number (tskid < 0, tskid > CFG_MAXTSKID)
E_NOEXS      [p]   Undefined (Task indicated by tskid does not exist)
E_OBJ        [k]   Object state error (Task specified by tskid is in the
                   DORMANT state)
E_CTX        [p]   Context error (Called from disabled system state)
             [k]   Context error (tskid=TSK_SELF(0) or the current task ID is
                   specified in a task context while dispatch is disabled or
                   the CPU is locked)
E_QOVR       [k]   Queuing overflow (Already in the SUSPENDED state)
```

Function:

Each service call suspends execution of the task specified by tskid and shifts the task to the SUSPENDED state. If the specified task is in the WAITING state, the task shifts to the WAITING-SUSPENDED state.

By specifying tskid = TSK_SELF (0), the current task can be specified.

The SUSPENDED state can be cancelled by calling the service call rsm_tsk or frsm_tsk.

Requests to suspend a task by calling the service call sus_tsk are not nested.

5.5.6 Resume Task, Resume Task Forcibly (rsm_tsk, frsm_tsk)

C-Language API:

```
ER ercd = rsm_tsk(ID tskid);
ER ercd = frsm_tsk(ID tskid);
```

Parameters:

```
ID            tskid       R0    Task ID
```

Return Parameters:

```
ER            ercd        R0    Normal termination (E_OK) or error code
```

Error Codes:

```
E_ID          [p]   Invalid ID number (tskid ≤ 0 or tskid > CFG_MAXTSKID)
E_NOEXS       [p]   Undefined (Task indicated by tskid does not exist)
E_OBJ         [k]   Object state error (Task indicated by tskid is not in the
                    SUSPENDED state)
E_CTX         [p]   Context error (Called from disabled system state)
```

Function:

Each service call releases the task specified by parameter tskid from the SUSPENDED state. When the task is in the WAITING-SUSPENDED state, the task is shifted to the WAITING state.

In the HI1000/4, the service calls rsm_tsk and frsm_tsk perform the same processing because the requests to suspend a task are not nested.

5.5.7 Delay Task (dly_tsk)

C-Language API:

```
ER ercd = dly_tsk(RELTIM dlytim);
```

Parameters:

RELTIM	dlytim	ER0	Delayed time

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_NOSPT	[p]	Unsupported function (Time management function is not used)
E_CTX	[p]	Context error (Called from disabled system state)
	[k]	(Called from dispatch-disabled state or CPU-locked state)
E_RLWAI	[k]	WAITING state is forcibly cancelled (rel_wai was accepted in the WAITING state)

Function:

The current task is transferred from the RUNNING state to a timed WAITING state, and waits until the time specified by dlytim has expired. When the time specified by dlytim has elapsed, the state of the current task enters the READY state. The current task is put into a WAITING state also when dlytim=0 is specified.

If a value larger than 1 is specified for the denominator of time tick cycles (CFG_TICDENO), the maximum value that can be specified for dlytim is H'ffffffff/denominator for time tick cycles. If a value larger than this is specified, operation is not guaranteed.

This service call differs from the service call tslp_tsk in that it terminates normally when execution is delayed by the amount of time specified by dlytim. Further, even if a service call wup_tsk or iwup_tsk is executed during the delay time, the WAITING state is not cancelled. The WAITING state is cancelled before the delay time has elapsed only when a service call rel_wai, irel_wai, or ter_tsk is called.

5.6 Synchronization and Communication (Semaphore)

Semaphores are controlled by the service calls listed in Table 5.11.

Table 5.11 Service Calls for Semaphore Control

			System State[*2]
Service Call[*1]		**Description**	**T/N/E/D/U/L/C**
sig_sem	[S]	Returns semaphore resource	T/E/D/U
isig_sem	[S]		N/E/D/U
wai_sem	[S]	Waits on semaphore resource	T/E/U
pol_sem	[S]	Polls and waits on semaphore resource	T/E/D/U
ipol_sem			N/E/D/U
twai_sem	[S]	Waits on semaphore resource with timeout function	T/E/U
ref_sem		Refers to semaphore state	T/E/D/U
iref_sem			N/E/D/U

Notes: 1. [S]: Standard profile service calls

2. T: Can be called from task context
N: Can be called from non-task context
E: Can be called from dispatch-enabled state
D: Can be called from dispatch-disabled state
U: Can be called from CPU-unlocked state
L: Can be called from CPU-locked state
C: Can be called from CPU exception handler

The semaphore specifications are listed in Table 5.12.

Table 5.12 Semaphore Specifications

Item	Description
Semaphore ID	1 to CFG_MAXSEMID (255 max.)
Maximum number of resources	1 to 65535
Attribute supported	TA_TFIFO: Task wait queue is managed on a FIFO basis

5.6.1 Returns Semaphore Resource (sig_sem, isig_sem)

C-Language API:

```
ER ercd = sig_sem(ID semid);
ER ercd = isig_sem(ID semid);
```

Parameters:

ID	semid	R0	Semaphore ID

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (semid ≤ 0 or semid > CFG_MAXSEMID)
E_NOEXS	[p]	Undefined (Semaphore indicated by semid does not exist)
E_QOVR	[k]	Queuing overflow (semcnt > Maximum number of resources)
E_CTX	[p]	Context error (Called from disabled system state)

Function:

Each service call returns one resource to the semaphore indicated by semid. If there is a task waiting for the semaphore indicated by semid, the task at the head of the wait queue is released from the WAITING state, and the resource is assigned to the task. If there are no tasks in the wait queue, the number of semaphore resources is incremented by one.

The maximum number of semaphore resources is that specified when the semaphore is initially defined.

5.6.2 Wait for Semaphore (wai_sem, pol_sem, ipol_sem, twai_sem)

C-Language API:

```
ER ercd = wai_sem(ID semid);
ER ercd = pol_sem(ID semid);
ER ercd = ipol_sem(ID semid);
ER ercd = twai_sem(ID semid, TMO tmout);
```

Parameters:

ID	semid	R0	Semaphore ID
<twai_sem>			
TMO	tmout	ER1	Timeout specification

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_NOSPT	[p]	Unsupported function (Time management function is not used or timeout function is not used (twai_sem))
E_PAR	[p]	Parameter error (tmout ≤ −2)
E_ID	[p]	Invalid ID number (semid ≤ 0 or semid > CFG_MAXSEMID)
E_NOEXS	[p]	Undefined (Semaphore indicated by semid does not exist)
E_CTX	[p]	Context error (Called from disabled system state)
	[k]	(Cannot be called (wai_sem, twai_sem) from dispatch-disabled state or CPU-locked state and tmout in twai_sem is not TMO_POL (0))
E_RLWAI	[k]	WAITING state is forcibly cancelled (rel_wai was accepted in the WAITING state)
E_TMOUT	[k]	Polling failed or timeout

Function:

Each service call acquires one resource from the semaphore specified by semid.

Each service call decrements the number of resources of the target semaphore by one if the number of resources of the target semaphore is equal to or greater than 1, and the task calling the service call continues execution. If no resources exist, the task calling the service call wai_sem or twai_sem shifts to the WAITING state, and with service call pol_sem or ipol_sem, error code E_TMOUT is immediately returned. The wait queue is managed on a FIFO basis.

The parameter tmout specified by service call twai_sem specifies the timeout period.

If a positive value is specified for the parameter tmout, error code E_TMOUT is returned when the tmout period has passed without the wait release conditions being satisfied. If tmout = TMO_POL (0) is specified, the same operation as for the service call pol_sem will be performed. If tmout =

TMO_FEVR (–1) is specified, the timeout monitoring is not performed. In this case, the same operation as for the service call wai_sem will be performed.

If a value larger than 1 is specified for the denominator of time tick cycles (CFG_TICDENO), the maximum value that can be specified for tmout is H'7fffffff/denominator for time tick cycles. If a value larger than this is specified, operation is not guaranteed.

5.6.3 Refer to Semaphore State (ref_sem, iref_sem)

C-Language API:

```
ER ercd = ref_sem(ID semid, T_RSEM *pk_rsem);
ER ercd = iref_sem(ID semid, T_RSEM *pk_rsem);
```

Parameters:

```
ID            semid     R0   Semaphore ID
T_RSEM        *pk_rsem  ER1  Pointer to the area where the semaphore state is
                             to be returned
```

Return Parameters:

```
ER            ercd      R0   Normal termination (E_OK) or error code
T_RSEM        *pk_rsem  --   Pointer to the area where the semaphore state is
                             stored
```

Packet Structure:

```
typedef     struct    t_rsem{
            ID        wtskid;  +0    2    Wait task ID
            UINT      semcnt;  +2    2    Current semaphore resource count
}T_RSEM;
```

Error Codes:

```
E_PAR       [p]   Parameter error (pk_rsem is 0)
E_ID        [p]   Invalid ID number (semid ≤ 0 or semid > CFG_MAXSEMID)
E_NOEXS     [p]   Undefined (Semaphore indicated by semid does not exist)
```

Function:

Each service call refers to the state of the semaphore indicated by the parameter semid.

Each service call returns the task ID at the head of the semaphore wait queue (wtskid) and the current semaphore resource count (semcnt), to the area specified by the parameter pk_rsem.

If there is no task waiting for a semaphore, TSK_NONE (0) is returned as a wait task ID.

5.7 Synchronization and Communication (Event Flag)

Event flags are controlled by the service calls listed in Table 5.13.

Table 5.13 Service Calls for Event Flag Control

			System State*2
Service Call*1		**Description**	**T/N/E/D/U/L/C**
set_flg	[S]	Sets event flag	T/E/D/U
iset_flg	[S]		N/E/D/U
clr_flg	[S]	Clears event flag	T/E/D/U
iclr_flg			N/E/D/U
wai_flg	[S]	Waits for event flag	T/E/U
pol_flg	[S]	Polls and waits for event flag	T/E/D/U
ipol_flg	[S]		N/E/D/U
twai_flg	[S]	Waits for event flag with timeout function	T/E/U
ref_flg		Refers to event flag state	T/E/D/U
iref_flg			N/E/D/U

Notes: 1. [S]: Standard profile service calls
2. T: Can be called from task context
N: Can be called from non-task context
E: Can be called from dispatch-enabled state
D: Can be called from dispatch-disabled state
U: Can be called from CPU-unlocked state
L: Can be called from CPU-locked state
C: Can be called from CPU exception handler

The event flag specifications are listed in Table 5.14.

Table 5.14 Event Flag Specifications

Item	**Description**
Event flag ID	1 to CFG_MAXFLGID (255 max.)
Event flag bit count	16 bits
Attribute supported	TA_TFIFO: Task wait queue is managed on a FIFO basis TA_WSGL: Does not permit multiple tasks to wait for the event flag TA_WMUL: Permits multiple tasks to wait for the event flag TA_CLR: Bit pattern is cleared when the wait task is cancelled

5.7.1 Set Event Flag (set_flg, iset_flg)

C-Language API:

```
ER ercd = set_flg(ID flgid, FLGPTN setptn);
ER ercd = iset_flg(ID flgid, FLGPTN setptn);
```

Parameters:

ID	flgid	R0	Event flag ID
FLGPTN	setptn	E0	Bit pattern to set

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (flgid ≤ 0 or flgid > CFG_MAXFLGID)
E_NOEXS	[p]	Undefined (Event flag indicated by flgid does not exist)
E_CTX	[p]	Context error (Called from disabled system state)

Function:

The event flag specified by flgid is ORed with the value indicated by the parameter setptn.

Each service call shifts a task to the READY state after the event flag value has been changed and when the wait release conditions of a task waiting for an event flag have been satisfied. Wait release conditions are checked in the queue order. All bits of the event flag bit pattern and service call are cleared when the TA_CLR attribute is set to the target event flag attribute.

When the TA_WMUL attribute is set to the event flag and the TA_CLR attribute is not specified, multiple wait tasks may be released when the service call set_flg is called only once. When multiple wait tasks are released, the WAITING state of the tasks are cancelled in the queue order of the event flag.

5.7.2 Clear Event Flag (clr_flg, iclr_flg)

C-Language API:

```
ER ercd = clr_flg(ID flgid, FLGPTN clrptn);
ER ercd = iclr_flg(ID flgid, FLGPTN clrptn);
```

Parameters:

ID	flgid	R0	Event flag ID
FLGPTN	clrptn	E0	Bit pattern to clear

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (flgid ≤ 0 or flgid > CFG_MAXFLGID)
E_NOEXS	[p]	Undefined (Event flag indicated by flgid does not exist)

Function:

The event-flag bits specified by flgid is ANDed with the value indicated by the parameter clrptn.

5.7.3 Wait for Event Flag (wai_flg, pol_flg, ipol_flg, twai_flg)

C-Language API:

```
ER ercd = wai_flg(ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN  *p_flgptn);
ER ercd = pol_flg(ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN  *p_flgptn);
ER ercd = ipol_flg(ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN  *p_flgptn);
ER ercd = twai_flg (ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN  *p_flgptn,
TMO tmout);
```

Parameters:

ID	flgid	R0	Event flag ID
FLGPTN	waiptn	E0	Wait bit pattern
MODE	wfmode	R1	Wait mode
FLGPTN	*p_flgptn	ER2	Pointer to the area where the bit pattern at waiting release is to be returned
<twai_flg>			
TMO	tmout	@ER7	Timeout value

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code
FLGPTN	*p_flgptn	--	Pointer to the area where the bit pattern at waiting release is stored

Error Codes:

E_NOSPT	[p]	Unsupported function (Time management function is not used or timeout function is not used (twai_flg))
E_PAR	[p]	Parameter error (waiptn=0, wfmode is invalid, tmout ≤ -2, or p_flgptn is 0)
E_ID	[p]	Invalid ID (flgid ≤ 0, flgid > CFG_MAXFLGID)
E_NOEXS	[p]	Undefined (Event flag indicated by flgid does not exist)
E_ILUSE	[k]	Illegal use of service call (A task is already waiting for the event flag with TA_WSGL attribute)
E_CTX	[p]	Context error (Called from disabled system state)
	[k]	(Cannot be called (wai_flg, twai_flg) from dispatch-disabled state or CPU-locked state and tmout in twai_flg is not TMO_POL (0))
E_TMOUT	[k]	Polling failed or timeout
E_RLWAI	[k]	WAITING state is forcibly cancelled (rel_wai was accepted in the WAITING state)

Function:

A task that has called one of these service calls waits until the event flag specified by the parameter flgid is set according to the waiting conditions indicated by the parameters waiptn and wfmode. Each service call returns the bit pattern of the event flag to the area indicated by p_flgptn when the wait release condition is satisfied.

If the attribute of the target event flag is TA_WSGL and another task is waiting for the event flag, error code E_ILUSE is returned.

If the wait release conditions are met before a task calls service call wai_flg, pol_flg, ipol_flg, or twai_flg, the service call will be completed immediately. If they are not met, the task will be sent to the wait queue when the service call wai_flg or twai_flg is called. With service call pol_flg or ipol_flg, error code E_TMOUT is immediately returned, then the task terminates.

The parameter wfmode is specified in the following format. See Table 5.15 for details.

wfmode:= ((TWF_ANDW || TWF_ORW))

Table 5.15 Wait Modes (wfmode)

wfmode	Code	Description
TWF_ANDW	H'0000	AND wait
TWF_ORW	H'0001	OR wait

If TWF_ANDW is specified as wfmode, the task waits until all the bits specified by waiptn have been set. If TWF_ORW is specified as wfmode, the task waits until any one of the bits specified by waiptn has been set in the specified event flag.

The parameter tmout specified by service call twai_flg specifies the timeout period.

If a positive value is specified for the parameter tmout, error code E_TMOUT is returned when the timeout period has passed without the waiting release conditions being satisfied. If tmout = TMO_POL (0) is specified, the same operation as for the service call pol_flg will be performed. If tmout = TMO_FEVR (–1) is specified, the timeout monitoring is not performed. In this case, the same operation as for service call wai_flg will be performed.

If a value larger than 1 is specified for the denominator of time tick cycles (CFG_TICDENO), the maximum value that can be specified for tmout is H'7fffffff/denominator for time tick cycles. If a value larger than this is specified, operation is not guaranteed.

If the TA_CLR attribute is specified for the target event flag, all bits in the bit pattern of the event flag are cleared when the task is released from the event flag WAITING state.

5.7.4 Refer to Event Flag State (ref_flg, iref_flg)

C-Language API

```
ER ercd = ref_flg(ID flgid, T_RFLG *pk_rflg);
ER ercd = iref_flg(ID flgid, T_RFLG *pk_rflg);
```

Parameters:

ID	flgid	R0	Event flag ID
T_RFLG	*pk_rflg	ER1	Pointer to the area where the event flag state is to be returned

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code
T_RFLG	*pk_rflg	--	Pointer to the packet where event flag state is stored

Packet Structure:

```
typedef   struct    t_rflg{
          ID        wtskid;    +0    2     Wait task ID
          FLGPTN    flgptn;    +2    2     Event flag bit pattern
}T_RFLG;
```

Error Codes:

E_PAR	[p]	Parameter error (pk_rflg is 0)
E_ID	[p]	Invalid ID number (flgid ≤ 0 or flgid > CFG_MAXFLGID)
E_NOEXS	[p]	Undefined (Event flag indicated by flgid does not exist)

Function:

Each service call refers to the state of the event flag indicated by the parameter flgid.

Each service call returns the task ID at the head of the event flag wait queue (wtskid) and the current event flag bit pattern (flgptn), to the area specified by the parameter pk_rflg.

If there is no task waiting for the specified event flag, TSK_NONE (0) is returned as a wait task ID.

5.8 Synchronization and Communication (Data Queue)

Data queues are controlled by the service calls listed in Table 5.16.

Table 5.16 Service Calls for Data Queue Control

			System State*2
Service Call*1		**Description**	**T/N/E/D/U/L/C**
snd_dtq	[S]	Sends data to data queue	T/E/U
psnd_dtq	[S]	Polls and sends data to data queue	T/E/D/U
ipsnd_dtq	[S]		N/E/D/U
tsnd_dtq	[S]	Sends data to data queue with timeout function	T/E/U
fsnd_dtq	[S]	Forcibly sends data to data queue	T/E/D/U
ifsnd_dtq	[S]		N/E/D/U
rcv_dtq	[S]	Receives data from data queue	T/E/U
prcv_dtq	[S]	Polls and receives data from data queue	T/E/D/U
trcv_dtq	[S]	Receives data from data queue with timeout function	T/E/U
ref_dtq		Refers to data queue state	T/E/D/U
iref_dtq			N/E/D/U

Notes: 1. [S]: Standard profile service calls

2. T: Can be called from task context
N: Can be called from non-task context
E: Can be called from dispatch-enabled state
D: Can be called from dispatch-disabled state
U: Can be called from CPU-unlocked state
L: Can be called from CPU-locked state
C: Can be called from CPU exception handler

The data queue specifications are listed in Table 5.17.

Table 5.17 Data Queue Specifications

Item	Description
Data queue ID	1 to CFG_MAXDTQID (255 max.)
Data queue area capacity (number of data items)	0 to 65535
Data size	32 bits
Attribute supported	TA_TFIFO: Task wait queue is managed on a FIFO basis

5.8.1 Send Data to Data Queue (snd_dtq, psnd_dtq, ipsnd_dtq, tsnd_dtq, fsnd_dtq, ifsnd_dtq)

C-Language API:

```
ER ercd = snd_dtq(ID dtqid, VP_INT data);
ER ercd = psnd_dtq(ID dtqid, VP_INT data);
ER ercd = ipsnd_dtq(ID dtqid, VP_INT data);
ER ercd = tsnd_dtq(ID dtqid, VP_INT data, TMO tmout);
ER ercd = fsnd_dtq(ID dtqid, VP_INT data);
ER ercd = ifsnd_dtq(ID dtqid, VP_INT data);
```

Parameters:

ID	dtqid	R0	Data Queue ID
VP_INT	data	ER1	Data sent to data queue
<tsnd_dtq>			
TMO	tmout	ER2	Timeout specification

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_NOSPT	[p]	Unsupported function (Time management function is not used or timeout function is not used (tsnd_dtq))
E_PAR	[p]	Parameter error (tmout ≤ -2)
E_ID	[p]	Invalid ID number (dtqid ≤ 0 or dtqid > CFG_MAXDTQID)
E_NOEXS	[p]	Undefined (Data queue indicated by dtqid does not exist)
E_CTX	[p]	Context error (Called from disabled system state)
	[k]	(Cannot be called (snd_dtq, tsnd_dtq) from dispatch-disabled state or CPU-locked state and tmout in tsnd_dtq is not TMO_POL (0))
E_TMOUT	[k]	Polling failed or timeout
E_RLWAI	[k]	WAITING state is forcibly cancelled (rel_wai was accepted in the WAITING state)
E_ILUSE	[k]	Illegal use of service call (fsnd_dtq or ifsnd_dtq is issued for the data queue whose data queue area capacity is 0)

Function:

The data specified by the parameter data (four bytes) is sent to the data queue specified by dtqid.

When receive-waiting tasks exist for the data queue, the data is not stored in the data queue, but instead is passed to the task at the head of the receive-waiting queue, and the WAITING state for that task is released.

When send-waiting tasks already exist for the data queue, the service calls snd_dtq and tsnd_dtq result in connecting to a wait queue (send-waiting queue) to wait for free space in the data queue, and the service calls psnd_dtq and ipsnd_dtq immediately return with an E_TMOUT error. Send-waiting queues are managed on a FIFO basis.

When neither receive-waiting tasks nor send-waiting tasks exist, the data is stored in the data queue. As a result, the number of data items is incremented by one.

When the number of data items equals the data queue area capacity, the calling task is linked to the send-waiting queue.

In the case of a service call tsnd_dtq, the wait time is specified to tmout.

If a positive value is specified for the parameter tmout, error code E_TMOUT is returned when the timeout period has passed without the wait release conditions being satisfied. If tmout = TMO_POL (0) is specified, the same operation as for the service call psnd_dtq will be performed. If tmout = TMO_FEVR (–1) is specified, timeout monitoring is not performed. In other words, the same operation as for service call snd_dtq will be performed.

When a value larger than 1 is specified for the denominator of time tick cycles (CFG_TICDENO), the maximum value that can be specified for tmout is H'7fffffff/denominator for time tick cycles. If a value larger than this is specified, operation is not guaranteed.

In the case of the service calls fsnd_dtq and ifsnd_dtq, when a receive-waiting task exists in the data queue, or when there is no receive-waiting task but there is a free space in the data queue, processing is the same as for service calls snd_dtq and isnd_dtq.

However, when a send-waiting task exists in the data queue, or when there is no send-waiting task but there is no free space in the data queue, the leading (oldest) data in the data queue is deleted, and data is sent to that area.

5.8.2 Receive Data from Data Queue (rcv_dtq, prcv_dtq, trcv_dtq)

C-Language API:

```
ER ercd = rcv_dtq(ID dtqid, VP_INT *p_data);
ER ercd = prcv_dtq(ID dtqid, VP_INT *p_data);
ER ercd = trcv_dtq(ID dtqid, VP_INT *p_data, TMO tmout);
```

Parameters:

ID	dtqid	R0	Data queue ID
VP_INT	*p_data	ER1	Start address of the area where received data is to be returned
<trcv_dtq>			
TMO	tmout	ER2	Timeout specification

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code
VP_INT	*p_data	--	Pointer to the area where received data is stored

Error Codes:

E_NOSPT	[p]	Unsupported function (Time management function is not used or timeout function is not used (trcv_dtq))
E_PAR	[p]	Parameter error (p_data is 0 or tmout ≤ -2)
E_ID	[p]	Invalid ID number (dtqid ≤ 0 or dtqid > CFG_MAXDTQID)
E_NOEXS	[p]	Undefined (Data queue indicated by dtqid does not exist)
E_CTX	[p]	Context error (Called from disabled system state)
	[k]	(Cannot be called (rcv_dtq, trcv_dtq) from dispatch-disabled state or CPU-locked state and tmout in trcv_dtq is not TMO_POL (0))
E_TMOUT	[k]	Polling failed or timeout
E_RLWAI	[k]	WAITING state is forcibly cancelled (rel_wai was accepted in the WAITING state)

Function:

Data is received from the data queue specified by dtqid, and stored in the area indicated by parameter data.

If there is data in the data queue, the leading data (the oldest message) is received. On receiving data from the data queue, the number of data items is decremented by 1. As a result, if data can be stored for a task in the send-waiting queue, data is sent and processed in the order of the wait queue.

If there is no data in the data queue, and there exists a data send-waiting task (such a circumstance can occur only when the data queue area capacity is 0), the data of the task at the head of data send-waiting queue is received. As a result, the WAITING state of the data send-waiting task is cancelled.

If there is no data in the data queue, and there are also no data send-waiting tasks, a service call rcv_dtq or trcv_dtq causes the calling task to be linked to a receive-waiting queue. In the case of a service call prcv_dtq, the call returns immediately with an E_TMOUT error. The receive-waiting queue is managed in FIFO order.

In the case of the service call trcv_dtq, tmout specifies the wait time.

If a positive value is specified for the parameter tmout, error code E_TMOUT is returned when the timeout period has passed without the wait release conditions being satisfied. If tmout = TMO_POL (0) is specified, the same operation as for the service call prcv_dtq will be performed. If tmout = TMO_FEVR (–1) is specified, timeout monitoring is not performed. In other words, the same operation as for service call rcv_dtq will be performed.

When a value larger than 1 is specified for the denominator of time tick cycles (CFG_TICDENO), the maximum value that can be specified for tmout is H'7fffffff/denominator for time tick cycles. If a value larger than this is specified, operation is not guaranteed.

5.8.3 Refer to Data Queue State (ref_dtq, iref_dtq)

C-Language API:

```
ER ercd = ref_dtq(ID dtqid, T_RDTQ *pk_rdtq);
ER ercd = iref_dtq(ID dtqid, T_RDTQ *pk_rdtq);
```

Parameters:

ID	dtqid	R0	Data queue ID
T_RDTQ	*pk_rdtq	ER1	Pointer to the packet where data queue state is to be returned

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code
T_RDTQ	*pk_rdtq	--	Pointer to the packet where data queue state is stored

Packet Structure:

```
typedef    struct    t_rdtq{
           ID        stskid;   0    2    Task ID waiting for sending
           ID        rtskid;   +2   2    Task ID waiting for receiving
           UINT      sdtqcnt;  +4   2    Number of data items in the data
                                         queue
}T_RDTQ;
```

Error Codes:

E_PAR	[p]	Parameter error (pk_rdtq is 0)
E_ID	[p]	Invalid ID number (dtqid ≤ 0 or dtqid > CFG_MAXDTQID)
E_NOEXS	[p]	Undefined (Data queue indicated by dtqid does not exist)

Function:

The state of the data queue specified by dtqid is referenced, and the send-waiting task IDs (stskid), the receive-waiting task IDs (rtskid), and the number of data items in the data queue (sdtqcnt) are returned to the area specified by pk_rdtq.

If there are no send-waiting tasks or receive-waiting tasks, TSK_NONE(0) is returned as the wait task ID.

5.9 Synchronization and Communication (Mailbox)

Mailboxes are controlled by the service calls listed in Table 5.18.

Table 5.18 Service Calls for Mailbox Control

			System State[*2]
Service Call[*1]		**Description**	**T/N/E/D/U/L/C**
snd_mbx	[S]	Sends data to mailbox	T/E/D/U
isnd_mbx			N/E/D/U
rcv_mbx	[S]	Receives data from mailbox	T/E/U
prcv_mbx	[S]	Polls and receives data from mailbox	T/E/D/U
iprcv_mbx			NE/D/U
trcv_mbx	[S]	Receives data from mailbox with timeout function	T/E/U
ref_mbx		Refers to mailbox state	T/E/D/U
iref_mbx			N/E/D/U

Notes: 1. [S]: Standard profile service calls

2. T: Can be called from task context
N: Can be called from non-task context
E: Can be called from dispatch-enabled state
D: Can be called from dispatch-disabled state
U: Can be called from CPU-unlocked state
L: Can be called from CPU-locked state
C: Can be called from CPU exception handler

The mailbox specifications are listed in Table 5.19.

Table 5.19 Mailbox Specifications

Item	**Description**
Mailbox ID	1 to CFG_MAXMBXID (255 max.)
Message priority	1 to CFG_MAXMSGPRI (255 max.)
Attribute supported	TA_TFIFO: Task wait queue is managed on a FIFO basis TA_TPRI: Task wait queue is managed on a priority basis TA_MFIFO: Message queue is managed on a FIFO basis TA_MPRI: Message queue is managed on a priority basis

5.9.1 Send Message to Mailbox (snd_mbx, isnd_mbx)

C-Language API:

```
ER ercd = snd_mbx(ID mbxid, T_MSG *pk_msg);
ER ercd = isnd_mbx(ID mbxid, T_MSG *pk_msg);
```

Parameters:

```
ID             mbxid           R0    Mailbox ID
T_MSG          *pk_msg         ER1   Start address of the message to be sent
```

Return Parameters:

```
ER             ercd            R0    Normal termination (E_OK) or error code
```

Packet Structure:

```
Mailbox message header
typedef      struct  t_msg{
             VP      msghead;  +0    4    Kernel management area
}T_MSG;
Mailbox message header with priority
typedef      struct  t_msg_pri{
             T_MSG   msgque;   +0    4    Message header
             PRI     msgpri;   +4    2    Message priority
}T_MSG PRI;
```

Error Codes:

```
E_PAR        [p]   Parameter error (pk_msg is 0)
             [k]   (msgpri ≤ 0, msgpri > CFG_MAXMSGPRI, or the first four
                   bytes of the message are other than 0)
E_ID         [p]   Invalid ID number (mbxid ≤ 0, mbxid < 0, or
                   mbxid > CFG_MAXMBXID)
E_NOEXS      [p]   Undefined (Mailbox indicated by mbxid does not exist)
E_CTX        [p]   Context error (Called from disabled system state)
```

Function:

Each service call sends a message specified by pk_msg to the mailbox specified by mbxid.

If there is a task waiting to receive a message in the mailbox, the task at the head of the wait queue receives the message and is released from the WAITING state. On the other hand, if there are no tasks waiting to receive a message, the message specified by pk_msg is linked to the end of the message queue. The message queue is managed according to the attribute specified at initial definition.

To send a message to a mailbox that has the TA_MFIFO attribute, the message must be created in RAM and must have the T_MSG structure at the head of the message, as shown in Figure 5.3. The contents of T_MSG must be 0 when sending a message.

To send a message to a mailbox that has the TA_MPRI attribute, the message must be created in RAM and must have the T_MSG_PRI structure at the head of the message, as shown in Figure 5.4. The contents of T_MSG must be 0 when sending a message.

Note that the T_MSG area is used by the kernel; therefore the area must not be modified after message transfer. If this area is modified, normal system operation cannot be guaranteed.

```
typedef struct  user_msg {
   T_MSG        t_msg;   /* T_MSG structure                         */
   B            data[8]; /* Example of user message data structure */
                         /* (Structure determined by user)         */
} USER_MSG;
```

Figure 5.3 Example of a Message Form

```
typedef struct  user_msg {
   T_MSG_PRI    t_msg;   /* T_MSG structure                         */
   B            data[8]; /* Example of user message data structure */
                         /* (Structure determined by user)         */
} USER_MSG;
```

Figure 5.4 Example of a Message Form with Priority

5.9.2 Receive Message from Mailbox (rcv_mbx, prcv_mbx, iprcv_mbx, trcv_mbx)

C-Language API:

```
ER ercd = rcv_mbx(ID mbxid, T_MSG **ppk_msg);
ER ercd = prcv_mbx(ID mbxid, T_MSG **ppk_msg);
ER ercd = iprcv_mbx(ID mbxid, T_MSG **ppk_msg);
ER ercd = trcv_mbx(ID mbxid, T_MSG **ppk_msg, TMO tmout);
```

Parameters:

ID	mbxid	R0	Mailbox ID
T_MSG	**ppk_msg	ER1	Pointer to the area where the start address of the received message is to be returned
<trcv_mbx>			
TMO	tmout	ER2	Timeout specification

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code
T_MSG	**ppk_msg	--	Pointer to the area where the start address of the received message is stored

Packet Structure:

```
<Mailbox message header>
typedef        struct  t_msg{
               VP      msghead;  +0    4    Kernel management area
}T_MSG;
<Mailbox message header with priority>
typedef        struct  t_msg_pri{
               T_MSG   msgque;   +0    4    Message header
               PRI     msgpri;   +4    2    Message priority
}T_MSG PRI;
```

Error Codes:

E_NOSPT	[p]	Unsupported function (Time management function is not used or timeout function is not used (trcv_mbx))
E_PAR	[p]	Parameter error (ppk_msg is 0 or tmout ≤ -2)
E_ID	[p]	Invalid ID number (mbxid ≤ 0 or mbxid > CFG_MAXMBXID)
E_NOEXS	[p]	Undefined (Mailbox indicated by mbxid does not exist)
E_CTX	[p]	Context error (Called from disabled system state)
	[k]	(Cannot be called (rcv_mbx, trcv_mbx) from dispatch-disabled state or CPU-locked state and tmout in trcv_mbx is not TMO_POL (0))
E_TMOUT	[k]	Polling failed or timeout
E_RLWAI	[k]	WAITING state is forcibly cancelled (rel_wai was accepted in the WAITING state)

Function:

Each service call receives a message from the mailbox specified by parameter mbxid. Then the start address of the received message is returned to the area indicated by parameter pk_msg.

With service calls rcv_mbx and trcv_mbx, if there are no messages in the mailbox, the task that called the service call is placed in the wait queue to receive a message. With service calls prcv_mbx and iprcv_mbx, if there are no messages in the mailbox, error code E_TMOUT is returned immediately. The wait queue is managed according to the attribute specified at initial definition.

Parameter tmout specified by service call trcv_mbx specifies the timeout period.

If a positive value is specified for parameter tmout, error code E_TMOUT is returned when the timeout period has passed without the wait release conditions being satisfied. If tmout = TMO_POL (0) is specified, the same operation as for service call prcv_mbx will be performed. If tmout = TMO_FEVR (–1) is specified, timeout monitoring is not performed. In other words, the same operation as for service call rcv_mbx will be performed.

If a value larger than 1 is specified for the denominator of time tick cycles (CFG_TICDENO), the maximum value that can be specified for tmout is H'7fffffff/denominator for time tick cycles. If a value larger than this is specified, operation is not guaranteed.

5.9.3 Refer to Mailbox State (ref_mbx, iref_mbx)

C-Language API:

```
ER ercd = ref_mbx(ID mbxid, T_RMBX *pk_rmbx);
ER ercd = iref_mbx(ID mbxid, T_RMBX *pk_rmbx);
```

Parameters:

ID	mbxid	R0	Mailbox ID
T_RMBX	*pk_rmbx	ER1	Pointer to the area where the mailbox state is to be returned

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code
T_RMBX	*pk_rmbx	--	Pointer to the packet where the mailbox state is stored

Packet Structure:

```
(1) T_RMBX
typedef   struct   t_rmbx{
          ID       wtskid;     +0    2    Wait task ID
          T_MSG    *pk_msg;    +2    4    Start address of the message to be
                                          received next
}T_RMBX;
(2) T_MSG
<Mailbox message header>
typedef   struct   t_msg{
          VP       msghead;    +0    4    Kernel management area
}T_MSG;
<Mailbox message header with priority>
typedef   struct   t_msg_pri{
          T_MSG    msgque;     +0    4    Message header
          PRI      msgpri;     +4    2    Message priority
}T_MSG_PRI;
```

Error Codes:

E_PAR	[p]	Parameter error (pk_rmbx is 0)
E_ID	[p]	Invalid ID number (mbxid ≤ 0 or mbxid > CFG_MAXMBXID)
E_NOEXS	[p]	Undefined (Mailbox indicated by mbxid does not exist)

Function:

Each service call refers to the state of the mailbox indicated by parameter mbxid.

Service calls ref_mbx and iref_mbx return the wait task ID (wtskid) and the start address of the message to be received next (pk_msg) to the area indicated by pk_rmbx.

If there is no task waiting for the specified message, TSK_NONE (0) is returned as a wait task ID.

If there is no message to be received next, NULL (0) is returned as a message start address.

5.10 Synchronization and Communication (Mutex)

Mutexes are controlled by the service calls listed in Table 5.20.

Table 5.20 Service Calls for Mutex Control

		System State[*2]
Service Call[*1]	**Description**	**T/N/E/D/U/L/C**
loc_mtx	Locks mutex	T/E/U
ploc_mtx	Polls and locks mutex	T/E/D/U
tloc_mtx	Locks mutex with timeout function	T/E/U
unl_mtx	Unlocks mutex	T/E/D/U
ref_mtx	Refers to mutex state	T/E/D/U

Notes: 1. [S]: Standard profile service calls

2. T: Can be called from task context
N: Can be called from non-task context
E: Can be called from dispatch-enabled state
D: Can be called from dispatch-disabled state
U: Can be called from CPU-unlocked state
L: Can be called from CPU-locked state
C: Can be called from CPU exception handler

The mutex specifications are listed in Table 5.21.

Table 5.21 Mutex Specifications

Item	**Description**
Mutex ID	1 to CFG_MAXMTXID (255 max.)
Attribute supported	TA_CEILING (Ceiling priority protocol)

Note: In the HI1000/4, when the TA_CEILING attribute is specified, the mutex is managed by the "simplified priority control rule". Under this rule, the management which changes the task's current priority to higher value is always operated, but the management which changes the task's priority to lower value is not operated only when the task releases all of mutexes.

5.10.1 Lock Mutex (loc_mtx, ploc_mtx, tloc_mtx)

C-Language API:

```
ER ercd = loc_mtx(ID mtxid);
ER ercd = ploc_mtx(ID mtxid);
ER ercd = tloc_mtx(ID mtxid, TMO tmout);
```

Parameters:

ID	mtxid	R0	Mutex ID
<tloc_mtx>			
TMO	tmout	ER1	Timeout specification

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_NOSPT	[p]	Unsupported function (Time management function is not used or timeout function is not used (tloc_mtx))
E_PAR	[p]	Parameter error (tmout ≤ –2)
E_ID	[p]	Invalid ID (mtxid ≤ 0, mtxid > CFG_MAXMTXID)
E_NOEXS	[p]	Undefined (Mutex indicated by mtxid does not exist)
E_ILUSE	[k]	Illegal use of service call (Calling task has already locked the mutex indicated by mtxid or calling task bpri > target mutex ceilpri)
E_CTX	[p]	Context error (Called from disabled system state)
	[k]	(Cannot be called (loc_mtx, tloc_mtx) from dispatch-disabled state or CPU-locked state and tmout in tloc_mtx is not TMO_POL (0))
E_RLWAI	[k]	WAITING state is forcibly cancelled (rel_wai was accepted in the WAITING state)
E_TMOUT	[k]	Polling failed or timeout

Function:

Service calls loc_mtx, ploc_mtx and tloc_mtx lock the mutex specified by parameter mtxid.

If the target mutex is not locked, the current task locks the mutex, and the service call processing is completed. At this time, the priority of the current task is raised to the ceiling priority of the mutex.

If the target mutex is locked, the current task is placed in a wait queue, and the current task enters the mutex lock-wait state. The wait queue is managed in priority order.

Parameter tmout specified by service call tloc_mtx specifies the timeout period.

If a positive value is specified for parameter tmout, error code E_TMOUT is returned when the timeout period has passed without the wait release conditions being satisfied. If tmout = TMO_POL (0) is specified, the same operation as for service call ploc_mtx will be performed. If tmout = TMO_FEVR (–1) is specified, timeout monitoring is not performed. In other words, the same operation as for service call loc_mtx will be performed.

When a value larger than 1 is specified for the denominator of time tick cycles (CFG_TICDENO), the maximum value that can be specified for tmout is H'7fffffff/CFG_TICDENO. If a value larger than this is specified, operation is not guaranteed.

5.10.2 Unlock Mutex (unl_mtx)

C-Language API:

```
ER ercd = unl_mtx(ID mtxid);
```

Parameters:

ID	mtxid	R0	Mutex ID

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID (mtxid ≤ 0, mtxid > CFG_MAXMTXID)
E_NOEXS	[p]	Undefined (Mutex indicated by mtxid does not exist)
E_ILUSE	[k]	Illegal use of service call (Duplicate lock of mutex or highest priority is exceeded)
E_CTX	[p]	Context error (Called from disabled system state)

Function:

The lock for the mutex specified by mtxid is released. If there are tasks waiting for the lock for the specified mutex, the WAITING state for the task at the head of the mutex wait queue is released, and the task whose WAITING state has been released is put into a state which locks the mutex. At this time, the priority of the locking task is raised to the ceiling priority of the mutex. If there are no tasks waiting for the mutex, the mutex is put into the unlocked state.

This kernel uses the simplified ceiling priority protocol for the TA_CEILING attribute, that is, as a result of this service call, only when there are no mutexes locked by the calling task, its priority is returned to the base priority. While the calling task locks another mutex, the current priority is not changed by this service call.

5.10.3 Refer to Mutex State (ref_mtx)

C-Language API:

```
     ER ercd = ref_mtx(ID mtxid, T_RMTX *pk_rmtx);
```

Parameters:

```
     ID             mtxid        R0    Mutex ID
     T_RMTX         *pk_rmtx     ER1   Pointer to the area where the mutex status
                                       is to be returned
```

Return Parameters:

```
     ER             ercd         R0    Normal termination (E_OK) or error code
     T_RMTX         *pk_rmtx     --    Pointer to the packet where the mutex
                                       status is stored
```

Packet Structure:

```
     typedef    struct    t_rmtx{
                ID        htskid;   +0    2    Task ID locking a mutex
                ID        wtskid;   +2    2    Start task ID of mutex waiting queue
     }T_RMTX;
```

Error Codes:

```
     E_PAR       [p]   Parameter error (pk_rmtx is 0)
     E_ID        [p]   Invalid ID number (mtxid ≤ 0 or mtxid > CFG_MAXMTXID)
     E_NOEXS     [p]   Undefined (Mutex indicated by mtxid does not exist)
```

Function:

Service call ref_mtx refers to the state of the mutex.

Service call ref_mtx returns the task ID that locks the mutex (htskid) and the start task ID of the mutex wait queue (wtskid) to the area indicated by pk_rmtx.

If there is no task that locks the target mutex, TSK_NOME (0) is returned to the htskid.

If there is no task waiting for the target mutex, TSK_NONE (0) is returned to the wtskid.

5.11 Memory Pool Management (Fixed-Size Memory Pool)

Fixed-size memory pools are controlled by the service calls listed in Table 5.22.

Table 5.22 Service Calls for Fixed-Size Memory Pool Control

			System State[*2]
Service Call[*1]		**Description**	**T/N/E/D/U/L/C**
get_mpf	[S]	Acquires fixed-size memory block	T/E/U
pget_mpf	[S]	Polls and acquires fixed-size memory block	T/E/D/U
ipget_mpf			N/E/D/U
tget_mpf	[S]	Acquires fixed-size memory block with timeout function	T/E/U
rel_mpf	[S]	Returns fixed-size memory block	T/E/D/U
ref_mpf		Refers to fixed-size memory pool state	T/E/D/U
iref_mpf			N/E/D/U

Notes: 1. [S]: Standard profile service calls

2. T: Can be called from task context
N: Can be called from non-task context
E: Can be called from dispatch-enabled state
D: Can be called from dispatch-disabled state
U: Can be called from CPU-unlocked state
L: Can be called from CPU-locked state
C: Can be called from CPU exception handler

The fixed-size memory pool specifications are listed in Table 5.23.

Table 5.23 Fixed-Size Memory Pool Specifications

Item	**Description**
Fixed-size memory pool ID	1 to CFG_MAXMPFID (255 max.)
Fixed-size memory pool count	1 to 65535
Fixed-size memory pool size	2 to 65530
Attribute supported	TA_TFIFO: Task wait queue is managed on a FIFO basis

5.11.1 Get Fixed-Size Memory Block (get_mpf, pget_mpf, ipget_mpf, tget_mpf)

C-Language API:

```
ER ercd = get_mpf(ID mpfid, VP *p_blk);
ER ercd = pget_mpf(ID mpfid, VP *p_blk);
ER ercd = ipget_mpf(ID mpfid, VP *p_blk);
ER ercd = tget_mpf(ID mpfid, VP *p_blk, TMO tmout);
```

Parameters:

ID	mpfid	R0	Fixed-size memory pool ID
VP	*p_blk	ER1	Pointer to the area where the start address of the memory block is to be returned
<tget_mpf>			
TMO	tmout	ER2	Timeout specification

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code
VP	*p_blk	--	Start address of the area where the start address of the memory block is stored

Error Codes:

E_NOSPT	[p]	Unsupported function (Time management function is not used or timeout function is not used (tget_mpf))
E_PAR	[p]	Parameter error (p_blk is 0, tmout ≤ -2)
E_ID	[p]	Invalid ID number (mpfid ≤ 0 or mpfid > CFG_MAXMPFID)
E_NOEXS	[p]	Undefined (Fixed-size memory pool indicated by mpfid does not exist)
E_CTX	[p]	Context error (Called from disabled system state)
	[k]	(Cannot be called (get_mpf, tget_mpf) from dispatch-disabled state or CPU-locked state and tmout in tget_mpf is not TMO_POL (0))
E_TMOUT	[k]	Polling failed or timeout
E_RLWAI	[k]	WAITING state is forcibly cancelled (rel_wai was accepted in the WAITING state)

Function:

Each service call gets one fixed-size memory block from the fixed-size memory pool indicated by mpfid, and returns the start address of the acquired memory block to the area indicated by p_blk.

If there are tasks already waiting for the memory pool, or if no task is waiting but there is no memory block available in the fixed-size memory pool, the task having called service call get_mpf or tget_mpf is placed in the memory acquiring wait queue, and the task having called service call pget_mpf or ipget_mpf is immediately returned with error code E_TMOUT. The queue is managed on a FIFO basis.

Parameter tmout of service call tget_mpf specifies the timeout period.

If a positive value is specified for parameter tmout, error code E_TMOUT is returned when the timeout period has passed without the wait release conditions being satisfied. If tmout = TMO_POL (0) is specified, the same operation as for service call pget_mpf will be performed. If tmout = TMO_FEVR (–1) is specified, timeout monitoring is not performed. In other words, the same operation as for service call get_mpf will be performed.

If a value larger than 1 is specified for the denominator of time tick cycles (CFG_TICDENO), the maximum value that can be specified for tmout is H'7fffffff/denominator for time tick cycles. If a value larger than this is specified, operation is not guaranteed.

5.11.2 Release Fixed-Size Memory Block (rel_mpf)

C-Language API:

```
ER ercd = rel_mpf(ID mpfid, VP blk);
```

Parameters:

ID	mpfid	R0	Fixed-size memory pool ID
VP	blk	ER1	Start address of memory block

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (blk is 0, blk is an odd value, blk is a value other than start address of memory block, or a returned blk is specified)
	[k]	(Specifies a value other than start address of memory block)
E_ID	[p]	Invalid ID number (mpfid ≤ 0 or mpfid > CFG_MAXMPFID)
E_NOEXS	[p]	Undefined (Fixed-size memory pool indicated by mpfid does not exist)
E_CTX	[p]	Context error (Called from disabled system state)

Function:

Each service call returns the memory block indicated by blk to the fixed-size memory pool indicated by mpfid.

The start address of the memory block acquired by service call get_mpf, pget_mpf, ipget_mpf, or tget_mpf must be specified for parameter blk.

If there are tasks waiting to get a memory block in the target fixed-size memory pool, the memory block returned by this service call is passed to the task at the head of the wait queue, releasing it from the WAITING state.

5.11.3 Refer to Fixed-Size Memory Pool State (ref_mpf, iref_mpf)

C-Language API:

```
ER ercd = ref_mpf(ID mpfid, T_RMPF *pk_rmpf);
ER ercd = iref_mpf (ID mpfid, T_RMPF *pk_rmpf);
```

Parameters:

ID	mpfid	R0	Fixed-size memory pool ID
T_RMPF	*pk_rmpf	ER1	Pointer to the packet where the fixed-size memory pool state is to be returned

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code
T_RMPF	*pk_rmpf	--	Pointer to the packet where the fixed-size memory pool state is stored

Packet Structure:

```
typedef    struct    t_rmpf{
           ID        wtskid;   +0    2    Wait task ID
           UINT      fblkcnt;  +2    2    Number of blocks of memory space
                                          available
}T_RMPF;
```

Error Codes:

E_PAR	[p]	Parameter error (pk_rmpf is 0)
E_ID	[p]	Invalid ID number (mpfid ≤ 0 or mpfid > CFG_MAXMPFID)
E_NOEXS	[p]	Undefined (Fixed-size memory pool indicated by mpfid does not exist)

Function:

Each service call refers to the state of the fixed-size memory pool indicated by mpfid.

Service calls ref_mpf and iref_mpf return the wait task ID (wtskid) and the number of blocks of memory space available (fblkcnt) to the area indicated by pk_rmpf.

If there is no task waiting for the specified memory pool, TSK_NONE (0) is returned as a wait task ID.

5.12 Memory Pool Management (Variable-Size Memory Pool)

Variable-size memory pools are controlled by the service calls listed in Table 5.24.

Table 5.24 Service Calls for Variable-Size Memory Pool Control

		System State[*2]
Service Call[*1]	**Description**	**T/N/E/D/U/L/C**
get_mpl	Acquires variable-size memory block	T/E/U
pget_mpl	Polls and acquires variable-size memory block	T/E/D/U
ipget_mpl		N/E/D/U
tget_mpl	Acquires variable-size memory block with timeout function	T/E/U
rel_mpl	Returns variable-size memory block	T/E/D/U
ref_mpl	Refers to variable-size memory pool state	T/E/D/U
iref_mpl		N/E/D/U

Notes: 1. [S]: Standard profile service calls

2. T: Can be called from task context
N: Can be called from non-task context
E: Can be called from dispatch-enabled state
D: Can be called from dispatch-disabled state
U: Can be called from CPU-unlocked state
L: Can be called from CPU-locked state
C: Can be called from CPU exception handler

The variable-size memory pool specifications are listed in Table 5.25.

Table 5.25 Variable-Size Memory Pool Specifications

Item	Description
Variable-size memory pool ID	1 to CFG_MAXMPLID (255 max.)
Variable-size memory pool size	18 to 65534
Attribute supported	TA_TFIFO: Task wait queue is managed on a FIFO basis

Fragmentation of Variable-Size Memory Pool: Repeated acquisition and return of memory blocks from the variable-size memory pool causes fragmentation of the available memory area in the memory pool; as a result, there is no contiguous memory area available even though the total size of the available memory area is sufficient, which results in a smaller maximum available contiguous memory area. When there are memory blocks that are not to be returned at the center of the memory pool, the size of the maximum available contiguous memory area will never be larger than a certain size because such a block behaves as a barrier. However, the kernel cannot de-fragment memory area. To avoid this problem, get a memory block that is not to be returned before any memory block with the possibility of being returned is acquired. This allows memory blocks to be allocated from the edge of the memory pool.

5.12.1 Get Variable-Size Memory Block (get_mpl, pget_mpl, ipget_mpl, tget_mpl)

C-Language API:

```
ER ercd = get_mpl (ID mplid, UINT blksz, VP *p_blk);
ER ercd = pget_mpl (ID mplid, UINT blksz, VP *p_blk);
ER ercd = ipget_mpl (ID mplid, UINT blksz, VP *p_blk);
ER ercd = tget_mpl (ID mplid, UINT blksz, VP *p_blk);
```

Parameters:

ID	mplid	R0	Variable-size memory pool ID
UINT	blksz	E0	Memory block size (Number of bytes)
VP	*p_blk	ER1	Pointer to the area where the start address of the memory block is to be returned
<tget_mpl>			
TMO	tmout	ER2	Timeout specification

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code
VP	*p_blk	--	Pointer to the area where the start address of the memory block is stored

Error Codes:

E_NOSPT	[p]	Unsupported function (Time management function is not used or timeout function is not used (tget_mpl))
E_PAR	[p]	Parameter error (p_blk is 0, blksz is other than a multiple of two, blksz is 0, tmout ≤ –2, or mplsz* – 16 < blksz)
E_ID	[p]	Invalid ID number (mplid ≤ 0 or mplid > CFG_MAXMPFID)
E_NOEXS	[p]	Undefined (Variable-size memory pool indicated by mplid does not exist)
E_CTX	[p]	Context error (Called from disabled system state)
	[k]	(Cannot be called (get_mpl, tget_mpl) from dispatch-disabled state or CPU-locked state and tmout in tget_mpl is not TMO_POL (0))
E_TMOUT	[k]	Polling failed or timeout
E_RLWAI	[k]	WAITING state is forcibly cancelled (rel_wai was accepted in the WAITING state)

Note: Memory pool size specified when the variable-size memory pool is initially defined

Function:

Each service call acquires a variable-size memory block with the size specified by blksz (number of bytes) from the variable-size memory pool indicated by mplid, and returns the start address of the acquired memory block to the area indicated by p_blk.

After the memory block has been acquired, the size of the variable-size memory pool free space will decrease by an amount given by the expression:

Decrease in size = blksz + 16 bytes

If there are tasks already waiting for the memory pool, or if no task is waiting but there is no memory block available, the task having called service call get_mpl or tget_mpl is placed in the memory block wait queue, and the task having called service call pget_mpl or ipget_mpl is immediately terminated with the error code E_TMOUT returned. The queue is managed on a first-in first-out (FIFO) basis.

Parameter tmout of service call tget_mpl specifies the timeout period.

If a positive value is specified for parameter tmout, error code E_TMOUT is returned when the timeout period has passed without the wait release conditions being satisfied. If tmout = TMO_POL (0) is specified, the same operation as for service call pget_mpl will be performed. If tmout = TMO_FEVR (–1) is specified, timeout watch is not performed. In other words, the same operation as for service call get_mpl will be performed.

If a value larger than 1 is specified for the denominator of time tick cycles (CFG_TICDENO), the maximum value that can be specified for tmout is H'7fffffff/denominator for time tick cycles. If a value larger than this is specified, operation is not guaranteed.

5.12.2 Release Variable-Size Memory Block (rel_mpl)

C-Language API:

```
ER ercd = rel_mpl(ID mplid, VP blk);
```

Parameters:

ID	mplid	R0	Variable-size memory pool ID
VP	blk	ER1	Start address of memory block

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (blk is 0 or an odd value)
	[k]	(blk is other than the memory block start address or blk has already been returned)
E_ID	[p]	Invalid ID number (mplid ≤ 0 or mplid > CFG_MAXMPLID)
E_NOEXS	[p]	Undefined (Variable-size memory pool indicated by mplid does not exist)
E_CTX	[p]	Context error (Called from disabled system state)

Function:

Each service call returns the memory block specified by blk to the variable-size memory pool specified by mplid.

The start address of the memory block acquired by service call get_mpl, pget_mpl, ipget_mpl, or tget_mpl must be specified as parameter blk.

After the memory block has been returned, the size of the variable-size memory pool free space will increase by an amount given by the expression:

Increase in size = (blksz specified at acquisition) + 16 bytes

When the target variable-size memory pool has a contiguous memory block requested by the task at the head of the memory block acquisition wait queue, the memory block is assigned to that task; as a result, the task is released from the WAITING state.

The same process will be done for the remaining tasks in the order of the wait queue if the remaining memory pool size still has enough contiguous memory blocks available.

5.12.3 Refer to Variable-Size Memory Pool State (ref_mpl, iref_mpl)

C-Language API:

```
ER ercd = ref_mpl (ID mplid, T_RMPL *pk_rmpl);
ER ercd = iref_mpl (ID mplid, T_RMPL *pk_rmpl);
```

Parameters:

ID	mplid	R0	Variable-size memory pool ID
T_RMPL	*pk_rmpl	ER1	Pointer to the packet where the variable-size memory pool state is to be returned

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code
T_RMPL	*pk_rmpl	--	Pointer to the packet where the variable-size memory pool state is stored

Packet Structure:

```
typedef    struct    t_rmpl{
           ID        wtskid;   +0    2    Wait task ID
           SIZE      fmplsz;   +2    4    Total size of available memory area
                                          (Number of bytes)
           UINT      fblksz;   +6    2    Maximum memory area available (Number
                                          of bytes)
}T_RMPL;
```

Error Codes:

E_PAR	[p]	Parameter error (pk_rmpl is 0)
E_ID	[p]	Invalid ID number (mplid ≤ 0 or mplid > CFG_MAXMPLID)
E_NOEXS	[p]	Undefined (Variable-size memory pool indicated by mplid does not exist)

Function:

Each service call refers to the status of the variable-size memory pool indicated by mplid and returns the wait task ID (wtskid), the current free memory area total size (fmplsz), and the maximum free memory space size (fblksz) to the area indicated by pk_rmpl.

The free space is usually fragmented. The maximum contiguous free space is returned to parameter fblksz. The block up to the size fblksz can be acquired immediately by calling service call get_mpl, pget_mpl, ipget_mpl, or tget_mpl.

If there is no task waiting to get a memory block, TSK_NONE (0) is returned as a wait task ID.

5.13 Time Management (System Clock)

System clock is controlled by the service calls listed in Table 5.26.

Table 5.26 Service Calls for System Clock Management

			System State[*2]
Service Call[*1]		**Description**	**T/N/E/D/U/L/C**
set_tim	[S]	Sets system clock	T/E/D/U
iset_tim			N/E/D/U
get_tim	[S]	Gets system clock	T/E/D/U
iget_tim			N/E/D/U

Notes: 1. [S]: Standard profile service calls
2. T: Can be called from task context
N: Can be called from non-task context
E: Can be called from dispatch-enabled state
D: Can be called from dispatch-disabled state
U: Can be called from CPU-unlocked state
L: Can be called from CPU-locked state
C: Can be called from CPU exception handler

Note that the HI1000/4 provides an original function to update the system clock.

The system clock management specifications are listed in Table 5.27.

Table 5.27 System Clock Management Specifications

Item	**Description**
System clock value	Unsigned 48 bits
System clock unit	1 ms
System clock update cycle	User-defined time tick update time [ms]
System clock initial value (at initialization)	H'000000000000

The system clock is expressed as 48-bit unsigned integer value by using the data type SYSTIM. The maximum value of the system clock is shown as follows.

[Case of time tick cycles ≤ 1]

Maximum value = H'ffffffffffff/denominator for time tick cycles (TIC_DENO)

[Case of time tick cycles > 1]

Maximum value = H'ffffffffffff

When the system clock exceeds the above maximum value at timer interrupt (isig_tim), the system clock is initialized to 0.

If a value larger than the above maximum value is specified in the set_tim service call, the system operation is not guaranteed.

5.13.1 Set System Clock (set_tim, iset_tim)

C-Language API:

```
ER ercd = set_tim (SYSTIM *p_systim);
ER ercd = iset_tim (SYSTIM *p_systim);
```

Parameters:

```
SYSTIM     *p_systim    ER0  Pointer to the packet where the current time
                             data is indicated
```

Return Parameters:

```
ER         ercd         R0   Normal termination (E_OK) or error code
```

Packet Structure:

```
typedef   struct   systim {
          UH       utime;   0     2    Current time data (upper)
          UW       ltime;   +2    4    Current time data (lower)
}SYSTIM;
```

Error Codes:

```
E_NOSPT   [p]   Unsupported function (Time management function is not used)
E_PAR     [p]   Parameter error (p_systim is 0)
```

Function:

Each service call changes the current system clock retained in the system to a value specified by p_systim.

If a value larger than 1 is specified for the denominator of time tick cycles (CFG_TICDENO), the maximum value that can be specified for tmout is H'ffffffff/denominator for time tick cycles. If a value larger than this is specified, operation is not guaranteed.

5.13.2 Get System Clock (get_tim, iget_tim)

C-Language API:

```
ER ercd = get_tim (SYSTIM *p_systim);
ER ercd = iget_tim (SYSTIM *p_systim);
```

Parameters:

```
SYSTIM        *p_systim     ER0  Start address of the packet where the current
                                 time data is to be returned
```

Return Parameters:

```
ER            ercd          R0   Normal termination (E_OK) or error code
SYSTIM        *p_systim     --   Start address of the packet where the current
                                 time data is stored
```

Packet Structure:

```
typedef    struct   systim {
           UH       utime;     0     2    Current time data (upper)
           UW       ltime;     +2    4    Current time data (lower)
}SYSTIM;
```

Error Codes:

```
E_NOSPT    [p]   Unsupported function (Time management function is not used)
E_PAR      [p]   Parameter error (p_systim is 0)
```

Function:

Each service call reads the current system clock and returns it to the area indicated by p_systim.

5.14 Time Management (Cyclic Handler)

Cyclic handler is controlled by the service calls listed in Table 5.28.

Table 5.28 Service Calls for Cyclic Handler

			System State[2]
Service Call[1]		**Description**	**T/N/E/D/U/L/C**
sta_cyc	[S]	Starts cyclic handler operation	T/E/D/U
ista_cyc			N/E/D/U
stp_cyc	[S]	Stops cyclic handler operation	T/E/D/U
istp_cyc			N/E/D/U
ref_cyc		Refers to the cyclic handler state	T/E/D/U
iref_cyc			N/E/D/U

Notes: 1. [S]: Standard profile service calls

2. T: Can be called from task context
N: Can be called from non-task context
E: Can be called from dispatch-enabled state
D: Can be called from dispatch-disabled state
U: Can be called from CPU-unlocked state
L: Can be called from CPU-locked state
C: Can be called from CPU exception handler

The cyclic handler specifications are listed in Table 5.29.

Table 5.29 Cyclic Handler Specifications

Item	Description
Cyclic handler ID	1 to CFG_MAXCYCID (254 max.)
Attribute supported	TA_HLNG: The task is written in a high-level language TA_ASM: The task is written in assembly language TA_STA: Starts cyclic handler operation TA_PHS: Reserves initiation phase

5.14.1 Start Cyclic Handler (sta_cyc, ista_cyc)

C-Language API:

```
ER ercd = sta_cyc (ID cycid);
ER ercd = ista_cyc (ID cycid);
```

Parameters:

```
ID              cycid       R0     Cyclic handler ID
```

Return Parameters:

```
ER              ercd        R0     Normal termination (E_OK) or error code
```

Error Codes:

```
E_NOSPT      [p]   Unsupported function (Time management function is not
                   used)
E_ID         [p]   Invalid ID number (cycid ≤ 0 or cycid > CFG_MAXCYCID)
E_NOEXS      [p]   Undefined (Cyclic handler specified by cycid does not
                   exist)
```

Function:

Each service call causes the cycle handler specified by cycid to enter the operation state.

If TA_PHS is not specified as a cyclic handler attribute, the cyclic handler is started each time the start cycle has passed, based on the timing at which the service calls are called.

If the cyclic handler specified by cycid is in the operating state and TA_PHS is not specified as its attribute, only the next timing of initiation is set after the service call is called.

If the cyclic handler specified by cycid is in the operating state and TA_PHS is specified as its attribute, the next timing of initiation is not set because the cyclic handler will be started based on the timing when the cyclic handler was initially defined.

5.14.2 Stop Cyclic Handler (stp_cyc, istp_cyc)

C-Language API:

```
ER ercd = stp_cyc (ID cycid);
ER ercd = istp_cyc (ID cycid);
```

Parameters:

ID	cycid	R0	Cyclic handler ID

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_NOSPT	[p]	Unsupported function (Time management function is not used)
E_ID	[p]	Invalid ID number (cycid ≤ 0 or cycid > CFG_MAXCYCID)
E_NOEXS	[p]	Undefined (Cyclic handler specified by cycid does not exist)

Function:

Each service call causes the cyclic handler indicated by parameter cycid to enter the not-operating state.

5.14.3 Refer to Cyclic Handler State (ref_cyc, iref_cyc)

C-Language API:

```
ER ercd = ref_cyc (ID cycid, T_RCYC *pk_rcyc);
ER ercd = iref_cyc (ID cycid, T_RCYC *pk_rcyc);
```

Parameters:

ID	cycid	R0	Cyclic handler ID
T_RCYC	*pk_rcyc	ER1	Pointer to the packet where the cyclic handler state is to be returned

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code
T_RCYC	*pk_rcyc	--	Pointer to the packet where the cyclic handler state is stored

Packet Structure:

```
typedef    struct    t_rcyc{
           STAT      cycstat;  +0    2    Cyclic handler operating state
           RELTIM    lefttim ; +2    4    Time until the cyclic handler is
                                          next initiated
}T_RCYC;
```

Error Codes:

E_NOSPT	[p]	Unsupported function (Time management function is not used)
E_PAR	[p]	Parameter error (pk_rcyc is 0)
E_ID	[p]	Invalid ID number (cycid ≤ 0 or cycid > CFG_MAXCYCID)
E_NOEXS	[p]	Undefined (Cyclic handler specified by cycid does not exist)

Function:

Each service call reads the cyclic handler state indicated by cycid and returns the cyclic handler operation state (cycstat) and the time until the cyclic handler is next initiated (lefttim), to the area indicated by parameter pk_rcyc.

The target cyclic handler operation state is returned to parameter cycstat.

Table 5.30 Handler Initiation State (cycstat)

cycstat	Code	Description
TCYC_STP	H'0000	The cyclic handler is not in the operating state
TCYC_STA	H'0001	The cyclic handler is in the operating state

The relative time until the target cyclic handler is next initiated is returned to parameter lefttim. When the target cyclic handler is not initiated, lefttim is undefined.

5.15 System State Management

System state is controlled by the service calls listed in Table 5.31.

Table 5.31 Service Calls for System State Management

Service Call[*1]		Description	System State[*2] T/N/E/D/U/L/C
rot_rdq	[S]	Rotates ready queue	T/E/D/U
irot_rdq	[S]		N/E/D/U
get_tid	[S]	Refers to task ID in RUNNING state	T/E/D/U
iget_tid	[S]		N/E/D/U/C
loc_cpu	[S]	Locks CPU	T/E/D/U/L
iloc_cpu	[S]		N/E/D/U/L
unl_cpu	[S]	Unlocks CPU	T/E/D/U/L
iunl_cpu	[S]		N/E/D/U/L
dis_dsp	[S]	Disables task dispatch	T/E/D/U
ena_dsp	[S]	Enables task dispatch	T/E/D/U
sns_ctx	[S]	Refers to task context	T/N/E/D/U/L/C
sns_loc	[S]	Refers to CPU-locked state	T/N/E/D/U/L/C
sns_dsp	[S]	Refers to dispatch-disabled state	T/N/E/D/U/L/C
sns_dpn	[S]	Refers to dispatch-pended state	T/N/E/D/U/L/C
vsta_knl	[s]	Starts kernel	T/E/D/U/L
ivsta_knl	[s]		N/E/D/U/L/C
vsys_dwn	[s]	Terminates the system	T/E/D/U/L
ivsys_dwn	[s]		N/E/D/U/L/C
ivbgn_int		Acquires start of interrupt handler to trace	N/E/D/U
ivend_int		Acquires end of interrupt handler to trace	N/E/D/U

Notes: 1. [S]: Standard profile service calls
[s]: Service calls that are not standard profile service calls but are needed in order to use the standard profile function

2. T: Can be called from task context
N: Can be called from non-task context
E: Can be called from dispatch-enabled state
D: Can be called from dispatch-disabled state
U: Can be called from CPU-unlocked state
L: Can be called from CPU-locked state
C: Can be called from CPU exception handler

5.15.1 Rotate Ready Queue (rot_rdq, irot_rdq)

C-Language API:

```
ER ercd = rot_rdq(PRI tskpri);
ER ercd = irot_rdq(PRI tskpri);
```

Parameters:

PRI	tskpri	R0	Task priority

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (tskpri < 0, tskpri > CFG_MAXTSKPRI, or tskpri = TPRI_SELF(0) is specified in a non-task context)
E_CTX	[p]	Context error (Called from disabled system state)

Function:

Each service call rotates the ready queue of the task priority indicated by parameter tskpri. In other words, the task at the head of the task priority ready queue is sent to the end of the queue, enabling the second task in the ready queue to be executed.

Specifying tskpri = TPRI_SELF (0) rotates the ready queue with the base priority of the current task. The base priority is the same as the current priority when the mutex function is not used; however, the current priority is not the same as the base priority while the mutex is locked. Thus, while the mutex is locked, the ready queue with the priority for the current task cannot be rotated even when TPRI_SELF is specified.

5.15.2 Get Task ID in RUNNING state (get_tid, iget_tid)

C-Language API:

```
ER ercd = get_tid(ID *p_tskid);
ER ercd = iget_tid(ID *p_tskid);
```

Parameters:

```
ID          *p_tskid    ER0   Pointer to the area where the task ID is
                              to be returned
```

Return Parameters:

```
ER          ercd        R0    Normal termination (E_OK) or error code
ID          *p_tskid    --    Pointer to the task ID
```

Error Codes:

```
E_PAR      [p]   Parameter error (p_tskid is 0)
```

Function:

Each service call gets the task ID in the RUNNING state and returns it to the area indicated by p_tskid. If each service call is called from task context, the current task ID is returned. If each service call is called from non-task context, the task ID that is being executed is returned. If there is no task in the RUNNING state, TSK_NONE (0) is returned.

Service calls get_tid and iget_tid can also be called from the CPU exception handler.

5.15.3 Lock CPU (loc_cpu, iloc_cpu)

C-Language API:

```
ER ercd = loc_cpu( );
ER ercd = iloc_cpu( );
```

Parameters:

```
None
```

Return Parameters:

```
ER              ercd         R0    Normal termination (E_OK)
```

Error Codes:

```
None
```

Function:

Each service call locks the CPU and disables interrupts and task dispatches.

The following describes the CPU-locked state:

- Tasks cannot be scheduled while the CPU is locked.
- Interrupts having a level equal to or lower than the kernel interrupt mask level are disabled.
- Only the following service calls can be called from the CPU-locked state. The system operation cannot be guaranteed when a service call other than the followings is called:
  - ext_tsk
  - loc_cpu, iloc_cpu
  - unl_cpu, iunl_cpu
  - sns_ctx
  - sns_loc
  - sns_dsp
  - sns_dpn
  - vsta_knl, ivsta_knl
  - vsys_dwn, ivsys_dwn

When the following service calls are called in the CPU locked state, the system returns to the CPU unlocked state.

- unl_cpu or iunl_cpu
- ext_tsk

The transition between the CPU-locked state and CPU-unlocked state occurs only when the service call loc_cpu, iloc_cpu, unl_cpu, iunl_cpu, or ext_tsk is called.

An interrupt handler, time event handler, or initialization routine, whose level is equal to or lower than the kernel interrupt mask level must unlock the CPU at termination. If the CPU is locked at termination, normal system operation cannot be guaranteed. Note that the CPU at the start of these handlers is unlocked.

If the CPU exception handler changes the CPU-locked/unlocked state, the handler must return to the former state before termination. If execution does not return to the former state, normal system operation cannot be guaranteed.

An error will not occur when service calls loc_cpu and iloc_cpu are called while the CPU is locked but queuing will not be done.

5.15.4 Unlock CPU (unl_cpu, iunl_cpu)

C-Language API:

```
ER ercd = unl_cpu( );
ER ercd = iunl_cpu( );
```

Parameters:

```
None
```

Return Parameters:

```
ER              ercd         R0    Normal termination (E_OK) or error code
```

Error Codes:

```
E_CTX           [p]    Context error (Called from disabled system state)
```

Function:

Each service call unlocks the CPU, which was locked by the service call loc_cpu or iloc_cpu. If the CPU enters the task-context dispatch-enabled state by the service call, task scheduling is performed.

When the system makes a transition to the CPU-locked state by calling service call iloc_cpu in the interrupt handler, service call iunl_cpu must be called to unlock the CPU before returning from the interrupt handler.

The CPU-locked state and dispatch-disabled state are managed individually. Thus, service call unl_cpu or iunl_cpu does not enable task dispatch by calling service call ena_dsp.

An error will not occur when service calls unl_cpu and iunl_cpu are called while the CPU is unlocked but queuing will not be done.

5.15.5 Disable Dispatch (dis_dsp)

C-Language API:

```
ER ercd = dis_dsp();
```

Parameters:

```
None
```

Return Parameters:

```
ER              ercd          R0    Normal termination (E_OK) or error code
```

Error Codes:

```
E_CTX           [p]    Context error (Called from disabled system state)
```

Function:

Service call dis_dsp disables task dispatch.

The following describes the dispatch-disabled state:

- Task scheduling is delayed, so that a task other than the current task cannot enter the RUNNING state.
- Interrupts can be accepted.
- Service calls to shift a task to the WAITING state cannot be called.

When the following service calls are called while task dispatch is disabled, the system returns to the task dispatch-enabled state.

- ena_dsp
- ext_tsk

When task dispatch is disabled, the task state is undefined. Therefore, note that if the current task refers to its state by calling the service call ref_tsk, the returned state is not always the RUNNING state.

The transition between the dispatch-disabled state and dispatch-enabled cancel state occurs only when service call dis_dsp, ena_dsp, or ext_tsk is called.

If the CPU exception handler changes the dispatch-disabled/enabled state, the handler must return to the former state before termination. If execution does not return to the former state, normal system operation cannot be guaranteed.

An error will not occur when the service call dis_dsp is called while task dispatch is disabled but queuing will not be done.

5.15.6 Enable Dispatch (ena_dsp)

C-Language API:

```
ER ercd = ena_dsp( );
```

Parameters:

```
None
```

Return Parameters:

```
ER              ercd         R0    Normal termination (E_OK) or error code
```

Error Codes:

```
E_CTX           [p]   Context error (Called from disabled system state)
```

Function:

The service call ena_dsp enables task dispatch disabled by service call dis_dsp. Task scheduling is then performed after the service call.

An error will not occur when the service call ena_dsp is called while task dispatch is enabled but queuing will not be done.

5.15.7 Refer to Context (sns_ctx)

C-Language API:

```
BOOL state = sns_ctx( );
```

Parameters:

```
None
```

Return Parameters:

```
BOOL            state        R0    Context
```

Function:

TRUE is returned when service call sns_ctx is called from non-task context. FALSE is returned when service call sns_ctx is called from task context.

Service call sns_ctx can be called in the CPU-locked state and from the CPU exception handler.

5.15.8 Refer to CPU-Locked State (sns_loc)

C-Language API:

```
BOOL state = sns_loc( );
```

Parameters:

```
None
```

Return Parameters:

```
BOOL             state        R0      CPU-locked state
```

Function:

Service call sns_loc returns TRUE when the CPU is locked. Service call sns_loc returns FALSE when the CPU is unlocked.

Service call sns_loc can be called in the CPU-locked state and from the CPU exception handler.

5.15.9 Refer to Dispatch-Disabled State (sns_dsp)

C-Language API:

```
BOOL state = sns_dsp( );
```

Parameters:

```
None
```

Return Parameters:

```
BOOL            state        R0      Dispatch-disabled state
```

Function:

Service call sns_dsp returns TRUE when task dispatch is disabled. Service call sns_dsp returns FALSE when task dispatch is enabled.

Service call sns_dsp can be called in the CPU-locked state and from the CPU exception handler.

5.15.10 Refer to Dispatch-Pended State (sns_dpn)

C-Language API:

```
BOOL state = sns_dpn( );
```

Parameters:

```
None
```

Return Parameters:

```
BOOL            state        R0     Dispatch-pended state
```

Function:

Service call sns_dpn returns TRUE when the task dispatch is pended. Otherwise, service call sns_dpn returns FALSE.

When the following conditions are satisfied, FALSE is returned. Otherwise, TRUE is returned.

- Task dispatch is not disabled.
- The CPU is unlocked.
- Task
- An interrupt is not masked by service call chg_ims.

Service call sns_dpn can be called in the CPU-locked state and from the CPU exception handler.

5.15.11 Start Kernel (vsta_knl, ivsta_knl)

C-Language API:

```
void vsta_knl( );
void ivsta_knl( );
```

Assembler API:

```
Branches to symbol "_vsta_knl"
Branches to symbol "_ivsta_knl"
```

Parameters:

```
None
```

Return Parameters:

```
Service call vsta_knl or ivsta_knl does not return any parameters to the
current task.
```

Function:

Service call vsta_knl starts the kernel.

If the kernel has already been started, the multitasking environment up to that point is all nullified.

This service call can also be called in the CPU-locked state and from the CPU exception handler. It can also be called before the kernel is started.

This service call should be called in a state with all interrupts masked.

An application program calling this service call must be linked with the kernel.

This service call is a function original to the HI1000/4.

5.15.12 System Down (vsys_dwn, ivsys_dwn)

C-Language API:

```
void vsys_dwn (H type, H inf1, B inf2, B inf3, H inf4, UW inf5);
void ivsys_dwn (H type, H inf1, B inf2, B inf3, H inf4, UW inf5);
```

Parameters:

```
H            type       R0     Error type
H            inf1       E0     System down information 1
B            inf2       R1L    System down information 2
B            inf3       R1H    System down information 3
H            inf4       E1     System down information 4
UW           inf5       ER2    System down information 5
```

Return Parameters:

```
Service call vsys_dwn or ivsys_dwn is not returned.
```

Function:

Service call vsys_dwn or ivsys_dwn passes control to the system down routine.

A value (H’1 to H'7fff) corresponding to the error type must be specified for the parameter type. Values other than those are reserved for future expansion.

The user must define system down information 1 to system down information 5 used when service call vsys_down or ivsys_dwn is issued by the user. For details on system down information for a system down caused by a system error, refer to Section 10, Information during System Down.

When accessing system down information, identify from the error type whether the system down was caused by a system error or was called by the user.

Service call vsys_dwn or ivsys_dwn can be called in the CPU-locked state and from the CPU exception handler.

This service call is a function original to the HI1000/4.

5.15.13 Acquire Start of Interrupt Handler as Trace Information (ivbgn_int)

C-Language API:

```
ER ercd = ivbgn_int(UINT dintno);
```

Parameters:

```
UINT            dintno       R0    Interrupt handler number
```

Return Parameters:

```
ER              ercd         R0    Normal termination (E_OK)
```

Error Codes:

```
None
```

Function:

The beginning of processing of the interrupt handler for the interrupt handler number specified by dintno is traced.

The interrupt handler number is the CPU vector number.

This service call should be called at the beginning of an interrupt handler. In addition, it should always be used in combination with ivend_int.

An error does not result if it is called from code other than an interrupt handler, but in such cases there is the possibility that the debugging extension trace display may be illegal.

If the trace function is not included, this service call does not perform any processing.

This service call is a function original to the HI1000/4.

This service call must be called at the beginning of every interrupt handler when the RTOS debugging function of the E6000H is used.

5.15.14 Acquire End of Interrupt Handler as Trace Information (ivend_int)

C-Language API:

```
ER ercd = ivend_int(UINT dintno);
```

Parameters:

```
UINT            dintno       R0    Interrupt handler number
```

Return Parameters:

```
ER              ercd         R0    Normal termination (E_OK)
```

Error Codes:

```
None
```

Function:

The end of processing of the interrupt handler for the interrupt handler number specified by dintno is traced.

The interrupt handler number is the CPU vector number.

This service call should be called at the end of an interrupt handler. In addition, it should always be used in combination with ivbgn_int.

An error does not result if it is called from code other than an interrupt handler, but in such cases there is the possibility that the debugging extension trace display may be illegal.

If the trace function is not installed, this service call does not perform any processing.

This service call is a function original to the HI1000/4.

This service call must be called at the end of every interrupt handler when the RTOS debugging function of the E6000H is used.

5.16 Interrupt Management

Interrupts are controlled by the service calls listed in Table 5.32.

Table 5.32 Service Calls for Interrupt Management

		System State[*2]
Service Call[*1]	**Description**	**T/N/E/D/U/L/C**
chg_ims	Changes interrupt mask	T/E/U
ichg_ims		N/E/U
get_ims	Refers to interrupt mask	T/E/D/U
iget_ims		N/E/D/U

Notes: 1. [S]: Standard profile service calls

2. T: Can be called from task context
N: Can be called from non-task context
E: Can be called from dispatch-enabled state
D: Can be called from dispatch-disabled state
U: Can be called from CPU-unlocked state
L: Can be called from CPU-locked state
C: Can be called from CPU exception handler

5.16.1 Interrupt Control Modes and Interrupt Mask Levels

The interrupt mask levels of each interrupt control mode are listed in Table 5.33. For details on the interrupt control modes, CCR and EXR values, and acceptable interrupts, refer to the corresponding hardware manual.

Table 5.33 Interrupt Control Modes and Interrupt Mask Levels

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

Note: — indicates an undetermined value.

5.16.2 Change Interrupt Mask (chg_ims, ichg_ims)

C-Language API:

```
ER ercd = chg_ims(IMASK imask);
ER ercd = ichg_ims(IMASK imask);
```

Parameters:

```
IMASK      imask     R0    Interrupt mask value
                           CFG_INTMD = 0:CR_IMS0 to CR_IMS1 (H'0 to H'1)
                           CFG_INTMD = 1:CR_IMS0 to CR_IMS3 (H'0 to H'3)
                           CFG_INTMD = 2:CR_IMS0 to CR_IMS7 (H'0 to H'7)
                           CFG_INTMD = 3:CR_IMS0 to CR_IMS8 (H'0 to H'8)
```

Return Parameters:

```
ER         ercd      R0    Normal termination (E_OK) or error code
```

Error Codes:

```
E_PAR       [p]   Parameter error (A value outside the range was specified
                  for imask)
E_CTX       [k]   Context error (Cannot be issued from CPU-locked state)
```

Function:

Each service call changes the current interrupt mask level to the level specified by imask. CR_IMSn (n: 0 to 8) is specified for imask according to the interrupt control mode (CFG_INTMD).

By directly writing values to CCR and EXR, the interrupt mask level can be set to disable or enable interrupts.

For the correspondence between the interrupt mask level (imask) and the CCR and EXR values, refer to Table 5.33.

When an interrupt is masked by the service call chg_ims or ichg_ims, a transition is made to the non-task context state. Accordingly, a service call that makes a transition to the WAITING state or a service call dedicated to task context cannot be issued.

When the service call chg_ims or ichg_ims causes a transition from the RUNNING state to the non-task context state, the interrupt mask needs to be cancelled with the service call chg_ims or ichg_ims for the task to return to the RUNNING state. Even if a service call needs to be called for task switching when execution is in the non-task context state, calling of the service call is delayed until the interrupt mask is returned to CR_IMS0 (H'0) with the service call chg_ims or ichg_ims.

Note: Service calls cannot be called while the interrupt mask level is being made higher than the kernel interrupt mask level (CFG_KNLMSKLVL) unless this service call is used to lower the interrupt mask level to a level equal to or below the kernel interrupt mask level. Otherwise, normal system operation cannot be guaranteed.

5.16.3 Refer to Interrupt Mask (get_ims, iget_ims)

C-Language API:

```
ER ercd = get_ims(IMASK *p_imask);
ER ercd = iget_ims(IMASK *p_imask);
```

Parameters:

IMASK	*p_imask	ER0	Start address of the area where the interrupt mask level is to be returned

Return Parameters:

ER	ercd	R0	Normal termination (E_OK) or error code
IMASK	*p_imask	--	Start address of the area where the interrupt mask level is stored

Error Codes:

E_PAR	[p]	Parameter error (p_imask is 0)

Function:

Each service call returns the current interrupt mask level.

The range and contents of the value that can be used as the interrupt mask level differ according to the interrupt control mode (CFG_INTMD).

5.17 System Configuration Management

System configuration is controlled by the service calls listed in Table 5.34.

Table 5.34 Service Calls for System Configuration Management

		System State[*2]
Service Call[*1]	**Description**	**T/N/E/D/U/L/C**
ref_ver	Refers to version information	T/E/D/U
iref_ver		N/E/D/U

Notes: 1. [S]: Standard profile service calls

2. T: Can be called from task context
N: Can be called from non-task context
E: Can be called from dispatch-enabled state
D: Can be called from dispatch-disabled state
U: Can be called from CPU-unlocked state
L: Can be called from CPU-locked state
C: Can be called from CPU exception handler

5.17.1 Refer to Version Information (ref_ver, iref_ver)

C-Language API:

```
ER ercd = ref_ver (T_RVER *pk_rver);
ER ercd = iref_ver (T_RVER *pk_rver);
```

Parameters:

```
T_RVER        *pk_rver     ER0    Pointer to the packet where version
                                  information is to be returned
```

Return Parameters:

```
ER            ercd         R0     Normal termination (E_OK) or error code
T_RVER        *pk_rver     --     Pointer to the packet where version
                                  information is stored
```

Packet Structure:

```
typedef    struct    t_rver {
           UH        maker;    0    2    Kernel manufacturer code
           UH        prid;     +2   2    Kernel ID number
           UH        spver;    +4   2    ITRON specification version number
           UH        prver;    +6   2    Kernel version number
           UH        prno[4];  +8   8    Kernel product management information
} T_RVER;
```

Error Codes:

```
E_PAR      [p]   Parameter error (pk_rver is 0)
```

Function:

Each service call refers to information on the version of the kernel used and returns it to the area indicated by pk_rver.

The following information is returned to the packet indicated by pk_rver.

- maker

  Indicates the manufacturer of this product.
  The maker value of this kernel is H'0115.
- prid

  Indicates the number to identify the OS or VLSI type as follows. The kernel id is shown.
  — HI1000/4: H'0011
- spver

  Indicates the specifications to which the kernel conforms to, as follows.
  — Bits 15 to 12: MAGIC (Number to identify the TRON specification series)
  H'5 (µITRON specifications) for this kernel

— Bits 11 to 0: SpecVer (Version number of the TRON specification on which the product is based)
H'401 (µITRON version 4.01.00) for this kernel

- prver

  Indicates the version number of this product.

  The prver value of this kernel is H’0140, which stands for version 1.04.00.

- prno

  Indicates the product management information and the product number.
  The prno[0] to prno[3] values of this kernel are all H'0000.

Section 6 Application Program Creation

6.1 Header Files

6.1.1 Standard Header File

The kernel has the following standard header files:

- itron.h (for C language)

  `itron.h` is a header file where the common ITRON specification definitions are described for C/C++ language. This file can be found in the hihead folder.
- itron.inc (for assembly language)

  `itron.inc` is a header file where the ITRON specification common definition is described for assembly language. This file can be found in the hihead folder.
- kernel.h and kernel_macro.h (for C language)

  `kernel.h` is a header file where the μITRON4.0 kernel specification definition is described for C/C++ language. `kernel.h` includes itron.h and kernel_macro.h that is output from the configurator. `kernel.h` can be found in the hihead folder.
- kernel.inc and kernel_macro.inc (for assembly language)

  `kernel.inc` is a header file where the μITRON4.0 kernel specification definition is described for assembly language. `kernel.inc` includes itron.inc and kernel_macro.inc that is output from the configurator. `kernel.inc` can be found in the hihead folder.

Section 5, Service Calls, describes the data types, constants, and macros defined in the above header files. Note however that the above header files include some constants and macros that are not described in Section 5, Service Calls. These are summarized in Table 6.1. For details, refer to the description on each header file.

Table 6.1 Constants and Macros

File Name	Macro and Constants	Description
kernel.h, kernel.inc	TMIN_TPRI	Lowest task priority (always 1)
	TMIN_MPRI	Lowest message priority (always 1)
	TKERNEL_MAKER	Kernel manufacturer code
	TKERNEL_PRID	Kernel ID number
	TKERNEL_SPVER	ITRON specification version number
	TKERNEL_PRVER	Kernel version number
	TMAX_ACTCNT	Maximum number of task initiation request queues (always 255)
	TMAX_WUPCNT	Maximum number of task wake-up request queues (always 255)
	TMAX_SUSCNT	Maximum number of nestings for task forced wait request (always 1)
	TBIT_FLGPTN	Number of event flag bits (always 16)
	TMAX_MAXSEM	Maximum number of resources in the semaphore (always 65535)
kernel_macro.h kernel_macro.inc	TIC_NUME	Numerator of time tick cycles
	TIC_DENO	Denominator of time tick cycles
	TMAX_TPRI	Highest task priority
	TMAX_MPRI	Highest message priority
	VTKNL_LVL	Kernel interrupt mask level
	VTIM_LVL	Timer interrupt mask level

6.2 Handling the CPU Resources

6.2.1 VBR and SBR Registers

To use the H8SX, initialize VBR and SBR by the application program before the kernel is initiated. To change the VBR contents after the kernel has been initiated, an interrupt vector table must be created at the address specified in VBR by copy or another method before the VBR contents are changed.

The SBR must be initialized when the SBR option is specified in HEW. The same value as set for the SBR option must be specified at initialization.

6.3 Reserved Names

External definition names beginning with _kernel_, _KERNEL_, and hi_ are reserved for the kernel, and cannot be used in application programs written in C language.

6.4 Tasks

Writing a Task: Figure 6.1 shows an example of writing a task as a function written in C language. For details, refer to the sample file, task.c. Use an ext_tsk service call to end a task. If the task is returned without issuing ext_tsk, ext_tsk is assumed to be issued and the same operation as when ext_tsk is issued is performed.

```
#include “kernel.h”             ← (a)
#include “kernel_id.h”

#pragma noregsave(Task)         ← (b)
void Task(VP_INT exinf)         ← (c)
{
    ext_tsk( );                 ← (d)
}                               ← (e)
```

Figure 6.1 Example of a C Language Task

(a) The kernel.h includes itron.h and kernel_macro.h which are output by the configurator.

(b) #pragma noregsave can be used because the task function does not need to guarantee register contents.

(c) When a task is initiated by act_tsk, passes exinf specified when the task is initially defined as a parameter; when a task is initiated by sta_tsk, passes stacd specified by sta_tsk as a parameter.

(d) Uses an ext_tsk service call to end a task.

(e) Calls ext_tsk automatically at the end of a task function.

Context Registers: The contents of the registers used in a task are saved and restored even if a task switch or an interrupt occurs. The registers that are saved and restored are called context registers. Note that CPU registers other than context registers must not be changed.

Context registers are determined according to the library used. For details, refer to Section 8.1, Supplied Kernel Libraries. Table 6.2 shows the context registers used by a task and their initialized contents when a task is initiated. Refer to the register contents shown in Table 6.2 also when writing a program in assembly language.

Note that some context register contents cannot be guaranteed after service calls. For details refer to Section 5.2.5, Register Contents Guaranteed after Calling Service Call.

Table 6.2 Task Context Registers and their Initialized Contents

Item	Initialized Contents
Program counter (PC)	Task start address specified at the task definition
Condition code register (CCR)	Interrupt mask released (0)
Extended register (EXR)	Interrupt mask released (0)
Stack pointer (ER7)	Pointer to the stack area of the task specified at the task definition
ER0 (assembly language)/first parameter (C language)	Extended information (exinf) of the task for act_tsk and the specified start code (stacd) for sta_tsk
General registers (ER1 to ER6) and multiply-and-accumulate registers (MACH and MACL)	Undefined (MACH and MACL are used when the function is selected)
Short address base register (SBR) (H8SX only)	Holds the SBR value at the kernel reset
Task base priority	Initial task priority specified at the task definition
Task current priority	Initial task priority specified at the task definition
Task wake-up request queues	0
Task suspend request nestings	0

Creating a Task: Tasks can be created by initially defining them by the configurator at system configuration. For details, refer to Section 7, Configuration.

6.5 Interrupt Handlers

6.5.1 Interrupt Handler Creation

Control passes to an interrupt handler without kernel intervention when an interrupt occurs. The register contents when the interrupt occurred need to be guaranteed by the interrupt handler.

An interrupt handler is normally created with the following procedure.

1. Save the contents of the registers to be used by the interrupt handler.
   — Save the stack pointer.
   - Change the stack pointer to indicate the stack area dedicated for the interrupt handlers. (This is not necessary when the interrupt handler does not use the stack.)
   - Save the register contents.
2. Perform the interrupt processing.
3. Restore the contents of the registers used by the interrupt handler.
   — Restore the contents of the registers.
   — Change the stack pointer. (This is not necessary when the interrupt handler does not use the stack.)
4. Call the ret_int routine (when equal to or lower than the kernel interrupt mask level) or execute the RTE instruction (when higher than the kernel interrupt mask level).

An interrupt handler can be written in C language by using the function (#pragma interrupt) to create an interrupt function of the C compiler.

The function that becomes the interrupt handler is declared using #pragma interrupt. The function Inhhdr is declared as the interrupt handler in Figure 6.2.

“Stack switching” and “interrupt function end” are specified as the interrupt specifications.

The stack switching specification sets the stack area to be switched to at the start of the interrupt handler, and the initial stack pointer is specified with “sp = <address>”. Specify a specific area for each interrupt level as the stack area.

The interrupt function end specification selects the method for returning when the interrupt handler terminates, and either the ret_int routine must be called or the RTE instruction must be executed when the interrupt handler terminates.

If the interrupt handler has an interrupt mask level equal to or lower than the kernel interrupt mask level, “sy = $ret_int” is specified. This causes the instruction “jmp @ret_int” to be executed when the interrupt handler terminates.

If the interrupt handler has an interrupt mask level higher than the kernel interrupt mask level, nothing needs to be written.

For details, refer to the H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User’s Manual.

An interrupt handler is shown in Figure 6.2.

Writing an Interrupt Handler: An interrupt handler is written as an interrupt function, as shown in Figure 6.2.

```
#include “kernel.h”
extern VP int_stk001;                                  ← (a)
static const VP p_stk=(VP)&int_stk001;                 ← (b)
#pragma interrupt(Inhhdr(sp=p_stk,sy=$ret_int))        ← (c)
void Inhhdr(void)                                      ← (d)
{
    //ivbgn_int(88); // call ivbgn_int(88)             ← (e)
    /* Interrupt handler processing */
    //ivend_int(88); // call ivend_int(88)
```

Figure 6.2 Example of a C-Language Interrupt Handler

(a) Specify the interrupt stack allocated by the configurator.

(b) Defines the initial stack pointer value as const type.
Interrupt handlers of the same interrupt level can share the same stack. Note that the NMI interrupt handler cannot have a dedicated stack.

(c) Declares the interrupt handler as an interrupt function by using #pragma interrupt. The following must be defined as interrupt specifications:

— Stack switching (sp=p_stk)
Do not switch stacks when the NMI interrupt handler is used.

— Interrupt function end (sy=$ret_int)
Specify sy=$ret_int for interrupt handlers with interrupt levels equal to or lower than the kernel interrupt mask level. No interrupt function end should be described for interrupt handlers (including NMI) with interrupt levels higher than the kernel interrupt mask level because such interrupt handlers must be returned by the RTE instruction.

(d) Describes the interrupt handler as a void-type function.

(e) Specifies the start and end of interrupt processing (ivbgn_int and ivend_int). They are necessary when using the RTOS debugging function of the E6000H.

Conditions for Interrupt Handler Processing: An interrupt handler must save and restore all register contents. Table 6.3 lists the conditions for interrupt handler processing.

Table 6.3 Interrupt Handler Processing Conditions

Item	Description
Interrupt mask	Initiated with the interrupts masked.
Usable registers	ER0 to ER6, MACH, and MACL (MACH and MACL are used when the function is selected) can be used. The register contents must be returned to the values at initiation before the interrupt handler processing terminates.
Stack pointer	The SP value must be returned to the value at initiation when control is returned to the interrupt factor.
Usable service calls	Service calls that can be called from non-task context. Note that service calls cannot be called by interrupt handlers with interrupt levels higher than the kernel interrupt mask level, and the NMI interrupt handler.
Usable stack area	Allocate a stack area by the configurator, and switch to the allocated interrupt stack at initiation. Interrupt handlers of the same interrupt level can share the same stack.
Termination method	Terminated by the ret_int routine. Return the stack pointer to the state at initiation when the interrupt handler processing terminates. Interrupt handlers with interrupt levels higher than the kernel interrupt mask level, and the NMI interrupt handler must be terminated with the RTE instruction.

When creating an interrupt handler note the following items.

Guaranteeing Interrupt Mask Level: The kernel can use four interrupt control modes.

The mask level of an interrupt is indicated by the interrupt mask bits in the CCR and EXR registers.

The interrupt handler processing for each interrupt mask level differs according to the interrupt control mode.

- Interrupt control mode 0

  Controlled by only the I bit in CCR. When the interrupt handler is initiated, the I bit in CCR is set. Do not clear the I bit in CCR by the interrupt handler. If the I bit is cleared, normal system operation is not guaranteed.
- Interrupt control mode 1

  Controlled by the I and UI bits in CCR. When the interrupt handler is initiated, the I and UI bits in CCR are set. For the interrupt handler with control level 0, clear the UI bit and change the interrupt mask level so that interrupts with control level 1 can be accepted. Do not clear the I bit of CCR for the interrupt handler with control level 1; otherwise, normal system operation is not guaranteed. The interrupt mask levels cannot be changed in the interrupt handler which has a control level of 1.
- Interrupt control mode 2

  Controlled by the I0 to I2 bits in EXR, and the I and UI bits in CCR are ignored. When the interrupt handler is initiated, the I0 to I2 bits in EXR are set to match the level of the interrupt generated.
- Interrupt control mode 3

  Controlled by the I and UI bits in CCR and the I0 to I2 bits in EXR. When the interrupt handler is initiated, the I and UI bits in CCR and the I0 to I2 bits in EXR are set. For the interrupt handler with control level 0, clear the I and UI bits of CCR and change the interrupt mask level so that interrupts with a higher priority can be accepted. Do not change EXR in the interrupt handler; otherwise, normal system operation is not guaranteed.

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. Accepting an interrupt generated during execution of the critical section in the kernel is normally delayed until execution of the critical section finishes. However, interrupts with interrupt levels higher than the kernel interrupt mask level are accepted immediately even during execution of the critical section.

Notes: 1. Service calls cannot be called 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.

2. If the interrupt control mode is 3 and the kernel interrupt mask level is 7, service calls cannot be issued from an interrupt handler whose control level is 1.

Notes on Interrupts:

- The user can freely create interrupt handlers. However, an interrupt handler with a long execution time reduces the throughput of the system. Interrupt handlers need to be created carefully since the system response is largely affected.
- The kernel interrupt mask level can be determined by a configurator definition. Service calls cannot be called by interrupt handlers with interrupt levels higher than this kernel interrupt mask level. Service calls cannot also be called by the NMI (Non Maskable Interrupt) interrupt handler. Normal system operation cannot be guaranteed when service calls are called by interrupt handlers with interrupt levels higher than the kernel interrupt mask level.
- Call the ret_int routine to return from an interrupt handler with an interrupt level equal to or lower than the kernel interrupt mask level. Normal system operation cannot be guaranteed when a method other than the ret_int routine is used.

6.5.2 Interrupt Handler Definition

An interrupt handler is defined by setting the start address of the interrupt handler in the respective vector table. Control passes directly to the interrupt handler without kernel intervention when an interrupt occurs.

For details on interrupt factors, refer to the corresponding hardware manual.

For details on definition of an interrupt handler, refer to Section 7, Configuration.

6.5.3 Undefined Interrupt Handler

An undefined interrupt handler is a program executed when an unexpected interrupt occurs in the system.

The sample program of an undefined interrupt handler calls the undefined interrupt processing (_KERNEL_H_ilint) of the kernel and passes control to the system down routine.

For the undefined interrupt information at system down, refer to Section 10, Information during System Down, and the corresponding hardware manual.

An undefined interrupt handler can be created similarly to an interrupt handler.

For details on definition of an undefined interrupt handler, refer to Section 7, Configuration.

6.6 CPU Exception Handler (Including TRAPA Instruction Exception)

6.6.1 Interrupt Handler Creation

Writing a CPU Exception Handler: A CPU exception handler (including the TRAPA instruction exception) is written with #pragma interrupt, similarly to an interrupt handler. However, since a CPU exception handler has the possibility of re-entry, it cannot have a dedicated stack. Therefore, do not specify stack switching (sp=) in #pragma interrupt. The CPU exception handler operates in the stack of the exception source.

Conditions for CPU Exception Handler Processing: Table 6.4 lists the conditions for CPU exception handler processing.

Table 6.4 CPU Exception Handler Processing Conditions

Item	Description
Interrupt mask	Initiated with the interrupts masked, and the I bit in CCR set to 1.
Usable registers	ER0 to ER6, MACH, and MACL (MACH and MACL are used when the function is selected) can be used. The register contents must be returned to the values at initiation before the CPU exception handler processing terminates.
Stack pointer	The stack of the exception source program is indicated at initiation. Since the CPU exception handler has the possibility of re-entry, the stack of the program where the exception occurred is used. The CPU exception handler cannot have a dedicated stack.
Usable service calls	Service calls that can be made from the CPU exception handler are as follows: sns_ctx, sns_loc, sns_dsp, sns_dpn, (i)get_tid, (i)vsta_knl, and (i)vsys_dwn However, no service calls can be made from a CPU-exception handler with a mask level higher than the kernel interrupt mask level.
Usable stack area	The stack of the program where the exception occurred is used. It is necessary to allocate a stack area including the space for the exception occurrence.
Termination method	Terminated by the ret_int routine. Use the RTE instruction to return from CPU-exception handlers with mask levels higher than the kernel interrupt mask level.

6.6.2 CPU Exception Handler Definition

A CPU exception handler is defined by setting the start address of the CPU exception handler in the respective vector table, similarly to an interrupt handler. Control passes directly to the CPU exception handler without kernel intervention when an exception occurs.

For details on exception sources, refer to the corresponding hardware manual.

For details on definition of a CPU exception handler, refer to Section 7, Configuration.

6.7 Cyclic Handler and Initialization Routine

6.7.1 Creation of Cyclic Handler and Initialization Routine

Writing a Cyclic Handler and Initialization Routine: A cyclic handler and initialization routine can be described as regular C language functions. Figure 6.3 shows an example of a cyclic handler written in C language. Figure 6.4 shows an example of an initialization routine written in C language. These handlers are executed in the non-task context state.

```
#include “kernel.h”

void HDR(VP_INT exinf)           ← (a)
{
    /* Cyclic handler processing */
}
```

Figure 6.3 Example of a C-Language Cyclic Handler

(a) Passes exinf specified at initial definition as a parameter.

```
#include “kernel.h”

void INI(void)
{
    /* Initialization routine processing */
}
```

Figure 6.4 Example of a C-Language Initialization Routine

Note: The timer initialization routine is automatically installed by selecting the time management function by the configurator. The timer initialization routine can be created by the user, but the symbol name for it must be _KERNEL_HIPRG_TIMINI.
The debug daemon initialization routine is automatically installed by selecting the object manipulation function by the configurator.

Conditions for Processing a Cyclic Handler and Initialization Routine: Table 6.5 lists the conditions for processing a cyclic handler and initialization routine.

Table 6.5 Conditions for Processing Cyclic Handler and Initialization Routine

Item	Description
Interrupt mask	[Cyclic handler] Initiated with the timer interrupt mask level. [Initialization routine] Initiated with the kernel interrupt mask level.
Usable registers	ER0 to ER6, MACH, and MACL (MACH and MACL are used when the function is selected) can be used. [Cyclic handler] Cyclic handler extended information is stored in ER0 (assembly language)/first parameter (C language).
Stack pointer	[Cyclic handler] The stack of the timer interrupt handler is used. The value of the stack pointer must be returned to the value at initiation before the cyclic handler processing terminates. [Initialization routine] The kernel stack is used.
Usable service calls	Service calls that can be called from non-task context.
Usable stack area	[Cyclic handler] At system configuration, allocate a stack area in which the memory size used by the cyclic handler is added to the timer interrupt handler stack. [Initialization routine] The kernel stack area is used. For the stack size used by the timer initialization routine, refer to description of the timer initialization routine in Section 9.10, Initialization Routine Stack.
Termination method	Terminated by the RTS instruction. Return the stack pointer to the state at initiation when the cyclic handler or initialization routine terminates.

6.7.2 Definition of Cyclic Handler and Initialization Routine

A cyclic handler and initialization routine are initially defined at system configuration. For details on definition of a cyclic handler or initialization routine, refer to Section 7, Configuration.

6.8 Timer Driver

A timer driver must be created to use the time management function of the kernel.

A timer driver consists of a timer initialization routine and a timer interrupt routine. The timer interrupt routine clears the interrupt factor and calls the timer interrupt handler (_KERNEL_H_timsys) of the kernel. The timer initialization routine is executed as an initialization routine.

6.8.1 Installing the Time Management Function

To use the time management function of the kernel, the following operations are required:

Defining Following Kernel Configuration Constants by Configurator:

- TIC_NUME: Numerator of time tick cycle
- TIC_DENO: Denominator of time tick cycle
- VTIM_LVL: Timer interrupt level

The cycle time when the time tick is provided is TIC_NUME/TIC_DENO [ms]. Based on this cycle time, the precision of the time parameter specified in the service call is determined. For example, when tslp_tsk(10) is executed, the timeout time is 12 to 15 ms if TIC_NUME = 3 and TIC_DENO = 1; the timeout time is 10 to 10.5 ms if TIC_NUME = 1 and TIC_DENO = 2. Note that at least one of TIC_NUME and TIC_DENO must be specified as 1.

In addition, if TIC_DENO is specified as a value greater than 1, the maximum value that can be specified to TMO-type and RELTIM-type parameters is limited to the available maximum value/TIC_DENO.

When the time management function is selected by the configurator, the timer initialization routine (_KERNEL_HIPRG_TIMINI) is automatically defined as an initialization routine.

Writing a Timer Driver: Here, the following two routines are created.

- Timer initialization routine: _KERNEL_HIPRG_TIMINI (fixed symbol name)
- Timer interrupt routine: _KERNEL_H_TIM (fixed symbol name)

The timer initialization routine is created as an initialization routine. For details, refer to Section 6.7, Cyclic Handler and Initialization Routine.

In the timer initialization routine, initialize the timer counter registers according to TIC_NUME (numerator of time tick cycle) and TIC_DENO (denominator of time tick cycle) defined in kernel_macro.h and kernel_macro.inc. In addition, specify the timer interrupt level for the interrupt controller according to VTIM_LVL defined in kernel_macro.h and kernel_macro.inc.

When a timer interrupt occurs, the timer interrupt routine _KERNEL_H_TIM is initiated as an interrupt handler. The timer interrupt routine clears the interrupt factor flag.

A timer interrupt routine is written, as shown in Figure 6.5. The timer interrupt processing routine operates as an interrupt handler. Table 6.3 lists the conditions for timer interrupt routine processing.

```
#include “kernel.h”
//#include “1650tmrdrv.h”                              ← (a)
extern KERNEL_HI_TIM_SP;
void KERNEL_H_timsys(void);
const VP P_intstkxx = &KERNEL_HI_TIM_SP;               ← (b)
#pragma interrupt(KERNEL_H_TIM(sp=P_intstkxx,          ← (c)
                       sy=$KERNEL_H_timsys))
void KERNEL_H_TIM(void)
{
    //unsigned char tsr = TPU0.TSR.BIT.TGFA;
    // dummy read
       TPU0.TSR.BIT.TGFA; // dummy read
       TPU0.TSR.BIT.TGFA = 0;
    // status register(TGFA flag) clear
    // ivbgn_int(88);  // call ivbgn_int(88)           ← (d)
    // Timer interrupt routine processing //
    // ivend_int(88);  // call ivend_int(88)
}
```

Figure 6.5 Example of a Timer Interrupt Routine

(a) Includes the timer header file.

(b) Defines the initial stack pointer value as const type.
Note: The symbol name for the timer interrupt stack is fixed in the system.

(c) Declares the timer interrupt routine as an interrupt function by using #pragma interrupt. The following must be defined:
- Stack switching (sp=P_intstkxx)
- Timer interrupt routine end (sy=$KERNEL_H_timsys)
  This function name is fixed.

(d) Specifies the start and end of interrupt processing (ivbgn_int and ivend_int). They are necessary when using the RTOS debugging function of the E6000H.

Linking the Timer Driver: The timer driver must be linked to the kernel.

6.8.2 Sample Timer Driver

The sample timer driver consists of the following:

- Source program file (file name: ***nnnnz***_tmrdrv.c)
  - Timer initialization routine (function: _KERNEL_HIPRG_TIMINI)
  - Timer interrupt handler (function: _KERNEL_H_TIM)

Table 6.6 shows the sample timer driver file provided and the timer module clock source assumed by the sample timer driver. For more details, refer to the contents of the header file.

Table 6.6 Sample Timer Driver Files and Clock Sources

Product	Target Microcomputer and Internal Timer Module		Assumed Timer Module Clock Source
HI1000/4	H8SX/1650	TPU	Peripheral clock (Pϕ) = 35 MHz
	H8SX/1525	TPU	Peripheral clock (Pϕ) = 35 MHz
	AE57	TMR1	Peripheral clock (Pϕ) = 20 MHz
	H8S/2655	TPU	Peripheral clock (Pϕ) = 20 MHz
	H8S/2148	FRT	Peripheral clock (Pϕ) = 20 MHz

The operating conditions of the sample timer driver are defined in the header file. The header file can be modified as required. Note however that to change the timer interrupt cycle time or timer interrupt level, the setting of the time tick cycle (CFG_TICNUME and CFG_TICDENO) or timer interrupt level (CFG_TIMINTLVL) must be changed by the configurator instead of changing the header file.

6.9 CPU Initialization Routine

6.9.1 Creation of CPU Initialization Routine

The CPU initialization routine is a program that is executed first after a CPU reset. For details, refer to Section 4.12.1, CPU Reset and Kernel Initiation. Also refer to the contents of the sample file ***nnnn***cpuini.c (C language). The symbol name (_KERNEL_H_CPUINI) for the CPU initialization routine is fixed in the system.

Perform necessary processing such as VBR and SBR register initialization in the CPU initialization routine.

6.9.2 Definition of CPU Initialization Routine

Define the start address of the CPU initialization routine to the reset vector (vector number 0). In addition, the kernel is normally initiated at the end of the CPU initialization routine.

- Vector number 0: Power-on reset

Create the reset vector in one of the following ways.

1. When the user creates a reset vector

   Create a reset vector as shown in Figure 6.6.

```
    .SECTION C_hivct, DATA, LOCATE=0     ← (a)
    .IMPORT   _KERNEL_H_CPUINI
:
    .DATA.L  _KERNEL_H_CPUINI            ← (b)
    .END
```

Figure 6.6 Example of Creating a Reset Vector

(a) Specifies C_hivct for the reset vector section name and allocates the section to address H'0.

(b) Specifies the program address for vector number 0 (power-on reset) as _KERNEL_H_CPUINI.

2. When defining the CPU initialization routine by the configurator

   The configurator automatically defines the CPU initialization routine address (_KERNEL_H_CPUINI) for interrupt number 0. Allocate the vector table section created by the configurator, C_hivct (when the vector table format without division is specified) or C_hiresvct (when the vector table format with division is specified), to address H'0 at linkage.

6.10 System Down Routine

The system down routine is created as the following C-language function. Note that this routine name is fixed.

```
    void    vsys_dwn (H type,H inf1,B inf2,B inf3,H inf4,UW inf5)
```

For the meaning of the parameters passed to the system down routine, refer to Section 10, Information during System Down.

The system down routine must be created and linked to the kernel.

Although the system down routine operates under abnormal conditions, it cannot use kernel functions such as service calls if the kernel fails (error type is negative). Do not return from a system down routine.

When debugging an application program, maintain the system down routine state and make the program enter an endless loop to analyze why a system down occurred and take measures to prevent system downs from occurring.

Section 7 Configuration

7.1 Read First

Before creating a system, please read and fully understand this section.

The following tools are used for actual configuration:

- Configurator supplied with this product
- HEW

The description in this section is provided on the assumption that the user has already mastered HEW. For information about HEW, refer to the HEW manual or online help. For information about the configurator operation, refer to Section 7.4, Configurator, or online help.

7.1.1 Linkage Method

To create load modules for the system, the following operations are necessary:

- Create application files
- Create configuration files using the configurator
- Use the build function of HEW and create load modules

Linkage: The HI1000/4 links the kernel, all configuration files, and application file into a single load module.

Figure 7.1 shows the flow for creating a load module.

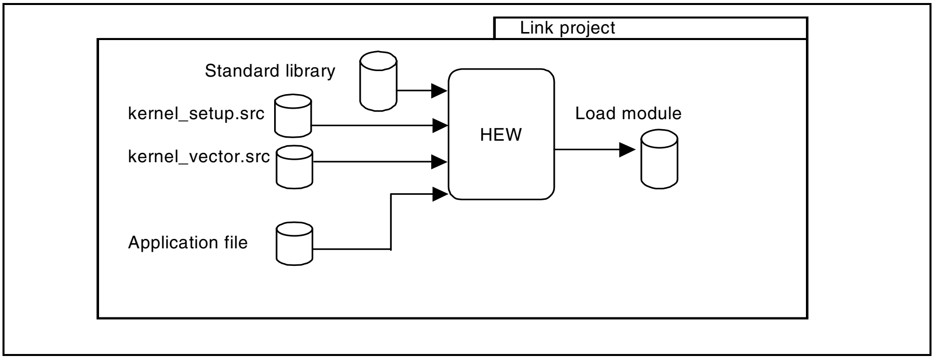


Figure 7.1 Load Module Creation

7.1.2 Configurator Output Files

Table 7.1 lists the files output from the configurator.

Table 7.1 Files Output from Configurator

No.	File Name	Contents
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.inc	File defining system initialization routine
5	kernel_vector.src	File defining vector table creation information

Setup File: Creates a kernel with only the required functions by linking it to the kernel library. Kernel tables such as the TCB (task control block) are also created. The kernel operates referring to or updating this information.

Header Files with Automatic ID Assignment Result (for C Language and Assembly Language): The configurator supports the automatic ID assignment function, and its result is output into kernel_id.h. For example, if "ID_MainTask" is specified as the task ID when a task is created by the configurator, kernel_id.h is output by the configurator as follows:

```
  #define ID_MainTask 1
```

Header File Defining Kernel Configuration Constants (for C Language and Assembly Language): kernel_macro.h and kernel_macro.inc are included from hihead\kernel.h and kernel.inc, respectively (see Section 6.1.1, Standard Header File).

File Defining System Initialization Routine: The configurator creates an initialization routine source file kernel_sysini.src that calls the defined initialization routine.

File Defining Vector Table Creation Information: The configurator creates a vector table source file kernel_vector.src in which the interrupts and CPU exception handlers are defined.

7.2 Directory Configuration

Table 7.2 lists the directory configuration at shipment. Refer to the release notes attached to the product for details on the supplied files. The following describes an example of H8SX/1650 advanced mode.

Table 7.2 Directory Configuration at Shipment

Directory and File Names*1	Description
kernel\hihead	Standard header file directory
kernel\hilib\	Kernel library storage directory*3
kernel\samples\h8sx\nnnnzsmp***nnnnz***smp.hws	Sample workspace*2
kernel\samples\h8sx\nnnnzsmp***nnnnz***smp***nnnnz***smp.hwp	Sample project*2
kernel\samples\h8sx\nnnnzsmp\src	Directory of sample files and system configuration files
kernel\samples\h8sx\nnnnzsmp\src\***nnnnz***.hcf	Sample configurator setting file (HCF file)
kernel\samples\h8sx\nnnnzsmp***nnnnz***smp\obj	Object file and load module output directory (for advanced/middle mode)

Notes: 1. ***nnnn*** is the CPU name used. ***z*** is the CPU operating mode (a: advanced mode, n: normal mode). For the H8SX/1650 advanced mode, ***nnnnz*** will be 1650a.

2. The user usually does not need to control sample project files because sample project files are defined in the sample workspace in advance.

3. For details on the subdirectory configuration of the kernel library, see Table 8.1.

7.3 Operating Procedure

The following describes the normal operating procedure:

1. Double-click kernel\samples\h8sx\nnnnzsmp\src\***nnnnz***.hcf to initiate the configurator.
2. Provide necessary settings for the configurator.
3. Save kernel\samples\h8sx\nnnnzsmp\src\***nnnnz***.hcf and create a configuration file, and name the folder as kernel\samples\h8sx\nnnnzsmp\src.
4. Terminate the configurator.
5. Double-click ***nnnnz***smp.hws to initiate HEW. Then, specify the project to be used, such as ***nnnnz***smp. Unused projects can be deleted.
6. Provide necessary operations, such as adding application files to HEW or setting the C compiler or linkage editor options, and execute the build. As a result, load module files are created in the respective directories, such as the kernel\samples\h8sx\nnnnzsmp***nnnnz***smp\obj folder.

Note: ***nnnn*** is the CPU name used. ***z*** is the CPU operating mode (a: advanced mode, n: normal mode).

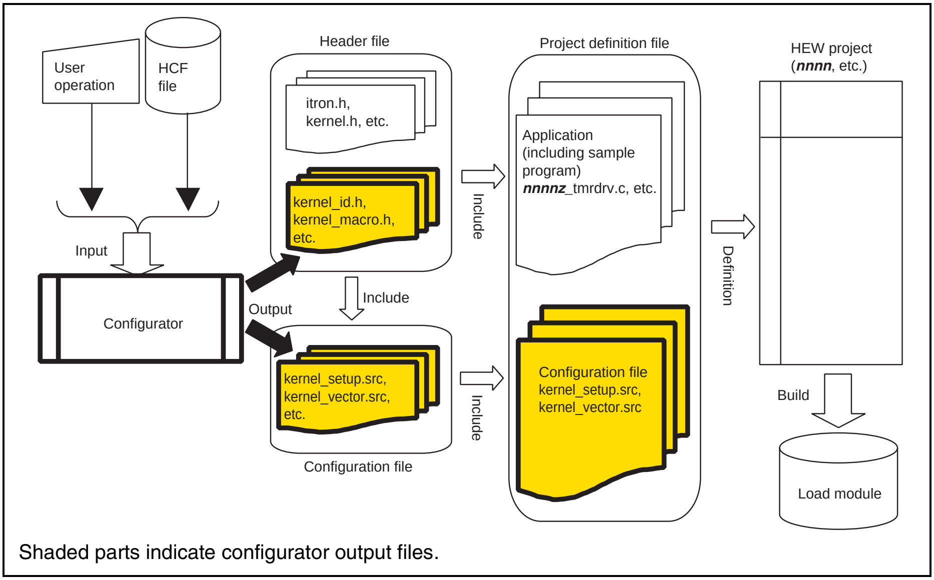
7.4 Configurator

This section describes basic configurator operations and settings. For details on configurator operations, refer to the configurator online help.

7.4.1 Overview

The configurator is a tool that is used to set the kernel operating parameters. The configurator creates header and assembler source files (see Table 7.1) according to the settings. The created files and applications are built (compiled and linked) into a system (load module).

Figure 7.2 shows the position of the configurator in the system configuration.


User operation
HCF file
Input
Configurator
Output
Header file
itron.h, kernel.h, etc.
kernel_id.h, kernel_macro.h, etc.
Include
kernel_setup.src, kernel_vector.src, etc.
Configuration file
Include
Include
Project definition file
Application (including sample program) *nnnnz*_tmrdrv.c, etc.
Configuration file kernel_setup.src, kernel_vector.src
Definition
HEW project (*nnnn*, etc.)
Build
Load module

Shaded parts indicate configurator output files.

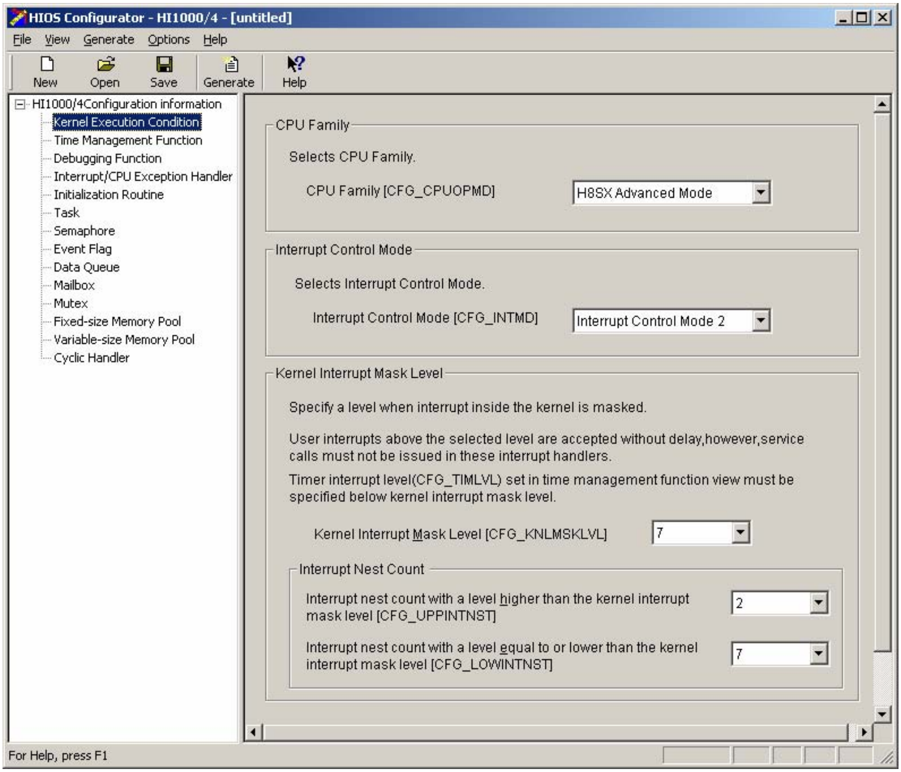
Figure 7.2 Position of Configurator in System Configuration

7.4.2 Configurator Construction

Select [All programs] from the [Start] menu in Windows and then select [HI1000-4 Configurator] in the [HI1000-4] program folder to initiate the configurator.

Figure 7.3 shows the configurator outline.

The configurator consists of a configuration information input part, a list window (on the left side), and a configuration information input window (on the right side). Input data in the configuration information input window and execute the Configuration File Creation command with the menu or the tool button. The configuration file is then created.


HIOS Configurator - HI1000/4 - [untitled]
File View Generate Options Help
New Open Save Generate Help
HI1000/4Configuration information
Kernel Execution Condition
Time Management Function
Debugging Function
Interrupt/CPU Exception Handler
Initialization Routine
Task
Semaphore
Event Flag
Data Queue
Mailbox
Mutex
Fixed-size Memory Pool
Variable-size Memory Pool
Cyclic Handler
For Help, press F1

CPU Family

Selects CPU Family.

CPU Family [CFG_CPUOPMD]: H8SX Advanced Mode

Interrupt Control Mode

Selects Interrupt Control Mode.

Interrupt Control Mode [CFG_INTMD]: Interrupt Control Mode 2

Kernel Interrupt Mask Level

Specify a level when interrupt inside the kernel is masked.

User interrupts above the selected level are accepted without delay,however,service calls must not be issued in these interrupt handlers.

Timer interrupt level(CFG_TIMLVL) set in time management function view must be specified below kernel interrupt mask level.

Kernel Interrupt Mask Level [CFG_KNLMSKLVL]: 7

Interrupt Nest Count

Interrupt nest count with a level higher than the kernel interrupt mask level [CFG_UPPINTNST]: 2

Interrupt nest count with a level equal to or lower than the kernel interrupt mask level [CFG_LOWINTNST]: 7

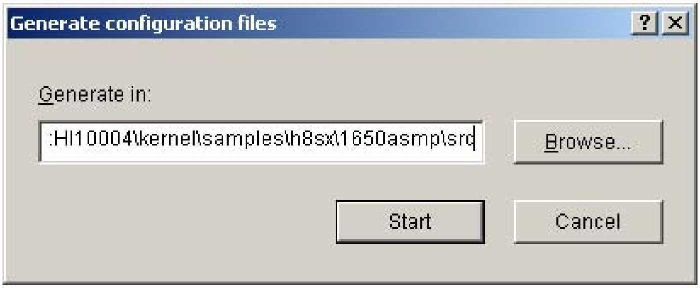
Figure 7.3 Configurator Outline

7.4.3 File Operation

Configurator Settings File (HCF File): Information set in the configurator can be saved in the HCF file.

Configuration File Creation: When [Configuration File Creation] is selected from the [Create] menu or the tool button [Create] is pressed, the dialog box shown in Figure 7.4 opens. Specify a folder where a configuration file is to be created. Note that the configuration file needs to be created in the kernel\samples\***mmmm******nnnnz***smp\src folder depending on the device used and its CPU family.

Note: ***mmmm*** and ***nnnn*** are names of the CPU family and the device, respectively.



Generate in: :HI10004\kernel\samples\h8sx\1650asmp\src

Figure 7.4 Folder Selection Dialog

The files listed in Table 7.1 are created in the specified folder.

Note: If a file with the same name already exists in the folder where the configuration file is to be created, the existing file is automatically overwritten.

7.4.4 Configurator Settings

Most configurator settings influence kernel operation. Table 7.3 lists the settings.

If a name is specified as an ID number by automatic ID assignment, the configurator automatically assigns the ID number and outputs the result to kernel_id.h. When specifying names, be careful not to specify already-used names or externally defined names such as function names.

Table 7.3 Configurator Setting Items

1. Kernel Operating Condition View		
1.1	CPU family	CFG_CPUOPMD
	Specify one of the following CPU families: • H8SX advanced mode • AE-5 advanced mode • H8S/2600 advanced mode • H8S/2600 normal mode • H8S/2000 advanced mode • H8S/2000 normal mode For details, refer to the hardware manual of the CPU used.	
1.2	Interrupt control mode	CFG_INTMD
	Specify one of the interrupt control modes listed below. Selectable interrupt control modes depend on the selected CPU family. • When the selected CPU is H8SX or AE-5: Interrupt control modes 0 and 2 • When the selected CPU is H8S/2600 or H8S/2000: Interrupt control modes 0 to 3 For details, refer to the hardware manual of the CPU used.	
1.3	Kernel interrupt mask level	CFG_KNLMSKLVL
	Specify the mask level for masking interrupts inside the kernel. The value that can be specified differs according to the interrupt control mode (CFG_INTMD). • Interrupt control mode 0: 1 • Interrupt control mode 1: 1 to 3 • Interrupt control mode 2: 1 to 7 • Interrupt control mode 3: 1 to 8 Interrupt handlers with a level higher than the kernel interrupt mask level must not issue a service call. The kernel interrupt mask level is output to kernel_macro.h and kernel_macro.inc.	
1.4	Interrupt nest count with a level higher than the kernel interrupt mask level	CFG_UPPINTNST
	Specify the maximum nest count for interrupts with a level higher than the kernel interrupt mask level (CFG_KNLMSKLVL).	
1.5	Interrupt nest count with a level equal to or lower than the kernel interrupt mask level	CFG_LOWINTNST
	Specify the maximum nest count for interrupts with a level equal to or lower than the kernel interrupt mask level (CFG_KNLMSKLVL).	

Table 7.3 Configurator Setting Items (cont)

2. Time Management Function View		
2.1	Use of time management function	CFG_TIMUSE
	When using service calls with time parameters, such as tslp_tsk or the cyclic handler, check this item. In this case, a timer driver must be created. For details, refer to Section 6.8, Timer Driver.	
2.2	Timer interrupt number	CFG_TIMINTNO
	Specify the interrupt vector number of the timer interrupt handler.	
2.3	Timer interrupt level	CFG_TIMINTLVL
	Specify the interrupt level of the timer interrupt handler. The timer interrupt level is output to kernel_macro.h and kernel_macro.inc. In the timer initialization handler, this value should be specified for the interrupt controller of the device.	
2.4	Timer interrupt handler stack size	CFG_TIMRSTKSZ
	Specify the stack size of the timer interrupt handler as the size calculated according to section 9.8, Timer Interrupt Stack.	
2.5	Numerator (TIC_NUME) and denominator (TIC_DENO) of time tick cycle	CFG_TICNUME, CFG_TICDENO
	The time tick cycle is TIC_NUME/TIC_DENO [ms]. Note that at least one of TIC_NUME and TIC_DENO must be 1. If TIC_DENO is specified as a value greater than 1, the maximum value that can be specified to TMO-type, RELTIM-type, and OVRTIM-type parameters is limited to the available maximum value/TIC_DENO. An integer in the range of 1 to 65535 can be specified for TIC_NUME. An integer in the range of 1 to 100 can be specified for TIC_DENO. TIC_NUME and TIC_DENO are output to kernel_macro.h and kernel_macro.inc. The time tick cycle of the timer driver must be set according to the information of kernel_macro.h and kernel_macro.inc.	
2.6	Synchronization and communication function with timeout	CFG_TOUTUSE
	Check CFG_TOUTUSE when using the following synchronization and communication functions with timeout: • Semaphore: twai_sem • Mailbox: trcv_mbx • Event flag: twai_flg • Mutex: tloc_mtx • Data queue: tsnd_dtq, trcv_dtq • Fixed-size memory pool: tget_mpf • Variable-size memory pool: tget_mpl	

Table 7.3 Configurator Setting Items (cont)

3. Debugging Function View		
3.1	Object manipulation functions	CFG_ACTION
	Check this item when using object manipulation functions with the debugging extension. When this item is checked, the debug daemon initialization routine is automatically installed as an initialization routine.	
3.2	Service call trace function	CFG_TRACE
	Check this item when using the service call trace function.	
3.3	Type of service call trace function	CFG_TRCTYPE
	Select the trace function type. For details, refer to Section 4.11.2, Service Call Trace Function. • Target trace • Tool trace • Target trace (simple version) • Tool trace (simple version)	
3.4	Number of acquired trace information items	CFG_TRCCNT
	Specify the number of information items that are to be acquired by the trace function.	
3.5	Number of acquired object states	CFG_TRCOBJCNT
	Specify the maximum number of object states that is to be acquired at trace acquisition as the object history. A value from 0 to 32 can be specified as the number of acquired object states. If target trace (simple version) or tool trace (simple version) is selected, the number of acquired object states is set to 0.	
3.6	Buffer size	CFG_TRCBUFSZ
	Specify the trace buffer size that will be used if the normal-version or simple-version target trace is selected.	
4. Interrupt Handler and CPU Exception Handler View		
4.1	Maximum interrupt vector number	CFG_MAXVCTNO
	Specify the maximum interrupt handler vector number. For details on the vector number, refer to the hardware manual of the CPU used.	
4.2	Vector table format	CFG_VCTFMT
	Specify the vector table format. • With division: Allocates the vector table from address H'0. • Without division: Allocates separate vector tables in address H'0 and the address specified by VBR.	

Table 7.3 Configurator Setting Items (cont)

4.3	Interrupt handler information setting	—
	Select the vector to be defined and specify items in 4.3.1 and 4.3.2 below as interrupt handler information.	
4.3.1	Address	—
	Specify the start address of the interrupt handler.	
4.3.2	Description language	—
	Select the high-level language (TA_HLNG) or assembly language (TA_ASM).	
4.4	Interrupt handler stack information setting	—
	Specify items in 4.4.1 and 4.4.2 below as interrupt handler stack information.	
4.4.1	Interrupt handler stack name (stack pointer)	—
	Specify the stack name (stack pointer) used by the interrupt handler.	
4.4.2	Interrupt handler stack size	—
	Specify the stack area used by the interrupt handler. For details on stack size calculation, refer to section 9.7, Interrupt Handler Stacks.	
5. Initialization Routine View		
5.1	Initialization routine address	—
	Specify the start address of the initialization routine.	
5.2	Initialization routine stack size	—
	Specify the stack size used by the initialization routine. As the initialization routine uses the kernel stack, the value specified here affects the kernel stack size. For details, refer to section 9.9, Kernel Stack.	
5.3	Description language	—
	Select the high-level language (TA_HLNG) or assembly language (TA_ASM).	
6. Task View		
6.1	Maximum task ID	CFG_MAXTSKID
	Specify the maximum task ID as a value from 1 to 255. The range of usable task ID numbers is 1 to CFG_MAXTSKID.	
6.2	Maximum task priority	CFG_MAXTSKPRI
	Specify the maximum task priority as a value from 1 to 31. The range of usable task priorities is 1 to CFG_MAXTSKPRI.	
6.3	Number of stacks	CFG_NUMTSKSTK
	Specify the number of task stacks. Task stacks are created for the number of specified tasks. Specify the created stack name (stack pointer) when defining the task.	
6.3.1	Stack area setting	—
	Define the stack area (stack pointer) and size used by the task.	

Table 7.3 Configurator Setting Items (cont)

6.4	Task information setting	—
	Specify items in 6.4.1 to 6.4.7 below as information of the respective task.	
6.4.1	ID number/ID name	—
	Specify the ID number or ID name. The range of usable task ID numbers is 1 to CFG_MAXTSKID.	
6.4.2	Task address	—
	Specify the start address of the task of the respective task ID.	
6.4.3	Priority at task initiation	—
	Define the priority at task initiation for the respective task ID. A value from 1 to CFG_MAXTSKPRI can be defined.	
6.4.4	Attribute	—
	If [Start Task after Creating It (TA_ACT)] is selected, the task enters the READY state on being initiated. Tasks are initiated in the ID order, regardless of the order in which they were defined. When this item is checked, the task extended information is passed when a task is initiated.	
6.4.5	Description language	—
	Select the high-level language (TA_HLNG) or assembly language (TA_ASM).	
6.4.6	Task stack	—
	Select the stack area used by the task of the respective task ID. When the task of the respective task ID shares a stack with another task, specify the same stack when defining the stack area.	
6.4.7	Extended information	—
	Specify the extended information for the task of the respective task ID. Extended information can be specified independently for each task ID. Extended information includes the start address of a memory area allocated as a packet for the purpose of storing information regarding the target object.	
7. Semaphore View		
7.1	Maximum semaphore ID	CFG_MAXSEMID
	Specify the maximum semaphore ID as a value from 0 to 255. The range of usable semaphore ID numbers is 1 to CFG_MAXSEMID. Specify 0 when no semaphore is used.	
7.2	Semaphore information setting	—
	Specify items in 7.2.1 to 7.2.3 below as information of the respective semaphore.	
7.2.1	ID number/ID name	—
	Specify the ID number or ID name. The range of usable semaphore ID numbers is 1 to CFG_MAXSEMID.	
7.2.2	Maximum number of semaphore resources	—
	Specify the maximum number of resources of the respective semaphore ID. A value from 1 to 65535 can be specified.	
7.2.3	Initial value of number of semaphore resources	—
	Specify the initial value of the number of resources of the respective semaphore ID. A value from 0 to the maximum number of semaphore resources can be specified.	

Table 7.3 Configurator Setting Items (cont)

8. Event Flag View		
8.1	Maximum event flag ID	CFG_MAXFLGID
	Specify the maximum event flag ID as a value from 0 to 255. The range of usable event flag ID numbers is 1 to CFG_MAXFLGID. Specify 0 when no event flag is used.	
8.2	Event flag information setting	—
	Specify items in 8.2.1 to 8.2.4 below as information of the respective event flag.	
8.2.1	ID number/ID name	—
	Specify the ID number or ID name. The range of usable event flag ID numbers is 1 to CFG_MAXFLGID.	
8.2.2	Attribute	—
	Specify the attribute of the respective event flag ID. • TA_WMUL: Waiting for more than one task is allowed • TA_CLR: Bit pattern is cleared when a wait task is released	
8.2.3	Task wait queue	—
	The task wait queue of the respective event flag ID is specified to be managed as follows: TA_TFIFO: Task wait queue is managed on a FIFO basis	
8.2.4	Initial bit pattern	—
	Specify the initial value of the bit pattern of the respective event flag ID. A 16-bit pattern can be specified.	
9. Data Queue View		
9.1	Maximum data queue ID	CFG_MAXDTQID
	Specify the maximum data queue ID as a value from 0 to 255. The range of usable data queue ID numbers is 1 to CFG_MAXDTQID. Specify 0 when no data queue is used.	
9.2	Data queue information setting	—
	Specify items in 9.2.1 to 9.2.3 below as information of the respective data queue.	
9.2.1	ID number/ID name	—
	Specify the ID number or ID name. The range of usable data queue ID numbers is 1 to CFG_MAXDTQID.	
9.2.2	Number of data items	—
	Specify the number of data items in the respective data queue ID from a range of 0 to 65535.	
9.2.3	Task wait queue	—
	The task wait queue of the respective data queue ID is specified to be managed as follows: TA_TFIFO: Task wait queue is managed on a FIFO basis	

Table 7.3 Configurator Setting Items (cont)

10. Mailbox View		
10.1	Maximum mailbox ID	CFG_MAXMBXID
	Specify the maximum mailbox ID as a value from 0 to 255. The range of usable data queue ID numbers is 1 to CFG_MAXMBXID. Specify 0 when no mailbox is used.	
10.2	Maximum message priority	CFG_MAXMSGPRI
	Specify the maximum message priority as a value from 1 to 255. The range of usable message priorities is 1 to CFG_MAXMSGPRI.	
10.3	Mailbox information setting	—
	Specify items in 10.3.1 to 10.3.4 below as information of the respective mailbox.	
10.3.1	ID number/ID name	—
	Specify the ID number or ID name. The range of usable mailbox ID numbers is 1 to CFG_MAXMBXID.	
10.3.2	Maximum message priority	—
	Specify the maximum message priority as a value from 1 to 255. The range of usable message priorities is 1 to CFG_MAXMSGPRI.	
10.3.3	Task wait queue	—
	The task wait queue of the respective mailbox ID is specified to be managed as follows: • TA_TFIFO: Task wait queue is managed on a FIFO basis • TA_TPRI: Task wait queue is managed on a priority basis	
10.3.4	Message queue	—
	The message queue of the respective mailbox ID is specified to be managed as follows: • TA_MFIFO: Message queue is managed on a FIFO basis • TA_MPRI: Message queue is managed on a priority basis	
11. Mutex View		
11.1	Maximum mutex ID	CFG_MAXMTXID
	Specify the number of mutexes used as a value from 0 to 255. The range of usable mutex ID numbers is 1 to CFG_MAXMTXID. Specify 0 when no mutex is used.	
11.2	Mutex information setting	—
	Specify items in 11.2.1 to 11.2.3 below as information of the respective mutex.	
11.2.1	ID number/ID name	—
	Specify the ID number or ID name. The range of usable mutex ID numbers is 1 to CFG_MAXMTXID.	
11.2.2	Attribute	—
	The attribute is specified as the ceiling priority protocol (TA_CEILING).	
11.2.3	Maximum ceiling priority	—
	Specify the maximum ceiling priority of the respective mutex ID. The maximum value that can be specified is CFG_MAXTSKPRI (maximum task priority).	

Table 7.3 Configurator Setting Items (cont)

12. Fixed-Size Memory Pool View		
12.1	Maximum fixed-size memory pool ID	CFG_MAXMPFID
	Specify the maximum fixed-size memory pool ID as a value from 0 to 255. The range of usable fixed-size memory pool ID numbers is 1 to CFG_MAXMPFID. Specify 0 when no fixed-size memory pool is used.	
12.2	Fixed-size memory pool information setting	—
	Specify items in 12.2.1 to 12.2.4 below as information of the respective fixed-size memory pool.	
12.2.1	ID number/ID name	—
	Specify the ID number or ID name. The range of usable fixed-size memory pool ID numbers is 1 to CFG_MAXMPFID.	
12.2.2	Number of acquirable memory blocks	—
	Specify the number of acquirable memory blocks in the respective fixed-size memory pool ID. A value from 1 to 65535 can be specified.	
12.2.3	Size	—
	Specify the size of a single fixed-size memory block of the respective fixed-size memory pool ID. A value from 2 to 65530 can be specified.	
12.2.4	Task wait queue	—
	The task wait queue of the respective fixed-size memory pool ID is specified to be managed as follows: TA_TFIFO: Task wait queue is managed on a FIFO basis	
13. Variable-Size Memory Pool View		
13.1	Maximum variable-size memory pool ID	CFG_MAXMPLID
	Specify the maximum variable-size memory pool ID as a value from 0 to 255. The range of usable variable-size memory pool ID numbers is 1 to CFG_MAXMPLID. Specify 0 when no variable-size memory pool is used.	
13.2	Variable-size memory pool information setting	—
	Specify items in 13.2.1 to 13.2.3 below as information of the respective variable-size memory pool.	
13.2.1	ID number/ID name	—
	Specify the ID number or ID name. The range of usable variable-size memory pool ID numbers is 1 to CFG_MAXMPLID.	
13.2.2	Variable-size memory pool area size	—
	Specify the size of the variable-size memory of the respective variable-size memory pool ID. A value form 18 to 65534 can be specified.	
13.2.3	Task wait queue	—
	The task wait queue of the respective variable-size memory pool ID is specified to be managed as follows: TA_TFIFO: Task wait queue is managed on a FIFO basis	

Table 7.3 Configurator Setting Items (cont)

14. Cyclic Handler View		
14.1	Maximum cyclic handler ID	CFG_MAXCYCID
	Specify the maximum cyclic handler ID as a value from 0 to 254. The range of usable cyclic handler ID numbers is 1 to CFG_MAXCYCID.	
14.2	Cyclic handler information setting	—
	Specify items in 14.2.1 to 14.2.7 below as information of the respective cyclic handler.	
14.2.1	ID number/ID name	—
	Specify the ID number or ID name. The range of usable cyclic handler ID numbers is 1 to CFG_MAXCYCID.	
14.2.2	Cyclic handler address	—
	Specify the start address of the cyclic handler of the respective cyclic handler ID.	
14.2.3	Extended information	—
	Specify the extended information for the cyclic handler of the respective cyclic handler ID. Extended information can be specified independently for each cyclic handler ID. Extended information includes the start address of a memory area allocated as a packet for the purpose of storing information regarding the target object.	
14.2.4	Initiation cycle of initiation information	—
	Specify the initiation cycle of the cyclic handler of the respective cyclic handler ID. A value from 0x1 to 0x7fffffff can be specified.	
14.2.5	Initiation phase of initiation information	—
	Specify the initiation phase of the cyclic handler of the respective cyclic handler ID. A value from 0x0 to initiation cycle of initiation information can be specified.	
14.2.6	Attribute	—
	Specify the cyclic handler attribute of the respective cyclic handler ID. • TA_STA: Sets the operating state after creating a cyclic handler • TA_PHS: Reserves the initiation phase	
14.2.7	Description language	—
	Select the high-level language (TA_HLNG) or assembly language (TA_ASM).	

Section 8 HEW Workspace and Projects

Use the following procedure to create load modules with the build function of HEW.

1. Add the files necessary for creating the load module in a project.
2. Specify the options for the C compiler, assembler, and optimizing linkage editor.
3. Execute a build.

This product supplies a workspace file, ***nnnnz***smp.hws, corresponding to various devices in the CPU family. Double-clicking ***nnnnz***smp.hws starts HEW and opens ***nnnnz***smp.hws.

Sample projects corresponding to various CPUs are defined beforehand in ***nnnnz***smp.hws.

The following sample projects are provided.

- 1650asmp: Project for H8SX/1650 advanced mode
- 1525asmp: Project for H8SX/1525 advanced mode
- 2655asmp: Project for H8S/2655 advanced mode
- 2655nsmp: Project for H8S/2655 normal mode
- 2148asmp: Project for H8S/2148 advanced mode
- 2148nsmp: Project for H8S/2148 normal mode
- ae57smp: Project for AE57

Select a project for the target device, and change the settings as explained below.

Double-clicking a sample project opens the HEW workspace window as shown in Figure 8.1.

Opening a project file and executing a build compiles, assembles, optimizes and links, and converts the files defined in the project to create a load module.

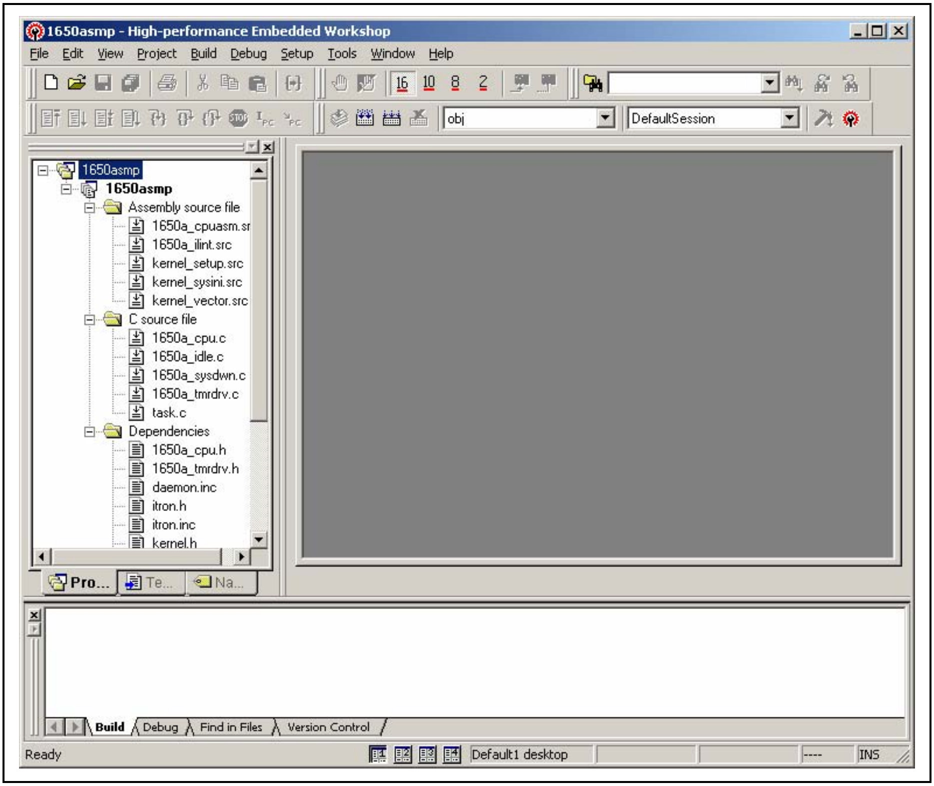


Figure 8.1 Project Selection

8.1 Supplied Kernel Libraries

The directory of the supplied kernel libraries is selected according to the CPU family used by the application and the enabled functions. Select a directory from among the directories of the supplied kernel libraries listed in Table 8.1 based on the conditions of (a) to (c) shown below.

(a) For the CPU family, select the CPU family used.

(b) When the selected CPU family is h8sx_adv, with or without MAC registers determines whether or not MAC registers are included in the task context.

(c) With or without E6000H debugging determines whether or not the RTOS debugging function (e.g. task time measurement) of the E6000H emulator is used.

Table 8.1 Directories of Supplied Kernel Libraries

		Function Enabled/Disabled	
CPU Family	**Directory Name***	**MAC Registers**	**E6000H RTOS Debugging**
h8sx_adv	mac_dbg	Enabled (mac)	Enabled (dbg)
	mac_nodbg		Disabled (nodbg)
	nomac_dbg	Disabled (nomac)	Enabled (dbg)
	nomac_nodbg		Disabled (nodbg)
ae57	dbg	Disabled	Enabled (dbg)
	nodbg		Disabled (nodbg)
h8s26_adv	dbg	Enabled	Enabled (dbg)
	nodbg		Disabled (nodbg)
h8s26_nor	dbg	Enabled	Enabled (dbg)
	nodbg		Disabled (nodbg)
h8s20_adv	dbg	Disabled	Enabled (dbg)
	nodbg		Disabled (nodbg)
h8s20_nor	dbg	Disabled	Enabled (dbg)
	nodbg		Disabled (nodbg)

Note: A directory name is indicated in the format of ***yyy_zzz***. ***yyy*** indicates whether the MAC registers are available (mac: enabled, or nomac: disabled). ***zzz*** indicates usage of the E6000H RTOS debugging function (dbg: enabled, or nodbg: disabled).

Next, the kernel library file is selected according to whether the parameter check function and shared stack function of the service calls are enabled.

Select the kernel library file to be used from among the kernel library files listed in Table 8.2 based on the conditions of (a) and (b) shown below.

(a) The parameter check is selected when using the function to check the parameters of the service calls. The settings of configuration information are to also be checked.

(b) The shared stack is selected when using the function to share the stack area by multiple tasks.

Note: When the shared stack function is to be used in an application, be sure to select the library that allows this. Normal system operation cannot be guaranteed if a different library is selected.

Table 8.2 Kernel Library Files

File Name*	Function Enabled/Disabled	
	Parameter Check	Shared Stack
hiknl_nn.lib	Disabled (n)	Disabled (n)
hiknl_ns.lib	Disabled (n)	Enabled (s)
hiknl_pn.lib	Enabled (p)	Disabled (n)
hiknl_ps.lib	Enabled (p)	Enabled (s)

Note: A file name is indicated in the format of hiknl_***xy***_lib. ***x*** indicates whether the parameter check function is enabled or not (p: enabled, or n: disabled). ***y*** indicates whether the shared stack function is enabled or not (s: enabled, or n: disabled).

8.2 Section Configuration

The allocation address of each module is determined in section units at the time of linkage. The sections are described here.

Table 8.3 lists the section names for the supplied files.

Sections of the HI1000/4 listed in Table 8.3 must all be allocated in a big-endian space. The areas used for service call parameters and return parameters must also be allocated in a big-endian space.

The first letter in a section name gives the section attribute.

P Attribute: Program sections, which can be located in ROM.

C Attribute: Constant sections, which can be located in ROM.

B Attribute: Non-initialized data sections, which must be located in RAM.

D or R Attribute: D attribute is an initialized data section, which can be located in ROM. When locating a D-attribute section in ROM, the contents of the D-attribute section must be copied to RAM before executing the program so as to enable the contents to be treated as variables. To be specific, the following procedures are required:

1. Create an R-attribute section with the same size as the D-attribute section by using the ROM support function provided by the optimizing linkage editor. Allocate the R-attribute section to a RAM.
2. Create a program for copying the contents of the D-attribute section to the R-attribute section and execute it at program initiation (usually a CPU initialization routine).

Table 8.3 Section Names

Directory and File Names*1	Section Name	Description
Kernel library	P_hiknl	Kernel program
kernel\samples\mmmm\nnnnzsmp\src \kernel_setup.src (configurator output file)	C_hisetup	Configuration information
	B_histack	Task stack area
	B_hiintstk	Interrupt stack area
	B_hitrc	Trace buffer
	B_hidtq	Data queue area
	B_himpf	Fixed-size memory pool area
	B_himpl	Variable-size memory pool area
	B_hidbginf	E6000H RTOS debugging area
	B_hiwrk	Kernel work area
kernel\samples\mmmm\nnnnzsmp\src \kernel_sysini.src	P_hiknl	Kernel initialization routine
kernel\samples\mmmm\nnnnzsmp\src \kernel_vector.src (configurator output file)	C_hiresvct*2 C_hivct	Vector table
kernel\samples\mmmm\nnnnzsmp\src \nnnnz_sysdwn.c	P_hisysdwn	System down routine
kernel\samples\mmmm\nnnnzsmp\src \nnnnz_ilint.src	P_hiknl	Processing for acquiring detailed information on undefined interrupt
kernel\samples\mmmm\nnnnzsmp\src \nnnnz_idle.c	P_hiidle	Kernel idling routine
kernel\samples\mmmm\nnnnzsmp\src \nnnnz_cpu.c kernel\samples\mmmm\nnnnzsmp\src \nnnnz_cpuasm.src	P_hicpuini	CPU initialization routine
kernel\samples\mmmm\nnnnzsmp\src \nnnnz_tmrdrv.c	P_hitmrdrv	Timer driver
Application files	Arbitrary	—

Notes: 1. ***mmmm*** and ***nnnn*** in the directory and file names indicate the family and CPU names used, respectively. ***z*** indicates the CPU operating mode (a: advanced mode, n: normal mode).

2. C_hiresvct is only output when the vector table format with division is specified by the configurator.

8.3 Creation of Load Module

Open the sample workspace file ***nnnnz***smp.hws appropriate to the device.

8.3.1 Adding Files to a Project

Table 8.4 lists the source program sample files to be added to the project. The sample project file already contains the files shown in this table.

Table 8.4 Source Program Files Added to Project

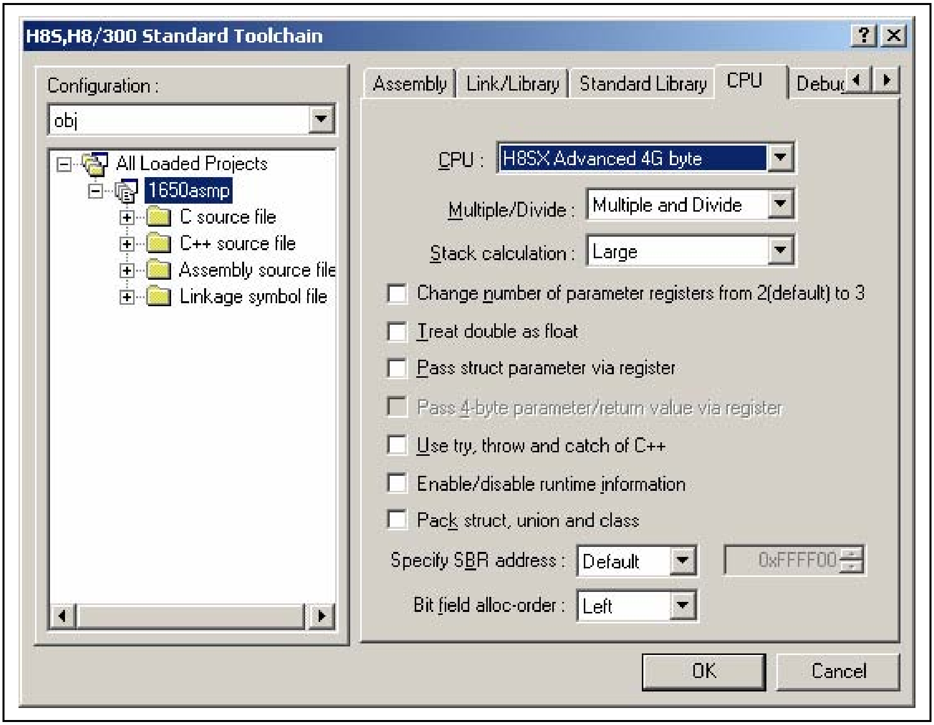
Directory and File Names*	Description	Remarks
kernel\samples\mmmm\nnnnzsmp\src \kernel_setup.src	Setup file	Mandatory
kernel\samples\mmmm\nnnnzsmp\src \kernel_sysini.src	Initialization routine	Mandatory
kernel\samples\mmmm\nnnnzsmp\src \kernel_vector.src	Interrupt vector table	Mandatory
kernel\samples\mmmm\nnnnzsmp\src \nnnnz_sysdwn.c	System down routine	Mandatory
kernel\samples\mmmm\nnnnzsmp\src \nnnnz_ilint.src	Processing for acquiring detailed information on undefined interrupt	Mandatory
kernel\samples\mmmm\nnnnzsmp\src \nnnnz_idle.c	System idling routine	Mandatory
kernel\samples\mmmm\nnnnzsmp\src \nnnnz_cpu.c kernel\samples\mmmm\nnnnzsmp\src \nnnnz_cpuasm.src	CPU initialization routine	Mandatory for executing by reset
kernel\samples\mmmm\nnnnzsmp\src \nnnnz_tmrdrv.c	Timer driver	
kernel\samples\mmmm\nnnnzsmp\src \task.c	Sample task	
Application files	—	

Note: ***mmmm*** and ***nnnn*** in the directory and file names indicate the family and CPU names used, respectively. ***z*** indicates the CPU operating mode (a: advanced mode, n: normal mode).

8.3.2 Setting CPU, Compiler, and Assembler Options

Specify the options of the CPU, compiler, and assembler using the H8S, H8/300 Standard Toolchain.

CPU Tab: Figure 8.2 shows the CPU tab window.



Configuration : obj

CPU : H8SX Advanced 4G byte

Multiple/Divide : Multiple and Divide

Stack calculation : Large

□ Change number of parameter registers from 2(default) to 3

□ Treat double as float

□ Pass struct parameter via register

□ Pass 4-byte parameter/return value via register

□ Use try, throw and catch of C++

□ Enable/disable runtime information

□ Pack struct, union and class

Specify SBR address : Default 0xFFFF00

Bit field alloc-order : Left

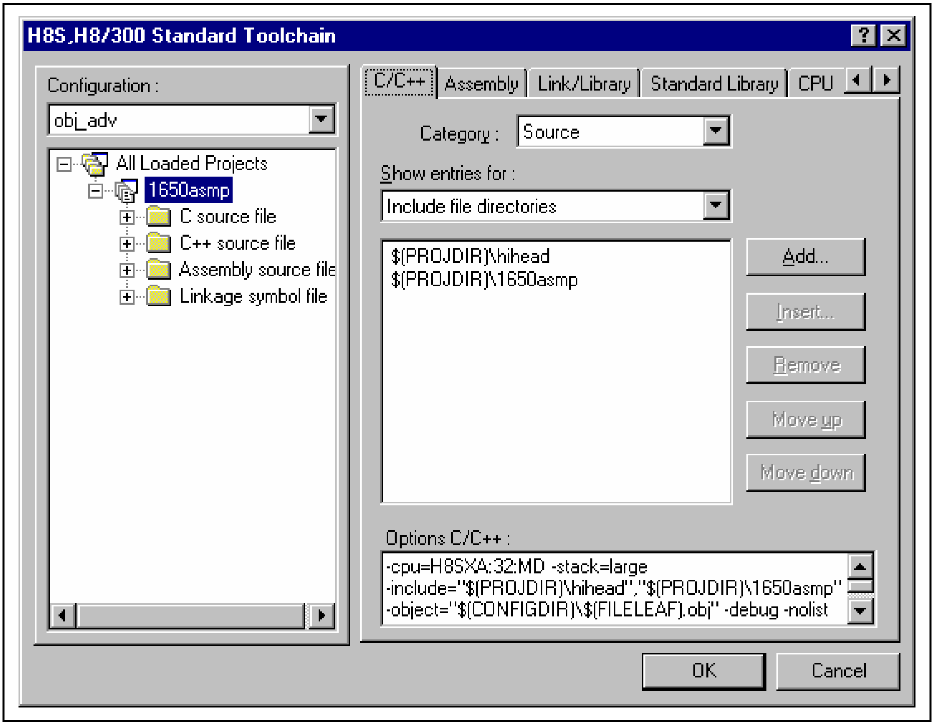
Figure 8.2 CPU Tab

Set the following in the CPU tab window.

- [CPU]: Specify the CPU type to be used.
- [Multiple/Divide]: Specify whether the multiplier/divider is used or not.
- [Stack calculation]: Specify "Large" or "Medium" for the stack address calculation.

  For the sample workspace, "Large" is selected. When "Medium" is selected, the stack address is calculated in two bytes, and therefore, the stack must be aligned to a 2-byte boundary.

C/C++ Tab or Assembly Tab: Figure 8.3 shows the C/C++ tab window.



Category : Source

Show entries for : Include file directories

```
$(PROJDIR)\hihead
$(PROJDIR)\1650asmp
```

Options C/C++ :

```
-cpu=H8SXA:32:MD -stack=large
-include="$(PROJDIR)\hihead","$(PROJDIR)\1650asmp"
-object="$(CONFIGDIR)\$(FILELEAF).obj" -debug -nolist
```

Figure 8.3 C/C++ Tab

Specify the include file directory in the Source category of the C/C++ tab window.

- [Include file directories] from [Source]:
  Specify the following:
  - $(WORKSPDIR)\..\..\..\hihead (header file storage directory)
  - $(WORKSPDIR)\..\..\..\hisys (system file storage directory)
  - $((WORKSPDIR)\src) (sample source file storage directory)

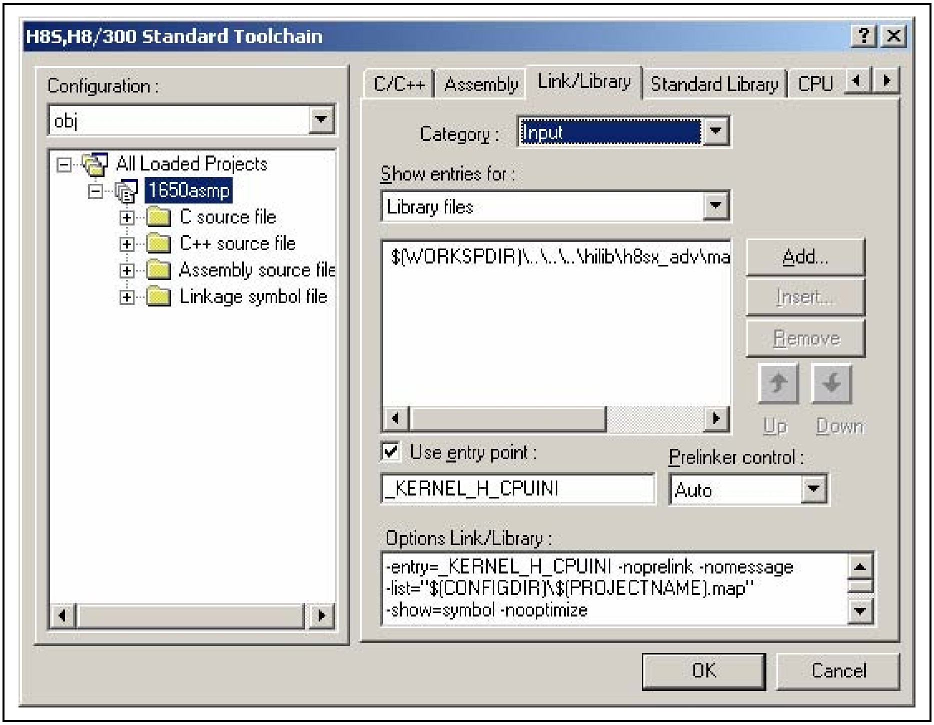
Similarly, specify the include file directory in the Source category of the Assembly tab window.

- [Include file directories] from [Source]:
  Specify the following:
  - $(WORKSPDIR)\..\..\..\hihead (header file storage directory)
  - $(WORKSPDIR)\..\..\..\hisys (system file storage directory)
  - $((WORKSPDIR)\src) (sample source file storage directory)

8.3.3 Setting Optimizing Linkage Editor Options

Specify the options of the optimizing linkage editor using the Link/Library tab of the H8S, H8/300 Standard Toolchain.

Input Category:



Configuration : obj

Category : Input

Show entries for : Library files

$(WORKSPDIR)\..\..\..\hilib\h8sx_adv\ma

☑ Use entry point : _KERNEL_H_CPUINI

Prelinker control : Auto

Options Link/Library :

```
-entry=_KERNEL_H_CPUINI -noprelink -nomessage
-list="$(CONFIGDIR)\$(PROJECTNAME).map"
-show=symbol -nooptimize
```

Figure 8.4 Input Category of Link/Library Tab

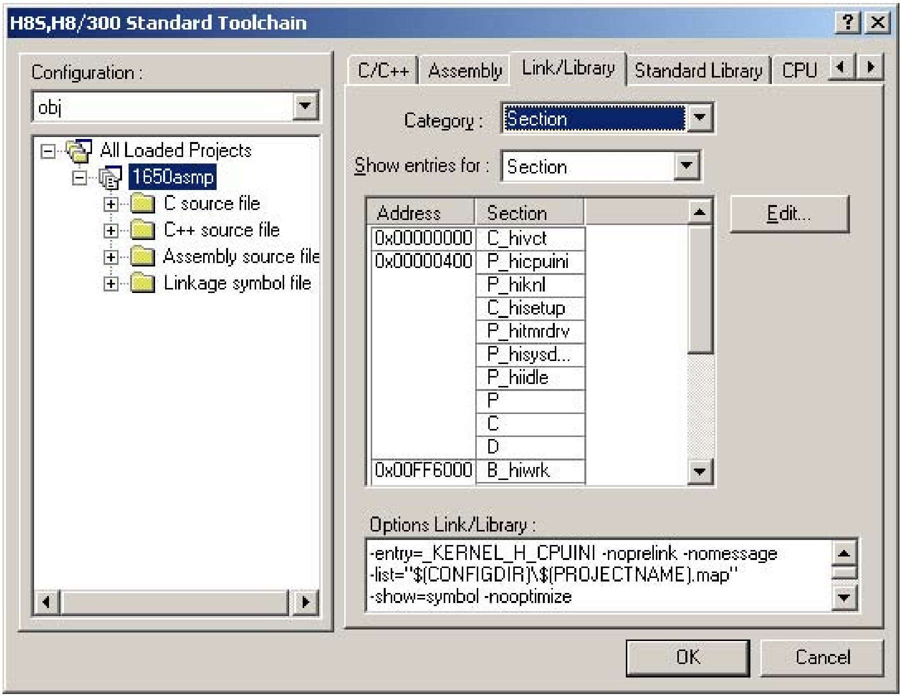
Specify library files, etc. in the Input category of the Link/Library tab.

- [Library files] of [Input]: Specify the kernel libraries to be used from Table 8.2 and Table 8.3.
- [Use entry point]: Specify the CPU initialization routine _KERNEL_H_CPUINI.

Section Category: Specify the allocation addresses of each section in the Section category of the Link/Library tab.

Specify the allocation addresses for the sections in the input files according to the memory structure of the target hardware.

For the HI1000/4 sections, refer to the section list in Table 8.3.



Configuration : obj

Category : Section

Show entries for : Section

Address	Section
0x00000000	C_hivct
0x00000400	P_hicpuini
	P_hiknl
	C_hisetup
	P_hitmrdrv
	P_hisysd...
	P_hiidle
	P
	C
	D
0x00FF6000	B_hiwrk

Options Link/Library :

```
-entry=_KERNEL_H_CPUINI -noprelink -nomessage
-list="$(CONFIGDIR)\$(PROJECTNAME).map"
-show=symbol -nooptimize
```

Figure 8.5 Section Category of Link/Library Tab

Basically, specify addresses for all sections in the input files. If a section name not specified exists in the input files or a section name specified does not exist in the input files, the optimizing linkage editor displays a warning message but will perform linkage normally.

The default settings, for example, sometimes display such warning messages if the application files do not have P, C, D, B, or R sections because these sections are not used in the linked application object. However, these warnings in no way affect use of the resulting load module.

Notes on section allocation are given below:

- Interrupt vector table (section names: C_hiresvct and C_hivct)

  When the vector table format without division is selected by the configurator, the interrupt vector table (section name: C_hivct) must be allocated to address H'0 and the VBR register value must be fixed at 0.

  When the vector table format with division is selected by the configurator, the reset vector (section name: C_hiresvct) must be allocated to address H’0 and the interrupt vector table (section name: C_hivct) must be allocated to the address specified in the VBR register.
- Kernel (section name: P_hiknl)

  Allocate the kernel from an even address. In addition, allocate the section between addresses H’xxxx0000 to H’xxxxFFFF. The upper addresses “xxxx” of the addresses to where the kernel is allocated should be equal.
- Kernel work area (section name: B_hiwrk)

  Allocate the kernel work area from an even address. In addition, allocate the section between addresses H’xxxx0000 to H’xxxxFFFF. The upper addresses “xxxx” of the addresses to where the kernel work area is allocated should be equal.
- Configuration information area (section name: C_hisetup)

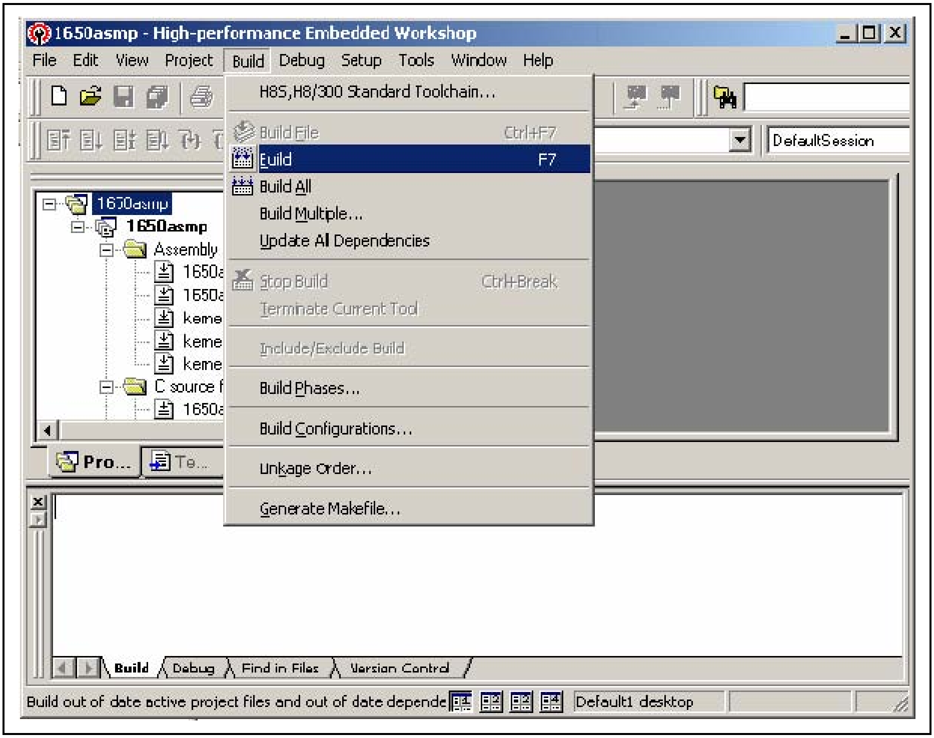
  Allocate the configuration information area from an even address. In addition, allocate the section between addresses H’xxxx0000 to H’xxxxFFFF. The upper addresses “xxxx” of the addresses to where the configuration information area is allocated should be equal.
- E6000H RTOS debugging information area (section name: B_hidbginf)

  The E6000H RTOS debugging information area is a section used when the task debugging function is used in the E6000H emulator. Allocate this area in the I/O area. For the I/O area addresses, refer to the user's manual of the E6000H used.

8.3.4 Executing a Build

The load module is created by executing a build after adding application files to the project and setting the compile, assemble, and optimizing linkage editor options.

To execute a build, choose [Build] or [Build All] from the [Build] menu in HEW as shown in Figure 8.6.



Build out of date active project files and out of date depende

Figure 8.6 Build Execution

Section 9 Calculation of Work Area Size

9.1 Work Areas

To facilitate memory allocation, a section is assigned for each work area as listed in Table C.1. Allocate these sections to suitable addresses at linkage.

Table 9.1 Work Areas

Work Area	Section Name	File Defining Sections
Stack area	B_histack	product\sample\nnnnzsmp\kernel_setup.src
Interrupt handler stack area	B_hiintstk	
Trace buffer area	B_hitrc	
Data queue area	B_hidtq	
Fixed-size memory pool area	B_himpf	
Variable-size memory pool area	B_himpl	
Kernel work area	B_hiwrk	product\sample\setup.inc
Debugging information management area	B_hidbginf	
Work area used by application program	Determined by user	Determined by user

For the size of each section, refer to the assemble listing.

Stack Area (Section B_histack): Stack area defined and allocated by the configurator.

Interrupt Handler Stack Area (Section B_hiintstk): Interrupt handler stack area defined and allocated by the configurator.

Trace Buffer Area (Section B_hitrc): The trace buffer area is allocated when the trace function is included.

Data Queue Area (Section B_hidtq): Area assigned to the data queues.

Fixed-Size Memory Pool Area (Section B_himpf): Area for fixed-size memory pools.

Variable-Size Memory Pool Area (Section B_himpl): Area for variable-size memory pools.

Kernel Work Area (Section B_hiwrk): Used for kernel operation; contains the task control block (TCB), event flag management block (FLGCB), and kernel stack area.

Debugging Information Management Area (Section B_hidbginf): Area where debugging information is to be output.

Work Area Used by Application Program: Area for variables used by application programs.

9.2 Stack Types

Each task or handler requires its own contiguous stack area. If a stack overflows, the system will operate incorrectly. Therefore, the user must determine the stack size required for execution of each task or handler and allocate enough area for each task or handler by referring to the following description. The stack types are shown below.

Task Stack: An independent stack used by each task ID. The kernel switches task stacks at task scheduling.

The task stack is switched by the kernel. Accordingly, the stack must not be switched by the task.

Interrupt Handler Stack: When an interrupt occurs, the stack must be allocated and switched by the interrupt handler. Unless it is switched, the interrupt handler uses the stack of the task that was executed before the interrupt occurred. Therefore, the task stack may overflow. NMI must be defined as an interrupt handler. However, the NMI has the possibility of re-entry, so stack switching must not be performed by the NMI interrupt handler. Stacks for tasks and handlers must be reserved considering the size used by the NMI interrupt handler since the stack before the NMI occurrence is used by the NMI interrupt handler.

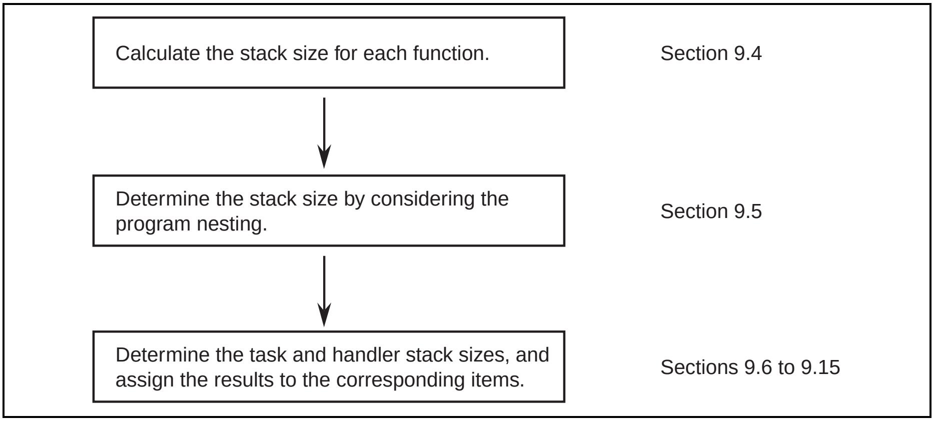
Kernel Stack: Stack used by the kernel. It is also used by the initialization routine. For the stack size used by the timer initialization routine, refer to the description of timer initialization routine in Section 9.10, Initialization Routine Stack.

Stack Used before Kernel Initiation: The stacks used by programs executed before kernel initiation, such as the CPU initialization routine, are not managed by the kernel. Therefore, the user can use the desired area for the stack. The stack pointer at power-on reset must be defined at the beginning of the CPU initialization routine.

For a microcomputer having built-in RAM, allocate the stack at reset to the built-in RAM. For a microcomputer without built-in RAM, the stack (the user system RAM) may not be accessed during reset depending on the bus state controller (BSC) status immediately after reset. In this case, do not run programs that use stacks and do not generate any interrupts or exceptions until the memory becomes accessible by changing the BSC settings. This is because register data is stored in the stack when interrupts or exceptions occur.

9.3 Stack Size Calculation Procedure

Use the stack size calculation procedure shown in Figure 9.1 to define the appropriate sizes in the corresponding definition parts.



Calculate the stack size for each function.

Section 9.4

Determine the stack size by considering the program nesting.

Section 9.5

Determine the task and handler stack sizes, and assign the results to the corresponding items.

Sections 9.6 to 9.15

Figure 9.1 Stack Size Calculation Procedure

9.4 Calculation of Stack Size for Each Function

C Language Function: When a C language function is initiated, a stack frame is allocated in the stack area. 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 STACK FRAME INFORMATION in the compile list output by the C compiler.

An example is shown below.

```
************ SOURCE LISTING ************
Line Pi 0----+----1----+----2----+----3----+----4----+---
FILE NAME: E:\TASK.C
    1     extern int h(char, char *, double );
    2     int
    3     h(char a, register char *b, double c)
    4     {
    5         char *d;
    6         d= &a;
    7         h(*d,b,c);
    8         {
    9              register int i;
   10              i= *d;
   11              return i;
   12         }
   13     }

******* STACK FRAME INFORMATION ********
FILE NAME: E:\TASK.C
Function (File E:\TASK  , Line     3): h
  Optimize Option Specified : No Allocation Information Available

Parameter Area Size     : 0x00000000 Byte(s)
Linkage Area Size       : 0x00000008 Byte(s)
Local Variable Size     : 0x00000000 Byte(s)
Temporary Size          : 0x00000000 Byte(s)
Register Save Area Size : 0x0000000c Byte(s)
Total Frame Size        : 0x00000014 Byte(s)    (1)
(omitted)
```

Figure 9.2 Compile List and Stack Size

The stack area size used by the function is indicated by (1), 20 bytes.

For details on parameters allocated to the parameter area on the stack, refer to the H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor User’s Manual.

Assembly Language Function: To calculate the stack size, examine the stack push and pop (in predecrement and postincrement register indirect addressing mode) instructions used in the program. When parameters are pushed onto the stack at function call, the area size for the parameters must be added to the stack size.

9.5 Stack Size Considering Programming Nesting

A stack size considering programming nesting is calculated with the following program start functions as a base point.

- Tasks
- Interrupt handlers
- Time event handlers
- Initialization routine
- CPU exception handlers (containing TRAPA instruction exception handlers)

Programming nesting includes all functions that are called from these start functions and the following program calls.

Calculate the total value of the stack sizes used by each function and determined according to section 9.4, Calculation of Stack Size for Each Function above for each nesting case.

When the CPU exception handlers (containing TRAPA instruction exception handlers) is nested, add the stack size shown in Table 9.2 for each nesting.

Table 9.2 Additional Stack Size of CPU Exception Handler

Item	Formula	Size (Byte)	Remarks
Stack size used independently by CPU exception handlers	Size used by user		
Stack size for CPU exception handlers	8		
Total			

An example of calculation is shown below where the stack size considering programming nesting is added. The program nesting shown in Figure 9.3 is used as an example.

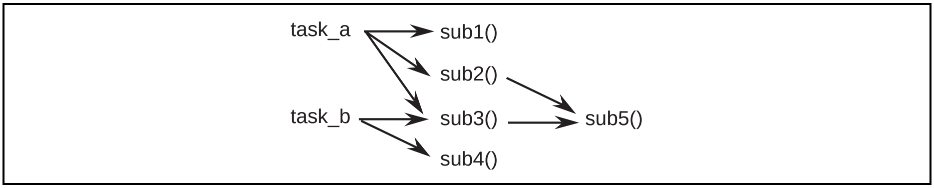


Figure 9.3 Programming Nesting State

The stack size of each function is assumed as follows:

Table 9.3 Stack Size of Each Function

Function	Size (Byte)	Remarks
task_a	56	Start function of task A
task_b	40	Start function of task B
sub1	88	task_a subroutine
sub2	8	task_a subroutine
sub3	24	Common subroutine
sub4	12	task_b subroutine
sub5	80	Common subroutine

The stack sizes of tasks A and B, considering the calling path, are shown in Table 9.4.

Table 9.4 Task Size Considering Calling Path

Task	Calling Path	Task Size (Byte)
Task A	task_a <56 bytes> → sub1 <88 bytes>	144
	task_a <56 bytes> → sub2 <8 bytes> → sub5 <80 bytes >	144
	task_a <56 bytes> → sub3 <24 bytes> → sub5 <80 bytes >	160 (maximum)
Task B	task_b <40 bytes> → sub3 <24 bytes> → sub5 <80 bytes >	144 (maximum)
	task_b <40 bytes> → sub4 <12 bytes>	52

9.6 Task Stacks

Stack Size Used by Each Task: The stack size of each task can be determined by substituting the size obtained according to Section 9.5 above into Table 9.5. Specify the calculated value as the task stack size in the configurator.

When multiple tasks share the same stack, calculate the stack size for each task according to Table 9.5, and then specify the largest size among the calculated values in the configurator.

When the shared stack is used, note that the size to be actually allocated is the size calculated here + 8 bytes used for stack management by the kernel.

Table 9.5 Stack Size Used by Each Task

Item	Formula	Size (Byte)	Remarks
Size obtained in sections 9.4 and 9.5	Size used by user		
Stack size used by OS	58 (MAC registers are used) 50 (MAC registers are not used)	58 or 50	Always necessary.
Addition considering nested interrupts	10 x LOWINTNST*1 + 10 x UPPINTNST*2		(Interrupt control mode 2 or 3 is used.)
	8 x LOWINTNST*1 + 8 x UPPINTNST*2		(Interrupt control mode 0 or 1 is used.)
Stack size for service call trace	6		Necessary when the service call trace function is used.
Stack size for undefined interrupts*3	Advanced mode: 10 Normal mode: 8		
Total			

Notes: 1. Nest count of interrupts equal to or lower than the kernel interrupt mask level

2. Nest count of interrupts (including NMI) higher than the kernel interrupt mask level

3. Necessary when an undefined interrupt occurs

Stack Area Allocation: The stack area for a task is automatically allocated by specifying the stack size for each task by the configurator. The total size used by the entire task stack area is obtained by the following equation:

Total size = Σ((each stack size specified in the configurator) + 8*)

Note: The 8 bytes are added only when the shared stack function is used.

9.7 Interrupt Handler Stacks

Stack Size Used by Each Interrupt Handler: The stack size of each interrupt handler can be determined by substituting the size obtained according to Section 9.5 above into Table 9.6.

Allocate a stack area independently for each interrupt handler.

Table 9.6 Stack Size Used by Each Interrupt Handler

Item	Formula	Size (Byte)	Remarks
Size obtained in sections 9.4 and 9.5	Size used by user		
Stack size used by OS	50		Necessary when a service call is issued.
Addition considering nested interrupts	10 x LOWINTNST[*1] + 10 x UPPINTNST[*2]		(Interrupt control mode 2 or 3 is used.)
	8 x LOWINTNST[*1] + 8 x UPPINTNST[*2]		(Interrupt control mode 0 or 1 is used.)
Stack size for service call trace	6		Necessary when the service call trace function is used.
Stack size for undefined interrupts[*3]	Advanced mode: 10 Normal mode: 8		
Total			

Notes: 1. Nest count of interrupts equal to or lower than the kernel interrupt mask level, and at the same time higher than the current interrupt level

2. Nest count of interrupts (including NMI) higher than the kernel interrupt mask level

3. Necessary when an undefined interrupt occurs

Stack Area Allocation: Since handlers of the same interrupt level are not activated concurrently, allocate the stack area of the interrupt handler that uses the largest stack area from among the same interrupt-level interrupt handlers as the handler stack area of the corresponding interrupt level. Then switch to the stack at the beginning of the interrupt handler. Refer to Section 6.5, Interrupt Handlers, when switching stacks by the interrupt handler. In this case, separate stacks can be used instead of sharing a stack within the same interrupt level handlers.

9.8 Timer Interrupt Stack

Stack Size Used by a Timer Interrupt Handler: The stack size of a timer interrupt handler can be determined by substituting the size into Table 9.7.

Table 9.7 Timer Interrupt Handler Stack Size

Item	Formula	Size (Byte)	Remarks
Size obtained in sections 9.4 and 9.5	Size used by user		
Necessary size	Advanced mode: 52 Normal mode: 50	52 or 50	Always necessary.
Addition considering nested interrupts	10 x LOWINTNST[*1] + 10 x UPPINTNST[*2]		(Interrupt control mode 2 or 3 is used.)
	8 x LOWINTNST[*1] + 8 x UPPINTNST[*2]		(Interrupt control mode 0 or 1 is used.)
Stack size for undefined interrupts[*3]	Advanced mode: 10 Normal mode: 8		
Stack size used independently by cyclic handler[*4]	Size used by user		
	50		Necessary when a service call is issued.
	6		Necessary when the service call trace function is used.
Total			

Notes: 1. Nest count of interrupts equal to or lower than the kernel interrupt mask level, and at the same time higher than the timer interrupt level

2. Nest count of interrupts (including NMI) higher than the kernel interrupt mask level

3. Necessary when an undefined interrupt occurs

4. When more than one cyclic handler is used, the stack size is calculated for each cyclic handler, and the largest stack size among the cyclic handlers is added.
When a cyclic handler is written in C language, the stack size independently used by the cyclic handler should be obtained from the frame size of the function output in the compile list.

Stack Area Allocation: The stack area for a timer interrupt handler is automatically allocated by specifying CFG_TMRSTKSIZ by the configurator.

Timer Interrupt Handler Stack Name: The name of the timer interrupt handler stack is fixed to KERNEL_HI_TIM_SP in the system, and its address indicates the bottom of the timer handler stack.

9.9 Kernel Stack

Stack Size Used by the Kernel: The stack size of the kernel can be determined by substituting the size into Table 9.8.

Table 9.8 Kernel Stack Size

Item	Formula	Size (Byte)	Remarks
Stack size used by OS	Advanced mode: 20 Normal mode: 16	20 or 16	Always necessary.
Addition considering nested interrupts	10 x LOWINTNST[*1] + 10 x UPPINTNST[*2]		(Interrupt control mode 2 or 3 is used.)
	8 x LOWINTNST[*1] + 8 x UPPINTNST[*2]		(Interrupt control mode 0 or 1 is used.)
Stack size for undefined interrupts[*3]	Advanced mode: 10 Normal mode: 8		
Total			

Notes: 1. Nest count of interrupts equal to or lower than the kernel interrupt mask level

2. Nest count of interrupts (including NMI) higher than the kernel interrupt mask level

3. Necessary when an undefined interrupt occurs

Stack Area Allocation: In the stack area for the kernel, the required size is automatically allocated by the configurator. The size of the allocated area is the larger of the following two sizes.

- Stack size calculated in Table 9.8
- Maximum of the stack sizes for all initialization routines according to Table 9.9 in Section 9.10, Initialization Routine Stack + 4

9.10 Initialization Routine Stack

Stack Size Used by Initialization Routine: The stack size of the initialization routine can be determined by substituting the size into Table 9.9.

Table 9.9 Initialization Routine Stack Size

Item	Formula	Size (Byte)	Remarks
Size obtained in sections 9.4 and 9.5	Size used by user		
Stack size used by OS	50	50	Necessary when a service call is issued.
Addition considering nested interrupts	10 x UPPINTNST[*1]		(Interrupt control mode 2 or 3 is used.)
	8 x UPPINTNST[*1]		(Interrupt control mode 0 or 1 is used.)
Stack size for undefined interrupts[*2]	Advanced mode: 10 Normal mode: 8		
Total			

Notes: 1. Nest count of interrupts (including NMI) higher than the kernel interrupt mask level

2. Necessary when an undefined interrupt occurs

Stack Area Allocation: Specify the stack size calculated here when the initialization routine is defined by the configurator. The initialization routine uses the kernel stack, which is automatically allocated for the necessary size by the configurator, considering the stack size of the initialization routine, as described in Section 9.9, Kernel Stack.

Timer Initialization Routine: The name of the timer initialization routine is fixed to KERNEL_HIPRG_TIMINI in the system. If the stack size used by the timer initialization routine is larger than the stack size obtained in the way described in Stack Area Allocation in Section 9.9, Kernel Stack, the stack must be switched in the timer initialization routine.

9.11 Trace Function Stack

Stack Size Used by the Trace Function: The stack size used by the trace function can be determined by substituting the size into Table 9.10.

This stack area is necessary only when the trace function is used.

Table 9.10 Trace Function Stack Size

Item	Formula	Size (Byte)	Remarks
Stack size used by OS	26	26	Normal-version target trace or tool trace is used.
	20	20	Simple-version target trace or tool trace is used.
Addition considering nested interrupts	10 x UPPINTNST[*1]		(Interrupt control mode 2 or 3 is used.)
	8 x UPPINTNST[*1]		(Interrupt control mode 0 or 1 is used.)
Stack size for undefined interrupts[*2]	Advanced mode: 10 Normal mode: 8		
Total			

Notes: 1. Nest count of interrupts (including NMI) higher than the kernel interrupt mask level

2. Necessary when an undefined interrupt occurs

Stack Area Allocation: In the stack area for the trace function, the required size is automatically allocated by selecting the trace function by the configurator.

9.12 Trace Buffer Area

Trace Buffer Area Size: The size of the trace buffer area can be determined by substituting the size into Table 9.11.

This trace buffer area is necessary only when the normal-version or simple-version target trace function is used.

Table 9.11 Trace Buffer Area Size

Item	Formula	Size (Byte)	Remarks
Trace buffer management area size	16	16	
Trace entry information area size	30 x Acquired trace count (TRCCNT) + Acquired object state count (TRCOBJCNT) x 4		Normal-version target trace is used.
	6 x Acquired trace count (TRCCNT)		Simple-version target trace is used.
Total			

Trace Buffer Area Allocation: The trace buffer area is allocated by defining the maximum trace information count and acquired object state count by the configurator.

9.13 Data Queue Area

Data Queue Area Size: The size of a data queue area can be determined by substituting the size into Table 9.12.

The total value of the area sizes obtained for each data queue ID becomes the size used by the entire data queue area.

Table 9.12 Data Queue Area Size

Item	Formula	Size (Byte)	Remarks
Data queue area size	Maximum data queue count (MAXDTQCNT) x 4		
Total			

Data Queue Area Allocation: The data queue area is allocated by specifying the maximum data queue count for each data queue ID by the configurator.

9.14 Fixed-Size Memory Pool Area

Fixed-Size Memory Pool Area Size: The size of a fixed-size memory pool area can be determined by substituting the size into Table 9.13.

The total value of the area sizes obtained for each fixed-size memory pool ID becomes the size used by the entire fixed-size memory pool area.

Table 9.13 Fixed-Size Memory Pool Area Size

Item	Formula	Size (Byte)	Remarks
Fixed-size memory pool area size	Number of fixed-size memory blocks x (Fixed-size memory block size + 4)		A 4-byte management area is necessary for each memory block.
Total			

Fixed-Size Memory Pool Area Allocation: The fixed-size memory pool area is allocated by specifying the number of fixed-size memory blocks and the fixed-size memory block size for each fixed-size memory pool ID by the configurator.

9.15 Variable-Size Memory Pool Area

Variable-Size Memory Pool Area Size: The size of a variable-size memory pool area can be determined by substituting the size into Table 9.14.

The total value of the area sizes obtained for each variable-size memory pool ID becomes the size used by the entire variable-size memory pool area.

Table 9.14 Variable-Size Memory Pool Area Size

Item	Formula	Size (Byte)	Remarks
Variable-size memory pool area size	Variable-size memory pool size + (16 x n*)		A 16-byte management area is necessary for each acquired memory block.
Total			

Note: n: Maximum number of variable-size memory blocks to be acquired

Variable-Size Memory Pool Area Allocation: The variable-size memory pool area is allocated by specifying the variable-size memory pool area size obtained in the above table for each variable-size memory pool ID by the configurator.

9.16 Kernel Work Area

Kernel Work Area Size: The size of the kernel work area can be determined by substituting the size into Table 9.15.

Calculate the work area used by the kernel using Table 9.15.

Table 9.15 Kernel Work Area Size

Item	Formula	Size (Byte)	Remarks
System management table (_KERNEL_HI_SYSMT)	14 + 4 x Maximum task priority (CFG_MAXTSKPRI)		Always necessary.
Task control block (_KERNEL_HI_TCB)	18 x Maximum task ID (CFG_MAXTSKID)		Always necessary.
Task control block 2 (_KERNEL_HI_TCB2)	8 x Maximum task ID (CFG_MAXTSKID)		Necessary when a service call with the timeout function is used.
Task control block 3 (_KERNEL_HI_TCB3)	2 x Maximum task ID (CFG_MAXTSKID)		Always necessary.
Event flag control block (_KERNEL_HI_FLGCB)	6 x Maximum event flag ID (CFG_MAXFLGID)		Necessary when an event flag is used.
Semaphore control block (_KERNEL_HI_SEMCB)	6 x Maximum semaphore ID (CFG_MAXSEMID)		Necessary when a semaphore is used.
Mailbox control block (_KERNEL_HI_SEMCB)	8 x Maximum mailbox ID (CFG_MAXMBXID)		Necessary when a mailbox is used.
Data queue control block 1 (_KERNEL_HI_DTQCB1)	14 x Maximum data queue ID (CFG_MAXDTQID)		Necessary when a data queue is used.
Data queue control block 2 (_KERNEL_HI_DTQCB2)	1 x Maximum data queue ID (CFG_MAXDTQID)		Necessary when a data queue is used.
Mutex control block (_KERNEL_HI_MTXCB)	10 x Maximum mutex ID (CFG_MAXMTXID)		Necessary when a mutex is used.
Task-lock mutex control block (_KERNEL_HI_TLMXCB)	4 x Maximum task ID (CFG_MAXTSKID)		Necessary when a mutex is used.
Fixed-size memory pool control block (_KERNEL_HI_MPFCB)	6 x Maximum fixed-size memory pool ID (CFG_MAXMPFID)		Necessary when a fixed-size memory pool is used.
Variable-size memory pool control block (_KERNEL_HI_MPLCB)	20 x Maximum variable-size memory pool ID (CFG_MAXMPLID)		Necessary when a variable-size memory pool is used.
Cyclic handler control block (_KERNEL_HI_CYHCB)	24 x Maximum cyclic handler ID (CFG_MAXCYCID)		Necessary when a cyclic handler is used.

Table 9.15 Kernel Work Area Size (cont)

Item	Formula	Size (Byte)	Remarks
Timer control blocks[*1, *2] (_KERNEL_HI_TIMCB, (_KERNEL_HI_TIMCB2, (_KERNEL_HI_TIMCB3)	10 + 4 + 14 (10 = TIMCB, 4 = TIMCB2, 14 = TIMCB3)		TIMCB: Necessary when the timer driver is used. TIMCB2: Necessary when a service call with the timeout function is used. TIMCB3: Necessary when a cyclic handler is used.
Trace buffer control block[*3] (_KERNEL_TBACB)	4		Necessary when the service call trace function is used.
TX trace control block (_KERNEL_TX_TRCCB)	4 + Acquired object state count (TRCOBJCNT) x 4		Necessary when the target trace or normal-version tool trace function is used.
	4		Necessary when the simple-version tool trace is used.
Total			

Notes: 1. When the timeout function check box (CFG_TOUTUSE) in the time management function selection is not checked by the configurator, areas used by the timeout function (TCB2 and TIMCB2 areas) are not allocated.

2. When the time management function check box (CFG_TIMUSE) in the time management function selection is not checked by the configurator, the timer control blocks (TIMCB, TIMCB2, and TIMCB3 areas) and timer related control blocks (TCB2 and CYHCB areas) are not allocated.

3. When the service call trace function check box (CFG_TRACE) in the debugging function selection is not checked by the configurator, the trace buffer control block is not allocated.

Section 10 Information during System Down

The system down routine is called when the system goes down. Information listed in Table 10.1 is passed to the system down routine.

Table 10.1 Information Passed to the System Down Routine

		Packet Contents				
Cause of System Going Down	**Error Type H type (R0)**	**System Down Information 1 H inf1 (E0)**	**System Down Information 2 B inf2 (R1L)**	**System Down Information 3 B inf3 (R1H)**	**System Down Information 4 H inf4 (E1)**	**System Down Information 5 UW inf5 (ER2)**
Configuration information error	H'fffb	Error code	H'00	H'00	H'0000	H'00000000
Service call ext_tsk being issued by the non-task context	H'fffd	E_CTX	CCR at exception occurrence	EXR at exception occurrence	0	PC at exception occurrence
Routine ret_int called from the task execution state or CPU-locked state	H'fffe	H'0000	tskid	H'00	H'0000	H'00000000
Undefined interrupt occurred	H'ffff	Interrupt vector number	CCR at exception occurrence	EXR at exception occurrence	tskid*	PC at exception occurrence

Note: When an undefined interrupt occurs in the non-task context state or CPU-locked state, H'0000 is passed to the system down routine.

Table 10.2 lists the invalid contents and error number of the configuration information errors.

Table 10.2 Invalid Contents of Configuration Information

No.		Invalid Item	Error Number
1	Invalid address	Kernel stack pointer (_KERNEL_HI_OS_SP) is 0 or an odd value	H’0101
		Timer interrupt stack pointer (_KERNEL_HI_TIM_SP) is 0 or an odd value	H’0102
		Start address of kernel work area (section name is B_hiknl) is 0 or an odd value	H’0103
		Start address of TIMCB area (_KERNEL_HI_TIMCB) is 0 or an odd value	H’0104
		Start address of TIMCB2 area (_KERNEL_HI_TIMCB2) is 0 or an odd value	H’0105
		Start address of TCB area (_KERNEL_HI_TCB) is 0 or an odd value	H’0106
		Start address of TCB2 area (_KERNEL_HI_TCB2) is 0 or an odd value	H’0107
		Start address of FLGCB area (_KERNEL_HI_FLGCB) is 0 or an odd value	H’0108
		Start address of SEMCB area (_KERNEL_HI_SEMCB) is 0 or an odd value	H’0109
		Start address of MBXCB area (_KERNEL_HI_MBXCB) is 0 or an odd value	H’010A
		Start address of MPFCB area (_KERNEL_HI_MPFCB) is 0 or an odd value	H’010B
		Start address of MPLCB area (_KERNEL_HI_MPLCB) is 0 or an odd value	H’010C
		Trace stack pointer (_KERNEL_HI_TRC_SP) is 0 or an odd value	H’010D
		Start address of trace management area (_KERNEL_TBACB) is 0 or an odd value	H’010E
		Start address of TIMCB3 area (_KERNEL_HI_TIMCB3) is 0 or an odd value	H’010F
		Start address of CYHCB area (_KERNEL_HI_CYHCB) is 0 or an odd value	H’0110
		Start address of TCB3 area (_KERNEL_HI_TCB3) is 0 or an odd value	H’0111
		Start address of DTQCB1 area (_KERNEL_HI_DTQCB1) is 0 or an odd value	H’0112
		Start address of MTXCB area (_KERNEL_HI_MTXCB) is 0 or an odd value	H’0113

Table 10.2 Invalid Contents of Configuration Information (cont)

No.		Invalid Item	Error Number
1	Invalid address (cont)	Start address of TLMXCB area (_KERNEL_HI_TLMXCB) is 0 or an odd value	H'0114
2	Invalid routine address	Start address of system initialization handler (_KERNEL_HIPRG_SYSINI) is an odd value	H'0201
		Start address of timer initialization setting routine (_KERNEL_HIPRG_TIMINI) is an odd value	H'0202
3	Invalid setting (out of range)	Interrupt control mode (CFG_INTMD) is 4 or higher	H'0301
		Kernel interrupt mask level (CFG_KNLMSKLVL) is 9 or higher	H'0302
		Maximum task priority (CFG_MAXTSKPRI) is 32 or higher	H'0303
		Maximum task ID (CFG_MAXTSKID) is 256 or higher	H'0304
		Maximum event flag ID (CFG_MAXFLGID) is 256 or higher	H'0305
		Maximum semaphore ID (CFG_MAXSEMID) is 256 or higher	H'0306
		Maximum mailbox ID (CFG_MAXMBXID) is 256 or higher	H'0307
		Maximum fixed-size memory pool ID (CFG_MAXMPFID) is 256 or higher	H'0308
		Maximum variable-size memory pool ID (CFG_MAXMPLID) is 256 or higher	H'0309
		Maximum cyclic handler ID (CFG_MAXCYCID) is 256 or higher	H'030A
		Maximum data queue ID (CFG_MAXDTQID) is 256 or higher	H'030B
		Maximum mutex ID (CFG_MAXMTXID) is 256 or higher	H'030C
		Maximum message priority (CFG_MAXMSGPRI) is 256 or higher	H'030D
4	Invalid table address (0 or odd value)	Start address of task definition table (_KERNEL_HI_TDT) is 0 or an odd value	H'0401
		Start address of fixed-size memory pool definition table (_KERNEL_HI_MPFDT) is 0 or an odd value	H'0402
		Start address of variable-size memory pool definition table (_KERNEL_HI_MPLDT) is 0 or an odd value	H'0403
		Start address of undefined interrupt handler (_KERNEL_HI_ILT) is 0 or an odd value	H'0404
		Start address of trace buffer information table (_KERNEL_INITRC) is 0 or an odd value	H'0405
		Start address of cyclic handler definition table (_KERNEL_HI_CYCDT) is 0 or an odd value	H'0406

Table 10.2 Invalid Contents of Configuration Information (cont)

No.		Invalid Item	Error Number
4	Invalid table address (0 or odd value) (cont)	Start address of semaphore definition table (_KERNEL_HI_SEMDT) is 0 or an odd value	H’0407
		Start address of event flag definition table (_KERNEL_HI_FLGDT) is 0 or an odd value	H’0408
		Start address of mailbox definition table (_KERNEL_HI_MBXDT) is 0 or an odd value	H’0409
		Start address of data queue definition table (_KERNEL_HI_DTQDT) is 0 or an odd value	H’040A
		Start address of time definition table (_KERNEL_HI_TIMDT) is 0 or an odd value	H’040B
5	Invalid item	Priority at task initiation is 0 or a value greater than the maximum task priority (CFG_MAXTSKPRI)	H’0501
		Task start address is 0 or an odd value	H’0502
		Task stack pointer is 0 or an odd value	H’0503
		Fixed-size memory block size (BLFLEN) is 0, an odd value, or 65530 bytes or more	H’0504
		Start address of fixed-size memory pool area is 0 or an odd value	H’0505
		Variable-size memory pool size is 0, an odd value, or 16 bytes or less	H’0506
		Start address of variable-size memory pool area is 0 or an odd value	H’0507
		Trace buffer address (TRACE BUFFER ADDRESS) is 0 or an odd value	H’0508
		Start address of cyclic handler is 0 or an odd value	H’0509
		Initiation cycle of cyclic handler is 0 or H’80000000 or more	H’050A
		Initiation phase of cyclic handler is greater than the initiation cycle	H’050B
		Initial value of number of semaphore resources is greater than the maximum number of semaphore resources	H’050C
		Event flag attribute is invalid	H’050D
		Message priority is greater than the maximum message priority (CFG_MAXMSGPRI)	H’050E
		Start address of data queue area is greater than the end address	H’050F
		Maximum mutex priority is greater than the maximum task priority (CFG_MAXTSKPRI)	H’0510

Table 10.2 Invalid Contents of Configuration Information (cont)

No.		Invalid Item	Error Number
5	Invalid item (cont)	Denominator of the time tick cycle is greater than 100	H’0511
		Neither the denominator nor numerator of the time tick cycle is 1	H’0512
6	Invalid area allocation	Area of section P_hiknl is not allocated in the range of addresses H’xxxx0000 to H’xxxxFFFF (xxxx is the same)	H’0601
		Area of section C_hisetup is not allocated in the range of addresses H’xxxx0000 to H’xxxxFFFF (xxxx is the same)	H’0602
		Area of section B_hiwrk is not allocated in the range of addresses H’xxxx0000 to H’xxxxFFFF (xxxx is the same)	H’0603

Section 11 Reference Listing

11.1 Service Calls

Task Management Function

No.	Service Call	C-Language API	Function
1	act_tsk	ER ercd= act_tsk (ID tskid);	Initiate task
	iact_tsk	ER ercd= iact_tsk (ID tskid);	
2	can_act	ER_UINT actcnt= can_act (ID tskid);	Cancel task initiation request
3	sta_tsk	ER ercd= sta_tsk (ID tskid, VP_INT stacd);	Start task (specifies start code)
	ista_tsk	ER ercd= ista_tsk (ID tskid, VP_INT stacd);	
4	ext_tsk	void ext_tsk ();	Exit current task
5	ter_tsk	ER ercd= ter_tsk (ID tskid);	Terminate task
6	chg_pri	ER ercd= chg_pri (ID tskid, PRI tskpri);	Change task priority
7	get_pri	ER ercd= get_pri (ID tskid, PRI *p_tskpri);	Refer to task priority
8	ref_tsk	ER ercd= ref_tsk (ID tskid, T_RTSK *pk_rtsk);	Refer to task state
	iref_tsk	ER ercd= iref_tsk (ID tskid, T_RTSK *pk_rtsk);	
9	ref_tst	ER ercd= ref_tst (ID tskid T_RTST *pk_rtst);	Refer to task state (simple version)
	iref_tst	ER ercd= iref_tst (ID tskid T_RTST *pk_rtst);	

Task Synchronous Management Function

No.	Service Call	C-Language API	Function
10	slp_tsk	ER ercd= slp_tsk ();	Sleep task
11	tslp_tsk	ER ercd= tslp_tsk (TMO tmout);	Sleep task with timeout
12	wup_tsk	ER ercd= wup_tsk (ID tskid);	Wakeup task
	iwup_tsk	ER ercd= iwup_tsk (ID tskid);	
13	can_wup	ER UINT wupcnt= can_wup (ID tskid);	Cancel wakeup task
14	rel_wai	ER ercd= rel_wai (ID tskid);	Release WAITING state forcibly
	irel_wai	ER ercd= irel_wai (ID tskid);	
15	sus_tsk	ER ercd= sus_tsk (ID tskid);	Shift to SUSPENDED state
16	rsm_tsk	ER ercd= rsm_tsk (ID tskid);	Resume task from SUSPENDED state
17	frsm_tsk	ER ercd= frsm_tsk (ID tskid);	Resume task from SUSPENDED state forcibly
18	dly_tsk	ER ercd= dly_tsk (RELTIM dlytim);	Delay task

No.	Service Call	C-Language API	Function
Synchronization and Communication Function			
Semaphore			
19	sig_sem	ER ercd= sig_sem (ID semid);	Return semaphore resource
	isig_sem	ER ercd= isig_sem (ID semid);	
20	wai_sem	ER ercd= wai_sem (ID semid);	Wait for semaphore
21	pol_sem	ER ercd= pol_sem (ID semid);	Poll and wait for semaphore
	ipol_sem	ER ercd= ipol_sem (ID semid);	
22	twai_sem	ER ercd= twai_sem (ID semid, TMO tmout);	Wait for semaphore with timeout
23	ref_sem	ER ercd= ref_sem (ID semid, T_RSEM *pk_rsem);	Refer to semaphore state
	iref_sem	ER ercd= iref_sem (ID semid, T_RSEM *pk_rsem);	
Event Flag			
24	set_flg	ER ercd= set_flg (ID flgid, FLGPTN setptn);	Set event flag
	iset_flg	ER ercd= iset_flg (ID flgid, FLGPTN setptn);	
25	clr_flg	ER ercd= clr_flg (ID flgid, FLGPTN clrptn);	Clear event flag
	iclr_flg	ER ercd= iclr_flg (ID flgid, FLGPTN clrptn);	
26	wai_flg	ER ercd= wai_flg (ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn);	Wait for event flag
27	pol_flg	ER ercd= pol_flg (ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn);	Poll and wait for event flag
	ipol_flg	ER ercd= ipol_flg (ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn);	
28	twai_flg	ER ercd= twai_flg (ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn, TMO tmout);	Wait for event flag with timeout
29	ref_flg	ER ercd= ref_flg (ID flgid, T_RFLG *pk_rflg);	Refer to event flag state
	iref_flg	ER ercd= iref_flg (ID flgid, T_RFLG *pk_rflg);	
Data Queue			
30	snd_dtq	ER ercd= snd_dtq (ID dtqid, VP_INT data);	Send data to data queue
31	psnd_dtq	ER ercd= psnd_dtq (ID dtqid, VP_INT data);	Poll and send data to data queue
	ipsnd_dtq	ER ercd= ipsnd_dtq (ID dtqid, VP_INT data);	
32	tsnd_dtq	ER ercd= tsnd_dtq (ID dtqid, VP_INT data, TMO tmout);	Send data to data queue with timeout
33	fsnd_dtq	ER ercd= fsnd_dtq (ID dtqid, VP_INT data);	Send data to data queue forcibly
	ifsnd_dtq	ER ercd= ifsnd_dtq (ID dtqid, VP_INT data);	
34	rcv_dtq	ER ercd= rcv_dtq (ID dtqid, VP_INT *p_data);	Receive data from data queue
35	prcv_dtq	ER ercd= prcv_dtq (ID dtqid, VP_INT *p_data);	Poll and receive data from data queue

No.	Service Call	C-Language API	Function
36	trcv_dtq	ER ercd= trcv_dtq (ID dtqid, VP_INT *p_data, TMO tmout);	Receive data from data queue with timeout
37	ref_dtq	ER ercd= ref_dtq (ID dtqid, T_RDTQ *pk_rdtq);	Refer to data queue state
	iref_dtq	ER ercd= iref_dtq (ID dtqid, T_RDTQ *pk_rdtq);	
Mailbox			
38	snd_mbx	ER ercd= snd_mbx (ID mbxid, T_MSG *pk_msg);	Send message to mailbox
	isnd_mbx	ER ercd= isnd_mbx (ID mbxid, T_MSG *pk_msg);	
39	rcv_mbx	ER ercd= rcv_mbx (ID mbxid, T_MSG **ppk_msg);	Receive message from mailbox
40	prcv_mbx	ER ercd= prcv_mbx (ID mbxid, T_MSG **ppk_msg);	Poll and receive message from mailbox
	iprcv_mbx	ER ercd= iprcv_mbx (ID mbxid, T_MSG **ppk_msg);	
41	trcv_mbx	ER ercd= trcv_mbx (ID mbxid, T_MSG **ppk_msg, TMO tmout);	Receive message from mailbox with timeout
42	ref_mbx	ER ercd= ref_mbx (ID mbxid, T_RMBX *pk_rmbx);	Refer to mailbox state
	iref_mbx	ER ercd= iref_mbx (ID mbxid, T_RMBX *pk_rmbx);	

Extended Synchronization and Communication Function

Mutex

43	loc_mtx	ER ercd= loc_mtx (ID mtxid);	Lock mutex
44	ploc_mtx	ER ercd= ploc_mtx (ID mtxid);	Poll and lock mutex
45	tloc_mtx	ER ercd= tloc_mtx (ID mtxid, TMO tmout);	Lock mutex with timeout
46	unl_mtx	ER ercd= unl_mtx (ID mtxid);	Unlock mutex
47	ref_mtx	ER ercd= ref_mtx (ID mtxid, T_RMTX *pk_rmtx);	Refer to mutex state

Memory Pool Management Function

Fixed-Size Memory Pool

48	get_mpf	ER ercd= get_mpf (ID mpfid, VP *p_blk);	Acquire fixed-size memory block
49	pget_mpf	ER ercd= pget_mpf (ID mpfid, VP *p_blk);	Poll and acquire fixed-size memory block
	ipget_mpf	ER ercd= ipget_mpf (ID mpfid, VP *p_blk);	
50	tget_mpf	ER ercd= tget_mpf (ID mpfid, VP *p_blk, TMO tmout);	Acquire fixed-size memory block with timeout
51	rel_mpf	ER ercd= rel_mpf (ID mpfid, VP blk);	Release fixed-size memory block
52	ref_mpf	ER ercd= ref_mpf (ID mpfid, T_RMPF *pk_rmpf);	Refer to fixed-size memory pool state
	iref_mpf	ER ercd= iref_mpf (ID mpfid, T_RMPF *pk_rmpf);	

No.	Service Call	C-Language API	Function
Variable-Size Memory Pool			
53	get_mpl	ER ercd= get_mpl (ID mplid, UINT blksz, VP *p_blk);	Acquire variable-size memory block
54	pget_mpl	ER ercd= pget_mpl (ID mplid, UINT blksz, VP *p_blk);	Poll and acquire variable-size memory block
	ipget_mpl	ER ercd= ipget_mpl (ID mplid, UINT blksz, VP *p_blk);	
55	tget_mpl	ER ercd= tget_mpl (ID mplid, UINT blksz, VP *p_blk, TMO tmout);	Acquire variable-size memory block with timeout
56	rel_mpl	ER ercd= rel_mpl (ID mplid, VP blk);	Release variable-size memory block
57	ref_mpl	ER ercd= ref_mpl (ID mplid, T_RMPL *pk_rmpl);	Refer to variable-size memory pool state
	iref_mpl	ER ercd= iref_mpl (ID mplid, T_RMPL *pk_rmpl);	

Time Management Function

System Clock Management			
58	set_tim	ER ercd= set_tim (SYSTIM *p_systim);	Set system clock
	iset_tim	ER ercd= iset_tim (SYSTIM *p_systim);	
59	get_tim	ER ercd= get_tim (SYSTIM *p_systim);	Get system clock
	iget_tim	ER ercd= iget_tim (SYSTIM *p_systim);	
Cyclic Handler			
60	sta_cyc	ER ercd= sta_cyc (ID cycid);	Start cyclic handler
	ista_cyc	ER ercd= ista_cyc (ID cycid);	
61	stp_cyc	ER ercd= stp_cyc (ID cycid);	Stop cyclic handler
	istp_cyc	ER ercd= istp_cyc (ID cycid);	
62	ref_cyc	ER ercd= ref_cyc (ID cycid, T_RCYC *pk_rcyc);	Refer to cyclic handler state
	iref_cyc	ER ercd= iref_cyc (ID cycid, T_RCYC *pk_rcyc);	

System Status Management Function

63	rot_rdq	ER ercd= rot_rdq (PRI tskpri);	Rotate ready queue
	irot_rdq	ER ercd= irot_rdq (PRI tskpri);	
64	get_tid	ER ercd= get_tid (ID *p_tskid);	Refer to task ID in running state
	iget_tid	ER ercd= iget_tid (ID *p_tskid);	
65	loc_cpu	ER ercd= loc_cpu ();	Lock CPU
	iloc_cpu	ER ercd= iloc_cpu ();	
66	unl_cpu	ER ercd= unl_cpu ();	Unlock CPU
	iunl_cpu	ER ercd= iunl_cpu ();	

No.	Service Call	C-Language API	Function
67	dis_dsp	ER ercd= dis_dsp ();	Disable dispatch
68	ena_dsp	ER ercd= ena_dsp ();	Enable dispatch
69	sns_ctx	BOOL state= sns_ctx ();	Refer to context
70	sns_loc	BOOL state= sns_loc ();	Refer to CPU-locked state
71	sns_dsp	BOOL state= sns_dsp ();	Refer to dispatch-disabled state
72	sns_dpn	BOOL state= sns_dpn ();	Refer to dispatch-pended state
73	vsta_knl	void vsta_knl ();	Start kernel
	ivsta_knl	void ivsta_knl ();	
74	vsys_dwn	void vsys_dwn (H type,H inf1,B inf2,B inf3,H inf4, UW inf5);	Terminate system
	ivsys_dwn	void ivsys_dwn (H type,H inf1,B inf2,B inf3,H inf4, UW inf5);	
75	ivbgn_int	ER ercd= ivbgn_int (UINT dintno);	Acquire start of interrupt handler as trace information
76	ivend_int	ER ercd= ivend_int (UINT dintno);	Acquire end of interrupt handler as trace information

Interrupt Management Function

77	chg_ims	ER ercd= chg_ims (IMASK imask);	Change interrupt mask
	ichg_ims	ER ercd= ichg_ims (IMASK imask);	
78	get_ims	ER ercd= get_ims (IMASK *p_imask);	Refer to interrupt mask
	iget_ims	ER ercd= iget_ims (IMASK *p_imask);	

System Configuration Management Function

79	ref_ver	ER ercd= ref_ver (T_RVER *pk_rver);	Refer to version information
	iref_ver	ER ercd= iref_ver (T_RVER *pk_rver);	

11.2 Service Call Error Codes

Table 11.1 Service Call Error Code List

Error Code (Mnemonic)	Error Code		Error Check Type *	Error Contents
E_OK	H'0000	(D'0)	[k]	Normal termination
E_NOSPT	H'fff7	(–D'9)	[p]	Unsupported function (function is undefined)
E_PAR	H'ffef	(–D'17)	[p]/[k]	Parameter error
E_ID	H'ffee	(–D'18)	[p]	Invalid ID number
E_CTX	H'ffe7	(–D'25)	[p]/[k]	Context error
E_ILUSE	H'ffe4	(–D'28)	[k]	Illegal use of service call
E_OBJ	H'ffd7	(–D'41)	[k]	Object state is invalid
E_NOEXS	H'ffd6	(–D'42)	[p]	Object does not exist
E_QOVR	H'ffd5	(–D'43)	[k]	Queuing overflow
E_RLWAI	H'ffcf	(–D'49)	[k]	WAITING state is forcibly cancelled
E_TMOUT	H'ffce	(–D'50)	[k]	Polling failed or timeout

Note: [p] is an error that is checked when the parameter check function is selected. [k] is an error that is always checked.

Section 12 Appendix

12.1 Interrupt Source Settings

The HI1000/4 provides a file for defining the interrupt sources in the H8SX/1650.

1650_intdef.h is the name of the interrupt source definition file. By using this file, each interrupt source in the H8SX/1650 can be enabled or disabled.

Include 1650_intdef.h to make interrupt source settings.

Table 12.1 Setting Interrupt Sources

Interrupt Source Setting	Description
dis_int	Disables the interrupt source.
ena_int	Enables the interrupt source.

12.1.1 Disabling Interrupt Source (dis_int)

C-Language API:

```
   dis_int(<interrupt_id>);
```

Parameters:

```
   <interrupt_id>             Interrupt source name or value
```

Function:

Disables the interrupt enable bit corresponding to the interrupt source specified by <interrupt_id>. For the interrupt source names (<interrupt_id>), refer to Table 12.2.

For the interrupt source name (<interrupt_id>), a variable or other #define statement definition must not be specified.

12.1.2 Enabling Interrupt Source (ena_int)

C-Language API:

```
ena_int(<interrupt_id>);
```

Parameters:

```
<interrupt_id>              Interrupt source name or value
```

Function:

Enables the interrupt enable bit corresponding to the interrupt source specified by <interrupt_id>. For the interrupt source names (<interrupt_id>), refer to Table 12.2.

For the interrupt source name (<interrupt_id>), a variable or other #define statement definition must not be specified.

Table 12.2 List of Interrupt Source Names and Values

Interrupt Source Name	Value	Interrupt Source Name	Value
IRQ0_ID	0	TPU4_TCIEV_ID	35
IRQ1_ID	1	TPU4_TCIEU_ID	36
IRQ2_ID	2	TPU5_TGIEA_ID	37
IRQ3_ID	3	TPU5_TGIEB_ID	38
IRQ4_ID	4	TPU5_TCIEV_ID	39
IRQ5_ID	5	TPU5_TCIEU_ID	40
IRQ6_ID	6	TMR0_CMIEA_ID	41
IRQ7_ID	7	TMR0_CMIEB_ID	42
IRQ8_ID	8	TMR0_OVIE_ID	43
IRQ9_ID	9	TMR1_CMIEA_ID	44
IRQ10_ID	10	TMR1_CMIEB_ID	45
IRQ11_ID	11	TMR1_OVIE_ID	46
SWDTIE_ID*	12	TMR2_CMIEA_ID	47
WDT_ID	13	TMR2_CMIEB_ID	48
ADIE_ID	14	TMR2_OVIE_ID	49
TPU0_TGIEA_ID	15	TMR3_CMIEA_ID	50
TPU0_TGIEB_ID	16	TMR3_CMIEB_ID	51
TPU0_TGIEC_ID	17	TMR3_OVIE_ID	52
TPU0_TGIED_ID	18	SCI0_TEIE_ID	53
TPU0_TCIEV_ID	19	SCI0_MPIE_ID	54
TPU1_TGIEA_ID	20	SCI0_RIE_ID	55
TPU1_TGIEB_ID	21	SCI0_TIE_ID	56
TPU1_TCIEV_ID	22	SCI1_TEIE_ID	57
TPU1_TCIEU_ID	23	SCI1_MPIE_ID	58
TPU2_TGIEA_ID	24	SCI1_RIE_ID	59
TPU2_TGIEB_ID	25	SCI1_TIE_ID	60
TPU2_TCIEV_ID	26	SCI2_TEIE_ID	61
TPU2_TCIEU_ID	27	SCI2_MPIE_ID	62
TPU3_TGIEA_ID	28	SCI2_RIE_ID	63
TPU3_TGIEB_ID	29	SCI2_TIE_ID	64
TPU3_TGIEC_ID	30	SCI4_TEIE_ID	65
TPU3_TGIED_ID	31	SCI4_MPIE_ID	66
TPU3_TCIEV_ID	32	SCI4_RIE_ID	67
TPU4_TGIEA_ID	33	SCI4_TIE_ID	68
TPU4_TGIEB_ID	34		

Note: Offered samples of the H8SX/1650 do not include SWDTIE_ID.

Renesas Microcomputer Development Environment System User's Manual HI1000/4 V.1.04 (H8SX, H8S Family Realtime OS)

Publication Date:	Rev.1.00, March 18, 2004 Rev.2.00, June 28, 2006
Published by:	Sales Strategic Planning Div. Renesas Technology Corp.
Edited by:	Customer Support Department Global Strategic Communication Div. Renesas Solutions Corp.

Renesas Technology Corp. Sales Strategic Planning Div. Nippon Bldg., 2-6-2, Ohte-machi, Chiyoda-ku, Tokyo 100-0004, Japan

RENESAS SALES OFFICES

http://www.renesas.com

Refer to "**http://www.renesas.com/en/network**" for the latest and detailed information.

Renesas Technology America, Inc.
450 Holger Way, San Jose, CA 95134-1368, U.S.A
Tel: <1> (408) 382-7500, Fax: <1> (408) 382-7501

Renesas Technology Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K.
Tel: <44> (1628) 585-100, Fax: <44> (1628) 585-900

Renesas Technology (Shanghai) Co., Ltd.
Unit 204, 205, AZIACenter, No.1233 Lujiazui Ring Rd, Pudong District, Shanghai, China 200120
Tel: <86> (21) 5877-1818, Fax: <86> (21) 6887-7898

Renesas Technology Hong Kong Ltd.
7th Floor, North Tower, World Finance Centre, Harbour City, 1 Canton Road, Tsimshatsui, Kowloon, Hong Kong
Tel: <852> 2265-6688, Fax: <852> 2730-6071

Renesas Technology Taiwan Co., Ltd.
10th Floor, No.99, Fushing North Road, Taipei, Taiwan
Tel: <886> (2) 2715-2888, Fax: <886> (2) 2713-2999

Renesas Technology Singapore Pte. Ltd.
1 Harbour Front Avenue, #06-10, Keppel Bay Tower, Singapore 098632
Tel: <65> 6213-0200, Fax: <65> 6278-8001

Renesas Technology Korea Co., Ltd.
Kukje Center Bldg. 18th Fl., 191, 2-ka, Hangang-ro, Yongsan-ku, Seoul 140-702, Korea
Tel: <82> (2) 796-3115, Fax: <82> (2) 796-2145

Renesas Technology Malaysia Sdn. Bhd
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No.18, Jalan Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: <603> 7955-9390, Fax: <603> 7955-9510

HI1000/4 V.1.04
User's Manual

Renesas Electronics Corporation
1753, Shimonumabe, Nakahara-ku, Kawasaki-shi, Kanagawa 211-8668 Japan