To our customers,

Old Company Name in Catalogs and Other Documents

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

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

April 1st, 2010
Renesas Electronics Corporation

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

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

RENESAS

Notice

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

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

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

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

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

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

User's Manual

RENESAS

HI7200/MP V.1.00

User's Manual

Renesas Microcomputer
Development Environment
System

Renesas Electronics
www.renesas.com

Rev.1.01 2007.09

Notes regarding these materials

1. This document is provided for reference purposes only so that Renesas customers may select the appropriate Renesas products for their use. Renesas neither makes warranties or representations with respect to the accuracy or completeness of the information contained in this document nor grants any license to any intellectual property rights or any other rights of Renesas or any third party with respect to the information in this document.
2. Renesas shall have no liability for damages or infringement of any intellectual property or other rights arising out of the use of any information in this document, including, but not limited to, product data, diagrams, charts, programs, algorithms, and application circuit examples.
3. You should not use the products or the technology described in this document for the purpose of military applications such as the development of weapons of mass destruction or for the purpose of any other military use. When exporting the products or technology described herein, you should follow the applicable export control laws and regulations, and procedures required by such laws and regulations.
4. All information included in this document such as product data, diagrams, charts, programs, algorithms, and application circuit examples, is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas products listed in this document, please confirm the latest product information with a Renesas sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas such as that disclosed through our website. (http://www.renesas.com)
5. Renesas has used reasonable care in compiling the information included in this document, but Renesas assumes no liability whatsoever for any damages incurred as a result of errors or omissions in the information included in this document.
6. When using or otherwise relying on the information in this document, you should evaluate the information in light of the total system before deciding about the applicability of such information to the intended application. Renesas makes no representations, warranties or guaranties regarding the suitability of its products for any particular application and specifically disclaims any liability arising out of the application and use of the information in this document or Renesas products.
7. With the exception of products specified by Renesas as suitable for automobile applications, Renesas products are not designed, manufactured or tested for applications or otherwise in systems the failure or malfunction of which may cause a direct threat to human life or create a risk of human injury or which require especially high quality and reliability such as safety systems, or equipment or systems for transportation and traffic, healthcare, combustion control, aerospace and aeronautics, nuclear power, or undersea communication transmission. If you are considering the use of our products for such purposes, please contact a Renesas sales office beforehand. Renesas shall have no liability for damages arising out of the uses set forth above.
8. Notwithstanding the preceding paragraph, you should not use Renesas products for the purposes listed below:
   (1) artificial life support devices or systems
   (2) surgical implantations
   (3) healthcare intervention (e.g., excision, administration of medication, etc.)
   (4) any other purposes that pose a direct threat to human life

   Renesas shall have no liability for damages arising out of the uses set forth in the above and purchasers who elect to use Renesas products in any of the foregoing applications shall indemnify and hold harmless Renesas Technology Corp., its affiliated companies and their officers, directors, and employees against any and all damages arising out of such applications.
9. You should use the products described herein within the range specified by Renesas, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas shall have no liability for malfunctions or damages arising out of the use of Renesas products beyond such specified ranges.
10. Although Renesas endeavors to improve the quality and reliability of its products, IC products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Please be sure to implement safety measures to guard against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other applicable measures. Among others, since the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
11. In case Renesas products listed in this document are detached from the products to which the Renesas products are attached or affixed, the risk of accident such as swallowing by infants and small children is very high. You should implement safety measures so that Renesas products may not be easily detached from your products. Renesas shall have no liability for damages arising out of such detachment.
12. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written approval from Renesas.
13. Please contact a Renesas sales office if you have any questions regarding the information contained in this document, Renesas semiconductor products, or if you have any other inquiries.

Preface

This manual describes how to use the HI7200/MP for the SH2A-DUAL microcomputer. Before using the HI7200/MP, please read this manual to fully understand the operating system.

Notes on Descriptions

Prefix	Prefixes H' and 0x indicate hexadecimal numbers. The prefix D' indicates a decimal number. Numbers with no prefix are decimal.
\	'\' is the directory delimiter.
cfg file	Kernel configuration file
xx.xx (e.g. system.stack_size)	A character string delimited by periods indicates one of the following: (a) The setting of an item in the cfg file (b) A member of a structure (c) A bit in a register
[Menu -> Menu item]	-> leads to the menu item (e.g. File -> Save).
<RTOS_INST>	System directory where header files and the configurator are installed. The system directory is **x.yy.zzww** under the directory specified by the user at the time of installation. The product version is indicated by **x.yy.zz** and **ww** is an internal identification number with a value from 00 to 99. Examples - Product version is V.1.01 Release 02: 1.01.02ww - Product version is V.2.11 Release 13: 2.11.13ww
<SAMPLE_INST>	Directory where sample files for the HI7200/MP are stored. The user can specify this directory while setting up the HI7200/MP.
$(xxxx)	Custom placeholder in the HEW. For example, $(RTOS_INST) and $(SAMPLE_INST) are custom placeholders that represent <RTOS_INST> and <SAMPLE_INST>, respectively.

Trademarks

1. μITRON is an acronym of the "Micro Industrial TRON" and TRON is an acronym of "The Real Time Operating system Nucleus".

   TRON, ITRON, and μITRON are the names of computer specifications and do not indicate a specific group of the commodity or the commodity.

   The μITRON4.0 specification is an open realtime-kernel specification defined by TRON Association. The document of the μITRON4.0 specification can be downloaded from the TRON Association homepage (http://www.assoc.tron.org).

   The copyright of the μITRON specification belongs to TRON Association.
2. Microsoft® Windows® 2000 and Microsoft® Windows® XP operating systems are registered trademarks of Microsoft Corporation in the United States and/or other countries.
3. SuperH™ is a trademark of Renesas Technology Corp..
4. All other product names are trademarks or registered trademarks of the respective holders.

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, Installation: Installation of the HI7200/MP

Section 3, Overview: Overview of the HI7200/MP

Section 4, Introduction to the Kernel: Basic information required for use of the HI7200/MP and key items regarding the kernel at the core of the HI7200/MP

Section 5, Kernel Functions: All functions of the kernel

Section 6, Kernel Service Calls: Specifications of the kernel service calls

Section 7, RPC Library: Specifications of the RPC library

Section 8, OAL: Specifications of OAL

Section 9, Spinlock Library: Specifications of the spinlock library

Section 10, IPI: Specifications of IPI functions

Section 11, SH2A-DUAL Cache-Support Library: Specifications of the SH2A-DUAL cache-support library

Section 12, Application Program Creation: Methods for writing a task or a handler

Section 13, Generating Load Modules: Procedures for generating load modules

Section 14, Configurator (cfg72mp): Specifications of cfg72mp

Section 15, GUI Configurator: Introduction to the GUI configurator. For usage of the GUI configurator, refer to the online help.

Section 16, Sample Programs: Descriptions of the provided sample programs

Section 17, Build: Methods to generate a load module by compilation

Section 18, Calculating Stack Size: Methods for calculating the sizes of stacks for use by tasks or handlers

Section 19, types.h: Description of types.h, in which basic data types are defined

Section 20, Notes on the FPU: Notes on using the FPU. Read this section before using a CPU that includes an FPU, whether or not you will actually use the FPU functions.

Section 2 Installation

2.1 Method of Installation

For the method of installation, read the release notes that come with the product.

2.2 Directory Structure

The HI7200/MP is installed in two directories: a system directory and sample directory. These two directories can be located under different directories.

2.2.1 System Directory (<RTOS_INST>)

Header files and the configurator are installed under the system directory. The system directory is referred to as <RTOS_INST> in this manual. The system directory is **x.yy.zzww** under the directory specified by the user at installation. **x.yy.zz** stands for the product version and **ww** is an internal identification number with a value from 00 to 99. Examples are shown below.

- Product version is V.1.01 Release 02: 1.01.02ww
- Product version is V.2.11 Release 13: 2.11.13ww

The structure of the directories under the system directory is as follows:

cfg72mp\	cfg72mp (command line configurator)
gui_config\	GUI configurator
manuals\	Manual
os\	
include\	Common header files (types.h, itron.h, etc.)
kernel\	Source code for kernel (only HI7200/MP with a source code license)
lib\	Library
rpc\	Source code for RPC library (only HI7200/MP with a source code license)
sh2adual_cache\	Source code for SH2A-DUAL cache support library
spinlock\	Source code for spinlock library
system\	System definition files

2.2.2 Sample Directory (<SAMPLE_INST>)

The sample directory, as its name indicates, is a directory in which the sample programs are installed. The sample directory is referred to as <SAMPLE_INST> in this manual. The location of the sample directory should be specified by the user at installation.

The structure under the sample directory is as follows:

R0K572650D000BR\
Sample programs for the "R0K572650D000BR" evaluation board equipped with the SH7265

Detailed descriptions of the sample programs under the R0K572650D000BR directory are given in section 16, Sample Programs.

Section 3 Overview

3.1 Overview

This product which is a realtime OS developed for the SH2A-DUAL microcomputer is designed to run on a system in which the features are distributed among the CPU cores.

A system using this OS can operate with an OS for each CPU running independently and synchronous communication between CPUs possible when required.

3.2 Features

3.2.1 Kernel

This product has a kernel based on µITRON4.0 specifications which are realtime OS specifications widely popular. Therefore, knowledge obtained from commercially available µITRON-related books or seminars can be used with little modification. Software components based on µITRON specifications can be embedded easily.

An API with conventional µITRON4.0 specifications can call a service call for the kernel of the other CPU core. Accordingly, an application program for a conventional single CPU can be easily distributed to the two CPUs. In other words, a programming model using a µITRON-specification OS for a conventional single CPU can be extended for a multicore environment.

3.2.2 RPC (Remote Procedure Call) Library

The RPC is used for calling a function in the other CPU in the same format as a normal function call. This facilitates feature distribution to each CPU on a function basis.

3.2.3 OAL

The OAL is a functional module in which the OS dependent part of the RPC has been extracted. Rewriting the OAL facilitates porting of the RPC to another OS.

3.2.4 Spinlock Library

Exclusive control is required when a shared resource is used by multiple CPUs. The spinlock library is prepared as a primitive for exclusive control between the CPUs.

The spinlock library is also used by the kernel and RPC.

3.2.5 IPI Function

The IPI function which is a primitive that performs communication between the CPUs provides receive ports using inter-CPU interrupts.

The IPI function is also used by the kernel and RPC.

3.2.6 Cache Support Library

The application can use the cache support library for the purpose of maintaining the coherency between the local cache of each CPU and the actual memory.

3.2.7 Sample Programs

Sample programs for understanding the OS functions and High-performance Embedded Workshop workspaces as the build environments are provided.

3.2.8 Configurator

The kernel configurator (cfg72mp) is prepared to facilitate configuration of a kernel suitable for the system. The user should create a kernel configuration file (cfg file) in the defined format.

A GUI configurator that can be operated through the GUI screen is also prepared for beginners as a tool to configure a cfg file.

3.2.9 Debugging Extension (Option)

The debugging extension which adds a multitasking debugging function to High-performance Embedded Workshop V.4 or later versions is prepared. The debugging extension supports the following functions.

- Refer to the status of objects, such as a task
- Perform operation to objects, such as starting a task or setting the task event flag
- Display the service call history

The debugging extension can be downloaded free of charge from our homepage. (However, the debugging extension to support this OS was still being developed at the time this manual was created.)

3.3 Multicore

This OS is on both of the CPUs and each OS operates individually. The application designer should statically determine the operations to be performed by each CPU (feature distribution) and implement the features on the respective OS.

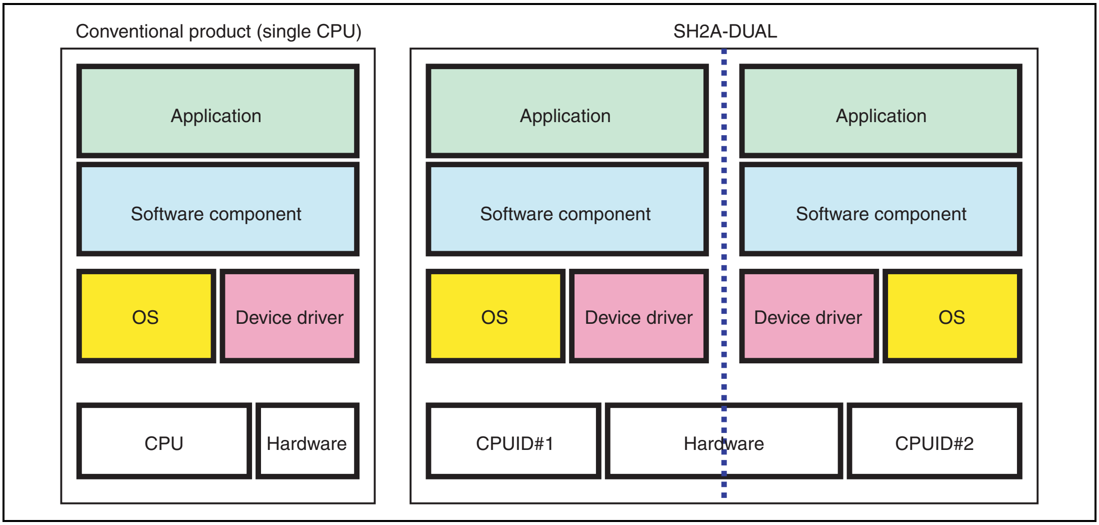


Figure 3.1 Software Structure

Which CPU is to control each hardware resource is also statically determined based on the feature distribution.

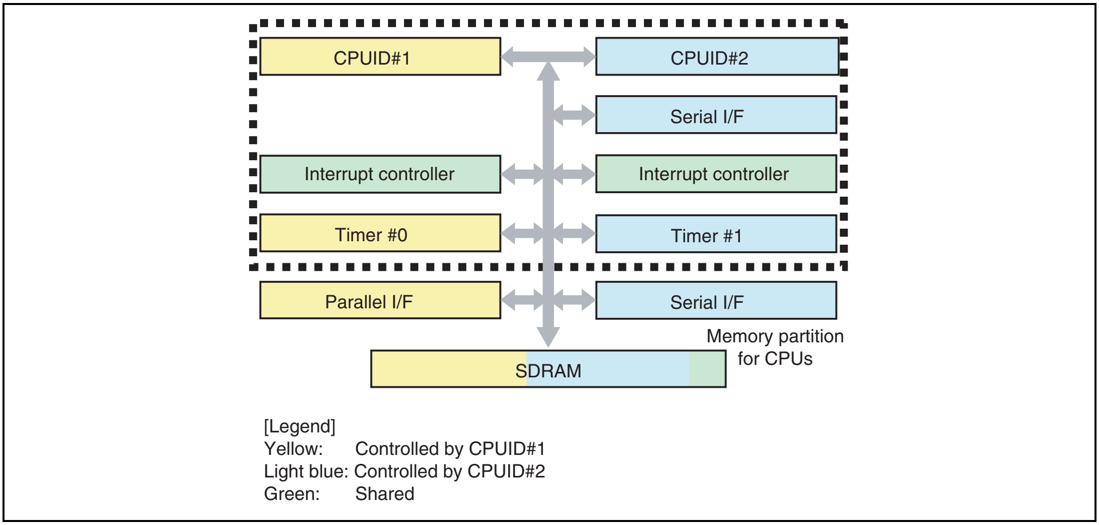


Figure 3.2 Hardware Resource Division (Image)

The memory space of the SH2A-DUAL is basically one plane that is shared by both CPUs. Memory is statically divided into ranges that can be used by each CPU. Naturally, memory shared by both CPUs can be used for communication between the CPUs.

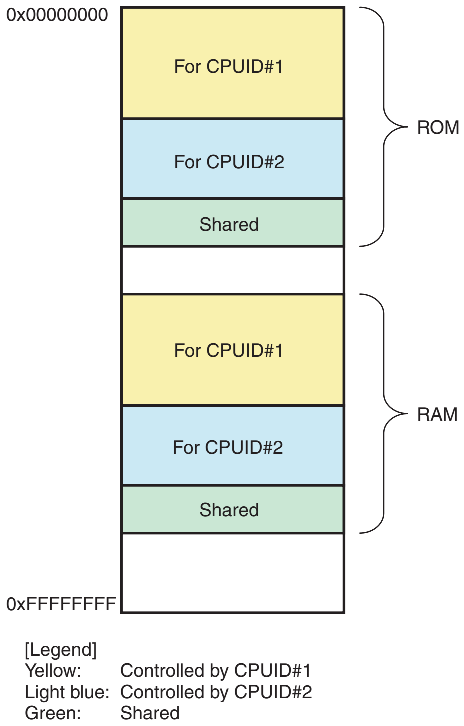


Figure 3.3 Memory Map Division (Image)

The basic methods for distributing the features to each CPU is a method of dividing the features on a function basis using the RPC library (figure 3.4) and a method of dividing the features on a task basis using remote service calls (figure 3.5).

When the RPC library is used, a function in another CPU can be called.

When a remote service call is used, access to a task or kernel object (e.g. semaphore) in the other CPU is possible with the same service call as in a conventional product.

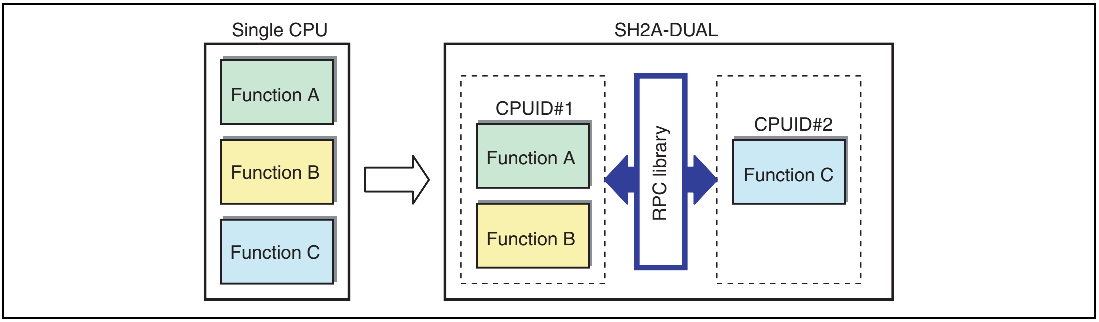


Figure 3.4 Feature Distribution Using RPC (Image)

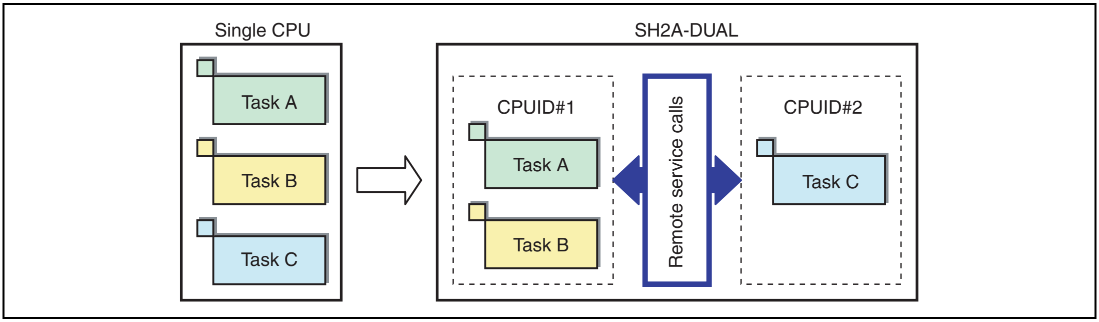


Figure 3.5 Feature Distribution Using Remote Service Calls (Image)

3.4 Operating Environment

The operating environment is shown in table 3.1.

Table 3.1 Operating Environment

Item	Operating Environment
Target CPU core	SH2A-DUAL
Host machine	IBM-PC/AT compatible machine operated under Windows® 2000 or Windows® XP
Compiler	Renesas C/C++ Compiler Package for SuperH™ RISC engine V.9.00 Release 04A or later
High-performance Embedded Workshop*	V.4.02.00 or later

*Note: When the provided High-performance Embedded Workshop workspace is used

Section 4 Introduction to the Kernel

4.1 Principles of Kernel Operation

In the HI7200/MP, an independent kernel operates for each CPU of an SH2A-DUAL microcomputer.

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

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

As is shown in figure 4.1, the kernel switches operation between various tasks as required.

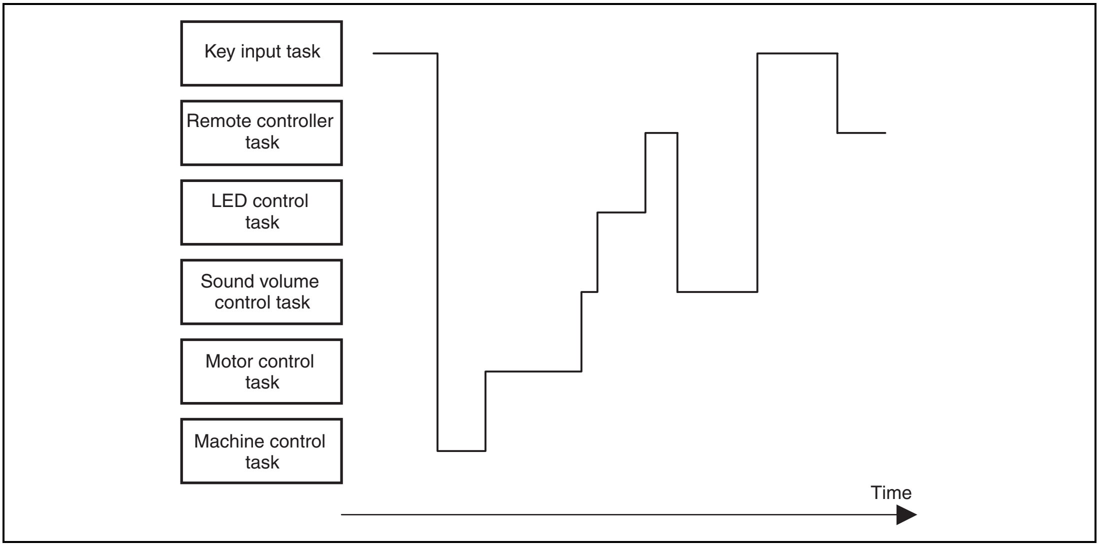


Figure 4.1 Operation of Multiple Tasks

This switching between tasks is called task dispatch.

The kernel 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 predetermined intervals as in a time-sharing system. This type of scheduling is generally called event-driven.

After a task is dispatched, execution of the task resumes from the point at which it was previously suspended (figure 4.2).

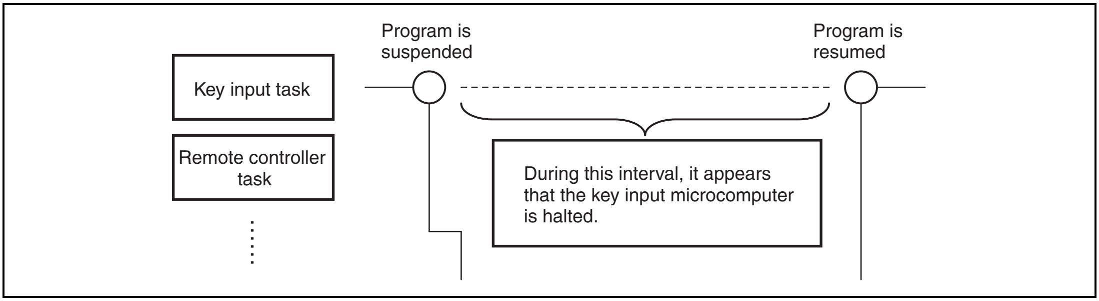

Program is suspended
Program is resumed
Key input task
Remote controller task
During this interval, it appears that the key input microcomputer is halted.

Figure 4.2 Suspending and Resuming a Task

In figure 4.2, when the control of execution is passed to another task from the key input task, execution of the program for the key input task appears to the programmer to have halted; that is, the key input microcomputer appears to have halted.

By restoring the contents of CPU registers that were stored when a task was suspended, the kernel resumes the execution of a task from the state in which it was suspended. In other words, dispatching a task means saving the contents of the CPU registers for the task currently being executed in a memory area prepared for the management of that task, and restoring the contents of the CPU registers for the task for which execution is being resumed (figure 4.3).

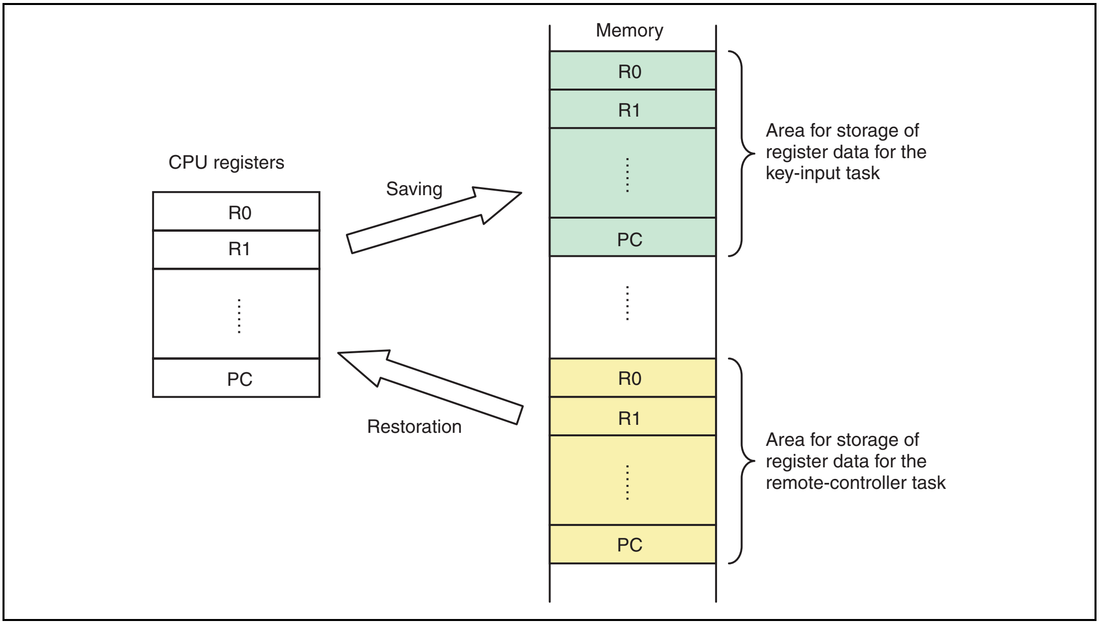

Memory
CPU registers
R0
R1
PC
Saving
Restoration
R0
R1
PC
Area for storage of register data for the key-input task
R0
R1
PC
Area for storage of register data for the remote-controller task

Figure 4.3 Task Dispatch

As well as the CPU registers, task execution requires stack areas. Separate stack space must be allocated for each task.

4.2 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 can be sent to the kernel.

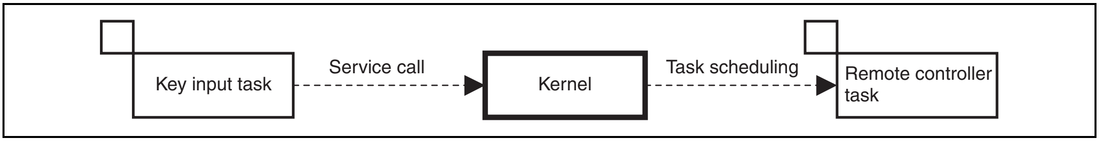


Figure 4.4 Service Call

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

```
act_tsk(ID_MAINTASK);
```

There are two types of service call: local service calls, which send requests only to the kernel for the current CPU, and remote service calls, which send requests to the kernel for the other CPU.

4.3 CPU ID

A CPU ID is a number that identifies a CPU core. Two purposes for which CPU IDs are useful are given below.

- Specifying the CPU to which the target object of a service call belongs
- Specifying the target CPU to which data is transmitted by using the IPI (see section 10, IPI)

In the HI7200/MP, the CPU IDs of CPU#0 and CPU#1 as defined in the specifications for the SH2A-DUAL microcomputer are 1 and 2, respectively. Note that counting for CPU IDs starts with 1, while counting for the CPU names defined in the specifications of the SH2A-DUAL microcomputer starts with 0.

4.4 Objects

4.4.1 Outline

The processing objectives of service calls, such as tasks or semaphores, are called objects. Objects are distinguished by their IDs. In service calls such as those for activating tasks and setting event flags, the IDs of target objects should be specified as parameters. In some service calls, it is also possible to handle objects of the kernel for the other CPU by specifying the corresponding object IDs.

4.4.2 ID Numbers

An ID number is represented as a 16-bit signed integer. The 16 bits of an ID number have the following meanings.

- Bit 15 (MSB): Sign of the local object ID
- Bits 14 to 12: CPU ID
- Bits 11 to 0: Local object ID

(1) CPU ID

To issue a remote service call, bits 14 to 12 of the target object ID should contain the corresponding CPU ID. Specifying VCPU_SELF as the CPU ID selects the same CPU as that for the caller of the service call. VCPU_SELF is a macro defined as 0 in kernel.h and is not based on the μITRON4.0 specification.

(2) Local Object ID

Positive local object IDs are assigned to objects. The maximum value for the local object IDs of objects should be defined as maxdefine in the cfg file. Although no object has a negative local object ID, some service calls allow the specification of negative local object IDs (only for special purposes).

4.4.3 Using ID Names to Specify Objects

To differentiate between objects, the kernel internally activates the objects by using their ID numbers. Specifically, control of the form "Start the task having the task ID number 1" might be applied. However, directly using literal task numbers in programs will give the programs very poor readability. If, for instance, the following statement is entered in a program, the programmer must always know which task has the local object ID number 1.

```
act_tsk(1); /* Start the task having local object ID number 1 in the
current CPU */
```

Moreover, anyone else viewing the program will not be able to see at a glance which task is No. 1. To avoid such inconvenience, the HI7200/MP provides means of specifying tasks by name (ID name). "configurator cfg72mp" automatically converts task ID names to task ID numbers. To be more specific, the configurator outputs a header file (e.g. kernel_id.h) that includes definitions of the following type, associating task ID names with task ID numbers.

```
#define ID_MAINTASK MAKE_ID(1, 1)
```
[1]

Figure 4.5 is a schematic view of the task identification system.

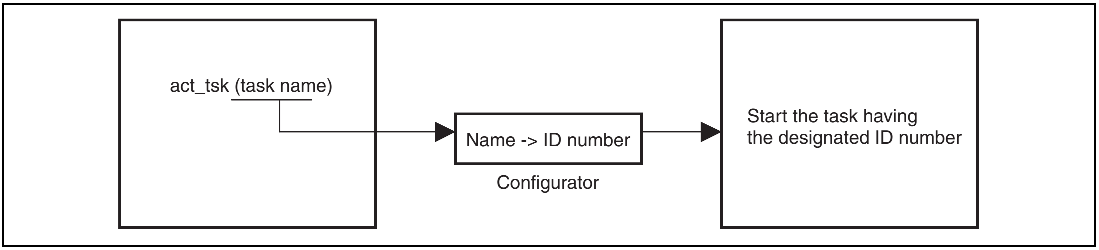


Figure 4.5 Task Identification

With this task identification system, our earlier example is now as follows.

```
act_tsk(ID_MAINTASK); /* Start the task having the ID name
"ID_MAINTASK" */
```

[1] MAKE_ID() is a macro for creating IDs consisting of the CPU ID and the local object ID. For details, refer to section 6.31.4, Function Macros Defined in kernel.h.

This call specifies invocation of the task corresponding to "ID_MAINTASK". Also note that the compiler's pre-processor converts task names to ID numbers in the generation of an executable program. Therefore, this feature does not reduce processing speeds.

Although the example on the previous page just referred to task identification, other objects that have ID numbers can also be given ID names.

4.5 Tasks

4.5.1 Task State

The kernel checks the task state to control whether to execute a task. For example, figure 4.6 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.

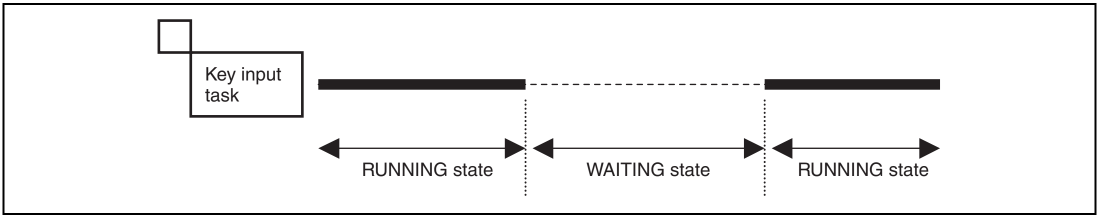


Figure 4.6 Task States

The kernel controls transitions between seven states, including the RUNNING and WAITING states, as shown in figure 4.7. A task makes the transitions between these seven states.

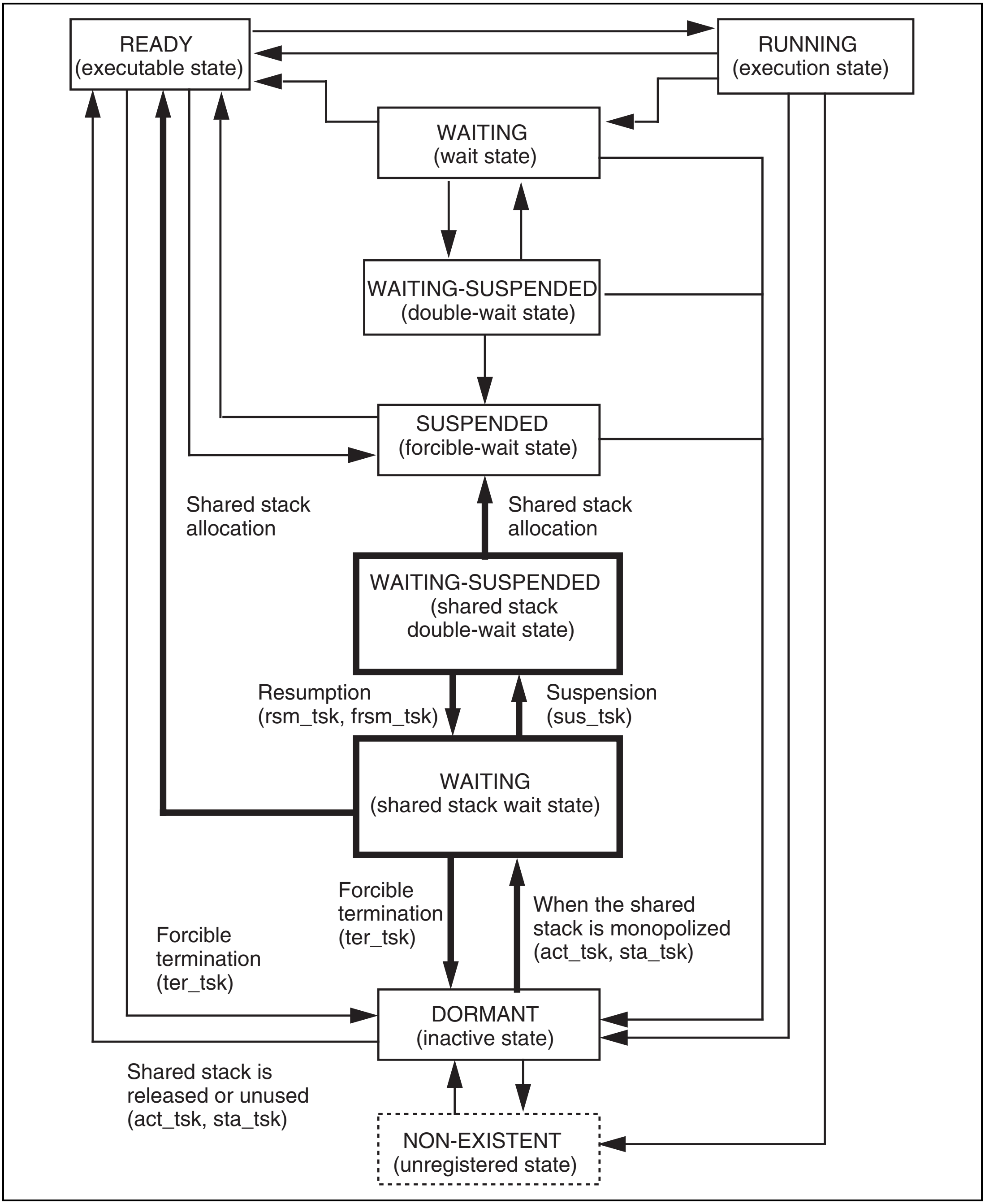


Figure 4.7 Task State Transition Diagram

(1) NON-EXISTENT State

The task has not been registered in the kernel. This is a virtual state.

(2) DORMANT State

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

(3) READY (executable) State

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

(4) RUNNING State

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

(5) WAITING State

When the task issues a service call such as tslp_tsk and the specified conditions are not satisfied, the task enters the WAITING state. A task is released from the WAITING state by the service call (such as wup_tsk) that corresponds to the call which initiated the WAITING state, after which the task enters the READY state.

(6) SUSPENDED State

A task has been suspended by another task through sus_tsk.

(7) WAITING-SUSPENDED State

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

4.5.2 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 range of available priorities is 1 to system.priority as defined in the cfg file.

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

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

Figure 4.8 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 non-empty ready queue for the highest priority and puts it in the RUNNING state.

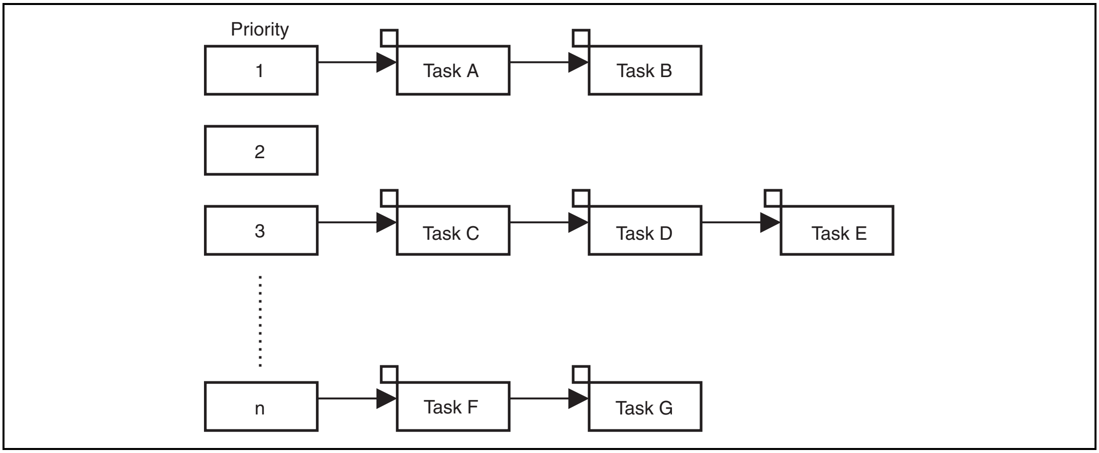


Figure 4.8 Ready Queues (Waiting for Execution)

4.5.3 Task Waiting Queues

A service call can make a task wait (enter the WAITING state) until a condition designated in terms of objects (such as semaphores and event flags) has been satisfied. For some types of objects, two or more tasks may be in the WAITING state. Attributes that select the order in which waiting tasks are handled are specifiable when the objects are created. The specifiable attributes are TA_TFIFO (handling on an FIFO basis) or TA_TPRI (handling on a priority basis). Tasks leave the WAITING state in the order specified for the waiting queue. Figures 4.9 and 4.10 show the order of task handling for objects with the respective attributes, where task D (priority: 9), task C (priority: 6), task A (priority: 1), and task B (priority: 5) have joined the waiting queue, in that order.

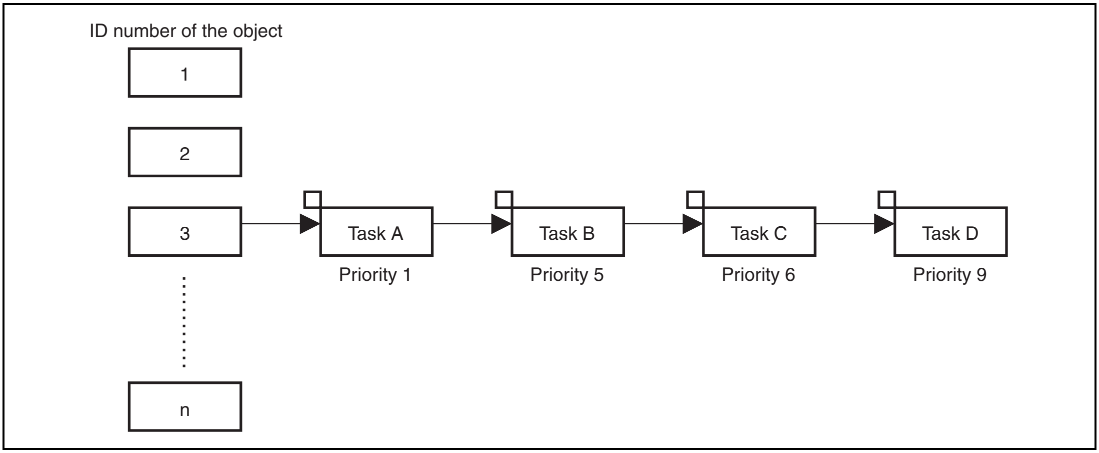


Figure 4.9 Waiting Queue with the Attribute TA_TPRI

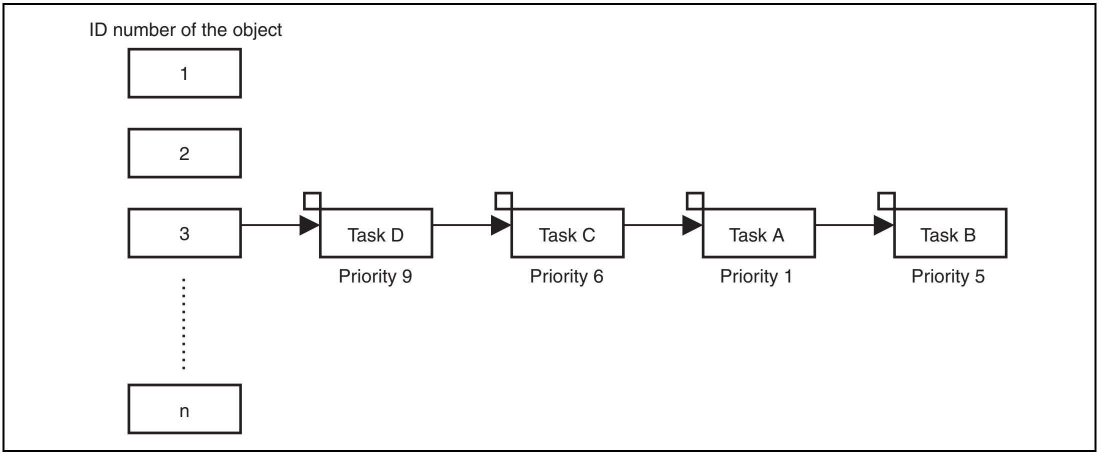


Figure 4.10 Waiting Queue with the Attribute TA_TFIFO

4.5.4 Task Stack

Each task needs a stack area. For the kernel, there are basically two types of stack: static stack and non-static stack. Tasks having local task ID numbers less than or equal to maxdefine.max_statictask in the cfg file use static stacks, and other tasks use non-static stacks. A single static stack can also be shared by multiple tasks (shared stack function).

(1) Static Stack

Multiple static stack areas can be defined by static_stack[] statements in the cfg file. For static_stack[], the size of the stack, a section name to be given to the stack, and the local IDs of tasks that are to use the stack should be specified. When two or more local task IDs are specified, the stack is shared by these tasks.

(2) Non-Static Stack

The types of non-static stack area are listed below.

(a) Use the default task-stack area

In this case, simply specify the size of the stack when creating a task by making a cre_tsk or acre_tsk service call or by a task[] statement in the cfg file. The kernel allocates the specified size of a stack area from the default task-stack area. There is only one default task-stack area, which is managed internally by the kernel just like a variable-sized memory pool. The section name for the default task-stack area is BC_hitskstk.

(b) Use the stack area allocated by the application

In this case, the application allocates the stack area. After that, specify the address and size of the stack when creating a task by making a cre_tsk or acre_tsk service call or by a task[] statement in the cfg file.

(c) Create a stack area according to the cfg file

Specify the size of the stack and a section name to be given to the stack when generating a task by a task[] statement in the cfg file.

Table 4.1 shows the differences between static and non-static stacks.

Table 4.1 Differences between Static and Non-Static Stacks

			Static Stack	**Non-Static Stack**
Local task ID numbers			1 to maxdefine.max_statictask	maxdefine.max_statictask + 1 or more
Create tasks by	cfg file		task[]	task[]
		ID number	Cannot be omitted (automatic allocation not available)	Can be omitted (automatic allocation available)
		Other	Definition is required in static_stack[]	-
	Service calls		vscr_tsk, ivscr_tsk	cre_tsk, icre_tsk, acre_tsk, iacre_tsk
Sharing of stack by tasks			Available	Not available

4.5.5 Shared Stack Function

More than one task can share a single static stack. The shared stack function is not defined in the µITRON4.0 specification.

To have two or more tasks share one static stack, type the local IDs of these tasks as static_stack[].tskid in the cfg file.

Only one task in a task group that shares a static stack can be executed at a time. When multiple tasks are initiated and share a stack, the task that was initiated first uses the stack first. The remaining tasks enter the shared-stack waiting state. Tasks in the shared-stack waiting state are managed as a first-in first-out (FIFO) queue, regardless of their priority. Tasks join the shared-stack waiting queue in the order in which they were initiated.

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

Figure 4.11 shows the task-state transitions for the shared stack function.

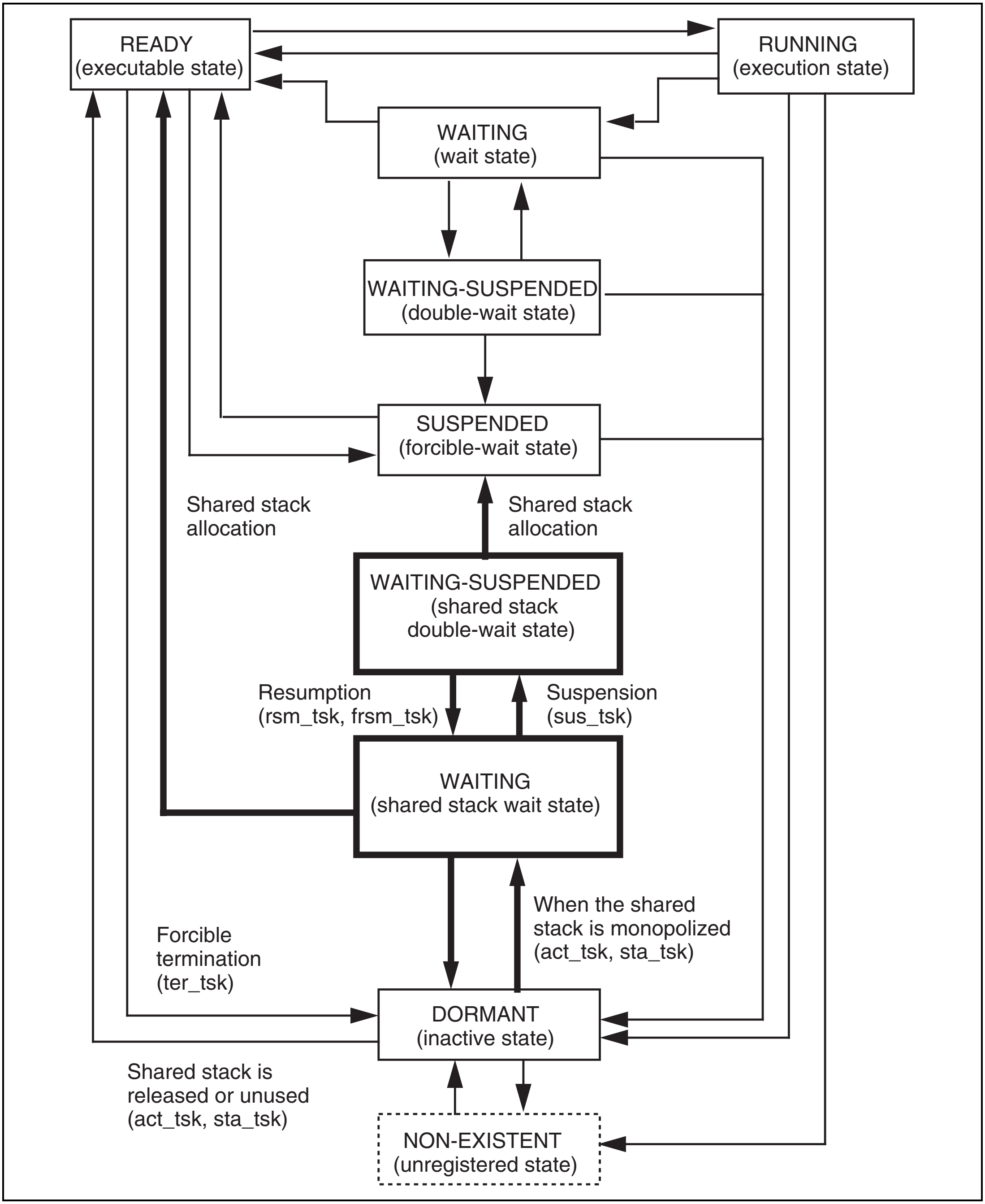


Figure 4.11 Task-State Transitions for the Shared Stack Function

4.6 System State

The system state is classified into the following orthogonal states.

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

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

4.6.1 Task Contexts and Non-Task Contexts

System is in either task contexts or non-task contexts. The difference between task contexts and non-task contexts is described in table 4.2.

Table 4.2 Task Contexts and Non-Task Contexts

Item	Task Contexts	Non-Task Contexts
Available service calls	Service calls that can be called from task contexts	Service calls that can be called from non-task contexts
Task scheduling	Refer to sections 4.6.2 and 4.6.3	Does not occur

The following forms of processing are executed in non-task contexts.

- Interrupt handlers
- Time event handlers (cyclic handlers, alarm handlers, and overrun handler)
- Portions of execution where the interrupt mask has been changed to a value other than 0 by the chg_ims service call

Note that extended service calls initiated in the above processing states are also executed in non-task contexts.

CPU exception handlers are executed in the same context as that before the exception occurred.

4.6.2 Dispatch-Disabled State/Dispatch-Enabled State

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

Issuing the dis_dsp service call changes the system state to dispatch-disabled state, while 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 is in dispatch-disabled state or not.

4.6.3 CPU-Locked State/CPU-Unlocked State

System is in either CPU-locked state or CPU-unlocked state. In CPU-locked state, interrupts and task scheduling are not allowed. Note, however, that interrupts with interrupt levels higher than that specified in the kernel interrupt mask level (system.system_IPL in the cfg file) are allowed. Any service calls that make tasks enter the WAITING state cannot be issued.

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

Service calls that can be issued in the CPU-locked state are restricted to those listed in table 4.3.

Table 4.3 Service Calls that can be Issued in the CPU-Locked State

loc_cpu, iloc_cpu	unl_cpu, iunl_cpu	sns_ctx	sns_loc	sns_dsp	sns_dpn
vsta_knl, ivsta_knl	vsys_dwn, ivsys_dwn	sns_tex	vsns_tmr	ext_tsk *	exd_tsk *

Note: These calls will release the system from the CPU-locked state.

4.6.4 Dispatch-Pending State

The dispatch-pending state means that processing with a higher priority than the dispatcher is in progress so that no other task can be executed. To be more specific, each of the following cases corresponds to the dispatch-pending state.

- Non-task context
- Dispatch-disabled state
- CPU-locked state
- Interrupt mask level (value indicated by the IMASK bits in SR) of the CPU is not 0

The sns_dpn service call can be used to check if the system is in the dispatch-pending state.

4.7 Processing Units and Precedence

An application program is executed in the following processing units.

Task: A task is a unit controlled by multitasking.

Task Exception Handling Routine: A task exception handling routine is executed when task exception handling is requested by a task in the ras_tex service call.

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, Alarm Handler, and Overrun Handler):
A time event handler is executed when a specified cycle or time has been reached.

Extended Service Call: An extended service call is used to call a module that is not linked.
When this extended service call is issued, the corresponding extended service call routine is called.

The various processing units are processed in the following order of precedence.

(1) Interrupt handlers, time event handlers and CPU exception handlers
(2) Dispatcher (part of kernel processing)
(3) Tasks

The dispatcher is kernel processing that switches the task being executed. Since interrupt handlers, time event handlers, and CPU exception handlers have higher precedence than the dispatcher, no tasks are executed while these handlers are running.

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

The precedence of a time event handler is the same as the timer interrupt level (clock.IPL).

The precedence of a CPU exception handler is higher than that of the processing where the CPU exception occurred and of the dispatcher. The precedence of a CPU exception handler is also lower than that of other processing that has higher precedence than the processing where the CPU exception occurred.

The order of precedence for tasks depends on the priority of the tasks.

The precedence of an extended service call routine is higher than that of the processing where the extended service call was called. The precedence of an extended service call routine is also lower than that of other processing that has higher precedence than the processing where the extended service call was called.

The precedence of a task's exception processing routine is higher than that of the task and lower than that of other higher-level tasks.

When the following service calls are made, a level of precedence other than those described above can be temporarily generated:

(a) When dis_dsp is called, the precedence will be between (1) and (2) above. The state is returned to the prior state by calling ena_dsp.
(b) When loc_cpu or iloc_cpu is called, the precedence will be the same as that of an interrupt handler having the same interrupt level as the kernel interrupt mask level (system.system_IPL). The state is returned to the prior state by calling unl_cpu or iunl_cpu.
(c) While the values of the IMASK bits in the SR register are changed to other than 0, the precedence is the same as for an interrupt handler at the same level.

4.8 Interrupts

An interrupt handler defined by the user is initiated in response to interrupt generation. The system goes down when no interrupt handler has been defined. Interrupt handlers are executed in non-task contexts.

4.8.1 Types of Interrupt Handler

Interrupt handlers can be divided into types in the following two ways.

- Kernel interrupt handlers and non-kernel interrupt handlers
- Direct interrupt handlers and normal interrupt handlers

(1) Kernel interrupt handlers and non-kernel interrupt handlers

- Kernel interrupt handlers

  These are interrupt handlers with an interrupt level lower than or equal to the kernel interrupt mask level (system.system_IPL). Service calls can be issued from within a kernel interrupt handler are those that can be called in non-task contexts. Note, however, that handling of kernel interrupts generated during kernel processing may be delayed until the interrupts become acceptable.

- Non-kernel interrupt handlers

  These are interrupt handlers with an interrupt level higher than the kernel interrupt mask level (system.system_IPL). Non-kernel interrupts generated during service-call processing are immediately accepted whether or not kernel processing is in progress. Note, however, that no service call can be issued from within a non-kernel interrupt handler.

(2) Direct interrupt handlers and normal interrupt handlers

A direct interrupt handler is directly initiated in response to an interrupt, while a normal interrupt handler is initiated via the kernel. Direct interrupt handlers thus incur less overhead. Also refer to the following information in selecting the types of interrupt handler you wish to use.

(a) Description format

Normal interrupt handlers are written as C-language functions. Direct interrupt handlers, on the other hand, are written as interrupt functions and require specification of the #pragma interrupt directive. Handler functions specified in this way are thus less portable than normal interrupt handlers. How #pragma interrupt should be specified depends on the conditions listed below. For details, refer to section 12.5.4, Direct Interrupt Handlers.

- Whether the interrupt handler is a kernel interrupt handler or non-kernel interrupt handler (i.e., whether the interrupt level is higher than system.system_IPL)
- Whether the interrupt uses register banks or not

(b) Interrupt level

Non-kernel interrupt handlers must be implemented as direct interrupt handlers. As stated in section 4.8.3, Restriction on Service Calls, service calls are not to be made from within these handlers.

(c) Stack

Normal interrupt handlers use the interrupt stack, which is the only specific stack area in the system. The size of the interrupt stack should be specified as system.stack_size in the cfg file. On the other hand, the stack area for use by a direct-interrupt handler should be allocated by the application. This stack area should be selected for use at the start of the direct interrupt handler and then returned to its previous state at the end of the handler (this processing is performed by specifying “sp=” in the #pragma interrupt directive). Handlers for interrupts at the same level can share the same stack area.

(d) Service calls

Since all of these interrupt handlers are executed in non-task contexts, service calls available in non-task contexts can be issued in cases other than those covered by (b).

(e) Definition

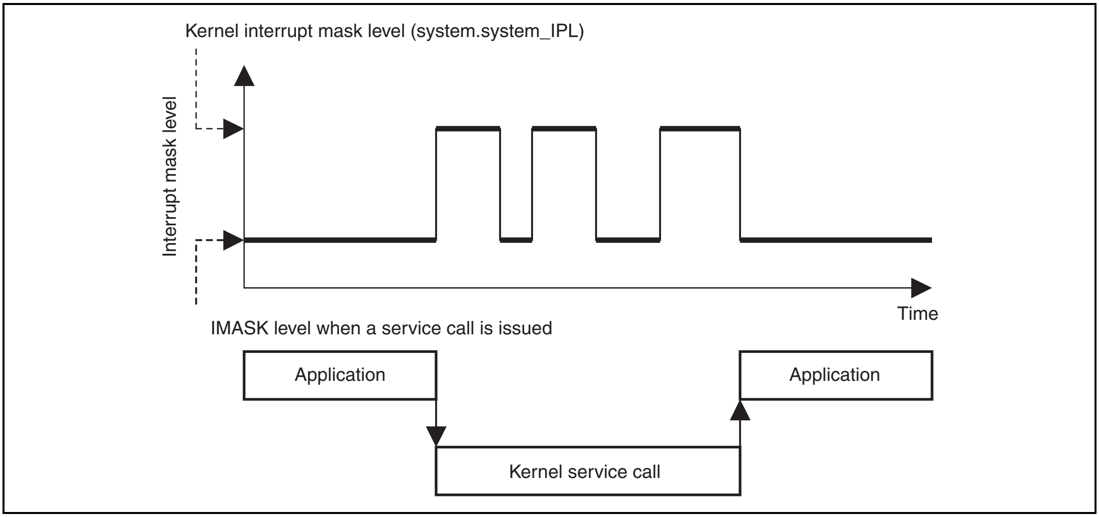
To define a direct interrupt handler, the VTA_DIRECT attribute should be specified.

4.8.2 Controlling Interrupts (by Setting IMASK Bits in the Register SR)

Specifications of the SH microcomputers allow control of the levels of interrupts accepted by the CPU by setting the IMASK bits in the register SR.

(1) Controlling the IMASK level during the period of a service call

Interrupts are enabled or disabled during the execution of a service call by setting the IMASK bits in the register SR. Since the execution of a service call should not be broken up, the IMASK level is changed to the kernel interrupt mask level (system.system_IPL) within the service call as required. Such periods are called kernel-level critical sections. In other periods, the IMASK level is the same as that before the service call. Figure 4.12 shows interrupt control during the period of a service call.



Kernel interrupt mask level (system.system_IPL)
Interrupt mask level
Time
IMASK level when a service call is issued
Application
Application
Kernel service call

Figure 4.12 Interrupt Control during the Period of a Service Call

(2) Controlling the IMASK level by the application

As in the example in the figure above, the IMASK level must be changed within service calls. The IMASK level can be changed from an application in the following three ways.

(a) Use loc_cpu or iloc_cpu

These calls change the IMASK level to the kernel interrupt mask level (system.system_IPL). Since the system enters the CPU-locked state, dispatching of tasks is postponed until the system is unlocked. While the system is in the CPU-locked state, the IMASK level should not be directly adjusted (method (c) below) to be lower than the kernel interrupt mask level (system.system_IPL). Furthermore, if the method described under (c) is used to change the IMASK level to be higher than the kernel interrupt mask level (system.system_IPL), the IMASK level must be set back to its original level before the system leaves the CPU-locked state.

(b) Use chg_ims or ichg_ims

These calls change the IMASK level to a desired value but are not available when the system is in the CPU-locked state. If the IMASK level is changed to a value other than 0 in a task context, the system is assumed to be in a non-task context. Note that the available service calls and the size of the stack for use by the service calls are different in a non-task context. To return the IMASK level to 0, use ichg_ims in the non-task context. Moreover, dispatching of tasks is postponed in the non-task context.

(c) Directly change the IMASK level (by using the intrinsic function set_imask() or set_cr() provided by the compiler)

The behavior is much the same as that of chg_ims except that this method is available even when the system is in the CPU-locked state. However, the following differences apply.

- This method incurs less overhead than chg_ims.
- Restrictions apply to the use of this method to change the IMASK level in task contexts. For details, see the following sections.

Here are some possible situations.

Controlling the IMASK level in task contexts

- To mask interrupts at a specific level

  (c) is recommended for better performance, although (b) is also available. However, do not use (c) when changing the IMASK level to be lower than the kernel interrupt mask level (system.system_IPL).

- To mask interrupts at the kernel interrupt mask level

  (a), (b), and (c) are available. (a) is recommended for portability and (c) for better performance.

Controlling the IMASK level in non-task contexts

- To mask interrupts at a specific level

  (c) is recommended for better performance, although (b) is also available.

- To mask interrupts at the kernel interrupt mask level

  (a), (b), and (c) are available. (a) is recommended for portability and (c) for better performance.

4.8.3 Restriction on Service Calls

When SR.IMASK is higher than the kernel interrupt mask level (system.system_IPL), no service call should be issued because service-call processing should not be broken up. If a service call is issued, interrupts will be accepted unexpectedly since the interrupt mask level is lowered during processing of the service call. This leads to incorrect operation of the system.

4.9 CPU Exceptions

When a CPU exception (e.g. an address error or a TRAPA instruction) occurs, a CPU exception handler defined by the user is initiated. The system goes down if no CPU exception handler has been defined.

4.9.1 Types of CPU Exception Handler

There are two CPU exception handlers: direct and normal. A direct CPU exception handler is directly initiated in response to a CPU exception, while a normal CPU exception handler is initiated via the kernel. Select the type of CPU exception handler you wish to use with reference to the following information.

(a) Description format

A normal CPU exception handler is written as a C-language function. When a CPU exception occurs, its number (vector number) and other information (see section 12.6.2) are passed to the normal CPU exception handler as parameters.

A direct CPU exception handler, on the other hand, is written as an interrupt function and requires specification of the #pragma interrupt directive. The portability of the handler function is thus lower than that of a normal CPU exception handler.

No parameters are passed to a direct CPU exception handler. A direct handler should thus be written in assembly language so that it can acquire the information on the CPU exception (see section 12.6.2).

(b) Context

When a CPU exception occurs, the context for the execution of the handler, regardless of its type, is the same as the context before the exception occurred. The same stack is also used. Thus, care must be taken to ensure that the handler does not make an overflow of the stack in use. Since a normal CPU exception handler stores the information on the CPU exception on the stack, such a handler uses more stack space than a direct CPU exception handler. If a CPU exception has occurred with task-dispatch not suspended (e.g. in a task context), task dispatch remains suspended during the execution of a normal CPU exception handler but is not suspended during execution of a direct CPU exception handler.

(c) Service calls

The service calls that can be issued from a normal CPU exception handler are limited to those listed in table 4.4. Also note the restriction described in section 4.8.3, Restriction on Service Calls.

Table 4.4 Service Calls that can be Issued from a Normal CPU Exception Handler

get_tid, iget_tid	ras_tex, iras_tex	sns_tex	sns_ctx	sns_loc
sns_dsp	sns_dpn	vsta_knl, ivsta_knl	vsys_dwn, ivsys_dwn	vsns_tmr

During the execution of a direct CPU exception handler, no service call should be issued if the CPU exception occurred within the kernel. If the CPU exception occurred within an application, however, the same service calls can be issued as were permitted before the CPU exception.

(d) Definition

Specify the VTA_DIRECT attribute to define a direct CPU exception handler.

4.9.2 Reserved Exceptions

TRAPA #60 to TRAPA #63 are reserved for use by the kernel. No handler processing can be defined for these TRAPA instructions.

Section 5 Kernel Functions

This section mainly describes the functions and usage of kernel service calls.

5.1 Task Management

The task management functions are used to perform task operations such as creating, deleting, starting, and ending tasks, and changing task priorities. For details on the task stack, refer to section 4.5.4, Task Stack. The HI7200/MP offers the following task management service calls.

(1) Create Task (cre_tsk or icre_tsk)

Creates a task with the specified ID.

(2) Create Task (acre_tsk or iacre_tsk)

Creates a task with an arbitrary ID that is automatically assigned by the kernel and returned.

(3) Delete Task (del_tsk)

Deletes the task with the specified ID.

(4) Activate Task (act_tsk, iact_tsk)

Activates the task with the specified ID. Unlike sta_tsk and ista_tsk, the activation requests by these service calls are queued, but a start code to be passed to the target task cannot be specified in these service calls. Extended information specified at the time of task creation is passed to the target task.

act_tsk can be issued to a task of the other CPU.

(5) Cancel Task Activation Requests (can_act, ican_act)

Cancels the activation requests that have been queued for the task with the specified ID.

can_act can be issued for a task of the other CPU.

(6) Activate Task (sta_tsk, ista_tsk)

Activates the task with the specified ID. In either service call, unlike in act_tsk or iact_tsk, requests for service call startup of this type are not queued, but a start code to be passed to the target task can be specified.

sta_tsk can be issued for a task of the other CPU.

(7) Terminate Current Task (ext_tsk)

Terminates the current task, placing the task in the DORMANT state. If activation requests for the task have been queued, task startup processing is performed again. In this case, the current task behaves as if it has been reset.

Behavior of the task in response to this service call is the same as the task returning from its entry function.

(8) Terminate and Delete Current Task (exd_tsk)

Terminates and deletes the current task.

(9) Terminate Another Task (ter_tsk)

Terminates another task that is not in the DORMANT state and places the task in the DORMANT state. If activation requests for the task have been queued, task startup processing is performed again. The vchg_tmd service call can mask termination requests issued by ter_tsk.

ter_tsk can be issued for a task of the other CPU.

(10) Change Task Priority (chg_pri, ichg_pri)

Changes the priority of the task with the specified ID. If the priority of a task is changed while the task is in the READY or RUNNING state, the ready queue is also updated (figure 5.1). Moreover, if the target task is placed in the wait queue of an object with the TA_TPRI attribute, the wait queue is also updated (figure 5.2).

chg_pri can be issued for a task of the other CPU.

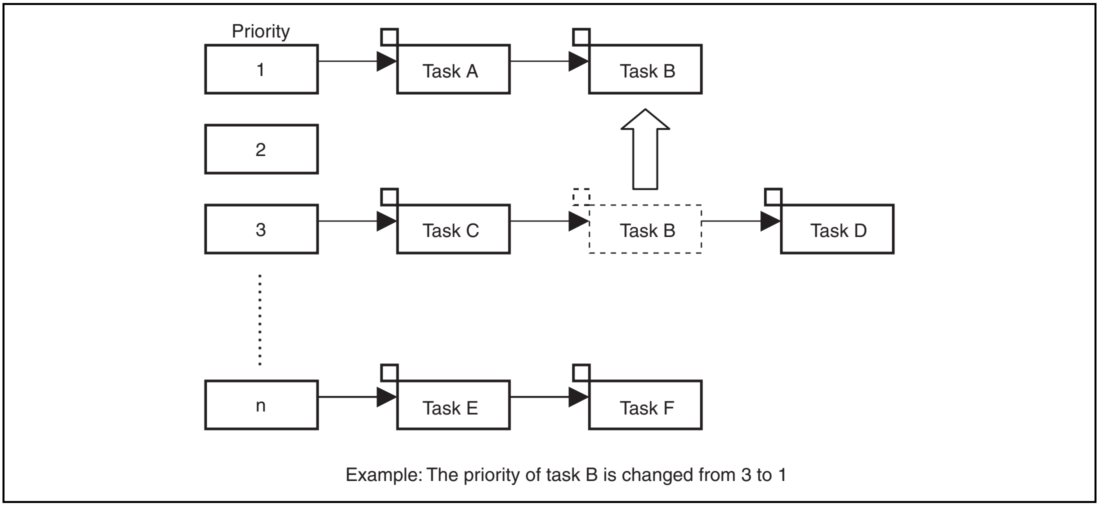


Figure 5.1 Changing Priority

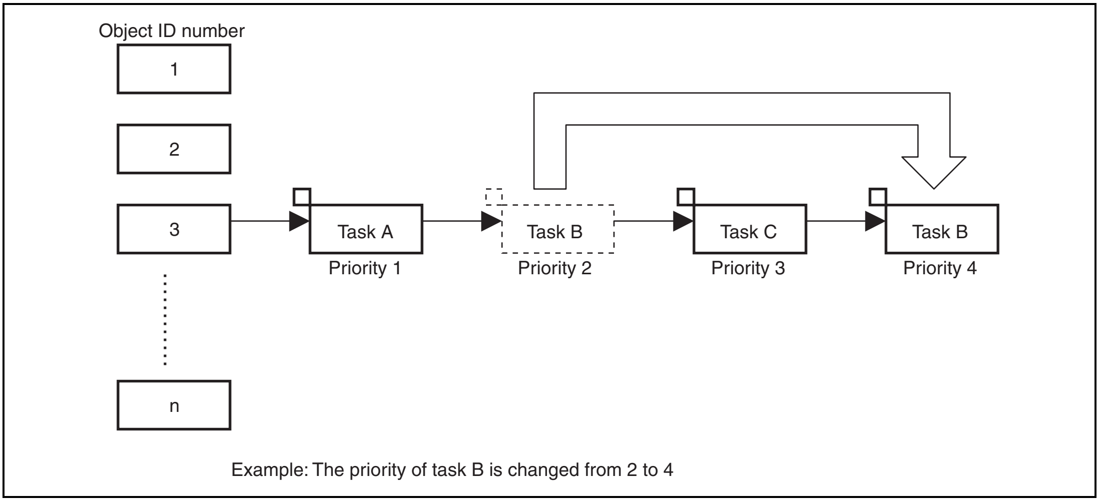


Figure 5.2 Re-Arranging the Wait Queue

However, it is generally recommended that these service calls not be used because changing the priority affects the behavior of the entire system.

A task has two priority levels: base priority and current priority. In general operation, these two priority levels are the same; they differ only while the task has a mutex locked. For details, refer to section 5.9, Mutexes.

(11) Get Task Priority (get_pri, iget_pri)

Acquires the priority of the task with the specified ID.

get_pri can be issued for a task of the other CPU.

(12) Reference Task State (ref_tsk, iref_tsk)

Refers to the state of the task with the specified ID.

ref_tsk can be issued for a task of the other CPU.

(13) Reference Task State: Simple Version (ref_tst, iref_tst)

Refers to the state of the task with the specified ID. Either service call produces less overhead than ref_tsk or iref_tsk because it refers to less information.

ref_tst can be issued for a task of the other CPU.

(14) Change Task Execution Mode (vchg_tmd)

Changes the execution mode of a task with the specified ID. The task execution mode is not defined in the µITRON4.0 specification.

A forcible termination request (service call ter_tsk) issued by another task may make a task enter the DORMANT state with unexpected timing, i.e. before the acquired resources have been released. Service call sus_tsk or isus_tsk may also suspend the execution of a task with unexpected timing.

Service call vchg_tmd can thus mask termination requests and suspension requests.

5.2 Task-Dependent Synchronization Functions

The task-dependent synchronization functions are used to achieve synchronization between tasks by placing tasks in the WAITING, SUSPENDED, or WAITING-SUSPENDED states, or to wake up tasks in the WAITING state. The HI7200/MP offers the following task-dependent synchronization service calls.

(1) Sleep Task (slp_tsk, tslp_tsk) and Wakeup Task (wup_tsk, iwup_tsk)

slp_tsk places the current task in the WAITING state. tslp_tsk performs the same function as slp_tsk except that a timeout period before wakeup is specifiable. wup_tsk or iwup_tsk wakes up a task that has been placed in the WAITING state by slp_tsk or tslp_tsk. While a task is not in a WAITING state initiated by slp_tsk or tslp_tsk, the issued wakeup requests are queued. If a task for which wakeup requests have been queued calls slp_tsk or tslp_tsk, the wakeup request count is decremented by one (-1) and the task does not enter the WAITING state (figure 5.3).

wup_tsk can be issued for a task of the other CPU.

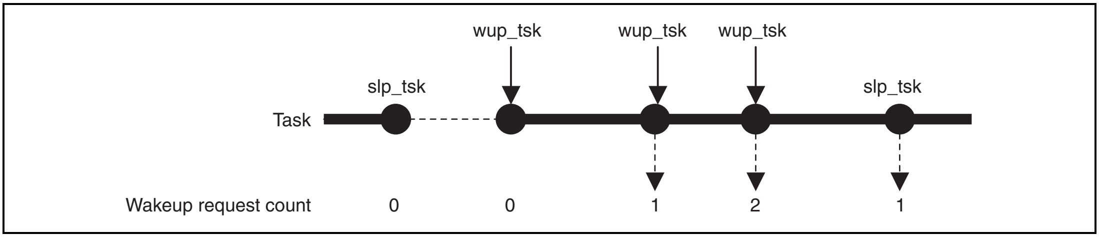


Figure 5.3 Wakeup Request Queue

(2) Cancel Wakeup Task (can_wup, ican_wup)

Cancels the wakeup requests queued for a task with the specified ID (figure 5.4).

can_wup can be issued for a task of the other CPU.

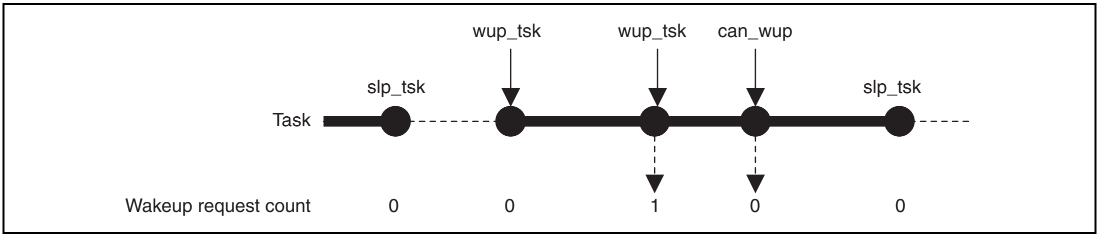


Figure 5.4 Canceling Wakeup Requests

(3) Suspend Task (sus_tsk, isus_tsk) and Resume Task (rsm_tsk, irsm_tsk, frsm_tsk, ifrsm_tsk)

Issuing sus_tsk or isus_tsk forcibly suspends the task with the specified ID (the SUSPENDED state). A task in the READY state is placed in the SUSPENDED state. A task in the WAITING state is placed in the WAITING-SUSPENDED state. Suspension requests issued by calling sus_tsk or isus_tsk are nested.

rsm_tsk or irsm_tsk decrements the suspension count for a task with the specified ID. When the number reaches 0, the task is taken out of the SUSPENDED state (figure 5.5).

frsm_tsk or ifrsm_tsk forcibly releases the task with the specified ID from the SUSPENDED state. The task is returned to its previous state (figure 5.6).

sus_tsk, rsm_tsk, and frsm_tsk can be issued for a task of the other CPU.

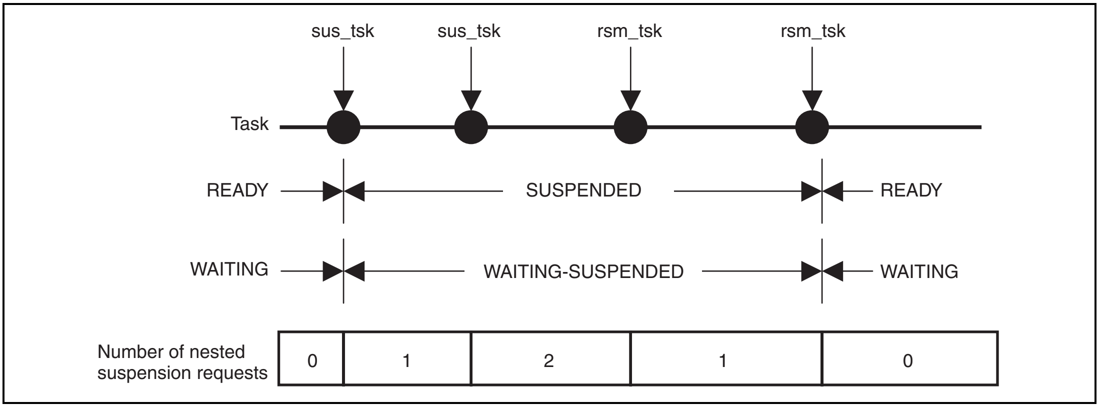


Figure 5.5 Suspending and Resuming Tasks

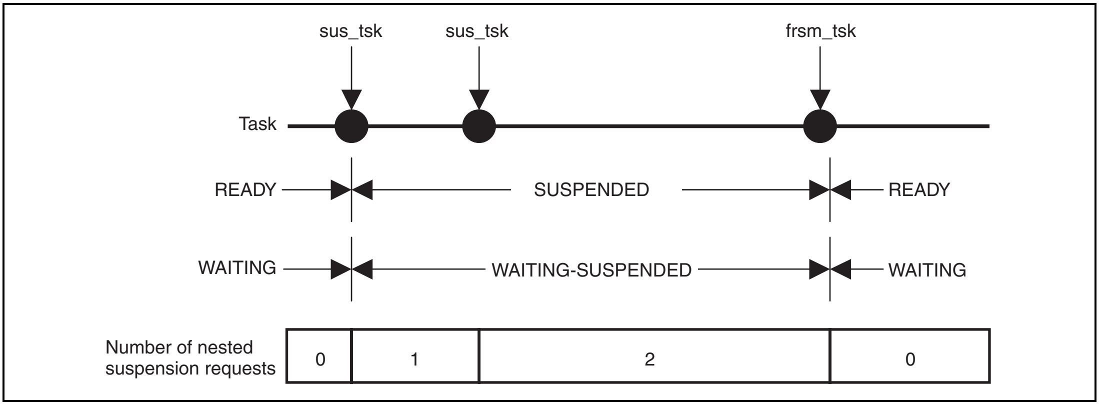


Figure 5.6 Suspending and Forcibly Resuming Tasks

(4) Forcible Release from WAITING State (rel_wai, irel_wai)

rel_wai or irel_wai forcibly releases the task with the specified ID from the WAITING state. Note that neither service call can release a task from the SUSPENDED state.

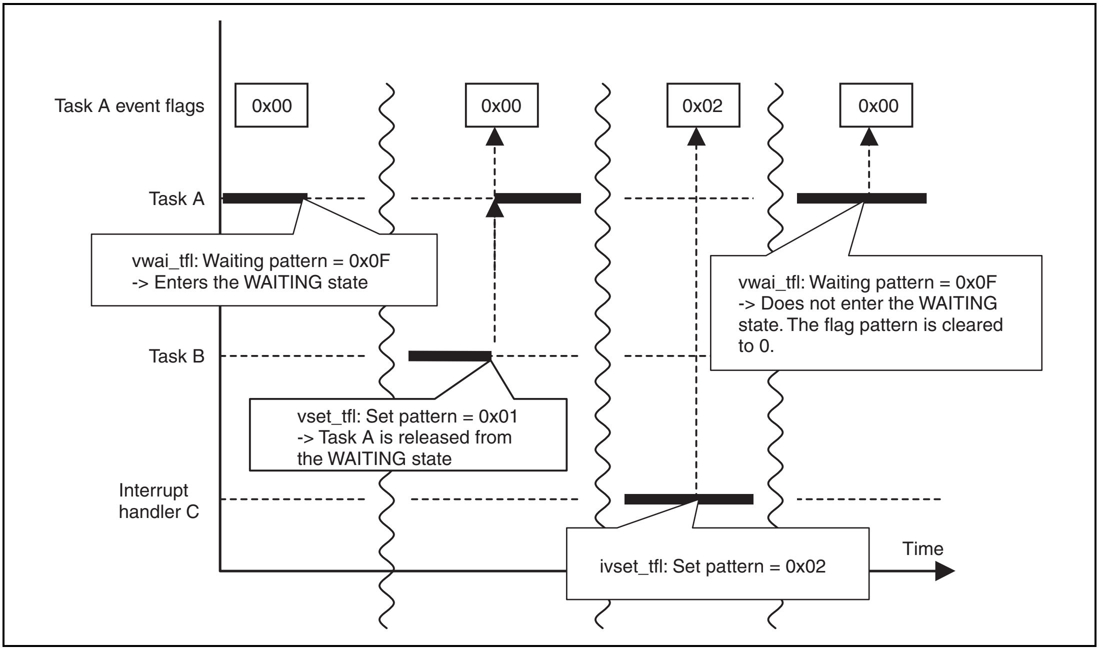
rel_wai can be issued for a task of the other CPU.

(5) Delay Task (dly_tsk)

Transfers the current task from the RUNNING state to a timed WAITING state.

5.3 Task Event Flags

Task event flags are bit patterns for tasks. A task can be made to wait until a specified bit is set in the task event flag for the current task; that is, it can be made to wait until a specified event occurs. Task event flags are not defined in the μITRON4.0 specification. Figure 5.7 shows an example of task event flag operation.



Task A event flags
0x00
0x00
0x02
0x00
Task A
vwai_tfl: Waiting pattern = 0x0F
-> Enters the WAITING state
Task B
vwai_tfl: Waiting pattern = 0x0F
-> Does not enter the WAITING state. The flag pattern is cleared to 0.
vset_tfl: Set pattern = 0x01
-> Task A is released from the WAITING state
Interrupt handler C
ivset_tfl: Set pattern = 0x02
Time

Figure 5.7 Example of Task Event Flag Operation

Control related to task event flags is implemented by the service calls listed below.

(1) Wait for Task Event Flag (vwai_tfl, vtwai_tfl)

vwai_tfl or vtwai_tfl makes a task wait until a specified bit of the task event flag has been set. Once the specified bit is set, the task is released from the WAITING state and the task event flag is cleared to 0. The call returns the pre-clearing bit pattern in the task event flag. If the specified bit has already been set, the task will not enter the WAITING state.

(2) Acquire Task Event Flag (vpol_tfl)

vpol_tfl checks if a specified bit in the event flag is set. The only difference between this service call and vwai_tfl or vtwai_tfl is that an error code is immediately returned and the task does not enter the WAITING state when the wait condition is not satisfied.

(3) Set Task Event Flag (vset_tfl, ivset_tfl)

vset_tfl or ivset_tfl sets a specified bit in the event flag of the task with the specified ID.

vset_tfl can be issued for a task of the other CPU.

(4) Clear Task Event Flag (vclr_tfl, ivclr_tfl)

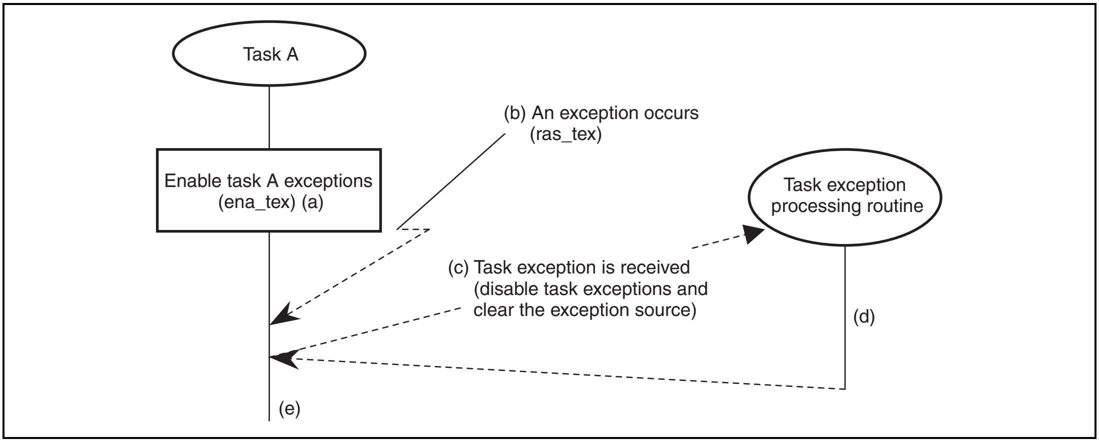
vclr_tfl or ivclr_tfl clears a specified bit in the event flag of the task with the specified ID.

vclr_tfl can be issued for a task of the other CPU.

5.4 Task Exception Handling

Task exception handling is performed when an exception occurs during task execution. Task exception handling is performed asynchronously with task processing and is similar to the function commonly referred to as "signals".

Figure 5.8 shows an example of task exception handling.



Task A
Enable task A exceptions (ena_tex) (a)
(b) An exception occurs (ras_tex)
Task exception processing routine
(c) Task exception is received (disable task exceptions and clear the exception source)
(d)
(e)

Figure 5.8 Example of Task Exception Handling

Description (letters indicate the order of operations):

(a) Task A enables task exceptions.

(b) Service call ras_tex issued during the execution of task A requests a task A exception. The pattern to indicate the reason for the exception should be specified in the call.

(c) When task A is scheduled for execution, the task exception handling routine is initiated instead of the main task A routine. At this time, further task exceptions are disabled, and the task’s current pending-exception pattern is cleared.

(d) The task exception handling routine is executed. The pattern for the pending exception is passed to the task exception handling routine.

(e) On return from the task exception handling routine, execution of the main task A routine is resumed.

Control of task exception handling is implemented by the service calls listed below.

(1) Define Task Exception Handling Routine (def_tex, idef_tex)

Defines a task exception handling routine for the task with the specified ID.

(2) Request Task Exception Handling (ras_tex, iras_tex)

Requests task exception handling for the task with the specified ID. The pattern to indicate the reason for the exception should be specified.

(3) Enable Task Exception Handling (ena_tex)

The current task is shifted to the task exception handling enabled state.

(4) Disable Task Exception Handling (dis_tex)

The current task is shifted to the task exception handling disabled state.

(5) Reference Task Exception Handling Disabled State (sns_tex)

Checks if task exception handling is disabled for the current task.

(6) Reference Task Exception Handling State (ref_tex, iref_tex)

Refers to the task exception handling state of the task with a specified ID.

5.5 Semaphores

A semaphore is an object used to prevent conflicts over resources such as devices or variables shared by multiple tasks. For example, if task switching occurs while task A is updating a shared variable and task B refers to this variable when updating of its value is not complete, task B may incorrectly read the shared variable. Such conflicts can be prevented by using semaphores.

A semaphore provides exclusive control and a synchronization function by expressing the existence of a resource or the number of resources as a counter.

Applications must be programmed so that semaphores are associated with resources to be exclusively controlled.

Note the following rules on exclusive control using a semaphore.

- A task should acquire the semaphore before using the associated resource
- A task should release the semaphore after its usage of the resource is finished

Figure 5.9 shows an example of semaphore usage.

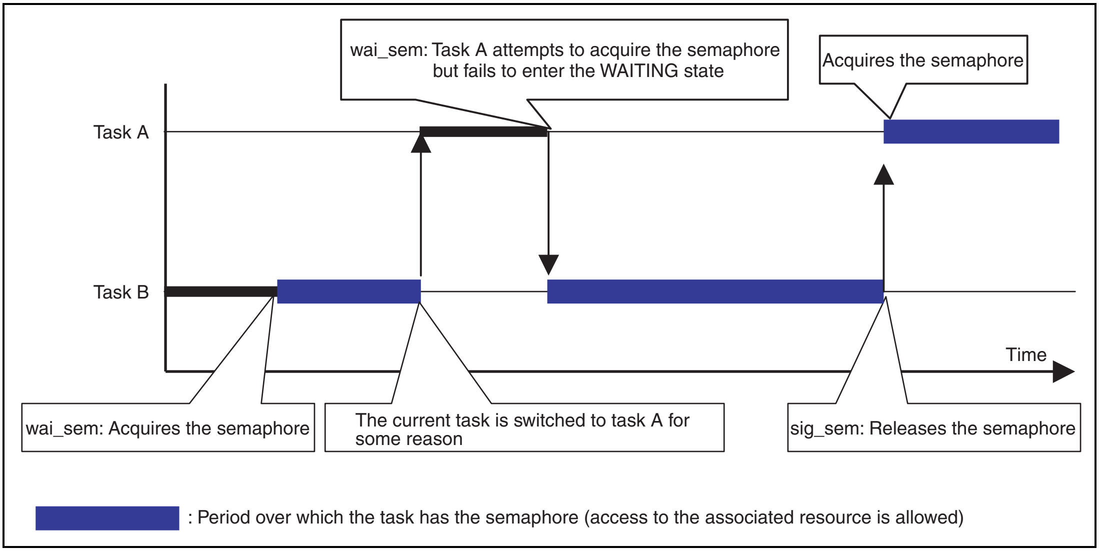

wai_sem: Task A attempts to acquire the semaphore but fails to enter the WAITING state
Acquires the semaphore
Task A
Task B
Time
wai_sem: Acquires the semaphore
The current task is switched to task A for some reason
sig_sem: Releases the semaphore
: Period over which the task has the semaphore (access to the associated resource is allowed)

Figure 5.9 Example of Semaphore Usage

Control related to semaphores is implemented by the service calls listed below.

(1) Create Semaphore (cre_sem, icre_sem)

Creates a semaphore with the specified ID.

(2) Create Semaphore (acre_sem, iacre_sem)

Creates a semaphore with an ID that is automatically assigned by the kernel and returned.

(3) Delete Semaphore (del_sem)

Deletes a semaphore.

(4) Acquire Semaphore Resource (wai_sem, twai_sem)

Acquires a semaphore. If the semaphore's counter has a positive value, the counter is decremented by one. If the semaphore cannot be acquired (semaphore count = 0), the task enters the WAITING state.

wai_sem and twai_sem can be issued for a semaphore of the other CPU.

(5) Acquire Semaphore Resource (pol_sem, ipol_sem)

Acquires a semaphore. The only difference between these service calls and wai_sem or twai_sem is that an error is immediately returned and the task does not enter the WAITING state when the semaphore count is 0.

pol_sem can be issued for a semaphore of the other CPU.

(6) Release Semaphore Resource (sig_sem, isig_sem)

Releases a semaphore. When a task is waiting to acquire a semaphore, either service call makes the task leave the WAITING state. If not, the counter is incremented by one.

sig_sem can be issued for a semaphore of the other CPU.

(7) Reference Semaphore's State (ref_sem, iref_sem)

Refers to the state of a semaphore, including its counter and the IDs of waiting tasks.

ref_sem can be issued for a semaphore of the other CPU.

5.5.1 Priority Inversion

When a semaphore is used for exclusive control of a resource, a problem called priority inversion may arise. This refers to the situation where a task that is not using a resource delays the execution of a task requesting the resource.

Figure 5.10 illustrates this problem. In this figure, tasks A and C are using the same resource, which task B does not use. Task A attempts to acquire a semaphore so that it can use the resource but enters the WAITING state because task C is already using the resource. Task B has a priority higher than task C and lower than task A. Thus, if task B is executed before task C has released the semaphore, release of the semaphore is delayed by the execution of task B. This also delays acquisition of the semaphore by task A. From the viewpoint of task A, a lower-priority task that is not even competing for the resource gets priority over task A. To avoid this problem, use a mutex instead of a semaphore.

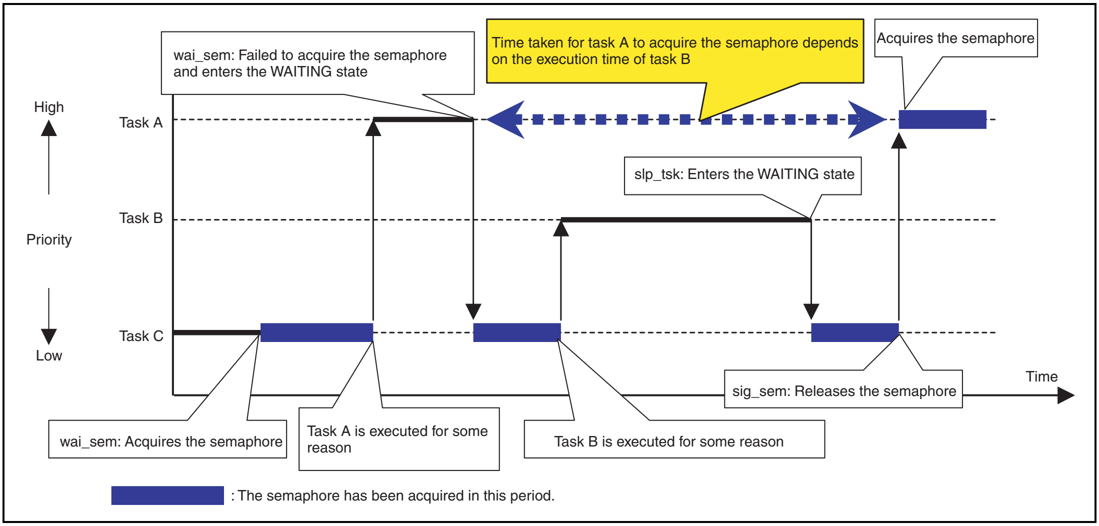

wai_sem: Failed to acquire the semaphore and enters the WAITING state
Time taken for task A to acquire the semaphore depends on the execution time of task B
Acquires the semaphore
High
Task A
slp_tsk: Enters the WAITING state
Task B
Priority
Task C
Low
Time
sig_sem: Releases the semaphore
wai_sem: Acquires the semaphore
Task A is executed for some reason
Task B is executed for some reason
: The semaphore has been acquired in this period.

Figure 5.10 Priority Inversion

5.6 Event Flags

An event flag is a group of bits that correspond to events. One event corresponds to one bit.

A task can be made to wait for one (OR condition) or all (AND condition) of the specified bits to be set. Whether more than one task is allowed to wait for a specific bit of an event flag to be set can be selected as an attribute when the event flag is created. Either of the following attributes is selectable.

- TA_WMUL (more than one task is allowed to wait)
- TA_WSGL (only one task is allowed to wait)

A TA_CLR attribute is also specifiable; in this case, the bit pattern of the event flag is cleared to 0 whenever the wait condition of a task is satisfied.

One feature of the event-flag mechanism is that multiple tasks can be released from the WAITING state at the same time. To allow this, specify the TA_WMUL attribute. Do not specify the TA_CLR attribute in this case.

Figure 5.11 shows an example of task execution control by an event flag. In this figure, six tasks, task A to task F, have been placed in a wait queue. After the flag pattern has been set to 0x0F by the service call set_flg, the pattern satisfies the wait conditions for three of the tasks (task A, task C, and task E). These tasks are sequentially removed from the head of the queue.

If this event flag has the TA_CLR attribute, when task A is released from the WAITING state, the bit pattern of the event flag will be set to 0, and task C and task E will not be removed from the queue.

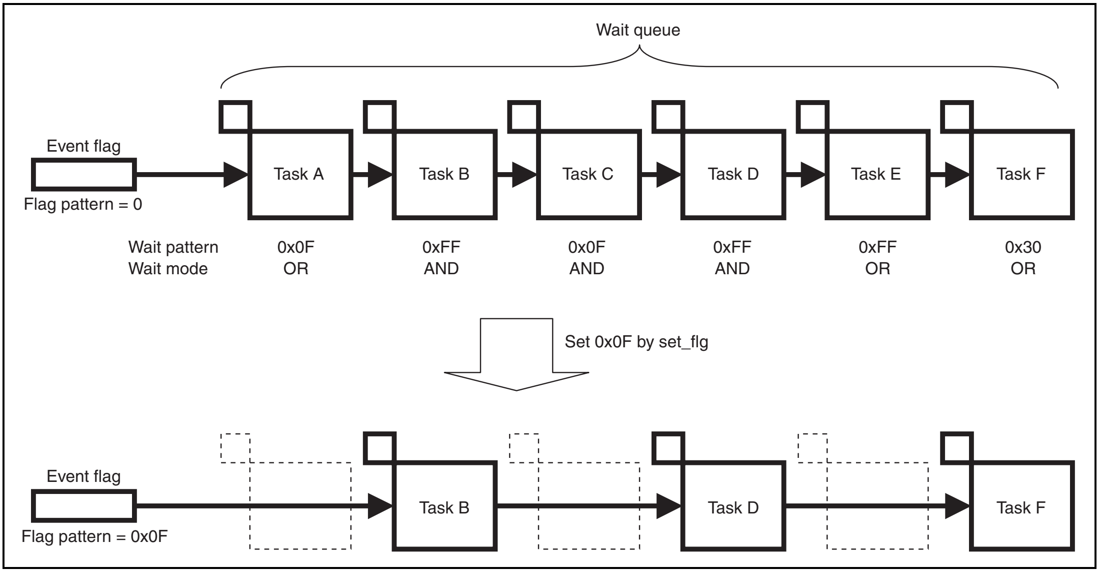


Figure 5.11 Task Execution Control by an Event Flag

Control related to event flags is implemented by the service calls listed below.

(1) Create Event Flag (cre_flg, icre_flg)

Creates an event flag with the specified ID.

(2) Create Event Flag (acre_flg, iacre_flg)

Creates an event flag with an ID that is automatically assigned by the kernel and returned.

(3) Delete Event Flag (del_flg)

Deletes an event flag.

(4) Wait for Event-Flag Setting (wai_flg, twai_flg)

Makes a task wait until specific bits in the event flag have been set. Select either of the following wait conditions.

- AND condition: The task waits until all of the specified bits have been set
- OR condition: The task waits until any of the specified bits has been set

When a task is released from the WAITING state, the value of the event flag at satisfaction of the wait condition is returned to the task that issued this service call. If the TA_CLR attribute has been specified for the event flag, the event flag is also cleared to 0. In this case, the value of the event flag immediately before it was cleared is returned to the task that issued this service call.

wai_flg and twai_flg can be issued for an event flag of the other CPU.

(5) Acquire Event Flag Value (pol_flg, ipol_flg)

Checks if specified bits in an event flag have been set. The only difference between these service calls and wai_flg or twai_flg is that an error code is immediately returned and the task does not enter the WAITING state if the condition is not satisfied.

pol_flg can be issued for an event flag of the other CPU.

(6) Set Event Flag (set_flg, iset_flg)

Sets an event flag to a specified bit pattern. This may release tasks with wait conditions that match the pattern.

set_flg can be issued for an event flag of the other CPU.

(7) Clear Event Flag (clr_flg, iclr_flg)

Clears specified bits of an event flag.

clr_flg can be issued for an event flag of the other CPU.

(8) Reference Event Flag State (ref_flg, iref_flg)

Refers to the state of an event flag, including its bit pattern and the IDs of waiting tasks.

ref_flg can be issued for an event flag of the other CPU.

5.7 Data Queues

A data queue is an object used to achieve the communication of single words (32-bit units) of data. Figure 5.12 shows the structure of a data queue.

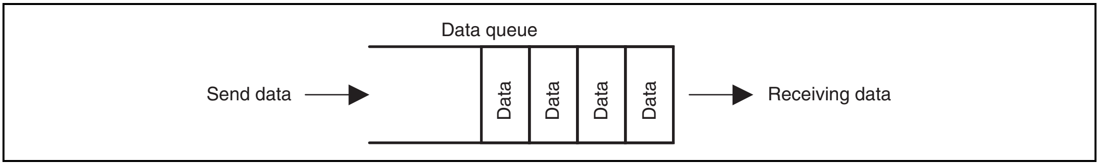


Figure 5.12 Data Queue

Data are sent to a data queue for storage. When data are received from a data queue, the oldest data are taken out first (on an FIFO basis). The maximum number of data items that can be queued in a data queue is specifiable when the data queue is created.

Areas for use as data queues can be allocated in the default data-queue area owned by the kernel or in an area specified by an application. Either method is selectable when the data queue is created. The size of the default data-queue area should be specified as memdtq.all_memsize in the cfg file.

Data queues are controlled by the service calls listed below.

(1) Create Data Queue (cre_dtq, icre_dtq)

Creates a data queue with the specified ID.

(2) Create Data Queue (acre_dtq, iacre_dtq)

Creates a data queue with an ID that is automatically assigned by the kernel and returned.

(3) Delete Data Queue (del_dtq)

Deletes a data queue.

(4) Send Data to Data Queue (snd_dtq, tsnd_dtq)

Sends data to a data queue. When the data queue is full of data, the calling task enters the WAITING state.

snd_dtq and tsnd_dtq can be issued for a data queue of the other CPU.

(5) Send Data to Data Queue (psnd_dtq, ipsnd_dtq)

Sends data to a data queue. The only difference between these service calls and snd_dtq or tsnd_dtq is that an error code is immediately returned and the calling task does not enter the WAITING state if the data queue is full.

psnd_dtq can be issued for a data queue of the other CPU.

(6) Forcibly Send Data to Data Queue (fsnd_dtq, ifsnd_dtq)

Sends data to a data queue. When the data queue is full of data, the oldest data are deleted and the new data are sent.

fsnd_dtq can be issued to a data queue of the other CPU.

(7) Receive Data from Data Queue (rcv_dtq, trcv_dtq)

Receives data from a data queue. When the data queue has no data, the calling task enters the WAITING state. If the data queue was full of data and a task was waiting to send data, this call releases the first task in the wait queue for sending data from the WAITING state.

rcv_dtq and trcv_dtq can be issued for a data queue of the other CPU.

(8) Receive Data from Data Queue (prcv_dtq, iprcv_dtq)

Receives data from a data queue. When the data queue has no data, an error code is returned. If the data queue was full of data and a task was waiting to send data, this call releases the first task in the wait queue for sending data from the WAITING state.

prcv_dtq can be issued for a data queue of the other CPU.

(9) Reference Data Queue State (ref_dtq, iref_dtq)

Refers to the state of a data queue, including the number of data stored in the queue and the IDs of tasks waiting to send or receive data.

ref_dtq can be issued for a data queue of the other CPU.

5.8 Mailboxes

A mailbox is an object used to send or receive messages, which are data of a designated size. Figure 5.13 shows the structure of a mailbox.

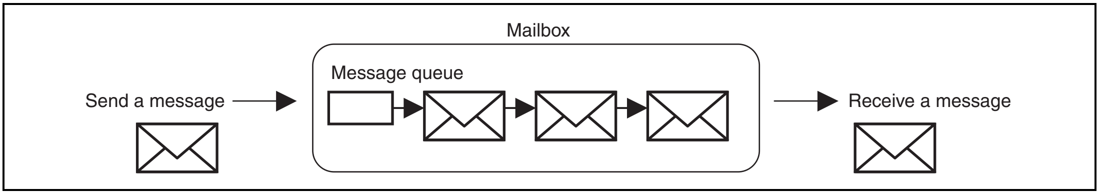


Figure 5.13 Mailbox

High-speed data communications are achieved regardless of the message size because only the addresses where the messages start are sent and received. Applications should create messages in memory areas that are accessible by both the sending and receiving tasks (i.e., messages should not be created in the local variable areas). A sending task should not access the message area after it has sent the message.

Using a mailbox for data communications between two CPUs also requires that either

- messages are created in non-cacheable areas or
- the contents of messages are written back into the actual memory before they are sent.

Messages in a mailbox with the TA_MPRI attribute have priority levels. Of the messages in the mailbox, that with the highest priority will be received first. If priority levels for messages are not necessary, specify TA_MFIFO rather than TA_MPRI.

Mailboxes are controlled by the service calls listed below.

(1) Create Mailbox (cre_mbx, icre_mbx)

Creates a mailbox with the specified ID.

(2) Create Mailbox (acre_mbx, iacre_mbx)

Creates a mailbox with an ID that is automatically assigned by the kernel and returned.

(3) Delete Mailbox (del_mbx)

Deletes a mailbox.

(4) Send Message to Mailbox (snd_mbx, isnd_mbx)

Sends a message to a mailbox.

snd_mbx can be issued for a mailbox of the other CPU.

(5) Receive Message from Mailbox (rcv_mbx, trcv_mbx)

Receives a message from a mailbox. When the mailbox has no message, the task is in the WAITING state until a message is sent to the mailbox.

rcv_mbx and trcv_mbx can be issued for a mailbox of the other CPU.

(6) Receive Message from Mailbox (prcv_mbx, iprcv_mbx)

Receives a message from a mailbox. The only difference between these service calls and rcv_mbx or trcv_mbx is that an error code is immediately returned and the task does not enter the WAITING state if the mailbox has no message.

prcv_mbx can be issued for a mailbox of the other CPU.

(7) Reference Mailbox State (ref_mbx, iref_mbx)

Refers to the address of the first message queued in the mailbox and the IDs of waiting tasks.

ref_mbx can be issued for a mailbox of the other CPU.

5.9 Mutexes

A mutex is an object used to achieve exclusive control. It differs from a semaphore on the following points.

(a) A priority ceiling protocol is applied to avoid priority inversion problems.

(b) A mutex can only be used for exclusive control of a single resource.

A detailed description of (a) is given below.

The priority ceiling protocol is the only method for controlling the priorities of tasks in mutexes of the HI7200/MP kernel. Strictly speaking, however, the protocol supports a simplified priority ceiling protocol. In the protocol, each mutex has a specified ceiling priority. When a task acquires (locks) a mutex, the priority of the task is raised to this ceiling priority. When the task releases (unlocks) the mutex, the kernel returns the priority of the task to the previous level unless the task has locked another mutex.

Figure 5.14 shows an example of mutex usage.

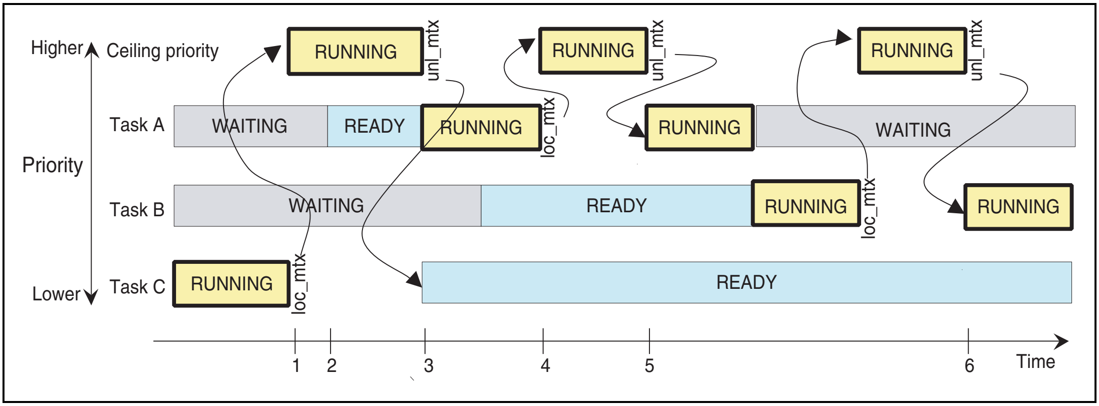


Figure 5.14 Example of Mutex Usage

Description:

1. Task C locks a mutex by issuing loc_mtx. The priority of task C is raised to the ceiling priority specified for the mutex.
2. Task A enters the READY state while task C is being executed at the ceiling priority. The priority of task A is higher than that initially specified for task C. However, task C now locks the mutex and is thus executed at the ceiling priority. Since this is higher than that of task A, task A cannot enter the RUNNING state. In other words, while task C has the mutex locked, execution of task C continues even if task A, with its higher initial priority, becomes ready.
3. Task C unlocks the mutex by issuing unl_mtx. The priority of task C returns to the initial level and higher-priority task A enters the RUNNING state.
4. Task A issues loc_mtx to raise its priority to the ceiling priority.
5. Task A issues unl_mtx to return its priority to the initial level.
6. Task B issues loc_mtx to raise its priority to the ceiling priority.
7. Task B issues unl_mtx to return its priority to the initial level.

Mutexes are controlled by the service calls listed below.

(1) Create Mutex (cre_mtx, icre_mtx)

Creates a mutex with the specified ID and ceiling priority.

(2) Create Mutex (acre_mtx, iacre_mtx)

Creates a mutex with an ID that is automatically assigned by the kernel and returned, and specifies the ceiling priority for the mutex.

(3) Delete Mutex (del_mtx)

Deletes a mutex.

(4) Lock Mutex (loc_mtx, tloc_mtx)

Locks a mutex and raises the priority of the locking task to its ceiling priority. When another task has already locked the mutex, the task that issued loc_mtx or tloc_mtx enters the WAITING state until the mutex is unlocked.

(5) Lock Mutex (ploc_mtx)

Locks a mutex and raises the priority of the locking task to its ceiling priority. The only difference between this service call and loc_mtx or tloc_mtx is that an error is immediately returned and the task that issued ploc_mtx does not enter the WAITING state when another task has already locked the mutex.

(6) Unlock Mutex (unl_mtx)

Unlocks a mutex. If a task is waiting to lock the mutex, this service call makes the task leave the WAITING state.

(7) Reference Mutex State (ref_mtx, iref_mtx)

Refers to the state of a mutex, including the ID of a task that has locked the mutex and of waiting tasks.

5.9.1 Base Priority and Current Priority

A task has two priority levels: base priority and current priority. Tasks are scheduled according to current priority.

While a task does not have a mutex locked, its current priority is always the same as its base priority.

When a task locks a mutex, only its current priority is raised to the ceiling priority specified for the mutex.

When priority-changing service call chg_pri or ichg_pri is issued, both the base priority and current priority are changed if the specified task does not have a mutex locked. When the specified task locks a mutex, only the base priority is changed. When the specified task has a mutex locked or is waiting to lock a mutex, an E_ILUSE error is returned if a priority higher than the ceiling priority of the mutex is specified.

The current priority can be checked through service call get_pri or iget_pri.

5.10 Message Buffers

Like a mailbox, a message buffer is an object for the sending and receiving of messages, which are data of a designated size. The only difference is that the actual contents of the messages are copied and passed. For this reason, the message area becomes available immediately after a message has been sent, regardless of whether or not the receiving task has received the message.

Figure 5.15 shows the structure of a message buffer.

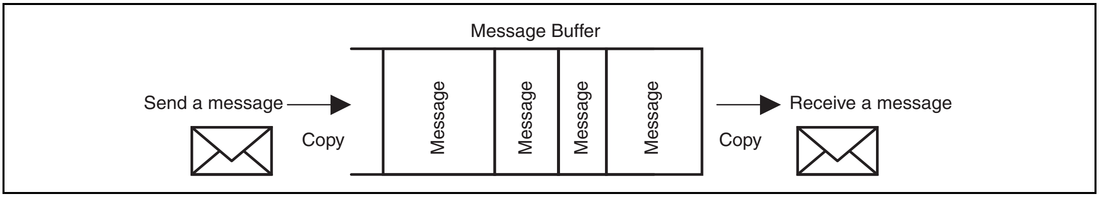


Figure 5.15 Message Buffer

Messages sent to a message buffer are stored in the buffer. When a message is received from a message buffer, the oldest message is taken out first (i.e. operation is FIFO).

Areas for use as message buffers can be allocated in the default message-buffer area owned by the kernel or in an area specified by an application. Either method is selectable when the message buffer is created. The size of the default message-buffer area should be specified as memmbf.all_memsize in the cfg file.

Message buffers are controlled by the service calls listed below.

(1) Create Message Buffer (cre_mbf, icre_mbf)

Creates a message buffer with the specified ID.

(2) Create Message Buffer (acre_mbf, iacre_mbf)

Creates a message buffer with an ID that is automatically assigned by the kernel and returned.

(3) Delete Message Buffer (del_mbf)

Deletes a message buffer.

(4) Send Message to Message Buffer (snd_mbf, tsnd_mbf)

Sends a message to a message buffer. For a message to be sent to a massage buffer, the message buffer must have at least the following amount of free space:

(size of the message in bytes rounded up to a multiple of 4) + 4

When the message buffer has less free space than is required, the task is in the WAITING state until enough space becomes available.

snd_mbf and tsnd_mbf can be issued for a message buffer of the other CPU.

(5) Send Message to Message Buffer (psnd_mbf, ipsnd_mbf)

Sends a message to a message buffer. The only difference between these service calls and snd_mbf or tsnd_mbf is that an error code is immediately returned and the task does not enter the WAITING state if the message buffer does not have enough free space.

psnd_mbf can be issued for a message buffer of the other CPU.

(6) Receive Message from Message Buffer (rcv_mbf, trcv_mbf)

Receives a message from a message buffer. If the message buffer has no messages, the task is in the WAITING state until a message is sent to the message buffer. When a message is received from the message buffer, free space in the message buffer increases by the following amount:

(size of the message in bytes rounded up to multiple of 4) + 4

If the amount of free space in the message buffer becomes larger than the size of a message that a task is waiting to send, the message is sent to the message buffer and the task leaves the WAITING state.

rcv_mbf and trcv_mbf can be issued for a message buffer of the other CPU.

(7) Receive Message from Message Buffer (prcv_mbf)

Receives a message from a message buffer. The only difference between this service call and rcv_mbf or trcv_mbf is that an error code is immediately returned and the task does not enter the WAITING state if the message buffer has no messages.

prcv_mbf can be issued for a message buffer of the other CPU.

(8) Reference Message Buffer State (ref_mbf, iref_mbf)

Refers to the state of a message buffer, including the number of messages it contains, the amount of free space, and the IDs of tasks waiting to send or receive messages.

ref_mbf can be issued for a message buffer of the other CPU.

5.11 Fixed-Sized Memory Pools

A fixed-sized memory pool is an object used to dynamically allocate and release memory blocks of fixed size. While fixed-sized memory pools cannot be used to acquire memory blocks of arbitrary size, their advantage over variable-sized memory pools is that acquiring and releasing blocks produces less overhead. To create a fixed-sized memory pool, specify the size and number of blocks. Figure 5.16 is a schematic view of a fixed-sized memory pool.

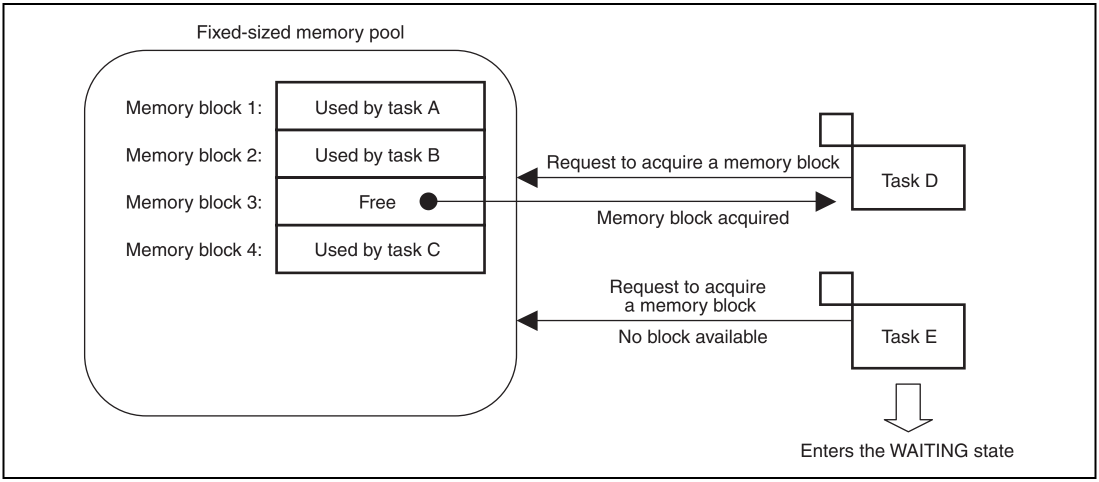


Figure 5.16 Fixed-Sized Memory Pool

Areas for use as fixed-sized memory pools can be allocated from the default fixed-sized memory pool area owned by the kernel or in an area specified by an application. Either method is selectable when the pool is created. The size of the default fixed-sized memory pool area should be specified as memmpf.all_memsize in the cfg file.

The user can choose either of the following management methods by the system.mpfmanage setting in the cfg file.

(1) Conventional method (with IN specified for system.mpfmanage)

The kernel places the kernel management tables adjacent to the memory blocks in the memory pool.

(2) Extended method (with OUT specified for system.mpfmanage)

The kernel places the kernel management tables outside the memory pool.

With this method, the application must specify the address of the management tables when creating the fixed-sized memory pool. The application must also allocate an area for management tables. However, this method eases alignment of the addresses of memory blocks acquired from fixed-sized memory pools.

Control related to fixed-sized memory pools is implemented by the service calls listed below.

(1) Create Fixed-Sized Memory Pool (cre_mpf, icre_mpf)

Creates a fixed-sized memory pool with the specified ID.

(2) Create Fixed-Sized Memory Pool (acre_mpf, iacre_mpf)

Creates a fixed-sized memory pool with an ID that is automatically assigned by the kernel and returned.

(3) Delete Fixed-Sized Memory Pool (del_mpf)

Deletes a fixed-sized memory pool.

(4) Get Memory Block (get_mpf, tget_mpf)

Acquires a fixed-sized memory block. When no memory block is available in the memory pool, the task is in the WAITING state until a memory block is released.

get_mpf and tget_mpf can be issued to a fixed-sized memory pool of the other CPU.

(5) Get Memory Block (pget_mpf, ipget_mpf)

Acquires a fixed-sized memory block. The only difference between these service calls and get_mpf or tget_mpf is that an error code is immediately returned and the task does not enter the WAITING state if no memory blocks are available in the memory pool.

pget_mpf can be issued for a fixed-sized memory pool of the other CPU.

(6) Release Memory Block (rel_mpf, irel_mpf)

Releases a fixed-sized memory block. When a task is waiting to acquire a memory block, either service call makes the task leave the WAITING state.

rel_mpf can be issued for a fixed-sized memory pool of the other CPU.

(7) Reference Fixed-Sized Memory Pool State (ref_mpf, iref_mpf)

Refers to the state of a fixed-sized memory pool, including the number of available memory blocks and the IDs of waiting tasks.

ref_mpf can be issued for a fixed-sized memory pool of the other CPU.

5.12 Variable-Sized Memory Pools

A variable-sized memory pool is an object used to dynamically allocate and release memory blocks of desired size. While variable-sized memory pools can be used to acquire memory blocks of arbitrary size, they have a disadvantage in that acquiring and releasing blocks produces more overhead than with fixed-sized memory pools. Also note that variable-sized memory pools are subject to fragmentation as described in section 5.12.1, Controlling Memory Fragmentation.

Areas for use as variable-sized memory pools can be allocated from the default variable-sized memory pool area owned by the kernel or in an area specified by an application. Either method is selectable when the pool is created. The size of the default variable-sized memory pool area should be specified as memmpl.all_memsize in the cfg file.

Control related to variable-sized memory pools is implemented by the service calls listed below.

(1) Create Variable-Sized Memory Pool (cre_mpl, icre_mpl)

Creates a variable-sized memory pool with a specified ID.

(2) Create Variable-Sized Memory Pool (acre_mpl, iacre_mpl)

Creates a variable-sized memory pool with an ID that is automatically assigned by the kernel and returned.

(3) Delete Variable-Sized Memory Pool (del_mpl)

Deletes a variable-sized memory pool.

(4) Get Memory Block (get_mpl, tget_mpl)

Acquires a variable-sized memory block. When a variable-sized memory block is acquired, a management table is created in the memory pool. The available space in the memory pool is thus reduced by the size of the memory block and that of the management table. When the memory pool lacks the space for allocation of the block, the task is in the WAITING state until the memory pool has enough available space. For details of the management table, refer to section 5.12.2, Management of Variable-Sized Memory Pools.

get_mpl and tget_mpl can be issued for a variable-sized memory pool of the other CPU.

(5) Get Memory Block (pget_mpl, ipget_mpl)

Acquires a variable-sized memory block. The only difference between these service calls and get_mpl or tget_mpl is that an error is immediately returned and the task does not enter the WAITING state when no memory block can be acquired from the memory pool.

pget_mpl can be issued for a variable-sized memory pool of the other CPU.

(6) Release Memory Block (rel_mpl, irel_mpl)

Releases a variable-sized memory block. Releasing a memory block increases the amount of available space in the variable-sized memory pool. For details, refer to section 5.12.2, Management of Variable-Sized Memory Pools.

When a task has been waiting to acquire a block and the release of another block gives the memory pool enough available space, the task leaves the WAITING state and acquires the requested memory block.

rel_mpl can be issued for a variable-sized memory pool of the other CPU.

(7) Reference Variable-Sized Memory Pool State (ref_mpl, iref_mpl)

Refers to the state of a variable-sized memory pool, including the total amount of available memory, the maximum size of a contiguous memory block, and the IDs of waiting tasks.

ref_mpl can be issued for a variable-sized memory pool of the other CPU.

5.12.1 Controlling Memory Fragmentation

The repeated acquisition and release of memory from variable-sized pools causes fragmentation of free space. Contiguous free space can thus become insufficient even when the total amount of free space is sufficient; this makes the acquisition of relatively large memory areas impossible (figure 5.17).

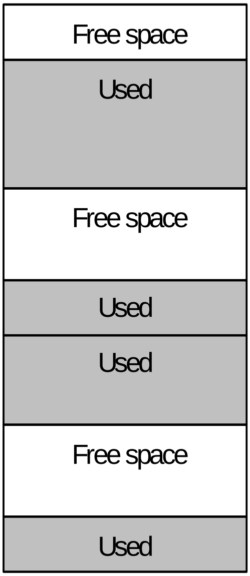


Figure 5.17 Fragmentation of Free Space

We strongly recommend specifying NEW for system.newmpl in the cfg file because this slightly reduces the degree of fragmentation. If PAST is specified for system.newmpl, on the other hand, the kernel manages variable-sized memory pools in the same manner as the version 1 of the HI7000/4 series.

Selecting NEW for system.newmpl allows specification of the VTA_UNFRAGMENT attribute to further reduce the fragmentation of variable-sized memory pools and adds parameters (minimum block size, number of sectors, and management table address) for the use of this attribute to the T_CMPL structure, which is specified at the time of variable-sized memory pool creation. Although specification of the VTA_UNFRAGMENT attribute generally helps in reducing fragmentation, the degree of fragmentation will still depend on usage of the variable-sized memory pools.

When the VTA_UNFRAGMENT attribute is specified, the variable-sized memory pools are managed by a sector-based method.

In this method, blocks with sizes up to (smallest block size in bytes × 8) are handled as "small blocks". The sizes allocated in response to block acquisition requests are rounded up as shown in table 5.1.

When a "small block" is requested, the kernel creates a sector consisting of blocks of the rounded-up size. The sector size is always minblksz × 32. This means that the number of blocks in a sector depends on the requested size.

Table 5.1 Small Block Control

Acquisition Request Size (blksz)*	Size after Rounding*	Number of Blocks in a Sector
0 < blksz ≤ minblksz	minblksz	32
minblksz < blksz ≤ minblksz × 2	minblksz × 2	16
minblksz × 2 < blksz ≤ minblksz × 4	minblksz × 4	8
minblksz × 4 < blksz ≤ minblksz × 8	minblksz × 8	4

Note: blksz: Requested size
minblksz: Smallest block size in bytes

Then the kernel allocates one of the memory blocks in the sector in response to the request. The remaining blocks in the sector are reserved for later requests for small blocks.

In this manner, small blocks are allocated contiguously to leave larger free spaces available.

Figure 5.18 shows an example of a variable-sized memory pool when the minimum block size is 32.

First a 32-byte memory block is requested. In response, sector [A] with 32 blocks × 32 bytes = 1024 bytes is allocated and 32-byte area [A-1] in the sector is allocated as the requested block (figure 5.18 (1)). When a 16-byte memory block is then requested, 32-byte area [A-2] in sector A is allocated in response (figure 5.18 (2)).

Next, a 36-byte memory block is requested. Since the size of each block in sector A is 32 bytes, no block in sector A can be assigned in response. Instead, new sector [B] is allocated for 16 blocks × 64 bytes (since the requested size, 36, is rounded up to a multiple of the minimum block size) = 1024 bytes, and 64-byte area [B-1] is allocated as the requested block (figure 5.18 (3)).

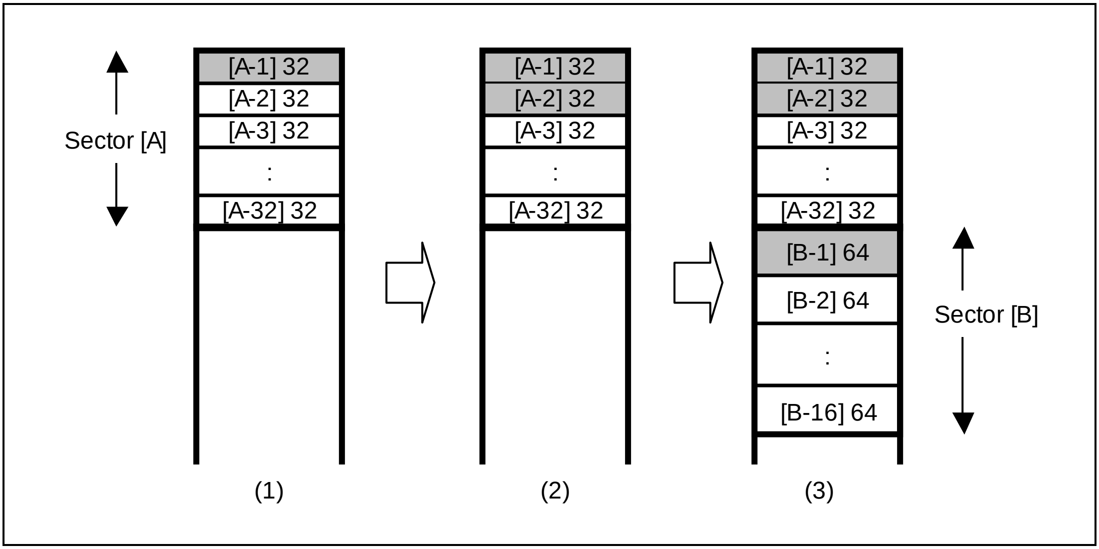


Figure 5.18 Example of Variable-Sized Memory Pool

If the maximum number of sectors has already been used or there is not enough contiguous free space to create a new sector, the requested size is allocated and a sector is not created. In such cases, free space may be fragmented. If there is too little contiguous free space for a block of the requested size, the memory block is allocated in a sector for larger blocks.

When all blocks in a sector are released, the sector itself is also released.

When a large block is requested (larger than minblksz × 8), the kernel always allocates a block of the requested size and does not create a sector.

5.12.2 Management of Variable-Sized Memory Pools

In each variable-sized memory pool, the kernel creates management tables to manage the allocated memory blocks. When determining the size required for a variable-sized memory pool, note that the pool area is used for kernel management tables as well as the memory block areas acquired by applications.

(1) When PAST is specified for system.newmpl

The kernel creates a 16-byte management table whenever a memory block is acquired. This management table is released when the memory block is released.

(2) When NEW is specified for system.newmpl

If the VTA_UNFRAGMENT attribute is specified, the kernel also creates a 32-byte management table when a new sector is created in response to acquisition of a memory block. This management table is released when the sector is returned.

The kernel also creates a 32-byte management table when a memory block is allocated outside the sectors while the VTA_UNFRAGMENT attribute is specified or when a memory block is allocated while the VTA_UNFRAGMENT attribute is not specified. This management table is released when the memory block is returned.

5.13 Time Management

The kernel provides the following functions related to time management:

- Reference to and setting of the system clock
- Time event handler (cyclic handler, alarm handler, and overrun 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 time used to define time parameters for the service calls is 1 ms. The system.tic_nume and system.tic_deno settings in the cfg file determine the cycle for supply of the basic time tick.

Using the time management functions requires the TIMER specification for clock.timer and creation of a timer driver. For how to create a timer driver, see section 12.9, Timer Driver. A sample timer driver is provided with this product.

5.13.1 Task Timeout

Timeout values for WAITING states are specifiable with service calls that start with **t**, such as tslp_tsk and twai_sem.

If the wait condition has not been satisfied after the specified timeout period has elapsed, the task is taken out of the WAITING state and the error code E_TMOUT is returned as the return value for the service call.

Timeouts can be used to detect abnormal behavior in the form of events that should have been generated within the timeout period but were not.

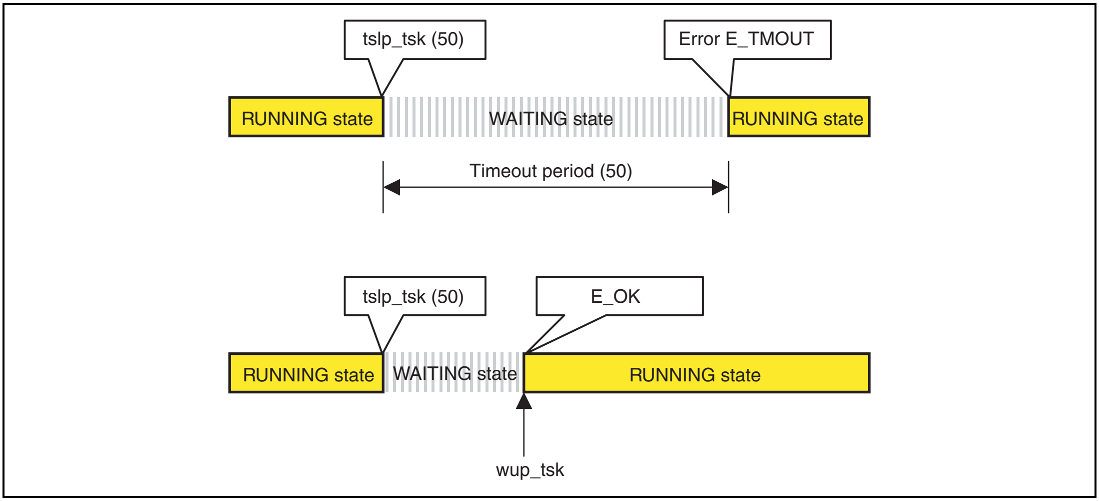


Figure 5.19 Timeout

5.13.2 Delaying Tasks

A task can be placed in the WAITING state for a specified time by using dly_tsk. When the specified time has elapsed, the task is taken out of the WAITING state and E_OK is returned as the return value.

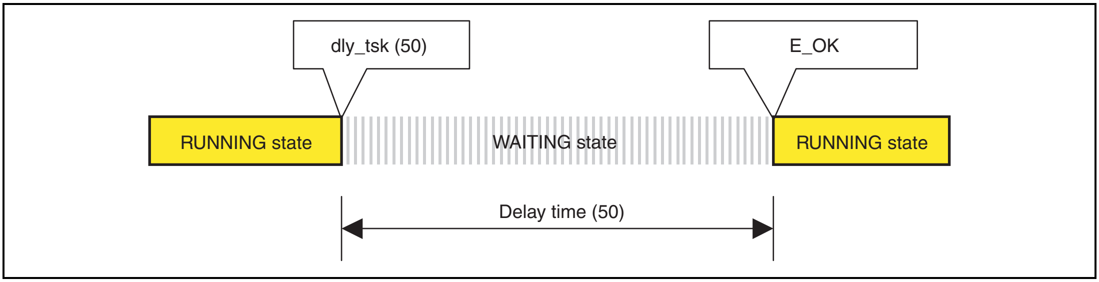


Figure 5.20 Delaying a Task

5.13.3 Stopping and Restarting the Timer

The hardware timer used by the kernel can be stopped by calling vstp_tmr and restarted by calling vrst_tmr or ivrst_tmr.

These functions are used to stop timer interrupts when placing the CPU in the sleep state. An upper time limit is specified for vstp_tmr. vstp_tmr only stops the timer after no timer event (task timeout or time event handler) has occurred within the upper time limit.

vrst_tmr or ivrst_tmr is used to restart the timer. When another method is available for measuring the time over which the timer is stopped, an elapsed time can be specified in vrst_tmr or ivrst_tmr. This elapsed time specification can be used to adjust the time kept by the kernel, which may lead to a timer event. When the time over which the timer is stopped is not measurable, always specify the elapsed time as 0.

Use vsns_tmr to check whether or not the timer is running.

These functions are not defined in the µITRON4.0 specification.

5.13.4 Cyclic Handlers

A cyclic handler is a time-event handler that is initiated cyclically at a specific interval after a specified initiation phase has elapsed. A cyclic handler is initiated either with or without an initiation phase to be preserved. When an initiation phase is to be preserved, initiation of the cyclic handler is based on the timing of cyclic-handler creation. When an initiation phase is not to be preserved, initiation of the cyclic handler is based on the timing with which the cyclic handler is started.

Figure 5.21 shows examples of cyclic handler operation.

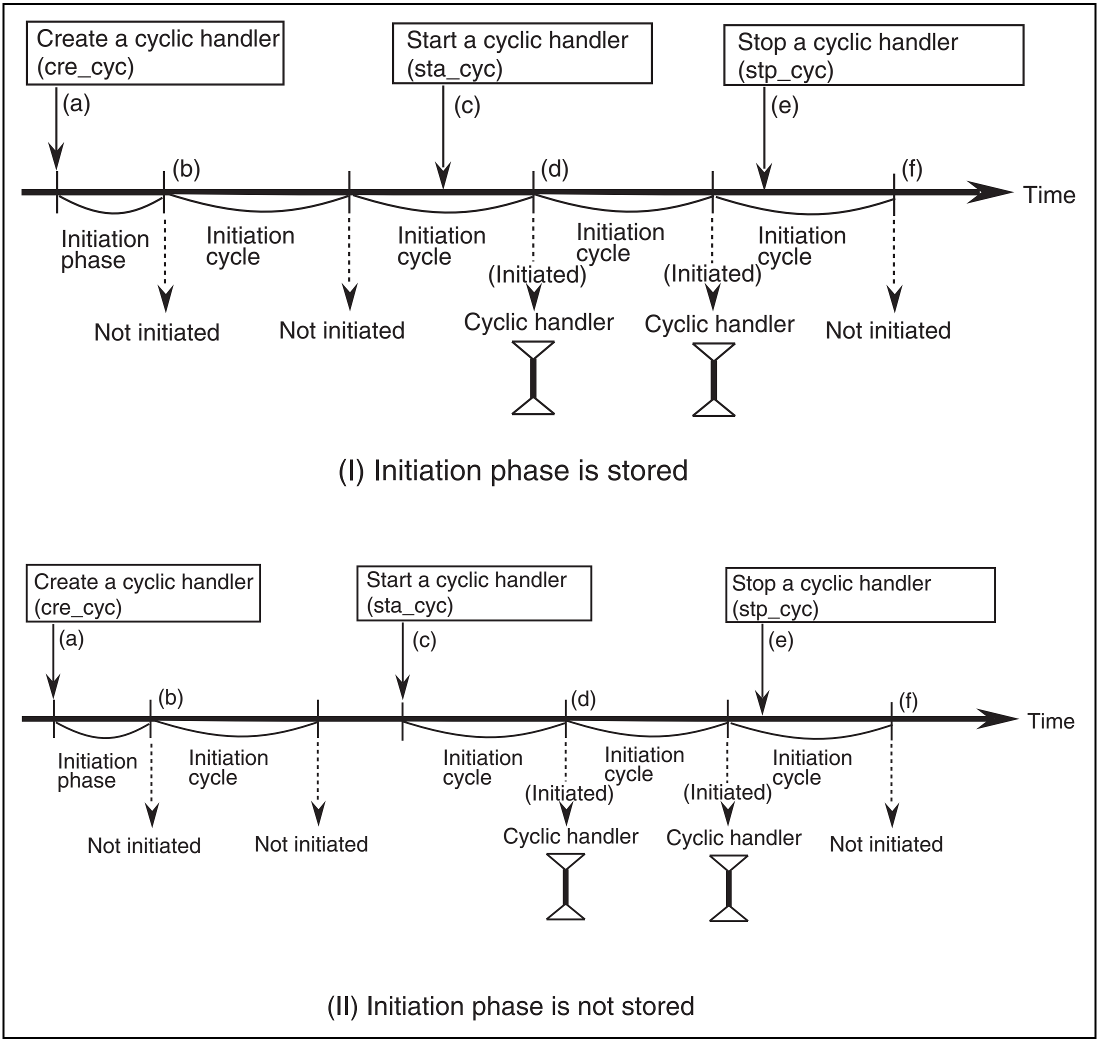


Figure 5.21 Examples of Cyclic Handler Operation

Extended information specified at the time of creation is passed to the cyclic handler. Cyclic handlers are controlled by the service calls listed below.

(1) Create Cyclic Handler (cre_cyc, icre_cyc)

Creates a cyclic handler with the specified ID.

(2) Create Cyclic Handler (acre_cyc, iacre_cyc)

Creates a cyclic handler with an ID that is automatically assigned by the kernel and returned.

(3) Delete Cyclic Handler (del_cyc)

Deletes a cyclic handler.

(4) Start Cyclic Handler (sta_cyc, ista_cyc)

Initiates a cyclic handler.

sta_cyc can be issued for a cyclic handler of the other CPU.

(5) Stop Cyclic Handler (stp_cyc, istp_cyc)

Stops a cyclic handler.

stp_cyc can be issued for a cyclic handler of the other CPU.

(6) Reference Cyclic Handler State (ref_cyc, iref_cyc)

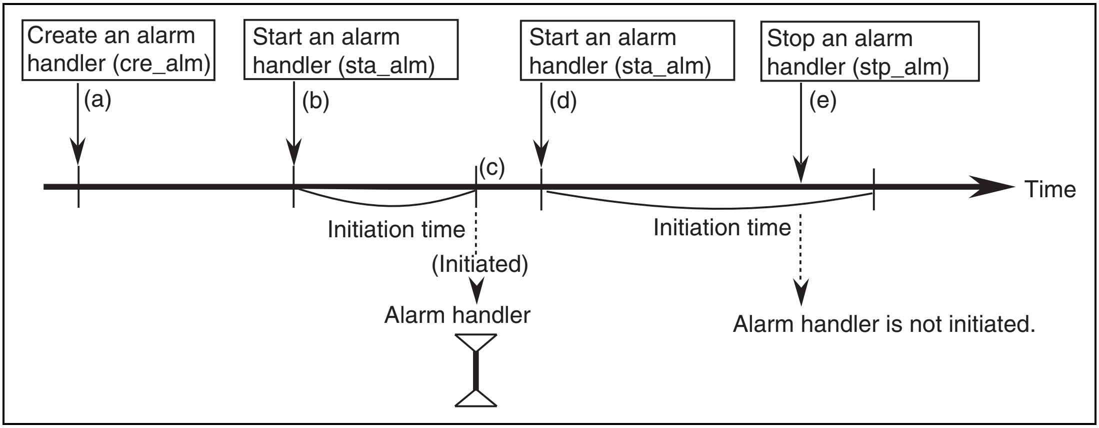
Refers to the operating state of the cyclic handler, including the time left until the cyclic handler is initiated.

ref_cyc can be issued for a cyclic handler of the other CPU.

5.13.5 Alarm Handler

An alarm handler is a time-event handler that is initiated once when the specified time is reached. Alarm handlers can be used to make processing run on a timetable.

Figure 5.22 shows an example of alarm handler operation.



Create an alarm handler (cre_alm)
Start an alarm handler (sta_alm)
Start an alarm handler (sta_alm)
Stop an alarm handler (stp_alm)
(a)
(b)
(c)
(d)
(e)
Time
Initiation time
(Initiated)
Alarm handler
Initiation time
Alarm handler is not initiated.

Figure 5.22 Example of Alarm Handler Operation

Extended information specified at the time of creation is passed to the alarm handler. Alarm handlers are controlled by the service calls listed below.

(1) Create Alarm Handler (cre_alm, icre_alm)

Creates an alarm handler with the specified ID.

(2) Create Alarm Handler (acre_alm, iacre_alm)

Creates an alarm handler with an ID that is automatically assigned by the kernel and returned.

(3) Delete Alarm Handler (del_alm)

Deletes an alarm handler.

(4) Start Alarm Handler (sta_alm, ista_alm)

Initiates an alarm handler after the specified time has elapsed.

sta_alm can be issued for an alarm handler of the other CPU.

(5) Stop Alarm Handler (stp_alm, istp_alm)

Stops an alarm handler.

stp_alm can be issued for an alarm handler of the other CPU.

(6) Reference Alarm Handler State (ref_alm, iref_alm)

Refers to the operating status of the alarm handler and the time left until the alarm handler is initiated.

ref_alm can be issued to an alarm handler of the other CPU.

5.13.6 Overrun Handler

The overrun handler is a time-event handler. The processor time limit can be set for each of the tasks. When a task uses the processor for a longer time than the limit, the overrun handler is started. Only one overrun handler can be defined in a single system.

Figure 5.23 shows an example of overrun handler operation.

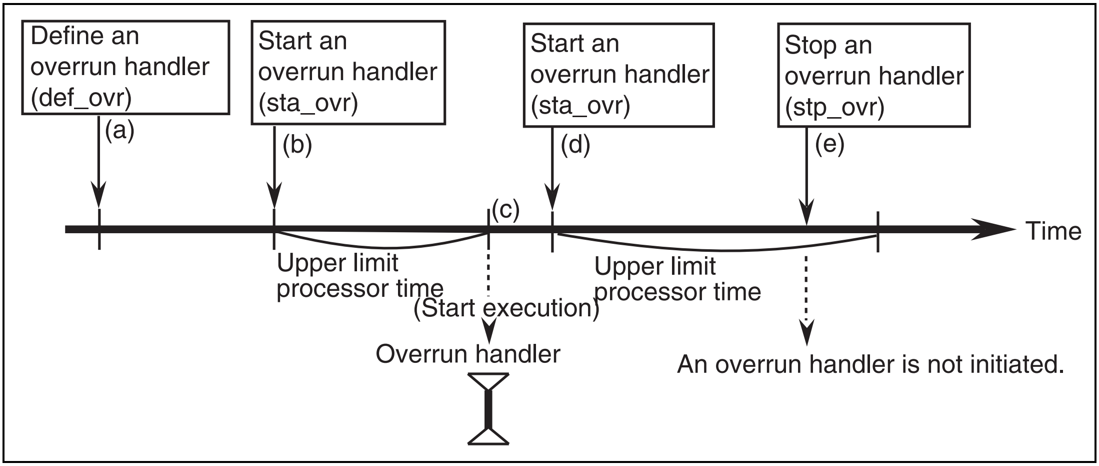

Define an overrun handler (def_ovr)
Start an overrun handler (sta_ovr)
Start an overrun handler (sta_ovr)
Stop an overrun handler (stp_ovr)
(a)
(b)
(c)
(d)
(e)
Time
Upper limit processor time
Upper limit processor time
(Start execution)
Overrun handler
An overrun handler is not initiated.

Figure 5.23 Example of Overrun Handler Operation

IDs and extended information of tasks for overrun handling are passed to the overrun handler. The overrun handler is controlled by the service calls listed below.

(1) Define Overrun Handler (def_ovr)

Defines an overrun handler.

(2) Start Overrun Monitoring of Task (sta_ovr, ista_ovr)

Sets an upper time limit for a specified task. When the task is executed for more than the specified time, the overrun handler is initiated.

sta_ovr can be issued for a task of the other CPU.

(3) Stop Overrun Monitoring of Task (stp_ovr, istp_ovr)

Stops the overrun monitoring of a specified task.

stp_ovr can be issued for a task of the other CPU.

(4) Reference Overrun Monitoring Status of Task (ref_ovr, iref_ovr)

Refers to the status of overrun monitoring for a specified task, including the time left until the upper limit.

ref_ovr can be issued for a task of the other CPU.

5.13.7 Time Precision

The unit of time used for setting time parameters, such as a timeout period, is 1 ms, but the precision of time is TIC_NUME/TIC_DENO [ms]. This precision applies to updating of the system clock and time management. TIC_NUME and TIC_DENO are respectively defined in system.tic_nume and system.tic_deno of the cfg file.

A time event (timeout occurrence or cyclic handler initiation) is generated after the specified time has passed.

Figure 5.24 shows examples of tslp_tsk(5) execution when the actual time is 9.2 ms.

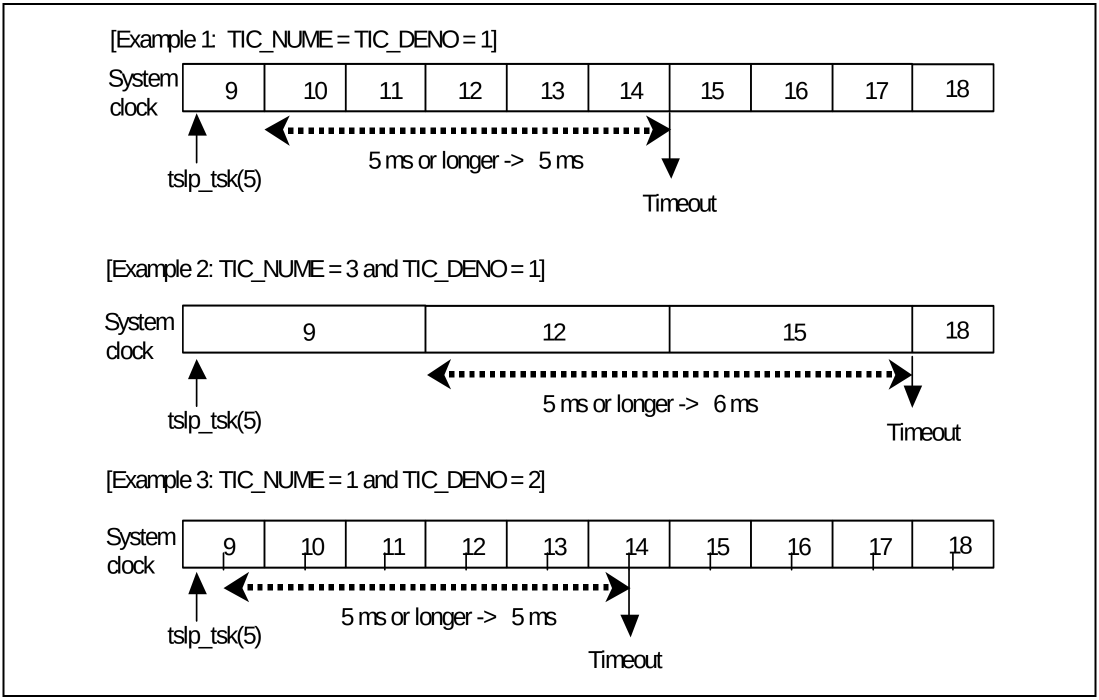


Figure 5.24 Time Precision (tslp_tsk)

The timing of initiation of a cyclic handler is as described below.

(1) Cyclic handler for which the TA_PHS attribute has not been specified

(a) Operation started by sta_cyc or ista_cyc

From the time of the sta_cyc or ista_cyc call, the timing of the nth round of handler initiation is the value of the following expression.

```
(Initiation cycle) × n
```

(b) Operation started by specifying the TA_STA attribute at the time of handler creation

From the time of creation, the timing of the nth round of handler initiation is the value obtained from the following expression.

```
(Initiation phase) + (Initiation cycle) × (n – 1)
```

(2) Cyclic handler for which the TA_PHS attribute has been specified

Handling is the same as case (b) under (1). However, whether or not the handler is actually initiated depends on its operational state at the given time on each cycle.

Figure 5.25 shows examples of the timing of a cyclic handler when the call to start it (sta_cyc) is made at an actual time of 9.5 ms and the initiation cycle is three. Note that when the initiation cycle is less than TIC_NUME/TIC_DENO (as in example 2), there may be cases where the initiation handler is initiated two or more times with the same timing.

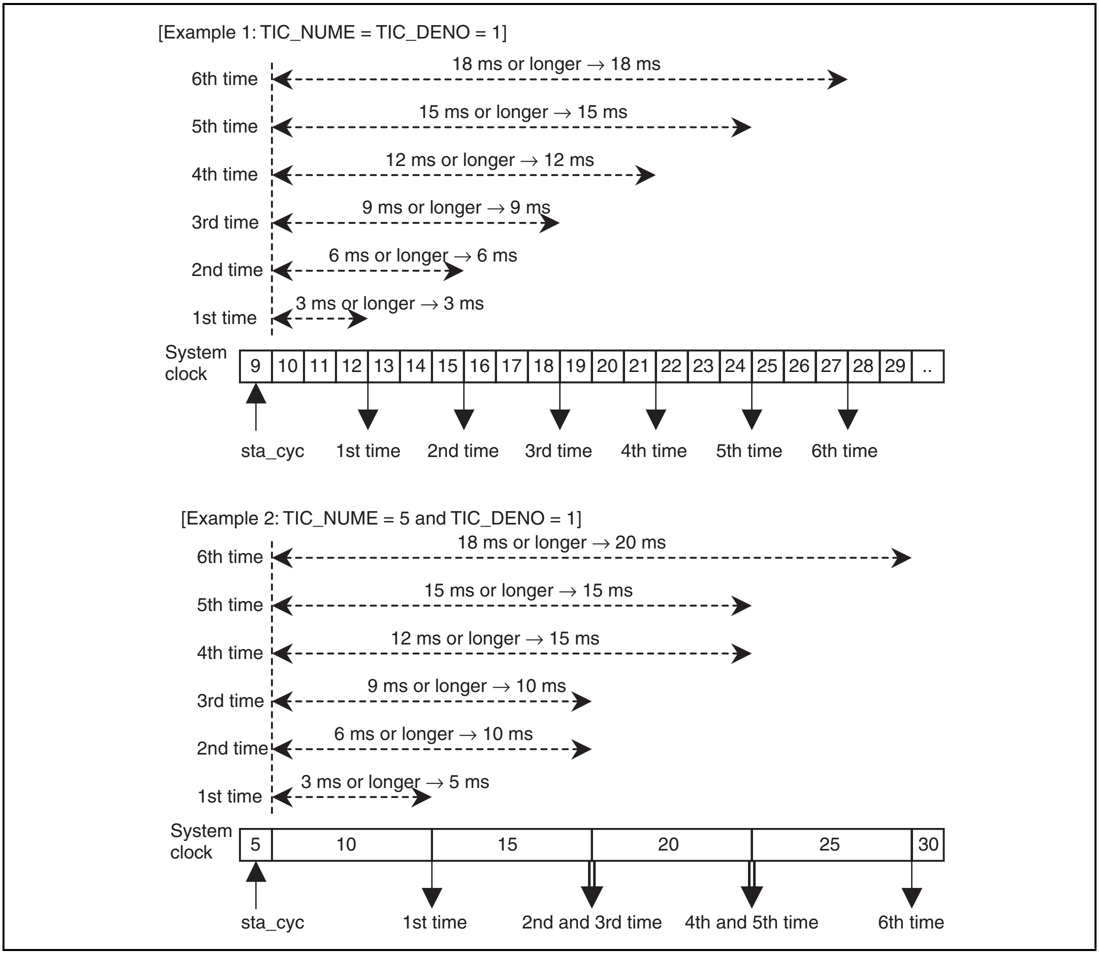


Figure 5.25 Time Precision (sta_cyc)

5.13.8 Notes on Time Management

The kernel performs the following processing when a timer interrupt occurs.

(a) Updates the system clock.
(b) Initiates and executes alarm handlers.
(c) Initiates and executes cyclic handlers.
(d) Initiates and executes the overrun handler.
(e) Performs task timeout processing specified by service calls with the timeout function and dly_tsk.

These processes are all performed with the timer interrupt level or lower interrupt levels masked. Among these processes, (b), (c), and (e) 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 the system clock

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

- The time for time event handler processing must be as short as possible.
- The time event handler cycles and the timeout values specified by timeout service calls must be set to the largest possible values. As an extreme example, if the cycle time of a cyclic handler is 1 ms and the handler's processing takes longer than 1 ms, that cyclic handler will be executed forever; and the system will hang.

5.14 System State Management

5.14.1 Managing System State

(1) Rotate Ready Queue (rot_rdq, irot_rdq)

This service call establishes the time-sharing system (TSS). That is, rotating the ready queue at regular intervals accomplishes the round-robin scheduling required for the TSS.

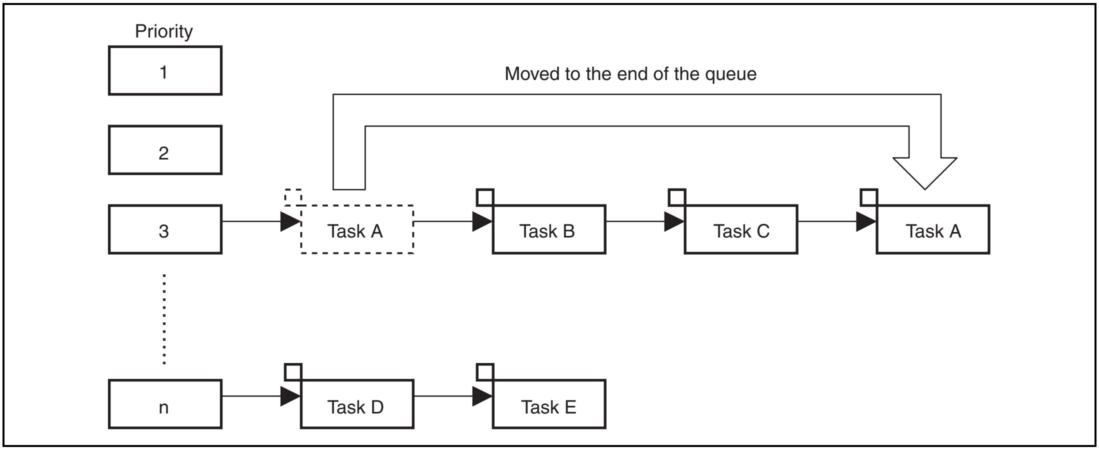


Figure 5.26 Management of the Ready Queue by rot_rdq

(2) Get Current Task ID (get_tid, iget_tid)

get_tid refers to the ID of the task in the RUNNING state. When iget_tid is issued in a non-task context, the ID of the task running at the time is acquired.

(3) Lock CPU (loc_cpu, iloc_cpu) and Unlock CPU (unl_cpu, iunl_cpu)

loc_cpu or iloc_cpu makes the system enter the CPU-locked state. To subsequently leave the CPU-locked state, issue unl_cpu or iunl_cpu.

(4) Disable Dispatch (dis_dsp) and Enable Dispatch (ena_dsp)

dis_dsp makes the system enter the dispatch-disabled state. To subsequently leave the dispatch-disabled state, issue ena_dsp.

(5) Check Context (sns_ctx)

Checks whether the system is in a task or non-task context.

(6) Check CPU-Locked State (sns_loc)

Checks if the system is in the CPU-locked state.

(7) Check Dispatch-Disabled State (sns_dsp)

Checks if the system is in the dispatch-disabled state.

(8) Check Dispatch-Pending State (sns_dpn)

Checks if the system is in the dispatch-pending state. The dispatch-pending state means that processing with a higher priority than the dispatcher is in progress so that no other task can be executed. To be more specific, each of the following cases corresponds to the dispatch-pending state.

- CPU-locked state
- Dispatch-disabled state
- Non-task context
- Execution of normal CPU exception handler
- IMASK level in SR is not 0

Unless the system is in the dispatch-pending state, all service calls to make a task enter the WAITING state are available. When developing software (e.g., middleware) that may be invoked from any system state, this service call (sns_dpn) is useful for checking the current system state to see whether a service call that makes a task enter the WAITING state can be processed or will lead to an error being returned.

5.14.2 Service Calls Associated with Initialization

(1) Start Kernel (vsta_knl, ivsta_knl)

Initiates the kernel according to the results of configuration.

(2) Initialize Remote Service-Call Environment (vini_rmt)

Initializes the remote service-call environment according to the result of configuration.

5.14.3 System Down (vsys_dwn, ivsys_dwn)

vsys_dwn makes the system go down and initiates the system-down routine.

5.14.4 Service Call Trace

The service call trace function acquires the history of service calls. The acquired trace information can be displayed by using the debugging extension.

Define whether or not the trace function is to be used as system.trace in the cfg file.

For details on the trace function, refer to the help information for the debugging extension.

(1) Trace Timing and Information to be Acquired

Trace information is acquired with the following timing.

- On issuing of and return from service calls
- Initiation and completion of tasks and task exception handling routines
- Transitions to the kernel-idling state

The following set of information is acquired.

- Type of service call
- Parameters for the service call
- Error codes for the service call
- Value of the program counter (PC)
- Trace serial number

A trace serial number is a serial number for a set of acquired trace information which is common to both CPUs.

(2) Trace Type

The trace information can be stored either in a buffer allocated to the RAM on the target system or in the trace memory of the simulator or emulator, as selected by the system.trace setting in the cfg file. The former is called a target trace and the latter is called a tool trace. For a target trace, specify the buffer size as system.trace_buffer.

Although environments where a tool trace is available are limited to those including a simulator or emulator, a tool trace does not require the buffer for tracing on the target system.

(3) Number of Objects

In the debugging extension, the state of the objects specified by the user can also be acquired with the trace timing. The maximum number of objects that can be traced at one time should be specified as system.trace_object in the cfg file.

(4) User Event Trace (vget_trc, ivget_trc)

Use vget_trc or ivget_trc to acquire any user-specified information with the user-specified timing.

(5) Tracing of Start and End of Interrupt Handlers or CPU Exception Handlers (ivbgn_int, ivend_int)

By default, trace information on the start and end of interrupt handlers and CPU exception handlers is not acquired. To make the debugging extension show the execution history of a handler, call ivbgn_int and ivend_int at the start and end of handler execution, respectively. A vector number should also be specified for these service calls as information to identify the handler.

(6) Note on Service Call Tracing

a. Degradation of performance

   When the service call tracing functions are used, the performance of the kernel is degraded.

b. Service call information not traced

   Service calls for non-task contexts of the form ixxx_yyy are all acquired as if they were service calls for task contexts of the form xxx_yyy.

   The following service calls cannot be traced.

   — cal_svc, ical_svc

   — vsta_knl, ivsta_knl

   — vsys_dwn, ivsys_dwn

c. Trace serial number

   When system.trace!=NO, the trace serial numbers may be inconsistent. To avoid this problem, call vsta_knl on CPUID#2 after starting the initialization routine or the first task on CPUID#1.

5.15 Interrupt Management

When an interrupt is generated, the corresponding interrupt handler is initiated. Interrupt handlers should be defined through interrupt_vector[] or by issuing a def_inh or idef_inh service call. Also refer to section 4.8, Interrupts.

(1) Define Interrupt Handler (def_inh, idef_inh)

Defines an interrupt handler for a specified vector number.

(2) Change Interrupt Mask Level (chg_ims, ichg_ims)

Changes the interrupt mask level (the IMASK bits in register SR) to a specified value.

(3) Reference Interrupt Mask Level (get_ims, iget_ims)

Refers to the current mask level (the IMASK bits in register SR).

5.16 Extended Service Calls

A service call processing routine can be created and defined in the kernel as an extended service call routine. An application can call an extended service call routine without being linked to the routine.

Each extended service call is identified by a positive function code. The maximum value for use as a function code should be defined as maxdefine.max_fncd in the cfg file.

Extended service calls are controlled by the service calls listed below.

(1) Define Extended Service-Call Routine (def_svc, idef_svc)

Defines an extended service-call routine with the specified function code.

(2) Call Extended Service-Call Routine (cal_svc, ical_svc)

Issues the extended service call with the specified function code. This initiates the corresponding extended service-call routine. In cal_svc or ical_svc, up to four 32-bit integers can be specified as parameters for passing to the extended service-call routine.

5.17 System Configuration Management

(1) Define CPU Exception Handler (def_exc, idef_exc)

Defines a CPU exception handler for a specified vector number.

(2) Define CPU Exception (TRAPA-Instruction Exception) Handler (vdef_trp, idef_trp)

Defines a CPU exception handler for a specified trap number.

(3) Reference Configuration Information (ref_cfg, iref_cfg)

Refers to configuration information such as the maximum local object ID for objects.

(4) Reference Version Information (ref_ver, iref_ver)

Refers to the version numbers of the kernel and the µITRON specification implemented in the kernel. The information acquired by ref_ver or iref_ver can also be acquired through kernel configuration macros (refer to section 6.31.2, Kernel Configuration Macros).

5.18 Profile Management

The profile management function samples the running task at a specified interval to give the user statistics on the execution ratio of tasks. This function is not defined in the µITRON4.0 specification.

Respective 32-bit profile counters are provided for overall time, each task, and kernel idling. The timer-interrupt processing for the kernel executed in cycles of TIC_NUME/TIC_DENO milliseconds governs updating of these counters. That is, incrementation of the individual profile counters for the running task or kernel idling, and that for the overall time, proceeds with this cycle (figure 5.27).

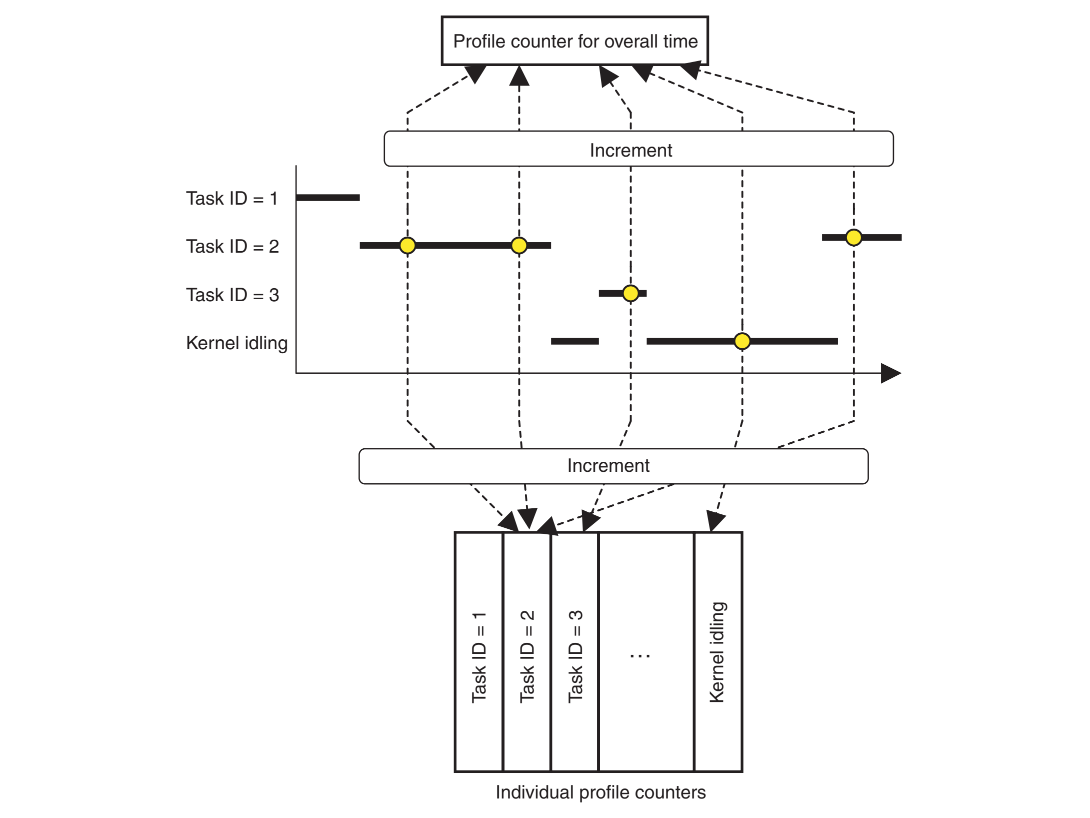


Figure 5.27 Profile Management

Although the results are not exact, a long period of measurement gives approximate execution times for tasks as calculated by the formula below.

Execution time (ms) =
Value of an individual profile counter for a task × (TIC_NUME/TIC_DENO)

In addition, the CPU usage of a task or of kernel idling can be estimated by dividing the value of the corresponding profile counter by the value of the profile counter for overall time.

Issuing vsta_knl initializes all profile counters to 0. Individual profile counters for tasks are also initialized when the corresponding tasks are deleted.

Also note that the kernel does not detect any overflow of the profile counters. For example, the counters will start to overflow in 50 days when TIC_NUME/TIC_DENO = 1 ms, and in 12 hours or so when TIC_NUME/TIC_DENO = 10 µs.

The HI7200/MP offers the following profile management function service calls.

(1) Reference Profile Counter (vref_prf, ivref_prf)

Refers to the value of a specified profile counter.

vref_prf can be issued for a task of the other CPU.

(2) Clear Profile Counter (vclr_prf, ivclr_prf)

Clears a specified profile counter.

vclr_prf can be issued for a task of the other CPU.

5.19 Kernel Idling

When there is no READY task, the kernel enters an endless loop and waits for interrupts.

The lowest-priority task is usually used to make transitions to low power consumption modes of the CPU.

Section 6 Kernel Service Calls

6.1 Calling Form

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

```
ercd = slp_tsk();
```

6.2 Header Files

Programs that issue service calls should include kernel.h which is located under <RTOS_INST>\os\include\.

kernel.h includes the files listed in table 6.1.

Table 6.1 Files Included in kernel.h

Directory	File Name	Description
<RTOS_INST>\os\include\	itron.h	Definitions for the ITRON specification common regulations.
	kernel_api.h	Definitions for the kernel service calls.
Specified by user	kernel_intspec.h	Definitions for the hardware specifications related to interrupts. For details, refer to section 17.3, Creating CPU Interrupt Specification Definition File (kernel_intspec.h).
Output by cfg72mp	kernel_macro.h	Definitions for the kernel configuration constants which are determined at configuration.
	mycpuid.h	Definitions for MYCPUID (current CPU ID).

6.3 Basic Data Types

The basic data types are shown in table 6.2. These basic data types are defined on the basis of types.h.

For types.h, refer to section 19, types.h.

Table 6.2 Basic Data Types

No.	Data Type	Meaning
1	B	8-bit signed integer
2	H	16-bit signed integer
3	W	32-bit signed integer
4	D	64-bit signed integer
5	UB	8-bit unsigned integer
6	UH	16-bit unsigned integer
7	UW	32-bit unsigned integer
8	UD	64-bit unsigned integer
9	VB	8-bit signed integer*
10	VH	16-bit signed integer*
11	VW	32-bit signed integer*
12	VD	64-bit signed integer*
13	VP	Pointer to void type function
14	FP	Pointer to void type function
15	INT	Signed integer (signed int)
16	UINT	Unsigned integer (unsigned int)
17	BOOL	Signed integer (signed int)
18	FN	32-bit signed integer
19	ER	32-bit signed integer
20	ID	16-bit signed integer
21	ATR	32-bit unsigned integer
22	STAT	32-bit unsigned integer
23	MODE	32-bit unsigned integer
24	PRI	16-bit signed integer
25	SIZE	32-bit unsigned integer
26	TMO	32-bit signed integer
27	RELTIM	32-bit unsigned integer
28	SYSTIM	A structure which contains the following members: Upper: 16-bit unsigned integer Lower: 32-bit unsigned integer
29	VP_INT	32-bit signed integer*
30	ER_BOOL	32-bit signed integer
31	ER_ID	32-bit signed integer
32	ER_UINT	32-bit signed integer
33	TEXPTN	32-bit unsigned integer
34	FLGPTN	32-bit unsigned integer
35	RDVPTN	32-bit unsigned integer
36	RDVNO	32-bit unsigned integer
37	OVRTIM	32-bit unsigned integer
38	INHNO	32-bit unsigned integer
39	EXCNO	32-bit unsigned integer
40	IMASK	32-bit unsigned integer

Note: * When the variable values of these data types are referred to or substituted, the type must be explicitly converted (casted).

6.4 Register Contents Guaranteed after Issuing Service Call

Some registers guarantee the contents after a service call is issued but some do not. This rule follows the Renesas C compiler. The details are shown in table 6.3.

Table 6.3 Register Contents Guaranteed after Issuing Service Call

Register	Register State after Service Call Return
SR, R8 to R15, PR, GBR, MACH, MACL	The register contents are guaranteed. IMASK bits in SR are updated when service call chg_ims, ichg_ims, loc_cpu, iloc_cpu, unl_cpu, or iunl_cpu is issued.
R0	Normal end (E_OK) or an error code is set.
R1 to R7	The register contents are guaranteed only when they are specified as return parameters.
TBR	The register contents are guaranteed when system.tbr is set as FOR_SVC (use for only service call) or TASK_CONTEXT (task context).
For SH2A-FPU: FR0 to FR11	The register contents are not guaranteed.
For SH2A-FPU: FPSCR, FPUL, FR12 to FR15	The register contents are guaranteed only when a service call is issued in either one of the following states. • From a task with the TA_COP1 attribute or a task exception handling routine • In dispatch-pending state

6.5 Return Value of Service Call and Error Code

6.5.1 Overview

For service calls that have return values, a positive value or 0 (E_OK) indicates a normal end, and a negative value indicates an error code. The meaning of the return value at a normal end differs according to the service call; however, only E_OK is returned at a normal end for many service calls.

However, for service calls that have a BOOL-type return value, this is not the case.

6.5.2 Parameter Check Function

In this kernel, detection of parameter errors can be omitted. If the parameter check function is omitted after debugging is completed, the overhead and code size can be reduced.

To omit the parameter check function, define NO for system.parameter_check.

6.5.3 Stack Overflow Detection

Stack overflow is a failure that is hard to determine where it has occurred. This function is a debugging function to assist in determining the location.

For only a service call issued in a task context, this function inspects whether the stack has overflowed when the service call has been issued. If the stack has overflowed, the system goes down.

However, this inspection is not performed for some service calls. The information on which service call is inspected is not released because it depends on the kernel version.

Note that this function inspects whether the stack has overflowed at only the point at which the service call was issued. Stack overflow in the application is not detected. For example, in a case where the number of nestings of the stack becomes greater than that when the service call was issued on the application side.

This function is always enabled. (There are no settings related to this function in the cfg file.)

6.5.4 Main Error Code and Suberror Code

An error code consists of a main error code (lower 8 bits) and a suberror code (the remaining upper bits). The suberror code of all error codes returned by this kernel is always –1.

The following macros are defined in standard header itron.h.

- ER MERCD(ER ercd);   Extracts the main error code from the error code.
- ER SERCD(ER ercd);   Extracts the suberror code from the error code.
- ER ERCD(ER mercd, ER sercd);   Generates the error code using the main error code and suberror code.

6.6 System State and Service Calls

Whether a service call can be issued depends on the system state.

6.6.1 Task Contexts and Non-Task Contexts

(1) Special Service Calls

The following service calls can be issued in either task contexts or non-task contexts.

- vsta_knl, ivsta_knl
- vsys_dwn, ivsys_dwn

(2) Service Calls Starting with sns or vsns

The service calls whose names start with "sns" or "vsns" can be issued in either task contexts or non-task contexts.

(3) Other Service Calls

The service calls whose names start with "i" are dedicated to non-task contexts, and the other service calls are for task contexts.

The service calls for task contexts are further classified into the following two types.

(a) Service calls for which no corresponding service calls starting with "i" are provided
(e.g. del_tsk; there is no idel_tsk)

If this type of service call is issued in a non-task context, an E_CTX error will be returned.

(b) Service calls for which corresponding service calls starting with "i" are provided
(e.g. act_tsk and iact_tsk)

In the kernel, the processing for a service call with "i" is the same as that for the corresponding service call without "i". Accordingly, when a service call starting with "i" is issued in a task context or when a service call without "i" is issued in a non-task context, no E_CTX error will be detected and the service call is processed correctly.

Note that this behavior is only for this version of kernel implementation, and it may change in a later version of the kernel.

To improve the portability of an application, programming is recommended to be done following the rule that the service calls starting with "i" are for non-task contexts and the other service calls are for task contexts.

6.6.2 CPU-Locked State

Service calls that can be issued in the CPU-locked state are listed below. No E_CTX error is detected when a service call other than these is issued in the CPU-locked state. In this case, correct system operation cannot be guaranteed. Note that, when a service call that shifts a task to the WAITING state is issued, an E_CTX error is detected.

- ext_tsk (CPU-locked state will be canceled)
- exd_tsk (CPU-locked state will be canceled)
- sns_tex
- loc_cpu, iloc_cpu
- unl_cpu, iunl_cpu
- sns_ctx
- sns_loc
- sns_dsp
- sns_dpn
- vsta_knl, ivsta_knl
- vsys_dwn, ivsys_dwn
- vsns_tmr

6.6.3 Dispatch-Disabled State

When a service call that shifts a task to the WAITING state is issued in this state, an E_CTX error is returned.

6.6.4 Normal CPU Exception Handler

The service calls that can be issued from the normal CPU exception handler are listed below.

- iras_tex
- sns_tex
- sns_loc
- sns_dsp
- sns_dpn
- get_tid, iget_tid
- vsta_knl, ivsta_knl
- vsys_dwn, ivsys_dwn
- vsns_tmr

No E_CTX error will be detected when a service call other than these is issued from the normal CPU exception handler. In this case, correct operation cannot be guaranteed.

6.6.5 When SR.IMASK is Changed to a Non-Zero Value in a Task Context

This state is handled as a non-task context.

6.7 ID Number

6.7.1 Overview

Software components operated by service calls, such as tasks and semaphores, are referred to as "objects". ID numbers are used to identify the objects. An ID number is expressed as a 16-bit signed integer.

An independent ID number space is prepared for each object type. The ID number space for each object type is one plane for all CPUs.

The ID number consists of the CPU ID and local object ID, as shown in figure 6.1.

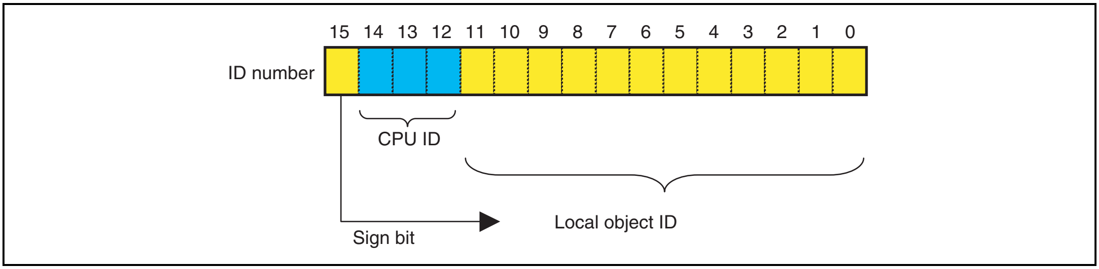


Figure 6.1 ID Number

(1) CPU ID

The CPU ID is assigned to each CPU starting from number 1.

In the SH2A-DUAL, the CPU ID of CPU#0 is determined as 1 and the CPU ID of CPU#1 is determined as 2. If a CPU ID other than that of the current CPU is specified for the object to be operated in a service call, that service call is handled as a "remote service call".

The same CPU as the caller of the service call can be specified by using VCPU_SELF. VCPU_SELF is a macro defined as 0 in kernel.h and is not defined in the µITRON4.0 specification.

The CPU ID is handled as unsigned data.

(2) Local Object ID

The local object ID ranges between 1 and _MAX_???. _MAX_??? is the "maximum ID" among the objects output to kernel_cfg.h by cfg72mp. For details, refer to section 6.31.2 (8), Kernel Configuration Macros Output to kernel_cfg.h by cfg72mp (not in the µITRON4.0 Specification).

Bit 15 in the ID number serves as the sign bit of the local object ID. A negative local object ID is used for making special specifications in some of the service calls.

6.7.2 Function Macros Related to ID Number

The following macros are prepared for facilitating use of the ID number, CPU ID, and local object ID. These macros are defined in kernel.h.

These macros are not defined in the µITRON4.0 specification.

• ID GET_CPUID(ID id)	Returns CPU ID of id
• ID GET_LOCALID(ID id)	Returns local object ID of id
• ID MAKE_ID(ID cpuid, ID localid)	Returns ID number consisting of cpuid and localid

6.8 Behavior of Service Calls

6.8.1 Remote Service Call and Local Service Call

In a service call that has the object ID to be operated as a parameter, the CPU that contains the object to be operated is specified by the CPU ID of that object ID. However, specification of another CPU is not permitted in some of the service calls.

If the object ID of another CPU is specified, the service call means a request to the kernel of another CPU and is referred to as a "remote service call".

If the object ID of the current CPU is specified or if the service call does not have the object ID as a parameter, the service call means a request to the kernel of the current CPU and is referred to as a "local service call".

The kernel structure is shown in figure 6.2.

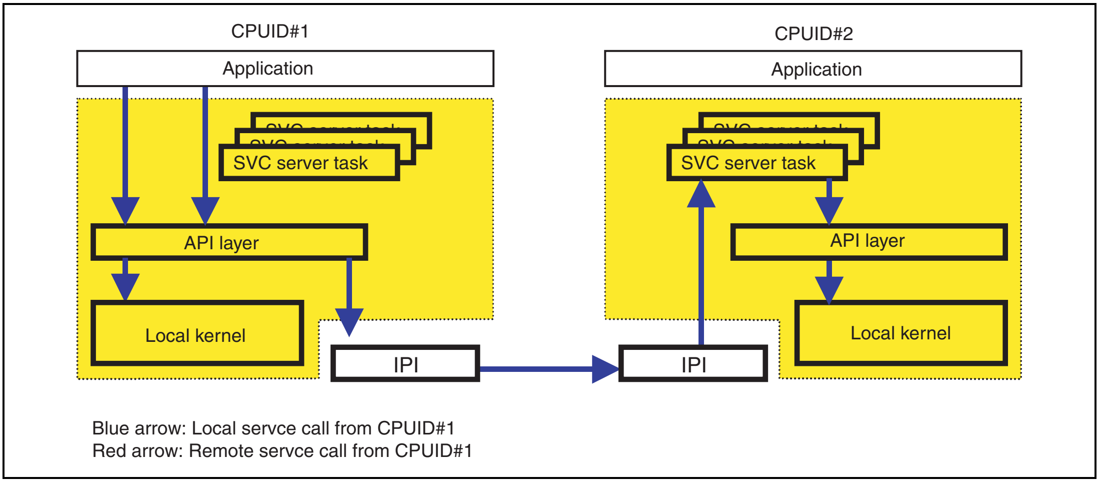


Figure 6.2 Kernel Structure

The region in yellow is the kernel of each CPU.

The local kernel is a module almost equivalent to the µITRON4.0 specification kernel for a conventional single CPU. It manages the tasks and objects in the current CPU and has the task scheduling function.

The API layer is an interface layer between the local kernel and application. When a service call is issued, first the API layer is executed. The API layer is executed in the same way as a normal function. Thus, the API layer operates in the same state as the caller (context type or stack), and task switching may be performed during processing.

The SVC server tasks accept remote service-call requests from another CPU and perform the role of issuing the service calls to the kernel of the current CPU instead of the calling tasks.

6.8.2 Behavior of Local Service Call

The API layer calls the local kernel of the current CPU. On returning from the local kernel, control will return to the caller after the return value has been set.

6.8.3 Behavior of Remote Service Call

The following definitions related to remote service calls are made in the cfg file of each CPU. For details, refer to section 14.3.10, Defining the Remote Service-Call Environment (remote_svc).

- Number of SVC server tasks (remote_svc.num_server)
- Priority of SVC server tasks (remote_svc.priority)
- Stack size used by SVC server tasks (remote_svc.stack_size)
- IPI port in use (remote_svc.ipi_portid)
- Maximum number of tasks waiting for an available SVC server task (remote_svc.num_wait)

First, in service call vini_rmt that initializes the remote service-call environment, SVC server tasks for the number of num_server are created and initiated using service call acre_tsk. SVC server tasks are then shifted to the WAITING state by service call slp_tsk until remote service-call requests are sent from another CPU.

A remote service call is processed using the following procedure.

1. When a remote service call is issued, the API layer attempts to use an available SVC server task of the target CPU. However, if vini_rmt in the current CPU or target CPU has not completed, an EV_NOINIT error will be returned immediately.

   If there are no available SVC server tasks in the target CPU, the calling task is shifted to the WAITING state by slp_tsk in the API layer until an SVC server task becomes available. However, if remote_svc.num_wait in the current CPU is set to 0, an EV_NORESOURCE error will be returned immediately.

   Before shifting the task to the WAITING state, memory is allocated using pget_mpf in order to manage the WAITING state. This fixed-sized memory pool is created using acre_mpf in vini_rmt. The number of memory blocks is specified by remote_svc.num_wait. When pget_mpf fails because there is no free memory block in the memory pool or for other reasons, an EV_NORESOURCE error will be returned to the caller.
2. After becoming able to use an SVC server task, the API layer then wakes up that SVC server task in the target CPU by using the IPI. The IPI port used here is remote_svc.ipi_portid in the target CPU. The calling task is shifted to the WAITING state by slp_tsk and remains in this state in the API layer until SVC server task processing has completed.

3. The woken up SVC server task in the target CPU issues the service call (local service call) requested to the kernel of the current CPU (target CPU when seen from the caller of the service call). The behavior at this point is the same as that in section 6.8.2, Behavior of Local Service Call. When returning from this service call, the SVC server task sets the return value and notifies the calling CPU that processing has completed through the IPI. The IPI port used here is remote_svc.ipi_portid in the calling CPU.
4. The requesting task that was kept in the WAITING state in the API layer is woken up. The API layer sets the return value and returns from the service call.

6.8.4 Notes on Remote Service Call

- When a remote service call is issued, the wakeup request count for the calling task must be 0.
- A service call that will change the task state, such as ter_tsk, rel_wai, and sus_tsk, must not be issued for an SVC server task or a task in the middle of calling a remote service call.
- In the case of a remote service call with a timeout setting, timeout will be specified in the service calls issued by SVC server tasks. The remote service call is sometimes shifted to the WAITING state by slp_tsk (no timeout specification) before it has been processed by the SVC server task as a client processing, as shown in section 6.8.3, Behavior of Remote Service Call.

6.9 Service Calls not in the µITRON4.0 Specification

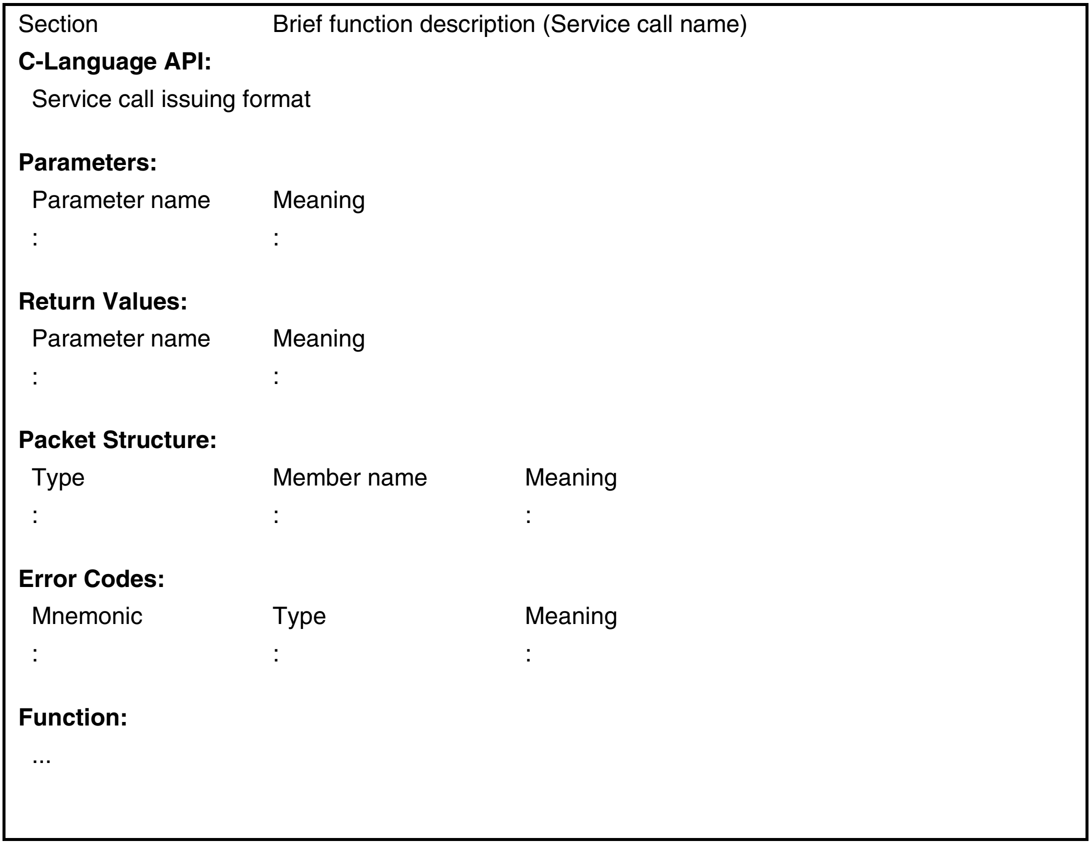
The service-call names starting with “v”, “iv”, or “V”, such as vset_tfl, are not defined in the µITRON4.0 specification.

The following "ixxx_yyy"-format service calls (starting with "i'") are also not defined in the µITRON4.0 specification. They are provided to enable the "xxx_yyy"-format service calls corresponding to the following service calls to be issued in non-task contexts because the "xxx_yyy"-format service calls are defined to be issued only in task contexts in the µITRON4.0 specification.

icre_tsk, iacre_tsk, ista_tsk, ichg_pri, iget_pri, iref_tsk, iref_tst, isus_tsk, irsm_tsk, frsm_tsk, idef_tex, iref_tex, icre_sem, iacre_sem, ipol_sem, iref_sem, icre_flg, iacre_flg, iclr_flg, ipol_flg, iref_flg, icre_dtq, iacre_dtq, iref_dtq, icre_mbx, iacre_mbx, isnd_mbx, iprcv_mbx, iref_mbx, icre_mbf, iacre_mbf, ipsnd_mbf, iref_mbf, icre_mpf, iacre_mpf, ipget_mpf, iref_mpf, icre_mpl, iacre_mpl, ipget_mpl, iref_mpl, iset_tim, iget_tim, icre_cyc, iacre_cyc, ista_cyc, istp_cyc, iref_cyc, iacre_alm, iacre_alm, ista_alm, istp_alm, iref_alm, ista_ovr, istp_ovr, iref_ovr, idef_inh, ichg_ims, iget_ims, idef_svc, ical_svc, idef_exc, iref_cfg, iref_ver

6.10 Service Call Description Form

Service calls are described in details as shown in figure 6.3 in this section.



Section	Brief function description (Service call name)

C-Language API:

Service call issuing format

Parameters:

Parameter name	Meaning
:	:

Return Values:

Parameter name	Meaning
:	:

Packet Structure:

Type	Member name	Meaning
:	:	:

Error Codes:

Mnemonic	Type	Meaning
:	:	:

Function:

...

Figure 6.3 Service Call Description Form

(1) Error Code

- E_CTX

  In this kernel, detection of an E_CTX error is limited. No E_CTX error will be detected in a service call for which E_CTX is not listed in "Error Code" in each service call description shown later.

- E_NOSPT

  An E_NOSPT error will be returned if an unconfigured service call (service call defined as NO in service_call) is issued.

  An E_NOSPT error will also be returned if a remote service call is issued to a kernel that has been configured with the number of SVC server tasks (remote_svc.num_server) specified as 0.

  This error code is not listed in "Error Code" in each service call description shown later.

- EV_NOINIT or EV_NORESOURCE

  There is a possibility that these errors will be returned when a remote service call is issued. For details, refer to section 6.8.3, Behavior of Remote Service Call.

  These error codes are not listed in "Error Code" in each service call description shown later.

(2) Error Code Type

- [k]: Detected in all states.
- [p]: Detected only when YES is set for system.parameter_check.

6.11 Task Management

Tasks are managed by the service calls listed in table 6.4.

Table 6.4 Service Calls for Task Management

Service Call*[1]	Description	System State*[2]						
		T	N	E	D	U	L	C
cre_tsk [s]	Creates task using non-static stack	O		O	O	O		
icre_tsk			O	O	O	O		
vscr_tsk [s]	Creates task using static stack	O		O	O	O		
ivscr_tsk			O	O	O	O		
acre_tsk	Creates task and assigns task ID automatically	O		O	O	O		
iacre_tsk			O	O	O	O		
del_tsk	Deletes task	O		O	O	O		
act_tsk [S] [R]	Activates task	O		O	Δ	O		
iact_tsk [S]			O	O	O	O		
can_act [S] [R]	Cancels task activation requests	O		O	Δ	O		
ican_act			O	O	O	O		
sta_tsk [B] [R]	Activates task and specifies startup code	O		O	Δ	O		
ista_tsk			O	O	O	O		
ext_tsk [B] [S]	Terminates current task	O		O	O	O	O	
exd_tsk [S]	Terminates and deletes current task	O		O	O	O	O	
ter_tsk [B] [S] [R]	Terminates another task	O		O	Δ	O		
chg_pri [B] [S] [R]	Changes task priority	O		O	Δ	O		
ichg_pri			O	O	O	O		
get_pri [S] [R]	Acquires task priority	O		O	Δ	O		
iget_pri			O	O	O	O		
ref_tsk [R]	Refers to task state	O		O	Δ	O		
iref_tsk			O	O	O	O		
ref_tst [R]	Refers to task state (simple version)	O		O	Δ	O		
iref_tst			O	O	O	O		
vchg_tmd	Changes task execution mode	O		O	O	O		

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
[B]: Basic profile service calls
[R]: Service calls that can be issued remotely

2. T: Can be called in a task context
   N: Can be called in a non-task context
   E: Can be called in dispatch-enabled state
   D: Can be called in dispatch-disabled state
   U: Can be called in CPU-unlocked state
   L: Can be called in CPU-locked state
   C: Can be called while executing the normal CPU exception handler
   O: Can be called in the state
   Δ: Can be called in the state only when a local object is the target

The task management specifications are listed in table 6.5.

Table 6.5 Task Management Specifications

Item	Description
Local task ID	1 to _MAX_TSK (1023 max.)
Task priority	1 to TMAX_TPRI (255 max.)
Maximum activation request count	15
Task attributes	TA_HLNG: Written in a high-level language TA_ASM: Written in assembly language TA_ACT: Task makes a transition to the READY state after it has been created TA_COP1: FPU is used

6.11.1 Create Task (cre_tsk, icre_tsk) (acre_tsk, iacre_tsk: Assign Task ID Automatically) (vscr_tsk, ivscr_tsk) (Using Static Stack)

C-Language API:

```
ER ercd = cre_tsk(ID tskid, T_CTSK *pk_ctsk);
ER ercd = icre_tsk(ID tskid, T_CTSK *pk_ctsk);
ER_ID tskid = acre_tsk(T_CTSK *pk_ctsk);
ER_ID tskid = iacre_tsk(T_CTSK *pk_ctsk);
ER ercd = vscr_tsk(ID tskid, T_CTSK *pk_ctsk);
ER ercd = ivscr_tsk(ID tskid, T_CTSK *pk_ctsk);
```

Parameters:

pk_ctsk Pointer to the packet where the task creation information is stored

<cre_tsk, icre_tsk, vscr_tsk, ivscr_tsk>

tskid Task ID

Return Values:

<cre_tsk, icre_tsk, vscr_tsk, ivscr_tsk>

Normal end (E_OK) or error code

<acre_tsk, iacre_tsk>

Created task ID (a positive value) or error code

Packet Structure:

```
typedef    struct    {
           ATR       tskatr;     Task attribute
           VP_INT    exinf;      Extended information
           FP        task;       Task start address
           PRI       itskpri;    Initial task priority
           SIZE      stksz;      Task stack size
           VP        stk;        Start address of task stack area
}T_CTSK;
```

Error Codes:

E_NOMEM [k] Insufficient memory
(Task stack area cannot be allocated in the memory)

E_RSATR [p] Reserved attribute (tskatr is invalid)

E_PAR [p] Parameter error
(1) stksz is other than a multiple of four, stksz = 0, or stksz ≥ 0x80000000
(2) itskpri ≤ 0 or TMAX_TPRI of current CPU < itskpri

E_ID [p] Invalid ID number
(1) CPU ID is invalid (cre_tsk, icre_tsk, vscr_tsk, ivscr_tsk)
(GET_CPUID (tskid) is not the current CPU)
(2) Out of local ID range
(a) GET_LOCALID (tskid) ≤ 0 or (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid) (cre_tsk, icre_tsk, vscr_tsk, ivscr_tsk)
(b) GET_LOCALID (tskid) ≤ (_MAX_STTSK of GET_CPUID (tskid)) (cre_tsk, icre_tsk)

E_OBJ [k] Object state is invalid (The task specified by tskid is not in the DORMANT state or the current task is specified) (cre_tsk, icre_tsk, vscr_tsk, ivscr_tsk)

E_NOID [k] No ID available (acre_tsk, iacre_tsk)

Function:

Service call cre_tsk, icre_tsk, acre_tsk, or iacre_tsk creates a task that uses the default task stack area or the stack area allocated by the user, while vscr_tsk or ivscr_tsk creates a task that uses a static stack. The created task makes a transition to the DORMANT state when the TA_ACT attribute is not specified, or to the READY state when the TA_ACT attribute is specified.

Each of these service calls can create a task belonging to the kernel of the current CPU. This kernel does not have service calls for creating an object belonging to the kernel of another CPU.

The processing that is performed at task creation is listed in table 6.6.

Table 6.6 Processing to be Performed at Task Creation

Contents
Clears the activation request count.
Resets the task state so that the task exception handling routine is not defined.
Resets the task state so that the processing time limit is not specified.
Assigns a stack (for cre_tsk and acre_tsk).

The following describes the meaning of the parameters.

(1) tskid

Service call vscr_tsk or ivscr_tsk creates a task that uses a static stack. 1 to (_MAX_STTSK of current CPU) can be specified for the local ID of tskid. VCPU_SELF or the current CPU ID must be specified for the CPU ID of tskid.

Service call cre_tsk or icre_tsk creates a task that uses the default task stack area or the stack area allocated by the user. A value from (_MAX_STTSK of current CPU + 1) to (_MAX_TSK of current CPU) can be specified for the local ID of tskid. VCPU_SELF or the current CPU ID must be specified for the CPU ID of tskid.

Service call acre_tsk or iacre_tsk also creates a task that uses the default task stack area or the stack area allocated by the user. However, either service call searches for an unused task ID, creates a task with that ID, and returns the ID to tskid. The range searched for the local task ID is (_MAX_STTSK of current CPU + 1) to (_MAX_TSK of current CPU). The CPU ID of the task ID that will be returned is the current CPU ID.

(2) tskatr

Parameter tskatr specifies the language in which the task is written and the coprocessor to be used as the attributes.

tskatr := ((TA_HLNG || TA_ASM) [|TA_ACT] [|TA_COP1])

- TA_HLNG (0x00000000): Written in a high-level language
- TA_ASM (0x00000001): Written in assembly language
- TA_ACT (0x00000002): Task makes a transition to the READY state after it has been created
- TA_COP1 (0x00000200): FPU is used

When the TA_ACT attribute is specified, extended information (exinf) is passed to the task as a parameter.

The FPU registers can also be guaranteed as task context by specifying the TA_COP1 attribute. Note that the TA_COP1 attribute is not defined in the µITRON4.0 specification.

(3) exinf

Parameter exinf can be widely used by the user, for example, to set information concerning tasks to be created.

(4) task

Specify the task start address.

(5) itskpri

Specify 1 to TMAX_TPRI as the task priority at initiation.

(6) stksz

Parameter stksz is valid only for service calls cre_tsk, icre_tsk, acre_tsk, and iacre_tsk and specifies the stack size of the task to be created. A multiple of four can be specified for the stack size.

Note that stksz has no meaning to service calls vscr_tsk and ivscr_tsk because each service call creates a task that uses the static stack. Parameter stksz is ignored in kernel processing.

(7) stk

Parameter stk is effective in cre_tsk, icre_tsk, acre_tsk, and iacre_tsk service calls.

When NULL is specified for stk, the kernel allocates a stack from the default task stack area. After that, the size of the free space in the default task stack area will decrease by an amount given by the following expression:

Decrease in size = stksz + 16 bytes

The stack area address allocated by application can be specified as stk. In this case, allocate a stack area for the size specified by stksz and specify the start address of the area.

Service calls vscr_tsk and ivscr_tsk are functions not defined in the µITRON4.0 specification.

6.11.2 Delete Task (del_tsk)

C-Language API:

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

Parameters:

tskid	Task ID

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (tskid) is not the current CPU) (2) Out of local ID range (GET_LOCALID (tskid) ≤ 0 or (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid))
E_CTX	[k]	Context error (Called in prohibited system state)
E_NOEXS	[k]	Non-existent object (Task specified by tskid does not exist)
E_OBJ	[k]	Object state is invalid (Task specified by tskid is not in the DORMANT state or the current task is specified)

Function:

Service call del_tsk deletes the task indicated by parameter tskid. The deleted task makes a transition to the NON-EXISTENT state. The profile counter of the deleted task ID is initialized to 0.

Only a task belonging to the kernel of the current CPU can be specified for tskid.

If the stack for the task specified by tskid has been allocated from the default task stack area, the stack for that task is released. After that, the size of the free space in the default task stack area will increase by an amount given by the following expression:

Increase in size = (stksz specified at creation) + 16 bytes

6.11.3 Activate Task (act_tsk, iact_tsk)

C-Language API:

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

Parameters:

```
tskid          Task ID
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_ID        [p]   Invalid ID number
                  (1) CPU ID is invalid (GET_CPUID (tskid) is invalid)
                  (2) Out of local ID range
                      (GET_LOCALID (tskid) < 0 or
                      (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid))
                  (3) tskid = TSK_SELF (0) when called in a non-task context
E_CTX       [k]   Context error (Called in prohibited system state when
                  GET_CPUID (tskid) is not the current CPU)
E_NOEXS     [k]   Non-existent object (Task specified by tskid does not exist)
E_QOVR      [k]   Queuing overflow (actcnt > 15)
```

Function:

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

The processing that is performed during task activation is listed in table 6.7.

Table 6.7 Processing to be Performed during Task Activation

Contents
Initializes base priority and current priority of the task.
Clears the wakeup request count.
Clears the suspension count.
Clears pending-exception causes.
Sets task exception handling disabled state.
Clears the flag pattern of the task event flag.

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

Extended information of the task specified at task creation will be passed to the task as the parameter.

If the static stack of the task indicated by tskid is not being used by any task when service calls act_tsk and iact_tsk are issued, 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 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 15 task activation requests from service calls act_tsk and iact_tsk can be queued.

In service call act_tsk, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call iact_tsk, a task belonging to the kernel of another CPU cannot be specified as tskid.

6.11.4 Cancel Task Activation Requests (can_act, ican_act)

C-Language API:

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

Parameters:

```
tskid          Task ID
```

Return Values:

```
Activation request count (positive value or 0), or error code
```

Error Codes:

```
E_ID        [p]   Invalid ID number
                  (1) CPU ID is invalid (GET_CPUID (tskid) is invalid)
                  (2) Out of local ID range
                      (GET_LOCALID (tskid) < 0 or
                      (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid))
                  (3) tskid = TSK_SELF (0) when called in a non-task context
E_CTX       [k]   Context error (Called in prohibited system state when
                  GET_CPUID (tskid) is not the current CPU)
E_NOEXS     [k]   Non-existent object (Task specified by tskid is not created)
```

Function:

Each service call returns the activation request count for the task specified by tskid, and clears the activation request count.

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

A task in the DORMANT state can also be specified; in this case each service call returns 0.

In service call can_act, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call ican_act, a task belonging to the kernel of another CPU cannot be specified as tskid.

6.11.5 Activate Task with Start Code (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:

tskid	Task ID
stacd	Start code

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (tskid) is invalid) (2) Out of local ID range (GET_LOCALID (tskid) ≤ 0 or (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid))
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (tskid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Task specified by tskid does not exist)
E_OBJ	[k]	Object state is invalid (The task specified by tskid is not in the DORMANT state or the current task is specified)

Function:

Each service call initiates the task indicated by parameter tskid. The initiated task makes a transition from the DORMANT state to the READY state. At this time, the processing to be performed during task activation (table 6.7) is performed. The start code indicated by parameter stacd will be passed to the initiated task as the parameter.

If the static stack of the task indicated by tskid is not being used by any task when service calls sta_tsk and ista_tsk are issued, 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 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.

In service call sta_tsk, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call ista_tsk, a task belonging to the kernel of another CPU cannot be specified as tskid.

6.11.6 Terminate Current Task (ext_tsk), Terminate and Delete Current Task (exd_tsk)

C-Language API:

```
void ext_tsk(void);
void exd_tsk(void);
```

Return Values:

```
Service calls ext_tsk and exd_tsk do not return to the position where they
were issued.
However, if service call ext_tsk or exd_tsk is issued without being installed
at system configuration, error code E_NOSPT is set in R0 and returned.
In addition, service calls ext_tsk and exd_tsk may generate the following
error, and in this case, the system will go down.
E_CTX        [k]   Context error (Called in prohibited system state)
```

Function:

Service call ext_tsk exits the current task normally. After execution of service call ext_tsk, the current task makes a transition from the RUNNING state to the DORMANT state. When an activation request is queued, service call ext_tsk exits the current task and then restarts the task.

The processing that is performed at task termination is listed in table 6.8.

Table 6.8 Processing to be Performed at Task Termination

Contents
Unlock the mutex locked by the task
Set the processing time limit as undefined

Service call exd_tsk exits the current task normally and deletes it. After execution of service call exd_tsk, the current task makes a transition from the RUNNING state to the NON-EXISTENT state. The profile counter of the deleted task is initialized to 0.

Service calls ext_tsk and exd_tsk do not 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 issues service calls ext_tsk and exd_tsk shares the stack with other tasks, the task at the head of the stack wait queue is removed and WAITING state is canceled. At this time, the processing to be performed during task activation (table 6.7) is performed for the task that is removed from the stack wait queue and the task makes a transition to the READY state.

If a stack allocated from the default task stack area is used by the task that issued service call exd_tsk, the stack for that task is released. After that, the size of the free space in the default task stack area will increase by an amount given by the following expression:

Increase in size = (stksz specified at creation) + 16 bytes

Service calls ext_tsk and exd_tsk can be issued while task dispatch is disabled or the CPU is locked. After either of the service calls is issued, the dispatch-disabled state or CPU-locked state is canceled.

Note that when the task returns from the start function, the same operation as for service call ext_tsk will be performed.

6.11.7 Terminate Another Task (ter_tsk)

C-Language API:

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

Parameters:

tskid	Task ID

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (tskid) is not the current CPU) (2) Out of local ID range (GET_LOCALID (tskid) ≤ 0 or (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid))
E_CTX	[k]	Context error (Called in prohibited system state)
E_NOEXS	[k]	Non-existent object (Task specified by tskid does not exist)
E_OBJ	[k]	Object state is invalid (Task specified by tskid is in the DORMANT state)
E_ILUSE	[k]	Illegal use of service call (ID of the current task is specified for tskid)

Function:

Service call ter_tsk forces a task specified by tskid to terminate. The terminated task enters the DORMANT state. At this time, the processing shown in table 6.8 is performed.

When the activation request is queued, the processing to be performed during task activation is performed, and the target task enters the READY state.

A request from a task to force another task to terminate is delayed in the following cases:

- If the task specified by tskid is masking requests from other tasks to force tasks to terminate by calling service call vchg_tmd

Service call ter_tsk does not release resources other than the mutexes (such as semaphores and memory blocks) acquired before the task is terminated. Therefore, the user must call service calls to release resources before calling service call ter_tsk.

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. At this time, the processing to be performed during task activation (table 6.7) is performed for the task that is removed from the stack wait queue and the task makes a transition to the READY state.

In service call ter_tsk, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state.

6.11.8 Change Task Priority (chg_pri, ichg_pri)

C-Language API:

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

Parameters:

tskid	Task ID
tskpri	Base priority of task

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (1) tskpri < 0 (2) tskpri > TMAX_TPRI of GET_CPUID (tskid)
E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (tskid) is invalid) (2) Out of local ID range (GET_LOCALID (tskid) < 0 or (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid)) (3) tskid = TSK_SELF (0) when called in a non-task context
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (tskid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Task specified by tskid does not exist)
E_ILUSE	[k]	Illegal use of service call (Ceiling priority is exceeded)
E_OBJ	[k]	Object state is invalid (Task is in the DORMANT state)

Function:

Each service call changes the base priority of the task specified by parameter tskid to the value specified by parameter tskpri. The current 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 initial priority that was specified at task creation.

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

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 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.

In service call chg_pri, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call ichg_pri, a task belonging to the kernel of another CPU cannot be specified as tskid.

6.11.9 Get Task Priority (get_pri, iget_pri)

C-Language API:

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

Parameters:

tskid	Task ID
p_tskpri	Pointer to the memory area where the current priority of the target task is to be returned

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (tskid) is invalid) (2) Out of local ID range (GET_LOCALID (tskid) < 0 or (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid)) (3) tskid = TSK_SELF (0) when called in a non-task context
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (tskid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Task specified by tskid does not exist)
E_OBJ	[k]	Object state is invalid (Task is in the DORMANT state)

Function:

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

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

In service call get_pri, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call iget_pri, a task belonging to the kernel of another CPU cannot be specified as tskid.

6.11.10 Reference 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:

```
tskid        Task ID
pk_rtsk      Pointer to the packet where the task state is to be returned
```

Return Values:

```
Normal end (E_OK) or error code
```

Packet Structure:

```
typedef    struct    {
           STAT      tskstat;     Task state
           PRI       tskpri;      Current priority of the task
           PRI       tskbpri;     Base priority of the task
           STAT      tskwait;     Wait cause
           ID        wobjid;      Wait object ID
           TMO       lefttmo;     Time to timeout
           UINT      actcnt;      Activation request count
           UINT      wupcnt;      Wakeup request count
           UINT      suscnt;      Suspension count
           UINT      tskmode;     Task execution mode
           UINT      tflptn;      Current task event flag value
}T_RTSK;
```

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (tskid) is invalid) (2) Out of local ID range (GET_LOCALID (tskid) < 0 or (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid)) (3) tskid = TSK_SELF (0) when called in a non-task context
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (tskid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Task specified by tskid does not exist)

Function:

Each service call refers to the state of the task indicated by 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.

  — TTS_RUN (0x00000001): RUNNING state
  — TTS_RDY (0x00000002): READY state
  — TTS_WAI (0x00000004): WAITING state
  — TTS_SUS (0x00000008): SUSPENDED state
  — TTS_WAS (0x0000000c): WAITING-SUSPENDED state
  — TTS_DMT (0x00000010): DORMANT state
  — TTS_STK (0x40000000): Shared-stack WAITING state
- tskpri

  Indicates the current task priority. When the task is in the DORMANT state, the initial priority of the task is returned.
- tskbpri

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

- tskwait*

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

  — TTW_SLP (0x00000001): WAITING state caused by slp_tsk or tslp_tsk
  — TTW_DLY (0x00000002): WAITING state caused by dly_tsk
  — TTW_SEM (0x00000004): WAITING state caused by wai_sem or twai_sem
  — TTW_FLG (0x00000008): WAITING state caused by wai_flg or twai_flg
  — TTW_SDTQ (0x00000010): WAITING state caused by snd_dtq or tsnd_dtq
  — TTW_RDTQ (0x00000020): WAITING state caused by rcv_dtq or trcv_dtq
  — TTW_MBX (0x00000040): WAITING state caused by rcv_mbx or trcv_mbx
  — TTW_MTX (0x00000080): WAITING state caused by loc_mtx or tloc_mtx
  — TTW_SMBF (0x00000100): WAITING state caused by snd_mbf or tsnd_mbf
  — TTW_RMBF (0x00000200): WAITING state caused by rcv_mbf or trcv_mbf
  — TTW_MPF (0x00002000): WAITING state caused by get_mpf or tget_mpf
  — TTW_MPL (0x00004000): WAITING state caused by get_mpl or tget_mpl
  — TTW_TFL (0x00008000): WAITING state caused by vwai_tfl or vtwai_tfl

- wobjid*

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

  The CPU ID for only the task indicated by parameter tskid is set in bits 14 to 12 of wobjid.

- lefttmo*

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

- actcnt*

  The current activation request count is returned.

- wupcnt*

  The current wakeup request count is returned.

- suscnt*

  The current suspension count is returned.

- tskmode*

  The task execution mode set in service call vchg_tmd, and whether there is a request that is delayed by service call vchg_tmd, are returned. The following value is returned to tskmode.

  — ECM_SUS (0x00000001): A suspension request is masked
  — ECM_TER (0x00000002): A forcible termination request is masked
  — PND_SUS (0x00000004): A suspension request is delayed
  — PND_TER (0x00000008): A forcible termination request is delayed

- tflptn*

  The current task event flag value is returned. However, if the task event flag function was not installed at system configuration, an undefined value is returned.

tskmode and tflptn are members not defined in the μITRON4.0 specification.

In service call ref_tsk, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call iref_tsk, a task belonging to the kernel of another CPU cannot be specified as tskid.

6.11.11 Reference 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:

```
tskid        Task ID
pk_rtst      Start address of the packet where the task state is to be
             returned
```

Return Values:

```
Normal end (E_OK) or error code
```

Packet Structure:

```
typedef    struct    {
           STAT      tskstat;     Task state
           STAT      tskwait;     Wait cause
}T_RTST;
```

Error Codes:

```
E_ID       [p]   Invalid ID number
                 (1) CPU ID is invalid (GET_CPUID (tskid) is invalid)
                 (2) Out of local ID range
                     (GET_LOCALID (tskid) < 0 or
                     (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid))
                 (3) tskid = TSK_SELF (0) when called in a non-task context
E_CTX      [k]   Context error (Called in prohibited system state when
                 GET_CPUID (tskid) is not the current CPU)
E_NOEXS    [k]   Non-existent object (Task specified by tskid does not exist)
```

Function:

Each service call refers to the state and the cause of the WAITING state of the task indicated by 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.

  — TTS_RUN (0x00000001): RUNNING state
  — TTS_RDY (0x00000002): READY state
  — TTS_WAI (0x00000004): WAITING state
  — TTS_SUS (0x00000008): SUSPENDED state
  — TTS_WAS (0x0000000c): WAITING-SUSPENDED state
  — TTS_DMT (0x00000010): DORMANT state
  — TTS_STK (0x40000000): Shared-stack WAITING state
- tskwait*

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

  — TTW_SLP (0x00000001): WAITING state caused by slp_tsk or tslp_tsk
  — TTW_DLY (0x00000002): WAITING state caused by dly_tsk
  — TTW_SEM (0x00000004): WAITING state caused by wai_sem or twai_sem
  — TTW_FLG (0x00000008): WAITING state caused by wai_flg or twai_flg
  — TTW_SDTQ (0x00000010): WAITING state caused by snd_dtq or tsnd_dtq
  — TTW_RDTQ (0x00000020): WAITING state caused by rcv_dtq or trcv_dtq
  — TTW_MBX (0x00000040): WAITING state caused by rcv_mbx or trcv_mbx
  — TTW_MTX (0x00000080): WAITING state caused by loc_mtx or tloc_mtx
  — TTW_SMBF (0x00000100): WAITING state caused by snd_mbf or tsnd_mbf
  — TTW_RMBF (0x00000200): WAITING state caused by rcv_mbf or trcv_mbf
  — TTW_MPF (0x00002000): WAITING state caused by get_mpf or tget_mpf
  — TTW_MPL (0x00004000): WAITING state caused by get_mpl or tget_mpl
  — TTW_TFL (0x00008000): WAITING state caused by vwai_tfl or vtwai_tfl

In service call ref_tst, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call iref_tst, a task belonging to the kernel of another CPU cannot be specified as tskid.

6.11.12 Change Task Execution Mode (vchg_tmd)

C-Language API:

```
ER ercd = vchg_tmd(UINT tmd);
```

Parameters:

```
tmd              Task execution mode to change
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_PAR           [p]    Parameter error (tmd is invalid)
E_CTX           [k]    Context error (Called in prohibited system state)
```

Function:

Service call vchg_tmd changes the execution mode of the current task. A mask for requests from other tasks can be specified in tmd as the task execution mode.

- ECM_SUS (0x00000001): Suspension request is masked
- ECM_TER (0x00000002): Forcible termination request is masked

When the suspension request is masked, even if service call sus_tsk or isus_tsk is issued, the request is delayed until the mask is canceled (with tmd = 0 specified) by service call vchg_tmd.

When the forced termination request is masked, even if service call ter_tsk is issued, the request is delayed until the mask is canceled (with tmd = 0 specified) by service call vchg_tmd.

In task execution mode, the state of the calling task is taken over as the task context in extended service call routines and task exception handling routines.

Delays of suspension requests and forcible termination requests can be referenced through service calls ref_tsk and iref_tsk.

This service call is a function not defined in the μITRON4.0 specification.

6.12 Task-Dependent Synchronization

The service calls for task-dependent synchronization are listed in table 6.9.

Table 6.9 Service Calls for Task-Dependent Synchronization

Service Call*1	Description	System State*2						
		T	N	E	D	U	L	C
slp_tsk [B] [S]	Shifts current task to the WAITING state	O		O		O		
tslp_tsk [S]	Shifts current task to the WAITING state with timeout function	O		O		O		
wup_tsk [B] [S] [R]	Wakes up task	O		O	Δ	O		
iwup_tsk [B] [S]			O	O	O	O		
can_wup [B] [S] [R]	Cancels wakeup task	O		O	Δ	O		
ican_wup			O	O	O	O		
rel_wai [B] [S] [R]	Forcibly cancels the WAITING state	O		O	Δ	O		
irel_wai [B] [S]			O	O	O	O		
sus_tsk [B] [S] [R]	Shifts the task to the SUSPENDED state	O		O	Δ	O		
isus_tsk			O	O	O	O		
rsm_tsk [B] [S] [R]	Resumes execution of a task in the SUSPENDED state	O		O	Δ	O		
irsm_tsk			O	O	O	O		
frsm_tsk [S] [R]	Forcibly resumes execution of a task in the SUSPENDED state	O		O	Δ	O		
ifrsm_tsk			O	O	O	O		
dly_tsk [B] [S]	Delays the current task	O		O		O		
vset_tfl [R]	Sets the task event flag	O		O	Δ	O		
ivset_tfl			O	O	O	O		
vclr_tfl [R]	Clears the task event flag	O		O	Δ	O		
ivclr_tfl			O	O	O	O		
vwai_tfl	Waits for the task event flag	O		O		O		
vpol_tfl	Polls and waits for the task event flag	O		O	O	O		
vtwai_tfl	Waits for the task event flag with timeout function	O		O		O		

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

[B]: Basic profile service calls

[R]: Service calls that can be issued remotely

2. T: Can be called in a task context

N: Can be called in a non-task context

E: Can be called in dispatch-enabled state

D: Can be called in dispatch-disabled state

U: Can be called in CPU-unlocked state

L: Can be called in CPU-locked state

C: Can be called while executing the normal CPU exception handler

O: Can be called in the state

∆: Can be called in the state only when a local object is the target

The task-dependent synchronization specifications are listed in table 6.10.

Table 6.10 Task-Dependent Synchronization Specifications

Item	Description
Maximum wakeup request count	15
Maximum suspension count	15
Number of task event flag bits	32 bits (lower 16 bits are reserved for future expansion)
Initial value of task event flag	Initialized as 0 at task activation
Wait condition of task event flag	Waits for a logical OR

6.12.1 Sleep Task (slp_tsk, tslp_tsk)

C-Language API:

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

Parameters:

<tslp_tsk>

tmout	Timeout specification

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (tmout ≤ –2)
E_CTX	[k]	Context error (Called in prohibited system state)
E_TMOUT	[k]	Timeout
E_RLWAI	[k]	WAITING state is forcibly canceled (rel_wai service call was issued while the task was in the WAITING state)

Function:

Each service call shifts the current task to the wakeup WAITING state. However, if wakeup requests are queued for the current task, the wakeup request count is decremented by one and task execution continues.

The wakeup WAITING state is canceled by service calls wup_tsk and iwup_tsk.

In service call tslp_tsk, parameter tmout 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 wakeup request count by one if the wakeup request count is a positive value. If the wakeup 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 is performed. In other words, timeout will not be monitored.

The maximum value that can be specified for tmout is (0x7FFFFFFF – TIC_NUME)/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

For details on time management, refer to section 5.13.7, Time Precision.

6.12.2 Wake up Task (wup_tsk, iwup_tsk)

C-Language API:

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

Parameters:

```
tskid       Task ID
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (tskid) is invalid) (2) Out of local ID range (GET_LOCALID (tskid) < 0 or (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid)) (3) tskid = TSK_SELF (0) when called in a non-task context
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (tskid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Task specified by tskid does not exist)
E_OBJ	[k]	Object state is invalid (Task specified by tskid is in the DORMANT state)
E_QOVR	[k]	Queuing overflow (wupcnt > 15)

Function:

Each service call releases a task from the wakeup WAITING state caused by slp_tsk or tslp_tsk. If the target task is not in the WAITING state, up to 15 requests to wake up a task can be queued.

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

In service call wup_tsk, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call iwup_tsk, a task belonging to the kernel of another CPU cannot be specified as tskid.

6.12.3 Cancel Wakeup Task (can_wup, ican_wup)

C-Language API:

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

Parameters:

tskid	Task ID

Return Values:

Wakeup request count (0 or a positive value) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (tskid) is invalid) (2) Out of local ID range (GET_LOCALID (tskid) < 0 or (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid)) (3) tskid = TSK_SELF (0) when called in a non-task context
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (tskid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Task specified by tskid is not created)
E_OBJ	[k]	Object state is invalid (Task specified by tskid is in the DORMANT state)

Function:

Each service call returns the wakeup request count for the task specified by tskid and invalidates all of those requests.

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

In service call can_wup, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call ican_wup, a task belonging to the kernel of another CPU cannot be specified as tskid.

6.12.4 Forcible Release from WAITING State (rel_wai, irel_wai)

C-Language API:

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

Parameters:

```
tskid          Task ID
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_ID           [p]    Invalid ID number
                      (1) CPU ID is invalid (GET_CPUID (tskid) is invalid)
                      (2) Out of local ID range
                          (GET_LOCALID (tskid) ≤ 0 or
                          (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid))
E_CTX          [k]    Context error (Called in prohibited system state when
                      GET_CPUID (tskid) is not the current CPU)
E_NOEXS        [k]    Non-existent object (Task specified by tskid is not
                      created)
E_OBJ          [k]    Object state is invalid (Task specified by tskid is not
                      in the WAITING state)
```

Function:

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

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

For canceling the SUSPENDED state, rsm_tsk, irsm_tsk, frsm_tsk, or ifrsm_tsk should be used. Note that there is no service call for canceling shared-stack WAITING state.

In service call rel_wai, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call irel_wai, a task belonging to the kernel of another CPU cannot be specified as tskid.

6.12.5 Suspend Task (sus_tsk, isus_tsk)

C-Language API:

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

Parameters:

tskid	Task ID

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (tskid) is invalid) (2) Out of local ID range (GET_LOCALID (tskid) < 0 or (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid)) (3) tskid = TSK_SELF (0) when called in a non-task context
E_CTX	[k]	Context error (1) tskid = TSK_SELF (0) or the current task ID is specified in a task context while dispatch is disabled (2) Called in prohibited system state when GET_CPUID (tskid) is not the current CPU
E_NOEXS	[k]	Non-existent object (Task specified by tskid does not exist)
E_OBJ	[k]	Object state is invalid (Task specified by tskid is in the DORMANT state)
E_QOVR	[k]	Queuing overflow (suscnt > 15)

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 canceled by calling service call rsm_tsk, irsm_tsk, frsm_tsk, or ifrsm_tsk.

Requests to suspend a task by calling service calls sus_tsk and isus_tsk are nested. Up to 15 requests can be queued.

Requests to suspend a task by calling service calls sus_tsk and isus_tsk are delayed in the following cases:

1. When the task specified by tskid masks the suspension request by calling service call vchg_tmd, the task enters the SUSPENDED state immediately after the suspension request is canceled by service call vchg_tmd (by specifying tmd = 0).
2. When the task specified by tskid has issued service call dis_dsp to disable task dispatch, the task enters the SUSPENDED state immediately after task execution resumes.

Delayed requests to suspend a task can be canceled by calling service call rsm_tsk, irsm_tsk, frsm_tsk, or ifrsm_tsk. Therefore, tasks are suspended when there is one or more delayed suspension request.

In service call sus_tsk, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call isus_tsk, a task belonging to the kernel of another CPU cannot be specified as tskid.

6.12.6 Resume Task (rsm_tsk, irsm_tsk), Force Task to Resume (frsm_tsk, ifrsm_tsk)

C-Language API:

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

Parameters:

tskid	Task ID

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (tskid) is invalid) (2) Out of local ID range (GET_LOCALID (tskid) ≤ 0 or (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid))
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (tskid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Task specified by tskid does not exist)
E_OBJ	[k]	Object state is invalid (Task specified by tskid is not in the SUSPENDED state)

Function:

Each service call releases the task specified by parameter tskid from the SUSPENDED state. Service calls rsm_tsk and irsm_tsk decrement, by one, the suspension count, and release the task from the SUSPENDED state when the suspension count becomes 0. Service calls frsm_tsk and ifrsm_tsk clear the suspension count to 0 and release the task from the SUSPENDED state. When the task is in the WAITING-SUSPENDED state, the task is shifted to the WAITING state.

In service call rsm_tsk or frsm_tsk, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call irsm_tsk or ifrsm_tsk, a task belonging to the kernel of another CPU cannot be specified as tskid.

6.12.7 Delay Task (dly_tsk)

C-Language API:

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

Parameters:

```
dlytim          Delayed time
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_CTX          [k]   Context error (Called in prohibited system state)
E_RLWAI        [k]   WAITING state is forcibly canceled
                     (rel_wai service call was issued 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 is returned to the READY state. The current task is put into the WAITING state even if dlytim = 0 is specified.

The maximum value that can be specified for dlytim is (0xFFFFFFFF – TIC_NUME)/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

This service call differs from 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, the WAITING state is not canceled. The WAITING state is canceled before the delay time has elapsed only when service call rel_wai, irel_wai, or ter_tsk is issued.

For details on time management, refer to section 5.13.7, Time Precision.

6.12.8 Set Task Event Flag (vset_tfl, ivset_tfl)

C-Language API:

```
ER ercd = vset_tfl(ID tskid, UINT setptn);
ER ercd = ivset_tfl(ID tskid, UINT setptn);
```

Parameters:

tskid	Task ID
setptn	Bit pattern to set

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (tskid) is invalid) (2) Out of local ID range (GET_LOCALID (tskid) < 0 or (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid)) (3) tskid = TSK_SELF (0) when called in a non-task context
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (tskid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Task specified by tskid does not exist)
E_OBJ	[k]	Object state is invalid (Task specified by tskid is in the DORMANT state)

Function:

The task event flag of the task indicated by parameter tskid is ORed with the value indicated by parameter setptn. Note that the lower 16 bits of the bit pattern to specify in parameter setptn must be set to 0 because the corresponding bits of the event flag are reserved for future expansion.

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

When the logical sum of the waiting pattern and the updated pattern of the task event flag is not 0, the task is released from the WAITING state and the task event flag is cleared to 0.

In service call vset_tfl, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call ivset_tfl, a task belonging to the kernel of another CPU cannot be specified as tskid.

These service calls are functions not defined in the μITRON4.0 specification.

6.12.9 Clear Task Event Flag (vclr_tfl, ivclr_tfl)

C-Language API:

```
ER ercd = vclr_tfl(ID tskid, UINT clrptn);
ER ercd = ivclr_tfl(ID tskid, UINT clrptn);
```

Parameters:

```
tskid         Task ID
clrptn        Bit pattern to clear
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_ID          [p]    Invalid ID number
                     (1) CPU ID is invalid (GET_CPUID (tskid) is invalid)
                     (2) Out of local ID range
                         (GET_LOCALID (tskid) < 0 or
                         (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid))
                     (3) tskid = TSK_SELF (0) when called in a non-task context
E_CTX         [k]    Context error (Called in prohibited system state when
                     GET_CPUID (tskid) is not the current CPU)
E_NOEXS       [p]    Non-existent object (Task specified by tskid does not
                     exist)
E_OBJ         [k]    Object state is invalid
                     (Task specified by tskid is in the DORMANT state)
```

Function:

The task event flag of the task indicated by parameter tskid are ANDed with the value indicated by parameter clrptn. Note that the lower 16 bits of the bit pattern to specify parameter clrptn must be set to 0xffff because the corresponding bits of the event flag are reserved for future expansion.

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

In service call vclr_tfl, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call ivclr_tfl, a task belonging to the kernel of another CPU cannot be specified as tskid.

These service calls are functions not defined in the μITRON4.0 specification.

6.12.10 Wait for Task Event Flag (vwai_tfl, vpol_tfl, vtwai_tfl)

C-Language API:

```
ER ercd = vwai_tfl(UINT waiptn, UINT *p_tflptn);
ER ercd = vpol_tfl(UINT waiptn, UINT *p_tflptn);
ER ercd = vtwai_tfl(UINT waiptn, UINT *p_tflptn, TMO tmout);
```

Parameters:

waiptn	Bit pattern to wait
p_tflptn	Pointer to the memory area where the bit pattern when releasing the WAITING state is to be returned
<vtwai_tfl>	
tmout	Timeout specification

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (waiptn = 0 or tmout ≤ −2)
E_CTX	[k]	Context error (Called in prohibited system state)
E_TMOUT	[k]	Timeout
E_RLWAI	[k]	WAITING state is forcibly canceled (rel_wai service call was called in the WAITING state)

Function:

Each service call waits for any bit of the task event flag specified by waiptn to be set. When the wait release condition is satisfied, the bit pattern of the task event flag is returned to the area indicated by parameter p_tflptn. At the same time, the task event flag value is cleared to 0.

Each service call immediately terminates if any bit specified by waiptn is already set when a service call is issued. If no bit is set, the task that issued service call vwai_tfl or vtwai_tfl enters the WAITING state. With service call vpol_tfl, error code E_TMOUT is immediately returned in this case. Tasks are released from the WAITING state when any bits specified by waiptn are set by service call vset_tfl or ivset_tfl.

The task event flag value is 0 at task activation.

In service call vtwai_tfl, parameter tmout specifies the timeout period.

If a positive value is specified for parameter tmout, error code E_TMOUT is returned when the tmout period has passed without the wait release condition being satisfied.

If tmout = TMO_POL (0) is specified, the same operation as for service call vpol_tfl 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 vwai_tfl will be performed.

The maximum value that can be specified for tmout is (0x7FFFFFFF – TIC_NUME)/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

For details on time management, refer to section 5.13.7, Time Precision.

These service calls are functions not defined in the μITRON4.0 specification.

6.13 Task Exception Handling

Task exception handling is controlled by the service calls listed in table 6.11.

Table 6.11 Service Calls for Task Exception Handling

Service Call*[1]		Description	System State*[2]						
			T	N	E	D	U	L	C
def_tex	[s]	Defines the task exception handling routine	O		O	O	O		
idef_tex				O	O	O	O		
ras_tex	[S]	Requests the task exception handling	O		O	O	O		O
iras_tex	[S]			O	O	O	O		O
dis_tex	[S]	Disables the task exception handling	O		O	O	O		
ena_tex	[S]	Enables the task exception handling	O		O	O	O		
sns_tex	[S]	Refers to the task exception handling disabled state	O	O	O	O	O	O	O
ref_tex		Refers to the task exception handling state	O		O	O	O		
iref_tex				O	O	O	O		

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
[B]: Basic profile service calls
[R]: Service calls that can be issued remotely

2. T: Can be called in a task context
N: Can be called in a non-task context
E: Can be called in dispatch-enabled state
D: Can be called in dispatch-disabled state
U: Can be called in CPU-unlocked state
L: Can be called in CPU-locked state
C: Can be called while executing the normal CPU exception handler
O: Can be called in the state
∆: Can be called in the state only when a local object is the target

The task exception specifications are listed in table 6.12.

Table 6.12 Task Exception Specifications

Item	Description
Exception cause	32 bits
Status at task activation	• Task exception handling disabled state • No pending-exception causes
Task exception handling routine attributes	TA_HLNG: Written in a high-level language TA_ASM: Written in assembly language TA_COP1: FPU is used

The task exception handling routine is initiated as the task context when the following conditions are satisfied.

- Task exception handling enabled state
- Pending-exception cause is not 0
- Task is in the RUNNING state
- Neither a non-task context nor a normal CPU exception handler is not executed

When the task returns from the task exception handling routine, the processing that is performed before the task exception handling routine was initiated is continued. At this time, the task enters the task exception enabled state. When the pending-exception cause is not 0, the task exception handling routine is initiated again.

6.13.1 Define Task Exception Handling Routine (def_tex, idef_tex)

C-Language API:

```
ER ercd = def_tex(ID tskid, T_DTEX *pk_dtex);
ER ercd = idef_tex(ID tskid, T_DTEX *pk_dtex);
```

Parameters:

```
tskid        Task ID
pk_dtex      Pointer to the packet where the task exception-processing-
             routine definition information is stored
```

Return Values:

```
Normal end (E_OK) or error code
```

Packet Structure

```
typedef   struct   {
          ATR      texatr;     Task exception handling routine attribute
          FP       texrtn;     Task exception handling routine initiation
                               address
}T_DTEX;
```

Error Codes:

```
E_RSATR   [p]   Reserved attribute (texatr is invalid)
E_ID      [p]   Invalid ID number
                (1) CPU ID is invalid (GET_CPUID (tskid) is not the current
                    CPU)
                (2) Out of local ID range
                    (GET_LOCALID (tskid) < 0 or
                    (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid))
                (3) tskid = TSK_SELF (0) when called in a non-task context
E_NOEXS   [k]   Non-existent object (Task specified by tskid does not exist)
```

Function:

The task exception handling routine indicated by tskid is defined as specified by pk_dtex.

Only a task belonging to the kernel of the current CPU can be specified for tskid.

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

Parameter texatr specifies the language in which the task exception handling routine was written and the coprocessor to be used as the attributes.

texatr := ((TA_HLNG || TA_ASM) [|TA_COP1])

- TA_HLNG (0x00000000): Written in a high-level language
- TA_ASM (0x00000001): Written in assembly language
- TA_COP1 (0x00000200): FPU is used

The FPU registers can also be guaranteed as task context by specifying the TA_COP1 attribute. Note that the TA_COP1 attribute is not defined in the μITRON4.0 specification.

texrtn specifies the start address of the task exception handling routine. When, in a service call def_tex or idef_tex, pk_dtex = NULL(0) is specified, the definition of the task exception handling routine for tskid is canceled. At this time the task pending-exception cause is cleared to 0, and the task is transferred to the task exception handling disabled state.

If a task exception handling routine has already been defined, the previous definition is canceled and is replaced with the new definition. At this time, pending-exception causes are not cleared and task exception handling is not disabled.

6.13.2 Request Task Exception Handling (ras_tex, iras_tex)

C-Language API:

```
ER ercd = ras_tex(ID tskid, TEXPTN rasptn);
ER ercd = iras_tex(ID tskid, TEXPTN rasptn);
```

Parameters:

tskid	Task ID
rasptn	Task exception cause of task exception handling to be requested

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (rasptn = 0)
E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (tskid) is not the current CPU) (2) Out of local ID range (GET_LOCALID (tskid) < 0 or (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid)) (3) tskid = TSK_SELF (0) when called in a non-task context
E_NOEXS	[k]	Non-existent object (Task specified by tskid does not exist)
E_OBJ	[k]	Object state is invalid (Task specified by tskid is in the DORMANT state or task exception handling routine is not defined)

Function:

Requests task exception handling by the task exception cause specified by rasptn, for the task specified by tskid. That is, the pending-exception cause for the task is ORed with the value indicated by parameter rasptn.

Only a task belonging to the kernel of the current CPU can be specified for tskid.

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

When the conditions for starting task exception handling routine are satisfied, the task exception handling routine is initiated.

Each service call can also be issued from the normal CPU exception handler.

6.13.3 Disable Task Exception Handling (dis_tex)

C-Language API:

```
ER ercd = dis_tex(void);
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_OBJ           [k]   Object state is invalid (Task exception handling routine
                      is not defined on the current task)
E_CTX           [k]   Context error (Called in prohibited system state)
```

Function:

The current task is transferred to the task exception handling disabled state.

6.13.4 Enable Task Exception Handling (ena_tex)

C-Language API:

```
ER ercd = ena_tex(void);
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_OBJ          [k]   Object state is invalid (Task exception handling routine
                     is not defined on the current task)
E_CTX          [k]   Context error (Called in prohibited system state)
```

Function:

The current task is transferred to the task exception enabled state.

When conditions for starting the task exception handling routine are satisfied through this service call, the task exception handling routine is initiated.

6.13.5 Reference Task Exception Handling Disabled State (sns_tex)

C-Language API:

```
BOOL state= sns_tex(void);
```

Return Values:

```
TRUE is returned when the task is in task exception handling disabled state
and FALSE is returned when the task is in task exception handling enabled
state
```

Function:

Checks whether a task in the RUNNING state is in the task exception handling disabled state.

A task in the RUNNING state is the current task when called in a task context, and when called in a non-task context is the task which had been run immediately prior to the transition to the non-task context. When a task is called in a non-task context, and no task is in the RUNNING state, TRUE is returned.

Tasks for which no task exception handling routines are defined are held in the task exception handling disabled state, so that when no task exception handling routine has been defined for a task in the RUNNING state, this service call returns TRUE.

This service call can also be issued in the CPU-locked state and from the normal CPU exception handler.

6.13.6 Reference Task Exception Handling State (ref_tex, iref_tex)

C-Language API:

```
ER ercd = ref_tex(ID tskid, T_RTEX *pk_rtex);
ER ercd = iref_tex(ID tskid, T_RTEX *pk_rtex);
```

Parameters:

tskid — Task ID

pk_rtex — Pointer to the packet where the task exception handling state is to be returned

Return Values:

Normal end (E_OK) or error code

Packet Structure:

```
typedef     struct    {
            STAT      texstat;     Task exception handling state
            TEXPTN    pndptn;      Pending-exception cause
}T_RTEX;
```

Error Codes:

E_ID [p] Invalid ID number

(1) CPU ID is invalid (GET_CPUID (tskid) is not the current CPU)

(2) Out of local ID range
(GET_LOCALID (tskid) < 0 or
(_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid))

(3) tskid = TSK_SELF (0) when called in a non-task context

E_NOEXS [k] Non-existent object (Task specified by tskid does not exist)

E_OBJ [k] Object state is invalid (Task specified by tskid is in the DORMANT state or task exception handling routine is not defined)

Function:

The state relating to task exception handling for the task specified by tskid is referenced, and the result is returned to the area indicated by pk_rtex.

One of the following values is returned for texstat, according to whether the target task is in a task exception enabled state or a task exception handling disabled state.

- TTEX_ENA (0x00000000): Task exception handling enabled state
- TTEX_DIS (0x00000001): Task exception handling disabled state

The pending-exception cause for the target task is returned as pndptn. If there are no unprocessed exception processing requests, 0 is returned as pndptn.

Only a task belonging to the kernel of the current CPU can be specified for tskid.

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

6.14 Synchronization and Communication (Semaphore)

Semaphores are controlled by the service calls listed in table 6.13.

Table 6.13 Service Calls for Synchronization and Communication (Semaphore)

Service Call*1	Description	System State*2						
		T	N	E	D	U	L	C
cre_sem [s]	Creates semaphore	O		O	O	O		
icre_sem			O	O	O	O		
acre_sem	Creates semaphore and assigns semaphore ID automatically	O		O	O	O		
iacre_sem			O	O	O	O		
del_sem	Deletes semaphore	O		O	O	O		
sig_sem [B] [S] [R]	Returns semaphore resource	O		O	Δ	O		
isig_sem [B] [S]			O	O	O	O		
wai_sem [B] [S] [R]	Waits for semaphore resource	O		O		O		
pol_sem [B] [S] [R]	Polls and waits for semaphore resource	O		O	Δ	O		
ipol_sem			O	O	O	O		
twai_sem [S] [R]	Waits for semaphore resource with timeout function	O		O		O		
ref_sem [R]	Refers to semaphore state	O		O	Δ	O		
iref_sem			O	O	O	O		

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
[B]: Basic profile service calls
[R]: Service calls that can be issued remotely

2. T: Can be called in a task context
N: Can be called in a non-task context
E: Can be called in dispatch-enabled state
D: Can be called in dispatch-disabled state
U: Can be called in CPU-unlocked state
L: Can be called in CPU-locked state
C: Can be called while executing the normal CPU exception handler
O: Can be called in the state
Δ: Can be called in the state only when a local object is the target

The semaphore specifications are listed in table 6.14.

Table 6.14 Semaphore Specifications

Item	Description
Local semaphore ID	1 to _MAX_SEM (1023 max.)
Maximum semaphore count	65535
Semaphore attributes	TA_TFIFO: Wait task queue is managed on a FIFO basis TA_TPRI: Wait task queue is managed on the current priority

6.14.1 Create Semaphore (cre_sem, icre_sem) (acre_sem, iacre_sem: Assign Semaphore ID Automatically)

C-Language API:

```
ER ercd = cre_sem(ID semid, T_CSEM *pk_csem);
ER ercd = icre_sem(ID semid, T_CSEM *pk_csem);
ER_ID semid = acre_sem(T_CSEM *pk_csem);
ER_ID semid = iacre_sem(T_CSEM *pk_csem);
```

Parameters:

```
pk_csem                Pointer to the packet where the semaphore creation
                       information is stored
<cre_sem, icre_sem>
semid                  Semaphore ID
```

Return Values:

```
<cre_sem, icre_sem>
Normal end (E_OK) or error code
<acre_sem, iacre_sem>
ID of created semaphore (a positive value) or error code
```

Packet Structure

```
typedef  struct   {
         ATR      sematr;     Semaphore attribute
         UINT     isemcnt     Initial semaphore resource count
         UINT     maxsem;     Maximum semaphore resource count
}T_CSEM;
```

Error Codes:

```
E_RSATR    [p]   Invalid attribute (sematr is invalid)
E_PAR      [p]   Parameter error (maxsem = 0, maxsem > 0xffff, or
                 isemcnt > maxsem)
```

E_ID	[p]	Invalid ID number (cre_sem, icre_sem) (1) CPU ID is invalid (GET_CPUID (semid) is not the current CPU) (2) Out of local ID range (GET_LOCALID (semid) ≤ 0 or (_MAX_TSK of GET_CPUID (semid)) < GET_LOCALID (semid))
E_OBJ	[k]	Object state is invalid (Semaphore indicated by semid already exists) (cre_sem, icre_sem)
E_NOID	[k]	No ID available (acre_sem, iacre_sem)

Function:

Each of these service calls creates a semaphore.

These service calls can create semaphores belonging to the kernel of the current CPU. This kernel does not have service calls for creating objects belonging to the kernel of another CPU.

Service calls cre_sem and icre_sem create a semaphore with an ID indicated by semid. 1 to (_MAX_SEM of current CPU) can be specified for the local ID of semid. VCPU_SELF or the current CPU ID must be specified for the CPU ID of semid.

Service calls acre_sem and iacre_sem search for an unused semaphore ID, create a semaphore with that ID, and return the ID to semid. The range searched for the local semaphore ID is 1 to (_MAX_SEM of current CPU). The CPU ID of the semaphore ID that will be returned is the current CPU ID.

Parameter sematr specifies the order of the tasks in the queue waiting to acquire the semaphore resource as the attribute.

sematr := (TA_TFIFO || TA_TPRI)

- TA_TFIFO (0x00000000): Wait task queue is managed on a FIFO basis
- TA_TPRI (0x00000001): Wait task queue is managed on the current priority

Parameter isemcnt specifies the initial value of the semaphore to be created. It can range from 0 to maxsem.

Parameter maxsem specifies the maximum number of resources of the semaphore to be created. It can range from 1 to 65,535.

6.14.2 Delete Semaphore (del_sem)

C-Language API:

```
ER ercd = del_sem(ID semid);
```

Parameters:

semid Semaphore ID

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (semid) is not the current CPU) (2) Out of local ID range (GET_LOCALID (semid) ≤ 0 or (_MAX_SEM of GET_CPUID (semid)) < GET_LOCALID (semid))
E_CTX	[k]	Context error (Called in prohibited system state)
E_NOEXS	[k]	Non-existent object (Semaphore indicated by semid does not exist)

Function:

Service call del_sem deletes the semaphore indicated by parameter semid.

Only semaphores belonging to the kernel of the current CPU can be specified for semid.

No error will occur even if there is a task waiting to acquire a resource with the semaphore indicated by semid. However, in that case, the task in the WAITING state will be released and error code E_DLT will be returned.

6.14.3 Release Semaphore Resource (sig_sem, isig_sem)

C-Language API:

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

Parameters:

semid	Semaphore ID

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (semid) is invalid) (2) Out of local ID range (GET_LOCALID (semid) ≤ 0 or (_MAX_SEM of GET_CPUID (semid)) < GET_LOCALID (semid))
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (semid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Semaphore indicated by semid does not exist)
E_QOVR	[k]	Queuing overflow (semcnt > maxsem*)

Note: * maxsem: Maximum number of semaphore resources specified at semaphore creation

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 semaphore count is incremented by one.

The maximum semaphore count is maxsem, which is specified at semaphore creation.

In service call sig_sem, semaphores belonging to the kernel of another CPU can be specified as semid, except for in dispatch-pending state. In service call isig_sem, semaphores belonging to the kernel of another CPU cannot be specified as semid.

6.14.4 Acquire Semaphore Resource (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:

semid	Semaphore ID
<twai_sem>	
tmout	Timeout specification

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (tmout ≤ -2)
E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (semid) is invalid) (2) Out of local ID range (GET_LOCALID (semid) ≤ 0 or (_MAX_SEM of GET_CPUID (semid)) < GET_LOCALID (semid))
E_CTX	[k]	Context error (Called in prohibited system state)
E_NOEXS	[k]	Non-existent object (Semaphore indicated by semid does not exist)
E_DLT	[k]	Waiting object deleted (Target semaphore indicated by semid has been deleted while waiting)
E_RLWAI	[k]	WAITING state is forcibly canceled (rel_wai service call was called 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 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 according to the attribute specified at creation.

In service call twai_sem, parameter tmout specifies the timeout period.

If a positive value is specified for 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 service call pol_sem will be performed.

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

The maximum value that can be specified for tmout is (0x7FFFFFFF – TIC_NUME)/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

For details on time management, refer to section 5.13.7, Time Precision.

In service call wai_sem, pol_sem, or twai_sem, semaphores belonging to the kernel of another CPU can be specified as semid, except for in dispatch-pending state. In service call ipol_sem, semaphores belonging to the kernel of another CPU cannot be specified as semid.

6.14.5 Reference 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:

```
semid          Semaphore ID
pk_rsem        Pointer to the packet where the semaphore state is to be
               returned
```

Return Values:

```
Normal end (E_OK) or error code
```

Packet Structure:

```
typedef     struct     {
            ID         wtskid;     Wait task ID
            UINT       semcnt;     Current semaphore count value
}T_RSEM;
```

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (semid) is invalid) (2) Out of local ID range (GET_LOCALID (semid) ≤ 0 or (_MAX_SEM of GET_CPUID (semid)) < GET_LOCALID (semid))
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (semid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Semaphore indicated by semid does not exist)

Function:

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

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

The CPU ID (1 or 2) where the specified semaphore belongs is always set in bits 14 to 12 of a wait task ID.

In the case where a task of another CPU is waiting to acquire a resource from the specified semaphore, an SVC server task belonging to the same CPU as the specified semaphore will actually wait to acquire a resource from the specified semaphore instead of the task of another CPU. Accordingly, the ID of the SVC server task will be returned to the wait task ID in such a case.

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

In service call ref_sem, semaphores belonging to the kernel of another CPU can be specified as semid, except for in dispatch-pending state. In service call iref_sem, semaphores belonging to the kernel of another CPU cannot be specified as semid.

6.15 Synchronization and Communication (Event Flag)

Event flags are controlled by the service calls listed in table 6.15.

Table 6.15 Service Calls for Synchronization and Communication (Event Flag)

Service Call*1	Description	System State*2						
		T	N	E	D	U	L	C
cre_flg [s]	Creates event flag	O		O	O	O		
icre_flg			O	O	O	O		
acre_flg	Creates event flag and assigns event flag ID automatically	O		O	O	O		
iacre_flg			O	O	O	O		
del_flg	Deletes event flag	O		O	O	O		
set_flg [B] [S] [R]	Sets event flag	O		O	∆	O		
iset_flg [S] [R]			O	O	O	O		
clr_flg [B] [S] [R]	Clears event flag	O		O	∆	O		
iclr_flg			O	O	O	O		
wai_flg [B] [S] [R]	Waits for event flag	O		O		O		
pol_flg [B] [S] [R]	Polls and waits for event flag	O		O	∆	O		
ipol_flg [S]			O	O	O	O		
twai_flg [S] [R]	Waits for event flag with timeout function	O		O		O		
ref_flg [R]	Refers to event flag state	O		O	∆	O		
iref_flg			O	O	O	O		

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
[B]: Basic profile service calls
[R]: Service calls that can be issued remotely

2. T: Can be called in a task context
N: Can be called in a non-task context
E: Can be called in dispatch-enabled state
D: Can be called in dispatch-disabled state
U: Can be called in CPU-unlocked state
L: Can be called in CPU-locked state
C: Can be called while executing the normal CPU exception handler
O: Can be called in the state
∆: Can be called in the state only when a local object is the target

The event flag specifications are listed in table 6.16.

Table 6.16 Event Flag Specifications

Item	Description
Local event flag ID	1 to _MAX_FLAG (1023 max.)
Event flag size	32 bits
Event flag attributes	TA_TFIFO: Wait task queue is managed on a FIFO basis TA_TPRI: Wait task queue is managed on the current priority 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: Clears the event flag at the time of waiting release

6.15.1 Create Event Flag (cre_flg, icre_flg) (acre_flg, iacre_flg: Assign Event Flag ID Automatically)

C-Language API:

```
ER ercd = cre_flg(ID flgid, T_CFLG *pk_cflg);
ER ercd = icre_flg(ID flgid, T_CFLG *pk_cflg);
ER_ID flgid = acre_flg(T_CFLG *pk_cflg);
ER_ID flgid = iacre_flg(T_CFLG *pk_cflg);
```

Parameters:

```
pk_cflg        Pointer to the packet where the event flag creation information
               is stored
<cre_flg, icre_flg>
flgid          Event flag ID
```

Return Values:

```
<cre_flg, icre_flg>
Normal end (E_OK) or error code
<acre_flg, iacre_flg>
Created event flag ID (a positive value) or error code
```

Packet Structure:

```
typedef    struct    {
           ATR       flgatr;    Event flag attribute
           FLGPTN    iflgptn;   Initial value of event flag
}T_CFLG;
```

Error Codes:

```
E_RSATR    [p]   Invalid attribute (flgatr is invalid)
E_ID       [p]   Invalid ID number (cre_flg, icre_flg)
                 (1) CPU ID is invalid (GET_CPUID (flgid) is not the current
                     CPU)
                 (2) Out of local ID range
                     (GET_LOCALID (flgid) ≤ 0 or
                     (_MAX_FLAG of GET_CPUID (flgid)) < GET_LOCALID (flgid))
E_OBJ      [k]   Object state is invalid (Event flag indicated by flgid
                 already exists) (cre_flg, icre_flg)
E_NOID     [k]   No ID available (acre_flg, iacre_flg)
```

Function:

Each of these service calls creates an event flag.

These service calls can create event flags belonging to the kernel of the current CPU. This kernel does not have service calls for creating objects belonging to the kernel of another CPU.

Service calls cre_flg and icre_flg create an event flag with an ID indicated by flgid. 1 to (_MAX_FLAG of current CPU) can be specified for the local ID of flgid. VCPU_SELF or the current CPU ID must be specified for the CPU ID of flgid.

Service calls acre_flg and iacre_flg search for an unused event flag ID, create an event flag with that ID, and return the ID to flgid. The range searched for the local event flag ID is 1 to (_MAX_FLAG of current CPU). The CPU ID of the event flag ID that will be returned is the current CPU ID.

Parameter flgatr specifies the order of the tasks in the queue waiting for the event flag and the number of tasks allowed to wait for the event flag as the attributes.

flgatr := ((TA_TFIFO || TA_TPRI) | (TA_WSGL || TA_WMUL) | [TA_CLR])

- TA_TFIFO (0x00000000): Wait task queue is managed on a FIFO basis
- TA_TPRI (0x00000001): Wait task queue is managed on the current priority
- TA_WSGL (0x00000000): Does not permit multiple tasks to wait for the event flag
- TA_WMUL (0x00000002): Permits multiple tasks to wait for the event flag
- TA_CLR (0x00000004): Clears event flag at the time of waiting release

If the TA_WSGL attribute is specified for flgatr, only one task can wait for the created event flag. In this case, the event flag performs the same operation when either attribute TA_TFIFO or TA_TPRI is specified. On the other hand, multiple tasks can enter the WAITING state when the TA_WMUL attribute is specified. If the TA_CLR attribute is specified for flgatr, all bits of the event flag bit pattern are cleared when the wait release condition is satisfied.

Parameter iflgptn specifies the initial value of the event flag.

6.15.2 Delete Event Flag (del_flg)

C-Language API:

```
ER ercd = del_flg(ID flgid);
```

Parameters:

`flgid`   `Event flag ID`

Return Values:

`Normal end (E_OK) or error code`

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (flgid) is not the current CPU) (2) Out of local ID range (GET_LOCALID (flgid) ≤ 0 or (_MAX_FLAG of GET_CPUID (flgid)) < GET_LOCALID (flgid))
E_CTX	[k]	Context error (Called in prohibited system state)
E_NOEXS	[k]	Non-existent object (Event flag indicated by flgid does not exist)

Function:

Service call del_flg deletes the event flag indicated by parameter flgid.

Only event flags belonging to the kernel of the current CPU can be specified as flgid.

No error will occur even if there is a task waiting for the conditions to be met in the event flag indicated by flgid. However, in that case, the task in the WAITING state will be released and error code E_DLT will be returned.

6.15.3 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:

flgid	Event flag ID
setptn	Bit pattern to set

Return Values:

Normal end (E_OK) or error code

Error Codes:

```
E_ID        [p]  Invalid ID number
                 (1) CPU ID is invalid (GET_CPUID (flgid) is invalid)
                 (2) Out of local ID range
                     (GET_LOCALID (flgid) ≤ 0 or
                     (_MAX_FLAG of GET_CPUID (flgid)) < GET_LOCALID (flgid))
E_CTX       [k]  Context error (Called in prohibited system state when
                 GET_CPUID (flgid) is not the current CPU)
E_NOEXS     [k]  Non-existent object (Event flag indicated by flgid does not
                 exist)
```

Function:

The event flag specified by flgid is ORed with the value indicated by 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 service call set_flg is issued only once. When multiple wait tasks are released, the tasks are released in the queue order of the event flag.

In service call set_flg, event flags belonging to the kernel of another CPU can be specified as flgid, except for in dispatch-pending state. In service call iset_flg, event flags belonging to the kernel of another CPU cannot be specified as flgid.

6.15.4 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:

flgid	Event flag ID
clrptn	Bit pattern to clear

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (flgid) is invalid) (2) Out of local ID range (GET_LOCALID (flgid) ≤ 0 or (_MAX_FLAG of GET_CPUID (flgid)) < GET_LOCALID (flgid))
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (flgid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Event flag indicated by flgid does not exist)

Function:

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

In service call clr_flg, event flags belonging to the kernel of another CPU can be specified as flgid, except for in dispatch-pending state. In service call iclr_flg, event flags belonging to the kernel of another CPU cannot be specified as flgid.

6.15.5 Wait for Event-Flag Setting (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:

flgid	Event flag ID
waiptn	Wait bit pattern
wfmode	Wait mode
p_flgptn	Pointer to the memory area where the bit pattern at waiting release is to be returned
<twai_flg>	
tmout	Timeout value

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (waiptn = 0, wfmode is invalid, or tmout ≤ -2)
E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (flgid) is invalid) (2) Out of local ID range (GET_LOCALID (flgid) ≤ 0 or (_MAX_FLAG of GET_CPUID (flgid)) < GET_LOCALID (flgid))
E_CTX	[k]	Context error (Called in prohibited system state)
E_NOEXS	[k]	Non-existent object (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_DLT	[k]	Waiting object deleted (Event flag indicated by flgid has been deleted in the WAITING state)
E_TMOUT	[k]	Polling failed or timeout
E_RLWAI	[k]	WAITING state was forcibly canceled (rel_wai service call was called in the WAITING state)

Function:

A task that has called one of these service calls waits until the event flag specified by parameter flgid is set according to the waiting conditions indicated by 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 issues 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 service call wai_flg or twai_flg is issued. With service call pol_flg or ipol_flg, error code E_TMOUT is immediately returned, and then the task terminates.

Parameter wfmode specifies the following as the attribute.

wfmode := ((TWF_ANDW || TWF_ORW))

- TWF_ANDW (0x00000000): AND wait
- TWF_ORW (0x00000001): 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.

In service call twai_flg, parameter tmout specifies the timeout period.

If a positive value is specified for parameter tmout, error code E_TMOUT is returned when the tmout period has passed without the waiting release conditions being satisfied.

If tmout = TMO_POL (0) is specified, the same operation as for service call pol_flg will be performed.

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

The maximum value that can be specified for tmout is (0x7FFFFFFF – TIC_NUME)/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

For details on time management, refer to section 5.13.7, Time Precision.

In service call wai_flg, pol_flg, or twai_flg, event flags belonging to the kernel of another CPU can be specified as flgid, except for in dispatch-pending state. In service call ipol_flg, event flags belonging to the kernel of another CPU cannot be specified as flgid.

6.15.6 Reference 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:

flgid	Event flag ID
pk_rflg	Pointer to the packet where the event flag state is to be returned

Return Values:

Normal end (E_OK) or error code

Packet Structure:

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

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (flgid) is invalid) (2) Out of local ID range (GET_LOCALID (flgid) ≤ 0 or (_MAX_FLAG of GET_CPUID (flgid)) < GET_LOCALID (flgid))
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (flgid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Event flag indicated by flgid does not exist)

Function:

Each of these service calls refers to the state of the event flag indicated by 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 parameter pk_rflg.

The CPU ID (1 or 2) where the specified event flag belongs is always set in bits 14 to 12 of a wait task ID.

In the case where a task of another CPU is waiting for the specified event flag to be set, an SVC server task belonging to the same CPU as the specified event flag will actually wait for the specified event flag to be set instead of the task of another CPU. Accordingly, the ID of the SVC server task will be returned to the wait task ID in such a case.

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

In service call ref_flg, event flags belonging to the kernel of another CPU can be specified as flgid, except for in dispatch-pending state. In service call iref_flg, event flags belonging to the kernel of another CPU cannot be specified as flgid.

6.16 Synchronization and Communication (Data Queue)

Data queues are controlled by the service calls listed in table 6.17.

Table 6.17 Service Calls for Synchronization and Communication (Data Queue)

Service Call*[1]	Description	System State*[2]						
		T	N	E	D	U	L	C
cre_dtq [s]	Creates data queue	O		O	O	O		
icre_dtq			O	O	O	O		
acre_dtq	Creates data queue and assigns data queue ID automatically	O		O	O	O		
iacre_dtq			O	O	O	O		
del_dtq	Deletes data queue	O		O	O	O		
snd_dtq [S] [R]	Sends data to data queue	O		O		O		
psnd_dtq [S] [R]	Polls and sends data to data queue	O		O	∆	O		
ipsnd_dtq [S]			O	O	O	O		
tsnd_dtq [S] [R]	Sends data to data queue with timeout function	O		O		O		
fsnd_dtq [S] [R]	Forcibly sends data to data queue	O		O	∆	O		
ifsnd_dtq [S]			O	O	O	O		
rcv_dtq [S] [R]	Receives data from data queue	O		O		O		
prcv_dtq [S] [R]	Polls and receives data from data queue	O		O	∆	O		
trcv_dtq [S] [R]	Receives data from data queue with timeout function	O		O		O		
ref_dtq [R]	Refers to data queue state	O		O	∆	O		
iref_dtq			O	O	O	O		

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
[B]: Basic profile service calls
[R]: Service calls that can be issued remotely

2. T: Can be called in a task context
N: Can be called in a non-task context
E: Can be called in dispatch-enabled state
D: Can be called in dispatch-disabled state
U: Can be called in CPU-unlocked state
L: Can be called in CPU-locked state
C: Can be called while executing the normal CPU exception handler
O: Can be called in the state
∆: Can be called in the state only when a local object is the target

The data queue specifications are listed in table 6.18.

Table 6.18 Data Queue Specifications

Item	Description
Local data queue ID	1 to _MAX_DTQ (1023 max.)
One word	32 bits
Data queue attributes	TA_TFIFO: Wait task queue is managed on a FIFO basis TA_TPRI: Wait task queue is managed on the current priority

6.16.1 Create Data Queue (cre_dtq, icre_dtq) (acre_dtq, iacre_dtq: Assign Data Queue ID Automatically)

C-Language API:

```
ER ercd = cre_dtq(ID dtqid, T_CDTQ *pk_cdtq);
ER ercd = icre_dtq (ID dtqid, T_CDTQ *pk_cdtq);
ER_ID dtqid = acre_dtq (T_CDTQ *pk_cdtq);
ER_ID dtqid = iacre_dtq (T_CDTQ *pk_cdtq);
```

Parameters:

```
pk_cdtq         Pointer to the packet where the data queue creation information
                is stored
<cre_dtq, icre_dtq>
dtqid           Data queue ID
```

Return Values:

```
<cre_dtq, icre_dtq>
Normal end (E_OK) or error code
<acre_dtq, iacre_dtq>
Created data queue ID (a positive value) or error code
```

Packet Structure:

```
typedef   struct   {
          ATR      dtqatr;   Data queue attribute
          UINT     dtqcnt;   Size of data queue area (the number of data items)
          VP       dtq;      Start address of data queue area
}T_CDTQ;
```

Error Codes:

E_NOMEM	[k]	Insufficient memory (Data queue area cannot be allocated in the memory)
E_RSATR	[p]	Invalid attribute (dtqatr is invalid)
E_PAR	[p]	Parameter error (TSZ_DTQ (dtqcnt) exceeds 32-bit area)
E_ID	[p]	Invalid ID number (cre_dtq, icre_dtq) (1) CPU ID is invalid (GET_CPUID (dtqid) is not the current CPU) (2) Out of local ID range (GET_LOCALID (dtqid) ≤ 0 or (_MAX_DTQ of GET_CPUID (dtqid)) < GET_LOCALID (dtqid))
E_OBJ	[k]	Object state is invalid (Data queue indicated by dtqid already exists) (cre_dtq, icre_dtq)
E_NOID	[k]	No ID available (acre_dtq, iacre_dtq)

Function:

Each of these service calls creates a data queue.

These service calls can create data queues belonging to the kernel of the current CPU. This kernel does not have service calls for creating objects belonging to the kernel of another CPU.

Service calls cre_dtq and icre_dtq create a data queue with the ID specified by dtqid. 1 to (_MAX_DTQ of current CPU) can be specified for the local ID of dtqid. VCPU_SELF or the current CPU ID must be specified for the CPU ID of dtqid.

Service calls acre_dtq and iacre_dtq search for an unused data queue ID, create a data queue with that ID, and return the ID to dtqid. The range searched for the local data queue ID is 1 to (_MAX_DTQ of current CPU). The CPU ID of the data queue ID that will be returned is the current CPU ID.

(1) dtqatr

Parameter dtqatr specifies the order of the tasks in the queue waiting to send data as the attribute.

dtqatr := (TA_TFIFO || TA_TPRI)

- TA_TFIFO (0x0000000): Wait task queue is managed on a FIFO basis
- TA_TPRI (0x00000001): Wait task queue is managed on the current priority

The tasks in the queue waiting to receive data are managed on a first-in first-out (FIFO) basis, regardless of dtqatr.

(2) dtqcnt

Parameter dtqcnt specifies the number of data items that can be stored in the data queue area.

It is also possible to create a data queue with a value of 0 specified for dtqcnt. Since data cannot be stored in a data queue created by dtqcnt = 0, the data sending task or data receiving task that has performed its operation first will enter the WAITING state. The WAITING state of that task is canceled when the other task has performed its operation. Thus, data sending tasks and data receiving tasks are completely synchronized.

(3) dtq

Parameter dtq specifies the start address of a free area to be used as a data queue. An area of TSZ_DTQ (dtqcnt) bytes from dtq is used as the data queue. TSZ_DTQ() is a macro used for calculating the data queue size.

When a value of 0 is specified for dtq, dtq does not have any meaning and is simply ignored.

When NULL is specified for dtq, the kernel allocates a data queue area of TSZ_DTQ (dtqcnt) bytes from the default data queue area. After that, the size of the free space in the default data queue area will decrease by an amount given by the following expression:

Decrease in size = TSZ_DTQ (dtqcnt) + 16 bytes

6.16.2 Delete Data Queue (del_dtq)

C-Language API:

```
ER ercd = del_dtq(ID dtqid);
```

Parameters:

```
dtqid           Data queue ID
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_ID          [p]   Invalid ID number
                    (1) CPU ID is invalid (GET_CPUID (dtqid) is not the
                        current CPU)
                    (2) Out of local ID range
                        (GET_LOCALID (dtqid) ≤ 0 or
                        (_MAX_DTQ of GET_CPUID (dtqid)) < GET_LOCALID (dtqid))
E_CTX         [k]   Context error (Called in prohibited system state)
E_NOEXS       [k]   Non-existent object (Data queue indicated by dtqid does
                    not exist)
```

Function:

The data queue specified by dtqid is deleted.

Only data queues belonging to the kernel of the current CPU can be specified as dtqid.

No error occurs even if there is a send-waiting task or receive-waiting task in the data queue specified by dtqid. However, the WAITING state of the task is canceled, and an error code E_DLT is returned.

On deletion, the size of the free space in the default data queue area will increase by an amount given by the following expression:

Increase in size = TSZ_DTQ (dtqcnt specified at creation) + 16 bytes

6.16.3 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:

dtqid	Data queue ID
data	Data sent to data queue
<tsnd_dtq>	
tmout	Timeout specification

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (tmout ≤ -2)
E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (dtqid) is invalid) (2) Out of local ID range (GET_LOCALID (dtqid) ≤ 0 or (_MAX_DTQ of GET_CPUID (dtqid)) < GET_LOCALID (dtqid))
E_CTX	[k]	Context error (Called in prohibited system state)
E_ILUSE	[k]	Illegal use of service call (fsnd_dtq, ifsnd_dtq is issued for the data queue which dtqcnt is 0)
E_NOEXS	[k]	Non-existent object (Data queue indicated by dtqid does not exist)
E_DLT	[k]	Waiting object deleted (Target data queue indicated by dtqid has been deleted while waiting)
E_TMOUT	[k]	Polling failed or timeout
E_RLWAI	[k]	WAITING state is forcibly canceled (rel_wai service call was called in the WAITING state)

Function:

The 4-byte data specified by parameter data is sent to the data queue specified by dtqid.

In addition, when the data queue created by dtqcnt = 0 is specified, service call fsnd_dtq or ifsnd_dtq generates an E_ILUSE error.

(1) When a Task Is Waiting to Receive Data in the Target Data Queue

The data is passed to the head task in the receive-waiting queue and the waiting state of the task is canceled.

(2) When No Task Is Waiting to Receive Data in the Target Data Queue

(a) When the data queue is not full

Parameter data is stored at the end of the data queue. The count of the data queue is incremented by one.

(b) When the data queue is full

— snd_dtq, tsnd_dtq

The calling task is connected to the queue waiting for the data queue to have free space (send-waiting queue).

In service call tsnd_dtq, parameter tmout specifies the timeout period.

If a positive value is specified for 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 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.

The maximum value that can be specified for tmout is (0x7FFFFFFF – TIC_NUME)/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

For details on time management, refer to section 5.13.7, Time Precision.

— psnd_dtq, ipsnd_dtq

In these service calls, error code E_TMOUT is returned immediately.

— fsnd_dtq, ifsnd_dtq

In these service calls, even if no task is waiting to send data in the target data queue, parameter data is stored at the end of the data queue after the data at the head of the data queue (the oldest data) has been deleted.

In service call psnd_dtq, snd_dtq, tsnd_dtq, or fsnd_dtq, data queues belonging to the kernel of another CPU can be specified as dtqid, except for in dispatch-pending state. In service call ipsnd_dtq or ifsnd_dtq, data queues belonging to the kernel of another CPU cannot be specified as dtqid.

6.16.4 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:

dtqid	Data queue ID
p_data	Pointer to the memory area where received data is to be returned
<trcv_dtq>	
tmout	Timeout specification

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (tmout ≤ -2)
E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (dtqid) is invalid) (2) Out of local ID range (GET_LOCALID (dtqid) ≤ 0 or (_MAX_DTQ of GET_CPUID (dtqid)) < GET_LOCALID (dtqid))
E_CTX	[k]	Context error (Called in prohibited system state)
E_NOEXS	[k]	Non-existent object (Data queue indicated by dtqid does not exist)
E_DLT	[k]	Waiting object deleted (Target data queue indicated by dtqid has been deleted while waiting)
E_TMOUT	[k]	Polling failed or timeout
E_RLWAI	[k]	WAITING state is forcibly canceled (rel_wai service call was called in the WAITING state)

Function:

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

If there is data in the data queue, the leading data (the oldest data) is received. On receiving data from the data queue, the data queue count 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 canceled.

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 wait queue to wait for data arrival (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 on a first-in first-out (FIFO) basis.

In service call trcv_dtq, parameter tmout specifies the timeout period.

If a positive value is specified for 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 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.

The maximum value that can be specified for tmout is (0x7FFFFFFF – TIC_NUME)/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

For details on time management, refer to section 5.13.7, Time Precision.

In service call prcv_dtq, rcv_dtq, or trcv_dtq, data queues belonging to the kernel of another CPU can be specified as dtqid, except for in dispatch-pending state.

6.16.5 Reference 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:

dtqid	Data queue ID
pk_rdtq	Pointer to the packet where the data queue state is to be returned

Return Values:

Normal end (E_OK) or error code

Packet Structure:

```
typedef    struct    {
           ID        stskid;      Task ID waiting for sending
           ID        rtskid;      Task ID waiting for receiving
           UINT      sdtqcnt;     The number of data in the data queue
}T_RDTQ;
```

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (dtqid) is invalid) (2) Out of local ID range (GET_LOCALID (dtqid) ≤ 0 or (_MAX_DTQ of GET_CPUID (dtqid)) < GET_LOCALID (dtqid))
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (dtqid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (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.

The CPU ID (1 or 2) where the specified data queue belongs is always set in bits 14 to 12 of a wait task ID.

In the case where a task of another CPU is waiting to send data or receive data from the specified data queue, an SVC server task belonging to the same CPU as the specified data queue will actually wait to perform data transmission/reception with the specified data queue instead of the task of another CPU. Accordingly, the ID of the SVC server task will be returned to the wait task ID in such a case.

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

In service call ref_dtq, data queues belonging to the kernel of another CPU can be specified as dtqid, except for in dispatch-pending state. In service call iref_dtq, data queues belonging to the kernel of another CPU cannot be specified as dtqid.

6.17 Synchronization and Communication (Mailbox)

Mailboxes are controlled by the service calls listed in table 6.19.

Table 6.19 Service Calls for Synchronization and Communication (Mailbox)

Service Call*[1]		Description	System State*[2]						
			T	N	E	D	U	L	C
cre_mbx	[s]	Creates mailbox	O		O	O	O		
icre_mbx				O	O	O	O		
acre_mbx		Creates mailbox and assigns mailbox ID automatically	O		O	O	O		
iacre_mbx				O	O	O	O		
del_mbx		Deletes mailbox	O		O	O	O		
snd_mbx	[B] [S] [R]	Sends data to mailbox	O		O	Δ	O		
isnd_mbx				O	O	O	O		
rcv_mbx	[B] [S] [R]	Receives data from mailbox	O		O		O		
prcv_mbx	[B] [S] [R]	Polls and receives data from mailbox	O		O	Δ	O		
iprcv_mbx				O	O	O	O		
trcv_mbx	[S] [R]	Receives data from mailbox with timeout function	O		O		O		
ref_mbx	[R]	Refers to mailbox state	O		O	Δ	O		
iref_mbx				O	O	O	O		

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

[B]: Basic profile service calls

[R]: Service calls that can be issued remotely

2. T: Can be called in a task context

N: Can be called in a non-task context

E: Can be called in dispatch-enabled state

D: Can be called in dispatch-disabled state

U: Can be called in CPU-unlocked state

L: Can be called in CPU-locked state

C: Can be called while executing the normal CPU exception handler

O: Can be called in the state

∆: Can be called in the state only when a local object is the target

The mailbox specifications are listed in table 6.20.

Table 6.20 Mailbox Specifications

Item	Description
Local mailbox ID	1 to _MAX_MBX (1023 max.)
Message priority	1 to TMAX_MPRI (255 max.)
Mailbox attributes	TA_TFIFO: Wait task queue is managed on a FIFO basis TA_TPRI: Wait task queue is managed on the current priority TA_MFIFO: Message queue is managed on a FIFO basis TA_MPRI: Message queue is managed on the current priority

6.17.1 Create Mailbox (cre_mbx, icre_mbx) (acre_mbx, iacre_mbx: Assign Mailbox ID Automatically)

C-Language API:

```
ER ercd = cre_mbx(ID mbxid, T_CMBX *pk_cmbx);
ER ercd = icre_mbx(ID mbxid, T_CMBX *pk_cmbx);
ER_ID mbxid = acre_mbx(T_CMBX *pk_cmbx);
ER_ID mbxid = iacre_mbx(T_CMBX *pk_cmbx);
```

Parameters:

pk_cmbx	Pointer to the packet where the mailbox creation information is stored
<cre_mbx, icre_mbx>	
mbxid	Mailbox ID

Return Values:

<cre_mbx, icre_mbx>

Normal end (E_OK) or error code

<acre_mbx, iacre_mbx>

Created mailbox ID (a positive value) or error code

Packet Structure:

```
typedef      struct    {
             ATR       mbxatr;     Mailbox attribute
             PRI       maxmpri;    Maximum value of message priority
             VP        mprihd;     Start address of message queue header with
                                   priority
}T_CMBX;
```

Error Codes:

E_RSATR	[p]	Invalid attribute (mbxatr is invalid)
E_PAR	[p]	Parameter error (maxmpri ≤ 0 or maxmpri > TMAX_MPRI of current CPU)
E_ID	[p]	Invalid ID number (cre_mbx, icre_mbx) (1) CPU ID is invalid (GET_CPUID (mbxid) is not the current CPU) (2) Out of local ID range (GET_LOCALID (mbxid) ≤ 0 or (_MAX_MBX of GET_CPUID (mbxid)) < GET_LOCALID (mbxid))
E_OBJ	[k]	Object state is invalid (Mailbox indicated by mbxid already exists) (cre_mbx, icre_mbx)
E_NOID	[k]	No ID available (acre_mbx, iacre_mbx)

Function:

Each of these service calls creates a mailbox.

These service calls can create mailboxes belonging to the kernel of the current CPU. This kernel does not have service calls for creating objects belonging to the kernel of another CPU.

Service calls cre_mbx and icre_mbx create a mailbox with an ID indicated by mbxid. 1 to (_MAX_MBX of current CPU) can be specified for the local ID of mbxid. VCPU_SELF or the current CPU ID must be specified for the CPU ID of mbxid.

Service calls acre_mbx and iacre_mbx search for an unused mailbox ID, create a mailbox with that ID, and return the ID to mbxid. The range searched for the local mailbox ID is 1 to (_MAX_MBX of current CPU). The CPU ID of the mailbox ID that will be returned is the current CPU ID.

Parameter mbxatr specifies the order of the receive-waiting tasks and messages in the wait queues as the attributes.

mbxatr := ((TA_TFIFO || TA_TPRI) | TA_MFIFO || TA_MPRI))

- TA_TFIFO (0x00000000): Message receive-waiting queue is managed on a FIFO basis
- TA_TPRI (0x00000001): Message receive-waiting queue is managed on the current priority
- TA_MFIFO (0x00000000): Message queue is managed on a FIFO basis
- TA_MPRI (0x00000002): Message queue is managed on the current priority

When TA_MPRI is specified for mbxatr, NULL must be specified for mprihd. The message-queue header area is created in the area specified by mprihd when a value other than NULL is specified by the µITRON4.0 specification. However, the kernel does not support a value other than NULL. If a value other than NULL is used, normal system operation cannot be guaranteed. If TA_MPRI is not specified, mprihd does not have any meaning and is simply ignored.

6.17.2 Delete Mailbox (del_mbx)

C-Language API:

```
ER ercd = del_mbx(ID mbxid);
```

Parameters:

mbxid	Mailbox ID

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (mbxid) is not the current CPU) (2) Out of local ID range (GET_LOCALID (mbxid) ≤ 0 or (_MAX_MBX of GET_CPUID (mbxid)) < GET_LOCALID (mbxid))
E_CTX	[k]	Context error (Called in prohibited system state)
E_NOEXS	[k]	Non-existent object (Mailbox indicated by mbxid does not exist)

Function:

Service call del_mbx deletes the mailbox indicated by parameter mbxid.

Only mailboxes belonging to the kernel of the current CPU can be specified as mbxid.

No error will occur even if there is a task waiting for a message in the mailbox indicated by mbxid. However, in that case, the task in the WAITING state will be released and error code E_DLT will be returned. If there is a message in the mailbox, no error will occur, but the kernel will not perform any processing for the message area. For example, the kernel will not automatically return the message area to the memory pool when a memory block acquired from the memory pool is used for a message.

6.17.3 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:

mbxid	Mailbox ID
pk_msg	Start address of the message to be sent

Return Values:

Normal end (E_OK) or error code

Packet Structure:

```
<Mailbox message header>
typedef      struct  {
             VP      msghead;     Kernel management area
}T_MSG;
<Mailbox message header with priority>
typedef      struct  {
             T_MSG   msgque;      Message header
             PRI     msgpri;      Message priority
}T_MSG_PRI;
```

Error Codes:

E_PAR	[p]	Parameter error (the first four bytes of the message is other than 0)
	[k]	(msgpri ≤ 0 or msgpri > TMAX_MPRI of current CPU)
E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (mbxid) is invalid) (2) Out of local ID range (GET_LOCALID (mbxid) ≤ 0 or (_MAX_MBX of GET_CPUID (mbxid)) < GET_LOCALID (mbxid))
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (mbxid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Mailbox indicated by mbxid does not exist)

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 placed at the end of the message queue. The message queue is managed according to the attribute specified at creation.

To send a message to a mailbox that has the TA_MFIFO attribute, the message must have the T_MSG structure at the head of the message, as shown in figure 6.4.

To send a message to a mailbox that has the TA_MPRI attribute, the message must have the T_MSG_PRI structure at the head of the message, as shown in figure 6.5.

Messages must be created in RAM for both the TA_MFIFO and TA_MPRI attributes, and the contents of the T_MSG area must be set to 0 before sending a message.

Note that the T_MSG area is used by the kernel; therefore, the area must not be modified after a message has been sent. After a message is sent, if this area is modified before receiving that message, normal system operation cannot be guaranteed.

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

Figure 6.4 Example of a Message Form

```
typedef struct {
    T_MSG_PRI  t_msg;    /* T_MSG_PRI structure                                   */
    B          data[8];  /* Example of user message data structure (any structure) */
} USER_MSG;
```

Figure 6.5 Example of a Message Form with Priority

In service call snd_mbx, mailboxes belonging to the kernel of another CPU can be specified as mbxid, except for in dispatch-pending state. In this case, messages must be created in a non-cacheable area.

In service call isnd_mbx, mailboxes belonging to the kernel of another CPU cannot be specified as mbxid.

6.17.4 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:

```
mbxid         Mailbox ID
ppk_msg       Pointer to the memory area where the start address of the received
              message is to be returned
<trcv_mbx>
tmout         Timeout specification
```

Return Values:

Normal end (E_OK) or error code

Packet Structure:

```
<Mailbox message header>
typedef        struct  {
               VP      msghead;    Kernel management area
}T_MSG;
<Mailbox message header with priority>
typedef        struct  {
               T_MSG   msgque;     Message header
               PRI     msgpri;     Message priority
}T_MSG_PRI;
```

Error Codes:

```
E_PAR      [p]   Parameter error (tmout ≤ -2)
E_ID       [p]   Invalid ID number
                 (1) CPU ID is invalid (GET_CPUID (mbxid) is invalid)
                 (2) Out of local ID range
                     (GET_LOCALID (mbxid) ≤ 0 or
                     (_MAX_MBX of GET_CPUID (mbxid)) < GET_LOCALID (mbxid))
E_CTX      [k]   Context error (Called in prohibited system state)
E_NOEXS    [k]   Non-existent object (Mailbox indicated by mbxid does not exist)
E_DLT      [k]   Waiting object deleted (Mailbox indicated by mbxid has been
                 deleted in the WAITING state)
E_TMOUT    [k]   Polling failed or timeout
E_RLWAI    [k]   WAITING state is forcibly canceled
                 (rel_wai service call was called 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 creation.

In service call trcv_mbx, parameter tmout specifies the timeout period.

If a positive value is specified for 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 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.

The maximum value that can be specified for tmout is (0x7FFFFFFF – TIC_NUME)/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

For details on time management, refer to section 5.13.7, Time Precision.

In service call prcv_mbx, rcv_mbx, or trcv_mbx, mailboxes belonging to the kernel of another CPU can be specified as mbxid, except for in dispatch-pending state. In service call iprcv_mbx, mailboxes belonging to the kernel of another CPU cannot be specified as mbxid.

6.17.5 Reference 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:

mbxid	Mailbox ID
pk_rmbx	Pointer to the packet where the mailbox state is to be returned

Return Values:

Normal end (E_OK) or error code

Packet Structure:

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

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (mbxid) is invalid) (2) Out of local ID range (GET_LOCALID (mbxid) ≤ 0 or (_MAX_MBX of GET_CPUID (mbxid)) < GET_LOCALID (mbxid))
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (mbxid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (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.

The CPU ID (1 or 2) where the specified mailbox belongs is always set in bits 14 to 12 of a wait task ID.

In the case where a task of another CPU is waiting to send a message or receive a message from the specified mailbox, an SVC server task belonging to the same CPU as the specified mailbox will actually wait to perform message transmission/reception with the specified mailbox instead of the task of another CPU. Accordingly, the ID of the SVC server task will be returned to the wait task ID in such a case.

If there is no task waiting for the specified mailbox, 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.

In service call ref_mbx, mailboxes belonging to the kernel of another CPU can be specified as mbxid, except for in dispatch-pending state. In service call iref_mbx, mailboxes belonging to the kernel of another CPU cannot be specified as mbxid.

6.18 Extended Synchronization and Communication (Mutex)

Mutexes are controlled by the service calls listed in table 6.21.

Table 6.21 Service Calls for Extended Synchronization and Communication (Mutex)

Service Call*1	Description	System State*2						
		T	N	E	D	U	L	C
cre_mtx	Creates mutex	O		O	O	O		
acre_mtx	Creates mutex and assigns mutex ID automatically	O		O	O	O		
del_mtx	Deletes mutex	O		O	O	O		
loc_mtx	Locks mutex	O		O		O		
ploc_mtx	Polls and locks mutex	O		O	O	O		
tloc_mtx	Locks mutex with timeout function	O		O		O		
unl_mtx	Unlocks mutex	O		O	O	O		
ref_mtx	Refers to mutex state	O		O	O	O		

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
[B]: Basic profile service calls
[R]: Service calls that can be issued remotely

2. T: Can be called in a task context
N: Can be called in a non-task context
E: Can be called in dispatch-enabled state
D: Can be called in dispatch-disabled state
U: Can be called in CPU-unlocked state
L: Can be called in CPU-locked state
C: Can be called while executing the normal CPU exception handler
O: Can be called in the state
∆: Can be called in the state only when a local object is the target

The mutex specifications are listed in table 6.22.

Table 6.22 Mutex Specifications

Item	Description
Local mutex ID	1 to _MAX_MTX (1023 max.)
Attribute supported	TA_CEILING: Priority ceiling protocol

Note: This kernel only supports the TA_CEILING attribute (priority ceiling protocol). In this kernel, the mutexes are managed by "simplified priority control rule". Under this rule, the management which changes the task's current priority to a higher value is always done, but the management which changes the task's priority to a lower value is done only when the task releases all of the mutexes.

6.18.1 Create Mutex (cre_mtx) (acre_mtx: Assign Mutex ID Automatically)

C-Language API:

```
ER ercd = cre_mtx(ID mtxid, T_CMTX *pk_cmtx);
ER_ID mtxid = acre_mtx(T_CMTX *pk_cmtx);
```

Parameters:

```
pk_cmtx         Pointer to the packet where the mutex creation information is
                stored
<cre_mtx>
mtxid           Mutex ID
```

Return Values:

```
<cre_mtx>
Normal end (E_OK) or error code
<acre_mtx>
Created mutex ID (a positive value) or error code
```

Packet Structure:

```
typedef    struct    {
           ATR       mtxatr;     Mutex attribute
           PRI       ceilpri;    Ceiling priority of mutex
}T_CMTX;
```

Error Codes:

E_RSATR	[p]	Invalid attribute (mtxatr is invalid)
E_PAR	[p]	Parameter error (ceilpri ≤ 0 or ceilpri > TMAX_TPRI of current CPU)
E_ID	[p]	Invalid ID number (cre_mtx) (1) CPU ID is invalid (GET_CPUID (mtxid) is not the current CPU) (2) Out of local ID range (GET_LOCALID (mtxid) ≤ 0 or (_MAX_MTX of GET_CPUID (mtxid)) < GET_LOCALID (mtxid))
E_OBJ	[k]	Object state is invalid (Mutex indicated by mtxid already exists) (cre_mtx)
E_NOID	[k]	No ID available (acre_mtx, iacre_mtx)

Function:

Each of these service calls creates a mutex.

These service calls can create mutexes belonging to the kernel of the current CPU. This kernel does not have service calls for creating objects belonging to the kernel of another CPU.

Service call cre_mtx creates a mutex with the ID specified by mtxid. 1 to (_MAX_MTX of current CPU) can be specified for the local ID of mtxid. VCPU_SELF or the current CPU ID must be specified for the CPU ID of mtxid.

Service call acre_mtx searches for an unused mutex ID, creates a mutex with that ID, and returns the ID to mtxid. The range searched for the local mutex ID is 1 to (_MAX_MTX of current CPU). The CPU ID of the mutex ID that will be returned is the current CPU ID.

Parameter mtxatr can specify only the priority ceiling protocol (TA_CEILING) as the attribute.

mtxatr := (TA_CEILING)

- TA_CEILING (0x00000003): Priority ceiling protocol

Parameter ceilpri specifies the ceiling priority for the mutex to be created. The range of values which can be specified is 1 to TMAX_TPRI.

6.18.2 Delete Mutex (del_mtx)

C-Language API:

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

Parameters:

```
mtxid           Mutex ID
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_ID          [p]   Invalid ID number
                    (1) CPU ID is invalid (GET_CPUID (mtxid) is not the
                        current CPU)
                    (2) Out of local ID range
                        (GET_LOCALID (mtxid) ≤ 0 or
                        (_MAX_MTX of GET_CPUID (mtxid)) < GET_LOCALID
                        (mtxid))
E_CTX         [k]   Context error (Called in prohibited system state)
E_NOEXS       [k]   Non-existent object (Mutex indicated by mtxid does not
                    exist)
```

Function:

Service call del_mtx deletes the mutex specified by parameter mtxid.

Only mutexes belonging to the kernel of the current CPU can be specified as mtxid.

No error occurs even when there is a lock-waiting task for the mutex specified by mtxid; but the WAITING state of the task is canceled, and E_DLT is returned as an error code.

When the target mutex is locked, the lock for the task locked by the mutex is canceled. As a result, only when all mutexes locking the task are removed, the task priority is returned to base priority.

The task locked by the deleted mutex is not notified that the mutex has been deleted. If an attempt is later made to release the mutex lock, an error is returned.

6.18.3 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:

mtxid	Mutex ID
<tloc_mtx>	
tmout	Timeout specification

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (tmout ≤ −2)
E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (mtxid) is not the current CPU) (2) Out of local ID range (GET_LOCALID (mtxid) ≤ 0 or (_MAX_MTX of GET_CPUID (mtxid)) < GET_LOCALID (mtxid))
E_CTX	[k]	Context error (Called in prohibited system state)
E_ILUSE	[k]	Illegal use of service call (1) The mutex specified by mtxid is already locked by the calling task (2) Base priority of the calling task > Ceiling priority of the target mutex
E_NOEXS	[k]	Non-existent object (Mutex indicated by mtxid does not exist)
E_DLT	[k]	Waiting object deleted (Mutex indicated by mtxid has been deleted in the WAITING state)
E_RLWAI	[k]	The WAITING state was forcibly canceled (rel_wai service call was called 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.

In service call tloc_mtx, parameter tmout specifies the timeout period.

If a positive value is specified for 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 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.

The maximum value that can be specified for tmout is (0x7FFFFFFF – TIC_NUME)/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

For details on time management, refer to section 5.13.7, Time Precision.

6.18.4 Unlock Mutex (unl_mtx)

C-Language API:

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

Parameters:

mtxid	Mutex ID

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (mtxid) is not the current CPU) (2) Out of local ID range (GET_LOCALID (mtxid) ≤ 0 or (_MAX_MTX of GET_CPUID (mtxid)) < GET_LOCALID (mtxid))
E_CTX	[k]	Context error (Called in prohibited system state)
E_ILUSE	[k]	Illegal use of service call (The calling task has not locked the target mutex)
E_NOEXS	[k]	Non-existent object (Mutex indicated by mtxid does not exist)

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.

The simplified priority ceiling protocol is used for the TA_CEILING attribute of this kernel. That is, only when all the mutex that are locked by the task are unlocked, the present priority of the task is returned to a base priority. When the task still locks other mutex after this call, the present priority does not change in this service call.

6.18.5 Reference Mutex State (ref_mtx)

C-Language API:

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

Parameters:

```
mtxid           Mutex ID
pk_rmtx         Pointer to the packet where the mutex state is to be returned
```

Return Values:

```
Normal end (E_OK) or error code
```

Packet Structure:

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

Error Codes:

```
E_ID       [p]   Invalid ID number
                 (1) CPU ID is invalid (GET_CPUID (mtxid) is not the current
                     CPU)
                 (2) Out of local ID range
                     (GET_LOCALID (mtxid) ≤ 0 or
                     (_MAX_MTX of GET_CPUID (mtxid)) < GET_LOCALID (mtxid))
E_NOEXS    [k]   Non-existent object (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.

The CPU ID (1 or 2) where the specified mutex belongs is always set in bits 14 to 12 of htskid and wtskid.

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

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

6.19 Extended Synchronization and Communication (Message Buffer)

Message buffers are controlled by the service calls listed in table 6.23.

Table 6.23 Service Calls for Extended Synchronization and Communication (Message Buffer)

Service Call*[1]	Description	System State*[2]						
		T	N	E	D	U	L	C
cre_mbf	Creates message buffer	O		O	O	O		
icre_mbf			O	O	O	O		
acre_mbf	Creates message buffer and assigns message buffer ID automatically	O		O	O	O		
iacre_mbf			O	O	O	O		
del_mbf	Deletes message buffer	O		O	O	O		
snd_mbf [R]	Sends message to message buffer	O		O		O		
psnd_mbf [R]	Polls and sends message to message buffer	O		O	Δ	O		
ipsnd_mbf			O	O	O	O		
tsnd_mbf [R]	Sends message to message buffer with timeout function	O		O		O		
rcv_mbf [R]	Receives message from message buffer	O		O		O		
prcv_mbf [R]	Polls and receives message from message buffer	O		O	Δ	O		
trcv_mbf [R]	Receives message from message buffer with timeout function	O		O		O		
ref_mbf [R]	Refers to message buffer state	O		O	Δ	O		
iref_mbf			O	O	O	O		

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

[B]: Basic profile service calls

[R]: Service calls that can be issued remotely

2. T: Can be called in a task context
   N: Can be called in a non-task context
   E: Can be called in dispatch-enabled state
   D: Can be called in dispatch-disabled state
   U: Can be called in CPU-unlocked state
   L: Can be called in CPU-locked state
   C: Can be called while executing the normal CPU exception handler
   O: Can be called in the state
   Δ: Can be called in the state only when a local object is the target

The message buffer specifications are listed in table 6.24.

Table 6.24 Message Buffer Specifications

Item	Description
Local message buffer ID	1 to _MAX_MBF (1023 max.)
Message buffer attributes	TA_TFIFO: Wait task queue is managed on a FIFO basis TA_TPRI: Wait task queue is managed on the current priority

6.19.1 Create Message Buffer (cre_mbf, icre_mbf) (acre_mbf, iacre_mbf: Assign Message Buffer ID Automatically)

C-Language API:

```
ER ercd = cre_mbf(ID mbfid, T_CMBF *pk_cmbf);
ER ercd = icre_mbf(ID mbfid, T_CMBF *pk_cmbf);
ER_ID mbfid = acre_mbf(T_CMBF *pk_cmbf);
ER_ID mbfid = iacre_mbf(T_CMBF *pk_cmbf);
```

Parameters:

```
pk_cmbf     Pointer to the packet where the message buffer creation information
            is stored
<cre_mbf, icre_mbf>
mbfid       Message buffer ID
```

Return Values:

```
<cre_mbf, icre_mbf>
Normal end (E_OK) or error code
<acre_mbf, iacre_mbf>
Created message buffer ID (a positive value) or error code
```

Packet Structure:

```
typedef    struct    {
           ATR       mbfatr;      Message buffer attribute
           UINT      maxmsz;      Maximum message size (Number of bytes)
           SIZE      mbfsz;       Message buffer size (Number of bytes)
           VP        mbf;         Start address of message buffer area
}T_CMBF;
```

Error Codes:

```
E_NOMEM    [k]   Insufficient memory (Message buffer area cannot be allocated
                 in the memory)
E_RSATR    [p]   Invalid attribute (mbfatr is invalid)
E_PAR      [p]   Parameter error
                 (1) mbfsz is other than a multiple of four
                 (2) maxmsz = 0
                 (3) mbfsz is other than 0 and maxmsz + 4 > mbfsz
E_ID       [p]   Invalid ID number (cre_mbf, icre_mbf)
                 (1) CPU ID is invalid (GET_CPUID (mbfid) is not the current
                     CPU)
                 (2) Out of local ID range
                     (GET_LOCALID (mbfid) ≤ 0 or
                     (_MAX_MBF of GET_CPUID (mbfid)) < GET_LOCALID (mbfid))
E_OBJ      [k]   Object state is invalid (Message buffer indicated by mbfid
                 already exists) (cre_mbf, icre_mbf)
E_NOID     [k]   No ID available (acre_mbf, iacre_mbf)
```

Function:

Each of these service calls creates a message buffer.

These service calls can create message buffers belonging to the kernel of the current CPU. This kernel does not have service calls for creating objects belonging to the kernel of another CPU.

Service calls cre_mbf and icre_mbf create a message buffer with an ID indicated by mbfid. 1 to (_MAX_MBF of current CPU) can be specified for the local ID of mbfid. VCPU_SELF or the current CPU ID must be specified for the CPU ID of mbfid.

Service calls acre_mbf and iacre_mbf search for an unused message buffer ID, create a message buffer with that ID, and return the ID to mbfid. The range searched for the local message buffer ID is 1 to (_MAX_MBF of current CPU). The CPU ID of the message buffer ID that will be returned is the current CPU ID.

(1) mbfatr

Parameter mbfatr specifies the order of the tasks in the queue waiting for sending a message to the message buffer as the attribute.

mbfatr := (TA_TFIFO || TA_TPRI)

- TA_TFIFO (0x00000000): Task queue waiting for sending a message is managed on a FIFO basis
- TA_TPRI (0x00000001): Task queue waiting for sending a message is managed on the current priority

The message queue and the task queue waiting for receiving a message are managed on a first-in first-out (FIFO) basis regardless of the mbfatr specification.

(2) mbfsz

Parameter mbfsz specifies the size of the message buffer to be created.

The following macro is provided to estimate the approximate size to be specified for mbfsz.

SIZE mbfsz = TSZ_MBF (UINT msgcnt, UINT msgsz)

Approximate size (bytes) of a message buffer area required to store the msgcnt number of msgsz-byte messages

A message buffer of mbfsz = 0 can also be created. In this case, no message can be stored in the message buffer, and the message-receiving task completely synchronizes with the message-sending task. In other words, when a service call to send a message is issued, the task stays in the WAITING state until another task calls a service call to receive a message. Similarly, when a task calls a service call to receive a message the task stays in the WAITING state until another task calls a service call to send a message. Note that for a message buffer with mbfsz = 0, there will be no copying via the message buffer.

(3) maxmsz

Parameter maxmsz specifies the maximum length of a message that can be held in a message buffer.

(4) mbf

Parameter mbf specifies the start address of a free area to be used as a message buffer. An area of mbfsz bytes from mbf is used as the message buffer.

When a value of 0 is specified for mbfsz, mbf does not have any meaning and is simply ignored.

When NULL is specified for mbf, the kernel allocates a message buffer area of mbfsz bytes from the default message buffer area. After that, the size of the free space in the default message buffer area will decrease by an amount given by the following expression:

Decrease in size = mbfsz + 16 bytes

6.19.2 Delete Message Buffer (del_mbf)

C-Language API:

```
ER ercd = del_mbf(ID mbfid);
```

Parameters:

```
mbfid           Message buffer ID
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_ID        [p]   Invalid ID number
                  (1) CPU ID is invalid (GET_CPUID (mbfid) is not the current
                      CPU)
                  (2) Out of local ID range
                      (GET_LOCALID (mbfid) ≤ 0 or
                      (_MAX_MBF of GET_CPUID (mbfid)) < GET_LOCALID (mbfid))
E_CTX       [k]   Context error (Called in prohibited system state)
E_NOEXS     [k]   Non-existent object (Message buffer indicated by mbfid does
                  not exist)
```

Function:

Service call del_mbf deletes the message buffer indicated by parameter mbfid.

Only message buffers belonging to the kernel of the current CPU can be specified as mbfid.

No error will occur even if there is a task waiting for receiving or sending a message in the message buffer indicated by mbfid. However, in that case, the task in the WAITING state will be released and error code E_DLT will be returned. In addition, if there is a message in the message buffer, no error will occur, but all stored messages will be deleted.

On deletion, the size of the free space in the default message buffer area will increase by an amount given by the following expression:

Increase in size = mbfsz specified at creation + 16 bytes

6.19.3 Send Message to Message Buffer (snd_mbf, psnd_mbf, ipsnd_mbf, tsnd_mbf)

C-Language API:

```
ER ercd = snd_mbf(ID mbfid, VP msg, UINT msgsz);
ER ercd = psnd_mbf(ID mbfid, VP msg, UINT msgsz);
ER ercd = ipsnd_mbf(ID mbfid, VP msg, UINT msgsz);
ER ercd = tsnd_mbf(ID mbfid, VP msg, UINT msgsz, TMO tmout);
```

Parameters:

mbfid	Message buffer ID
msg	Start address of the message to send
msgsz	Size of the message to send (number of bytes)
<tsnd_mbf>	
tmout	Timeout specification

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (msgsz = 0 or tmout ≤ −2)
	[k]	(msgsz > maxmsz*)
E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (mbfid) is invalid) (2) Out of local ID range (GET_LOCALID (mbfid) ≤ 0 or (_MAX_MBF of GET_CPUID (mbfid)) < GET_LOCALID (mbfid))
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (mbfid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Message buffer indicated by mbfid does not exist)
E_DLT	[k]	Waiting object deleted (Message buffer indicated by mbfid has been deleted during the WAITING state)
E_TMOUT	[k]	Polling failed or timeout
E_RLWAI	[k]	WAITING state is forcibly canceled (rel_wai service call was called in the WAITING state)

Note: * maxmsz: Maximum length of a message specified at message buffer creation

Function:

Each service call sends a message specified by msg to the message buffer specified by mbfid. The message size is specified by parameter msgsz.

If there is a task waiting to receive a message, the message sent by the service call is not placed in the message buffer. Instead, the message is passed to the task at the head of the receive wait queue, releasing the task from the WAITING state.

If there are already tasks waiting to send a message to the message buffer, the task that called service call snd_mbf or tsnd_mbf is placed in the queue to wait for free space in the message buffer (send-waiting queue). With service calls psnd_mbf and ipsnd_mbf, error code E_TMOUT is immediately returned. The send-waiting queue is managed according to the attribute specified at task creation.

If there are no tasks waiting to send or receive a message, the message sent from a task is stored in the message buffer. After that, the size of the free space in the default message buffer area will decrease by an amount given by the following expression:

Decrease in size = msgsz + 4 bytes

However, if the free space in the message buffer is less than the above size (including when the buffer size is 0), the task that issued the service call is placed in the send-waiting queue.

ipsnd_mbf can also be issued from a non-task context. Since the priority of a non-task context is higher than that of a task, when the target message buffer has the TA_TPRI attribute and the buffer has enough free space, the specified message is copied to the buffer even if there exists a task that has been waiting to be transmitted.

In service call tsnd_mbf, parameter tmout specifies the timeout period.

If a positive value is specified for 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 service call psnd_mbf will be performed.

If tmout = TMO_FEVR (–1) is specified, the same operation as for service call snd_mbf will be performed. In other words, timeout monitoring is not performed.

The maximum value that can be specified for tmout is (0x7FFFFFFF – TIC_NUME)/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

For details on time management, refer to section 5.13.7, Time Precision.

In service call snd_mbf, psnd_mbf, or tsnd_mbf, message buffers belonging to the kernel of another CPU can be specified as mbfid, except for in dispatch-pending state. In this case, messages must be created in a non-cacheable area.

In service call ipsnd_mbf, message buffers belonging to the kernel of another CPU cannot be specified as mbfid.

6.19.4 Receive Message from Message Buffer (rcv_mbf, prcv_mbf, trcv_mbf)

C-Language API:

```
ER_UINT msgsz = rcv_mbf(ID mbfid, VP msg);
ER_UINT msgsz = prcv_mbf(ID mbfid, VP msg);
ER_UINT msgsz = trcv_mbf(ID mbfid, VP msg, TMO tmout);
```

Parameters:

mbfid	Message buffer ID
msg	Pointer to the memory area where the received message is to be stored
<trcv_mbf>	
tmout	Timeout specification

Return Values:

Size of the received message (number of bytes, a positive value) or error code

Error Codes:

E_PAR	[p]	Parameter error (tmout ≤ -2)
E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (mbfid) is invalid) (2) Out of local ID range (GET_LOCALID (mbfid) ≤ 0 or (_MAX_MBF of GET_CPUID (mbfid)) < GET_LOCALID (mbfid))
E_CTX	[k]	Context error (Called in prohibited system state)
E_NOEXS	[k]	Non-existent object (Message buffer indicated by mbfid does not exist)
E_DLT	[k]	Waiting object deleted (Target message buffer indicated by mbfid has been deleted during the WAITING state)
E_TMOUT	[k]	Polling failed or timeout
E_RLWAI	[k]	WAITING state is forcibly canceled (rel_wai service call was called in the WAITING state)

Function:

Each service call receives a message from the message buffer specified by parameter mbfid and stores the received message in the area indicated by msg. The received message size is returned as the return parameter.

If there are already messages in the message buffer, the task receives the message at the head of the queue (the oldest message). After the message in the message buffer has been received, the size of the free space in the message buffer will increase by an amount given by the following expression:

Increase in size = msgsz + 4 bytes

If, as a result, the free space in the message buffer becomes larger than the size of the message to be sent by the task at the head of the send-waiting queue, the message is sent and stored in the message buffer and the task is released from the WAITING state. The same will be done for the remaining tasks in the order of the send-waiting queue if the message can be stored.

If there are no messages in the message buffer and there are tasks waiting to send a message, the message of the task at the head of the send-waiting queue is received by the service call. As a result, the task is released from the WAITING state.

If there are no messages in the message buffer and there are no tasks in the queue to send a message, the task that called service call rcv_mbf or trcv_mbf is placed in the queue to wait to receive a message (receive-waiting queue). With service call prcv_mbf, error code E_TMOUT is immediately returned. The receive-waiting queue is managed on a first-in first-out (FIFO) basis.

Parameter msg points to a RAM area whose size is specified by maxmsz by service call cre_mbf, icre_mbf, acre_mbf, or iacre_mbf.

In service call trcv_mbf, parameter tmout specifies the timeout period.

If a positive value is specified for 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 service call prcv_mbf 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_mbf will be performed.

The maximum value that can be specified for tmout is (0x7FFFFFFF – TIC_NUME)/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

For details on time management, refer to section 5.13.7, Time Precision.

In these service calls, message buffers belonging to the kernel of another CPU can be specified as mbfid, except for in dispatch-pending state. In this case, a non-cacheable area must be specified as the area pointed to by parameter msg.

6.19.5 Reference Message Buffer State (ref_mbf, iref_mbf)

C-Language API:

```
ER ercd = ref_mbf(ID mbfid, T_RMBF *pk_rmbf);
ER ercd = iref_mbf(ID mbfid, T_RMBF *pk_rmbf);
```

Parameters:

```
mbfid           Message buffer ID
pk_rmbf         Pointer to the packet where the message buffer state is to be
                returned
```

Return Values:

```
Normal end (E_OK) or error code
```

Packet Structure:

```
typedef    struct    {
           ID        stskid;    Start task ID of the queue waiting to send a
                                message
           ID        rtskid;    Start task ID of the queue waiting to receive
                                a message
           UINT      smsgcnt;   Number of messages in message buffer
           SIZE      fmbfsz;    Size of free buffer (Number of bytes)
}T_RMBF;
```

Error Codes:

```
E_ID       [p]   Invalid ID number
                 (1) CPU ID is invalid (GET_CPUID (mbfid) is invalid)
                 (2) Out of local ID range
                     (GET_LOCALID (mbfid) ≤ 0 or
                     (_MAX_MBF of GET_CPUID (mbfid)) < GET_LOCALID (mbfid))
E_CTX      [k]   Context error (Called in prohibited system state when
                 GET_CPUID (mbfid) is not the current CPU)
```

Function:

Each service call refers to the state of the message buffer indicated by parameter mbfid and returns the task ID of the task waiting to send a message (stskid), task waiting to receive a message (rtskid), the size of the next message to be received (smsgcnt), and the available free buffer size (fmbfsz) to the area indicated by pk_rmbf.

The CPU ID (1 or 2) where the specified message buffer belongs is always set in bits 14 to 12 of stskid and rtskid.

If no task is waiting to receive or send a message, TSK_NONE (0) is returned as a wait task ID.

In service call ref_mbf, message buffers belonging to the kernel of another CPU can be specified as mbfid, except for in dispatch-pending state. In service call iref_mbf, message buffers belonging to the kernel of another CPU cannot be specified as mbfid.

6.20 Memory Pool Management (Fixed-Sized Memory Pool)

Fixed-sized memory pools are controlled by the service calls listed in table 6.25.

Table 6.25 Service Calls for Memory Pool Management (Fixed-Sized Memory Pool)

Service Call*1	Description	System State*2						
		T	N	E	D	U	L	C
cre_mpf [s]	Creates fixed-sized memory pool	O		O	O	O		
icre_mpf			O	O	O	O		
acre_mpf	Creates fixed-sized memory pool and assigns fixed-sized memory pool ID automatically	O		O	O	O		
iacre_mpf			O	O	O	O		
del_mpf	Deletes fixed-sized memory pool	O		O	O	O		
get_mpf [B] [S] [R]	Acquires fixed-sized memory block	O		O		O		
pget_mpf [B] [S] [R]	Polls and acquires fixed-sized memory block	O		O	Δ	O		
ipget_mpf			O	O	O	O		
tget_mpf [S] [R]	Acquires fixed-sized memory block with timeout function	O		O		O		
rel_mpf [B] [S] [R]	Releases fixed-sized memory block	O		O	Δ	O		
irel_mpf			O	O	O	O		
ref_mpf [R]	Refers to fixed-sized memory pool state	O		O	Δ	O		
iref_mpf			O	O	O	O		

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
[B]: Basic profile service calls
[R]: Service calls that can be issued remotely

2. T: Can be called in a task context
   N: Can be called in a non-task context
   E: Can be called in dispatch-enabled state
   D: Can be called in dispatch-disabled state
   U: Can be called in CPU-unlocked state
   L: Can be called in CPU-locked state
   C: Can be called while executing the normal CPU exception handler
   O: Can be called in the state
   Δ: Can be called in the state only when a local object is the target

The fixed-sized memory pool specifications are listed in table 6.26.

Table 6.26 Fixed-Sized Memory Pool Specifications

Item	Description
Local fixed-sized memory pool ID	1 to _MAX_MPF (1023 max.)*
Fixed-sized memory pool attributes	TA_TFIFO: Wait task queue is managed on a FIFO basis TA_TPRI: Wait task queue is managed on the current priority
Management method	Whether to place management information in the memory pool area can be chosen with system.mpfmanage in the cfg file.

Note: * Though the maximum value of _MAX_MPF is 1023, the maximum value that can be specified for maxdefine.max_mpf in the cfg file is 1022.

6.20.1 Create Fixed-Sized Memory Pool (cre_mpf, icre_mpf) (acre_mpf, iacre_mpf: Assign Memory Pool ID Automatically)

C-Language API:

```
ER ercd = cre_mpf(ID mpfid, T_CMPF *pk_cmpf);
ER ercd = icre_mpf(ID mpfid, T_CMPF *pk_cmpf);
ER_ID mpfid = acre_mpf(T_CMPF *pk_cmpf);
ER_ID mpfid = iacre_mpf(T_CMPF *pk_cmpf);
```

Parameters:

```
pk_cmpf          Pointer to the packet where the fixed-sized memory pool
                 creation information is stored
<cre_mpf, icre_mpf>
mpfid                                  Fixed-sized memory pool ID
```

Return Values:

```
<cre_mpf, icre_mpf>
Normal end (E_OK) or error code
<acre_mpf, iacre_mpf>
Created fixed-sized memory pool ID (a positive value) or error code
```

Packet Structure:

```
(1) system.mpfmanage is IN
     typedef   struct   {
               ATR      mpfatr;    Fixed-sized memory pool attribute
               UINT     blkcnt;    Number of blocks in memory pool
               UINT     blksz;     Block size of fixed-sized memory pool (Number of
                                   bytes)
               VP       mpf;       Start address of the fixed-sized memory pool
                                   area
     }T_CMPF;
(2) system.mpfmanage is OUT
     typedef   struct   {
               ATR      mpfatr;    Fixed-sized memory pool attribute
               UINT     blkcnt;    Number of blocks in memory pool
               UINT     blksz;     Block size of fixed-sized memory pool (Number of
                                   bytes)
               VP       mpf;       Start address of the fixed-sized memory pool
                                   area
               VP       mpfmb;     Start address of the fixed-sized memory block
                                   management area
     }T_CMPF;
```

Error Codes:

E_NOMEM	[k]	Insufficient memory (Memory pool area cannot be allocated in the memory)
E_RSATR	[p]	Invalid attribute (mpfatr is invalid)
E_PAR	[p]	Parameter error (blkcnt = 0, blksz is other than a multiple of four, or blksz = 0)
	[k]	TSZ_MPF (blkcnt, blksz) exceeds the 32-bit range
E_ID	[p]	Invalid ID number (cre_mpf, icre_mpf) (1) CPU ID is invalid (GET_CPUID (mpfid) is not the current CPU) (2) Out of local ID range (GET_LOCALID (mpfid) ≤ 0 or (_MAX_MPF of GET_CPUID (mpfid)) < GET_LOCALID (mpfid))
E_OBJ	[k]	Object state is invalid (Fixed-sized memory pool indicated by mpfid already exists) (cre_mpf, icre_mpf)
E_NOID	[k]	No ID available (acre_mpf, iacre_mpf)

Function:

Each of these service calls creates a fixed-sized memory pool.

These service calls can create fixed-sized memory pools belonging to the kernel of the current CPU. This kernel does not have service calls for creating objects belonging to the kernel of another CPU.

Service calls cre_mpf and icre_mpf create a fixed-sized memory pool with an ID indicated by mpfid. 1 to (_MAX_MPF of current CPU) can be specified for the local ID of mpfid. VCPU_SELF or the current CPU ID must be specified for the CPU ID of mpfid.

Service calls acre_mpf and iacre_mpf search for an unused fixed-sized memory pool ID, create a fixed-sized memory pool with that ID, and return the ID to mpfid. The range searched for the local fixed-sized memory pool ID is 1 to (_MAX_MPF of current CPU). The CPU ID of the fixed-sized memory pool ID that will be returned is the current CPU ID.

Parameter mpfatr specifies the order of the tasks in the queue waiting to acquire a memory block as the attribute.

mpfatr := (TA_TFIFO || TA_TPRI)

- TA_TFIFO (0x00000000): Task queue waiting to acquire a memory block is managed on a FIFO basis
- TA_TPRI (0x00000001): Task queue waiting to acquire a memory block is managed by the current priority

Parameter blkcnt specifies the total number of memory blocks to be created.

The size of the memory block to be created is specified by blksz, and must be a multiple of four.

When NULL is specified for mpf, the kernel automatically allocates a fixed-sized memory pool. This fixed-sized memory pool will be allocated from the default fixed-sized memory pool area specified by the configurator. After that, the size of the free space in the default fixed-sized memory pool area will decrease by an amount given by the following expression:

Decrease in size = TSZ_MPF(blkcnt, blksz) + 16 bytes

The following macro is provided to estimate the approximate size to be specified for mpfsz.

SIZE TSZ_MPF(UINT blkcnt, UINT blksz)

Approximate size (bytes) of a fixed-sized memory pool area required to store the blkcnt number of blksz-byte memory blocks

Note that the definition of the TSZ_MPF() macro differs depending on the system.mpfmanage setting as follows.

(1) system.mpfmanage is IN

TSZ_MPF (blkcnt, blksz) = (blksz + 4 bytes) × blkcnt

(2) system.mpfmanage is OUT

TSZ_MPF (blkcnt, blksz) = blksz × blkcnt

The address of the allocated fixed-sized memory pool can be specified as mpf. In this case, allocate an area whose size is calculated by TSZ_MPF (blkcnt, blksz), and specify the address as mpf.

If system.mpfmanage is OUT, the start address for the fixed-sized memory block management area must be specified as mpfmb. In this case, allocate an area whose size is calculated by VTSZ_MPFMB (blkcnt, blksz), and specify the address as mpfmb.

mpfmb is a member not defined in the μITRON4.0 specification.

If there is a possibility that another CPU will access the memory block acquired from the fixed-sized memory pool, the fixed-sized memory pool area must be in a non-cached area. If, in particular, there is a possibility that another CPU will access the memory block acquired from the fixed-sized memory pool which was allocated from the default fixed-sized memory pool area, the section (BC_himpf) for the default fixed-sized memory pool area must be placed at a non-cacheable area at linkage.

6.20.2 Delete Fixed-Sized Memory Pool (del_mpf)

C-Language API:

```
ER ercd = del_mpf(ID mpfid);
```

Parameters:

```
mpfid           Fixed-sized memory pool ID
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_ID        [p]   Invalid ID number
                  (1) CPU ID is invalid (GET_CPUID (mpfid) is not the current
                      CPU)
                  (2) Out of local ID range
                      (GET_LOCALID (mpfid) ≤ 0 or
                      (_MAX_MPF of GET_CPUID (mpfid)) < GET_LOCALID (mpfid))
E_CTX       [k]   Context error (Called in prohibited system state)
E_NOEXS     [k]   Non-existent object (Fixed-sized memory pool indicated by
                  mpfid does not exist)
```

Function:

Service call del_mpf deletes the fixed-sized memory pool indicated by mpfid.

Only fixed-sized memory pools belonging to the kernel of the current CPU can be specified as mpfid.

No error will occur even if there is a task waiting to acquire a memory block in the fixed-sized memory pool area indicated by mpfid. However, in that case, the task in the WAITING state will be released and error code E_DLT will be returned.

When the fixed-sized memory pool allocated from the default fixed-sized memory pool area is deleted (NULL is specified for mpf at creation), the size of the free space in the default fixed-sized memory pool area will increase by an amount given by the following expression:

Increase in size
= TSZ_MPF (blkcnt specified at creation, blksz specified at creation) + 16 bytes

The kernel will not perform any processing even when a block has already been acquired.

6.20.3 Get Fixed-Sized 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:

mpfid	Fixed-sized memory pool ID
p_blk	Pointer to the memory area where the start address of the memory block is to be returned
<tget_mpf>	
tmout	Timeout specification

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (tmout ≤ -2)
E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (mpfid) is invalid) (2) Out of local ID range (GET_LOCALID (mpfid) ≤ 0 or (_MAX_MPF of GET_CPUID (mpfid)) < GET_LOCALID (mpfid))
E_CTX	[k]	Context error (Called in prohibited system state)
E_NOEXS	[k]	Non-existent object (Fixed-sized memory pool indicated by mpfid does not exist)
E_DLT	[k]	Waiting object deleted (Fixed-sized memory pool indicated by mpfid has been deleted)
E_TMOUT	[k]	Polling failed or timeout
E_RLWAI	[k]	WAITING state was forcibly canceled (rel_wai service call was called in the WAITING state)

Function:

Each service call gets one fixed-sized memory block from the fixed-sized 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-sized memory pool, the task having called service call get_mpf or tget_mpf is placed in the queue for waiting to acquire a memory block, and the task having called service call pget_mpf or ipget_mpf is immediately returned with error code E_TMOUT. The queue is managed according to the attribute specified at creation.

In service call tget_mpf, parameter tmout specifies the timeout period.

If a positive value is specified for 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 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.

The maximum value that can be specified for tmout is (0x7FFFFFFF – TIC_NUME)/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

For details on time management, refer to section 5.13.7, Time Precision.

In service call pget_mpf, get_mpf, or tget_mpf, fixed-sized memory pools belonging to the kernel of another CPU can be specified as mpfid, except for in dispatch-pending state. In service call ipget_mpf, fixed-sized memory pools belonging to the kernel of another CPU cannot be specified as mpfid.

6.20.4 Release Fixed-Sized Memory Block (rel_mpf, irel_mpf)

C-Language API:

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

Parameters:

mpfid	Fixed-sized memory pool ID
blk	Start address of memory block

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (blk is other than a multiple or four)
	[k]	(blk is other than the start address of the memory block or blk has already been released)
E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (mpfid) is invalid) (2) Out of local ID range (GET_LOCALID (mpfid) ≤ 0 or (_MAX_MPF of GET_CPUID (mpfid)) < GET_LOCALID (mpfid))
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (mpfid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Fixed-sized memory pool indicated by mpfid does not exist)

Function:

Each service call returns the memory block indicated by blk to the fixed-sized 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-sized memory pool, the memory block released by this service call is passed to the task at the head of the wait queue, releasing it from the WAITING state.

In service call rel_mpf, fixed-sized memory pools belonging to the kernel of another CPU can be specified as mpfid, except for in dispatch-pending state. In service call irel_mpf, fixed-sized memory pools belonging to the kernel of another CPU cannot be specified as mpfid.

6.20.5 Reference Fixed-Sized 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:

```
mpfid          Fixed-sized memory pool ID
pk_rmpf        Pointer to the packet where the fixed-sized memory pool state
               is to be returned
```

Return Values:

```
Normal end (E_OK) or error code
```

Packet Structure:

```
typedef    struct    {
           ID        wtskid;     Wait task ID
           UINT      fblkcnt;    Number of available blocks
}T_RMPF;
```

Error Codes:

```
E_ID        [p]   Invalid ID number
                  (1) CPU ID is invalid (GET_CPUID (mpfid) is invalid)
                  (2) Out of local ID range
                      (GET_LOCALID (mpfid) ≤ 0 or
                      (_MAX_MPF of GET_CPUID (mpfid)) < GET_LOCALID (mpfid))
E_CTX       [k]   Context error (Called in prohibited system state when
                  GET_CPUID (mpfid) is not the current CPU)
E_NOEXS     [k]   Non-existent object (Fixed-sized memory pool indicated by
                  mpfid does not exist)
```

Function:

Each service call refers to the status of the fixed-sized memory pool indicated by mpfid.

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

The CPU ID (1 or 2) where the specified fixed-sized memory pool belongs is always set in bits 14 to 12 of a wait task ID.

In the case where a task of another CPU is waiting to acquire a memory block in the specified fixed-sized memory pool, an SVC server task belonging to the same CPU as the specified fixed-sized memory pool will actually wait to acquire a memory block in the specified fixed-sized memory pool instead of the task of another CPU. Accordingly, the ID of the SVC server task will be returned to the wait task ID in such a case.

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

In service call ref_mpf, fixed-sized memory pools belonging to the kernel of another CPU can be specified as mpfid, except for in dispatch-pending state. In service call iref_mpf, fixed-sized memory pools belonging to the kernel of another CPU cannot be specified as mpfid.

6.21 Memory Pool Management (Variable-Sized Memory Pool)

Variable-sized memory pools are controlled by the service calls listed in table 6.27.

Table 6.27 Service Calls for Memory Pool Management (Variable-Sized Memory Pool)

Service Call*1	Description	System State*2						
		T	N	E	D	U	L	C
cre_mpl	Creates variable-sized memory pool	O		O	O	O		
icre_mpl			O	O	O	O		
acre_mpl	Creates variable-sized memory pool and assigns variable-sized memory pool ID automatically	O		O	O	O		
iacre_mpl			O	O	O	O		
del_mpl	Deletes variable-sized memory pool	O		O	O	O		
get_mpl [R]	Acquires variable-sized memory block	O		O		O		
pget_mpl [R]	Polls and acquires variable-sized memory block	O		O	Δ	O		
ipget_mpl			O	O	O	O		
tget_mpl [R]	Acquires variable-sized memory block with timeout function	O		O		O		
rel_mpl [R]	Releases variable-sized memory block	O		O	Δ	O		
irel_mpl			O	O	O	O		
ref_mpl [R]	Refers to variable-sized memory pool state	O		O	Δ	O		
iref_mpl			O	O	O	O		

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
[B]: Basic profile service calls
[R]: Service calls that can be issued remotely

2. T: Can be called in a task context
N: Can be called in a non-task context
E: Can be called in dispatch-enabled state
D: Can be called in dispatch-disabled state
U: Can be called in CPU-unlocked state
L: Can be called in CPU-locked state
C: Can be called while executing the normal CPU exception handler
O: Can be called in the state
∆: Can be called in the state only when a local object is the target

The variable-sized memory pool specifications are listed in table 6.28.

Table 6.28 Variable-Sized Memory Pool Specifications

Item	Description
Local variable-sized memory pool ID	1 to _MAX_MPL (1023 max.)
Management method	Selecting NEW for system.newmpl in the cfg file improves the following: • Acquisition and release of memory blocks becomes faster when a large number of memory blocks are used in the memory pool. • The degree of fragmentation is reduced. • The VTA_UNFRAGMENT attribute can be used to further reduce fragmentation of the free space.
Variable-sized memory pool attributes	TA_TFIFO: Wait task queue is managed on a FIFO basis VTA_UNFRAGMENT: Sector management

The free space in the variable-sized memory pool may be fragmented. Also refer to section 5.12.1, Controlling Memory Fragmentation.

6.21.1 Create Variable-Sized Memory Pool (cre_mpl, icre_mpl) (acre_mpl, iacre_mpl: Assign Variable-Sized Memory Pool ID Automatically)

C-Language API:

```
ER ercd = cre_mpl(ID mplid, T_CMPL *pk_cmpl);
ER ercd = icre_mpl(ID mplid, T_CMPL *pk_cmpl);
ER_ID mplid = acre_mpl(T_CMPL *pk_cmpl);
ER_ID mplid = iacre_mpl(T_CMPL *pk_cmpl);
```

Parameters:

```
pk_cmpl     Pointer to the packet where the variable-sized memory pool
            creation information is stored
<cre_mpl, icre_mpl>
mplid       Variable-sized memory pool ID
```

Return Values:

```
<cre_mpl, icre_mpl>
Normal end (E_OK) or error code
<acre_mpl, iacre_mpl>
Created variable-sized memory pool ID (a positive value) or error code
```

Packet Structure:

```
(1) system.newmpl is PAST
     typedef    struct    {
                ATR       mplatr;     Variable-sized memory pool attribute
                SIZE      mplsz;      Size of memory pool (Number of bytes)
                VP        mpl;        Start address of the variable-sized memory
                                      pool area
     }T_CMPL;
(2) system.newmpl is NEW
     typedef    struct    {
                ATR       mplatr;     Variable-sized memory pool attribute
                SIZE      mplsz;      Size of memory pool (Number of bytes)
                VP        mpl;        Start address of the variable-sized memory
                                      pool area
                VP        mplmb;      Start address of the variable-sized memory
                                      block management area
                UINT      minblksz;   Minimum block size
                UINT      sctnum;     Maximum sector number
     }T_CMPL;
```

Error Codes:

E_NOMEM	[k]	Insufficient memory (Memory pool area cannot be allocated in the memory)
E_RSATR	[p]	Invalid attribute (mplatr is invalid)
E_PAR	[p]	Parameter error (1) mplsz is other than a multiple of four (2) mplsz < TSZ_MPL(1, 4) (3) mplsz ≥ 0x80000000 (4) minblksz = 0 for the VTA_UNFRAGMENT attribute (5) sctnum = 0 for the VTA_UNFRAGMENT attribute (6) mplsz < minblksz*32 for the VTA_UNFRAGMENT attribute
E_ID	[p]	Invalid ID number (cre_mpl, icre_mpl) (1) CPU ID is invalid (GET_CPUID (mplid) is not the current CPU) (2) Out of local ID range (GET_LOCALID (mplid) ≤ 0 or (_MAX_MPL of GET_CPUID (mplid)) < GET_LOCALID (mplid))
E_OBJ	[k]	Object state is invalid (Variable-sized memory pool indicated by mplid already exists) (cre_mpl, icre_mpl)
E_NOID	[k]	No ID available (acre_mpl, iacre_mpl)

Function:

Each of these service calls creates a variable-sized memory pool.

These service calls can create variable-sized memory pools belonging to the kernel of the current CPU. This kernel does not have service calls for creating objects belonging to the kernel of another CPU.

Service calls cre_mpl and icre_mpl create a variable-sized memory pool with an ID indicated by mplid. 1 to (_MAX_MPL of current CPU) can be specified for the local ID of mplid. VCPU_SELF or the current CPU ID must be specified for the CPU ID of mplid.

Service calls acre_mpl and iacre_mpl search for an unused variable-sized memory pool ID, create a variable-sized memory pool with that ID, and return the ID to mplid. The range searched for the local variable-sized memory pool ID is 1 to (_MAX_MPL of current CPU). The CPU ID of the variable-sized memory pool ID that will be returned is the current CPU ID.

(1) mplatr

Specify the logical OR of the following values for mplatr.

(a) Order of tasks in the queue for waiting for memory block acquisition

Only TA_TFIFO can be specified.

— TA_TFIFO (0x00000000): Task queue waiting for memory is managed on a FIFO basis

(b) Management method

When NEW has been specified for system.newmpl in the cfg file, the VTA_UNFRAGMENT attribute can be specified.

— VTA_UNFRAGMENT (0x80000000): Sector management (reducing fragmentation in free space)

The VTA_UNFRAGMENT attribute is suitable for a memory pool from which a large number of small memory blocks are to be acquired. When this attribute is specified, small blocks are collectively allocated in specialized contiguous areas to keep larger possible contiguous areas.

Only when the VTA_UNFRAGMENT attribute is specified, mplmb, minblksz, and sctnum become valid. When sctnum is set to a larger value than mplsz/(minblksz × 32), mplsz/(minblksz × 32) is assumed.

For details, refer to section 5.12.1, Controlling Memory Fragmentation.

(2) mplsz

Parameter mplsz specifies the size of the variable-sized memory pool to be created. Also refer to section 5.12.2, Management of Variable-Sized Memory Pools.

The following macro is provided to estimate the approximate size to be specified for mplsz.

SIZE mplsz = TSZ_MPL (UINT blkcnt, UINT blksz)

Approximate size (bytes) of a variable-sized memory pool area required to store the blkcnt number of blksz-byte memory blocks

This macro calculates the size assuming that the VTA_UNFRAGMENT attribute is not selected. The expression for calculating the size depends on the system.newmpl setting.

(3) mpl

Parameter mpl specifies the start address of a free area to be used as a variable-sized memory pool. The kernel allocates an mplsz-byte area starting from address mpl as a variable-sized memory pool.

When NULL is specified for mpl, the kernel allocates an mplsz-byte area from the default variable-sized memory pool area. After that, the size of the free space in the default variable-sized memory pool area will decrease by an amount given by the following expression:

Decrease in size = mplsz + 16 bytes

If there is a possibility that another CPU will access the memory block acquired from the variable-sized memory pool, the variable-sized memory pool area must be in a non-cacheable area. If, in particular, there is a possibility that another CPU will access the memory block acquired from the variable-sized memory pool which was allocated from the default variable-sized memory pool area, the section (BC_himpl) for the default variable-sized memory pool area must be placed at a non-cacheable area at linkage.

(4) mplmb

mplmb is a member not defined in the μITRON4.0 specification.

Parameter mplmb is only valid when the VTA_UNFRAGMENT attribute is specified; it is ignored in other cases.

Allocate an area for the size calculated by the following macro, and specify the start address of the area as mplmb.

VTSZ_MPLMB (maximum number of sectors)

(5) minblksz and sctnum

These members are not defined in the μITRON4.0 specification.

These parameters are valid only when the VTA_UNFRAGMENT attribute is specified. For details, refer to section 5.12.2, Management of Variable-Sized Memory Pools.

Supplement:

The address of a memory block is aligned with a 4-byte boundary.

To align the address of a memory block with the cache line size (16 or 32), allocate the area as follows (N means the alignment size).

- When NEW is specified for system.newmpl and VTA_UNFRAGMENT is not specified
  - — Allocate a memory pool area to the N-byte boundary address by the application, and specify that address when creating a memory pool.
  - — Specify a multiple of N as the size of every memory block to be acquired.
- When NEW is specified for system.newmpl and VTA_UNFRAGMENT is specified
  - — Allocate a memory pool area to the N-byte boundary address by the application, and specify that address when creating a memory pool.
  - — Specify N for the minimum block size.
  - — Specify a multiple of N as the size of every memory block to be acquired.
- When PAST is specified for system.newmpl

  (a) Alignment when N = 16

  - Allocate a memory pool area to the 16-byte boundary address by the application, and specify that address when creating a memory pool.
  - Specify a multiple of 16 as the size of every memory block to be acquired.

  (b) Alignment when N = 32

  - Allocate a memory pool area to the address obtained by (32-byte boundary address – 16) by the application, and specify that address when creating a memory pool.
  - Specify (a multiple of N + 16) as the size of every memory block to be acquired.

6.21.2 Delete Variable-Sized Memory Pool (del_mpl)

C-Language API:

```
ER ercd = del_mpl(ID mplid);
```

Parameters:

```
mplid           Variable-sized memory pool ID
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_ID        [p]   Invalid ID number
                  (1) CPU ID is invalid (GET_CPUID (mplid) is not the current
                      CPU)
                  (2) Out of local ID range
                      (GET_LOCALID (mplid) ≤ 0 or
                      (_MAX_MPL of GET_CPUID (mplid)) < GET_LOCALID (mplid))
E_CTX       [k]   Context error (Called in prohibited system state)
E_NOEXS     [k]   Non-existent object (Variable-sized memory pool indicated by
                  mplid does not exist)
```

Function:

Service call del_mpl deletes the variable-sized memory pool specified by mplid.

Only variable-sized memory pools belonging to the kernel of the current CPU can be specified as mplid.

No error will occur even if there is a task waiting to acquire a memory block in the variable-sized memory pool specified by mplid. However, in that case, the task in the WAITING state will be released and error code E_DLT will be returned.

When the variable-sized memory pool allocated from the default variable-sized memory pool area is deleted (NULL is specified for mpl at creation), the size of the free space in the default variable-sized memory pool area will increase by an amount given by the following expression:

Increase in size = (mplsz specified at creation) + 16 bytes

The kernel will not perform any processing even when a block has already been acquired.

6.21.3 Get Variable-Sized 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, TMO tmout);
```

Parameters:

mplid	Variable-sized memory pool ID
blksz	Memory block size (Number of bytes)
p_blk	Pointer to the memory area where the start address of the memory block is to be returned
<tget_mpl>	
tmout	Timeout specification

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (blksz is other than a multiple of four or 0, or tmout ≤ –2)
	[k]	(mplsz* – 16 < blksz)
E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (mplid) is invalid) (2) Out of local ID range (GET_LOCALID (mplid) ≤ 0 or (_MAX_MPL of GET_CPUID (mplid)) < GET_LOCALID (mplid))
E_CTX	[k]	Context error (Called in prohibited system state)
E_NOEXS	[k]	Non-existent object (Variable-sized memory pool indicated by mplid does not exist)
E_DLT	[k]	Waiting object deleted (The memory pool specified by mplid has been deleted)
E_TMOUT	[k]	Polling failed or timeout
E_RLWAI	[k]	WAITING state was forcibly canceled (rel_wai service call was called in the WAITING state)

Note: * mplsz: Memory pool size specified at variable-sized memory pool creation

Function:

Each service call acquires a variable-sized memory block with the size specified by blksz (number of bytes) from the variable-sized 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 free space in the variable-sized memory pool will decrease. For details, refer to section 5.12.2, Management of Variable-Sized Memory Pools.

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.

In service call tget_mpl, parameter tmout specifies the timeout period.

If a positive value is specified for 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 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.

The maximum value that can be specified for tmout is (0x7FFFFFFF – TIC_NUME)/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

For details on time management, refer to section 5.13.7, Time Precision.

In service call get_mpl, pget_mpl, or tget_mpl, variable-sized memory pools belonging to the kernel of another CPU can be specified as mplid, except for in dispatch-pending state. In service call ipget_mpl, variable-sized memory pools belonging to the kernel of another CPU cannot be specified as mplid.

6.21.4 Release Variable-Sized Memory Block (rel_mpl, irel_mpl)

C-Language API:

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

Parameters:

mplid	Variable-sized memory pool ID
blk	Start address of memory block

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_PAR	[k]	Parameter error (blk is other than the start address of the memory block or blk has already been released)
E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (mplid) is invalid) (2) Out of local ID range (GET_LOCALID (mplid) ≤ 0 or (_MAX_MPL of GET_CPUID (mplid)) < GET_LOCALID (mplid))
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (mplid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Variable-sized memory pool indicated by mplid does not exist)

Function:

Each service call returns the memory block specified by blk to the variable-sized 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 released, the size of the free space in the variable-sized memory pool will increase. For details, refer to section 5.12.2, Management of Variable-Sized Memory Pools.

After the memory block has been released, if the target variable-sized memory pool has a contiguous free area of the size requested by the task at the head of the memory block acquisition wait queue, a memory block is assigned to that task and 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 still has enough contiguous free space.

In service call rel_mpl, variable-sized memory pools belonging to the kernel of another CPU can be specified as mplid, except for in dispatch-pending state. In service call irel_mpl, variable-sized memory pools belonging to the kernel of another CPU cannot be specified as mplid.

6.21.5 Reference Variable-Sized 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:

```
mplid           Variable-sized memory pool ID
pk_rmpl         Pointer to the packet where the variable-sized memory pool
                state is to be returned
```

Return Values:

```
Normal end (E_OK) or error code
```

Packet Structure:

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

Error Codes:

```
E_ID        [p]   Invalid ID number
                  (1) CPU ID is invalid (GET_CPUID (mplid) is invalid)
                  (2) Out of local ID range
                      (GET_LOCALID (mplid) ≤ 0 or
                      (_MAX_MPL of GET_CPUID (mplid)) < GET_LOCALID (mplid))
E_CTX       [k]   Context error (Called in prohibited system state when
                  GET_CPUID (mplid) is not the current CPU)
E_NOEXS     [k]   Non-existent object (Variable-sized memory pool indicated by
                  mplid does not exist)
```

Function:

Each service call refers to the status of the variable-sized memory pool indicated by mplid.

Service calls ref_mpl and iref_mpl return the wait task ID (wtskid), the total size of the current free space (fmplsz), and the maximum memory block size available (fblksz) to the area indicated by pk_rmpl.

The CPU ID (1 or 2) where the specified variable-sized memory pool belongs is always set in bits 14 to 12 of a wait task ID.

In the case where a task of another CPU is waiting to acquire a memory block in the specified variable-sized memory pool, an SVC server task belonging to the same CPU as the specified variable-sized memory pool will actually wait to acquire a memory block in the specified variable-sized memory pool instead of the task of another CPU. Accordingly, the ID of the SVC server task will be returned to the wait task ID in such a case.

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

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

In service call ref_mpl, variable-sized memory pools belonging to the kernel of another CPU can be specified as mplid, except for in dispatch-pending state. In service call iref_mpl, variable-sized memory pools belonging to the kernel of another CPU cannot be specified as mplid.

6.22 Time Management (System Clock)

The system clock is controlled by the service calls listed in table 6.29.

Table 6.29 Service Calls for System Clock Management

Service Call*[1]	Description	System State*[2]						
		T	N	E	D	U	L	C
set_tim [S]	Sets system clock	O		O	O	O		
iset_tim			O	O	O	O		
get_tim [S]	Gets system clock	O		O	O	O		
iget_tim			O	O	O	O		
isig_tim [S]	Supplies time tick	(Automatically executed by specifying TIMER for clock.timer)						
vstp_tmr	Stops the timer	O		O	O	O		
vrst_tmr	Restarts the timer	O		O	O	O		
ivrst_tmr			O	O	O	O		
vsns_tmr	Refers to timer state	O	O	O	O	O	O	O

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

[B]: Basic profile service calls

[R]: Service calls that can be issued remotely

2. T: Can be called in a task context
N: Can be called in a non-task context
E: Can be called in dispatch-enabled state
D: Can be called in dispatch-disabled state
U: Can be called in CPU-unlocked state
L: Can be called in CPU-locked state
C: Can be called while executing the normal CPU exception handler
O: Can be called in the state
∆: Can be called in the state only when a local object is the target

The system clock management specifications are listed in table 6.30.

Table 6.30 System Clock Management Specifications

Item	Description
System clock value	Unsigned 48 bits
System clock unit	1 [ms]
System clock update cycle	TIC_NUME/TIC_DENO [ms]
System clock initial value (at initialization)	0x000000000000

The system clock is expressed as a 48-bit unsigned integer by using a structure of data type “SYSTIM”. The maximum value of the system clock is shown below.

- When TIC_NUME/TIC_DENO ≤ 1:
  Maximum value = 0x7FFFFFFFFFFF/TIC_DENO
- When TIC_NUME/TIC_DENO > 1:
  Maximum value = 0x7FFFFFFFFFFF

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

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

6.22.1 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:

```
p_systim   Pointer to the packet where the system clock to be set is
           indicated
```

Return Values:

```
Normal end (E_OK) or error code
```

Packet Structure:

```
typedef   struct   {
          UH       utime;     Current time data (upper)
          UW       ltime;     Current time data (lower)
}SYSTIM;
```

Function:

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

If a value larger than 1 is specified for TIC_DENO (the denominator for time tick cycles), the maximum value that can be specified is 0x7FFFFFFFFFFF/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

For details on time management, refer to section 5.13.7, Time Precision.

6.22.2 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:

```
p_systim      Pointer to the packet where the system clock is to be returned
```

Return Values:

```
Normal end (E_OK) or error code
```

Packet Structure:

```
typedef     struct   {
            UH       utime;      Current time data (upper)
            UW       ltime;      Current time data (lower)
}SYSTIM;
```

Function:

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

6.22.3 Supply Time Tick (isig_tim)

Function:

Service call isig_tim updates the system clock.

When TIMER is specified for clock.timer in the cfg file, the system is configured such that service call isig_tim is executed automatically in cycles of TIC_NUME/TIC_DENO [ms]. In other words, this function is not a service call, and so cannot be issued from an application.

When a time tick is supplied, the kernel performs the following time-related processing.

(1) Calling of timer driver interrupt processing function (tdr_int_tmr())

(2) Update of system clock (+1)

(3) Update of profile counters

(4) Startup of time event handler

(5) Timeout processing for tasks in the WAITING state due to service calls with a timeout, such as tslp_tsk

In order to use kernel functions related to time, the timer driver must be included. For details, refer to section 12.9, Timer Drivers.

6.22.4 Stop Timer (vstp_tmr)

C-Language API:

```
ER ercd = vstp_tmr(RELTIM limit);
```

Parameters:

limit	Upper time limit

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_PAR	[p]	Parameter error (0x80000000 ≤ limit ≤ 0xFFFFFFFE)
E_OBJ	[k]	Object state is invalid (1) The given period until the nearest timer event is less than limit (2) Kernel timer is already stopped
E_CTX	[k]	Context error (Called in prohibited system state)

Function:

Service call vstp_tmr stops the kernel timer.

However, in case the period until occurrence of the nearest timer event (timeout by txxx_yyy, delay by dly_tsk, cyclic handler, or alarm handler) is less than parameter limit, an E_OBJ error will be returned.

A value from 1 to 0x7FFFFFFF should be specified for limit.

When 0xFFFFFFFF is specified for limit, the timer can be stopped only when there are no timer events.

When 0 is specified for limit, the timer can be stopped even if there are timer events.

While the kernel timer is stopped, time-related processing that is supposed to be performed by the kernel results as follows:

(1) The system clock is not updated.

(2) Timer events, such as a cyclic handler and timeout of a task, do not occur. If service call tslp_tsk or sta_alm, which creates a new timer event is issued, an E_OBJ error will be returned.

(3) Counting of the task overrun monitor period is stopped.

(4) The profile counters are not updated.

In this service call, tdr_stp_tmr() of the timer driver is called back to stop operation of the timer hardware.

This service call is normally used for also stopping timer interrupt processing when the CPU is shifted to SLEEP mode.

This service call is a function not defined in the µITRON4.0 specification.

6.22.5 Restart Timer (vrst_tmr, ivrst_tmr)

C-Language API:

```
ER ercd = vrst_tmr(RELTIM eratim);
ER ercd = ivrst_tmr(RELTIM eratim);
```

Parameters:

```
eratim         Elapsed time
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_OBJ       [k]   Object state is invalid (Kernel timer is not stopped)
```

Function:

Each service call restarts operation of the kernel timer as if the period specified by eratim has elapsed from the point at where the kernel timer was stopped.

When a value other than 0 is specified for eratim, a timer event may occur.

In these service calls, tdr_rst_tmr() of the timer driver is called back to restart operation of the timer hardware.

These service calls are functions not defined in the µITRON4.0 specification.

6.22.6 Reference Timer State (vsns_tmr)

C-Language API:

```
BOOL  state = vsns_tmr(void);
```

Return Values:

TRUE is returned when the kernel timer is stopped and FALSE is returned when the kernel timer is operating

Function:

Service call vsns_tmr checks whether the kernel timer is stopped.

This service call can be issued in the CPU-locked state and from the normal CPU exception handler.

This service call is a function not defined in the μITRON4.0 specification.

6.23 Time Management (Cyclic Handler)

Cyclic handlers are controlled by the service calls listed in table 6.31.

Table 6.31 Service Calls for Cyclic Handler

Service Call*[1]	Description	System State*[2]						
		T	N	E	D	U	L	C
cre_cyc [s]	Creates cyclic handler	O		O	O	O		
icre_cyc			O	O	O	O		
acre_cyc	Creates cyclic handler and assigns cyclic handler ID automatically	O		O	O	O		
iacre_cyc			O	O	O	O		
del_cyc	Deletes cyclic handler	O		O	O	O		
sta_cyc [B] [S] [R]	Starts cyclic handler operation	O		O	∆	O		
ista_cyc			O	O	O	O		
stp_cyc [B] [S] [R]	Stops cyclic handler operation	O		O	∆	O		
istp_cyc			O	O	O	O		
ref_cyc [R]	Refers to the cyclic handler state	O		O	∆	O		
iref_cyc			O	O	O	O		

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
[B]: Basic profile service calls
[R]: Service calls that can be issued remotely

2. T: Can be called in a task context
N: Can be called in a non-task context
E: Can be called in dispatch-enabled state
D: Can be called in dispatch-disabled state
U: Can be called in CPU-unlocked state
L: Can be called in CPU-locked state
C: Can be called while executing the normal CPU exception handler
O: Can be called in the state
∆: Can be called in the state only when a local object is the target

The cyclic handler specifications are listed in table 6.32.

Table 6.32 Cyclic Handler Specifications

Item	Description
Local cyclic handler ID	1 to _MAX_CYH (15 max.)*
Cyclic handler attributes	TA_HLNG: Written in a high-level language TA_ASM: Written in assembly language TA_STA: Cyclic handler makes a transition to the operating state after it has been created TA_PHS: Reserves initiation phase

Note: * Though the maximum value of _MAX_CYH is 15, the maximum value that can be specified for maxdefine.max_cyh in the cfg file is 14.

6.23.1 Create Cyclic Handler (cre_cyc, icre_cyc) (acre_cyc, iacre_cyc: Assign Cyclic Handler ID Automatically)

C-Language API:

```
ER ercd = cre_cyc(ID cycid, T_CCYC *pk_ccyc);
ER ercd = icre_cyc(ID cycid, T_CCYC *pk_ccyc);
ER_ID cycid = acre_cyc(T_CCYC *pk_ccyc);
ER_ID cycid = iacre_cyc(T_CCYC *pk_ccyc);
```

Parameters:

pk_ccyc	Pointer to the packet where the cyclic handler creation information is stored
<cre_cyc, icre_cyc>	
cycid	Cyclic handler ID

Return Values:

<cre_cyc, icre_cyc>

Normal end (E_OK) or error code

<acre_cyc, iacre_cyc>

Created cyclic handler ID number (a positive value) or error code

Packet Structure:

```
typedef    struct    {
           ATR       cycatr;    Cyclic handler attribute
           VP_INT    exinf;     Extended information
           FP        cychdr;    Cyclic handler address
           RELTIM    cyctim;    Cyclic handler initiation cycle
           RELTIM    cycphs;    Cyclic handler initiation phase
}T_CCYC;
```

Error Codes:

E_RSATR	[p]	Invalid attribute (cycatr is invalid)
E_PAR	[p]	Parameter error (cyctim = 0 or cycphs > cyctim)
E_ID	[p]	Invalid ID number (cre_cyc, icre_cyc) (1) CPU ID is invalid (GET_CPUID (cycid) is not the current CPU) (2) Out of local ID range (GET_LOCALID (cycid) ≤ 0 or (_MAX_CYH of GET_CPUID (cycid)) < GET_LOCALID (cycid))
E_OBJ	[k]	Object state is invalid (Cyclic handler specified by cycid already exists) (cre_cyc, icre_cyc)
E_NOID	[k]	No ID available (acre_cyc, iacre_cyc)

Function:

Each of these service calls creates a cyclic handler.

These service calls can create cyclic handlers belonging to the kernel of the current CPU. This kernel does not have service calls for creating objects belonging to the kernel of another CPU.

Service calls cre_cyc and icre_cyc create a cyclic handler with an ID indicated by cycid. 1 to (_MAX_CYH of current CPU) can be specified for the local ID of cycid. VCPU_SELF or the current CPU ID must be specified for the CPU ID of cycid.

Service calls acre_cyc and iacre_cyc search for an unused cyclic handler ID, create a cyclic handler with that ID, and return the ID to cycid. The range searched for the local cyclic handler ID is 1 to (_MAX_CYH of current CPU). The CPU ID of the cyclic handler ID that will be returned is the current CPU ID.

The cyclic handler is a time event handler for a non-task context and is initiated at specified time intervals.

Parameter cycatr specifies the language in which the handler was written and the attribute at initiation as the attributes.

cycatr := ((TA_HLNG ‖ TA_ASM) | [TA_STA] | [TA_PHS])

- TA_HLNG (0x00000000): Written in a high-level language
- TA_ASM (0x00000001): Written in assembly language
- TA_STA (0x00000002): Cyclic handler makes a transition to the operating state after it has been created
- TA_PHS (0x00000004): Preserves the initiation phase

When TA_STA is specified, the cyclic handler makes a transition to the operating state after it has been created. When TA_STA is not specified, the cyclic handler does not operate until service call sta_cyc or ista_cyc is issued. When TA_PHS is specified, the initiation phase of the cyclic handler is kept before activating the cyclic handler, and the next time to initiate the handler is determined. When TA_PHS is not specified, the next time to initiate the cyclic handler is determined based on the time that service call sta_cyc or ista_cyc is issued.

Parameter exinf specifies the extended information to be passed as a parameter when initiating the cyclic handler. Parameter exinf can be widely used by the user, for example, to set information concerning cyclic handlers to be defined.

Parameter cychdr specifies the start address of the cyclic handler.

Parameter cyctim specifies the handler initiation state.

Parameter cycphs specifies the handler initiation phase.

The first time to initiate the cyclic handler occurs after cycphs (initiation phase) has passed since the service call that creates the cyclic handler has been issued. The cyclic handler is then initiated at every cyctim (initiation interval).

The maximum value that can be specified for cyctim and cycphs is (0x7FFFFFFF – TIC_NUME)/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

For details on time management, refer to section 5.13.7, Time Precision.

6.23.2 Delete Cyclic Handler (del_cyc)

C-Language API:

```
ER ercd = del_cyc(ID cycid);
```

Parameters:

```
cycid            Cyclic handler ID
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_ID        [p]   Invalid ID number
                  (1) CPU ID is invalid (GET_CPUID (cycid) is not the current
                      CPU)
                  (2) Out of local ID range
                      (GET_LOCALID (cycid) ≤ 0 or
                      (_MAX_CYH of GET_CPUID (cycid)) < GET_LOCALID (cycid))
E_CTX       [k]   Context error (Called in prohibited system state)
E_NOEXS     [k]   Non-existent object (Cyclic handler specified by cycid does
                  not exist)
```

Function:

Service call del_cyc deletes the cyclic handler specified by parameter cycid.

Only cyclic handlers belonging to the kernel of the current CPU can be specified as cycid.

6.23.3 Start Cyclic Handler (sta_cyc, ista_cyc)

C-Language API:

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

Parameters:

cycid	Cyclic handler ID

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (cycid) is invalid) (2) Out of local ID range (GET_LOCALID (cycid) ≤ 0 or (_MAX_CYH of GET_CPUID (cycid)) < GET_LOCALID (cycid))
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (cycid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Cyclic handler specified by cycid does not exist)

Function:

Each service call causes the cycle handler specified by cycid to enter the operating 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 issued.

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

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 initiation of the cyclic handler is based on the timing at which the handler has been started.

In service call sta_cyc, cyclic handlers belonging to the kernel of another CPU can be specified as cycid, except for in dispatch-pending state. In service call ista_cyc, cyclic handlers belonging to the kernel of another CPU cannot be specified as cycid.

6.23.4 Stop Cyclic Handler (stp_cyc, istp_cyc)

C-Language API:

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

Parameters:

```
cycid            Cyclic handler ID
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_ID        [p]   Invalid ID number
                  (1) CPU ID is invalid (GET_CPUID (cycid) is invalid)
                  (2) Out of local ID range
                      (GET_LOCALID (cycid) ≤ 0 or
                      (_MAX_CYH of GET_CPUID (cycid)) < GET_LOCALID (cycid))
E_CTX       [k]   Context error (Called in prohibited system state when
                  GET_CPUID (cycid) is not the current CPU)
E_NOEXS     [k]   Non-existent object (Cyclic handler specified by cycid does
                  not exist)
```

Function:

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

In service call stp_cyc, cyclic handlers belonging to the kernel of another CPU can be specified as cycid, except for in dispatch-pending state. In service call istp_cyc, cyclic handlers belonging to the kernel of another CPU cannot be specified as cycid.

6.23.5 Reference 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:

```
cycid           Cyclic handler ID
pk_rcyc         Pointer to the packet where the cyclic handler state is to
                be returned
```

Return Values:

```
Normal end (E_OK) or error code
```

Packet Structure:

```
typedef    struct    {
           STAT      cycstat;     Cyclic handler operating state
           RELTIM    lefttim;     Remaining time until the cyclic handler is
                                  initiated
}T_RCYC;
```

Error Codes:

```
E_ID       [p]   Invalid ID number
                 (1) CPU ID is invalid (GET_CPUID (cycid) is invalid)
                 (2) Out of local ID range
                    (GET_LOCALID (cycid) ≤ 0 or
                    (_MAX_CYH of GET_CPUID (cycid)) < GET_LOCALID (cycid))
E_CTX      [k]   Context error (Called in prohibited system state when
                 GET_CPUID (cycid) is not the current CPU)
E_NOEXS    [k]   Non-existent object (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 operating state (cycstat) and the time remaining until the cyclic handler is initiated (lefttim), to the area indicated by parameter pk_rcyc.

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

- TCYC_STP (0x00000000): The cyclic handler is not in the operating state
- TCYC_STA (0x00000001): 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.

In service call ref_cyc, cyclic handlers belonging to the kernel of another CPU can be specified as cycid, except for in dispatch-pending state. In service call iref_cyc, cyclic handlers belonging to the kernel of another CPU cannot be specified as cycid.

6.24 Time Management (Alarm Handler)

Alarm handlers are controlled by the service calls listed in table 6.33.

Table 6.33 Service Calls for Alarm Handler

Service Call*1	Description	System State*2						
		T	N	E	D	U	L	C
cre_alm	Creates alarm handler	O		O	O	O		
icre_alm			O	O	O	O		
acre_alm	Creates alarm handler and assigns alarm handler ID automatically	O		O	O	O		
iacre_alm			O	O	O	O		
del_alm	Deletes alarm handler	O		O	O	O		
sta_alm [R]	Starts alarm handler operation	O		O	Δ	O		
ista_alm			O	O	O	O		
stp_alm [R]	Stops alarm handler operation	O		O	Δ	O		
istp_alm			O	O	O	O		
ref_alm [R]	Refers to the alarm handler state	O		O	Δ	O		
iref_alm			O	O	O	O		

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

[B]: Basic profile service calls

[R]: Service calls that can be issued remotely

2. T: Can be called in a task context
N: Can be called in a non-task context
E: Can be called in dispatch-enabled state
D: Can be called in dispatch-disabled state
U: Can be called in CPU-unlocked state
L: Can be called in CPU-locked state
C: Can be called while executing the normal CPU exception handler
O: Can be called in the state
Δ: Can be called in the state only when a local object is the target

The alarm handler specifications are listed in table 6.34.

Table 6.34 Alarm Handler Specifications

Item	Description
Local alarm handler ID	1 to _MAX_ALH (15 max.)
Alarm handler attributes	TA_HLNG: Written in a high-level language TA_ASM: Written in assembly language

6.24.1 Create Alarm Handler (cre_alm, icre_alm) (acre_alm, iacre_alm: Assign Alarm Handler ID Automatically)

C-Language API:

```
ER ercd = cre_alm(ID almid, T_CALM *pk_calm);
ER ercd = icre_alm(ID almid, T_CALM *pk_calm);
ER_ID almid = acre_alm(T_CALM *pk_calm);
ER_ID almid = iacre_alm(T_CALM *pk_calm);
```

Parameters:

```
pk_calm          Pointer to the packet where the alarm handler creation
                 information is stored
<cre_alm, icre_alm>
almid            Alarm handler ID
```

Return Values:

```
<cre_alm, icre_alm>
Normal end (E_OK) or error code
<acre_alm, iacre_alm>
Created alarm handler ID (a positive value) or error code
```

Packet Structure:

```
typedef      struct      {
             ATR         almatr;      Alarm handler attribute
             VP_INT      exinf;       Extended information
             FP          almhdr;      Alarm handler address
}T_CALM;
```

Error Codes:

E_RSATR	[p]	Invalid attribute (almatr is invalid)
E_ID	[p]	Invalid ID number (cre_alm, icre_alm) (1) CPU ID is invalid (GET_CPUID (almid) is not the current CPU) (2) Out of local ID range (GET_LOCALID (almid) ≤ 0 or (_MAX_ALH of GET_CPUID (almid)) < GET_LOCALID (almid))
E_OBJ	[k]	Object state is invalid (Alarm handler specified by almid already exists) (cre_alm, icre_alm)
E_NOID	[k]	No ID available (acre_alm, iacre_alm)

Function:

Each of these service calls creates an alarm handler.

These service calls can create alarm handlers belonging to the kernel of the current CPU. This kernel does not have service calls for creating objects belonging to the kernel of another CPU.

Service calls cre_alm and icre_alm create an alarm handler with an ID indicated by almid. 1 to (_MAX_ALH of current CPU) can be specified for the local ID of almid. VCPU_SELF or the current CPU ID must be specified for the CPU ID of almid.

Service calls acre_alm and iacre_alm search for an unused alarm handler ID, create an alarm handler with that ID, and return the ID to almid. The range searched for the local alarm handler ID is 1 to (_MAX_ALH of current CPU). The CPU ID of the alarm handler ID that will be returned is the current CPU ID.

The alarm handler is a time event handler for a non-task context and is initiated at the specified time only once.

Parameter almatr specifies the language in which the handler was written as the attribute.

almatr := (TA_HLNG || TA_ASM)

- TA_HLNG (0x00000000): Written in a high-level language
- TA_ASM (0x00000001): Written in assembly language

Parameter exinf specifies extended information to be returned as a parameter when initiating the alarm handler. Parameter exinf can be widely used by the user, for example, to set information concerning alarm handlers to be defined.

Parameter almhdr specifies the start address of the alarm handler.

The time to initiate the alarm handler is not set immediately after creating the alarm handler. The alarm handler is in the stopped state.

6.24.2 Delete Alarm Handler (del_alm)

C-Language API:

```
ER ercd = del_alm(ID almid);
```

Parameters:

```
almid           Alarm handler ID
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_ID         [p]   Invalid ID number
                   (1) CPU ID is invalid (GET_CPUID (almid) is not the current
                       CPU)
                   (2) Out of local ID range
                       (GET_LOCALID (almid) ≤ 0 or
                       (_MAX_ALH of GET_CPUID (almid)) < GET_LOCALID (almid))
E_CTX        [k]   Context error (Called in prohibited system state)
E_NOEXS      [k]   Non-existent object (Alarm handler specified by almid does
                   not exist)
```

Function:

Service call del_alm deletes the alarm handler specified by parameter almid.

Only alarm handlers belonging to the kernel of the current CPU can be specified as almid.

6.24.3 Start Alarm Handler (sta_alm, ista_alm)

C-Language API:

```
ER ercd = sta_alm(ID almid, RELTIM almtim);
ER ercd = ista_alm(ID almid, RELTIM almtim);
```

Parameters:

almid	Alarm handler ID
almtim	Alarm handler initiation time

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (almid) is invalid) (2) Out of local ID range (GET_LOCALID (almid) ≤ 0 or (_MAX_ALH of GET_CPUID (almid)) < GET_LOCALID (almid))
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (almid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Alarm handler specified by almid does not exist)

Function:

The starting time for the alarm handler specified by almid is set to the relative time specified by almtim after the moment at which the service call is issued, to start operation of the alarm handler.

If a time is set for an alarm handler already in operation, the previous starting time setting is canceled, and the new starting time is set.

If almtim is set to 0, the alarm handler is started at the next time tick.

The maximum value that can be specified for almtim is (0x7FFFFFFF – TIC_NUME)/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

For details on time management, refer to section 5.13.7, Time Precision.

In service call sta_alm, alarm handlers belonging to the kernel of another CPU can be specified as almid, except for in dispatch-pending state. In service call ista_alm, alarm handlers belonging to the kernel of another CPU cannot be specified as almid.

6.24.4 Stop Alarm Handler (stp_alm, istp_alm)

C-Language API:

```
ER ercd = stp_alm (ID almid);
ER ercd = istp_alm (ID almid);
```

Parameters:

almid	Alarm handler ID

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (almid) is invalid) (2) Out of local ID range (GET_LOCALID (almid) ≤ 0 or (_MAX_ALH of GET_CPUID (almid)) < GET_LOCALID (almid))
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (almid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Alarm handler specified by almid does not exist)

Function:

Each service call releases the alarm handler initiation time indicated by parameter almid, and stops alarm handler operation.

In service call stp_alm, alarm handlers belonging to the kernel of another CPU can be specified as almid, except for in dispatch-pending state. In service call istp_alm, alarm handlers belonging to the kernel of another CPU cannot be specified as almid.

6.24.5 Reference Alarm Handler State (ref_alm, iref_alm)

C-Language API:

```
ER ercd = ref_alm(ID almid, T_RALM *pk_ralm);
ER ercd = iref_alm(ID almid, T_RALM *pk_ralm);
```

Parameters:

almid	Alarm handler ID
pk_ralm	Pointer to the packet where the alarm handler state is to be returned

Return Values:

Normal end (E_OK) or error code

Packet Structure:

```
typedef    struct   {
           STAT     almstat;      Alarm handler operating state
           RELTIM   lefttim;      Remaining time until the alarm handler is
                                  initiated
}T_RALM;
```

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (almid) is invalid) (2) Out of local ID range (GET_LOCALID (almid) ≤ 0 or (_MAX_ALH of GET_CPUID (almid)) < GET_LOCALID (almid))
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (almid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Alarm handler specified by almid does not exist)

Function:

Each service call reads the alarm handler state indicated by almid and returns the alarm handler operating state (almstat) and remaining time until the alarm handler is initiated (lefttim) to the area indicated by parameter pk_ralm.

The target alarm handler operating state is returned to parameter almstat.

- TALM_STP (0x00000000): The alarm handler is not in the operating state
- TALM_STA (0x00000001): The alarm handler is in the operating state

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

In service call ref_alm, alarm handlers belonging to the kernel of another CPU can be specified as almid, except for in dispatch-pending state. In service call iref_alm, alarm handlers belonging to the kernel of another CPU cannot be specified as almid.

6.25 Time Management (Overrun Handler)

Overrun handler is controlled by the service calls listed in table 6.35.

Table 6.35 Service Calls for Overrun Handler

Service Call*1	Description	System State*2						
		T	N	E	D	U	L	C
def_ovr	Defines overrun handler	O		O	O	O		
sta_ovr [R]	Starts overrun handler operation	O		O	Δ	O		
ista_ovr			O	O	O	O		
stp_ovr [R]	Stops overrun handler operation	O		O	Δ	O		
istp_ovr			O	O	O	O		
ref_ovr [R]	Refers to overrun handler state	O		O	Δ	O		
iref_ovr			O	O	O	O		

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
[B]: Basic profile service calls
[R]: Service calls that can be issued remotely
2. T: Can be called in a task context
N: Can be called in a non-task context
E: Can be called in dispatch-enabled state
D: Can be called in dispatch-disabled state
U: Can be called in CPU-unlocked state
L: Can be called in CPU-locked state
C: Can be called while executing the normal CPU exception handler
O: Can be called in the state
Δ: Can be called in the state only when a local object is the target

Only one overrun handler can be defined in the system. The overrun handler is a time event handler.

The processor time used by the task includes the execution times of a task, the service calls issued by the task, and the interrupt handler that is initiated during execution of the task. Used processor time is not counted while the task is not in the RUNNING state.

The overrun handler specifications are listed in table 6.36.

Table 6.36 Overrun Handler Specifications

Item	Description
Processor time unit (OVRTIM)	Same as system clock (1 [ms])
Overrun handler attributes	TA_HLNG: Written in a high-level language TA_ASM: Written in assembly language

6.25.1 Define Overrun Handler (def_ovr)

C-Language API:

```
ER ercd = def_ovr(T_DOVR *pk_dovr);
```

Parameters:

```
pk_dovr          Pointer to the packet where the overrun handler definition
                 information is stored
```

Return Values:

```
Normal end (E_OK) or error code
```

Packet Structure:

```
typedef      struct   {
             ATR      ovratr;     Overrun handler attribute
             FP       ovrhdr;     Overrun handler address
}T_DOVR;
```

Error Codes:

```
E_RSATR     [p]   Invalid attribute (ovratr is invalid)
```

Function:

The overrun handler is defined using the contents specified by pk_dovr for the current CPU.

The overrun handler is a time event handler for a non-task context which is started when the processor is used by a task for a time exceeding a preset time.

Parameter ovratr specifies the language in which the handler was written as the attribute.

ovratr := (TA_HLNG || TA_ASM)

- TA_HLNG (0x00000000): Written in a high-level language
- TA_ASM (0x00000001): Written in assembly language

Parameter ovrhdr specifies the start address of the overrun handler.

When pk_dovr = NULL (0) is specified in service call def_ovr, the overrun handler definition is canceled.

When an overrun handler has already been defined, if this service call is issued, the preceding definition is canceled and the new definition takes its place.

6.25.2 Start Overrun Handler (sta_ovr, ista_ovr)

C-Language API:

```
ER ercd = sta_ovr(ID tskid, OVRTIM ovrtim);
ER ercd = ista_ovr(ID tskid, OVRTIM ovrtim);
```

Parameters:

tskid	Task ID
ovrtim	Processing time limit

Return Values:

Normal end (E_OK) or error code

Error Codes:

E_ID	[p]	Invalid ID number (1) CPU ID is invalid (GET_CPUID (tskid) is invalid) (2) Out of local ID range (GET_LOCALID (tskid) < 0 or (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid)) (3) tskid = TSK_SELF (0) when called in a non-task context
E_CTX	[k]	Context error (Called in prohibited system state when GET_CPUID (tskid) is not the current CPU)
E_NOEXS	[k]	Non-existent object (Task specified by tskid does not exist)
E_OBJ	[k]	Object state is invalid (Overrun handler has not been defined)

Function:

Overrun handler operation begins for the task specified by tskid.

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

The processing time limit for the task is set to the time specified by ovrtim, and the processor time used is cleared to 0. If the overrun handler has already been operating, the previously set processing time limit is canceled, and the new processing time limit is set.

When the processor time used exceeds the processing time limit, the overrun handler is started.

If 0 is specified for ovrtim, the overrun handler is started on the first time tick after the task begins to use the processor.

The maximum value that can be specified for ovrtim is (0x7FFFFFFF – TIC_NUME)/TIC_DENO. If a value larger than this is specified, operation is not guaranteed.

For details on time management, refer to section 5.13.7, Time Precision.

In service call sta_ovr, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call ista_ovr, a task belonging to the kernel of another CPU cannot be specified as tskid.

6.25.3 Stop Overrun Handler (stp_ovr, istp_ovr)

C-Language API:

```
ER ercd = stp_ovr(ID tskid);
ER ercd = istp_ovr(ID tskid);
```

Parameters:

```
tskid              Task ID
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_ID        [p]     Invalid ID number
                    (1) CPU ID is invalid (GET_CPUID (tskid) is invalid)
                    (2) Out of local ID range
                        (GET_LOCALID (tskid) < 0 or
                        (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid))
                    (3) tskid = TSK_SELF (0) when called in a non-task context
E_CTX       [k]     Context error (Called in prohibited system state when
                    GET_CPUID (tskid) is not the current CPU)
E_NOEXS     [k]     Non-existent object (Task specified by tskid does not
                    exist)
E_OBJ       [k]     Object state is invalid
                    (Overrun handler has not been defined)
```

Function:

Each service call releases the processing time limit for the task indicated by parameter tskid and stops overrun handler operation.

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

In service call stp_ovr, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call istp_ovr, a task belonging to the kernel of another CPU cannot be specified as tskid.

6.25.4 Reference Overrun Handler State (ref_ovr, iref_ovr)

C-Language API:

```
ER ercd = ref_ovr(ID tskid, T_ROVR *pk_rovr);
ER ercd = iref_ovr(ID tskid, T_ROVR *pk_rovr);
```

Parameters:

```
tskid           Task ID
pk_rovr         Pointer to the packet where the overrun handler state is to
                be returned
```

Return Values:

```
Normal end (E_OK) or error code
```

Packet Structure:

```
typedef    struct    {
           STAT      ovrstat;     Overrun handler operating state
           OVRTIM    leftotm;     Remaining processor time
}T_ROVR;
```

Error Codes:

```
E_ID       [p]   Invalid ID number
                 (1) CPU ID is invalid (GET_CPUID (tskid) is invalid)
                 (2) Out of local ID range
                     (GET_LOCALID (tskid) < 0 or
                     (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid))
                 (3) tskid = TSK_SELF (0) when called in a non-task context
E_CTX      [k]   Context error (Called in prohibited system state when
                 GET_CPUID (tskid) is not the current CPU)
E_NOEXS    [k]   Non-existent object (Task specified by tskid does not
                 exist)
E_OBJ      [k]   Object state is invalid
                 (Overrun handler has not been defined)
```

Function:

The state of the overrun handler for the task specified by tskid is referenced, and the state of operation of the overrun handler (ovrstat) and the remaining processor time (leftotm) are returned to the area specified by pk_rovr.

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

As the operating state of the overrun handler, the processing time limit setting is returned as ovrstat.

- TOVR_STP (0x00000000): No processing time limit is set
- TOVR_STA (0x00000001): A processing time limit is set

The processor time remaining until the overrun handler is started due to the target task is returned as leftotm. If no processing time limit is set for the task, the value of leftotm is undefined.

In service call ref_ovr, a task belonging to the kernel of another CPU can be specified as tskid, except for in dispatch-pending state. In service call iref_ovr, a task belonging to the kernel of another CPU cannot be specified as tskid.

6.26 System State Management

The system state is controlled by the service calls listed in table 6.37.

Table 6.37 Service Calls for System State Management

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

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
[B]: Basic profile service calls
[R]: Service calls that can be issued remotely

2. T: Can be called in a task context
N: Can be called in a non-task context
E: Can be called in dispatch-enabled state
D: Can be called in dispatch-disabled state
U: Can be called in CPU-unlocked state
L: Can be called in CPU-locked state
C: Can be called while executing the normal CPU exception handler
O: Can be called in the state
∆: Can be called in the state only when a local object is the target

6.26.1 Rotate Ready Queue (rot_rdq, irot_rdq)

C-Language API:

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

Parameters:

```
tskpri          Task priority
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_PAR       [p]   Parameter error (tskpri < 0, tskpri > TMAX_TPRI of current
                  CPU, or tskpri = TPRI_SELF (0) is specified in a non-task
                  context)
```

Function:

Each service call rotates the current CPU's ready queue of the task priority indicated by parameter tskpri. In other words, the task at the head of the ready queue for the task priority 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, the ready queue for the priority where the current task is included, cannot be rotated even when TPRI_SELF is specified.

6.26.2 Get Current Task ID (get_tid, iget_tid)

C-Language API:

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

Parameters:

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

Return Values:

```
Normal end (E_OK)
```

Function:

Each service call refers to the ID of the task in the RUNNING state for the current CPU and returns it to the area indicated by p_tskid.

If each service call is issued in a task context, the current task ID is returned. If each service call is issued in a non-task context, the task ID that is being executed at that point is returned. The CPU ID (1 or 2) for the current CPU is set in bits 14 to 12. If there is no task in the RUNNING state, TSK_NONE (0) is returned.

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

6.26.3 Lock CPU (loc_cpu, iloc_cpu)

C-Language API:

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

Return Values:

```
Normal end (E_OK)
```

Function:

Each service call shifts the system state of the current CPU's kernel to the CPU-locked state, and disables interrupts and task dispatches.

The following describes the CPU-locked state:

- Tasks cannot be scheduled in the CPU-locked state.
- Interrupts, having a level equal to or below the kernel interrupt mask level (system.system_IPL) specified in the cfg file, are disabled.

- Only the following service calls can be issued in the CPU-locked state. Normal system operation cannot be guaranteed when a service call other than the following is issued:
  - ext_tsk
  - exd_tsk
  - sns_tex
  - loc_cpu, iloc_cpu
  - unl_cpu, iunl_cpu
  - sns_ctx
  - sns_loc
  - sns_dsp
  - sns_dpn
  - vsta_knl, ivsta_knl
  - vsys_dwn, ivsys_dwn
  - vsns_tmr

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

- unl_cpu or iunl_cpu
- ext_tsk or exd_tsk

The transition between CPU-locked state and CPU-unlocked state occurs only when service call loc_cpu, iloc_cpu, unl_cpu, iunl_cpu, ext_tsk, or exd_tsk is issued. An interrupt handler whose level is equal to or lower than the kernel interrupt mask level, time event handler, initialization routine, and task exception handling routine must unlock the CPU at termination. If the CPU is locked at termination, normal system operation cannot be guaranteed. Note that the CPU is always unlocked at the start of these handlers.

If the CPU exception handler switches the CPU-locked state and CPU-unlocked state, the system must be returned to the state at handler initiation before the handler terminates. If the former state is not recovered, normal system operation cannot be guaranteed.

If service calls loc_cpu and iloc_cpu are issued in CPU-unlocked state, no error will occur, but queuing will not be done.

6.26.4 Unlock CPU (unl_cpu, iunl_cpu)

C-Language API:

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

Return Values:

```
Normal end (E_OK)
```

Function:

Each service call cancels the CPU-locked state of the current CPU's kernel; CPU was locked by service call loc_cpu or iloc_cpu. If service call unl_cpu is issued in the dispatch-enabled state, task scheduling is performed.

When the system makes a transition to the CPU-locked state by issuing service call iloc_cpu in the interrupt handler, service call iunl_cpu must be issued 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 which was disabled by service call dis_dsp.

If service calls unl_cpu and iunl_cpu are issued in CPU-unlocked state, no error will occur, but queuing will not be done.

6.26.5 Disable Dispatch (dis_dsp)

C-Language API:

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

Return Values:

```
Normal end (E_OK)
```

Error Codes:

```
E_CTX       [k]   Context error (Called in prohibited system state)
```

Function:

Service call dis_dsp shifts the system state of the current CPU's kernel to the dispatch-disabled state.

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 issued.

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

- ena_dsp
- ext_tsk or exd_tsk

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

If the CPU exception handler switches the dispatch-disabled state and dispatch-enabled state, the system must be returned to the state at handler initiation before the handler terminates. If the former state is not recovered, normal system operation cannot be guaranteed.

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

If service call dis_dsp is issued while task dispatch is disabled, no error will occur, but queuing will not be done.

This service call must always be issued in a task context while the CPU is unlocked. If this service call is issued in any other state, normal system operation cannot be guaranteed.

6.26.6 Enable Dispatch (ena_dsp)

C-Language API:

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

Return Values:

```
Normal end (E_OK)
```

Error Codes:

```
E_CTX       [k]   Context error (Called in prohibited system state)
```

Function:

Service call ena_dsp cancels the dispatch-disabled state of the current CPU's kernel; dispatch was disabled by service call dis_dsp. Task scheduling is then performed after the system is able to execute tasks.

If service call ena_dsp is issued while task dispatch is enabled, no error will occur, but queuing will not be done.

This service call must always be issued in a task context while the CPU is unlocked. If this service call is issued in any other state, normal system operation cannot be guaranteed.

6.26.7 Check Context (sns_ctx)

C-Language API:

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

Return Values:

```
TRUE is returned when this service call is issued in a non-task context and
FALSE is returned when this service call is issued in a task context
```

Function:

Service call sns_ctx checks the current context type.

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

6.26.8 Check CPU-Locked State (sns_loc)

C-Language API:

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

Return Values:

```
TRUE is returned when the CPU is locked and FALSE is returned when the CPU is
unlocked
```

Function:

Service call sns_loc checks whether the CPU is locked.

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

6.26.9 Check Dispatch-Disabled State (sns_dsp)

C-Language API:

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

Return Values:

```
TRUE is returned when task dispatch is disabled and FALSE is returned when
task dispatch is enabled
```

Function:

Service call sns_dsp checks whether task dispatch is disabled.

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

6.26.10 Check Dispatch-Pending State (sns_dpn)

C-Language API:

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

Return Values:

```
TRUE is returned when task dispatch is pended and FALSE is returned when task
dispatch is not pended
```

Function:

Service call sns_dpn checks whether task dispatch is pended.

When any one of the following conditions is satisfied, task dispatch is pended.

- Task dispatch is disabled.
- The CPU is locked.
- A non-task context is being executed.
- The normal CPU exception handler is being executed.
- Interrupt mask level (value indicated by the IMASK bits in SR) is not 0

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

6.26.11 Start Kernel (vsta_knl, ivsta_knl)

C-Language API:

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

Function:

Each service call starts the kernel.

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

Each service call can also be issued in the CPU-locked state and from the normal CPU exception handler. Each service call can be issued even before the kernel is started.

Each service call should be issued in a state with all interrupts masked (SR.IMASK = 15).

An application issuing these service calls must be linked with the kernel.

Control will not return to the caller from these service calls.

The following outlines processing performed by these service calls.

1. Creates the interrupt vector table and initializes VBR.
2. Initializes the kernel internal table.
3. Creates objects specified in the cfg file.
4. Sets the IMASK bits in SR to the kernel interrupt mask level (system.system_IPL).
5. Calls the timer initialization function tdr_ini_tmr().
6. Calls the initialization routines specified in the cfg file.
7. Enters the multitasking environment.

When system.trace!=NO, the trace serial numbers may be inconsistent. To avoid this problem, call vsta_knl on CPUID#2 after starting the initialization routine or the first task on CPUID#1.

These service calls are functions not defined in the μITRON4.0 specification.

6.26.12 Initialize Remote Service-Call Environment (vini_rmt)

C-Language API:

```
ER ercd = vini_rmt(void);
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_CTX          [k]   Context error (Called in prohibited system state)
E_SYS          [k]   System error
EV_NORESOURCE  [k]   Insufficient resource
                     (1) Failed in IPI port creation
                     (2) Failed in SVC server task creation
                     (3) Failed in fixed-sized memory pool creation
```

Function:

Service call vini_rmt performs initialization processing for accepting remote service calls from another CPU and sending remote service calls to another CPU.

If remote_svc.num_server is not 0, the following initialization processing is performed to accept remote service calls from another CPU.

1. Creates an IPI port.
   The IPI port specified by remote_svc.ipi_portid is created using IPI_create().

2. Creates and initiates SVC server tasks.

   SVC server tasks are created using service call acre_tsk for the number specified by remote_svc.num_server. The main contents of the parameter packet (T_CTSK structure) passed to acre_tsk are as follows:

   — tskatr (task attribute): TA_HLNG | TA_ACT
   — task (task start address): Address of SVC server task function in the kernel
   — itskpri (initial priority): remote_svc.priority
   — stksz (stack size): remote_svc.stack_size
   — stk (stack address): Area automatically allocated at configuration (inside of section BC_hirmtstk)

If remote_svc.num_wait is not 0, the following initialization processing is performed to send remote service calls to another CPU.

3. Creates a fixed-sized memory pool.

   A single fixed-sized memory pool with memory blocks for the number specified by remote_svc.num_wait is created using service call acre_mpf. The main contents of the parameter packet (T_CMPFstructure) passed to acre_mpf are as follows:

   — mpfatr (fixed-sized memory pool attribute): TA_TFIFO
   — blkcnt (number of memory blocks): remote_svc.num_wait
   — blksz (memory block size): 20 bytes
   — mpf (start address for fixed-sized memory pool): Area automatically allocated at configuration (inside of section BD_hirmtmpf)
   — mpfmb (fixed-sized memory pool management table address) (only when system.mpfmanage is OUT): Area automatically allocated at configuration (inside of section BC_hiwrk)

Remote service calls can be issued to and from the current CPU after operation of service call vini_rmt has completed.

IPI_init() must be completed before service call vini_rmt is issued. Furthermore, if the current CPU ID is not 1, service call vini_rmt needs to be completed in the master CPUID#1 before service call vini_rmt is issued in the current CPU. If these requirements are not satisfied, operation becomes undefined.

Service call vini_rmt should be issued only once immediately after the kernel has been initiated.

Service call vini_rmt is implemented only in the API layer. Therefore, each processing of this service call is executed in the same context as the caller.

This service call is a function not defined in the μITRON4.0 specification.

6.26.13 System Down (vsys_dwn, ivsys_dwn)

C-Language API:

```
void vsys_dwn(W type, VW inf1, VW inf2, VW inf3);
void ivsys_dwn(W type, VW inf1, VW inf2, VW inf3);
```

Parameters:

type	Error type
inf1	System abnormal information 1
inf2	System abnormal information 2
inf3	System abnormal information 3

Function:

Each service call passes control to the system down routine.

For parameter type, a value (1 to 0x7FFFFFFF) corresponding to the generated error must be specified as the error type. Value 0 or smaller values are reserved for system use.

The system down routine is also executed when abnormal operation is detected in the kernel.

Service calls vsys_dwn and ivsys_dwn can be issued in the CPU-locked state and from the normal CPU exception handler.

Control will not return to the caller from these service calls.

These service calls are functions not defined in the μITRON4.0 specification.

6.26.14 Get Trace Information (vget_trc, ivget_trc)

C-Language API:

```
ER ercd = vget_trc(VW para1, VW para2, VW para3, VW para4);
ER ercd = ivget_trc(VW para1, VW para2, VW para3, VW para4);
```

Parameters:

para1	Parameter 1
para2	Parameter 2
para3	Parameter 3
para4	Parameter 4

Return Values:

Normal end (E_OK)

Function:

Each service call traces and acquires information required by the user.

Parameters para1 to para4 can be used freely by the user to distinguish the information to be acquired.

The acquired trace information can be displayed by using a debugging extension (DX).

If NO is specified for system.trace in the cfg file, these service calls do not perform any processing.

These service calls are functions not defined in the μITRON4.0 specification.

6.26.15 Get Start of Interrupt Handlers as Trace Information (ivbgn_int)

C-Language API:

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

Parameters:

```
dintno          Interrupt handler number
```

Return Values:

```
Normal end (E_OK)
```

Function:

Service call ivbgn_int traces the beginning of processing of the interrupt handler for the interrupt handler number specified by dintno.

The CPU interrupt vector number is specified for the interrupt handler number.

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

An error does not occur if this service call is issued from code other than an interrupt handler, but in such cases there is a possibility that the trace display by the debugging extension may be illegal.

If NO is specified for system.trace in the cfg file, this service call does not perform any processing.

This service call is a function not defined in the μITRON4.0 specification.

6.26.16 Get End of Interrupt Handlers as Trace Information (ivend_int)

C-Language API:

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

Parameters:

```
dintno          Interrupt handler number
```

Return Values:

```
Normal end (E_OK)
```

Function:

Service call ivend_int traces the end of processing of the interrupt handler for the interrupt handler number specified by dintno.

The CPU interrupt vector number is specified for the interrupt handler number.

This service call should be issued at the end of an interrupt handler. In addition, it should always be used in combination with service call ivbgn_int.

An error does not occur if this service call is issued from code other than an interrupt handler, but in such cases there is a possibility that the trace display by the debugging extension may be illegal.

If NO is specified for system.trace in the cfg file, this service call does not perform any processing.

This service call is a function not defined in the μITRON4.0 specification.

6.27 Interrupt Management

Interrupts are controlled by the service calls listed in table 6.38.

Table 6.38 Service Calls for Interrupt Management

Service Call*1	Description	System State*2						
		T	N	E	D	U	L	C
def_inh	Defines interrupt handler	O		O	O	O		
idef_inh			O	O	O	O		
chg_ims	Changes interrupt mask	O		O	O	O		
ichg_ims			O	O	O	O		
get_ims	Refers to interrupt mask	O		O	O	O		
iget_ims			O	O	O	O		

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
[B]: Basic profile service calls
[R]: Service calls that can be issued remotely
2. T: Can be called in a task context
N: Can be called in a non-task context
E: Can be called in dispatch-enabled state
D: Can be called in dispatch-disabled state
U: Can be called in CPU-unlocked state
L: Can be called in CPU-locked state
C: Can be called while executing the normal CPU exception handler
O: Can be called in the state
∆: Can be called in the state only when a local object is the target

The interrupt management specifications are listed in table 6.39.

Table 6.39 Interrupt Management Specifications

Item	Description
Interrupt handler number	4 to _MAX_INT (511 max.)
Interrupt handler attributes	TA_HLNG: Written in a high-level language TA_ASM: Written in assembly language

6.27.1 Define Interrupt Handler (def_inh, idef_inh)

C-Language API:

```
ER ercd = def_inh(INHNO inhno, T_DINH *pk_dinh);
ER ercd = idef_inh(INHNO inhno, T_DINH *pk_dinh);
```

Parameters:

inhno	Interrupt handler number
pk_dinh	Pointer to the packet where the interrupt handler definition information is stored

Return Values:

Normal end (E_OK) or error code

Packet Structure:

```
typedef    struct    {
           ATR       inhatr;      Handler attribute
           FP        inthdr;      Handler address
           UINT      inhsr;       (For future expansion)
}T_DINH;
```

Error Codes:

E_RSATR	[p]	Invalid attribute (inhatr is invalid)
E_PAR	[p]	Parameter error Invalid number was specified (inhno = 0 to 3 or 60 to 63, or inhno > _MAX_INT of current CPU)

Function:

Each service call defines the interrupt handler for the interrupt handler number specified by inhno in the kernel of the current CPU with the contents specified by pk_dinh.

These service calls can be used only when RAM or RAM_ONLY_DIRECT is specified for system.vector_type in the cfg file.

The CPU vector number is specified for the interrupt handler number.

These service calls cannot be used to define handlers for interrupt handler numbers 0 to 3 (power-on reset, manual reset). Interrupt handler numbers 60 to 63 are reserved numbers for system use and must not be specified.

Parameter inhatr specifies the following as the attributes.

inhatr := (TA_HLNG || TA_ASM) [|(VTA_DIRECT || VTA_REGBANK)]

- TA_HLNG (0x00000000): Written in a high-level language
- TA_ASM (0x00000001): Written in assembly language
- VTA_DIRECT (0x80000000): Direct attribute
- VTA_REGBANK (0x40000000): Normal interrupt handler making use of the register banks

If the VTA_DIRECT attribute is specified, the defined handler is initiated without any kernel intervention when an interrupt occurs. Such kind of handler is referred to as a "direct interrupt handler". If the VTA_DIRECT attribute is not specified, the handler is initiated through kernel intervention when an interrupt occurs. Such kind of handler is referred to as a "normal interrupt handler".

VTA_DIRECT must be specified when defining an interrupt handler with an interrupt level higher than the kernel interrupt mask level.

Note that the method for writing a handler differs depending on whether VTA_DIRECT is specified. For details, refer to section 12.5, Interrupt Handlers.

When RAM_ONLY_DIRECT is specified for system.vector_type in the cfg file, it is not possible to define handlers without the VTA_DIRECT attribute.

VTA_REGBANK is valid only when all of the following conditions are satisfied. Otherwise, the specification of VTA_REGBANK has no meaning.

(a) The VTA_DIRECT attribute is not specified.

(b) BANKLEVELxx is specified for system.regbank.

(c) A value other than 0 is specified for INTSPEC_IBNR_ADR1 (for CPUID#1) or INTSPEC_IBNR_ADR2 (for CPUID#2) for the relevant CPU in the "CPU interrupt specification definition file (kernel_intspec.h)". (A CPU that provides register banks is used.)

(d) A vector number other than the vector number corresponding to INTSPEC_NOBANK_VEC??? defined in the "CPU interrupt specification definition file (kernel_intspec.h)" is specified. (A vector number for an interrupt source with which the register banks can be used is specified due to the CPU specifications.)

When all of these conditions hold, VTA_REGBANK must be specified as appropriate according to the interrupt level of the interrupt handler to be defined as shown below. If these are not followed, the interrupt handler will not operate correctly.

(i) When BANKLEVELxx corresponding to the interrupt level of the interrupt handler to be used is specified for system.regbank

VTA_REGBANK must be specified.

(ii) When BANKLEVELxx corresponding to the interrupt level of the interrupt handler to be used is not specified for system.regbank

VTA_REGBANK must not be specified.

inhsr is reserved for future expansion and is simply ignored.

When pk_dinh = NULL (0) is specified, the definition of inhno is canceled.

inhsr is a member not defined in the μITRON4.0 specification.

6.27.2 Change Interrupt Mask Level (chg_ims, ichg_ims)

C-Language API:

```
ER ercd = chg_ims(IMASK imask);
ER ercd = ichg_ims(IMASK imask);
```

Parameters:

```
imask       Interrupt mask value
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_PAR        [p]   Parameter error (A value other than SR_IMS00 to SR_IMS15
                   was specified for imask)
```

Function:

Each service call changes the CPU's interrupt mask (IMASK bits in SR) to the level specified by imask.

imask can be specified as follows:

- SR_IMSnn (0x0000000m) Changes interrupt mask level to nn.

  nn: Character string indicating a two-digit decimal number from 0 to 15 (00, 01, 02, ... , 15)

  m: nn converted to a hexadecimal number

For details on controlling the interrupt mask, refer to section 4.8.2, Controlling Interrupts (by Setting IMASK Bits in the Register SR).

6.27.3 Reference Interrupt Mask Level (get_ims, iget_ims)

C-Language API:

```
ER ercd = get_ims(IMASK *p_imask);
ER ercd = iget_ims(IMASK *p_imask);
```

Parameters:

```
p_imask         Pointer to the memory area where the interrupt mask
                level is to be returned
```

Return Values:

```
Normal end (E_OK)
```

Function:

Each service call refers to the interrupt mask bits (IMASK bits) of the current CPU status register (SR) and returns the interrupt mask level to the area indicated by p_imask.

The value to be returned to p_imask has the same format as parameter imask used by service call chg_ims.

6.28 Service Call Management

Service calls are controlled by the service calls listed in table 6.40.

Table 6.40 Service Calls for Service Call Management

Service Call*1	Description	System State*2						
		T	N	E	D	U	L	C
def_svc	Defines extended service call routine	O		O	O	O		
idef_svc			O	O	O	O		
cal_svc	Calls service call routine	O		O	O	O		
ical_svc			O	O	O	O		

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

[B]: Basic profile service calls

[R]: Service calls that can be issued remotely

2. T: Can be called in a task context
N: Can be called in a non-task context
E: Can be called in dispatch-enabled state
D: Can be called in dispatch-disabled state
U: Can be called in CPU-unlocked state
L: Can be called in CPU-locked state
C: Can be called while executing the normal CPU exception handler
O: Can be called in the state
∆: Can be called in the state only when a local object is the target

The service call management specifications are listed in table 6.41.

Table 6.41 Service Call Management Specifications

Item	Description
Function code of extended service call	1 to _MAX_FNCD (1023 max.)
Parameter that can be passed	0 to 4 VP_INT type parameters
Extended service call routine attributes	TA_HLNG: Written in a high-level language TA_ASM: Written in assembly language

6.28.1 Define Extended Service-Call Routine (def_svc, idef_svc)

C-Language API:

```
ER ercd = def_svc(FN fncd, T_DSVC *pk_dsvc);
ER ercd = idef_svc(FN fncd, T_DSVC *pk_dsvc);
```

Parameters:

```
fncd            Function code of extended service call
pk_dsvc         Pointer to the packet where the extended service call routine
                definition information is stored
```

Return Values:

```
Normal end (E_OK) or error code
```

Packet Structure:

```
typedef  struct {
         ATR    svcatr;       Extended service call routine attribute
         FP     svcrtn;       Extended service call routine address
}T_DSVC;
```

Error Codes:

```
E_RSATR       [p]   Invalid attribute (svcatr is invalid)
E_PAR         [p]   Parameter error (fncd ≤ 0 or
                    fncd > _MAX_FNCD of current CPU)
```

Function:

For the kernel of the current CPU, each service call defines an extended service call routine for the extended function code indicated by fncd with the contents specified by pk_dsvc.

Parameter svcatr specifies the language in which the routine was written as the attribute.

svcatr := ((TA_HLNG || TA_ASM))

- TA_HLNG (0x00000000): Written in a high-level language
- TA_ASM (0x00000001): Written in assembly language

Parameter svcrtn specifies the start address of the extended service call routine.

If pk_dsvc = NULL (0) is specified for svcatr in these service calls, the extended service call routine defined for fncd is canceled.

The state of the calling task is taken over in extended service call routines.

6.28.2 Call Extended Service-Call Routine (cal_svc, ical_svc)

C-Language API:

```
ER_UINT ercd = cal_svc(FN fncd, …);
ER_UINT ercd = ical_svc(FN fncd, …);
```

Parameters:

```
fncd           Function code of extended service call
In "…" above, up to four VP_INT-type parameters can be substituted. If more
than four parameters are specified, only the first four parameters are passed
to the extended service call routine.
par1           Parameter 1
par2           Parameter 2
par3           Parameter 3
par4           Parameter 4
```

Return Values:

```
Return value from service call
```

Error Codes:

```
E_RSFN     [p]   Reserved function code (fncd is invalid or cannot be used)
```

Function:

Each service call executes the extended service call routine corresponding to the function code specified by parameter fncd.

Up to four VP_INT-type parameters can be specified. In the extended service call routine to be called, par1 to par4 are stored in R4 to R7, respectively, and passed.

For details, refer to section 12.4, Extended Service Call Routines.

6.29 System Configuration Management

System configuration is controlled by the service calls listed in table 6.42.

Table 6.42 Service Calls for System Configuration Management

Service Call*1	Description	System State*2						
		T	N	E	D	U	L	C
def_exc	Defines CPU exception handler	O		O	O	O		
idef_exc			O	O	O	O		
vdef_trp	Defines CPU exception handler (TRAPA instruction exception)	O		O	O	O		
ivdef_trp			O	O	O	O		
ref_cfg	Refers to configuration information	O		O	O	O		
iref_cfg			O	O	O	O		
ref_ver	Refers to version information	O		O	O	O		
iref_ver			O	O	O	O		

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
[B]: Basic profile service calls
[R]: Service calls that can be issued remotely
2. T: Can be called in a task context
N: Can be called in a non-task context
E: Can be called in dispatch-enabled state
D: Can be called in dispatch-disabled state
U: Can be called in CPU-unlocked state
L: Can be called in CPU-locked state
C: Can be called while executing the normal CPU exception handler
O: Can be called in the state
∆: Can be called in the state only when a local object is the target

The system configuration management specifications are listed in table 6.43.

Table 6.43 System Configuration Management Specifications

Item	Description
CPU exception handler number	4 to _MAX_INT (511 max.)
CPU exception handler attributes	TA_HLNG: Written in a high-level language TA_ASM: Written in assembly language

6.29.1 Define CPU Exception Handler (def_exc, idef_exc)

C-Language API:

```
ER ercd = def_exc(EXCNO excno, T_DEXC *pk_dexc);
ER ercd = idef_exc(EXCNO excno, T_DEXC *pk_dexc);
```

Parameters:

```
excno           CPU exception handler number
pk_dexc         Pointer to the packet where the CPU exception handler
                definition information is stored
```

Return Values:

```
Normal end (E_OK) or error code
```

Packet Structure:

```
typedef    struct  {
           ATR     excatr;      Handler attribute
           FP      exchdr;      Handler address
           UINT    excsr;       (For future expansion)
}T_DEXC;
```

Error Codes:

```
E_RSATR   [p]   Invalid attribute (excatr is invalid)
E_PAR     [p]   Parameter error
                Invalid number was specified (inhno = 0 to 3 or 60 to 63, or
                inhno > _MAX_INT of current CPU)
```

Function:

For the kernel of the current CPU, each service call defines a CPU exception handler for the CPU exception handler number specified by excno with the contents specified by pk_dexc.

These service calls can be used only when RAM or RAM_ONLY_DIRECT is specified for system.vector_type in the cfg file.

The CPU vector number is specified for the CPU exception handler number.

These service calls cannot be used to define handlers for CPU exception handler numbers 0 to 3 (power-on reset, manual reset). CPU exception handler numbers 60 to 63 are reserved numbers for system use and must not be specified.

Parameter excatr specifies the following as the attributes.

excatr := (TA_HLNG || TA_ASM) [|VTA_DIRECT]

- TA_HLNG (0x00000000): Written in a high-level language
- TA_ASM (0x00000001): Written in assembly language
- VTA_DIRECT (0x80000000): Direct attribute

If the VTA_DIRECT attribute is specified, the defined handler is initiated without any kernel intervention when a CPU exception occurs. Such kind of handler is referred to as a "direct CPU exception handler". If the VTA_DIRECT attribute is not specified, the handler is initiated through kernel intervention when a CPU exception occurs. Such kind of handler is referred to as a "normal CPU exception handler".

Note that the method for writing a handler differs depending on whether VTA_DIRECT is specified. For details, refer to section 12.6, CPU Exception Handlers (Including TRAPA Exceptions).

When RAM_ONLY_DIRECT is specified for system.vector_type in the cfg file, it is not possible to define handlers without the VTA_DIRECT attribute.

excsr is reserved for future expansion and is simply ignored. The SR value when a CPU exception handler is actually initiated is determined by the CPU exception handling.

When pk_dexc = NULL (0) is specified, the definition of excno is canceled.

The normal CPU exception handler is executed in a context referred to as the "CPU exception handler execution state", which is different from the context in which the CPU exception cause occurred. Service calls which can be issued from this state are limited to the following service calls. If service calls other than the following are issued, operation is not guaranteed.

- sns_tex
- sns_ctx
- sns_loc
- sns_dsp
- sns_dpn
- get_tid, iget_tid
- ras_tex, iras_tex
- vsta_knl, ivsta_knl
- vsys_dwn, ivsys_dwn
- vsns_tmr

The direct CPU exception handler is executed in the same context as that before the CPU exception cause occurred. Accordingly, the service calls that can be issued are the same as those before the CPU exception occurred and will not be determined statically.

In order to define a CPU exception handler for a TRAPA instruction, service call vdef_trp or ivdef_trp should be used instead of service calls def_exc and idef_exc.

excsr is a member not defined in the μITRON4.0 specification.

6.29.2 Define CPU Exception (TRAPA-Instruction Exception) Handler (vdef_trp, ivdef_trp)

C-Language API:

```
ER ercd = vdef_trp(UINT dtrpno, T_DTRP *pk_dtrp);
ER ercd = ivdef_trp(UINT dtrpno, T_DTRP *pk_dtrp);
```

Parameters:

dtrpno	Trap number
pk_dtrp	Pointer to the packet where the CPU exception (TRAPA instruction exception) handler definition information is stored

Return Values:

Normal end (E_OK) or error code

Packet Structure:

```
typedef   struct   {
          ATR      trpatr;    CPU exception (TRAPA instruction exception)
                              handler attribute
          FP       trphdr;    CPU exception (TRAPA instruction exception)
                              handler address
          UINT     trpsr;     (For future expansion)
}T_DTRP;
```

Error Codes:

E_RSATR	[p]	Invalid attribute (trpatr is invalid)
E_PAR	[p]	Parameter error Invalid number was specified (inhno = 0 to 3 or 60 to 63, or inhno > _MAX_INT of current CPU)

Function:

For the kernel of the current CPU, each service call defines a CPU exception (TRAPA instruction exception) handler for the trap number specified by dtrpno with the contents specified by pk_dtrp.

These service calls can be used only when RAM or RAM_ONLY_DIRECT is specified for system.vector_type in the cfg file.

The CPU vector number is specified for the trap number.

These service calls cannot be used to define handlers for trap numbers 0 to 3 (power-on reset, manual reset). Trap numbers 60 to 63 are reserved numbers for system use and must not be specified.

Parameter trpatr specifies the following as the attributes.

trpatr := (TA_HLNG || TA_ASM) [|VTA_DIRECT]

- TA_HLNG (0x00000000): Written in a high-level language
- TA_ASM (0x00000001): Written in assembly language
- VTA_DIRECT (0x80000000): Direct attribute

If the VTA_DIRECT attribute is specified, the defined handler is initiated without any kernel intervention when a TRAPA instruction exception occurs. Such kind of handler is referred to as a "direct CPU exception handler". If the VTA_DIRECT attribute is not specified, the handler is initiated through kernel intervention when a TRAPA instruction exception occurs. Such kind of handler is referred to as a "normal CPU exception handler".

Note that the method for writing a handler differs depending on whether VTA_DIRECT is specified. For details, refer to section 12.6, CPU Exception Handlers (Including TRAPA Exceptions).

When RAM_ONLY_DIRECT is specified for system.vector_type in the cfg file, it is not possible to define handlers without the VTA_DIRECT attribute.

trpsr is reserved for future expansion and is simply ignored. The SR value when a CPU exception handler is actually initiated is determined by the CPU exception handling.

When pk_dtrp = NULL (0) is specified, the definition of dtrpno is canceled.

The normal CPU exception handler is executed in a context referred to as the "CPU exception handler execution state", which is different from the context in which the CPU exception cause occurred. Service calls which can be issued from this state are limited to the following service calls. If service calls other than the following are issued, operation is not guaranteed.

- sns_tex
- sns_ctx
- sns_loc
- sns_dsp
- sns_dpn
- get_tid, iget_tid
- ras_tex, iras_tex
- vsta_knl, ivsta_knl
- vsys_dwn, ivsys_dwn
- vsns_tmr

The direct CPU exception handler is executed in the same context as that before the CPU exception cause occurred. Accordingly, the service calls that can be issued are the same as those before the CPU exception occurred and will not be determined statically.

These service calls are functions not defined in the μITRON4.0 specification.

6.29.3 Reference Configuration Information (ref_cfg, iref_cfg)

C-Language API:

```
ER ercd = ref_cfg(T_RCFG *pk_rcfg);
ER ercd = iref_cfg(T_RCFG *pk_rcfg);
```

Parameters:

```
pk_rcfg      Pointer to the packet where the configuration information is to
             be returned
```

Return Values:

```
Normal end (E_OK)
```

Packet Structure:

```
typedef   struct  {
          ID      maxtskid;    Maximum local task ID
          ID      ststkid;     Maximum local task ID using static stack
          ID      maxsemid;    Maximum local semaphore ID
          ID      maxflgid;    Maximum local event flag ID
          ID      maxdtqid;    Maximum local data queue ID
          ID      maxmbxid;    Maximum local mailbox ID
          ID      maxmtxid;    Maximum local mutex ID
          ID      maxmbfid;    Maximum local message buffer ID
          ID      maxmplid;    Maximum local variable-sized memory pool ID
          ID      maxmpfid;    Maximum local fixed-sized memory pool ID
          ID      maxcycid;    Maximum local cyclic handler ID
          ID      maxalmid;    Maximum local alarm handler ID
          ID      maxs_fncd;   Maximum extended service call function code
}T_RCFG;
```

Function:

Each service call returns the system configuration information to the area indicated by pk_rcfg.

The following parameters are returned to the packet specified by pk_rcfg.

- maxtskid: Maximum local task ID (_MAX_TSK)
- ststkid: Maximum local task ID using static stack (_MAX_STTSK)
- maxsemid: Maximum local semaphore ID (_MAX_SEM)
- maxflgid: Maximum local event flag ID (_MAX_FLAG)
- maxdtqid: Maximum local data queue ID (_MAX_DTQ)
- maxmbxid: Maximum local mailbox ID (_MAX_MBX)
- maxmtxid: Maximum local mutex ID (_MAX_DTQ)

- maxmbfid: Maximum local message buffer ID (_MAX_MBF)
- maxmplid: Maximum local variable-sized memory pool ID (_MAX_MPL)
- maxmpfid: Maximum local fixed-sized memory pool ID (_MAX_MPF)
- maxcycid: Maximum local cyclic handler ID (_MAX_CYH)
- maxalmid: Maximum local alarm handler ID (_MAX_ALH)
- maxs_fncd: Maximum extended service call function code (_MAX_FNCD)

The members of the T_RCFG structure are all not defined in the μITRON4.0 specification; the μITRON4.0 specification does not define anything about the contents of the T_RCFG structure.

6.29.4 Reference 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:

```
pk_rver        Pointer to the packet where the version information is to be
               returned
```

Return Values:

```
Normal end (E_OK)
```

Packet Structure:

```
typedef    struct    {
           UH        maker;      Manufacturer
           UH        prid;       Identification number
           UH        spver;      Specification version
           UH        prver;      Product version
           UH        prno[4];    Product management information
}T_RVER;
```

Function:

Each service call reads information on the version of the kernel currently in use and returns it to the area indicated by pk_rver.

The following information is returned to the packet indicated by pk_rver.

(1) maker

Parameter maker indicates the manufacturer of this kernel. The value for this kernel is 0x0115, which means Renesas.

(2) prid

Parameter prid indicates the number to identify the kernel or VLSI type. The value for this kernel is 0x0013.

(3) spver

Parameter spver indicates the specifications to which the kernel conforms, as follows:

- Bits 15 to 12: MAGIC (Number to identify the TRON specification series)
  0x5 (μITRON specifications) for this kernel
- Bits 11 to 0: SpecVer (Version number of the TRON specification on which the product is based)
  0x403 (μITRON4.0 specifications Ver.4.03.00) for this kernel

(4) prver

Parameter prver indicates the version number of the kernel.

The value of prver is different for each product version. Refer to the release notes attached to the product. For example, the value of prver is 0x0100 for V.1.00 Release 00.

(5) prno

Parameter prno indicates the product management information and the product number.

The prno[0] to prno[3] values of this kernel are 0x0000.

6.30 Profile Management

The profile function is controlled by the service calls listed in table 6.44.

Table 6.44 Service Calls for Profile Management

Service Call*[1]	Description	System State*[2]						
		T	N	E	D	U	L	C
vref_prf [R]	Refers to profile counter state	O		O	Δ	O		
ivref_prf			O	O	O	O		
vclr_prf [R]	Clears profile counter	O		O	Δ	O		
ivclr_prf			O	O	O	O		

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
[B]: Basic profile service calls
[R]: Service calls that can be issued remotely
2. T: Can be called in a task context
N: Can be called in a non-task context
E: Can be called in dispatch-enabled state
D: Can be called in dispatch-disabled state
U: Can be called in CPU-unlocked state
L: Can be called in CPU-locked state
C: Can be called while executing the normal CPU exception handler
O: Can be called in the state
Δ: Can be called in the state only when a local object is the target

Respective 32-bit profile counters are provided for overall time, each task, and kernel idling.

The timer-interrupt processing for the kernel executed in cycles of TIC_NUME/TIC_DENO milliseconds governs updating of these counters. That is, incrementation of the individual profile counters for the running task or kernel idling, and that for the overall time, proceeds with this cycle.

Although the results are not exact, a long period of measurement gives approximate execution times for tasks as calculated by the formula below.

Execution time [ms] =
Value of an individual profile counter for a task × (TIC_NUME/TIC_DENO)

In addition, the CPU usage of a task or of kernel idling can be estimated by dividing the value of the corresponding profile counter by the value of the profile counter for overall time.

Issuing vsta_knl initializes all profile counters to 0. Individual profile counters for tasks are also initialized when the corresponding tasks are deleted.

Also note that the kernel does not detect any overflow of the profile counters. For example, the counters will start to overflow in 50 days when TIC_NUME/TIC_DENO = 1 ms, and in 12 hours or so when TIC_NUME/TIC_DENO = 10 µs.

Profile information is acquired even when these service calls are not selected at configuration, and the acquired profile information can be displayed by using a debugging extension (DX).

6.30.1 Reference Profile Counter (vref_prf, ivref_prf)

C-Language API:

```
ER ercd = vref_prf(ID tskid, UW *p_count, UW *p_allcount);
ER ercd = ivref_prf(ID tskid, UW *p_count, UW *p_allcount);
```

Parameters:

```
tskid          Task ID
p_count        Pointer to the memory area where the profile counter for tskid
               is to be returned
p_allcount     Pointer to the memory area where the value of the profile
               counter for the overall time is to be returned
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_ID        [p]    Invalid ID number
                   (1) CPU ID is invalid (GET_CPUID (tskid) is invalid)
                   (2) Out of local ID range
                       (GET_LOCALID (tskid) < VKNL_IDLE,
                       VKNL_IDLE < GET_LOCALID (tskid) < 0,
                       (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid))
                   (3) tskid = TSK_SELF (0) when called in a non-task context
E_CTX       [k]    Context error (Called in prohibited system state when
                   GET_CPUID (tskid) is not the current CPU)
E_NOEXS     [k]    Non-existent object (Task specified by tskid does not exist)
                   (Only when GET_LOCALID (tskid) > 0)
```

Function:

Each service call acquires profile information for the task specified by tskid and returns the acquired profile information to the area indicated by p_count.

When VKNL_IDLE is specified for the local ID of tskid, kernel idling is specified for the specified CPU ID.

By specifying tskid = TSK_SELF (0) when these service calls are issued from a task context, the current task of the current CPU is specified.

The value of the profile counter for the overall time is returned to the area indicated by p_allcount.

There is no means to learn whether the contents of p_count or p_allcount have overflowed.

In service call vref_prf, another CPU ID can be specified as the CPU ID of tskid, except for in dispatch-pending state. In service call ivref_prf, another CPU ID cannot be specified as the CPU ID of tskid.

6.30.2 Clear Profile Counter (vclr_prf, ivclr_prf)

C-Language API:

```
ER ercd = vclr_prf(ID tskid);
ER ercd = ivclr_prf(ID tskid);
```

Parameters:

```
tskid         Task ID
```

Return Values:

```
Normal end (E_OK) or error code
```

Error Codes:

```
E_ID        [p]   Invalid ID number
                  (1) CPU ID is invalid (GET_CPUID (tskid) is invalid)
                  (2) Out of local ID range
                      (GET_LOCALID (tskid) < VCTX_ALL,
                      VKNL_IDLE < GET_LOCALID (tskid) < 0,
                      (_MAX_TSK of GET_CPUID (tskid)) < GET_LOCALID (tskid))
                  (3) tskid = TSK_SELF (0) when called in a non-task context
E_CTX       [k]   Context error (Called in prohibited system state when
                  GET_CPUID (tskid) is not the current CPU)
E_NOEXS     [k]   Non-existent object (Task specified by tskid does not exist)
                  (Only when GET_LOCALID (tskid) > 0)
```

Function:

Each service call clears the profile counter for the task specified by tskid.

When VKNL_IDLE is specified for the local ID of tskid, the kernel idling profile counter for the specified CPU ID is cleared to 0.

When VTSK_ALL is specified for the local ID of tskid, the profile counters for all tasks of the specified CPU ID are cleared to 0.

When VCTX_ALL is specified for the local ID of tskid, all profile counters (for overall time, each task, and kernel idling) for the specified CPU ID are cleared to 0.

By specifying tskid = TSK_SELF (0) when these service calls are issued from a task context, the current task of the current CPU is specified.

In service call vclr_prf, another CPU ID can be specified as the CPU ID of tskid, except for in dispatch-pending state. In service call ivclr_prf, another CPU ID cannot be specified as the CPU ID of tskid.

6.31 Macros

6.31.1 Constant Macros

(1) Error Codes

Error codes are defined in itron.h as shown below.

```
#define E_OK            0L              /* normal end                       */

/*---- internal error class ----*/
#define E_SYS           (-5L)           /* system error                     */

/*---- no support error class ----*/
#define E_NOSPT         (-9L)           /* no support function              */
#define E_RSFN          (-10L)          /* reserved function code number    */
#define E_RSATR         (-11L)          /* reserved attribute code number   */

/*---- parameter error class ----*/
#define E_PAR           (-17L)          /* parameter error                  */
#define E_ID            (-18L)          /* reserved id number               */

/*---- context error class ----*/
#define E_CTX           (-25L)          /* context error                    */
#define E_MACV          (-26L)          /* memory access violation          */
#define E_OACV          (-27L)          /* object access violation          */
#define E_ILUSE         (-28L)          /* service call illegal use         */

/*---- resource insufficiency error class ----*/
#define E_NOMEM         (-33L)          /* no memory                        */
#define E_NOID          (-34L)          /* no ID                            */

/*---- object status error class ----*/
#define E_OBJ           (-41L)          /* object status error              */
#define E_NOEXS         (-42L)          /* object non existent              */
#define E_QOVR          (-43L)          /* queuing over flow                */

/*---- wait release error class ----*/
#define E_RLWAI         (-49L)          /* wait status forced release       */
#define E_TMOUT         (-50L)          /* time out                         */
#define E_DLT           (-51L)          /* delete object                    */

/*---- other errors ---*/
#define EV_NOINIT       (-97L)          /* not initialized                  */
#define EV_NORESOURCE   (-98L)          /* no resource                      */
#define EV_OBJ          (-99L)          /* kernel busy                      */
```

(2) General

Definition Name	Where Definition Is	Defined Value	Description
NULL	itron.h	Same as the definition in the C standard include file stddef.h. Defined only when NULL has not been defined.	
TRUE		1	BOOL-type data "TRUE"
FALSE		0	BOOL-type data "FALSE"

(3) Object Attributes

Definition Name	Where Definition Is	Defined Value	Description
TA_NULL	itron.h	0UL	No object attribute specification
TA_HLNG	kernel.h	0x00000000UL	Program is started in high-level language interface*
TA_ASM		0x00000001UL	Program is started in assembly language interface*
TA_COP1		0x00000200UL	FPU is used
TA_TFIFO		0x00000000UL	Wait task queue is managed on a FIFO basis
TA_TPRI		0x00000001UL	Wait task queue is managed on the task priority
TA_MFIFO		0x00000000UL	Message queue is managed on a FIFO basis
TA_MPRI		0x00000002UL	Message queue is managed on the message priority
TA_ACT		0x00000002UL	Task makes a transition to the READY state after it has been created
TA_WSGL		0x00000000UL	Does not permit multiple tasks to wait for the event flag
TA_WMUL		0x00000002UL	Permits multiple tasks to wait for the event flag
TA_CLR		0x00000004UL	Clears the event flag at the time of waiting release
TA_CEILING		0x00000003UL	Priority ceiling protocol for mutexes
TA_STA		0x00000002UL	Cyclic handler makes a transition to the operating state after it has been created
TA_PHS		0x00000004UL	Preserves the initiation phase of the cyclic handler
VTA_REGBANK		0x40000000UL	Normal interrupt handler making use of register banks
VTA_DIRECT		0x80000000UL	Direct interrupt handler or direct CPU exception handler
VTA_ UNFRAGMENT		0x80000000UL	Sector management is specified for variable-sized memory pools

Note: * In this kernel, the same operation is performed no matter whether TA_HLNG or TA_ASM has been specified.

(4) Timeout Specification

Definition Name	Where Definition Is	Defined Value	Description
TMO_POL	itron.h	0L	Polling
TMO_FEVR		−1L	Waiting forever

(5) Service Call Operating Modes

Definition Name	Where Definition Is	Defined Value	Description
TWF_ANDW	kernel.h	0x00000000UL	AND wait for event flags
TWF_ORW		0x00000001UL	OR wait for event flags

(6) Task State

Definition Name	Where Definition Is	Defined Value	Description
TTS_RUN	kernel.h	0x00000001UL	RUNNING state
TTS_RDY		0x00000002UL	READY state
TTS_WAI		0x00000004UL	WAITING state
TTS_SUS		0x00000008UL	SUSPENDED state
TTS_WAS		0x0000000cUL	WAITING-SUSPENDED state
TTS_DMT		0x00000010UL	DORMANT state
TTS_STK*		0x40000000UL	Shared-stack WAITING state

Note: * Not defined in the μITRON4.0 specification.

(7) Cause of WAITING State of Task

Definition Name	Where Definition Is	Defined Value	Description
TTW_SLP	kernel.h	0x00000001UL	Waiting for wakeup (slp_tsk, tslp_tsk)
TTW_DLY		0x00000002UL	Waiting for the specified time to elapse (dly_tsk)
TTW_SEM		0x00000004UL	Waiting to acquire a semaphore resource (wai_sem, twai_sem)
TTW_FLG		0x00000008UL	Waiting for the event flag to be set (wai_flg, twai_flg)
TTW_SDTQ		0x00000010UL	Waiting to send data to the data queue (snd_dtq, tsnd_dtq)
TTW_RDTQ		0x00000020UL	Waiting to receive data from the data queue (rcv_dtq, trcv_dtq)
TTW_MBX		0x00000040UL	Waiting to receive a message from the mailbox (rcv_mbx, trcv_mbx)
TTW_MTX		0x00000080UL	Waiting to lock the mutex (loc_mtx, tloc_mtx)
TTW_SMBF		0x00000100UL	Waiting to send a message to the message buffer (snd_mbf, tsnd_mbf)
TTW_RMBF		0x00000200UL	Waiting to receive a message from the message buffer (rcv_mbf, trcv_mbf)
TTW_MPF		0x00002000UL	Waiting to get a fixed-sized memory block (get_mpf, tget_mpf)
TTW_MPL		0x00004000UL	Waiting to get a variable-sized memory block (get_mpl, tget_mpl)
TTW_TFL*		0x00008000UL	Waiting for any bit of the task event flag to be set (vwai_tfl, vtwai_tfl)

Note: * Not defined in the μITRON4.0 specification.

(8) Operating State

Definition Name	Where Definition Is	Defined Value	Description
TTEX_ENA	kernel.h	0x00000000UL	Task exception handling enabled state
TTEX_DIS		0x00000001UL	Task exception handling disabled state
TCYC_STP		0x00000000UL	Cyclic handler is not in the operating state
TCYC_STA		0x00000001UL	Cyclic handler is in the operating state
TALM_STP		0x00000000UL	Alarm handler is not in the operating state
TALM_STA		0x00000001UL	Alarm handler is in the operating state
TOVR_STP		0x00000000UL	No processing time limit is set
TOVR_STA		0x00000001UL	An processing time limit is set

(9) Others

Definition Name	Where Definition Is	Defined Value	Description
TSK_SELF	kernel.h	0	Current task is specified
TSK_NONE		0	The required task does not exist
TPRI_SELF		0	Base priority of the current task is specified
TPRI_INI		0	Initial task priority is specified
VCPU_SELF*		0	Current CPU is specified
VCPU_MAX*		2	Maximum CPU ID
VKNL_IDLE*		–32768	Kernel idling is specified
VTSK_ALL*		–32767	All tasks are specified
VCTX_ALL*		–32766	All contexts are specified
ECM_SUS*		0x00000001UL	Forcible suspension request
ECM_TER*		0x00000002UL	Forcible termination request
PND_SUS*		0x00000004UL	Forcible suspension request is pended
PND_TER*		0x00000008UL	Forcible termination request is pended
SR_IMS00*		0x00000000UL	Interrupt mask level = 0
SR_IMS01*		0x00000001UL	Interrupt mask level = 1
SR_IMS02*		0x00000002UL	Interrupt mask level = 2
SR_IMS03*		0x00000003UL	Interrupt mask level = 3
SR_IMS04*		0x00000004UL	Interrupt mask level = 4
SR_IMS05*		0x00000005UL	Interrupt mask level = 5
SR_IMS06*		0x00000006UL	Interrupt mask level = 6
SR_IMS07*		0x00000007UL	Interrupt mask level = 7
SR_IMS08*		0x00000008UL	Interrupt mask level = 8
SR_IMS09*		0x00000009UL	Interrupt mask level = 9
SR_IMS10*		0x0000000aUL	Interrupt mask level = 10
SR_IMS11*		0x0000000bUL	Interrupt mask level = 11
SR_IMS12*		0x0000000cUL	Interrupt mask level = 12
SR_IMS13*		0x0000000dUL	Interrupt mask level = 13
SR_IMS14*		0x0000000eUL	Interrupt mask level = 14
SR_IMS15*		0x0000000fUL	Interrupt mask level = 15

Note: * Not defined in the μITRON4.0 specification.

6.31.2 Kernel Configuration Macros

Some kernel configuration macros will be output to kernel_macro.h, kernel_def.h, or kernel_cfg.h by cfg72mp.

(1) Range of Priority

Definition Name	Where Definition Is	Defined Value	Description
TMIN_TPRI	kernel.h	1	Minimum value of task priority
TMAX_TPRI	kernel_macro.h	system.priority	Maximum value of task priority
TMIN_MPRI	kernel.h	1	Minimum value of message priority
TMAX_MPRI	kernel_macro.h	system.message_pri	Maximum value of message priority

(2) Version Information

Definition Name	Where Definition Is	Defined Value	Description
TKERNEL_MAKER	kernel.h	0x0115	Kernel manufacturer code
TKERNEL_PRID		0x0013	Kernel identification number
TKERNEL_SPVER		0x5403	ITRON specification version number
TKERNEL_PRVER		*	Kernel version number

Note: * Refer to the release notes attached to the product. For example, the value of prver is 0x0100 for V.1.00 Release 00.

(3) Maximum Number of Queued Requests and Nesting Levels

Definition Name	Where Definition Is	Defined Value	Description
TMAX_ACTCNT	kernel.h	15U	Maximum activation request count
TMAX_WUPCNT		15U	Maximum wakeup request count
TMAX_SUSCNT		15U	Maximum suspension count
TMAX_SEMCNT		65535U	Maximum number of semaphore resources

(4) Number of Bits in Bit Pattern

Definition Name	Where Definition Is	Defined Value	Description
TBIT_TEXPTN	kernel.h	32U	Number of task exception cause bits
TBIT_FLGPTN		32U	Number of event flag bits

(5) Time Tick Cycle

Definition Name	Where Definition Is	Defined Value	Description
TIC_NUME	kernel_macro.h	system.tic_nume	Numerator of time tick cycle
TIC_DENO		system.tic_deno	Denominator of time tick cycle

(6) Other Kernel Configuration Macros Output to kernel_macro.h by cfg72mp (not in the μITRON4.0 Specification)

(a) Kernel configuration macros

Definition Name	Defined Value	Description
VTCFG_TBR	"_" prefixed to the symbol defined for system.tbr becomes the value defined for this macro.	
VTCFG_MPFMANAGE	"_" prefixed to the symbol defined for system.mpfmanage becomes the value defined for this macro.	
VTCFG_NEWMPL	"_" prefixed to the symbol defined for system.newmpl becomes the value defined for this macro.	
VTCFG_VECTYPE	"_" prefixed to the symbol defined for system.vector_type becomes the value defined for this macro.	
VTCFG_REGBANK	"_" prefixed to the symbol defined for system.regbank becomes the value defined for this macro.	
TIM_LVL	clock.IPL	Timer interrupt priority level

(b) Constant macros used in definitions

Classification	Definition Name	Where Definition Is	Defined Value
Common	_NOTUSE	kernel.h	0UL
For VTCFG_TBR	_NOMANAGE		0UL
	_FOR_SVC		1UL
	_TASK_CONTEXT		2UL
For VTCFG_MPFMANAGE	_IN		0UL
	_OUT		1UL
For VTCFG_NEWMPL	_PAST		0UL
	_NEW		1UL
For VTCFG_VECTYPE	_ROM_ONLY_DIRECT		0UL
	_RAM_ONLY_DIRECT		1UL
	_ROM		2UL
	_RAM		3UL
For VTCFG_REGBANK	_ALL		0x40000000UL
	_BANKLEVEL01		0x00000002UL
	_BANKLEVEL02		0x00000004UL
	_BANKLEVEL03		0x00000008UL
	_BANKLEVEL04		0x00000010UL
	_BANKLEVEL05		0x00000020UL
	_BANKLEVEL06		0x00000040UL
	_BANKLEVEL07		0x00000080UL
	_BANKLEVEL08		0x00000100UL
	_BANKLEVEL09		0x00000200UL
	_BANKLEVEL10		0x00000400UL
	_BANKLEVEL11		0x00000800UL
	_BANKLEVEL12		0x00001000UL
	_BANKLEVEL13		0x00002000UL
	_BANKLEVEL14		0x00004000UL
	_BANKLEVEL15		0x00008000UL

(7) Kernel Configuration Macros Output to kernel_def.h by cfg72mp (not in theμITRON4.0 Specification)

Among the kernel configuration macros output to kernel_def.h by cfg72mp, the macros whose specifications are to be open externally are described here. However, the compatibility with future versions is not guaranteed for the specifications of these macros.

Definition Name	Defined Value	Description
_SYSTEM_IPL	system.system_IPL	Kernel interrupt mask level
_MAX_STTSK	maxdefine.max_statictask	Maximum local ID of task using static stack
_MAX_INT	maxdefine.max_int	Maximum interrupt vector number

(8) Kernel Configuration Macros Output to kernel_cfg.h by cfg72mp (not in the μITRON4.0 Specification)

Among the kernel configuration macros output to kernel_cfg.h by cfg72mp, the macros whose specifications are to be open externally are described here. However, the compatibility with future versions is not guaranteed for the specifications of these macros.

Definition Name	Defined Value	Description
_SYSTEM_STACK_SIZE	system.stack_size	Interrupt stack size
_SYSTEM_KERNEL_STACK_SIZE	system.kernel_stack_size	Kernel stack size
_MAX_TSK	maxdefine.max_task	Maximum local task ID
_MAX_SEM	maxdefine.max_sem	Maximum local semaphore ID
_MAX_FLG	maxdefine.max_flag	Maximum local event flag ID
_MAX_DTQ	maxdefine.max_dtq	Maximum local data queue ID
_MAX_MBX	maxdefine.max_mbx	Maximum local mailbox ID
_MAX_MTX	maxdefine.max_mtx	Maximum local mutex ID
_MAX_MBF	maxdefine.max_mbf	Maximum local message buffer ID
_MAX_MPF	maxdefine.max_mpf	Maximum local variable-sized memory pool ID
_MAX_MPL	maxdefine.max_mpl	Maximum local fixed-sized memory pool ID
_MAX_CYH	maxdefine.max_cyh	Maximum local cyclic handler ID
_MAX_ALH	maxdefine.max_alh	Maximum local alarm handler ID
_MAX_FNCD	maxdefine.max_fncd	Maximum extended service call function code
_MEMSTK_ALLMEMSIZE	memstk.all_memsize	Size of default task stack area
_MEMDTQ_ALLMEMSIZE	memdtq.all_memsize	Size of default data queue area
_MEMMBF_ALLMEMSIZE	memmbf.all_memsize	Size of default message buffer area
_MEMMPF_ALLMEMSIZE	memmpf.all_memsize	Size of default fixed-sized memory pool area
_MEMMPL_ALLMEMSIZE	memmpl.all_memsize	Size of default variable-sized memory pool area

6.31.3 Function Macros Defined in itron.h

(1) ER MERCD(ER ercd)

Description: Returns the main error code for ercd.

Header File: itron.h

Parameters:	ercd	Error code

Return Values: Main error code for ercd

(2) ER SERCD(ER ercd)

Description: Returns the suberror code for ercd.

Header File: itron.h

Parameters:	ercd	Error code

Return Values: Suberror code for ercd

Remarks The suberror code of the error code that will be returned from the kernel is always –1.

(3) ER ERCD(ER mercd, ER sercd)

Description: Returns the error code consisting of the main error code (mercd) and suberror code (sercd).

Header File: itron.h

Parameters:	mercd	Main error code
	sercd	Suberror code

Return Values: Error code

Remarks The suberror code of the error code that will be returned from the kernel is always –1.

6.31.4 Function Macros Defined in kernel.h

(1) UH GET_CPUID(ID id)

Description: Returns the CPU ID of id.

Header File: kernel.h

Parameters: id Object ID

Return Values: CPU ID

Remarks: This macro is a function not defined in the μITRON4.0 specification.

(2) ID GET_LOCALID(ID id)

Description: Returns the local object ID of id.

Header File: kernel.h

Parameters: id Object ID

Return Values: Local object ID

Remarks: This macro is a function not defined in the μITRON4.0 specification.

(3) ID MAKE_ID(UH cpuid, ID localid)

Description: Returns the object ID consisting of the CPU ID (cpuid) and local object ID (localid).

Header File: kernel.h

Parameters: cpuid CPU ID

localid Local object ID

Return Values: Object ID

Remarks: This macro is a function not defined in the μITRON4.0 specification.

(4) SIZE TSZ_DTQ(UINT dtqcnt)

Description: Returns the size of a data queue area in which the dtqcnt number of data items can be stored.

Header File: kernel.h

Parameters: dtqcnt Number of data items

Return Values: Size of data queue area

(5) SIZE TSZ_MBF(UINT msgcnt, UINT msgsz)

Description:	Returns the approximate size of a message buffer area in which msgcnt number of msgsz-byte messages can be stored.	
Header File:	kernel.h	
Parameters:	msgcnt	Number of messages
	msgsz	Message size
Return Values:	Approximate size of message buffer area	

(6) SIZE TSZ_MPF(UINT blkcnt, UINT blksz)

Description:	Returns the size of a fixed-sized memory pool from which blkcnt number of blksz-byte memory blocks can be acquired.	
Header File:	kernel.h	
Parameters:	blkcnt	Number of memory blocks
	blksz	Memory block size
Return Values:	Size of fixed-sized memory pool	
Remarks:	The value returned from this macro differs depending on system.mpfmanage.	

(7) SIZE VTSZ_MPFMB(UINT blkcnt, UINT blksz)

Description:	Returns the size of the management area required for a fixed-sized memory pool from which blkcnt number of blksz-byte memory blocks can be acquired.	
Header File:	kernel.h	
Parameters:	blkcnt	Number of memory blocks
	blksz	Memory block size
Return Values:	Size of fixed-sized memory pool management area	
Remarks:	This macro is defined only when system.mpfmanage==OUT. This macro is a function not defined in the μITRON4.0 specification.	

(8) SIZE TSZ_MPL(UINT blkcnt, UINT blksz)

Description:	Returns the approximate size of a variable-sized memory pool from which blkcnt number of blksz-byte memory blocks can be acquired.	
Header File:	kernel.h	
Parameters:	blkcnt	Number of memory blocks
	blksz	Memory block size
Return Values:	Approximate size of variable-sized memory pool	
Remarks:	The value returned from this macro differs depending on system.newmpl.	

6.32 Directory and File Structure

<RTOS_INST>\os\include\

itron.h	ITRON specification definition file
kernel.h	Kernel specification definition file
kernel_api.h	Service call API definition file
kernel_dbg.h	Debugging function API definition file
sh2afpu.h	Header file for using the FPU in handlers

<RTOS_INST>\os\lib\release\

hiknl.lib	Base library (without debugging information)
fpu_knl.lib	Patch library with FPU support (without debugging information)
hiexpand.lib	Patch library without FPU support for debugging (without debugging information)
fpu_expand.lib	Patch library with FPU support for debugging (without debugging information)

<RTOS_INST>\os\system\	System definition files

The following directories are provided with only a product that includes the source code.

<RTOS_INST>\os\lib\debug\	
hiknl.lib	Base library (with debugging information)
fpu_knl.lib	Patch library with FPU support (with debugging information)
hiexpand.lib	Patch library without FPU support for debugging (with debugging information)
fpu_expand.lib	Patch library with FPU support for debugging (with debugging information)
<RTOS_INST>\os\kernel\	Workspace, etc. for creating libraries
<RTOS_INST>\os\kernel\knl_src\	Source code
<RTOS_INST>os\kernel\fpu_expand\	Project directory
<RTOS_INST>os\kernel\fpu_expand\debug\	Configuration directory (with debugging information)
<RTOS_INST>os\kernel\fpu_expand\release\	Configuration directory (without debugging information)
<RTOS_INST>os\kernel\fpu_knl\	Project directory
<RTOS_INST>os\kernel\fpu_knl\debug\	Configuration directory (with debugging information)
<RTOS_INST>os\kernel\fpu_knl\release\	Configuration directory (without debugging information)
<RTOS_INST>os\kernel\hiexpand\	Project directory
<RTOS_INST>os\kernel\hiexpand\debug\	Configuration directory (with debugging information)
<RTOS_INST>os\kernel\hiexpand\release\	Configuration directory (without debugging information)
<RTOS_INST>os\kernel\hiintfc\	Project directory
<RTOS_INST>os\kernel\hiintfc\debug\	Configuration directory (with debugging information)
<RTOS_INST>os\kernel\hiintfc\release\	Configuration directory (without debugging information)
<RTOS_INST>os\kernel\hiknl\	Project directory
<RTOS_INST>os\kernel\hiknl\debug\	Configuration directory (with debugging information)
<RTOS_INST>os\kernel\hiknl\release\	Configuration directory (without debugging information)
<RTOS_INST>os\kernel\intdwn\	Project directory
<RTOS_INST>os\kernel\intdwn\debug\	Configuration directory (with debugging information)
<RTOS_INST>os\kernel\intdwn\release\	Configuration directory (without debugging information)
<RTOS_INST>os\kernel\svcapi\	Project directory
<RTOS_INST>os\kernel\svcapi\debug\	Configuration directory (with debugging information)
<RTOS_INST>os\kernel\svcapi\release\	Configuration directory (without debugging information)
<RTOS_INST>os\kernel\svcrmt\	Project directory
<RTOS_INST>os\kernel\svcrmt\debug\	Configuration directory (with debugging information)
<RTOS_INST>os\kernel\svcrmt\release\	Configuration directory (without debugging information)

6.33 Building the Library (Only for a Product with the Source Code)

Building the library is not usually necessary. If you wish to build the library (e.g. for debugging), you should use the provided High-performance Embedded Workshop workspace (kernel.hws).

Section 7 RPC Library

7.1 Overview

The RPC is used to execute functions registered in advance in another CPU. Figure 7.1 shows a conceptual diagram of the RPC.

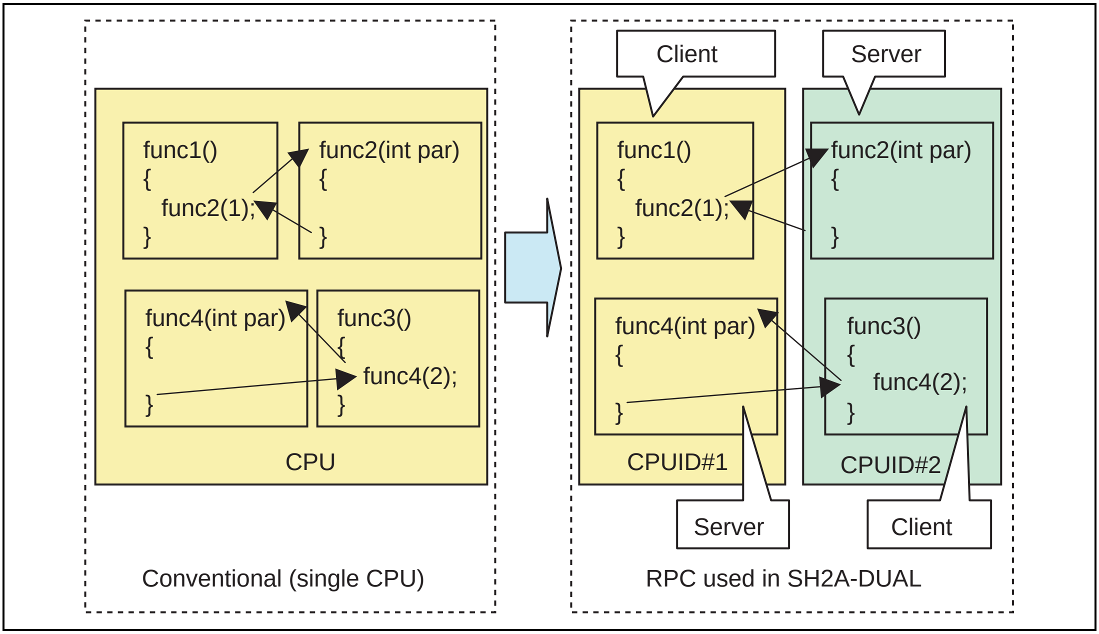


Figure 7.1 Conceptual Diagram of RPC

In the RPC, the function that calls a function is referred to as the client, whereas, the function that executes the function is referred to as the server.

A server of the current CPU can also be called by the RPC. This allows (static) CPU switching to be performed relatively easily for each server. However, it must be noticed that calling a server of the current CPU naturally takes more time than a normal function call.

Table 7.1 Overview of RPC Library

No.	Item	Component
1	Hardware resources used by this software	None
2	Software components used by this software	(1) OAL (2) IPI (3) Spinlock library
3	Other software components using this software	None

7.2 Overview of RPC Operation

The RPC configuration and operation with the client as CPUID#1 and the server as CPUID#2 are described with reference to figure 7.2.

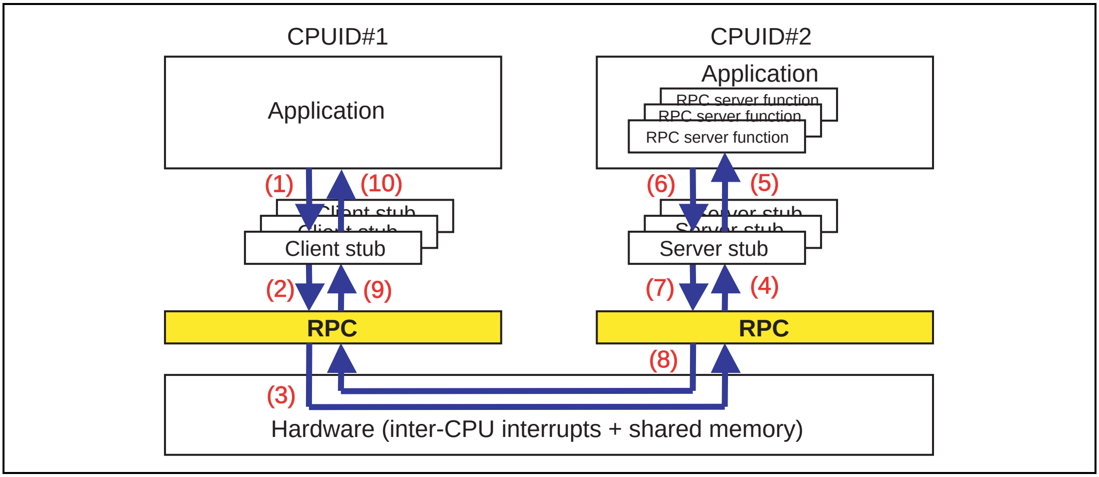


Figure 7.2 RPC Configuration and Operation

(1) The CPUID#1 application makes a function call with the same API as the RPC server function in order to have CPUID#2 execute the RPC server function.

(2) In CPUID#1, a client stub with the same name and having the same API as the RPC server function will be called. The client stub converts the I/O parameters into a format understandable by the RPC and then calls the remote function call API (rpc_call) of the RPC. Note that the client stub has to be implemented by the server creator.

(3) rpc_call() of CPUID#1 is executed in the same context as the called task. rpc_call() transfers the received information to the server area and wakes up the CPUID#2 server task using an IPI primitive. The caller of rpc_call() is kept in the WAITING state in rpc_call() until execution of the RPC server function has finished.

The server area should be allocated in non-cacheable shared memory.

(4) The CPUID#2 server task calls the server stub corresponding to the request. The server stub has to be implemented by the user and must be registered in advance. The server stub is executed in the server task context defined for each server.

Note that a server task is created and initiated in advance by the server start API (rpc_start_server()).

(5) Information, such as, input parameters that were transferred in step (3) is passed to the server stub. Based on the information given from the RPC, the server stub transforms the input parameters to match the interface specifications for the RPC server function and then it calls the RPC server function. Note that the server stub has to be implemented by the server creator.

(6) The RPC server function returns to the server stub.

(7) The server stub sets the output information and returns to the server task.

(8) The server task, using an IPI primitive, wakes up the client task that was shifted to the WAITING state in step (3).

(9) The woken up client task transfers the output information set in step (7) to the client area as part of the rpc_call() processing and then returns to the client stub.

(10) The client stub performs necessary processing, such as, setting of the return information, and then returns to the caller.

7.3 Server

7.3.1 Server ID

The server ID is information for identifying the server, and the ID of each server must be unique in the entire system.

The server ID is specified when the server is started. Normally, the server ID needs to be statically determined by the system engineer so that the same server ID is not used for more than one server in the entire system.

The server ID is represented as a 32-bit unsigned integer. To facilitate making rules for preventing the same server ID from being used, any value can be set.

However, a server ID of 0x80000000 or higher should not be used because it is kept for future use by the OS.

The maximum number of servers that can be registered in each CPU is specified in rpc_init() used for initiating the RPC library.

7.3.2 Function ID

A server has at least one server function. Each server function is identified by a serial function ID starting from 0. The function ID is specified when a server call (rpc_call) is made.

A server can have up to 32767 function IDs.

7.3.3 Server Task

The server stub and server function are executed in the server task context. This means that the server stub is called from the server task.

The code entity of the server task is within the RPC library, and the server task is created when the server is started. The priority of the server task and the stack size used by the server task can be specified when the server is started.

7.3.4 Server Stub and Server Function

The server stub is called from the server task within the RPC library. The parameter information to be passed to the server function is transferred to the server stub in the format defined by the RPC specifications. The server stub converts this information to match the API specifications for the specified server function and then calls the specified server function.

The server stub has to be implemented by the server creator.

7.3.5 Client Stub

The client stub has the same API specifications as those for the specified function. The client stub converts the parameters into the format defined by the RPC specifications and then calls the RPC.

The client stub has to be implemented by the server creator.

7.3.6 Server Conflicts

This RPC is designed to have only a single server task for each server in order to reduce the overhead by simplifying management of the parameter transfer area between the client and server.

This means the server can process only one RPC call request at a time.

When two or more RPC calls are requested to the same server simultaneously, the server processes the requests in the order they were called. A client waiting to be processed will be blocked.

7.4 Synchronous Mode and Asynchronous Mode

The RPC supports synchronous mode (RPC_ACK) and asynchronous mode (RPC_UNACK) as call modes. In which mode the call will be made is specified when the RPC call request is made.

For an RPC call in synchronous mode, the client task is blocked until server execution is finished. The client can acquire the data output from the server.

For an RPC call in asynchronous mode, the client task immediately returns after making the request for server execution. In asynchronous mode, the client cannot acquire the data output from the server. Furthermore, the client does not have means to acknowledge whether the server processing has finished.

Note however that in either mode, there is a possibility that the client will be blocked in a case shown in section 7.3.6, Server Conflicts.

7.5 Sending and Receiving Parameters

7.5.1 Features

- The RPC itself does not have a buffer for sending and receiving parameters.
- The server has a "server parameter area" for storing the input parameters from the client and the output parameters to the client. The server parameter area must be allocated in a non-cacheable area.
- The server parameter area can be allocated by the server creator in a static manner or can be automatically allocated by the RPC in an on-demand manner.
- The input parameters specified by the client are directly copied to the server parameter area. Similarly, the output parameters from the server are directly copied to the area specified by the client. Accordingly, either type of copy is performed once.
- Copy is performed by the RPC library in the client using a CPU instruction or by the user-created function called back from the RPC library in the client. When the latter copy method is used, the parameters can be copied through DMA transfer.

7.5.2 IOVEC Structure

In the RPC, the input parameters (client → server) and output parameters (server → client) are specified by a structure array called IOVEC. The IOVEC structure allows parameters scattered over noncontiguous memory to be handled efficiently.

```
typedef struct {
    void *pBaseAddress;      // Start address of data area
    UINT32 ulSize;           // Area size (number of bytes)
} IOVEC;
```

Note that ulSize = 0 means there is no area.

7.5.3 Server Parameter Area

The server has a parameter area for storing the input parameters from the client and the output parameters to the client. In the RPC, the server can process only one client request at a time. Therefore, each server has only one server parameter area.

There are two methods for allocating the server parameter area: allocate an area on-demand or use an area that has been allocated statically. A server using the former method is called a "dynamic server" and will be started using rpc_start_server().A server using the latter method is called a "static server" and will be started using rpc_start_server_with_paramarea().

In either case, the server area must be allocated in a non-cacheable area for maintaining the coherency between the server and client.

(1) Dynamic Server

A dynamic server is started using rpc_start_server().

When a dynamic server is called via an RPC call, the server allocates an area of the size required by that call using OAL_GetMemory(). This area is released when processing of the call has finished.

The OAL must be configured so that the memory allocated by OAL_GetMemory() becomes a non-cacheable area.

(2) Static Server

A static server is started using rpc_start_server_with_paramarea().

In rpc_start_server_with_paramarea(), the address and size of the server parameter area are specified. The server parameter area is allocated by the application.

Though the dynamic server can be used to make efficient use of memory when the size of the I/O parameters is not constant, the processing time is longer than the static server because memory is allocated and released dynamically.

7.5.4 Server Parameter Area Size Required by RPC Call

rpc_call() or rpc_call_copycbk() is used to request an RPC call.

The server parameter area size necessary for accepting an RPC call can be calculated from the formula below.

```
Necessary size = sizeof(rpc_server_stub_info)
   + ∑ ALIGNUP4 (pCallInfo->pInputIOVectorTable->ulSize)
   + (pCallInfo->ulOutputIOVectorTableSize) × sizeof(IOVEC) ........................... (a)
   + ∑ ALIGNUP4 (pCallInfo->pOutputIOVectorTable->ulSize) .......................... (b)
```

(a) and (b) are both calculated as 0 when asynchronous mode is specified.

ALIGNUP4 (data) which is defined as a function macro in types.h indicates that "data" has been rounded up to a multiple of 4.

(1) For a Dynamic Server

When an RPC call has been requested, the server allocates memory for the size calculated in the above formula using OAL_GetMemory(). When memory allocation fails, rpc_call() returns an RPC_E_NOMEM error.

In a case where ulMaxParamAreaSize is specified as a value other than 0 in rpc_start_server(), if the requested size in rpc_call() exceeds ulMaxParamAreaSize, the server does not execute OAL_GetMemory() and the RPC call returns an RPC_E_PAR error.

(2) For Static Server

If the size calculated in the above formula exceeds ulMaxParamAreaSize specified in rpc_start_server_with_paramarea(), the RPC call returns an RPC_E_PAR error.

7.5.5 Parameter Copy Methods

(1) Using rpc_call()

Both the input and output parameters are copied by rpc_call() in the client.

To be specific, the input parameters specified by the client are copied to the server parameter area by rpc_call() in the client. The output parameters set in the server parameter area by the server stub are also copied to the output parameter area specified by the client by rpc_call() in the client.

(2) Using rpc_call_copycbk()

In rpc_call_copycbk(), the callback functions for performing the copy process is specified. Two callback functions will be specified. Callback functions are called back respectively by rpc_call_copycbk() in the client when the input parameters need to be copied to the server parameter area and when the output parameters need to be copied from the server parameter area to the output parameter area specified by the client.

7.5.6 Application Examples

(1) Reduction of Copy Overhead

In a multiprocessor system where the memory spaces are shared, the copy overhead can be reduced by sending and receiving only the pointer.

In the SH2A-DUAL, a multicore system without a cache snoop controller, the coherency between the client CPU's cache and server CPU's cache cannot be guaranteed. Therefore, the area indicated by the pointer must be a non-cacheable area.

(2) Utilization of Cache

The server acquires input parameters from the server parameter area and also sets the output parameters. Since the server parameter area is a non-cacheable area, performance degradation caused by accesses to these parameters not being cached may become a problem depending on how frequently the server accesses these parameters.

In such a case, the server should first transfer the parameters to an area allocated in a cacheable area.

7.6 OS Resources Used by RPC

7.6.1 Task

Each server has a server task.

A server task is created by rpc_start_server() or rpc_start_server_with_paramarea() and deleted by rpc_stop_server(). The priority of the server task and the stack size used by the server task are specified in rpc_start_server() or rpc_start_server_with_paramarea().

The entry function of a server task is contained in the RPC library, and this entry function calls the server stub.

Note that all API functions of the RPC library are executed as normal functions. Accordingly, they are executed in the same context as the caller.

7.6.2 OAL_GetMemory()

In the RPC, memory is allocated by OAL_GetMemory() in the following cases. The OAL must be configured so that the memory allocated by OAL_GetMemory() becomes a non-cacheable area.

(1) Parameter Area Allocation by Dynamic Server

When rpc_call() or rpc_call_copycbk() is issued for a dynamic server, the server task allocates the parameter area using OAL_GetMemory().

(2) Waiting for Server to be Called

When rpc_call() or rpc_call_copycbk() is kept waiting for the specified server to become free, a memory area for managing the WAITING state is allocated using OAL_GetMemory().

7.6.3 IPI

In the RPC, a single IPI port is used. rpc_init() is used to specify which IPI port is to be used.

The IPI must be configured so that the IPI port specified by rpc_init() is usable.

7.6.4 Spinlock Library

In the RPC, RW lock is used for exclusive control between the CPUs.

7.7 Provided Files

<RTOS_INST>\os\include\	
rpc_pubic.h	API definition header file
<RTOS_INST>\os\lib\debug\ *	
rpc.lib *	Library (with debugging information)
<RTOS_INST>\os\lib\release\	
rpc.lib	Library (without debugging information)
<SAMPLE_INST>\R0K572650D000BR\cpuid1\rpc_config\	
rpc_table.c	Management table (see section 7.9, Building the System)
<SAMPLE_INST>\R0K572650D000BR\cpuid2\rpc_config\	
rpc_table.c	Management table (see section 7.9, Building the System)
<RTOS_INST>\os\rpc\ *	Workspace, etc. for creating the library
<RTOS_INST>\os\rpc\rpc\ *	Project, etc. for creating the library
<RTOS_INST>\os\rpc\rpc\include\ *	Internal definition file
<RTOS_INST>\os\rpc\rpc\source\ *	Source code

The directories with an asterisk (*) are provided only in products with the source code.

Note that rpc_table.c has the same contents in each directory.

7.8 Building the Library (Only for a Product with the Source Code)

Building the library is not usually necessary. If you wish to build the library (e.g. for debugging), you should use the provided High-performance Embedded Workshop workspace (rpc.hws).

7.9 Building the System

7.9.1 Configuration of Kernel

When using the RPC, the kernel must be configured suitably.

(1) system.system_IPL

The interrupt level of rpc_config.ulIPIPortID specified in rpc_init() must be lower than or equal to system.system_IPL.

(2) maxdefine.max_task and memstk.all_memsize

In rpc_start_server() or rpc_start_server_with_paramarea(), a server task is created by OAL_CreateTask() (acre_tsk). The number of server tasks that have the possibility of being created at the same time is rpc_config.ulTableSize which is specified in rpc_init(). This must be taken into consideration when specifying maxdefine.max_task.

The server task uses the default task stack area. This must be taken into consideration when specifying memstk.allmemsize.

(3) service_call

The RPC (OAL to be accurate) uses the following service calls so they must be installed.

- acre_tsk
- act_tsk
- exd_tsk
- slp_tsk
- wup_tsk
- acre_mpl
- del_mpl
- pget_mpl
- rel_mpl
- get_tid
- dis_dsp
- ena_dsp
- sns_ctx
- sns_dsp
- sns_dpn

7.9.2 Configuration of IPI

In rpc_init(), an IPI port is created (IPI_Create()) with the specified rpc_config.ulIPIPortID.

The IPI must be configured so that this IPI port is usable.

7.9.3 Building the System

Programs that use functions of this API must be linked to the RPC library.

rpc_table.c is compiled for each CPU and linked to the RPC library. rpc_table.c must not be edited.

For the sections of the RPC library and rpc_table.c, refer to section 17.5.2, Sections.

7.10 API Functions

Table 7.2 API Functions

No.	Classification	API Name	Function
1	Initialization	rpc_init	Initializes RPC library
2	Termination	rpc_shutdown	Terminates RPC library
3	For the server	rpc_start_server	Starts dynamic server
4		rpc_start_server_with_ paramarea	Starts static server
5		rpc_stop_server	Stops server
6	For the client	rpc_connect	Connects server
7		rpc_disconnect	Disconnects server
8		rpc_call	Calls server function
9		rpc_call_copycbk	Calls server function (data transfer callback)
10	Others	rpc_get_server_properties	Acquires server properties

7.10.1 Header File

Include rpc_public.h.

7.10.2 Basic Data Types

In the RPC library, the basic data types defined in types.h are used.

For types.h, refer to section 19, types.h.

The structure used in each API function is described in the relevant API section.

7.10.3 Initialize RPC Library (rpc_init)

C-Language API:

```
INT32 rpc_init(rpc_config *pConfig);
```

Parameters:

pConfig	Pointer to the RPC library initialization information packet

Packet Structure:

```
typedef     struct  {
                rpc_info          *pRpcTable;
                UINT32            ulTableSize;
                UINT32            ulCmdRspRangeBaseValue;
                UINT32            RedirectionTaskStackSize;
                UINT32            ServerTaskStackSize;
                UINT32            MFIFramePriority;
                UINT32            RPCTaskPriority;
                UINT32            ulIPIPortID;
} rpc_config;
```

Note that the rpc_info structure is not described because it is an RPC internal specification.

Return Values:

RPC_E_OK	Normal end
RPC_E_SYS	The OS state is invalid
RPC_E_NOINIT	RPC is not initialized (only when MYCPUID is a value other than 1)
RPC_E_NORESOURCE	Failed in IPI port creation

Function:

Initializes the RPC library environment of the current CPU according to the pConfig contents. When the current CPU is CPUID#1, the RPC library environment shared by the CPUs is initialized.

The IPI primitives IPI_init() and OAL_Init() have to be finished before calling this function. To call this function in CPUID#2, rpc_init() has to be finished in CPUID#1 before then.

This function must be called from a context state of the task level with the interrupts not masked. In such a state, this function can be called even when preempt is disabled.

This function should be called only once in each CPU at the beginning. Even when no server is created in the current CPU, initialization by this function is required when an RPC call is requested to another CPU.

(1) pRpcTable and ulTableSize

These specify the table area for managing the RPC servers. The number of servers that can be created simultaneously in the current CPU is specified in ulTableSize. If no server is created in the current CPU, specify ulTableSize as 0.

Allocate a non-cacheable area of the size calculated in the formula below and specify the start address in pRpcTable. pRpcTable must be an address at the 4-byte boundary.

```
Size = sizeof(rpc_info) × ulTableSize
```

When ulTableSize is 0, the members in the rpc_config_info structure, except for ulIPIPortID, are all ignored.

(2) ServerTaskStackSize

Specifies the stack size used by the server task.

For details, refer to section 18.6.4, Stack Size Used by SVC Server Task (remote_svc.stack_size).

(3) ulIPIPortID

Specifies the ID of the IPI port used to accept return notification from the RPC requested to another CPU and also used to accept RPC requests from another CPU.

In this function, the IPI port specified by ulIPIPortID is created by IPI_create().

The interrupt level of ulIPIPortID must be lower than or equal to the kernel interrupt mask level (system.system_IPL).

(4) ulCmdRspRangeBaseValue, RedirectionTaskStackSize, MFIFramePriority, and RPCTaskPriority

These are reserved for future expansion and are simply ignored.

7.10.4 Terminate RPC Library (rpc_shutdown)

C-Language API:

```
INT32 rpc_shutdown(void);
```

Return Values:

```
RPC_E_OK            Normal end
RPC_E_SYS           The OS state is invalid
RPC_E_NOINIT        RPC is not initialized
RPC_E_STATE         A started server exists
```

Function:

Terminates the RPC library environment of the current CPU. The IPI port created in rpc_init() is deleted using IPI_delete(). However, if there is a server already started in the current CPU, an error is returned.

When the current CPU is CPUID#1, the RPC library environment shared by the CPUs is also terminated. However, if there is a server already started in another CPU, an error is returned.

When this function is executed successfully, the API functions of the RPC requested from the current CPU will all return an RPC_E_NOINIT error from here on. When the current CPU is CPUID#1, the API functions of the RPC requested from all CPUs will all return an RPC_E_NOINIT error from here on.

This function must be called from a context state of the task level with the interrupts not masked. In such a state, this function can be called even when preempt is disabled.

When this function is called before initialization by rpc_init() has been performed, the operation is undefined.

7.10.5 Start Dynamic Server (rpc_start_server)

C-Language API:

```
INT32 rpc_start_server( rpc_server_info *pServerInfo );
```

Parameters:

pServerInfo	Pointer to the server registration information packet

Packet Structure:

```
typedef     struct  {
                UINT32          ulRPCServerID;
                UINT32          ulRPCServerVersion;
                UINT32          ServerStubTaskPriority;
                UINT32          (**ServerStubList)(rpc_server_stub_info *);
                UINT32          ulNumFunctions;
                UINT32          ulStubStackSize;
                UINT32          ulMaxParamAreaSize;
                void            *pUserDefinedData;
} rpc_server_info;
For rpc_server_stub_info, refer to section 7.11.1, Server Stub.
```

Return Values:

RPC_E_OK	Normal end
RPC_E_SYS	The OS state is invalid (failed in OAL_CanWait())
RPC_E_NOINIT	RPC is not initialized
RPC_E_PAR	Parameter error 0 < ulMaxParamSize < sizeof (rpc_server_stub_info)
RPC_E_NORESOURCE	Servers for the number of rpc_config.ulTableSize are already started
RPC_E_STATE	Server of ulServerID is already started
RPC_E_CREATETASK	Failed in server task creation

Function:

Starts the server in the current CPU according to the information specified in pServerInfo.

The server started by this API dynamically allocates the parameter area used for communication with the client using OAL_GetMemory() as soon as a call request from the client has been accepted.

This API creates a server task for the server to be registered and initiates the server task. The server task is kept waiting (WAITING state) by OAL_SleepTask() until it is called by the client.

(1) ulRPCServerID

Specifies the server ID to be registered.

If a server ID already registered is specified, an error is returned.

(2) ulRPCServerVersion

Specifies the version of the server to be registered.

(3) ServerStubTaskPriority

Specifies the priority of the server task. The relationship with the other tasks in the CPU in which the server operates must be taken into consideration when specifying the priority. When a priority not supported by the OS is specified, an error is returned.

(4) ServerStubList and ulNumFunctions

ServerStubList is the address of the function table that holds the server stub function address of the function ID from 0 to (ulNumFunctions – 1).

Since this function table is referenced by the RPC library until the server is stopped, it must be created in a static area.

(5) ulStubStackSize

Stands for the stack size used by the server stub.

For the method of calculating the stack size, refer to section 18.11.2, RPC Library.

(6) ulMaxParamAreaSize

Specifies the maximum acceptable size for the parameter area dynamically allocated by the server. For an RPC call request that requires a parameter area larger than this limit, an error is returned.

If ulMaxParamAreaSize is specified as 0, the acceptable size is unlimited. However, when the limit for the allocatable size by OAL_GetMemory() is exceeded, the RPC call returns an error.

(7) pUserDefinedData

The data specified here is passed to the server stub without changes. The RPC does not make use of this information at all. The data does not have to be a pointer.

This function must be called from a context state of the task level with the interrupts not masked and preempt enabled.

7.10.6 Start Static Server (rpc_start_server_with_paramarea)

C-Language API:

```
INT32 rpc_start_server_with_paramarea(
          rpc_server_info *pServerInfo,
          UINT8 *pParamArea );
```

Parameters:

pServerInfo	Pointer to the server registration information packet
pParamArea	Start address of parameter area

Packet Structure:

```
typedef     struct  {
               UINT32           ulRPCServerID;
               UINT32           ulRPCServerVersion;
               UINT32           ServerStubTaskPriority;
               UINT32           (**ServerStubList)(rpc_server_stub_info *);
               UINT32           ulNumFunctions;
               UINT32           ulStubStackSize;
               UINT32           ulMaxParamAreaSize;
               void             *pUserDefinedData;
} rpc_server_info;
```

For rpc_server_stub_info, refer to section 7.11.1, Server Stub.

Return Values:

RPC_E_OK	Normal end
RPC_E_SYS	The OS state is invalid (failed in OAL_CanWait())
RPC_E_NOINIT	RPC is not initialized
RPC_E_PAR	Parameter error ulMaxParamSize < sizeof (rpc_server_stub_info)
RPC_E_NORESOURCE	Servers for the number of rpc_config.ulTableSize are already started
RPC_E_STATE	Server of ulServerID is already started
RPC_E_CREATETASK	Failed in server task creation

Function:

Starts the server in the current CPU according to the information specified in pServerInfo.

The server created by this API uses the area specified by pParamArea and ulMaxParamAreaSize as the parameter area used for communication with the client.

This API creates a server task for the server to be registered and initiates the server task. The server task is kept waiting (WAITING state) by OAL_SleepTask() until it is called by the client (by rpc_call).

Only the differences with the parameters in rpc_start_server() are described below.

(1) ulMaxParamAreaSize and pParamArea

Allocate a free area of ulMaxParamAreaSize bytes and specify the start address in pParamArea.

Since this server parameter area is referenced by the RPC library until the server is stopped, it must be allocated in a static area. The server parameter area must also be aligned to the 4-byte boundary. pParamArea must be an address at the 4-byte boundary.

In the SH2A-DUAL, the parameter area must be allocated in a non-cacheable area.

7.10.7 Stop Server (rpc_stop_server)

C-Language API:

```
INT32 rpc_stop_server(
          UINT32  ulServerID;
          UINT32  ulServerVersion,
          void    (*cbk)(UINT32),
          UINT32  ulParam);
```

Parameters:

ulServerID	Server ID
ulServerVersion	Server version
cbk	Server stop callback function
ulParam	Data passed to server stop callback function

Return Values:

RPC_E_OK	Normal end
RPC_E_SYS	The OS state is invalid (failed in OAL_CanWait())
RPC_E_NOINIT	RPC is not initialized
RPC_E_STATE	Server of ulServerID is not started in current CPU
RPC_E_VER	Version does not match

Function:

Stops the server with the server ID specified in ulServerID. The version of the server to be stopped is specified in ulServerVersion.

In this API, servers of another CPU cannot be stopped.

The task that was waiting because it had called rpc_call() is released from the WAITING state, and an error is returned.

This API ends normally even when the server is in the middle of processing a client request. However, the server will actually be stopped when the server function being processed has finished.

When the server has completely stopped, the callback function specified by cbk is called. If NULL is specified in cbk, the callback function will not be called.

The callback function should simply notify a certain event and then return without delay. The callback function should not be used to block a task or perform a process other than event notification.

7.10.8 Connect Server (rpc_connect)

C-Language API:

```
INT32 rpc_connect (
          UINT32  ulServerID;
          UINT32  ulServerVersion);
```

Parameters:

```
ulServerID         Server ID
ulServerVersion    Server version
```

Return Values:

```
RPC_E_OK           Normal end
```

Function:

Connects the server specified in ulServerID. The specified server should be connected using this API before issuing rpc_call().

This API is reserved for future expansion. In the current implementation, this API is defined in rpc_public.h as shown below to always normally end and the above specification is not implemented.

```
#define rpc_connect(ulServerID, ulServerVersion)  RPC_E_OK
```

7.10.9 Disconnect Server (rpc_disconnect)

C-Language API:

```
INT32 rpc_disconnect(
          UINT32  ulServerID;
          UINT32  ulServerVersion);
          void    (*cbk)(UINT32),
          UINT32  ulParam);
```

Parameters:

ulServerID	Server ID
ulServerVersion	Server version
cbk	Callback function
ulParam	Parameter passed to callback function

Return Values:

RPC_E_OK	Normal end

Function:

Cancels connection with the server specified in ulServerID.

A task other than the client task that has requested connection can cancel connection using this API as long as it is in the same CPU or OS the client task that has requested connection.

When connection is canceled, the callback function specified by cbk is called. If NULL is specified in cbk, the callback function will not be called.

The callback function should simply notify a certain event and then return without delay. The callback function should not be used to block a task or perform a process other than event notification.

This API is reserved for future expansion. In the current implementation, this API is defined in rpc_public.h as shown below to always normally end and the above specification is not implemented.

```
#define rpc_disconnect(ulServerID, ulServerVersion, cbk, ulParam)  RPC_E_OK
```

7.10.10 Call Server Function (rpc_call)

C-Language API:

```
INT32 rpc_call(rpc_call_info *pCallInfo );
```

Parameters:

pCallInfo	Pointer to the server function call information packet

Packet Structure:

```
typedef    struct  {
              UINT32                ulMarshallingType;
              UINT32                ulServerID;
              UINT32                ulServerVersion;
              UINT32                ulServerProcedureID;
              IOVEC                 *pInputIOVectorTable;
              UINT32                ulInputIOVectorTableSize;
              IOVEC                 *pOutputIOVectorTable;
              UINT32                ulOutputIOVectorTableSize;
              UINT32                *pulLastOutputIOVectorSize;
              UINT32                *pulReturnValue;
              enum rpc_ack_mode     AckMode;
} rpc_call_info;
```

Return Values:

RPC_E_OK	Normal end
RPC_E_SYS	The OS state is invalid (failed in OAL_CanWait())
RPC_E_NOINIT	RPC is not initialized
RPC_E_GETTASKID	Failed in OAL_GetTaskID()
RPC_E_STATE	Server of ulServerID is not started
RPC_E_VER	Version does not match
RPC_E_PARM	(1) AckMode is other than RPC_ACK or RPC_UNACK (2) Necessary size for calling exceeded the acceptable size of the server (3) ulProcedureID ≥ rpc_server_info.ulNumFunctions
RPC_E_STOP	Server was stopped during call waiting
RPC_E_NOMEM	Insufficient memory (1) Failed in allocation of call waiting management area (2) The server has failed in allocation of parameter area

Function:

Calls the server function of the function ID specified in ulServerProcedureID of the server specified in ulServerID. The server function is executed by the server CPU.

The contents of the input vectors specified by pInputIOVectorTable are all transferred to the parameter area of the server.

RPC_ACK (synchronous mode) or RPC_UNACK (asynchronous mode) can be specified as the call mode (AckMode).

In synchronous mode, the task that has called this API waits in this API until execution of the server function finishes and then returns from this API.

In asynchronous mode, this API requests execution to the server and then returns without waiting for execution of the server function to finish.

The data output from the server can be received in synchronous mode but not in asynchronous mode.

This API is processed in the following phases.

(1) Phase 1: Acquires Right to Use Server (Call Waiting)

Acquires the right to use the server.

If the specified server is processing another client request, the right to use the server cannot be obtained. This API is kept waiting by OAL_SleepTask() until that client request is finished. This is called "call waiting". Call waiting is performed regardless of the AckMode setting. More than one client task may enter this waiting state. These client tasks are managed on a FIFO basis.

Before issuing OAL_SleepTask(), OAL_GetMemory() should be issued to allocate memory for managing the waiting state. When OAL_GetMemory() fails, an RPC_E_NOMEM error is returned immediately.

(2) Phase 2: Allocates Server Parameter Area (Dynamic Server Only)

Requests allocation of the parameter area to the specified server and is then kept waiting by OAL_SleepTask() until that process is completed. This waiting is performed regardless of the AckMode setting.

When the server fails in parameter area allocation, an RPC_E_NOMEM error is returned immediately.

(3) Phase 3: Transfers Input Parameters

The contents of the input parameter area defined by the specified input IOVEC array (arrays for the number of ulInputIOVectorTableSize, starting from pInputIOVectorTable) are copied to the server parameter area.

(4) Phase 4: Requests Execution to Server

Requests execution of the server function specified in ulServerProcedureID.

In synchronous mode, the task that has called this API is kept waiting in this API by OAL_SleepTask() until execution of the server function finishes.

On the other hand, in asynchronous mode, the task that has called this API does not wait and immediately returns from this API without executing the subsequent phases. Processing equivalent to phases 6 and 7 is performed when the RPC server task returns from the server stub function.

(5) Phase 5: Retrieves Output Parameters

In synchronous mode, the output parameters set in the server parameter area are copied to the area defined by pOutputIOVectorTable and ulOutputIOVectorTableSize.

(6) Phase 6: Releases Server Parameter Area (Dynamic Server Only)

Requests release of the parameter area to the server and is then kept waiting by OAL_SleepTask() until that process is completed.

(7) Phase 7: Releases Right to Use Server

Releases the right to use the server. This allows the server to process another client request. If there are tasks waiting to call the server, the task at the head of the waiting queue is woken up. Processing is resumed from phase 2 for that task.

The parameters are described as follows:

(1) ulMarshallingType

Specifies the marshalling type. How to handle the marshalling type should be determined between the client stub and server stub. The RPC library itself does not use this information. ulMarshallingType is passed to the server stub without changes.

When the following conditions are satisfied, ulMarshallingType does not need to be used in general.

(a) The client's CPU and server's CPU are the same (byte order, etc.) (SH2A-DUAL falls under this category).

(b) The client and server have the same compiler environment.

(2) ulServerID, ulServerVersion, and ulServerProcedureID

The server function specified by ulServerProcedureID for the server specified by ulServerID is called.

When the version of the specified server does not match ulServerVersion, an error is returned.

(3) pInputIOVectorTable and ulInputIOVectorTableSize

These specify the area for the input parameters passed to the server function. The input parameter area can be allocated in a cacheable area. The number of elements in the IOVEC array pointed to by pInputIOVectorTable is specified in ulInputIOVectorTableSize.

The number of input parameters (ulInputIOVectorTableSize) must always be fixed. IOVEC.ulSize must always be a fixed size except for the last IOVEC.ulSize. This is because these are used to obtain the address for storing the input parameters in each server stub. The last IOVEC.ulSize can be a variable size.

When there is no information to give, specify ulInputIOVectorTableSize as 0.

The IOVEC array contents are not updated by this API.

(4) pOutputIOVectorTable, ulOutputIOVectorTableSize, and pulLastOutputIOVectorSize

These specify the area for receiving the output from the server function. The number of elements in the IOVEC array pointed to by pOutputIOVectorTable is specified in ulOutputIOVectorTableSize.

When there is no information to receive, specify ulOutputIOVectorTableSize as 0.

The data output from the server is stored in the area specified by each IOVEC.

To areas indicated by IOVEC except for the last IOVEC, ulSize bytes of that IOVEC are output. To the area indicated by the last IOVEC, ulSize bytes of that IOVEC will not be output. The actually output size is returned to the area pointed to by pulLastOutputIOVectorSize.

The IOVEC array contents are not updated by this API.

(5) pulReturnValue

The return value of the server function is returned to *pulReturnValue.

(6) AckMode

Either one of the following can be specified.

- RPC_ACK (synchronous mode)

  The task that has called this API waits in this API until the server function returns.

- RPC_UNACK (asynchronous mode)

  After requesting execution of the server function, this API immediately returns without waiting for execution of the server function to finish. pOutputIOVectorTable, ulOutputIOVectorTableSize, and pulReturnValue are ignored when asynchronous mode is specified.

A server function in the current CPU can also be called by this API. However, since the same procedure for a call to another CPU is followed even for the current CPU, this API will take longer than a normal function call.

7.10.11 Call Server Function (Data Transfer Callback) (rpc_call_copycbk)

C-Language API:

```
INT32 rpc_call_copycbk(
          rpc_call_info *pCallInfo,
          void (*CopyCbk1)(void *, const void *, UINT32),
          void (*CopyCbk2)(void *, const void *, UINT32) );
```

Parameters:

pCallInfo	Pointer to the server function call information packet
CopyCbk1	Start address of the function to copy the client's input parameters to the server parameter area
CopyCbk2	Start address of the function to copy the parameters output from the server to the client's output parameter area

Packet Structure:

```
typedef    struct  {
              UINT32                ulMarshallingType;
              UINT32                ulServerID;
              UINT32                ulServerVersion;
              UINT32                ulServerProcedureID;
              IOVEC                 *pInputIOVectorTable;
              UINT32                ulInputIOVectorTableSize;
              IOVEC                 *pOutputIOVectorTable;
              UINT32                ulOutputIOVectorTableSize;
              UINT32                *pulLastOutputIOVectorSize;
              UINT32                *pulReturnValue;
              enum rpc_ack_mode     AckMode;
} rpc_call_info;
```

Return Values:

RPC_E_OK	Normal end
RPC_E_SYS	The OS state is invalid (failed in OAL_CanWait())
RPC_E_NOINIT	RPC is not initialized
RPC_E_GETTASKID	Failed in OAL_GetTaskID()
RPC_E_STATE	Server of ulServerID is not started
RPC_E_VER	Version does not match
RPC_E_PARM	(1) AckMode is other than RPC_ACK or RPC_UNACK (2) Necessary size for calling exceeded the acceptable size of the server (3) ulProcedureID ≥ rpc_server_info.ulNumFunctions
RPC_E_STOP	Server was stopped during call waiting

`RPC_E_NOMEM`	`Insufficient memory`
	`(1) Failed in allocation of call waiting management area`
	`(2) The server has failed in allocation of parameter area`

Function:

Calls the server function specified in ulServerProcedureID for the server specified in ulServerID. The server function is executed by the server CPU.

This API is the same as rpc_call() except for the following points.

Transferring the input parameters to the server parameter area (phase 3) and retrieving the output parameters from the server parameter area (phase 5) in rpc_call() are performed not by rpc_call() but by the callback functions specified in CopyCbk1 and CopyCbk2, respectively. If NULL is specified in CopyCbk1 or CopyCbk2, the relevant callback function will not be called.

7.10.12 Acquire Server Properties (rpc_get_server_properties)

C-Language API:

```
INT32 rpc_get_server_properties(
          UINT32  ulServerID,
          rpc_server_properties *pProp );
```

Parameters:

ulServerID	Server ID
pProp	Pointer to the server property information packet

Packet Structure:

```
typedef    struct  {
              UINT32              ulServerVersion;
              UINT32              ulMaxParamArea;
} rpc_server_properties;
```

Return Values:

RPC_E_SYS	The OS state is invalid (failed in OAL_CanWait())
RPC_E_NOINIT	RPC is not initialized
RPC_E_STATE	Server of ulServerID is not started

Function:

Acquires the information on the server specified by ulServerID and returns it to the area pointed to by pProp.

pProp -> ulServerVersion returns the server version. pProp -> ulMaxParamArea returns rpc_server_info.ulMaxParamAreaSize which was specified when the server was started.

7.11 Stubs

The server stub and client stub must be created by the server creator. Both of them must be prepared for each server function.

7.11.1 Server Stub

The server stub is called from the RPC server task. The server stub should be created according to the following specifications. Any function name can be used.

```
UINT32 stub_function(rpc_server_stub_info *pInfo );


typedef struct {
    UINT32              ulProcedureID;
    enum rpc_ack_mode   AckMode;
    UINT32              ulMarshallingType;
    UINT8               *pucParamArea;
    UINT32              ulMaxParamArea;
    UINT32              ulInParamSize;
    IOVEC               *pOutputIOVectorTable;
    UINT32              ulOutputIOVectorTableSize;
    void                *pUserDefinedData;
} rpc_server_stub_info;
```

(1) ulProcedureID

Receives rpc_call_info.ulServerProcedureID specified in rpc_call() or rpc_call_copycbk().

(2) AckMode

Receives rpc_call_info.AckMode (RPC_ACK or RPC_UNACK) specified in rpc_call() or rpc_call_copycbk().

(3) ulMarshallingType

Receives rpc_call_info.ulMarshallingType specified in rpc_call() or rpc_call_copycbk().

(4) pucParamArea and ulInParamSize

Receives information indicating the input parameters from the client.

The input parameters specified in rpc_call() or rpc_call_copycbk() are stored in an area that starts from pucParamArea and whose size is ulInParamSize bytes. pucParamArea is an address in the server parameter area and it must be aligned to the 4-byte boundary.

The size calculated from the formula below is set in ulInParamSize.

Σ (ALIGNUP4(rpc_call_info.pInputIOVectorTable[i]->ulSize)) .... (a)

The storage address for the 0th parameter becomes pucParamArea.

The storage address for the kth (k = 1 … rpc_call_info.ulInputIOVectorTableSize – 1) parameter is calculated by the formula below.

pucParamArea+ $\sum_{i=0}^{k-1}$(ALIGNUP4(rpc_call_info.pInputIOVectorTable[i]-> ulSize)).. (b)

The server stub and client stub have to be implemented according to the specification of "the number of parameters is always fixed and the size of the parameters except for the last parameter is also fixed" in order for the server stub to correctly calculate the storage address for each parameter using the above formula (b).

(5) pOutputIOVectorTable and ulOutputIOVectorTableSize

Receives information indicating the area where to store the parameters to be output to the client.

pOutputIOVectorTable is the start address of the output IOVEC array.
ulOutputIOVectorTableSize indicates the number of elements in that array.
rpc_call_info.ulInputIOVectorTableSize specified by the client in rpc_call() or rpc_call_copycbk() will be set in ulOutputIOVectorTableSize.

Each output IOVEC has the following settings. This means that the default output area is set in advance.

- pBaseAddress: Address in the server parameter area (does not overlap with the input parameter area).
- ulSize: Size specified by the client in rpc_call() or rpc_call_copycbk(). In other words, the size of data the client can receive.

Note that when the client does not request output or when the call was made in asynchronous mode, ulOutputIOVectorTableSize and pOutputIOVectorTable are specified as 0.

The server stub sets the output data to the area specified by each output IOVEC and returns. At this point, the output IOVEC should be handled as follows:

- pBaseAddress: Normally do not change this setting. In a case where overlapping of the input parameter area and output parameter area is desired, specify pBaseAddress to satisfy the condition of "area indicated by IOVEC is within the server parameter area".
- ulSize: Change ulSize for only the last IOVEC. ulSize for the last IOVEC should be changed to the actual size.

The RPC copies the data indicated by the IOVEC array, at the point on returning from the server stub, to the output parameter area (area pointed to by the IOVEC array of rpc_call_info.pOutputIOVectorTable) specified by the client. ulSize of the last IOVEC is returned to the area pointed to by rpc_call_info.pulLastOutputIOVectorSize for the client.

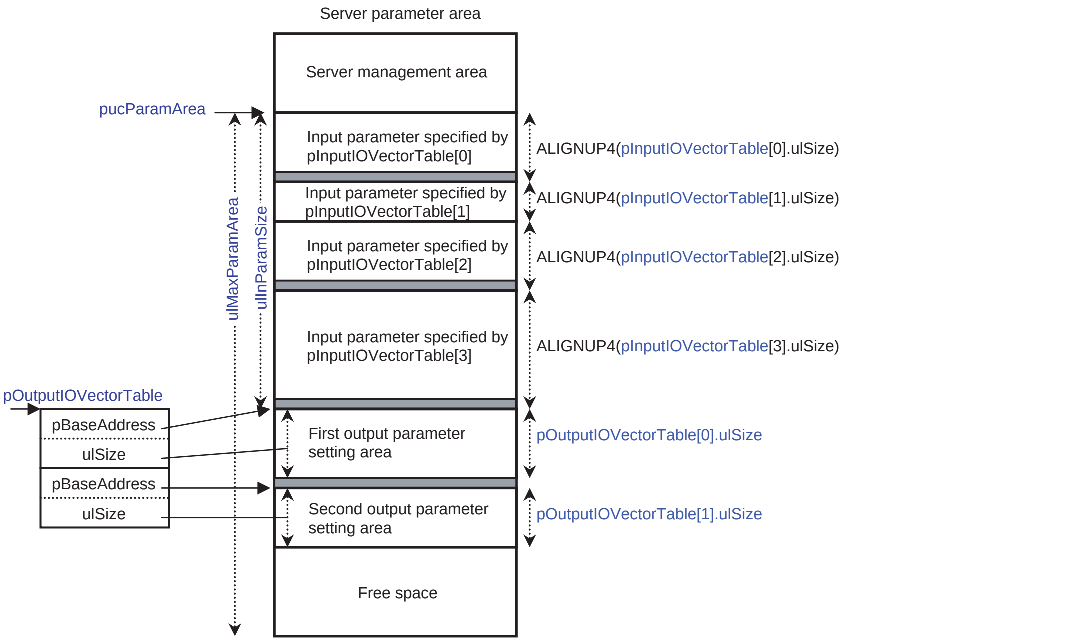
(6) ulMaxParamArea

The size the server stub can use is passed from pucParamArea to ulMaxParamArea. The output parameter storage area indicated by each IOVEC of pOutputIOVectorTable is included in this size. Note that the ulMaxParamArea value sent to the server stub differs from rpc_server_info.ulMaxParamAreaSize which is specified when the server is started.

(7) pUserDefinedData

Receives user definition information specified in rpc_start_server() or rpc_start_server_with_paramarea().

Figure 7.3 shows an example of the server parameter area when there are four input parameters and two output parameters.


Server parameter area
Server management area
pucParamArea
Input parameter specified by pInputIOVectorTable[0]
Input parameter specified by pInputIOVectorTable[1]
Input parameter specified by pInputIOVectorTable[2]
Input parameter specified by pInputIOVectorTable[3]
ALIGNUP4(pInputIOVectorTable[0].ulSize)
ALIGNUP4(pInputIOVectorTable[1].ulSize)
ALIGNUP4(pInputIOVectorTable[2].ulSize)
ALIGNUP4(pInputIOVectorTable[3].ulSize)
ulMaxParamArea
ulInParamSize
pOutputIOVectorTable
pBaseAddress
ulSize
pBaseAddress
ulSize
First output parameter setting area
Second output parameter setting area
pOutputIOVectorTable[0].ulSize
pOutputIOVectorTable[1].ulSize
Free space

The gray areas indicate areas which are not used due to boundary adjustment.
The red letters represent information in rpc_server_stub_info that is passed to the server stubs.
The blue letters represent information in rpc_call_info that is specified by an RPC call on the client side.

Figure 7.3 Server Parameter Area Example

7.11.2 Client Stub

The client stub has the same function names as the original server functions from the application, and it should be implemented to issue an RPC call using rpc_call() or rpc_call_copycbk().

7.12 Server Stop Callback Function

This callback function is called from the server task. This callback function should be created according to the following specification. Any function name can be used.

```
void StopServer(UINT32 ulParam);
```

ulParam specified in rpc_stop_server() is passed to ulParam.

7.13 CopyCbk1 and CopyCbk2 Callback Functions

These callback functions are called from rpc_call_copycbk(). These callback functions should be created according to the following specification. Any function name can be used.

```
void CopyFunc(void *pDest, const void *pSource, UINT32 ulSize);
```

ulSize bytes from the address specified in pSource are copied to the address pointed to by pDest.

When performing copy with a function other than the CPU, such as the DMAC, and operand cache is enabled, note the following.

(1) CopyCbk1

pDest points to the server parameter area (non-cacheable area).

pSource points to the input parameter area specified by the client. When operand cache is in write-back mode, if the contents indicated by pSource are registered in operand cache and have not been written back to the actual memory, the contents in the area pointed to by pSource have to be written back from operand cache to the actual memory before DMA transfer. Otherwise, the DMAC may read from pSource the data before write-back.

(2) CopyCbk2

pSource points to the area for storing the parameters output from the server (non-cacheable area).

pDest points to the output parameter area specified by the client. If the area indicated by pDest is registered in operand cache, the operand cache contents of the area pointed to by pDest have to be invalidated before DMA transfer. Otherwise, when the client reads from the output parameter area after DMA transfer, the read becomes an operand cache hit and the client reads the operand cache contents instead of the data transferred to the actual memory by the DMAC. When operand cache is in write-back mode, if the contents indicated by pDest are registered in operand cache and have not been written back to the actual memory, when a write-back due to operand cache replacement occurs after DMA transfer, the data transferred to the actual memory via DMA transfer may be damaged.

Since the operand cache contents of a specific address range cannot be invalidated in the SH2A-DUAL, either the entire operand cache has to be invalidated or the output parameter area specified by the client has to be restricted to a non-cacheable area.

Section 8 OAL

8.1 Overview

The OAL localizes the OS dependent part of the RPC so that the RPC can be easily ported to another OS. This improves the portability of the RPC.

Only the minimum OS functions required by the RPC are implemented in this OAL.

Table 8.1 Overview of OAL

No.	Item	Component
1	Hardware resources used by this software	None
2	Software components used by this software	None
3	Other software components using this software	RPC

8.2 Provided Files

<RTOS_INST>\os\include\

oal.h	API definition header file

<SAMPLE_INST>\R0K572650D000BR\cpuid1\ipi\

oal_config.h	Configuration file (see section 8.3.1, Configuration)
oal.c	Source code

<SAMPLE_INST>\R0K572650D000BR\cpuid2\ipi\

oal_config.h	Configuration file (see section 8.3.1, Configuration)
oal.c	Source code

Note that oal.c has the same contents in each directory.

8.3 Configuration and Build

The OAL must be separately configured for each CPU.

8.3.1 Configuration

(1) Configuration of OAL

Define the following in oal_config.h.

```
#define OAL_MEMSIZE 0x1000
```

The size of the memory area handled by OAL_GetMemory() should be defined in OAL_MEMSIZE.

(2) Configuration of Kernel

Refer to section 7.9.1 (3), service_call.

8.3.2 Build

Compile oal.c and link it with programs using API functions.

For the OAL sections, refer to section 17.5.2, Sections.

8.4 API Functions

Table 8.2 lists the API functions.

Table 8.2 API Functions

No.	Type	API Name	Function
1	Function	OAL_Init	Initializes OAL
2	Function	OAL_Shutdown	Terminates OAL
3	Function	OAL_DisablePreempt	Disables task preemption
4	Function	OAL_EnablePreempt	Enables task preemption
5	Function	OAL_IsDisablePreempt	Confirms task preemption state
6	Function	OAL_CanWait	Confirms whether current task can wait
7	Function	OAL_IsNotTaskLevel	Confirms context type
8	Function	OAL_IsMaskInterrupt	Confirms processor interrupt mask
9	Function	OAL_CreateTask	Creates task
10	Function	OAL_ActivateTask	Activates task
11	Function	OAL_DestroyTask	Exits and deletes current task
12	Function	OAL_GetTaskID	Acquires identification information on current task
13	Function	OAL_SleepTask	Shifts current task to WAITING state
14	Function	OAL_WakeupTask	Wakes up task
15	Function	OAL_GetMemory	Allocates memory
16	Function	OAL_ReleaseMemory	Releases memory

8.4.1 Header File

Include oal.h.

8.4.2 Basic Data Types

The basic data types defined in types.h are used.

For types.h, refer to section 19, types.h.

8.4.3 Return Value

For APIs that have return values, basically a positive value or 0 indicates normal end, and a negative value indicates an error.

The following values are defined as error return values. However, what kind of error is returned under what kind of condition depends on the OAL implementation and OS. Therefore, in an application using the OAL, determining the error return value for a purpose other than debugging is not recommended. For example, a condition that returns the OAL_E_PAR error in a certain OS may return another error in a different OS.

Due to the above reason, the error return values are not described in the subsequent sections on APIs.

`#define OAL_E_OK`	`0L`	Normal end
`#define OAL_E_PAR`	`(-1L)`	Parameter error
`#define OAL_E_SYS`	`(-4L)`	System state error
`#define OAL_E_STATE`	`(-5L)`	OS object state error
`#define OAL_E_NOMEM`	`(-7L)`	Insufficient memory
`#define OAL_E_NORESOURCE`	`(-8L)`	Insufficient resource
`#define OAL_E_TIMEOUT`	`(-16L)`	Timeout
`#define OAL_E_RELEASED`	`(-17L)`	WAITING state is forcibly canceled

8.4.4 Initialize OAL (OAL_Init)

C-Language API:

```
INT32 OAL_Init(void);
```

Function:

Initializes the OAL and starts it.

Operation when this API function is called in a context state other than the task level is undefined.

8.4.5 Terminate OAL (OAL_Shutdown)

C-Language API:

```
void OAL_Shutdown(void);
```

Function:

Terminates the OAL.

Operation when this API function is called from a state in which OAL_DisablePreempt() was called is undefined.

Operation when this API function is called in a context state other than the task level is undefined.

8.4.6 Disable Task Preemption (OAL_DisablePreempt)

C-Language API:

```
void OAL_DisablePreempt(void);
```

Function:

Disables task preemption.

This API function must not be called from a state in which OAL_DisablePreempt() was called.

Operation when this API function is called in a context state other than the task level is undefined.

8.4.7 Enable Task Preemption (OAL_EnablePreempt)

C-Language API:

```
void OAL_EnablePreempt(void);
```

Function:

Enables task preemption.

This API function can even be called from a state in which OAL_DisablePreempt() was called.

Operation when this API function is called in a context state other than the task level is undefined.

8.4.8 Confirm Task Preemption State (OAL_IsDisablePreempt)

C-Language API:

```
INT32 OAL_IsDisablePreempt(void);
```

Function:

When task preemption is disabled, 1 is returned. When task preemption is enabled, 0 is returned.

No error value will be returned.

Operation when this API function is called in a context state other than the task level is undefined.

8.4.9 Confirm Whether Current Task Can Wait (OAL_CanWait)

C-Language API:

```
INT32 OAL_CanWait(void);
```

Function:

If the calling context can enter the WAITING state of the OS, 1 is returned. If transition is not possible, 0 is returned.

No error value will be returned.

8.4.10 Confirm Context Type (OAL_IsNotTaskLevel)

C-Language API:

```
INT32 OAL_IsNotTaskLevel(void);
```

Function:

If the calling context is at the task level, 0 is returned. Otherwise, 1 is returned.

No error value will be returned.

8.4.11 Confirm Processor Interrupt Mask (OAL_IsMaskInterrupt)

C-Language API:

```
INT32 OAL_IsMaskInterrupt (void);
```

Function:

If there is an interrupt masked by the processor interrupt mask, 1 is returned. If no interrupts are masked, 0 is returned.

No error value will be returned.

8.4.12 Create Task (OAL_CreateTask)

C-Language API:

```
INT32 OAL_CreateTask(
          void                  **pTaskID
          void                  *pTaskStartAddress,
          void                  *pArg,
          UINT32                ulTaskPriority,
          UINT32                ulStackSize,
          enum OAL_TASK_START   AutoStart);
```

Parameters:

pTaskID	Pointer to the memory area where the identification information on the created task is to be returned
pTaskStartAddress	Task start address
pArg	Parameter to be passed to task
ulTaskPriority	Task priority
ulStackSize	Stack size
AutoStart	Task start specification

Function:

Creates a task and returns the identification information on the created task to *pTaskID. The task identification information is used for specifying the task in another API of OAL.

OAL_AUTO_START or OAL_NO_START can be specified for AutoStart.

When OAL_AUTO_START is specified, the specified task immediately enters the executable state on the OS. When OAL_NO_START is specified, the specified task is only created and not executed. To execute the task, the task has to be separately started by OAL_ActivateTask().

The parameter to be passed to the task is specified in pArg.

This API can even be called from a state in which OAL_DisablePreempt() was called.

Operation when this API function is called in a context state other than the task level is undefined.

8.4.13 Activate Task (OAL_ActivateTask)

C-Language API:

```
INT32 OAL_ActivareTask(void *TaskID);
```

Parameters:

```
TaskID                   Task identification information
```

Function:

Activates the task.

This API can even be called from a state in which OAL_DisablePreempt() was called.

Operation when this API function is called in a context state other than the task level is undefined.

8.4.14 Exit and Delete Current Task (OAL_DestroyTask)

C-Language API:

```
INT32 OAL_DestroyTask(void);
```

Function:

Exits the current task and deletes it.

Operation when this API function is called from a state in which OAL_DisablePreempt() was called is undefined.

Operation when this API function is called in a context state other than the task level is undefined.

8.4.15 Get Current Task Identification Information (OAL_GetTaskID)

C-Language API:

```
INT32 OAL_GetTaskID(void **pTaskID);
```

Parameters:

```
pTaskID              Pointer to the memory area where the identification
                     information on the task is to be returned
```

Function:

Returns the identification information on the current task to *pTaskID.

Operation when this API function is called from a state in which OAL_DisablePreempt() was called is undefined.

Operation when this API function is called in a context state other than the task level is undefined.

8.4.16 Shift Current Task to WAITING State (OAL_SleepTask)

C-Language API:

```
INT32 OAL_SleepTask(void);
```

Function:

Shifts the current task to the WAITING state. The WAITING state is canceled by OAL_WakeupTask().

Correct operation is not guaranteed when this API function is called from a state in which the task cannot be shifted to the WAITING state of the OS.

Operation when this API function is called from a state in which OAL_DisablePreempt() was called is undefined.

Operation when this API function is called in a context state other than the task level is undefined.

8.4.17 Wakeup Task (OAL_WakeupTask)

C-Language API:

```
INT32 OAL_WakeupTask(void *TaskID);
```

Parameters:

TaskID	Task identification information

Function:

Cancels the WAITING state of the task.

This API can even be called from a state in which OAL_DisablePreempt() was called.

This API can even be called in a context state other than the task level.

8.4.18 Allocate Memory (OAL_GetMemory)

C-Language API:

```
INT32 OAL_GetMemory(
          UINT32   ulSize,
          void     **ppAddress);
```

Parameters:

ulSize	Allocation size
ppAddress	Pointer to the memory area where the allocated memory address is to be returned

Function:

Allocates memory for the size specified by ulSize and returns the start address to the area indicated by ppAddress.

When memory cannot be allocated, an error is returned immediately without any wait.

The memory start address to be returned is aligned at the 4-byte boundary.

This API can even be called from a state in which OAL_DisablePreempt() was called.

Operation when this API function is called in a context state other than the task level is undefined.

8.4.19 Release Memory (OAL_ReleaseMemory)

C-Language API:

```
INT32 OAL_ReleaseMemory(void *pAddress);
```

Parameters:

pAddress	Allocated memory address

Function:

Releases memory whose start address is pAddress. pAddress has to be the start address of the memory allocated by OAL_GetMemory().

This API can even be called from a state in which OAL_DisablePreempt() was called.

Operation when this API function is called in a context state other than the task level is undefined.

Section 9 Spinlock Library

9.1 Overview

Usage of the spinlock library permits only one CPU to access the resources shared by the CPUs at one time.

This can also be achieved by using semaphores in the kernel. The differences from using semaphores in the kernel are shown below.

(1) Smaller overhead than when using semaphores in the kernel

(2) Can be used even when an interrupt handler is in the dispatch-pended state (semaphores in the kernel cannot be used in the dispatch-pended state)

However, there are notes for using the spinlock library. Be sure to read section 9.3, Spinlock Behavior and Usage Notes.

Table 9.1 Overview of Spinlock Library

No.	Item	Component
1	Hardware resources used by this software	None (However, the semaphore register specified by a parameter is accessed during semaphore lock)
2	Software components used by this software	None
3	Other software components using this software	(1) Kernel (remote service call, etc.) (2) RPC library (3) IPI

Use of the spinlock library should be avoided as much as possible in the application. Repeated use of the spinlock library indicates that the degree of linkage between the CPUs is high in the application so that distributing the features to each CPU may be troublesome. In addition, the possibility of the kind of failures described in section 9.3 (Spinlock Behavior and Usage Notes) occurring is increased.

9.2 Basic Usage Method

In the spinlock library, the resources shared by the CPUs are exclusively controlled using the "lock variables" in memory or the "semaphore registers" in the SH2A-DUAL. The relationship between the lock variables or semaphore registers and the resources shared by the CPUs is determined by the application.

In an application that desires to access the resources shared by the CPUs, call an API to perform lock before accessing the shared resources. Then, after access to the shared resources has finished, call an API to perform unlock.

9.3 Spinlock Behavior and Usage Notes

In an API to perform lock, whether lock has already been performed is checked. If lock has not been performed, the state is immediately updated to locked and returned. If lock has already been performed, busy-wait is performed until lock is canceled.

In other words, when attempting to obtain a lock variable that is already locked, the CPU time continues to be futilely consumed until the program that had performed lock calls the API to perform unlock.

This busy-wait operation may cause the following problems. Fully understand this section and make sure no such problems occur.

9.3.1 Exclusive Control in the Same CPU and Deadlock

The spinlock function supports exclusive control for programs that are processed by multiple CPUs.

If two or more programs in the same CPU may access the resources shared by the CPUs simultaneously, exclusive control in the same CPU is necessary whether the spinlock function will be used or not. If the spinlock function is used without performing this, a deadlock may occur.

- Example 1

  If task A and task B in the same CPU may simultaneously access a certain resource shared by the CPUs, task A and task B must be exclusively controlled by the semaphore function of the kernel or by the function to disable task dispatch.

  If this is not performed, a deadlock occurs in the following case.

  Assume that while task A has performed lock, task B with a higher priority preempts task A.

  When task B attempts to perform lock, the attempt fails and busy-wait is performed. However, since the priority of task A which has performed lock is lower than the priority of task B, task A will never be executed. Accordingly, task B keeps waiting to perform lock that will not be canceled, and results into a deadlock.
- Example 2

  If task A and an interrupt handler in the same CPU may simultaneously access a certain resource shared by the CPUs, task A and the interrupt handler must be exclusively controlled by disabling the interrupt or locking the CPU.

  If this is not performed, a deadlock occurs in the following case.

  Assume that while task A has performed lock, an interrupt handler is initiated.

  When the interrupt handler attempts to perform lock, the attempt fails and busy-wait is performed. However, task A which has performed lock will never be executed unless the interrupt handler is finished. Accordingly, the interrupt handler keeps waiting to perform lock that will not be canceled, and results into a deadlock.

9.3.2 Problem of Locked Period

Busy-wait performed by spinlock just wastes the CPU time.

To reduce this waste, the period for performing lock should be as short as possible. This will shorten the busy-wait period that may be generated when another program attempts to perform lock.

The user must be careful not to unintentionally prolong the locked period. Examples of such cases are shown below.

- Example 1

  A certain task was coded to access the resources shared by the CPUs after performing lock, and then immediately cancel lock. However, at task execution, the task was preempted by another task before canceling lock. Another CPU attempted to perform lock during this period, and busy-wait was performed for a long time.

  Such a case can be improved by disabling task dispatch before performing lock.

- Example 2

  A certain task was coded so that after dispatch-disabled state was entered, the task accesses the resources shared by the CPUs after performing lock, and then immediately cancels lock. However, at task execution, an interrupt occurred before canceling lock so that unlock was delayed until the interrupt handler was finished. Another CPU attempted to perform lock during this period, and busy-wait was performed for a long time.

  Such a case can be improved by disabling interrupts before performing lock.

9.4 Three Spinlock Functions

This spinlock library provides the following three types of spinlock functions.

(1) Normal Lock

A basic lock function that performs exclusive control between the CPUs by applying the TST instruction to the lock variable located in the memory shared by the CPUs.

(2) RW Lock

Similar to normal lock, this function performs exclusive control between the CPUs by applying the TST instruction to the lock variable located in the memory shared by the CPUs. However, this function is more efficient than normal lock because exclusive control of reference (read) accesses is omitted. Compared to normal lock, RW lock is suitable for exclusive control of resources that are often only referenced (read).

(3) Semaphore Lock

The semaphore lock function performs exclusive control between the CPUs by using the semaphore registers in the SH2A-DUAL. Note that these semaphore registers have nothing to do with the semaphores in the kernel.

In normal lock or RW lock, access to a lock variable during busy-wait for performing lock occupies the bus to the memory where the lock variable is stored. This sometimes degrades the access performance to that bus from another CPU.

The benefit of semaphore lock is that such kind of down side is small because a semaphore register is accessed through a different bus than the memory bus.

The IPI uses the semaphore lock function.

9.5 Lock Variables for Normal Lock and RW Lock

9.5.1 Entity of Lock Variable

The entities of lock variables should be defined in CPUID#1. At this time, a dedicated section different from the others should be used. When performing linkage in CPUID#1, the symbol address file (fsy extension) of that section is output, and assembling and linking that symbol address file in CPUID#2 enables the CPUID#2 program to perform symbol resolution for the lock variables in CPUID#1.

9.5.2 RAM where Lock Variables are Placed

Lock variables must be placed in memory connected via the same memory bus to each CPU that accesses those lock variables. Accesses must be non-cacheable.

Examples for SH7205 or SH7265 are shown below.

(1) When Placing Lock Variables in On-Chip RAM

Accesses to on-chip RAM are not cached.

Place the lock variables at an address in the shadow area (0xFFD80000 to 0xFFDA7FFF) of on-chip RAM at linkage. Both CPUs use the same high-speed on-chip RAM access bus to access this address.

(2) When Placing Lock Variables in External RAM

When placing lock variables in SDRAM connected to the SDRAM0 space (0x18000000 to 0x1BFFFFFF), place the lock variables at an address in the non-cacheable shadow area (0x38000000 to 0x3BFFFFFF) of the SDRAM0 space.

9.6 Provided Files

<RTOS_INST>\os\include\	
spinlock.h	API definition header file
<RTOS_INST>\os\lib\debug\	
spinlock.lib	Library (with debugging information)
<RTOS_INST>\os\lib\release\	
spinlock.lib	Library (without debugging information)
<RTOS_INST>\os\spinlock\	Workspace, etc. for creating the library
<RTOS_INST>\os\spinlock\spinlock\	Project, etc. for creating the library
<RTOS_INST>\os\spinlock\spinlock\include\	Internal definition file (assembly language)
<RTOS_INST>\os\spinlock\spinlock\source\	Source code
<RTOS_INST>\os\spinlock\spinlock\debug\	Configuration directory (with debugging information)
<RTOS_INST>\os\spinlock\spinlock\release\	Configuration directory (without debugging information)

9.7 Building the Library

Building the library is not usually necessary. If you wish to build the library (e.g. for debugging), you should use the provided High-performance Embedded Workshop workspace (spinlock.hws).

9.8 Building the System

For the spinlock library sections, refer to section 17.5.2, Sections.

9.9 API Functions

Table 9.2 lists the API functions. Each API function is implemented as a C function macro or C function.

Table 9.2 API Functions

No.	Classification	Function Name	Function
1	Normal lock	SPIN_InitLock	Initializes normal lock variable
2		SPIN_Lock	Performs normal lock
3		SPIN_TryLock	Tries to perform normal lock
4		SPIN_Unlock	Cancels normal lock
5		SPIN_IsLocked	Checks normal lock state
6	RW lock	SPIN_InitRWLock	Initializes RW lock variable
7		SPIN_ReadLock	Performs read lock
8		SPIN_ReadTryLock	Tries to perform read lock
9		SPIN_ReadUnlock	Cancels read lock
10		SPIN_IsReadLocked	Checks read lock state
11		SPIN_WriteLock	Performs write lock
12		SPIN_WriteTryLock	Tries to perform write lock
13		SPIN_WriteUnlock	Cancels write lock
14		SPIN_IsWriteLocked	Checks write lock state
15	Semaphore lock	SPIN_InitSemLock	Initializes semaphore register
16		SPIN_SemLock	Performs semaphore lock
17		SPIN_SemTryLock	Tries to perform semaphore lock
18		SPIN_SemUnlock	Cancels semaphore lock

9.9.1 Header File

Include spinlock.h.

9.9.2 Basic Data Types

The basic data types defined in types.h are used.

For types.h, refer to section 19, types.h.

9.9.3 Note

Error detection is not performed in these API functions.

9.10 Normal Lock

9.10.1 Initialize Normal Lock Variable (SPIN_InitLock)

C-Language API:

```
void SPIN_InitLock(LOCK *pLock);
```

Parameters:

```
          pLock     Pointer to lock variable
```

Packet Structure:

```
typedef   struct    {
          UINT8     ucLock;        Lock variable
} LOCK;
```

Function:

Initializes the lock variable indicated by pLock.

The lock variable is managed by the spinlock library and cannot be directly changed from the application.

pLock must be an address to which access is non-cacheable.

9.10.2 Perform Normal Lock (SPIN_Lock)

C-Language API:

```
void SPIN_Lock(LOCK *pLock);
```

Parameters:

```
          pLock      Pointer to lock variable
```

Packet Structure:

```
typedef   struct     {
          UINT8      ucLock;        Lock variable
} LOCK;
```

Function:

Locks the access right for the resource associated with pLock.

pLock must be a pointer to the lock variable that has already been initialized in SPIN_InitLock().

If the access right is already locked, the busy loop is executed in this function to wait for lock to be canceled. After lock cancellation, the access right is locked.

Lock performed by this function is canceled by SPIN_Unlock().

9.10.3 Try to Perform Normal Lock (SPIN_TryLock)

C-Language API:

```
INT32 SPIN_TryLock(LOCK *pLock);
```

Parameters:

```
            pLock     Pointer to lock variable
```

Return Values:

```
1: Lock succeeded
0: Lock failed
```

Packet Structure:

```
typedef   struct    {
          UINT8     ucLock;        Lock variable
} LOCK;
```

Function:

Locks the access right for the resource associated with pLock.

pLock must be a pointer to the lock variable that has already been initialized in SPIN_InitLock().

If the access right is already locked, lock fails and 0 is returned as the return value. If the access right is not locked, lock succeeds and 1 is returned as the return value.

Lock performed by this function is canceled by SPIN_Unlock().

9.10.4 Cancel Normal Lock (SPIN_Unlock)

C-Language API:

```
void SPIN_Unlock(LOCK *pLock);
```

Parameters:

```
          pLock     Pointer to lock variable
```

Packet Structure:

```
typedef   struct    {
          UINT8     ucLock;        Lock variable
} LOCK;
```

Function:

Unlocks the access right for the resource associated with pLock.

pLock must be a pointer to the lock variable that was acquired in SPIN_Lock() or SPIN_TryLock().

9.10.5 Check Normal Lock State (SPIN_IsLocked)

C-Language API:

```
INT32 SPIN_IsLocked(LOCK *pLock);
```

Parameters:

```
          pLock     Pointer to lock variable
```

Packet Structure:

```
typedef   struct    {
          UINT8     ucLock;        Lock variable
} LOCK;
```

Return Values:

```
1: Locked
0: Not locked
```

Function:

1 is returned if pLock is locked and 0 is returned if pLock is not locked.

9.11 RW Lock

9.11.1 Initialize RW Lock Variable (SPIN_InitRWLock)

C-Language API:

```
void SPIN_InitRWLock(RWLOCK *pRWLock);
```

Parameters:

```
          pRWLock    Pointer to lock variable
```

Packet Structure:

```
typedef   struct    {
          UINT8     ucWriteLock;   Write lock variable
          UINT8     ucReadLock;    Read lock variable
} RWLOCK;
```

Function:

Initializes the RW lock variable indicated by pRWLock.

The RW lock variable is managed by the spinlock library and must not be directly changed from the application.

pRWLock must be an address to which access is non-cacheable.

9.11.2 Perform Read Lock (SPIN_ReadLock)

C-Language API:

```
void SPIN_ReadLock(RWLOCK *pRWLock);
```

Parameters:

```
          pRWLock    Pointer to lock variable
```

Packet Structure:

```
typedef   struct    {
          UINT8     ucWriteLock;   Write lock variable
          UINT8     ucReadLock;    Read lock variable
} RWLOCK;
```

Function:

Locks the read access right for the resource associated with pRWLock.

pRWLock must be a pointer to the lock variable that has already been initialized in SPIN_InitRWLock().

Read lock can be nested. The maximum number of nestings is 255.

If write lock has already been performed, the busy loop is executed in this function to wait for write lock to be canceled. After lock cancellation, read lock is performed. If write lock has not been performed, even though read lock has been performed, read lock succeeds immediately.

Lock performed by this function is canceled by SPIN_ReadUnlock().

9.11.3 Try to Perform Read Lock (SPIN_ReadTryLock)

C-Language API:

```
INT32 SPIN_ReadTryLock(RWLOCK *pRWLock);
```

Parameters:

```
            pRWLock    Pointer to lock variable
```

Return Values:

```
1: Read lock succeeded
0: Read lock failed
```

Packet Structure:

```
typedef   struct     {
          UINT8      ucWriteLock;   Write lock variable
          UINT8      ucReadLock;    Read lock variable
} RWLOCK;
```

Function:

Locks the read access right for the resource associated with pRWLock.

pRWLock must be a pointer to the lock variable that has already been initialized in SPIN_InitRWLock().

Read lock can be nested. The maximum number of nestings is 255.

If write lock has already been performed, read lock fails and 0 is returned as the return value. If write lock has not been performed, even though read lock has been performed, read lock succeeds immediately and 1 is returned as the return value.

Lock performed by this function is canceled by SPIN_ReadUnlock().

9.11.4 Cancel Read Lock (SPIN_ReadUnlock)

C-Language API:

```
void SPIN_ReadUnlock(RWLOCK *pRWLock);
```

Parameters:

```
          pRWLock    Pointer to lock variable
```

Packet Structure:

```
typedef   struct     {
          UINT8      ucWriteLock;   Write lock variable
          UINT8      ucReadLock;    Read lock variable
} RWLOCK;
```

Function:

Unlocks the read access right for the resource associated with pRWLock.

pRWLock must be a pointer to the lock variable that was acquired in SPIN_ReadLock() or SPIN_ReadTryLock().

9.11.5 Check Read Lock State (SPIN_IsReadLocked)

C-Language API:

```
INT32 SPIN_IsReadLocked(RWLOCK *pRWLock);
```

Parameters:

```
          pRWLock    Pointer to lock variable
```

Packet Structure:

```
typedef   struct     {
          UINT8      ucWriteLock;   Write lock variable
          UINT8      ucReadLock;    Read lock variable
} RWLOCK;
```

Return Values:

```
1: Locked
0: Not locked
```

Function:

1 is returned if pRWLock is read-locked and 0 is returned if pRWLock is not read-locked.

9.11.6 Perform Write Lock (SPIN_WriteLock)

C-Language API:

```
void SPIN_WriteLock(RWLOCK *pRWLock);
```

Parameters:

```
          pRWLock   Pointer to lock variable
```

Packet Structure:

```
typedef   struct    {
          UINT8     ucWriteLock;   Write lock variable
          UINT8     ucReadLock;    Read lock variable
} RWLOCK;
```

Function:

Locks the write access right for the resource associated with pRWLock.

pRWLock must be a pointer to the lock variable that has already been initialized in SPIN_InitRWLock().

If read lock or write lock has already been performed, the busy loop is executed in this function to wait for those locks to be canceled. After lock cancellation, write lock is performed.

Lock performed by this function is canceled by SPIN_WriteUnlock().

9.11.7 Try to Perform Write Lock (SPIN_WriteTryLock)

C-Language API:

```
INT32 SPIN_WriteTryLock(RWLOCK *pRWLock);
```

Parameters:

```
          pRWLock    Pointer to lock variable
```

Return Values:

```
1: Write lock succeeded
0: Write lock failed
```

Packet Structure:

```
typedef   struct    {
          UINT8     ucWriteLock;   Write lock variable
          UINT8     ucReadLock;    Read lock variable
} RWLOCK;
```

Function:

Locks the write access right for the resource associated with pRWLock.

pRWLock must be a pointer to the lock variable that has already been initialized in SPIN_InitRWLock().

If read lock or write lock has already been performed, write lock fails and 0 is returned as the return value. If neither read lock nor write lock has been performed, write lock succeeds and 1 is returned as the return value.

Lock performed by this function is canceled by SPIN_WriteUnlock().

9.11.8 Cancel Write Lock (SPIN_WriteUnlock)

C-Language API:

```
void SPIN_WriteUnlock(RWLOCK *pRWLock);
```

Parameters:

```
            pRWLock   Pointer to lock variable
```

Packet Structure:

```
typedef   struct    {
          UINT8     ucWriteLock;   Write lock variable
          UINT8     ucReadLock;    Read lock variable
} RWLOCK;
```

Function:

Unlocks the write access right for the resource associated with pRWLock.

pRWLock must be a pointer to the lock variable that was acquired in SPIN_WriteLock() or SPIN_WriteTryLock().

9.11.9 Check Write Lock State (SPIN_IsWriteLocked)

C-Language API:

```
INT32 SPIN_IsWriteLocked(RWLOCK *pRWLock);
```

Parameters:

```
            pRWLock   Pointer to lock variable
```

Packet Structure:

```
typedef   struct    {
          UINT8     ucWriteLock;   Write lock variable
          UINT8     ucReadLock;    Read lock variable
} RWLOCK;
```

Return Values:

```
1: Locked
0: Not locked
```

Function:

1 is returned if pRWLock is write-locked and 0 is returned if pRWLock is not write-locked.

9.12 Semaphore Lock

9.12.1 Initialize Semaphore Register (SPIN_InitSemLock)

C-Language API:

```
void SPIN_InitSemLock(UINT8 *pucSemRegister);
```

Parameters:

```
            pucSemRegister     Semaphore register address
```

Function:

Initializes the semaphore register indicated by pucSemRegister.

The address of a semaphore register in the microcomputer used must be specified in pucSemRegister.

9.12.2 Perform Semaphore Lock (SPIN_SemLock)

C-Language API:

```
void SPIN_SemLock(UINT8 *pucSemRegister);
```

Parameters:

```
            pucSemRegister     Semaphore register address
```

Function:

Locks the access right for the resource associated with pucSemRegister.

The address of a semaphore register in the microcomputer used must be specified in pucSemRegister.

If the access right is already locked, the busy loop is executed in this function to wait for lock to be canceled. After lock cancellation, the access right is locked.

Lock performed by this function is canceled by SPIN_SemUnlock().

9.12.3 Try to Perform Semaphore Lock (SPIN_SemTryLock)

C-Language API:

```
INT32 SPIN_SemTryLock(UINT8 *pucSemRegister);
```

Parameters:

```
            pucSemRegister     Semaphore register address
```

Return Values:

```
1: Lock succeeded
0: Lock failed
```

Function:

Locks the access right for the resource associated with pucSemRegister.

The address of a semaphore register in the microcomputer used must be specified in pucSemRegister.

If the access right is already locked, lock fails and 0 is returned as the return value. If the access right is not locked, lock succeeds and 1 is returned as the return value.

Lock performed by this function is canceled by SPIN_SemUnlock().

9.12.4 Cancel Semaphore Lock (SPIN_SemUnlock)

C-Language API:

```
void SPIN_SemUnlock(UINT8 *pucSemRegister);
```

Parameters:

```
            pucSemRegister     Semaphore register address
```

Function:

Unlocks the access right for the resource associated with pucSemRegister.

The address of a semaphore register in the microcomputer used must be specified in pucSemRegister.

Section 10 IPI

10.1 Overview

The IPI is software providing a primitive function for communication between the processors.

The IPI can create "IPI ports" for receiving data from another CPU. Up to eight IPI ports can be created.

Table 10.1 Overview of IPI

No.	Item	Component
1	Hardware resources used by this software	(1) Inter-processor interrupt function (2) Semaphore registers (accessed within the spinlock library)
2	Software components used by this software	Spinlock library
3	Other software components using this software	(1) Kernel (remote service calls) (2) RPC library

10.2 IPI Structure

The IPI consists of API functions and inter-processor interrupt handler functions.

The API functions process the APIs of the IPI.

An inter-processor interrupt handler function is a processing function executed when an inter-processor interrupt occurs. The user must appropriately register these functions in the kernel as interrupt handlers. (IPI_init(), which initializes the IPI, does not define the interrupt handlers in the kernel.)

10.3 Port ID

Each CPU has port IDs that have values from 0 to 7. The port IDs and inter-processor interrupts have a one-to-one correspondence, as shown in table 10.2.

Table 10.2 Relationship between Port ID and Inter-Processor Interrupt

Port ID	Vector Number	Inter-Processor Interrupt Level
0	21	15
1	22	14
2	23	13
3	24	12
4	25	11
5	26	10
6	27	9
7	28	8

10.4 Overview of Operation

First, the port ID to be handled by the IPI must be defined at IPI configuration. In other words, the vector number of the inter-processor interrupt used by the IPI must be determined.

In an application that receives data through communication using the IPI, IPI ports should be created first using IPI_create(). At this time, the port ID to be created and the callback function that is executed at data reception should be registered.

In an application that transmits data to an IPI port of another CPU, IPI_send() is used. At this time, the target CPUID, port ID, and data to be transmitted should be specified. The transmit data size is (1 byte + 4 bytes).

IPI_send() issues an inter-processor interrupt to the CPU to which data is transmitted. In the CPU to which data is transmitted, this interrupt calls the callback function registered in IPI_create(). The transmitted data is passed to the callback function.

10.5 Notes

An API function may perform semaphore lock for an IPI port. If another program in the same CPU calls an API to perform semaphore lock for the same IPI port, there is a possibility that a deadlock occurs. In such a case, exclusive control within the same CPU should be performed by the program that calls the API function.

For details on a deadlock, refer to section 9.3.1, Exclusive Control in the Same CPU and Deadlock.

10.6 Provided Files

<RTOS_INST>\os\include\

ipi.h	API definition header file

<SAMPLE_INST>\R0K572650D000BR\cpuid1\ipi\

ipi_config.h	Configuration file (see section 10.7.1, Configuration)
ipi_defs.h	Internal definition file
ipi.c	Source code

<SAMPLE_INST>\R0K572650D000BR\cpuid2\ipi\

ipi_config.h	Configuration file (see section 10.7.1, Configuration)
ipi_defs.h	Internal definition file
ipi.c	Source code

Note that ipi_defs.h and ipi.c have the same contents in each directory.

10.7 Configuration and Build

The IPI must be separately configured for each CPU.

10.7.1 Configuration

Define the following in ipi_config.h.

(1) Definition of Ports to be Used

Defines whether to enable usage of the port for each ID.

When using the inter-processor interrupt for a purpose other than the IPI, set 0 as the definition for that port ID. If a port ID defined as 0 is specified in IPI_create(), an error is returned.

```
/*** Defines using ports ***/
#define PORT0    1    /* 1:use PORT0, 0:not use PORT0 */
#define PORT1    1    /* 1:use PORT1, 0:not use PORT1 */
#define PORT2    1    /* 1:use PORT2, 0:not use PORT2 */
#define PORT3    1    /* 1:use PORT3, 0:not use PORT3 */
#define PORT4    1    /* 1:use PORT4, 0:not use PORT4 */
#define PORT5    1    /* 1:use PORT5, 0:not use PORT5 */
#define PORT6    1    /* 1:use PORT6, 0:not use PORT6 */
#define PORT7    1    /* 1:use PORT7, 0:not use PORT7 */
```

Note the following when making the definition.

(a) When accepting a remote service call from another CPU

When a value other than 0 is specified for remote_svc.num_server (number of SVC servers) in the cfg file, make the definition so that the IPI port ID specified in remote_svc.ipi_portid is enabled.

Note that the interrupt level of remote_svc.ipi_portid must be equal to or lower than system.system_IPL. Otherwise, cfg72mp reports an error.

(b) When accepting RPC from another CPU

When a value other than 0 is specified for rpc_config.ulTableSize (number of RPC servers that can be registered) in rpc_init(), make the definition so that the IPI port ID specified in rpc_config.ulIPIPortID is enabled.

Note that the interrupt level of rpc_config.ulIPIPortID must be equal to or lower than system.system_IPL. Otherwise, correct operation is not guaranteed.

(2) Semaphore Register Address Used in Each Port

Each port uses a semaphore register for exclusive control between the processors. The address of the semaphore register used in each port is defined here. Note that the definition of the semaphore register address for a port that has been defined as 0 in "(1) Definition of Ports to be Used" will be ignored. The address of a single semaphore register must not be specified for two or more ports even when the ports belong to different CPUs.

```
/*** Defines using ports ***/
#define PORT0_SEMADR 0xFFFC1E00
#define PORT1_SEMADR 0xFFFC1E04
#define PORT2_SEMADR 0xFFFC1E08
#define PORT3_SEMADR 0xFFFC1E0C
#define PORT4_SEMADR 0xFFFC1E10
#define PORT5_SEMADR 0xFFFC1E14
#define PORT6_SEMADR 0xFFFC1E18
#define PORT7_SEMADR 0xFFFC1E1C
```

(3) Base Address of Inter-Processor Interrupt Control Register

Specify the address of C0IPCR15 in the interrupt controller with a constant expression, regardless of CPUID.

```
/*** Defines C0IPCR15 register address ***/
#define ADDR_C0IPCR15 0xFFFC1C00
```

(4) Stack Size Used by an Interrupt Handler

Define the stack size used by the inter-processor interrupt handler for each port. Note that the definition of the stack size for a port that has been defined as 0 in "(1) Definition of Ports to be Used" will be ignored.

```
/*** Defines stack size for interrupt handlers ***/
#define PORT0_STKSZ 0x400
#define PORT1_STKSZ 0x400
#define PORT2_STKSZ 0x400
#define PORT3_STKSZ 0x400
#define PORT4_STKSZ 0x400
#define PORT5_STKSZ 0x400
#define PORT6_STKSZ 0x400
#define PORT7_STKSZ 0x400
```

10.7.2 Build

Compile ipi.c and link it with programs using API functions (e.g. RPC).

Note that ipi.c includes mycpuid.h that is output from cfg72mp, and uses the macro MYCPUID which is defined in that file.

The interrupt handlers need to be defined in the kernel using methods, such as registering it in the cfg file.

For the IPI sections, refer to section 17.5.2, Sections.

10.8 API Functions

Table 10.3 lists the API functions.

Table 10.3 API Functions

No.	API Name	Function
1	IPI_init	Initializes IPI
2	IPI_create	Creates IPI port
3	IPI_delete	Deletes IPI port
4	IPI_send	Transmission to IPI port

10.8.1 Header File

Include ipi.h.

10.8.2 Basic Data Types

The basic data types defined in types.h are used.

For types.h, refer to section 19, types.h.

10.8.3 Initialize IPI (IPI_init)

C-Language API:

```
INT32 IPI_init(void);
```

Return Values:

```
IPI_E_OK            Normal end
IPI_E_NOINIT        IPI is not initialized (detected only when MYCPUID = 2)
```

Function:

Initializes the IPI.

To call this API function from CPUID#2, IPI_init() has to be already completed in CPUID#1.

This API should be called by each CPU only once in the beginning.

Current CPU's IPI Ports Locked during API Function Execution:

All ports defined as "used" in ipi_config.h

Note:

Though the interrupt handlers described later are usually implemented by dynamically defining them in the kernel in this initialization function, this is not performed in the state at shipment. This is because the interrupt handlers cannot be dynamically defined when the interrupt vector table is in ROM. The interrupt handlers described later should be defined as appropriate vector numbers (vector numbers for inter-processor interrupts) at kernel configuration.

10.8.4 Create IPI Port (IPI_create)

C-Language API:

```
INT32 IPI_create(
          UINT32   ulPortID,
          void     (*pCallBack)(UINT8, UINT32));
```

Parameters:

ulPortID	Target port ID
pCallBack	Callback function address

Return Values:

IPI_E_OK	Normal end
IPI_E_NOINIT	IPI is not initialized
IPI_E_PAR	ulPortID is 8 or higher
IPI_E_STATE	Port ID already created has been specified Port ID defined as "not used" in configuration file has been specified

Function:

Creates the IPI port of the port ID specified by ulPortID in the current CPU.

The address of the callback routine called when data is transmitted to the relevant port should be specified in pCallBack. For the callback function, refer to section 10.10, Callback Function.

Current CPU's IPI Ports Locked during API Function Execution:

Port specified by ulPortID

10.8.5 Delete IPI Port (IPI_delete)

C-Language API:

```
INT32 IPI_delete(
          UINT32   ulPortID);
```

Parameters:

ulPortID	Target port ID

Return Values:

IPI_E_OK	Normal end
IPI_E_NOINIT	IPI is not initialized
IPI_E_PAR	ulPortID is 8 or higher
IPI_E_STATE	Port ID not created has been specified Port ID in the middle of transmission has been specified Port ID defined as "not used" in configuration file has been specified

Function:

Deletes the IPI port of the port ID specified by ulPortID.

Current CPU's IPI Ports Locked during API Function Execution:

Port specified by ulPortID

10.8.6 Transmission to IPI Port (IPI_send)

C-Language API:

```
INT32 IPI_send(
          UINT32   ulCpuID,
          UINT32   ulPortID,
          UINT8    ucCode,
          UINT32   ulData );
```

Parameters:

ulCpuID	Target CPUID
ulPortID	Port ID
ucCode	Transmission code
ulData	Transmit data

Return Values:

IPI_E_OK	Normal end
IPI_E_NOINIT	IPI is not initialized
IPI_E_PAR	ulPortID is 8 or higher
IPI_E_STATE	Port ID not created has been specified Port ID defined as "not used" in configuration file has been specified

Function:

Transmits ucCode and ulData to the port specified by ulPortID in the CPU specified by ulCpuID.

ucCode and ulData are passed to the callback function of the port to which ucCode and ucData are transmitted. For the callback function, refer to section 10.10, Callback Function.

In this function, an inter-processor interrupt is requested to the target CPU. After that, busy-wait is performed in this function until the target CPU accepts the interrupt.

If the IPI supports interrupt requests to the current CPU, transmission can also be performed to the current CPU. However, in this case, this function must be called from a state in which the relevant inter-processor interrupt can be accepted. Otherwise, the CPU gets deadlocked.

Current CPU's IPI Ports Locked during API Function Execution:

Port specified by ulPortID when ulCpuID is the current CPU

10.9 Inter-Processor Interrupt Handlers

An inter-processor interrupt handler is contained in each port of each CPU.

The inter-processor interrupt handlers are shown in table 10.4.

Table 10.4 Inter-Processor Interrupt Handlers

Port ID	Vector Number	Inter-Processor Interrupt Level	Interrupt Handler Function Name
0	21	15	IPI_Port0Handler
1	22	14	IPI_Port1Handler
2	23	13	IPI_Port2Handler
3	24	12	IPI_Port3Handler
4	25	11	IPI_Port4Handler
5	26	10	IPI_Port5Handler
6	27	9	IPI_Port6Handler
7	28	8	IPI_Port7Handler

As described above, the interrupt handlers are not defined in the kernel by IPI_init() in the state at shipment. Therefore the user needs to define these handlers in the kernel.

The state at shipment is as follows:

- Interrupt handlers are implemented as the HI7200/MP direct interrupt handlers.
- Interrupt handlers are defined in the kernel by the sample cfg file.

10.10 Callback Function

In IPI_create(), the callback function that is called at data reception should be specified.

This callback function must be written in the following format. The function name can be set freely.

```
  void callback(UINT8 ucCode, UINT32 ulData)
```

The values specified in IPI_send() are passed to ucCode and ulData, respectively.

The callback function is called from each inter-processor interrupt handler of the IPI. Take this point in consideration when calculating the stack size used by the interrupt handler.

Section 11 SH2A-DUAL Cache-Support Library

11.1 Overview

The SH2A-DUAL cache-support library provides functions that allow maintenance of the local caches for each of the CPUs within the SH2A-DUAL. These functions can be used to maintain coherence between data in the cache and in the actual memory and so on.

Note, however, that the cache-support library does not provide any facilities for maintaining coherence between the caches for the two CPUs.

Table 11.1 Outline of the SH2A-DUAL Cache-Support Library

Item	Description
Hardware resource used	Cache memory within the SH2A-DUAL
Software components used by this software	None
Other software components using this software	None

11.2 Notes

(1) Incorrect use of the cache-support library may affect system operation; for example, coherence between the cache and the actual memory may not be maintainable. Before using the cache-support library, ensure that you fully understand the specifications of the caches in the target microcomputer and the behavior of the library.

(2) The cache-support library is just a set of functions. Tasks may be switched or interrupts may be accepted while these functions are being executed. Prevent the occurrence of such events as required before executing cache-support functions.

11.3 Directory and File Structure

<RTOS_INST>\os\include\	
sh2adual_cache.h	API definition header file
<RTOS_INST>\os\lib\debug\	
sh2adual_cache.lib	Library (with debugging information)
<RTOS_INST>\os\lib\release\	
sh2adual_cache.lib	Library (without debugging information)
<RTOS_INST>\os\sh2adual_cache\	Workspace, etc. for creating the library
<RTOS_INST>\os\sh2adual_cache\sh2adual_cache\	Project, etc. for creating the library
<RTOS_INST>\os\sh2adual_cache\sh2adual_cache\include\	Internal definitions
<RTOS_INST>\os\sh2adual_cache\sh2adual_cache\source\	Source code
<RTOS_INST>\os\sh2adual_cache\sh2adual_cache\Debug\	Configuration directory (with debugging information)
<RTOIS_INST>\os\sh2adual_cache\sh2adual_cache\Release\	Configuration directory (without debugging information)

11.4 Building the Library

Building the library is not usually necessary. If you wish to build the library (e.g. for debugging), you should use the provided High-performance Embedded Workshop workspace (sh2adual_cache.hws).

11.5 Building the System

Programs that use functions of this API must be linked to the cache-support library. For sections of the cache-support library, refer to section 17.5.2, Sections.

11.6 API Functions

Table 11.2 lists the API functions.

Table 11.2 Outline of the SH2A-DUAL Cache-Support Library

Classification	API Name	Description
Initialize	sh2adual_ini_cac	Initializes the cache
Clear	sh2adual_clr_cac	Clears the cache
Flush	sh2adual_fls_cac	Flushes the cache
Invalidate	sh2adual_inv_cac	Invalidates the cache

11.6.1 Header File

Include include\sh2adual_cache.h.

11.6.2 Basic Data Types

The basic data types defined in types.h are used.

For types.h, refer to section 19, types.h.

11.6.3 Initialize Cache (sh2adual_ini_cac)

C-Language API:

```
INT32 sh2adual_ini_cac(UINT32 ulCacAtr);
```

Parameters:

`ulCacAtr`	Cache-initialization attribute

Return Value:

`CAC_E_OK`	Normal end

Function:

This function initializes the caches. More specifically, the CCR1 register is updated to a value determined by ulCacAtr as described below. CCR1 is updated while SR.IMASK = 15.

Any of the following values or the logical OR of a combination of these values can be specified for ulCacAtr. This function does not check the value for errors.

This function writes 1 to the CCR1.ICF and CCR1.OCF bits regardless of the ulCacAtr setting; that is, any contents of the cache before this function call are cleared.

- TCAC_IC_ENABLE (H'00000100)
  Setting this value enables the instruction cache (CCR1.ICE = 1); otherwise, the instruction cache is disabled (CCR1.ICE = 0).
- TCAC_OC_ENABLE (H'00000001)
  Setting this value enables the operand cache (CCR1.OCE = 1); otherwise, the operand cache is disabled (CCR1.OCE = 0).
- TCAC_OC_WT (0x00000002)
  Setting this value selects the write-through mode as the write mode for the target-cache area (CCR1.WT = 1); otherwise, the write-back mode is selected (CCR1.WT = 0).

This API function does not manipulate the CCR2 register.

Do not call this API function when the cache memory is in the module-standby state.

11.6.4 Clear Cache (sh2adual_clr_cac)

C-Language API:

```
INT32 sh2adual_clr_cac(void *pStart, void *pEnd, UINT32 ulMode);
```

Parameters:

pStart	Address where cache clearing starts
pEnd	Address where cache clearing ends
ulMode	Target cache

Return Values:

CAC_E_OK	Normal end
CAC_E_PAR	Parameter error (pStart > pEnd or incorrect ulMode)

Function:

This function clears the caches. More specifically, the contents of the specified cache(s) are invalidated, and if the operand cache contains data that have not been written back into memory, the data are written into memory. Even when the cache-lock mode has been selected, all cache entries that are locked will be cleared. This function does not unlock the entries.

The target cache is specified by ulMode. Any one of the following values can be specified for ulMode.

- TC_FULL (H'00000000): Both the instruction cache and operand cache are to be cleared.
- TC_EXCLUDE_IC (H'00000001): Only the operand cache is to be cleared (the instruction cache is excluded).
- TC_EXCLUDE_OC (H'00000002): Only the instruction cache is to be cleared (the operand cache is excluded).

If the target cache has been disabled, however, it will not be cleared.

The address range for which the corresponding cache entries are to be cleared is specified by pStart and pEnd. If pStart is not an integer multiple of 16, it is rounded down to the nearest such number; if pEnd is not of the form (integer multiple of 16) - 1, it is rounded up to the nearest such number.

(1) Clearing Cache for a Specified Address Range

This function clears the entries corresponding to the address range from pStart to pEnd in the cache(s) specified by ulMode. If this address range includes a non-cacheable area, entries for that area are not cleared.

When the operand cache is specified as a target (when TC_FULL or TC_EXCLUDE_IC is specified for ulMode), this function copies dirty entries (entries that have not been written into memory) back into memory before clearing them.

This processing is achieved by manipulating the memory-mapped cache. During processing, the value of the SR.IMASK bit remains the same as when the function was called. If interrupts should not be accepted during processing of this function, mask interrupts beforehand.

(2) Clearing All Entries

Specifying pStart = 0 and pEnd = H'ffffffff clears all entries in the cache(s) specified by ulMode. This function performs the following processing.

(a) When TC_FULL or TC_EXCLUDE_OC is specified for ulMode, this function sets the CCR1.ICF bit to 1 to invalidate all entries in the instruction cache. CCR1 is updated while SR.IMASK = 15.

(b) After step (a), if TC_FULL or TC_EXCLUDE_IC has been specified for ulMode, this function writes V = 0 and U = 0 to all entries in the memory-mapped operand cache. At the same time, the dirty entries (U = 1: entries that have not been written back into memory) are copied back into memory. During processing, the value of the SR.IMASK bit remains the same as when the function was called. If interrupts should not be accepted during processing of this function, mask interrupts beforehand.

This function reads the contents of the CCR1 register. If the contents of CCR1 are changed during execution of this function, its operation is undefined.

Do not call this API function when the cache memory is in the module-standby state.

11.6.5 Flush Operand Cache (sh2adual_fls_cac)

C-Language API:

```
INT32 sh2adual_fls_cac(void * pStart, void *pEnd);
```

Parameters:

`pStart`	Address where cache flushing starts
`pEnd`	Address where cache flushing ends

Return Values:

`CAC_E_OK`	Normal end
`CAC_E_PAR`	Parameter error (pStart > pEnd)

Function:

This function flushes the operand cache. More specifically, if the operand cache contains data that have not been written back into memory, these data are copied back into memory. Even when the cache-lock mode has been selected, all cache entries that are locked will be flushed. This does not unlock the entries.

The address range for which the corresponding cache entries are to be flushed is specified by pStart and pEnd. If pStart is not an integer multiple of 16, it is rounded down to the nearest such number; if pEnd is not of the form (integer multiple of 16) - 1, it is rounded up to the nearest such number.

When the operand cache is disabled or in the write-through mode, nothing is done in response to a call of this function; that is, execution simply returns.

(1) Flushing Entries for a Specified Address Range

This function flushes the entries corresponding to the address range from pStart to pEnd in the operand cache, that is, when the specified entries have not been written into memory, the entries are copied back into memory. If this address range includes a non-cacheable area, the entries are not flushed.

This processing is achieved by manipulating the cache based on allocation to memory. During processing, the value of the SR.IMASK bit remains the same as when the function is called. If interrupts should not be accepted during processing of this function, mask interrupts beforehand.

(2) Flushing All Entries

Specifying pStart = 0 and pEnd = H'ffffffff flushes all entries in the operand cache. This function performs the following processing.

This function reads all entries of the operand cache that have been allocated to memory, and writes V = 1 and U = 0 to the valid (V = 1) entries. During processing, the value of the SR.IMASK bit remains the same as when the function was called. If interrupts should not be accepted during processing of this function, mask interrupts beforehand.

This function reads the contents of the CCR1 register. If the contents of CCR1 are changed during execution of this function, operation of the function is undefined. Do not call this API function when the cache memory is in the module-standby state.

11.6.6 Invalidate Cache (sh2adual_inv_cac)

C-Language API:

```
INT32 sh2adual_inv_cac(UINT32 ulMode);
```

Parameter:

`ulMode`	Target cache

Return Values:

`CAC_E_OK`	Normal end
`CAC_E_PAR`	Parameter error (incorrect ulMode)

Function:

This function invalidates the cache.

The target cache is specified by ulMode. Any of the following values can be specified for ulMode.

- TC_FULL (H'00000000): Both the instruction cache and operand cache are to be invalidated.
- TC_EXCLUDE_IC (H'00000001): Only the operand cache is to be invalidated (the instruction cache is excluded).
- TC_EXCLUDE_OC (H'00000002): Only the instruction cache is to be invalidated (the operand cache is excluded).

If the target cache has been disabled, however, it will not be invalidated.

The CCR1 register is updated as follows according to ulMode while SR.IMASK = 15.

- ulMode = TC_FULL

  If the instruction cache is enabled, CCR1.ICF = 1; if the operand cache is enabled, CCR1.OCF = 1.
- ulMode = TC_EXCLUDE_IC

  If the operand cache is enabled, CCR1.OCF = 1.
- ulMode = TC_EXCLUDE_OC

  If the instruction cache is enabled, CCR1.ICF = 1.

Do not call this API function when the cache memory is in the module-standby state.

Section 12 Application Program Creation

12.1 About the FPU

If you intend to use the SH2A-FPU, be sure to read section 20, Notes on the FPU, whether or not you will actually need the floating-point operations.

12.2 Tasks

(1) Writing a Task

As shown in figure 12.1, tasks are written as normal C-language functions. Use an ext_tsk or exd_tsk service call to end a task. If execution returns without ext_tsk or exd_tsk having been called, operation is the same as if ext_tsk was explicitly called.

```
#include "kernel.h"
#pragma noregsave(Task)       <- Since the task entry function does not
                                 need to guarantee the restoration of
                                 register contents, #pragma noregsave
                                 can be specified.
void Task(VP_INT exinf)       <- When a task is initiated by the
{                               TA_ACT attribute or act_tsk, exinf
                                specified at the time of task creation
                                is passed as a parameter; when a task
                                is initiated by sta_tsk, stacd
                                specified by sta_tsk is passed as a
                                parameter.
    /* task processing */
   if(…)
    ext_tsk( );               <- Use an ext_tsk or exd_tsk service
                                call to end a task.
}                             <- If the call has not been made, ext_tsk
                                is automatically called at the end of
                                the function.
```

Figure 12.1 Example of a Task

A task can also be an endless loop. Figure 12.2 shows an example.

```
#include "kernel.h"
#pragma noregsave(Task)
void Task(VP_INT exinf)
{
    while(1) {
        /* task processing */
    }
}
```

Figure 12.2 Example of a Task as an Endless Loop

(2) Rules on Using Registers

Table 12.1 shows rules on using registers in tasks. Refer to this information when debugging or creating tasks in assembly language.

Table 12.1 Rules on Using Registers in Tasks

Registers	Guarantee*[1]	Initial Value
PC	Not necessary	Address where task starts
SR	*[2]	H'00000000
R0 to R3	Not necessary	Undefined
R4	Not necessary	[Activation by TA_ACT attribute or act_tsk] exinf as specified at the time of task creation [Activation by sta_tsk] stacd as specified by sta_tsk
R5 to R14, MACH, MACL, GBR	Not necessary	Undefined
R15	Necessary	Last address of stack area for the task
PR	Necessary	Undefined
TBR	*[3]	*[3]
[SH2A-FPU] FPSCR	Not necessary *[4]	H'00040001
[SH2A-FPU] FPUL, FR0 to FR15	Not necessary *[4]	Undefined

Notes: 1. Indicates whether the values before the task was launched must be restored to the registers when execution is returned from the entry function.

2. Except in the CPU-locked state, IMASK = 0 must be guaranteed.

3. Depends on the system.tbr setting.

(1) system.tbr = NOMANAGE: The kernel does not manipulate the TBR.

(2) system.tbr = FOR_SVC: Do not modify the TBR.

(3) system.tbr = TASK_CONTEXT: No guarantee is required and the initial value is undefined.

4. Available only when the TA_COP1 attribute has been specified; restoration of prior values need not be guaranteed.

12.3 Task Exception Handling Routines

(1) Writing a Task Exception Handling Routine

As shown in figure 12.3, task exception handling routines are written as normal C-language functions.

```
#includle "kernel.h"
#pragma noregsave (Texrtn)               <- Since a task exception
                                           handling routine does not
                                           need to guarantee the
                                           restoration of register
                                           contents, #pragma noregsave
                                           can be specified.
void Texrtn(TEXPTN texptn, VP_INT exinf) <- The source of the exception
                                           and extended information are
                                           passed as parameters.
{
  /* Task exception handling routine */
 }
```

Figure 12.3 Example of a Task Exception Handling Routine

(2) Rules on Using Registers

Table 12.2 shows rules on using registers in task exception handling routines. Refer to this information when debugging or creating a task exception handling routine in assembly language.

Table 12.2 Rules on Using Registers in a Task Exception Handling Routine

Registers	Guarantee*[1]	Initial Value
PC	Not necessary	Address of the task exception handling routine
SR	*[2]	0
R0 to R3	Not necessary	Undefined
R4	Not necessary	Task exception pattern
R5	Not necessary	Extended information on the task
R6 to R14, MACH, MACL, GBR	Not necessary	Undefined
R15	Necessary	Points to the task's stack area
PR	Necessary	Undefined
TBR	*[3]	*[3]
[SH2A-FPU] FPSCR	Not necessary *[4]	H'00040001
[SH2A-FPU] FPUL, FR0 to FR15	Not necessary *[4]	Undefined

Notes: 1. Indicates whether the values before the task was launched must be restored to the registers when execution is returned from the entry function.

2. Except in the CPU-locked state, IMASK = 0 must be guaranteed.

3. Depends on the system.tbr setting.

(1) system.tbr = NOMANAGE: The kernel does not manipulate the TBR.

(2) system.tbr = FOR_SVC: Do not modify the TBR.

(3) system.tbr = TASK_CONTEXT: Restoration must be guaranteed. The initial value is undefined.

4. Available only when the TA_COP1 attribute is specified.

12.4 Extended Service Call Routines

(1) Writing an Extended Service Call Routine

As shown in figure 12.4, extended service call routines are written as normal C-language functions.

```
#include "kernel.h"
ER_UINT Svcrtn(VP_INT par1,VP_INT par2)  <- Parameters specified by
                                            cal_svc are passed to the
                                            extended service call
                                            routine.
{
  /* Extended service call routine */
  return E_OK;                           <- Passes the return value
 }                                          to the caller.
```

Figure 12.4 Example of an Extended Service Call Routine

(2) Rules on Using Registers

An extended service call routine is executed by issuing an cal_svc or ical_svc service call in the same way as a normal function call. Therefore, extended service call routines can use registers in the same way as normal C language functions. For details, refer to the SuperH™ RISC engine C/C++ Compiler User's Manual.

Parameters 1 to 4 specified by cal_svc are stored in the R4 to R7 registers. Note that the caller of cal_svc or ical_svc determines whether or not the FPU registers can be used in an extended service call routine.

12.5 Interrupt Handlers

12.5.1 Types of Interrupt Handler

There are two types of interrupt handler: normal interrupt handlers and direct interrupt handlers.

Normal interrupt handlers, which are initiated via the kernel when an interrupt occurs, are written as normal C-language functions.

Direct interrupt handlers, on the other hand, are registered in the interrupt-exception vector table for the CPU. Since these handlers are directly initiated in response to interrupts, they provide faster operation than normal interrupt handlers. Direct interrupt handlers must be written as interrupt functions (with the #pragma interrupt directive specified).

All of these interrupt handlers are executed in non-task contexts.

12.5.2 Register Banks

In this kernel, whether the interrupt uses register banks or not determines how direct interrupt handlers are written and how normal interrupt handlers are defined. Table 12.3 shows how the usage or non-usage of register banks by interrupts is differentiated.

Table 12.3 Usage of Register Banks

kernel_intspec.h		**cfg File**		
INTSPEC _IBNR_ADR	**INTSPEC _NOBANK_VEC??? Correspondence to a Defined Interrupt No.?**	**system.regbank**	**Interrupt Level**	**Register Bank Used?**
0	—	—	—	No
Other than 0	Yes	—	—	No
	No	NOTUSE	—	No
		ALL	—	Yes
		BANKLEVELxx	Level not defined in system.regbank	No (do not specify VTA_REGBANK when defining a normal interrupt handler)
			Level defined in system.regbank	Yes (VTA_REGBANK must be specified when defining a normal interrupt handler)

12.5.3 Normal Interrupt Handlers

When defining a normal interrupt handler, the VTA_REGBANK attribute must be specified if the interrupt will use register banks. Conversely, the VTA_REGBANK attribute must not be specified if the interrupt is not to use register banks (see table 12.3).

Handlers for interrupts (including the NMI) with a level higher than the kernel interrupt mask level (system.system_IPL) must be written and defined as direct interrupt handlers. If such handlers are written and defined as normal interrupt handlers, correct system operation cannot be guaranteed.

(1) Writing a Normal Interrupt Handler

As shown in figure 12.5, normal interrupt handlers are written as normal C-language functions.

```
#include "kernel.h"
#pragma noregsave(Inh)     <- Since a normal interrupt handler
                              does not need to guarantee the
                              restoration of register contents when
                              the interrupt uses register banks,
                              #pragma noregsave can be specified.
void Inh(void)             <- An interrupt handler is defined as a
{                             function having no parameter or return
                              value.
  /* Handler processing */
}
```

Figure 12.5 Example of a Normal Interrupt Handler

(2) Rules on Using Registers

Table 12.4 shows rules on using registers in normal interrupt handlers. Refer to this information when debugging or creating a normal interrupt handler in assembly language.

Table 12.4 Rules on Using Registers in a Normal Interrupt Handler

Registers	Guarantee*[1]	Initial Value
PC	Not necessary	Address of the normal interrupt handler
SR	*[2]	IMASK: Interrupt level. While a handler is being executed, IMASK must not be lower than the current interrupt level. Other bits: Undefined
R0 to R7	Not necessary	Undefined
R8 to R14, MACH, MACL, GBR	Necessary *[3]	Undefined
R15	Necessary	Points to the interrupt handler's stack area. When an interrupt is accepted, entry/exit processing by the kernel switches the stack to that for the interrupt handlers. All normal interrupt handlers use the same interrupt stack area. The size of the interrupt stack area is defined by system.stack_size. Note that the size must be calculated carefully in consideration of interrupt nesting. For details, refer to section 18.7, Normal Interrupt Handler Stack (system.stack_size).
PR	Necessary	Undefined
TBR	*[4]	*[4]
[SH2A-FPU] FPSCR, FPUL, FR12 to FR15	Necessary *[5]	Undefined

Notes: 1. Indicates whether the values before the task was launched must be restored to the registers when execution is returned from the entry function.

2. The IMASK level must be guaranteed.

3. No guarantee is required when the interrupt uses register banks (see table 12.3).

4. Depends on the system.tbr setting.

(1) system.tbr = NOMANAGE: The kernel does not manipulate the TBR.

(2) system.tbr = FOR_SVC: Do not modify the TBR.

(3) system.tbr = TASK_CONTEXT: Restoration must be guaranteed. The initial value is undefined.

5. To use the FPU in a handler, refer to section 20.3, Floating-Point Operations in Handlers.

12.5.4 Direct Interrupt Handlers

When defining a direct interrupt handler, the VTA_DIRECT attribute must be specified.

Handlers for interrupts (including the NMI) with a level higher than the kernel interrupt mask level (system.system_IPL) must be written and defined as direct interrupt handlers. If these handlers are written and defined as normal interrupt handlers, correct system operation cannot be guaranteed.

(1) Writing a Direct Interrupt Handler

As shown in figure 12.6, direct interrupt handlers are written as interrupt functions. Take care to make specifications for an interrupt function in accord with the conditions listed in table 12.5.

```
#include "kernel.h"
#define stksz 512                                              (1)
VW  stk[stksz/sizeof(VW)];
static const VP  p_stk=(VP)&stk[stksz/sizeof(VW)];     (2)
#pragma interrupt(DirectInh(sp=p_stk,tn=25))             (3)
void DirectInh(void)                                           (4)
{
    /* Handler processing */
}
```

Figure 12.6 Example of a Direct Interrupt Handler

Description:

(1) Allocate a stack area for the interrupt handler.

This is to avoid an overflow of the interrupted program's stack. Interrupt handlers at the same interrupt level can share a stack.

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

(3) Declare the handler as an interrupt function by using a #pragma interrupt statement. Specify the following items.

(a) "sp=" (switching stack)

For interrupts other than the NMI, the stack must be switched. In such cases, specify the variable defined in (2).

(b) "tn=" (return by TRAPA)

For details, see table 12.5.

(c) "resbank" (restore bank register)

If the interrupt uses register banks, "resbank" must be specified. For details, see table 12.5.

(4) Write the handler as a function having no parameter or return value.

Table 12.5 "tn=" and "resbank"

Register Bank Used?	Interrupt Level	"tn="	"resbank"
No *	Higher than system.system_IPL	Prohibited	Prohibited
	Less than or equal to system.system_IPL	"tn=63"	
Yes *	Higher than system.system_IPL	Prohibited	Necessary
	Less than or equal to system.system_IPL	"tn=62"	

Note: See table 12.3.

(2) Rules on Using Registers

Table 12.6 shows rules on using registers in direct interrupt handlers. Refer to this information when debugging or creating a direct interrupt handler in assembly language.

Table 12.6 Rules on Using Registers in a Direct Interrupt Handler

Registers	Guarantee*[1]	Initial Value
PC	Not necessary	Address of the direct interrupt handler
SR	*[2]	IMASK: Interrupt level. While a handler is being executed, IMASK must not be lower than the current interrupt level. Other bits: Same as before the interrupt
R0 to R14, MACH, MACL, GBR	Necessary *[3]	Undefined
R15	Necessary	Points to the stack area for the interrupted program. [Interrupts other than the NMI] To keep the stack in use by the program that was running prior to the interrupt from overflowing, switch the stack to a dedicated stack for the interrupt handler. If this is not done, usage of the interrupted stack by the interrupt handler may cause it to overflow. Direct interrupt handlers at the same interrupt level can share a stack since such interrupt handlers will not be executed simultaneously. When the stack is shared by direct interrupt handlers at the same interrupt level, definition of the stack size must be based on the largest amount of stack usage by a direct interrupt handler. A direct interrupt handler is permitted to use four bytes of the interrupted stack. [The NMI] Do not switch the stack in cases where there is a possibility of nested NMIs. Since the NMI interrupt handler uses the same stack as was in use when the NMI was generated, the amount of stack to be used by the NMI interrupt handler must be added to the stacks for tasks and interrupt handlers.
PR	Necessary *[3]	Undefined
TBR	*[4]	*[4]
[SH2A-FPU] FPSCR, FPUL, FR0 to FR15	Necessary *[5]	Undefined

Notes: 1. Indicates whether the values before the task was launched must be restored to the registers when execution is returned from the entry function (RTE or TRAPA instruction).

2. The IMASK level must be guaranteed.

3. No guarantee is required when the interrupt uses register banks (see table 12.3).

4. Depends on the system.tbr setting.

   (1) system.tbr = NOMANAGE: The kernel does not manipulate the TBR.

   (2) system.tbr = FOR_SVC: Do not modify the TBR.

   (3) system.tbr = TASK_CONTEXT: Restoration must be guaranteed. The initial value is undefined.

5. To use the FPU in a handler, refer to section 20.3, Floating-Point Operations in Handlers.

12.6 CPU Exception Handlers (Including TRAPA Exceptions)

12.6.1 Types of CPU Exception Handler

There are two types of CPU exception handler: normal CPU exception handlers and direct CPU exception handlers.

Normal CPU exception handlers, which are initiated via the kernel when a CPU exception occurs, are written as normal C-language functions. When a CPU exception occurs, its number (vector number) and other information are passed to the normal CPU exception handler as parameters.

Direct CPU exception handlers, on the other hand, are registered in the interrupt-exception vector table for the CPU. Direct CPU exception handlers must be written as interrupt functions (with the #pragma interrupt directive specified). Parameters are not passed to direct CPU exception handlers.

12.6.2 Normal CPU Exception Handlers

(1) Writing a Normal CPU Exception Handler

As shown in figure 12.7, normal CPU exception handlers are written as normal C-language functions.

```
#include "kernel.h"
void Exc(UW excno, VT_EXC *pk_exc)     <- A CPU exception handler is
                                          defined as a function having no
                                          parameter or return value.
                                          CPU exception number and
{                                         other information on the
    /* Handler processing */              exception are passed as excno
}                                         and pk_exc, respectively.
```

Figure 12.7 Example of a Normal CPU Exception Handler

The specification of the VT_EXC attribute is as follows:

```
typedef struct {
    UW  r0;    /* R0 when the CPU exception occurred */
    UW  r1;    /* R1 when the CPU exception occurred */
    UW  r2;    /* R2 when the CPU exception occurred */
    UW  r3;    /* R3 when the CPU exception occurred */
    UW  r4;    /* R4 when the CPU exception occurred */
    UW  r5;    /* R5 when the CPU exception occurred */
    UW  r6;    /* R6 when the CPU exception occurred */
    UW  r7;    /* R7 when the CPU exception occurred */
    UW  pr;    /* PR when the CPU exception occurred */
    UW  pc;    /* PC when the CPU exception occurred */
    UW  sr;    /* SR when the CPU exception occurred */
} VT_EXC;
```

When a normal CPU exception handler is initiated, R8 to R14, GBR, MACH, and MACL have the same values as when the CPU exception occurred. The value of R15 at the time of the CPU exception can be calculated by using the following formula: pk_exc + sizeof (VT_EXC).

(2) Rules on Using Registers

Table 12.7 shows rules on using registers in normal CPU exception handlers. Refer to this information when debugging or creating a normal CPU exception handler in assembly language.

Table 12.7 Rules on Using Registers in a Normal CPU Exception Handler

Registers	Guarantee*[1]	Initial Value
PC	Not necessary	Address of the normal CPU exception handler
SR	*[2]	Same as before the CPU exception
R0 to R3, R6, R7	Not necessary	Undefined
R4	Not necessary	excno (vector number of the CPU exception that occurred)
R5	Not necessary	pk_exc
R8 to R14, MACH, MACL, GBR	Necessary	Same as when the CPU exception occurred
R15	Necessary	Points to the stack area for the program that generated the exception. As a CPU exception handler is re-entrant, the CPU exception handler uses the same stack as the program which generated the exception. A CPU exception handler cannot have a dedicated stack.
PR	Necessary	Undefined
TBR	*[3]	*[3]
[SH2A-FPU] FPSCR, FPUL, FR12 to FR15	Necessary *[4]	Same as when the CPU exception occurred

Notes: 1. Indicates whether the values before the task was launched must be restored to the registers when execution is returned from the entry function.

2. The IMASK level must be guaranteed.

3. Depends on the system.tbr setting.

(1) system.tbr = NOMANAGE: The kernel does not manipulate the TBR.

(2) system.tbr = FOR_SVC: Do not modify the TBR.

(3) system.tbr = TASK_CONTEXT: Restoration must be guaranteed. The initial value is undefined.

4. To use the FPU in a handler, refer to section 20.3, Floating-Point Operations in Handlers.

12.6.3 Direct CPU Exception Handlers

(1) Writing a Direct CPU Exception Handler

As shown in figure 12.8, direct CPU exception handlers are written as interrupt functions.

```
#include "kernel.h"
#pragma interrupt(DirectExc)                                (1)
void DirectExc(void)                                        (2)
{
    /* Handler processing */
}
```

Figure 12.8 Example of a Direct CPU Exception Handler

Description:

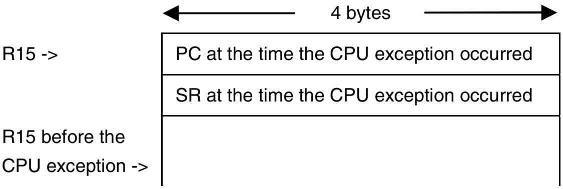
(1) Declare the handler as an interrupt function by using a #pragma interrupt statement. Do not add anything as the interrupt specification of #pragma interrupt.

(2) Write the handler as a function having no parameter or return value.

(2) Rules on Using Registers

Table 12.8 shows rules on using registers in direct CPU exception handlers. Refer to this information when debugging or creating a direct CPU exception handler in assembly language.

Table 12.8 Rules on Using Registers in a Direct CPU Exception Handler

Registers	Guarantee*1	Initial Value
PC	Not necessary	Address of the direct CPU exception handler
SR	*2	Same as before the CPU exception. While a handler is being executed, IMASK must not be lower than the current interrupt level.
R0 to R14, MACH, MACL, GBR	Necessary	Same as before the CPU exception
R15	Necessary	Points to the stack area for the program that generated the exception. As a CPU exception handler is re-entrant, the CPU exception handler uses the same stack as the program which generated the exception. A CPU exception handler cannot have a dedicated stack.  4 bytes R15 -> PC at the time the CPU exception occurred SR at the time the CPU exception occurred R15 before the CPU exception ->
PR	Necessary	Undefined
TBR	*3	*3
[SH2A-FPU] FPSCR, FPUL, FR0 to FR15	Necessary *4	Undefined

Notes: 1. Indicates whether the values before the task was launched must be restored to the registers when execution is returned from the entry function (by an RTE or TRAPA instruction).

2. The IMASK level must be guaranteed.

3. Depends on the system.tbr setting.

(1) system.tbr = NOMANAGE: The kernel does not manipulate the TBR.

(2) system.tbr = FOR_SVC: Do not modify the TBR.

(3) system.tbr = TASK_CONTEXT: Restoration must be guaranteed. The initial value is undefined.

4. To use the FPU in a handler, refer to section 20.3, Floating-Point Operations in Handlers.

12.7 Time Event Handlers

(1) Writing a Time Event Handler

Time event handlers are written as normal C-language functions. Figure 12.9 shows an example of a cyclic handler and an alarm handler. Figure 12.10 shows an example of an overrun handler written in the C language. These handlers are executed in non-task contexts.

```
#include "kernel.h"
void Handler(VP_INT exinf)      <- exinf is passed as a parameter
                                   specified at the time of generation.
{
    /* Handler processing */
}
```

Figure 12.9 Example of a Cyclic Handler and an Alarm Handler

```
#include “kernel.h”
void Overhdr (ID tskid, VP_INT exinf)   <- tskid indicating the
                                           initiating factor and
                                           exinf for the task are
                                           passed.
{
    /* Handler processing */
}
```

Figure 12.10 Example of an Overrun Handler

(2) Rules on Using Registers

Table 12.9 shows rules on using registers in time event handlers. Refer to this information when debugging or creating a time event handler in assembly language.

Table 12.9 Rules on Using Registers in a Time Event Handler

Registers	Guarantee*[1]	Initial Value
PC	Not necessary	Address of the time event handler
SR	*[2]	(1) IMASK When the handler is initiated by a timer interrupt: timer interrupt level (clock.IPL) When the handler is initiated by calling vrst_tmr: whichever is greater of the timer interrupt level (clock.IPL) and the IMASK level when vrst_tmr was called While a handler is being executed, IMASK must not be lower than the current interrupt level. (2) Other bits: Undefined
R0 to R3	Not necessary	Undefined
R4	Not necessary	Cyclic handler or alarm handler: Extended information on the handler Overrun handler: Target task ID
R5	Not necessary	Cyclic handler or alarm handler: Undefined Overrun handler: Extended information on the target task
R6, R7	Not necessary	Undefined
R8 to R14, MACH, MACL, GBR	Necessary	Undefined
R15	Necessary	Points to the timer stack area.
PR	Necessary	Undefined
TBR	*[3]	*[3]
[SH2A-FPU] FPSCR, FPUL, FR12 to FR15	Necessary *[4]	Undefined

Notes: 1. Indicates whether the values before the task was launched must be restored to the registers when execution is returned from the entry function.

2. The IMASK level must be guaranteed.

3. Depends on the system.tbr setting.
   (1) system.tbr = NOMANAGE: The kernel does not manipulate the TBR.
   (2) system.tbr = FOR_SVC: Do not modify the TBR.
   (3) system.tbr = TASK_CONTEXT: Restoration must be guaranteed. The initial value is undefined.
4. To use the FPU in a handler, refer to section 20.3, Floating-Point Operations in Handlers.

12.8 Initialization Routines

(1) Writing an Initialization Routine

Initialization routines are written as normal C-language functions. Figure 12.11 shows an example of an initialization routine. Initialization routines are executed in non-task contexts.

```
#include "kernel.h"
void InitRoutine(VP_INT exinf)        <- exinf is passed as a parameter
                                      specified at the time of
                                      generation.
{
    /* Handler processing */
}
```

Figure 12.11 Example of an Initialization Routine

(2) Rules on Using Registers

Table 12.10 shows rules on using registers in initialization routines. Refer to this information when debugging or creating an initialization routine in assembly language.

Table 12.10 Rules on Using Registers in an Initialization Routine

Registers	Guarantee*1	Initial Value
PC	Not necessary	Address of the initialization routine
SR	*2	(1) IMASK: Kernel interrupt mask level (system.system_IPL) While a routine is being executed, IMASK must not be lower than the current interrupt level. (2) Other bits: Undefined
R0 to R3	Not necessary	Undefined
R4	Not necessary	Extended information on the routine
R5 to R7	Not necessary	Undefined
R8 to R14, MACH, MACL, GBR	Necessary	Undefined
R15	Necessary	Points to the kernel stack area.
PR	Necessary	Undefined
TBR	*3	*3
[SH2A-FPU] FPSCR, FPUL, FR12 to FR15	Necessary *4	Undefined

Notes: 1. Indicates whether the values before the task was launched must be restored to the registers when execution is returned from the entry function.

2. The IMASK level must be guaranteed.

3. Depends on the system.tbr setting.

(1) system.tbr = NOMANAGE: The kernel does not manipulate the TBR.

(2) system.tbr = FOR_SVC: Do not modify the TBR.

(3) system.tbr = TASK_CONTEXT: Restoration must be guaranteed. The initial value is undefined.

4. To use the FPU in a routine, refer to section 20.3, Floating-Point Operations in Handlers.

12.9 Timer Drivers

When TIMER has been specified for clock.timer in the cfg file, it is necessary to create a timer driver and link it to the kernel. Each CPU requires its own timer driver.

A timer driver consists of the functions listed below. Implement a timer driver with reference to the information on the following pages and in timer driver files provided as samples.

- tdr_ini_tmr(): Initialize the timer
- tdr_int_tmr(): Execute timer-interrupt handling
- tdr_stp_tmr(): Stop the timer
- tdr_rst_tmr(): Restart the timer

12.9.1 tdr_ini_tmr(): Initialize Timer

Format:

```
        void tdr_ini_tmr(void);
```

Parameter:

None

Return Value:

None

Function:

This function initializes the timer.

For initialization, use the macros listed below. They are output to kernel_macro.h by cfg72mp. In particular, the time cycle for timer interrupts must be TIC_NUME/TIC_DENO [ms]. kernel_macro.h is included from within kernel.h.

- TIC_NUME: Numerator of the timer-interrupt cycle (ms)
- TIC_DENO: Denominator of the timer-interrupt cycle (ms)
- TIM_LVL: Timer interrupt level

When TIMER has been specified for clock.timer, cfg72mp automatically takes the following actions.

(a) Registers tdr_ini_tmr() as an initialization routine.

(b) Defines the processing module within the kernel library as a direct interrupt handler for the interrupt number specified as clock.number. tdr_int_tmr() will be called from this kernel module.

(2) Rules on Using Registers

Table 12.11 shows rules on using registers in tdr_ini_tmr(). Refer to this information when debugging or creating tdr_ini_tmr() in assembly language.

Table 12.11 Rules on Using Registers in tdr_ini_tmr()

Registers	Guarantee*[1]	Initial Value
PC	Not necessary	tdr_ini_tmr()
SR	*[2]	IMASK: Kernel interrupt mask level (system.system_IPL) While tdr_ini_tmr() is being executed, IMASK must not be lower than the current interrupt level. Other bits: Undefined
R0 to R7	Not necessary	Undefined
R8 to R14, MACH, MACL, GBR	Necessary	Undefined
R15	Necessary	Points to the kernel stack area.
PR	Necessary	Undefined
TBR	*[3]	*[3]
[SH2A-FPU] FPSCR, FPUL, FR12 to FR15	Necessary *[4]	Undefined

Notes: 1. Indicates whether the values before the task was launched must be restored to the registers when execution is returned from the entry function.

2. The IMASK level must be guaranteed.

3. Depends on the system.tbr setting.

(1) system.tbr = NOMANAGE: The kernel does not manipulate the TBR.

(2) system.tbr = FOR_SVC: Do not modify the TBR.

(3) system.tbr = TASK_CONTEXT: Restoration must be guaranteed. The initial value is undefined.

4. To use the FPU in tdr_ini_tmr(), refer to section 20.3, Floating-Point Operations in Handlers.

12.9.2 tdr_int_tmr(): Execute Timer-Interrupt Handling

Format:

```
void tdr_int_tmr(void);
```

Parameter:

None

Return Value:

None

Function:

This function clears the timer interrupt source.

This function is called from within a timer-interrupt direct interrupt handler in the kernel: in other words, it is called in a non-task context. Service calls for non-task contexts can be issued from within this function.

Table 12.12 shows rules on using registers in tdr_int_tmr(). Refer to this information when debugging or creating tdr_int_tmr() in assembly language.

Table 12.12 Rules on Using Registers in tdr_int_tmr()

Registers	Guarantee*[1]	Initial Value
PC	Not necessary	tdr_int_tmr()
SR	*[2]	IMASK: Timer interrupt level (clock.IPL) While tdr_int_tmr() is being executed, IMASK must not be lower than the value before the task was launched. Other bits: Undefined
R0 to R7	Not necessary	Undefined
R8 to R14, MACH, MACL, GBR	Necessary	Undefined
R15	Necessary	Points to the timer stack area.
PR	Necessary	Undefined
TBR	*[3]	*[3]
[SH2A-FPU] FPSCR, FPUL, FR12 to FR15	Necessary *[4]	Undefined

Notes: 1. Indicates whether the values before the task was launched must be restored to the registers when execution is returned from the entry function.

2. The IMASK level must be guaranteed.

3. Depends on the system.tbr setting.

(1) system.tbr = NOMANAGE: The kernel does not manipulate the TBR.

(2) system.tbr = FOR_SVC: Do not modify the TBR.

(3) system.tbr = TASK_CONTEXT: Restoration must be guaranteed. The initial value is undefined.

4. To use the FPU in tdr_int_tmr(), refer to section 20.3, Floating-Point Operations in Handlers.

12.9.3 tdr_stp_tmr(): Stop Timer

Format:

```
void tdr_stp_tmr(void);
```

Parameter:

None

Return Value:

None

Function:

This function stops the timer so that timer interrupts will not occur.

tdr_stp_tmr() is a callback function from the vstp_tmr service call and is executed as part of the processing of the vstp_tmr service call. Service calls for non-task contexts can be issued from within this function.

This function need not be implemented when the vstp_tmr service call has not been selected in the cfg file.

Table 12.13 shows rules on using registers in tdr_stp_tmr(). Refer to this information when debugging or creating tdr_stp_tmr() in assembly language.

Table 12.13 Rules on Using Registers in tdr_stp_tmr()

Registers	Guarantee*[1]	Initial Value
PC	Not necessary	tdr_stp_tmr()
SR	*[2]	IMASK: Timer interrupt level (clock.IPL) While tdr_stp_tmr() is being executed, IMASK must not be lower than the value before the task was launched. Other bits: Undefined
R0 to R7	Not necessary	Undefined
R8 to R14, MACH, MACL, GBR	Necessary	Undefined
R15	Necessary	Points to the timer stack area.
PR	Necessary	Undefined
TBR	*[3]	*[3]
[SH2A-FPU] FPSCR, FPUL, FR12 to FR15	Necessary *[4]	Undefined

Notes: 1. Indicates whether the values before the task was launched must be restored to the registers when execution is returned from the entry function.

2. The IMASK level must be guaranteed.

3. Depends on the system.tbr setting.

(1) system.tbr = NOMANAGE: The kernel does not manipulate the TBR.

(2) system.tbr = FOR_SVC: Do not modify the TBR.

(3) system.tbr = TASK_CONTEXT: Restoration must be guaranteed. The initial value is undefined.

4. To use the FPU in tdr_stp_tmr(), refer to section 20.3, Floating-Point Operations in Handlers.

12.9.4 tdr_rst_tmr(): Restart Timer

Format:

```
void tdr_rst_tmr(void);
```

Parameter:

None

Return Value:

None

Function:

This function restarts the timer so that timer interrupts will occur.

tdr_rst_tmr() is a callback function from the vrst_tmr or ivrst_tmr service call and is executed as part of the processing of the vrst_tmr or ivrst_tmr service call. Service calls for non-task contexts can be issued from within this function.

This function need not be implemented unless the vrst_tmr or ivrst_tmr service call has been selected in the cfg file.

Table 12.14 shows rules on using registers in tdr_rst_tmr(). Refer to this information when debugging or creating tdr_rst_tmr() in assembly language.

Table 12.14 Rules on Using Registers in tdr_rst_tmr()

Registers	Guarantee*[1]	Initial Value
PC	Not necessary	tdr_rst_tmr()
SR	*[2]	IMASK: Whichever of the following values is greater: - timer interrupt level (clock.IPL); - IMASK level at the time vrst_tmr or ivrst_tmr was issued. While tdr_rst_tmr() is being executed, IMASK must not be lower than the value before the task was launched. Other bits: Undefined
R0 to R7	Not necessary	Undefined
R8 to R14, MACH, MACL, GBR	Necessary	Undefined
R15	Necessary	Points to the timer stack area.
PR	Necessary	Undefined
TBR	*[3]	*[3]
[SH2A-FPU] FPSCR, FPUL, FR12 to FR15	Necessary *[4]	Undefined

Notes: 1. Indicates whether the values before the task was launched must be restored to the registers when execution is returned from the entry function.

2. The IMASK level must be guaranteed.

3. Depends on the system.tbr setting.

(1) system.tbr = NOMANAGE: The kernel does not manipulate the TBR.

(2) system.tbr = FOR_SVC: Do not modify the TBR.

(3) system.tbr = TASK_CONTEXT: Restoration must be guaranteed. The initial value is undefined.

4. To use the FPU in tdr_rst_tmr(), refer to section 20.3, Floating-Point Operations in Handlers.

12.10 System-Down Routines

The system-down routine is written as the following C-language function. Note that the name of the routine is fixed.

```
    void    _kernel_sysdwn(W type, VW inf1, VW inf2, VW inf3)
```

The system-down routine must be created and linked to the kernel.

Table 12.15 lists the specifications of parameters passed to the system-down routine.

Although the system-down routine can perform processing in response to abnormal conditions, it cannot use kernel functions such as system calls if the internal operation of the kernel led to the system going down (error type is negative).

Furthermore, execution does not return from a system-down routine.

When debugging an application program, preserve the state at the time the system went down, make the program enter an endless loop, analyze the reasons for the system going down, and apply countermeasures.

Table 12.15 Parameters Passed to the System-Down Routine

Cause	Error Type: W type (R4)	System-Down Information 1: VW inf1 (R5)	System-Down Information 2: VW inf2 (R6)	System-Down Information 3: VW inf3 (R7)
vsys_dwn or ivsys_down service call	1 to H'7fffffff	Parameters of the vsys_dwn or ivsys_down service call		
Contents of the packet containing the initial registration information created by the cfg file are incorrect (corrupted data etc.)	0	0	Address where incorrect data were detected	Undefined
The initial registration information in the cfg file is incorrect		Error code (negative) *1	Object type *1	Object number *1
A context error has occurred due to an ext_tsk service call being issued in a non-task context	H'ffffffff (-1)	E_CTX (H'ffffffe7)	Address where ext_tsk was called	Undefined
A context error has occurred due to an exd_tsk service call being issued in a non-task context	H'fffffffe (-2)	E_CTX (H'ffffffe7)	Address where ext_tsk was called	Undefined
Detection of a stack overflow due to a service call issued in a task context	H'fffffff8 (-8)	Task ID	First address of the task stack area	Value of the stack pointer on detection of the overflow
An undefined interrupt has occurred.	H'fffffff0 (-16)	Vector number	Undefined	Undefined
An undefined CPU exception has occurred.			PC value when the exception occurred	VT_EXC *pk_exc *2

Notes: 1. The error code, object type, and object number correspond to the object for which initial registration failed, as listed in table 12.16.

2. Valid only when system.vector_type is ROM or RAM.

Table 12.16 Object Types and Numbers for Failures in Initial Registration

Object	Error Code (inf1) *	Object Type (inf2)	Object Number (inf3)
Interrupt or CPU exception handler	def_inh	0	Vector number
Task	cre_tsk	1	Local task ID
Task exception handling routine	def_tex	2	Local task ID
Semaphore	cre_sem	3	Local semaphore ID
Event flag	cre_flg	4	Local event-flag ID
Data queue	cre_dtq	5	Local data-queue ID
Mailbox	cre_mbx	6	Local mailbox ID
Mutex	cre_mtx	7	Local mutex ID
Message buffer	cre_mbf	8	Local message-buffer ID
Fixed-sized memory pool	cre_pf	9	Local fixed-sized memory pool ID
Variable-sized memory pool	cre_mpl	10	Local variable-sized memory pool ID
Cyclic handler	cre_cyc	11	Local cyclic-handler ID
Alarm handler	cre_alm	12	Local alarm-handler ID
Overrun handler	def_ovr	13	Undefined
Extended service call routine	def_svc	14	Function code

Note: The error code is that returned by the service call listed in this column.

Section 13 Generating Load Modules

13.1 Introduction

In the HI7200/MP, a separate load module should be generated for each CPU.

Use the following procedures to generate a load module in most cases.

(1) Creating a High-performance Embedded Workshop workspace project

When using the High-performance Embedded Workshop, copy the provided sample High-performance Embedded Workshop workspace project and use it as a template.

(2) Coding application programs

Code application programs with reference to provided sample programs.

(3) Creating a cfg file

Create a cfg file (with extension .cfg), which defines the task entry addresses or stack sizes, by using a text editor with reference to the provided sample cfg file.

A cfg file can also be created through the GUI configurator.

(4) Executing the configurator

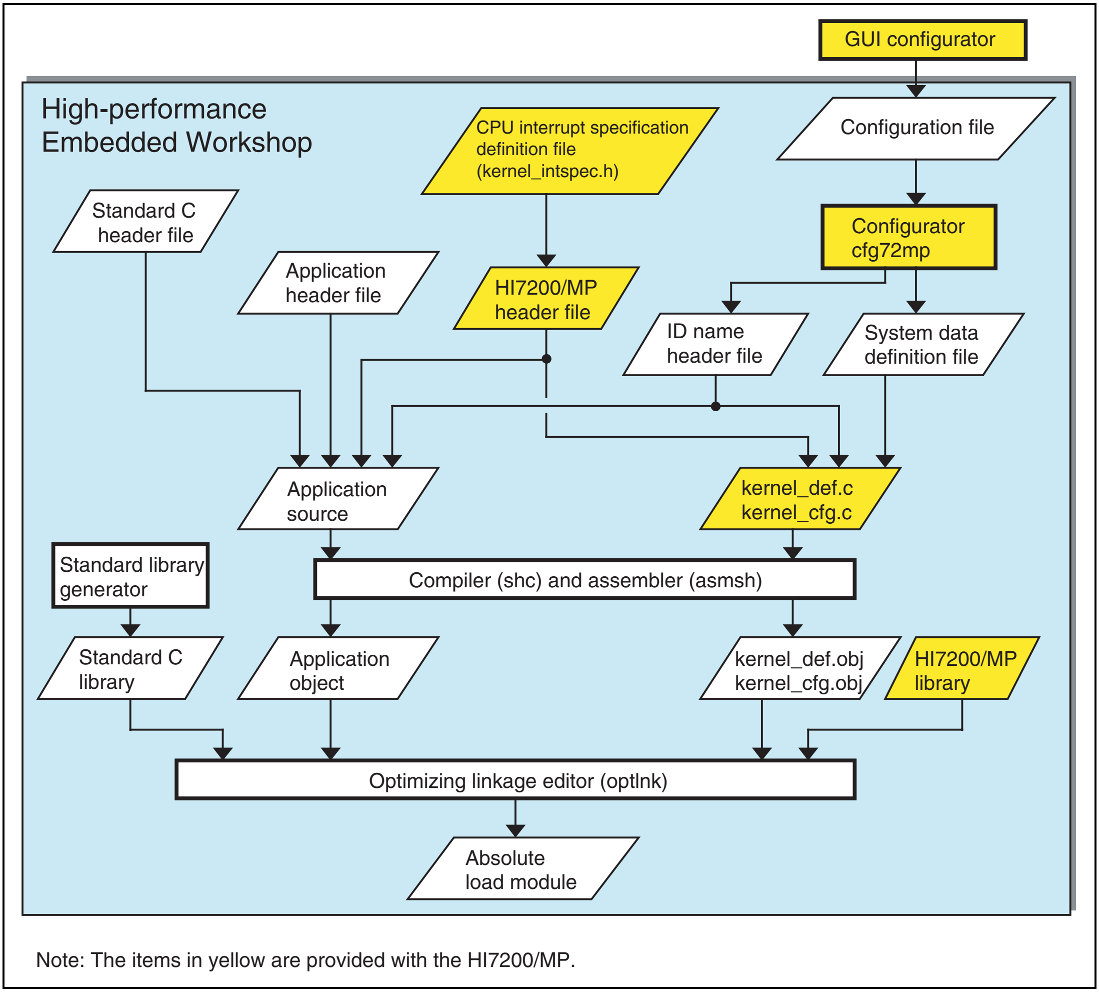
Use configurator cfg72mp to create configuration and header files.

To complete the configurator processing and build processing in a single execution, register cfg72mp as a custom build phase in the High-performance Embedded Workshop.

(5) Generating a load module through a build

Use the High-performance Embedded Workshop to execute a build and generate a load module.

Figure 13.1 shows a flowchart of load module generation.


GUI configurator
High-performance Embedded Workshop
CPU interrupt specification definition file (kernel_intspec.h)
Configuration file
Standard C header file
Configurator cfg72mp
Application header file
HI7200/MP header file
ID name header file
System data definition file
Application source
kernel_def.c kernel_cfg.c
Standard library generator
Compiler (shc) and assembler (asmsh)
Standard C library
Application object
kernel_def.obj kernel_cfg.obj
HI7200/MP library
Optimizing linkage editor (optlnk)
Absolute load module

Note: The items in yellow are provided with the HI7200/MP.

Figure 13.1 Flowchart of Load Module Generation

Section 14 Configurator (cfg72mp)

14.1 Representation Format in cfg File

This section describes the representation format of the definition data in the cfg file.

14.1.1 Comment Statement

A statement from a double slash (//) to the end of a line is handled as a comment and no processing is applied.

14.1.2 End of Statement

A statement must end with a semicolon (;).

14.1.3 Definition Statement

A definition statement must be written in the cfg file in either of the following formats.

Format 1:

```
Definition name {
    Definition item name = Setting ;
    Definition item name = Setting ;
   . . .
};
```

Format 2:

```
Definition name[ Number ] {
    Definition item name = Setting ;
    Definition item name = Setting ;
   . . .
};
```

Definition names and definition item names are character strings that are prescribed in the cfg file specification, which are collectively called keywords. The format of keywords is the same as that of symbols to be described later.

Settings are values that should be determined by the user. They can be written in the format of numeric values, symbols, or external reference names to be described later. Available formats depend on the keyword to define. Some keywords allow multiple settings to be made.

Numbers are used to distinguish between multiple definitions of a same type, such as definitions of multiple tasks. Whether to use a number depends on the definition name.

Numbers must be written in the numeric value format. The meaning of a number depends on the definition name; for example, it can be a task ID, a semaphore ID, or an interrupt vector number. For some definition names, numbers can be omitted.

14.1.4 Numeric Value

A numeric value must be written in one of the following formats.

- Hexadecimal

  Add "0x" or "0X" at the beginning of a numeric value or add "h" or "H" at the end. In the latter format, be sure to add "0" at the beginning when the value begins with an alphabetic letter from A to F or a to f. Note that the configurator does not distinguish between uppercase and lowercase letters for alphabetic letters (A to F or a to f) used in numeric value representation. [2]
- Decimal

  Simply write an integer value as is usually done (23, for example). Note that a decimal value must not begin with "0".
- Octal

  Add "0" at the beginning of a numeric value or add "O" or "o" at the end.
- Binary

  Add "B" or "b" at the end of a numeric value. Note that a binary value must not begin with "0".

[2] The configurator distinguishes uppercase and lowercase letters except for A to F and a to f in numeric value representation.

Table 14.1 Examples of Numeric Value Representation

Format	Example
Hexadecimal	0xf12
	0Xf12
	0a12h
	0a12H
	12h
	12H
Decimal	32
Octal	017
	17o
	17O
Binary	101110b
	101010B

A numeric value can include operators. Table 14.2 shows the available operators.

Table 14.2 Operators

Operator	Precedence	Direction of Computation
()	High	Left to right
-(unary minus)		Right to left
* / %		Left to right
+ -(binary minus)	Low	Left to right

The following are examples of numeric values.

- 123
- 123 + 0x23
- (23/4 + 3) * 2
- 100B + 0aH

A numeric value greater than 0xFFFFFFFF must not be specified.

14.1.5 Symbol

A symbol is a string of numeric characters, uppercase alphabetic letters, lowercase alphabetic letters, underscores (_), and question marks (?). It must not begin with a numeric character.

The following are examples of symbols.

- _TASK1
- IDLE3

A symbol is used to specify an object ID name or a section name.

14.1.6 External Reference Name

An external reference name is an external reference symbol name in C language, which consists of numeric characters, uppercase alphabetic letters, lowercase alphabetic letters, underscores (_), and dollar signs ($). It must not begin with a numeric character and must end with "()".

An external reference name is used to refer to the address of an external function or variable from the cfg file.

Table 14.3 shows examples of external reference names.

Table 14.3 Examples of External Reference Names

External Definition Symbol to be Referred to	Representation of External Reference Name
main() function in C language	main()
int data in C language	(This cannot be represented because it is not an address.)
Address of int data in C language	data()
Address of int *pointer in C language	pointer()
Address of the int array[] array in C language	array()
Label _LABEL1 in the assembly language	LABEL1()
Label LABEL2 in the assembly language	(Cannot be referred to.)

14.1.7 Note

The configurator does not detect errors regarding duplicate specifications of ID names, section names, and external reference names in the cfg file. In most cases, such errors will be reported when the file output from the configurator is compiled.

14.2 Default cfg File

For most definition items, if the user omits settings, the settings in the default cfg file are used. The default cfg file is <RTOS_INST>\cfg72mp\default.cfg. Be sure not to edit this file.

14.3 Definition Items in cfg File

The following items should be defined in the cfg file.

- System definition (system)
- Maximum ID definition (maxdefine)
- Default task stack area definition (memstk)
- Default data queue area definition (memdtq)
- Default message buffer area definition (memmbf)
- Default fixed-sized memory pool area definition (memmpf)
- Default variable-sized memory pool area definition (memmpl)
- System clock definition (clock)
- Remote service-call environment definition (remote_svc)
- Task definition (task[])
- Static stack area definition (static_stack[])
- Semaphore definition (semaphore[])
- Event flag definition (flag[])
- Data queue definition (dataqueue[])
- Mailbox definition (mailbox[])
- Mutex definition (mutex[])
- Message buffer definition (message_buffer[])
- Fixed-sized memory pool definition (memorypool[])
- Variable-sized memory pool definition (variable_memorypool[])
- Cyclic handler definition (cyclic_hand[])
- Alarm handler definition (alarm_hand[])
- Overrun handler definition (overrun_hand)
- Extended service call routine definition (extend_svc[])
- Interrupt handler and CPU exception handler definition (interrupt_vector[])
- Initialization routine definition (init_routine[])
- Service call definition (service_call)

14.3.1 Description Format

Description: Describes the definition item.

Definition Format: Shows the definition format that can be used for the definition item.

Specifiable Range: Shows the range of values that can be set.

Default Setting: Describes the value or processing when the definition is omitted.

GUI Configurator Item: Shows the corresponding definition item name in the GUI configurator.

Remarks: Describes the specification in special cases.

14.3.2 Defining the System (system)

This defines general information regarding the kernel system. The system definition must not be omitted.

Format

```
system {
       cpuid             = <Setting>; // (1) CPUID
       stack_size        = <Setting>; // (2) Interrupt stack size
       kernel_stack_size = <Setting>; // (3) Kernel stack size
       priority          = <Setting>; // (4) Maximum task priority
       system_IPL        = <Setting>; // (5) Kernel interrupt mask level
       message_pri       = <Setting>; // (6) Maximum message priority
       tic_deno          = <Setting>; // (7) Time tick denominator
       tic_nume          = <Setting>; // (8) Time tick numerator
       tbr               = <Setting>; // (9) TBR register usage
       parameter_check   = <Setting>; // (10) Service call parameter check
       mpfmanage         = <Setting>; // (11) Fixed-sized memory pool management
       newmpl            = <Setting>; // (12) Variable-sized memory pool
                                            management
       trace             = <Setting>; // (13) Service call trace
       trace_buffer      = <Setting>; // (14) Buffer size for service call trace
       trace_object      = <Setting>; // (15) Object count for service call trace
       action            = <Setting>; // (16) Object manipulation
       vector_type       = <Setting>; // (17) Interrupt vector type
       regbank           = <Setting>; // (18) Register bank usage
};
```

Contents

(1) CPUID (cpuid)

Description: Defines the ID of the CPU on which this kernel is to run.

Definition Format: Numeric value

Specifiable Range: 1 or 2

Default Setting: This definition must not be omitted (an error will result).

GUI Configurator Item: CFG_MYCPUID

(2) Interrupt stack size (stack_size)

Description: Defines the size of the stack used by normal interrupt handlers. The specified value is rounded up to a multiple of four. For the calculation of the value to be set, refer to section 18.7, Normal Interrupt Handler Stack (system.stack_size).

According to this definition, the BC_hiirqstk section of stack_size bytes is generated.

Definition Format: Numeric value

Specifiable Range: 128 to 0x20000000

Default Setting: Setting in the default cfg file (0x1000 at shipment) (with a warning)

GUI Configurator Item: CFG_IRQSTKSZ

Remarks: When system.vector_type is set to ROM_ONLY_DIRECT or RAM_ONLY_DIRECT, this definition has no meaning. The BC_hiirqstk section is not generated in this case.

(3) Kernel stack size (kernel_stack_size)

Description: Defines the size of the stack used by the kernel. The specified value is rounded up to a multiple of four. For the calculation of the value to be set, refer to section 18.10, Kernel Stack (system.kernel_stack_size).

According to this definition, the BC_hiknlstk section of kernel_stack_size bytes is generated.

Definition Format: Numeric value

Specifiable Range: 256 to 0x20000000

Default Setting: Setting in the default cfg file (0x400 at shipment) (with a warning)

GUI Configurator Item: CFG_KNLSTKSZ

(4) Maximum task priority (priority)

Description: Defines the maximum priority of the tasks used in the application.

Definition Format: Numeric value

Specifiable Range: 1 to 255

Default Setting: Setting in the default cfg file (255 at shipment) (with a warning)

GUI Configurator Item: CFG_MAXTSKPRI

(5) Kernel interrupt mask level (system_IPL)

Description: Defines the interrupt mask level used when a critical section of the kernel is executed. An interrupt higher in priority than this mask level is treated as a non-kernel interrupt.

Definition Format: Numeric value

Specifiable Range: 1 to 15

Default Setting: Setting in the default cfg file (15 at shipment) (with a warning)

GUI Configurator Item: CFG_KNLMSKLVL

(6) Maximum message priority (message_pri)

Description: Defines the maximum priority of the messages used in the mailbox function.

When the mailbox function is not used, this definition has no meaning.

Definition Format: Numeric value

Specifiable Range: 1 to 255

Default Setting: Setting in the default cfg file (255 at shipment) (with a warning)

GUI Configurator Item: CFG_MAXMSGPRI

(7) Time tick denominator (tic_deno)

Description: Defines the denominator of the time tick. At least the time tick numerator or denominator must be set to 1.

The time tick (cycle of kernel timer interrupts) is calculated by the following equation.

Time tick (ms) = tic_nume / tic_deno

For example, to set the system clock cycle to 10 ms, specify tic_deno = 1 and tic_nume = 10.

To set it to 0.1 ms, specify tic_deno = 10 and tic_nume = 1.

If neither the denominator nor numerator is 1 (tic_deno = 5 and tic_nume = 4, for example), an error will result.

The unit of time used by the service calls is always milliseconds regardless of the tic_nume and tic_deno settings. tic_nume and tic_deno define the precision of the time managed by the kernel.

When the time management function is not used (system.timer is set to NOTIMER), the tic_deno and tic_deno definitions have no meaning.

Definition Format: Numeric value

Specifiable Range: 1 to 100

Default Setting: Setting in the default cfg file (1 at shipment) (with a warning)

GUI Configurator Item: CFG_TICDENO

(8) Time tick numerator (tic_nume)

Description: Defines the numerator of the time tick. For details, refer to the above description of tic_deno.

Definition Format: Numeric value

Specifiable Range: 1 to 65535

Default Setting: Setting in the default cfg file (1 at shipment) (with a warning)

GUI Configurator Item: CFG_TICNUME

(9) TBR register usage (tbr)

Description: Defines the usage of the TBR register in the CPU.

Definition Format: Symbol

Specifiable Range: Select from the following.

- NOMANAGE: The TBR is not used or managed.
- FOR_SVC: The TBR is used to issue service calls.
- TASK_CONTEXT: The TBR is managed as the task context register.

Default Setting: Setting in the default cfg file (NOMANAGE at shipment) (with a warning)

GUI Configurator Item: CFG_TBR

(10) Service call parameter check (parameter_check)

Description: Defines whether to detect errors in service call parameters. When the parameter check function is not selected, the parameter errors indicated as [p] in section 6. Kernel Service Calls, are not detected; in this case, the operation is undefined if a service call includes a parameter error. Note that the service call processing is faster when the parameter check function is not selected.

Definition Format: Symbol

Specifiable Range: Select from the following.

- YES: Detects errors in service call parameters.
- NO: Does not detect errors in service call parameters.

Default Setting: Setting in the default cfg file (YES at shipment) (with a warning)

GUI Configurator Item: CFG_PARCHK

(11) Fixed-sized memory pool management (mpfmanage)

Description: Defines the management of fixed-sized memory pools; in particular, this item defines whether to store the kernel management information in the memory pool area (IN) or not (OUT). For the difference between these management methods, refer to section 5.11, Fixed-Sized Memory Pools.

When the fixed-sized memory pool function is not used, this definition has no meaning.

Definition Format: Symbol

Specifiable Range: Select from the following.

- IN: Stores the kernel management information in the memory pool area.
- OUT: Does not store the kernel management information in the memory pool area.

Default Setting: Setting in the default cfg file (IN at shipment) (with a warning)

GUI Configurator Item: CFG_MPFMANAGE

(12) Variable-sized memory pool management (newmpl)

Description: Defines whether to use the conventional method (PAST) or new method (NEW) to manage variable-sized memory pools.

The new method has the following advantages over the conventional method.

- Reduces the degree of fragmentation of the free space in variable-sized memory pools.
- Reduces the overhead of memory block acquisition and release.

For the difference between these management methods, refer to section 5.12.1, Controlling Memory Fragmentation.

When the variable-sized memory pool function is not used, this definition has no meaning.

Definition Format: Symbol

Specifiable Range: Select from the following.

- PAST: Conventional method

— NEW: New method

Default Setting: Setting in the default cfg file (PAST at shipment) (with a warning)

GUI Configurator Item: CFG_NEWMPL

(13) Service call trace (trace)

Description: Defines whether to incorporate the service call trace function. The acquired service call trace information can be displayed through the debugging extension.

This function increases the service call processing time. For the service call trace, refer to section 5.14.4, Service Call Trace.

Definition Format: Symbol

Specifiable Range: Select from the following.

— NO: Does not incorporate the service call trace function.

— TARGET_TRACE: Stores the service call trace information in the buffer on the target system. The buffer size should be specified in trace_buffer described below.

— TOOL_TRACE: Stores the service call trace information in the emulator or simulator manufactured by Renesas. Note that some emulators do not support this function. In an emulator not supporting this function, the service call trace information cannot be displayed through the debugging extension even if TOOL_TRACE is specified.

Default Setting: Setting in the default cfg file (NO at shipment) (with a warning)

GUI Configurator Item: CFG_TRACE

(14) Buffer size for service call trace (trace_buffer)

Description: Defines the buffer size when trace is set to TARGET_TRACE. Specify the size in bytes. The specified value is rounded up to a multiple of four.

Definition Format: Numeric value

Specifiable Range: 512 to 0x20000000

Default Setting: Setting in the default cfg file (0x10000 at shipment) (with a warning)

GUI Configurator Item: CFG_TRCBUFSZ

Remarks: When system.trace != TARGET_TRACE, this definition has no meaning.

(15) Object count for service call trace (trace_object)

Description: Defines the number of objects that can be acquired through the service call trace function.

Definition Format: Numeric value

Specifiable Range: 0 to 32

Default Setting: Setting in the default cfg file (4 at shipment) (with a warning)

GUI Configurator Item: CFG_TRCOBJCNT

Remarks: When system.trace is set to NO, this definition has no meaning.

(16) Object manipulation (action)

Description: Defines whether to incorporate the function for issuing service calls from the debugger, such as the debugging extension, that supports the OS debugging function.

As the object manipulation function uses a cyclic handler, the maxdefine.max_cyh, service_call.cre_cyc, and icre_cyc values may be corrected in some cases.

Even when this item is set to YES, if the time management function is not selected (clock.timer == NOTIMER), a warning message is output and this item is corrected to NO.

Definition Format: Symbol

Specifiable Range: Select from the following.

- YES: Incorporates the object manipulation function.
- NO: Does not incorporate the object manipulation function.

Default Setting: Setting in the default cfg file (NO at shipment) (with a warning)

GUI Configurator Item: CFG_ACTION

(17) Interrupt vector type (vector_type)

Description: Defines the type of interrupt handlers to be used and allocation of interrupt vector tables.

Definition Format: Symbol

Specifiable Range: Select from the following. Table 14.4 shows their differences.

- ROM_ONLY_DIRECT
- RAM_ONLY_DIRECT
- ROM
- RAM

Default Setting: Setting in the default cfg file (ROM at shipment) (with a warning)

GUI Configurator Item: CFG_DIRECT, CFG_VCTRAM

Table 14.4 Interrupt Vector Type

	ROM_ONLY_DIRECT	RAM_ONLY_DIRECT	ROM	RAM
Available handlers	• Direct interrupt handler • Direct CPU exception handler		• Direct interrupt handler • Direct CPU exception handler • Normal interrupt handler • Normal CPU exception handler	
def_int, def_exc, and vdef_trp service calls *	Not incorporated	Incorporated	Not incorporated	Incorporated
GUI configurator setting	CFG_DIRECT: On CFG_VCTRAM: Off	CFG_DIRECT: On CFG_VCTRAM: On	CFG_DIRECT: Off CFG_VCTRAM: Off	CFG_DIRECT: Off CFG_VCTRAM: On

Note: * The definitions of these service calls set in service_call are ignored.

(18) Register bank usage (regbank)

Description: Defines the usage of the on-chip register bank in the processor.

Note that depending on the kernel_intspec.h setting, the register bank might not be used for any interrupt regardless of the setting here. For details, refer to section 17.3, Creating CPU Interrupt Specification Definition File (kernel_intspec.h).

In addition, note that the format of direct interrupt handler creation depends on whether the register bank is used.

Definition Format: Symbol

Specifiable Range: See table 14.5.

Table 14.5 regbank Definition

Usage	Definition
The register bank function is not used.	NOTUSE
The register bank is used for all interrupts. *	ALL
Whether to use the register bank is specified for each interrupt level. *	Specify the following symbol according to the interrupt level to use the register bank. To specify multiple levels, separate the symbols with a comma (,). BANKLEVEL01: Interrupt level 1 BANKLEVEL02: Interrupt level 2 BANKLEVEL03: Interrupt level 3 BANKLEVEL04: Interrupt level 4 BANKLEVEL05: Interrupt level 5 BANKLEVEL06: Interrupt level 6 BANKLEVEL07: Interrupt level 7 BANKLEVEL08: Interrupt level 8 BANKLEVEL09: Interrupt level 9 BANKLEVEL10: Interrupt level 10 BANKLEVEL11: Interrupt level 11 BANKLEVEL12: Interrupt level 12 BANKLEVEL13: Interrupt level 13 BANKLEVEL14: Interrupt level 14 BANKLEVEL15: Interrupt level 15

Note: * The register bank is not used for the interrupt sources that are defined not to use the register bank in kernel_intspec.h.

Default Setting: Setting in the default cfg file (ALL at shipment) (with a warning)

GUI Configurator Item: CFG_REGBANK

Remarks: The kernel initializes the necessary registers in the interrupt controller as follows when a vsta_knl call is issued if INTSPEC_IBNR_ADR1 (for CPUID#1) or INTSPEC_IBNR_ADR2 (for CPUID#2) in kernel_intspec.h is not 0.

(1) When system.regbank is set to ALL:
IBNR is initialized to 0x4000.

(2) When system.regbank is set to BANKLEVELxx:
IBNR is initialized to 0xC000, and IBCR is initialized so that the specified levels of interrupts use the register bank.

(3) When system.regbank is set to NOTUSE:
IBNR is initialized to 0.

14.3.3 Defining the Maximum IDs (maxdefine)

This defines the maximum local ID for each kernel object. Each kernel object can use local IDs within the range from 1 to the defined maximum ID.

Definitions can be omitted except for static_task. If a definition is omitted, the minimum value is automatically set (for example, enough value to use all the corresponding objects defined (xxxx[]) in the cfg file).

Even when a definition is not omitted, if the specified value is smaller than the ID or vector number specified in the corresponding object definition (xxxx[ ]) in the cfg file, the maximum ID is automatically increased. In this case, a warning message will be output.

Format

```
maxdefine{
       max_task          = <Setting>; // Maximum local task ID
       max_statictask    = <Setting>; // Maximum local task ID using static stack
       max_sem           = <Setting>; // Maximum local semaphore ID
       max_flag          = <Setting>; // Maximum local event flag ID
       max_dtq           = <Setting>; // Maximum local data queue ID
       max_mbx           = <Setting>; // Maximum local mailbox ID
       max_mtx           = <Setting>; // Maximum local mutex ID
       max_mbf           = <Setting>; // Maximum local message buffer ID
       max_mpf           = <Setting>; // Maximum local fixed-sized memory pool ID
       max_mpl           = <Setting>; // Maximum local variable-sized memory pool
                                         ID
       max_cyh           = <Setting>; // Maximum local cyclic handler ID
       max_alh           = <Setting>; // Maximum local alarm handler ID
       max_fncd          = <Setting>; // Maximum extended service call function
                                         code
       max_int           = <Setting>; // Maximum interrupt vector number
};
```

Contents

(1) Maximum local task ID (max_task)

Description: A value from 1 to max_task can be used for a local task ID.

Definition Format: Numeric value

Specifiable Range: 1 to 1023

Default Setting: Automatically calculated.

GUI Configurator Item: CFG_MAXTSKID

(2) Maximum local task ID using static stack (max_statictask)

Description: In this OS, tasks with local task IDs 1 to max_statictask use the static stack defined as described in section 14.3.12, Static Stack Area Definition (static_stack[]). When this item is set to 0, no task uses the static stack.

Definition Format: Numeric value

Specifiable Range: 0 to 1023

Default Setting: Setting in the default cfg file (0 at shipment) (with a warning)

GUI Configurator Item: CFG_STSTKID

(3) Maximum local semaphore ID (max_sem)

Description: A value from 1 to max_sem can be used for a local semaphore ID. When max_sem is set to 0, no semaphore can be used.

Definition Format: Numeric value

Specifiable Range: 0 to 1023

Default Setting: Automatically calculated.

GUI Configurator Item: CFG_MAXSEMID

Remarks: When ((service_call.cre_sem == NO) && (service_call.icre_sem == NO)), max_sem is assumed to be 0. All semaphore[] settings are ignored.

(4) Maximum local event flag ID (max_flag)

Description: A value from 1 to max_flag can be used for a local event flag ID. When max_flag is set to 0, no event flag can be used.

Definition Format: Numeric value

Specifiable Range: 0 to 1023

Default Setting: Automatically calculated.

GUI Configurator Item: CFG_MAXFLGID

Remarks: When ((service_call.cre_flg == NO) && (service_call.icre_flg == NO)), max_flag is assumed to be 0. All eventflag[] settings are ignored.

(5) Maximum local data queue ID (max_dtq)

Description: A value from 1 to max_dtq can be used for a local data queue ID. When max_dtq is set to 0, no data queue can be used.

Definition Format: Numeric value

Specifiable Range: 0 to 1023

Default Setting: Automatically calculated.

GUI Configurator Item: CFG_MAXDTQID

Remarks: When ((service_call.cre_dtq == NO) && (service_call.icre_dtq == NO)), max_dtq is assumed to be 0. All dataqueue[] settings are ignored. The memdtq definition is also ignored, and the default data queue area is not generated.

(6) Maximum local mailbox ID (max_mbx)

Description: A value from 1 to max_mbx can be used for a local mailbox ID. When max_mbx is set to 0, no mailbox can be used.

Definition Format: Numeric value

Specifiable Range: 0 to 1023

Default Setting: Automatically calculated.

GUI Configurator Item: CFG_MAXMBXID

Remarks: When ((service_call.cre_mbx == NO) && (service_call.icre_mbx == NO)), max_mbx is assumed to be 0. All mailbox[] settings are ignored.

(7) Maximum local mutex ID (max_mtx)

Description: A value from 1 to max_mtx can be used for a local mutex ID. When max_mtx is set to 0, no mutex can be used.

Definition Format: Numeric value

Specifiable Range: 0 to 1023

Default Setting: Automatically calculated.

GUI Configurator Item: CFG_MAXMTXID

Remarks: When (service_call.cre_mtx == NO), max_mtx is assumed to be 0. All mutex[] settings are ignored.

(8) Maximum local message buffer ID (max_mbf)

Description: A value from 1 to max_mbf can be used for a local message buffer ID. When max_mbf is set to 0, no message buffer can be used.

Definition Format: Numeric value

Specifiable Range: 0 to 1023

Default Setting: Automatically calculated.

GUI Configurator Item: CFG_MAXMBFID

Remarks: When ((service_call.cre_mbf == NO) && (service_call.icre_mbf == NO)), max_mbf is assumed to be 0. All message_buffer[] settings are ignored. The memmbf definition is also ignored, and the default message buffer area is not generated.

(9) Maximum local fixed-sized memory pool ID (max_mpf)

Description: A value from 1 to max_mpf can be used for a local fixed-sized memory pool ID. When max_mpf is set to 0, no fixed-sized memory pool can be used.

Definition Format: Numeric value

Specifiable Range: 0 to 1022

Default Setting: Automatically calculated.

GUI Configurator Item: CFG_MAXMPFID

Remarks: When ((service_call.cre_mpf == NO) && (service_call.icre_mpf == NO)), max_mpf is assumed to be 0. All memorypool[] settings are ignored. The memmpf definition is also ignored, and the default fixed-sized memory pool area is not generated.

Note that when remote_svc.num_wait is a positive value, both cre_mpf and icre_mpf are corrected to YES.

(10) Maximum local variable-sized memory pool ID (max_mpl)

Description: A value from 1 to max_mpl can be used for a local variable-sized memory pool ID. When max_mpl is set to 0, no variable-sized memory pool can be used.

Definition Format: Numeric value

Specifiable Range: 0 to 1023

Default Setting: Automatically calculated.

GUI Configurator Item: CFG_MAXMPLID

Remarks: When ((service_call.cre_mpl == NO) && (service_call.icre_mpl == NO)), max_mpl is assumed to be 0. All variable_memorypool[] settings are ignored. The memmpl definition is also ignored, and the default variable-sized memory pool area is not generated.

(11) Maximum local cyclic handler ID (max_cyh)

Description: A value from 1 to max_cyh can be used for a local cyclic handler ID. When max_cyh is set to 0, no cyclic handler can be used.

Definition Format: Numeric value

Specifiable Range: 0 to 14

Default Setting: Automatically calculated.

GUI Configurator Item: CFG_MAXCYCID

Remarks: When ((service_call.cre_cyc == NO) && (service_call.icre_cyc == NO)), max_cyh is assumed to be 0. All cyclic_hand[] settings are ignored.

Note that when system.action is YES, both cre_cyc and icre_cyc are corrected to YES.

(12) Maximum local alarm handler ID (max_alh)

Description: A value from 1 to max_alh can be used for a local alarm handler ID. When max_alh is set to 0, no alarm handler can be used.

Definition Format: Numeric value

Specifiable Range: 0 to 15

Default Setting: Automatically calculated.

GUI Configurator Item: CFG_MAXALMID

Remarks: When ((service_call.cre_alm == NO) && (service_call.icre_alm == NO)), max_alh is assumed to be 0. All alarm_hand[] settings are ignored.

(13) Maximum extended service call function code (max_fncd)

Description: A value from 1 to max_fncd can be used for an extended service call function code. When max_fncd is set to 0, no extended service call can be used.

Definition Format: Numeric value

Specifiable Range: 0 to 1023

Default Setting: Automatically calculated.

GUI Configurator Item: CFG_MAXSVCCD

Remarks: When ((service_call.def_svc == NO) && (service_call.idef_svc == NO)), max_fncd is assumed to be 0. All extend_svc [] settings are ignored.

(14) Maximum interrupt vector number (max_int)

Description: A value from 0 to max_int can be used for an interrupt or exception vector number. Note that this OS does not manage the reset vectors (vector numbers 0 to 3).

Operation is undefined when an interrupt with a vector number greater than max_int occurs while the system is working.

Definition Format: Numeric value

Specifiable Range: 64 to 511

Default Setting: Automatically calculated.

GUI Configurator Item: CFG_MAXVCTNO

14.3.4 Defining the Default Task Stack Area (memstk)

According to this definition, the BC_hitskstk section of all_memsize bytes is generated.

Format

```
memstk{
       all_memsize     = <Setting>;  // Default task stack area size
};
```

Contents

(1) Default task stack area size (all_memsize)

Description: Defines the size of the default task stack area. The specified value is rounded up to a multiple of four. For the calculation of the value to be set, refer to section 18.6.3, Calculation of Default Task Stack Area Size (memstk.all_memsize).

According to this definition, the BC_hitskstk section of all_memsize bytes is generated.

Definition Format: Numeric value

Specifiable Range: 0 to 0x20000000

Default Setting: Automatically calculated according to all task[] definitions as described in section 18.6.3, Calculation of Default Task Stack Area Size (memstk.all_memsize).

GUI Configurator Item: CFG_TSKSTKSZ

14.3.5 Defining the Default Data Queue Area (memdtq)

According to this definition, the BC_hidtq section of all_memsize bytes is generated.

Format

```
memdtq {
       all_memsize      = <Setting>; // Default data queue area size
};
```

Contents

(1) Default data queue area size (all_memsize)

Description: Defines the size of the default data queue area. The specified value is rounded up to a multiple of four.

The required size of the default data queue area is calculated by the following equation.

Σ(TSZ_DTQ(data count) + 0x10) + 0x1C

The Σ term in the equation should be calculated for the data queues that satisfy all of the following conditions.

(1) The data count is not 0.

(2) The data queue address is NULL.

Note that when the result of the Σ term calculation is 0, 0x10 is used instead of it.

According to this definition, the BC_hidtq section of all_memsize bytes is generated.

Definition Format: Numeric value

Specifiable Range: 0 to 0x20000000

Default Setting: Automatically calculated according to all dataqueue[] definitions by the above equation.

GUI Configurator Item: CFG_DTQSZ

14.3.6 Defining the Default Message Buffer Area (memmbf)

According to this definition, the BC_himbf section of all_memsize bytes is generated.

Format

```
 memmbf{
         all_memsize      = <Setting>; // Default message buffer area size
 };
```

Contents

(1) Default message buffer area size (all_memsize)

Description: Defines the size of the default message buffer area. The specified value is rounded up to a multiple of four.

The required size of the default message buffer area is calculated by the following equation.

Σ(message buffer size + 0x10) + 0x1C

The Σ term in the equation should be calculated for the message buffers that satisfy all of the following conditions.

(1) The message buffer size is not 0.

(2) The message buffer address is NULL.

Note that when the result of the Σ term calculation is 0, 0x10 is used instead of it.

According to this definition, the BC_himbf section of all_memsize bytes is generated.

Definition Format: Numeric value

Specifiable Range: 0 to 0x20000000

Default Setting: Automatically calculated according to all message_buffer[] definitions by the above equation.

GUI Configurator Item: CFG_MBFSZ

14.3.7 Defining the Default Fixed-Sized Memory Pool Area (memmpf)

According to this definition, the BC_himpf section of all_memsize bytes is generated.

Format

```
memmpf{
       all_memsize      = <Setting>; // Default fixed-sized memory pool area size
};
```

Contents

(1) Default fixed-sized memory pool area size (all_memsize)

Description: Defines the size of the default fixed-sized memory pool area. The specified value is rounded up to a multiple of four.

The required size of the default fixed-sized memory pool area is calculated by the following equation.

Σ(TSZ_MPF(block count, block size) + 0x10) + 0x1C

The Σ term in the equation should be calculated for the fixed-sized memory pools whose address is not NULL. When the result of the Σ term calculation is 0, 0x10 is used instead of it.

Note that the definition contents of the TSZ_MPF() macro depend on the system.mpfmanage setting.

According to this definition, the BC_himpf section of all_memsize bytes is generated.

Definition Format: Numeric value

Specifiable Range: 0 to 0x20000000

Default Setting: Automatically calculated by the above equation.

GUI Configurator Item: CFG_MPFSZ

14.3.8 Defining the Default Variable-Sized Memory Pool Area (memmpl)

According to this definition, the BC_himpl section of all_memsize bytes is generated.

Format

```
memmpl{
       all_memsize  = <Setting>; // Default variable-sized memory pool area size
};
```

Contents

(1) Default variable-sized memory pool area size (all_memsize)

Description: Defines the size of the default variable-sized memory pool area. The specified value is rounded up to a multiple of four.

The required size of the default variable-sized memory pool area is calculated by the following equation.

Σ(memory pool size + 0x10) + 0x1C

The Σ term in the equation should be calculated for the variable-sized memory pools whose address is not NULL. When the result of the Σ term calculation is 0, 0x10 is used instead of it.

According to this definition, the BC_himpl section of all_memsize bytes is generated.

Definition Format: Numeric value

Specifiable Range: 0 to 0x20000000

Default Setting: Automatically calculated by the above equation.

GUI Configurator Item: CFG_MPLSZ

14.3.9 Defining the System Clock (clock)

This defines the information related to the system clock.

Format

```
clock {
       timer        = <Setting>; // (1) Timer mode
       IPL          = <Setting>; // (2) Timer interrupt level
       number       = <Setting>; // (3) Timer interrupt vector number
       stack_size   = <Setting>; // (4) Timer stack size
};
```

Contents

(1) Timer mode (timer)

Description: Defines whether to use the time management function of the kernel. When using the time management function, link a timer driver with the kernel.

When this item is set to TIMER, the following descriptions are assumed in the cfg file.

```
// Definition of timer driver initialization routine
init_routine[] {
    exinf = 0;
    entry_address = tdr_ini_tmr();
};

// Definition of kernel timer interrupt handler
interrupt_vector[<clock.number setting>] {
    direct = ON;
    regbank = ON;
    entry_address = _kernel_isig_tim();
};
```

Definition Format: Symbol

Specifiable Range: Select from the following.

— TIMER: Uses the time management function of the kernel.

— NOTIMER: Does not use the time management function of the kernel.

Default Setting: Setting in the default cfg file (NOTIMER at shipment) (with a warning)

GUI Configurator Item: CFG_TIMUSE

(2) Timer interrupt level (IPL)

Description: Defines the level of the kernel timer interrupt. When the timer mode is set to NOTIMER, this definition has no meaning.

Specify a value not greater than the kernel interrupt mask level (system.system_IPL); otherwise, an error will result.

Definition Format: Numeric value

Specifiable Range: 1 to 15

Default Setting: Setting in the default cfg file (15 at shipment) (with a warning)

(If system.system_IPL is smaller than 15, an error will result.)

GUI Configurator Item: CFG_TIMINTLVL

Remarks: When clock.timer is set to NOTIMER, this definition has no meaning.

(3) Timer interrupt vector number (number)

Description: Defines the vector number used by the kernel timer.

If the vector number specified here is used for an interrupt_vector[] definition, the configurator will report an error.

Definition Format: Numeric value

Specifiable Range: 64 to 511

Default Setting: This definition must not be omitted (an error will result).

GUI Configurator Item: CFG_TIMINTNO

Remarks: When clock.timer is set to NOTIMER, this definition has no meaning.

(4) Timer stack size (stack_size)

Description: Defines the timer stack size. The specified value is rounded up to a multiple of four. For the calculation of the value to be set, refer to section 18.9, Timer Stack (clock.stack_size).

According to this definition, the BC_hitmrstk section of stack_size bytes is generated.

Definition Format: Numeric value

Specifiable Range: 0 to 0x20000000

Default Setting: Setting in the default cfg file (0x100 at shipment) (with a warning)

GUI Configurator Item: CFG_TMRSTKSZ

Remarks: When clock.timer is set to NOTIMER, this definition has no meaning.

14.3.10 Defining the Remote Service-Call Environment (remote_svc)

Format

```
remote_svc{
       num_server   = <Setting>; // (1) Number of SVC server tasks
       priority     = <Setting>; // (2) Priority of SVC server tasks
       stack_size   = <Setting>; // (3) Stack size used by each SVC server task
       ipi_portid   = <Setting>; // (4) IPI port ID to be used
       num_wait     = <Setting>; // (5) Maximum number of tasks waiting for SVC
                                    servers task
};
```

Contents

(1) Number of SVC server tasks (num_server)

Description: Defines the number of SVC server tasks. This is the number of remote service calls that this kernel can accept from the other CPU at the same time.

When this item is set to 0, no remote service calls from the other CPU can be accepted. Note that remote service call requests can always be issued to the other CPU regardless of this definition.

When a value other than 0 is specified, the service_call settings for the following are corrected to YES.

— acre_tsk and iacre_tsk

— ter_tsk

— del_tsk

— slp_tsk

— wup_tsk and iwup_tsk

Definition Format: Numeric value

Specifiable Range: 0 to 1023

Default Setting: Setting in the default cfg file (1 at shipment) (with a warning)

GUI Configurator Item: CFG_REMOTE_NUMSERVER

(2) Priority of SVC server tasks (priority)

Description: Defines the priority of SVC server tasks that process the remote service-call requests.

Definition Format: Numeric value

Specifiable Range: 1 to 255

Default Setting: Setting in the default cfg file (1 at shipment) (with a warning)

GUI Configurator Item: CFG_REMOTE_PRIORITY

Remarks: When remote_svc.num_server is set to 0, this definition has no meaning.

(3) Stack size used by each SVC server task (stack_size)

Description: Defines the stack size used by each SVC server task. The specified value is rounded up to a multiple of four. For the calculation of the value to be set, refer to section 18.6.4, Stack Size Used by SVC Server Task (remote_svc.stack_size).

According to this definition, the BC_hirmtstk section of stack_size × num_server bytes is generated.

Definition Format: Numeric value

Specifiable Range: 128 to 0x20000000

Default Setting: Setting in the default cfg file (0x200 at shipment) (with a warning)

GUI Configurator Item: CFG_REMOTE_STKSZ

Remarks: When remote_svc.num_server is set to 0, this definition has no meaning.

(4) IPI port ID to be used (ipi_portid)

Description: Defines the IPI port ID to be used to accept remote service calls and return information from remote service calls requested to the other CPU.

The IPI interrupt level is calculated by (15 – port ID), which must not exceed system.system_IPL.

Definition Format: Numeric value

Specifiable Range: 0 to 7

Default Setting: This definition must not be omitted (an error will result).

GUI Configurator Item: CFG_REMOTE_IPI

(5) Maximum number of tasks waiting for SVC server task (num_wait)

Description: When a remote service call is issued, if no SVC server task is available in the target CPU, the calling task enters the WAITING state. num_wait defines the maximum number of tasks waiting for an available SVC server task of the other CPU at the same time.

If all of the following conditions are satisfied when a remote service call is issued, the remote service call immediately returns the EV_NORESOURCE error.

(1) There is no available SVC server task in the target CPU.

(2) The number of tasks waiting for available SVC server tasks has already reached the num_wait count in the calling kernel.

To never wait for available SVC server tasks, set num_wait to 0.

If the specified value exceeds maxdefine.max_task, it is corrected to the max_task value.

If the specified value is not 0, the service_call settings for the following are corrected to YES.

— slp_tsk
— wup_tsk and iwup_tsk
— cre_mpf and icre_mpf
— acre_mpf and iacre_mpf
— del_mpf
— pget_mpf
— rel_mpf

Definition Format: Numeric value

Specifiable Range: 0 to 1023

Default Setting: Setting in the default cfg file (0 at shipment) (with a warning).

GUI Configurator Item: CFG_REMOTE_NUMWAIT

14.3.11 Defining a Task (task[])

This defines (creates) a task.

task[] corresponds to the [Creation of Task] dialog box of the GUI configurator.

Tasks are broadly classified into two types: tasks using the static stacks defined by static_stack[] and those using non-static stacks. These types are distinguished by the local task ID.

To define a task that uses a static stack, the local ID number cannot be omitted.

Table 14.6 shows the task types.

Table 14.6 Task Types

	task[] Definition Items			
Stack Area	**Local ID Number**	**stack_size**	**stack_section**	**stack_address**
Allocated in the default task stack area	system.max_statictask + 1 or greater (when the definition is omitted, a value satisfying this condition is assigned)	Should be specified	Should not be specified	Should not be specified
Stack area generated by the configurator		Should be specified	Should be specified	Should not be specified
Stack area allocated by the user		Should be specified	Should not be specified	Should be specified
Static stack is used	1 to system.max_statictask (definition must not be omitted)	Ignored	Ignored	Ignored

Format

```
task[<Local ID number>]{              // (1) Local ID number
       name              = <Setting>;  // (2) ID name
       export            = <Setting>;  // (3) ID name exporting
       entry_address     = <Setting>;  // (4) Start address of task
       stack_size        = <Setting>;  // (5) Stack size
       stack_section     = <Setting>;  // (6) Section name assigned to stack area
       stack_address     = <Setting>;  // (7) Start address of user-allocated
                                              stack area
       priority          = <Setting>;  // (8) Initial priority of task
       initial_start     = <Setting>;  // (9) Initial state after creation
       exinf             = <Setting>;  // (10) Extended information
       fpu               = <Setting>;  // (11) FPU use in task
       tex_address       = <Setting>;  // (12) Start address of task exception
                                              handling routine
       tex_fpu           = <Setting>;  // (13) FPU use in task exception handling
                                              routine
};
```

Contents

(1) Local ID number

Description: A local ID number must be a value from 1 to 1023.

The local ID number can be omitted; in this case, the configurator automatically assigns a local ID number equal to or greater than maxdefine.max_statictask + 1.

Note that a task with a local ID number from 1 to maxdefine.max_statictask uses a static stack defined as described in section 14.3.12, Static Stack Area Definition (static_stack[]).

Definition Format: Numeric value

Specifiable Range: 1 to 1023

Default Setting: A local ID number equal to or greater than maxdefine.max_statictask + 1 is automatically assigned.

(2) ID name (name)

Description: Defines an ID name for the task. The specified ID name is output to the ID name header file.

Definition Format: Symbol

Specifiable Range: None

Default Setting: If this definition is omitted, the task is handled with no ID name and is not output to the ID name header file.

Note that the ID name should be defined when the local ID number definition is omitted. If the ID name is omitted in this case, an error will be reported.

(3) ID name exporting (export)

Description: Defines whether to export the ID name to the other CPU.

Definition Format: Symbol

Specifiable Range: YES or NO

Default Setting: Setting in the default cfg file (NO at shipment) (without a warning)

Remarks: When the ID name is not defined, this definition has no meaning.

(4) Start address of task (entry_address)

Description: Defines the start address of the task.

Definition Format: External reference name or numeric value

Specifiable Range: An even number in the range from 0 to 0xFFFFFFFF when a numeric value is specified.

Default Setting: This definition must not be omitted (an error will be reported).

(5) Stack size (stack_size)

Description: Defines the stack size used by the task. The specified value is rounded up to a multiple of four. For the calculation of the stack size required for the task, refer to section 18.6.1, Calculation of Stack Size.

Definition Format: Numeric value

Specifiable Range: 128 to 0x20000000

Default Setting: Setting in the default cfg file (0x100 at shipment) (with a warning)

Remarks: When the local ID number is a value from 1 to maxdefine.max_statictask, the task uses a static stack and this definition has no meaning.

(6) Section name assigned to stack area (stack_section)

Description: stack_section should be defined when having the configurator generate the stack area. Refer also to table 14.6.

If both stack_section and stack_address are defined, an error will result.

The actual section name is generated by adding "B" at the beginning of the character string specified in stack_section. The user must allocate this section at an appropriate address at linkage.

Definition Format: Symbol

Specifiable Range: None

Default Setting: See table 14.6.

Remarks: When the local ID number is a value from 1 to maxdefine.max_statictask, the task uses a static stack and this definition has no meaning.

(7) Start address of user-allocated stack area (stack_address)

Description: When using the user-allocated stack area, define the start address of the stack area through stack_address. The area of stack_size bytes starting from stack_address must be allocated by the user.

Definition Format: External reference name or numeric value

Specifiable Range: A multiple of four in the range from 0 to 0xFFFFFFFF when a numeric value is specified.

Default Setting: See table 14.6.

Remarks: When the local ID number is a value from 1 to maxdefine.max_statictask, the task uses a static stack and this definition has no meaning.

(8) Initial priority of task (priority)

Description: Defines the priority of the task at initiation. Specify a value from 1 to 255 not greater than the maximum task priority (system.priority).

Definition Format: Numeric value

Specifiable Range: 1 to 255

Default Setting: Setting in the default cfg file (1 at shipment) (with a warning)

(9) Initial state after creation (initial_start)

Description: Defines whether the initial task state is READY or DORMANT. This item corresponds to the TA_ACT attribute of a task.

Definition Format: Symbol

Specifiable Range: Select from the following.

- ON: Moves the task to the READY state after kernel activation.
- OFF: Moves the task to the DORMANT state after kernel activation.

Default Setting: Setting in the default cfg file (OFF at shipment) (without a warning)

(10) Extended information (exinf)

Description: Defines extended information for the task.

Definition Format: External reference name or numeric value

Specifiable Range: 0 to 0xFFFFFFFF when a numeric value is specified.

Default Setting: Setting in the default cfg file (0 at shipment) (without a warning)

(11) FPU use in task (fpu)

Description: Defines whether to use the FPU in the task; that is, whether to include the FPU registers in task contexts. This item corresponds to the TA_COP1 attribute of a task.

Definition Format: Symbol

Specifiable Range: Select from the following.

- ON: Uses the FPU.
- OFF: Does not use the FPU.

Default Setting: Setting in the default cfg file (OFF at shipment) (with a warning)

(12) Start address of task exception handling routine (tex_address)

Description: Defines the start address of the task exception handling routine. Do not define this item when not defining a task exception handling routine.

Definition Format: External reference name or numeric value

Specifiable Range: An even number in the range from 0 to 0xFFFFFFFF when a numeric value is specified.

Default Setting: No task exception handling routine is defined.

Remarks: When (service_call.def_tex == NO) && (service_call.idef_tex == NO), this definition has no meaning.

(13) FPU use in task exception handling routine (tex_fpu)

Description: Defines whether to use the FPU in the task exception handling routine; that is, whether to include the FPU registers in task exception handling routine contexts. This item corresponds to the TA_COP1 attribute of a task exception handling routine.

Definition Format: Symbol

Specifiable Range: Select from the following.

— ON: Uses the FPU.

— OFF: Does not use the FPU.

Default Setting: Setting in the default cfg file (OFF at shipment) (with a warning)

Remarks: When (service_call.def_tex == NO) && (service_call.idef_tex == NO), this definition has no meaning.

14.3.12 Defining a Static Stack Area (static_stack[])

This defines a static stack area and associates it with the tasks that use the stack area. When maxdefine.max_staticstack > 0, static_stack should be defined.

Multiple static_stack definitions are allowed.

Each local task ID from 1 to maxdefine.max_staticstack must be specified only once through all static_stack[].tskid definitions.

When maxdefine.max_statictask is 0, the static_stack[] definition is ignored; in this case, a warning message will be output.

Format

```
static_stack[<stack number>]{            // (1) Stack number
       tskid = <Setting>(,<Setting>,…); // (2) Local task ID using the stack
       stack_size        = <Setting>;   // (3) Stack size
       stack_section     = <Setting>;   // (4) Section name assigned to stack
                                               area
};
```

Contents

(1) Stack number

Description: Defines the number used by the configurator to distinguish between static_stack[] definitions. A unique stack number should be assigned to each static_stack definition.

Stack numbers do not need to be sequential numbers starting from 1.

Definition Format: Numeric value

Specifiable Range: 1 to 1023

Default Setting: This definition must not be omitted (an error will result).

(2) Local task ID using the stack

Description: Defines the local ID number or ID name of the task that uses this stack area.

A local ID number must be a value from 1 to maxdefine.max_statictask, but it does not need to be defined through task[].

An ID name can be specified only when a task[] is defined for the name, a local ID number is specified in the task[] definition, and the local ID number is a value from 1 to maxdefine.max_statictask.

To assign multiple tasks to the stack area, separate them with a comma (,); in this case, these multiple tasks share the static stack (shared stack function).

Definition Format: Symbol or numeric value

Specifiable Range: A task[].name setting for which a local ID number is defined and that is not greater than maxdefine.max_statictask when a symbol (ID name) is specified.

A value from 1 to maxdefine.max_statictask when a numeric value is specified.

Default Setting: This definition must not be omitted (an error will result).

(3) Stack size (stack_size)

Description: Defines the size of the stack area. The specified value is rounded up to a multiple of four. For the calculation of the stack size required for the task, refer to section 18.6, Task Stack.

Definition Format: Numeric value

Specifiable Range: 128 to 0x20000000

Default Setting: Setting in the default cfg file (0x100 at shipment) (with a warning)

(4) Section name assigned to stack area (stack_section)

Description: Defines a section name to be assigned to the stack area. The configurator generates an uninitialized data section specified in stack_size with a name generated by adding "B" at the beginning of the character string specified in stack_section. The user must allocate this section at an appropriate address at linkage.

Note that when the GUI configurator is used, stack_section is always defined as C_histstk (restriction of GUI configurator).

Definition Format: Symbol

Specifiable Range: None

Condition for Omitting Definition: This definition can always be omitted.

Default Setting: Setting in the default cfg file (C_histstk at shipment) (with a warning)

14.3.13 Defining a Semaphore (semaphore[])

This defines (creates) a semaphore.

semaphore[] corresponds to the [Creation of Semaphore] dialog box of the GUI configurator.

When both service_call.cre_sem and icre_sem are set to NO, the semaphore[] definition is ignored; in this case, a warning message will be output.

Format

```
semaphore[<Local ID number>]{        // (1) Local ID number
       name            = <Setting>;  // (2) ID name
       export          = <Setting>;  // (3) ID name exporting
       max_count       = <Setting>;  // (4) Maximum semaphore counter value
       initial_count   = <Setting>;  // (5) Initial semaphore counter value
       wait_queue      = <Setting>;  // (6) Wait queue attribute
};
```

Contents

(1) Local ID number

Description: A local ID number must be a value from 1 to 1023.

The ID number can be omitted; in this case, the configurator automatically assigns an ID number.

Definition Format: Numeric value

Specifiable Range: 1 to 1023

Default Setting: A local ID number is automatically assigned.

(2) ID name (name)

Description: Defines an ID name for the semaphore. The specified ID name is output to the ID name header file.

Definition Format: Symbol

Specifiable Range: None

Default Setting: If this definition is omitted, the semaphore is handled with no ID name and is not output to the ID name header file.

Note that the ID name should be defined when the local ID number definition is omitted. If the ID name is omitted in this case, an error is reported.

(3) ID name exporting (export)

Description: Defines whether to export the ID name to the other CPU.

Definition Format: Symbol

Specifiable Range: YES or NO

Default Setting: Setting in the default cfg file (NO at shipment) (without a warning)

Remarks: When the ID name is not defined, this definition has no meaning.

(4) Maximum semaphore counter value (max_count)

Description: Defines the maximum value of the semaphore counter.

Definition Format: Numeric value

Specifiable Range: 1 to 65535

Default Setting: Setting in the default cfg file (1 at shipment) (with a warning)

(5) Initial semaphore counter value (initial_count)

Description: Defines the initial value of the semaphore counter. It must not exceed the maximum semaphore counter value.

Definition Format: Numeric value

Specifiable Range: 0 to 65535

Default Setting: Setting in the default cfg file (1 at shipment) (with a warning)

(6) Wait queue attribute (wait_queue)

Description: Defines the wait queue attribute.

Definition Format: Symbol

Specifiable Range: Select from the following.

- — TA_TFIFO: The wait queue is managed on a FIFO basis.
- — TA_TPRI: The wait queue is managed on a task priority basis.

Default Setting: Setting in the default cfg file (TA_TFIFO at shipment) (with a warning)

14.3.14 Defining an Event Flag (flag[])

This defines (creates) an event flag.

flag[] corresponds to the [Creation of Event Flag] dialog box of the GUI configurator.

Both service_call.cre_flg and icre_flg are set to NO, the flag[] definition is ignored; in this case, a warning message will be output.

Format

```
flag[<Local ID number>]{               // (1) Local ID number
       name              = <Setting>; // (2) ID name
       export            = <Setting>; // (3) ID name exporting
       initial_pattern   = <Setting>; // (4) Initial event flag bit pattern
       wait_queue        = <Setting>; // (5) Wait queue attribute
       wait_multi        = <Setting>; // (6) Multiple-wait attribute
       clear_attribute   = <Setting>; // (7) Clear attribute
};
```

Contents

(1) Local ID number

Description: A local ID number must be a value from 1 to 1023.
The local ID number can be omitted; in this case, the configurator automatically assigns a local ID number.

Definition Format: Numeric value

Specifiable Range: 1 to 1023

Default Setting: A local ID number is automatically assigned.

(2) ID name (name)

Description: Defines an ID name for the event flag. The specified ID name is output to the ID name header file.

Definition Format: Symbol

Specifiable Range: None

Default Setting: If this definition is omitted, the event flag is handled with no ID name and is not output to the ID name header file.

Note that the ID name should be defined when the local ID number definition is omitted. If the ID name is omitted in this case, an error is reported.

(3) ID name exporting (export)

Description: Defines whether to export the ID name to the other CPU.

Definition Format: Symbol

Specifiable Range: YES or NO

Default Setting: Setting in the default cfg file (NO at shipment) (without a warning)

Remarks: When the ID name is not defined, this definition has no meaning.

(4) Initial event flag bit pattern (initial_pattern)

Description: Defines the initial bit pattern for the event flag.

Definition Format: Numeric value

Specifiable Range: 0 to 0xFFFFFFFF

Default Setting: Setting in the default cfg file (0 at shipment) (with a warning)

(5) Wait queue attribute (wait_queue)

Description: Defines the wait queue attribute.

Definition Format: Symbol

Specifiable Range: Select from the following.

— TA_TFIFO: The wait queue is managed on a FIFO basis.

— TA_TPRI: The wait queue is managed on a task priority basis.

Default Setting: Setting in the default cfg file (TA_TFIFO at shipment) (with a warning)

(6) Multiple-wait attribute (wait_multi)

Description: Defines whether to permit multiple tasks to wait for the event flag.

Definition Format: Symbol

Specifiable Range: Select from the following.

— TA_WMUL: Permits multiple tasks to wait for the event flag.

— TA_WSGL: Does not permit multiple tasks to wait for the event flag.

Default Setting: Setting in the default cfg file (TA_WMUL at shipment) (with a warning)

(7) Clear attribute (clear_attribute)

Description: Defines the clear attribute of the event flag.

Definition Format: Symbol

Specifiable Range: Select from the following.

— YES: Specifies the clear attribute.

— NO: Does not specify the clear attribute.

Default Setting: Setting in the default cfg file (NO at shipment) (with a warning)

14.3.15 Defining a Data Queue (dataqueue[])

This defines (creates) a data queue.

dataqueue[] corresponds to the [Creation of Data Queue] dialog box of the GUI configurator.

When both service_call.cre_dtq and icre_dtq are set to NO, the dataqueue[] definition is ignored; in this case, a warning message will be output.

Format

```
dataqueue[<Local ID number>]{         // (1) Local ID number
       name            = <Setting>; // (2) ID name
       export          = <Setting>; // (3) ID name exporting
       buffer_size     = <Setting>; // (4) Maximum data count for data queue
       section         = <Setting>; // (5) Section name assigned to data queue
                                         area
       address         = <Setting>; // (6) Start address of data queue area
       wait_queue      = <Setting>; // (7) Wait queue attribute
};
```

Contents

(1) Local ID number

Description: A local ID number must be a value from 1 to 1023.

The local ID number can be omitted; in this case, the configurator automatically assigns a local ID number.

Definition Format: Numeric value

Specifiable Range: 1 to 1023

Default Setting: A local ID number is automatically assigned.

(2) ID name (name)

Description: Defines an ID name for the data queue. The specified ID name is output to the ID name header file.

Definition Format: Symbol

Specifiable Range: None

Default Setting: If this definition is omitted, the data queue is handled with no ID name and is not output to the ID name header file.

Note that the ID name should be defined when the local ID number definition is omitted. If the ID name is omitted in this case, an error is reported.

(3) ID name exporting (export)

Description: Defines whether to export the ID name to the other CPU.

Definition Format: Symbol

Specifiable Range: YES or NO

Default Setting: Setting in the default cfg file (NO at shipment) (without a warning)

Remarks: When the ID name is not defined, this definition has no meaning.

(4) Maximum data count for data queue (buffer_size)

Description: Specifies the maximum number of data items for the data queue. A data queue with the data count set to 0 can also be created.

Definition Format: Numeric value

Specifiable Range: 0 to 0x08000000

Default Setting: Setting in the default cfg file (1 at shipment) (with a warning)

(5) Section name assigned to data queue area (section)

Description: This item should be defined when having the configurator generate the data queue area.

Table 14.7 shows the methods for specifying the data queue area.

Table 14.7 Data Queue Area Specification Methods

Data Queue Area	section	address
Allocated in the default data queue area	Should not be specified	Should not be specified
Data queue area generated by the configurator.	Should be specified	Should not be specified
Data queue area allocated by the user.	Should not be specified	Should be specified

If both section and address are defined, an error will result.

When section is specified, the actual section name is generated by adding "B" at the beginning of the character string specified in section. The user must allocate this section at an appropriate address at linkage.

Definition Format: Symbol

Specifiable Range: None

Default Setting: See table 14.7.

Remarks: When buffer_size is set to 0, this definition has no meaning.

(6) Start address of data queue area (address)

Description: When using the user-allocated data queue area, define the start address of the data queue area here. The area of TSZ_DTQ(buffer_size) bytes starting from address must be allocated by the user.

Refer also to the above item, (5) Section name assigned to data queue area (section).

Definition Format: External reference name or numeric value

Specifiable Range: A multiple of four in the range from 0 to 0xFFFFFFFF when a numeric value is specified.

Default Setting: See table 14.7.

Remarks: When buffer_size is set to 0, this definition has no meaning.

(7) Wait queue attribute (wait_queue)

Description: Defines the send-wait queue attribute. Note that the receive-wait queue is always managed on a FIFO basis.

Definition Format: Symbol

Specifiable Range: Select from the following.

— TA_TFIFO: The wait queue is managed on a FIFO basis.

— TA_TPRI: The wait queue is managed on a task priority basis.

Default Setting: Setting in the default cfg file (TA_TFIFO at shipment) (with a warning)

14.3.16 Defining a Mailbox (mailbox[])

This defines (creates) a mailbox.

mailbox[] corresponds to the [Creation of Mailbox] dialog box of the GUI configurator.

When both service_call.cre_mbx and icre_mbx are set to NO, the mailbox[] definition is ignored; in this case, a warning message will be output.

Format

```
mailbox[<Local ID number>]{          // (1) Local ID number
       name           = <Setting>;  // (2) ID name
       export         = <Setting>;  // (3) ID name exporting
       wait_queue     = <Setting>;  // (4) Wait queue attribute
       message_queue  = <Setting>;  // (5) Message queue attribute
       max_pri        = <Setting>;  // (6) Maximum message priority
};
```

Contents

(1) ID number

Description: A local ID number must be a value from 1 to 1023.

The local ID number can be omitted; in this case, the configurator automatically assigns a local ID number.

Definition Format: Numeric value

Specifiable Range: 1 to 1023

Default Setting: A local ID number is automatically assigned.

(2) ID name (name)

Description: Defines an ID name for the mailbox. The specified ID name is output to the ID name header file.

Definition Format: Symbol

Specifiable Range: None

Default Setting: If this definition is omitted, the mailbox is handled with no ID name and is not output to the ID name header file.

Note that the ID name should be defined when the local ID number definition is omitted. If the ID name is omitted in this case, an error is reported.

(3) ID name exporting (export)

Description: Defines whether to export the ID name to the other CPU.

Definition Format: Symbol

Specifiable Range: YES or NO

Default Setting: Setting in the default cfg file (NO at shipment) (without a warning)

Remarks: When the ID name is not defined, this definition has no meaning.

(4) Wait queue attribute (wait_queue)

Description: Defines the wait queue attribute.

Definition Format: Symbol

Specifiable Range: Select from the following.

- TA_TFIFO: The wait queue is managed on a FIFO basis.
- TA_TPRI: The wait queue is managed on a task priority basis.

Default Setting: Setting in the default cfg file (TA_TFIFO at shipment) (with a warning)

(5) Message queue attribute (message_queue)

Description: Defines the message queue attribute.

Definition Format: Symbol

Specifiable Range: Select from the following.

- TA_MFIFO: Messages are stored on a FIFO basis.
- TA_MPRI: Messages are stored on a message priority basis.

Default Setting: Setting in the default cfg file (TA_MFIFO at shipment) (with a warning)

(6) Maximum message priority (max_pri)

Description:	Be sure to define the maximum message priority here when message_queue is set to TA_MPRI. Specify a value from 1 to 255. It must not exceed the maximum message priority specified in system.message_pri.
Definition Format:	Numeric value
Specifiable Range:	1 to 255
Default Setting:	Setting in the default cfg file (1 at shipment) (with a warning)
Remarks:	When message_queue is set to TA_MFIFO, this definition has no meaning.

14.3.17 Defining a Mutex (mutex[])

This defines (creates) a mutex.

mutex[] corresponds to the [Creation of Mutex] dialog box of the GUI configurator.

When service_call.cre_mtx is set to NO, the mutex[] definition is ignored; in this case, a warning message will be output.

Format

```
mutex[<Local ID number>]{         // (1) Local ID number
       name          = <Setting>; // (2) ID name
       protocol      = <Setting>; // (3) Priority ceiling protocol
       ceil_pri      = <Setting>; // (4) Ceiling priority
};
```

Contents

(1) Local ID number

Description:	A local ID number must be a value from 1 to 1023. The local ID number can be omitted; in this case, the configurator automatically assigns a local ID number.
Definition Format:	Numeric value
Specifiable Range:	1 to 1023
Default Setting:	A local ID number is automatically assigned.

(2) ID name (name)

Description: Defines an ID name for the mutex. The specified ID name is output to the ID name header file.

Definition Format: Symbol

Specifiable Range: None

Default Setting: If this definition is omitted, the mutex is handled with no ID name and is not output to the ID name header file.

Note that the ID name should be defined when the local ID number definition is omitted. If the ID name is omitted in this case, an error is reported.

(3) Priority ceiling protocol (protocol)

Description: Defines the priority ceiling protocol. In this version, only TA_CEILING (priority ceiling protocol) can be specified.

Definition Format: Symbol

Specifiable Range: Select from the following.

— TA_CEILING: Priority ceiling protocol

Default Setting: Setting in the default cfg file (TA_CEILING at shipment) (without a warning)

(4) Ceiling priority (ceil_pri)

Description: Defines the ceiling priority used in the priority ceiling protocol.

Specify a value from 1 to 255. It must not exceed the maximum task priority specified in system.priority.

Definition Format: Numeric value

Specifiable Range: 1 to 255

Default Setting: Setting in the default cfg file (1 at shipment) (with a warning)

14.3.18 Defining a Message Buffer (message_buffer[])

This defines (creates) a message buffer.

message_buffer[] corresponds to the [Creation of Message Buffer] dialog box of the GUI configurator.

When both service_call.cre_mbf and icre_mbf are set to NO, the message_buffer[] definition is ignored; in this case, a warning message will be output.

Format

```
message_buffer[<Local ID number>]{  // (1) Local ID number
       name           = <Setting>;  // (2) ID name
       export         = <Setting>;  // (3) ID name exporting
       buffer_size    = <Setting>;  // (4) Message buffer size
       section        = <Setting>;  // (5) Section name assigned to message
                                         buffer area
       address        = <Setting>;  // (6) Start address of message buffer area
       max_msgsz      = <Setting>;  // (7) Maximum message size
       wait_queue     = <Setting>;  // (8) Wait queue attribute
};
```

Contents

(1) Local ID number

Description: A local ID number must be a value from 1 to 1023.

The local ID number can be omitted; in this case, the configurator automatically assigns a local ID number.

Definition Format: Numeric value

Specifiable Range: 1 to 1023

Default Setting: A local ID number is automatically assigned.

(2) ID name (name)

Description: Defines an ID name for the message buffer. The specified ID name is output to the ID name header file.

Definition Format: Symbol

Specifiable Range: None

Default Setting: If this definition is omitted, the message buffer is handled with no ID name and is not output to the ID name header file.

Note that the ID name should be defined when the local ID number definition is omitted. If the ID name is omitted in this case, an error is reported.

(3) ID name exporting (export)

Description: Defines whether to export the ID name to the other CPU.

Definition Format: Symbol

Specifiable Range: YES or NO

Default Setting: Setting in the default cfg file (NO at shipment) (without a warning)

Remarks: When the ID name is not defined, this definition has no meaning.

(4) Message buffer size (buffer_size)

Description: Defines the message buffer size in bytes. The specified value is rounded up to a multiple of four.

A message buffer with the size set to 0 can also be created.

Definition Format: Numeric value

Specifiable Range: 0 or 8 to 0x20000000

Default Setting: Setting in the default cfg file (32 at shipment) (with a warning)

(5) Section name assigned to message buffer area (section)

Description: This item should be defined when having the configurator generate the message buffer area.

Table 14.8 shows the methods for specifying the message buffer area.

Table 14.8 Message Buffer Area Specification Methods

Message Buffer Area	section	address
Allocated in the default message buffer area	Should not be specified	Should not be specified
Message buffer area generated by the configurator	Should be specified	Should not be specified
Message buffer area allocated by the user	Should not be specified	Should be specified

If both section and address are defined, an error will result.

When section is specified, the actual section name is generated by adding "B" at the beginning of the character string specified in section. The user must allocate this section at an appropriate address at linkage.

Definition Format: Symbol

Specifiable Range: None

Default Setting: See table 14.8.

Remarks: When buffer_size is set to 0, this definition has no meaning.

(6) Start address of message buffer area (address)

Description: When using the user-allocated message buffer area, define the start address of the message buffer area here. The area of buffer_size bytes starting from address must be allocated by the user.

Refer also to the above item, (5) Section name assigned to message buffer area (section).

Definition Format: External reference name or numeric value

Specifiable Range: A multiple of four in the range from 0 to 0xFFFFFFFF when a numeric value is specified.

Default Setting: See table 14.8.

Remarks: When buffer_size is set to 0, this definition has no meaning.

(7) Maximum message size (max_msgsz)

Description: Defines the maximum message size in bytes. The specified value is rounded up to a multiple of four. When buffer_size > 0, max_msgsz must not exceed (buffer_size - 4).

Definition Format: Numeric value

Specifiable Range: 4 to 0x20000000

Default Setting: Setting in the default cfg file (4 at shipment) (with a warning)

(8) Wait queue attribute (wait_queue)

Description:	Defines the send-wait queue attribute. Note that the receive-wait queue is always managed on a FIFO basis.
Definition Format:	Symbol
Specifiable Range:	Select from the following. — TA_TFIFO: The wait queue is managed on a FIFO basis. — TA_TPRI: The wait queue is managed on a task priority basis.
Default Setting:	Setting in the default cfg file (TA_TFIFO at shipment) (with a warning)

14.3.19 Defining a Fixed-Sized Memory Pool (memorypool[])

This defines (creates) a fixed-sized memory pool.

memorypool[] corresponds to the [Creation of Fixed-size Memory Pool] dialog box of the GUI configurator.

When system.mpfmanage is set to OUT, the fixed-sized memory block management area (mpfmb in the T_CMPF structure) is necessary and is automatically generated. The section name of this area is BC_hicfg.

When both service_call.cre_mpf and icre_mpf are set to NO, the memorypool[] definition is ignored; in this case, a warning message will be output. When remote_svc.num_server > 0, both service_call.cre_mpf and icre_mpf are corrected to YES.

Format

```
memorypool[<Local ID number>]{       // (1) Local ID number
       name           = <Setting>;  // (2) ID name
       export         = <Setting>;  // (3) ID name exporting
       section        = <Setting>;  // (4) Section name assigned to pool area
       address        = <Setting>;  // (5) Start address of pool area
       num_block      = <Setting>;  // (6) Block count
       siz_block      = <Setting>;  // (7) Block size
       wait_queue     = <Setting>;  // (8) Wait queue attribute
};
```

Contents

(1) Local ID number

Description: A local ID number must be a value from 1 to 1022.

The local ID number can be omitted; in this case, the configurator automatically assigns a local ID number.

Definition Format: Numeric value

Specifiable Range: 1 to 1022 (the maximum value is 1023 in the kernel specification but it is 1022 in the cfg file)

Default Setting: A local ID number is automatically assigned.

(2) ID name (name)

Description: Defines an ID name for the fixed-sized memory pool. The specified ID name is output to the ID name header file.

Definition Format: Symbol

Specifiable Range: None

Default Setting: If this definition is omitted, the memory pool is handled with no ID name and is not output to the ID name header file.

Note that the ID name should be defined when the local ID number definition is omitted. If the ID name is omitted in this case, an error is reported.

(3) ID name exporting (export)

Description: Defines whether to export the ID name to the other CPU.

Definition Format: Symbol

Specifiable Range: YES or NO

Default Setting: Setting in the default cfg file (NO at shipment) (without a warning)

Remarks: When the ID name is not defined, this definition has no meaning.

(4) Section name assigned to pool area (section)

Description: This item should be defined when having the configurator generate the pool area.

Table 14.9 shows the methods for specifying the pool area.

Table 14.9 Fixed-Sized Memory Pool Area Specification Methods

Fixed-Sized Memory Pool Area	section	address
Allocated in the default fixed-sized memory pool area	Should not be specified	Should not be specified
Pool area generated by the configurator	Should be specified	Should not be specified
Pool area allocated by the user	Should not be specified	Should be specified

If both section and address are defined, an error will result.

When section is specified, the actual section name is generated by adding "B" at the beginning of the character string specified in section. The user must allocate this section at an appropriate address at linkage.

Definition Format: Symbol

Specifiable Range: None

Default Setting: See table 14.9.

(5) Start address of pool area (address)

Description: When using the user-allocated pool area, define the start address of the pool area here. The area of TSZ_MPF(num_block, siz_block) bytes starting from address must be allocated by the user.

Refer also to the above item, (4) Section name assigned to pool area (section).

Definition Format: External reference name or numeric value

Specifiable Range: A multiple of four in the range from 0 to 0xFFFFFFFF when a numeric value is specified.

Default Setting: See table 14.9.

(6) Block count (num_block)

Description: Defines the number of blocks in the memory pool.

Definition Format: Numeric value

Specifiable Range: 1 to 0x08000000

Default Setting: Setting in the default cfg file (4 at shipment) (with a warning)

(7) Block size (siz_block)

Description: Defines the block size in bytes. The specified value is rounded up to a multiple of four.

Definition Format: Numeric value

Specifiable Range: 4 to 0x20000000

Default Setting: Setting in the default cfg file (4 at shipment) (with a warning)

(8) Wait queue attribute (wait_queue)

Description: Defines the wait queue attribute.

Definition Format: Symbol

Specifiable Range: Select from the following.

- TA_TFIFO: The wait queue is managed on a FIFO basis.
- TA_TPRI: The wait queue is managed on a task priority basis.

Default Setting: Setting in the default cfg file (TA_TFIFO at shipment) (with a warning)

14.3.20 Defining a Variable-Sized Memory Pool (variable_memorypool[])

This defines (creates) a variable-sized memory pool.

variable_memorypool[] corresponds to the [Creation of Variable-size Memory Pool] dialog box of the GUI configurator.

When both service_call.cre_mpl and icre_mpl are set to NO, the variable_memorypool[] definition is ignored; in this case, a warning message will be output.

Format

```
variable_memorypool[<Local ID number>]{ // (1) Local ID number
       name           = <Setting>;      // (2) ID name
       export         = <Setting>;      // (3) ID name exporting
       heap_size      = <Setting>;      // (4) Memory pool size
       wait_queue     = <Setting>;      // (5) Wait queue attribute
       mpl_section    = <Setting>;      // (6) Section name assigned to pool area
       mpl_address    = <Setting>;      // (7) Start address of pool area
       unfragment     = <Setting>;      // (8) Fragmentation reduction
       min_blksz      = <Setting>;      // (9) Minimum block size
       num_sector     = <Setting>;      // (10) Sector count
};
```

Contents

(1) Local ID number

Description:	A local ID number must be a value from 1 to 1023. The local ID number can be omitted; in this case, the configurator automatically assigns a local ID number.
Definition Format:	Numeric value
Specifiable Range:	1 to 1023
Default Setting:	A local ID number is automatically assigned.

(2) ID name (name)

Description: Defines an ID name for the variable-sized memory pool. The specified ID name is output to the ID name header file.

Definition Format: Symbol

Specifiable Range: None

Default Setting: If this definition is omitted, the memory pool is handled with no ID name and is not output to the ID name header file.

Note that the ID name should be defined when the local ID number definition is omitted. If the ID name is omitted in this case, an error is reported.

(3) ID name exporting (export)

Description: Defines whether to export the ID name to the other CPU.

Definition Format: Symbol

Specifiable Range: YES or NO

Default Setting: Setting in the default cfg file (NO at shipment) (without a warning)

Remarks: When the ID name is not defined, this definition has no meaning.

(4) Memory pool size (heap_size)

Description: Defines the memory pool size in bytes. The specified value is rounded up to a multiple of four.

Definition Format: Numeric value

Specifiable Range: 36 to 0x20000000

Default Setting: Setting in the default cfg file (0x200 at shipment) (with a warning)

(5) Wait queue attribute (wait_queue)

Description: Defines the wait queue attribute. In this version, only TA_TFIFO can be specified.

Definition Format: Symbol

Specifiable Range: Select from the following.

— TA_TFIFO: The wait queue is managed on a FIFO basis.

Default Setting: Setting in the default cfg file (TA_TFIFO at shipment) (without a warning)

(6) Section name assigned to pool area (mpl_section)

Description: This item should be defined when having the configurator generate the pool area.

Table 14.10 shows the methods for specifying the pool area.

Table 14.10 Variable-Sized Memory Pool Area Specification Methods

Variable-Sized Memory Pool Area	mpl_section	mpl_address
Allocated in the default variable-sized memory pool area	Should not be specified	Should not be specified
Pool area generated by the configurator	Should be specified	Should not be specified
Pool area allocated by the user	Should not be specified	Should be specified

If both mpl_section and mpl_address are defined, an error will result.

When mpl_section is specified, the actual section name is generated by adding "B" at the beginning of the character string specified in mpl_section. The user must allocate this section at an appropriate address at linkage.

Definition Format: Symbol

Specifiable Range: None

Default Setting: See table 14.10.

(7) Start address of pool area (mpl_address)

Description: When using the user-allocated pool area, define the start address of the pool area here. The area of heap_size bytes starting from mpl_address must be allocated by the user.

Refer also to the above item, (6) Section name assigned to pool area (mpl_section).

Definition Format: External reference name or numeric value

Specifiable Range: A multiple of four in the range from 0 to 0xFFFFFFFF when a numeric value is specified.

Default Setting: See table 14.10

(8) Fragmentation reduction (unfragment)

Description: When system.newmpl is set to NEW, the VTA_UNFRAGMENT attribute can be specified to reduce fragmentation.

Definition Format: Symbol

Specifiable Range: Select from the following.

— ON: Specifies the VTA_UNFRAGMENT attribute.

— OFF: Does not specify the VTA_UNFRAGMENT attribute.

Default Setting: Setting in the default cfg file (OFF at shipment) (with a warning)

Remarks: When system.newmpl is set to PAST, this definition has no meaning.

(9) Minimum block size (min_blksz)

Description: When the VTA_UNFRAGMENT attribute is specified, the minimum block size should be specified. The specified value is rounded up to a multiple of four.

The minimum block size must not exceed (heap_size / 32).

Definition Format: Numeric value

Specifiable Range: 4 to 0x20000000

Default Setting: Setting in the default cfg file (4 at shipment) (with a warning)

Remarks: When system.newmpl is set to PAST or unfragment is set to OFF, this definition has no meaning.

(10) Sector count (num_sector)

Description: When the VTA_UNFRAGMENT attribute is specified, the sector count should be specified. When the specified value is greater than (heap_size / (min_blksz × 32)), the value is corrected to this calculation result.

Definition Format: Numeric value

Specifiable Range: 1 to 0x 400000

Default Setting: Setting in the default cfg file (1 at shipment) (with a warning)

Remarks: When system.newmpl is set to PAST or unfragment is set to OFF, this definition has no meaning.

14.3.21 Defining a Cyclic Handler (cyclic_hand[])

This defines (creates) a cyclic handler.

cyclic_hand[] corresponds to the [Creation of Cyclic Handler] dialog box of the GUI configurator.

When both service_call.cre_cyc and icre_cyc are set to NO, the cyclic_hand [] definition is ignored; in this case, a warning message will be output. When system.action is set to YES, both service_call.cre_mpf and icre_mpf are corrected to YES.

Format

```
cyclic_hand[<Local ID number>]{       // (1) Local ID number
       name             = <Setting>; // (2) ID name
       export           = <Setting>; // (3) ID name exporting
       interval_counter = <Setting>; // (4) Initiation cycle
       start            = <Setting>; // (5) Cyclic handler state
       phsatr           = <Setting>; // (6) Preserving initiation phase
       phs_counter      = <Setting>; // (7) Initiation phase
       entry_address    = <Setting>; // (8) Start address of handler
       exinf            = <Setting>; // (9) Extended information
};
```

Contents

(1) Local ID number

Description: A local ID number must be a value from 1 to 14.

The local ID number can be omitted; in this case, the configurator automatically assigns a local ID number.

Definition Format: Numeric value

Specifiable Range: 1 to 14

Default Setting: A local ID number is automatically assigned.

(2) ID name (name)

Description: Defines an ID name for the cyclic handler. The specified ID name is output to the ID name header file.

Definition Format: Symbol

Specifiable Range: None

Default Setting: If this definition is omitted, the cyclic handler is handled with no ID name and is not output to the ID name header file.

Note that the ID name should be defined when the local ID number definition is omitted. If the ID name is omitted in this case, an error is reported.

(3) ID name exporting (export)

Description: Defines whether to export the ID name to the other CPU.

Definition Format: Symbol

Specifiable Range: YES or NO

Default Setting: Setting in the default cfg file (NO at shipment) (without a warning)

Remarks: When the ID name is not defined, this definition has no meaning.

(4) Initiation cycle (interval_counter)

Description: Defines the initiation cycle in ms.

Specifiable Range: 1 to 0x7FFFFFFF

Default Setting: Setting in the default cfg file (1 at shipment) (with a warning)

(5) Cyclic handler state (start)

Description: Defines the attribute regarding the cyclic handler state.

Definition Format: Symbol

Specifiable Range: Select from the following.

- ON: Starts the cyclic handler operation (with the TA_STA attribute)
- OFF: Does not start the cyclic handler (without the TA_STA attribute)

Default Setting: Setting in the default cfg file (OFF at shipment) (with a warning)

(6) Preserving initiation phase (phsatr)

Description: Defines the attribute regarding the function for preserving the initiation phase.

Definition Format: Symbol

Specifiable Range: Select from the following.

- ON: Preserves the initiation phase (with the TA_PHS attribute)
- OFF: Does not preserve the initiation phase (without the TA_PHS attribute)

Default Setting: Setting in the default cfg file (OFF at shipment) (with a warning)

(7) Initiation phase (phs_counter)

Description: Defines the initiation phase in ms. The initiation phase must not exceed the initiation cycle.

Definition Format: Numeric value

Specifiable Range: 0 to 0x7FFFFFFF

Default Setting: Setting in the default cfg file (0 at shipment) (with a warning)

(8) Start address of cyclic handler (entry_address)

Description: Specifies the start address of cyclic handler.

Definition Format: External reference name or numeric value

Specifiable Range: An even number in the range from 0 to 0xFFFFFFFF when a numeric value is specified.

Default Setting: This definition must not be omitted (an error will result).

(9) Extended information (exinf)

Description: Defines the extended information of the cyclic handler.

Definition Format: External reference name or numeric value

Specifiable Range: 0 to 0xFFFFFFFF when a numeric value is specified; no limitation on the range when an external reference name is selected.

Default Setting: Setting in the default cfg file (0 at shipment) (without a warning)

14.3.22 Defining an Alarm Handler (alarm_hand[])

This defines (creates) an alarm handler.

alarm_hand[] corresponds to the [Creation of Alarm Handler] dialog box of the GUI configurator.

When both service_call.cre_cyc and icre_alm are set to NO, the alarm_hand[] definition is ignored; in this case, a warning message will be output.

Format

```
alarm_hand[<Local ID number>]{       // (1) Local ID number
       name            = <Setting>;  // (2) ID name
       export          = <Setting>;  // (3) ID name exporting
       entry_address   = <Setting>;  // (4) Start address of handler
       exinf           = <Setting>;  // (5) Extended information
};
```

Contents

(1) Local ID number

Description: A local ID number must be a value from 1 to 15.

The local ID number can be omitted; in this case, the configurator automatically assigns a local ID number.

Definition Format: Numeric value

Specifiable Range: 1 to 15

Default Setting: A local ID number is automatically assigned.

(2) ID name (name)

Description: Defines an ID name for the alarm handler. The specified ID name is output to the ID name header file.

Definition Format: Symbol

Specifiable Range: None

Default Setting: If this definition is omitted, the alarm handler is handled with no ID name and is not output to the ID name header file.

Note that the ID name should be defined when the local ID number definition is omitted. If the ID name is omitted in this case, an error is reported.

(3) ID name exporting (export)

Description: Defines whether to export the ID name to the other CPU.

Definition Format: Symbol

Specifiable Range: YES or NO

Default Setting: Setting in the default cfg file (NO at shipment) (without a warning)

Remarks: When the ID name is not defined, this definition has no meaning.

(4) Start address of alarm handler (entry_address)

Description: Specifies the start address of alarm handler execution.

Definition Format: External reference name or numeric value

Specifiable Range: An even number in the range from 0 to 0xFFFFFFFF when a numeric value is specified.

Default Setting: This definition must not be omitted (an error will result).

(5) Extended information (exinf)

Description: Defines the extended information of the alarm handler.

Definition Format: External reference name or numeric value

Specifiable Range: 0 to 0xFFFFFFFF when a numeric value is specified; no limitation on the range when an external reference name is selected.

Default Setting: Setting in the default cfg file (0 at shipment) (without a warning)

14.3.23 Defining an Overrun Handler (overrun_hand)

This defines an overrun handler.

overrun_hand corresponds to the [Overrun Handler] page of the GUI configurator.

Only one overrun handler can be defined in a system. Therefore, the overrun_hand definition can be done only once, unlike the other objects.

When service_call.def_ovr is set to NO, the overrun_hand definition is ignored; in this case, a warning message will be output.

Format

```
overrun_hand{
       entry_address    = <Setting>; // Start address of overrun handler
};
```

Contents

(1) Start address of overrun handler (entry_address)

Description: Specifies the start address of overrun handler.

Definition Format: External reference name or numeric value

Specifiable Range: An even number in the range from 0 to 0xFFFFFFFF when a numeric value is specified.

Default Setting: This definition must not be omitted (an error will result).

14.3.24 Defining an Extended Service Call Routine (extend_svc[])

This defines an extended service call routine.

extend_svc[] corresponds to the [Definition of Extended Service Call] dialog box of the GUI configurator.

When both service_call.def_svc and idef_svc are set to NO, the extend_svc[] definition is ignored; in this case, a warning message will be output.

Format

```
extend_svc[<function code >]{          // (1) Function code
        entry_address    = <Setting>; // (2) Start address of extended service
                                          call routine
};
```

Contents

(1) Function code

Description: A function code must be a value from 1 to 1023.

Definition Format: Numeric value

Specifiable Range: 1 to 1023

Default Setting: This definition must not be omitted (an error will result).

(2) Start address of extended service call routine (entry_address)

Description: Defines the start address of extended service call routine.

Definition Format: External reference name or numeric value

Specifiable Range: An even number in the range from 0 to 0xFFFFFFFF when a numeric value is specified.

Default Setting: This definition must not be omitted (an error will result).

14.3.25 Defining an Interrupt Handler or a CPU Exception Handler (interrupt_vector[])

This defines an interrupt handler or a CPU exception handler.

A vector number must not be omitted and the same number must not be specified multiple times.

Vector numbers 0 to 3 are reset vectors, which cannot be defined in the configurator. Vector numbers 60 to 63 are reserved for OS use and cannot be defined.

No handler can be defined with a vector number equal to the timer interrupt number (clock.number); if attempted, an error will result.

interrupt_vector[] corresponds to the [Definition of Interrupt/CPU Exception Handler] dialog box of the GUI configurator.

Format

```
interrupt_vector[<vector number>]{   // (1) Vector number
       entry_address    = <Setting>; // (2) Start address of handler
       direct           = <Setting>; // (3) Direct attribute
       regbank          = <Setting>; // (4) Register bank attribute
};
```

Contents

(1) Vector number

Description: Defines a vector number. Specify a value from 4 to 59 or 64 to 511.

Definition Format: Numeric value

Specifiable Range: 4 to 59 or 64 to 511

Default Setting: This definition must not be omitted (an error will result).

(2) Start address of handler (entry_address)

Description: Defines the start address of handler.

Definition Format: External reference name or numeric value

Specifiable Range: An even number in the range from 0 to 0xFFFFFFFF when a numeric value is specified

Default Setting: This definition must not be omitted (an error will result).

(3) Direct attribute (direct)

Description: Defines whether to add the VTA_DIRECT attribute.

For an interrupt handler (including the NMI handler) having an interrupt level higher than system.system_IPL, be sure to add the VTA_DIRECT attribute.

Definition Format: Symbol

Specifiable Range: Select from the following.

- YES: Adds the VTA_DIRECT attribute.
- NO: Does not add the VTA_DIRECT attribute.

Default Setting: Setting in the default cfg file (YES at shipment) (with a warning)

Remarks: When system.vector_type is set to ROM_ONLY_DIRECT or RAM_ONLY_DIRECT while direct = NO, the handler is not defined; in this case, a warning message will be output.

(4) Register bank attribute (regbank)

Description: Defines whether to add the register bank attribute (VTA_REGBANK) for the interrupt handler without the direct attribute (VTA_DIRECT).

The register bank attribute is valid only when all of the following conditions are satisfied. When these conditions are not satisfied, the register bank attribute setting has no meaning and a warning message will not be output in this case.

(a) direct is set to OFF.

(b) system.regbank is set to BANKLEVELxx.

(c) The value specified for INTSPEC_IBNR_ADR1 (for CPUID#1) or INTSPEC_IBNR_ADR2 (for CPUID#2) in the CPU interrupt specification definition file (kernel_intspec.h) is not 0. (That is, the CPU supporting the register bank is used.)

(d) The specified vector number is not a number that corresponds to INTSPEC_NOBANK_VECxxx defined in the CPU interrupt specification definition file (kernel_intspec.h). (That is, the specified vector number is for an interrupt source that is allowed to use the register bank in the CPU specifications.)

When these conditions are satisfied, VTA_REGBANK should be appropriately specified as follows according to the interrupt level of the target interrupt handler. If this attribute is not specified appropriately, the interrupt handler will not operate correctly.

(i) When system.regbank is set to BANKLEVELxx that corresponds to the interrupt level of the target interrupt handler:

Specify regbank = YES.

(ii) When system.regbank is not set to BANKLEVELxx that corresponds to the interrupt level of the target interrupt handler:

Specify regbank = NO.

Definition Format: Symbol

Specifiable Range: Select from the following.

- YES: Adds the VTA_REGBANK attribute.
- NO: Does not add the VTA_REGBANK attribute.

Default Setting: Setting in the default cfg file (NO at shipment) (with a warning)

14.3.26 Defining an Initialization Routine (init_routine[])

This registers (defines) an initialization routine.

Multiple initialization routines can be registered in a system.

init_routine[] corresponds to the [Registration of Initialization Routine] dialog box of the GUI configurator.

When the kernel is started, the initialization routines are executed in the order of appearance in the cfg file.

Format

```
init_routine[] {
       entry_address    = <Setting>; // (1) Start address of initialization
                                        routine
       exinf            = <Setting>; // (2) Extended information
};
```

Contents

(1) Start address of initialization routine (entry_address)

Description: Defines the start address of initialization routine.

Definition Format: External reference name or numeric value

Specifiable Range: An even number in the range from 0 to 0xFFFFFFFF when a numeric value is specified

Default Setting: This definition must not be omitted (an error will be reported).

(2) Extended information (exinf)

Description: Defines the extended information of the initialization routine. The initialization routine receives the extended information specified here as a parameter.

Definition Format: External reference name or numeric value

Specifiable Range: 0 to 0xFFFFFFFF when a numeric value is specified; no limitation on the range when an external reference name is selected.

Default Setting: Setting in the default cfg file (0 at shipment) (without a warning)

14.3.27 Defining Service Calls (service_call)

This defines the service calls to be incorporated in the target.

service_call corresponds to the [Service Calls Selection] page of the GUI configurator.

Format

```
service_call {
   <service call name>= <Setting>;
      …
};
```

To use a kernel object, the corresponding cre_xxx or def_xxx service call should be incorporated in principle. Table 14.11 shows the service calls necessary for each object.

Table 14.11 Service Calls Necessary for Each Object

Object	Necessary Service Call	When Necessary Service Call is not Installed
Task	—	Always available
Task exception handling routine	def_tex or idef_tex	All task[].tex_address and task[].tex_fpu are ignored.
Semaphore	cre_sem or icre_sem	All semaphore[] definitions are ignored and maxdefine.max_sem is assumed as 0.
Event flag	cre_flg or icre_flg	All flag[] definitions are ignored and maxdefine.max_flag is assumed as 0.
Data queue	cre_dtq or icre_dtq	All dataqueue[] definitions are ignored and maxdefine.max_dtq is assumed as 0. The default data queue area is not generated.
Mailbox	cre_mbx or icre_mbx	All mailbox[] definitions are ignored and maxdefine.max_mbx is assumed as 0.
Mutex	cre_mtx	All mutex[] definitions are ignored and maxdefine.max_mtx is assumed as 0.
Message buffer	cre_mbf or icre_mbf	All message_buffer[] definitions are ignored and maxdefine.max_mbf is assumed as 0. The default message buffer area is not generated.
Fixed-sized memory pool	cre_mpf or icre_mpf	All memorypool[] definitions are ignored and maxdefine.max_mpf is assumed as 0. The default fixed-sized memory pool area is not generated.
Variable-sized memory pool	cre_mpl or icre_mpl	All variable_memorypool[] definitions are ignored and maxdefine.max_mpl is assumed as 0. The default variable-sized memory pool area is not generated.
Cyclic handler	cre_cyc or icre_cyc	All cyclic_hand[] definitions are ignored and maxdefine.max_cyh is assumed as 0.
Alarm handler	cre_alm or icre_alm	All alarm_hand[] definitions are ignored and maxdefine.max_alh is assumed as 0.
Overrun handler	def_ovr	The overrun_hand definition is ignored.
Extended service call	def_svc or idef_svc	All extend_svc[] definitions are ignored and maxdefine.max_fncd is assumed as 0.
Interrupt handler or CPU exception handler	—	Always available.

For the service calls shown in table 14.12, definitions are corrected; in this case, a warning message will be output.

Table 14.12 Service Calls whose Definitions are Corrected

Service Call	Condition of Correction
cre_tsk, icre_tsk, ext_tsk, slp_tsk, wup_tsk, iwup_tsk, dis_dsp, ena_dsp, sns_dpn, vsta_knl, ivsta_knl, vini_rmt, vsys_dwn, ivsys_dwn	Always corrected to YES. *
vscr_tsk, ivscr_tsk	Corrected to YES when maxdefine.max_statictask > 0; otherwise, corrected to NO. *
def_inh, idef_inh, def_exc, idef_exc, vdef_trp, ivdef_trp	Corrected to NO when system.vector_type is set to ROM or ROM_ONLY_DIRECT; otherwise, YES. *
cre_cyc, icre_cyc	Corrected to YES when system.action is set to YES.
acre_tsk, iacre_tsk, del_tsk, ter_tsk	Corrected to YES when remote_svc.num_server > 0.
cre_mpf, icre_mpf, acre_mpf, iacre_mpf, del_mpf, pget_mpf, rel_mpf,	Corrected to YES when remote_svc.num_wait > 0.

Note: * As a result of this correction, the settings in the user-specified cfg file and default cfg file are ignored.

When clock.timer is set to NOTIMER, the definitions of the service calls that require timer operation, such as dly_tsk or cre_alm, are automatically corrected to NO. In this case, note that no warning message will be output.

Contents

(1) Setting

Description: Defines whether to incorporate each service call in the system.

If a service call that is not incorporated is issued, an E_NOSPT error will be returned.

Definition Format: Symbol

Specifiable Range: Select from the following.

— YES: Incorporates the service call.

— NO: Does not incorporate the service call

Default Setting: Setting in the default cfg file (all set to NO at shipment) (without a warning)

14.4 Configurator Execution

14.4.1 Overview

Figure 14.1 gives an overview of the configurator operation.

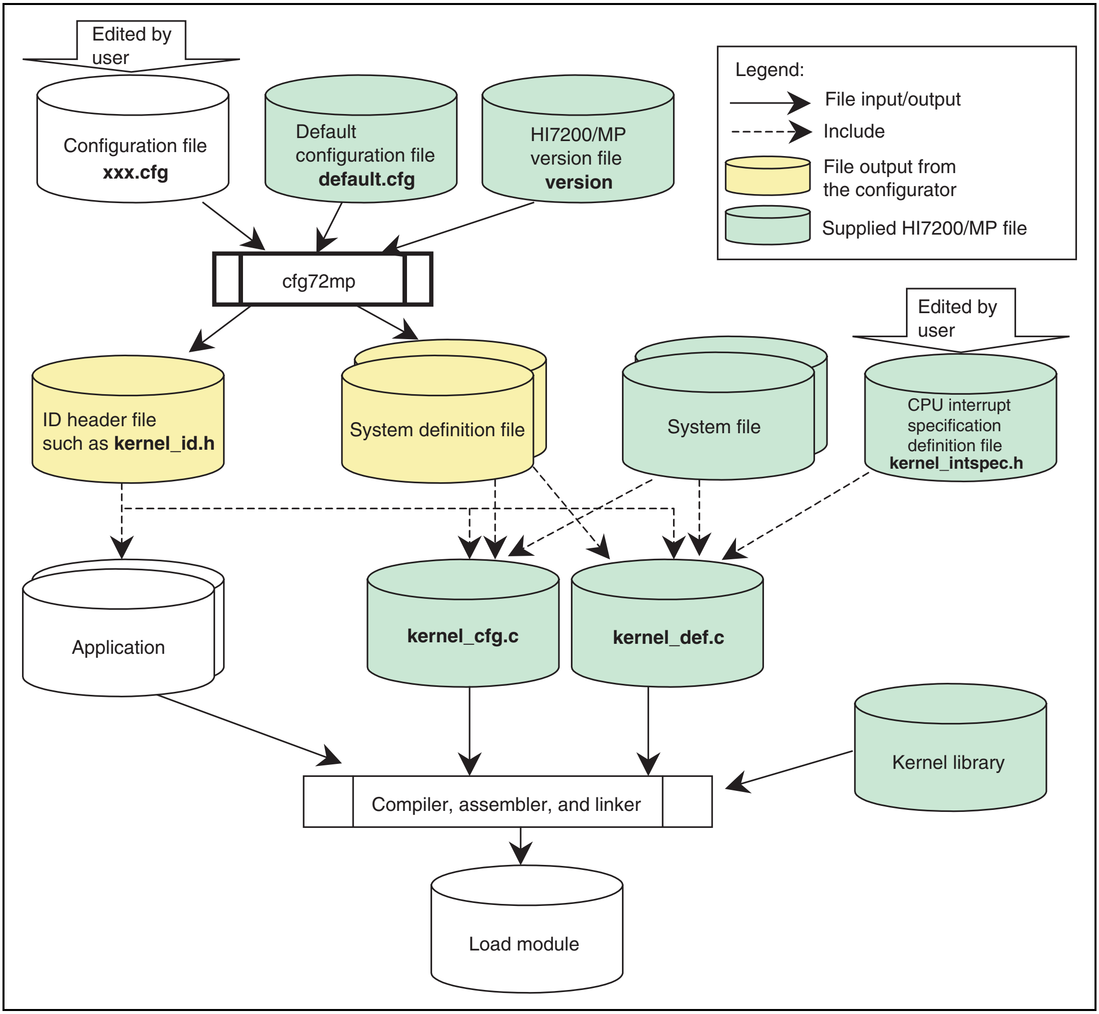


Figure 14.1 Overview of Configurator Operation

For details of the CPU interrupt specification definition file, refer to section 17.3, Creating CPU Interrupt Specification Definition File (kernel_intspec.h).

14.4.2 Environment Setting

The following environment variable should be set appropriately.

In the sample High-performance Embedded Workshop workspace provided, cfg72mp is registered as a custom build phase, in which the environment variable is set.

- LIB72MP

  Path to the default.cfg and version files

14.4.3 Files Required to Execute Configurator

- cfg file (XXXX.cfg)

  This file contains a description of the initial setup items for the system. This should be created by the user.
- Default cfg file (default.cfg)

  This file contains default settings that are used in most cases when settings in the cfg file are omitted. This file should be placed in the directory indicated by environment variable "LIB72MP".
- Version file (version)

  This file contains a description of the HI7200/MP version. This file should be placed in the directory indicated by environment variable "LIB72MP." cfg72mp reads this file and outputs HI7200/MP version information to the startup message.

14.4.4 Files Output by cfg72mp

cfg72mp outputs the following files to the current directory. The output directory cannot be specified by the user.

(1) Current CPUID definition file (mycpuid.h)

This file contains the definition of the ID of the CPU (system.cpuid) in which the configured kernel is to run. This file is included in kernel.h. The following shows an example of CPUID definition output when system.cpuid is 1.

```
  #define MYCPUID 1U
```

(2) ID name header files

These files contain definitions of object ID numbers. Include them in the application when necessary. For details, refer to section 14.6, ID Name Header Files.

The ID name header file includes kernel.h.

(3) kernel_macro.h

This file is included in kernel.h. For its contents, refer to section 14.7, kernel_macro.h.

(4) System definition files

The system definition files are included in kernel_def.c and kernel_cfg.c to create a system in accordance with the specified configuration. The contents of these files are implementation-dependent; the compatibility with future versions will not be guaranteed either. The application must not include these files.

cfg72mp outputs the following system definition files.

1. kernel_cfg.h
2. kernel_cfg_area.h
3. kernel_cfg_extern.h
4. kernel_cfg_inireg.h
5. kernel_cfg_inirtn.h
6. kernel_cfg_ststk.h
7. kernel_def.h
8. kenrel_def_area.h
9. kernel_def_extern.h
10. kernel_def_inireg.h
11. kernel_def_inirtn.h

14.4.5 Starting Configurator

Input the following command line to start the configurator.

```
 C> cfg72mp[-vV] cfg file name
```

When the extension of the cfg file name is omitted, it is assumed as ".cfg".

14.4.6 Command Options

(1) –v option

Displays the description of the command options and detailed version information.

(2) –V option

Displays the information of the files generated through the command execution.

14.4.7 Note

After cfg72mp is executed, be sure to recompile the files that include kernel.h because cfg72mp outputs kernel_macro.h that is included in kernel.h.

14.5 Error Messages

14.5.1 Error Output Format and Error Levels

This section describes the meaning of the error messages output in the following format.

```
  Error number  (Error level)  Error message
```

Errors are classified into two levels as shown in table 14.13.

Table 14.13 Error Levels

Error Level		Operation
(W)	Warning	Continues processing.
(E)	Error	Aborts processing.

14.5.2 List of Messages

(1) Error Messages

0001 (E) Illegal option --> <*character*>

The command option has an error.

0002 (E) Illegal argument --> <*string*>

The startup format has an error.

0003 (E) Invalid option

The command option or startup format has an error.

0100 (E) syntax error

Syntax error.

1000 (E) Not enough memory

Insufficient memory.

2000 (E) Can't write open <*file name*>

The file cannot be generated. Check the directory attribute and the free space in the disk.

2002 (E) Can't open version file

The version file "version" cannot be found in the current directory or the directory indicated by environment variable "LIB72MP".

2003 (E) can't open default configuration file

The default cfg file (default.cfg) cannot be found in the current directory or the directory indicated by environment variable "LIB72MP".

2004 (E) Can't open configuration file <*file name*>

The specified configuration file cannot be accessed.

3000 (E) Zero divide error

The cfg file has a zero division expression.

3001 (E) Illegal *XXXX* --> <*setting*>

The setting for definition name XXXX is illegal.

3002 (E) Illegal number expression --> <*string*>

The specified string cannot be converted to a numeric value.

3003 (E) Unknown token --> <*string*>

The specified string cannot be recognized as a definition name.

3003 (E) Unknown *XXXX* --> <*string*>

The string specified as the setting for definition name XXXX is not allowed in the cfg72mp specification.

3004 (E) Number of tasks exceeds upper limit(1023)

The number of tasks exceeds the upper limit (1023).

3005 (E) Illegal number of *XXXX* --> <*value*>

There are too many XXXX[] definitions.

4000 (E) *XXXX* not defined

Definition name XXXX is not defined.

4002 (E) *XXXX*[].*YYYY* not defined

XXXX[].YYYY is not defined.

4003 (E) When “name" is omitted, “ID" cannot be omitted.

When "name" is omitted, the local ID number must always be specified.

4200 (E) Double definition <*XXXX*>

Definition name XXXX is defined multiple times.

`4201 (E) Double definition` *`xxxx`*`[`*`number`*`]`

The object definition item is defined multiple times.

`4202 (E) System timer's vector <`*`Vector number`*`> conflict`

The vector number specified in interrupt_vector[] is already specified as clock.number.

`4203 (E) Double definition taskid=`*`Local ID`*` in static_stack[`*`stack no.`*`] and [`*`stack no.`*`]`

A single task is specified for multiple static_stack[].tskid settings.

`4300 (E) The ID of task[] with name="`*`ID name`*`" does not use static stack`

In static_stack[].tskid, the name of task[] that has a local ID number larger than maxdefine.max_statictask or task[] without a local ID number is specified. When an ID name is used to specify a task that uses a static stack, a local ID number must be specified in the corresponding task[] and the ID number must not exceed maxdefine.max_statictask.

`4301 (E) The task ID=`*`Local ID`*` does not use static stack`

In static_stack[].tskid, a local ID number larger than maxdefine.max_statictask is specified. When a local ID number is used to specify a task that uses a static stack, the local ID number must not exceed maxdefine.max_statictask.

`4302 (E) The task[] with name=`*`ID name`*` is not defined`

No task[] definition that has the ID name specified in static_stack[].tskid is found.

`4303 (E) Static stack for tskid=`*`Local ID`*` is not assigned`

No static stack is assigned to the task with the specified local ID number. Add the local task ID in any of the static_stack[].tskid settings.

`4400 (E)` *`YYYY`*` must set `*`ZZZZ`*` or less in `*`XXXX`*` definition`

XXXX.YYYY should ZZZZ or smaller.

`4401 (E)` `*YYYY* must set *ZZZZ* or more in *XXXX* definition`

XXXX.YYYY should ZZZZ or greater.

`4402 (E)` `Can't define both *XXXX* keyword and *YYYY* keyword in *ZZZZ* definition`

XXXX and YYYY cannot be specified for definition item ZZZZ at the same time.

`4403 (E)` `*XXXX* exceeds 512MB`

XXXX exceeds 512 Mbytes.

`4404 (E)` `Total of required default *XXXX* size exceeds 512MB`

The size of the required default XXXX area exceeds 512 Mbytes.

`4406 (E)` `Too big task[*Local ID*]'s priority --> <*priority*>`

The specified task[Local ID].priority exceeds system.priority.

`4407 (E)` `Too big IPL --> <*clock.IPL setting*>`

clock.IPL exceeds system.system_IPL.

`4408 (E)` `Either system.tic_deno or system.tic_nume must be 1.`

Either system.tic_nume or system.tic_deno must be 1.

(2) Warning Messages

`8000 (W)` `*XXXX* is not defined.`

Definition name XXXX is omitted; the setting in the default cfg file is used.

`8100 (W)` `Already definition *XXXX*`

Definition name XXXX has already been defined; the first definition is used.

`8101 (W)` `*XXXX*[*number*] definition is ignored`

The corresponding service call is not selected; the object definition is ignored.

`8200 (W) XXXX is corrected to YYYY`

The setting for definition name XXXX is corrected to YYYY.

`8201 (W) XXXX is not multiple of 4 --> <YYYY>`

The setting for definition name XXXX is rounded up to YYYY.

14.6 ID Name Header Files

14.6.1 Overview

The ID names of objects are output to ID name header files in the following format.

```
  #define <ID name>  MAKE_ID(<CPUID>, <ID number>)
```

<CPUID> indicates the CPUID specified in system.cpuid.

<ID name> is a user-specified ID name.

MAKE_ID() is a macro for creating an ID number from the CPUID and local ID; it is defined in kernel.h. The ID name header files include kernel.h.

14.6.2 Types of ID Name Header Files

(1) kernel_id.h

To use an ID name specified in the cfg file, be sure to include kernel_id.h.

kernel_id.h contains definitions of the ID names that are not exported to the other CPU (xxx[].export is set to NO) and it includes kernel_id_cpu1.h or kernel_id_cpu2.h described below, which each contain definitions of the ID names that are exported to the other CPU.

(2) kernel_id_cpu1.h and kernel_id_cpu2.h

These files contain definitions of the ID names that are exported to the other CPU (xxx[].export is set to YES).

cfg72mp outputs kernel_id_cpu1.h when system.cpuid is 1 or kernel_id_cpu2.h when system.cpuid is 2.

(3) kernel_id_sys.h, kenrel_id_sys_cpu1.h, and kernel_id_sys_cpu2.h

These files are reserved for future extensions. No definition statements will be output to them in most cases.

14.7 kernel_macro.h

This file is included in kernel.h.

```
#define TMAX_TPRI  10
#define TMAX_MPRI 10
#define TIC_NUME 1UL
#define TIC_DENO 1UL
#define TIM_LVL 13UL
#define TIM_INHNO 64UL
#define VTCFG_TBR _FOR_SVC
#define VTCFG_MPFMANAGE _OUT
#define VTCFG_NEWMPL _NEW
#define VTCFG_VECTYPE _ROM
#define VTCFG_REGBANK (_BANKLEVEL01|_BANKLEVEL14|_BANKLEVEL15)
```

(1) TMAX_TPRI

This indicates the maximum task priority (system.priority).

(2) TMAX_MPRI

This indicates the maximum message priority (system.message_pri).

(3) TIC_NUME and TIC_DENO

These indicate the numerator (clock_tic_nume) and denominator (clock.tic_deno) of the time tick, respectively. These definitions are ignored when clock.timer is set to NOTIMER.

(4) TIM_LVL

This indicates the timer interrupt level (clock.IPL). It is set to 0 when clock.timer is set to NOTIMER.

When creating a timer driver, implement the timer initialization processing according to the TIC_NUME, TIC_DENO, and TIM_LVL settings.

(5) VTCFG_TBR

This indicates the TBR usage (system.tbr). A symbol generated by adding "_" at the beginning of the symbol specified in system.tbr is set here. The definition value for each symbol is as follows.

```
#define _NOMANAGE      0UL
#define _FOR_SVC       1UL
#define _TASK_CONTEXT  2UL
```

(6) VTCFG_MPFMANAGE

This indicates the fixed-sized memory pool management (system.mpfmanage). A symbol generated by adding "_" at the beginning of the symbol specified in system.mpfmanage is set here. The definition value for each symbol is as follows.

```
#define _IN        0UL
#define _OUT       1UL
```

(7) VTCFG_NEWMPL

This indicates the variable-sized memory pool management (system.newmpl). A symbol generated by adding "_" at the beginning of the symbol specified in system.nemwpl is set here. The definition value for each symbol is as follows.

```
#define _PAST      0UL
#define _NEW       1UL
```

(8) VTCFG_VECTYPE

This indicates the interrupt vector type (system.vector_type). A symbol generated by adding "_" at the beginning of the symbol specified in system.vector_type is set here. The definition value for each symbol is as follows.

```
#define _ROM_ONLY_DIRECT 0UL
#define _RAM_ONLY_DIRECT 1UL
#define _ROM             2UL
#define _RAM             3UL
```

(9) VTCFG_REGBANK

This indicates the register bank usage (system.regbank). A symbol generated by adding "_" at the beginning of the symbol specified in system.regbank is set here. The definition value for each symbol is as follows.

```
#define _NOTUSE        0UL
#define _ALL           0x40000000UL
#define _BANKLEVEL01   0x00000002UL
#define _BANKLEVEL02   0x00000004UL
#define _BANKLEVEL03   0x00000008UL
#define _BANKLEVEL04   0x00000010UL
#define _BANKLEVEL05   0x00000020UL
#define _BANKLEVEL06   0x00000040UL
#define _BANKLEVEL07   0x00000080UL
#define _BANKLEVEL08   0x00000100UL
#define _BANKLEVEL09   0x00000200UL
#define _BANKLEVEL10   0x00000400UL
#define _BANKLEVEL11   0x00000800UL
#define _BANKLEVEL12   0x00001000UL
#define _BANKLEVEL13   0x00002000UL
#define _BANKLEVEL14   0x00004000UL
#define _BANKLEVEL15   0x00008000UL
```

(10) VTCFG_TRACE

This indicates the service call trace (system.trace). A symbol generated by adding "_" at the beginning of the symbol specified in systemtrace is set here. The definition value for each symbol is as follows.

```
#define _NO             0UL
#define _TARGET_TRACE   1UL
#define _TOOL_TRACE     2UL
```

(11) VTCFG_TRACE_OBJECT

This indicates the number of objects that can be acquired by service call trace (system.trace_object). 0 is output when system.trace is set to NO.

Section 15 GUI Configurator

The GUI configurator is a tool used to input various types of kernel configuration information on the GUI screen to create a cfg file. The output cfg file should be input to cfg72mp.

The GUI configurator provides an easy way to configure the kernel without learning how to write a cfg file.

Figure 15.1 shows the relationship among the GUI configurator, cfg file, and cfg72mp.

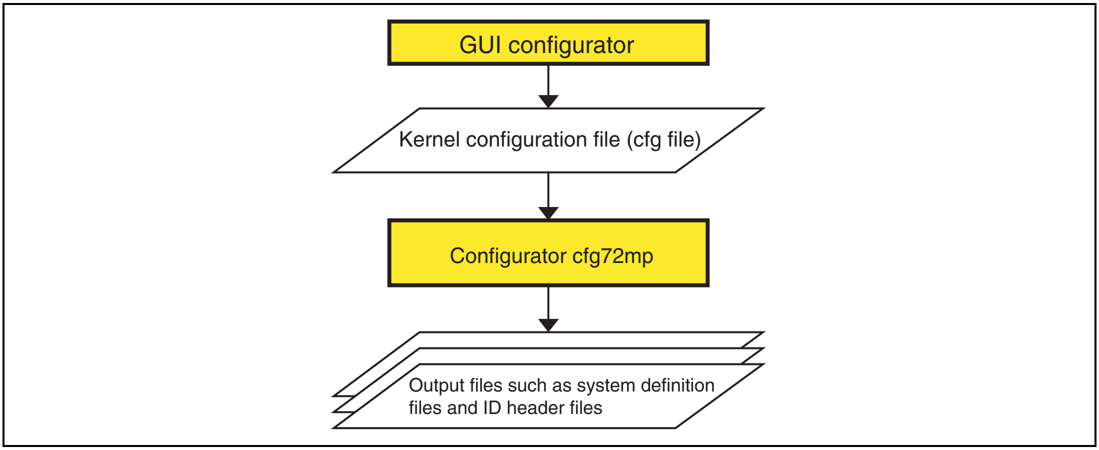

GUI configurator
Kernel configuration file (cfg file)
Configurator cfg72mp
Output files such as system definition files and ID header files

Figure 15.1 Relationship among GUI Configurator, cfg File, and cfg72mp

For information on the operation of the GUI configurator, refer to the online help.

Section 16 Sample Programs

This section describes a sample of programs stored in the <SAMPLE_INST>\R0K572650D000BR directory.

This sample is created to show the behavior of the OS functions. Use of an emulator, such as the E10A-USB, to check the operation is assumed; this sample does not externally input or output any data.

16.1 Target Hardware

This sample is created for use on the Renesas R0K572650D00BR evaluation board equipped with the SH7265 microcomputer.

Table 16.1 gives an overview of the board specifications and figure 16.1 shows the SH7265 memory map on the R0K572650D00BR.

Table 16.1 Overview of R0K572650D00BR Specifications

Item	Specifications
Microcomputer	• SH7265 (R5S72653P200BG) — Input clock (XIN): 16.67 MHz — Bus clock: 66.67 MHz max. — CPU clock: 200 MHz max.
External memory	• NOR flash memory (CS0 space, 16-bit bus): 16 Mbytes — S29GL128M90TFIR2 (manufactured by SPANSION) × 1 • SDRAM (SDRAM0 space, 16-bit bus): 32 Mbytes — EDS2516APTA-75 (manufactured by Elpida) × 1

	SH7265 logical space
0x00000000	CS0 space (64 Mbytes)
0x04000000	CS1 space (64 Mbytes)
0x08000000	CS2 space (64 Mbytes)
0x0C000000	CS3 space (64 Mbytes)
0x10000000	CS4 space (64 Mbytes)
0x14000000	CS5 space (64 Mbytes)
0x18000000	SDRAM0 space (64 Mbytes)
0c1C000000	SDRAM1 space (64 Mbytes)
0x20000000	CS0 to CS5, SDRAM0, and SDRAM1 spaces (cache-disabled)
0x40000000	Reserved (access-prohibited)
0xE8000000	On-chip peripheral modules
0xEC000000	Reserved (access-prohibited)
0xFF400000	On-chip peripheral modules
0xFFC00000	Reserved (access-prohibited)
0xFFD80000	Fast on-chip RAM0 (shadow) (64 Kbytes)
0xFFD90000	Reserved (access-prohibited)
0xFFDA0000	Fast on-chip RAM1 (shadow) (32 Kbytes)

	R0K572650D00BR memory map
0x00000000 0x00FFFFFF	Flash memory (16 Mbytes) 16-bit bus
	User area
0x04000000	User area
0x08000000	User area
0x0C000000	User area
0x10000000	User area
0x14000000	User area
0x18000000 0x19FFFFFF	SDRAM (32 Mbytes)
0x1C000000 0x1DFFFFFF	SDRAM (32 Mbytes) Not mounted (only patterns are printed)
0x20000000	CS0 to CS5, SDRAM0, and SDRAM1 spaces (cache-disabled)
0x40000000	Reserved (access-prohibited)
0xE8000000	On-chip peripheral modules
0xEC000000	Reserved (access-prohibited)
0xFF400000	On-chip peripheral modules
0xFFC00000	Reserved (access-prohibited)
0xFFD80000	Fast on-chip RAM0 (shadow) (64 Kbytes)
0xFFD90000	Reserved (access-prohibited)
0xFFDA0000	Fast on-chip RAM1 (shadow) (32 Kbytes)

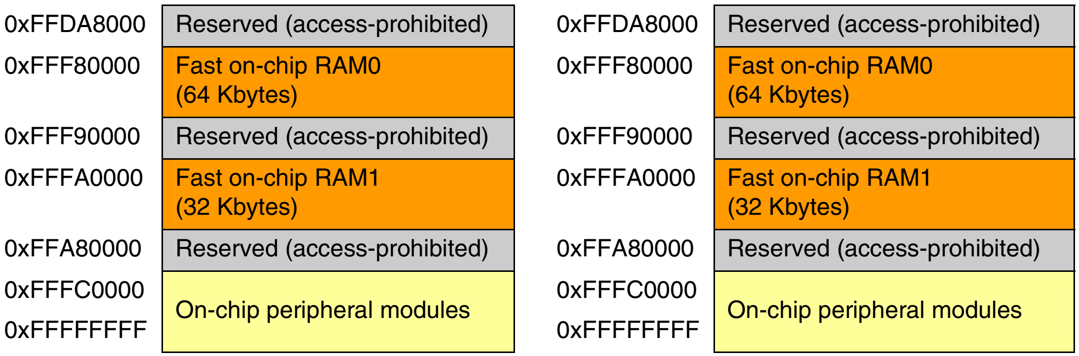


Figure 16.1 SH7265 Memory Map

16.2 Directory Structure

The following shows the structure of the directories under the <SAMPLE_INST>\R0K572650D000BR directory.

include\	Header files common to both CPUs
iodefine\	Hardware definition header files, peripheral clock frequency definition, and kernel_intspec.h
cpuid1\	Workspace directory for CPUID#1
include\	Header file common to CPUID#1 sample programs
cfg_out\	cfg file, configurator output files, kernel_def.c, and kernel_cfg.c
ipi\	IPI
oal\	OAL
rpc_config\	RPC data file (rpc_table.c)
reset\	Reset handling
os_timer\	Timer driver for on-chip CMT of SH7205 or SH7265
init_task\	Initial startup task
stdlib\	Initialization functions and low-level functions of the standard library
sysdwn\	System down routines
dummy_prog\	Dummy programs
rpc_sample_clnt\	RPC client stub example
rpc_caller\	RPC call example
remote_svc_sample\	Remote service call example
prj_cpuid1\	Project directory
debug\	"debug" configuration directory
cpuid2\	Workspace directory for CPUID#2
include\	Header file common to CPUID#2 sample programs
cfg_out\	cfg file, configurator output files, kernel_def.c, and kernel_cfg.c

ipi\	IPI
oal\	OAL
rpc_config\	RPC data file (rpc_table.c)
reset\	Reset handling
os_timer\	Timer driver for on-chip CMT of SH7205 or SH7265
init_task\	Initial startup task
stdlib\	Initialization functions and low-level functions of the standard library
sysdwn\	System down routines
dummy_prog\	Dummy programs
rpc_sample_svr\	RPC server stub and RPC server function example
remote_svc_sample\	Remote service call example
prj_cpuid2\	Project directory
debug\	"debug" configuration directory

The following files are not sample programs but are stored in this directory for convenience.

(1) kernel_def.c, kernel_cfg.c

(2) IPI

(3) OAL

(4) RPC data file (rpc_table.c)

16.3 Startup Processing

This section describes the procedures of this sample processing until each CPU moves to the multitasking environment after a reset. Please fully understand this description and the sample code to avoid wasting time to check the startup state in the early stages of application system development.

16.3.1 Overview

Figure 16.2 is a schematic flowchart of procedures until each CPU moves to the multitasking environment after a reset.

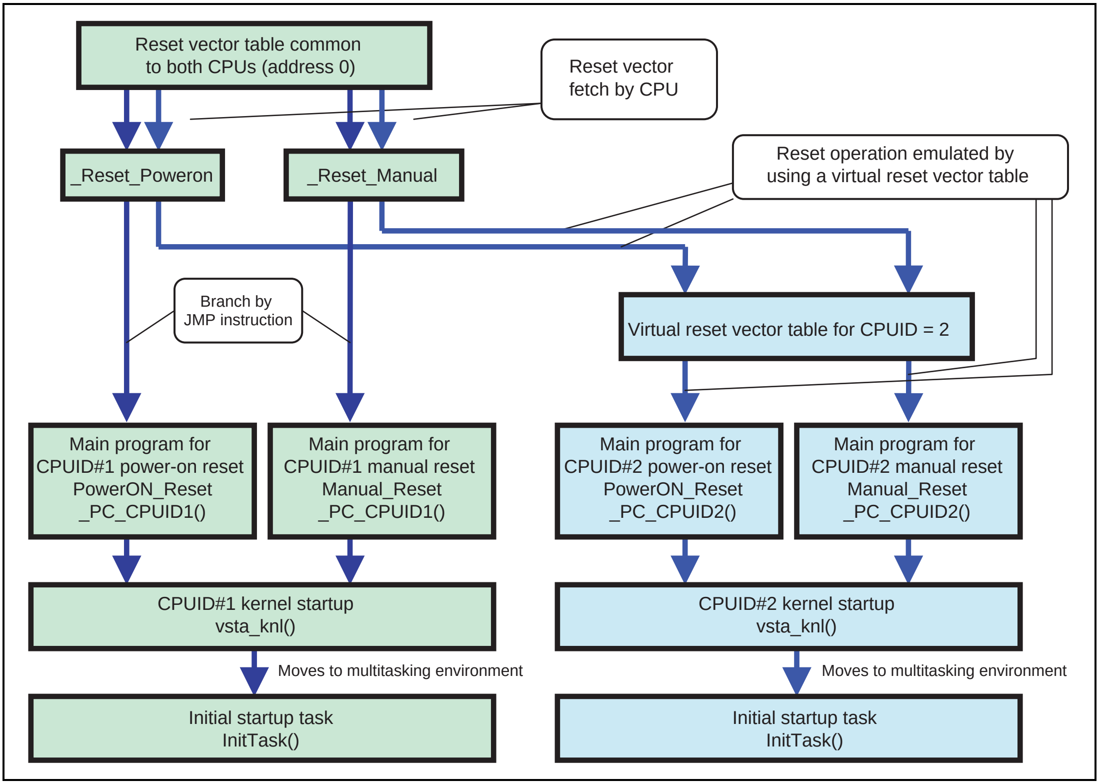


Figure 16.2 Schematic Flowchart of Startup Procedures

The legend for figure 16.2 is as follows.

- Red arrow: CPUID#1 operation
- Blue arrow: CPUID#2 operation
- Shaded in green: CPUID#1 linkage unit
- Shaded in blue: CPUID#2 linkage unit

As the initialization processing of the CPUs should be executed in a specified order, it is controlled by using initialization flags. For details, refer to section 16.3.9, Synchronization of Startup Phases in Two CPUs.

Table 16.2 shows the source files described in this section.

Table 16.2 Source Files Related to Startup Processing

Directory	File Name	Function
cpuid1\reset	reset.src	(1) Reset vector table (address 0) (2) _Reset_Poweron (3) _Reset_Manual
	resetprg1.c	Main processing for CPUID#1 reset: (1) PowerON_Reset_PC_CPUID1() (2) Manual_Reset_PC_CPUID1()
	hwsetup1.c	Initialization of common hardware and CPUID#1-dedicated hardware: HardwareSetup_CPUID1()
	cpg1.c	CPG (FRQCR0) initialization: io_set_cpg_couid1()
	uram.c	On-chip RAM initial settings: io_set_uram()
	bsc_cs0.c	CS0 initialization: io_init_bsc_cs0()
	bscsdram.c	SDRAM space initialization: io_init_sdram()
cpuid1\init_task\	init_task1.c	CPUID#1 initial startup task: InitTask1()
cpuid2\reset\	reset\vreset.src	Virtual reset vector table for CPUID#2
	reset\resetprg2.c	Main processing for CPUID#2 reset: (1) PowerON_Reset_PC_CPUID2() (2) Manual_Reset_PC_CPUID2()
	reset\hwsetup2.c	Initialization of CPUID#2-dedicated hardware: HardwareSetup_CPUID2()
	reset\cpg2.c	CPG (FRQCR2) initialization: io_set_cpg_cpuid2()
cpuid2\init_task\	init_task\init_task2.c	CPUID#2 initial startup task: InitTask2()

Figure 16.3 is a flowchart of the CPUID#1 startup procedures and figure 16.4 is a flowchart of the CPUID#2 startup procedures. The rectangles shaded in yellow in the figures indicate API calls provided in the HI7200/MP.

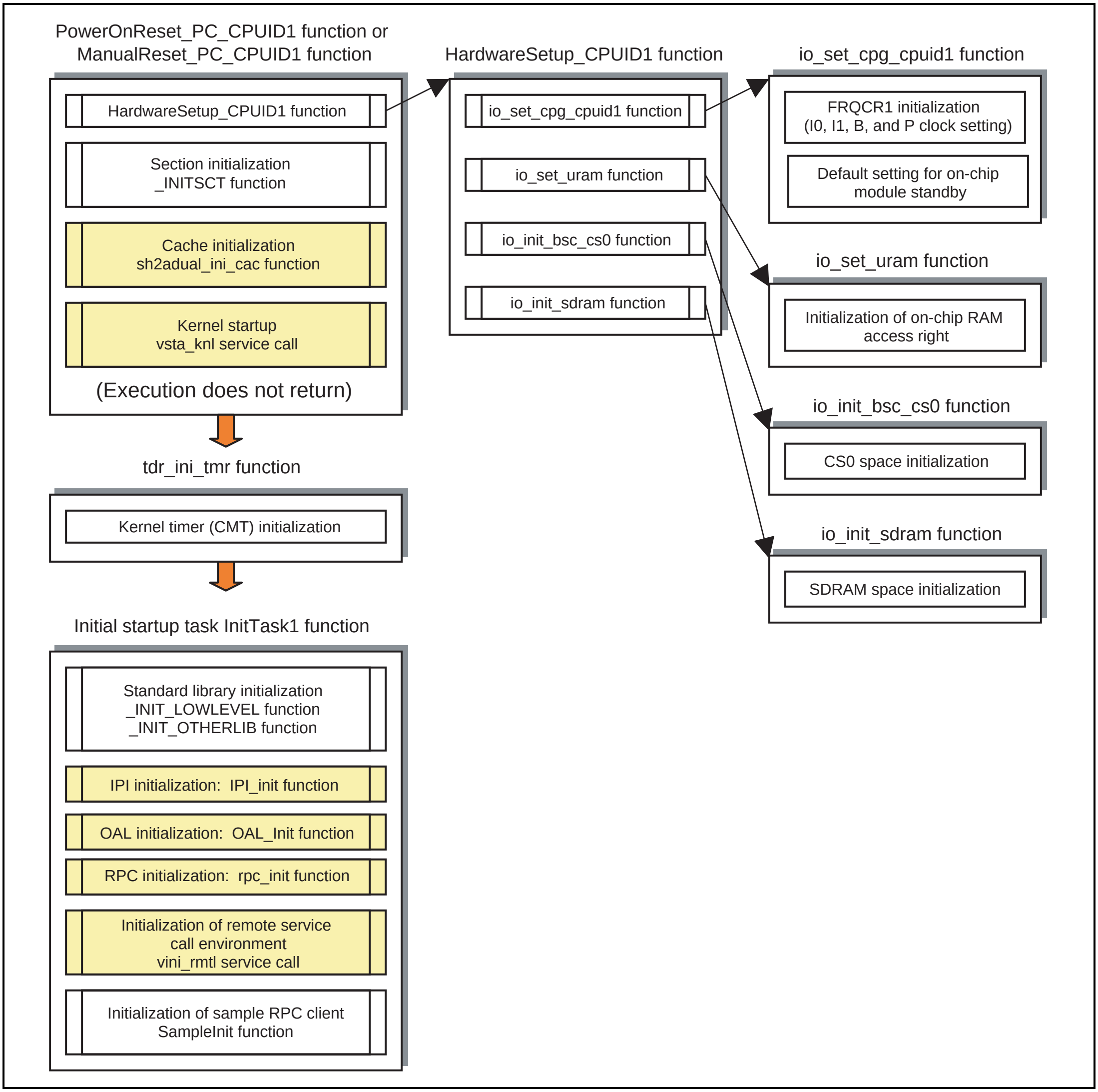

PowerOnReset_PC_CPUID1 function or
ManualReset_PC_CPUID1 function
HardwareSetup_CPUID1 function
Section initialization
_INITSCT function
Cache initialization
sh2adual_ini_cac function
Kernel startup
vsta_knl service call
(Execution does not return)
tdr_ini_tmr function
Kernel timer (CMT) initialization
Initial startup task InitTask1 function
Standard library initialization
_INIT_LOWLEVEL function
_INIT_OTHERLIB function
IPI initialization: IPI_init function
OAL initialization: OAL_Init function
RPC initialization: rpc_init function
Initialization of remote service
call environment
vini_rmtl service call
Initialization of sample RPC client
SampleInit function
HardwareSetup_CPUID1 function
io_set_cpg_cpuid1 function
io_set_uram function
io_init_bsc_cs0 function
io_init_sdram function
io_set_cpg_cpuid1 function
FRQCR1 initialization
(I0, I1, B, and P clock setting)
Default setting for on-chip
module standby
io_set_uram function
Initialization of on-chip RAM
access right
io_init_bsc_cs0 function
CS0 space initialization
io_init_sdram function
SDRAM space initialization

Figure 16.3 CPUID#1 Startup Flowchart

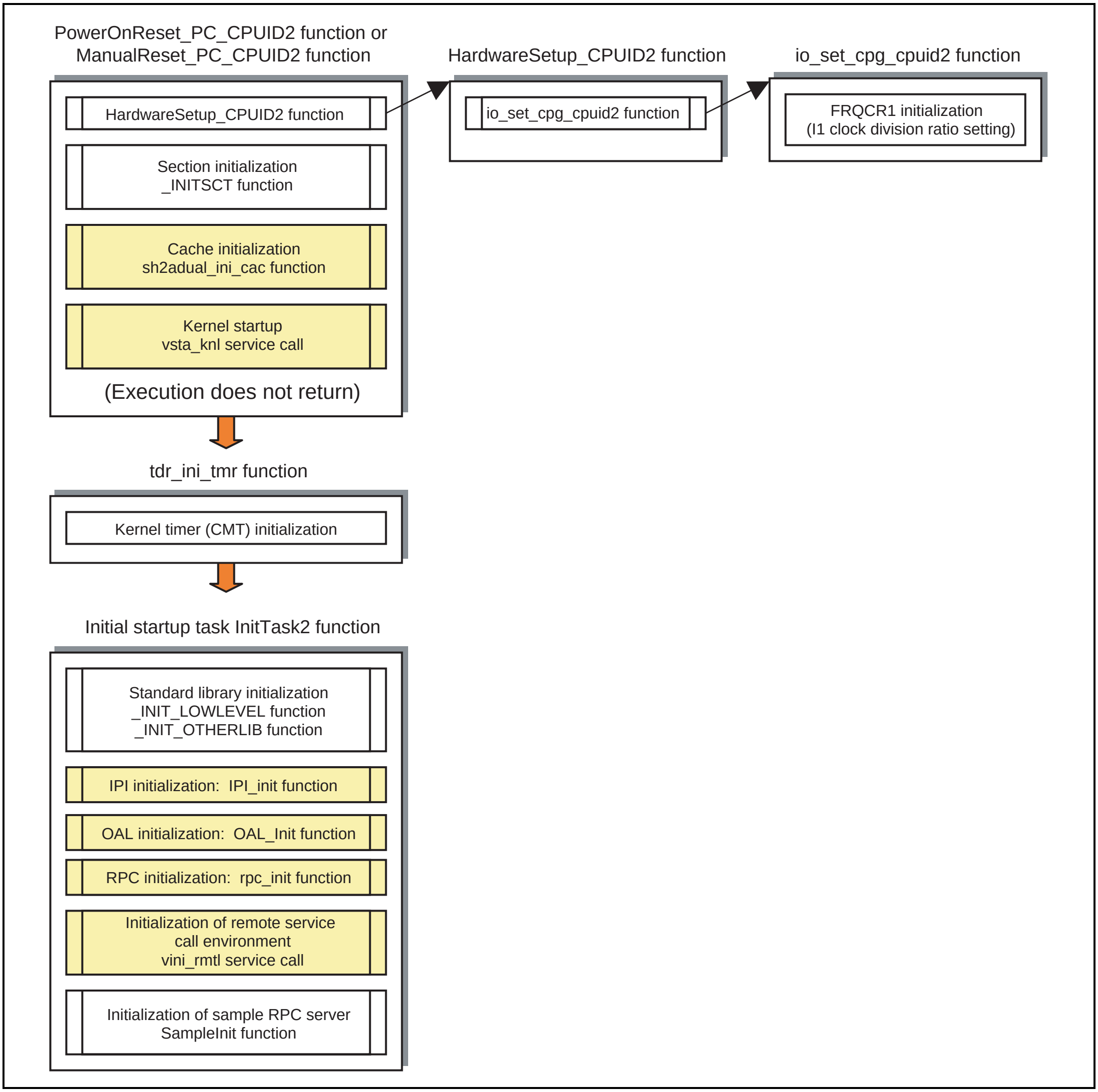

PowerOnReset_PC_CPUID2 function or
ManualReset_PC_CPUID2 function
HardwareSetup_CPUID2 function
Section initialization
_INITSCT function
Cache initialization
sh2adual_ini_cac function
Kernel startup
vsta_knl service call
(Execution does not return)
HardwareSetup_CPUID2 function
io_set_cpg_cpuid2 function
io_set_cpg_cpuid2 function
FRQCR1 initialization
(I1 clock division ratio setting)
tdr_ini_tmr function
Kernel timer (CMT) initialization
Initial startup task InitTask2 function
Standard library initialization
_INIT_LOWLEVEL function
_INIT_OTHERLIB function
IPI initialization: IPI_init function
OAL initialization: OAL_Init function
RPC initialization: rpc_init function
Initialization of remote service
call environment
vini_rmtl service call
Initialization of sample RPC server
SampleInit function

Figure 16.4 CPUID#2 Startup Flowchart

16.3.2 Reset Vectors (cpuid1\reset\reset.src)

In the SH7265, each CPU fetches a vector from the reset vector table at address 0 to start execution.

The user should create a reset vector table such as this sample file for this kernel.

Although this kernel requires a separate load module to be generated for each CPU, a reset vector table should be created for the CPUID#1 linkage unit.

Table 16.3 shows the contents registered in the reset vector table. The reset vector table and the two programs shown in table 16.3 are in reset.src for CPUID#1. This file is written in assembly language.

Table 16.3 Reset Vector Table

Vector No.	Description	Registered Contents
0	Power-on reset PC	_Reset_Poweron: Program for power-on reset
1	Power-on reset SP	End address of on-chip RAM0
2	Manual reset PC	_Reset_Manual: Program for power-on reset
3	Manual reset SP	End address of on-chip RAM0

_Reset_Poweron and _Reset_Manual are executed in both CPUs simultaneously.

The overview of these program operations is given below.

After a reset, the CPU reset handling initializes the stack pointer (R15) as shown in table 16.3 and both CPUs start executing the appropriate program from the above. Note that the stack pointer is initialized for use in CPUID#1; do not use the stack before initializing the stack pointer for CPUID#2. These programs are written in assembly language to ensure that the stack is not used in these programs.

Next, whether CPUID#1 or CPUID#2 is in execution should be determined.

When CPUID#1 is in execution, execution branches through a JMP instruction to the following reset main program for CPUID#1, which is written in C language. Execution does not return after the branch.

- For a power-on reset: PowerON_Reset_PC_CPUID1()
- For a manual reset: Manual_Reset_PC_CPUID1()

On the other hand, when CPUID#2 is in execution, CPUID#2 is in busy-wait state until CPUID#1 completes initialization of common hardware (refer to section 16.3.3, Reset Main Program for CPUID#1), and after that, CPUID#2 emulates operation of a reset only for CPUID#2 by referring to the virtual reset vector table for CPUID#2 (section 16.3.5, Virtual Reset Vector Table for CPUID#2); that is, the stack pointer is initialized according to the virtual reset vector table and execution branches through a JMP instruction to the address registered in the virtual reset vector table. Execution does not return after the branch.

Figure 16.5 shows the source code in reset.src and its description.

```
;****************************************************************************
;* Import virtual reset vector table symbol of CPUID#2
;*
;* When CPUID#1 is linked, you must define the absolute address of
;* the symbol "_ResetVectorTable_CPUID2" manually.
;* When CPUID#2 is linked, you must locate "CC_resetvct" section to the above address.
;* If you locate this section to other address, you must re-link CPUID#1 side with
;* new address definition for "_ResetVectorTable_CPUID2".
;****************************************************************************
    .import     _ResetVectorTable_CPUID2    ; In "CC_resetvct" section of CPUID#2


;****************************************************************************
;* Please define copied program size for CPUID#2
;****************************************************************************
;*** for POWERON RESET
COPYSIZE_POWERON    .assign     20
    ; [Caution!]
    ; The size must be equal to
    ; POWERON_EXEC_RAM1_END - POWERON_EXEC_RAM1_START)

;*** for MANUAL RESET
COPYSIZE_MANUAL     .assign     20
    ; [Caution!]
    ; The size must be equal to
    ; (MANUAL_EXEC_RAM1_END - MANUAL_EXEC_RAM1_START)


;****************************************************************************
;* Definition
;****************************************************************************
```

External reference declaration of the virtual reset vector table for CPUID#2
(address is forcibly defined at linkage)

Size of CPUID#2 program to implement busy-wait in on-chip RAM at a power-on reset

Size of CPUID#2 program to implement busy-wait in on-chip RAM at a manual reset

Figure 16.5 cpuid1\reset\reset.src

32	URAMEND_CPUID1 .assign H'FFF90000 ;* End of URAM0 address (reset stack for CPUID#1)	End address of on-chip RAM0 (initial stack pointer value for CPUID#1)
33		
34	CPUIDR .assign H'FFFC1404 ;* CPUIDR register address	CPUIDR register address
35	CPUIDR_CPU2 .assign H'40000000	
36		
37	DELAY_CPUID2 .assign H'200 ;* delay count of CPUID#2 to wait for CPUID#1	
38	;* to initialize s_ucIsInitHW	
39		
40	;**	Reset vector table common to both CPUs
41	;* Reset vector table	
42	;**	Section name = CC_resetvct
43	.section CC_resetvct, data, align=4	
44	.export _ResetVectorTable	
45		
46	_ResetVectorTable:	
47	;* 0 : Power-on Reset (PC)	
48	.data.l _Reset_Poweron ; in this file	
49	;* 1 : Power-on Reset (SP)	
50	.data.l URAMEND_CPUID1	
51	;* 2 : Manual Reset (PC)	
52	.data.l _Reset_Manual ; in this file	
53	;* 3 : Manual Reset (SP)	
54	.data.l URAMEND_CPUID1	
55		
56	;**	
57	;* _Reset_Poweron	Reset_Poweron program
58	;* Power-on reset program for both CPU (reset vector entry)	
59	;* This program should not use stack.	
60	;**	
61	.section PC_reset	
62	.export _Reset_Poweron	

Figure 16.5 cpuid1\reset\reset.src (cont)

63	.import _PowerON_Reset_PC_CPUID1	
64		
65	_Reset_Poweron:	
66	;*** if I'm CPUID#1 then POWERON_CPUID1, else POWERON_CPUID2	Determines which CPU is in execution.
67	mov.l #CPUIDR,r0	
68	mov.l @r0,r0	
69	mov.l #CPUIDR_CPU2,r1	
70	tst r0,r1	
71	bf POWERON_CPUID2	
72		
73	;************* For CPUID#1 *************	Path to be executed when CPUID#1 is in execution
74	;*** initialize s_ucIsInitHW	Initializes each initialization flag (A1 in figure 16.11).
75	mov.l #_s_ucIsInitHW,r1	
76	mov #0,r0	
77	mov.b r0,@r1	
78		
79	;*** initialize s_ucIsInitKnlCPUID1	
80	mov.l #_s_ucIsInitKnlCPUID1,r1	
81	mov #0,r0	
82	mov.b r0,@r1	
83		
84	;*** initialize s_ucIsInitEnvCPUID1	
85	mov.l #_s_ucIsInitEnvCPUID1,r1	
86	mov #0,r0	
87	mov.b r0,@r1	
88		
89	;*** initialize s_ucIsInitEnvCPUID2	
90	mov.l #_s_ucIsInitEnvCPUID2,r1	
91	mov #0,r0	
92	mov.b r0,@r1	

Figure 16.5 cpuid1\reset\reset.src (cont)

93		
94	;*** wait for the completion that CPUID#2 set s_ucIsInitHW = 1	Waits until CPUID#2 sets s_ucIsInitHW to 1 (B11 in figure 16.11).
95	POWERON1_HW_WAIT:	
96	mov.b @r1,r0	
97	cmp/eq #1,r0	
98	bf POWERON1_HW_WAIT	
99		
100	;*** jump to PowerON_Reset_PC_CPUID1()	Branches to PowerON_Reset_PC_CPUID1().
101	mov.l #_PowerON_Reset_PC_CPUID1,r0	
102	jmp @r0	
103	nop	
104		
105	;************* For CPUID#2 *************	Path to be executed when CPUID#2 is in execution
106	POWERON_CPUID2:	
107	;*** get ResetVectorTable_CPUID2 address to R7	
108	mov.l #_ResetVectorTable_CPUID2,r7	
109		
110	;*** initialize R15	Initializes the stack pointer according to the virtual reset vector table for CPUID#2.
111	mov.l @(4,r7),r15 ; load SP(R15)	
112		
113	;*** delay to wait for CPUID#1 to initialize s_ucIsInitHW	Delays execution for a period long enough for CPUID#1 to initialize s_ucIsInitHW (B21 in figure 16.11).
114	mov.l #DELAY_CPUID2,r0	
115		
116	POWERON2_DELAY:	
117	dt r0	
118	bf POWERON2_DELAY	
119		
120	;*** jump to RAM1	Copies the program to on-chip RAM1 and calls it (B22 in figure 16.11).
121	mov.l #POWERON_EXEC_RAM1_START,r1	
122	mov.l #R_POWERON_EXEC_RAM1_START,r2	
123	mov.l #COPYSIZE_POWERON,r3	
124	add r1,r3	
125		

Figure 16.5 cpuid1\reset\reset.src (cont)

126	POWERON_LOOP_COPY:	
127	mov.w @r1+,r0	
128	mov.w r0,@r2+	
129	cmp/hi r1,r3	
130	bt POWERON_LOOP_COPY	
131		
132	mov.l #R_POWERON_EXEC_RAM1_START,r0	
133	jsr/n @r0	
134		
135	;*** jump to PowerON_Reset_PC_CPUID2()	Branches to the address (PowerON_Reset_PC_CPUID2()) registered in the virtual reset vector table for CPUID#2.
136	mov.l @(0,r7),r1 ; load PC	
137	jmp @r1	
138	nop	
139		
140	.pool	
141		
142	;*** This code is copied to RAM1 *****************	Program to be copied to on-chip RAM1
143	POWERON_EXEC_RAM1_START:	
144	;*** set s_ucIsInitHW = 1	Sets s_ucIsInitHW to 1 (B23 in figure 16.11).
145	mov.l #_s_ucIsInitHW,r1	
146	mov #1,r0	
147	mov.b r0,@r1	
148		
149	;*** wait for the completion that CPUID#1 set s_ucIsInitHW=2	Waits until CPUID#2 sets s_ucIsInitHW to 2 (B24 in figure 16.11).
150	POWERON2_HW_WAIT:	
151	mov.b @r1,r0	
152	cmp/eq #2,r0	
153	bf POWERON2_HW_WAIT	
154		
155	rts ; do not "rts/n"	Returns control.
156	nop	
157	.pool	
158	POWERON_EXEC_RAM1_END:	

Figure 16.5 cpuid1\reset\reset.src (cont)

```
;*** end of copied program


;*****************************************************************************
;* _Reset_Manual                                                 Reset_Manual program
;* Manual reset program for both CPU (reset vector entry)
;*  This program should not use stack.
;*****************************************************************************
   .export    _Reset_Manual
   .import    _Manual_Reset_PC_CPUID1

_Reset_Manual
;*** if I'm CPUID#1 then MANUAL_CPUID1, else MANUAL_CPUID2       Determines which CPU is in
   mov.l   #CPUIDR,r0                                            execution.
   mov.l   @r0,r0
   mov.l   #CPUIDR_CPU2,r1
   tst     r0,r1
   bf      MANUAL_CPUID2

;************* For CPUID#1 *************                        Path to be executed when
                                                                 CPUID#1 is in execution
;*** initialize s_ucIsInitHW                                     Initializes each initialization flag
   mov.l   #_s_ucIsInitHW,r1                                     (A1 in figure 16.11).
   mov     #0,r0
   mov.b   r0,@r1

;*** initialize s_ucIsInitKnlCPUID1
   mov.l   #_s_ucIsInitKnlCPUID1,r1
   mov     #0,r0
   mov.b   r0,@r1

;*** initialize s_ucIsInitEnvCPUID1
   mov.l   #_s_ucIsInitEnvCPUID1,r1
   mov     #0,r0
```

Figure 16.5 cpuid1\reset\reset.src (cont)

mov.b r0,@r1	
;*** initialize s_ucIsInitEnvCPUID2	
mov.l #_s_ucIsInitEnvCPUID2,r1	
mov #0,r0	
mov.b r0,@r1	
;*** wait for the completion that CPUID#2 set s_ucIsInitHW = 1	Waits until CPUID#2 sets s_ucIsInitHW to 1 (B11 in figure 16.11).
MANUAL1_HW_WAIT:	
mov.b @r1,r0	
cmp/eq #1,r0	
bf MANUAL1_HW_WAIT	
;*** jump to Manual_Reset_PC_CPUID1()	Branches to ManualReset_PC_CPUID1().
mov.l #_Manual_Reset_PC_CPUID1,r0	
jmp @r0	
nop	
;************* For CPUID#2 *************	Path to be executed when CPUID#2 is in execution
MANUAL_CPUID2:	
;*** get ResetVectorTable_CPUID2 address to R7	
mov.l #_ResetVectorTable_CPUID2,r7	
;*** initialize R15	Initializes the stack pointer according to the virtual reset vector table for CPUID#2.
mov.l @(12,r7),r15 ; load SP(R15)	
;*** delay to wait for CPUID#1 to initialize s_ucIsInitHW	Delays execution for a period long enough for CPUID#1 to initialize s_ucIsInitHW (B21 in figure 16.11).
mov.l #DELAY_CPUID2,r0	
MANUAL2_DELAY:	
dt r0	
bf MANUAL2_DELAY	

Figure 16.5 cpuid1\reset\reset.src (cont)

225	;*** jump to RAM1	Copies the program to on-chip RAM1 and calls it (B22 in figure 16.11).
226	mov.l #MANUAL_EXEC_RAM1_START,r1	
227	mov.l #R_MANUAL_EXEC_RAM1_START,r2	
228	mov.l #COPYSIZE_MANUAL,r3	
229	add r1,r3	
230		
231	MANUAL_LOOP_COPY:	
232	mov.w @r1+,r0	
233	mov.w r0,@r2+	
234	cmp/hi r1,r3	
235	bt MANUAL_LOOP_COPY	
236		
237	mov.l #R_MANUAL_EXEC_RAM1_START,r0	
238	jsr/n @r0	
239		
240	;*** jump to Manual_Reset_PC_CPUID2()	
241	mov.l @(8,r7),r1 ; load PC	
242	jmp @r1	
243	nop	
244		
245	.pool	
246		
247	;*** This code is copied to RAM1 *****************	Program to be copied to on-chip RAM1
248	MANUAL_EXEC_RAM1_START:	
249	;*** set s_ucIsInitHW = 1	Sets s_ucIsInitHW to 1(B23 in figure 16.11).
250	mov.l #_s_ucIsInitHW,r1	
251	mov #1,r0	
252	mov.b r0,@r1	
253		
254	;*** wait for the completion that CPUID#1 set s_ucIsInitHW=2	Waits until CPUID#2 sets s_ucIsInitHW to 2 (B24 in figure 16.11).
255	MANUAL2_HW_WAIT:	
256	mov.b @r1,r0	
257	cmp/eq #2,r0	
258	bf MANUAL2_HW_WAIT	

Figure 16.5 cpuid1\reset\reset.src (cont)

rts ; do not "rts/n"	Returns control.
nop	
.pool	
MANUAL_EXEC_RAM1_END:	
;*** end of copied program	
;**	
;* Program Section in RAM1 for CPUID#2	
;**	
.section BD_URAM1, data, align=4	Section allocated to on-chip RAM1
R_POWERON_EXEC_RAM1_START:	Destination area in on-chip RAM for program copy (power-on reset)
.res.b COPYSIZE_POWERON	
R_MANUAL_EXEC_RAM1_START:	Destination area in on-chip RAM for program copy (manual reset)
.res.b COPYSIZE_MANUAL	
;**	
;* Flags in RAM0	Initialization flags
;**	
.section BL_S_URAM0, data, align=4	
.export _s_ucIsInitHW, _s_ucIsInitKnlCPUID1, _s_ucIsInitEnvCPUID1, _s_ucIsInitEnvCPUID2	
_s_ucIsInitHW:	
.res.b 1	
_s_ucIsInitKnlCPUID1:	
.res.b 1	
_s_ucIsInitEnvCPUID1:	
.res.b 1	
_s_ucIsInitEnvCPUID2:	
.res.b 1	
.end	

Figure 16.5 cpuid1\reset\reset.src (cont)

16.3.3 Reset Main Program for CPUID#1 (cpuid1\reset\resetprg1.c)

The following two reset main programs are necessary for CPUID#1.

- For power-on reset: PowerON_Reset_PC_CPUID1()
- For manual reset: Manual_Reset_PC_CPUID1()

(1) PowerON_Reset_PC_CPUID1()

This function is initiated through a branch from _Reset_Poweron in reset.src when a power-on reset is generated during CPUID#1 execution. This function must not return.

This function performs the following processing.

1. Initializes the common hardware and CPUID#1-dedicated hardware (calls HardwareSetup_CPUID1()).
2. Initializes the section (calls _INITSCT()).
3. Initializes the cache (calls sh2adual_ini_cac()). [3]
4. Starts the kernel (calls vsta_knl).

Note that the standard libraries are initialized by the initial startup task because the kernel functions may be required to initialize the standard libraries in some cases; for example, when a reentrant library is used.

(2) Manual_Reset_PC_CPUID1()

This function is initiated through a branch from _Reset_Manual in reset.src when a manual reset is generated during CPUID#1 execution. This function must not return.

This function performs almost the same processing as PowerON_Reset_PC_CPUID1().

[3] The cache can be initialized before this, but note the following case.
For the section initialization processing involving a program section copy from ROM to RAM, if the cache is initialized in copy-back mode before section initialization, the contents of the operand cache corresponding to the program code area to which the program has been copied should be copied back to the actual memory after section initialization. Otherwise, an illegal instruction code will be executed if instructions are fetched from the program code address before the program code that has been written to the operand cache during section initialization is copied back to the actual memory through cache entry replacing.

Figure 16.6 shows the source code in resetprg1.c and its description.

```
/****************************************************************************
 Includes
****************************************************************************/
#include <machine.h>

#include "types.h"
#include "kernel.h"
#include "sh2adual_cache.h"

#include "io_sys.h"
#include "io_multicore.h"

#include "initsct.h"

/****************************************************************************
 Prototypes
****************************************************************************/
void PowerON_Reset_PC_CPUID1(void);
void Manual_Reset_PC_CPUID1(void);

/****************************************************************************
 External reference
****************************************************************************/
extern void HardwareSetup_CPUID1(void);

/****************************************************************************
 Section definition
****************************************************************************/
#pragma section C_reset

```

Specifies the section name.

Figure 16.6 cpuid1\reset\resetprg1.c

```
/******************************************************************************
 Initialize section information
******************************************************************************/
/*** D section information table ***/
static const ST_DTBL dtbl[]= {
    MACRO_ENTRY_DTBL("DC_stdlib", "RC_stdlib")
};
/*** B section information table ***/
static const ST_BTBL btbl[]= {
    MACRO_ENTRY_BTBL("BC_stdlib"),
    MACRO_ENTRY_BTBL("BC_sample"),
    MACRO_ENTRY_BTBL("BC_heap")
};

/******************************************************************************/
/**  Power-on Reset function
 *    @retval None
 *    @note This routine is called from "_Reset_Poweron" in "reset.src",
 *          and must not return.
******************************************************************************/
#pragma noregsave(PowerON_Reset_PC_CPUID1)
void PowerON_Reset_PC_CPUID1(void)
{
    extern UINT8    s_ucIsInitHW;   /* defined in reset.src */

    set_imask(15);                  /* set SR.IMASK = 15 */

    /*** check CPUID ***/
    if(IO_CPUIDR.BIT._ID != (MYCPUID)-1U) {
        while(1){           /* error */
        };
    }

```

Section initialization information table

PowerON_Reset_PC_CPUID1()

The register contents do not need to be saved because execution does not return.

Stops processing when the unexpected CPU is in execution.

Figure 16.6 cpuid1\reset\resetprg1.c (cont)

```
    /*** initialize shared hardware and CPUID#1 hardware ***/
    HardwareSetup_CPUID1();
    s_ucIsInitHW = 2;

```

Sets s_ucIsInitHW to 2 after initialization of common hardware (B12 in figure 16.11).

```
    /*** initialize section ***/
    _INITSCT(dtbl, sizeof dtbl, btbl, sizeof btbl);

```

Initializes the section.

```
    /*** initialize cache ***/
    sh2adual_ini_cac(TCAC_IC_ENABLE|TCAC_OC_ENABLE);

```

Initializes the cache.

```
    /*** start kernel (never return) ***/
    vsta_knl();

    /*** (NEVER return from vsta_knl()) ***/
    while(1) {
    }
}

```

Starts the kernel (do not return control here).

```
/****************************************************************************/
/** Manual Reset function
 *    @retval None
 *    @note This routine is called from "_Reset_Manual" in "reset.src",
 *         and must not return.
****************************************************************************/
#pragma noregsave(Manual_Reset_PC_CPUID1)
void Manual_Reset_PC_CPUID1(void)
{
    extern UINT8    s_ucIsInitHW;   /* defined in reset.src */

    set_imask(15);                  /* set SR.IMASK = 15 */

    /*** clear DSFR.MRES ***/
    IO_SYS.DSFR.BIT._MRES = 0;

```

ManualReset_PC_CPUID1()

Figure 16.6 cpuid1\reset\resetprg1.c (cont)

98	/*** initialize shared hardware and CPUID#1 hardware ***/	
99	HardwareSetup_CPUID1();	Sets s_ucIsInitHW to 2 after initialization of common hardware (B12 in figure 16.11).
100	s_ucIsInitHW = 2;	
101		
102	/*** initialize section ***/	Initializes the section.
103	_INITSCT(dtbl, sizeof dtbl, btbl, sizeof btbl);	
104		
105	/*** initialize cache ***/	Initializes the cache.
106	sh2adual_ini_cac(TCAC_IC_ENABLE|TCAC_OC_ENABLE);	
107		
108	/*** start kernel (never return) ***/	Starts the kernel (do not return control here).
109	vsta_knl();	
110		
111	/*** (NEVER return from vsta_knl()) ***/	
112	while(1)	
113	{	
114	}	
115	}	

Figure 16.6 cpuid1\reset\resetprg1.c (cont)

16.3.4 Common Hardware and CPUID#1 Resource Initialization Function HardwareSetup_CPUID1() (cpuid1\reset\hwsetup1.c)

The hardware resources shared by both CPUs and the CPUID#1 hardware resources are initialized through HardwareSetup_CPUID1() executed in CPUID#1. HardwareSetup_CPUID1() is called from PowerON_Reset_PC_CPUID1() and Manual_Reset_PC_CPUID1().

HardwareSetup_CPUID1() calls the following functions.

(1) io_set_cpg_cpuid1() (cpg1.c)

This function sets FRQCR0 to specify the clocks as follows.

- CPUID#1 internal clock (I0ϕ): 200 MHz
- CPUID#2 internal clock (I1ϕ): 200 MHz
- Bus clock (Bϕ): 66.67 MHz
- Peripheral clock (Pϕ): 33.33 MHz

This function also makes the default settings determining whether each on-chip module moves to a standby state.

Note that the I1ϕ division ratio setting in FRQCR1 is done through io_set_cpg_cpuid2() (cpg2.c) in CPUID#2 because FRQCR1 can only be modified by CPUID#2 according to the LSI specification.

In addition to this function, the peripheral clock frequency is defined in iodefine\pclock.h; when modifying the peripheral clock, be sure to also modify the definition in pclock.h.

(2) io_set_uram() (uram.c)

This function initializes the access rights to the on-chip RAM. This sample initializes them as shown in table 16.4.

Table 16.4 Initialization of On-Chip RAM Access Rights

RAM Page	Access from CPUID#1	Access from CPUID#2	Access from DMAC
RAM0 page 0	Readable/Writable	Readable/Writable	Readable/Writable
RAM0 page 1	Readable/Writable	Read-only	Readable/Writable
RAM0 page 2	Readable/Writable	Read-only	Readable/Writable
RAM0 page 2	Readable/Writable	Read-only	Readable/Writable
RAM1 page 0	Read-only	Readable/Writable	Readable/Writable
RAM1 page 1	Read-only	Readable/Writable	Readable/Writable

(3) io_init_bsc_cs0() (bsc_cs0.c)

This function makes the settings for the pin function controller (PFC) and bus state controller (BSC) to specify the timing of the access to the flash memory in the CS0 space.

(4) io_init_sdram() (bscsdram.c)

This function makes the settings for the pin function controller (PFC) and bus state controller (BSC) to enable the SDRAM space in the SDRAM0 space.

16.3.5 Virtual Reset Vector Table for CPUID#2 (cpuid2\reset\vreset.src)

The virtual reset vector table is dedicated for CPUID#2 and is referred to from _Reset_Poweron and _Reset_Manual in reset.src for CPUID#1.

The symbol name of the table in assembly language is _ResetVectorTable_CPUID2, and the section name is CC_vresetvct.

This table is linked to the CPUID#2 side but is referred to from reset.src linked to the CPUID#1 side. Therefore, the following steps are necessary.

(1) The address where the virtual reset vector table is to be allocated should be determined in advance.
(2) At linkage on the CPUID#1 side, the _ResetVectorTable_CPUID2 symbol should be forcibly defined to be the address determined in step (1).
(3) At linkage on the CPUID#2 side, the CC_vresetvct section should be allocated to the address determined in step (1).

Table 16.5 shows the contents registered in the reset vector table.

Table 16.5 Virtual Reset Vector Table

Vector No.	Description	Registered Contents
0	Power-on reset PC	PowerON_Reset_PC_CPUID2(): Power-on reset main program
1	Power-on reset SP	End address of on-chip RAM1
2	Manual reset PC	Manual_Reset_PC_CPUID2(): Manual reset main program
3	Manual reset SP	End address of on-chip RAM1

PowerON_Reset_PC_CPUID2() and Manual_Reset_PC_CPUID2() are reset main programs in resetprg2.c for CPUID#2, and are initiated by a branch through a JMP instruction from _Reset_Poweron and _Reset_Manual in reset.src, respectively. When a reset main program is initiated, the stack pointer is initialized according to the respective vector number (1 or 3).

Figure 16.7 shows the source code in vreset.src and its description.

;**	
;* Definition	
;**	
URAMEND_CPUID2 .assign H'FFFA8000 ;* End of URAM1 address (reset stack for CPUID#2)	End address of on-chip RAM1 (initial stack pointer value for CPUID#2)
;**	
;* Virtual reset vector table	Virtual reset vector table
;*	
;* This table is referred by "reset.src".	
;* This is virtual reset vector table for CPUID#2.	
;* The "reset.src" emulates "Reset" by referring this table.	
;* When linking of CPUID#2, do not change the location address of the	
;* "CC_resetvct" section. If changed, CPUID#1 must be re-linked with	
;* changed this symbol address.	
;**	
.section CC_vresetvct, data, align=4	Section name = CC_vresetvct
.export _ResetVectorTable_CPUID2	External reference declaration for the virtual reset vector table symbol
.import _PowerON_Reset_PC_CPUID2	External reference definition of PowerON_Reset_PC_CPUID2()
.import _Manual_Reset_PC_CPUID2	External reference definition of Manual_Reset_PC_CPUID2()
_ResetVectorTable_CPUID2:	
;* 0 : Power-on Reset (PC)	Power-on reset PC = PowerON_Reset_PC_CPUID2()
.data.l _PowerON_Reset_PC_CPUID2 ; in resetprg.c	
;* 1 : Power-on Reset (SP)	Power-on reset SP = End address of on-chip RAM1
.data.l URAMEND_CPUID2	
;* 2 : Manual Reset (PC)	Manual reset PC = Manual_Reset_PC_CPUID2()
.data.l _Manual_Reset_PC_CPUID2 ; in resetprg.c	
;* 3 : Manual Reset (PC)	Manual reset SP = End address of on-chip RAM1
.data.l URAMEND_CPUID2	
.end	

Figure 16.7 cpuid2\reset\vreset.src

16.3.6 Reset Main Program for CPUID#2 (cpuid2\reset\resetprg2.c)

The following two reset main programs are necessary for CPUID#2.

- For power-on reset: PowerON_Reset_PC_CPUID2()
- For manual reset: Manual_Reset_PC_CPUID2()

Note that the common hardware resources are initialized by CPUID#1; after the initialization is completed, this function is called in the CPUID#2.

(1) PowerON_Reset_PC_CPUID2()

This function is initiated through a branch from _Reset_Poweron in reset.src when a power-on reset is generated during CPUID#2 execution. This function must not return.

This function performs the following processing.

1. Initializes the CPUID#2 hardware (calls HardwareSetup_CPUID2()).
2. Initializes the section (calls _INITSCT()).
3. Initializes the cache (calls sh2adual_ini_cac()).[4]
4. Starts the kernel (calls vsta_knl).

Note that the standard libraries are initialized by the initial startup task because the kernel functions may be required to initialize the standard libraries in some cases; for example, when a reentrant library is used.

(2) Manual_Reset_PC_CPUID2()

This function is initiated through a branch from _Reset_Manual in reset.src when a manual reset is generated during CPUID#2 execution. This function must not return.

[4] The cache can be initialized before this, but note the following case.
For the section initialization processing involving a program section copy from ROM to RAM, if the cache is initialized in copy-back mode before section initialization, the contents of the operand cache corresponding to the program code area to which the program has been copied should be copied back to the actual memory after section initialization. Otherwise, an illegal instruction code will be executed if instructions are fetched from the program code address before the program code that has been written to the operand cache during section initialization is copied back to the actual memory through cache entry replacing.

This function performs almost the same processing as PowerON_Reset_PC_CPUID2().

Figure 16.8 shows the source code in resetprg2.c and its description.

/***	
Includes	
***/	
#include "types.h"	
#include "kernel.h"	
#include "sh2adual_cache.h"	
#include "io_multicore.h"	
#include "initsct.h"	
/***	
Prototypes	
***/	
void PowerON_Reset_PC_CPUID2(void);	
void Manual_Reset_PC_CPUID2(void);	
/***	
External reference	
***/	
extern void HardwareSetup_CPUID2(void);	
/***	
Section definition	
***/	
#pragma section C_reset	Specifies the section name.

Figure 16.8 cpuid2\reset\resetprg2.c

```
/******************************************************************************
 Initialize section information
******************************************************************************/
/*** D section information table ***/
static const ST_DTBL dtbl[]= {
    MACRO_ENTRY_DTBL("DC_stdlib", "RC_stdlib")
};
/*** B section information table ***/
static const ST_BTBL btbl[]= {
    MACRO_ENTRY_BTBL("BC_stdlib"),
    MACRO_ENTRY_BTBL("BC_sample"),
    MACRO_ENTRY_BTBL("BC_heap")
};

/******************************************************************************/
/**  Power-on Reset function
 *    @retval None
 *    @note This routine is called from "_Reset_Poweron" in "reset.src",
 *          and must not return.
******************************************************************************/
#pragma noregsave(PowerON_Reset_PC_CPUID2)
void PowerON_Reset_PC_CPUID2(void)
{
    extern UINT8    s_ucIsInitKnlCPUID1;    /* defined in reset.src */

    set_imask(15);                  /* set SR.IMASK = 15 */

    /*** check CPUID ***/
    if(IO_CPUIDR.BIT._ID != (MYCPUID)-1U) {
        while(1) {          /* error */
        };
    }

```

Section initialization information table

PowerON_Reset_PC_CPUID2()

The registers do not need to be saved because execution does not return.

Stops processing when the unexpected CPU is in execution.

Figure 16.8 cpuid2\reset\resetprg2.c (cont)

```
    /*** initialize CPUID#2 hardware ***/
    HardwareSetup_CPUID2();

    /*** initialize section ***/
    _INITSCT(dtbl, sizeof dtbl, btbl, sizeof btbl);

    /*** initialize cache ***/
    sh2adual_ini_cac(TCAC_IC_ENABLE|TCAC_OC_ENABLE);

    /*** wait for s_ucIsInitKnlCPUID1=1 ***/
    while(s_ucIsInitKnlCPUID1 == 0) {
    }

    /*** start kernel (never return) ***/
    vsta_knl();

    /*** (NEVER return from vsta_knl()) ***/
    while(1) {
    }
}

/*****************************************************************************/
/** Manual Reset function
 *    @retval None
 *    @note This routine is called from "_Reset_Manual" in "reset.src",
 *         and must not return.
******************************************************************************/
#pragma noregsave(Manual_Reset_PC_CPUID2)
void Manual_Reset_PC_CPUID2(void)
{
    extern UINT8    s_ucIsInitKnlCPUID1;    /* defined in reset.src */

    set_imask(15);                /* set SR.IMASK = 15 */

```

Initializes the CPUID#2 hardware.

Initializes the section.

Initializes the cache.

Waits until CPUID#1 sets s_ucIsInitKnlCPUID1 to 1 (C21 in figure 16.11).

Starts the kernel (do not return control here).

ManualReset_PC_CPUID2()

Figure 16.8 cpuid2\reset\resetprg2.c (cont)

/*** initialize CPUID#2 hardware ***/	Initializes the CPUID#2 hardware
HardwareSetup_CPUID2();	
/*** initialize section ***/	Initializes the section.
_INITSCT(dtbl, sizeof dtbl, btbl, sizeof btbl);	
/*** initialize cache ***/	Initializes the cache.
sh2adual_ini_cac(TCAC_IC_ENABLE|TCAC_OC_ENABLE);	
/*** wait for s_ucIsInitKnlCPUID1=1 ***/	Waits until CPUID#1 sets s_ucIsInitKnlCPUID1 to 1 (C21 in figure 16.11).
while(s_ucIsInitKnlCPUID1 == 0) {	
}	
/*** start kernel (never return) ***/	Starts the kernel (do not return control here).
vsta_knl();	
/*** (NEVER return from vsta_knl()) ***/	
while(1) {	
}	
}	

Figure 16.8 cpuid2\reset\resetprg2.c (cont)

16.3.7 CPUID#1 Initial Startup Task InitTask1() (cpuid1\init\init_task1.c)

InitTask1() is registered in the cfg file for CPUID#1 as the first task to be executed in CPUID#1.

This task initializes the following.

1. Standard libraries (_INIT_LOWLEVEL() and _INIT_OTHERLIB() calls)
2. IPI (IPI_init() call)
3. OAL (OAL_Init() call)
4. RPC (rpc_init() call)
   The IPI and OAL initialization should be completed before an rpc_init() call.
5. Remote service call environment (vini_rmt call)
   The IPI initialization should be completed before a vini_rmt call.
6. Sample RPC client (SampleInit() call)

Figure 16.9 shows the source code in InitTask1() and its description.

`/*****************************************************************************`	
` Includes`	
` *****************************************************************************/`	
`#include "types.h"`	
`#include "kernel.h"`	
`#include "kernel_id.h"`	
`#include "rpc_public.h"`	
`#include "oal.h"`	
`#include "ipi.h"`	
`#include "lowsrc.h"`	
`#include "otherlib.h"`	
`#include "rpc_sample.h"`	
`/*****************************************************************************`	
` Prototypes`	
` *****************************************************************************/`	
`void InitTask1(VP_INT exinf);`	
`/*****************************************************************************`	
` Defines`	
` *****************************************************************************/`	
`#define NUM_SERVER  10UL`	Number of servers specified by rpc_init() (actually, no server is registered).
`#define RPCSERVER_STKSZ     0x200UL`	Stack size for the server tasks specified by rpc_init()
`#define RPCSERVER_IPIPORT   2UL     /* interrupt level = 13 */`	ID of the IPI port specified by rpc_init()
`/*****************************************************************************`	
` Data`	
` *****************************************************************************/`	

Figure 16.9   cpuid1\init\init_task1.c

#pragma section L_sample	Creates rpc_info[] to be passed to rpc_init() in a non-cacheable area.
static rpc_info RpcInfo[NUM_SERVER];	
/**	
Section definition	
**/	
#pragma section C_sample	Specifies the section name.
/**/	
/** Initial task	Initial startup task (InitTask1())
* This task calls various API to initialize OS,	
* and then notifies CPUID#2 to have completed initialization phase of CPUID#1	
* by setting s_ucIsInitEnvCPUID1.	
* After that, this task waits to complete the initialization phase of CPUID#2.	
* Afterwards, this task exits and be deleted.	
* <br>This task is created and activated by "task[]" definition in .cfg file.	
* @param exinf Undefined	
* @retval None	
**/	
void InitTask1(VP_INT exinf)	
{	
extern UINT8 s_ucIsInitKnlCPUID1; /* defined in reset.src */	External reference of s_ucIsInitKnlCPUID1
extern UINT8 s_ucIsInitEnvCPUID1; /* defined in reset.src */	External reference of s_ucIsInitEnvCPUID1
extern UINT8 s_ucIsInitEnvCPUID2; /* defined in reset.src */	External reference of s_ucIsInitEnvCPUID2
static const rpc_config ConfigInfo = {	rpc_config structure to be passed to rpc_init()
RpcInfo, /* rpc_info *pRpcTable; */	Pointer to the rpc_info structure array
NUM_SERVER, /* UINT32 ulTableSize; */	Number of servers that can be registered

Figure 16.9   cpuid1\init\init_task1.c (cont)

60	0UL, /* UINT32 ulCmdRspRangeBaseValue; */	Reserved member
61	0UL, /* UINT32 RedirectionTaskStackSize; */	Reserved member
62	RPCSERVER_STKSZ, /* UINT32 ServerTaskStackSize; */	Stack size for the server task
63	0UL, /* UINT32 MFIFramePriority; */	Reserved member
64	1UL, /* UINT32 RPCTaskPriority; */	Reserved member
65	RPCSERVER_IPIPORT /* UINT32 ulIPIPortID; */	ID of the IPI port used by RPC (the interrupt level should not exceed the kernel interrupt mask level (system.system_IPL))
66	};	
67		
68	/*** disable dispatch ***/	Disables dispatch.
69	dis_dsp();	
70		
71	/*** set s_ucIsInitKnlCPUID1 ***/	Sets s_ucIsInitKnlCPUID1 to 1 (C11 in figure 16.11)
72	s_ucIsInitKnlCPUID1 = 1;	
73		
74	/*** initialize standard library ***/	Initializes the low-level interface routines in the standard library.
75	if(_INIT_LOWLEVEL() != 1) {	
76	while(1){ /* error */	
77	}	
78	}	
79		
80	_INIT_OTHERLIB();	Initializes _s1ptr and rand().
81		
82	/*** initialize IPI ***/	Initializes IPI.
83	IPI_init();	
84		
85	/*** initialize OAL ***/	Initializes OAL.
86	OAL_Init();	
87		
88	/*** initialize RPC ***/	Initializes the RPC library.
89	rpc_init(&ConfigInfo);	
90		
91	/*** initialize remote-SVC ***/	Initializes the remote service call environment.
92	vini_rmt();	

Figure 16.9 cpuid1\init\init_task1.c (cont)

93		
94	/*** set ucIsInitializedCPUID1 ***/	Sets s_ucIsInitEnvCPUID1 to 1 (D11 in figure 16.11)
95	s_ucIsInitEnvCPUID1 = 1;	
96		
97	/*** wait for s_ucIsInitEnvCPUID2=1 ***/	Waits until CPUID#2 sets s_ucIsInitEnvCPUID2 to 1 (E11 in figure 16.11).
98	while(s_ucIsInitEnvCPUID2 == 0) {	
99	}	
100		
101	/*** enable dispatch ***/	Enables dispatch.
102	ena_dsp();	
103		
104	/*** connect to sample RPC server ***/	Initializes the sample RPC client.
105	SampleInit();	
106		
107	/*** exit and delete this task ***/	Exits and deletes the initial startup task.
108	exd_tsk();	
109	}	

Figure 16.9 cpuid1\init\init_task1.c (cont)

16.3.8 CPUID#2 Initial Startup Task InitTask2() (cpuid2\init\init_task2.c)

InitTask2() is registered in the cfg file for CPUID#2 as the first task to be executed in CPUID#2.

This task initializes the following.

1. Standard libraries (_INIT_LOWLEVEL() and _INIT_OTHERLIB() calls)
2. IPI (IPI_init() call)
3. OAL (OAL_Init() call)
4. RPC (rpc_init() call)
   The IPI and OAL initialization should be completed before an rpc_init() call.
5. Remote service call environment (vini_rmt call)
   The IPI initialization should be completed before a vini_rmt call.
6. Sample RPC server (SampleInit() call)

Figure 16.10 shows the source code in InitTask2() and its description.

```
/****************************************************************************
 Includes
 ****************************************************************************/
#include "types.h"
#include "kernel.h"
#include "kernel_id.h"

#include "rpc_public.h"
#include "oal.h"
#include "ipi.h"

#include "lowsrc.h"
#include "otherlib.h"

#include "rpc_sample.h"

/****************************************************************************
 Prototypes
 ****************************************************************************/
void InitTask2(VP_INT exinf);

/****************************************************************************
 Defines
 ****************************************************************************/
#define NUM_SERVER  10UL                                          Number of servers specified by
                                                                  rpc_init()
#define RPCSERVER_STKSZ     0x200UL                               Stack size for the server tasks
                                                                  specified by rpc_init()
#define RPCSERVER_IPIPORT   2UL     /* interrupt level = 13 */    ID of the IPI port specified by
                                                                  rpc_init()


/****************************************************************************
 Data
 ****************************************************************************/
```

Figure 16.10   cpuid2\init\init_task2.c

32	#pragma section L_sample	Creates rpc_info[] to be passed to rpc_init() in a non-cacheable area.
33	static rpc_info RpcInfo[NUM_SERVER];	
34		
35	/***	
36	Section definition	
37	***/	
38	#pragma section C_sample	Specifies the section name.
39		
40	/***/	
41	/** Initial task	Initial startup task (InitTask2())
42	* At first, this task waits to complete the initialization phase of CPUID#1.	
43	* and then calls various API to initialize OS.	
44	* After that, this task notifies CPUID#1 to have completed initialization	
45	* phase of CPUID#2 by setting s_ucIsInitializedCPUID2.	
46	* Afterwards, this task exits and be deleted.	
47	* <br>This task is created and activated by "task[]" definition in .cfg file.	
48	* @param exinf Undefined	
49	* @retval None	
50	***/	
51	void InitTask2(VP_INT exinf)	
52	{	
53	extern UINT8 s_ucIsInitEnvCPUID1; /* defined in reset.src */	External reference of s_ucIsInitEnvCPUID1
54	extern UINT8 s_ucIsInitEnvCPUID2; /* defined in reset.src */	External reference of s_ucIsInitEnvCPUID2
55		
56	static const rpc_config ConfigInfo = {	rpc_config structure to be passed to rpc_init()
57	RpcInfo, /* rpc_info *pRpcTable; */	Pointer to the rpc_info structure array
58	NUM_SERVER, /* UINT32 ulTableSize; */	Number of servers that can be registered
59	0UL, /* UINT32 ulCmdRspRangeBaseValue; */	Reserved member
60	0UL, /* UINT32 RedirectionTaskStackSize; */	Reserved member
61	RPCSERVER_STKSZ, /* UINT32 ServerTaskStackSize; */	Stack size for the server task

Figure 16.10   cpuid2\init\init_task2.c (cont)

0UL, /* UINT32 MFIFramePriority; */	Reserved member
1UL, /* UINT32 RPCTaskPriority; */	Reserved member
RPCSERVER_IPIPORT /* UINT32 ulIPIPortID; */ };	ID of the IPI port used by RPC (the interrupt level should not exceed the kernel interrupt mask level (system.system_IPL))
/*** disable dispatch ***/ dis_dsp();	Disables dispatch.

/*** wait for s_ucIsInitEnvCPUID1=1 ***/
while(s_ucIsInitEnvCPUID1 == 0) {
}
	Waits until CPUID#1 sets s_ucIsInitEnvCPUID to 1 (D21 in figure 16.11).

/*** initialize standard library ***/
if(_INIT_LOWLEVEL() != 1) {
while(1){ /* error */
}
}
	Initializes the low-level interface routines in the standard library.

_INIT_OTHERLIB();	Initializes _s1ptr and rand().
/*** initialize IPI ***/ IPI_init();	Initializes IPI.
/*** initialize OAL ***/ OAL_Init();	Initializes OAL.
/*** initialize RPC ***/ rpc_init(&ConfigInfo);	Initializes the RPC library.
/*** initialize remote-SVC ***/ vini_rmt();	Initializes the remote service call environment.

Figure 16.10 cpuid2\init\init_task2.c (cont)

94	/*** enable dispatch ***/	Enables dispatch.
95	ena_dsp();	
96		
97	/*** start sample RPC server ***/	Initializes the sample RPC server
98	SampleInit();	
99		
100	/*** set s_ucIsInitEnvCPUID2 ***/	Sets the s_ucIsInitEnvCPUID2 flag (E21 in figure 16.11).
101	s_ucIsInitEnvCPUID2 = 1;	
102		
103	/*** exit and delete this task ***/	Exits and deletes the initial startup task.
104	exd_tsk();	
105	}	

Figure 16.10 cpuid2\init\init_task2.c (cont)

16.3.9 Synchronization of Startup Phases in Two CPUs

This sample uses the four flags described below to enable each CPU to check the progress of processing in the other CPU during initialization. These flags are defined in reset.src linked to the CPUID#1 side. As these flags need to be accessed before the initialization process required for external RAM access, they are placed in on-chip RAM0, which does not need to be initialized and can be accessed from both CPUs.

The following describes the role of each flag. Figure 16.11 shows the synchronization of startup phases with a focus on these four flags.

(1) s_ucIsInitHW: Common hardware resource initialization variable

In this sample, CPUID#1 initializes the common hardware resources. CPUID#2 executes no processing before the common hardware resources are initialized because it cannot perform an operation such as SDRAM access in this state.

When CPUID#1 sets the bus state controller to initialize the CS0 space, CPUID#2 must not access CS0 according to the bus state controller specification. Therefore, CPUID#2 is placed in the busy-wait state in the on-chip RAM1 until CPUID#1 completes CS0 initialization.

s_ucIsInitHW is a variable used to control the order of processing as described above.

The meaning of s_ucIsInitHW is as follows.

- 0: Initial state
- 1: CPUID#1 can initialize the common hardware resources (CPUID#2 is in busy-wait state in the on-chip RAM)
- 2: CPUID#1 has completed the initialization of the common hardware resources

(2) s_ucIsInitKnlCPUID1: CPUID#1 kernel initialization flag

s_ucIsInitKnlCPUID1 is a flag that indicates that CPUID#1 has completed the kernel initialization.

This kernel has a restriction that serial numbers cannot be acquired as the trace numbers if CPUID#2 issues a service call before CPUID#1 completes the kernel initialization. In this sample, s_ucIsInitKnlCPUID1 is used to control CPUID#2 so that CPUID#2 does not issue vsta_knl until CPUID#1 completes the kernel initialization (until control moves to the initial startup task).

The meaning of s_ucIsInitKnlCPUID1 is as follows.

- 0: Initial state
- 1: CPUID#1 has completed the kernel initialization

(3) s_ucIsInitEnvCPUID1: CPUID#1 software environment initialization flag

The HI7200/MP has the following restrictions on the order of APIs related to initialization.

- IPI_init() must not be issued in CPUID#2 before completion of IPI_init() in CPUID#1.
- rpc_init() must not be issued in CPUID#2 before completion of vini_rmt in CPUID#1.
- vini_rmt must not be issued in CPUID#2 before completion of vini_rmt in CPUID#1.

In addition, the order of processing in the application may have similar restrictions (this situation does not apply to this sample).

s_ucIsInitEnvCPUID1 is a flag to control the order of processing as described above. In particular, this flag is used to control CPUID#2 so that CPUID#2 calls the above initialization APIs only after the initial startup task in the CPUID#1 completes the above initialization APIs.

The meaning of s_ucIsInitEnvCPUID1 is as follows.

- 0: Initial state
- 1: CPUID#1 has completed the initialization of various software environments

(4) s_ucIsInitEnvCPUID2: CPUID#2 software environment initialization flag

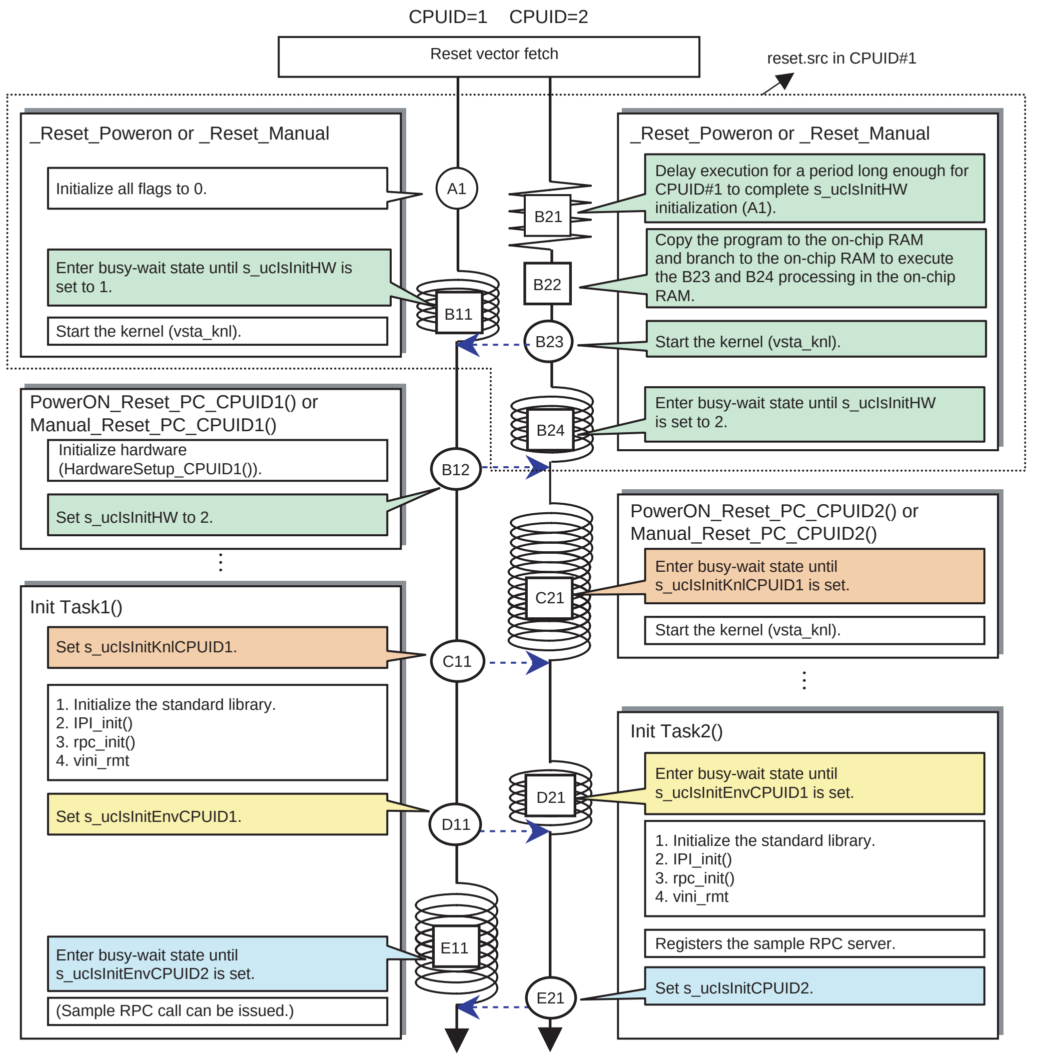
When the RPC is used, the RPC servers should be registered before RPC calls.

In addition, the order of processing in the application may have similar restrictions (this situation does not apply to this sample).

s_ucIsInitEnvCPUID2 is a flag to control the order of processing as described above. In particular, this flag is used to control CPUID#1 so that CPUID#1 does not issue RPC calls before the initial startup task in CPUID#2 completes registration of the sample RPC server.

The meaning of s_ucIsInitEnvCPUID2 is as follows.

- 0: Initial state
- 1: CPUID#2 has completed the initialization of various software environments


CPUID=1
CPUID=2
Reset vector fetch
reset.src in CPUID#1
_Reset_Poweron or _Reset_Manual
Initialize all flags to 0.
Enter busy-wait state until s_ucIsInitHW is set to 1.
Start the kernel (vsta_knl).
A1
B11
B21
B22
B23
B24
B12
C11
C21
D11
D21
E11
E21
_Reset_Poweron or _Reset_Manual
Delay execution for a period long enough for CPUID#1 to complete s_ucIsInitHW initialization (A1).
Copy the program to the on-chip RAM and branch to the on-chip RAM to execute the B23 and B24 processing in the on-chip RAM.
Start the kernel (vsta_knl).
Enter busy-wait state until s_ucIsInitHW is set to 2.
PowerON_Reset_PC_CPUID1() or Manual_Reset_PC_CPUID1()
Initialize hardware (HardwareSetup_CPUID1()).
Set s_ucIsInitHW to 2.
PowerON_Reset_PC_CPUID2() or Manual_Reset_PC_CPUID2()
Enter busy-wait state until s_ucIsInitKnlCPUID1 is set.
Start the kernel (vsta_knl).
Init Task1()
Set s_ucIsInitKnlCPUID1.
1. Initialize the standard library.
2. IPI_init()
3. rpc_init()
4. vini_rmt
Set s_ucIsInitEnvCPUID1.
Enter busy-wait state until s_ucIsInitEnvCPUID2 is set.
(Sample RPC call can be issued.)
Init Task2()
Enter busy-wait state until s_ucIsInitEnvCPUID1 is set.
1. Initialize the standard library.
2. IPI_init()
3. rpc_init()
4. vini_rmt
Registers the sample RPC server.
Set s_ucIsInitCPUID2.

Legend: Green: Access to s_ucIsInitHW
Orange: Access to s_ucIsInitKnlCPUID1
Yellow: Access to s_ucIsInitEnvCPUID1
Blue: Access to s_ucIsInitEnvCPUID2
A1, B11, B21, B22, B23, and B24: See figure 16.5, cpuid1\reset\reset.src.
B12: See figure 16.6, cpuid1\reset\resetprg1.c.
C21: See figure 16.8, cpuid2\reset\resetprg2.c.
C11, D11, and E11: See figure 16.9, cpuid1\init\init_task1.c.
D21 and E21: See figure 16.10, cpuid2\init\init_task2.c.

Figure 16.11 Synchronization of Startup Phases

16.4 Example of RPC Usage

16.4.1 Overview

In the provided example of RPC usage, CPUID#1 is the client and CPUID#2 is the server. This section describes how to pass parameters.

Table 16.6 shows the server functions.

Table 16.6 Server Functions

No.	Server Function Name	Function Specification
1	INT32 SampleAdd (INT32 lPar1, INT32 lPar2)	Adds lPar1 and lPar2 and returns the result.
2	UINT32 SampleStrlen (INT8 * pString)	Returns the length of the character string indicated by pString.
3	void SampleSort1(INT32 * pData)	Sorts the 10-element array indicated by pData.
4	void SampleSort2(INT32 * pData)	Sorts the 10-element array indicated by pData.
5	void SampleMemcopy(void *pDest, const void *pSrc, UINT32 ulSize)	Copies ulSize-byte data from pSrc to pDest.
6	INT32 SampleCreateTask(void *entry , INT32 lPriority, UINT32 ulStackSize, void *UserData)	Creates and starts a task in the server CPU (without the TA_COP1 attribute).
7	INT32 SampleKillTask(INT32 lTaskID)	Forcibly terminates and deletes the task in the server CPU.
8	INT32 SampleRefTaskState (INT32 lTaskID, UINT32 *pulState)	Refers to the state of the task in the server CPU.

Table 16.7 shows the source files described in this section.

Table 16.7 Source Files for RPC Usage Example

Directory	File	Description
include\	rpc_sample.h	Defines the APIs for initializing (SampleInit()) and terminating (SampleShutdown()) the RPC server and client.
	sample_add.h	Defines the API for server function SampleAdd().
	sample_strlen.h	Defines the API for server function SampleStrlen().
	sample_sort.h	Defines the APIs for server functions SampleSort1() and SampleSort2().
	sample_memcopy.h	Defines the API for server function SampleMemcopy().
	sample_svc.h	Defines the APIs for server functions SampleCreateTask (), SampleKillTask(), and SampleRefTaskState().
cpuid1\ include\	rpc_sample_clnt.h	Defines client API SampleGetLastRPCErr().
cpuid1\ rpc_sample_clnt\	rpc_sample_clnt.c	RPC client stubs • SampleAdd() • SampleStrlen() • SampleSort1() • SampleSort2() • SampleMemcopy() • SampleCreateTask() • SampleKillTask() • SAMPLE_SampleRefTaskState() Client initialization API function: SampleInit() Client termination API function: SampleShutdown()
	rpc_sample_private.h *	Private header for RPC client and server stubs
cpuid1\ rpc_caller\	rpc_caller.c	Function (task) for issuing an RPC call

Table 16.7 Source Files for RPC Usage Example (cont)

Directory	File	Description
cpuid2\ rpc_sample_svr\	rpc_sample_svr.c	RPC server stubs • rpcsvr_SAMPLE_SampleAdd() • rpcsvr_SAMPLE_SampleStrlen() • rpcsvr_SAMPLE_SampleSort1() • rpcsvr_SAMPLE_SampleSort2() • rpcsvr_SAMPLE_SampleMemcopy() • rpcsvr_SAMPLE_SampleCreateTask() • rpcsvr_SAMPLE_SampleKillTask() • rpcsvr_SAMPLE_SampleRefTaskState() Server initialization API function: SampleInit() Server termination API function: SampleShutdown()
	rpc_sample_private.h *	Private header for RPC client and server stubs
	sample_add.c	Server function SampleAdd()
	sample_strlen.c	Server function SampleStrlen()
	sample_sort.c	Server functions SampleSort1()" and SampleSort2(), and internal function SampleSortMain()
	sample_memcopy.c	Defines the API for server function SampleMemcopy().
	sample_svc.c	Server functions SampleCreateTask (), SampleKillTask(), and SampleRefTaskState()

Note: * These files have the same contents.

16.4.2 Registration of RPC Servers (CPUID#2)

In this example, each server function (corresponding server stub) described above is registered as a single server.

A server is registered by calling rpc_start_server() in server initialization API SampleInit(). SampleInit() is called by the initial startup task.

16.4.3 SampleAdd()

SampleAdd() has two INT32-type input parameters: lPar1 and lPar2.

The client stub prepares two IOVECs and sets the input address and size parameters in the IOVECs. This example uses two IOVECs for ease of comprehension; it is more efficient to use only one IOVEC.

The input parameters are stored in the area indicated by pInfo->pucParamArea by being aligned with 4-byte boundaries in the order of server function APIs; the server stub reads and passes them to the server functions.

Figures 16.12 and 16.13 respectively show the source codes of the client stub and server stub.

```
/**********************************************************************************/
/** Client stub of "SampleAdd()"
 *    @param lPar1  data1 to be added
 *    @param lPar2  data2 to be added
 *    @retval Result
**********************************************************************************/
INT32 SampleAdd ( INT32 lPar1, INT32 lPar2)
{
    UINT32          ulLastOutputIOVectorSize;
    rpc_call_info   info;
    IOVEC           input[2];                                                          Prepares two input IOVECs
    UINT32          ulReturn;
    INT32           lRPCRet;

    info.ulMarshallingType          = 0UL;
    info.ulServerID                 = SV_ID_SAMPLE;
    info.ulServerVersion            = SV_VER_SAMPLE;
    info.ulServerProcedureID        = RPC_SAMPLE_SAMPLEADD;
    info.AckMode                    = RPC_ACK;
    info.pInputIOVectorTable        = input;
    info.ulInputIOVectorTableSize   = sizeof(input) / sizeof (IOVEC);
    info.pOutputIOVectorTable       = NULL;                                            No output
    info.ulOutputIOVectorTableSize = 0UL;
    info.pulLastOutputIOVectorSize = &ulLastOutputIOVectorSize;
    info.pulReturnValue             = &ulReturn;                                       Return value setting area

    input[ 0 ].pBaseAddress = &lPar1;                                                  Sets lPar1 information.
    input[ 0 ].ulSize = sizeof(INT32);

    input[ 1 ].pBaseAddress = &lPar2;                                                  Sets lPar2 information.
    input[ 1 ].ulSize = sizeof(INT32);

```

Figure 16.12 SampleAdd() (Client Stub)

33	lRPCRet = rpc_call(&info);	RPC call
34	if(lRPCRet != RPC_E_OK) {	Sets the error code in lLastErr and returns control with a return value of 0 if the RPC call generates an error.
35	lLastErr = lRPCRet;	
36	ulReturn = 0UL;	
37	}	
38		
39	return ((INT32)ulReturn);	
40	}	

Figure 16.12 SampleAdd() (Client Stub) (cont)

1	/**/	
2	/** Server stub of "SampleAdd()"	
3	* @param pInfo pointer to rpc_server_stub info structure	
4	* @retval Return value of server function	
5	***/	
6	static UINT32 rpcsvr_SAMPLE_SampleAdd (rpc_server_stub_info *pInfo)	
7	{	
8	INT32 lPar1;	
9	INT32 lPar2;	
10	INT32 lReturn;	
11		
12	lPar1 = *(INT32 *)(pInfo->pucParamArea);	Acquires lPar1 from the server parameter area.
13	lPar2 = *(INT32 *)(pInfo->pucParamArea + sizeof(INT32));	Acquires lPar2 from the server parameter area.
14		
15	lReturn = SampleAdd (lPar1, lPar2);	Calls the server function.
16		
17	pInfo->ulOutputIOVectorTableSize = 0UL;	No output
18		
19	return ((UINT32)lReturn);	
20	}	

Figure 16.13 rpcsvr_SAMPLE_SampleAdd() (Server Stub)

16.4.4 SampleStrlen()

SampleStrlen() is an example that inputs a pointer.

SampleStrlen() has an INT8*-type input parameter: pString.

The client stub prepares an IOVEC, and sets pString and the length of its character string in the IOVEC.

The character string is stored in the area indicated by pInfo->pucParamArea; the server stub reads and passes it to the server function.

Figures 16.14 and 16.15 respectively show the source codes of the client stub and server stub.

```
/************************************************************************************/
/** Client stub of "SampleStrlen()"
 *    @param  pString  pointer to the string to be counted
 *    @retval length of the string
************************************************************************************/
UINT32 SampleStrlen ( INT8 * pString)
{
    UINT32          ulLastOutputIOVectorSize;
    rpc_call_info   info;
    IOVEC           input[1];
    UINT32          ulReturn;
    INT32           lRPCRet;

    info.ulMarshallingType          = 0UL;
    info.ulServerID                 = SV_ID_SAMPLE;
    info.ulServerVersion            = SV_VER_SAMPLE;
    info.ulServerProcedureID        = RPC_SAMPLE_SAMPLESTRLEN;
    info.AckMode                    = RPC_ACK;
    info.pInputIOVectorTable        = input;
```

Prepares an IOVEC.

Figure 16.14 SampleStrlen() (Client Stub)

info.ulInputIOVectorTableSize = sizeof(input) / sizeof (IOVEC);	
info.pOutputIOVectorTable = NULL;	No output
info.ulOutputIOVectorTableSize = 0UL;	
info.pulLastOutputIOVectorSize = &ulLastOutputIOVectorSize;	
info.pulReturnValue = &ulReturn;	Return value setting area
input[0].pBaseAddress = pString;	Sets pString.
input[0].ulSize = strlen(pString) + 1UL;	This strlen() is a standard library call; it is not part of SampleStrlen().
lRPCRet = rpc_call(&info);	RPC call
if(lRPCRet != RPC_E_OK) {	Sets the error code in lLastErr and returns control with a return value of 0 if the RPC call generates an error.
lLastErr = lRPCRet;	
ulReturn = 0UL;	
}	
return ulReturn;	
}	

Figure 16.14   SampleStrlen() (Client Stub) (cont)

```
/************************************************************************************/
/** Server stub of "SampleStrlen()"
 *    @param pInfo  pointer to rpc_server_stub info structure
 *    @retval Return value of server function
***********************************************************************************/
static UINT32 rpcsvr_SAMPLE_SampleStrlen (rpc_server_stub_info *pInfo)
{
   INT8 * pString;
   UINT32 ulReturn;
                                                        Acquires pString from the server
   pString = (INT8 *)(pInfo->pucParamArea);             parameter area.

   ulReturn = SampleStrlen (pString);                   Calls the server function.

   pInfo->ulOutputIOVectorTableSize = 0UL;              No output

   return (ulReturn);
}
```

Figure 16.15 rpcsvr_SAMPLE_SampleStrlen() (Server Stub)

16.4.5 SampleSort1() and SampleSort2()

SampleSort1() and SampleSort2() are examples that input a pointer. They work in the same way except that SampleSort1() copies the data indicated by the pointer and passes it to the server while SampleSort2() passes the pointer itself to the server. SampleSort2() is fast because it does not copy data. but it has a restriction that the pointer must always indicate a non-cacheable area.

Read the codes of the provided client stub and server stub with reference to the descriptions in the above sections.

16.4.6 SampleMemcopy()

SampleMemcopy() is also a server-function example that passes the input pointer itself to the server.

16.4.7 SampleCreateTask(), SampleKillTask(), and SampleRefTaskState()

These are examples in which the kernel service calls are extended so that they can be issued also to the kernel in the server through the RPC function.

In terms of the RPC, SampleRefTaskState() is an example that outputs a parameter, which is described below.

SampleRefTaskState() has a UINT32*-type output parameter: pulState.

The client stub prepares an output IOVEC and sets pulState and its size (sizeof(UINT32)) in the IOVEC. The RPC for the client writes data to the area indicated by the output IOVEC and therefore, pulState does not need to indicate a non-cacheable area.

In the server, the RPC appropriately sets pInfo->pOutputIOVectorTable[0] (so that it indicates an address in the server parameter area) and calls the server stub. The server stub acquires pulState from the indicated area and calls the server function.

Figures 16.16 and 16.17 respectively show the source codes of the client stub and server stub.

```
/***************************************************************************/
/** Client stub of "SampleRefTaskState()"
 *    @param  lTaskID  task ID to be referred
 *    @param  pulState pointer to be stored task state.
 *            <br>  0x00000001 ... RUNNING
 *            <br>  0x00000002 ... READY
 *            <br>  0x00000004 ... WAITING
 *            <br>  0x00000008 ... SUSPENDED
 *            <br>  0x0000000C ... WAITING and SUSPENDED
 *            <br>  0x00000010 ... DORMANT
 *            <br>  0x40000000 ... STACK-WAIT
 *    @retval 0(success), Negative value(fail)
***************************************************************************/
INT32 SampleRefTaskState ( INT32 lTaskID, UINT32 *pulState)
{
   UINT32         ulLastOutputIOVectorSize;
   rpc_call_info   info;
   IOVEC          input[1];                                        Prepares an input IOVEC.
   IOVEC          output[1];                                       Prepares an output IOVEC.
   UINT32         ulReturn;
   INT32          lRPCRet;

   info.ulMarshallingType         = 0UL;
   info.ulServerID                = SV_ID_SAMPLE;
   info.ulServerVersion           = SV_VER_SAMPLE;
   info.ulServerProcedureID       = RPC_SAMPLE_SAMPLEREFTASKSTATE;
   info.AckMode                   = RPC_ACK;
   info.pInputIOVectorTable       = input;
   info.ulInputIOVectorTableSize  = sizeof(input) / sizeof (IOVEC);
   info.pOutputIOVectorTable      = output;
   info.ulOutputIOVectorTableSize = sizeof(output) / sizeof (IOVEC);
   info.pulLastOutputIOVectorSize = &ulLastOutputIOVectorSize;
   info.pulReturnValue            = &ulReturn;

```

Figure 16.16 SampleRefTaskState() (Client Stub)

35	input[0].pBaseAddress = &lTaskID;	Sets lTaskID as an input.
36	input[0].ulSize = sizeof(INT32);	
37		
38	output[0].pBaseAddress = pulState;	Sets pulState as an output.
39	output[0].ulSize = sizeof(UINT32);	
40		
41	lRPCRet = rpc_call(&info);	RPC call
42	if(lRPCRet != RPC_E_OK) {	Sets the error code in lLastErr and returns control with a return value of -1 if the RPC call generates an error.
43	lLastErr = lRPCRet;	
44	ulReturn = 0xFFFFFFFFUL; /* return value = -1 */	
45	}	
46		
47	return ((INT32)ulReturn);	
48	}	

Figure 16.16 SampleRefTaskState() (Client Stub) (cont)

```
/********************************************************************************/
/** Server stub of "SampleRefTaskState()"
 *    @param pInfo  pointer to rpc_server_stub info structure
 *    @retval Size actually read
********************************************************************************/
static UINT32 rpcsvr_SAMPLE_SampleRefTaskState (rpc_server_stub_info *pInfo)
{
   INT32 lTaskID;
   UINT32 *pulState;
   INT32 lRet;

   lTaskID = *(INT32 *)(pInfo->pucParamArea);                   Acquires lTaskID from the server
                                                                parameter area.

   pulState = pInfo->pOutputIOVectorTable[0].pBaseAddress;      Acquires pulState.

   lRet = SampleRefTaskState (lTaskID, pulState);               Calls the server function.

   pInfo->ulOutputIOVectorTableSize = 1UL;

   return ((UINT32)lRet);
}
```

Figure 16.17 rpcsvr_SAMPLE_SampleRefTaskState() (Server Stub)

16.4.8 Example of RPC Call (CPUID#1)

TaskRpcCaller() in cpuid1\rpc_caller\rpc_caller.c is an example of issuing provided sample RPC calls in sequence. TaskRpcCaller() is registered as a task in the cfg file.

16.4.9 Initialization and Termination of Servers (CPUID#2)

This RPC usage example provides an API function for initializing the server environment (SampleInit()) and an API function for terminating the server (SampleShutdown()). The function names are the same as those in the client.

In the configuration at shipment, SampleInit() is called by the initial startup task. SampleShutdown() is not used.

SampleInit() registers servers by using rpc_start_server().

SampleShutdown() deletes servers by using rpc_stop_server().

Figure 16.18 shows the source codes of SampleInit() and SampleShutdown().

```
/*********************************************************************************/
/** Initialize RPC-Sample for server-side
 *    @retval Return code of rpc_start_server()
*********************************************************************************/
INT32 SampleInit ( void)
{
    /*** Server stub list ***/
    static UINT32 (* const rpcsvr_SAMPLE_StubTable[])(rpc_server_stub_info*) =      Server stub table
    {
        rpcsvr_SAMPLE_SampleAdd,
        rpcsvr_SAMPLE_SampleStrlen,
        rpcsvr_SAMPLE_SampleSort1,
        rpcsvr_SAMPLE_SampleSort2,
        rpcsvr_SAMPLE_SampleMemcopy,
        rpcsvr_SAMPLE_SampleCreateTask,
        rpcsvr_SAMPLE_SampleKillTask,
        rpcsvr_SAMPLE_SampleRefTaskState
    };

    /*** Server information ***/
    static const rpc_server_info rpcsvr_SAMPLE_ServerInfo =                          Server information
    {
        SV_ID_SAMPLE,                   /* ulRPCServerID */
        SV_VER_SAMPLE,                  /* ulRPCServerVersion */
        1UL,                            /* ServerStubTaskPriority */
        rpcsvr_SAMPLE_StubTable,        /* ServerStubList */
        sizeof(rpcsvr_SAMPLE_StubTable) / sizeof(rpcsvr_SAMPLE_StubTable[0]),
                                        /* ulNumFunctions */
        0x400UL,                        /* ulStubStackSize */
        0UL,                            /* ulMaxParamAreaSize */
        NULL                            /* user_data */
    };

```

Figure 16.18   SampleInit() and SampleShutdown() (In the Server)

```
    return rpc_start_server ( &rpcsvr_SAMPLE_ServerInfo);                     Registers servers.
}

/********************************************************************************/
/** Shutdown RPC-Sample for server-side
 *    @retval Return code of rpc_stop_server()
********************************************************************************/
INT32 SampleShutdown ( void)
{
    return rpc_stop_server( SV_ID_SAMPLE, SV_VER_SAMPLE, NULL, 0UL);          Deletes servers.
}
```

Figure 16.18 SampleInit() and SampleShutdown() (In the Server) (cont)

16.4.10 Initialization and Termination of Clients (CPUID#1)

This RPC sample provides an API function for initializing the client environment (SampleInit()) and an API function for terminating the client (SampleShutdown()).

In the configuration at shipment, SampleInit() is called by the initial startup task. SampleShutdown() is not used.

SampleInit() starts connection with the server by using rpc_connect().

SampleShutdown() terminates the connection with the server by using rpc_disconnect().

Figure 16.19 shows the source codes of SampleInit() and SampleShutdown().

```
/**********************************************************************************/
/** Initialize RPC-Sample for client-side
 *    @retval Return code of rpc_connect()
**********************************************************************************/
INT32 SampleInit ( void)
{
    return rpc_connect( SV_ID_SAMPLE, SV_VER_SAMPLE);                    Connects with the server.

}

/**********************************************************************************/
/** Shutdown RPC-Sample for client-side
 *    @retval Return code of rpc_disconnect()
**********************************************************************************/
INT32 SampleShutdown ( void)
{
    return rpc_disconnect( SV_ID_SAMPLE, SV_VER_SAMPLE, NULL, 0);        Disconnects from the server.
}
/**********************************************************************************/
```

Figure 16.19 SampleInit(), SampleShutdown() (in the Client)

16.4.11 Initialization of RPC Library (rpc_init() Call)

In both the client and server, the initial startup task calls rpc_init().

For the rpc_config structure passed to rpc_init(), see figure 16.9 for the client or figure 16.10 for the server.

16.5 Remote Service Call Example

As an example of remote service call usage, a simple message communication program using a mailbox and a fixed-sized memory pool is provided.

Specifically, in this example, a task in CPUID#1 acquires a message area from the fixed-sized memory pool in CPUID#2 and sends a message to a mailbox in CPUID#2. A task in CPUID#2 receives the message from the mailbox and returns the message area to the fixed-sized memory pool.

The mailbox and fixed-sized memory pool are created according to the cfg file for CPUID#2. In the cfg file, export = ON is specified to output the ID names of the mailbox and memory pool to kernel_id_cpu2.h. The file for CPUID#1 includes this kernel_id_cpu2.h.

As the message is accessed from both CPUs, the fixed-sized memory pool area is allocated to a non-cacheable area.

Table 16.8 shows the source files of the remote service call example.

Table 16.8 Source Files of Remote Service Call Example

Directory	File Name	Function
include\	user_msg.h	Message type definition
cpuid1\ remote_svc_sample\	remote_send.c	Message-sending task (TaskSend())
cpuid2\ remote_svc_sample\	remote_recv.c	Message-receiving task (TaskRecv())

Figures 16.20 and 16.21 respectively show source codes of the message-sending task in CPUID#1 and message-receiving task in CPUID#2.

```
/*******************************************************************************
/** Sample task that send message by using remote service call.
 ******************************************************************************/
void TaskSend(VP_INT exinf)
{
    USER_MSG    *message;
    UINT32      ulIndex;

    for(ulIndex = 0UL ; ; ulIndex++) {
        /* get non-cached message area */
        if(pget_mpf(EXID2_MPF_NONCACHED, (VP *)&message) != E_OK) {
            ext_tsk();
        }

        /* create message */
        message->osarea.msghead = NULL;
        message->ulData = ulIndex;

        /* send message to CPU2 */
        snd_mbx(EXID2_MBX_COMM, (T_MSG *)&message);
    }
}
```

Acquires a message area from a fixed-sized memory pool.

Specifies a message.

Sends the message.

Figure 16.20 Message-Sending Task (cpuid1\remote_svc_sample\sample_send.c)

```
/******************************************************************************
/** Sample task that receives message.
 ******************************************************************************/
void TaskRecv(VP_INT exinf)
{
    USER_MSG    *p_message;
    UINT32      ulIndex;

    while(TRUE) {
        /* receive message from CPU1 */
        if(rcv_mbx(EXID2_MBX_COMM, (T_MSG **)&p_message) != E_OK) {      Receives a message.
            ext_tsk();
        }

        /* do operation according to message */                          Processing according to the
                                                                          message.
        /* release memory */
        rel_mpf(EXID2_MPF_NONCACHED, (VP)p_message);                      Releases the message area.
    }
}
```

Figure 16.21   Message-Receiving Task (cpuid2\remote_svc_sample\sample_recv.c)

16.6 Timer Driver

A timer driver is provided to control the CMT in the SH7205 or SH7265.

CPUID#1 uses channel 0 and CPUID#2 uses channel 1 of the CMT.

Table 16.9 shows the source files of the timer driver.

Table 16.9 Source Files of Timer Driver

Directory	File Name	Function
cpuid1\ os_timer\	tmrdrv.c *[1]	Timer driver for CMT in SH7205 or SH7265
	tmrdrv.h *[2]	Internal definition
cpuid2\ os_timer\	tmrdrv.c *[1]	Timer driver for CMT in SH7205 or SH7265
	tmrdrv.h *[2]	Internal definition

Note: *1. These files have the same contents.
*2 These files have the same contents.

The file contents are the same for both CPUs; they are implemented so that they are conditionally compiled according to MYCPUID.

16.7 Standard Libraries

16.7.1 Overview

In this example, standard libraries are included for both CPUs as follows.

- Included functions: stdlib.h and string.h
- Configured as reentrant libraries

As described in the compiler user's manual, note that when configuring a library as reentrant in the application that uses a standard library, macro name "_REENTRANT" should be defined in a #define statement (#define _REENTRANT) before including standard include files or _REENTRANT should be defined through a define option at compilation. The provided High-performance Embedded Workshop project adopts the latter.

Note also that stdio.h is not included. When using stdio.h, low-level interface routines for that should be added.

Table 16.10 shows the source files related to standard libraries.

Table 16.10 Source Files Related to Standard Libraries

Directory	File Name*	Function
cpuid1\stdlib\	lowsrc.c	Low-level interface routines, _INIT_LOWLEVEL()
	lowsrc_config.h	Configuration file for low-level interface routine
	otherlib.c	_INIT_OTHERLIB()
	initsct.c	_INITSCT()
cpuid1\include\	lowsrc.h	lowsrc.c external header
	initsct.h	initsct.c external header
cpuid2\stdlib\	lowsrc.c	Low-level interface routines, _INIT_LOWLEVEL()
	lowsrc_config.h	Configuration file for low-level interface routine
	otherlib.c	_INIT_OTHERLIB()
	initsct.c	_INITSCT()
cpuid2\include\	lowsrc.h	lowsrc.c external header
	initsct.h	initsct.c external header

Note: * Files with the same name have the same contents.

16.7.2 Low-Level Interface Routines

To use standard I/O or memory management libraries or configure libraries as reentrant, appropriate low-level interface routines should be created.

Table 16.11 shows the low-level interface routines specified in the compiler and the implementation in this sample.

Table 16.11 Low-Level Interface Routines

Low-Level Interface Routines Specified in Compiler	Implementation in This Sample	Function
open()	No	Opens a file
close()	No	Closes a file
read()	No	Reads a file
write()	No	Writes to a file
lseek()	No	Specifies a read/write position in a file
sbrk()	Yes	Allocates a memory area
sbrk__X()	No	Allocates an X memory area (for a microcomputer with a DSP)
sbrk__Y()	No	Allocates a Y memory area (for a microcomputer with a DSP)
errno_adr() *	Yes	Acquires an errno address
wait_sem() *	Yes	Acquires a semaphore
signal_sem() *	Yes	Releases a semaphore

Note: * Required when using a reentrant library.

wait_sem() and signal_sem() are low-level interface routines for exclusive control. In the above table, "semaphore" is a term used for standard library functions in the compiler user's manual and differs from "semaphore" in the HI7200/MP.

wait_sem() and signal_sem() uses the mutex function in the kernel to implement exclusive control.

16.7.3 Initialization of Standard Library Environment (_INIT_LOWLEVEL() and _INIT_OTHERLIB())

This sample provides _INIT_LOWLEVEL() for initializing the low-level interface routines and _INIT_OTHERLIB() for initializing strtok() and rand().

These initialization functions are called from the initial startup task in both CPUs.

16.7.4 Section Initialization (_INITSCT())

Although the standard library of the compiler provides a standard _INITSCT(), this sample does not use it but implements an original _INITSCT() because using _INITSCT() in the standard library complicates the procedures for transferring the standard library code from ROM to RAM.

The _INITSCT() in this sample has the following additional arguments to initialize a desired section with a desired timing in comparison with the _INITSCT() in the standard library.

- Standard library: void _INITSCT(void);
- This sample: void _INITSCT(

```
    const ST_DTBL *dtbl_top,   // Start address of the ST_DTBL array
    UINT32 dtbl_sz,            // Size of the ST_DTBL array (bytes)
    const ST_BTBL *btbl_top,   // Start address of the ST_BTBL array
    UINT32 btbl_sz);           // Size of the ST_BTBL array (bytes)
```

Each structure has the following elements.

```
typedef struct {  // Initialization information on transfer sections
    UINT8   *SecD_Start;  // Start address of source section
    UINT32  SecD_Size;    // Size of source section
    UINT8   *SecR_Start;  // Start address of destination section
} ST_DTBL;

typedef struct {  // Initialization information on section to be cleared to 0
    UINT8   *SecB_Start;  // Start address of section
    UINT32  SecB_Size;    // Size of section
} ST_BTBL;
```

To make these argument settings easier, the following macros are provided. They are defined in initsct.h.

- MACRO_ENTRY_DTBL(dname, rname)
  Creates ST_DTBL to transfer the "dname" section to the "rname" section.
- MACRO_ENTRY_BTBL(bname)
  Creates ST_BTBL to clear the "bname" section to 0.

16.7.5 Standard Library Configuration (lowsrc_config.h)

Specify the necessary define statements with reference to the following.

```
/*****************************************************************************
 * Defines
*****************************************************************************/
/* size of area managed by sbrk */
#define HEAPSIZE 0x400UL

/* Mutex ceiling priority for wait_sem (only for reentrant library) */
#ifdef _REENTRANT
#define PRI_SBRK    1   /* for sbrk() */
#define PRI_S1PTR   1   /* for _s1ptr(strtok()) */
#define PRI_IOB     1   /* for iob */
#endif

/* Mutex timeout for wait_sem (only for reentrant library) */
#ifdef _REENTRANT
#define SEM_TMOUT   30000L  /* 30000 msec */
#endif

/* Number of tasks for errno (only for reentrant library) */
#ifdef _REENTRANT
#define NUM_TASK    _MAX_TSK
#endif
```

Figure 16.22 lowsrc_config.h

Table 16.12 Setting Items in Standard Library Configuration File

Item	Description
HEAPSIZE	Size of the heap area managed by sbrk()
PRI_SBRK*	Ceiling priority of the mutex used for sbrk() exclusive control
PRI_S1PTR*	Ceiling priority of the mutex used for _s1ptr exclusive control
PRI_IOB*	Ceiling priority of the mutex used for iob exclusive control
SEM_TMOUT*	Timeout for mutex lock
NUM_TASK*	Maximum local task ID

Note: * Required when using a reentrant library.

16.7.6 Source Codes

(1) lowsrc.c (low-level interface routine, _INIT_LOWLEVEL())

```
/*****************************************************************************
 * Include
 *****************************************************************************/
#include <stddef.h>

#include "kernel.h"
#include "types.h"

#include "lowsrc.h"
#include "lowsrc_config.h"

/*****************************************************************************
 * Heap area
 *****************************************************************************/
#pragma section C_heap
static  union  {                        /* memory-pool area */
    INT32  dummy ;                      /* Dummy for 4-byte boundary          */
    INT8   heap[ALIGNUP4(HEAPSIZE)];    /* Declaration of the area managed   */
                                        /*                          by sbrk */
}heap_area ;
#pragma section


/*****************************************************************************
 * Prototypes
 *****************************************************************************/
INT8  *sbrk(size_t size);
#ifdef _REENTRANT
INT wait_sem(INT semnum);
INT signal_sem(INT semnum);
INT *errno_addr(void);
#endif

```

Heap area (a unique section name is assigned)

Figure 16.23 lowsrc.c

34	static INT sbrk_init(void);	
35	static INT sem_init(void);	
36	static void errno_init(void);	
37		
38		
39		
40	/***	
41	* Section	
42	***/	
43	#pragma section C_stdlib	Section name definition
44		
45		
46	/***	
47	* Data	
48	***/	
49	/*** for sbrk() ***/	Variable for holding the variable-sized memory pool ID used in sbrk()
50	static ID sbrk_mplid; /* memory-pool ID */	
51		
52	/*** for semaphore ***/	
53	#ifdef _REENTRANT	
54	#define NUM_SEM 3	
55		
56	#define SEM_SBRK 1 /* semnum for sbrk() */	Semaphore number for sbrk(), which is specified in the standard library
57	#define SEM_S1PTR 2 /* semnum for _s1ptr(strtok()) */	Semaphore number for _s1ptr, which is specified in the standard library
58	#define SEM_IOB 3 /* semnum for iob */	Semaphore number for iob, which is specified in the standard library
59		
60	static ID mtx_id[NUM_SEM]; /* mutex ID for each semnum */	Variable for holding the mutex ID used in wait_sem()
61	#endif /* end of _REENTRANT */	
62		

Figure 16.23   lowsrc.c (cont)

```
/*** for errno ***/
#ifdef _REENTRANT
static INT  errno_context[NUM_TASK+1];
#endif  /* end of _REENTRANT */

/*****************************************************************************
/**  Allocate memory
 *    @param size Required memory size
 *    @retval Pointer to allocated memory(Pass), -1(Failure)
 *****************************************************************************/
INT8  *sbrk(size_t size)
{
    ER      ercd;
    INT8    *p;

    if(sbrk_mplid != 0) {
        ercd = pget_mpl(sbrk_mplid, ALIGNUP4(size), (VP *)&p);
        if(ercd != E_OK) {
            p = (INT8 *)(-1);
        }
    }
    else {
        p = (INT8 *)(-1);
    }

    return p;
}


```

errno area for each task (task ID is used as the index). Index = 0 indicates an area for use in non-task contexts.

sbrk() function

Acquires a memory area from the variable-sized memory pool.

Figure 16.23 lowsrc.c (cont)

```
/*****************************************************************************
/**  Initialize sbrk environment
 *    @param None
 *    @retval 1(Pass), 0(Failure)
 *****************************************************************************/
static INT sbrk_init(void)                                  sbrk_init() function
{
    ER_ID   mplid;
    INT     rtn;

    static const T_CMPL cmpl = {                            Variable-sized memory pool
        TA_TFIFO,                                           creation information
        sizeof(heap_area.heap),
        (VP)(heap_area.heap),
#if ((VTCFG_NEWMPL) == _NEW)
        NULL,
        0U,
        0U
#endif
    };

    mplid = acre_mpl(&cmpl);                                Creates a heap area as a
    if(mplid > 0L) {                                        variable-sized memory pool.
        sbrk_mplid = mplid;
        rtn = 1;
    }
    else {
        sbrk_mplid = 0;                                     Sets sbrk_mplid to 0 if an error
        rtn = 0;                                            occurs.
    }

    return rtn;
}
```

Figure 16.23 lowsrc.c (cont)

/**	
/** Get semaphore	
* @param semnum Semaphore number 1(malloc), 2(strtok), 3(iob)	
* @retval 1(Pass), 0(Failure)	
* @Note When calling from non-task context, this function returns error.	
**/	
#ifdef _REENTRANT	
INT wait_sem(INT semnum)	
{	wait_sem() function
INT rtn;	
ID mtxid;	
mtxid = mtx_id[semnum-1];	Acquires the mutex ID for the semaphore number.
rtn = 0;	
if(mtxid != 0) {	
if(tloc_mtx(mtxid, SEM_TMOUT) == E_OK) {	Locks the mutex.
rtn = 1;	
}	
}	
return rtn;	
}	
#endif /* end of _REENTRANT */	
/**	
/** Release semaphore	
* @param semnum Semaphore number 1(malloc), 2(strtok), 3(iob)	
* @retval 1(Pass), 0(Failure)	
* @Note When calling from non-task context, this function returns error.	
**/	

Figure 16.23 lowsrc.c (cont)

```
#ifdef _REENTRANT
INT signal_sem(INT semnum)                                      signal_sem() function
{
    INT rtn;
    ID  mtxid;

    mtxid = mtx_id[semnum-1];                                   Acquires the mutex ID for the
                                                                semaphore number.
    if(mtxid != 0) {
        if(unl_mtx( mtxid) == E_OK) {                           Unlocks the mutex.
            rtn = 1;
        }
    }
    return rtn;
}
#endif  /* end of _REENTRANT */


/*****************************************************************************
/**  Initialize wait_sem/signal_sem environment
 *    @param None
 *    @retval 1(Pass), 0(Failure)
 *****************************************************************************/
#ifdef _REENTRANT
static INT sem_init(void)                                       sem_init() function
{
    ER_ID   mtxid;
    INT     rtn, i;
    static const T_CMTX cmtx[NUM_SEM] = {                       Mutex creation information
        {TA_CEILING, PRI_SBRK},     /* for sbrk() */
        {TA_CEILING, PRI_S1PTR},    /* for _s1ptr(strtok()) */
        {TA_CEILING, PRI_IOB}       /* for iob */
    };

```

Figure 16.23   lowsrc.c (cont)

193	for(i = 0 ; i < NUM_SEM ; i++) {	
194	mtx_id[i] = 0;	Clears mtx_id[] to 0.
195	}	
196		
197	rtn = 1;	
198	for(i = 0 ; i < NUM_SEM ; i++) {	Creates each mutex.
199	mtxid = acre_mtx(&cmtx[i]);	
200	if(mtxid > 0L) {	
201	mtx_id[i] = mtxid;	Sets mtx_id[] to the ID of a mutex if the mutex has been created successively.
202	}	
203	else {	
204	rtn = 0;	
205	break;	
206	}	
207	}	
208		
209	return rtn;	
210	}	
211	#endif /* end of _REENTRANT */	
212		
213		
214		
215	/**	
216	/** Return "errno" address for cuurent context	
217	* @param None	
218	* @retval Pointer to "errno"	
219	**/	
220	#ifdef _REENTRANT	
221	INT *errno_addr(void)	errno_addr() function
222	{	
223	INT *rtn;	
224	ER ercd;	
225	ID id;	

Figure 16.23   lowsrc.c (cont)

```

    if(sns_ctx() == FALSE) {
        /* Case task context */
        get_tid(&id);
        id = GET_LOCALID(id);
    }
    else {
        /* Case non-task context */
        id = 0;
    }
    return (&errno_context[id]);
}
#endif  /* end of _REENTRANT */


/*******************************************************************************
/**  Initialize errno environment
 *    @param None
 *    @retval None
 *******************************************************************************/
#ifdef _REENTRANT
static void errno_init(void)                                          errno_init() function
{
    INT i;

    for( i= 0 ; i < sizeof(errno_context)/sizeof(INT) ; i++) {
        errno_context[i] = 0;
    }
}
#endif  /* end of _REENTRANT */



```

Figure 16.23   lowsrc.c (cont)

```
/******************************************************************************
/**  Initialize low-level environment
 *    @param None
 *    @retval 1(Pass), 0(Failure)
 ******************************************************************************/
INT _INIT_LOWLEVEL(void)
{
    INT rtn;

    /* initialize sbrk */
    rtn = sbrk_init();
    if(rtn == 0) {
        return rtn;
    }

    /* initialize semaphore */
#ifdef _REENTRANT
    rtn = sem_init();
    if(rtn == 0) {
        return rtn;
    }
#endif  /* end of _REENTRANT */

    /* initialize errno */
#ifdef _REENTRANT
    errno_init();
#endif  /* end of _REENTRANT */

    return rtn;
}
```

_INIT_LOWLEVEL() function

Figure 16.23   lowsrc.c (cont)

(2) otherlibc (_INIT_OTHERLIB())

_INIT_OTHERLIB() initializes global variable "_s1ptr" used in strtok() and calls srand(). The source code is not shown in this manual.

(3) initsct.c (_INITSCT())

```
/******************************************************************************
 * Include
 ******************************************************************************/
#include "types.h"

#include "initsct.h"


/******************************************************************************
 * Section
 ******************************************************************************/
#pragma section C_stdlib                                          Specifies the section name.


/******************************************************************************
/**  Initialize sections                                           _INITSCT()
 *    @param dtbl_top information table address of D section
 *    @param dtbl_sz size of section of information table
 *    @param btbl_top information table address of B section
 *    @param btbl_sz size of section of information table
 *    @retval None
 ******************************************************************************/
void _INITSCT(
    const ST_DTBL   *dtbl_top,
    UINT32          dtbl_sz,
    const ST_BTBL   *btbl_top,
    UINT32          btbl_sz
   )
```

Figure 16.24   initsct.c

```
{
    const ST_DTBL   *dtbl;
    const ST_BTBL   *btbl;
    UINT32  tblcnt;
    UINT32  sz;
    UINT8   *rp, *wp;

    /*** Copy D-section to R-section ***/
    dtbl = dtbl_top;
    tblcnt = dtbl_sz/sizeof(ST_DTBL);

    while(tblcnt > 0UL) {
        rp = dtbl->SecD_Start;
        wp = dtbl->SecR_Start;
        sz = dtbl->SecD_Size;

        while(sz > 0UL) {
            *wp = *rp;      /* Copy D --> R */
            rp++;
            wp++;
            sz--;
        }
        dtbl++;
        tblcnt--;
    }

    /*** 0-clear B-section ***/
    btbl = btbl_top;
    tblcnt = btbl_sz/sizeof(ST_BTBL);

    while(tblcnt > 0UL) {
        wp = btbl->SecB_Start;
        sz = btbl->SecB_Size;

```

Figure 16.24 initsct.c (cont)

```
        while(sz > 0UL) {
            *wp = 0U;       /* 0-clear */
            wp++;
            sz--;
        }
        btbl++;
        tblcnt--;
    }
}
```

Figure 16.24 initsct.c (cont)

16.8 Dummy Objects

16.8.1 Dummy Programs

This sample provides several dummy programs as templates for coding and examples of cfg files for users.

Table 16.3 shows the source files of the dummy programs. The files for both CPUs have the same contents. They are registered in the kernel through the cfg file.

Table 16.13 Source Files of Dummy Programs

Directory	File Name	Function
cpuid1\dummy_prog\	dummy_prog.c	Dummy program for CPUID#1 • DummyTask() (task) • DummyCyclicHandler() (cyclic handler) • DummyAlarmHandler() (alarm handler) • DummyExtendedSVC() (extended service call routine) • DummyInitRoutine() (initialization routine) • DummyNormalIntHandler508() (normal interrupt handler) • DummyDirectIntHandler509() (direct interrupt handler)
cpuid2\dummy_prog\	dummy_prog.c	Dummy program for CPUID#2 • DummyTask() (task) • DummyCyclicHandler() (cyclic handler) • DummyAlarmHandler() (alarm handler) • DummyExtendedSVC() (extended service call routine) • DummyInitRoutine() (initialization routine) • DummyNormalIntHandler510() (normal interrupt handler) • DummyDirectIntHandler511() (direct interrupt handler)

(1) Dummy task (DummyTask()) and dummy extended service call (DummyExtendedSVC())

The dummy task calls a dummy extended service call. The dummy extended service call routine returns control without any processing. The function code for the dummy extended service call is 1.

(2) Dummy cyclic handler (DummyCyclicHandler())

This is initiated at regular intervals but returns control without any processing.

(3) Dummy alarm handler (DummyAlarmHandler())

This is initiated only once but returns control without any processing.

(4) Dummy initialization routine (DummyInitRoutine())

This returns control without any processing.

(5) Dummy normal interrupt handlers (DummyNormalIntHandler508() and DummyNormalIntHandler510())

Dummy normal interrupt handlers are defined for vector number 508 for CPUID#1 and vector number 510 for CPUID#2.

These handlers return control without any processing. Interrupts for these handlers never occur.

(6) Dummy direct interrupt handlers (DummyDirectIntHandler509() and DummyDirectIntHandler511())

Dummy direct interrupt handlers are defined for vector number 509 for CPUID#1 and vector number 511 for CPUID#2.

These handlers return control without any processing. Interrupts for these handlers never occur.

16.8.2 Other Dummy Objects

Like dummy programs, several kernel objects are registered as examples of cfg file descriptions. For details, read the sample cfg file contents.

16.9 I/O Register Definitions, Peripheral Clock Definition, and kernel_intspec.h

This sample provides the I/O definition files shown in table 16.14.

Table 16.14 I/O Definition Files

Directory	File Name	Function	
iodefine\	kernel_intspec.h	CPU interrupt hardware specification definitions	
	pclock.h	Peripheral clock frequency definition	
	io_bsc.h	SH7265 on-chip peripheral register definitions	Bus state controller (BSC)
	io_cmt.h		Compare match timer (CMT)
	io_cpg.h		Clock pulse generator (CPG)
	io_intc.h		Interrupt controller (INTC)
	io_multicore.h		Definition related to multicore environment
	io_port.h		Definition related to I/O ports
	io_stb.h		Definition related to low-power modes
	io_sys.h		Definition related to system control
	io_wdt.h		Watchdog timer (WDT)

kernel_intspec.h is an important file for informing the kernel of the interrupt hardware specifications. For details, refer to section 17.3, Creating CPU Interrupt Specification Definition File (kernel_intspec.h).

pclock.h defines the frequency of the peripheral clock. The timer driver uses this definition.

16.10 List of Kernel Objects

In this sample, objects are created according to the cfg file contents or by service calls issued in the sample programs.

16.10.1 Tasks

Table 16.15 Tasks (CPUID#1)

Category	Creation and Initiation Method	ID Name	Priority
Initial startup task	task[] in cfg file	ID1_TASK_INIT	1
SVC server task	Created by vini_rmt for the number of remote_svc.num_server in cfg file *[1]	(None)	1
Remote service call example *[2]	task[] in cfg file	ID1_TASK_SEND	3
Task for issuing RPC call	task[] in cfg file	ID1_TASK_RPCCALLER	5
Dummy task	task[] in cfg file	ID1_TASK_DUMMY	10

Notes: *1. The remote_svc.num_server setting at shipment is 3.

*2. This task file (<SAMPLE_INST>\R0K572650D000BR\cpuid1\remote_svc_sample\remote_send.c) includes kernel_id_cpu2.h generated for CPUID#2 through cfg72mp to refer to the object IDs for CPUID#2.

Table 16.16 Tasks (CPUID#2)

Category	Creation and Initiation Method	ID Name	Priority
Initial startup task	task[] in cfg file	ID2_TASK_INIT	1
SVC server task	Created by vini_rmt for the number of remote_svc.num_server in cfg file *	(None)	1
Remote service call example	task[] in cfg file	ID2_TASK_RECV	3
Server task for RPC example	Created by rpc_start_server()	(None)	5
Dummy task	task[] in cfg file	ID2_TASK_DUMMY	10

Note: * The remote_svc.num_server setting at shipment is 3.

16.10.2 Other Objects

Table 16.17 Other Objects (CPUID#1)

Object Type	Category	Creation Method	ID Name
Semaphore	Dummy	semaphore[] in cfg file	ID1_SEM_DUMMY
Event flag	Dummy	flag[] in cfg file	ID1_FLG_DUMMY
Data queue	Dummy	dataqueue[] in cfg file	ID1_DTQ_DUMMY
Mailbox	Dummy	mailbox[] in cfg file	ID1_MBX_DUMMY
Mutex	For sbrk() exclusive control	acre_mtx call from _INIT_LOWLEVEL()	(None)
	For _s1ptr exclusive control	acre_mtx call from _INIT_LOWLEVEL()	(None)
	For iob exclusive control	acre_mtx call from _INIT_LOWLEVEL()	(None)
Message buffer	Dummy	message_buffer[] in cfg file	ID1_MBF_DUMMY
Fixed-sized memory pool	Dummy	memorypool[] in cfg file	ID1_MPF_DUMMY
Variable-sized memory pool	Dummy	variable_memorypool[ ] in cfg file	ID1_MPL_DUMMY
	For OAL	acre_mpl call from OAL_Init()	—
Cyclic handler	Dummy	cyclic_hand[] in cfg file	ID1_CYC_DUMMY
Alarm handler	Dummy	alarm_hand[] in cfg file	ID1_ALM_DUMMY
Overrun handler	(Not used)	—	—
Normal interrupt handler	Dummy	interrupt_vector[] in cfg file	(Vector number 508)
Direct interrupt handler	Dummy	interrupt_vector[] in cfg file	(Vector number 509)
CPU exception handler	(Not used)	—	—
Extended service call	Dummy	extend_svc[] in cfg file	—
Initialization routine	Dummy	init_routine[] in cfg file	—

Table 16.18 Other Objects (CPUID#2)

Object Type	Category	Creation Method	ID Name
Semaphore	Dummy	semaphore[] in cfg file	ID2_SEM_DUMMY
Event flag	Dummy	flag[] in cfg file	ID2_FLG_DUMMY
Data queue	Dummy	dataqueue[] in cfg file	ID2_DTQ_DUMMY
Mailbox	Remoter service call example	mailbox[] in cfg file	EXID2_MBX_REMOTE (export to the other CPU)
Mutex	For sbrk() exclusive control	acre_mtx call from _INIT_LOWLEVEL()	(None)
	For _s1ptr exclusive control	acre_mtx call from _INIT_LOWLEVEL()	(None)
	For iob exclusive control	acre_mtx call from _INIT_LOWLEVEL()	(None)
Message buffer	Dummy	message_buffer[] in cfg file	ID2_MBF_DUMMY
Fixed-sized memory pool	Remoter service call example	memorypool[] in cfg file	EXID2_MPF_REMOTE (export to the other CPU)
Variable-sized memory pool	Dummy	variable_memorypool[] in cfg file	ID2_MPL_DUMMY
	For OAL	acre_mpl call from OAL_Init()	—
Cyclic handler	Dummy	cyclic_hand[] in cfg file	ID2_CYC_DUMMY
Alarm handler	Dummy	alarm_hand[] in cfg file	ID2_ALM_DUMMY
Overrun handler	(Not used)	—	—
Normal interrupt handler	Dummy	interrupt_vector[] in cfg file	(Vector number 510)
Direct interrupt handler	Dummy	interrupt_vector[] in cfg file	(Vector number 511)
CPU exception handler	(Not used)	—	—
Extended service call	Dummy	extend_svc[] in cfg file	—
Initialization routine	Dummy	init_routine[] in cfg file	—

16.11 cfg Files

16.11.1 CPUID#1 (cpuid1\cfg_out\sample.cfg)

(1) system definition

1	system {	
2	cpuid = 1;	CPUID
3	stack_size = 0x1000;	Interrupt stack size
4	kernel_stack_size = 0x400;	Kernel stack size
5	priority = 255;	Maximum task priority
6	system_IPL = 14;	Kernel interrupt mask level
7	message_pri = 255;	Maximum message priority
8	tic_deno = 1;	Time tick cycle = TIC_NUME/TIC_DENO = 1 ms
9	tic_nume = 1;	
10	tbr = FOR_SVC;	Uses the TBR register only for service calls.
11	parameter_check = YES;	Detects errors in kernel service call parameters.
12	mpfmanage = IN;	Places a management table in the fixed-sized memory pool.
13	newmpl = NEW;	Manages the variable-sized memory pool with the new method.
14	trace = TARGET_TRACE;	Uses the target trace function for service call trace.
15	trace_buffer = 0x10000;	Trace buffer size
16	trace_object = 5;	Number of objects to be acquired in service call trace
17	action = YES;	Uses the object manipulation function
18	vector_type = ROM;	Interrupt vector type
19	regbank = ALL;	Uses the register bank for all interrupt sources that can use it.
20	};	

(2) maxdefine definition

maxdefine {	
max_task = 20;	Maximum local task ID
max_statictask = 0;	Maximum local task ID that uses the static stack
max_sem = 10;	Maximum local semaphore ID
max_flag = 10;	Maximum local event flag ID
max_dtq = 10;	Maximum local data queue ID
max_mbx = 10;	Maximum local mailbox ID
max_mtx = 10;	Maximum local mutex ID
max_mbf = 10;	Maximum local message buffer ID
max_mpf = 10;	Maximum local fixed-sized memory pool ID
max_mpl = 10;	Maximum local variable-sized memory pool ID
max_cyh = 10;	Maximum local cyclic handler ID
max_alh = 10;	Maximum local alarm handler ID
max_fncd = 10;	Maximum function code for extended service calls
max_int = 511;	Maximum vector number
};	

(3) memstk definition

memstk {	
all_memsize = 0x4000;	Size of default task stack area
};	

(4) memdtq definition

memdtq {	
all_memsize = 0x4000;	Size of default data queue area
};	

(5) memmbf definition

1	memmbf {	
2	all_memsize = 0x4000;	Size of default message buffer area
3	};	

(6) memmpf definition

1	memmpf {	
2	all_memsize = 0x4000;	Size of default fixed-sized memory pool area
3	};	

(7) memmplf definition

1	memmpl {	
2	all_memsize = 0x4000;	Size of default variable-sized memory pool area
3	};	

(8) clock definition

1	clock {	
2	timer = TIMER;	Uses the time management function.
3	IPL = 13;	Timer interrupt level
4	number = 118; // CMT0-ch0	Timer interrupt number (CH0 of CMT0 in SH7265)
5	stack_size = 0x800;	Timer stack size
6	};	

(9) remote_svc definition

1	remote_svc {	
2	num_server = 3;	Number of SVC server tasks
3	priority = 1;	Priority of SVC server tasks
4	stack_size = 0x400;	Stack size for SVC server tasks
5	ipi_portid = 1; // Interrupt priority = 14	IPI port ID used for remote service calls (interrupt level = 14)
6	num_wait = 20;	Maximum number of tasks waiting for an available SVC server task
7	};	

(10) task[] definition

1	// InitTask1() ****************	InitTask1() (initial startup task)
2	task[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
3	name = ID1_TASK_INIT;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	entry_address = InitTask1();	Start address of the task
6	stack_size = 0x400; // the stack is allocated from default area.	Stack size (allocates a stack area from the default task stack area)
7	// stack_section = <input section name>;	(Section name assigned to the stack area)
8	// stack_address = <input start address of stack area>;	(Start address of the stack area)
9	priority = 1;	Task priority at initiation
10	initial_start = ON;	Initial state
11	exinf = 0;	Extended information
12	fpu = OFF;	Does not use the FPU.
13	};	
14		
15	// TaskSend() ****************	TaskSend() (remote service call example)
16	task[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
17	name = ID1_TASK_SEND;	ID name
18	// export = <YES or NO>;	Does not export the ID name.
19	entry_address = TaskSend();	Start address of the task
20	stack_size = 0x400;	Stack size (generates a stack area with the specified section name)
21	stack_section = C_TSKSTK; // the section mane is "BC_TSKSTK".	Section name assigned to stack area = BC_TSKSTK
22	// stack_address = <input start address of stack area>;	(Start address of the stack area)
23	priority = 3;	Task priority at initiation
24	initial_start = ON;	Initial state
25	exinf = 0;	Extended information
26	fpu = OFF;	Does not use the FPU.
27	};	
28		

29	// TaskRpcCaller() ****************	TaskRpcCaller() (RPC call task)
30	task[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
31	name = ID1_TASK_RPCCALLER;	ID name
32	// export = <YES or NO>;	Does not export the ID name.
33	entry_address = TaskRpcCaller();	Start address of the task
34	stack_size = 0x400; // the stack is allocated from default area.	Stack size (allocates a stack area from the default task stack area)
35	// stack_section = <input section name>;	(Section name assigned to the stack area)
36	// stack_address = <input start address of stack area>;	(Start address of the stack area)
37	priority = 5;	Task priority at initiation
38	initial_start = ON;	Initial state
39	exinf = 0;	Extended information
40	fpu = OFF;	Does not use the FPU.
41	};	
42		
43	// DummyTask() ****************	DummyTask() (dummy task)
44	task[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
45	name = ID1_TASK_DUMMY;	ID name
46	// export = <YES or NO>;	Does not export the ID name.
47	entry_address = DummyTask();	Start address of the task
48	stack_size = 0x200; // the stack is allocated from default area.	Stack size (allocates a stack area from the default task stack area)
49	// stack_section = <input section name>;	(Section name assigned to the stack area)
50	// stack_address = <input start address of stack area>;	(Start address of the stack area)
51	priority = 10;	Task priority at initiation
52	initial_start = ON;	Initial state
53	exinf = 0;	Extended information
54	fpu = OFF;	Does not use the FPU.
55	};	

(11) semaphore[] definition

1	// Dummy semaphore ***********	Dummy semaphore
2	semaphore[1] { // the ID is 1.	Sets the local ID number to 1.
3	name = ID1_SEM_DUMMY;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	wait_queue = TA_TFIFO;	Wait queue attribute
6	max_count = 1;	Maximum value of the semaphore counter
7	initial_count = 1;	Initial value of the semaphore counter
8	};	

(12) eventflag[] definition

1	// Dummy eventflag ***********	Dummy event flag
2	flag[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
3	name = ID1_FLG_DUMMY;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	wait_queue = TA_TFIFO;	Wait queue attribute
6	initial_pattern = 0;	Initial bit pattern
7	wait_multi = TA_WSGL;	Does not allow multiple-wait.
8	clear_attribute = YES;	Clear attribute
9	};	

(13) dataqueue[] definition

1	// Dummy dataqueue ***********	Dummy data queue
2	dataqueue[2] { // the ID is 2.	Sets the local ID number to 2.
3	name = ID1_DTQ_DUMMY;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	buffer_size = 256;	Maximum data count
6	section = C_DTQ; // the section mane is "BC_DTQ".	Section name assigned to the data queue area
7	// address = <input start address of dataqueue area>;	(Start address of the data queue area)
8	wait_queue = TA_TFIFO;	Wait queue attribute
9	};	

(14) mailbox[] definition

1	// Dummy mailbox ***********	Dummy mailbox
2	mailbox[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
3	name = ID1_MBX_DUMMY;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	wait_queue = TA_TFIFO;	Wait queue attribute
6	message_queue = TA_MFIFO;	Order of messages in the queue
7	max_pri = 255;	Maximum message priority (this definition has no meaning when TA_MFIFO is selected)
8	};	

(15) mutex[] definition

1	// Dummy mutex ***********	Dummy mutex
2	mutex[1] { // the ID is 1.	Sets the local ID number to 1.
3	name = ID1_MTX_DUMMY;	ID name
4	protocol = TA_CEILING;	Priority ceiling protocol
5	ceil_pri = 1;	Ceiling priority
6	};	

(16) message_buffer[] definition

1	// Dummy message buffer ***********	Dummy message buffer
2	message_buffer[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
3	name = ID1_MBF_DUMMY;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	buffer_size = 0x400;	Buffer size
6	section = C_MBF; // the section mane is "BC_MBF".	Section name assigned to the buffer area
7	// address = <input start address of buffer area>;	(Start address of the buffer area)
8	max_msgsz = 0x100;	Maximum message size
9	wait_queue = TA_TFIFO;	Wait queue attribute
10	};	

(17) memorypool[] definition

1	// Dummy fixed-size memory pool ***********	Dummy fixed-sized memory pool
2	memorypool[] { // the ID is assigned by configurator.	Sets the local ID number to 2.
3	name = ID1_MPF_DUMMY;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	section = C_MPF; // the section name is "BC_MPF".	Section name assigned to the pool area
6	// address = <input start address of pool area>;	(Start address of the pool area)
7	num_block = 32;	Block count
8	siz_block = 16;	Block size
9	wait_queue = TA_TFIFO;	Wait queue attribute
10	};	

(18) variable_memorypool[] definition

1	// Dummy variable size memory pool ***********	Dummy variable-sized memory pool
2	variable_memorypool[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
3	name = ID1_MPL_DUMMY;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	heap_size = 0x400;	Pool size
6	mpl_section = C_MPL;	Section name assigned to the pool area
7	// mpl_address = <input start address of pool area>;	(Start address of the pool area)
8	unfragment = OFF;	Uses the fragmentation reduction function.
9	wait_queue = TA_TFIFO;	Wait queue attribute
10	};	

(19) cyclic_hand[] definition

1	// Dummy cyclic handler ***********	Dummy cyclic handler
2	cyclic_hand[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
3	name = ID1_CYC_DUMMY;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	interval_counter = 100;	Initiation cycle
6	start = ON;	Starts the handler.
7	phsatr = OFF;	Does not preserve the initiation phase.
8	phs_counter = 30;	Initiation phase
9	exinf = 0;	Extended information
10	entry_address = DummyCyclicHandler();	Start address of the handler
11	};	

(20) alarm_hand[] definition

1	// Dummy alarm handler ***********	Dummy alarm handler
2	alarm_hand[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
3	name = ID1_ALM_DUMMY;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	exinf = 0;	Extended information
6	entry_address = DummyAlarmHandler();	Start address of the handler
7	};	

(21) overrun_hand[] definition

1	// Dummy overrun handler ***********	Dummy overrun handler
2	overrun_hand {	
3	entry_address = DummyOverrunHandler();	Start address of the handler
4	};	

(22) extend_svc[] definition

1	// Dummy extended SVC ***********	Dummy extended service call
2	extend_svc[1] { // the function code is 1.	Sets the function code to 1.
3	entry_address = DummyExtendSVCRoutine();	Start address of the extended service call routine
4	};	

(23) interrupt_vector[] definition

1	// Direcr interrupt handler for IPI port ID#0 ***********	Direct interrupt handler for IPI port ID#0 (ipi.c)
2	interrupt_vector[21] {	Sets the vector number to 21.
3	direct = YES;	Specifies the direct attribute.
4	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
5	entry_address = IPI_Port0Handler();	Start address of the handler
6	};	
7		
8	// Direcr interrupt handler for IPI port ID#1 ***********	Direct interrupt handler for IPI port ID#1 (ipi.c)
9	interrupt_vector[22] {	Sets the vector number to 22.
10	direct = YES;	Specifies the direct attribute.
11	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
12	entry_address = IPI_Port1Handler();	Start address of the handler
13	};	
14		

15	// Direcr interrupt handler for IPI port ID#2 ***********	Direct interrupt handler for IPI port ID#2 (ipi.c)
16	interrupt_vector[23] {	Sets the vector number to 23.
17	direct = YES;	Specifies the direct attribute.
18	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
19	entry_address = IPI_Port2Handler();	Start address of the handler
20	};	
21		
22	// Direcr interrupt handler for IPI port ID#3 ***********	Direct interrupt handler for IPI port ID#3 (ipi.c)
23	interrupt_vector[24] {	Sets the vector number to 24.
24	direct = YES;	Specifies the direct attribute.
25	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
26	entry_address = IPI_Port3Handler();	Start address of the handler
27	};	
28		
29	// Direcr interrupt handler for IPI port ID#4 ***********	Direct interrupt handler for IPI port ID#4 (ipi.c)
30	interrupt_vector[25] {	Sets the vector number to 25.
31	direct = YES;	Specifies the direct attribute.
32	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
33	entry_address = IPI_Port4Handler();	Start address of the handler
34	};	
35		
36	// Direcr interrupt handler for IPI port ID#5 ***********	Direct interrupt handler for IPI port ID#5 (ipi.c)
37	interrupt_vector[26] {	Sets the vector number to 26.
38	direct = YES;	Specifies the direct attribute.
39	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
40	entry_address = IPI_Port5Handler();	Start address of the handler
41	};	
42		
43	// Direcr interrupt handler for IPI port ID#6 ***********	Direct interrupt handler for IPI port ID#6 (ipi.c)

44	interrupt_vector[27] {	Sets the vector number to 27.
45	direct = YES;	Specifies the direct attribute.
46	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
47	entry_address = IPI_Port6Handler();	Start address of the handler
48	};	
49		
50	// Direcr interrupt handler for IPI port ID#7 ***********	Direct interrupt handler for IPI port ID#7 (ipi.c)
51	interrupt_vector[28] {	Sets the vector number to 28.
52	direct = YES;	Specifies the direct attribute.
53	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
54	entry_address = IPI_Port7Handler();	Start address of the handler
55	};	
56		
57	// Dummy normal interrupt handler ***********	Dummy normal interrupt handler
58	interrupt_vector[508] { // Normal interrupt handler	Sets the vector number to 508.
59	direct = NO;	Does not specify the direct attribute.
60	regbank = YES;	Uses the register bank (this setting has no meaning because system.regbank is set to ALL).
61	entry_address = DummyNormalIntHandler508();	Start address of the handler
62	};	
63		
64	// Dummy direct interrupt handler ***********	Dummy direct interrupt handler
65	interrupt_vector[509] { // Direct interrupt handler	Sets the vector number to 509.
66	direct = YES;	Specifies the direct attribute.
67	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
68	entry_address = DummyDirectIntHandler509();	Start address of the handler
69	};	

(24) init_routine[] definition

// Dummy initialization routine ***********	Dummy initialization routine
init_routine[] {	
exinf = 0;	Extended information
entry_address = DummyInitRoutine();	Start address of the initialization routine
};	

(25) service_call definition

Only the following service calls are defined as NO.

- vscr_tsk and ivscr_tsk (creating a task using the static stack)
- All service calls for task exception handling
- def_inh and idef_inh (defining an interrupt handler)
- def_exc and idef_exc (defining a CPU exception handler)
- vdef_trp and ivdef_trp (defining a TRAPA exception handler)

16.11.2 CPUID#2 (cpuid2\cfg_out\sample.cfg)

(1) system definition

1	system {	
2	cpuid = 2;	CPUID
3	stack_size = 0x1000;	Interrupt stack size
4	kernel_stack_size = 0x400;	Kernel stack size
5	priority = 255;	Maximum task priority
6	system_IPL = 14;	Kernel interrupt mask level
7	message_pri = 255;	Maximum message priority
8	tic_deno = 1;	Time tick cycle = TIC_NUME/TIC_DENO = 1 ms
9	tic_nume = 1;	
10	tbr = FOR_SVC;	Uses the TBR register only for service calls.
11	parameter_check = YES;	Detects errors in kernel service call parameters.
12	mpfmanage = IN;	Places a management table in the fixed-sized memory pool.
13	newmpl = NEW;	Manages the variable-sized memory pool with the new method.
14	trace = TARGET_TRACE;	Uses the target trace function for service call trace.
15	trace_buffer = 0x10000;	Trace buffer size
16	trace_object = 5;	Number of objects to be acquired in service call trace
17	action = YES;	Uses the object manipulation function
18	vector_type = ROM;	Interrupt vector type
19	regbank = ALL;	Uses the register bank for all interrupt sources that can use it.
20	};	

(2) maxdefine definition

1	maxdefine {	
2	max_task = 20;	Maximum local task ID
3	max_statictask = 0;	Maximum local task ID that uses the static stack
4	max_sem = 10;	Maximum local semaphore ID
5	max_flag = 10;	Maximum local event flag ID
6	max_dtq = 10;	Maximum local data queue ID
7	max_mbx = 10;	Maximum local mailbox ID
8	max_mtx = 10;	Maximum local mutex ID
9	max_mbf = 10;	Maximum local message buffer ID
10	max_mpf = 10;	Maximum local fixed-sized memory pool ID
11	max_mpl = 10;	Maximum local variable-sized memory pool ID
12	max_cyh = 10;	Maximum local cyclic handler ID
13	max_alh = 10;	Maximum local alarm handler ID
14	max_fncd = 10;	Maximum function code for extended service calls
15	max_int = 511;	Maximum vector number
16	};	

(3) memstk definition

1	memstk {	
2	all_memsize = 0x4000;	Size of default task stack area
3	};	

(4) memdtq definition

1	memdtq {	
2	all_memsize = 0x4000;	Size of default data queue area
3	};	

(5) memmbf definition

1	memmbf {	
2	all_memsize = 0x4000;	Size of default message buffer area
3	};	

(6) memmpf definition

1	memmpf {	
2	all_memsize = 0x4000;	Size of default fixed-sized memory pool area
3	};	

(7) memmplf definition

1	memmpl {	
2	all_memsize = 0x4000;	Size of default variable-sized memory pool area
3	};	

(8) clock definition

1	clock {	
2	timer = TIMER;	Uses the time management function.
3	IPL = 13;	Timer interrupt level
4	number = 119; // CMT0-ch1	Timer interrupt number (CH1 of CMT0 in SH7265)
5	stack_size = 0x800;	Timer stack size
6	};	

(9) remote_svc definition

1	remote_svc {	
2	num_server = 3;	Number of SVC server tasks
3	priority = 1;	Priority of SVC server tasks
4	stack_size = 0x400;	Stack size for SVC server tasks
5	ipi_portid = 1; // Interrupt priority = 14	IPI port ID used for remote service calls (interrupt level = 14)
6	num_wait = 20;	Maximum number of tasks waiting for an available SVC server task
7	};	

(10) task[] definition

1	// InitTask2() ******************	InitTask2() (initial startup task)
2	task[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
3	name = ID2_TASK_INIT;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	entry_address = InitTask2();	Start address of the task
6	stack_size = 0x400; // the stack is allocated from default area.	Stack size (allocates a stack area to the default task stack area)
7	// stack_section = <input section name>;	(Section name assigned to the stack area)
8	// stack_address = <input start address of stack area>;	(Start address of the stack area)
9	priority = 1;	Task priority at initiation
10	initial_start = ON;	Initial state
11	exinf = 0;	Extended information
12	fpu = OFF;	Does not use the FPU.
13	};	
14		

15	// TaskRecv() ****************	TaskRecv() (remote service call example)
16	task[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
17	name = ID2_TASK_RECV;	ID name
18	// export = <YES or NO>;	Does not export the ID name.
19	entry_address = TaskRecv();	Start address of the task
20	stack_size = 0x400;	Stack size (creates a stack area with the specified section name)
21	stack_section = C_TSKSTK; // the section mane is "BC_TSKSTK".	Section name assigned to stack area = BC_TSKSTK
22	// stack_address = <input start address of stack area>;	(Start address of the stack area)
23	priority = 3;	Task priority at initiation
24	initial_start = ON;	Initial state
25	exinf = 0;	Extended information
26	fpu = OFF;	Does not use the FPU.
27	};	
28		
29	// DummyTask() ****************	DummyTask() (dummy task)
30	task[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
31	name = ID2_TASK_DUMMY;	ID name
32	// export = <YES or NO>;	Does not export the ID name.
33	entry_address = DummyTask();	Start address of the task
34	stack_size = 0x200; // the stack is allocated from default area.	Stack size (allocates a stack area to the default task stack area)
35	// stack_section = <input section name>;	(Section name assigned to the stack area)
36	// stack_address = <input start address of stack area>;	(Start address of the stack area)
37	priority = 10;	Task priority at initiation
38	initial_start = ON;	Initial state
39	exinf = 0;	Extended information
40	fpu = OFF;	Does not use the FPU.
41	};	

(11) semaphore[] definition

1	// Dummy semaphore ***********	Dummy semaphore
2	semaphore[1] { // the ID is 1.	Sets the local ID number to 1.
3	name = ID2_SEM_DUMMY;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	wait_queue = TA_TFIFO;	Wait queue attribute
6	max_count = 1;	Maximum value of the semaphore counter
7	initial_count = 1;	Initial value of the semaphore counter
8	};	

(12) eventflag[] definition

1	// Dummy eventflag ***********	Dummy event flag
2	flag[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
3	name = ID2_FLG_DUMMY;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	wait_queue = TA_TFIFO;	Wait queue attribute
6	initial_pattern = 0;	Initial bit pattern
7	wait_multi = TA_WSGL;	Does not allow multiple-wait.
8	clear_attribute = YES;	Clear attribute
9	};	

(13) dataqueue[] definition

1	// Dummy dataqueue ***********	Dummy data queue
2	dataqueue[2] { // the ID is 2.	Sets the local ID number to 2.
3	name = ID2_DTQ_DUMMY;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	buffer_size = 256;	Maximum data count
6	section = C_DTQ; // the section mane is "BC_DTQ".	Section name assigned to the data queue area
7	// address = <input start address of dataqueue area>;	(Start address of the data queue area)
8	wait_queue = TA_TFIFO;	Wait queue attribute
9	};	

(14) mailbox[] definition

1	// Mailbox for remote-SVC sample ***********	Mailbox used in remote service call example
2	mailbox[1] { // the ID is 1.	Sets the local ID number to 1.
3	name = EXID2_MBX_COMM;	ID name
4	export = YES;	Exports the ID name.
5	wait_queue = TA_TFIFO;	Wait queue attribute
6	message_queue = TA_MFIFO;	Order of messages in the queue
7	max_pri = 255;	Maximum message priority (this definition has no meaning when TA_MFIFO is selected)
8	};	
9		
10	// Dummy mailbox ***********	Dummy mailbox
11	mailbox[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
12	name = ID2_MBX_DUMMY;	ID name
13	// export = <YES or NO>;	Does not export the ID name.
14	wait_queue = TA_TFIFO;	Wait queue attribute
15	message_queue = TA_MFIFO;	Order of messages in the queue
16	max_pri = 255;	Maximum message priority (this definition has no meaning when TA_MFIFO is selected)
17	};	

(15) mutex[] definition

1	// Dummy mutex ***********	Dummy mutex
2	mutex[1] { // the ID is 1.	Sets the local ID number to 1.
3	name = ID2_MTX_DUMMY;	ID name
4	protocol = TA_CEILING;	Priority ceiling protocol
5	ceil_pri = 1;	Ceiling priority
6	};	

(16) message_buffer[] definition

1	// Dummy message buffer ***********	Dummy message buffer
2	message_buffer[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
3	name = ID2_MBF_DUMMY;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	buffer_size = 0x400;	Buffer size
6	section = C_MBF; // the section mane is "BC_MBF".	Section name assigned to the buffer area
7	// address = <input start address of buffer area>;	(Start address of the buffer area)
8	max_msgsz = 0x100;	Maximum message size
9	wait_queue = TA_TFIFO;	Wait queue attribute
10	};	

(17) memorypool[] definition

1	// Fixed-size memory pool for remote-SVC sample ***********	Fixed-sized memory pool used in remoter service call example
2	memorypool[1] { // the ID is 1.	Sets the local ID number to 1.
3	name = EXID2_MPF_NONCACHED;	ID name
4	export = YES;	Exports the ID name.
5	section = D_MPF; // the section name is "BD_MPF".	Section name assigned to the pool area
6	// address = <input start address of pool area>;	(Start address of the pool area)
7	num_block = 16;	Block count
8	siz_block = 8; // sizeof(USER_MSG)	Block size
9	wait_queue = TA_TFIFO;	Wait queue attribute
10	};	
11		
12	// Dummy fixed-size memory pool ***********	Dummy fixed-sized memory pool
13	memorypool[] { // the ID is assigned by configurator.	Sets the local ID number to 2.
14	name = ID2_MPF_DUMMY;	ID name
15	// export = <YES or NO>;	Does not export the ID name.
16	section = C_MPF; // the section name is "BC_MPF".	Section name assigned to the pool area
17	// address = <input start address of pool area>;	(Start address of the pool area)
18	num_block = 32;	Block count
19	siz_block = 16;	Block size
20	wait_queue = TA_TFIFO;	Wait queue attribute
21	};	

(18) variable_memorypool[] definition

1	// Dummy variable size memory pool ***********	Dummy variable-sized memory pool
2	variable_memorypool[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
3	name = ID2_MPL_DUMMY;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	heap_size = 0x400;	Pool size
6	mpl_section = C_MPL;	Section name assigned to the pool area
7	// mpl_address = <input start address of pool area>;	(Start address of the pool area)
8	unfragment = OFF;	Uses the fragmentation reduction function.
9	wait_queue = TA_TFIFO;	Wait queue attribute
10	};	

(19) cyclic_hand[] definition

1	// Dummy cyclic handler ***********	Dummy cyclic handler
2	cyclic_hand[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
3	name = ID2_CYC_DUMMY;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	interval_counter = 100;	Initiation cycle
6	start = ON;	Starts the handler.
7	phsatr = OFF;	Does not preserve the initiation phase.
8	phs_counter = 30;	Initiation phase
9	exinf = 0;	Extended information
10	entry_address = DummyCyclicHandler();	Start address of the handler
11	};	

(20) alarm_hand[] definition

1	// Dummy alarm handler ***********	Dummy alarm handler
2	alarm_hand[] { // the ID is assigned by configurator.	Assigns an ID number automatically.
3	name = ID2_ALM_DUMMY;	ID name
4	// export = <YES or NO>;	Does not export the ID name.
5	exinf = 0;	Extended information
6	entry_address = DummyAlarmHandler();	Start address of the handler
7	};	

(21) overrun_hand[] definition

1	// Dummy overrun handler ***********	Dummy overrun handler
2	overrun_hand {	
3	entry_address = DummyAlarmHandler();	Start address of the handler
4	};	

(22) extend_svc[] definition

1	// Dummy extended SVC ***********	Dummy extended service call
2	extend_svc[1] { // the function code is 1.	Sets the function code to 1.
3	entry_address = DummyExtendSVCRoutine();	Start address of the extended service call routine
4	};	

(23) interrupt_vector[] definition

1	// Direcr interrupt handler for IPI port ID#0 ***********	Direct interrupt handler for IPI port ID#0 (ipi.c)
2	interrupt_vector[21] {	Sets the vector number to 21.
3	direct = YES;	Specifies the direct attribute.
4	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
5	entry_address = IPI_Port0Handler();	Start address of the handler
6	};	
7		
8	// Direcr interrupt handler for IPI port ID#1 ***********	Direct interrupt handler for IPI port ID#1 (ipi.c)
9	interrupt_vector[22] {	Sets the vector number to 22.
10	direct = YES;	Specifies the direct attribute.
11	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
12	entry_address = IPI_Port1Handler();	Start address of the handler
13	};	
14		
15	// Direcr interrupt handler for IPI port ID#2 ***********	Direct interrupt handler for IPI port ID#2 (ipi.c)
16	interrupt_vector[23] {	Sets the vector number to 23.
17	direct = YES;	Specifies the direct attribute.
18	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
19	entry_address = IPI_Port2Handler();	Start address of the handler
20	};	
21		
22	// Direcr interrupt handler for IPI port ID#3 ***********	Direct interrupt handler for IPI port ID#3 (ipi.c)
23	interrupt_vector[24] {	Sets the vector number to 24.
24	direct = YES;	Specifies the direct attribute.
25	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).

26	entry_address = IPI_Port3Handler();	Start address of the handler
27	};	
28		
29	// Direcr interrupt handler for IPI port ID#4 ***********	Direct interrupt handler for IPI port ID#4 (ipi.c)
30	interrupt_vector[25] {	Sets the vector number to 25.
31	direct = YES;	Specifies the direct attribute.
32	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
33	entry_address = IPI_Port4Handler();	Start address of the handler
34	};	
35		
36	// Direcr interrupt handler for IPI port ID#5 ***********	Direct interrupt handler for IPI port ID#5 (ipi.c)
37	interrupt_vector[26] {	Sets the vector number to 26.
38	direct = YES;	Specifies the direct attribute.
39	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
40	entry_address = IPI_Port5Handler();	Start address of the handler
41	};	
42		
43	// Direcr interrupt handler for IPI port ID#6 ***********	Direct interrupt handler for IPI port ID#6 (ipi.c)
44	interrupt_vector[27] {	Sets the vector number to 27.
45	direct = YES;	Specifies the direct attribute.
46	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
47	entry_address = IPI_Port6Handler();	Start address of the handler
48	};	
49		
50	// Direcr interrupt handler for IPI port ID#7 ***********	Direct interrupt handler for IPI port ID#7 (ipi.c)
51	interrupt_vector[28] {	Sets the vector number to 28.
52	direct = YES;	Specifies the direct attribute.
53	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
54	entry_address = IPI_Port7Handler();	Start address of the handler

55	};	
56		
57	// Dummy normal interrupt handler ***********	Dummy normal interrupt handler
58	interrupt_vector[510] { // Normal interrupt handler	Sets the vector number to 510.
59	direct = NO;	Does not specify the direct attribute.
60	regbank = YES;	Uses the register bank (this setting has no meaning because system.regbank is set to ALL).
61	entry_address = DummyNormalIntHandler510();	Start address of the handler
62	};	
63		
64	// Dummy direct interrupt handler ***********	Dummy direct interrupt handler
65	interrupt_vector[511] { // Direct interrupt handler	Sets the vector number to 511.
66	direct = YES;	Specifies the direct attribute.
67	regbank = YES;	Uses the register bank (this setting has no meaning because the direct attribute is selected).
68	entry_address = DummyDirectIntHandler511();	Start address of the handler
69	};	

(24) init_routine[] definition

1	// Dummy initialization routine ***********	Dummy initialization routine
2	init_routine[] {	
3	exinf = 0;	Extended information
4	entry_address = DummyInitRoutine();	Start address of the initialization routine
5	};	

(25) service_call definition

Only the following service calls are defined as NO.

- vscr_tsk and ivscr_tsk (creating a task using the static stack)
- All service calls for task exception handling
- def_inh and idef_inh (defining an interrupt handler)
- def_exc and idef_exc (defining a CPU exception handler)
- vdef_trp and ivdef_trp (defining a TRAPA exception handler)

16.12 IPI Ports

In this sample, both CPUID#1 and CPUID#2 use IPI ports for remote service call and RPC functions.

To use IPI ports, settings should be made in several files such as the cfg file, IPI configuration file, and rpc_init() in the RPC library. This section describes how to make IPI port settings without making wrong settings.

In the IPI configuration file, define whether to allow use of each port ID.

For the port ID to be used by remote service calls, select an available port ID that is not higher in the interrupt level than the kernel interrupt mask level. Define it through remote_svc.ipi_portid in the cfg file. According to this setting, vini_rmt executes IPI_create() with this port ID.

For the port ID to be used for RPC, select an available port ID that is not higher in the interrupt level than the kernel interrupt mask level. Specify it in ulIPIPortID of the rpc_config structure to be passed to rpc_init(). According to this setting, rpc_init() executes IPI_create() with this port ID.

The IPI configuration files are cpuid1\ipi\ipi_config.h for CPUID#1 and cpuid2\ipi\ipi_config.h for CPUID#2.

rpc_init() is called by the initial startup task in cpuid1\init_task\init_task.c in CPUID#1 and by that in cpuid2\init_task\init_task.c in CPUID#2.

The kernel interrupt mask level (system.system_IPL) is set to 14 for both CPUs.

Table 16.19 shows the IPI port usage in this sample.

Table 16.19 IPI Ports

			CPUID#1		CPUID#2	
Port ID	Vector No.	Inter-Processor Interrupt Level	IPI Configu-ration	IPI_create() State	IPI Configu-ration	IPI_create() State
0	21	15	Available	(Not used)	Available	(Not used)
1	22	14	Available	For remote service calls (remote_svc.ipi_portid)	Available	For remote service calls (remote_svc.ipi_portid)
2	23	13	Available	For RPC (rpc_config.ulIPIPortID)	Available	For RPC (rpc_config.ulIPIPortID)
3	24	12	Available	(Not used)	Available	(Not used)
4	25	11	Available	(Not used)	Available	(Not used)
5	26	10	Available	(Not used)	Available	(Not used)
6	27	9	Available	(Not used)	Available	(Not used)
7	28	8	Available	(Not used)	Available	(Not used)

16.13 Porting to Other Hardware

When porting these sample programs to other hardware, modify the following files according to the target hardware specifications.

(1) Files in the iodefine\ directory (I/O register definitions, peripheral clock definition, and kernel_intspec.h)

(2) Files in the cpuid1\reset\ directory and cpuid2\reset\ directory (settings related to reset)

(3) Files in the cpuid1\os_timer\ directory and cpuid2\os_timer\ directory (timer driver)

Section 17 Build

This section mainly explains the build method using the sample High-performance Embedded Workshop workspaces under the <SAMPLE_INST>R0K572650D000BR directory.

A certain level of knowledge on the following tools is necessary to understand this section.

- High-performance Embedded Workshop
- Toolchain
- cfg72mp

The sample High-performance Embedded Workshop workspaces are as follows:

- For CPUID#1: <SAMPLE_INST>\R0K572650D000BR\cpuid1\cpuid1.hws
- For CPUID#2: <SAMPLE_INST>\R0K572650D000BR\cpuid2\cpuid2.hws

17.1 Setting Custom Placeholder $(RTOS_INST)

Custom placeholder "$(RTOS_INST)" is used in the provided workspaces. It stands for the system directory of the HI7200/MP.

In a case where the system directory needs to be changed, such as when the HI7200/MP is upgraded or when the workspaces are moved to another machine, the user workspace $(RTOS_INST) must be changed.

For the method of adding or changing the custom placeholder, refer to the High-performance Embedded Workshop manual or online help.

17.2 Registering cfg72mp to Workspaces as Custom Build Phase

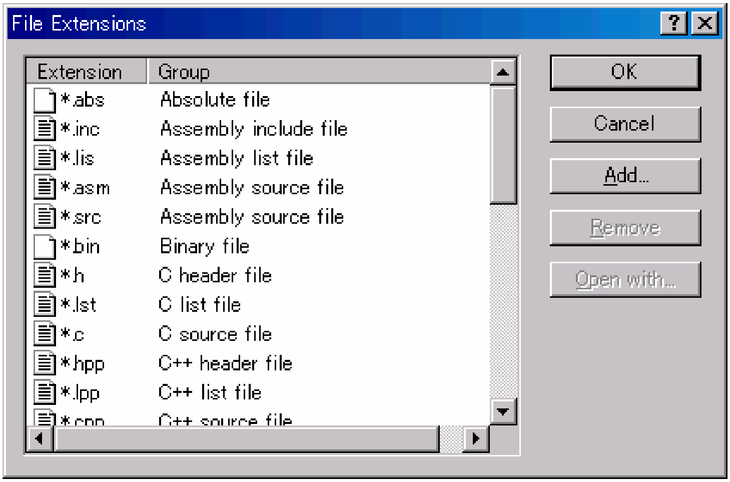
Register cfg72mp to the workspaces as the custom build phase. This enables cfg72mp to be executed by the build operations of the High-performance Embedded Workshop. The method for registering the custom build phase is explained here.

Note that the procedure described below is unnecessary for the provided High-performance Embedded Workshop workspaces because cfg72mp is already registered as the custom build phase.

17.2.1 Registering the File Extension

In order to use the custom build phase, ".cfg" needs to be registered as the file extension handled by the custom build phase.

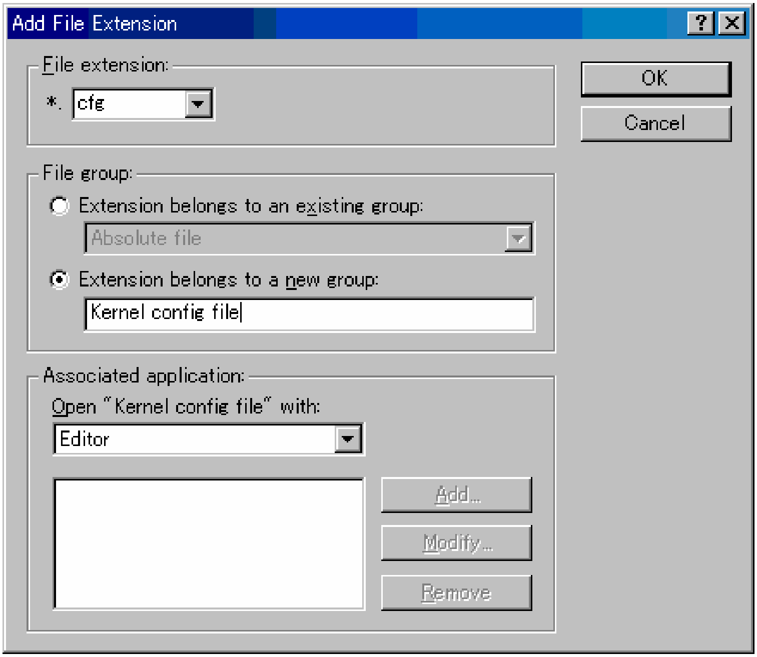
Selecting [Project -> File Extensions] from the High-performance Embedded Workshop menu bar can open the dialog box in figure 17.1.



Extension	Group
*.abs	Absolute file
*.inc	Assembly include file
*.lis	Assembly list file
*.asm	Assembly source file
*.src	Assembly source file
*.bin	Binary file
*.h	C header file
*.lst	C list file
*.c	C source file
*.hpp	C++ header file
*.lpp	C++ list file
*.cpp	C++ source file

Figure 17.1 [File Extensions] Dialog Box

Click the [Add...] button to open the [Add File Extension] dialog box, and then register ".cfg".



Add File Extension

File extension:

*. cfg

File group:

○ Extension belongs to an existing group:

Absolute file

● Extension belongs to a new group:

Kernel config file

Associated application:

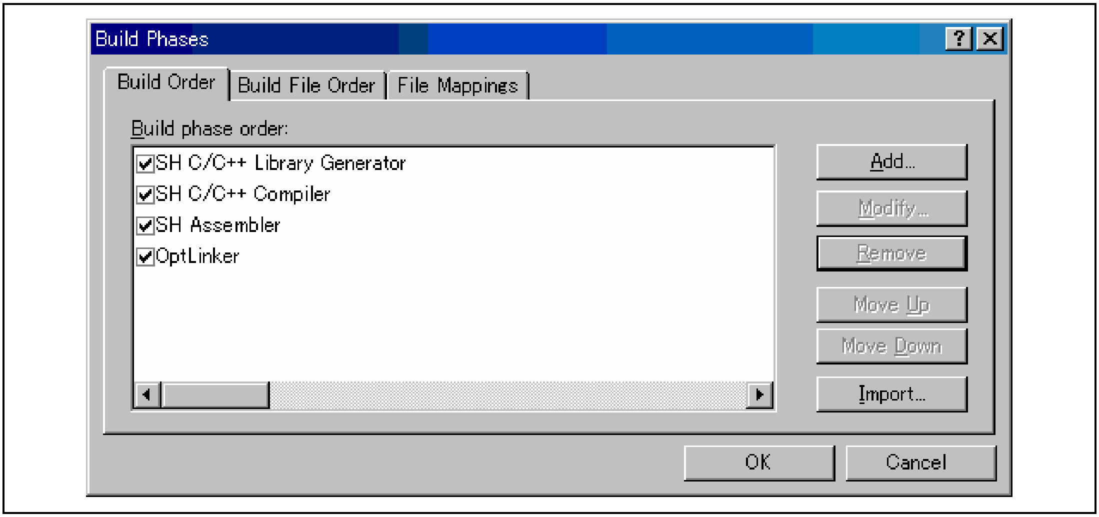
Open "Kernel config file" with:

Editor

Figure 17.2 [Add File Extension] Dialog Box

17.2.2 Creating the cfg72mp Custom Build Phase

(1) Selecting [Build -> Build Phases] from the High-performance Embedded Workshop menu bar can open the dialog box in figure 17.3.



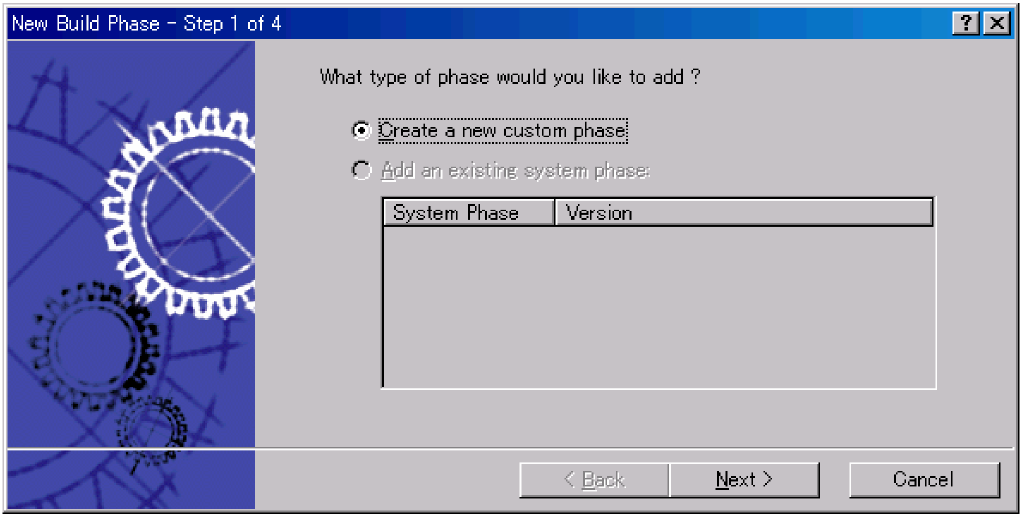
Build phase order:

☑SH C/C++ Library Generator
☑SH C/C++ Compiler
☑SH Assembler
☑OptLinker

Figure 17.3 [Build Phases] Dialog Box

Click the [Add...] button in this dialog box.

(2) The following dialog box is displayed. The cfg72mp custom build phase can be set in the subsequent dialog boxes.



New Build Phase - Step 1 of 4

What type of phase would you like to add ?

● Create a new custom phase

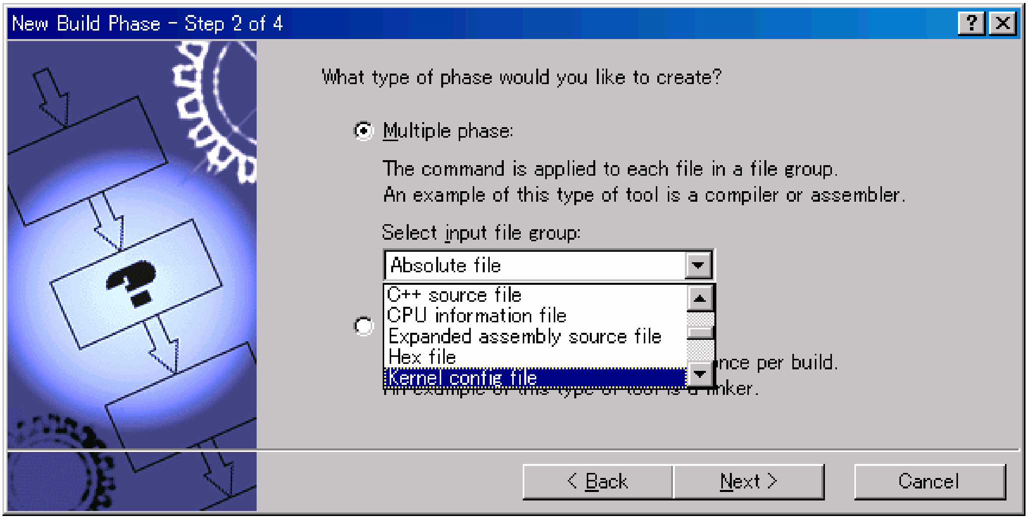
○ Add an existing system phase:

System Phase	Version

Figure 17.4 [New Build Phase — Step 1 of 4] Dialog Box

Click the [Next] button.

(3) Select [Multiple phase]. Then select "Kernel config file" in [Select input file group] and click the [Next] button.



New Build Phase - Step 2 of 4

What type of phase would you like to create?

● Multiple phase:

The command is applied to each file in a file group.
An example of this type of tool is a compiler or assembler.

Select input file group:

Absolute file

C++ source file
CPU information file
Expanded assembly source file
Hex file
Kernel config file

○ ...nce per build.
An example of this type of tool is a linker.

Figure 17.5 [New Build Phase — Step 2 of 4] Dialog Box

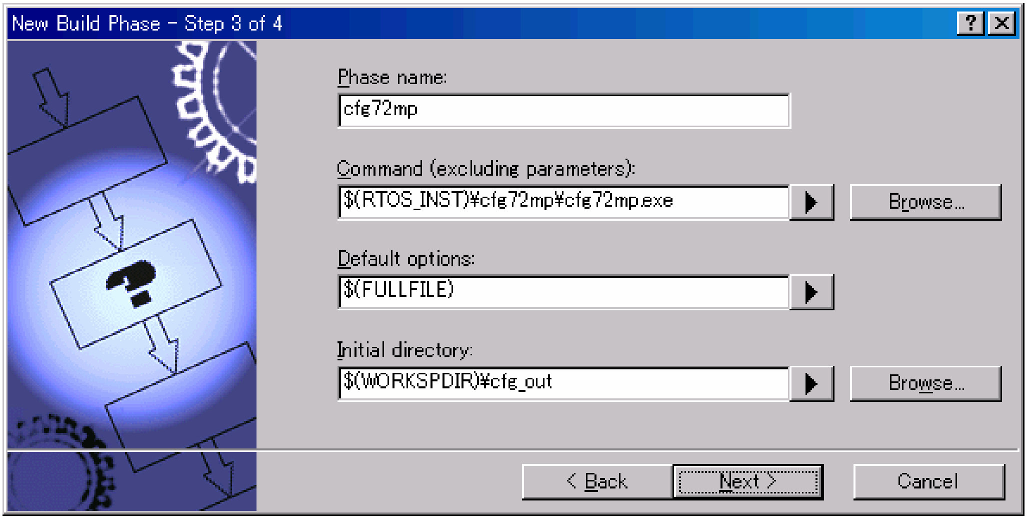
(4) Set the custom build phase information.

Any name can be set in [Phase name]. In this example, "cfg72mp" is set.

Specify "$(RTOS_INST)\cfg72mp\cfg72mp.exe" in [Command (excluding parameters)].

Specify "$(FULLFILE)" in [Default options].

Specify "$(WORKSPDIR)\cfg_out" in [Initial directory].



New Build Phase - Step 3 of 4

Phase name:
cfg72mp

Command (excluding parameters):
$(RTOS_INST)¥cfg72mp¥cfg72mp.exe

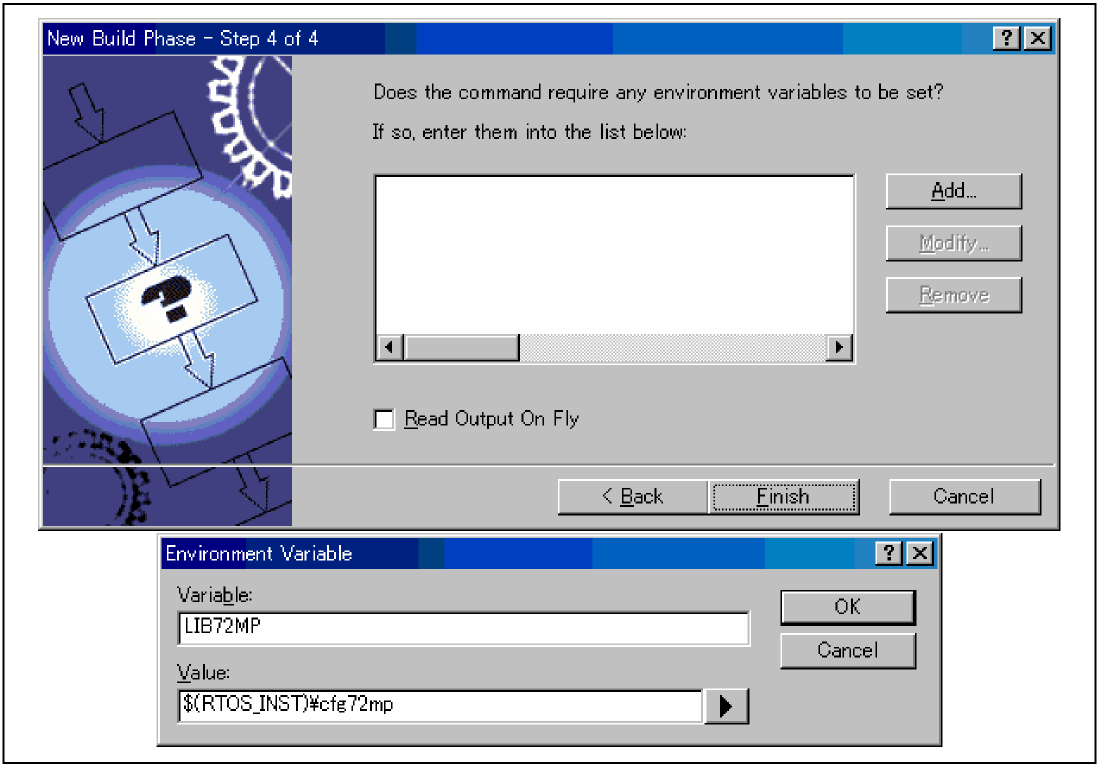
Default options:
$(FULLFILE)

Initial directory:
$(WORKSPDIR)¥cfg_out

Figure 17.6 [New Build Phase — Step 3 of 4] Dialog Box

Click the [Next] button.

(5) Set the environment variables. Click the [Add...] button to open the [Environment Variable] dialog box and make the setting as shown below.



New Build Phase - Step 4 of 4
Does the command require any environment variables to be set?
If so, enter them into the list below:
Add...
Modify...
Remove
□ Read Output On Fly
< Back
Finish
Cancel
Environment Variable
OK
Cancel

Variable: LIB72MP

Value: $(RTOS_INST)¥cfg72mp

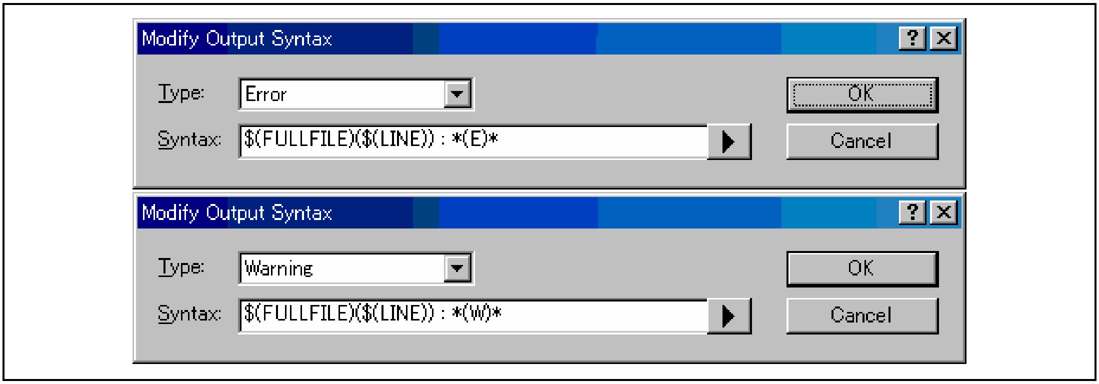
Figure 17.7 [New Build Phase — Step 4 of 4] Dialog Box

Creating the cfg72mp custom build phase is completed at this point. Click the [Finish] button.

(6) Next, set the message syntaxes of cfg72mp. When this setting is made, double-clicking the cfg72mp error or warning message displayed in the High-performance Embedded Workshop's [Build] window makes the display jump to the relevant location in the cfg file.

Select cfg72mp in the [Build Phases] dialog box and click the [Modify...] button. This will open the [Modify cfg72mp] dialog box.

Select the [Output Syntax] tab and set the error and warning syntaxes as shown below.



Modify Output Syntax

Type: Error

Syntax: $(FULLFILE)($(LINE)) : *(E)*

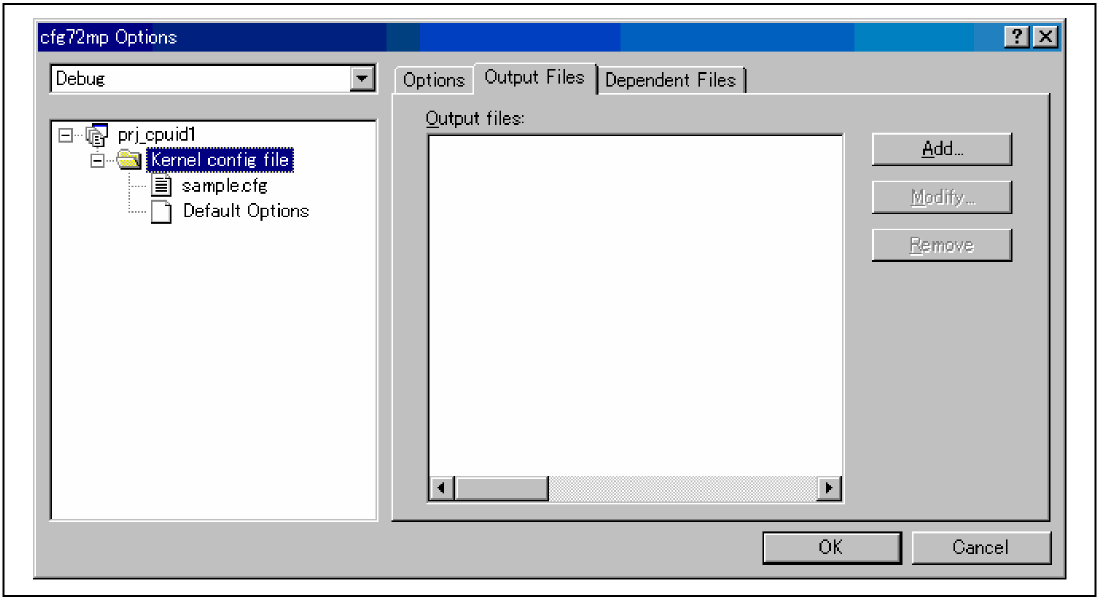
Modify Output Syntax

Type: Warning

Syntax: $(FULLFILE)($(LINE)) : *(W)*

Figure 17.8 Registering cfg72mp Output Syntaxes

(7) After that, make a setting to delete the cfg72mp output files by using [Clean Current Project] and [Clean All Projects] of the High-performance Embedded Workshop.
Select the [Build -> cfg72mp...] menu to open the [cfg72mp Options] dialog box. Then select the [Output Files] tab and click the [Add...] button with the "Kernel config file" folder icon selected to register the information shown in table 17.1.



cfg72mp Options

Debug

- prj_cpuid1
  - Kernel config file
    - sample.cfg
    - Default Options

Output files:

Figure 17.9 [cfg72mp Options] Dialog Box

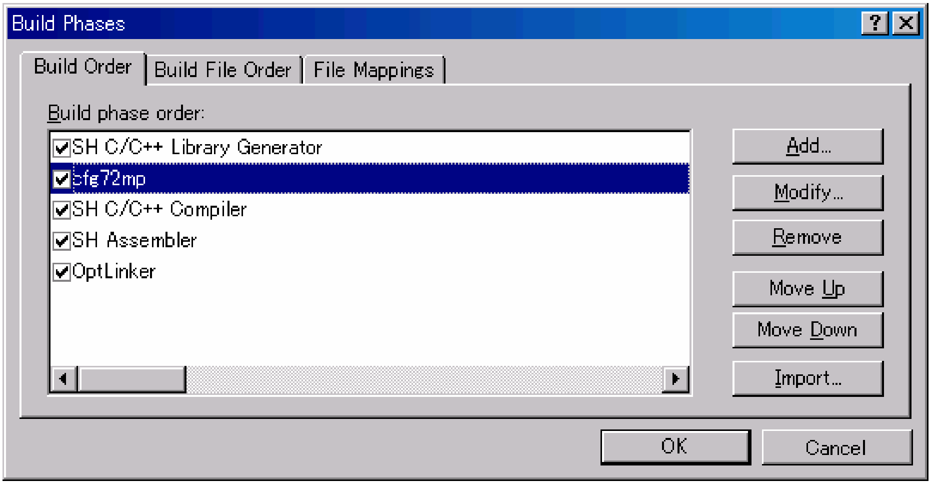
Table 17.1 cfg72mp Output Files

Setting	Remarks
$(FILEDIR)\kernel_cfg.h	
$(FILEDIR)\kernel_cfg_area.h	
$(FILEDIR)\kernel_cfg_extern.h	
$(FILEDIR)\kernel_cfg_inireg.h	
$(FILEDIR)\kernel_cfg_inirtn.h	
$(FILEDIR)\kernel_cfg_ststk.h	
$(FILEDIR)\kernel_def.h	
$(FILEDIR)\kernel_def_area.h	
$(FILEDIR)\kernel_def_extern.h	
$(FILEDIR)\kernel_def_inireg.h	
$(FILEDIR)\kernel_def_inirtn.h	
$(FILEDIR)\kernel_id.h	
$(FILEDIR)\kernel_id_cpu1.h	CPUID#1 only
$(FILEDIR)\kernel_id_cpu2.h	CPUID#2 only
$(FILEDIR)\kernel_id_sys.h	
$(FILEDIR)\kernel_id_sys_cpu1.h	CPUID#1 only
$(FILEDIR)\kernel_id_sys_cpu2.h	CPUID#2 only
$(FILEDIR)\kernel_macro.h	
$(FILEDIR)\mycpuid.h	

17.2.3 Setting Build Phases

(1) Setting Build Order

The created cfg72mp custom build phase is displayed on the lowest line. Use the [Move Up] or [Move Down] button to move the phases so that the order of "cfg72mp" becomes higher than that of "SH C/C++ Compiler" as shown below.



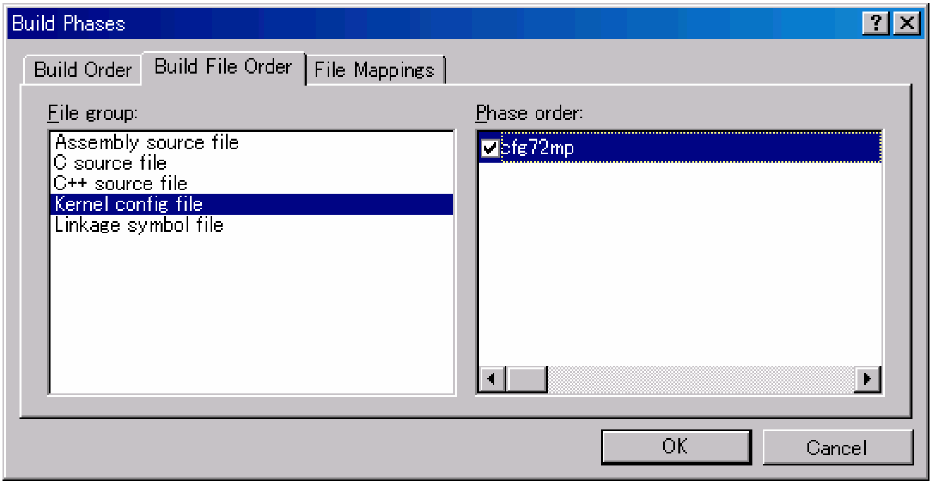
Build phase order:

- ☑ SH C/C++ Library Generator
- ☑ cfg72mp
- ☑ SH C/C++ Compiler
- ☑ SH Assembler
- ☑ OptLinker

Figure 17.10 [Build Phases] Dialog Box (Build Order)

(2) Setting Build File Order

In the [Build File Order] tab, select "Kernel config file" from [File group] and select "cfg72mp" in [Phase order] in the right pane.



File group:
- Assembly source file
- C source file
- C++ source file
- Kernel config file
- Linkage symbol file

Phase order:
- ☑ cfg72mp

Figure 17.11 [Build Phases] Dialog Box (Build File Order)

Setting the build phases is completed at this point.

17.3 Creating CPU Interrupt Specification Definition File (kernel_intspec.h)

kernel_intspec.h is an important file used to inform the kernel of the CPU interrupt hardware specifications, such as vector numbers that cannot use the register banks, e.g. NMI and exceptions. One file is created to be shared by CPUID#1 and CPUID#2. The file is stored in the <SAMPLE_INST>\R0K572650D000BR\iodefine\ directory at shipment.

kernel_intspec.h is included from kernel.h.

An example of kernel_intspec.h is shown in the following.

```
/***************************************************************************
1. Define IBNR register address (INTSPEC_IBNR_ADR)
   Specify 0 when the CPU used does not support register-bank.
/***************************************************************************/
#define INTSPEC_IBNR_ADR1       0xFFFD940E      /**< IBNR register address for CPUID#1 */
#define INTSPEC_IBNR_ADR2       0xFFFD950E      /**< IBNR register address for CPUID#2 */


/***************************************************************************
2. Define the vector number that can not use register-bank(INTSPEC_NOBANK_VECxxx)
   "xxx" is an expression of the vector number by three decimal digits.
   These definitions are ignored when INTSPEC_IBNR_ADR is 0.
   Note, you don't have to define for vector number 0...3.
/***************************************************************************/
#define INTSPEC_NOBANK_VEC004   /**< exception */
#define INTSPEC_NOBANK_VEC005   /**< exception */
#define INTSPEC_NOBANK_VEC006   /**< exception */
#define INTSPEC_NOBANK_VEC007   /**< exception */
#define INTSPEC_NOBANK_VEC008   /**< exception */
#define INTSPEC_NOBANK_VEC009   /**< exception */
#define INTSPEC_NOBANK_VEC010   /**< exception */
#define INTSPEC_NOBANK_VEC011   /**< NMI (cannot use register-bank) */
#define INTSPEC_NOBANK_VEC012   /**< user break interrupt (cannot use register-bank)  */
#define INTSPEC_NOBANK_VEC013   /**< exception */
/* #define INTSPEC_NOBANK_VEC014 *   H-UDI */
#define INTSPEC_NOBANK_VEC015   /**< exception */
#define INTSPEC_NOBANK_VEC016   /**< exception */
#define INTSPEC_NOBANK_VEC017   /**< exception */
#define INTSPEC_NOBANK_VEC018   /**< exception */
#define INTSPEC_NOBANK_VEC019   /**< exception */
#define INTSPEC_NOBANK_VEC020   /**< SCO interrupt (cannot use register-bank) */
/* #define INTSPEC_NOBANK_VEC021* inter-processor interrupt */
/* #define INTSPEC_NOBANK_VEC022* inter-processor interrupt */
/* #define INTSPEC_NOBANK_VEC023* inter-processor interrupt */
/* #define INTSPEC_NOBANK_VEC024* inter-processor interrupt */
/* #define INTSPEC_NOBANK_VEC025* inter-processor interrupt */
/* #define INTSPEC_NOBANK_VEC026* inter-processor interrupt */
/* #define INTSPEC_NOBANK_VEC027* inter-processor interrupt */
/* #define INTSPEC_NOBANK_VEC028* inter-processor interrupt */
#define INTSPEC_NOBANK_VEC029 /**< exception */
#define INTSPEC_NOBANK_VEC030 /**< exception */
#define INTSPEC_NOBANK_VEC031 /**< exception */
#define INTSPEC_NOBANK_VEC032 /**< TRAPA */
#define INTSPEC_NOBANK_VEC033 /**< TRAPA */
#define INTSPEC_NOBANK_VEC034 /**< TRAPA */
#define INTSPEC_NOBANK_VEC035 /**< TRAPA */
#define INTSPEC_NOBANK_VEC036 /**< TRAPA */
#define INTSPEC_NOBANK_VEC037 /**< TRAPA */
#define INTSPEC_NOBANK_VEC038 /**< TRAPA */
#define INTSPEC_NOBANK_VEC039 /**< TRAPA */
#define INTSPEC_NOBANK_VEC040 /**< TRAPA */
#define INTSPEC_NOBANK_VEC041 /**< TRAPA */
#define INTSPEC_NOBANK_VEC042 /**< TRAPA */
#define INTSPEC_NOBANK_VEC043 /**< TRAPA */
#define INTSPEC_NOBANK_VEC044 /**< TRAPA */
#define INTSPEC_NOBANK_VEC045 /**< TRAPA */
#define INTSPEC_NOBANK_VEC046 /**< TRAPA */
#define INTSPEC_NOBANK_VEC047 /**< TRAPA */
#define INTSPEC_NOBANK_VEC048 /**< TRAPA */
#define INTSPEC_NOBANK_VEC049 /**< TRAPA */
#define INTSPEC_NOBANK_VEC050 /**< TRAPA */
#define INTSPEC_NOBANK_VEC051 /**< TRAPA */
#define INTSPEC_NOBANK_VEC052 /**< TRAPA */
#define INTSPEC_NOBANK_VEC053 /**< TRAPA */
#define INTSPEC_NOBANK_VEC054 /**< TRAPA */
#define INTSPEC_NOBANK_VEC055 /**< TRAPA */
#define INTSPEC_NOBANK_VEC056 /**< TRAPA */
#define INTSPEC_NOBANK_VEC057 /**< TRAPA */
#define INTSPEC_NOBANK_VEC058 /**< TRAPA */
#define INTSPEC_NOBANK_VEC059 /**< TRAPA */
#define INTSPEC_NOBANK_VEC060 /**< TRAPA */
#define INTSPEC_NOBANK_VEC061 /**< TRAPA */
#define INTSPEC_NOBANK_VEC062 /**< TRAPA */
#define INTSPEC_NOBANK_VEC063 /**< TRAPA */
```

17.3.1 IBNR Register Addresses (INTSPEC_IBNR_ADR1 and INTSPEC_IBNR_ADR2)

The addresses for the IBNR register for CPUID#1 and the IBNR register for CPUID#2 of the interrupt controller are defined with constant expressions.

If there are no IBNR registers, that is, if a CPU not supporting the register banks is used, specify 0. In this case, the system.regbank setting in the cfg file loses meaning and all interrupts cannot use the register banks.

The IBNR register addresses for SH7205 and SH7265 are shown below for reference.

- CPU#0 (CPUID#1): 0xFFFD940E
- CPU#1 (CPUID#2): 0xFFFD950E

17.3.2 Vector Numbers That Cannot Use Register Banks (INTSPEC_NOBANK_VECxxx)

The vector numbers for causes that cannot use the register banks due to the CPU specifications, such as exceptions, NMI, and user breaks are defined. To be specific, define the "INTSPEC_NOBANK_VECxxx" macro for those vector numbers. "xxx" represents a vector number written as a three-digit decimal number.

17.4 kernel_def.c and kernel_cfg.c

These two files are used for importing the system definition files generated by cfg72mp. They are stored under the <SAMPLE_INST>\R0K572650D000BR\cpuid1\cfg_out\ directory (for CPUID#1) and the <SAMPLE_INST>\R0K572650D000BR\cpuid2\cfg_out\ directory (for CPUID#2). The user is not permitted to edit these files.

When compiling kernel_def.c and kernel_cfg.c, <RTOS_INST>\os\system\ must be specified as the include path.

When compiling kernel_def.c, the code=asmcode option also needs to be specified.

17.5 Sections

The user must allocate each section at a suitable address at linkage.

17.5.1 Rules for Section Names

Some sections are restricted. For example, some sections need consideration when allocating them to non-cacheable areas.

All sections provided by the HI7200/MP are named according to the following rules so that the user can allocate sections easily under such kind of restrictions. It is recommended for the application to also follow these rules.

```
PC_hiknl
```

(1) First character

- P: Program section
- C: Constant section
- B: Uninitialized data section
- D: Initialized data section (ROM section)
- R: Initialized data section (RAM section, which is generated with [ROM to RAM mapped sections] specified in the linkage editor)

(2) Second character

- C: Cacheable access enabled
- D: Cacheable access disabled
- L: Spinlock variable area (cacheable access disabled and must be accessed via the same bus by all CPUs)

Table 17.2 shows the on-chip RAM address space of the SH7265. In the SH7265, accesses to on-chip RAM are always handled as non-cacheable accesses. However, the section including spinlock variables need to be allocated to address B.

Table 17.2 On-Chip RAM Address Space of SH7265

Page	Address A (Non-Cacheable Access)	Address B (Non-Cacheable Access and Access via the Same Bus by All CPUs)
RAM0 page 0	0xFFF80000 to 0xFFF83FFF	0xFFD80000 to 0xFFD83FFF
RAM0 page 1	0xFFF84000 to 0xFFF87FFF	0xFFD84000 to 0xFFD87FFF
RAM0 page 2	0xFFF88000 to 0xFFF8BFFF	0xFFD88000 to 0xFFD8BFFF
RAM0 page 3	0xFFF8C000 to 0xFFF8FFFF	0xFFD8C000 to 0xFFD8FFFF
RAM1 page 0	0xFFFA0000 to 0xFFFA3FFF	0xFFDA0000 to 0xFFDA3FFF
RAM1 page 1	0xFFFA4000 to 0xFFFA7FFF	0xFFDA4000 to 0xFFDA7FFF

(3) Third character

Fixed to "_".

(4) Fourth and subsequent characters

As desired.

17.5.2 Sections

Table 17.3 Sections for Kernel Library, kernel_def.c, and kernel_cfg.c

Section	Remarks
PC_hiknl	Kernel program
CC_hicfg	Kernel internal data
CC_hijmptbl	Kernel internal data
CC_hivct	Kernel internal data*2
CC_hiinttbl	Kernel internal data
BC_hivct	Kernel internal data*2
BC_hiinttbl	Kernel internal data
BC_hiknlstk	Kernel stack
BC_hiirqstk	Interrupt stack
BC_hitmrstk	Timer stack
BC_hiwrk	Kernel internal data
BC_hirmtstk	SVC server task stack
BD_hirmtmpf	Fixed-sized memory pool area for remote SVC
BD_hiwrk	Kernel internal data
BD_hitooltrc	Tool trace area*3
BC_hitrcbuf	Target trace buffer
BL_S_hiwrk	Kernel internal data (only when MYCPUID = 1)*1
BC_hitskstk	Default task stack area
BC_hidtq	Default data queue area
BC_himbf	Default message buffer area
BC_himpf	Default fixed-sized memory pool area
BC_himpl	Default variable-sized memory pool area

Notes: 1. At linkage of CPUID#1, the symbol address file for this section needs to be output so that CPUID#2 can link that symbol address file.

2. In vsta_knl, the VBR register is initialized according to system.vector_type as follows:

(1) When system.vector_type is ROM or ROM_ONLY_DIRECT: Start address of CC_hivct section – 16

(2) When system.vector_type is RAM or RAM_ONLY_DIRECT: Start address of BC_hivct section – 16

3. The size of this section is four bytes. The kernel writes the trace data to this section.

Table 17.4 Sections for RPC Library and rpc_table.c

Section	Remarks
PC_rpc	
CC_rpc	
BC_rpc	
BL_S_rpc	Only when MYCPUID = 1*

Note: * At linkage of CPUID#1, the symbol address file for this section needs to be output so that CPUID#2 can link that symbol address file.

Table 17.5 Section for Spinlock Library

Section	Remarks
PC_spin	

Table 17.6 Sections for SH2A-DUAL Cache Support Library

Section	Remarks
PC_cache	
PD_cache	
CC_cache	

Table 17.7 Sections for IPI

Section	Remarks
PC_ipi	
CC_ipi	
BC_ipi	
BL_S_ipi	Only when MYCPUID = 1*

Note: * At linkage of CPUID#1, the symbol address file for this section needs to be output so that CPUID#2 can link that symbol address file.

Table 17.8 Sections for OAL

Section	Remarks
PC_oal	
CC_oal	
BC_oal	
BD_oalpool	Variable-sized memory pool area for OAL

Table 17.9 Sample Sections (CPUID#1)

Classification	Section	Remarks
Reset	CC_resetvct	Reset vector table
	PC_reset	
	CC_reset	
	BL_S_URAM0	Flag (in on-chip RAM0) used for synchronization of both CPUs at start-up*
	BD_URAM1	Area (in on-chip RAM1) to write the program to keep CPUID#2 waiting until CPUID#1 initializes the shared hardware
Sections created by cfg file	BC_TSKSTK	Stack used by the TaskSend() task
	BC_DTQ	Dummy data queue area
	BC_MBF	Dummy message buffer area
	B52	Dummy fixed-sized memory pool area
	BC_MPL	Dummy variable-sized memory pool area
Standard libraries, lowsrc.c, otherlib.c	PC_stdlib	
	CC_stdlib	
	DC_stdlib	
	BC_stdlib	
	RC_stdlib	Created by ROM option of linkage editor
	BC_heap	Heap area (lowsrc.c)
Timer driver	PC_tmrdrv	
	CC_tmrdrv	
System down	PC_sysdwn	
	BC_sysdwn	
Samples	PC_sample	
	CC_sample	
	BC_sample	
	BL_sample	rpc_info structure (init_task1.c) passed to rpc_init()
	BD_memcopy	Non-cacheable area used by RPC samples

Note: * At linkage of CPUID#1, the symbol address file for this section needs to be output so that CPUID#2 can link that symbol address file.

Table 17.10 Sample Sections (CPUID#2)

Classification	Section	Remarks
Reset	CC_vresetvct	Virtual reset vector table
	PC_reset	
	CC_reset	
Sections created by cfg file	BC_TSKSTK	Stack used by the TaskSend() task
	BC_DTQ	Dummy data queue area
	BC_MBF	Dummy message buffer area
	BD_MPF	Fixed-sized memory pool area used by remote service call examples
	BC_MPF	Dummy fixed-sized memory pool area
	BC_MPL	Dummy variable-sized memory pool area
Standard libraries, lowsrc.c, otherlib.c	PC_stdlib	
	CC_stdlib	
	DC_stdlib	
	BC_stdlib	
	RC_stdlib	Created by ROM option of linkage editor
	BC_heap	Heap area (lowsrc.c)
Timer driver	PC_tmrdrv	
	CC_tmrdrv	
System down	PC_sysdwn	
	BC_sysdwn	
Samples	PC_sample	
	CC_sample	
	BC_sample	
	BL_sample	rpc_info structure (init_task2.c) passed to rpc_init()
CPUID#1 symbol address file (prj_cpuid1.fsy)	P	P section with a size of 0 bytes

17.5.3 Common Symbols (Exporting Symbols from CPUID#1 to CPUID#2)

For symbols to be shared by both CPUs, the entities should be defined in CPUID#1. At this time, separate section names are assigned to the symbol entities. At linkage of CPUID#1, a specification is made to output the symbols in that section to the symbol address file (file extension: fsy).

In CPUID#2, register the symbol address file into a project as a file to be assembled. This enables a CPUID#2 program to reference CPUID#1 symbols.

Note that both CPUID#1 and CPUID#2 are not supposed to reference each other's symbols. This only causes dependencies to become recurrent.

17.5.4 Virtual Reset Vector Table of CPUID#2

The entity of the virtual reset vector table is created in CPUID#2 but the virtual reset vector table is referenced from CPUID#1 programs (cpuid1\reset\reset.src). Because CPUID#2 symbols being referenced by CPUID#1 is not permitted to avoid the build dependencies between CPUs becoming recurrent as described above, this sample has the following specifications.

(1) The address for allocating the virtual reset vector table should be determined in advance.

(2) At linkage of CPUID#1, the symbol of the virtual reset vector table should be forcibly defined in the address determined in (1).

(3) At linkage of CPUID#2, the section of the virtual reset vector table should be allocated to the address determined in (1).

17.5.5 Memory Map of this Sample

In this sample, the sections are allocated on the assumption of downloading this sample to SDRAM and executing it without using flash memory for facilitating initial evaluation of the board. Note the following.

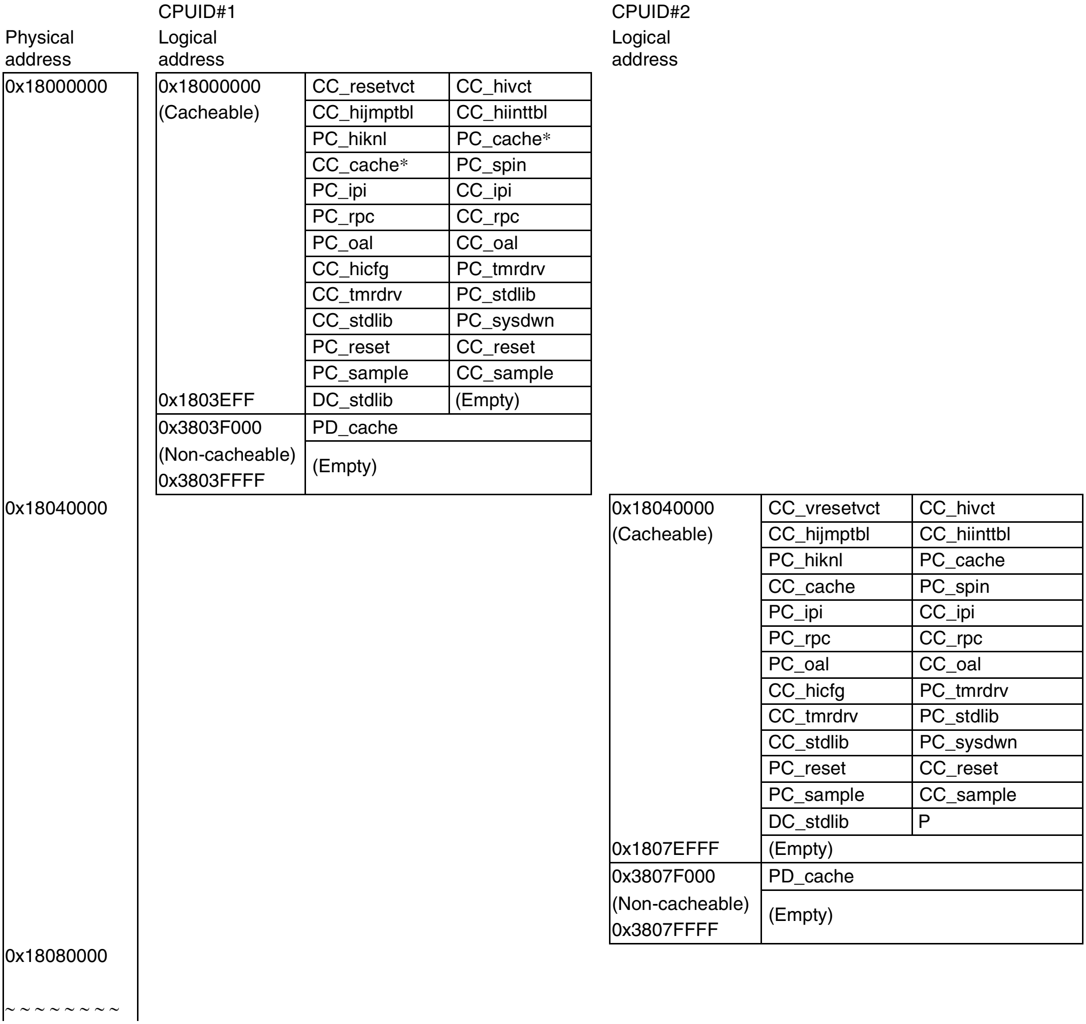
(1) Before downloading the sample, SDRAM must be initialized to make it accessible.

(2) Since the reset vector table (CC_resetvct section) cannot be allocated from address 0, this sample cannot be executed at a reset. Before executing the sample, use the debugger to manually initialize PC, SR, and R15 so they become the same values at a reset.

Sample batch files for facilitating the above items is provided in this product. For details, refer to section 17.10, Download to Target System.

(1) Allocation of Program and Constant Sections

These sections can be allocated in ROM at the final stage.



Physical address

Physical address
0x18000000
0x18040000
0x18080000
~ ~ ~ ~ ~ ~ ~ ~ ~

CPUID#1

Logical address		
0x18000000 (Cacheable)	CC_resetvct	CC_hivct
	CC_hijmptbl	CC_hiinttbl
	PC_hiknl	PC_cache*
	CC_cache*	PC_spin
	PC_ipi	CC_ipi
	PC_rpc	CC_rpc
	PC_oal	CC_oal
	CC_hicfg	PC_tmrdrv
	CC_tmrdrv	PC_stdlib
	CC_stdlib	PC_sysdwn
	PC_reset	CC_reset
	PC_sample	CC_sample
0x1803EFF	DC_stdlib	(Empty)
0x3803F000	PD_cache	
(Non-cacheable) 0x3803FFFF	(Empty)	

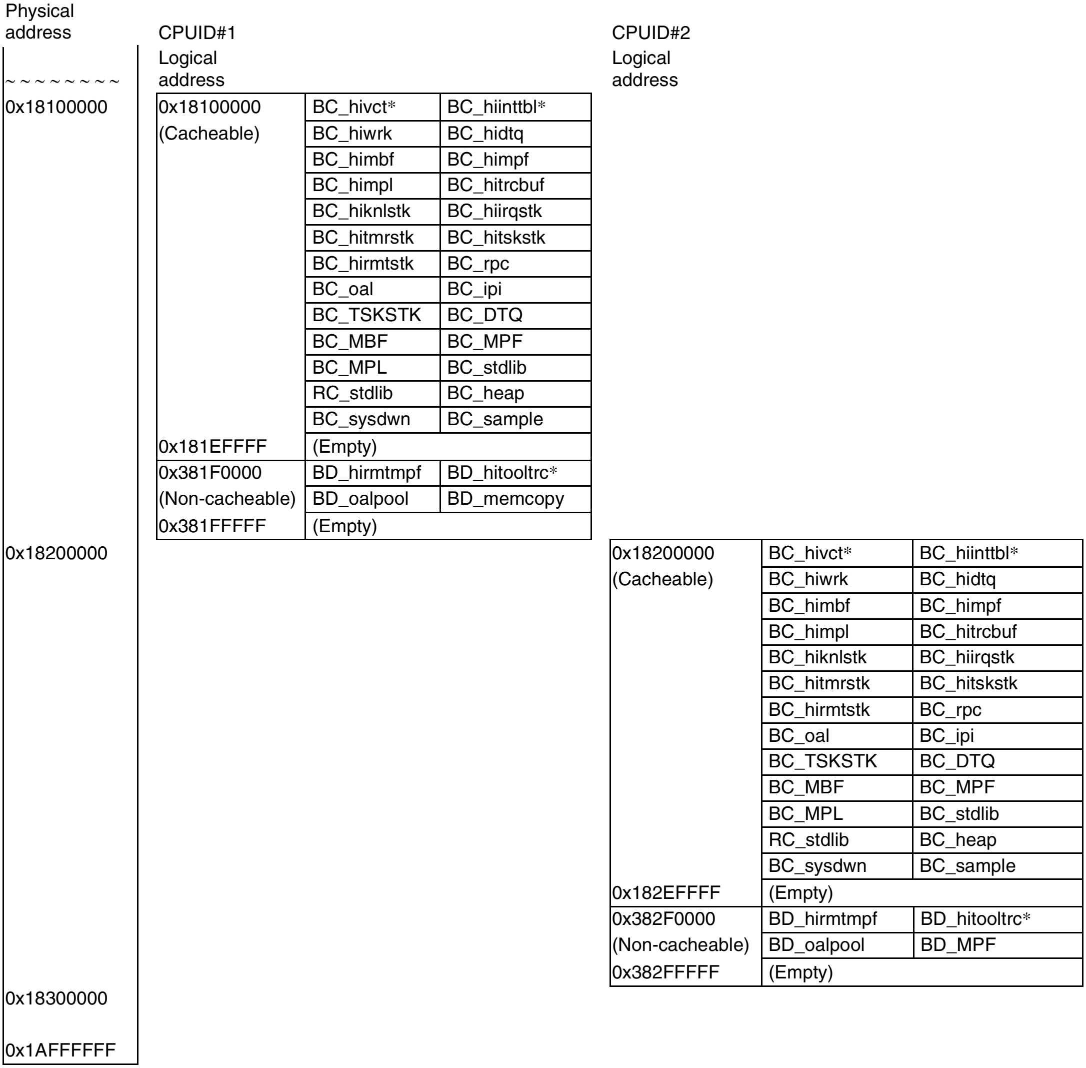
CPUID#2

Logical address		
0x18040000 (Cacheable)	CC_vresetvct	CC_hivct
	CC_hijmptbl	CC_hiinttbl
	PC_hiknl	PC_cache
	CC_cache	PC_spin
	PC_ipi	CC_ipi
	PC_rpc	CC_rpc
	PC_oal	CC_oal
	CC_hicfg	PC_tmrdrv
	CC_tmrdrv	PC_stdlib
	CC_stdlib	PC_sysdwn
	PC_reset	CC_reset
	PC_sample	CC_sample
	DC_stdlib	P
0x1807EFFF	(Empty)	
0x3807F000	PD_cache	
(Non-cacheable) 0x3807FFFF	(Empty)	

Note: * At shipment, the entity of this section does not exist.

Figure 17.12 Allocation of Program and Constant Sections

(2) Allocation of Variable Sections (SDRAM)



CPUID#1

Logical address		
0x18100000	BC_hivct*	BC_hiinttbl*
(Cacheable)	BC_hiwrk	BC_hidtq
	BC_himbf	BC_himpf
	BC_himpl	BC_hitrcbuf
	BC_hiknlstk	BC_hiirqstk
	BC_hitmrstk	BC_hitskstk
	BC_hirmtstk	BC_rpc
	BC_oal	BC_ipi
	BC_TSKSTK	BC_DTQ
	BC_MBF	BC_MPF
	BC_MPL	BC_stdlib
	RC_stdlib	BC_heap
	BC_sysdwn	BC_sample
0x181EFFFF	(Empty)	
0x381F0000	BD_hirmtmpf	BD_hitooltrc*
(Non-cacheable)	BD_oalpool	BD_memcopy
0x381FFFFF	(Empty)	

CPUID#2

Logical address		
0x18200000	BC_hivct*	BC_hiinttbl*
(Cacheable)	BC_hiwrk	BC_hidtq
	BC_himbf	BC_himpf
	BC_himpl	BC_hitrcbuf
	BC_hiknlstk	BC_hiirqstk
	BC_hitmrstk	BC_hitskstk
	BC_hirmtstk	BC_rpc
	BC_oal	BC_ipi
	BC_TSKSTK	BC_DTQ
	BC_MBF	BC_MPF
	BC_MPL	BC_stdlib
	RC_stdlib	BC_heap
	BC_sysdwn	BC_sample
0x182EFFFF	(Empty)	
0x382F0000	BD_hirmtmpf	BD_hitooltrc*
(Non-cacheable)	BD_oalpool	BD_MPF
0x382FFFFF	(Empty)	

Note: * At shipment, the entity of this section does not exist.

Figure 17.13 Allocation of Variable Sections (SDRAM)

(3) Allocation of Variable Sections (On-Chip RAM)

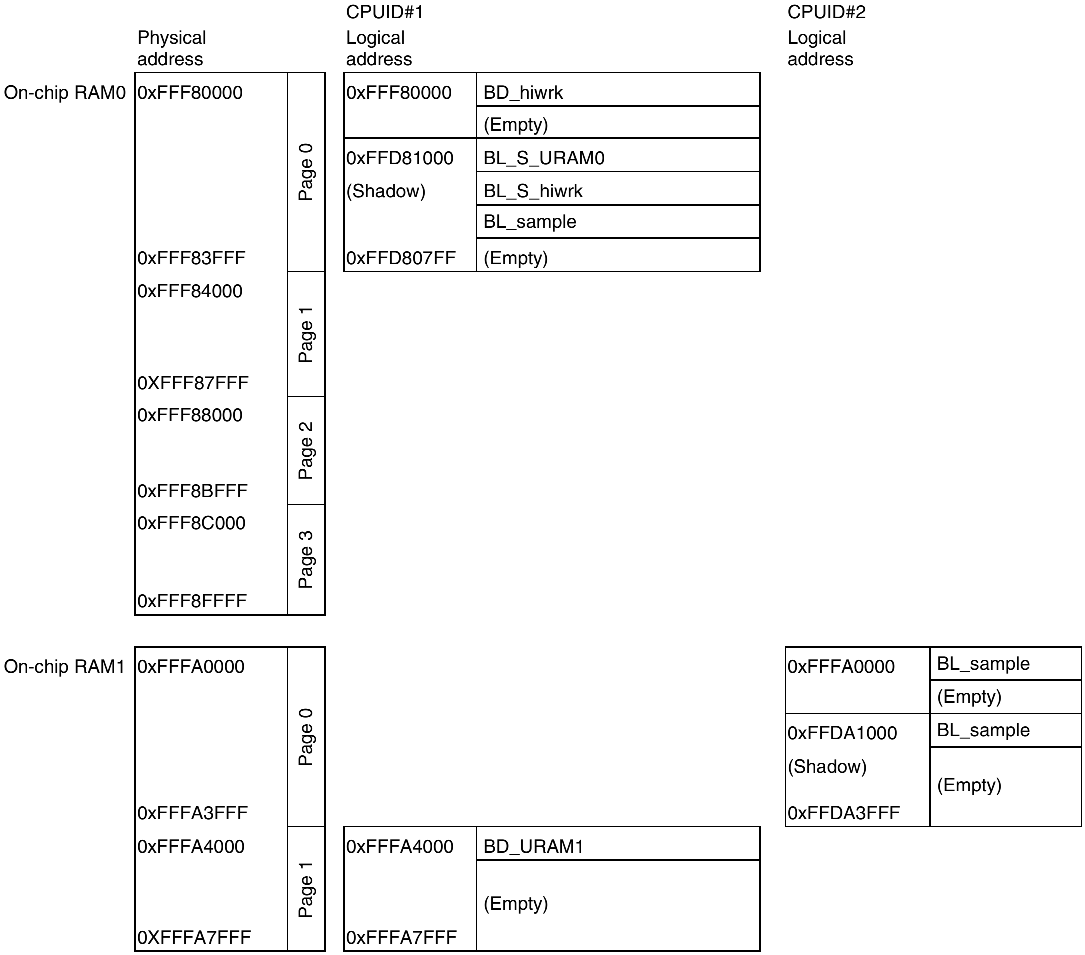


Figure 17.14 Allocation of Variable Sections (On-Chip RAM)

17.6 Kernel Library

The kernel library is divided into several files which must be input based on an appropriate priority. In the High-performance Embedded Workshop's library specification screen, the library with the highest priority is displayed at the top of the libraries. If the priority setting is incorrect, normal operation is not possible even though no error occurs at linkage.

Table 17.11 Kernel Library Priority

CPU Core in Use	Linkage Priority
SH-2A	hiknl.lib
SH2A-FPU	(1) fpu_knl.lib (2) hiknl.lib

17.7 Build Order of Each CPU

17.7.1 Basic Form

The basic form is to perform build of CPUID#1 first and then perform build of CPUID#2.

The symbol address file output from the optimizing linkage editor in CPUID#1 becomes the input to the assembler phase in CPUID#2 (figure 17.15) so that the CPUID#1 symbols can be exported to CPUID#2, as described in section 17.5.3, Common Symbols (Exporting Symbols from CPUID#1 to CPUID#2). Therefore, build of CPUID#2 must be performed after build of CPUID#1 has completed.

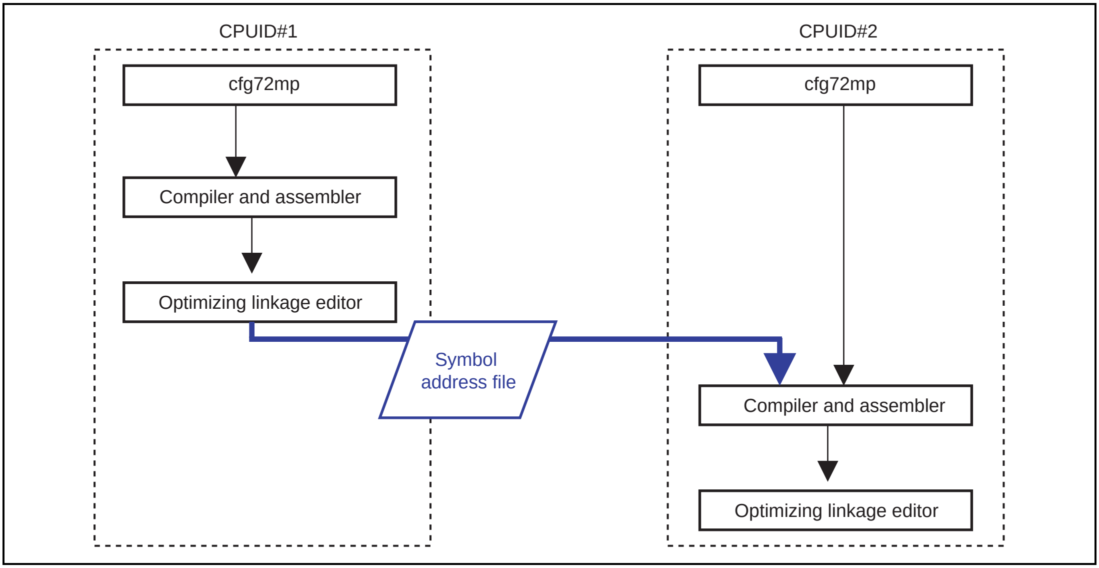


Figure 17.15 Dependencies between Build Phases of Each CPU (Basic Form)

17.7.2 Exporting the ID Name

(1) When CPUID#1 Includes an ID Name Header File of CPUID#2 (Deviation from Basic Form)

Figure 17.16 shows the dependencies between the build phases of each CPU. Before performing build of CPUID#1, cfg72mp in CPUID#2 must be executed to update the ID name header file of CPUID#2.

The sample has the form shown in the figure below.

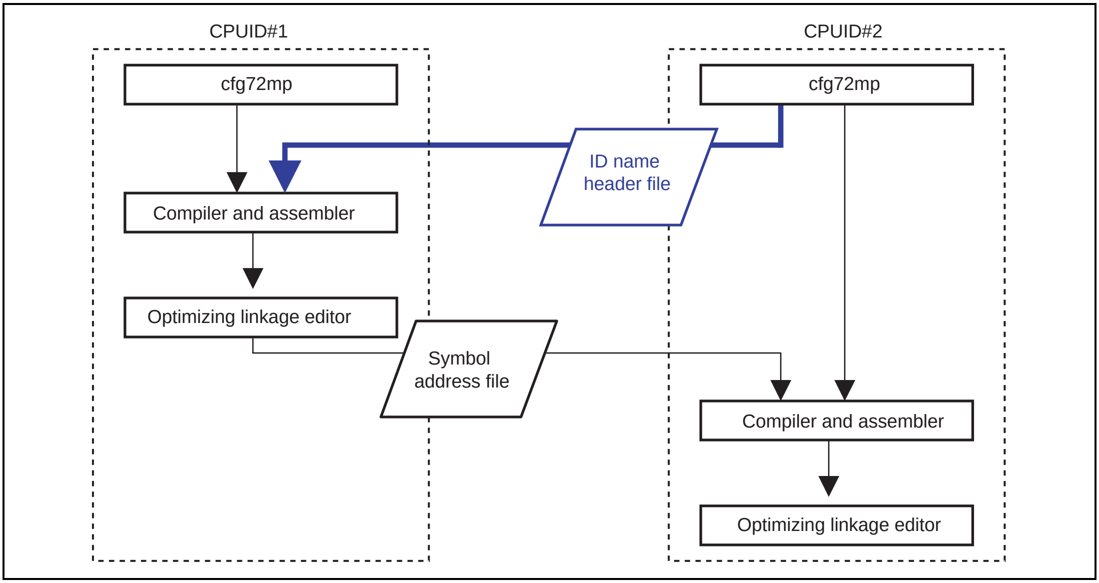


Figure 17.16 When CPUID#1 Includes an ID Name Header File of CPUID#2

(2) When CPUID#2 Includes an ID Name Header File of CPUID#1 (Same as Basic Form)

Figure 17.17 shows the dependencies between the build phases of each CPU. Before performing build of CPUID#2, cfg72mp in CPUID#1 must be executed to update the ID name header file of CPUID#1. The sample has the same form as the basic form.

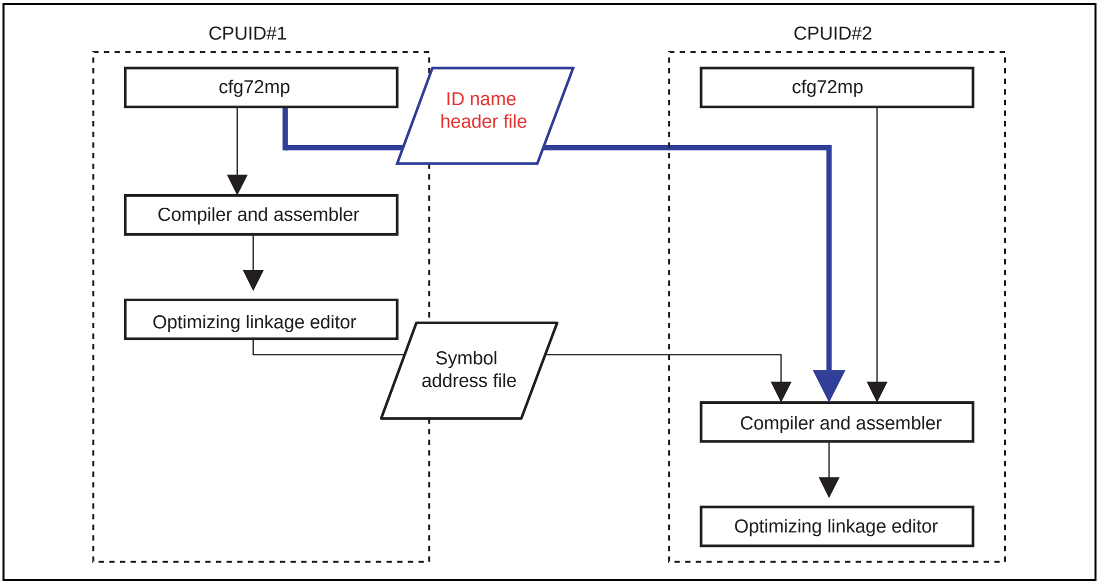


Figure 17.17 When CPUID#2 Includes an ID Name Header File of CPUID#1

(3) When Both CPUs Include ID Name Header Files of Each Other (Deviation from Basic Form)

Figure 17.18 shows the dependencies between the build phases of each CPU. First, execute cfg72mp in both CPUs to update the ID name header file of each CPU, and then perform build of each CPU in the basic form.

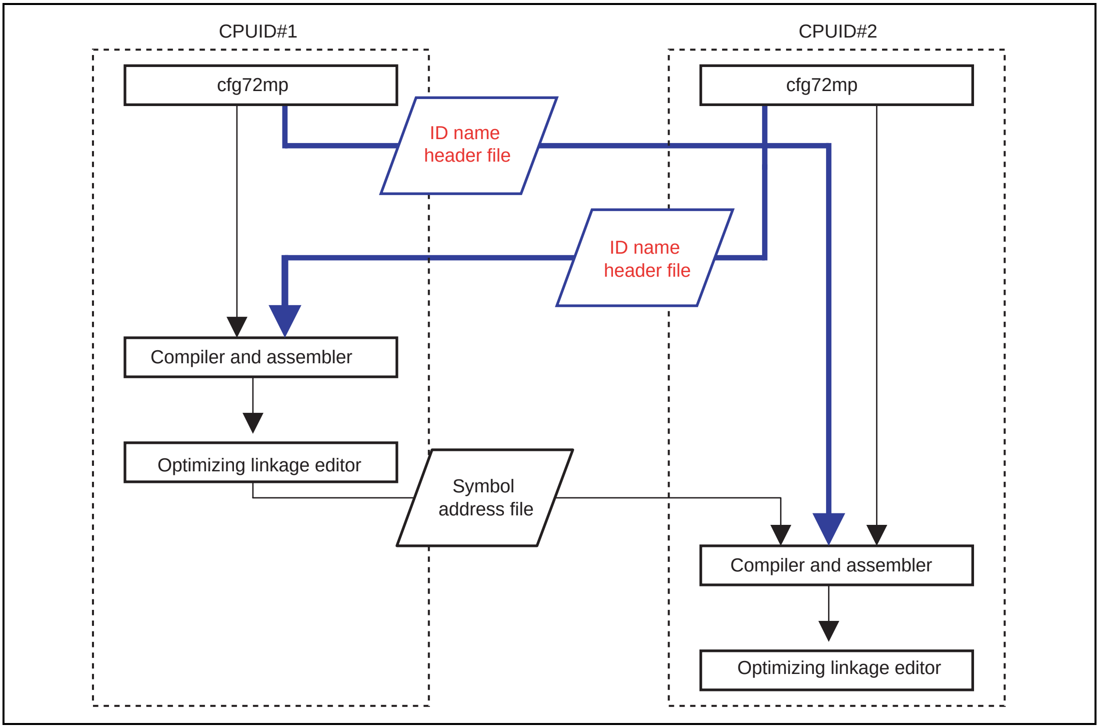


Figure 17.18 When Both CPUs Include ID Name Header Files of Each Other

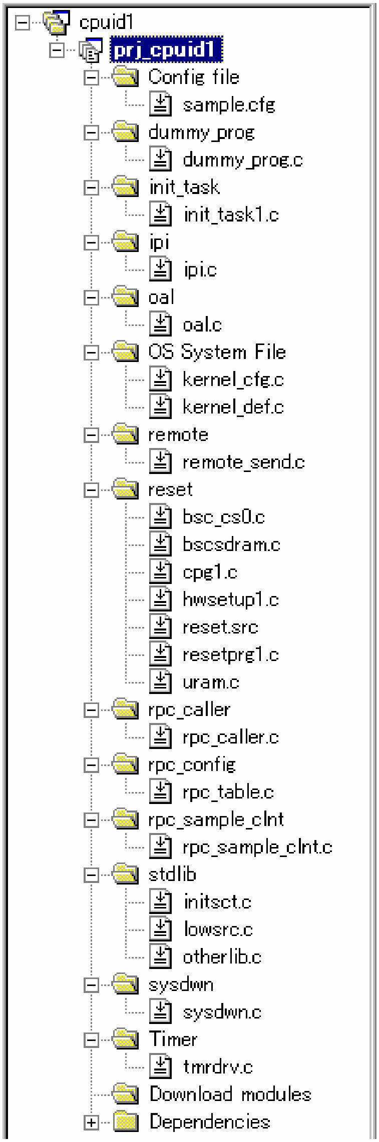
17.8 Description of Build of CPUID#1 (cpuid1\cpuid1.hws)

The workspace file in CPUID#1 is cpuid1\cpuid1.hws. Open the cpuid1.hws file. cpuid1.hws includes a project called "prj_cpuid1". Generate the load module files of CPUID#1 using this project.

The main settings for the provided project are explained in this section.

17.8.1 Registered Sources

The sources registered in the prj_cpuid1 project are shown in figure 17.19. All sources (C-language sources, assembly language sources, and cfg file) in the directories under cpuid1\ are registered. For each source, refer to section 16, Sample Programs.



```
cpuid1
  prj_cpuid1
    Config file
      sample.cfg
    dummy_prog
      dummy_prog.c
    init_task
      init_task1.c
    ipi
      ipi.c
    oal
      oal.c
    OS System File
      kernel_cfg.c
      kernel_def.c
    remote
      remote_send.c
    reset
      bsc_cs0.c
      bscsdram.c
      cpg1.c
      hwsetup1.c
      reset.src
      resetprg1.c
      uram.c
    rpc_caller
      rpc_caller.c
    rpc_config
      rpc_table.c
    rpc_sample_clnt
      rpc_sample_clnt.c
    stdlib
      initsct.c
      lowsrc.c
      otherlib.c
    sysdwn
      sysdwn.c
    Timer
      tmrdrv.c
    Download modules
    Dependencies
```

Figure 17.19 Sources Registered in prj_cpuid1 Project

Note the following.

(1) Config file (sample.cfg)

The cfg file which is stored in the cpuid1\cfg_out\ directory.

(2) OS System File (kernel_def.c and kernel_cfg.c)

These files which are stored in the cpuid1\cfg_out\ directory are used to include the cfg72mp output files. The user is not permitted to change these files.

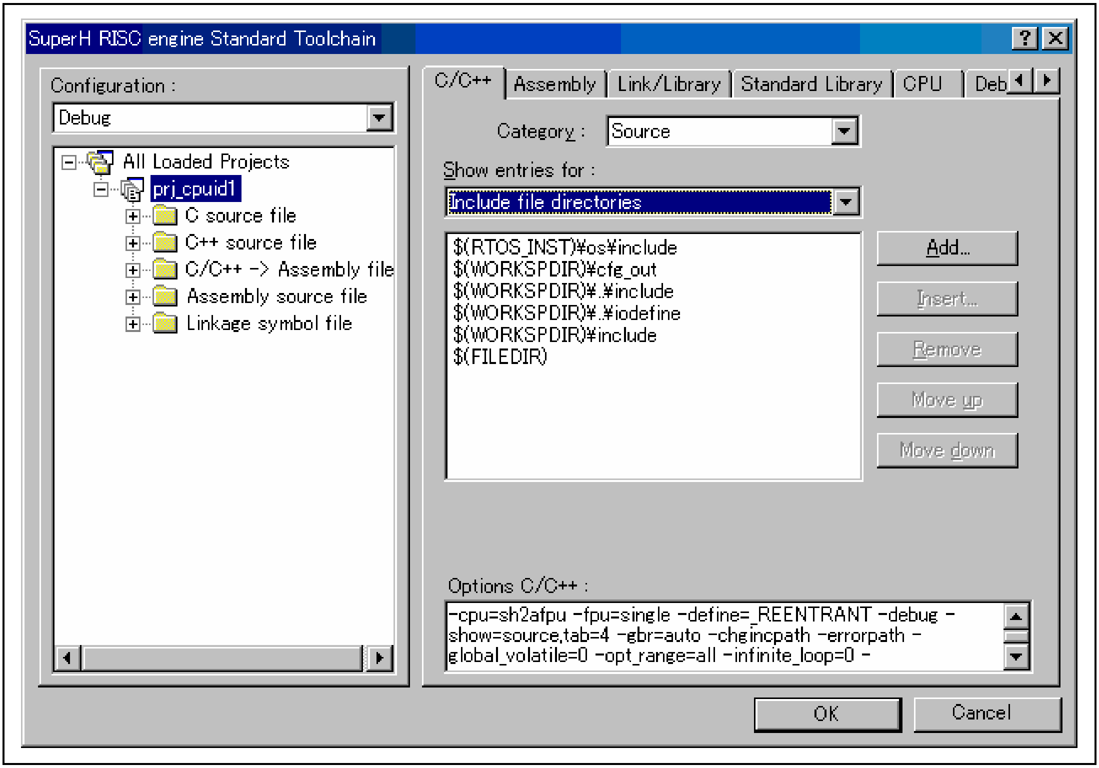
(3) cpuid1\remote_svc_sample\remote_send.c

This file includes kernel_id_cpu2.h which is generated by executing cfg72mp in CPUID#2. When cfg72mp is executed in CPUID#2, this file needs to be recompiled.

17.8.2 Compiler Options

(1) Include Directory

Figure 17.20 shows the common settings for all sources.



Configuration : Debug

Category : Source

Show entries for : Include file directories

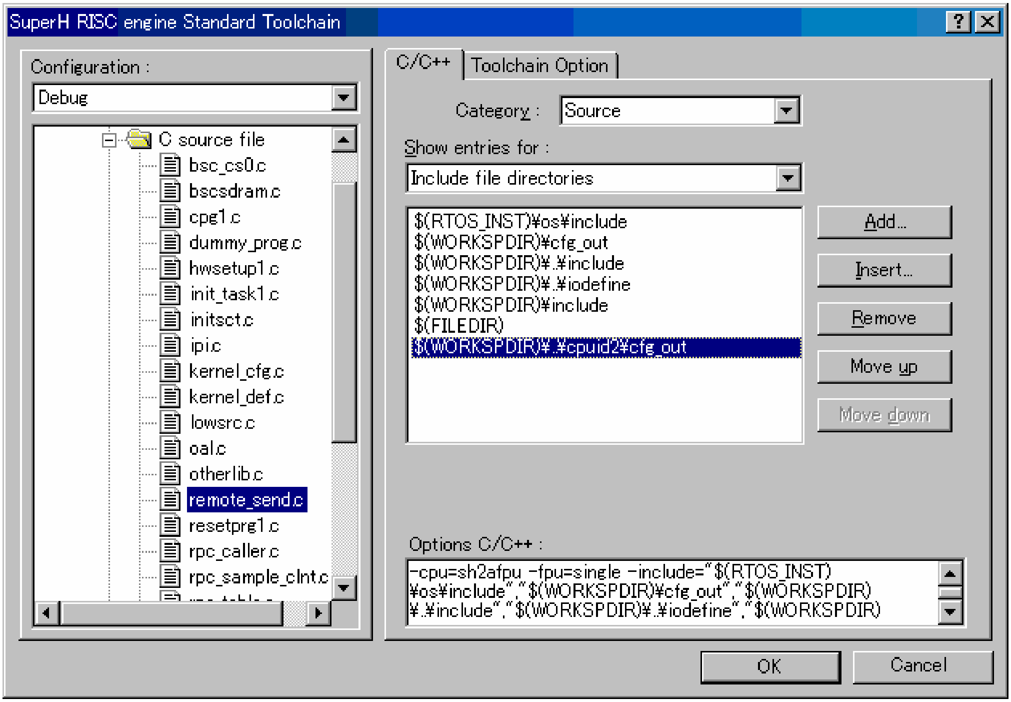
```
$(RTOS_INST)¥os¥include
$(WORKSPDIR)¥cfg_out
$(WORKSPDIR)¥..¥include
$(WORKSPDIR)¥..¥iodefine
$(WORKSPDIR)¥include
$(FILEDIR)
```

Options C/C++ :

```
-cpu=sh2afpu -fpu=single -define=_REENTRANT -debug -
show=source,tab=4 -gbr=auto -chgincpath -errorpath -
global_volatile=0 -opt_range=all -infinite_loop=0 -
```

Figure 17.20 Include File Directories of Compiler (Common Settings)

Since remote_send.c includes kernel_id_cpu2.h of CPUID#2, its storage path is added as shown in figure 17.21. "$(WORKSPDIR)\cfg_out" (cpuid1\cfg_out\) and "$(WORKSPDIR)\..\cpuid2\cfg_out" (cpuid2\cfg_out\) contain respective files for CPUID#1 and CPUID#2 which have the same file name, e.g. kernel_id.h. Accordingly, "$(WORKSPDIR)\cfg_out" for CPUID#1 must be given priority (must be displayed at a higher position in the screen).



Configuration : Debug

bsc_cs0.c
bscsdram.c
cpg1.c
dummy_prog.c
hwsetup1.c
init_task1.c
initsct.c
ipi.c
kernel_cfg.c
kernel_def.c
lowsrc.c
oal.c
otherlib.c
remote_send.c
resetprg1.c
rpc_caller.c
rpc_sample_clnt.c

Category : Source

Show entries for : Include file directories

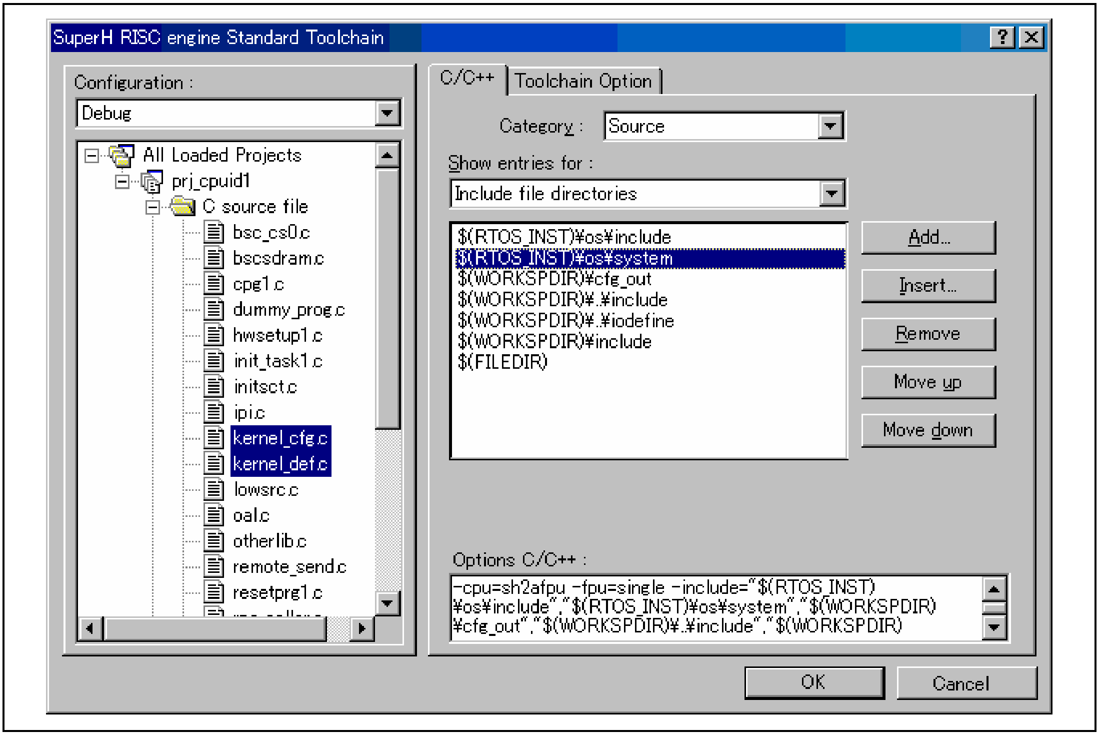
```
$(RTOS_INST)¥os¥include
$(WORKSPDIR)¥cfg_out
$(WORKSPDIR)¥.¥include
$(WORKSPDIR)¥.¥iodefine
$(WORKSPDIR)¥include
$(FILEDIR)
$(WORKSPDIR)¥.¥cpuid2¥cfg_out
```

Options C/C++ :

```
-cpu=sh2afpu -fpu=single -include="$(RTOS_INST)
¥os¥include","$(WORKSPDIR)¥cfg_out","$(WORKSPDIR)
¥.¥include","$(WORKSPDIR)¥.¥iodefine","$(WORKSPDIR)
```

Figure 17.21 Include File Directories of Compiler (remote_send.c)

"$(RTOS_INST)\os\system" is defined for kernel_cfg.c and kernel_def.c.



Configuration : Debug

```
All Loaded Projects
  prj_cpuid1
    C source file
      bsc_cs0.c
      bscsdram.c
      cpg1.c
      dummy_prog.c
      hwsetup1.c
      init_task1.c
      initsct.c
      ipi.c
      kernel_cfg.c
      kernel_def.c
      lowsrc.c
      oal.c
      otherlib.c
      remote_send.c
      resetprg1.c
```

Category : Source

Show entries for : Include file directories

```
$(RTOS_INST)¥os¥include
$(RTOS_INST)¥os¥system
$(WORKSPDIR)¥cfg_out
$(WORKSPDIR)¥.¥include
$(WORKSPDIR)¥.¥iodefine
$(WORKSPDIR)¥include
$(FILEDIR)
```

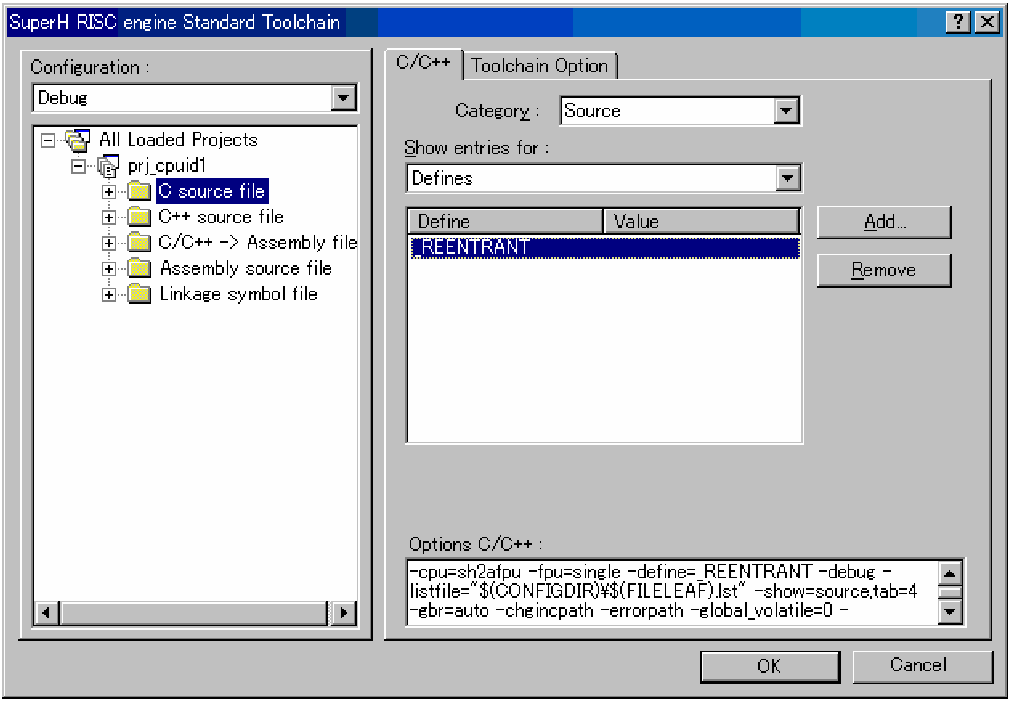
Options C/C++ :

```
-cpu=sh2afpu -fpu=single -include="$(RTOS_INST)
¥os¥include","$(RTOS_INST)¥os¥system","$(WORKSPDIR)
¥cfg_out","$(WORKSPDIR)¥.¥include","$(WORKSPDIR)
```

Figure 17.22 Include File Directories of Compiler (kernel_cfg.c, kernel_def.c)

(2) Macro Definitions

In this sample, the "_REENTRANT" macro is defined because the standard library is used as a reentrant library as shown in figure 17.23.



Configuration : Debug

Category : Source

Show entries for : Defines

Define	Value
_REENTRANT	

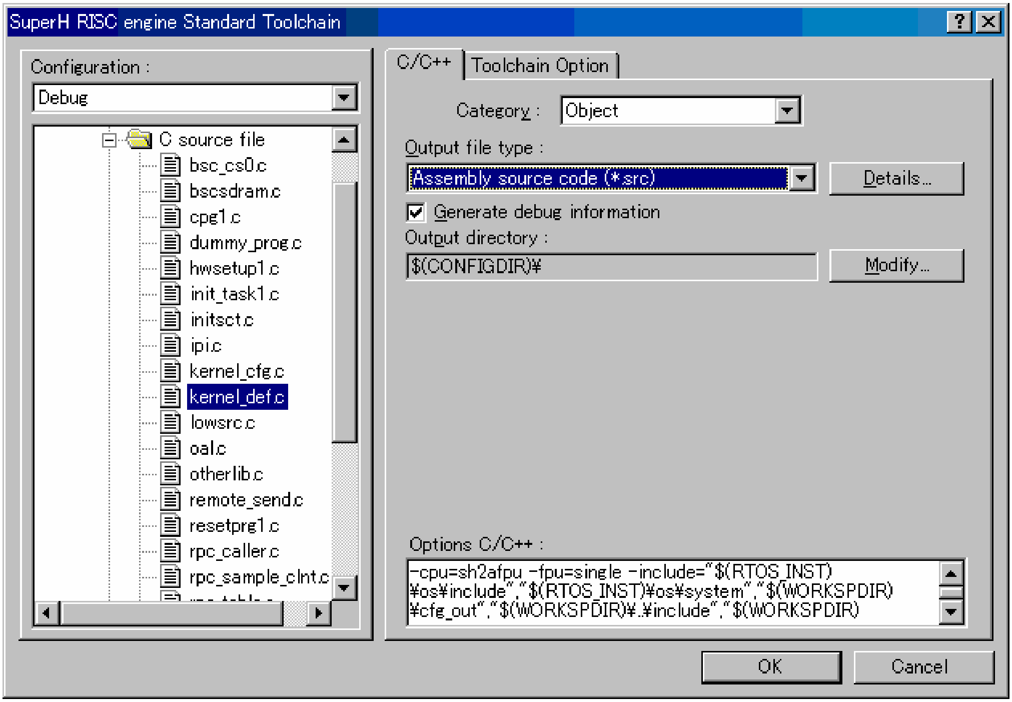
Options C/C++ :

```
-cpu=sh2afpu -fpu=single -define=_REENTRANT -debug -
listfile="$(CONFIGDIR)¥$(FILELEAF).lst" -show=source,tab=4
-gbr=auto -chgincpath -errorpath -global_volatile=0 -
```

Figure 17.23 Macro Definitions in Compiler

(3) Output File Type

The output file type is "Assembly source code" for kernel_def.c only because it uses inline assemble as shown in figure 17.24.



Configuration : Debug

C source file
- bsc_cs0.c
- bscsdram.c
- cpg1.c
- dummy_prog.c
- hwsetup1.c
- init_task1.c
- initsct.c
- ipi.c
- kernel_cfg.c
- kernel_def.c
- lowsrc.c
- oal.c
- otherlib.c
- remote_send.c
- resetprg1.c
- rpc_caller.c
- rpc_sample_clnt.c

Category : Object

Output file type : Assembly source code (*.src)

☑ Generate debug information

Output directory : $(CONFIGDIR)¥

Options C/C++ :

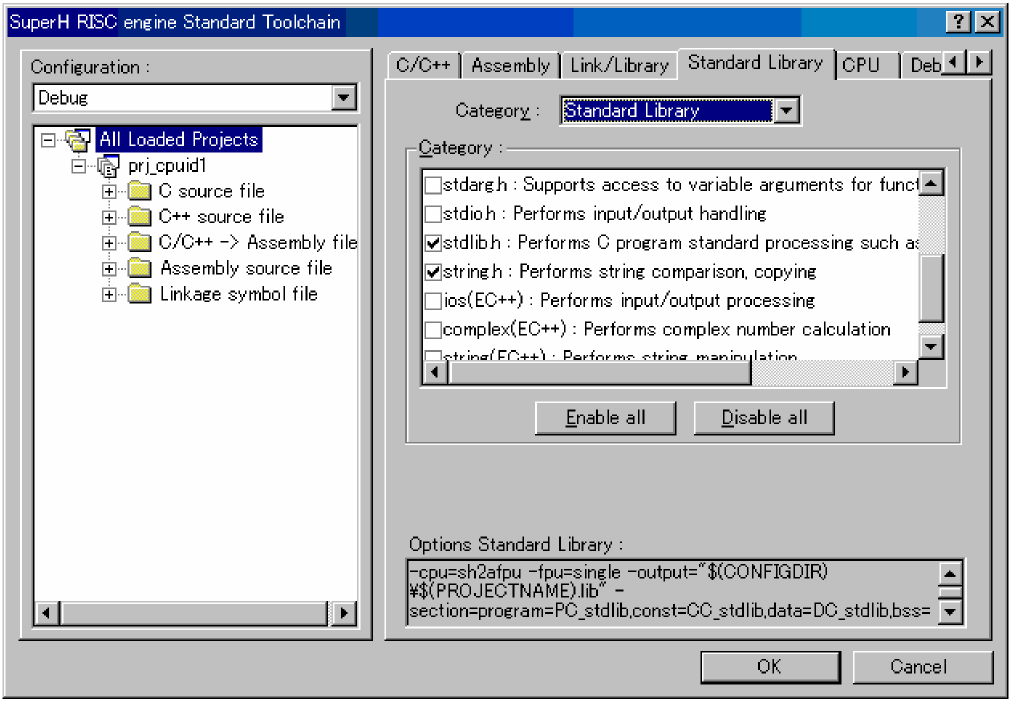
```
-cpu=sh2afpu -fpu=single -include="$(RTOS_INST)
¥os¥include","$(RTOS_INST)¥os¥system","$(WORKSPDIR)
¥cfg_out","$(WORKSPDIR)¥.¥include","$(WORKSPDIR)
```

Figure 17.24 Output File Type in Compiler

17.8.3 Standard Library Generator

(1) Embedded Standard Library Functions

As described in section 16.7, Standard Libraries, only stdlib.h and string.h are selected in this sample as shown in figure 17.25.



Configuration : Debug

Category : Standard Library

Category :

- ☐ stdarg.h : Supports access to variable arguments for funct
- ☐ stdio.h : Performs input/output handling
- ☑ stdlib.h : Performs C program standard processing such as
- ☑ string.h : Performs string comparison, copying
- ☐ ios(EC++) : Performs input/output processing
- ☐ complex(EC++) : Performs complex number calculation
- ☐ string(EC++) : Performs string manipulation

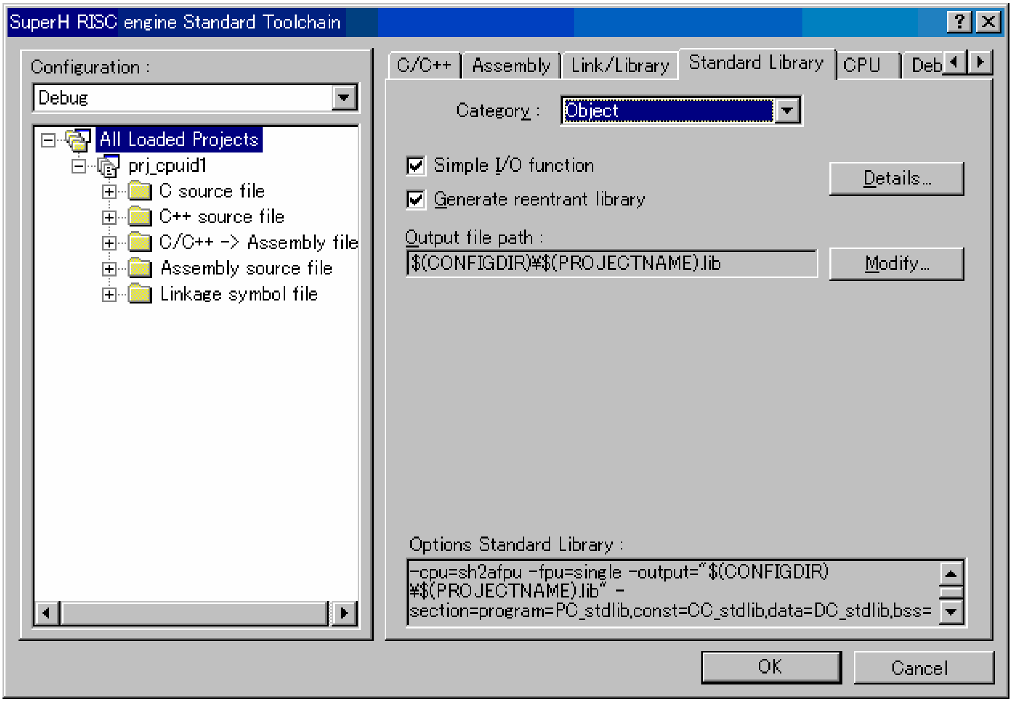
Options Standard Library :

```
-cpu=sh2afpu -fpu=single -output="$(CONFIGDIR)
¥$(PROJECTNAME).lib" -
section=program=PC_stdlib,const=CC_stdlib,data=DC_stdlib,bss=
```

Figure 17.25 Library Function Selection in Standard Library Generator

(2) Object

An object is generated as a reentrant library as shown in figure 17.26.



Configuration : Debug

Category : Object

☑ Simple I/O function

☑ Generate reentrant library

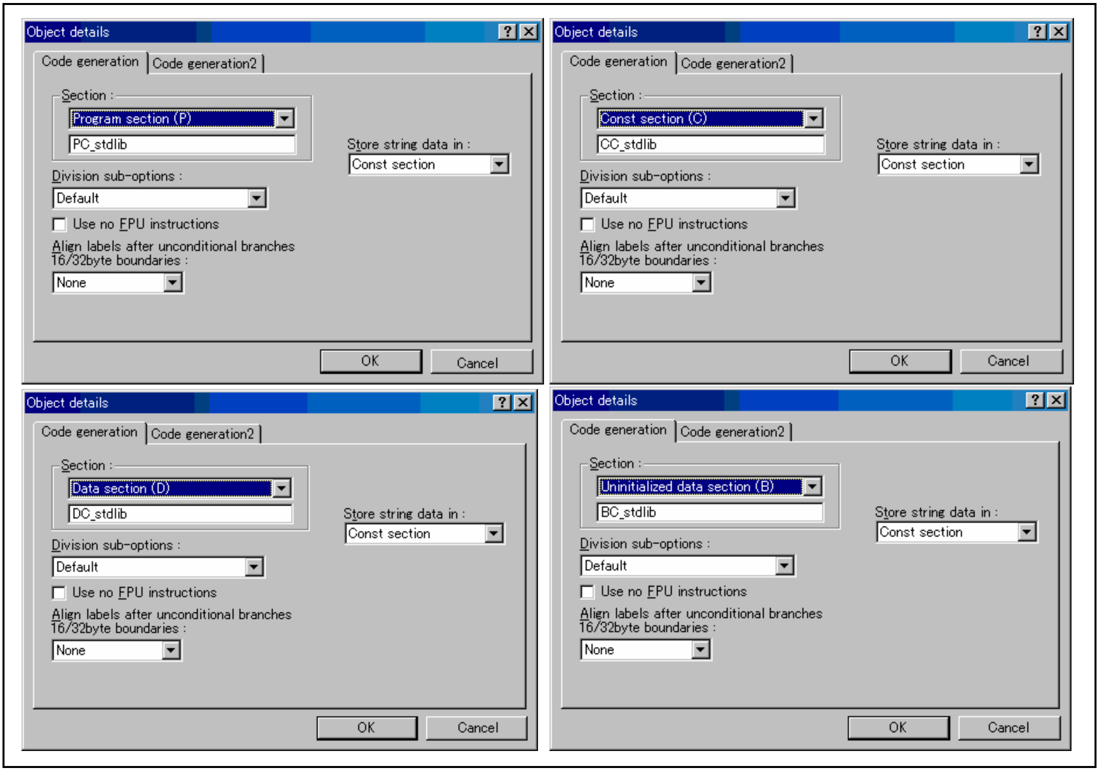
Output file path : $(CONFIGDIR)¥$(PROJECTNAME).lib

Options Standard Library :

```
-cpu=sh2afpu -fpu=single -output="$(CONFIGDIR)
¥$(PROJECTNAME).lib" -
section=program=PC_stdlib,const=CC_stdlib,data=DC_stdlib,bss=
```

Figure 17.26 Reentrant Library in Standard Library Generator

In the [Object details] dialog box opened by clicking the [Details...] button, the section names are set as shown in figure 17.27. The same section names are also used in lowsrc.c and otherlib.c.



Section : Program section (P)
PC_stdlib
Store string data in : Const section
Division sub-options : Default
□ Use no FPU instructions
Align labels after unconditional branches 16/32byte boundaries : None

Section : Const section (C)
CC_stdlib
Store string data in : Const section
Division sub-options : Default
□ Use no FPU instructions
Align labels after unconditional branches 16/32byte boundaries : None

Section : Data section (D)
DC_stdlib
Store string data in : Const section
Division sub-options : Default
□ Use no FPU instructions
Align labels after unconditional branches 16/32byte boundaries : None

Section : Uninitialized data section (B)
BC_stdlib
Store string data in : Const section
Division sub-options : Default
□ Use no FPU instructions
Align labels after unconditional branches 16/32byte boundaries : None

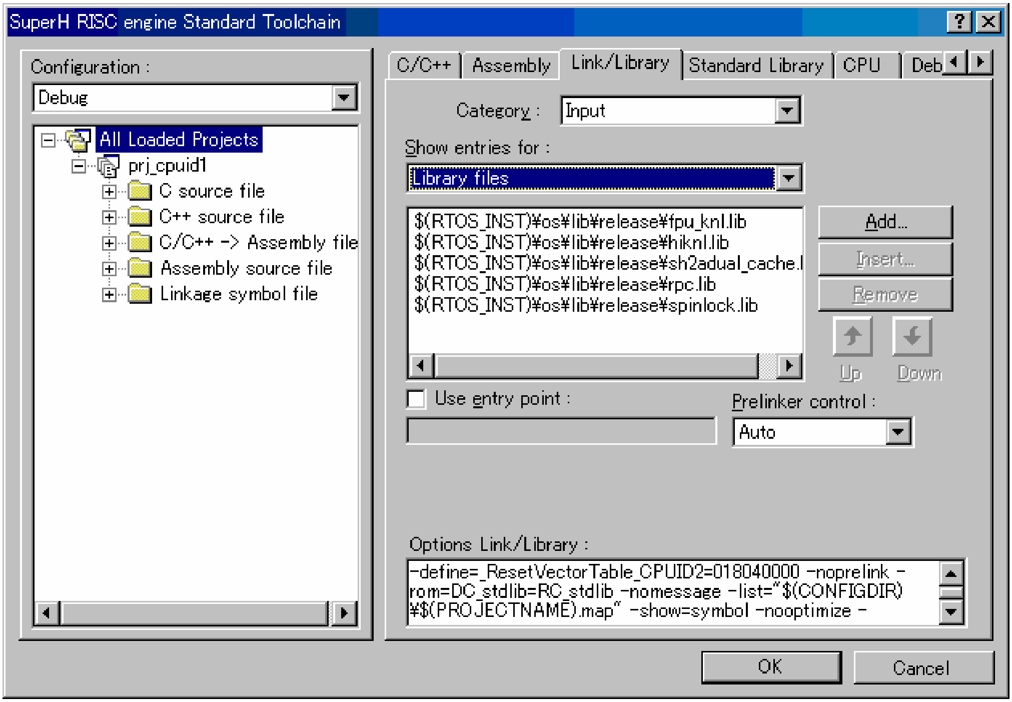
Figure 17.27 Section Name Setting in Standard Library Generator

17.8.4 Optimizing Linkage Editor

(1) Library Input

The following libraries provided by the HI7200/MP are input as shown in figure 17.28. For the kernel library input, refer to section 17.6, Kernel Library.

- fpu_knl.lib (kernel)
- hiknl.lib (kernel)
- sh2adual_cache.lib (SH2A-DUAL cache support library)
- rpc.lib (RPC library)
- spinlock.lib (spinlock library)



Configuration : Debug

Category : Input

Show entries for : Library files

```
$(RTOS_INST)¥os¥lib¥release¥fpu_knl.lib
$(RTOS_INST)¥os¥lib¥release¥hiknl.lib
$(RTOS_INST)¥os¥lib¥release¥sh2adual_cache.l
$(RTOS_INST)¥os¥lib¥release¥rpc.lib
$(RTOS_INST)¥os¥lib¥release¥spinlock.lib
```

☐ Use entry point :

Prelinker control : Auto

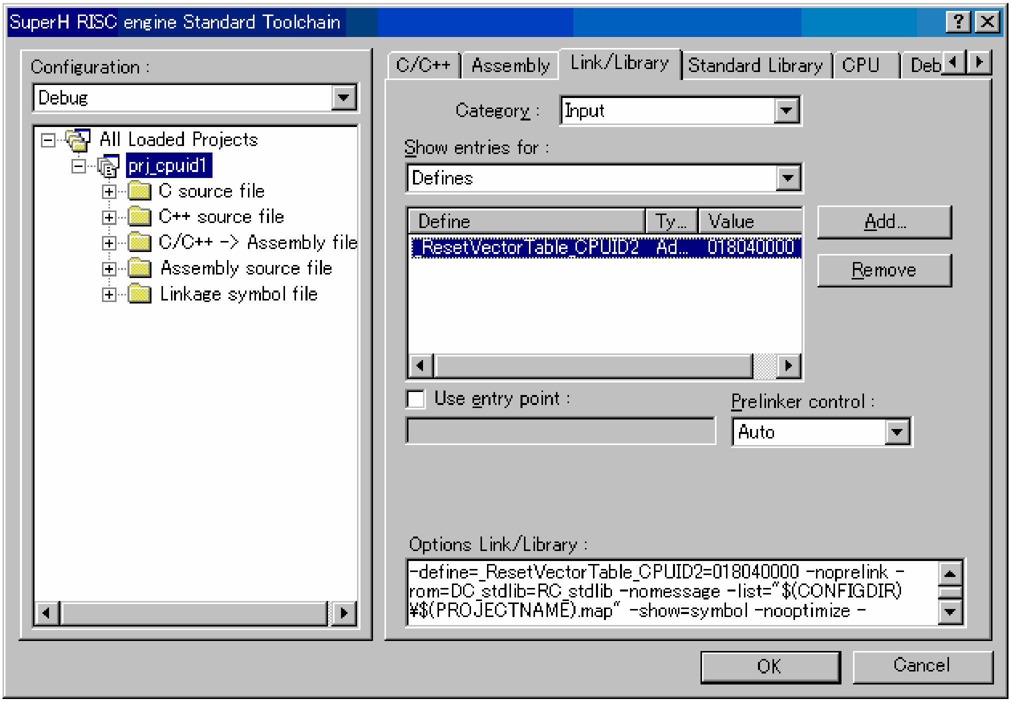
Options Link/Library :

```
-define=_ResetVectorTable_CPUID2=018040000 -noprelink -
rom=DC_stdlib=RC_stdlib -nomessage -list="$(CONFIGDIR)
¥$(PROJECTNAME).map" -show=symbol -nooptimize -
```

Figure 17.28 Library Input in Optimizing Linkage Editor

(2) Symbol Definition for Virtual Reset Vector Table of CPUID#2

As described in section 17.5.4, Virtual Reset Vector Table of CPUID#2, the address for the "_ResetVectorTable_CPUID2" symbol of the virtual reset vector table of CPUID#2 is forcibly defined as 0x18040000 as shown in figure 17.29.



Configuration : Debug

Category : Input

Show entries for : Defines

Define	Ty...	Value
_ResetVectorTable_CPUID2	Ad...	018040000

☐ Use entry point :

Prelinker control : Auto

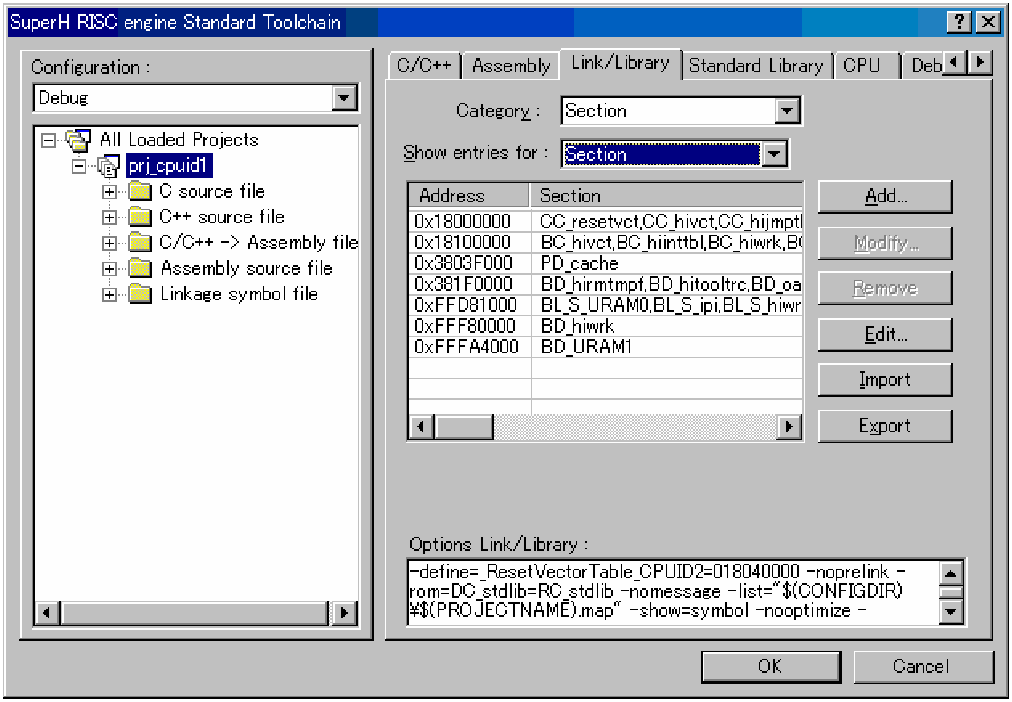
Options Link/Library :

```
-define=_ResetVectorTable_CPUID2=018040000 -noprelink -
rom=DC_stdlib=RC_stdlib -nomessage -list="$(CONFIGDIR)
¥$(PROJECTNAME).map" -show=symbol -nooptimize -
```

Figure 17.29 Symbol Definition for Virtual Reset Vector Table of CPUID#2 in Optimizing Linkage Editor

(3) Section Allocation

Though not all sections can be confirmed in figure 17.30, sections are allocated as described in section 17.5.5, Memory Map of this Sample.



Configuration : Debug

Category : Section

Show entries for : Section

Address	Section
0x18000000	CC_resetvct,CC_hivct,CC_hijmptl
0x18100000	BC_hivct,BC_hiinttbl,BC_hiwrk,B(
0x3803F000	PD_cache
0x381F0000	BD_hirmtmpf,BD_hitooltrc,BD_oa
0xFFD81000	BL_S_URAM0,BL_S_ipi,BL_S_hiwr
0xFFF80000	BD_hiwrk
0xFFFA4000	BD_URAM1

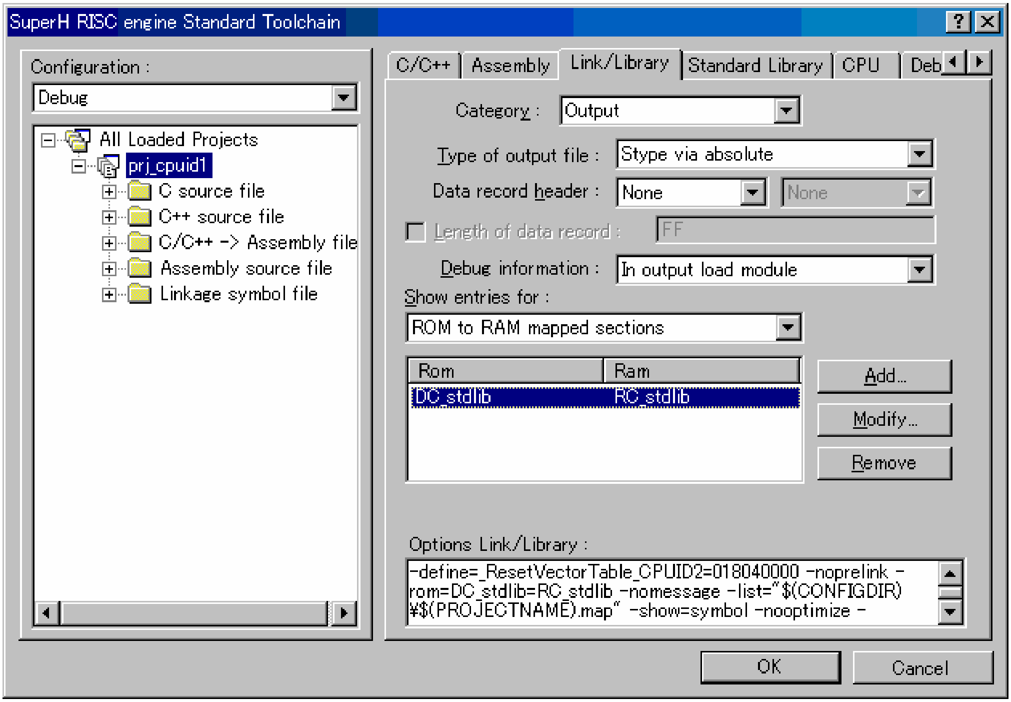
Options Link/Library :

```
-define=_ResetVectorTable_CPUID2=018040000 -noprelink -
rom=DC_stdlib=RC_stdlib -nomessage -list="$(CONFIGDIR)
¥$(PROJECTNAME).map" -show=symbol -nooptimize -
```

Figure 17.30 Section Allocation in Optimizing Linkage Editor

(4) ROM to RAM Mapping

Sections for which ROM to RAM mapping has to be performed, such as an initialized data section, are set in this sample as shown in figure 17.31.



Configuration : Debug

Category : Output

Type of output file : Stype via absolute

Data record header : None | None

□ Length of data record : FF

Debug information : In output load module

Show entries for : ROM to RAM mapped sections

Rom	Ram
DC_stdlib	RC_stdlib

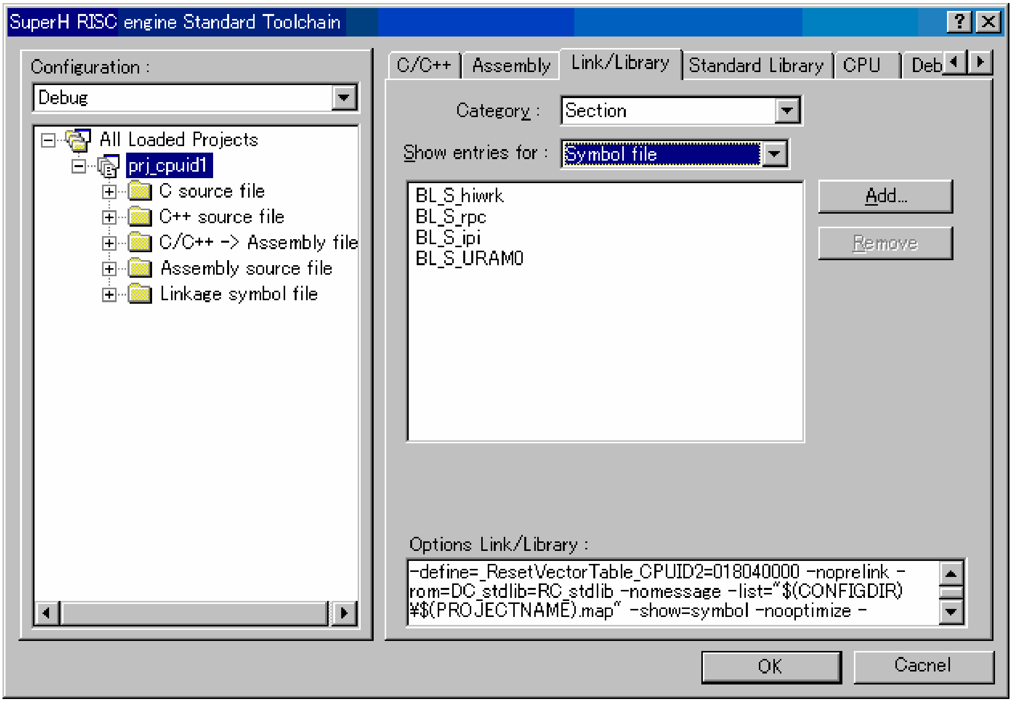
Options Link/Library :

```
-define=_ResetVectorTable_CPUID2=018040000 -noprelink -
rom=DC_stdlib=RC_stdlib -nomessage -list="$(CONFIGDIR)
¥$(PROJECTNAME).map" -show=symbol -nooptimize -
```

Figure 17.31 ROM to RAM Mapping in Optimizing Linkage Editor

(5) Output of Symbol Address File

As described in section 17.5.3, Common Symbols (Exporting Symbols from CPUID#1 to CPUID#2), a setting is made to make the CPUID#1 symbols open to CPUID#2 as shown in figure 17.32. Note that the symbol address file is generated with a file name of "prj_cpuid1.fsy" under the cpuid1\prj_cpuid1\debug\ directory which is the High-performance Embedded Workshop configuration directory. prj_cpuid1.fsy is used as a source of CPUID#2.


SuperH RISC engine Standard Toolchain

Configuration : Debug

All Loaded Projects
- prj_cpuid1
  - C source file
  - C++ source file
  - C/C++ -> Assembly file
  - Assembly source file
  - Linkage symbol file

Category : Section

Show entries for : Symbol file

BL_S_hiwrk
BL_S_rpc
BL_S_ipi
BL_S_URAM0

Options Link/Library :

```
-define=_ResetVectorTable_CPUID2=018040000 -noprelink -
rom=DC_stdlib=RC_stdlib -nomessage -list="$(CONFIGDIR)
¥$(PROJECTNAME).map" -show=symbol -nooptimize -
```

Figure 17.32 Output of Symbol Address File in Optimizing Linkage Editor

(6) Notes

1. L1100 warning

   The L1100 warning (shown below) meaning that the specified section could not be found may be output sometimes at linkage.

   ```
   L1100 (W) Cannot find "PC_cache" specified in option "start"
   ```

   If the section that could not be found is a section listed in section 17.5.2, Sections, this is not a problem because it does not exist in some cases depending on configuration.

2. L1320 warning

   When more than one kernel library is specified, the L1320 warning (shown below) may be output for a number of times at linkage. This is because the kernel adopts an implementation method in which the same symbols and different programs are stored in more than one library file. There is no problem with the generated load module.

   ```
   L1320 (W) Duplicate symbol "__kernel_act_tsk" in "C:\…fpu_knl.lib(fpu_acttsk)"
   ```

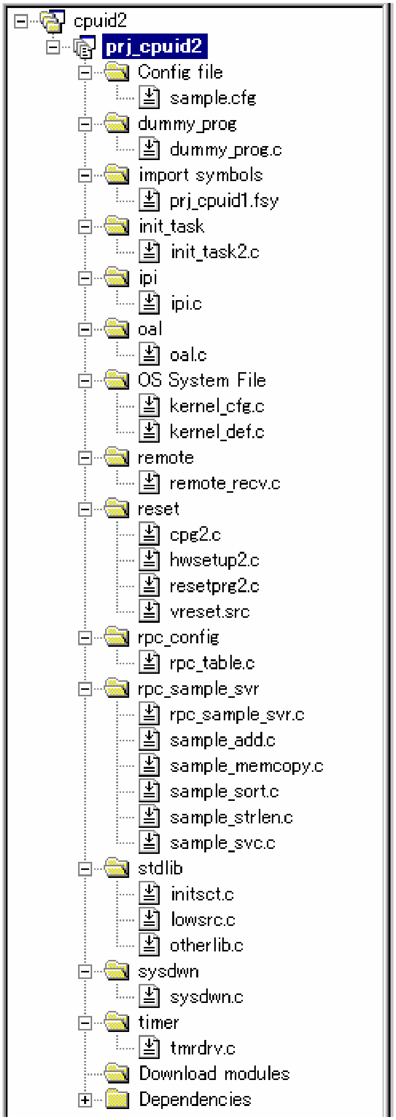
17.9 Description of Build of CPUID#2 (cpuid2\cpuid2.hws)

The workspace file in CPUID#2 is cpuid2\cpuid2.hws. Open the cpuid2.hws file. cpuid2.hws includes a project called "prj_cpuid2". Generate the load module files of CPUID#2 using this project.

The main settings for the provided project are explained in this section.

17.9.1 Registered Sources

The sources registered in the prj_cpuid2 project are shown in figure 17.33. All sources (C-language sources, assembly language sources, and cfg file) in the directories under cpuid2\ are registered. For each source, refer to section 16, Sample Programs.



```
cpuid2
  prj_cpuid2
    Config file
      sample.cfg
    dummy_prog
      dummy_prog.c
    import symbols
      prj_cpuid1.fsy
    init_task
      init_task2.c
    ipi
      ipi.c
    oal
      oal.c
    OS System File
      kernel_cfg.c
      kernel_def.c
    remote
      remote_recv.c
    reset
      cpg2.c
      hwsetup2.c
      resetprg2.c
      vreset.src
    rpc_config
      rpc_table.c
    rpc_sample_svr
      rpc_sample_svr.c
      sample_add.c
      sample_memcopy.c
      sample_sort.c
      sample_strlen.c
      sample_svc.c
    stdlib
      initsct.c
      lowsrc.c
      otherlib.c
    sysdwn
      sysdwn.c
    timer
      tmrdrv.c
    Download modules
    Dependencies
```

Figure 17.33 Sources Registered in prj_cpuid2 Project

Note the following.

(1) Config file (sample.cfg)

The cfg file which is stored in the cpuid2\cfg_out\ directory.

(2) OS System File (kernel_def.c and kernel_cfg.c)

These files which are stored in the cpuid2\cfg_out\ directory are used to include the cfg72mp output files. The user is not permitted to change these files.

(3) import symbols (prj_cpuid1.fsy)

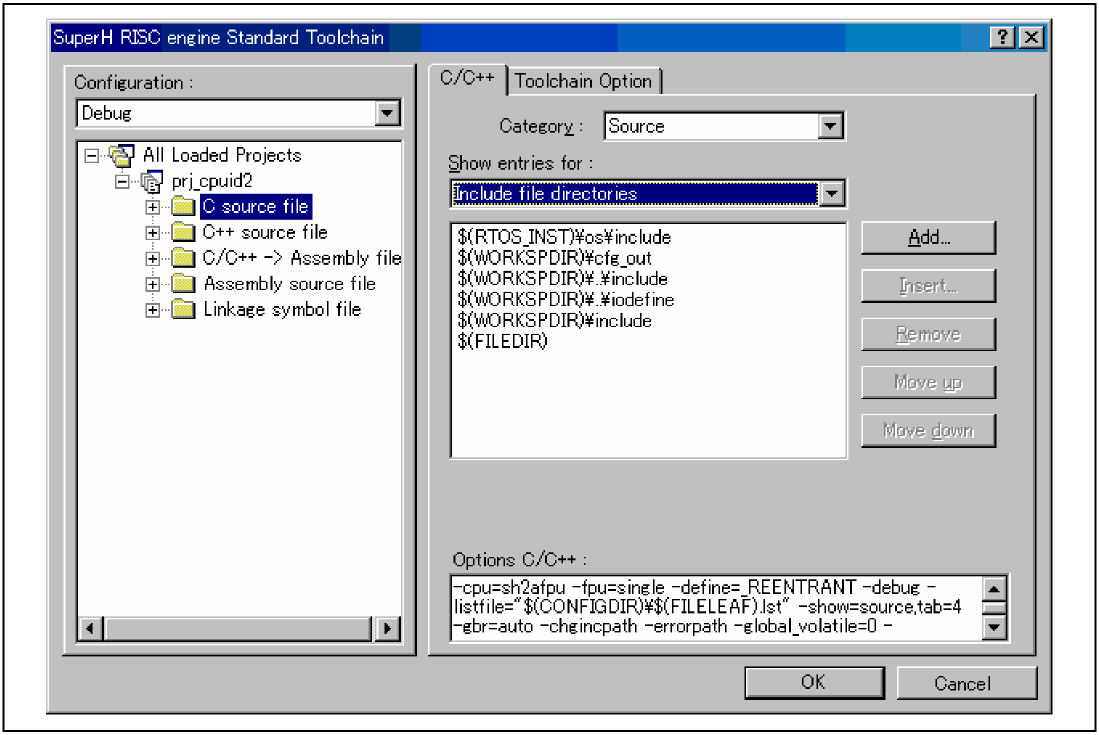
This is the symbol address file exported by CPUID#1, and it is stored in the cpuid1\prj_cpuid1\debug\ directory which is the High-performance Embedded Workshop configuration directory for CPUID#1. Note that when the High-performance Embedded Workshop configuration name of CPUID#1 is changed, this file must be registered again manually.

This file is generated by linkage of CPUID#1. Therefore when linkage of CPUID#1 has been executed, this file needs to be reassembled.

17.9.2 Compiler Options

(1) Include Directory

Figure 17.34 shows the common settings for all sources.

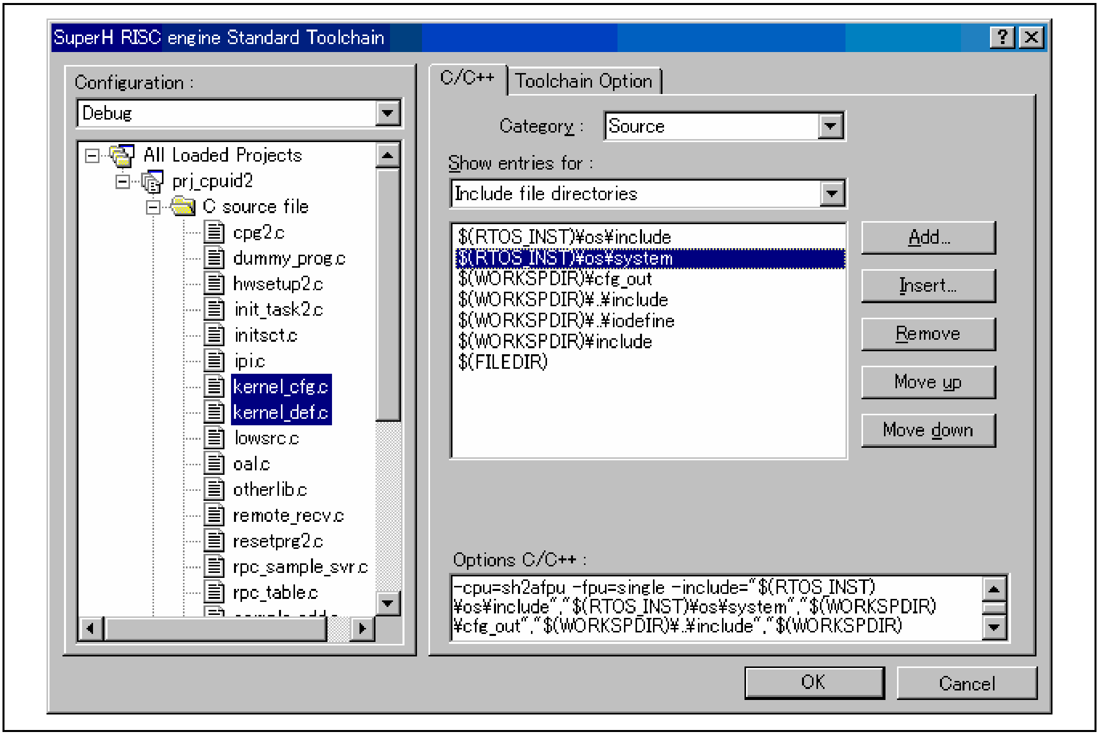


```
$(RTOS_INST)¥os¥include
$(WORKSPDIR)¥cfg_out
$(WORKSPDIR)¥.¥include
$(WORKSPDIR)¥.¥iodefine
$(WORKSPDIR)¥include
$(FILEDIR)
```

```
-cpu=sh2afpu -fpu=single -define=_REENTRANT -debug -
listfile="$(CONFIGDIR)¥$(FILELEAF).lst" -show=source,tab=4
-gbr=auto -chgincpath -errorpath -global_volatile=0 -
```

Figure 17.34 Include File Directories of Compiler

"$(RTOS_INST)\os\system" is added for kernel_cfg.c and kernel_def.c.



Configuration : Debug

- All Loaded Projects
  - prj_cpuid2
    - C source file
      - cpg2.c
      - dummy_prog.c
      - hwsetup2.c
      - init_task2.c
      - initsct.c
      - ipi.c
      - kernel_cfg.c
      - kernel_def.c
      - lowsrc.c
      - oal.c
      - otherlib.c
      - remote_recv.c
      - resetprg2.c
      - rpc_sample_svr.c
      - rpc_table.c

Category : Source

Show entries for : Include file directories

```
$(RTOS_INST)¥os¥include
$(RTOS_INST)¥os¥system
$(WORKSPDIR)¥cfg_out
$(WORKSPDIR)¥.¥include
$(WORKSPDIR)¥.¥iodefine
$(WORKSPDIR)¥include
$(FILEDIR)
```

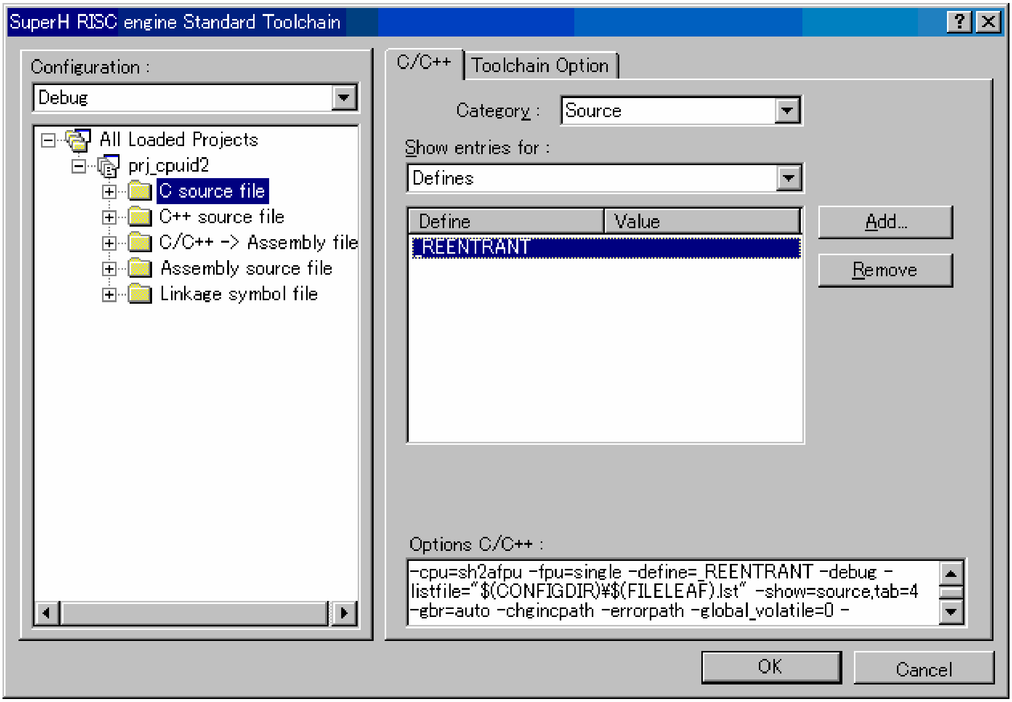
Options C/C++ :

```
-cpu=sh2afpu -fpu=single -include="$(RTOS_INST)
¥os¥include","$(RTOS_INST)¥os¥system","$(WORKSPDIR)
¥cfg_out","$(WORKSPDIR)¥.¥include","$(WORKSPDIR)
```

Figure 17.35 Include File Directories of Compiler (kernel_cfg.c, kernel_def.c)

(2) Macro Definitions

In this sample, the "_REENTRANT" macro is defined because the standard library is used as a reentrant library as shown in figure 17.36.



Category : Source

Show entries for : Defines

Define	Value
_REENTRANT	

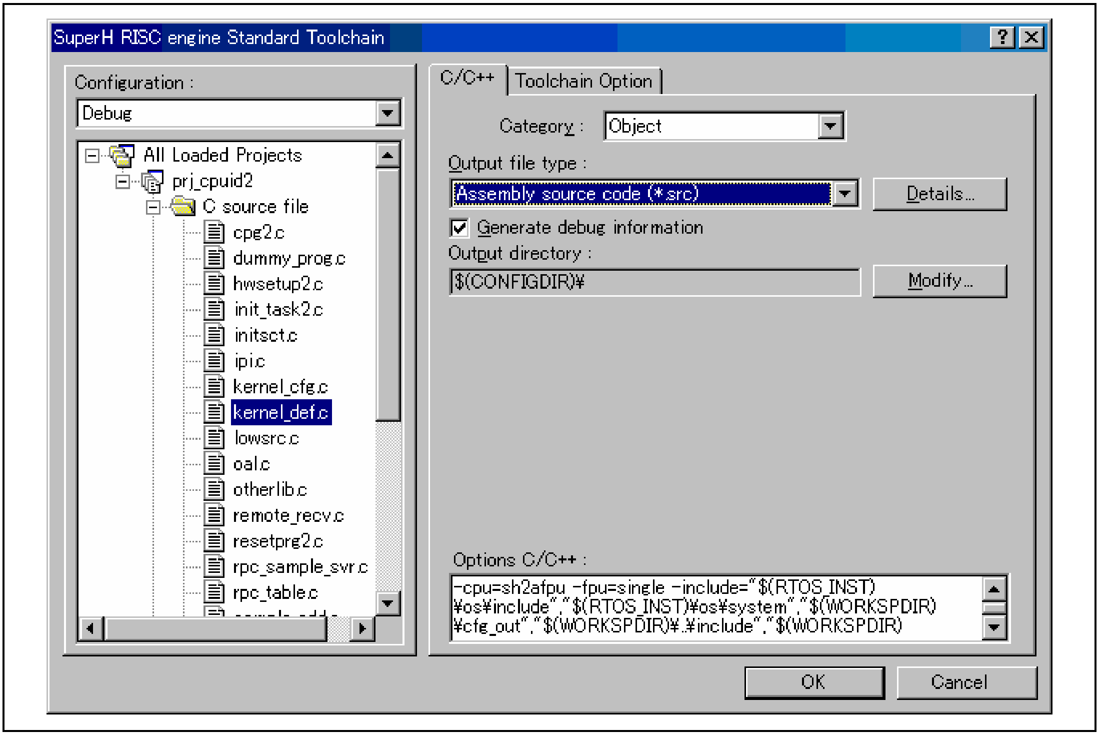
Options C/C++ :

```
-cpu=sh2afpu -fpu=single -define=_REENTRANT -debug -
listfile="$(CONFIGDIR)¥$(FILELEAF).lst" -show=source,tab=4
-gbr=auto -chgincpath -errorpath -global_volatile=0 -
```

Figure 17.36 Macro Definitions in Compiler

(3) Output File Type

The output file type is "Assembly source code" for kernel_def.c only because it uses inline assemble as shown in figure 17.37.



SuperH RISC engine Standard Toolchain

Configuration : Debug

- All Loaded Projects
  - prj_cpuid2
    - C source file
      - cpg2.c
      - dummy_prog.c
      - hwsetup2.c
      - init_task2.c
      - initsct.c
      - ipi.c
      - kernel_cfg.c
      - kernel_def.c
      - lowsrc.c
      - oal.c
      - otherlib.c
      - remote_recv.c
      - resetprg2.c
      - rpc_sample_svr.c
      - rpc_table.c

Category : Object

Output file type : Assembly source code (*.src)

☑ Generate debug information

Output directory : $(CONFIGDIR)¥

Options C/C++ :

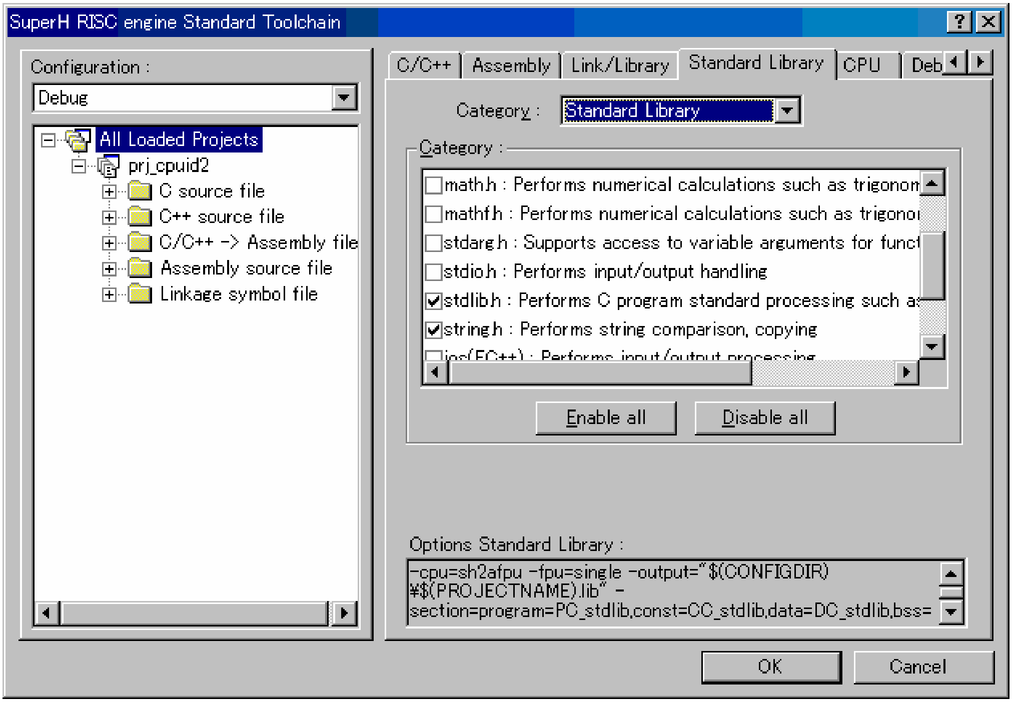
```
-cpu=sh2afpu -fpu=single -include="$(RTOS_INST)
¥os¥include","$(RTOS_INST)¥os¥system","$(WORKSPDIR)
¥cfg_out","$(WORKSPDIR)¥.¥include","$(WORKSPDIR)
```

Figure 17.37 Output File Type in Compiler

17.9.3 Standard Library Generator

(1) Embedded Standard Library Functions

As described in section 16.7, Standard Libraries, only stdlib.h and string.h are selected in this sample as shown in figure 17.38.



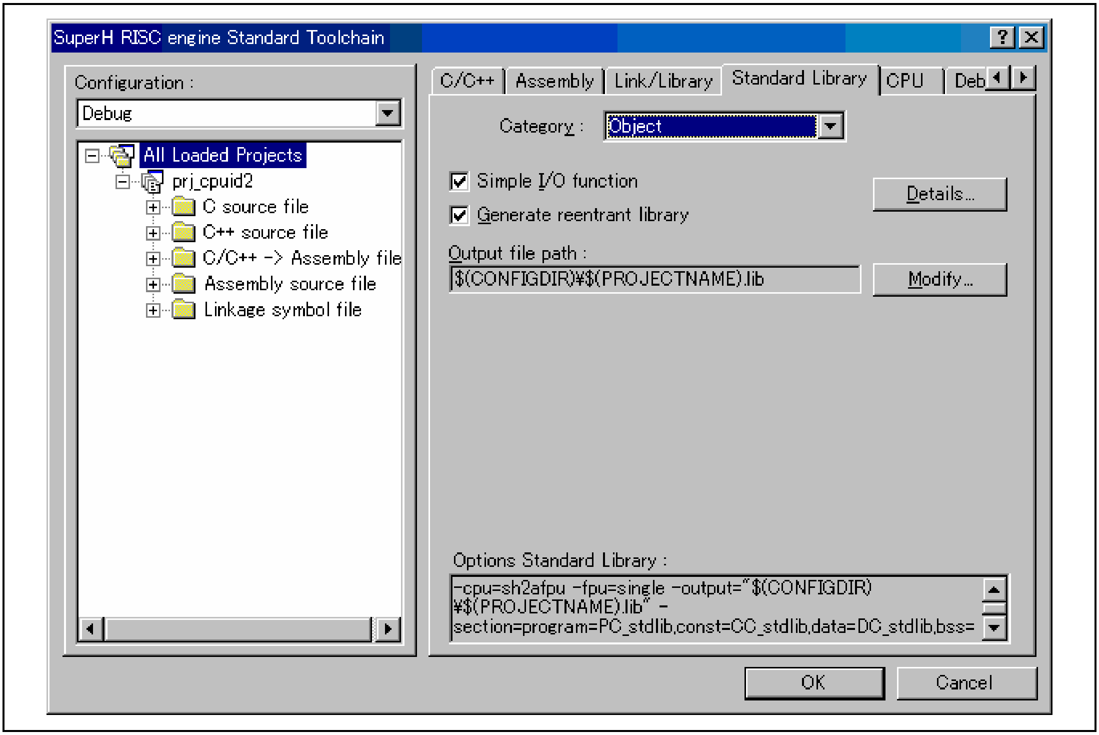
□ math.h : Performs numerical calculations such as trigonom
□ mathf.h : Performs numerical calculations such as trigonoı
□ stdarg.h : Supports access to variable arguments for funct
□ stdio.h : Performs input/output handling
☑ stdlib.h : Performs C program standard processing such as
☑ string.h : Performs string comparison, copying
□ ios(EC++) : Performs input/output processing

```
-cpu=sh2afpu -fpu=single -output="$(CONFIGDIR)
¥$(PROJECTNAME).lib" -
section=program=PC_stdlib,const=CC_stdlib,data=DC_stdlib,bss=
```

Figure 17.38 Library Function Selection in Standard Library Generator

(2) Object

An object is generated as a reentrant library as shown in figure 17.39.



Configuration : Debug

Category : Object

☑ Simple I/O function

☑ Generate reentrant library

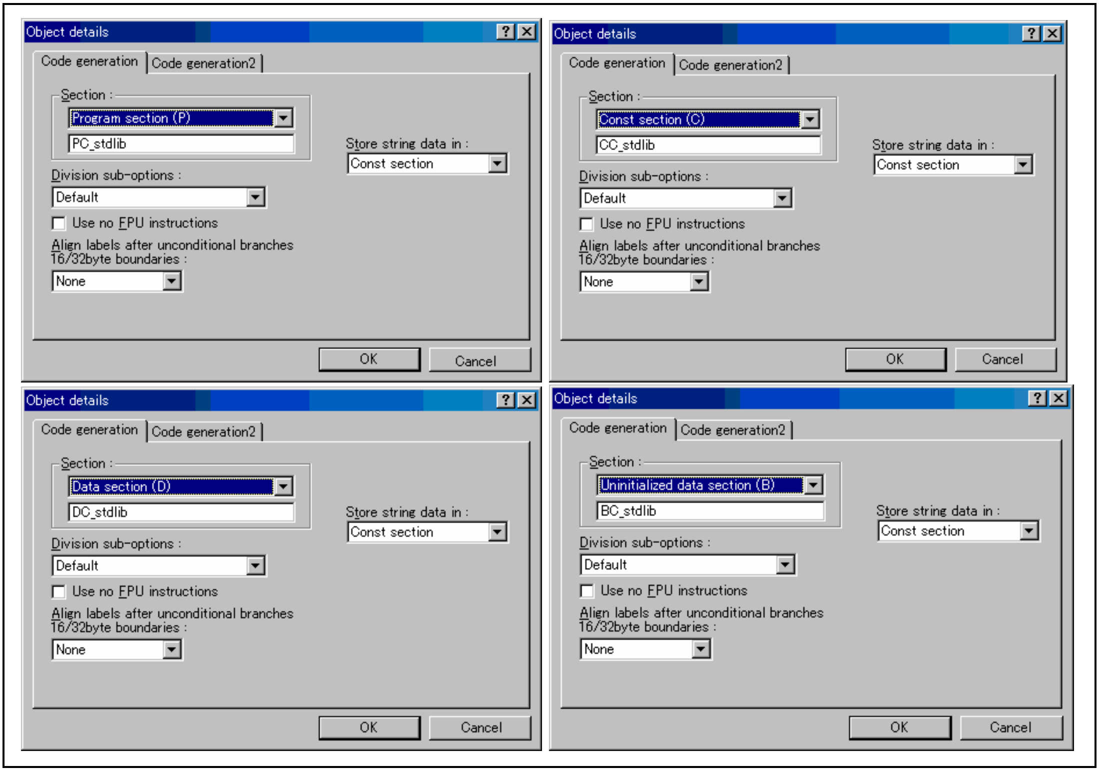
Output file path : $(CONFIGDIR)¥$(PROJECTNAME).lib

Options Standard Library :

```
-cpu=sh2afpu -fpu=single -output="$(CONFIGDIR)
¥$(PROJECTNAME).lib" -
section=program=PC_stdlib,const=CC_stdlib,data=DC_stdlib,bss=
```

Figure 17.39 Reentrant Library in Standard Library Generator

In the [Object details] dialog box opened by clicking the [Details...] button, the section names are set as shown in figure 17.40. The same section names are also used in lowsrc.c and otherlib.c.



Section : Program section (P)
PC_stdlib
Store string data in : Const section
Division sub-options : Default
□ Use no FPU instructions
Align labels after unconditional branches 16/32byte boundaries : None

Section : Const section (C)
CC_stdlib
Store string data in : Const section
Division sub-options : Default
□ Use no FPU instructions
Align labels after unconditional branches 16/32byte boundaries : None

Section : Data section (D)
DC_stdlib
Store string data in : Const section
Division sub-options : Default
□ Use no FPU instructions
Align labels after unconditional branches 16/32byte boundaries : None

Section : Uninitialized data section (B)
BC_stdlib
Store string data in : Const section
Division sub-options : Default
□ Use no FPU instructions
Align labels after unconditional branches 16/32byte boundaries : None

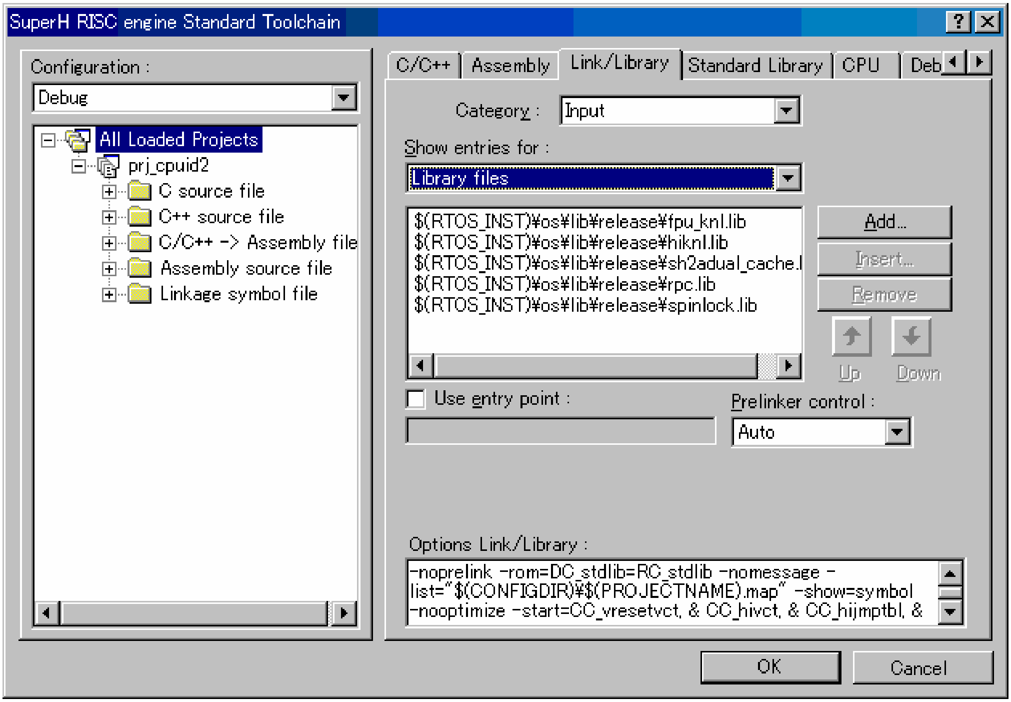
Figure 17.40 Section Name Setting in Standard Library Generator

17.9.4 Optimizing Linkage Editor

(1) Library Input

The following libraries provided by the HI7200/MP are input as shown in figure 17.41. For the kernel library input, refer to section 17.6, Kernel Library.

- fpu_knl.lib (kernel)
- hiknl.lib (kernel)
- sh2adual_cache.lib (SH2A-DUAL cache support library)
- rpc.lib (RPC library)
- spinlock.lib (spinlock library)



Configuration : Debug

Category : Input

Show entries for : Library files

```
$(RTOS_INST)¥os¥lib¥release¥fpu_knl.lib
$(RTOS_INST)¥os¥lib¥release¥hiknl.lib
$(RTOS_INST)¥os¥lib¥release¥sh2adual_cache.l
$(RTOS_INST)¥os¥lib¥release¥rpc.lib
$(RTOS_INST)¥os¥lib¥release¥spinlock.lib
```

☐ Use entry point :

Prelinker control : Auto

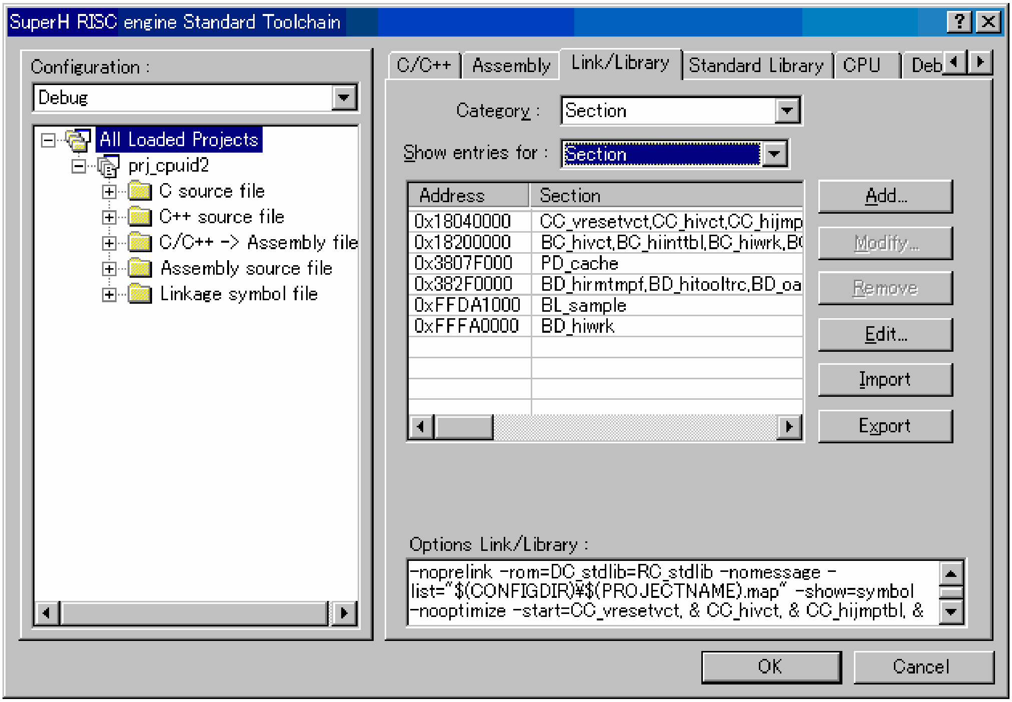
Options Link/Library :

```
-noprelink -rom=DC_stdlib=RC_stdlib -nomessage -
list="$(CONFIGDIR)¥$(PROJECTNAME).map" -show=symbol
-nooptimize -start=CC_vresetvct, & CC_hivct, & CC_hijmptbl, &
```

Figure 17.41 Library Input in Optimizing Linkage Editor

(2) Section Allocation

Though not all sections can be confirmed in figure 17.42, sections are allocated as described in section 17.5.5, Memory Map of this Sample. The point to notice in particular is that the "CC_vresetvct" section for the virtual reset vector table is allocated at the address (0x18040000) at which the "_ResetVectorTable_CPUID2" symbol was defined in section 17.8.4 (2) Symbol Definition for Virtual Reset Vector Table of CPUID#2.



Configuration : Debug

Category : Section

Show entries for : Section

Address	Section
0x18040000	CC_vresetvct,CC_hivct,CC_hijmp
0x18200000	BC_hivct,BC_hiinttbl,BC_hiwrk,B(
0x3807F000	PD_cache
0x382F0000	BD_hirmtmpf,BD_hitooltrc,BD_oa
0xFFDA1000	BL_sample
0xFFFA0000	BD_hiwrk

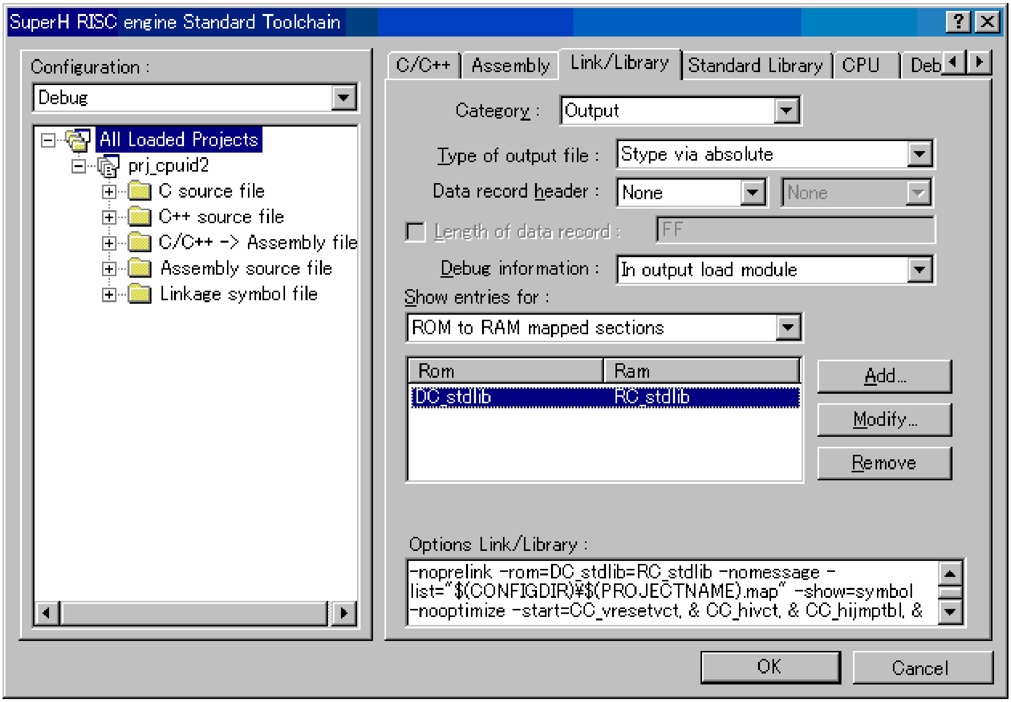
Options Link/Library :

```
-noprelink -rom=DC_stdlib=RC_stdlib -nomessage -
list="$(CONFIGDIR)¥$(PROJECTNAME).map" -show=symbol
-nooptimize -start=CC_vresetvct, & CC_hivct, & CC_hijmptbl, &
```

Figure 17.42 Section Allocation in Optimizing Linkage Editor

(3) ROM to RAM Mapping

Sections for which ROM to RAM mapping has to be performed, such as an initialized data section, are set in this sample as shown in figure 17.43.



Configuration : Debug

Category : Output

Type of output file : Stype via absolute

Data record header : None None

□ Length of data record : FF

Debug information : In output load module

Show entries for : ROM to RAM mapped sections

Rom	Ram
DC_stdlib	RC_stdlib

Options Link/Library :

```
-noprelink -rom=DC_stdlib=RC_stdlib -nomessage -
list="$(CONFIGDIR)¥$(PROJECTNAME).map" -show=symbol
-nooptimize -start=CC_vresetvct, & CC_hivct, & CC_hijmptbl, &
```

Figure 17.43 ROM to RAM Mapping in Optimizing Linkage Editor

(4) Notes

1. L1100 warning

   The L1100 warning (shown below) meaning that the specified section could not be found may be output sometimes at linkage.

   ```
   L1100 (W) Cannot find "PC_cache" specified in option "start"
   ```

   If the section that could not be found is a section listed in section 17.5.2, Sections, this is not a problem because it does not exist in some cases depending on configuration.

2. L1320 warning

   When more than one kernel library is specified, the L1320 warning (shown below) may be output for a number of times at linkage. This is because the kernel adopts an implementation method in which the same symbols and different programs are stored in more than one library file. There is no problem with the generated load module.

   ```
   L1320 (W) Duplicate symbol "__kernel_act_tsk" in "C:\…fpu_knl.lib(fpu_acttsk)"
   ```

17.10 Download to Target System

This section briefly describes the procedure for downloading the generated sample load modules to SDRAM (R0K572650D000BR) and executing them. The procedure is basically the same even when using a target system created by the user. Refer to the E10A-USB manual for details on downloading load modules to flash memory.

1. Open the CPUID#1 workspace.
2. Open the CPUID#2 workspace.
3. Connect the target via the High-performance Embedded Workshop of CPUID#1.
4. Connect the target via the High-performance Embedded Workshop of CPUID#2.
5. Input a reset command from each High-performance Embedded Workshop.
6. Initialize SDRAM, etc. using the command line in the High-performance Embedded Workshop of CPUID#1. (This enables download to SDRAM.)
7. Download load modules to SDRAM from each High-performance Embedded Workshop.
8. Since the reset vector table (_ResetVectorTable) generated in CPUID#1 has been downloaded to SDRAM, initialize PC and R15 via each High-performance Embedded Workshop based on the reset vector table. The actual initial values are shown below. This process is unnecessary when the load modules are downloaded to flash memory.
    - PC: Contents at the _ResetVectorTable address of CPUID#1 (= _Reser_Poweron of reset.src of CPUID#1)
    - R15: Contents at the (_ResetVectorTable + 4) address of CPUID#1 (= last address of on-chip RAM0)

This product provides the following High-performance Embedded Workshop batch files for simplifying steps 6 and 8.

(1) cpuid1\prj_cpuid1\hwsetup.hdc

Performs an initialization process equivalent to HardwareSetup_CPUID1(). SDRAM is initialized.

(2) cpuid1\prj_cpuid1\reset_cpu1.hdc

Performs the initialization process in step 8 for CPUID#1.

(3) cpuid2\prj_cpuid2\reset_cpu2.hdc

Performs the initialization process in step 8 for CPUID#2. PC and R15 are initialized based on the memory contents of addresses 0x180000000 and 0x18000004, respectively. Address 0x18000000 is the address where _ResetVectorTable (CC_resetvct section) in CPUID#1 is allocated. When the address for where to allocate the CC_resetvct section is changed in CPUID#1, modify the address in the batch file to the new address.

Section 18 Calculation of Stack Size

18.1 Stack Types

If a stack overflows, the system will operate incorrectly. Therefore, the user must determine the stack size required for each task or handler execution and allocate enough area for each task or handler by referring to the following description.

There are the following types of stacks.

- Task stack
- Interrupt stack (normal interrupt handler)
- Direct interrupt handler stack
- Timer stack
- Kernel stack

Stack Used before Kernel Initiation: The stacks used by programs executed before kernel initiation, such as immediately after a reset, are not managed by the kernel. Therefore, the user can use the desired area for the stack.

For a microcomputer having on-chip RAM, usually allocate the stack used at a reset to on-chip RAM. For a microcomputer without on-chip RAM, the stack used at a reset (mounted external RAM) may sometimes not be accessed depending on the bus state controller (BSC) status immediately after a reset. In this case, do not run programs that use stacks or do not generate any interrupts or exceptions until the RAM becomes accessible by changing the BSC settings. This is because register data is stored in the stack when interrupts or exceptions occur.

18.2 Basics of Stack Size Calculation

The procedure for calculating the stack size necessary for tasks and handlers are described in the subsequent sections. Here, the basic items for size calculation are given.

18.2.1 Size Consumed by Function Tree

Calculate the size consumed by the function tree that starts from the function that triggers execution of an application program. Such kind of a function is the entry function of a task or handler.

In figure 18.1, the size consumed by the function tree is calculated as 16 + 20 + 32 = 68 bytes.

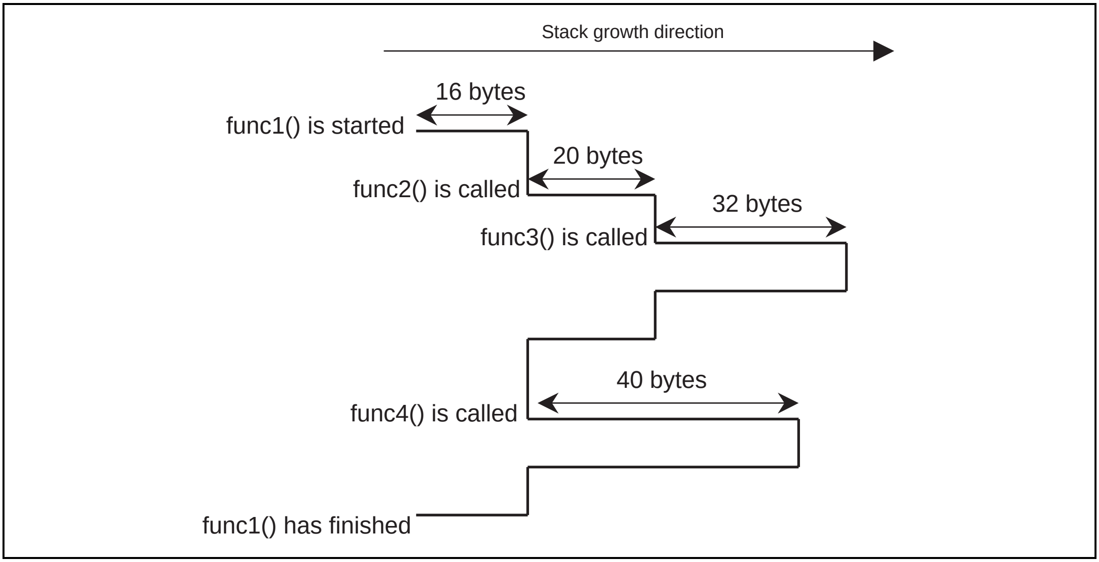


Figure 18.1 Size Consumed by Function Tree

The stack size used by each function can be referenced in "frame size" in the compile list file.

18.2.2 Kernel Service Calls

When calculating the stack size, handle a kernel service call as a function call. Refer to the release notes for the actual size.

18.2.3 RPC Library Call

When calculating the stack size, handle an RPC library call as a function call. Refer to the release notes for the actual size.

18.2.4 OAL, IPI, SH2A-DUAL Cache Support Library, and Spinlock Library

When calculating the stack size, handle these similarly to user-created functions.

18.2.5 Extended Service Calls

When calculating the stack size, handle an extended service call as a function call in which the extended service call routine is the called function. However, 8 bytes must be added for each extended service call.

18.2.6 Normal CPU Exception Handler and Direct CPU Exception Handler

These CPU exception handlers take over the stack used when the exception has occurred. In other words, these CPU exception handlers can be considered as a kind of function tree from the sense of stack consumption. Therefore, add the stack size used by a CPU exception handler in the same way as handling a function call.

The following values must also be added for each CPU exception.

- Normal CPU exception handler: 44 bytes
- Direct CPU exception handler: 8 bytes

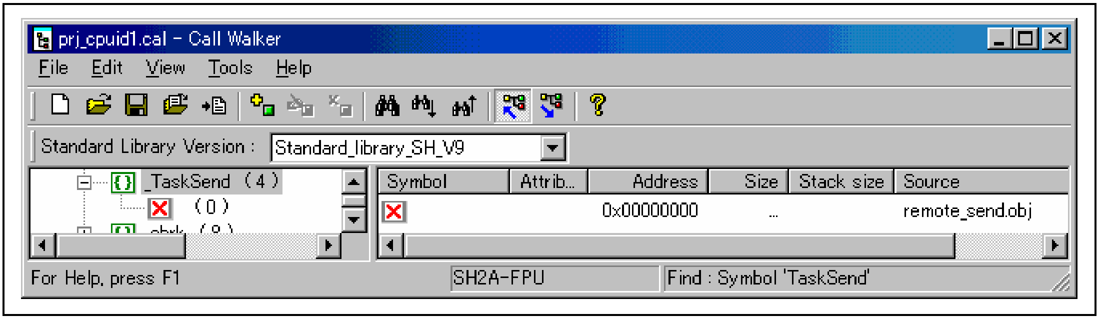
18.3 Usage Notes for Call Walker

The Call Walker provided in the compiler package is a utility for analyzing the relationship between function calls performed by symbol reference and displaying the stack amount used by that function tree. The notes for using the Call Walker are explained below.

(1) Kernel Service Calls

Due to its implementation method, the Call Walker cannot identify which function is called and when it is called in a case where a call is made using the function table. Since kernel service calls use the function table, they cannot be identified.

An example of the TaskSend() function making calls using the function table for two or more times is shown in figure 18.2. In the Call Walker, calls made via the function table are displayed as a single "×" icon, regardless of how many times they have been made. The size used in this case will be treated as 0 bytes.



Standard Library Version : Standard_library_SH_V9

_TaskSend (4)
☒ (0)

Symbol	Attrib...	Address	Size	Stack size	Source
☒		0x00000000	...		remote_send.obj

For Help, press F1 | SH2A-FPU | Find : Symbol 'TaskSend'

Figure 18.2 Call Walker Display Example for a Function Making Calls via the Function Table

(2) API Functions of RPC Library and OAL

Because these API functions internally issue kernel service calls, the size (stack amount used) in the Call Walker window will be smaller than the actual stack amount used.

(3) [Realtime OS Option]

The Call Walker has a feature to import the database file for the size used by each realtime OS and display the used stack size with the size defined by that file added to it, at each point a service call is issued.

This feature cannot be used in the HI7200/MP because the Call Walker cannot identify at which point a service call is issued and which service call type it is in the first place due to the reason shown in (1) Kernel Service Calls.

(4) Effective Use of Call Walker

Because of the reason shown in (1) Kernel Service Calls, the stack size used by a service call can be accurately reflected only by inspecting the service call issued by each function and adding the stack size manually. This however is quite complicated work.

This work can be made extremely easy by adding the maximum stack size used by a service call without exception to the maximum tree size calculated by the Call Walker. With this method, however, there is a high possibility that a size greater than the actually required size will be obtained.

18.4 Usage Notes for NMI

The method for calculating the size in this section is on the assumption that the NMI is not used. When the NMI is used, the user has to consider its effect.

18.5 Notes on Changes in Stack Size

The necessary stack size varies according to the causes below.

- Version of the compiler in use
- Compiler options, e.g. optimization
- Version of the HI7200/MP in use

In a case where the stack is no longer large enough because the above items were changed, the stack size to be allocated or the stack size specified in the OS has to be increased. It is recommended therefore to allocate or specify the stack size with a certain margin for avoiding such trouble and preventing overflows due to incorrect size calculation.

18.6 Task Stack

Basically, a different stack is used by each task ID. The kernel switches the task stacks at task dispatching.

In addition to this section, refer to section 4.5.4, Task Stack.

18.6.1 Calculation of Stack Size

The necessary size can be calculated by the formula below.

Necessary size =
Size consumed by function tree starting from entry function of task .... (a-1)
+ Context size of task .... (a-2)
+ Size consumed by function tree starting from entry function of
task exception handling routine .... (b-1)
+ Context size of task exception handling routine .... (b-2)
+ Addition considering nested interrupts .... (c)

(b-1) and (b-2) are 0 bytes when the task exception handling routine is not used.

(a-1) and (b-1) are sizes calculated according to section 18.2, Basics of Stack Size Calculation.

For the context sizes of (a-2) and (b-2), refer to table 18.1.

The task exception handling routine is normally not nested. However, the kernel specifications permit nesting of the task exception handling routine. When the task exception handling routine is nested, calculate (b-1) with programming nesting added and calculate (b-2) with the nest count multiplied.

Table 18.1 Task Context Size

TA_COP1 Attribute	Task	Task Exception Handling Routine
Not specified	84 bytes	88 bytes
Specified	84 + 72 (for FPU) = 156 bytes	88 + 72 (for FPU) = 160 bytes

The value of (c) (addition considering nested interrupts) differs depending on the cfg file setting.

The following symbols are used here:

UPPINTNST: Interrupt nest count with a level higher than the kernel interrupt mask level (system.system_IPL)

LOWINTNST: Interrupt nest count with a level equal to or lower than the kernel interrupt mask level

(1) system.vector_type is ROM_ONLY_DIRECT or RAM_ONLY_DIRECT

$$\text{Addition considering nested interrupts} = 8 \times \text{UPPINTNST} + 16 \times \text{LOWINTNST}$$

(2) system.vector_type is ROM or RAM

(a) system.regbank is ALL

$$\text{Addition considering nested interrupts} = 8 \times \text{UPPINTNST} + 16 \times \text{LOWINTNST} + \alpha$$

In a case where LOWINTNST includes an interrupt that is defined in kernel_intspec.h to not use register banks, if that interrupt is used as a normal interrupt, 36 bytes are added as α.

(b) system.regbank is other than ALL

$$\text{Addition considering nested interrupts} = 8 \times \text{UPPINTNST} + \underline{24 \times \text{LOWINTNST} + 20}$$

When LOWINTNST is 0, the underlined portion is treated as 0.

18.6.2 Specification Location for Stack Size

(1) Task using non-static stack

(a) Generating a task by making a cre_tsk, icre_tsk, acre_tsk, or iacre_tsk service call
Specify the size of the stack in stksz of the T_CTSK structure.

(b) Generating a task by the cfg file
Specify the size of the stack in task[].stack_size.

(2) Task using static stack
Specify the size of the stack in static_stack[].stack_size in the cfg file. Also specify the task ID that uses that stack in static_stack[].tskid.

18.6.3 Calculation of Default Task Stack Area Size (memstk.all_memsize)

Specify the size of the default task stack area in memstk.all_memsize in the cfg file. The value to be specified can be calculated as follows:

Default task stack area size =
Σ ((Stack size specified when generating a task that uses the default task stack)
+ 0x10) + 0x1C

Note that the unique section name of BC_hitskstk is given to the default task stack area.

18.6.4 Stack Size Used by SVC Server Task (remote_svc.stack_size)

Specify as the stack size used by SVC server tasks the value obtained by applying the following values into the formula in section 18.6.1, Calculation of Stack Size.

(a-1): max([A], [B])
[A]: Size used by local service call corresponding to requested remote service call
[B]: Size used by IPI_send()
The value of [A] which depends on implementation is listed in the release notes.
(a-2): 84 bytes (no TA_COP1 attribute specification)
(b-1), (b-2): 0 bytes (task exception handling routine is not used)

18.6.5 RPC Server Task and Server Stub

A server stub is called from an RPC server task in the RPC library.

The necessary stack size for RPC server tasks can be calculated from the formula below.

Necessary size =
Size consumed by RPC server task function in RPC library .... (a)
+ Necessary size obtained by treating server stub as task entry function .... (b)

(a) The genuine size consumed by the function tree in the RPC library. This size is specified as rpc_config.ServerTaskStackSize in rpc_init(). For the value to be specified here, refer to the release notes.

(b) The size calculated according to section 18.6.1, Calculation of Stack Size, with the server stub treated as the task entry function. Specify the value of (b) as rpc_server_info.ulStubStackSize which is specified in rpc_start_server() and rpc_start_server_with_paramarea().

Note that the size to be actually allocated as the stack for the RPC server tasks is (a) + (b).

The stack for the RPC server tasks is assigned from the default task stack area by OAL_CreateTask(). Take this into consideration when performing the calculation described in section 18.6.3, Calculation of Default Task Stack Area Size (memstk.all_memsize).

18.7 Normal Interrupt Handler Stack (system.stack_size)

The interrupt stack is used by normal interrupt handlers, and there is only one interrupt stack in each CPU. When a normal interrupt occurs, the kernel switches the stack to the interrupt stack. However, when a normal interrupt is nested, the kernel does not switch the stack.

A service call issued from the normal interrupt handler and a function called back from that service call take over the same stack and use it.

Each CPU has only a single interrupt stack. The interrupt stack area is allocated by specifying its size in system.stack_size in the cfg file. The section name of the interrupt stack area is BC_hiirqstk.

18.7.1 Calculation of Stack Size Used by Each Handler

Calculate the necessary size using the formula below.

```
Necessary size =
     Size consumed by function tree starting from entry function of handler ........ (a)
```

(a) is a size calculated according to section 18.2, Basics of Stack Size Calculation.

18.7.2 Calculation of and Specification Location for Interrupt Stack Area Size (system.stack_size)

Specify the size of the interrupt stack area in system.stack_size in the cfg file. The value to be specified can be calculated as shown below.

The following symbols are used here:

UPPINTNST: Interrupt nest count with a level higher than the kernel interrupt mask level (system.system_IPL)

LOWINTNST: Interrupt nest count with a level equal to or lower than the kernel interrupt mask level

(1) system.vector_type is ROM_ONLY_DIRECT or RAM_ONLY_DIRECT

Since there are no normal interrupt handlers in this case, the interrupt stack area is not allocated regardless of the system.stack_size setting.

(2) system.vector_type is ROM or RAM

(a) system.regbank is ALL

$$\begin{aligned}&\text{Interrupt stack area size} =\\&\quad \Sigma\ (\text{Size used by handler using largest stack size at each interrupt level}) + 4\\&\quad + 8 \times \text{UPPINTNST} + \underline{16 \times (\text{LOWINTNST} - 1)} + \alpha\end{aligned}$$

When LOWINTNST is 0 or 1, the underlined portion is treated as 0.

In a case where LOWINTNST includes an interrupt that is defined in kernel_intspec.h to not use register banks, if that interrupt is used as a normal interrupt, 36 bytes are added as α.

(b) system.regbank is other than ALL

$$\begin{aligned}&\text{Interrupt stack area size} =\\&\quad \Sigma\ (\text{Size used by handler using largest stack size at each interrupt level}) + 4\\&\quad + 8 \times \text{UPPINTNST} + \underline{24 \times (\text{LOWINTNST} - 1) + 20}\end{aligned}$$

When LOWINTNST is 0 or 1, the underlined portion is treated as 0.

18.8 Direct Interrupt Handler Stack

For direct interrupt handlers, a separate stack is allocated for each handler by the application. When a handler is activated, the stack must be switched to the stack of that handler, and the stack must be returned to the original stack when the handler is finished.

Note that interrupt handlers with interrupt levels higher than the kernel interrupt mask level (system.system_IPL) must be defined as direct interrupt handlers.

18.8.1 Calculation of Stack Size

The necessary size for each handler can be calculated by the formula below.

Necessary size =
Size consumed by function tree starting from entry function of handler .... (a)
+ Addition considering nested interrupts .... (b)

(a) is a size calculated according to section 18.2, Basics of Stack Size Calculation.

The value of (b) (addition considering nested interrupts) differs depending on the cfg file setting.

The following symbols are used here:

UPPINTNST:	Interrupt nest count with a level higher than the kernel interrupt mask level (system.system_IPL) and any interrupt level in the current CPU
LOWINTNST:	Interrupt nest count with a level equal to or lower than the kernel interrupt mask level and also higher than the interrupt levels in the current CPU

(1) system.vector_type is ROM_ONLY_DIRECT or RAM_ONLY_DIRECT, or there is no normal interrupt with a level higher than that of the relevant direct interrupt

Addition considering nested interrupts = 8 × UPPINTNST + 16 × LOWINTNST

(2) Other than (1)

The "interrupt stack area size" that is calculated according to section 18.7, Normal Interrupt Handler Stack (system.stack_size), with the target interrupt limited to a normal interrupt with a level higher than the interrupt levels in the current CPU, is used as the addition considering nested interrupts. Note that this calculation result is simply the addition considering nested interrupts, and it is not a value to be set in system.stack_size.

18.8.2 Specification Location for Stack Size

The stack area of direct interrupt handlers should be allocated by the user. For details, refer to section 12.5.4, Direct Interrupt Handlers.

18.8.3 Shared Stack Function

Direct interrupt handlers of the same interrupt level can share the stack since such interrupt handlers do not use the stack simultaneously.

18.9 Timer Stack (clock.stack_size)

The timer stack is used by interrupt handlers for timer interrupts in the kernel. The following programs use the timer stack because they are called from interrupt handlers for timer interrupts in the kernel.

- Time event handler
- tdr_int_tmr()

tdr_stp_tmr() called back from vstp_tmr and tdr_rst_tmr() called back from vrst_tmr or ivrst_tmr also use the timer stack.

Each CPU has only a single timer stack. The timer stack area is allocated by specifying its size in clock.stack_size in the cfg file. The section name of the timer stack area is BC_hitmrstk.

Calculate the necessary size using the formula below.

```
Necessary size = max([A], [B], [C], [D], [E], [F], [G]) +
      Addition considering nested interrupts

[A]:  TMR_A + (Size used by tdr_int_tmr())
[B]:  TMR_B + (Maximum size used by time event handler)
[C]:  TMR_C + (Maximum size used by time event handler) (Only when vrst_tmr or
      ivrst_tmr is used)
[D]:  TMR_D
[E]:  TMR_E + (Size used by tdr_stp_tmr()) (Only when vstp_tmr is used)
[F]:  TMR_F + (Size used by tdr_rst_tmr()) (Only when vrst_tmr or ivrst_tmr is used)
[G]:  TMR_G
```

The values of TMR_A and others which depend on implementation are listed in the release notes.

When system.action is YES, the "debug demon" for object manipulation functions is automatically generated as a cyclic handler, and so its size must be considered when determining the "maximum size used by time event handler". The size used by the debug demon which also depends on implementation is listed in the release notes.

The addition considering nested interrupts differs depending on the cfg file setting.

The following symbols are used here:

UPPINTNST: Interrupt nest count with a level higher than the kernel interrupt mask level (system.system_IPL)

LOWINTNST: Interrupt nest count with a level equal to or lower than the kernel interrupt mask level and also higher than that of the timer interrupt

(1) system.vector_type is ROM_ONLY_DIRECT or RAM_ONLY_DIRECT, or there is no normal interrupt with a level higher than that of the timer interrupt

Addition considering nested interrupts = 8 × UPPINTNST + 16 × LOWINTNST

(2) Other than (1)

The "interrupt stack area size" that is calculated according to section 18.7, Normal Interrupt Handler Stack (system.stack_size), with the target interrupt limited to a normal interrupt with a level higher than the interrupt levels in the current CPU, is used as the addition considering nested interrupts. Note that this calculation result is simply the addition considering nested interrupts, and it is not a value to be set in system.stack_size.

Specify the timer stack size obtained as described above in clock.stack_size in the cfg file.

18.10 Kernel Stack (system.kernel_stack_size)

The kernel stack is used by service calls issued in task contexts and also by the initialization routine.

Each CPU has only a single kernel stack. The kernel stack area is allocated by specifying its size in system.kernel_stack_size in the cfg file. The section name of the kernel stack area is BC_hiknlstk.

Calculate the necessary size using the formula below.

Necessary size = max([A], [B], [C], [D], [E])

[A]: KNL_A + Addition 1 considering nested interrupts (Only when system.trace!=NO)
[B]: KNL_B + Addition 1 considering nested interrupts
[C]: KNL_C + Addition 1 considering nested interrupts (Only when system.action==YES)
[D]: KNL_D + (Maximum size used by initialization routine) + Addition 1 considering nested interrupts
[E]: KNL_E + Addition 2 considering nested interrupts

The values of KNL_A and others which depend on implementation are listed in the release notes.

Note that tdr_ini_tmr() of the timer driver must be considered when determining the "maximum size used by initialization routine".

"addition 1 considering nested interrupts" and "addition 2 considering nested interrupts" differ depending on the cfg file setting.

The following symbols are used here:

UPPINTNST:	Interrupt nest count with a level higher than the kernel interrupt mask level (system.system_IPL)
LOWINTNST:	Interrupt nest count with a level equal to or lower than the kernel interrupt mask level

(1) system.vector_type is ROM_ONLY_DIRECT or RAM_ONLY_DIRECT

Addition 1 considering nested interrupts = 8 × UPPINTNST
Addition 2 considering nested interrupts = 8 × UPPINTNST + 16 × LOWINTNST

(2) system.vector_type is ROM or RAM

(a) system.regbank is ALL

Addition 1 considering nested interrupts = 8 × UPPINTNST
Addition 2 considering nested interrupts = 8 × UPPINTNST + 16 × LOWINTNST+ α

In a case where LOWINTNST includes an interrupt that is defined in kernel_intspec.h to not use register banks, if that interrupt is used as a normal interrupt, 36 bytes are added as α.

(b) system.regbank is other than ALL

Addition 1 considering nested interrupts = 8 × UPPINTNST
Addition 2 considering nested interrupts = 8 × UPPINTNST + $\underline{24 \times \text{LOWINTNST} + 20}$

When LOWINTNST is 0, the underlined portion is treated as 0.

Specify the kernel stack size obtained as described above in system.kernel_stack_size in the cfg file.

18.11 Size Used by Features Provided by HI7200/MP

The stack size used by the functions provided as libraries depend on the product version. Refer to the release notes attached to the product.

18.11.1 Kernel

The release notes contain the following information.

- Stack size used by each service call
- Constant values related to the timer stack (TMR_A, etc.) and the size used by the debug demon
- Constant values related to the kernel stack (KNL_A, etc.)
- Size to be specified as remote_svc.stack_size (stack size used by SVC server tasks)
- Stack size used by callback functions registered in the IPI

  An IPI port is created using IPI_create() in a vini_rmt service call. When calculating the stack size used by inter-processor interrupt handlers of the IPI, this size should be added.

18.11.2 RPC Library

(1) Stack Size Used by Server Stubs (rpc_server_info.ulStubStackSize)

In rpc_start_server() and rpc_start_server_with_paramarea(), specify the stack size used by the server stubs in rpc_server_info.ulStubStackSize.

The value to be specified here is the value calculated according to section 18.6.1, Calculation of Stack Size, with the server stub or the callback function specified in rpc_stop_server() treated as the task entry function.

(2) Stack Size Used by Callback Function Specified in rpc_disconnect()

This callback function will not be executed in the current implementation.

(3) Information in Release Notes

The following information is contained in the release notes.

- Stack size used by each API function
- Stack size to be specified as rpc_config.ServerTaskStackSize in rpc_init()
- Stack size used by callback functions registered in the IPI

The RPC library creates an IPI port using IPI_create(). When calculating the stack size used by inter-processor interrupt handlers of the IPI, this size should be added.

18.11.3 API Functions of OAL

Calculate the stack size using a method similar to that used for the application functions.

18.11.4 IPI

(1) Stack Size Used by API Functions

Calculate the stack size using a method similar to that used for the application functions.

(2) Stack Size Used by Callback Function Specified in IPI_create()

This callback function is called from an inter-processor interrupt handler of the IPI. Take this into consideration when calculating the stack size used by inter-processor interrupt handlers.

18.11.5 API Functions of Spinlock Library

Calculate the stack size using a method similar to that used for the application functions.

Note that all spinlock library functions are written in assembly language and are defined to not use the stack at all in the current implementation.

18.11.6 API Functions of Cache Support Library

Calculate the stack size using a method similar to that used for the application functions.

Note that some internal functions are written in assembly language and those functions are defined to not use the stack at all in the current implementation.

Section 19 types.h

In the kernel and RPC library, derived data types are defined based on the data types defined in types.h.

types.h is stored in the <RTOS_INST>\os\include\ directory.

Table 19.1 shows the data types defined in types.h.

Table 19.1 Data Types Defined in types.h

Data Type	Meaning
VOID	void
INT	signed int
INT8	signed char
INT16	signed short
INT32	signed long
INT64	signed long long
UINT	unsigned int
UINT8	unsigned char
UINT16	unsigned short
UINT32	unsigned long
UINT64	unsigned long long

Section 20 Notes on the FPU

This section gives notes on the FPU incorporated in the SH2A-FPU. In particular, be sure to read section 20.1.3, Options fpu and fpscr, whether or not you will actually need the floating-point operations.

20.1 Compiler Options

20.1.1 Consistency of Options

As stated in the compiler user’s manual, the following compiler options associated with the FPU must be consistent within each linkage unit.

- fpu
- fpscr
- round

20.1.2 cpu Option

Always specify “cpu = sh2afpu”.

20.1.3 Options fpu and fpscr

In general, specify “single” or “double” for the fpu option. If floating-point operations are not to be used, specify fpu=single.

Omitting the fpu option is possible but not recommended. If you wish to omit the fpu option, specify fpscr=safe. Otherwise, correct operation cannot be guaranteed.

20.2 Floating-Point Operations in Tasks and Task Exception Handling Routines

20.2.1 TA_COP1 Attribute

If tasks or task exception handling routines are to execute floating-point operations, specify the TA_COP1 attribute.

20.2.2 Initialization of FPSCR

In this kernel, the value of FPSCR on the initiation of a task or a task exception handling routine is defined as H'00040001 (SZ = 0, PR = 0, DN = 1, RM = B'01). This initial value is the default value for the relevant compiler options.

When a task or task exception handling routine is to execute floating-point operations and one of the following compiler options has been specified (i.e. the setting is not the default), FPSCR must be initialized at the start of the entry function. Specifically, FPSCR must be initialized to the value assumed by the compiler at the start of each function as stated in section 20.5, Handling by the Compiler (Reference).

- fpu=double
- round=nearest

Figure 20.1 shows an example of initializing FPSCR in a task under the following conditions:

- cpu=sh2afpu
- fpu=double
- round=nearest

```
#include <machine.h>            /* Included to make intrinsic
                                function set_fpscr() available. */
#define INI_FPSCR 0x000C0000    /* Initial FPSCR value
                                (SZ=0, PR=1, DN=1, RM=B'00) */
#pragma noregsave(Task)
void Task(VP_INT exinf)
{
    set_fpscr(INI_FPSCR);       /* FPSCR is set at the start of
                                the task. */
    /* Task processing */
    ext_tsk();
}
```

Figure 20.1 Example of Initialization of FPSCR in a Task

20.3 Floating-Point Operations in Handlers

The following passages are notes on the execution of floating-point operations in the following handlers:

- Normal interrupt handlers
- Direct interrupt handlers
- Normal CPU exception handlers
- Direct CPU exception handlers
- Time event handlers
- Initialization routines
- Timer drivers (tdr_ini_tmr(), tdr_int_tmr(), tdr_stp_tmr(), and tdr_rst_tmr())

20.3.1 Overview

(1) Guarantee Restoration of FPU Registers

When floating-point operation is required, these handlers need to explicitly guarantee restoration of values to all FPU registers.

(2) Initialize the FPSCR

The initial value of the FPSCR for a normal/direct CPU exception or interrupt handler is the same as the value before the CPU exception or interrupt. The initial value of the FPSCR for the other handlers, on the other hand, is undefined.

When any of these handlers is to perform floating-point operations, the FPSCR must be initialized as shown in 20.5, Handling by the Compiler (Reference) at the start of the entry function of these handlers.

20.3.2 Coding

The HI7200/MP provides the macros listed below to ease handling of the FPU registers. These macros are defined in <RTOS_INST>\os\include\sh2fapu.h. The "code=asmcode" option must be specified at the time of compilation because these macros use #pragma inline_asm.

(1) void IniFPU (VT_FPU *pk_save, UW ini_fpscr)

This macro should be used at the start of the handler. It saves the contents of FPU registers including the FPSCR in the area pointed to by pk_save, and initializes FPSCR to ini_fpscr.

(2) void EndFPU(VT_FPU *pk_save)

This macro should be used at the end of the handler. It restores the contents of FPU registers including the FPSCR from the area pointed to by pk_save.

Figure 20.2 shows an example of a handler that initializes the FPSCR and guarantees restoration of the contents of FPU registers.

```
#include “sh2afpu.h”            /* Include "sh2afpu.h" */
#define INI_FPSCR 0x00040001    /* Initial FPSCR value
                                (SZ=0, PR=0, DN=1, RM=B'01) */
void HandlerMain(void)          /* Handler main routine */
{
    /* Handler processing */
}

void Handler(void)              /* Handler entry function */
{
    T_FPU area;                 /* For saving FPU registers */
    IniFPU (&area, INI_FPSCR);  /* Save FPU registers and
                                initialize FPSCR */
    HandlerMain();              /* Call HandlerMain(), which performs
                                main processing*/
    EndFPU (&area);             /* Restore FPU registers */
}
```

Figure 20.2 Example of a Handler that Initializes FPSCR and Preserves the Contents of FPU Registers

20.4 Floating-Point Operations in Extended Service-Call Routines

The compiler handles issuing of extended service calls as function calls to which floating-point data is not passed.

20.4.1 When Called from Task Contexts

The TA_COP1 attribute must be specified for the calling task or task-exception handling routine.

20.4.2 When Called from Non-Task Contexts

The contents of all FPU registers must be guaranteed by callers such as interrupt handlers. Refer to section 20.3, Floating-Point Operations in Handlers.

20.5 Handling by the Compiler (Reference)

This section explains handling by the compiler. The compiler only generates object code to change the FPSCR if the fpu option has been omitted.

(1) FPSCR.PR (Precision Mode)

Table 20.1 Handling of the FPSCR.PR Bit by the Compiler

Compiler Option		Precision Mode Assumed by the Compiler (FPSCR.PR Bit)*[1]	Precision Mode at the End of the Function*[2]	Remarks
FPU Option	FPSCR Option *[3]			
Single	(Not specifiable)	Single precision (0)	Single precision (0)	The compiler does not generate any object code to change the PR bit.
Double	(Not specifiable)	Double precision (1)	Double precision (1)	
Omitted (Mix)	safe	Single precision (0)	Single precision (0)	
	aggressive	Single precision (0)	Undefined	Not specifiable for this kernel

Notes: 1. The compiler assumes this precision mode in generating code from the start of each function.

2. The compiler generates code to select this precision mode at the end of each function.

(2) FPSCR.RM (Rounding Mode)

Table 20.2 Handling of the FPSCR.RM Bits by the Compiler

Compiler Option	Rounding Mode Assumed by the Compiler (FPSCR.DN Bit)*	Remarks
Round Option		
Zero	Round to Zero (B'01)	The compiler does not generate any object code to change the RM bits.
Nearest	Round to Nearest (B'00)	

Note: The compiler assumes this rounding mode in generating code at the top of the function.

(3) FPSCR.SZ (Transfer Size Mode)

The compiler always assumes SZ = 0 (the unit of data for the FMOV instruction is 32 bits) and does not generate any object code to change the SZ bit.

Renesas Microcomputer Development Environment System
User's Manual
HI7200/MP V.1.00

Publication Date: Rev.1.01, September 27, 2007
Published by: Sales Strategic Planning Div.
Renesas Technology Corp.
Edited by: Customer Support Department
Global Strategic Communication Div.
Renesas Solutions Corp.

Renesas Technology Corp. Sales Strategic Planning Div. Nippon Bldg., 2-6-2, Ohte-machi, Chiyoda-ku, Tokyo 100-0004, Japan

RENESAS SALES OFFICES

http://www.renesas.com

Refer to "**http://www.renesas.com/en/network**" for the latest and detailed information.

Renesas Technology America, Inc.
450 Holger Way, San Jose, CA 95134-1368, U.S.A
Tel: <1> (408) 382-7500, Fax: <1> (408) 382-7501

Renesas Technology Europe Limited
Dukes Meadow, Millboard Road, Bourne End, Buckinghamshire, SL8 5FH, U.K.
Tel: <44> (1628) 585-100, Fax: <44> (1628) 585-900

Renesas Technology (Shanghai) Co., Ltd.
Unit 204, 205, AZIACenter, No.1233 Lujiazui Ring Rd, Pudong District, Shanghai, China 200120
Tel: <86> (21) 5877-1818, Fax: <86> (21) 6887-7898

Renesas Technology Hong Kong Ltd.
7th Floor, North Tower, World Finance Centre, Harbour City, 1 Canton Road, Tsimshatsui, Kowloon, Hong Kong
Tel: <852> 2265-6688, Fax: <852> 2730-6071

Renesas Technology Taiwan Co., Ltd.
10th Floor, No.99, Fushing North Road, Taipei, Taiwan
Tel: <886> (2) 2715-2888, Fax: <886> (2) 2713-2999

Renesas Technology Singapore Pte. Ltd.
1 Harbour Front Avenue, #06-10, Keppel Bay Tower, Singapore 098632
Tel: <65> 6213-0200, Fax: <65> 6278-8001

Renesas Technology Korea Co., Ltd.
Kukje Center Bldg. 18th Fl., 191, 2-ka, Hangang-ro, Yongsan-ku, Seoul 140-702, Korea
Tel: <82> (2) 796-3115, Fax: <82> (2) 796-2145

Renesas Technology Malaysia Sdn. Bhd
Unit 906, Block B, Menara Amcorp, Amcorp Trade Centre, No.18, Jalan Persiaran Barat, 46050 Petaling Jaya, Selangor Darul Ehsan, Malaysia
Tel: <603> 7955-9390, Fax: <603> 7955-9510

Colophon 6.0

HI7200/MP V.1.00
User's Manual

Renesas Electronics Corporation
1753, Shimonumabe, Nakahara-ku, Kawasaki-shi, Kanagawa 211-8668 Japan