NEC

User's Manual

RX4000 (μITRON 4.0)

Real-Time Operating System

Basics

Target devices

VR4100 Series™
VR5000 Series™

Target real-time OS

RX4000V4 Ver. 4.10

Document No. U14833EJ2V0UM00 (2nd edition)
Date Published February 2003 N CP(K)

[MEMO]

VR Series, VR4100 Series, VR5000 Series, VR4100, VR4121, VR4122, VR4131, VR4181, VR4181A, VR5000, VR5000A, VR5432, VR5500, and VR10000 are trademarks of NEC Electronics Corporation.

Green Hills Software is a trademark of Green Hills Software, Inc.

TRON is the abbreviation for The Real-time Operating system Nucleus.

ITRON is the abbreviation for Industrial TRON.

μITRON is the abbreviation for Micro Industrial TRON.

TRON, ITRON, and μITRON are not the names of specific products or a product group.

The μITRON4.0 specification is an open real-time kernel specification that was mainly established by the ITRON sectional meeting of the TRON Association.

The μITRON4.0 specification is available from the ITRON project home page (http://www.itron.gr.jp/).

- **The information in this document is current as of February, 2002. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC Electronics sales representative for availability and additional information.**
- No part of this document may be copied or reproduced in any form or by any means without the prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.
- NEC 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 NEC Electronics products listed in this document or any other liability arising from the use of such products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.
- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of a customer's equipment shall be done under the full responsibility of the customer. NEC Electronics assumes no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.
- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.
- NEC Electronics products are classified into the following three quality grades: "Standard", "Special" and "Specific".

  The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated "quality assurance program" for a specific application. The recommended applications of an NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.

  "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.

  "Special": 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, life support systems and medical equipment for life support, etc.

The quality grade of NEC Electronics products is "Standard" unless otherwise expressly specified in NEC Electronics data sheets or data books, etc. If customers wish to use NEC Electronics products in applications not intended by NEC Electronics, they must contact an NEC Electronics sales representative in advance to determine NEC Electronics' willingness to support a given application.

(Note)

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

(2) "NEC Electronics products" means any product developed or manufactured by or for NEC Electronics (as defined above).

M8E 02. 11-1

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC Electronics product in your application, please contact the NEC Electronics office in your country to obtain a list of authorized representatives and distributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and other legal issues may also vary from country to country.

NEC Electronics America, Inc. (U.S.)
Santa Clara, California
Tel: 408-588-6000
800-366-9782
Fax: 408-588-6130
800-729-9288

NEC Electronics (Europe) GmbH
Duesseldorf, Germany
Tel: 0211-65 03 01
Fax: 0211-65 03 327

- **Sucursal en España**
  Madrid, Spain
  Tel: 091-504 27 87
  Fax: 091-504 28 60

- **Succursale Française**
  Vélizy-Villacoublay, France
  Tel: 01-30-67 58 00
  Fax: 01-30-67 58 99

- **Filiale Italiana**
  Milano, Italy
  Tel: 02-66 75 41
  Fax: 02-66 75 42 99

- **Branch The Netherlands**
  Eindhoven, The Netherlands
  Tel: 040-244 58 45
  Fax: 040-244 45 80

- **Tyskland Filial**
  Taeby, Sweden
  Tel: 08-63 80 820
  Fax: 08-63 80 388

- **United Kingdom Branch**
  Milton Keynes, UK
  Tel: 01908-691-133
  Fax: 01908-670-290

NEC Electronics Hong Kong Ltd.
Hong Kong
Tel: 2886-9318
Fax: 2886-9022/9044

NEC Electronics Hong Kong Ltd.
Seoul Branch
Seoul, Korea
Tel: 02-528-0303
Fax: 02-528-4411

NEC Electronics Shanghai, Ltd.
Shanghai, P.R. China
Tel: 021-6841-1138
Fax: 021-6841-1137

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-2719-2377
Fax: 02-2719-5951

NEC Electronics Singapore Pte. Ltd.
Novena Square, Singapore
Tel: 6253-8311
Fax: 6250-3583

J02.11

PREFACE

Target Readers

This manual is aimed at users engaged in the design or development of VR4100 Series or VR5000 Series application systems.

Purpose

The purpose of this manual is to give users an understanding of the functions of the RX4000 shown under **Organization** below.

Organization

This manual is broadly divided into the following sections.

- Overview
- Kernel
- Scheduler
- Task management
- Synchronous communication management
- Memory management
- Time management
- Interrupt management
- System configuration management
- Service call management
- Initialization management
- Interface library
- Service calls

How to Read This Manual

Readers need to have general knowledge of electricity, logic circuits, microcontrollers, and C and assembly languages.

To learn about the hardware or instruction functions of the VR4100 Series or VR5000 Series, refer to the user's manual for the relevant product.

Conventions

Note:	Footnote for item marked with **Note** in the text
Caution:	Information requiring particular attention
Remark:	Supplementary information
Numerical representation:	Binary ... XXXX or B'XXXX
	Decimal ... XXXX
	Hexadecimal ... 0xXXXX or H'XXXX

Prefixes indicating powers of 2 (address space and memory capacity):

K (kilo): $2^{10} = 1,024$

M (mega): $2^{20} = 1,024^2$

Related Documents

The related documents indicated in this publication may include preliminary versions. However, preliminary versions are not marked as such.

Documents related to VR4100 Series

Document Name	Document No.
VR4100 Series Architecture User's Manual	U15509E
VR4121™ User's Manual	U13569E
μPD30121 (VR4121) Data Sheet	U14691E
VR4122™ User's Manual	U14327E
μPD30122 (VR4122) Data Sheet	U15585E
VR4131™ Hardware User's Manual	U15350E
μPD30131 (VR4131) Data Sheet	U16219E
VR4181™ Hardware User's Manual	U14272E
μPD30181 (VR4181) Data Sheet	U14273E
VR4181A™ Hardware User's Manual	U16049E
μPD30181A, 30181AY (VR4181A) Data Sheet	U16277E

Documents related to VR5000 Series

Document Name	Document No.
VR5000™, VR5000A™ User's Manual	U11761E
μPD30500, 30500A Data Sheet	U12031E
VR5432™ User's Manual	U13751E
μPD30541 Data Sheet	U13504E
VR5500™ User's Manual	U16044E
μPD30550 Data Sheet	U15700E
VR5000, VR10000™ Instruction User's Manual	U12754E

Documents related to development tools (user's manuals)

Document Name		Document No.
RX4000 (μITRON4.0) (Real-time OS)	Basics	This manual
	Installation	U14834E
	Technical	U14835E
AZ4000 Ver. 4.0 System Performance Analyzer		U15031E

CONTENTS

LIST OF FIGURES (1/2)

LIST OF FIGURES (2/2)

LIST OF TABLES

CHAPTER 1 OVERVIEW

The RX4000 (μITRON4.0) is an embedded real-time, multitask control OS that provides a highly efficient real-time, multitasking environment to increase the application range of processor control units. This product is a high-speed, compact OS capable of being stored in and run from the ROM of a target system.

1.1 Real-Time OS

Control equipment demands systems that can rapidly respond to events occurring both internal and external to the equipment. Conventional systems have utilized simple interrupt processing as a means of satisfying this demand. With the recent improvements in the performance and functionality of control equipment, however, it has proved difficult for systems to satisfy these requirements by means of simple interrupt processing alone.

In other words, the task of managing the order in which internal and external events are processed has become increasingly difficult as systems have increased in complexity and programs have become ever larger.

Real-time operating systems have been designed to overcome this problem.

The main goals of a real-time OS are to respond to internal and external events rapidly and execute programs in the optimum order.

1.2 Multitask OS

A “task” is the minimum unit in which a program can be executed by an OS. “Multitasking” is the name given to the mode of operation in which a single processor processes multiple tasks concurrently.

Actually, the processor can handle no more than one program (instruction) at a time. But, by switching the processor’s attention to individual tasks on a regular basis (at a certain timing), it appears that the tasks are being processed simultaneously.

A multitask OS therefore enables the parallel processing of tasks by switching the tasks to be executed as determined by the system.

A major goal of a multitask OS is to improve the processing capability of the overall system through the parallel processing of multiple tasks.

1.3 Features

The RX4000 has the following features.

1. Conformity with μITRON4.0 Specification

As a representative embedded type control OS architecture, the RX4000 is used to perform design that is compliant with the μITRON4.0 Specification. Thus, by constructing software in accordance with the μITRON4.0 Specification, users can improve the software's portability and capitalization.

Because the μITRON4.0 Specification enables selection of an incorporation range in extended areas outside the standard profile, the following functions have been realized in the RX4000.

- All functions specified in the standard profile
- Function to dynamically create and delete resources
- Extended synchronous communication function (mutex)
- Memory pool management function (fixed-length memory pool, variable-length memory pool)
- Interrupt processing function
- Service call management function (extended service call routine)
- System configuration management function
- State-referencing service calls (service calls prefixed with "ref_")

2. Customization

Because processing that supports interrupts and CPU exceptions must have an exceptionally fast response time, by supplying the source code of these functions as a sample, users are able to customize the RX4000 to create the most suitable OS for their peripheral hardware and other environments.

3. ROMization

The RX4000 is a real-time, multitask OS that has been designed on the assumption that it will be incorporated into the target system; it has been made as compact as possible to enable it to be loaded into a system's ROM.

4. Utilities supporting system construction

The RX4000 provides the following utilities to aid in system construction:

- CF4000 (System configurator)
- RD4000 (Task debugger)

5. Cross tools

The RX4000 supports the following cross tools for the VR Series™:

- CCMIPE (Green Hills Software™, Inc.)

1.4 Configuration

The RX4000 consists of four subsystems: a kernel, an interface library, a system configurator, and a task debugger. These subsystems are outlined below.

1. **Kernel**
   The center or nucleus of the RX4000 is called the kernel. The kernel is embedded in the target system along with the processing program (task/non-task), and provides real-time multitask processing such as task switching, as well as the various services requested via service calls.

2. **Interface library**
   When a processing program (task/non-task) is written in a high-level language such as C and a service call is issued, an external function format is used as the service call issuance format. The input format that can be understood by the kernel, however, differs from the external function format. The interface library is therefore used to translate a service call issued in an external function format into the kernel input format. The interface library thus acts as an agent between processing programs and the kernel.
   The interface library provided by the RX4000 supports the C cross compiler shown in the tools in **1.7 Development Environment**.

3. **System configurator**
   The system configurator is a tool that operates on the development machine and is used to output the tables (system information tables) in which the data required by the kernel during initialization (such as the number of tasks or semaphores, and the memory allocation) is stored.

4. **Task debugger**
   The task debugger is a tool that operates on the development machine and is used to improve debugging at the software development stage by displaying via a GUI information managed by the OS, such as the state of a task or the number of semaphores.

1.5 Applications

The RX4000 can be used in the following application fields.

- Control systems such as in automobiles and robots
- Measuring instruments
- Control devices such as exchangers, numerical controls, communication controls, and plant controls
- OA machines such as facsimiles and photocopiers
- Data collection and data calculation systems such as medical treatment devices and space monitoring devices

1.6 Execution Environment

The RX4000 is operated in target systems equipped with the following hardware.

1. Processor

VR4100 Series
VR5000 Series

2. Peripheral hardware

In order to enable support of a wide variety of hardware environments, the hardware-dependant section of the kernel in the RX4000 is supplied as a separate source file, which can be used to rewrite this section to accord with the system used, thereby eliminating the need for specific peripheral hardware.

1.7 Development Environment

The hardware and software environments required to develop an application system in which the RX4000 is embedded are shown below.

Table 1-1. Development Environment

Type	Name	Remarks
Host OS	Windows 98/NT 4.0 Windows 2000/Me/XP Solaris 2.5.x or later	
Cross tools	CCMIPE	V1.8.9 R400 (Green Hills Software)
Debuggers	PARTNER-ET II/Win Multi	V2.0 (KMC: Kyoto Microcomputer Corporation) V1.8.9 R4.0.2
Evaluation boards	CMB-VR4131 Ostrich VR4181A RTE-VR5500-CB PARTNER-ET II	VR4131 evaluation board [SolutionGear II] (NEC Eloctronics) VR4181A evaluation board (NEC Electronics) VR5500 evaluation board (Midas Lab Co., Ltd.) ROM emulator for board connection (KMC)

CHAPTER 2 KERNEL

This chapter describes the kernel, which is the core of the RX4000.

2.1 Overview

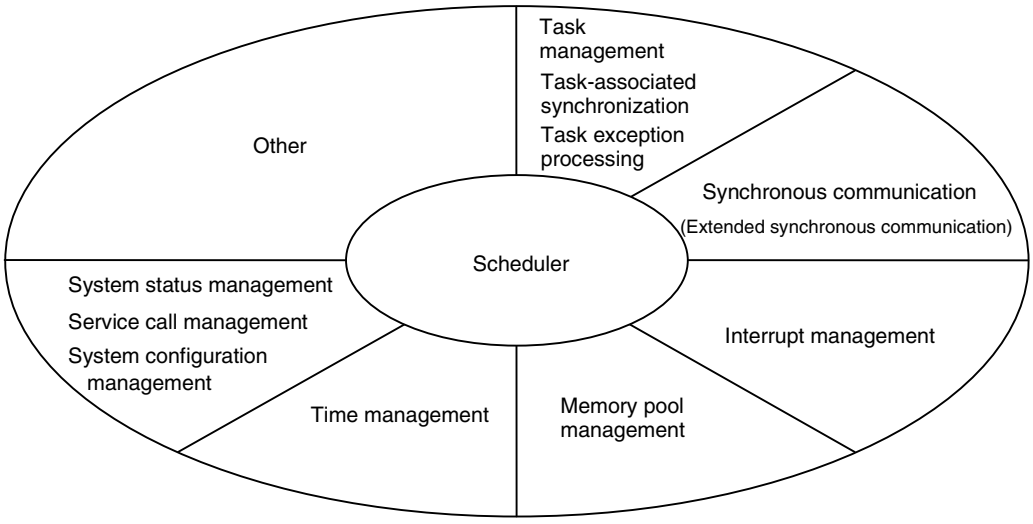
The kernel forms the heart of the RX4000 and is the main body of the OS embedded in the target system together with the application program (tasks, handlers, etc.). The kernel provides the following two major functions. Note that the module that controls the former is usually known as the scheduler.

- Switching execution to the task selected as the most suitable task to be executed next, according to an event that occurs internal or external to the target system
- Provision of functions corresponding to the service calls (system calls) issued from the tasks or handlers

2.2 Configuration

The kernel consists of a scheduler and management modules that exist in each function. These functions can be used by issuing the service calls provided by the RX4000. The configuration of the RX4000 kernel is shown below.

Figure 2-1. Kernel Configuration



2.3 Functions

This section overviews the functions of the scheduler and management modules.

Refer to **CHAPTER 3 SCHEDULER** to **CHAPTER 5 SYNCHRONOUS COMMUNICATION MANAGEMENT** for details of the individual management modules.

1. Scheduler

The scheduler manages and determines the order in which tasks are executed and assigns tasks the right to use the CPU. In the RX4000, the task execution order is determined according to an assigned priority and by applying the FCFS (First Come First Served) system. When activated, the scheduler determines the priorities assigned to the tasks, selects an optimum task from those ready to be executed (running or ready state) (by assessing which task has the highest priority and entered the ready state the earliest) and assigns that task the right to use the CPU.

2. Task management

This module manipulates and manages the states of a task, the minimum unit in which processing is performed by the RX4000. The module can, for example, create, activate, execute, stop, terminate, and delete a task.

3. Task-associated synchronization

This module is used to perform synchronization that is not reliant on objects other than tasks, such as semaphores. This module can, for example, put a task unconditionally into a self-imposed waiting state, interrupt execution of another task, or release another task from a waiting state. For further details of this module, refer to the explanation in **CHAPTER 4 TASK MANAGEMENT**.

4. Task exception management

This module is newly added for μITRON4.0 and is used to enable the writing of not only the routine for normal execution of the main body of tasks (task main processing routine), but also a task exception processing routine that is activated by the occurrence of an event exceptional to a task. A more detailed explanation of the task exception management function can be found in **CHAPTER 4 TASK MANAGEMENT**.

5. Synchronous communication management

This module provides the following five functions related to exclusive control, wait, and communication, which comprise the three types of synchronous communication between tasks. Note that in accordance with μITRON4.0, the correct categorization of a mutex is as an extended synchronous communication function.

Table 2-1. Synchronous Communication Function

Type	Provided Function
Exclusion	Semaphores, mutexes
Wait	Event flags
Communication	Data queues, mailboxes

6. Memory management

The memory area is managed by the RX4000 as two types of areas: memory pools and memory blocks. A memory pool consists of memory blocks grouped together in one large chunk. When memory is necessary to execute applications such as tasks or handlers, memory blocks can be acquired from a memory pool, and then returned to the memory pool when no longer required. Performing this kind of dynamic memory manipulation allows users to make efficient use of a limited memory area.

7. Time management

The RX4000 manages a software clock to enable specific operations at a fixed interval or at a specified time, providing functions such as system clock management, task timeout (wait time upper limit setting), and cyclic handlers.

8. System status management

System management functions provided in this module include enabling/disabling dispatch (task switching) and referencing the status of the system. Note that this category appears for the first time in μITRON4.0; the majority of its associated service calls are classified into other categories in μITRON3.0.

9. Interrupt management

The two kinds of interrupt management functions provided in this module are interrupt control functions for processing such as enabling/disabling interrupts, and interrupt processing management functions for activating interrupt service routines corresponding to interrupt sources. Note that because interrupt management is heavily dependent on the hardware operated by the application, a source file has been provided as a sample, allowing users to customize these functions.

10. Service call management

This module provides a function that can be used to register functions described by users as service calls.

11. System configuration management

This module provides a function to allow users to register handlers that are used to process exceptions detected by the CPU, and a function to define the initialization routines called when the system is initialized.

12. Initialization processing

This module is used for processing such as initializing the hardware required for the RX4000 to operate and initializing the kernel itself. Note that whereas the classifications of the functions in 1 to 11 above are specified by μITRON4.0, initialization processing is not subject to this specification.

2.4 Objects

The tasks (task main processing routine, task exception processing routine), semaphores, event flags, data queues, mailboxes, mutexes, fixed-length memory pools, variable-length memory pools, memory blocks, cyclic handlers, interrupt service routines, extended service call routines, and CPU exception handlers, which are created or defined by the kernel and are under its management, are known as objects or kernel objects. (Note that these are also sometimes called resources.)

2.4.1 Object control blocks

The RX4000 is equipped with data structures known as control blocks to manage the objects listed above. The control block saves information such as the status of the object, and is therefore the (logical) substance of the object to be processed by the kernel. Control blocks and objects exist on a one-to-one basis.

2.4.2 Object identification number

To uniquely identify the objects or control blocks in the RX4000, a number specific to each resource type is used. The way this number is called and the range of its values differs depending on the resource. These numbers are classified as follows.

1. **ID number**
   In the RX4000, each object (task, semaphore, event flag, data queue, mailbox, mutex, fixed-length memory pool, variable-length memory pool, cyclic handler, or interrupt service routine) is accessed using an ID number. The ID numbers are used as an index of the control block array when an object is accessed, and each object (task, semaphore, etc.) is assigned an ID number with a unique value of 1 or higher. The maximum value (or range of usable ID numbers) is that specified for each object in the system information table.

2. **Interrupt number**
   Interrupt service routines have an interrupt (source) number for specifying the interrupt source, which differs from the ID number described above. The interrupt (source) number is a number that uniquely identifies each interrupt source, and must be assigned by the user.

3. **Extended function code**
   Extended function codes are used to access extended function routines. Like an ID number, an extended function code consists of a unique value of 1 or higher and is used as an index of the control block array. The maximum value of the extended function code is that specified in the system information table.

4. **Handler number**
   When a CPU exception handler is defined, the CPU exception handler is assigned a unique value of 1 or higher known as its handler number, which like an ID number is used as an index of the control block array. The maximum value of this number is also specified in the system information table. Note that although the handler number accords with the ID number in the kernel, it is necessary not only to identify the CPU exception handler, but also to clarify its correspondence with the exception source.

5. **Address**
   The memory block is identified by its top address.

2.4.3 Creating and deleting objects

When an object is created by issuing a service call such as cre_tsk, the index block specified from the control block array in the system pool is secured and initialized. The control block array is fixed when the kernel is initialized, based on the maximum value of each object as defined in the system information table. Object creation therefore means the occupation of the relevant control block, and object deletion means the invalidation of the control block that was occupied, thereby putting it in a state whereby it can be occupied again when a new object is created. The size of the system pool is thus not affected by the creation or deletion of objects.

2.4.4 Automatic assignment of ID number

An object can be created without specifying an ID number by issuing a service call beginning with acre_, such as acre_tsk. When acre_xxx is issued, the kernel searches for an unoccupied control block in the non-existent state from the control block array for the object to be created.

If the search is successful, the ID number of that object is returned as the return value of the service call when creation processing is complete. An unsuccessful search results in the return of the error code E_NOID, which indicates that ID assignment failed.

Note that the ID numbers that can be used by objects created by automatic assignment can be specified from the range assignable to each resource type. Therefore, because a control block indexed with an ID number that falls outside this range will not be subject to the aforementioned search processing, if automatic assignment fails, it does not necessarily mean that the maximum number of creatable objects already exists.

CHAPTER 3 SCHEDULER

This chapter explains the scheduling performed by the RX4000.

3.1 Overview

The module that manages the sequence in which tasks are executed and performs task switching is known as the scheduler. The scheduler is activated each time a service call issued or an interrupt is generated. This chapter describes the functions of the scheduler and outlines the processing performed by it.

3.2 Drive Method

The RX4000 scheduler uses an event-driven technique in which the scheduler activates driving in response to the occurrence of some kind of event, such as the issuance of a service call that may cause a task state to change from a processing program such as a task or a handler, or the generation of an interrupt (or more precisely, restoration from an interrupt).

3.3 Scheduling Methods

When driven, the scheduler selects the task to be executed next according to predetermined methods, which in the RX4000 are priority order and FCFS (First Come First Served).

3.3.1 Priority method

Each task is assigned a priority, which determines the sequence in which it will be executed.

The scheduler checks the priority of each task that can be executed (in the running or ready state), selects the task with the highest priority, and assigns that task the right to use the CPU.

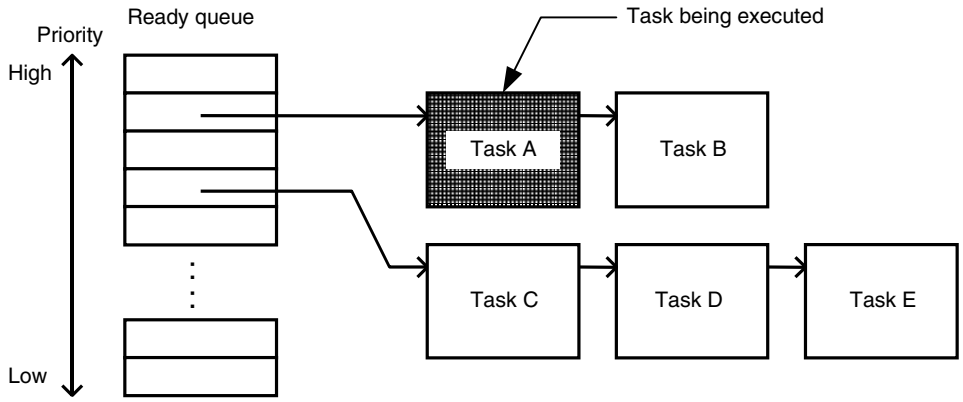
3.3.2 FCFS method

The RX4000 can assign the same priority to more than one task. Because the priority method is used for task scheduling, there is a possibility that more than one task with the same (highest) priority will be selected. In this case, the scheduler selects the task that first became executable from this group of tasks (i.e., the task that has been in an executable state for the longest time) and assigns it the right to use the CPU. This is known as the FCFS (First Come First Served) system.

3.4 Ready Queue

In the RX4000, tasks in the executable (ready) or execution (running) states are queued by the FIFO method in a hash table (ready queue) arranged according to priority order. The scheduler searches the ready queue activation from the top priority and assigns the task with the highest priority in the queue the right to use the CPU. In other words, the priority and FCFS methods are implemented in the running state.

Figure 3-1. Ready Queue



3.5 Implementing a Round Robin Method

In scheduling based on the priority and FCFS methods, even if a task has the same priority as that currently running, it cannot be executed unless the task to which the right to use the CPU was assigned first enters another state or relinquishes control of the processor.

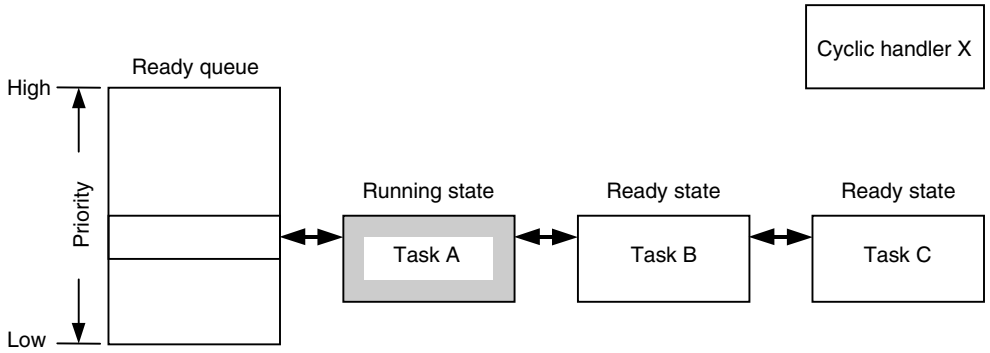
The scheduling method generally used to avoid the above drawback is known as a round robin method. Although the RX4000 does not include a round robin function itself, it does provide a service call, (i)rot_rdq, which can be issued periodically to enable this scheduling method. The round robin method is realized in the following way.

(Assumed conditions)

- Task priority
  Task A = Task B = Task C
- State of tasks
  Task A: Running state
  Task B: Ready state
  Task C: Ready state
- Cyclic handler X is periodically activated and issues the irot_rdq service call

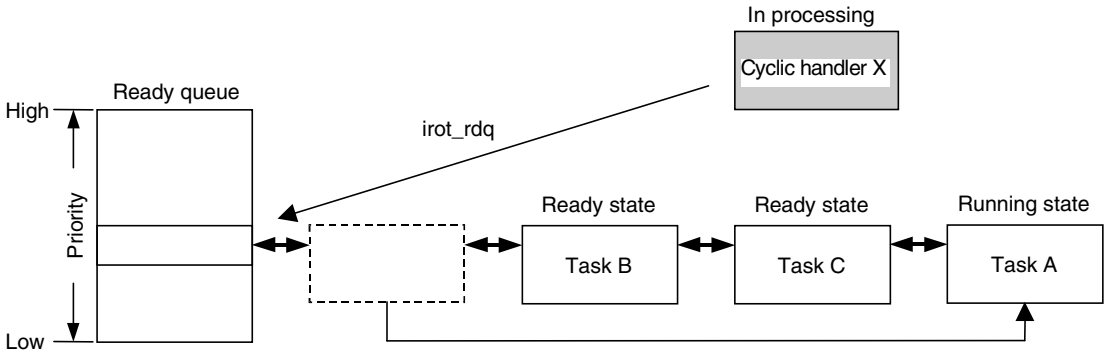
1. Task A is currently being executed. The other tasks (B and C) have the same priority as task A but cannot be executed because task A has control of the CPU (see **Figure 3-2**).

Figure 3-2. State 1 of Ready Queue During Round Robin Processing



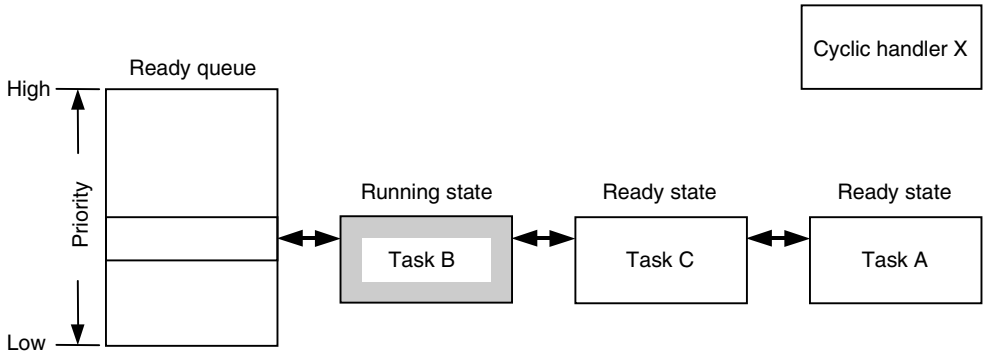
2. Cyclic startup handler X is activated when a predetermined period of time has passed, specifies the priority held by task A, and issues the service call irot_rdq. Task A then shifts to the bottom of the ready queue and task B moves to the top (see **Figure 3-3**).

Figure 3-3. State 2 of Ready Queue During Round Robin Processing



3. Cyclic startup handler X finishes processing and the scheduler is activated. Because task A was shifted to the bottom of the ready queue and is therefore in the ready state, the scheduler searches for the task to execute next. Because there are no tasks with a higher priority than tasks A, B, and C, task B, which is currently at the head of the queue, is executed (see **Figure 3-4**).

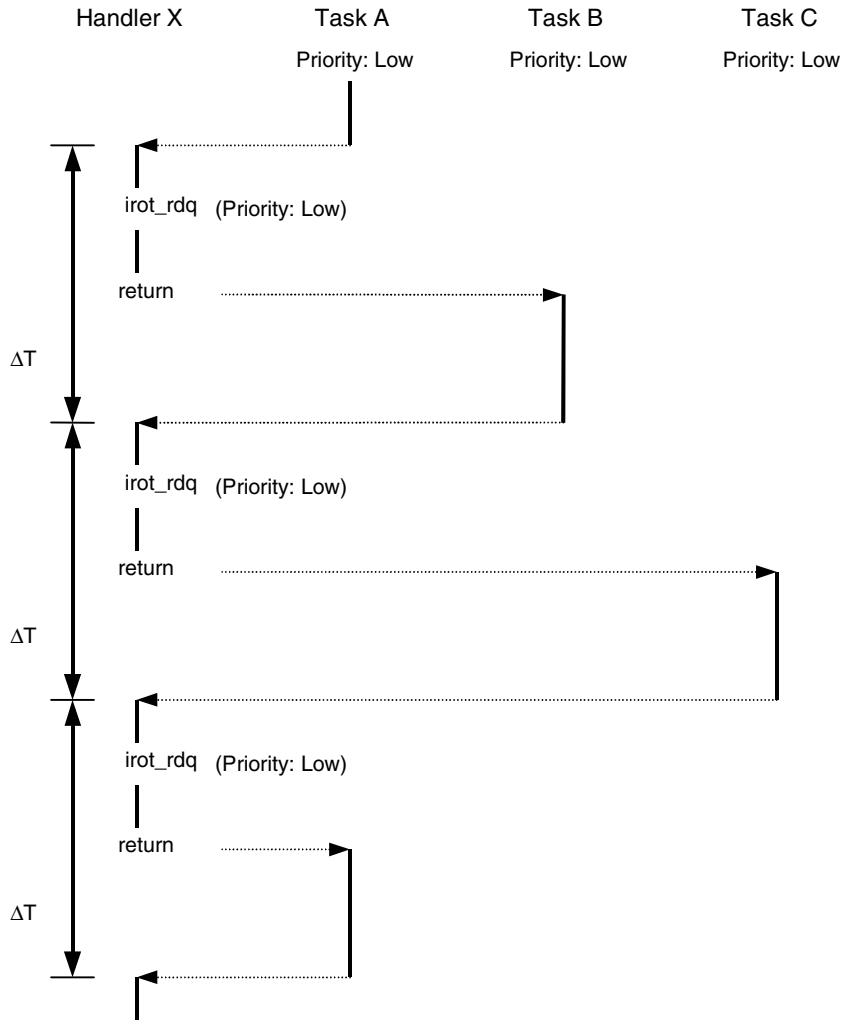
Figure 3-4. State 3 of Ready Queue During Round Robin Processing



Repeating the processing described above realizes a round robin system.

Figure 3-5 shows the processing flow when round robin scheduling, in which 1 to 3 above are repeated, is performed.

Figure 3-5. Processing Flow of Round Robin Scheduling



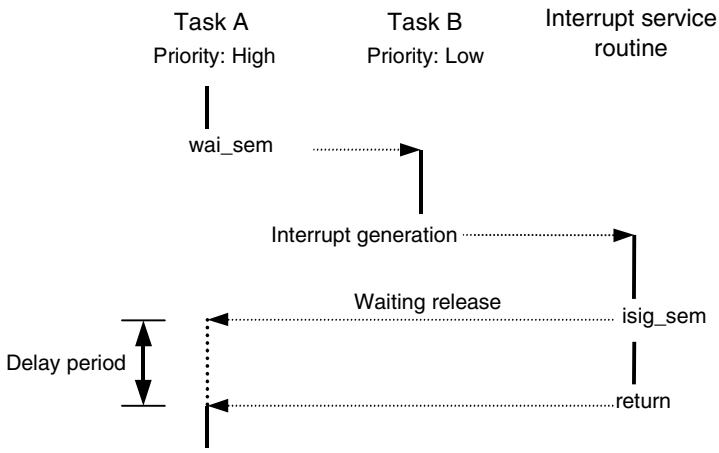
3.6 Scheduling Delay

The execution order among processing units in the RX4000 such as tasks, interrupt service routines, and the scheduler is set as follows, based on the μITRON4.0 Specification.

CPU exception handler = Interrupt service routine > Cyclic handler
> Scheduler > Task exception processing routine
> Task (task main processing routine) > Idle routine

In other words, even if scheduling is required such as releasing the waiting state of a task while a CPU exception handler, interrupt service routine, or cyclic handler is being executed, the scheduler cannot be driven because the processing under execution has a higher priority than the scheduler. This is because the processing performed by CPU exception handlers, interrupt service routines, and cyclic handlers is generally highly urgent and indivisible, and must therefore be carried out quickly. Consequently, because scheduling will not occur while the above are being processed, there may be a delay between when a service call that requires scheduling is issued and when the scheduling actually takes place (see **Figure 3-6**).

Figure 3-6. Scheduling Delay



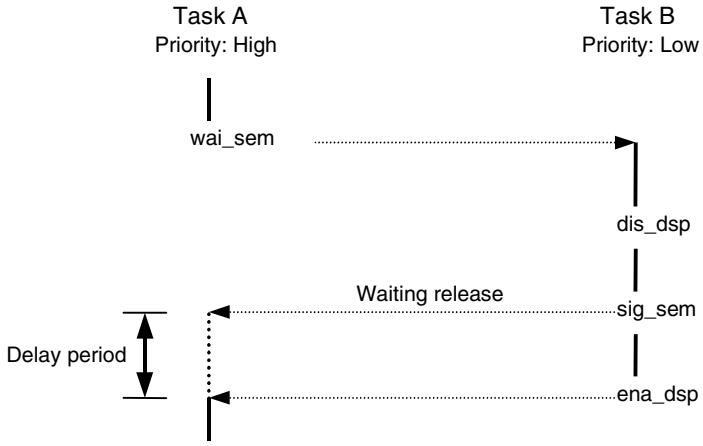
3.7 Enabling/Disabling Scheduling

In the RX4000, dispatch, or scheduler activation, can be explicitly enabled or disabled by the issuance of a service call from a task. If dispatch is disabled by a service call, scheduling processing is suspended and does not resume until a service call is issued to enable dispatch. The service calls used to enable and disable dispatch are as follows.

dis_dsp: Disables dispatch
ena_dsp: Enables dispatch
loc_cpu: Puts the system in a CPU-locked state
unl_cpu: Releases the system from the CPU-locked state

The processing flow when enabling and disabling dispatch is shown below.

Figure 3-7. Disabling Dispatch



3.8 Idle Routines

An idle routine is a processing routine that is activated from the scheduler if all the tasks are either complete or in a waiting state, meaning that there are no longer any tasks in the ready state and therefore subject to scheduling (idle state). Idle routines can be described by users to accord with the configuration of their system, enabling capitalization of the low power consumption mode functions provided by the target VR Series processor.

Note that an idle routine is described as a void type function without an argument.

Example)

```
void idle_routine(void)
{
        standby();
        return;
}
```

A service call must not be issued in an idle routine. Nor can MIPS16 instructions be used.

3.8.1 Registering idle routines

Idle routines are registered by specifying the static API VATT_IDL or via the service call vatt_idl. One idle routine must always be registered in the system, so if the above processing is not performed, the kernel will register a default idle routine. A new idle routine can be registered when another idle routine is already registered (the previously registered data will be discarded), but idle routine registration cannot be canceled.

The default idle routine simply performs unlimited loop processing.

3.8.2 Executing and terminating idle routines

If the scheduler is activated and judges that there are no executable tasks available, the kernel immediately executes an idle routine. When the idle routine has completed C language return or equivalent processing, it is executed again.

Return from an idle state to the normal state occurs when the scheduler is activated after the processing returns from servicing an interrupt that was generated in the idle routine, following which a task became executable. In this case, the processing does not return to where the interrupt was generated and the next processing of the idle routine is discarded.

3.8.3 PID

If the idle routine registered by the user references PID (Position Independent Data), the value of the gp register that should be set as the PID parameter must be set when the idle routine is registered. The assigned address is set in the gp register when the idle routine is activated.

3.8.4 Coprocessor

The FPU (coprocessor 1) cannot be used in an idle routine.

3.8.5 Stack

An idle routine uses the stack that is used by the system (syssp of the system base table).

CHAPTER 4 TASK MANAGEMENT

This chapter describes the task management performed by the RX4000.

4.1 Overview

The processing program managed by the kernel is made up of units such as tasks, interrupt service routines, and cyclic handlers. Unlike interrupt service routines, which are activated by interrupt generation, or cyclic handlers, which are activated after a certain time has elapsed, tasks, which are the basic processing units of the system, are not activated unless expressly manipulated by service calls.

In the RX4000, tasks are managed using data structures that correspond to tasks on a one-to-one basis. These data structures are known as task control blocks. The task control block secures all the data required for task management when tasks are created, and discards this data when tasks are deleted. For further details of task control blocks, refer to the **RX4000 (μITRON4.0) Technical User's Manual (U14835E)**. The major items included in a task control block are shown below.

- Task address
- Priority
- Current status
- ID number of management object for which task is waiting
- Stack pointer, etc.

Note that tasks are divided into task main processing routine and task exception processing routine blocks. Since the task main processing routine is what is usually executed, unless specified otherwise, it is described in this document simply as a "task".

Tasks are described as void type functions with one VP_INT type argument. The extended data exinf (activated by act_tsk) or the task activation code stacd (activated by sta_tsk) is passed for this argument.

Example)

```
void task(VP_INT exinf)
{
        ...
        ext_tsk();
}
```

4.2 Creating Tasks

Tasks are created by issuing the service calls cre_tsk and acre_tsk. With acre_tsk, assignment of the ID number can be performed by the kernel. Specifying the static API CRE_TSK enables processing equivalent to cre_tsk to be performed at system initialization. CRE_TSK also allows activation processing to be performed at the same time as task creation.

In task creation processing, the kernel recognizes data (text) expanded in the memory as a task and places that task under its management. Task creation processing involves securing and initializing the index (ID) block specified from the task control block array in the system pool, and securing the stack area from the stack pool. Each created task has a unique ID number. The ID number must be an integer in a range of 1 to 0x7fff, and the maximum number that can be assigned is the number specified in the CF definition file.

Note that the size of the stack used by a task must be specified when the task is created. For how to calculate this size, refer to the **RX4000 (μITRON4.0) Installation User's Manual (U14834E)**.

4.3 Deleting Tasks

In cases such as when a task has completed processing and restarting that task is no longer necessary, the task can be put into a non-existent state by issuing del_tsk; a process known as task deletion. It is also possible to perform task termination and deletion together using the service call exd_tsk. When a task is deleted, its task control block is invalidated, and can be used for a newly created task. Note that when a task is deleted, its stack area is also released.

4.4 Activating Tasks

Two service calls are supplied in the RX4000 for activating tasks: (i)act_tsk and (i)sta_tsk. (i)act_tsk is used to retain an activation request.

4.4.1 Activation with activation request retained

When (i)act_tsk is issued, if the target task is in the dormant state, it shifts from the dormant state to the ready state, becoming subject to scheduling by the kernel. If the target task is in a state other than the dormant (or non-existent) state, when (i)act_tsk is issued, the activation request is retained, causing the target task to be reactivated as soon as it has terminated.

The extended data exinf specified when a task is created is passed as the activation parameter for tasks activated by (i)act_tsk. Note that if TA_ACT is assigned as the attribute of a task created by the static API CRE_TSK, the activation processing of that task is equivalent to that of act_tsk.

4.4.2 Activation with activation request not retained

(i)sta_tsk is provided in the RX4000 as a task activation method compatible with the μITRON3.0 specification, in which tasks are activated without retention of the activation request.

If (i)sta_tsk is issued for a task in the dormant state, the task shifts from the dormant state to the ready state, becoming subject to scheduling by the kernel. If (i)sta_tsk is issued for a task that is in a state other than dormant, an error occurs, and the activation request is not retained.

For tasks activated by (i)sta_tsk, the VP_INT type data (4 bytes), which was specified as the (i)sta_tsk parameter, can be received as the task activation code instead of the extended data exinf.

4.5 Terminating Tasks

Task termination occurs when a task that was activated and executing processing shifts back into the dormant state. Tasks can be terminated either normally or forcibly, as defined by the service call that terminates the task. Note that a terminated task's priority, wakeup request number, and suspend request number are initialized to the values set when the task was created.

4.5.1 Normal termination

When a task has completed processing and no longer needs to be subject to scheduling, it issues the service call ext_tsk or exd_tsk and self-terminates. Task termination via ext_tsk or exd_tsk is commonly known as normal termination. Note that ext_tsk effects termination only, whereas exd_tsk both terminates and deletes the task. If the task is locking any mutexes, these locks are released when the task terminates.

4.5.2 Forcible termination

If necessary, a task can be urgently terminated by issuing the service call ter_tsk. A task that receives this service call shifts into the dormant state and terminates, regardless of the state it was in at that time. This processing is known as forcible termination.

If the task is locking any mutexes, these locks are released when the task is forcibly terminated.

4.5.3 Reactivating terminated tasks

In μITRON4.0, the activation request for a task can be retained, and that task can be reactivated as soon as it terminates (whether normally or forcibly). In other words, the task is shifted immediately from the dormant state back to the ready state upon termination.

Note, however, that the activation request for a task that was terminated by exd_tsk is ignored, and the task is shifted to the non-existent state.

4.6 Task States

The task changes its state according to how resources required to execute the processing are acquired, whether an event occurs, and so on. Task states are always managed by the kernel, which performs the processing required according to that state. The seven states applicable to tasks are described below.

(1) Non-existent

A task in this state has not been created or has been deleted, and is not registered in the kernel. A task in the non-existent state is therefore not managed by kernel even though its code is located in memory.

(2) Dormant

A task in this state has just been created or has already completed its processing. A task in the dormant state is not subject to scheduling by the kernel, even though it is under the kernel's management.

(3) Ready

A task in this state is ready to perform its processing, but has been waiting to be assigned the right to use the CPU (execution right) because another task with the same or higher priority is being executed.
A task in the ready state is subject to scheduling by the kernel, and can be shifted to the running state as soon as it receives the execution right.

(4) Running

A task in this state has been assigned the execution right and is either currently performing its processing or has had its processing interrupted while an interrupt service routine or handler is being executed.
Within the entire system, only a single task can be in the running state at any one time.

(5) Waiting

A task in this state has been stopped because the requirements for performing its processing, such as the acquisition of a semaphore or memory block, have not been satisfied. The processing of this task is resumed from the point at which it was stopped. At this time, the data (context) required to execute the program is restored to the state it was in immediately before the stoppage. The acquisition status, etc., of resources that are unrelated to the wait are not affected by the stoppage/resumption of execution.
The waiting state is further divided into the following types of states, according to the wait source. For further details of these states, refer to the next and subsequent chapters.

Type	Waiting for:	Service call Causing Wait
Wake-up wait	Reception of wake-up request	(t)slp_tsk
Time elapse wait	Lapse of time	dly_tsk
Semaphore wait	Acquisition of resource from semaphore	(t)wai_sem
Event flag wait	Establishment of event flag condition	(t)wai_flg
Data transmission wait	Completion of data write to buffer	(t)snd_dtq
Data reception wait	Reception of data	(t)rcv_dtq
Message reception wait	Reception of a message	(t)rcv_mbx
Mutex wait	Locking a mutex	(t)loc_mtx
Fixed-length memory block wait	Acquisition of fixed-length memory block	(t)get_mpf
Variable-length memory block wait	Acquisition of variable-length memory block	(t)get_mpl

(6) Suspended

A task in this state has been forcibly stopped by the issuance of the service call (i)sus_tsk. In μITRON3.0, (i)sus_tsk was only recognized if issued by another task or handler, but in μITRON4.0, a task can issue sus_tsk to itself.

Because the suspended state can be nested by issuing (i)sus_tsk in multiple (i.e., more than once for the same task), it is necessary to issue the same number of (i)rsm_tsk service calls, or issue (i)frsm_tsk, to release the suspended state.

The processing of this task is resumed from the point at which it was stopped; i.e., the point at which it was preempted by the task or handler that issued (i)sus_tsk. At this time, the data required to execute the program, such as the register values, is restored to the state it was in immediately before the stoppage. The acquisition status, etc., of task resources that are unrelated to the wait are not affected by the stoppage/resumption of execution.

(7) Waiting-suspended

This state is a combination of the waiting and suspended states described in (5) and (6) above.

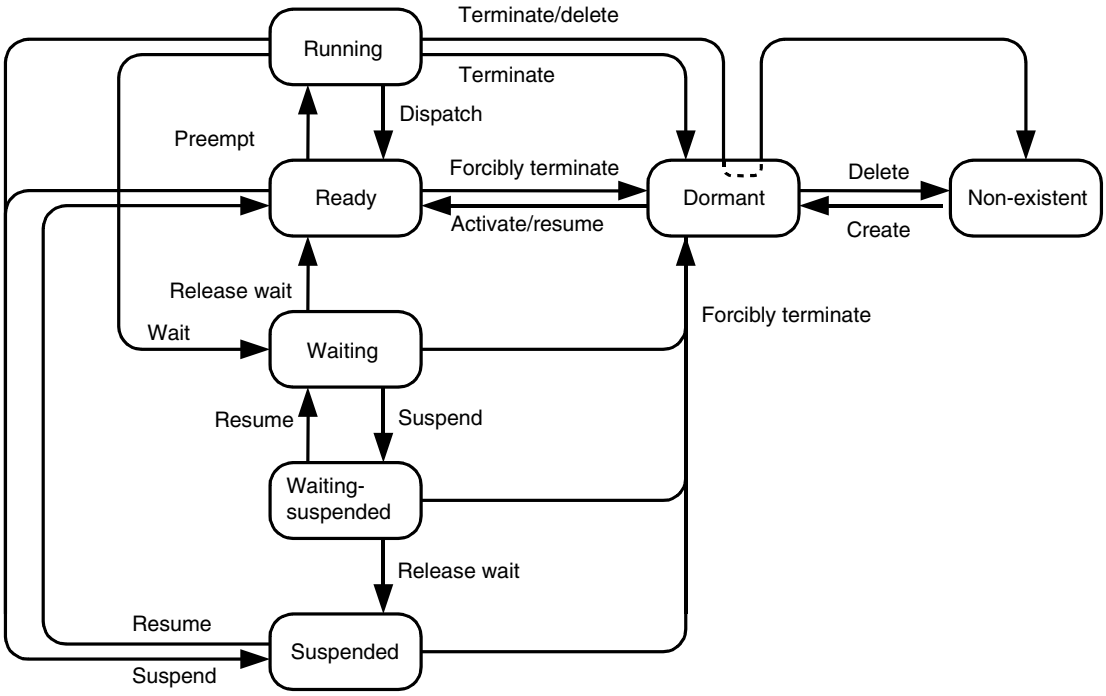
A task in this state has entered the waiting state upon exiting the suspended state, or has entered the suspended state upon exiting the waiting state.

Like the suspended state, the waiting-suspended state enables nesting of states through the multiple issuance of (i)sus_tsk. It is therefore necessary to issue the wait release service call (rel_wai, sig_sem, etc.) and the same number of (i)rsm_tsk service calls as (i)sus_tsk calls or (i)fsm_tsk to shift a task in the waiting-suspended state to the ready state (or in some cases, immediately to the running state).

4.7 Task State Transition

Tasks are repeatedly shifted between the states described in **4.6 Task States** by means of service call issuance. Task states and state transitions are shown in Figure 4-1 below.

Figure 4-1. Task State Transition



4.8 Task Delay and Timeout

Tasks in the waiting state described in **4.6 (5) Waiting** above can be released after an arbitrary period of time has elapsed. It is also possible to delay the execution of a task for a specified amount of time using the service call dly_tsk. This kind of processing is known as timer operation. For details of timer operation, refer to **CHAPTER 7 TIME MANAGEMENT**. Timer operation as related to tasks is summarized below.

(1) Timeout

When a task is shifted to the waiting state, the maximum time that task is to stay in the waiting state can be set by specifying a timeout time. In other words, if the wait release conditions are not met before the specified time elapses after a service call was issued, the task is forcibly released from the waiting state. This is known as timeout.

The error code E_TMOUT indicating a timeout is returned as the return value of a service call for tasks in a timeout state.

(2) Time elapse wait

A task can be made to stay in the waiting state for only a specified time by issuing the service call dly_tsk. In other words, a task is subject to waiting for a specified time, after the elapse of which the task is released.

4.9 Task Priority Order

A task has three values related to its priority: its initial priority, its current priority, and its base priority. In the RX4000, the lower the value of the priority assigned to a task, the higher the priority.

(1) Initial priority

This is the priority of a task immediately after activation. This value can only be specified when the task is created, and cannot be subsequently changed. When a task terminates and is reactivated, its priority is initialized to this value.

(2) Current priority

This value indicates the priority of the task after it has been activated. The task execution order or the place of a task in the task queue is determined by its current priority value. The current priorities of tasks can be dynamically changed by the service call chg_pri (ichg_pri) and can be obtained by the service call get_pri (iget_pri).

(3) Base priority

When a task is not participating in mutex-based synchronization, the base priority is always the same as the current priority. When the task is synchronized using a mutex, the current priority of the task may change, depending on the priority control protocol of the target mutex. At this time, the priority order to which the task returns after releasing the mutex lock is known as the base priority.

4.10 Task Context

The environment in which the program operates is generally known as the context. In other words, the context indicates the stack area (space) used by the CPU operation mode or program and the statuses (registers) saved to the stack during program execution.

In the same way, the environment in which tasks operate is known as the task context. Therefore, when processing such as service call issuance from a task is performed, it is sometimes called “performing processing from a task context”. Moreover, when task switching, such as shifting a task to the waiting state, occurs, the value of the register of when the task was being executed is saved to (or restored from) the stack area. This register value saved in the stack area is called the task context.

For details of the types of registers that include task contexts and the structures used when contexts are written to the memory, refer to the **RX4000 (μITRON4.0) Technical User’s Manual (U14835E)**.

4.11 PID

If PID (Position Independent Data) is used for the code created when the task program is compiled or assembled, the address that is to be the base of a task when it is created must be assigned as a parameter (gp of the task creation packet T_CTSK). The assigned address is set in the gp register when the task is activated and saved to or restored from the context when the task is switched.

Note that this base address is unconditionally set in the gp register regardless of its attribute, etc., and therefore must be set for programs that reference the gp register. If the gp register is not referenced, set NULL = 0.

4.12 Obtaining Task Information

Task information such as the task status can be obtained by using the service calls ref_tsk, iref_tsk, ref_tst, and iref_tst. For details of each service call, refer to **CHAPTER 13**.

4.13 Coprocessor

When a task is assigned the attribute TA_COP at creation, thus specifying use of the FPU (coprocessor 1), FCR31 (control/status register of FPU) at activation has the same value as FCR31 of the task executed immediately before this task is activated. If it is necessary to use FCR31 of the task to be activated, to generalize the processing and reduce the overhead, instead of using the kernel to perform the processing that sets FCR31, this processing must be described in the prolog section of the function described as a task.

Also, the number of FPU registers that can be used by tasks with the attribute TA_COP is determined uniformly for the entire system, and therefore specified according to the configuration of the system (CF definition file). The status registers required for task execution (CU, FR) are set by the kernel according to the TA_COP attribute and configuration specifications.

In addition, the FCR31 and FPU registers are included in the context of tasks to which the TA_COP attribute has been assigned, and are saved/restored when those tasks are switched.

4.14 Task Exceptions

4.14.1 Task exception processing routines

A task exception processing routine is a routine that is activated when an event exceptional to a task occurs. Because tasks can be divided into task main processing and task exception processing routines, the task exception processing routine is actually a part of a task and therefore operates on the same context as the task main processing routine.

In other words, when a task exception processing routine is activated, the applied parameters, such as the stack to be used, the priority order, the status of interrupts (enabled/disabled), and the use of the coprocessor, are identical to those of the task, or task main processing routine with which the exception processing routine is paired.

Note that task exception processing routines are described as void type functions with FLGPTN and VP_INT type arguments. The bit pattern of the exception source that triggers activation of the task exception processing routine (described later) and the extended data held by the tasks are passed for these arguments.

Example)

```
void task_exception(FLGPTN texptn, VP_INT exinf)
{
        ...
        return;
}
```

4.14.2 Defining task exception processing routines

A task exception processing routine is defined by issuing def_tex. Equivalent processing to def_tex can also be realized by specifying the static API DEF_TEX when the system is activated up.

def_tex specifies a task ID number (or the ID number of the task (task main processing routine) with which it is paired) as its parameter. Note that if def_tex is issued for a task for which a task exception processing routine has already been defined, the previous definition is deleted, and the new exception processing routine definition becomes valid.

4.14.3 Canceling task exception processing routine definitions

A task exception processing routine definition is canceled by issuing def_tex with NULL specified in the definition packet address of the exception processing routine.

4.14.4 Task exception enabled/disabled states

Tasks have two states related to task exception processing: a task exception enabled state and a task exception disabled state. In the latter state, task exception processing requests are held pending, and the task exception processing routine is not activated. The conditions under which the enabled/disabled state changes are summarized below.

Task exceptions become disabled when:

- A task is activated
- def_tex is issued and a new exception processing routine is defined
- def_tex is issued and an exception processing routine is redefined in the task exception disabled state (continuance of disabled state)
- def_tex is issued and an exception processing routine is canceled
- dis_tex is issued
- A task exception processing routine is activated

Task exceptions become enabled when:

- ena_tex is issued
- Processing shifts from a task exception processing routine to a task (except when dis_tex is issued in an exception processing routine)
- def_tex is issued and an exception processing routine is redefined in the task exception enabled state (continuance of enabled state)

4.14.5 Task exception processing requests

When some kind of event exceptional to a task occurs, (i)ras_tex is issued and a task exception processing execution request is sent to that task. To issue (i)ras_tex, a 32-bit bit string (TEXPTN type), which is held as the exception source when the exception processing routine is activated (pending exception source), is specified. If (i)ras_tex is issued more than once before the task exception processing routine is actually activated, the activation source is updated with the logical sum of the values specified by the multiple (i)ras_tex service calls.

When a task exception processing routine is activated, the pending exception source is passed as the parameter of the task exception processing routine, and then cleared to 0. Although it is possible to request exception processing for the same task by issuing iras_tex from an interrupt servicing routine or cyclic handler activated while the task exception processing routine is being executed, in this case, the pending exception source will only be updated, and there will be no effect on the exception source of the task exception processing routine being processed.

4.14.6 Activating task exception processing routines

A task exception processing routine is activated by the kernel when all of the following four conditions are met.

- Task exceptions are in the enabled state
- The pending exception source is not 0
- The task is in the running state
- The non task context of an interrupt service routine or a cyclic handler is not under execution

In other words, the conditions for tasks when task exceptions are enabled are:

(1) The task has been put in the running state by the issuance of a dispatch and the pending exception source is not 0 at that time
(2) The pending interrupt source is not 0 when interrupt processing has finished and processing returns to the task
(3) ras_tex has been issued by a task to itself
(4) The pending interrupt source is not 0 when the task exception processing routine has finished

In the above cases, the task exception processing routine is activated immediately before control is shifted to the task (main processing routine). The processing flow when a task exception processing routine is activated is shown below.

Figure 4-2. Example of Task Exception Processing Routine Activation (1)

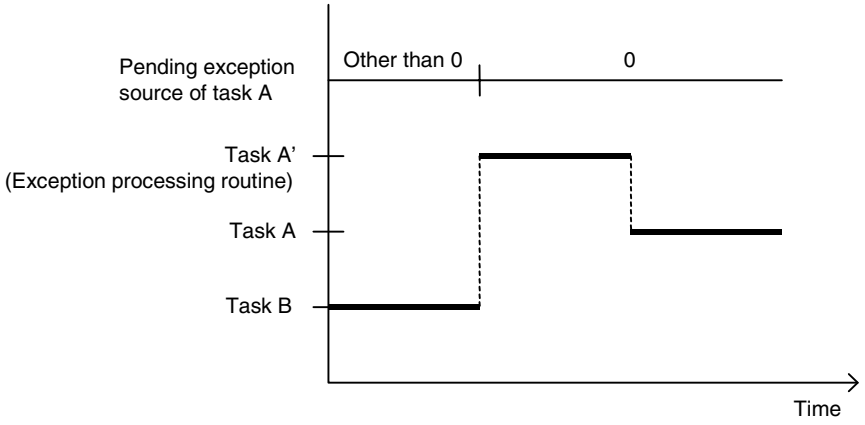


Figure 4-3. Example of Task Exception Processing Routine Activation (2)

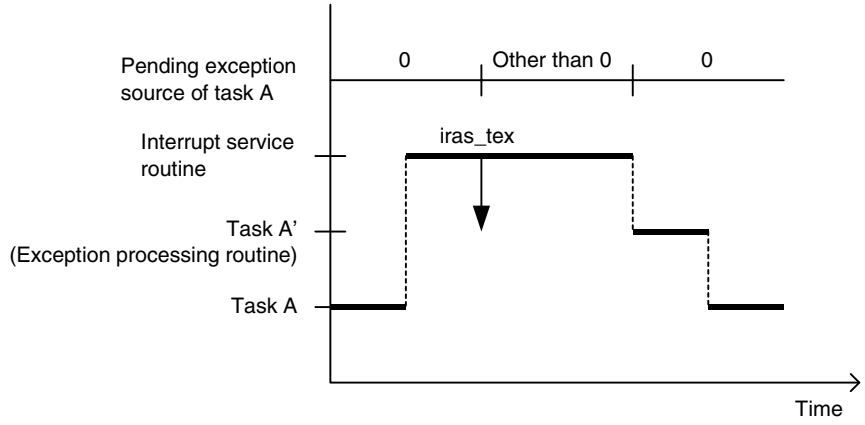


Figure 4-4. Example of Task Exception Processing Routine Activation (3)

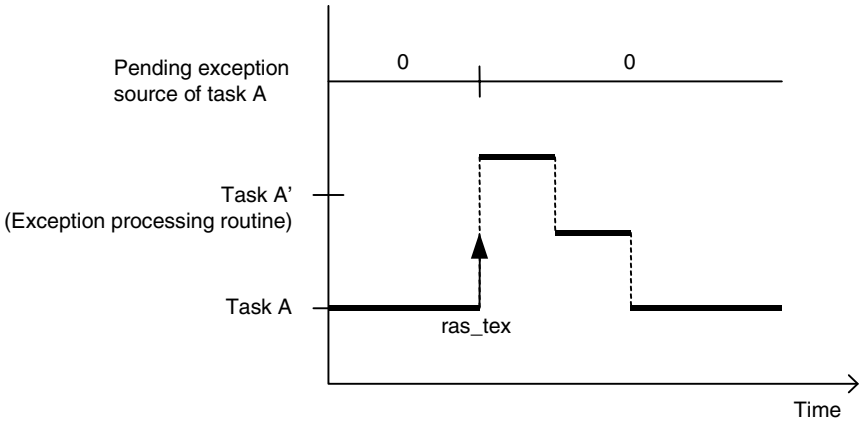
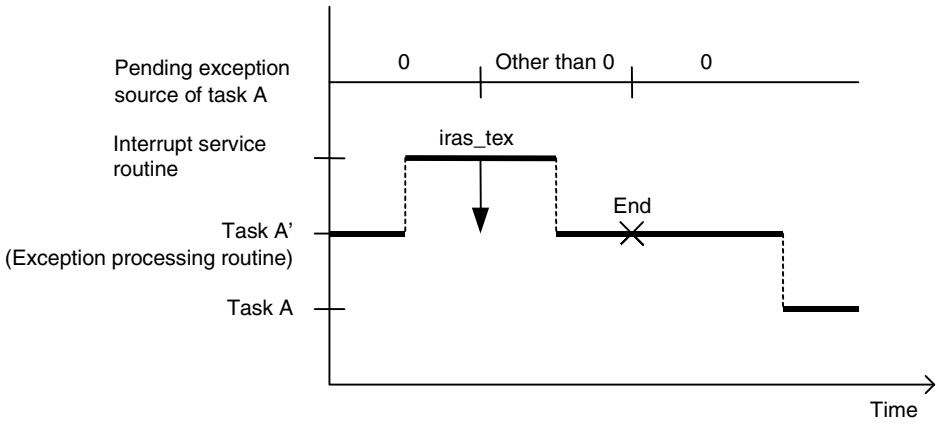


Figure 4-5. Example of Task Exception Processing Routine Activation (4)



Note that in case (3), “the task itself” includes tasks undergoing task exception processing. In other words, if task exceptions have been enabled by the issuance of ena_tex during task exception processing, multiple task exception processing routines may be activated by the exception request issued from interrupt processing, etc. However, be aware that in this case, an unlimited number of task exception processing routines may be inadvertently activated.

The exception source and extended data held by the task can be received as parameters for activated task exception processing routines.

4.14.7 Terminating task exception processing routines

C language return or equivalent processing is carried out to terminate a task exception processing routine. Once a task exception processing routine is terminated, the kernel rechecks the pending exception source, and if it is not 0, reactivates the task exception processing routine. If the pending exception source is 0, the task exception disabled state of the task is released (except for cases when dis_tex has been issued during task exception processing), and control is returned to the task main processing routine.

4.14.8 Issuing service calls from task exception processing routines

All the service calls that can be issued from a task can be issued from a task exception processing routine. In other words, there is no difference between the service calls that can be issued from the task's main processing and exception processing routines. Therefore, if a service call that transfers a task to a waiting state, such as slp_tsk, or a service call that terminates a task, such as ext_tsk or exd_tsk, is issued from a task exception processing routine, the processing that is subsequently carried out is the same as if the service call was issued from the task main processing routine.

4.14.9 Obtaining task exception processing routine information

Information on the task exception processing routine can be obtained by using the service calls sns_tex, ref_tex, and iref_tex. For details of each service call, refer to **CHAPTER 13**.

CHAPTER 5 SYNCHRONOUS COMMUNICATION MANAGEMENT

This chapter describes the synchronous communication management performed by the RX4000.

5.1 Overview

In an environment where multiple tasks are executed concurrently (multitasking), the result produced by a certain task may determine the next task to be executed or affect the processing performed by the subsequent task. In other words, it may happen that some task execution conditions vary with the processing performed by another task, or that the processing performed by some tasks is related.

Liaison functions are therefore required between tasks so that task execution will be suspended to await the result output by another task. These functions are known as synchronization and communication functions, and in the RX4000 the synchronization and communication functions provided include semaphores, event flags, data queues, mailboxes, and mutexes. These functions are described in the following sections.

5.2 Semaphores

The elements required to execute tasks are known as resources, which include all the hardware components, such as the processor, memory, and I/O devices, and software components, such as the files and programs.

Because it is often impossible for multiple tasks to simultaneously use the same resource, a multitasking environment requires a function to prevent possible resource contention. As a means of realizing the exclusive control of resources that would prevent this kind of contention, the RX4000 provides non-negative counter-type semaphores. Semaphores contain counters for controlling the number of resources, and by treating semaphores as abstract resources, it becomes possible to perform exclusive control between tasks.

5.2.1 Creating semaphores

Semaphores can be created either by issuing the service call (a)cre_sem, or by specifying the static APICRE_SEM, which performs equivalent processing to (a)cre_sem when the system is initialized.

If (a)cre_sem is issued, the attribute, initial counter value, and maximum counter value (etc.) are stored in the block corresponding to the ID number that specifies the semaphore control block area secured as an array, and that control block is then initialized.

The semaphore ID number consists of a unique number of a value 1 or higher. The maximum value that can be specified is the one defined in the system information table, up to a maximum of 0x7fff numbers. Any value of 0 or higher can be specified for a semaphore's initial counter value, and any value of 1 or higher for the maximum counter value, but the former cannot exceed the latter. The maximum specifiable value is 0x7fffff.

5.2.2 Deleting semaphores

A semaphore is deleted by issuing the del_sem service call. In other words, once del_sem is issued, the kernel invalidates the specified semaphore control block and puts the target semaphore in the non-existent state.

Even if a task exists that has acquired a resource for the deleted semaphore, that semaphore will still be deleted. In this case, all the waiting tasks are released from the waiting state and the error code E_DLT indicating that the semaphore has been deleted is returned as the return value of the service calls wai_sem and twai_sem. Note, however, that tasks that have already acquired a semaphore will not be notified of that semaphore's deletion.

After a semaphore is deleted, a semaphore with the same ID number as the deleted semaphore can be newly created.

5.2.3 Acquiring resources

Resources are acquired from semaphores by issuing one of the service calls wai_sem, twai_sem, or pol_sem.

If there is a request to acquire a resource, the kernel checks the value of the resource counter of the target semaphore and assigns the resource to the task if the counter is 1 or higher (decrementing the counter by 1), or carries out the processing corresponding to the issued service call if the counter is 0. In other words, a task is put in the resource acquisition waiting state by the service calls wai_sem and twai_sem, and waits until the resource has been acquired, in the case of wai_sem, or until either the resource has been acquired or a specified time has elapsed, in the case of twai_sem. The issuance of pol_sem results in a service call error, and the task is informed that resource acquisition failed.

Tasks in the resource acquisition waiting state are registered in the waiting task queue of the specified semaphore. It is therefore possible to have multiple tasks waiting for the same semaphore, in which case a request for resource acquisition sent to a semaphore for which tasks are already waiting will not result in an error; the task will simply be put in the resource acquisition waiting state.

Tasks can be registered in the waiting task queue of a semaphore by two methods, depending on the attribute specified when the semaphore was created. One method is registering tasks in the queue in the order of their priorities, which is used for semaphores with the attribute TA_TPRI. The other method is registering tasks in the queue in the order in which they requested resources, which is used for semaphores with the attribute TA_TFIFO. If waiting tasks are released by the issuance of the service call (i)sig_sem, the tasks are always released in order from the top of the queue.

5.2.4 Returning resources

A resource is returned to a semaphore by issuing the service call (i)sig_sem.

When (i)sig_sem is issued, the kernel checks the waiting queue of the target semaphore and if there are tasks registered in the queue, it immediately assigns the task at the top of the queue the resource and releases it from the waiting state. At this time, the value of the resource counter remains unchanged. If there are no tasks registered in the waiting queue, the resource counter value is incremented by 1. Note that because only one resource can be manipulated per service call, multiple tasks are never released simultaneously by one issuance of (i)sig_sem.

5.2.5 Obtaining semaphore information

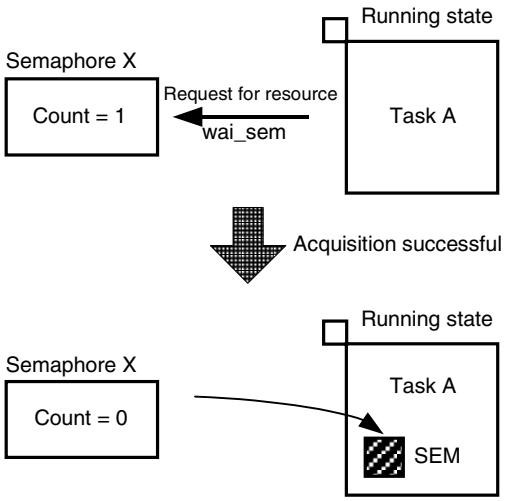
Semaphore information such as the number of resources can be obtained by using the service calls ref_sem and iref_sem. For details of each service call, refer to **CHAPTER 13**.

5.2.6 Examples of exclusive control using semaphores

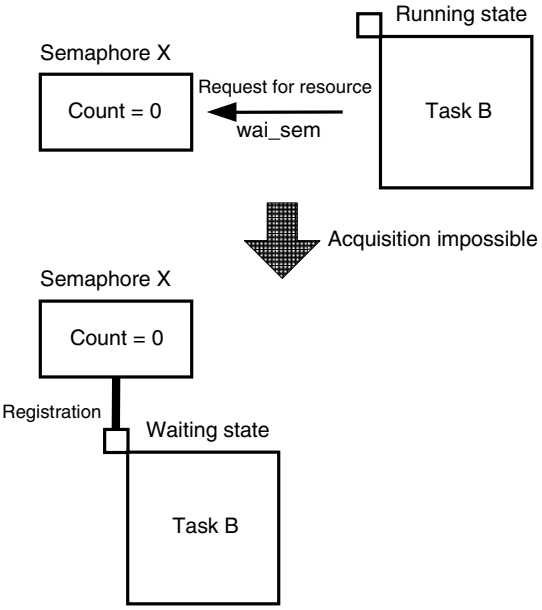
Some examples of exclusive control using semaphores are shown below.

(1) cre_sem is issued creating a semaphore. The value of the initial counter is 1.

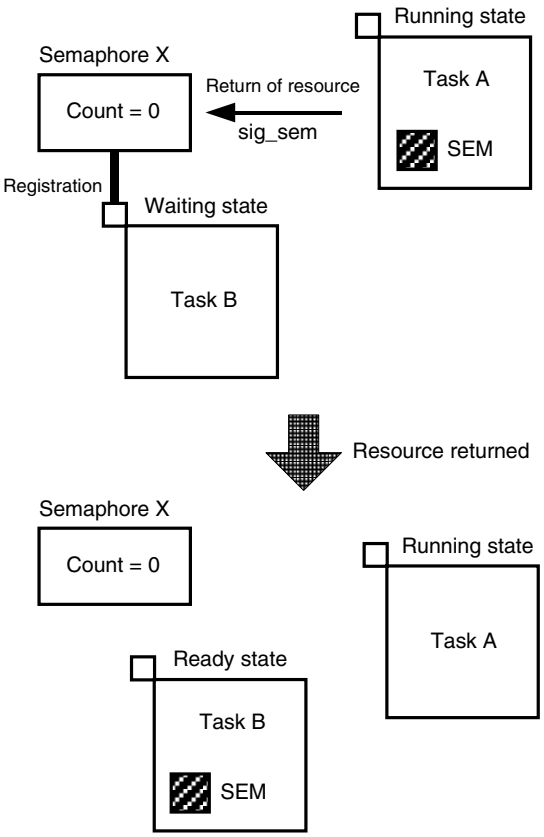
(2) For task A to use a resource, wai_sem is issued and a resource is acquired from the semaphore.



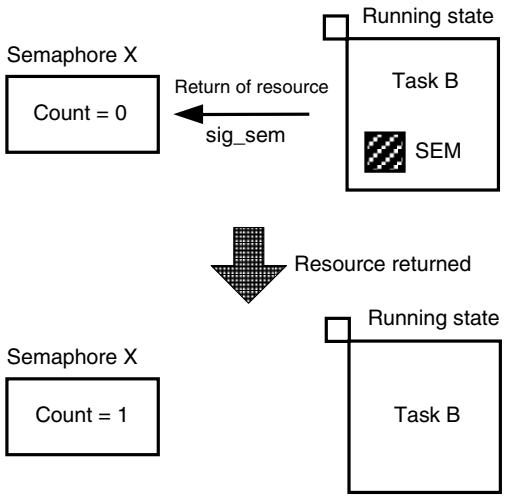
(3) For task B to use a resource, wai_sem is issued and an attempt is made to acquire a resource from a semaphore. However, because the value of the semaphore resource counter is 0, task B is put in the waiting state and is registered in the semaphore's waiting task queue.



(4) When task A has finished using the resource, sig_sem is issued and the resource is returned to the semaphore. Task B acquires the resource and is subsequently released from the resource acquisition waiting state, and removed from the semaphore's waiting task queue.



(5) When task B has finished using the resource, sig_sem is issued and the resource is returned to the semaphore. The semaphore's resource counter is subsequently incremented by 1.



5.3 Event Flags

In multitask processing, an intertask wait function is required in which other tasks wait to resume execution of processing until execution of a given task is complete. To support this function, event flags are provided in the RX4000 to allow other tasks to judge whether or not the “processing complete” event has occurred.

An event flag is a set of data consisting of 1-bit flags that indicate whether a particular event has occurred. The event flags used in the RX4000 are 32 bits, which are handled as a set of information with each bit or a combination of bits having a specific meaning.

5.3.1 Creating event flags

Event flags can be created either by issuing the service call (a)cre_flg, or by specifying the static API CRE_FLG, which performs equivalent processing to (a)cre_flg when the system is initialized.

If (a)cre_flg is issued, the attribute, initial counter value, and maximum counter value (etc.) are stored in the block corresponding to the ID number that specifies the event flag control block area secured as an array, and that control block is then initialized.

The event flag ID number consists of a unique number of a value 1 or higher. The maximum value that can be specified is the one defined in the system information table, up to a maximum of 0x7fff numbers.

Any 32-bit value can be specified for an event flag’s initial bit pattern.

5.3.2 Deleting event flags

An event flag is deleted by issuing the service call del_flg. When del_flg is issued, the kernel invalidates the specified event flag control block and puts the target event flag in the non-existent state.

Even if a task exists that has satisfied the wait release conditions for the deleted event flag, that event flag will still be deleted. In this case, all the waiting tasks are released from the waiting state and the error code E_DLT indicating that the event flag has been deleted is returned as the return value of the service calls wai_flg and twai_flg.

After an event flag is deleted, an event flag with the same ID number as the deleted event flag can be newly created.

5.3.3 Waiting for events

To wait for the establishment of an event using an event flag, the required bit pattern is specified for either wai_flg, twai_flg, or (i)pol_flg, which is then issued.

When one of these service calls is issued, the kernel compares the bit pattern of the event flag when the service call was issued with the specified bit pattern, and determines whether an event has been established based on the wait conditions described later.

If an event has already been established, the service call is immediately terminated normally, allowing task processing to continue. If an event is not established, wai_flg and twai_flg make the task wait: until the event flag satisfies the wait release conditions in the case of the former, and until either the event flag satisfies the wait release conditions or a specified time has elapsed in the case of the latter. The issuance of (i)pol_flg results in a service call error, and the task is informed that an event was not established.

Note that event flags have two attributes: TA_WSGL and TA_WMUL, and those with the former attribute cannot be simultaneously held by multiple tasks (whereas those with the latter attribute can). Accordingly, if wai_flg, twai_flg, or (i)pol_flg is issued for an event flag with the attribute TA_WSGL already being held by a waiting task, an error occurs unconditionally, regardless of whether an event was established or not, and the error code E_OBJ is returned.

Because event flags with the attribute TA_WMUL can be held by multiple tasks, waiting tasks with event flags are registered in the waiting task queue, either in FIFO or priority order (TA_TFIFO and TA_TPRI attribute respectively). If (i)set_flg is issued, the kernel determines the wait release conditions of the tasks in the order of this waiting task queue.

5.3.4 Setting event flags

The value of an event flag is set by issuing either (i)set_flg or (i)clr_flg.

(i)set_flg is issued when any bit of the event flag is set to 1. When (i)set_flg is issued, the bit pattern of the event flag when the service call was issued is ORed with the specified bit pattern and set to the event flag as a new bit pattern. (i)clr_flg is issued when any bit of the event flag is set to 0. When (i)clr_flg is issued, the bit pattern of the event flag when the service call was issued is ANDed with the specified bit pattern and set to the event flag as a new bit pattern.

When there is a task waiting for an event flag, if (i)set_flg is issued and the event flag is updated, wait release condition judgement processing is carried out. This judgement processing is carried out either until a wait release condition is first satisfied (for event flags with the attribute TA_CLR), or for all the event flags. When (i)clr_flg is issued, only event flag update processing is carried out. This is because the judgement concerning the establishment of an event is made by checking whether all or any one of the specified bits are 1, as described in 5.3.5 Wait conditions, making it impossible for a condition to be established by issuing (i)clr_flg.

5.3.5 Wait conditions

One of the following judgement methods can be specified as a wait condition for wai_flg, twai_flg, and (i)pol_flg when determining whether an event has been established.

(1) AND wait (TWF_ANDW)

The waiting state continues until all bits to be set to 1 in the required bit pattern have been set in the relevant event flag. In other words, if the specified bit pattern is waiptn and the bit pattern of the event flag is curptn, the event establishment condition is as follows.

```
curptn & waiptn == waiptn
```

(2) OR wait (TWF_ORW)

The wait state continues until any bit to be set to 1 in the required bit pattern has been set in the relevant event flag. The event establishment condition is therefore as follows.

```
curptn & waiptn != 0
```

5.3.6 Event flag clear attribute

If an event flag has the attribute TA_CLR, it is cleared to 0 when the required condition is satisfied. An event flag with the attribute TA_WMUL held by multiple waiting tasks is cleared to 0 as soon as the first task is released from the waiting state. Accordingly, because the bit pattern subject to condition judgement is 0, wait release condition judgement is not performed for tasks registered behind this task in the waiting task queue.

5.3.7 Obtaining event flag information

Task information such as the bit pattern of an event flag can be obtained by using the service calls ref_flg and iref_flg. For details of each service call, refer to **CHAPTER 13**.

5.3.8 Examples of wait function using event flags

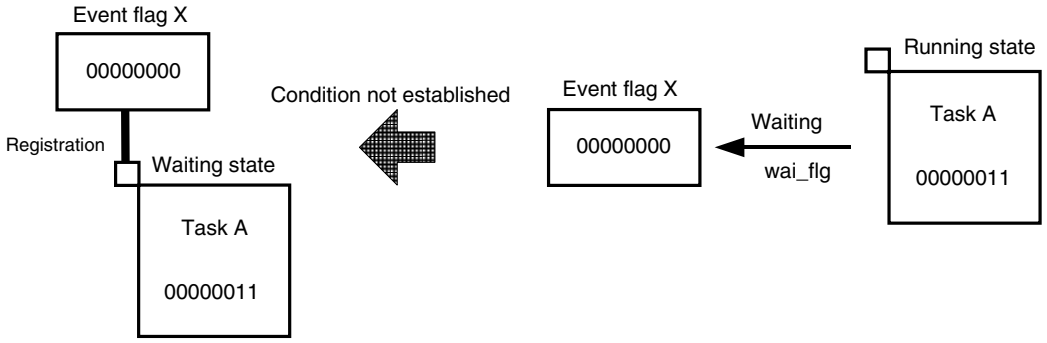
Some examples of wait processing using event flags are shown below.

The numbers in the diagrams indicate the current bit pattern of the event flag and the bit pattern required by the task. All these numbers are in binary. Note that although the bit width of event flags is usually 32 bits, for simplification purposes, this has been reduced to 8 bits in the examples.

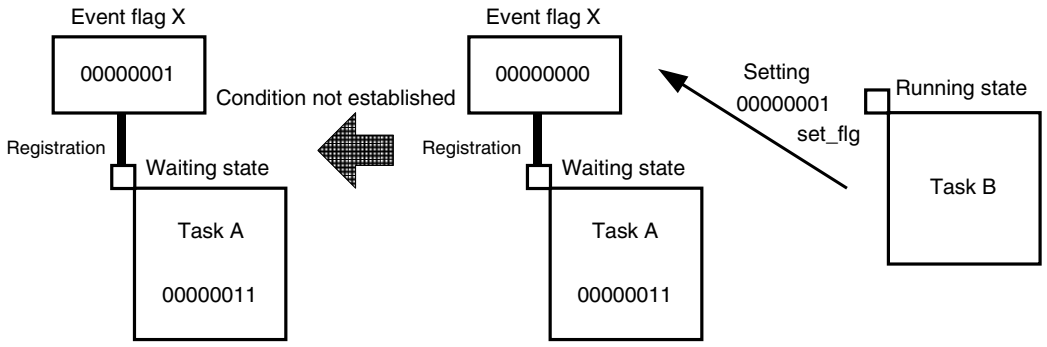
(1) cre_flg is issued creating an event flag. TA_CLR is specified for the event flag attribute and the initial bit pattern is 00000000.

(2) wai_flg is issued to make task A wait. At this time, the required bit pattern is 00000011 and the wait mode is TWF_ANDW.

Because the current bit pattern of the event flag is 0, the condition is not established, and task A enters the waiting state and is registered in the event flag's waiting task queue.

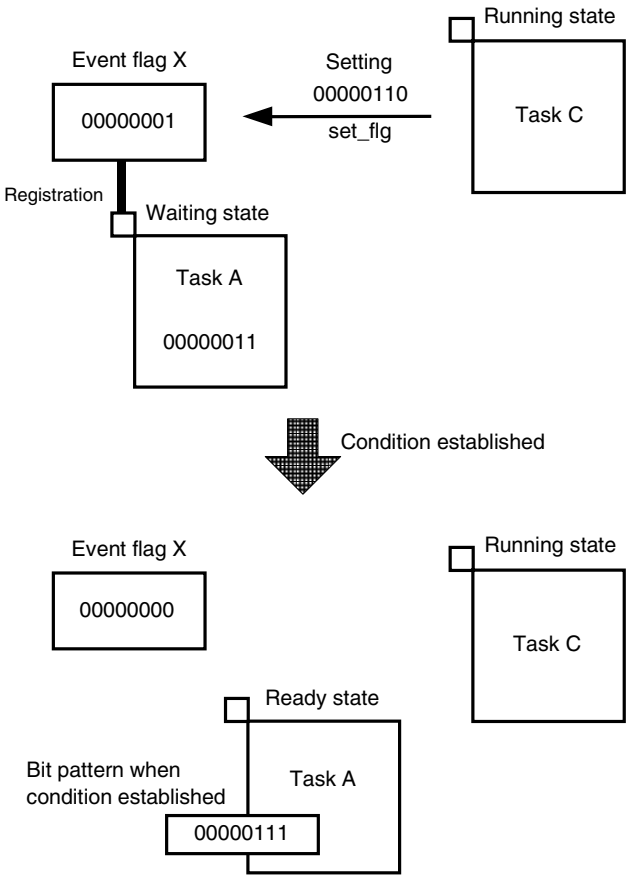


(3) set_flg is issued to make task B wait for task A and the value of event flag X is updated to 00000001. However, because the bit pattern required by task A is 00000011 and the wait mode is TWF_ANDW, not all the required bits have been set in event flag X, so task A continues waiting.



(4) set_flg is issued to make task C wait for task A and the value of event flag X is set to 00000110. The value of the flag before set_flg was issued, 00000001, and the newly set value, 00000110, are ORed, and event flag X is updated with the resulting value, 00000111, which satisfies the wait release condition of task A.

Task A is therefore released from the waiting state and the bit pattern of when task A was released from waiting, 00000111, is passed to task A as the return parameter. Also, because event flag X has the attribute TA_CLR, it is cleared to 0 after task A is released from waiting.



5.4 Data Queues

In a multitasking environment, when multiple tasks operate together to perform a specified processing, an intertask communication function is required to inform other tasks of the execution results of a certain task. In the RX4000, data queues and mailboxes are provided as a means of realizing this function. For a data queue, transmit data is fixed to 4 bytes and there is a limit to the total amount of data that can be stored. For a mailbox on the other hand, data of any format can be transmitted, and there is no limit to the total amount of storable data. Users can therefore select and use whichever of these has the features that accord with their system configuration. This section describes data queues, which consist of a ring buffer in which the data to be communicated is stored and a waiting task queue in which waiting tasks are registered. Data queues not only enable communication between tasks, but also make it possible to have tasks wait in response to data.

5.4.1 Creating data queues

Data queues can be created either by issuing the service call (a)cre_dtq, or by specifying the static API CRE_DTQ, which performs equivalent processing to (a)cre_dtq when the system is initialized.

When (a)cre_dtq is issued, firstly the area of the ring buffer to be used by the data queue is secured from the user pool. After that, the attribute, buffer address, and buffer size (etc.) are stored in the block corresponding to the ID number that specifies the data queue control block area, and that control block is then initialized.

The data queue ID number consists of a unique number of a value 1 or higher. The maximum value that can be specified is the one defined in the system information table, up to a maximum of 0x7fff numbers.

Note that data queues without buffers can also be created.

5.4.2 Deleting data queues

A data queue is deleted by issuing the del_dtq service call. When del_dtq is issued, the kernel returns the area of the ring buffer being used by the specified data queue to the user pool and then invalidates the control block and puts the target data queue in the non-existent state.

Even if a task exists that has transmit/receive data for the deleted data queue, that data queue will still be deleted. In this case, all the waiting tasks are released from the waiting state and the error code E_DLT indicating that the data queue has been deleted is returned as the return value of the service call.

After a data queue is deleted, a data queue with the same ID number as the deleted data queue can be newly created.

5.4.3 Receiving data

Data is received from the data queue by issuing one of the service calls rcv_dtq, trcv_dtq, or (i)prcv_dtq.

If there is a data reception request, the kernel checks whether there is unreceived data in the ring buffer being used by the data queue. If unreceived data exists, the task receives the data at the top of the ring buffer and continues processing. The kernel then checks the transmission waiting task queue of the data queue, and if there is a task waiting in this queue, it stores the transmit data of the waiting task in the space in the ring buffer area made available by the previous data reception processing, at which point the task is released from the waiting state. Accordingly, a dispatch occurs when the task that was in the waiting state has a higher priority than the task currently running.

If there is no data stored in the ring buffer, the service calls rcv_dtq and trcv_dtq put the task into a data reception waiting state: until data is received in the case of the former, and until either data is received or a specified time has elapsed in the case of the latter. The issuance of (i)prcv_dtq results in a service call error, and the error code E_TMOUT is returned, indicating that polling failed.

Tasks waiting to receive data are registered in the waiting task queue of the data queue. Because multiple tasks can be waiting in the waiting task queue of the same data queue, a request for data reception sent to a data queue in whose waiting task queue tasks are already waiting will not result in an error; the task will simply be put in the data reception waiting state. Tasks can only be registered in the waiting task queue in FIFO order. Waiting tasks are therefore registered in the queue in the order in which data reception requests were sent, and released in that order when data is transmitted by a service call such as snd_dtq.

5.4.4 Obtaining data queue information

Data queue information such as the number of data not received can be obtained by using the service calls ref_dtq and iref_dtq. For details of each service call, refer to **CHAPTER 13**.

5.4.5 Transmitting data

Data is transmitted to the data queue by issuing one of the service calls snd_dtq, (i)psnd_dtq, tsnd_dtq, or (i)fsnd_dtq.

If there is a data transmission request, the kernel checks the reception waiting task queue of the data queue, and if there is a task waiting in this queue, it assigns data to the task at the top of the queue and releases that task from the waiting state. Although it is possible for the task that transmitted the data to continue processing, if the task just released from the waiting state has a higher priority, then the task currently running is dispatched.

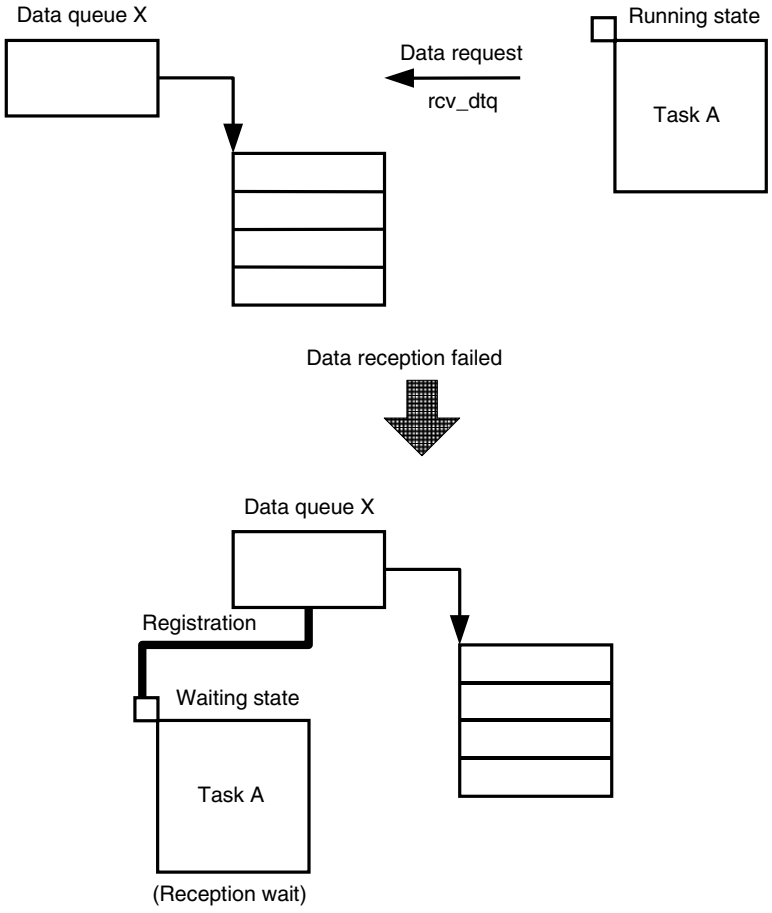
If there are no tasks waiting for reception, the kernel checks the ring buffer of the data queue to see if there is any free space. If there is free space, the transmit data is copied to the buffer and the task continues processing. If there is no free space, the service calls snd_dtq and tsnd_dtq put the task into a data transmission waiting state and that task is registered in the waiting task queue of the data queue until the buffer can be written to. At this time, tasks are registered in the waiting task queue in the order (FIFO or priority order) specified by the attribute assigned when that data queue was created. In the case of tsnd_dtq, the task is also released when a specified time has elapsed. If (i)fsnd_dtq is issued, the oldest data in the ring buffer is discarded and transmission is forcibly carried out. The issuance of (i)psnd_dtq results in an error if the buffer is not empty, and the error code E_TMOUT is returned, indicating that polling failed.

5.4.6 Examples of communication using data queues

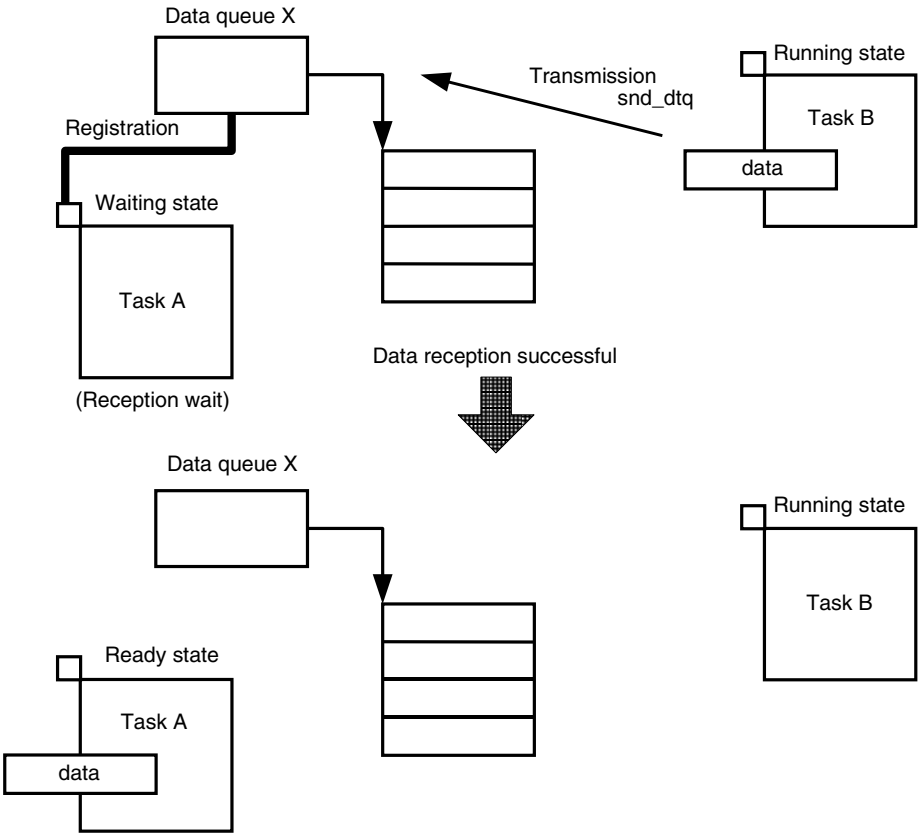
Some examples of intertask communication and synchronization using data queues are shown below.

(1) cre_dtq is issued creating a data queue. There are 4 buffers.

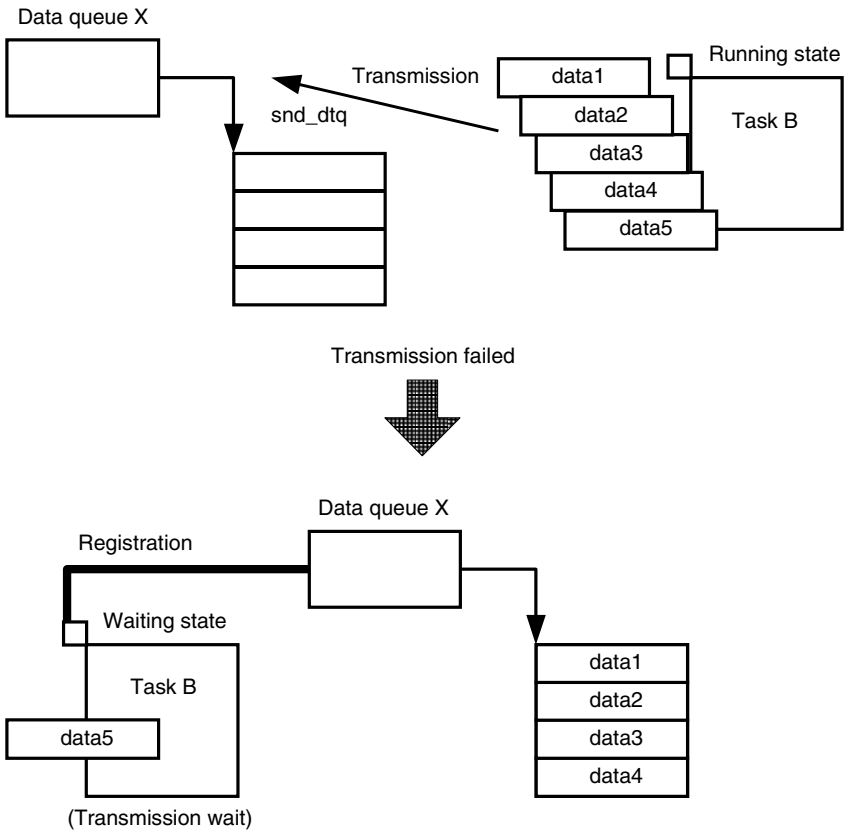
(2) Task A sends a data reception request to data queue X. However, because there is no data stored in the buffers of data queue X, task A is put in the data reception waiting state.



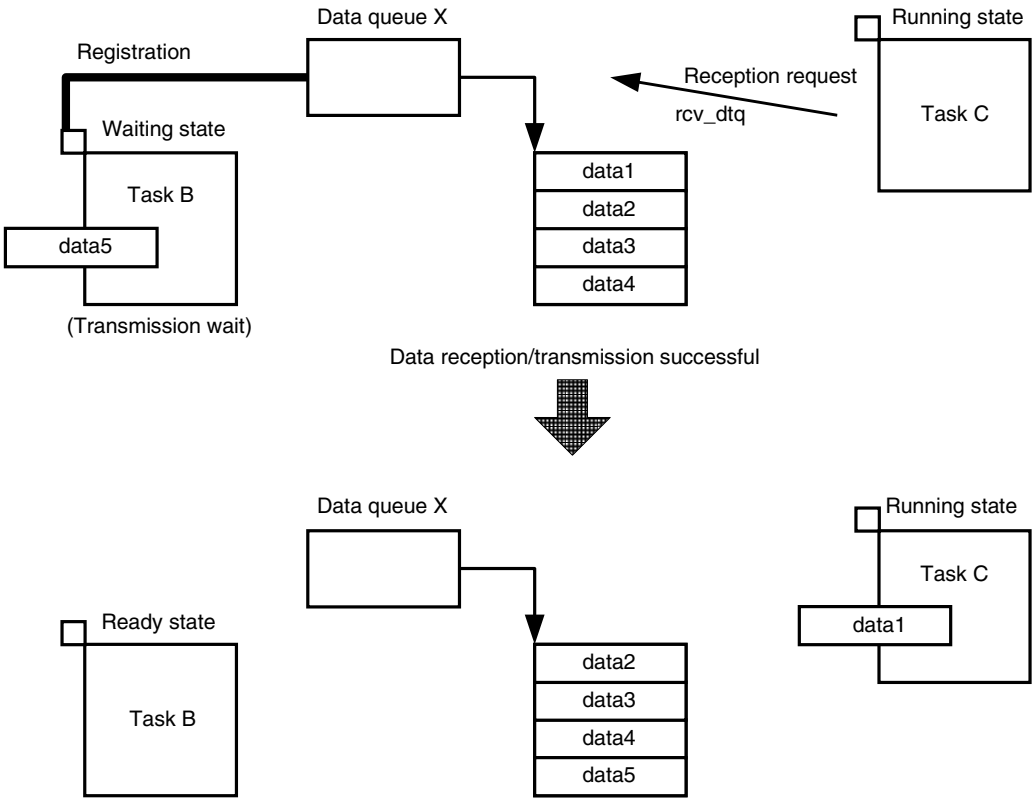
(3) Task B transmits data to data queue X. Task A therefore receives this data and is released from the waiting state.



(4) Following this, task B transmits five items of data in succession. However, because the data queue can only store up to four data items in its buffers, task B fails to transmit the fifth data item and is put in the data transmission waiting state.



(5) Here, task C sends a data reception request, receives the data “data1” from the top buffer, and continues processing. Task B, which is waiting to transmit data, then completes data transmission, and data5 is stored in the ring buffer.



5.5 Mailboxes

In the RX4000, data queues and mailboxes are provided as a means of realizing an intertask communication function. For a data queue, transmit data is fixed to 4 bytes and there is a limit to the total amount of data that can be stored. For a mailbox on the other hand, data of any format can be transmitted, and there is no limit to the total amount of storable data. Users can therefore select and use whichever of these has the features that best suit their system configuration. This section describes mailboxes, which consist of a receive task waiting queue and a transmit message mail queue. Mailboxes not only enable communication between tasks, but also make it possible to have tasks wait in response to messages.

5.5.1 Creating mailboxes

Mailboxes can be created either by issuing the service call (a)cre_mbx, or by specifying the static API CRE_MBX, which performs equivalent processing to (a)cre_mbx when the system is initialized.

When (a)cre_mbx is issued, the attribute, etc., is stored in the block corresponding to the ID number that specifies the mailbox control block array, and that control block is then initialized.

The mailbox ID number consists of a unique number of a value 1 or higher. The maximum value that can be specified is the one defined in the system information table, up to a maximum of 0x7fff numbers.

5.5.2 Deleting mailboxes

A mailbox is deleted by issuing the del_mbx service call. When del_mbx is issued, the kernel invalidates the specified mailbox control block and puts the target mailbox in the non-existent state.

Even if a task exists that has a receive message for the deleted mailbox, that mailbox will still be deleted. In this case, all the waiting tasks are released from the waiting state and the error code E_DLT indicating that the mailbox has been deleted is returned as the return value of the service call. Similarly, even if there is a message that has been registered as waiting for reception, the mailbox will still be deleted. However, in this case, even if the message is a memory block acquired from the memory pool, it will not be returned to the memory pool.

5.5.3 Messages

All items of data (memory) exchanged between tasks via mailboxes are called “messages.” Using a mailbox, an arbitrary task, handler, or processing routine can access the data, or messages, stored in the memory. In the RX4000, however, the address of a message is only passed to the receiving side; the contents of the message are not copied to any other area.

(1) Allocating message areas

Any memory area, such as a variable-length memory block, a fixed-length memory block, or a statically secured area, can be used for messages. However, to transmit messages, an area is required for the kernel to perform message management (message header), for which the top 4 bytes of the area is used (or the top 6 bytes in the case of messages with a specified priority). Accordingly, an area larger than 4 (or 6) bytes must be secured for the message area.

The top address of the message area is passed to the mailbox when a message is transmitted. This address must be a value that is an integral multiple of 4 (an integral multiple of 8 is recommended); otherwise operation cannot be guaranteed.

For further details of the structure of the message header, refer to the **RX4000 (μITRON4.0) Technical User’s Manual (U14835E)**.

(2) Contents of messages

The length and composition of messages to be transmitted to mailboxes are not prescribed in the RX4000; rather they are determined by the tasks, handlers, and processing routines that communicate with each other (i.e., by protocols).

(3) Message priority order

In the RX4000, if there are no tasks waiting for message reception when a message is transmitted, the transmitted message is registered in the message queue of the mailbox. The order in which messages are registered can be specified by the attribute of the mailbox, so for mailboxes with the attribute TA_MPRI, messages are registered in order of priority. Values from 1 to 255 can be used to assign priority: the smaller the value the higher the priority.

The priority of a message is stored in the 2 bytes following the 4th byte from the top of the message area. Therefore when messages are communicated via a mailbox with the attribute TA_MPRI, the essence of the message activates after the 6th byte from the top of the message area.

5.5.4 Transmitting messages

Messages are transmitted by issuing the service call snd_mbx.

If there is a message transmission request, the kernel checks the message reception waiting task queue of the mailbox, and if there is a task waiting in this queue, it releases that task from the waiting state, at which point the task receives the message. If there are no tasks waiting, messages are registered in the message queue of the mailbox in the order specified by the attribute assigned when that mailbox was created (FIFO or priority order), and are saved for the next message reception request.

5.5.5 Receiving messages

Messages are received by issuing one of the service calls rcv_mbx, trcv_mbx, or (i)prcv_mbx.

If there is a message reception request, the kernel checks the message queue of the mailbox. If there are unreceived messages registered in the queue, the message at the top of the queue is released and passed to the task with that address. If there are no messages registered in the queue, when the service calls rcv_mbx and trcv_mbx are issued, the task is put into a mail reception waiting state: until mail is received in the case of the former, and until either mail is received or a specified time has elapsed in the case of the latter. The issuance of (i)prcv_mbx results in message reception failure, and the error code E_TMOUT is returned, indicating that polling failed.

Tasks waiting to receive messages are registered in the waiting task queue of the mailbox. Because multiple tasks can be waiting in the waiting task queue of the same mailbox, a request for message reception sent to a mailbox in whose waiting task queue tasks are already waiting will not result in an error; the task will simply be put in the message reception waiting state.

Tasks can be registered in the waiting task queue of the mailbox in two ways. In the case of mailboxes with an attribute of TA_TPRI, tasks are registered in the queue in order of their respective priorities, and in the case of mailboxes with an attribute of TA_TFIFO, tasks are registered in the order in which message reception requests were sent. Tasks are released from waiting in order activation from the top of the queue.

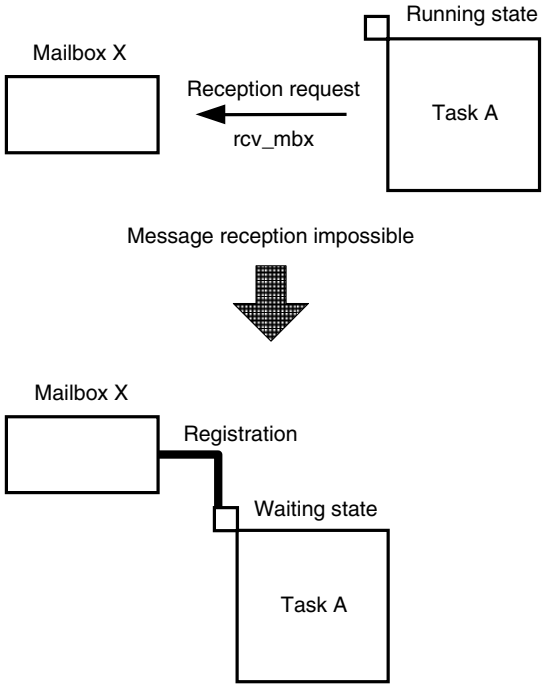
5.5.6 Obtaining mailbox information

Mailbox information such as the presence or absence of a waiting task can be obtained by using the service calls ref_mbx and iref_mbx. For details of each service call, refer to **CHAPTER 13**.

5.5.7 Examples of message communication using mailboxes

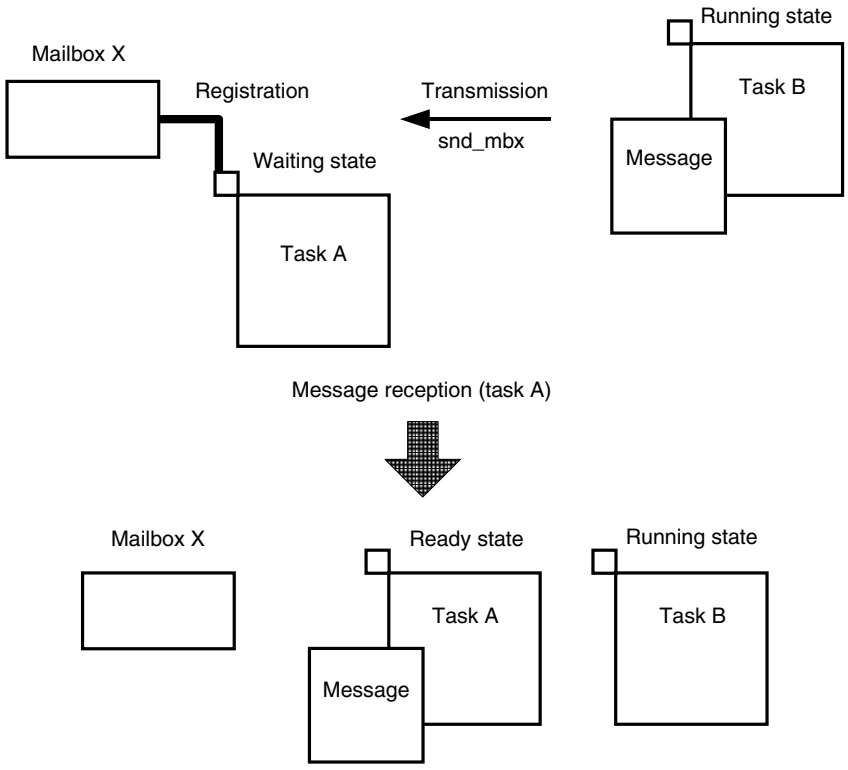
Some examples of communication using mailboxes are shown below.

(1) cre_mbx is issued creating a mailbox.
(2) Task A issues rcv_mbx to receive a message and a message reception request is sent to mailbox X. However, because no messages have been sent to mailbox X, task A is put in the message reception waiting state.

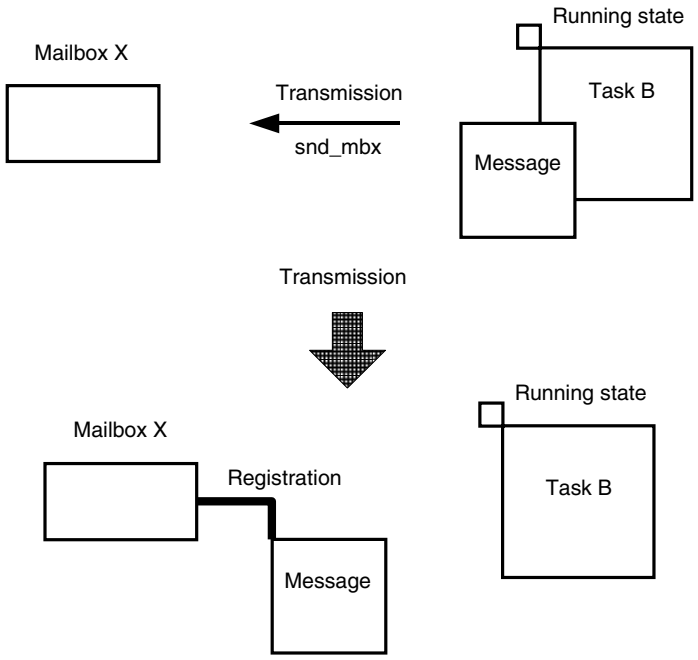


(3) Task B prepares a message for transmission. A memory block or any memory area is used for the message. The first 4 bytes of the message area are reserved for kernel management and therefore cannot store message contents. The following 2 bytes store the message's priority. However, if the destination of the message is a mailbox with the attribute TA_MFIFO, these 2 bytes can be used as message area. Users can store data in any format from the 6th byte of the message area. For further details, refer to the **RX4000 (μITRON4.0) Technical User's Manual (U14835E)**.

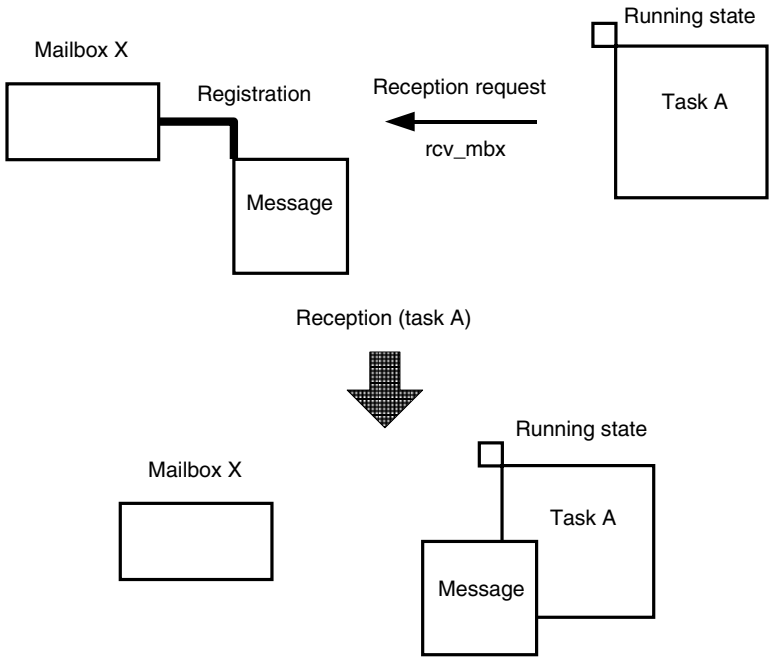
(4) Task B transmits a message to mailbox X, at which point task A receives the message and is released from the waiting state.



(5) Following this, task B transmits another message to mailbox X. However, because there are no longer any tasks in mailbox X waiting to receive messages, the message is registered in the mailbox and waits to be received.



(6) Here, task A sends another reception request to mailbox X. Because there is a receivable message registered in mailbox X, task A immediately receives that message and continues processing.



5.6 Mutexes

Mutexes are objects used to perform exclusive control between tasks when common resources are used, and support the priority inheritance and priority ceiling protocols as mechanisms for preventing the inversion of an unlimited priority order that accompanies exclusive control.

5.6.1 Creating mutexes

Mutexes can be created either by issuing the service call (a)cre_mtx, or by specifying the static API CRE_MTX, which performs equivalent processing to (a)cre_mtx when the system is initialized.

When (a)cre_mtx is issued, the attribute, priority order limit, etc., is stored in the block corresponding to the ID number that specifies the mutex control block array, and that control block is then initialized.

The mutex ID number consists of a unique number of a value 1 or higher. The maximum value that can be specified is the one defined in the system information table, up to a maximum of 0x7fff numbers.

5.6.2 Deleting mutexes

A mutex is deleted by issuing the service call del_mtx. When del_mtx is issued, the kernel invalidates the specified mutex control block and puts the target mutex in the non-existent state.

Even if a task exists that is waiting to lock the mutex to be deleted, that mutex will still be deleted. In this case, all the waiting tasks are released from the waiting state and the error code E_DLT indicating that the mutex has been deleted is returned as the return value of the service call (loc_mtx, tloc_mtx).

Also, even if the target mutex has been locked by a task, the mutex will still be deleted. However, the task locking the mutex will not be informed of its deletion. It is therefore necessary to inform this task of the mutex deletion by performing processing such as issuing the service call unl_mtx to check for the return of the error code E_NOEXS or E_ILUSE. Note also that if the current priority of the task locking the mutex has been raised via the priority inheritance or priority ceiling protocol, the priority may be returned to the base priority by deletion of the locked mutex.

5.6.3 Priority inheritance protocol

The priority inheritance protocol is a mutex priority control protocol used to prevent the inversion of an unlimited priority order. When processing related to a mutex with the TA_INHERIT attribute occurs, this protocol changes the current priority value of the task locking the mutex to whichever is the highest of the following.

- The task's base priority value
- The same priority value as the task with the highest priority among tasks waiting to lock a mutex with the attribute TA_INHERIT

5.6.4 Priority ceiling protocol

The priority ceiling protocol is a mutex priority control protocol used to prevent the inversion of an unlimited priority order. When a task locks a mutex with the attribute TA_CEILING, the priority of that task is changed to the top priority value of the mutexes. Note that a task with a priority higher than this top priority cannot lock a mutex with the attribute TA_CEILING.

5.6.5 Simplified priority order control rules

In the μITRON4.0 Specification, two kinds of priority control rules related to mutexes are prescribed: detailed rules and simplified rules, of which the latter are used in the RX4000.

The priority order of tasks that use mutexes to perform synchronization is therefore as follows.

(When mutex is locked)

Whichever has the highest priority among the following.

- The task's base priority value
- The same priority value as the task with the highest priority among tasks waiting to lock a locked mutex with the attribute TA_INHERIT
- The highest top priority among locked mutexes with the attribute TA_CEILING.

(After mutex lock is released)

When all the locked mutexes are released, a task's current priority changes to its base priority.

5.6.6 Locking mutexes

A mutex is locked by issuing one of the service calls loc_mtx, tloc_mtx, or ploc_mtx.

If there is a mutex lock request, the kernel checks whether that mutex is already locked by another task. If the mutex is not locked by another task, the mutex can be locked by the requesting task. At this time, if the target mutex has the attribute TA_CEILING, the current priority value of the task is raised to the top priority value of the mutexes. If the mutex is locked by another task, the requesting task is put in the waiting state and registered in the waiting task queue of the mutex. The order in which tasks are registered in this queue is FIFO order if the target mutex has the attribute TA_TFIFO, and priority order if the mutex has one of the other attributes TA_TPRI, TA_INHERIT, or TA_CEILING. Also, if the mutex has the attribute TA_INHERIT and the current priority of the waiting task is higher than that of the task locking the mutex, the latter task's priority is changed.

Note that when tloc_mtx is issued, the is released from waiting if the mutex cannot be locked within a specified time. Note also that a mutex cannot be locked by processing units other than tasks.

5.6.7 Releasing mutex locks

A mutex lock is released by issuing the service call unl_mtx.

If a mutex lock is released when there are tasks waiting, the task at the top of the waiting task queue locks that mutex. At this time, if the mutex has the attribute TA_CEILING, the priority value of the task newly locking the mutex is changed to the top priority value of the mutexes. If the mutex has the attribute TA_INHERIT, because the task queue has already been sorted into priority order, the priority of the task newly locking the mutex remains unchanged.

Note that if all mutexes being locked by a task are released, the task's priority changes to its base priority.

5.6.8 Obtaining mutex information

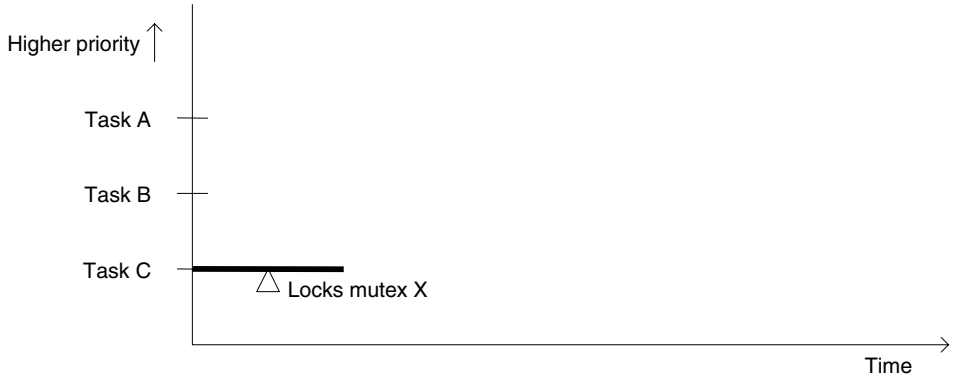
Mutex information such as the ID number of a locked task can be obtained by using the service calls ref_mtx and iref_mtx. For details of each service call, refer to **CHAPTER 13**.

5.6.9 Examples of synchronization using mutexes

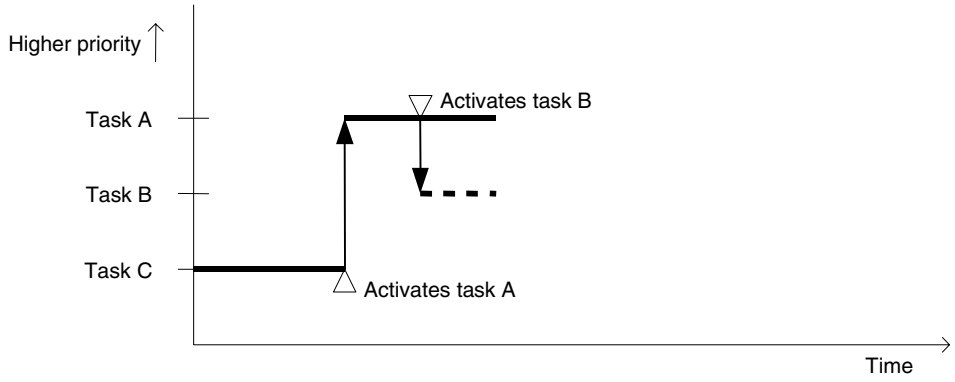
Some examples of mutex priority control using a mutex with the attribute TA_INHERIT are shown below.

(1) cre_mtx is issued creating a mutex.

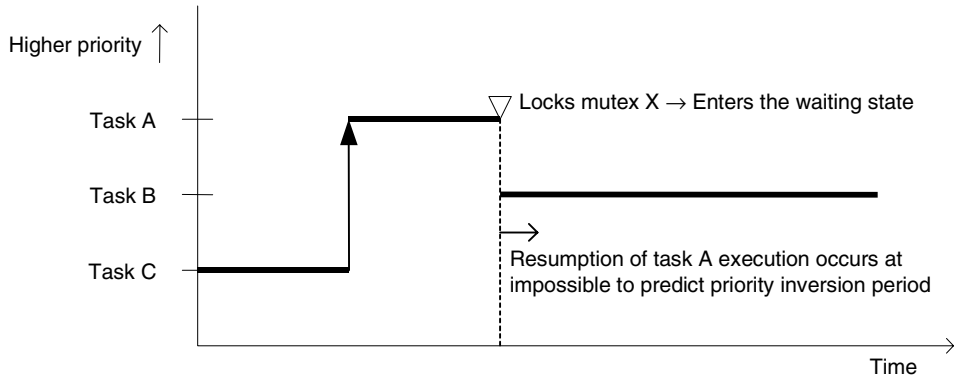
(2) In order to use a resource that needs to be exclusively controlled by a mutex, task C locks mutex X.



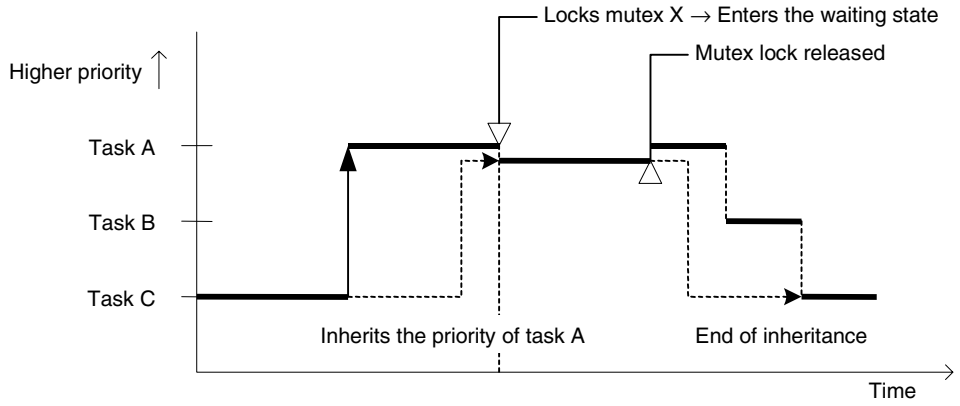
(3) Task C activates task A, which has a higher priority. Task A then activates task B, which has a priority lower than task A but higher than task C.



(4) In order to use a resource, task A attempts to lock mutex X. At this time, if the mutex does not support the priority inheritance protocol, task A will fail to lock the mutex, and enter the waiting state, at which point processing will shift to task B. Because task C cannot be executed until the processing of task B is complete, task A must wait for task B, which has a lower priority than itself, to be executed (despite the fact task B is not locking a mutex).



In actual fact, however, as soon as task A enters the waiting state after failing to lock the mutex, priority inheritance occurs and task C's priority is raised to that of task A. Consequently, because the processing of task C, which is locking the mutex, now has a higher priority than that of task B, the period in which a task with a lower priority than task A is being executed is minimized.



CHAPTER 6 MEMORY MANAGEMENT

This chapter describes memory management in the RX4000.

6.1 Overview

The RX4000 does not support the virtual memory function provided in the target VR Series processors. The kernel must therefore assume that all the programs included in the kernel are operating in 32-bit kernel mode. Also, because TLB-related processing is not performed, the entire program and data area is expected to be located in the range of physical addresses 0x00000000 to 0x1fffffff.

The kernel divides the memory into heap areas known as pools, which it controls, and other areas. The types of pools and types of objects located in pools are shown in the table below.

Note that the size and address of a pool is determined at configuration, and that area is initialized at system initialization.

Table 6-1. Memory Area

Area	Located Objects
System pool	System base table Ready queue Object control block
Stack pool	Stack area for tasks Stack area for interrupts
User pool	Main body of fixed-length memory pool Main body of variable-length memory pool Ring buffer for data queue
Other than above	Program code Area for global variables Heap area controlled by user, etc.

6.2 System Pool

The system base table, ready queue, and object control blocks are located in the system pool. The size of this area is determined by the maximum size data of the objects assigned from the configurator based on the CF definition file. The system pool is created as an area in which dynamic addition and deletion does not occur at system initialization.

A large part of the data directly related to kernel operations is stored in the system pool. Consequently, if the system pool is illegally overwritten and made impossible to reference, the kernel will no longer be able to operate.

6.3 Stack Pool

The stack pool is used to secure the interrupt stack area used by interrupt service routines, etc., and the task stack area provided for each task.

Dynamic creation/deletion of the interrupt stack is assumed not to occur during system operation, and because the stack size is determined according to a description in the CF definition file, the interrupt stack area is statically secured as an undeletable area at kernel initialization.

Sections from the stack pool other than the area for the interrupt stack are initialized as a single large variable-length memory pool controlled by the kernel. When a task is created, a memory block secured from this memory pool is assigned to the task as a stack. When a task is deleted, the stack area is returned to this memory pool.

Note that this memory pool cannot be directly accessed from a task or handler processing program via a service call.

6.4 User Pool

The user pool is used to secure the main body of the fixed-length and variable-length memory pools and the ring buffer area used for the data queue.

The user pool is initialized as a single large variable-length memory pool from which memory blocks are secured in memory pool or ring buffer sized portions when a fixed-length memory pool, variable-length memory pool, or data queue is created. The secured memory blocks are assigned to objects created as the main body of a fixed-length memory pool.

Note that this memory pool (user pool) cannot be directly accessed from a task or handler processing program via a service call.

6.5 Fixed-Length Memory Pools

These are memory pools from which memory blocks can be repeatedly acquired and returned by a task or handler processing program via a service call. Unlike a variable-length memory pool, the size of the memory blocks that can be acquired is fixed for each memory pool, enabling more simple and faster access by the processing.

6.5.1 Creating fixed-length memory pools

A fixed-length memory pool is created by issuing the service call (a)cre_mpf. When acre_mpf is issued, the ID number of the memory pool can be assigned by the kernel. Also, specifying the static API CRE_MPF allows processing equivalent to cre_mpf to be carried out at kernel initialization.

When (a)cre_mpf is issued, an area of the specified size is secured from the user pool and becomes the memory pool main body. If this area is successfully secured, data such as the attribute and the address of the memory pool main body area is stored in the fixed-length memory control block corresponding to the specified ID number, and the area is initialized.

6.5.2 Deleting fixed-length memory pools

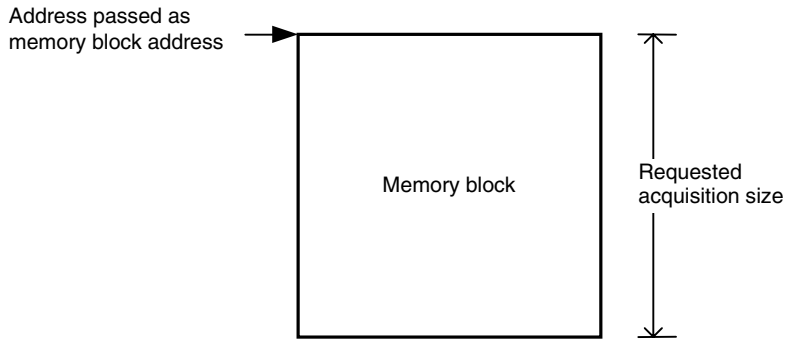
A fixed-length memory pool is deleted by issuing the service call del_mpf. When del_mpf is issued, the kernel returns the area of the memory pool main body to the user pool and then invalidates the control block. At this time, if there is a task waiting to acquire a memory block from the fixed-length memory pool to be deleted, all tasks are released from the waiting state. For tasks released from waiting, the error code E_DLT is returned indicating that the memory pool was deleted.

Note that even if there are parts of memory blocks in the target fixed-length memory pool that have not been returned, the fixed-length memory pool will still be deleted. Care is required at this time because the task or handler that had acquired that memory block is not informed of the deletion of the memory pool.

6.5.3 Fixed-length memory blocks

A fixed-length memory pool is made up of memory blocks all with the same size. The size of the memory blocks in a particular memory pool is specified by the user, and does not include a header area for managing the block. Note that the memory block size must be an integral multiple of 8. If a value that is not an integral multiple of 8 is assigned as the memory block size parameter when a fixed-length memory pool is created, the kernel will automatically round the size to an integral multiple of 8. The structure of a fixed-length memory block is shown below.

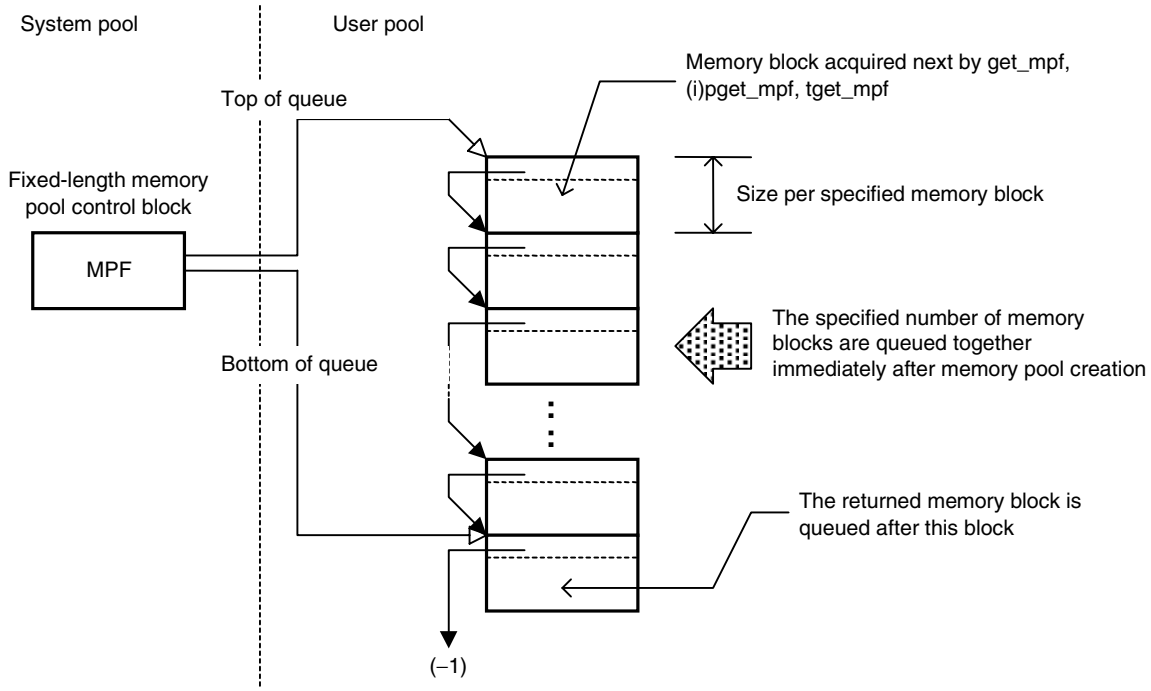
Figure 6-1. Fixed-Length Memory Block



6.5.4 Structure of fixed-length memory pool

A fixed-length memory pool has a structure in which fixed-size memory blocks are joined together in a queue. If there is a request to acquire a memory block, memory blocks are distributed from the top of the queue, and when returned are placed at the bottom of the queue. Therefore memory blocks are not necessarily queued in the order of their addresses, as shown in the figure below.

Figure 6-2. Fixed-Length Memory Pool



6.5.5 Acquiring fixed-length memory blocks

A memory block is acquired from a fixed-length memory pool by issuing one of the service calls get_mpf, (i)pget_mpf, or tget_mpf.

If there is a request to acquire a memory block, the kernel checks the memory block queue of the target memory pool. If the queue has memory blocks queued in it, the task is assigned the memory block at the top of the queue. If there are no queued memory blocks, the task is put in the memory block acquisition waiting state, and waits until a memory block has been acquired, in the case of get_mpf, or until either a memory block has been acquired or a specified time has elapsed, in the case of tget_mpf. The issuance of (i)pget_mpf results in an error, and the error code E_TMOUT is returned, indicating that polling failed.

Tasks in the memory block acquisition waiting state are registered in the waiting task queue of the memory pool. The waiting order at this time depends on the attribute of the memory pool: in FIFO order if the attribute is TA_TFIFO or in priority order if the attribute is TA_TPRI, and if (i)rel_mpf is issued, memory blocks are acquired in the order they are queued, and the task is released from waiting.

Note that because 0 is stored in the top four bytes of the acquired memory block, when this memory block is transmitted to a mailbox as a message, because no value other than 0 is stored in this area (which is used by the kernel to manage the message), it is possible to check whether a memory block has been registered in the mailbox by checking the top four bytes.

6.5.6 Returning fixed-length memory blocks

A memory block is returned to a fixed-length memory pool by issuing the service call (i)rel_mpf.

If there is a request to return a memory block, the kernel checks the waiting task queue of the target memory pool, and if there are tasks registered in the queue, it immediately assigns the task at the top of the queue the returned memory block and releases it from the waiting state. If there are no tasks registered in the waiting queue, the memory block is returned to the bottom of the memory pool's free memory block queue.

Note that memory blocks must be returned to the same memory pool from which they were acquired. If a different memory pool is specified and (i)rel_mpf is issued, operation cannot be guaranteed. Neither can operation be guaranteed if areas other than fixed-length memory blocks, such as variable-length memory blocks or statically secured areas, are registered together in a mailbox as messages and (i)rel_mpf is issued.

6.5.7 Obtaining fixed-length memory pool information

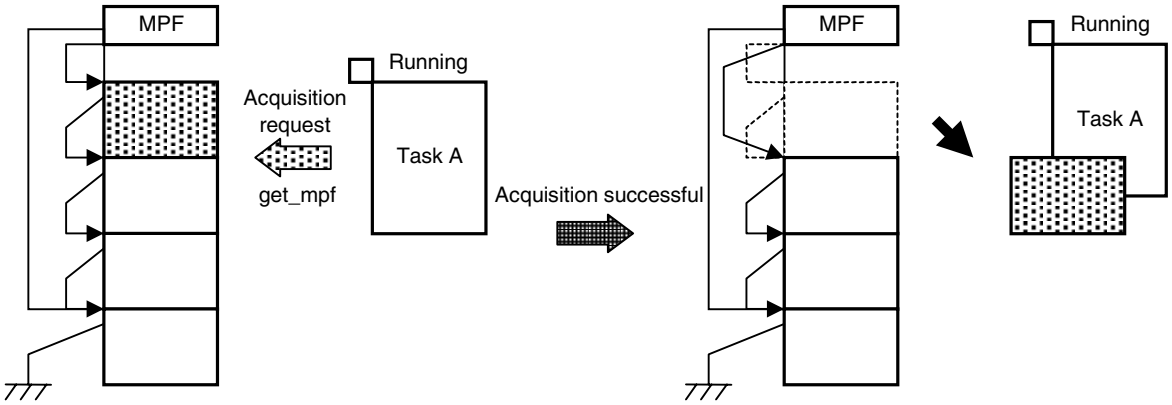
Fixed-length memory pool information such as the presence or absence of a waiting task can be obtained by using the service calls ref_mpf and iref_mpf. For details of each service call, refer to **CHAPTER 13**.

6.5.8 Examples of acquiring memory blocks from fixed-length memory pools

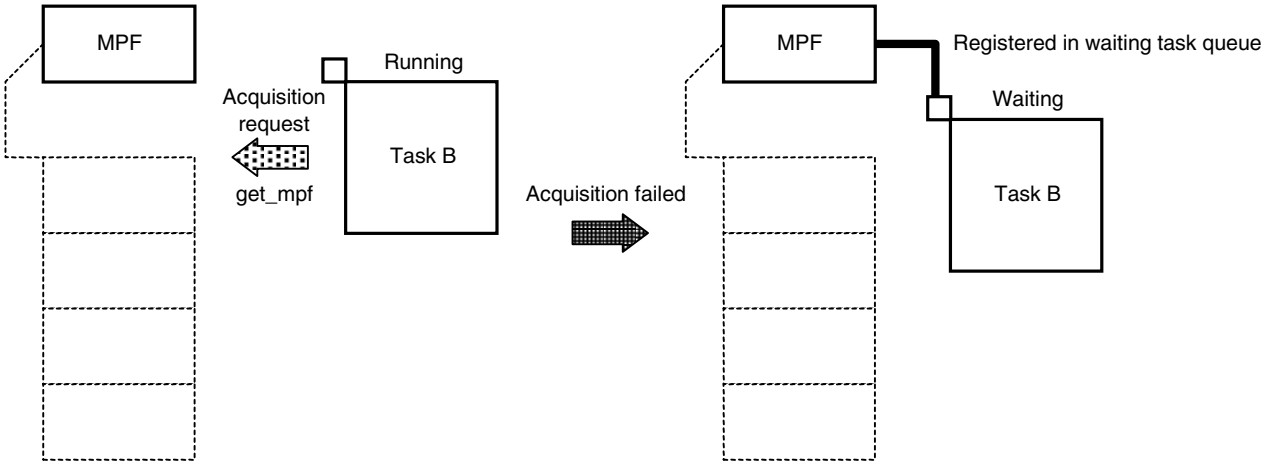
Some examples of what occurs when memory blocks are acquired from fixed-length memory pools are shown below.

(1) cre_mpf is issued creating a fixed-length memory pool. This pool contains 4 memory blocks.

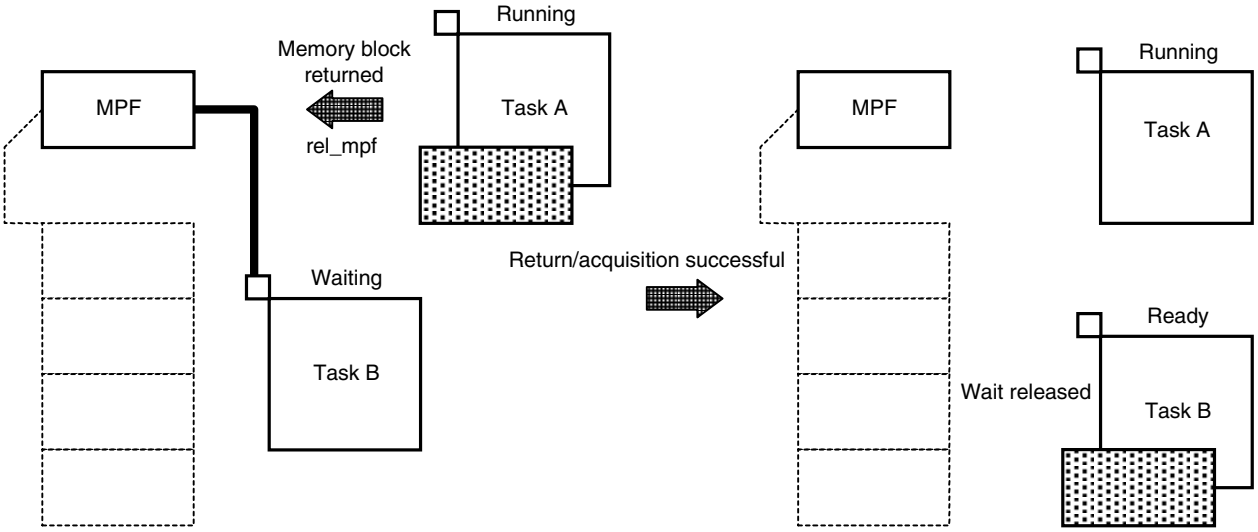
(2) Task A issues get_mpf and requests acquisition of a memory block from MPF. Because there are memory blocks queued in MPF's memory block queue, task A immediately acquires one memory block.



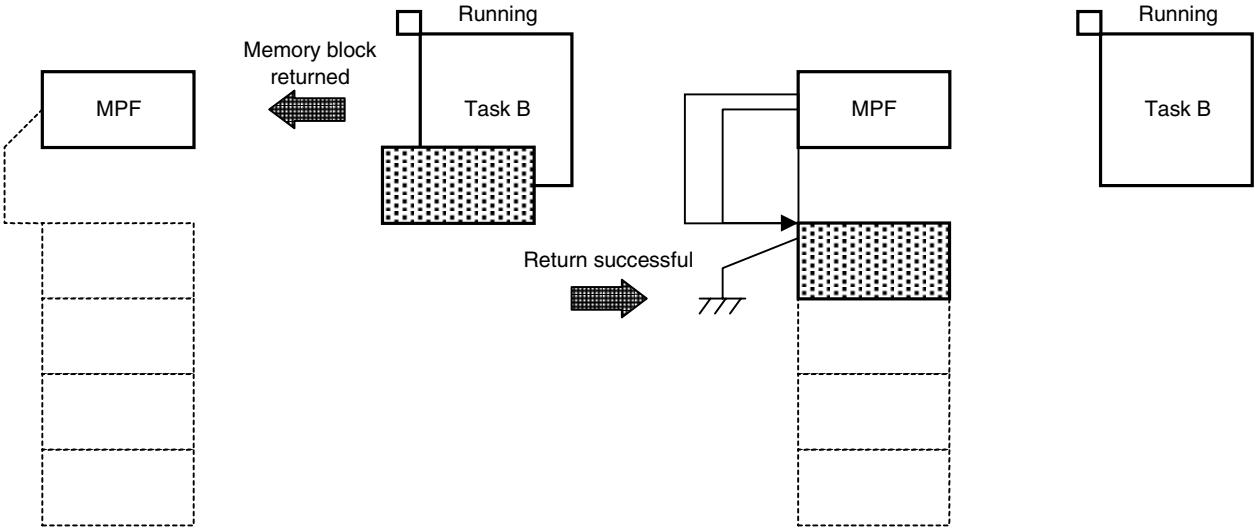
(3) Task B issues get_mpf and requests acquisition of a memory block from MPF. However, because other tasks have in the meantime acquired memory blocks from MPF leaving no available memory blocks, task B fails to acquire a memory block and is put is the waiting state.



(4) Task A returns a memory block to MPF. The returned memory block is immediately acquired by task B, which is then released from the waiting state.



(5) Task B has finished using the memory block, issues rel_mpf and returns the memory block to MPF. Because there are no tasks waiting for a memory block in MPF, the memory block joins the MPF's memory block queue.



6.6 Variable-Length Memory Pools

Like fixed-length memory pools, these are memory pools from which memory blocks can be repeatedly acquired and returned by a task or handler processing program via a service call. Unlike a fixed-length memory pool, however, the size of the memory blocks that can be acquired can be specified when an acquisition request is sent.

6.6.1 Creating variable-length memory pools

A variable-length memory pool is created by issuing the service call (a)cre_mpl. When acre_mpl is issued, the ID number of the memory pool can be assigned by the kernel. Also, specifying the static API CRE_MPL allows processing equivalent to cre_mpl to be carried out at kernel initialization.

When (a)cre_mpl is issued, the kernel secures an area of the specified size from the user pool and makes it the memory pool main body. If this area is successfully secured, data such as the attribute and the address of the memory pool main body area is stored in the variable-length memory control block corresponding to the specified ID number, and the control block is initialized.

6.6.2 Deleting variable-length memory pools

A variable-length memory pool is deleted by issuing the service call del_mpl. When del_mpl is issued, the kernel returns the area of the memory pool main body to the user pool and then invalidates the control block. At this time, if there is a task waiting to acquire a memory block from the variable-length memory pool to be deleted, all tasks are released from the waiting state. For tasks released from waiting, the error code E_DLT is returned indicating that the memory pool was deleted.

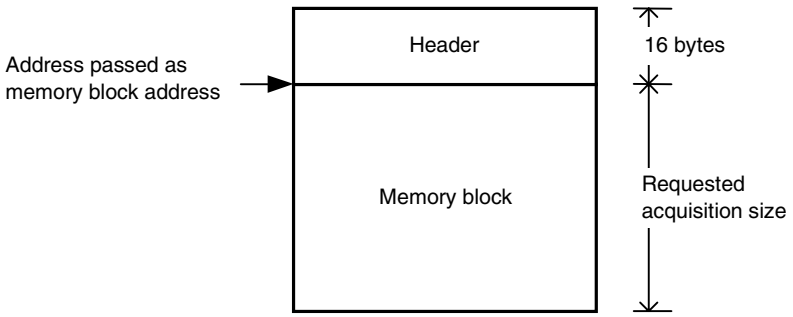
Note that even if there are parts of memory blocks in the target variable-length memory pool that have not been returned, the variable-length memory pool will still be deleted. Care is required at this time because the task or handler that had acquired that memory block is not informed of the deletion of the memory pool.

6.6.3 Variable-length memory blocks

A variable-length memory pool is made up of memory blocks without fixed sizes. Also, to maintain a state in which the largest possible blocks can be acquired, processing is performed to join together memory blocks to form connected areas upon return. A header for block management (16 bytes) is therefore included in the memory block, making the size of the area actually removed from memory pool when a memory block is acquired larger than the specified size. Note that the specified memory block size must be an integral multiple of 8. If a value that is not an integral multiple of 8 is specified when a memory block is acquired, the kernel will automatically round the size to an integral multiple of 8.

The structure of a variable-length memory block is shown below.

Figure 6-3. Variable-Length Memory Block



6.6.4 Structure of variable-length memory pool

Immediately after creation, a variable-length memory pool consists of a single large memory block. If memory blocks are repeatedly acquired, this block becomes gradually smaller, and if acquisition and return repeatedly occur, the area of the memory pool may become divided into islands. For further details, refer to the **RX4000 (μITRON4.0) Technical User's Manual (U14835E)**.

Figure 6-4. Variable-Length Memory Pool Structure (Immediately After Creation)

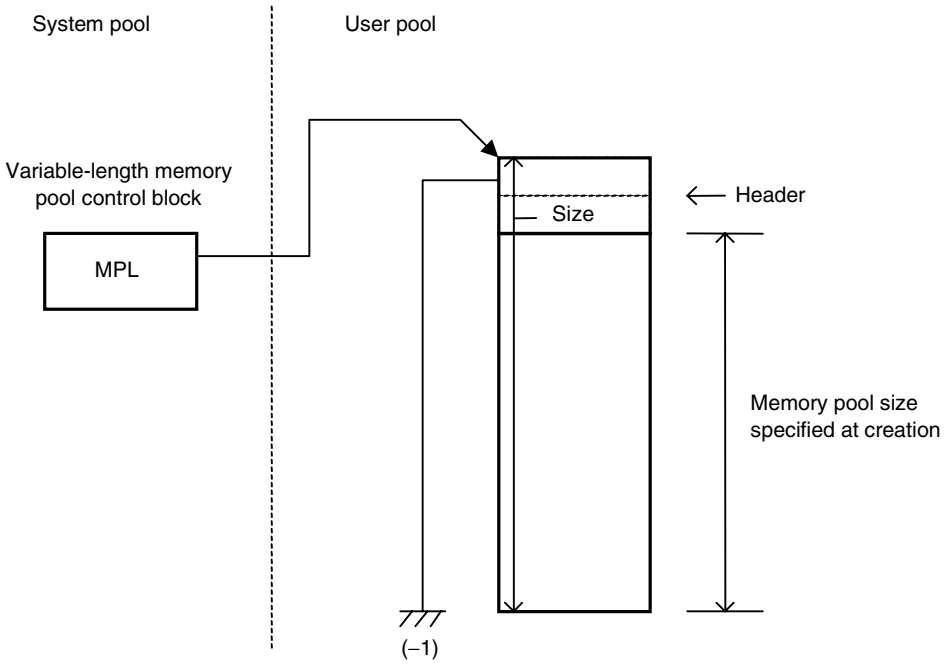


Figure 6-5. Variable-Length Memory Pool Structure (After Memory Block Acquisition)

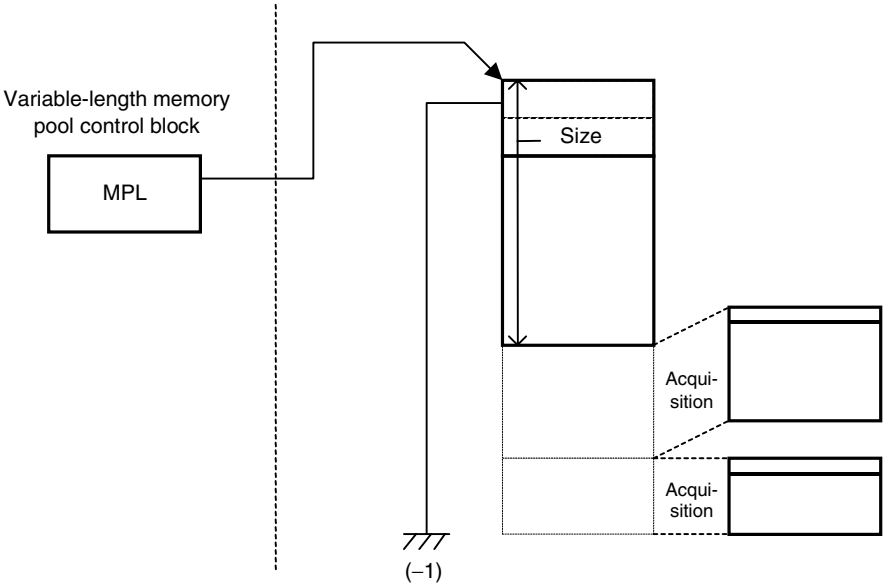
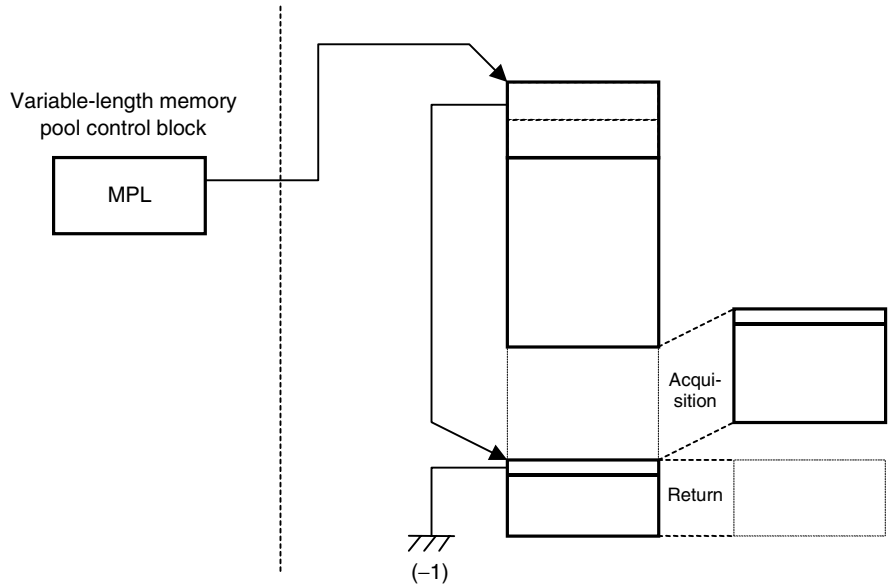


Figure 6-6. Variable-Length Memory Pool Structure (When Area Divided into Islands)

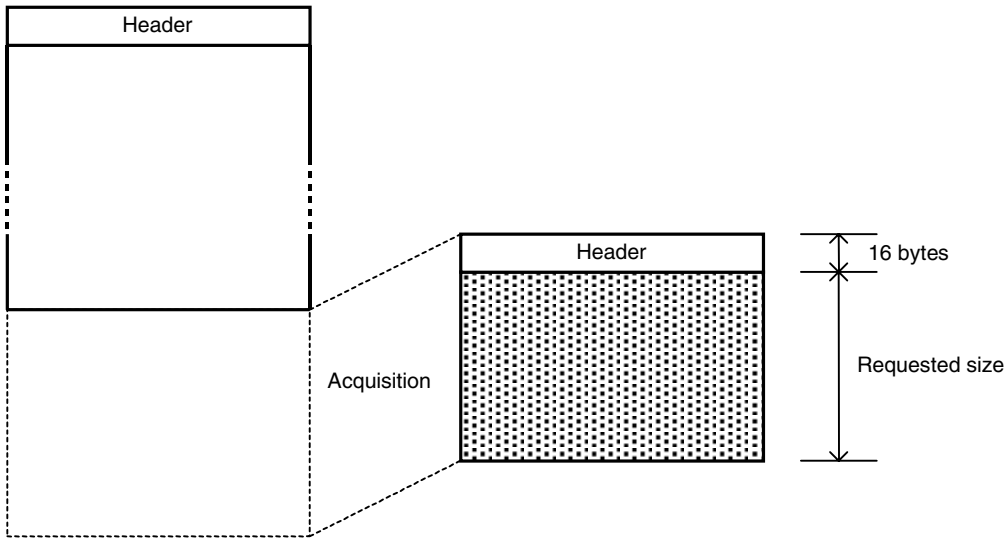


6.6.5 Acquiring variable-length memory blocks

A memory block is acquired from a variable-length memory pool by issuing one of the service calls get_mpl, (i)pget_mpl, or tget_mpl.

If there is a request to acquire a memory block, the kernel checks the memory block queue of the target memory pool. If the queue contains a memory space large enough for acquisition, a memory block of the requested size is removed from the memory pool and assigned to the task. Note, however, that because a header for block management is added to the memory block, the size of the area actually removed from memory pool when a memory block is acquired is larger than the requested size (see **Figure 6-7**).

Figure 6-7. Memory Block Acquisition



If the queue does not contain a memory space large enough for acquisition, the task is put in the memory block acquisition waiting state, and waits until a memory block has been acquired, in the case of get_mpl or until either a memory block has been acquired or a specified time has elapsed, in the case of tget_mpl. The issuance of (i)pget_mpl results in an error, and the error code E_TMOUT is returned, indicating that polling failed.

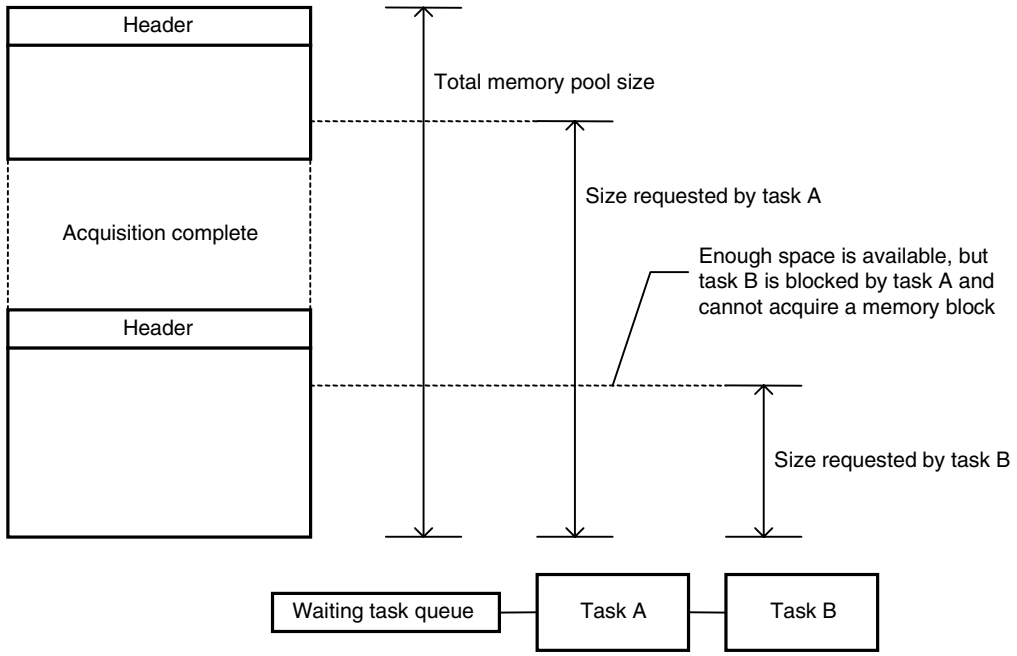
Tasks in the memory block acquisition waiting state are registered in the waiting task queue of the memory pool. The waiting order at this time depends on the attribute of the memory pool: in FIFO order if the attribute is TA_TFIFO or in priority order if the attribute is TA_TPRI, and if (i)rel_mpl is issued and there is sufficient memory space in the memory pool, memory blocks are acquired in the order they are queued, and the task is released from waiting.

Note that because 0 is stored in the top four bytes of the acquired memory block, when this memory block is transmitted to a mailbox as a message, because no value other than 0 is stored in this area (which is used by the kernel to manage the message), it is possible to check whether a memory block has been registered in the mailbox by checking the top four bytes.

Note also that when there are multiple tasks simultaneously waiting to acquire a memory block from the same memory pool, if one of those tasks has requested a large-sized memory block, because the memory block acquisition feasibility check is performed in the order in which the tasks are waiting in the queue, a later task may be blocked and forced to wait even if there is sufficient memory space for it to acquire the requested memory block (see **Figure 6-8**).

In other words, if the memory pool has the attribute TA_TFIFO and there are tasks already waiting, a task coming later will be unable to acquire a memory block, regardless of the requested size, and will be forced to wait, or polling will fail. If there are tasks waiting in the queue of a memory pool with the attribute TA_TPRI, memory block acquisition will only be possible if a memory block of the requested size is available and if the priority of the requesting task is higher than that of the task at the top of the queue. If ipget_mpl is issued for a memory pool with the attribute TA_TPRI from an interrupt servicing routine, because the interrupt processing has a higher priority than the task processing, only the check to ascertain whether an acquirable block is available will be made.

Figure 6-8. Case When Task Is Waiting Even Though Sufficient Area Is Available



6.6.6 Returning variable-length memory blocks

Memory blocks are returned to variable-length memory pools by issuing the service call (i)rel_mpl.

If (i)rel_mpl is issued, the memory block is immediately returned to the memory pool, regardless of the existence of waiting tasks. When a memory block is returned, the kernel checks whether there are any free memory blocks in the area to which the returned memory block is connected, and if there are, performs processing to link two or three memory blocks together to form a single large block (see **Figures 6-9**, **6-10**, and **6-11**).

Figure 6-9. Memory Block Linking 1

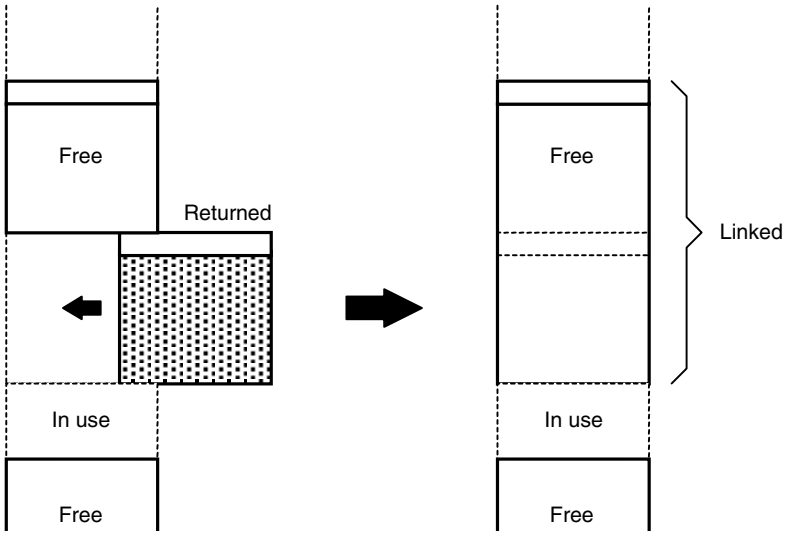


Figure 6-10. Memory Block Linking 2

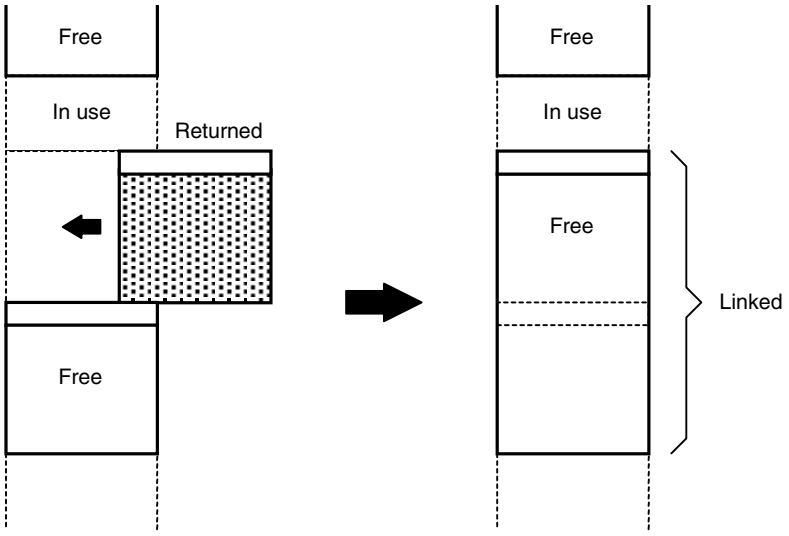
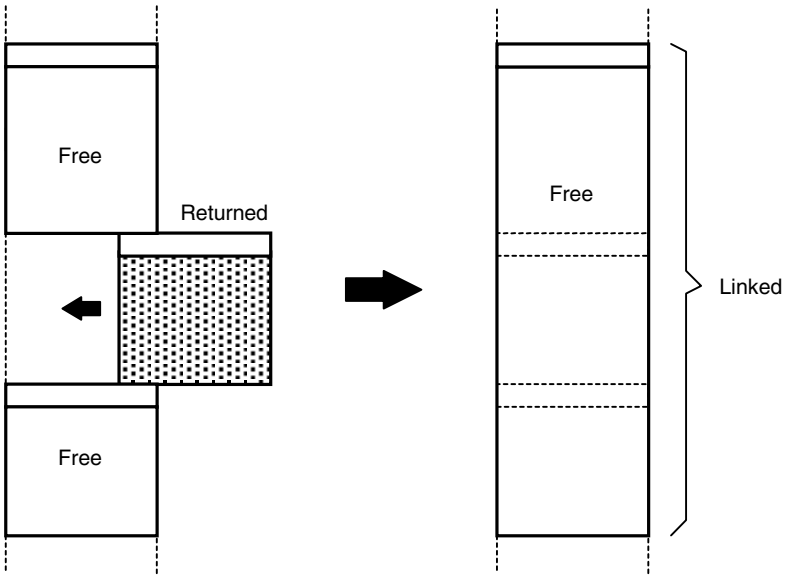


Figure 6-11. Memory Block Linking 3



When the memory block return processing has finished, the memory pool's waiting task queue is checked. If there are tasks waiting, a check is made whether a memory block of the size requested by the task at the top of the queue can be acquired. If it can, an appropriate memory block is assigned to the task and the task is removed from the waiting queue and released from the waiting state. This processing is then repeated until the bottom of the queue is reached or a task is unable to acquire a memory block.

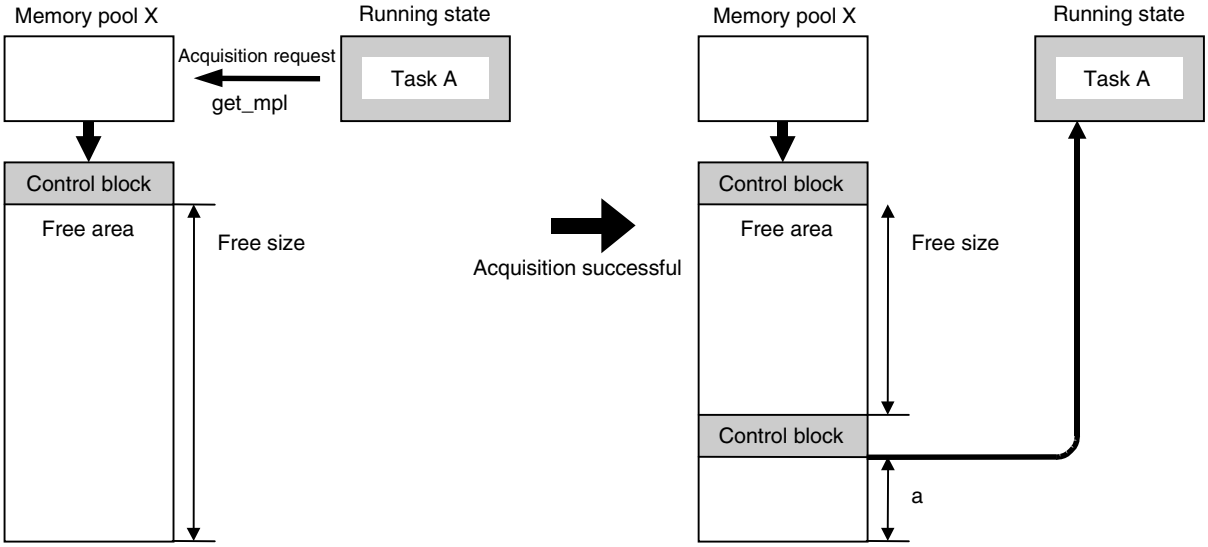
6.6.7 Obtaining variable-length memory pool information

Variable-length memory pool information such as the presence or absence of a waiting task can be obtained by using the service calls ref_mpl and iref_mpl. For details of each service call, refer to **CHAPTER 13**.

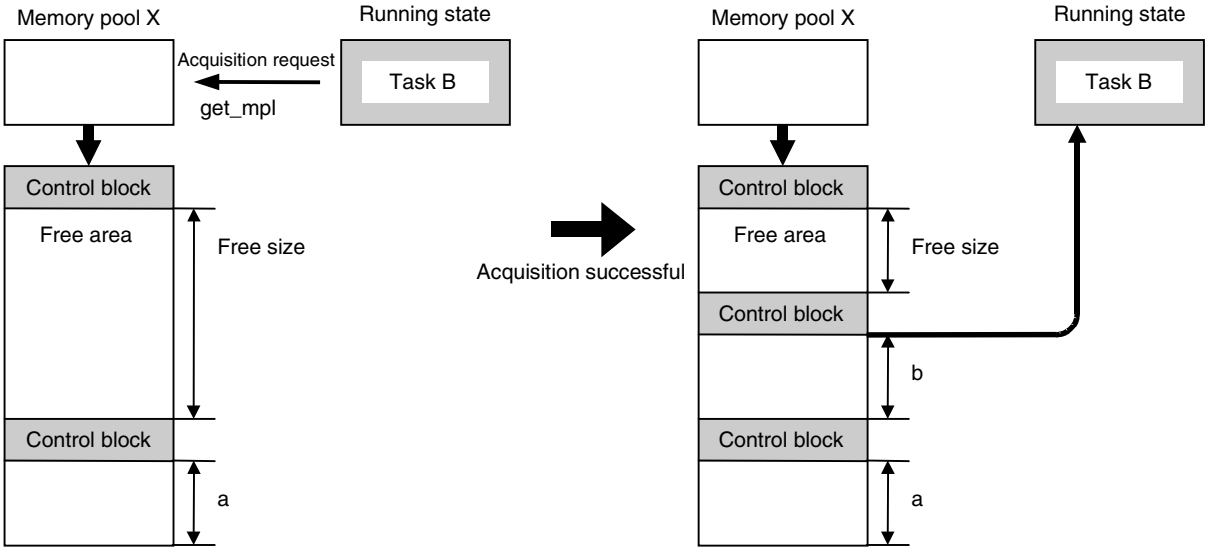
6.6.8 Examples of acquiring memory blocks from variable-length memory pools

Some examples of what occurs when memory blocks are acquired from variable-length memory pools are shown below.

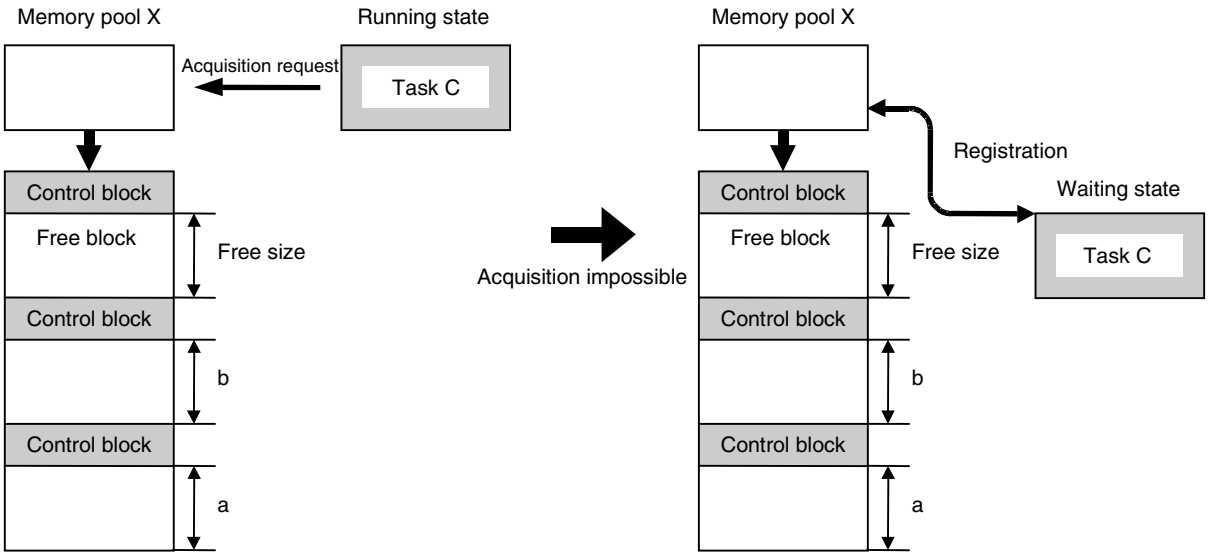
(1) Task A issues get_mpl and acquires a memory block of size a from the memory pool.



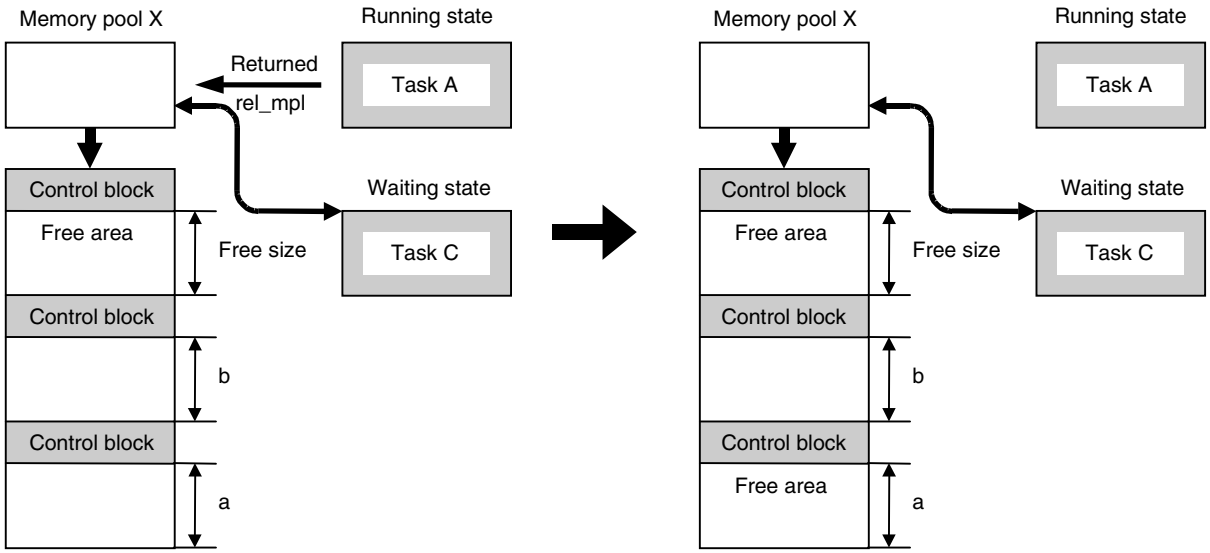
(2) Task B issues get_mpl and acquires a memory block of size b from the memory pool.



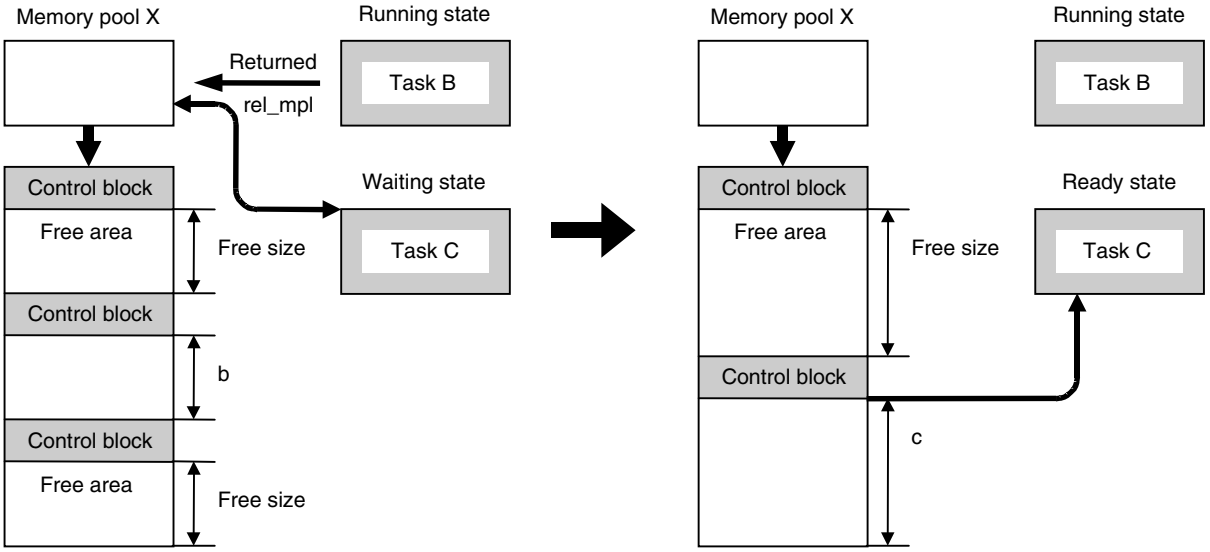
(3) Task C issues get_mpl and requests acquisition of a memory block of size c from the memory pool. However, because the free area is smaller than size c, task C is unable to acquire the requested memory block and is put into the waiting state.



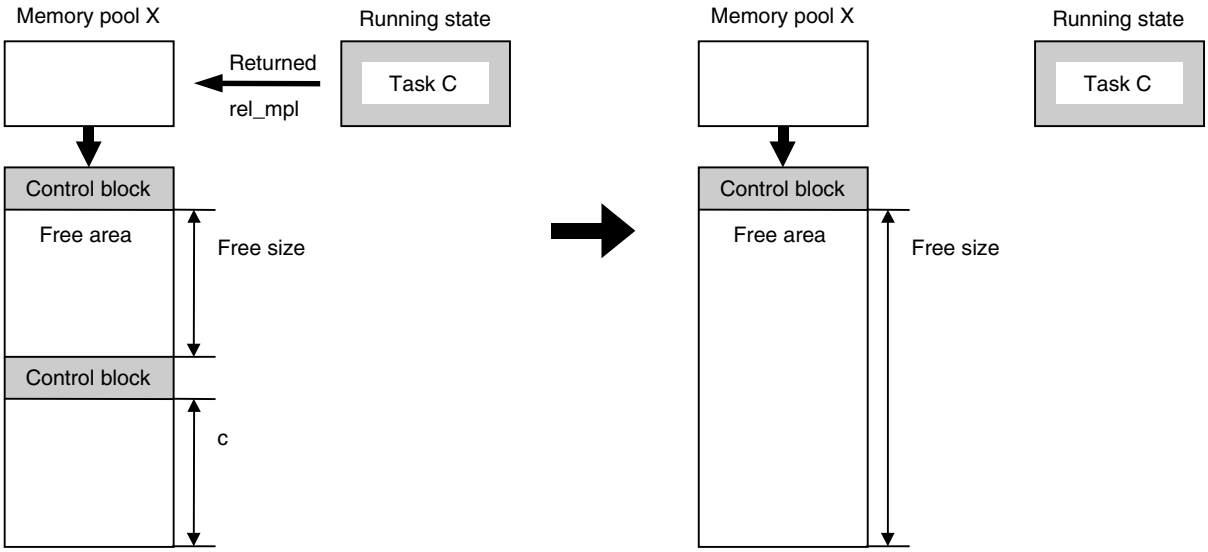
(4) Task A issues rel_mpl and returns a memory block (of size a) to the memory pool. The total free size of the memory pool increases accordingly, but because it is divided into two small memory blocks, the connected area requested by task C cannot be secured. Task C is therefore put into the waiting state.



(5) Task B issues rel_mpl and returns a memory block (of size b) to the memory pool. Accordingly, small free memory blocks are linked together to form a single large memory block. The connected area requested by task C can therefore be secured, so task C is assigned the requested memory block and released from the waiting state.



(6) Task C issues rel_mpl and returns a memory block (of size c) to the memory pool. Accordingly, all the memory blocks have been returned, and the memory pool returns to the state it was in at creation.



CHAPTER 7 TIME MANAGEMENT

This chapter describes time management in the RX4000.

7.1 Overview

In a real-time system, time-synchronization processing is often used when processing tasks and handlers. In the RX4000, a software timer is used to trigger timer interrupts issued at periodic intervals, and system clock, task delay and timeout, and cyclic handler functions are provided.

7.2 System Clock

The system clock indicates the time (system time) upon which systems that incorporate the RX4000 are based. The system clock consists of a 64-bit counter and is set to 0 as soon as the kernel has completed initialization. The system clock is reset to 0 when the passage of time causes the 64-bit range of its counter to be exceeded (overflow is ignored). The system clock operates in units of milliseconds.

7.2.1 Setting the system clock

The system clock can be set to any value during system operation by issuing the service call (i)set_tim. Note, however, that the difference in time between the previous and new system clock values will not be treated has having elapsed, leaving task timeouts or cyclic handler activation unaffected by a new system clock setting.

7.2.2 Reading the system clock

The system clock can be read by issuing the service call (i)get_tim. The system clock reading is stored in the structure SYSTIM expressed by a 64-bit value.

7.2.3 Updating the system clock

The system clock can be updated by issuing the service call isig_tim. If isig_tim is issued, the kernel recognizes that the unit time defined in the CF definition file has elapsed.

In other words, isig_tim must be issued each time the unit time defined in the CF files elapses, making it necessary to describe an interrupt service routine that will input the CP0 timer interrupts, etc., of the processor, from which isig_tim is issued. Processing such as for task timeout or cyclic handler activation is not activated by the issuance of isig_tim; it is activated when the aforementioned interrupt service routine finishes processing.

Note that if the interval at which isig_tim is issued is undefined, the accuracy of the time until task timeout or the cyclic handler's activation interval cannot be guaranteed.

7.3 Delaying Tasks

The execution of a task can be delayed by issuing the service call dly_tsk. If dly_tsk is issued, the task is put in the waiting state and waits until the specified time has elapsed. It is therefore possible to delay the execution of a task for a specified time. The delay time is specified in units of milliseconds, not in number of interrupt ticks.

7.4 Timeout

When a service call is issued that might put a task into a waiting state, such as waiting for wakeup or to acquire a resource, it is possible to set the upper limit of this waiting time. The upper limit is known as the timeout time, and when a task is released from waiting following the elapse of the timeout time, the task is said to have “timed out”. When a task has timed out, the error code E_TMOUT is returned as the return value of the service call that caused the task to wait, indicating that a timeout occurred. The timeout time is specified in units of milliseconds, not in number of interrupt ticks.

Note that the relationship between the timeout time and the time until the task actually times out differs from that specified in μITRON3.0. That is, because the kernel’s time management involves causing (discrete) interrupts to be input at a fixed interval, there is a slight difference between the timeout time and the time that actually elapses. Unlike μITRON3.0 (RX4000 Ver3.0), in which control was performed to prevent the time that actually elapses exceeding the specified timeout time, in μITRON4.0, because the timeout time is taken as the “minimum time that should elapse”, timeout-specified tasks in the waiting state are timed out at the occurrence of the first time event (isig_tim) following the elapse of the specified time, so that the time that actually elapses is never less than the timeout time.

The service calls that can make a timeout specification are shown in the table below.

Table 7-1. Service calls That Can Specify Timeout

Service call	Function
tslp_tsk	Waiting for reception of wakeup request
twai_sem	Acquiring a resource from a semaphore
twai_flg	Waiting for establishment of an event flag condition
tsnd_dtq	Transmitting data to a data queue
trcv_dtq	Receiving data from a data queue
trcv_mbx	Receiving mail from a mailbox
tloc_mtx	Locking a mutex
tget_mpf	Acquiring a memory block from a fixed-length memory pool
tget_mpl	Acquiring a memory block from a variable-length memory pool

7.5 Cyclic Handlers

Cyclic handlers are handlers used to execute specific processing at a fixed time interval.

Note that this type of handler was known as a cyclically activated handler in μITRON3.0. The following names and terminology have accordingly been changed in the μITRON4.0.

Table 7-2. Cyclic Handler Terminology Correspondence Table

μITRON3.0	μITRON4.0
Cyclically activated handler	Cyclic handler
Define a cyclically activated handler	Create a cyclic handler
Activity state	State
TCY_ON (ON)	TCYC_STA (STA)
TCY_OFF (OFF)	TCYC_STP (STP)
def_cyc	cre_cyc

Note that a cyclic handler is described as a void type function with a VP_INT type argument. The extended data exinf of an activated cyclic handler is passed for this argument.

Example)

```
void cyclic(VP_INT exinf)
{
        ...
        return;
}
```

7.5.1 Creating cyclic handlers

Cyclic handlers are created by issuing the service call (a)cre_cyc. When acre_cyc is issued, the ID number of the cyclic handler can be assigned by the kernel. Also, specifying the static API CRE_CYC allows processing equivalent to cre_cyc to be carried out at kernel initialization. When cre_cyc is issued, the cyclic handler control block in the system pool is secured, and initialized based on the cyclic handler creation data passed as a parameter.

The cyclic handler ID number consists of a unique number of a value 1 or higher. The maximum value that can be specified is the one defined in the system information table.

7.5.2 Deleting cyclic handlers

Cyclic handlers are deleted by issuing the service call del_cyc. If del_cyc is issued, the control block of the target cyclic handler is invalidated, and a cyclic handler with the same ID number as the deleted cyclic handler can be newly created.

7.5.3 Cyclic handler states

Cyclic handlers have two states: operating (TCYC_STA) and stopped (TCYC_STP). In the former state, the cyclic handler is activated after the time that passes when the system clock is updated is counted and the specified activation interval time has elapsed. In the latter state, only the time is counted, and the handler is not activated, even after the specified activation interval time has elapsed.

The initial state following creation of a cyclic handler depends on the specification of the attribute TA_STA (when specified, the handler is created in the operating state). The state of a cyclic handler can be changed after creation by issuing the service call (i)sta_cyc or (i)stp_cyc.

7.5.4 Activating a cyclic handler

A cyclic handler is activated as a subroutine called from the kernel when it performs interrupt end processing at the completion of an interrupt service routine for which isig_tim was issued. The stack when the cyclic handler is executed is therefore used as the interrupt stack.

Note that if the next cycle time elapses while a cyclic handler is being executed, the cyclic handler is reactivated once the current processing has finished. However, because this kind of state raises the cyclic handler's CPU occupancy rate to an extremely high level, it is known as an abnormal state.

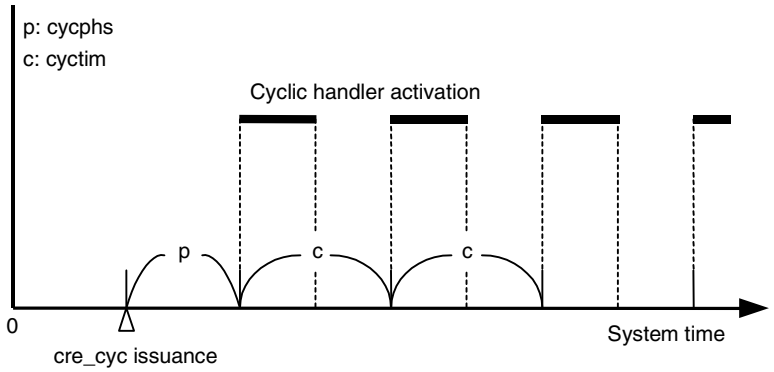
7.5.5 Terminating cyclic handlers

A cyclic handler is terminated by C language return or equivalent processing. As soon as a cyclic handler is terminated, the next cyclic handler for which the activation cycle time has elapsed is activated, or processing is returned from interrupt processing.

7.5.6 Cyclic handler phase

A cyclic handler can be made to have a phase as a means of regulating its activation timing. The cyclic handler phase is the time from cyclic handler creation to first activation, and is specified independently of the cyclic handler's activation interval.

Figure 7-1. Cyclic Handler Phase



It is also possible to save this phase using the cyclic handler attribute TA_PHS. Saving a phase involves reckoning the time from when a cyclic handler is created to when the next cyclic handler is activated [phase + (activation cycle × n)] when a cyclic handler is operating due to the issuance of the service call (i)sta_cyc. If phase saving is not performed, the elapse of one cycle time reckoned from the issuance of (i)sta_cyc becomes the next activation time.

Figure 7-2. Phase Saving

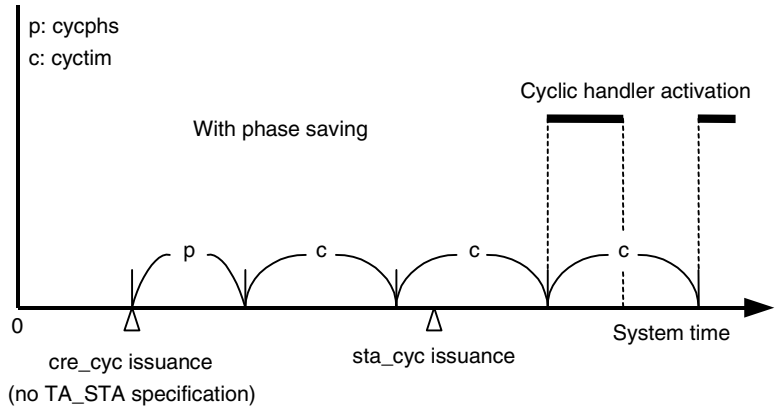
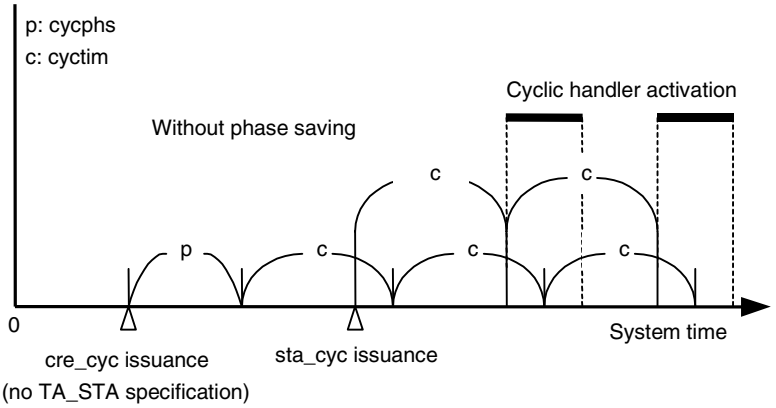


Figure 7-3. No Phase Saving



7.5.7 Interrupts during cyclic handler execution

The cyclic handler operates with interrupts managed by the kernel (interrupts for which it is possible to perform processing in which a kernel service is received, such as interrupt service routine activation) masked. Therefore, to enable these interrupts during cyclic handler execution, the user must explicitly set interrupts to enabled (by setting the IM bit of the status register).

Note that because the cyclic handler processing and timer interrupt processing are independent of each other, it is possible to acknowledge timer interrupts even during cyclic handler execution by enabling interrupts.

7.5.8 PID

If the code created when the cyclic handler program was compiled or assembled uses PID (Position Independent Data), the address that is to be the base of a cyclic handler when it is activated must be assigned as a parameter (gp of the cyclic handler creation packet T_CCYC). The assigned address is set in the gp register when the cyclic handler is activated.

Note that this base address is unconditionally set in the gp register regardless of its attribute, etc., and therefore must be set for programs that reference the gp register. If the gp register is not referenced, set NULL = 0.

7.5.9 Use of coprocessor in cyclic handler

To use the coprocessor in a cyclic handler, the save and restore processing for the registers related to the coprocessor (FPR0 to FPR31, FCR31) must be described in the cyclic handler's activate and end sections, respectively. Furthermore, to assign FCR31 (control/status register) uniquely to the cyclic handler, describe processing to set this register in the activate section of the cyclic handler.

7.5.10 Issuance of service calls from cyclic handler

The service calls that can be issued from the cyclic handler are the same as those calls that can be issued from interrupt processing. Therefore, service calls in the form of ixxx_yyy starting with i and service calls in the form of sns_yyy can be issued. Service calls in the form of xxx_yyy without i can also be issued from the cyclic handler. For example, even if act_tsk, from which the i of iact_tsk is omitted, is issued, an error does not occur, and the same processing as that of iact_tsk is performed. For details, refer to **13.8**.

7.5.11 Obtaining cyclic handler information

Cyclic handler information such as the activation interval can be obtained by using the service calls ref_cyc and iref_cyc. For details of each service call, refer to **CHAPTER 13**.

CHAPTER 8 INTERRUPT MANAGEMENT

This chapter describes interrupt management in the RX4000.

8.1 Overview

Processing in response to interrupt sources and CPU exceptions can be said to be one of the most important functions in a system using a real-time OS. This is because, assuming the RX4000 is used embedded in a control device, the OS can improve the performance of the system as a whole by responding quickly to peripheral devices such as sensors.

One of the features of a general-purpose OS such as Windows or UNIX is that processing dependent on the device, such as interrupt processing, is concealed, enabling use of an interface common to multiple applications. It could be argued, however, that in order to maximize the performance of the CPU or peripheral device, processing dependent on the device such as interrupt and exception processing should not be concealed, which is why with the RX4000, users are supplied with a source file as a sample, which can be rewritten to accord with the user's application system. This chapter should be read assuming that the source file is in an unmodified state.

8.2 Interrupt Management

The interrupt management performed by the kernel can be divided depending on the contents into interrupt control and interrupt processing management. Interrupt control refers to interrupt enable/disable processing, which includes not only an explicit enable/disable function via service calls, but also a function whereby the kernel enables or disables interrupts in order to preserve the integrity of the data while it is performing processing.

Interrupt processing management refers to the function of registering and executing processing routines in order to carry out processing in response to an acknowledged interrupt.

8.2.1 Interrupt control

The RX4000 disables interrupts in the target processor (VR4100 Series or VR5000 Series) by manipulating the IE or IM bit of the status register. Interrupts can also be enabled/disabled via control from an external interrupt controller.

The service calls provided by the RX4000 to control interrupts are (i)loc_cpu, (i)unl_cpu, which enable/disable interrupts by manipulating the IE bit, (i)chg_ims, which uses the IM bit, and dis_int, ena_int, which use an external interrupt controller.

dis_int and ena_int also set the variable cpusts, which determines whether interrupts are enabled or disabled in the OS.

The kernel also uses the IM bit of the status register to disable interrupts while it is performing processing.

In this way interrupts can always be acknowledged in interrupt processing in which service calls provided by the kernel are not used.

8.2.2 Interrupt processing management

When using the source file provided as a sample without modification, the interrupt processing flow is as shown below. This flow is also shown in diagram form in Figure 8-1.

(1) An interrupt is generated and control shifts to a CPU interrupt/exception vector.

(2) At the interrupt/exception vector, in the sample, processing only branches to the interrupt/exception processing main body.

(3) Because control shifts to a single vector for all (general) exceptions/interrupts in the VR4100 Series or VR5000 Series, the exception source is determined by the initial section of the interrupt/exception processing, which then divides. CPU and service call exceptions are not taken into consideration here.

(4) Registers that may be corrupted by interrupt processing, such as temporary registers set up by the C compiler, are saved in the initial section of the interrupt processing.
(v0 to v1, a0 to a3, t0 to t9, ra, epc, sr)

(5) After these registers have been saved, the kernel's internal function "_kernel_int_handle" is called, which notifies the kernel of the activation of interrupt processing. Note that in order to call the above function, the values of the epc, cause, and status registers must be assigned as parameters when an interrupt is generated.

(6) Once the kernel has received notification of the activation of interrupt processing, the interrupt service routine enters a state in which it can be activated (service calls can be issued). After the state has shifted, the kernel calls back the function "_user_int_handle" described by the user (function name fixed). Before calling the function "_user_int_handle", the kernel switches the stack to the interrupt stack so that all the subsequent processing is performed on the interrupt stack. It is therefore necessary to add the size of the stack consumed by interrupt processing to the stack of each task.

(7) After the interrupt source has been determined by "_user_int_handle", ivsta_isr is issued, activating an interrupt service routine in response to the generated interrupt source.

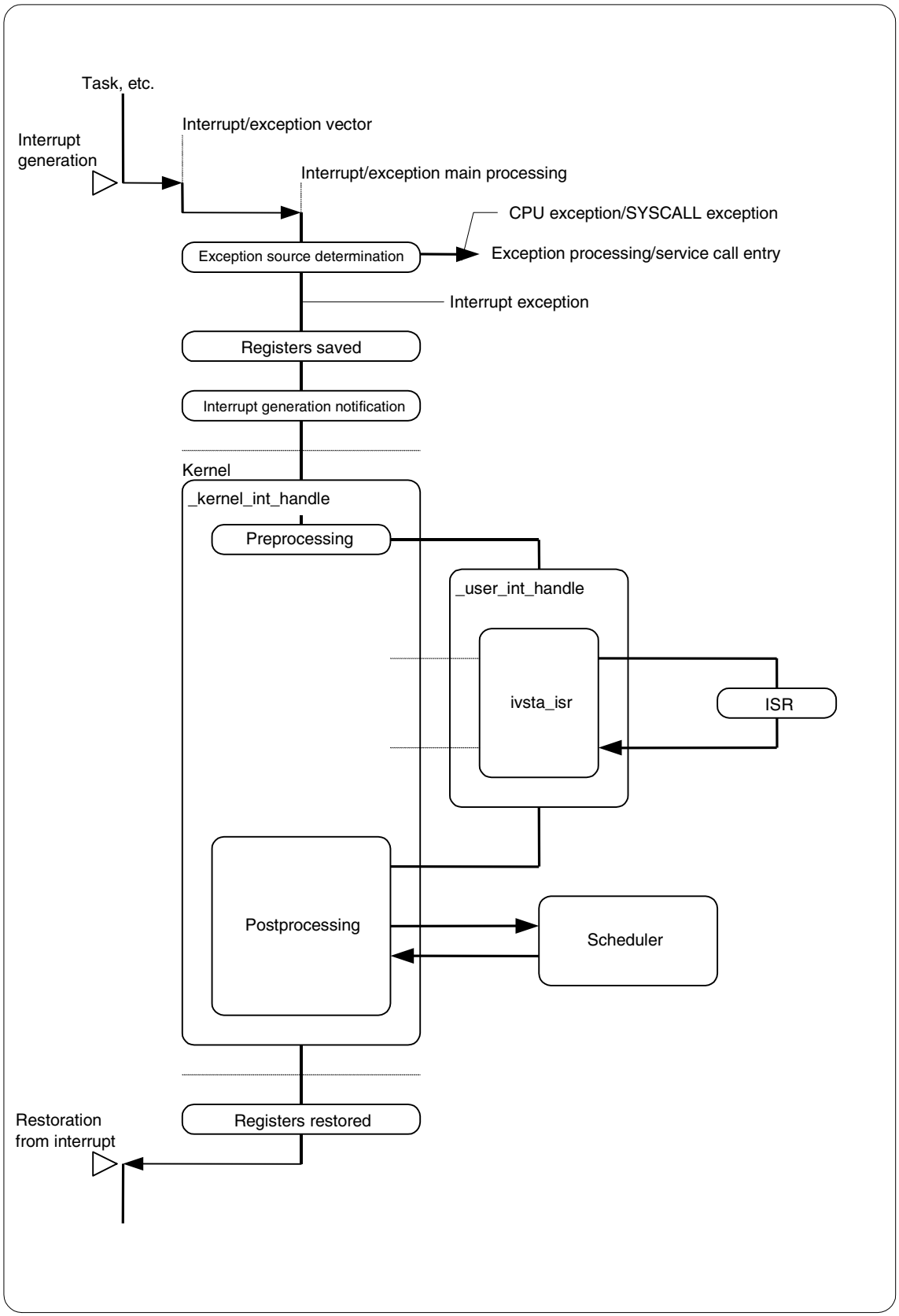
(8) The required interrupt processing is performed by the interrupt servicing routine.

(9) "_user_int_handle" is then terminated by C language return or equivalent processing. At this time, if an event in generated in the interrupt processing that requires scheduling, the kernel activates the scheduler and switches tasks.

(10) If scheduling is not required in (9) above, or if the interrupted task is re-dispatched after scheduling, control returns to the point at which "_kernel_int_handle" was called.

(11) The registers saved in (4) are restored, via the eret instruction, to the point where the interrupt was generated.

Figure 8-1. Interrupt Processing Flow



8.3 Saving/Restoring Registers

Registers are saved in the initial section of the interrupt processing in accordance with the function call conventions of the VR Series C compiler from either Green Hills Software, Inc. or Metrowerks Corp. When the interrupt processing involves the execution of processing described in C language, because the function call conventions stipulate that the preprocessing and postprocessing values stored in the save register (s0 to s7) must be the same, there is no particular need to save this register. The kernel will save and restore the gp, fp, and sp registers as required. For other registers (at, v0 to v1, t0 to t9, ra, epc, status), operation after being restored from an interrupt is not guaranteed unless their values are saved when the interrupt is generated and then returned to their original values when restored from interrupt processing.

Note that the size of stack consumed when these registers are saved is not included in the size of the stack automatically augmented by the kernel when a task is created.

8.4 Interrupt Generation Notification

In order to utilize the services provided by the kernel, such as issuing service calls via interrupt processing carried out when an interrupt is generated, the kernel must be notified of the activation of interrupt processing. However, there may also be cases when it is desirable to terminate interrupt processing without receiving services from the kernel in order to speed up the processing. Cautions to be observed when using these two types of processing are outlined below.

(1) Terminating interrupt processing without using kernel's services

In this case, it is unnecessary to inform the kernel of the activation of interrupt processing. However, the user must describe the entire processing, from activate to finish. Because the kernel cannot participate at all, service call issuance and multiple interrupt acknowledgement are completely disabled in the processing. Note also that if the interrupt processing consumes the stack, unless stack switching is specified, operations will be carried out on the task stack, making it imperative that the entire task stack be augmented by the size of the stack consumed by this processing. It is therefore recommended to employ this type of processing only for very light processing, such as simple reading and writing data from/to I/O.

(2) Using kernel's services in interrupt processing

The kernel can be notified of the activation of interrupt processing by describing the kernel's internal function "_kernel_int_handle" in the initial section of the interrupt processing. Once notified, the kernel performs processing such as switching the stack to the interrupt stack, and putting the service calls described later into an activation enabled state. Interrupt processing performed by the user is enabled when the function "_user_int_handle" is called back from the kernel.

In this function, issuance of service calls activation with i and multiple interrupt acknowledgement are enabled, allowing execution of the main body of interrupt processing that uses the kernel's services (without necessarily having to activate an interrupt service routine).

8.5 Interrupt Service Routines

The routine that is activated when an interrupt is generated and performs processing in response to the interrupt source is known as an interrupt service routine (ISR).

An interrupt service routine is described as a void type function with one VP_INT type argument. The extended data exinf of the activated interrupt service routine is passed for this argument.

Example)

```
void int_serial(VP_INT exinf)
{
        ...
        return;
}
```

8.5.1 Interrupt service routine ID number and interrupt number

As with other objects, an interrupt service routine has a unique ID number to identify its routine. However, with interrupt service routines, it is also necessary to specify an interrupt number in order to identify the interrupt source corresponding to a particular service routine.

The same interrupt number can be specified for more than one interrupt service routine.

8.5.2 Creating interrupt service routines

Interrupt service routines can be created either by issuing the service call (a)cre_isr, or by specifying the static API CRE_ISR, which performs equivalent processing to acre_isr when the system is initialized.

If (a)cre_isr is issued, the kernel stores data such as the attribute and the interrupt number in the interrupt service routine control block corresponding to the specified ID number and then initializes that block.

A queue for managing routines with the same interrupt number (interrupt table) is also controlled from the system base table, where processing is performed to register control blocks in a queue corresponding to the number held by the routine.

8.5.3 Deleting interrupt service routines

An interrupt service routine is deleted by issuing the del_isr service call.

When del_isr is issued, the kernel invalidates the specified interrupt servicing routine control block and deletes the control block from the interrupt table controlled by the system base table.

After an interrupt service routine is deleted, an interrupt service routine with the same ID number as the deleted routine can be newly created.

8.5.4 Activating interrupt service routines

An interrupt service routine is activated after an interrupt has been acknowledged and the source of the interrupt determined. The interrupt service routine is activated by specifying the interrupt source number for ivsta_isr (ivsta_isr does not cause a service call exception). If there is more than one routine for the same interrupt source number, ivsta_isr causes all the interrupt service routines to be activated. The order of activation at this time is the order in which the routines were created.

8.5.5 Terminating interrupt service routines

An interrupt service routine is terminated by C language return or equivalent processing. A return value is not required. Control therefore returns to either the next interrupt service routine to be activated or to the point at which ivsta_isr was issued.

8.5.6 PID

If PID (Position Independent Data) is used for the code created when the interrupt service routine program is compiled or assembled, the address that is to be the base of an interrupt service routine when it is created must be assigned as a parameter (gp of the interrupt service routine creation packet T_CISR). The assigned address is set in the gp register when the interrupt service routine is activated.

Note that this base address is unconditionally set in the gp register regardless of its attribute, etc., and therefore must be set for programs that reference the gp register. If the gp register is not referenced, set NULL = 0.

8.5.7 Use of coprocessor in interrupt service routine

To use the coprocessor in an interrupt service routine, the save and restore processing for the registers related to the coprocessor (FPR0 to FPR31, FRC31) must be described in the interrupt service routine's activate and end sections, respectively. Furthermore, to assign FCR31 (control/status register) uniquely to the interrupt service routine, describe processing to set this register in the activate section of the interrupt service routine.

8.5.8 Issuance of service calls from interrupt service routines

Service calls in the form of ixxx_yyy starting with i and service calls in the form of sns_yyy can be issued from an interrupt service routine. Service calls in the form of xxx_yyy without i can also be issued from an interrupt service routine. For example, even if act_tsk, from which the i of iact_tsk is omitted, is issued, an error does not occur, and the same processing as that of iact_tsk is performed. For details, refer to **13.8**.

CHAPTER 9 SYSTEM CONFIGURATION MANAGEMENT

This chapter describes system configuration management in the RX4000.

9.1 Overview

System configuration management includes the following functions.

- Defining and activating a CPU exception handler to be called when the CPU detects an exception
- Defining and activating an initialization routine to be called at system initialization
- Defining and activating an idle routine for when there are no tasks in the running or ready states (refer to **CHAPTER 1** for details of idle routines)

9.2 Exception Processing

If the CPU detects an exception such as 0 remainder, the kernel activates a corresponding exception handler. The exception processing and CPU exception handlers provided in the RX4000 are described below. Note that as with interrupt processing, a source file is also provided as a sample, allowing users to describe exception processing that best suits their system.

9.2.1 Exception processing flow

The assumed exception processing flow of the RX4000 is shown below. This flow is also shown in diagram form in Figure 9-1.

(1) An exception occurs and control shifts to a CPU interrupt/exception vector.

(2) At the interrupt/exception vector, in the sample, processing only branches to the exception processing main body.

(3) Because control shifts to a single vector for all exceptions/interrupts in the VR4100 Series or VR5000 Series, the exception source is determined by the initial section of the interrupt/exception processing, which then divides.

(4) If the exception that occurred is a service call exception, control shifts to the initial processing section of the service call. Also, if a device such as a monitor for debugging is being used, be aware that control may have to be passed to the debug side, depending on the type of exception.

(5) All registers used are saved to the stack in the initial section of the exception processing.

(6) After these registers have been saved, the kernel's internal function "_kernel_exc_handle" is called, which notifies the kernel of the activation of exception processing. Note that in order to call the above function, the values of the epc, cause, and status registers and the top address of the register save area must be assigned as parameters when an exception occurs.

(7) Once the kernel has received notification of the activation of exception processing, the CPU exception handlers enter a state in which they can be activated (service calls can be issued). After the state has shifted, the kernel calls back the function "_user_exc_handle" described by the user (function name fixed).

(8) After the exception source has been determined by "_user_exc_handle", ivsta_exc is issued, activating a CPU exception handler corresponding to source of the exception that occurred.

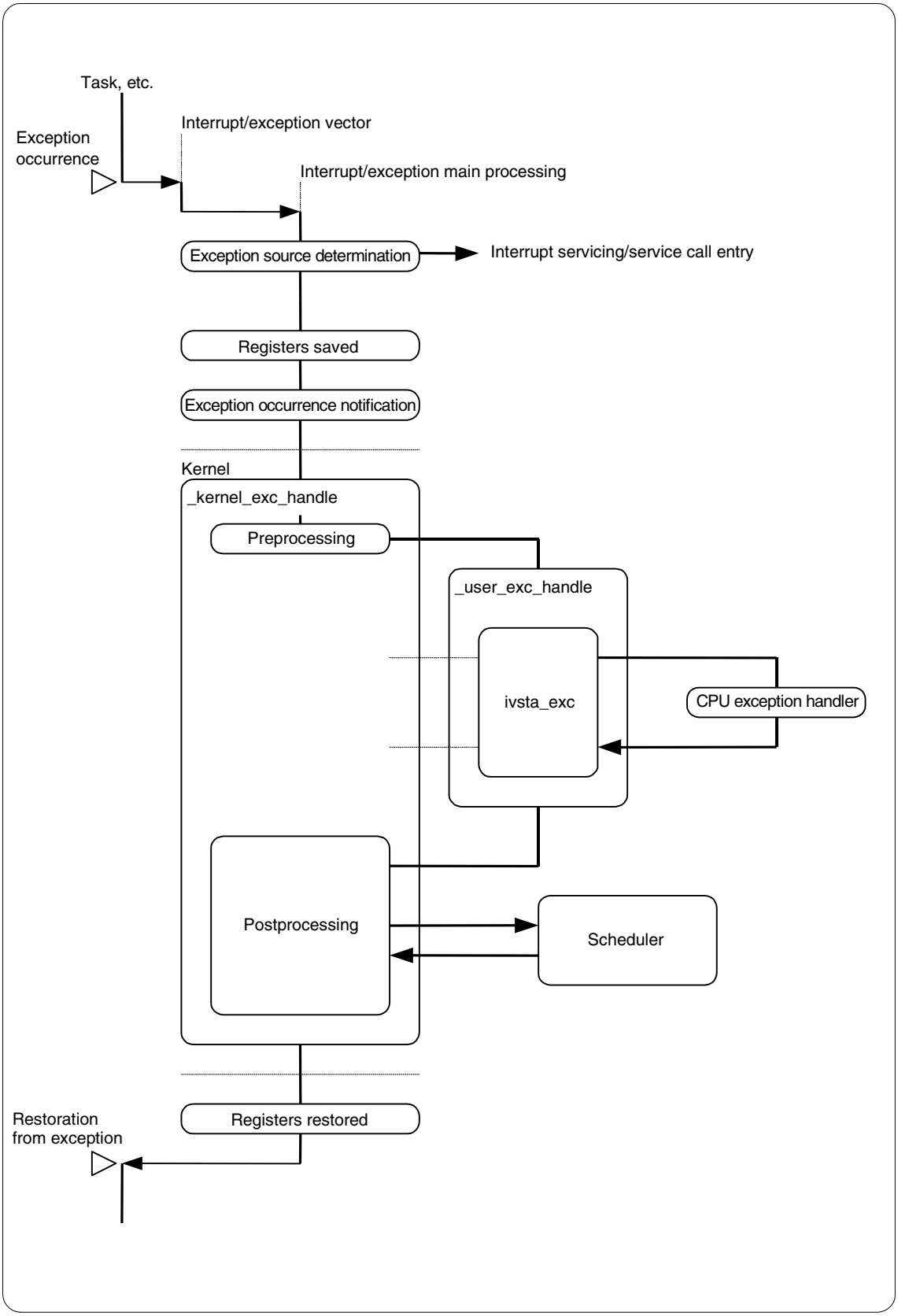
(9) The required exception processing is performed by the CPU exception handler.

(10) "_user_exc_handle" is then terminated by C language return or equivalent processing. At this time, if an event in generated in the exception processing that requires scheduling, the kernel activates the scheduler and switches tasks.

(11) If scheduling is not required in (10) above, or if the task that caused the exception is re-dispatched after scheduling, control returns to the point at which "_kernel_exc_handle" was called.

(12) The registers saved in (5) are restored, via the eret instruction, to the point where the exception occurred. Note that in cases when restoring the saved epc register to its indicated address inadvertently causes the reoccurrence of an exception, the value of epc must be changed to an appropriate value.

Figure 9-1. Exception Processing Flow



9.2.2 Saving/restoring registers

All registers including FPU registers are saved in the initial section of the exception processing. This is because by informing the exception main processing of the address of this save area, the state in which an exception occurs can be ascertained during exception processing. When it is not necessary to save all the registers, registers can be saved in accordance with the function call conventions of the VR Series C compiler from Green Hills Software, Inc. When the exception processing involves the execution of processing described in C language, because the function call conventions stipulate that the preprocessing and postprocessing values stored in the save register (s0 to s7) must be the same, there is no particular need to save this register. The kernel will save and restore the gp, fp, sp, and FPU registers as required.

In either case, for other registers (at, v0 to v1, t0 to t9, ra, epc, status), operation after being restored from an exception is not guaranteed unless their values are saved when the exception occurred and then returned to their original values when restored from exception processing.

Note that the size of stack consumed when these registers are saved is not included in the size of the stack automatically augmented by the kernel when a task is created.

9.2.3 Exception occurrence notification

In order to utilize the services provided by the kernel, such as issuing service calls via exception processing carried out when an exception occurs, the kernel must be notified of the activation of exception processing. However, there may also be cases when it is desirable to terminate exception processing without receiving services from the kernel in order to speed up the processing. Cautions to be observed when using these two types of processing are outlined below.

(1) Terminating exception processing without using kernel's services

In this case, it is unnecessary to inform the kernel of the activation of exception processing. However, the user must describe the entire processing, from activate to finish. Because the kernel cannot participate at all, service call issuance is completely disabled in the processing.

(2) Using kernel's services in exception processing

The kernel can be notified of the activation of exception processing by describing the kernel's internal function "_kernel_exc_handle" in the initial section of the exception processing. Once notified, the kernel puts the CPU exception handlers into an activation-enabled state. Exception processing performed by the user is enabled when the function "_user_exc_handle" is called back from the kernel.

In this function, issuance of service calls activation with i is enabled, allowing the kernel's services to be received (without necessarily having to activate a CPU exception handler).

9.2.4 CPU exception handlers

A CPU exception handler is a handler that is activated when an exception occurs. A CPU exception handler operates using the identical context (stack) to the location of the exception. The service calls that can be issued are the same as those that can be issued for an interrupt service routine. The description format of a CPU handler is shown below. Note, however, that this format can be changed arbitrarily by the user by changing the source file. The address where the data of registers saved when an exception occurred is stored is passed as a parameter for the source file provided as a sample.

Example)

```
void exchdr(VP fp)
{
        ...
        return;
}
```

(1) Exception handler number

A number known as the exception handler number is used to identify the CPU exception handler. Any number 0 or greater but less than the maximum number of exception handlers can be used. However, the relationship between the exception handler number and the exception source must be determined by the user (by the type of processing performed by the CPU exception handler).

It is assumed that the value of the ExcCode field of the cause register in the source file provided as a sample will be used for the exception handler number.

(2) Defining/deleting CPU exception handlers

CPU exception handlers can be defined either by issuing the service call def_exc, or by specifying the static API DEF_EXC, which performs equivalent processing to def_exc when the system is initialized.

If def_exc is issued, the kernel secures a control block in which it stores data such as the attribute and activation address of the CPU exception handler, and then initializes that block.

A CPU exception handler definition is deleted by issuing the def_exc service call with the address of the CPU exception handler definition packet set as NULL.

(3) Activating/terminating CPU exception handlers

A CPU exception handler is activated after an exception has occurred and the source of the exception determined. A CPU exception handler is activated by specifying the exception handler number for vsta_exc, and terminated by C language return or equivalent processing.

(4) PID

If PID (Position Independent Data) is used for the code created when the CPU exception handler program is compiled or assembled, the address that is to be the base of a CPU exception handler when it is defined must be assigned as a parameter (gp of the CPU exception handler definition packet T_DEXC). The assigned address is set in the gp register when the CPU exception handler is activated.

Note that this base address is unconditionally set in the gp register regardless of its attribute, etc., and therefore must be set for programs that reference the gp register. If the gp register is not referenced, set NULL = 0.

(5) Use of coprocessor in CPU exception handler

To use the coprocessor in a CPU exception handler, the save and restore processing for the registers related to the coprocessor (FPR0 to FPR31, FRC31) must be described in the CPU exception handler's activate and end sections, respectively. Furthermore, to assign FCR31 (control/status register) uniquely to the CPU exception handler, describe processing to set this register in the activate section of the CPU exception handler.

(6) Issuance of service calls from CPU exception handler

The service calls that can be issued from a CPU exception handler are the same as those that can be issued from an interrupt service routine: all service calls activation with i in the ixxx_yyy format.

9.3 Initialization Routines

An initialization routine is a processing routine for initialization called between kernel activation and when the first task is executed (scheduler activated). An initialization routine is only activated at system initialization and unlike other objects does not have a control block.

Service calls can be issued from an initialization routine. These service calls include all those that can be issued from tasks, except ext_tsk and exd_tsk. Note that depending on the service call, the result of the processing may be meaningless (i.e., in the case of the service call sns_ctx).

The description format of an initialization routine is shown below. Extended data is passed as the parameter.

Example)

```
void inirtn(VP_INT exinf)
{
        ...
        return;
}
```

9.3.1 Defining initialization routines

An initialization routine is defined using the static API ATT_INI in the CF definition file; it cannot be defined via a service call while the system is operating.

9.3.2 Activating initialization routines

An initialization routine is incorporated as a subroutine called by the kernel after the kernel is initialized. Note that all processing corresponding to static API is carried out as an initialization routine created by the configurator.

9.3.3 Terminating initialization routines

An initialization routine is terminated by C language return or equivalent processing. Control then shifts to either the next initialization routine to be activated, or to the scheduler. Note that if the state of the system is changed due to the issuance of loc_cpu or dis_dsp, these service calls will be forcibly cancelled when the initialization routine is complete.

CHAPTER 10 SERVICE CALL MANAGEMENT

10.1 Overview

Service call management is a function used to register functions described by users in the system as service calls. The registered service calls are known as extended service call routines.

10.2 Extended Service call Routines

User-described functions that are registered in the system as service calls are known as extended service call routines. Extended service call routines each have a number, or extended function code, that identifies the routine, the specification of which together with the issuance of the service call (i)cal_svc calls the corresponding routine.

An extended service call routine operates as the extension of the task or handler that issued (i)cal_svc. Therefore, although standard service calls can be issued from a service call routine, the service calls that can be issued and the processing after issuance depend on the place (context) where (i)cal_svc was issued. Also, if a task exception processing request is sent to a task that issued an extended service call routine from interrupt processing that occurred while the extended service call routine was being executed, because the extended service call routine is assumed to be task processing, the task exception processing routine will be activated before the extended service call routine resumes processing.

The description format of an extended service call routine is shown below. Note, however, that the parameter types and names are defined by the user. In the example below, arg1 corresponds to the (i)cal_svc parameter arg 1.

Example)

```
ER svcrtn(VW arg1, VW arg2, VW arg3)
{
        ...
        return E_OK;
}
```

10.2.1 Defining extended service call routines

Extended service call routines are defined and deleted by issuing the service call def_svc. Items such as the routine's address and extended function code are specified as the parameters when defining the routine.

10.2.2 Activating and terminating extended service call routines

An extended service call routine is activated by specifying the extended function code and parameters passed to the service call for (i)cal_svc and then issuing this service call. An extended service call routine is terminated by C language return or equivalent processing. At this time, an error code corresponding to the processing contents of the service call is returned as the return value.

When starting an extended service call routine using (i)cal_svc, only up to three parameters can be passed. To pass four or more parameters, the user must describe an interface library for the extended service call routine. For describing an interface library for the extended service call routine, refer to **RX4000 (μITRON4.0) Installation System Specification**.

10.2.3 PID

If PID (Position Independent Data) is used for the code created when the extended service call routine program is compiled or assembled, the address that is to be the base of an extended service call routine when it is defined must be assigned as a parameter (gp of the extended service call routine definition packet T_DSVC). The assigned address is set in the gp register when the extended service call routine is activated.

Note that this base address is unconditionally set in the gp register regardless of its attribute, etc., and therefore must be set for programs that reference the gp register. If the gp register is not referenced, set NULL = 0.

10.2.4 Use of coprocessor in extended service call routine

To use the coprocessor in an extended service call routine, the coprocessor must be in a state whereby it can be used by the task or interrupt service routine that called the extended service call routine. In other words, only tasks with the attribute TA_COP can call an extended service call routine in which the coprocessor is used. When issuing an extended service call routine from an interrupt service routine, the save and restore processing for the registers related to the coprocessor (FPR20 to FPR31) must be described in the extended service call routine to be issued.

CHAPTER 11 INITIALIZATION PROCESSING

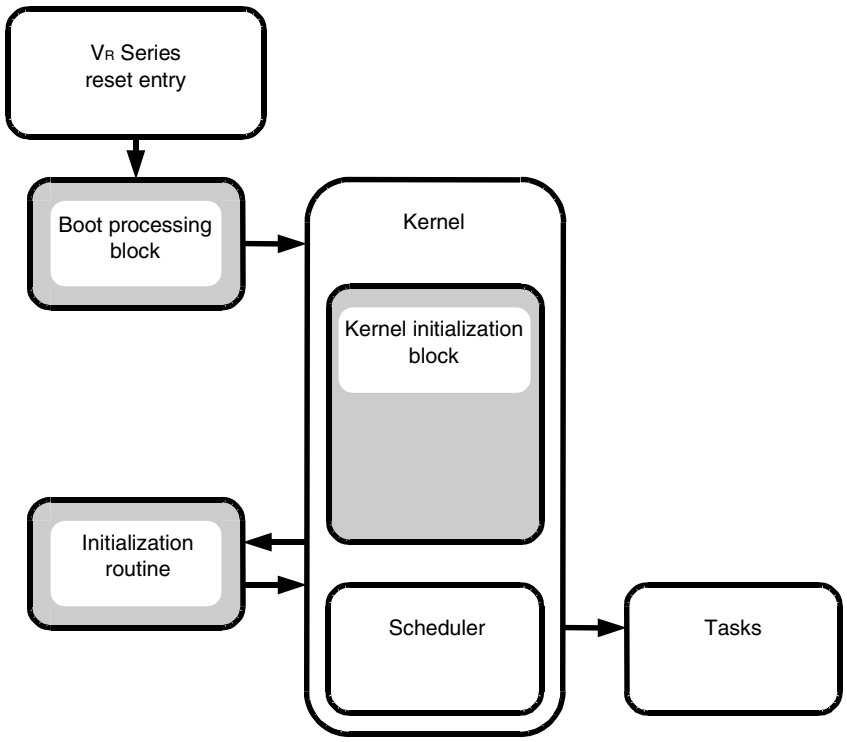
This chapter describes the initialization processing that is carried out by the kernel when the system is activated. For further details, refer to the **RX4000 (μITRON4.0) Installation User's Manual (U14834E)**.

11.1 Overview

Initialization processing consists of initializing the hardware operated by the RX4000, initializing the kernel, and initializing the software.

Figure 11-1 shows the initialization processing flow.

Figure 11-1. Initialization Processing Flow



11.2 Boot Processing Block

The boot processing block performs the initialization processing that must be complete before the kernel is activated. This processing involves initializing the peripheral hardware and copying data from ROM to RAM.

11.3 Kernel Initialization Block

In the kernel initialization block, the system information table that is based on the CF definition file and is output by the configurator is referenced, and the kernel itself is initialized. The system can receive no service calls from the kernel until this processing is complete. Processing in the kernel initialization block involves the following.

- Interrupt initialization
- Pool creation and initialization
- Management object creation and initialization

11.3.1 Interrupt initialization

This processing includes initializing interrupts and setting the values of interrupt masks for operation when the kernel has disabled interrupts. Because interrupt processing is dependent upon the execution environment of the user, a user own coding block (such as _kernel_ini_custom) is used. For details, refer to **RX4000 (μITRON4.0) Installation System Specification**.

11.3.2 Pool creation and initialization

This processing involves creating and initializing the system, user, and stack pools based on the memory data specified in the CF definition file.

11.3.3 Management object creation and initialization

(1) Creation of system base table

This processing involves creating and initializing the system base table based on the system data specified in the CF definition file.

(2) Creation of ready queue

This processing involves creating and initializing the ready queue based on the system data specified in the CF definition file.

(3) Creation of objects

Objects are created by static API described in the CF definition file based on object creation data. Also, if specified, activation processing (for tasks) and operation state shift processing (for cyclic handlers) is also carried out.

11.4 Initialization Routines

Initialization routines are processing routines used to perform initialization called between when kernel initialization is complete and when the first task is executed. Initialization routines are registered by describing ATT_INI in the CF definition file. Additional modules may also be added implicitly as initialization routines using the configurator in case of future kernel function expansion.

Initialization routines are used to perform initialization after the kernel is activated, such as setting the initial status of tasks. Refer to **CHAPTER 1** for further details.

CHAPTER 12 INTERFACE LIBRARY

This chapter describes the interface library function. For further details, refer to the **RX4000 (μITRON4.0) Installation User's Manual (U14834E)**.

12.1 Overview

When an application program is written in C, and a service call is issued, the service call is described in an external function format. The format in which the service call is actually issued, however, differs from the external function format. An interface library is therefore required to translate a service call issued in external function format into the kernel issuance format and thus act as an agent between application programs and the kernel. The interface library provided in the RX4000 supports the C compiler package for the VR Series from Green Hills Software, Inc.

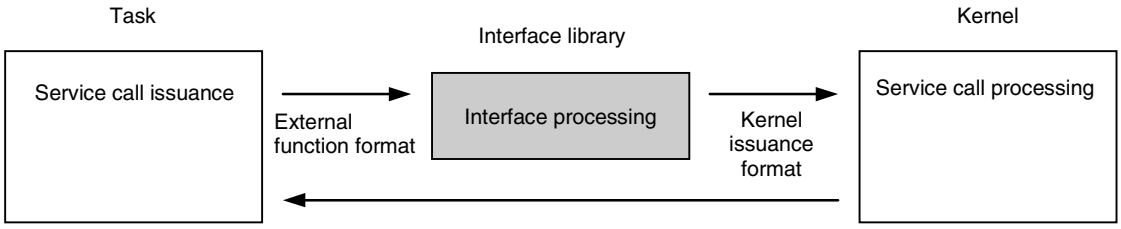
There are two types of interface libraries: one is for service calls and the other is for extended service calls that must be described by the user. For the interface library for extended service calls, refer to **10.2**.

12.2 Function and Position of Interface Library

The interface library is an interface program used to translate service calls issued from the application program in an external function format into the issuance format of the kernel, and is thus positioned between the application program and the kernel.

The interface library contains a function to shift control to the kernel once it has made the parameter and other settings required by the kernel to perform processing. The position of the interface library is shown below.

Figure 12-1. Position of Interface Library



CHAPTER 13 SERVICE CALLS

This chapter describes the service calls provided in the RX4000.

13.1 Overview

Service calls are provided to enable manipulation of a variety of objects from the user's processing program (task, interrupt handler, etc.), and operate by calling the service routines of the kernel. By issuing service calls, users can indirectly manipulate objects managed by the kernel via synchronous communication or interrupt servicing.

13.2 Types of Functions

The service call (vatt_idl) provided in the RX4000 include those that comply with the μITRON4.0 Specification (165 types), and those that are unique to the RX4000 (3 types). These service calls are further divided into 10 groups, according to their function, as shown below.

(1) Task management function service calls (20 types)

These service calls are used for functions such as direct manipulation of task states and referencing.

cre_tsk	acre_tsk	del_tsk	act_tsk	iact_tsk
can_act	ican_act	sta_tsk	ista_tsk	ext_tsk
exd_tsk	ter_tsk	chg_pri	ichg_pri	get_pri
iget_pri	ref_tsk	iref_tsk	ref_tst	iref_tst

(2) Task-associated synchronization function service calls (15 types)

These service calls are used for synchronization functions associated with tasks that are not dependent on other resources such as semaphores.

slp_tsk	tslp_tsk	wup_tsk	iwup_tsk	can_wup
ican_wup	rel_wai	irel_wai	sus_tsk	isus_tsk
rsm_tsk	irsm_tsk	frsm_tsk	ifrsm_tsk	dly_tsk

(3) Task exception processing function service calls (8 types)

These service calls are used for functions related to task exception processing.

def_tex	ras_tex	iras_tex	dis_tex	ena_tex
sns_tex	ref_tex	iref_tex		

(4) Synchronous communication management function service calls (59 types)

These service calls are used for synchronous communication functions that are independent of tasks, i.e., functions related to semaphores, event flags, data queues, mailboxes, and mutex.

cre_sem	acre_sem	del_sem	sig_sem	isig_sem
wai_sem	pol_sem	ipol_sem	twai_sem	ref_sem
iref_sem	cre_flg	acre_flg	del_flg	set_flg
iset_flg	clr_flg	iclr_flg	wai_flg	pol_flg
ipol_flg	twai_flg	ref_flg	iref_flg	cre_dtq
acre_dtq	del_dtq	snd_dtq	psnd_dtq	ipsnd_dtq
tsnd_dtq	fsnd_dtq	ifsnd_dtq	rcv_dtq	prcv_dtq
iprcv_dtq	trcv_dtq	ref_dtq	iref_dtq	cre_mbx
acre_mbx	del_mbx	snd_mbx	isnd_mbx	rcv_mbx
prcv_mbx	iprcv_mbx	trcv_mbx	ref_mbx	iref_mbx
cre_mtx	acre_mtx	del_mtx	loc_mtx	ploc_mtx
tloc_mtx	unl_mtx	ref_mtx	iref_mtx	

(5) Memory pool management function service calls (22 types)

These service calls are used for functions related to fixed- and variable-length memory pools.

cre_mpf	acre_mpf	del_mpf	get_mpf	pget_mpf
ipget_mpf	tget_mpf	rel_mpf	irel_mpf	ref_mpf
iref_mpf	cre_mpl	acre_mpl	del_mpl	get_mpl
pget_mpl	ipget_mpl	tget_mpl	rel_mpl	irel_mpl
ref_mpl	iref_mpl			

(6) Time management function service calls (14 types)

These service calls are used to perform processing related to time.

set_tim	iset_tim	get_tim	iget_tim	isig_tim
cre_cyc	acre_cyc	del_cyc	sta_cyc	ista_cyc
stp_cyc	istp_cyc	ref_cyc	iref_cyc	

(7) System status management function service calls (14 types)

These service calls are used for functions related to the status of the entire system, such as disabling dispatch.

rot_rdq	irot_rdq	get_tid	iget_tid	loc_cpu
iloc_cpu	unl_cpu	iunl_cpu	dis_dsp	ena_dsp
sns_ctx	sns_loc	sns_dsp	sns_dpn	

(8) Interrupt management function service calls (10 types)

These service calls are used for functions related to interrupt servicing.

cre_isr	acre_isr	del_isr	dis_int	ena_int
chg_ims	ichg_ims	get_ims	iget_ims	

(9) System configuration management function service calls (3 types)

These include service calls related to CPU exception handlers and initialization routines.

def_exc	vatt_idl	

(10) Service call management function service calls (3 types)

These service calls are used for functions related to extended service call routines.

def_svc	cal_svc	ical_svc

13.3 Return Values (Error Codes)

The service calls provided in the RX4000 return error codes or return values that are prescribed in the μITRON4.0 Specification. The names of symbols used for error codes can be used by including the header file kernel.h. The return values that can be returned by service calls are listed below.

Table 13-1. Error Code List

Symbol	Value	Meaning
E_OK	0	Normal termination
E_NOSPT	−9	Reserved function code
E_RSFN	−10	Unsupported function
E_RSATR	−11	Illegal attribute
E_PAR	−17	Illegal parameter
E_ID	−18	Specified ID number exceeds range
E_CTX	−25	Context error
E_ILUSE	−28	Illegal use of service call
E_NOMEM	−33	Insufficient memory
E_NOID	−34	ID number cannot be assigned
E_OBJ	−41	Target object in illegal state
E_NOEXS	−42	Target object does not exist
E_QOVR	−43	Queuing overflow
E_RLWAI	−49	Waiting forcibly released by (i)rel_wai
E_TMOUT	−50	Timeout, polling failure
E_DLT	−51	Target resource deleted while being waited for

13.4 Data Types

The parameters or error codes of the service calls provided in the RX4000 are defined and declared based on data types that conform to the μITRON4.0 Specification. The data types used in the RX4000 are shown below. Note that these data types can be used by including the header file kernel.h.

13.4.1 General data types

The data types listed below are general data types prescribed by μITRON4.0.

typedef char	B;	…	Signed 8-bit integer
typedef short	H;	…	Signed 16-bit integer
typedef long	W;	…	Signed 32-bit integer
typedef unsigned char	UB;	…	Unsigned 8-bit integer
typedef unsigned short	UH;	…	Unsigned 16-bit integer
typedef unsigned long	UW;	…	Unsigned 32-bit integer
typedef char	VB;	…	Variable data type value (8 bits)
typedef short	VH;	…	Variable data type value (16 bits)
typedef long	VW;	…	Variable data type value (32 bits)
typedef void	*VP;	…	Variable data type value (pointer)
typedef void	(*FP) ();	…	Program activate address
typedef int	INT;	…	Signed integer (processor width)
typedef unsigned int	UINT;	…	Unsigned integer (processor width)
typedef int	VP_INT	…	Variable data type value (pointer) or signed integer (processor width)

13.4.2 RX4000 data types

Based on the general data types, dedicated types for data that has a special meaning, such as an ID number, are declared for the parameters of the service calls provided in the RX4000.

typedef H	FN;	…	Function code
typedef H	ID;	…	Object ID number
typedef W	BOOL;	…	Bool value
typedef UH	INTNO;	…	Interrupt number
typedef H	EXCNO;	…	Exception handler number
typedef UH	ATR;	…	Object attribute
typedef UH	STAT;	…	Object state
typedef W	ER;	…	Error code
typedef W	ER_ID;	…	Error code or object ID number
typedef W	ER_BOOL;	…	Error code or boolean value
typedef INT	ER_UNIT;	…	Error code or unsigned integer (Signed integer with the same bit width as ER or UNIT, whichever is greater. The valid number of bits of an unsigned integer is 1 bit shorter than UNIT.)
typedef H	PRI;	…	Priority order
typedef UINT	TEXPTN;	…	Task exception source
typedef UINT	FLGPTN;	…	Event flag bit pattern
typedef UH	MODE;	…	Service call operating mode
typedef UW	SIZE;	…	Memory size
typedef W	TMO;	…	Timeout time
typedef UW	RELTIM;	…	Relative time
typedef struct t_systim {			
UW ltime;			
UW utime;			
}	SYSTIM;	…	System time
typedef struct t_c*xxx* {			
Reference of each service call page			
}	T_C*XXX*	…	Object creation packet
typedef struct t_d*xxx* {			
Reference of each service call page			
}	T_D*XXX*	…	Object definition packet
typedef struct t_r*xxx* {			
Reference of each service call page			
}	T_R*XXX*	…	Object data packet

13.5 Macros

When describing application programs such as tasks or interrupt handlers, the following macros can be treated as defined constants, and therefore cannot be redefined by the user. These definitions are made in the header file kernel.h.

For items whose Value column is blank, refer to the page on which that service call is defined. CF in the Value column indicates that the value is determined by the configurator.

Table 13-2. Macro List (1/2)

Macro Name	Value	Meaning
TA_*XXX*		Object attribute
TWF_*XXX*		Service call operating mode
TTS_*XXX*		Object state
TSK_SELF	0	Self task specification
TSK_NONE	0	No corresponding task exists
TPRI_SELF	0	Self task base priority specification
TPRI_INI	0	Specification of task priority at activation
MERCD(ER ercd)	–	Return main error code section of the error code ercd (currently always –1)
SERCD(ER ercd)	–	Return sub error code section of the error code ercd (value indicated in error code list)
ERCD(ER mercd, ER sercd)	–	Generates and returns an error code from main error code mercd and sub-error code sercd.
TMIN_TPRI	1	Minimum value of task priority order (highest priority)
TMAX_TPRI	CF	Maximum value of task priority order (lowest priority)
TMIN_MPRI	1	Minimum value of message priority order (highest priority)
TMAX_MPRI	0x7fff	Maximum value of message priority order (lowest priority)
TKERNEL_MAKER	0x000d	Kernel manufacturer code (NEC Electronics)
TKERNEL_PRID	0x0000	Kernel identification number
TKERNEL_SPVER	0x0400	Version number of μITRON Specification
TKERNEL_PRVER	0x0400	Version number of kernel
TMAX_ACTCNT	127	Maximum queue limit for activation requests
TMAX_WUPCNT	127	Maximum queue limit for wakeup requests
TMAX_SUSCNT	127	Maximum queue limit for suspend requests
TBIT_TEXPTN	32	Number of bits in task exception source
TBIT_FLGPTN	32	Number of bits in event flag
TTIC_NUME	CF	Numerator of time tick cycle
TTIC_DENO	1	Denominator of time tick cycle
TSZ_DTQ(UINT dtqcnt)		cre_dtq page reference
TSZ_MPRIHD		cre_mbx page reference
TSZ_MPF		cre_mpf page reference
TSZ_MPL		cre_mpl page reference
TMAX_MAXSEM	0x7fffffff	Maximum value of maximum number of semaphores

Table 13-2. Macro List (2/2)

Macro Name	Value	Meaning
E_*XXX*		Error code
NULL	0	Invalid pointer
TRUE	1	True
FALSE	0	False

13.6 Parameter Value Range

Some of the service call parameters supported by the RX4000 have a range of permissible values, while others allow the use of only specific values reserved by the system. The ranges of parameter values are shown below.

Table 13-3. Parameter Value Ranges

Parameter Type	Range
Task ID No.	1 to Notes 1,2
Semaphore ID No.	1 to Note 2
Event flag ID No.	1 to Note 2
Data queue ID No.	1 to Note 2
Mailbox ID No.	1 to Note 2
Mutex ID No.	1 to Note 2
Fixed-length memory pool ID No.	1 to Note 2
Variable-length memory pool ID No.	1 to Note 2
Cyclic handler ID No.	1 to Note 2
Interrupt ID No.	0 to Note 2
Exception handler ID No.	0 to Note 2
Extended function code	1 to Note 2
Task priority	1 to Note 3
Message priority	1 to 0x7fff
Timeout time	1 to 0x7fffffffNote 4
Cycle time	1 to 0xffffffff
Delay time	1 to 0xffffffff
System time	0 to 0xffffffffffff
User stack size	0 to 0x7fffffff
Memory block size	1 to 0x7fffffff
gp value	0 to 0xffffffffNote 5

Notes 1. 0 is reserved by the system

2. Value specified in the CF definition file (maximum value: 0x7fff)

3. Value specified in the CF definition file (maximum value: 255)

4. 0 and –1 are reserved by the system

5. The operation is not guaranteed if a value other than a multiple of 4 is specified.

13.7 Context From Which Service calls Can Be Issued

A context that can be regarded as a part of task processing is called a task context, and a context that cannot is called a non-task context. The context in which a CPU exception handler and extended service call routine are issued is dependent upon the original context that calls these routines.

The following table shows to which context each routine belongs.

Table 13-4. Context That Can Issue Service calls

Routine Name	Context
Task	Task
Task exception processing routine	Task
Interrupt service routine	Non-task
Cyclic handler	Non-task
Initialization routine	Non-task
CPU exception handler	Task/non-task
Extended service call routine	Task/non-task
Idle routine	Non-task

Service calls that start with “i” are issued from a non-task context, and the other service calls are issued from a task context. However, service calls for a non-task context in the form of ixxx_yyy can be issued from a task context and service calls without “i” in the form of xxx_yyy can be issued from a non-task context. This is to improve the ease of using service calls. Note, however, that not all service calls in the form of xxx_yyy can be issued from a non-task context.

Service calls in the form of sns_yyy can be issued from any context.

Although an idle routine is regarded as a non-task context, all service calls can be issued from an idle routine.

The contexts from which service calls can be issued are listed below.

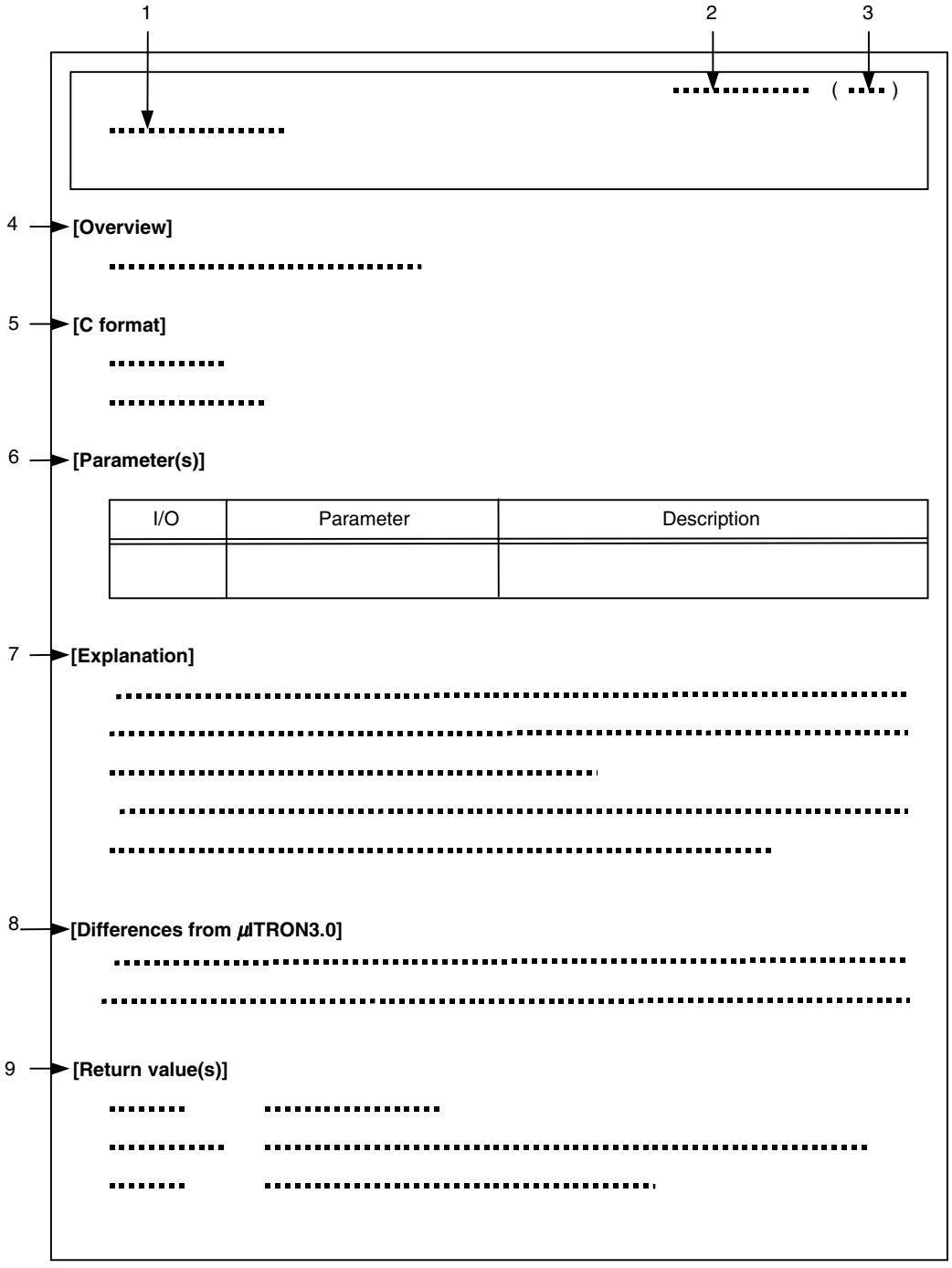
Table 13-5. Context From Which Service calls Can Be Issued

Service call Form	Task Context	Non-Task Context
xxx_yyy	√	Δ (Some service calls can be issued.)
vxxx_yyy	√	–
ixxx_yyy	√	√
ivxxx_yyy	√	√
sns_yyy	√	√

13.8 Explanation of Service calls

This section explains the service calls provided in the RX4000 in accordance with the following format.

Figure 13-1. Service call Description Format



(1) Name

Indicates the name of the service call.

(2) Semantics

Indicates the source of the name of the service call.

(3) Overview

Outlines the functions of the service call.

(4) C format

Indicates the format to be used when describing a service call to be issued in C.

(5) Parameter(s)

Service call parameters are explained in the following format.

I/O	Parameter	Description
A	B	C

A. Parameter classification
   I ... Input parameter
   O ... Output parameter
B. Parameter type and name
C. Description of parameter

(6) Explanation

Explains the function of a service call.

(7) Differences from μITRON3.0

Explains points that differ from the μITRON3.0 Specification or the RX4000 Ver3.x (which complies with the μITRON3.0 Specification).

(8) Return value

Explains the service call's return value.

13.8.1 Task management function service calls

This section describes the task management function service calls listed in the following table.

Table 13-6. Task Management Function Service calls

Name	Function
cre_tsk	Creates a task
acre_tsk	Creates a task (automatic assignment of ID No.)
del_tsk	Deletes a task
act_tsk/iact_tsk	Activates a task
can_act/ican_act	Invalidates an activation request
sta_tsk/ista_tsk	Activates a task (activation request not retained)
ext_tsk	Terminates this task
exd_tsk	Terminates and deletes this task
ter_tsk	Forcibly terminates another task
chg_pri/ichg_pri	Changes the priority level of a task
get_pri/iget_pri	Obtains the priority level of a task
ref_tsk/iref_tsk	References the state of a task
ref_tst/iref_tst	References the state of a task (simple version)

cre_tsk

Create Task

[Overview]

Creates a task.

[C format]

```
#include <kernel.h>
ER cre_tsk(ID tskid, T_CTSK *pk_ctsk);
```

[Parameters]

I/O	Parameter		Description
I	ID	tskid	ID number of task to be created
I	T_CTSK *	pk_ctsk	Address of task creation packet

Configuration of T_CTSK

```
typedef struct t_ctsk {
        ATR         tskatr;     /* Task attribute */
        VP_INT      exinf;      /* Extended data */
        FP          task;       /* Activation address */
        PRI         itskpri;    /* Initial priority */
        SIZE        stksz;      /* Stack size */
        VP          stk;        /* Top address of stack area */
        VP          gp;         /* PID base address (gp) */
        VP          tp;         /* Reserved area */
} T_CTSK;
```

[Explanation]

The task with the ID number specified by tskid is created based on the data stored in the packet pk_ctsk. In other words, after securing the stack area to be used by the task, data such as the attribute and task activation address is stored in the task control block of the specified ID number, and that block is then initialized. Accordingly, the task shifts from the non-existent to the dormant state.

When TA_ACT is specified for the task attribute, however, act_tsk or equivalent processing is performed after creation and the task is immediately activated. The task therefore shifts straight from the non-existent state to the ready or running state.

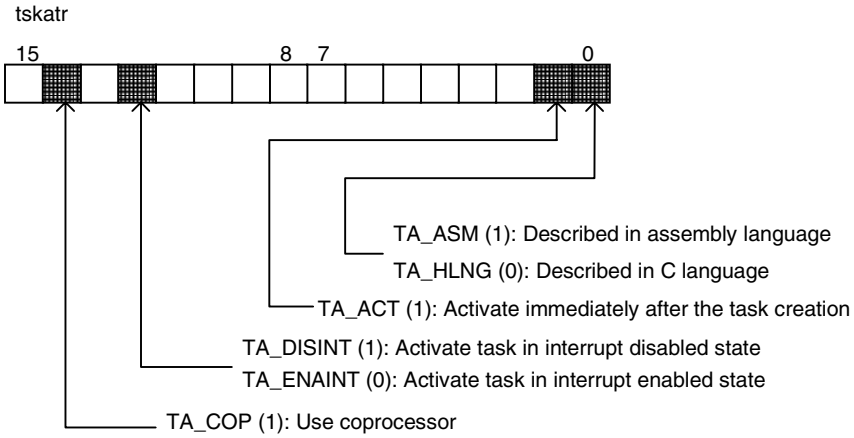
If the same task ID is specified while the task still exists, and if cre_tsk is issued, the task generation request is not queued. The E_OBJ error is returned.

The task creation packet T_CTSK is described in detail below.

- tskatr

  Specifies data such as whether the coprocessor is used or not and whether interrupts are enabled or disabled at activation as task attributes. Values that can be specified as task attributes are shown below.

Figure 13-2. Task Attributes



  Bit 0 is used to specify the task's description language. TA_HLNG is specified if the task is described in C language and TA_ASM if assembly language is used. In this version, however, there are no differences in the processing for these two attributes.

  Bit 1 is used to specify whether a task is to be activated after creation. If the attribute TA_STA is assigned to a task, act_tsk or equivalent processing is performed for that task after it is created.

  Bit 12 is used to specify whether interrupts are enabled or disabled immediately after a created task is activated. If TA_DISINT is set, the task is activated in the interrupt disabled state. If TA_ENAINT is set, the task is activated in the interrupt enabled state. Note that disabling interrupts means that interrupts that can initiate interrupt processing in which kernel services are received are masked.

  Bit 14 is used to specify whether the created task will use the coprocessor (FPU). If TA_COP is set, the preprocessing and postprocessing required when the coprocessor is used is performed.

  Note that when a combination of these attributes is set, the logical sum of the above values is set for tskatr.

- exinf

  Stores user-specific information regarding the creation of tasks. This information can be referenced as parameters for tasks activated by (i)act_tsk.

- task

  Sets the activation address of the task to be created.

- itskpri

  Sets the initial priority (task initial priority) of the task to be created. When a task that has been terminated is reactivated, the priority of that task is determined by the value set in itskpri and not by the value at termination. The range of the values that can be specified for itskpri is from 0x1 to the maximum priority (specified by the CF definition file).

- stksz

  Specifies the size of the stack area used by the task. When task creation processing is performed, the kernel secures a stack area from the stack pool based on this size specification. Failure to secure this area results in an error, and the error code E_NOMEM is returned.

  Note that because the kernel augments this area with items such as context size, the size of the area actually secured from the stack pool is larger than the specified value. For further details, refer to the **RX4000 (μITRON4.0) Instruction User's Manual (U14834E)**.
- stk

  This field specifies the top address of the stack area used by the task to be created. However, because this function is not supported in the RX4000, stk should always be set to NULL. Settings other than NULL are ignored.
- gp

  Sets the base address (gp) of the PID (Position Independent Data) used by the task to be created. If PID is not used by the task, set gp to NULL.
- tp

  Reserved area. tp should always be set to NULL. Settings other than NULL are ignored.

[Differences from μITRON3.0]

1. The values of the macros TA_ASM and TA_HLNG used when specifying the task attribute have been reversed (TA_ASM 0 → 1, TA_HLNG 1 → 0). Caution is therefore required when porting task programs in which values are directly specified without using a macro from μITRON3.0 to μITRON4.0.
2. TA_ACT has been added to the task attributes.
3. It is no longer necessary to use an attribute to specify the use of PID by a task.
4. stk has been added to the task creation packet (T_CTSK) members.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	cre_tsk is not included in the system
E_RSATR	−11	The task attribute is illegal – TA_COP was specified in the FPU disabled mode – An attribute that does not exist in the specifications was specified
E_PAR	−17	The parameter is illegal – The initial priority is outside the range (itskpri ≤ 0, maximum priority value < itskpri)
E_ID	−18	The task ID number is outside the range (tskid ≤ 0, maximum task count < tskid)
E_CTX	−25	cre_tsk was issued from a non-task context cre_tsk was issued in the CPU lock state
E_NOMEM	−33	A stack of the requested size cannot be secured
E_OBJ	−41	A task with the same ID number already exists

acre_tsk

Create Task with Automatic ID Assignment

[Overview]

Creates a task (Automatic assignment of ID number)

[C format]

```
#include <kernel.h>
ER_ID acre_tsk(T_CTSK *pk_ctsk);
```

[Parameter]

I/O	Parameter		Description
I	T_CTSK *	pk_ctsk	Address of task creation packet

[Explanation]

A task is created based on the task creation data stored in pk_ctsk and its ID number is returned. In other words, a usable task control block in the non-existent state is searched, assigned, and initialized, and the stack area is secured. The ID number of that block is then returned as the return value. If the return value is a negative value, an error occurs. The correspondence between negative return values and errors is shown in **[Return values]** below.

Refer to the description of cre_tsk for details of the task creation packet.

[Differences from μITRON3.0]

acre_tsk is a newly created service call equivalent to cre_tsk but with the addition of an automatic ID number assignment function.

[Return values]

Symbol	Value	Meaning
(Positive integer)		ID number of the created task (normal termination)
E_RSFN	−10	acre_tsk is not included in the system A value other than NULL was specified for the stack area address pk_ctsk.stk
E_RSATR	−11	The task attribute is illegal – TA_COP was specified in the FPU disabled mode – An attribute that does not exist in the specifications was specified
E_PAR	−17	The parameter is illegal – The initial priority is outside the range (itskpri ≤ 0, maximum priority value < 0)
E_CTX	−25	acre_tsk was issued from a non-task context acre_tsk was issued in the CPU lock state
E_NOMEM	−33	A stack of the requested size can not be secured
E_NOID	−34	An ID number cannot be assigned (failed to secure TCB)

del_tsk

Delete Task

[Overview]

Deletes a task.

[C format]

```
#include <kernel.h>
ER del_tsk(ID tskid);
```

[Parameter]

I/O	Parameter		Description
I	ID	tskid	ID number of task to be deleted

[Explanation]

The task specified by tskid is deleted. In other words, the control block being used by the task is invalidated, and the task is shifted from the dormant state to the non-existent state. The stack area being used by the task is also returned to the stack pool. After deletion, the ID number of the deleted task can be used for a newly created task.

del_tsk is used for tasks in the dormant state. It is therefore impossible for a task to issue this service call to itself. Tasks perform self-deletion via the service call exd_tsk. If del_tsk is issued for a task not in the dormant state, an error occurs, and the error code E_OBJ is returned.

[Differences from μITRON3.0]

None

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	del_tsk is not included in the system
E_ID	−18	The task ID number is outside the range (tskid ≤ 0, maximum task count < tskid)
E_CTX	−25	del_tsk was issued from a non-task context del_tsk was issued in the CPU lock state
E_OBJ	−41	The target task is not in the dormant state
E_NOEXS	−42	The target task does not exist

act_tsk/iact_tsk

Activate Task

[Overview]

Activates a task (Activation request retained)

[C format]

```
#include <kernel.h>
ER act_tsk(ID tskid);
```

[Parameter]

I/O	Parameter		Description
I	ID	tskid	ID number of task to be activated

[Explanation]

The task specified by tskid is activated. In other words, the task is shifted from the dormant state to the ready state. If the target task has already been activated and is no longer in the dormant state, the activation request is retained, and the target task is reactivated as soon as it is terminated. An activation request can be retained up to 127 times; if the number of retained activation requests exceeds 127, an error occurs, and the error code E_QOVR is returned. act_tsk can therefore be issued 128 times in succession for a task in the dormant state.

Note that the extended data exinf set as a parameter when the task was created is passed for tasks activated by act_tsk. The task can handle this extended data as a function parameter (first argument).

If TSK_SELF (=0) is set for tskid, the task itself is the target of the service call.

When issuing this service call from a non-task block, such as an interrupt service routine, use iact_tsk.

iact_tsk is intended to be issued from a non-task context but it can also be issued from a task context. act_tsk can be issued from a non-task context.

[Differences from μITRON3.0]

act_tsk/iact_tsk is a newly created service call equivalent to sta_tsk but with the addition of an activation request retention function. However, unlike sta_tsk, act_tsk passes the extended information from task creation to the task instead of the task activation code.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	act_tsk/iact_tsk is not included in the system
E_ID	−18	The task ID number is outside the range (tskid < 0, maximum task count < tskid) TSK_SELF is specified and issued from a non-task context.
E_CTX	−25	act_tsk/iact_tsk was issued in the CPU lock state
E_NOEXS	−42	The target task does not exist
E_QOVR	−43	The number of times the activation request was retained exceeded 127

can_act/ican_act

Cancel Activation

[Overview]

Invalidates task activation requests

[C format]

```
#include <kernel.h>
ER_UINT can_act(ID tskid);
```

[Parameter]

I/O	Parameter		Description
I	ID	tskid	ID number of task whose activation requests are to be invalidated

[Explanation]

can_act/ican_act invalidates all the activation requests retained for the task specified by tskid and returns the number of invalidated activation requests. Even when an activation request has not been issued for the target task, 0 is returned without error occurrence. If the return value is a negative value, an error occurs. The correspondence between negative return values and errors is shown in **[Return values]** below.

When TSK_SELF (=0) is specified for tskid, the task itself becomes the target of the service call.

ican_act is intended to be issued from a non-task context but it can also be issued from a task context. can_act can be issued from a non-task context.

[Differences from μITRON3.0]

can_act/ican_act is a newly created service call.

[Return values]

Symbol	Value	Meaning
(Integer of 0 or greater)		Number of invalidated activation requests (normal termination)
E_RSFN	−10	can_act/ican_act is not included in the system
E_ID	−18	The task ID number is outside the range (tskid < 0, maximum task count < tskid) TSK_SELF is specified and issued from a non-task context.
E_CTX	−25	can_act/ican_act was issued in the CPU lock state
E_OBJ	−41	The target task is in the dormant state
E_NOEXS	−42	The target task does not exist

Start Task

sta_tsk/ista_tsk

[Overview]

Activates a task (Activation request not retained)

[C format]

```
#include <kernel.h>
ER sta_tsk(ID tskid, VP_INT stacd);
```

[Parameters]

I/O	Parameter		Description
I	ID	tskid	ID number of task to be activated
I	VP_INT	stacd	Task activation code

[Explanation]

The task specified by tskid is activated. In other words, the task is shifted from the dormant state to the ready state. Also, the value specified by stacd is passed to the task to be activated as the task activation code. The activated task can handle the task activation code as a function parameter (first argument).

Unlike act_tsk, if sta_tsk is issued when the target task has already been activated and is in a state other than the dormant state, an error occurs, and the error code E_OBJ is returned.

To issue this service call from a non-task context such as an interrupt service routine, use ista_tsk.

ista_tsk is intended to be issued from a non-task context but it can also be issued from a task context. sta_tsk can be issued from a non-task context.

[Differences from μITRON3.0]

The type of the task activation code stacd has changed from INT to VP_INT. Note that VP_INT is a new type defined by μITRON4.0.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	sta_tsk/ista_tsk is not included in the system
E_ID	−18	The task ID number is outside the range (tskid ≤ 0, maximum task count < tskid)
E_CTX	−25	sta_tsk/ista_tsk was issued from a non-task context sta_tsk/ista_tsk was issued in the CPU lock state
E_OBJ	−41	The target task is not in the dormant state
E_NOEXS	−42	The target task does not exist

ext_tsk

Exit Task

[Overview]

Terminates a task

[C format]

```
#include <kernel.h>
void ext_tsk(void);
```

[Parameter]

None

[Explanation]

A task terminates itself normally. In other words, a task shifts itself from the running state to the dormant state. All data specified for the terminated task such as its priority is initialized. However, even if the task had acquired a resource such as a semaphore or a memory block prior to termination, these will not be released. If the task is locking a mutex, only the mutex will be released, which may result in tasks waiting to lock a mutex being released from the waiting state.

If the task is retaining activation requests, after this processing, the activation request counter will be decremented by 1 and the task will be immediately reactivated.

If ext_tsk is issued in the dispatch disabled state or not issued from a task, operation cannot be guaranteed.

[Differences from μITRON3.0]

The mutex release processing and reactivation processing when activation requests are retained have been added.

[Return value]

None

Exit and Delete Task

exd_tsk

[Overview]

Terminates and deletes a task

[C format]

```
#include <kernel.h>
void exd_tsk(void);
```

[Parameter]

None

[Explanation]

A task terminates itself normally and is simultaneously deleted. In other words, a task shifts itself from the running state to the non-existent state, and the control block and stack area being used by the task are released. However, even if the task had acquired a resource such as a semaphore or a memory block prior to termination, these will not be released. If the task is locking a mutex, only the mutex will be released, which may result in tasks waiting to lock a mutex being released from the waiting state.

Even if an activation request is retained by the task for which exd_tsk is issued, exd_tsk ignores this request and deletes the task. The request that was retained automatically becomes invalid.

If exd_tsk is issued in the dispatch disabled state or not issued from a task, operation cannot be guaranteed.

[Differences from μITRON3.0]

The mutex release processing has been added.

[Return value]

None

ter_tsk

Terminate Task

[Overview]

Forcibly terminates another task

[C format]

```
#include <kernel.h>
ER ter_tsk(ID tskid);
```

[Parameter]

I/O	Parameter		Description
I	ID	tskid	ID number of task to be terminated

[Explanation]

The task specified by tskid is forcibly terminated. In other words, the task with the tskid ID number is forcibly shifted from the ready or running state to the dormant state. Therefore, all the information of the task that has been terminated, such as the priority, is initialized. However, even if the task had acquired a resource such as a semaphore or a memory block prior to termination, these will not be released. If the task is locking a mutex, only the mutex will be released, which may result in tasks waiting to lock a mutex being released from the waiting state.

If the target task is waiting at the top of the waiting task queue to acquire a memory block from a variable-length memory pool, the task is removed from the top of the queue and the tasks waiting in the second and subsequent positions in the queue are released from the memory block acquisition waiting state.

If the task is retaining activation requests, after this processing, the activation request counter will be decremented by 1 and the task will be immediately reactivated.

[Differences from μITRON3.0]

The mutex release processing and reactivation processing when activation requests are retained have been added.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ter_tsk is not included in the system
E_ID	−18	The task ID number is outside the range (tskid ≤ 0, maximum task count < tskid)
E_CTX	−25	ter_tsk was issued from a non-task context ter_tsk was issued in the CPU lock state
E_OBJ	−41	The target task is in the dormant state or issued this service call to itself
E_NOEXS	−42	The target task does not exist

Change Task Priority

chg_pri/ichg_pri

[Overview]

Changes the priority of a task

[C format]

```
#include <kernel.h>
ER chg_pri(ID tskid, PRI tskpri);
```

[Parameters]

I/O	Parameter		Description
I	ID	tskid	ID number of task whose priority is to be changed
I	PRI	tskpri	Priority after change

[Explanation]

The base priority of the task specified by tskid is changed to the value specified by tskpri. If TSK_SELF = 0 is set for tskid, the task itself is the target of the service call.

An integer of 1 or higher is specified for tskpri; the lower the value, the higher the priority. The maximum specifiable value (lowest priority) is the one specified at configuration. By specifying TPRI_INI = 0 for tskpri, it is also possible to make the base priority after the change the same as the activation priority specified when the task was created.

If the target task is not locking a mutex, the base priority and the current priority are the same, and therefore the current priority of the target task also changes to tskpri. If the target task is locking a mutex with the attribute TA_INHERIT or TA_CEILING and a priority lower than the task's current priority is specified, it is possible that only the base priority will change (i.e., the current priority will not change). Also if this service call is issued to a task locking a mutex with the attribute TA_CEILING and the specified priority is higher than the top priority of the mutex, an error occurs, and the error code E_ILUSE is returned.

If the target task is in the running or ready state, the task shifts to the bottom of the corresponding priority section of the ready queue. Even if the priority before chg_pri was issued is the same as that after the change, the task is still moved to the bottom of the ready queue. It is therefore possible to make a task relinquish the right to use the CPU by making the task itself the target, specifying the same priority as the current one, and issuing chg_pri. Note that if chg_pri is issued to a task in the running or ready state while dispatch is disabled, ready queue manipulation is the only processing performed, and the task in the running state continues processing.

If the target task is queuing in a waiting task queue in priority order, issuing chg_pri may change the order of that queue (or the order in which the tasks are released from waiting). Especially, if chg_pri is issued and changes the task at the top of a queue for tasks waiting to acquire a memory block from a variable-length memory pool, the task may acquire a memory block and be released from waiting. Also, if the target task is waiting to lock a mutex with the attribute TA_INHERIT, priority inheritance processing may occur as a result of the change in the order of the ready queue.

ichg_pri is intended to be issued from a non-task context but it can also be issued from a task context. chg_pri can be issued from a non-task context.

[Differences from μITRON3.0]

Processing related to mutex has been added.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	chg_pri/ichg_pri is not included in the system
E_PAR	−17	The priority is outside the range (tskpri < 0, maximum priority < tskpri)
E_ID	−18	The task ID number is outside the range (tskid < 0, maximum task count < tskid) TSK_SELF is specified and issued from a non-task context.
E_CTX	−25	chg_pri/ichg_pri was issued in the CPU lock state
E_ILUSE	−28	The priority after the change is higher than the maximum priority of the mutex
E_OBJ	−41	The target task is in the dormant state
E_NOEXS	−42	The target task does not exist

Get Task Priority

get_pri/iget_pri

[Overview]

Obtains the priority of a task

[C format]

```
#include <kernel.h>
ER get_pri(ID tskid, PRI *p_tskpri);
```

[Parameters]

I/O	Parameter		Description
I	ID	tskid	ID number of task whose priority is to be obtained
O	PRI *	p_tskpri	Address where priority is stored

[Explanation]

The current priority of the task specified by tskid is obtained and stored at the address specified by p_tskpri. If TSK_SELF = 0 is set for tskid, the task itself is the target of the service call.

iget_pri is intended to be issued from a non-task context but it can also be issued from a task context. get_pri can be issued from a non-task context.

[Differences from μITRON3.0]

get_pri/iget_pri is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	get_pri/iget_pri is not included in the system
E_ID	−18	The task ID number is outside the range (tskid < 0, maximum task count < tskid) TSK_SELF is specified and issued from a non-task context.
E_CTX	−25	get_pri/iget_pri was issued in the CPU lock state
E_OBJ	−41	The target task is in the dormant state
E_NOEXS	−42	The target task does not exist

Refer Task Status

ref_tsk/iref_tsk

[Overview]

Obtains task information

[C format]

```
#include <kernel.h>
ER ref_tsk(ID tskid, T_RTSK *pk_rtsk);
```

[Parameters]

I/O	Parameter		Description
I	ID	tskid	ID number of task whose information is to be obtained
O	T_RTSK *	pk_rtsk	Pointer to task information packet

Configuration of T_RTSK

```
typedef struct t_rtsk {
        STAT        tskstat;    /* Task state */
        PRI         tskpri;     /* Current priority of task */
        PRI         tskbpri;    /* Base priority of task */
        STAT        tskwait;    /* Wait source */
        ID          wobjid;     /* ID number of object waited for */
        TMO         lefttmo;    /* Time until timeout */
        UINT        actcnt;     /* Number of activation requests */
        UINT        wupcnt;     /* Number of wakeup requests */
        UINT        suscnt;     /* Number of nested suspend requests */
        ATR         tskatr;     /* Task attribute */
        PRI         itskpri;    /* Task priority at activation */
} T_RTSK;
```

[Explanation]

Information related to the task specified by tskid is stored in the packet specified by pk_rtsk. If TSK_SELF = 0 is set for tskid, the task itself is the target of the service call. The task information packet T_RTSK is described in detail below.

- tskstat

  The value indicating the current state of the task is stored. The corresponding values are as follows.

Table 13-7. Task Status

Symbol	Value	Meaning
TTS_RUN	0x0001	Running state
TTS_RDY	0x0002	Ready state
TTS_WAI	0x0004	Waiting state
TTS_SUS	0x0008	Suspended state
TTS_WAS	0x000c	Waiting-suspended state
TTS_DMT	0x0010	Dormant state

- tskpri

  The current priority of the target task is stored.
- tskbpri

  The base priority of the target task is stored.
- tskwait

  When the target task is in the waiting state, the value indicating the wait source is stored. The corresponding values are as follows.

Table 13-8. Wait Source

Symbol	Value	Meaning
TTW_NONE	0x0000	Not in waiting state
TTW_SLP	0x0001	Waiting due to slp_tsk/tslp_tsk
TTW_DLY	0x0002	Waiting due to dly_tsk
TTW_SEM	0x0004	Waiting due to wai_sem/twai_sem
TTW_FLG	0x0008	Waiting due to wai_flg/twai_flg
TTW_SDTQ	0x0010	Waiting due to snd_dtq/tsnd_dtq
TTW_RDTQ	0x0020	Waiting due to rcv_dtq/trcv_dtq
TTW_MBX	0x0040	Waiting due to rcv_mbx/trcv_mbx
TTW_MTX	0x0080	Waiting due to loc_mtx/tloc_mtx
TTW_MPF	0x2000	Waiting due to get_mpf/tget_mpf
TTW_MPL	0x4000	Waiting due to get_mpl/tget_mpl

- wobjid
  When the target task is in the waiting state, the ID number of the target object is stored.
- lefttmo
  When the target task has a timeout function and is in the waiting state, the time left before the timeout is stored.
- actcnt
  The number of activation requests retained by the target task is stored.
- wupcnt
  The number of wakeup requests retained by the target task is stored.
- suscnt
  The number of suspend requests retained by the target task is stored.
- tskatr
  The target task attributes are stored. The values stored have the meanings described in cre_tsk.
- itskpri
  The initial priority of the target task is stored.

iref_tsk is intended to be issued from a non-task context but it can also be issued from a task context. ref_tsk can be issued from a non-task context.

[Differences from μITRON3.0]

1. The order of the parameters has changed.
   ref_tsk (T_RTSK *pk_rtsk, ID tskid); → ref_tsk (ID tskid, T_RTSK *pk_rtsk);
2. The members of the task information packet T_RTSK have changed.
   (Deleted) Extended data exinf
   (Added) Base priority tskbpri, time left before timeout lefttmo, number of activation requests actcnt, task attributes tskatr, initial priority of task itskpri
3. The values for a task's wait sources have changed due to the addition of functions such as data queues.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ref_tsk/iref_tsk is not included in the system
E_ID	−18	The task ID number is outside the range (tskid < 0, maximum task count < tskid) TSK_SELF is specified and issued from a non-task context.
E_CTX	−25	ref_tsk/iref_tsk was issued in the CPU lock state
E_NOEXS	−42	The target task does not exist

Refer Task Status

ref_tst/iref_tst

[Overview]

Obtains task information (Simplified version)

[C format]

```
#include <kernel.h>
ER ref_tst(ID tskid, T_RTST *pk_rtst);
```

[Parameters]

I/O	Parameter		Description
I	ID	tskid	ID number of task whose information is to be obtained
O	T_RTST *	pk_rtst	Pointer to task information packet

Configuration of T_RTST

```
typedef struct t_rtst {
        STAT        tskstat;        /* Task state */
        STAT        tskwait;        /* Wait source */
} T_RTST;
```

[Explanation]

The states and wait sources of the task specified by tskid are stored in the packet specified by pk_rtst. If TSK_SELF = 0 is set for tskid, the task itself is the target of the service call.

The values obtained by the task state tskstat and wait source tskwait are the same as the values obtained by ref_tsk. For details, refer to the description of ref_tsk.

iref_tst is intended to be issued from a non-task context but it can also be issued from a task context. ref_tst can be issued from a non-task context.

[Differences from μITRON3.0]

ref_tst/iref_tst is a newly created service call that restricts the information obtained by ref_tsk to realize high-speed processing.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ref_tst/iref_tst is not included in the system
E_ID	−18	The task ID number is outside the range (tskid < 0, maximum task count < tskid) TSK_SELF is specified and issued from a non-task context.
E_CTX	−25	ref_tst/iref_tst was issued in the CPU lock state
E_NOEXS	−42	The target task does not exist

13.8.2 Task-associated synchronization function service calls

This section describes the task-associated synchronization function service calls listed in the following table.

Table 13-9. Task-Associated Synchronization Function Service calls

Name	Function
slp_tsk	Places a task in the wakeup waiting state
tslp_tsk	Places a task in wakeup waiting state (with timeout)
wup_tsk/iwup_tsk	Wakes up a task
can_wup/ican_wup	Invalidates a wakeup request
rel_wai/irel_wai	Forcibly releases a task from waiting
sus_tsk/isus_tsk	Forcibly places a task in the waiting state
rsm_tsk/irsm_tsk	Releases the forcible waiting state of a task
frsm_tsk/ifrsm_tsk	Releases the forcible waiting states of all tasks
dly_tsk	Places the task in the time lapse waiting state

slp_tsk

Sleep Task

[Overview]

Places a task in the wakeup waiting state

[C format]

```
#include <kernel.h>
ER slp_tsk(void);
```

[Parameter]

None

[Explanation]

A task shifts itself from the running state to the waiting (wakeup waiting) state. Tasks in the wakeup waiting state are released from waiting by the issuance of a wakeup request via wup_tsk. However, a task that has already issued a wakeup request to itself is not returned to the waiting state; the counter that retains the wakeup requests is merely decremented by 1. Therefore to place a task with retained wakeup requests in the wakeup waiting state, it is necessary to either issue (i)can_wup before issuing (t)slp_tsk, or to issue (t)slp_tsk (number of retained wakeup requests + 1) times.

Tasks placed in the waiting state due to slp_tsk are released from waiting by one of the following occurrences.

1. Reception of wakeup request via (i)wup_tsk (E_OK)
2. Forcible release of waiting state by (i)rel_wai (E_RLWAI)

[Differences from μITRON3.0]

None

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	slp_tsk is not included in the system
E_CTX	−25	slp_tsk was issued from a non-task context slp_tsk was issued in the CPU lock state slp_tsk was issued in the dispatch disabled state
E_RLWAI	−49	The waiting state is forcibly released by (i)rel_wai

Sleep Task with Timeout

tslp_tsk

[Overview]

Places a task in the wakeup waiting state (with timeout)

[C format]

```
#include <kernel.h>
ER tslp_tsk(TMO tmout);
```

[Parameter]

I/O	Parameter		Description
I	TMO	tmout	Timeout time [ms]

[Explanation]

A task shifts itself from the running state to the waiting (wakeup waiting) state for only the amount of time specified by tmout. A task in the wakeup waiting state is released from waiting either by the issuance of a wakeup request via wup_tsk, or when a timeout occurs due to the elapse of the specified time.

However, a task that has already issued a wakeup request to itself is not returned to the waiting state; the counter that retains the wakeup requests is merely decremented by 1. Therefore to place a task with retained wakeup requests in the wakeup waiting state, it is necessary to either issue (i)can_wup before issuing (t)slp_tsk, or to issue (t)slp_tsk (number of retained wakeup requests + 1) times.

Note that if TMO_POL = 0 is specified for tmout, it means that 0 has been specified for the timeout time, resulting in the invalidation of one wakeup request. If TMO_FEVR = −1 is specified for tmout, it means that an unlimited timeout time has been specified for tmout, resulting in the same operation as slp_tsk.

Tasks placed in the waiting state due to tslp_tsk are released from waiting by one of the following occurrences.

1. Reception of wakeup request via (i)wup_tsk (E_OK)
2. Timeout due to elapse of specified time (E_TMOUT)
3. Forcible release of waiting state by (i)rel_wai (E_RLWAI)

[Differences from μITRON3.0]

None

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	tslp_tsk is not included in the system
E_PAR	−17	The timeout time is illegal (tmout < TMO_FEVR)
E_CTX	−25	tslp_tsk was issued from a non-task context tslp_tsk was issued in the CPU lock state tslp_tsk was issued in the dispatch disabled state
E_RLWAI	−49	The waiting state was forcibly released by (i)rel_wai
E_TMOUT	−50	Timeout

Wakeup Task

wup_tsk/iwup_tsk

[Overview]

Wakes up a task

[C format]

```
#include <kernel.h>
ER wup_tsk(ID tskid);
```

[Parameter]

I/O	Parameter		Description
I	ID	tskid	ID number of task to be woken up

[Explanation]

A wakeup request is sent to the task specified by tskid and that task is woken up from the wakeup waiting state. If the target task is not in the wakeup waiting state, the wakeup request is retained. If TSK_SELF = 0 is set for tskid, the task itself is the target of the service call.

A task with retained wakeup requests is not placed in the wakeup waiting state unless either (i)can_wup is issued before (t)slp_tsk, or (t)slp_tsk is issued (number of retained wakeup requests + 1) times. Wakeup requests can be retained a maximum of 127 times. If wup_tsk is issued causing more than 127 wakeup requests to be retained, an error occurs and the error code E_QOVR is returned.

iwup_tsk is intended to be issued from a non-task context but it can also be issued from a task context. wup_tsk can be issued from a non-task context.

[Differences from μITRON3.0]

None

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	wup_tsk/iwup_tsk is not included in the system
E_ID	−18	The task ID number is outside the range (tskid < 0, maximum task count < tskid) TSK_SELF is specified and issued from a non-task context.
E_CTX	−25	wup_tsk/iwup_tsk was issued in the CPU lock state
E_OBJ	−41	The target task is in the dormant state.
E_NOEXS	−42	The target task does not exist
E_QOVR	−43	The number of wakeup requests retained exceeds 127

Cancel Wakeup Task

can_wup/ican_wup

[Overview]

Invalidates wakeup requests

[C format]

```
#include <kernel.h>
ER_UINT can_wup(ID tskid);
```

[Parameter]

I/O	Parameter		Description
I	ID	tskid	ID number of task whose wakeup requests are to be invalidated

[Explanation]

The wakeup requests retained for the task specified by tskid are invalidated, and the number of invalidated requests is returned. Even if no wakeup requests are being retained for the target task, 0 is returned without error occurrence. If the return value is a negative value, an error occurs. The correspondence between negative return values and errors is shown in **[Return values]** below. If TSK_SELF = 0 is set for tskid, the task itself is the target of the service call.

ican_wup is intended to be issued from a non-task context but it can also be issued from a task context. can_wup can be issued from a non-task context.

[Differences from μITRON3.0]

The interface has changed to a method in which the number of wakeup requests is returned not by the pointer but by the return value.

[Return values]

Symbol	Value	Meaning
(Integer of 0 or greater)		Number of invalidated wakeup requests (normal termination)
E_RSFN	−10	can_wup/ican_wup is not included in the system
E_ID	−18	The task ID number is outside the range (tskid < 0, maximum task count < tskid) TSK_SELF is specified and issued from a non-task context.
E_CTX	−25	can_wup/ican_wup was issued in the CPU lock state
E_OBJ	−41	The target task is in the dormant state
E_NOEXS	−42	The target task does not exist

Release Wait

rel_wai/irel_wai

[Overview]

Forcibly releases the waiting state of another task

[C format]

```
#include <kernel.h>
ER rel_wai(ID tskid);
```

[Parameter]

I/O	Parameter		Description
I	ID	tskid	ID number of task to be released from waiting state

[Explanation]

The task specified by tskid is forcibly released from the waiting state. E_RLWAI is returned to the task released from waiting as the error code of the service call that caused the wait. If the target task is not in the waiting state, an error occurs, and the error code E_OBJ is returned.

A task cannot be released from the suspended state using rel_wai. If the target task is in the waiting-suspended state, it is only released from the waiting state, and therefore shifts to the suspended state. If it is necessary to also release the task from the suspended state, use the frsm_tsk service call in combination with this one.

If the target task is waiting at the top of the waiting task queue to acquire a memory block from a variable-length memory pool, issuing rel_wai may cause the tasks waiting in the second and subsequent positions in the queue to be released from the memory block acquisition waiting state.

irel_wai is intended to be issued from a non-task context but it can also be issued from a task context. rel_wai can be issued from a non-task context.

[Differences from μITRON3.0]

None

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	rel_wai/irel_wai is not included in the system
E_ID	−18	The task ID number is outside the range (tskid ≤ 0, maximum task count < tskid)
E_CTX	−25	rel_wai/irel_wai was issued in the CPU lock state
E_OBJ	−41	The target task is not in the waiting state
E_NOEXS	−42	The target task does not exist

Suspend Task

sus_tsk/isus_tsk

[Overview]

Forcibly places a task in a waiting state

[C format]

```
#include <kernel.h>
ER sus_tsk(ID tskid);
```

[Parameter]

I/O	Parameter		Description
I	ID	tskid	ID number of task to be forcibly placed in waiting state

[Explanation]

The task specified by tskid is forcibly placed in a waiting state and becomes suspended. If the target task is in the waiting state, it enters the waiting-suspended state, which is a combination of the waiting and suspended states. A task in this state has entered the suspended state upon release from the waiting state, or has entered the waiting state upon release from the suspended state.

If (i)sus_tsk is issued more than once for the same task, the task enters a multiplexed suspended state. It is therefore necessary to issue the same number of (i)rsm_tsk service calls as (i)sus_tsk calls or (i)frsm_tsk to release a task from the suspended state. In other words, suspend requests sent by issuing (i)sus_tsk can be nested, up to 127 times. (i)sus_tsk can therefore be issued up to 127 times in succession for a task not in the suspended state. If the number of nested suspend requests exceeds 127, an error occurs, and the error code E_QOVR is returned.

Note that if TSK_SELF = 0 is set for tskid, the task itself is the target of the service call. However, if a task issues this service call to itself in the dispatch disabled state, an error occurs, and the error code E_CTX is returned.

isus_tsk is intended to be issued from a non-task context but it can also be issued from a task context. sus_tsk can be issued from a non-task context.

[Differences from μITRON3.0]

It is now possible for a task to issue sus_tsk to itself without causing an error. TSK_SELF can therefore be used as the constant that specifies self-issuance. However, an error occurs if self-issuance is specified in the dispatch disabled state.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	sus_tsk/isus_tsk is not included in the system
E_ID	−18	The task ID number is outside the range (tskid < 0, maximum task count < tskid) TSK_SELF is specified and issued from a non-task context.
E_CTX	−25	sus_tsk/isus_tsk was issued in the CPU lock state Self-issuance was specified in the dispatch disabled state
E_OBJ	−41	The target task is in the dormant state
E_NOEXS	−42	The target task does not exist
E_QOVR	−43	The number of nested suspend requests exceeds 127

Resume Task

rsm_tsk/irsm_tsk

[Overview]

Resumes operation of a task in the suspended state

[C format]

```
#include <kernel.h>
ER rsm_tsk(ID tskid);
```

[Parameter]

I/O	Parameter		Description
I	ID	tskid	ID number of task whose operation is to be resumed

[Explanation]

The task specified by tskid is released from the suspended state and resumes operation. If the target task is in the waiting-suspended state, it is only released from the suspended state, and therefore shifts into the waiting state.

If (i)sus_tsk was issued more than once for the target task and there are nested suspend requests, only one suspend request is released. In this case therefore, even if (i)rsm_tsk is issued, the suspended or waiting-suspended state will continue. To release all the suspend requests, either issue (i)rsm_tsk the same number of times as (i)sus_tsk was issued, or issue (i)frsm_tsk.

irsm_tsk is intended to be issued from a non-task context but it can also be issued from a task context. rsm_tsk can be issued from a non-task context.

[Differences from μITRON3.0]

None

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	rsm_tsk/irsm_tsk is not included in the system
E_ID	−18	The task ID number is illegal (tskid ≤ 0, maximum task count < tskid)
E_CTX	−25	rsm_tsk/irsm_tsk was issued in the CPU lock state
E_OBJ	−41	The target task is not in the suspended state
E_NOEXS	−42	The target task does not exist

Force Resume Task

frsm_tsk/ifrsm_tsk

[Overview]

Forcibly resumes operation of a task in the suspended state

[C format]

```
#include <kernel.h>
ER frsm_tsk(ID tskid);
```

[Parameter]

I/O	Parameter		Description
I	ID	tskid	ID number of task whose operation is to be resumed

[Explanation]

The task specified by tskid is forcibly released from the suspended state and resumes operation. If the target task is in the waiting-suspended state, only the suspended state is released and the task therefore shifts to the waiting state. Unlike (i)rsm_tsk, even if multiple (i)sus_tsk service calls have been issued to the target task, issuance of this service call causes the suspended state to be unconditionally released.

ifrsm_tsk is intended to be issued from a non-task context but it can also be issued from a task context. frsm_tsk can be issued from a non-task context.

[Differences from μITRON3.0]

None

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	frsm_tsk/ifrsm_tsk is not included in the system
E_ID	−18	The task ID number is outside the range (tskid ≤ 0, maximum task count < tskid)
E_CTX	−25	frsm_tsk/ifrsm_tsk was issued in the CPU lock state
E_OBJ	−41	The target task is not in the suspended state
E_NOEXS	−42	The target task does not exist

dly_tsk

Delay Task

[Overview]

Places a task in the time lapse waiting state

[C format]

```
#include <kernel.h>
ER dly_tsk(RELTIM dlytim);
```

[Parameter]

I/O	Parameter		Description
I	RELTIM	dlytim	Delay time [ms]

[Explanation]

A task places itself in a state whereby it is waiting for the time specified by dlytim to elapse. Compared with tslp_tsk, which causes a task to terminate normally and then remain in the waiting state until a wakeup request is received, dly_tsk causes a task to terminate normally and then remain in the waiting state until a specified amount of time has elapsed. Therefore, even if (i)wup_tsk is issued to a task in the time lapse waiting state, a wakeup request is sent, but the task is not released from the waiting state. The time lapse waiting state is released when another task issues (i)rel_wai.

Note that for service calls that shift tasks into waiting states with timeout functions, such as tslp_tsk, a waiting timeout time of 0 means polling activates, whereas for dly_tsk, even if the delay time is 0, the waiting state continues. At this time, waiting is released at the occurrence of the first time event (issuance of isig_tim) after the issuance of dly_tsk.

Tasks in the waiting state due to the issuance of dly_tsk are released from waiting by one of the following occurrences.

1. Elapse of specified time (E_OK)
2. Forcible release of waiting by (i)rel_wai (E_RLWAI)

[Differences from μITRON3.0]

The type of the delay time dlytim has changed from DLYTIME to RELTIM.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	dly_tsk is not included in the system
E_CTX	−25	dly_tsk/idly_tsk was issued from a non-task context dly_tsk/idly_tsk was issued in the CPU lock state dly_tsk/idly_tsk was issued in the dispatch disabled state
E_RLWAI	−49	The waiting state was forcibly released by (i)rel_wai

13.8.3 Task exception processing function service calls

This section describes the task exception processing function service calls listed in the following table.

Table 13-10. Task Exception Processing Function Service calls

Name	Function
def_tex	Defines a task exception processing routine
ras_tex/iras_tex	Requests a task exception processing routine
dis_tex	Disables task exception processing
ena_tex	Enables task exception processing
sns_tex	References the task exception processing disabled status
ref_tex/iref_tex	Obtains the task exception processing routine status

def_tex

Define Task Exception Routine

[Overview]

Defines a task exception processing routine

[C format]

```
#include <kernel.h>
ER def_tex(ID tskid, T_DTEX *pk_dtex);
```

[Parameters]

I/O	Parameter		Description
I	ID	tskid	ID number of task for which task exception processing routine is to be defined.
I	T_DTEX *	pk_dtex	Address of a task exception processing routine definition packet.

Configuration of T_DTEX

```
typedef struct t_dtex {
        ATR             texatr;         /* Task exception processing routine attribute */
        FP              texrtn;         /* Activate address of task exception processing routine */
} T_DTEX;
```

[Explanation]

A task exception processing routine that belongs to a task specified as tskid is defined based on the information stored in the task exception processing routine definition packet pk_dtex. If TSK_SELF = 0 is set for tskid, the task itself is the target of the service call.

If def_tex is issued again to a task for which an exception processing routine has been already defined, the old definition is canceled and a new exception processing routine is defined. To cancel the definition of an exception processing routine, specify NULL as pk_dtex.

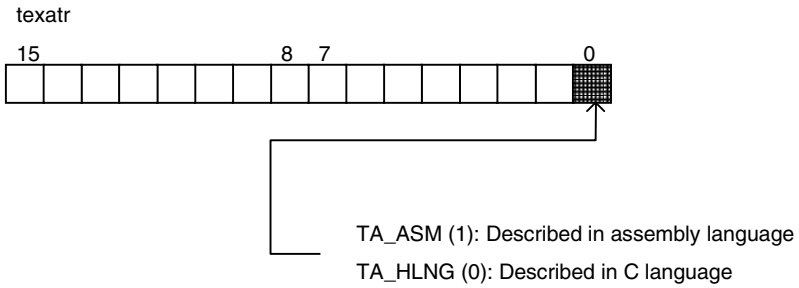
Task exceptions are disabled and the source of a pending exception is also cleared if a new exception processing routine is defined or if the definition of an exception processing routine is canceled. If an exception processing routine is defined again, the task disabling/enabling status and pending exception source remain unchanged.

The task exception processing routine definition information is explained in detail below.

- texatr

  This is the attribute of a task exception processing routine that specifies a language in which the task exception processing routine is described. If the routine is described in C, specify TA_HLNG. If it is described in an assembly language, specify TA_ASM. Regardless of which of these two is described, there are no differences in the processing of the kernel of the current version.

Figure 13-3. Task Exception Processing Routine Attribute



- texrtn

  This is the activate address of the task exception processing routine to be defined.

If the task exception processing routine uses PID (Position Independent Data), it is assumed that its base address gp is the same as that of the task. The exception processing routine takes over the value of the gp register of the task.

The same execution context (stack) of the task exception processing routine as the task is used.

[Differences from μITRON3.0]

def_tex is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	def_tex is not included in the system
E_RSATR	−11	The task exception processing routine attribute is illegal
E_ID	−18	The task ID number is outside the range (tskid < 0, maximum task count < tskid)
E_CTX	−25	def_tex was issued from a non-task context def_tex was issued in the CPU lock state
E_NOEXS	−42	The target task does not exist

Raise Task Exception Routine

ras_tex/iras_tex

[Overview]

Issues a task exception processing request

[C format]

```
#include <kernel.h>
ER ras_tex(ID tskid, TEXPTN rasptn);
```

[Parameters]

I/O	Parameter		Description
I	ID	tskid	ID number of task to which exception processing routine to be activated belongs
I	TEXPTN	rasptn	Task exception processing routine activation source

[Explanation]

A request is issued to the task specified by tskid to activate an exception processing routine. If TSK_SELF = 0 is set for tskid the task itself becomes the target of the service call.

A task exception processing routine is activated when the task enters the running state if the activation request is retained. If ras_tex is issued to the task itself, the exception processing routine is immediately activated. A task exception processing routine that has been activated can receive the bit pattern specified by rasptn as the activation source. If (i)ras_tex is issued more than once to the same task, the exception processing routine can receive the logical sum of the activation sources specified by each (i)ras_tex as the activation source.

iras_tex is intended to be issued from a non-task context but it can also be issued from a task context. ras_tex can be issued from a non-task context.

[Differences from μITRON3.0]

ras_tex/iras_tex is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ras_tex/iras_tex is not included in the system
E_PAR	−17	The activation source is 0 (rasptn = 0)
E_ID	−18	The task ID number is outside the range (tskid < 0, maximum task count < tskid) TSK_SELF is specified and issued from a non-task context.
E_CTX	−25	ras_tex/iras_tex was issued in the CPU lock state
E_OBJ	−41	The target task is in the dormant state No exception processing routine is defined for the target task
E_NOEXS	−42	The target task does not exist

Disable Task Exception

dis_tex

[Overview]

Disables activation of a task exception processing routine

[C format]

```
#include <kernel.h>
ER dis_tex(void);
```

[Parameter]

None

[Explanation]

Activation of the task exception processing routine of the task itself, i.e., the task that has issued this service call is disabled. Consequently, the task exception processing routine will not be activated until ena_tex is issued, and all activation requests issued by (i)ras_tex in the meantime are held pending.

Even if dis_tex is issued again while the task exception processing routine is disabled, the disabled status merely continues and no error occurs. Even if dis_tex is issued two times or more, however, disabling the activation of the task exception processing routine is canceled by issuing ena_tex only once.

[Differences from μITRON3.0]

dis_tex is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	dis_tex is not included in the system
E_CTX	−25	dis_tex was issued from a non-task context dis_tex was issued in the CPU lock state
E_OBJ	−41	No task exception processing routine is defined

Enable Task Exception

ena_tex

[Overview]

Enables activation of a task exception processing routine

[C format]

```
#include <kernel.h>
ER ena_tex(void);
```

[Parameter]

None

[Explanation]

Activation of the task exception processing routine of the task itself, i.e., the task that has issued this service call is enabled. Consequently, the task exception processing routine is immediately activated if a request to activate the task exception processing routine is retained by the task itself.

Even if ena_tex is issued again while the task exception processing routine is disabled, the enabled status merely continues and no error occurs. Even if ena_tex is issued more than once, however, activation of the task exception processing routine is prohibited by issuing dis_tex once.

[Differences from μITRON3.0]

ena_tex is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ena_tex is not included in the system
E_CTX	−25	ena_tex was issued from a non-task context ena_tex was issued in the CPU lock state
E_OBJ	−41	No task exception processing routine is defined

Sense Task Exception Routine Status

sns_tex

[Overview]

References the status of a task exception processing routine

[C format]

```
#include <kernel.h>
BOOL sns_tex(void);
```

[Parameter]

None

[Explanation]

The value of TRUE or FALSE is returned depending on the status of the task exception processing routine of the task in the running state, i.e., the task currently under execution (the issuing task), or of the task that is interrupted by an exception during execution.

For the meaning of the value returned, refer to **[Return values]** below. The type of the return value is BOOL. Usually, TRUE(1) or FALSE(0) is returned. If sns_tex is not included in the system, however, E_RSFN (−10) may be returned as an exception.

sns_tex can be always issued regardless of the status of the context.

[Differences from μITRON3.0]

sns_tex is a newly created service call.

[Return values]

Symbol	Value	Meaning
TRUE	1	Task exception processing is disabled No task exception processing routine is defined No task is in the running state
FALSE	0	Task exception processing is enabled
E_RSFN	−10	sns_tex is not included in the system

Refer Task Exception Routine Status

ref_tex/iref_tex

[Overview]

Obtains task exception processing routine information

[C format]

```
#include <kernel.h>
ER ref_tex(ID tskid, T_RTEX *pk_rtex);
```

[Parameters]

I/O	Parameter		Description
I	ID	tskid	ID number of task from which exception processing routine information is obtained
O	T_RTEX *	pk_rtex	Address of task exception processing routine information packet

Configuration of T_RTEX

```
typedef struct t_rtex {
        STAT        texstat;    /* Status of task exception processing routine */
        TEXPTN      pndptn;     /* Pending exception source */
        ATR         texatr;     /* Task exception processing routine attribute */
} T_RTEX;
```

[Explanation]

Information on the task exception processing routine of the task specified as tskid is obtained and stored in the packet specified by pk_rtex. If TSK_SELF = 0 is set for taskid the task itself becomes the target of the service call.

The task exception processing routine information is described in detail below.

- texstat
  Indicates the current status of the task exception processing routine (i.e., whether the routine is enabled or disabled). The routine is disabled if TTEX_DIS is stored for this parameter and enabled if TTEX_ENA is stored.
- pndptn
  Stores the exception source of the pending task exception processing routine.
- texatr
  Stores the attribute of the task exception processing routine. For the meaning of the value stored, refer to the description of def_tex.

iref_tex is intended to be issued from a non-task context but it can also be issued from a task context. ref_tex can be issued from a non-task context.

[Differences from μITRON3.0]

ref_tex/iref_tex is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ref_tex/iref_tex is not included in the system
E_ID	−18	The task ID number is outside the range (tskid < 0, maximum task count < tskid) TSK_SELF is specified and issued from a non-task context.
E_CTX	−25	ref_tex/iref_tex was issued in the CPU lock state
E_OBJ	−41	The target task is in the dormant state No exception processing routine is defined for the target task
E_NOEXS	−42	The target task does not exist

13.8.4 Synchronization/communication management function service calls

This section describes the synchronization/communication management function service calls listed in the following table.

Table 13-11. Synchronization/Communication Management Function Service calls (1/2)

Name	Function
cre_sem	Creates a semaphore
acre_sem	Creates a semaphore (automatic assignment of ID No.)
del_sem	Deletes a semaphore
sig_sem/isig_sem	Returns resources
wai_sem	Acquires resources
pol_sem/ipol_sem	Acquires resources (polling)
twai_sem	Acquires resources from a semaphore (with timeout)
ref_sem/iref_sem	Obtains semaphore information
cre_flg	Creates an event flag
acre_flg	Creates an event flag (automatic assignment of ID No.)
del_flg	Deletes an event flag
set_flg/iset_flg	Sets an event flag
clr_flg/iclr_flg	Clears an event flag
wai_flg	Waits for satisfaction of condition
pol_flg/ipol_flg	Waits for satisfaction of condition (polling)
twai_flg	Waits for satisfaction of condition (with timeout)
ref_flg/iref_flg	Obtains event flag information
cre_dtq	Creates a data queue
acre_dtq	Creates a data queue (automatic assignment of ID No.)
del_dtq	Deletes a data queue
snd_dtq	Sends data
psnd_dtq/ipsnd_dtq	Sends data (polling)
tsnd_dtq	Sends data (with timeout)
fsnd_dtq/ifsnd_dtq	Forcibly sends data
rcv_dtq	Receives data
prcv_dtq/iprcv_dtq	Receives data (polling)
trcv_dtq	Receives data (with timeout)
ref_dtq/iref_dtq	Obtains data queue information
cre_mbx	Creates a mailbox
acre_mbx	Creates a mailbox (automatic assignment of ID No.)
del_mbx	Deletes a mailbox
snd_mbx/isnd_mbx	Sends mail
rcv_mbx	Receives mail

Table 13-11. Synchronization/Communication Management Function Service calls (2/2)

Name	Function
prcv_mbx/iprcv_mbx	Receives mail (polling)
trcv_mbx	Receives mail (with timeout)
ref_mbx/iref_mbx	Obtains mailbox information
cre_mtx	Creates a mutex
acre_mtx	Creates a mutex (automatic assignment of ID No.)
del_mtx	Deletes a mutex
loc_mtx	Locks a mutex
ploc_mtx	Locks a mutex (polling)
tloc_mtx	Locks a mutex (with timeout)
unl_mtx	Unlocks a mutex
ref_mtx/iref_mtx	Obtains mutex information

cre_sem

Create Semaphore

[Overview]

Creates a semaphore

[C format]

```
#include <kernel.h>
ER cre_sem(ID semid, T_CSEM *pk_csem);
```

[Parameters]

I/O	Parameter		Description
I	ID	semid	ID number of semaphore to be created
I	T_CSEM *	pk_csem	Address of semaphore creation packet

Configuration of T_CSEM

```
typedef struct t_csem {
        ATR     sematr;         /* Semaphore attribute */
        UINT    isemcnt;        /* Default value of semaphore */
        UINT    maxsem;         /* Maximum value of semaphore */
} T_CSEM;
```

[Explanation]

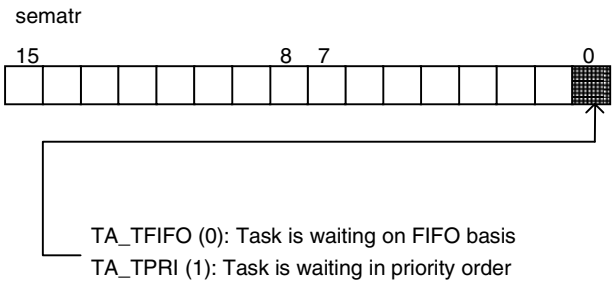
A semaphore with the ID number specified by semid is created based on the information stored in the packet pk_csem. In other words, a control block is assigned to the semaphore to be created, and the semaphore is initiarized.

The semaphore creation packet T_CSEM is explained in detail below.

- sematr

  Specifies the order in which a task waits for the resources of a semaphore as the attribute of the semaphore. The values that can be specified as the semaphore attribute are as follows:

Figure 13-4. Semaphore Attribute



To bit 0 of sematr, specify the order in which a task waits for the resources from the semaphore to be created. If TA_TFIFO is specified, the task waits on an FIFO basis. If TA_TPRI is specified, the task waits in the order of priority.

- isemcnt

  Specifies the default value of the resource counter of the semaphore as an integer of 0 or more. The maximum value that can be specified is TMAX_MAXSEM (0x7fffffff). However, the value of maxsem explained below must not be exceeded. If a value greater than maxsem is specified, an error occurs and the error code E_PAR is returned.
- maxsem

  Specifies the maximum value of the resource counter of the semaphore as an integer of 1 or more. The maximum value that can be specified is TMAX_MAXSEM (0x7fffffff). If the resource counter issues (i)sig_sem that exceeds this value, an error occurs.

[Differences from μITRON3.0]

1. The extended data exinf has been deleted from the members of the semaphore creation packet T_CSEM.
2. The type of the default value of the resource counter, isemcnt, and the maximum value of the resource counter, maxsem, has been changed from INT to UINT.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	cre_sem is not included in the system
E_RSATR	−11	The semaphore attribute is illegal
E_PAR	−17	The parameter is illegal − The default value of the semaphore is outside the range (maxsem < isemcnt) − The maximum value of the semaphore is outside the range (maxsem = 0)
E_ID	−18	The semaphore ID number is outside the range (semid ≤ maximum semaphore count < semid)
E_CTX	−25	cre_sem was issued from a non-task context cre_sem was issued in the CPU lock state
E_OBJ	−41	A semaphore with the same ID number already exists

acre_sem

Create Semaphore with Automatic ID Assignment

[Overview]

Creates a semaphore (Automatic assignment of ID number).

[C format]

```
#include <kernel.h>
ER_ID acre_sem(T_CSEM *pk_csem);
```

[Parameter]

I/O	Parameter		Description
I	T_CSEM *	pk_csem	Address of semaphore creation packet

[Explanation]

A semaphore is created based on the semaphore creation information stored in pk_csem, and the ID number of the semaphore is returned. In other words, a semaphore control block that is available but is not created is searched, assigned, and initialized, and its ID number returned. If the return value is negative, an error occurs. For the meaning of the return value, refer to **[Return values]** below.

For the details of the semaphore creation packet, refer to the description of cre_sem.

[Differences from μITRON3.0]

acre_sem is a newly created service call that is equivalent to cre_sem, but with the addition of an automatic ID number assignment function.

[Return values]

Symbol	Value	Meaning
(Positive integer)		ID number of created semaphore (normal termination)
E_RSFN	−10	acre_sem is not included in the system
E_RSATR	−11	The semaphore attribute is illegal
E_PAR	−17	The parameter is illegal – The default value of the semaphore is outside the range (maxsem < isemcnt) – The maximum value of the semaphore is outside the range (maxsem = 0)
E_CTX	−25	acre_sem was issued from a non-task context acre_sem was issued in the CPU lock state
E_NOID	−34	The ID number cannot be assigned (reserving a control block has failed)

del_sem

Delete Semaphore

[Overview]

Deletes a semaphore

[C format]

```
#include <kernel.h>
ER del_sem(ID semid);
```

[Parameter]

I/O	Parameter		Description
I	ID	semid	ID number of semaphore to be deleted

[Explanation]

The semaphore for which the semaphore control block specified by semid has been invalidated is deleted. After deletion, a semaphore can be created using the same ID number.

If tasks are waiting for the semaphore, all the tasks are released from the waiting state. The value E_DLT indicating that the semaphore has been deleted is returned for the error code of the service call (t)wai_sem that caused the tasks to be released from the waiting state.

A semaphore can be also deleted even if a task that is waiting for the resource from the semaphore exists. Note that, at this time, deletion of the semaphore is not reported to the waiting task.

[Differences from μITRON3.0]

None

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	del_sem is not included in the system
E_ID	−18	The semaphore ID number is outside the range (semid ≤ 0, maximum semaphore count < semid)
E_CTX	−25	del_sem was issued from a non-task context del_sem was issued in the CPU lock state
E_NOEXS	−42	The target semaphore does not exist

Signal Semaphore

sig_sem/isig_sem

[Overview]

Returns a resource to a semaphore

[C format]

```
#include <kernel.h>
ER sig_sem(ID semid);
```

[Parameter]

I/O	Parameter		Description
I	ID	semid	ID number of semaphore to which resource is to be returned

[Explanation]

A resource is returned to the semaphore specified by semid.

If tasks are waiting for resources from the semaphore, the first task in the waiting task queue is released from the waiting state. In this case, the value of the resource counter of the semaphore is not changed. If no task is waiting for resources, the value of the resource counter is incremented by 1. If (i)sig_sem is issued in such a manner that the value of the counter exceeds the maximum value specified when the semaphore was created, an error occurs and the error code E_QOVR is returned.

isig_sem is intended to be issued from a non-task context but it can also be issued from a task context. sig_sem can be issued from a non-task context.

[Differences from μITRON3.0]

None

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	sig_sem/isig_sem is not included in the system
E_ID	−18	The semaphore ID number is outside the range (semid ≤ 0, maximum semaphore count < semid)
E_CTX	−25	sig_sem/isig_sem was issued in the CPU lock state
E_NOEXS	−42	The target semaphore does not exist
E_QOVR	−43	The counter value of the semaphore has exceeded the maximum value

wai_sem

Wait on Semaphore

[Overview]

Acquires a resource from a semaphore

[C format]

```
#include <kernel.h>
ER wai_sem(ID semid);
```

[Parameter]

I/O	Parameter		Description
I	ID	semid	ID number of semaphore from which resource is to be acquired

[Explanation]

A resource is acquired from the semaphore specified by semid.

If the value of the resource counter of the target semaphore is 1 or greater, the counter value is decremented by 1 and the task continues processing. If the counter value is 0, the task enters the resource waiting state and is registered to the task queue in the waiting order (on an FIFO basis or according to priority) specified when the semaphore was created.

Tasks placed in the waiting state due to wai_sem are released from waiting by one of the following occurrences.

1. (i)sig_sem is issued and the task acquires a resource (E_OK).
2. The task is forcibly released from the waiting state by (i)rel_wai (E_RLWAI).
3. del_sem is issued and the semaphore is deleted (E_DLT).

[Differences from μITRON3.0]

None

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	wai_sem is not included in the system
E_ID	−18	The semaphore ID number is outside the range (semid ≤ 0, maximum semaphore count < semid)
E_CTX	−25	wai_sem was issued from a non-task context wai_sem was issued in the CPU lock state wai_sem was issued in the dispatch disabled state
E_NOEXS	−42	The target semaphore does not exist
E_RLWAI	−49	The task is released from the waiting state by (i)rel_wai
E_DLT	−51	The semaphore is deleted while a task is waiting

pol_sem/ipol_sem

Poll Semaphore

[Overview]

Acquires a resource from a semaphore (polling)

[C format]

```
#include <kernel.h>
ER pol_sem(ID semid);
```

[Parameter]

I/O	Parameter		Description
I	ID	semid	ID number of semaphore from which resource is to be acquired

[Explanation]

A resource is acquired from the semaphore specified by semid.

If the value of the resource counter of the target semaphore is 1 or greater, the counter value is decremented by 1 and the task continues processing. If the counter value is 0, polling fails and the error code E_TMOUT is returned.

ipol_sem is intended to be issued from a non-task context but it can also be issued from a task context. pol_sem can be issued from a non-task context.

[Differences from μITRON3.0]

The name of preq_sem has been changed to pol_sem.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	pol_sem/ipol_sem is not included in the system
E_ID	−18	The semaphore ID number is outside the range (semid ≤ 0, maximum semaphore count < semid)
E_CTX	−25	pol_sem/ipol_sem was issued in the CPU lock state
E_NOEXS	−42	The target semaphore does not exist
E_TMOUT	−50	Polling has failed

twai_sem

Wait on Semaphore with Timeout

[Overview]

Acquires a resource from a semaphore (with timeout)

[C format]

```
#include <kernel.h>
ER twai_sem(ID semid, TMO tmout);
```

[Parameters]

I/O	Parameter		Description
I	ID	semid	ID number of semaphore from which resource is to be acquired
I	TMO	tmout	Timeout time [milliseconds]

[Explanation]

A resource is acquired from the semaphore specified by semid.

If the value of the resource counter of the target semaphore is 1 or greater, the counter value is decremented by 1 and the task continues processing. If the counter value is 0, the task waits for a resource for the time specified as tmout, and is registered to the task queue in the waiting mode (on an FIFO basis or according to priority) specified when the semaphore was created.

If TMO_POL = 0 is specified as tmout, 0 is specified as the timeout time and this service call performs the same operation as pol_sem. If TMO_FEVR = −1 is specified as tmout, the timeout time is specified to be inifite and the service call performs the same operation as wai_sem.

Tasks placed in the waiting state due to the issuance of twai_sem are released from waiting by one of the following occurrences.

1. (i)sig_sem is issued and the task acquires a resource (E_OK).
2. The specified time has elapsed (timeout) (E_TMOUT).
3. The task is forcibly released from the waiting state by (i)rel_wai (E_RLWAI).
4. del_sem is issued and the semaphore is deleted (E_DLT).

[Differences from μITRON3.0]

None

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	twai_sem is not included in the system
E_PAR	−17	The timeout time is outside the range (tmout < TMO_FEVR = −1)
E_ID	−18	The semaphore ID number is outside the range (semid ≤ 0, maximum semaphore count < semid)
E_CTX	−25	twai_sem was issued from a non-task context twai_sem was issued in the CPU lock state twai_sem was issued in the dispatch disabled state
E_NOEXS	−42	The target semaphore does not exist
E_RLWAI	−49	The task has been released from the waiting state by (i)rel_wai
E_TMOUT	−50	The specified time has elapsed
E_DLT	−51	The semaphore is deleted while a task is waiting for it

ref_sem/iref_sem

Refer Semaphore Status

[Overview]

Obtains semaphore information

[C format]

```
#include <kernel.h>
ER ref_sem(ID semid, T_RSEM *pk_rsem);
```

[Parameters]

I/O	Parameter		Description
I	ID	semid	ID number of semaphore whose information is to be obtained
O	T_RSEM *	pk_rsem	Address of semaphore information packet

Configuration of T_RSEM

```
typedef struct t_rsem {
        ID          wtskid;     /* Presence/absence of waiting task */
        UINT        semcnt;     /* Current number of resources of semaphore */
        ATR         sematr;     /* Semaphore attribute */
        UINT        maxsem;     /* Maximum number of resources of semaphore */
} T_RSEM;
```

[Explanation]

Information on the semaphore specified by semid is stored in the packet specified by pk_rsem.

wtskid indicates whether there is a task waiting for resources from the target semaphore. If a waiting task exists, the ID number of the first task in the waiting task queue is stored in wtskid.

If there is no waiting task, TSK_NONE = 0 is stored in wtskid.

semcnt stores the current value of the resource counter of the semaphore.

sematr stores the semaphore attribute. For the meaning of the value stored in sematr, refer to the description of cre_sem.

maxsem stores the maximum value of the resource counter of the semaphore.

iref_sem is intended to be issued from a non-task context but it can also be issued from a task context. ref_sem can be issued from a non-task context.

[Differences from μITRON3.0]

1. The order of the parameters has been changed.
2. The extended data exinf has been deleted from the members of the semaphore information packet T_RSEM.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ref_sem/iref_sem is not included in the system
E_ID	−18	The semaphore ID number is outside the range (semid ≤ 0, maximum semaphore count < semid)
E_CTX	−25	ref_sem/iref_sem was issued from a non-task context ref_sem/iref_sem was issued in the CPU lock state
E_NOEXS	−42	The target semaphore does not exist

cre_flg

Create Eventflag

[Overview]

Creates an event flag

[C format]

```
#include <kernel.h>
ER cre_flg(ID flgid, T_CFLG *pk_cflg);
```

[Parameters]

I/O	Parameter		Description
I	ID	flgid	ID number of event flag to be created
I	T_CFLG *	pk_cflg	Address of event flag creation packet

Configuration of T_CFLG

```
typedef struct t_cflg {
        ATR             flgatr;         /* Event flag attribute */
        FLGPTN          iflgptn;        /* Default value of event flag */
} T_CFLG;
```

[Explanation]

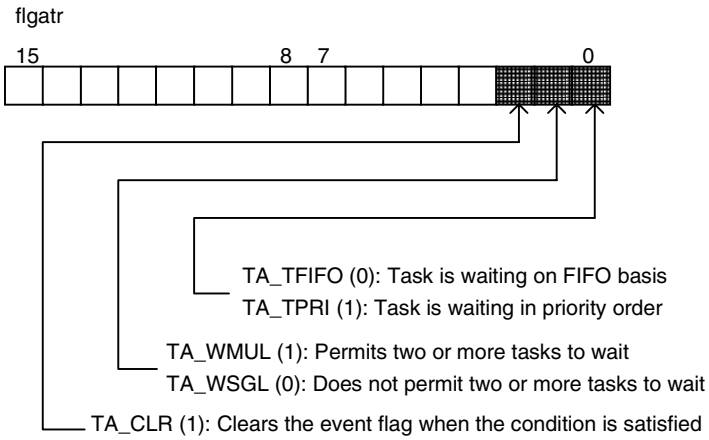
The event flag with the ID number specified by flgid is created, based on the information stored in the packet pk_cflg. In other words, a control block is assigned to the event flag to be created, and the event flag is initialized.

The event flag creation packet T_CFLG is described in detail below.

- flgatr

  Specifies the order in which a task waits for the event flag to be created as the attribute of the event flag. The values that can be specified as the event flag attribute are as follows:

Figure 13-5. Event Flag Attribute



To bit 0 of flgatr, specify the order in which a task waits. If TA_TFIFO is specified, the task waits on an FIFO basis. If TA_TPRI is specified, the task waits in the order of priority. These attributes, however, are meaningless if the attribute of the event flag is TA_WSGL, which is explained below.

Bit 1 specifies whether two or more tasks are permitted to wait for the event flag. If TA_WMUL is specified, two or more tasks can wait for the event flag. If TA_WSGL is specified, two or more tasks cannot wait.

Bit 2 specifies whether the bit pattern is created when the condition of the event flag is satisfied. If attribute TA_CLR is given, the event flag is cleared to 0 when the condition of the event flag has been satisfied and the task has been released from the waiting state.

- iflgptn

  Stores the default value of the event flag to be created.

[Differences from μITRON3.0]

1. The extended data exinf has been deleted from the members of the event flag creation packet T_CFLG.
2. The type of the default value of the event flag iflgptn has been changed from INT to FLGPTN. FLGPTN is a type newly defined for μITRON4.0.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	cre_flg is not included in the system
E_RSATR	−11	The event flag attribute is illegal
E_ID	−18	The event flag ID number is outside the range (flgid ≤ 0, maximum event flag count < flgid)
E_CTX	−25	cre_flg was issued from a non-task context cre_flg was issued in the CPU lock state
E_OBJ	−41	An event flag with the same ID number already exists

acre_flg

Create Eventflag with Automatic ID Assignment

[Overview]

Creates an event flag (Automatic assignment of ID number)

[C format]

```
#include <kernel.h>
ER_ID acre_flg(T_CFLG *pk_cflg);
```

[Parameter]

I/O	Parameter		Description
I	T_CFLG *	pk_cflg	Address of event flag creation packet

[Explanation]

An event flag is created based on the event flag creation information stored in pk_cflg, and the ID number of the event flag is returned. In other words, an event flag control block that is available but is not created is searched, assigned and initialized, and its ID number returned. If the return value is negative, an error occurs. For the meaning of the return value, refer to **[Return values]** below.

For the details of the event flag creation packet, refer to the description of cre_flg.

[Differences from μITRON3.0]

acre_flg is a newly created service call that is equivalent to cre_flg, but with the addition of an automatic ID number assignment function.

[Return values]

Symbol	Value	Meaning
(Integer of 1 or greater)		ID number of created event flag (normal termination)
E_RSFN	−10	acre_flg is not included in the system
E_RSATR	−11	The event flag attribute is illegal
E_CTX	−25	acre_flg was issued from a non-task context acre_flg was issued in the CPU lock state
E_NOID	−34	The ID number cannot be assigned (reserving a control block has failed)

del_flg

Delete Eventflag

[Overview]

Deletes an event flag

[C format]

```
#include <kernel.h>
ER del_flg(ID flgid);
```

[Parameter]

I/O	Parameter		Description
I	ID	flgid	ID number of event flag to be deleted

[Explanation]

The control block of the event flag specified by flgid is invalidated and the specified event flag is deleted. After deletion, an event flag can be created by using the same ID number.

If tasks are waiting for the event flag, all the tasks are released from the waiting state. The value E_DLT indicating that the event flag has been deleted is returned for the error code of the service call (t)wai_flg, which caused the tasks to be released from the waiting state.

[Differences from μITRON3.0]

None

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	del_flg is not included in the system
E_ID	−18	The event flag ID number is outside the range (flgid ≤ 0, maximum event flag count < flgid)
E_CTX	−25	del_flg was issued from a non-task context del_flg was issued in the CPU lock state
E_NOEXS	−42	The target event flag does not exist

Set Eventflag

set_flg/iset_flg

[Overview]

Sets an event flag

[C format]

```
#include <kernel.h>
ER set_flg(ID flgid, FLGPTN setptn);
```

[Parameters]

I/O	Parameter		Description
I	ID	flgid	ID number of event flag to be set
I	FLGPTN	setptn	Bit pattern to be set

[Explanation]

The bit pattern specified by setptn is set to the event flag specified by flgid. After setting, the bit pattern is the logical sum of the previously set bit pattern and setptn.

If a task waiting for the target event flag exists and the condition for releasing the task from the waiting state is satisfied by issuance of set_flg, the task is released from the waiting state. For details of the conditions under which a task is released from the waiting state, refer to the description of wai_flg.

iset_flg is intended to be issued from a non-task context but it can also be issued from a task context. set_flg can be issued from a non-task context.

[Differences from μITRON3.0]

The type of the bit pattern setptn to be set has been changed from UINT to FLGPTN. FLGPTN is a type newly defined for μITRON4.0.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	set_flg/iset_flg is not included in the system
E_ID	−18	The event flag ID number is outside the range (flgid ≤ 0, maximum event flag count < flgid)
E_CTX	−25	set_flg/iset_flg was issued in the CPU lock state
E_NOEXS	−42	The target event flag does not exist

clr_flg/iclr_flg

Clear Eventflag

[Overview]

Clears an event flag

[C format]

```
#include <kernel.h>
ER clr_flg(ID flgid, FLGPTN clrptn);
```

[Parameters]

I/O	Parameter		Description
I	ID	flgid	ID number of event flag to be cleared
I	FLGPTN	clrptn	Bit pattern to be cleared

[Explanation]

The bit specified by clrptn of the event flag specified by flgid is cleared. After clearing, the bit pattern is the logical sum of the previously set bit pattern and clrptn.

If a task waiting for the target event flag exists, the task waits until the specified bit is set. Therefore, it is not released from the waiting state even when the condition under which the task is released from the waiting state is satisfied by issuance of clr_flg.

iclr_flg is intended to be issued from a non-task context but it can also be issued from a task context. clr_flg can be issued from a non-task context.

[Differences from μITRON3.0]

The type of the bit pattern setptn to be set has been changed from UINT to FLGPTN. FLGPTN is a type newly defined for μITRON4.0.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	clr_flg/iclr_flg is not included in the system
E_ID	−18	The event flag ID number is outside the range (flgid ≤ 0, maximum event flag count < flgid)
E_CTX	−25	clr_flg/iclr_flg was issued from a non-task context clr_flg/iclr_flg was issued in the CPU lock state
E_NOEXS	−42	The target event flag does not exist

wai_flg

Wait Eventflag

[Overview]

Waits until the condition of an event flag is satisfied

[C format]

```
#include <kernel.h>
ER wai_flg(ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn);
```

[Parameters]

I/O	Parameter		Description
I	ID	flgid	ID number of event flag whose condition is to be satisfied
I	FLGPTN	waiptn	Waiting bit pattern
I	MODE	wfmode	Waiting mode
O	FLGPTN *	p_flgptn	Address to which bit pattern is stored when task has been released from waiting state

[Explanation]

A task is kept waiting in the waiting mode specified by wfmode until the bit pattern specified by waiptn of the event flag specified by flgid is set. If the bit pattern of the target event flag has already satisfied the condition under which the task is released from the wait state, the task is not kept waiting but continues processing.

The waiting mode wfmode specifies how the task waits, by the following values:

Name	Value	Meaning	Expression *flgptn = Current Bit Pattern
TWF_ANDW	H'0000	AND wait	flgptn & waiptn == waiptn
TWF_ORW	H'0001	OR wait	flgptn & waiptn != 0

If TWF_ANDW is specified, the task waits until all the bits specified by waiptn of the bit pattern of the event flag are set (AND wait). If TWF_ORW is specified, the task waits until any of the bits specified by waiptn of the bit pattern of the event flag is set (OR wait).

To address p_flgptn, the bit pattern of the event flag when the condition is satisfied is stored.

Some event flags allow two or more tasks to wait (even flags with the TA_WMUL attribute) while the others do not (event flags with the TA_WSGL attribute). If wai_flg is issued to an event flag with the TA_WSGL attribute for which a task is waiting, the service call results in an error regardless of whether the condition is satisfied or not, and the error code E_OBJ is returned.

For an event flag with the TA_WMUL attribute, the order in which waiting tasks are registered to the queue is determined by the attribute of the event flag. If the attribute is TA_TFIFO, the tasks wait on an FIFO basis. If the attribute is TA_TPRI, the tasks wait according to their priorities. When set_flg is issued, however, two or more tasks may be released from the waiting state all at once because it is checked whether the condition under which the task is released from the waiting state is satisfied or not. Therefore, the first task in the queue is not always released from the waiting state first. If two or more tasks are released from the waiting state and if the event flag has the attribute TA_CLR, the event flag is cleared as soon as the first task has been released from the waiting state. Consequently, the bit pattern (0) is compared with the waiting bit pattern, and the processing to release the task from the waiting state is completed as soon as the bit pattern has been compared.

Tasks placed in the waiting state due to wai_flg are released from waiting by one of the following occurrences.

1. (i)set_flg is issued and the event flag satisfies the condition (E_OK).
2. (i)rel_wai is issued and the task is forcibly released from the waiting state (E_RLWAI).
3. del_flg is issued and the event flag is deleted (E_DLT).

[Differences from μITRON3.0]

1. The order of the parameters has been changed.
2. The type of the bit pattern of the event flag has been changed from UINT to FLGPTN.
3. The type of the waiting mode wfmode has been changed from UINT to MODE.
   FLGPTN and MODE are types newly defined for μITRON4.0.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	wai_flg is not included in the system
E_PAR	−17	The parameter is illegal – The waiting bit pattern is 0 (waiptn = 0) – The waiting mode specification is illegal
E_ID	−18	The event flag ID number is outside the range (flgid ≤ 0, maximum event flag count < flgid)
E_CTX	−25	wai_flg was issued from a non-task context wai_flg was issued in the CPU lock state wai_flg was issued in the dispatch disabled state
E_OBJ	−41	A waiting task already exists (when the attribute is TA_WSGL)
E_NOEXS	−42	The target event flag does not exist
E_RLWAI	−49	The task has been forcibly released from the waiting state by (i)rel_wai
E_DLT	−51	The event flag is deleted while the task is waiting

Poll Eventflag

pol_flg/ipol_flg

[Overview]

Waits until the condition of an event flag is satisfied (Polling)

[C format]

```
#include <kernel.h>
ER pol_flg(ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn);
```

[Parameters]

I/O	Parameter		Description
I	ID	flgid	ID number of event flag whose condition is to be satisfied
I	FLGPTN	waiptn	Waiting bit pattern
I	MODE	wfmode	Waiting mode
O	FLGPTN *	p_flgptn	Address to which bit pattern is stored when task has been released from waiting state

[Explanation]

The function to place a task in the waiting state is removed from wai_flg. It is completed normally if the target event flag has already satisfied the condition under which the task is released from the wait state. If the condition is not satisfied, pol_flg returns the error code E_TMOUT to indicate that polling has failed. For other details, refer to the description of wai_flg.

A task is not placed in the wait state even if pol_flg is issued. Even if pol_flg is issued to the event flag with the TA_WSGL attribute for which a task is already waiting, an error occurs and the error code E_OBJ is returned, like wai_flg, regardless of whether the condition is satisfied or not.

ipol_flg is intended to be issued from a non-task context but it can also be issued from a task context. pol_flg can be issued from a non-task context.

[Differences from μITRON3.0]

Refer to the description of wai_flg.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	pol_flg/ipol_flg is not included in the system
E_PAR	−17	The parameter is illegal – The waiting bit pattern is 0 (waiptn = 0) – The waiting mode specification is illegal
E_ID	−18	The event flag ID number is outside the range (flgid ≤ 0, maximum event flag count < flgid)
E_CTX	−25	pol_flg/ipol_flg was issued in the CPU lock state
E_OBJ	−41	A waiting task already exists (when the attribute is TA_WSGL)
E_NOEXS	−42	The target event flag does not exist
E_TMOUT	−50	Polling has failed

twai_flg

Wait Eventflag with Timeout

[Overview]

Waits until the condition of an event flag is satisfied (with timeout)

[C format]

```
#include <kernel.h>
ER twai_flg(ID flgid, FLGPTN waiptn, MODE wfmode, FLGPTN *p_flgptn, TMO tmout);
```

[Parameters]

I/O	Parameter		Description
I	ID	flgid	ID number of event flag whose condition is to be satisfied
I	FLGPTN	waiptn	Waiting bit pattern
I	MODE	wfmode	Waiting mode
O	FLGPTN *	p_flgptn	Address to which bit pattern is stored when task has been released from waiting state
I	TMO	tmout	Timeout time [milliseconds]

[Explanation]

twai_flg is a service call that adds a timeout function to wai_flg. It is completed normally if the target event flag has already satisfied the condition under which the task is released from the waiting state. If the condition is not satisfied, the task waits for the time specified by tmout. If the specified time has elapsed without the requested bit pattern being set while the task is waiting, the task is released from the waiting state and the error code E_TMOUT indicating a timeout is returned.

If TMO_POL = 0 is specified as tmout, 0 is specified as the timeout time and the service call performs the same operation as pol_flg. If TMO_FEVR = −1 is specified, the timeout time is specified to be inifite, and the service call performs the same operation as wai_flg.

For other details, refer to the description of wai_flg.

Tasks placed in the waiting state due to twai_flg are released from waiting by one of the following occurrences.

1. (i)set_flg is issued and the event flag satisfies the condition (E_OK).
2. The specified time has elapsed and timeout occurs (E_TMOUT).
3. The task is forcibly released from the waiting state by (i)rel_wai (E_RLWAI).
4. del_flg is issued and the event flag is deleted (E_DLT).

[Differences from μITRON3.0]

1. The order of the parameters has been changed.
2. The type of the bit pattern of the event flag has been changed from UINT to FLGPTN.
3. The type of the waiting mode wfmode has been changed from UINT to MODE.
   FLGPTN and MODE are types newly defined for μITRON4.0.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	twai_flg is not included in the system
E_PAR	−17	The parameter is illegal – The waiting bit pattern is 0 (waiptn = 0) – The waiting mode specification is illegal – The timeout time is illegal (tmout < TMO_FEVR)
E_ID	−18	The event flag ID number is outside the range (flgid ≤ 0, maximum event flag count < flgid)
E_CTX	−25	twai_flg was issued from a non-task context twai_flg was issued in the CPU lock state twai_flg was issued in the dispatch disabled state
E_OBJ	−41	A waiting task already exists (when the attribute is TA_WSGL)
E_NOEXS	−42	The target event flag does not exist
E_RLWAI	−49	The task has been forcibly released from the waiting state by (i)rel_wai
E_TMOUT	−50	Timeout
E_DLT	−51	The event flag is deleted while the task is waiting

Refer Eventflag Status

ref_flg/iref_flg

[Overview]

Obtains event flag information

[C format]

```
#include <kernel.h>
ER ref_flg(ID flgid, T_RFLG *pk_rflg);
```

[Parameters]

I/O	Parameter		Description
I	ID	flgid	ID number of event flag whose information is to be obtained
O	T_RFLG *	pk_rflg	Address of event flag information packet

Configuration of T_RFLG

```
typedef struct t_rflg {
        ID          wtskid;     /* Presence/absence of waiting task */
        FLGPTN      flgptn;     /* Current bit pattern of event flag */
        ATR         flgatr;     /* Event flag attribute */
} T_RFLG;
```

[Explanation]

Information on the event flag specified by flgid is stored in the packet specified by pk_rflg.

wtskid indicates whether there is a task waiting until the condition of the event flag is satisfied. If a waiting task exists, the ID number of the first task in the waiting task queue is stored in wtskid. If there is no waiting task, TSK_NONE = 0 is stored in wtskid. flgptn stores the current bit pattern of the event flag. flgatr stores the attribute of the event flag. For the meaning of the stored value, refer to the description of cre_flg.

iref_flg is intended to be issued from a non-task context but it can also be issued from a task context. ref_flg can be issued from a non-task context.

[Differences from μITRON3.0]

1. The extended data exinf has been deleted from the members of the event flag information packet T_RFLG.
2. The attribute flgatr has been added to the members of the event flag information packet T_RFLG.
3. The type of the current bit pattern flgptn has been changed from UINT to FLGPTN.
   FLGPTN is a type newly defined for μITRON4.0.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ref_flg/iref_flg is not included in the system
E_ID	−18	The event flag ID number is outside the range (flgid ≤ 0, maximum event flag count < flgid)
E_CTX	−25	ref_flg/iref_flg was issued from a non-task context ref_flg/iref_flg was issued in the CPU lock state
E_NOEXS	−42	The target event flag does not exist

cre_dtq

Create Data Queue

[Overview]

Creates a data queue

[C format]

```
#include <kernel.h>
ER cre_dtq(ID dtqid, T_CDTQ *pk_cdtq);
```

[Parameters]

I/O	Parameter		Description
I	ID	dtqid	ID number of data queue to be created
I	T_CDTQ *	pk_cdtq	Address of data queue creation packet

Configuration of T_CDTQ

```
typedef struct t_cdtq {
        ATR             dtqatr;         /* Data queue attribute */
        UINT            dtqcnt;         /* Number of ring buffers */
        VP              dtq;            /* Reserved area */
} T_CDTQ;
```

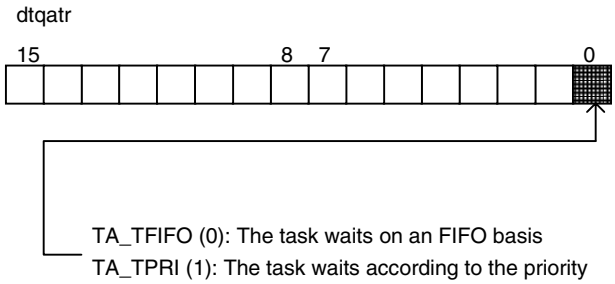
[Explanation]

The data queue with the ID number specified by dtqid is created based on the information stored in the packet pk_cdtq. In other words, a control block is assigned to the data queue to be created, and an area of the ring buffer used by the data queue is reserved from the user pool.

The data queue creation packet T_CDTQ is explained in detail below.

- dtqatr
  Specifies how a task is to wait in the data queue as the data queue attribute. The values that can be specified as the data queue attribute is as follows:

Figure 13-6. Data Queue Attribute



Bit 0 specifies how the task waits in the data queue for data transmission. If TA_TFIFO is specified, the task waits on an FIFO basis. If TA_TPRI is specified, it waits according to the priority. The waiting mode of the task when it waits for data reception is fixed to the FIFO basis, regardless of the data queue attribute.

- dtqcnt
  Specifies the size of the ring buffer used by the data queue to be created by the number of buffers. The size of one buffer is sizeof (VP_INT), and the size of the area actually reserved from the user pool, including the size of the header for management, is calculated by the macro TSZ_DTQ (dtqcnt).
  By setting dtqcnt to 0, a data queue that executes communication without buffers and by using only task waiting can be created.
- dtq
  This is a reserved area and must be always set to NULL. Even if this is set to other than NULL, it is ignored.

[Differences from μITRON3.0]

cre_dtq is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	cre_dtq is not included in the system
E_RSATR	−11	The data queue attribute is illegal
E_ID	−18	The data queue ID number is outside the range (dtqid ≤ 0, maximum data queue count < dtqid)
E_CTX	−25	cre_dtq was issued from a non-task context cre_dtq was issued in the CPU lock state
E_NOMEM	−33	The area of the ring buffer cannot be reserved
E_OBJ	−41	A data queue with the same ID number already exists

acre_dtq

Create Data Queue with Automatic ID Assignment

[Overview]

Creates a data queue (Automatic assignment of ID number)

[C format]

```
#include <kernel.h>
ER_ID acre_dtq(T_CDTQ *pk_cdtq);
```

[Parameter]

I/O	Parameter		Description
I	T_CDTQ *	pk_cdtq	Address of data queue creation packet

[Explanation]

A data queue is created based on the information stored in the packet pk_cdtq and the ID number of the data queue is returned. In other words, a data queue control block that is available but has not been created yet is searched, assigned, and initialized, the area of ring buffers is reserved from the user pool, and the ID number of the data queue is returned. If the return value is negative, an error occurs. For the meaning of the return value, refer to **[Return values]** below.

For the details of the data queue creation packet, refer to the description of cre_dtq.

[Differences from μITRON3.0]

acre_dtq is a newly created service call equivalent to cre_dtq, but with the addition of an automatic ID number assignment function.

[Return values]

Symbol	Value	Meaning
(Positive integer)		ID number of created data queue (normal termination)
E_RSFN	−10	acre_dtq is not included in the system
E_RSATR	−11	The data queue attribute is illegal
E_CTX	−25	acre_dtq was issued from a non-task context acre_dtq was issued in the CPU lock state
E_NOMEM	−33	The area of the ring buffer cannot be reserved
E_NOID	−34	The ID number cannot be assigned (reserving the control block has failed)

del_dtq

Delete Data Queue

[Overview]

Deletes a data queue

[C format]

```
#include <kernel.h>
ER del_dtq(ID dtqid);
```

[Parameter]

I/O	Parameter		Description
I	ID	dtqid	ID number of data queue to be deleted

[Explanation]

The control block of the data queue specified by dtqid is invalidated, the area of the ring buffer used by the data queue is returned to the user pool, and the data queue is deleted. After deletion, a new data queue can be created by using the same ID number.

If waiting tasks exist in the specified data queue, all the tasks are released from the waiting state. The value E_DLT indicating that the data queue has been deleted is returned for the error code of the service calls (t)snd_dtq and (t)rcv_dtq that caused the tasks to be kept waiting. Because the data queue can be also deleted even if data that has not been received is stored in the data queue, this data is lost as soon as the data queue has been deleted.

[Differences from μITRON3.0]

del_dtq is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	del_dtq is not included in the system
E_ID	−18	The data queue ID number is outside the range (dtqid ≤ 0, maximum data queue count < dtqid)
E_CTX	−25	del_dtq was issued from a non-task context del_dtq was issued in the CPU lock state
E_NOEXS	−42	The target data queue does not exist

snd_dtq

Send Data to Data Queue

[Overview]

Transmits data to a data queue

[C format]

```
#include <kernel.h>
ER snd_dtq(ID dtqid, VP_INT data);
```

[Parameters]

I/O	Parameter		Description
I	ID	dtqid	ID number of data queue to which data is to be transmitted
I	VP_INT	data	Transmit data

[Explanation]

The data specified by data is transmitted to the data queue specified by dtqid.

If tasks are waiting in the target data queue for data reception, the first task in the queue is allowed to receive the data and is released from the queue. If no waiting task exists, the data is stored in the ring buffer of the data queue on a FIFO basis.

If the buffer is already filled with the other data, the task waits for data transmission and remains in the waiting state until the buffer has a vacancy and data transmission has been completed. At this time, the task is registered to the waiting task queue of the data queue. Tasks are registered to the queue according to their priority (TA_TPRI attribute) or on an FIFO basis (TA_TFIFO attribute), depending on the attribute specified when the data queue was created.

Tasks placed in the waiting state due to snd_dtq are released from waiting by one of the following occurrences.

1. (t)rcv_dtq or (i)prcv_dtq is issued and the buffer now has a vacancy (E_OK).
2. (i)rel_wai is issued and the task is forcibly released from the waiting state (E_RLWAI).
3. del_dtq is issued and the data queue is deleted (E_DLT).

[Differences from μITRON3.0]

snd_dtq is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	snd_dtq is not included in the system
E_ID	−18	The data queue ID number is outside the range (dtqid ≤ 0, maximum data queue count < dtqid)
E_CTX	−25	snd_dtq was issued from a non-task context snd_dtq was issued in the CPU lock state snd_dtq was issued in the dispatch disabled state
E_NOEXS	−42	The target data queue does not exist
E_RLWAI	−49	The task is forcibly released from the waiting state by (i)rel_wai
E_DLT	−51	The data queue is deleted while the task is waiting

psnd_dtq/ipsnd_dtq

Poll and Send Data to Data Queue

[Overview]

Transmits data to a data queue (polling)

[C format]

```
#include <kernel.h>
ER psnd_dtq(ID dtqid, VP_INT data);
```

[Parameters]

I/O	Parameter		Description
I	ID	dtqid	ID number of data queue to which data is to be transmitted
I	VP_INT	data	Transmit data

[Explanation]

The data specified by data is transmitted to the data queue specified by dtqid.

If tasks are waiting in the target data queue for data reception, the first task in the queue is allowed to receive the data and is released from the queue. If no waiting task exists, the data is stored in the ring buffer of the data queue on a FIFO basis.

If the buffer is already filled with the other data, an error occurs and the error code E_TMOUT is returned.

ipsnd_dtg is intended to be issued from a non-task context but it can also be issued from a task context. psnd_dtg can be issued from a non-task context.

[Differences from μITRON3.0]

psnd_dtq/ipsnd_dtq is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	psnd_dtq/ipsnd_dtq is not included in the system
E_ID	−18	The data queue ID number is outside the range (dtqid ≤ 0, maximum data queue count < dtqid)
E_CTX	−25	psnd_dtq/ipsnd_dtq was issued in the CPU lock state
E_NOEXS	−42	The target data queue does not exist
E_TMOUT	−50	Polling has failed

tsnd_dtq

Send Data to Data Queue with Timeout

[Overview]

Transmits data to a data queue (with timeout)

[C format]

```
#include <kernel.h>
ER tsnd_dtq(ID dtqid, VP_INT data, TMO tmout);
```

[Parameters]

I/O	Parameter		Description
I	ID	dtqid	ID number of data queue to which data is to be transmitted
I	VP_INT	data	Transmit data
I	TMO	tmout	Timeout time [milliseconds]

[Explanation]

The data specified by data is transmitted to the data queue specified by dtqid.

If tasks are waiting in the target data queue for data reception, the first task in the queue is allowed immediately to receive the data and is released from the queue. If no waiting task exists, the data is stored in the ring buffer of the data queue on a FIFO basis.

If the buffer is already filled with the other data, the task waits for data transmission only for the time specified as tmout, and it is kept waiting until the buffer has a vacancy and data transmission is completed, or until the specified time has elapsed. At this time, the task is registered to the waiting task queue of the data queue. The task is registered to the queue according to the priority (TA_TPRI attribute) or on an FIFO basis (TA_TFIFO attribute), depending on the attribute specified when the data queue was created.

If TMO_POL = 0 is specified as tmout, 0 is specified as the timeout time and tsnd_dtq performs the same operation as psnd_dtq. If TMO_FEVR = −1 is specified as tmout, the timeout time is specified to be infinite and this service call performs the same operation as snd_dtq.

Tasks placed in the waiting state due to tsnd_dtq are released from waiting by one of the following occurrences.

1. (t)rcv_dtq or (i)prcv_dtq is issued and the buffer now has a vacancy (E_OK).
2. The specified time has elapsed and therefore timeout has occurred (E_TMOUT).
3. (i)rel_wai is issued and the task is forcibly released from the waiting state (E_RLWAI).
4. del_dtq is issued and the data queue is deleted (E_DLT).

[Differences from μITRON3.0]

tsnd_dtq is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	tsnd_dtq is not included in the system
E_PAR	−17	The parameter is illegal − The timeout time is illegal (tmout < TMO_FEVR = −1)
E_ID	−18	The data queue ID number is outside the range (dtqid ≤ 0, maximum data queue count < dtqid)
E_CTX	−25	tsnd_dtq wad issued from a non-task context tsnd_dtq wad issued in the CPU lock state tsnd_dtq wad issued in the dispatch disabled state
E_NOEXS	−42	The target data queue does not exist
E_RLWAI	−49	The task is forcibly released from the waiting state by (i)rel_wai
E_TMOUT	−50	Timeout
E_DLT	−51	The data queue is deleted while the task is waiting

fsnd_dtq/ifsnd_dtq

Force Send Data to Data Queue

[Overview]

Transmits data to a data queue (overwriting data)

[C format]

```
#include <kernel.h>
ER fsnd_dtq(ID dtqid, VP_INT data);
```

[Parameters]

I/O	Parameter		Description
I	ID	dtqid	ID number of data queue to which data is to be transmitted
I	VP_INT	data	Transmit data

[Explanation]

The data specified by data is transmitted to the data queue specified by dtqid.

If tasks are waiting in the target data queue for data reception, the first task in the queue is immediately allowed to receive the data and is released from the queue. If no waiting task exists, the data is stored in the ring buffer of the data queue on a FIFO basis. If the buffer is already filled with the other data, the oldest data is forcibly overwritten. Overwriting takes place even if a task waiting for data transmission exists.

If fsnd_dtq is issued to a data queue without buffers, an error occurs and the error code E_ILUSE is returned.

ifsnd_dtg is intended to be issued from a non-task context but it can also be issued from a task context. fsnd_dtg can be issued from a non-task context.

[Differences from μITRON3.0]

fsnd_dtq/ifsnd_dtq is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	fsnd_dtq/ifsnd_dtq is not included in the system
E_ID	−18	The data queue ID number is outside the range (dtqid ≤ 0, maximum data queue count < dtqid)
E_CTX	−25	fsnd_dtq/ifsnd_dtq was issued in the CPU lock state
E_ILUSE	−28	fsnd_dtq/ifsnd_dtq was issued to a data queue without buffer
E_NOEXS	−42	The target data queue does not exist

rcv_dtq

Receive Data from Data Queue

[Overview]

Receives data from a data queue

[C format]

```
#include <kernel.h>
ER rcv_dtq(ID dtqid, VP_INT* p_data);
```

[Parameters]

I/O	Parameter		Description
I	ID	dtqid	ID number of data queue from which data is to be received
O	VP_INT*	p_data	Address to which received data is to be stored

[Explanation]

Data is received from the data queue specified by dtqid and written to the address specified by p_data. If the buffer of the data queue is already filled with data and if a task waiting for data transmission exists, the transmission processing of the waiting task is completed and the task is released from the waiting state.

If no data that has been received exists in the buffer of the target data queue, the task waits for data reception and is registered to the reception waiting task queue of the data queue. At this time, the task waits only on an FIFO basis.

Tasks placed in the waiting state due to rcv_dtq are released from waiting by one of the following occurrences.

1. (t)snd_dtq, (i)psnd_dtq, or (I)fsnd_dtq is issued and data is received (E_OK).
2. (i)rel_wai is issued and the task is forcibly released from the waiting state (E_RLWAI).
3. del_dtq is issued and the data queue is deleted (E_DLT).

[Differences from μITRON3.0]

rcv_dtq is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	rcv_dtq is not included in the system
E_ID	−18	The data queue ID number is outside the range (dtqid ≤ 0, maximum data queue count < dtqid)
E_CTX	−25	rcv_dtq was issued from a non-task context rcv_dtq was issued in the CPU lock state rcv_dtq was issued in the dispatch disabled state
E_NOEXS	−42	The target data queue does not exist
E_RLWAI	−49	The task is forcibly released from the waiting state by (i)rel_wai
E_DLT	−51	The target data queue does not exist

prcv_dtq/iprcv_dtq

Poll and Receive Data from Data Queue

[Overview]

Receives data from a data queue (polling)

[C format]

```
#include <kernel.h>
ER prcv_dtq(ID dtqid, VP_INT* p_data);
```

[Parameters]

I/O	Parameter		Description
I	ID	dtqid	ID number of data queue from which data is to be received
O	VP_INT*	p_data	Address to which received data is to be stored

[Explanation]

Data is received from the data queue specified by dtqid, and written to the address specified by p_data. If the buffer of the data queue is already filled with data and if a task waiting for data transmission exists, the transmission processing of the waiting task is completed and the task is released from the waiting state.

If no data that has been received exists in the buffer of the target data queue, an error occurs and the error code E_TMOUT is returned.

iprcv_dtg is intended to be issued from a non-task context but it can also be issued from a task context. prcv_dtg can be issued from a non-task context.

[Differences from μITRON3.0]

prcv_dtq/iprcv_dtq is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	prcv_dtq/iprcv_dtq is not included in the system
E_ID	−18	The data queue ID number is outside the range (dtqid ≤ 0, maximum data queue count < dtqid)
E_CTX	−25	prcv_dtq/iprcv_dtq was issued in the CPU lock state
E_NOEXS	−42	The target data queue does not exist
E_TMOUT	−50	Polling has failed

trcv_dtq

Receive Data from Data Queue with Timeout

[Overview]

Receives data from a data queue (with timeout)

[C format]

```
#include <kernel.h>
ER trcv_dtq(ID dtqid, VP_INT* p_data, TMO tmout);
```

[Parameters]

I/O	Parameter		Description
I	ID	dtqid	ID number of data queue from which data is to be received
O	VP_INT*	p_data	Address to which received data is to be stored
I	TMO	tmout	Timeout time [milliseconds]

[Explanation]

Data is received from the data queue specified by dtqid, and written to the address specified by p_data. If the buffer of the data queue is already filled with data and if a task waiting for data transmission exists, the transmission processing of the waiting task is completed and the task is released from the waiting state.

If no data that has been received exists in the buffer of the target data queue, the task waits for data reception only for the time specified as tmout and is registered to the reception waiting task queue of the data queue. At this time, the task waits only on an FIFO basis.

If TMO_POL = 0 is specified as tmout, 0 is specified as the timeout time and trcv_dtq performs the same operation as prcv_dtq. If TMO_FEVR = −1 is specified as tmout, the timeout time is specified to be infinite and this service call performs the same operation as rcv_dtq.

Tasks placed in the waiting state due to trcv_dtq are released from waiting by one of the following occurrences.

1. (t)snd_dtq, (i)psnd_dtq, or (i)fsnd_dtq is issued and data is received (E_OK).
2. The specified time has elapsed and therefore timeout has occurred (E_TMOUT).
3. (i)rel_wai is issued and the task is forcibly released from the waiting state (E_RLWAI).
4. del_dtq is issued and the data queue is deleted (E_DLT).

[Differences from μITRON3.0]

trcv_dtq is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	trcv_dtq is not included in the system
E_PAR	−17	The parameter is illegal − The timeout time is illegal (tmout < TMO_FEVR = −1)
E_ID	−18	The data queue ID number is outside the range (dtqid ≤ 0, maximum data queue count < dtqid)
E_CTX	−25	trcv_dtq was issued from a non-task context trcv_dtq was issued in the CPU lock state trcv_dtq was issued in the dispatch disabled state
E_NOEXS	−42	The target data queue does not exist
E_RLWAI	−49	The task is forcibly released from the waiting state by (i)rel_wai
E_TMOUT	−50	Timeout
E_DLT	−51	The data queue is deleted while the task is waiting

ref_dtq/iref_dtq

Refer Data Queue Status

[Overview]

Obtains data queue information

[C format]

```
#include <kernel.h>
ER ref_dtq(ID dtqid, T_RDTQ *pk_rdtq);
```

[Parameters]

I/O	Parameter		Description
I	ID	dtqid	ID number of data queue whose information is to be obtained
O	T_RDTQ *	pk_rdtq	Address of data queue information packet

Configuration of T_RDTQ

```
typedef struct t_rdtq {
        ID          stskid;     /* ID number of task waiting for transmission */
        ID          rtskid;     /* ID number of task waiting for reception */
        UINT        sdtqcnt;    /* Number of data not received */
        ATR         dtqatr;     /* Data queue attribute */
        UINT        dtqcnt;     /* Size of ring buffer (number of ring buffers)*/
        VP          dtq;        /* Reserved area */
} T_RDTQ;
```

[Explanation]

Information is obtained on the data queue specified by dtqid, and stored in the packet specified by pk_rdtq. The data queue information is described in detail below.

- stskid
  Stores the ID number of the first task in the waiting task queue if tasks waiting for data transmission exist in the target data queue. If no task exists, TSK_NONE = 0 is stored.
- rtskid
  Stores the ID number of the first task in the waiting task queue if tasks waiting for data reception exist in the target data queue. If no task exists, TSK_NONE = 0 is stored.
- sdtqcnt
  Indicates the number of data stored in the ring buffer of the specified queue, waiting to be received by a task.
- dtqatr
  dtqatr stores the data queue attribute of the target data queue. For the meaning of the value stored, refer to the description of cre_dtq.

- dtqcnt
  Indicates the size of the ring buffer used by the target data queue as the number of ring buffers.
- dtq
  This is a reserved area and always stores NULL.

iref_dtg is intended to be issued from a non-task context but it can also be issued from a task context. ref_dtg can be issued from a non-task context.

[Differences from μITRON3.0]

ref_dtq/iref_dtq is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ref_dtq/iref_dtq is not included in the system
E_ID	−18	The data queue ID number is outside the range (dtqid ≤ 0, maximum data queue count < dtqid)
E_CTX	−25	ref_dtq/iref_dtq was issued in the CPU lock state
E_NOEXS	−42	The target data queue does not exist

cre_mbx

Create Mailbox

[Overview]

Creates a mailbox

[C format]

```
#include <kernel.h>
ER cre_mbx(ID mbxid, T_CMBX *pk_cmbx);
```

[Parameters]

I/O	Parameter		Description
I	ID	mbxid	ID number of mailbox to be created
I	T_CMBX *	pk_cmbx	Address of mailbox creation packet

Configuration of T_CMBX

```
typedef struct t_cmbx {
        ATR         mbxatr;     /* Mailbox attribute */
        PRI         maxmpri;    /* Mail priority range */
        VP          mprihd;     /* Reserved area */
} T_CMBX;
```

[Explanation]

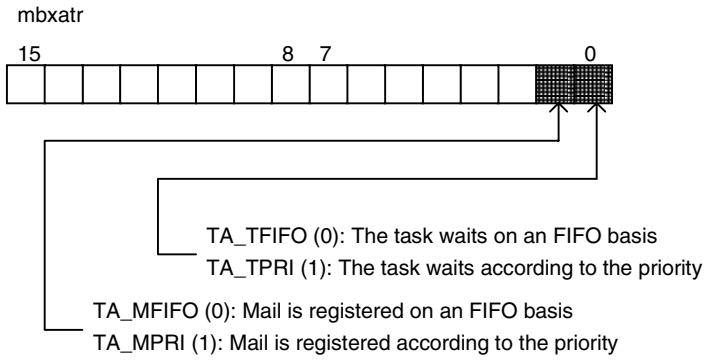
A mailbox with the ID number specified by mbxid is created based on the information stored in the packet pk_cmbx. In other words, a control block is assigned to the created mailbox and the control block is initialized.

The mailbox creation packet T_CMBX is described in detail below.

- mbxatr

  Specifies how tasks wait in the mailbox created as a mailbox attribute. The value that can be specified as the mailbox attribute is as follows:

Figure 13-7. Mailbox Attribute



Bit 0 specifies how the task waiting for mail reception is registered to the waiting task queue if no mail is registered to the mailbox. If TA_TFIFO is specified, the task waits on an FIFO basis. If TA_TPRI is specified, it waits according to the priority.

Bit 1 specifies how mail is registered in the mailbox when it is transmitted, if there is no task waiting for mail reception. If TA_MFIFO is specified, the mail is registered on an FIFO basis. If TA_MPRI is specified, the priority of the mail is referenced and the mail is registered according to its priority.

- maxmpri

  Specifies the range (maximum value) of the priority of the mail if TA_MPRI is specified as the mailbox attribute. An integer of 1 or greater and 0x7fff or less can be specified. The lower the value, the higher the priority of the mail. This value is ignored if TA_MFIFO is specified.
- mprihd

  This is a reserved area. Always specify NULL. Anything other than NULL is ignored even if specified.

[Differences from μITRON3.0]

The extended data exinf has been deleted from the members of the mailbox creation packet T_CMBX, and maxmpri and mprihd have been added.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	cre_mbx is not included in the system
E_RSATR	−11	The mailbox attribute is illegal
E_PAR	−17	The parameter is illegal − The priority of the mail is outside the range (maxmpri ≤ 0, 0x7fff < maxmpri)
E_ID	−18	The mailbox ID number is outside the range (mbxid ≤ 0, maximum mailbox count < mbxid)
E_CTX	−25	cre_mbx was issued from a non-task context cre_mbx was issued in the CPU lock state
E_OBJ	−41	A mailbox with the same ID number already exists

acre_mbx

Create Mailbox with Automatic ID Assignment

[Overview]

Creates a mailbox (Automatic assignment of ID number)

[C format]

```
#include <kernel.h>
ER_ID acre_mbx(T_CMBX *pk_cmbx);
```

[Parameter]

I/O	Parameter		Description
I	T_CMBX *	pk_cmbx	Address of mailbox creation packet

[Explanation]

A mailbox is created based on the mailbox creation information stored in pk_cmbx, and the ID number of the mailbox is returned. In other words, a mailbox control block that is available but has not been created is searched, assigned, and initialized, and the ID number of the control block is returned. If the return value is negative, an error occurs. For the meaning of the return value, refer to **[Return values]** below.

For the details of the mailbox creation packet, refer to the description of cre_mbx.

[Differences from μITRON3.0]

acre_mbx is a newly created service call equivalent to cre_mbx, but with the addition of an automatic ID number assignment function.

[Return values]

Symbol	Value	Meaning
(Positive integer)		ID number of created mailbox (normal termination)
E_RSFN	−10	acre_mbx is not included in the system
E_RSATR	−11	The mailbox attribute is illegal
E_PAR	−17	The parameter is illegal – The mail priority is outside the range (maxmpri ≤ 0, 0x7fff < maxmpri)
E_CTX	−25	acre_mbx was issued from a non-task context acre_mbx was issued in the CPU lock state
E_NOID	−34	The ID number cannot be assigned (reserving a control block has failed)

del_mbx

Delete Mailbox

[Overview]

Deletes a mailbox

[C format]

```
#include <kernel.h>
ER del_mbx(ID mbxid);
```

[Parameter]

I/O	Parameter		Description
I	ID	mbxid	ID number of mailbox to be deleted

[Explanation]

The control block of the mailbox specified by mbxid is invalidated and the mailbox is deleted. After deletion, a new mailbox can be created using the same ID number.

If waiting tasks exist in the target mailbox, all the waiting tasks are released from the waiting state. Value E_DLT that indicates that the mailbox has been deleted is returned for the error code of service call (t)rcv_mbx that caused the tasks to wait.

Even if a message waiting to be received is registered to the target mailbox, the mailbox is deleted. Note that, even if a memory block is registered as a message, the memory block is not returned to the memory pool.

[Differences from μITRON3.0]

The RX4000 V3.x automatically releases a memory block registered as a message to a memory pool when del_mbx is issued, but the RX4000 V4.0 does not release the memory block.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	del_mbx is not included in the system
E_ID	−18	The mailbox ID number is outside the range (mbxid ≤ 0, maximum mailbox count < mbxid)
E_CTX	−25	del_mbx was issued from a non-task context del_mbx was issued in the CPU lock state
E_NOEXS	−42	The target mailbox does not exist.

Send Mail to Mailbox

snd_mbx/isnd_mbx

[Overview]

Transmits a message to a mailbox

[C format]

```
#include <kernel.h>
ER snd_mbx(ID mbxid, T_MSG* pk_msg);
```

[Parameters]

I/O	Parameter		Description
I	ID	mbxid	ID number of mailbox to which message is to be transmitted
I	T_MSG *	pk_msg	Address of message packet

[Explanation]

The message specified by pk_msg is transmitted to the mailbox specified by mbxid.

If waiting tasks exist in the task queue of the target mailbox, the first task in the waiting task queue is released from the waiting state and is immediately allowed to receive the message. If no waiting task exists, the transmitted message is registered to the message queue of the mailbox. At this time, the message is registered in the order specified (on an FIFO basis or according to priority) by an attribute when the message was created. The registered message is received in the order in which it was registered. To register a priority for a message, use the structure T_MSG_PRI instead of the structure T_MSG. For details, refer to the **RX4000 (μITRON4.0) Technical User's Manual (U14835E)**.

An area where the kernel manages messages (message header) is necessary for a message packet (pk_msg). The first 4 bytes of this area (6 bytes if the message has a priority) are used. Therefore, the user must write the body of the message after the message header. For details, refer to **5.5 Mailbox**.

A message is transmitted or received not by copying the message itself but by transmitting or receiving its address. Therefore, it is necessary to prevent the memory area used for the message from being rewritten even after the message has been transmitted.

isnd_mbx is intended to be issued from a non-task context but it can also be issued from a task context. snd_mbx can be issued from a non-task context.

[Differences from μITRON3.0]

The name of snd_msg has been changed to snd_mbx. With the RX4000 V3.x, an error occurred if snd_msg was issued with a message already registered to the mailbox specified. With the RX4000 V4.0, however, no error occurs. Multiple transmission must be checked by the user.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	snd_mbx/isnd_mbx is not included in the system
E_PAR	−17	The parameter is illegal – The priority of the message is outside the range (msgpri ≤ 0, maximum priority value < msgpri)
E_ID	−18	The mailbox ID number is outside the range (mbxid ≤ 0, maximum mailbox count < mbxid)
E_CTX	−25	snd_mbx/isnd_mbx was issued in the CPU lock state
E_NOEXS	−42	The target mailbox does not exist

rcv_mbx

Receive Mail from Mailbox

[Overview]

Receives a message from a mailbox

[C format]

```
#include <kernel.h>
ER rcv_mbx(ID mbxid, T_MSG** ppk_msg);
```

[Parameters]

I/O	Parameter		Description
I	ID	mbxid	ID number of mailbox from which message is to be received
O	T_MSG **	ppk_msg	Address to store address of receive message packet

[Explanation]

A message is received from the mailbox specified by mbxid, and the first address of the message is stored in the area specified by ppk_msg.

If no message is registered to the target mailbox and therefore cannot be received, the task waits for message reception and is registered to the waiting task queue of the mailbox in the sequence (on an FIFO basis or according to priority) specified when the mailbox was created.

Tasks placed in the waiting state due to rcv_mbx are released from waiting by one of the following occurrences.

1. (i)snd_mbx is issued and the task receives a message (E_OK).
2. (i)rel_wai is issued and the task is forcibly released from the waiting state (E_RLWAI).
3. del_mbx is issued and the mailbox is deleted (E_DLT).

[Differences from μITRON3.0]

The name of rcv_msg has been changed to rcv_mbx. In addition, the order of the parameters has also been changed.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	rcv_mbx is not included in the system
E_ID	−18	The mailbox ID number is outside the range (mbxid ≤ 0, maximum mailbox count < mbxid)
E_CTX	−25	rcv_mbx was issued from a non-task context rcv_mbx was issued in the CPU lock state rcv_mbx was issued in the dispatch disabled state
E_NOEXS	−42	The target mailbox does not exist
E_RLWAI	−49	The task is forcibly released from the waiting state by (i)rel_wai
E_DLT	−51	The mailbox is deleted while a task is waiting

prcv_mbx/iprcv_mbx

Poll and Receive Mail from Mailbox

[Overview]

Receives a message from a mailbox (polling)

[C format]

```
#include <kernel.h>
ER prcv_mbx(ID mbxid, T_MSG** ppk_msg);
```

[Parameters]

I/O	Parameter		Description
I	ID	mbxid	ID number of mailbox from which message is to be received
O	T_MSG **	ppk_msg	Address to store address of receive message packet

[Explanation]

A message is received from the mailbox specified by mbxid, and the address of the message is stored in the area specified by ppk_msg.

If a message is registered to the target mailbox, the task receives the message and continues processing. If no message is registered, polling fails and the error code E_TMOUT is returned.

iprcv_mbx is intended to be issued from a non-task context but it can also be issued from a task context. prcv_mbx can be issued from a non-task context.

[Differences from μITRON3.0]

The name of prcv_msg has been changed to prcv_mbx. In addition, the order of the parameters has also been changed.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	prcv_mbx/iprcv_mbx is not included in the system
E_ID	−18	The mailbox ID number is outside the range (mbxid ≤ 0, maximum mailbox count < mbxid)
E_CTX	−25	prcv_mbx/iprcv_mbx was issued in the CPU lock state
E_NOEXS	−42	The target mailbox does not exist
E_TMOUT	−50	Polling has failed

trcv_mbx

Receive Mail from Mailbox with Timeout

[Overview]

Receives a message from a mailbox (with timeout)

[C format]

```
#include <kernel.h>
ER trcv_mbx(ID mbxid, T_MSG** ppk_msg, TMO tmout);
```

[Parameters]

I/O	Parameter		Description
I	ID	mbxid	ID number of mailbox from which message is to be received
O	T_MSG **	ppk_msg	Address to store address of receive message packet
I	TMO	tmout	Timeout time [milliseconds]

[Explanation]

A message is received from the mailbox specified by mbxid, and the address of the message is stored in the area specified by ppk_msg.

If a message is registered to the target mailbox, the task receives the message and continues processing. If no message is registered to the mailbox, the task waits only for the time specified by tmout. The task is registered to the waiting task queue of the mailbox in the order (on an FIFO basis or according to priority) specified by the attribute when the mailbox was created.

If TMO_POL = 0 is specified as tmout, 0 is specified as the timeout time and trcv_mbx performs the same operation as prcv_mbx. If TMO_FEVR = −1 is specified as tmout, the timeout time is specified to be infinite and this service call performs the same operation as rcv_mbx.

Tasks placed in the waiting state due to trcv_mbx are released from waiting by one of the following occurrences.

1. (i)snd_mbx is issued and the task receives a message (E_OK).
2. The specified time elapses and timeout occurs (E_TMOUT).
3. (i)rel_wai is issued and the task is forcibly released from the waiting state (E_RLWAI).
4. del_mbx is issued and the mailbox is deleted (E_DLT).

[Differences from μITRON3.0]

The name of trcv_msg has been changed to trcv_mbx. In addition, the order of the parameters has also been changed.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	trcv_mbx is not included in the system
E_PAR	−17	The timeout time is illegal (tmout < TMO_FEVR = −1)
E_ID	−18	The mailbox ID number is outside the range (mbxid ≤ 0, maximum mailbox count < mbxid)
E_CTX	−25	trcv_mbx was issued from a non-task context trcv_mbx was issued in the CPU lock state trcv_mbx was issued in the dispatch disabled state
E_NOEXS	−42	The target mailbox does not exist
E_RLWAI	−49	The task is forcibly released from the waiting state by (i)rel_wai
E_TMOUT	−50	Timeout
E_DLT	−51	The mailbox is deleted while a task is waiting

Refer Mailbox Status

ref_mbx/iref_mbx

[Overview]

Obtains mailbox information

[C format]

```
#include <kernel.h>
ER ref_mbx(ID mbxid, T_RMBX *pk_rmbx);
```

[Parameters]

I/O	Parameter		Description
I	ID	mbxid	ID number of mailbox whose information is to be obtained
O	T_RMBX *	pk_rmbx	Address of mailbox information packet

Configuration of T_RMBX

```
typedef struct t_rmbx {
        ID          wtskid;     /* Presence/absence of waiting task */
        T_MSG *     pk_msg;     /* Presence/absence of message */
        ATR         mbxatr;     /* Mailbox attribute */
} T_RMBX;
```

[Explanation]

Information is obtained on the mailbox specified by mbxid and stored in the packet specified by pk_mbx. The mailbox information is described in detail below. To issue this service call from a non-task context such as an interrupt service routine, use iref_mbx.

- wtskid
  Stores the ID number of the first task in the waiting task queue if tasks waiting for message reception exist in the target mailbox. If no task exists, TSK_NONE = 0 is stored.
- pk_msg
  Stores the address of the first message in the queue of the messages that are registered to the target mailbox and have not been received. If no message is registered, NULL is stored.
- mbxatr
  Stores the mailbox attribute of the target mailbox. For the meaning of the stored value, refer to the description of cre_mbx.

iref_mbx is intended to be issued from a non-task context but it can also be issued from a task context. ref_mbx can be issued from a non-task context.

[Differences from μITRON3.0]

The extended data exinf has been deleted from the mailbox information and mailbox attribute mbxatr has been added. The type of wtskid indicating the presence or absence of a waiting task has been changed from BOOL_ID to ID. In addition, the order of the parameters has also been changed.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ref_mbx/iref_mbx is not included in the system
E_ID	−18	The mailbox ID number is outside the range (mbxid ≤ 0, maximum mailbox count < mbxid)
E_CTX	−25	ref_mbx/iref_mbx was issued in the CPU lock state
E_NOEXS	−42	The target mailbox does not exist

cre_mtx

Create Mutex

[Overview]

Creates a mutex

[C format]

```
#include <itron.h>
ER cre_mtx(ID mtxid, T_CMTX *pk_cmtx);
```

[Parameters]

I/O	Parameter		Description
I	ID	mtxiid	ID number of mutex to be created
I	T_CMTX *	pk_cmtx	Address of mutex creation packet

Configuration of T_CMTX

```
typedef struct t_cmtx {
        ATR             mtxatr;         /* Mutex attribute */
        PRI             ceilpri;        /* Priority ceiling value */
} T_CMTX;
```

[Explanation]

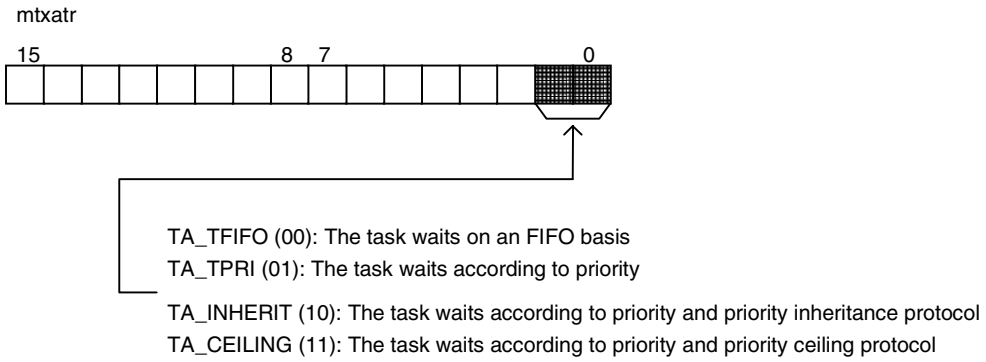
A mutex with the ID number specified by mtxid is created based on the information stored in the packet pk_cmtx. In other words, a control block is assigned to the mutex to be created and the control block is initialized.

The mutex creation packet T_CMTX is described in detail below.

- mtxatr

  Specifies the priority control protocol of the mutex to be created and the order in which tasks wait as a mutex attribute. The values that can be specified as a mutex attribute and their meanings are as follows:

Figure 13-8. Mutex Attribute



Bits 0 and 1 of mtxatr specify the order in which tasks are registered to the waiting task queue, and a priority control protocol if tasks wait for a mutex.

The tasks wait on an FIFO basis if TA_TFIFO is specified; if any other attribute is specified, they wait according to priority. No priority control is performed if the TA_TFIFO and TA_TPRI attributes are specified. If the TA_INHERIT attribute is specified, a priority inheritance protocol is employed. If TA_CEILING attribute is specified, a priority ceiling protocol is employed.

- ceilpri

  Specifies the ceiling value of the priority if TA_CEILING is specified as the mutex attribute. If a task waits for a mutex with the priority ceiling protocol selected, and if the priority of the task that locks the mutex is lower than that specified by ceilpri, the priority of the task is changed to that specified by ceilpri.

  If an attribute other than TA_CEILING is specified as the mutex attribute, the value of ceilpri is meaningless.

[Differences from μITRON3.0]

cre_mtx is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	cre_mtx is not included in the system
E_RSATR	−11	The mutex attribute is illegal
E_PAR	−17	The parameter is illegal – The priority ceiling value is outside the range (ceilpri ≤ 0, maximum priority value < cilpri)
E_ID	−18	The mutex ID number is outside the range (mtxid ≤ 0, maximum mutex count < mtxid)
E_CTX	−25	cre_mtx was issued from a non-task context cre_mtx was issued in the CPU lock state
E_OBJ	−41	A mutex with the same ID number already exists

acre_mtx

Create Mutex with Automatic ID assignment

[Overview]

Creates a mutex (Automatic assignment of ID number)

[C format]

```
#include <kernel.h>
ER_ID acre_mtx(T_CMTX *pk_cmtx);
```

[Parameter]

I/O	Parameter		Description
I	T_CMTX *	mtxid	Address of mutex creation packet

[Explanation]

A mutex is created based on the information stored in the packet pk_cmtx, and the ID number of the mutex is returned. In other words, a mutex control block that is available but has not been created is searched, assigned, and initialized, and its ID number is returned. If the return value is negative, it is an error. For the meaning of the return value, refer to **[Return values]** below.

For details of the mutex creation packet, refer to the description of cre_mtx.

[Differences from μITRON3.0]

acre_mtx is a newly created service call equivalent to cre_mtx, but with the addition of an automatic ID number assignment function.

[Return values]

Symbol	Value	Meaning
(Integer of 1 or greater)		ID number of the created mutex (normal termination)
E_RSFN	−10	acre_mtx is not included in the system
E_RSATR	−11	The mutex attribute is illegal
E_PAR	−17	The parameter is illegal − The priority ceiling value is outside the range (ceilpri ≤ 0, maximum priority value < ceilpri)
E_CTX	−25	acre_mtx was issued from a non-task context acre_mtx was issued in the CPU lock state
E_NOID	−34	The ID number cannot be assigned

del_mtx

Delete Mutex

[Overview]

Deletes a mutex

[C format]

```
#include <itron.h>
ER del_mtx(ID mtxid);
```

[Parameter]

I/O	Parameter		Description
I	ID	mtxid	ID number of mutex to be deleted

[Explanation]

The control block of the mutex specified by mtxid is invalidated and the mutex is deleted. After deletion, a new mutex can be created using the same ID number.

If waiting tasks exist for the target mutex, all the tasks are released from the waiting state. The value E_DLT indicating that the mutex has been deleted is returned for the error code of the service call (t)loc_mtx that caused the tasks to wait.

Even if the target mutex is locked by a task, it is deleted. However, deletion of the mutex is not immediately reported to this task. The task learns that the mutex has been deleted if E_NOEXS or E_ILUSE is returned as the error code of unl_mtx when lock of the mutex is released.

If the task locks a mutex with the TA_INHERIT or TA_CEILING attribute with the priority raised by either of the priority control protocols, and if the mutex is deleted by del_mtx (any other mutex is not locked), the priority of the task drops to the base priority.

[Differences from μITRON3.0]

del_mtx is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	del_mtx is not included in the system
E_ID	−18	The mutex ID number is outside the range (mtxid ≤ 0, maximum mutex count < mtxid)
E_CTX	−25	del_mtx was issued from a non-task context del_mtx was issued in the CPU lock state
E_NOEXS	−42	The target mutex does not exist

loc_mtx

Lock Mutex

[Overview]

Locks a mutex

[C format]

```
#include <kernel.h>
ER loc_mtx(ID mtxid);
```

[Parameter]

I/O	Parameter		Description
I	ID	mtxiid	ID number of mutex to be locked

[Explanation]

The mutex specified by mtxid is locked.

If the target mutex can be locked, i.e., if locking the mutex is successful, the task continues processing. If the mutex has the TA_CEILING attribute at this time and if the ceiling of the priority (ceilpri) specified when the mutex was created is higher than the current priority of the task that locks the mutex, the priority of the task is raised to the ceiling value.

If the mutex has been already locked by another task, the task enters the waiting state and is registered to the waiting task queue in the order (on an FIFO basis or according to priority) specified when the mutex was created. If the mutex has the TA_INHERIT attribute at this time and if the current priority of the task that has entered the waiting state is higher than the current priority of the task that is locking the mutex, the priority of the locking task is lowered to the current priority of the waiting task.

If the target mutex has the TA_CEILING attribute and if the base priority of the task is higher than the ceiling value of the priority of the mutex, an error occurs and the error code E_ILUSE is returned.

The current priority raised by the mutex is changed to the base priority of the task when the task releases the lock of the mutex (if the task locks two or more mutexes, when it releases the lock of all the mutexes).

Tasks placed in the waiting state due to loc_mtx are released from waiting by one of the following occurrences.

1. unl_mtx is issued and the task locks a new mutex (E_OK).
2. The locking task is terminated and a new mutex is locked (E_OK).
3. (i)rel_wai is issued and the task is forcibly released from the waiting state (E_RLWAI).
4. del_mtx is issued and the mutex is deleted (E_DLT).

[Differences from μITRON3.0]

loc_mtx is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	loc_mtx is not included in the system
E_ID	−18	The mutex ID number is outside the range (mtxid ≤ 0, maximum mutex count < mtxid)
E_CTX	−25	loc_mtx was issued from a non-task context loc_mtx was issued in the CPU lock state loc_mtx was issued in the dispatch disabled state
E_ILUSE	−28	The base priority is higher than the priority ceiling value (when TA_CEILING attribute is specified)
E_NOEXS	−42	The target mutex does not exist
E_RLWAI	−49	The task is forcibly released from the waiting state by (i)rel_wai
E_DLT	−51	The mutex is deleted while a task is waiting

ploc_mtx

Poll and Lock Mutex

[Overview]

Locks a mutex (polling)

[C format]

```
#include <kernel.h>
ER ploc_mtx(ID mtxid);
```

[Parameter]

I/O	Parameter		Description
I	ID	mtxiid	ID number of mutex to be locked

[Explanation]

The mutex specified by mtxid is locked.

If the target mutex can be locked, i.e., if locking the mutex is successful, the task continues processing. If the mutex has the TA_CEILING attribute at this time and if the current priority of the locking task is lower than the priority ceiling value (ceilpri) specified when the mutex was created, the current priority of the task is raised to the ceiling value.

If the mutex has been already locked by another task and cannot be locked, the error code E_TMOUT is returned.

If the target mutex has the TA_CEILING attribute and if the base priority of the locking task is higher than the priority ceiling value of the mutex, an error occurs and the error code E_ILUSE is returned.

The current priority raised by the mutex is changed to the base priority of the task when the task releases the lock of the mutex (if the task locks two or more mutexes, when it releases the lock of all the mutexes).

[Differences from μITRON3.0]

ploc_mtx is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ploc_mtx is not included in the system
E_ID	−18	The mutex ID number is outside the range (mtxid ≤ 0, maximum mutex count < mtxid)
E_CTX	−25	ploc_mtx was issued from a non-task context ploc_mtx was issued in the CPU lock state
E_ILUSE	−28	The base priority is higher than the priority ceiling value (when TA_CEILING attribute is specified)
E_NOEXS	−42	The target mutex does not exist
E_TMOUT	−50	Polling has failed

tloc_mtx

Lock Mutex with Timeout

[Overview]

Locks a mutex (with timeout)

[C format]

```
#include <kernel.h>
ER tloc_mtx(ID mtxid, TMO tmout);
```

[Parameters]

I/O	Parameter		Description
I	ID	mtxid	ID number of mutex to be locked
I	TMO	tmout	Timeout time [milliseconds]

[Explanation]

The mutex specified by mtxid is locked.

If the target mutex can be locked, i.e., if locking the mutex is successful, the task continues processing. If the mutex has the TA_CEILING attribute at this time and if the ceiling of the priority (ceilpri) specified when the mutex was created is higher than the current priority of the task that locks the mutex, the current priority of the task is raised to the ceiling value.

If the mutex has been already locked by another task, the task enters the waiting state for the time specified as tmout and is registered to the waiting task queue in the order (on an FIFO basis or according to priority) specified when the mutex was created. If the mutex has the TA_INHERIT attribute at this time and if the current priority of the task that has entered the waiting state is higher than the current priority of the task that is locking the mutex, the priority of the locking task is raised to the same current priority as the waiting task.

If the target mutex has the TA_CEILING attribute and if the base priority of the task is higher than the ceiling value of the priority of the mutex, an error occurs and the error code E_ILUSE is returned.

The current priority raised by the mutex is changed to the base priority of the task when the task releases the lock of the mutex (if the task locks two or more mutexes, when it releases the lock of all the mutexes).

Tasks placed in the waiting state due to tloc_mtx are released from waiting by one of the following occurrences.

1. unl_mtx is issued and the task locks a new mutex (E_OK).
2. The locking task is terminated and a new mutex is locked (E_OK).
3. The specified time elapses and timeout occurs (E_TMOUT).
4. (i)rel_wai is issued and the task is forcibly released from the waiting state (E_RLWAI).
5. del_mtx is issued and the mutex is deleted (E_DLT).

[Differences from μITRON3.0]

tloc_mtx is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	tloc_mtx is not included in the system
E_PAR	−17	The parameter is illegal − The timeout time is illegal (tmout < TMO_FEVR = −1)
E_ID	−18	The mutex ID number is outside the range (mtxid ≤ 0, maximum mutex count < mtxid)
E_CTX	−25	tloc_mtx was issued from a non-task context tloc_mtx was issued in the CPU lock state tloc_mtx was issued in the dispatch disabled state
E_ILUSE	−28	The base priority is higher than the priority ceiling value (when TA_CEILING attribute is specified)
E_NOEXS	−42	The target mutex does not exist
E_RLWAI	−49	The task is forcibly released from the waiting state by (i)rel_wai
E_TMOUT	−50	Timeout
E_DLT	−51	The mutex is deleted while a task is waiting

unl_mtx

Unlock Mutex

[Overview]

Releases lock of a mutex

[C format]

```
#include <kernel.h>
ER unl_mtx(ID mtxid);
```

[Parameter]

I/O	Parameter		Description
I	ID	mtxid	ID number of mutex whose lock is to be released

[Explanation]

The lock of the mutex specified by mtxid is released.

If waiting tasks exist in the target mutex, the first task in the waiting task queue is released from the waiting state and immediately locks the mutex.

If the current priority of the task that released the lock has been changed based on the priority control protocol specified by the mutex attribute, the current priority of the task is changed to the base priority. If the task locks two or more mutexes, the priority is changed after the task has released the lock of all the mutexes.

The lock of the mutex must be released by the task that locked it. If a task other than the locking task issues unl_mtx, an error occurs and the error code E_ILUSE is returned.

[Differences from μITRON3.0]

unl_mtx is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	unl_mtx is not included in the system
E_ID	−18	The mutex ID number is outside the range (mtxid ≤ 0, maximum mutex count < mtxid)
E_CTX	−25	unl_mtx was issued from a non-task context unl_mtx was issued in the CPU lock state
E_ILUSE	−28	The target mutex is not locked
E_NOEXS	−42	The target mutex does not exist

Refer Mutex Status

ref_mtx/iref_mtx

[Overview]

Obtains mutex information

[C format]

```
#include <kernel.h>
ER ref_mtx(ID mtxid, T_RMTX *pk_rmtx);
```

[Parameters]

I/O	Parameter		Description
I	ID	mtxid	ID number of mutex whose information is to be obtained
O	T_RMTX *	pk_rmtx	Address of mutex information packet

Configuration of T_RMTX

```
typedef struct t_rmtx {
        ID          htskid;     /* ID number of locking task */
        ID          wtskid;     /* ID number of task waiting to lock */
        ATR         mtxatr;     /* Mutex attribute */
        PRI         ceilpri;    /* Priority ceiling value */
} T_RMTX;
```

[Explanation]

Information is obtained on the mutex specified by mtxid and stored in the packet specified by pk_rmtx. The mutex information is described in detail below.

- htskid
  Stores the ID number of the task currently locking the target mutex. If the mutex is not locked, TSK_NONE = 0 is stored.
- wtskid
  Stores the ID number of the task waiting to lock the target mutex. If two or more tasks are waiting, the ID number of the first task in the waiting task queue is stored. If no waiting task exists, TSK_NONE = 0 is stored.
- mtxatr
  Stores the attribute of the target mutex. For the meaning of the stored value, refer to the description of cre_mtx.
- ceilpri
  Stores the priority ceiling value of the target mutex. This value is valid only when the mutex has the TA_CEILING attribute; otherwise, it will be meaningless.

iref_mtx is intended to be issued from a non-task context but it can also be issued from a task context. ref_mtx can be issued from a non-task context.

[Differences from μITRON3.0]

ref_mtx is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ref_mtx/iref_mtx is not included in the system
E_ID	−18	The mutex ID number is outside the range (mtxid ≤ 0, maximum mutex count < mtxid).
E_CTX	−25	ref_mtx/iref_mtx was issued in the CPU lock state
E_NOEXS	−42	The target mutex does not exist

13.8.5 Memory pool management function service calls

This section describes the memory pool management function service calls listed in the following table.

Table 13-12. Memory Pool Management Function Service calls

Name	Function
cre_mpf	Creates a fixed-length memory pool
acre_mpf	Creates a fixed-length memory pool (automatic assignment of ID No.)
del_mpf	Deletes a fixed-length memory pool
rel_mpf/irel_mpf	Returns a memory block
get_mpf	Acquires a memory block
pget_mpf/ipget_mpf	Acquires a memory block (polling)
tget_mpf	Acquires a memory block (with timeout)
ref_mpf/iref_mpf	Obtains fixed-length memory pool information
cre_mpl	Creates a variable-length memory pool
acre_mpl	Creates a variable-length memory pool (automatic assignment of ID No.)
del_mpl	Deletes a variable-length memory pool
rel_mpl/irel_mpl	Returns a memory block
get_mpl	Acquires a memory block
pget_mpl/ipget_mpl	Acquires a memory block (polling)
tget_mpl	Acquires a memory block (with timeout)
ref_mpl/iref_mpl	Obtains variable-length memory pool information

cre_mpf

Create Fixed-Size Memory Pool

[Overview]

Creates a fixed-length memory pool

[C format]

```
#include <kernel.h>
ER cre_mpf(ID mpfid, T_CMPF *pk_cmpf);
```

[Parameters]

I/O	Parameter		Description
I	ID	mpfid	ID number of fixed-length memory pool to be created
I	T_CMPF *	pk_cmpf	Address of fixed-length memory pool creation packet

Configuration of T_CMPF

```
typedef struct t_cmpf {
        ATR             mpfatr;         /* Fixed-length memory pool attribute */
        UINT            blkcnt;         /* Number of memory blocks */
        UINT            blksz;          /* Size of memory block */
        VP              mpf;            /* Reserved area */
} T_CMPF;
```

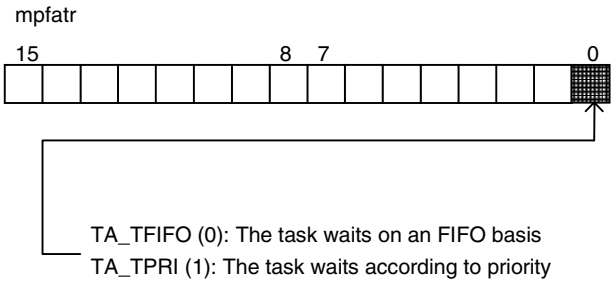
[Explanation]

A fixed-length memory pool with the ID specified by mpfid is created based on the information stored in the fixed-length memory pool creation packet pk_cmpf. In other words, a control block is assigned to the fixed-length memory pool to be created, the control block is initialized, and the area that is the main body of the memory pool is reserved from the user pool.

The fixed-length memory creation packet T_CMPF is described in detail below.

- mpfatr
  Specifies the order in which tasks wait in the waiting task queue of the memory pool to be created as a fixed-length memory pool attribute. The values that can be specified as the fixed-length memory pool attribute are as follows:

Figure 13-9. Fixed-Length Memory Pool Attribute



If TA_TFIFO is specified by mpfatr, the task waits on an FIFO basis. If TA_TPRI is specified, the task waits according to the priority.

- blkcnt
  Specifies the number of memory blocks constituting the fixed-length memory pool to be created.
- blksz
  Specifies the size of one memory block constituting the fixed-length memory pool to be created in bytes. This value must be an integral multiple of 8.
  If any other value is specified, a size that is the lowest integral multiple of 8 greater than the specified size is assumed.
  Therefore, the total of the area that can be used for memory blocks is (blksz × blkcnt). The size of the area that is extracted from the user pool, however, is greater than this value because a header that manages the area of the entire memory pool is appended.
- mpf
  This is a reserved area. Always specify NULL. Anything other than NULL is ignored even if specified.

[Differences from μITRON3.0]

The extended data exinf has been deleted from the members of the fixed-length memory pool creation information and the reserved area mpf has been added.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	cre_mpf is not included in the system
E_RSATR	−11	The fixed-length memory pool attribute is illegal
E_PAR	−17	The parameter is illegal − The number of memory blocks is 0 (blkcnt = 0) − The size of the memory block is 0 (blksz = 0)
E_ID	−18	The fixed-length memory pool ID number is outside the range (mpfid ≤ 0, maximum fixed-length memory pool count < mpfid)
E_CTX	−25	cre_mpf was issued from a non-task context cre_mpf was issued in the CPU lock state
E_NOMEM	−33	The area for the memory pool cannot be reserved
E_OBJ	−41	A fixed-length memory pool having the same ID number already exists

acre_mpf

Create Fixed-Size Memory Pool with Automatic ID Assignment

[Overview]

Creates a fixed-length memory pool (Automatic assignment of ID number)

[C format]

```
#include <kernel.h>
ER_ID acre_mpf(T_CMPF *pk_cmpf);
```

[Parameter]

I/O	Parameter		Description
I	T_CMPF *	pk_cmpf	Address of fixed-length memory pool creation packet

[Explanation]

A fixed-length memory pool is created based on the information stored in the fixed-length memory pool creation packet pk_cmpf, and the ID number of the memory pool is returned. In other words, a fixed-length memory pool control block that is available and has not been created is searched, assigned, and initialized, the area of the main body of the memory pool is reserved from the user pool, and the ID number of the memory pool is returned. If the return value is negative, it is an error. For the meaning of the return value, refer to **[Return values]** below.

For details of the fixed-length memory pool creation packet, refer to the description of cre_mpf.

[Differences from μITRON3.0]

acre_mpf is a newly created service call equivalent to cre_mpf, but with the addition of an automatic ID number assignment function.

[Return values]

Symbol	Value	Meaning
(Integer of 1 or greater)		ID number of the fixed-length memory pool created (normal termination)
E_RSFN	−10	acre_mpf is not included in the system
E_RSATR	−11	The fixed-length memory pool attribute is illegal
E_PAR	−17	The parameter is illegal − The number of memory blocks is 0 (blkcnt = 0) − The size of the memory block is 0 (blksz = 0)
E_CTX	−25	acre_mpf was issued from a non-task context acre_mpf was issued in the CPU lock state
E_NOMEM	−33	The area for the memory pool cannot be reserved
E_NOID	−34	The ID number cannot be assigned (reserving a control block has failed)

del_mpf

Delete Fixed-Size Memory Pool

[Overview]

Deletes a fixed-length memory pool

[C format]

```
#include <kernel.h>
ER del_mpf(ID mpfid);
```

[Parameter]

I/O	Parameter		Description
I	ID	mpfid	ID number of the fixed-length memory pool to be deleted

[Explanation]

The control block of the fixed-length memory pool specified by mpfid is invalidated, the area of the main body of the fixed-length memory pool is returned to the user pool, and the fixed-length memory pool is deleted. After deletion, a new fixed-length memory pool with the same ID number can be created.

The memory pool is deleted regardless of whether memory blocks acquired from the fixed-length memory pool to be deleted have been released or not. Note, therefore, that deletion of the memory pool is not reported to the task that is reserving (using) the memory blocks.

If tasks waiting to get a memory block from the fixed-length memory pool exist, all the tasks are released from the waiting state. The value E_DLT indicating that the fixed-length memory pool has been deleted is returned for the error code of the service call (t)get_mpf that caused the tasks to wait.

[Differences from μITRON3.0]

None

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	del_mpf is not included in the system
E_ID	−18	The fixed-length memory pool ID number is outside the range (mpfid ≤ 0, maximum fixed-length memory pool count < mpfid)
E_CTX	−25	del_mpf was issued from a non-task context del_mpf was issued in the CPU lock state
E_NOEXS	−42	The target fixed-length memory pool does not exist

rel_mpf/irel_mpf

Release Memory Block to Fixed-Size Memory Pool

[Overview]

Returns a memory block

[C format]

```
#include <kernel.h>
ER rel_mpf(ID mpfid, VP blk);
```

[Parameters]

I/O	Parameter		Description
I	ID	mpfid	ID number of fixed-length memory pool to which memory block is to be returned
I	VP	blk	Address of memory block to be returned

[Explanation]

The memory block specified by blk is returned to the fixed-length memory pool specified by mpfid. If a task is waiting for a memory block from the specified memory pool, this task is allowed to acquire the returned memory block and is released from the waiting state.

The memory pool to which the memory block is to be returned must be the same memory pool from which the memory block was acquired. If the memory block is returned to a different memory pool, the operation is not guaranteed. If the memory block to be returned is registered to a mailbox as a message, or if an area other than that of memory blocks is specified, the operation is not guaranteed.

Even if a memory block is returned, the contents of the memory block are not cleared.

irel_mpf is intended to be issued from a non-task context but it can also be issued from a task context. rel_mpf can be issued from a non-task context.

[Differences from μITRON3.0]

The name of rel_blf has been changed to rel_mpf.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	rel_mpf/irel_mpf is not included in the system
E_ID	−18	The fixed-length memory pool ID number is outside the range (mpfid ≤ 0, maximum fixed-length memory pool count < mpfid)
E_CTX	−25	rel_mpf/irel_mpf was issued in the CPU lock state
E_NOEXS	−42	The target fixed-length memory pool does not exist

get_mpf

Get Memory Block from Fixed-Size Memory Pool

[Overview]

Acquires a memory block

[C format]

```
#include <kernel.h>
ER get_mpf(ID mpfid, VP *p_blk);
```

[Parameters]

I/O	Parameter		Description
I	ID	mpfid	ID number of fixed-length memory pool from which memory block is to be acquired
O	VP *	p_blk	Address to store address of memory block

[Explanation]

A memory block is acquired from the fixed-length memory pool specified by mpfid, and the first address of the memory block is stored in the address specified by p_blk.

If no vacant memory block exists in the memory pool when get_mpf is issued, the task waits for the memory block and is registered to the waiting task queue of the memory pool until it can get the memory block. The task is registered to the queue in the order (on an FIFO basis or according to priority) specified when the memory pool was created.

Tasks placed in the waiting state due to get_mpf are released from waiting by one of the following occurrences.

1. (i)rel_mpf is issued and the task acquires a memory block (E_OK).
2. (i)rel_wai is issued and the task is forcibly released from the waiting state (E_RLWAI).
3. del_mpf is issued and the memory pool is deleted (E_DLT).

[Differences from μITRON3.0]

The name of get_blf has been changed to get_mpf.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	get_mpf is not included in the system
E_ID	−18	The fixed-length memory pool ID number is outside the range (mpfid ≤ 0, maximum fixed-length memory pool count < mpfid)
E_CTX	−25	get_mpf was issued from a non-task context get_mpf was issued in the CPU lock state get_mpf was issued in the dispatch disabled state
E_NOEXS	−42	The target fixed-length memory pool does not exist
E_RLWAI	−49	The task is forcibly released from the waiting state by (i)rel_wai
E_DLT	−51	The fixed-length memory pool is deleted while a task is waiting

Poll and Get Memory Block from Fixed-Size Memory Pool

pget_mpf/ipget_mpf

[Overview]

Acquires a memory block (Polling)

[C format]

```
#include <kernel.h>
ER pget_mpf(ID mpfid, VP *p_blk);
```

[Parameters]

I/O	Parameter		Description
I	ID	mpfid	ID number of fixed-length memory pool from which memory block is to be acquired
O	VP *	p_blk	Address to store address of acquired memory block

[Explanation]

A memory block is acquired from the fixed-length memory pool specified by mpfid, and the first address of the memory block is stored in the address specified by p_blk. If no vacant memory block exists in the memory pool when get_mpf is issued, an error occurs and the error code E_TMOUT is returned.

ipget_mpf is intended to be issued from a non-task context but it can also be issued from a task context. pget_mpf can be issued from a non-task context.

[Differences from μITRON3.0]

The name of pget_blf has been changed to pget_mpf.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	pget_mpf/ipget_mpf is not included in the system
E_ID	−18	The fixed-length memory pool ID number is outside the range (mpfid ≤ 0, maximum fixed-length memory pool count < mpfid)
E_CTX	−25	pget_mpf/ipget_mpf was issued in the CPU lock state
E_NOEXS	−42	The target fixed-length memory pool does not exist
E_TMOUT	−50	Polling has failed

tget_mpf

Get Memory Block from Fixed-Size Memory Pool with Timeout

[Overview]

Acquires a memory block (with timeout)

[C format]

```
#include <kernel.h>
ER tget_mpf(ID mpfid, VP *p_blk, TMO tmout);
```

[Parameters]

I/O	Parameter		Description
I	ID	mpfid	ID number of fixed-length memory pool from which memory block is to be acquired
O	VP *	p_blk	Address to store address of acquired memory block
I	TMO	tmout	Timeout time [milliseconds]

[Explanation]

A memory block is acquired from the fixed-length memory pool specified by mpfid, and the first address of the memory block is stored in the address specified by p_blk.

If no vacant memory block exists in the memory pool when tget_mpf is issued, the task waits for the memory block and is registered to the waiting task queue of the memory pool until it can get the memory block or the time specified by tmout elapses. The task is registered to the queue in the order (on an FIFO basis or according to priority) specified when the memory pool was created.

If TMO_POL = 0 is specified as tmout, 0 is specified as the timeout time and tget_mpf performs the same operation as pget_mpf. If TMO_FEVR = −1 is specified as tmout, the timeout time is specified to be infinite and the service call performs the same operation as get_mpf.

Tasks placed in the waiting state due to tget_mpf are released from waiting by one of the following occurrences.

1. (i)rel_mpf is issued and the task acquires a memory block (E_OK).
2. The specified time elapses and timeout occurs (E_TMOUT).
3. (i)rel_wai is issued and the task is forcibly released from the waiting state (E_RLWAI).
4. del_mpf is issued and the memory pool is deleted (E_DLT).

[Differences from μITRON3.0]

The name of tget_blf has been changed to tget_mpf.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	tget_mpf is not included in the system
E_PAR	−17	The parameter is illegal − The timeout time is illegal (tmout < TMO_FEVR = −1)
E_ID	−18	The fixed-length memory pool ID number is outside the range (mpfid ≤ 0, maximum fixed-length memory pool count < mpfid)
E_CTX	−25	tget_mpf was issued from a non-task context tget_mpf was issued in the CPU lock state tget_mpf was issued in the dispatch disabled state
E_NOEXS	−42	The target fixed-length memory pool does not exist
E_RLWAI	−49	The task is forcibly released from the waiting state by (i)rel_wai
E_TMOUT	−50	Timeout
E_DLT	−51	The fixed-length memory pool is deleted while a task is waiting

ref_mpf/iref_mpf

Refer Fixed-Size Memory Pool Status

[Overview]

Obtains fixed-length memory pool information

[C format]

```
#include <kernel.h>
ER ref_mpf(ID mpfid, T_RMPF *pk_rmpf);
```

[Parameters]

I/O	Parameter		Description
I	ID	mpfid	ID number of fixed-length memory pool whose information is to be obtained
O	T_RMPF *	pk_rmpf	Address of fixed-length memory pool information packet

Configuration of T_RMPF

```
typedef struct t_rmpf {
        ID          wtskid;     /* Presence/absence of waiting task */
        UINT        fblkcnt;    /* Number of vacant memory blocks */
        ATR         mpfatr;     /* Fixed-length memory pool attribute */
} T_RMPF;
```

[Explanation]

Information is obtained on the fixed-length memory pool specified by mpfid and stored in the packet specified by pk_rmpf. The fixed-length memory pool information is described in detail below. To issue this service call from a non-task context such as an interrupt service routine, use iref_mpf.

- wtskid
  Indicates the presence or absence of tasks waiting for a memory block from the target fixed-length memory pool. If waiting tasks exist, the ID number of the first task in the waiting task queue is stored. If no waiting task exists, TSK_NONE = 0 is stored.
- fbrkcnt
  Stores the number of vacant memory blocks in the target memory block.
- mpfatr
  Stores the attribute of the target fixed-length memory pool. For the meaning of the stored value, refer to the description of cre_mpf.

iref_mpf is intended to be issued from a non-task context but it can also be issued from a task context. ref_mpf can be issued from a non-task context.

[Differences from μITRON3.0]

The extended data exinf has been deleted from the members of the fixed-length memory pool information and the fixed-length memory pool attribute mpfatr has been added.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ref_mpf/iref_mpf is not included in the system
E_ID	−18	The fixed-length memory pool ID number is outside the range (mpfid ≤ 0, maximum fixed-length memory pool count < mpfid)
E_CTX	−25	ref_mpf/iref_mpf was issued in the CPU lock state
E_NOEXS	−42	The target fixed-length memory pool does not exist

cre_mpl

Create Variable-Size Memory Pool

[Overview]

Creates a variable-length memory pool

[C format]

```
#include <kernel.h>
ER cre_mpl(ID mplid, T_CMPL *pk_cmpl);
```

[Parameters]

I/O	Parameter		Description
I	ID	mplid	ID number of variable-length memory pool to be created
I	T_CMPL *	pk_cmpl	Address of variable-length memory pool creation packet

Configuration of T_CMPL

```
typedef struct t_cmpl {
        ATR             mplatr;         /* Variable-length memory pool attribute */
        SIZE            mplsz;          /* Variable-length memory pool size */
        VP              mpl;            /* Reserved area */
} T_CMPL;
```

[Explanation]

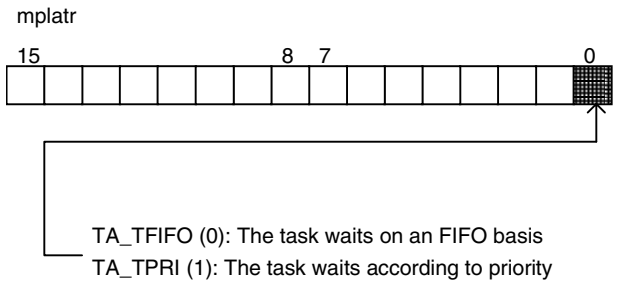
A variable-length memory pool with the ID specified by mplid is created based on the information stored in the variable-length memory pool creation packet pk_cmpl. In other words, a control block is assigned to the variable-length memory pool to be created, the control block is initialized, and an area that is the main body of the memory pool is reserved from the user pool.

The variable-length memory creation packet T_CMPL is described in detail below.

- mplatr

  Specifies the order in which tasks wait in the waiting task queue of the memory pool to be created as a variable-length memory pool attribute. The values that can be specified as the variable-length memory pool attribute are as follows:

Figure 13-10. Variable-Length Memory Pool Attribute



Bit 0 specifies the order in which tasks wait for a memory block from the memory pool to be created. If TA_TFIFO is specified by mplatr, the task waits on an FIFO basis. If TA_TPRI is specified, the task waits according to the priority.

- mplsz

  Specifies the size (in bytes) of the variable-length memory pool to be created. This value must be an integral multiple of 8. If any other value is specified, a size that is the lowest integral multiple of 8 greater than the specified size is assumed.

  When a memory pool is actually created, it is greater than the value specified by mplsz because a header area that manages all the memory pools is necessary in addition to the area of the main body of the memory pool.
- mpl

  This is a reserved area. Always specify NULL. Anything other than NULL is ignored even if specified.

[Differences from μITRON3.0]

The extended data exinf has been deleted from the members of the variable-length memory pool creation packet T_CMPL, and the reserved area mpl has been added. In addition, the type of the memory pool size mplsz has been changed from INT to SIZE. SIZE is a type newly defined for μITRON4.0.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	cre_mpl is not included in the system
E_RSATR	−11	The variable-length memory pool attribute is illegal
E_PAR	−17	The parameter is illegal − The size of the memory pool is illegal (mplsz = 0)
E_ID	−18	The variable-length memory pool ID number is outside the range (mplid ≤ 0, maximum variable-length memory pool count < mplid)
E_CTX	−25	cre_mpl was issued from a non-task context cre_mpl was issued in the CPU lock state
E_NOMEM	−33	The area for the memory pool cannot be reserved
E_OBJ	−41	A variable-length memory pool with the same ID number already exists

acre_mpl

Create Variable-Size Memory Pool with Automatic ID Assignment

[Overview]

Creates a variable-length memory pool (Automatic assignment of ID number)

[C format]

```
#include <kernel.h>
ER_ID acre_mpl(T_CMPL *pk_cmpl);
```

[Parameter]

I/O	Parameter		Description
I	T_CMPL *	pk_cmpl	Address of variable-length memory pool creation packet

[Explanation]

A variable-length memory pool is created based on the information stored in the variable-length memory pool creation packet pk_cmpl, and the ID number of the memory pool is returned. In other words, a variable-length memory pool control block that is available and has not been created is searched, assigned, and initialized, the area of the main body of the memory pool is reserved from the user pool, and the ID number of the memory pool is returned. If the return value is negative, an error occurs. For the meaning of the return value, refer to **[Return values]** below. For details of the variable-length memory pool creation packet, refer to the description of cre_mpl.

[Differences from μITRON3.0]

acre_mpl is a newly created service call equivalent to cre_mpl, but with the addition of an automatic ID number assignment function.

[Return values]

Symbol	Value	Meaning
(Positive integer)		ID number of the variable-length memory pool created (normal termination)
E_RSFN	−10	acre_mpl is not included in the system
E_RSATR	−11	The variable-length memory pool attribute is illegal
E_PAR	−17	The parameter is illegal – The size of the memory pool is illegal (mplsz = 0)
E_CTX	−25	acre_mpl was issued from a non-task context acre_mpl was issued in the CPU lock state
E_NOMEM	−33	The area for the memory pool cannot be reserved
E_NOID	−34	The ID number cannot be allocated (reserving a control block has failed)

del_mpl

Delete Variable-Size Memory Pool

[Overview]

Deletes a variable-length memory pool

[C format]

```
#include <kernel.h>
ER del_mpl(ID mplid);
```

[Parameter]

I/O	Parameter		Description
I	ID	mplid	ID number of variable-length memory pool to be deleted

[Explanation]

The control block of the variable-length memory pool specified by mplid is invalidated, the area of the main body of the variable-length memory pool is returned to the user pool, and the variable-length memory pool is deleted. After deletion, a new variable-length memory pool with the same ID number can be created.

The memory pool is deleted regardless of whether memory blocks acquired from the variable-length memory pool to be deleted have been released or not. Note, therefore, that deletion of the memory pool is not reported to the task that is reserving (using) the memory blocks.

If tasks waiting to acquire a memory block from the variable-length memory pool exist, all the tasks are released from the waiting state. The value E_DLT indicating that the variable-length memory pool has been deleted is returned for the error code of the service call (t)get_mpl that caused the tasks to wait.

[Differences from μITRON3.0]

None

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	del_mpl is not included in the system
E_ID	−18	The variable-length memory pool ID number is outside the range (mplid ≤ 0, maximum variable-length memory pool count < mplid)
E_CTX	−25	del_mpl was issued from a non-task context del_mpl was issued in the CPU lock state
E_NOEXS	−42	The target variable-length memory pool does not exist

rel_mpl/irel_mpl

Release Memory Block to Variable-Size Memory Pool

[Overview]

Returns a memory block

[C format]

```
#include <kernel.h>
ER rel_mpl(ID mplid, VP blk);
```

[Parameters]

I/O	Parameter		Description
I	ID	mplid	ID number of variable-length memory pool to which memory block is to be returned
I	VP	blk	Address of memory block to be returned

[Explanation]

The memory block specified by blk is returned to the variable-length memory pool specified by mplid. If a task is waiting for a memory block from the specified memory pool, this task is allowed to acquire the released memory block and is released from the waiting state.

The memory pool to which the memory block is to be returned must be the same memory pool from which the memory block was acquired. If the memory block is returned to a different memory pool, an error occurs and the error code E_PAR is returned. The operation is not guaranteed if rel_mpl is issued to an area other than a memory block area. If the memory block to be returned is registered to a mailbox as a message, the operation is not guaranteed.

irel_mpl is intended to be issued from a non-task context but it can also be issued from a task context. rel_mpl can be issued from a non-task context.

[Differences from μITRON3.0]

The name of rel_blk has been changed to rel_mpl.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	rel_mpl/irel_mpl is not included in the system
E_PAR	−17	The parameter is illegal − The source and return destination of the memory block are different
E_ID	−18	The variable-length memory pool ID number is outside the range (mplid ≤ 0, maximum variable-length memory pool count < mplid)
E_CTX	−25	rel_mpl/irel_mpl was issued in the CPU lock state
E_NOEXS	−42	The target variable-length memory pool does not exist

get_mpl

Get Memory Block from Variable-Size Memory Pool

[Overview]

Acquires a memory block

[C format]

```
#include <kernel.h>
ER get_mpl(ID mplid, UINT blksz, VP *p_blk);
```

[Parameters]

I/O	Parameter		Description
I	ID	mplid	ID number of variable-length memory pool from which memory block is to be acquired
I	UINT	blksz	Requested block size [bytes]
O	VP *	p_blk	Address to store address of memory block

[Explanation]

A memory block of the size specified by blksz is acquired from the variable-length memory pool specified by mplid, and the first address of the memory block is stored in the address specified by p_blk.

The requested size blksz must be a value that is an integral multiple of 8. If any other value is specified, a size that is the lowest integral multiple of 8 greater than blksz is automatically assumed. The actual size of the memory block extracted from the memory pool is greater than the specified size because a header area that manages the memory block is included.

If no vacant memory block exists in the memory pool when get_mpl is issued, the task waits for the memory block and is registered to the waiting task queue of the memory pool until it can acquire the memory block. The task is registered to the queue in the order (on an FIFO basis or according to priority) specified when the memory pool was created. If a memory block is released by rel_mpl, it is checked whether a waiting task can get the memory block in the order in which the tasks were placed in the queue. If two or more tasks are waiting, therefore, they may be blocked by a task requesting a larger size. Consequently, the tasks may be kept waiting even though the requested size is smaller than the vacant area of the memory pool.

Tasks placed in the waiting state due to get_mpl are released from waiting by one of the following occurrences.

1. (i)rel_mpl is issued and the task acquires a memory block (E_OK).
2. A blocking task is released from the waiting task queue and acquires a memory block (E_OK).
3. (i)rel_wai is issued and the task is forcibly released from the waiting state (E_RLWAI).
4. del_mpl is issued and the memory pool is deleted (E_DLT).

[Differences from μITRON3.0]

The name of get_blk has been changed to get_mpl. In addition, the order of the parameters has also been changed.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	get_mpl is not included in the system
E_PAR	−17	The parameter is illegal − The memory block size is illegal (blksz = 0)
E_ID	−18	The variable-length memory pool ID number is outside the range (mplid ≤ 0, maximum variable-length memory pool count < mplid)
E_CTX	−25	get_mpl was issued from a non-task context get_mpl was issued in the CPU lock state get_mpl was issued in the dispatch disabled state
E_NOEXS	−42	The target variable-length memory pool does not exist
E_RLWAI	−49	The task is forcibly released from the waiting state by (i)rel_wai
E_DLT	−51	The variable-length memory pool is deleted while a task is waiting

pget_mpl/ipget_mpl

Poll and Get Memory Block from Variable-Size Memory Pool

[Overview]

Acquires a memory block (Polling)

[C format]

```
#include <kernel.h>
ER pget_mpl(ID mplid, UINT blksz, VP *p_blk);
```

[Parameters]

I/O	Parameter		Description
I	ID	mplid	ID number of variable-length memory pool from which memory block is to be acquired
I	UINT	blksz	Requested block size [bytes]
O	VP *	p_blk	Address to store address of memory block

[Explanation]

A memory block of the size specified by blksz is acquired from the variable-length memory pool specified by mplid, and the first address of the memory block is stored in the address specified by p_blk. If no vacant memory block exists in the memory pool when get_mpl is issued, an error occurs and the error code E_TMOUT is returned.

The requested size blksz must be a value that is an integral multiple of 8. If any other value is specified, a size that is the lowest integral multiple of 8 greater than blksz is automatically assumed. The actual size of the memory block extracted from the memory pool is greater than the specified size because a header area that manages the memory block is included.

Even if the memory pool has a vacant area large enough to be acquired, if the attribute of the memory pool is TA_TFIFO and a waiting task exists, or if the memory pool attribute is TA_TPRI, a waiting task exists, and the priority of the waiting task is higher than the task that has issued get_mpl, acquiring the memory block fails and the error code E_TMOUT is returned.

ipget_mpl is intended to be issued from a non-task context but it can also be issued from a task context. pget_mpl can be issued from a non-task context.

[Differences from μITRON3.0]

The name of pget_blk has been changed to pget_mpl. In addition, the order of the parameters has also been changed.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	pget_mpl/ipget_mpl is not included in the system
E_PAR	−17	The parameter is illegal − The memory block size is illegal (blksz = 0)
E_ID	−18	The variable-length memory pool ID number is outside the range (mplid ≤ 0, maximum variable-length memory pool count < mplid)
E_CTX	−25	pget_mpl/ipget_mpl was issued in the CPU lock state
E_NOEXS	−42	The target variable-length memory pool does not exist
E_TMOUT	−50	Polling has failed

tget_mpl

Get Memory Block from Variable-Size Memory Pool with Timeout

[Overview]

Acquires a memory block (with timeout)

[C format]

```
#include <kernel.h>
ER tget_mpl(ID mplid, UINT blksz, VP *p_blk, TMO tmout);
```

[Parameters]

I/O	Parameter		Description
I	ID	mplid	ID number of variable-length memory pool from which memory block is to be acquired
I	UINT	blksz	Requested block size [bytes]
O	VP *	p_blk	Address to store address of memory block
I	TMO	tmout	Timeout time [milliseconds]

[Explanation]

A memory block of the size specified by blksz is acquired from the variable-length memory pool specified by mplid, and the first address of the memory block is stored in the address specified by p_blk.

The requested size blksz must be a value that is an integral multiple of 8. If any other value is specified, a size that is the lowest integral multiple of 8 greater than blksz is automatically assumed. The actual size of the memory block extracted from the memory pool is greater than the specified size because a header area that manages the memory block is included.

If no vacant memory block exists in the memory pool when tget_mpl is issued, the task waits for the memory block and is registered to the waiting task queue of the memory pool until it can get the memory block or the time specified by tmout elapses. The task is registered to the queue in the order (on an FIFO basis or according to priority) specified when the memory pool was created. If a memory block is released by rel_mpl, it is checked whether a waiting task can get the memory block in the order in which the tasks are placed in the queue. If two or more tasks are waiting, therefore, they may be blocked by a task requesting a larger size. Consequently, the tasks may be kept waiting even though the requested size is smaller than the vacant area of the memory pool.

If TMO_POL = 0 is specified as tmout, 0 is specified as the timeout time and tget_mpl performs the same operation as pget_mpl. If TMO_FEVR = −1 is specified as tmout, the timeout time is specified to be infinite and the service call performs the same operation as get_mpl.

Tasks placed in the waiting state due to tget_mpl are released from waiting by one of the following occurrences.

1. (i)rel_mpl is issued and the task acquires a memory block (E_OK).
2. The blocking task is released from the waiting task queue and acquires a memory block (E_OK).
3. The specified time elapses and timeout occurs (E_TMOUT).
4. (i)rel_wai is issued and the task is forcibly released from the waiting state (E_RLWAI).
5. del_mpl is issued and the memory pool is deleted (E_DLT).

[Differences from μITRON3.0]

The name of tget_blf has been changed to tget_mpl. In addition, the order of the parameters has also been changed.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	tget_mpl is not included in the system
E_PAR	−17	The parameter is illegal − The memory block size is illegal (blksz = 0) − The timeout time is illegal (tmout < TMO_FEVR = −1)
E_ID	−18	The variable-length memory pool ID number is outside the range (mplid ≤ 0, maximum variable-length memory pool count < mplid)
E_CTX	−25	tget_mpl was issued from a non-task context tget_mpl was issued in the CPU lock state tget_mpl was issued in the dispatch disabled state
E_NOEXS	−42	The target variable-length memory pool does not exist
E_RLWAI	−49	The task is forcibly released from the waiting state by (i)rel_wai
E_TMOUT	−50	Timeout
E_DLT	−51	The variable-length memory pool is deleted while a task is waiting

Refer Variable-Size Memory Pool Status

ref_mpl/iref_mpl

[Overview]

Obtains memory pool information

[C format]

```
#include <kernel.h>
ER ref_mpl(ID mplid, T_RMPL *pk_rmpl);
```

[Parameters]

I/O	Parameter		Description
I	ID	mplid	ID number of variable-length memory pool whose information is to be obtained
O	T_RMPL *	pk_rmpl	Address of variable-length memory pool information packet

Configuration of T_RMPL

```
typedef struct t_rmpl {
        ID      wtskid;     /* Presence/absence of waiting task */
        SIZE    fmplsz;     /* Total size of vacant area */
        UINT    fblksz;     /* Maximum size of block that can be acquired */
        ATR     mplatr;     /* Variable-length memory pool attribute */
} T_RMPL;
```

[Explanation]

Information is obtained on the variable-length memory pool specified by mplid and stored in the packet specified by pk_rmpl. The variable-length memory pool information is described in detail below.

- wtskid
  Indicates the presence or absence of tasks waiting for a memory block from the target variable-length memory pool. If waiting tasks exist, the ID number of the first task in the waiting task queue is stored. If no waiting task exists, TSK_NONE = 0 is stored.
- fmplsz
  Stores the total of the vacant area that can be acquired as a memory block from the specified memory pool. As memory blocks are repeatedly acquired and released from a memory pool, the vacant areas may exist in the memory pool like islands. Therefore, a memory block of size fmplsz may not always be able to be acquired.
- fblksz
  Stores the size of the largest block of the vacant memory blocks that are available from the specified memory pool.
- mplatr
  Stores the attribute of the target variable-length memory pool. For the meaning of the stored value, refer to the description of cre_mpl.

iref_mpl is intended to be issued from a non-task context but it can also be issued from a task context. ref_mpl can be issued from a non-task context.

[Differences from μITRON3.0]

The extended data exinf has been deleted from the members of the variable-length memory pool information. In addition, the type of wtskid indicating the presence or absence of a waiting task has been changed from BOOL_ID to ID, and the type of the total size of available area fmplsz and the maximum size of the available block fblksz has been changed from INT to SIZE and UNIT. SIZE is a type newly defined for μITRON4.0.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ref_mpl/iref_mpl is not included in the system
E_ID	−18	The variable-length memory pool ID number is outside the range (mplid ≤ 0, maximum variable-length memory pool count < mplid)
E_CTX	−25	ref_mpl/iref_mpl was issued in the CPU lock state
E_NOEXS	−42	The target variable-length memory pool does not exist

13.8.6 Time management function service calls

This section explains the time management function service calls listed in the following table.

Table 13-13. Time Management Function Service calls

Name	Function
set_tim/iset_tim	Sets the system time
get_tim/iget_tim	Obtains the system time
isig_tim	Supplies the time tick (issued from ISR)
cre_cyc	Creates a cyclic handler
acre_cyc	Creates a cyclic handler (automatic assignment of ID No.)
del_cyc	Deletes a cyclic handler
sta_cyc/ista_cyc	Activates a cyclic handler
stp_cyc/istp_cyc	Stops a cyclic handler
ref_cyc/iref_cyc	Obtains cyclic handler information

ISR: Interrupt Service Routine

Set System Time

set_tim/iset_tim

[Overview]

Sets the time of the system

[C format]

```
#include <kernel.h>
ER set_tim(SYSTIM *p_systim);
```

[Parameter]

I/O	Parameter		Description
I	SYSTIM *	p_systim	Address of system time packet

Configuration of SYSTIM

```
typedef struct t_systim {
        UW              ltime;          /* Time (lower 32 bits)*/
        UW              utime;          /* Time (higher 32 bits)*/
} SYSTIM;
```

[Explanation]

The time of the system is set to the time stored in p_systim. Even if the time has been changed, it does not mean that the time difference between the previous time and new time has elapsed, so the timeout of tasks and cyclic handlers is not affected at all.

iset_tim is intended to be issued from a non-task context but it can also be issued from a task context. set_tim can be issued from a non-task context.

[Differences from μITRON3.0]

The type of the system time packet has been changed from SYSTIME to SYSTIM.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	set_tim/iset_tim is not included in the system
E_CTX	−25	set_tim/iset_tim was issued in the CPU lock state

get_tim/iget_tim

Get System Time

[Overview]

Obtains the time of the system

[C format]

```
#include <kernel.h>
ER get_tim(SYSTIM *p_systim);
```

[Parameter]

I/O	Parameter		Description
O	SYSTIM *	p_systim	Address of system time packet

Configuration of SYSTIM

```
typedef struct t_systim {
        UW          ltime;          /* Time (lower 32 bits)*/
        UW          utime;          /* Time (higher 32 bits)*/
} SYSTIM;
```

[Explanation]

The current time of the system is obtained and stored in the time packet p_systim.

iget_tim is intended to be issued from a non-task context but it can also be issued from a task context. get_tim can be issued from a non-task context.

[Differences from μITRON3.0]

The type of the system time packet has been changed from SYSTIME to SYSTIM.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	get_tim/iget_tim is not included in the system
E_CTX	−25	get_tim/iget_tim was issued in the CPU lock state

isig_tim

Signal Clock Tick

[Overview]

Supplies a time tick (Issued from an interrupt service routine)

[C format]

```
#include <kernel.h>
ER isig_tim(void);
```

[Parameter]

None

[Explanation]

The kernel is notified that the time of 1 tick of the timer interrupt has elapsed.

Therefore, the kernel judges that the basic clock cycle specified by the static API DEF_TIM has elapsed when isig_tim is issued once, and updates the system time, checks the timeout of waiting tasks, and activates a cyclic handler.

The user must therefore set an interrupt to be generated in every basic clock cycle, by using the timer of the CPU, and describe an interrupt service routine corresponding to the interrupt.

Processing such as timeout of a task and activation of a cyclic handler is performed in batch when execution returns from an interrupt. Therefore, do not describe processing that expects completion of processing by the kernel (timeout of a task occurs or the cyclic handler has been activated) between when isig_tim is issued and when the interrupt service routine is terminated.

[Differences from μITRON3.0]

isig_tim is a newly created service call. With μITRON4.0, the lapse of time must be explicitly reported to the kernel by using isig_tim.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	isig_tim is not included in the system
E_CTX	−25	isig_tim was issued from a task context isig_tim was issued in the CPU lock state

Create Cyclic Handler

cre_cyc

[Overview]

Creates a cyclic handler

[C format]

```
#include <kernel.h>
ER cre_cyc(ID cycid, T_CCYC *pk_ccyc);
```

[Parameters]

I/O	Parameter		Description
I	ID	cycid	ID number of cyclic handler to be created
I	T_CCYC *	pk_ccyc	Address of cyclic handler creation packet

Configuration of T_CCYC

```
typedef struct t_ccyc {
        ATR         cycatr;     /* Cyclic handler attribute */
        VP_INT      exinf;      /* Extended data */
        FP          cychdr;     /* Activation address of cyclic handler */
        RELTIM      cyctim;     /* Activation time interval of cyclic handler */
        RELTIM      cycphs;     /* Phase of cyclic handler */
        VP          gp;         /* PID base address of cyclic handler */
        VP          tp;         /* Reserved area */
} T_CCYC;
```

[Explanation]

A cyclic handler with the an ID number specified by cycid is created based on the information stored in the cyclic handler creation packet pk_ccyc. In other words, a control block is assigned to the cyclic handler to be created, and the control block is initialized.

The cyclic handler creation packet T_CCYC is explained in detail below.

- cycatr

  Specifies the initial status of the cyclic handler immediately after the handler has been created, and the description language, as cyclic handler attributes.

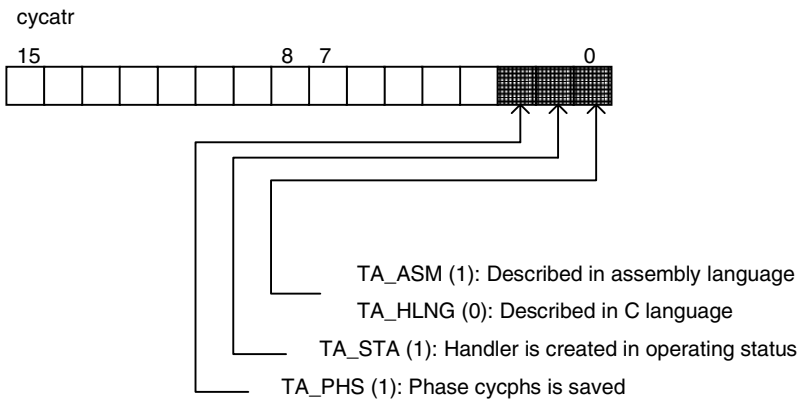
  Bit 0 specifies the language in which the handler is described. TA_HLNG means the handler is described in C language and TA_ASM means assembly language. With the current version, however, these two attributes are not differentiated in the kernel processing.

  Bit 1 specifies the status of the cyclic handler immediately after it has been created. If TA_STA is specified, the handler is created in the operating status. If it is not specified, the handler is created in the stopped status.

Bit 2 specifies whether the created cyclic handler saves the phase cycphs to be explained below. If TA_PHS is specified, the phase is saved. By issuing sta_cyc after the handler has been created, the phase cycphs becomes valid when the handler is activated.

If two or more attributes are specified at the same time, set the logical sum of the attribute values to cycatr.

Figure 13-11. Cyclic Handler Attribute



- exinf

  Sets user-defined information on the cyclic handler to be created. This value is passed to the cyclic handler when it has been created.
- cychdr

  Specifies the activation address of the cyclic handler to be created.
- cyctim

  Specifies the interval in milliseconds at which the created cyclic handler is to be activated.
- cycphs

  Specifies the phase of the cyclic handler to be created in milliseconds. This means that the created cyclic handler is activated cycphs [milliseconds] after cre_cyc has been issued. This phase can be saved by specifying TA_PHS as the cyclic handler attribute. In other words, the activation timing of the cyclic handler can be fixed, regardless of changes in status after the cyclic handler has been created.

  cycphs is meaningless if neither TA_STA nor TA_PHS is given as a handler attribute.
- gp

  Specifies the base address (gp) of PID (Position Independent Data) used by the cyclic handler to be created. Set NULL if the cyclic handler does not use PID.
- tp

  This is a reserved area. Always set NULL to this area. However, a value other than NULL is ignored even if set.

[Differences from μITRON3.0]

1. Change of terminology and service call name

   “Defining cyclic handler” → “Creating cyclic handler”

   “Cyclically activated handler definition number” → “Cyclic handler ID number”

   def_cyc → cre_cyc

   “Activated status” → “Cyclic handler status” or “status”

   Contents of status: “ON status” → “Operating (STA) status”

   “OFF status” → “Stop (STP) status”
2. An activation phase can be specified for the cyclic handler.
3. Explicit specification by an attribute is not necessary even when the cyclic handler uses PID.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	cre_cyc is not included in the system
E_RSATR	−11	The cyclic handler attribute is illegal
E_PAR	−17	The parameter is illegal – The activation phase is illegal (cyctim = 0) – The activation phase is outside the range (cycphs = 0)
E_ID	−18	The ID number of the cyclic handler is outside the range (cycid ≤ 0, maximum cyclic handler count < 0)
E_CTX	−25	cre_cyc was issued from a non-task context cre_cyc was issued in the CPU lock state
E_OBJ	−41	A cyclic handler with the same ID number already exists

Create Cyclic Handler with Automatic ID Assignment

acre_cyc

[Overview]

Creates a cyclic handler (Automatic assignment of ID number)

[C format]

```
#include <kernel.h>
ER_ID acre_cyc(T_CCYC *pk_ccyc);
```

[Parameter]

I/O	Parameter		Description
I	T_CCYC *	pk_ccyc	Address of cyclic handler creation packet

[Explanation]

A cyclic handler is created based on the information stored in the cyclic handler creation packet pk_ccyc, and the ID number of the handler is returned. In other words, a cyclic handler control block that is available but has not been created is searched, assigned, and initialized, and the ID number of the block is returned. If the return value is negative, an error occurs. For the meaning of the return value, refer to **[Return values]** below.

For details of the cyclic handler creation packet T_CCYC, refer to the description of cre_cyc.

[Differences from μITRON3.0]

acre_cyc is a newly created service call equivalent to cre_cyc, but with of the addition of an automatic ID number assignment function.

[Return values]

Symbol	Value	Meaning
(Positive integer)		ID number of created cyclic handler (normal termination)
E_RSFN	−10	acre_cyc is not included in the system
E_RSATR	−11	The cyclic handler attribute is illegal
E_PAR	−17	The parameter is illegal − The activation phase is illegal (cyctim = 0) − The activation phase is illegal (cycphs = 0)
E_CTX	−25	acre_cyc was issued from a non-task context acre_cyc was issued in the CPU lock state
E_NOID	−34	The ID number cannot be assigned (reserving a control block has failed)

del_cyc

Delete Cyclic Handler

[Overview]

Deletes a cyclic handler

[C format]

```
#include <kernel.h>
ER del_cyc(ID cycid);
```

[Parameter]

I/O	Parameter		Description
I	ID	cycid	ID number of cyclic handler to be deleted

[Explanation]

The control block of the cyclic handler specified by cycid is invalidated, and the cyclic handler is deleted. After deletion, a new cyclic handler can be created using the same ID number.

[Differences from μITRON3.0]

del_cyc is a newly created service call that can take the place of def_cyc (*cycno*, NADR).

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	del_cyc is not included in the system
E_ID	−18	The ID number of the cyclic handler is outside the range (cycid ≤ 0, maximum cyclic handler count < cycid)
E_CTX	−25	del_cyc was issued from a non-task context del_cyc was issued in the CPU lock state
E_NOEXS	−42	The target handler does not exist

Start Cyclic Handler

sta_cyc/ista_cyc

[Overview]

Activates a cyclic handler

[C format]

```
#include <kernel.h>
ER sta_cyc(ID cycid);
```

[Parameter]

I/O	Parameter		Description
I	ID	cycid	ID number of cyclic handler to be activated

[Explanation]

The cyclic handler specified by cycid is placed in the operable status so that the handler is ready to be activated.

After issuance of sta_cyc, the specified cyclic handler is activated at the timing determined by the saved phase if the handler has the attribute TA_PHS (refer to 7.5.6). If the handler does not have TA_PHS, the handler is activated when the activation cycle has elapsed after issuance of sta_cyc. Even if this service call has been issued to a cyclic handler already in the operable status, therefore, the activation time is re-set if the handler does not have the attribute TA_PHS.

ista_cyc is intended to be issued from a non-task context but it can also be issued from a task context. sta_cyc can be issued from a non-task context.

[Differences from μITRON3.0]

sta_cyc/ista_cyc is a newly created service call, and its function is equivalent to act_cyc (*cycno*, TCY_ON|TCY_INI) or act_cyc (*cycno*, TCY_ON).

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	sta_cyc/ista_cyc is not included in the system
E_ID	−18	The ID number of the cyclic handler is outside the range (cycid ≤ 0, maximum cyclic handler count < cycid)
E_CTX	−25	sta_cyc/ista_cyc was issued in the CPU lock state
E_NOEXS	−42	The target handler does not exist

Stop Cyclic Handler

stp_cyc/istp_cyc

[Overview]

Stops a cyclic handler

[C format]

```
#include <kernel.h>
ER stp_cyc(ID cycid);
```

[Parameter]

I/O	Parameter		Description
I	ID	cycid	ID number of a cyclic handler to be stopped

[Explanation]

The cyclic handler specified by cycid is placed in the stopped status. Consequently, the handler is not activated even when the cyclic time has elapsed. If the specified cyclic handler is already in the stopped status, nothing is executed and no error occurs.

istp_cyc is intended to be issued from a non-task context but it can also be issued from a task context. stp_cyc can be issued from a non-task context.

[Differences from μITRON3.0]

stp_cyc/istp_cyc is a newly created service call, and its function is equivalent to act_cyc (*cycno*, TCY_OFF).

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	stp_cyc/istp_cyc is not included in the system
E_ID	−18	The ID number of the cyclic handler is outside the range (cycid ≤ 0, maximum cyclic handler count < cycid)
E_CTX	−25	stp_cyc/istp_cyc was issued in the CPU lock state
E_NOEXS	−42	The target handler does not exist

Refer Cyclic Handler Status

ref_cyc/iref_cyc

[Overview]

Obtains cyclic handler information

[C format]

```
#include <kernel.h>
ER ref_cyc(ID cycid, T_RCYC *pk_rcyc);
```

[Parameters]

I/O	Parameter		Description
I	ID	cycid	ID number of cyclic handler whose information is to be obtained
O	T_RCYC *	pk_rcyc	Address of cyclic handler information packet

Configuration of T_RCYC

```
typedef struct t_rcyc {
        STAT        cycstat;    /* Operation status of cyclic handler */
        RELTIM      lefttim;    /* Time up to next activation */
        ATR         cycatr;     /* Cyclic handler attribute */
        RELTIM      cyctim;     /* Activation interval */
        RELTIM      cycphs;     /* Activation phase */
} T_RCYC;
```

[Explanation]

Information is obtained on the cyclic handler specified by cycid and stored in the packet specified by pk_rcyc. The cyclic handler information is described in detail below.

- cycstat
  Stores a value indicating the current status of the cyclic handler. If this value is TCYC_STP(0), it means that the handler stops. If it is TCYC_STA(1), it means that the handler is operating (this does not mean that the handler is under execution).
- lefttim
  Stores the time up to the next activation of the cyclic handler in milliseconds.
  If lefttime = 0, it means that the time is less than the timer interrupt input interval (the cyclic handler is activated when isig_tim is issued next time), and does not mean that the handler is “under execution” or “stopped”.
- cycatr
  Stores the attribute of the specified cyclic handler. For the meaning of the value stored, refer to the description of cre_cyc.

- cyctim
  Stores the activation interval of the specified cyclic handler (in milliseconds).
- cycphs
  Stores the activation phase of the specified cyclic handler (in milliseconds).

iref_cyc is intended to be issued from a non-task context but it can also be issued from a task context. ref_cyc can be issued from a non-task context.

[Differences from μITRON3.0]

1. The specification order of the parameters has been changed.
2. The cyclic handler specification number is called a cyclic handler ID number, and its type has been changed from HNO to ID.
3. In connection with the above, the error code E_ID that may be returned has been added.
4. “Activated status” is called cyclic handler status or just status. Each status has been also changed from ON to STA and from OFF to STP.
5. Extended data exinf has been deleted from the members of the cyclic handler information packet T_RCYC, and cyclic handler attribute cycatr and phase cycphs have been added.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ref_cyc/iref_cyc is not included in the system
E_ID	−18	The ID number of the cyclic handler is outside the range (cycid ≤ 0, maximum cyclic handler count < cycid)
E_CTX	−25	ref_cyc/iref_cyc was issued from a non-task context ref_cyc/iref_cyc was issued in the CPU lock state
E_NOEXS	−42	The target handler does not exist

13.8.7 System status management function service calls

This section explains the system status management function service calls listed in the following table.

Table 13-14. System Status Management Function Service calls

Name	Function
rot_rdq/irot_rdq	Rotates a ready queue
get_tid/iget_tid	Obtains the ID number of a task in the running state
loc_cpu/iloc_cpu	Locks the CPU of the system
unl_cpu/iunl_cpu	Releases the CPU lock of the system
dis_dsp	Disables dispatch
ena_dsp	Enables dispatch
sns_ctx	References the execution context
sns_loc	Checks whether the CPU is locked
sns_dsp	Checks whether dispatch is disabled
sns_dpn	Checks whether dispatch is pending

rot_rdq/irot_rdq

Rotate Ready Queue

[Overview]

Rotates a ready queue

[C format]

```
#include <kernel.h>
ER rot_rdq(PRI tskpri);
```

[Parameter]

I/O	Parameter		Description
I	PRI	tskpri	Priority according to which ready queue is rotated.

[Explanation]

A ready queue is rotated corresponding to the priority specified by tskpri. In other words, the first task in the ready queue is moved to the end of the queue to change the execution sequence of the tasks with the same priority. By issuing rot_rdq at fixed time intervals, therefore, round-robin scheduling can be implemented.

If TPRI_SELF = 0 is set for tskpri, the ready queue corresponding to the base priority of the issuing task can be rotated. If rot_rdq is issued with TPRI_SELF or the priority of the issuing task directly specified, the task in the running state moves to the end of the ready queue. Consequently, the task relinquishes the right of execution. If rot_rdq is issued with TPRI_SELF specified while dispatch is disabled, only the ready queue is manipulated and the task continues its processing.

If the priority that only one task is placed in the ready queue or the priority that no task is placed in the queue is specified when rot_rdq is issued, nothing happens and no error occurs.

To issue this service call from a non-task context such as an interrupt service routine, use irot_rdq.

irot_rdq is intended to be issued from a non-task context but it can also be issued from a task context. rot_rdq can be issued from a non-task context.

[Differences from μITRON3.0]

TPRI_RUN is deleted and TPRI_SELF is newly provided.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	rot_rdq/irot_rdq is not included in the system
E_PAR	−17	The priority is outside the range (tskpri < 0, maximum priority value < tskpri)
E_CTX	−25	rot_rdq/irot_rdq was issued in the CPU lock state

Get Task Identifier

get_tid/iget_tid

[Overview]

Obtains the ID number of the task in the running state

[C format]

```
#include <kernel.h>
ER get_tid(ID * p_tskid);
```

[Parameter]

I/O	Parameter		Description
O	ID *	p_tskid	Address to store task ID number

[Explanation]

The ID number of the task in the running state, i.e., the ID number of the issuing task is stored in the area specified by p_tskid. If this service call is issued from an interrupt service routine that has been activated in the idle state, TSK_NONE = 0 is stored in p_tskid.

iget_tid is intended to be issued from a non-task context but it can also be issued from a task context. get_tid can be issued from a non-task context.

[Differences from μITRON3.0]

The meaning (definition) of the service call has been changed from “obtaining the ID number of the issuing task” to “obtaining the ID number of the task in the running state”. Consequently, even if this service call is issued from a non-task block such as an interrupt service routine, no error occurs and the task ID number can be obtained. No functional change has been made.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	get_tid/iget_tid is not included in the system
E_CTX	−25	get_tid/iget_tid was issued from a non-task context get_tid/iget_tid was issued in the CPU lock state

Lock CPU

loc_cpu/iloc_cpu

[Overview]

Locks the CPU of the system

[C format]

```
#include <kernel.h>
ER loc_cpu(void);
```

[Parameter]

None

[Explanation]

Interrupts and dispatch are disabled, and the CPU of the system is locked. While the CPU is locked, no service call other than loc_cpu, unl_cpu, and sns_xxx can be issued. If any service call other than these is issued while the CPU is locked, the service call returns the error code E_CTX.

If loc_cpu is issued while the CPU is locked, the lock status of the CPU merely continues and no error occurs. Even if loc_cpu is issued more than once, the lock of CPU is released by the first unl_cpu.

It is assumed that timer operations such as delaying tasks or cyclic handlers are processed by a software timer using a timer interrupt. While interrupts are disabled, therefore, these operations are not performed. If interrupts are disabled for too long a time, the wait time specified for these operations cannot be guaranteed.

Interrupts are disabled by loc_cpu, by clearing the IE bit of the status register. The operation is not guaranteed if the status register is directly manipulated and the IE bit is set while the CPU is locked. The interrupt mask (IM bit) is not affected.

iloc_cpu is intended to be issued from a non-task context but it can also be issued from a task context. loc_cpu can be issued from a non-task context.

[Differences from μITRON3.0]

Issuing service calls, with some exceptions, is disabled while the CPU is locked. In addition, disabling dispatch by loc_cpu is distinguished from disabling dispatch by dis_dsp.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	loc_cpu/iloc_cpu is not included in the system

Unlock CPU

unl_cpu/iunl_cpu

[Overview]

Releases the CPU lock of the system

[C format]

```
#include <kernel.h>
ER unl_cpu(void);
```

[Parameter]

None

[Explanation]

The CPU lock of the system is released and interrupts and dispatch are enabled. If the system was disabled by dis_dsp from dispatching tasks before the CPU was locked, however, only interrupts are enabled by issuance of unl_cpu. The interrupt mask (IM field of the status register) is not affected.

iunl_cpu is intended to be issued from a non-task context but it can also be issued from a task context. unl_cpu can be issued from a non-task context.

[Differences from μITRON3.0]

μITRON3.0 unconditionally enables interrupts and dispatch by unl_cpc. μITRON4.0 returns the status to that before loc_cpu was issued.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	unl_cpu/iunl_cpu is not included in the system
E_CTX	−25	unl_cpu/iunl_cpu was issued from a non-task context

dis_dsp

Disable Dispatch

[Overview]

Disables dispatch

[C format]

```
#include <kernel.h>
ER dis_dsp(void);
```

[Parameter]

None

[Explanation]

Dispatch of tasks is disabled until ena_dsp is issued. Therefore, tasks will not change their state from running to ready. They cannot enter the waiting state, either. However, because interrupts are not disabled, an interrupt handler can be activated even while dispatch is disabled. Therefore, there is no possibility that a task is preempted by another task, though it may be preempted by an interrupt handler.

Therefore, even if a task with a priority higher than that of the issuing task is placed in the ready state by a service call issued from an interrupt handler or the task itself, dispatch to that task does not take place and is postponed until dispatch is enabled.

If a service call that may keep a task waiting is issued while dispatch is disabled, an error occurs and the error code E_CTX is returned. Similarly, if sus_tsk is issued from an interrupt handler to a task in the running state, the error code E_CTX is returned.

If dis_dsp is issued again while dispatch is already disabled, the dispatch disabled state merely continues and no error occurs. Even if dis_dsp is issued more than once, however, dispatch is enabled by issuing ena_dsp only once.

[Differences from μITRON3.0]

Disabling dispatch by dis_dsp and disabling dispatch by loc_cpu are distinguished.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	dis_dsp is not included in the system
E_CTX	−25	dis_dsp was issued from a non-task context dis_dsp was issued in the CPU lock state

Enable Dispatch

ena_dsp

[Overview]

Enables dispatch

[C format]

```
#include <kernel.h>
ER ena_dsp(void);
```

[Parameter]

None

[Explanation]

Dispatch of tasks is enabled. In other words, dispatch disabled by dis_dsp is enabled. Consequently, dispatch that has been postponed by a service call issued from an interrupt handler or the issuing task, such as when a task with a higher priority than the task itself is in the ready state, is processed after this service call has been issued.

Even if a task that is not in the dispatch disabled state issues ena_dsp, the status in which dispatch is enabled merely continues and no error occurs.

[Differences from μITRON3.0]

None

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ena_dsp is not included in the system
E_CTX	−25	ena_dsp was issued from a non-task context ena_dsp was issued in the CPU lock state

Sense Context

sns_ctx

[Overview]

References a context

[C format]

```
#include <kernel.h>
BOOL sns_ctx(void);
```

[Parameter]

None

[Explanation]

The context currently under execution is referenced and TRUE is returned if a non-task context such as an interrupt handler is executed; otherwise, FALSE is returned.

If this service call is issued from a CPU exception handler, the context used when a CPU exception has occurred is referenced, and TRUE or FALSE is returned.

The type of the return value of sns_ctx is BOOL. Usually, a value of 0 or 1 is returned. As an exception, E_RSFN (−10) is returned if sns_ctx is not included in the system.

[Differences from μITRON3.0]

sns_ctx is a newly created service call.

[Return values]

Symbol	Value	Meaning
TRUE	1	A non-task context is under execution
FALSE	0	A task context is under execution
E_RSFN	−10	sns_ctx is not included in the system

Sense CPU Locked or Not

sns_loc

[Overview]

Checks whether the CPU is locked

[C format]

```
#include <kernel.h>
BOOL sns_loc(void);
```

[Parameter]

None

[Explanation]

The current system status is checked to see whether the CPU is locked by (i)loc_cpu, and the value TRUE or FALSE is returned.

The type of the return value of sns_loc is BOOL. Usually, a value of 0 or 1 is returned. As an exception, E_RSFN (−10) is returned if sns_loc is not included in the system.

[Differences from μITRON3.0]

sns_loc is a newly created service call to simplify referencing the status with ref_sys to realize high-speed processing.

[Return values]

Symbol	Value	Meaning
TRUE	1	The CPU is locked
FALSE	0	The CPU is not locked
E_RSFN	−10	sns_loc is not included in the system

Sense Dispatch Enabled or Not

sns_dsp

[Overview]

Checks whether dispatch is disabled

[C format]

```
#include <kernel.h>
BOOL sns_dsp(void);
```

[Parameter]

None

[Explanation]

The current system status is checked to see whether dispatch is disabled by dis_dsp, and TRUE is returned if dispatch is disabled; otherwise, FALSE is returned.

Note that disabling dispatch by dis_dsp is distinguished from disabling dispatch by (i)loc_cpu. If sns_dsp is issued after (i)loc_cpu has been issued, TRUE is returned if dispatch is disabled after issuance of (i)unl_cpu; otherwise, FALSE will be returned.

The type of the return value of sns_dsp is BOOL. Usually, a value of 0 or 1 is returned. As an exception, E_RSFN (−10) is returned if sns_dsp is not included in the system.

[Differences from μITRON3.0]

sns_dsp is a newly created service call to simplify referencing the status with ref_sys to realize high-speed processing.

[Return values]

Symbol	Value	Meaning
TRUE	1	Dispatch is disabled
FALSE	0	Dispatch is not disabled
E_RSFN	−10	sns_dsp is not included in the system

Sense Dispatch Pending

sns_dpn

[Overview]

Checks whether dispatch is pending

[C format]

```
#include <kernel.h>
BOOL sns_dsp(void);
```

[Parameter]

None

[Explanation]

The current system status is checked to see whether dispatch is pending, and TRUE is returned if dispatch is pending; otherwise, FALSE is returned.

Pending dispatch includes when dispatch is disabled by loc_cpu while dispatch has been disabled by dis_dsp and when processing with a higher priority than the dispatcher is under execution. When sns_dpn returns FALSE, therefore, it means that a service call that may keep a task waiting can be issued.

The type of the return value of sns_dpn is BOOL. Usually, a value of 0 or 1 is returned. As an exception, E_RSFN (–10) is returned if sns_dpn is not included in the system.

[Differences from μITRON3.0]

sns_dpn is a newly created service call to simplify referencing the status with ref_sys to realize high-speed processing.

[Return values]

Symbol	Value	Meaning
TRUE	1	Dispatch is pending
FALSE	0	Dispatch is not pending
E_RSFN	–10	sns_dpn is not included in the system

13.8.8 Interrupt management function service calls

This section explains the interrupt management function service calls. Note that the descriptions in this section apply if the source file supplied as a sample is used without modification. The user can change the functions of the source file as necessary.

Table 13-15. Interrupt Management Function Service calls

Name	Function
cre_isr	Creates an ISR
acre_isr	Creates an ISR (automatic assignment of ID No.)
del_isr	Deletes an ISR
dis_int	Disables interrupts
ena_int	Enables interrupts
chg_ims/ichg_ims	Changes the interrupt mask
get_imsi/get_ims	References the interrupt mask

cre_isr

Create Interrupt Service Routine

[Overview]

Creates an interrupt service routine

[C format]

```
#include <kernel.h>
ER cre_isr(ID isrid, T_CISR *pk_cisr);
```

[Parameters]

I/O	Parameter		Description
I	ID	isrid	ID number of interrupt service routine to be created
I	T_CISR *	pk_cisr	Address of interrupt service routine creation packet

Configuration of T_CISR

```
typedef struct t_disr {
        ATR         isratr;     /* Interrupt service routine attribute */
        VP_INT      exinf;      /* Extended data */
        INTNO       intno;      /* Interrupt number */
        FP          isr;        /* Interrupt service routine address */
        VP          gp;         /* PID base address */
        VP          tp;         /* Reserved area */
} T_CISR;
```

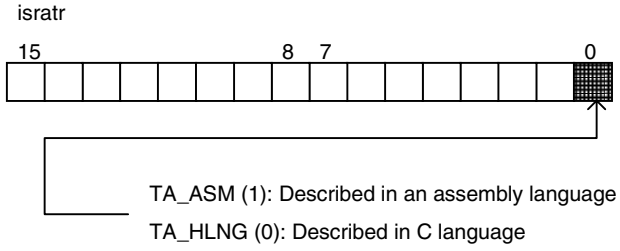
[Explanation]

An interrupt service routine with the ID number specified by isrid is created based on the information stored in the interrupt service routine creation packet pk_cisr. The interrupt service routine creation packet T_CISR is described in detail below.

- isratr

  Specifies the attribute of the interrupt service routine to be created. The following interrupt service routine attributes are available:

Figure 13-12. Interrupt Service Routine Attribute



Bit 0 specifies the language in which the interrupt service routine is described. TA_ASM means that the routine is described in an assembly language. TA_HLNG means the C language. With the current version, however, these two attributes are not differentiated in the kernel processing.

- exinf
  Stores user-defined information on the interrupt service routine to be created. This extended data is passed to the interrupt service routine as a parameter when the routine is activated.
- intno
  Specifies a number that uniformly identifies the interrupt corresponding to the interrupt service routine to be created. Interrupt numbers can be arbitrarily defined by the user. However, −1 must not be used as it is reserved for the system.
- isr
  Specifies the activation address of the interrupt service routine.
- gp
  Specifies the base address (gp) of PID (Position Independent Data) used by the interrupt service routine to be created. Set NULL if the routine does not use PID.
- tp
  This is a reserved area. Always set NULL to this area. However, a value other than NULL is ignored even if set.

[Differences from μITRON3.0]

It is defined that processing corresponding to an interrupt vector is performed by an “interrupt handler” and the processing corresponding to an interrupt source is performed by an “interrupt service routine”. Because the RX4000 has only one interrupt vector for the VR4100 Series and VR5000 Series, it does not supply a function to define interrupt handlers but supplies a function to create interrupt service routines.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	cre_isr is not included in the system
E_RSATR	−11	The interrupt service routine attribute is illegal
E_PAR	−17	The parameter is illegal – The interrupt number is illegal (intno > maximum interrupt number)
E_ID	−18	The interrupt service routine ID number is outside the range (isrid ≤ 0, maximum interrupt service routine count < isrid)
E_CTX	−25	cre_isr was issued from a non-task context cre_isr was issued in the CPU lock state
E_OBJ	−41	An interrupt service routine with the same ID number already exists

acre_isr

Create Interrupt Service Routine with Automatic ID Assignment

[Overview]

Creates an interrupt service routine (Automatic assignment of ID number)

[C format]

```
#include <kernel.h>
ER_ID acre_isr(T_CISR *pk_cisr);
```

[Parameter]

I/O	Parameter		Description
I	T_CISR *	pk_cisr	Address of interrupt service routine creation packet

[Explanation]

An interrupt service routine is created based on the information stored in the interrupt service routine creation packet pk_cisr, and the ID number of the interrupt service routine is returned. In other words, an interrupt service routine control block that is available but has not been created is searched, assigned, and initialized, and its ID number is returned. If the return value is negative, an error occurs. For the meaning of the return value, refer to **[Return values]** below.

For details of the interrupt service routine creation packet, refer to the description of cre_isr.

[Differences from μITRON3.0]

Refer to the description of cre_isr.

[Return values]

Symbol	Value	Meaning
(Integer of 1 or greater)		ID number of the created interrupt service routine (normal termination)
E_RSFN	−10	acre_isr is not included in the system
E_RSATR	−11	The interrupt service routine attribute is illegal
E_PAR	−17	The parameter is illegal − The interrupt number is illegal (intno > maximum interrupt number)
E_CTX	−25	acre_isr was issued from a non-task context acre_isr was issued in the CPU lock state
E_NOID	−34	The ID number cannot be assigned (reserving a control block has failed)

del_isr

Delete Interrupt Service Routine

[Overview]

Deletes an interrupt service routine

[C format]

```
#include <kernel.h>
ER del_isr(ID isrid);
```

[Parameter]

I/O	Parameter		Description
I	ID	isrid	ID number of interrupt service routine to be deleted

[Explanation]

The interrupt service routine specified by isrid is deleted. However, an interrupt service routine created by the static API ATT_ISR cannot be deleted.

[Differences from μITRON3.0]

This service call is newly created with the introduction of interrupt service routines. It is equivalent to def_int (*dintno*, NADR) in μITRON3.0.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	del_isr is not included in the system
E_ID	−18	The interrupt service routine ID number is outside the range (isrid ≤ 0, maximum interrupt service routine count < isrid)
E_CTX	−25	del_isr was issued from a non-task context del_isr was issued in the CPU lock state
E_NOEXS	−42	The target interrupt service routine does not exist

dis_int

Disable Interrupt

[Overview]

Disables interrupts.

[C format]

```
#include <kernel.h>
ER dis_int(INTNO intno);
```

[Parameter]

I/O	Parameter		Description
I	INTNO	intno	Interrupt number

[Explanation]

The interrupt controller corresponding to the interrupt number specified by intno is manipulated, and the interrupt of that number is disabled. In the sample source file supplied, interrupts to the interrupt controller provided in the VR Series evaluation boards, CMB-VR4131, Ostrich VR4181A, and RTE-VR5500-CB, are disabled.

The interrupt number INO_CPU = −1 is reserved. If INO_CPU is specified as intno, IM of the status register is changed and interrupts are disabled (masked).

[Differences from μITRON3.0]

RX4000 V3.x disables interrupts by manipulating the IE bit of the status register. In contrast, μITRON4.0 disables interrupts by manipulating a specified interrupt controller. In addition, INO_CPU can be specified.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	dis_int is not included in the system
E_PAR	−17	The interrupt number is illegal
E_CTX	−25	dis_int was issued from a non-task context dis_int was issued in the CPU lock state

ena_int

Disable Interrupt

[Overview]

Enables interrupts

[C format]

```
#include <kernel.h>
ER ena_int(INTNO intno);
```

[Parameter]

I/O	Parameter		Description
I	INTNO	intno	Interrupt number

[Explanation]

The interrupt controller corresponding to the interrupt number specified by intno is manipulated, and the interrupt of that number is enabled. In the sample source file supplied, interrupts to the interrupt controller provided in the VR Series evaluation boards, CMB-VR4131, Ostrich VR4181A, and RTE-VR5500-CB, are enabled.

The interrupt number INO_CPU = −1 is reserved. If INO_CPU is specified as intno, IM of the status register is changed and interrupts are enabled (unmasked).

[Differences from μITRON3.0]

RX4000 V3.x enables interrupts by manipulating the IE bit of the status register. In contrast, μITRON4.0 enables interrupts by manipulating a specified interrupt controller. In addition, INO_CPU can be specified.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	ena_int is not included in the system
E_PAR	−17	The interrupt number is illegal
E_CTX	−25	ena_int was issued in the CPU lock state

Change Interrupt Mask

chg_ims/ichg_ims

[Overview]

Changes the interrupt mask

[C format]

```
#include <kernel.h>
ER chg_ims(UINT intms);
```

[Parameter]

I/O	Parameter		Description
I	UINT	intms	Interrupt mask to be set

[Explanation]

The interrupt mask of the CPU (IM field of the status register) is changed to the interrupt mask specified by intms.

ichg_ims is intended to be issued from a non-task context but it can also be issued from a task context. chg_ims can be issued from a non-task context.

[Differences from μITRON3.0]

None

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	chg_ims/ichg_ims is not included in the system
E_PAR	−17	The interrupt mask is outside the range (0xff < intms)
E_CTX	−25	chg_ims/ichg_ims was issued in the CPU lock state

Get Interrupt Mask

get_ims/iget_ims

[Overview]

Obtains the interrupt mask

[C format]

```
#include <kernel.h>
ER get_ims(UINT *p_intms);
```

[Parameter]

I/O	Parameter		Description
I	UINT *	p_intms	Address to store interrupt mask

[Explanation]

The interrupt mask of the CPU (IM field of the status register) is obtained and stored in the address specified by p_intms.

iget_ims is intended to be issued from a non-task context but it can also be issued from a task context. get_ims can be issued from a non-task context.

[Differences from μITRON3.0]

The name of ref_ims has been changed to get_ims.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	get_ims/iget_ims is not included in the system
E_CTX	−25	get_ims/iget_ims was issued in the CPU lock state

13.8.9 System configuration management function service calls

This section explains the system configuration management function service calls listed in the following table.

Table 13-16. System Configuration Management Function Service calls

Name	Function
def_exc	Defines a CPU exception handler
ivsta_exc	Activates a CPU exception handler

def_exc

Define Exception Handler

[Overview]

Defines a CPU exception handler

[C format]

```
#include <kernel.h>
ER def_exc(EXCNO excno, T_DEXC *pk_dexc);
```

[Parameters]

I/O	Parameter		Description
I	EXCNO	excno	CPU exception handler number
I	T_DEXC *	pk_dexc	Address of CPU exception handler definition packet

Configuration of T_DEXC

```
typedef struct t_dexc {
        ATR     excatr;     /* CPU exception handler attribute */
        FP      exchdr;     /* Activation address of CPU exception handler */
        VP      gp;         /* PID base address */
        VP      tp;         /* Reserved area */
} T_DEXC;
```

[Explanation]

A CPU exception handler with the exception handler number specified by excno is defined based on the CPU exception handler definition packet specified by pk_dexc. The RX4000 supplies a sample that allows the user to describe how to identify the exception source and activate a handler if a CPU exception occurs, and therefore the user must make sure that the exception source corresponds to the CPU exception handler definition number. In addition, the supplied sample assumes that the value of the ExcCode field of the cause register is used as the exception handler number.

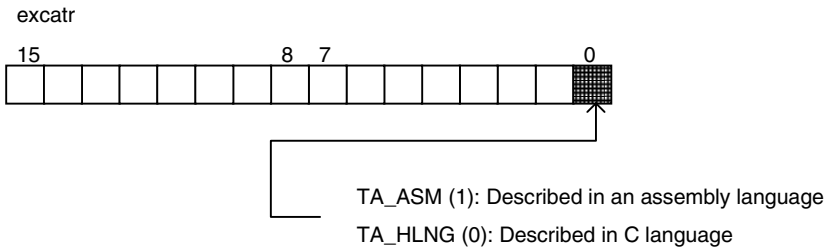
If def_exc is issued with a definition number for which a CPU exception handler has been already defined, the previous definition is overwritten and new definition is made. To clear the definition of a CPU exception handler, specify NULL as pk_dexc.

The CPU exception handler definition packet is described in detail below.

- excatr

  Specifies the language in which the CPU exception handler to be defined is described. TA_ASM means that the handler is described in an assembly language. TA_HLNG means the C language. With the current version, however, these two attributes are not differentiated in the kernel processing.

Figure 13-13. Exception Handler Attribute



- exchdr
  Specifies the activation address of the CPU exception handler to be defined.
- gp
  Specifies the base address of PID used by the CPU exception handler to be defined. Set NULL if the handler does not use PID.
- tp
  This is a reserved area. Always set NULL to this area. However, a value other than NULL is ignored even if set.

[Differences from μITRON3.0]

The RX4000 V3.x allows only one CPU exception handler to exist in the system. The RX4000 (μITRON4.0) allows two or more CPU exception handlers to be defined. However, the user must describe what processing the hander should perform depending on the source of the exception, and how the handler is activated.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	def_exc is not included in the system
E_RSATR	−11	The CPU exception handler attribute is illegal
E_PAR	−17	The CPU exception handler definition number is outside the range (excno < 0, maximum CPU exception handler count ≤ dexcno)
E_CTX	−25	def_exc was issued from a non-task context def_exc was issued in the CPU lock state

vatt_idl

Attach Idle Routine

[Overview]

Defines an idle routine

[C format]

```
#include <kernel.h>
ER vatt_idl(T_DIDL *pk_didl);
```

[Parameter]

I/O	Parameter		Description
I	T_DIDL *	pk_didl	Address of idle routine definition packet

Configuration of T_DIDL

```
typedef struct t_didl {
        ATR         idlatr;     /* Idle routine attribute */
        FP          idlrtn;     /* Idle routine activation address */
        VP          gp;         /* PID base address */
        VP          tp;         /* Reserved area */
} T_DIDL;
```

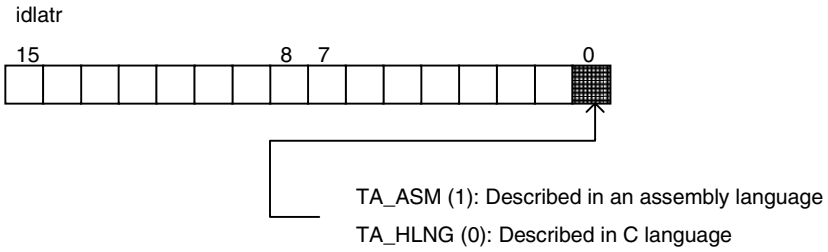
[Explanation]

An idle routine is defined, based on the information stored in the idle routine definition packet pk_didl. Only one idle routine can exist in the system, and one idle routine must be defined. Therefore, the definition of the idle routine is overwritten each time vatt_idl has been issued.

The idle routine definition packet is described in detail below.

- idlatr
  Specifies the language in which the idle routine to be defined is described. TA_ASM means that the routine is described in an assembly language. TA_HLNG means the C language. With the current version, however, these two attributes are not differentiated in the kernel processing.

Figure 13-14. Exception Handler Attribute



- idlrtn
  Specifies the activation address of the idle routine to be defined.
- gp
  Specifies the base address of PID used by the idle routine to be defined. Set NULL if the routine does not use PID.
- tp
  This is a reserved area. Always set NULL to this area. However, a value other than NULL is ignored even if set.

[Differences from μITRON3.0]

The RX4000 V3.x calls the idle routine an idle handler. Definition of the idle routine by a service call is not supported. By adding the new service call vatt_idl, the idle routine can be now defined by a service call. vatt_idl is the original extended function of the RX4000 and is not defined in the μITRON4.0 specification.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	vatt_idl is not included in the system
E_RSATR	−11	The idle routine attribute is illegal
E_CTX	−25	vatt_idl was issued from a non-task context vatt_idl was issued in the CPU lock state

13.8.10 Service call management function service calls

This section explains the service call management function service calls listed in the following table.

Table 13-17. Service call Management Function Service calls

Name	Function
def_svc	Defines an extended service call routine
cal_svc/ical_svc	Activates an extended service call routine

def_svc

Define Extended Service Call Routine

[Overview]

Defines an extended service call routine

[C format]

```
#include <kernel.h>
ER def_svc(FN fncd, T_DSVC *pk_dsvc);
```

[Parameters]

I/O	Parameter		Description
I	FN	fncd	Function code
I	T_DSVC *	pk_dsvc	Address of extended service call routine definition packet

Configuration of T_DSVC

```
typedef struct t_dsvc {
        ATR         svcatr;     /* Extended service call routine attribute */
        FP          svcrtn;     /* Activation address of extended service call routine */
        VP          gp;         /* PID base address */
        VP          tp;         /* Reserved area */
} T_DEXC;
```

[Explanation]

An extended service call routine with the function code specified by fncd is defined based on the extended service call routine definition packet specified by pk_dsvc.

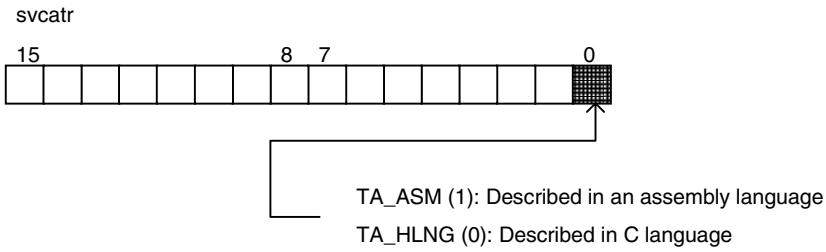
Therefore, def_svc can register a user-described function to the kernel as a service call. If def_svc is issued by using a function code for which an extended service call routine has been already defined, the previous definition is overwritten by new definition. To clear the definition of the extended service call routine, set NULL to pk_dsvc.

The extended service call routine definition packet is described in detail below.

- svcatr

  Specifies the language in which the extended service call routine to be defined is described. TA_ASM means that the routine is described in an assembly language. TA_HLNG means the C language. With the current version, however, these two attributes are not differentiated in the kernel processing.

Figure 13-15. Extended Service call Routine Attribute



- svcrtn

  Specifies the activation address of the extended service call routine to be defined.
- gp

  Specifies the base address of PID used by the extended service call routine to be defined. Set NULL if the routine does not use PID.
- tp

  This is a reserved area. Always set NULL to this area. However, a value other than NULL is ignored even if set.

[Differences from μITRON3.0]

The extended SVC handler is now called an extended service call routine.

[Return values]

Symbol	Value	Meaning
E_OK	0	Normal termination
E_RSFN	−10	def_svc is not included in the system
E_PAR	−10	The function code is outside the range (fncd ≤ 0, maximum extended service call routine count < fncd)
E_RSATR	−11	The extended service call routine attribute is illegal
E_CTX	−25	def_svc was issued from a non-task context def_svc was issued in the CPU lock state

Call Extended Service Call Routine

cal_svc/ical_svc

[Overview]

Calls a service call

[C format]

```
#include <kernel.h>
ER cal_svc(FN fncd, VW arg1, VW arg2, VW arg3);
```

[Parameters]

I/O	Parameter		Description
I	FN	fncd	Function code
I	VW	arg1	First argument of service call
I	VW	arg2	Second argument of service call
I	VW	arg3	Third argument of service call

[Explanation]

cal_svc is a general-purpose interface library that calls service calls including extended service call routines, and calls the service call with the function code fncd. If the function code of a standard service call is specified as fncd, E_RSFN error is returned.

Three parameters, arg1 to arg3, can be passed to the service call routine (main processing block). To issue this service call from a non-task context such as an interrupt service routine, use ical_svc.

ical_svc is intended to be issued from a non-task context but it can also be issued from a task context. cal_svc can be issued from a non-task context.

[Differences from μITRON3.0]

cal_svc/ical_svc is a newly created service call.

[Return values]

Symbol	Value	Meaning
E_RSFN	−10	An unused function code is specified
Others		Return value of the service call

APPENDIX INDEX

[S]

[T]